# Introduction to Data Structure and Algorithms

**Why Data Structures?**

1. Data structures study how data are stored in a computer so that operations can be implemented efficiently
2. Data structures are especially important when you have a large amount of information
3. Conceptual and concrete ways to organize data for efficient storage and manipulation.

**Methods of Interpreting bit Setting**

1. Binary Number System
    - Non Negative
    - Negative
        - Ones Complement Notation
        - Twos Complement Notation
2. Binary Coded Decimal
3. Real Number
4. Character String

**Non-Negative Binary System**

In this System each bit position represents a power of 2. The right most bit position represent $2^0$ which equals 1. The next position to the left represents $2^1$= 2 and so on. An Integer is represented as a sum of powers of 2. A string of all 0s represents the number 0. If a 1 appears in a particular bit position, the power of 2 represented by that bit position is included in the Sum. But if a 0 appears, the power of 2 is not included in the Sum. For example 10011, the sum is $2^0 + 2^1 + 2^4 = 19$

**Ones Complement Notation**

Negative binary number is represented by ones Complement Notation. In this notation we represent a negative number by changing each bit in its absolute value to the opposite bit setting. For example, since 001001100 represent 38, 11011001 is used to represent -38. The left most number is reserved for the sign of the number. A bit String Starting with a 0 represents a positive number, where a bit string starting with a 1 represents a negative number.

**Twos Complement Notation**

In Twos Complement Notation is also used to represent a negative number. In this notation 1 is added to the Ones Complement Notation of a negative number. For example, since 11011001 represents -38 in Ones Complement Notation 11011010 used represent -38 in Twos Complement Notation.

**Binary Coded Decimal**

In this System a string of bits may be used to represent integers in the *Decimal Number System* . Four bits can be used to represent a Decimal digit between 0 and 9 in the binary notation. A string of bits of arbitrary length may be divided into consecutive sets of four bits. With each set representing a decimal digit. The string then represents the number that is formed by those decimal digits in conventional decimal notation. For example, in this system the bit string 00110101 is separated into two strings of four bits each: 0011 and 0101. The first of these represents the decimal digit 3 and the second represents the decimal 5, so that the entire string represents the integer 35.

In the binary coded decimal system we use 4 bits, so this four bits represent sixteen possible states. But only 10 of those sixteen possibilities are used. That means, whose binary values are 10 or larger, are invalid in *Binary Coded Decimal* System.

**Binary Coded Decimal**

In this System a string of bits may be used to represent integers in the *Decimal Number System* . Four bits can be used to represent a Decimal digit between 0 and 9 in the binary notation. A string of bits of arbitrary length may be divided into consecutive sets of four bits. With each set representing a decimal digit. The string then represents the number that is formed by those decimal digits in conventional decimal notation. For example, in this system the bit string 00110101 is separated into two strings of four bits each: 0011 and 0101. The first of these represents the decimal digit 3 and the second represents the decimal 5, so that the entire string represents the integer 35.

In the binary coded decimal system we use 4 bits, so this four bits represent sixteen possible states. But only 10 of those sixteen possibilities are used. That means, whose binary values are 10 or larger, are invalid in *Binary Coded Decimal* System.

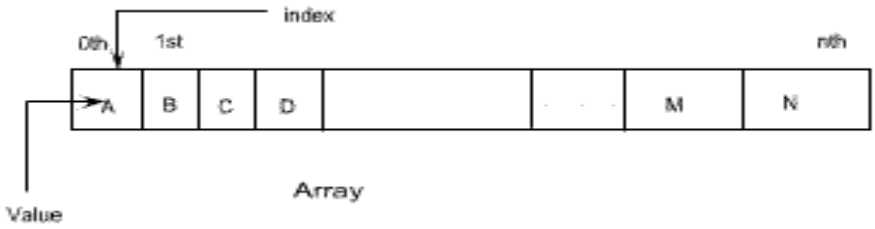
The assignment of bit string to character may be entirely arbitrary, but it must be adhered to consistently. It may be that some convenient rule is used in assigning bit string to character. The number of bits varies computer wise used to represent a character. Some computers are use 7-bit (therefore allow up to 128 possible characters), some computers are use 8-bits (up to 256 character), and some use 10-bits (up to 1024 possible characters). The number of bits necessary to represent a character in a particular computer is called the **byte size** and a group of bits that number is called a **byte.**

**Array**

In computer programming, a group of homogeneous elements of a specific data type is known as an array, one of the simplest data structures. Arrays hold a series of data elements, usually of the

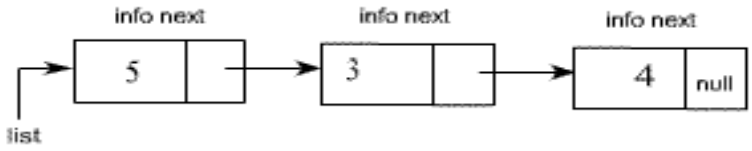
same size and data type. Individual elements are accessed by their position in the array. The position is given by an index, which is also called a subscript. The index usually uses a consecutive range of integers, (as opposed to an associative array) but the index can have any ordinal set of values.

Some arrays are *multi-dimensional*, meaning they are indexed by a fixed number of integers, for example by a tuple of four integers. Generally, one- and two-dimensional arrays are the most common. Most programming languages have a built-in *array* data type.




**Link List**

In computer science, a **linked list** is one of the fundamental data structures used in computer programming. It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next and/or previous nodes. A linked list is a self-referential data type because it contains a link to another data of the same type. Linked lists permit insertion and removal of nodes at any point in the list in constant time, but do not allow random access.




**Types of Link List**

1. Linearly-linked List
    - Singly-linked list
    - Doubly-linked list
2. Circularly-linked list
    - Singly-circularly-linked list
    - Doubly-circularly-linked list
3. Sentinel nodes

**Stack**

A stack is a linear Structure in which item may be added or removed only at one end. There are certain frequent situations in computer science when one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the end of the list, not in the middle. Two of the Data Structures that are useful in such situations are *Stacks* and *queues.* A stack is a list of elements in which an elements may be inserted or deleted only at one end, called the *Top.* This means, in particular, the elements are removed from a stack in the reverse order of that which they are inserted in to the stack. The stack also called *"last-in first -out* (LIFO)" list.

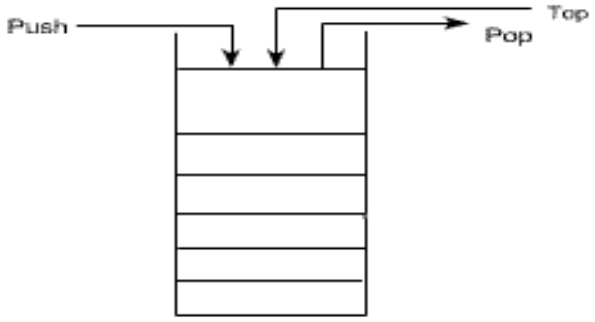



Fig: Stack

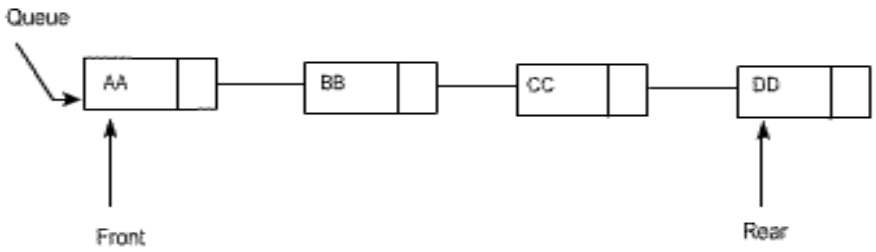
Special terminology is used for two basic operation associated with stack :

1. "Push" is the term used to insert an element into a stack.
2. "Pop" is the term used to delete an element from a stack.

## Queue

A queue is a linear list of elements in which deletions can take place only at one end, called the "front*"* and insertion can take place only at the other end, called "rear ". The term "front " and " *rear* " are used in describing a linear list only when it is implemented as a queue.

