SPPU New Syllabus

A Book Of

DATA STRUCTURES

For B.C.A.(Science) : Semester - III

[Course Code 231 : Credit - 4]

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 and Associate Professor,
Computer Science Department,
MES Abasaheb Garware College,
Pune 4

Mrs. Madhuri Khadatare
M. Sc. (Computer Science) , NET
Assistant Professor,
Computer Science Department,
MES Abasaheb Garware College,
Pune 4

Price ₹ 260/-

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

N5411

Syllabus ...

1. **Introduction to data structure** **[06 Hrs.]**
   - Data types and data objects
   - Abstract Data Types (ADT)
   - Data structure
   - Algorithm analysis: Frequency counts, Space and Time complexity, Asymptotic notation : BigO, Omega (Ω)
   - Algorithms and its complexity using simple example algorithms
2. **Arrays** **[06 Hrs.]**
   - Introduction and definition
   - Matrix representation using arrays: Row and column major, operations on matrices, Sparse Matrix
   - Sorting techniques with time complexity: Bubble sort, Insertion sort, Merge sort, Quick sort
   - Searching techniques with time Complexity: Linear search, Binary search
3. **Linked Lists** **[12 Hrs.]**
   - Introduction and Definition
   - Representation: Static &Dynamic
   - Types of linked lists: singly, doubly, circular
   - Operations on link list: create, display, insert, delete, reverse, search, sort, concatenation, Merge
   - Real world applications of Link list (implementation not expected)
4. **Stacks and Queues** **[12 Hrs.]**
   - Representation of Stack: Using arrays and Linked Lists
   - Operations on stack: push, pop
   - Applications of Stack : Recursion, Infix to postfix, postfix to infix
   - Representation of Queues : Static & Dynamic
   - Operations on queue: insert, delete
   - Types of queue: Circular queue and Priority queue
   - Real world Applications of queue (Implementation not expected)
5. **Trees** **[12 Hrs.]**
   - Introduction and Tree terminologies: Definitions: Tree, root, child, leaf, level, height, depth
   - Binary trees: Types: Rooted, full, complete and skewed.
   - Representation of Trees: Using arrays and Linked Lists
   - Types of Traversal: Preorder, Inorder, Postorder, Applications of Binary trees
   - Binary Search Tree (BST): Introduction and definition
6. **Graphs** **[12 Hrs.]**
   - Introduction and Graph terminologies
   - Representation of a Graph – Adjacency matrix, Adjacency list, Adjacency multi-list
   - Graph Traversals – DFS, BFS
   - Applications of graphs - Topological sort

■■■

Contents ...

1. Introduction to data structure 1.1 – 1.22

2. Arrays 2.1 – 2.36

3. Linked Lists 3.1 – 3.60

4. Stacks and Queues 4.1 – 4.60

5. Trees 5.1 – 5.42

6. Graphs 6.1 – 6.34

■■■

1... Introduction to Data Structure

Objectives...

- To know the basic concepts of Data Structures.
- To study of data types and use of Data Types.
- To study different types of Data Structures.
- To learn about the need of Data Structure.
- To know the power of ADT.

1.1 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 you 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, data structure is represented as:

Algorithm + Data structure = Program

- In this chapter we are going to study the fundamental concepts related to data structure. So it is very essential to see that what are the different types of data? How are they organized?

1.2 DATA TYPE AND DATA OBJECT

1.2.1 Data

- Data refers to value or set of values. Data is divided into atomic or non-atomic (composite) data. Example: Marks obtained by the student:

  (a) Atomic data: It is a single or non-decomposable entity. For example, an integer 3241 is considered 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.

(b) **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.

1.2.2 Data Types

- Data type is term used to describe the type of data that variable may hold in the programming language.
- The Fig. 1.1 shows the data types in 'C' language.

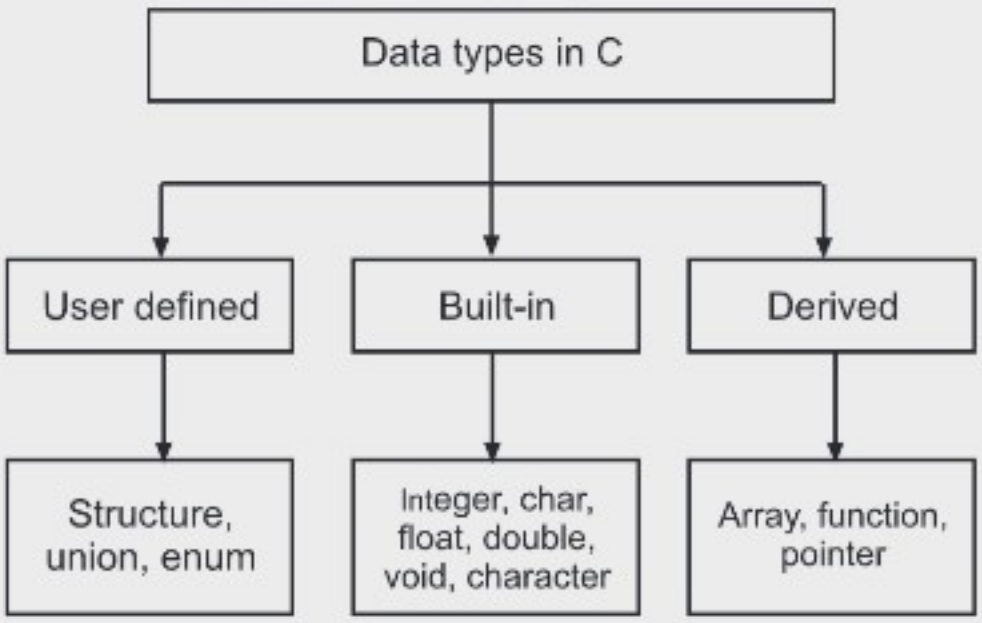


Fig. 1.1: Data types in 'C'

1.2.2.1 Built-In or Primitive Data Types

- Following table shows basic data types in C and their storage size and range.

Table 1.1: Size and Range of basic data types

Data Type	Storage size	Range
Char	1 byte	–128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	–128 to 127
int	2 or 4 bytes	–32,768 to 32,767 or –2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short int	2 bytes	–32,768 to 32,767
unsigned short int	2 bytes	0 to 65,535
long int	8 bytes	–9223372036854775808 to 9223372036854775807
float	4 bytes	1.2E-38 to 3.4E+38
Double	8 bytes	2.3E-308 to 1.7E+308
long double	10 bytes	3.4E-4932 to 1.1E+4932

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.

Void Data Type

- Void is an empty data type that has no value. This can be used in functions and pointers. It is used as the types returned by functions that generate no value.

1.2.2.2 User-Defined Data Types

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 is 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 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;
}
```

- In structure, the memory size is 12+4=16, but in union the memory storage is allocated to largest one that is 12 here.

1.2.2.3 Derived Data Types [S-18]

- 'C' provides data types like arrays, functions, pointers, structures and unions:

1. **Arrays:** Assign a single name to whole group of elements (homogeneous).

   For example: int A[10];

   defined an array of A with 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.
     ```

1.2.3 Data Object [S-19, 18]

- A data object represents a container for data values (a place where data values may be 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 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, ...}
  ```

- 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. 1.2: Data object

A data object x containing the data value 9.

- 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".

1.3 ABSTRACT DATA TYPES (ADT) [S-19]

- 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).
- 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 following 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 doesn't 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.
  3. take things out.

  Such a specification is an Abstract Data Type.

Definition:

- An Abstract Data Type (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.

1.3.1 Example of ADT in 'C'

- 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.

1.3.2 Advantages of ADT

- Advantages of ADT are listed below:
  1. Encapsulation
  2. Information hiding
  3. Implementation details hiding
  4. The ADT is a useful guideline to implementers and a useful tool to programmers who wish to use the data type correctly.
  5. ADT is reusable and ensures robust data structure.
  6. Encapsulation ensures that data cannot be corrupted.

1.4 DATA STRUCTURE [W-18]

- 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.

Definition:

- A data structure is 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 written as d, where
  - **Domain (D):** Denotes the data objects.
  - **Function (F):** Denotes the set of operations that can be carried out on the data objects.
  - **Axioms (A):** Denotes the properties and rules of the operations.
- This 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.

1.4.1 Operations on Data Structures [W-18]

- The basic operations that are performed on data structures are as follows:
  - **Insertion**: Insertion is used to adding a new data element in a data structure.
  - **Deletion**: Deletion is used to remove a data element from a data structure if it is found.

- **Searching**: Searching is used to find out the location of the data element in a data structure.
- **Traversal**: Traversal of a data structure means processing all the data elements present in it.
- **Sorting**: It is used to arranging data elements of a data structure in a specified order is called sorting.
- **Merging**: Combining elements of two similar data structures to form a new data structure of the same type, is called merging.

1.4.2 NEED OF DATA STRUCTURE

- Data structure is a fundamentals in computer science. Computer deals with manipulation of various kinds of data, wherefrom the concept of data structure comes.
  1. Data structures are used to study, how data are stored in a computer so that operations can be implemented efficiently
  2. Data structures are especially important when you have a large amount of information.
  3. Conceptual and concrete way 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.
- **Example:** Let us consider two ways of the data (phone numbers and names) organization 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 sequentially 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.

1.4.3 ADVANTAGES OF DATA STRUCTURES

- Different advantages of data structures are listed below:
  1. Allows easier processing of data.
  2. It allows information stored on disk very efficiently.
  3. These are necessary for designing an efficient algorithm.
  4. We can access data anytime and anywhere.
  5. It is secure way of storage of data.
  6. Graphs models real life problems.
  7. It allows processing of data on software system.

1.5 TYPES OF DATA STRUCTURES

- Data structures are divided into two types:
  - Linear data structures.
  - Non-Linear data structures.

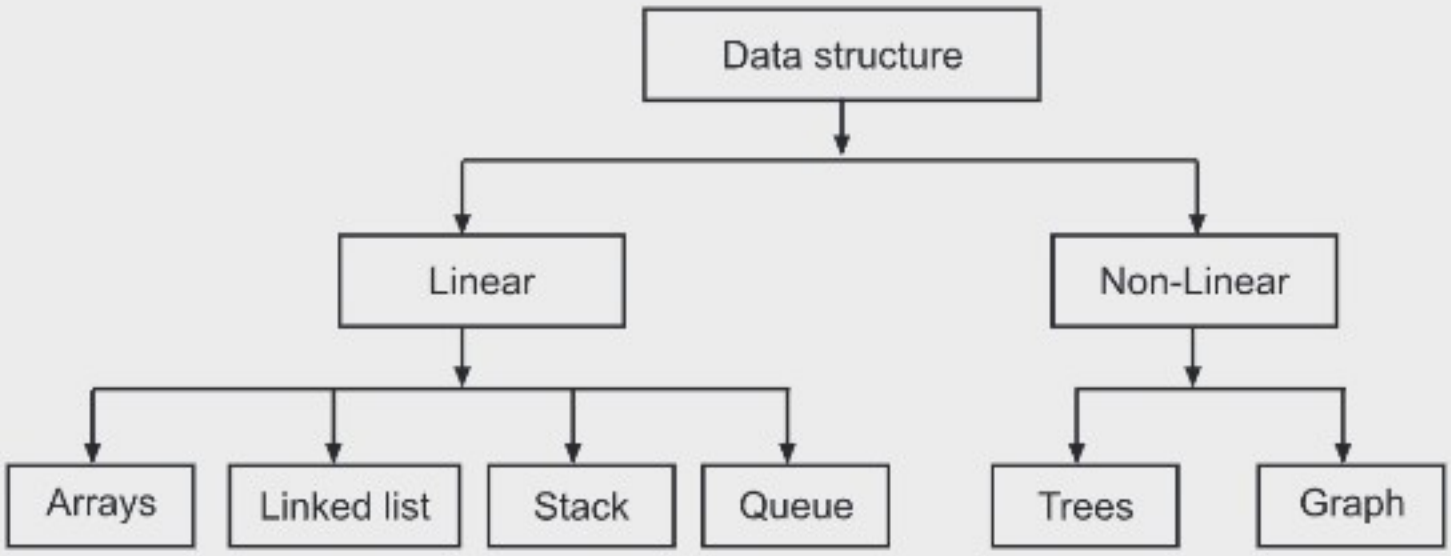


Fig. 1.3: Types of data structure

1.5.1 Linear Data Structures

- A data structure is said to be linear if its elements form a sequence or a linear list.
- In this type, 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 structure includes array, linked list, stack and queues.

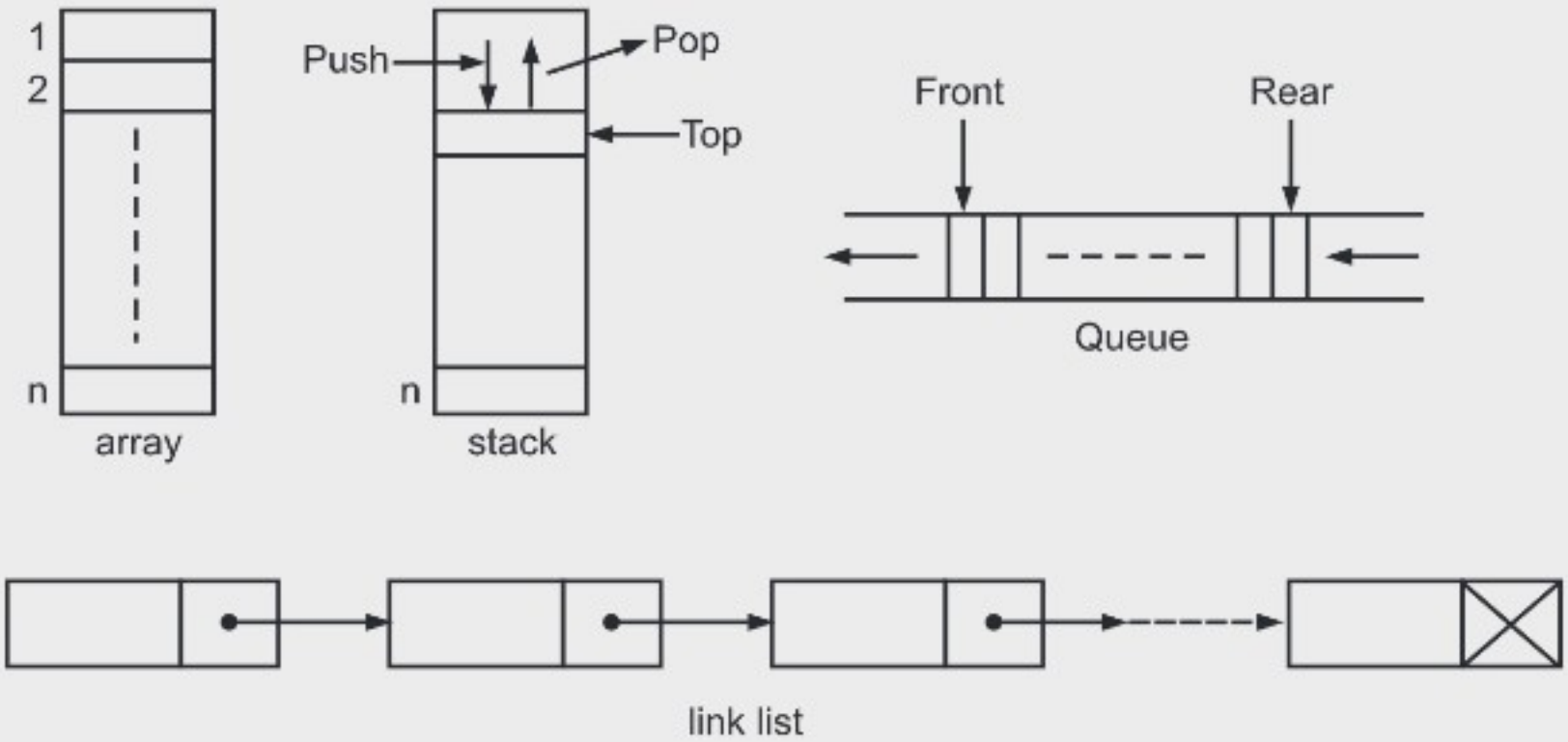


Fig. 1.4 (a): Representation of linear data structure

1.5.2 Non-Linear Data Structures

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

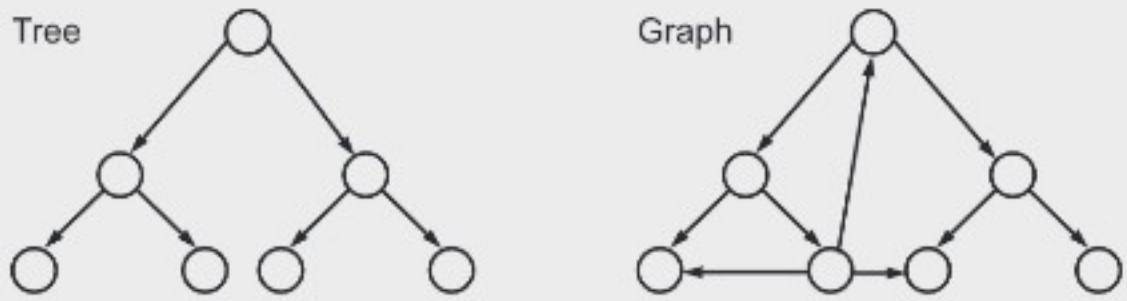


Fig. 1.4 (b): Representation of non-linear data structure

- The following table shows the difference between linear and nonlinear data structures.

Table 1.2: Difference between Linear and Non-linear data structures

Sr. No	Linear data structures	Non-linear data structures
1.	In a linear data structure, data elements are arranged in a linear order.	In a non-linear data structure, data elements are attached in hierarchically manner.
2.	In linear data structure, single level is involved.	Whereas in non-linear data structure, multiple levels are involved.
3.	Data items can be traversed in a single run.	Data items can not be traversed in a single run.
4.	These are easy to implement in the computer's memory.	These are difficult to implement in the computer's memory as compared to implementing linear data structures.
5.	Memory is not utilized in an efficient way.	Memory is utilized in an efficient way.
6.	Linear data structures are organized in a way similar to how the computer memory is organized.	Non-linear data structures are organized in a different way than the computer's memory.
7.	Examples: Array, Linked List, Stack and Queue	Examples: Tree, Graph

1.6 ALGORITHM ANALYSIS

Definition: [S-18]

- 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.

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.

Characteristics of Algorithm: [S-19, 18]

- 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 atleast 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, than it can be done exactly and in finite time by a person using only paper and pencil.

Example: To find the largest number from 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" goto step 5

Step 4 : If (y > z)
then "y is largest number"
else "z is largest number"

Step 5 : Stop

- We can find that this algorithm satisfies all the above 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.

1.6.1 Performance Analysis [S-19]

- 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).

1.6.1.1 Space Complexity

Definition: [W-18]

- "Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm".
- When we design an algorithm to solve a problem, it needs some computer memory to complete its execution.
- 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
  4. And for few other things
- Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows:
  - **Instruction Space**: It is the amount of memory used to store compiled version of instructions.
  - **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
  - **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:

```
int square(int x)
{
return x*x;
}
```

- 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).
- 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.

Example:

```
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 'n*2' bytes of memory to store array variable 'a[]', 2 bytes of memory for integer parameter 'n', 4 bytes of memory for local integer variables 'sum' and 'i'. (2 bytes each) 2 bytes of memory for return value. (Assume int takes 2 bytes).
- 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.

1.6.1.2 Time Complexity

Frequency count:

- Frequency count tells how many times the statement is executed in the program. So it is mean that the frequency count defines your program time complexity.

Definition:

- "The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution."
- Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity.
- 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:
  (i) Time required for executing it once.
  (ii) Number of times the statement is executed.
- Product of above two gives time required for that particular statement.
- The total number of times a statements get 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.

  Example:

```
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.

  Example:

```
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	1+(n+1)+n	2n+2
sum = sum + A[i];	2	n	2n
return sum;	1	1	1
}			
			4n+4

- 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+4' 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.

1.6.2 Asymptotic Notation

- 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.
- **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..

  Algorithm 1 : $5n^2 + 2n + 1$

  Algorithm 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.

Types of Asymptotic Notations:

1. Big-Oh(O)
2. Big-Omega(Ω)

1.6.2.1 Big - Oh Notation (O) [S-19, W-18]

- 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.

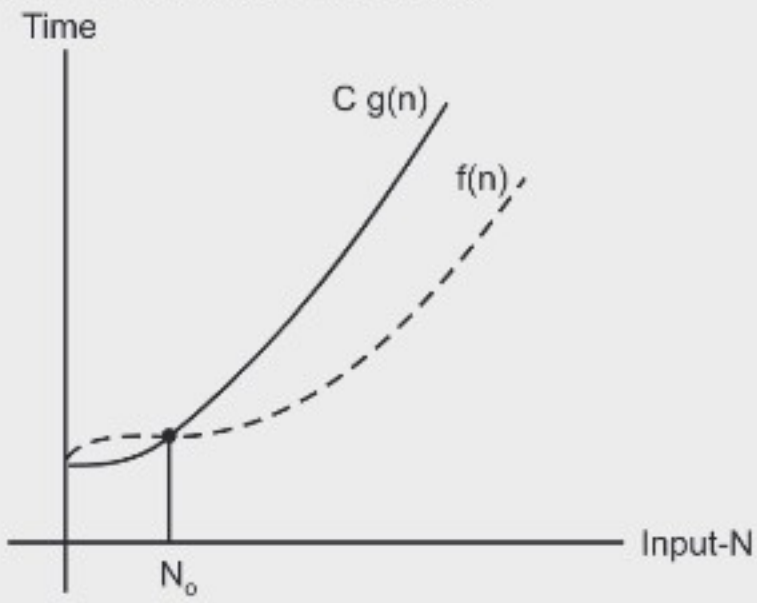


Fig. 1.5

- 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:

  $$3n + 2 = O(n)$$

1.6.2.2 Big-Omega Notation( )

- Big-Omega notation is used to define the lower bound of an algorithm in terms of Time Complexity.
- 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.

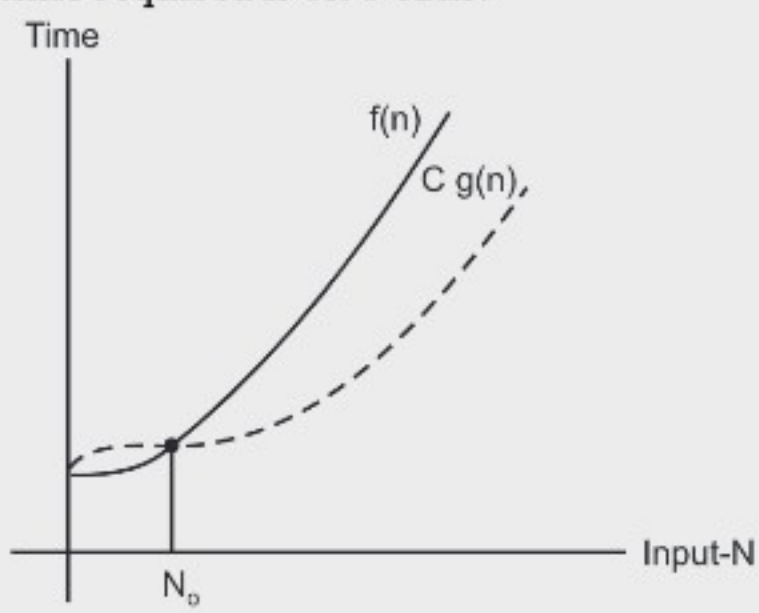


Fig. 1.6

- 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:

  $$3n + 2 = \Omega(n)$$

1.7 Simple Algorithms and its Complexity as Example

Example 1: Find space requirement of following statements:

(a) double x[3]

Ans.: 24 bytes

(b) int max[10] [10] (Assume int takes 4 bytes)

Ans.: 400 bytes

(d) a[2] [3] [4]

Ans.: 96 bytes

Example 2: Find time requirement of following statements:

(a)
```
for(i = 0, i < n; i++)
    printf("%d \n", i);
```

Ans.: 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;
```

Ans.: Here, statement x = x * y executes n * n times. Hence, frequency count is n^2.

(c)
```
int a = 0;
for (i = 0; i < N; i++)
  for (j = N; j > i; j--)
    a = a + i + j;
```

Ans.: The above code runs total no of times

= $n + (n - 1) + (n - 2) + \ldots 1 + 0$

= $n * (n + 1) / 2$

= $1/2 * n^2 + 1/2 * n$

$O(n^2)$ times.

Example 3: 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(n)

Example 4: 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 = $O(n^3)$

Example 5: 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)

Example 6: 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)**

Example 7: What is time complexity? Compute time complexity for the following algorithm:

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

Solution:

	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**

Summary

- A data structure is a way of organizing data that considers not only the items stored, but also their relationship to each other. Advance knowledge about the relationship between data items allows designing of efficient algorithms for the manipulation of data.
- Different areas in which data structures are applied extensively are Compiler Design, Operating System, Database Management System, Statistical analysis package, Numerical Analysis, Graphics, Artificial Intelligence, and Simulation etc.
- ADT is data declaration encapsulated with operations on it.
- Data structures are classified into linear and non-linear types.
- Need of data structure is essential for retrieving the information, manipulating the data and storage utilization.
- An algorithm is a set of instructions to accomplish a particular task.
- Input, Output, Definiteness, Finiteness, Effectiveness are the characteristics of an algorithm.
- Time complexity and Space complexity are performance measures of a program.
- Analysis of an algorithm is done on the following basis:
  - Best case time complexity
  - Worst case time complexity
  - Average case time complexity
- Time complexity is the run-time required for program.
- Space complexity is the memory or storage required for the program.
- Big O, Ω are the notations used in performance analysis.

Check Your Understanding

1. A mathematical model with a collection of operation defined on that model 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
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
   (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
    (d) 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

Answers

1. (b)	2. (d)	3. (c)	4. (d)	5. (c)	6. (c)	7. (a)
8. (c)	9. (c)	10. (b)	11. (a)	12. (b)	13. (b)	

Practice Questions

Q.I Answer the following questions in short:

1. What do you mean by Data Structure?
2. List out the areas in which data structures are applied extensively?
3. What do you mean by asymptotic notations?
4. What are the different characteristics of algorithm?
5. What are the different types of data structure?

Q.II Answer the following questions:

1. Why we need Data Structure?
2. Explain the term Data Object.
3. What is ADT? Enlist its different advantages.
4. What are the advantages of Data structure?
5. Compare linear and Non-linear data structure.
6. Write short note on types of Data structure.
7. With suitable example describe ADT.
8. Write note on Space Complexity.
9. Explain Time Complexity with an example.
10. Write a note on Big oh and Big Omega notation

Q.III Define the following terms:

1. Data type, 2. Data object, 3. Time Complexity, 4. Space Complexity, 5. Big-Oh notation, 6. Omega notation, 7. Algorithm

Previous Exam Questions

Summer 2019

1. _____ is an amount of time taken by a program for execution is the running time of program. **[1 M]**
(i) Time complexity (ii) Space complexity
(iii) Memory complexity (iv) Both (i) and (ii)

Ans. (i)

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

Ans. Refer to Section 1.2.3.

3. List different characteristics of an algorithm. **[1 M]**

Ans. Refer to Section 1.6.

4. Explain complexity in detail. **[4 M]**

Ans. Refer to Section 1.6.1.

5. Define the following terms: **[2 M]**

(i) Abstract Data Type(ADT)

Ans. Refer to Section 1.3.

(ii) Big O notation.

Ans. Refer to Section 1.6.2.1.

Winter 2018

1. A mathematical model with a collection of operations defined on that model is called: **[1 M]**

(i) Data structure (ii) Abstract data type
(iii) Primitive data type (iv) Algorithm

Ans. (ii)

2. What is data structure? Explain the different operations performed on data structure. **[5 M]**

Ans. Refer to Section 1.4.

3. Explain Big-Oh notation with the help of an example. **[3 M]**

Ans. Refer to Section 1.6.2.1.

4. What is time complexity? Compute time complexity for the following algorithm: **[3 M]**

```
If(x>y)
{
x=x+1;
}
Else
{
For(I=1;I<=N;I++)
{
    X=x+1;
}
}
```

Ans. Refer to Example 7.

5. What is space complexity? **[1 M]**

Ans. Refer to Section 1.6.1.1.

Summer 2018

1. The complexity of linear search is _____. **[1 M]**

(i) O (log(n)) (ii) O (n)
(iii) O (n log n) (iv) none of these

Ans. (ii)

2. Define: Data object. **[1 M]**

Ans. Refer to Section 1.2.3.

3. Explain the derived data types. **[4 M]**

Ans. Refer to Section 1.2.2.3.

4. Define algorithm. Give its characteristics. **[3 M]**

Ans. Refer to Section 1.6.

■■■

2... Arrays

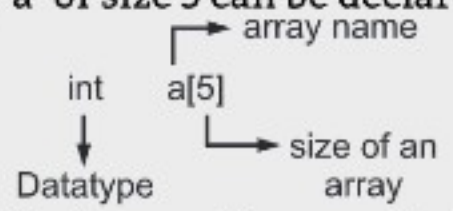
Objectives...

- To learn representation of an Array as ADT.
- To study different types of Array.
- To study different applications of Array.
- To study memory representation of One-dimensional and Multidimensional Array.
- To study basic concepts of searching and sorting.
- To learn different Sorting methods and compare with their time complexity.

2.1 INTRODUCTION TO ARRAY

2.1.1 Definition

- 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 name, data type and size.
- **Example**: In 'C' integer array 'a' of size 5 can be declared as:



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

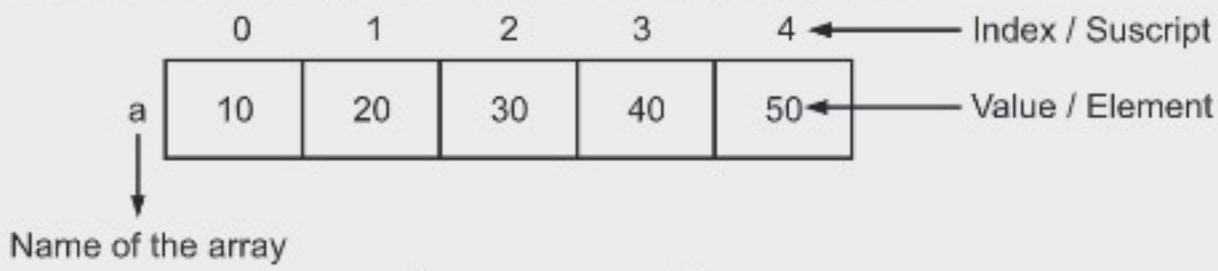


Fig. 2.1: Array of integers

- It allows storing 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 an index or subscript. An index or subscript of an array can be 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**.

2.1.2 Applications of Array

- Arrays are used to Store List of values.
- Arrays are used to Perform Matrix Operations.
- Arrays are used to implement Search Algorithms.
- Arrays are used to implement Sorting Algorithms.
- Arrays are used to implement Data structures.
- Arrays are also used to implement CPU Scheduling Algorithms.

2.1.3 Array Initialization and Representation

- Arrays can be initialized at the time of declaration.

  Example:

  1. int x[5] = {15, 25, 30, 35, 40}
  2. float B[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:

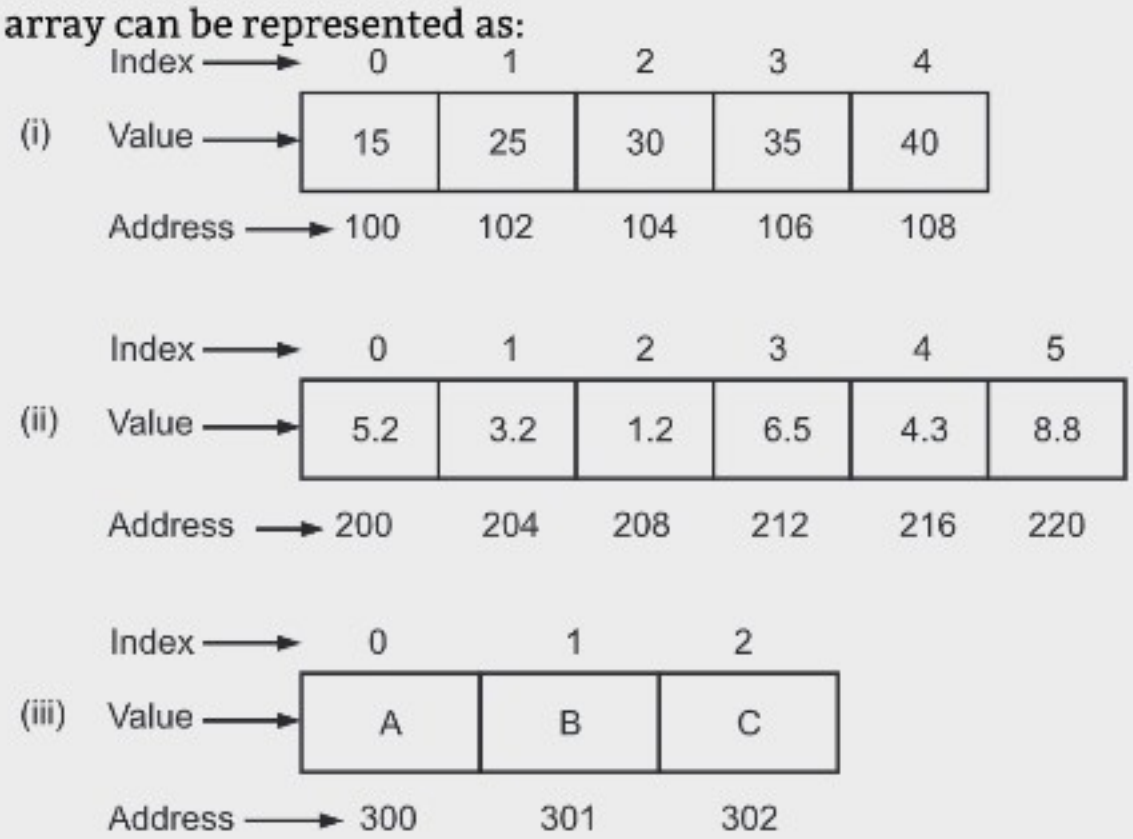


Fig. 2.2: Representation of array

- 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 (I)).
  - Float → 4 bytes (see address field of Fig. 2.2 (II)).
  - Char → 1 byte (see address field of Fig. 2.2 (III)).

2.2 TYPES OF ARRAY AND MATRIX REPRESETATION USING ARRAY

- Arrays are classified into two types. They are as follows...
  1. Single Dimensional Array / One Dimensional Array
  2. Multi Dimensional Array

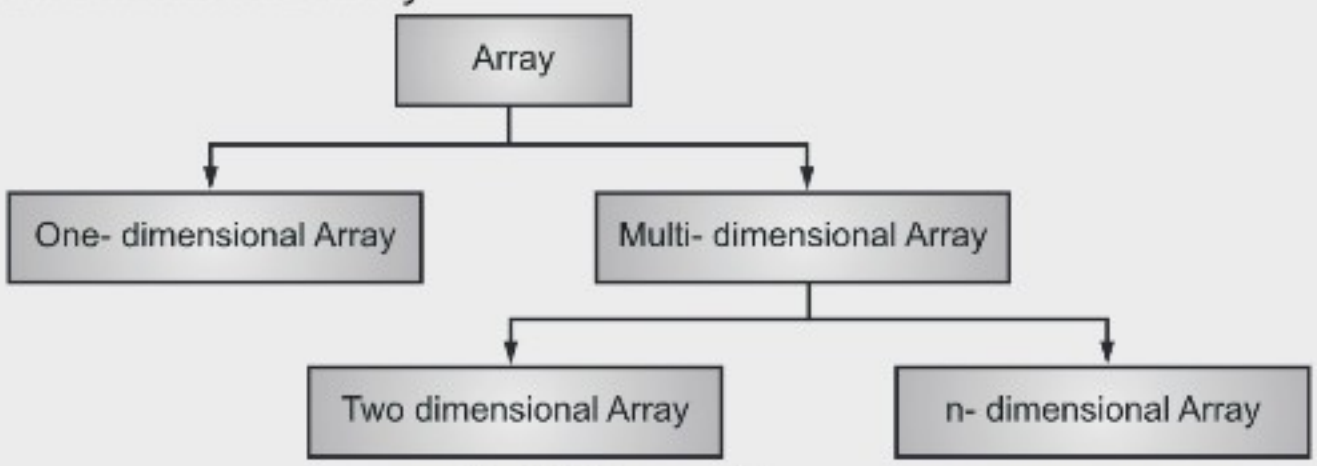


Fig. 2.3: Types of Array

2.2.1 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.
- 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.

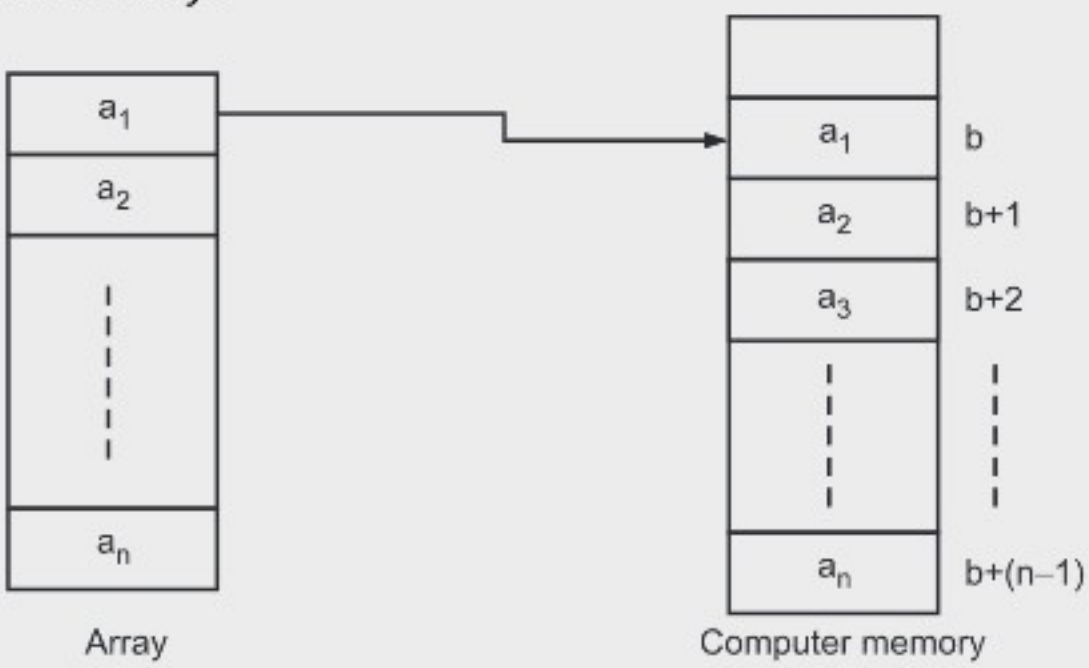


Fig. 2.4: Memory representation of array N

- 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 ith element from base address)

  Where offset is computed as:

  Offset of ith 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 + (2 × 2)
= 104

size
Offset

a [4] = 100 + (4 * size)
(i.e. 5th element) = 100 + (4 × 2)
= 108

2.2.2 Two Dimensional Array

- A two dimensional array is a collection of elements placed in m rows and n columns. The Fig. 2.5 shows the representation of 2-D array memory. The array is A[m][n] containing m rows and n columns.

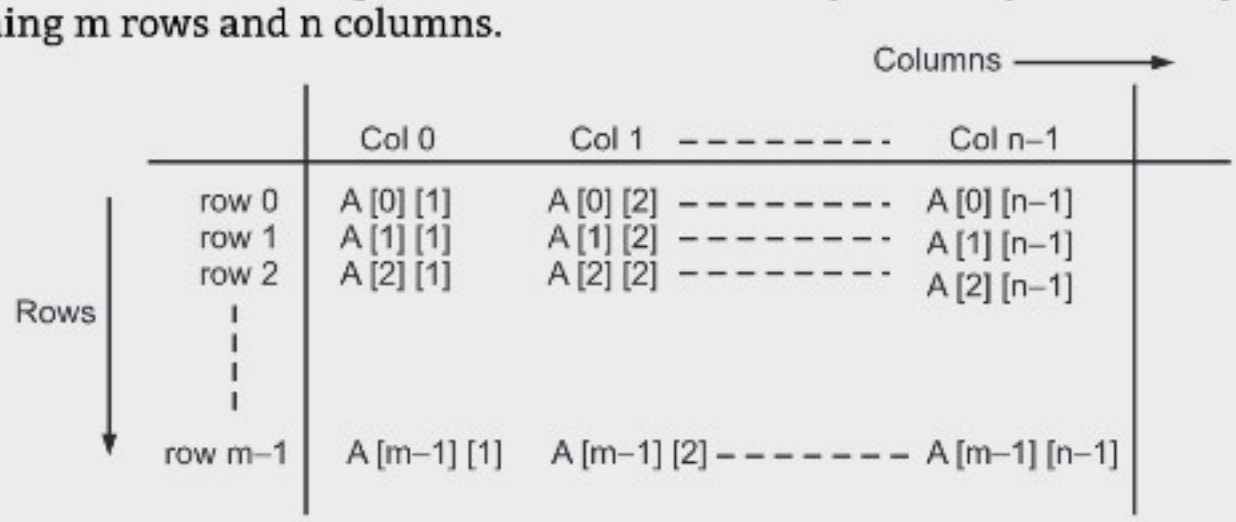


Fig. 2.5: Logical View of Array A[m] [n]

A 2-D array is a logical data structure useful in programming.

A 2-D array differentiates between the logical and physical view of data.

Memory Representation of 2D:

- 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: **[S-18]**

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

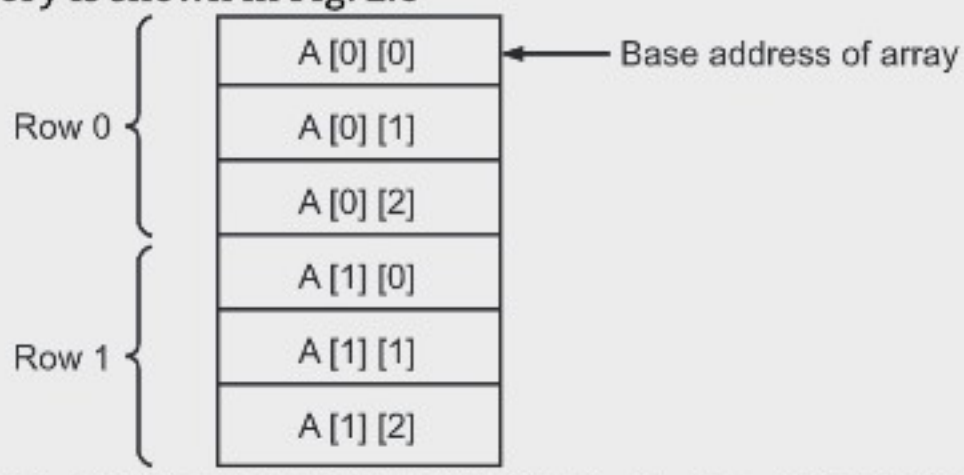


Fig. 2.6: Representation of 2-D array in row-major order

Let us calculate the address of element in 2-D 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 j^{th} 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}
}
```

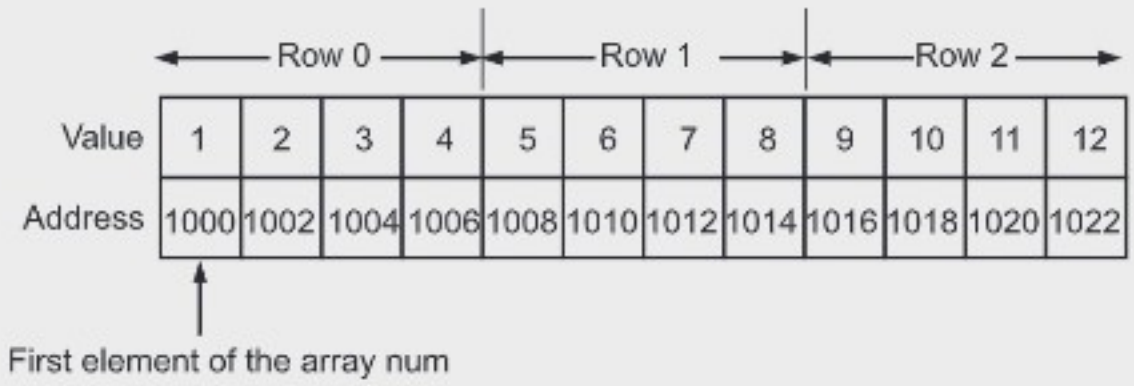


Fig. 2.7: Row-major representation of 2-D array in memory

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

$$\begin{aligned} num[2][3] &= baseAddress + [(i \times no_of_cols + j) \times size_of_data_type] \\ &= 1000 + [(2 \times 4 + 3) \times 2] \\ &= 1000 + [(8 + 3) \times 2] \\ &= 1000 + 22 \\ &= 1022 \end{aligned}$$

Example 3: To calculate address of A(1, 2) of integer array of size 2×3 with base address 100. **[S-18]**

$$\begin{aligned} Address\ (1, 2) &= baseAddress + [(i \times no_of_cols + j) \times size_of_data_type] \\ &= 100 + [(1 \times 3 + 2) \times 2] \\ &= 100 + [5 \times 2] \\ &= 110 \end{aligned}$$

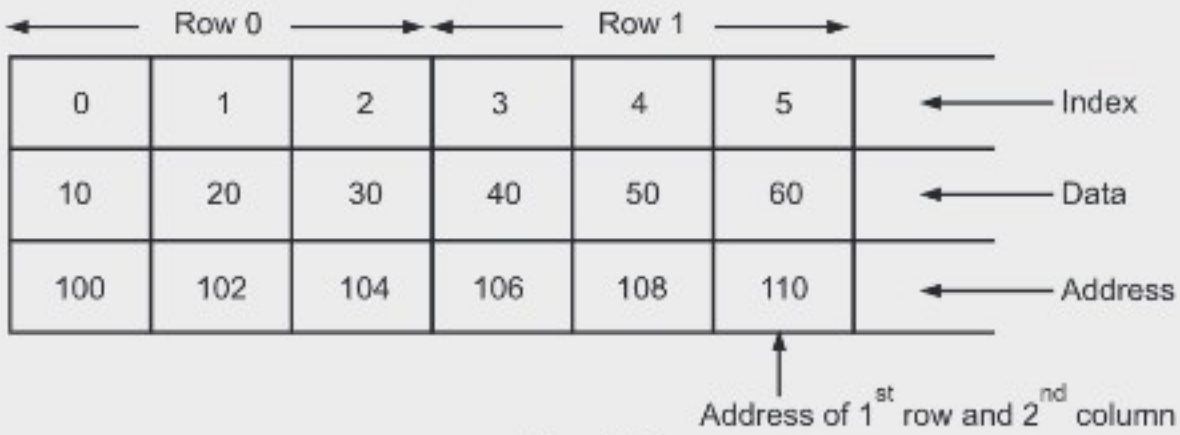


Fig. 2.8

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 2-D array A[2][3] representation in memory.

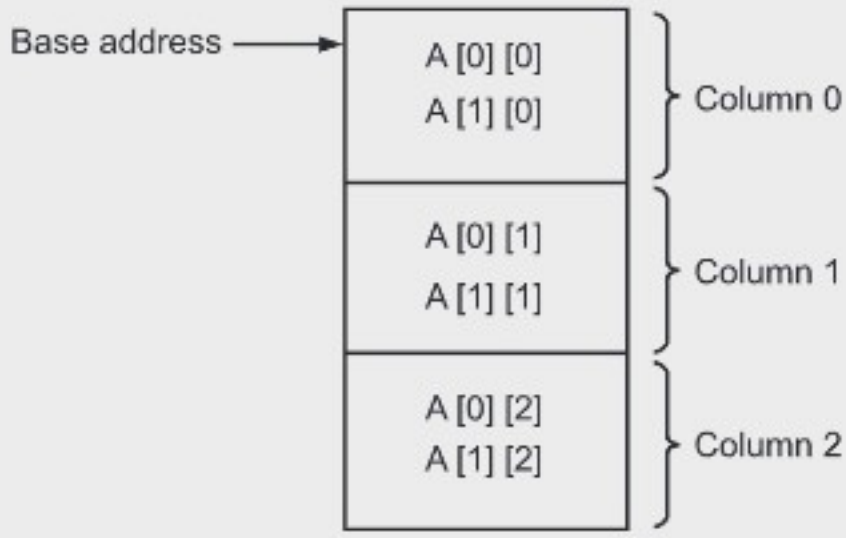


Fig. 2.9: Representation of 2-D array in column major order

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]?

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

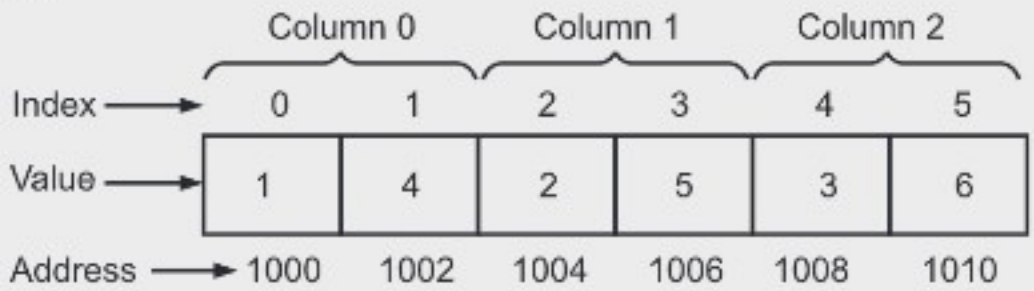


Fig. 2.10

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

$$\begin{aligned} A[1][2] &= \text{baseAddress} + [(j \times \text{no_of_rows} + i) \times \text{size_of_data_type}] \\ &= 1000 + [(2 \times 2 + 1) \times 2] \\ &= 1000 + [(4 + 1) \times 2] \\ &= 1000 + 10 \\ &= 1010 \end{aligned}$$

2.2.2.1 Sparse Matrix

- A sparse matrix is two-dimensional data object with m rows and n columns, therefore having total $m \times n$ elements in which most of the elements have value 0.
- To represent sparse matrix we can use 2D array. Sparse matrix contains lesser non-zero elements than zeros. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with triples - (Row, Column, Value), where
  - **Row:** Row index non-zero element.
  - **Column:** Column index of non-zero element.
  - **Value:** Value of the non-zero element present at index (row, column).

$$\begin{bmatrix} 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & 4 \\ 3 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 \end{bmatrix} \longrightarrow$$

Row	0	1	2	2
Column	1	3	0	3
Value	7	4	3	6

Use of Sparse Matrix:

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements.

2.2.3 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.
- The array can have more than two dimensions. An n-dimensional array A is a collection of $m_1 \times m_2 \times m_3 \ldots \times m_n$ elements. All the elements in the array are referred by subscripts, $k_1, k_2, \ldots k_n$ where,

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

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

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

The array has $2 \times 3 \times 4 = 24$ elements in it.

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

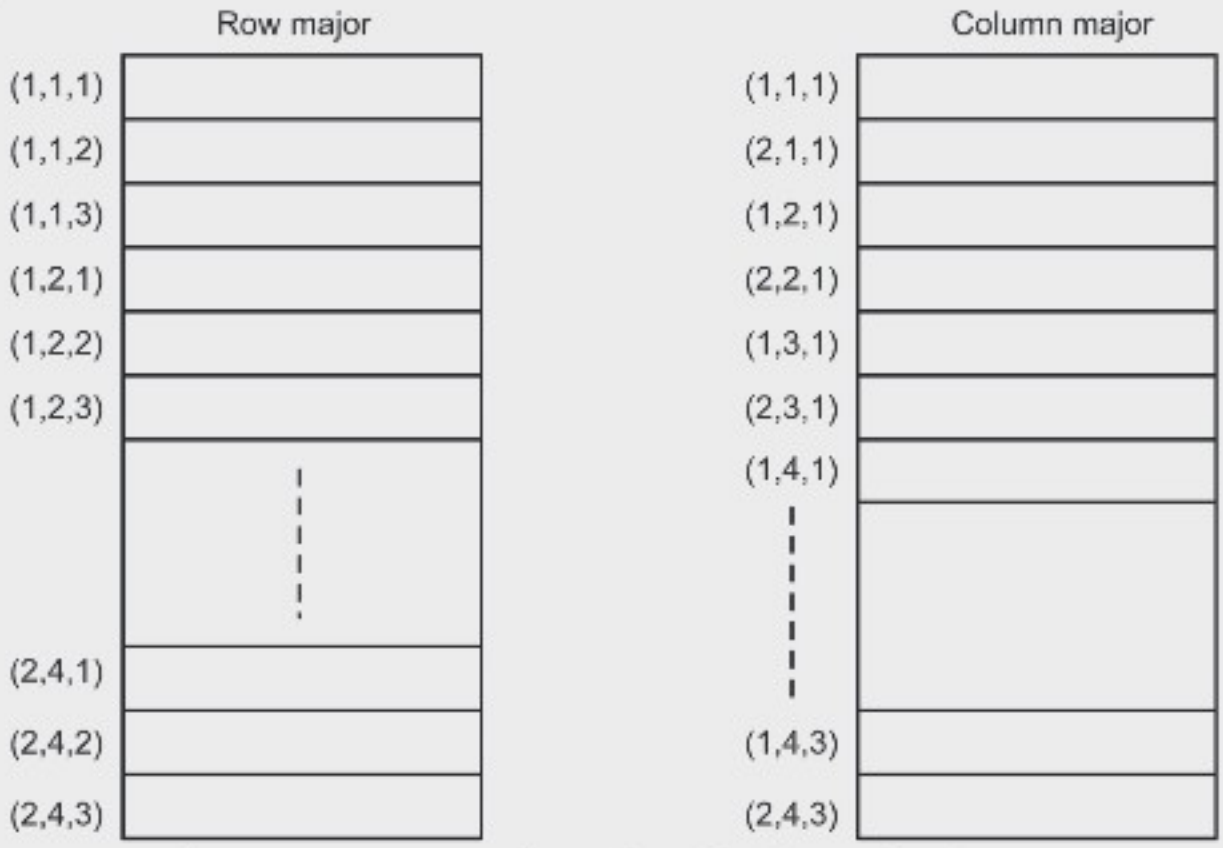


Fig. 2.11: Representation of multi-dimensional array

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]?

Solution:

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.

$$\begin{aligned} a[i]\,[j] &= \text{Base address} + (i \times \text{no_of_col} + j) \times \text{size_bytes} \\ &= 1000 + (5 \times 16 + 0) \times 4 \\ &= 1000 + 80 \times 4 \\ &= 1000 + 320 \\ &= 1320 \end{aligned}$$

2.2.4 Operations on Matrices

- There are various operations that can be applied on matrix. Some of them are listed below:
  1. **Addition:** You can add matrices together as long as their dimensions are the same, i.e. both matrices have the same number of rows and columns. To add two matrices, you add the numbers of each matrix that are in the same element position.

     Example:

     $$\begin{bmatrix} \mathbf{8} & 4 & 2 \\ 6 & 1 & 5 \end{bmatrix} + \begin{bmatrix} \mathbf{3} & 10 & 4 \\ 5 & 6 & 1 \end{bmatrix} + \begin{bmatrix} \mathbf{11} & 14 & 6 \\ 11 & 7 & 6 \end{bmatrix}$$

     For the elements in bold: 8 + 3 = 11

2. **Subtraction:** You can subtract matrices in a similar way to addition. Both matrices need to have the same dimensions, and you subtract the numbers of the second matrix from the first that are in the same element position.

   Example:

   $$\begin{bmatrix} \mathbf{8} & 4 & 2 \\ 6 & 1 & 5 \end{bmatrix} - \begin{bmatrix} \mathbf{3} & 10 & 4 \\ 5 & 6 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{5} & -6 & -2 \\ 1 & -5 & 4 \end{bmatrix}$$

   For the elements in bold: 8 - 3 = 5

3. **Multiplication:** You can multiply a matrix by some value by multiplying each element with that value. The value can be positive or negative.

   Example:

   $$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 4 & 4 \\ 10 & 8 \end{bmatrix}$$

4. **Transpose:** To transpose a matrix, we swap the rows for the columns. To indicate that we are transposing a matrix, we add a "T" to the top right-hand corner of the matrix.

   Example:

   $$\begin{bmatrix} 4 & 1 & 5 \\ 3 & 6 & 2 \end{bmatrix}^T = \begin{bmatrix} 4 & 3 \\ 1 & 6 \\ 5 & 2 \end{bmatrix}$$

2.3 SORTING CONCEPTS

- 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

Sort Stability:

- 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.
- 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

Efficiency of Sorting Algorithms

- 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.

Objectives of Sorting Algorithms:

- The objectives of sorting algorithms are listed below:
  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).

2.3.1 Sorting Techniques with Time Complexity

- Sorting is a technique to rearrange the elements in ascending or descending order, which can be numerical, lexicographical, or any user-defined order.
- For example, consider a telephone directory which consists of four fields; phone number, name, address, pin code.
- So, a large data is maintained in the form of records. If we want to search a phone number and name is in alphabetically sorted, then we can search easily. It would be very difficult if records were unsorted.
- The sorting algorithm are divided into two categories:
  1. **Internal Sorts:** This 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. Example: Bubble sort, Insertion sort, Quick sort.

2. **External Sorts:** Sorting large amount of data requires external or secondary memory. This process uses external memory such as Hard Disk, Floppy Disk, Magnetic Tape etc. to store the data which is 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. Example: Merge Sort.

Following are the sorting methods:

1. Bubble sort
2. Insertion sort
3. Merge sort
4. Quick sort

2.3.1.1 Bubble Sort

- 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 element 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 6: Consider the array A containing following elements to be sorted using bubble sort. **[S-19, W-18]**

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 [14] [15] [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 [13] [14] [15] [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 2.1: 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
```

