1. Explain in detail about sorting and different types of sorting techniques

Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, lexicographical, or any user-defined order. Sorting is a process through which the data is arranged in ascending or descending order. Sorting can be classified in two types;

Internal Sorts:- This method uses only the primary memory during sorting process. All data items are held in main memory and no secondary memory is required this sorting process. If all the data that is to be sorted can be accommodated at a time in memory is called internal sorting. There is a limitation for internal sorts; they can only process relatively small lists due to memory constraints. There are 3 types of internal sorts.

 (i) SELECTION SORT :-

Ex:-   Selection sort algorithm, Heap Sort algorithm

(ii) INSERTION SORT :-

Ex:-   Insertion sort algorithm, Shell Sort algorithm

(iii) EXCHANGE SORT :-   **Ex:- Bubble Sort Algorithm, Quick sort algorithm**

External Sorts:- Sorting large amount of data requires external or secondary memory. This process uses external memory such as HDD, to store the data which is not fit into the main memory. So, primary memory holds the currently being sorted data only. All external sorts are based on process of merging. Different parts of data are sorted separately and merged together.

Ex:- Merge Sort

2. Write a program to explain bubble sort. Which type of technique does it belong. (b) What is the worst case and best case time complexity of bubble sort?

```c
/* bubble sort implementation  */
#include<stdio.h>
#include<conio.h>
void main()
{
 int i,n,temp,j,arr[25];
 clrscr();
 printf("Enter the number of elements in the Array: ");
 scanf("%d",&n);
 printf("\nEnter the elements:\n\n");
 for(i=0 ; i<n ; i++)
 {
  printf(" Array[%d] = ",i);
  scanf("%d",&arr[i]);
 }

 for(i=0 ; i<n ; i++)
 {
  for(j=0 ; j<n-i-1 ; j++)
  {
   if(arr[j]>arr[j+1]) //Swapping Condition is Checked
   {
    temp=arr[j];
    arr[j]=arr[j+1];
    arr[j+1]=temp;
   }
  }

 }
```

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

Time Complexity of Bubble Sort :

The complexity of sorting algorithm is depends upon the number of comparisons that are made. Total comparisons in Bubble sort is: $n (n – 1) / 2 \approx n^2 – n$

Best case
: $O(n_2)$

Average case
: $O(n_2)$

Worst case
: $O(n_2)$

3. Explain the algorithm for bubble sort and give a suitable example.
        algorithm for exchange sort with a suitable example.

In bubble sort method the list is divided into two sub-lists sorted and unsorted. The smallest element is bubbled from unsorted sub-list. After moving the smallest element the imaginary wall moves one element ahead. The bubble sort was originally written to bubble up the highest element in the list. But there is no difference whether highest / lowest element is bubbled. This method is easy to understand but time consuming. In this type, two successive elements are compared and swapping is done. Thus, step-by-step entire array elements are checked. Given a list of 'n' elements the bubble sort requires up to n-1 passes to sort the data.
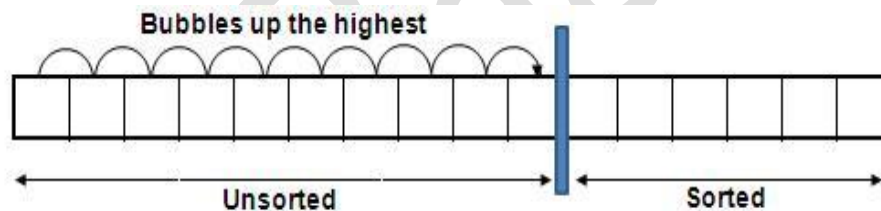
Algorithm for Bubble Sort: Bubble_Sort (A [ ] , N)

    Step 1 : Repeat For P = 1 to N – 1   Begin

    Step 2 :   Repeat For J = 1 to N – P   Begin

    Step 3 :   If (A [J] < A [J – 1])

                    Swap (A [J] , A [J – 1])   End For

                End For

    Step 4 : Exit

Example:

Ex:- A list of unsorted elements are: 10 47 12 54 19 23

(Bubble up for highest value shown here)

Original List	After Pass 1	After Pass 2	After Pass 3	After Pass 4	After Pass 5
10	54	54	54	54	54
47	10	47	47	47	47
12	47	10	23	23	23
54	12	23	10	19	19
19	23	12	19	10	12
23	19	19	12	12	10

A list of sorted elements now : 54 47 23 19 12 10

Show the bubble sort results for each pass for the following initial array of elements. 35 18 7 12 5 23 16 3 1

```
enter number of elements to be sorted:8

enter elements of array:35 18 7 12 5 23 16 31

The given list
    35      18      7      12      5      23      16      31

iteration    1      18      7      12      5      23      16      31      35
iteration    2      7      12      5      18      16      23      31      35
iteration    3      7      5      12      16      18      23      31      35
iteration    4      5      7      12      16      18      23      31      35
iteration    5      5      7      12      16      18      23      31      35
iteration    6      5      7      12      16      18      23      31      35
iteration    7      5      7      12      16      18      23      31      35
iteration    8      5      7      12      16      18      23      31      35

The final sorted list
    5      7      12      16      18      23      31      35
```

3.Write a program to explain insertion sort . Which type of technique does it belong.

(or)

Write a C-program for sorting integers in ascending order using insertion sort.

/*Program to sort elements of an array using insertion sort method*/

```c
#include<stdio.h>

#include<conio.h>

void main( )

{

  int a[10],i,j,k,n;

   clrscr( );

   printf("How many elements you want to sort?\n");

   scanf("%d",&n);

   printf("\nEnter the Elements into an array:\n");

  for (i=0;i<n;i++)

    scanf("%d",&a[i]);

  for(i=1;i<n;i++)

   {

   k=a[i];

    for(j= i-1; j>=0 && k<a[j]; j--)

     a[j+1]=a[j];

   a[j+1]=k;

   } printf("\n\n Elements after sorting: \n");
 for(i=0;i<n;i++)

   printf("%d\n", a[i]);
   getch( );

}
```

OUTPUT:

How many elements you want to sort ? : 6

Enter elements for an array :   78   23   45   8   32 36

After Sorting the elements are :   8   23   32   36   45 78

4.Explain the algorithm for insertion sort and give a suitable example.

Both the selection and bubble sorts exchange elements. But insertion sort does not exchange elements**.** In insertion sort the element is inserted at an appropriate place similar to card insertion. Here the list is divided into two parts sorted and unsorted sub-lists. In each pass, the first element of unsorted sub list is picked up and moved into the sorted sub list by inserting it in suitable position. Suppose we have 'n' elements, we need n-1 passes to sort the elements.

Insertion sort works this way:

It works the way you might sort a hand of playing cards:

1. We start with an empty left hand [sorted array] and the cards face down on the table [unsorted array].

2. Then remove one card [key] at a time from the table [unsorted array], and insert it into the correct position in the left hand [sorted array].