Queues are also called "first-*in first-out* "(FIFO) list. Since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enter in a queue is the order in which they leave. The real life example: the people waiting in a line at Railway ticket Counter form a queue, where the first person in a line is the first person to be waited on. An important example of a queue in computer science occurs in timesharing system, in which programs with the same priority form a queue while waiting to be executed.



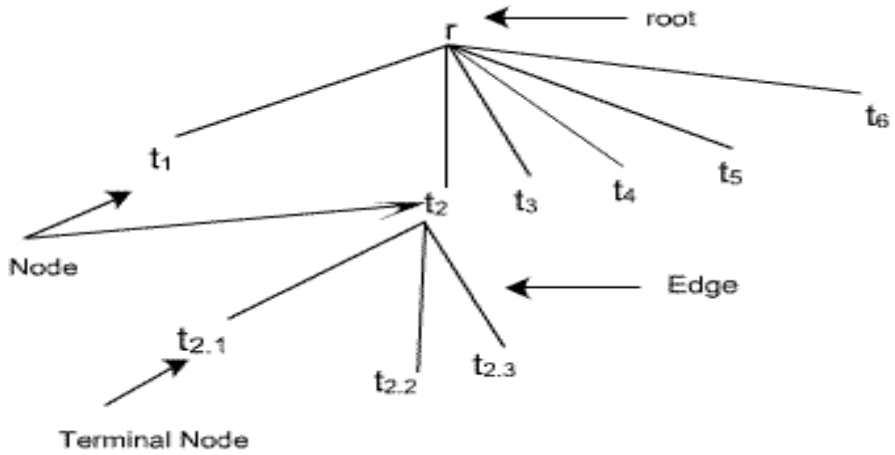

**Tree**

Data frequently contain a hierarchical relationship between various elements. This non-linear Data structure which reflects this relationship is called a rooted tree graph or, tree.

This structure is mainly used to represent data containing a hierarchical relationship between elements, e.g. record, family tree and table of contents.

A tree consist of a distinguished node *r* , called the ***root*** and zero or more (sub) tree t1 , t2 , ... tn , each of whose roots are connected by a directed edge to *r* .

In the tree of figure, the root is A, Node t 2 has *r* as a parent and t 2.1 , t 2.2 and t 2.3 as children. Each node may have arbitrary number of children, possibly zero. Nodes with no children are known as leaves.

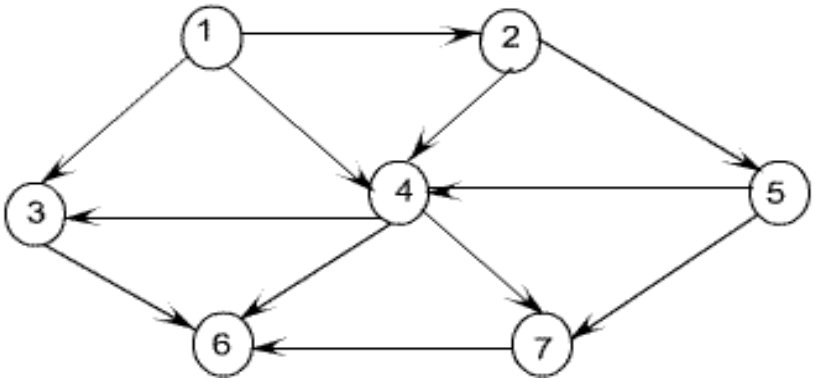



# Graph

A *graph* consists of a set of *nodes* (or *Vertices* ) and a set of *arc* (or *edge* ). Each arc in a graph is specified by a pair of nodes. A node n is *incident* to an arc x if n is one of the two nodes in the ordered pair of nodes that constitute x. The *degree* of a node is the number of arcs incident to it.

The *indegree* of a node n is the number of arcs that have n as the head, and the outdegree of n is the number of arcs that have n as the tail.

The graph is the nonlinear data structure. The graph shown in the figure represents 7 vertices and 12 edges. The Vertices are { 1, 2, 3, 4, 5, 6, 7} and the arcs are {(1,2), (1,3), (1,4), (2,4), (2,5), (3,4), (3,6), (4,5), (4,6), (4,7), (5,7), (6,7) }. Node (4) in figure has indegree 3, outdegree 3 and degree 6.




**Abstract Data Type**

1. Abstract Data Types (ADT's) are a model used to understand the design of a data structure
2. *'Abstract* ' implies that we give an implementation-independent view of the data structure
3. ADTs specify the type of data stored and the operations that support the data
4. Viewing a data structure as an ADT allows a programmer to focus on an idealized model of the data and its operations

## Problem:

Consider the stack NAME in fig 1.01, which is stored alphabetically.

Suppose Nirmal is to be inserted in to the stack. How many name must be moved to the new location?

Suppose Sourav is to be deleted from the stack. How many names must be removed to the new location?

fig 1.01

The following is a tree structure given by means of level numbers as discussed below:
01 Employee 02 Name 02 Emp. Code 02 Designation 03 Project Leader 03 Project Manager 02 Address

Draw the corresponding tree diagram.

## Introduction to Algorithms

The word algorism**,** came from the 9th century Persian mathematician "Abu Abdullah Muhammad bin Musa
**al-Khwarizmi**" , which means the method of doing arithmetic using Indo-Arabic decimal system. It is also the
root of the word "algorithm".

An algorithm is a well defined computational method that takes some value(s) as input and produces some value(s) as output. In other words, an algorithm is a sequence of computational steps that transforms input(s) into output(s).An algorithm is correct if for every input, it halts with correct output. A correct algorithm solves the given problem, where as an incorrect algorithm might not halt at all on some input instance, or it might halt with other than designed answer.

Each algorithm must have

- Specification: Description of the computational procedure.
- Pre-conditions: The condition(s) on input.
- Body of the Algorithm: A sequence of clear and unambiguous instructions.
- Post-conditions: The condition(s) on output.

- Algorithms are written in pseudo code that resembles programming languages like C and Pascal.
- Consider a simple algorithm for finding the factorial of ***n***.

Algorithm Factorial (n)

Step 1: FACT = 1
Step 2: for i = 1 to n do
Step 3: FACT = FACT * i
Step 4: print FACT

**Specification**: Computes *n*!.
**Pre-condition**: *n* >= 0
**Post-condition**: FACT = *n*!

- For better understanding conditions can also be defined after any statement, to specify values in particular variables.
- Pre-condition and post-condition can also be defined for loop, to define conditions satisfied before starting and after completion of loop respectively.
- What is remain true before execution of the $i^{th}$ iteration of a loop is called "loop invariant".
- These conditions are useful during debugging proces of algorithms implementation.
- Moreover, these conditions can also be used for giving correctness proof.

## Time Complexity

To measure the time complexity in absolute time unit has the following problems

The time required for an algorithm depends on number of instructions executed, which is a complex polynomial.

The execution time of an instruction depends on computer's power. Since, different computers take different amount of time for the same instruction.

Different types of instructions take different amount of time on same computer.

Complexity analysis technique abstracts away these machine dependent factors . In this approach, we assume all instruction takes constant amount of time for execution.

Asymptotic bounds as polynomials are used as a measure of the estimation of the number of instructions to be executed by the algorithm . Three main types of asymptotic order notations are used in practice:

**1.**

$\Theta$ - ***notation*** : For a given function g(n), $\Theta(g(n))$ is defined as

$$\theta(g(n)) = \left\{ \begin{array}{l} f(n): \text{ there exist } c_1 > 0, c_2 > 0 \text{ and } n_0 \in N \\ \qquad \text{such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n), \text{ for all } n \ge n_0 \end{array} \right\}$$

In other words, a function $f(n)$ is said to belong to $\theta(g(n))$, if there exists positive constants and such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for sufficiently large value of n. For example, This is because we can find constants $c_1 = \frac{1}{2}$ and $c_2 = 1$ such that, for all $n \ge 2$.