2.3.1.2 Insertion Sort

- 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: **[W-18]**

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 7:

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
0	1	2	3	4

i.e. a[1] = 22 and j = j − 1 = −1

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:

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

Program 2.2: 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
```

2.3.1.3 Quick Sort

- 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. It 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 low^{th} 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 8: Sort the following list using quick sort method. **[S-18]**

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

Solution:

24	30	27	32	11	21	19

Select pivot = 24

Step 1 :

	lbound ↓ 0	1	2	3	4	5	Unbound ↓ 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 2.3: 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
```

2.3.1.4 Merge Sort

- 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: **[S-19]**

- 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 recurse 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)

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

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

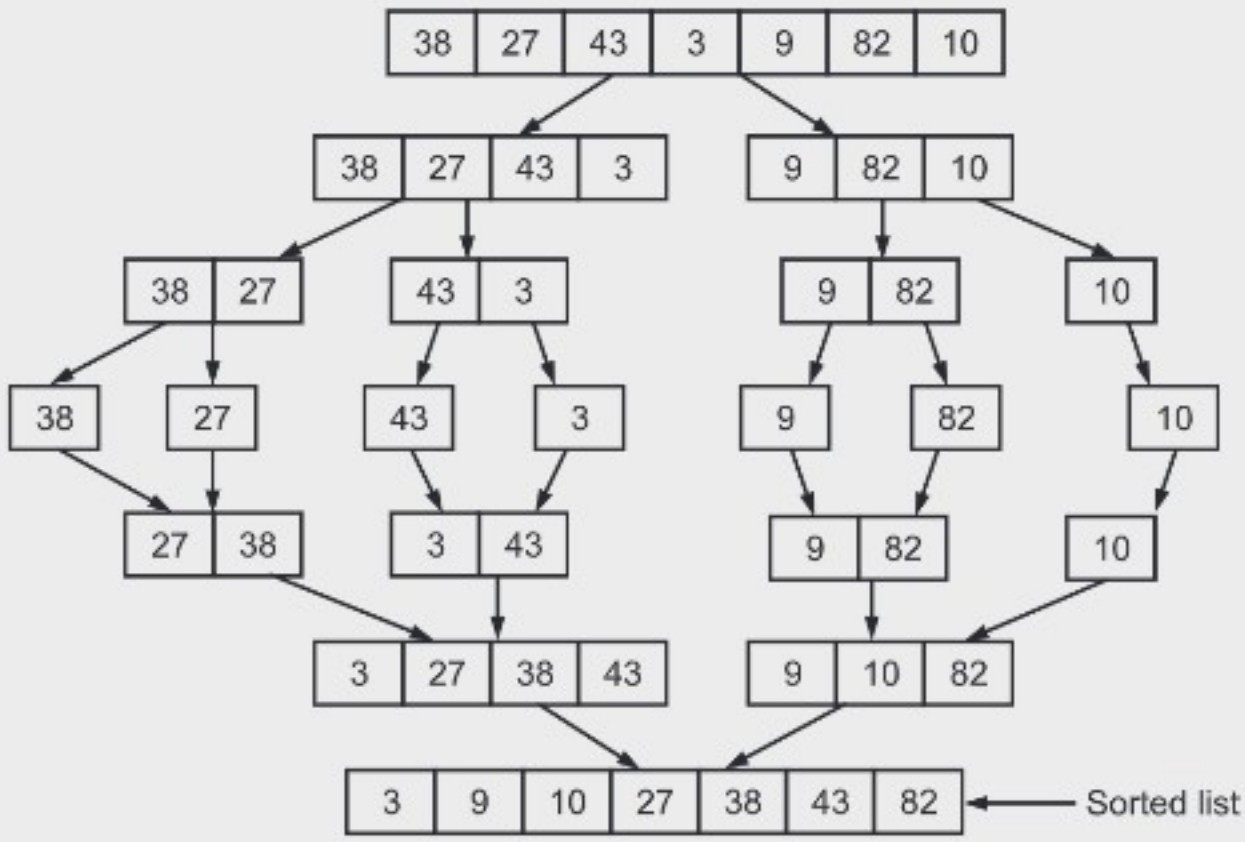


Fig. 2.12

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)

Program 2.4: 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 SEARCHING TECHNIQUES

- Searching algorithms are used to find the element in the list.
- Two searching algorithms are:
  1. Linear search
  2. Binary search

2.4.1 Linear Search or Sequential Search [S-19]

- 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 stop 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 2.5: 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.
3. When a key element matches the first element in the array, then linear search algorithm is best case because executing time of linear search algorithm is O(n), where n is the number of elements in an array.

Disadvantages:

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 doesn't matches any element then linear search algorithm is a worst case.

2.4.2 Binary Search [W-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 3 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 10:

Consider array a

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

n = 7, key = 11

Step 1 : Initialization lb = 0, ub = 6

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

Step 3 : As 11 < a[3], first half need to be checked

ub = mid - 1 = 3 - 1 = 2

Step 4 : $\text{mid} = \left[\frac{(0+2)}{2}\right] = \left[\frac{2}{2}\right] = 1$

Step 5 : As 11 > a[1], second half need to be checked.

lb = mid + 1 = 1 + 1 = 2

Step 6 : $\text{mid} = \left[\frac{(2+2)}{2}\right] = \left[\frac{4}{2}\right] = 2$

Now 11 == a[2] return 2

i.e. 11 is present at position 2 in an array a.

Example 11: n = 9, key = 25.

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

– 1

↖ BS (A, 0, 8, 25) mid = 4

↓↑ – 1

BS (A, 0, 3, 25) mid = 1

↓↑ – 1

BS (A, 2, 3, 25) mid = 2

↓↑ – 1

BS (A, 3, 3, 25) mid = 3

↓↑ – 1

BS (A, 4, 3, 25)

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 : $mid = \left[\left(\frac{low + high}{2}\right)\right]$
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 12:

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.

Advantage:

It is very time efficient as time complexity is O(log n).

Disadvantage:

Array should be sorted before applying this algorithm on it.

Complexity:

The maximum number of key comparisons are $\log_2 n$.

So complexity is O(log n).

Summary

- A one-dimensional array is of fixed size sequence of elements of the same type.
- Array elements can be access using index.
- A two-dimensional array is a representation of a table with rows and columns.
- A multidimensional array is represent in memory with row major or column major order.
- An in-place algorithm is an algorithm that does not need an extra space and produces an output in the same memory that contains the data by transforming the input 'in-place'. However, a small constant extra space used for variables is allowed.
- Some common internal sorting algorithms include: Bubble Sort, Insertion Sort, Quick Sort.
- An external sorting algorithm needs extra space to put the elements in as it's sorting them. Usually this means O(n) extra space.
- Merge sort is an example of external sorting algorithm.
- It should be clear that no single sort is best for all applications. Certain properties such as the number, size, distribution and order of keys of a given key set plays an important role in determination of sorting technique which should be used for that particular application.

- Quick sort is also called as partition exchange sort.
- Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. There are two searching algorithms Linear search and Binary search

Check Your Understanding

1. Which of the following data structure store the homogeneous data elements?
   (a) Arrays (b) Records
   (c) Pointers (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

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)	

Practice Questions

Q.I Answer the following questions in short:

1. How are arrays declared?
2. How are two-dimensional array elements stored in the memory?
3. Is it possible to have negative index?
4. Calculate the address of A(2,3) of integer array of size 4 * 4 with base address 100.
5. What do you mean by a stable sorting method?
6. List the examples of stable sort.
7. What is the best case and worst case time complexity of merge sort?

Q.II Answer the following questions:

1. Explain the row major representation of an array.
2. What is difference between internal and external sorting technique.
3. Discuss the internal and external sorting by taking suitable example of each type.
4. Write the algorithm of bubble sort.
5. 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
6. Explain insertion sort algorithm with time complexity.
7. 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
8. Sort the following sequence using quick sort algorithm.

   35, 45, 42, 57, 26
9. What is the best case and worst case efficiency of the following algorithm:
   (a) Bubble sort
   (b) Insertion sort
   (c) Quick sort
10. Explain linear searching technique in detail.
11. Write a C function for sequential search on sorted array.
12. Write a binary search algorithm to search a given value called val in a list of N numbers called series.

Q.III Define the term:

1. Array
2. Pivot
3. Sort order
4. Sort stability
5. Internal sort
6. External sort
7. Linear search

Previous Exam Questions

Summer 2019

1. Sort the following numbers using bubble sort method: **[5 M]**
108, 3, 97, 65, 71, 23, 57, 93, 100

Ans. Refer to Example 6.

2. Write algorithm for merge sort. **[4 M]**

Ans. Refer to section 2.3.1.4.

3. Explain linear searching technique in detail. **[3 M]**

Ans. Refer to section 2.4.1.

Winter 2018

1. The quick sort algorithm exploit _____ design technique. **[1 M]**

(i) Greedy (ii) Divide and conquer
(iii) Dynamic programming (iv) Backtracking

Ans. (ii)

2. Sort the following data using bubble sort: **[3 M]**
55, 22, 66, 33, 44, 11

Ans. Refer to Example 6.

3. Explain insertion sort algorithm with time complexity. **[3 M]**

Ans. Refer to section 2.3.1.2

4. Write Binary Search Algorithm to search a given value called val in a list of N numbers called Series. **[4 M]**

Ans. Refer to section 2.4.2

Summer 2018

1. The complexity of linear search is _____. **[1 M]**

(i) O (log(n)) (ii) O (n)
(iii) O (n log n) (iv) none of these

Ans. (ii)

2. Calculate address of A (2, 3) of integer array of size 4 × 4 with base address 100. **[1 M]**

Ans. Refer to Example 3.

3. Sort the given array using quick sort algorithm **[5 M]**

35, 45, 42, 57, 26

Ans. Refer to Example 8.

4. Write the algorithm for bubble sort. **[4 M]**

Ans. Refer to Section 2.3.1.1.

5. Explain the row-major representation of an array. **[3 M]**

Ans. Refer to Section 2.2.2.

■■■

3... Linked List

Objectives...

- To study basic concepts of Linked List.
- To study need of Dynamic Data Structure.
- To study Linked List as Dynamic Data Structure.
- To study different Operations on Linked List.
- To study various Applications of Linked List.

3.1 INTRODUCTION [S-19]

- In array data structure, space cannot be increased when required as array is static data structure. This drawback is removed with another linear data structure called linked list.
- The arrays are static in nature, once the array is declared; its size remains the same. Linked list can be expanded or shrunk as per requirement.
- The memory allocated to 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 deletions are easy due to pointers.
- Array supports Random Access, which means elements can be accessed directly using their index. Linked List supports Sequential Access, to access any element/node in a linked list.
- Memory is allocated as soon as the array is declared, at compile time. It's also known as Static Memory Allocation. Memory is allocated at runtime, as and when a new node is added. It's also known as Dynamic Memory Allocation.

3.2 LINKED LIST

3.2.1 Definition [S-18]

- A linked list or one-way list is a linear collection of data elements, called nodes. Each node contain two parts, first part is data and second part is link to its successor (and predecessor).

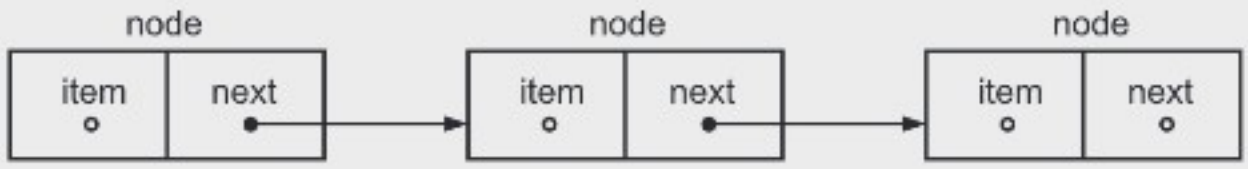


Fig. 3.1: Structure of linked list

- The last node in the list contains a NULL pointer to indicate that it is the end or tail of the list.
- **Example:** We all know about very curious game "Treasure Hunt". In this, player is provided the initial clue of the first location. From first locations clue, player gets the second and so on. To reach the final destination, the players have 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.

3.2.2 Linked Organization Characteristics

- Elements can be placed anywhere in the memory.
- Dynamic allocation i.e. size need not be known in advance.
- Insertion and deletion of data do not require any data movement.
- Each element 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: Linked list with 4 nodes.

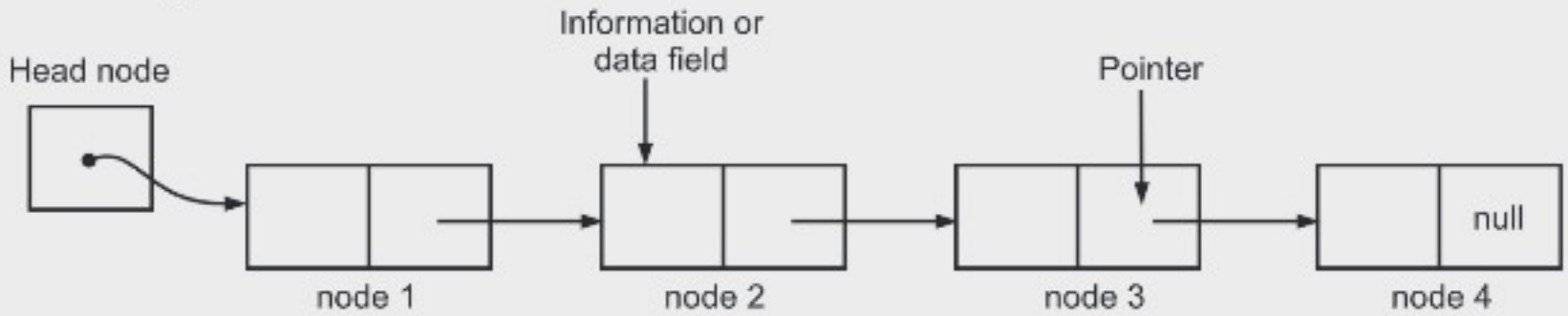


Fig. 3.2: Linked list with 4 nodes

- Each node have 2 fields; 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, then 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.

3.2.3 Primitive Operations on Linked List

- Primitive operations on linked list are given as below:
  1. Create
  2. Traverse
  3. Insert a node
  4. Delete a node
- 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

3.3 REPRESENTATION OF LINKED LIST IN MEMORY

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

3.3.1 Static Representation of Linked List (Using Array)

- 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.
- Following example elaborates concept of static linked list.
- **Example 1:** 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. 3.3. The elements are not stored in contiguous block of locations as well as not stored in same order as in the list L.

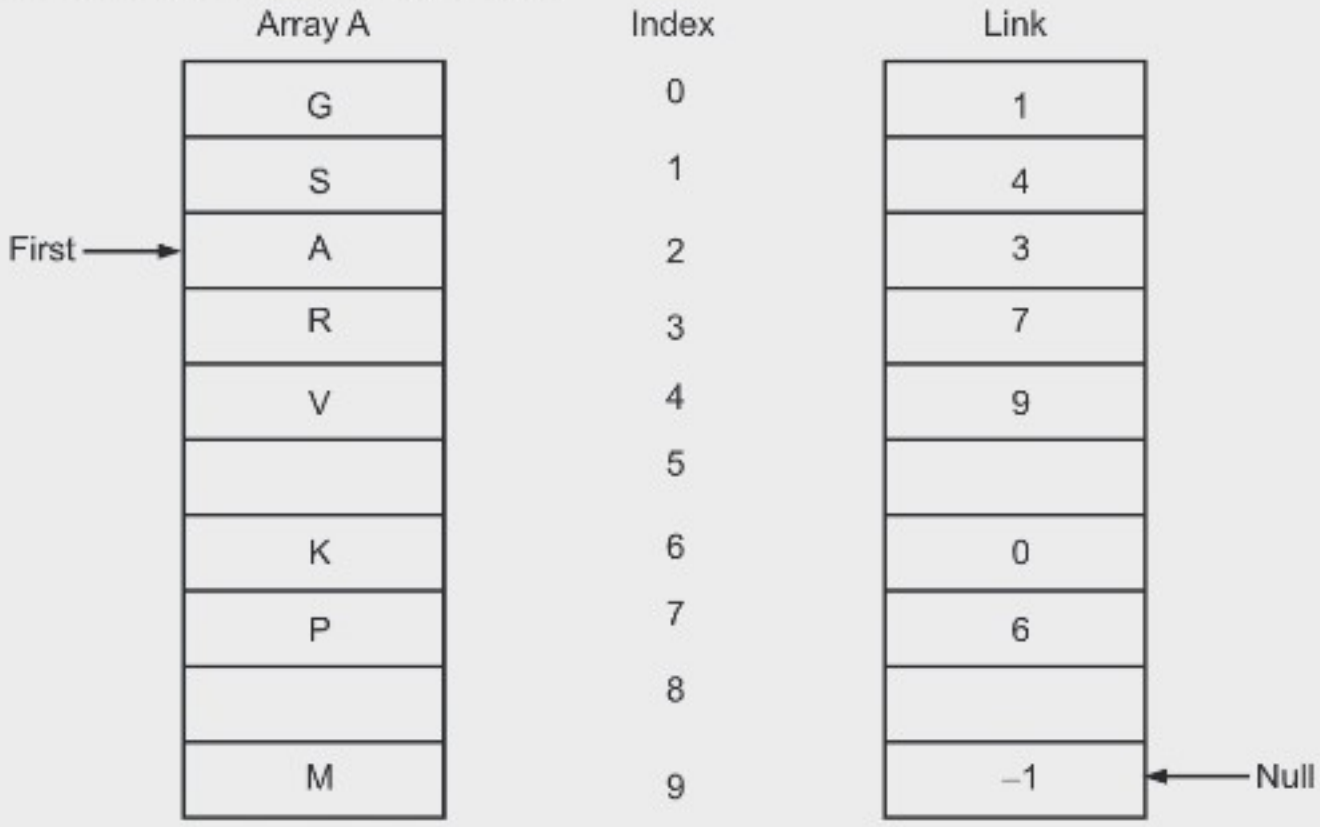


Fig. 3.3: Representation of linked list using array

- 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. 3.4 shows the above representation is different manner, after omitting unused locations.

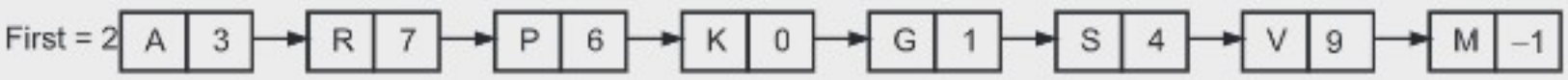


Fig. 3.4: Linked organization

- **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: **[S-18]**

- Array representation of linked list consists of following drawbacks:
  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.

3.3.2 Dynamic representation of Linked List

- 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 maintain data elements in a logical order rather than in physical order.
- **Example:** This is an example of singly linked list.

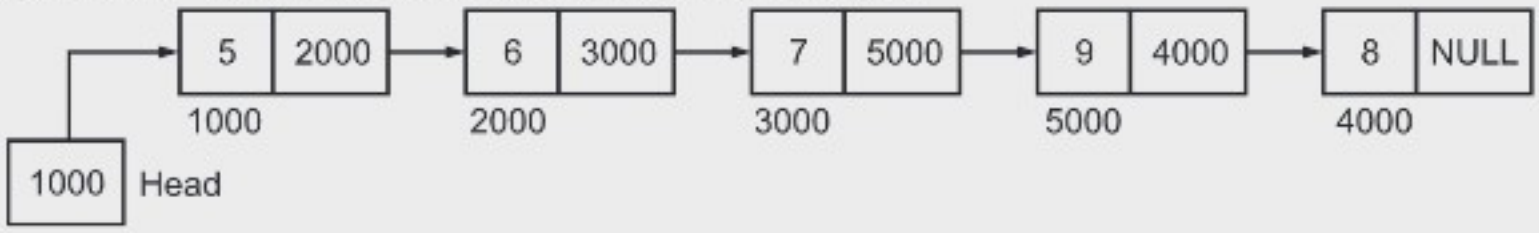


Fig. 3.5: Representation of linked list using dynamic memory allocation

- **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.

3.4 TYPES OF LINKED LIST [W-18, S-18]

- Linked list are classified into following three types:

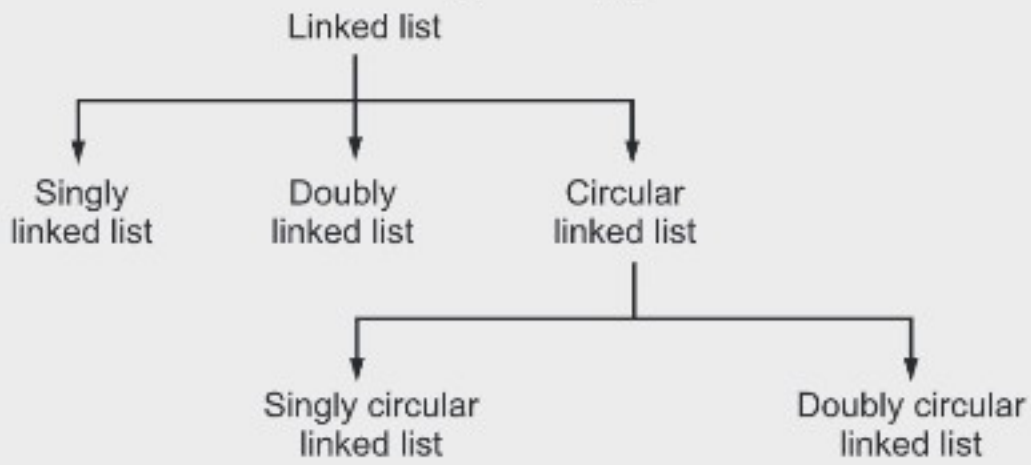


Fig. 3.6: Types of linked list

3.5 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. 3.7 shows the singly linked list.

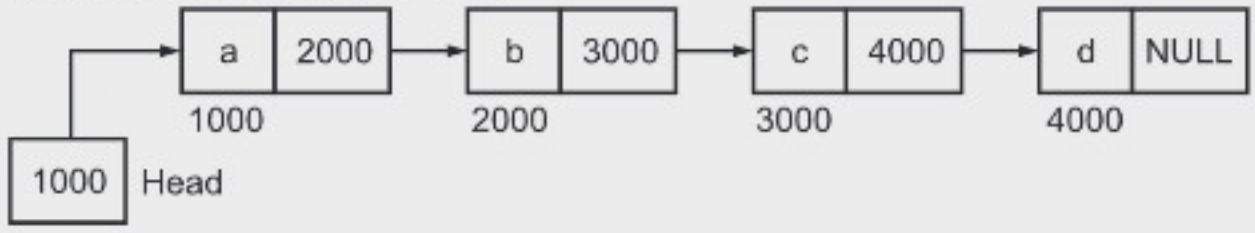


Fig. 3.7: Singly linked list

3.5.1 OPERATIONS ON LINKED LIST IN C [S-19]

3.5.1.1 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 2: 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'.
  - `newnode=(struct node *)malloc (sizeof(struct node));`
  - `strcpy(newnode->data,"Sharvari");`
  - `newnode->next=NULL;`

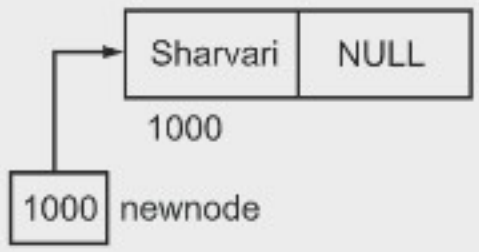


Fig. 3.8 (a)

Step 3 : Append this node in the linked list.

As `head = NULL`, case 1 is true.

```
head=last=newnode;
```

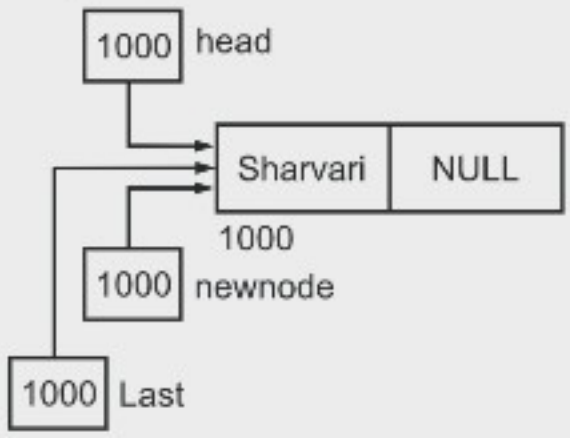


Fig. 3.8 (b)

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;`

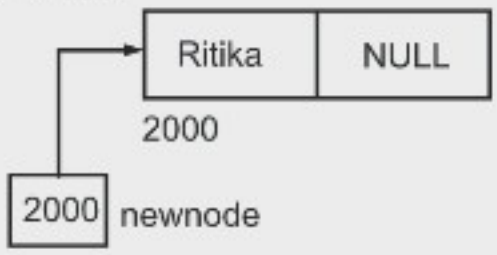


Fig. 3.8 (c)

Step 5 : Append this node in the linked list

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

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

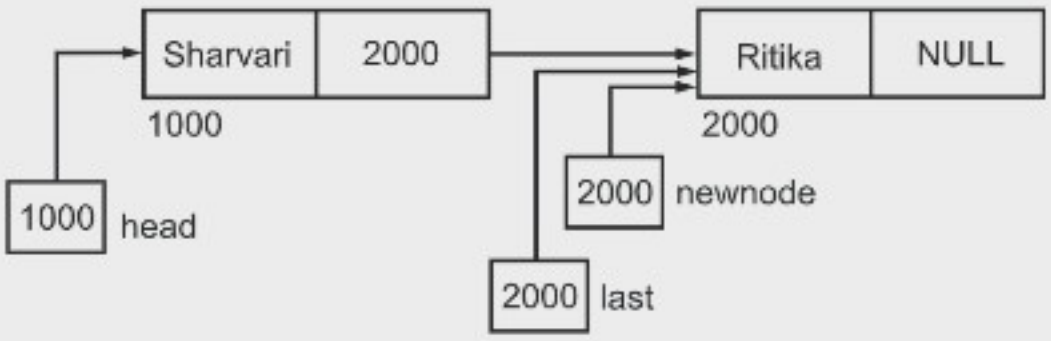


Fig. 3.8 (d)

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

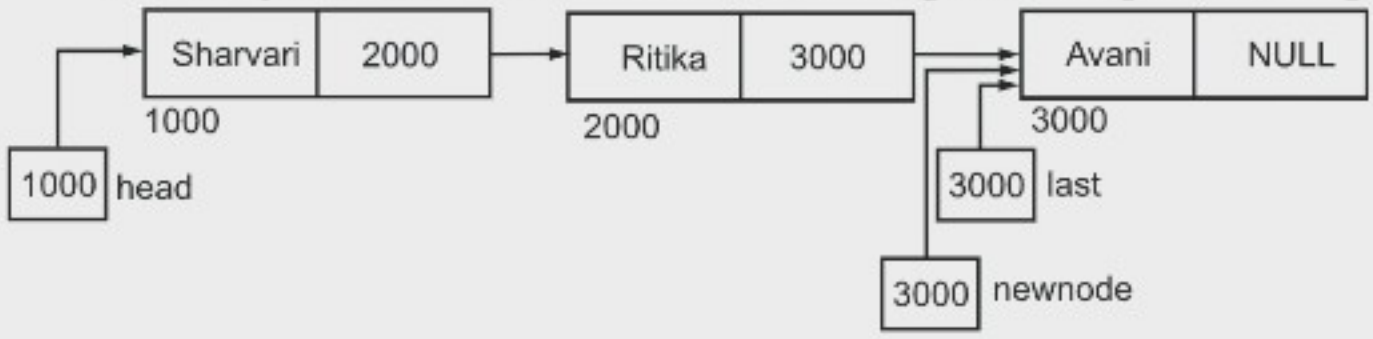


Fig. 3.8 (e)

Fig. 3.8 (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().

3.5.1.2 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: **[S-19]**

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

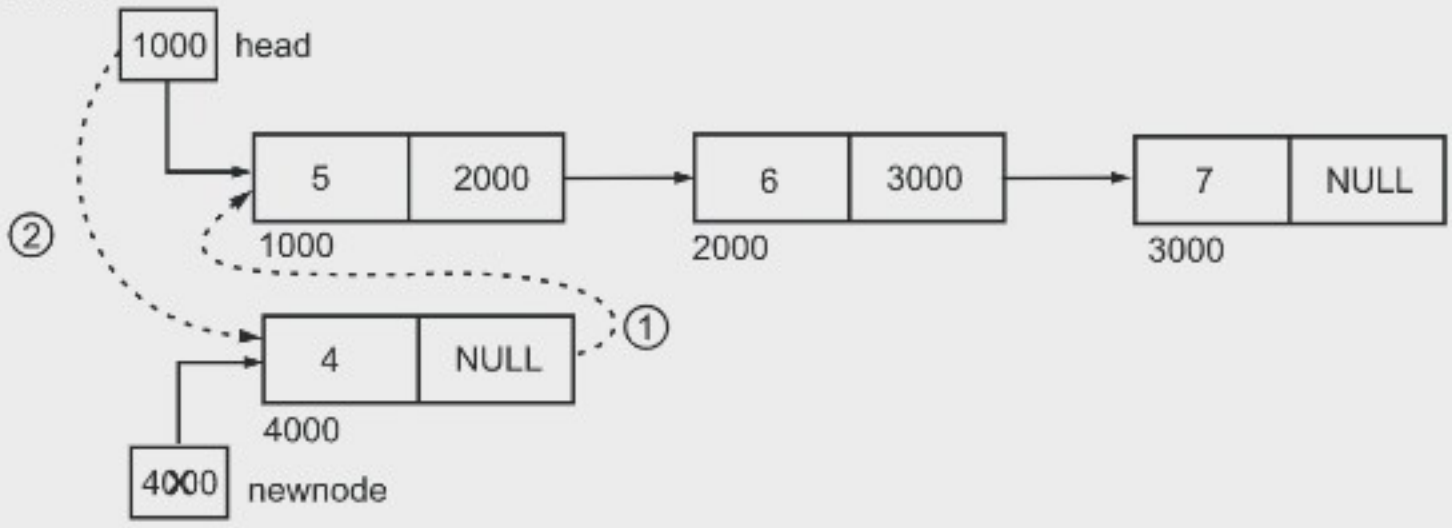


Fig. 3.9: (a) Before Insertion at the beginning

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

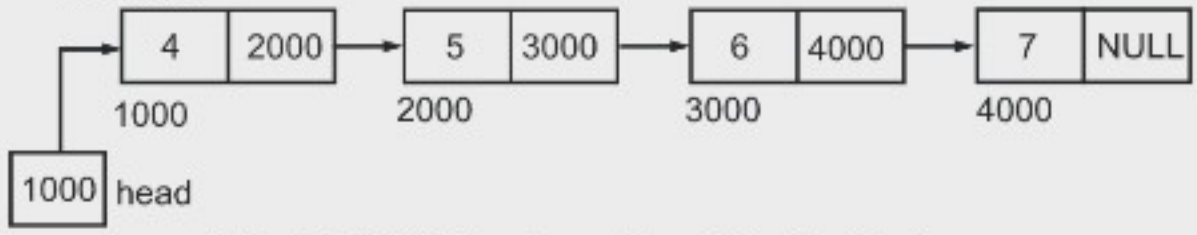


Fig. 3.9: (b) After insertion at the beginning

- **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.

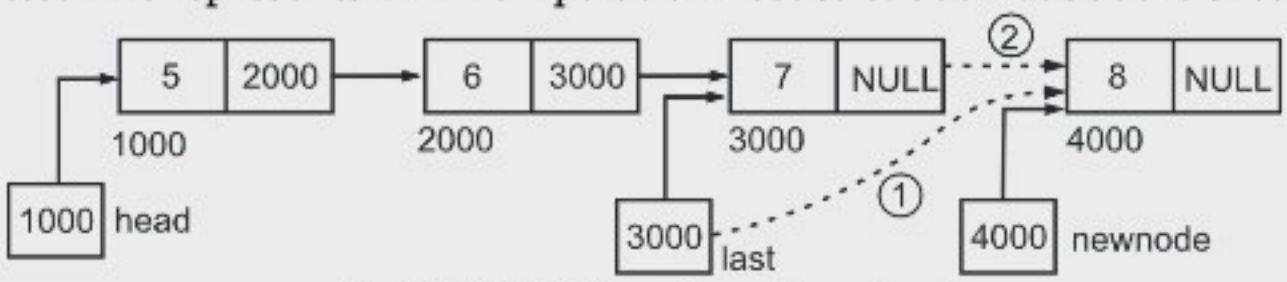


Fig. 3.10: (a) Before Insertion at end

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

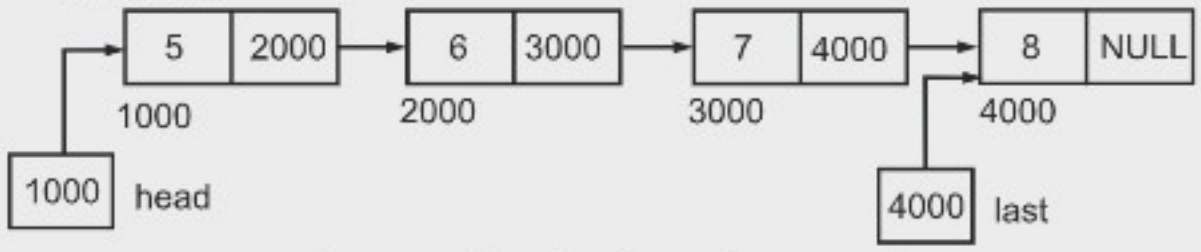


Fig. 3.10: (b) After insertion at end

- **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.

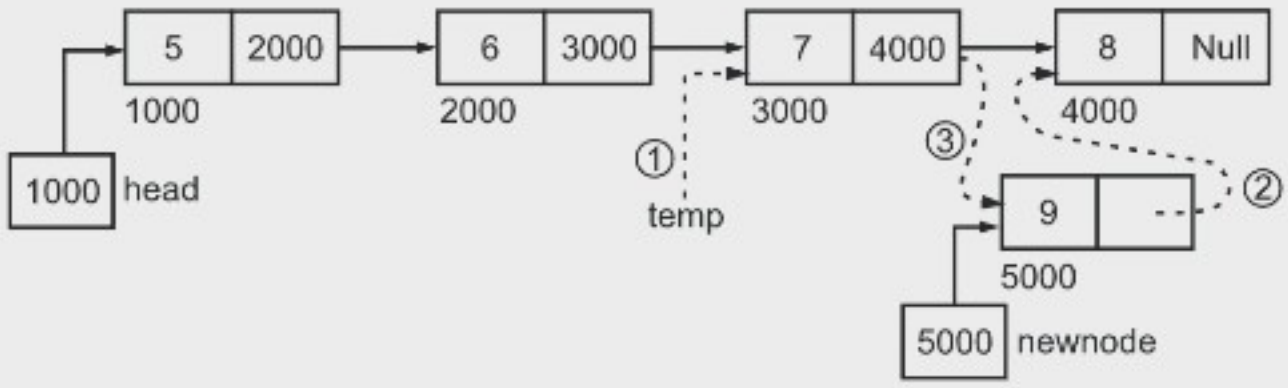


Fig. 3.11: (a) Before Insertion at fourth position

- 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 (b).
  - `Traverse temp pointer from head to till position-1.`
  - `newnode -> next = temp -> next;`
  - `temp->next=newnode;`

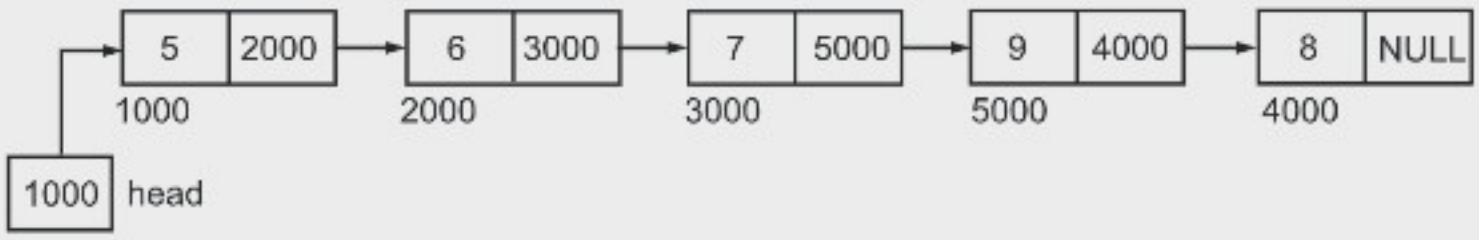


Fig. 3.11: (b) After Insertion at any specified node

- **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;
}
```

3.5.1.3 Deleting a Node from Linked List [S-19]

- 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:

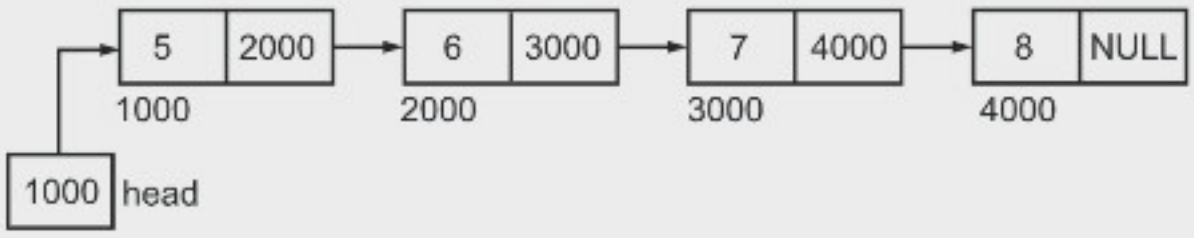


Fig. 3.12 (a): Input linked list

1. Deletion from beginning:

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

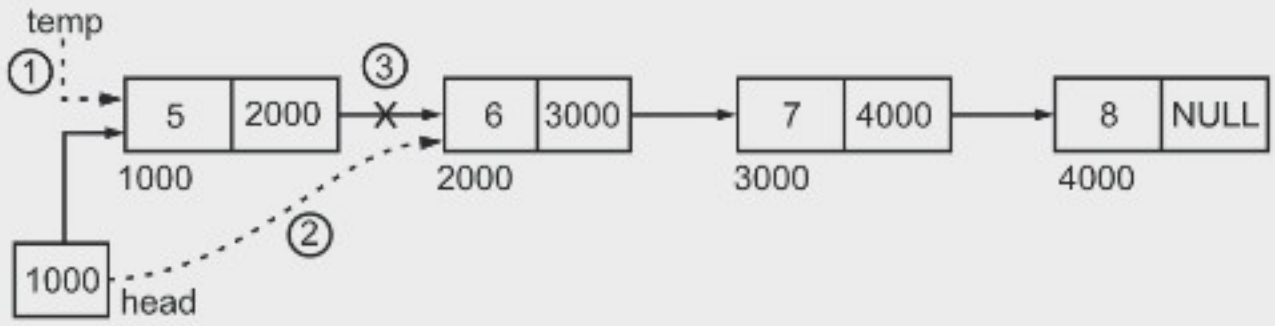


Fig. 3.12: (b) Before deletion from front in SLL

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

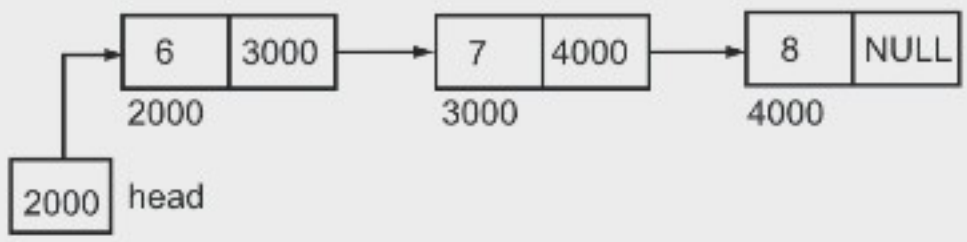


Fig. 3.12: (c) After deletion from front in SLL

- **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'.

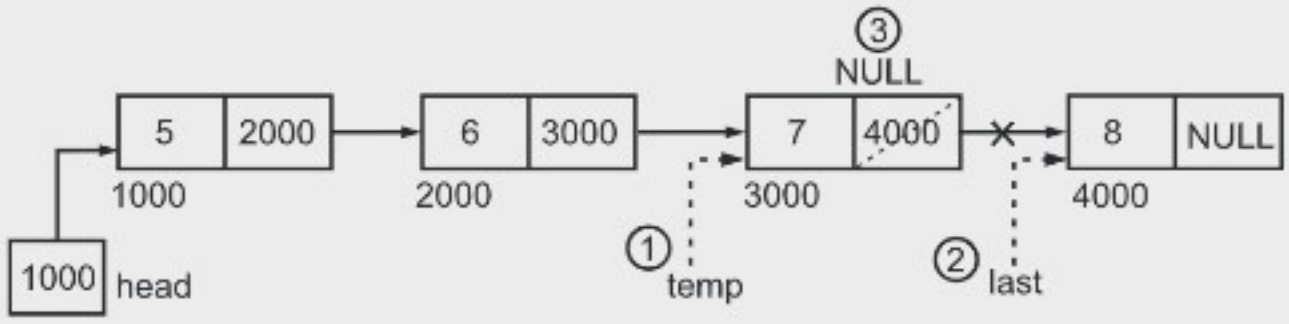


Fig. 3.13 (a): Before deletion from end in SLL

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

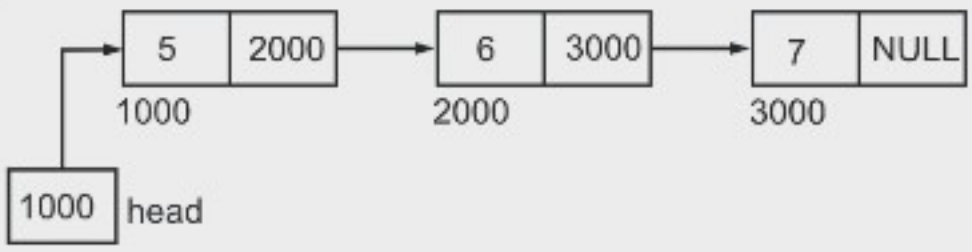


Fig. 3.13 (b): After deletion from end in SLL

- **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'.

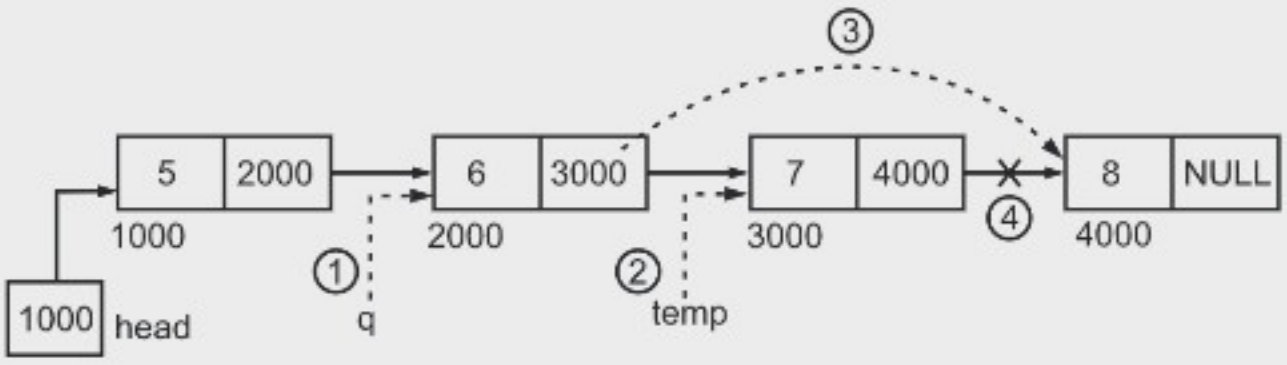


Fig. 3.14: (a) Before Deletion at specified position

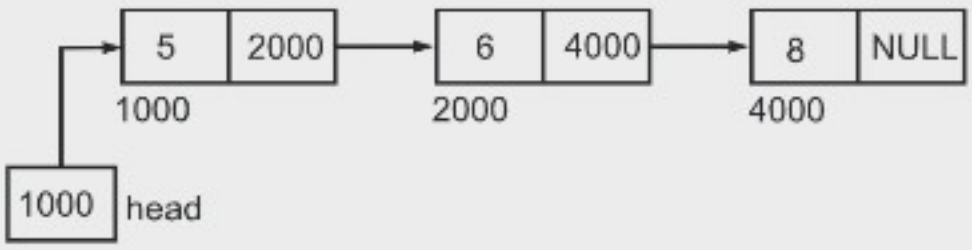


Fig. 3.14: (b) After Deletion at specified position

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;
}
```

3.5.1.4 Traversing/Display a Linked List [S-18]

- 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.

Algorithm:

(i) Get the address of first node, say 'temp'.

(ii) If 'temp' is not null, repeat step (iii) and step (iv).

(iii) Process the data field of 'temp'. Process includes display, update etc.

(iv) Move to next node using 'next' field of 'temp'.

(v) 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.

3.5.1.5 Reverse a Linked List [W-18]

1. **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. **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);
}
```

3.5.1.6 Concatenation of Linked List

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.

3.5.1.7 Length of Linked List

C function to compute length of singly linked list is written as:

```
int length (struct node *head)
{
    int cnt = 0;
    while (head!=NULL)
    {
        cnt++;
        head=head->link;
    }
    return cnt;
}
```

3.5.1.8 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 element then the location of the element is returned from the function.

Algorithm: [W-18]

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");
        }
    }
}
```

3.5.1.9 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
        }
}
```

3.5.1.10 Merge the Elements of Linked List

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 3.1: To implement singly linked list.

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
static int total_nodes;
struct node *create(struct node *head,int n)
{
   int i;
   struct node *newnode=NULL,*last=NULL;
```

```
    for(i=1;i<=n;i++)
    {
        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(head==NULL)
            head=last=newnode;
            else
            {
                last->next=newnode;
                last=newnode;
            }
        }
     }
    total_nodes=n;
    return head;
}
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;
    }
}
struct node * 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)
        {
            newnode->next=head;
            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;
}
struct node * delete(struct node *head, int pos)
{
    int i;
    struct node *temp=NULL,*q=NULL;
    temp=head;
    if(pos==1)
        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);
        --total_nodes;
        return head;
}
int main()
{
    struct node *head=NULL;
    int i,ch,pos,ans,n;
    do
    {
        printf("Enter +ve no of nodes to be create\n");
        scanf("%d",&n);
    }while(n<=0);
    head=create(head,n);
    do
     {
        printf("\n 1-Insert\n 2-Delete\n 3-Display\n");
        printf("\nEnter your choice\n");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:printf("Enter position no at which you want to insert a
                   node\n");
                   scanf("%d",&pos);
                   if(pos>0 && pos<= total_nodes+1)
                       head=insert(head,pos);
                   else
                       printf("Invalid position\n");
                   break;
```

```
            case 2:if(head==NULL)
                      printf("\nLinked List is Empty\n");
                   else
                   {
                      printf("Enter position no from which you want to delete
                      a node\n");
                      scanf("%d",&pos);
                      if(pos>0 && pos<= total_nodes)
                         head=delete(head,pos);
                      else
                         printf("Invalid position\n");
                   }
                   break;
            case 3:if(head==NULL)
                      printf("\nLinked List is Empty\n");
                   else
                      display(head);
                   break;
                }
              }while(ch>0 && ch < 4);
            return 0;
         }
```

Output:

```
Enter +ve no of nodes to be create
2
Enter node data
10
Enter node data
20
   1-Insert
   2-Delete
   3-Display
Enter your choice
1
Enter position no at which you want to insert a node
1
```

```
Enter node data
30
   1-Insert
   2-Delete
   3-Display
Enter your choice
3
Linked list contents are :
30 10 20
   1-Insert
   2-Delete
   3-Display
Enter your choice
2
Enter position no from which you want to delete a node
1
Deleted element is : 30
   1-Insert
   2-Delete
   3-Display
Enter your choice
3
Linked list contents are :
10 20
   1-Insert
   2-Delete
   3-Display
Enter your choice
```

3.6 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 i^{th} node given only pointer to i^{th} 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.

