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:

When a programmer develops a program, he needs to concentrate on three things viz., developing user interface, process and the data storage. In C language the user interface is managed with the help of IO functions and the process is managed with the help of operators, branching, loops, functions etc. the data is managed with the help of variables, arrays, pointers, structures etc.

In the study of computer science, the professional must study the organization for which the management is to be done and should make a study of the flow of data and how the data is to be organized in computer. .

The data structure name indicates itself that organizing the data in memory. There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C language.

The data structure is not any programming language like C, C++, java, etc. It is a set of algorithms that we can use in any programming language to structure the data in the memory.

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage.

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily. It represents the knowledge of data to be organized in memory. It should be designed and

implemented in such a way that it reduces the complexity and increases the efficiency.

The different data structures used in computer science are stacks, queues, linked lists, trees, graphs etc. To structure the data in memory number of algorithms were proposed, and all these algorithms are known as Abstract data types. These abstract data types are the set of rules.

Data Representation:

Data type is an attribute associated with a piece of data that tells a computer system how to interpret its value. Understanding data types ensures that data is collected in the preferred format and the value of each property is as expected.

The basic unit of data representation is a BIT i.e. a Binary Digit, either 0 or 1. Various combinations of these two values represent different data values in different systems.

Eight bits together for a BYTE and it represents a character or a symbol in keyboard. A character or more no. of characters forms a string. Thus, the string is an ADT that provides many facilities to do process on strings like storing, concatenating, counting characters etc.

Integer is another representation of data, which lets to do numeric addition, subtraction, multiplication etc operations. The ADT of integer permits to store negative numbers and also positive numbers. But based on the software system being used, the capacity may vary from system to system.

Real data type representation supports floating point values like 2.5 etc. The real data type is represented with different names in different software system.

Character representation of data, is used for storing and managing values on which numeric operations are NOT performed. For example, if we need to store salary of an employee, we may represent data as integer ADT or a real ADT. But the same employee ID, or name or mobile number etc, on which e need not perform any numeric operation, then we can represent such data as character ADT, or a string ADT. There are different codes called EBCDIC, ASCII are used to store the character type data.

Abstract Data Types:

An abstract data type (ADT) is defined as a mathematical model of data objects that make up data type and the functions that operate on those data objects. The ADT is the specification of logical and mathematical properties of a data type or data structure.

The ADT acts as a useful guideline to implement the data type correctly. This specification is implemented with syntaxes of languages. The ADT describes two things.

➔ how the data is related with each other
➔ the statements of operations that can be performed on that data type.

For example, the computer science defined INTEGER type ADT, specifying that it stores a numeric value and that can be performed with addition, subtraction etc processes. The languages C, C++, java etc are created a data type "int" as per the specifications given in that ADT.

The INTEGER-ADT defines the set of numbers from "–infinite, … -3., -2, -1 , 0 , 1, 2, 3, 4,… infinite" . And in C language it is implemented with "int" data type. Based on language, there are some other issues to be identified as, range of the data type, the memory space occupied by the value, different operations that can be performed etc.

For example, Stack is one of the data structures, where data can be inserted and removed only from one end. The new element joining the set is kept on TOP and only the top most can be removed from it. In other words, only one element at TOP is accessible.

The informal ADT of STACK is as follows :

STACK: {1,2,3,4…n} , TOP
Condition : TOP=null

Insert operation: push()
    Precondition: top!=n
        insert (data)
        TOP - > data

Removal operation: pop()
    Precondition: TOP!=null
        remove(top)
        top - > current data

Data Types

Data has different definitions from different views. From the view of a database programmer, it is collection of raw facts. From programmers view, it is a value available in user environment that has no meaning to user. In the view of system / language architecture, a value that is stored in the memory in the form of bits.

Though there can be a universal (common) representation for data, but to store 8 bit data (character) or a 16/32 bit data (integer) or any floating value or a string, there are different issues to be considered, as memory space occupied. A floating value occupies large space than an integer value. If same memory space is occupied by for character, integer and also for a float, then large volume of memory is occupied for small data value. And different types of data values need different interpretation (type of process) to be done, and it becomes difficult if a universal representation is used in computers to manage data.

Considering all above problems, different data type representations are provided in all languages, to have better memory management and provide different processes to be done on the different type of elements.

Primitive Data Types

Every system has its own set of native data types called basic data types or primitive data types that defines how the data is internally represented and stored, and retrieved from memory.

The primitive data types can be directly controlled by computer commands. That means it is defined by the system and compiler. The following are Primitive data structure in C.

1. Integer
2. Character
3. Double
4. Float
5. String

Integer: In the integer, it includes all mathematical values, but it is not include decimal value. It is represented by the int keyword in the program.

Character: The character is used to define a single alphabet in the programming language. It is represented by the char keyword in the program.

String: The group of the character is called a string. It is represented by the string keyword in the program. The string is written with a double quotation mark ("-"). For example: “My name is Bob”.

Float and Double: Float and double is used for real value.

Data Structure and Structured Type

A Data Structure refers to a set of data elements i.e. computer variables that are connected in some logical manner. That logical manner is like using indexing in case of arrays and using pointers to point to next element in case of structures.

A simple data type can store only one value at a time. A structured data type is one in which each data item is a collection of other data items. In a structured data type, the entire collection uses a single identifier (name). The purpose of structured data types is to group related data of various types for convenient access using the same identifier. Similarly, data can then be retrieved as an aggregate, or each component (element) in the collection can be accessed individually. Furthermore, a structured data type is a user-defined data type (UDT) with elements that are divisible, and can be used separately (independently) or as a single unit, as appropriate.

C simple data types:

1. integral (char, short, int, long, and bool)
2. enum
3. floating (float, double, long double)

C structured data types:

1. array
2. struct
3. union

***Even though individual characters in a string can be accessed, the string data type is not considered a structured data type.

Atomic Type

Generally a data structure contains 2 parts. The DATA and LINK (a pointer that stores address to next data element). This makes it easy to store the DATA and relating it to some other data by storing its address in a pointer of LINK part.

The atomic data type is an element that stores only data. So, by itself, it cannot point / link to other data values. So using these atomic types of the language, the programmer need to create his own TYPE specification (ADT) that

can store a data value and also that can store address of next element to link data values, to for a data STRUCTURE.

