UNIT I

CHAPTER 1 Introduction

University Prescribed Syllabus

Introduction to Data Structures : Concept of data, Data object, Data structure, Concept of Primitive and non-primitive, linear and Nonlinear, static and dynamic, persistent and ephemeral data structures, Definition of ADT.

Analysis of algorithm : Frequency count and its importance in analysis of an algorithm, Time complexity and Space complexity of an algorithm Big 'O', 'Ω' and 'Θ' notations.

Sequential Organization : Single and multidimensional array and address calculation.

Linked Organization : Concept of linked organization, Singly Linked List, Doubly Linked List, Circular Linked List (Operations: Create, Display, Search, Insert, Delete).

Case Study : Set Operation, String Operation

Syllabus Topic : Introduction to Data Structures : Concept of Data, Data Object, Data Structure

1.1 INTRODUCTION TO DATA STRUCTURES : CONCEPT OF DATA, DATA OBJECT, DATA STRUCTURE

- In the modern era Data and its information is considered as very important part related to any organization.
- **Data** is nothing but the collection of facts and figures or data is value or group of values which is in a particular format.
- While developing different types of applications, one has to store such data in a standard format. Only the storage of data is not sufficient, later on we have to perform various types of operations on such like insertion, deletion, modifications etc.
- Hence the data must be stored in a systematic format so that one can easily perform different operations on it.
- All the programming languages provide a set of built in data types to store the data such as int, float, char etc.
- To store data various features are provided by programming languages such as variable, array, structure, union etc.
- As the modern day programming problems are complex and large, all these features are not sufficient to store the huge data.

UQ. 1.1.1 *Define Data structures.*

SPPU - Dec. 16, Dec. 17, Dec.18, 2 Marks

❑ *Definition of Data Structure : Data Structure is a way of collecting as well as organizing data in such a way that various operations can be performed on it in an effective way. A data structure is a logical model of a particular organization of data.*

- Data Structure is regarding the representation of data elements in terms of specific relationship, for the purpose of better organization and storage. For example, consider we have data regarding cricket player's name "Rohit" and age 30. Here the data "Rohit" is of String type and 30 is of integer type data.
- This data can be organized as a record like Player record. Now records of all the players can be collected and stored in a file or database as a data structure. For example, "Dhoni" 36, "Yuvraj" 36, "Hardik" 24.
- In simple language, we can say that Data Structures are structures which are programmed to store sequential data, so that it will be easy to perform various operations on it.
- The knowledge of data is represented by it which is to be organized in memory.
- The design and implementation of Data Structures is in such a manner that the complexity gets reduced and efficiency gets increased.

1.1.1 Data Object

UQ. 1.1.2 *Define the term data object.*

SPPU - Dec. 16, Dec. 17, Dec.18, 2 Marks

❑ *Definition of Data Object : A data object is a region of storage that contains a value or group of values.*

- Each value can be accessed using its identifier or a more complex expression that refers to the object.
- In addition, each object has a unique *data type*. The data type of an object determines the storage allocation for that object and the interpretation of the values during subsequent access.
- It is also used in any type checking operations. Both the identifier and data type of an object are established in the object *declaration*.

1.1.2 Need of Data Structure

GQ. 1.1.3 *Why do we need data structure ?* **(2 Marks)**

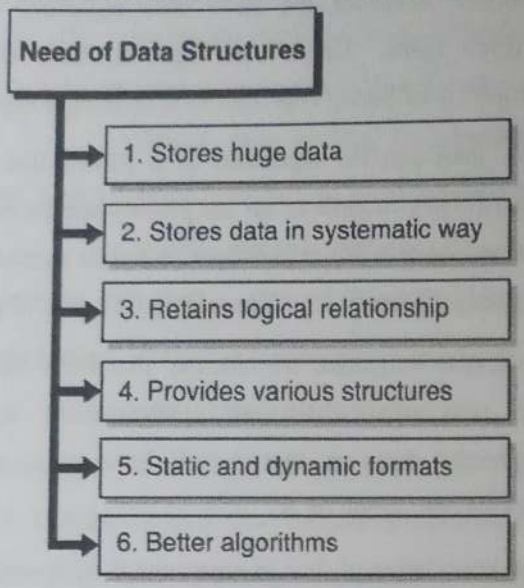


Fig. 1.1.1 : Need of Data Structures

1. Stores huge data

- As we have seen that modern day computing problems are complex and large.
- The basic data elements provided by programming languages like variable, array, structures are not sufficient to store such huge data.
- Data structures provide a way to store such huge data.

2. Stores data in systematic way

- After storing data, variety of operations has to be performed on data of various types like numeric, string, date etc.
- Data structures help to store the data in systematic way to ease the operations. It makes easy to store and manipulate data.

3. Retains logical relationship

Data structures help to retain the logical relationship between the data elements.

4. Provides various structures

Data structures provide various structures such as stack, queue, linked list etc. to store the data as per the need of application.

5. Static and dynamic formats

Data structures provide static as well as dynamic formats to store the data.

6. Better algorithms

Data structures provide better algorithms to apply on organized data to improve efficiency of program.

1.2 TYPES OF DATA STRUCTURE

GQ. 1.2.1 *Give classification of data structure and give two examples of each.* **(4 Marks)**

In computer science, Data Structure is classified into two categories (Refer Fig. 1.2.1).

1. Primitive Data Structure
2. Non Primitive Data Structure

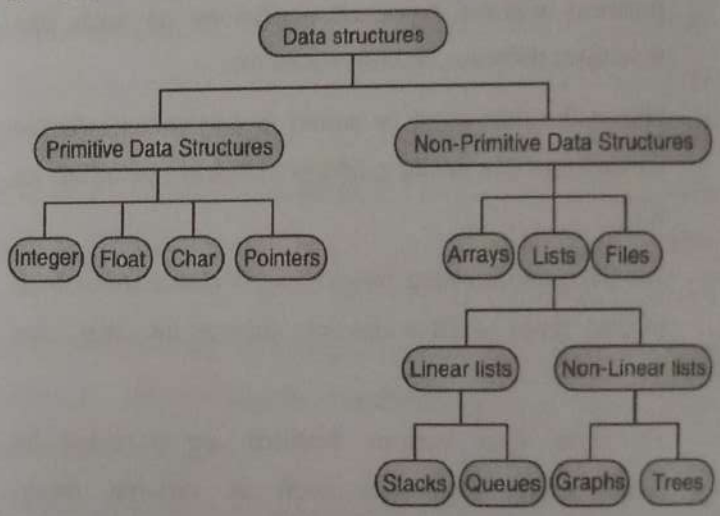


Fig. 1.2.1 : Classification of Data Structures

Syllabus Topic : Concept of Primitive Data Structures

1.2.1 Primitive Data Structures

GQ. 1.2.2 *Define the term : Primitive data structure.* (Refer Fig. 1.2.2) **(1 Mark)**

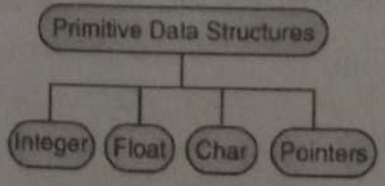


Fig. 1.2.2 : Primitive Data Structures

❑ *Definition of Primitive Data Structures Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers.*

- These are the basic data types provided by various programming languages.

☞ **Examples**

- Integers, Floating point numbers, Character constants, String constants and Pointers come under this category.

Syllabus Topic : Concept of Non-Primitive Data Structures

1.2.2 Non - Primitive Data Structures

GQ. 1.2.3 *Define the term : Non primitive data structures. (Refer Fig. 1.2.3).* **(1 Mark)**

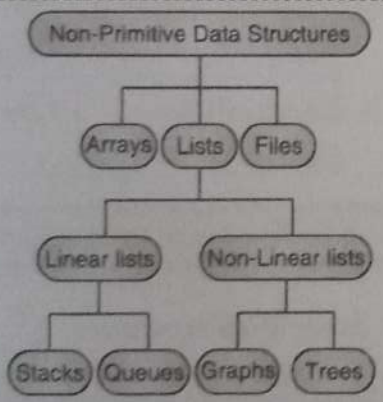


Fig. 1.2.3 : Non-Primitive Data Structures

❑ *Definition of Non-primitive data structures Non-primitive data structures are more complicated data structures and are derived from primitive data structures.*

- They emphasize on grouping same or different data items with relationship between each data item.

☞ **Examples**

Arrays, Lists and Files come under this category.

Syllabus Topic : Concept of Linear Data Structures

1.2.3 Concept of Linear Data Structures

UQ. 1.2.4 *Explain linear data structure with suitable example.* **SPPU - May 17, 4 Marks**

GQ. 1.2.5 *Define and explain term : Linear Data structure.* **(1 Mark)**

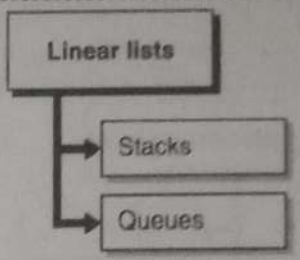


Fig. 1.2.4 : Linear Data Structures

❑ *Definition of Linear data structure*

The data structure where data items are organized sequentially or linearly one after another is called linear data structure.

- Data elements in a liner data structure are traversed one after the other and only one element can be directly reached while traversing. All the data items in linear data structure can be traversed in single run.
- These kinds of data structures are very easy to implement because memory of computer also has been organized in linear fashion.

☞ **Examples of linear data structures are**

Stack and Queue

a. Stack

- **Stack** is a data structure in which addition and deletion of element is allowed at the same end called as TOP of the stack.
- A Stack is a LIFO (Last In First Out) data structure where element that added last will be retrieved first.

b. Queue

- A **Queue** is a data structure in which addition of element is allowed at one end called as REAR and deletion is allowed at another end called as FRONT.

- A Queue is a FIFO (First In First Out) data structure where element that added first will be retrieved first.

Syllabus Topic : Concept of Non Linear Data Structures

1.2.4 Concept of Non Linear Data Structures

GQ. 1.2.6 *Define and explain term : Non-linear Data structure.* (Refer Fig. 1.2.5). **(1 Mark)**

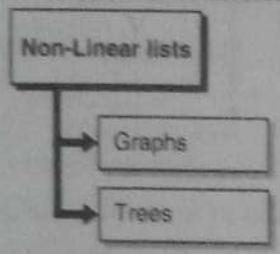


Fig. 1.2.5 : Non Linear Data Structures

Definition of Non linear data structure

The data structure in which the data items are not organized sequentially or in linear fashion is called as non linear data structure.

- In other words, a data element of the non linear data structure could be connected to more than one element to reflect a special relationship among them.
- All the data elements in non linear data structure cannot be traversed in single run.

Examples of non linear data structures

Trees and Graphs

a. Graph

A Graph is a collection of a finite number of vertices and edges which connect these vertices. Edges represent relationships among vertices that stores data elements.

b. Tree

A Tree is collection of nodes where these nodes are arranged hierarchically and form a parent child relationship.

Difference between Linear and Non Linear Data Structure

UQ. 1.2.7 *Differentiate between linear and non-linear data structures with example.* **SPPU - Dec. 17, Dec.19, 4 Marks**

Parameter	Linear Data Structure	Non-Linear Data Structure
Relation	Every item is related to its previous and next item.	Every item is attached with many other items.
Arrangement	Data is arranged in linear sequence.	Data is arranged in non-linear sequence.
Traversing	Data items can be traversed in a single run.	Data cannot be traversed in a single run.
Implementation	Implementation is Easy.	Implementation is difficult.
Examples	Array, Stack, Queue, Linked List.	Tree, Graph.

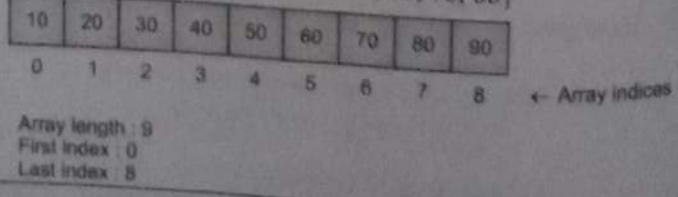
Syllabus Topic : Static Data Structures

1.2.5 Static Data Structures

UQ. 1.2.8 *Explain static data structure with example.* **SPPU - May 17, 2 Marks**

- In Static data structure the size of the structure is fixed. It is possible to modify the content of the data structure but without making changes in the memory space allocated to it.
- Example of Static Data Structures : Array

int a[9] = {10, 20, 30, 40, 50, 60, 70, 80}



☞ **Advantage of static data structure**

1. Faster access to elements (when compared with dynamic data structures).

☞ **Disadvantage of static data structure**

1. Add, remove or modify elements is not directly possible. If done, it is a resource consuming process.
2. Fixed size.
3. Resources allocated at creation of data structure, even if elements are not contain any value.

Syllabus Topic : Dynamic Data Structures

1.2.6 Dynamic Data Structures

UQ. 1.2.9 *Write note on : Dynamic Data Structures.* **SPPU -May 17, 2 Marks**

- Dynamic data structures are designed to facilitate change of data structures in the runtime.
- It is possible to change the assigned values of elements, as it was with static structures.
- Also, in dynamic structures the initially allocated memory size is not a problem. It is possible to add new elements, remove existing elements or do any kind of operation on data set without considering about the memory space allocated initially.
- Example of Dynamic Data Structures : Linked List.

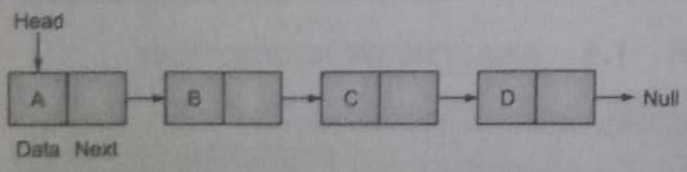


Fig. 1.2.6

☞ **Advantage of dynamic data structure**

1. Ability to efficiently add, remove or modify elements
2. Flexible size.
3. Effective use of resources - because resources are allocated at the runtime, as required.

☞ **Disadvantage of dynamic data structure**

1. Slower access to elements (when compared with static data structures)

Syllabus Topic : Persistent Data Structures

1.2.7 Persistent Data Structures

GQ. 1.2.10 *Write note on Persistent Data Structures.* **(2 Marks)**

- In computing, a persistent data structure is a data structure that always preserves the previous version of itself when it is modified.
- Such data structures are effectively immutable, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure.
- A data structure is **partially persistent** if all versions can be accessed but only the newest version can be modified.
- The data structure is fully persistent if every version can be both accessed and modified.
- If there is also a meld or merge operation that can create a new version from two previous versions, the data structure is called confluently persistent. Structures that are not persistent are called ephemeral.

☞ **Partial versus Full Persistence**

- In the partial persistence model, a programmer may query any previous version of a data structure, but may only update the latest version.
- This implies a linear ordering among each version of the data structure.
- In fully persistent model, both updates and queries are allowed on any version of the data structure. In some cases the performance characteristics of querying or updating older versions of a data structure may be allowed to degrade, as is true with the Rope data structure.

- In addition, a data structure can be referred to as confluently persistent if, in addition to being fully persistent, two versions of the same data structure can be combined to form a new version which is still fully persistent.

Syllabus Topic : Ephemeral Data Structures

1.2.8 Ephemeral Data Structures

- An ephemeral data structure is one for which only one version is available at a time: after an update operation, the structure as it existed before the update is lost.
- Much of the data stored on computers, such as the data stored in random access memory (RAM) and caches, is temporary and therefore referred to as "ephemeral" (which means transitory, or existing only briefly).
- These temporary, transient files are deleted as often as every few hours.

Syllabus Topic : Abstract Data Types (ADT)

1.3 ABSTRACT DATA TYPES (ADT)

UQ. 1.3.1 *Define the following term with example : (i) Abstract data type.*

SPPU - Dec. 16, Dec. 17, 2 Marks

UQ. 1.3.2 *What is ADT?* SPPU - May 19, 2 Marks

Definition of Abstract Data Type (ADT)

In computer science, an Abstract Data Type (ADT) is a mathematical model for data types, where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.

- Nowadays, programmer needs to handle some special kind of data.
- In such case it becomes difficult to store such data using the basic built in data type.
- Hence to fulfill the application requirement, the programmer needs to create his own data types.
- For this purpose programmer has to decide the type of data, operations to perform on the data and rules to follow while performing operations. This is implemented with the help of ADT.
- ADT refers to defining our own data types.
- ADT is useful tool which helps to specify logical properties of data type without the need of going into details.
- Simply we can say that Data type is collection of values and operations to be performed on those values. A mathematical construct is formed by this collection. The ADT refers to this mathematical construct.
- The definition of ADT can be written in another way also.

Another Definition of Abstract Data Type(ADT)

ADT is set of D, F and A

where,

D (Domains) - Data objects,

F (Functions) - Set of operations which can be carried out on data objects,

A (Axioms) - Properties and rules of the operations.

Syllabus Topic : Analysis of Algorithms

1.4 ANALYSIS OF ALGORITHMS

UQ. 1.4.1 *Explain the importance of algorithm analysis.* SPPU - Dec. 18, 2 Marks

- From two different algorithms provided for a task how do we find out which one is better?
- One simple method of doing this is; implement both of the two algorithms and execute the two programs on computer for various different inputs and observe which one takes less time.

- There are number of problems with this approach for analysis of algorithms.
  (1) It might be possible that for some of the provided inputs, the first algorithm gives better results than the second. And for some inputs second algorithm gives better results than the first.
  (2) It might also be possible that for some of the provided inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.
- **Asymptotic Analysis** is the best solution on above issues in the process of analyzing algorithms.
- In Asymptotic Analysis, the performance is evaluated in terms of input size (the actual running time is not measured).
- It is calculated that how does the time (or space) taken by an algorithm get increased with the input size. For example, we will consider the problem of searching an element in the given sorted array.

1.4.1 Best, Worst and Average Case Analysis of an Algorithms

GQ. 1.4.2 *Explain the different cases to analyse an algorithm.* **(6 Marks)**

We can use three different cases to analyze an algorithm :

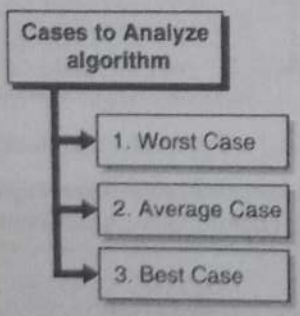


Fig. 1.4.1 : Cases to Analyze algorithm

Let us consider the following implementation of Linear Search.

```
#include <iostream.h>
/* Linearly search of ele in arr_s[]. If ele is present then
return  its index, otherwise return -1 */
int ele_search(int arr_s[], int n, int ele)
{
  int i;
  for (i=0; i<n; i++)
  {
    if (arr_s[i] == ele)
      return i;
  }
  return -1;
}

int main()
{
  int arr_s[] = {2, 7, 10, 25};
  int ele = 10;
  int n = sizeof(arr_s)/sizeof(arr_s[0]);

cout<<ele<<" is present at index "<<ele_search(arr_s, n,
ele));
  return 0;
}
```

1.4.1(A) Worst Case Analysis

- In the analysis of worst case, upper bound is calculated on running time of an algorithm.
- One should have knowledge about the case that leads to maximum number of operations to be executed.
- In case of Linear Search, the worst case will occur when the element to be searched (here *ele* in this example) is absent in the array.
- When ele is absent, the ele_search() function compares it with all the elements present in arr_s[] one by one. Therefore, the worst case time complexity of linear search would be $\theta(n)$.

