SUBJECT CODE : 210242

As per Revised Syllabus of

SAVITRIBAI PHULE PUNE UNIVERSITY

Choice Based Credit System (CBCS)

S.E. (Computer) Semester - I

FUNDAMENTALS OF DATA STRUCTURES

(For END SEM Exam - 70 Marks)

Mrs. Anuradha A. Puntambekar

M.E. (Computer)
Formerly Assistant Professor in
P.E.S. Modern College of Engineering,
Pune

Dr. Priya Jeevan Pise

Ph.D. (Computer Engineering)
Associate Professor & Head
Indira College of Engineering
& Management, Pune

Dr. Prashant S. Dhotre

Ph.D. (Computer Engineering)
Associate Professor
JSPM's Rajarshi Shahu College of Engineering,
Tathawade, Pune

TECHNICAL PUBLICATIONS
SINCE 1993
An Up-Thrust for Knowledge

FUNDAMENTALS OF DATA STRUCTURES

(For END SEM Exam - 70 Marks)

Subject Code : 210242

S.E. (Computer Engineering) Semester - I

Published by :

TECHNICAL PUBLICATIONS
SINCE 1993 An Up-Thrust for Knowledge

Amit Residency, Office No.1, 412, Shaniwar Peth,
Pune - 411030, M.S. INDIA, Ph.: +91-020-24495496/97
Email : sales@technicalpublications.org Website : www.technicalpublications.org

Printer :
Yogiraj Printers & Binders
Sr.No. 10/1A,
Ghule Industrial Estate, Nanded Village Road,
Tal. - Haveli, Dist. - Pune - 411041.

ISBN 978-93-332-2154-2

9789333221542

SPPU 19

PREFACE

The importance of **Fundamentals of Data Structures** is well known in various engineering fields. Overwhelming response to our books on various subjects inspired us to write this book. The book is structured to cover the key aspects of the subject **Fundamentals of Data Structures.**

The book uses plain, lucid language to explain fundamentals of this subject. The book provides logical method of explaining various complicated concepts and stepwise methods to explain the important topics. Each chapter is well supported with necessary illustrations, practical examples and solved problems. All the chapters in the book are arranged in a proper sequence that permits each topic to build upon earlier studies. All care has been taken to make students comfortable in understanding the basic concepts of the subject.

Representative questions have been added at the end of each section to help the students in picking important points from that section.

The book not only covers the entire scope of the subject but explains the philosophy of the subject. This makes the understanding of this subject more clear and makes it more interesting. The book will be very useful not only to the students but also to the subject teachers. The students have to omit nothing and possibly have to cover nothing more.

We wish to express our profound thanks to all those who helped in making this book a reality. Much needed moral support and encouragement is provided on numerous occasions by our whole family. We wish to thank the **Publisher** and the entire team of **Technical Publications** who have taken immense pain to get this book in time with quality printing.

Any suggestion for the improvement of the book will be acknowledged and well appreciated.

Authors
A. A. Puntambekar
Dr. Priya Jeevan Pise
Dr. Prashant S. Dhotre

Dedicated to God.

SYLLABUS

Fundamentals of Data Structures (210242)

Credit	Examination Scheme and Marks
03	End_Sem (Theory) : 70 Marks

Unit III Searching and Sorting

Searching : Search Techniques-Sequential Search/Linear Search, Variant of Sequential Search- Sentinel Search, Binary Search, Fibonacci Search, and Indexed Sequential Search. **Sorting** : Types of Sorting-Internal and External Sorting, General Sort Concepts-Sort Order, Stability, Efficiency, and Number of Passes, Comparison Based Sorting Methods-Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, Shell Sort, Non-comparison Based Sorting Methods-Radix Sort, Counting Sort, and Bucket Sort, Comparison of All Sorting Methods and their complexities. **(Chapter - 3)**

Unit IV Linked List

Introduction to Static and Dynamic Memory Allocation, **Linked List** : Introduction of Linked Lists, Realization of linked list using dynamic memory management operations, Linked List as ADT, **Types of Linked List** : singly linked, linear and Circular Linked Lists, Doubly Linked List, Doubly Circular Linked List, Primitive Operations on Linked List-Create, Traverse, Search, Insert, Delete, Sort, Concatenate. Polynomial Manipulations-Polynomial addition. Generalized Linked List (GLL) concept, Representation of Polynomial using GLL. **(Chapter - 4)**

Unit V Stack

Basic concept, stack Abstract Data Type, Representation of Stacks Using Sequential Organization, stack operations, Multiple Stacks, Applications of Stack- Expression Evaluation and Conversion, Polish notation and expression conversion, Need for prefix and postfix expressions, Postfix expression evaluation, Linked Stack and Operations. **Recursion** - concept, variants of recursion - direct, indirect, tail and tree, Backtracking algorithmic strategy, use of stack in backtracking. **(Chapter - 5)**

Unit VI Queue

Basic concept, Queue as Abstract Data Type, Representation of Queue using Sequential organization, Queue Operations, Circular Queue and its advantages, Multi-queues, Linked Queue and Operations. **Deque**-Basic concept, types (Input restricted and Output restricted), Priority Queue- Basic concept, types(Ascending and Descending). **(Chapter - 6)**

Table of Contents

Unit - III

Unit - IV

Unit - V

Unit - VI

Unit - III

3 Searching and Sorting

Syllabus

Searching : *Search Techniques-Sequential Search/Linear Search, Variant of Sequential Search-Sentinel Search, Binary Search, Fibonacci Search, and Indexed Sequential Search.*

Sorting : *Types of Sorting-Internal and External Sorting, General Sort Concepts-Sort Order, Stability, Efficiency, and Number of Passes, Comparison Based Sorting Methods-Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, Shell Sort, Non-comparison Based Sorting Methods-Radix Sort, Counting Sort, and Bucket Sort, Comparison of All Sorting Methods and their complexities.*

Contents

Part I : Searching

3.1 Introduction to Searching and Sorting

Importance of Searching and Sorting

Searching technique is essential for locating the position of the required element from the heap of data.

Application of Sorting

Sorting is useful for arranging the data in desired order. After sorting the required element can be located easily.

1. The sorting is useful in database applications for arranging the data in desired order.
2. In the dictionary like applications the data is arranged in sorted order.
3. For searching the element from the list of elements, the sorting is required.
4. For checking the uniqueness of the element the sorting is required.
5. For finding the closest pair from the list of elements the sorting is required.

3.2 Search Techniques

- When we want to find out particular record efficiently from the given list of elements then there are various methods of searching that element. These methods are called **searching methods.** Various algorithms based on these searching methods are known as **searching algorithm**s.
- The **basic characteristic** of any searching algorithm is -

  i. It should be **efficient**

  ii. **Less number of computations** must be involved in it.

  iii. The **space** occupied by searching algorithms must be **less.**
- The most commonly used searching algorithms are -

  i. Sequential or linear search

  ii. Indexed sequential search

  iii. Binary search
- The element to be searched from the given list is called the **key** element.

Let us discuss various searching algorithms.

3.2.1 Sequential Search/Linear Search

- Sequential search is technique in which the given list of elements is scanned from the beginning. The key element is compared with every element of the list. If the match is found the searching is stopped otherwise it will be continued to the end of the list.
- Although this is a **simple method,** there are some **unnecessary comparisons** involved in this method.
- The time complexity of this algorithm is **O(n).** The time complexity will increase linearly with the value of n.
- For **higher value of n** sequential search is **not satisfactory solution.**
- **Example**

Array

	Roll no	Name	Marks
0	15	Parth	96
1	2	Anand	40
2	13	Lalita	81
3	1	Madhav	50
4	12	Arun	78
5	3	Jaya	94

Fig. 3.2.1 Represents students Database for sequential search

From the above Fig. 3.2.1 the array is maintained to store the students record. The record is not sorted at all. If we want to search the student's record whose roll number is 12 then with the key-roll number we will see the every record whether it is of roll number = 12. We can obtain such a record at Array [4] location.

C++ Function

```
int search(int a[size],int key)
{
 for(i=0;i<n;i++)
        {
                if(a[i]==key)
                        return 1;
        }
return 0;
}
```

Python Program

```
def search(arr,x):

    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1

print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)

print("Resultant array is\n")
print(array)

print("\n Enter the key element to be searched: ")
key = int(input())

location = search(array,key)
print("The element is present at index:",location)
```

Output

```
How many elements are there in Array?
5
 Enter element in Array
30
 Enter element in Array
10
 Enter element in Array
40
 Enter element in Array
50
 Enter element in Array
20
Resultant array is
[30, 10, 40, 50, 20]
 Enter the key element to be searched:
40
The element is present at index: 2
>>>
```

Advantages of sequential searching

1. It is simple to implement.
2. It does not require specific ordering before applying the method.

Disadvantages of sequential searching

1. It is less efficient.

3.2.2 Variant of Sequential Search-Sentinel Search

- The sentinel value is a specialized value that acts as a flag or dummy data. This specialized value is used in context of an algorithm which uses its presence as a condition of termination.

Typical examples of sentinel values are -

1. Null character used at the end of the string.
2. Null pointer at the end of the linked list.
3. End of file character.

- In sentinel search, when searching for a particular value in an unsorted list, every element will be compared against this value.
- In this search, the last element of the array is replaced with the element to be searched and then the linear search is performed on the array without checking whether the current index is inside the index range of the array or not because the element to be searched will definitely be found inside the array even if it was not present in the original array since the last element got replaced with it. So, the index to be checked will never be out of bounds of the array.

For example

```
Input: [10,20,30,40,50,60]
Key: 40
Result: The element is present

Input: [10,20,30,40,50,60]
Key: 99
Result: The element is not present in the list.
```

Python Program

```
def Sentsearch(arr,x):
    l = len(arr)
    arr.append(x)
    i = 0
    while(arr[i]!=x):
        i = i+1
    if(i!=l):
```

```
        return i
    else:
        return -1

print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)

print("Resultant array is\n")
print(array)

print("\n Enter the key element to be searched: ")
key = int(input())

location = Sentsearch(array,key)
if(location != -1):
    print("The element is present at index:",location)
else:
    print("\n The element is not present in the list")
```

Output

```
How many elements are there in Array?
5
 Enter element in Array
30
 Enter element in Array
10
 Enter element in Array
50
 Enter element in Array
20
 Enter element in Array
40
Resultant array is
[30, 10, 50, 20, 40]
 Enter the key element to be searched:
50
The element is present at index: 2
```

3.2.3 Binary Search

- **Concept :** Binary search is a searching algorithm in which the list of elements is divided into two sublists and the key element is compared with the middle element. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves(sublists) depending upon the result produced through the match.

This algorithm is considered as an efficient searching algorithm.

Algorithm for Binary Search

1. if(low>high)
2. return;
3. mid =(low+high)/2;
4. if(x==a[mid])
5. return (mid);
6. if(x<a[mid])
7. search for x in a[low] to a[mid-1];
8. else
9. search for x in a[mid+1] to a[high];

Example :

As mentioned earlier the necessity of this method is that all the elements should be sorted. So let us take an array of sorted elements.

array

0	1	2	3	4	5	6	7
– 40	11	33	37	42	45	99	100

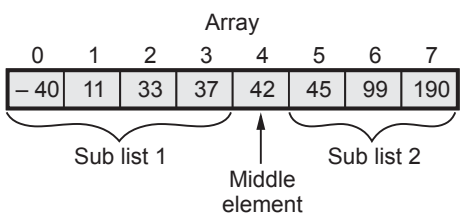
Step 1 : Now the key element which is to be searched is = 99 ∴ key = 99.

Step 2 : Find the middle element of the array. Compare it with the key

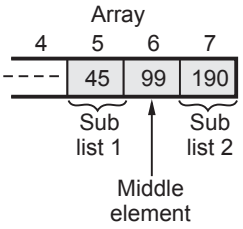
if middle ? key

i.e. if 42 ? 99

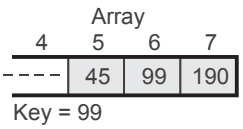
if 42 < 99 search the sublist 2



Now handle only sublist 2. Again divide it, find mid of sublist 2



if middle ? key
$=$
i.e. if 99 ? 99
$=$



So match is found at 7^{th} position of array i.e. at array [6]

Thus by binary search method we can find the element 99 present in the given list at array $[6]^{th}$ location.

Non Recursive Python Program

```
def Binsearch(arr,KEY):
    low = 0
    high = len(arr)-1
    m = 0
    while(low<=high):
        m =(low+high) //2       #mid of the array is obtained
        if(KEY<arr[m]):
            high = m-1          #search the left sub list
        elif(KEY>arr[m]):
            low = m+1           #search the right sub list
        else:
            return m

    return -1                   #if element is not present in the list
print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)

print("Resultant array is\n")
print(array)

print("\n Enter the key element to be searched: ")
key = int(input())

location = Binsearch(array,key)
```

```
if(location != -1):
    print("The element is present at index:",location)
else:
    print("\n The element is not present in the list")
```

Output

```
How many elements are there in Array?
5
 Enter element in Array
10
 Enter element in Array
20
 Enter element in Array
30
 Enter element in Array
40
 Enter element in Array
50
Resultant array is
[10, 20, 30, 40, 50]

Enter the key element to be searched:
40
The element is present at index: 3
>>>
```

Recursive Python Program

```
def BinRsearch(arr,KEY,low,high):
    if(high >= low):
        m =(low+high)//2                        #mid of the array is obtained
        if(arr[m] == KEY):
            return m
        elif(arr[m]>KEY):
            return BinRsearch(arr,KEY,low,m-1)  #search the left sub list
        else:
            return BinRsearch(arr,KEY,m+1,high) #search the right sub list
    else:
        return -1                               #if element is not present in the list

print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)
```

```
print("Resultant array is\n")
print(array)

print("\n Enter the key element to be searched: ")
key = int(input())
location = BinRsearch(array,key,0,len(array)-1)
if(location != -1):
    print("The element is present at index:",location)
else:
    print("\n The element is not present in the list")
```

Output

```
How many elements are there in Array?
5
 Enter element in Array
10
 Enter element in Array
20
 Enter element in Array
30
 Enter element in Array
40
 Enter element in Array
50
Resultant array is
[10, 20, 30, 40, 50]
 Enter the key element to be searched:
20
The element is present at index: 1
>>>
```

Example 3.2.1 *How many comparisons are required to find 73 in the following array using binary search 12, 25, 32, 37, 41, 48, 58, 60, 66, 73, 74, 79, 83, 91, 95 ?*

Solution :

Step 1 :

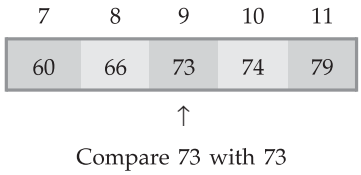
0	1	2	3	4	5	6	7	8	9	10	11
12	25	32	37	41	48	58	60	66	73	74	79

↑

Compare 73 with 58

Number of comparison = 1.

Step 2 : As 73 > 58. Search 73 between A[7] to A[11]



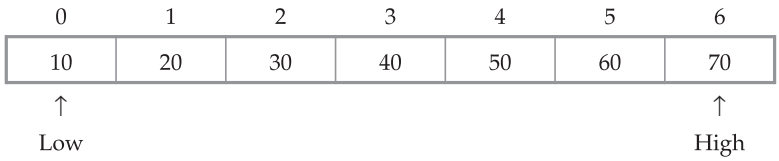
Number 73 is found.

∴ **Number of comparison = 2.**

Thus with two comparisons number 73 is found.

Example 3.2.2 *Apply binary search method to search 60 from the list 10, 20, 30, 40, 50, 60, 70.*

Solution : Consider a list of elements stored in array A as



The KEY element (i.e. the element to be searched) is 60.

Now to obtain middle element we will apply a formula :

$$m = (low + high)/2$$

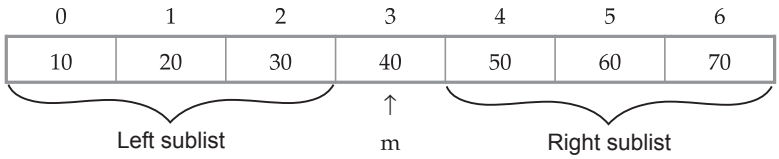
$$m = (0 + 6)/2$$

$$m = 3$$

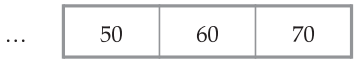
Then Check A[m] $\stackrel{?}{=}$ KEY

i.e. A[3] $\stackrel{?}{=}$ 60 NO A[3] = 40 and 40 < 60

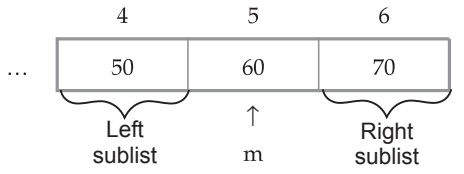
∴ Search the right sublist.



The right sublist is



Now we will again divide this list and check the mid element.



is $A[m] \stackrel{?}{=} KEY$

i.e. $A[5] \stackrel{?}{=} 60$ Yes, i.e. the number is present in the list.

Thus we can search the desired number from the list of elements.

Advantages and Disadvantages of Binary Search

Advantage

(1) It is efficient technique.

Disadvantages

(1) It requires specific ordering before applying the method.

(2) It is complex to implement.

Comparison between Linear Search and Binary Search

Sr. No.	Linear search method	Binary search method
1.	The linear search is a searching method in which the element is searched by scanning the entire list from first element to the last.	The binary search is a searching method in which the list is sub divided into two sub-lists. The middle element is then compared with the key element and then accordingly either left or right sub-list is searched.
2.	Many times entire list is searched.	Only sub-list is searched for searching the key element.
3.	It is simple to implement.	It involves computation for finding the middle element.
4.	It is less efficient searching method.	It is an efficient searching method.

3.2.4 Fibonacci Search

In binary search method we divide the number at mid and based on mid element i.e. by comparing mid element with key element we search either the left sublist or right sublist. Thus we go on dividing the corresponding sublist each time and comparing mid element with key element. If the key element is really present in the list, we can reach to the location of key in the list and thus we get the message that the "element is present in the list" otherwise get the message. "element is not present in the list."

In Fibonacci search rather than considering the mid element, we consider the indices as the numbers from fibonacci series. As we know, the Fibonacci series is -

0 1 1 2 3 5 8 13 21 …

To understand how Fibonacci search works, we will consider one example, suppose, following is the list of elements.

arr [ ]

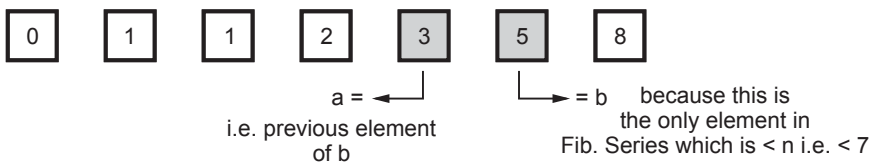
10	20	30	40	50	60	70
1	2	3	4	5	6	7

Here n = total number of elements = 7

We will always compute 3 variables i.e. a, b and f.

Initially f = n = 7.

For setting a and b variables we will consider elements from Fibonacci series.



Now we have

f = 7

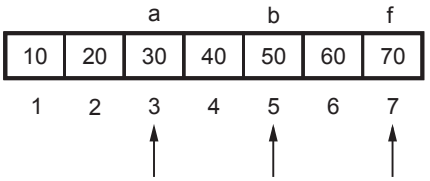
b = 5

a = 3

With these initial values we will start searching the key element from the list. Each time we will compare key element with arr [f]. That means

If (Key < arr [f])	If (Key > arr [f])
f = f – a	f = f + a
b = a	b = b – a
a = b – a	a = a – b

Suppose we have **f = 7, b = 5, a = 3**



If **Key = 20**

i.e. Key < arr[f]

i.e. 20 < 70

∴ f = f – a = 7 – 3 = 4

b = a = 3

a = b – a = 2

Again we compare if (Key < arr [f])

i.e. 20 < arr [4]

i.e. if (20 < 40) → Yes

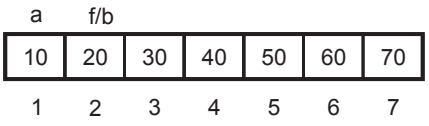
At Present f = 4, b = 3, a = 2

∴ f = f – a = 4 – 2 = 2

b = a = 2

a = b – a = 3 – 2 = 1

Now we get f = 2, b = 2, a = 1



If **Key = 60**

i.e. Key < arr [f]

60 < 70

∴ f = f – a = 7 – 3 = 4

b = a = 3

a = b – a = 2

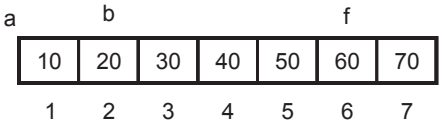
Again we compare if (Key > arr [f])

∴ i.e. 60 > arr [f] i.e. 40

∴ f = f + a = 4 + 2 = 6

b = b – a = 3 – 2 = 1

a = a – b = 2 – 3 = – 1



If (Key < arr [f])

i.e. if (60 < 60) → No

If (Key < arr [f]) i.e. if (20 < 20) → No
If Key > arr [f]) i.e. if (20 > 20) → No
That means "**Element is present at f = 2 location**"

If (key > arr [f])
i.e. if (60 > 60) → No
That means **"Element is present at f = 6 location."**

Analysis : The time complexity of fibonacci search is **O(logn).**

Algorithm

Let the length of given array be **n [0...n-1]** and the element to be searched be **key**

Then we use the following steps to find the element with minimum steps :

1. Find the **smallest Fibonacci number greater than or equal to n.** Let this number be f(m^{th} element).
2. Let the two Fibonacci numbers preceding it be **a**($m\text{-}1^{th}$ element) and **b**($m\text{-}2^{th}$ element)

   While the array has elements to be checked :

   Compare key with the last element of the range covered by **b**

   (a) If **key** matches, return index value

   (b) Else if **key is less** than the element, move the third Fibonacci variable two Fibonacci down, indicating removal of approximately two-third of the unsearched array.

   (c) Else key is greater than the element, move the third Fibonacci variable one Fibonacci down. Reset offset to index. Together this results into removal of approximately front one-third of the unsearched array.
3. Since there might be a single element remaining for comparison, check if **a** is '1'. If Yes, compare key with that remaining element. If match, return index value.

For example -

According to the algorithm we have to sort the elements of the array prior to applying Fibonacci search. Consider sorted array of elements as -

0	1	2	3	4	5	6
10	20	30	40	50	60	70

∴ n = 7, we want to find key = 60

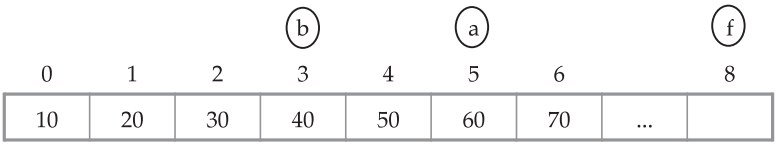
Now check Fibbonacci series.

As n < 8

set f = 8

a = 5

b = 3



Set offset = – 1

∴ i = 2 ∵ 1 = min (offset + b, n – 1)

A[i] = A[2] < (key = 60)

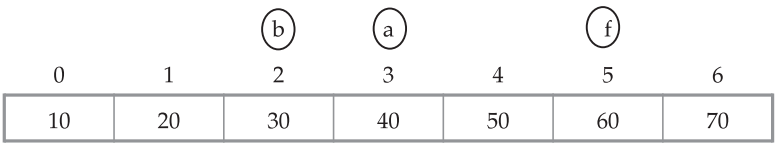
Here key is greater than the element

We move f one fibbonacci down (step 2C of algorithm)

∴ f = 5

a = 3

b = 2



Set offset = i i.e. = **2**

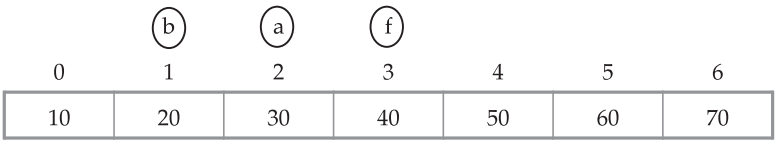
Now i = 4 ∵ i = min (offset + b, n – 1)

Here key is again greater than the element. So we move f, one fibbonacci down

∴ f = 3

a = 2

b = 1



Set offset = i i.e. **4**

Now new i = 5 ∵ i = min (offset + b, n – 1)

Here a[i] = key. Hence return value of i as the position of key element.

Note that due to fibonacci numbering the search portion is restricted and we need to compare very less number of elements.

Python Program

```
def FibSearch(arr, key, n):
    # Initialize Fibonacci numbers
    b = 0
    a = 1
    f = b + a

    # f is going to store the smallest
    # Fibonacci Number greater than or equal to n
    while (f < n):
        b = a
        a = f
        f = b + a
      # Marks the eliminated range from front
    offset = -1;

    # while there are elements to be inspected.
    # we compare arr[i] with key.
    while (f > 1):

        # Check if b is a valid location
        i = min(offset+b, n-1)

        # If key is greater than the value at
        # index b, cut the subarray array
        # from offset to i
        if (arr[i] < key):
            f = a
            a = b
            b = f - a
            offset = i

        # If key is lesser than the value at
        # index b, cut the subarray
        # after i+1
        elif (arr[i] > key):
            f = b
            a = a - b
            b = f - a

        # element found. return index
        else :
            return i

    # comparing the last element with key
    if(a and arr[offset+1] == key):
        return offset+1;
```

```
    # element not found. return -1
    return -1

print("\n Program For Fibonacci Search")
print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)

print("Resultant array is\n")
print(array)

print("\n Enter the key element to be searched: ")
key = int(input())
print("\n The element is present at index: ",FibSearch(array,key,n))
```

Output

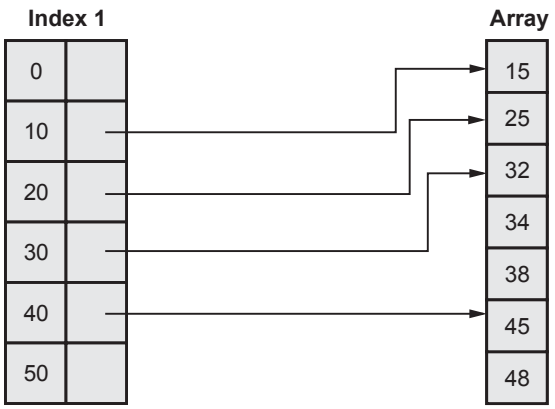
```
Program For Fibonacci Search
How many elements are there in Array?
7
 Enter element in Array
10
 Enter element in Array
20
 Enter element in Array
30
 Enter element in Array
40
 Enter element in Array
50
 Enter element in Array
60
 Enter element in Array
70
Resultant array is
[10, 20, 30, 40, 50, 60, 70]
 Enter the key element to be searched:
60
 The element is present at index:  5
>>>
```

Analysis : From the above algorithm it is clear if we have to search the larger section of the array then the time taken will be more and will result into worst case and it's complexity wil be O(log n). If on the very first search, we get our element then it will be considered as the best case and complexcity will be O(1).

3.2.5 Indexed Sequential Search

- Index sequential search is a searching technique in which a separate table containing the indices of the actual elements is maintained.
- The actual search of the element is done with the help of this index table.

