DATA STRUCTURES USING C

UNIT – I: Introduction to Data Structures: Introduction to the Theory of Data Structures, Data Representation, Abstract Data Types, Data Types, Primitive Data Types, Data Structure and Structured Type, Atomic Type, Difference between Abstract Data Types, Data Types, and Data Structures, Refinement Stages.

Principles of Programming and Analysis of Algorithms: Software Engineering, Program Design, Algorithms, Different Approaches to Designing an Algorithm, Complexity, Big 'O' Notation, Algorithm Analysis, Structured Approach to Programming, Recursion, Tips and Techniques for Writing Programs in 'C'.

Introduction to the Theory of Data Structures:

The study of computer science includes the study of organization and flow of data in a computer. Data structures is the branch of computer science that releases the knowledge of

- how the data should be organized?
- how the flow of data should be controlled? and
- how the data structure should be designed and implemented to reduce the complexityand increases the efficiency of the algorithm?

Theory of data structures also helps you to understand and use the concept of abstraction, analyze problems step by step and develop algorithms to solve real world problems. It enables you to design and implement various data structures like stack, queues, linked lists, trees and graphs.

Effective use of principals of data structures increases efficiency of algorithms to solve problems like searching, sorting.

Need of data structures:

- A data structure helps you to understand the relationship of one data element with the otherand organize it within the memory.
- Sometimes the organization might be simple and can be very clearly vision.
- A data structure helps you to analyze the data, store it and organize it in a logical ormathematical manner.
- As applications are getting complexed and amount of data is increasing day by day, theremay arise the following problems:

Processor speed: To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

Multiple requests: If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process

In order to solve the above problems, data structures are used.

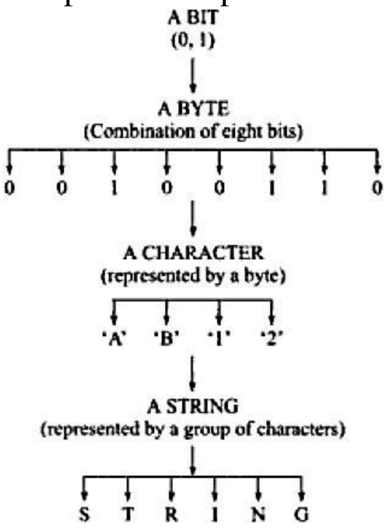
Data Representation:

Various methods are used to represent data in computers.

- The basic unit of data representation is a **bit i.e., 0 or 1**.
- 8 bits together form one **byte** which represents a **character** and one or more than onecharacters are used to form a **string.**

Integer Representation:

An integer is the basic data type which is commonly used for storing negative as well as non-negative integer numbers. The non-negative data is represented using **binary number system**. In this, each bit position represents the power of 2.The right most bit position represents 2^0 which is 1.



For negative binary numbers the methods of representation used are one's **complement and two's complement.**

In **one's complement** method, the number is represented by complementing each bit.

Ex:

0 0 1 0 0 1 1 0

1's complement is **1 1 0 1 1 0 0 1**

In **two's complement** method, 1 is added to one's complement representation of the negative number.

Ex: 1's complement of **0 0 1 0 0 1 1 0** is **1 1 0 1 1 0 0 1** and can get the 2'scomplement value by adding 1 to one's complement

1 1 0 1 1 0 0 1

1

1 1 0 1 1 0 1 0

Real Number Representati

The method used to represent real numbers in computers is floating – point notation. In this notation, the real number is represented by a number called a **mantissa**, times abase raised to an integer power, called an exponent.

For Example, 209.52 could be represented as 20952 x 10^{-2}. The mantissa is 20952 and exponent is -2. Both mantissa and exponents are 2's complementary binary integers.

Now the above example can be representation as

20952 is 00000000101000111011 (it is in 24-bit representation)

-2 is 11111110 (it is in 8-bit representation)

20952 x 10^{-2} is represented as 0000000101000111011. 11111110

Character Representation:

Computers work in binary. As a result, all characters, whether they are letters, punctuation or digits,are stored as binary numbers. All of the characters that a computer can use are called a character set.

Two standards are in common use:

- American Standard Code for Information Interchange (ASCII)
- Unicode

ASCII

ASCII uses seven bits, giving a character set of 128 characters. The characters are represented in atable called the ASCII table. The 128 characters include:

- 32 control codes (mainly to do with printing)
- 32 punctuation codes, symbols, and space
- 26 upper-case letters
- 26 lower-case letters
- numeric digits 0-9

Extended ASCII

Extended ASCII uses eight bits, giving a character set of 256 characters. This allows for specialcharacters such as those with accents in languages such as French and Spanish.

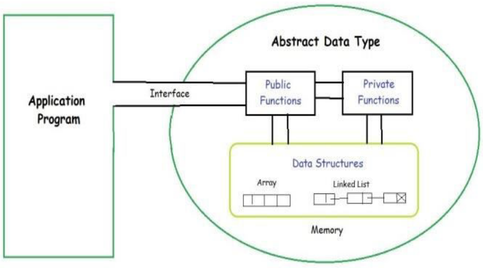
Unicode

While suitable for representing English characters, 256 characters is far too small to hold every character in other languages, such as Chinese or Arabic. Unicode uses 16 bits, giving a range of 65,536 characters. It more suitable for languages with a large number of characters, but it requires more memory.

Abstract Data Types:-

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.

The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.



The user of data type does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented.

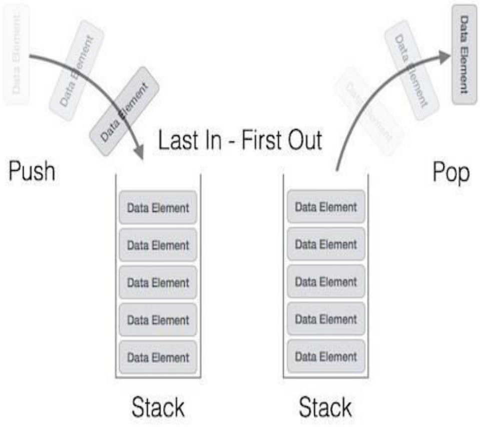
So a user only needs to know what a data type can do, but not how it will be implemented.

Example: - Stack ADT

Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

Data types

Each programming language has its own set of data types. There are several definitions like

- A method of interpreting a bit pattern is often called a data type.
- The term data type refers to the implementation of the mathematical model specified by an ADT.
- A data type is the abstract concept Defined by a set of logical properties.

Once such abstract data type is defined and the legal operations involving that type or specified it can then be implemented.

Why do we need a data type?

We can think of a Universal data type that may hold any value like character, integer, float or any complex number. Use of such data type has two disadvantages

- Large volume of memory will be occupied by even a small size of data.
- Different types of data required different interpretation of bit strings while reading or writing.

Primitive data types

Every computer has a set of native data types this means that it is constructed with a mechanism for manipulating bit pattern at a given location as binary numbers.

Primitive data types are basic data types of any language that form the basic unit for the data structure Defined by the user.

A primitive data type defines how the data will be internally represented in, stored, and retrieved from the memory.

Few primitive data types which are commonly available with most programming languages are

- Integer
- character
- real or float numbers

Integer:

An integer data type is a primitive data type that may represent a range of numbers from $-2^{(n-1)}$ +1 to $2^{(n-1)}$ +1 where n depends upon the number of bits used to constitutes one word in the computer.

Character:

At a more general level, information can be represented in the form of characters. Any symbol from set 0 – 9, A – Z, a – z and other special symbols is a character. Most of the computers use 8 bits to represent a character. So that it has 256 characters of 8 bits. The number of bits necessary to represent a character in a particular computer is called the byte size.

Real or float numbers:

Real number consists of two parts mantissa and characteristic. A real number data type is generally denoted with the term float. This is because computers usually represent a real number using a floating point notation. There are many varieties of float floating point mutations and each has individual characteristic.

A real number is represented as $m \times n^r$

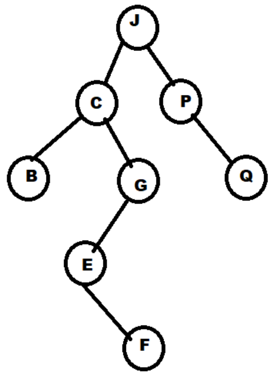
Where m is the mantissa and n is the base (which is fixed 10) and r is the exponent.

Data structure and structure type:

The term Data Structure refers to a set of computer variables that are connected in some logical or mathematical manner.

More preciously, a data structure can be defined as the structural relationship present within the data set and that should be viewed in 2 tuple (N,R) where ‘N’ is the finite set of nodes representing the data structure and ‘R’ is the set of relationships among those nodes .

For example in a tree Data Structure each node is related to each other in a parent child relationship.



Structural type refers to a data structure which is made up of one or more elements known as components. These elements are similar data structures that exist in the language. The components of the structured data type are grouped together according to a set of rules for example the representation of polynomials requires at least two components

- coefficient

- exponent

The two components together form a composite type structure to represent a polynomial.

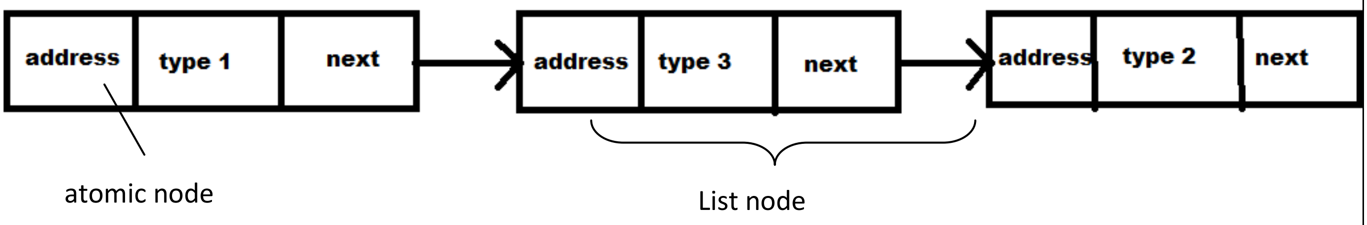
Atomic type:

Generally, Data Structure is represented by a memory block which has two parts.

- Data storage
- address storage

This facilities in storing the data and relating it to some other data by means of storing pointers in the address part.

An atomic type data is a data structure that consists only the data items and not the pointers does for a list of data items several atomic type nodes may exist each with a single data item corresponding to one of the legal data types.



In the above list of atomic nodes maintained using list of notes in each node type represents the type of data stored in the atomic note to which the list node points. 1 stands integer type, 2 for real number and 3 for character type or any different assumption can be made at implementation level to indicate different data types

Difference between abstract data types, data types and data structures:

To avoid confusion between abstract data types data types and data structures it is relevant to understand the relationship between the tree

- An abstract type is the specification of the data type which specifies the logical and mathematical model of the data type.
- Data Type is the implementation of the extract data type.
- Data structures refers to the collection of computer variables that are connected in some specific manner .

Thus, there seems to be an open relationship between the tree that is a data type has its root in the Abstract data type and the data structure comprises a set of computer variables of same or different data types.

Refinement stages

The best approach to solve a complex problem is to divide it into smaller parts such that each part becomes an independent module which is easy to manage.

An example of this approach is the system development life cycle methodology.

This helps in understanding the problem analyzing solution and handling this problem efficiently.

The application or the nature of problem determines the number of refinement stages required in the specification process. Different problems have different number of requirement stages but in general there are four levels of refinement processes.

- Conceptual or abstract level
- Algorithmic for data structures
- Programming or implementation
- Applications

Conceptual level:

At this level we decide how the data is related to each other and what operations are needed details about how to store data and how various operations are performed on the data or not decide at this level.

Algorithm or data structure level:

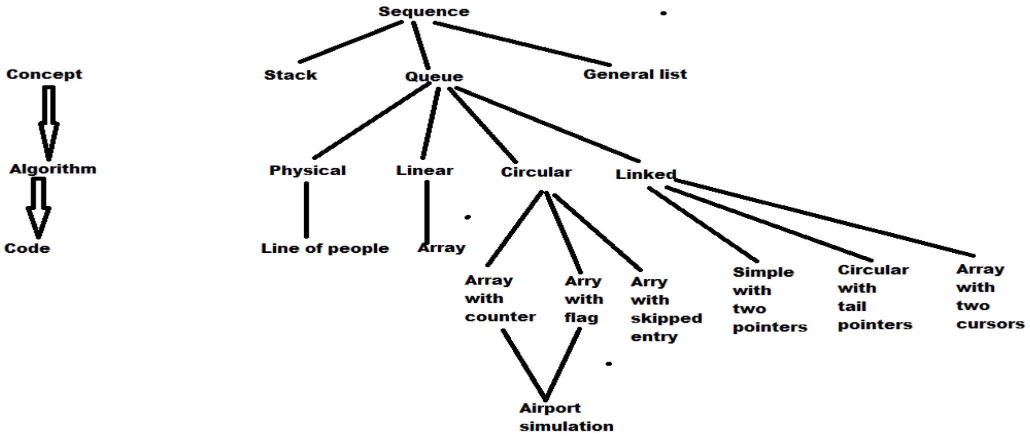
Data Structure level we decide about the operations on the data as needed by our problem for example we decide what kind of data structure will be required to solve the problem.

Programming or implementation level:

At implementation level we decide the details of how the data structure will be represented in the Computer memory.

Application level :

This level settles all details required for a particular application such as names for variables are special requirements for the operations imposed by applications.



Refinement of a QUEUE

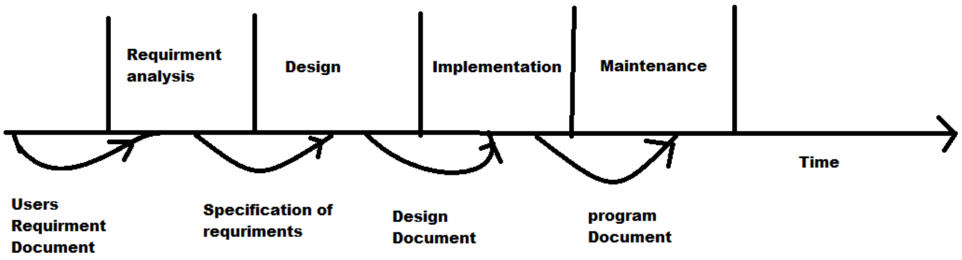
In the above figure, the first two levels are often called conceptual. The middle two levels can be called an algorithmic yes they are concerned with representing data and the operations performed on the same last level is basically concerned with the programming.

Software engineering

Software engineering is the theory and practice of methods helpful for the construction and maintenance of large software systems. Development of good software is a tedious process which continuous for long time before the software or program gets the final score of India is put into use. There are many stages in the software development cycle. The process is often referred to as software development life cycle (SDLC).

The different steps in software development life cycle are as follows

- Analyze the problem preciously and completely
- Built a prototype and experiment with it until all specifications are finalized



- Design the algorithm using the tools of data structures
- Verify the algorithm such that its correctness is self-evident
- Analyze the algorithm to determine its requirements
- Code the algorithm in to an appropriate programming language
- Test and evaluate the program with carefully chosen data
- Refine and repeat the foregoing steps until the software is complete
- Optimize code to improve performance
- Maintain the program so that it meets the changing needs of its users

Program design

Program design can be considered as an important phase of the software development life cycle. It is in this space that the algorithms and data structures to solve a problem or proposed. Some important points are

- As the design stage involves taking the specification and designing solutions to the problem.
- Another important point which should be kept in mind while developing a solution strategy is that it should work correctly in all conditions.
- Generally the people who use the system or not aware of the program design you have adopted. Thus, carries a system manual which is a detailed guide to how the design was achieved.
- Large program should be divided into small modulus and the submodules by following one of the two decomposition approaches.
- Top down approach
- Bottom up approach

Other important criteria by which a program can be judged for execution time and storage requirement.

Algorithms

The term algorithm refers to the sequence of instructions that must be followed to solve a problem. In other words an algorithm is a logical representation of the instructions which should be executed to perform a meaningful task.

- An algorithm has certain characteristics
- Each instruction should be unique and concise.
- Each instruction should be relative in nature and should not be repeated infinitely.
- Repetition of same task should be avoided.
- The result should be available to the user after the algorithm terminates.

After an algorithm has been designed its efficiency must be analyzed. This involves determining whether the algorithm is economical in the use of computer resources. The importance of efficiency of an algorithm is in the correctness that is does it always produce the correct result and program complexity which is considered as both the difficulty of implementing an algorithm along with its efficiency.

Different approaches to design an algorithm

System may be divided into smaller units called modulus. Advantage of modularity is that it always the principle of separation of concerns to be applied into two phases: When dealing with details of each module in isolation and when dealing with overall characteristics of all modules and their relationships in order to integrate them into a system: Modularity enhances design clarity, which in turn cases implementation debugging testing documenting and maintaining of the product.

A system consists of components, which have components of their own. Indeed system is a hierarchy of components. The highest level component corresponds to the total system. To design such a hierarchy there are two possible approaches.

- Top down approach
- Bottom up approach

Top down approach:

A top down design approach identifying the major components of the system are program. Decomposing them into the lower level components and iterating until the desired level of the module complexity is achieved. Top down design method takes the form of stepwise refinement. In this we start with the topmost module and incrementally add module that it calls.

In top down approach which start from a abstract design. In each step design is defined into most concentrate level until we reach the level where no more refinement is needed and the design can be implemented directly.

Bottom up approach:

A bottom up design thought with designing the most basic or primitive components and the proceeds to higher level components. Bottom up method works layers of abstraction. Starting from the very bottom the operations that provide a layer of abstraction or implemented. The operations of this there are then used to implement more powerful operations and still higher layer of abstraction. Until the stage is reached with the operations supported by the layer are those designed by the system.

Top-down vs Bottom-up Approch

Top-Down Approach	Bottom-Up Approach
Top-Down Approach is Theory-driven.	Bottom-Up Approach is Data-Driven.
Emphasis is on doing things (algorithms).	Emphasis is on data rather than procedure.
arge programs are divided into smaller programs which is known as decomposition.	rograms are divided into what are known as objects is called Composition.
Communication is less among the modules.	ommunication is a key among the modules.
ely used in debugging, module documentation, etc.	Widely used in testing.
e top-down approach is mainly used by Structured programming languages like C, Fortran, etc.	bottom-up approach is used by Object-Oriented ogramming languages like C++, C#, Java, etc.
y contains redundancy as we break up the problem into smaller fragments, then build that section separately.	s approach contains less redundancy if the data ncapsulation and data hiding are being used.

Complexity

When we talk of complexity in context of computers we call it computational complexity. Computational complexity is a characterization of the time or space requirements for solving a problem by a particular algorithm. These requirements are expressed in terms of a single parameter that represents the size of the problem.

The program requires to main considerations

- Time complexity
- space complexity

Time complexity:

While measuring the time complexity of an algorithm we concentrate on developing only the frequency count for all case statements. This is because it is often difficult to get reliable timing figure because of clock limitations and the multiprogramming or The Sharing environment.

Algorithm A	a = a + 1

Algorithm
B
	for x = 1 to n step 1
	a = a + 1
	Loop

Algorithm
C
	for x = 1 to n step 1
	for y = 1 to n step 1
	a = a + 1
	Loop

In the above example

Algorithm has only one statement and it is independent so we can say that the frequency count of algorithm a is 1. In algorithm bi the key statements for 3 e which is having assignment operation with yellow for show the number of times it is executed is n so that the frequency found for this algorithm is n.

According to the third algorithm the frequency count for the statement equals to a + 1 is n square as the inner loop runs and times each time the outer loop runs the outer loop also runs for n times so n square is said to be different in increasing order of magnitude on n.

Space complexity:

The space needed by the program is the sum of the following components.

- Fixed space requirement: This include the instruction space for simple variables fixed size structured variables and constants.
- Variable space requirement: This consists of space needed by a structured variable whose size depends on particular instance of variables. It also includes the additional space required when the function uses recursion.

Big "O" Notation:

If f(n) Represents the computing time of some algorithm and g(n) represents a known standard function like n,n2,n log n, etc. then to write;

f(n) is O g(n)

means that f(n) of n is equal to biggest order of function g(n). This implies only when:

$|f(n)|<=C|g(n)|$ for all sufficiently large integers n. Where she is the constant whose value depends upon various factors.

Big O Notation helps to determine the time as well as space complexity of the algorithm. Using the big O notation, the time taken by the algorithm space required to run the algorithm can be ascertained. This information is useful to set the prerequisite is of algorithms and to develop and design efficient algorithms in terms of time and space complexity.

The big O notation has been extremely useful to classify algorithms by their performance. Developers use this notation to reach to the best solution for the given problem.

Most common computing times of algorithm

If the complexity of any algorithm is O(1) it means that the computing time of the algorithm is constant O(n) and it is called linear time which implies that it is directly proportional to n. O(n^2) is called as quadratic time, O(n^3) is the cubic time , O(2^n) is exponential time, O(log n) and O(n log n) are the logarithmic times.

The common computing times of algorithms in the order of performance or as follows:

- O(1)
- O(log n)
- O(n)
- O(n log n)
- O(n^2)
- O(n^3)
- O(2^n)

Algorithm analysis

Different ways of solving a problem and there are different algorithms which can be designed to solve a problem. Therefore there is a difference between a problem and an algorithm. A problem has a single problem statement that describes it in some general terms. However there are many different ways to solve the problem and some of the solutions may be more efficient than the others.

Algorithm focuses on computation of space and time complexity which usually depends on the size of the algorithm and input.

There are different types of time complexities which can be analyzed for the algorithm.

- Best case time complexity
- Average case time complexity
- Worst case time complexity

Best case time complexity :

The best case time complexity of an algorithm is a measure of the minimum time that the algorithm will required for an input of size n. The running time of mini algorithms where is not only for the inputs of different sizes but also for the different inputs of same size.

Average case time complexity :

The time that an algorithm will be required to execute a typical input data of size n is known as average case time complexity. We can say that the value that is obtained by averaging the running time of an algorithm for all possible inputs of size n can be determined average case time complexity.

Worst case time complexity :

Time complexity of an algorithm is a measure of the maximum time that the algorithm will require for an input of size n. Therefore, if various algorithms for sorting are taken into account and say "n" input data items are supplied in reverse order for any sorting algorithm, then the algorithm will require n^2 operations to perform the sort which will correspond to the worst case time complexity of algorithm.