3.6.1 Definition

- 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 Figs. 3.15.

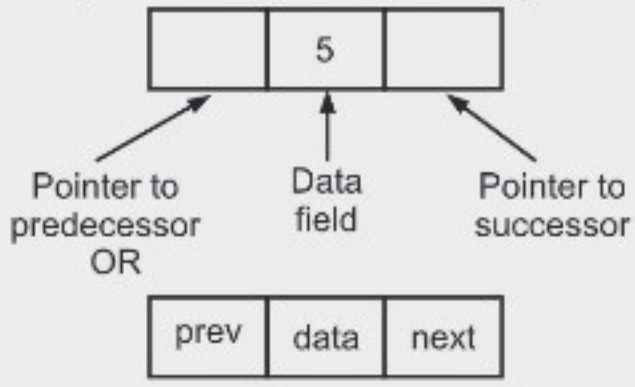


Fig. 3.15: A node structure

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

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

Example of doubly link list:

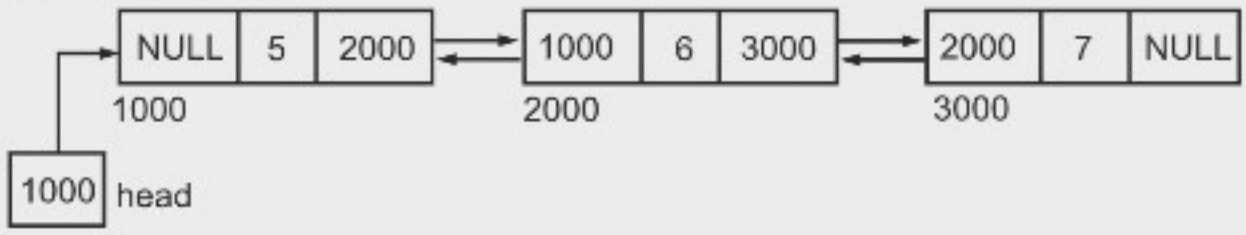


Fig. 3.16: Doubly linked list

3.6.2 Operations on Doubly Linked List (DLL)

3.6.2.1 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 3: 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;`

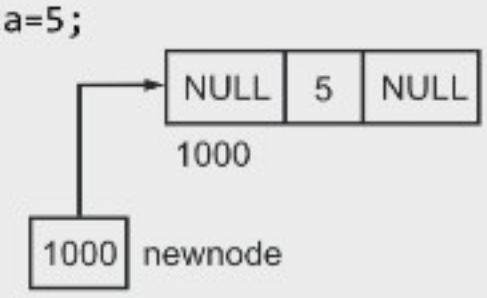


Fig. 3.17 (a)

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

`As head=NULL, case 1 is true.`

`head=last=newnode;`

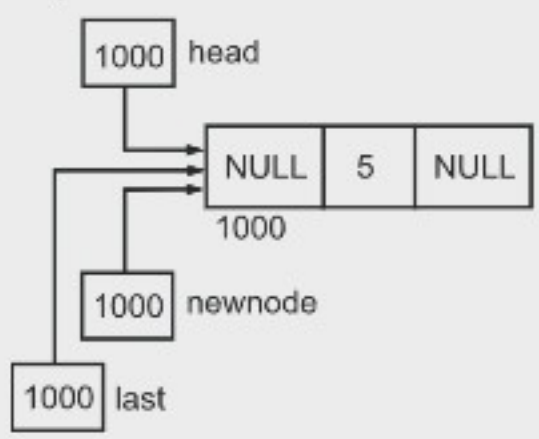


Fig. 3.17 (b)

Step 4 : **Create a new node-** Consider node data as 6.

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

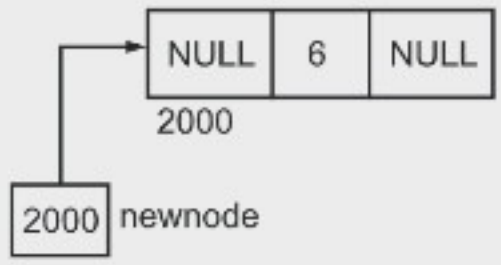


Fig. 3.17 (c)

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;`

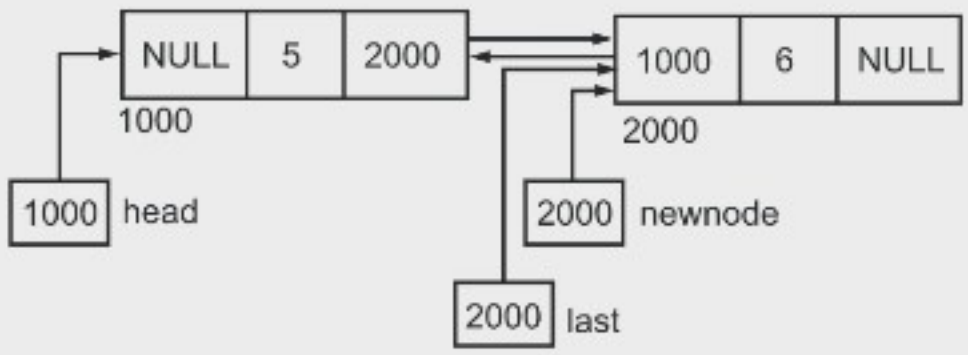


Fig. 3.17 (d)

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

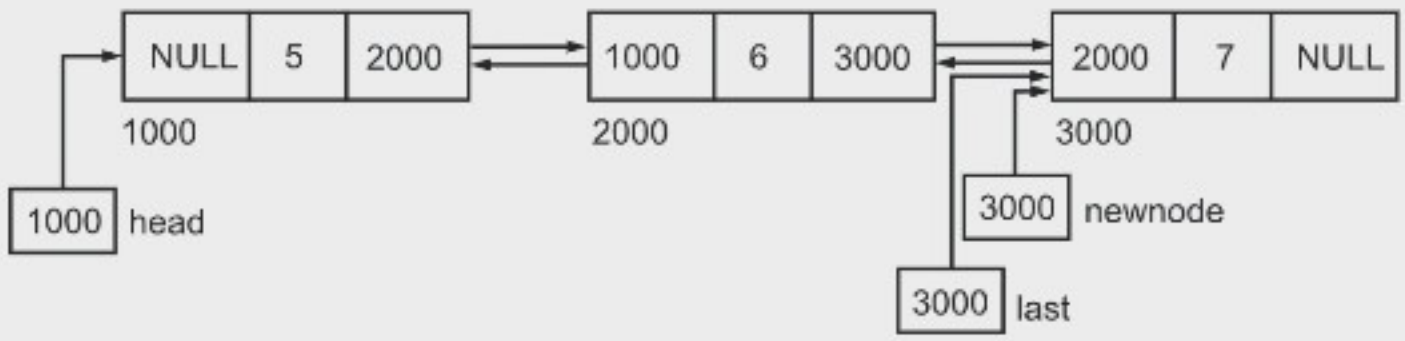


Fig. 3.17 (a) (b) (c) (d): Example of Create and Append node in doubly linked list

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;
}
```

3.6.2.2 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.

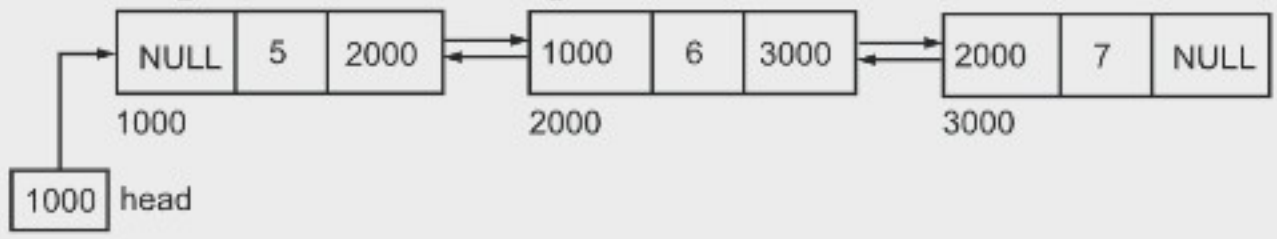


Fig. 3.18 (a): Input Doubly Linked List

1. To insert node at first position: W-18

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

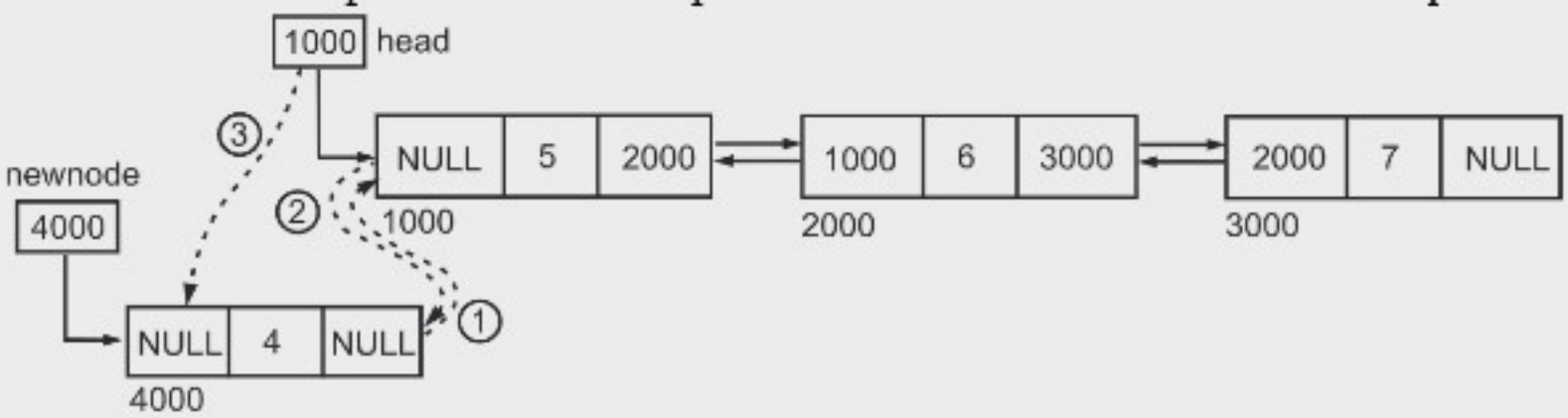


Fig. 3.18: (b) Before Insertion at the beginning in DLL

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

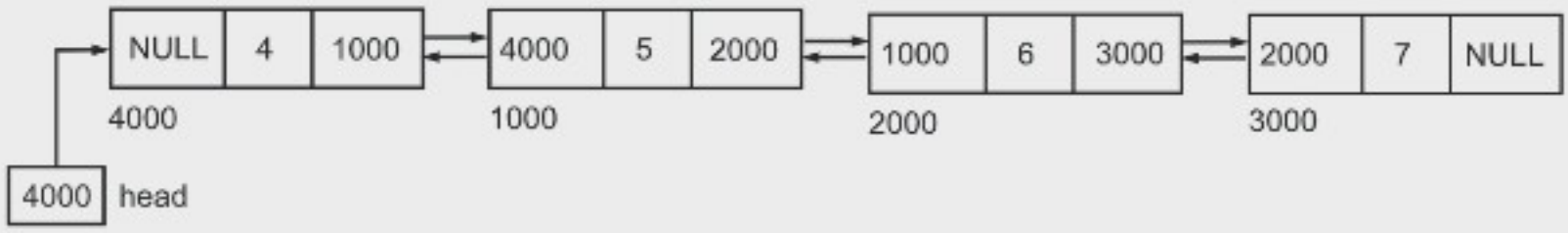


Fig. 3.18: (c) After insertion at the beginning in DLL

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.

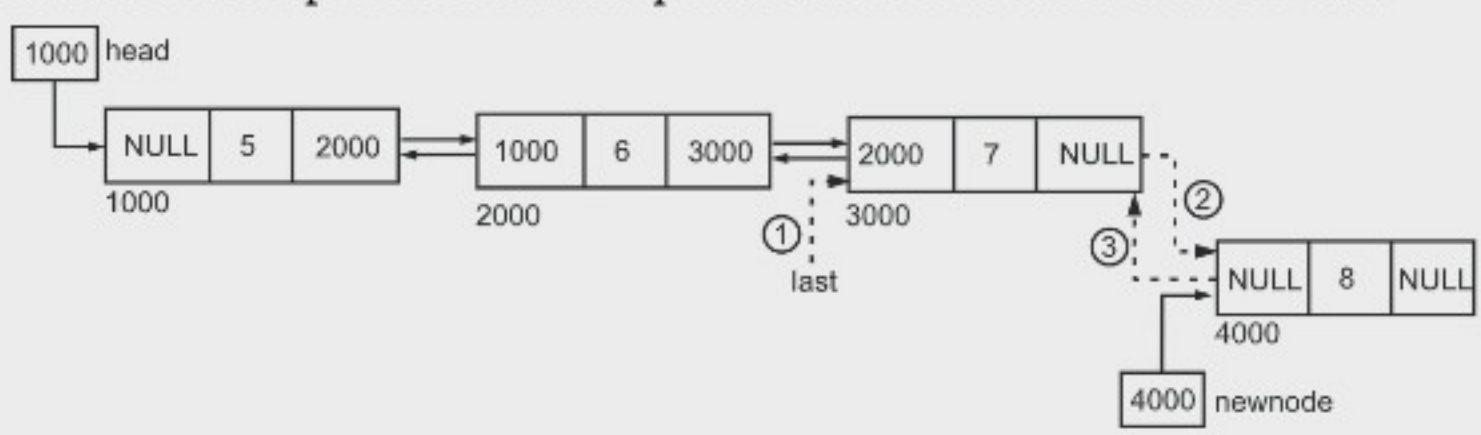


Fig. 3.19: (a) Before Insertion at end in DLL

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

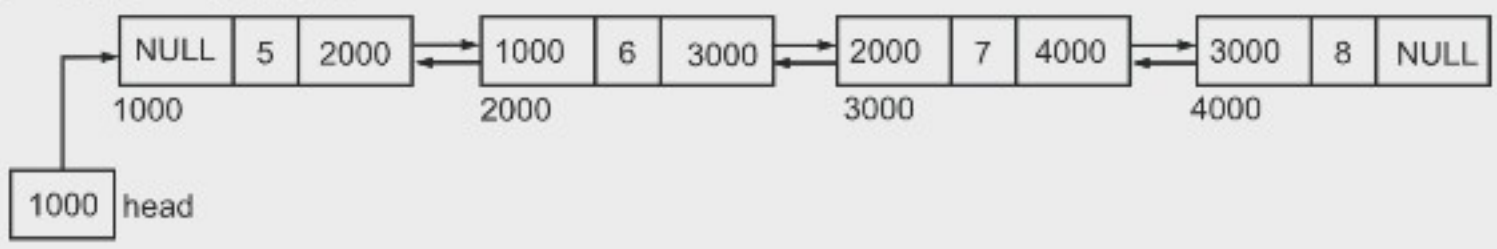


Fig. 3.19: (b) After insertion at end in DLL

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.

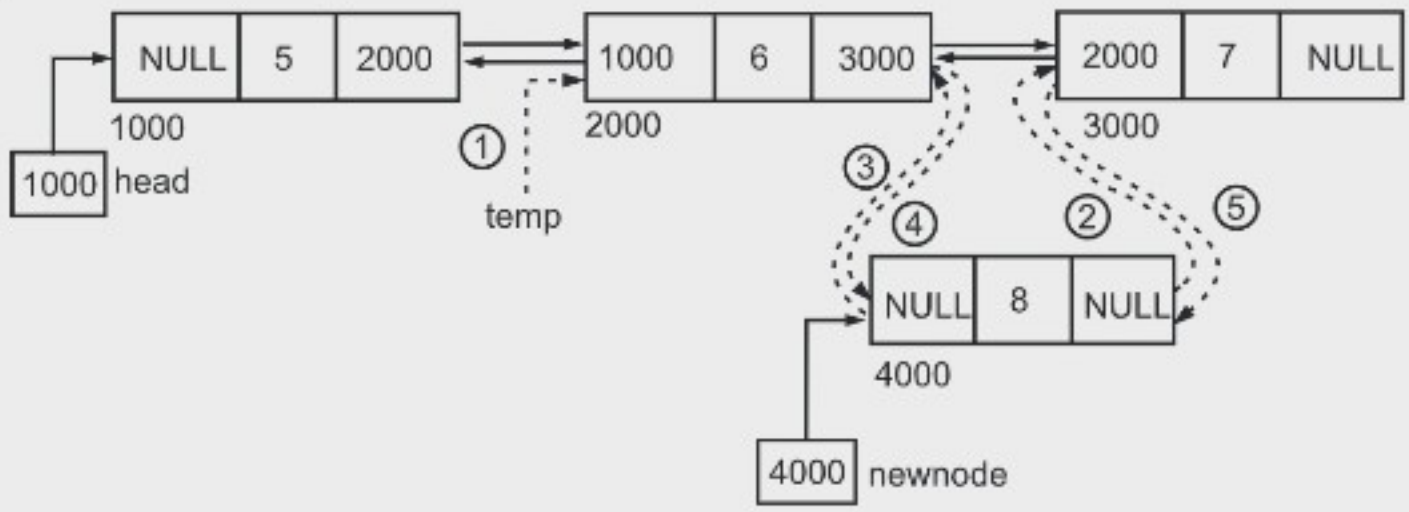


Fig. 3.20: (a) Before Insertion at fourth position in DLL

- 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;`

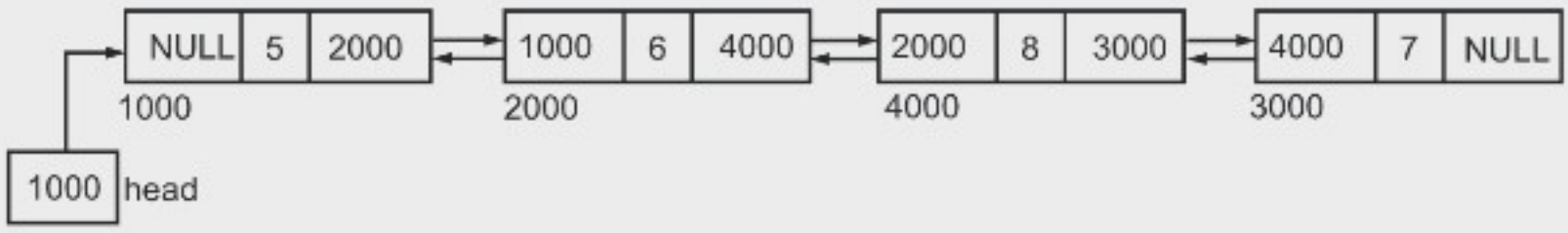


Fig. 3.20: (b) After Insertion at any specified node in DLL

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;
}
```

3.6.2.3 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:

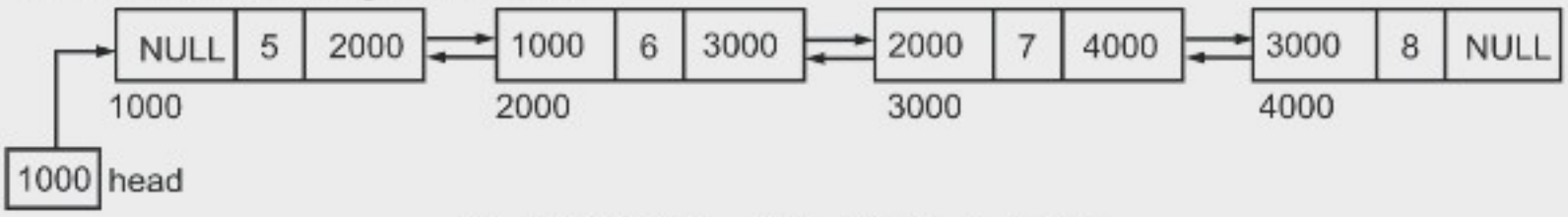


Fig. 3.21 (a): Input Doubly Linked List

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.

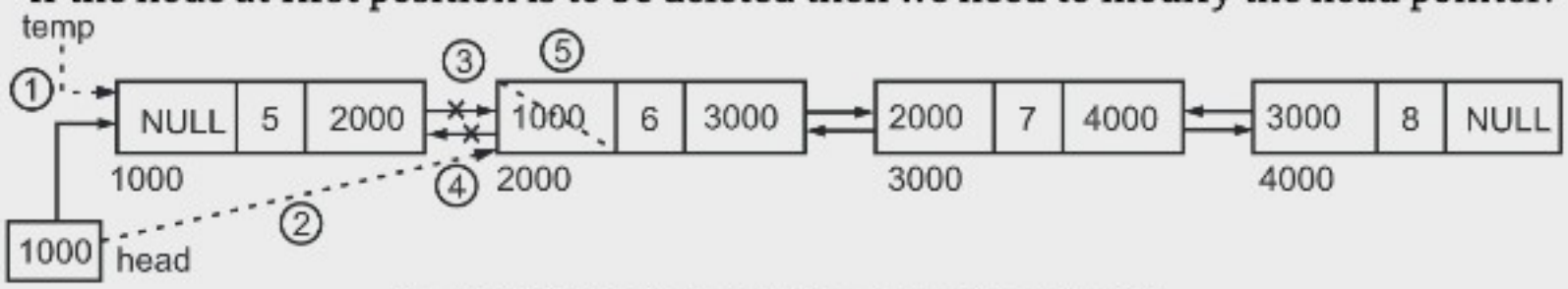


Fig. 3.21: (b) Before deletion from front in DLL

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

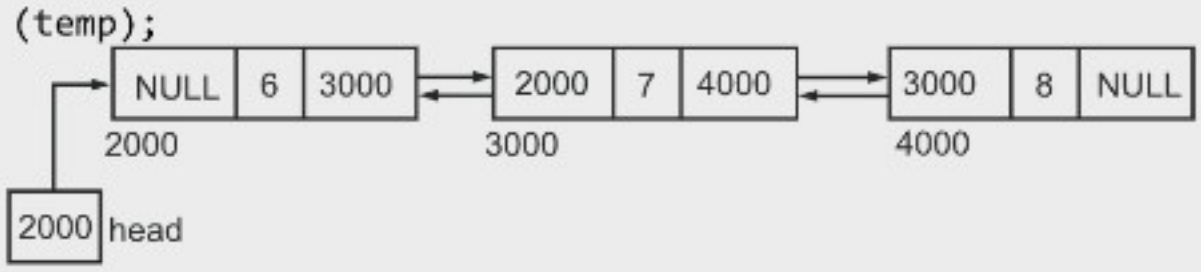


Fig. 3.21: (c) After deletion from front in DLL

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'.

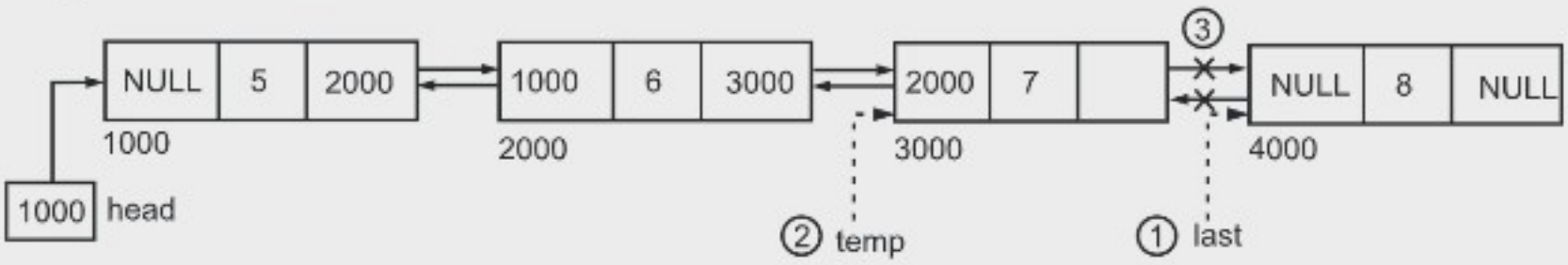


Fig. 3.22 (a): Before deletion at end in DLL

- `Temp -> next = NULL;`
- `free(last);`

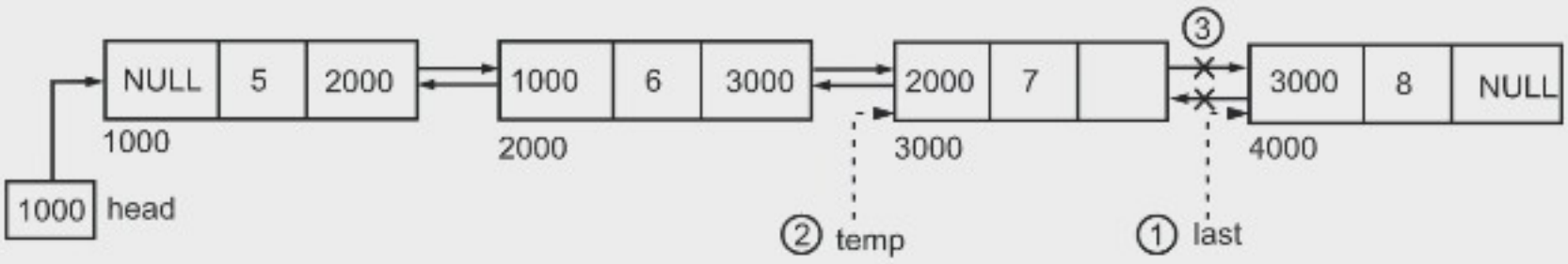


Fig. 3.22 (b): After deletion at end in DLL

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);`

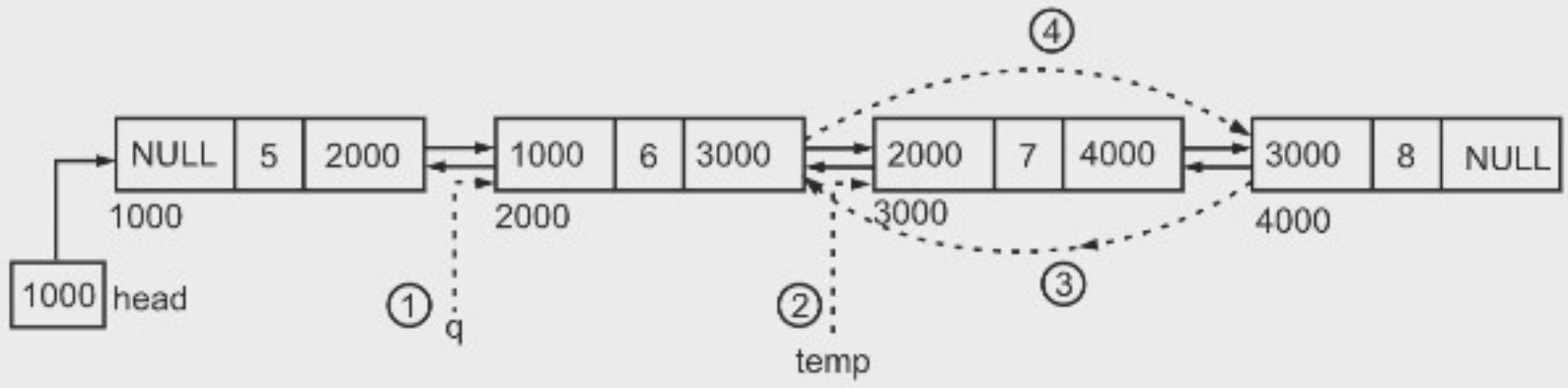


Fig. 3.23: (a) Before Deletion at specified position in DLL

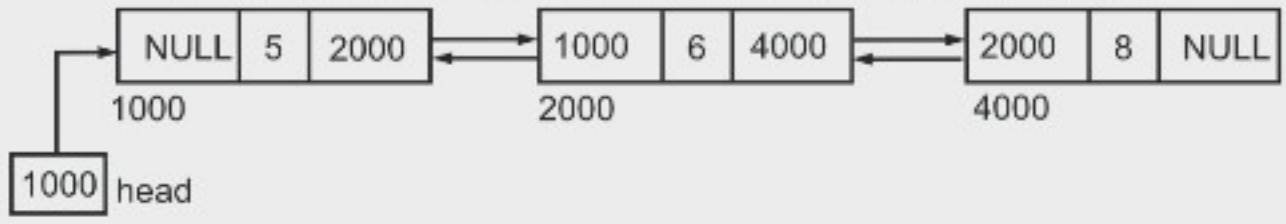


Fig. 3.23: (b) After Deletion at specified position in DLL

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;
}
```

3.6.2.4 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 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 3.2: To implement doubly linked list.

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

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

static int total_nodes;

struct node *create(struct node *head,int n)
{
    int i;
    struct node *newnode=NULL,*last=NULL;

    for(i=1;i<=n;i++)
    {
        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->prev=newnode->next=NULL;

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

```
                last=newnode;
            }
        }
    }
  total_nodes=n;
  return head;
}
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;
    }
}
struct node * 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->prev=newnode->next=NULL;
       if(pos==1)
       {
          if(head==NULL)
             head=newnode;
          else
          {
             newnode->next=head;
             head->prev=newnode;
             head=newnode;
          }
       }
```

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

```
        for(i = 1;i<pos-1; i++)
        q=q->next;
        temp=q->next;
        if(temp->next!=NULL)
           (temp->next)->prev=q;
        q->next=temp->next;
     }
     printf("Deleted element is : %d\n",temp->data);
     temp->prev=temp->next=NULL;
     free(temp);
     --total_nodes;
     return head;
  }
  int main()
  {
     struct node *head=NULL;
     int i,ch,pos,n;
     do
     {
        printf("Enter +ve no of nodes to be create\n");
        scanf("%d",&n);
     }while(n<=0);
     head=create(head,n);
     do
     {
        printf("\n 1-Insert\n 2-Delete\n 3-Display\n");
        printf("\nEnter your choice\n");
        scanf("%d",&ch);
        switch(ch)
        {
           case 1:printf("Enter position no at which you
                                                 want to insert a node\n");
                  scanf("%d",&pos);
                  if(pos>0 && pos<= total_nodes+1)
                     head=insert(head,pos);
                  else
                     printf("Invalid position\n");
                  break;
```

```
            case 2:if(head==NULL)
                      printf("\nLinked List is Empty\n");
                   else
                   {
                      printf("Enter position no from which you want to
                                                        delete a node\n");
                      scanf("%d",&pos);
                      if(pos>0 && pos<= total_nodes)
                         head=delete(head,pos);
                      else
                         printf("Invalid position\n");
                   }
                   break;
            case 3:if(head==NULL)
                      printf("\nLinked List is Empty\n");
                   else
                      display(head);
                   break;
         }
      }while(ch>0 && ch < 4);
    return 0;
   }
```

3.6.3 Differentiate between Singly Linked List & Doubly Linked List

- Following table compares SLL and DLL.

Singly Linked List (SLL)	Doubly Linked List (DLL)
1. Singly linked list allows us to go one way direction.	1. Doubly linked list has two way directions next and previous.
2. Singly linked list uses less memory per node (one pointer) as compare to DLL.	2. Doubly linked list uses more memory per node as compare to SLL.
3. Complexity of Insertion and Deletion at known position is O (n).	3. 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.	4. 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: (a) info, and (b) link.	5. In doubly linked list Each node contains at least three parts: (a) info, (b) link to next node, and (c) link to previous node.
6. Singly linked list is unidirectional i.e., only one direction.	6. It is bidirectional.

3.7 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.

3.7.1 Definition

- A circular list is a linked list in which last node points to the head node.
- A circular list have 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).

3.7.2 Types of Circular Linked List

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.

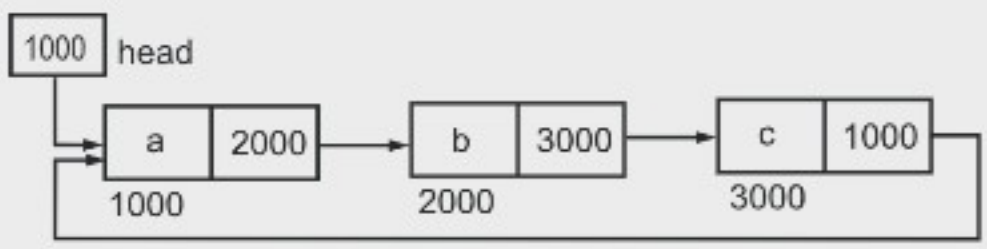


Fig. 3.24: Singly Circular Linked List (CLL)

- 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 contain 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 contain address of the last node in its previous pointer.

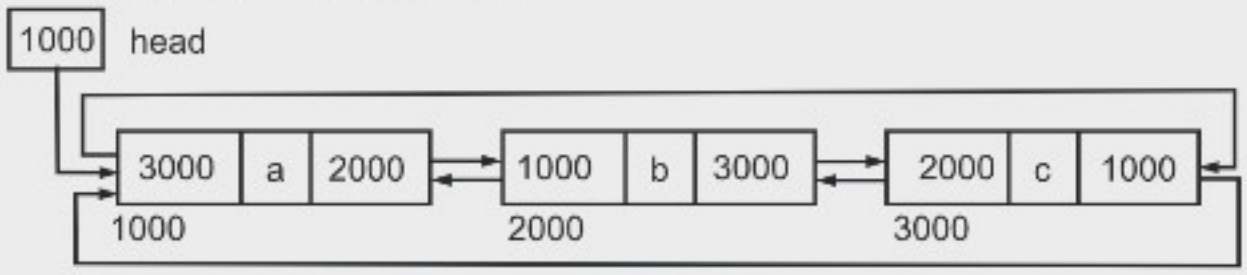


Fig. 3.25: Doubly Circular Linked List

- 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.

3.7.3 Operations on Circular Linked List

3.7.3.1 Circular Singly Linked List

1. Creating a Circular Singly Linked List: **[W-18]**

- 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:

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

Example:

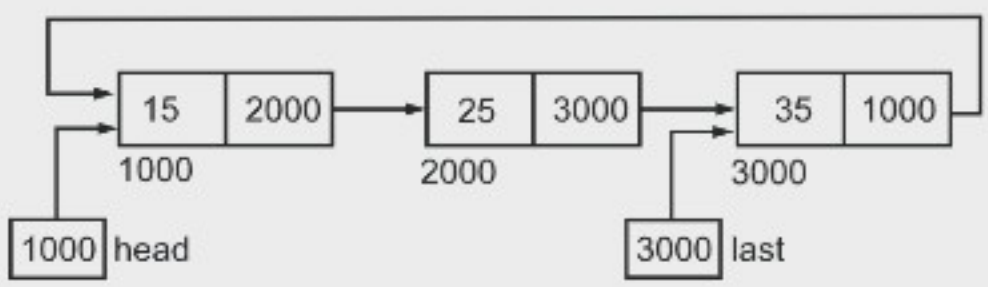


Fig. 3.26 (a): Example of circular singly linked list

2. Insert the element in Circular singly linked list:

- The following figures (a) and (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
```

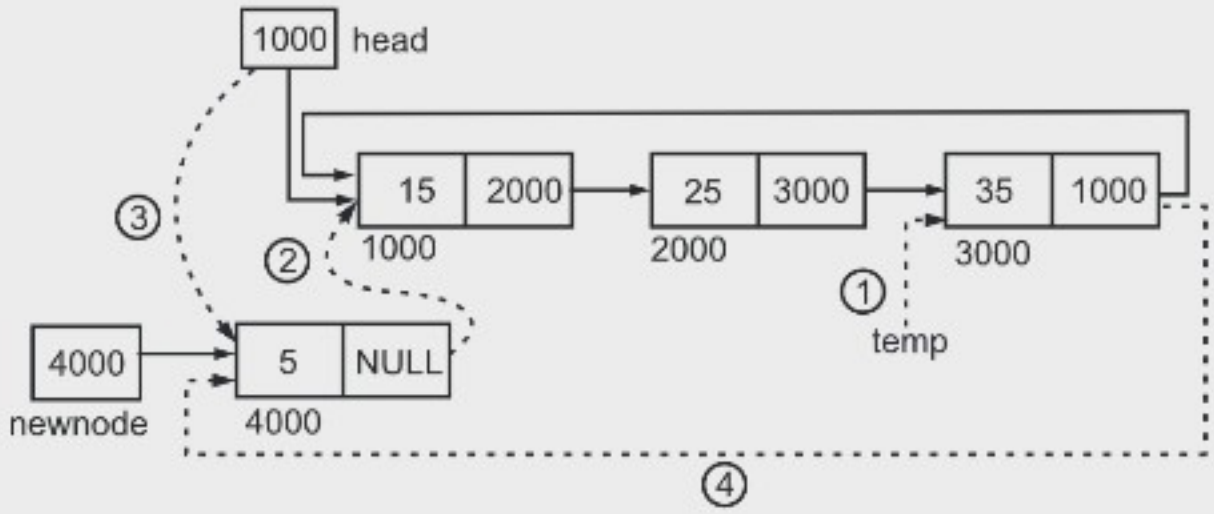


Fig. 3.26 (b): Insertion at front in CSLL

- 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
```

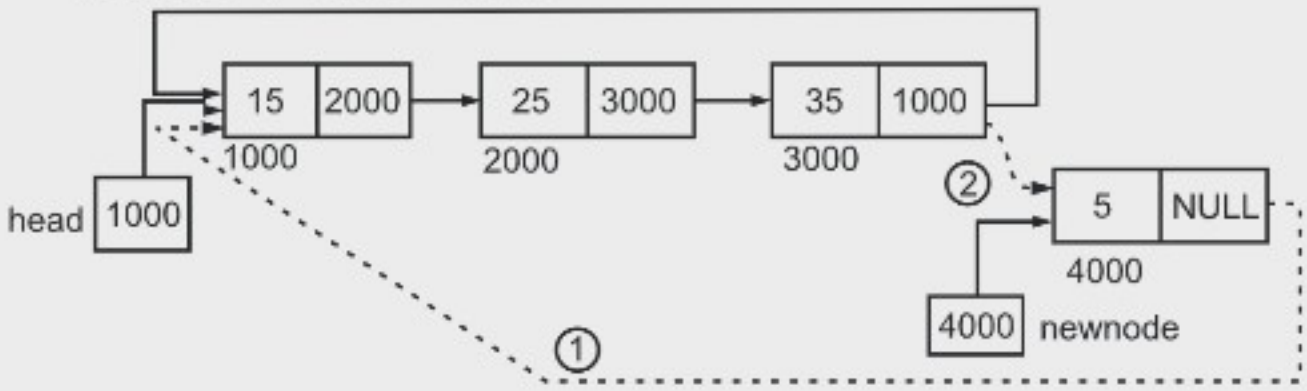


Fig. 3.26 (c): Insertion at end in CSLL

(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...

1. Deleting from Beginning of the list.
2. Deleting from End of the list.
3. Deleting a Specific Node.

(i) 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 following figure show deletion at front in Circular singly Linked List.

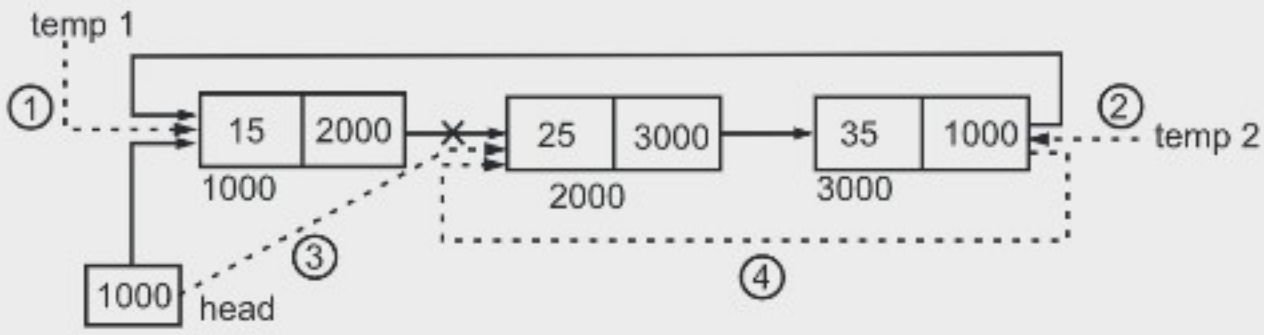


Fig. 3.27 (a): Deletion at beginning in CSLL

- **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;
}
```

(ii) 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.

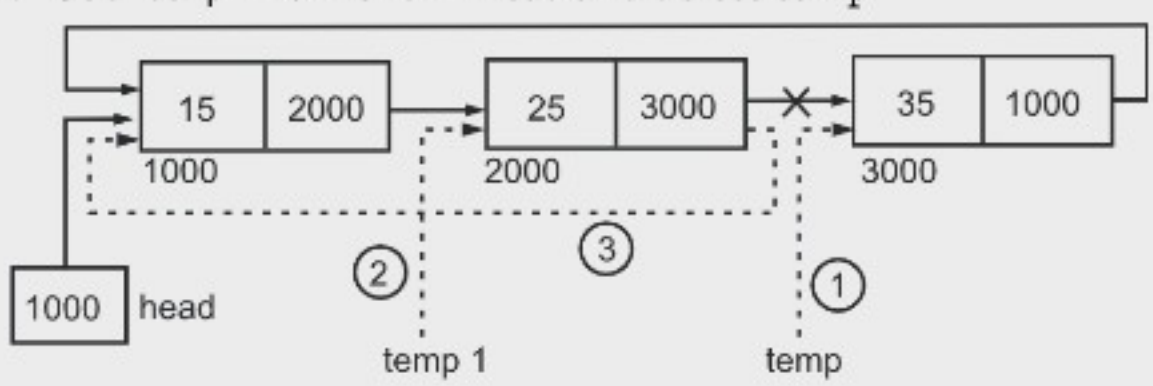


Fig. 3.27 (b): Deletion from end in CSLL

- **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;
}
```

(iii) 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. 3.12 (a) and Fig. 3.12 (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;
}
```