For example -



- Note that the sorted index table is maintained. First we search the index table and then with the help of an index, actual array is searched for the element.
- For instance - If we want to search an element 34, then we first locate to index 30 which is present in the index table and then, we compare element 32 and then 34 in the actual array.

Python Program

```
def IndexSeq(arr,key,n):
    Elements = [0]*10
    Index = [0]*10
    flag = 0
    ind = 0
    start=end=0
    for i in range(0,n,2) :
        # Storing element
        Elements[ind] = arr[i]
        # Storing the index
        Index[ind] = i
```

```
        ind += 1

    if (key < Elements[0]):
        print("Element is not present")
        exit(0)

    else:
        for i in range(1, ind + 1) :
            if(key < Elements[i]) :
                start = Index[i - 1]
                end = Index[i]
                break

    for i in range(start, end + 1) :
        if (key == arr[i] ):
            flag = 1
            break

    if flag == 1 :
        print("Element is Found at index", i)
    else :
        print("Element is not present")

print("\n Program For Index Sequential Search")
array = [11,15,22,24,26,32,34,38]
n = len(array)
key=32
IndexSeq(array,key,n)
```

Output

```
Program For Index Sequential Search
Element is Found at index 5
```

Part II : Sorting

3.3 Types of Sorting-Internal and External Sorting

Definition : Sorting is systematic arrangement of data.

3.3.1 Internal and External Sorting

Sorting can be of two types **internal sorting and external sorting.**

Internal Sorting :

- The internal sorting is a sorting in which the data resides in the main memory of the computer.

- Various methods that make use of internal sorting are -
- 1. Bubble Sort 2. Insertion Sort 3. Selection Sort 4. Quick Sort 5. Radix Sort and so on.

- **External Sorting :**
  - For many applications it is not possible to store the entire data on the main memory for two reasons, i) amount of **main memory available is smaller** than amount of data. ii)Secondly the **main memory is a volatile device** and thus will lost the data when the power is shut down. To overcome these problems the data is stored on the secondary storage devices. The technique which is used to sort the data which resides on the secondary storage devices are called **external sorting.**
  - The data stored on secondary memory is part by part loaded into main memory, sorting can be done over there. The sorted data can be then stored in the intermediate files. Finally these intermediate files can be merged repeatedly to get sorted data. Thus **huge amount of data** can be sorted using this technique.

For example : Consider that there are 10,000 records that has to be sorted. Clearly we need to apply external sorting method. Suppose main memory has a capacity to store 500 records in blocks, with each block size of 100 records.

The sorted 5 blocks (i.e. 500 records) are stored in intermediate file. This process will be repeated 20 times to get all the records sorted in chunks.

In the second step, we start merging a pair of intermediate files in the main memory to get output file.

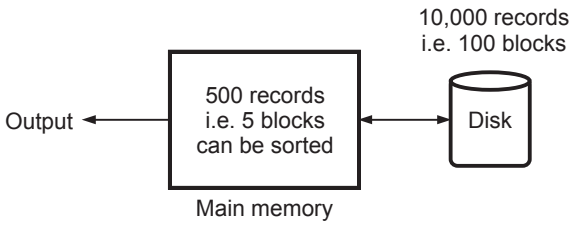


Fig. 3.3.1

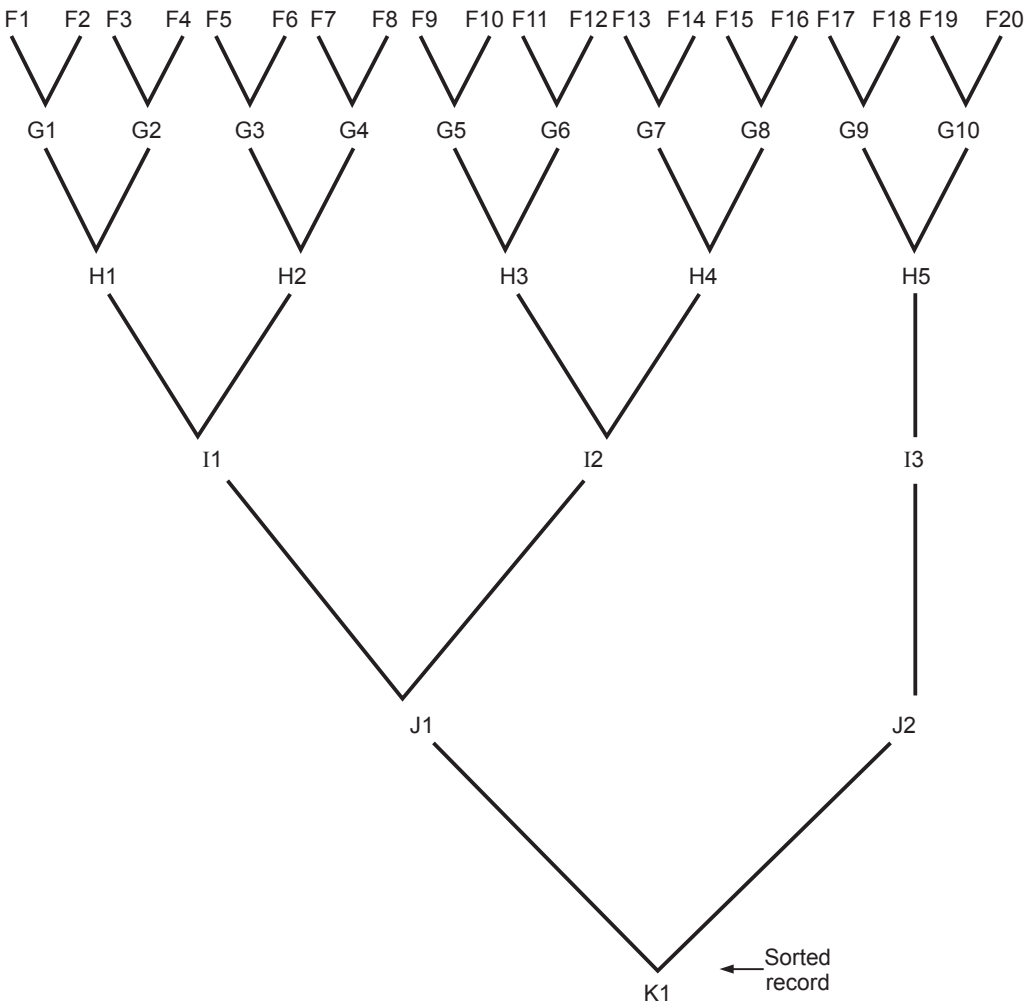


Fig. 3.3.2

3.4 General Sort Concepts

Before learning the actual sorting techniques let us understand some basic concepts of sorting.

3.4.1 Sort Order

The sorting is a technique by which we expect the list of elements to be arranged as we expect. Sorting order is nothing but the arrangement of the elements in some specific manner. Usually the sorting order is of two types -

Ascending order : It is the sorting order in which the elements are arranged from low value to high value. In other words elements are in increasing order.

For example : 10, 50, 40, 20, 30

can be arranged in ascending order after applying some sorting technique as

10, 20, 30, 40, 50

Descending Order : It is the sorting order in which the elements are arranged from high value to low value. In other words elements are in decreasing order. It is reverse of the ascending order.

For example : 10, 50, 40, 20, 30

can be arranged in descending order after applying some sorting technique as

50, 40, 30, 20, 10

While sorting the elements, we always consider a specific order and expect our data to be arranged in that order.

3.4.2 Sort Stability

The sorting stability means comparing the records of same value and expecting them in the same order even after sorting them.

For example :

(Pune, BalGandharva)

(Pune, Shaniwarwada)

(Nasik, Panchavati)

(Mumbai, Gateway-of-India)

Now in the above list, we will sort the list according to the first alphabet of the city. The ascending order for the alphabets P(for Pune), N(for Nasik), M(for Mumbai) will be M, N, P. The sorting stability can be achieved by arranging the records as follows.

(Mumbai, Gateway-Of-India)

(Nasik, Panchavati)

(Pune, BalGandharva)

(Pune, Shaniwarwada)

Note that in the above list same record as Pune is twice. But we have preserved the original sequence as it is after comparing them. Thus the stability is achieved in sorting the records.

3.4.3 Efficiency and Passes

One of the major issue in the sorting algorithms is its efficiency. If we can efficiently sort the records then that adds value to the sorting algorithm. We usually denote the efficiency of sorting algorithms in terms of time complexity. The time complexities are given in terms of big-on notations.

Commonly there are $O(n^2)$ and O(nlogn) time complexities for various algorithms. The sorting techniques such as bubble sort, insertion sort, selection sort, shell sort has the time complexity $O(n^2)$ and the techniques such as merge sort, quick sort has the time complexity as O(nlogn). The quick sort is the fastest algorithm and bubble sort is the slowest one! Efficiency also depends on number of records to be sorted. After all sorting efficiency is nothing but how much time that algorithm have taken to sort the elements. That is why it is convenient to give the efficiency of sorting algorithms in terms of time complexity.

While sorting the elements in some specific order there is lot of arrangement of elements. The phases in which the elements are moving to acquire their proper position is called **passes**.

For example : 10, 30, 20, 50, 40

Pass 1 : 10, 20, 30, 50, 40

Pass 2 : 10, 20, 30, 40, 50

In the above method we can see that data is getting sorted in two passes distinctly. By applying a logic as comparison of each element with its adjacent element gives us the result in two passes.

Sorting Techniques

As mentioned above sorting is an important activity and every time we insert or delete the data we need to sort the remaining data. Therefore it should be carried out efficiently. Various algorithms are developed for sorting such as

1. Bubble Sort 2. Selection Sort 3. Insertion Sort
4. Radix Sort 5. Shell Sort 6. Merge Sort 7. Quick Sort

3.5 Comparison based Sorting Methods

3.5.1 Bubble Sort

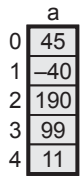
This is the simplest kind of sorting method in this method. We do this bubble sort procedure in several iterations, which are called passes.

Example bubble sort :

Consider 5 unsorted elements are

45 – 40 190 99 11

First store those elements in the array a



Pass 1

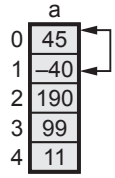
In this pass each element will be compared with its neighbouring element.

Compare 45 and – 40

Is 45 > – 40 ∴ Interchange

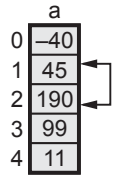
i.e. compare a[0] and a[1], after interchange

∴ a[0] = – 40



Compare a[1] and a[2]

Is 45 > 190 ∴ No interchange

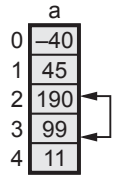


Compare a[2] and a[3]

Is 190 > 99 ∴ Interchange

a[2] = 99

a[3] = 190

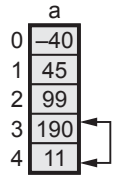


Compare a[3] and a[4]

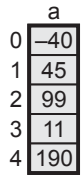
Is 190 > 11 ∴ Interchange

a[3] = 11

a[4] = 190



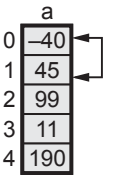
After first pass the array will hold the elements which are sorted to some extent.



Pass 2

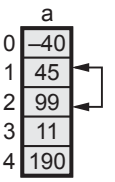
Compare a[0] and a[1]

No interchange



Compare a[0] and a[1]

No interchange.

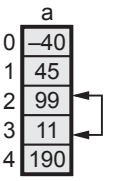


Compare a[2] and a[3]

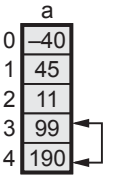
Since 99 > 11 interchange

∴ a[2] = 11

a[3] = 99



No interchange



Pass 3

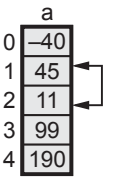
First compare a[0] and a[1], no interchange

∴ Compare a[1] and a[2]

45 > 11 ∴ Interchange

a[1] = 11

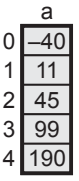
a[2] = 45



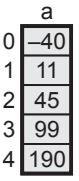
Next compare a[2] and a[3] similarly

a[3] and a[4]

No interchange.



This is end of pass 3. This process will be thus continued till pass 4. Finally at the end of last pass the array will hold all the sorted elements like this,



Since the comparison positions look like bubbles, therefore it is called **bubble sort**.

Algorithm :

1. Read the total number of elements say n
2. Store the elements in the array
3. Set the i = 0 .
4. Compare the adjacent elements.
5. Repeat step 4 for all n elements.
6. Increment the value of i by 1 and repeat step 4, 5 for i< n
7. Print the sorted list of elements.
8. Stop.

Python Program

```
def Bubble(arr,n):
    i = 0

    for i in range(n-1):
        for j in range(0,n-i-1):
            if(arr[j] > arr[j+1]):
                temp = arr[j]
                arr[j] = arr[j+1]
                arr[j+1] = temp
        print("\nPass#",(i+1))
        print(arr)

print("\n Program For Bubble Sort")
print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)

print("Original array is\n")
print(array)

print("\n Sorted Array is")
Bubble(array,n)
```

Output

```
Program For Bubble Sort
How many elements are there in Array?
5
 Enter element in Array
```

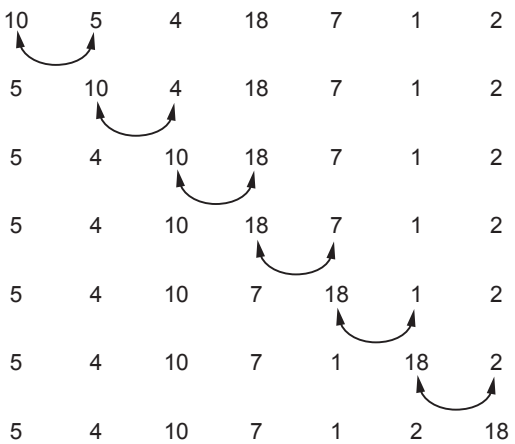
```
30
 Enter element in Array
10
 Enter element in Array
20
 Enter element in Array
50
 Enter element in Array
40
Original array is
[30, 10, 20, 50, 40]
 Sorted Array is
Pass# 1
[10, 20, 30, 40, 50]
Pass# 2
[10, 20, 30, 40, 50]
Pass# 3
[10, 20, 30, 40, 50]
Pass# 4
[10, 20, 30, 40, 50]
>>>
```

Analysis : In above algorithm basic operation if (a[j] > a[j +1]) which is executed within nested for loops. Hence time complexity of bubble sort is $\Theta(n)^2$.

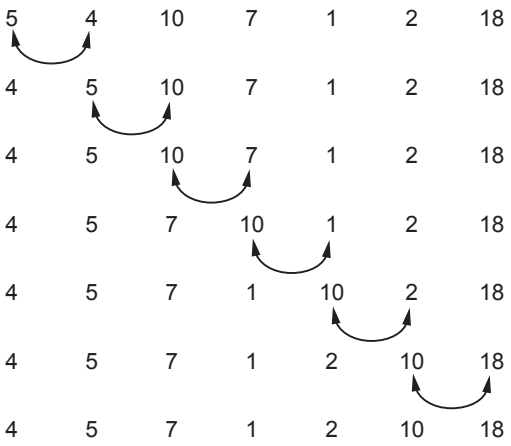
Example 3.5.1 *Show the output of each pass for the following list*
10, 5, 4, 18, 17, 1, 2

Solution : Let, 10, 5, 4, 18, 17, 1, 2 be the given list of elements. We will compare adjacent elements say A[i] and A[j]. If A[i] > A[j] then swap the elements.

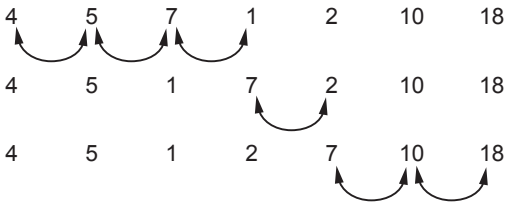
Pass I



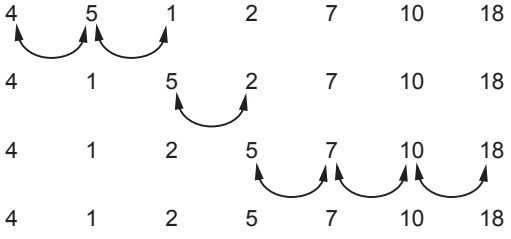
Pass II



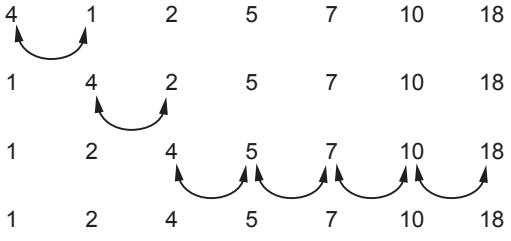
Pass III



Pass IV



Pass V



Pass VI

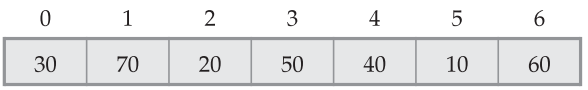
This is the sorted list of elements.

3.5.2 Insertion Sort

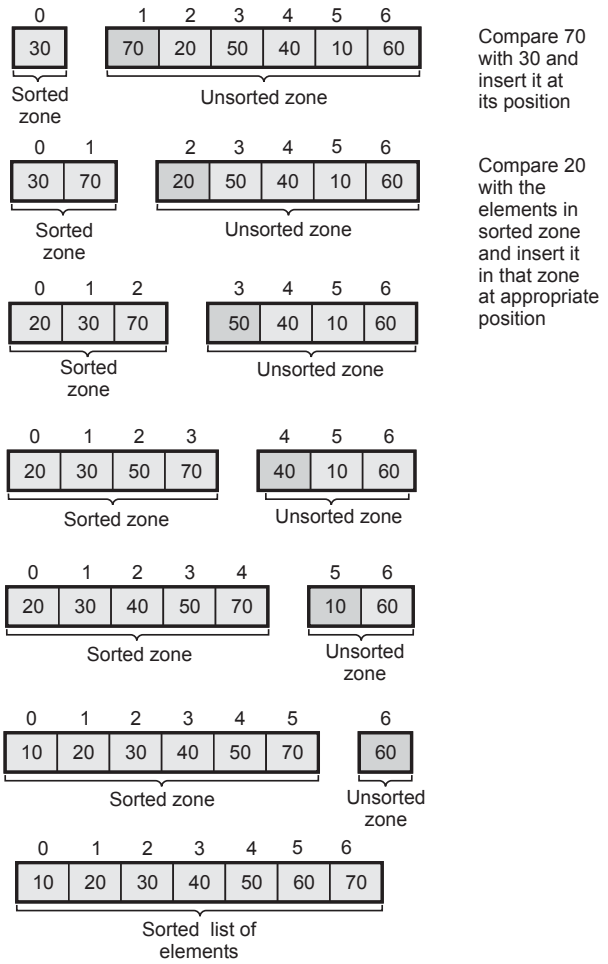
In this method the elements are inserted at their appropriate place. Hence is the name **insertion sort**. Let us understand this method with the help of some example -

For Example

Consider a list of elements as,



The process starts with first element



Algorithm

Although it is very natural to implement insertion using recursive(top down) algorithm but it is very efficient to implement it using bottom up(iterative) approach.

```
Algorithm Insert_sort(A[0…n-1])
//Problem Description: This algorithm is for sorting the
//elements using insertion sort
//Input: An array of n elements
//Output: Sorted array A[0…n-1] in ascending order
  for i ← 1 to n-1 do
{
            temp ← A[i]//mark A[i]th element
            j ← i-1//set j at previous element of A[i]
            while(j>=0)AND(A[j]>temp)do
            {
      //comparing all the previous elements of A[i] with
      //A[i].If any greater element is found then insert
      //it at proper position
            A[j+1] ← A[j]
            j ← j-1
            }
            A[j+1] ← temp //copy A[i] element at A[j+1]
}
```

Analysis

When an array of elements is almost sorted then it is **best case** complexity. The best case time complexity of insertion sort is **O(n).**

If an array is randomly distributed then it results in **average case** time complexity which is $\mathbf{O(n^2)}$.

If the list of elements is arranged in descending order and if we want to sort the elements in ascending order then it results in **worst case** time complexity which is $\mathbf{O(n^2)}$.

Advantages of insertion sort

1. *Simple* to implement.
2. This method is *efficient* when we want to sort small number of elements. And this method has excellent performance on almost sorted list of elements.
3. More efficient than most other simple $O(n^2)$ algorithms such as selection sort or bubble sort.
4. This is a *stable* (does not change the relative order of equal elements).

5. It is called *in-place* sorting algorithm (only requires a constant amount O(1) of extra memory space). The in-place sorting algorithm is an algorithm in which the input is overwritten by output and to execute the sorting method it does not require any more additional space.

Python Program

```
def InsertSort(arr,n):
    i = 1
    for i in range(n):
        temp = arr[i]
        j = i-1
        while((j>=0) & (arr[j]>temp)):
            arr[j+1] = arr[j]
            j = j-1
        arr[j+1] = temp

    print(arr)
print("\n Program For Insertion Sort")
print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)

print("Original array is\n")
print(array)

print("\n Sorted Array is")
InsertSort(array,n)
```

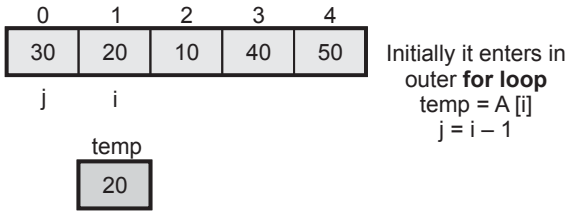
Output

```
Program For Insertion Sort

How many elements are there in Array?
5
 Enter element in Array
30
 Enter element in Array
10
 Enter element in Array
50
 Enter element in Array
40
 Enter element in Array
```

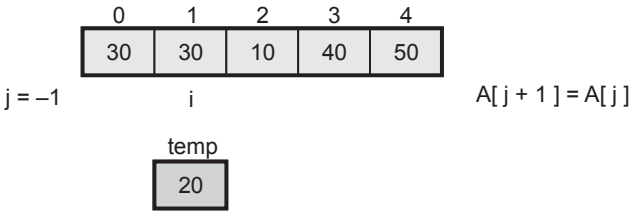
```
20
Original array is

[30, 10, 50, 40, 20]

 Sorted Array is
[10, 20, 30, 40, 50]
>>>
```

Logic Explanation :

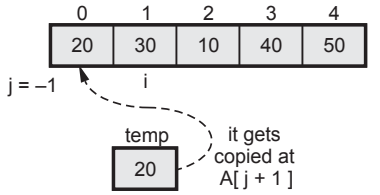
For understanding the logic of above python program consider a list of unsorted elements as,



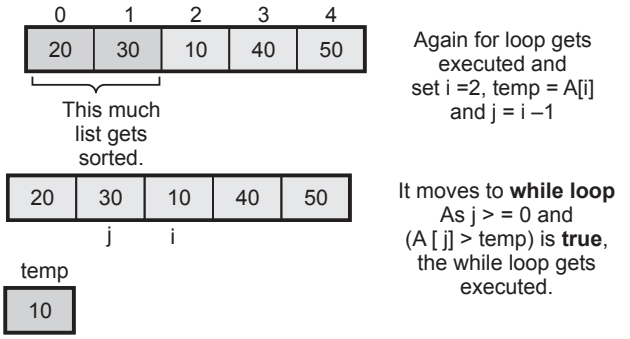
Then the control moves to **while loop**. As j >= 0 and A[j] > temp is **True**, the while loop will be executed.

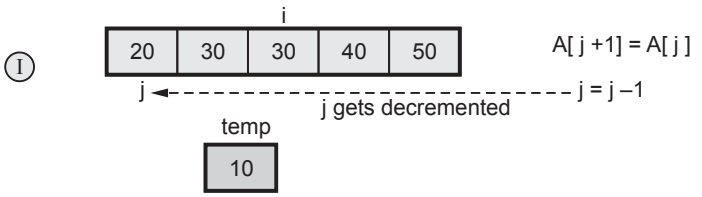


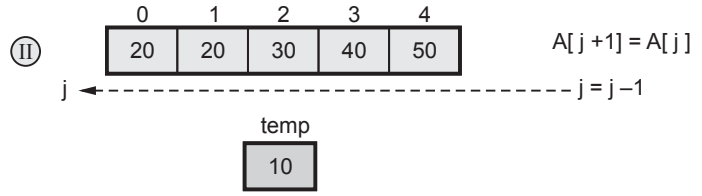
Now since j >= 0 is false, control comes out of while loop.

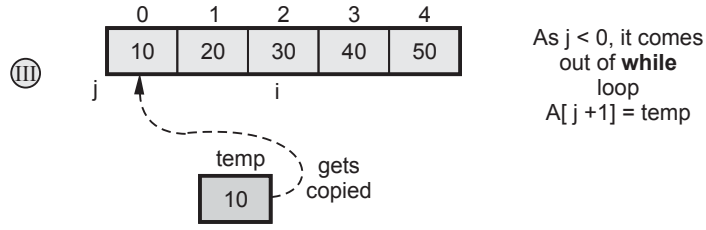


Then list becomes,

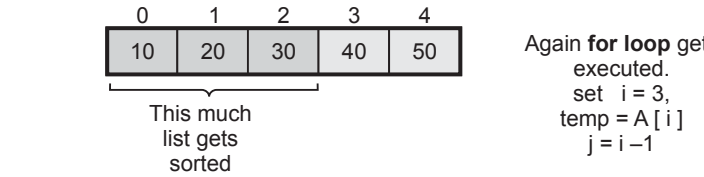



I
i
20 30 30 40 50
A[j +1] = A[j]
j
j = j –1
j gets decremented
temp
10

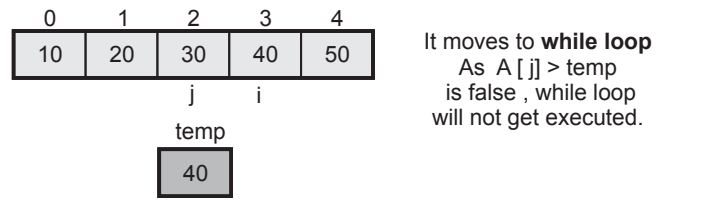

II
0 1 2 3 4
20 20 30 40 50
A[j +1] = A[j]
j
j = j –1
temp
10

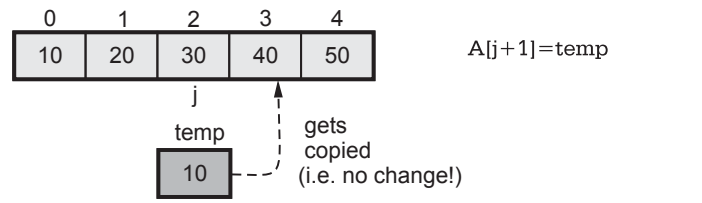

III
0 1 2 3 4
10 20 30 40 50
j
i
As j < 0, it comes out of **while** loop
A[j +1] = temp
temp
gets copied
10

Thus,

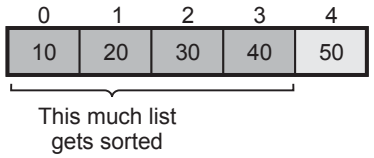

0 1 2 3 4
10 20 30 40 50
This much list gets sorted
Again **for loop** gets executed.
set i = 3,
temp = A [i]
j = i –1

Then,


0 1 2 3 4
10 20 30 40 50
j
i
temp
40
It moves to **while loop**
As A [j] > temp
is false , while loop
will not get executed.