1.4.1(B) Average Case Analysis

- In average case analysis, all possible inputs are taken and computing time for all those inputs is calculated.
- All the calculated values are added in sum and this sum is divided by total number of inputs.
- One should know or predict the distribution of cases.
- Here for problem of linear search, we assume that distribution of all the cases is uniform (including the case of ele not being present in array).
- Hence here summation of all the cases will be done and the sum is divided by (n + 1).

1.4.1(C) Best Case Analysis

- In the analysis of best case, lower bound is calculated on running time of an algorithm.
- One should have knowledge about the case that leads to minimum number of operations to be executed.
- In case of Linear Search, the best case will occur when the element to be searched (here ele in this example) is present at the first place in the array.
- The number of operations in the best case is constant (not dependent on n). Hence time complexity in the best case would be $\theta(1)$

1.5 STANDARD MEASURES OF EFFICIENCY

- Some algorithms are more efficient than others. User would always like to select an efficient algorithm, hence it would be better to get metrics for comparing algorithm efficiency.
- The *complexity* of an algorithm is considered as a function which describes the efficiency of the algorithm in terms of the amount of data the algorithm must process.
- In general there are natural units for the domain and range of this function. There are two important complexity measures of the efficiency of an algorithm: Space complexity and Time Complexity.

1.6 COMPLEXITY OF ALGORITHMS

UQ. 1.6.1 *What is complexity of algorithm?* **SPPU - Dec. 18, 2 Marks**

- Before going to define the actual term complexity, first we will see some real life scenarios.
- Consider an example of railway reservation counter where people come to book their tickets.
- The time required for customer to book the tickets will depend on the number of service windows available, size of queue and time taken by each representative to perform the process of ticket booking.
- A person has two options; either he can go in a queue which is very short or go in a queue where representative is very fast.
- Thus in real world, every task depends on how much time it takes to complete the execution and how much space it consumes for the process.

> *Defintion of Complexity : Complexity is defined as the running time required to execute a process and it depends on space as well as time.*

Types of Complexity

GQ. 1.6.2 *Explain types of complexity.* **(4 Marks)**

There are two types of complexity :

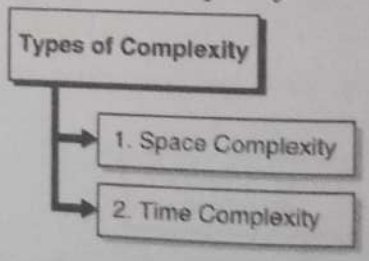


Fig. 1.6.1 : Types of Complexity

Syllabus Topic : Space Complexity

1.6.1 Space Complexity

UQ. 1.6.3 *What is space complexity of algorithm ? Explain its importance with example.* **SPPU - May 17, 3 Marks**

Definition of Space complexity

The amount of computer memory required to solve the Given Problem of particular size is called as space complexity.

- The space complexity depends on two components

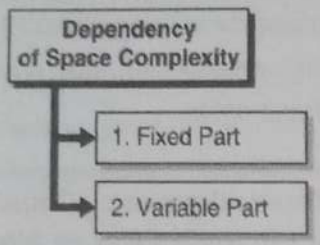


Fig. 1.6.2 : Dependency of Space Complexity

1. Fixed Part

It is required for instruction space i.e byte code. Variable space, constants space etc. This part is not dependent on characteristics of inputs and outputs.

2. Variable Part

Instance of input and output data. This part consists of space necessary for variables, size of which depends upon specific problem instance being solved, space required for reference variables and recursion stack space.

Space(S) = Fixed Part + Variable Part

Example 1

```
sum(a,b)
{
return a + b;
}
```

Here two variables(a and b) are necessary to store values hence,

Space Complexity (S)= 2.

Example 2

```
sum( a[], n)
{
sum =0;
for(i=0;i<=n;i++)
{
sum = sum + a[i];
return sum;
}
```

Space complexity is calculated as follows :

- One computer word is required to store n.
- n space is required to store array a[].
- One space for variable sum and one for i is required.

Total Space(S) = 1 + n +1 +1 = n + 3

Syllabus Topic : Time Complexity

1.6.2 Time Complexity

GQ. 1.6.4 Define and explain term : Time Complexity. **(3 Marks)**

Definition of Time Complexity : The time which is required for analysis of given problem of particular size is known as the Time Complexity.

Time complexity depends on two components.

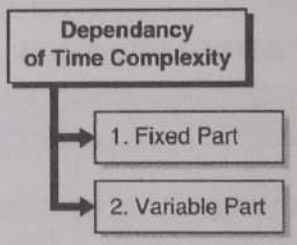


Fig. 1.6.3 : Dependency of Time Complexity

1. Fixed Part : Compile time.

2. Variable Part

- Run time which depends upon problem instance.
- Usually run time is considered and compile time is ignored.

Different methods to measure time complexity

1. **Stop watch :** Use a stop watch to get time in seconds or milliseconds.
2. **Step Count** : Count number of program steps.

- As comments are not evaluated, they are not considered as program step.
- In while loop steps are equal to the number of times loop gets repeatedly executed.

- In for loop, steps are equal to number of times an expression is checked for condition.
- A single expression is considered as a single step.
- **Example :** a + b +c + d + d / a – b - f is one step.

☞ **Example of step count**

GQ. 1.6.5 *Write an example of Step count.* **(4 Marks)**

```
sum( a[], 5)
{
    sum =0;
    for(i=0;i<=5;i++)
    {
            sum = sum + a[i];
      }
    return sum;
}
```

Table 1.6.1 : Step Count

Instruction	Step Count
Algorithm sum(a[ ], n)	0
{	0
sum = 0;	1
for(i=0;i<=n;i+)	n + 1
{	0
sum = sum + a[i]	N
}	0
return sum;	1
}	0
Total Steps = 1 + n + 1 + n + 1 = 2n + 3	

- Step count is generally considered as a difficult approach if there is need to compare results.
- For example if we have used two different techniques to solve a single problem and T(1) is the running time of first technique while T(2) is the running time of second technique.

Say

$T(1) = (n + 1)$

and

$T(2)$ is $(n^2 + 1)$.

- It is difficult to decide which is the better technique, hence for the purpose of comparisons 'Rate of growth' i.e. asymptotic notations of time space complexity functions is better option.

Syllabus Topic : Frequency Count and its importance in Analysis of an Algorithm

1.7 FREQUENCY COUNT AND ITS IMPORTANCE IN ANALYSIS OF AN ALGORITHM

- A better way to analyze an algorithm is to realize that the running time of a program is determined primarily by two factors :
  - The cost of executing each statement.
  - The frequency of execution of each statement
- A primitive operation is defined to be an operation that corresponds to a low-level, basic computation with a constant execution time. Examples of primitive operations include evaluating an expression, assigning a value to a variable, indexing into an array, etc.
- Then, to analyze an algorithm, we simply determine the frequency of primitive operations, and characterize this as a function of the input size. In doing so, we have the benefit of taking into account all possible inputs, and our result is independent of any hardware/software environment.
- In counting the frequency of primitive operations in programs, it is useful to note that programs typically use loops to enumerate input data items. As such, we can often just count the number of operations or steps taken inside of the loops present in the program (where each statement within a loop is counted as a "step").

Let's take a look at some examples.

Example 1

```
sum = 0.0;
for (int i = 0; i < n; i++) {
  sum += array[i]; // <----- a primitive operation
}
```

- Ignoring the update to the loop variable ii, we note the marked statement above is executed n times, so the *cost function* is C(n) = nC(n) = n.

Example 2

```
sum = 0.0;
for (int i = 0; i < n; i += 2) {
  sum += array[i];  // <----- a primitive operation
}
```

- Here however, since the counter variable ii is going up by 2 with each pass through the loop, the number of times the marked statement above gets executed this time is halved -- leading to

C(n)=n/2C(n)=n/2.

1.8 ASYMPTOTIC NOTATIONS

UQ. 1.8.1 *What are different asymptotic notation ?*

SPPU -May 17, Dec. 18, 6 Marks

UQ. 1.8.2 *Explain Big "oh" (O), Omega and Theta notations with an example.*

SPPU - Dec. 19, 6 Marks

- **Asymptotic notations** are used to get the rate of growth. Rather than dealing with exact expressions, it is possible to deal with asymptotic behaviour.
- The meaning of asymptotic is to approach a value or curve arbitrarily closely.
- The main spotlight is on the exponential behaviour of the specified equation.

Syllabus Topic : Big O Notation

1.8.1 Big O Notation

UQ. 1.8.3 *Explain Big O notation with example.*

SPPU - Dec. 16, 2 Marks

- There are always multiple solutions while resolving a computer related problem. All such individual solutions will usually be in the form of different algorithms or instructions having different logics and there is need to compare the algorithms to find out which one is more proficient solution.
- In such situation the Big O analysis provide programmers some basis regarding computing and measuring the efficiency of a particular algorithm.
- We can conclude that the Big - O helps to identify the time and space complexity of the algorithm.
- That means using Big - O notation, the time taken by the algorithm and the space required to run the algorithm is determined.
- Big - Oh notation is used to define the upper bound of an algorithm in terms of Time Complexity.
- That means 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...

> **Definition** : *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 >= nO, C > O and nO >= 1. Then we can represent f(n) as O(g(n)).*

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

- Consider the graph of Fig. 1.8.1 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.

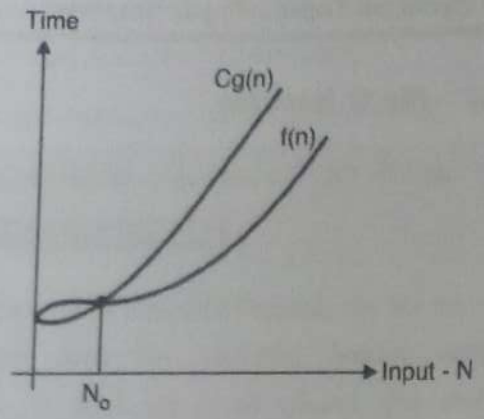


Fig. 1.8.1

- In Fig. 1.8.1 after a particular input value n0, always Cg(n) is greater than f(n) which indicates the algorithm's upper bound.
- The best way to understand Big - Oh thoroughly is to produce some examples in code. So, below are some common orders of growth along with descriptions and examples.

1.8.1(A) Orders of Growth Functions

GQ. 1.8.4 *Explain orders of growth in Big O notation.* **(4 Marks)**

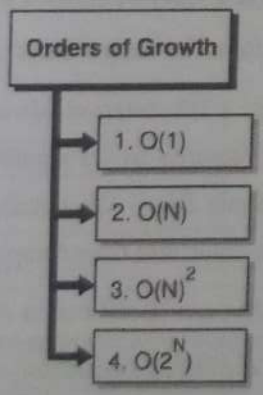


Fig. 1.8.2 : Orders of Growth

1. O(1)

- O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

```
bool IsFirstElementNull(IList<string> elements)
{
  return elements[0] == null;
}
```

2. O(N)

- O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.
- The example below also demonstrates how Big - Oh favours the worst-case performance scenario.
- A matching string could be found during any iteration of the for loop and the function would return early.
- But Big - Oh notation will always assume the upper limit where the algorithm will perform the maximum number of iterations.

```
bool ContainsValue(IList<string> elements, string value)
{
  foreach (var element in elements)
  {
    if (element == value) return true;
  }
  return false;
}
```

3. O(N)^2

- O(N^2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set.
- This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in O(N^3), O(N^4) etc.

```
bool ContainsDuplicates(IList<string> elements)
{
  for (var outer = 0; outer < elements.Count; outer++)
  {
    for (var inner = 0; inner < elements.Count; inner++)
    {
      if (outer == inner) continue;
      if (elements[outer] == elements[inner]) return true;
    }
  }
  return false;
}
```

4. $O(2^N)$

- $O(2^N)$ denotes an algorithm whose growth doubles with each addition to the input data set.
- The growth curve of an $O(2^N)$ function is exponential – which is rising meteorically.
- An example of an $O(2^N)$ function is the recursive calculation of Fibonacci numbers :

```
int Fibonacci(int number)
{
  if (number <= 1) return number;
  return Fibonacci(number - 2) + Fibonacci(number - 1);
}
```

1.8.1(B) Limitations of Big 'O' Notation

GQ. 1.8.5 *State limitations of the Big 'O' notation.* **(3 Marks)**

1. The Big O time complexity of an algorithm only considers large input.
2. Two algorithms could have the same Big-O time complexity but one might be practically faster than the other.
3. Big O only gives the worst case bound. For competition purposes, one can often assume test data which will exhibit worst case behaviour, however for most practical test cases, the time taken may be far below the worst case bound.
4. Big O ignores constants whether it is constant overhead or constant factor. But sometimes in practical solutions, these can be substantial and can't be ignored.

Syllabus Topic : Omega Ω

1.8.2 Omega (Ω) Notation

UQ. 1.8.6 *Explain Omega notation with example.* **SPPU - Dec. 16, 2 Marks**

- Big - Omega notation (Ω) is used to define the lower bound of an algorithm in terms of Time Complexity.
- That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows :

Definition of Big - Omega Notation

Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C x g(n) for all n >= n0, C > 0 and n0 >= 1. Then we can represent f(n) as Ω(g(n)).

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

- 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.

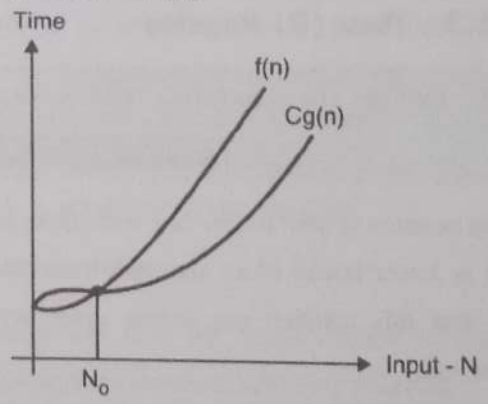


Fig. 1.8.3

- In above graph after a particular input value n0, always C x g(n) is less than f(n) which indicates the algorithm's lower bound.

Example

Consider $F(n) = 3n^2+5$ and $g(n) = 8n$

For n=1

$F(n) = 3(1)2+5$

$= 3+5$

$= 8$

$g(n) = 8n$

$= 8(1)$

= 8

F(n) = g(n)

If n = 2 then,

F(n) = 3(2)2+5

= 12+5

=17

g(n) = 8n

= 8(2)

= 16

F (n) > g (n)

Thus for n>2 we get F (n)>c * g (n).

Syllabus Topic : Theta Notation

1.8.3 Theta (Θ) Notation

UQ. 1.8.7 *Explain Theta notation with example.*

SPPU - Dec.16, 4 Marks

- Theta notation is used to describe both upper bound as well as lower bound of an algorithm hence it can be said that this notation can define exact asymptotic behaviour.
- In the real case scenario, it cannot be assumed that the algorithm will for all time run on best and worst cases, the average running time is said to be lies between best and worst and theta notation can be used to represent it.
- An easy method to get Theta notation of any expression is dropping of low order terms and ignoring the respective leading constants.
- For example, consider the following expression:

$3n^3 + 6n^2 + 6000 = \Theta(n^3)$

- It is always better to drop lower order terms since there will always be a n0 after which $\Theta(n^3)$ has higher values than Θn^2 regardless of the constants involved.
- For a given function g(n), we denote $\Theta(g(n))$ is following set of functions.

$\Theta(g(n)) = \{f(n)$: there exist positive constants c1, c2 and n_0 such that $0 <= c1*g(n) <= f(n) <= c2*g(n)$ for all $n >= n_0\}$

- The above definition indicates that, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n ($n >= n_0$).
- The definition of theta also needs that f(n) should not be negative for values of n greater than n_0.

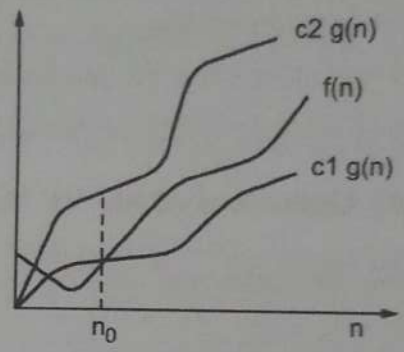


Fig. 1.8.4

Syllabus Topic : Sequential Organization

1.9 SEQUENTIAL ORGANIZATION

GQ. 1.9.1 *Explain following term : Sequential Organization.* **(1 Mark)**

- In computer science, sequential access means a group of elements is accessed in a predetermined, ordered sequence.
- In data structures, a data structure is said to have sequential access if one can only visit the values it contains in one particular order. The canonical example is an Array.

1.9.1 Concept of Linear Data Structures

GQ. 1.9.2 *Define linear data structures.* **(1 Mark)**

Definition of linear data structure

The data structure where data items are organized sequentially or linearly one after another is called linear data structure.

- Data elements in a linear data structure are traversed one after the other and only one element can be directly reached while traversing. All the data items in linear data structure can be traversed in single run.
- These kinds of data structures are very easy to implement because memory of computer also has been organized in linear fashion.

1.9.2 Array

UQ. 1.9.3 *Explain concept of array with suitable example.* **SPPU - Dec. 16, 4 Marks**

- Let's consider a situation where a program needs number of similar type of data elements to be stored. A variable is used to store one data element at a time. So to store number of data elements; number of variables are required. This solution has many problems :
  - As the list of variables increases the length of program also increases.
  - To manipulate those variables several assignment statements also needed.
  - Programmer needs to remember names of all the variables.
- An alternate solution to above situation is to store similar type of data elements is create an **array**.

GQ. 1.9.4 *What is array ?* **(2 Marks)**

Definition of Array : Array is a collection of elements of similar data types referred by the same variable name. Contiguous memory block is allocated to all these array elements.

- Array elements are of same data type i.e. once array is declared as integer, then all values which are present in an array will be of type integer only.

