

**'UNIT -I NOTES****Subject:- Algorithm & Data Structures****Semester:- IV****Unit – I****Syllabus**

An Introduction to data structure: Introduction, Definition, Classification of data structure, Concept of data, Data types, Abstract data Types (ADT), Features of structured program. Introduction to algorithms: Definition and Characteristics of an Algorithm, Apriori analysis, Time and space complexity, Average , Best and Worst case complexities, Big „O“ Notations, Asymptotic notations, Top-Down and bottom-up programming techniques, Recursion, Divide and conquer strategy. (e.g. Quick sort,)

Q:1 Give Introduction to Data Structures, Time & space analysis of algorithm**Introduction to Data Structures**

Ans:-Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have data player's name "Virat" and age 26. Here "Virat" is of **String** data type and 26 is of **integer** data type.

We can organize this data as a record like **Player** record. Now we can collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily.

Basic types of Data Structures

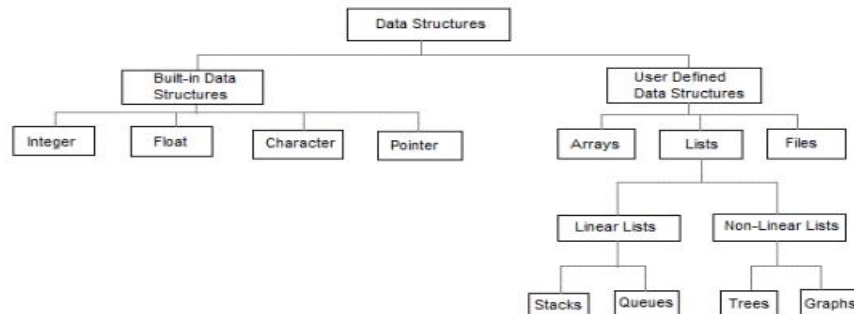
As we discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Abstract Data Structure** are :

- Linked List
- Tree

- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.



INTRODUCTION TO DATA STRUCTURES

Q:2What is Algorithm ?

Ans:-An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as **pseudocode** or using a **flowchart**.

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties:

1. Time Complexity
2. Space Complexity

Space Complexity

Its the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

An algorithm generally requires space for following components :

- **Instruction Space** : Its the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
- **Data Space** : Its the space required to store all the constants and variables value.
- **Environment Space** : Its the space required to store the environment information needed to resume the suspended function.

Time Complexity

Time Complexity is a way to represent the amount of time needed by the program to run to completion. We will study this in details in our section.

Time Complexity of Algorithms

Time complexity of an algorithm signifies the total time required by the program to run to completion. The time complexity of algorithms is most commonly expressed using the **big O notation**.

Time Complexity is most commonly estimated by counting the number of elementary functions performed by the algorithm. And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

Calculating Time Complexity

Now lets tap onto the next big topic related to Time complexity, which is How to Calculate Time Complexity. It becomes very confusing some times, but we will try to explain it in the simplest way.

Now the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity. In general you can think of it like this : statement;

Above we have a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to N.

```
for(i=0; i < N; i++)  
{  
    statement;  
}
```

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for(i=0; i < N; i++)  
{  
    for(j=0; j < N; j++)  
    {  
        statement;  
    }  
}
```

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by $N * N$.

```
while(low <= high)  
{
```

```

mid = (low + high) / 2;
if (target < list[mid])
    high = mid - 1;
else if (target > list[mid])
    low = mid + 1;
else break;
}

```

This is an algorithm to break a set of numbers into halves, to search a particular field (we will study this in detail later). Now, this algorithm will have a **Logarithmic** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2 (N is high-low here). This is because the algorithm divides the working area in half with each iteration.

```

void quicksort(int list[], int left, int right)
{
    int pivot = partition(list, left, right);
    quicksort(list, left, pivot - 1);
    quicksort(list, pivot + 1, right);
}

```

Taking the previous algorithm forward, above we have a small logic of Quick Sort (we will study this in detail later). Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times (where N is the size of list). Hence time complexity will be **$N \cdot \log(N)$** . The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

Q:3 Explain Types of Notations for Time Complexity

Ans:- Now we will discuss and understand the various notations used for Time Complexity.

1. **Big Oh** denotes "*fewer than or the same as*" <expression> iterations.
2. **Big Omega** denotes "*more than or the same as*" <expression> iterations.
3. **Big Theta** denotes "*the same as*" <expression> iterations.

❖ Big-O, Big-Theta, and Big-Omega

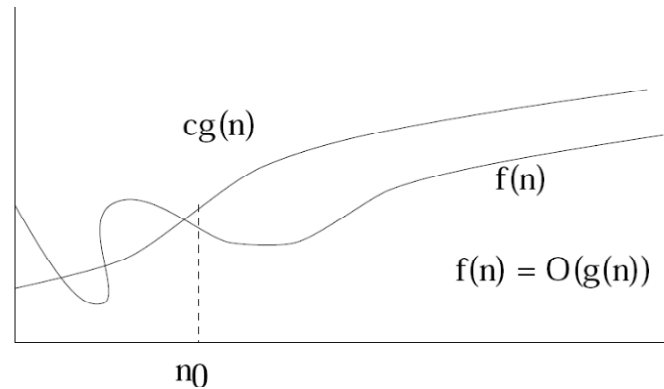
Growth of Functions and Asymptotic Notation

- When we study algorithms, we are interested in characterizing them according to their efficiency.
- We are usually interested in the order of growth of the running time of an algorithm, not in the exact running time. This is also referred to as the *asymptotic running time*.
- We need to develop a way to talk about rate of growth of functions so that we can compare algorithms.
- *Asymptotic notation* gives us a method for classifying functions according to their rate of growth.

Big-O (Oh) Notation:

- **Definition:** $f(n) = O(g(n))$ iff there are two positive constants c and n_0 such that $|f(n)| \leq c |g(n)|$ for all $n \geq n_0$
- If $f(n)$ is nonnegative, we can simplify the last condition to $0 \leq f(n) \leq c g(n)$ for all $n \geq n_0$
- We say that " $f(n)$ is big-O of $g(n)$."

- As n increases, $f(n)$ grows no faster than $g(n)$. In other words, $g(n)$ is an asymptotic upper bound on $f(n)$.



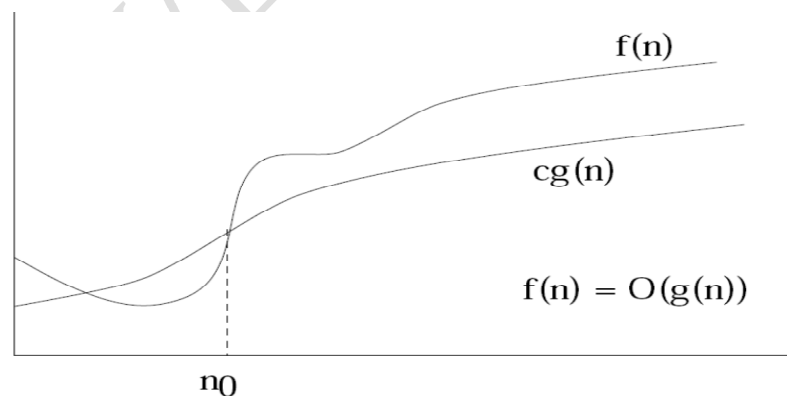
Example: $n^2 + n = O(n^3)$

Proof:

- Here, we have $f(n) = n^2 + n$, and $g(n) = n^3$
- Notice that if $n \geq 1$, $n \leq n^3$ is clear.
- Also, notice that if $n \geq 1$, $n^2 \leq n^3$ is clear.
- Side Note: In general, if $a \leq b$, then $na \leq nb$ whenever $n \geq 1$. This fact is used often in these types of proofs.
- Therefore, $n^2 + n \leq n^3 + n^3 = 2n^3$
- We have just shown that $n^2 + n \leq 2n^3$ for all $n \geq 1$
- Thus, we have shown that $n^2 + n = O(n^3)$ (by definition of Big-O, with $n_0 = 1$, and $c = 2$.)

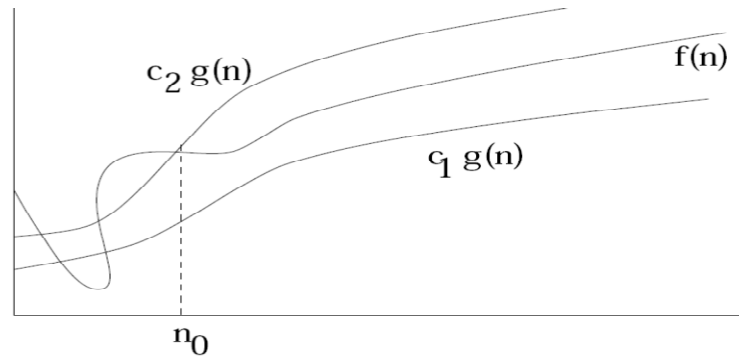
Big - Ω (Omega) Notation

- Definition: $f(n) = \Omega(g(n))$ iff there are two positive constants c and n_0 such that $|f(n)| \geq c|g(n)|$ for all $n \geq n_0$
- If $f(n)$ is nonnegative, we can simplify the last condition to $0 \leq c g(n) \leq f(n)$ for all $n \geq n_0$
- We say that " $f(n)$ is omega of $g(n)$."
- As n increases, $f(n)$ grows no slower than $g(n)$. In other words, $g(n)$ is an asymptotic lower bound on $f(n)$.



Big- Θ (Theta) Notation

- Definition: $f(n) = \Theta(g(n))$ iff there are three positive constants c_1 , c_2 and n_0 such that $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ for all $n \geq n_0$
- If $f(n)$ is nonnegative, we can simplify the last condition to $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$
- We say that " $f(n)$ is theta of $g(n)$."
- As n increases, $f(n)$ grows at the same rate as $g(n)$. In other words, $g(n)$ is an asymptotically tight bound on $f(n)$.



➤ Best, Worst, and Average-Case Complexity

Suppose M is an algorithm and n is the size of input data. The complexity of an algorithm m is the function $f(n)$ which gives the running time and/or storage space requirement of the algorithm in terms of size n of the input data.

The term “Complexity” shall refer to the running time of the algorithm.

To understand the notions of the best, worst, and average-case complexity, one must think about running an algorithm on all possible instances of data that can be fed to it. For the problem of sorting, the set of possible input instances consists of all the possible arrangements of all the possible numbers of keys. We can represent every input instance as a point on a graph, where the x-axis is the size of the problem (for sorting, the number of items to sort) and the y-axis is the number of steps taken by the algorithm on this instance. Here we assume, quite reasonably, that it doesn't matter what the values of the keys are, just how many of them there are and how they are ordered. It should not take longer to sort 1,000 English names than it does to sort 1,000 French names, for example.

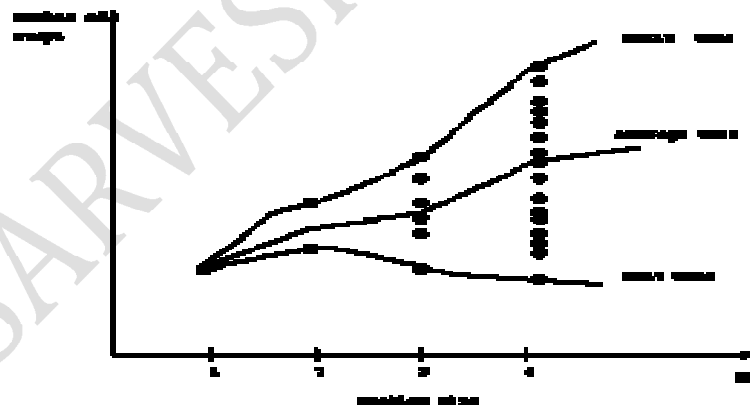


Figure: Best, worst, and average-case complexity

As shown in Figure, these points naturally align themselves into columns, because only integers represent possible input sizes. After all, it makes no sense to ask how long it takes to sort 10.57 items. Once we have these points, we can define three different functions over them:

The worst-case complexity of the algorithm is the function defined by the maximum number of steps taken on any instance of size n . It represents the curve passing through the highest point of each column.

The best-case complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size n . It represents the curve passing through the lowest point of each column.

Finally, **the average-case complexity** of the algorithm is the function defined by the average number of steps taken on any instance of size n .

Q:4 Write Short note on Searching

Ans:-Searching in Linear Array:

The process of finding a particular element of an array is called **Searching**. If the item is not present in the array, then the search is unsuccessful.

There are two types of search (**Linear search** and **Binary Search**)

❖ Linear or Sequential Search

To perform a linear search of data held in an array, the search starts at one end (usually the low numbered element of the array) and examines each element in the array until one of two conditions is met, either Condition 1: the target has been found or Condition 2: the end of the data has been reached (the target value is not in the data set).

Note that the algorithm requires that both tests are performed and that the search terminates when one of the conditions becomes true. The second test is required to prevent the algorithm from attempting to search past the end of the data. For illustration, consider the following data set. Again, element 0 is leftmost:

14	23	7	2	19	9
----	----	---	---	----	---

To search for the value 7 in the array, we start by examining the first element of the array. This does not match the target, so we increment the index counter, and try again. We now examine the next element of the array, which has the value of 23. This does not match the target, so we again increment the counter. This now means that we are examining the element containing 7. This is what we are looking for, so the search is terminated, and the result of the search is reported back to the calling function. It is usual to return the index of the element containing the target, but there may be circumstances where a different return value may be needed.

14	23	7	2	19	9
----	----	---	---	----	---

Linear Search Algorithm

Algorithm: (Linear Search)

LINEAR (A, SKEY)

Here **A** is a Linear Array with N elements and **SKEY** is a given item

of information to search. This algorithm finds the location of SKEY in A and if successful, it returns its location otherwise it returns -1 for unsuccessful.

1. Repeat for i = 0 to N-1
2. if(A[i] = SKEY) return i [Successful Search]
[End of loop]
3. return -1 [Un-Successful]
4. Exit.

Program for Linear Search:

Linear search in c programming: The following code implements linear search (Searching algorithm) which is used to find whether a given number is present in an array and if it is present then at what location it occurs. It is also known as sequential search. It is very simple and works as follows: We keep on comparing each element with the element to search until the desired element is found or list ends.