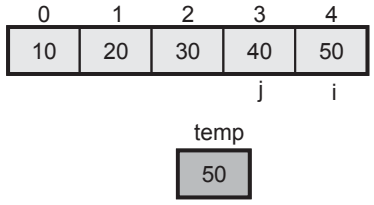

0 1 2 3 4
10 20 30 40 50
j
temp
10
gets copied (i.e. no change!)
A[j+1]=temp

Then,

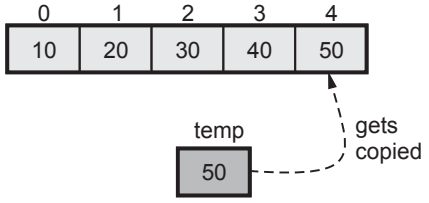


Again for loop gets executed
set i = 4
temp = A [i]
j = i –1

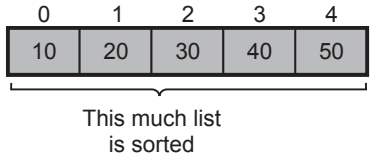
Then,



It moves to **while loop**
As A[j] > temp is false, while loop will not get executed.



A[j+1]=temp



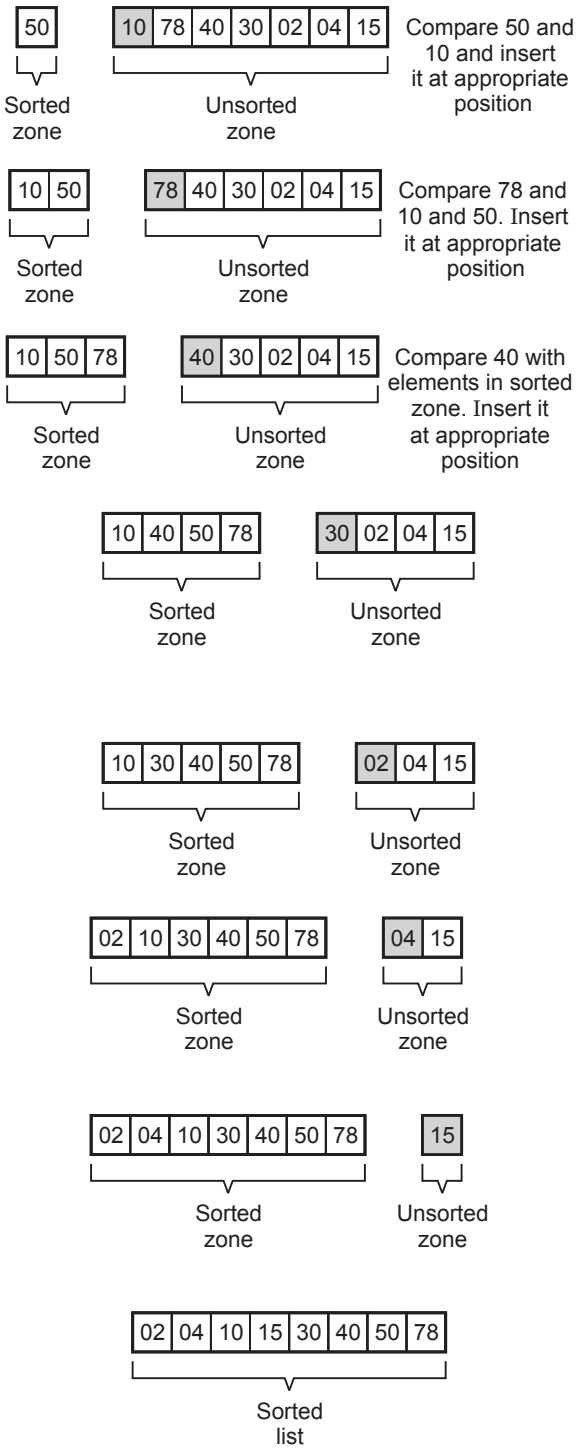
Thus we have scanned the entire list and inserted the elements at corresponding locations. Thus we get the sorted list by insertion sort.

Example 3.5.2 *Sort the following numbers using insertion sort. Show all passes*
50, 10, 78, 40, 30, 02, 04, 15

Solution : Consider the list of elements as -

0	1	2	3	4	5	6	7
50	10	78	40	30	02	04	15

The process starts with first element.



Example 3.5.3 *Compare the insertion sort and selection sort with i) Efficiency ii) Sort stability iii) Passes*

Solution : i) Efficiency : The code for selection sort is as follows

```
for(i=0;i<n-1;i++)
        for(j=i+1;j<n;j++)
        {
            if(ai>aj)
                swap ai and aj
        }
```

In the i^{th} pass, n-i comparisons will be needed to select the smallest element. n-1 passes are required to sort the array. Thus, the number of comparisons needed to sort an array having n elements

$$= (n-1)+(n-2)+...+2+1 = \frac{n(n-1)}{2} = \frac{1}{2}\left(n^2 - n\right)$$

$$\approx O\left(n^2\right)$$

Insertion sort :

```
for(i=1;i<n;i++)
{
        k=a[i];
        for(j=i-1;j>=0&&y<a[i])
            a[j+1]=a[j];
            a[j+1]=1;
}
```

Inner loop is data sensitive. If the input list is presented for sorting is presorted then the test a[i] > y in the inner loop will fail immediately. Thus, only one comparison will be made in each pass.

Thus the total number of comparisons (Best case)

$= n - 1 \approx n$ for large n.

If the numbers to be sorted are initially in descending order then the inner loop will make i iterations in i^{th} pass.

$\therefore$ Total number of comparisons.

$$= 1+2+3+...+(n-1) = \frac{n(n-1)}{2} = \frac{1}{2}\left(n^2 - n\right) \approx n^2 \text{ for large n.}$$

ii) Sort stability : Both the techniques are stable. Both of them have same time complexity.

iii) Passes : Both selection and insertion sort requires n-1 passes.

Insertion sort is more stable than selection sort.

Insertion sort requires less overhead while sorting data. Selection sort requires more number of comparisons. In both the algorithms, elements are read in linear fashion and adjacent elements are compared. Hence identical numbers will maintain their relative sequence even in the sort list.

3.5.3 Selection Sort

Scan the array to find its smallest element and swap it with the first element. Then, starting with the second element scan the entire list to find the smallest element and swap it with the second element. Then starting from the third element the entire list is scanned in order to find the next smallest element. Continuing in this fashion we can sort the entire list.

Generally, on pass i $(0 \leq i \leq n-2)$, the smallest element is searched among last $n-i$ elements and is swapped with A[i]

$$A[0] \leq A[1] \leq \ldots \leq A[i-1] \mid \overbrace{A[i], \ldots, A[k], \ldots, A[n-1]}^{\text{Last } n-i \text{ elements}}$$

A[k] is smallest element

so swap A[i] and A[k]

The list gets sorted after n–1 passes.

Example 1 :

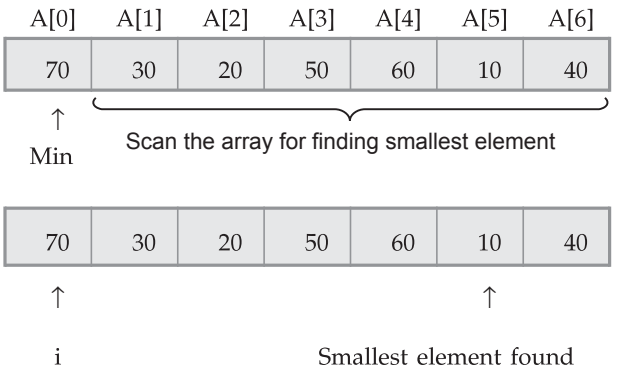
Consider the elements

70, 30, 20, 50, 60, 10, 40

We can store these elements in array A as :

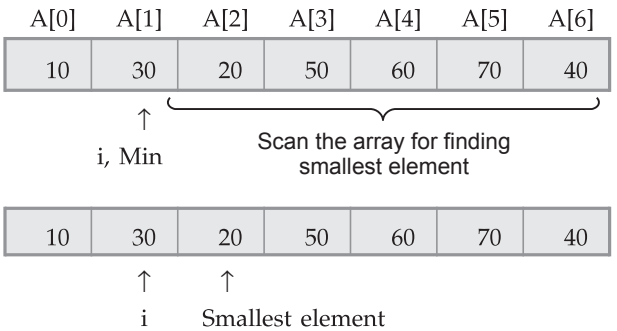
	70	30	20	50	60	10	40
	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
	↑	↑					
Initially set	Min	j					

1st pass :



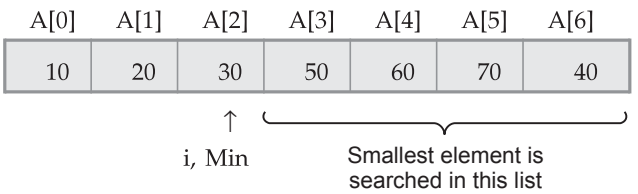
Now swap A[i] with smallest element. Then we get,

2nd pass :



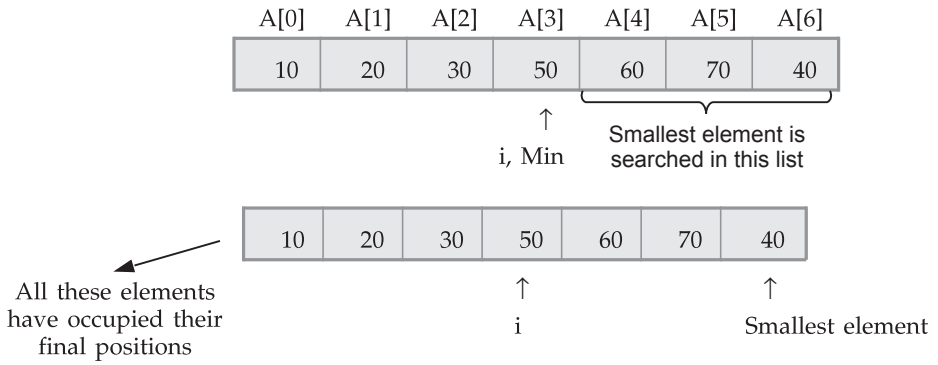
Swap A[i] with smallest element. The array becomes,

3rd pass :



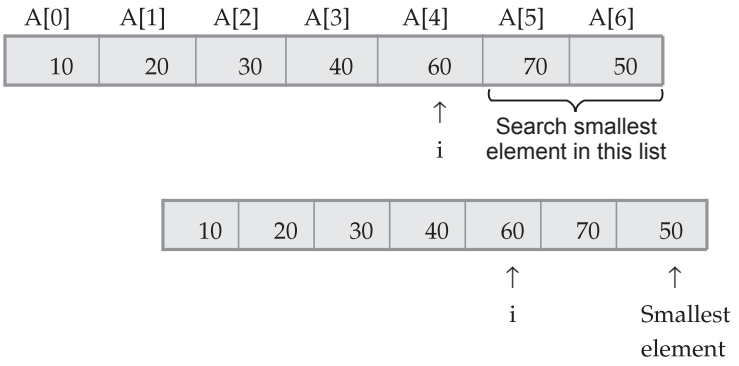
As there is no smallest element than 30 we will increment i pointer.

4th pass :

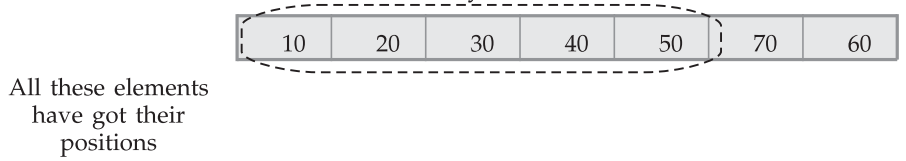


Swap A[i] with smallest element. The array then becomes,

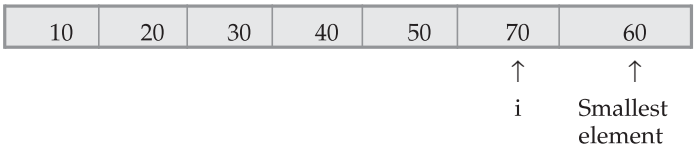
5th pass :



Swap A[i] with smallest element. The array then becomes,



6th pass :



Swap A[i] with smallest element. The array then becomes,

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

This is a sorted array.

Python Program

```
def SelectionSort(arr,n):
    for i in range(n):
        Min = i
        for j in range(i+1,n):
            if(arr[j]<arr[Min]):
                Min = j
        temp=arr[i]
        arr[i]=arr[Min]
        arr[Min]=temp

    print(arr)

print("\n Program For Selection Sort")
print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)

print("Original array is\n")
print(array)

print("\n Sorted Array is")
SelectionSort(array,n)
```

Output

```
Program For Selection Sort
How many elements are there in Array?
5
 Enter element in Array
30
```

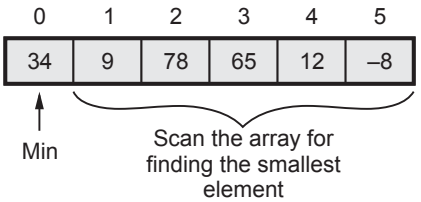
```
 Enter element in Array
50
 Enter element in Array
10
 Enter element in Array
20
 Enter element in Array
40
Original array is
[30, 50, 10, 20, 40]
 Sorted Array is
[10, 20, 30, 40, 50]
>>>
```

Example 3.5.4 *Sort the following and show the status after every pass using selection sort : 34, 9, 78, 65, 12, -8*

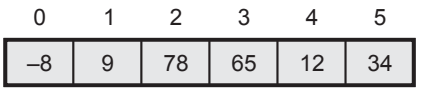
Solution : Let

be the given elements.

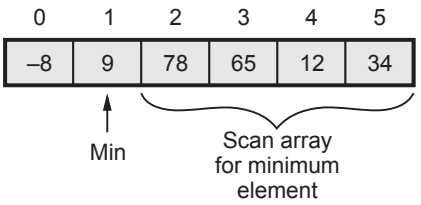
Pass 1 : Consider the elements A[0] as the first element. Assume this as the minimum element.



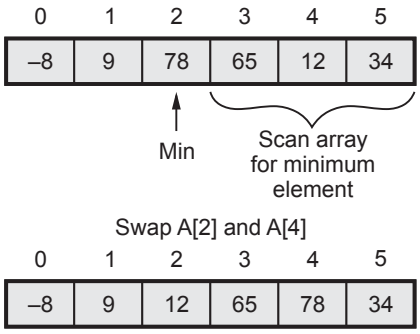
If smallest element is found, swap it with A[0]



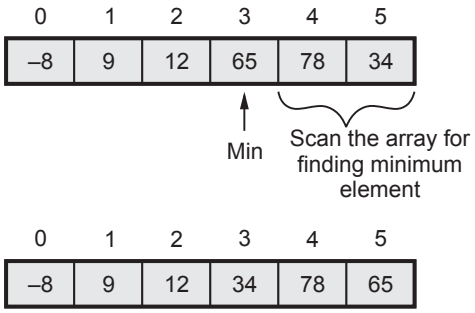
Pass 2 :



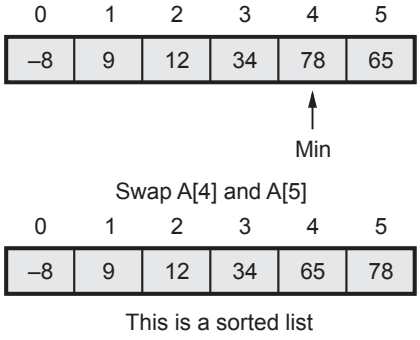
Pass 3 :



Pass 4 :



Pass 5 :



3.5.4 Quick Sort

Quick sort is a sorting algorithm that uses the divide and conquer strategy. In this method division is dynamically carried out. The three steps of quick sort are as follows :

Divide : Split the array into two sub arrays that each element in the left sub array is less than or equal the middle element and each element in the right sub array is greater than the middle element. The splitting of the array into two sub arrays is based on pivot

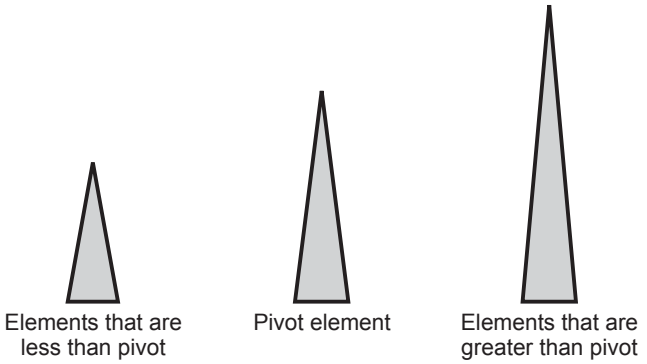


Fig. 3.5.1

element. All the elements that are less than pivot should be in left sub array and all the elements that are more than pivot should be in right sub array.

Conquer : Recursively sort the two sub arrays.

Combine : Combine all the sorted elements in a group to form a list of sorted elements.

Algorithm

The quick sort algorithm is performed using following two important functions -

Quick and *partition*. Let us see them -

```
Algorithm Quick(A[0…n-1],low,high)
//Problem Description : This algorithm performs sorting of
//the elements given in Array A[0…n-1]
//Input: An array A[0…n-1] in which unsorted elements are
//given. The low indicates the leftmost element in the list
//and high indicates the rightmost element in the list
//Output: Creates a sub array which is sorted in ascending
//order
if(low<high)then
//split the array into two sub arrays
m ← partition(A[low…high])// m is mid of the array
Quick(A[low…m-1])
Quick(A[mid+1…high])
```

In above algorithm call to partition algorithm is given. The *partition* performs arrangement of the elements in ascending order. The recursive *quick* routine is for dividing the list in two sub lists. The pseudo code for *Partition* is as given below -

```
Algorithm Partition (A[low…high])
//Problem Description: This algorithm partitions the
//subarray using the first element as pivot element
//Input: A subarray A with low as left most index of the
```

```
//array and high as the rightmost index of the array.
//Output: The partitioning of array A is done and pivot
//occupies its proper position. And the rightmost index of
//the list is returned
pivot ← A[low]
i ← low
j ← high+1
while(i<=j) do
{
 while(A[i]<=pivot) do
  i ← i+1
 while(A[j]>=pivot) do
  j ← j–1;
 if(i<=j) then
              swap(A[i],A[j])//swaps A[i] and A[j]
}
swap(A[low],A[j])//when i crosses j swap A[low] and A[j]
return j//rightmost index of the list
```

The partition function is called to arrange the elements such that all the elements that are less than pivot are at the left side of pivot and all the elements that are greater than pivot are all at the right of pivot. In other words pivot is occupying its proper position and the partitioned list is obtained in an ordered manner.

Analysis

When pivot is chosen such that the array gets divided at the mid then it gives the best case time complexity. The best case time complexity of quick sort is $\mathbf{O(n\log_2 n)}$.

The worst case for quick sort occurs when the pivot is minimum or maximum of all the elements in the list. This can be graphically represented as -

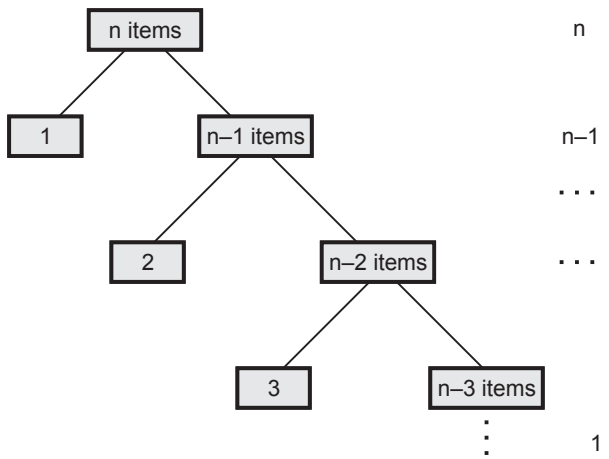


Fig. 3.5.2

This ultimately results in $\mathbf{O(n^2)}$ time complexity. When array elements are randomly distributed then it results in average case time complexity, and it is $\mathbf{O(n\log_2 n)}$.

Example 3.5.5 *Consider following numbers, sort them using quick sort. Show all passes to sort the values in ascending order*

25, 57, 48, 37, 12, 92, 86, 33

Solution : Consider the first element as a pivot element.

25	57	48	37	12	92	86	33
↑	↑						↑
Pivot	i						j

Now, if A[i] < Pivot element then increment i. And if A[j] > Pivot element then decrement j. When we get these above conditions to be false, Swap A[i] and A[j]

25	33	48	37	12	92	86	57
↑	i						j
Pivot							

25	33	48	37	12	92	86	57
↑	i			j			
Pivot							

Swap A[i] and A[j]

Low							High
25	12	48	37	33	92	86	57
	j	i					

As j > i Swap A[Low] and A[j]

After pass 1 :

[12]	25	[48	37	33	92	86	57]
		↑	↑				↑
		Pivot	i				j
12	25	[48	37	33	92	86	57]
					i		j
		Low					
12	25	[48	37	33	92	86	57]
				j	i		

As j > i, we will swap A[j] and A[Low]

After pass 2 :

12 25 [33 37] 48 [92 86 57]

After pass 3 :

12 25 33 37 48 [92 86 57]

Assume 92 to be pivot element

12 25 33 37 48 [92 86 57]

↑ i j

Pivot

12 25 33 37 48 92 86 57

j i

As j > i swap 92 with 57.

After pass 4 :

12 25 33 37 48 57 [86 92]

After pass 5 :

12 25 33 37 48 57 86 92

is a sorted list.

Python Program

```
def Quick(arr,low,high):
    if(low<high):
        m=Partition(arr,low,high)
        Quick(arr,low,m-1)
        Quick(arr,m+1,high)

def Partition(arr,low,high):
    pivot = arr[low]
    i=low+1
    j=high
    flag = False
    while(not flag):
        while(i<=j and arr[i]<=pivot):
            i = i + 1
        while(i<=j and arr[j]>=pivot):
```

```
            j = j - 1

        if(j<i):
            flag = True
        else:
            temp = arr[i]
            arr[i] = arr[j]
            arr[j] = temp

    temp = arr[low]
    arr[low] = arr[j]
    arr[j] = temp
    return j

print("\n Program For Quick Sort")
print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)

print("Original array is\n")
print(array)

Quick(array,0,n-1)
print("\n Sorted Array is")
print(array)
```

Output

```
Program For Quick Sort
How many elements are there in Array?
8
 Enter element in Array
50
 Enter element in Array
30
 Enter element in Array
10
 Enter element in Array
90
 Enter element in Array
80
 Enter element in Array
20
 Enter element in Array
```

```
40
 Enter element in Array
70
Original array is
[50, 30, 10, 90, 80, 20, 40, 70]
 Sorted Array is
[10, 20, 30, 40, 50, 70, 80, 90]
>>>
```

3.5.5 Shell Sort

This method is a improvement over the simple insertion sort. In this method the elements at fixed distance are compared. The distance will then be decremented by some fixed amount and again the comparison will be made. Finally, individual elements will be compared. Let us take some example.

Example : If the original file is

	0	1	2	3	4	5	6	7
X array	25	57	48	37	12	92	86	33

Step 1 : Let us take the distance k = 5

So in the first iteration compare

(x [0], x[5])

(x [1], x [(6])

(x [2], x[7])

(x[3])

(x[4])

i.e. first iteration

After first iteration,

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
25	57	33	37	12	92	86	48

Step 2 : Initially k was 5. Take some d and decrement k by d. Let us take d = 2

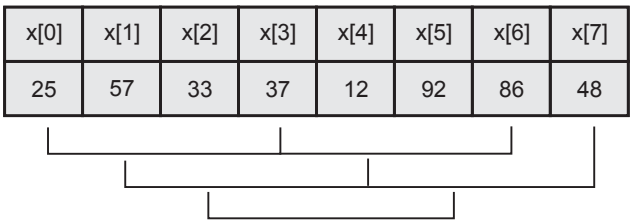
$\therefore$ k = k – d i.e. k = 5 – 2 = 3

So now compare

(x[0], x[3], x[6]), (x[1], x[4], x[7])

(x[2], x[5])

Second iteration



After second iteration

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
25	12	33	37	48	92	86	57

Step 3 : Now k = k – d ∴ k = 3 – 2 = 1

So now compare

(x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7])

This sorting is then done by simple insertion sort. Because simple insertion sort is highly efficient on sorted file. So we get

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
12	25	33	37	48	57	86	92

Python Program

```
def ShellSort(arr,n):
    d = n//2
    while d > 0:
        for i in range(d,n):
            temp = arr[i]
            j = i
            while(j >= d and arr[j-d] >temp):
                arr[j] = arr[j-d]
                j -= d

            arr[j] = temp
        d = d//2

print("\n Program For Shell Sort")
print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
```

```
    item = int(input())
    array.append(item)

print("Original array is\n")
print(array)

ShellSort(array,n)
print("\n Sorted Array is")
print(array)
```

Output

```
Program For Shell Sort
How many elements are there in Array?
5
 Enter element in Array
30
 Enter element in Array
20
 Enter element in Array
10
 Enter element in Array
40
 Enter element in Array
50
Original array is
[30, 20, 10, 40, 50]
 Sorted Array is
[10, 20, 30, 40, 50]
>>>
```

Analysis :

Best Case : The best case in the shell sort is when the array is already sorted in the right order. The number of comparisons is less. In that case the inner loop does not need to any work and a simple comparison will be sufficient to skip the inner sort loop. the other loops gives O(nlogn). The best case of O(n) is reached by using a constant number of increments. Hence the best case time complexity of shell sort is **O(nlogn)**

Worst Case and Average Case : The running time of Shellsort depends on the choice of increment sequence. The problem with Shell's increments is that pairs of increments are not necessarily relatively prime and smaller increments can have little effect. The wort case and average case time complexity is **O(n)**.

3.6 Non-comparison based Sorting Methods

3.6.1 Radix Sort

In this mehod sorting can be done digit by digit and thus all the elements can be sorted.

Example for Radix sort

Consider the unsorted array of 8 elements.

45 37 05 09 06 11 18 27

Step 1 : Now sort the element according to the last digit.

Last digit	0	1	2	3	4	5	6	7	8	9
Elements		11				45, 05	06	37,27	18	09

Now sort this number

Last digit	Element
0	
1	11
2	
3	
4	
5	05,45
6	06
7	27,37
8	18
9	09

Step 2 : Now sort the above array with the help of second last digit.

Second last digit	Element
0	05,06,09
1	11,18
2	27
3	37
4	45
5	
6	
7	
8	
9	

Since the list of element is of two digit that is why, we will stop comparing. Now whatever list we have got (shown in above array) is of sorted elements. Thus finally the sorted list by radix sort method will be

05 06 09 11 18 27 37 45

Example 3.6.1 *Sort the following data in ascending order using Radix Sort :*
25, 06, 45, 60, 140, 50,

Solution :

Step 1 :

Sort the elements according to last digit and sort them.