1.9.3 Need of Array

First scenario

- To store weight of 5 children we need 5 variables as follows :

```
void main( )
{
int rajesh, suvrna, surya, pritesh, chitra;
rajesh = 25;
chitra = 20;
suvrna = 30;
surya = 19;
pritesh = 22;
}
```

- If count of variables is small then this idea will work. Similarly to store weight of 100, 1000, or more children programmer should declare that much number of variables which is bad idea. Also it is difficult to retrieve particular child's weight as the programmer needs to remember all the variables.

Second Scenario

- Another idea is to use single variable to store these information as shown in following program.

```
void main( )
{
int weight ;
weight = 25 ;
weight = 20 ;
cout<< "\n weight of child = "<< weight;
}
```

- No doubt, this program will print the most recently updated value of the variable i.e. **20**. Because initially the variable **weight** is assigned to 25; and when 20 is assigned to **weight**, previous value of **weight** i.e. 25 is lost and new value i.e. **20** is assigned to **weight**. A situation in which more than one values needs to be stored at a time, this idea will not work.

- Array can handle lots of elements using single name. Instead of declaring lots of variables, use a subscripted variable called as array to store the weights of children. Using the subscript we can manage random elements of the array.

Example

- Consider the following group

weight = { 25, 20, 30, 19, 22}

- It represents weights of five children. To refer the element 20 of the group the notation weight 2 is used. Similarly, to refer the element 22 of the group the notation weight 5 is used.
- But in the case of array, the element 25 is referred as **weight[0] ; b**ecause the array

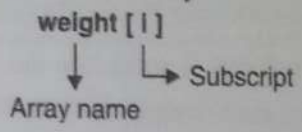


Fig. 1.9.1

- Element's counting starts with 0 instead of 1. **Similarly,** the element 22 is referred as **weight[4].** In general to refer any element of an array following notation should be used
- Here **weight** is the subscripted variable (array), whereas **i** is its subscript. Where i is in the range 0-4 in our example.

☞ **Need of array**

- Array variables are needed because of following requirements of user :
  - Store multiple values in the variable with same name.
  - Store multiple values (with same data type) together i.e. store them one after another.
  - For easy access or retrieval of stored data elements.
- Note that the elements should be of same data type means it can be group of integers (**int** s), group of real numbers (**float** s or **double** s), group of characters (**char** s), etc. Usually array of character is called as string.

1.9.4 Characteristics of an Array

GQ. 1.9.5 *What are the characteristics of array ?* **(2 Marks)**

1. An array can hold only similar type of data.
2. Array is a subscripted variable which holds several elements at a time.
3. Using subscript random elements can be referred.
4. Contiguous memory is allocated for elements of array.
5. Always the array_name is followed by [ ] which tells the compiler that we are dealing with an array. Between [and] bracket int type of value is needed.
6. An array is a collection of similar elements.
7. The first element in the array is numbered 0, so the last element is 1 less than the size of the array.
8. Before using an array its type and dimension must be declared.

1.9.5 Array Declaration

- While using array in a program the first step is to declare an array. An array declaration is performed to tell the compiler what type of data the array will hold and how much data elements it can hold.

Syntax

Following syntax is used to declare an array :

```
data_type array_name [size];
```

Example

- To declare an array which holds age of five persons following statement is used.

```
int age[5];
```

- Here, **int** is the data type of the array **age** and the number **5** specifies the maximum number of elements which the age array can hold.

Arrangement of array elements in memory after declaration

- As soon as the control goes to the statement of declaring array **age[ ]**, Immediately contiguous memory block of 10 bytes get reserved in memory for 5 integers.
- According to 16 bit compiler each integer occupies 2 bytes in memory; so 5 * 2 = 10 bytes should be reserved for array of 5 integers.
- The memory structure would be as follows :

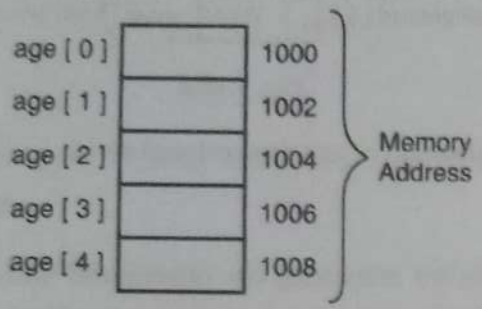


Fig. 1.9.2 : Memory structure after declaring array of 5 integers

- Size of array can be calculated by using the sizeof() operator. Like :

```
sizeof(age);
```

it will give the memory space occupied by the array in bytes. Suppose age is an integer array and has the capacity of storing 5 numbers. Then the **sizeof(age)** will output 5 * 2 = **10.**

- All the locations will contain garbage values until we initialize the array.

1.9.6 Array Initialization

- Array initialization is the process of assigning the value to array elements. After declaring the array it will contain garbage values until we initialize it. There are different ways to initialize an array.

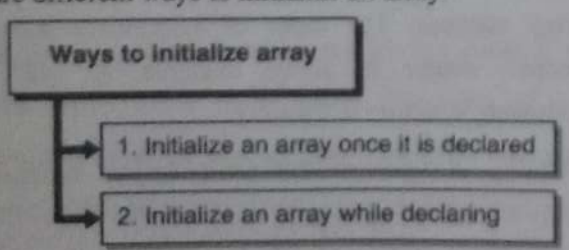


Fig. 1.9.3 : Ways to initialize array

1. Initialize an array once it is declared

Example

```
int age[5];          // declaring array
age [0] = 20;        // initialize 20 to first location
age [1] = 25;        // initialize 25 to second location
age [2] = 30;        // initialize 30 to third location
age [3] = 40;        // initialize 40 to fourth location
age [4] = 45;        // initialize 45 to fifth location
```

2. Initialize an array while declaring

We can combine the declaration and initialization of array like :

Example

```
int age [5] = {20,25,30,40,45};
int age [ ] = {20,25,30,40,45};
```

- If the array is initialized and declared at a time then the size of array is optional (as shown in example 3). The values going to be initialized are written between two curly braces (i.e. { and }) and separated by comma(,).
- Above two statements assigns the value **20** to first element i.e. **age[0]**, **25** is assigned to second element i.e. **age [1]**, **30** is assigned to third element i.e. **age [2]**, **40** is assigned to fourth element i.e. **age [3]**, and **45** is assigned to fifth element i.e. **age [4]**.
- Arrangement of array elements in memory after initialization shown in Fig. 1.9.4.
- Only the difference between Examples 1 and 2, 3 is that the order of assignment statements in Example 1 will not matter while arranging the array element in memory. After execution of any of the Examples 1, 2, or 3 there will be same memory structure as follows :

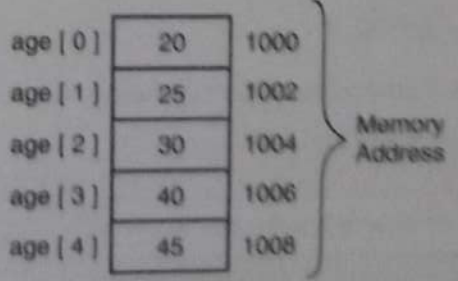


Fig 1.9.4 : Memory structure after initializing the array of 5 integers

1.10 TYPES OF ARRAY

GQ. 1.10.1 *Explain types of array.* **(4 Marks)**

- Arrays are categorized according to the dimensions used to define them. Here dimension indicates the number of rows and columns used to set size of array.
- Array is categorized into following types :
(Refer Fig. 1.10.1)

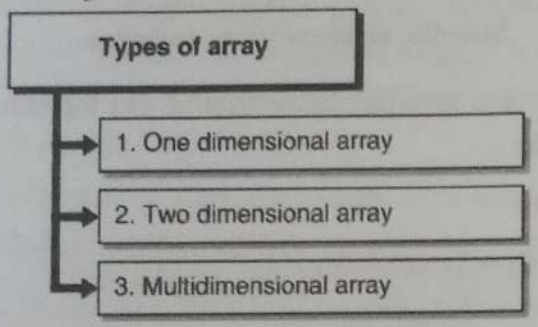


Fig. 1.10.1: Types of array

Syllabus Topic : Single Dimensional Array

1.10.1 Single Dimensional Array

- An array with single subscript is called as **one dimensional array**.

Declaration of one dimensional array

Syntax

- Syntax for declaring one dimensional array is given below.

```
data_type array_name [ size ];
```

Example

- Declare an array to store percentage of 10 students.

```
float percentage[10];
```

- Memory arrangement after declaring one dimensional array shown in Fig. 1.10.2.

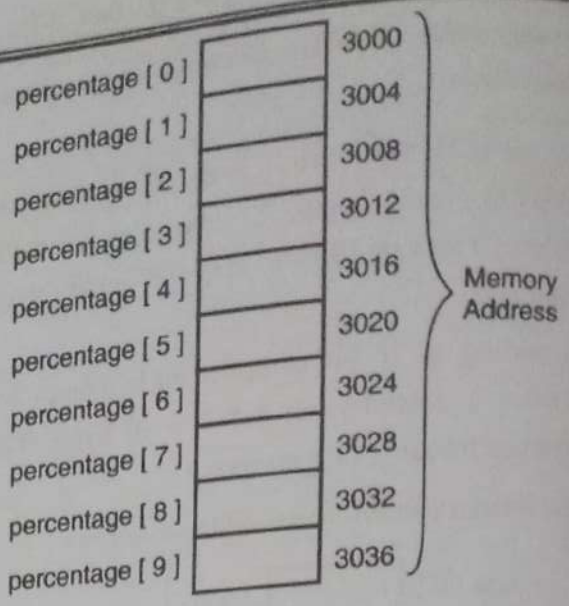


Fig. 1.10.2

Initialization of one dimensional array

Syntax

- Syntax for initializing one dimensional array is given below.

```
array_name [subscript] = value;
```

Example

- Initialize an array to store percentage of 10 students

```
percentage[0] = 99.07;
percentage[1] = 54.04;
percentage[2] = 60.60;
percentage[3] = 79.47;
percentage[4] = 92.70;
percentage[5] = 80.60;
percentage[6] = 85.27;
percentage[7] = 90.30;
percentage[8] = 99.70;
percentage[9] = 94.50;
```

- Using the subscript we can assign value to specific array element. The order of assignment statement doesn't matter in above example. It will form following structure in memory.
- Memory arrangement after declaring one dimensional array shown in Fig. 1.10.3.

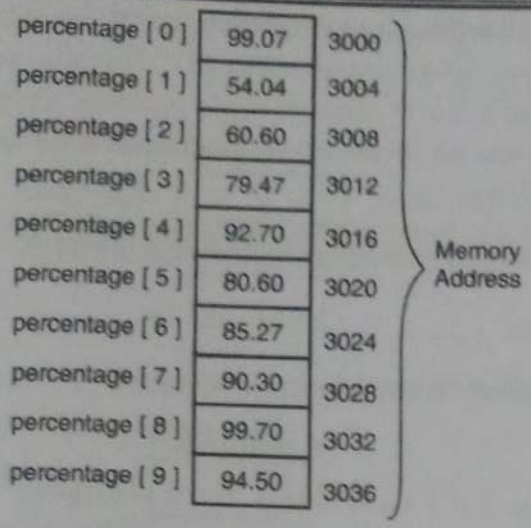


Fig. 1.10.3

Entering or Writing Data into One Dimensional Array

- We can enter data into array by array initialization or by accepting array elements from user. We have seen how to initialize data, now we are going to study how to accept array element from user and store it in array.
- The following code is used to accept ten elements from user and store into one dimensional array:

```
for ( i = 0 ; i <10 ; i++ )
{
    cout<< "\nEnter Percentage of student "<<i+1 ;
    cin>> percentage[i] ;
}
```

- In this example for loop is used to repeat the statements 10 times.
- cout<<) function is used to request user to enter the data.
- cin>>) function is used to accept input from user.
- &percentage[i] tells the compiler the location where the entered value to be stored. The value of i varies from 0 to 9 so percentage[i] specifies the position where the entered elements are stored in the array.

Accessing or Reading Data from an One Dimensional Array

- While accessing array elements we can use loop just like used in writing the data into array. Only difference is that in reading data from array doesn't require the **cin>>)** function.
- The following code is used to access elements of one dimensional array:

```
for ( i = 0 ; i <10 ; i++ )
{
    cout<< "\n Percentage of student : "<< i+1<<" :
"<< percentage[i] ;
}
```

- The **for** loop is used to repeat the statements 10 times.
- **cout<<)** function is used to display the array elements on screen.
- The value of **i** varies from 0 to 9 so **percentage[i]** specifies which array element to be read.

Program 1.10.1

Write a program to declare array to store percentage of 10 students. Accept percentage from user and print on the screen.

Soln. :

Program

```
#include <conio.h>
#include <iostream.h>
int main()
{
float percentage[10];
int i;
for ( i = 0 ; i <10 ; i++ )
{
cout<< "\nEnter Percentage of student : "<< i+1 ;
cin>>percentage[i] ;
```

Array declaration

Accepting array elements from user.

```
}
for ( i = 0 ; i <10 ; i++ )
{
cout<<"\n Percentage of student i+1<<" : "<<
                                    percentage[i] ;
}
return 1;
}
```

Output

```
E:\tejs\pheonix\Untitled1.exe

Enter Percentage of student 1 :  83.67
Enter Percentage of student 2 :  89.45
Enter Percentage of student 3 :  76.98
Enter Percentage of student 4 :  34.65
Enter Percentage of student 5 :  65.34
Enter Percentage of student 6 :  67.34
Enter Percentage of student 7 :  98.87
Enter Percentage of student 8 :  56.89
Enter Percentage of student 9 :  97.67
Enter Percentage of student 10 :          67.87

 Percentage of student 1 :        83.67
 Percentage of student 2 :        89.45
 Percentage of student 3 :        76.98
 Percentage of student 4 :        34.65
 Percentage of student 5 :        65.34
 Percentage of student 6 :        67.34
 Percentage of student 7 :        98.87
 Percentage of student 8 :        56.89
 Percentage of student 9 :        97.67
 Percentage of student 10 :       67.87
```

Syllabus Topic : MultiDimensional Array

1.10.2 MultiDimensional Array

A. Two Dimensional Array

UQ. 1.10.2 *Explain the two-dimensional array in detail with column and row major representation and address calculation in both the cases.* **SPPU - Dec.16, 6 Marks**

- An array with two subscripts is called as **two dimensional array**. 2D arrays are mostly used to perform matrix operations.

☞ **Declaration of two dimensional array**

Syntax

- Syntax for declaring two dimensional array is given below.

```
data_type array_name [ row size ][ column size ];
```

Example

- Declare an array to store a 2 × 2 matrix

```
int matrix[2][2];
```

- The 2 × 2 matrix stores 2 * 2 = 4 values. In our example data type of array is integer means the four values are of int type.
- In user's view the matrix looks like :

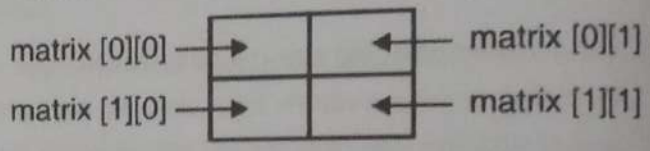


Fig. 1.10.4

- But in memory it will form different structure.
- Memory arrangement after declaring two dimensional array shown in Fig. 1.10.5.

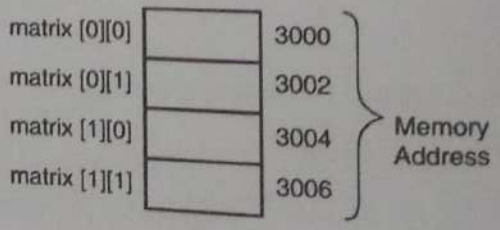


Fig. 1.10.5

☞ **Initialization of two dimensional array**

Syntax

- Syntax for initializing 2D array is given below:

```
array_name [row_number] [column_number] = value;
```

Example

- Initialize an array to store values for 2 × 2 matrix

```
matrix[1][1] = 9;
matrix[0][0] = 7;
matrix[0][1] = 6;
matrix[1][0] = 5;
```

- Using the subscript we can assign value to specific array element. The order of assignment statement doesn't matter in above example. It will assign the values given as in Fig. 1.10.6.

matrix [0][0]	7	6	matrix [0][1]
matrix [1][0]	5	9	matrix [1][1]

Fig. 1.10.6

- In memory initialized values are stored in sequential row as shown in Fig. 1.10.7.

matrix [0][0]	7	3000
matrix [0][1]	6	3002
matrix [1][0]	5	3004
matrix [1][1]	9	3006

Memory Address

Fig. 1.10.7

Entering or Writing Data into Two Dimensional Arrays

- We can enter data into array by array initialization or by accepting array elements from user.
- We have seen how to initialize data, now we are going to study how to accept array element from user and store into 2D array.
- The following code is used to accept values for 2 × 2 matrix from user and store into 2D array.

```
cout<< "\nEnter data for 2D matrix\n" ;
for ( i = 0 ; i <2 ; i++ )          ---> Outer for loop
{
    for ( j = 0 ; j <2 ; j++ )      ---> Inner for loop
    {
    cout<< "\n matrix [ "<<i<<"] ["<<j<<"]" ;
    cin>>matrix[i][j];
    }
}
```

- For 2D array, two for loops are used. One for loop (i.e. outer for loop) for row and another for loop (i.e. inner for loop) is for column.
- **cout<<)** function is used to request user to enter the data.
- **cin>>)** function is used to take input from user. **&matrix[i][j]** tells the compiler to store the entered value at i^{th} row and j^{th} column in matrix. Here value of **i** and **j** varies from **0** to **1.**

Accessing or Reading Data from an Two Dimensional Array

- As we use two for loops for writing the data into 2D array, likewise we can use two for loops to read the data from 2D array.
- Only difference is that in reading data from array doesn't require the **cin>>)** function. The following code is used to access elements of two dimensional array:

```
cout<<"\nThe matrix is:\n";
for ( i = 0 ; i <2 ; i++ )
{
    for ( j = 0 ; j <2 ; j++ )
    {
    cout<< matrix[i][j] ;
    }
    cout<<"\n";
}
```

➢ Program 1.10.2

Write a program to accept values for 2 x 2 matrix and print them.

☑ Soln. :

Program

```
# include<iostream.h>
#include<conio.h>
int main()
{
int matrix[2][2],i,j;
cout<< "\nEnter data for 2D matrix\n" ;
for ( i = 0 ; i <2 ; i++ )
{
    for ( j = 0 ; j <2 ; j++ )
    {
    cout<< "\n matrix [ "<<i<<"] ["<<j<<"]" ;
    cin>>matrix[i][j] ;
    }
}
```

Accepts array elements from user for 2D array.

```
cout<<"\nThe matrix is:\n";
for ( i = 0 ; i <2  ; i++ )
{
    for ( j = 0 ; j <2  ; j++ )
    {
     cout<<matrix[i][j];
    }
    cout<<"\n";
}
return 1;
}
```

Prints 2D array elements

Output

```
E:\teju\pheonix\2darray.exe
Enter data for 2D matrix
 matrix [0] [0] :        12
 matrix [0] [1] :        1
 matrix [1] [0] :        32
 matrix [1] [1] :        10
The matrix is:
 12   1
 32   10
```

B. More than 2- dimensions

GQ. 1.10.3 *Write syntax to declare 3D arrays.* **(2 Marks)**

- An array with more than two subscripts is called as **multidimensional array**.