```
#include <stdio.h>

int main()
{
    int array[100], search, c, n;

    printf("Enter the number of elements in array\n");
    scanf("%d",&n);

    printf("Enter %d integer(s)\n", n);

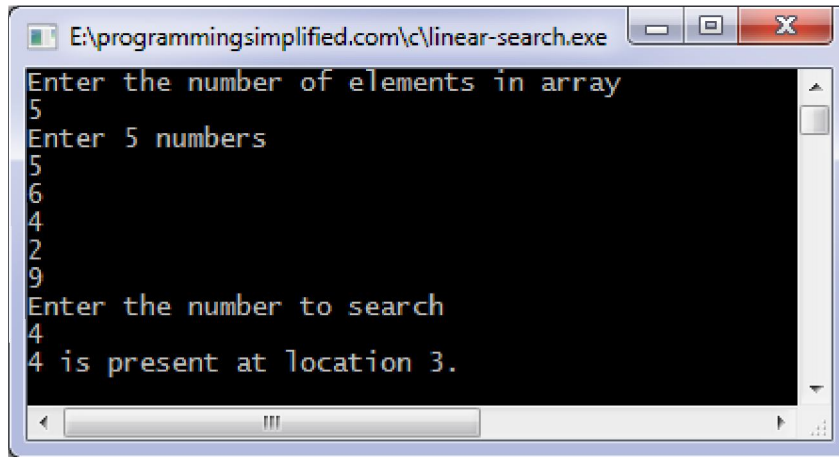
    for (c = 0; c < n; c++)
        scanf("%d", &array[c]);

    printf("Enter the number to search\n");
    scanf("%d", &search);

    for (c = 0; c < n; c++)
    {
        if (array[c] == search)  /* if required element found */
        {
            printf("%d is present at location %d.\n", search, c+1);
            break;
        }
    }
    if (c == n)
        printf("%d is not present in array.\n", search);

    return 0;
}
```


Output:



```
E:\programmingsimplified.com\c\linear-search.exe
Enter the number of elements in array
5
Enter 5 numbers
5
6
4
2
9
Enter the number to search
4
4 is present at location 3.
```

Complexity:

Worst case performance	$O(n)$
Best case performance	$O(1)$
Average case performance	$O(n)$
Worst case space complexity	$O(n)$

2) Binary Search

Binary search is also known as binary chop, as the data set is cut into two halves for each step of the process. It is a very much faster search method than linear search, but to be effective the data set must be in sorted order in the array. If the data set changes rapidly and requires regular re-sorting then this will offset the speed gain offered by binary search over linear search.

To perform binary search, three index variables are required. By tradition these are called 'top', 'middle' and 'bottom'.

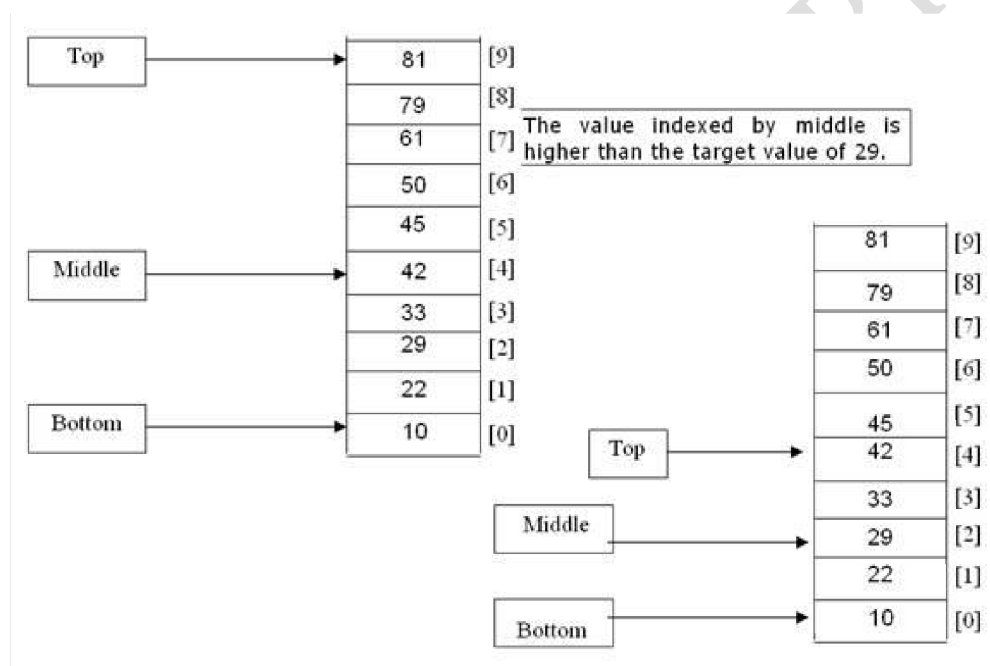
Top is initialized to one end of the array, often 0, and bottom is set to indicate the other end of the array.

Once these two variables are set, the value of middle can be computed. Middle is set to the midway value between top and bottom.

The value indexed by middle is compared with the target value. There are initially three possible outcomes that we have to consider:

- 1: The value indexed by middle matches the target. In this case the search has found the target and the function can return a value indicating that the search has succeeded.
2. The value is higher than the middle value in which case only the values from middle to end need to be searched
3. The value is lower than the middle value in which case the search is carried out between zero and the middle value.

To illustrate this process, consider the following scenario - the data in the array is sorted and the target is 29. We start by setting top to 9, bottom to 0, and calculating middle to be $(0 + 9) / 2$. This rounds down to 4 using C integer arithmetic, so middle is set to 4.



From this we can conclude that the target value is in the lower half of the table. This means that top must be set to middle and a new value of middle calculated. In this case the value of middle will be $(4 + 0) / 2$ which C will deliver as 2. The contents of array element 2 matches the target, so in this case the search is successfully concluded.

3) Binary Search Algorithm

Search as the name suggests, is an operation of finding an item from the given collection of items. Binary Search algorithm is used to find the position of a specified value (an 'Input Key') given by the user in a sorted list.

Algorithm:

Here **A** is a sorted Linear Array with **N** elements and **SKEY** is a given item of information to search. This algorithm finds the location of **SKEY** in **A** and if successful, it returns its location otherwise it returns -1 for unsuccessful.

BinarySearch (A, SKEY)

1. [Initialize segment variables.]

Set START=0, END=N-1 and MID=INT((START+END)/2).

2. Repeat Steps 3 and 4 while START ≤ END and A[MID]≠SKEY.

3. If SKEY < A[MID]. Then

Set END=MID-1.

Else Set START=MID+1.

[End of If Structure.]

4. Set MID=INT((START +END)/2).

[End of Step 2 loop.]

5. If A[MID]= SKEY then Set LOC= MID

Else:

Set LOC = -1

[End of IF structure.]

6. return LOC and Exit

Program for Binary Search:

This code implements binary search in c language. It can only be used for sorted arrays, but it's fast as compared to linear search. If you wish to use binary search on an array which is not sorted then you must sort it using some sorting technique say merge sort and then use binary search algorithm to find the desired element in the list. If the element to be searched is found then its position is printed.

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

```
int main()
```

```
{
```

```
int c, first, last, middle, n, search, array[100];
```

```
printf("Enter number of elements\n");
```

```
scanf("%d",&n);
```

```
printf("Enter %d integers\n", n);
```

```
for ( c = 0 ; c < n ; c++ )
```

```
scanf("%d",&array[c]);
```

```
printf("Enter value to find\n");
```

```
scanf("%d",&search);
```

```
first = 0;
```

```
last = n - 1;
```

```
middle = (first+last)/2;
```

```

while( first <= last )
{
    if ( array[middle] < search )
        first = middle + 1;
    else if ( array[middle] == search )
    {
        printf("%d found at location %d.\n", search, middle+1);
        break;
    }
    else
        last = middle - 1;

    middle = (first + last)/2;
}
if ( first > last )
    printf("Not found! %d is not present in the list.\n", search);

return 0;
}

```

Output:

```

E:\programmingsimplified.com\c\binary-search.exe
Enter number of elements
7
Enter 7 integers
-4
5
8
9
11
43
485
Enter value to find
11
11 found at location 5.

```

Complexity:

Worst case performance	$O(\log n)$
Best case performance	$O(1)$
Average case performance	$O(\log n)$
Worst case space complexity	$O(1)$

Q:5 Explain Sorting in detail.

Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order. The term Sorting comes into picture with the term Searching. There are so many things in our real life that we need to search, like a particular record in database, roll numbers in merit list, a particular telephone number, any particular page in a book etc.

Sorting arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted. For example, suppose we have a record of students, every such record will have the following data:

- Roll No.
- Name
- Age
- Class

Here Student roll no. can be taken as key for sorting the records in ascending or descending order. Now suppose we have to search a Student with roll no. 15, we don't need to search the complete record we will simply search between the Students with roll no. 10 to 20.

Sorting Efficiency

There are many techniques for sorting. Implementation of particular sorting technique depends upon situation. Sorting techniques mainly depends on two parameters. First parameter is the execution time of program, which means time taken for execution of program. Second is the space, which means space taken by the program.

Types of Sorting Techniques

There are many types of Sorting techniques, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering in next sections.

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

1. Selection Sort:

Selection sorting is conceptually the simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then finds the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

Example:

Original Array	After 1st pass	After 2nd pass	After 3rd pass	After 4th pass	After 5th pass
3 6 1 8 4 5	1 --- 6 3 8 4 5	1 3 --- 6 8 4 5	1 3 4 --- 8 6 5	1 3 4 5 6 8	1 3 4 5 6 8

In the first pass, the smallest element found is 1, so it is placed at the first position, then leaving first element, smallest element is searched from the rest of the elements, 3 is the smallest, so it is then placed at the second position. Then we leave 1 and 3, from the rest of the elements, we search for the smallest and put it at third position and keep doing this, until array is sorted.

Program for Selection Sort:

This code implements selection sort algorithm to arrange numbers of an array in ascending order. With a little modification it will arrange numbers in descending order.

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

```
int main()
```

```
{
```

```
    int array[100], n, c, d, position, swap;
```

```
    printf("Enter number of elements\n");
```

```
    scanf("%d", &n);
```

```
    printf("Enter %d integers\n", n);
```

```
    for ( c = 0 ; c < n ; c++ )
```

```
        scanf("%d", &array[c]);
```

```
    for ( c = 0 ; c < ( n - 1 ) ; c++ )
```

```
    {
```

```
        position = c;
```

```
        for ( d = c + 1 ; d < n ; d++ )
```

```
        {
```

```
            if ( array[position] > array[d] )
```

```
                position = d;
```

```
        }
```

```
    } if ( position != c )
```

```

    {
        swap = array[c];
        array[c] = array[position];
        array[position] = swap;
    }
}

printf("Sorted list in ascending order:\n");

for ( c = 0 ; c < n ; c++ )
    printf("%d\n", array[c]);

return 0;
}

```

Output of program:

```

E:\programmingsimplified.com\c\selection-sort.exe
Enter number of elements
10
Enter 10 integers
12
8
-6
2
4
5
3
7
4
2
Sorted list in ascending order:
-6
2
2
3
4
4
5
7
8
12

```

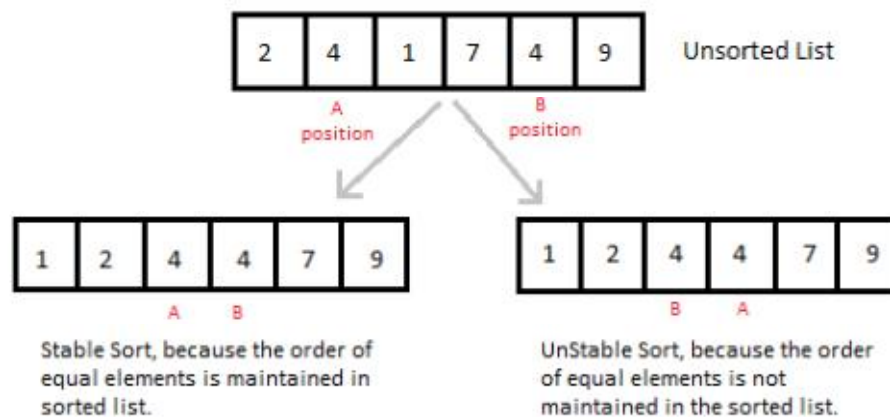
Complexity:

Worst case performance	$O(n^2)$
Best case performance	$O(n^2)$
Average case performance	$O(n^2)$
Worst case space complexity	$O(n)$ total, $O(1)$ auxiliary

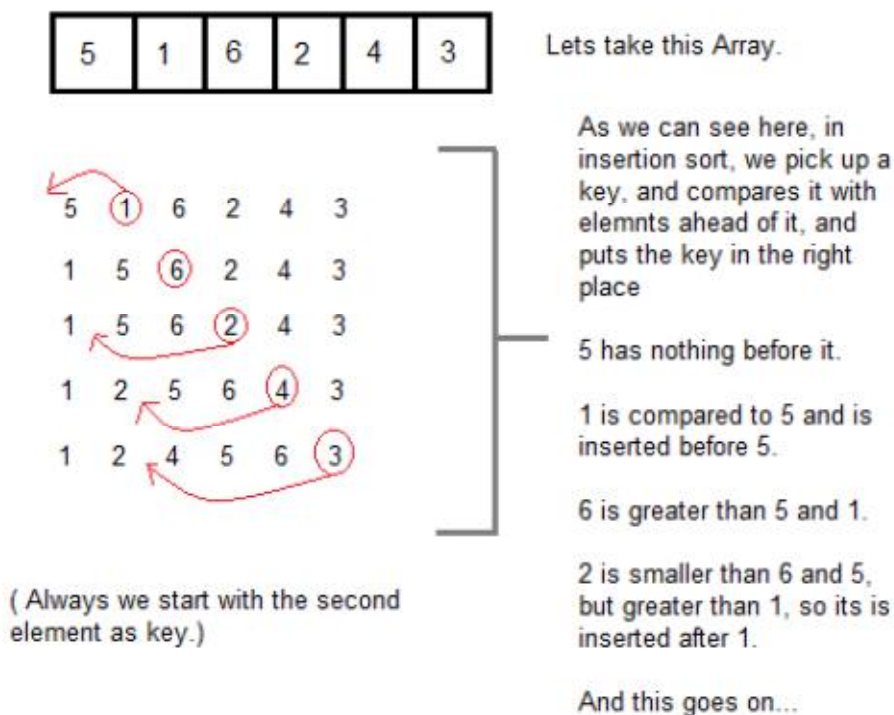
2. Insertion Sort:

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

1. It has one of the simplest implementation
2. It is efficient for smaller data sets, but very inefficient for larger lists.
3. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
4. It is better than Selection Sort and Bubble Sort algorithms.
5. Its space complexity is less, like Bubble Sorting, insertion sort also requires a single additional memory space.
6. It is **Stable**, as it does not change the relative order of elements with equal keys



Example:



Program for Insertion Sort:

This code implements insertion sort algorithm to arrange numbers of an array in ascending order. With a little modification it will arrange numbers in descending order.


```

#include <stdio.h>

int main()
{
    int n, array[1000], c, d, t;

    printf("Enter number of elements\n");
    scanf("%d", &n);

    printf("Enter %d integers\n", n);

    for (c = 0; c < n; c++)
    {
        scanf("%d", &array[c]);
    }

    for (c = 1; c <= n - 1; c++)
    {
        d = c;

        while ( d > 0 && array[d] < array[d-1])
        {
            t = array[d];
            array[d] = array[d-1];
            array[d-1] = t;

            d--;
        }
    }

    printf("Sorted list in ascending order:\n");

    for (c = 0; c <= n - 1; c++) {
        printf("%d\n", array[c]);
    }

    return 0;
}

```

Output of program:

```

E:\programmingsimplified.com\c\insertion-sort.exe
Enter number of elements
5
Enter 5 integers
4
3
-1
2
1
Sorted list in ascending order:
-1
1
2
3
4

```

Complexity:

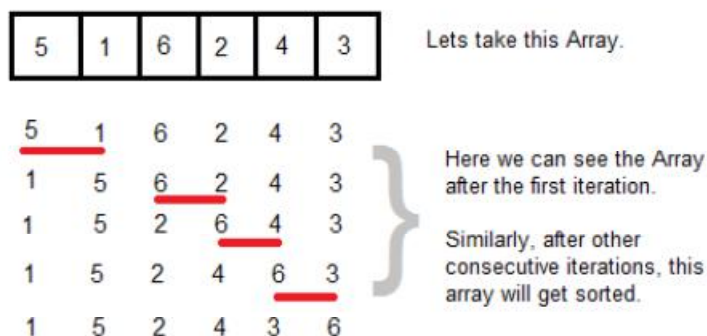
Worst case performance	$O(n^2)$ comparisons, swaps
Best case performance	$O(n)$ comparisons, $O(1)$ swaps
Average case performance	$O(n^2)$ comparisons, swaps
Worst case space complexity	$O(n)$ total, $O(1)$ auxiliary

3. Bubble Sort:

Bubble Sort is an algorithm which is used to sort N elements that are given in a memory for eg: an Array with N number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.



Bubble Sort Algorithm:

```

bubbleSort( A : list of sortable items )
  n = length(A)

```

```

repeat
  swapped = false
  for i = 1 to n-1 inclusive do
    /* if this pair is out of order */
    if A[i-1] > A[i] then
      /* swap them and remember something changed */
      swap( A[i-1], A[i] )
      swapped = true
    end if
  end for
until not swapped
end procedure

```

Program for Bubble Sort :

Code for bubble sort to sort numbers or arrange them in ascending order. You can easily modify it to print numbers in descending order.