Last digit	Element
0	50, 60, 140
1	
2	
3	
4	

5	25, 45
6	06
7	
8	
9	

Step 2 :

Sort the elements according to second last digit and sort them.

Second last digit	Element
0	06
1	
2	25
3	
4	45, 140
5	50
6	60
7	
8	

Step 3 :

Sort the elements according to 100^{th} position of the element and sort them.

100^{th} position	Element
0	06, 25, 45, 50, 60
1	140
2	
3	
4	
5	

6	
7	
8	
9	

Thus the sorted list of elements is

06, 25, 45, 50, 60, 140

Algorithm :

1. Read the total number of elements in the array.
2. Store the unsorted elements in the array.
3. Now the simple procedure is to sort the elements by digit by digit.
4. Sort the elements according to the last digit then second last digit and so on.
5. Thus the elements should be sorted for up to the most significant bit.
6. Store the sorted element in the array and print them.
7. Stop.

Python Program

```
def RadixSort(arr):
    MaxElement = max(arr)
    place = 1
    while MaxElement          //place > 0:
        countingSort(arr,place)
        place = place * 10

def countingSort(arr,place):
    n = len(arr)
    result = [0]*n
    count = [0] *10
    #calculating the count of elements based on digits place
    i = 0
    for i in range(n):
        index = arr[i]            // place
        count[index % 10] += 1

    for i in range(1, 10):
        count[i] = count[i]+count[i - 1]
    #placing the elements in sorted order
    i = n - 1
    while i >= 0:
```

```
        index = arr[i]              // place
        result[count[index % 10] - 1] = arr[i]
        count[index % 10] -= 1
        i = i-1
    #placing back the sorted elements in original
    for i in range(0, n):
        array[i] = result[i]

print("\n Program For Radix Sort")
print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)

print("Original array is\n")
print(array)

RadixSort(array)
print("\n Sorted Array is")
print(array)
```

Output

```
Program For Radix Sort
How many elements are there in Array?
6
 Enter element in Array
121
 Enter element in Array
235
 Enter element in Array
55
 Enter element in Array
973
 Enter element in Array
327
 Enter element in Array
179
Original array is
[121, 235, 55, 973, 327, 179]
 Sorted Array is
[55, 121, 179, 235, 327, 973]
>>>
```

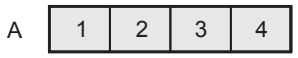
3.6.2 Counting Sort

Concept : Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

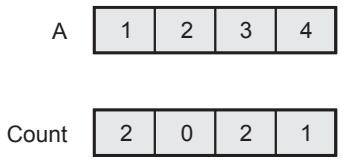
Let us understand this technique with the help of an example

Example 3.6.2 *Apply counting sort for the following numbers to sort in ascending order.*
4, 1, 3, 1, 3

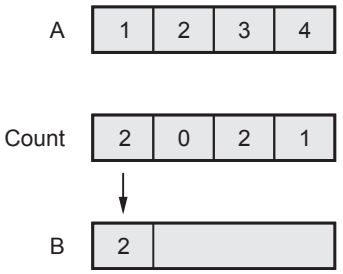
Solution : Step 1 : We will find the min and max values from given array. The min = 1 and max = 4. Hence create an array A from 1 to 4.



Now create another array named **count.** Just count the number of occurrences of each element and store that count in count array at corresponding location of element i.e. element 1 appeared twice, element 2 is not present, element 3 appeared twice and element 4 appeared once.

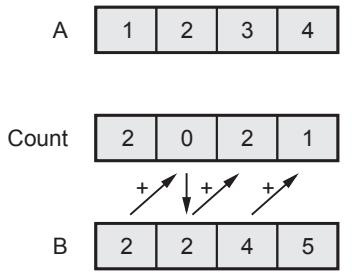


Step 2 : Now we will create another array B in which we will store sum of counts for given index.

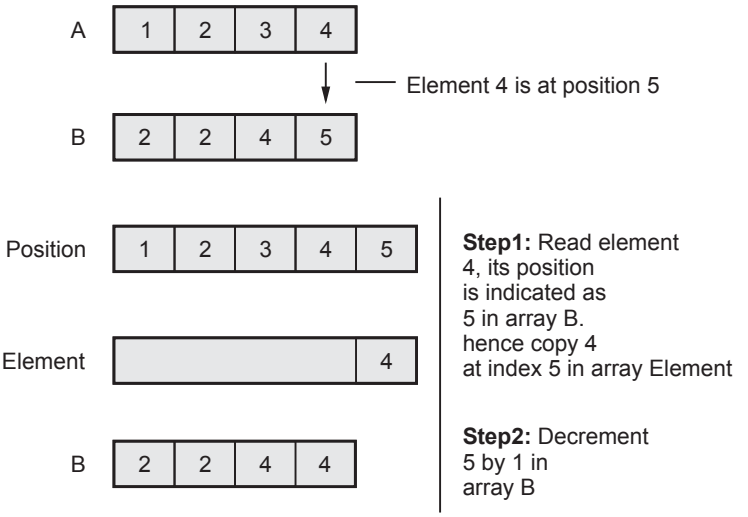


Simply copy first index value of count array to B.

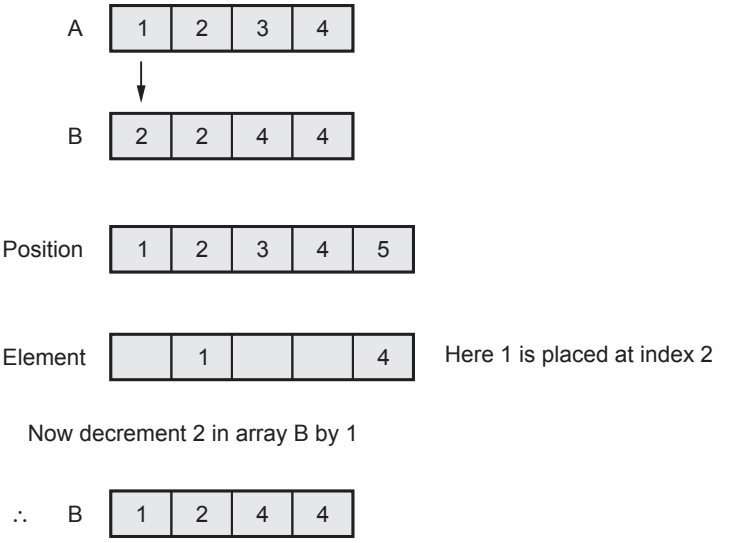
For filling up rest of the elements to B array copy the sum of previous index value of B with current index value of count array.



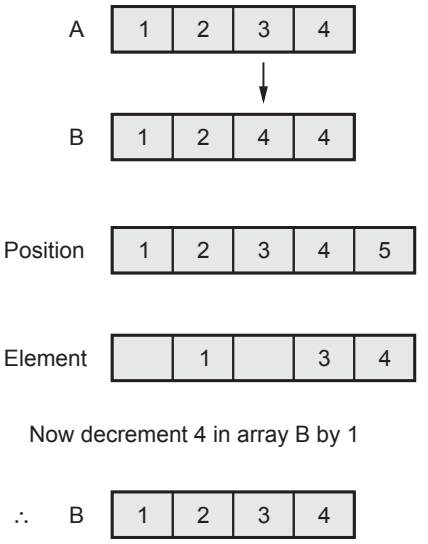
Step 3 : Now consider array A and B for creating two more arrays namely **Position** and **Element.**



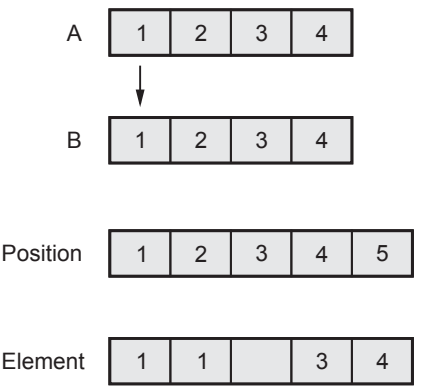
Step 4 : Next element is 1. The position of it is 2.



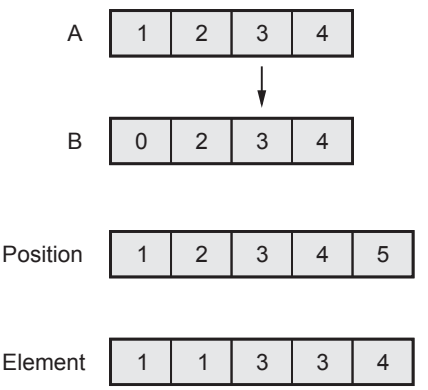
Step 5 : Next element is 3. The position of it is 4 in array B.



Step 6 : Next element is 1. The position of it in array B is 1.



Step 7 : Next element is 3. In array B its position is 3.



Step 8 : Thus we get sorted list in array **Element** as

1	1	3	3	4

Python Program

```
def countingSort(arr):
    n = len(arr)
    result = [0]*n
    count = [0] *10
    #calculating the count of elements
    i = 0
    for i in range(n):
        count[arr[i]] += 1

    for i in range(1, 10):
        count[i] = count[i]+count[i - 1]
    #placing the elements in sorted order
    i = n - 1
    while i >= 0:
        result[count[arr[i]] - 1] = arr[i]
        count[arr[i]] -= 1
        i = i-1
    #placing back the sorted elements in original
    for i in range(0, n):
        array[i] = result[i]

print("\n Program For Counting Sort")
print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)

print("Original array is\n")
print(array)

countingSort(array)
print("\n Sorted Array is")
print(array)
```

Output

```
Program For Counting Sort
How many elements are there in Array?
7
```

```
 Enter element in Array
5
 Enter element in Array
3
 Enter element in Array
5
 Enter element in Array
1
 Enter element in Array
3
 Enter element in Array
5
 Enter element in Array
4
Original array is
[5, 3, 5, 1, 3, 5, 4]

 Sorted Array is
[1, 3, 3, 4, 5, 5, 5]
>>>
```

3.6.3 Bucket Sort

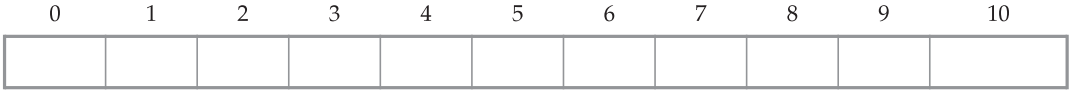
Bucket sort is a sorting technique in which array is partitioned into buckets. Each bucket is then sorted individually, using some other sorting algorithm such as insertion sort.

Algorithm

1. Set up an array of initially empty buckets.
2. Put each element in corresponding bucket.
3. Sort each non empty bucket.
4. Visit the buckets in order and put all the elements into a sequence and print them.

Example 3.6.3 *Sort the elements using bucket sort. 56, 12, 84, 56, 28, 0, -13, 47, 94, 31, 12, -2.*

Solution : We will set up an array as follows



Range –20 to –1 0 to 10 10 to 20 20 to 30 30 to 40 40 to 50 50 to 60 60 to 70 70 to 80 80 to 90 90 to 100

Now we will fill up each bucket by corresponding elements

0	1	2	3	4	5	6	7	8	9	10
–13 –2	0	12 12	28	31	47	56 56			84	94

Now sort each bucket

0	1	2	3	4	5	6	7	8	9	10
–13 –2	0	12 12	28	31	47	56 56			84	94

Print the array by visiting each bucket sequentially.

-13, -2, 0, 12, 12, 28, 31, 47, 56, 56, 84, 94.

This is the sorted list.

Example 3.6.4 *Sort the following elements in ascending order using bucket sort. Show all passes 121, 235, 55, 973, 327, 179*

Solution : We will set up an array as follows -

0 to 100	100 to 200	200 to 300	300 to 400	400 to 500	500 to 600	600 to 700	700 to 800	800 to 900	900 to 1000

Now we will fill up each bucket by corresponding element in the list

55	121, 179	235	237						973
0 to 100	100 to 200	200 to 300	300 to 400	400 to 500	500 to 600	600 to 700	700 to 800	800 to 900	900 to 1000

Now visit each bucket and sort it individually.

Finally read each element of the bucket and place in some array say b[]. The elements from array b are printed to display the sorted list

55	121	179	235	327	973

Python Program

```
def BucketSort(arr):
    bucket = []
    slot = 10
    for i in range(slot):
        bucket.append([])
    #fill the bucket with elements
    for j in arr:
        k = int(slot*j)
        bucket[k].append(j)
    #sort individual bucket
    for i in range(len(array)):
        bucket[i] = sorted(bucket[i])
    #Retrieve sorted elements from bucket to original array
        m = 0
        for i in range(len(arr)):
            for j in range(len(bucket[i])):
                arr[m] = bucket[i][j]
                m = m + 1
        return arr

print("\n Program For Bucket Sort")
print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)

print("Original array is\n")
print(array)

BucketSort(array)
print("\n Sorted Array is")
print(array)
```

Drawbacks

1. For bucket sort the maximum value of the element must be known.
2. We must have to create enough buckets in the memory for every element to place in the array.

Analysis

The best case, worst case and average case time complexity of this algorithm is O(n)

3.7 Comparison of all Sorting Methods and their Complexities

Sorting technique	Best case	Average case	Worst case
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Radix sort	O(nlogn)	O(nlogn)	O(nlogn)
Quick sort	O(nlogn)	O(nlogn)	$O(n^2)$
Heap sort	O(nlogn)	O(nlogn)	O(nlogn)
Shell sort	O(nlogn)	O(n)	O(n)

❑❑❑

Unit - IV

4 Linked List

Syllabus

Introduction to Static and Dynamic Memory Allocation, Linked List: Introduction, of Linked Lists, Realization of linked list using dynamic memory management, operations, Linked List as ADT, Types of Linked List: singly linked, linear and Circular Linked Lists, Doubly Linked List, Doubly Circular Linked List, Primitive Operations on Linked List-Create, Traverse, Search, Insert, Delete, Sort, Concatenate. Polynomial Manipulations-Polynomial addition. Generalized Linked List (GLL) concept, Representation of Polynomial using GLL.

Contents

4.1 Introduction to Static and Dynamic Memory Allocation

- The **static memory management** means allocating or deallocating of memory at compilation time while the word **dynamic** refers to allocation or deallocation of memory while program is running (after compilation).
- The **advantage of dynamic memory management** in handling the linked list is that we can create as many nodes as we desire and if some nodes are not required we can deallocate them. Such a deallocated memory can be reallocated for some other nodes.
- Thus the total memory utilization is possible using dynamic memory allocation.

Sr. no.	Static memory	Dynamic memory
1.	The memory allocation is done at compile time.	Memory allocation is done at dynamic time.
2.	Prior to allocation of memory some fixed amount of it must be decided.	No need to know amount of memory prior to allocation.
3.	Wastage of memory or shortage of memory.	Memory can be allocated as per requirement.
4.	e.g. Array.	e.g. Linked list.

- As shown in Fig. 4.1.1, the program uses the memory which is divided into three parts **static area, local data** and **heap.**
- The static area stores the global data. The stack is for local data area i.e for local variables and heap area is used to allocate and deallocate memory under program's control.
- The **stack** and **heap** area are the part of dynamic memory management. Note that the stack and heap grow towards each other. Their areas are flexible.

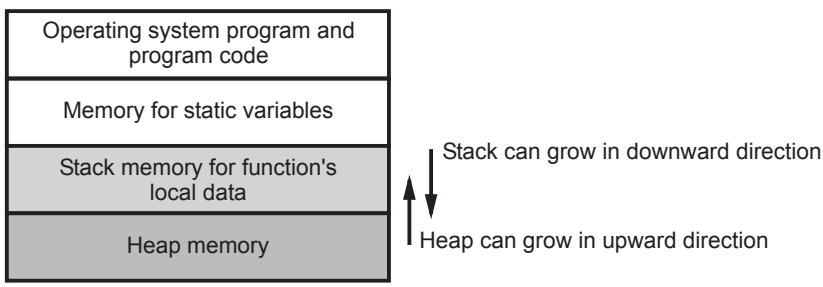


Fig. 4.1.1 Memory model

4.2 Introduction of Linked Lists

A linked list is a set of nodes where each node has two fields ' **data'** and **'link'**. The 'data' field stores actual piece of information and 'link' field is used to point to next node. Basically 'link' field is nothing but **address only**.

Fig. 4.2.1 Structure of node

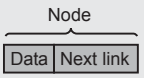
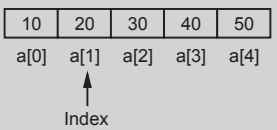
Hence link list of integers 10, 20, 30, 40 is

Fig. 4.2.2

Note that the **'link'** field of **last node consists of NULL** which indicates **end of list**.

4.2.1 Linked List Vs. Array

The comparison between linked list and arrays is as shown below -

Sr. No.	Linked List	Array
1.	The linked list is a collection of nodes and each node is having one data field and next link field. For example 	The array is a collection of similar types of data elements. In arrays the data is always stored at some index of the array. For example 
2.	Any element can be accessed by sequential access only.	Any element can be accessed randomly i.e. with the help of index of the array.
3.	Physically the data can be deleted.	Only logical deletion of the data is possible.
4.	Insertions and deletion of data is easy.	Insertions and deletion of data is difficult.
5.	Memory allocation is dynamic. Hence developer can allocate as well as deallocate the memory. And so no wastage of memory is there.	The memory allocation is static. Hence once the fixed amount of size is declared then that much memory is allocated. Therefore there is a chance of either memory wastage or memory shortage.

4.3 Realization of Linked List using Dynamic Memory Management

For performing the linked list operations we need to allocate or deallocate the memory dynamically. The dynamic memory allocation and deallocation can be done using new and delete operators in C++.

The dynamic memory allocation is done using an operator **new**. The syntax of dynamic memory allocation using **new** is

new data type;

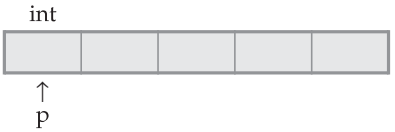
For example :

```
int  *p;
p=new int;
```

We can allocate the memory for more than one element. For instance if we want to allocate memory of size in for 5 elements we can declare.

```
int *p;
p=new int[5];
```

In this case, the system dynamically assigns space for five elements of type *int* and returns a pointer to the first element of the sequence, which is assigned to p. Therefore, now, *p* points to a valid block of memory with space for five elements of type *int*.



The program given below allocates the memory for any number of elements and the memory for those many number of elements get deleted at the end of the program.

The memory can be deallocated using the **delete** operator. The syntax is

delete variable_name;

For example

```
delete p;
```

4.4 Linked List as ADT

In ADT the implementation details are hidden. Hence the ADT will be -

```
Abstract DataType List

{
    Instances : List is a collection of elements which are arranged in a linear manner.
    Operations : Various operations that can be carried out on list are -
    1. Insertion : This operation is for insertion of element in the list.
    2. Deletion : This operation removed the element from the list.
```

```
        3. Searching : Based on the value of the key element the desired element can be
                       searched.
        4. Modification : The value of the specific element can be changed without changing
                          its location.
        5. Display : The list can be displayed in forward or in backward manner.
}
```

4.5 Representation of Linked List

Representation of Linked List using C++

```
class node
{
 public:
        int data;
        node *next;
};
class sll
{
 private:
        node *head;
public:
        void create();
        void print();
        .
        .
        .
};
```

Data members of list

Operations on list

4.6 Primitive Operations on Linked List

Various operations that can be performed on list are -

1. Creation
2. Insertion
3. Deletion
4. Reverse
5. Search
6. Display

1. Creation of linked list

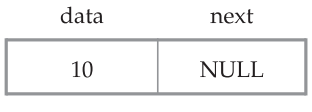
```
void sll :: create()
{
        node *temp, *New;
        int val,flag;
        char ans ='y';
        flag=TRUE;
        do
```

```
    {
        cout<<"\nEnter the data :";
        cin>>val;
        // allocate memory to new node
         New = new node;
        if ( New == NULL )

            cout<<"Unable to allocate memory\n";
        New–> data  = val ;
        New–> next = NULL;
        if( flag==TRUE ) // Executed only for the first time
        {
         head=New;
         temp = head;
         flag=FALSE;
        }
        else
        {
            /*temp last keeps track of the most recently
              created node*/
          temp->next = New;
          temp = New;
        }
        cout<<"\n Do you want to enter more elements?(y/n)";
        ans = getche();
    }while(ans=='y'||ans=='Y');
    cout<<"\nThe Singly Linked List is created\n";
    getch();
    clrscr();
}
```

Creation of linked list (logic explanation part) :

Initially one variable flag is taken whose value is initialized to TRUE (i.e. 1). The purpose of flag is for making a check on creation of first node. That means if flag is TRUE then we have to create head node or first node of linked list. Naturally after creation of first node we will reset the flag (i.e. assign FALSE to flag) Consider that we have entered the element value 10 initially then,

Step 1 :




```
New = new node ;
/* memory gets allocated for
New node */
New → data = Val;
/* value 10 will be put in data field of New */
```

Step 2 :

data	next
10	NULL

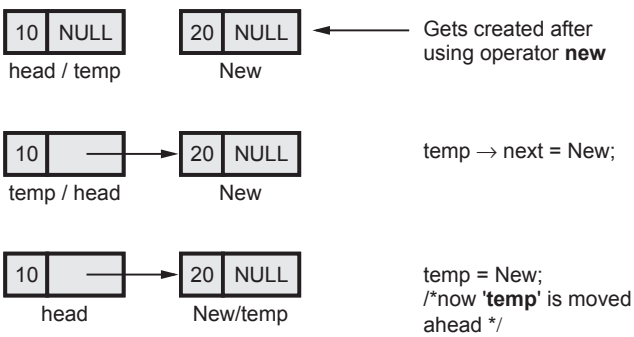
New/

head/temp

```
if (flag = = TRUE)
{
head = New
temp = head;
/* We have also called this node as temp because
head's address will be preserved in 'head' and we can
change 'temp' node as per requirement */
flag = FALSE;
/* After creation of first node flag is reset */
```

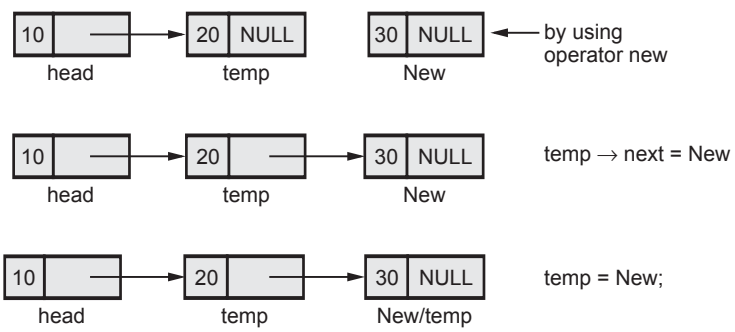
Step 3 :

If **head** node of linked list is created we can further create a linked list by attaching the subsequent nodes. Suppose we want to insert a node with value 20 then,



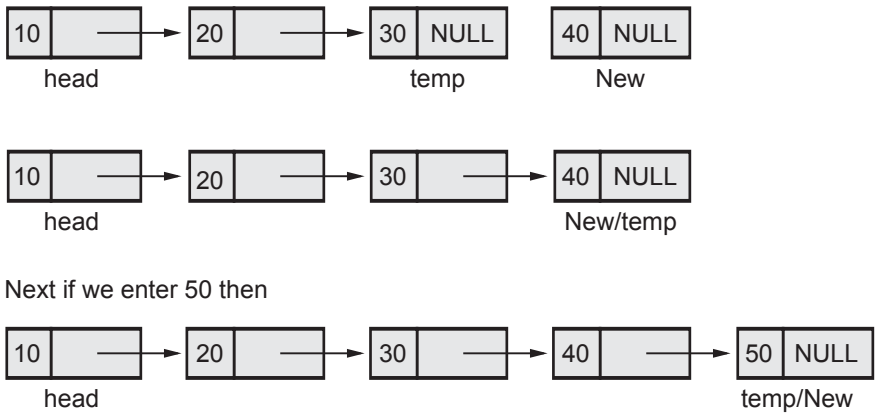
Step 4 :

If user wants to enter more elements then let say for value 30 the scenario will be,



Then for value 40 -

Step 5 :



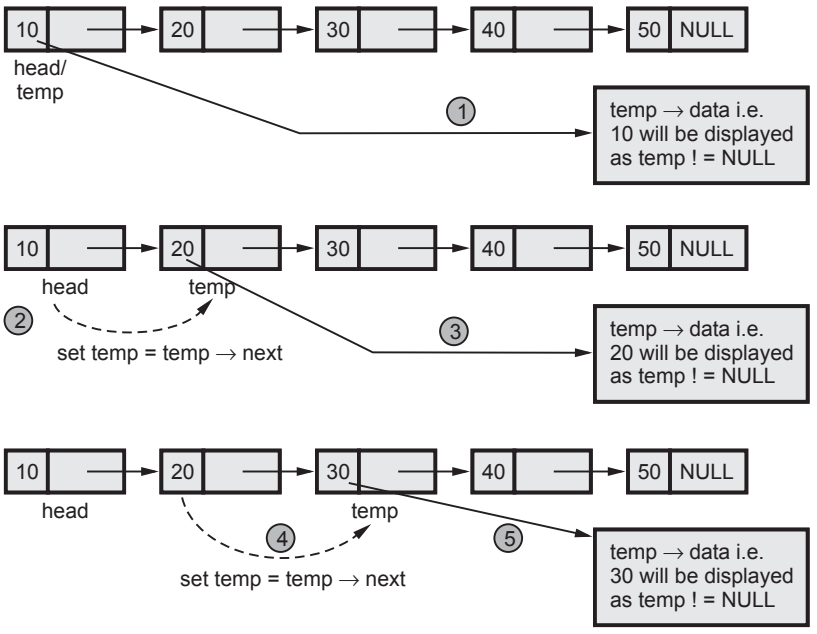
is the final linked list.

2. Display of linked list

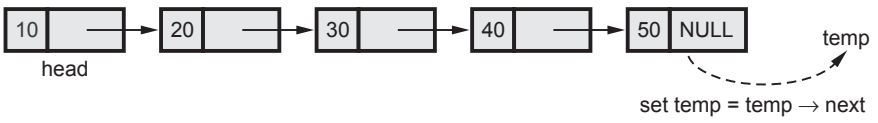
We are passing the address of **head** node to the display routine and calling head as the **'temp'** node. If the linked list is not created then naturally **head = temp** node will be **NULL.** Therefore the message "The list is empty " will be displayed.

```
void sll ::display()
{
        node *temp ;
        temp = head;
        if ( temp == NULL )
        {
            cout<<"\nThe list is empty\n";
            getch(); clrscr();
            return;
        }
        while ( temp != NULL )
        {
              cout<<temp–>data<<"  ";
              temp = temp –> next;
        }
        getch();

}
```

If we have created some linked list like this then -



Continuing in this fashion we can display remaining nodes 40, 50. When



As now value of temp becomes NULL we will come out of **while** loop. As a result of such display routine we will get,

10 → 20 → 30 → 40 → 50 → NULL

will be printed on console.