4. Traversing / Display circular singly linked list **[W-18]**

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(" ]");
}
```

3.7.3.2 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: **[W-19]**

(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;`

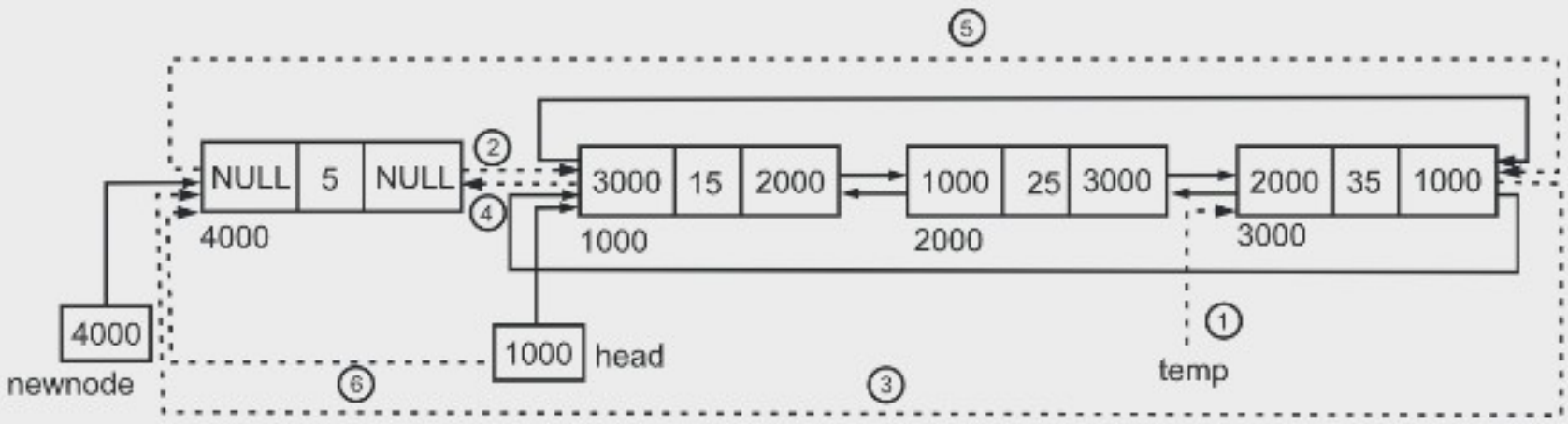


Fig. 3.28 (a): Before insertion at front in CDLL

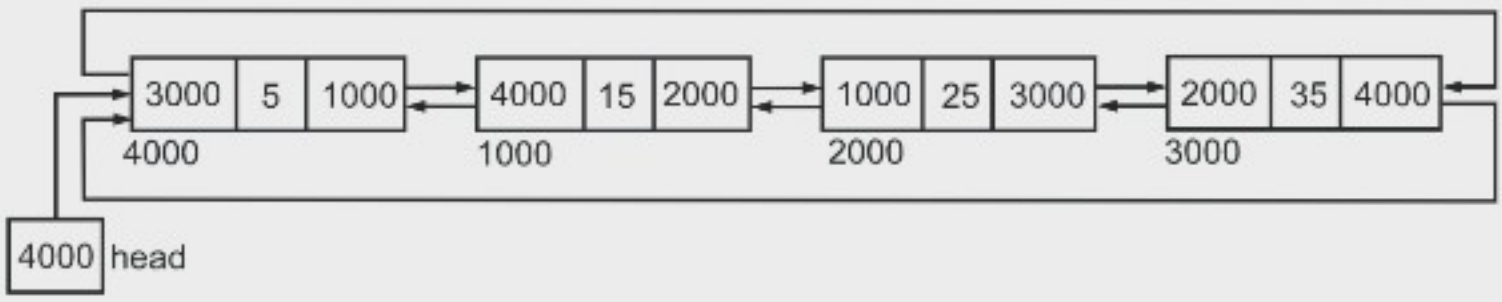


Fig. 3.28 (b): After insertion at front in CDLL

- **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;

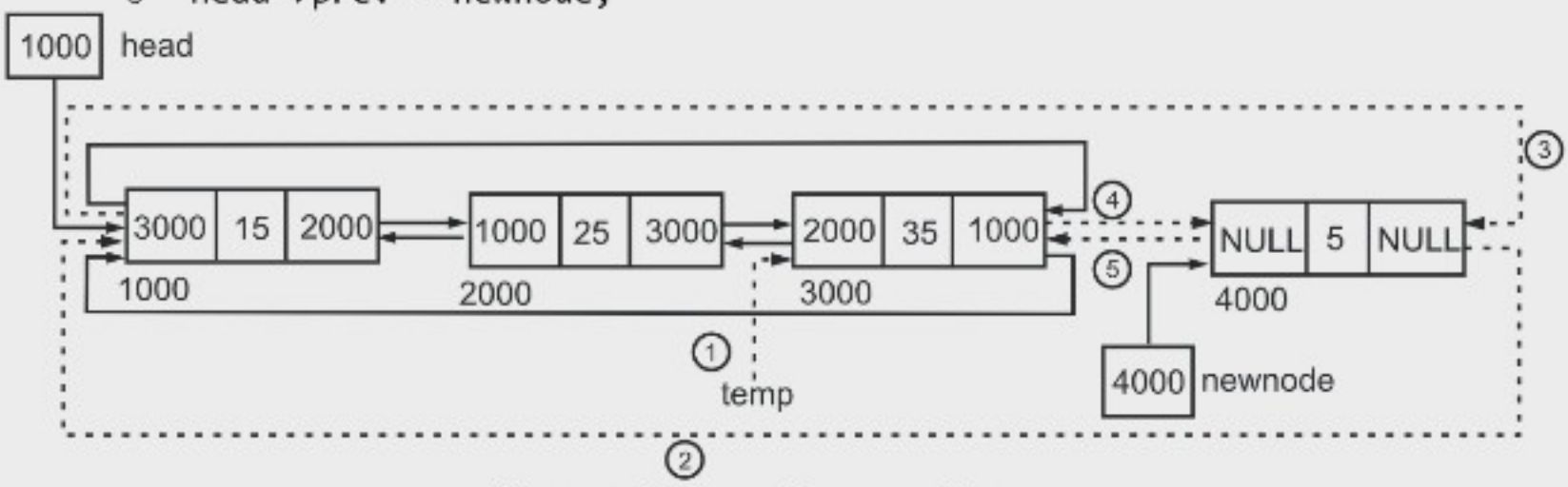


Fig. 3.28 (c): Insertion at end in CDLL

- **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 Fig 3.19(a) and 3.19(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.`

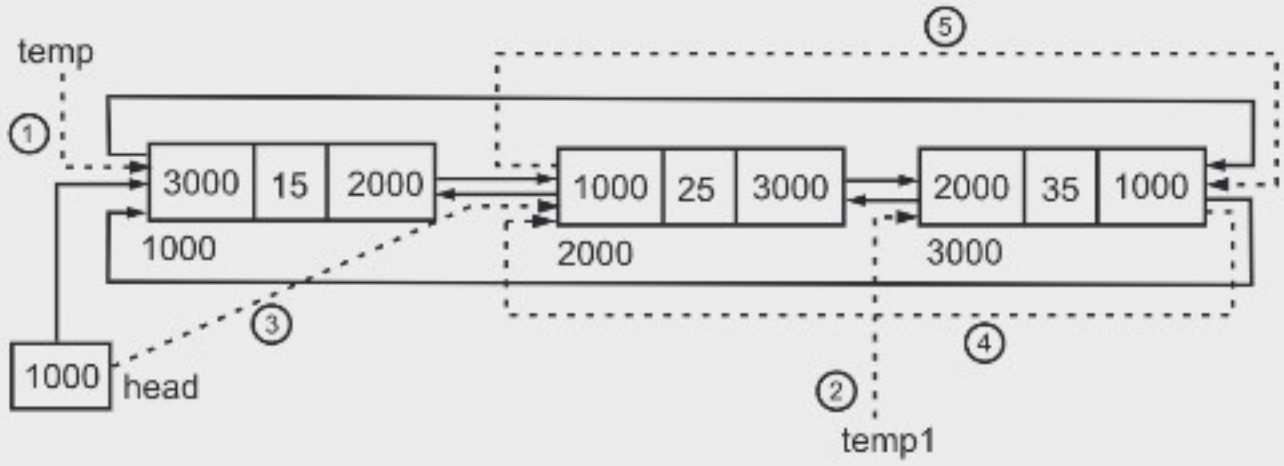


Fig. 3.29 (a): Deletion from beginning in CDLL

- **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.

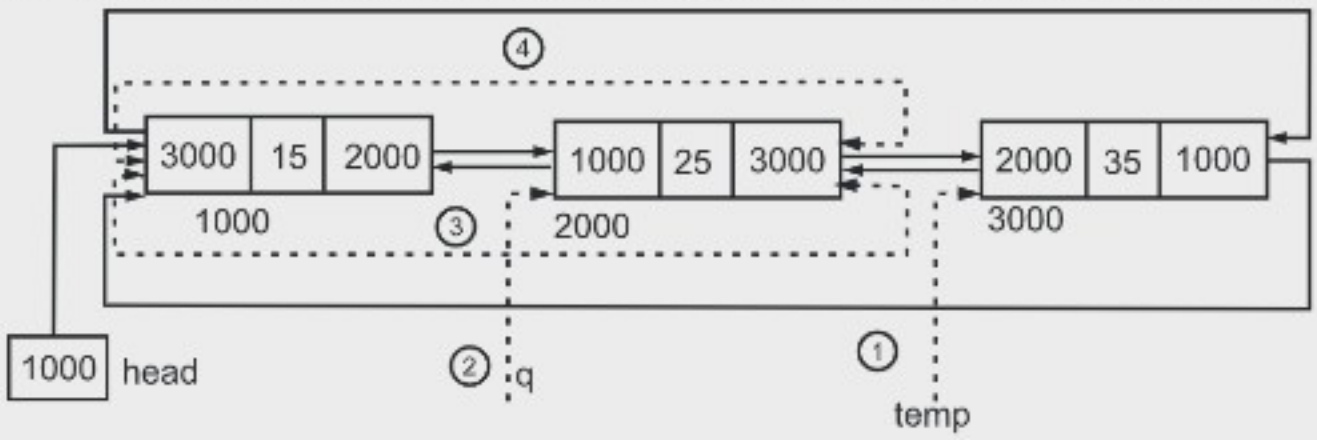


Fig. 3.29 (b): Deletion from end in CDLL

- **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 3.22 (a) and Fig 3.22 (b) for this operation.

3.7.4 Compare Singly, Doubly and Circular Linked Lists

Following table differentiate between singly, doubly and circularly linked lists.

Parameters	Singly	Doubly	Circular
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)

3.8 Real world applications of Link list without implementation [S-18]

- The applications of linked list are as follows:
  - **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.
  - **Previous and next page in web browser:** We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.

- **Music Player:** Songs in music player are linked to previous and next song. You can play songs either from starting or ending of the list.
- **Polynomial addition and multiplication:** Polynomial equations can be added and multiplied easily using linked lists. Linked lists hold each coefficient and exponential part as information parts.
- **Radix sort:** Radix sorts, and address calculation sort use linked lists to perform sorting.
- **To perform arithmetic operations on large numbers:** Using existing data types such as int, long, it is not possible to perform arithmetic operations large numbers. Linked lists solve this problem by splitting them into digits and holding them in nodes.
- **Symbol table implementation:** Symbol tables are used in compiler construction and related fields. These can be used as dictionary for the values. Linked list are best suited for implementing this tables.

Summary

- A linked list is a linear data structure where each element is a separate object.
- Each element (also called as a node) of a list is comprising of two items - the data and a link to the next node.
- A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand.
- Any application which has to deal with an unknown number of objects will need to use a linked list.
- The basic operations perform on the linked list are create list, traverse the list, insertion and deletion of a node.
- Singly linked list are useful data structures, especially if you need to automatically allocate and deallocate space in a list.
- A Doubly Linked List (DLL) contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in singly linked list.
- To make the traversal operation easy, doubly connected linked list are used, in which every node contains linked to its left and right neighbors.
- Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.
- Dynamic memory allocation is used in linked list to manage memory space efficiently.
- The applications of linked list are process queue, garbage collection, polynomial evaluation etc.

Check Your Understanding

1. 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
2. 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.
3. 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.
4. 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)
5. _____ 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
6. What does an empty linked list consist?
   (a) a variable (b) two nodes
   (c) data and a link (d) a null head pointer
7. 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
8. 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.

9. 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)

Answers

1. (c)	2. (c)	3. (a)	4. (a)	5. (c)	6. (d)	7. (c)	8. (b)	9. (b)

Practice Questions

Q.I Answer the following questions in short:

1. Differentiate array and linked list.
2. What is a linked list? How is it represented?
3. How is an element in an array different from the element in a linked list?
4. What are the fields of a node in a doubly linked list?
5. List out operations on linked list.

Q.II Answer the following questions:

1. Which operation do you think is easier, accessing an element to an array, or accessing an element to a linked list? Justify your answer.
2. What are the advantages of using linked lists over arrays?
3. State the disadvantages of static representation of Linked List.
4. Describe different types of linked list.
5. Explain any two operations of linked list.
6. Write a 'C' function to add a node at the beginning of singly linked list.
7. Write a 'C' function to delete an element from singular linked list.
8. Write a 'C' function to delete last node from linked list.
9. Write a 'C' function to delete node from circular linked list at any position.
10. Write a 'C' function to insert a node in circular doubly linked list.
11. Write an algorithm for traversing a linked list and also give its function.
12. List the applications of linked list and explain any one in detail.
13. Write a program that reads the name, age and salary of 10 persons and maintains them in a linked list sorted by name.
14. There are two linked lists A and B containing the following data:

    A : 3, 7, 10, 15, 16, 9, 22, 17, 32

    B : 16, 2, 9, 13, 37, 8, 10, 1, 28

    - A linked list C that contains only those elements that are common in linked lists A and B.
    - A linked list D which contains all elements of A as well as B ensuring that there is no repetition of elements.

Q.III Define the terms:

1. Node
2. Linked List
3. Generalized linked list
4. Singly linked list
5. Doubly linked list

Previous Exam Questions

Summer 2019

1. Explain any two operations on linked list. **[4 M]**

Ans. Refer to Section 3.5.1.

2. Write function to delete an element from singly linked list(SLL). **[4 M]**

Ans. Refer to Section 3.5.1.3.

3. Write a C function to insert node in circular doubly linked list. **[5 M]**

Ans. Refer to Section 3.7.3.2.

4. Write program to add the node at the beginning of singly linked list. **[4 M]**

Ans. Refer to Section 3.5.1.2.

5. Differentiate between array and linked list. **[4 M]**

Ans. Refer to Section 3.1.

Winter 2018

1. In a circular linked list: **[1 M]**

(i) Components are all linked together in some sequential manner.

(ii) There is no beginning and no end.

(iii) Components are arranged hierarchically.

(iv) Forward and backward traversal within the list is permitted.

Ans. (ii)

2. List types of Linked List. **[1 M]**

Ans. Refer to section 3.4.

3. Write a C function for the following: **[5 M]**

(i) To insert a node at Beginning in the doubly linked list.

Ans. Refer to section 3.6.2.2.

4. Write a C Function to create and display Circular Linked list. **[4 M]**

Ans. Refer to section 3.7.3.1.

5. Write a C Function to reverse a singly linked List. **[5 M**

Ans. Refer to section 3.5.1.5.

6. Write an algorithm to search an element from a given singly linked list. **[3 M]**

Ans. Refer to section 3.5.1.8.

Summer 2018

1. An empty linked list consists of _____. **[1 M]**

(i) a variable (ii) data and a link

(iii) two nodes (iv) a null head pointer

Ans. (iv)

2. Define: Linked list **[1 M]**

Ans. Refer to Section 3.2.1

3. Describe different types of linked list. **[5 M]**

Ans. Refer to Section 3.4.

4. Write an algorithm for traversing a linked list and also give its function. **[4 M]**

Ans. Refer to Section 3.5.1.4

5. List the applications of linked list and explain any one. **[3 M]**

Ans. Refer to Section 3.8.

6. State the disadvantages of array representation of linked list. **[3 M]**

Ans. Refer to Section 3.3.1.

■■■

4... Stacks and Queues

Objectives...

- To study basic concepts of Stack and Queue.
- To study Stack Data Type including:
  - Last-In-First-Out (LIFO) Data Access
  - Operations on Stack
- To understand Static and Dynamic implementation of Stack and Queue.
- To learn about Operation on Queue and Stack.
- To understand various types of Queues, Circular Queue and Priority Queue.
- To study different Applications of Stacks and Queue.
- To study Expression Evaluation and conversion (Infix and Postfix).
- To learn about How to Recursion is Handle Using Stack?
- To study the use of stacks in expression conversion, reversing data etc.
- To study the use of queue in various areas.

STACK

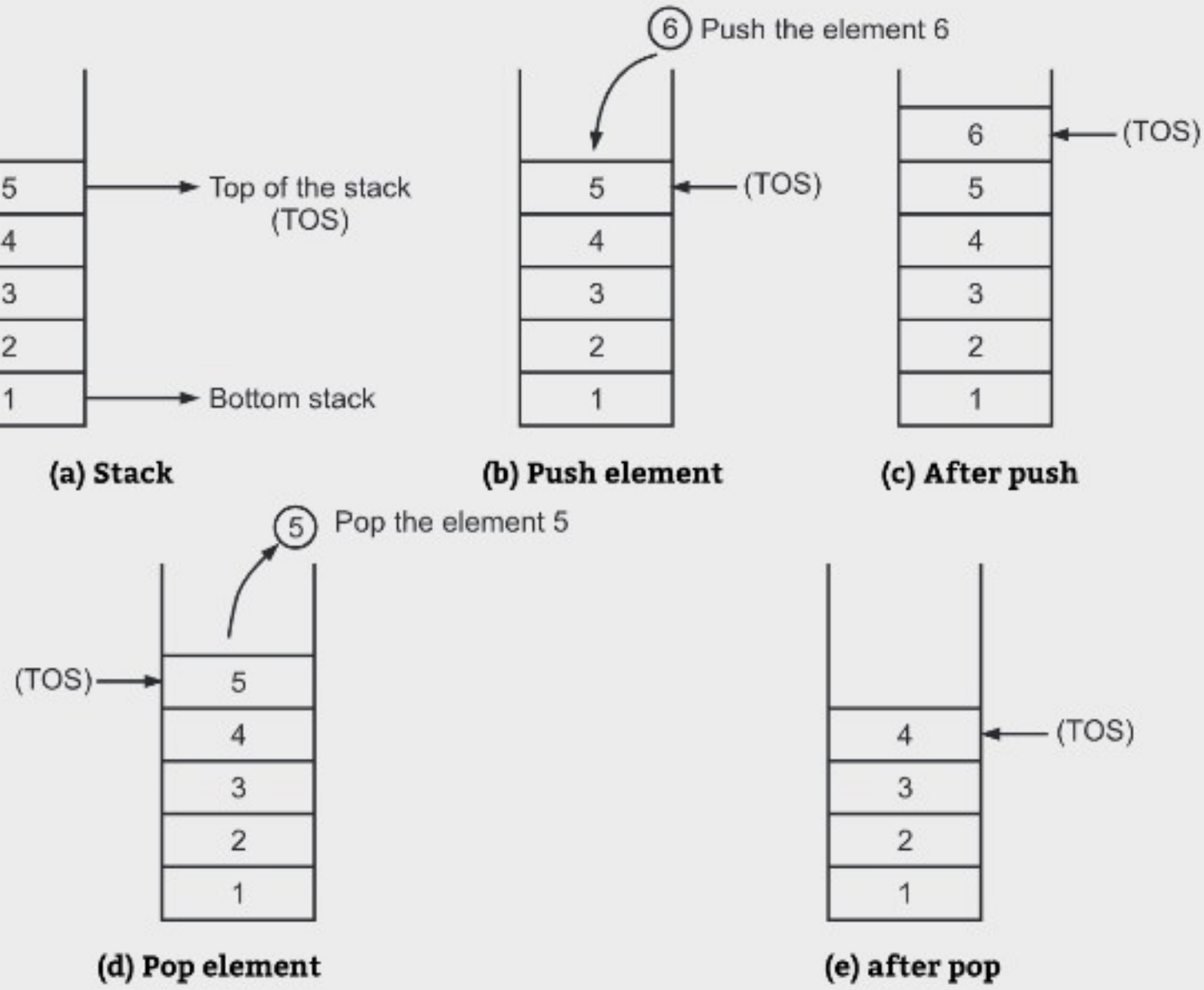
4.1 INTRODUCTION TO STACK

- 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 and Queue are 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.

4.1.1 Definition of Stack

- A stack is an ordered collection of homogeneous data elements 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. 4.1 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.



(a) Stack (b) Push element (c) After push

(d) Pop element (e) after pop

Fig. 4.1: Stack with push and pop operations

4.1.2 Stack Related Terms

1. **Stack:** It is a memory position, which is used for storing the elements in technique LIFO.
2. **Top:** The top pointer points to the top element of stack from where elements are inserted or deleted.
3. **Stack underflow:** When there is no elements in the stack to be delete then stack is underflow i.e. stack is empty.
4. **Stack overflow:** When the stack contains number of elements is equal to its capacity and no more elements can be added, then the stack is overflow i.e stack is full.

4.2 PRIMITIVE OPERATIONS ON STACK [S-19, 18]

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.

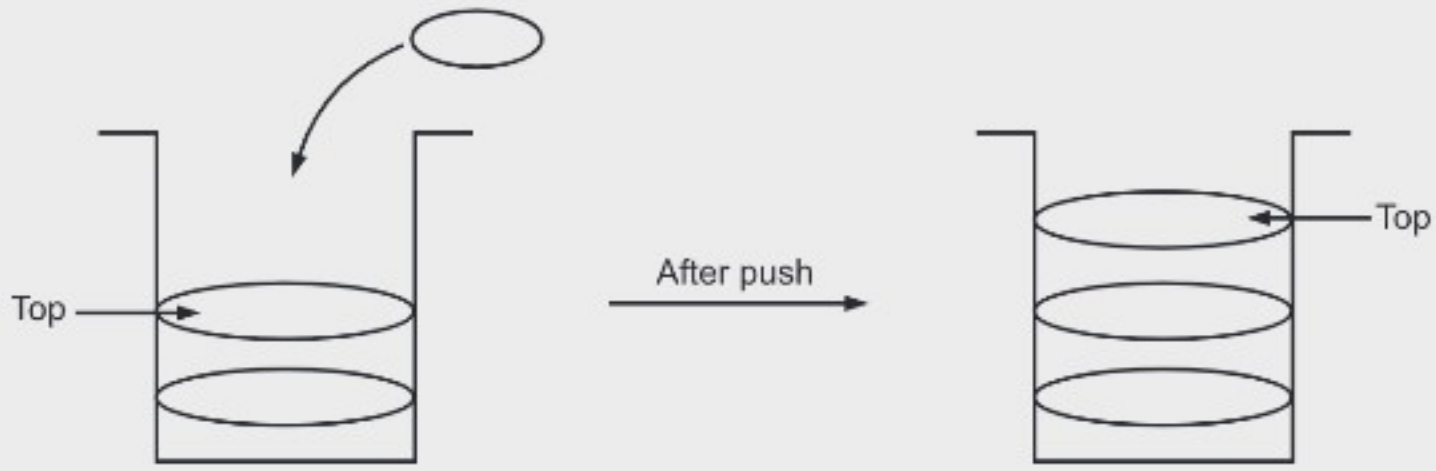


Fig. 4.2: Push Operation

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 an topmost element, the size of stack is reduce by 1 and the element below it becomes the top.

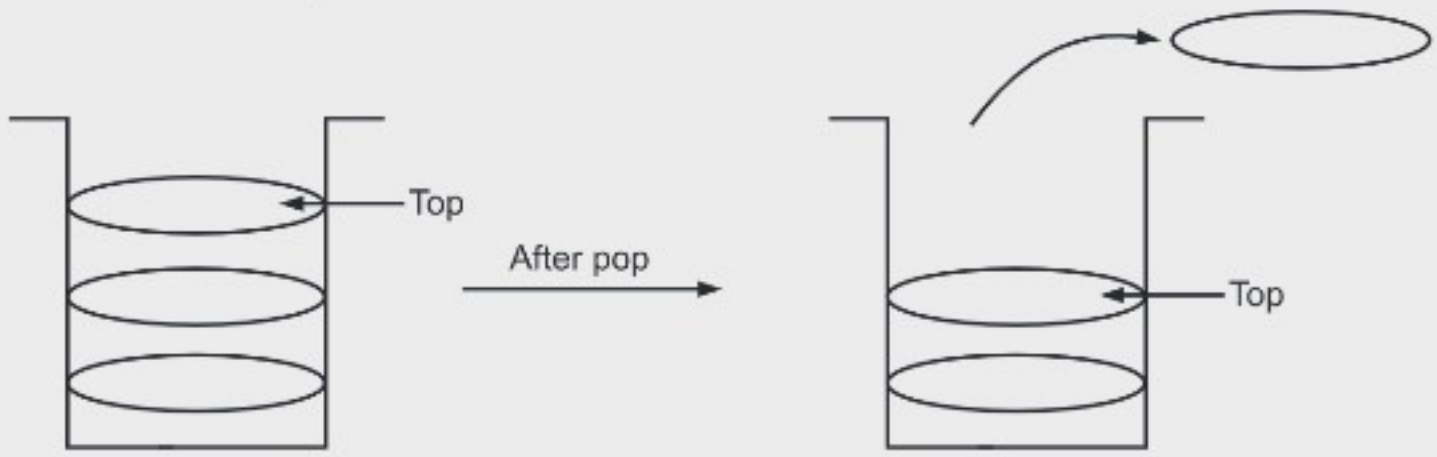


Fig. 4.3: Pop Operation

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.

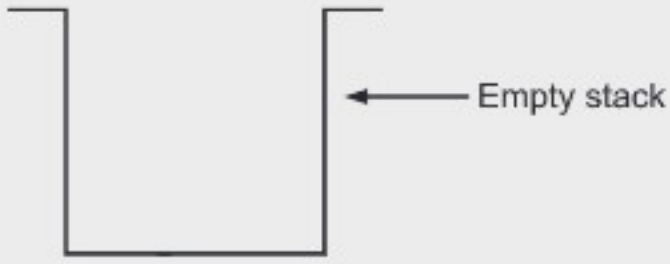


Fig. 4.4: Empty Stack

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.

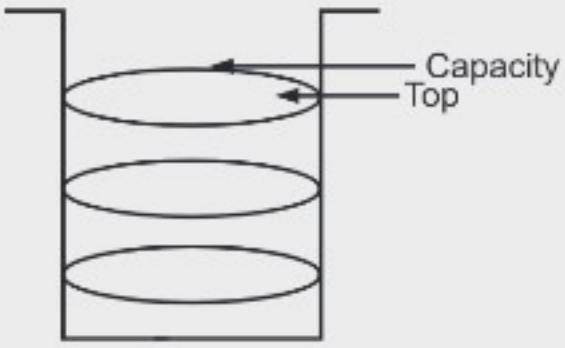


Fig. 4.5: Full Stack

4.3 REPRESENTATIONS OF STACK

- Stack data structure can be implemented using static and dynamic data structure.
- Static implementation of stack (using arrays).
- Dynamic implementation of stack (using linked list).

4.3.1 Static Implementation of Stack (Using Arrays) [S-19, 18]

- 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. 4.6 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.

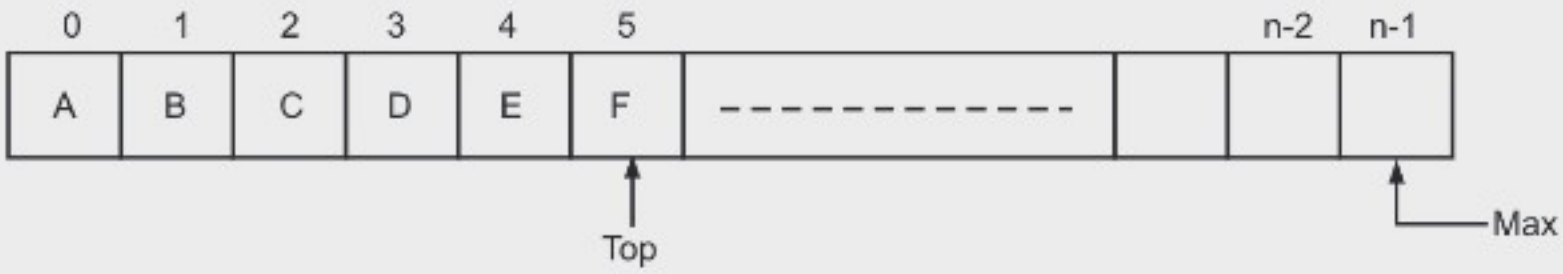


Fig. 4.6: Stack using array

- 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; n^{th} 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.

4.3.1.1 Operations on Stack with respect to Array Implementation

- 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.

The structure of stack is defined as follows:

```
#define MAX 20
struct stack
   {
      int data[max];
      int top;
   };
struct stack s1
```

1. **init():** To initialize stack S.

- In this function top is initialized to -1 to indicate the stack is empty.

```
s1.top = – 1
```

Here we created stack named s1 in which we can insert at the most 10 integer values.

2. **IsEmpty ():**

- 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) includes:

```
if (s1.top == –1)
      return 1;
else
      return 0;
```

3. IsFull ():

- 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) includes:

```
if (s1.top == size-1)
      return 1;
else
      return 0;
```

4. Push an element (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 (s1.top == n - 1)
      printf ("stack overflow \n");
else
{
      s1.top++;                          } ⇒ s1.data[++s1.top] = element;
      s1.data[s1.top] = element;
}
```

5. Pop ():

- 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 (s1.top != -1)
      return(s1.data[s1.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.

Example 1: 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);
```

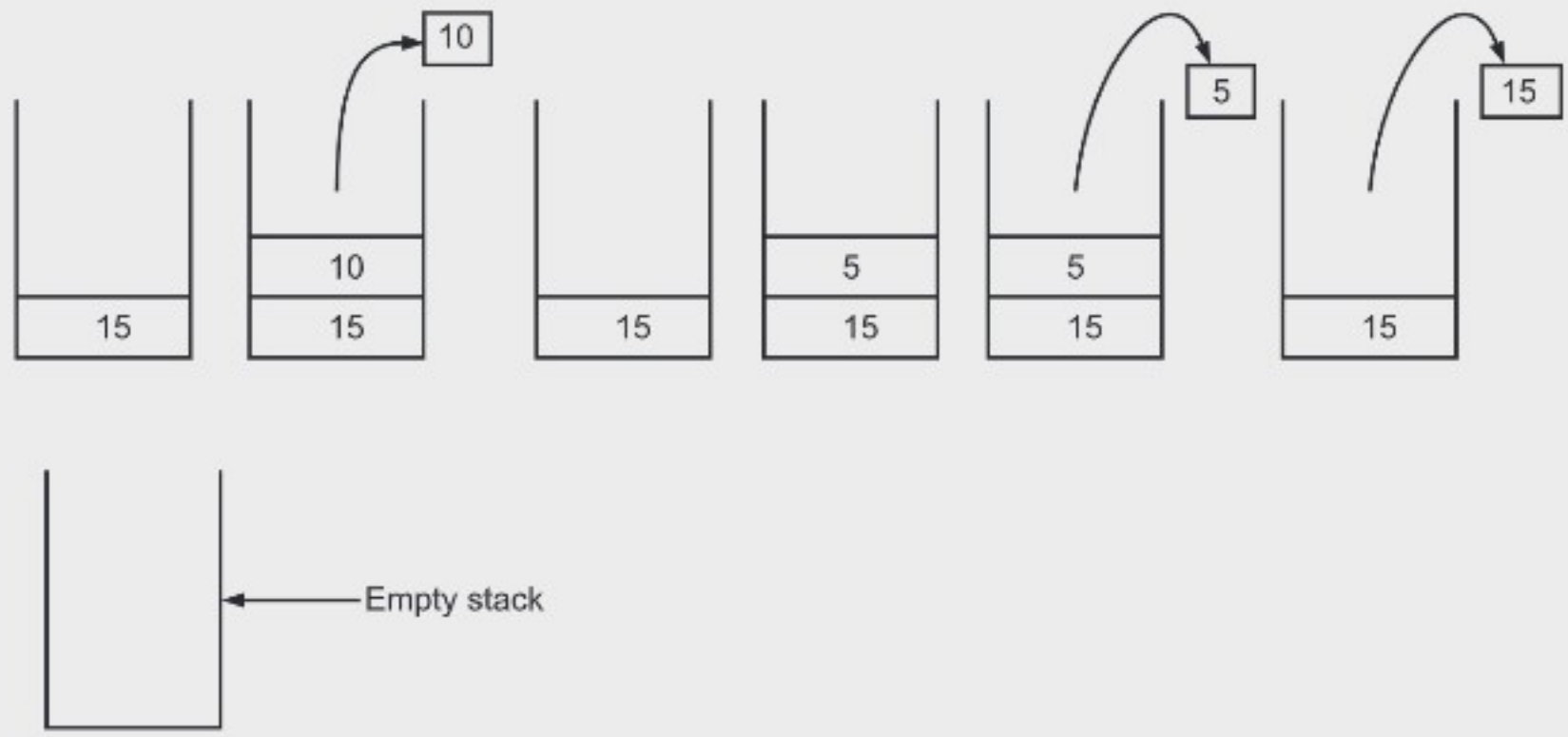


Fig. 4.7: Operation on stack

Program 4.1: Implementation of Static stack.

```
#include<stdio.h>
#define MAX 20
   struct stack
   {
      int data[MAX];
      int top;
   };
/* function for initialize a stack */
   void init(struct stack *s)
   {
      s -> top = -1;
   }
/* function for checking stack is empty or not.*/
   int isEmpty(struct stack *s)
   {
      if(s->top==-1)
       return 1;
      else
       return 0;
   }
/* function for checking stack is full or not.*/
   int isFull(struct stack *s)
   {
      if(s->top==MAX-1)
```

```
        return 1;
       else
        return 0;
    }
/* add elements to the stack */
    void push(struct stack *s, int item)
    {
       s -> data[++s->top]=item;
    }
/* pop item from the stack */
    int pop(struct stack *s)
    {
       return (s -> data[s -> top--]);
    }
/* Display stack */
     void dispItems(struct stack *s)
     {
         int i;
         if(s->top == -1)
         {
             printf("STACK IS EMPTY."); return;
         }
         for(i=s->top;i>=0;i--)
             printf("%d->",s->data[i]);
         printf("\n");
     }
    void main()
    {
       struct stack s1;     /* structure stack variable s1 is created */
       int ch,x;
       init(&s1);
       do
       {
       printf("1-Push\n2-Pop\n");
       printf("Enter your choice\n");
       scanf("%d",&ch);
```

```
        switch(ch)
        {
        case 1: if(isFull(&s1))
                   printf("Stack is Full\n");
                else
                {
                   printf("Enter element to be push into stack\n");
                   scanf("%d",&x);
                   push(&s1,x);
                   printf("\nSTACK ELEMENTS:");
                   dispItems(&s1);
                }
                break;
        case 2: if(isEmpty(&s1))
                   printf("Stack is Empty\n");
                else
                   printf("Popped element is: %d\n", pop(&s1));
                   printf("\nSTACK ELEMENTS:");
                   dispItems(&s1);
                   break;
           }
      }while(ch>0 && ch<3);
   }
```

4.3.2 Dynamic implementation of Stack (using Linked List)

- 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 allows 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. 4.8 shows the pictorial representation of stack in linked list containing three elements (C,B,A).

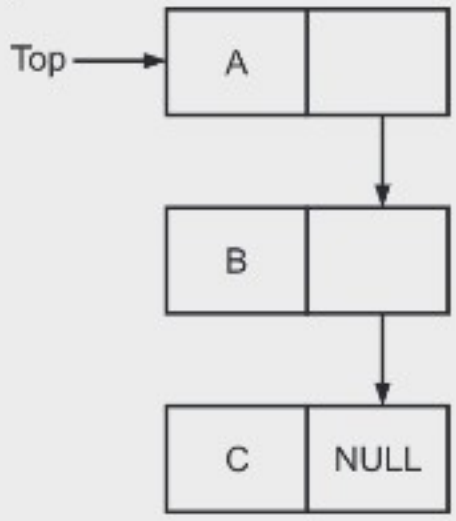


Fig. 4.8: Stack as a linked list data structure

The structure defined to represent dynamic stack:

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

4.3.2.1 Operations on Stack with Respect to Linked list Representation

1. **struct node *S,*Top=NULL;**
2. **push (S, 15)**

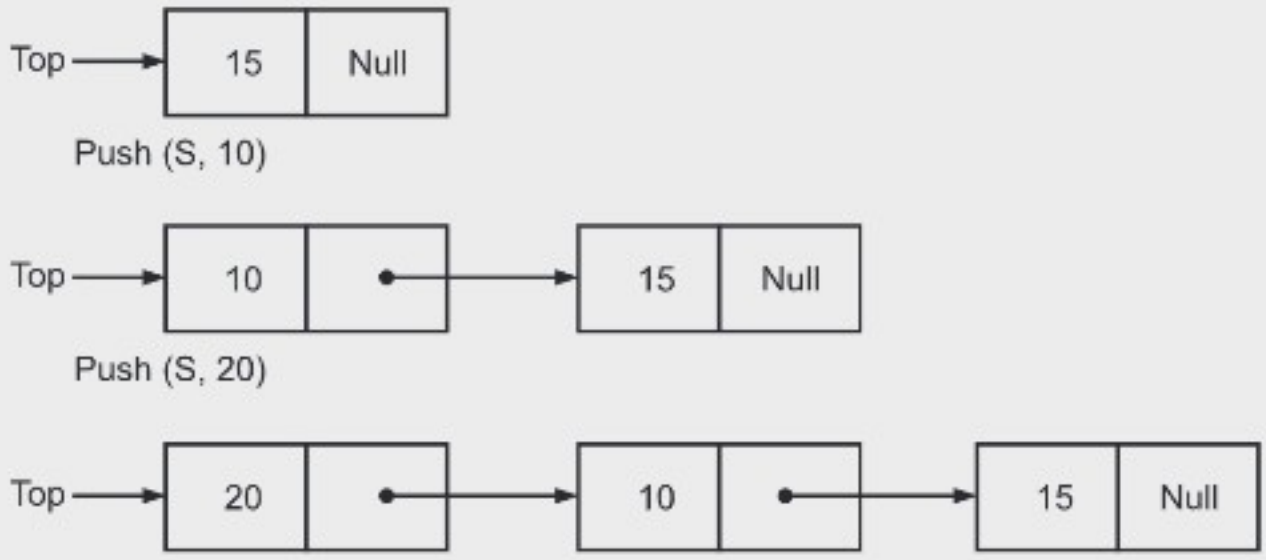


Fig. 4.9: Push operation on stack

Here, stack grows at the 'top' end and the element is always pushed at top.

- **The Push() function can be written as follows:**

```
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).

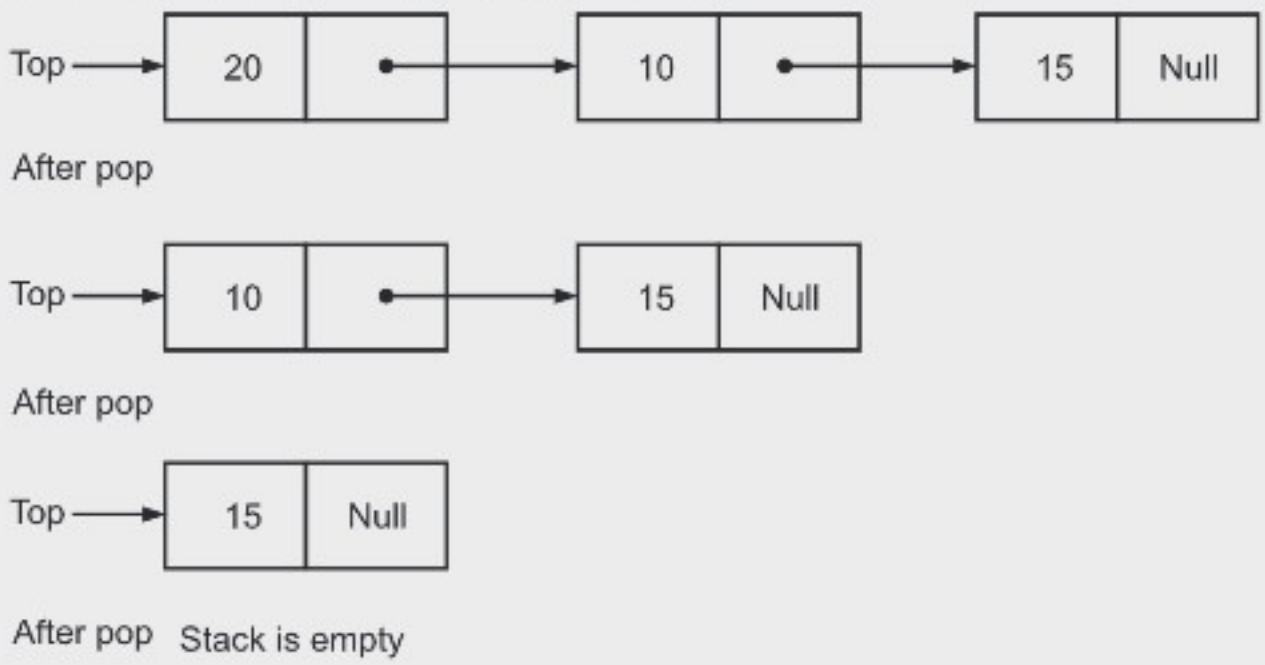


Fig. 4.10: Pop operation of stack

- 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.
- **The Pop() function can be written as follows:**

```
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;
    }
```

Program 4.2: Implementation of Dynamic stack (using linked list).

```
#include<stdio.h>
#include<conio.h>
struct Node
{
    int data;
    struct Node *next;
}*top;
void push(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if(top == NULL)
        top=newNode;
    else
    {
        newNode->next = top;
        top = newNode;
    }
    printf("\nInsertion is Success!!!\n");
}
```

```
void pop()
{
   if(top == NULL)
      printf("\nStack is Empty!!!\n");
   else
   {
      struct Node *temp = top;
      top = top->next;
      printf("\nDeleted element: %d", temp->data);
      free(temp);
   }
}
void display()
{
   if(top == NULL)
      printf("\nStack is Empty!!!\n");
   else
   {
      struct Node *temp = top;
      while(temp != NULL)
      {
       printf("%d--->",temp->data);
       temp = temp -> next;
      }
      printf("--->NULL");
   }
}
void main()
{
   int choice, value;
   printf("\n:: Stack using Linked List::\n");
```

```
    while(1)
    {
       printf("\n****** MENU ******\n");
       printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
       printf("Enter your choice: ");
       scanf("%d",&choice);
       switch(choice)
       {
        case 1: printf("Enter the value to be insert: ");
                scanf("%d", &value);
                push(value);
                break;
        case 2: pop(); break;
        case 3: display(); break;
        case 4: exit(0);
        default: printf("\nWrong selection!!! Please try again!!!\n");
       }
    }
}
```

4.4 APPLICATIONS OF STACKS [S-19, W-18]

- 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.
  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 is find the solution for Knight Tour problem or N-Queen Problem etc.

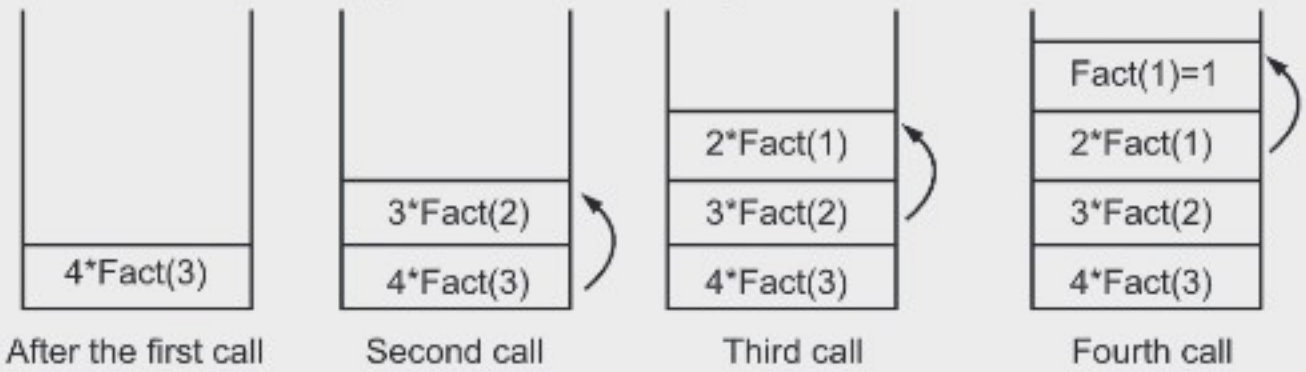
7. **Function Call**: Stack is used to keep information about the active functions or subroutines.

4.4.1 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 is 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.

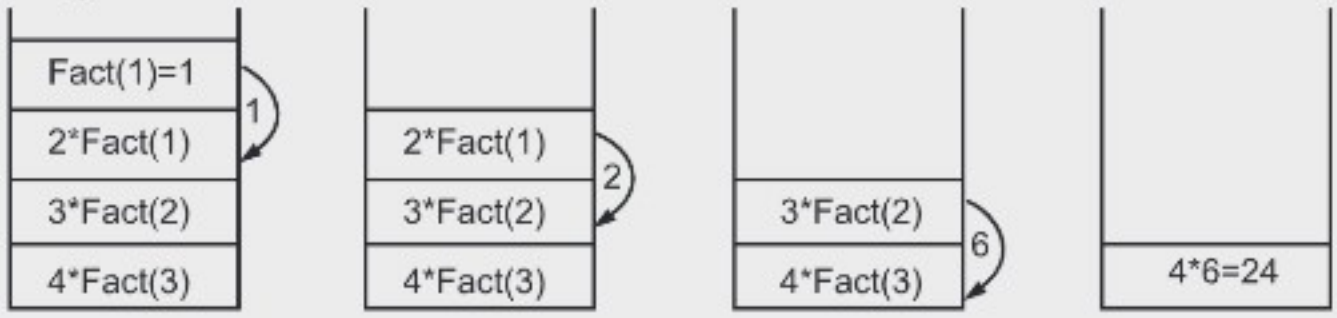


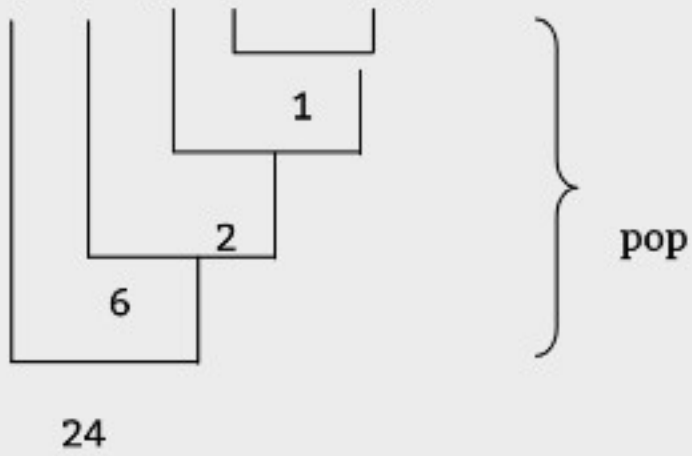
Fig. 4.11: Factorial of number using recursion

- 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 maintain.

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.

4.4.2 Expression Conversion and Evaluation

- Infix, Prefix and Postfix notations are introduced by Jan Lukasiewicz. We consider simple arithmetic expressions with binary operators: +, -, *, /, ^ (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: **[W-18, S-18]**

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 is:

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.

4.4.2.1 Algorithm to Convert an INFIX Expression to a POSTFIX Expression [S-19]

Input: Infix expression.

Output: Postfix expression.

Step 1 : Read infix expression from left to right and character by character.

Step 2 : For each character 'ch' in the input string:

If ch is an operand then output ch.

Step 3 : If ch is an operator then following action should be taken-

- Check whether there is any operator already present in stack or not.
- If stack is empty then push operator onto stack.
- If present then check whether priority of ch is greater than priority of topmost stack operator.

- If priority of ch is greater then push ch onto stack.
- Else pop and print operator until one of lower priority operator is encountered or the stack becomes empty and finally push ch onto stack.

Step 4 : If ch is a left parentheses push ch into the stack.

Step 5 : If ch is a right parentheses Pop and output tokens until a left parentheses is popped (but do not output left parenthesis)

Example 2: For better understanding, let us consider following example P/Q – R * S + T convert it into postfix. **[W-18]**

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 4.3: 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/+
```

4.4.2.2 Algorithm to Convert POSTFIX Expression to an INFIX Expression

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 4.4: 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
```

4.4.3 Algorithm for Evaluation of Postfix Expression

- Algorithm for Evaluation of Postfix Expression.
- Read postfix expression character by character from left to right.

Step 1 : If (read character = operand) then push the element in the stack

Step 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.

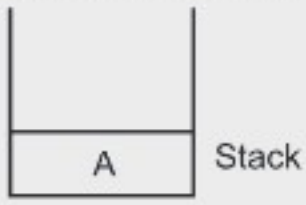
Step 3 : If string is ended, pop the result.

Example 4: Evaluate the postfix expression. **[S-18]**

AB + C – (Let A = 3, B = 4, C = 2)

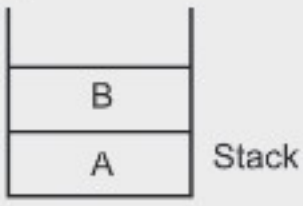
Solution: Read given postfix string from left to right.

1. First element is operand A, push A onto stack.



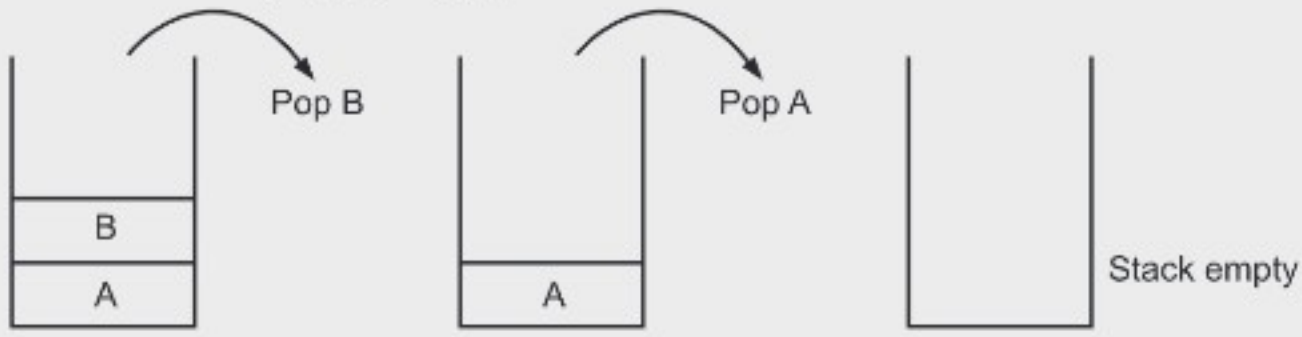
(a)

2. Second element is operand B, push B onto the stack.



(b)

3. The third element is operator '+', pop 2 elements from stack (i.e. A and B) and then evaluate them by applying operator ' + '.



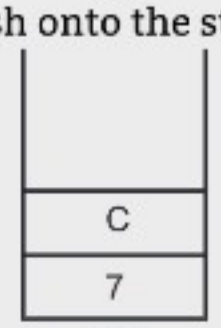
(c)

Evaluating, A + B = 3 + 4 = 7.

4. Push the result onto the stack.

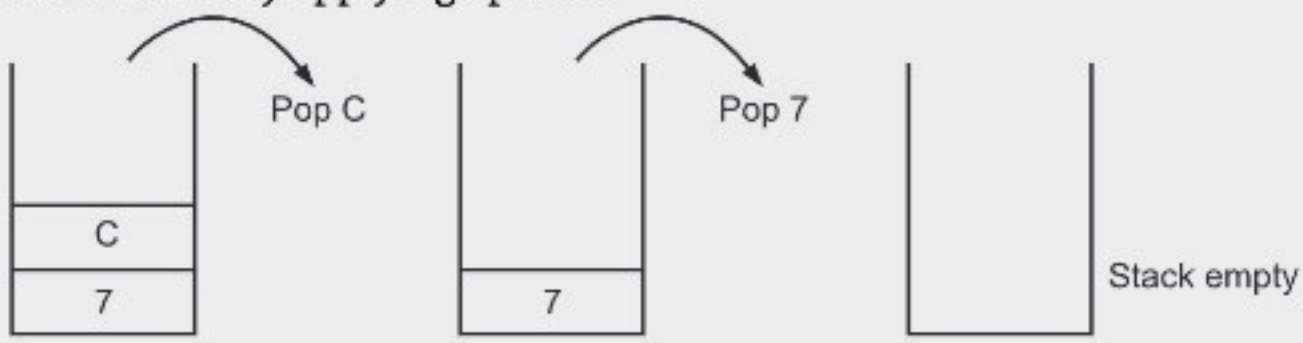
(d)

5. Next element in operand C, push onto the stack.



(e)

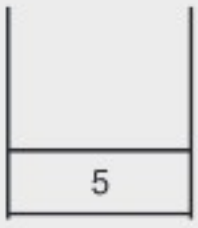
6. Next element is operator ' - ', pop 2 elements from stack (i.e. C and 7) and then evaluate them by applying operator ' - '.



(f)

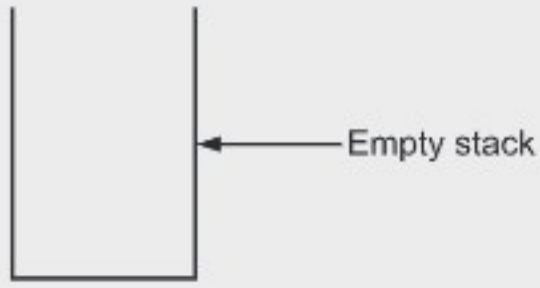
Evaluating = 7 – C = 7 – 2 = 5.

7. Push result 5 onto the stack.



(g)

8. End of expression, therefore pop result from stack.



(h)

Result of given postfix expression is: 5

Fig. 4.12: Evaluation of postfix string

Program 4.5: Program to evaluate a postfix expression.

```
#include<stdio.h>
#define MAX 30
   struct stack
   {
      int data[MAX];
      int top;
   };
/* initialization of stack */
   void inits(struct stack * s)
   {
      s->top = -1;
   }
/*insert element onto the stack */
   void push (struct stack * s, int num)
   {
      s->data[++s->top] = num;
   }
/* delete element from the stack */
   int pop (struct stack * s)
   {
      return (s->data[s->top--]);
   }
/* To evaluate postfix expression */
   void evaluate(struct stack *s, char op, int num1, int num2)
   {
      int ans;
      switch(op)
      {
         case '+'  :  ans = num1 + num2;
                      break;
```

```
            case '-'  :   ans = num1 - num2;
                          break;
            case '*'  :   ans=num1 * num2;
                          break;
            case '/'  :   ans = num1/num2;
                          break;
            case '^'  :
            case '$'  :   ans = pow(num1,num2);
                          break;
         }
         push (s, ans);
      }
   /* function to check whether the input character is operator or not */
      int gettype (char ch)
      {
         switch(ch)
         {
            case '+':
            case '-':
            case '*':
            case '/':
            case '$':
            case '^': return 1;
            default: return 0;
         }
      }
      void main()
      {
         char postfix[MAX];
         int num1, num2, element, i, ans;
         struct stack s;
         inits(&s);
         printf("\n Enter postfix expression");
         scanf("%s", postfix);
         for(i=0; postfix[i]!='\0'; i++)
         {
            switch(gettype (postfix[i]))
            {
               case 0: push(&s,postfix[i]-'0');
                       break;
```

```
                case 1: num2 = pop(&s);
                        num1 = pop(&s);
                        evaluate(&s, postfix[i],num1,num2);
            }
          }
      printf("The answer of %s=%d", postfix, s.data[s.top]);
    }
```

Output:

```
Enter postfix expression 528+*
The answer of 528+*=50
```

QUEUE

4.5 INTRODUCTION TO QUEUE

- 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 your 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 multiuser system, there is queue of process (jobs) waiting to get CPU.

4.5.1 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. The first element in a queue will be removed first from the queue i.e. data are processed through the queue in the order in which they are received. Therefore, queue are called First-In-First-Out (FIFO).

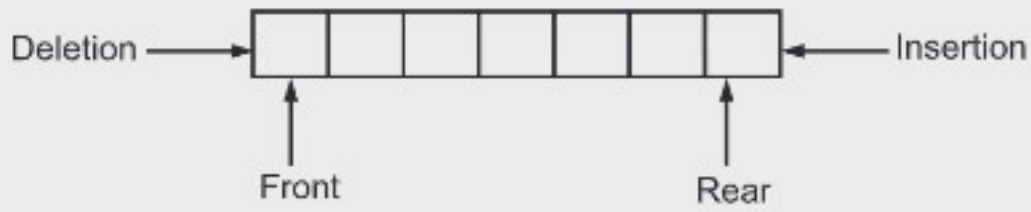


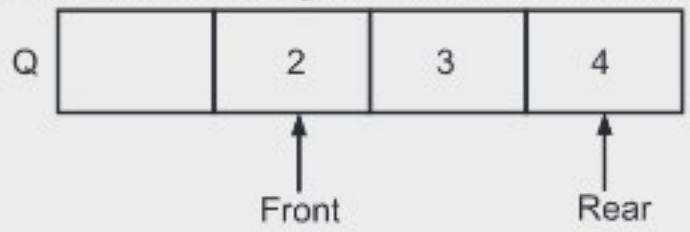
Fig. 4.13: A Queue

4.5.2 Example of Queue

- Consider a queue of four elements we can represent the queue as shown in Fig. 4.14 (a).

(a)

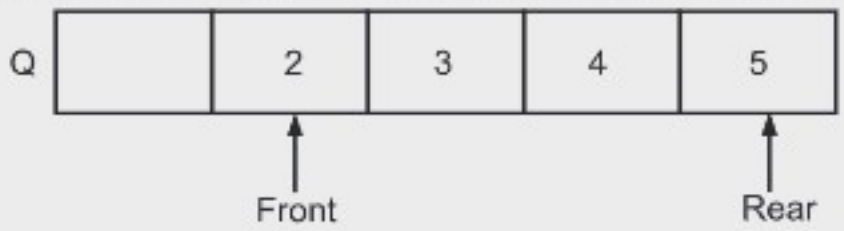
- If we want to delete the element from queue it is deleted only at front end; we get



(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. 4.14 (c).



(c)

Fig. 4.14: Queue with operations

4.6 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. **Insert (Q, x):** It adds the element x to the rear end of queue Q and returns the resulting queue.
  3. **Delete (Q):** It deletes the element from the front end of queue and returns the resulting queue.
  4. **Isfull (Q):** It returns true if queue is full otherwise returns false.
  5. **IsEmpty (Q):** It returns true if queue is empty otherwise returns false.
  6. **peek(Q):** It is often used to return the value of first element without deleting it.

4.7 REPRESENTATION OF QUEUE

- There are two ways to represent a queue in memory:
  1. Static implementation of Queue(using an array).
  2. Dynamic implementation of Queue (using linked list).

4.7.1 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 queue may expand or shrink dynamically. There is no limitation on the number of items in the queue.

- The Fig. 4.15 shows the representation of a queue as an array:

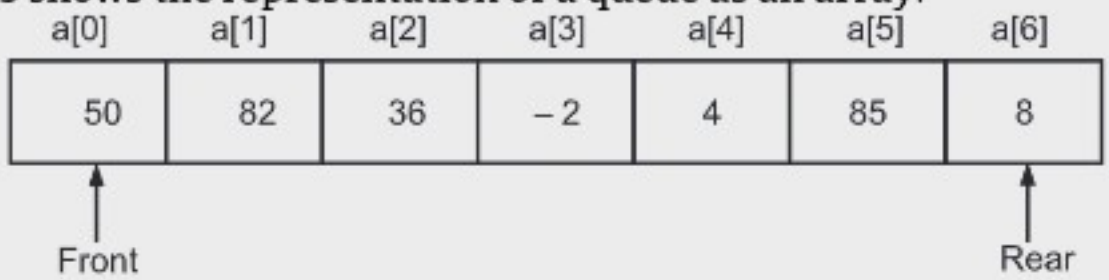


Fig. 4.15: Representation of a queue as an array

- For implementation of queue as an array we required a one-dimensional array, say data and two variables front and rear.
- **The Structure of Queue is as follows:**

```
#define max 20
struct queue
{
    int a[max];
    int front, rear;
};
```

4.7.1.1 Various Operations on Queue (Using Array)

1. **Create (Q):** This operation create an empty queue. Front and Rear will indicates empty queue. Hence, as in C language array index ranges between 0 to (max-1), front and rear are initialize to –1. **[S-18]**

   C function for create queue:

```
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. If front = rear then IsEmpty(Q) returns true otherwise returns false. **[S-18]**

   C function to check queue is empty or not:

```
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. **[S-18]**

C function to check queue is full or not:

```
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). **[S-18]**

   C function to insert an element in queue:

```
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.

   C function to delete an element from queue:

```
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 4.6: 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;
   }
```

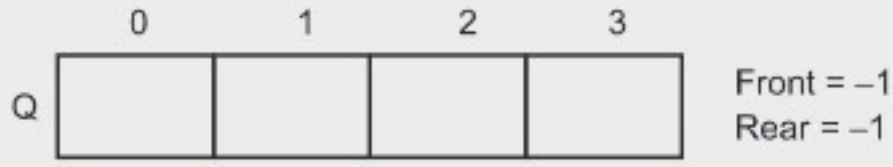
```
int isEmptyQ(struct queue *q)
{
   if(q->front == q->rear)
      return 1;
   else
      return 0;
}
int isFullQ(struct queue *q)
{
      if (q -> rear == MAX - 1)
         return 1;
      else
         return 0;
}
void insertQ(struct queue *q, int x)
{
   q ->data[++(q -> rear)] = x;

   }
   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);
}
```

Example 5: Let us consider the example which uses the all above functions.

Let Q be an empty queue with front = rear = – 1.