```

#include <stdio.h>

int main()
{
  int array[100], n, c, d, swap;

  printf("Enter number of elements\n");
  scanf("%d", &n);

  printf("Enter %d integers\n", n);

  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);

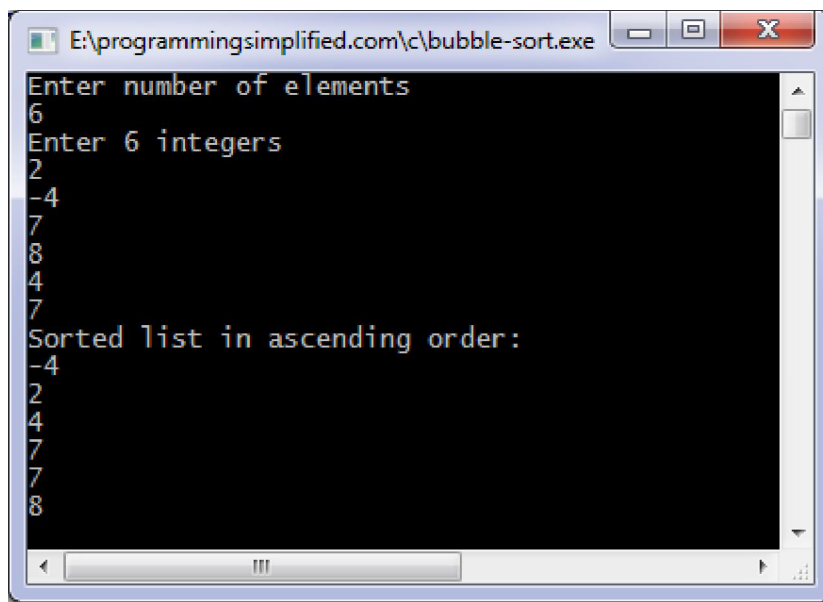
  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (array[d] > array[d+1]) /* For decreasing order use < */
      {
        swap = array[d];
        array[d] = array[d+1];
        array[d+1] = swap;
      }
    }
  }
  printf("Sorted list in ascending order:\n");

  for ( c = 0 ; c < n ; c++ )
    printf("%d\n", array[c]);

  return 0;
}

```

Output of program:



```
E:\programmingsimplified.com\c\bubble-sort.exe
Enter number of elements
6
Enter 6 integers
2
-4
7
8
4
7
Sorted list in ascending order:
-4
2
4
7
7
8
```

Complexity:

Worst case performance	$O(n^2)$
Best case performance	$O(n)$
Average case performance	$O(n^2)$
Worst case space complexity	$O(1)$ auxiliary

4. Radix Sort:

Radix Sort is a clever and intuitive little sorting algorithm. Radix Sort puts the elements in order by comparing the **digits of the numbers**. I will explain with an example.

Consider the following 9 numbers:

493 812 715 710 195 437 582 340 385

We should start sorting by comparing and ordering the **one's** digits:

Digit	Sublist
0	340 710
1	
2	812 582
3	493
4	
5	715 195 385
6	

7	437
8	
9	

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again:

340 710 812 582 493 715 195 385 437

Note: The **order** in which we divide and reassemble the list is **extremely important**, as this is one of the foundations of this algorithm.

Now, the sublists are created again, this time based on the **ten's** digit:

Digit	Sublist
0	
1	710 812 715
2	
3	437
4	340
5	
6	
7	
8	582 385
9	493 195

Now the sublists are gathered in order from 0 to 9:

710 812 715 437 340 582 385 493 195

Finally, the sublists are created according to the **hundred's** digit:

Digit	Sublist
0	
1	195
2	
3	340 385
4	437 493
5	582
6	
7	710 715
8	812
9	

At last, the list is gathered up again:

195 340 385 437 493 582 710 715 812

And now we have a fully sorted array! Radix Sort is very simple, and a computer can do it fast. When it is programmed properly, Radix Sort is in fact **one of the fastest sorting algorithms** for numbers or strings of letters.

Program For Radix Sort:

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

radix_sort(int array[], int n);
void main()
{
    int array[100],n,i;
    clrscr();
    printf("Enter the number of elements to be sorted: ");
    scanf("%d",&n);
    printf("\nEnter the elements to be sorted: \n");
    for(i = 0 ; i < n ; i++ )
    {
        printf("\tArray[%d] = ",i);
        scanf("%d",&array[i]);
    }

    printf("\nArray Before Radix Sort:"); //Array Before Radix Sort
    for(i = 0; i < n; i++)
    {
        printf("%8d", array[i]);
    }
    printf("\n");

    radix_sort(array,n);

    printf("\nArray After Radix Sort: "); //Array After Radix Sort
    for(i = 0; i < n; i++)
    {
        printf("%8d", array[i]);
    }
    printf("\n");
    getch();
}

radix_sort(int arr[], int n)
{
    int bucket[10][5],buck[10],b[10];
    int i,j,k,l,num,div,large,passes;

    div=1;
    num=0;
    large=arr[0];

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

```

{
    if(arr[i] > large)
    {
        large = arr[i];
    }

    while(large > 0)
    {
        num++;
        large = large/10;
    }

    for(passes=0 ; passes<num ; passes++)
    {
        for(k=0 ; k<10 ; k++)
        {
            buck[k] = 0;
        }
        for(i=0 ; i<n ; i++)
        {
            l = ((arr[i]/div)%10);
            bucket[l][buck[l]++] = arr[i];
        }

        i=0;
        for(k=0 ; k<10 ; k++)
        {
            for(j=0 ; j<buck[k] ; j++)
            {
                arr[i++] = bucket[k][j];
            }
        }
        div*=10;
    }
}

```

Output of Program:

```

Enter the number of elements to be sorted: 7
Enter the elements to be sorted:
    Array[0] = 777
    Array[1] = 269
    Array[2] = 158
    Array[3] = 341
    Array[4] = 265
    Array[5] = 989
    Array[6] = 506
Array Before Radix Sort:    777    269    158    341    265    989    506
Array After Radix Sort:    158    265    269    341    506    777    989

```

Complexity of Radix Sort:

Worst case performance	$O(kN)$
Worst case space complexity	$O(k + N)$

5. Shell Sort:

One of the sources of inefficiency in the sorts that we have discussed so far is that the amount of 'unsortedness' reduces slowly as items are moved small distances or only limited numbers of items are moved at a time. Shell sort is a well established sort that aims to overcoming these limitations by moving many items large distances at a time.

Conceptually, Shell sort divides the data set into a number of smaller arrays and performs an insertion sort on these smaller arrays. There is some guidance on the initial number of slices to divide the array into, but every data set seems to perform slightly differently. Here we take a conventional approach and divide the set into three initially.

Consider the dataset:

16 4 3 13 5 6 8 9 10 11 12 17 15 18 19 7 1 2 14 20

We can divide this into three smaller slices:

16 4 3 13 5 6 8
9 10 11 12 17 15 18
19 7 1 2 14 20

Once we have performed this slicing we can sort each slice:

First we sort the first column (16 9 and 19) to give 9 16 19

Next the second column (4 10 and 7) to give 4 7 10

Column three (3 11 and 1) to give 1 3 11

Column four (13 12 and 2) to give 2 12 13

Column five (5 17 and 14) to give 5 14 17

Column six (6 15 and 20) to give 6 15 20

and column seven (8 and 18) to give 8 18

Reassembling the array we now have:

9 4 1 2 5 6 8 16 7 3 12 14 15 18 19 10 11 13 17 20

We now slice the array into a different number of slices. For this example we will use five slices, but other values are possible.

9 4 1 2 5 6 8 16 7 3 12 14 15 18 19 10 11 13 17 20

Slicing this into five we get:

9	4	1	2
5	6	8	16
7	3	12	14
15	18	19	10
11	13	17	20

Again we sort each column to give:

5	3	1	2
7	4	8	10
9	6	12	14
11	13	17	16
15	18	19	20

Logically reassembling, we now have the dataset:

5	3	1	2	7	4	8	10	9	6	12	14	11	13	17	16	15	18	19	20
---	---	---	---	---	---	---	----	---	---	----	----	----	----	----	----	----	----	----	----

We can now slice the array into 10:

5	3
1	2
7	4
8	10
9	6
12	14
11	13
17	16
15	18
19	20

Sorting the columns gives us:

1	2
5	3
7	4
8	6
9	10

11 13
12 14
15 16
17 18
19 20

Reassembling we get:

1 2 5 3 7 4 8 6 9 10 11 13 12 14 15 16 17 18 19 20

The majority of the elements are now near to where they should be, and the last pass of the algorithm is a conventional insertion sort.

Final Sorted List

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----

Program for Shell Sort:

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.     int array[5]={4,5,2,3,6},i1=0;
6.     ShellSort(array,5);
7.     printf("After Sorting:");
8.     for(i1=0;i1<5;i1++)
9.         {printf("%d",array[i1]);
10.        }
11.    return 0;
12. }
13. void ShellSort(int *array, int number_of_elements)
14. {
15.     int i, j, increment, temp;
16.     for(increment = number_of_elements/2;increment > 0; increment /= 2)
17.     {
18.         for(i = increment; i<number_of_elements; i++)
19.         {
20.             temp = array[i];
21.             for(j = i; j >= increment ;j-=increment)
22.             {
23.                 if(temp < array[j-increment])
24.                 {
25.                     array[j] = array[j-increment];
26.                 }
27.             }
28.             else
29.             {
30.                 break;
31.             }
32.         }
33.     }
34. }
```

```

30.}
31.}
32.array[j] = temp;
33.}
34.}
35.}

```

Output of Program:

Complexity Shell Sort:

Worst case performance	$O(n^2)$
Best case performance	$O(n \log^2 n)$
Average case performance	depends on gap sequence
Worst case space complexity	$O(n)$ total, $O(1)$ auxiliary

Q:9 Explain Quick Sort Algorithm

Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less additional space. It is based on the rule of **Divide and Conquer** (also called *partition-exchange sort*). This algorithm divides the list into three main parts

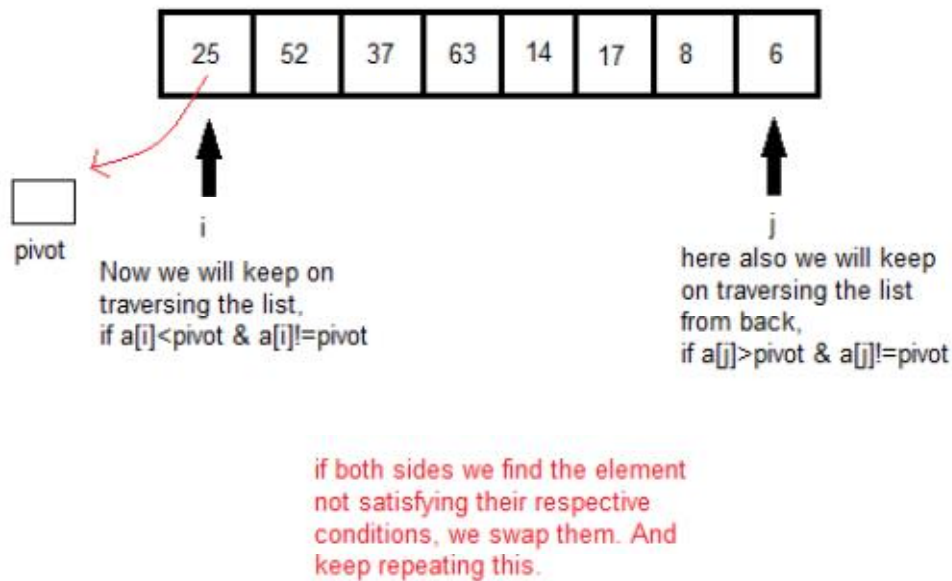
1. Elements less than the Pivot element
2. Pivot element
3. Elements greater than the pivot element

In the list of elements, mentioned in below example, we have taken **25** as pivot. So after the first pass, the list will be changed like this.

6 8 17 14 **25** 63 37 52

Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than it on the right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep doing this until the complete list is sorted.