☞ **Declaration of multidimensional array**

- For simplicity we will study 3D array which has 3 subscripts.

Syntax

- Syntax for declaring multidimensional array is given below:

```
data_type array_name [size1][size2][size3];
```

Example

- Declare an multidimensional array

```
int array[2][2][2];
```

- This array stores 2 * 2 * 2 = 8 values. In this example data type of array is integer means the values should be of int type.
- In user's view the multidimensional array looks like:

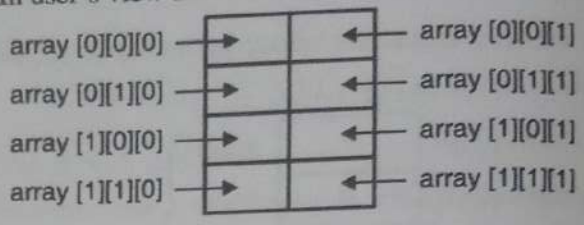


Fig. 1.10.8

- But in memory it will form different structure.
- Memory arrangement after declaring multidimensional array shown in Fig. 1.10.9.

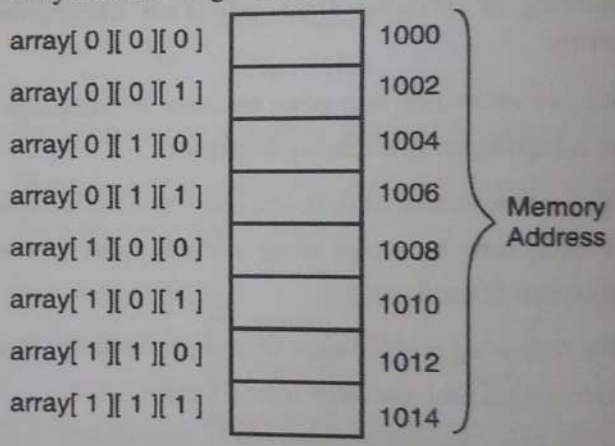


Fig. 1.10.9

☞ **Initialization of multidimensional array**

Syntax

- Syntax for initializing multidimensional array is given below .

```
array_name [row_number] [row_number] [column_number] = value;
```

Example

- Initialize the multidimensional array

```
array[0][1][1] = 9;
array[0][0][0] = 7;
array[0][0][1] = 6;
array[0][1][0] = 5;
array[1][0][0] = 2;
array[1][1][1] = 3;
array[1][0][1] = 4;
array[1][1][0] = 8;
```

- Using the subscript we can assign value to specific array element. The order of assignment statement doesn't matter in above example. It will assign the values given as below.

array [0][0][0]	7	6	array [0][0][1]
array [0][1][0]	5	9	array [0][1][1]
array [1][0][0]	2	4	array [1][0][1]
array [1][1][0]	8	3	array [1][1][1]

Fig. 1.10.10

- In memory initialized values are stored in sequential row by row manner as shown in Fig. 1.10.11.

array[0][0][0]	7	1000
array[0][0][1]	6	1002
array[0][1][0]	5	1004
array[0][1][1]	9	1006
array[1][0][0]	2	1008
array[1][0][1]	4	1010
array[1][1][0]	8	1012
array[1][1][1]	3	1014

Memory Address

Fig. 1.10.11

☞ **Entering or Writing Data into multidimensional Array**

- The following code is used to accept ten elements from user and store into 3D array :

```
cout<< "\nEnter data for 3D array \n" ;
for ( i = 0 ; i <2 ; i++ )
{
    for ( j = 0 ; j <2 ; j++ )
    {
    for ( k = 0 ; k <2 ; k++ )
     {
    cout<< "\n array [ "<<i<<"] ["<<j<<"]
["<<k<<"]";
     cin>>array[i][j][k] ) ;
    }
    }
}
```

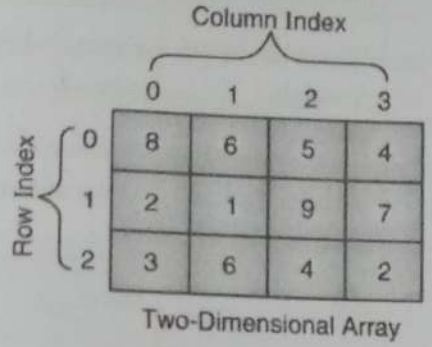
- As we know for 2D array two for loops are used similarly for 3D array three for loops will be used.
- **cout<<)** function is used to request user to enter the data.
- cin>>) function is used to take input from user.
- &array[i][j][k] tells the compiler to store the entered value at j^{th} row and k^{th} column of i^{th} row in matrix. Here value of **i, j** and **k** varies from **0** to **1.**

☞ **Accessing or Reading Data from an multidimensional Array**

- As we use three for loops for writing the data into 3D array likewise we use three for loops to read the data from 3D array.
- Only difference is that in reading data from array doesn't require the **cin>>)** function. The following code is used to access elements of 3D array:

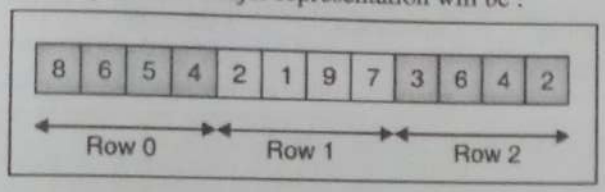
```
cout<<"\nThe array is:\n";
for ( i = 0 ; i <2 ; i++ )
{
    for ( j = 0 ; j <2 ; j++ )
    {
     for ( k = 0 ; k <2 ; k++ )
     {
    cout<< array[i][j][k] ;
     }
    }
    cout<<"\n";
}
```

- For row-major order, all elements of the first row come before all elements of the second row, etc.
- Consider following array.

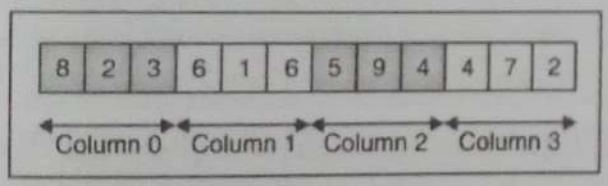


Two-Dimensional Array

For this array the row-major representation will be :



- For column-major order, all elements of the first column come before all elements of the second column, etc.
- For the given array the column-major representation will be :



☞ **Address Calculation of 2D array**

A. Row major representation

- Address of [i][j] = base address + i *c * element_size
  + j * element_size
  = base address + (i*c*+j) * element_size

Example

- Adresss of arr[1][2] (Consider base address as 1000)

  Arr[1][2 = 1000 + (1*3+2) * sizeof(int)
  = 1000 + 10 = 1010

B. Column major representation

Address of [i][j] = base address + j *r * element_size
+ i * element_size
= base address + (j*r*+i) * element_size

Example

- Adresss of arr[1][2] (Consider base address as 1000)

  Arr[1][2] = 1000 + (2*4+1) * sizeof(int)
  = 1000 + 18 = 1018

Syllabus Topic : Concept of Linked Organization

1.11 CONCEPT OF LINKED ORGANIZATION

- Array is a group of elements with same data type.
- Array is considered as an example of static memory allocation. i.e. the size of array is fixed. While declaring an array we have to compulsory mention the size.

Example

int arr[5];

Element					
Index	0	1	2	3	4

- This is an array *arr* of type *int* and size 5.
- Size indicates the maximum elements which can be stored in the array.
- In this array we can store maximum of 5 elements.
- But in real life application, it is difficult to predict the maximum elements to be stored as the data is accepted from end user at run time.
- Hence if there is requirement of storing elements more than size of the array, then memory shortage occurs and we cannot store more elements than the size of an array.
- Now in the above array, if only three elements are stored, then the memory will be wasted.

Memory Waste

Element	11	12	13		
Index	0	1	2	3	4

- Here the total memory wastage is 2 + 2 = 4 bytes.
- Array has both memory shortage as well as memory wastage problems.

– To solve these problems, DS provides the concept of linked list which is an example of **dynamic memory allocation**.

1.11.1 Comparison between Array and Linked List

UQ. 1.11.1 *Compare array and link list*

SPPU - May 17, 3 Marks

UQ. 1.11.2 *Compare linked list with arrays with reference to the following aspects :*
(i) Accessing any element randomly
(ii) Insertion and deletion of an element
(iii) Utilization of computer memory.

SPPU - Dec. 19, 8 Marks

Parameters	Array (Sequential Organization)	Linked list (Linked Organization)
Basic	It is a consistent set of a fixed number of data items.	It is an ordered set comprising a variable number of data items.
Size	Specified during declaration.	No need to specify; can grow and shrink during execution.
Storage Allocation	Element position is allocated during compile time.	Element position is assigned during run time.
Order of the elements	Stored consecutively	Stored randomly
Accessing the element	Direct or randomly accessed by specifying the array index.	Sequentially accessed, i.e., Traverse starting from the first node in the list by the pointer.
Insertion and deletion of element	Slow relatively as shifting is required.	Easier, fast and efficient.
Searching	Binary search and linear search	Linear search
Memory required	less	More
Memory Utilization	Ineffective	Efficient

GQ. 1.11.3 *What is Link List ?* **(2 Marks)**

❑ *Definition of Linked list : Linked list is a linear data structure which consists of group of elements called as nodes. Every node contains two parts: data and next. Data contains the actual element while next contains the address of next node.*

GQ. 1.11.4 *Draw representation of linear linked list.* **(2 Marks)**

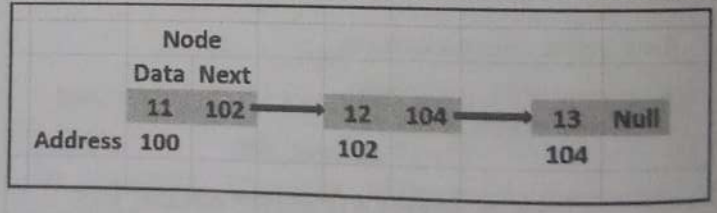


Fig. 1.11.1 : Representation of linear linked list

1.11.2 Basic Terminologies of Linked List

GQ. 1.11.5 *Define the term basic terminologies of linked list.* **(1 Mark)**

☞ **Node**

– Every element in a linked list is called as a node. A node consists of two parts, Data and Next.

- Data contains the actual element while next contains the address of next node.

☞ **Information**

- Information is nothing but the actual element (record) which is present in the data part of node.
- E.g. roll number or name of student.

☞ **Next**

GQ. 1.11.6 *Define the term : next pointer of linked list.* **(1 Mark)**

- Next is second part of a node. It contains address of next node. It helps to go from one node to other node.

☞ **Address**

GQ. 1.11.7 *Define the term : address for linked list.* **(1 Mark)**

- Next part of every node contains the memory address of next node. This is very useful to go from one node to another i.e. for traversing purpose.
- The last node always contains NULL value in the next part as there is no next node.
- When there is single node, then it contains NULL value in the next part.

☞ **Pointer**

- A pointer always points to the first node of the linked list. The first node to which the pointer points is called as **head or header node.**
- Pointer is considered as reference to the Linked List.

☞ **NULL Pointer**

GQ. 1.11.8 *Define the term : NULL pointer for linked list.* **(1 Mark)**

- When the linked list is empty, i.e. there is no node in it then NULL value is set to the pointer.

E.g. P = NULL;

☞ **Empty List**

GQ. 1.11.9 *Define the term empty list for linked list.* **(1 Mark)**

- When there is no node in any linked list, then it is called as empty linked list.

1.11.3 Advantages of Linked List

GQ. 1.11.10 *State advantages of linked list..* **(2 Marks)**

1. Linked lists are dynamic data structures, which can grow or shrink (by allocating or de-allocating memory) while the program is running.
2. Insertion and deletion operations can be easily implemented in a linked list.
3. Dynamic data structures such as stacks and queues can be implemented using a linked list.
4. There is no need to define initial size for a linked list.
5. Items can be added or removed from the middle of the list.
6. Backtracking is possible in two way linked list (Doubly Linked List).

1.11.4 Disadvantages of Linked List

GQ.1.11.11 *Discuss disadvantages of linked list over array.* **(2 Marks)**

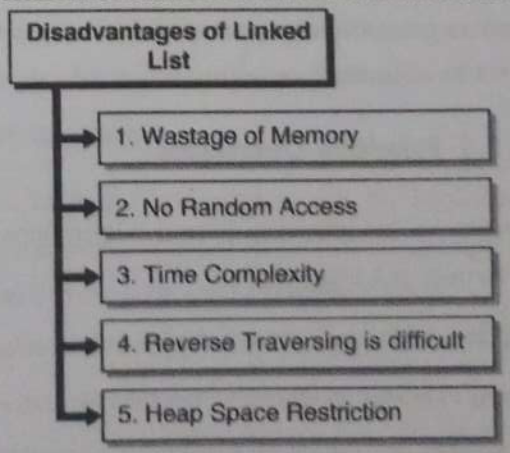


Fig. 1.11.2 : Disadvantages of Linked List

1. Wastage of Memory

In linked list Pointer needs extra memory for storage.

2. No Random Access

- In array it is possible to access n^{th} element with ease just by using a[n].
- In Linked list there is no random access provided to user, user has to access each node sequentially.
- Just consider that if user wants to access n^{th} node then he has to traverse linked list upto n^{th} node.

3. Time Complexity

- In linked list the nodes are not stored in the **contiguous memory Locations.**
- Hence the access time for Individual node is **O(n)** while in Array it is O(1).

4. Reverse Traversing is difficult

- In case of singly linked list, it is **not possible to traverse linked list from end to beginning.**
- With the help of doubly linked list it becomes possible but **still it increases the required storage space for back pointer.**

5. Heap Space Restriction

- Whenever there is dynamically memory allocation, the memory is utilized from heap.
- Memory is allocated to Linked List at run time if and only if there is space available in heap.
- If there is not sufficient space available in heap then it will not be allocated.

1.11.5 Primitive Operations

Following are the important primitive operations which can be performed on Linked List :

- Insertion - Adds an element to the list.
- Deletion - Deletes an element from the list.
- Display - Displays the complete list.
- Search - Searches an element using the given key.
- Traversal - Visiting each node of the list.

1.12 REPRESENTATION OF LINKED LIST

A linked list can be represented in two ways :

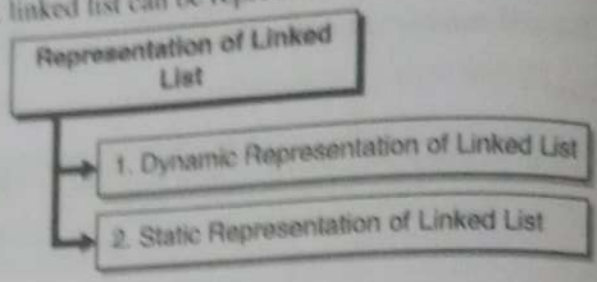


Fig.1.12.1 : Representation of Linked List

1.12.1 Dynamic Representation of Linked List

GQ. 1.12.1 *Define dynamic memory allocation.* **(2 Marks)**

Definition of Dynamic memory allocation : Dynamic memory allocation means memory can be allocated or de-allocated as per the requirement at run time.

Importance of dynamic memory allocation

- No need to initially occupy large amount of memory.
- Memory can be allocated as well as de-allocated as per necessity.
- It avoids the memory shortage as well as memory wastage.

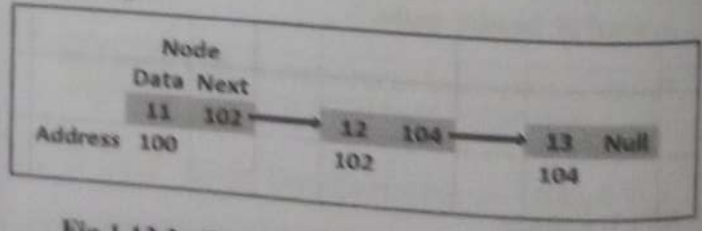


Fig.1.12.2 : Representation of linear linked list

- In Fig. 1.12.2 we can observe that a linked list is made up of number of nodes. Every node contains two parts : data and next.
- The first node contains 11 in data part as value and 102 in next part which is address of next node.

- Every node is connected with next node with the help of address.
- The next part of last node has null value as there is no further node.

1.12.2 Static Representation of Linked List

- Let LIST is linear linked list. It needs two linear arrays for memory representation. Let these linear arrays are INFO and LINK.
- INFO[K] contains the information part and LINK[K] contains the next pointer field of node K.

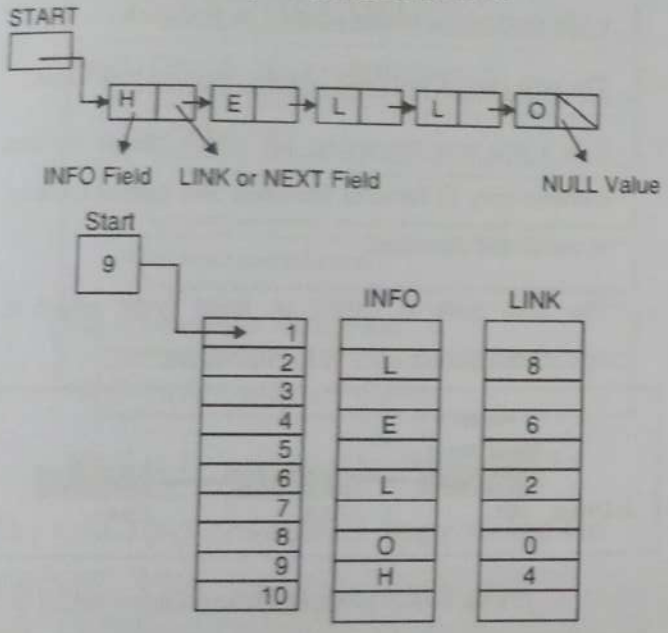


Fig. 1.12.3

- A variable START is used to store the location of the beginning of the LIST and NULL is used as next pointer sentinel which indicates the end of LIST. It is shown in Fig. 1.12.3.

Here,

START = 9 ⇒ INFO[9] = H is the first character

LINK[9] = 4 ⇒ INFO[4] = E is the second character

LINK[4] = 6 ⇒ INFO[6] = L is the third character

LINK[6] = 2 ⇒ INFO[2] = L is the fourth character

LINK[2] = 8 ⇒ INFO[8] = O is the fifth character

LINK[8] = 0 ⇒ The NULL value, so the LIST ends here.

