Unit III

3 Searching and Sorting

3.1 : Introduction to Searching and Sorting

Q.1 What are the applications of sorting ?

Ans. : 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 : Searching Techniques

Q.2 What is searching ?

Ans. : 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 algorithms.**

Most commonly used searching algorithms are -

i) Sequential or linear search

ii) Indexed sequential search

iii) Binary search

Q.3 Explain sequential search technique.

Ans. : • 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. Q.3.1 Represents students database for sequential search

From the above Fig. Q.3.1 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;
}
```

Q.4 Explain Binary search algorithm with suitable example.

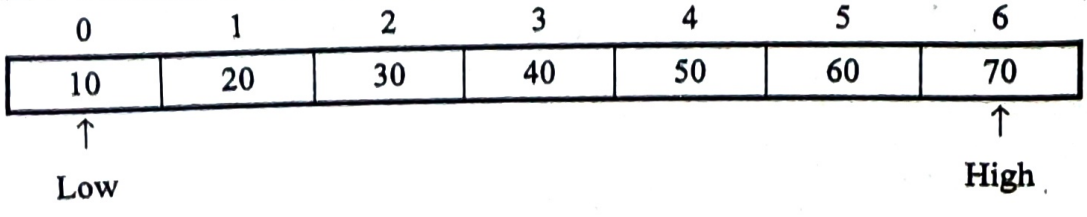
Ans. : • Concept : Binary search is a searching algorithm in which the list of elements is divided into two sublists and the key element is compared with the middle element. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves(sublists) 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];

Q.5 Apply binary search method to search 60 from the list 10, 20, 30, 40, 50, 60, 70.

Ans. : 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 :

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

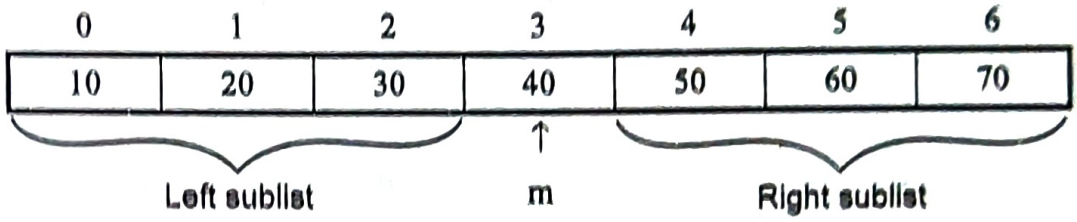
$$m = (0 + 6)/2$$

$$m = 3$$

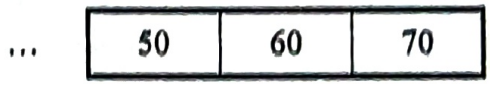
Then Check $A[m] \stackrel{?}{=} KEY$

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

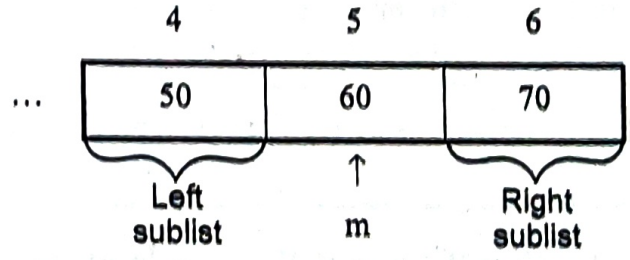
∴ Search the right sublist.



The right sublist is



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



$m = (low + high)/2$

$m = (4 + 6)/2$

∴ $m = 5$

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.

Q.6 What is the difference between linear search and binary search ?

Ans. : 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.

Q.7 Explain the Fibonacci Search technique with suitable example.

☞ **[SPPU : June-22, Marks 9]**

Ans. : 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.

a = i.e. previous element of b

= b because this is the only element in Fib. Series which is < n i.e. < 7

Now we have

f = 7

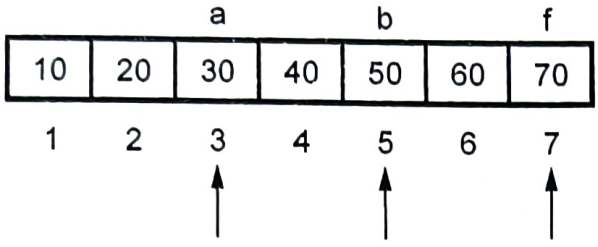
b = 5

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

$\therefore f = f - a = 7 - 3 = 4$

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

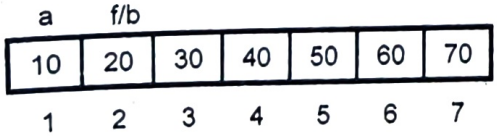
$\therefore f = f - a = 4 - 2 = 2$

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

$\therefore f = f - a = 7 - 3 = 4$

b = a = 3

a = b – a = 2

Again we compare

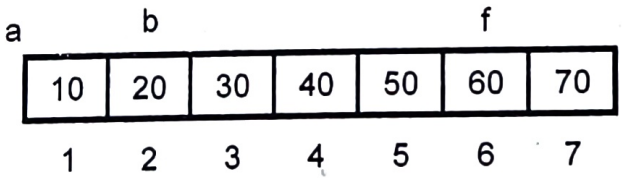
if (Key > arr [f])

$\therefore$ i.e. 60 > arr [f] i.e. 40

$\therefore f = f + a = 4 + 2 = 6$

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 (key > arr [f])
if (20 < 20) → No	
If Key > arr [f]) i.e.	i.e. if (60 > 60) → No
if (20 > 20) → No	
That means	That means
"Element is present at	**."Element is present at**
f = 2 location"	**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

$\therefore$ n = 7, we want to find key = 60

Now check Fibbonacci series.

As n < 8

set f = 8

a = 5

b = 3

			(b) 3	4	(a) 5	6	(f) 8
0	1	2					
10	20	30	40	50	60	70	...

Set offset = – 1

$\therefore$ i = 2 $\because$ 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)

$\therefore$ f = 5

a = 3

b = 2

		(b) 2	(a) 3	4	(f) 5	6
0	1					
10	20	30	40	50	60	70

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

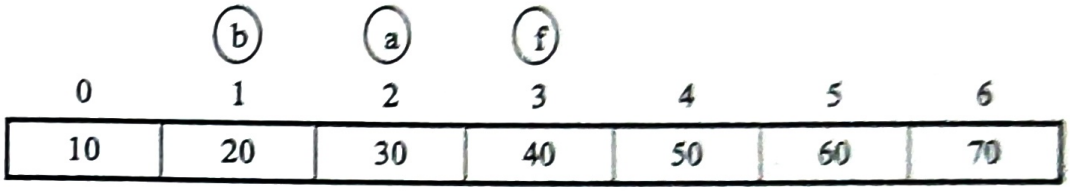
Now i = 4 $\because$ i = min (offset + b, n – 1)

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

$\therefore$ f = 3

a = 2

b = 1



Set offset = i i.e. 4

Now new i = 5 $\because$ 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.

Q.8 Write a Python program for index sequential search.

Ans. :

```
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)
```

For example -

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

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

$\therefore$ n = 7, we want to find key = 60

Now check Fibbonacci series.

As n < 8

set f = 8

a = 5

b = 3

			(b)		(a)		
0	1	2	3	4	5	6	
10	20	30	40	50	60	70	...

Set offset = – 1

$\therefore$ i = 2 $\because$ 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)

$\therefore$ f = 5

a = 3

b = 2

		(b)	(a)		(f)
0	1	2	3	4	5
10	20	30	40	50	60

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

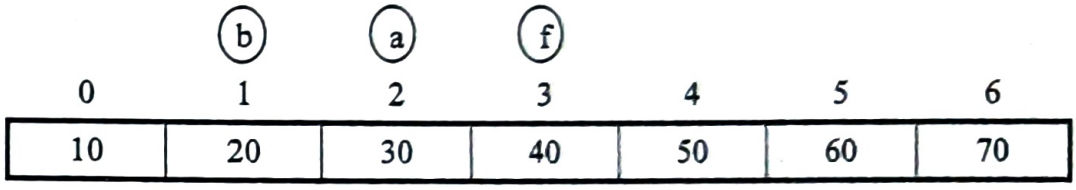
Now i = 4 $\because$ i = min (offset + b, n – 1)

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

$\therefore$ f = 3

a = 2

b = 1



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

Now new i = 5 $\because$ 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.

Q.8 Write a Python program for index sequential search.

Ans. :

```
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
```

3.3 : Types of Sorting

Q.9 Explain - internal and external sorting techniques.

[SPPU : June-22, Marks 9]

Ans. : 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.**

3.4 : General Sort Concepts

Q.10 Explain the terms - ascending order and descending order.

Ans. : 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

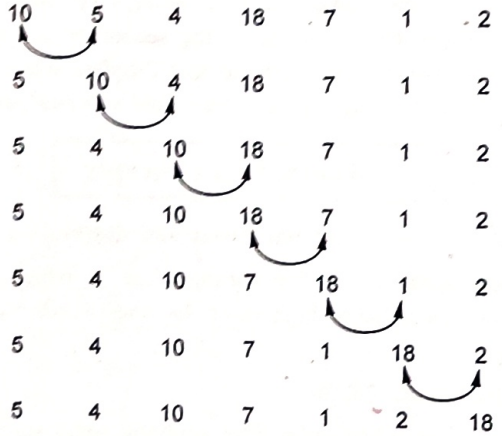
50, 40, 30, 20, 10

3.5 : Comparison based Sorting Methods

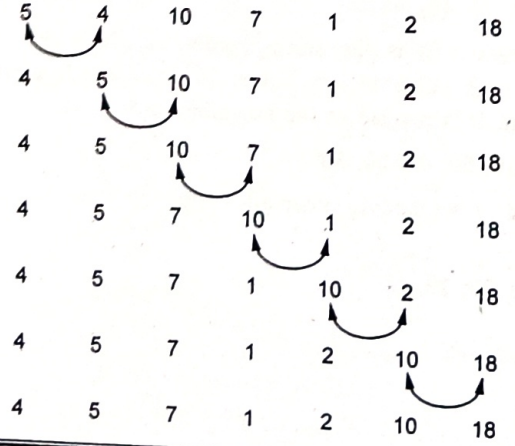
Q.11 Show the output of each pass for the following list 10, 5, 4, 18, 17, 1, 2.

Ans. : 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.

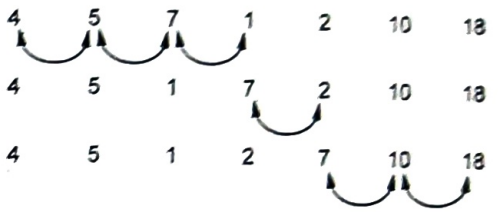
Pass I



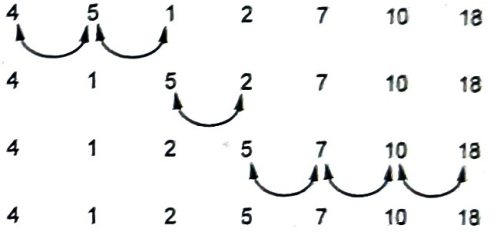
Pass II



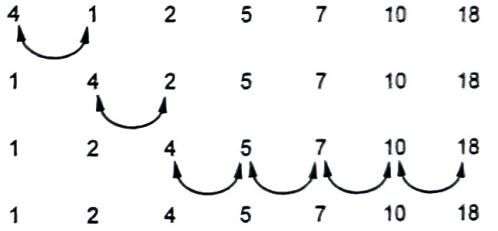
Pass III



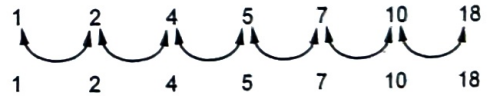
Pass IV



Pass V



Pass VI



This is the sorted list of elements.

Q.12 Write a Python program for sorting the elements using Bubble sort.

Ans. :