Another important step which can be considered in the analysis of an algorithm is identifying the Abstract operation on which an algorithm is based.

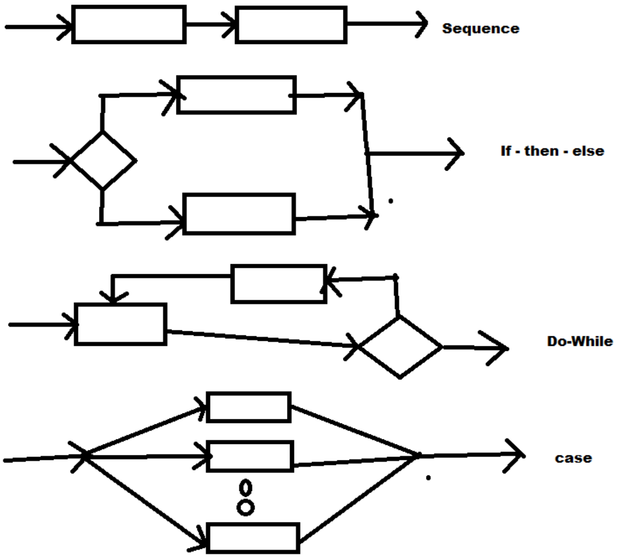
Structured approach to programming :

Structured program is a subset of software engineering. It is a method for designing and coding programs in a systematic, organized manner. The main concentration of structured programming is on the technical aspects of programming. Where are software engineering puts equal priority on technical and managerial and psychological and financial aspects of software development?

The term structured programming was coined by Dijkstra in the article structure programming. We deal with various tools required for building a structured program.

Program = algorithm + data structure

Some of the control structures which can be used for structured programming.



Structured programming Emphasis functional specifications and tries to ensure that only one primary function is allocated to any module.

Recursion

A recursive routine is one whose design includes a call to itself. In design phase of software development we use various problem solving methods in which recursion can be one of the powerful tools.

Example:

Factorial(a)

Int a;

{

    Int fact=1

```
        If(a>1)
        Fact=a*factorial(a-1)
        Return(fact)
}
```

In the above example factorial function is calculating the values from the given number to 1. Each time it can be multiplied the number "a" with its previous number. This process will be continue till the number reaches to 1. So, the same procedure can be done in several times. In that way the factorial is used in recursive method to calculate the factorial of a given number.

Looping and recursion

Loops are used when we want to execute a part of the program or block of statement several times there are many statements for the loop in purpose for example while, do while, for, switch etc..

Recursive function is the function which calls itself again and again.

Principles of recursion

Some basic principles which are used while designing algorithm with recursion are

Find the Key step

When beginning with the design of algorithm true recursion one should try to find out the key step for the solution.

find the stopping rule

The stopping rule indicates that the problem or a substantial part of it is done and the execution of the algorithm can be stopped.

Outline your algorithm

After determining the key steps for the problem and finding the cases which are to be handled the next step is to come by this to using an if statement to select between them.

Check termination

There should be taken to ensure that there will always terminate after the finite number of steps and the stopping rule should also be satisfied.

Draw a recursion tree

The key tool for the analysis of recursive algorithm is the recursion tree as it helps in determining the amount of memory that the program will require and the total size of tree reflects the number of times the key step will be performed and the total time needed for the program.

Example:

Factorial without recursion:

```
Main()
{
        If(n=0)
          Printf("factorial of zero is 1");
         Else
         {
            While(n>1)
            {
               Fact*=n;
               N--;
             }
           }
}
```

Factorial using recursion:

```
Factorial(a)
Int a;
{
      Int fact=1
      If(a>1)
      Fact=a*factorial(a-1)
      Return(fact)
}
```

Comparison between recursion and iteration

For recursion, the recursion tree can help in providing Useful information for deciding when recursion should and should not be used.

A recursion tree can be defined as a part of tree showing the recursive calls. Therefore if a procedure or function makes only one recursive call to itself, it's a recursion tree is simple, rather it is a chain, each vertex having only one child.

Example:

In the problem of finding the factorial of a number, the main task is to calculate the factorial from (n-1) down to 1. Recursion tree for calculating factorial from bottom to top we can obtain an iterator program. Thus, when the tree is reduced to chain, the transformation of recursion to iteration is easy and saves both space and time.

Thus, in this example of finding factorial, after studying the recursion tree it was found that using iteration is easy and economical than recursion.

But on the other hand, the recursion tree for calculating Fibonacci series is not a chain but contains many vertices signifying duplicate tasks. When a recursive program is Run, it sets up a stack to use while traveling the tree, but if the result stored on the stacks are discarded rather than kept in some other data structure for further use, then a great deal of duplication of work may occur as in recursive calculation of Fibonacci series.

Therefore, for the Fibonacci numbers we need additional temporary variables to hold the information required for calculating the current number.

Finally, by setting up for taking another data structure for such type of calculation, it is possible to change any recursive program into the non recursive form.

Tips and techniques for writing programs in C

A program in any language is a collection of one or more functions. Every function is a collection of statements which performs a specific task. For example, a program written in C language can have the following format.

```
Comments

Preprocessor directives

Global variables

Main()

{

    Local variables

     Statements
```

```
      ----------------------
     -------------------------
}
Func1()
{
      Local variables
       Statements
      ----------------------
     -------------------------
}
Func2()
{
      Local variables
       Statements
      ----------------------
     -------------------------
}
```

C program start with comments between /* and */. Comments can be given anywhere in the program.

The preprocessor directives are executed before C program code passes through the compiler. These preprocessor directives make programs more efficient. Most commonly used directives are #include which includes files, # define which defines the Macro name and macro expansion.

Example:

```
#include<stdio.h>
#define true 1;
```

Decoration for global variables, which have same data type and same name throughout the function and are defined outside the main() function. But the declaration of too many Global variables is not advisable. To make the program more efficient constants are used. Constants are values that can be stored in the memory and can be changed during the education of program. They can be defined for numeric, character and string data.

Another word which can be used is typedef which is used for defining new data types.

Syntax:-

Typedef type and data name.

Here type is the data type and data name is the user defined and name.

C program contains the main() function but a program should be divided into functions. A function is a self contained sub-program which performs some specific, well defined task.

UNIT – II

Arrays: Introduction to Linear and Non- Linear Data Structures, One- Dimensional Arrays, Array Operations, Two- Dimensional arrays, Multidimensional Arrays, Pointers and Arrays, an Overview of Pointers

Linked Lists: Introduction to Lists and Linked Lists, Dynamic Memory Allocation, Basic Linked List Operations, Doubly Linked List, Circular Linked List, Atomic Linked List, Linked List in Arrays, Linked List versus Arrays

Q:What is Data Structure? Explain types of data structures?

Data structure is a specialized way for organizing, processing, retrieving and storing data. **Data structure** is a representation of the logical relationship between individual elements of data.The data structure is not any programming language like C, C++, java, etc. It is a set of algorithms that we can use any programming language to structure the data in the memory.

Data structure covers the following points

- Amount of memory require to store.
- Amount of time require to process.
- Representation of data in memory.
- Operations performed on that data.

Classification of Data Structures:

Data Structures are generally classified into primitive and non-primitive data structures.

1. Primitive Data Structures:

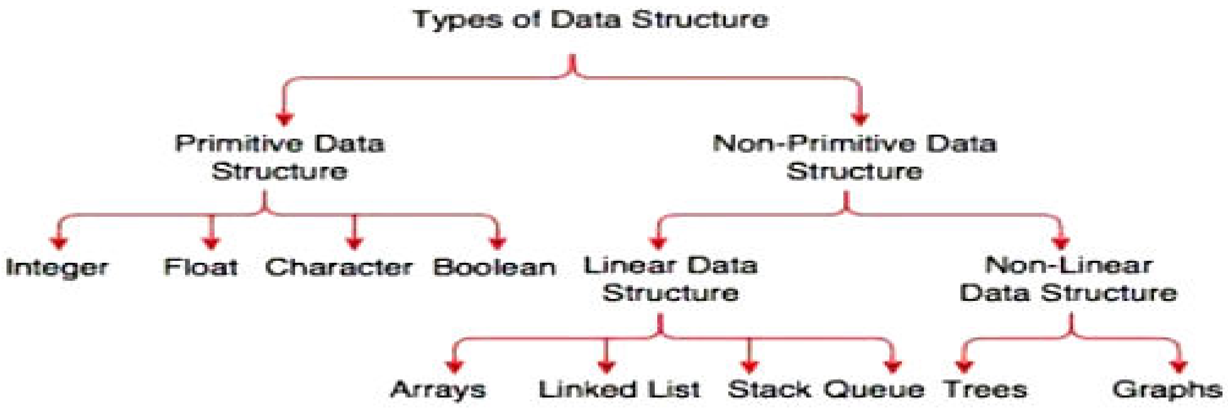
Basic data types of programming languages such as integer, real, character and boolean are known as primitive data structures. These data types consist of characters that cannot be divided, hence they are also called as simple data types.

2. Non-Primitive Data Structures:

Non-Primitive Data Structures are used for processing complex numbers. Arrays, Linked Lists, Stacks, Queues, Graphs and Tress are examples of Non-Primitive Data Structures. Based on structure and arrangement of data, Non-Primitive Data Structures are further classified into two types. They are:

A. Linear Data Structures

B. Non - Linear Data Structures



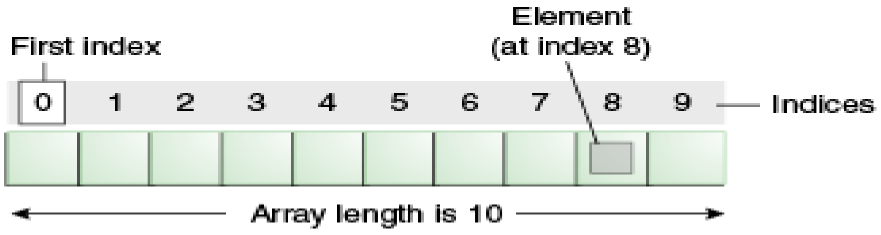
The following figure shows the classification of data structures.

A. Linear Data Structures

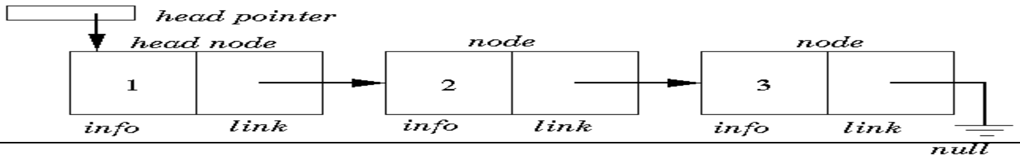
A data structure is said to be liner if its elements form a sequence or a liner list. In linear data structures, the data is arranged in a linear fashion although the way they are stored in memory need not be sequential. the term linear indicates the relationship between adjacent elements.

Example:

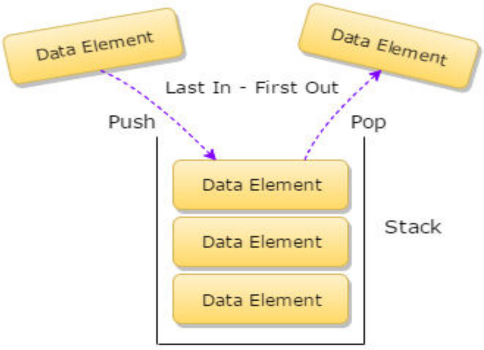
1. Array: An array is a Linear Data structure. It is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. Array elements can be accessed with the help of index.



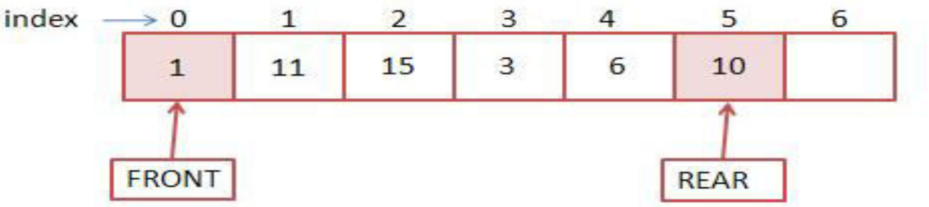
2. Linked list A linked list is a linear data structure, which is connected together via links. Linked List is a sequence of nodes which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array.



3. **Stacks:** stack is a **linear data structure**, collection of items of the same type. Stack follows the Last In First Out (LIFO) fashion wherein the last element entered is the first one to be popped out. In stacks, the insertion and deletion of elements happen only at one endpoint of it.



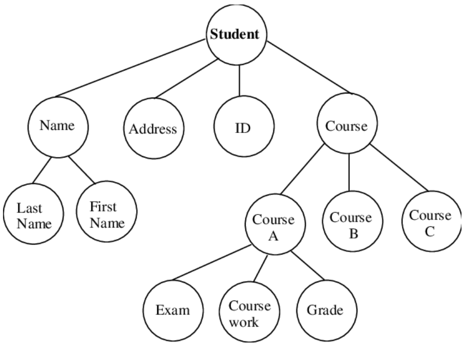
4. Queues : Queue is a linear data structure that follows the ***FIFO*** rule (First In First Out*).* Insertion is done from the back (the rear end) and deletion is done from the front.



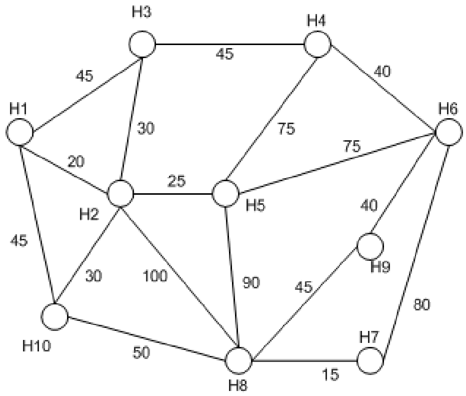
B. Non – Linear Data Structures:

A data structure is said to be Non-Linear, if the data items or elements not arranged in any sequence those are arranged in a random order.

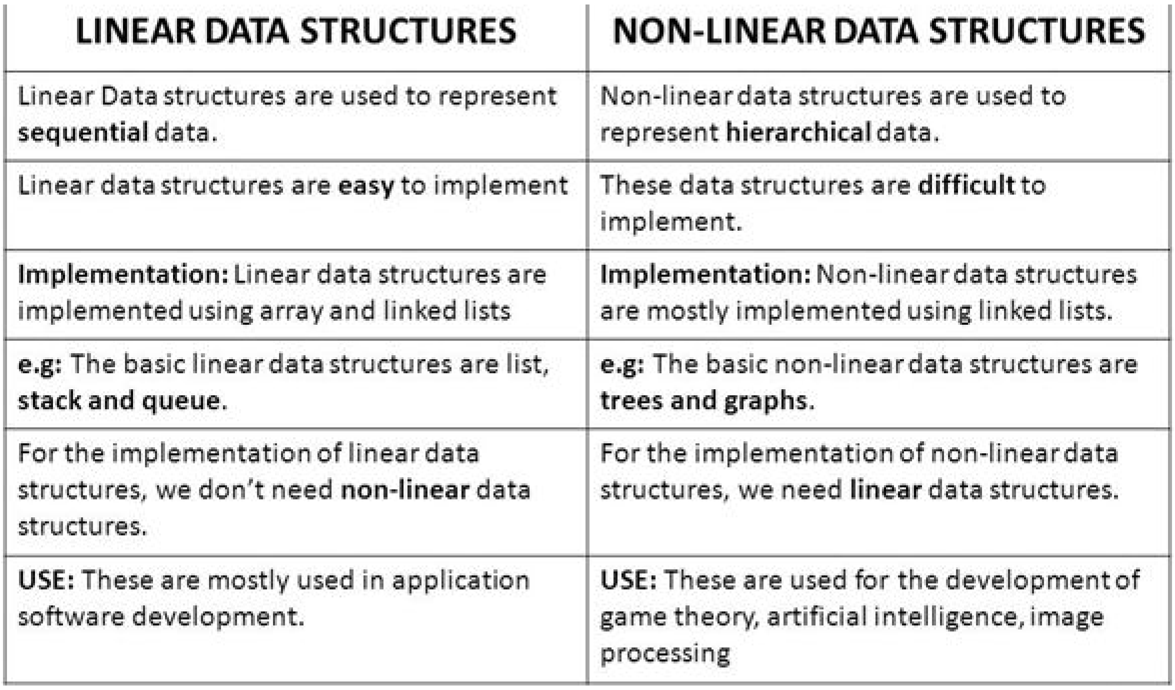
Tree: The Tree data structure uses a hierarchical form of structure to represent its elements. One of the famous tree data structures is the Binary tree. A tree uses a node-like structure to make a hierarchical form, where each node represents a value



Graph: A graph is a non linear data structure basically uses two components vertices and edges. In the graph, Edges are used to connect vertices.



Q: What are the Differences between Linear and Non Linear Data Structures?



LINEAR DATA STRUCTURES	NON-LINEAR DATA STRUCTURES
Linear Data structures are used to represent **sequential** data.	Non-linear data structures are used to represent **hierarchical** data.
Linear data structures are **easy** to implement	These data structures are **difficult** to implement.
Implementation: Linear data structures are implemented using array and linked lists	**Implementation:** Non-linear data structures are mostly implemented using linked lists.
e.g: The basic linear data structures are list, **stack and queue.**	**e.g:** The basic non-linear data structures are **trees and graphs.**
For the implementation of linear data structures, we don't need **non-linear** data structures.	For the implementation of non-linear data structures, we need **linear** data structures.
USE: These are mostly used in application software development.	**USE:** These are used for the development of game theory, artificial intelligence, image processing

Q: Explain about Arrays and their types ?

Array Data Structure:

An array is defined as **finite ordered collection of homogenous** elements, stored in contiguous memory locations.

- **finite** means data range must be defined.
- **ordered** means data must be stored in continuous memory addresses.
- **homogenous** means data must be of similar data type.

An array is a variable which can store collection of similar data items in continuous memory locations with single name at a time. It provides static allocation, which means the space allocation done once during the compile time cannot be changed during run-time.

Types of Arrays:

Arrays are classified into two types. They are as follows...

1. Single Dimensional Array / One Dimensional Array
2. Multi Dimensional Array

1. Single Dimensional Array:

Single dimensional arrays are used to store a row of values of same data type. In single dimensional array, data is stored in linear form. Single dimensional arrays are also called as one-dimensional arrays, Linear Arrays or simply 1-D Arrays.

Declaration of Single Dimensional Array:

We use the following general syntax for declaring a single dimensional array...

Syntax: datatype arrayName [size] ;

Example:- int marks[60] ;

The above declaration of single dimensional array reserves 60 continuous memory locations of 2 bytes each with the name 'marks' and tells the compiler to allow only integer values into those memory locations.

Initialization of Single Dimensional Array:

Initialization means assigning values to values to variables at the time its declaration.

We use the following general syntax for declaring and initializing a single dimensional array with size and initial values.

Syntax: datatype arrayName [size] = {value1, value2, ...} ;

Compile time Array initialization

Compile time initialization of array elements is same as ordinary variable initialization. The general form of initialization of array is,

data-type array-name[size] = { list of values };

```
/* Here are a few examples */
int marks[4]={ 67, 87, 56, 77 };    // integer array initialization

float area[5]={ 23.4, 6.8, 5.5 };   // float array initialization
```

```
int marks[4]={ 67, 87, 56, 77, 59 };    // Compile time erro
```

Runtime Array initialization

An array can also be initialized at runtime using scanf() function. This approach is usually used for initializing large arrays, or to initialize arrays with user specified values.

```
#include<stdio.h>
void main()
{
   int arr[4];
   int i, j;
   printf("Enter array element");
   for(i = 0; i < 4; i++)
   {
      scanf("%d", &arr[i]);    //Run time array initialization
   }
   for(j = 0; j < 4; j++)
   {
      printf("%d\n", arr[j]);
   }
}
```

Accessing Elements of Single Dimensional Array:

To access the elements of single dimensional array we use array name followed by index value of the element that to be accessed. Here the index value must be enclosed in square braces. The index value in an array is also called as subscript or indices.

We use the following general syntax to access individual elements of single dimensional array.

Syntax: arrayName [indexValue]

Example: marks [2] = 99 ;

In the above statement, the third element of 'marks' array is assinged with value '99'.

2. Multi Dimensional Array:

An array of arrays is called as multi dimensional array. In simple words, an array created with more than one dimension is called as multi dimensional array. Multi dimensional array can be of two dimensional array or three dimensional array or four dimensional array or more.

. The 2-D arrays are used to store data in the form of table. We also use 2-D arrays to create mathematical matrices. In two dimensional arrays first index specifies row index and the second index specifies column index.

Declaration of Two Dimensional Array:

We use the following general syntax for declaring a two dimensional array...

Syntax: datatype arrayName [rowSize] [columnSize] ;

Example: int matrix_A [2][3] ;

The above declaration of two dimensional array reserves 6 continuous memory locations of 4 bytes each in the form of 2 rows and 3 columns.

Initialization of Two Dimensional Array:

We use the following general syntax for declaring and initializing a two dimensional array with specific number of rows and coloumns with initial values.

Syntax: datatype arrayName [rows][colmns] = {{r1c1value, r1c2value, ...},{r2c1, r2c2,...}...} ;

Example: int matrix_A [2][3] = { 1, 2, 3,4, 5,6 } ;

Accessing Individual Elements of Two Dimensional Array

To access elements of a two dimensional array we use array name followed by row index value and column index value of the element that to be accessed. We use the following general syntax to access the individual elements of a two dimensional array...

Syntax: arrayName [rowIndex] [columnIndex]

Example: matrix_A [0][1] = 10 ;

Program: Write a c program to perform matrix addition?

```
#include <stdlib.h>
#include <stdio.h>
    int main()
  {

  int a[ 2 ][ 3 ] = { { 5, 6, 7 }, { 10, 20, 30 } };
  int b[ 2 ][ 3 ] = { { 1, 2, 3 }, {  3,  2,  1 } };
  int sum[ 2 ][ 3 ], row, column;

     for( i = 0; i < 2; i++ )
   {  for( j= 0; j< 3; j++ )
       sum[i][j] =  a[i][j]+b[i][j];
```

```
      }

    printf( "The sum is: \n\n" );

    for( i = 0; i < 2; i++ )
    {   for( j= 0; j< 3; j++ )
          printf( "\t%d",a[i][j] );
       printf( '\n' );
    }

    return 0;
  }
```

Advantages:

- It is used to represent multiple data items of same type by using only single name.
- It can be used to implement other data structures like linked lists, stacks, queues, trees, graphs etc.
- It allows to store the elements in any dimensional array - supports multidimensional array like 2D arrays are used to represent matrices.
- Iterating the arrays using their index is faster compared to any other methods like linked list etc.
- Arrays are well known in applications such as searching, sorting and matrix operations.