1.13 LINKED LIST AS ADT

UQ. 1.13.1 *Explain with suitable example : Linked list as an ADT.*

SPPU - Dec. 16, May 19, 4 Marks

- **Abstract data types**, commonly abbreviated ADTs, are a way of classifying data structures based on how they are used and the behaviors they provide. They do not specify how the data structure must be implemented but simply provide a minimal expected interface and set of behaviors.
- **Data Structure** is a concrete implementation of a data type. It's possible to analyze the time and memory complexity of a Data Structure but not from a data type. The Data Structure can be implemented in several ways and its implementation may vary from language to language.

Linked List - Abstract Data Type

- Linked List is an *Abstract Data Type (ADT)* that holds a collection of **Nodes.** The nodes can be accessed in a sequential way. **Linked List doesn't provide a random access to a Node.**
- Usually, those Nodes are connected to the next node and/or with the previous one, this gives the linked effect. When the Nodes are connected with only the next pointer the list is called Singly Linked List and when it's connected by the next and previous the list is called Doubly Linked List.
- The Nodes stored in a Linked List can be anything from primitives types such as integers to more complex types like instances of classes.

ADT Interface

- The Linked List interface can be implemented in different ways which is important to have operations to **insert** a new node and to **remove** a Node.

Another ADT Operations

- **Create :** Creation of linked list.
- **Traversal :** A Linked List can be traversed, is possible to navigate in the list using the nodes next element.

- **Search :** Data of Linked list can be compared with data to be searched in linked manner.
- **Destroy :** Free the memory.

1.14 HEAD POINTER AND HEADER NODE

GQ. 1.14.1 *Explain the concept of header node.* **(2 Marks)**

- A header linked list is a linked list which always contains a special node called the *header node* at the beginning of the list.
- It is an extra node kept at the front of a list. Such a node does not represent an item in the list. The information portion might be unused.
- More often, the information portion of such a node could be used to keep global information about the entire list such as :
  - Number of nodes (not including the header) in the list.
  - Pointer to the current node in the list.

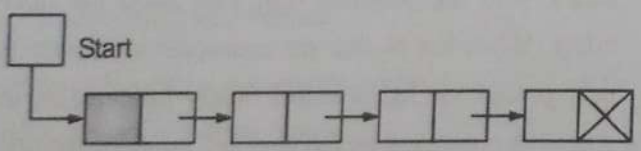


Fig. 1.14.1 : Header Node

1.15 TYPES OF LINKED LIST

There are three types of linked lists :

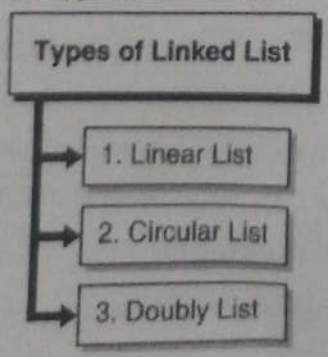


Fig. 1.15.1 : Types of Linked List

Syllabus Topic : Singly Linked List

1.16 LINEAR / SINGLY LINKED LIST

UQ. 1.16.1 *Write short note on SLL.* **SPPU - Dec. 17, 4 Marks**

- It is also called as singly linked list. It is the basic form of linked list.
- In this linked list, every node is made up of two parts : data and next. Data part contains the actual element while next part contains address of next node.
- The next part of last node always contains null value.
- It is a one way traversing list, which means we can traverse only in forward direction. We cannot traverse in backward direction.
- The first node is called as **head** node which is considered as reference to the linked list.

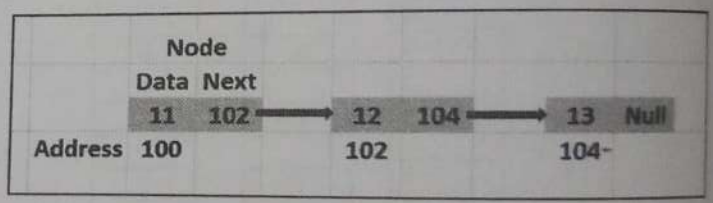


Fig. 1.16.1 : Singly Linked List

Syllabus Topic : Operations : Create, Display, Search, Insert, Delete

1.16.1 Operations : Create, Display, Search, Insert, Delete (Singly Linked List)

GQ. 1.16.2 *List operations on linked list.* **(2 Marks)**

- There are various types of operations which can be performed on singly linked list.

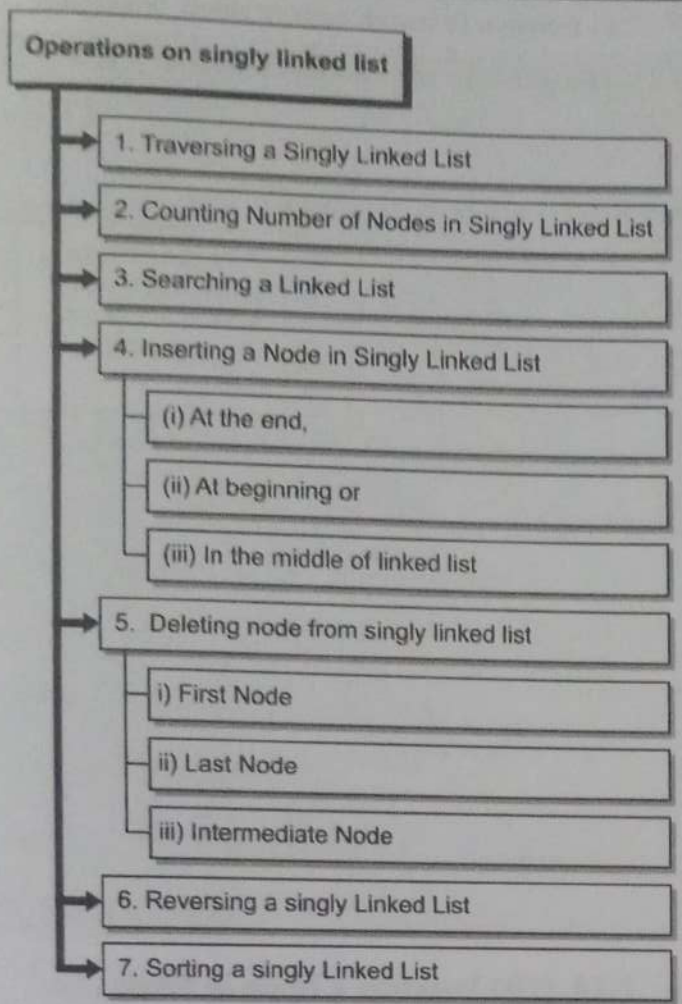


Fig. 1.16.2 : Operations on singly linked list

1.16.1(A) Traversing a Singly Linked List

GQ. 1.16.3 *Write an algorithm to traverse a singly linked list.* **(2 Marks)**

Traversing means the process of visiting each node at least once.

☞ **Algorithm to traverse singly linked list Steps**

Step 1 : If head = NULL
print ('List is empty')
else
Step 2 : q = head
Step 3 : print (q->data)
Step 4 : go to next node
Step 5 : if q != NULL
Repeat from step 3
Step 6 : End

☞ **C++ function to traverse and display data**

GQ. 1.16.4 *Write C++ function to traverse and display data.* **(2 Marks)**

```
void display ( struct node *q )
{
/* address of head is passed to q */
   while ( q != NULL )
   {
       cout<<" "<< q -> data ;
       q = q -> next ;
   }
}
```

From beginning data part of every node is printed and pointer shifts to next node

1.16.1(B) Counting Number of Nodes in Singly Linked List

☞ **Algorithm to count number of nodes in singly linked list**

GQ. 1.16.5 *Write an algorithm to count number of nodes in singly linked list.* **(3 Marks)**

▶ **Steps**

Step 1 : If *head* is NULL then
print ('List is empty')
else
Step 2 : q = head AND counter= 0
Step 3 : counter = counter +1
Step 4 : go to next node
Step 5 : if *q* != NULL
Repeat from step 3
Step 6 : return *counter*

☞ **C++ function to count number of nodes in Singly Linked List**

GQ. 1.16.6 *Write C++ function to count number of nodes in singly linked list.* **(4 Marks)**

```
int count ( struct node *q )
{
int count = 0;
/* address of head is passed to q */
    while ( q != NULL )
    {
        count++;
        q = q -> next ;
    }
    return(count)
}
```

Counter is incremented for every node while traversing

1.16.1(C) Searching a Linked List

GQ. 1.16.7 *Write an algorithm for searching a node in linked list..* **(4 Marks)**

We can search for a node in the linked list. The data part can be accepted from user to search. This data part is compared with data parts of all the nodes.

Algorithm to search node in singly linked list Steps

Step 1 : Read data element to search
Step 2 : If head is NULL then
print ('List is empty')
else
Step 3 : q = head
Step 4 : if data = q->data
print (q->data)
return
Step 5 : go to next node
Step 6 : if q != NULL
Repeat from step 3

C++ function to search node in singly linked list

GQ. 1.16.8 *WAP to search an element in a link list.* **(5 Marks)**

```
void search ( struct node *q, int data )
{
    while ( q != NULL )
    {
      if(data ==q->data)
       {
    cout<<"Element found "<<data ;
       return;
       }
        q = q -> next ;
    }
    cout<<n Element not found";
}
```

Address of head is passed to q and element to search is assigned to data

1.16.1(D) Inserting a Node in Singly Linked List

A node can be inserted at any place in the linked list :

(i) At the end,

(ii) At beginning or

(iii) In the middle of linked list

(i) Adding at the end of singly Linked List (append)

Algorithm to add node at the end of singly linked list

GQ. 1.16.9 *Write an algorithm to insert new node at the end of a single linked list.* **(7 Marks)**

▶ **Steps**

Step 1 : q = head

Step 2 : Read data

Step 3 : alloc (*newNode*)

newNode→data = data

newNode→next = NULL

Step 4 : If *q* is NULL then

q = newNode and return

Step 5 : *temp* = *q*

Step 6 : go to next node

Step 7 : if temp->next is not NULL then

Repeat Step 6

Step 8 : Set address of newnode in next of temp

☞ **Representation**

- When first node is added (i.e. when linked list is empty).

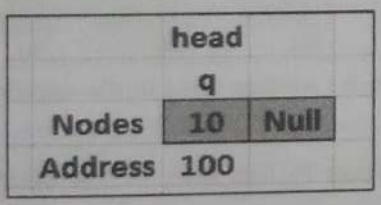


Fig. 1.16.3 : Adding First Node

- When already nodes are present :

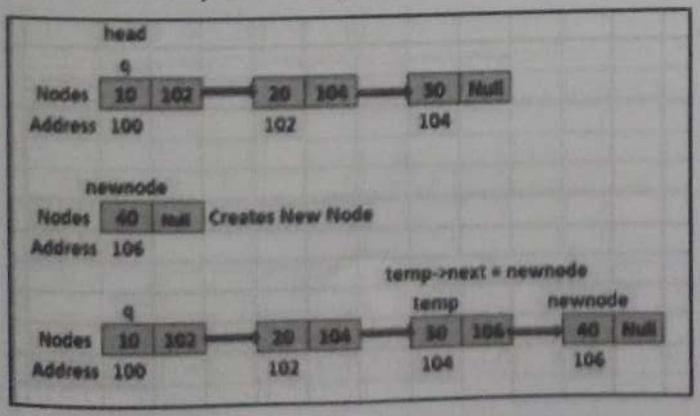


Fig. 1.16.4 : Adding node in existing list

☞ **C++ function to add node at the end of singly linked list**

UQ. 1.16.10 *Write pseudo C++ code / function to insert a node at end of singly linked list (SLL).*

SPPU - May 17, May 19, 3 Marks

```
void append ( struct node **q, int num )
{
    struct node *temp, *newnode ;
    newnode = (struct node*) malloc ( sizeof ( struct node ) )
;
    newnode -> data = num ;
    newnode -> next = NULL ;
if ( *q == NULL )
        {
        *q = newnode ;
        }
 else
    {
        temp = *q ;
        while ( temp -> next != NULL )
          temp = temp -> next;
        temp -> next = newnode ;
    }
}
```

If the list is empty, create first node

Go to last node

(ii) Adding node at the beginning of Singly Linked List

☞ **Algorithm to add node at the beginning of singly linked list**

GQ. 1.16.11 *Write an algorithm to insert new node at the beginning of a single linked list.*

(2 Marks)

Steps

Step 1 : Set q at head
Step 2 : Read data
Step 3 : alloc (newNode)
newNode→data = data
Step 4 : newNode→next = q
Step 5 : Set q at newnode

Representation

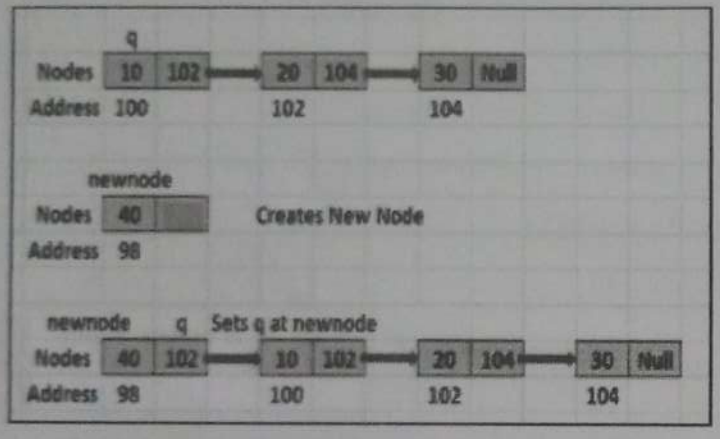


Fig. 1.16.5 : Adding node at the beginning of list

C++ function to add node at the beginning of linked list

UQ. 1.16.12 *Write pseudo C++ code to insert a node at start of singly linked list (SLL).*

SPPU - May 17, 3 Marks

```
void addatbeg ( struct node **q, int num )
{
    struct node *temp, *newnode ;
    newnode = (struct node*) malloc ( sizeof ( struct node ) )
;
    newnode -> data = num ;
    newnode -> next = *q;
    *q = newnode;
}
```

The address of previous first node is assigned to next part of new node which makes new node as first node

(iii) Adding at the middle

Algorithm to add node at the middle of singly linked list

Steps

Step 1 : temp = head;
Step 2 : read num
Step 3 : traverse temp to location
Step 4 : alloc (newNode)
Step 5 : newNode → data = num
Link newnode with next node
Link newnode with previous node

Representation

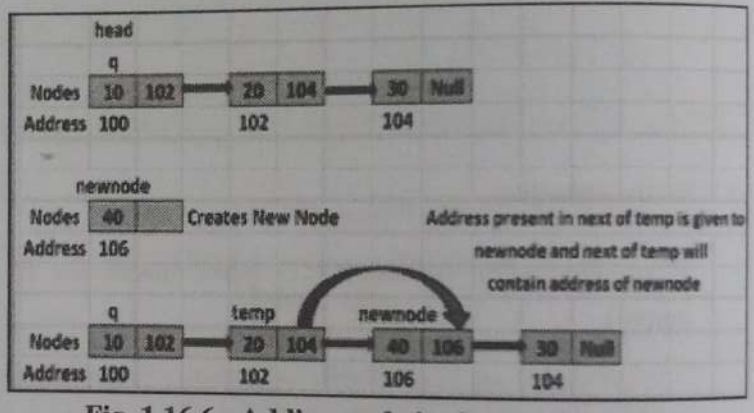


Fig. 1.16.6 : Adding node in the middle of list

C++ function to add node at the middle of singly linked list

GQ. 1.16.13 *Write C++ function to add node at the middle of singly linked list.* **(4 Marks)**

```
void addafter ( struct node *q, int num, int loc)
{
    struct node *temp, *r, *newnode ;
    int i ;
    temp = q ;
    for ( i = 0 ; i < loc-1 ; i++ )
    {
        temp = temp -> next ;
        /* if end of linked list is encountered */
```

Temp traversed up to location

```
    if ( temp == NULL )
    {
        cout<<n Location is wrong : "<< loc ;
        return ;
    }
}
/* insert new node */
newnode = (struct node*) malloc ( sizeof ( struct node ) )
;
newnode -> data = num ;
newnode -> next = temp -> next ;
temp -> next = newnode ;
}
```

Setting node at the middle by exchanging addresses

1.16.1(E) Deleting a Node from Singly Linked List

(i) Deleting first node from singly linked list

GQ. 1.16.14 *Write an algorithm to delete an element from a singly linked list.* **(7 Marks)**

Steps

Step 1 : q = head;

Step 2 : temp = head;

Step 3 : traverse q to next node

Step 4 : delete temp

Representation

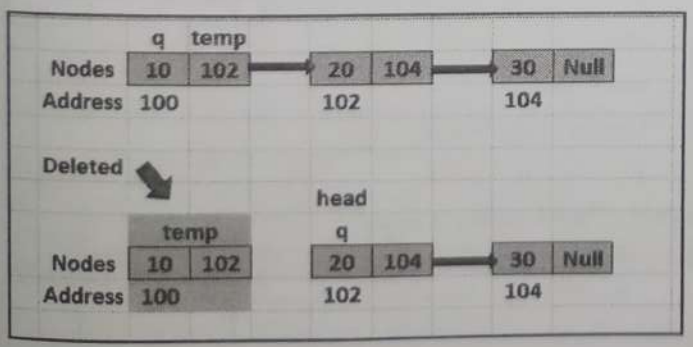


Fig. 1.16.7 : Deleting starting node

C++ function to delete node which is at the beginning of linked list

```
void deletefirst (struct node **q)
{
    struct node *temp, *r ;
    temp = *q ;
    *q = temp->next;
    free(temp);
}
```

Sets *q* pointer to second node

(ii) Deleting last node from singly linked list

Algorithm to delete node which is at the last in the singly linked list

GQ. 1.16.15 *Write an algorithm to delete last node in linked list.* **(4 Marks)**

Steps

Step 1 : q = head;

Step 2 : temp = head;

Step 3 : alloc(old)

Step 4 : old = temp;

temp = temp→next

Step 5 : if temp→next != NULL repeat step 4

Step 6 : Set NULL value in old->next

Step 7 : Delete temp

Representation

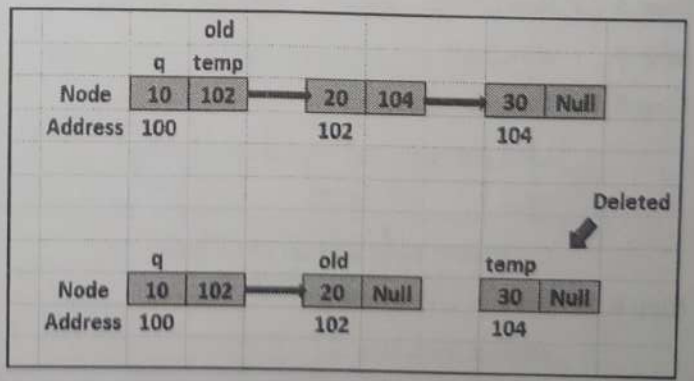


Fig. 1.16.8 : Deleting last node from SLL

☞ **C++ function to delete node which is at the last in linked list**

UQ. 1.16.16 *Write C++ function to delete node which is at the last in the singly linked list.* **SPPU - May 19, 3 Marks**

```
void deletelast (struct node **q)
{
    struct node *temp,*old;
    temp = *q;
while (temp->next != NULL)
{
      old = temp;
      temp = temp->next;
}
old -> next = temp -> next ;
free(temp);
}
```

temp is set to last node and deleted

(iii) Deleting node from middle of singly linked list

Here we accept data of node to be deleted from user.