Example:



DIVIDE AND CONQUER - QUICK SORT

Algorithm Quick Sort:

int function Partition (Array A, int Lb, int Ub);

begin

select a pivot from $A[Lb] \dots A[Ub]$;

reorder $A[Lb] \dots A[Ub]$ such that:

all values to the left of the pivot are \leq pivot

all values to the right of the pivot are \geq pivot

return pivot position;

end;

procedure QuickSort (Array A, int Lb, int Ub);

begin

if $Lb < Ub$ then

$M = \text{Partition}(A, Lb, Ub)$;

QuickSort (A, Lb, $M - 1$);

QuickSort (A, M , Ub);

end;

Program for Quick Sort:

/* $a[]$ is the array, p is starting index, that is 0,
and r is the last index of array. */

void **quicksort**(int $a[]$, int p , int r)

```

{
  if(p < r)
  {
    int q;
    q = partition(a, p, r);
    quicksort(a, p, q);
    quicksort(a, q+1, r);
  }
}

```

```

int partition(int a[], int p, int r)
{
  int i, j, pivot, temp;
  pivot = a[p];
  i = p;
  j = r;
  while(1)
  {
    while(a[i] < pivot && a[i] != pivot)
      i++;
    while(a[j] > pivot && a[j] != pivot)
      j--;
    if(i < j)
    {
      temp = a[i];
      a[i] = a[j];
      a[j] = temp;
    }
    else
    {
      return j;
    }
  }
}

```

Complexity :

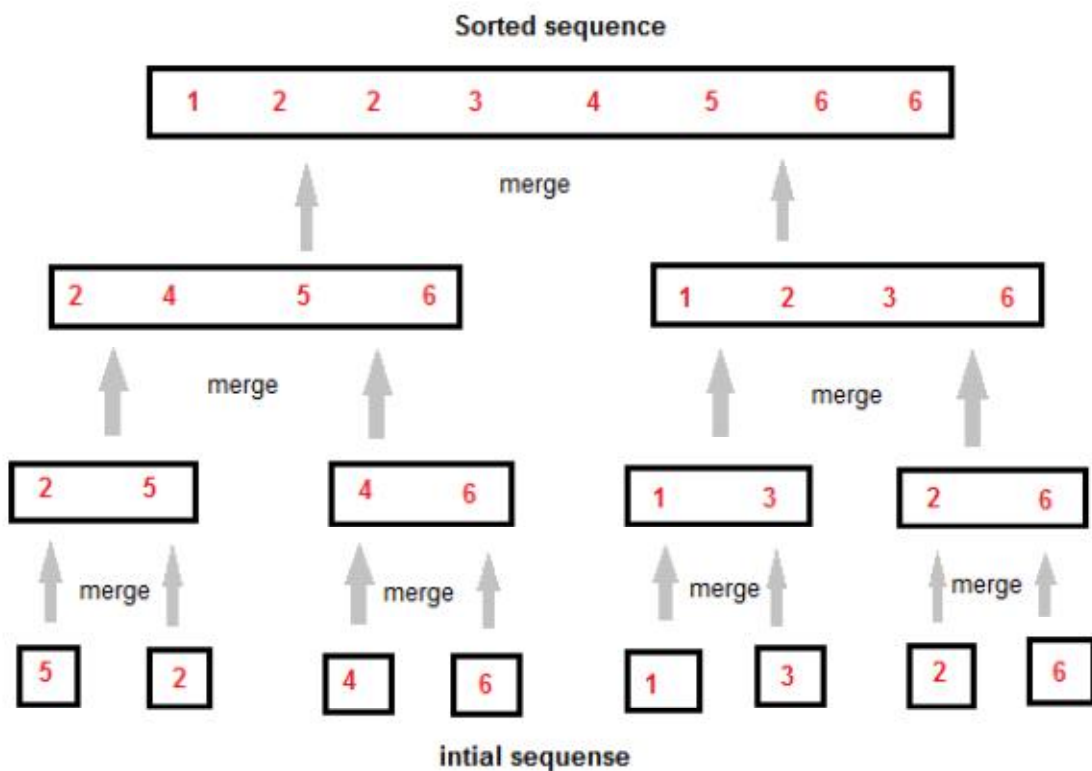
<u>Worst case performance</u>	$O(n^2)$
<u>Best case performance</u>	$O(n \log n)$ (simple partition) or $O(n)$ (three-way partition and equal keys)
<u>Average case performance</u>	$O(n \log n)$
<u>Worst case space complexity</u>	$O(n)$ auxiliary (naive) $O(\log n)$ auxiliary

Q:10 Explain Merge Sort Algorithm

Merge Sort follows the rule of **Divide and Conquer**. But it doesn't divide the list into two halves. In merge sort the unsorted list is divided into N sublists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merges these sublists, to produce new sorted sublists, and at last one sorted list is produced.

Merge Sort is quite fast, and has a time complexity of $O(n \log n)$. It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

Example:



Like we can see in the above example, merge sort first breaks the unsorted list into sorted sublists, and then keeps merging these sublists, to finally get the complete sorted list.

Program for Merge Sort:

/* a[] is the array, p is starting index, that is 0,
and r is the last index of array. */

Lets take $a[5] = \{32, 45, 67, 2, 7\}$ as the array to be sorted.

```
void mergesort(int a[], int p, int r)
{
    int q;
    if(p < r)
    {
```

```

    q = floor( (p+r) / 2);
    mergesort(a, p, q);
    mergesort(a, q+1, r);
    merge(a, p, q, r);
}
}

```

void **merge**(int a[], int p, int q, int r)

```

{
    int b[5];    //same size of a[]
    int i, j, k;
    k = 0;
    i = p;
    j = q+1;
    while(i <= q && j <= r)
    {
        if(a[i] < a[j])
        {
            b[k++] = a[i++];    // same as b[k]=a[i]; k++; i++;
        }
        else
        {
            b[k++] = a[j++];
        }
    }

    while(i <= q)
    {
        b[k++] = a[i++];
    }

    while(j <= r)
    {
        b[k++] = a[j++];
    }

    for(i=r; i >= p; i--)
    {
        a[i] = b[--k];    // copying back the sorted list to a[]
    }
}

```

Complexity :

<u>Worst case performance</u>	$O(n \log n)$
<u>Best case performance</u>	$O(n \log n)$ typical, $O(n)$ natural variant

Average case performance	$O(n \log n)$
Worst case space complexity	$O(n)$ auxiliary

8. Heap Sort :

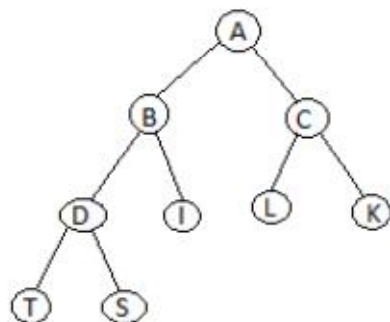
Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts :

- Creating a Heap of the unsorted list.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

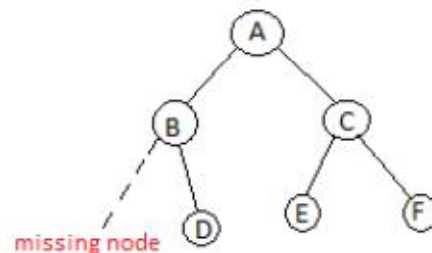
What is a Heap ?

Heap is a special tree-based data structure, that satisfies the following special heap properties :

1. **Shape Property** : Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.

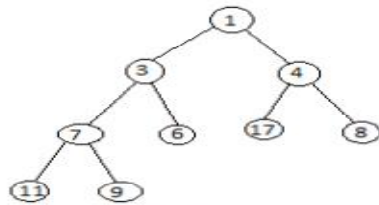


Complete Binary Tree



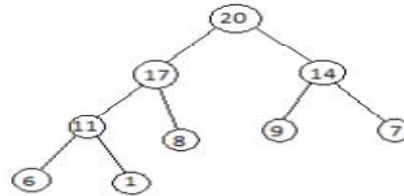
In-Complete Binary Tree

2. **Heap Property** : All nodes are either *[greater than or equal to]* or *[less than or equal to]* each of its children. If the parent nodes are greater than their children, heap is called a **Max-Heap**, and if the parent nodes are smaller than their child nodes, heap is called **Min-Heap**.



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

How Heap Sort Works

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure(Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest(depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly untill we have the complete sorted list in our array.

In the below algorithm, initially **heapsort()** function is called, which calls **buildheap()** to build heap, which inturn uses **satisfyheap()** to build the heap.

Program for Heap Sort:

/* Below program is written in C++ language */

```

void heapsort(int[], int);
void buildheap(int [], int);
void satisfyheap(int [], int, int);

void main()
{
    int a[10], i, size;
    cout << "Enter size of list"; // less than 10, because max size of array is 10
    cin >> size;
    cout << "Enter" << size << "elements";
    for( i=0; i < size; i++)
    {
        cin >> a[i];
    }
    heapsort(a, size);
    getch();
}
  
```

```

void heapsort(int a[], int length)
  
```

```

{
    buildheap(a, length);
    int heapsize, i, temp;
    heapsize = length - 1;
    for( i=heapsize; i >= 0; i--)
    {
        temp = a[0];
        a[0] = a[heapsize];
        a[heapsize] = temp;
        heapsize--;
        satisfyheap(a, 0, heapsize);
    }
    for( i=0; i < length; i++)
    {
        cout << "\t" << a[i];
    }
}

```

void **buildheap**(int a[], int length)

```

{
    int i, heapsize;
    heapsize = length - 1;
    for( i=(length/2); i >= 0; i--)
    {
        satisfyheap(a, i, heapsize);
    }
}

```

void **satisfyheap**(int a[], int i, int heapsize)

```

{
    int l, r, largest, temp;
    l = 2*i;
    r = 2*i + 1;
    if(l <= heapsize && a[l] > a[i])
    {
        largest = l;
    }
    else
    {
        largest = i;
    }
    if( r <= heapsize && a[r] > a[largest])
    {
        largest = r;
    }
    if(largest != i)
    {
        temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;
        satisfyheap(a, largest, heapsize);
    }
}

```

```

}
}

```

Complexity :

<u>Worst case performance</u>	$O(n \log n)$
<u>Best case performance</u>	$\Omega(n), O(n \log n)$
<u>Average case performance</u>	$O(n \log n)$
<u>Worst case space complexity</u>	$O(1)$ auxiliary

➤ **Complexity Comparison o Sorting Techniques:**

Technique	Time					
Sort	Average	Best	Worst	Space	Stability	Remarks
<u>Bubble sort</u>	$O(n^2)$	$O(n^2)$	$O(n^2)$	Constant	Stable	Always use a modified bubble sort
<u>Selection Sort</u>	$O(n^2)$	$O(n^2)$	$O(n^2)$	Constant	Stable	Even a perfectly sorted input requires scanning the entire array
<u>Insertion Sort</u>	$O(n^2)$	$O(n)$	$O(n^2)$	Constant	Stable	In the best case (already sorted), every insert requires constant time
<u>Heap Sort</u>	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	Constant	Instable	By using input array as storage for the heap, it is possible to achieve constant space
<u>Merge Sort</u>	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	Depends	Stable	On arrays, merge sort requires $O(n)$ space; on linked lists, merge sort requires constant space
<u>Quicksort</u>	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	Constant	Stable	Randomly picking a pivot value

						(or shuffling the array prior to sorting) can help avoid worst case scenarios such as a perfectly sorted array.
--	--	--	--	--	--	---

➤ Representation of Array:

Array definition:-

Array is linear, homogeneous data structures whose elements are stored in contiguous memory locations.