Disadvantages:

- It allows us to enter only fixed number of elements into it. We cannot alter the size of the array once array is declared. We must know in advance(compile time) that how many elements are to be stored in array.
- The elements of array are stored in consecutive memory locations. So insertions and deletions are very difficult and time consuming.

Applications of Arrays:

In programming languages arrays are used in wide range of applications. Few of them are as follows...

- Arrays are used to Store List of values.
- Arrays are used to Perform Matrix Operations like matrix addition, subtraction, multiplication, transpose.
- Arrays are used to implement Search Algorithms like linear search, binary search.

- Arrays are used to implement Sorting Algorithms like bubble sort, selection sort, insertion sort, quick sort.
- Arrays are used to implement Data structures like stack, queue.
- Arrays are also used to implement CPU Scheduling Algorithms.
- Arrays are efficient for sparse matrix representation to save memory

Operations on Array Data Structures:

The basic operations that are performed on data structures are as follows:

- **Insertion:** Insertion means addition of a new data element in a data structure.
- **Deletion:** Deletion means removal of a data element from a data structure if it is found.
- **Searching:** Searching involves searching for the specified data element in a data structure.
- **Traversal:** Traversal of a data structure means processing all the data elements present in it.
- **Sorting:** Arranging data elements of a data structure in a specified order is called sorting
- **Merging:** Combining elements of two similar data structures to form a new data structure of the same type, is called merging.
- **Copying :** Copying array elements to another array will yield an array of the same length and elements as the original one.

Q: Write a c Program to perform Array operations?

```
#include<stdio.h>
#include<stdlib.h>
int arr[25],n,i;
void insert(int ele,int loc);
void delete(int loc);
void display();
void main()
{
   int ch,ele,loc;
   printf("enter no of elements");
   scanf("%d",&n);
   printf("enter elements into array");
   for(i=0;i<n;i++)
   scanf("%d",&arr[i]);
   while(1)
   {
        printf("menu\n1.insert\n2.delete\n3.display\n4.exit\n");
        printf("enter your choice");
        scanf("%d",&ch);
        switch(ch)
        {
           case 1: printf("enter element and location");
                  scanf("%d%d",&ele,&loc);
                  insert(ele,loc);
                  break;
           case 2:printf("enter location to delete");
                  scanf("%d",&loc);
                  delete(loc);
                  break;
           case 3: display();
                  break;
           case 4: exit(0);

        }
     }
}
void insert(int ele,int loc)
{
   for(int i=n-1;i>=loc;i--)
   {
      arr[i+1]=arr[i];
   }
   arr[loc]=ele;
   n=n+1;
}

void delete(int loc)
{
   printf("deleted element is %d",arr[loc]);
   for(i=loc;i<n;i++)
   {
        arr[i]=arr[i+1];
    }
    n=n-1;
}
void display()
{
    printf("elements in the array is :\n");
    for(i=0;i<n;i++)
    {
        printf("%d\n",arr[i]);
    }
}
```

Q: Explain about Pointers?

The pointer is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

Consider the following example to define a pointer which stores the address of an integer.

```
int n=10;
int *p=&n;
```

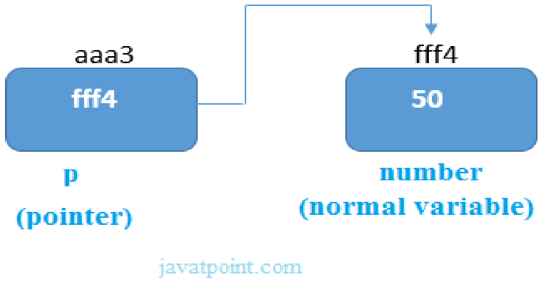
Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer. We use the following syntax to declare a pointer variable.

Syntax: datatype *pointerName

```
int *a;//pointer to int
char *c;//pointer to char
```

Pointer Example



An example of using pointers to print the address and value is given below.

As you can see in the above figure, pointer variable stores the address of number variable, i.e., fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of * (**indirection operator**), we can print the value of pointer variable p.

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

```
int main(){
int number=50;
int *p;
p=&number;//stores the address of number variable
printf("Address of p variable is %x \n",p); // p contains the address of the number therefore printing p gives the address of number.
printf("Value of p variable is %d \n",*p); // As we know that * is used to dereference a pointer therefore if we print *p, we will get the value stored at the address contained by p.
return 0;
}
```

Output

```
Address of number variable is fff4

Address of p variable is fff4

Value of p variable is 50
```

Pointer to an array

We can also point the whole array using pointers.

Using the array pointer, we can easily manipulate the multi-dimensional array.

Example

```
int arr[5] = {10, 20, 30, 40, 50};
int  *ptr[5];
ptr = &arr;
```

Where, ptr points the entire array.

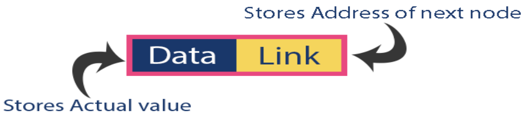
Advantages of pointers:

- Pointers are more efficient inhandling Arrays and Structures.
- Pointers allow references to function and thereby helps in passing of function as arguments to other functions.
- It reduces length of the program and its execution time as well.
- It allows C language to support Dynamic Memory management

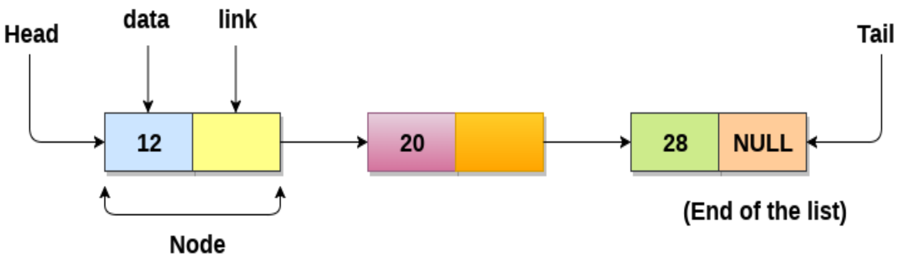
Q: Explain about Linked list and its types?

Linked List

Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.Each element in a linked list is called as "Node".



- A node contains two fields 1. Data Part 2. Address Part
- **DataPart**:. Data value stored at that particular address
- **Address Part:** It is a pointer which contains the address of the next node in the memory.
- The last node of the address part of the list contains null.



Types of Linked lists:

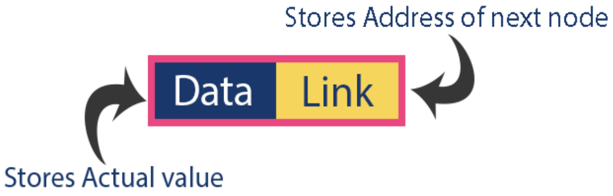
- Singly linked list(single linked list)
- Doubly Linked list (Double linked List)
- Circular linked list.

Singly Linked List:

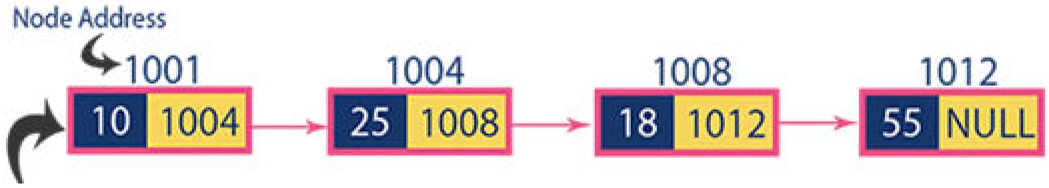
Single linked list is a sequence of elements in which every element has link to its next element in the sequence.

In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data and link field. The data field is used to store actual value of that node and link field is used to store the address of the next node in the sequence.

The graphical representation of a node in a single linked list is as follows...



Example



In a single linked list, the address of the first node is always stored in a reference node known as "head" , last node is known as "tail" node and its link field must be NULL.

Operations

In a single linked list we perform the following operations...

- Insertion
- Deletion
- Display

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

The node can be inserted in three ways

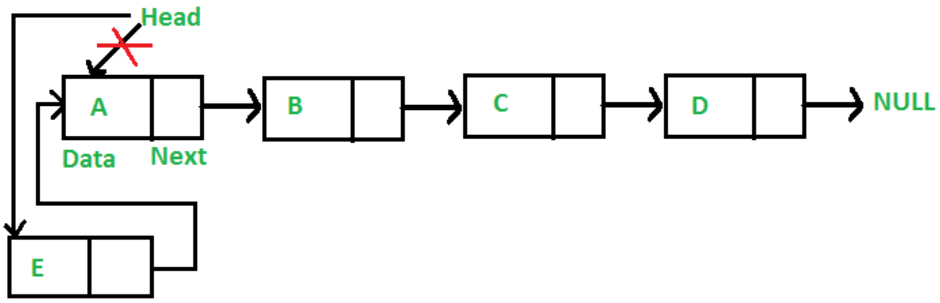
1. **Insert at the beginning**
2. **Insert at the End**
3. **Insert at the Middle**

1)Inserting a node at begin:

It involves inserting any element at the front of the list. The steps are as follows

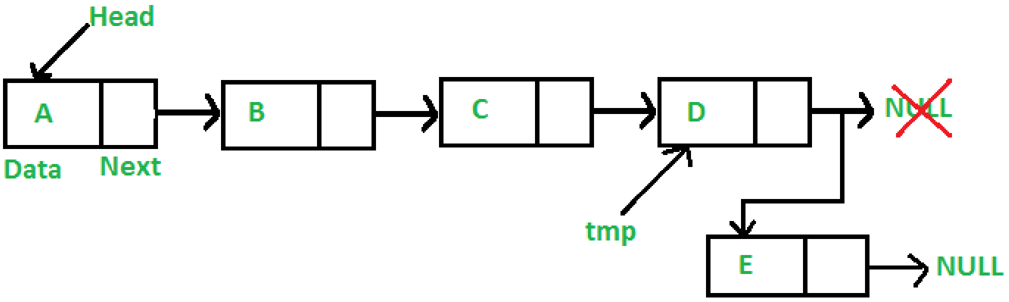
Insert at the beginning

- Allocate memory for new node
- Store data
- Change next of new node to point to head
- Change head to point to recently created node



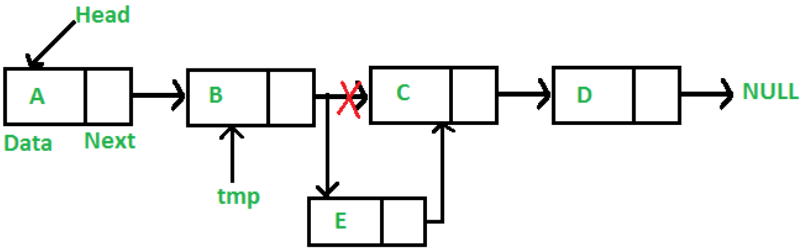
Insert at the End

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node



Insert at the Middle

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between



Node Creation:

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

Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

The Deletion of the node can be done in 3 ways.

1)Deletion at beginning

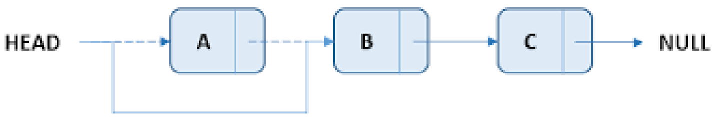
2) Deletion at ending position

3) Deletion at middle postion

1)Deletion at beginning:

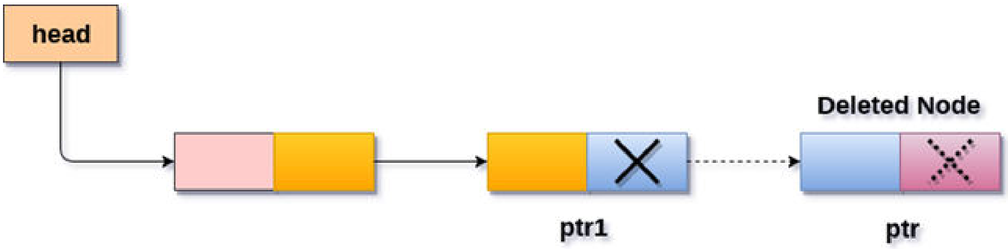
It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers

head = head->next;



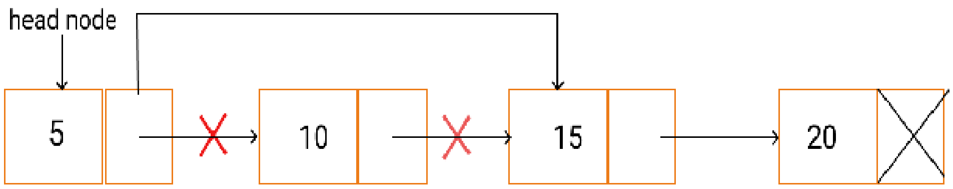
2) Deletion at ending position

- To delete the last node, start traversing the list from the head node and continue traversing until the address part of the node is **NULL.**
- Keep track of the **second last node** in some temporary variable say **prev_node**.
- Once the address part of the node is NULL, set the **address part** of the **prev_node** as **NULL** and then delete the last node.



3)Deletion at specified Position:

- Ask the node position which has to be deleted.
- Start traversing the list from the head node and move up to that **position.**
- While traversing, keep track of the previous node to the node to be deleted.
- Once you reach the position make the previous node link next node link of position node



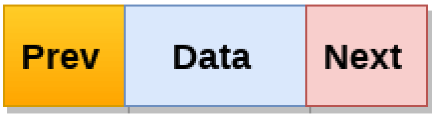
Program: Write a c program to implement linked list operations

```
#include <stdio.h>
            #include <stdlib.h>
            Struct  node
            {         int data;
            struct node *next;
            };
            void create(int data);
            void display();
            struct node *head, *tail = NULL;
            void main()
            {       create(10);create(20);create(30);create(40);
            display();
            }
            void create(int data)
            {
            struct node *newNode = (struct node*)malloc(sizeof(struct node));
            newNode->data = data;
            newNode->next = NULL;
            if(head == NULL) {
            head =tail= newNode;
            }
            else {
            tail->next = newNode;
            tail = newNode;   }
            }
```

```
void display() {
struct node *temp = head;
if(head == NULL) {
printf("List is empty\n");
return;
}
printf("Nodes of singly linked list: \n");
while(temp != NULL) {
printf("%d ",temp->data);
temp = temp->next;
}
printf("\n");
}
```

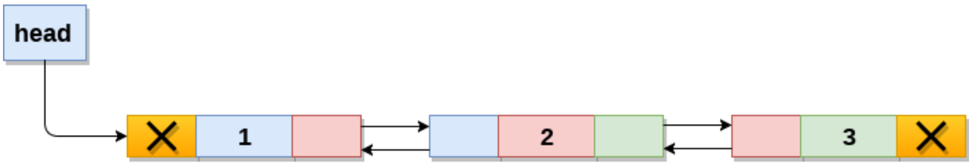
Doubly linked list :

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



Node

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



Doubly Linked List

In C, structure of a node in doubly linked list can be given as :

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

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes.

However, **doubly linked list** overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Node Creation

```
struct node

{

   struct node *prev;

   int data;

   struct node *next;

};
```

All the remaining operations regarding doubly linked list are described in the following table.

Operation	Description
Inseinsertion at beginning	Adding the node into the linked list at beginning.
Inseinsertion at end	Adding the node into the linked list to the end.
Inseinrtion after specified noded	Ading the node into the linked list after the specified node.
Deletion at beginning	Removing the node from beginning of the list
Deletion at the end	Removing the node from end of the list.
letion of the node having given data	Removing the node which is present just after the node containing the given data.
Searching	Comparing each node data with the item to be searched and urn the location of the item in the list if the item found else return null.
Traversing	V Visiting each node of the list at least once in order to rperform some specific operation like searching, sorting, display,

etc.

Algorithm To create and Insert an element to the Double Lined List:

- Define a Node which represents a node in the list. It will have three properties: data, previous which will point to the previous node and next which will point to the next node.
- Create a doubly linked list, and it has two nodes: head and tail. Initially, head and tail will point to null.

Create() will add node to the list:

1. It first checks whether the head is null, then it will insert the node as the head.
2. Both head and tail will point to a newly added node.
3. Head's previous pointer will point to null and tail's next pointer will point to null.
4. If the head is not null, the new node will be inserted at the end of the list such that new node's previous pointer will point to tail.
5. **The new node will become the new tail. Tail's next pointer will point to null.**

display() will show all the nodes present in the list.

- Define a new node 'current' that will point to the head.
- Print current.data till current points to null.
- Current will point to the next node in the list in each iteration.

Q: Write a program to implement Double Linked List

```
#include<stdlib.h>

#include<stdio.h>

void create();

void ftraverse();

void rtraverse();

struct dnode

{

        struct dnode *prev;

        int data;

        struct dnode *next;

};
```

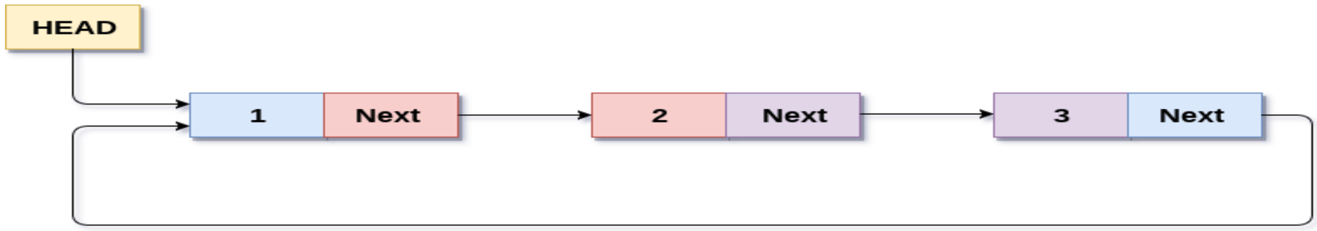
```
struct dnode *first,*last;

void main()

{
        create();

        create();

        create();

        printf("\nForward traversing =");

        ftraverse();

        printf("\nReverse traversing =");

        rtraverse();

}

void create()

{

        struct dnode *temp,*new;

        new=(struct dnode*)malloc(sizeof(struct dnode));

        printf("Enter the data =");

        scanf("%d",&new->data);

        new->prev=NULL;

        new->next=NULL;

        if(first==NULL)

        {
                first=last=new;
          }

        else

        {

                last->next=new;

                new->prev=last;

                last=new;

        }}
```

```
void ftraverse()
{

        struct dnode   *temp=first;

        while(temp!=NULL)

        {

                printf("%d\t",temp->data);

                temp=temp->next;

        }

}


void rtraverse()

{

        struct dnode   *temp=last;

        while(temp!=NULL)

        {

                printf("%d\t",temp->data);

                temp=temp->prev;

        }

}
```

Circular Singly Linked List

- ***Circular linked list*** *is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.*
- In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list.
- We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending.
- There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



Circular Singly Linked List

Example: circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Operations on Circular Singly linked list:

Operation	Description
Insertion at beginning	Adding a node into circular singly linked list at the beginning.
Insertion at the end	Adding a node into circular singly linked list at the end.
Deletion at beginning	Removing the node from circular singly linked list at the beginning.
Deletion at the end	Re moving the node from circular singly linked list at the end.
Searching	compare each element of the node with the given item and ret the location at which the item is present in the list otherwise return null.
Traversing	visiting each element of the list at least once in order to perform some specific operation.

Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.
- We can store values of primitive types or objects in the singly linked list

Applications of linked list in computer science –

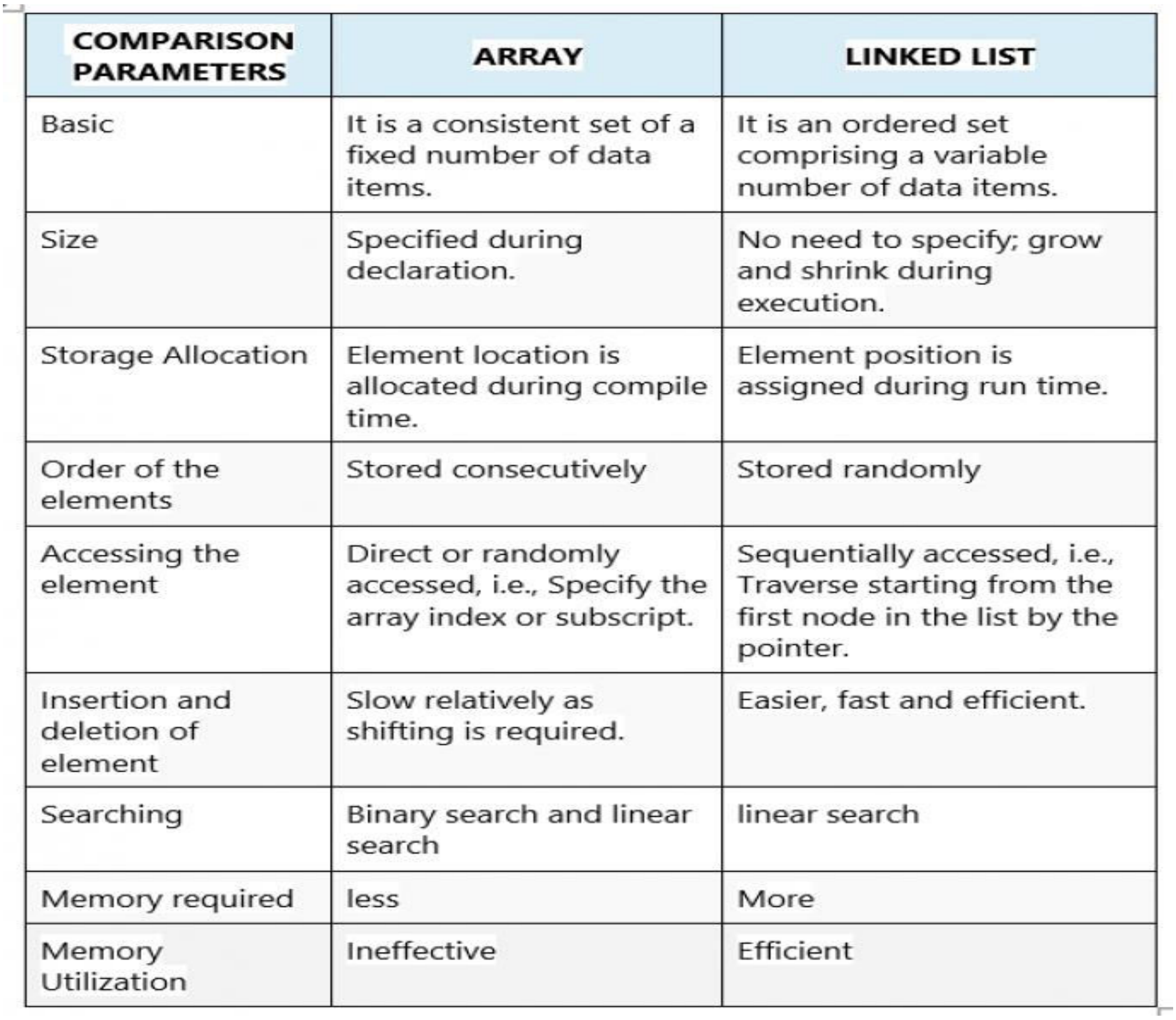
- Implementation of stacks and queues can be done

- Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
- Dynamic memory allocation : We use linked list of free blocks.
- Maintaining directory of names
- Performing arithmetic operations on long integers
- representing sparse matrices

Applications of linked list in real world-

- **Image viewer –** Previous and next images are linked, hence can be accessed by next and previous button.
- **Previous and next page in web browser –** We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
- **Music Player –** Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

Q: Differences between linked list and arrays?