3. Insertion of any element at anywhere in the linked list

There are three possible cases when we want to insert an element in the linked list -

a) Insertion of a node as a head node

b) Insertion of a node as a last node

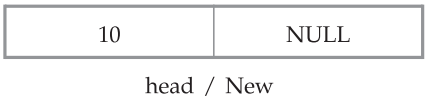
c) Insertion of a node after some node.

We will see the case a) first -

```
void sll:: insert_head()
{
node *New,*temp;
 New=new node;
cout<<"\n Enter The element which you want to insert";
 cin>>New->data;
 if(head==NULL)
            head=New;
else
 {
temp=head;
New->next=temp;
head=New;
}
}
```

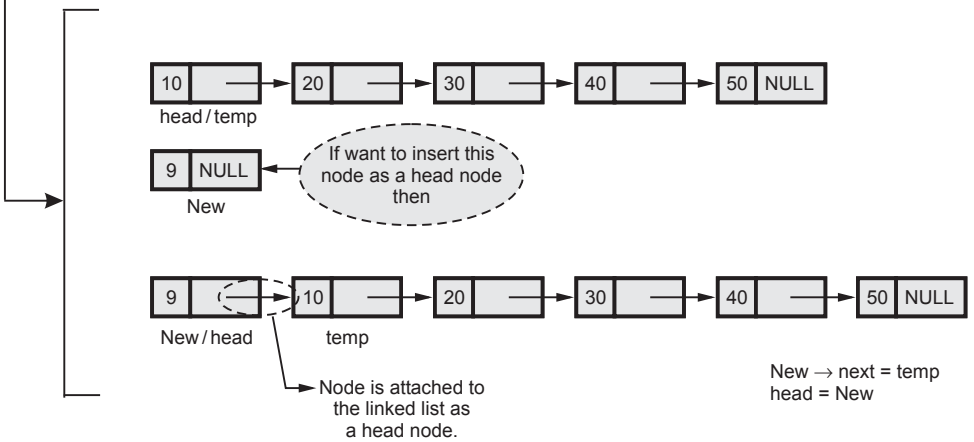
There is no node in the linked list. That means the linked list is empty

If there is no node in the linked list then value of head is NULL. At that time if we want to insert 10 then




```
Cin >>New → data
if (head = = NULL)
head = New;
```

Otherwise suppose linked list is already created like this

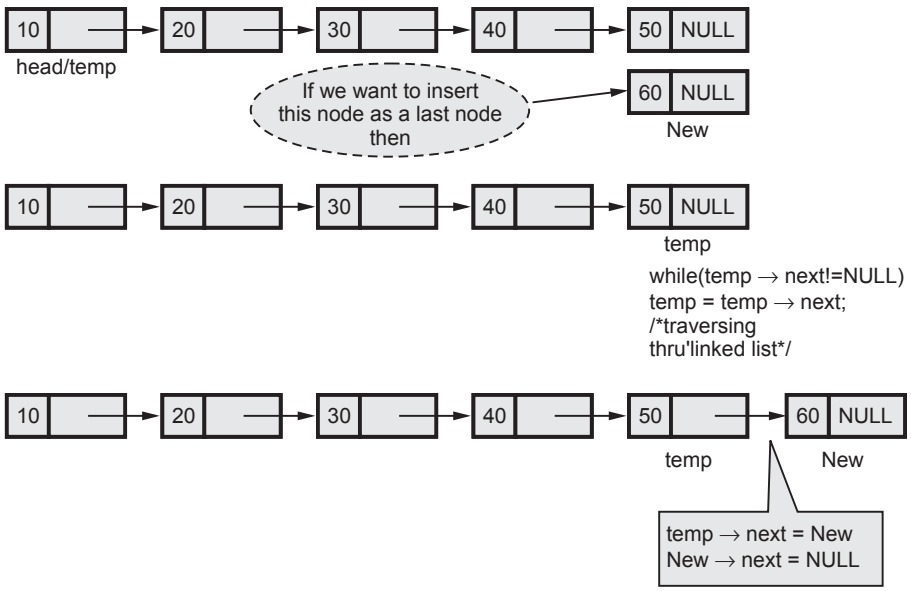


Now we will insert a node at the end -

```
void sll::insert_last()
{
 node *New,*temp;
 cout<<"\nEnter The element which you want to insert";
 cin>>New->data;
 if(head==NULL)
       head=New;
 else
 {
       temp=head;
       while(temp->next!=NULL)
             temp=temp->next;
       temp->next=New;
       New->next=NULL;
 }
}
```

Finding the end of the linked list. Then **temp** will be a last node.

To attach a node at the end of linked list assume that we have already created a linked list like this -

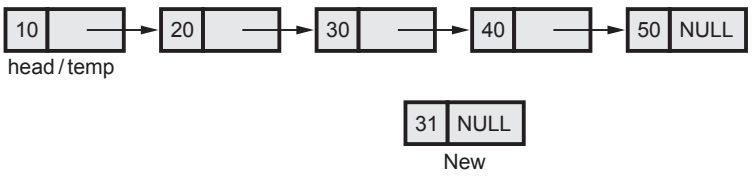


Now we will insert a new node after some node at intermediate position

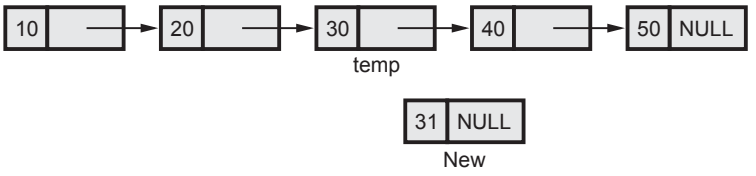
```
void sll:: insert_after()
{
 int key;
 node  *temp,*New;
 New= new node;
```

```
cout<<"\n Enter The element which you want to insert";
cin>>New->data;
if(head==NULL)
{
 head=New;
}
else
{
 cout<<"\n Enter The element after which you want to insert the node";
 cin>>key;
 temp=head;
 do
 {
      if(temp->data==key)
      {
          New->next=temp->next;
          temp->next=New;
          break;
       }
   else
        temp=temp->next;
 }while(temp!=NULL);
}
}
```

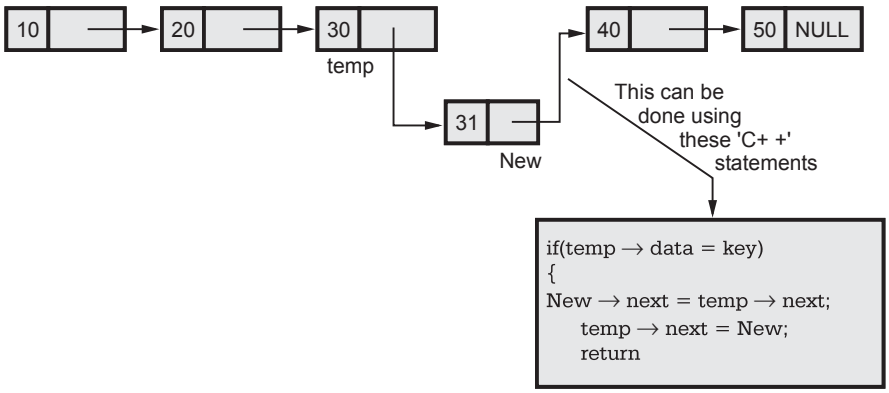
If we want to insert 31 in the linked list which is already created. We want to insert this 31 after node containing 30 then



As we will search for the value 30, key = 30.



Then,



Thus desired node gets inserted at desired position.

4. Deletion of any element from the linked list

```
void sll:: dele()
{
        node *temp, *prev ;
        int  key;
        temp=head;
        clrscr();
        cout<<"\nEnter the data of the node you want to delete: ";
        cin>>key;
        while(temp!=NULL)
        {
          if(temp–>data==key)
                break;
          prev=temp;
          temp=temp–>next;
  }
  if(temp==NULL)
  cout<<"\nNode not found";
  else
  {
          if(temp==head) //first node
                head=temp–>next;
          else
                prev–>next=temp–>next;  //intermediate or end node
        delete temp;
        cout<<"\nThe Element is deleted\n";
  }
        getch();

}
```

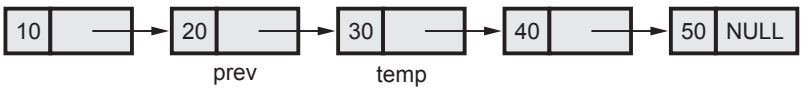
Firstly Node to be deleted is searched in the linked list

Once the node to be deleted is found then get the previous node of that node in variable **prev**

If we want to delete the head node then set its adjacent node as a new head node and then release the memory

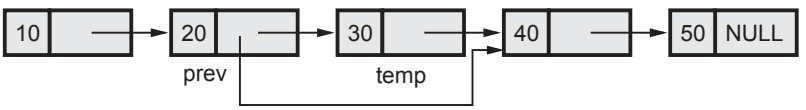
Suppose we have,

Suppose we want to delete node 30. Then we will search the node containing 30. Mark the node to be deleted as **temp**. Then we will obtain previous node of **temp** Mark previous node as **prev**

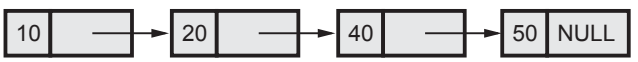


Then,

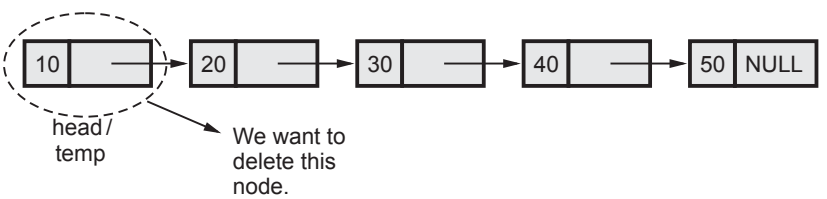
```
prev → next = temp → next
```



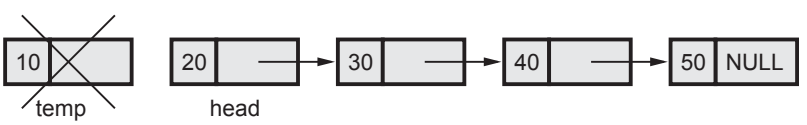
Now we will **delete** the **temp** node. Then the linked list will be



Another case is, if we want to delete a head node then -



This can be done using following statements




```
head = temp → next;
delete temp;
```

5. Searching of desired element in the linked list

The search function is for searching the node containing desired value. We pass the head of the linked list to this routine so that the complete linked list can be searched from the beginning.

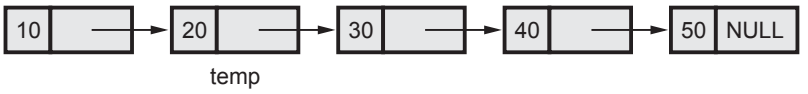
```
void sll::search(int key)
{
 node *temp;
 int found;
 temp = head;
 if ( temp == NULL )
 {
  cout<<"The Linked List is empty\n";
  getch();
  clrscr();
 }
 found = FALSE;
 while ( temp != NULL && found==FALSE)
 {
  if ( temp–>data != key)
    temp = temp –> next;
  else
    found = TRUE;
 }
 if ( found==TRUE )
 {
  cout<<"\nThe Element is present in the list\n";
  getch();
 }
 else
 {
  cout<<"The Element is not present in the list\n";
  getch();
 }
}
```

If node containing desired data is obtained in the linked list then set **found** variable to **TURE**

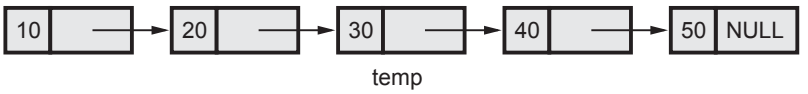
Consider that we have created a linked list as

Suppose key = 30 i.e. we want a node conaining value 30 then compare **temp** → **data** and **key** value. If there is no match then we will mark next node as **temp**.

Is temp → data $\stackrel{?}{=}$ key No



Is temp → data $\stackrel{?}{=}$ key No



Is temp → data $\stackrel{?}{=}$ key Yes

Hence print the message **"The Element is present in the list"**

Thus in search operation the entire list can be scanned in search of desired node. And still, if required node is not obtained then we have to print the message.

"The Element is not present in the list"

Let us see the complete program for it -

'C++' Program

```
/*************************************************************************
 Demonstration Program to perform various operations on
 singly link lists.
*************************************************************************/

// List of include files

#include<iostream.h>
#include <conio.h>
#define TRUE 1
#define FALSE 0
// class

class sll
{
private:
 struct node
 {
       int data;
       structnode  *next;
 }*head;
public:
sll();
```

```
void create();
void display();
void search(int key);
void insert_head();
void insert_after();
void insert_last();
void dele();
~sll();
};

/*
-------------------------------------------------------------------------
The  Constructor defined
-------------------------------------------------------------------------
*/
sll::sll()
 {
  head=NULL;//initialize head to NULL
 }
/*
-------------------------------------------------------------------------
        The Destructor defined
-------------------------------------------------------------------------
*/
sll::~sll()
{
  node *temp,*temp1;
  temp=head->next;
  delete head;
  while(temp!=NULL) //free the memory allocated
  {
   temp1=temp->next;
   delete temp;
   temp=temp1;
  }
}
/*
-------------------------------------------------------------------------
        The Create function
-------------------------------------------------------------------------
*/
void sll :: create()
{

        node *temp, *New;
        int val,flag;
        char ans ='y';
```

```
        flag=TRUE;
        do
        {
                cout<<"\nEnter the data :";
                cin>>val;
                // allocate memory to new node
                 New = new node;
                if ( New == NULL )
                        cout<<"Unable to allocate memory\n";
                New–> data  = val ;
                New–> next = NULL;
                if( flag==TRUE ) // Executed only for the first time
                {
                 head=New;
                 temp = head;
                 flag=FALSE;
                }
                else
                {
                        /*temp last keeps track of the most recently
                          created node*/
                   temp–>next = New;
                   temp = New;
                }
                cout<<"\n Do you want to enter more elements?(y/n)";
                ans = getche();
        }while(ans=='y'||ans=='Y');
        cout<<"\nThe Singly Linked List is created\n";
        getch();
        clrscr();
}

/*
-------------------------------------------------------------------------
        The display function
-------------------------------------------------------------------------
*/
void sll ::display()
{
        node *temp ;
        temp = head;
        if ( temp == NULL )
        {
             cout<<"\nThe list is empty\n";


             getch(); clrscr();
```

```
            return;
        }
        while ( temp != NULL )
        {
            cout<<temp–>data<<"  ";
            temp = temp –> next;
        }
        getch();

}

void sll::search(int key)
{
 node *temp;
 int found;
 temp = head;
 if ( temp == NULL )
 {
  cout<<"The Linked List is empty\n";
  getch();
  clrscr();
 }
 found = FALSE;
 while ( temp != NULL && found==FALSE)
 {
  if ( temp–>data != key)
    temp = temp –> next;
  else
    found = TRUE;
 }
 if ( found==TRUE )
 {
  cout<<"\nThe Element is present in the list\n";
  getch();
 }
 else
 {
  cout<<"The Element is not present in the list\n";
  getch();
 }
}

/*
--------------------------------------------------------------------
        The dele function
--------------------------------------------------------------------
*/
```

```
void sll:: dele()
{
        node *temp, *prev ;
        int  key;
        temp=head;
        clrscr();
        cout<<"\nEnter the data of the node you want to delete: ";
        cin>>key;
        while(temp!=NULL)
        {
          if(temp–>data==key)//traverse till required node to delete
          break;              //is found
          prev=temp;
          temp=temp–>next;
  }
  if(temp==NULL)
  cout<<"\nNode not found";
  else
  {
          if(temp==head) //first node
                head=temp–>next;
          else
                prev–>next=temp–>next;  //intermediate or end node
        delete temp;
        cout<<"\nThe Element is deleted\n";
        }

        getch();

}

/*
---------------------------------------------------------------------
          Function to insert at end
---------------------------------------------------------------------
*/
void sll::insert_last()
{
 node *New,*temp;
 cout<<"\nEnter The element which you want to insert";
 cin>>New–>data;
 if(head==NULL)
        head=New;
 else
 {
        temp=head;
```

```
        while(temp->next!=NULL)
                temp=temp->next;
        temp->next=New;
        New->next=NULL;
 }
}

/*
---------------------------------------------------------------------
          Function to insert after a node
---------------------------------------------------------------------
*/
void sll:: insert_after()
{
 int key;
 node  *temp,*New;
 New= new node;
 cout<<"\n Enter The element which you want to insert";
 cin>>New->data;
 if(head==NULL)
 {
  head=New;
 }
 else
 {
  cout<<"\n Enter The element after which you want to insert the node";
  cin>>key;
  temp=head;
  do
  {
        if(temp->data==key)
        {
             New->next=temp->next;
             temp->next=New;
             break;
         }
     else
          temp=temp->next;
  }while(temp!=NULL);
 }
}
/*
---------------------------------------------------------------------
Function to insert at the beginning
---------------------------------------------------------------------
*/
```

```
void sll:: insert_head()
{
 node *New,*temp;
 New=new node;
 cout<<"\n Enter The element which you want to insert";
 cin>>New–>data;
 if(head==NULL)
        head=New;
 else
 {
  temp=head;
  New–>next=temp;
  head=New;
 }
}
/*
---------------------------------------------------------------------
        The main function
        Input : None
        Output: None
        Parameter Passing Method : None
---------------------------------------------------------------------
*/
void main()
{
        sll s;
        int choice,val,ch1;
        char ans = 'y';
        do
        {
        clrscr();
        cout<<"\nProgram to Perform Various operations on Linked List";
        cout<<"\n1.Create";
        cout<<"\n2.Display";
        cout<<"\n3.Search";
        cout<<"\n4.Insert an element in a list";
        cout<<"\n5.Delete an element from list";
        cout<<"\n6.Quit";
        cout<<"\nEnter Your Choice ( 1–6) ";
        cin>>choice;
        switch( choice)
         {
                case 1:   s.create();
                          break;

                case 2:   s.display();
                          break;
```

```
            case 3:  cout<<"Enter the element you want to search";
                     cin>>val;
                     s.search(val);
                     break;
            case 4:  clrscr();
                     cout<<"\nThe list is:\n";
                     s.display();
                     cout<<"\nMenu";
                     cout<<"\n1.Insert at beginning\n2.Insert after";
                              cout<<"\n3.Insert at end";
                              cout<<"\nEnter your choice";
                              cin>>ch1;
                              switch(ch1)
                              {
                              case 1:s.insert_head();
                                                    break;
                              case 2:s.insert_after();
                                                    break;
                              case 3:s.insert_last();
                                                    break;
                              default:cout<<"\nInvalid choice";
                              }
                          break;
      case 5: s.dele();
                          break;
      default: cout<<"\nInvalid choice";
      }
      cout<<"\nContinue?";
      cin>>ans;
      }while(ans=='y'||ans=='Y');
      getch();
      return;
}
```

Output

```
Program to Perform Various operations on Linked List
1.Create
2.Display
3.Search
4.Insert an element in a list
5.Delete an element from list
6.Quit
Enter Your Choice ( 1–6) 1

Enter the data :10

 Do you want to enter more elements?(y/n)y
Enter the data :20
```

```
 Do you want to enter more elements?(y/n)y
Enter the data :30

 Do you want to enter more elements?(y/n)y
Enter the data :40

 Do you want to enter more elements?(y/n)y
Enter the data :50

 Do you want to enter more elements?(y/n)n
The Singly Linked List is created
Continue?
Program to Perform Various operations on Linked List
1.Create
2.Display
3.Search
4.Insert an element in a list
5.Delete an element from list
6.Quit
Enter Your Choice ( 1–6) 2
10  20  30  40  50
Continue?

Program to Perform Various operations on Linked List
1.Create
2.Display
3.Search
4.Insert an element in a list
5.Delete an element from list
6.Quit
Enter Your Choice ( 1–6) 3
Enter the element you want to search 30

The Element is present in the list
```

4.6.1 Programming Examples based on Linked List Operations

Example 4.6.1 *Given a singly linked list, write a function swap to swap every two nodes e.g. 1->2->3->4->5->6 should become 2->1->4->3->6->5.*

Solution : Assumption

1. The linked is created using create routine.
2. The display routine is written to display the nodes of the linked list.

3. The swap routine for swapping two nodes is as follows -

```
void swap(node **head)
{

   // If linked list is empty or there is only one node in list
   if (*head == NULL || (*head)->next == NULL)
      return;

   node *prev = *head;//maintaining the previous node
   node *current = (*head)->next;//and current node.
       //Thus two nodes are focused at a time

   *head = current;  // Change head before proceeeding

   // Traverse the list
   while (true)
   {
      node *temp = current->next;
      current->next = prev; // Change next of current as previous node


      if (temp == NULL || temp->next == NULL)
      {
         prev->next = temp;
         break;
      }

      // Change next of previous to next of temp
      prev->next = temp->next;

      // Update previous and current nodes
      prev = temp;
      current = prev->next;
   }
}
```

Example 4.6.2 *Write a C program to implement insertion to the immediate left of kth node in singly linked list.*

Solution :

```
#include<iostream.h>
//using namespace std;
#define TRUE 1
#define FALSE 0
// class definition

class sll
```

```
{
private:
        struct node
        {
                int data;
                structnode  *next;
        }*head;
public:
        sll();
        void create();
        void display();
        void insert(int k);
        ~sll();
};

/*
--------------------------------------------------------------------------------
The  Constructor defined
--------------------------------------------------------------------------------
*/
sll::sll()
{
        head = NULL;//initialize head to NULL
}
/*
--------------------------------------------------------------------------------
The Destructor defined
--------------------------------------------------------------------------------
*/
sll::~sll()
{
        node *temp, *temp1;
        temp = head->next;
        delete head;
        while (temp != NULL) //free the memory allocated
        {
                temp1 = temp->next;
                delete temp;
                temp = temp1;
        }
}
/*
--------------------------------------------------------------------------------
The Create function
--------------------------------------------------------------------------------
*/
void sll::create()
```

```
{

        node *temp, *New;
        temp = NULL;
        int val, flag;
        char ans = 'y';
        flag = TRUE;
        do
        {
                cout << "\nEnter the data :";
                cin >> val;
                // allocate memory to new node
                New = new node;
                if (New == NULL)
                        cout << "Unable to allocate memory\n";
                New-> data = val;
                New-> next = NULL;
                if (flag == TRUE) // Executed only for the first time
                {
                        head = New;
                        temp = head;
                        flag = FALSE;
                }
                else
                {
                        /*temp last keeps track of the most recently
                        created node*/
                        temp->next = New;
                        temp = New;
                }
                cout << "\n Do you want to enter more elements?(y/n)";
                cin >> ans;
        } while (ans == 'y' || ans == 'Y');
        cout << "\nThe Singly Linked List is created\n";

}

/*
------------------------------------------------------------------------
The display function
------------------------------------------------------------------------
*/
void sll::display()
{
        node *temp;
        temp = head;
        if (temp == NULL)
```

```
        {
                cout << "\nThe list is empty\n";
                return;
        }
        while (temp != NULL)
        {
                cout << temp->data << "  ";
                temp = temp -> next;
        }
}

void sll::insert(int k)
{
        node *temp,*prev_node,*New;
        int count;
        temp = head;
        prev_node = head;
        count = 1;
        if (temp == NULL)
        {
                cout << "The Linked List is empty\n";
        }
        while (temp != NULL)
        {
                if(count==k)
                        break;
                else
                {
                //keeping track of previous node
                        prev_node = temp;
                        temp = temp->next;
                        count++;
                }
        }
        if (count==k)
        {
                New = new node;
                cout << "\n Enter the new node: ";
                cin >> New->data;
                prev_node->next = New;
                New->next = temp;
                cout << "\n The element is inserted in the linked list!!!";
        }
        else
        {
                cout << "The Element is not present in the list\n";
```

```
        }
}

void main()
{
        sll s;
        int choice, val;
        char ans = 'y';
        do
        {
                cout << "\nProgram to Perform Various operations on Linked List";
                cout << "\n1.Create";
                cout << "\n2.Display";
                cout << "\n3.Insert left to Kth node ";
                cout << "\n4.Quit";
                cout << "\nEnter Your Choice ( 1-4) ";
                cin >> choice;
                switch (choice)
                {
                case 1:       s.create();
                        break;
                case 2:       s.display();
                        break;
                case 3:       cout << "Enter the number of a node";
                        cin >> val;
                        s.insert(val);
                        break;
                default: cout << "\nInvalid choice";
                }
                cout << "\nContinue?";
                cin >> ans;
        } while (ans == 'y' || ans == 'Y');
        return;
}
```

Example 4.6.3 *Given an ordered linked list whose node represented by 'key' as information and next as 'link' field. Write C program to implement deleting number of nodes(consecutive) whose 'key' values are greater than or equal to 'Kmin' and less than 'Kmax'*

Solution :

i) The Pseudo code for deleting elements less than or equal to Kmax is as follows -

```
void sll::delet()
{
        node *temp,*NextNode;
        int Kmax;
        temp = head;
```

```
        cout << "\n Enter the node value as Kmin";
        cin >> Kmax;
        if (temp == NULL)
        {
                cout << "The Linked List is empty\n";
        }
        while (temp != NULL)
        {
                if (Kmax == temp->data)
                        break;
                else
                {
                        NextNode = temp->next;
                        temp->next = NULL;
                        delete(temp);
                        temp = NextNode;
                }
        }
        NextNode = temp->next;
        temp->next = NULL;
        delete(temp);
        temp = NextNode;
        head = temp;
}
```

ii) The Pseudo code for deleting elements greater than or equal to Kmin is as follows -

```
void sll::delet()
{
        node *temp,*NextNode;
        int Kmin;
        temp = head;
        cout << "\n Enter the node value as Kmin";
        cin >> Kmin;
        if (temp == NULL)
        {
                cout << "The Linked List is empty\n";
        }
        while(Kmin != temp->data)
        {
                if(Kmin==temp->data)
                        break;
                else
                {
                        temp = temp->next;
                }
        }
        while (temp != NULL)
```

```
        {
                NextNode = temp->next;
                temp->next = NULL;
                delete(temp);
                temp = NextNode;
        }
}
```

Example 4.6.4 *Write an algorithm to count the number of nodes between given two nodes in a linked list.*

Solution :

```
void count_between_nodes(node *temp1, node *temp2)
{
  int count=0;
  //temp1 represents the starting node
 //temp2 represents the end node
 //we have to count number of nodes between temp1 and temp2
 while(temp1->next!=temp2)
   count++;
 printf("\n Total number of nodes between % and %d is %d ", temp1->data, temp2->data,
count);
}
```

Example 4.6.5 *Write an algorithm to find the location of an element in the given linked list. Is the binary search will be suitable for this search ? Explain the reason.*

Solution :

```
void search_element(node *head,int key)
 {
   node *Temp;
   int count=1;
   //head is a starting node of the linked list
   //key is the element that is to be searched in the linked list.
   Temp=head;
    while(Temp->next!=NULL)
    {
      if(temp->data==key)
      {
        printf("\n The element is present at location %d",count);
        break;
      }
      count++;
      Temp=Temp->next;
   }
}
```

The binary search can be suitable for this algorithm only if the elements in the linked list are arranged in sorted order. Otherwise this method will not be suitable because sorting the linked list is not efficient as it requires the swapping of the pointers.