The individual element that stores a value and has a pointer to next element is called a NODE. So a NODE will have data and a next pointer to point to next element. When a new node with a value is inserted, its address is stored into the next pointer of the last node, so that the last node's next pointer links to the new node inserted.

Using the atomic types, the programmers implement different algorithms to form a data structure.

Difference between Abstract Data Types, Data Types, and Data Structures

The ABSTRACT data type is the specification of the data type, which specifies the logical and mathematical model of data. For example INTEGER is an ADT that specifies how the WHOLE NUMBERS from –infinite to + infinite are to be represented in system,

The data type is the implementation of the abstract data type by a specific software system. For example the data type "int" implements the ADT- INTEGER in C and the same is represented as "number" by other software system called oracle.

The data structure refers to the collection of computer variables that are connected in a specific manner. Using the user defined structures in C and classes in C++/java, we can develop different ways of data storage and management like, Linked Lists, Trees and Graphs etc called data structures.

Refinement Stages

Refinement is the idea that software is developed by moving through the levels of abstraction, beginning at higher levels and moving down by refining the software by providing more detail at each increment. At higher levels, the software is merely its design models; at lower levels there will be some code; at the lowest level the software has been completely developed. This follows a top-down approach. This entire refinement stages are called SDLC (Software Development Life Cycle).

More practically the refinement is the process of elaboration. We start with a statement of function that is defined at a high level of abstraction. That is the

statement explains function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. Refinement reason that the designer to elaborate on the original statement providing more and more detail as each successive refinement occurs.

The basic refinement stages are discussed below:

Conceptual level: this is also called abstract level refinement. In this the data requirements, their relationships and the operations to be done are identified. The process to be done is NOT identified in this level.

Algorithmic level: this is also called data structure level. At this level, the process to be done on data is identified and the required data structure is identified for storage and management of data, as such either to use a stack or a queue or a linked list etc.

Implementation level: this is also called programming level refinement. At this level, it is decided how the data structure is to be implemented in the memory. For example, if the program demands a BINARY TREE in algorithmic level, then at this level it is decided whether to develop a linked list to implement binary tree or use an array to implement binary tree.

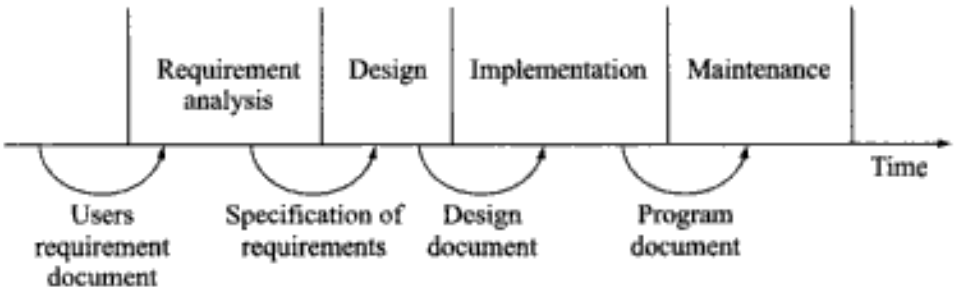
Application level: This level settles all details required for a particular application like variables names, special requirements for operations required for applications.

PRINCIPLES OF PROGRAMMING AND ANALYSIS OF ALGORITHMS

Software Engineering

Software Engineering is the practice of methods helpful for construction and maintenance of large software systems. Software engineering is defined as a process of analyzing user requirements and then designing, building, and testing software application which will satisfy those requirements.

It is tedious (very difficult) process of developing a software system (project) and it takes a very long time to put it into use. The system undergoes different stages before put into use and this is simply called SDLC (Software Development Life Cycle).



1. **Analyze** the problem precisely and completely.
2. **Build** a prototype and **experiment** with it until all specifications are finalized.
3. **Design** the algorithm using the tools of data structures.
4. **Verify** the algorithm, such that its correctness is self-evident.
5. **Analyze** the algorithm to determine its requirements.
6. **Code** the algorithm into an appropriate programming language.
7. **Test** and **evaluate** the program with carefully chosen data.
8. **Refine** and **Repeat** the foregoing steps until the software is complete.
9. **Optimize** the code to improve performance.
10. **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 phase that the algorithms and data structures to solve a problem are proposed. Some of the various points that can help us evaluate the proposed program designs are as follows:

- As the design stage involves taking the specification and designing solutions to the problems, the designer needs to adopt a design strategy. The strategy adopted while designing should be according to the given specifications.
- 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 are not aware of the program design you have adopted. Thus, there is a system manual which is a detailed guide to how the design was achieved. In addition a user manual serves as a reference for the users who are not familiar with the system or machines.
- A large program should be divided into small modules and submodules by following one of the two decomposition approaches—top-down approach or bottom-up approach.
- Other important criteria by which a program can be judged are execution time and storage requirement.

Algorithms

The ALGORITHM is to the logical representation of the program. It is defined as “a sequence of instructions that must be followed to solve a problem”.

The general characteristics of algorithm are:

- each instruction must be unique
- each instruction must be relative in nature of the program
- repetition of the same task is to be avoided, i.e. infinite recursive execution must be avoided
- the result must be available as a meaningful statement to the user on completion of algorithm execution.

Once algorithm is designed, it is checked for its correctness by giving some input values and checking for correct outputs. After this simple testing of algorithm, its efficiency is analyzed. This analysis includes the CPU time consumed, memory consumed by the algorithm. This is called measuring the complexity of the program.

Different Approaches to Designing an Algorithm

A complex software system is divided into small units called modules (using user defined functions while coding). The advantage of modularity is each module can be checked independently without concentrating on other modules and also provides a facility of integrating all modules together and making a structured walk through. There are two important approaches in designing the algorithms. They are TOP-DOWN approach and BOTTOM-UP approach.

The to-down approach is also called water fall modal. In this modal, the higher level components are identified and they are further decomposed into smaller units. It provides a step by step refinement. It starts with abstract level and further moves down to the results.

The bottom up approach is opposite to the above approach. Here the lowest level component is identified and the design starts with that and moves to the top level components in the design. That is, this approach starts with the last layers of the abstraction and moves up to the higher layers of the abstraction.