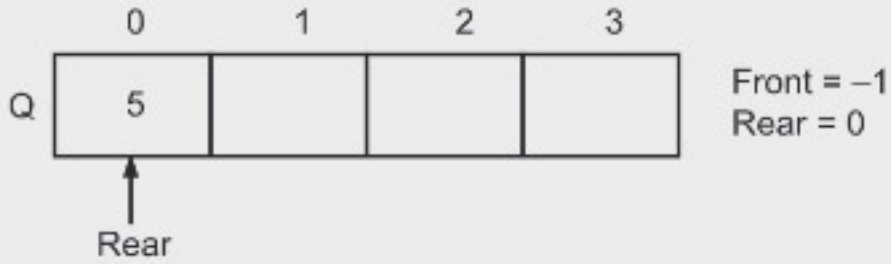
Let max = 4



(a)

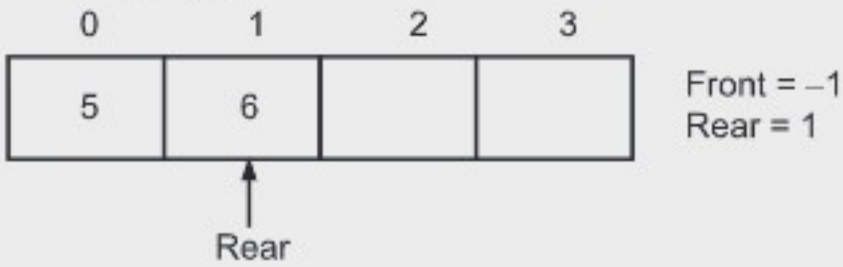
Consider the following statements:

(i) insertQ(Q,5)



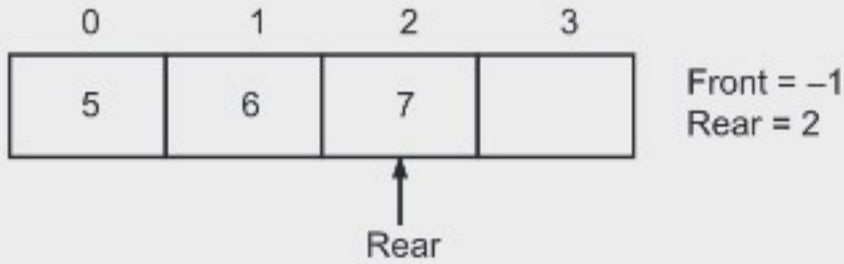
(b)

(ii) insertQ(Q,6)



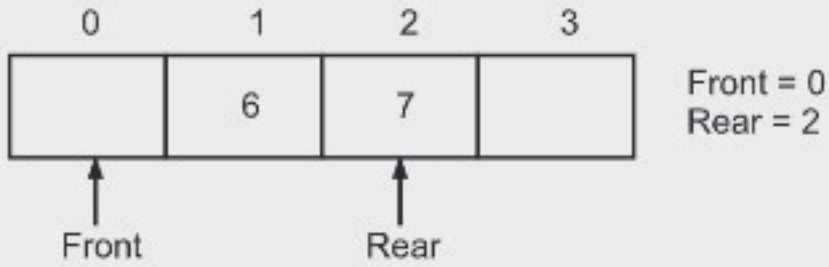
(c)

(iii) insertQ(Q,7)



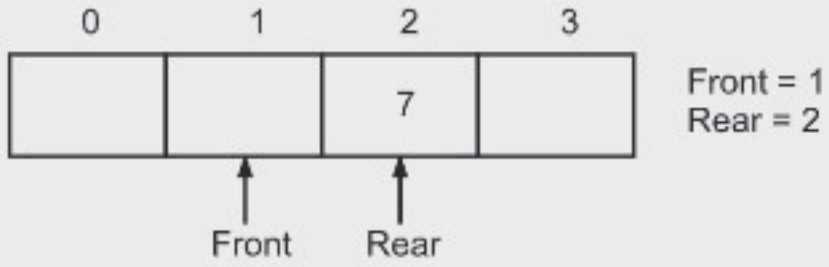
(d)

(iv) x = deleteQ(Q)



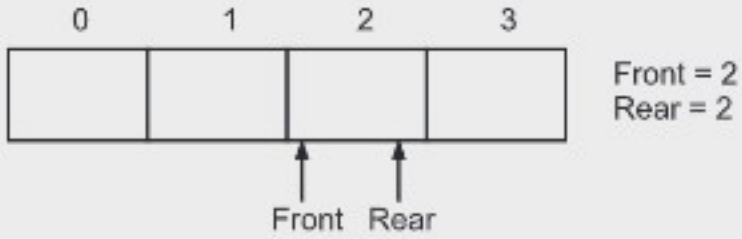
(e)

(v) x = deleteQ(Q)



(f)

(vi) x = deleteQ(Q)

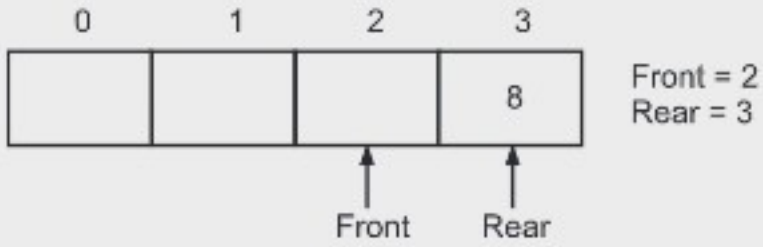


(g)

(vii) x = deleteQ(Q)

This generates "Empty Queue" error as front = rear = 2.

(viii) insertQ(Q,8)



(h)

Fig. 4.16: Array representation of queue of above example

(ix) insertQ(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.

4.7.2 Dynamic Implementation of Queue

- We have discuss queue are represent using sequential organization i.e. array, but this representation have drawbacks.

Disadvantages of Array Representation:

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. 4.17 shows the linked queue.

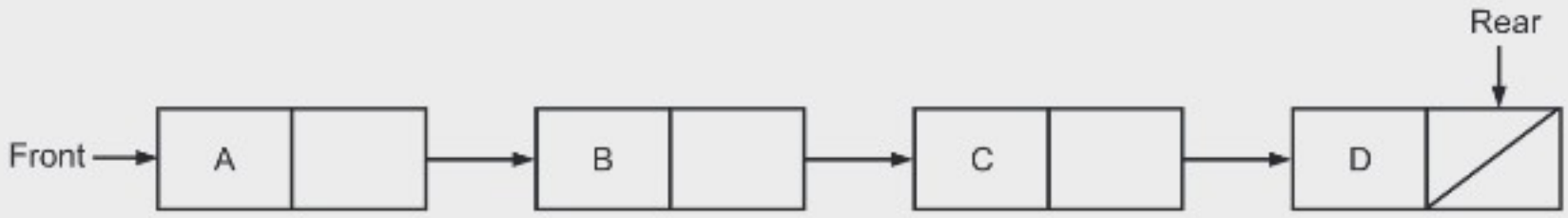


Fig. 4.17: Linked queue

Data Structure for Queue using Linked List:

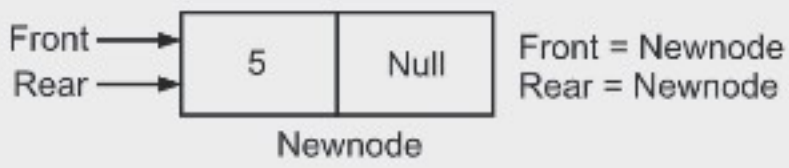
```
struct queue
{
    int data;
    struct queue *next;
} *front, *rear;
```

4.7.2.1 The various Operations on Queue (Using Linked List)

(i) Initialization of queue.

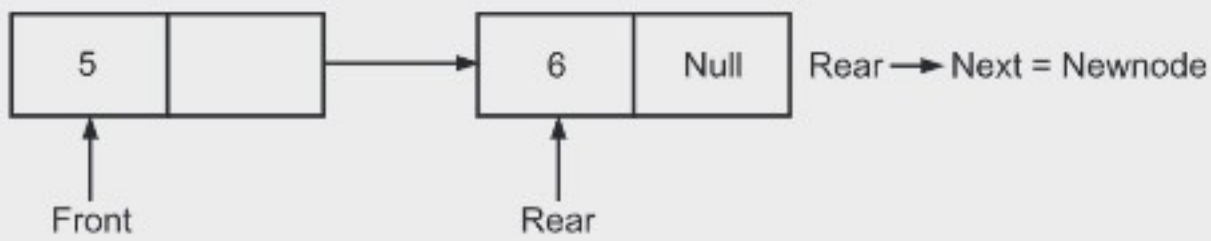
front = rear = NULL indicates List is null

(ii) Insert 5



(a)

(iii) Insert 6



(b)

Fig. 4.18: Insert operation of queue

C Function for insert element in Queue: [W-18]

```
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;
    }
}
```

(iv) delete

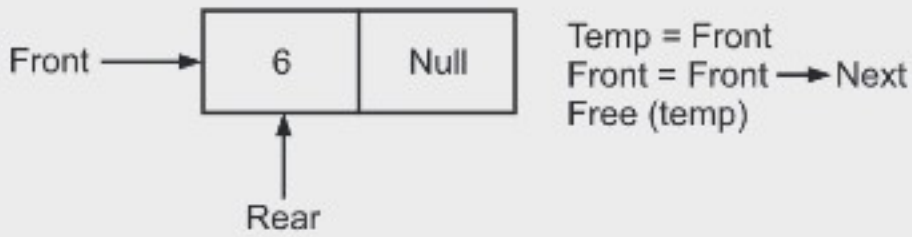


Fig. 4.19: Delete element from queue

(v) delete

front = rear = null

C Function for delete element in Queue: **[W-18]**

```
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);
}
```

Program 4.7: 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);
    }
```

4.8 TYPES OF QUEUE

1. **Simple Queue:** As is clear from the name itself, simple queue lets us perform the operations simply. i.e., the insertion and deletions are performed likewise.
2. **Circular Queue:** Unlike the simple queues, in a circular queue each node is connected to the next node in sequence but the last node's pointer is also connected to the first node's address. Hence, the last node and the first node also gets connected making a circular link overall.
3. **Priority Queue:** Priority queue makes data retrieval possible only through a predetermined priority number assigned to the data items.

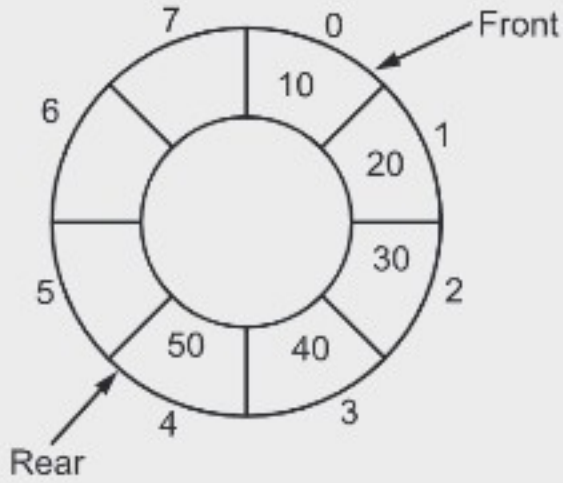
   While the deletion is performed in accordance to priority number (the data item with highest priority is removed first), insertion is performed only in the order.

4.8.1 Circular Queue

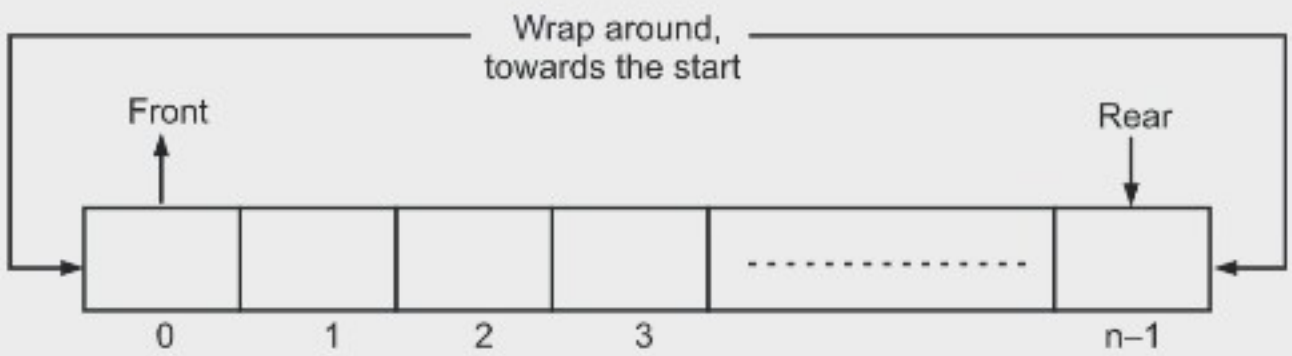
- So far we have discussed linear queues. Linear queue using array has certain drawbacks. There is possibility that the queue is reported full even though slots of the queue are empty i.e. actually queue is not full.

4.8.1.1 Definition [W-18, S-18]

- 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. The Fig. 4.9 shows the pictorial representation of a circular queue.



(a) Logical view



(b) Physical view

Fig. 4.20: 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:

- **Memory management:** Circular queue is used in memory management.
- **Process Scheduling:** A CPU uses a queue to schedule processes.
- **Traffic Systems:** Queues are also used in traffic systems.

4.8.1.2 Operations of Circular Queue

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.

4.8.1.3 Advantages of Circular Queue [S-19, S-18]

- Advantages of circular queue are listed below:
  - 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.
  - It avoids shifting of queue elements.
  - Circular queues are used in many applications such as memory management, CPU scheduling, traffic system.

Note: *Refer insert, del and display functions in following program for insert, delete, display operations of circular queue.*

Program 4.8: Program that implements circular queue as an array.

```
# include<stdio.h>
# define MAX 5

int cqueue_arr[MAX];
int front = -1;
int rear = -1;
```

```
/*Begin of insert*/
void insert(int item)
{
   if((front == 0 && rear == MAX-1) || (front == rear+1))
   {
      printf("Queue Overflow \n");
      return;
   }
   if (front == -1)  /*If queue is empty */
   {
      front = 0;
      rear = 0;
   }
   else
   {
      if(rear == MAX-1)      /*rear is at last position of queue */
         rear = 0;
      else
         rear = rear+1;
   }
   cqueue_arr[rear] = item ;
}
/*End of insert*/
/*Begin of del*/
void del()
{
   if (front == -1)
   {
      printf("Queue Underflow\n");
      return ;
   }
   printf("Element deleted from queue is: %d\n",cqueue_arr[front]);
   if(front == rear) /* queue has only one element */
   {
      front = -1;
      rear=-1;
   }
```

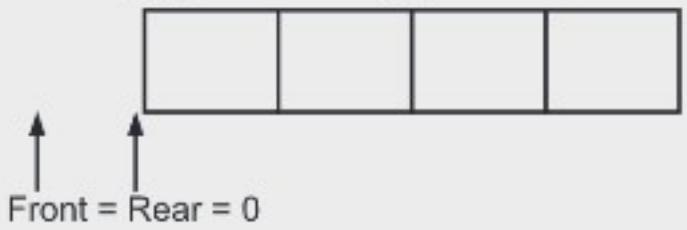
```
    else
    {
        if(front == MAX-1)
            front = 0;
        else
            front = front+1;
    }
}
/*End of del() */

/*Begin of display*/
void display()
{
    int front_pos = front,rear_pos = rear;
    if(front == -1)
    {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements:\n");
    if( front_pos <= rear_pos )
        while(front_pos <= rear_pos)
        {
            printf("%d ",cqueue_arr[front_pos]);
            front_pos++;
        }
    else
    {
        while(front_pos <= MAX-1)
        {
            printf("%d ",cqueue_arr[front_pos]);
            front_pos++;
        }
        front_pos = 0;
```

```
        while(front_pos <= rear_pos)
        {
            printf("%d ",cqueue_arr[front_pos]);
            front_pos++;
        }
    }
    printf("\n");
}
/*End of display*/

/*Begin of main*/
int main()
{
    int choice,item;
    do
    {
        printf("1.Insert\n");
        printf("2.Delete\n");
        printf("3.Display\n");
        printf("4.Quit\n");
        printf("Enter your choice: ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                printf("Input the element for insertion in queue: ");
                scanf("%d", &item);
                insert(item);
                break;
            case 2:
                del();
                break;
            case 3:
                display();
                break;
```

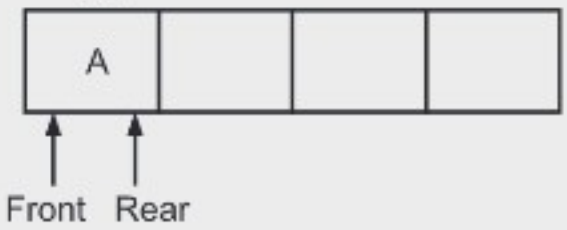
```
            case 4:
                break;
                default:
                printf("Wrong choice\n");
        }
    }while(choice!=4);
    return 0;
}
/*End of main*/
```

Example 6: Trace the insertion and deletion operations on circular queue where max = 4.

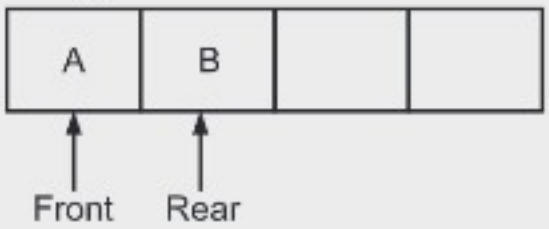
1. Initially queue is empty



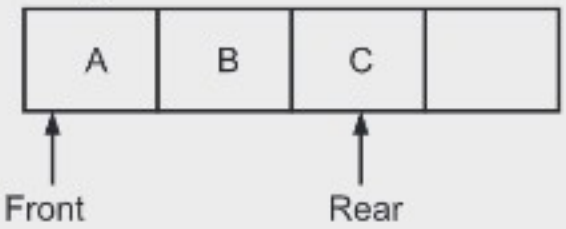
2. add(A)



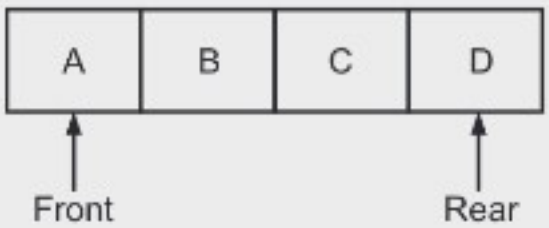
3. add(B)



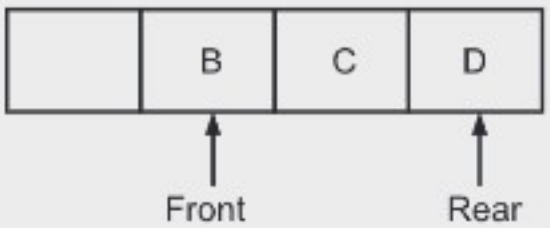
4. add(C)



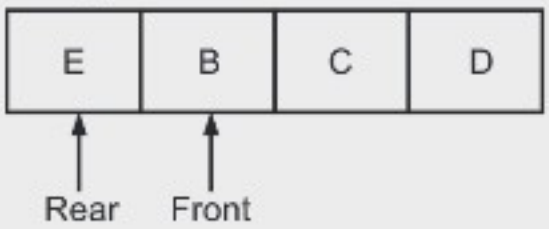
5. add(D)



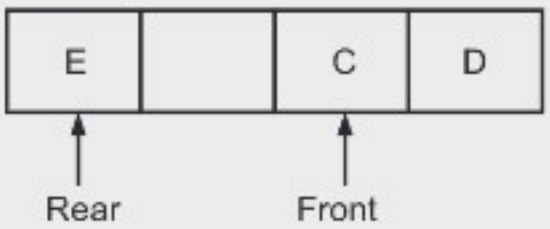
6. delete()



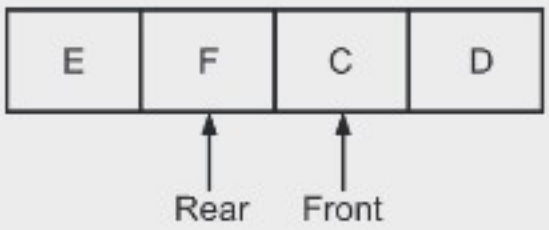
7. add(E)



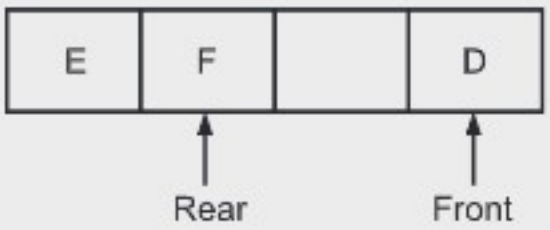
8. delete()



9. add(F)



10. delete()



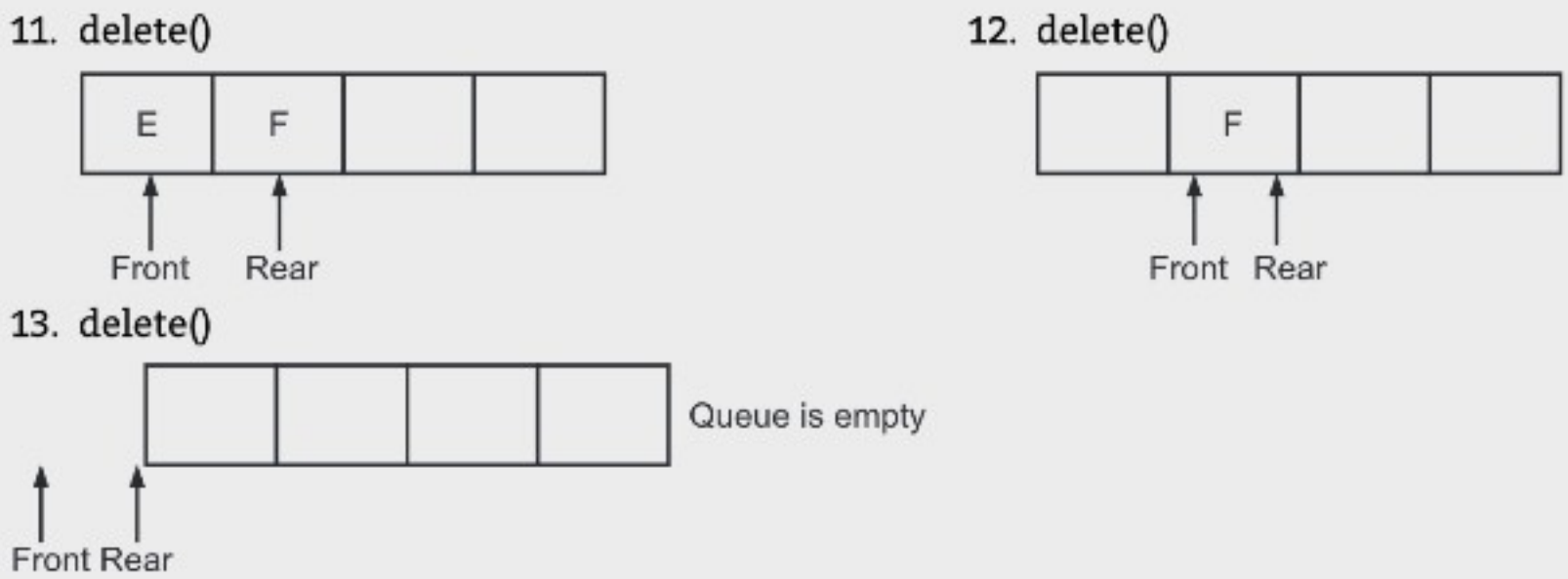


Fig. 4.21: Operations on circular queue

4.8.2 Priority Queue [S-19, 18]

4.8.2.1 Definition

- 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**.
- 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.
- 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 i.e on First Come First Served basis.

Representation of priority queue using linked list in memory:

- A Node of 3 fields i.e. it contains data, priority and pointer to next. Fig. 4.22 shows the priority queue of four elements where node B and node C has same priority, A has higher priority.

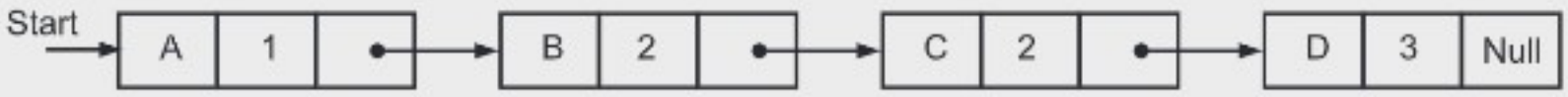


Fig. 4.22: Priority queue

4.8.2.2 Examples of Priority Queue

1. Priority queues are used for implementing job scheduling by the operating system where jobs with higher priority are to be processed first.

2. A list of patients in an emergency room, each patient might be given a ranking that depends on the severity of the patients illness.
3. A list of background jobs on a multiuser computer, each background job is given a priority level.

4.8.2.3 Operations on a Priority Queue

1. **IsEmpty:** To check priority queue is empty or not.
2. **IsFull:** To check priority queue is full or not.
3. **Insert:** If not Full, insert an element.
4. **Delete:** If not empty, delete the item with higher priority.

4.8.2.4 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.

Note: *Refer functions of following program initPQ(), isFullPQ(), isEmptyPQ(), insertPQ(), deletePQ() for operations of queue like Initialize, IsEmpty, IsFull, Insert, Delete.*

Program 4.9: 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);
}
```

4.9 APPLICATIONS OF QUEUE [S-18]

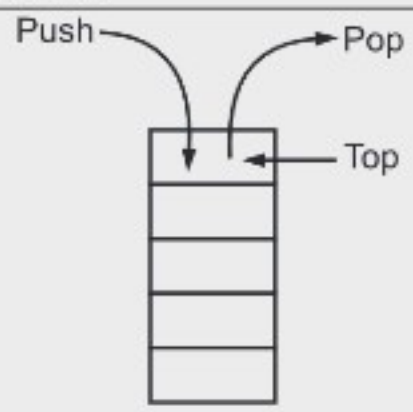
- Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios:
  1. Queues are used in job scheduling by the operating system. For example, printer, 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 multiuser systems queues are used.
5. In batch programming to process the job sequentially.

4.10 STACK VS QUEUE [W-18]

	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.
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

contd. ...

Visualized as	Stacks are visualized as vertical collections.	Queues are visualized as horizontal collections.
Diagrammatic representation	 **Fig. 4.23**	**Fig. 4.24**
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)

Examples

Example 7: 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 8: 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 9: Trace the output of following code segment.

```
1. 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));
```

Output: 1 2 3 4 5 7

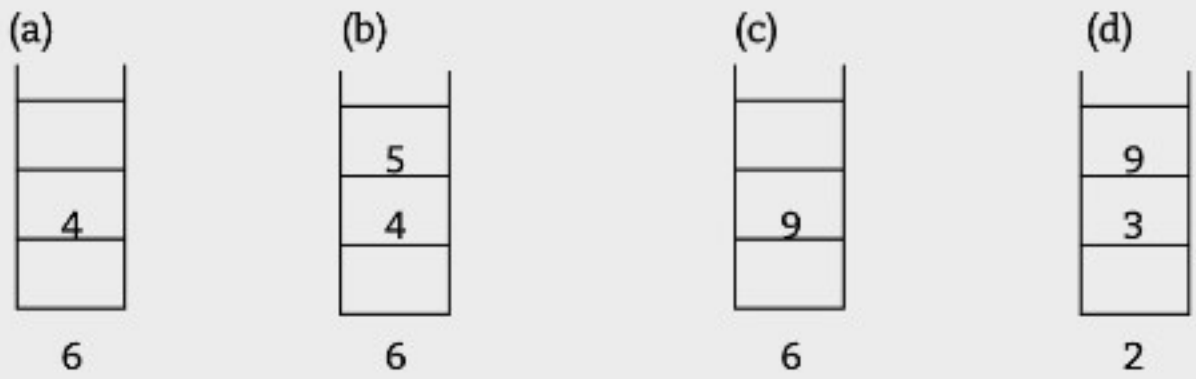
Summary

- A stack and queue is a linear data structure.
- A stack is implemented in LIFO manner and Queue is implemented in FIFO manner.
- In stack insertion is done at one end called top.
- In queue insertion is done at one end called rear and deletion at other end called front.
- Stack primitive operations: init, push, pop, peek, IsEmpty, IsFull
- Queue primitive operations: init, enqueue, dequeue, IsEmpty, IsFull
- An insertion in stack is called push and deletion is called pop.
- An insertion in queue is called enqueue and deletion is called dequeue.
- A stack and queue can be represents statically, (using array) and dynamically (using linked list).
- Expression has different forms infix and postfix.
- The operator is place between operands in infix expression.
- The operator is place after operands is postfix expression.
- Postfix expression is also known as Reverse Polish notation.
- The application of stack are expression conversion, expression evaluation, back-tracking, reversing a list, function call and recursion etc.
- Types of Queues: Circular queue and Priority queue.
- Circular queues are the queue allow to wrap around from end to start.
- Priority queue is finite collection of elements where each element is assigned a priority.
- Queues are used in many applications like: Job scheduling, simulation, process scheduling in multi user system, printer queue etc.

Check Your Understanding

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?



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 can't 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. 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

15. A queue follows _____.

(a) FIFO (First In First Out) principle (b) LIFO (Last In First Out) principle

(c) Ordered array (d) Linear tree

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. (a)	

Practice Questions

Q.I Answer the following questions in short:

1. Illustrate different forms of writing an arithmetic expression with proper example.
2. Enlist various applications of stack.
3. List the different primitive stack operation.
4. What are different operations performed on queue?
5. What are the applications of queue?
6. What is priority queue? How they are implemented?
7. State the operations perform on priority queue.

Q.II Answer the following questions:

1. If the characters of a line of text are pushed on a stack in order from left to right, and then the stack is popped until empty, what does the sequence of popped characters represent?
2. In the following sequence a letter means push that letter on the stack and * means pop a character from the stack. Indicate the character resulting from each pop.
   LAST***IN***FIRST***OUT***STACK***
3. Consider the a/b + c*d expression. Let a = 6, b = 2, c = 3 and d = 4. Show how the given expression will be evaluated for the given values with the help of stack.
4. Evaluate the following postfix expression using stack.
   623 + – 382 / + *2$3 + where $ = exponentiation
5. What is the result of evaluating AB-CD*/ postfix expression where A=2, B=10, C=4, D=1.
6. Consider the postfix expression PQ+RS-*.Give the steps for evaluation when P=9, Q=8, R=7, S=4.
7. Explain the static implementation of stack.
8. Write a C function to push an element to a stack (dynamic representation).
9. Give an algorithm to evaluate postfix expressions.
10. Give an algorithm to convert an expression in infix from into postfix form.
11. Show the stack contents and output while converting the following infix string to postfix.
    A / B $ C + D * E – A * C
12. Transform the following infix expressions into reverse polish notation:
    (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)
    (v) (((A + (B ^ C) – D)) * (E – (A/C)))

13. Write short note on
    (i) Reverse Polish Notation
    (ii) Use of stack in function calls
14. Stacks are called Last in First out (LIFO) because the first element pushed onto the stack is always the last one popped. Using push( ), pop( ), and any other functions you need, write a program that reads a line from the terminal and determines whether or not it is a palindrome. A palindrome is a string that is the same spelled forward or backward. For example, "Madam was I pop I saw Madam."
15. Show how stack is used in recursive function for calculating factorial.
16. Show the Stack contents and output while converting the following infix string to postfix: A/B$C+D*E-A*C
17. With the output for the following program segment, show content of the stack after every push and pop operation. Make necessary assumptions.

```
initStack(S)
push(S, 10)
push(S, 6)
I = pop(S)
while(I > 0)
{
push(S, I* 10);
I - - ;
}
push(S, I * 10);
while(! StackEmpty(S))
  printf("%d", pop(S));
```

18. Write the functions to implement following operation on queue.
    (i) Insert, (ii) Delete, (iii) IsFull, (iv) IsEmpty
19. Write a C function to print queue data.
20. Write a C function for adding and deleting an element from Queue (use Dynamic representation).
21. Write necessary functions in C to implement dynamic queue.
22. Write necessary functions in C to implement circular queue in an array.
23. Explain the concept of priority queue with a suitable example.
24. What is priority queue? Write pseudo 'C' function to insert and delete item from priority queue.
25. Write difference between stack and queue.

Q.III Define the terms:

(i) Stack

(ii) Queue

(iii) Circular Queue

(iv) Priority Queue

(v) Stack overflow

(vi) Stack underflow

(vii) Recursion

(viii) Reverse polish notation

Previous Exam Questions

Summer 2019

1. Elements are not added in which position of stacks? **[1 M]**

(i) top (ii) middle

(iii) bottom (iv) **both (ii) and (iii)**

2. _____ is not the type of queue. **[1 M]**

(i) Priority queue (ii) Circular queue

(iii) Ordinary queue (iv) **Single ended queue**

3. Deletion can takes place one end called front operation carried out in _____ **[1 M]**

(i) Stack (ii) **Queue**

(iii) Linked list (iv) Array

4. List out different operations on stack. **[1 M]**

Ans. Refer to Section 4.2.

5. What are applications of stacks? **[3 M]**

Ans. Refer to Section 4.4.

6. Consider the following expression: a/b +c*d

Let a=6, b=2, c=3 and d=4

Show how the given expression will be evaluated for the given values with the help of stack. **[4 M]**

Ans. First convert that expression in postfix form and then refer Example 4.

7. Explain static implementation of stack. **[4 M]**

Ans. Refer to Section 4.3.1.

8. Give one advantage of Queue. **[1 M]**

Ans. Refer to Section 4.8.1.4.

9. Write short note on priority Queue. **[3 M]**

Ans. Refer to Section 4.8.2

Winter 2018

1. The postfix form of the expression (A + B) * (C * D – E)*F / G is: **[1 M]**

   (i) AB + CD * E – FG /** (ii) AB + CD * E – F ** G /

   (iii) AB + CD * E – * F * G / (iv) AB + CDE * – * F * G /

2. The data structure required to check whether an expression contains balanced parenthesis is: **[1 M]**

   (i) Stack (ii) Tree

   (iii) Queue (iv) Array

3. What is Polish Notation? **[1 M]**

Ans. Refer to Section 4.4.2.

4. What the output for the following program segment, show content of the stack after every push and pop operation. Make necessary assumptions. **[4 M]**

```
initStack(S)
push(S, 10)
push(S, 6)
I = pop(S)
while(I > 0)
{
    push(S, I* 10);
    I - - ;
}
push(S, I * 10);
while(! StackEmpty(S))
printf("%d", pop(S));
```

Ans. Refer to Example 1.

5. Show the Stack contents and output while converting the following infix string to postfix: **[3 M]**

   A/B$C+D*E-A*C

Ans. Refer to Example 2.

6. Write any four applications of Stack. Explain. **[4 M]**

Ans. Refer to Section 4.4

7. Define Circular Queue. **[1 M]**

Ans. Refer to Section 4.8.1.1

8. Write a C function for adding and deleting an element from Queue (use Dynamic representation). **[4 M]**

Ans. Refer to Section 4.7.2.1

9. Differentiate between Stack and Queue. **[4 M]**

Ans. Refer to Section 4.10

Summer 2018

1. State true or false. **[1 M]**

Stack allow us to add and remove elements from one end i.e. top

(i) true (ii) false

Ans. (i)

2. List the different primitive stack operations. **[1 M]**

Ans. Refer to Section 4.2.

3. Consider the postfix expression PQ + RS – *. Given the steps for evaluation when P = 9, Q = 8, R = 7 and S = 4. **[4 M]**

Ans. Refer to Example 4.

4. Illustrate different forms of writing an arithmetic expression with proper example. **[5 M]**

Ans. Refer to Section 4.4.2.

5. Explain the static implementation of stack. **[4 M]**

Ans. Refer to Section 4.3.1.

6. List the applications of queue. **[4 M]**

Ans. Refer to Section 4.9.

7. What is a circular queue. Give its advantages. **[4 M]**

Ans. Refer to Section 4.8.1.

8. Write the functions to implement following operations on queue **[4 M]**

(i) insert (ii) delete

(iii) isfull (iv) isempty

Ans. Refer to Section 4.7.1.1.

9. What is Priority Queue. How are they implemented **[3 M]**

Ans. Refer to Section 4.8.2.

■■■

5... Trees

Objectives...

- To study basic concepts of Trees.
- To study basic concepts of Binary Search Trees.
- To understand various Applications of Trees.
- To learn How Trees are useful for Solving a Wide Range of Problems?
- To study Binary Search Tree and How it is useful for Fast Retrieval of Data?
- To study the Applications of Trees such as Heap Sort and AVL.

5.1 INTRODUCTION

- In the previous chapters we have already discuss the sequential and linked representation of linear data structure such as arrays, linked list, stack and queues.
- But many times we are required to have more than two paths from data object (non-linear) when we have to represent one or many relationship. The best way is to use non-linear data structure.
- Tree is a non-linear data structure. Non-linear data structures are capable of expressing more complex relationship than linear data structure. In general, where ever the hierarchical relationship among data is to be preserved tree is used.

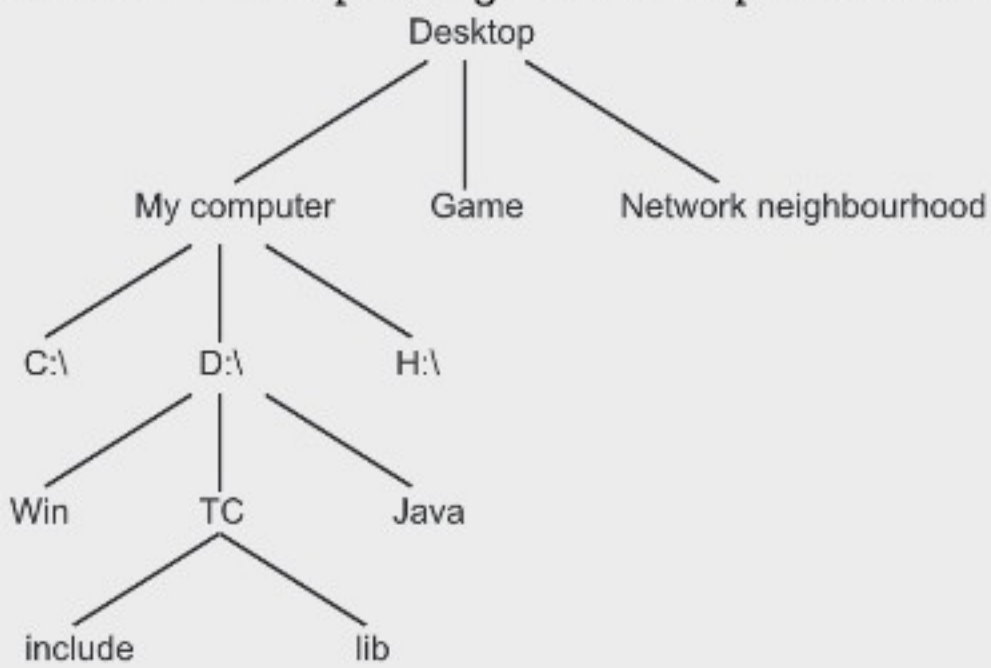


Fig. 5.1: Tree representation of directory system

- A Tree is a widely used Abstract Data Type (ADT) that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.
- A tree is, used for improving database search time, game programming, 3D graphics, data compression and in file systems.
- Let us see example, directory structure stored by operating system, the operating system organizes files into directories and subdirectories. This can be viewed as tree shown in Fig. 5.1.

Common operations on tree:

- Basically, tree consists of following operations:
  - Searching for an item (traversing).
  - Adding the new item.
  - Deleting the item.
  - Removing a whole section of a tree (called pruning).
  - Adding a whole section to a tree (called grafting).
  - Finding the root of any node.
  - Enumerating all the items.

Common uses for trees are:

- To manipulate hierarchical data.
- To make information easily searchable.
- To manipulate sorted lists of data.

5.2 BASIC CONCEPTS AND TERMINOLOGY

5.2.1 Definition

- A Tree is a finite set of one or more nodes such that:
  1. There is a specially designated node called the root.
  2. Remaining nodes are partitioned into ($n \geq 0$) disjoint sets $T_1, T_2, \ldots, T_n$ where each of these sets is a tree. Each $T_1, T_2, T_3 \ldots T_n$ are called sub trees of the root.
- The Fig. 5.2 (a) shows the empty tree, there are no nodes. The Fig. 5.2 (b) shows the tree with only one node. The tree in Fig. 5.2 (c) has 12 nodes.

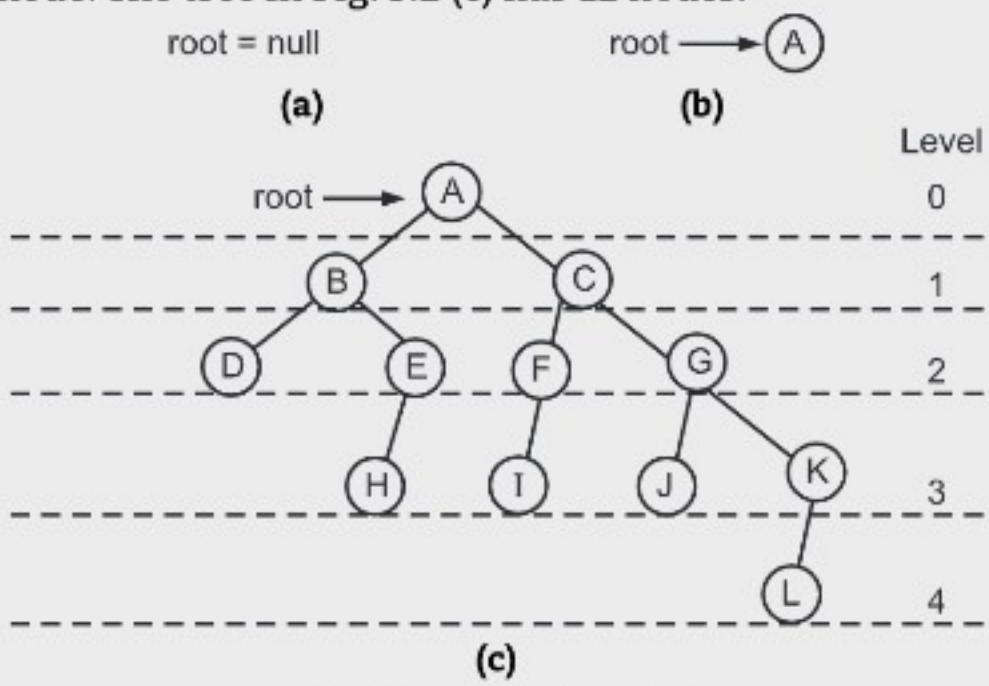


Fig. 5.2: Tree structure

1. The tree consists of 12 nodes (Refer Fig. 5.2 (c)).
2. The root of tree is A, it has 2 subtrees. The roots of these subtrees are called the children of the root.
3. The nodes with no subtrees are called terminal node or leaves. There are 5 leaves in the tree of Fig. 5.2 (c).
4. Node B is left successor and Node C is right successor of Node A.

- Because family relationship can be modeled as trees, we often call the root of a tree (or subtree) the parent and the nodes of the subtrees the children; the children of a node are called siblings.

5.2.2 Terminology

- A tree consists of following terminology:

1. **Null tree:** A tree with no nodes is a null tree (Refer Fig. 5.2 (a)).
2. **Node:** In tree data structure, every individual element is called as Node. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure. Fig. 5.2 (c) has 12 nodes.
3. **Root Node**: In a tree data structure, the first node is called as Root Node. A root node is special node which doesn't have parent node. Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node.
4. **Parent Node**: In a tree data structure, the node which is a predecessor of any node is called as Parent Node. In simple words, the node which has a branch from it to any other node is called a Parent node.
5. **Child Node:** In a tree data structure, the node which is descendant of any node is called as Child Node. In simple words, the node which has a link from its parent node is called as child node.
6. **Siblings:** In a tree data structure, nodes which belong to same Parent are called as Siblings. In simple words, the nodes with the same parent are called Sibling nodes. **[W-18]**
7. **Leaf Node:** In a tree data structure, the node which does not have a child is called as Leaf Node.
8. **Degree of a node:** In a tree data structure, the total number of children of a node is called as Degree of that Node. In simple words, the Degree of a node is total number of children it has. The degree of A is 2, degree of K is 1 and degree of L is 0, (Refer Fig. 5.2 (c)).
9. **Degree of a tree:** The degree of a tree is a maximum degree of the nodes in the tree. In simple words, the highest degree of a node among all the nodes in a tree is called as Degree of tree.

   In Fig. 5.2 (c), degree of the tree is 2.

10. **Level of node:** In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).

    The level of node = Level of its parent + 1. The nodes with the same level number are said to the same generation. **[S-19]**

11. **Height:** In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree.

12. **Depth:** In a tree data structure, the total number of edges from root node to a particular node is called as DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree.

5.3 BINARY TREE AND IT'S TYPES [W-18]

1. Binary Tree:

- A binary tree is a special form of a tree. A binary tree is finite set of nodes, which is either empty or partitioned into three sets, one which is the root and two disjoint binary trees called left subtree and right subtree. It is a tree where every node can have at the most two branches (children).

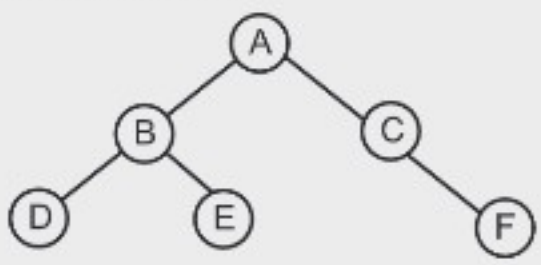


Fig. 5.3: Binary tree

2. Strictly Binary Tree: **[S-19]**

- A strictly binary tree is a binary tree where all non-leaf nodes have two branches.

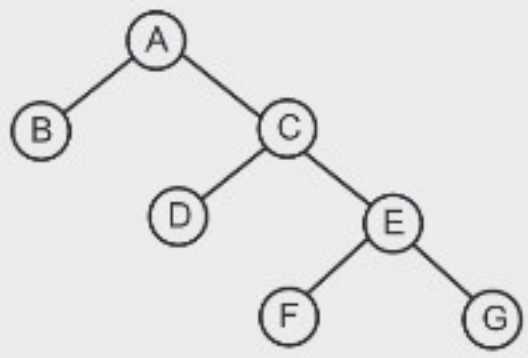


Fig. 5.4: Strictly binary tree

3. Complete Binary Tree:

- In a complete binary tree all levels except the last are completely filled, and in the last level all nodes are to the left as much as possible. This means that all nodes have two

children except the nodes at the lowest two levels. At the lowest level the nodes have (by definition) zero children, and at the level above that nodes can have 0, 1 or 2 children.

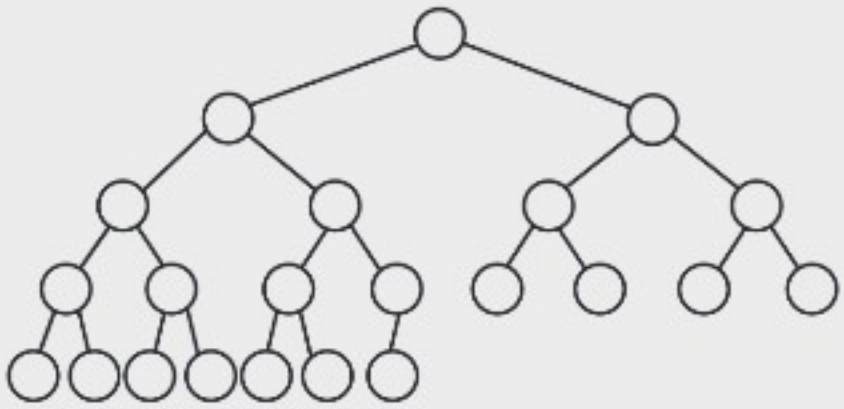


Fig. 5.5: Complete binary tree of height 2

- The maximum no. of nodes on level i is 2^i. The complete binary tree with a total of d levels contains:

$$\text{Number of nodes} = \sum_{i=0}^{d-1} 2^i = 2^d - 1$$

$$= 2^3 - 1 \text{ (total levels = 3 i.e. 0 to 2)}$$

$$= 7 \text{ nodes}$$

4. Full Binary Tree:

- A full binary tree is defined as a binary tree in which all nodes have either zero or two child nodes.

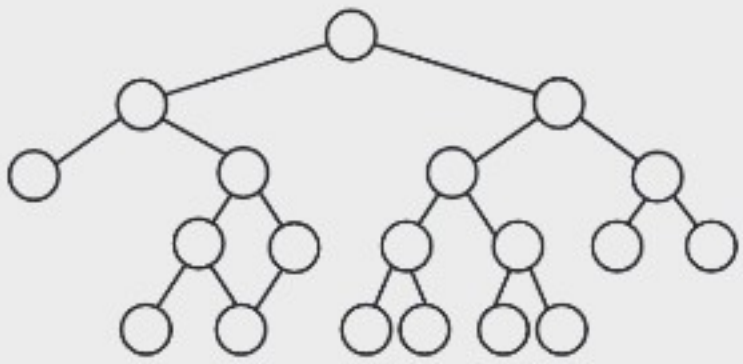


Fig. 5.6: Full binary tree

5. Perfect Binary Tree:

- A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.

(a) Perfect binary tree (b) Not a Perfect binary tree

Fig. 5.7

6. Skewed Binary Tree:

- A tree in which every node have either only left subtree or right subtree is called as skewed binary tree. The tree can be left skewed tree or right skewed tree.

(a) Left skewed binary tree **(b) Right skewed binary tree**

Fig. 5.8: Skewed binary tree

7. Rooted Tree: A rooted tree is a tree with a designated node (topmost node) called the root. In rooted tree every edge directly or indirectly originates from the root.

Fig. 5.9: Rooted Binary Tree

8. Binary Search Tree:

- A binary search tree is a binary tree in which the nodes are arranged according to their values. The left node has a value less than its parent and the right node has a value greater than the parent node. Hence, all nodes in the left subtree have values less than the root and the nodes in the right subtree have values greater than the root.

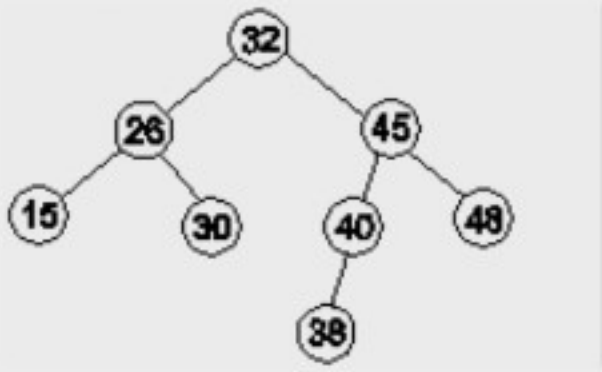


Fig. 5.10: Binary search tree

5.4 REPRESENTATION OF BINARY TREE

- Binary trees can be represented in two ways:
  1. Sequential representation using array.
  2. Linked representation.

- We shall give more emphasis to linked representation as is more popular than the corresponding sequential structure. The two main reasons are:
  - A binary tree has a natural implementation in linked storage.
  - The linked structure is more convenient for insertions and deletions.
- Let us study both representations in detail.

5.4.1 Sequential Representation of Binary Tree (Using Array)

- One of the way to represent binary tree using array is to store nodes level by level, starting from the zero level where the root is present.
- Such representation needs sequential numbering of the nodes starting with nodes on level zero, then those on level 1 and so on.
- Already, we have defined complete tree. A complete binary tree of height h has $(2^{h+1} - 1)$ nodes in it. The nodes can be stored in one dimensional array, TREE, with the node numbered at location TREE(i). A array of size $(2^{h+1} - 1)$ or $2^d - 1$ (where d = no. of levels) is required.
- The root node is stored in the first memory location as the first element in the array.
- Following rules can be used to decide the location of any i^{th} node of a tree:
  1. For any node with index i, $1 \le i \le n$;
     (a) PARENT (i) = $\left\lceil \frac{i}{2} \right\rceil$ if $i \ne 1$
     If i = 1 then it is root which has no parent
     (b) LCHILD (i) = 2 * i, if $2i \le n$
     If 2i > n, then i has no left child
     (c) RCHILD(i) = 2i + 1, if $2i + 1 \le n$
     If (2i + 1) > n, then i has no right child.

Example 1:

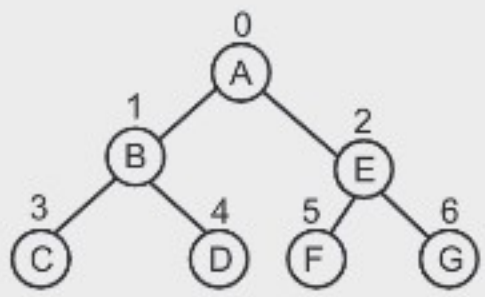


Fig. 5.11: Perfect binary tree

In Fig. 5.11,

Number of levels = 3 (0 to 2) and height = 2

Therefore, we need the array of size $2^{2+1} - 1$ is $2^3 - 1 = 7$.

The representation of the above binary tree using array is as followed:

1	2	3	4	5	6	7
A	B	C	D	E	F	G

We will number each node of tree starting from the root. Nodes on same level will be numbered left to right.

Example 2:

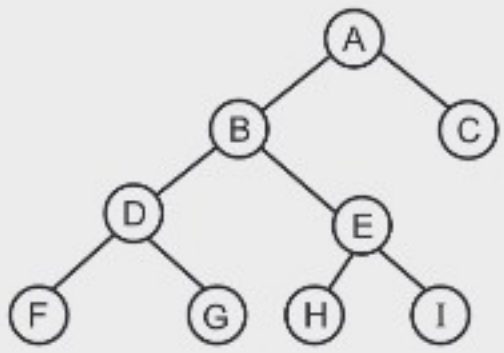


Fig. 5.12: Complete binary tree

Here, depth = 4 (level), therefore we need the array of size $2^4 - 1 = 15$. The representation of the above binary tree using array is as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	–	–	F	G	H	I	–	–	–	–

level 0 1 2 3

We can apply the above rules to find array representation.

1. Parent of node E (node 5) $= \frac{i}{2} = \frac{5}{2} = 2$ i.e. B.

   Hence, node B is at position 2 in the array.
2. Left child (i) = 2i.