☞ **Algorithm to delete node which is at the middle in the linked list**

GQ. 1.16.17 *Write Algorithm to delete node which is at the middle in the linked list.*

(4 Marks)

▸ **Steps**

Step 1 : q = head;
Step 2 : temp = head;
Step 3 : Read num : data of node to be deleted
Step 4 : alloc(old)
Step 5 : If temp->data = num
Link previous and next node of temp
Else
Go to next node and set old = temp
Step 6 : if temp is not NULL
Repeat from step 5
Step 7 : Delete temp

☞ **Representation**

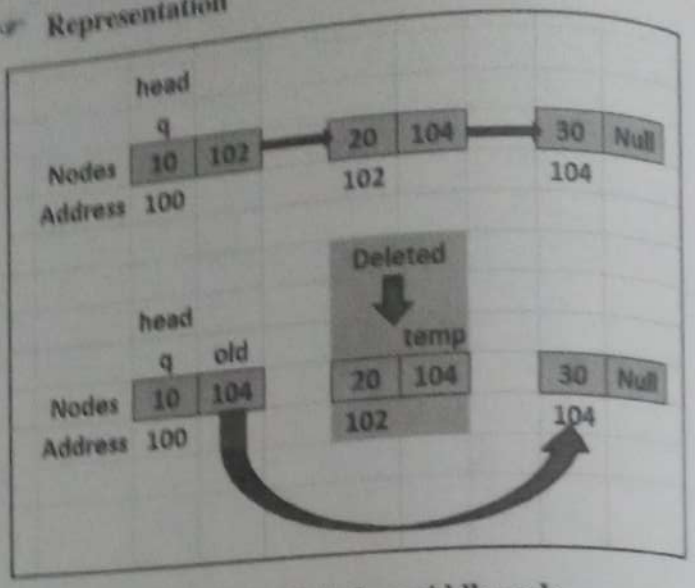


Fig. 1.16.9 : Deleting middle node

☞ **C++ function to delete node which is at the middle in singly linked list**

GQ. 1.16.18 *Write C++ code to delete intermediate node from singly linked list.*

(4 Marks)

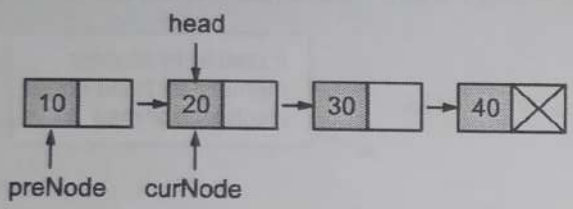
```
void deletemiddle (struct node **q, int num)
{
struct node *temp,*old;
temp = *q;
while (temp->next != NULL)
{
    if(temp->data == num)
    {
        old -> next = temp -> next;
        free(temp);
        return;
    }
    else
    old = temp;
    temp = temp->next;
    }
cout<<"\n Element not found";
}
```

When node found, its previous and next nodes are linked. And then the node is deleted

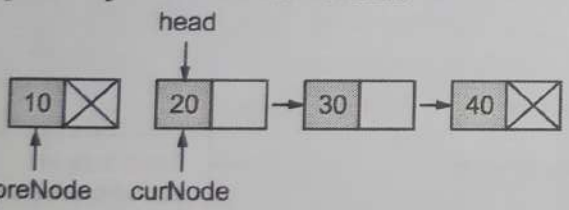
1.16.1(F) Reversing a Singly Linked List

Steps to reverse a singly Linked List

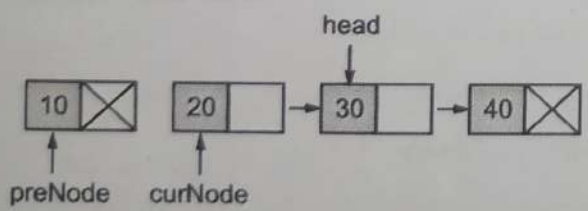
- Create two more pointers other than head namely prevNode and curNode that will hold the reference of previous node and current node respectively. Make sure that prevNode points to first node i.e. prevNode = head.
- head should now point to its next node i.e. the second node head = head->next.
- curNode should also points to the second node i.e. curNode = head.



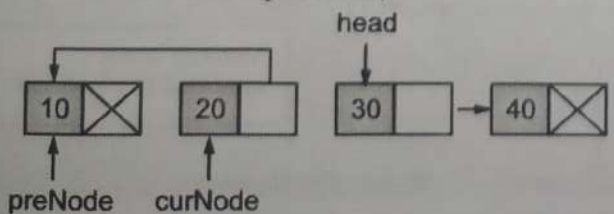
1. Now, disconnect the previous node i.e. the first node from others. We will make sure that it points to none. As this node is going to be our last node. Perform operation prevNode->next = NULL.



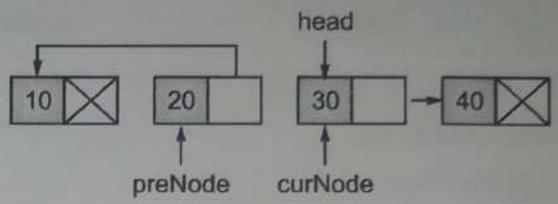
2. Move head node to its next node i.e. head = head->next.



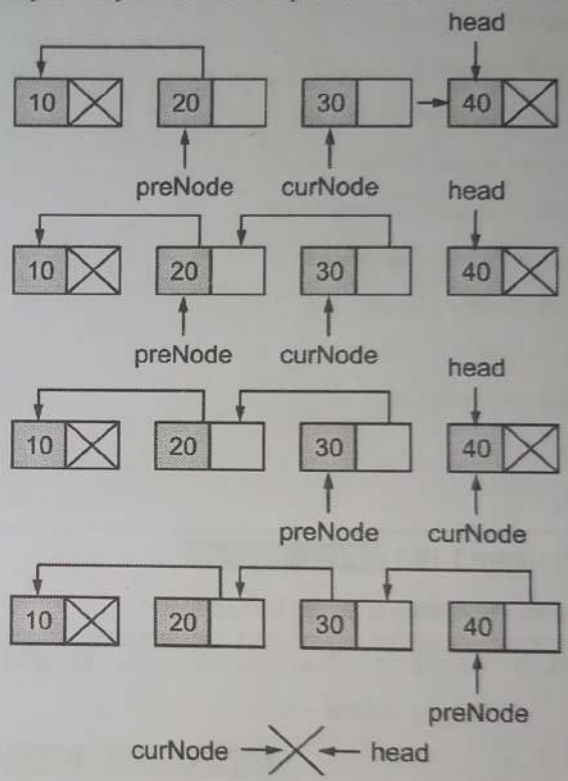
3. Now, re-connect the current node to its previous node i.e. curNode->next = prevNode;.



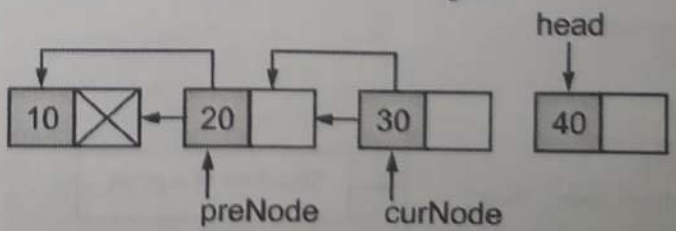
4. Point the previous node to current node and current node to head node. Means they should now point to prevNode = curNode; and curNode = head.



5. Repeat steps 3-5 till head pointer becomes NULL.



6. Now, after all nodes has been re-connected in the reverse order. Make the last node as the first node. Means the **head** pointer should point to prevNode pointer. Perform head = prevNode;. Finally you end up with a reversed linked list of its original.



Algorithm to reverse a singly linked list

GQ. 1.16.19 *Write an algorithm to perform the following operations on singly linked list : Reverse.* **(3 Marks)**

```
Begin :
  If (head != NULL) then
    prevNode ← head
    head ← head.next
    curNode ← head
    prevNode.next ← NULL
    While (head != NULL) do
      head ← head.next
      curNode.next ← prevNode
      prevNode ← curNode
      curNode ← head
    End while
    head ← prevNode
  End if
End
```

Program 1.16.1 SPPU - Dec. 17

Program for Reversing a Singly Linked List

UQ. 1.16.20 *Write a 'C++' function to reverse a singly linked list.*

SPPU - Dec. 17, 4 Marks

Soln. :

```
using namespace std;
#include<iostream>
#include <stdlib.h>
struct node1
{
  int data;
  struct node1 *next;
}*head;

class reverse_link
{
    public:
void createList(int node)
{
  struct node1 *newNode, *temp;
  int data, i;
  if(node <= 0)
  {
    cout<<"List size must be greater than zero.\n";
    return;
  }
  head = (struct node1 *)malloc(sizeof(struct node1));

  if(head == NULL)
  {
    cout<<"Unable to allocate memory.";
  }
  else
  {

    cout<<"Enter the data of node 1: ";
    cin>>data;

    head->data = data;
    head->next = NULL;
    temp = head;
    for(i=2; i<=node; i++)
    {
newNode = (struct node1 *)malloc(sizeof(struct node1));
      if(newNode == NULL)

      {
        cout<<"Unable to allocate memory.";
```

Structure of a node

If unable to allocate memory for head node display error msg

Read data of node from the user

Memory is not allocated to newnode

```
            break;
        }
        else
        {
            cout<<"Enter the data of node "<<i<<": ";
            cin>>data;
            newNode->data = data;
            newNode->next = NULL;
            temp->next = newNode;
            temp = temp->next;
        }
    }
  }
}
void reverseList()
{
  struct node1 *prevNode, *curNode;
  if(head != NULL)
  {
    prevNode = head;
    curNode = head->next;
    head = head->next;
    prevNode->next = NULL;
    while(head != NULL)
    {
      head = head->next;
      curNode->next = prevNode;
      prevNode = curNode;
      curNode = head;
    }
    head = prevNode; // Make last node as head
  }
}
```

To make first node as last node

```
void displayList()
{
  struct node1 *temp;
  if(head == NULL)
  {
    cout<<"List is empty.";
  }
  else
  {
    temp = head;
    while(temp != NULL)
    {
      cout<<" "<< temp->data;
      temp = temp->next;
    }
  }
}

};

int main()
{
  int node, choice;
  reverse_link r;
  cout<<"Enter the total number of nodes: ";
  cin>>node;
  r.createList(node);
  cout<<"\nData in the list\n";
  r.displayList();
  cout<<"\n\nReversed singly linked list\n";
  r.reverseList();
  r.displayList();

}
```

Output

```
Enter the total number of nodes: 5
Enter the data of node 1: 11
Enter the data of node 2: 12
Enter the data of node 3: 13
Enter the data of node 4: 14
Enter the data of node 5: 15

Data in the list
 11  12  13  14  15

Reversed singly linked list
 15  14  13  12  11
```

1.16.1(G) Sorting a Singly Linked List

Algorithm to sort singly linked list

GQ. 1.16.21 *Write an algorithm to perform the following operations on singly linked list : Sort.* **(3 Marks)**

```
do
   interchangeped ← 0;
   ptr1 ← start;
   while (ptr1->next ! ← lptr)
   {
      if (ptr1->data > ptr1->next->data)
      {
         interchange(ptr1, ptr1->next);
         interchangeped ← 1;
      }
      ptr1 ← ptr1->next;
   }
   lptr ← ptr1;
  } while (interchangeped = true);
}
End
```

Program to sort singly linked list

```
using namespace std;
#include<iostream>
#include <malloc.h>

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

class sortlink
{
    public:
        /* Function to add node at the beginning of a linked list */
void addNode(struct Node **start_ref, int data)
{
  struct Node *ptr1 = (struct Node*)malloc(sizeof(struct Node));
  ptr1->data = data;
  ptr1->next = *start_ref;
  *start_ref = ptr1;
}
```

Function to display nodes in a given linked list

```
void display(struct Node *start)
{
  struct Node *temp = start;
  cout<<"\n";
  while (temp!=NULL)
  {
    cout<<" "<<temp->data;
    temp = temp->next;
  }
}
```

Bubble sort the given linked list

```
void bubble_Sort(struct Node *start)
{
  int interchangeped, i;
```

```
    struct Node *ptr1;
    struct Node *lptr = NULL;

    if (start == NULL)
      return;

    do
    {
      interchangeped = 0;
      ptr1 = start;

      while (ptr1->next != lptr)
      {
        if (ptr1->data > ptr1->next->data)
        {
          interchange(ptr1, ptr1->next);
          interchangeped = 1;
        }
        ptr1 = ptr1->next;
      }
      lptr = ptr1;
    }
    while (interchangeped);
}
```

Checking for empty list

Function to interchange data of two nodes a and b

```
void interchange(struct Node *a, struct Node *b)
{
    int temp = a->data;
    a->data = b->data;
    b->data = temp;
}
};
int main()
{
    int arr[] = {43, 14, 56, 67, 31}, i;

    struct Node *start = NULL;
    sortlink sl;
    for (i = 0; i<5 ; i++)
      sl.addNode(&start, arr[i]);

    cout<<"\n Before sorting ";
    sl.display(start);

    sl.bubble_Sort(start);

    cout<<"\n After sorting ";
    sl.display(start);

    getchar();
    return 0;
}
```

Start with empty linked list

Display list before sorting

Sort the linked list

Display list after sorting

Output

```
 Before sorting
31 67 56 14 43
 After sorting
14 31 43 56 67
```

1.16.1(H) Menu Driven Program on All Operations of Singly Linked List

GQ. 1.16.22 *Write Pseudo C/C++ code to represent singly linked list an ADT.* **(6 Marks)**

```
using namespace std;
#include<iostream>
#include <malloc.h>
struct node
{
    int data ;
    struct node *next ;
} ;
class singly
```

Structure containing a data part and link part

```
{
public:
    void append ( struct node **q, int num )
{
    struct node *temp, *newnode ;
newnode = (struct node*) malloc ( sizeof ( struct node ) ) ;

    newnode -> data = num ;
        newnode -> next = NULL ;
if ( *q == NULL )
    {
        *q = newnode ;
        }
 else
    {
        temp = *q ;
        /* go to last node */
        while ( temp -> next != NULL )
            temp = temp -> next;
        temp -> next = newnode ;
    }
}
void display ( struct node *q )
{
    while ( q != NULL )
    {
        cout<<" "<<q->data ;
        q = q -> next ;
    }
}
void search ( struct node *q, int data )
{
    while ( q != NULL )
    {
     if(data ==q->data)
        {
        cout<<"Element found "<<data ;
        return;
        }
         q = q -> next ;
    }
    cout<<"\n Element not found";
}
void addatbeg ( struct node **q, int num )
{
    struct node *temp, *newnode ;
newnode = (struct node*) malloc ( sizeof ( struct node ) ) ;
    newnode -> data = num ;
    newnode -> next = *q;
*q = newnode;
}

void addafter ( struct node *q, int num, int loc )
{
    struct node *temp, *r, *newnode ;
    int i ;
    temp = q ;
    for ( i = 0 ; i < loc-1 ; i++ )
    {
        temp = temp -> next ;
        if ( temp == NULL )
        {
            cout<<"\n Location is wrong "<<loc ;
            return ;
        }
    }
    /* insert new node */
```

If the list is empty, create first node

Go to last node

From beginning data part of every node is printed and pointer shifts to next node

Address of head is passed to q and element to search is assigned to data

The address of previous first node is assigned to next part of new node which makes new node as first node

Temp traversed up to location

```
    newnode = (struct node*) malloc ( sizeof ( struct node ) )
;
    newnode -> data = num ;
    newnode -> next = temp -> next ;
    temp -> next = newnode ;
}
void deletefirst (struct node **q)
{
    struct node *temp, *r ;
    temp = *q ;
    *q = temp->next;
    free(temp);
}
void deletelast (struct node **q)
{
    struct node *temp,*old;
    temp = *q;
while (temp->next != NULL)
{
      old = temp;
      temp = temp->next;
}
old -> next = temp -> next ;
free(temp);
}
};
main()
{
struct node *head ;
singly sl;
int ch=0,ele,loc;
    head = NULL ;
```

Setting node at the middle by exchanging addresses

Sets q pointer to second node

Temp is set to last node and deleted

Empty linked list

```
while(ch!=8)
{
cout<<"\n 1 : Append";
cout<<"\n 2 : Display";
cout<<"\n 3 : Search";
cout<<"\n 4 : Addafter";
cout<<"\n 5 : Addatbeg";
cout<<"\n 6 : DeleteFirst";
cout<<"\n 7 : DeleteLast";
cout<<"\n 8 : Exit";
cout<<"\n Select your choice : ";
cin>>ch;
switch(ch)
{
case 1:
cout<<"\n Enter element to append : ";
cin>>ele;
sl.append(&head,ele);
break;
case 2:
sl.display(head);
break;

case 3:
cout<<"\n Enter element to search : ";
cin>>ele;
sl.search(head,ele);
break;

case 4:
cout<<"\n Enter element and location : ";
cin>>ele>>loc;
sl.addafter(head,ele,loc);
break;
```

As per user's choice operation will be performed

```
case 5:
cout<<"\n Enter element to add at beginning : ";
cin>>ele;
sl.addatbeg(&head,ele);
break;

case 6:
sl.deletefirst(&head);
break;

case 7:
sl.deletelast(&head);
break;

default:
cout<<"\n Invalid choice";
}
}
}
```

Output

```
C:\DS3.exe

1 : Append
2 : Display
3 : Search
4 : Addafter
5 : Addatbeg
6 : DeleteFirst
7 : DeleteLast
8 : DeleteMiddle
9 : Exit
Select your choice : 1

Enter element to append : 11

1 : Append
2 : Display
3 : Search
4 : Addafter
5 : Addatbeg
6 : DeleteFirst
7 : DeleteLast
8 : DeleteMiddle
9 : Exit
Select your choice :
```

Syllabus Topic : Doubly Linked List

1.17 DOUBLY LINKED LIST

UQ. 1.17.1 *Write note on DLL.* **SPPU - Dec. 17, 3 Marks**

UQ. 1.17.2 *Explain with example Doubly Linked List.* **SPPU - Dec. 18, May 19, 3 Marks**

- In singular linked list, we have seen that traversing is allowed only in one direction i.e. forward direction. We cannot traverse in backward direction.
- Hence from any node if there is need to go to previous node, then it is not possible.
- DS solve this problem by providing the concept of doubly linked list.