Example 4.6.6 *Write a C++ function to perform the merging of two linked lists.*

Solution :

```
node *merge(node *temp1,node *temp2)
{
        node *prev_node1,*next_node2,*head;
        if(temp1==NULL)
        {
                temp1=temp2;
                head=temp2;
        }
if(temp1->data<temp2->data)
                head=temp1;
else
                head=temp2;
while(temp1!=NULL && temp2!=NULL)
{
/*while data of 1st list is smaller then traverse*/
                while((temp1->data<temp2->data) && (temp1!=NULL))
                {
                        prev_node1=temp1; /*store prev node of Sll1 */
                        temp1=temp1->next;
                }
                if(temp1==NULL)
                        break;
        /*when temp1's data>temp1 it will come out
        Of while loop so adjust links with temp1's prev node*/
                prev_node1->next=temp2;
                next_node2=temp2->next; /*store next node of Sll2*/
                temp2->next=temp1;
                temp1=temp2;
                temp2=next_node2;
}
if(temp1==NULL&&temp2!=NULL) /*attach rem nodes of Sll2*/
{
                while(temp2!=NULL)
                {
                        prev_node1->next=temp2;
                        prev_node1=temp2;
                        temp2=temp2->next;
                }
}
return head;
}
```

Example 4.6.7 *Write a C function to concatenate two singly linked list.*

Solution :

```
void concat(node *head1,node *head2)
{
        node *temp1,*temp2;
        temp1=head1;
        temp2=head2;
        while(temp1->next!=NULL)
                temp1=temp1->next;/*searching end of first list*/
        temp1->next=temp2;/*attaching head of the second list*/
        printf("\n The concatenated list is ...\n");
        temp1=head1;
        while(temp1!=NULL)
        { /*printing the concatenated list*/
                printf("  %d",temp1->Data);
                temp1=temp1->next;
        }
}
```

Example 4.6.8 *Write a C++ code to recursive routine to erase a linked list (delete all node from the linked list).*

Solution :

```
node sll::*list_free(struct node *temp)
{
 if(temp->next!=NULL)
 {
        temp1=temp->next;/*temp1 is declared globally*/
        temp->next=NULL;
        delete temp;
        list_free(temp1);/*recursive call*/
 }
 temp=NULL;
 return temp;
}
```

Example 4.6.9 *Write a program in C++ to return the position of an element X in a list L.*

Solution : The routine is as given below -

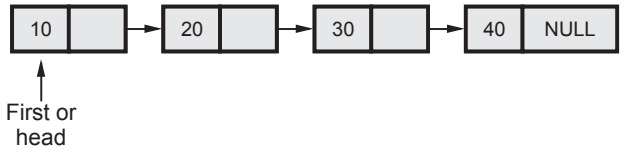
```
 Return_position(node *head,int key)
 {
   /* head represents the starting node of the List*/
  /* key represents the element X in the list*/
  int count=0;
  node *temp;
  temp=head;
  while(temp->data!=key)&&(temp!=NULL)
  {
```

```
   temp=temp->next;
   count=count+1;
 }
 if(temp->data==key)
   return count;
else if(temp==NULL)
   return -1;  /* -1 indicates that the element X is not present in the list*/
}
```

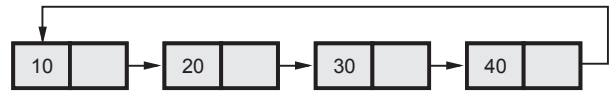
4.7 Types of Linked List

There are four types of linked list.

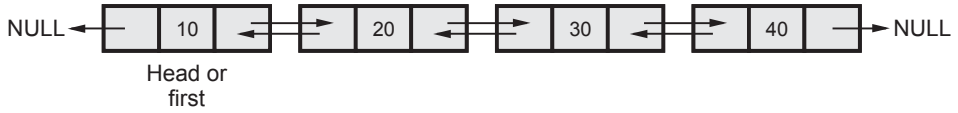
1. Singly linked list : It is called singly linked list because it contains only **one link** which points to the next node. The very first node is called **head** or **first.**



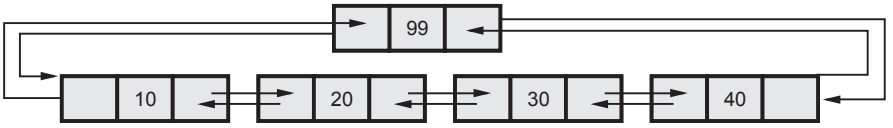
2. Singly circular linked list : In this type of linked list the next link of the last node points to the first node. Thus the overall structure looks circular. Following figure represents this type of linked list.



3. Doubly linear list : In this type of linked list there are two pointers associated with each node. The two pointer are - **next** and **previous.** As the name suggests the next pointer points to the next node and the previous pointer points to the previous node.



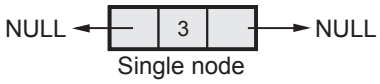
4. Doubly circular linked list : In this type of linked list there are two pointers **next** and **previous** to each node and the next pointer of last node points to the first node and previous pointer of the first node points to the last node making the structure circular.



4.8 Doubly Linked List

The typical structure of each node in doubly linked list is like this.

Fig. 4.8.1

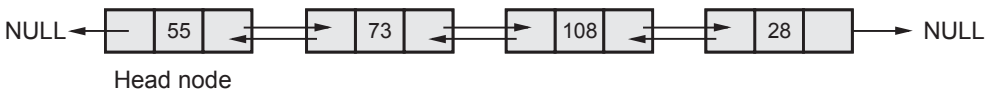


'C++' structure of doubly linked list :

```
typedef struct node
{
      int Data;
      struct node *prev;
      struct node *next;
}
```

The linked representation of a doubly linked list is

Thus the doubly linked list can traverse in both the directions, forward as well as backwards.



Logic for doubly linked list

In doubly linked list there are following operations

1. Create 2. Display 3. Insert 4. Delete

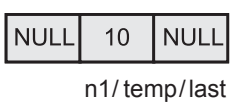
The node in doubly linked list (DLL) will look like this

4.8.1 Operations on Doubly Linked List

1. Creation of node

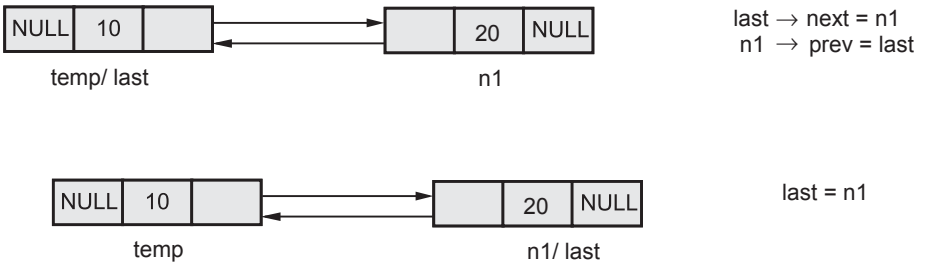
Initially set new node as

For the first time flag = 0

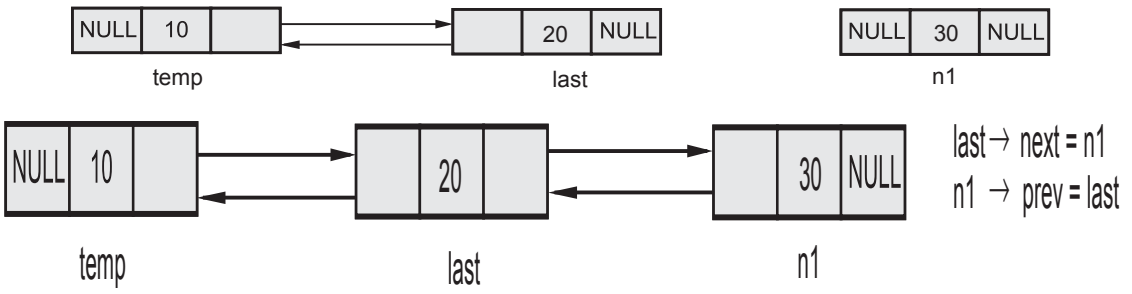



```
temp = n1;
last = temp;
```

Now set flag = 1, if we want to create more node then -



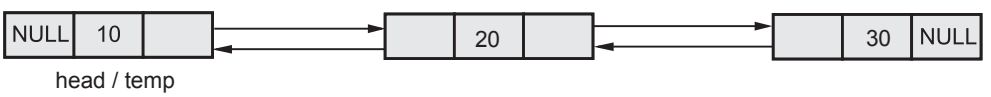
If we want to attach node 30 then



This we can continue to create a doubly linked list.

2. Display of Doubly linked list

We assign **head** node address to **temp** node.

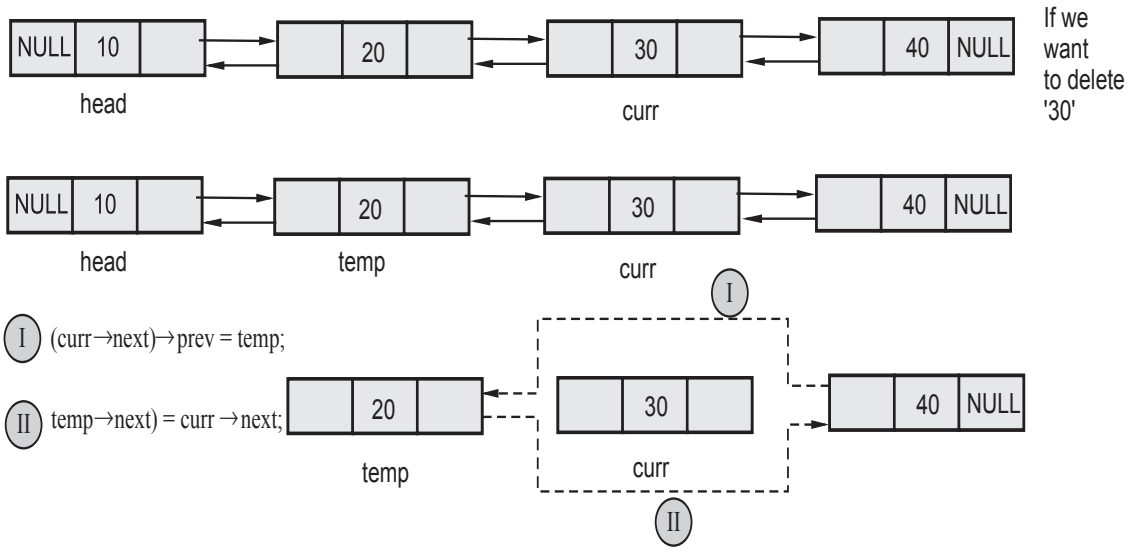



```
while (temp → next ! = NULL)
{
      " display data at temp node.
     temp = temp → next
}
```

This will display **10 20 30** as a doubly linked.

3. Deletion of a node in doubly linked list

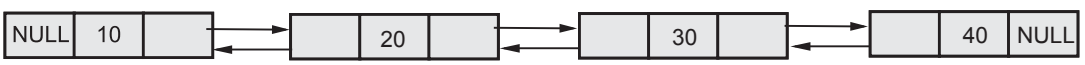
Consider,



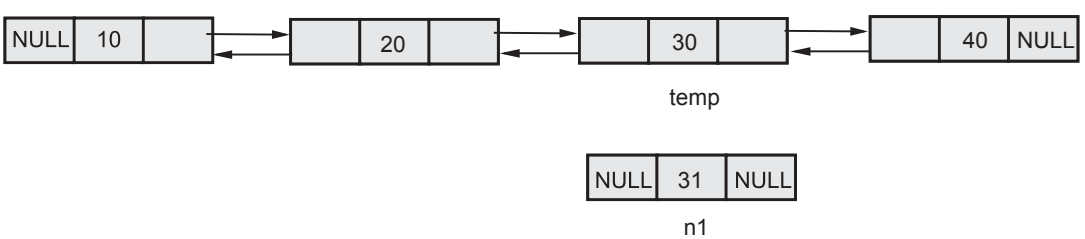
Then **delete curr.**

4. Insertion of a node

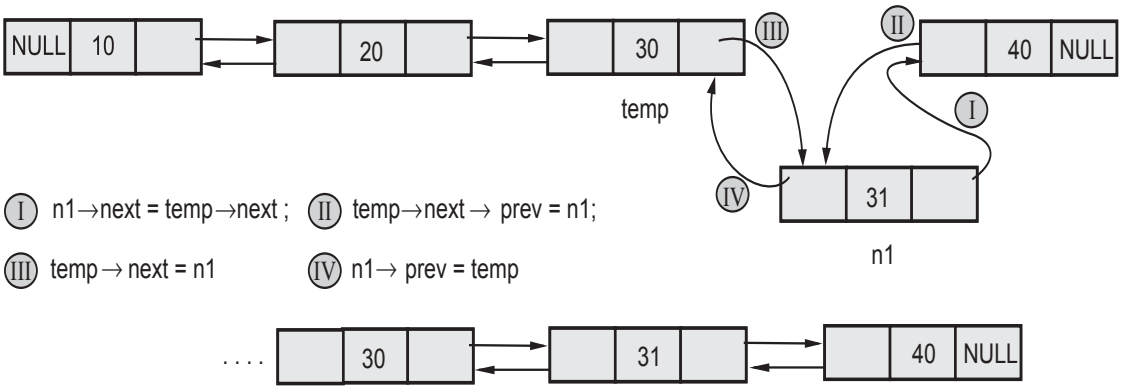
Consider a list



Suppose we want to insert 31 after 30 then first search for node 30 call it as **temp** node. Then create a node containing value 31.



Then



Thus **"Node inserted"**.

```
/*********************************************************************
 Demonstration Program to perform various operations such as insertion,
 deletion,finding the lenghth of the doublly linked list.
*********************************************************************/

// List of include files
#include<iostream.h>
#include <conio.h>
// class
class dll
{
 private:
 struct node
 {
     int Data;
     struct  node  *next;
     struct  node  *prev;
 }*head;
 public:
 dll();
 void create();
 void print();
 void insert_beg();
 void insert_after();
 void insert_end();
 void del();
 int length();
 ~dll();
};
```

```
/*
-------------------------------------------------------------------
The Constructor defined
-------------------------------------------------------------------
*/
 dll::dll()
 {
  head=NULL;
 }

/*
-------------------------------------------------------------------------
The destructor defined
-------------------------------------------------------------------------
*/
dll::~dll()
{
 node *temp,*temp1; //free the memory
 temp=head;
 while(temp!=NULL)
 {
     temp1=temp–>next;
     delete temp;
     temp=temp1;
 }
}
/*
-------------------------------------------------------------------------
        The Create function
-------------------------------------------------------------------------
*/
void dll :: create()
{
        // Local declarations here
        node  *n1,*last,*temp;
        char ans ='y';
        int flag=0;
        int val;
        do
        {
                cout<<"\nEnter the data :";
                cin>>val;

                // allocate new node
                n1 = new node;
                if ( n1 == NULL )
```

```
                cout<<"Unable to allocate memory\n";
            n1 –> Data  = val ;
            n1 –> next = NULL;
            n1 –> prev = NULL;
            if (flag==0) // Executed only for the first time
            {
                temp=n1;
                last=temp;
                flag=1;
            }
            else
            {
                // last keeps track of the most recently
                // created node

                last–>next=n1;
                n1–>prev=last;
                last=n1;
            }
            cout<<"\n\nEnter more?";
            ans = getche();
        }while(ans=='y's||ans=='Y');
        cout<<"\nThe List is created\n";
        head=temp;
        getch();
}

/*
-------------------------------------------------------------------------
        The length function
        Output:Returns number of items in list
        Calls : None
-------------------------------------------------------------------------
*/

int dll ::length()
{
        node *curr ;
        int  count;

        count = 0;
        curr = head;
        if ( curr == NULL )
        {
            cout<<"The list is empty\n";
            getch();
            return 0;
```

```
        }
        while ( curr != NULL )
        {
             count++;
             curr = curr –> next; // traverse till end
        }
        getch();
        return count;
}

/*
---------------------------------------------------------------------
        The print function
        Output:None, Displays data
        Calls : None
---------------------------------------------------------------------
*/
void dll ::print()
{
        node *temp ;

        temp = head;
        if ( temp == NULL )
        {
             cout<<"\nThe list is empty\n";
             getch(); clrscr();
             return;
        }
        else
        {
        cout<<"\nThe list is:";
        while(temp != NULL)
        {
             cout<<temp–>Data<<"  ";
             temp = temp –> next;
        }
        }
        getch();

}

/*
---------------------------------------------------------------------
        The del function
        Output:deletes an item from the list
---------------------------------------------------------------------
*/
```

```
void dll:: del()
{
        node *curr, *temp;
        int  data;
        curr=head;
        cout<<"\nEnter the data of the node you want to delete: ";
        cin>>data;
        while(curr!=NULL)
        {
          if(curr–>Data==data) //traverse till the required node to delete
          break;               //is found
          curr=curr–>next;
  }
  if(curr==NULL)
  cout<<"\nNode not found";
  else
  {
          if(curr==head)
          {
                if(head–>next==NULL&&head–>prev==NULL)//only one node
                                head=NULL;
                else
                {
                head=curr–>next;// first node
                head–>prev=NULL;
                }
          }
          else
          {
                temp=curr–>prev; //intermediate or end node
                if(curr–>next!=NULL)
                (curr–>next)–>prev=temp;
                temp–>next=curr–>next;
          }
                delete curr; //free memory
                cout<<"\nThe item is deleted\n";
        }

        getch();

}

/*--------------------------------------------------------------------
          Function to insert at end
--------------------------------------------------------------------*/
void dll::insert_end()
```

```
{
 node *temp,*n1;
 int val,flag=0;
 cout<<"\nEnter the data of the new node to insert";
 cin>>val;
 temp=head;
 if(temp==NULL)
  flag=1;
 else
 {
  while(temp–>next!=NULL)  // traverse till end
        temp=temp–>next;
 }

 // allocate new node
 n1 = new node;
 if ( n1 == NULL )
         cout<<"\nUnable to allocate memory\n";
  n1–> Data  = val ;
  n1 –> next = NULL;
  n1 –> prev = NULL;
  if(flag==0)
  {
  temp–>next=n1;  // attach at end
  n1–>prev=temp;
  }
  else
  head=n1;       //if list empty make this node as head node
  cout<<"\nNode inserted";
}


/*-------------------------------------------------------------------
          Function to insert after a node
-------------------------------------------------------------------*/
void dll:: insert_after()
{
 node *temp,*n1;
 int val,val1;
 cout<<"\nEnter the data of the new node to insert";
 cin>>val;
 cout<<"\nEnter the data of the node after which to insert";
 cin>>val1;
 temp=head;
 while(temp!=NULL)
 {
  if(temp–>Data==val1) //traverse till the required node after which to insert
```

```
        break;
 temp=temp–>next;
 }
 if(temp!=NULL)
 {
 /* allocate new node */
 n1 = new node;
 if ( n1 == NULL )
       cout<<"Unable to allocate memory\n";
  n1 –> Data  = val ;
  n1 –> next = NULL;
  n1 –> prev = NULL;
  /* after temp attach*/
  n1–>next=temp–>next;
  temp–>next–>prev=n1;
  temp–>next=n1;
  n1–>prev=temp;
  cout<<"\nNode inserted";
 }
 else
 cout<<"\nNode after which to insert not found";
}
/* ----------------------------------------------------------------
Function to insert at the beginning
----------------------------------------------------------------*/

void dll:: insert_beg()
{
 node *temp,*n1;
 int data;

 cout<<"\nEnter the data of the new node to insert";
 cin>>data;

 /* allocate new node */
 n1 = new node;
 if ( n1 == NULL )
 cout<<"Unable to allocate memory\n";
  n1 –> Data  = data ;
  n1 –> next = NULL;
  n1 –> prev = NULL;
 if(head)
 {
  n1–>next=head; //attach before head
  head->prev=n1;
 }
 head=n1; //make this node as head
```

```
 cout<<"\nNode inserted";
}


/*
-------------------------------------------------------------------------
        The main function
        Input : None
        Output: None
        Parameter Passing Method : None
-------------------------------------------------------------------------
*/
void main()
{
        dll d;
        int ch,ch1,cnt;
        char ans = 'y';
        do
        {
        clrscr();
        cout<<"\n"<<"MENU";
        cout<<"\n1.Create\n2.Display\n3.Insert\n4.Delete\n5.Length";
        cout<<"\nEnter ur choice:";
        cin>>ch;
        switch(ch)
        {
        case 1: d.create();
                  break;
        case 2: d.print();
                   break;
        case 3: clrscr();
              d.print();
              cout<<"\nMenu";
              cout<<"\n1.Insert at beginning\n2.Insert after";
              cout<<"\n3.Insert at end";
              cout<<"\nEnter your choice";
              cin>>ch1;
              switch(ch1)
              {
              case 1: d.insert_beg();
                      d.print();
                      break;
              case 2: d.insert_after();
                       d.print();
                       break;
              case 3: d.insert_end();
```

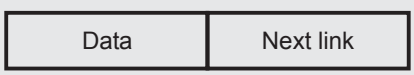
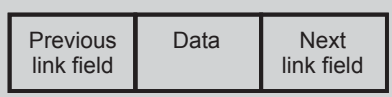
```
                    d.print();
                    break;
            default:cout<<"\nInvalid choice";
                }
              break;
      case 4:  d.print();
                    d.del();
                    d.print();
                    break;
      case 5: cnt=d.length();
                    cout<<"The length is: "<<cnt;
                    break;
      default: cout<<"\nInvalid choice";
      }
      cout<<"\n Do You Want To Continue?";
      cin>>ans;
      }while(ans=='y'||ans=='Y');
      getch();
      return;
}
```

Output

```
MENU
1.Create
2.Display
3.Insert
4.Delete
5.Length
Enter ur choice:1
Enter the data :1

Enter more?y
Enter the data :2

Enter more?y
Enter the data :3

Enter more?y
Enter the data :4

Enter more?y
Enter the data :5

Enter more?n
The List is created
```

```
 Do You Want To Continue?

MENU
1.Create
2.Display
3.Insert
4.Delete
5.Length
Enter ur choice:2

The list is:1  2  3  4  5
 Do You Want To Continue?y
```

4.8.2 Comparison between Singly and Doubly Linked List

Sr. No.	Singly Linked List	Doubly Linked List
1.	Singly linked list is a collection of nodes and each node is having one data field and next link field. For example : 	Doubly linked list is a collection of nodes and each node is having one data field, one previous link field and one next link field. For example : 
2.	The elements can be accessed using next link.	The elements can be accessed using both previous link as well as next link.
3.	No extra field is required; hence node takes less memory in SLL.	One field is required to store previous link hence node takes more memory in DLL.
4.	Less efficient access to elements.	More efficient access to elements.

Example 4.8.1 *Write a method in C++ to join two doubly linked lists into a single doubly linked list. In a join elements of second list are appended to the end of first list.*

Solution :

```
/*****************************************************************
Program to join two doubly linked list
*****************************************************************/

// List of include files

#include<iostream.h>
```

```
#include <conio.h>
// class defination
class dll
{
 private:
 struct node
 {
        int Data;
        structnode  *next;
        structnode  *prev;
 }*head;
 public:
 dll();
 void create();
 void print();
 void join(dll,dll);
};

/*
-------------------------------------------------------------------
 Constructor
-------------------------------------------------------------------
*/
dll::dll()
{
  head=NULL;
}
/*
-------------------------------------------------------------------------
        The Create function
-------------------------------------------------------------------------
*/
void dll :: create()
{
        node *New,last,*temp;
        int num,flag=0;
        char ans;
        // flag to indicate whether a new node
         //is created for the first time or not
        do
        {
                cout<<"\n Enter the Element :";
                cin>>num;
                /* allocate new node */
                New =new node;
                New -> Data  = num;
                New -> next = NULL;
```

```
            New->prev=NULL;
            if ( flag == 0 )
            {
              head = New; // First node is created
              temp=New;
              flag=1;
            }
            else
            {
                temp->next=New;//remaining nodes are created
                New->prev=temp;
                temp=New;
            }
            cout<<"\n Do You Want to insert More Elements?";
            ans=getch();
      }while(ans=='y');
      cout<<"\n The List Is Created";
}

/*
--------------------------------------------------------------------
      The print function
--------------------------------------------------------------------
*/
void dll ::print()
{
      node *temp ;
      temp = head;
      if ( temp == NULL )
      {
          cout<<"\nThe list is empty\n";
          getch(); clrscr();
          return;
      }
      else
      {
      cout<<"\nThe list is:";
      while(temp != NULL)
      {
          cout<<temp->Data<<"  ";
          temp = temp -> next;
      }
      }
      getch();
}
```

```
/*
-------------------------------------------------------------------
The join Function
-------------------------------------------------------------------
*/
void dll::join(dll d1,dll d2)
{
 node *temp1,*temp2,*prev_node1,*next_node2;
 temp1=d1.head;
 temp2=d2.head;
 if(temp1==NULL)
 temp1=temp2;
 while(temp1->next!=NULL)
 {
  temp1=temp1->next;//reaching at the end of first DLL
 }
 temp1->next=temp2;

}
/*
-------------------------------------------------------------------
        The main function
-------------------------------------------------------------------
*/
void main()
{
      dll d1,d2;
      int ch,ch1,cnt;
      char ans = 'y';
      do
      {
        clrscr();
        cout<<"\nEnter the data for 1 st Doubly Linked List ";
        d1.create();
        d1.print();
        cout<<"\nEnter the data for 2nd Doubly Linked List ";
        d2.create();
        d2.print();
        d1.join(d1,d2);
        d1.print();
        cout<<"\nDo you want to Continue?";
        cin>>ans;
        }while(ans=='y'||ans=='Y');
        return;
}
```

Output

```
Enter the data for 1 st Doubly Linked List
Enter the Element :10

Do You Want to insert More Elements?
Enter the Element :20

Do You Want to insert More Elements?
Enter the Element :30

Do You Want to insert More Elements?
The List Is Created
The list is:10  20  30
Enter the data for 2nd Doubly Linked List
Enter the Element :40

Do You Want to insert More Elements?
Enter the Element :50

Do You Want to insert More Elements?
The List Is Created
The list is:40  50
The list is:10  20  30  40  50
```

Review Questions

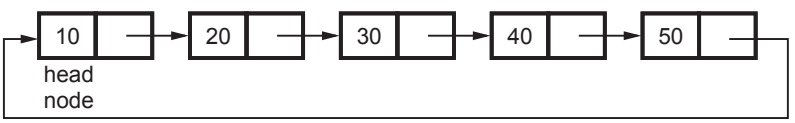
1. *Explain the representation of doubly linked list.*
2. *Write about operations performed on doubly linked list.*
3. *List and Explain doubly linked list operations.*

Advantages of Doubly Linked List over Singly Linked List

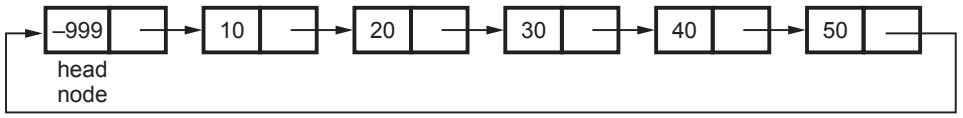
- The doubly linked list has two pointer fields. One field is previous link field and another is next link field.
- Because of these two pointer fields we can access any node efficiently whereas in singly linked list only one pointer field is there which stores forward pointer, which makes accessing of any node difficult one.