   For example, left child of E $= 2 \times 5 = 10$ i.e. H.

   Since, E is the 5th node of tree.
3. Right child (i) = 2i + 1

For example: Right child of D $= 2 \times 4 + 1 = 8 + 1 = 9$ i.e. G.

Since, D is the 4th node of tree and G is the 9th element of an array.

Example 3: Consider the example of skewed tree.

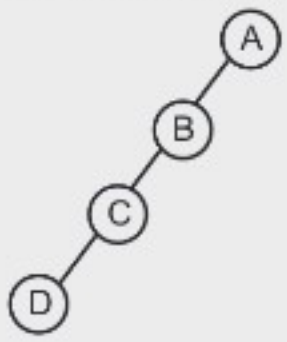


Fig. 5.13: Skewed binary tree

The tree has following array representation.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B		C				D							

The array size is $2^4 - 1 = 15$.

Thus, only four positions are occupied in the array and 11 are wasted.

- From the above example it is clear that binary tree using array seems to be easiest, but the method has certain drawbacks. In most of the representation there will be a lot of unused space. For complete binary trees the representation is ideal as no space is wasted.
- But for the skewed binary tree (See Fig. 5.13), less than half of the array is used and the more is left unused. In the worst case a skewed binary tree of depth k will required $2^{k+1} - 1$ locations of array and occupying only few of them.

Advantages of Array representation of Binary Tree:

1. We can access any node from other node by calculating index.
2. Data are stored without any pointers to their successor or ancestor.
3. Some programming languages, (For example, BASIC, FORTRAN) where dynamic allocation is not possible we use array representation to store a tree.

Disadvantages of Array representation of Binary Tree:

1. We cannot store information of more nodes than the specified array size. Array size cannot be changed during execution.
2. It allows only static allocation.
3. Other than on almost complete binary trees, many array entries are unutilized.
4. Inserting a new node to it or deleting a node from it are inefficient with this representation, because these require considerable data, movement up and down the array to maintain the levels which demand excessive amount of processing time.

5.4.2 Linked Representation of Binary Tree

- Another way to represent a binary tree is linked list, which is more memory efficient than the array representation.
- All nodes should be allocated dynamically. The fundamental component of binary tree is node. We need each node with data and link fields. Also we need a separate pointer variable to enable us to know where the tree is in memory. The most commonly used name to this pointer is root which will point to the root of the tree.
- Each node consists of three fields, Lchild, Data and Rchild, which is shown in Fig. 5.14.

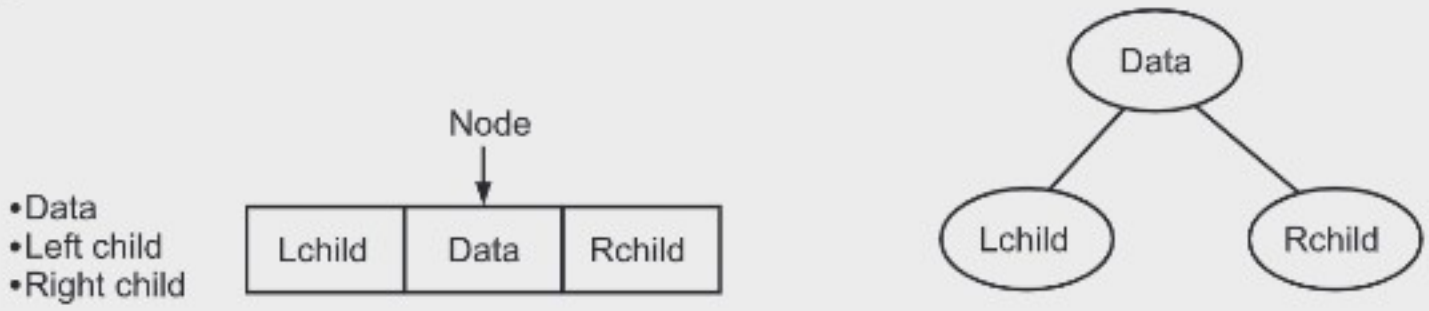
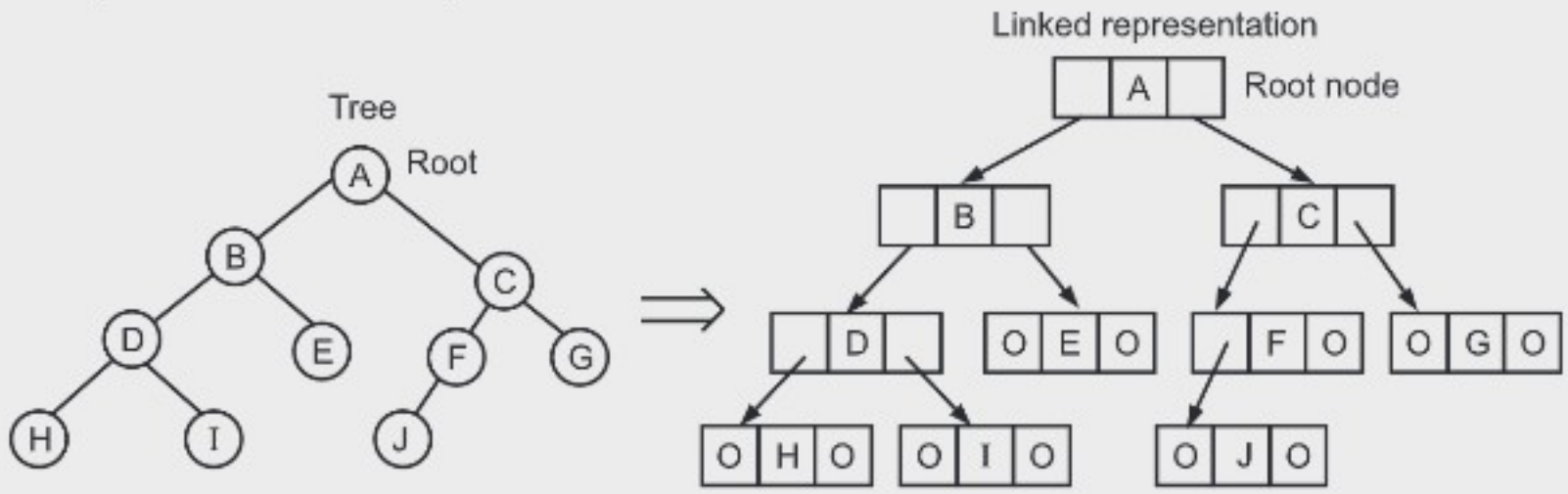
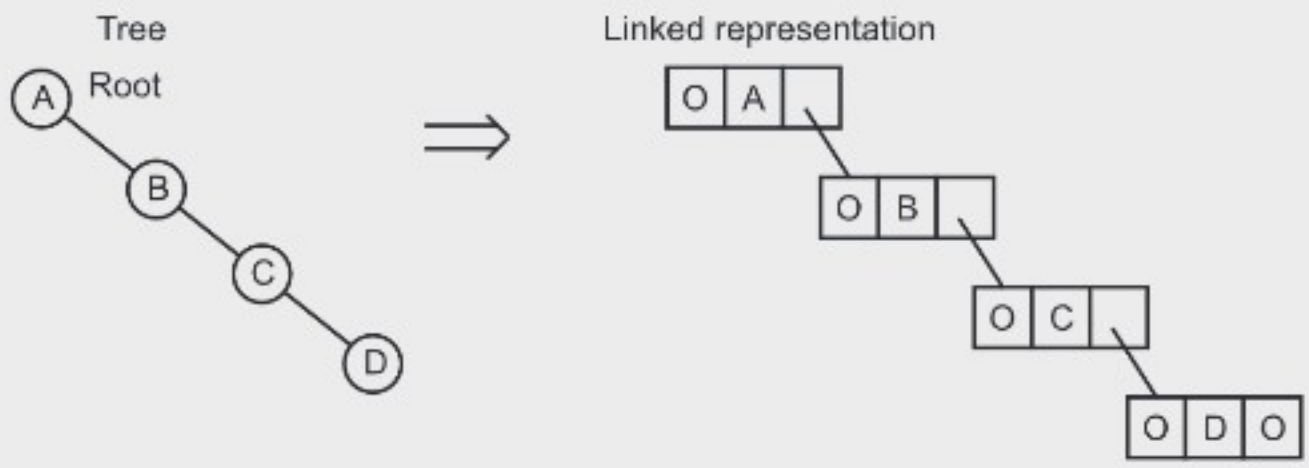


Fig. 5.14: Fields of Binary tree in linked representation

- The data field contains the given values. The **Lchild** field holds the address of its left node and the **Rchild** field holds the address of right node. The Fig. 5.15 (a) and (b) shows the linked representation of binary tree.
- In Fig. 5.15 (a) and (b), NULL i.e zero (0) stored at **Lchild** and **Rchild** field represent that respective child is not present



(a)



(b)

Fig. 5.15: Representation of Binary tree

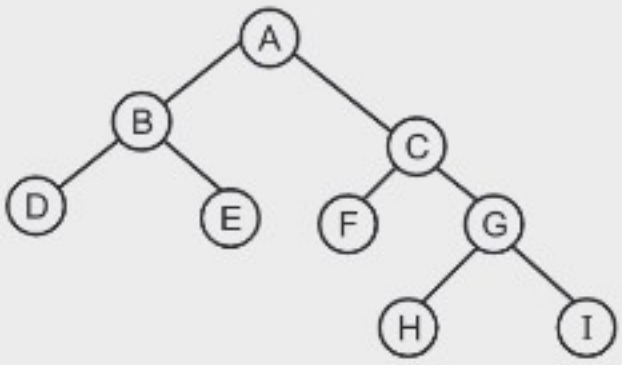
- Each node represents information (data) and two links - Lchild and Rchild are used to store addresses of left child and right child of a node.

Declaration in 'C':

- We can define the node structure as follows:

```
struct tnode
{
    struct tnode *lchild;
    <datatype> data;    /* data type is a data type of field data*/
    struct tnode *rchild;
};
```

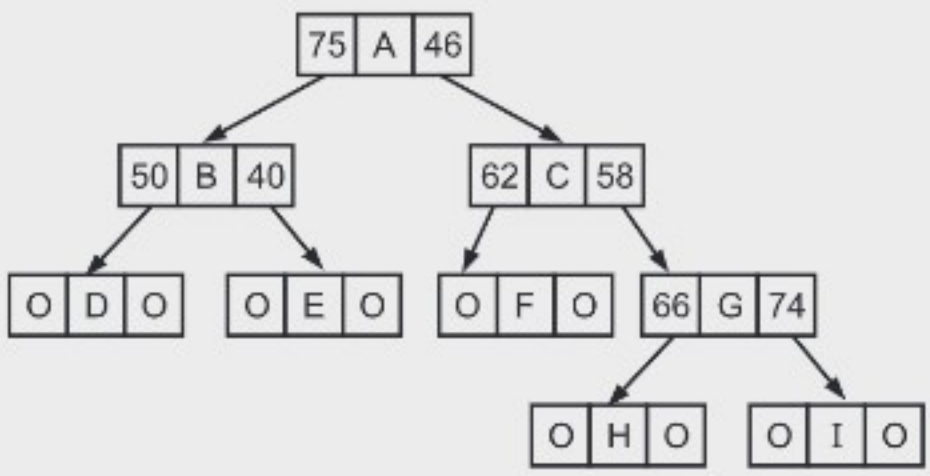
- Using this declaration for linked representation, the binary tree representation can be logically viewed as in Fig. 5.16 (c). In Fig. 5.16 (b) physical representation shows the memory allocation of nodes.



(a) A binary tree

Address	Node		
	Lchild	Data	Rchild
50	0	D	0
75	50	B	40
40	0	E	0
89	75	A	46
62	0	F	0
46	62	C	58
66	0	H	0
58	66	G	74
74	0	I	0

(b) A binary tree and its various nodes (physical view)



(c) Logical view

Fig. 5.16: Linked representation of binary tree

Advantages of linked representation of Binary Tree:

1. Efficient use of memory than that of sequential representation.
2. Insertion and deletion operations are more efficient.
3. Use for dynamic memory allocation.

Disadvantages of linked representation of Binary Tree:

1. If we want to access a particular node, we have to traverse from root to that node; there is no direct access to any node.
2. Since, two additional pointer are present (left child and right child), the memory needed per node is more than that of sequential representation.
3. Programming languages not supporting dynamic memory management are not useful for this representation.

5.5 OPERATIONS ON BINARY TREES

- The primitives operations on binary tree are as follows:
  1. **Create:** Creating a binary tree.
  2. **Traversal:** To visit all the nodes in a binary tree.
  3. **Insertion:** To insert a node in the binary tree.
  4. **Deletion:** To delete a node from the binary tree.

5.5.1 Insertion

- Insertion operation is used to insert a new node into a binary tree; either it is branch node (internal node) or leaf node. Inserting a branch node is generally based on some criteria which is usually in context of a special form of a binary tree. Fig. 5.17 shows the insertion of node as a left child of a node having data 'D'.
- The Insertion involves two steps:
  1. Search for a node in the given binary tree after which insertion is to be made.
  2. Create a link for a new node and new node becomes either left child or right child.

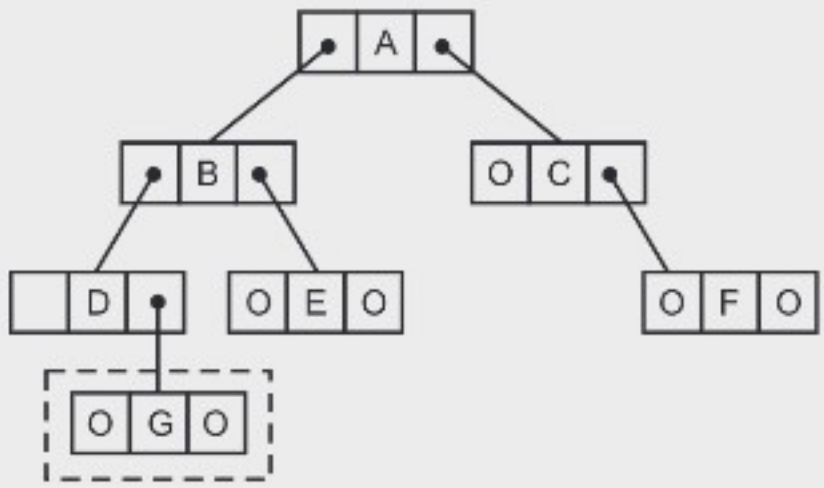


Fig. 5.17: Insertion of node G

5.5.2 Deletion

- Deletion operation deletes any node from any non-empty binary tree. In order to delete a node in a binary tree, it is required to reach at the parent node of the node to be deleted. The link field of the parent node which stores the address of the node to be deleted is then set by a null entry as shown in Fig. 5.18.

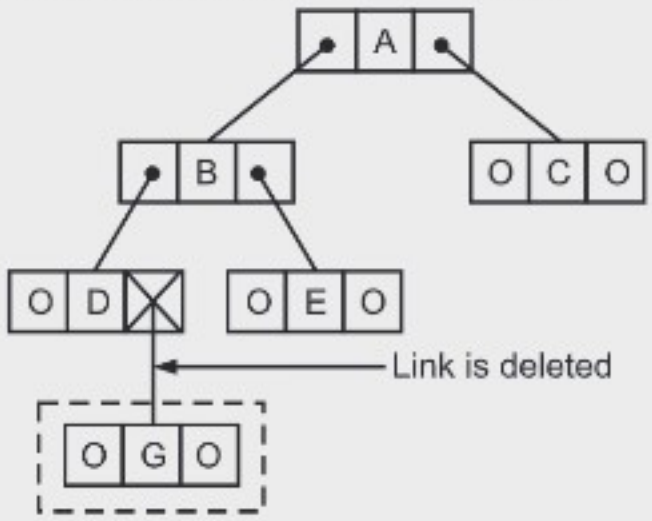


Fig. 5.18: Node with data G is deleted and rchild of D becomes null

5.6 BINARY TREE TRAVERSAL [S-18]

- When we want to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree, displaying order of nodes depends on the traversal method.
- Binary tree Traversal means visiting every node exactly once. There are many operations performed on tree such as insertion, deletion, searching a node etc. For all these operations we need to traverse a tree. A full traversal on a binary tree gives a linear ordering of the data in the tree.
- **Expression Tree:** An expression tree is a representation of expressions arranged in a tree-like data structure. In other words, it is a tree with leaves as operands of the expression and nodes contain the operators. Similar to other data structures, data interaction is also possible in an expression tree. [W-18]
- Example: Here is the expression tree for the expression:(5-x)*y+6/(x + z)

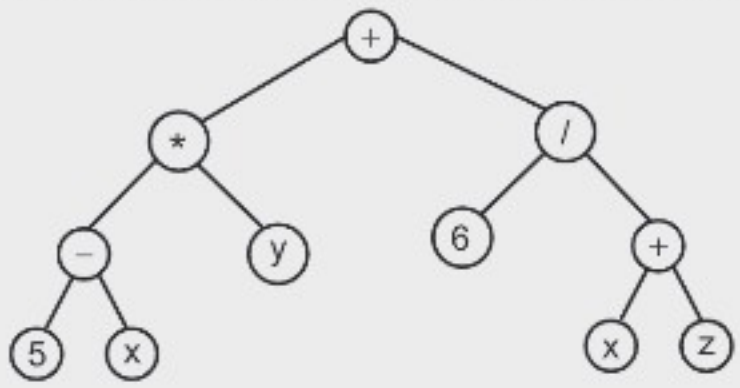


Fig. 5.19: Expression Tree for above Example

- There are three methods of traversing a binary tree:
  - Preorder traversal
  - Inorder traversal
  - Postorder traversal
- These methods are defined recursively, so that traversing a binary tree involves visiting the root and traversing its left and right subtree.

5.6.1 Preorder Traversal (DLR)

- In Preorder traversal, the root node is visited before the left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This preorder traversal is applicable for every root node of all subtrees in the tree.

(i)	Process the root Data (D)	(Data)
(ii)	Traverse the left subtree of D in preorder	(Left)
(iii)	Traverse the right subtree of D in preorder	(Right)
	Preorder ⇒ Data – Left – Right	**(DLR)**

Example:

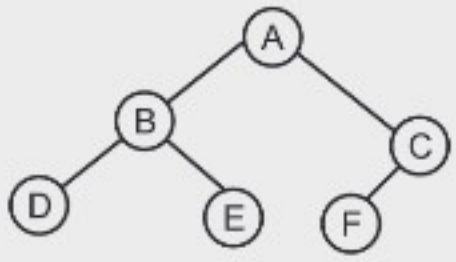


Fig. 5.20 (a): Tree

- A preorder traversal of a tree in Fig. 5.20 (a) visit node in a sequence ABDECF.
- For the expression tree, preorder yields a prefix expression.

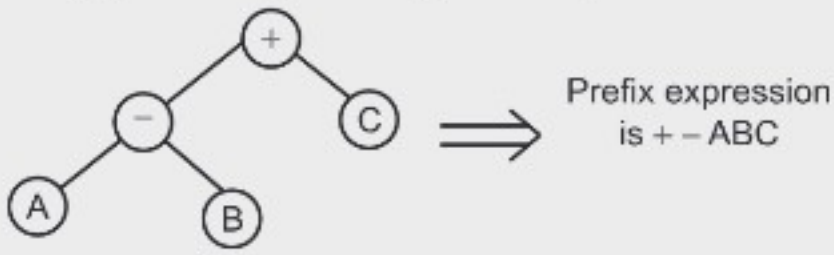


Fig. 5.20 (b): Prefix expression

- In preorder traversal, we visit a node, traverse left and continue again. When we cannot continue, move right and begin again or move back, until we can move right and stop. Preorder function can be written as both recursive and non-recursive way.

Recursive preorder traversal:

Algorithm:

```
Step 1:begin
Step 2:if tree not empty
        visit the root
        preorder (left child)
        preorder(right child)
Step 3:end
```

C function for recursive preorder traversal:

```
void preorder (struct treenode * root)
{
    if (root)
    {
        printf("%d \t", root → data);      /*data is integer type */;
        preorder (root → left);
        preorder (root → right);
    }
}
```

Example:

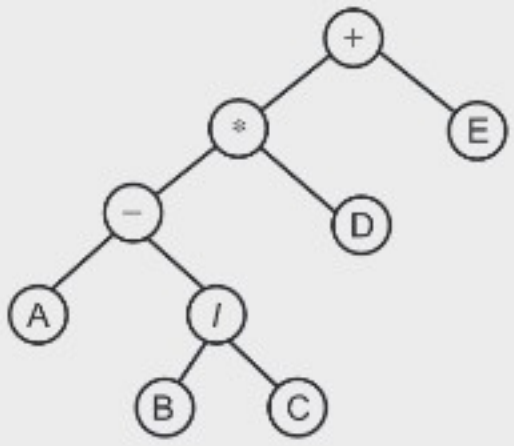


Fig. 5.21: Binary tree for expression

((A – B/C) * D) + E

- In the Fig. 5.21, tree contains an arithmetic expressions. It gives us prefix expression as:

  + * – A/BCDE

  The preorder traversal is also called as depth first traversal.

5.6.2 Inorder Traversal

- In Inorder traversal, the root node is visited between the left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting the right child node.
- This inorder traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

(i)	Traverse the left substree of D in inorder	(Left)
(ii)	Process of root Data (D)	(Data)
(iii)	Traverse the right subtree of R in inorder	(Right)
	Inorder ⇒ Left – Data – Right	**(LDR)**

Recursive inorder traversal:

Algorithm:

```
Step 1:begin
Step 2:if tree is not empty
       inorder (left child)
       visit the root
       inorder (right child)
Step 3:end
```

'C' function for Inorder Traversal:

- The in order function calls for moving down the tree towards the left until, you can go on further. Then visit the node, move one node to the right and continue again.

```
void inorder (tnode * root)
{
   if(root)
   {
      inorder (root → left);
      printf("%d", root → data);
      inorder (root → right);
   }
}
```

- This traversal is also called as symmetric traversal.

5.6.3 Postorder Traversal

Postorder (LRD):

- In Postorder traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

(i)	Traverse the left subtree of D in postorder	(Left)
(ii)	Traverse the right subtree of D in postorder	(Right)
(iii)	Process of root Data (D)	(Data)
	Postorder ⇒ Left – Right – Data	**(LRD)**

Recursive Postorder Traversal:

Algorithm:

```
Step 1:begin
Step 2:if tree not empty
       postorder (left child )
       postorder(right)
       visit the root
Step 3:end
```

'C' Function for Postorder traversal:

```
void postorder (tnode * root)
   {
      if (root)
      {
         postorder(root → left);
         postorder(root → right);
         printf("%d", root → data);
      }
   }
```

Examples 4: Traverse each of the following binary tree in inorder, preorder and postorder. **[S-18]**

(a)

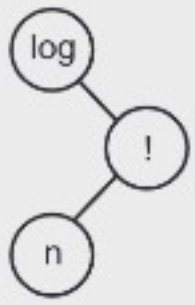


Fig. 5.22

Traversals: Preorder: log ! n

Inorder: log n!

Postorder: n! log

(b)

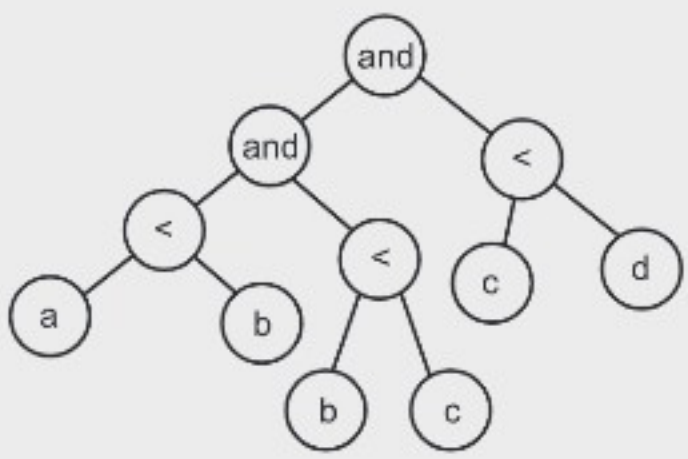


Fig. 5.23

Traversals: Preorder: and and < a b < b c < c d

Inorder: a < b and b < c and c < d

Postorder: a b < b c and c d < and

(c)

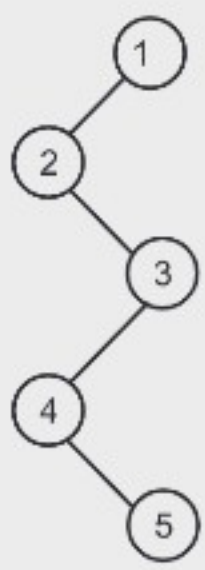


Fig. 5.24

Traversals: Preorder: 1 2 3 4 5
Inorder: 2 4 5 3 1
Postorder: 5 4 3 2 1

(d)

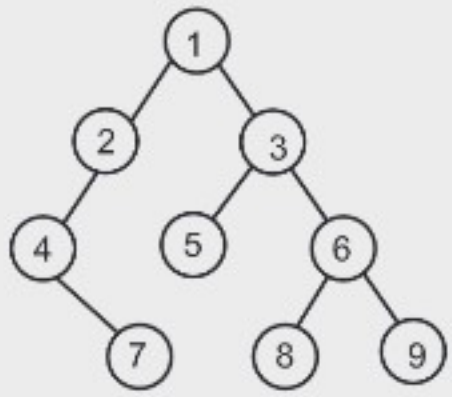


Fig. 5.25

Traversals: Preorder: 1 2 4 7 3 5 6 8 9
Inorder: 4 7 2 15 3 8 6 9
Postorder: 7 4 2 5 8 9 6 3 1

5.7 BINARY SEARCH TREE (BST)

- Consider an example where we want to search a data from the linked list. We have to search sequentially which is slower than binary search. If the list is ordered list stored in contiguous sequential storage, binary search is faster.
- If we want to insert or delete a data from the list, then in sequential list more data movements are required. With link list only few pointer manipulations are required, so binary search provides quick insertion and deletion. Hence, binary trees provides an excellent solution for searching, inserting and deleting a node.
- The Binary Search Tree (BST) is a binary tree with the property that the value in a node is grater than any value in a node's left subtree and less than any value in the node's right subtree.
- In this section we study the operations on binary search trees. Also two types of binary search trees representation: static and dynamic. Static representation of binary search tree in which the set of values in the nodes is known in advance and in dynamic, the values in a tree may change over time.

Definition of BST:

- A Binary Search Tree (BST) is a binary tree which is either empty or non-empty. If it is non-empty then every node contains a key which is distinct and satisfies the following properties:
  1. Values less than its parent are placed at left side of parent node.
  2. Values greater than its parent are placed at right side of parent node.
  3. The left and right subtrees of are again binary search trees.

Example of binary search trees:

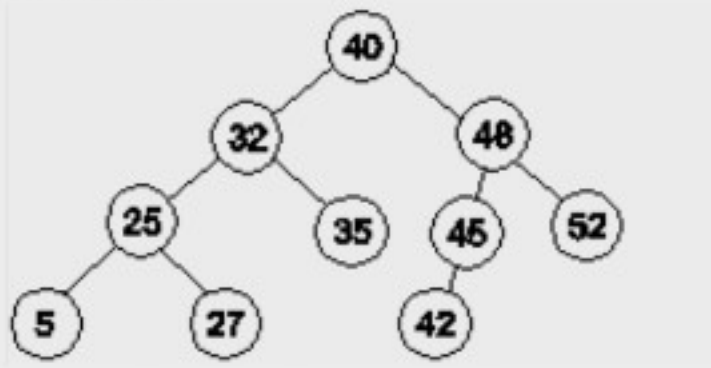


Fig. 5.26: Binary search trees

Example 5: Create a binary search tree for the following word sequence:

Tushar, Amit, Beena, Pranav, Hemant, Neeta, Deepa, Rohit.

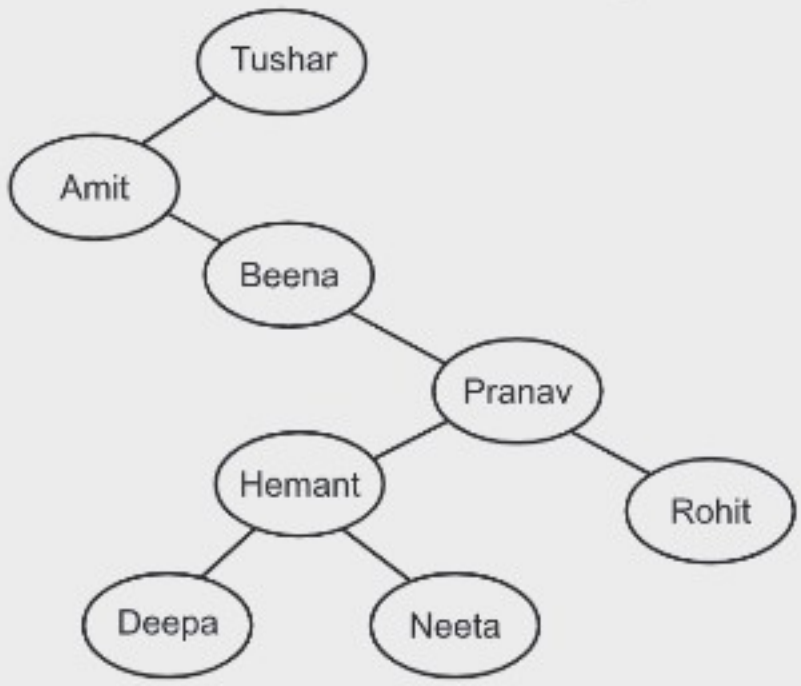


Fig. 5.27

5.7.1 Operations on Binary Search Tree

- Following operations are commonly perform on BST,
  - Search a key
  - Insert a key
  - Delete a key
  - Traverse the tree

5.7.1.1 Creating Binary Search Tree

- The function create() simply creates a empty binary search tree. This initializes the root pointer to NULL.

```
struct tree *root = NULL;
```

- The actual tree is build through a call to insert BST function, for every key to be inserted.

Algorithm:

(i) root = NULL

(ii) read key and create a new node to store key value

(iii) if root is null then new node is root

(iv) t = root, flag = false

(v) while (t ≠ null) and (flag = false) do

Case 1: if key < t → data
attach new node to t → left

Case 2: if key > t → data
attach new node to t → right

Case 3: if t → data = key
then print "key already exists"

(vi) stop.

C function for creating BST:

```
BSTNODE createBST(BSTNode *root)
{
    BSTNODE *newnode, *temp;
    char ans;
    do
    {
        newnode=(BSTNODE *) malloc(sizeof(BSTNODE));
        printf("\n Enter the element to be inserted:");
        scanf("%d", &newnode→data);
        newnode→left=newnode→right=NULL;
        if(root==NULL)
            root=newnode;
        else
        {
            temp=root;
            while(temp!=NULL)
            {
                if(newnode→data<temp→data)
                {
                    if(temp→left==NULL)
                    {
                        temp→left=newnode;
                        break;
                    }
                        else
                    temp=temp→left;
                    }
```

```
            else if(newnode→data>temp→data)
            {  if(temp→right==NULL)
               {
                  temp→right=newnode;
                  break;
               }
               else
                  temp=temp→right;
            }
         }
         printf("\n Do u want to add more numbers?");
         scanf("%c", &ans);
      }  while(ans=='y' || ans =='y')
      return(root);
   }
```

Example 6: Construct Binary Search Tree for the following elements: **[W-18, S-18]**

15, 11, 13, 8, 9, 17, 16, 18

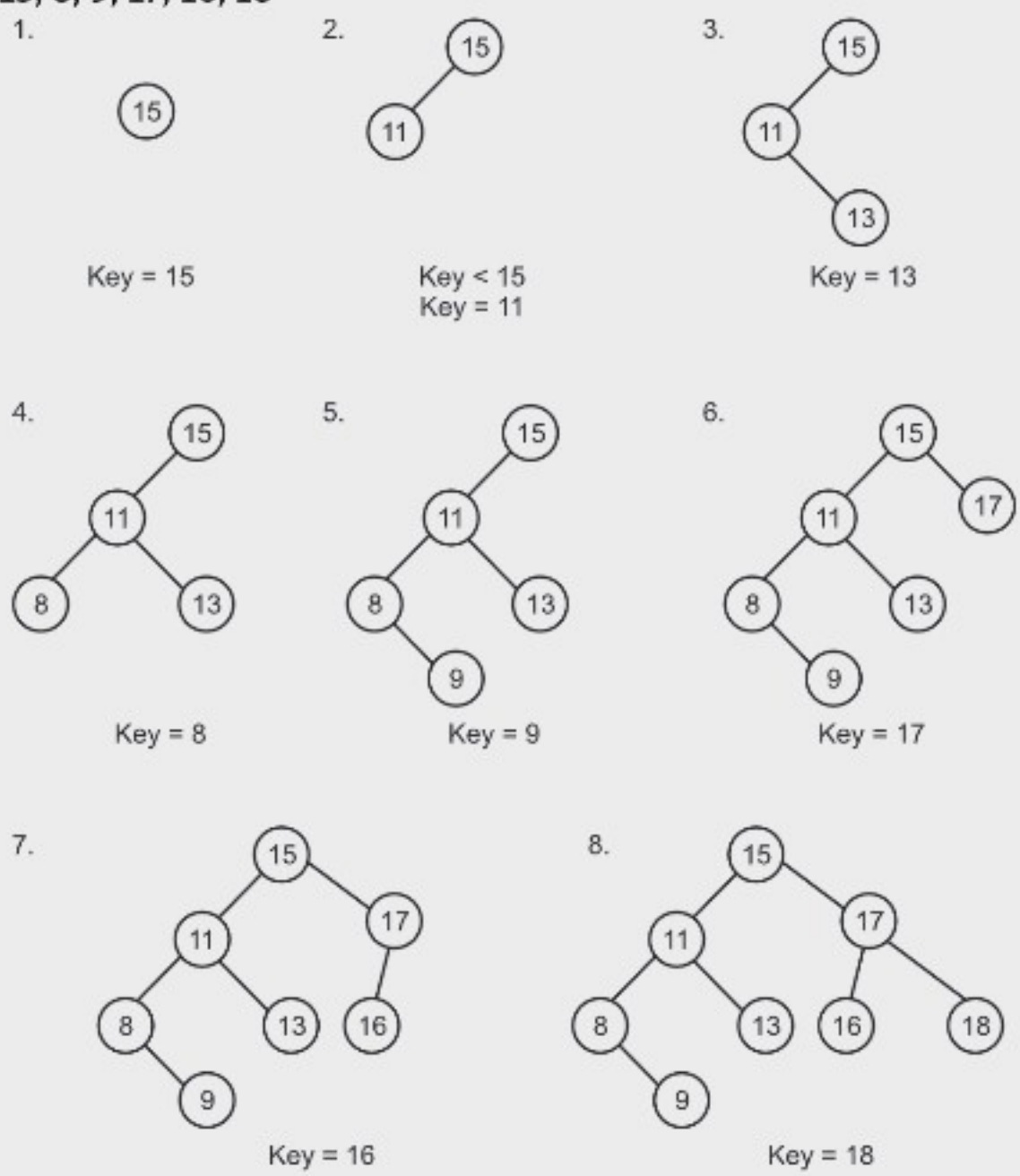


Fig. 5.28

- Here, all values in the left subtrees are less than the root and all values in the right subtrees are greater than the root.

5.7.1.2 Searching in a BST

- To search a target key, we first compare it with the key at the root of the tree. If it is the same, then the search ends and if it is less than key at root, search the target key in left subtree else search in the right subtree.

  Example:

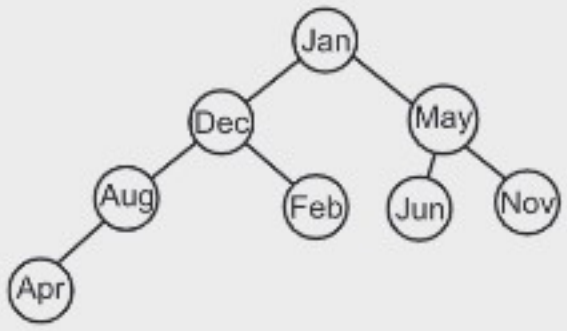


Fig. 5.29: Binary search tree

- To search 'Apr', we first compare 'Apr' with key of root 'Jan'. Since, Apr < Jan (alphabetical order) we move to left child and next compare with 'Dec'. Since, Apr < Dec, we move to the left again and compare with Aug. As Apr < Aug we move to the left. Here we find the key 'Apr' and search is successfully. If not, we continue searching until we hit an empty subtree. By using a pointer to move through the tree, we can return the value of the pointer to send the result of the search back to the caller function.

Algorithm: Search – BST (Key)

Steps:

1. Initialize t = root, flag = false
2. while (t ≠ null) and (flag = false) do

   Case 1: t· data = key
   flag = true //successful search

   Case 2: Key < t → data
   t = t → left // goto left subtree

   Case 3: Key > t → data
   t = t → right //goto right subtree

   End case

   End while
3. If (flag = true) then

   display "Key is found at node", t

   else

   display "Key is not exist"

   end if;
4. Stop

The Non Recursive C Function for search key:

```
BSTNODE *search (BSTNODE *root, int key)
{
    BSTNODE *temp=root;
    while (temp != NULL) && (temp   data != NULL)
    {
        if (key < temp → data)
        temp = temp → left;
        else
        temp = temp → right;
    }
    return(temp);
}
```

The recursive C function for search key:

```
BSTNODE *research (BSTNODE *root, int key)
{
    BSTNODE *temp=root;
    if (temp == NULL) || (temp → data == key)
        return(temp);
    else
        if (key < temp → data)
            research (temp → left, key);
        else
            research (temp → right, key);
}
```

5.7.1.3 Inserting a Node into BST

- We insert a node into binary search tree in such a way that resulting tree satisfies the properties of BST.

Algorithm: Insert_BST (Key)

Steps: 1. t = root, flag = false

2. while (t ≠ null) & (flag = false) do

Case 1: key < t → data

t1 = t;

t = t → left;

Case 2: key > t → data

t1 = t

t = t → right

Case 3: t → data = key

flag = true

display "item already exist"

break

end case

End while

3. If (t = null) then

new = getnode (node) //create node

new → data = key

new → left = null //initialize a node

new → right = null

if (t1 → data < key) then //insert at right

t1 → right = new

else t1 → left = new //insert at left

endif

4. stop

Recursive C function for inserting node into BST: [S-19]

```
BSTNODE *Insert-BST (BSTNODE *root, int n)
{
   if (root == NULL)
   {
      root = (BSTNODE *) malloc (sizeof(BSTNODE));
      root → data = n;
      root → left = root → right = NULL;
   }
   else
   if (n < root → data)
      root → left = Insert-BST (root → left, n);
   else
      root → right = Insert-BST (root → right, n);
   return(root);
}
```

Non-Recursive C function for inserting node into BST:

```
BSTNODE *Insert-BST (BSTNODE *root, int n)
{
    BSTNODE *temp, *newnode;
    newnode = (BSTNODE*) malloc (sizeof(BSTNODE));
    newnode → data = n;
    newnode → left = root → right = NULL;
    if (root==NULL)
        root = newnode;
    else
    {
        temp = root;
        while (temp)
        {
            if (n < temp → data)
            {
                if (temp → left==NULL)
                {
                    temp → left = newnode;
                    break;
                }
                else
                    temp = temp → left;
                else if (n > temp → data)
                {
                    if(temp → right==NULL)
                    {
                        temp → right = newnode;
                        break;
                    }
                    else
                        temp = temp → Rchild;}
                }
            }
        return root;
        }
    }
}
```

Example 7: Consider the insertion of keys: Heena, Deepa, Tina, Meena, Beena, Anita. Initially tree is empty.

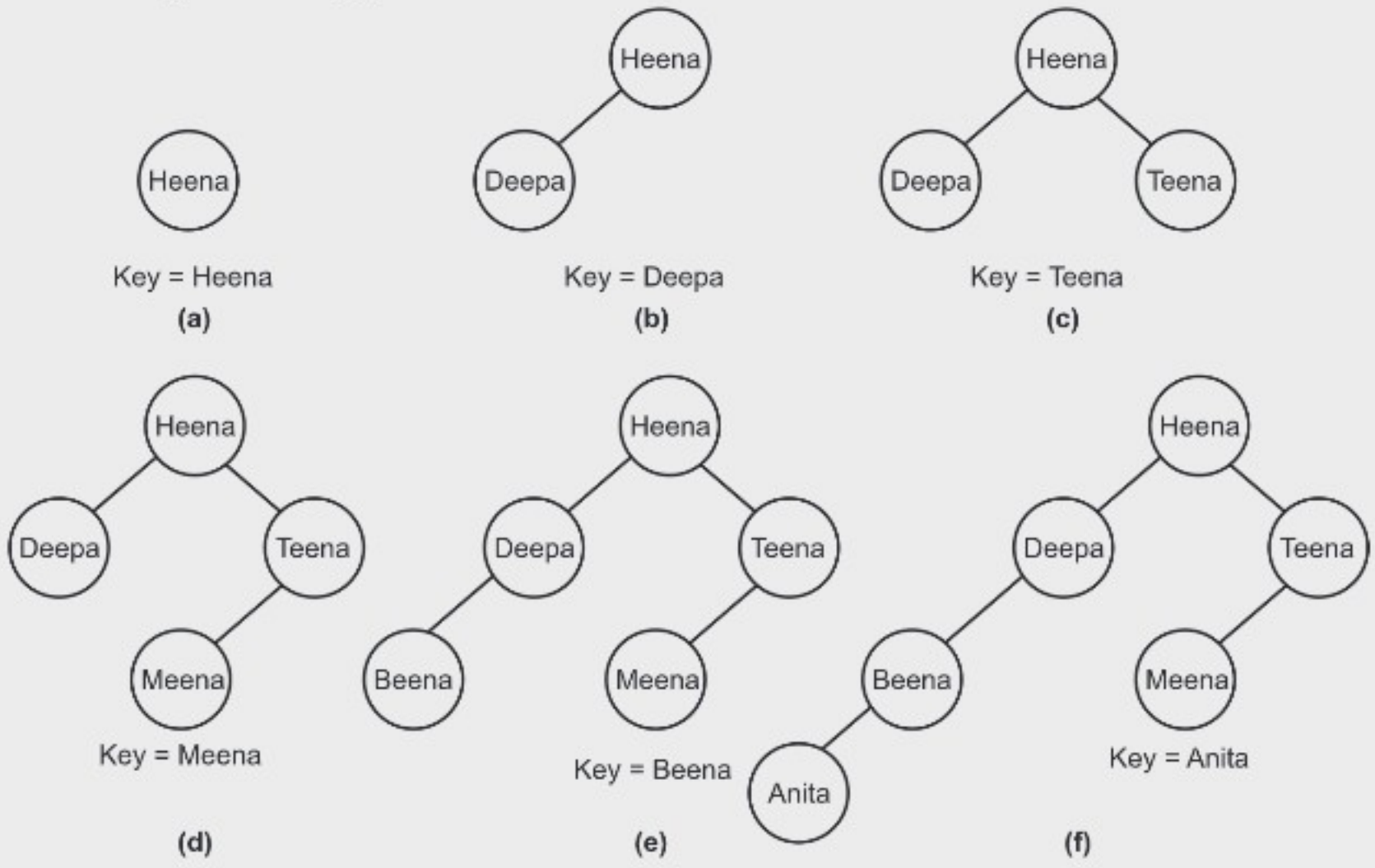


Fig. 5.30: Insertion in BST

- The first key 'Heena' is inserted, it becomes a root. Since, 'Deepa' is before 'Heena', it is inserted at left and so on for all keys.
- Now, suppose if we want to insert a key 'Geeta'. It is first compared with root. Since, 'Geeta' < 'Heena', search is proceed to left side. Since, 'Geeta' > 'Deepa' and right child of 'Deepa' is null then 'Geeta' is inserted as a right child of 'Deepa'.

5.7.1.4 Deleting Node from BST

- This is frequently used operation of BST. Let T be a BST and X is the node with key (K) to be deleted from T, if it exists in the tree let y be a parent node of X.
- There are three cases with respective to node to be deleted:
  1. X is a leaf node.
  2. X has one child.
  3. X has both child nodes.

Algorithm: Delete BST (key)

Steps:

1. t = root, flag = false
2. while (t ≠ null) and (flag = false) do

   Case 1: key < t→data
   parent = t
   t = t → left

```
Case 2:  key > t → data
         parent = t
         t = t → right
Case 3:  t → data = key
         flag = true
end case
end while
3. If flag = false
   Then display "item not exist".
   exit.
4. /* case 1 if node has no child */
   If (t → left = null) and (t → right = null) then
    if (parent → left = t) then
      parent → left = null          //set pointer to its parent when node is left child
   else
     parent → right = null
5. /* case 2 if node contains one child */
   If (parent → left = t) then       //when node contains only one child
    if (t → left = null) then        //if node is left child
       parent → left = t → right
    else
       parent → left = t → left
   endif
   else
       if (parent → right = t) then
           if (t → left = null) then
               parent → right = t → right
           else
               parent → right = t → left
           endif
       endif
   endif
6. /* Case 3 if node contains both child */
   t1 = succ(t)      //find inorder successor of the node
   key1 = t1 → data
   Delete BST (key1)  //delete inorder successor
   t → data = key //replace data with the data of an order successor
7. Stop.
```

Non-recursive C function for deleting node from BST:

```
BSTNODE *non_rec_DeleteBST (BSTNODE *root, int n)
{
    BSTNODE *temp, *parent, *child, *succ, *parsucc;
    temp==root;
    parent=NULL;
    while(temp != NULL)
    {
        if (n == temp → data)
            break;
        parent = temp;
        if(n < temp → data)
            temp = temp → left;
        else
            temp = temp → right;
        }
        if(temp == NULL)
        {
            printf("\nNumber not found.");
            return(root);
        }
        if(temp→left !=NULL && temp→right !=NULL)
        // a node to be deleted has 2 children
        {
            parsucc=temp;
            succ=temp→right;
            while(succ→left!=NULL)
            {
                parsucc=succ;
                succ=succ→left;
            }
            temp → data = succ → data;
            temp = succ;
            parent = parsucc;
        }
        if(temp→left! = NULL) //node to be deleted has left child
            child=temp→left;
        else //node to be deleted has right child or no child
            child=temp→right;
```

```
        if(parent==NULL) //node to be deleted is root node
           root=child;
        elseif(temp==parent → left) //node is left child
           parent→left=child;
        else //node is right child
           parent→right=child;
        free(temp);
        return(root);
     }
```

Recursive C function for deletion from BST:

```
   BSTNODE *rec_deleteBST(BSTNODE *root, int n)
   {
      BSTNODE *temp, *succ;
      if(root==NULL)
      {
         printf("\n Number not found.");
         return(root);
      }
      if(n<root→data)  //deleted from left subtree
         root→left=rec_deleteBST(root→left, n);
      elseif(n>root→data) //deleted from right subtree
         root→right=rec_deleteBST(root→right, n);
      else  //Number to be deleted is found
      {
         if(root→left ! = NULL && root→right ! =NULL)//2 children
         {
            succ=root→right;
            while(succ→left)
            succ=succ→left;
            root→data=succ→data;
            root→right=rec_deleteBST(root→right, succ→data);
         }
         else
         {
            temp=root;
            if(root→left !=NULL)  //only left child
               root=root→left;
            elseif (root→right!=NULL) //only right child
               root=root→right;
```

```
            else  //no child
               root=NULL;
            free(temp);
         }
      }
      return(root);
   }
```

Example 8: Consider all above three cases.

Case 1: If the node to be deleted is a leaf node, then we only replace the link to the deleted node by null.

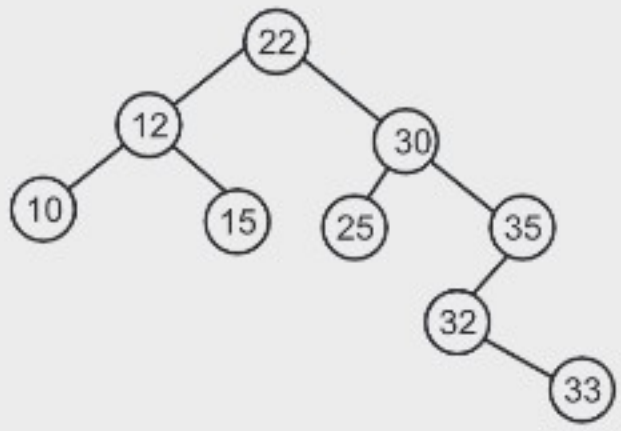


Fig. 5.31: Binary search tree

After deletion of leaf node with data = 33.

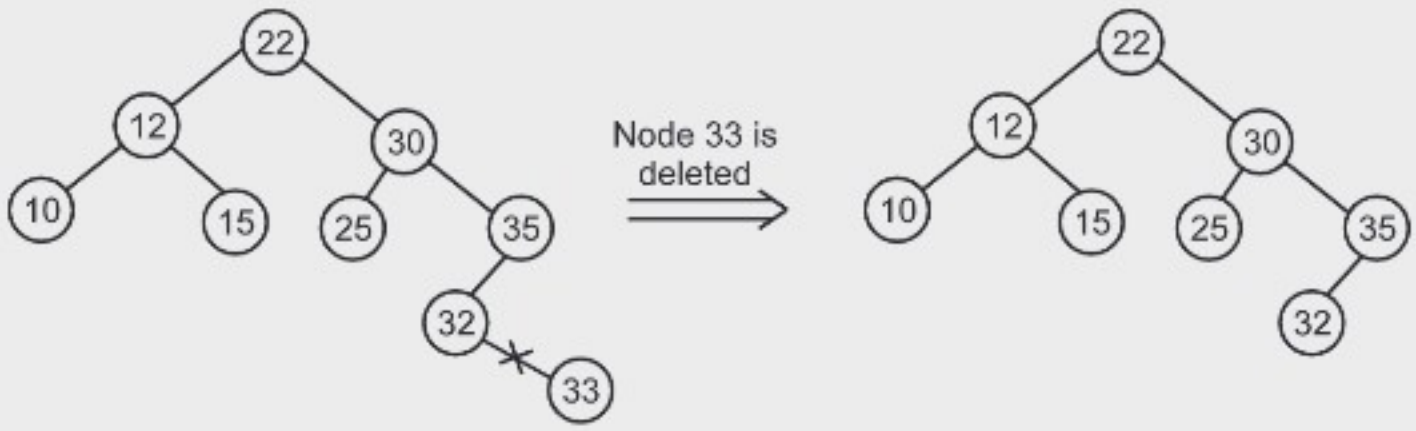


Fig. 5.32: Deletion of node = 33

Case 2: If the node to be deleted has a single child node, then we adjust the link from parent node to point to its subtree.

Consider tree of Fig. 5.33, delete the node with data = 35 which has single child = 32.

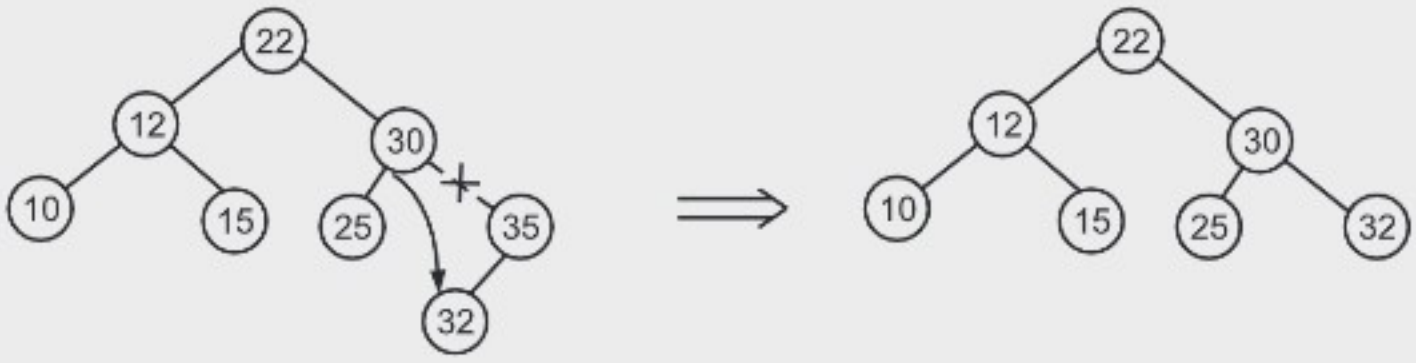


Fig. 5.33: Deletion of node = 35

Case 3: The node to be deleted having both child nodes. When the node to be deleted has both non-empty subtrees, the problem is difficult. One of the solution is to attach the right subtree in place of the deleted node and then attach the left subtree to the appropriate node of right subtree.

From Fig. 5.34, we delete the node = 12.