❑ *Definition of Doubly linked list : Doubly linked list is the list in which every node contains three parts : data, previous and next. Data contains the actual element while the previous and next parts contain the addresses of previous and next nodes.*

☞ **Representation of doubly linked list**

GQ. 1.17.3 *A doubly linked list with numbers to be created. Write node structure and algorithm to create the list.* **(2 Marks)**

- As every node contains address of previous node along with next node, it is possible to traverse in both the directions. Forward as well as backward.

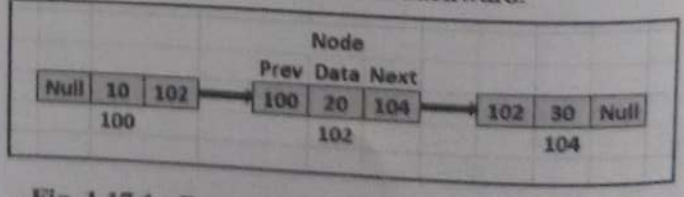


Fig. 1.17.1 : Representation of doubly Linked List

- The previous part of first node and next part of last node always contains NULL value.

1.17.1 Advantages of Doubly Linked List over Singly List

GQ. 1.17.4 *What are the advantages of doubly linked list ?* **(3 Marks)**

1. Doubly Linked List provides two pointers : next and previous which contain addresses of next and previous nodes.
2. A Doubly Linked List can be traversed in both forward as well as backward direction.
3. The delete operation in Doubly Linked List is considered as more efficient if pointer to the node to be deleted is given.

1.17.2 Operations of Doubly Linked List

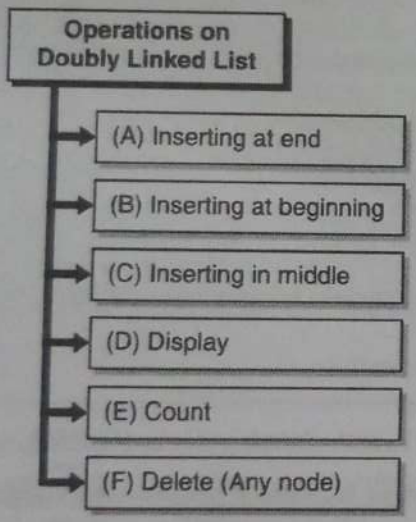


Fig. 1.17.2 : Operations on doubly linked list

(A) Inserting node at the end of DLL

☞ **Algorithm to add node at the end of DLL**

▸ **Steps**

Step 1 : q = head
Step 2 : Read data
Step 3 : alloc (newNode)
newNode→data = data
newNode→prev = NULL
newNode→next = NULL
Step 4 : If q is NULL then
q = newNode and return
Step 5 : temp = q
Step 6 : go to next node
Step 7 : if temp->next is not NULL then
Repeat Step 6
Step 8 : alloc(r)
r->data = data
Step 9 : Set address of r in next of temp
Step 10 : Set address of temp in prev of r

☞ **Representation**

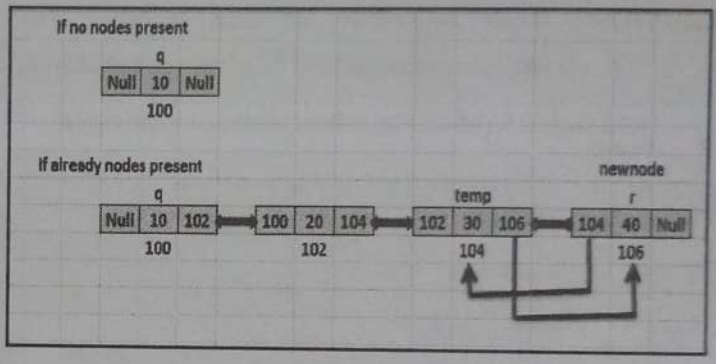


Fig. 1.17.3 : Adding node at the end of DLL

☞ **Function to add node at the end of DLL**

UQ. 1.17.5 *Write C++ function to add node at the end of DLL.* **SPPU - Dec. 16, 4 Marks**

```
void append ( struct node **q, int num )
{
    struct node *r, *temp = *q ;
    if ( *q == NULL )        <-- Creating first node
    {
        *q = malloc ( sizeof ( struct node ) ) ;
        ( *q ) -> prev = NULL ;
        ( *q ) -> data = num ;
        ( *q ) -> next = NULL ;
    }
    else
    {
```

```
    while ( temp -> next != NULL )
        temp = temp -> next ;
    r = malloc ( sizeof ( struct node ) ) ;
    r -> data = num ;
    r -> next = NULL ;
    r -> prev = temp ;
    temp -> next = r ;
  }
}
```

temp is set to last node. New node r is created and its address is stored in temp->next

(B) Inserting node at the beginning of DLL

☞ **Algorithm to add node at the beginning of DLL**

GQ. 1.17.6 *Write algorithm to add node at the beginning of DLL.* **(4 Marks)**

▶ **Steps**

Step 1 : q = head

Step 2 : Read data

Step 3 : alloc (*temp*)

Step 4 : Link *temp*'s previous and next nodes to *temp*.

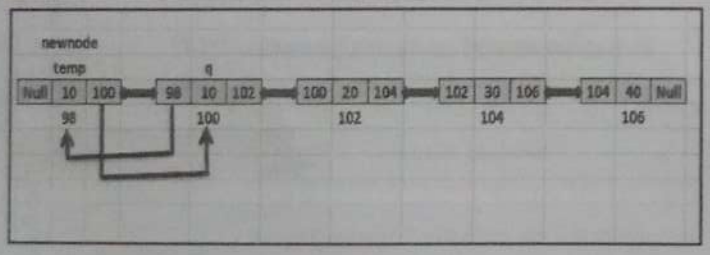


Fig. 1.17.4 : Adding node at the beginning of DLL

☞ **Function to add node at the beginning of DLL**

GQ. 1.17.7 *Write C++ function to add node at the beginning of DLL.* **(4 Marks)**

```
tbeg ( svoid addatruct node **q, int num )
{
    struct node *temp ;
    temp = malloc ( sizeof ( struct node ) ) ;
    temp -> prev = NULL ;
    temp -> data = num ;
    temp -> next = *q ;
    ( *q ) -> prev = temp ;
    *q = temp ;
}
```

(C) Inserting node at the middle of DLL

☞ **Algorithm to add node in the middle of DLL**

GQ. 1.17.8 *Write Algorithm to add node in the middle of DLL* **(3 Marks)**

▶ **Steps**

Step 1 : q = head

Step 2 : Read element and location

Step 3 : Set q to node after which new node has to be added

Step 4 : alloc (temp)

Step 5 : temp →data = ele

set address of q in temp->prev

temp -> next = q -> next ;

temp -> next -> prev = temp ;

Step 6 : Set address of temp in next of q

☞ **Representation**

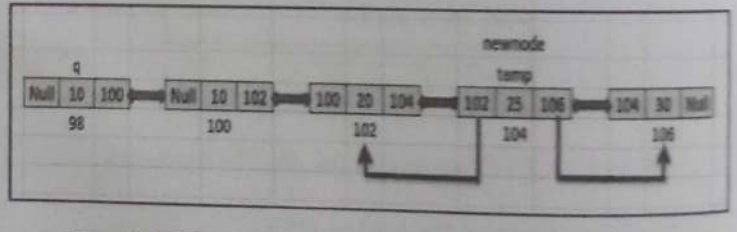


Fig. 1.17.5 : Adding node in the middle of DLL

☞ **Function to add node in the middle of DLL**

GQ. 1.17.9 *Write C++ function to add node in the middle of DLL* **(3 Marks)**

```
void addafter ( struct node *q, int ele, int loc )
{
    struct node *temp ;
    int i ;
    for ( i = 0 ; i < loc ; i++ )
```

```
    {
        q = q -> next ;
        if ( q == NULL )
        {
            cout<<"\nInvalid location "<<loc ;
            return ;
        }
    }
    q = q -> prev ;
    temp = (struct node*) malloc ( sizeof ( struct node ) ) ;
    temp -> data = ele ;
    temp -> prev = q ;
    temp -> next = q -> next ;
    temp -> next -> prev = temp ;
    q -> next = temp ;
}
void display ( struct node *q )
{
    while ( q != NULL )
    {
        cout<< q -> data <<"->" ;
        q = q -> next ;
    }
}
```

Set q to node after which new node has to be added

Sets new node temp. Temp's previous and next nodes are linked to temp.

(D) Traversing and Displaying doubly linked list

☞ **Algorithm to traverse and display doubly linked list**

GQ. 1.17.10 *Write an algorithm to implement following operations in DLL : Traversal.* **(2 Marks)**

▶ **Steps**

Step 1 : If *head* is NULL then
print ('List is empty')
else

Step 2 : *q = head*

Step 3 : print (*q->data*)

Step 4 : go to next node

Step 5 : if *q* != NULL
Repeat from step 3

☞ **Function to display doubly linked list**

GQ. 1.17.11 *Write 'C++' functions to implement TRAVERSE operation in doubly linked list.* **(2 Marks)**

(E) Counting nodes of Doubly Linked List

☞ **Algorithm to count nodes of Doubly Linked List**

GQ. 1.17.12 *Write Algorithm to count nodes of Doubly Linked List.* **(4 Marks)**

▶ **Steps**

Step 1 : If *head* is NULL then
print ('List is empty')
else

Step 2 : q = head AND counter= 0

Step 3 : counter = counter +1

Step 4 : go to next node

Step 5 : if *q* != NULL
Repeat from step 3

Step 6 : return *counter*

☞ **Function to count nodes of Doubly Linked List**

GQ. 1.17.13 *Write a C++ function to find maximum element from doubly linked list.* **(2 Marks)**

```
int count ( struct node * q )
{
    int cnt = 0 ;
    while ( q != NULL )
    {
        q = q -> next ;
        cnt++ ;
    }
    return cnt ;
}
```

(F) Searching and Deleting node from Doubly Linked List

GQ. 1.17.14 *Explain delete operation of doubly linked list.* **(7 Marks)**

☞ **Algorithm to delete node from Doubly Linked List**

UQ. 1.17.15 *Write algorithm to delete node from Doubly Linked List.*

SPPU- Dec. 16, 4 Marks

Steps

Step 1 : q = temp = head

Step 2 : Read num

Step 3 : Traverse temp

Step 4 : Compare temp->data with num

Step 5 : If data found :

If it is first node, set q to next node

Set NULL value in q->prev

If it is last node, Set NULL value in next part of temp->prev node

If it is intermediated node

Link the previous and next node of temp

Step 6 : Delete temp

Step 7 : If data not found and temp!= NULL

Repeat from Step 3

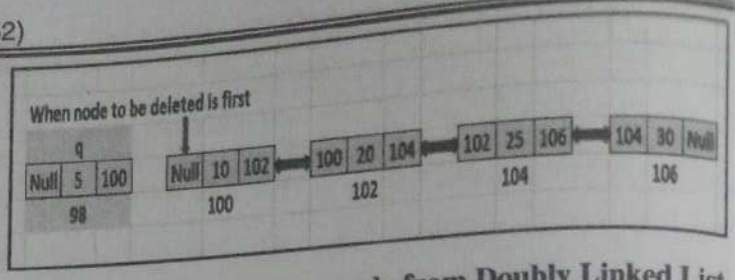


Fig. 1.17.6 : Deleting first node from Doubly Linked List

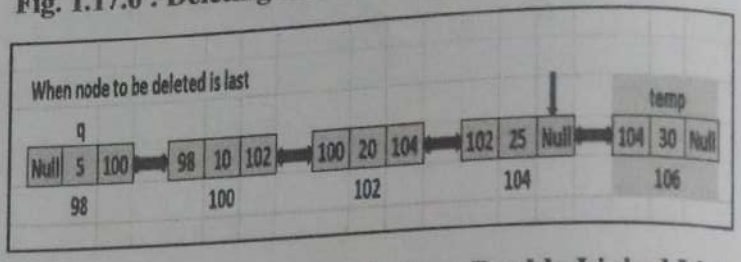


Fig. 1.17.7: Deleting last node from Doubly Linked List

☞ **Algorithm to delete intermediate node from Doubly Linked List**

GQ. 1.17.16 *Write an algorithm to delete intermediate node from Doubly Linked List.* **(3 Marks)**

Steps

Step 1 : q = temp = head

Step 2 : Read num

Step 3 : Traverse temp

Step 4 : Compare temp->data with num

Step 5 : If data found :

If it is intermediated node

Link the previous and next node of temp

Step 6 : Delete temp

Step 7 : If data not found and temp!= NULL

Repeat from Step 3

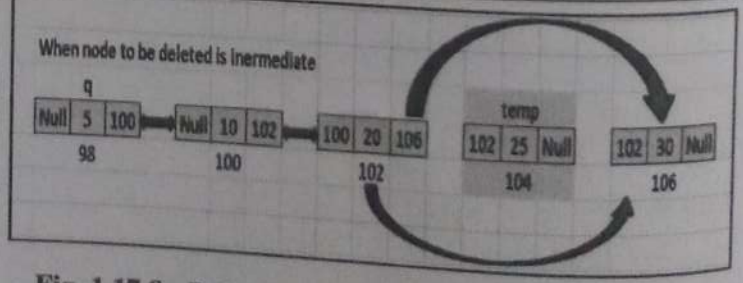


Fig. 1.17.8 : Deleting intermediate node from Doubly Linked List

Function to delete node from Doubly Linked List

UQ. 1.17.17 *Write pseudo code to delete a node from doubly linked list (DLL)*

SPPU - May 17, 6 Marks

```
void delete ( struct node **q, int num )
{
    struct node *temp = *q ;
    while ( temp != NULL )
    {
        if ( temp -> data == num )
        {
if ( temp == *q )                          <- If node to be deleted is the first node
{
*q = ( *q ) -> next ;
( *q ) -> prev = NULL ;
}
else
{
    if ( temp -> next == NULL )            <- If node to be deleted is the last node
    temp -> prev -> next = NULL ;
    else                                   <- If node to be deleted is intermediate node
    {
        temp -> prev -> next = temp -> next ;
        temp -> next -> prev = temp -> prev ;
    }
    free ( temp ) ;
    cout<<"\n Node deleted "<<num;
}
        return ;
        }
        temp = temp -> next ;
    }
    cout<<num<<" not found." ;
}
```

Program to perform various operations on doubly linked list

GQ. 1.17.18 *Write pseudo C/C++ code to represent doubly linked list as an ADT.*

(6 Marks)

```
using namespace std;
#include<iostream>
#include <malloc.h>

struct node
{
    struct node *prev ;
    int data ;
    struct node * next ;
} ;
class dbly
{
    public:
void append ( struct node **q, int num )
{
    struct node *r, *temp = *q ;
    if ( *q == NULL )                      <- Creating first node
    {
        *q = (struct node*) malloc ( sizeof ( struct node ) ) ;
        ( *q ) -> prev = NULL ;
        ( *q ) -> data = num ;
        ( *q ) -> next = NULL ;
    }
    else
    {
        while ( temp -> next != NULL )
            temp = temp -> next ;
        r = (struct node*) malloc ( sizeof ( struct node ) ) ;
        r -> data = num ;
        r -> next = NULL ;
        r -> prev = temp ;
        temp -> next = r ;
    }
}                                          <- temp is set to last node. New node r is created and its address is stored in temp->next

void addatbeg ( struct node **q, int num )
```

```
{
    struct node *temp ;
    temp = (struct node*) malloc ( sizeof ( struct node ) ) ;
    temp -> prev = NULL ;
    temp -> data = num ;

    temp -> next = *q ;
    ( *q ) -> prev = temp ;
    *q = temp ;
}
```

Address of previous first node q is stored in new node temp->next and q is set to temp to make it first node

```
void addafter ( struct node *q, int ele, int loc )
{
    struct node *temp ;
    int i ;
    for ( i = 0 ; i < loc ; i++ )
    {
        q = q -> next ;
        if ( q == NULL )
        {
            cout<<"\nInvalid location "<<loc ;
            return ;
        }
    }
    q = q -> prev ;
    temp = (struct node*) malloc ( sizeof ( struct node ) ) ;
    temp -> data = ele ;
    temp -> prev = q ;
    temp -> next = q -> next ;
    temp -> next -> prev = temp ;
    q -> next = temp ;
}
```

Set q to node after which new node has to be added

Sets new node temp. Temp's previous and next nodes are linked to temp.

```
void display ( struct node *q )
{
    while ( q != NULL )
    {
        cout<< q -> data <<"->" ;
        q = q -> next ;
    }
}
int count ( struct node * q )
{
    int cnt = 0 ;
    while ( q != NULL )
    {
        q = q -> next ;
        cnt++ ;
    }
    return cnt ;
}
void deletenode ( struct node **q, int num )
{
    struct node *temp = *q ;
    while ( temp != NULL )
    {
        if ( temp -> data == num )
        {
            if ( temp == *q )
            {
                *q = ( *q ) -> next ;
                ( *q ) -> prev = NULL ;
            }
            else
            {
                if ( temp -> next == NULL )
                    temp -> prev -> next = NULL ;
                else
                {
                    temp -> prev -> next = temp -> next ;
                    temp -> next -> prev = temp -> prev ;
                }
                free ( temp ) ;
```

if node to be deleted is the first node

If node to be deleted is the last node

If node to be deleted is intermediate node

```
        cout<<"\n Node deleted "<<num;
            }
            return ;
        }
        temp = temp -> next ;
    }
    cout<<num<<" not found.";
}
};
int main( )
{
    struct node *head ;
    int ch;
    int ele,loc;
    head = NULL ;
    dbly d;
while(ch!=7)
    {
    cout<<"\n\n 1 : Append";
    cout<<"\n 2 : Addatbeg";
    cout<<"\n 3 : Addafter";
    cout<<"\n 4 : Display";
    cout<<"\n 5 : Count";
    cout<<"\n 6 : Delete";
    cout<<"\n 7 : Exit";
    cout<<"\n Select your choice : ";
    cin>>ch;
    switch(ch)
    {
    case 1:
    cout<<"\n Enter element to append : ";
    cin>>ele;
    d.append(&head,ele);
    break;
    case 2:
    cout<<"\n Enter element to add at beginning : ";
    cin>>ele;
    d.addatbeg(&head,ele);
    break;
    case 3:
    cout<<"\n Enter element and location : ";
    cin>>ele>>loc;
    d.addafter(head,ele,loc);
    break;
    case 4:
    d.display(head);
    break;
    case 5:
    cout<<"\nNo. of elements in the DLL = "<<d.count
(head) ;
    break;
    case 6:
    cout<<"\n Enter element to delete : ";
    cin>>ele;
    d.deletenode(&head,ele);
    break;
    case 7:
    exit(0);
    break;
    default:
    cout<<"\n Invalid choice";
    }
    }
}
```

Empty list

As per user's choice operation will be performed

Output

```
C:\LINK4.exe

1 : Append
2 : Addatbeg
3 : Addafter
4 : Display
5 : Count
6 : Delete
7 : Exit
Select your choice : 1
```

1.17.3 Difference between Singly and Doubly Linked List

GQ. 1.17.19 *Differentiate Singly Linked List and Doubly Linked List.*

(4 Marks)

Parameter	Singly Linked List	Doubly Linked List
Node Structure	Node contains two parts: data and link to next node.	Node contains three parts : data and links to previous and next nodes.
Traversing	Only forward traversing is allowed.	Forward and backward both traversing is allowed.
Memory	It uses less memory per node (single pointer)	It uses more memory per node(two pointers).
Use	Singly linked list can mostly be used for stacks.	Doubly linked list can be used to implement stacks, heaps, binary trees.
Complexity	Complexity of Insertion and Deletion at known position is O (n).	Complexity of Insertion and Deletion at known position is O (1).