**2.**

$$O(g(n)) = \left\{ \begin{array}{l} f(n): \textit{there exists } c > 0, \textit{ and } n_0 \in N \\ \qquad \textit{such that } 0 \le f(n) \le cg(n), \textit{ for all } n \ge n_0 \end{array} \right\}$$

For example, as for all

**3.** $\Omega$- ***notation:*** This notation provides asymptotic lower bound. For a given $g(n)$, $\Omega(g(n))$ is defined as

$$\Omega(g(n)) = \left\{ \begin{array}{l} f(n): \textit{there exists } c > 0, \textit{ and } n_0 \in N \\ \qquad \textit{such that } 0 \le cg(n) \le f(n), \textit{ for all } n \ge n_0 \end{array} \right\}$$

For example, $n^3 \in \Omega(n^2)$ because $n^3 \ge n^2$, for all $n \ge 0$

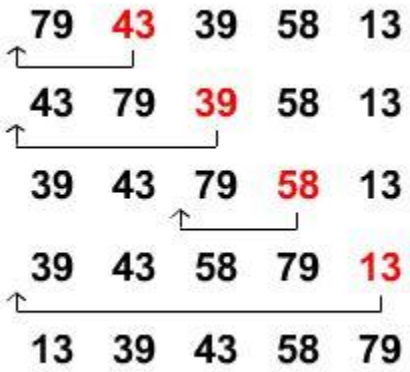
# Sorting

**Problem Definition**: Sort given *n* numbers by non-descending order.
There are many sorting algorithm. Insertion sort is a simple algorithm.
Insertion Sort: We can assume up to first number is sorted. Then sort up to two numbers. Next, sort up to three numbers. This process continues till we sort all *n* numbers.
Consider the following example of five integer:

79 43 39 58 13 : Up to first number, 79, is sorted.
43 79 39 58 13 : Sorted up to two numbers.
39 43 79 58 13 : Sorted up to three numbers.

39 43 58 79 13: Sorted up to four numbers.
13 39 43 58 79: Sorted all numbers.

That is, if first (i-1) numbers are sorted then insert $i^{th}$ number into its correct position. This can be done by shifting numbers right one number at a time till a position for $i^{th}$ number is found. That is, shift number at $(i-1)^{th}$ position to $i^{th}$ position, number in $(i-2)^{th}$ position to $(i-1)^{th}$ position, and so on, till we find a correct position for the number in $i^{th}$ position. This method is depicted in the figure on right side.




The algorithmic description of insertion sort is given below.

***Algorithm Insertion Sort (a[n])***

Step 1: for i = 2 to n do
Step 2: current_num = a[i]
Step 3: j = i
Step 4: while (( j >1) and (a[j-1] > current_num)) do
Step 5: a[j] = a[j-1]
Step 6: j = j-1
Step 7: a[j] = current_num

Execution time of an algorithm depends on numbers of instruction executed.Consider the following algorithm fragment:
for i = 1 to n do
sum = sum + i ;
The for loop executed $n$+1 times for i values 1,2,..... $n$, $n$+1. Each instruction in the body of the loop is executed once for each value of $i$ = 1,2,....., $n$. So number of steps executed is 2$n$+1.
Consider another algorithm fragment:
for i = 1 to n do for j = 1 to n do k = k +1

From previous example, number of instruction executed in the inner loop is $2n+1$, which the body of outer loop is.

Total number of instruction executed is

$= n+1+n(2n+1) = 2n^2+2n+1$

```
/** Full Implementation of Insertion Sort in C **/

#include<stdio.h>
#include<conio.h>
#DEFINE MAX_SIZE 50

void main(){
int i , j , n , cn  ;
int x[MAX_SIZE];

/** Some Look Good Code **/
Printf("******* Implementing Insertion ******");

/** getting the Number of Input array Elements **/

Printf("Enter the Number of Element You want to sort :");
scanf("%d",&n);

/*** Code to capture the Input array Elements ***/
for( i = 0 ; i < n ; i++ )
  {
printf("Enter %d element :" , (i+1));
scanf("%d", &x[i]);
  }

/*** Code to Display Input Array ***/

printf("Input Array Elements for Insertion Sort ::");

for( i = 0 ; i < n ; i++ )
  {
    printf("%d \t" , x[i]);
  }

/*** Code For Insertion Sort ***/

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

```
{
   cn = x[i] ;
   j = I ;
      while  (j > 0  && ( x[j-1] > cn )
          {
             x[j]=x[j-1];
             j=j-1;
             x[j]=cn;

          }
    }
}

/*** Code to Display the Output Array ***/

for( i = 0 ; i < n ; i++ )
 {
   printf("%d \t" , x[i]);
 }

getch();
}
```

## Analysis of Insertion Sort

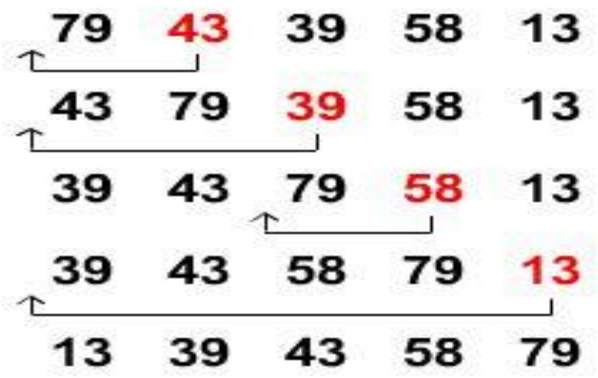
- Step 1: The for loop executed *n* times, for *i* values from 2, 3, .....*n*+1
- Step 2, 3, and 7: Each executed once for each iteration of the for loop. That is, each *n*-1 times.
- Step 4: The while loop is executed at most *i* times, for *i* = 2, 3, ....., *n*. That is, $n \times (n+1)/2 - 1$.
- step 5 and 6: Each instruction executed *i*-1 times, for each *i*. Hence, $(n-1) \times n / 2$ for each instruction.
- Total number of instruction executed in the worst case is $n^2 + 4n - 4$. Hence the time complexity of the algorithm is in $\theta(n^2)$.

  Similarly, we can analyze space required for the algorithm also. The Insertion sort algorithm has to store all n inputs and using three more variables. So, the space complexity of the algorithm is in $\theta(n)$.

Another way of finding number of instruction executed is by recursive equation. Let *T*(*n*) be the time required to sort n numbers. *T*(*n*) can be expressed as a sum of *T*(*n*-1) and the time required to insert $n^{th}$

element in the sorted array of n-1 element.

The time required to insert an element in sorted array of *n*-1 elements takes cn steps, where c is a positive constant. This is because to insert $n^{th}$ element, in worst case, we have to shift all *n*-1 elements one after other. See the following figure.




- Hence, the recurrence relation for Insertion sort is

$T(n) = T(n-1) + cn$, if $n > 1$

$= 1$ if $n = 1$

- There are three main approach to solve recurrence relations. They are substitution, iterative and master theorem methods. In this notes we discuss iterative approach only. For other methods see Algorithms by Cormen et al.

Iterative Method:

$T(n) = T(n-1) + cn$
$= T(n-2) + c(n-1) + cn$
$= T(n-3) + c(n-2) + c(n-1) + cn$
.
.
$= T(1) + 2c + 3c + \ldots\ldots\ldots\ldots + c(n-2) + c(n-1) + cn$
$= 1 + 2c + 3c + \ldots\ldots\ldots\ldots + c(n-2) + c(n-1) + cn$
$< c( 1 + 2 + 3 + \ldots\ldots\ldots\ldots + n-2 + n-1 + n)$
$= c\ (n\ (n+1)/2)$
$= c\ n^2/2 + c\ n/2$
$= O(n^2)$

- Consider another recurrence:

T(n)

= O(1) if n = 1

= 2T(n/2) + O(n) if n > 1

- In the above recurrence relation O(1) means a constant. So we can replace with some constant $c_1$.
- Similarly,O(n) means a function of order n. So we can replace with $C_2$n.Hence, the recurrence can be rewritten as

<= $C_1$ if n=1 T(n) <= 2T(n/2) + $C_2$n if n > 1

- Solution by Iterative method:

<= 2T(n/2) + $C_2$n T(n)
<= 2( 2T(n/2) + $C_2$n/2) + $C_2$n
<= $2^2$T( $n/2^2$) + $C_2$n + $C_2$n
<= $2^2$(2 T( $n/2^3$) + $C_2$ $n/2^2$) + $C_2$n + $C_2$n
<= $2^3$T( $n/2^3$) + $C_2$n + $C_2$n + $C_2$n .
.
.
<= $2^i$ T( $n/2^i$) + $C_2$ n i

Assume $n = 2^i$ for some value of i.That is, i = log n
T(n )= nT(1) + C $_2$n*log(n)
= C $_1$n + C $_2$n*log(n)
= O(n*log(n))

If $n \neq 2^i$ for some i, consider
< 2nT(1) + $c_2$n*log(n)
T(n) = 2$c_1$n + $c_2$n*log(n) = O(n*log(n))

**Sink Sort /Bubble Sort**

- Main idea in this method is to compare two adjacent elements and put them in proper order if they are not.
- Do this by scanning the element from first to last.
- After first scan, largest element is placed at last position. This is like a largest object sinking to bottom first.
- After second, scan second largest is placed at second last position.

- After n passes all elements are placed in their correct positions, hence sorted.

| Input | Pass 1 | Pass 2 | Pass 3 | Pass 4 | Pass 5 | Pass 6 | Pass 7 | Pass 8 |
|---|---|---|---|---|---|---|---|---|
| 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 32 | 18 | 18 | 18 | 18 | 18 | 14 | 14 | 14 |
| 18 | 24 | 24 | 19 | 19 | 14 | 18 | 18 | 18 |
| 24 | 30 | 19 | 24 | 14 | 19 | 19 | 19 | 19 |
| 30 | 19 | 28 | 14 | 24 | 24 | 24 | 24 | 24 |
| 19 | 28 | 14 | 28 | 28 | 28 | 28 | 28 | 28 |
| 28 | 14 | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
| 14 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 |

- The algorithmic description of the Sink sort is given below:

Algorithm Sink-Sort (a[n])

Step 1 : for *i* = 0 to n-2 do
Step 2 : for *j* = 0 to n-i do
Step 3 : if (a[*j*] > a[*j*+1]) then
Step 4 : swap(a[*j*],a[*j*+1]);

- The algorithm can be modified, such that, after each pass next smallest element reach to the top position.This is like a bubble coming to top. Hence called **Bubble-Sort**.