COMPARISON PARAMETERS	ARRAY	LINKED LIST
Basic	It is a consistent set of a fixed number of data items.	It is an ordered set comprising a variable number of data items.
Size	Specified during declaration.	No need to specify; grow and shrink during execution.
Storage Allocation	Element location is allocated during compile time.	Element position is assigned during run time.
Order of the elements	Stored consecutively	Stored randomly
Accessing the element	Direct or randomly accessed, i.e., Specify the array index or subscript.	Sequentially accessed, i.e., Traverse starting from the first node in the list by the pointer.
Insertion and deletion of element	Slow relatively as shifting is required.	Easier, fast and efficient.
Searching	Binary search and linear search	linear search
Memory required	less	More
Memory Utilization	Ineffective	Efficient

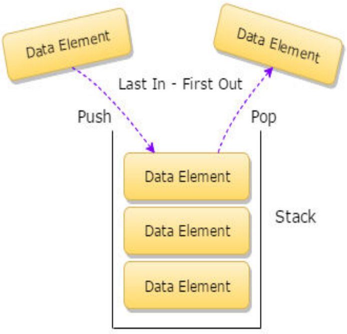
UNIT – III

Stacks: Introduction to Stacks, Stack as an Abstract Data Type, Representation of Stacks through Arrays, Representation of Stacks through Linked Lists, Applications of Stacks, Stacks and Recursion

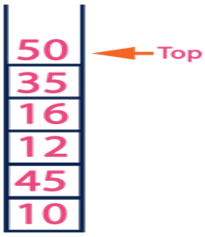
Queues: Introduction, Queue as an Abstract data Type, Representation of Queues, Circular Queues, Double Ended Queues- Deques, Priority Queues, Application of Queues

Q: Explain about Stack?

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack adding and removing of elements are performed at single position which is known as "top". That means, new element is added at top of the stack and an element is removed from the top of the stack only. In stack, the insertion and deletion operations are performed based on LIFO (Last In First Out) principle. The first element which is inserted into stack is deleted last the last element which is inserted into stack is deleted first.

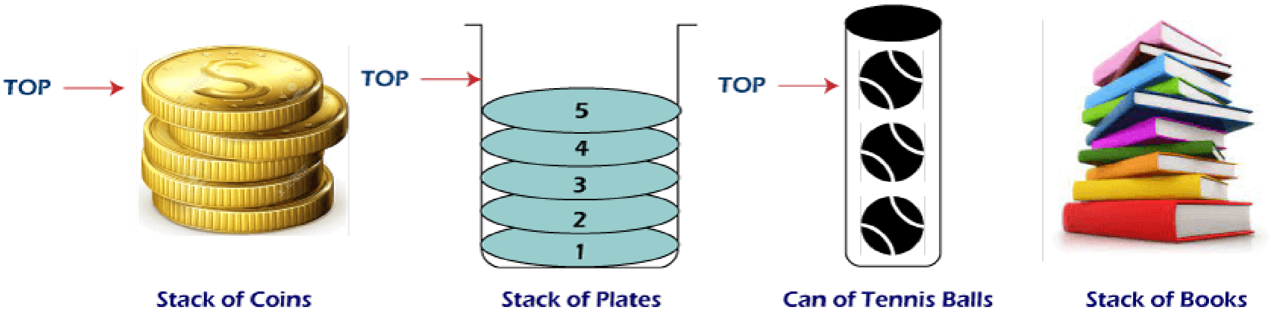


In a stack, the insertion operation is performed using a function called "push" and deletion operation is performed using a function called "pop". In the figure, PUSH and POP operations are performed at top position in the stack. That means, both the insertion and deletion operations are performed at one end i.e., at Top.



Example:

If we want to create a stack by inserting 10,45,12,16,35 and 50. Then 10 becomes the bottom most element and 50 is the top most element. Top is at 50 as shown in the image below.



Stack ADT: An abstract data type is a data type defined in terms of a type and a set of operations on that type. Each operation is defined in terms of its input and output without specifying how the data type is implemented. A stack ADT, defined in terms of push and pop operations.

The Stack ADT Definition A stack is a restricted list in which entries are added and removed from the same end, called the top. This strategy is known as last-in-first-out (LIFO) strategy. Operations (methods) on stacks:

push (item): Insert element into stack

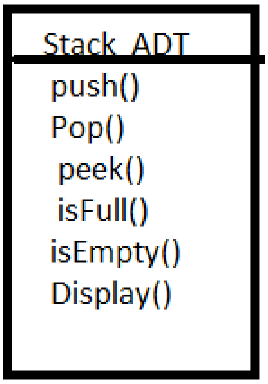
pop () : Removes an element from stack

Peek () : Returns the top element from the stack

Is Empty () : Returns true if the stack is empty

is Full() : Returns true if the stack is full

display : Used to display all the elements



Implementation of the Stack ADT.

The following operations are performed on the stack.

a) Push operation (To insert an element on to the stack)
b) Pop operation (To delete an element from the stack)
c) Display operation (To display elements of the stack)

Stack data structure can be represented in two ways. They are as follows.

1. Using Array
2. Using Linked List

When stack is implemented using array, that stack can organize only limited number of elements. When stack is implemented using linked list, that stack can organize unlimited number of elements.

1. Stack Using Array

A stack data structure can be implemented using a one-dimensional array. But stack implemented using array stores only a fixed number of data values. This implementation is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable called **'top'**. Initially, the top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

Stack Operations using Array

A stack can be implemented using array as follows...
Before implementing actual operations, first follow the below steps to create an empty stack.

Step 1 - Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.

Step 2 - Declare all the **functions** used in stack implementation.

Step 3 - Create a one dimensional array with fixed size (**int stack[SIZE]**)

Step 4 - Define a integer variable **'top'** and initialize with **'-1'**. (**int top = -1**)

Step 5 - In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at **top** position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

Step 1 - Check whether **stack** is **FULL**. (**top == SIZE-1**)

Step 2 - If it is **FULL**, then display **"Stack is FULL!!! Insertion is not possible!!!"** and terminate the function.

Step 3 - If it is **NOT FULL**, then increment **top** value by one (**top++**) and set stack[top] to value (**stack[top] = value**).

pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

Step 1 - Check whether **stack** is **EMPTY**. (**top == -1**)

Step 2 - If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top--**).

display() - Displays the elements of a Stack

We can use the following steps to display the elements of a stack...

Step 1 - Check whether **stack** is **EMPTY**. (**top == -1**)

Step 2 - If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top. Display **stack[i]** value and decrement **i** value by one (**i--**).

Step 3 - Repeat above step until **i** value becomes '0'.

Implementation of Stack using Array

```
#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#define SIZE 10

void push(int);

void pop();

void display();

int stack[SIZE], top = -1;

void main()

{

  int value, choice;

  clrscr();

  while(1){

    printf("\n\n***** MENU *****\n");

    printf("1. Push\n2. Pop\n3. Display\n4. Exit");

    printf("\nEnter your choice: ");

    scanf("%d",&choice);

    switch(choice){

                case 1: printf("Enter the value to be insert: ");
```

```
                          scanf("%d",&value);
                          push(value);
                          break;
                case 2: pop();
                          break;
                case 3: display();
                          break;
                case 4: exit(0);
                default: printf("\nWrong selection!!! Try again!!!");
    }
  }
}
void push(int value){
  if(top == SIZE-1)
    printf("\nStack is Full!!! Insertion is not possible!!!");
  else{
    top++;
    stack[top] = value;
    printf("\nInsertion success!!!");
  }
}
void pop(){
  if(top == -1)
    printf("\nStack is Empty!!! Deletion is not possible!!!");
  else{
    printf("\nDeleted : %d", stack[top]);
    top--;
  }
}
void display(){
  if(top == -1)
    printf("\nStack is Empty!!!");
  else{
    int i;
    printf("\nStack elements are:\n");
    for(i=top; i>=0; i--)
                printf("%d\n",stack[i]);
  }
```

}

Stack Using Linked List

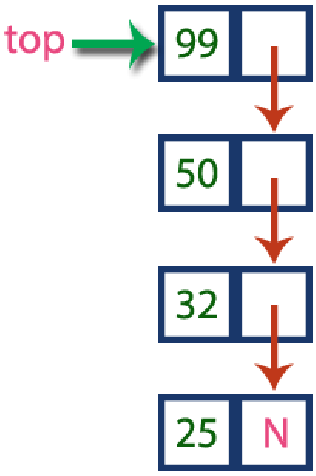
The major problem with the stack implemented using an array is, it works only for a fixed number of data values. That is the amount of data must be specified at the beginning of the implementation itself. Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use.

A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.

Example

In the above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.



Stack Operations using Linked List

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

Step 1 - Include all the **header files** which are used in the program. And declare all the **user defined functions**.

Step 2 - Define a '**Node**' structure with two members **data** and **next**.

Step 3 - Define a **Node** pointer '**top**' and set it to **NULL**.

Step 4 - Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether stack is **Empty** (**top** == **NULL**)

Step 3 - If it is **Empty**, then set **newNode** → **next** = **NULL**.

Step 4 - If it is **Not Empty**, then set **newNode** → **next** = **top**.

Step 5 - Finally, set **top** = **newNode**.

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

Step 1 - Check whether **stack** is **Empty** (**top == NULL**).

Step 2 - If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function

Step 3 - If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.

Step 4 - Then set '**top** = **top** → **next**'.

Step 5 - Finally, delete '**temp**'. (**free(temp)**).

display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

Step 1 - Check whether stack is **Empty** (**top** == **NULL**).

Step 2 - If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.

Step 3 - If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.

Step 4 - Display '**temp** → **data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp** → **next** != **NULL**).

Step 5 - Finally! Display '**temp** → **data** ---> **NULL**'.

Implementation of Stack using Linked List | C Programming

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct Node
{
  int data;
  struct Node *next;
}*top = NULL;

void push(int);
void pop();
void display();
void main()
{
  int choice, value;
  clrscr();
  printf("\n:: Stack using Linked List ::\n");
  while(1){
    printf("\n****** MENU ******\n");
    printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
                case 1: printf("Enter the value to be insert: ");
                          scanf("%d", &value);
                          push(value);
                          break;
                case 2: pop(); break;
                case 3: display(); break;
                case 4: exit(0);
                default: printf("\nWrong selection!!! Please try again!!!\n");
    }
  }
}


void push(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  if(top == NULL)
    newNode->next = NULL;
  else
    newNode->next = top;
  top = newNode;
  printf("\nInsertion is Success!!!\n");
}
```

```
void pop()
{
  if(top == NULL)
    printf("\nStack is Empty!!!\n");
  else{
    struct Node *temp = top;
    printf("\nDeleted element: %d", temp->data);
    top = temp->next;
    free(temp);
  }
}
void display()
{
  if(top == NULL)
    printf("\nStack is Empty!!!\n");
  else{
    struct Node *temp = top;
    while(temp->next != NULL){
            printf("%d--->",temp->data);
            temp = temp -> next;
    }
    printf("%d--->NULL",temp->data);
  }
}
```

Applications of Stack

Evaluation of Arithmetic Expressions

A stack is a very effective data structure for evaluating arithmetic expressions in programming languages. An arithmetic expression consists of operands and operators.

Expression Conversion

An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.

Syntax Parsing

Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

Parenthesis Checking

Stack is used to check the proper opening and closing of parenthesis({})

String Reversal

Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.

Managing Function calls and Return mechanism

Every time a function is called it is pushed back into the stack along with the return address. When function is returned, it is first popped out of the stack and then the control is transferred to the return address (from where the function was called).

Backtracking

Backtracking is another application of Stack. It is a recursive algorithm that is used for solving the optimization problem N Queens problem. Backtracking means we will perform a safe move for a queen at the time we make the move.

Matching HTML Tags in Web Developing. To build web pages we use several HTML tags.so we need to take care of opening and closing tags.

Memory Management: The operating system allocates memory segment for stack which includes arguments, the return value and local variable of a function. Stack variable will exist only until the function created them is running and after that they are all popped out of the stack and data is lost forever.

Undo and Redo Mechanism

This mechanism is also performed with the help of stack.So stack stores thehistory of user mechanisms, and if a new action is performed it is added to thetop of the stack. So an undo mechanism does nothing but pops the item from the stack.

Q: Explain about Expression Conversions

Expression Conversions:

An expression is a collection of operators and operands that represents a specific value. The operator is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc., Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types:

Based on the operator position, expressions are divided into THREE types. They are as follows...

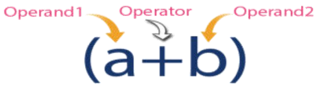
1. Infix Expression
2. Postfix Expression
3. Prefix Expression

Infix Expression:

In infix expression, operator is used in between operands.

The general structure of an Infix expression is as follows...

Example: Operand1 Operator Operand2

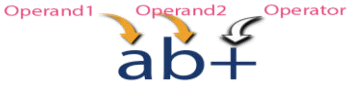


Postfix Expression:

In postfix expression, operator is used after operands. We can say that "Operator follows the Operands".

The general structure of Postfix expression is as follows...

Example: Operand1 Operand2 Operator



Prefix Expression

In prefix expression, operator is used before operands. We can say that "Operands follows the Operator".

The general structure of Prefix expression is as follows...

Example: Operator Operand1 Operand2

Operator
Operand1
Operand2
+ab

Expression Conversion

Any expression can be represented using three types of expressions (Infix, Postfix and Prefix). We can also convert one type of expression to another type of expression like Infix to Postfix, Infix to Prefix, Postfix to Prefix and vice versa.

To convert any Infix expression into Postfix or Prefix expression we can use the following procedure...

- Find all the operators in the given Infix Expression.
- Find the order of operators evaluated according to their Operator precedence.
- Convert each operator into required type of expression (Postfix or Prefix) in the same order.

Operators	**Symbols**
Parenthesis	{ }, ( ), [ ]
Exponential notation	^
Multiplication and Division	*, /
Addition and Subtraction	+, -

Most commonly used operators and their precedence is given in table

Example: The following **Infix Expression to be converted into Postfix Expression**...

D = A + B * C

Step 1: The Operators in the given Infix Expression : = , + , *

Step 2: The Order of Operators according to their preference : * , + , =

Step 3: Now, convert the first operator * ----- D = A + B C *

Step 4: Convert the next operator + ----- D = A BC* +

Step 5: Convert the next operator = ----- D ABC*+ =

Finally, given Infix Expression is converted into Postfix Expression as follows...

D A B C * + =

Infix to Postfix Conversion using Stack Data Structure(Algorithm)

To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps...

Algorithm

Step 1 : Scan the Infix Expression from left to right.

Step 2 : If the scanned character is an operand, append it with final Infix to Postfix string.

Step 3 : Else,

Step 3.1 : If the precedence order of the scanned(incoming) operator is greater than the precedence order of the operator in the stack (or the stack is empty or the stack contains a ‘(‘ or ‘[‘ or ‘{‘), push it on stack.

Step 3.2 : Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

Step 4 : If the scanned character is an '(' or '[' or '{', push it to the stack.

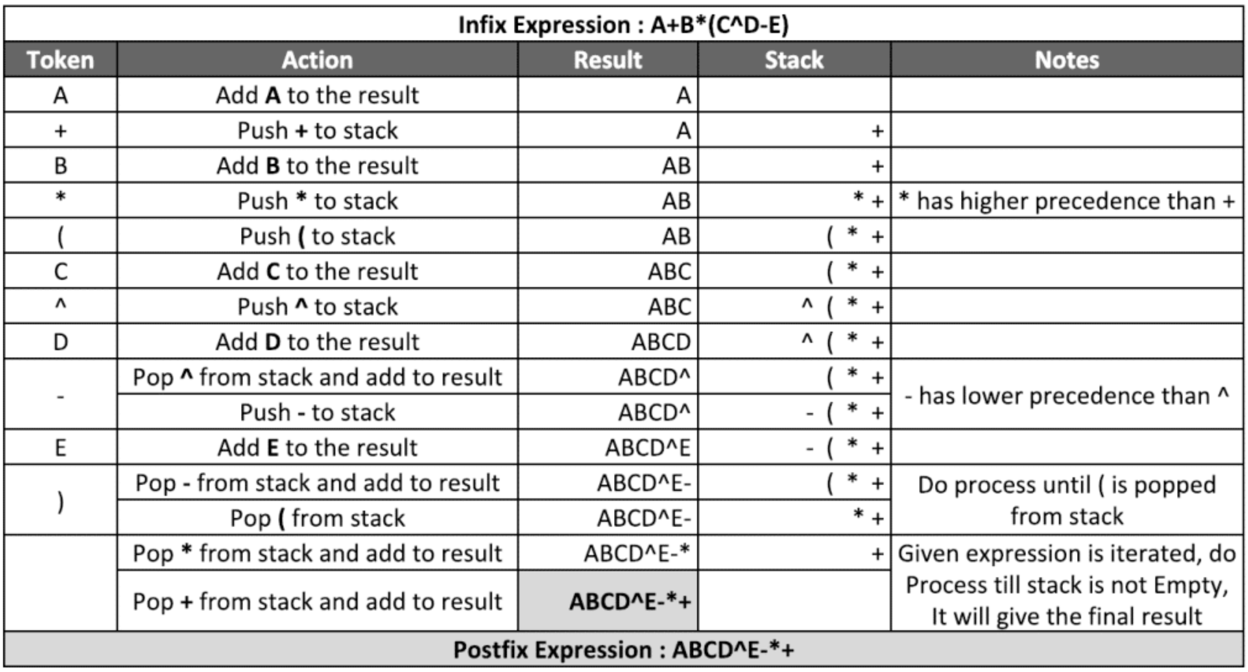
Step 5 : If the scanned character is an ')'or ']' or '}', pop the stack and and output it until a '(' or '[' or '{' respectively is encountered, and discard both the parenthesis.

Step 6 : Repeat steps 2-6 until infix expression is scanned.

Step 7 : Print the output

Step 8 : Pop and output from the stack until it is not empty.

Example: convert infix into postfix



Infix Expression : A+B*(C^D-E)

Token	Action	Result	Stack	Notes
A	Add **A** to the result	A		
+	Push **+** to stack	A	+	
B	Add **B** to the result	AB	+	
*	Push * to stack	AB	* +	* has higher precedence than +
(	Push **(** to stack	AB	(* +	
C	Add **C** to the result	ABC	(* +	
^	Push **^** to stack	ABC	^ (* +	
D	Add **D** to the result	ABCD	^ (* +	
-	Pop **^** from stack and add to result	ABCD^	(* +	- has lower precedence than ^
	Push **-** to stack	ABCD^	- (* +	
E	Add **E** to the result	ABCD^E	- (* +	
)	Pop **-** from stack and add to result	ABCD^E-	(* +	Do process until (is popped from stack
	Pop **(** from stack	ABCD^E-	* +	
	Pop * from stack and add to result	ABCD^E-*	+	Given expression is iterated, do Process till stack is not Empty, It will give the final result
	Pop **+** from stack and add to result	**ABCD^E-*+**		

Postfix Expression : ABCD^E-*+

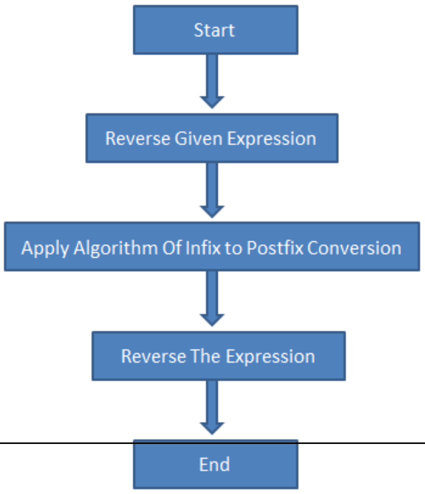
Convert infix to prefix:

Step1: consider the expression

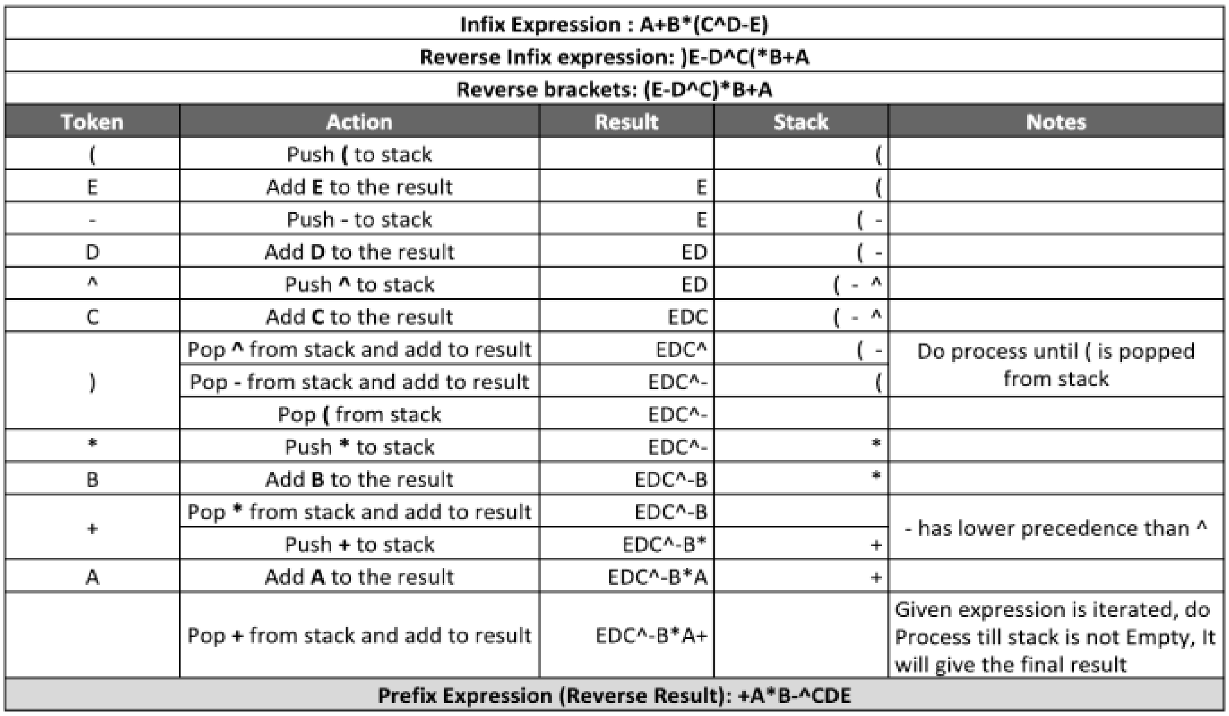
Step2: reverse the given expression

Step3: apply the algorithm for infix to postfix

Step4: apply the reverse to the expression to get the result.



Example: covert infix to prefix



Infix Expression : A+B*(C^D-E)

Reverse Infix expression:)E-D^C(*B+A

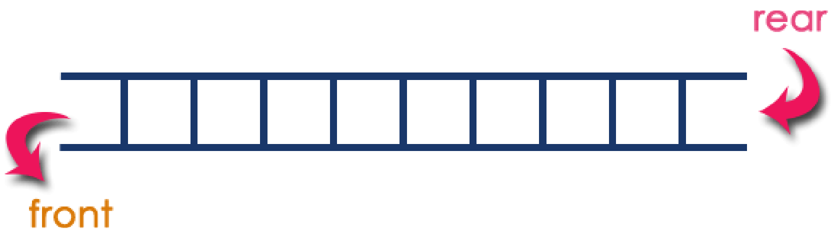
Reverse brackets: (E-D^C)*B+A

Token	Action	Result	Stack	Notes
(	Push **(** to stack		(	
E	Add **E** to the result	E	(	
-	Push **-** to stack	E	(-	
D	Add **D** to the result	ED	(-	
^	Push **^** to stack	ED	(- ^	
C	Add **C** to the result	EDC	(- ^	
)	Pop **^** from stack and add to result	EDC^	(-	Do process until (is popped from stack
	Pop **-** from stack and add to result	EDC^-	(	
	Pop **(** from stack	EDC^-		
*	Push ***** to stack	EDC^-	*	
B	Add **B** to the result	EDC^-B	*	
+	Pop ***** from stack and add to result	EDC^-B		- has lower precedence than ^
	Push **+** to stack	EDC^-B*	+	
A	Add **A** to the result	EDC^-B*A	+	
	Pop **+** from stack and add to result	EDC^-B*A+		Given expression is iterated, do Process till stack is not Empty, It will give the final result

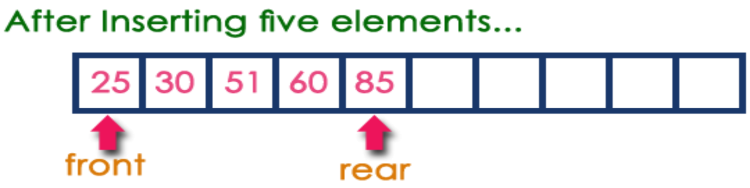
Prefix Expression (Reverse Result): +A*B-^CDE

Q: Explain about Queue Data structure?

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing of elements are performed at two different positions. The insertion is performed at one end and deletion is performed at other end. In a queue data structure, the insertion operation is performed at a position which is known as 'rear' and the deletion operation is performed at a position which is known as 'front'. In queue data structure, the insertion and deletion operations are performed based on FIFO (First In First Out) principle.



In a queue data structure, the insertion operation is performed using a function called "enQueue()" and deletion operation is performed using a function called "deQueue()".
Example: Queue after inserting 25, 30, 51, 60 and 85.



Operations on a Queue:

The following operations are performed on a queue data structure...

enQueue(value) - To insert an element into the queue

deQueue() - To delete an element from the queue

display() - To display the elements of the queue

peek() − Gets the element at the front of the queue without removing it.

isfull() − Checks if the queue is full.

isempty() − Checks if the queue is empty.

Queue data structure can be implemented in two ways. They are as follows...

1. Using Array
2. Using Linked List

When a queue is implemented using array, that queue can organize only limited number of elements. When a queue is implemented using linked list, that queue can organize unlimited number of elements.

1. Queue implementation by Using Array:

A queue data structure can be implemented using one dimensional array. But, queue implemented using array can store only fixed number of data values. The implementation of queue data structure using array is very simple, we define a one dimensional array of specific size and insert or delete the values into that array by using FIFO (First In First Out) principle with the help of variables 'front' and 'rear'.

Initially both 'front' and 'rear' are set to -1. Whenever, we want to insert a new value into the queue, increment 'rear' value by one and then insert at that position. Whenever we want to delete a value from the queue, then increment 'front' value by one and then display the value at 'front' position as deleted element.

Before we implement actual operations, first follow the below steps to create an empty queue.

Step 1: Declare all the user defined functions which are used in queue implementation.

Step 2: Create a one dimensional array with above defined SIZE (int queue[SIZE])

Step 3: Define two integer variables 'front' and 'rear' and initialize both with '-1'(int front = -1,rear = -1)

Step 4: Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

enQueue Operation - Inserting value into the queue:

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

Step 1: Check whether queue is FULL. (rear == SIZE-1)

Step 2: If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.

Step 3: If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value.

deQueue Opeartion- Deleting a value from the Queue

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from front position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

Step 1: Check whether queue is EMPTY. (front == rear)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then increment the front value by one (front ++). Then display queue[front] as deleted element. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

display() - Displays the elements of a Queue:

We can use the following steps to display the elements of a queue...

Step 1: Check whether queue is EMPTY. (front == rear)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front+1'.

Step 4: Display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i' value is equal to rear (i <= rear)

Ex: Program for implementing Queue data structure and perform operations like creation, insert, delete, display using Array Datastructure