Arrays are subscripted variables stored in contiguous memory locations.

Types of Arrays:

One-dimensional Array or linear array: requires only one index to access an element.

Two-Dimensional Array: requires two indices to access an element.

Multidimensional Array: requires two or more indices to access an element.

Size of linear array: $\text{size} = \text{ub} - \text{lb} + 1$

Where ub represents upper bound or largest index of array,
lb represents lower bound or smallest index of array.

Indices of array are integer numbers.

In C index starts from 0, that is the smallest index of array is 0.

In C index are written in brackets [].

Representation of One-dimensional array in memory:-

Suppose name of linear array is arr and it has 5 elements. Then its elements are represented as:

arr				
0	1	2	3	4
arr[0],	arr[1],	arr[2],	arr[3],	arr[4].

Address calculation in one-dimensional Array:-

Since array elements are stored in contiguous memory locations, the computer needs to not to know the address of every element but the address of only first element. The address

of first element is called base address of array. Given the address of first element, address of any other element is calculated using the formula:-

$$\text{Loc (arr [k])} = \text{base (arr)} + w * k \text{ (in C)}$$

Where k is the index of array whose address we want to calculate and w is the number of bytes per storage location of for one element of array.

If not given explicitly, we take index set as 1,2,3,4,.....n where n the upper bound of the array.

Example:- Suppose that array arr is declared as integers with size 20 and its first element is stored at address 1000. Calculate the address of 4th element of array when the index of the array starts from 0.

Here, base address=1000, k=3, w=2.

Thus, loc(arr[3])=1000 + 2*3=1006

Representation of two dimensional array in memory:

Suppose name of two-dimensional array is mat and it has 3 rows and 4 columns. Then its elements are represented as:

mat[0][0], mat[1][0], mat[2][0]

mat[0][1], mat[1][1], mat[2][1]

mat[0][2], mat[1][2], mat[2][2]

mat

	0	1	3
0			
1			
2			

Elements of two-dimensional arrays are stored in two ways:-

- (ii) Column major order: Elements are stored column by column, i.e. all elements of first column are stored, and then all elements of second column stored and so on.

mat[0][0]
Column 0 mat[1][0]
mat[2][0]

mat[0][1]
Column 1 mat[1][1]
mat[2][1]
mat[0][2]
Column 2 mat[1][2]
mat[2][2]

- (iii) **Row major order:** Elements are stored row by row, i.e. all elements of first row are stored, and then all elements of second row stored and so on.

mat[0][0]
Row 0 mat[0][1]
mat[0][2]
mat[1][0]
Row 1 mat[1][1]
mat[1][2]

mat[2][1]
mat[2][0]
Row 2
mat[2][2]

Ordered List:

A list in which the elements are arranged so that the key values are placed in ascending or descending sequence.

Ordered Lists are maintained in sequence according to the data or, when available a key that identifies the data.

Sparse Matrices:-

An $m \times n$ matrix A is said to be sparse if many of its elements are zero. A matrix that is not sparse is called dense matrix. It is not possible to define an exact boundary between dense and sparse matrices.

Q:12 Write Short note on Abstract Data Structure:

Notice that your code may be understood and augmented by third parties in your absence. Even if you flood your code with documentation, its readability is not ensured. An important thing you require is a design document. That is not at the programming level, but at a more abstract level.

Data abstraction is the first step. A problem is a problem of its own nature. It deals with input and output in specified formats not related to any computer program. For example, a weather forecast system reads gigantic databases and outputs some prediction. Where is C coming in the picture in this behavioral description? One can use any other computer language, perhaps assembly languages, or even hand calculations, to arrive at the solution.

What is an abstract data type?

An abstract data type (ADT) is an object with a generic description independent of implementation details. This description includes a specification of the components from which the object is made and also the behavioral details of the object. Instances of abstract objects include mathematical objects (like numbers, polynomials, integrals, vectors), physical objects (like pulleys, floating bodies, missiles), animate objects (dogs, Pterodactyls, Indians) and objects (like poverty, honesty, inflation) that are abstract even in the natural language sense. You do not see C in Pterodactyls. Only when you want to simulate a flying Pterodactyl, you would think of using a graphics package in tandem with a computer language. Similarly, inflation is an abstract concept. When you want to model it and want to predict it for the next 10 years, you would think of writing an extrapolation program in C.

Specifying only the components of an object does not suffice. Depending on the problem you are going to solve, you should also identify the properties and behaviors of the object and perhaps additionally the pattern of interaction of the object with other objects of same and/or different types. Thus in order to define an ADT we need to specify:

- The components of an object of the ADT.
- A set of procedures that provide the behavioral description of objects belonging to the ADT.

There may be thousands of ways in which a given ADT can be implemented, even when the coding language remains constant. Any such implementation must comply with the content-wise and behavioral description of the ADT.

Examples

- **Integers:** An integer is an abstract data type having the standard mathematical meaning. In order that integers may be useful, we also need to specify operations (arithmetic operations, gcd, square root etc.) and relations (ordering, congruence etc.) on integers.
- **Real numbers:** There are mathematically rigorous ways of defining real numbers (Dedekind cuts, completion of rational numbers, etc). To avoid these mathematical details, let us plan to represent real numbers by decimal expansions (not necessarily terminating). Real numbers satisfy standard arithmetic and other operations and the usual ordering.
- **Complex numbers:** A complex number may be mathematically treated as an ordered pair of real numbers. An understanding of real numbers is then sufficient to represent complex numbers. However, the complex arithmetic is markedly different from the real arithmetic.
- **Polynomials** with real (or complex or integer or rational) coefficients with the standard arithmetic.
- **Matrices** with real (or complex or integer or rational) entries with the standard matrix arithmetic (which may include dimension, rank, nullity, etc).
- **Sets** are unordered collections of elements. We may restrict our study to sets of real (or complex) numbers and talk about union, intersection, complement and other standard operations on sets.
- A **multiset** is an unordered collection of elements (say, numbers), where each element is allowed to have multiple occurrences. For example, an aquarium is a multiset of fish types. One can add or delete fishes to or from an aquarium.
- A **book** is an ADT with attributes like name, author(s), ISBN, number of pages, subject, etc. You may think of relations like comparison of difficulty levels of two books.

Examples

- **Integers:** C provides so many integer variables and still I have to write my integers. You may have to. For most common-place applications C's built-in integer data types are sufficient. But not always. Suppose my target application is designing a cryptosystem, where one deals with very big integers, like those of bit-sizes one to several thousand bits. Our C's maximum integer length is 64 bits. That is grossly inadequate to address the cryptosystem designer's problem. ANSI standards dictate use of integers of length at most 32 bits, which are even poorer for cryptography, but

at the minimum portable across platforms. At any rate, you need your customized integer data types.

A common strategy is to break big integers into pieces and store each piece in a built-in data type. To an inexperienced user breaking with respect to the decimal representation seems easy and intuitive. But computer's world is binary. So breaking with respect to the binary representation is much more efficient in terms of space and running time. So we plan to use an array of unsigned long variables to store the bits of a big integer. Each such variable is a 32-bit word and is capable of storing 32 bits of a big integer. Therefore, if we plan to work with integers of size no larger than 10,000 bits, we require an array of size no more than 313 unsigned long variables. The zeroth location of the array holds the least significant 32 bits of a big integer, the first location the next 32 bits, and so on. Since all integers are not necessarily of size 10,000 bits, it is also necessary to store the actual word-size of a big integer. Finally, if we also plan to allow negative integers, we should also reserve a location for storing the sign information. So here is a possible implementation of the big integer data type.

```
typedef struct {
    unsigned long words[313];
    unsigned int wordSize;
    unsigned char sign;
} bigint;
```

This sounds okay, but has an efficiency problem. When you pass a bigint data to a function, the entire words array is copied element-by-element. That leads to unreasonable overheads during parameter passing. We can instead use an array of 315 unsigned long variables and use its 313-th and 314-th locations to store the size and sign information. The first 313 locations (at indexes 0 through 312) represent the magnitude of the integer as before.

```
#define SIZEIDX 313
#define SIGNIDX 314
typedef unsigned long goodbigint[315];
```

Now goodbigint is a simple array and so passing it to a function means only a pointer is passed. Quite efficient, right?

These big integers are big enough for cryptographic applications, but cannot represent integers bigger than big, for example, integers of bit-size millions to billions. Whenever we use static arrays, we have to put an upper limit on the size. If we have to deal with integers of arbitrary sizes (as long as memory permits), we have no option other than using dynamic memory and allocate the exact amount of memory needed to store a very big integer. But then since the maximum index of the dynamic array is not fixed, we have to store the size and sign information at the beginning of the array. Thus the magnitude of the very big integer is stored starting from the second array index. This leads to somewhat clumsy translation between word indices and array indices.

```
#define SIZEIDX 0
#define SIGNIDX 1
typedef unsigned long *verybigint;
```

A better strategy is to use a structure with a dynamic words pointer.

```
typedef struct {  
    unsigned long *words;  
    unsigned int size;  
    unsigned char sign;  
} goodverybigint;
```

So you have to pay a hell lot of attention, when implementation issues come. Good solutions come from experience and innovativeness.

Being able to define integers for a variety of applications is not enough. We need to do arithmetic (add, subtract, multiply etc.) on these integers. It is beyond the scope of this elementary course to go into the details of these arithmetic routines. It suffices here only to highlight the difference between abstract specifications and application-specific implementations. Both are important.

A complete example : the ordered list ADT

Let us now define a new ADT which has not been encountered earlier in your math courses. We call this ADT the ordered list. It is a list of elements, say characters, in which elements are ordered, i.e., there is a zeroth element, a first element, a second element, and so on, and in which repetitions of elements are allowed. For an ordered list L, let us plan to have the following functionality:

`L = init();`

Initialize L to an empty list.

`L = insert(L,ch,pos);`

Insert the character ch at position pos in the list L and return the modified list. Report error if pos is not a valid position in L.

`delete(L,pos);`

Delete the character at position pos in the list L. Report error if pos is not a valid position in L.

`isPresent(L,ch);`

Check if the character ch is present in the list L. If no match is found, return -1, else return the index of the leftmost match.

`getElement(L,pos);`

Return the character at position pos in the list L. Report error if pos is not a valid position in L.

`print(L);`

Print the list elements from start to end.

We will provide two complete implementations of this ADT. We assume that the element positions are indexed starting from 0.