```
/** Full Implementation of Bubble Sort in C **/

#include<stdio.h>
#include<conio.h>
#DEFINE MAX_SIZE 50

void main(){
int i , j , n , temp;
int x[MAX_SIZE];

/** Some Look Good Code **/
Printf("******* Implementing Bubble Sort *******");

/** getting the Number of Input array Elements **/

Printf("Enter the Number of Element You want to sort :");
```

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

        /*** Code to capture the Input array Elements ***/
        for( i = 0 ; i < n ; i++ )
          {
        printf("Enter %d element :" , (i+1));
        scanf("%d", &x[i]);
          }

        /*** Code to Display Input Array ***/

        printf("Input Array Elements for Bubble Sort ::");

        for( i = 0 ; i < n ; i++ )
          {
            printf("%d \t" , x[i]);
          }

        /*** Code For Bubble Sort ***/

        for ( i = 0 ; i < n-1 ; i++)
        {
           for ( j = 0 ; j < (n-1-i) ;  j++)
            {
                  if(x[j]>x[j+1])
                   {
                      temp=x[j];
                      x[j]=x[j+1];
                      x[j+1]=temp;

                    }
             }
        }

        /*** Code to Display the Output Array ***/

        for( i = 0 ; i < n ; i++ )
          {
            printf("%d \t" , x[i]);
          }

        getch();
}
```

## Selection Sort:

- As we have discussed earlier, purpose of sorting is to arrange a given set of records in order by specified key. In the case of student records, key can be roll number. In the case of employees records, key can be employ identification number. That is, sorting a set of records based on specific key.
- Insertion and Sink sorts compare keys of two records and swap them if the keys are not in order. This is not just swapping keys, we have to swap whole records associated with those keys.
- The time required to swap records is proportional to its size, that is number of fields.
- In these methods, same record may be swapped many times, before reaching its final position in sorted order.
- This method, selection sort, minimized the number of swaps. Hence efficient if the size of the records are large.
- Look at keys of all records to find a record with smallest key and place the record in the first place.
- Next, find the smallest key record among remaining records and swap with the record at second position.
- Continue this procedure till all records are placed correctly.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 12 | 32 | 18 | 24 | 30 | 19 | 28 | 14 |
| After Pass 1 | 12 | 32 | 18 | 24 | 30 | 19 | 28 | 14 |
| Pass 2 | 12 | 14 | 18 | 24 | 30 | 19 | 28 | 32 |
| Pass 3 | 12 | 14 | 18 | 24 | 30 | 19 | 28 | 32 |
| Pass 4 | 12 | 14 | 18 | 19 | 30 | 24 | 28 | 32 |
| Pass 5 | 12 | 14 | 18 | 19 | 24 | 30 | 28 | 32 |
| Pass 6 | 12 | 14 | 18 | 19 | 24 | 28 | 30 | 32 |
| Pass 7 | 12 | 14 | 18 | 19 | 24 | 28 | 30 | 32 |

Pseudo code of the algorithm is given below.

***Algorithm Selection Sort (a[n])***
Step 1: for *i* = 0 to n-2 do
Step 2: lowest-key = *i*
Step 3: for *j* = *i*+1 to n-1 do {
Step 4: if (a[*i*].key > a[*j*].key) then
Step 5: lowest_key = *j*
Step 6: swap(a[*i*], a[lowest_key]); }

```
/** Full Implementation of Selection Sort in C **/

#include<stdio.h>
#include<conio.h>
#DEFINE MAX_SIZE 50

void main(){
int i , j , n , temp;
int x[MAX_SIZE];

/** Some Look Good Code **/
Printf("******* Implementing Selection Sort *******");

/** getting the Number of Input array Elements **/

Printf("Enter the Number of Element You want to sort :");
scanf("%d",&n);

/*** Code to capture the Input array Elements ***/
for( i = 0 ; i < n ; i++ )
  {
printf("Enter %d element :" , (i+1));
scanf("%d", &x[i]);
  }

/*** Code to Display Input Array ***/

printf("Input Array Elements for Selection Sort ::");

for( i = 0 ; i < n ; i++ )
  {
    printf("%d \t" , x[i]);
  }

/*** Code For Selection Sort ***/

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

          if(x[i]>x[j+1])
           {
              temp=x[i];
```

```
                x[i]=x[j+1];
                x[j+1]=temp;

              }
        }
    }

    /*** Code to Display the Output Array ***/

    for( i = 0 ; i < n ; i++ )
     {
       printf("%d \t" , x[i]);
     }

    getch();
}
```

## Merge Sort:

- The time complexity of the sorting algorithms discussed till now are in O($n^2$). That is, the numbers of comparisons performed by these algorithms are bound above by c $n^2$, for some constant c > 1.
- Can we have better sorting algorithms? Yes, merge sort method sorts given a set of number in O(n log n) time.
- Before discussing merge sort, we need to understand what is the meaning of merging?

**Merge sort Method:**

**Definition:** Given two sorted arrays, a[p] and b[q], create one sorted array of size p + q of the elements of a[p] and a[q] .