```
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)
```

Q.13 Write an algorithm for insertion sort.

Ans. : 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 $O(n^2)$.

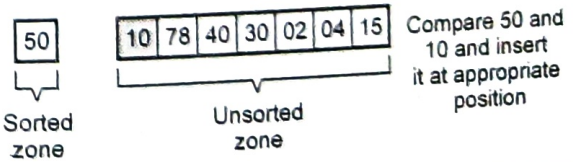
If the list of elements is arranged in descending order and if we want to sort the elements in ascending order then it results in **worst case** time complexity which is $O(n^2)$.

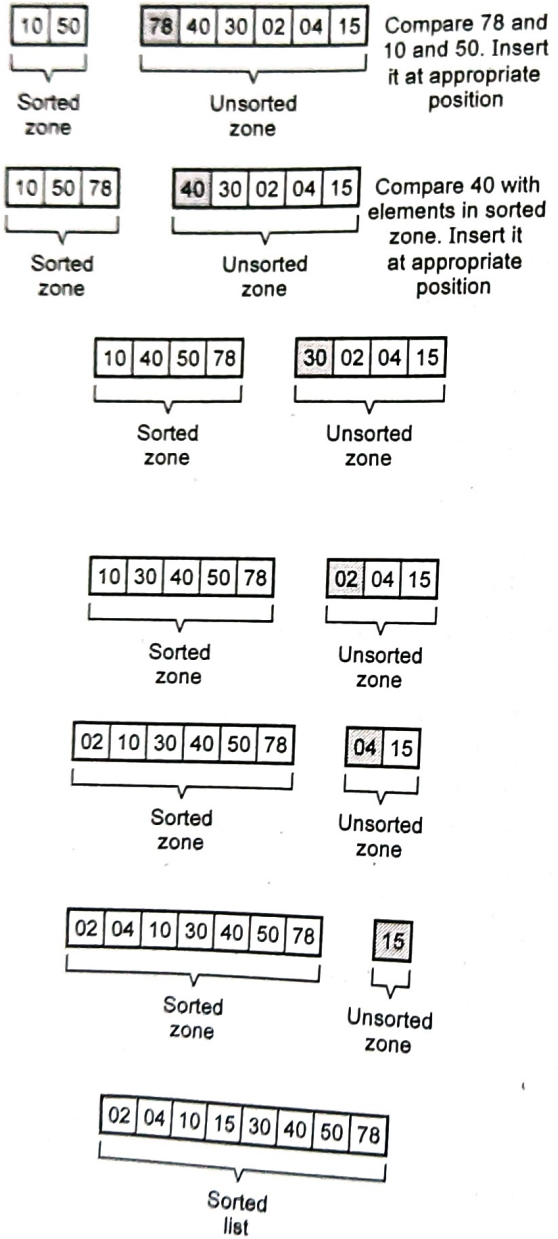
Q.14 Sort the following numbers using insertion sort. Show all passes 50, 10, 78, 40, 30, 02, 04, 15.

Ans. : Consider the list of elements as -

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

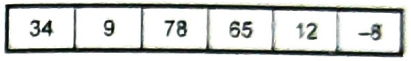
The process starts with first element.





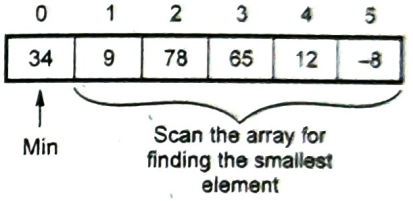
Q.15 Sort the following and show the status after every pass using selection sort : 34, 9, 78, 65, 12, – 8.

Ans. : Let

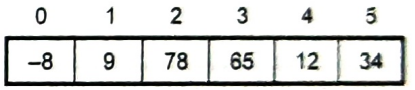


be the given elements.

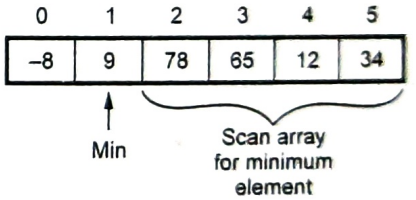
Pass 1 : Consider the elements A[0] as the first element. Assume this as the minimum element.



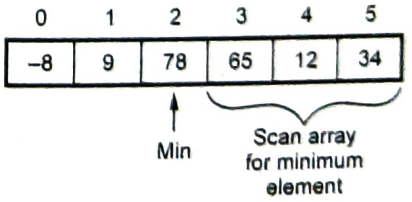
If smallest element is found, swap it with A[0]



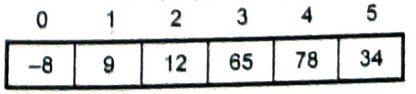
Pass 2 :



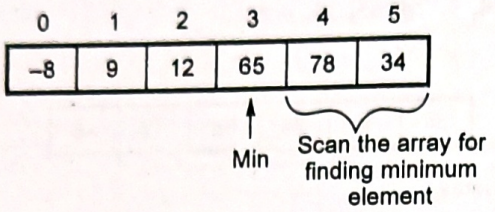
Pass 3 :

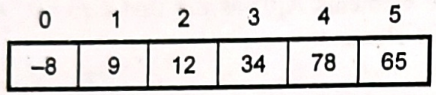


Swap A[2] and A[4]

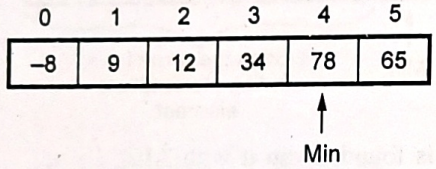


Pass 4 :

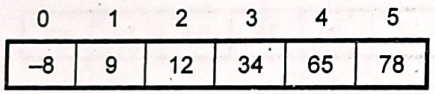




Pass 5 :



Swap A[4] and A[5]



This is a sorted list

Q.16 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.

Ans. : 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]
>>>

Q.17 Explain Shell sort method with suitable example.

Ans. : 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.

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

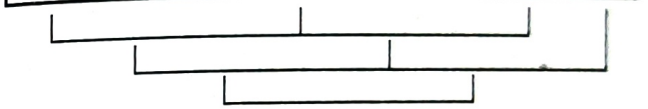
So now compare

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

(x[2], x[5])

Second iteration

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



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 $\therefore$ 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

Q.18 Sort the following data in ascending order using Radix Sort : 25, 06, 45, 60, 140, 50. ☞ **[SPPU : June-22, Marks 9]**

Ans. :

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]
>>>
```

Q.19 Write an algorithm for radix sort.

Ans. :

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.

Q.20 What is counting sort ? Explain it with suitable example.

Ans. : 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.

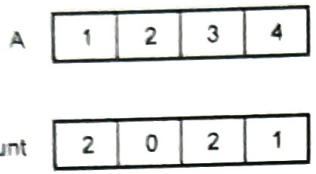
Q.21 Apply counting sort for the following numbers to sort in ascending order. 4, 1, 3, 1, 3.

Ans. : 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.

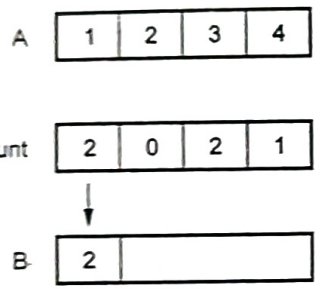
A	1	2	3	4

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.

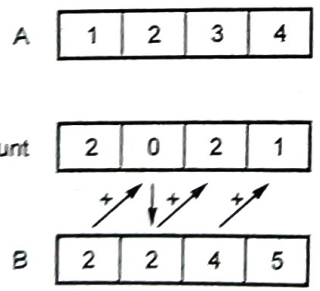


Step 2 : Now we will create another array B in which we will store sum of counts for given index.

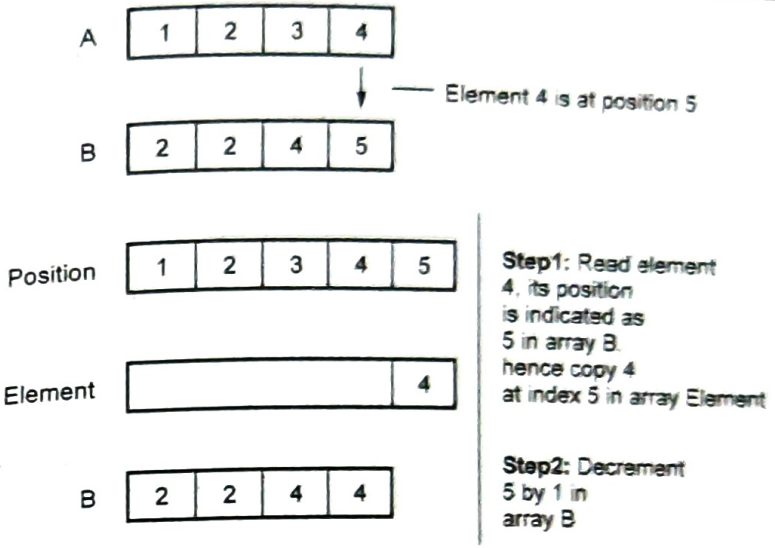


Simply copy first index value of count array to B.