Implementation using static arrays:

Let us restrict the number of elements in the ordered list to be ≤ 100 . One can then use an array of characters of this size. Moreover, one needs to maintain the current size of the list. Thus the list data type can be defined as:

```
#define MAXLEN 100
```

```
typedef struct {  
    int len;  
    char element[MAXLEN];  
} olist;
```

Let us now implement all the associated functions one by one.

```
olist init ()  
{  
    olist L;
```

```
    L.len = 0;  
    return L;
```

```
}
```

```
olist insert ( olist L , char ch , int pos )
```

```
{
```

```
    int i;
```

```
    if ((pos < 0) || (pos > L.len)) {  
        fprintf(stderr, "insert: Invalid index %d\n", pos);  
        return L;
```

```
    }
```

```
    if (L.len == MAXLEN) {  
        fprintf(stderr, "insert: List already full\n");  
        return L;
```

```
    }
```

```
    for (i = L.len; i > pos; --i) L.element[i] = L.element[i-1];
```

```
    L.element[pos] = ch;
```

```
    ++L.len;
```

```
    return L;
```

```
}
```

```
olist delete ( olist L , int pos )
```

```
{
```

```
    int i;
```

```
    if ((pos < 0) || (pos >= L.len)) {  
        fprintf(stderr, "delete: Invalid index %d\n", pos);  
        return L;
```

```
    }
```

```

    for (i = pos; i <= L.len - 2; ++i) L.element[i] = L.element[i+1];
    --L.len;
    return L;
}

int isPresent ( olist L , char ch )
{
    int i;

    for (i = 0; i < L.len; ++i) if (L.element[i] == ch) return i;
    return -1;
}

char getElement ( olist L , int pos )
{
    if ((pos < 0) || (pos >= L.len)) {
        fprintf(stderr, "getElement: Invalid index %d\n", pos);
        return '\0';
    }
    return L.element[pos];
}

void print ( olist L )
{
    int i;

    for (i = 0; i < L.len; ++i) printf("%c", L.element[i]);
}

```

Here is a possible main() function with these calls.

```

int main ()
{
    olist L;

    L = init();
    L = insert(L,'a',0);
    printf("Current list is : "); print(L); printf("\n");
    L = insert(L,'b',0);
    printf("Current list is : "); print(L); printf("\n");
    L = delete(L,5);
    printf("Current list is : "); print(L); printf("\n");
    L = insert(L,'c',1);
    printf("Current list is : "); print(L); printf("\n");
    L = insert(L,'b',3);
    printf("Current list is : "); print(L); printf("\n");
    L = delete(L,2);
    printf("Current list is : "); print(L); printf("\n");
}

```

**'UNIT -II NOTES****Subject:- Algorithm & Data Structures****Semester:- IV****Syllabus**

Stacks and Queue: Definition and Terminology, Concept of stack, Stack implementation, Operation on stack, Algorithms for push and pop, Implementing stack using pointers, Application of stacks, Evaluation of polish notation, multiple stack.

Queue: Queue as ADT Implementation of queue, Operation on queue, Limitations, Circular queue, Double ended queue (dequeue), Priority queue, Application of queues, multiple queues.

Q.1. What is a stack? What are the various operations associated with stack?

Ans. Stack : It is a linear data structure. **A stack is a list of** elements in which an element may be inserted or deleted only at one end, called the 'top of the stack. The last item to be added or inserted to the stack is the first to be removed or deleted. So, stacks are also called as Last in First Out (**i.e. LIFO**), type of data structure.

Two basic operations are associated with the stack. They are:

1. **PUSH :** To insert an element into a stack.
2. **POP :** To delete or remove an element from the stack.

Q.2. Explain how stacks are represented using arrays.

Ans. Stacks are represented in computers, by means of a linear linked list or a linear array.

in representation by an array S. a pointer variable TOP, is used to hold the location of the top element of the stack. A variable N is used to give the maximum number of elements that can be held by the stack.

TOP=5 N=10

Q.3. Give the representation of stack in C.

Ans. in Pascal, stack is represented using arrays. An array has a fixed number of elements whereas a stack is a dynamic object whose size is always changing as items are popped or pushed.

By declaring an array with a large range, it can be used as a stack. During operations the stack grows and shrinks in the space reserved for it. One end of the array is used as a fixed bottom of the stack and the other end is a constantly changing top. A variable TOP is used to keep a track of the current position of the top of stack.

A stack is represented in C as:

```
#define MAX 100
```

```
struct stack
```

```
(
```

```
int item [MAX];
```

```
int top: 1...MAX;
```

```
) s;
```

The stack is initialized before using.

Q.4. Explain push and write procedure for it.

Ans. Push : As element is inserted into the stack, the value of the top is incremented by 1. Before adding, it is necessary to check the value of the top, If it is maximum, then the stack is full and then it is not possible to push any element onto the stack.

The procedure is as given below:

```
push(struct stack s, int x)
```

```
{
```

```
if (s.top == MAX)
```

```
printf("Stack full")
```

```
else
```

```
{
```

```
s.top = s.top + 1;
```

```
s.item[s.top] = x;
```

```
}
```

```
}
```

Q.5. Explain pop and write the procedure for it.

Ans. Pop: The deletion of an item takes place from the top. If top = 0, then the stack is empty and any item cannot be deleted from it. After deleting the element from the stack, the value of the top is decremented by 1. So, the next

element becomes the top,

The procedure is as given:

pop(struct stack s);

mt x : integer;

if(s.top = 0)

printf("Stack is Empty"); else

{

x = s.item (s.top);

s.top = s.top - 1;

}

}

Q.6. Write short notes on implementation of multiple stack in an array.

OR Discuss in detail the terms multiple stacks and queues.

(W-031)

Ans. Multiple stack : A single stack is represented using arrays, S[1..n]. If there are 2 stacks, then also array S is used. S[1] is used as the bottom of stack 1 and S[n] as bottom of stack2 grows towards left. If size of stack1 is n1 and that of stack2 is n2, then

$n = n_1 + n_2$ for stack1 and stack2.

$n = 2 \times n_1$

1 2 3

stack 2

stack 1

As a single dimensional array has only two fix points s[1] and s[n] and a stack requires a fixed point for its bottom—most element. So to represent more than 2 stacks in memory in sequential order, the available memory is divided into segments and each segment is allocated to each stack.

If the size of each stack is known, then the segments can be divided in proportion to the size of stack. If size is not known, the segments are divided into equal sizes.

A separate array B is used to store the number of stacks. B[i] stores the number of stacks i. It points to a position one less than the position in s for the bottom—most element of the stack i. T[i] points to the top—most element of stack i.

$V[1..2] \quad [n/n] \quad 2[m] \quad nJ \quad m$

b[1] b[2] b[3] b[b+ 1]

t[1] t[2] t[3]

```

else
{
x = s.item (s.top);
s.top = s.top - 1;

```

Initial configuration for empty stacks:

The initial segment is given by:

$B[i] = T[i] = (m - 1) \cdot i$ in

Any stack i can grow from $b[i] + 1$ to $b[i + 1]$.

Similar implementation for multiple queue is also possible.

Q.7. Two stacks are to be represented in an array $N[1 \dots r]$. Write the procedure to add $(1, x)$ and delete $(1, x)$ from the 1^{th} stack. *IS-63j*

Ans. The procedure to add elements i^{th} stack is given as:

add(int I, int x)

```

{
if(t[i] = b[i + 1])
printf( "stack full\n");
else

```

```

{
t[i] = t[i] + i;
v[t[i]] =

```

I

I

The procedure to delete the topmost element of stack i is as given:

delete(int I, int x)

```

{
if(t[i] = b[i])
printf("stack empty");
else

```

```

{

```

```

x=v[t[i]];
t[i]=t[i]-1;

```

I

I

The stack full condition does not mean that all n locations of s are filled. There may be lot of unused or empty space between stacks i and $i+1$.

Q.8. Explain the terms “Overflow” and “Underflow” with respect to STACK.

Ans. Stack is a non-linear data structure in which insertion and deletion can be performed at the same end.

The stack is implemented using array, then the array has some finite numbers of location. When we insert element into stack and if the top will reach at position of last location of array means all location, of array are full, then overflow will occur. When we delete element from stack and if the stack does not have any element, means (top=0), then the underflow condition may occur.

QUEUE:

In Pascal, queue are declared as:

```
const maxq = 100;
type queue = record
  item: array [1.. maxq] of integer
  front, rear: 0..maxq
end
var q : queue
Front rear
```

Q.9. Write a short note on Queue.

Ans. A queue is a linear list of elements in which deletion can take place at only one end, called as front and insertion can take place at the other called as rear. Queues are also called as First In First Out (FIFO) type of data structure because of the order in which they leave the queue.

Queues are represented in the memory by linear array. Two pointer variables are used: FRONT contains the location of the front element of the queue and REAR contains the location of the rear element. If FRONT = REAR, then the queue is empty.

Q.10. Write a procedure to insert an element in queue.

Ans. When an element is inserted into the queue, the value of REAR is incremented by 1 i.e. **REAR := REAR + 1.**

The procedure to add an element to the queue is given by:

addq(int x)

I

```
if(q.rear = MAXQ)
  printf(“Queue full”);
else
```

I

```
q.rear=q.rear+ 1;
q.item (q.rear)
```

)

)

Q.11. Write a procedure to delete an element from the queue.

Ans. **When an element is deleted from the queue**, the value of FRONT is incremented **by 1, I.e.,**

FRONT = FRONT + 1

The procedure to delete an item from a queue is given as deleteq(int x)

I

if(q.front == q.rear)
printf("Queue empty");

else

I

x = q.item [q.front];
q.front = q.front + I;

)

)

Q.12. What are FIFO (First-in-First-out) and LIFO (Last-in-First-out) lists. Discuss atleast three differences between them.

Ans. FIFO stands for first-in-first-out. These lists are the one in which the element which is inserted first is taken out first. Queue is based on FIFO list. FIFO list are generally used for batch processing system.

LIFO stands for last-in-first-out. These lists are the ones in which the element which is inserted last is taken out first. Stack is implemented using LIFO list. LIFO list are generally used for recursion.

Q.13. Define circular queue.

Ans. A more efficient queue representation is obtained by regarding the array $Q(1 : n)$ as circular. It now becomes more Convenient to declare the array as $Q(0 : n - 1)$. When $rear = n - 1$, the next element is entered at $Q(0 : n - 1)$. When $rear = n - 1$, the next element is entered at $Q(0)$ in case that spot is free. Front will always point one position counterclockwise from the first element in the queue. Again $front = rear$ if and only if the queue is empty. Initially, we have $front = rear = 1$. Figure illustrates some of the possible configurations for a circular queue containing the four elements ii — J4with $n > 4$.

FIFO	LIFO
1) FIFO stands for first-in-first- out.	LIFO stands for last-in-first out.

2) Here the element which is inserted first is taken out first.	Here the element which is inserted last is taken out first.
3) Queue is implemented using FIFO list.	Stack is implemented using LIFO list.
4) FIFO list are used for batch processing.	LIFO list are used for recursion.

(4)

(n-4)

front = n-4, rear

Circular queue of size n, elements are stored from front to rear. J2, J3, 14

In order to add an element it will be necessary to move rear one position clockwise i.e.

if rear = n-1 then rear = 0

else rear = rear + 1

using the modulo operator which computes remainders, this is first rear = (rear + 1) mod n. Similarly, it will be necessary to move front one position clockwise each time a deletion is made. Again, using the modulo operation, this can be accomplished by front = (front + 1) mod n. An examination of the algorithms indicates that addition and deletion can now be carried out in a fixed amount of time or O(1).