- Assume that we have to keep p+q sorted numbers in an array c[p+q].
- One way of doing this is by copying elements of a[p] and b[q] into c[p+q] and sort c[p+q]which has time complexity of at least O(n log n).
- Another efficient method is by merging which is of time complexity of O(n) .
- Method is simple, take one number from each array a and b, place the smallest element in the array c. Take the next elements and repeat the procedure till all p+q element are placed in the array c.

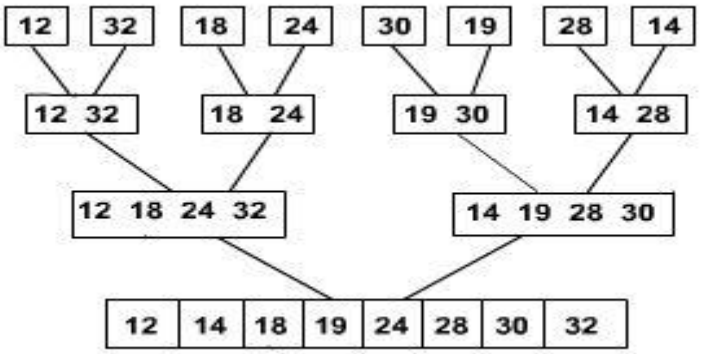
Pseudo code of the merging is given below.

Algorithm Merge (a[p], b[q])

Step 1: i = 0
Step 2: j = 0
Step 3: k = 0
Step 4: while (i < p) && (j < q) do {
Step 5: if (a[i] < b[j]) {
Step 6: c[k] = a[i] ;
Step 7: i = i + 1 ; }
Step 8: else {
Step 9: c[k] = b[j]
Step 10: j = j + 1 }
Step 11: k = k + 1 }
Step 12: if (i == p) then
Step 13: while ( j < q) do {
Step 14: c[k] = b[j] ;
Step 15: k = k + 1 ;
Step 16: j = j + 1 ; }
Step 17: else
Step 18: while(i < p) do
Step 19: { c[k] = a[i] ;
Step 20: k = k + 1 ;
Step 21: i = i + 1 ; }

We can assume each element in a given array as a sorted sub array.Take adjacent arrays and merge to obtain a sorted array of two elements. Next step, take adjacent sorted arrays, of size two, in pair and merge them to get a sorted array of four elements. Repeat the step until whole array is sorted.

Following figure illustrates the procedure.



## Quick Sort

- Another very interesting sorting algorithm is Quick sort. Its worst time complexity is O( $n^2$ ), but it is a faster then many optimal algorithms for many input instances.
- It is an another divide and conquer algorithm.
- Main idea of the algorithm is to partition the given elements into two sets such that the smaller numbers into one set and larger number into another. This partition is done with respective an element called pivot. Repeat the process for both the sets until each partition contains only one element.

- Assume that procedure Merge1(a, i, j, k) takes two sorted arrays a[i..j] and a[j+1..k] as an input and puts their merged output in the array a[i..k]. That is, in the same locations. Pseudo code of the merge sort is given below.

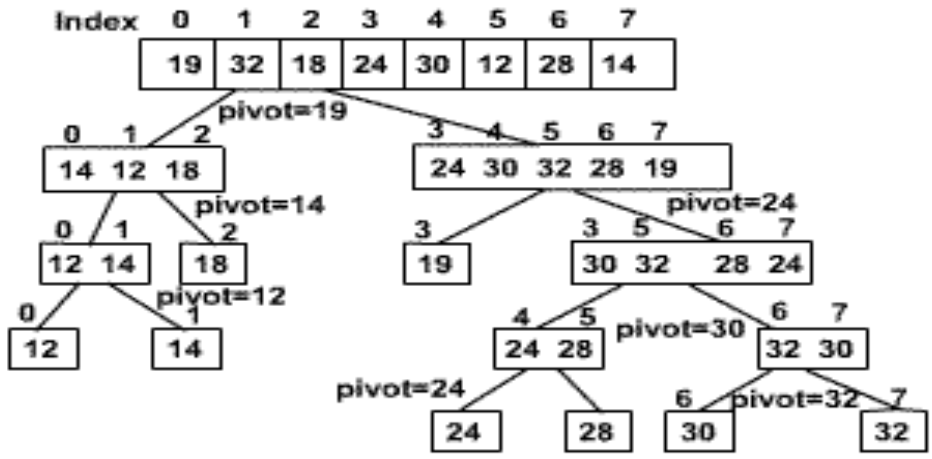
Algorithm Merge-Sort (a, p, q)

Step 1: if (p < q) {
Step 2: mid = (p+q)/2 ;
Step 3: Merge-Sort(a, p, mid) ;
Step 4: Merge-Sort(a, mid+1, q) ;
Step 5:

Merge (a,p,mid, q) ;

}

- The above procedure sorts the elements in the sub array a[p..q].
- To sort given any a[n] invoke the algorithm with parameters (a, 0, n -1).



Pseudo code of the partition is given below.

***Algorithm partition (a, p, q)***
Step 1: pivot = a[p];
Step 2: i = p-1 ; j=q+1;
Step 3: while (i<j)
Step 5: repeat
Step 6: i = i + 1
Step 7: until a[i] >= pivot;
Step 8: repeat
Step 9: j = j - 1
Step 10: until a[j] <= pivot;
Step 11: if (i < j) then
Step 12: swap (a[i], a[j]);
Step 13: return j

Algorithm quick-sort (a, p, q)

Step 1: if (p < q) {
Step 2: mid = partition (a,p,q);
Step 3: quick-sort (a, p, mid);
Step 4: quick-sort (a, mid+1, q); }

Radix Sort:

- All the algorithms discussed till now are comparison based algorithms. That is, comparison is the key for sorting.
- Assume that the input number are three decimal digits. First we sort the numbers using least significant digit. Then by next significant digit and so on.
- See following example for illustration.

| Input | Sorted by LS digit | Sorted by Middle digit | Sorted by Most Sig digit |
|---|---|---|---|
| 327 | 470 | 418 | 146 |
| 476 | 382 | 327 | 173 |
| 285 | 173 | 146 | 259 |
| 418 | 285 | 259 | 285 |
| 568 | 476 | 568 | 327 |
| 382 | 146 | 470 | 382 |
| 146 | 327 | 173 | 418 |
| 259 | 418 | 476 | 470 |
| 173 | 568 | 382 | 476 |
| 470 | 259 | 285 | 568 |

- Next question is how to sort with respective to a least significant digit or any significant digit?
- We can use any sorting algorithm we have discussed.
- Another way by maintaining a BIN for each digit from 0 to 9. If the digit is X put the number in BIN X. Concatenate the BINs from 0 to 9 to get a sorted list of numbers of that significant digit.

Algorithm radix-sort (a)

Step 1: for i = 0 to 9 do
Step 2: empty-bin(i);
Step 3: for position = least significant digit to most significant digit
Step 4: for i = 1 to n do
Step 5: x = digit in the position of a[i];
Step 6: put a[i] in BIN x;
Step 7: i =1;
Step 8: for j = 0 to 9 do
Step 9: while (BIN(j) <> empty) do
Step 10: a[i] = get-element(BIN(j));
Step 11: i = i + 1;

- Procedure get-element(bin) get the next element from the bin.
- we can use Queues for implementing Bins.

**Linear Data Structures**

**Definition**

A **data structure** is said to be ***linear*** if its elements form a sequence or a **linear** list.

**Examples:**

- Array
- Linked List
- Stacks
- Queues

**Operations on linear Data Structures**

- *Traversal* : Visit every part of the **data structure**
- *Search* : Traversal through the data structure for a given element
- *Insertion* : Adding new elements to the **data structure**
- *Deletion* : Removing an element from the **data structure**.
- *Sorting* : Rearranging the elements in some type of order(e.g Increasing or Decreasing)
- *Merging* : Combining two similar **data** structures into one

**STACK Introduction:**

1. Stack is basically a data object
2. The operational semantic (meaning) of stack is LIFO i.e. last in first out

   **Definition :** It is an ordered list of elements n , such that n>0 in which all insertions and deletions are made at one end called the top.