4.9 Circular Linked List

The circular linked list is as shown below -



or



The Circular Linked List (CLL) is similar to singly linked list except that the last node's next pointer points to first node.

Various operations that can be performed on circular linked list are,

1. Creation of circular linked list.
2. Insertion of a node in circular linked list.
3. Deletion of any node from linked list.
4. Display of circular linked list.

We will see each operation along with some example.

1. Creation of circular linked list

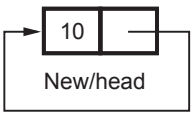
```
void sll ::Create()
{
 char ans;
 int flag=1;
 node *New,*temp;
 clrscr();
 do
 {
       New = new node;
       New–>next=NULL;
       cout<<"\n\n\n\tEnter The Element\n";
       cin>>New–>data;
       if(flag==1)    /*flag for setting the starting node*/
       {
              head = New;
              New–>next=head;
              flag=0;/*reset flag*/
       }
       else            /* find last node in list */
       {
        temp=head;
        while (temp–>next != head)/*finding the last node*/
               temp=temp–>next; /*temp is a last node*/
        temp–>next=New;
        New–>next=head; /*each time making the list
```

Single node in the Circular list

```
                circular*/
        }
        cout<<"\n Do you want to enter more nodes?(y/n)";
        ans=getch();
        }while(ans=='y'||ans=='Y');
}
```

Initially we will allocate memory for **New** node using a function **get_node()** . There is one variable **flag** whose purpose is to check whether first node is created or not. That means flag is 1 (set) then first node is not created. Therefore after creation of first node we have to reset the flag (making flag = 0).

Initially,



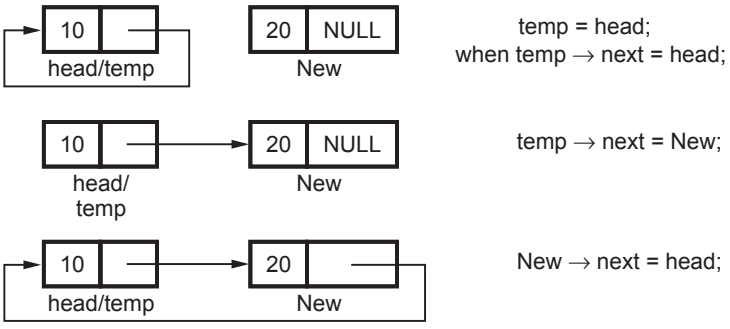
Suppose we have taken element '10' the flag = 1,

head = New;
New ← next = head;
flag = 0;

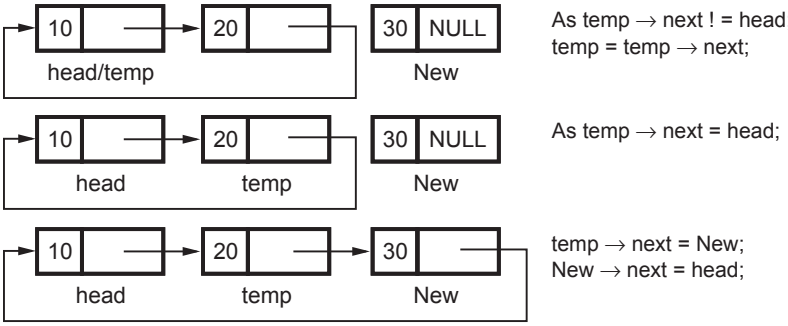
Here variable **head** indicates starting node.

Now as flag = 0, we can further create the nodes and attach them as follows.

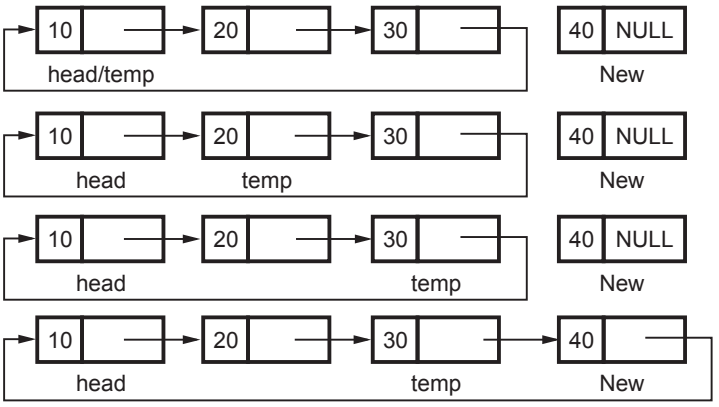
When we have taken element '20'



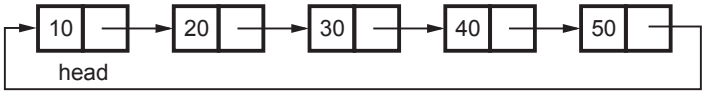
If we want to insert 30 then



If we want to insert 40 then



Thus we can create a circular linked list by inserting one more element 50. It is as shown below -



2. Display of Circular linked list

```
void sll::Display()
{
        node *temp;
        temp = head;
        if(temp == NULL)
         cout<<"\n Sorr ,The List Is Empty\n";
        else
        {
         do
         {
                cout<<"\t"<<temp->data;
                temp = temp->next;
         }while(temp != head);/*Circular linked list*/
        }
        getch();
}
```

The next node of last node is head node

3. Insertion of circular linked list

While inserting a node in the linked list, there are 3 cases –

a) inserting a node as a head node

b) inserting a node as a last node

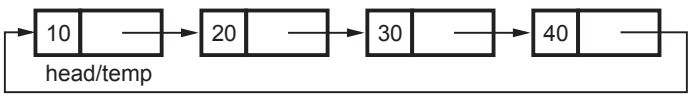
c) inserting a node at intermediate position

The functions for these cases is as given below –

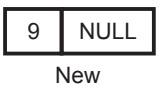
```
void sll::insert_head()
{
 node *New,*temp;
 New=new node;
 New–>next=NULL;
 cout<<"\n Enter The element which you want to insert ";
 cin>>New–>data;
 if(head==NULL)
        head=New;
 else
 {
              temp=head;
              while(temp–>next!=head)
                    temp=temp–>next;
              temp–>next=New;
              New–>next=head;
              head=New;
              cout<<"\n The node is inserted!";
 }
}
/*Insertion of node at last position*/
void sll::insert_last()
{
       node *New,*temp;
       New=new node;
       New–>next=NULL;
       cout<<"\n Enter The element which you want to insert ";
       cin>>New–>data;
       if(head==NULL)
              head=New;
       else
       {
        temp=head;
        while(temp–>next!=head)
                    temp=temp–>next;
        temp–>next=New;
        New–>next=head;
        cout<<"\n The node is inserted!";
       }
}
void sll::insert_after()
{
  int key;
  node *New,*temp;
  New= new node;
  New–>next=NULL;
```

```
 cout<<"\n Enter The element which you want to insert ";
 cin>>New->data;
 if(head==NULL)
 {
    head=New;
 }
 else
 {
       cout<<"\n Enter The element after which you want to insert the node ";
       cin>>key;
       temp=head;
       do
       {
             if(temp->data==key)
             {
                   New->next=temp->next;
                   temp->next=New;
                   cout<<"\n The node is inserted";
                   return;
             }
             else
                    temp=temp->next;
    }while(temp!=head);
  }
}
```

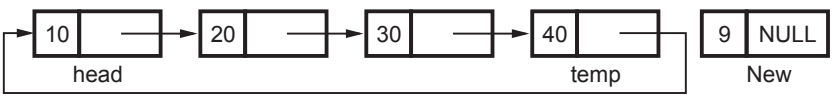
Suppose linked list is already created as -

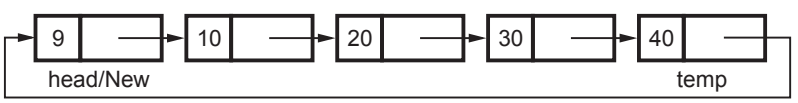


If we want to insert a **New** node as a head node then,



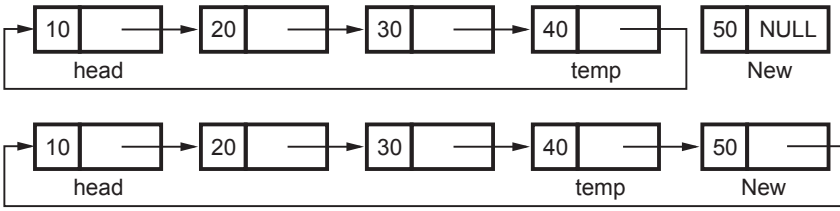
Then



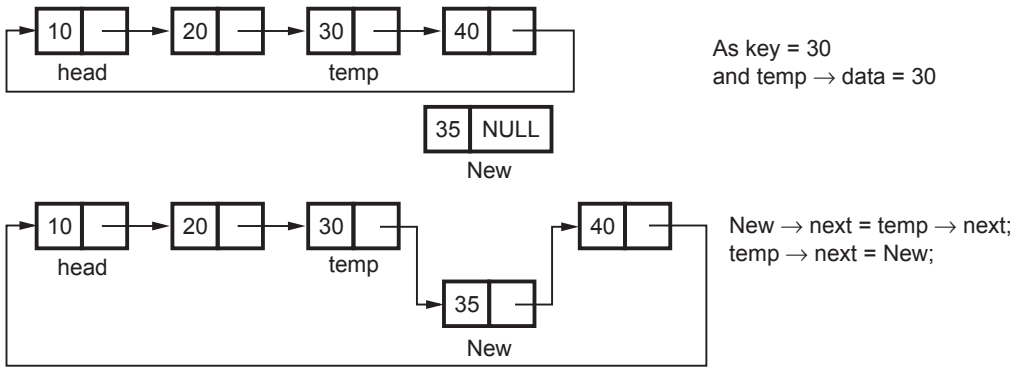


temp → next = New;
New → next = head;
head = New;

If we want to insert a New node as a last node consider a linked list



If we want to insert an element 35 after node 30 then,



4. Deletion of any node

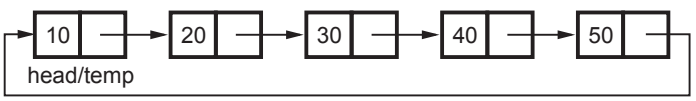
```
void sll::Delete()
{
 int key;
 struct node *temp,*temp1;
 cout<<"\n Enter the element which is to be deleted";
 cin>>key;
 temp=head;
 if(temp->data==key)/*If header node is to be deleted*/

{
        temp1=temp->next;
        if(temp1==temp)
  /*if single node is present in circular linked list
  and we want to delete it*/
        {
              temp=NULL;
              head=temp;
              cout<<"\n The node is deleted";
        }
        else  /*otherwise*/
        {
          while(temp->next!=head)
            temp=temp->next;/*searching for the last node*/
              temp->next=temp1;
```

If a single node is present in the list and we want to delete it

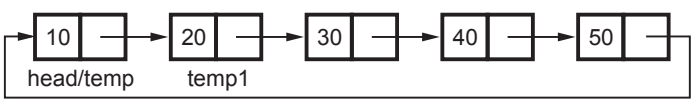
```
            head=temp1;/*new head*/
            cout<<"\n The node is deleted";
        }
 }
 else
 {
       while(temp->next!=head) /* if intermediate node is to
                        be deleted*/
       {
        if((temp->next)->data==key)
        {
            temp1=temp->next;
            temp->next=temp1->next;
            temp1->next=NULL;
            delete temp1;
            cout<<"\n The node is deleted";
        }
        else
            temp=temp->next;
       }
 }
}
```

The previous node of the node to be deleted is searched. Here **temp 1** is the node to be deleted and **temp** is the previous node of **temp 1**

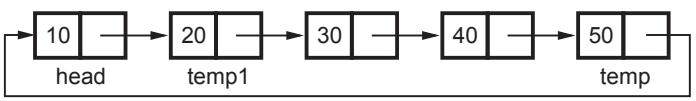
Suppose we have created a linked list as,



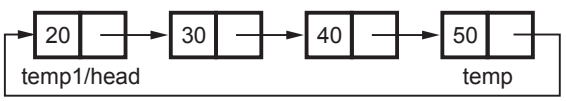
If we want to delete temp → data i.e. node 10 then,



temp1 = temp → next;

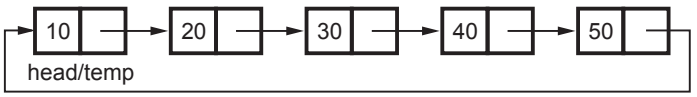


while (temp → next ! head)
temp = temp → next;

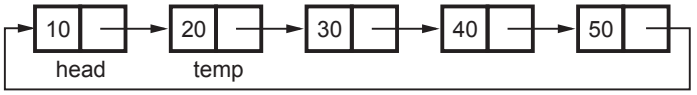


temp → next = temp1;
head = temp1;

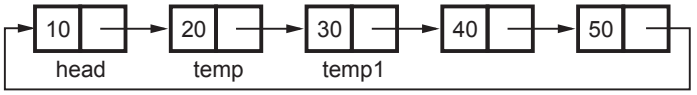
If we want to delete an intermediate node from a linked list which is given below



We want to delete node with '30' then

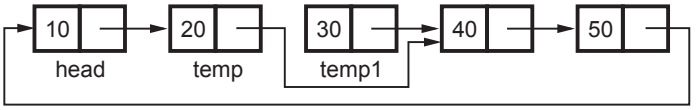


Key = 30
and
temp → next → data
= key, hence

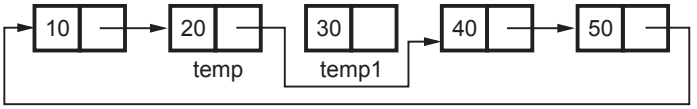


temp1 = temp → next;

Now



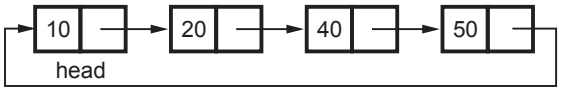
temp → next
= temp1 → next;



temp1 → next = NULL

Then delete temp1 ; / * to deallocate memory */

The linked list can be -



Thus node with value 30 is deleted from CLL.

5. Searching a node from circular linked list

```
void sll::Search(int num)
{
        node *temp;
        int found;
        temp=head;
        if ( temp == NULL )
         {
          cout<<"The Linked List is empty\n";
          getch();
          clrscr();
         }
        found=0;
        while(temp->next!=head && found==0)
```

```
        {
         if(temp->data == num)
          found=1;/*if node is found*/
         else
           temp=temp->next;
        }
        if(found==1)
         {
         cout<<"\n The node is present";
         getch();
        }
        else
        {
         cout<<"\n The node is not present";
         getch();
        }
}
```

while searching a node from circular linked list we go on comparing the data field of each node starting from the **head** node. If the node containing desired data is found we declare that the node is present.

'C++' Program

```
/*
*************************************************************************
 Program To Perform The Various Operations On The Circular Linked List
*************************************************************************
*/

#include<iostream.h>
#include <conio.h>
#include<stdlib.h>
// class


class sll
{
private:
struct node
{
 int data;
 struct node *next;
}*head;
public:
sll();
void Create();
void Display();
```

```
void Search(int key);
void insert_head();
void insert_after();
void insert_last();
void Delete();
~sll();
};

/*
---------------------------------------------------------------------
The  Constructor defined
---------------------------------------------------------------------
*/
sll::sll()
 {
  head=NULL;//initialize head to NULL
 }
/*
---------------------------------------------------------------------
The Destructor defined
---------------------------------------------------------------------
*/
sll::~sll()
{
  node *temp,*temp1;
  temp=head->next;
  delete head;
  while(temp!=NULL) //free the memory allocated
  {
   temp1=temp->next;
   delete temp;
   temp=temp1;
  }
}
/*
---------------------------------------------------------------------
        The Create function
---------------------------------------------------------------------
*/
void sll ::Create()
{
 char ans;
 int flag=1;
 node *New,*temp;
 clrscr();
 do
 {
```

```
        New = new node;
        New->next=NULL;
        cout<<"\n\n\n\tEnter The Element\n";
        cin>>New->data;
        if(flag==1)    /*flag for setting the starting node*/
        {
                head = New;
                New->next=head;  /*the circular list of a single
                        node*/
                flag=0;/*reset flag*/
        }
        else            /* find last node in list */
        {
         temp=head;
         while (temp->next != head)/*finding the last node*/
                temp=temp->next; /*temp is a last node*/
         temp->next=New;
         New->next=head; /*each time making the list
                    circular*/
        }
        cout<<"\n Do you want to enter more nodes?(y/n)";
        ans=getch();
        }while(ans=='y'||ans=='Y');
 }
void sll::Display()
{
        node *temp;
        temp = head;
        if(temp == NULL)
         cout<<"\n Sorry ,The List Is Empty\n";
        else
        {
         do
         {
                cout<<"\t"<<temp->data;
                temp = temp->next;
         }while(temp != head);/*Circular linked list*/
        }
        getch();
}

void sll::insert_head()
{
 node *New,*temp;
 New=new node;
 New->next=NULL;
 cout<<"\n Enter The element which you want to insert ";
```

```
 cin>>New–>data;
 if(head==NULL)
        head=New;
 else
 {
                temp=head;
                while(temp->next!=head)
                        temp=temp–>next;
                temp–>next=New;
                New–>next=head;
                head=New;
                cout<<"\n The node is inserted!";
 }
}
/*Insertion of node at last position*/
void sll::insert_last()
{
        node *New,*temp;
        New=new node;
        New–>next=NULL;
        cout<<"\n Enter The element which you want to insert ";
        cin>>New–>data;
        if(head==NULL)
                head=New;
        else
        {
         temp=head;
         while(temp–>next!=head)
                temp=temp–>next;
         temp–>next=New;
         New–>next=head;
         cout<<"\n The node is inserted!";
        }
}
void sll::insert_after()
{
  int key;
  node *New,*temp;
  New= new node;
  New–>next=NULL;
  cout<<"\n Enter The element which you want to insert ";
  cin>>New–>data;
  if(head==NULL)
  {
     head=New;
  }
  else
```

```
 {
       cout<<"\n Enter The element after which you want to insert the node ";
       cin>>key;
       temp=head;
       do
       {
             if(temp–>data==key)
             {
                   New–>next=temp–>next;
                   temp–>next=New;
                   cout<<"\n The node is inserted";
                   return;
             }
             else
                    temp=temp–>next;
    }while(temp!=head);
  }
}

void sll::Search(int num)
{
       node *temp;
       int found;
       temp=head;
       if ( temp == NULL )
        {
         cout<<"The Linked List is empty\n";
         getch();
         clrscr();
        }
       found=0;
       while(temp–>next!=head && found==0)
       {
        if(temp–>data == num)
         found=1;/*if node is found*/
        else
          temp=temp–>next;
       }
       if(found==1)
        {
        cout<<"\n The node is present";
        getch();
       }
       else
       {
        cout<<"\n The node is not present";
        getch();
```

```
        }
}
void sll::Delete()
{
 int key;
 struct node *temp,*temp1;
 cout<<"\n Enter the element which is to be deleted";
 cin>>key;
 temp=head;
 if(temp–>data==key)/*If header node is to be deleted*/
 {
        temp1=temp–>next;
        if(temp1==temp)
  /*if single node is present in circular linked list
  and we want to delete it*/
        {
              temp=NULL;
              head=temp;
              cout<<"\n The node is deleted";
        }
        else  /*otherwise*/
        {
          while(temp–>next!=head)
            temp=temp–>next;/*searching for the last node*/
              temp–>next=temp1;
              head=temp1;/*new head*/
              cout<<"\n The node is deleted";
        }
 }
 else
 {
       while(temp–>next!=head) /* if intermediate node is to
                          be deleted*/
       {
        if((temp–>next)–>data==key)
        {
              temp1=temp–>next;
              temp–>next=temp1–>next;
              temp1–>next=NULL;
              delete temp1;
              cout<<"\n The node is deleted";
        }
        else
              temp=temp–>next;
       }
 }
}
```

```
void main()
{
        sll s;
        char ch='y';
        int num,choice,choice1;
        do
        {
         clrscr();
         cout<<"\n Program For Circular Linked List\n";
         cout<<" 1.Insertion of any node\n\n";
         cout<<" 2. Display of Circular List\n\n";
         cout<<" 3. Insertion of a node in Circular List\n\n";
         cout<<" 4. Deletion of any node \n\n";
         cout<<" 5. Searching a Particular Element in The List\n\n";
         cout<<" 6.Exit ";
         cout<<"\n Enter Your Choice ";
         cin>>choice;
         switch(choice)
         {
                case 1 :s.Create();
                        break;
                case 2 :s.Display();
                        break;
                case 3: cout<<"\n 1. Insert a node as a head node";
                        cout<<"\n 2. Insert a node as a last node";
                        cout<<"\n 3. Insert a node at intermediate position in the linked list";
                        cout<<"\n Enter your choice for insertion of node";
                        cin>>choice1;
                        switch(choice1)
                        {
                          case 1:s.insert_head();
                                 break;
                          case 2:s.insert_last();
                                 break;
                          case 3:s.insert_after();
                                 break;
                         }
                          break;
                case 4:s.Delete();
                       break;
                case 5:cout<<"\n Enter The Element Which Is To Be Searched ";
                       cin>>num;
                       s.Search(num);
                       break;
                case 6:exit(0);
        }
```

```
        cout<<"\nDo you want to go to Main Menu?\n";
        ch = getch();
 }while(ch == 'y' || ch == 'Y');
}
```

Output

```
 Program For Circular Linked List
 1. Insertion of any node

 2. Display of Circular List

 3. Insertion of a node in Circular List

 4. Deletion of any node

 5. Searching a Particular Element in The List

 6.Exit
 Enter Your Choice 1



 Enter The Element
10
 Do you want to enter more nodes?(y/n)

Enter The Element
20
 Do you want to enter more nodes?(y/n)

Enter The Element
30
 Do you want to enter more nodes?(y/n)

Enter The Element
40

 Do you want to enter more nodes?(y/n)
Do you want to go to Main Menu?

 Program For Circular Linked List
 1.Insertion of any node

 2. Display of Circular List

 3. Insertion of a node in Circular List
```

```
4. Deletion of any node

5. Searching a Particular Element in The List

6. Exit
Enter Your Choice 2
        10      20      30      40
Do you want to go to Main Menu?

Program For Circular Linked List
1.Insertion of any node

2. Display of Circular List

3. Insertion of a node in Circular List

4. Deletion of any node

5. Searching a Particular Element in The List

6. Exit
Enter Your Choice 3

1. Insert a node as a head node
2. Insert a node as a last node
3. Insert a node at intermediate position in the linked list
Enter your choice for insertion of node 1

Enter The element which you want to insert 9

 The node is inserted!
Do you want to go to Main Menu?

 Program For Circular Linked List
 1.Insertion of any node

 2. Display of Circular List

 3. Insertion of a node in Circular List

 4. Deletion of any node

 5. Searching a Particular Element in The List

 6.Exit
 Enter Your Choice 2
       9       10      20      30      40
```

```
Do you want to go to Main Menu?

 Program For Circular Linked List
 1.Insertion of any node

 2. Display of Circular List

 3. Insertion of a node in Circular List

 4. Deletion of any node

 5. Searching a Particular Element in The List

 6.Exit
 Enter Your Choice 3

 1. Insert a node as a head node
 2. Insert a node as a last node
 3. Insert a node at intermediate position in the linked list
 Enter your choice for insertion of node2

 Enter The element which you want to insert 50

 The node is inserted!
Do you want to go to Main Menu?

 Program For Circular Linked List
 1.Insertion of any node

 2. Display of Circular List

 3. Insertion of a node in Circular List

 4. Deletion of any node

 5. Searching a Particular Element in The List

 6.Exit
 Enter Your Choice 2
       9       10       20       30       40       50
Do you want to go to Main Menu?

 Program For Circular Linked List
 1.Insertion of any node

 2. Display of Circular List
```

```
3. Insertion of a node in Circular List

4. Deletion of any node

5. Searching a Particular Element in The List

6.Exit
Enter Your Choice 3

1. Insert a node as a head node
2. Insert a node as a last node
3. Insert a node at intermediate position in the linked list
Enter your choice for insertion of node 3

Enter The element which you want to insert 35

Enter The element after which you want to insert the node 30

The node is inserted
Do you want to go to Main Menu?

Program For Circular Linked List
1.Insertion of any node

2. Display of Circular List

3. Insertion of a node in Circular List

4. Deletion of any node

5. Searching a Particular Element in The List

6.Exit
Enter Your Choice 2
      9      10      20      30      35      40      50
Do you want to go to Main Menu?

Program For Circular Linked List
1.Insertion of any node

2. Display of Circular List

3. Insertion of a node in Circular List

4. Deletion of any node

5. Searching a Particular Element in The List
```

```
6.Exit
Enter Your Choice 4

Enter the element which is to be deleted 30

The node is deleted
Do you want to go to Main Menu?

Program For Circular Linked List
1.Insertion of any node

2. Display of Circular List

3. Insertion of a node in Circular List

4. Deletion of any node

5. Searching a Particular Element in The List

6.Exit
Enter Your Choice 2
      9       10       20       35       40       50

Program For Circular Linked List
1.Insertion of any node

2. Display of Circular List

3. Insertion of a node in Circular List

4. Deletion of any node

5. Searching a Particular Element in The List

6.Exit
Enter Your Choice 5

Enter The Element Which Is To Be Searched 40

The node is present
Do you want to go to Main Menu?

Program For Circular Linked List
1.Insertion of any node

2. Display of Circular List
```

```
3. Insertion of a node in Circular List

4. Deletion of any node
5. Searching a Particular Element in The List

6.Exit
Enter Your Choice 5

Enter The Element Which Is To Be Searched 75

 The node is not present
Do you want to go to Main Menu?

 Program For Circular Linked List
 1.Insertion of any node

 2. Display of Circular List

 3. Insertion of a node in Circular List

 4. Deletion of any node

 5. Searching a Particular Element in The List

 6.Exit
 Enter Your Choice 6
```

Advantages of Doubly Linked List over Singly Linked List

- In circular list the next pointer of last node points to head node, whereas in doubly linked list each node has **two pointers** : One previous pointer and another is next pointer.
- The **main advantage** of circular list over doubly linked list is that with the help of single pointer field we can **access head** node quickly.
- Hence some amount of memory get saved because in circular list only one pointer field is reserved.