Complexity

In computer programming, the term complexity refers to computational complexity. In which we measure the space required for an algorithm in computer's memory and time consumed by the algorithm during execution.

The complexity is represented as f(n), where n denotes the size and time required is represented with a function f().

i.e. the f(n) means, time required for an algorithm for 'n' number of inputs. While measuring the time complexity, we consider only count of key statements. Key statements mean only the basic instructions of the algorithm. Because, in the computer system as all the resources are shared by different resources, exact time cannot be measured.

Considering the following algorithm:

```
ALGORITHM 1   :     a=a+1

ALGORITHM 2   :     for  x=1 to n step 1
      a=a+1

ALGORITHM 2   :     for  x=1 to n step 1
                    for  y=1 to n step 1
      a=a+1
```

In the algorithm,1 a=a+1 is independent and executed only once. To the count of frequency of basic statement is '1'.

In the second algorithm, the statement is within a loop, and the loop runs for f times, so the frequency count becomes 'n'.

In the last algorithm, the frequency becomes n^2.

So here in last algorithm, the magnitude (count / limit of execution / size ….. PARIMANAM in telugu) increases by squaring the size of 'n' like……

if 'n' is 1, then the magnitude 1
if 'n' is 2, then the magnitude 2
if 'n' is 3, then the magnitude 9

if ‘n’ is 4, then the magnitude 16
if ‘n’ is 5, then the magnitude 25 and so on and if it is 10… its magnitude becomes 100.

The space complexity is total fixed memory required and total variable memory required for storing data.

Big ‘O’ Notation

This is used to measure the cost of an algorithm.

The letter ‘O’ is actually pronounced as “OMICRON” and it is a Greek letter which means “rate of growth” and in other words “order of”.

So Big ‘O’ notation was actually called big omicron notation during late 18 hundreds and now a days is big ORDER notation that specifies the rate of growth in an algorithm.

It is an algebraic expression to measure the cost, in which different representations are used like f(n) and g(n) etc. where the computing time of the algorithm is expressed as f(n) and the GROWTH of a specific function is represented as g(n) where ‘n’ is either no. of inputs or number of outputs or no. of processes executed in that function.

So f(n) is O g(n), pronounced as f(n) is order of g(n), that means, the complexity of the algorithm is in order of growth of the algorithm. It is represented as f(n) = O(g(n)).

Taking example of the above three algorithms discussed in complexity analysis, the f(n) the overall complexity is measured based on O(g(n)) and the performance of the algorithm is decided and the programmers can prefer a specific less complex algorithm.

The big -O notation is widely used to measure the complexity of the algorithm or performance of the algorithm that can be preferred for implementation.

In measuring the performance, the complexity is measured either with constants like O(n) , which is called a linear time, $O(n^2)$ called quadratic time,

$O(n^3)$ is cubic time and $O(n^m)$ is exponential time and further, large algorithms need to measure the complexity with logarithms. O(log(n)) etc.

Algorithm Analysis

An algorithm is a logic written in simple English to solve a problem. A problem may be of one statement and there can be different number of LOGICS / ways / algorithms to solve that problem. Some solutions may be more efficient than other s to solve that problem.

For implementation of an algorithm to solve a specific problem, we calculate the complexity of that algorithm and choose the best one. While analyzing the algorithm's complexity we can classify them into following:

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

Best case time complexity of an algorithm is a measure of MINIMUM time the algorithm takes for 'N' inputs. Here, not only the 'n' inputs, but also their order of input is also to be considered.

The worst case time complexity is a measure of maximum time it takes for 'n' inputs. And the average time is a measure of average of best and worst cases. But the average case time is not considered much.

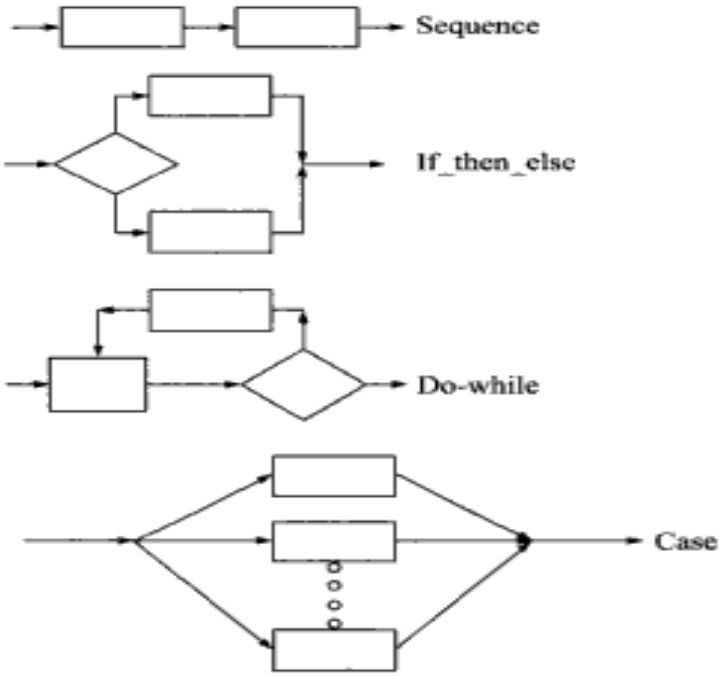
Structured Approach to Programming

Structured programming is a subset of software engineering. It is a method of designing and coding programs in a systematic way.

The structured approach mainly concentrates on technical aspects of the problem and the software engineering concentrates on multiple aspects like technical aspects, financial aspects, psychological aspects and managerial aspects.

Literally a program is ALGORITHMS+DATA STRUCTURES.

In the structured approach, a pictorial representation of an algorithm is made first for easy analysis of flow of control and then the implementation of algorithm is done. The different control symbols used are shown below:



Recursion

Recursion is a concept of calling a function within itself. In software engineering while developing solutions for problems, the recursive routines are the powerful tool.

When a programmer needs same process to be done on some data values again and again, the recursion is the best solution.

Considering the example of factorial algorithm, factorial of a number can be found in either a recursive method or in a non-recursive method.

```
int factorial(int n)
{
        int fact=1;
        if(n>1)
                fact=n * fact(n-1); //recursive call.

        return fact;
}
```

In above function, if the argument given is 5, then the code executed on value 5 is repeatedly executed on further on values 4, 3, 2 and the once the n becomes 1, then the "return" returns itself, by skipping the recursive call.