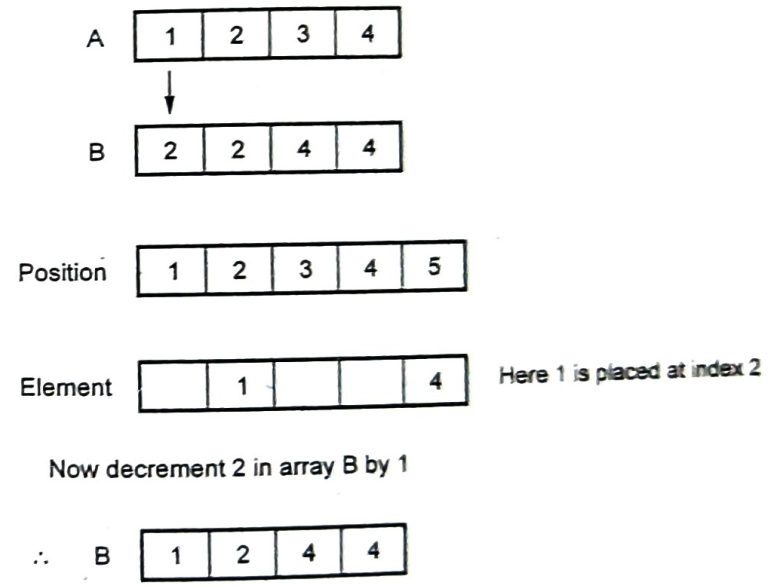
For filling up rest of the elements to B array copy the sum of previous index value of B with current index value of count array.



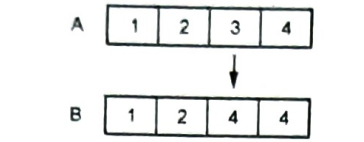
Step 3 : Now consider array A and B for creating two more arrays **Position** and **Element.**

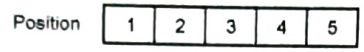


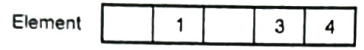
Step 4 : Next element is 1. The position of it is 2.



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

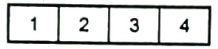




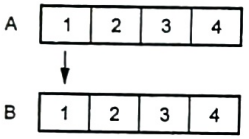


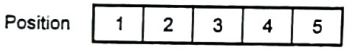
Now decrement 4 in array B by 1

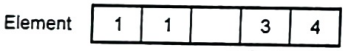
∴ B



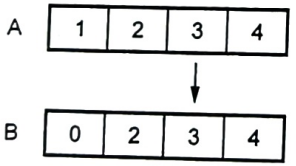
Step 6 : Next element is 1. The position of it in array B is 1.

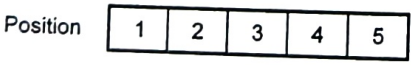


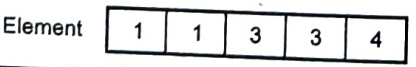




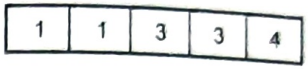
Step 7 : Next element is 3. In array B its position is 3.







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



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]
>>>
```

Q.22 What is Bucket sort algorithm ?

Ans. : 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.

Q.23 Sort the elements using bucket sort. 56, 12, 84, 56, 28, 0, – 13, 47, 94, 31, 12, – 2.

Ans. : We will set up an array as follows

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

Now sort each bucket

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

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.

3.7 : Comparison of all Sorting Methods and their Complexities

Q.24 Compare the best case, worst case and average case time complexitites of various sorting algorithms.

Ans. :

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)

END...

Unit IV

4 Linked List

4.1 : Introduction to Static and Dynamic Memory Allocation

Q.1 Give the difference of static memory and dynamic memory.

[SPPU : June-22, Marks 9]

Ans. :

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.

4.2 : Introduction to Linked List

Q.2 What is Linked List ?

Ans. : 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. Q.2.1 Structure of node

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

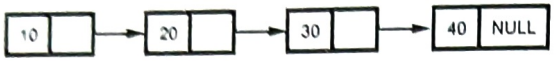
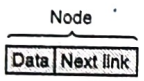
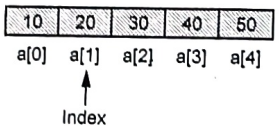


Fig. Q.2.2

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

Q.3 Give the difference between Linked list and Arrays.

Ans. :

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

Q.4 Explain the new and delete operators of dynamic memory management

Ans. : 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;

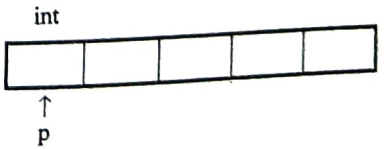
For example :

```
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

Q.5 Give an ADT for linked list.

Ans. : 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

Q.6 Give representation of linked list.

Ans. :

```
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

Q.7 Write a function to insert a node in a linked list.

[SPPU : June-22, Marks 9]

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

1) Insertion of a node as a head node

2) Insertion of a node as a last node

3) Insertion of a node after some node

```
/*
-------------------------------------------------------
    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;
 }
}
```

Q.8 Write and explain deletion operation of a node from linked list.

Ans. :

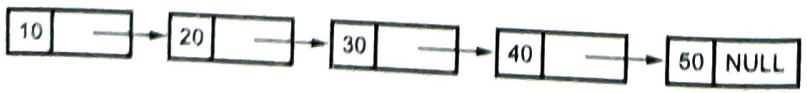
```
void sll :: dele()
{
   node *temp, *prev ;
   int  key;
   temp=head;
   clrscr();
   cout<<"\nEnter the data of the node you want to delete: ";
   cin>>key;
   while(temp!=NULL)
   {
    if(temp->data==key)
        break;
    prev=temp;
    temp=temp->next;
 }
 if(temp==NULL)
 cout<<"\nNode not 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

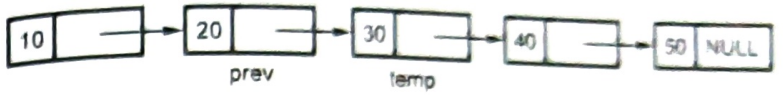
Once the node to be deleted is found then get the previous node of that node in variable **prev**

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,

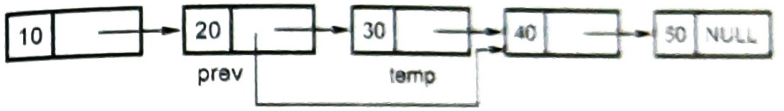


Suppose we want to delete node 30. Then we will search the node containing 30. Mark the node to be deleted as **temp**. Then we will obtain previous node of **temp**. Mark previous node as **prev**.



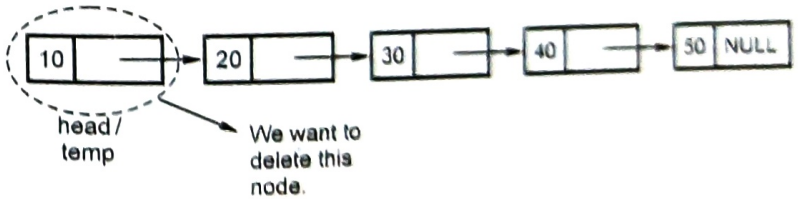
Then,

$$prev \rightarrow next = temp \rightarrow next$$

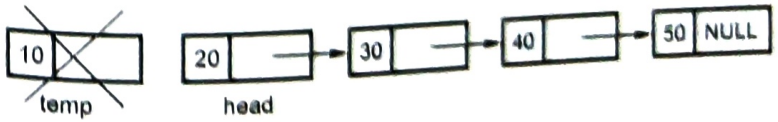


Now we will **delete** the **temp** node. Then the linked list will be

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



This can be done using following statements




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

Q.9 Write an algorithm to count the number of nodes between given two nodes in a linked list.

Ans. :

```
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 %d and %d is %d ",
temp1->data, temp2->data, count);
}
```

Q.10 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.

Ans. :

```
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.

Q.11 Write a C++ function to perform the merging of two linked lists.

Ans. :

```
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 SII1 */
         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 SII2*/
      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;
}
```

Q.12 Write a program in C++ to return the position of an element X in a list L.

Ans. : 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*/
}
```


Q.13 Write a C function to concatenate two singly linked list.

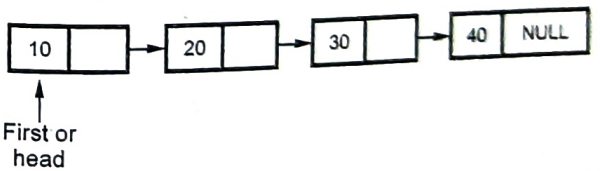
Ans. :

```
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;
  } }
```

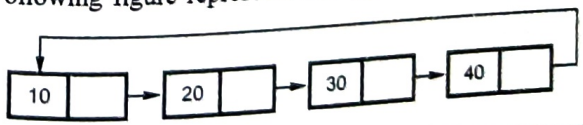
4.7 : Types of Linked List

Q.14 What are the types of linked lists ?

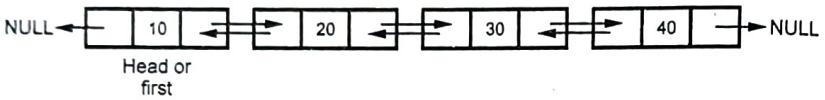
Ans. : 1. Singly linked list : It is called singly linked list because it contains only **one link** which points to the next node. The very first node is called **head** or **first.**



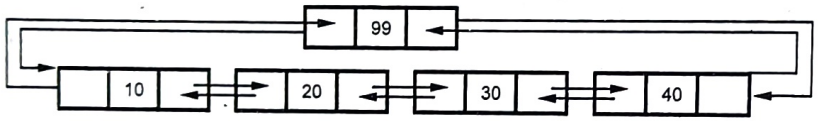
2. Singly circular linked list : In this type of linked list the next link of the last node points to the first node. Thus the overall structure looks circular. Following figure represents this type of linked list.



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

Q.15 What is doubly linked list ? Give the node structure of it.