   **Primary operations defined on a stack:**
   1. PUSH : add an element at the top of the list.
   2. POP : remove the at the top of the list.
   3. Also "IsEmpty()" and IsFull" function, which tests whether a stack is empty or full respectively.

**Example :**

1. **Practical daily life :** a pile of heavy books kept in a vertical box,dishes kept one on top of another
2. **In computer world:** In processing of subroutine calls and returns ; there is an explicit use of stack of return addresses.
   Also in evaluation of arithmetic expressions , stack is used.

   Large number of stacks can be expressed using a single one dimensional stack only. Such an array is called a multiple stack array.

   Push and Pop:

**Algorithms**

***Push*** **(item, array, n, top)**

```
{

 if ( top<= 0)
Then print " stack is empty".

else

{
top = top + 1;
array[top] = item ;

 }
```

***Pop*** **(item, array, top)**

```
{

  if ( n> = top)
Then print "Stack is full" ;

  else {

item = array[top];
top = top – 1;
        }

  }
```

```
/*** Impementation of Stack in C ***/
#include<stdio.h>
#define MAX_SIZE 5
int stack[MAX_SIZE];
int top;
void push(int);
int pop();
void show();

void main()
{
        int item,choice;
        top=0;
        do
        {     printf("**** STACK IN C ****");
              Printf("\n MENU \n");
              printf("1. Push \n 2.Pop \n 3. Show \n 4. Exit\n");
              printf("Enter Choice: ");
              scanf("%d",&choice);

        switch(choice)
         {
           case 1:
                        printf("\n Enter element to insert");
                        scanf("%d",item);
                        push(item);
                        break;
```

```
            case 2:
                        printf("\nThe removed element is:%d",pop());
                       break;

            case 3:
                        if(top==0)
                        {
                        printf("\nThe stack is empty, I can't show you any ements");
                       }
                        else
                       {
                       show();
                      }
                        break;
          case 4:
                        exit(1);
         default:
                        printf("\nDo you understand english and numbers??");
                        }
   }
while (1);

}


void push(int item)
{
           if(top==n-1)
         {
              Printf("Stack is Full");
         }
        else
         {
          Stack[top]=item;
          top++;
         }
}

int pop()
{
        If(top==0)
         {
           Printf("Stack is Empty !");
         }
```

```
        else
         {
           top--;
           item=stack[top];
           return item;
         }
}
void show()
{
          int x=tos;
          printf("\nThe Stack elements are.....\n");
          while(x!=0)
          printf("\t%s\n",stack[--x]);
}
```

**QUEUE - A List of Elements**:

**Introduction:**

1. It is basically a data object
2. The operational semantic of queue is FIFO i.e. first in first out

**Definition:**

It is an ordered list of elements n , such that n>0 in which all deletions are made at one end called the front end and all insertions at the other end called the rear end .

**Primary operations defined on a Queue:**

1. EnQueue: This is used to add elements into the queue at the back end.
2. DeQueue: This is used to delete elements from a queue from the front end.
3. Also "IsEmpty()" and "IsFull()" can be defined to test whether the queue is Empty or full.

**Example :**

1. **PRACTICAL EXAMPLE** : A line at a ticket counter for buying tickets operates on above rules

   2. **IN COMPUTER WORLD**: In a batch processing system, jobs are queued up for processing.

## Circular Queue

In a queue if the array elements can be accessed in a circular fashion the queue is a circular queue.

**Primary operations defined for a circular queue are:**

1.**add_circular -** It is used for addition of elements to the circular queue.
**2. Delete_circular-** It is used for deletion of elements from the queue.

We will see that in a circular queue , unlike static linear array implementation of the queue ; the memory is utilized more efficient in case of circular queue's.

The shortcoming of static linear that once rear points to n which is the max size of our array we cannot insert any more elements even if there is space in the queue is removed efficiently using a circular queue.

As in case of linear queue , we'll see that condition for zero elements still remains the same i.e.. rear=front

**PRIORITY QUEUE:**

Often the items added to a queue have a priority associated with them: this priority determines the order in which they exit the queue - highest priority items are removed first.

Queues are dynamic collections which have some concept of order. This can be either based on order of entry into the queue - giving us First-In-First-Out (FIFO) or Last-In-First-Out (LIFO) queues. Both of these can be built with linked lists: the simplest "add-to-head" implementation of a linked list gives LIFO behavior. A minor modification - adding a tail pointer and adjusting the addition method implementation - will produce a FIFO queue.

## Performance

A straightforward analysis shows that for both these cases, the time needed to add or delete an item is constant and *independent of the number of items in the queue* . Thus we class both addition and deletion as an ***O* (1)** operation. For any given real machine + operating system + language combination, addition may take ***c 1*** seconds and deletion ***c 2*** seconds, but we aren't interested in the value of the constant, it will vary from machine to machine, language to language, *etc* . The key point is that the time is not dependent on ***n*** - producing ***O* (1)** algorithms.

Once we have written an ***O* (1)** method, there is generally little more that we can do from an algorithmic point of view. Occasionally, a better approach may produce a lower constant time. Often, enhancing our compiler, run-time system, machine, *etc* will produce some

significant improvement. However ***O* (1)** methods are already very fast, and it's unlikely that effort expended in improving such a method will produce much real gain!

## PRIORITY QUEUE:

Often the items added to a queue have a **priority** associated with them: this priority determines the order in which they exit the queue - highest priority items are removed first.

This situation arises often in process control systems. Imagine the operator's console in a large automated factory. It receives many routine messages from all parts of the system: they are assigned a low priority because they just report the normal functioning of the system - they update various parts of the operator's console display simply so that there is some confirmation that there are no problems. It will make little difference if they are delayed or lost.

However, occasionally something breaks or fails and alarm messages are sent. These have high priority because some action is required to fix the problem (even if it is mass evacuation because nothing can stop the imminent explosion!).
Typically such a system will be composed of many small units, one of which will be a buffer for messages received by the operator's console. The communications system places messages in the buffer so that communications links can be freed for further messages while the console software is processing the message. The console software extracts messages from the buffer and updates appropriate parts of the display system. Obviously we want to sort messages on their priority so that we can ensure that the alarms are processed immediately and not delayed behind a few thousand routine messages while the plant is about to explode.

As we have seen, we could use a tree structure - which generally provides ***O* (log *n* )** performance for both insertion and deletion. Unfortunately, if the tree becomes unbalanced, performance will degrade to ***O(n)*** in pathological cases. This will probably not be acceptable when dealing with dangerous industrial processes, nuclear reactors, flight control systems and other *life-critical* systems.

```
/***  Implementation of queue using arrays in C ***/

# include <stdio.h>
# include <conio.h>
# define SIZE 10
int arr[SIZE], front = -1, rear = -1, i ;
void insert() ;
void delete () ;
void display() ;
void main()
{
int choice ;
```

```
do
{

  printf("\n 1.INSERT \n 2.DELETE \n 3. Show  \n 4.Exit \n") ;
 printf("Enter your choice [1-4] : ") ;
scanf("%d", &choice) ;
switch(choice)
{
case 1 : printf("\n Enter element to insert");
                    scanf("%d",item);
                 insert(item) ;
                 break ;
      case 2 :
                delete() ;
                break ;
     case 3 :
              show() ;
              break ;
    case 4 : exit(1);
              break ;
    default :
               printf("Invalid option") ;
   }
} while(1) ;
getch() ;
}

void insert(int item)
{
if(rear == SIZE - 1)
{
printf("Queue is full (overflow)") ;
}
else
{
rear++ ;
arr[rear]=item ;
   if(front == -1)
     {
       front++ ;
     }
}
}
```

```
        void delete()
        {
        if(front == -1)
        {
        printf("Queue is empty (underflow)") ;
        }
        else
        {
        printf ("The DEQUEUE element is : %d, arr[front]) ;

          if(front == rear)
              {
               front = rear = -1 ;
              }
         else
              {
               front++ ;
               }
        }
        }
        void display()
        {
        if(front == -1)
        {
        printf("Queue is empty (underflow)") ;
        }
        else
        {
        printf(" The elements in queue are :FRONT ->") ;

        for(i = front ; i <= rear ; i++)
        {
            printf(" ... %d", arr[i]) ;
        }
        printf(" ... <- REAR") ;

        }
}
```