3. To find the correct position for the card, we compare it with each of the cards already in the hand, from right to left.
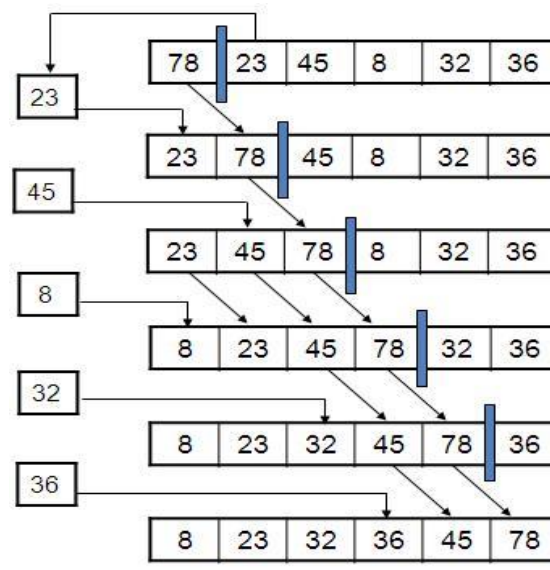
INSERTION_SORT (*A*)

1. **FOR** j ← 2 **TO** length[*A*]

2. **DO** key ← *A*[*j*]

3. {Put *A*[*j*] into the sorted sequence *A*[1 . . *j* − 1]}
4. *i* ← *j* − 1
5. **WHILE** *i* > 0 and *A*[*i*] > key
6. **DO** *A*[*i* +1] ← *A*[*i*]
7. *i* ← *i* − 1
8. *A*[*i* + 1] ← key

Example: Following figure (from CLRS) shows the operation of INSERTION-SORT on the array A= (5, 2, 4, 6, 1, 3). Each part shows what happens for a particular iteration with the value of *j* indicated. *j* indexes the "current card" being inserted into the hand.
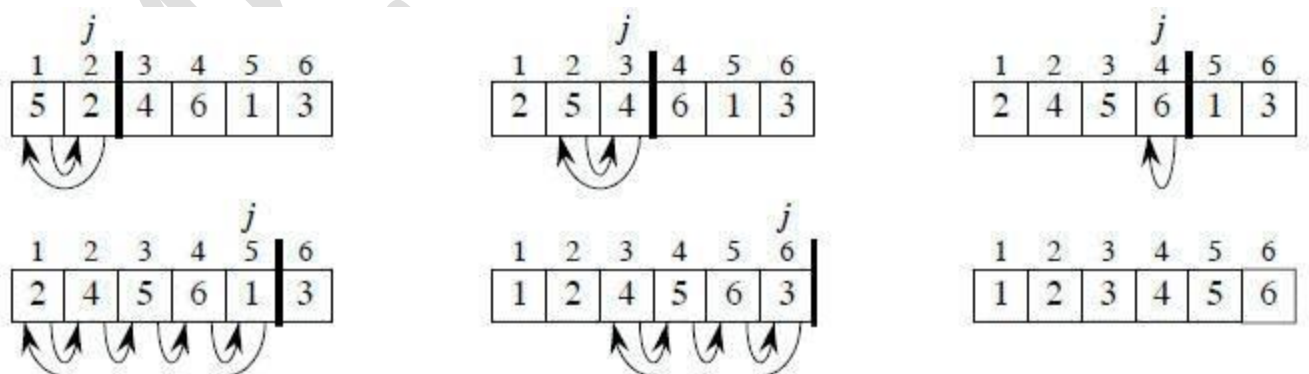
Read the figure row by row. Elements to the left of *A*[*j*] that are greater than *A*[*j*] move one position to the right, and *A*[*j*] moves into the evacuated position.

Ex:- A list of unsorted elements are: 78 23 45 8 32 36 . The results of insertion sort for

each pass is as follows:-

A list of sorted elements now : 8 23 32 36 45 78

5.Demonstrate the insertion sort results for each insertion for the following initial array
of elements

. 25 6 15 12 8 34 9 18 2

```
enter number of elements to be sorted:9

enter elements of array:25 6 15 12 8 34 9 18 2

The given list
     25    6    15    12     8    34     9    18     2

iteration  1 :     6    25    15    12     8    34     9    18     2
iteration  2 :     6    15    25    12     8    34     9    18     2
iteration  3 :     6    12    15    25     8    34     9    18     2
iteration  4 :     6     8    12    15    25    34     9    18     2
iteration  5 :     6     8    12    15    25    34     9    18     2
iteration  6 :     6     8     9    12    15    25    34    18     2
iteration  7 :     6     8     9    12    15    18    25    34     2
iteration  8 :     2     6     8     9    12    15    18    25    34


The final sorted list
      2     6     8     9    12    15    18    25    34
```

6.Write a program to explain selection sort. Which type of technique does it belong.

```c
/* program to sort elements of an array using selection
sort*/ #include<stdio.h>

void main( )
{

int i,j,t,n,min,a[10];

clrscr( );

printf("\n How many elements you want to sort?
"); scanf("%d",&n);

printf("\Enter elements for an array:");

for(i=0;i<n;i++)

scanf("%d",&a[i]);

for(i=0;i<n;i++)

{

min=i;

for(j=i+1;j<n;j++)

if(a[j] > a[min])

{

min=j;

}

t=a[i];

a[i]=a[min];

a[min]=t;

} printf("\nAfter sorting the elements
are:"); for(i=0;i<n;i++)

printf("%d ",a[i]);

getch( );

}
```

OUTPUT

Enter elements for an array :   **2 6   4 8   5**

After Sorting the elements are :   **8   6 5   4   2**

7. Explain the algorithm for selection sort and give a suitable example.

In selection sort the list is divided into two sub-lists sorted and unsorted. These two lists are divided by imaginary wall. We find a smallest element from unsorted sub-list and swap it to the beginning. And the wall moves one element ahead, as the sorted list is increases and unsorted list is decreases.

Assume that we have a list on n elements. By applying selection sort, the first element is compared with all remaining (n-1) elements. The smallest element is placed at the first location. Again, the second element is compared with remaining (n-1) elements. At the time of comparison, the smaller element is swapped with larger element. Similarly, entire array is checked for smallest element and then swapping is done accordingly. Here we need n-1 passes or iterations to completely rearrange the data.

Algorithm: Selection_Sort (A [ ] , N)

Step 1 :
      Repeat For K = 0 to N – 2   Begin

Step 2 :
      Set POS = K

Step 3 :
      Repeat for J = K + 1 to N – 1   Begin

        If A[J] < A [POS]

        Set POS = J

      End For

Step 5 :
      Swap A [K] with A [POS]

      End For

Step 6 :
      Exit

Ex:- A list of unsorted elements are: 23 78 45 8 32 56

A list of sorted elements now : 8 23 32 45 56   78

8.Show the quick sort results for each exchange for the following initial array of elements 35 54 12 18 23 15 45 38

```
***************QUICK SORT************
 enter number of elements to be sorted:8

 enter elements of array:35 54 12 18 23 15 45 38

The given list
    35     54     12     18     23     15     45     38

    23     15     12     18     35     54     45     38
    18     15     12     23     35     54     45     38
    12     15     18     23     35     54     45     38
    12     15     18     23     35     54     45     38
    12     15     18     23     35     38     45     54
    12     15     18     23     35     38     45     54

The final sorted list:
    12     15     18     23     35     38     45     54

_
```
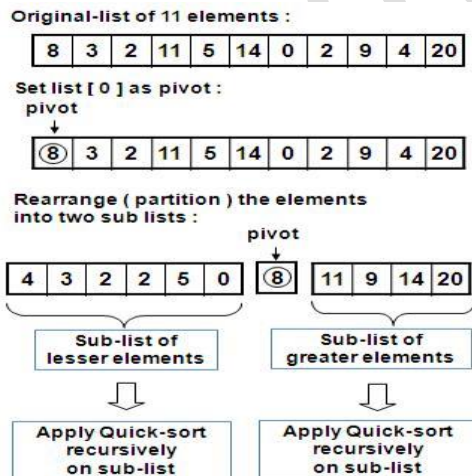
9. Explain the algorithm for QUICK sort (partition exchange sort) and give a suitable example.