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

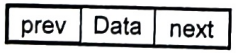
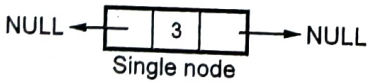


Fig. Q.15.1

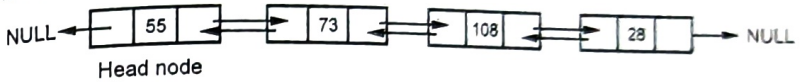


'C++' structure of doubly linked list :

```
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.



Q.16 Write a method to create a doubly linked list.

[SPPU : June-22, Marks 3]

Ans. :

```
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'||ans=='Y');
  cout<<"\nThe List is created\n";
  head=temp;
  getch();
}
```

Q.17 Write C++ function to delete a node from doubly linked list.

☞[SPPU : June-22, Marks 3]

Ans. :

```
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
```

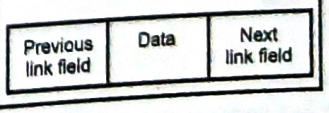


```
      {
      head=curr->next;// first node
      head->prev=NULL;
      }
    }
    else
    {
      temp=curr->prev; //intermediate or end node
      if(curr->next!=NULL)
      (curr->next)->prev=temp;
      temp->next=curr->next;
    }
      delete curr; //free memory
      cout<<"\nThe item is deleted\n";
  }

  getch();

}
```

Q.18 What is the difference between singly and doubly linked list ?

☞ [SPPU : June-22, Marks 3]

Ans. :

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.

Q.19 What are the advantages of doubly linked list over the singly linked list ?

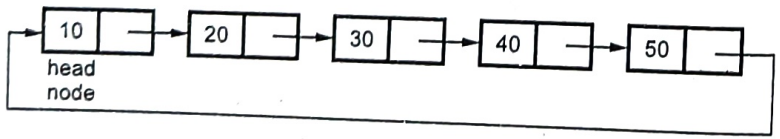
Ans. : • The doubly linked list has two pointer fields. One field is previous link field and another is next link field.

- 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.

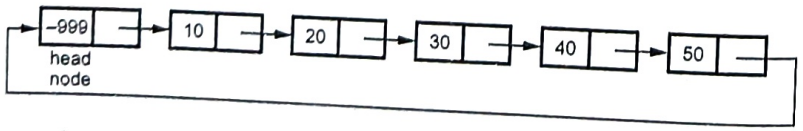
4.9 : Circular Linked List

Q.20 What is circular linked list ?

Ans. : 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.

Q.21 Write a C++ function to create a circular linked list.

Ans. :

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

Single node in the Circular list

Q.22 What is the advantage of circular linked list over singly linked list ?

Ans. : Advantage of circular 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

Q.23 What is doubly circular linked list ?

Ans. : The doubly circular linked list can be represented as follows

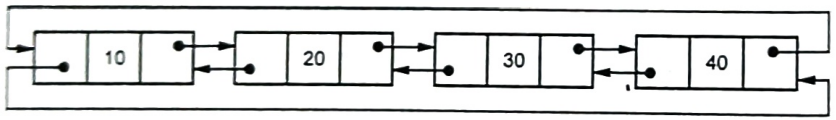


Fig. Q.23.1 Doubly circular list

The node structure will be

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

4.11 : Applications of Linked List

Q.24 What are the applications of linked list ?

Ans. : Various applications of linked list are -

1. Linked list can be used to implement linear data structures such as stacks and queues.

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

Q.25 Explain how to represent a polynomial using linked list ?

Ans. : 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.

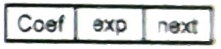


Fig. Q.25.1 Node of polynomial

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;
```

Q.26 Write a function for addition of two polynomials.

[SPPU : June-22, Marks 9]

Ans. :

```
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);
}
```

4.13 : Generalized Linked List (GLL)

Q.27 Explain the concept of Generalized Linked List with suitable example.

Ans. : 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	Data	Down pointer	Next pointer

Flag = 1 means down pointer exists.

= 0 means next pointer exists.

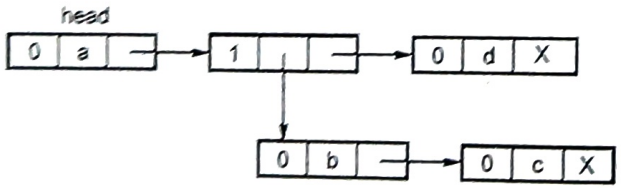
Data means the atom

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

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

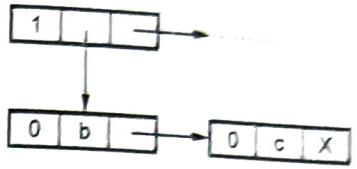
Example of GLL

(a , (b , c), d)



In above example the head node is

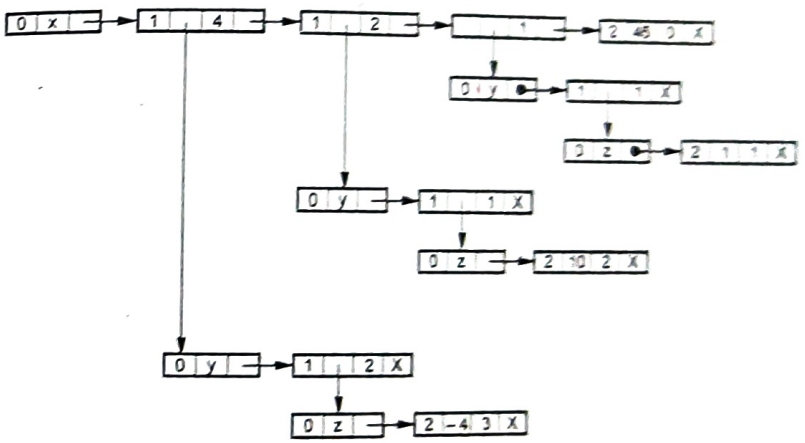
In this case the first field is 0, it indicates that the second field is variable. If first field is 1 means the second field is a down pointer, means some list is starting.



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.

Q.28 Consider $-4x^4y^2z^3 + 10x^2yz^2 + 7xyz + 45$ as a polynomial. Represent it as GLL.

Ans. :



END...

Unit V

5 Stack

5.1 : Basic Concept

Q.1 Define the term - Stack.

Ans. : A stack is an **ordered list** in which all insertions and deletions are made at **one end**, called the **top**. If we have to make stack of elements 10, 20, 30, 40, 50, 60 then 10 will be the bottommost element and 60 will be the topmost element in the stack. A stack is shown in Fig. Q.1.1.

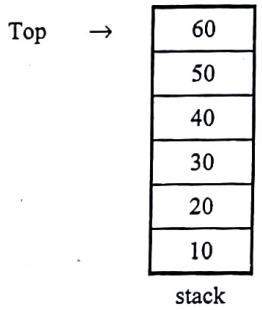


Fig. Q.1.1 Stack

5.2 : Stack Abstract Data Type

Q.2 Write an ADT for Stack.

Ans. : Stack is a data structure which posses LIFO i.e. Last In First Out property. The abstract data type for stack can be as given below.

Abstract DataType stack

{

Instances : Stack is a collection of elements in which insertion and deletion of elements is done by one end called top.

Preconditions :

1. Stfull () : This condition indicates whether the stack is full or not. If the stack is full then we cannot insert the elements in the stack.

2. Stempty () : This condition indicates whether the stack is empty or not. If the stack is empty then we cannot pop or remove any element from the stack.

Operations :

1. Push : By this operation one can push elements onto the stack. Before performing push we must check stfull () condition.
2. Pop : By this operation one can remove the elements from stack. Before popping the elements from stack We should check stempty () condition.

}

5.3 : Representation of Stack using Sequential Organization

Q.3 What are the two methods of representing a stack using sequential organization? Explain them with suitable examples.

Ans. : Declaration 1 :

```
#define size 100
int stack[size], top = -1;
```

In the above declaration stack is nothing but an array of integers. And most recent index of that array will act as a top.

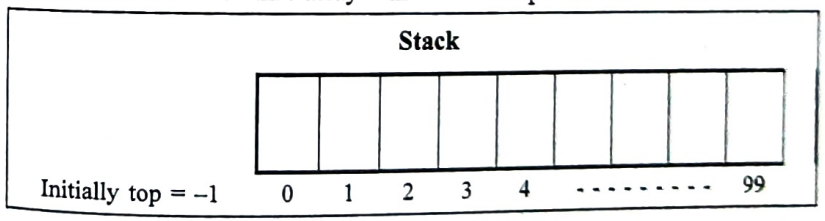


Fig. Q.3.1 Stack using one dimensional array

The stack is of the size 100. As we insert the numbers, the top will get incremented.

Declaration 2 :

```
#define size 10
struct stack {
        int s[size];
```

```
    int top;
}st;
```

In the above declaration stack is declared as a structure.

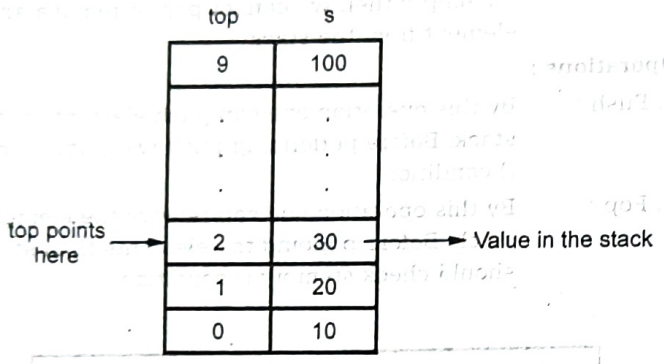


Fig. Q.3.1 Stack using structure

5.4 : Stack Operations

Q.4 Explain the push and pop operations of stack.

[SPPU : June-22, Marks 8]

Ans. : 1. Push

Push is a function which inserts new element at the top of the stack. The function is as follows.

```
void push(int item)
{
st.top++; / * top pointer is set to next location */
st.s[st.top] =item; /* placing the element at that location */
}
```

The push function takes the parameter **item** which is actually the element which we want to insert into the stack - means we are pushing the element onto the stack.

2. Pop

It deletes the element at the top of the stack. The function pop is as given below -

Note that always top element is to be deleted.


```
int pop()
{
int item;
item = st.s[st.top];
st.top - -;
return(item);
}
```

Push operation can be shown by following Fig. Q.4.1.