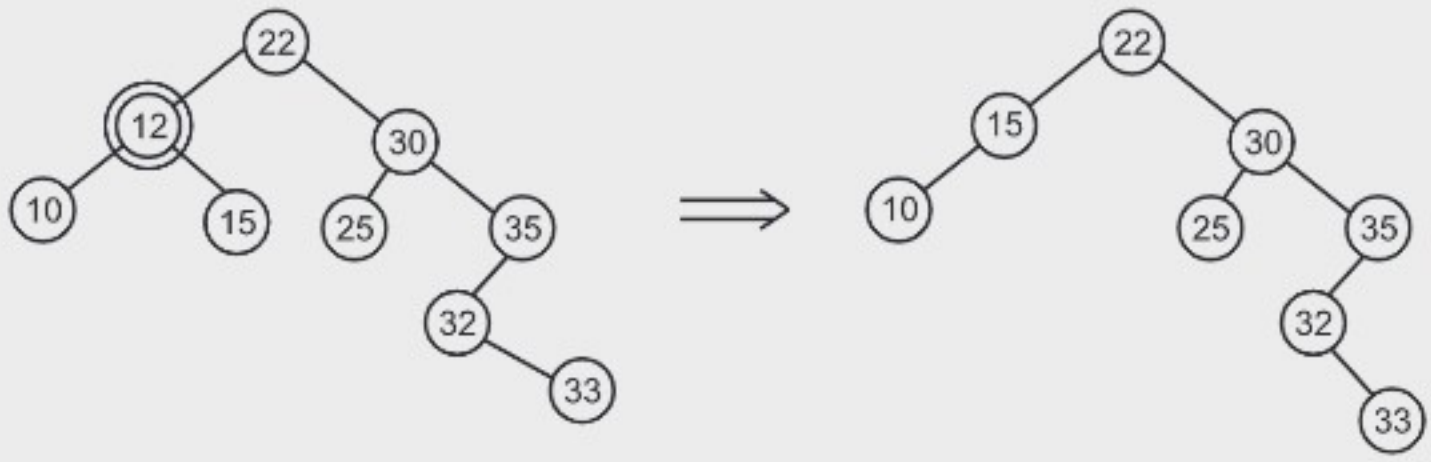


Fig. 5.34: Deletion of node = 12

Pictorial representation is,

Fig. 5.35: Pictorial representation

The another approach is to delete x from T by first deleting inorder successor of node x say z, then replace the data content in node x by the data content in node z. In order successor means the node which comes after node x during the inorder traversal of T.

Example 9: Consider Fig. 5.36 and delete node = 15

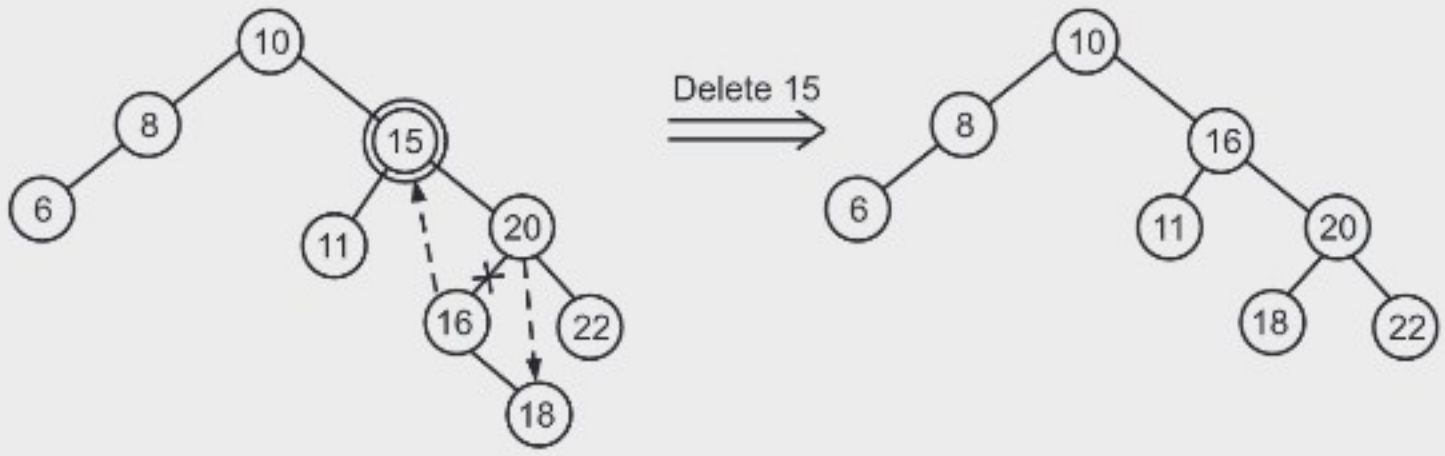


Fig. 5.36: Deleting node = 15

Program 5.1: Write a program in 'C' for inserting the elements in the Binary Search Tree and display them in preorder, inorder and postorder.

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   struct node *left;
   int data;
   struct node *right;
};
struct node * insert(struct node * p,int d)
{
   if (p==NULL)
   {
      p=(struct node *) malloc(sizeof(struct node));
      p -> left = p -> right = NULL;
      p -> data = d;
   }
   else
   {
      if (d < p -> data)
         p -> left = insert(p -> left, d);
      else
         if(d > p -> data)
         p -> right = insert(p -> right, d);
      }
   return(p);
}
struct node * search(struct node *p,int x)
{
   if(p != NULL)
   {
      if (p -> data == x)
      return p;

      if(x < p -> data)
          return (search(p -> left, x));
      else
          return (search(p -> right, x));
   }
}
```

```
void preorder(struct node *p)
{
    if(p!=NULL)
    {
        printf("%d \t", p -> data);
        preorder(p -> left);
        preorder(p -> right);
    }
}
void inorder(struct node *p)
{
    if(p!=NULL)
    {
        inorder(p -> left);
        printf("%d \t", p -> data);
        inorder(p -> right);
    }
}
void postorder(struct node *p)
{
    if(p!=NULL)
    {
        postorder(p -> left);
        postorder(p -> right);
        printf("%d \t", p -> data);
    }
}
int main()
{
    int ch,x,ans;
    struct node *tree=NULL,*temp=NULL;
    do
    {
        printf ("\n1) Insert\n2) Search\n3) Preorder\n4)
                                                Inorder\n5) Postorder\n");
        printf("enter your choice\n");
        scanf("%d", &ch);
```

```
        switch(ch)
        {
           case 1 :  printf("enter data to insert into tree\n");
                     scanf("%d", &x);
                     tree=insert (tree, x);
                     break;
           case 2:   if(tree == NULL)
                        printf("\n Tree is NULL \n");
                     else
                     {
                        printf("enter key to be search in tree\n");
                        scanf("%d", &x);
                        temp=search(tree,x);
                        if(temp != NULL)
                              printf(" \n %d Key is found \n",temp->data);
                        else
                              printf(" \n Key is not found \n");
                     }
                     break;
           case 3:   if(tree == NULL)
                        printf("\n Tree is NULL \n");
                     else
                        preorder(tree);
                     break;
           case 4:   if(tree == NULL)
                        printf("\n Tree is NULL \n");
                     else
                        inorder(tree);
                     break;
           case 5:   if(tree == NULL)
                        printf("\n Tree is NULL \n");
                     else
                  postorder(tree);
                  break;
        }
     }while(ch>0 && ch <6);
  }
```

5.8 C FUNCTIONS FOR BINARY TREE

5.8.1 Computing the Height of a Binary Tree

Algorithm:

Step 1 : If node is NULL then return 0.

Step 2 : We use a recursive approach to find the maximum of left subtree height and right subtree height and add 1 for the parent node. This process continues till we reach the deepest leaf node.

- Recursive function which returns the height of linked binary tree:

```
int tree_height(struct node *root)
{
   if (root == NULL)
      return 0;
   else
      return 1 + max (tree_height (root → left),tree_height (root →
      right));
}
```

- Here, function max() is defined as follows:

```
int max(int x, int y)
{
   int max;
      if (x > y)
         max = x;
      else
         max = y;
   return max;
}
```

5.8.2 Counting total Nodes in a Binary Tree [W-18]

- To count total nodes in a tree, we traverse a tree. We can use any of the three traversal methods.

Algorithm:

Step 1 : If node is NULL then return 0.

Step 2 : Else recursively count total nodes in binary tree using following formula.

count = 1 + count nodes of left subtree + count nodes of right subtree

Step 3 : return count

'C' function to count total nodes of a binary tree:

```
int CountNode(struct node *root)
{
    static int count;
    if (root== NULL)
        return 0;
    else
        count=1+CountNodes(root → left)+CountNodes(root → right);
    return count;
}
```

5.8.3 Counting Leaf Nodes of a Binary Tree [S-18]

- The node which do not have child node is called as leaf node.

Algorithm:

Step 1 : If node is NULL then return 0.

Step 2 : Else If left and right child nodes are NULL return 1.

Step 3 : Else recursively calculate leaf count of the tree using below formula.

Leaf count of a tree = Leaf count of left subtree + Leaf count of right subtree

'C' function to count leaf nodes of a binary tree:

```
int CountLeaf(Tree * root)
{
    if (root==NULL)
        leafcount = 0;
    else
          if ((root → left == NULL) && (root → right == NULL))
               return(1);
          else
               return((CountLeaf (root → left) + CountLeaf (root → right));
}
```

5.8.4 Counting Non-Leaf Nodes of a Binary Tree

- The node which have at least one child is called as non-leaf node. We recursively traverse the given tree. While traversing, we count non-leaf nodes in left and right subtrees and add 1 to the result.

Algorithm:

Step 1 : If node is NULL then return 0.

Step 2 : Else If left and right child nodes are NULL return 1.

Step 3 : Else recursively calculate non-leaf count of the tree using below formula.

Non-leaf count of a tree = 1 + Non-leaf count of left subtree + Non-leaf count of right subtree

'C' function to count non-leaf nodes of a binary tree:

```
int count_non_leaf(struct node * root)
{
if(root == NULL)
      return 0;
if(root -> left == NULL && root -> right == NULL)
      return 0;
return(1 + count_non_leaf(root -> left) + count_non_leaf(root->right));
}
```

5.8.5 Mirror image of a Binary Tree [S-19]

- The mirror image of a tree contains its left and right subtrees interchanged as shown in Fig. 5.37.

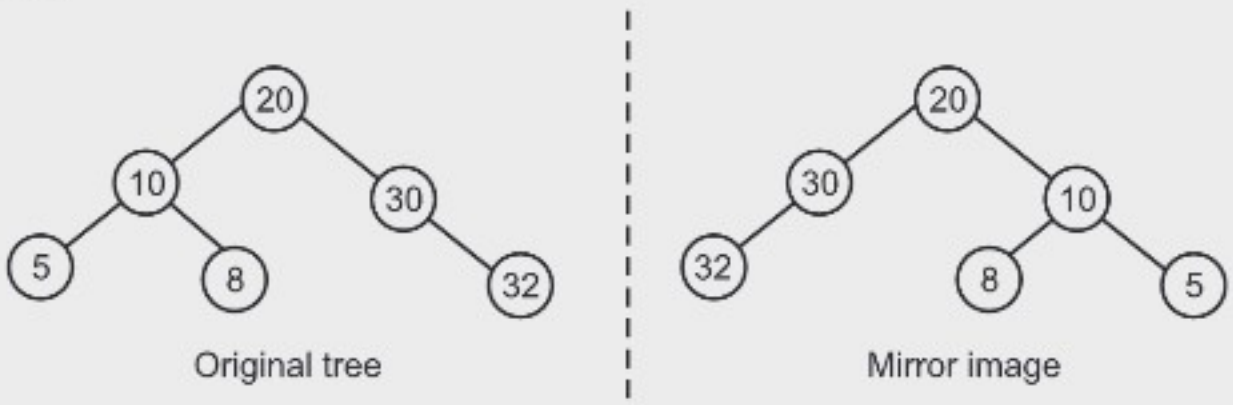


Fig. 5.37: Mirror image of a binary tree

- Here, we start with lower level (leaves) and move towards upwards till the children of the root are interchanged.

Recursive Algorithm:

Step 1 : Call Mirror for left-subtree i.e., Mirror(left-subtree)

Step 2 : Call Mirror for right-subtree i.e., Mirror(right-subtree)

Step 3 : Swap left and right subtrees.
temp = left-subtree
left-subtree = right-subtree
right-subtree = temp

'C' function to find mirror of binary tree:

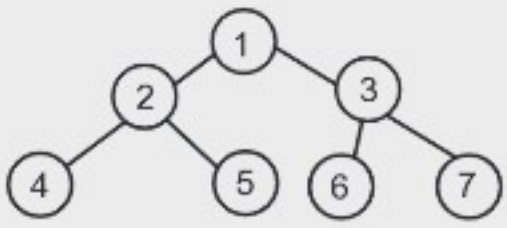
```
struct node * mirror(struct node * root)
{
   struct node * temp=NULL;
   if (root != NULL)
   {
       temp = root → left;
       root → left = mirror(root → right);
       root → right = mirror(temp);
   }
   return root;
}
```

Summary

- Tree is a non-linear data structure.
- Tree is a acyclic graph.
- Tree data structure is a collection of data (Node) which is organized in hierarchical structure recursively.
- In any tree with N nodes there will be maximum of 'N-1' edges.
- In a tree an Edge is connecting link between two nodes.
- Every non-leaf node is called as Internal node.
- Every leaf node is called as External node.
- Degree of node is total number of children it has.
- Height of all leaf nodes is '0'.
- A binary tree with n internal nodes (non-leaf node) has n+1 external nodes (leaf node).
- In binary tree every node except the root has exactly one parent.
- Binary tree can be represent using array or linked list.
- There are three standard way for traversing a binary tree: preorder, inorder and postorder.
- Preorder Traversal: Process all nodes of a tree by processing the root, then recursively processing all subtrees.
- Inorder Traversal: Process all nodes of a tree by recursively processing the left subtree, then processing the root, and finally the right subtree.
- Postorder Traversal: Process all nodes of a tree by recursively processing all subtrees, then finally processing the root.
- Inorder traversal gives nodes in non-decreasing order.
- Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree.
- Postorder traversal is also useful to get the postfix expression of an expression tree.
- We can reconstruct a binary tree from its inorder and preorder traversal.
- A binary search tree (BST), also known as an ordered binary tree, is a node-based data structure in which each node has no more than two child nodes. Each child must either be a leaf node or the root of another binary search tree. The left sub-tree contains only nodes with keys less than the parent node; the right sub-tree contains only nodes with keys greater than the parent node.
- The operations that can be performed on a binary tree are insertions, deletions, searching, traversing, computing height, counting leaf nodes, non-leaf nodes and total nodes, mirroring the tree.

Check Your Understanding

1. Children of same parent are called _____.
   (a) Node (b) Child node
   (c) Sibling (d) None of the above
2. The number of edges from the root to the node is called _____ of the tree.
   (a) Height (b) Depth
   (c) Length (d) Width
3. The number of edges from the node to the deepest leaf is called _____ of the tree.
   (a) Height (b) Depth
   (c) Length (d) Width
4. Consider the following tree:



5. If the postorder traversal gives (ab – cd*+) then the label of the nodes 1, 2, 3, 4, 5, 6 will be _____.
   (a) +, –, *, a, b, c, d (b) a, –, b, +, c, *, d
   (c) a, b, c, d, –, *, + (d) –, a, b, +, *, c, d
6. The indegree of _____ of a tree is always zero.
   (a) any node (b) a branch node
   (c) the root node (d) a leaf node
7. A list of integers is read in, one at a time and a binary search tree is constructed. Next the tree is traversed and the integers are printed. Which traversal would print result in the original order of input?
   (a) preorder (b) postorder
   (c) inorder (d) none of the above
8. A binary tree T has n leaf nodes. The number of nodes of degree 2 in T is _____.
   (a) $\log_2 n$ (b) $n - 1$
   (c) n (d) 2^n
9. A binary tree in which every non-leaf node has non-empty left and right subtrees is called a strictly binary tree. Such a tree with 10 leaves _____.
   (a) cannot have more than 1g nodes (b) has exactly 19 nodes
   (c) has exactly 17 nodes (d) cannot have more than 17 nodes

10. The depth of complete binary tree with n nodes is _____.
    (a) $\log_2 (n + 1) - 1$ (b) $\log_2 n$
    (c) $\log_2 (n - 1) + 1$ (d) $\log_2 n + 1$
11. The number of binary trees with 3 nodes which when traversed in postorder gives the sequence A, B, C is _____.
    (a) 3 (b) 9
    (c) 7 (d) 5

Answers

1. (c)	2. (b)	3. (a)	4. (a)	5. (c)	6. (d)	7. (b)	8. (b)	9. (a)	10. (d)	11. (d)

Practice Questions

Q.I Answer the following Questions in short:

1. What is tree?
2. What are the different terminologies of tree?
3. What is mean by Degree of node?
4. List out various types of binary tree.
5. What is mean by complete binary tree?

Q.II Answer the following Questions:

1. Construct the 14 binary tree with four nodes.
2. Explain different types of binary tee with suitable example.
3. What are the different tree traversal methods? Explain with example.
4. Can the binary tree be constructed naively, when inorder and inorder and postorder traversal sequences are given?
5. What is BST? Construct BST for the following data:
   MON, SUN, TUE, FRI, SAT, WED, THUS.
6. Construct BST for the following data:
   (a) 16, 12, 14, 9, 10, 18
   (b) 11, 7, 15, 25, 18, 5, 12, 20
7. Write C function to count leaf nodes of a binary tree.
8. Write C function to count depth of binary tree.
9. Write an algorithm to count the total number of nodes in Binary tree.
10. Write a recursive function in C that creates a mirror image of a binary tree.
11. Write a function to count the number of leaf nodes of a given tree.
12. Write a 'C' function to print minimum and maximum element from a given binary search tree.
13. Write C function to insert node into Binary search tree.
14. Write an algorithm to count leaf nodes in a tree.

Q.III Define following terms:

(i) Leaf node

(ii) Height of tree

(iii) Level of tree

(iv) Complete binary tree

(v) Full binary Tree

(vi) Binary search tree

(vii) Strictly binary tree

Previous Exam Questions

Summer 2019

1. _____ is non-linear data structure. **[1 M]**

(i) Strings (ii) Stacks

(iii) Trees (iv) Array

Ans. (iii)

2. Children of the same parent are called _____. **[1 M]**

(i) Node (ii) Child node

(iii) Siblings (iv) Node of the above

Ans. (iii)

3. What is mirroring the given tree means? Explain it with example. **[3 M]**

Ans. Refer to Section 5.8.5.

4. Define the following terms: **[2 M]**

(i) Strictly binary tree

Ans. Refer to Section 5.3.

(ii) Height of tree

Ans. Refer to Section 5.2.2.

5. Write a function in C to insert a node in binary search tree. **[4 M]**

Ans. Refer to Section 5.7.1.3.

Winter 2018

1. A full binary tree with 2n + 1 nodes contain: **[1 M]**

(i) n leaf nodes (ii) n non-leaf nodes

(iii) n – 1 leaf nodes (iv) n – 1 non-leaf nodes

Ans. (ii)

2. Define Leaf Nodes. **[1 M]**

Ans. Refer to Section 5.2.2

3. Write an algorithm to count the number of nodes in Binary tree. **[4 M]**

Ans. Refer to Section 5.8.2

4. What is Expression Tree? Construct expression tree for the following: **[3 M]**

(A + B) * (C – D)

Ans. Refer to Section 5.6

5. Explain different types of Binary tree with suitable example. **[4 M]**

Ans. Refer to Section 5.3

6. What is BST? Construct BST for the following data: **[5 M]**

MON, SUN, TUE, FRI, SAT, WED, THUS.

Ans. Refer to Example 6

Summer 2018

1. The indegree of _____ of a tree is always zero. **[1 M]**

(i) any node (ii) branch mode

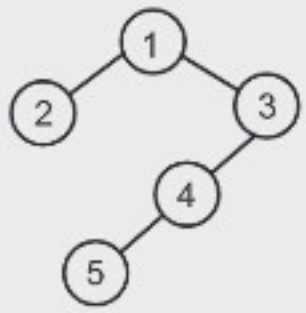
(iii) the root node (iv) a leaf node

Ans. (iii)

2. Traverse the following tree in: **[1 M]**

(i) Preorder

(ii) Inorder



Ans. Refer Example 4

3. What are the different tree traversal methods. Explain with example. **[5 M]**

Ans. Refer to Section 5.6.

4. Write the function to count leaf nodes of a binary tree. **[4 M]**

Ans. Refer to Section 5.8.3.

5. Construct a Binary search tree for the following elements: **[3 M]**

16, 12, 14, 9, 10, 18

Ans. Refer Example 6

■■■

6... Graphs

Objectives...

- To study basic concepts of Graph.
- To understand basic Graph Terminology.
- To learn representation of Graph.
- To study different Operations of Graph.
- To learn Applications of Graphs.

6.1 INTRODUCTION

- In many problems relationships among the data is complicated than those represented by trees or list. Graph is one of the powerful and natural model to represent such relationships. Thus, the graph structure can be used to solve many real life problems. Graph is used to solve concrete physical problem very fast.

6.1.1 Definition of Graph [S-19, W-18]

- A graph G is a set of two tuples G = (V, E), where V is a finite non-empty set of vertices and E is the set of pairs of vertices called edges.

  Example:

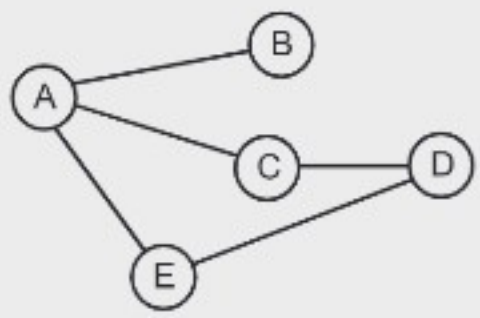


Fig. 6.1: Graph

The set of vertices in the above graph is, V = {A, B, C, D, E}

The set of edges, E = {(A, B), (A, C), (C, D), (A, E), (E, D)}

[S-19]

- **Multigraph**: A graph in which multiple edges may connect the same pair of vertices is called a multigraph.

  Since there can be multiple edges between the same pair of vertices, the multiplicity of edge tells the number of edges between two vertices.

6.1.2 Types of Graph

- There are two types of graph:
  - Undirected graph
  - Directed graph

1. Undirected Graph:

- In an undirected graph, the pair of vertices representing any edge is unordered i.e. the pairs (v1, v2) and (v2, v1) represent the same edge. In other words the edges have no direction in undirected graph.

Example:

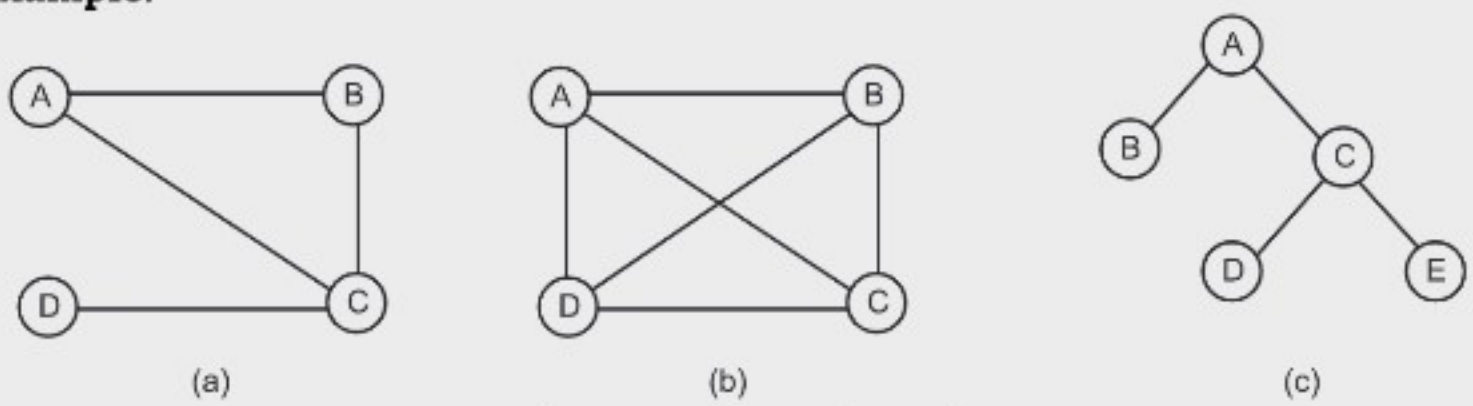


Fig. 6.2: Undirected graph

In Fig. 6.2 (a) G = (V, E)

where, V = {A, B, C, D} and E = {(A, B), (A, C), (B, C), (C, D)}

In Fig. 6.2 (b) G = (V, E)

where, V = {A, B, C, D} and E = {(A, B), (A, C), (A, D), (B, D), (B, C), (D, C)}

In Fig. 6.2 (c) G = (V, E)

where, V = {A, B, C, D, E} and E= {(A, B), (A, C), (C, D), (C, E)}

This undirected graph is called Tree. Tree is a special case of graph.

2. Directed Graph:

- In a directed graph each edge is represented by a directed pair (v1, v2), v1 is the tail and v2 is head of the edge i.e. the pairs (v1, v2) and (v2, v1) are different edges. In other words the edges have direction in directed graph. Directed graph is also called as Digraph.
- Fig. 6.3 shows the three directed graphs.

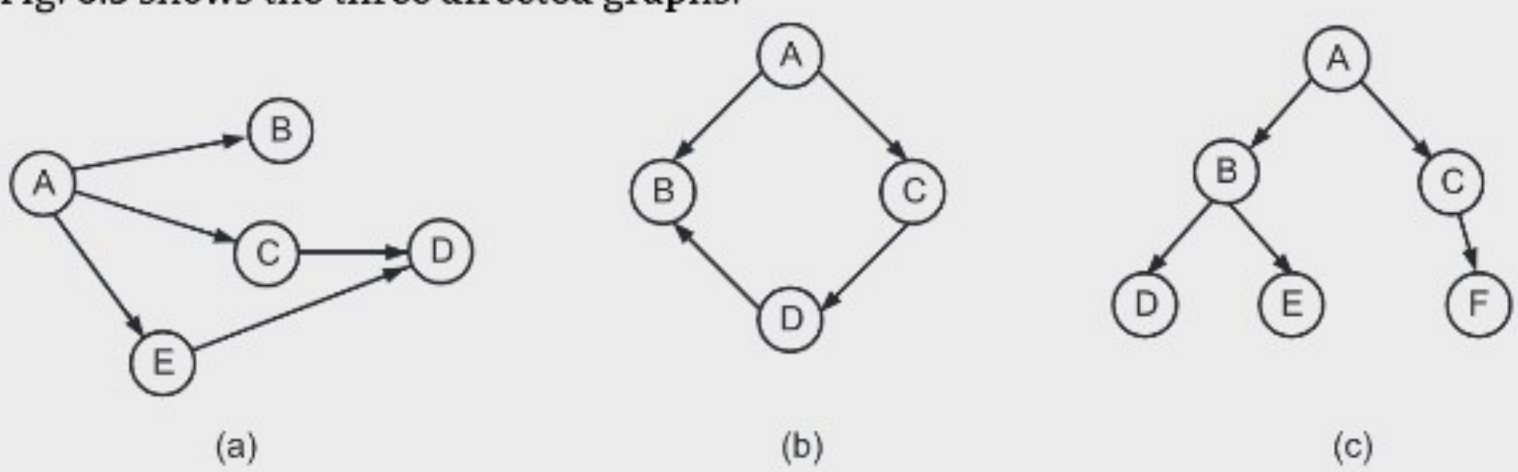


Fig. 6.3: Directed graphs

In Fig. 6.3 (a) G = (V, E)
where, V = {A, B, C, D, E} and E = {(A, B), (A, C), (C, D), (A, E), (E, D)}
In Fig. 6.3 (b) G = (V, E)
where, V = {A, B, C, D} and E = {(A, B), (A, C), (C, D), (D, B)}
In Fig. 6.3 (c) G = (V, E)
where, V = {A, B, C, D} and E = {(A, B), (A, C), (B, D), (B, E),(C, F)}

6.2 GRAPH TERMINOLOGY

1. **Adjacent Vertex:** When there is an edge from one vertex to another then these vertices are called adjacent vertices.
2. **Cycle:** A path from a vertex to itself is called a cycle. Thus, a cycle is a path in which the initial and final vertices are same.
   For example: Fig. 7.4 shows the path (A, B, C, A) or (A, C, D, B, A) are cycles of different lengths. If a graph contains a cycle it is cyclic otherwise acyclic.

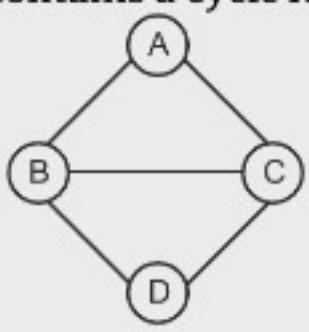


Fig. 6.4: Cycle

[S-19, 18; W-18]

3. **Complete Graph:** A graph G is said to be complete if every vertex in a graph is adjacent to every other vertex. In this graph, number of edges = n(n – 1)/2, where n = no. of vertices.

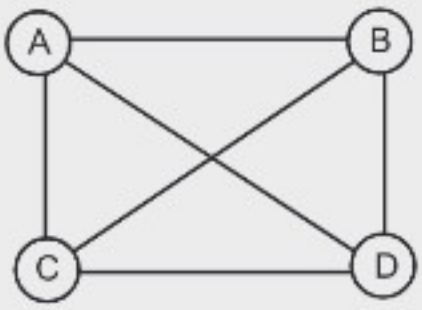
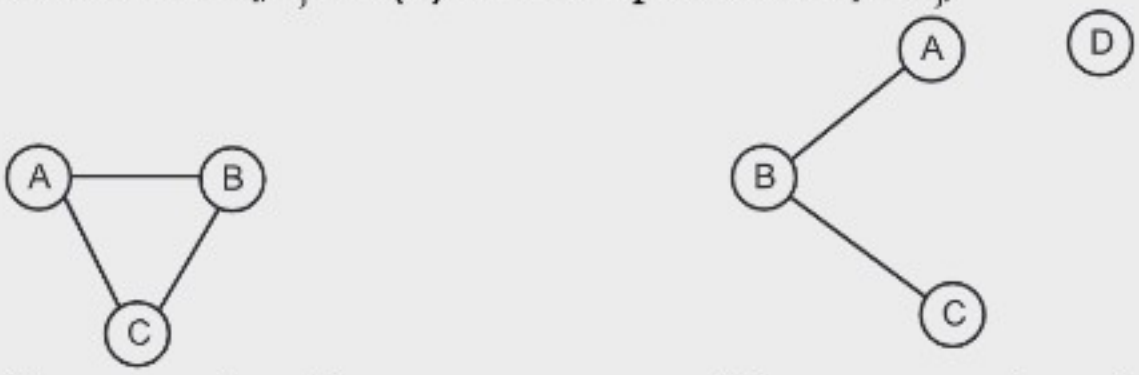


Fig. 6.5: Complete graph

Example: In Fig. 6.5, No of vertices n = 4, Number of edges = 4(4 – 1)/2 = 6.

4. **Connected Graph:** An undirected graph G is said to be connected if for every pair of distinct vertices v_i, v_j in V(G) there is a path from v_i to v_j.



(a) connected graph **(b) non-connected graph**

Fig. 6.6

5. **Degree of a vertex:** It is the number of edges incident to a vertex. It is written as deg(U), where U is a vertex.
6. **In-degree of a vertex:** In directed graph, in-degree of a vertex 'v' is the number of edges for which 'v' is the head.

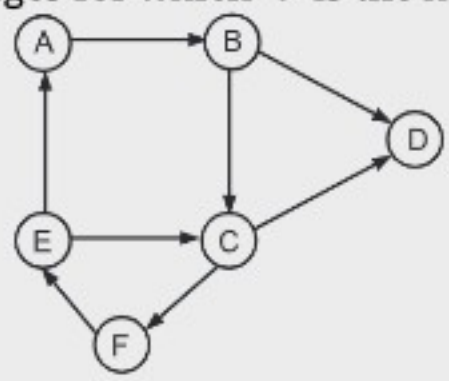


Fig. 6.7: Digraph

Indegree of vertex B = 1
Indegree of vertex C = 2
Indegree of vertex D = 2
Indegree of vertex E = 1
Indegree of vertex F = 1
Indegree of vertex A = 1

7. **Out-degree of a vertex:** In directed graph, the out-degree of a vertex 'v' is the total number of edges for which 'v' is the tail.

   From Fig. 6.7,

   Outdegree of A = 1
   Outdegree of B = 2
   Outdegree of C = 2
   Outdegree of D = 0
   Outdegree of E = 2
   Outdegree of F = 1
8. **Isolated Vertex:** If any vertex does not belong to any edge then it is called isolated vertex.

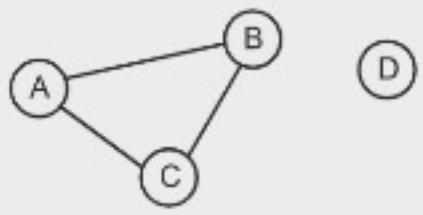


Fig. 6.8: Isolated node D in the graph

9. **Source vertex:** A vertex with in-degree zero is called a source vertex, i.e. vertex has only outgoing edges and no incoming edges. For example, in Fig. 6.9, 'C' is source vertex.
10. **Sink vertex:** A vertex with out-degree zero is called a sink vertex i.e. vertex has only incoming edge and no outgoing edge. For example, in Fig. 6.9, 'B' is sink node.
11. **Acyclic graph:** A graph without cycle is called acyclic graph.

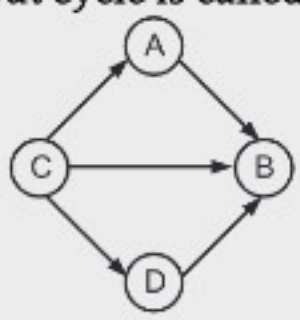


Fig. 6.9: Acyclic graph

12. Subgraph: A subgraph of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.

Fig. 6.10

13. Weighted graph: A weighted graph is a graph in which every edge is assigned a weight. In Fig. 6.11 weight in a graph denote the distance between the two vertices connected by the corresponding edge. The weight of an edge is also called its cost. In case of a weighted graph, an edge is a 3-tuple (U, V, W) where U and V are vertices and W is weight of an edge (U, V).

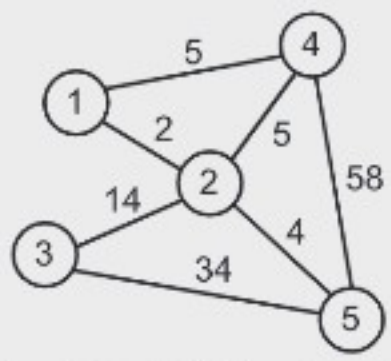


Fig. 6.11: Weighted graph

14. Pendant Vertex: When in-degree of vertex is one and out-degree is zero then such a vertex is called pendant vertex.

15. Spanning Tree: A spanning tree of a graph G=(V, E) is a subgraph of G having all vertices of G and containing only those edges that are necessary to join all the vertices in the graph. A spanning tree does not contain cycle in it.

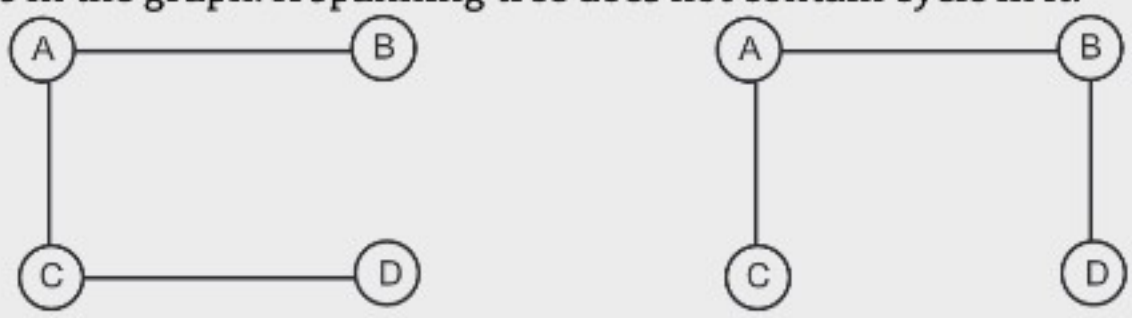
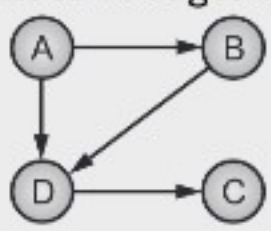


Fig. 6.12: Spanning trees

16. Path: A path is a sequence of distinct vertices, each adjacent to the next. The length of such a path is number of edges on that path. **[S-18]**



Path from A to C is

$A \rightarrow B \rightarrow D \rightarrow C$

Length of this parth = 3

Fig. 6.13: Path

6.3 REPRESENTATION OF GRAPH [S-19, W-18]

- Graph is represented by the following three ways:
  1. Sequential representation (by means of Adjacency matrix).
  2. Linked representation (by means of Linked list).
  3. Adjacency Multi-list representation.

6.3.1 Sequential Representation of Graph

- The graph when represented using sequential representation using matrix, is commonly known as **Adjacency matrix.**

1. Adjacency matrix:

- Let G=(V, E) be a graph with n vertices, n>=1. An adjacency matrix of G is a 2-dimentional $n \times n$ array, say A, with the following property:

$$A[i][j] = 1 \text{ if the edge } (v_i, v_j) \text{ is in } E(G)$$
$$= 0 \text{ if there is no such edge in } G$$

- If the graph is undirected then

$$A[i][j] = A[j][i] = 1$$

Example 1: Construct adjacency matrix for following undirected graphs.

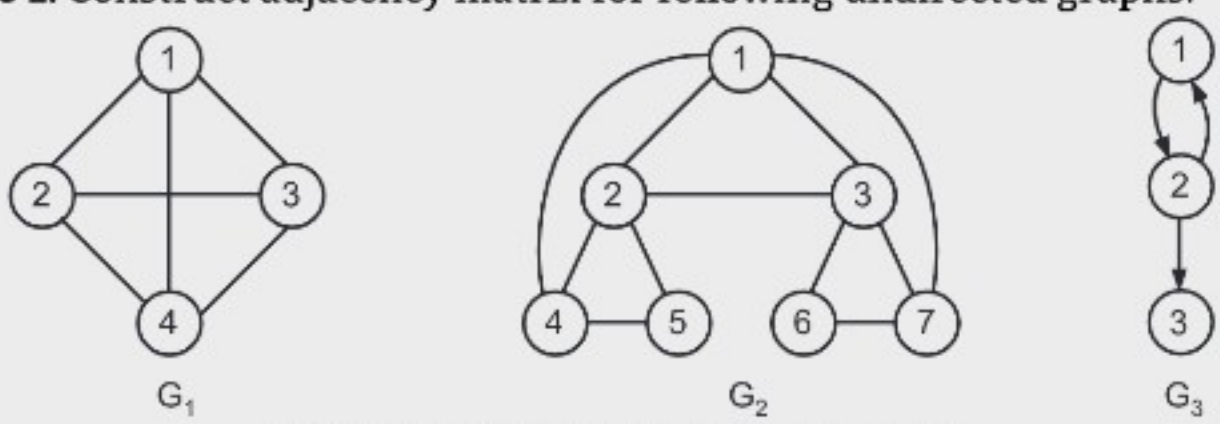


Fig. 6.14: Undirected Graph G_1, G_2 and G_3

- The graphs G_1, G_2 and G_3 of Fig. 6.13 are represented using adjacency matrix,

	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

G1

	1	2	3
1	0	1	0
2	1	0	1
3	0	0	0

G3

	1	2	3	4	5	6	7
1	0	1	1	1	0	0	1
2	1	0	1	1	1	0	0
3	1	1	0	0	0	1	1
4	1	1	0	0	1	0	0
5	0	1	0	1	0	0	0
6	0	0	1	0	0	0	1
7	1	0	1	0	0	1	0

G_2

Adjacency matrix representation of a weighted graph:

For weighted graph, the matrix A is represented as,

A[i] [j] = weight of the edge (i, j)
= 0 otherwise

Here weight is label associated with edge.

Example 2: Construct adjacency matrix for following weighted graph.

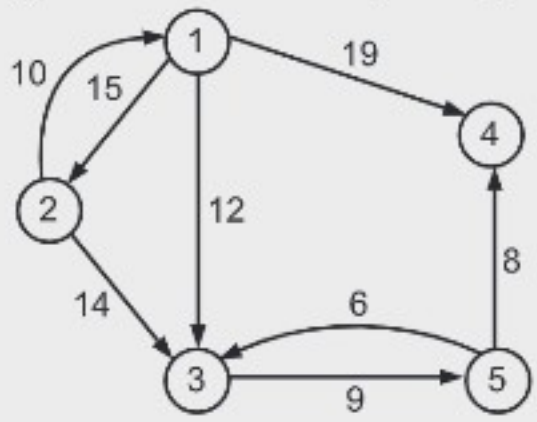


Fig. 6.15

	1	2	3	4	5
1	0	15	12	19	0
2	10	0	14	0	0
3	0	0	0	0	9
4	0	0	0	0	0
5	0	0	6	8	0

Note: The adjacency matrix for an undirected graph is symmetric and the adjacency matrix for directed graph need not be symmetric.

- The space needed to represent a graph using adjacency matrix is n^2, where n is number of vertices.
- For undirected graph we can store only the upper or lower triangle of the matrix, as the matrix is symmetric.
- Finding all the neighbors of a vertex requires searching a whole row of matrix i.e. requires O(n) of time.
- When the graph is sparse, we requires $O(n^2)$ of time.

6.3.2 Linked Representation of Graphs

- **Drawback of adjacency matrix representation:**
  - When a graph is sparse i.e. when most of the vertices have few neighbors, very few entries are non-zero.
  - In sequential representation it is difficult to insert and delete nodes in G, since, the size of matrix A need to be change; so there may be many changes in matrix A.
- So the graph G is represented in memory using linked representation.
- In this representation, the n rows of the adjacency list are represented as n linked lists; one list per vertex of graph.

1. **Adjacency list representation for directed graph:**

- Consider the graph G. The Fig. 6.16(a) shows the graph G and linked representation of G in memory. The linked representation will contain two lists:
  1. **A node vertex list:** To keep the track of all the N nodes on the graph.
  2. **An edge list:** To keep the information of adjacent vertices of each and every vertex of a graph.
- Head node is used to represent each list, i.e. we can represent G by an array Head, where Head [i] is a pointer to the adjacency list of vertex i.

Example 3: Construct adjacency list for following directed graph.

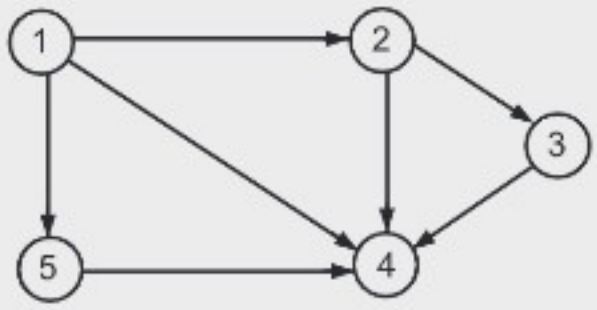


Fig. 6.16 (a)

Solution:

Vertex and its Adjacent Vertices	
Vertex	**Adjacency Vertices**
1	2, 4, 5
2	3, 4
3	4
4	–
5	4

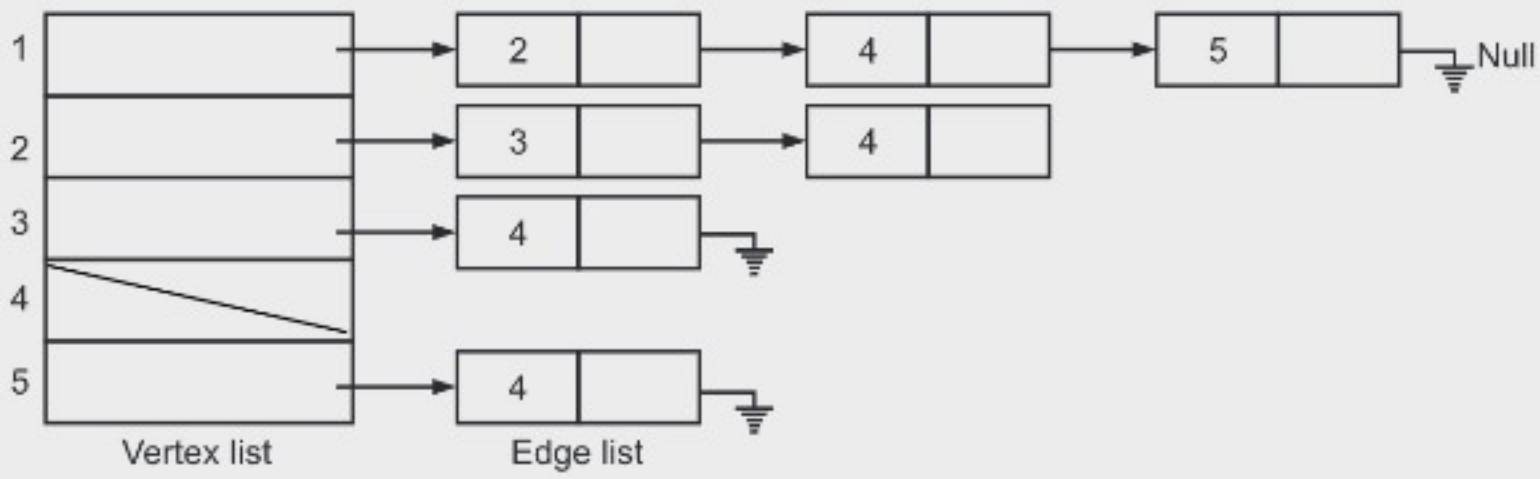


Fig. 6.16 (b): Adjacency list representation (directed graph)

2. Adjacency list representation for undirected graph:

- In this representation, the n rows of the adjacency matrix are represented as n linked lists. The nodes in list i represent the vertices that are adjacent to vertex i. Head node is used to represent each list.

Example 4: Construct adjacency list for following undirected graph.

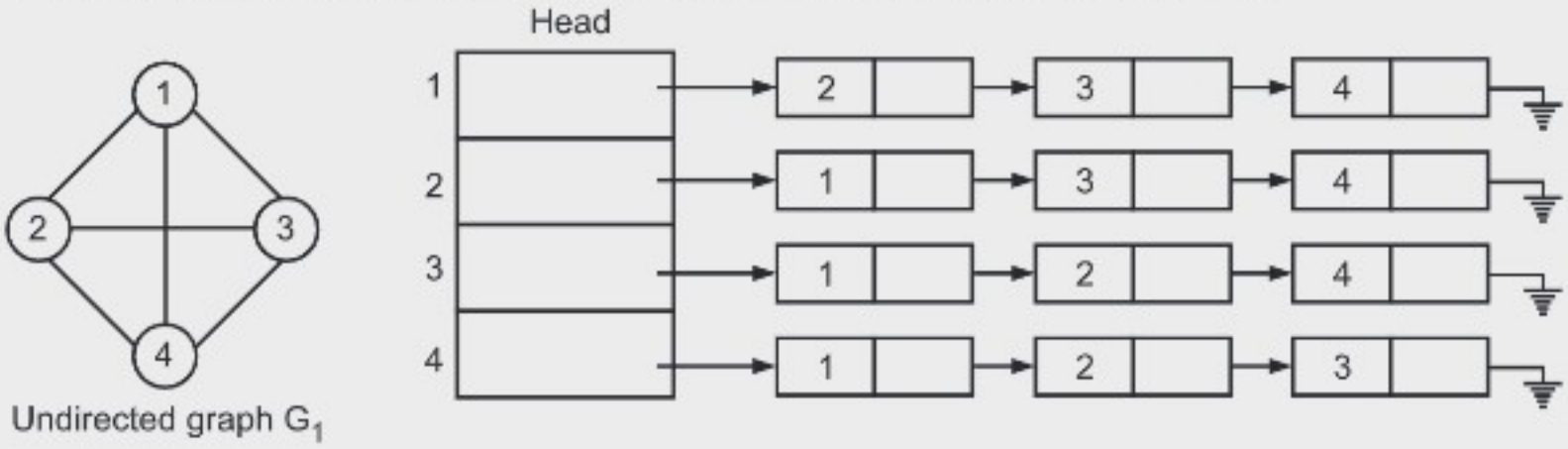


Fig. 6.17: Adjacency list representation of G_1 (Undirected Graph)

- **The structure for adjacency list representation can be defined in C as follows:**