Quick sort is based on partition. It is also known as partition exchange sorting. The basic concept of quick sort process is pick one element from an array and rearranges the remaining elements around it. This element divides the main list into two sub lists. This chosen element is called pivot. Once pivot is chosen, then it shifts all the elements less than pivot to left of value pivot and all the elements greater than pivot are shifted to the right side. This procedure of choosing pivot and partition the list is applied recursively until sub-lists consisting of only one element.

Ex:- A list of unsorted elements are: 8   3   2   11   5   14   0   2   9   4   20

Algorithm for quick sort:

It is also known as partition exchange sort. It was invented by CAR Hoare. It is based on partition. The basic concept of quick sort process is pick one element from an array and rearranges the remaining elements around it. This element divides the main list into two sub lists. This chosen element is called pivot. Once pivot is chosen, then it shifts all the elements less than pivot to left of value pivot and all the elements greater than pivot are shifted to the right side. This procedure of choosing pivot and partition the list is applied recursively until sub-lists consisting of only one element.

quicksort(q)

 varlist less, pivotList, greater

 if length(q) ≤ 1

 return q

 select a pivot value pivot from q

 for each x in q except the pivot element

 if x < pivot then add x to less

 if x ≥ pivot then add x to greater

 add pivot to pivotList

 return concatenate(quicksort(less), pivotList, quicksort(greater))

Time Complexity of Quick sort:

Best case : O (n log n)

Average case : O (n log n)

Worst case : O (n^2)

Advantages of quick sort:

1. This is faster sorting method among all.

2. Its efficiency is also relatively good.

3. It requires relatively small amount of memory.

Disadvantages of quick sort:

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

10. Explain the algorithm for Merge sort and give a suitable example.

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

Algorithm for merge sort:

Merge sort is based on the **divide-and-conquer** paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with sub-problems, we state each sub-problem as sorting a sub-array $A[p .. r]$. Initially, $p = 1$ and $r = n$, but these values change as we recurse through sub-problems.

To sort $A[p .. r]$:

1. **Divide Step**

   If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split $A[p .. r]$ into two sub-arrays $A[p .. q]$ and $A[q + 1 .. r]$, each containing about half of the elements of $A[p .. r]$. That is, q is the halfway point of $A[p .. r]$.

2. **Conquer Step**

   Conquer by recursively sorting the two sub-arrays $A[p .. q]$ and $A[q + 1 .. r]$.

3. **Combine Step**

   Combine the elements back in $A[p .. r]$ by merging the two sorted sub-arrays $A[p .. q]$

   and $A[q + 1 .. r]$ into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).
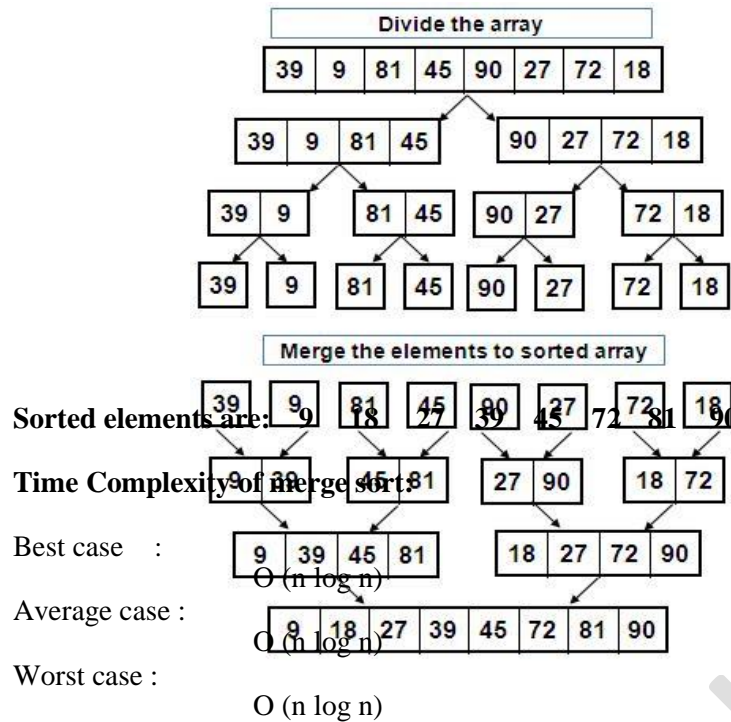
Note that the recursion bottoms out when the sub-array has just one element, so that it is trivially sorted.

To sort the entire sequence A[1 .. n], make the initial call to the procedure MERGE-SORT $(A, 1, n)$.

MERGE-SORT (A, p, r)

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

Ex:- A list of unsorted elements are: 39   9   81   45   90   27   72   18

Divide the array

| 39 | 9 | 81 | 45 | 90 | 27 | 72 | 18 |

| 39 | 9 | 81 | 45 | | 90 | 27 | 72 | 18 |

| 39 | 9 | | 81 | 45 | | 90 | 27 | | 72 | 18 |

| 39 | | 9 | | 81 | | 45 | | 90 | | 27 | | 72 | | 18 |

Merge the elements to sorted array

Sorted elements are: 9   18   27   39   45   72   81   90

Time Complexity of merge sort:

Best case   :
O (n log n)

Average case :
O (n log n)

Worst case :
O (n log n)

---

11. Write a program to implement Quick sort.

/* program to sort elements of an array using Quick Sort */

```c
#include<stdio.h>

void quicksort(int[ ],int,int);
void main( )
{
 int low, high, pivot, t, n, i, j, a[10];
 clrscr( );
 printf("\nHow many elements you want to sort ?");
 scanf("%d",&n);
 printf("\Enter elements for an array:");
 for(i=0; i<n; i++)
  scanf("%d",&a[i]);
 low=0;
 high=n-1;
 quicksort(a,low,high);
 printf("\After Sorting the elements are:");
 for(i=0;i<n;i++)
  printf("%d ",a[i]);
 getch( );
 }

 void quicksort(int a[ ],int low,int high)
 {
 int pivot,t,i,j;
 if(low<high)
 {
 pivot=a[low];
 i=low+1;
 j=high;
 while(1)
```

```
    while(pivot>a[i]&&i<=high)
       i++;
     while(pivot<a[j]&&j>=low)
      j--;
     if(i<j)
     {
      t=a[i];
      a[i]=a[j];
      a[j]=t;
     }
     else
     break;
    }
   a[low]=a[j];
   a[j]=pivot;
   quicksort(a,low,j-1);
   quicksort(a,j+1,high);
  }
}
```