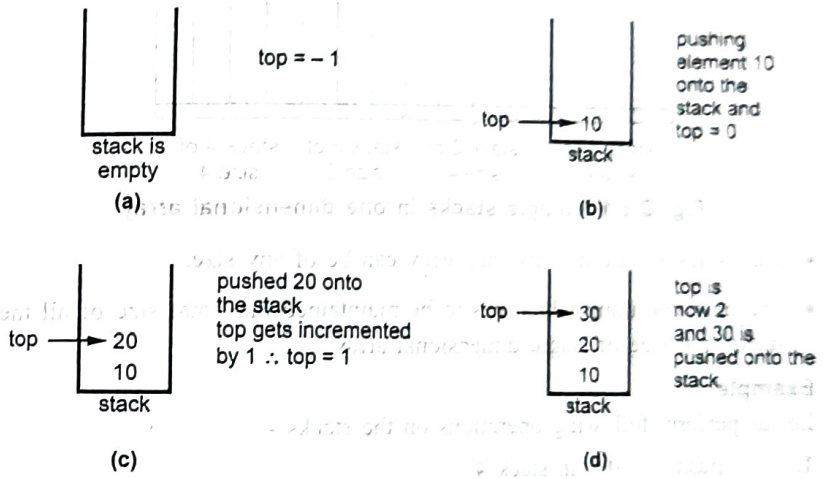


Fig. Q.4.1 Performing push operation

The pop operation can be shown by following Fig. Q.4.2.

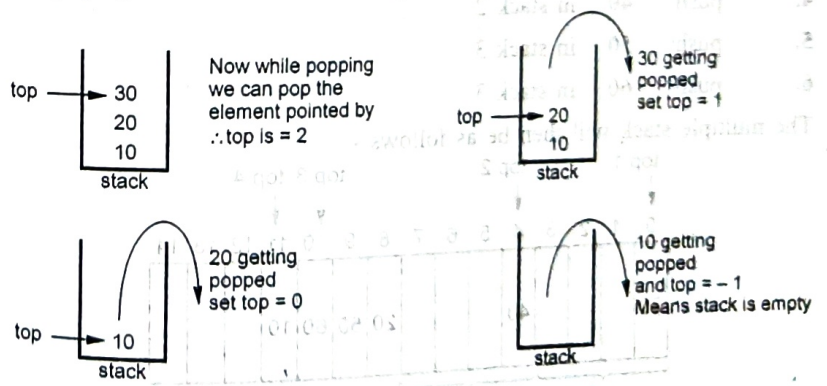


Fig. Q.4.2 Performing pop operation

5.5 : Multiple Stacks

Q.5 Explain the concept of multiple stacks with suitable example.

Ans. : • In a single array any number of stacks can be adjusted. And push and pop operations on each individual stack can be performed.

- The following Fig. Q.5.1 shows how multiple stacks can be stored in a single dimensional array

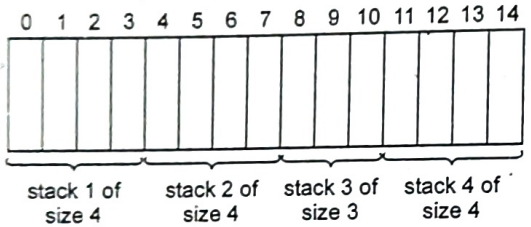


Fig. Q.5.1 Multiple stacks in one dimensional array

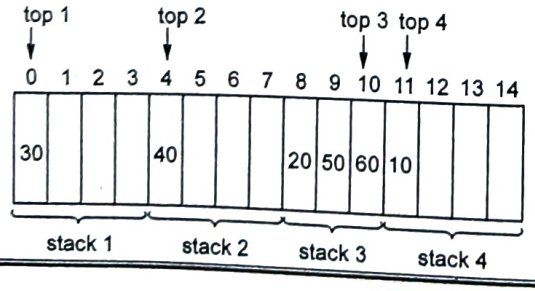
- Each stack in one dimensional array can be of any size.
- The only one thing which has to be maintained that total size of all the stacks < = size of single dimensional array.

Example

Let us perform following operations on the stacks -

1. push 10 in stack 4
2. push 20 in stack 3
3. push 30 in stack 1
4. push 40 in stack 2
5. push 50 in stack 3
6. push 60 in stack 3

The multiple stack will then be as follows -



- Here stack 3 is now full and we can not insert the elements in stack 3.
- The top 1, top 2, top 3, top 4 indicates the various top pointers for stack 1, stack 2, stack 3 and stack 4 respectively.
  - **stack 1**-array [0] will be lower bound and array [3] will be upper bound. That is we can perform stack 1 operation for array [0] to array [3] only.
  - **stack 2**-The area for stack 2 will be from array [4] to array [7]
  - **stack 3**-From array [8] to array [10]
  - **stack 4**-From array [11] to array [14]

Q.6 Give a pseudo code for implementing two stacks in a single array.

Ans. : Two stacks can be adjusted in a single array with n numbers, which is as shown below

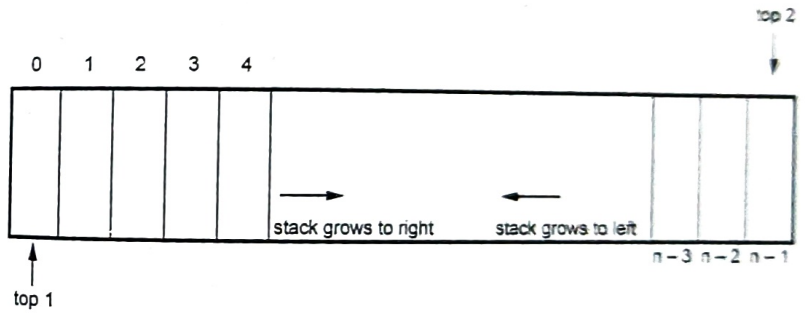


Fig. Q.6.1 Two stacks in single array

- One stack starts at the leftmost end of array and other at the rightmost end of array.
- The insertion in stack1 moves top1 to right while insertion in stack2 moves **top2** to left.
- When stack is full both the **top1** and **top2** positions are adjacent to each other.
- The **advantage** of this arrangement is that every location of the array can be utilized.

- The implementation routines are -

```
# define MAX 80
int stack[MAX];
int top1=-1, top2=n;
void push (int item, int stackno)
{
   if(stackno == 1)
   {
   /* pushing in stack1 */
      if(top1+1 == top2)
         cout<<"stack1 is full";
      top1++;
      stack [top1]=item;
   }
   else
   {
      if (top2-1 == top1)
      cout<<"stack2 is full";
      top2--
      stack[top2]=item;
   }
int pop (int stackno)
{
   int item;
   if (stackno == 1)
   {
      if (top1 == - 1)
         cout<<"stack1 is empty";
      item = stack [top 1];
      top1++
      return(item);
   }
   else
```



```
   {
      if (top2 == MAX)
         cout<<"stack2 is empty";
      item = stack [top2];
      top2--
      return(item);
   }
}
```

5.6 : Applications of Stack

Q.7 Enlist the applications of stack.

Ans. : Various applications of stack are -

1. Stack is used for converting one form of expression to another.
2. Stack is also useful for evaluating the expression.
3. Stack is used for parsing the well formed parenthesis.
4. Decimal to binary conversion can be done by using stack.
5. The stack is used for reversing the string.
6. In recursive routines for storing the function calls the stack is used.

5.7 : Polish Notation and Expression Conversion

Q.8 Write an algorithm for conversion of infix to postfix expression.

Ans. : The algorithm is as follows -

Read an expression from left to right each character one by one

1. If an operand is encountered then store it in postfix array.
2. If '(' is read, then simply push it onto the stack. Because the (has highest priority when read as an input.
3. If ')' is reads, then pop all the operators until (is read. Discard (. Store the popped characters in the postfix array.
4. If operator is read then
   1. If instack operator has greatest precedence (or equal to) over the incoming operator then pop the operator and add it to postfix

expression(postfix array). Repeat this step until we get the instack operator of higher priority than the current incoming operator. Finally push the incoming operator onto the stack.

2. Else push the operator.

5. If stack is not empty then pop all the operators and store in postfix array.

6. Finally print the array of postfix expression.

Q.9 Write a function to convert an infix to postfix expression.

☞ **[SPPU : June-22, Marks 9]**

Ans. :

```
void postfix()
{
   int i,j=0;
   char ch,next_ch;
   for(i=0;i<strlen(in);i++)
   {
      ch= in[i];
      switch(ch)
      {
         case '(': push(ch);
                   break;
         case ')':while((next_ch=pop())!='(')
                     post[j++] = next_ch;
                   break;
         case '+':
         case '-':
         case '*':
         case '/':
         case
        '^':while(!stempty()&&(precedence(stk[top])>=precedence(ch)))
                     post[j++]=pop();
                  push(ch);//else part
                  break;
        default:post[j++]=ch;//if operand is encountered simply
        add it to postfix
```



```
      }
   }
   while(!stempty())
      post[j++]=pop();
   post[j]='\0';
   for(i=0;i<strlen(post);i++)
      printf("%c",post[i]);
}
```

Q.10 Give the postfix and prefix expression - (a+b*c)/(x+y/z)

Ans. : Infix to postfix conversion

Input read	Action	Stack	Output
(	Push (	(	
a	Print a	(	a
+	Push +	(+	a
b	Print b	(+	ab
*	Push	(+ *	ab
c	Print c	(+ *	abc
)	Pop *, Print. Pop +, Print Pop (		abc * +
/	Push /	/	abc * +
(	Push (	/ (	abc * +
x	Print x	/ (	abc * + x
+	Push +	/ (+	abc * + x
y	Print y	/ (+	abc * + xy
/	Push /	/ (+ /	abc * + xy
z	Print z	/ (+ /	abc * + xyz
)	Pop /, Print. Pop +, Print Pop (	/	abc * + xyz / +
end of input	Pop /, Print		**abc * + xyz / + /** is required postfix expression.

Infix to Prefix Conversion

Reverse the input as) z/y + x (/) c * b + a (and then read each character one at a time.