In non-recursive methods, we may use any of the iterative statements like a while loop, or a do-while or for loops. But while getting into complex situations, the recursive method is preferable.

```
int factorial(int n)
{
        int fact=1;
        while(n>1){
                fact=fact * n;
        }
        printf("%d",f);
}
```

In above both cases, the code gives same results, but when the complexity of the algorithm is compared, the iterative method has less complexity than the recursive method.

But further when going into heavy algorithms, the iterative statements become inefficient to provide solution for many aspects where the recursive methods are adapted.

Principles of recursion: there are some principles to be followed while designing recursive algorithm is as follows.

Find Key step to be executed recursively

Find a stopping rule, when the recursion is to be stopped

Identify the outline of the algorithm

Check the termination whether it is done based on stopping rule

Draw a recursion tree to determine the amount of memory and number of times the recursion is executed.

Tips and Techniques for Writing Programs in ‘C’

Any program in any programming language generally follows its own structure. The general outlook / structure of a C program looks as :

```
Comments
Pre-processor
Global variables/ declarations
main()
{
        Process
}

function1()
{
        process
}
function2()
{
        process
}
```

The comments can be given anywhere. These are used to indicate what is going on in the code in that location.

The preprocessor directives are used to link required libraries and develop macros. Like we use #include to link a library file and #define to develop macros for writing code efficiently. The macros are also called symbolic constants in the program.

Syntaxes:

```
# include<file name>
#define macro_name macro expansion
```

The global variables can be normal variables or constant variables that are accessible throughout the program. If we need some variables to have fixed values and do not change during running of program, then they can be declared as constants.

The typedef is a keyword used to define a new data type as of existing data type. Literally, it is giving a new name to existing data type.

The main() is the starting point of the program and the program DRIVER code is placed within main() as the C runtime calls the main() function.

Entire program is not placed within main(). Only the driver code is placed in main() and the further program is divided into functions.

Following above tips, we can write an efficient program in C.

UNIT II
ARRAYS

Introduction to Linear and Non- Linear Data Structures

Data structures are classified into two types linear data structure and Non-Linear data structure. In a linear data structure, the elements form a sequence. Such linear data structure can be represented in memory in two ways.

By having linear relationship between elements represented by means of sequential memory locations and these are called arrays.

By having relationship between elements that do not have sequential memory and represented with the help of pointers, and these are called linked lists.

Arrays are useful when the number of elements is fixed. And the linked lists are preferable when the number of items in the collection is not fixed.

There are other non-linear data structures like trees and graphs can be performed on these two data structures viz. arrays and linked lists.

The different operations that can be performed on data structures are traversal (moving across each element), searching, and insertion, deletion, merging and sorting elements.

One- Dimensional Arrays:

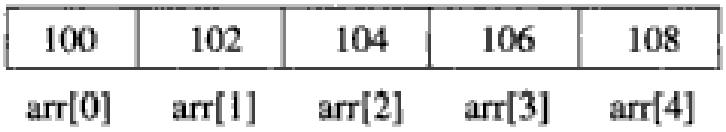
An array is collection of finite number of elements of similar type that are stored in adjacent memory locations. Where each location is identified with a unique number called index number. The first index number is called lower bound and it is always ZERO and the last index of the array elements is called upper bound and it is always equal to number of elements -1.

In C, we use a special symbol [ ] to declare and manipulate the arrays.

The [ ] are used to specify size of array or bounds or number of elements in that array while declaring and the same [ ] are used to specify the index number of each element in the array.

In the example int arr[10]; there is an array of 10 integer elements array is declared. And all the elements will have sequential memory allocated.

It looks in memory as shown below:



Where the 100,102 etc are the memory addresses of each element.

The array can also be initialized while declaring as shown below:

int arr[5]={10,20,30,40,50};

an array of characters is called a string and it can be initialized in 2 ways :

char name[]=”geetha”;
char name[]={’g’,’e’,’e’,’t’,’h’,’a’};

when a string is stored in a char array, the C-RE placed a NULL value at end of string for identification.

The address of the first element in the array is called BASE address.

Array Operations:

There are different operations that can be performed on elements of array:

Traversal : to display or count or to do any other process, each data element need to be accessed. This accessing each element is called traversal.

Algorithm for traversal:

1. Initialize counter
   I=LB
2. Repeat steps 3 & 4 while I < UB
3. Visit element
   Apply process to arr[I]
4. Increase counter I
   I=I+1
5. Exit

Insertion: insertion is adding an element in the array in specified location. Inserting a value at last element is simple. But inserting an element at middle of the array is more expensive. For inserting a value in middle of the array, we must move all elements from Nth location to their next locations first and then insert value at specified location. Provided that, the array must have enough space to move elements.

Algorithm:

1. Initialize counter
   I=UB
2. Repeat steps 3 & 4 while I >POS
3. Visit element
   Move arr[I] into arr [I + 1]
4. Decrease counter I
   I=I-1
5. Set arr[POS-1] = NEW_VALUE
6. Exit

Deletion : deletion from array is removal of an element at specified location. Literally we have nothing to do in deleting. Simply if we move all elements to its previous locations from the position of deletion, the element at specified position will be deleted automatically.

Algorithm:

1. Initialize counter

I=POS-1

2. Repeat steps 3 & 4 while I < UB
3. Visit element
   Move arr[I + 1] into arr [I]
4. Increase counter I
   I=I+1
5. Exit

**The searching and sorting are discussed later

Two- Dimensional arrays:

When a program needs much data to be managed in a tabular format, then the arrays can be used by providing 2 references for each data element for flexible manipulation.

For providing 2 references for each data element, we may declare the array with 2 bounds specified. For example int arr[2][3];

In this case, it is treated as a 2 dimensional array, and mathematically it is treated as a MATRIX, that has rows and columns.

Though we treat a 2 dimensional array as a matrix, i.e. in tabular form, the memory allocated in RAM will be sequential. We LOGICALLY treat it as a matrix.