Q.14. Write a procedure to implement a queue using

circular link list. **IS-031**

Ans. / Program to implement a queue

using circular linked list */

include "alloc.h"

struct node

{

int data

struct node link;

}

main ()

{

/*

(n-3)

(3

(Q) (n—I)
front '0; rear '4

Cr. —3) (n —2)'

(0) (n—I)

struct node *front, *rear;

front = rear = NULL;

addcirq(&front, &rear, 10);

addcirq(&front, &rear, 11);

addcirq(&front, &rear, 12);

addcirq(&front, &rear, 13);

addcirq(&front, &rear, 14);

addcirq(&front, &rear, 15);

clrscr();

cirq-display (front);

delcirq (&front, &rear);

delcirq (&front, &rear);

printf (“\n\n After deletion : \n”);

cirq-display (front);

I

/* adds a new element at the end of queue /

addcirq (struct node **f, struct node **r, **mt** item)

{

struct node *q;

/ create new node /

q = malloc (size of (struct node));

q —> data = item;

/ * if the queue is empty /

if(*f == NULL)

else

(*r) —> link = q;

*r =

(*r) link =

/*removes an element from front of queue / delcirq (struct node **f, struct node r)

(

struct node *q;

mt item,

/* if queue is empty /

if (f == NULL)

```
printf ("queue is empty"); else
```

```
I
```

```
if(f== *r)
```

```
(
```

```
item = (0 .data; free (f);
```

```
•f= NULL;
```

```
r=NULL
```

```
)
```

```
else
```

```
I
```

```
/ delete the node /
```

```
q*f;
```

```
item = q —p data;
```

```
f=('f) —,link;
```

```
(r) —, link =
```

```
free (q);
```

```
return (item);
```

```
)
```

```
I
```

```
/ displays whole of the queue'!
```

```
cirq-display (struct node f)
```

```
I
```

```
struct node *q = f, p = NULL;
```

```
pnntf ("\\nfront —>");
```

```
I' traverse the entire linked list /
```

```
while (q!= p)
```

```
I
```

```
printf ("%2d", q —data); q=q —link;
```

```
p=f
```

```
I
```

```
printf(" _>.....frontu);
```

```
I .
```

Q.15. Write a procedure to insert an item in circular queue.

OR Write a function which will implement circular queue using array. (W-031

Ans.

```
qinsert(int x)
```

```
(
```

```

if(((q.front=1)&&(q.rear = maxq)) || (q.front = q.rear + 1)) printf("Queue is full");
if(q.front = q.rear)
I
q.front= I;
q.rear= I;
)
else
q.rear = (q.rear + 1) mod maxq;
q.item [q.rear] = x;
I

```

Q.16. Write a procedure to delete an item from a circular queue.

Ans.

qdelete (in x)

```

I
if(q.front = NULL)
printf("Queue is Empty");
if(q.front = rear)

```

```

I
qfront = NULL;
q.rear = NULL;
)
else
q.fron = (q.front + 1) mod n;
x = q.item[q.front];

```

I

Q.17. Distinguish between STACK and QUEUE. (mm. four)

Stack

1) Stack is non-linear data structure in which insertion and deletion can be done from top i.e. from one end only.

Queue

1) Queue is non-linear data structure in which insertion and deletion can be done from rear and front.

Q.18. Give the difference between plain queue and circular queue.

Ans. The main difference between two queues is the method to check the full and empty conditions of the queue. In circular queue the same method is used to check whether the queue is full or empty i.e. front = rear.

The another difference with plain queue is, if the size of the queue is "n—i", then it

is only possible to make use of $n-1$ positions of the queue. Because if all the “ n ” positions are used, then it will not be possible to distinguish the full and empty conditions as $front = rear$.

Q.19. A double ended queue (deque) is a linear list in which additions and deletions are made at either end. Obtain a data representation, mapping a dequeue into a one dimensional array. Write the algorithm to add and delete elements at either end of the deque.

2) This is a first in last out (FILO) or last in first out (LIFO).	2) This is first in first out (FIFO).
3) For insertion we have to check limit of stack i.e. ($top == n$)	3) For insertion, we have to check limit of queue with rear i.e. ($rear == n$)
4) For deletion we have to check ($top == 0$)	4) For deletion we have to check ($front == rear$)
5) Example, dishes arrangement one above another.	5) Example: Line of vehicles waiting for green signal.

Ans. It is a linear list having 2 ends, or it is a double ended queue, in which elements can be added or removed at either end but not in the middle.

The deque can be represented using a circular array with 2 pointers LEFT and RIGHT, which point to the 2 ends of dequeue.

LEFT = 2

RIGHT = 5
8 9 10

LEFT = 9 RIGHT = 1

I 2 3 4 5 6 7 8 9 10 II

There are 2 types of deque : an input restricted deque & an output restricted deque.

An input restricted deque is a deque which allows insertion at only one end and deletion at both the ends of the list.

The algorithm to insert an element in a deque is given as:

procedure DQINSERT (DQ, RIGHT, LEFT, X)

1. If $LEFT \geq RIGHT$ then

overflow, return

2. if INSERT ON LEFT then

```

LEFT := LEFT - 1
DQ(LEFT) := X
else
RIGHT := RIGHT + 1
DQ(RIGHT) := X
endif
3. RETURN.

```

The algorithm to delete an element from deque is given as:

procedure DQDELETE (DQ, RIGHT, LEFT, X)

1. if LEFT = NULL & RIGHT = NULL then **2. if DELETE from LEFT then**

X := DQ(LEFT);

LEFT := LEFT + 1

else

X:=DQ(RIGHT)

RIGHT := RIGHT-i

endif

3. RETURN.

Q.20. Explain the priority queue.

OR What do you mean by a priority queue? Write a function to add an item in a priority queue and a function to delete an element. Priority of each and every element is stored

explicitly with every element. [S-02]

OR Explain difference between queue and priority queue in

short. [W-03]

Ans. Priority Queue: Priority queue is a queue in which each element has been assigned a priority and operations of insertion or **deletion are done as per the priority. For** insertion, an element with higher priority is processed before any other element with lower priority. Two elements with the same priority are processed according to the order in which they were added to the queue.

Priority queues are represented in memory using list and arrays.

In case of representation using arrays, a separate circular queue is used for each level of priority. Each queue has its own pair of pointers FRONT and REAR. A 2D array is used to represent the priority queue. Each row has the same priority and the row number denotes the priority level. FRONT[I] and REAR[I] contain the front and rear element of row K of queue, I is the priority number.

For deletion, the elements with higher priority is deleted first and lower next.

If two elements have same priority, then they will be deleted in the order where they were added.

Q.21. What are the applications of stack? Explain recursion through an example.

Aim. Recursion : If a function refers itself, it is said to be a recursive function.

A procedure contains either a call statement to itself or a call statement to a second procedure which again calls back to the original procedure. The procedure is then called as a recursive procedure. For a procedure or a function to be recursive, it must have the following 2 properties:

1. There must be a base criteria for which the subprogram does not call itself.
 2. Each time a subprogram calls itself, it must be closer to the base criteria.
- The subprograms with these 2 properties are said to be well defined. A recursive solution to a problem is more expensive than a non- recursive solution, both in terms of time and space but this expense is very small in comparison to the logical simplicity and self- documentation of a recursive program.

Q.22. Explain the various types of expressions.

Ans. A mathematical expression involves constants and operations.

1
2
3
4
5
6

FRONT	REAR
3	4
1	3
1	1
5	3
0	0
4	4

There are 3 types of notations depending on the arrangement of operator and operands.

1. Infix notation : The operator is placed between the 2 operands. Eg. MB, C*D, GIJ

With this notation, the knowledge of operator precedence and the use of parenthesis

is essential.

2. Prefix or polish notation : The operator is placed before its two operands.

+ AB, *CD, **IGJ**

The order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expressions. So, the need for parenthesis and operator precedence is eliminated.

3. Postfix or Reverse Polish Notation : The operator is placed after its 2 operands.

AB+,CD,GJ!

There is no need of parenthesis to determine the order of the operations in any arithmetic.

The arithmetic expression written in infix notation is evaluated by the computer in 2 steps. First, it converts the expression to postfix notation and then it evaluates the postfix expression.

Q.23. Write a short note on the evaluation of postfix expression.

OR Write an algorithm to evaluate a postfix expression.

(S-02)

Ans. Suppose P is an arithmetic expression written in postfix notation. The algorithm evaluates the expression P with the use of stack S.

It is assumed that the last character in P is #.

The algorithm is:

1. Scan P from left to right and repeat step 2 & 3 for each element of P. until '#' is encountered.

2. If an operand is encountered, put it on stack s.

3. If an operator is encountered, then

a. Remove the 2 elements of stack, where A is the top element and B is next to the top element.

b. Evaluate: B operator A.

c. Place the result of b back on stack S.

endif

end of step 1 loop

4. Set value of expression as the top element on stack s.

5. Exit.

Consider an expression P

P:3,6,4,_,*, 12,4,/,*

The algorithm works as:

Q.24. Explain how to evaluate postfix expression using operator shifting.

Ans. Consider the expression given in POSTFIX form, convert the expression into INFIX and then solve.

El 171 ü c*

lb. I., I., I., I., I., d., I., P., T.,

Scan the expression from the L.H.S. If the operator is found then place it in between two operands present just before the operator. For example, the first operator $-$, then place it in between 7 and 3.

Step 1: $7-3$

Similar process is repeated for all the tokens present in the expression.

Step2: $12/(7-3)$

°

Symbol scanned	STACK	
3	3	
6	3,	6
4	3,	6,4
-	3,	2
*	6	
12	6,	12
4	6,	12,4
r	6,	3
4'	18	
#	18	

Siep3: $12/(73)2(1+5)*+$

Step4: $12/(7_3)2*(1+5)+$

Step5: $12/(7_3)+2*(1+5)$

FinalresUlt $12/4+2*6=3+12=15$

Step 1: $3+1=4$

Step2: $4** 27, 4-, 2, *, , 5, _$

Step 3: $16, 7-4, 2, *, , 5, _$

Step4: $16, 3*2, +, 5, _$

Step 5: $16+6,5,—$

Step 6: $22-5= 17$

Q.25. Write a program to evaluate a postfix expression. Ans. The program to evaluate the postfix expression is given as, # define MAX 25

```
void push(int*, int*, int);
```

```
int pop(int *, int *);
```

```
main()
```

```
char str[MAX], *s;
```

```
int n1, n2, n3, nn;
```

```
int stack[MAX], top = -1;
```

```
clrscr();
```

```
printf("\n Enter the postfix expression to be evaluated:"); gets(str);
```

```
s=str
```

```
while(*s)
```

```
{
```

```
/*skip whitespace, if any */
```

```
if(s=="|| s'\t')
```

```
{
```

```
s++;
```

```
continue;
```