Input read	Action	Stack	Output
)	Push)	)	
z	Print z	)	z
/	Push	) /	z
y	Print y	) /	zy
+	Pop /, Print push +	) +	zy /
x	Print x	) +	zy / x
(	Pop +, Print + Pop)		zy / x +
/	Push /	/	zy / x +
)	Push	/)	zy / x +
c	Print c	/)	zy / x + c
*	Push *	/) *	zy / x + c
b	Print b	/) *	zy / x + c
+	Pop *, Print it push +	/)	zy / x + cb *
a	Print	/) +	zy / x + cb * a
(	Pop +, Print + Pop)	/	zy/ x + cb * a +
end of input	Pop/, Print it	empty	zy / x + cb * a + /
	Reverse the output and print it.		/ + a * bc + x / yz is prefix expression.

Q.11 Convert the following expression into postfix form. Show all the steps and stack content : 4$2*3−3+8/4(1+1)

Ans. :

Input symbol	Action	Stack	Postfix expression
4	Print 4	empty	4
$	Push $	$	4

2	Print 2	$	42
*	POP $, Print it. Push *	*	42 $
3	Print 3	*	42 $ 3
−	POP *, Print it Push −	−	42 $ 3*
3	Print it	−	42 $ 3*3
+	POP −, Print it. Then push +	+	42 $ 3*3 −
8	Print 8	+	42 $ 3*3 − 8
/	Push /	+ /	42 $ 3*3 − 8
4	Print 4	+ /	42 $ 3*3 − 84
/	POP/, Print it Then Push/	+ /	42 $ 3*3 − 84 /
(	Push (	+ / (	42 $ 3*3 − 84 /
1	Print 1	+ / (	42 $ 3*3 − 84 / 1
+	Push +	+ / (+	42 $ 3*3 − 84 / 1
1	Print 1	+ / (+	42 $ 3*3 − 84 / 11
)	POP +, Print it POP (	+ /	42 $ 3*3 − 84 / 11 +
end of input	POP/, Print POP +, Print	empty	42 $ 3*3 − 84 / 11 + / +

The postfix expression is **42$3*3 − 84/11+/+**

5.8 : Postfix Expression Evaluation

Q.12 Write an algorithm for evaluation of postfix expression.

Ans. :

1. Read the postfix expression from left to right.

2. If the input symbol read is an operand then push it on to the stack.
3. If the operator is read POP two operands and perform arithmetic operations if operator is
   - ^ then result = operand 1 ^ operand 2
   - - then result = operand 1 - operand 2
   - * then result = operand 1 * operand 2
   - / then result = operand 1 / operand 2
4. Push the result onto the stack.
5. Repeat steps 1-4 till the postfix expression is not over.

Q.13 Write a function for evaluation of postfix expression.

Ans. :

```
double EVAL::post(char exp[])
{
   char  ch,*type;
   double  result, val, op1, op2;
   int i;
   st.top = 0;
   i=0;
   ch = exp[i];
   while ( ch !='$' )
   {
      if ( ch >= '0' && ch <= '9')
      type ="operand";

      else if ( ch == '+' || ch == '-'||
       ch == '*'|| ch == '/' ||
       ch == '^' )
      type="operator";
      if( strcmp(type,"operand")==0)/*if the character is operand*/
      {
      val = ch - 48;
      push(val);
      }
```

The characters '0', '1', ... '9' will be converted to their values, so that they will perform arithmetic operation.


```
      else

      if (strcmp(type,"operator")==0)/*if it is operator*/
      {
       op2 = pop();
       op1 = pop(); //popping two operands to perform
                    arithmetic operation
       switch(ch)
      {
      case '+' : result = op1 + op2;
                    break;
      case '-' : result = op1 - op2;
                    break;
      case '*' : result = op1 * op2;
                    break;
      case '/' : result = op1 / op2;
                    break;
      case '^' : result = pow(op1,op2);
                    break;
   }/* switch */
   push(result);
   }
      i++;
      ch=exp[i];
   } /* while */
   result = pop(); /*pop the result*/
   return(result);
}
```

Finally result will be pushed onto the stack.

5.9 : Linked Stack and Operations

Q.14 Explain the linked implementation of stack.

[SPPU : June-22, Marks 4]

Ans. : • The advantage of implementing stack using linked list is that we need not have to worry about the size of the stack.

- Since we are using linked list as many elements we want to insert those many nodes can be created. And the nodes are dynamically getting created so there won't be any stack full condition.
- The typical structure for linked stack can be

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

- Each node consists of data and the next field. Such a node will be inserted in the stack. Following figure represents stack using linked list.

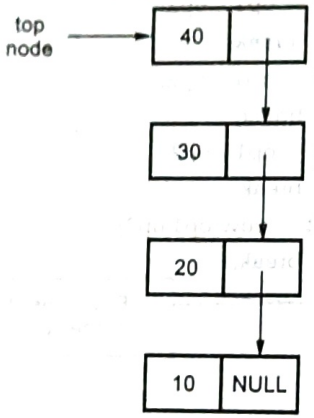


Fig. Q.14.1 Representing the linked stack

Q.15 Write functions for Push and pop operations using the linked implementation. **☞[SPPU : June-22, Marks 4]**

Ans. :

```
/*
    ------------------------------------------
    The Push function
    ------------------------------------------
*/
void Lstack::Push(int Item, node **top)
{
```



```
    node *New;
    New = new node;
    New->data=Item;
    New -> next = *top;
    *top = New;
}
/*
    ------------------------------------------
    The Pop function
    ------------------------------------------
*/
int Lstack::Pop(node **top)
{
    int item;
    node *temp;
    item = (*top) ->data;
    temp = *top;
    *top = (*top) -> next;
    delete temp;
    return(item);
}
```

5.10 : Recursion-Concept

Q.16 What is recursion ? Also enlist the properties of Recursion.

Ans. : Definition : Recursion is a programming technique in which the function calls itself repeatedly for some input.

By recursion the same task can be performed repeatedly.

Properties of Recursion

Following are two fundamental principles of recursion

1. There must be at least one condition in recursive function which do not involve the call to recursive routine. This condition is called a "way out" of the sequence of recursive calls. The is called **base case property.**

2. The invoking of **each recursive call must reduce** to some manipulation and must go closer to base case condition.

Q.17 Generate n^{th} term Fibonacci sequence with recursion.

Ans. :

```
#include<iostream>
using namespace std;
int fib(int n)
{
   int x,y;
   if(n<=1)
      return n;
   x=fib(n-1);
   y=fib(n-2);
   return (x+y);
}
int main(void)
{
   int n,num;
   cout<<"\n Enter location in fibonacci series: ";
   cin>>n;
   num=fib(n);
   cout<<"\n The number at "<<n<<" position is "<<num<<" in
   fibonacci series";
   return 0;
}
```

Output

```
Enter location in fibonacci series: 3
The number at 3 position is 2 in fibonacci series
```

Q.18 Explain how stack plays an important role in implementing a recursive function.

Ans. : For illustrating the use of stack in recursive functions, we will consider an example of finding factorial. The recursive routine for obtaining factorial is as given below -

```
int fact (int num)
{
```



```
   int a, b;
   if (num = = 0)
      return 1 ;
   else
   {
      a = num - 1 ;
      b = fact (a) ;
      f = a * b;
      return f ;
   }
}
/* Call to the function */
ans = fact (4) ;
```

For the above recursive routine we will see the use of stack. For the stack the only principle idea is that – when there is a call to a function the values are pushed onto the stack and at the end of the function or at the return the stack is popped

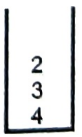
Suppose we have num = 4. Then in function **fact** as num ! = 0 else part will be executed. We get

a = num – 1

a = 3

And a recursive call to fact (3) will be given. Thus internal stack stores

Next fact (2) will be invoked. Then

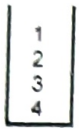


∵ a = num – 1

a = 3 – 1

b = fact (2)

Then,



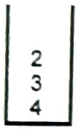
$\because a = num - 1$

$a = 2 - 1$

$b = fact\ (1)$

Now next as num = 0, we return the value 1.

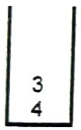
$\therefore a = 1, b = 1$, we return $f = a * b = 1$ from function **fact**. Then 1 will be popped off.



Now the top of the stack is the value of a

Then we get $b = fact\ (2) = 1$

$\therefore f = a * b = 2 * 1 = 2$ will be returned.



Here a = 3

Then $b = fact\ (3)$

$b = 2$

$\therefore f = a * b = 6$ will be returned.

The stack will be

Here a = 4

Then $b = fact\ (4)$

$= 6$

$\therefore f = a * b = 24$ will be returned.

As now stack is empty. We return $f = 24$ from the function **fact**.

Q.19 What direct and indirect recursion ?

☞ [SPPU : June-22, Marks 9]

Ans. : 1. Direct Recursion

Definition : A C function is directly recursive if it contains an explicit call to itself.

For example

```
int fun1(int n)
{
   if(n<=0)
      return n;
   else
      return fun1(n-1);
}
```

Call to itself

2. Indirect Recursion

Definition : A C function is indirectly recursive if function1 calls function2 and function2 ultimately calls function1.

For example

```
int fun1(int n)
{
   if(n<=0)
      return n;
   else
      return fun2(n);
}
int fun2(int m)
{
   return fun1(n-1);
}
```

Q.20 What is recursive tree ? Explain.

Ans. : A tree recursion is a kind of recursion in which the recursive function contains the pending operation that involves another recursive call to the function.

The implementation of **fibonnaci** series is a classic example of tree recursion.

C Function

```
int fib(int n)
{
    if(n==0)
        return 0;
    if(n==1)
        return 1;
    return fib(n-1)+fib(n-2);
}
```

```
fib(4)
= fib(3) + fib(2)
= [fib(2) + fib(1)] + [fib(1) + fib(0)]
= [(fib(1) + fib(0)] + [1] + [1 + 0]
= [1+0] + 1 + 1
fib(4) = 3
```

The call tree can be represented as follows

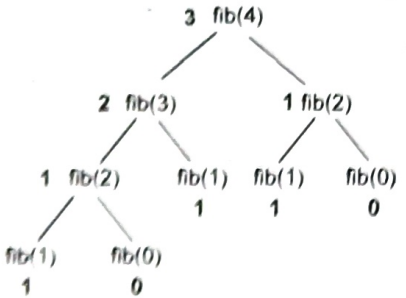


Fig. Q.20.1 Recursive tree

5.11 : Backtracking Algorithmic Strategy

Q.21 What is backtracking ?