```
#include<stdio.h>
#include<conio.h>
#define SIZE 10
void enQueue(int);
void deQueue();
void display();
int queue[SIZE], front = -1, rear = -1;
void main()
{
  int value, choice;
  clrscr();
  while(1){
    printf("\n\n***** MENU *****\n");
    printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&choice);
    switch(choice){
                case 1: printf("Enter the value to be insert: ");
```

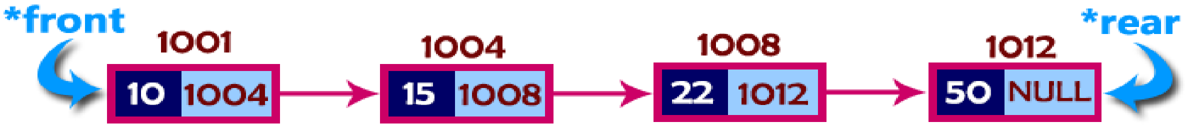
```
                          scanf("%d",&value);
                          enQueue(value);
                          break;
                case 2: deQueue();
                          break;
                case 3: display();
                          break;
                case 4: exit(0);
                default: printf("\nWrong selection!!! Try again!!!");
    }
  }
}
void enQueue(int value){
  if(rear == SIZE-1)
    printf("\nQueue is Full!!! Insertion is not possible!!!");
  else{
    if(front == -1)
                front = 0;
    rear++;
    queue[rear] = value;
    printf("\nInsertion success!!!");
  }
}
void deQueue(){
  if(front == rear)
    printf("\nQueue is Empty!!! Deletion is not possible!!!");
  else{
    printf("\nDeleted : %d", queue[front]);
    front++;
    if(front == rear)
                front = rear = -1;
  }
}
void display(){
  if(rear == -1)
    printf("\nQueue is Empty!!!");
  else{
    int i;
    printf("\nQueue elements are:\n");
    for(i=front; i<=rear; i++)
            printf("%d\t",queue[i]);
  }
}
```

Queue Using Linked List

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use.

A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want. In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

Example



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

Operations

To implement queue using linked list, we need to set the following things before implementing actual operations.

Step 1 - Include all the **header files** which are used in the program. And declare all the **user defined functions**.

Step 2 - Define a '**Node**' structure with two members **data** and **next**.

Step 3 - Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.

Step 4 - Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

Step 1 - Create a **newNode** with given value and set '**newNode → next**' to **NULL**.

Step 2 - Check whether queue is **Empty** (**rear == NULL**)

Step 3 - If it is **Empty** then, set **front = newNode** and **rear = newNode**.

Step 4 - If it is **Not Empty** then, set **rear → next = newNode** and **rear = newNode**.

deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

Step 1 - Check whether **queue** is **Empty** (**front == NULL**).

Step 2 - If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function

Step 3 - If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.

Step 4 - Then set '**front = front → next**' and delete '**temp**' (**free(temp)**).

display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

Step 1 - Check whether queue is **Empty** (**front == NULL**).

Step 2 - If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.

Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.

Step 4 - Display '**temp → data --->**' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp → next != NULL**).

Step 5 - Finally! Display '**temp → data ---> NULL**'.

Implementation of Queue Datastructure using Linked List - C Programming

```
#include<stdio.h>
#include<conio.h>
struct Node
{
   int data;
   struct Node *next;
}*front = NULL,*rear = NULL;

void insert(int);
void delete();
void display();

void main()
{
   int choice, value;
   clrscr();
   printf("\n:: Queue Implementation using Linked List ::\n");
   while(1){
      printf("\n****** MENU ******\n");
      printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d",&choice);
      switch(choice){
```

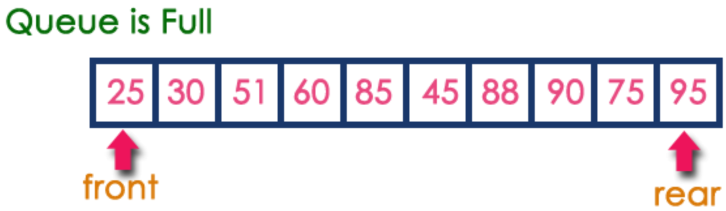
```
                case 1: printf("Enter the value to be insert: ");
                          scanf("%d", &value);
                          insert(value);
                          break;
                case 2: delete(); break;
                case 3: display(); break;
                case 4: exit(0);
                default: printf("\nWrong selection!!! Please try again!!!\n");
    }
  }
}
void insert(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  newNode -> next = NULL;
  if(front == NULL)
    front = rear = newNode;
  else{
    rear -> next = newNode;
    rear = newNode;
  }
  printf("\nInsertion is Success!!!\n");
}
void delete()
{
  if(front == NULL)
    printf("\nQueue is Empty!!!\n");
  else{
    struct Node *temp = front;
    front = front -> next;
    printf("\nDeleted element: %d\n", temp->data);
    free(temp);
  }
}
void display()
{
```

```
if(front == NULL)
  printf("\nQueue is Empty!!!\n");
else{
  struct Node *temp = front;
  while(temp->next != NULL){
            printf("%d--->",temp->data);
            temp = temp -> next;
  }
  printf("%d--->NULL\n",temp->data);
} }
```

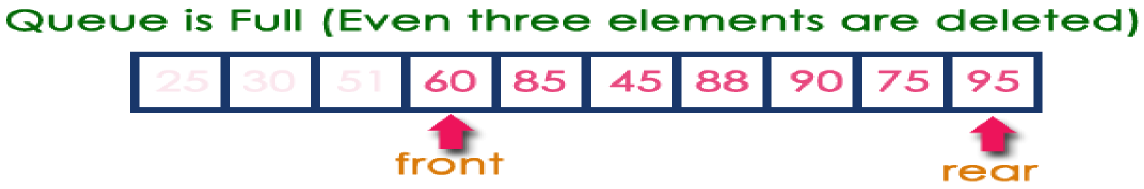
Q: Explain about Circular Queue Datastructure

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once the queue becomes full, we can not insert the next element until all the elements are deleted from the queue.

For example, consider the queue below...
The queue after inserting all the elements into it is as follows...



Now consider the following situation after deleting three elements from the queue...



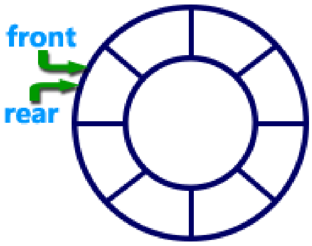
It says that Queue is Full and we cannot insert the new element because '**rear**' is still at last position. In the above situation, even though we have empty positions in the queue we can not make use of them to insert the new element. This is the major problem in a normal queue data structure. To overcome this problem we use a circular queue data structure.

What is Circular Queue?

A Circular Queue can be defined as follows...

A circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

Graphical representation of a circular queue is as follows...



Implementation of Circular Queue

To implement a circular queue data structure using an array, we first perform the following steps before we implement actual operations.

Step 1 - Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.

Step 2 - Declare all **user defined functions** used in circular queue implementation.

Step 3 - Create a one dimensional array with above defined SIZE (**int cQueue[SIZE]**)

Step 4 - Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**. (**int front = -1, rear = -1**)

Step 5 - Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

enQueue(value) - Inserting value into the Circular Queue

In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue. We can use the following steps to insert an element into the circular queue...

Step 1 - Check whether **queue** is FULL. ((**rear == SIZE-1 && front == 0**) || (**front == rear+1**))

Step 2 - If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.

Step 3 - If it is **NOT FULL**, then check **rear == SIZE - 1 && front != 0** if it is **TRUE**, then set **rear = -1**.

Step 4 - Increment **rear** value by one (**rear++**), set **queue[rear]** = **value** and check '**front == -1**' if it is **TRUE**, then set **front = 0**.

deQueue() - Deleting a value from the Circular Queue

In a circular queue, deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from **front** position. The deQueue() function doesn't take any value as a parameter. We can use the following steps to delete an element from the circular queue...

Step 1 - Check whether **queue** is **EMPTY**. (**front == -1 && rear == -1**)

Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then display **queue[front]** as deleted element and increment the **front** value by one (**front ++**). Then check whether **front == SIZE**, if it is **TRUE**, then set **front = 0**. Then check whether both **front - 1** and **rear** are equal (**front -1 == rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front = rear = -1**).

display() - Displays the elements of a Circular Queue

We can use the following steps to display the elements of a circular queue...

Step 1 - Check whether **queue** is **EMPTY**. (**front == -1**)

Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front**'.

Step 4 - Check whether '**front <= rear**', if it is **TRUE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.

Step 5 - If '**front <= rear**' is **FALSE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until'**i <= SIZE - 1**' becomes **FALSE**.

Step 6 - Set **i** to **0**.

Step 7 - Again display '**cQueue[i]**' value and increment **i** value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.

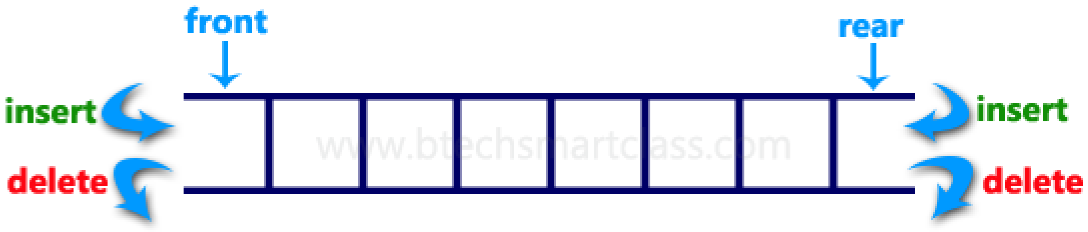
Implementation of Circular Queue Datastructure using array - C Programming

```
#include<stdio.h>
#include<conio.h>
#define SIZE 5
void enQueue(int);
void deQueue();
void display();
int cQueue[SIZE], front = -1, rear = -1;
void main()
{
  int choice, value;
  clrscr();
  while(1){
    printf("\n****** MENU ******\n");
    printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
            case 1: printf("\nEnter the value to be insert:  ");
                    scanf("%d",&value);
                    enQueue(value);
                    break;
            case 2: deQueue();
                    break;
            case 3: display();
                    break;
            case 4: exit(0);
            default: printf("\nPlease select the correct choice!!!\n");
    }
  }
}
void enQueue(int value)
{
  if((front == 0 && rear == SIZE - 1) || (front == rear+1))
    printf("\nCircular Queue is Full! Insertion not possible!!!\n");
  else{
    if(rear == SIZE-1 && front != 0)
            rear = -1;
    cQueue[++rear] = value;
    printf("\nInsertion Success!!!\n");
    if(front == -1)
            front = 0;
  }
}
void deQueue()
{
  if(front == -1 && rear == -1)
    printf("\nCircular Queue is Empty! Deletion is not possible!!!\n");
  else{
    printf("\nDeleted element : %d\n",cQueue[front++]);
    if(front == SIZE)
            front = 0;
    if(front-1 == rear)
            front = rear = -1;
  }
}
void display()
{
  if(front == -1)
    printf("\nCircular Queue is Empty!!!\n");
  else{
    int i = front;
    printf("\nCircular Queue Elements are : \n");
    if(front <= rear){
            while(i <= rear)
              printf("%d\t",cQueue[i++]);
    }
```

```
else{
            while(i <= SIZE - 1)
              printf("%d\t", cQueue[i++]);
            i = 0;
            while(i <= rear)
              printf("%d\t",cQueue[i++]);
} } }
```

Q: Explain about Double Ended Queue Datastructure

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**). That means, we can insert at both front and rear positions and can delete from both front and rear positions.



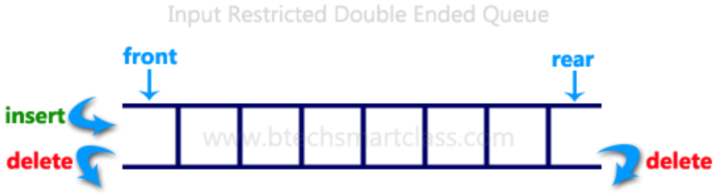
www.btechsmartclass.com

Double Ended Queue can be represented in TWO ways, those are as follows...

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

Input Restricted Double Ended Queue

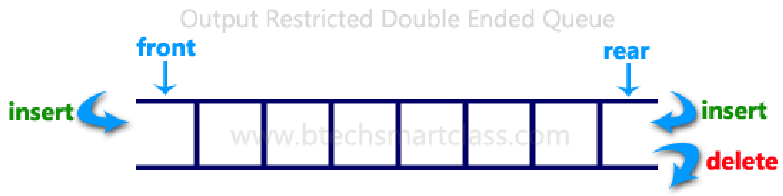
In input restricted double-ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



www.btechsmartclass.com

Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



Implementation of Double Ended Queue Datastructure using double linked list - C Programming

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct dqnode
{
    struct dqnode *prev;
    int data;
    struct dqnode *next;
};
struct dqnode *front,*rear,*temp,*newnode;
int x;
void insertAtFront()
{
    newnode=(struct dqnode*)malloc(sizeof(struct dqnode));
    printf("\nEnter an element to insert At front=");
    scanf("%d",&x);
    newnode->data=x;
    newnode->next=NULL;
    if(front==NULL)
        front=rear=newnode;
    else
    {
        newnode->next=front;
        front->prev=newnode;
        front=newnode;
    }
}
void insertAtRear()
{
    newnode=(struct dqnode*)malloc(sizeof(struct dqnode));
    printf("\nEnter an element to insert At rear=");
    scanf("%d",&x);
    newnode->data=x;
    newnode->next=NULL;
    if(front==NULL)
            front=rear=newnode;
    else
    {
            newnode->prev=rear;
            rear->next=newnode;
            rear=newnode;
    }
}
void display()
{
    temp=front;
    printf("\nElements in the DeQueue =");
    while(temp!=NULL)
    {
            printf("%d\t",temp->data);
            temp=temp->next;
    }

}
void deleteAtFront()
{
    if(front==NULL)
    {
            printf("\nDEQueue is empty");
    }
    else
    {       temp=front;
            x=front->data;
            front=front->next;
            front->prev=NULL;
            printf("\nElement deleted at front =%d",x);
    }
}
```

```
void deleteAtRear()
{
    if(front==NULL)
    {
            printf("\nDEQueue is empty");
    }
    else
    {
            x=rear->data;
            temp=rear;
            rear=rear->prev;
            rear->next=NULL;
            free(temp);
            printf("\nElement deleted at rear =%d",x);
    }
}
void main()
{
clrscr();
insertAtRear();
insertAtRear();
display();
insertAtFront();
display();
deleteAtFront();
display();
deleteAtRear();
display();
getch();
    }
```

Applications of Queue: Applications are

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

2. Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.

3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e. First come first served.

Unit-4

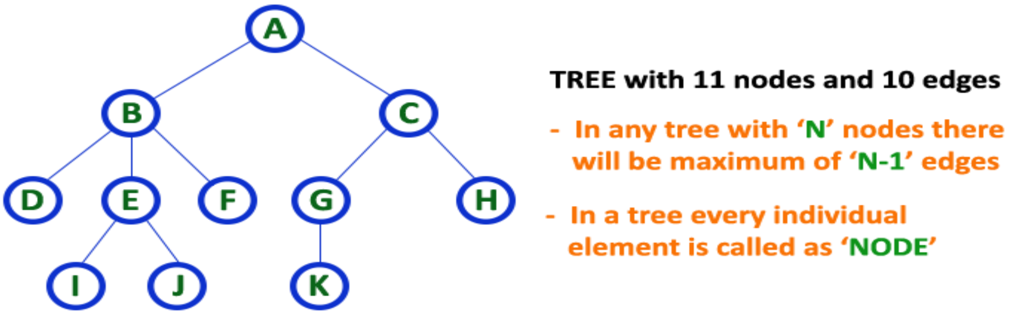
Tree Terminology

In linear data structure, data is organized in sequential order and in non-linear data structure, data is organized in random order. Tree is a very popular data structure used in wide range of applications. A tree data structure can be defined as follows...