OUTPUT:
How many elements you want to sort ?
: 6
Enter elements for an array :   78   23   45   8   32 36
After Sorting the elements
are :   8   23   32   36   45 78

12. Write a program to implement Merge sort.

/* program to sort elements of an array using Merge Sort */

#include<stdio.h>

void disp( );

void mergesort(int,int,int);

void msortdiv(int,int);

int a[50],n;

void main( )

{

 int i;

 clrscr( );

 printf("\nEnter the n value:");
 scanf("%d",&n);
 printf("\nEnter elements for an array:");

 for(i=0;i<n;i++)

  scanf("%d",&a[i]);

```
printf("\nBefore Sorting the elements are:");

disp( );

msortdiv(0,n-1);

printf("\nAfter Sorting the elements are:");

disp( );

getch( );

}

void disp( )

{

int i;

for(i=0;i<n;i++)

printf("%d ",a[i]);

}

void mergesort(int low,int mid,int high)
{
int t[50],i,j,k;
i=low;
j=mid+1;
k=low;

while((i<=mid) && (j<=high))

{

if(a[i]>=a[j])

t[k++]=a[j++];

else

t[k++]=a[i++];

}

while(i<=mid)

t[k++]=a[i++];

while(j<=high)

t[k++]=a[j++];

for(i=low;i<=high;i++)

a[i]=t[i];

}
```

```
 void msortdiv(int low,int high)
 {

   int mid;

   if(low!=high)

   {
    mid=((low+high)/2);
    msortdiv(low,mid);
    msortdiv(mid+1,high);
    mergesort(low,mid,high);

   }

 }
```

OUTPUT:

How many elements you want to sort ? : 7

Enter elements for an array :   88 45   54   8   32   6 12

After Sorting the elements are :   6   8   12   32   45   54 88

13. Explain a sorting technique which follows divide and conquer mechanism with an example. (quick & merge sorts)

Divide and Conquer:- This is a special case of recursion in which given problem is divided into two or more sub-problems of exactly same type and solution to problem is expressed in terms of solution to sub-problem. i.e. Dividing the list of elements into two approximately equal parts recursively and find solution independently then merged together into single list. Quick and merge sorts are based on Divide and Conquer concept.

The divide and conquer strategy solves a problem by :

 1. Breaking into sub problems that are themselves smaller instances of the same type of problem.

 2. Recursively solving these sub problems.

 3. Appropriately combining their answers.

 Two types of sorting algorithms which are based on this divide and conquer algorithm :

1. Quick sort: Quick sort also uses few comparisons (somewhat more than the other two). Like heap sort it can sort "in place" by moving data in an array.

2. Merge sort: Merge sort is good for data that's too big to have in memory at once, because its pattern of storage access is very regular. It also uses even fewer comparisons than heap sort, and is especially suited for data stored as linked lists.

14. Write and explain linear search procedure or algorithm with a suitable example.

Linear search technique is also known as sequential search technique. The linear search is a method of searching an element in a list in sequence. In this method, the array is searched for the required element from the beginning of the list/array or from the last element to first element of array and continues until the item is found or the entire list/array has been searched.

Algorithm:

Step 1: set-up a flag to indicate "element not found"

Step 2: Take the first element in the list

Step 3: If the element in the list is equal to the desired element

> Set flag to "element found"

> Display the message "element found in the list" >
Go to step 6

Step 4: If it is not the end of list,

> Take the next element in the list
> Go to step 3

Step 5: If the flag is "element not found"

Display the message "element not found" Step 6: End of the Algorithm

Advantages:

1. It is simple and conventional method of searching data. The linear or sequential name implies that the items are stored in a systematic manner.

2. The elements in the list can be in any order. i.e. The linear search can be applied on sorted or unsorted linear data structure.

Disadvantage:

1. This method is insufficient when large number of elements is present in list.

2. It consumes more time and reduces the retrieval rate of the system.

Time complexity: O(n)

15. Formulate recursive algorithm for binary search with its timing analysis.

Binary search is quicker than the linear search. However, it cannot be applied on unsorted data structure. The binary search is based on the approach **divide-and-conquer**. The binary search starts by testing the data in the middle element of the array. This determines target is whether in the first half or second half. If target is in first half, we do not need to check the second half and if it is in second half no need to check in first half. Similarly we repeat this process until we find target in the list or not found from the list. Here we need 3 variables to identify first, last and middle elements.

To implement binary search method, the elements must be in sorted order. Search is performed as follows:
- The key is compared with item in the middle position of an array
- If the key matches with item, return it and stop
- If the key is less than mid positioned item, then the item to be found must be in first half of array, otherwise it must be in second half of array.
- Repeat the procedure for lower (or upper half) of array until the element is found.

Recursive Algorithm:
Binary_Search(a,key,lb,ub)
begin

Step 1:   [initialization]
              lb=0
              ub=n-1;
Step 2:
                  [search for the item]
Step 3:
              Repeat through step 4 while lower bound(lb) is less than upper bound.
Step 4:
                  [obtain the index of middle value]
              mid = (lb+ub)/2
                  [compare to search for item]
              if(key < a[mid]) then
              ub=mid-1
              otherwise if(key > a[mid]) then
              lb=mid+1;
              otherwise if(key==a[mid]) Write "match found"
              return (mid)
          return Binary_Search(a,key,lb,ub)

Step 5: [unsuccessful search]

Step 6: Write "match not found"
        [end of algorithm]

16. Write a C program that searches a value in a stored array using linear search.

```c
#include<stdio.h>
int linear(int [ ],int,int);
void main( )
{
int a[20], pos = -1, n, k, i;
clrscr( );
printf("\nEnter the n value:");
scanf("%d",&n);
printf("\nEnter elements for an array:");
for(i=0; i<n ;i++)
 scanf("%d",&a[i]);
printf("\nEnter the element to be searched:");
scanf("%d",&k);

pos=linear(a,n,k);

if(pos != -1)

  printf("\n Search successful element found at position %d",pos);

else

  printf("\n Search unsuccessful, element not found");

getch( );

}

int linear(int a[ ],int n,int k)

{

  int i;

  for(i=0;i<n;i++)

  {

  if(a[i]==k)

   return(i);

  }

  return -1;

}
```

Output:-

Enter the n value   :   5

Enter elements for an array   :   11   2   23   14   55

Enter the element to be searched: 14

Search successful element found at position   :   3

17.Write a program for recursive binary search to find the given element within array. For What data binary search is not applicable?

```c
/* recursive binary search*/

 #include<stdio.h>
int bsearch(int [ ],int, int, int);
void main( )
{
 int a[20],pos,n,k,i,lb,ub;
 clrscr( );
 printf("\nEnter the n value:");
 scanf("%d",&n);
 printf("\nEnter elements for an array:");
 for(i=0;i<n;i++)
  scanf("%d",&a[i]);
 printf("\nEnter the key value:");
 scanf("%d",&k);
 lb=0;
 ub=n-1;
 pos=bsearch(a,k,lb,ub);
 if(pos!=-1)
  printf("Search successful, element found at position
 %d",pos); else
  printf("Search unsuccessful, element not found");
 getch( );

 }

 int bsearch(int a[ ], int k, int lb, int ub)
 {
 int mid;
 while(ub>=lb)
 {
 mid=(lb+ub)/2;
 if(k<a[mid])
   ub=mid-1;
  else if(k>a[mid])

   lb=mid+1;

else if(k==a[mid])

   return(mid);

   return(bsearch(a,k,lb,ub));

 }

 return -1;

 }
```