## ALGORITHM FOR ADDITION AND DELETION OF ELEMENTS IN CIRCULAR QUEUE

Data structures required for circular queue:

1. **front** counter which points to one position anticlockwise to the 1st element
2. **rear** counter which points to the last element in the queue

3. an **array** to represent the queue

```
 add _ circular ( item,queue,rear,front)
{
            rear=(rear+1)mod n;
             if (front == rear )
              then print " queue is full "
             else {
                        queue [rear]=item;
                   }
}
```

delete operation :

**delete_circular** (item,queue,rear,front)

```
{
   if(front = = rear)
   print ("queue is empty");
  else {
      front= front+1;
      item= queue[fromt];
         }
}
```

```
/*** Circular Queue Implementation using Array in C ***/
#include <stdio.h>
#include<ctype.h>
# define MAXSIZE 5
int front=1;
Int rear=1;;

int cq[MAXSIZE];
void main()
{
void add(int,int [],int,int,int);
int del(int [],int ,int ,int );
int choice=1,i,item;

printf("Circular Queue  Using Array");
while(will ==1)
{
```

```
printf("MAIN MENU: \n 1.Add element to Circular Queue \n2.Delete element from
the Circular Queue");
Printf("Enter Choice:");
scanf("%d", &choice);

switch(choice)
{
case 1:
        printf("Enter the data to insert... ");
        scanf("%d",&item);
        add(item);
        break;
case 2:
         i=del();
        printf("Value returned from delete function is %d ",i);
        break;
default:
        printf("Invalid Choice . ");
}

printf(" Do you want to do more operations on Circular Queue ( 1 for yes, any other
key to exit) ");
scanf("%d" , &choice);
} //end of outer while
} //end of main

void add(int item)
{
rear++;
rear= (rear%MAX_SIZE);
if(front ==rear)
{
  printf("CIRCULAR QUEUE FULL");
  return;
}
else
{
   cq[rear]=item;
   printf("Rear = %d Front = %d ",rear,front);
}
}
int del()
{
 int a;
```

```
 if(front == rear)
  {
   printf("CIRCULAR QUEUE IS EMPTY");
   return (0);
}
else
{
  front++;
  front = front%MAX_SIZE;
  a=cq[front];
  return(a);
  printf("Rear = %d Front = %d ",rear, front);
}
}
```

## TOWER OF HANOI PROBLEM

Tower of Hanoi is a historical problem, which can be easily expressed using recursion. There are N disks of decreasing size stacked on one needle, and two other empty needles. It is required to stack all the disks onto a second needle in the decreasing order of size. The third needle can be used as a temporary storage. The movement of the disks must confirm to the following rules,

1. Only one disk may be moved at a time
2. A disk can be moved from any needle to any other.
3. The larger disk should not rest upon a smaller one.

Question : write a c program to implement tower of Hanoi using stack ?

```
/* Program of towers of Hanoi. */
        #include <stdio.h>
        #include <conio.h>

        void move ( int, char, char, char ) ;

        void main( )
          {
            int n = 3 ;
            clrscr( ) ;
            move ( n, 'A', 'B', 'C' ) ;
            getch( ) ;
          }
          void move ( int n, char sp, char ap, char ep )
            {
                    if ( n == 1 )
```

```
                printf ("\nMove from %c to %c ", sp, ep ) ;
                else
                  {
                     move ( n - 1, sp, ep, ap ) ;
                     move ( 1, sp, ' ', ep ) ;
                     move ( n - 1, ap, sp, ep ) ;    }
        }
```

## Linked List

```
/*** Implementation of general linked list operations ***/
# include<stdio.h>
# include<conio.h>
# include <malloc.h>
struct node
{
int data;
struct node *link;
};

void main()
{
int a=111,b=2,c=3;
Int  choice, wish, num;
struct node *ptr,*ptr2,*result,*temp;
void add(struct node *, int );
struct node * search(struct node *);
void display(struct node *);
void invert(struct node *);
void del(struct node *,int);
struct node * concat(struct node *,struct node *);
ptr=NULL;
ptr2=NULL;
result=NULL; //result for storing the result of concatenation
choice=1;

while(choice ==1)
{
printf(" **** Main Menu ****  \n
          1. Add element \n
          2.Delete element \n
          3.Search element \n
```

```
            4Linked List concatenation \n
            5.Invert linked list \n
            6. Display elements  \n
Please enter the choice:");
scanf("%d",& choice);
switch(choice)
{
case 1:
            printf("Enter the element you want to add ");
            scanf("%d", &item);
            add(&ptr, item);
           display(ptr);
           break;
case 2:
          printf("Enter the element to delete ");
          scanf("%d",&item);
          del(ptr,item);
          break;
case 3:
           temp = search(ptr);
           printf("Address of first occurrence is %u ",temp);
           break;
case 4:
            /* Inputs given internally for demo only */
           printf(" Now demonstrating linked list concatenation Press any key to
continue...");
           add(&ptr2,2);
           add(&ptr2,4);
          add(&ptr2,6);
          getch();
          printf("Displaying second Linked List");
          display(ptr2);
          getch();
          result = concat(ptr,ptr2);
          printf("Now Displaying the result of concatenation");
          display(result);
          getch();
          break;
case 5:

          printf("Inverting the list ...Press any key to continue...");
          invert(ptr);
          break;
case 6:
```

```
            display(ptr);
            break;
default:
            printf("Illegal choice");
}
printf("DO you want to continue ( press 1 for yes ");
scanf("%d",& choice);
} //end of while
}

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

void display(struct node *pt)
{

while(pt!=NULL)
{

printf("

Data : %d",pt->data);
printf("
Link : %d",pt->link);
pt=pt->link;
```

```
}
}


void invert(struct node *ptr)
{

struct node *p,*q,*r;
p=ptr;
q=NULL;

while(p!=NULL)
{
r=q;
q=p;
p=p->link;
q->link=r;
}
ptr = q;
display(ptr);
}


// CONCATENATION OF LINKED LISTS

struct node * concat(struct node *p,struct node *q)
{
struct node *x,*r;


if (p==NULL)
r=q;

if (q==NULL)
r=p;
else
{
x=p;
r=x;
while(x->link!=NULL)
x=x->link;
x->link=q;
}
return(r);
```

```
}

/* SEARCHING AN ELEMENT IN THE LINKED LIST
THIS FUNCTION FINDS THE FIRST OCCURENCE OF
 A DATA AND RETURNS A POINTER TO ITS ADDRESS */

struct node * search(struct node *p)
{
struct node *temp;
int num;
temp = p;
printf("
Enter the data that you want to search ");
scanf("%d",&num);
printf("
Link of temp %u", temp->link);
while(temp->link!=NULL)
{
printf("
In while ");
if(temp->data == num)
return(temp);
temp=temp->link;
}
return(NULL);
}

/*** DELETING DATA FROM THE LINKED LIST ***/

void del(struct node *p,int num)
{

struct node *temp,*x;
temp=p;
x= NULL;

while (temp->link !=NULL)
{
if(temp->data == num)
{
if (x==NULL)
{
p = temp->link;
free(temp);
```

```
return;
}
else
{
x->link = temp->link;
free(temp);
return;
}
} //end of outer if
x=temp;
temp=temp->link;
} //end of while
printf("

No such entry to delete ");
} //end of fn.
```

**Algorithm:(Taken from wikipedia)**

- While there are tokens to be read:
    - Read a token.
    - If the token is a number, then add it to the output queue.
    - If the token is a function token, then push it onto the stack.
    - If the token is a function argument separator (e.g., a comma):
    - Until the topmost element of the stack is a left parenthesis, pop the element from the stack and push it onto the output queue. If no left parentheses are encountered, either the separator was misplaced or parentheses were mismatched.
    - If the token is an operator, $o_1$, then:
    - while there is an operator, $o_2$, at the top of the stack, and either

$o_1$ is associative or left-associative and its precedence is less than (lower precedence) or equal to that of $o_2$, or
$o_1$ is right-associative and its precedence is less than (lower precedence) that of $o_2$,

pop $o_2$ off the stack, onto the output queue;

- push $o_1$ onto the stack.
- If the token is a left parenthesis, then push it onto the stack.
- If the token is a right parenthesis:

- Until the token at the top of the stack is a left parenthesis, pop operators off the stack onto the output queue.
- Pop the left parenthesis from the stack, but not onto the output queue.
- If the token at the top of the stack is a function token, pop it and onto the output queue.
- If the stack runs out without finding a left parenthesis, then there are mismatched parentheses.