In the above declaration int a[2][3]; the logical view of the matrix is as follows:

Arr[0][0]	Arr[0][1]	Arr[0][2]
Arr[1][0]	Arr[1][1]	Arr[1][2]

But it is represented in memory sequentially as :

Arr[0][0]	Arr[0][1]	Arr[0][2]	Arr[1][0]	Arr[1][1]	Arr[1][2]
100	102	104	106	108	110

The 2 D array can be used as row major matrix or column major matrix, where in row major, the first dimension refers to row and 2nd refers to column and in the

column major matrix first dimension refers to column number and the 2nd dimension refers to row.

When using a 2D array, we use 2 loops to refer each dimension in the matrix. When using a 2 D Array as a matrix if many of the values in a matrix are ZEROs then it is represented as a sparse matrix.

The term transposing matrix is used when a matrix is converted such that, the first matrix's rows and columns are treated as columns and rows of second matrix.

This is not in your syllabus. But something important in DS. So understand properly.

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have 0 value, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix ?

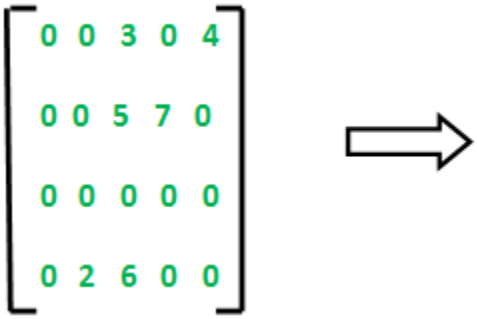
- Storage: There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- Computing time: Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with triples- (Row, Column, value).

Sparse Matrix Representations can be done in many ways. Following is one of the ways:

2D array is used to represent a sparse matrix in which there are three rows named as

- Row: Index of row, where non-zero element is located
- Column: Index of column, where non-zero element is located
- Value: Value of the non zero element located at index – (row,column)

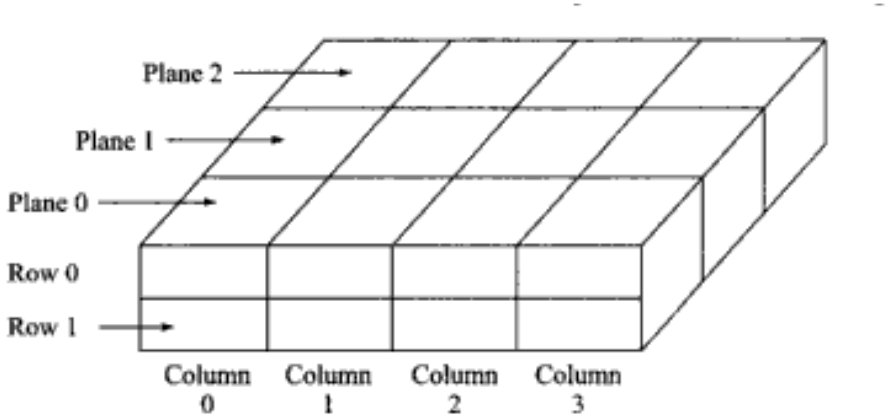


Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

This is very much useful in managing GRAPHS.

Multidimensional Arrays: the arrays that have more than 2 dimensions called multi dimensional arrays. For example a 3 dimensional array is declared as int a[2][3][4];

Each element in a 3D array has three referencing indexes i.e. subscripts. The first specifies the PLANE number the second specifies ROW number and the 3rd specified the column number.



Pointers and Arrays:

The pointer is a special data type used for dynamic memory management. In C, when an array is declared, there will be a pointer declared with the same name of array and it will be held with the base address of the array. i.e. an array is a synonymous pointer.

When using an array, we use [] to specify the index number of the element. When using pointer syntax, we use pointer incremented by the index number. As pointer arithmetic operation is done on no. of bytes based on type of pointer, if it is an int *p; and held with address 100, and if we do p+1, it will be increased by 2 bytes and it becomes 102.

The pointers can be done only with addition operation. We can not do subtraction, multiplication or division on pointers, as it may lead to negative numbers, where the memory addresses can not be in negative values.

So we can use the synonymous pointer to access array elements. For example, a[2] can be accessed as * (a + 2).

Overview of Pointers:

A pointer is a variable which contains address of some memory location. Pointers are popular in programming because they provide direct access to memory locations, support dynamic memory management and these improve efficiency of some routines.

In main memory each byte is identified with a byte number. When we declare a variable, the first byte number reserved for that variable is called address of that variable. it can be fetched with the operator &. The & takes variable name and returns its address. Similarly, the operator * is called de-referencing operator, it takes address and returns the value at that address. The & means ADDRESS OF and the * means VALUE AT.

The pointer arithmetic operation is different from normal arithmetic operation. When we manipulate the pointers without *, the addresses are manipulated. That is when a pointer ‘p’ is done ++, it increases the address in that pointer, based on data type of the pointer. If it is an integer pointer increases by 2 bytes and if a float, increases by 4 bytes.

When the pointer is manipulated with '*', then the values pointed by that pointer are manipulated. A pointer declared in a function can point to any memory location directly even if that memory has no scope in that function.

So the function call by reference has more advantage than function call by value to do process on the variables that have their scope in other functions. In call by value, the variable names are passed as arguments and received into formal arguments and formal arguments are processed. But in call by reference, the address are passed as arguments and received into pointers at function definition and the data is processed through pointers. So the advantage of call by reference OR manipulating data used pointers is that the original values are manipulated with pointers, wherein the call by value doesn't affect original values.

Something important is to remember is when a pointer is freed, it should not be left as it is, and these are called dangling pointers and these misbehave in program. A good programmer must assign NULL to the pointer once it is freed.

CHAPTER 2
Linked Lists

Linked Lists

A linked list is a very flexible, dynamic data structure in which elements (called nodes) form a sequential list. In a linked list, each node is allocated space as it is added to the list. Every node in the list points to the next node in the list. Therefore, in a linked list, every node contains data of the node and a pointer to next node. The last node in the list contains a NULL pointer to indicate that it is the end or tail of the list. The total number of nodes that may be added to a list is limited only by the amount of memory available.