OUTPUT:

Enter 'n' value   :   6

Enter elements for an array   :   10   32   25   84   55   78

Enter the element to be searched   :   78

Search successful, Element found at Position   :   5

18. Write a C program that searches a value in a stored array using recursive linear search.

/*   recursive program for Linear Search*/

```c
#include<stdio.h>
int linear(int [ ],int,int);
void main( )
{
 int a[20],pos=-1,n,k,i;
 clrscr();
 printf("\nEnter  n value:");
 scanf("%d",&n);
 printf("\nEnter elements for an array:");
 for(i=0;i<n;i++)
   scanf("%d",&a[i]);
 printf("\n Enter the element to be searched:");
 scanf("%d",&k)

  pos=linear(a,n,k);

if(pos!=-1)

   printf("\n Search successful, Element found at Position %d",pos);

 else

  printf("Search unsuccessful, element not found ");

 getch( );

 }

 int linear(int a[ ],int n,int k)

 {

 int i;

 for(i=n-1;i>=0;i--)

 {

 if(a[i]==k)

  return(i);

 else

  {

   n=n-1;

   return(linear(a,n,k));

  }

 }

 return -1;

}
```

Output:-

Enter 'n' value :   6

Enter elements for an array   :   10   32   22   84   55   78

Enter the element to be searched   :   55

Search successful, Element found at Position   :   4

19 . Write a C program that searches a value in a stored array using non recursive binary search.

```c
#include<stdio.h>

int bsearch(int [ ],int,int);

void main( )

{

int a[20],pos,n,k,i;

clrscr();

printf("\nEnter the n value:");

scanf("%d",&n);

printf("\nEnter elements for an array:");

for(i=0;i<n;i++)

scanf("%d",&a[i]);

printf("\nEnter the key value:");

scanf("%d",&k);

pos=bsearch(a,n,k);

if(pos!= -1)

printf("Search successful, element found at position %d",pos); else

printf("Search unsuccessful, element not found");

getch( );

}

int bsearch(int a[ ],int n, int k)

{

int lb=0,ub,mid;

lb=0;
```

```
ub=n-1;

while(ub>=lb)

{

mid=(lb+ub)/2;

if(k<a[mid])

   ub=mid-1;

else if(k>a[mid])

   lb=mid+1;

else if(k==a[mid])

   return(mid);

}

return -1;

}
```

OUTPUT

Enter 'n' value :   67

Enter elements for an array   :   35   10   32   25   84   55   78

Enter the element to be searched   :   25

Search successful, Element found at Position   :   3

Data structures

1. **Define a data structure. What are the different types of data structures? Explain each of them with suitable example.**

We can define that Data structure is a kind of representation of logical relationship between related data elements. In data structure, decision on the operations such as storage, retrieval and access must be carried out between the logically related data elements.

Data structure can be nested, i.e. we can have a Data structure that consists of other Data structure. Data structure is a most convenient way to handle data of different types including ADT for a known problem.

Data structures are classified in several ways:

Linear : Elements are arranged in sequential fashion. Ex : Array, Linear list, stack, queue

Non-Linear : Elements are not arranged in sequence. Ex : trees, graphs

Homogenous : All Elements are belongs to same data type. Ex : Arrays

Non-Homogenous : Different types of Elements are grouped and form a data structure. Ex: classes

Dynamic : Memory allocation of each element in the data structure is done before their usage using

D.M.A functions Ex : Linked Lists

Static : All elements of a data structure are created at the beginning of the program. They cannot be

resized. Ex : Arrays.

Generally Data structures are classified into two types as follows:
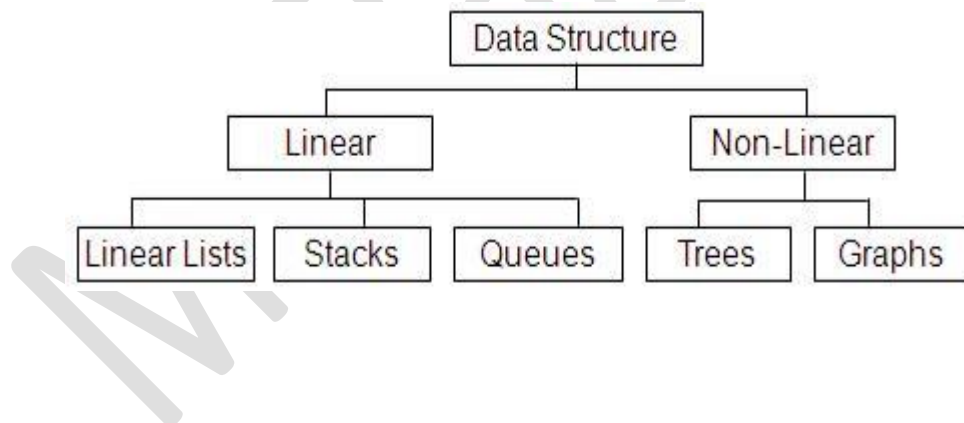
2. Define tree. What is a subtree? Define the following terms. children nodes, siblings, root node, leaves level and degree of tree .

Tree:

A tree is a set of nodes which is either null or with one node designated as the root and the remaining nodes partitioned into smaller trees, called sub-trees.
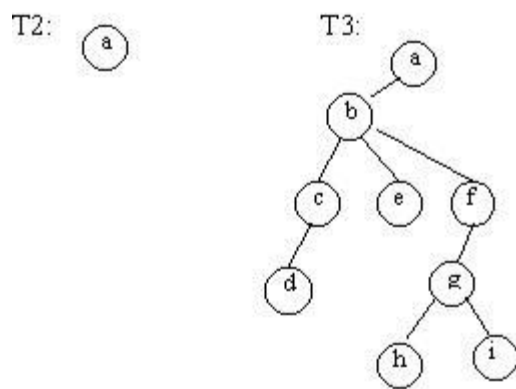
Example:

T1={} (NULL Tree)

T2={a} a is the root, the rest is T1

T3={a, {b,{c,{d}},{e},{f,{g,{h},{i}}}}}

graphical representation:

- The level of a node is the length of the path from the root to that node

- The depth of a tree is the maximum level of any node of any node in the tree
- The degree of a node is the number of partitions in the subtree which has that node as the root
- Nodes with degree=0 are called leaves

Subtrees:

A subtree is a portion of a tree data structure that can be viewed as a complete tree in itself. Any node in a tree T, together with all the nodes below it, comprise a subtree of T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree (in analogy to the term proper subset).

Leaf nodes:

Nodes at the bottommost level of the tree are called leaf nodes. Since they are at the bottommost level, they do not have any children.

Internal nodes

An internal node or inner node is any node of a tree that has child nodes and is thus not a leaf node.

3. Write C-structure for implementing Stack using an array. Using this structure, write functions for push and pop operations.

STACK

Stack is an ordered collection of data elements into which new elements may be inserted and from which elements may be deleted at one end called the "TOP" of stack.