- When there are no more tokens to read:
  - While there are still operator tokens in the stack:
  - If the operator token on the top of the stack is a parenthesis, then there are mismatched parentheses.
  - Pop the operator onto the output queue.

- Exit.

```
/*** Implementation of the Above Algorithm in C ***/
#include<stdio.h>
#include<string.h>
#define size 10
char stack[size];
int top=0, item;
void push();
char pop();
void show();
char infix[30],output[30];
int prec(char);

int main()
{
   int i=0,j=0,k=0,length;
   char temp;
   printf("\nEnter an infix expression:");
   scanf("%s",infix);
   printf("\nThe infix expresson is %s",infix);
  length=strlen(infix);
  for(i=0;i<length;i++)
    {
    /*** Numbers are added to the out put QUEUE ***/
if(infix[i]!='+' && infix[i]!='-' && infix[i]!='*' && infix[i]!='/' && infix[i]!='^' &&
infix[i]!=')' && infix[i]!='(' )
          {
```

```
            output[j++]=infix[i];
            printf("\nThe element added to Q is:%c",infix[i]);
          }
/*** If an operator or a bracket is encountered...***/
else
  {
 /*** If there are no elements in the stack, the operator is added to it ***/
     if(top==0)
        {
          push(infix[i]);
          printf("\nThe pushed element is:%c",infix[i]);
         }
        else
          {      /*** Operators or pushed or popped based on the order of precedence ***/
            if(infix[i]!=')' && infix[i]!='(')
            {
               if(    prec(infix[i]) <= prec(stack[tos-1])  )
                 {
                   temp=pop();
                   printf("\n the poped element is :%c",temp);
                   output[j++]=temp;
                   push(infix[i]);
                   printf("\n The pushed element is :%c",infix[i]);
                   show();
                 }
               else
                 {
                   push(infix[i]);
                   printf("\nThe pushed element is:%c",infix[i]);
                   show();
                 }
          }
 else
  {
     if(infix[i]=='(')
        {
            push(infix[i]);
          printf("\nThe pushed-- element is:%c",infix[i]);
        }
     if(infix[i]==')')
        {
            temp=pop();
          while(temp!='(')
           {
```

```
                output[j++]=temp;
              printf("\nThe element added to Q is:%c",temp);
                printf("\n the popped element is :%c",temp);
              temp=pop();
              }
        }
}
}
}
 printf("\nthe infix expression is: %s",output);
}
  while(tos!=0)
      {
        output[j++]=pop();
       }
 printf("the infix expression is: %s\n",output);

}
/*** Functions for operations on stack ***/
void push(int item)
{
        stack[top]=item;
        top++;
}
char pop()
{
        top--;
        return(stack[top]);
}
void show()
{
        int x=top;
        printf("--The Stack elements are.....");
        while(x!=0)
        printf("%c, ",stack[--x]);
}
//Function to get the precedence of an operator
int prec(char symbol)
{
if(symbol== '(')
return 0;
if(symbol== ')')
return 0;
if(symbol=='+' || symbol=='-')
```

```
return 1;
if(symbol=='*' || symbol=='/')
return 2;
if(symbol=='^')
return 3;
return 0;
}
```

## Stack using Linked List

In the lists first we have to create a nodde with the fields info and next where info Stores the values in the stack and next is a pointer which points to the next node i.e., it Stores the next node address.

```
type dy strust stack *stpr;
struct stack
{
int info;
stpr next;
};
```

In implementing the stack using linked lists first we create a header and insert the elements to that header.

```
create_head
first create memory for header
H=stpr malloc(sizeof(struct stack));
Check for availability of memory
If(T==NULL) “out of space”
Else
Make the next field of the header NULL
H->next=NULL;
```

PUSH(sptr H,INT x)

in the operation push we have to insert the element at the next of the header in linked list.to insert we have to create a node and insert element in the information field and we have to place that node in right position. take header in another pointer variable
sptr P=H;
create a new node
n=(sptr)malloc(sizeof(struct sptr));
if(n==NULL) "out of space"
else
n->info=x;
n->next=H->next;
H->next=n;

POP

int pop(sptr H)
in this pop(delete) operation we have to delete the element in the node next to header
check wether elements exists or not
if(H->next==NULL) "no elements in the stack"
else
store the element in a variable x
x=H->next->info
delete the node i.e.,change the links in the linked lists
P=H->next;
H->next=H->next->next;
we have to free the memory location by H->next;
delete(P);

Here is the code on how to implement stack using linked list using C programming. The Comments will help you get to know how the code works.

```
#include<conio.h>
#include<stdio.h>
typedef struct node *nptr;
struct node
{
int data;
nptr next;
};
nptr createhead(void);
void display(nptr s);
void push(nptr s,int x);
int pop(nptr s);
nptr createhead() /*Function to create head*/
{
```

```
nptr H;
H=(nptr)malloc(sizeof(struct node));
if(H!=NULL)
{
H->next=NULL;
return(H);
}
else
printf("\n\tOut of Space");
return;
}
void display(nptr s) /*Function to display the elements*/
{
nptr p;
p=s;
while(p->next!=NULL)
{
printf("%5d",p->next->data);
p=p->next;
}
}
void push(nptr s,int x) /*Function to push the elements*/
{
nptr temp;
temp=(nptr)malloc(sizeof(struct node));
if(temp==NULL)
{
printf("Out of Space");
return;
}
else
{
temp->data=x;
temp->next=s->next;
s->next=temp;
}
}
int pop(nptr s) /*Function to pop the elements*/
{
nptr temp;
int y;
if(s->next==NULL)
{
printf("Underflow on Pop");
return(-1);
```

```
}
else
{
y=s->next->data;
temp=s->next;
s->next=temp->next;
free(temp);
return(y);
}
}
```

## Queue Using Linked List

```
#include<malloc.h>
#include<stdio.h>
struct node{
int data;
struct node *next;
};

void init(struct node *n){
n->next=NULL;
}
void enqueue(struct node *root,int data){
struct node *j=(struct node*)malloc(sizeof(struct node));
j->data=data;
j->next=NULL;
struct node *temp ;
temp=root;
while(temp->next != NULL)
{
temp=temp->next;
}
temp->next=j;
printf("Data Inserted is : %d\n",data);
}
void Dequeue(struct node *root)
{
if(root->next==NULL)
{
printf("No Element to Delete \n");
}
else
```

```
{
struct node *temp;
temp=root->next;
root->next=temp->next;
printf("Data Deleted is : %d\n",temp->data);
free(temp);
}
}
void main()
{
struct node my_queue;
Init(&my_queue);
Enqueue(&my_queue,10);
Enqueue(&my_queue,50);
Enqueue(&my_queue,570);
Enqueue(&my_queue,5710);
Dequeue(&my_queue);
Dequeue(&my_queue);
Dequeue(&my_queue);
}
```