Ans. : • *Backtracking is a method in which*

1. *The desired solution is expressible as an n tuple* $(x_1, x_2, \ldots x_n)$ *where* x_i *is chosen from some finite set* S_i.
2. *The solution maximizes or minimizes or satisfies a criterion function C* $(x_1, x_2, \ldots x_n)$

- The basic idea of backtracking is to build up a **vector**, one component at a time and to test whether the vector being formed has any chance of success.
- The major **advantage** of backtracking algorithm is that we can realize the fact that the partial vector generated does not lead to an optimal solution. In such a situation that vector can be ignored.
- Backtracking algorithm determines the solution by systematically searching the solution space (i.e. set of all feasible solutions) for the given problem.

Q.22 Define the terms - problem state, state space, solution states, Answer states, dead node.

Ans. : Backtracking algorithms determine problem solutions by systematically searching for the solutions using **tree structure.**

For example -

Consider a 4-queen's problem. It could be stated as "there are 4 queens that can be placed on 4×4 chessboard. Then no two queens can attack each other".

Following Fig. Q.22.1 shows tree organization for this problem.

- Each node in the tree is called **a problem state.**
- All paths from the root to other nodes define the **state space of** the problem.
- The solution states are those problem states s for which the path from root to s defines a **tuple** in the solution space.

  In some trees the leaves define the **solution states.**
- **Answer states** : These are the leaf nodes which correspond to an element in the set of solutions, these are the states which satisfy the implicit constraints.

For example- Refer Fig. Q.22.1 (See Fig. Q.22.1 on next page)

- A node which is been generated and all whose children have not yet been generated is called **live node.**
- The live node whose children are currently being expanded is called **E-node.**

- A **dead node** is a generated node which is not to be expanded further or all of whose children have been generated.

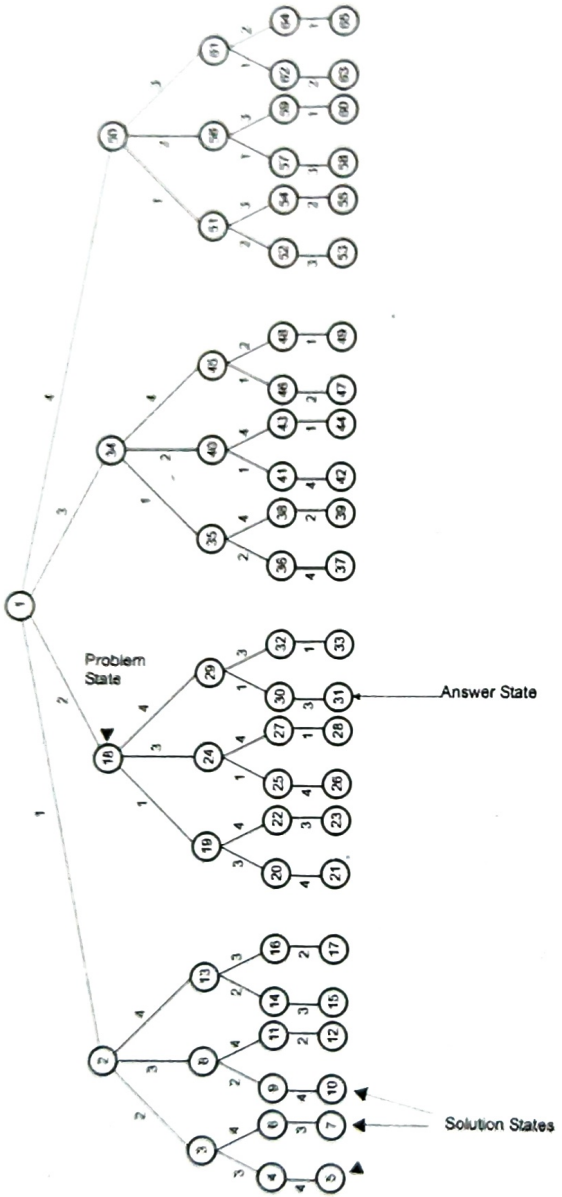


Fig. Q.22.1

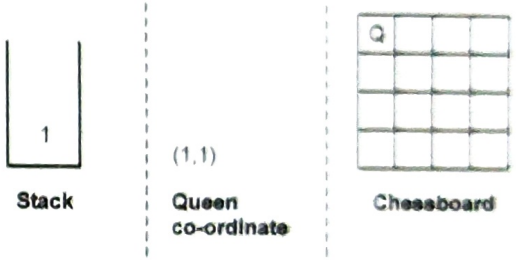
5.12 : Use of Stack in Backtracking

Q.23 With suitable example, explain how stack can be used in backtracking ?

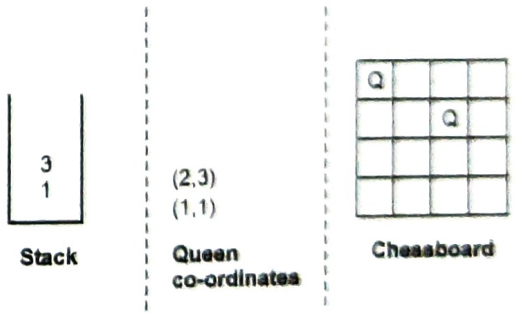
Ans. : The stack can be used in backtracking to handle the recursive procedures.

Let us consider, 4 Queen's problem once again to understand the role of stack in backtracking.

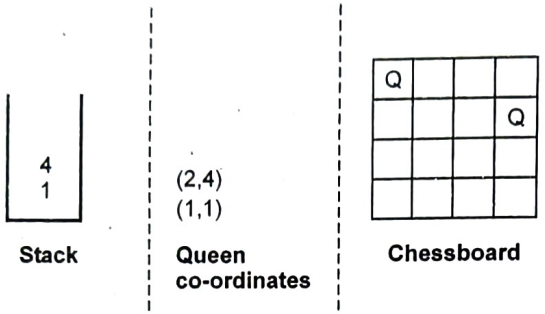
Step 1 : Place queen 1 on 1st row 1st column push only column number onto the stack.



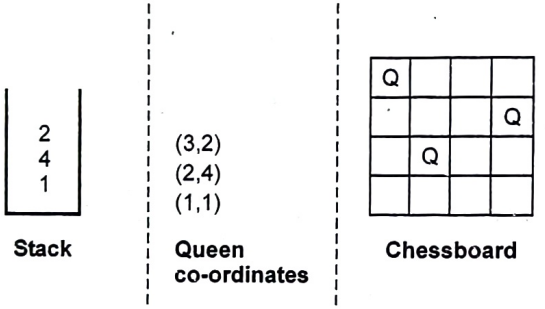
Step 2 : Then place 2 at 2nd row, 3rd column.



Step 3 : This is dead end, because 3rd queen can not be placed in next column as there is no acceptable poistion for queen 3. Hence algorithm backtrack by popping the 2nd queen's position. Now let us place queen 2 at 4th column.

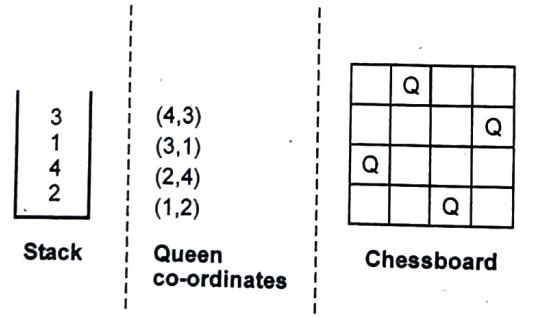


Step 4 : Place 3rd Queen at 2nd column.



Step 5 : Now, there is no valid place for queen 4. So we needed to backtrack all the moves by popping the column index from the stack to try out other possible places.

Step 6 : Finally the possible solution can be.



Algorithm :

1. Start with a queen from first row, first column and search for valid position.
2. If we find a valid position in current row, push the position (i.e. Column number) onto the stack. Then start again on next row.
3. If we don't find a valid position in the current row then we backtrack to previous row i.e. POP the column position for previous row from the stack and search for a valid position.
4. When the stack size is equal to n (for n queens) then that means we have placed n queens on the board. This is a solution to n queen's problem.

END...

Unit VI

6 Queue

6.1 : Basic Concept

Q.1 What is queue ? ☞ **[SPPU : June-22, Marks 3]**

Ans. : Definition : The queue can be formally defined as ordered collection of elements that has two ends named as **front** and **rear**. From the front end one can delete the elements and from the rear end one can insert the elements.

For example :

The typical example can be a queue of people who are waiting for a city bus at the bus stop. Any new person is joining at one end of the queue, you can call it as the rear end. When the bus arrives the person at the other end first enters in the bus. You can call it as the front end of the queue.

Following Fig. Q.1.1 represents the queue of few elements.

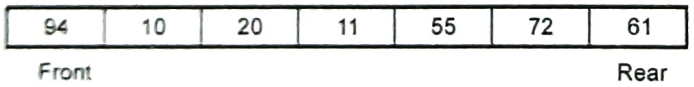


Fig. Q.1.1 Queue

Q.2 Compare Stack and Queue.

Ans. :

Sr. No.	Stack	Queue
1.	The stack is a LIFO data structure. That is the element which is inserted last will be removed first from the stack.	The queue is a FIFO data structure. That means the element which is inserted first will be removed first.
2.	The insertion and deletion of the elements in the stack is done from only one end, called top.	The insertion of the element in the queue is done by the end called rear and the deletion of the element from the queue is done by the end called front.

6.2 : Queue as Abstract Data Type

Q.3 Give an ADT for Queue.

Ans. : The ADT for queue is as given below -

AbstractDataType Queue

{

Instances :

Que[MaX] is a finite collection of elements in which insertion of element is by rear end and deletion of element is by front end.

Precondition :

The front and rear should be within the maximum size MAX.

Before insertion operation, whether the queue is full or not is checked.

Before any deletion operation, whether the queue is empty or not is checked.

Operations :

1. Create() - The queue is created by declaring the data structure for it.
2. insert() - The element can be inserted in the queue by rear end.
3. delet() - The element at front end deleted each time.
4. Display() - The elements of queue are displayed from front to rear.

}

6.3 : Queue Operations

Q.4 Explain the insertion of element in queue implemented using arrays.

Ans. : The insertion of any element in the queue will always take place from the rear end.