    -- A stack is a last-in-first-out (LIFO) structure.

    -- Insertion operation is referred as "PUSH" and deletion operation is referred as "POP".

    -- The most accessible element in the stack is the element at the position "TOP".

    -- Stack must be created as empty.

    -- Whenever an element is pushed into stack, it must be checked whether the stack is full or not.

    -- Whenever an element is popped form stack, it must be checked whether the stack is empty or

not.

    -- We can implement the stack ADT either with array or linked list.


Applications of stack:

1) Stacks are used in conversion of expression from infix notation to postfix and prefix notation.

2) Stacks are used for evaluation of infix and postfix forms.

3) Stacks are used in tree traversal techniques.

4) Recursive functions are implemented using stacks. The copies of variables at each level of recursion are stored in stack.

5) Compilers use stacks in syntax analysis phase to check whether a particular statement in a program is syntactically correct or not.

6) Computers use stack during interrupts and function calls. The information regarding actual parameters return values, return addresses and machine status is stored in stack.

7) Stacks are used in depth first search of a graph.

4. What are the operations on Linear Lists? Differentiate between using Arrays and Linked Lists for implementation of Linear Lists.

In computer science, a list or sequence is an abstract data type that implements an ordered collection of values, where the same value may occur more than once. An instance of a list is a computer representation of the mathematical concept of a finite sequence. Each instance of a value in the list is usually called an item, entry, or element of the list; if the same value occurs multiple times, each occurrence is considered a distinct item.
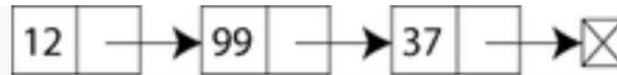
A singly linked list structure, implementing a list with 3 integer elements.

The name list is also used for several concrete data structures that can be used to implement abstract lists, especially linked lists.

The so-called static list structures allow only inspection and enumeration of the values.

A mutable or dynamic list may allow items to be inserted, replaced, or deleted during the list's existence.

Operations on a list:

Implementation of the list data structure may provide some of the following operations:

- a constructor for creating an empty list;

- an operation for testing whether or not a list is empty;

- an operation for prepending an entity to a list

- an operation for appending an entity to a list

- an operation for determining the first component (or the "head") of a list

- an operation for referring to the list consisting of all the components of a list except for its first (this is called the "tail" of the list.)

- deleting an element from a list

- searching an element from a list

- displaying an entire list

There are two types of implementation for Data structures.

(i) Array Implementation:- This uses static structures that are determined during the compilation process. Memory is allocated at compile time.

Advantages of Arrays:

- Searching an array for an individual element is efficient.

- Less time taken than Lists for searching an element.

Limitations of Arrays:

- Fixed in size : Once an array is created, the size of array cannot be increased or decreased.

- Wastage of space : If number of elements are less, leads to wastage of space.

- Sequential Storage : Array elements are stored in contiguous memory locations. At the times it might so happen that enough contiguous locations might not be available. Even though the total space requirement of an array can be met through a combination of non-contiguous blocks of memory, we would still not be allowed to create the array.

- Possibility of overflow : If program ever needs to process more than the size of array, there is a possibility of overflow and code breaks.

- Difficulty in insertion and deletion : In case of insertion of a new element, each element after the specified location has to be shifted one position to the right. In case of deletion of an element, each element after the specified location has to be shifted one position to the left.

(i) Pointer or Linked List Implementation:- This uses dynamic structures that are determined during the execution of program. Memory is allocated dynamically during execution.

Advantages of Lists:

- size : Once List is created, the size of List may be increased or decreased dynamically.

- Storage : List elements are stored in anywhere in memory locations. It is need not to be contiguous only. i.e. To allocate additional memory space and release unwanted space at the time of execution is possible.

- Easy in insertion and deletion : Dynamic insertion of an element into list in any position and deletion from list is possible.

- It is dynamic data structure with the ability to grow and shrink as per the program requirement.

Limitations of Lists:

- Additional memory required for link fields.

- It is time consuming for searching an element than arrays.

5. Write Structure for implementing Linked List of integers. Write C-function for insertion operation in Linked List.

/*program for implementation of insertion on a singly linked list*/

Singly Linked List:-

In Singly Linked list, two successive nodes of the linked list are linked with each other in sequential linear manner. Each node has at least two members, one of which points to the next Node in the list and the other holds the data. These are defined as Single Linked Lists because they can only point to the next Node in the list but not to the previous.

Singly Linked List contains a pointer variable called as head which points the first node of the list.

The address part of the last node is NULL which indicates the end of list.

Insertion of elements in to the Singly Linked List:-

There are 3 categories for insertion of elements into the list.

(i) Insertion at Beginning:-

Consider a list of nodes, 10, 20, 30 as data, and address of each node 1000, 2000, 3000 respectively.

We want to add new node at beginning for this list.

Step1: Create a new node. Assume its address is 500 and data to be inserted is 5.

Step2: Assign the data to the data part of new node.

Step3: Now, assign the address part of head to the next part of new node.

Step4: Assign the address of new node to the address of head

```
    if (head = = NULL)              //  when list is empty

      {

       head = new;

     new-> next = NULL:

      }

      else                       // if list already contains elements


      {

            new-> next = head;

            head = new;

      }
```

(ii) Insertion at given position:-

Consider a list of nodes, 10, 20, 30 as data, and address of each node 1000, 2000, 3000 respectively.

We want to add new node at specified location in this list.

Step1: Create a new node. Assume its address is 2500 and data to be inserted is 25.

Step2: Assign the data to the data part of new node. Specify the position where the element to be inserted. Assume it is 3.

Step3: Now, traverse the list up to position 2 node.

Step4: Assign the next part of node 2 to next part of new node (it it node3).

Step5: Assign the address of new node to next part of node 2.

```
  if (pos = = 1)                    //  this is first node
{
 new->next=head;
 head = new;
}
 else                         // if list already contains elements
 {
 i=1;  temp=head;
   while ( i != pos -1)
     {
        temp=temp->next;
        i++;
     }
  new-> next = temp->next;
  temp->next = new;
```

(iii) Insertion at end:-

Consider a list of nodes, 10, 20, 30 as data, and address of each node 1000, 2000, 3000 respectively.

We want to add new node at end of list.

Step1: Create a new node. Assume its address is 4500 and data to be inserted is 40.

Step2: Assign the data to the data part of new node. Now, traverse the list till the last node

    reached.

Step3: The next part of last node is assigned with address of new node.

Step5: Assign the next part of new node with NULL.

```
  if (head = = NULL)             //  when list is empty
 {
  head = new;
 new-> next = NULL;
 prev=new;
 }
 else                       // if list already contains elements
 {
    while(prev->next != NULL)
        prev=prev->next;
```

```
prev->next=new;

new-> next = NULL;

 prev = new;

}
```

6. Define tree and binary tree. Explain in detail.

Tree as a data structure

• A tree is a data structure that is made of nodes and pointers, much like a linked list. The difference

between them lies in how they are organized: • The top node in the tree is called the root and all

other nodes branch off from this one. • Every node in the tree can have some number of children. Each child node can in turn be the parent node to its children and so on.