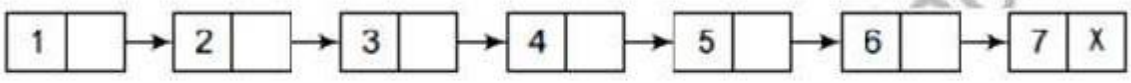


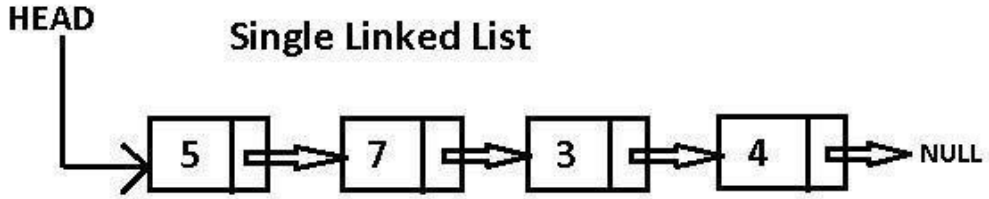
Figure 2.2 Simple linked list

Advantage: Easier to insert or delete data elements
Disadvantage: Slow search operation and requires more memory space
A linked list, in simple terms, is a linear collection of data elements. These data elements are called nodes. Linked list is a data structure which in turn can be used

to implement other data structures. Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations. A linked list can be perceived as a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.

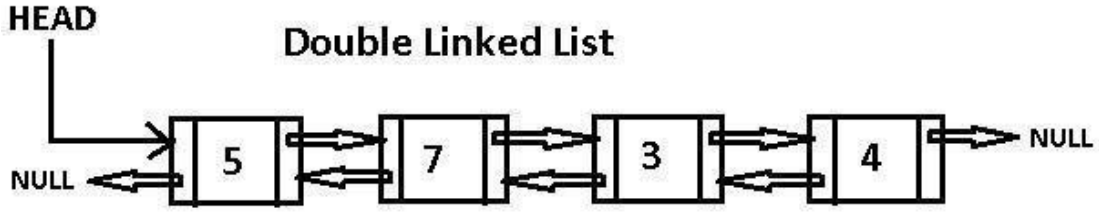
SINGLE LINKED LIST

A single linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node. A singly linked list allows traversal of data only in one way. So these are unidirectional.



DOUBLE LINKED LIST

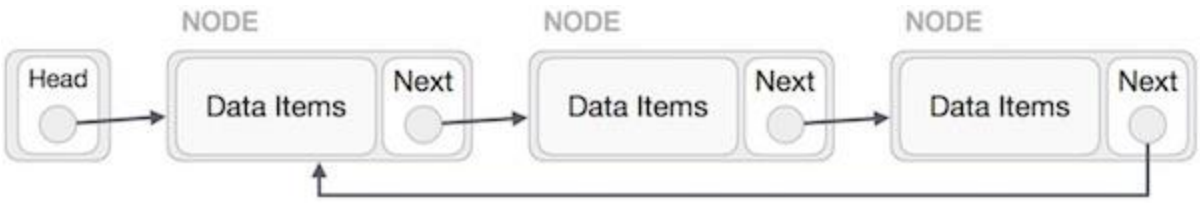
A double linked list is similar to single linked list but has two pointers in the node, where the next pointer points to next node and the previous pointer points to previous node. A double linked list as it has next and previous pointers , we can traverse in both directions, so these are bi-directional.



CIRCULAR LINKED LIST

Singly Linked List as Circular

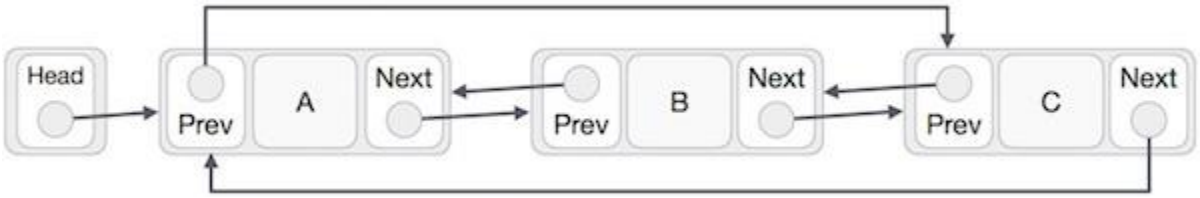
In singly linked list, the next pointer of the last node points to the first node.



The next pointer of the last node points to the first node.

Doubly Linked List as Circular

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



As per the above illustration, following are the important points to be considered.

- The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
- The first link's previous points to the last of the list in case of doubly linked list.

Basic Operations

Following are the important operations supported by a circular list.

- insert − Inserts an element at the start of the list.
- delete − Deletes an element from the start of the list.
- display − Displays the list.

Algorithms for each operation :

```
insertFirst(data):
Begin
  create a new node
  node -> data := data
  if the list is empty, then
    head := node
    next of node = head
```

```
  else
    temp := head
    while next of temp is not head, do
    temp := next of temp
    done
    next of node := head
    next of temp := node
    head := node
  end if
End
```

```
deleteFirst():
Begin
  if head is null, then
    it is Underflow and return
  else if next of head = head, then
    head := null
    deallocate head
  else
    ptr := head
    while next of ptr is not head, do
      ptr := next of ptr
    next of ptr = next of head
    deallocate head
    head := next of ptr
  end if
End
```

```
isplay():
Begin
  if head is null, then
    Nothing to print and return
  else
    ptr := head
    while next of ptr is not head, do
      display data of ptr
      ptr := next of ptr
```

```
    display data of ptr
  end if
End
```

Applications of linked list in computer science –

1. Implementation of stacks and queues
2. Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
3. Dynamic memory allocation : We use linked list of free blocks.
4. Maintaining directory of names
5. Performing arithmetic operations on long integers
6. Manipulation of polynomials by storing constants in the node of linked list
7. representing sparse matrices

Applications of linked list in real world-

1. Image viewer – Previous and next images are linked, hence can be accessed by next and previous button.
2. 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.
3. 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.

Atomic Linked List:

When we develop a linked list we place data member within the node structure, where the data member can be of any primitive data type variable. In this case, we can store that specific data type value into that data member of that node. We may have any number of data members and store data into all data members in to that node.