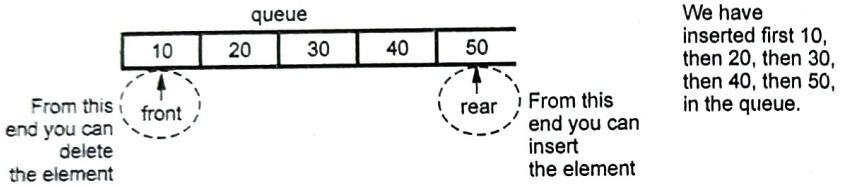


Fig. Q.4.1 Representing the insertion

```
int MyQ::insert(int item)
{
 if(Q.front ==-1)
 Q.front++;
 Q.que[++Q.rear] = item;
 return Q.rear;
}
```

This condition will occur initially when queue is empty

Always increment the rear pointer and place the element in the queue.

Q.5 Explain the deletion of element from queue implemented using arrays.

Ans. : The deletion of any element in the queue takes place by the front end always.

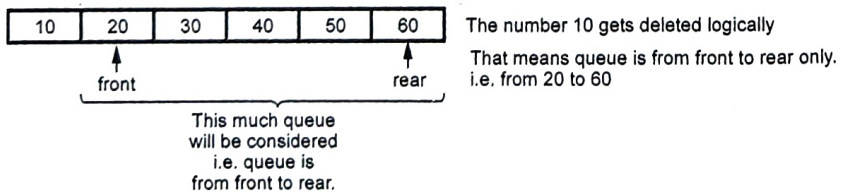
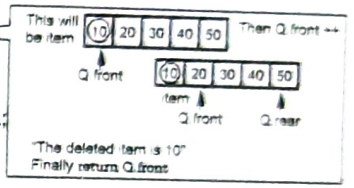


Fig. Q.5.1 Representing the deletion


```
int MyQ::delet()

{
 int item;
 item = Q.que[Q.front];
 Q.front++;
 cout<<"\n The deleted item is "<<item;
 return Q.front;
}
```



6.4 : Circular Queue

Q.6 Explain the concept of circular queue.

☞ **[SPPU : June-22, Marks 3]**

Ans. : Circular queue is a queue in which read and front are adjacent to each other. It can be represented as follows -

For rear and front pointers following formula is used

rear = (rear + 1)%SIZE
front = (front + 1)%SIZE
where SIZE represents the SIZE of a queue.

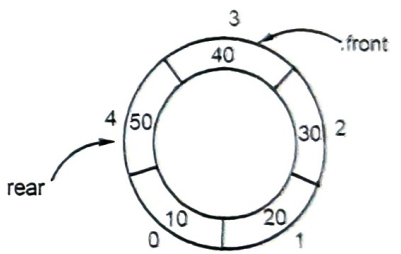


Fig. Q.6.1 Circular queue

Q.7 Implement insert and delete operations of circular queue.

OR Write Pseudo C++ code to implement circular queue using arrays. ☞**[SPPU : June-22, Marks 9]**

Ans. :

```
/*
-----------------------------------------------------
insert function
-----------------------------------------------------
*/

void Queue::insert(int item)
```

```
{
 if (front = =(rear + 1)%MAX)
 {
    cout << "Queue is full\n";
 }
 else
 {
    //setting front pointer for a single element in Queue
    if(front = =-1)
    front=rear=0;
    else
    rear=(rear+1)%MAX;
    Que[rear]=item;
 }
}
/*
-----------------------------------------------------------
delet function
-----------------------------------------------------------
*/
int Queue::delet()
{
 int val;
 if(front==-1)
 {
    cout << "Queue is empty\n";
    return 0; // return null on empty Queue
 }
 val= Que[front];//item to be deleted
 if(front==rear)//when single element is present
 {
    front=rear=-1;
 }
 else
     front=(front+1)%MAX;
 return val;
}
```


6.5 : Multi-queues

Q.8 Write a short note on - Multiple queues.

Ans. : • One of the application of queues is categorization of data. And multiple queues can be used to store variety of data. We can implement multiple queues using single dimensional arrays.

- In a one dimensional array, multiple queues can be placed. Insertion from its rear end and deletion from its front end can be possible for desired queue. Refer Fig. Q.8.1

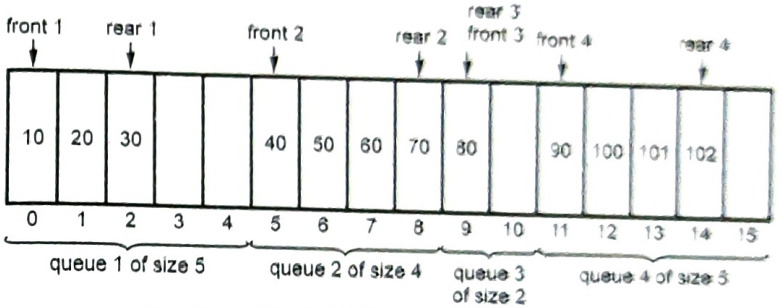


Fig. Q.8.1 Multiple queues using single array

- There are four queues having their own front and rear positioned at appropriate points in a single dimensional array.
- We can perform insertion and deletion of any element for any queue. We can declare the messages "queue full" and "queue empty" at appropriate situations for that particular queue.

6.6 : Linked Queue and Operations

Q.9 Write a routine to insert an element in a linked queue.

[SPPU : June-22, Marks 9]

Ans. :

```
void Lqueue::insert()
{
 char ch;
 Q *temp;
```

```
clrscr();
temp =new Q;//allocates memory for temp node
temp->next=NULL;
cout<<"\n\n\n\tInsert the element in the Queue\n";
cin>>temp->data;

if(front == NULL)//creating first node
{
 front= temp;
 rear=temp;
}
else                    //attaching other nodes
 {
  rear->next=temp;
  rear=rear->next;
  }
}
```

6.7 : Dequeue

Q.10 Explain the concept of Dequeue with example.

Ans. : Dequeue is a data structure in which we can insert the element both by front and rear end. Similarly we can delete the element from deque by both the rear and front ends.

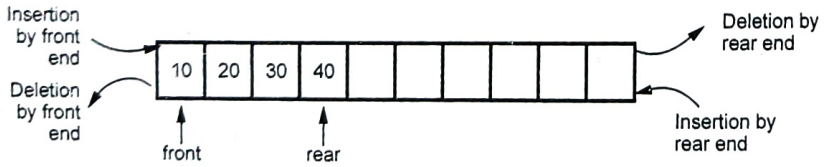
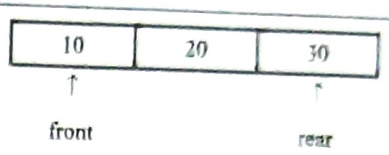


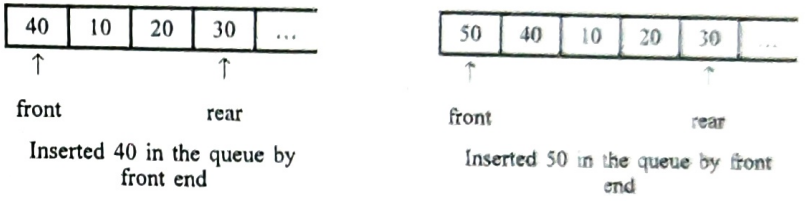
Fig. Q.10.1 Doubly ended queue

As we know, normally we insert the elements by rear end and delete the elements from front end. Let us say we have inserted the elements 10, 20, 30 by rear end.



Now if we wish to insert any element from front end then first we have to shift all the elements to the right.

For example if we want to insert 40 by front end then, the deque will be



(a) Insertion by front end

(b) Deletion by rear end

Fig. Q.10.2 Operations on deque

We can place –1 for the element which has to be deleted.

6.8 : Priority Queue

Q.11 Write a short note on - priority queue.

☞ **[SPPU : June-22, Marks 8]**

Ans. : The elements in the priority queue have specific ordering. There are two types of priority queues -

1. **Ascending Priority Queue** - It is a collection of items in which the items can be inserted arbitarily but only smallest element can be removed.

2. **Descending Priority Queue** - It is a collection of items in which insertion of items can be in any order but only largest element can be removed.

In priority queue, the elements are arranged in any order and out of which only the smallest or largest element allowed to delete each time.

The implementation of priority queue can be done using arrays or linked list. The data structure heap is used to implement the priority queue effectively.

Q.12 What are the applications of priority queue.

Ans. :

1. The typical example of priority queue is scheduling the jobs in operating system. Typically operating system allocates priority to jobs. The jobs are placed in the queue and position1 of the job in priority queue determines their priority. In operating system there are three kinds of jobs. These are real time jobs, foreground jobs and background jobs. The operating system always schedules the real time jobs first. If there is no real time job pending then it schedules foreground jobs. Lastly if no real time or foreground jobs are pending then operating system schedules the background jobs.
2. In network communication, to manage limited bandwidth for transmission the priority queue is used.
3. In simulation modeling, to manage the discrete events the priority queue is used.

END...

SOLVED MODEL QUESTION PAPER (In Sem)

Fundamentals of Data Structures

S.E. (Comp. / AI&DS) Semester - III [As Per 2019 Pattern]

Time : 1 Hour] **[Maximum Marks : 30**

N.B. : ***i) Attempt Q.1 or Q.2, Q.3 or Q.4.***

ii) Neat diagrams must be drawn wherever necessary.

iii) Figures to the right side indicate full marks.

iv) Assume suitable data, if necessary.

Q.1 a) *What is ADT ? Write ADT for an arrays.* **(Refer Q.3 of Chapter - 1)** **[5]**

b) *Explain asymptotic notations Big O, Theta and Omega with one example each.* **(Refer Q.18 of Chapter - 1)** **[10]**

OR

Q.2 a) *What is recurrence relation ? Explain with example.* **(Refer Q.22 of Chapter - 1)** **[3]**

b) *Determine the frequency counts for all the statements in the following program segment.*

```
i=10;
for(i=10;i<=n;i++)
  for(j=1;j<i;j++)
x=x+1;
```

(Refer Q.21 of Chapter - 1) **[4]**

c) *Explain the greedy strategy with suitable example. Comment on its time complexity.* **(Refer Q.34 of Chapter - 1)** **[8]**

Q.3 a) *Explain the concept of ordered list.* **(Refer Q.16 of Chapter - 2)** **[3]**

b) *Explain the types of arrays.* **(Refer Q.4 of Chapter - 2)** **[4]**

c) *How to merge two arrays ? Explain it with suitable example.* **(Refer Q.8 of Chapter - 2)** **[8]**