• Child nodes can have links only from a single parent. • Any node higher up than the parent is called an ancestor node. • Nodes having no children called leaves.• Any node which is neither a root, nor a leaf is called an interior node. • The height of a tree is defined to be the length of the longest path from the root to a leaf in that tree (including the path to root) • A common example of a tree structure is the binary tree.

Binary Trees Definition: A binary tree is a tree in which each node can have maximum two children. Thus each node can have no child, one child or two children. The pointers help us to identify whether it is a left child or a right child. Application of a Binary tree
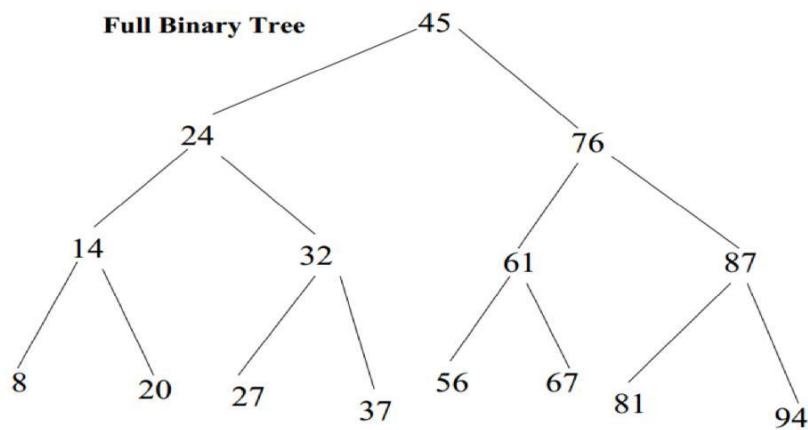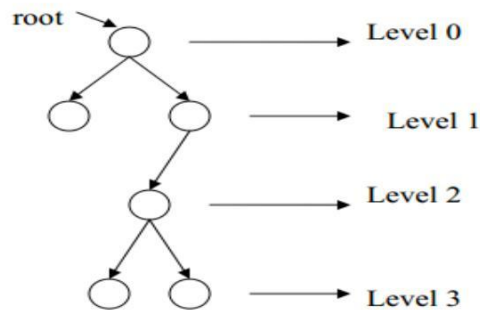
Examples of binary trees:
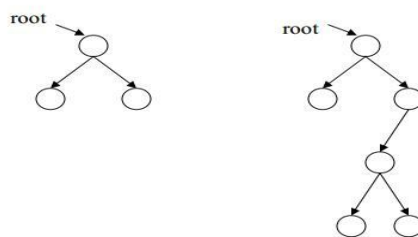
• The following are NOT binary trees:
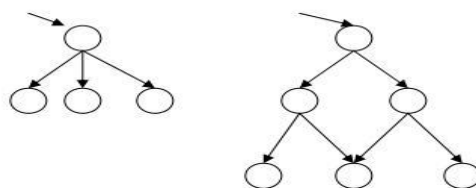
• tree, then n1 is the parent of n2 and n2 is the left or right child of n1.

• The level of a node in a binary tree:

- The root of the tree has level 0

- The level of any other node in the tree is one more than the level of its parent

Full Binary Tree

How many nodes?

Level 0 : 1 node (height 1)

Level 1: 2 nodes (height 2)

Level 3 : 4 nodes (height 3)

Level 3: 8 nodes (height 4)

Total number of nodes

$n = 2h - 1$ (maximum)

$h = \log (n+1)$

7. Define graph and its terms. Explain in detail.

A graph data structure consists of a finite (and possibly mutable) set of ordered pairs,

called edges or arcs, of certain entities called nodes or vertices. As in mathematics, an edge (x, y) is said to point or go from x to y. The nodes may be part of the graph structure, or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some edge value, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

Operations:

The basic operations provided by a graph data structure G usually include:

- adjacent(G, x, y): tests whether there is an edge from node x to node y.

- neighbors(G, x): lists all nodes y such that there is an edge from x to y.

- add(G, x, y): adds to G the edge from x to y, if it is not there.

- delete(G, x, y): removes the edge from x to y, if it is there.

- get_node_value(G, x): returns the value associated with the node x.

- set_node_value(G, x, a): sets the value associated with the node x to a.
Structures that associate values to the edges usually also provide:

- get_edge_value(G, x, y): returns the value associated to the edge (x,y).

- set_edge_value(G, x, y, v): sets the value associated to the edge (x,y) to v.

8. Implement stack using arrays.

// WAP for implementation of stack operations using arrays

```c
#include<stdio.h>
#define MAX 5
int stack[MAX];
int top=-1;
void push();
void pop();
void display();
void main()
{
 int ch;
 clrscr();
 while(1)
 {
 printf("\n **** MENU **** \n\n 1.PUSH \n 2.POP    \n 3.DISPLAY \n 4.EXIT \n");
 printf("\nenter your choice : \n");
 scanf("%d",&ch);
 switch(ch)
 {
 case 1:push();
        break;
 case 2:pop();
        break;
```

```
  case 3:display();
          break;
  case 4:return;

  }
 }
}

void push()
{
 int item;
 printf("\nenter value for stack :");
 scanf("%d",&item);
 if(top==MAX-1)
 printf("\nstack is overflow \n");
 else
 {
  top++;
  stack[top]=item;
 }
}

void pop()
{
 int item;
 if(top==-1)
 printf("\nStack is underflow  \n");
 else
 {
  item=stack[top];
  top--;

 }
 printf("\ndeleted item is : %d",item);
}

void display()
{
 int i;
 printf("\nThe elements in the stack are :\n");
 for(i=top;i>=0;i--)
  printf("%d\n",stack[i]);
}
```

9. Explain operations of Queue .

- Queue is a linear data structure that permits insertion of new element at one end and deletion of an element at the other end.

-- The end at which insertion of a new element can take place is called ' rear ' and the end at which deletion of an element take place is called ' front '.

-- The first element that gets added into queue is the first one to get removed from the list, Hence Queue is also referred to as First-In-First-Out (FIFO) list.

-- Queue must be created as empty.

-- Whenever an element is inserted into queue, it must be checked whether the queue is full or not.

-- Whenever an element is deleted form queue, it must be checked whether the queue is empty or not.