Tree is a non-linear data structure which organizes data in hierarchical structure i.e each element is attached to one or more elements directly beneath it..

- In tree data structure, every individual element is called as Node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchicalstructure
- In a tree data structure, if we have N number of nodes then we can have a maximum of N-1 number of links.

Example:



TREE with 11 nodes and 10 edges

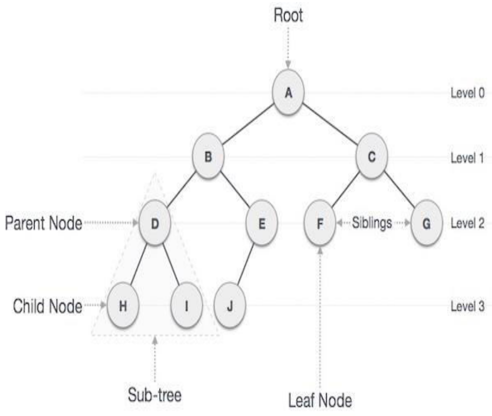
- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

Terminology

In a tree data structure, we use the following terminology...

Root: In a tree data structure, the first node is called as Root Node. Every tree must have root node. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

Edge: In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.



Parent: In a tree data structure, the node which is predecessor of any node is called as PARENT NODE. Parent node can also be defined as "The node which has child / children".

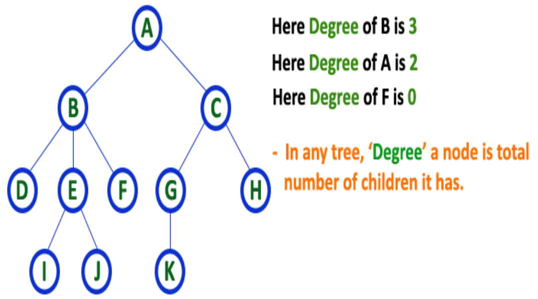
Child: In a tree data structure, the node which is descendant of any node is called as CHILD Node.In a tree, all the nodes except root are child nodes.

Siblings: In a tree data structure, nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with same parent are called as Sibling nodes.

Leaf: In a tree data structure, the node which does not have a child is called as LEAF Node. In simple words, a leaf is a node with no child. Leaf nodes are also called as External Nodes. Leaf node is also called as 'Terminal' node.

Internal Nodes: In a tree data structure, the node which has atleast one child is called as INTERNAL Node.

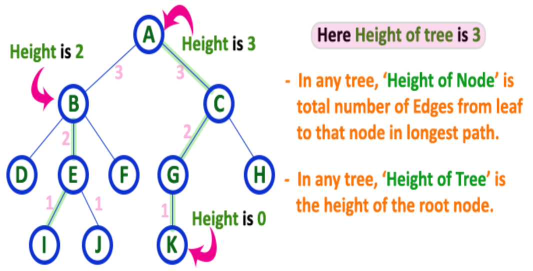
Degree: In a tree data structure, the total number of children of a node is called as DEGREE of that Node



Level: In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on

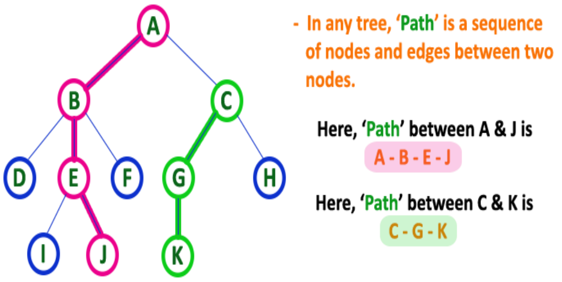
Height:

In a tree data structure, the total number of egdes from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.


Height is 2
Height is 3
Height is 0
A
B
C
D
E
F
G
H
I
J
K
3
3
2
2
1
1
1
Here Height of tree is 3
- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

Path:

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes.Length of a Path is total number of nodes in that path. In below example the path A - B - E - J has length 4.


A
B
C
D
E
F
G
H
I
J
K
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.
Here, 'Path' between A & J is
A - B - E - J
Here, 'Path' between C & K is
C - G - K

Tree Types: Trees can be classified as

1)Binary Tree

2) Full Binary Tree

3) Complete Binary Tree

4) Binary Search Tree

Binary Tree: A binary tree is a tree in which a parent node which consists of maximum 2 childrens. So it supports 0-2 nodes only.

Full Binary Tree: A Binary tree is known as full binary tree in which each parent node except leaf nodes compulsory consists of two childrens.

Complete Binary Tree: A binary tree is a complete binary tree in which all elements forms a sequence.

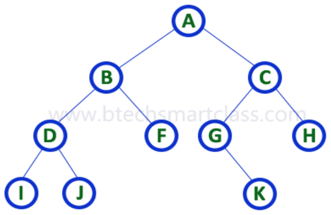
Binary Search Tree: A tree is a binary search tree it must be a binary tree. In binary search tree the left child value always less than parent value and right child value always greater than the parent value.

Binary Tree Representation: A binary tree data structure is represented using two methods. Those methods are as follows...

1)Array Representation

2)Linked List Representation

Consider the following binary tree...



1**. Array Representation/Sequential Representation:**

The simplest way to represent binary tree is array representation .we use a one dimensional rray (1-D Array) to store set of values.

A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

In which

- The parent element of a binary tree is stored in first location of the array.
- If the parent node position is "I" its left child is placed in (2i+1)position and its right child is (2i+2) position.

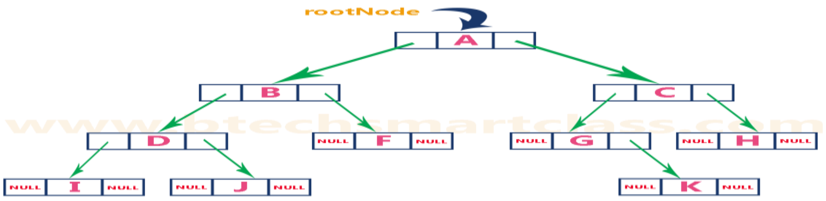
2. Linked List Representation:

We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address

If a node does not have any child then the corresponding pointer field made as null.

Left Child Address	Data	Right Child Address

In this linked list representation, a node has the following structure...



Binary Tree Traversals:

When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called as **Binary Tree Traversal.**

There are three types of binary tree traversals.

In - Order Traversal

Pre - Order Traversal

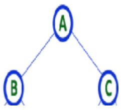
Post - Order Traversal

Consider the following binary tree...

1. In - Order Traversal (visit leftChild - root - rightChild)

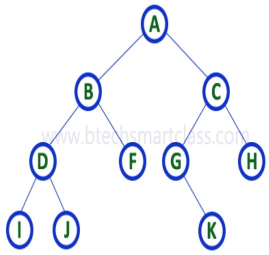
In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree.. This is performed recursively for all nodes in the tree.

In order traversal is: **BAC**



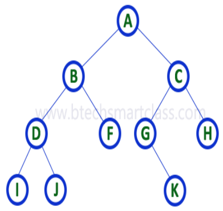
EXAMPLE:(only for understanding .No need to write)

I - D - J - B - F - A - G - K - C - H

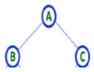


2. Pre - Order Traversal (visit first root - leftChild - rightChild)

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.



Example: Pre order Traversal is: **BAC**



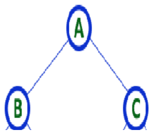
That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is **A - B - D - I - J - F - C - G - K – H**

3. Post - Order Traversal (first visit leftChild - rightChild - root)

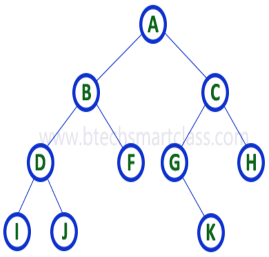
In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Post Order Traversal is: **BCA**



Example:

Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.



Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C – A

Binary Search Tree:

In a binary tree, every node can have maximum of two children but there is no order of nodes based on their values. In binary tree, the elements are arranged as they arrive to the tree, from top to bottom and left to right.

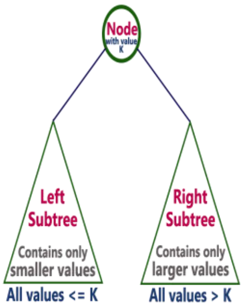
A binary tree has the following time complexities...

Search Operation - O(n)

Insertion Operation - O(1)

Deletion Operation - O(n)

To enhance the performance of binary tree, we use special type of binary tree known as Binary Search Tree. Binary search tree mainly focus on the search operation in binary tree. Binary search tree can be defined as follows...

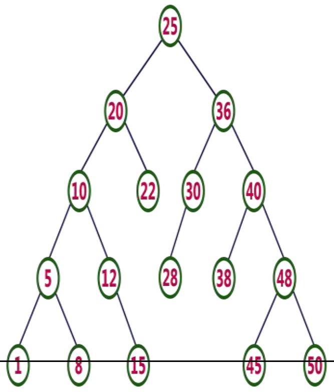


Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

In a binary search tree, all the nodes in left subtree of any node contains smaller values and all the nodes in right subtree of that contains larger values as shown in following figure...

Example:

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



Every Binary Search Tree is a binary tree but all the Binary Trees need not to be binary search trees.

Operations on a Binary Search Tree

The following operations are performed on a Binary search tree...

- Search
- Insertion
- Deletion

Search Operations in BST

In a binary search tree, **the search operation is performed with O(log n) time complexity**. The search operation is performed as follows...

Step 1: Read the search element from the user

Step 2: Compare, the search element with the value of root node in the tree.

Step 3: If both are matching, then display "Given node found!!!" and terminate the function

Step 4: If both are not matching, then check whether search element is smaller or larger than that node value.

Step 5: If search element is smaller, then continue the search process in left subtree.

Step 6: If search element is larger, then continue the search process in right subtree.

Step 7: Repeat the same until we found exact element or we completed with a leaf node

Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.

Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in BST:

In a binary search tree, the **insertion operation is performed with O(log n) time complexity**. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1: Create a newNode with given value and set its left and right to NULL.

Step 2: Check whether tree is Empty.

Step 3: If the tree is Empty, then set set root to newNode.

Step 4: If the tree is Not Empty, then check whether value of newNode is smaller or larger than the node (here it is root node).

Step 5: If newNode is smaller than or equal to the node, then move to its left child. If newNode is larger than the node, then move to its right child.

Step 6: Repeat the above step until we reach to a leaf node (e.i., reach to NULL).

Step 7: After reaching a leaf node, then isert the newNode as left child if newNode is smaller or equal to that leaf else insert it asright child.

Deletion Operation in BST:

In a binary search tree, the deletion operation is performed with O(log n) time complexity. Deleting a node from Binary search tree has follwing three cases...

Case 1: Deleting a Leaf node (A node with no children)

Case 2: Deleting a node with one child

Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

Step 1: Find the node to be deleted using search operation

Step 2: Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

Step 1: Find the node to be deleted using search operation

Step 2: If it has only one child, then create a link between its parent and child nodes.

Step 3: Delete the node using free function and terminate the function.

Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

Step 1: Find the node to be deleted using search operation

Step 2: If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.

Step 3: Swap both deleting node and node which found in above step.

Step 4: Then, check whether deleting node came to case 1 or case 2 else goto steps 2

Step 5: If it comes to case 1, then delete using case 1 logic.

Step 6: If it comes to case 2, then delete using case 2 logic.

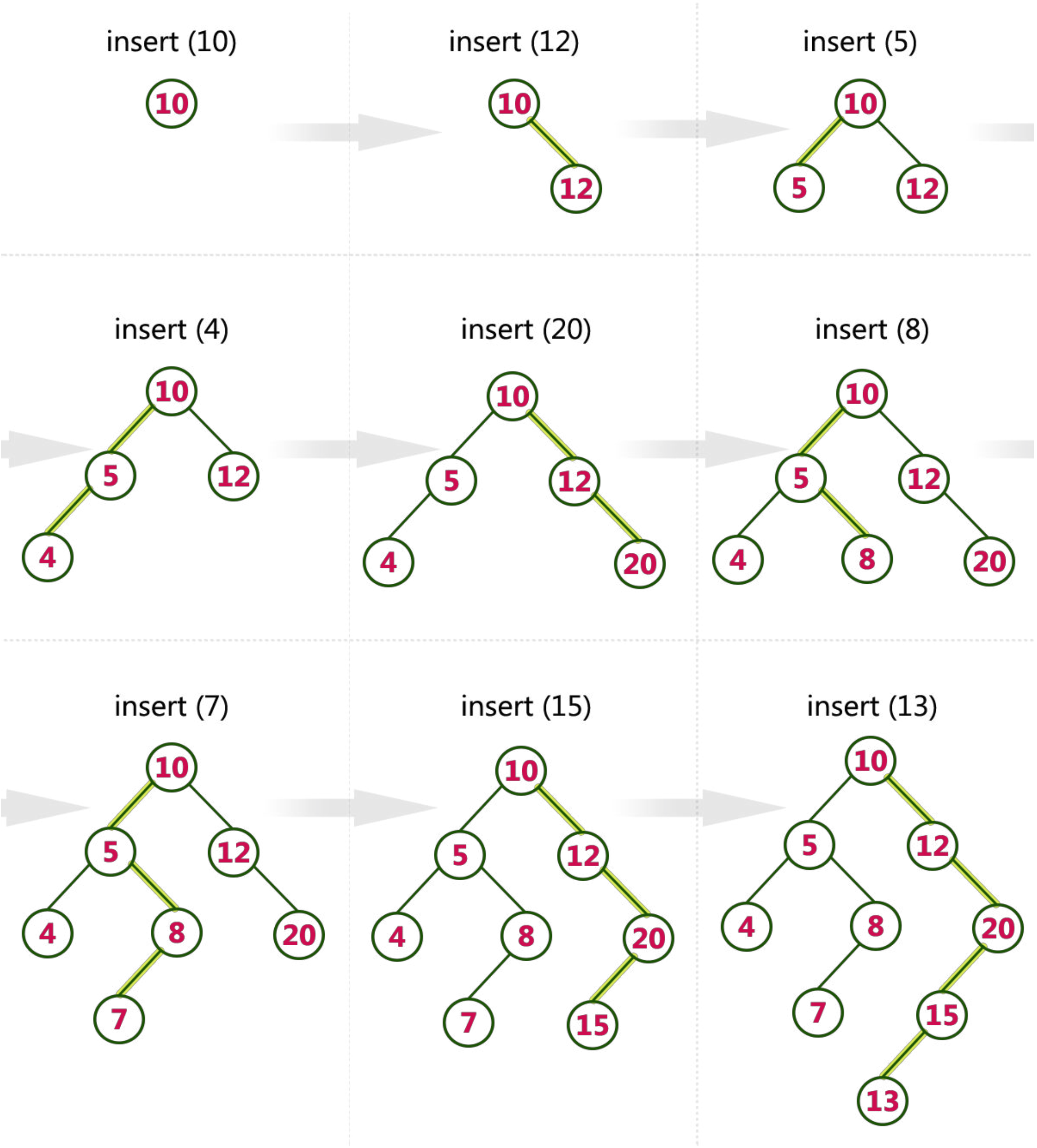
Step 7: Repeat the same process until node is deleted from the tree.

Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13

Above elements are inserted into a Binary Search Tree as follows...



C program for implementing "Binary Search Tree".

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

struct tnode
{
    struct tnode *left;
    int data;
    struct tnode *right;
};

struct tnode *newnode,*root,*temp1,*temp2;
int x;
void create()
{
    printf("\nEnter an element =");
    scanf("%d",&x);
    while(x!=-1)
    {     newnode=(struct tnode*)malloc(sizeof(struct tnode));
          newnode->data=x;
          newnode->left=newnode->right=NULL;
          if(root==NULL)
                    root=newnode;
          else
          {
                    temp1=root;
                    while(temp1!=NULL)
                    {      temp2=temp1;
                           if(temp1->data>x)
                                  temp1=temp1->left;
                           else
                                  temp1=temp1->right;
                    }
                 if(temp2->data>x)
                           temp2->left=newnode;
                 else
                           temp2->right=newnode;
              }
         printf("\nEnter an element =");
         scanf("%d",&x);
    }
}


void inorder(struct tnode *temp)
{
    if(temp==NULL)
           return;
    else
     {
           inorder(temp->left);
```

```
            printf("%d ",temp->data);
            inorder(temp->right);
     }
}
void preorder(struct tnode *temp)
{
    if(temp==NULL)
            return;
    else
     {
            printf("%d ",temp->data);
            preorder(temp->left);
            preorder(temp->right);
     }
}
void postorder(struct tnode *temp)
{
    if(temp==NULL)
            return;
    else
     {
            preorder(temp->left);
            preorder(temp->right);
            printf("%d ",temp->data);
     }