4.10 Doubly Circular Linked List

The doubly circular linked list can be represented as follows

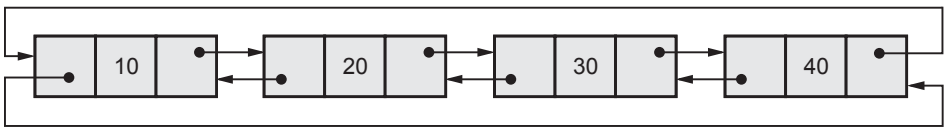


Fig. 4.10.1 Doubly circular list

```
The node structure will be
struct node
{
        int data;
        struct node *next;
        struct node *prev;
};
```

C++ Function for creation of doubly Circular List

```
struct node *Create()
{
        char ans;
        int flag = 1;
        struct node *head, *New, *temp;
        struct node *get_node();
        head = NULL;
        do
        {
                New = new node;
                cout<<"\n\n\n\tEnter The Element\n";
                cin>>New->data;

                if (flag == 1)    /*flag for setting the starting node*/
                {
                        head = New;
                        New->next = head;  /*the circular list of a single node*/
                        head->prev = New;
                        flag = 0;/*reset flag*/
                }
                else            /* find last node in list */
                {
                        temp = head;
                        while (temp->next != head)/*finding the last node*/
                                temp = temp->next; /*temp is a last node*/
                        temp->next = New;
                        New->prev = temp;
                        head->prev = New;
                        New->next = head; /*each time making the list circular*/
                }
                cout<<"\n Do you want to enter more nodes ? ";
                cin>>ans;
        } while (ans == 'y' || ans == 'Y');
        return head;
}
```

4.11 Applications of Linked List

Various applications of linked list are -

1. Linked list can be used to implement linear data structures such as stacks, and qeues.
2. Linked list is useful for implementing the non-linear data structures, such as tree and graph.
3. Polynomial representation, and operations such as addition, multiplication and evaluation can be performed using linked list.

4.12 Polynomial Manipulations

4.12.1 Representation of a Polynomial using Linked List

As we know a polynomial has the main fields as coefficient, exponent in linked list, it will have one more field called **'link'** field to point to next term in the polynomial. If there are n terms in the polynomial then n such nodes has to be created.

The typical node will look like this,

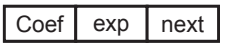


Fig. 4.12.1 Node of polynomial

For example : To represent $3x^2 + 5x + 7$ the link list will be,

In each node, the exponent field will store exponent corresponding to that term, the coefficient field will store coefficient corresponding to that term and the link field will point to next term in the polynomial. Again for simplifying the algorithms such as addition of two polynomials we will assume that the polynomial terms are stored in descending order of exponents.

The node structure for a singly linked list for representing a term of polynomial can be defined as follows :

```
typedef struct Pnode
{
        float coef;
        int exp;
        struct node *next;
} p;
```

Advantages of linked representation over arrays :

1. Only one pointer will be needed to point to first term of the polynomial.
2. No prior estimation on number of terms in the polynomial is required. This results in flexible and more space efficient representation.
3. The insertion and deletion operations can be carried out very easily without movement of data.

Disadvantage of linked representation over arrays :

1. We can not access any term randomly or directly we have to go from start node always.

4.12.2 Addition of Two Polynomials Represented using Singly Linear Link List

Logic for polynomial addition by linked list :

Step 1 : First of all we create two linked polynomials.

For example

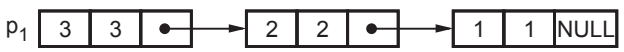
$$P_1 = 3x^3 + 2x^2 + 1x$$

$$P_2 = 5x^5 + 3x^2 + 7$$

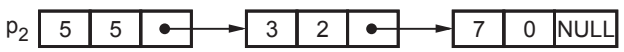
Each node in the polynomial will look like this,

p node

The linked list for p_1 will be,

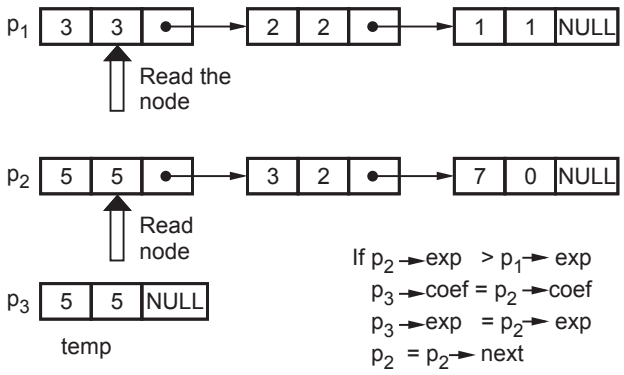


Similarly the p_2 will be,

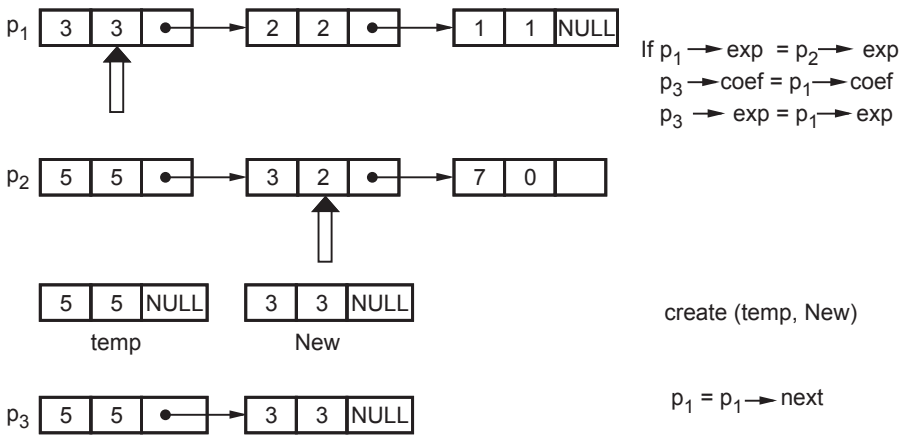


Step 2 :

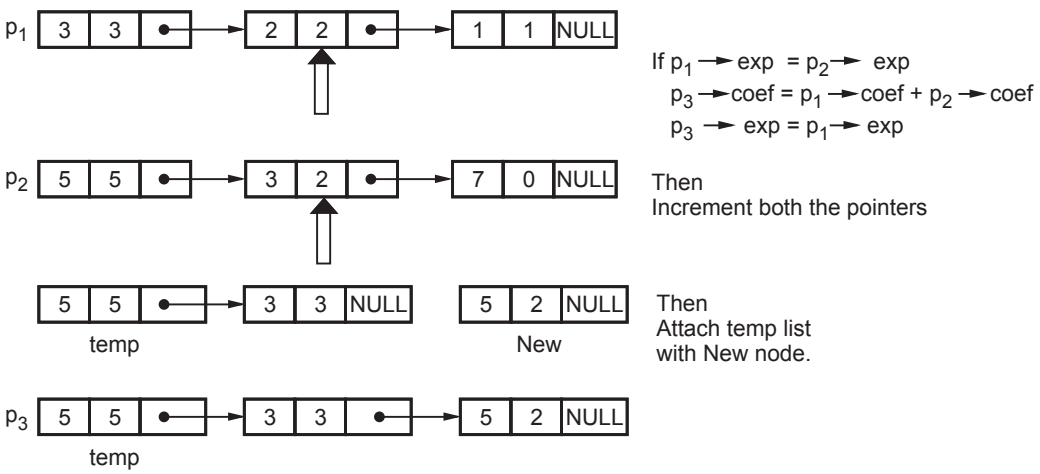
For addition of two polynomials if exponents of both the polynomials are same then we add the coefs. For storing the result we will create the third linked list say p_3. The processing will be as follows :


p1
3 3
2 2
1 1
NULL
Read the node
p2
5 5
3 2
7 0
NULL
Read node
p3
5 5
NULL
temp
If p2 → exp > p1 → exp
p3 → coef = p2 → coef
p3 → exp = p2 → exp
p2 = p2 → next

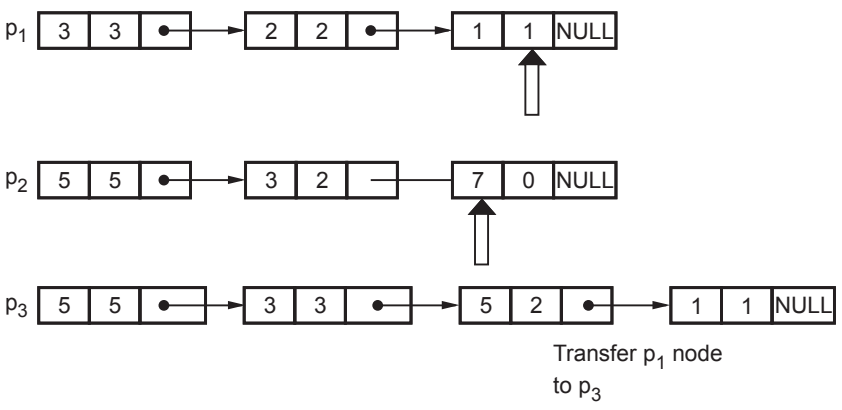
Step 3 :


p1
3 3
2 2
1 1
NULL
If p1 → exp = p2 → exp
p3 → coef = p1 → coef
p3 → exp = p1 → exp
p2
5 5
3 2
7 0
5 5
NULL
temp
3 3
NULL
New
create (temp, New)
p1 = p1 → next
p3
5 5
3 3
NULL

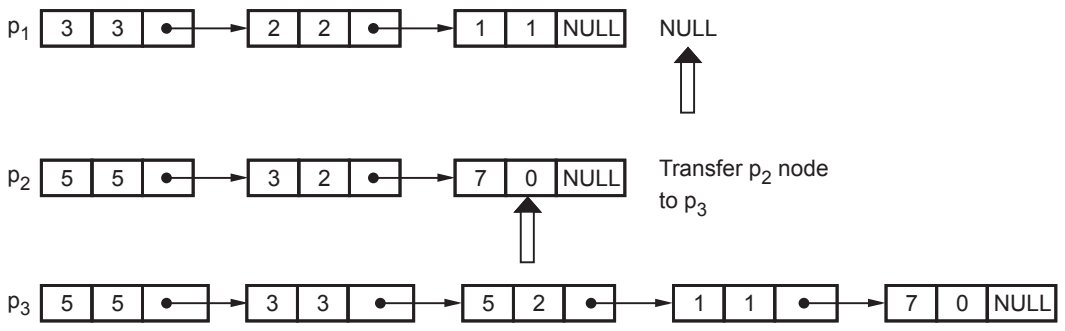
Step 4 :


p1
3 3
2 2
1 1
NULL
If p1 → exp = p2 → exp
p3 → coef = p1 → coef + p2 → coef
p3 → exp = p1 → exp
p2
5 5
3 2
7 0
NULL
Then
Increment both the pointers
5 5
3 3
NULL
temp
5 2
NULL
New
Then
Attach temp list
with New node.
p3
5 5
3 3
5 2
NULL
temp

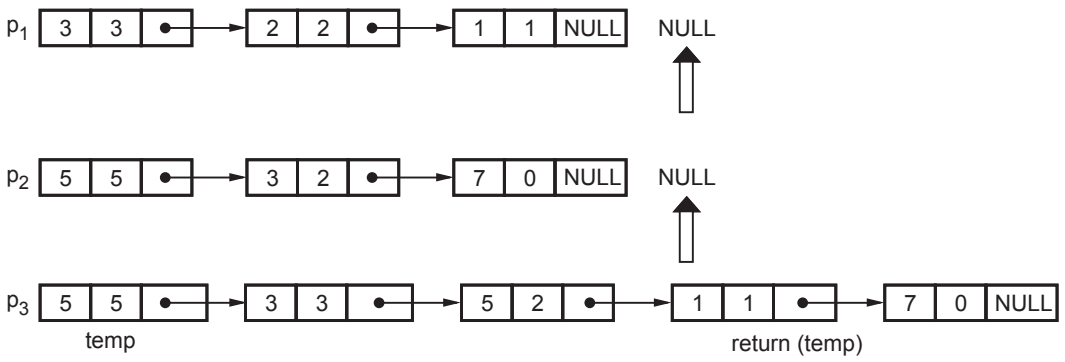
Step 5 :



Step 6 :



Finally



p_3 list is the addition of two polynomials.

'C++' Program

```
/*************************************************************************
        Program To Perform Addition Of Two Polynomials Using
        Singly Linear Linked List
*************************************************************************/
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>

#define TRUE  1
#define FALSE   0
class Lpadd
{
       private:
       typedef struct  pnode
       {
              float coef;
              int exp;
              structpnode  *next;
       }p;
       p *head;
       public:
       Lpadd();
       void get_poly(),add(Lpadd,Lpadd);
       void display();
       p *Attach(int,float,p *);
       ~Lpadd();
};
/*
-------------------------------------------------------------------------
The constructor defined
-------------------------------------------------------------------------
*/
Lpadd::Lpadd()
{
       head = NULL ;
}

void Lpadd::get_poly()
{
       p  *New, *last;
       int Exp,flag;
       float Coef;
       char ans='y';
       flag = TRUE; // flag to indicate whether a new node
              // is created for the first time or not
       cout<<"\nEnter the polynomial in desending order of exponent\n";
```

```
        do
        {
                cout<<"\nEnter the Coefficient and Exponent of a term :";
                cin>>Coef>>Exp;
                // allocate new node
                New=new p;
                New–>next=NULL;
                if ( New == NULL )
                  cout<<"\nMemory can not be allocated";
                 New –> coef = Coef;
          //putting coef,exp values in the node
                 New –> exp  = Exp ;
                 if ( flag==TRUE ) // Executed only for the first time
                 {                       //for creating the first node
                        head = New;
                        last = head;
                        flag = FALSE;
                 }
                 else
                  {
                        // last keeps track of the most recently
                          // created node
                        last –>next = New;
                        last = New;
                }
                cout<<"\n Do you Want To Add more Terms?(y/n)";
                ans=getch();
        }while(ans=='y');
        return;
}
void Lpadd::display()
{
        p *temp ;
        temp=head;
        if ( temp == NULL )
        {
                cout<<"The polynomial is empty\n";
                getch();
                return;
        }
        cout<<endl;
        while ( temp–>next != NULL )
        {
                cout<<temp–>coef<<" x^  "<<temp–>exp<<"+ ";
                temp = temp –> next;
        }
        cout<<temp –>coef<<" x^  "<<temp –>exp;
```

```
        getch();
}

void Lpadd::add(Lpadd p1,Lpadd p2)
{
        p *temp1, *temp2,*dummy;
        float Coef;
        temp1 =p1.head;
        temp2 =p2.head;
        head = new p;
        if ( head == NULL )
                cout<<"\nMemory can not be allocated";
        dummy = head;//dummy is a start node
        while ( temp1 != NULL && temp2 != NULL )
        {
                if(temp1–>exp==temp2–>exp)
                {
                        Coef = temp1–>coef + temp2–>coef;
                        head = Attach(temp1–>exp,Coef,head);
                        temp1 = temp1–>next;
                        temp2 = temp2–>next;
                 }
                else if(temp1–>exp<temp2–>exp)
                {
                        Coef = temp2 –> coef;
                        head =Attach(temp2–>exp, Coef,head);
                        temp2 = temp2 –> next ;
                }
                else if(temp1–>exp>temp2–>exp)
                {
                        Coef = temp1 –> coef;
                        head =Attach(temp1–>exp, Coef,head);
                        temp1 = temp1 –> next ;
                }
        }
        //copying the contents from first polynomial to the resultant
        //poly
        while ( temp1 != NULL )
        {
                head = Attach(temp1–>exp,temp1–>coef,head);
                temp1 = temp1 –> next;
        }
        //copying the contents from second polynomial to the resultant poly.
        while ( temp2 != NULL )
        {
                head = Attach(temp2–>exp,temp2–>coef,head);
                temp2 = temp2 –> next;
```

```
        }
        head –>next = NULL;
        head = dummy –> next;//Now set temp as starting node
        delete dummy;
        return;
}
p *Lpadd::Attach( int Exp, float Coef, p *temp)
{
        p *New, *dummy;

        New = new p;
        if (New == NULL )
          cout<<"\n Memory can not be allocated \n";
        New–>exp  = Exp;
        New–>coef = Coef;
        New–>next = NULL;
        dummy = temp;
        dummy –> next = New;
        dummy = New;
        return(dummy);
}
Lpadd::~Lpadd()
{
        p *temp;
        temp=head;
        while(head!=NULL)
        {
                temp=temp–>next;
                delete head;
                head=temp;
        }
}
void main()
{
        Lpadd p1,p2,p3;
        cout<<"\n Enter the first polynomial\n\n";
        p1.get_poly();
        cout<<"\nEnter the Second polynomial\n\n";
        p2.get_poly();
        cout<<"\nThe first polynomial is \n";
        p1.display();
        cout<<"\nThe second polynomial is\n";
        p2.display();
        p3.add(p1,p2);
        cout<<"\nThe Addition of the Two polynomials is...\n";
        p3.display();
        exit(0);
}
```

Output

```
 Enter the first polynomial

Enter the polynomial in desending order of exponent

Enter the Coefficient and Exponent of a term : 3 3

 Do you Want To Add more Terms?(y/n)
Enter the Coefficient and Exponent of a term : 2 2

 Do you Want To Add more Terms?(y/n)
Enter the Coefficient and Exponent of a term : 1 1

 Do you Want To Add more Terms?(y/n)
Enter the Coefficient and Exponent of a term : 5 0

 Do you Want To Add more Terms?(y/n)n

Enter the Second polynomial

Enter the polynomial in desending order of exponent

Enter the Coefficient and Exponent of a term : 5 1

 Do you Want To Add more Terms?(y/n)
Enter the Coefficient and Exponent of a term : 7 0
 Do you Want To Add more Terms?(y/n)
The first polynomial is
3 x^ 3+ 2 x^ 2+ 1 x^ 1+ 5 x^ 0
The second polynomial is
5 x^ 1+ 7 x^ 0
The Addition of the Two polynomials is...
3 x^ 3+ 2 x^ 2+ 6 x^ 1+ 12 x^ 0
```

4.13 Generalized Linked List (GLL)

4.13.1 Concept of Generalized Linked List

A generalized linked list A, is defined as a finite sequence of n>=0 elements,

$a_1, a_2 a_3, \ldots, a_n$, such that ai are either atoms or the list of atoms. Thus

$A = (a_1, a_2 a_3, \ldots, a_n)$

Where n is total number of nodes in the list.

Now to represent such a list of atoms we will have certain assumptions about the node structure

Flag = 1 means down pointer exists.

= 0 means next pointer exists.

Data means the atom

Down pointer is address of node which is down of the current node.

Next pointer is the address of the node which is attached as the next node.

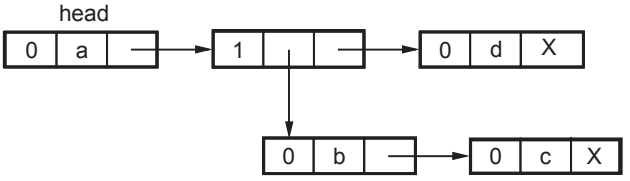
With this typical node structure let us represent the generalized linked list for the above. Let us take few example to learn how actually the generalized linked list can be shown,

Typical 'C' structure of GLL -

```
typedef struct node
  {
   char c;                /*Data*/
   int ind;               /*Flag*/
   struct node *next,*down; /*next & down   pointer */
  }gll;
```

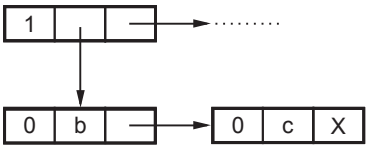
Example of GLL [List representation]

1. (a , (b , c), d)



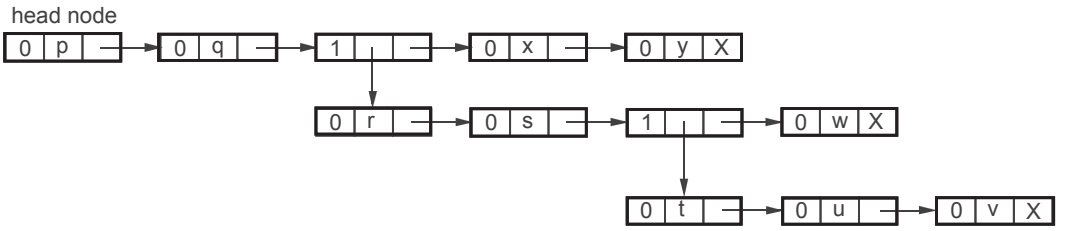
In above example the head node is 

In this case the first field is 0, it indicates that the second field is variable. If first field is 1 means the second field is a down pointer, means some list is starting.

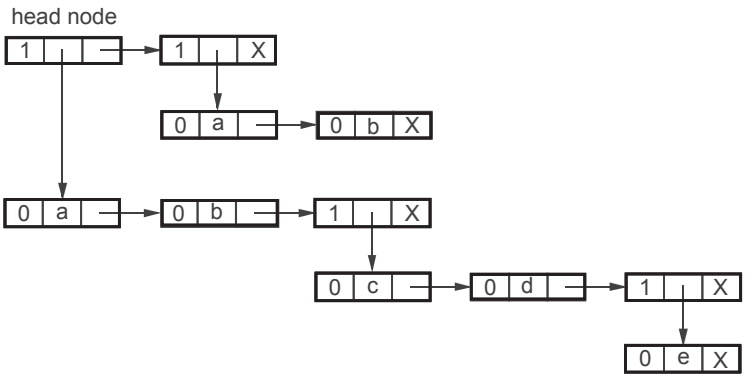


The above figure indicates (b , c). In this way the generalised linked list can be built. The X in the next pointer field indicates NULL value.

2. G = (p , q, (r, s, (t, u, v), w), x, y)

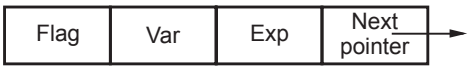


3. ((a, b, (c, d, (e))), (a, b))



4.13.2 Polynomial Representation using Generalized Linked List

The typical node structure for representation of polynomial is

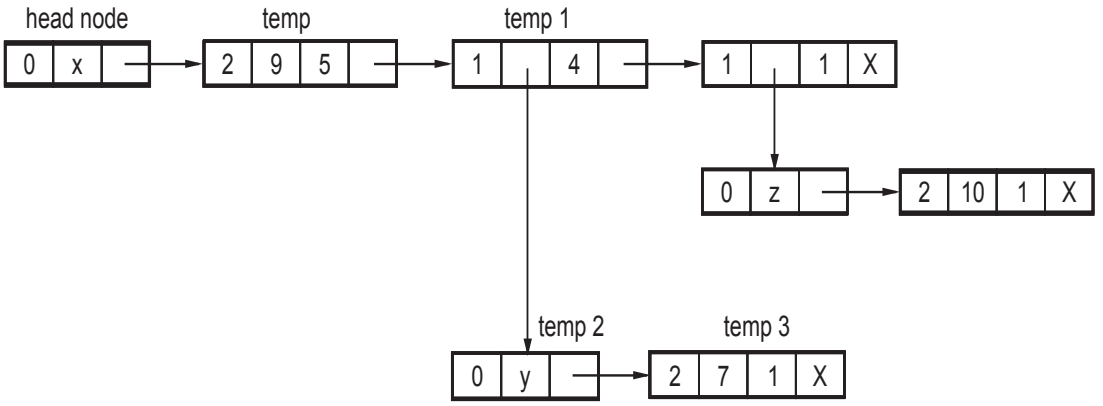


Let us see each field one by one Flag = 0 means variable is present

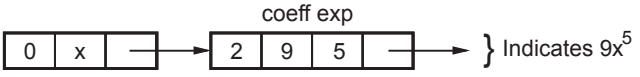
Flag = 1. Means down pointer is present

Flag = 2. Means coefficient and exponent is present

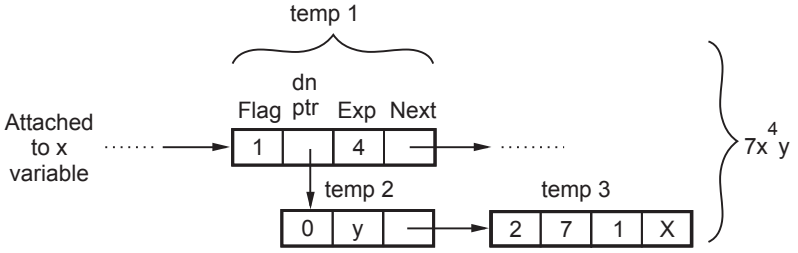
Example 1 : $9x^5 + 7xy^4 + 10xz$



In the above example the head node is of variable x. The temp node shows the first field as 2 means coefficient and exponent are present.



Since temp node is attached to head node and head node is having variable x, temp node having coefficient = 9 and exponent = 5. The above two nodes can be read as $9x^5$. Similarly in the figure.



The node temp1 can be read as x^4. The flag field is 1 means down pointer is there

The temp2 = y

temp3 = coefficient = 7

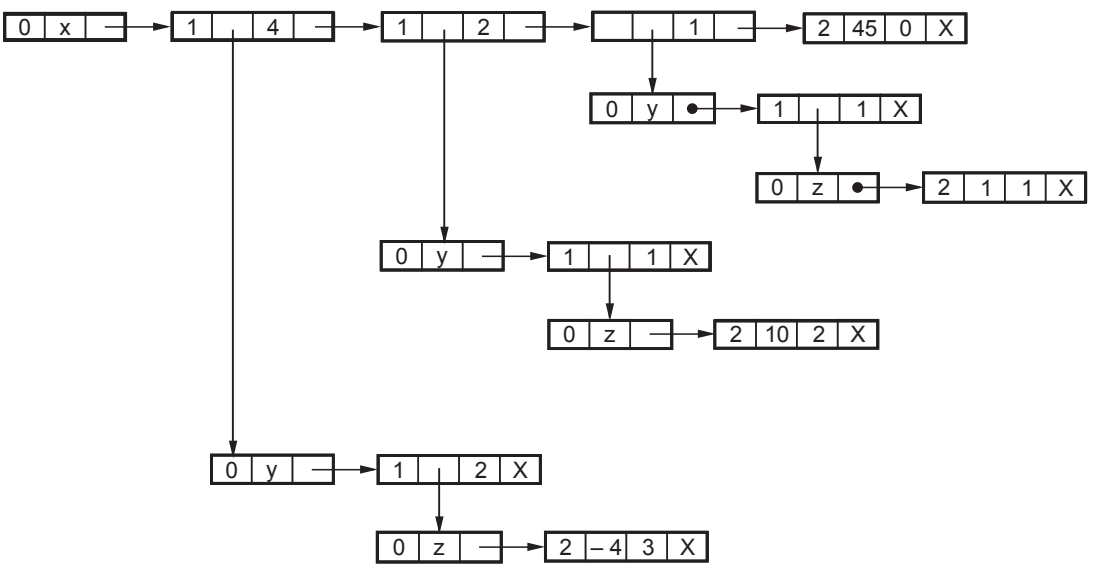
exponent = 1

Flag = 2 means the node contains coefficient and exponent values

temp2 is attached to temp3 this means $7y^1$ and temp2 is also attached to temp1 means

temp1	×	temp2
x^4	×	$7y^1$
	=	$7x^4y^1$ value is represented by above figure.

Example 2 : $-4x^4y^2z^3 + 10x^2yz^2 + 7xyz + 45$



4.13.3 Advantages of Generalized Linked List

1. For representing the list of atoms which may contain the multiple sub-lists the representation using GLL is the efficient option.
2. As singly linked list, doubly linked list, circular linked list or the arrays are not efficient ways for multi-variable polynomials. The use of GLL in performing various operations on multi-variable polynomial increases the efficiency of the algorithm.

❑❑❑