-- We can implement the queue ADT either with array or linked list.
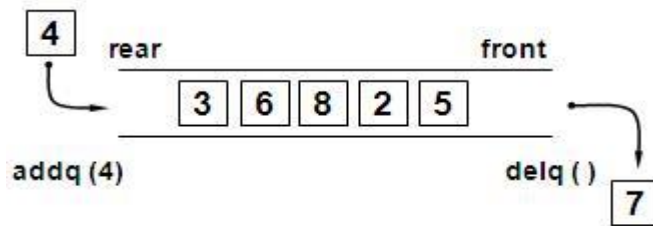
> **Applications of Queues:**
>
> **Execution of Threads**

> **Job Scheduling**
> **Event queuing**
> **Message Queuing**

10 Implement queue using arrays

/* Implentation of queue using arrays */

```c
# include <stdio.h>
# define SIZE 10

int arr[ SIZE ], front = -1, rear = -1, i ;
void enqueue() ;
void dequeue() ;
void display() ;
int main()
{
   int ch ;
   do
     {
       printf( "\n[1].ENQUEUE [2].DEQUEUE [3].Display [4].Exit\n" )
       ; printf( "Enter your choice [1-4] : " ) ; scanf( "%d", &ch ) ;

       switch ( ch )
         {
         case 1 :

            enqueue() ;
            break ;

         case 2 :
            dequeue() ;
            break ;

         case 3 :

            display() ;
            break ;

         case 4 :
            break ;
```

```
            default :
               printf( "Invalid option\n" ) ;
            }
        }

    while ( ch != 4 ) ;
}

void enqueue()
{

    if ( rear == SIZE – 1 )
        {
            printf( "Queue is full (overflow)\n" ) ;

            return ;
        }

    rear++ ;
    printf( "Enter the element to ENQUEUE : " ) ;
    scanf( "%d", &arr[ rear ] ) ;

    if ( front == -1 )
        front++ ;
}

void dequeue()

{
    if ( front == -1 )
        {

            printf( "Queue is empty (underflow)\n" );
            return ;
        }

    printf( "The DEQUEUE element is : %d\n", arr[ front ] ) ;

    if ( front == rear )

        front = rear = -1 ;
    else
        front++ ;
}

void display()
{
    if ( front == -1 )
        {
            printf( "Queue is empty (underflow)\n" ) ;

            return ;
        }

    printf( "The elements in queue are : FRONT -> " ) ;

    for ( i = front ; i <= rear ; i++ )
        printf( " … %d", arr[ i ] ) ;

    printf( " … <- REAR\n" ) ;
}
```

Linear list and the searching operation on linear list.

Linear linked list:-

Linear linked list is a linear collection of data elements, called nodes, where the linear order is

given by means of pointers.

Each node is divided into two parts:

• The first part contains the information of the element and

• The second part contains the address of the next node (link /next pointer field) in the list.

11. What is meant by linear list? Explain the searching operation on linear list.
Searching through the linked list:-

Both the insert and delete for lists require that we search the list. For insert we need to know the predecessor to the node to be inserted, for delete we need to know the predecessor to the node to be deleted. Also to locate a node for processing, such as adding to a count or printing its contents we need to know its location. This means that our search must return both the predecessor and the current location.

To search a list on a key, we need a key field. For simple lists, the key and the data can be the same fields. More complex structures require a separate key field.

Given a target key, the search attempts to locate the requested node in the linear list. If the node in the list matches the target value, the search returns true if no key matches, it returns false.

The predecessor and current pointers are set according to the rules:

Condition	ppre	pcur	Return
Target<first node	Null	First node	0
Target==first node	Null	First node	1
First<target<last	Largest node<target	First node>target	0
Target==middle node	Nodes predecessor	Equal node	1
Target==last node	Last predecessor	Last node	1
Target>last node	Last node	NULL	0

1. Use the operations push, pop, stacktop, and empty to construct operations on stack, which do each of the following:
Given an integer n, set i to the n the element from the top of stack, leavin

stack unchanged Set I to the bottom element of stack, leaving the stack empty.
2. Show how to implement a queue of integers in C by using an array int q[QUEUESIZE], where q[0] is used to indicate the front of the queue , q[1] is used to indicate its rear and where q[2] through q[QUEUESIZE -1] contain elements on the queue. Show

how to initialize such an array to represent the empty queue and write routines remove, insert and empty for such an implementation.

Algorithm for push and pop operations of a stack:

- algorithm of push
  if(top==Max-1)
  {

  print the stack is full";

  }

  else top++;
  arr[top]=item;
  "

algorithm of pop

if(top==-1)

{

print " the stack is empty";

}

else

return arr[top];

top--;

}

12. Define a stack and write an algorithm for the operation son stack?

Stacks are the subclass of lists that permits the insertion and deletion operation to be performed at only one end. They are LIFO(last in first out)list. An example of a stack is a railway system for shunting cars in this system ,the last railway car to be placed on the stack is the first to leave.

Operation on stacks

A pointer TOP keeps track of the top element in the stack. Initially when the stack is empty ,TOP has a value of zero and when the stack contains a single element, TOP has a value of one and so on. Each time a new element is inserted in the stack, the pointer is incremented by one before the element is placed in the stack. The pointer is decremented by one each time a deletion is made from the stack.

Push

This is the insertion operation. When the value is entered it is inserted into an array.

Algorithm

Push(S,TOP,X).This procedure inserts an element X to the top of a stack which is represented by a vector S containing N elements with a pointer TOP denoting the top element of the stack.

1. [check for stack
   overflow] if TOP >=N

      then Write("stack overflow")

            Return

2.   [increment Top]

TOPßTOP+1

3.   [insert element]
     S[TOP]ßX
4.   [finished]

Return

The first step of this algorithm checks for overflow condition. If such a condition exits then the insertion operation cannot be performed .

Pop

This operation is used to remove a data item form the stack.

Algorithm

Pop(S,TOP).This function removes the top element from the stack which is represented by the vector S and returns this element. TOP is a pointer to the top of the stack.
1.   [check for underflow of the stack]

if TOP=0

then Write("stack underflow on pop)

Exit

2.   [decrement pointer]
     TOPßTOP-1
3.   [return former top element of the stack]

Return(S[TOP+1])

An underflow condition is checked in the first step .If there is an underflow then appropriate actions should take place.

Peep

This operation is to read the value on the top of the stack without removing it.

Algorithm

Peep(S,TOP,I).Given a vector S of N elements representing a sequentially allocated stack, and a pointer TOP denoting the top element of the stack, this function returns the value of the ith element from the top of the stack. The element is not deleted by this function.

1.   [check for stack underflow]
     if TOP-I+1<=0

then Write ("stack underflow on peep")
        action in response to underflow
        Exit

2.   [Return Ith element from the top of the
        stack] Return(s[TOP-I+1])

13. Define a Queue and write an algorithm for the operations on Queue?

A queue is an ordered collection of items where the addition of new items happens at one end, called the "rear," and the removal of existing items occurs at the other end, commonly called the "front." As an element enters the queue it starts at the rear and makes its way toward the front, waiting until that time when it is the next element to be removed.

The most recently added item in the queue must wait at the end of the collection. The item that has been in the collection the longest is at the front. This ordering principle is sometimes called **FIFO**, **first-in first-out**. It is also known as "first-come first-served."

A queue is structured, as described above, as an ordered collection of items which are added at one end, called the "rear," and removed from the other end, called the "front." Queues maintain a FIFO ordering property. The queue operations are given below.

- Queue() creates a new queue that is empty. It needs no parameters and returns an empty queue.

- enqueue(item) adds a new item to the rear of the queue. It needs the item and returns nothing.

- dequeue() removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.

- isEmpty() tests to see whether the queue is empty. It needs no parameters and returns a boolean value.

- size() returns the number of items in the queue. It needs no parameters and returns an integer.

Enqueue(queue, n, front, rear, item)//inserts item into a queue

1.   [queue already filled?]

2.   [find new value of rear]

3.   Set queue[rear] := item [inserts new element].

4.   Return;

Dequeue(queue, n, front, rear, item) //deletes element from q and assign it to variable item.

1.   [Queue already empty?]

2.   Set item : = queue[front].

3.   [find new value of front]

4.   Return.