```
#define MAX_V 20
struct node
{
    int vertex;
    struct node * next;
}
struct node head[MAX_V];
```

- In this representation every edge (v_i, v_j) of an undirected graph is represented twice, once in the list of v_i and in the list of v_j.
- For directed graph time required is O(n) to determine whether, there is an edge from vertex i to vertex j and for undirected graph time is O(n + e).

Example 5: Give adjacency list representation of following graph.

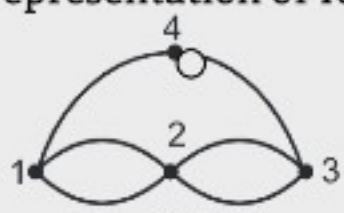


Fig. 6.18 (a)

Solution:

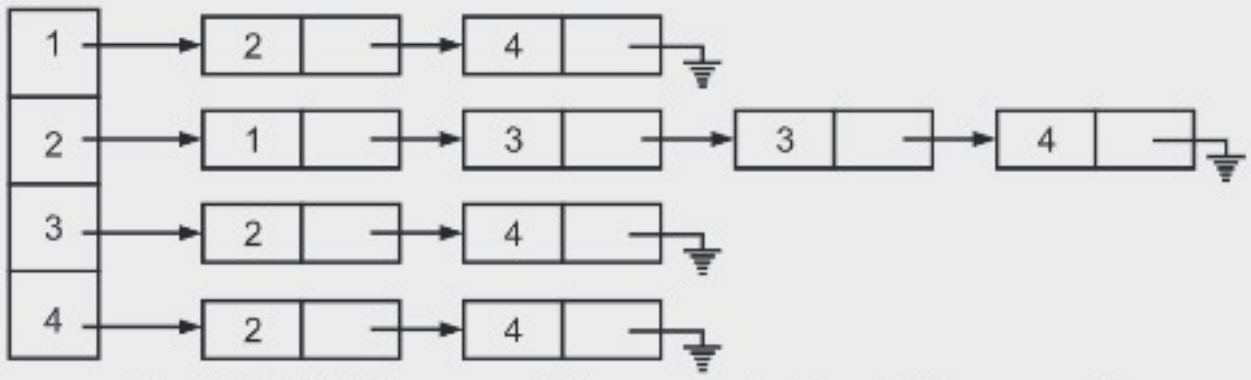


Fig. 6.18 (b): Adjacency list representation of above graph

Example 6: Give adjacency list representation of following graph.

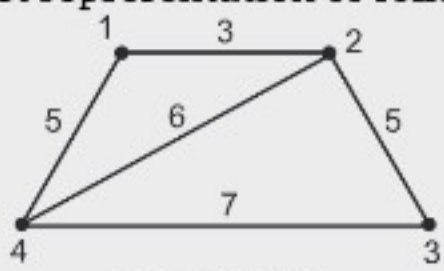


Fig. 6.19 (a)

Adjacency List:

Graph

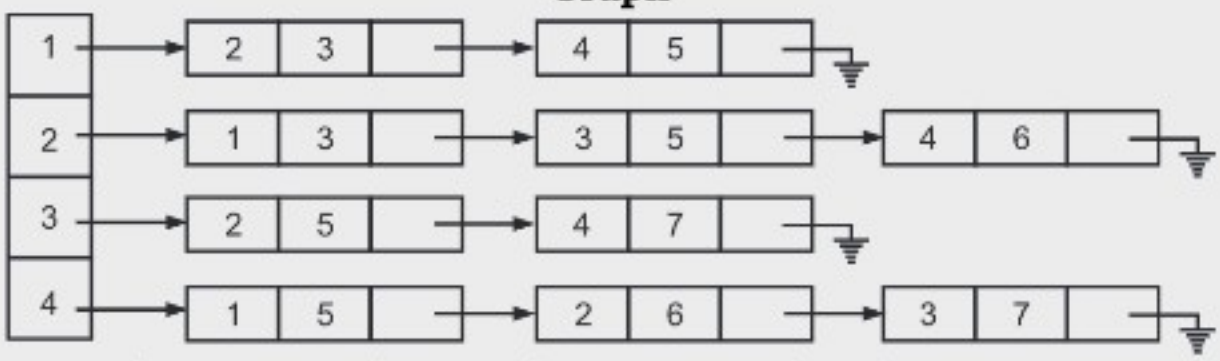


Fig. 6.19 (b): Adjacency list representation of above graph

6.3.3 Adjacency Multi-lists

- Adjacency Multi-list is an edge, rather than vertex based, graph representation. Typically, the following structure is used to represent an edge.
- With adjacency lists, each edge in an undirected graph appears twice in the list. Also, there is an obvious asymmetry for digraphs - it is easy to find the vertices a given vertex is adjacent to (simply follow its adjacency list), but hard to find the vertices adjacent to a given vertex (we must scan the adjacency lists of all vertices). These can be rectified by a structure called an adjacency multi-list.
- In Adjacency Multi-list nodes may be shared among several list (an edge is shared by two different paths).

visited	vertex at tail	vertex at head	Pointer to next edge containing vertex at tail	Pointer to next edge containing vertex at head

Example 7: Consider following undirected graph G=(V,E)

where V={0,1,2,3}, E={(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)}

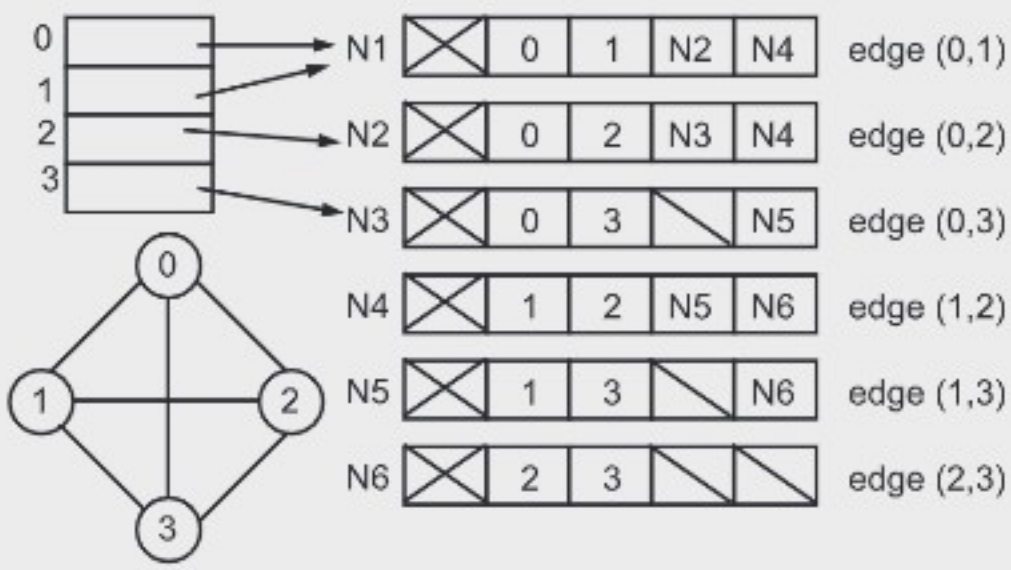


Fig. 6.20: Adjacency Multi-list representation

The lists are:

$V_0 = N_1 \rightarrow N_2 \rightarrow N_3$

$V_1 = N_1 \rightarrow N_4 \rightarrow N_5$

$V_2 = N_2 \rightarrow N_4 \rightarrow N_6$

$V_3 = N_3 \rightarrow N_5 \rightarrow N_6$

Example 8: Construct the adjacency matrix and adjacency list for the following graph. **[S-18]**

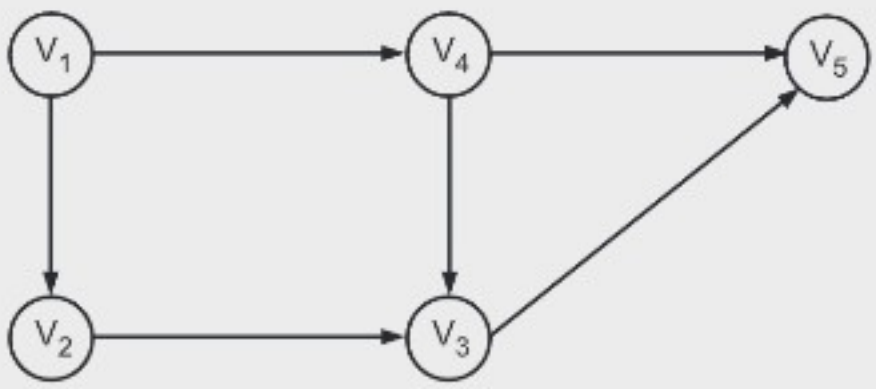


Fig. 6.21

Solution: Adjacency matrix:

$$\begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Adjacency list:

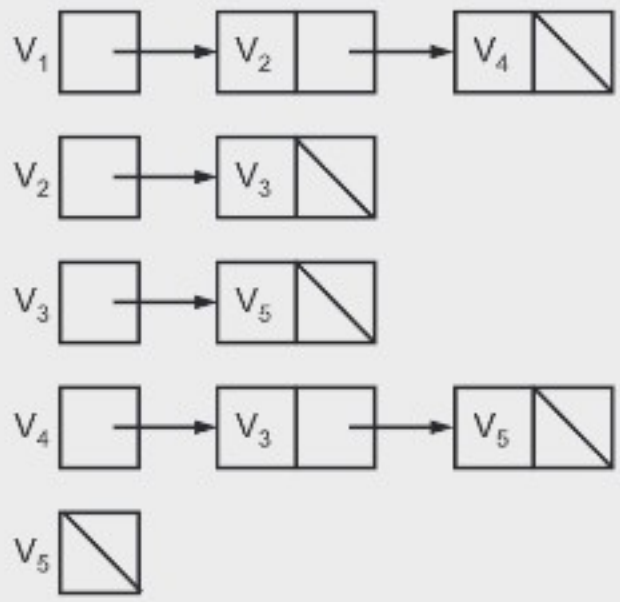


Fig. 6.22: Adjacency list representation of above graph

6.4 GRAPH TRAVERSAL

- In many situations, we need to visit all the vertices and edges in a systematic fashion, called as graph traversal.
- The graph traversal is used to decide the order of vertices to be visit in the search process.
- A graph traversal finds the edges to be used in the search process without creating loops that means using graph traversal we visit all vertices of graph without getting into looping path.
- The two techniques are used for traversals:
  1. Depth First Search (DFS)
  2. Breadth First Search (BFS)

6.4.1 Depth First Search (DFS)

- DFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops.
- We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

- We use the following steps to implement DFS traversal.

  Step 1 : Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

  Step 2 : Visit any one of the adjacent vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.

  Step 3 : Repeat step 2 until there are no new vertex to be visited from the vertex on top of the stack.

  Step 4 : When there is no new vertex to be visited then use back tracking and pop one vertex from the stack.(Back tracking is coming back to the vertex from which we came to current vertex)

  Step 5 : Repeat steps 2, 3 and 4 until stack becomes Empty.

  Step 6 : When stack becomes Empty, then stop. DFS traversal will be sequence of vertices in which they are visited using above steps.

- **The algorithm for DFS can be outlined as follows:**

```
1.  for V = 1 to n do (Recursive)
    visited[V] = 0  {unvisited}
2.  i = 1  {start at vertex 1)
3.  DFS (i)
    begin
     visited[i] = 1
      display vertex i
     for each vertex j adjacent to i do
     if (visited[j] = 0)
       then DFS(j)
    end.
```

- Non-recursive DFS can be implemented by using a stack for pushing all unvisited vertices adjacent to the one being visited and popping the stack to find the next unvisited vertex.

Algorithm using stack for DFS (Non-recursive):

```
1.  Push start vertex onto STACK
2.  While (not empty (STACK)) do
        begin
          v = POP (STACK)
          if (not visited (v))
            begin
              visited[v] = 1
              push all adjacent vertices of v onto STACK
            end
       end
3.  Stop.
```

'C' function for recursive DFS:

```
int visited[MAX]={0};
DFS (int i)
{     int k;
        visited[i] = 1;
        printf("%d",i);
          for(k=0;k<n;k++)
             if (A[i,k]==1  && visited[k]==0)
                   DFS (k);
}
```

- Let us consider graph 6.23 drawn below:

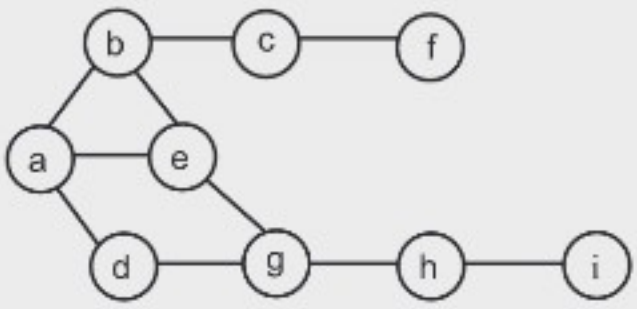
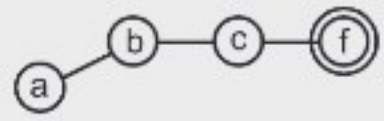
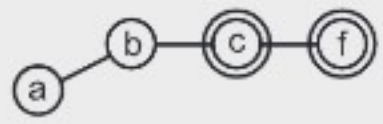
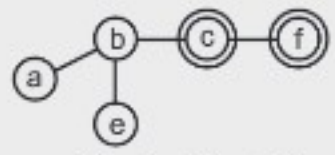
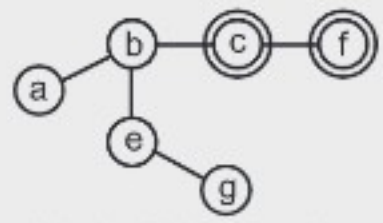
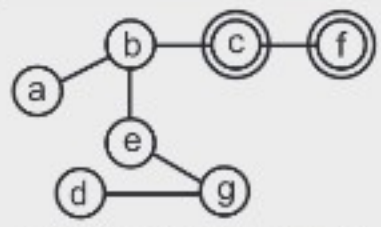
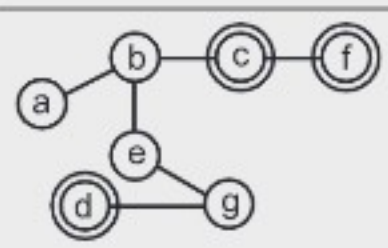
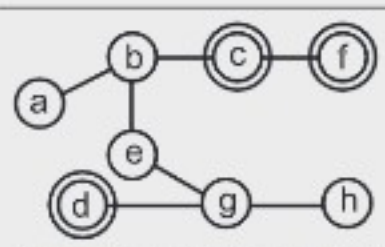
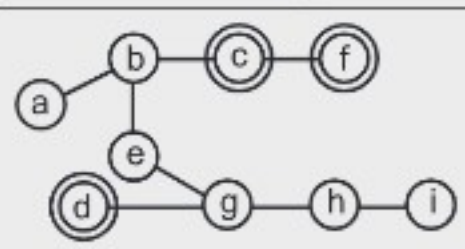


Fig. 6.23: Graph

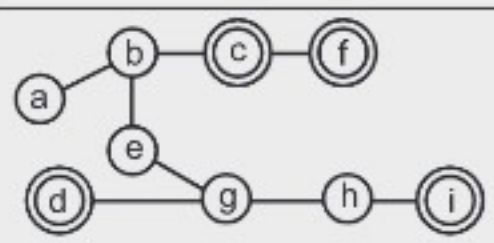
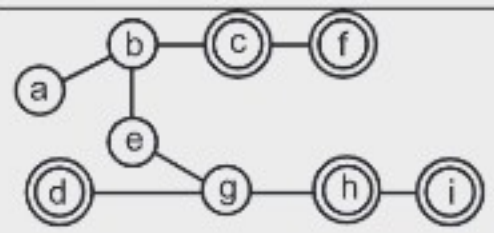
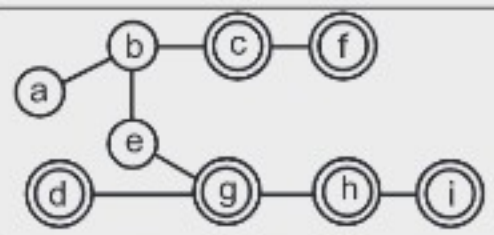
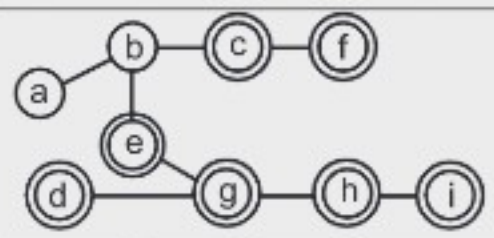
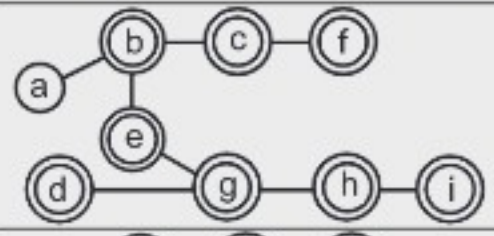
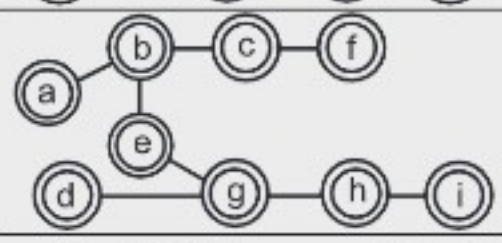
- Let us traverse the graph using DFS algorithm which uses stack. Let 'a' be a start vertex. Initially stack is empty. Doubled circle indicates backtracking point.

Steps	Stack	Spanning Tree
1. Select vertex 'a' as starting point (visit 'a'). Push 'a' onto the stack.	a	a Visited = {a}
2. Visit any adjacent vertex of 'a' which is not visited ('b'). Push newly visited vertex 'b' onto the stack.	b a	a b Visited = {a, b}
3. Visit any adjacent vertex of 'b' which is not visited ('c'). Push 'c' onto the stack.	c b a	a b c Visited = {a, b, c}
4. Visit any adjacent vertex of 'c' which is not visited ('f'). Push 'f' on to the stack.	f c b a	a b c f Visited = {a, b, c, f}

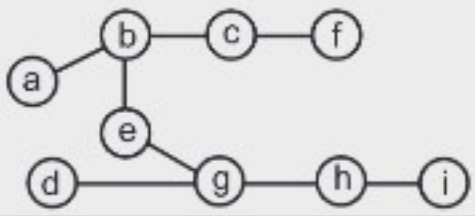
contd. ...

5. There is now new vertex to be visited from 'f'. So use backtrack. POP 'f' from the stack.	c b a	
6. There is no new vertex to be visited from 'c'. So use backtrack. Pop 'c' from the stack.	b a	
7. Visit any adjacent vertex of 'b' which is not visited ('e'). Push 'e' onto the stack.	e b a	 visited = [a,b,c,f,e]
8. Visit any adjacent vertex of 'e' which is not visited ('g'). Push 'g' onto the stack.	g e b a	 visited = [a,b,c,f,e,g]
9. Visit any adjacent vertex of 'g' which is not visited ('d'). Push 'd' onto the stack.	d g e b a	 visited = [a,b,c,f,e,g,d]
10. There is not new vertex to be visited from 'd'. So use backtrack. Pop 'd' from the stack.	g e b a	
11. Visit any adjacent vertex of 'g' which is not visited ('h'). Push 'h' onto the stack.	h g e b a	 visited = [a,b,c,f,e,g,d,h]
12. Visit any adjacent vertex of 'h' which is not visited ('i'). Push 'i' onto the stack.	i h g e b a	 visited = [a,b,c,f,e,g,d,i]

contd. ...

13. There is no new vertex to be visited from 'i'. So use backtrack. Pop 'i' from the stack.	h g e b a	
14. There is no new vertex to be visited from 'h'. So use backtrack. Pop 'h' from the stack.	g e b a	
15. There is no new vertex to be visited from 'g'. So use backtrack. Pop 'g' from the stack.	e b a	
16. There is no new vertex to be visited from 'e'. So use backtrack. Pop 'e' from the stack.	b a	
17. There is no new vertex to be visited from 'b'. So use backtrack. Pop 'b' from the stack.	a	
18. There is no new vertex to be visited from 'a'. So use backtrack. Pop 'a' from the stack.		

Stack becomes Empty. So Stop DFS traversal. Final result of DFS traversal is following spanning tree.



6.4.2 Breadth-First Search (BFS)

- Another systematic way of visiting the vertices is Breadth-First Search.
- Starting at vertex v and mark it as visited, the BFS differs from DFS, in that all unvisited vertices adjacent to i are visited next.
- Then unvisited vertices adjacent to these vertices are visited and so on until the entire graph has been traversed.

- The approach is called 'breadth first' because from vertex i that we visited, we search as broadly as possible by next visiting all the vertices adjacent to i.
- For example, the BFS of graph of Fig. 6.24 results in visiting the nodes in the following order: 1, 2, 3, 4, 5, 6, 7, 8.

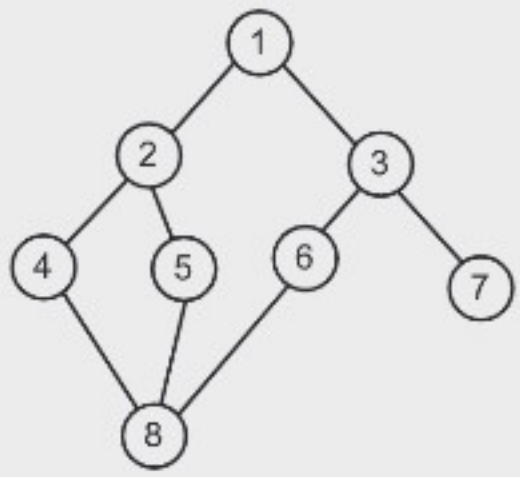


Fig. 6.24

- This search algorithm uses a queue to store the adjacent vertices of each vertex of the graph as and when it is visited.
- These vertices are then taken out from queue in a sequence (FIFO) and their adjacent vertices are visited and so on until all the vertices have been visited.
- The algorithm terminates when the queue is empty. The algorithm for BFS is given below. The algorithm initializes the boolean array visited [ ] to 0 (false) i.e. marks each vertex as unvisited.

```
for i = 1 to n do
    visited[i] = 0.
```

Algorithm for BFS:

```
BFS (i)
begin
    visited[i] = 1
    add (Queue, i)
    while not empty (Queue) do
      begin
          i = delete (Queue)
          for all vertices j adjacent to i do
            begin
            if (not visited[j] = 1)
            add (Queue, j)
                    visited[j] = 1
          end
    end
end
```

- Here, while loop is executed n time as n is the number of vertices and each vertex is inserted in a queue once. If adjacency list representation is used then adjacent nodes are computed in for loop.

'C' function for recursive BFS:

```
BFS (int i)
{
    int k,visited[MAX];
    struct queue Q;
    for(k=0;k<n;k++)
    visited[k] = 0;
    visited[i] = 1;
    printf("%d",i);
    insert(Q,i);
    while (!Empty(Q))
    {
        j = delete(Q);
        for(k=0;k<n;k++)
        {
            if (A[j,k]&&!visited[k])
            {
                insert(Q,k);
                visited[k] = 1;
            printf("%d",k);
            }
        }
    }
}
```

- Let us consider following graph,

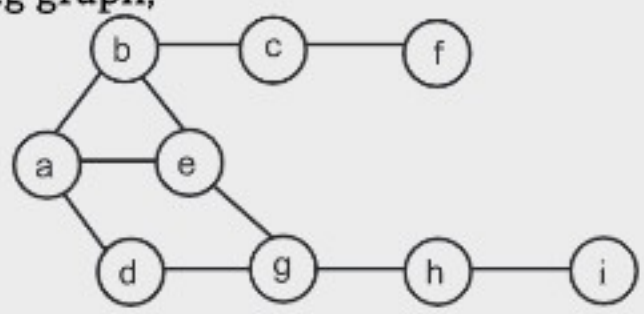
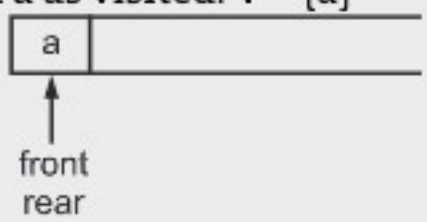


Fig. 6.25

- Let us traverse the graph using non-recursive algorithm which uses queue. Let 'a' be a start vertex. Initially queue is empty. Initial set of visited vertices, v = ϕ.

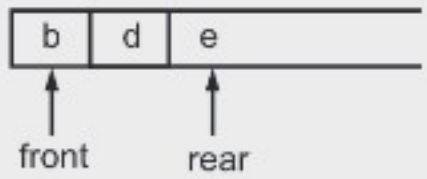
  (i) Add 'a' to the queue. Mark a as visited. V = {a}



(ii) As queue is not empty,

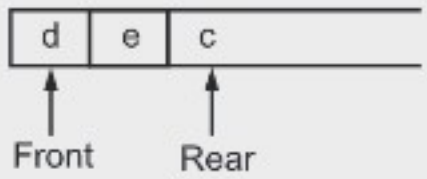
vertex = delete( ) = 'a'.

Add all not visited adjacent vertices of 'a' to the queue. Also mark them visited.
V = {a, b, d, e}



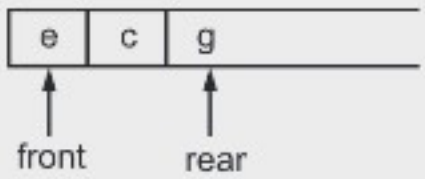
(iii) As queue is not empty,

Vertex = delete( ) = 'b'. Insert all adjacent, unvisited vertices of 'b' to the queue.
V = {a, b, d, e, c}



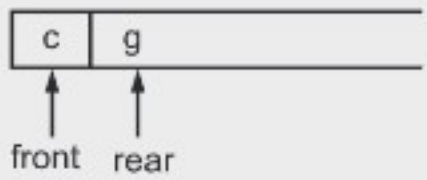
(iv) As queue is not empty,

Vertex = delete( ) = 'd'. Now insert all adjacent, unvisited vertices of 'd' to the queue and mark them as visited
V = {a, b, d, e, c, g}



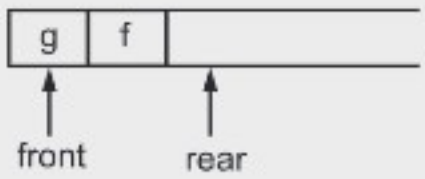
(v) As queue is not empty,

Vertex = delete( ) = 'e'. There are no unvisited adjacent vertices of 'e'. The queue is as:
V = {a, b, d, e, c, g}.



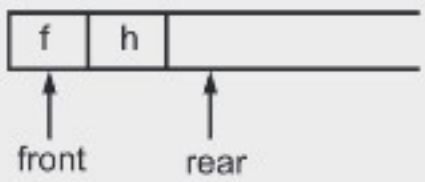
(vi) As queue is not empty, vertex = delete( ) = 'c'.

Insert all adjacent, unvisited vertices of 'c' to the queue and mark them visited.
V = {a, b, d, e, c, g, f}



(vii) As queue is not empty, vertex = delete( ) = 'g'.

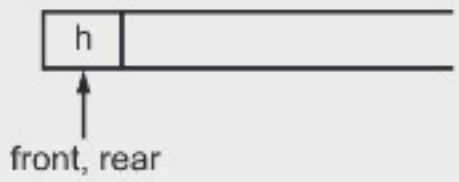
Insert all unvisited adjacent vertices of 'g' to the queue and mark them visited.
V = {a, b, d, e, c, g, f, h}



(viii) As queue is not empty, vertex = delete( ) = 'f'.

There are no unvisited adjacent vertices of 'f'. The queue is as:

V = {a, b, d, e, c, g, f, h}



(ix) As queue is not empty, vertex = delete( ) = 'h'.

Insert its unvisited adjacent vertices to queue and Mark them.

V = {a, b, d, e, c, g, f, h, i}

(x) As queue is not empty,

Vertex = delete( ) = ' i ', No adjacent vertices of i are unvisited.

(xi) As queue is empty, stop.

The sequence in which vertices are visited by BFS is as:

a, b, d, e, c, g, f, h, i.

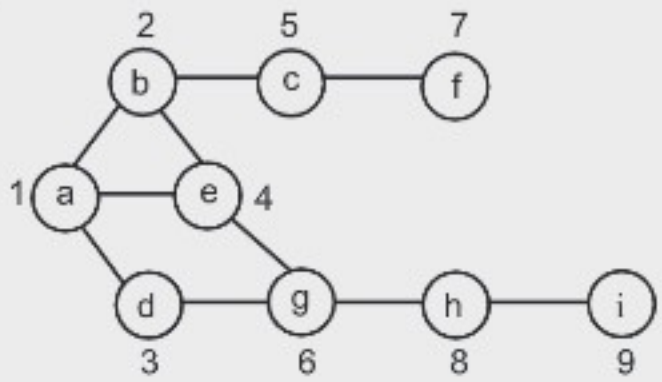


Fig. 6.26

Program 6.1: To create a graph and represent using adjacency matrix and traverse in BFS order.

```
#include<stdio.h>
#include<stdlib.h>
struct q
{
    int data[20];
    int front, rear;
} q1;
struct node
{
    int vertex;
    struct node * next;
}  * v[10];
```

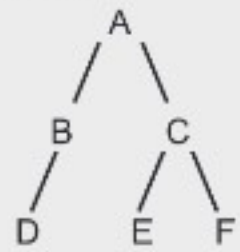
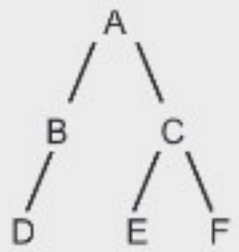
```
void add (int n)
{
   q1.rear++;
   q1.data[q1.rear]=n;
}
int del()
{
   q1.front++;
   return q1.data[q1.front];
}
void initq()
{
   q1.front = q1.rear = - 1;
}
int emptyq()
{
   return (q1.rear == q1.front);
}
void create(int m[10][10], int n)
{
   int i, j;
   char ans;
   for (i=0; i<n; i++)
   for (j=0; j<n; j++)
   {
      m[i][i] = 0;
      if(i != j)
      {
         printf("\n \t Is there an edge between %d and %d:", i+1, j+1);
         scanf("%d", &m[i][j]);
      }
   }
} /* end of create */
void disp(int m[10][10], int n)
{
   int i, j;
   printf("\n \t the adjacency matrix is: \n");
```

```
        for(i=0; i<n; i++)
        {
            for (j=0; j<n; j++)
            printf("%5d", m[i][j]);
            printf("\n");
        }
}  /* end of display */
void create1 (int m[10][10], int n)
{
    int i, j;
    struct node *temp, *newnode;
    for(i=0; i<n; i++)
    {
        v[i] = NULL;
        for(j=0; j<n; j++)
        {
            if(m[i][j] == 1)
            {
                newnode=(struct node *) malloc(sizeof(struct node));
                newnode -> next = NULL;
                newnode -> vertex = j+1;
                if(v[i]==NULL)
                v[i] = temp = newnode;
                else
                {
                      temp -> next = newnode;
                      temp = newnode;
                }
            }
        }
    }
}
void displist(int n)
{
    struct node *temp;
    int i;
```

```
    for (i=0; i<n; i++)
    {
        printf("\nv%d | ", i+1);
            temp = v[i];
            while (temp)
            {
                printf("v%d -> ", temp -> vertex);
                temp = temp -> next;
            }
            printf("null");
    }
}
void bfs(int m[10][10], int n)
{
    int i, j, v, w;
    int visited[20];
    initq();
    for(i=0; i<n; i++)
    visited[i] = 0;
    printf("\n \t The BFS traversal is: \n");
    v=0;
    visited[v] = 1;
    add(v);
    while(! emptyq())
    {
        v = del();
        printf("\n v%d ", v + 1);
            printf("\n");
            for(w = 0; w < n; w++)
            if((m[v][w] ==1) && (visited[w] == 0))
            {
                add(w);
                visited[w] = 1;
            }
    }
}
/* main program */
```

```
void main()
{
   int m[10][10], n;
   printf("\n \t enter no. of vertices");
      scanf("%d", &n);
      create(m,n);
      disp(m,n);
      create1(m,n);
      displist(n);
      bfs(m,n);
}
```

6.5 DFS VS. BFS

DFS	BFS
1. DFS stands for Depth first search	1. BFS stands for Breadth first search
2. DFS visit nodes depth wise	2. BFS visit nodes level by level
3. Uses stack data structure	3. Uses Queue data structure
4. DFS is faster compare to BFS	4. BFS is slower compare to DFS
5. It requires less memory	5. It requires more memory
6. Backtracking is allowed	6. Backtracking is not allowed
7. DFS is not optimal for finding shortest path	7. BFS is optimal for finding shortest path
8. Generally gets trapped into infinite loop	8. Can never get trapped into infinite loop
Example:  A, B, D, C, E, F	**Example:**  A, B, C, D, E, F

6.6 APPLICATIONS OF GRAPH

1. Minimum spanning trees are useful in many applications such as in finding the least amount of wire needed to connect group of computers, houses or cities.
2. To find the shortest path. For example, suppose, there are several warehouses which are distributed over geographically distant locations. It is required to transport goods from a given warehouse to another. The various paths possible among the warehouses. So it is required to find a path where the sum of the weights are minimum.
3. **Critical path:** A path of longest length in the AOE network is a critical path. **[S-19]**

4. Another application is coloring of map. We have to color a map so that no two adjacent regions have the same color. This problem can be presented by using graph.
5. The traversal of a graph like Depth First traversal has many important applications such as finding the components of a graph, detecting cycles in an undirected graph etc.
6. Many other problems in which graphs are used are:
   (a) Topological sorting of a graph
   (b) Spanning trees
   (c) Critical path finding
   (d) To route from one location to other
   (e) Reliability analysis (it is an important issue in system design, manufacturer and maintenance) or network reliability problem.

- Let us discuss few of the applications of graphs.

6.6.1 AOV Network

- A large project is subdivided into several small projects called activities. If all the activities of the project are complete then the entire project is completed. In the project some of the activities are started independently. But some of the activities are started only if their previous activities are completed. For example, Result is printed only after the marks are entered. Hence, data entry is the previous activity of result preparation. The relationship between such a activities are represented using directed graph. The graph has an edge (i, j) if activity i is a prerequisite for activity j. The graph can be represent using AOV network.

Definition of AOV Network:

- A directed graph G in which the vertices represent tasks or activities and the edges represent precedence relations between activities is called an activity on vertex network or AOV network.

Topological Ordering:

- Topological sorting of vertices of a Directed Acyclic Graph (DAG) is an ordering of the vertices $v_1, v_2, \ldots v_n$ in such a way, that if there is an edge directed towards vertex v_j from vertex v_i, then v_i comes before v_j.
- Topological ordering is not possible if the graph has a cycle. The Fig. 6.27 shows the topological sorting.

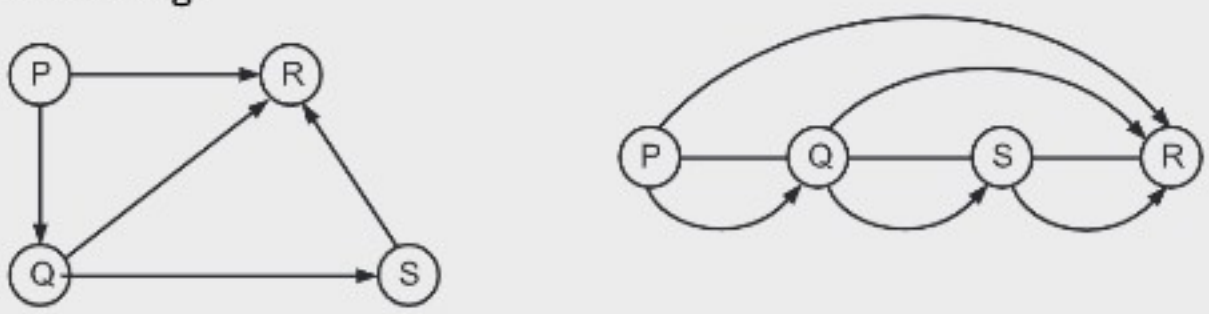


Fig. 6.27: A graph and its topological ordering

Design of Topological Sort:

1. Input the AOV network with n no. of vertices.
2. For i = 1 to n, if every vertex has a predecessor (check in-degree) the graph has a cycle go to step 8.
3. Pick a vertex V which has no predecessors and add in order set.
4. Output V.
5. Delete V along with all outgoing edges of V from graph.
6. Reduce in-degree of nodes adjacent to V by 1.
7. If vertices exists in G then repeat steps from 3 to 6 otherwise go to step 8
8. Stop.

Example 9: Consider a graph of Fig. 6.28(a) which is AOV network.

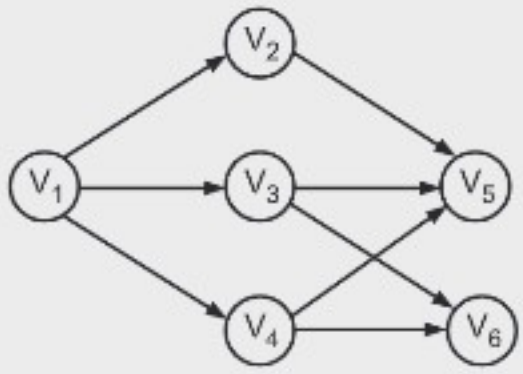
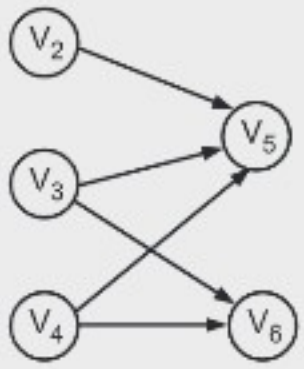


Fig. 6.28 (a) AOV network

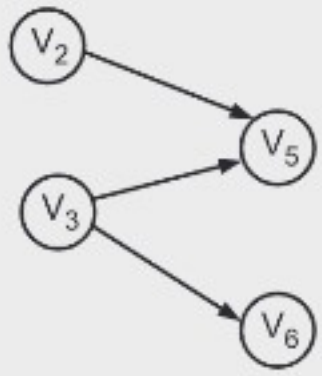
(a) In initial graph, indegree of $V_1 = 0$.

Therefore, pick V_1 and edges <V_1, V_2> <V_1, V_3> <V_1, V_4> are deleted along with V_1.

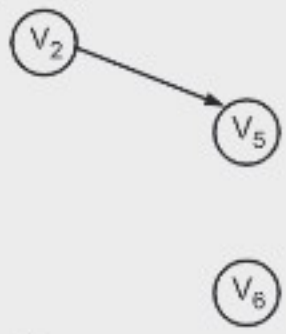


(b) After adding V_1 to an order set, indegree (V_2) = indegree (V_3) = indegree (V_4) = 0.

Therefore, pick any of them. Assume V_4 is picked and added to the order list. The vertex V4 and its outgoing edges are deleted.



(c) Now we can pick either V_2 or V_3. Assume V_3 is picked and added to the order list. The vertex V_3 and its outgoing edges are deleted.



(d) V_6 is picked and added to an order set.

(e) V_2 is picked and added to an order set.

(f) V_5 is picked and added to an order set.

Fig. 6.28 (b): Action on AOV network to find topological order

Therefore, topological order is,

$V_1 \rightarrow V_4 \rightarrow V_3 \rightarrow V_6 \rightarrow V_2 \rightarrow V_5$

Topological order may differ depending upon vertex selection if in intermediate steps more than one vertex has in-degree 0.

Program 6.2: Accept vertices and edges for a graph and represent it as adjacency list.

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 30
struct node
{
   int data;
   struct node *next;
}*v[MAX];
int **a;
void insert(int);
void display_matrix(int);
void matrix_to_list(int);
void display_list(int);
int main()
{
   int i=0,n;
   printf("\nHow many vertices you want to create:\n");
   scanf("%d",&n);
   a=(int **)malloc(n * sizeof(int *));
```

```
 if(a==NULL)
 {
     printf("memory allocation failed");
     return -1;
 }
for(i=0;i<n;i++)
{
     a[i]=(int *)malloc(n * sizeof(int));
     if(a[i]==NULL)
     {
        printf("memory allocation failed");
        return -2;
     }
}
insert(n);
printf("\nAdjencency Matrix is:\n");
display_matrix(n);
 matrix_to_list(n);
 printf("\nAdjencency list is:\n");
 display_list(n);
}
 void insert(int n)
 {
   int i=0,j=0;
   printf("\nEnter the Adjenecency matrix values:\n");
   for(i=0;i<n;i++)
   {
         for(j=0;j<n;j++)
         {
            a[i][j]==0;
            printf("\nif there is directed pathe from v%d to v%d then
            enter 1 else enter 0:\n",i+1,j+1);
            scanf("%d",&a[i][j]);
         }
   }
 }
```

```
void display_matrix(int n)
{
   int i=0,j=0;
   for(i=0;i<n;i++)
   {
      for(j=0;j<n;j++)
      {
         printf("%d\t",a[i][j]);
      }
      printf("\n");
   }
}
void matrix_to_list(int n)
{
   int i=0,j=0;
   struct node *last,*newnode;
   for(i=0;i<n;i++)
   {
      v[i]=NULL;
      for(j=0;j<n;j++)
      {
         if(a[i][j]==1)
         {
            newnode=(struct node *)malloc(sizeof(struct node));
            newnode->next=NULL;
            newnode->data=j+1;
            if(v[i]==NULL)
               v[i]=last=newnode;
            else
            {
               last->next=newnode;
               last=newnode;
            }
         }
      }
   }
}
```

```
void display_list(int n)
{
    int i=0;
    struct node *temp;
    for(i=0;i<n;i++)
    {
        printf("\nList for v%d:-->",i+1);
        temp=v[i];
        while(temp!=NULL)
        {
            printf("v%d-->",temp->data);
            temp=temp->next;
        }
    }
}
```

Example 10: Draw graph for following adjacency list.

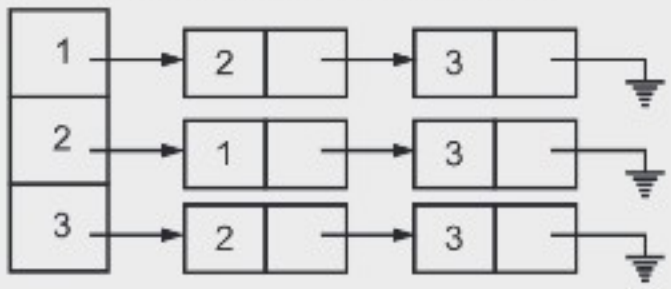


Fig. 6.29 (a)

Solution:

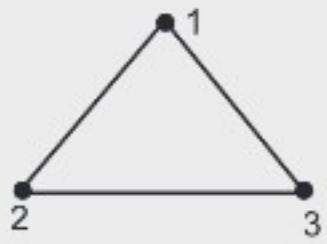


Fig. 6.29 (b)

Example 11: For the following graph, give adjacency list representation and also give the result of depth first traversal and breadth first traversal.

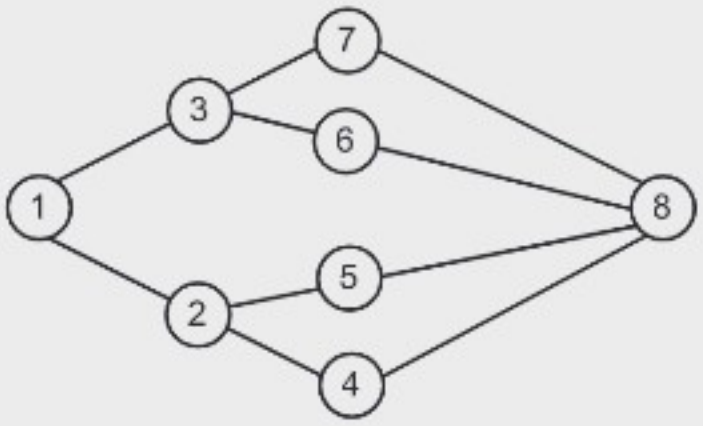


Fig. 6.30 (a)

Solution: Adjacency list for above group is as:

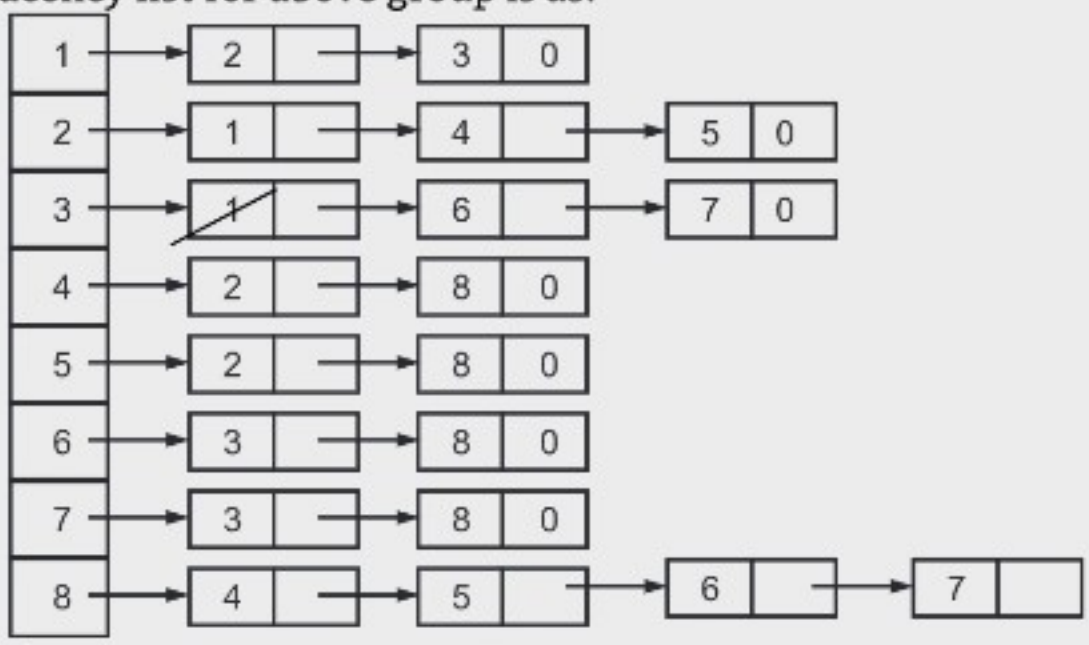


Fig. 6.30 (b)

Let 1 be the start vertex.
Result of DFS: 1, 2, 4, 8, 5, 6, 3 and 7.
and
BFs: 1, 2, 3, 4, 5, 6, 7 and 8.

Example 12: Consider the following specification of a graph G.

V(G) = {1, 2, 3, 4)
E (G) = {(1, 2), (1, 3), (3, 3), (3, 4), (4, 1)}

(a) Draw a picture of the undirected graph.
(b) Draw its adjacency matrix.

Solution: (a)

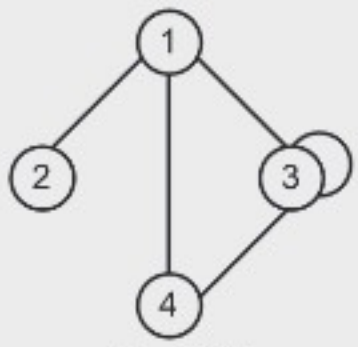


Fig. 6.31

(b) Adjacency matrix of G

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Summary

- Graph is representation of relation. Vertices represent elements and edges represent relationships.
- Graph are of Two types: directed and undirected.
- An adjacency matrix is a data structure for representing a graph by keeping a matrix of 0's and 1's in which each 1 corresponds with an edge.
- There are two standard graph traversal methods, i.e. DFS and BSF.

- In DFS, descendants are processed before moving to an adjacent node.
- In BFS, all of the adjacent vertices are processed before processing the descendants of a vertex.
- Graphs are used in many applications such as topological sorting of a graph, critical path finding, shortest path finding etc.
- Use vertices to represent activities, and directed edges to indicate the order of activities. Vi must be preceded by the activity Vj. This directed graph is called the AOV network (Activity On Vertices).
- Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge (u,v), vertex u comes before v in the ordering.
- Topological Sorting for a graph is not possible if the graph is not a DAG.

Check Your Understanding

Multiple Choice Questions:

1. A graph is a collection of _____.
   (a) Rows and columns (b) Vertices and edges
   (c) Equations (d) None of these
2. The degree of any vertex of graph is _____.
   (a) The number of edges incident with vertex
   (b) Number of vertex in a graph
   (c) Number of vertices adjacent to that vertex
   (d) Number of edges in a graph
3. A vertex with degree one in a graph is called _____.
   (a) A leaf (b) Pendant vertex
   (c) Adjacency list (d) None
4. The maximum degree of any vertex in a simple graph with n vertices is _____.
   (a) n – 1 (b) n + 1
   (c) 2n – 1 (d) N
5. The number of distinct simple graph with upto 3 nodes is _____.
   (a) 15 (b) 10
   (c) 7 (d) 9
6. Which of the following ways can be used to represent a graph?
   (a) Adjacency Matrix (b) Adjacency List
   (c) Adjacency multi-list (d) All of the above
7. The Data structure used in standard implementation of Breadth First Search is?
   (a) Stack (b) Queue
   (c) Linked List (d) Tree
8. Depth First Search is equivalent to which of the traversal in the Binary Trees?
   (a) Pre-order Traversal (b) Post-order Traversal
   (c) Level-order Traversal (d) In-order Traversal
9. The Data structure used in standard implementation of Depth First Search is?
   (a) Stack (b) Queue
   (c) Linked List (d) Tree

10. In most of the cases, topological sort starts from a node which has _____.
    (a) Maximum Degree (b) Minimum Degree
    (c) Any degree (d) Zero Degree

Answers

1. (b)	2. (a)	3. (b)	4. (a)	5. (a)	6. (d)	7. (b)	8. (a)	9. (a)	10. (d)

Practice Questions

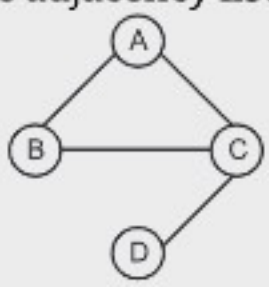
Q.I Answer the following questions in short:

1. What is graph? Give various methods of representing graph.
2. Give total degree of each node of the following graph:

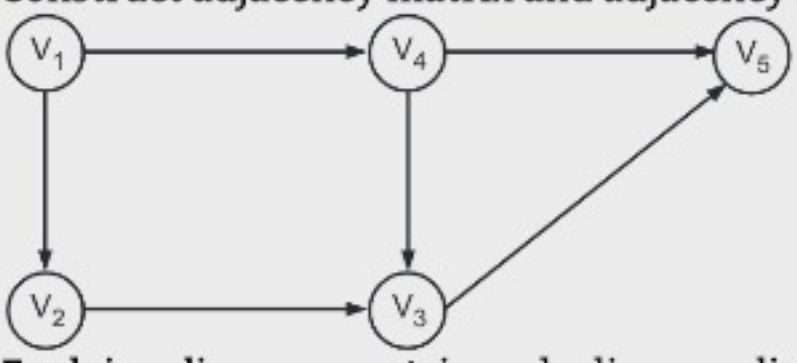
3. What is DFS and BFS?
4. What are the various applications of graph?
5. List out different graph representation technique.

Q.II Answer the following questions:

1. List and explain different graph representation technique.
2. Consider the following specification of a graph G.
   V(G) = {1, 2, 3, 4)
   E (G) = {(1, 2), (1, 3), (3, 3), (3, 4), (4, 1)}
   (a) Draw a picture of the undirected graph.
   (b) Draw its adjacency matrix.
3. Suggest suitable Node structure for a weighted graph's adjacency list representation.
   Give adjacency list representation of following graph.

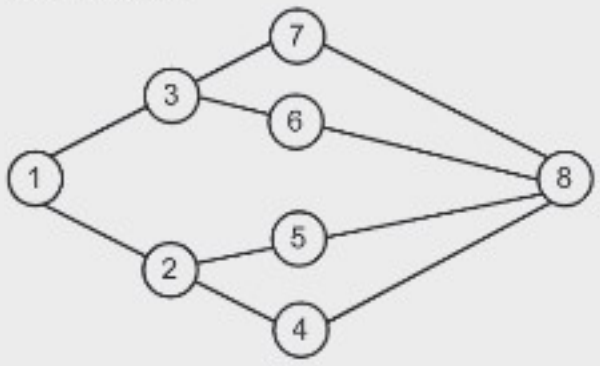


4. Explain adjacency multi-list representation of graph.
5. Construct adjacency matrix and adjacency list for the following graph:

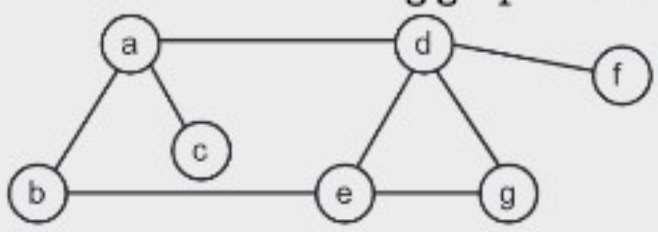


6. Explain adjacency matrix and adjacency list with suitable example.
7. List graph traversal techniques and the data structure used in them.

8. For the following graph, give result of Breadth first traversal and Depth first traversal.



9. Consider an undirected graph G whose edges are
   (V_1, V_2) (V_1, V_3) (V_2, V_3) (V_2, V_4) (V_3, V_4)
10. Consider the following graph and traverse using BFS and DFS.



11. Explain DFS algorithm with suitable example
12. Explain BFS with example.
13. Write difference between DFS and BFS.
14. Explain AOV network with example.

Q.III Define the terms:

1. Edge
2. Isolated vertex
3. Complete graph
4. Indegree and Outdegree
5. Adjacency matrix
6. Multigraph

Previous Exam Questions

Summer 2019

1. _____ is closed graph with length 3 or more, such that starts and ends with same vertex. **[1 M]**

(i) Cycle (ii) Path

(iii) Degree (iv) Vertex

Ans. (i)

2. Define complete graph. **[1 M]**

Ans. Refer to Section 6.2.

3. Define the following terms: **[1 M]**

(i) Multigraph

Ans. Refer to Section 6.1.1

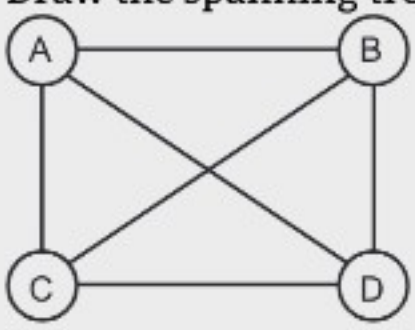
(ii) Critical path.

Ans. Refer to Section 6.6.

4. What is graph? How is it represented? **[4 M]**

Ans. Refer to Section 6.1.1 and 6.3.

5. Draw the spanning tree of graph G: **[1 M]**



Ans. Refer to Section 6.2.

Winter 2018

1. The maximum degree of any vertex in a simple graph with n vertices is ____. **[1 M]**

(i) $n-1$ (ii) $2n-1$

(iii) $n+1$ (iv) n

Ans. (i)

2. Define Complete Graph. **[1 M]**

Ans. Refer to Section 6.2.

3. What is Graph? Give various methods of representing graphs. **[4 M]**

Ans. Refer to Section 6.1.1 and 6.3.

Summer 2018

1. The number of distinct simple graph upto 3 nodes is _____. **[1 M]**

(i) 15 (ii) 10

(iii) 7 (iv) 9

Ans. **(iii)**

2. Define complete graph **[1 M]**

Ans. Refer to Section 6.2.

3. Define the terms: (i) Node (ii) Edge (iii) Path **[3 M]**

Ans. **(i) Node:** Refer to Section 6.2.

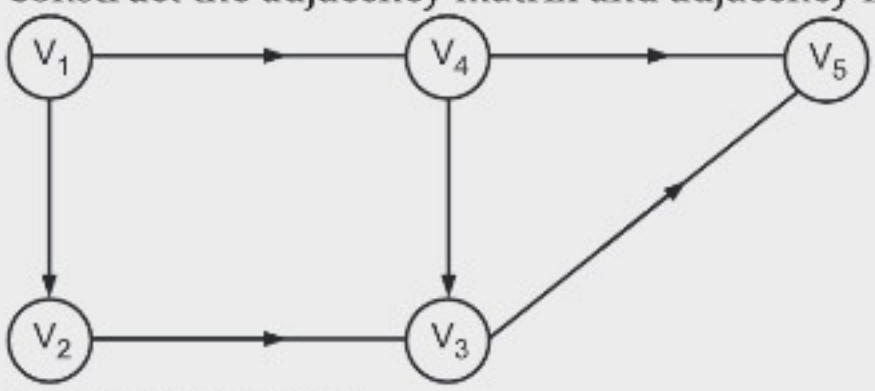
(ii) Edge: Refer to Section 6.1.1

(iii) Path: Refer to Section 6.2.

4. What is a critical path. **[3 M]**

Ans. Refer to Section 6.5.2.

5. Construct the adjacency matrix and adjacency list for the following graph. **[4 M]**



Ans. Refer to Example 8.

■■■