But in atomic linked list, we don't place the data member of any primitive data type. Instead of primitive data member, we create a union data member within that node. Where the union is created to store elements of all primitive types, and use can use only one at a time. In this case,

The node structure will have an extra variable to remember the type data stored into the member of that union, so that in C, we can use appropriate format specifier to display data.

```
union atomicNode
{
        char cdata;
        int idata;
};
struct node
{
        int type;
        union atomicNode data;
        struct node *next;
};
```

Here in each node, the type of data, data and then next pointer to link to next node are stored into each node.

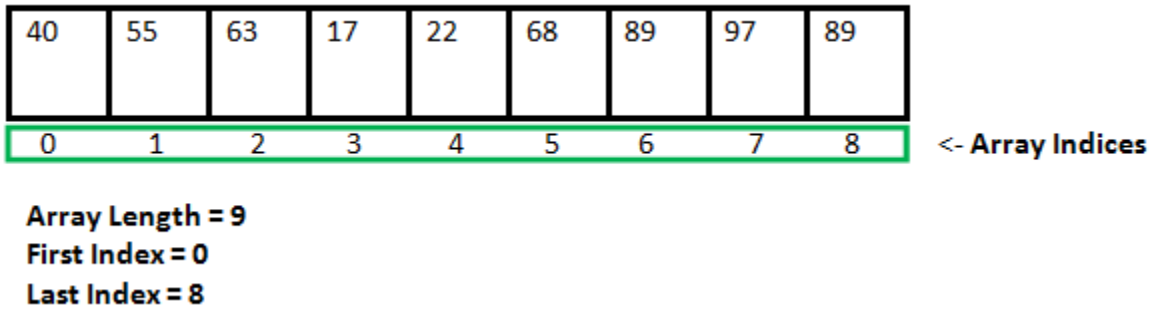
Linked List in Arrays:

A typical linked list includes 2 parts, the data value and the next pointer. The same linked list can be implemented with arrays without using pointers, where the data part is managed by an array and the next elements address is managed by another array.

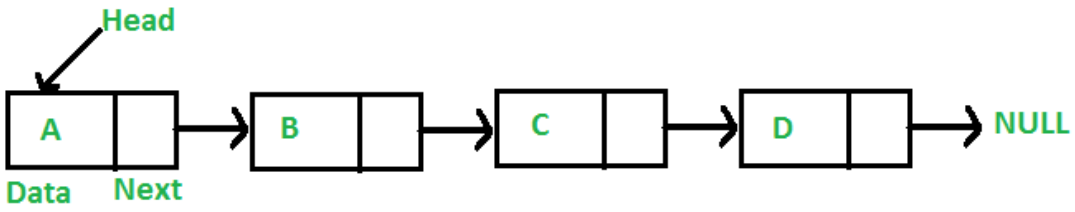
But as the arrays have indexing mechanism to identify each element, it raises burden on system to manage 2 arrays, one to store data and the other to store address and manipulate as a linked list. Instead of this storage of address of element of one array into another and fetching data of one array based on the address stored in same index of second array, we can simply use indexes of first array itself.

Linked List versus Array:

Arrays store elements in contiguous memory locations, resulting in easily calculable addresses for the elements stored and this allows a faster access to an element at a specific index. Linked lists are less rigid in their storage structure and elements are usually not stored in contiguous locations, hence they need to be stored with additional tags giving a reference to the next element. This difference in the data storage scheme decides which data structure would be more suitable for a given situation.



Data storage scheme of an array



Data storage scheme of a linked list

Major differences are listed below:

- Size: Since data can only be stored in contiguous blocks of memory in an array, its size cannot be altered at runtime due to risk of overwriting over other data. However in a linked list, each node points to the next one such that data can exist at scattered (non-contiguous) addresses; this allows for a dynamic size which can change at runtime.
- Memory allocation: For arrays at compile time and at runtime for linked lists. but, dynamically allocated array also allocates memory at runtime.
- Memory efficiency: For the same number of elements, linked lists use more memory as a reference to the next node is also stored along with the data. However, size flexibility in linked lists may make them use less memory overall; this is useful when there is uncertainty about size or there are large variations in

the size of data elements; memory equivalent to the upper limit on the size has to be allocated (even if not all of it is being used) while using arrays, whereas linked lists can increase their sizes step-by-step proportionately to the amount of data.

- Execution time: Any element in an array can be directly accessed with its index; however in case of a linked list, all the previous elements must be traversed to reach any element. Also, better cache locality in arrays (due to contiguous memory allocation) can significantly improve performance. As a result, some operations (such as modifying a certain element) are faster in arrays, while some other (such as inserting/deleting an element in the data) are faster in linked lists.

Following are the points in favour of Linked Lists.

(1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, the upper limit is rarely reached.

(2) Inserting a new element in an array of elements is expensive because a room has to be created for the new elements and to create room existing elements have to be shifted.

For example, suppose we maintain a sorted list of IDs in an array id[ ].

id[ ] = [1000, 1010, 1050, 2000, 2040, …..].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

So Linked list provides the following two advantages over arrays

1) Dynamic size
2) Ease of insertion/deletion

Linked lists have following drawbacks:

1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do a binary search with linked lists.
2) Extra memory space for a pointer is required with each element of the list.
3) Arrays have better cache locality that can make a pretty big difference in performance.
4) It takes a lot of time in traversing and changing the pointers.
5) It will be confusing when we work with pointers.

UNIT III
STACKS

Introduction to Stacks:

A stack is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack. Stack is called a last-in, first-out (LIFO) structure because the last element which is added to the stack is the first element which is deleted from the stack.

A stack supports three basic operations: push, pop, and peep or peek. The push operation adds an element to the top of the stack. The pop operation removes the element from the top of the stack. And the Top or peep operation returns the value of the topmost element of the stack (without deleting it). As the data when represented from zero-th index till the TOP, it looks as the element is deleted.

Stack as an Abstract Data Type:

A Stack can also be defined as ADT. A stack of elements of a particular data type is a finite (limited/ known count) sequence of elements with below specified operations:

Initialize the stack to be empty

Determine whether the stack is empty or not

Determine whether the stack is full or not

If stack is not full, then add an element / node at end of stack called TOP. And this operation is called PUSH

If stack is not empty, then retrieve the element at TOP, called peek / peek

If stack is not empty, then remove an element from TOP, called POP.