}
void main()
{
    clrscr();
    root=NULL;
    create();
    printf("\ninorder   traversing =");
    inorder(root);
    printf("\npreorder  traversing =");
    preorder(root);
    printf("\npostorder traversing =");
    postorder(root);
    getch();
}
```

UNIT-V

Introduction to Graphs

Graph is a non linear data structure, it contains a set of points known as nodes (or vertices) and set of linkes known as edges (or Arcs) which connets the vertices. A graph is defined as follows...

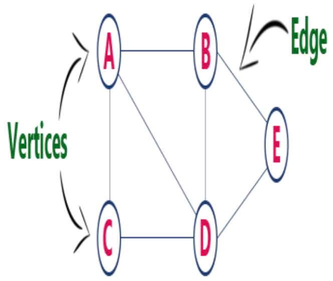
Graph is a collection of vertices and arcs which connects vertices in the graph

Graph is a collection of nodes and edges which connects nodes in the graph

Generally, a graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

Example

The following is a graph with 5 vertices and 6 edges. This graph G can be defined as G = (V , E) Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



Graph Terminology

We use the following terms in graph data structure...

Vertex: A individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

Edge: An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex , ending Vertex).

For example, in above graph, the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

Undirected Edge - An undirected edge is a bidirectional edge. If there is a undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

Directed Edge - A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).

Weighted Edge - A weighted edge is an edge with cost on it.

Undirected Graph: A graph with only undirected edges is said to be undirected graph.

Directed Graph: A graph with only directed edges is said to be directed graph.

Mixed Graph: A graph with undirected and directed edges is said to be mixed graph.

End vertices or Endpoints: The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.

Origin: If an edge is directed, its first endpoint is said to be origin of it.

Destination: If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.

Adjacent: If there is an edge between vertices A and B then both A and B are said to be adjacent.

Outgoing Edge: A directed edge is said to be outgoing edge on its orign vertex.

Incoming Edge: A directed edge is said to be incoming edge on its destination vertex.

Degree: Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree: Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree: Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges: If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

Self-loop: An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph:A graph is said to be simple if there are no parallel and self-loop edges.

Path: A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

Explain about Graph Representations?

Graph data structure is represented using following representations...

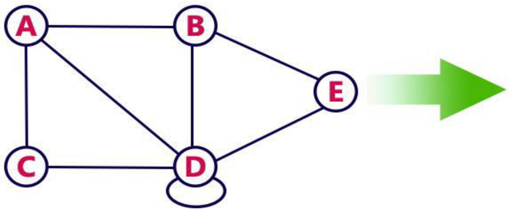
- Adjacency Matrix
- Incidence Matrix
- Adjacency List

Adjacency Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices. That means if a graph with 4 vertices can be represented using a matrix of 4X4 class.

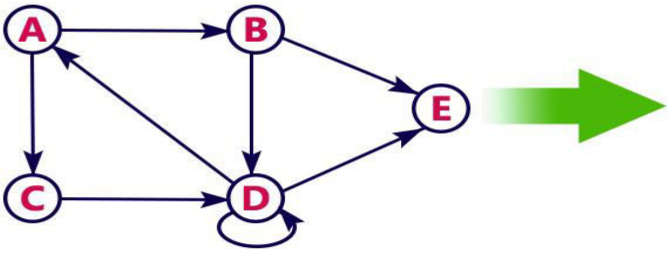
In this matrix, rows and columns both represents vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

Directed graph representation...

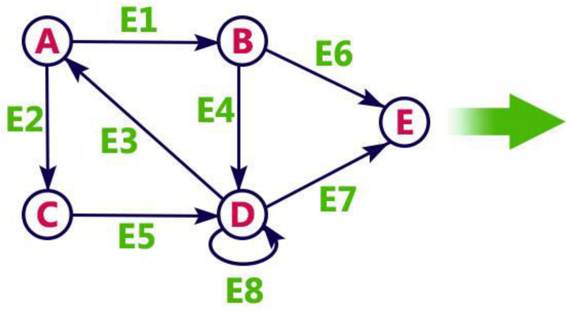


$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

Incidence Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of edges. That means if a graph with 4 vertices and 6 edges can be represented using a matrix of 4X6 class.

In this matrix, rows represents vertices and columns represents edges. This matrix is filled with either 0 or 1 or -1. Here, 0 represents row edge is not connected to column vertex, 1 represents row edge is connected as outgoing edge to column vertex and -1 represents row edge is connected as incoming edge to column vertex.

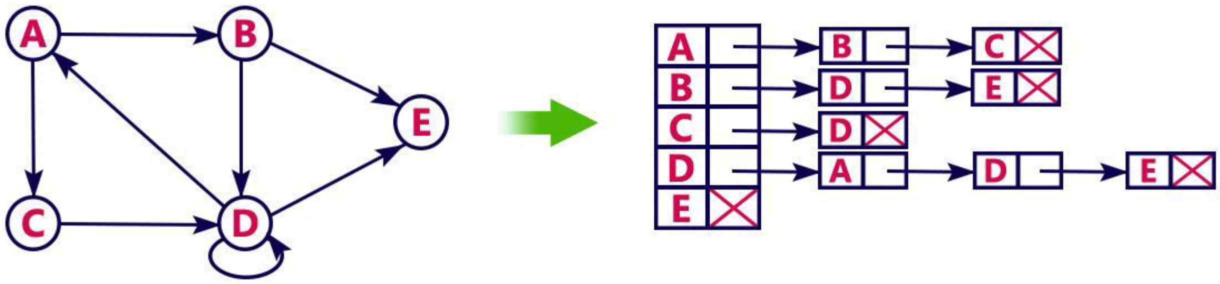


$$\begin{array}{c|cccccccc} & E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \\ \hline A & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ B & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ C & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \end{array}$$

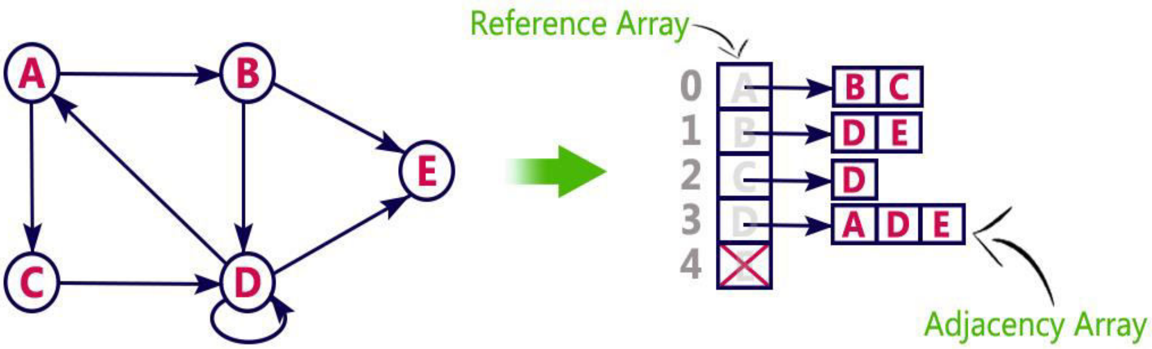
For example, consider the following directed graph representation...

Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices. For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using array as follows..



Q)Explain Graph Traversals

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process. A graph traversal finds the egdes to be used in the search process without creating loops that means using graph traversal we visit all verticces of graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

- DFS (Depth First Search)
- BFS (Breadth First Search)

DFS (Depth First Search): DFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

We use the following steps to implement DFS traversal...

Step 1: Define a Stack of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 3: Visit any one of the adjacent vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.

Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.

Step 5: When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.

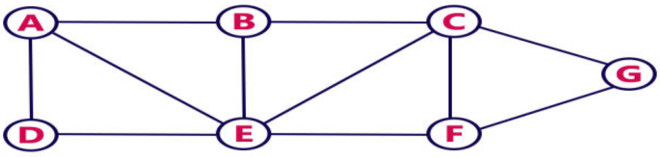
Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Back tracking is coming back to the vertex from which we came to current vertex.

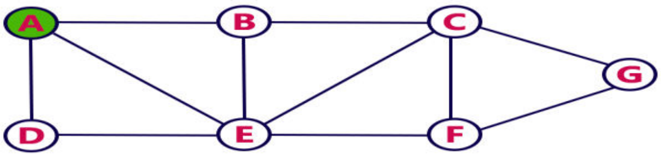
Example:

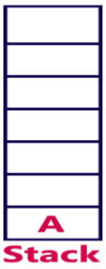
Consider the following example graph to perform DFS traversal



Step 1:

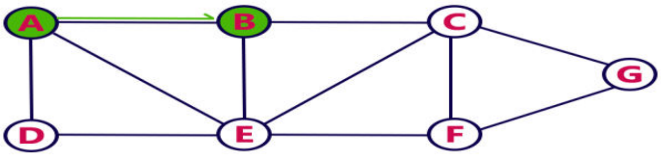
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

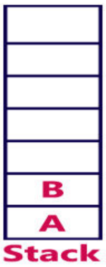




Step 2:

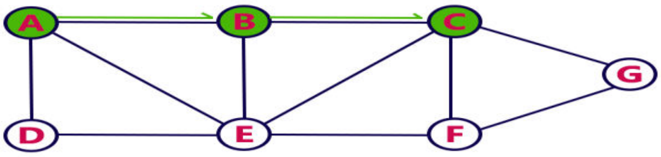
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

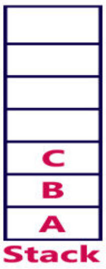




Step 3:

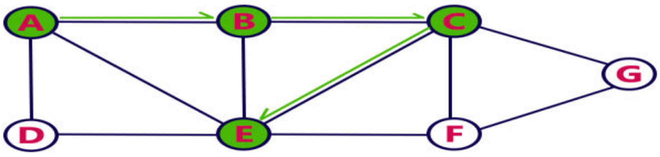
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

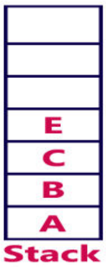




Step 4:

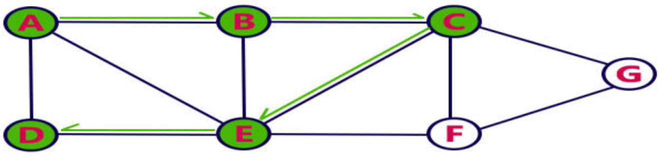
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

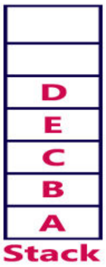




Step 5:

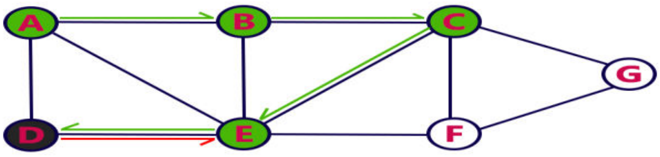
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

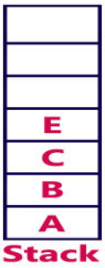




Step 6:

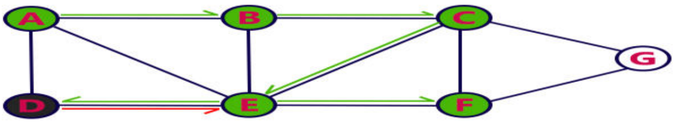
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

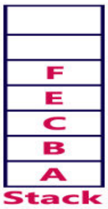




Step 7:

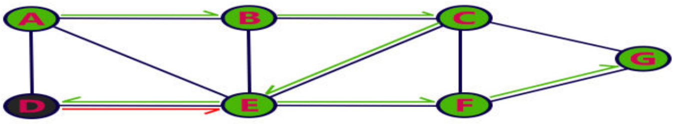
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

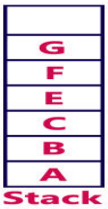




Step 8:

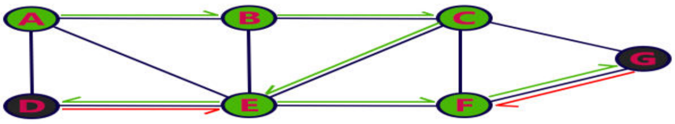
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

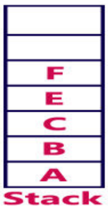




Step 9:

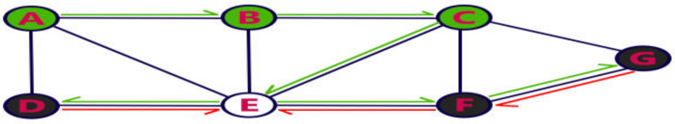
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

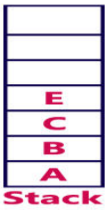




Step 10:

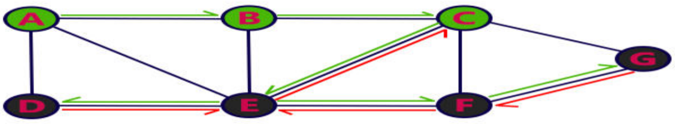
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

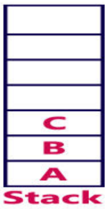




Step 11:

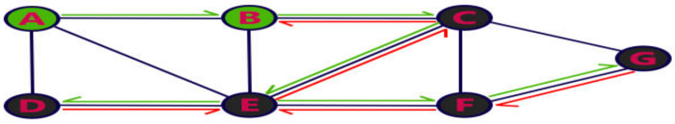
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

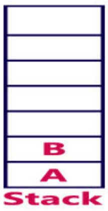




Step 12:

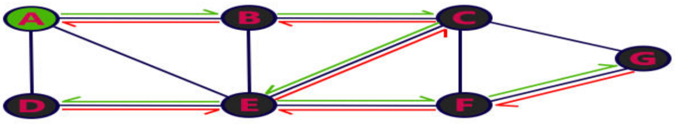
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

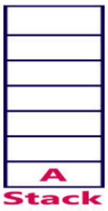




Step 13:

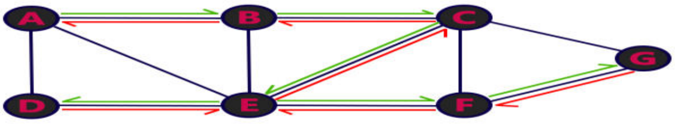
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

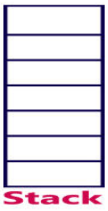




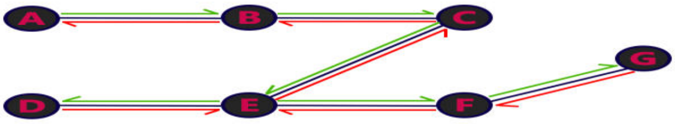
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.





- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



Graph Traversals - BFS

BFS (Breadth First Search)

BFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use **Queue** data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

We use the following steps to implement BFS traversal...

Step 1: Define a Queue of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

Step 3: Visit all the adjacent vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.

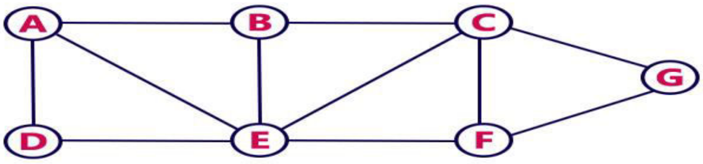
Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.

Step 5: Repeat step 3 and 4 until queue becomes empty.

Step 6: When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

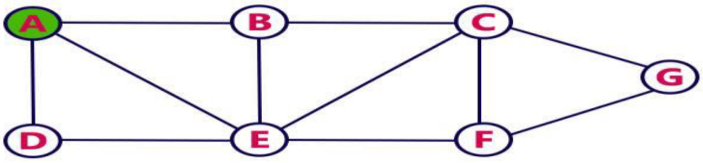
Example

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

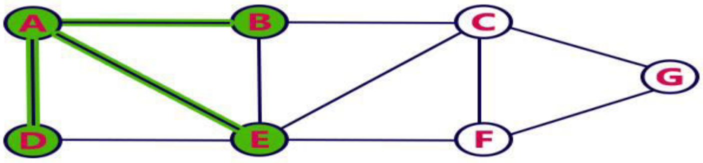


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

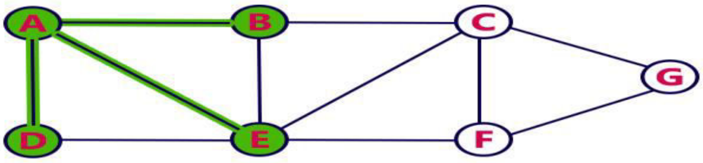


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

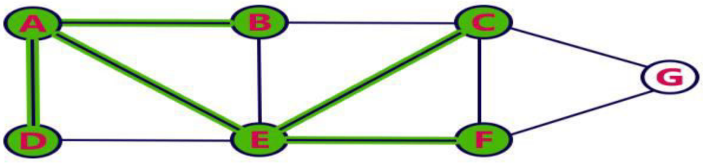


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

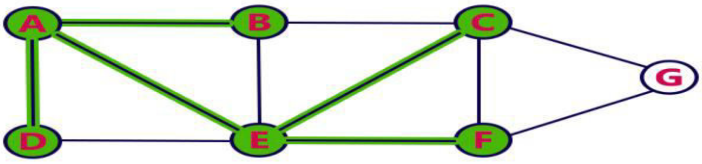


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

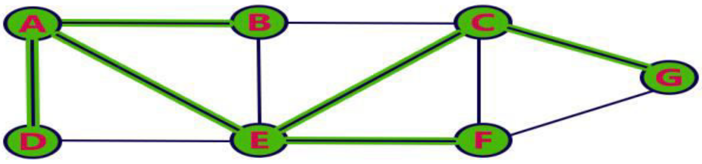


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

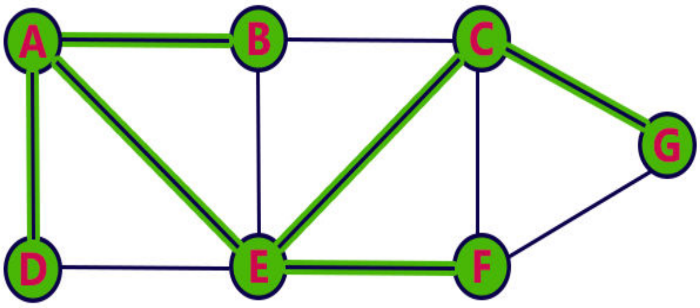


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

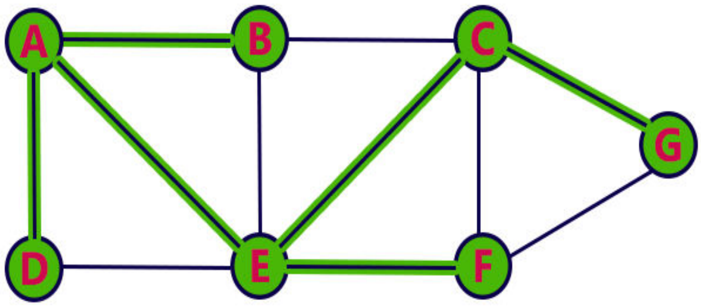


Queue

						G

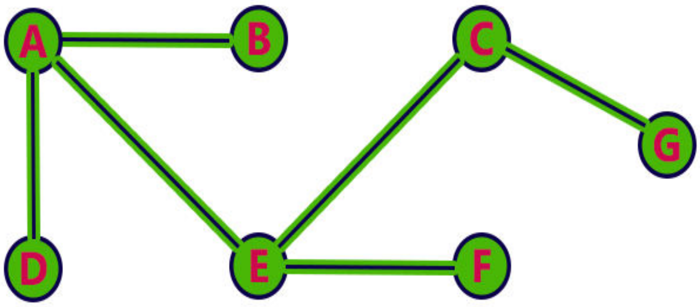
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...

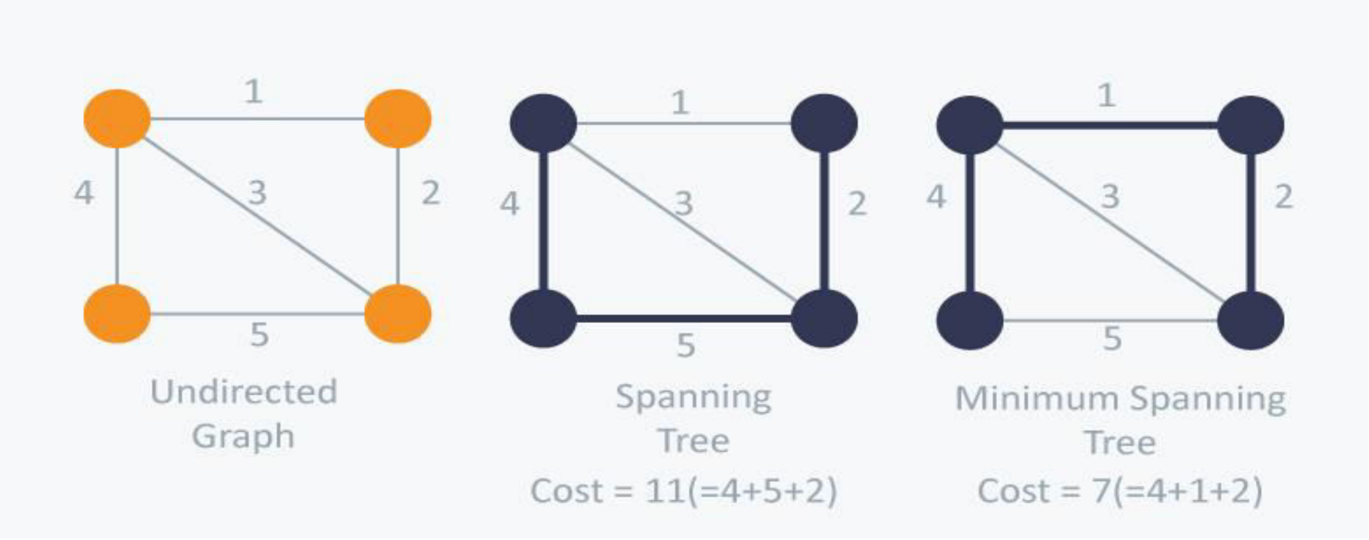


Q:What is a Minimum Spanning Tree?

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

Cluster Analysis,Handwriting recognition,Image segmentation etc



There are two famous algorithms for finding the Minimum Spanning Tree:

Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Algorithm Steps:

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

Prim's Algorithm

Prim's Algorithm also use Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an edge in Kruskal's, we add vertex to the growing spanning tree in Prim's.

Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

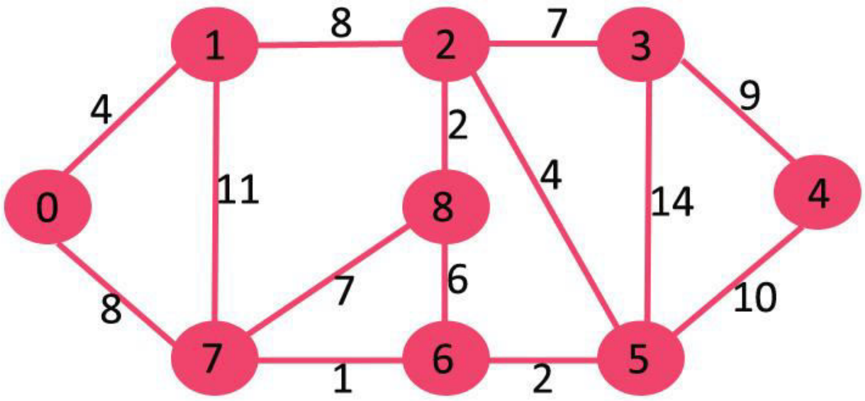
Dijkstra’s shortest path algorithm

- Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree.
- Like Prim’s MST, we generate a SPT (shortest path tree) with a given source as a root.
- We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree.
- At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source.
- The detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm

- Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
- Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
  While sptSet doesn’t include all vertices
  a) Pick a vertex u which is not there in sptSet and has a minimum distance value.
- **b)** Include u to sptSet.
  ….**c)** Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

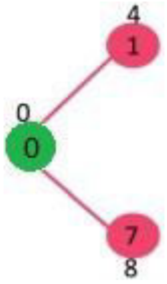
Let us understand with the following example:



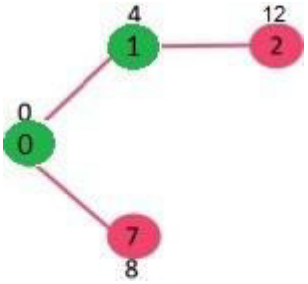
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite.

Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8

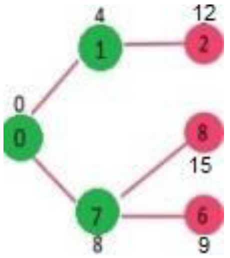
. The following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



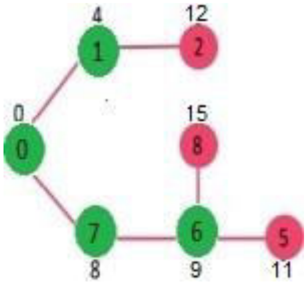
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until *sptSet* includes all vertices of the given graph. Finally, we get the following Shortest Path Tree (SPT).

Application of Graphs:

- **Computer Science:** In computer science, graph is used to represent networks of communication, data organization, computational devices etc.

  In **computer science** graph theory is used for the **study of algorithms** like:

  - Dijkstra's Algorithm
  - Prims's Algorithm
  - Kruskal's Algorithm
- **Physics and Chemistry:** Graph theory is also used to study molecules in chemistry and physics.
- **Social Science:** Graph theory is also widely used in sociology.
- **Mathematics:** In this, graphs are useful in geometry and certain parts of topology such as knot theory.
- **Biology:** Graph theory is useful in biology and conservation efforts.
- Graph theory is used to find **shortest path in road** or a network.
- In **Google Maps**, various locations are represented as vertices or nodes and the roads are represented as edges and graph theory is used to find the shortest path between two nodes.

Sortings and searchings

Sorting:

Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, lexicographical, or any user-defined order.

Sorting can be classified in two types: 1. Internal Sorting

2. External Sorting

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

There are 3 types of internal sorts.

(i) Selection sort :- ex:- selection sort algorithm, heap sort algorithm

(ii) Insertion sort :- ex:- insertion sort algorithm, shell sort algorithm

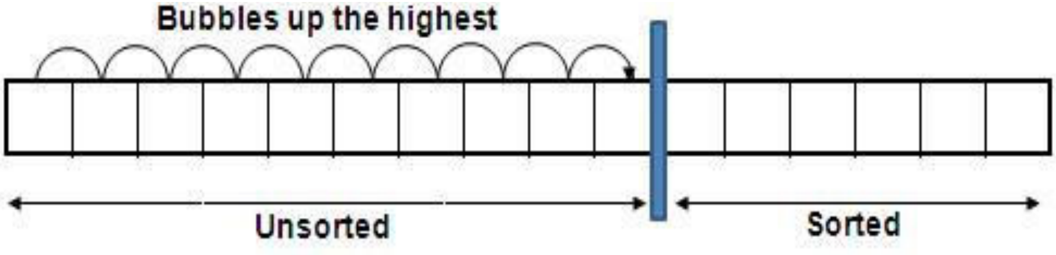
(iii) Exchange sort :- **e**x:- Bubble Sort Algorithm, Quick sort algorithm

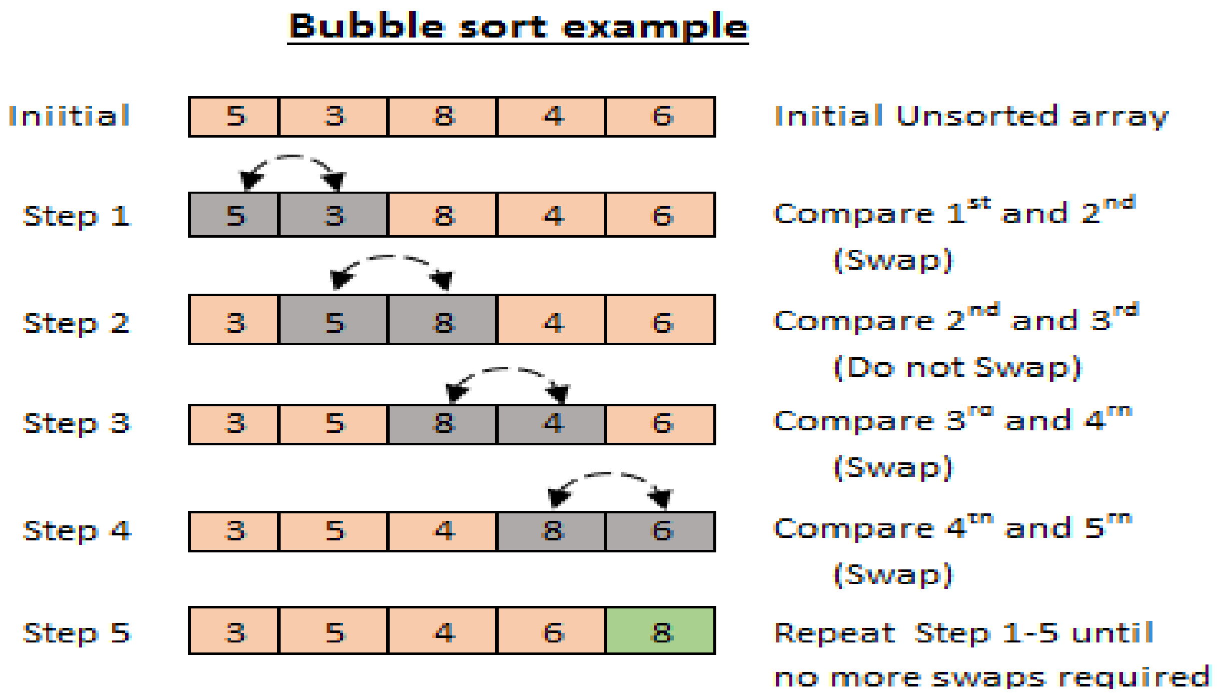
External Sorts:- Sorting large amount of data requires external or secondary memory. This process uses external memory such as HDD, to store the data which is not fit into the main memory. So, primary memory holds the currently being sorted data only. All external sorts are based on process of merging.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

Ex:- Merge Sort.

Bubble sort:

Bubble sort is a simple sorting technique. In bubble sort method the list is divided into two sub-lists sorted and unsorted. This sorting follows comparison-based algorithm in which each pair of adjacent elements are compared and the elements are swapped if they are not in order. . The bubble sort was originally written to bubble up the highest element in the list. Given a list of 'n' elements the bubble sort requires up to n-1 passes to sort the data.




Bubble sort example
Iniitial
5 3 8 4 6
Initial Unsorted array
Step 1
5 3 8 4 6
Compare 1st and 2nd (Swap)
Step 2
3 5 8 4 6
Compare 2nd and 3rd (Do not Swap)
Step 3
3 5 8 4 6
Compare 3rd and 4th (Swap)
Step 4
3 5 4 8 6
Compare 4th and 5th (Swap)
Step 5
3 5 4 6 8
Repeat Step 1-5 until no more swaps required

Algorithm :