Syllabus Topic : Circular Linked List / Singly Circular Linked List

1.18 CIRCULAR LINKED LIST / SINGLY CIRCULAR LINKED LIST

UQ. 1.18.1 *Explain with suitable example : Circular linked list.*

SPPU - Dec. 16, May 19, 3 Marks

UQ. 1.18.2 *Compare Linear and circular linked list.* **SPPU - May 17, 3 Marks**

UQ. 1.18.3 *Write short notes on CLL.*

SPPU - Dec. 17, 2 Marks

- In a singly linear linked list, the traversing is allowed in forward direction only. In this list, the next part of last node contains NULL value.
- Just consider there are 100 nodes. Currently we are at node number 98 and we want to go to node number 3, then it is not possible.
- DS solves this issue by providing concept of **Circular List** in which the next part of last node contains address of first node.
- Hence we can directly jump from last node to first node.

❑ *Definition : Circular linked list is the linked list in which next part of last node contains address of first node to form a circle.*

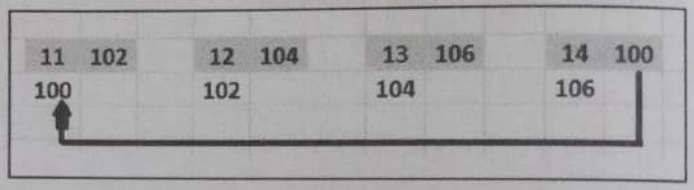


Fig. 1.18.1 : Circular Linked List

Declaration

- It is possible to declare a node in a circular linked list as any other node as shown below:

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

- For implementing the circular linked list, we maintain an external pointer "last" which points to the last node in the circular linked list.
- That means last->next will point to the first node in the linked list.
- It ensures that when a new node is inserted at the beginning or at the end of the list, there is no need to traverse the entire list. This is because the last points to the last node while last->next points to the first node.
- This wouldn't have been possible if we had pointed the external pointer to the first node.

Basic Operations

- The circular linked list supports all of the basic operations such as insertion, deletion, and traversal of the list.

A) Insertion

- We can insert a node in a circular linked list at any position; either as a first node (empty list), in the beginning, in the end, or in between other nodes.
- Here we will discuss different insertion operations using a graphical representation.

1. Insert in an empty list

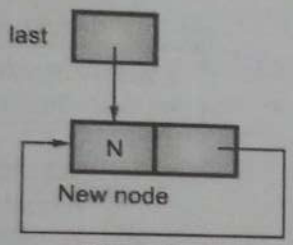


Fig. 1.18.2

- When there are no nodes in circular list, that means the list is empty and the last pointer is null, then we insert a new node N by pointing the last pointer to the node N.
- The next pointer of N will point to the node N itself as there is only one node. Thus N becomes the first as well as last node in the list.

2. Insert at the beginning of the list

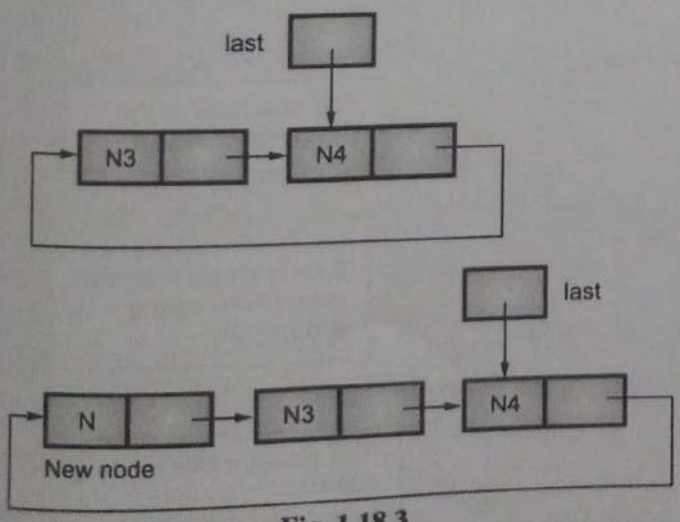


Fig. 1.18.3

- As shown in the Fig. 1.18.3, when a new node is inserted at the beginning of the list, the next pointer of the last node points to the new node N thereby making it as first node.

```
N->next = last->next
Last->next = N
```

3. Insert at the end of the list

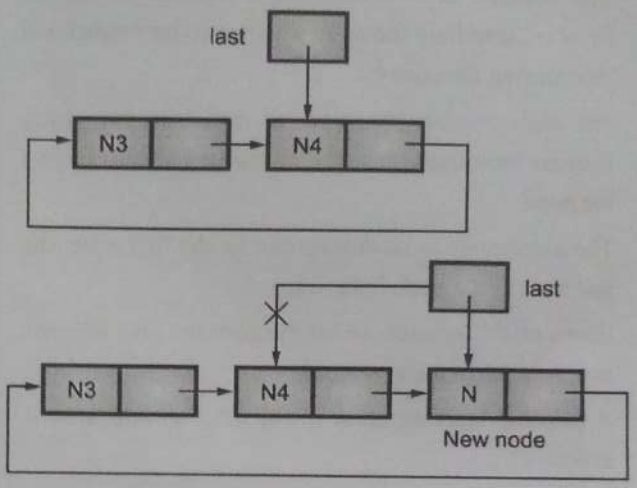


Fig. 1.18.4

- To insert a new node at the end of the list, we follow these steps:

```
N-> next = last ->next;
last ->next = N
last = N
```

4. Insert in between the list

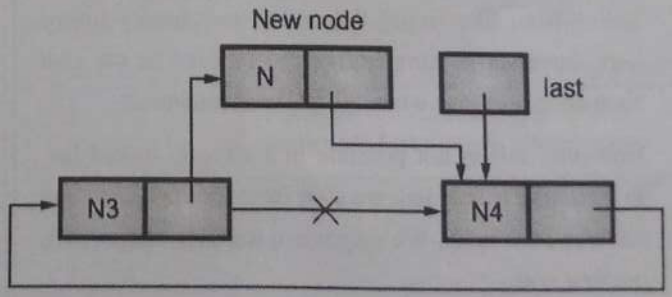


Fig. 1.18.5

- Assume that there is requirement of inserting a new node N between N3 and N4; we first required to traverse the list and locate the node after which the new node is to be inserted, in this case, its N3.

- After the node is located, we perform the following steps.

```
N -> next = N3 -> next;
N3 -> next = N
```

This inserts a new node N after N3.

B) Searching and Deletion

- The deletion of a node from a circular linked list involves searching the node which is to be deleted and then freeing its memory.
- For this purpose we need to maintain two extra pointers *curr* and *prev* and then traverse the list to find the node.
- The given node to be deleted can be the first node, the last node or the node in between.
- Based on the location we set the *curr* and *prev* pointers and then delete the *curr* node.
- A pictorial representation of the deletion operation is as follows :

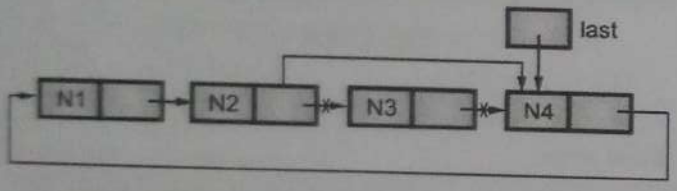


Fig. 1.18.6

C) Traversal

- Traversal is a method of visiting each node. In linear linked lists, like singly linked list and doubly linked lists, traversal is comparatively very easy as we visit each node and stop when NULL is encountered.
- However, this is not possible in a circular linked list. In a circular linked list, we start from the first node and traverse each node. We stop when we once again reach the first node.

1.18.1 Program on Circular Linked List

Performing all operations on Circular Linked List

UQ. 1.18.4 *Write pseudo C/C++ code to represent circular linked list as an ADT.*

SPPU - May 19, 6 Marks

```
#include<iostream>
#include<malloc.h>
using namespace std;

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

struct CircNode *addInEmpty(struct CircNode *last, int
new_data)
{

  if (last != NULL)
  return last;

  struct CircNode *temp = new CircNode;

  //assign data to new node
  temp -> data = new_data;
  last = temp;

  last->next = last;

  return last;
}

struct CircNode *addAtBegin(struct CircNode *last, int
new_data)
{
    if (last == NULL)
  return addInEmpty(last, new_data);

  struct CircNode *temp = new CircNode;
```

Insert a new node in an empty list

If last is not null then list is not empty, so return

Allocate memory for node

Create the link

Add new node at the beginning of the list

If list is empty then add the node by calling addInEmpty

Else create a new node

Set new data to node

```
temp -> data = new_data;
temp -> next = last -> next;
last -> next = temp;

return last;
}
```

Add new node at the end of the list

```
struct CircNode *addAtEnd(struct CircNode *last, int new_data)
{
```

If list is empty then add the node by calling addInEmpty

```
if (last == NULL)
return addInEmpty(last, new_data);
```

Else create a new node

```
struct CircNode *temp = new CircNode;
```

Assign the data

```
temp -> data = new_data;
temp -> next = last -> next;
last -> next = temp;
last = temp;

return last;
}
```

Add a new node in between the nodes

```
struct CircNode *addAfter(struct CircNode *last, int new_data, int after_item)
{
//return null if list is empty
if (last == NULL)
return NULL;

struct CircNode *temp, *p;
p = last -> next;
do
{
  if (p ->data == after_item)
  {
    temp = new CircNode;
    temp -> data = new_data;
    temp -> next = p -> next;
    p -> next = temp;

    if (p == last)
    last = temp;
    return last;
   }
  p = p -> next;
} while(p != last -> next);

cout << "The node with data "<<after_item << " is not present in the list." << endl;
return last;
}
//traverse the circular linked list
void traverseList(struct CircNode *last) {
struct CircNode *p;

// If list is empty, return.
if (last == NULL) {
cout << "Circular linked List is empty." << endl;
return;
}
p = last -> next; // Point to the first Node in the list.
```

Traverse the list starting from first node until first node is visited again

```
do {
    cout << p -> data << "==>";
    p = p -> next;
    } while(p != last->next);
  if(p == last->next)
  cout<<p->data;
  cout<<"\n\n";
  }
```

```
//delete the node from the list
void deleteNode(CircNode** head, int key)
{
  // If linked list is empty retun
  if (*head == NULL)
  return;
```

If the list contains only a single node,delete that node; list is empty

```
  if((*head)->data==key && (*head)->next==*head) {
    free(*head);
    *head=NULL;
    }
CircNode *last=*head,*d;

// If key is the head
if((*head)->data==key) {
  while(last->next!=*head) // Find the last node of the list
  last=last->next;
```

Point last node to next of head or second node of the list

```
    last->next=(*head)->next;
  free(*head);
  *head=last->next;
  }
```

End of list is reached or node to be deleted is not present in the list

```
while(last->next!=*head&&last->next->data!=key)
{
  last=last->next;
}
```

Node to be deleted is found, so free the memory and display the list

```
if(last->next->data==key)
{
    d=last->next;
    last->next=d->next;
    cout<<"The node with data "<<key<<" deleted from the list"<<endl;
    free(d);
    cout<<endl;
    cout<<"Circular linked list after deleting "<<key<<" is as follows:"<<endl;
    traverseList(last);
    }
  else
  cout<<"The node with data "<< key << " not found in the list"<<endl;
  }

// main Program
int main()
{
  struct CircNode *last = NULL;

  last = addInEmpty(last, 30);
  last = addAtBegin(last, 20);
  last = addAtBegin(last, 10);
  last = addAtEnd(last, 40);
  last = addAtEnd(last, 60);
  last = addAfter(last, 50,40 );
  cout<<"The circular linked list created is as follows:"<<endl;
  traverseList(last);
  deleteNode(&last,10);
  return 0;
}
```

Output

```
D:\link1.exe
The circular linked list created is as follows:
10==>20==>30==>40==>50==>60==>10

The node with data 10 deleted from the list

Circular linked list after deleting 10 is as follows:
20==>30==>40==>50==>60==>20
```

Applications of Circular Linked List(

UQ. 1.18.5 *Give practical applications of circular linked* — SPPU - May 17, 4 Marks

1. Circular lists are used in applications where the entire list is accessed one-by-one in a loop. Example : Operating systems may use it to switch between various running applications in a circular loop.
2. It is also used by Operating system to share time for different users, generally uses Round-Robin time sharing mechanism.
3. Multiplayer games uses circular list to swap between players in a loop.
4. Implementation of Advanced data- structures like Fibonacci Heap.

Syllabus Topic : Case Studies : Set Operation, String Operation

1.19 CASE STUDIES : SET OPERATION, STRING OPERATION

A. Set operations

The set operations are Union, Intersection, and set difference.

i. Union

- Union of the sets A and B, denoted by A ∪ B, is the set of distinct element belongs to set A or set B, or both.
- **Example :** Find the union of A = {2, 3, 4} and B = {3, 4, 5};

  Solution : A ∪ B = {2, 3, 4, 5}.

ii. Intersection

- The intersection of the sets A and B, denoted by A ∩ B, is the set of elements belongs to both A and B i.e. set of the common element in A and B.
- **Example :** Consider the previous sets A and B.

  Find out A ∩ B.

  Solution : A ∩ B = {3, 4}.

iii. Set Difference

- Difference between sets is denoted by 'A – B'. It is the set containing elements of set A which are not present in set B.
- **Example :** Consider the previous sets A and B.

  Find out A - B.

  Solution : A - B = {2}.

B. String operations

String Operations without using Library Functions

➢ **Program 1.19.1 :** Convert lower character into upper character.

```
# include<iostream.h>
# include<stdio.h>
# include<conio.h>
main()
{
char array[5];
int i=0;
for(i=0;i<5;i++)
{
cout<<"\n Enter character in lowercase:";
array[i]=getche();
```

getche() is a predefined library function used to accept character from user.

```
array[i]=array[i] -32;
```

To convert lower case to upper case we have to subtract 32 from it.

```
}
for (i=0;i<5;i++)
{
cout<<"\n\n The converted character is "<<array[i];
}
getch();
}
```

Output

```
E:\teju\pheonix\upper to lower.exe

Enter character in lowercase:h
Enter character in lowercase:e
Enter character in lowercase:l
Enter character in lowercase:l
Enter character in lowercase:o

The converted character is  H

The converted character is  E

The converted character is  L

The converted character is  L

The converted character is  O
```

➢ **Program 1.19.2 :** Calculate the length of string

GQ. 1.19.1 *Write a program to accept a string and display the length of it without using standard library function.* **(3 Marks)**

☑ **Soln. :**

Program

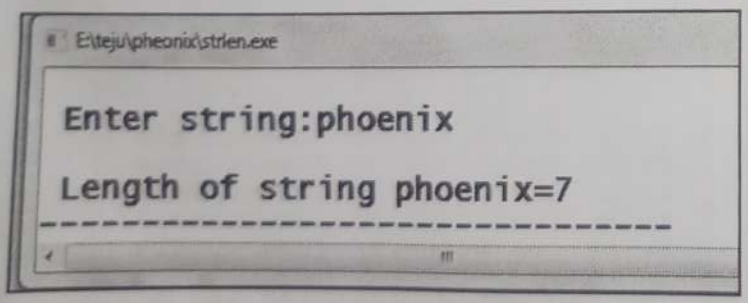
```
# include<iostream.h>
int main()
{
int i=0;
char str[10];
cout<<"\n Enter string:";

cin>>str;

while (str[i]!='\0')
{
i++;
}
cout<<"\n Length of string << str<<" "<<i;
return 1;
}
```

Loop will continue until end of string.

Output



➢ **Program 1.19.3 :** Compare two string.

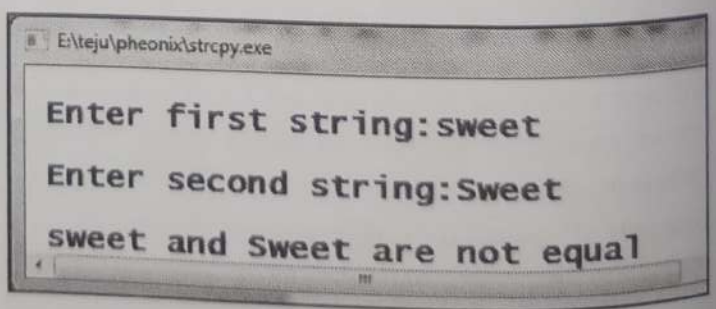
GQ. 1.19.2 *Write a program to accept two string and check whether these are equal or not.*

☑ **Soln. :**

Program

```
# include<iostream.h>
int main()
{
char first_str[10], second_str[10];
int i=0, flag=1;
cout<<"\n Enter first string:";
cin>>first_str;
cout<<"\n Enter second string:";
cin>>second_str;
while(first_str[i]!='\0' && second_str[i]!='\0')
{
if (first_str[i]!=second_str[i])
{
flag=0;
break;
}
}
if (flag==1)
{
cout<< first_str<<" and "<<second_str<<" are equal";
}
else
{
cout<< first_str<<" and "<<second_str<<" are not equal";
}
return 1;
}
```

Output



..Chapter Ends