```
          Bubble_Sort ( A [ ] , N )
Step 1   :    Repeat For P = 1 to N – 1       Begin
Step 2   :    Repeat  For J = 1 to N – P      Begin
Step 3  :     If ( A [ J]        < A [ J – 1 ] )]
              Swap ( A [ J ] , A [ J – 1])
               End For
                End For
Step 4   :      Exit
```

Program for implementing bubble sort:

```
#include<stdio.h>
void main ()
{
 int i, j,temp,n,a[20];
 printf("enter no of elements");
 scanf("%d",&n);
 printf("enter elements");
 for(i=0;i<n;i++)
 scanf("%d",&a[i]);
   for(i = 0; i<n-1-i; i++)
   {
       for(j = i; j<n; j++)
       {
           if(a[j] < a[j+1])
           {
               temp = a[i];
               a[i] = a[j];
               a[j] = temp;
           }
       }
   }
   printf("Printing Sorted Element List ...\n");
   for(i = 0; i<n; i++)
   {
       printf("%d\n",a[i]);
   }
}
```

Time Complexity of Bubble Sort :

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

Best case	:	O (n)
Average case	:	O (n^2)
Worst case	:	O (n^2)

Insertion sort:

Both the selection and bubble sorts exchange elements. But insertion sort does not exchange elements. Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort the element is inserted at an appropriate place similar to pack cards insertion. Here the list is divided into two parts sorted and unsorted sub-lists. In each pass, the first element of unsorted sub list is picked up and moved into the sorted sub list by inserting it in suitable position until all the elements are sorted in the list.. Suppose we have ‘n’ elements, we need n-1 passes to sort the elements.

The insertion sort algorithm is performed using following steps...

Step 1: Asume that first element in the list is in sorted portion of the list and remaining all elements are in unsorted portion.

Step 2: Consider first element from the unsorted list and insert that element into the sorted list in order specified.

Step 3: Repeat the above process until all the elements from the unsorted list are moved into the sorted list.

Program for implementing insertion sort:

```
#include<stdio.h>

void main ()

{

 int i,j, k,temp,n,a[20];

 printf("enter no of elements");

 scanf("%d",&n);

 printf("enter elements");

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

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

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

  {
        temp = a[k];
        j= k-1;
        while(j>=0 &&  a[j]>temp)
        {
            a[j+1] = a[j];
            j = j-1;
        }
        a[j+1] = temp;
    }
 printf("\nprinting sorted elements...\n");
  for(i=0;i<n;i++)
    {
        printf("\n%d\n",a[i]);
    }
}
```

Time complexity :

Worst case:o(n^2)

Best case:o(n)

Average case for a random array: o(n2)

Merge sort:

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

Algorithm for merge sort:

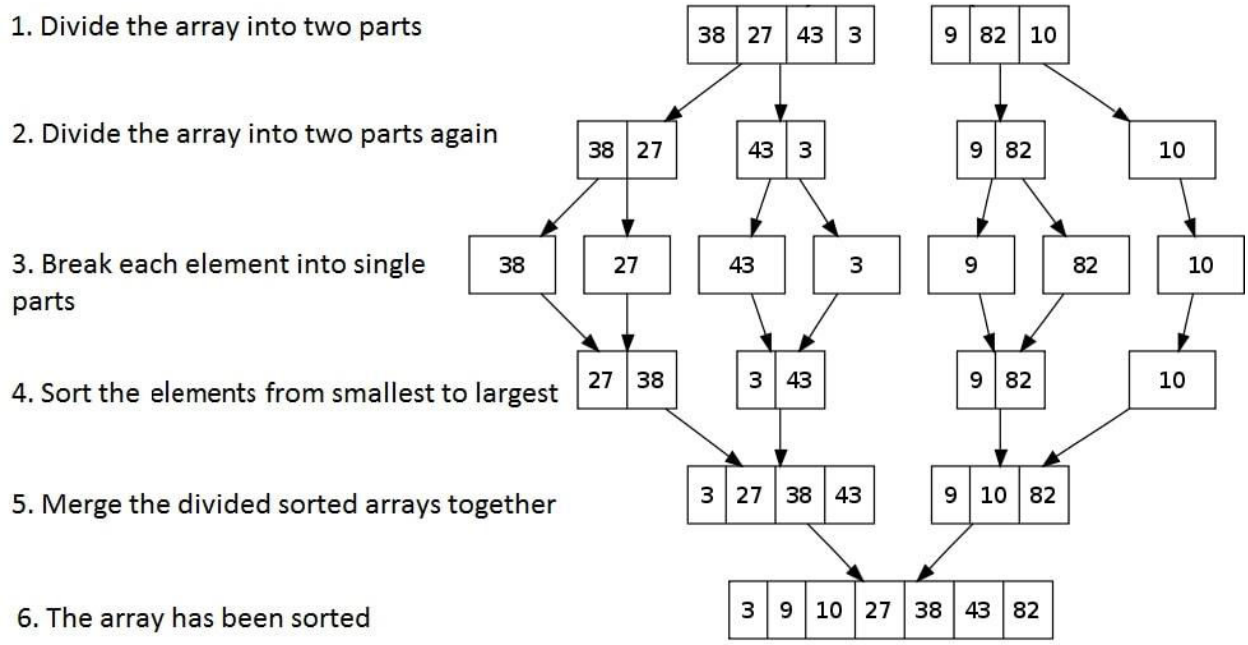
```
MergeSort(arr[], left,  right)
If  (left<right) then
    1. Find the middle point to divide the array into two halves:
        middle m = (l+r)/2
    2. Call mergeSort for first half:
        Call mergeSort(arr, l, m)
```

```
3. Call mergeSort for second half:
       Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
       Call merge(arr, l, m, r)
```

The merge sort algorithm recursively divides the array into halves until we reach the base case of array with 1 element. After that, the merge function picks up the sorted sub-arrays and merges them to gradually sort the entire array

Example:

Program to implement Mergesort:


1. Divide the array into two parts
38 27 43 3
9 82 10
2. Divide the array into two parts again
38 27
43 3
9 82
10
3. Break each element into single parts
38
27
43
3
9
82
10
4. Sort the elements from smallest to largest
27 38
3 43
9 82
10
5. Merge the divided sorted arrays together
3 27 38 43
9 10 82
6. The array has been sorted
3 9 10 27 38 43 82

Program for implementing MergeSort

```
#include<stdio.h>
void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);
int main()
{
int a[30],n,i;
printf("Enter no of elements:");
scanf("%d",&n);
printf("Enter array elements:");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
mergesort(a,0,n-1);
printf("\nSorted array is :");
for(i=0;i<n;i++)
printf("%d ",a[i]);
return 0;
}
 void mergesort(int a[],int i,int j)
{
int mid;
if(i<j)
{
mid=(i+j)/2;
mergesort(a,i,mid); //left recursion
mergesort(a,mid+1,j); //right recursion
merge(a,i,mid,mid+1,j); //merging of two sorted sub-arrays
}
}
 void merge(int a[],int i1,int j1,int i2,int j2)
{
int temp[50]; //array used for merging
int i,j,k;
i=i1; //beginning of the first list
j=i2; //beginning of the second list
k=0;
while(i<=j1 && j<=j2) //while elements in both lists
{
if(a[i]<a[j])
temp[k++]=a[i++];
else
temp[k++]=a[j++];
}
while(i<=j1) //copy remaining elements of the first list
temp[k++]=a[i++];
while(j<=j2) //copy remaining elements of the second list
temp[k++]=a[j++];
//Transfer elements from temp[] back to a[]
for(i=i1,j=0;i<=j2;i++,j++)
a[i]=temp[j];
}
```

Time Complexity :

Best case:o(nlog(n))

Worst case: : o(nlog(n))

Average case: : o(nlog(n))

SEARCHING IN DATA STRUCTURES:

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

There are two types searching techniques:

1. UnOrdered searching- Linear Search
2. Ordered Searching - Binary Search

Linear Search Algorithm (Sequential Search Algorithm)

Linear search algorithm finds given element in a list of elements with O(n) time complexity where n is total number of elements in the list. This search process starts comparing of search element with the first element in the list. If both are matching then return the position of that element and display the result is "element found" otherwise search element is compared with next element in the list. If both are matched, then return the position of that element and display the result is "element found". Otherwise, repeat the same with the next element in the list until search element is compared with last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". That means, the search element is compared with element by element in the list.

Linear search is implemented using following steps...

Step 1: Read the search element from the user

Step 2: Compare, the search element with the first element in the list.

Step 3: If both are matching, then display "Given element found!!!" and also return position of that element then terminate the function

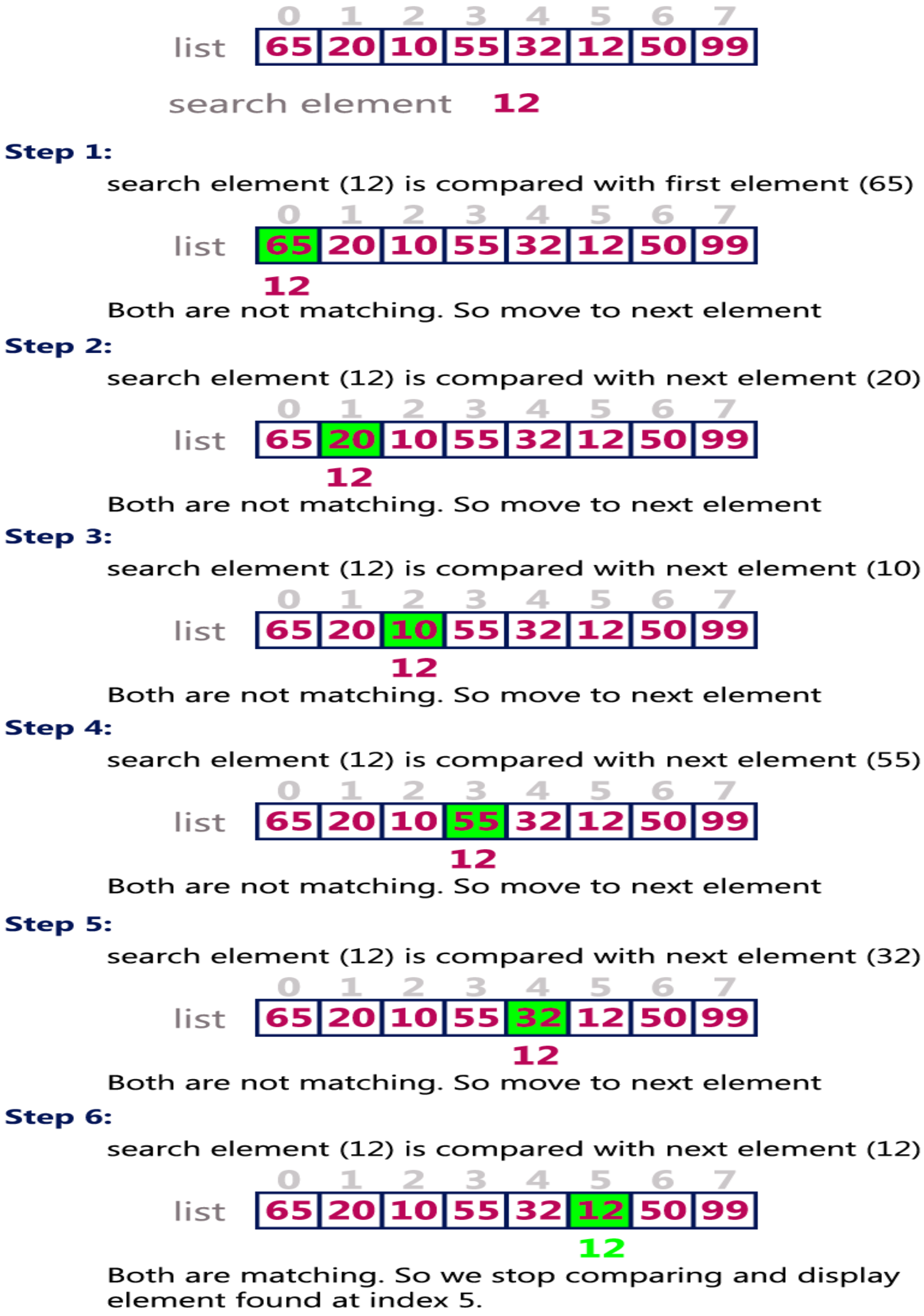
Step 4: If both are not matching, then compare search element with the next element in the list.

Step 5: Repeat steps 3 and 4 until the search element is compared with the last element in the list.

Step 6: If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function.

Example

Consider the following list of element and search element...



	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

search element 12

Step 1:

search element (12) is compared with first element (65)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
	12							

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
		12						

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
			12					

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
				12				

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
					12			

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
						12		

Both are matching. So we stop comparing and display
element found at index 5.

Complexity of Linear Search Algorithm:

Linear search executes in O(n) time where n is the number of elements in the array.

Obviously, the best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made.

Likewise, the worst case will happen when either VAL is not present in the array or it is equal to the last element of the array. In both the cases, n comparisons will have to be made.

Time Complexity :

Best case:o(n2)

Worst case: :o(n2)

Average case: :o(n2)

program to implement linear search:

```
#include <conio.h>
 #include<stdio.h>
 int main()
{
   int a[50],i,n,key;
   printf("Enter size of the  array : ");
   scanf("%d", &n);
   printf("Enter elements in array : ");
   for(i=0; i<n; i++)
   {
      scanf("%d",&a[i]);
   }
    printf("Enter the key : ");
   scanf("%d", &key);
       for(i=0; i<n; i++)
        {
           if(a[i]==key)
             {
                          printf("element found at %d position ",i);
                         return 0;
              }

      }

         printf("element  not  found");
}
```

Binary Search:

The binary search algorithm can be used with only sorted list of element. That means, binary search can be used only with list of element which are already arranged in a order. The binary search cannot be used for list of element which are not in order.

This search process starts comparing of the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sub list of the middle element.

If the search element is larger, then we repeat the same process for right sub list of the middle element. We repeat this process until we find the search element in the list or until we left with a sub list of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list".

Binary search is implemented using following steps...

Step 1: Read the search element from the user

Step 2: Find the middle element in the sorted list

Step 3: Compare, the search element with the middle element in the sorted list.

Step 4: If both are matching, then display "Given element found!!!" and terminate the function

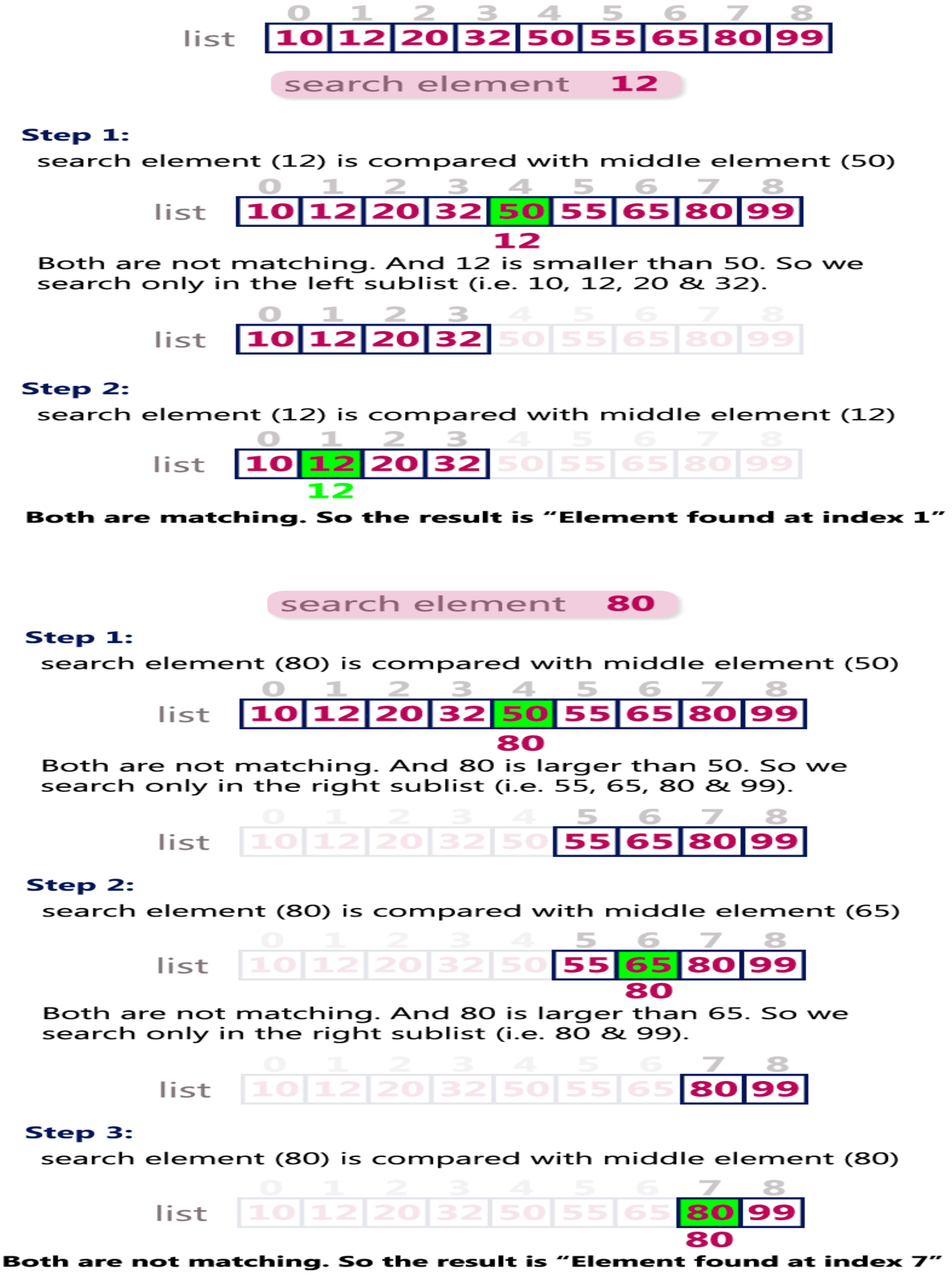
Step 5: If both are not matching, then check whether the search element is smaller or larger than middle element.

Step 6: If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.

Step 7: If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.

Step 8: Repeat the same process until we find the search element in the list or until sublist contains only one element.

Step 9: If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.



	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

search element 12

Step 1:

search element (12) is compared with middle element (50)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

12

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Step 2:

search element (12) is compared with middle element (12)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

12

Both are matching. So the result is “Element found at index 1”

search element 80

Step 1:

search element (80) is compared with middle element (50)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

80

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Step 2:

search element (80) is compared with middle element (65)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

80

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Step 3:

search element (80) is compared with middle element (80)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

80

Both are not matching. So the result is “Element found at index 7”

example

```
#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 isn't present in the list.\n", search);

  return 0;
}
```

Time Complexity :

Best case:o(1)

Worst case: :o(log n)

Average case: :o(log n)