Representation of Stacks through Arrays:

In the computer's memory, stacks can be implemented using arrays or linked lists.

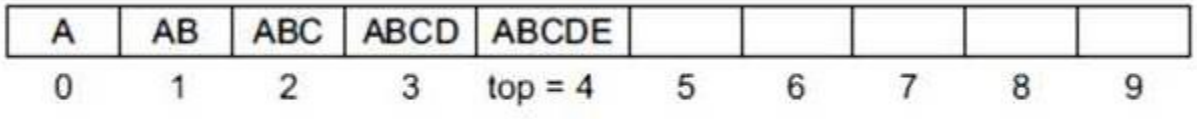


Figure 2.3 Array representation of a stack

The process of storing data onto a stack is called push() and removing data from stack is called pop(). The algorithm for implementing the stack ADT for each operation is as follows:

Algorithm for PUSH:
IF TOP = MAX-1 // if stack is full
PRINT OVERFLOW;
Else
TOP = TOP + 1;
STACK[TOP]=VAL;
End if;
END //End of Push

Algorithm Pop()
IF TOP <= -1 Then // If stack is empty
PRINT UNDERFLOW or EMPTY
Else // if stack not empty
VAL = STACK[TOP]; // move top element in to a val variable
PRINT VAL,"deleted"
TOP = TOP - - OR (TOP-1); // decrement top value by 1.
End if;
END//End of Pop

Implementation of stack through arrays :
#include<stdio.h>
#include<conio.h>

```
void menu();
void push(int);
int pop();
int peek();
void disp();

int stk[5],top=0;

void main()
{
	clrscr();
	menu();
}
void menu()
{
	printf("MENU\n");
	printf("1  PUSH\n");
	printf("2  POP\n");
	printf("3  PEEK\n");
	printf("0  CLOSE\n");

	int ch;
	printf("Enter your choice :");
	scanf("%d",&ch);
	if(ch==0)
	{
		exit(0);
	}
	else if(ch==1)
	{
		int x;
		printf("Enter a number :");
		scanf("%d",&x);
		push(x);
		disp();
	}
	else if(ch==2)
	{
		int x=pop();
		printf("value popped : %d\n",x);
```

```
            disp();
        }
        else if(ch==3)
        {
            printf("Value peeked : %d\n",peek());
            menu();
        }
        else
        {
            printf("Wrong Choice\n");
            menu();
        }
}

void push(int x)
{
        if(top<5)
        {
            stk[top]=x;
            top++;
            printf("pushed %d\n",x);
        }
        else
        {
            printf("stack is full\n");
            menu();
        }
}

int pop()
{
        int x=0;
        if(top>0)
        {
            x=stk[--top];
        }
        else
        {
            printf("Stack is empty\n");
            menu();
```

```
        }
        return x;
}

int peek()
{
        return stk[top-1];
}

void disp()
{
        printf("values on stack are :");
        for(int i=0;i<top;i++)
                printf("%d ", stk[i]);
        printf("\n");
        menu();
}
```

Representation of Stacks through Linked Lists:

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be over flown if the space left in the memory heap is not enough to create a node.

Adding a node to the stack is referred to as push operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose,

assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Deleting a node from the top of stack is referred to as pop operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

1. Check for the underflow condition: The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
2. Adjust the head pointer accordingly: In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Literally, the stack implementation on linked list is very similar to normal linked list, but in normal linked list, the next head holds the first node and each node links to next node and in stack implementation, the top points to last node and the link pointer points to previous node.

Code :

```
#include <stdio.h>
#include<conio.h>
#include<alloc.h>
#include<process.h>
struct Node
{
        int data;
        struct Node* link;
};

struct Node* top;

void push(int data)
{
        struct Node* temp;
        temp = (Node*) malloc(sizeof(Node));
        temp->data = data;
        temp->link = top;
        top = temp;
}
```

```
int isEmpty()
{
	return top == NULL;
}

int peek()
{
	return top->data;
}

void pop()
{
	struct Node* temp;

	if (top == NULL)
	{
		printf("\nStack Underflow");
		exit(1);
	}
	else
	{

		temp = top;
		top = top->link;
		temp->link = NULL;
		free(temp);
	}
}

void display()
{
	struct Node* temp;

	if (top == NULL)
	{
		printf( "\nStack Underflow");
		exit(1);
	}
	else
```

```
	{
		temp = top;
		while (temp != NULL)
		{

			// Print node data
			printf("%d->", temp->data );

			temp = temp->link;
		}
	}
}

// Driver Code
int main()
{
	top=NULL;
	// Push the elements of stack
	clrscr();
	push(11);
	push(22);
	push(33);
	push(44);

	// Display stack elements
	display();

	// Print top element of stack
	printf("\nTop element is %d\n",peek());

	// Delete top elements of stack
	pop();
	pop();

	// Display stack elements
	display();

	printf("\nTop element is %d\n",peek());
	return 0;
}
```

Applications of Stacks

Following is the various Applications of Stack in Data Structure:

- Evaluation of Arithmetic Expressions
- Backtracking
- Delimiter Checking
- Reverse a Data
- Processing Function Calls

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.

Evaluation of expressions based on precedence of operators and converting the expression from infix into either pre-fix notation or post-fix notation can be very flexibly done using stacks.

Backtracking

Backtracking is another application of Stack. It is a recursive algorithm that is used for solving the optimization problem. For example the back button in a browser.

Delimiter Checking

The common application of Stack is delimiter checking, i.e., parsing that involves analyzing a source program syntactically. It is also called parenthesis checking. When the compiler translates a source program written in some programming language such as C, C++ to a machine language, it parses the program into multiple individual parts such as variable names, keywords, etc. By scanning from left to right. The main problem encountered while translating is the unmatched delimiters. We make use of different types of delimiters include the parenthesis checking (,), curly braces {,} and square brackets [,], and common delimiters /* and */. Every opening delimiter must match a closing delimiter, i.e., every opening parenthesis should be followed by a matching closing parenthesis. Also, the delimiter can be nested. The opening delimiter that occurs later in the source program should be closed before those occurring earlier. To perform a delimiter checking, the compiler makes use of a stack. When a compiler translates a source program, it reads the characters one at a time, and if it finds an opening

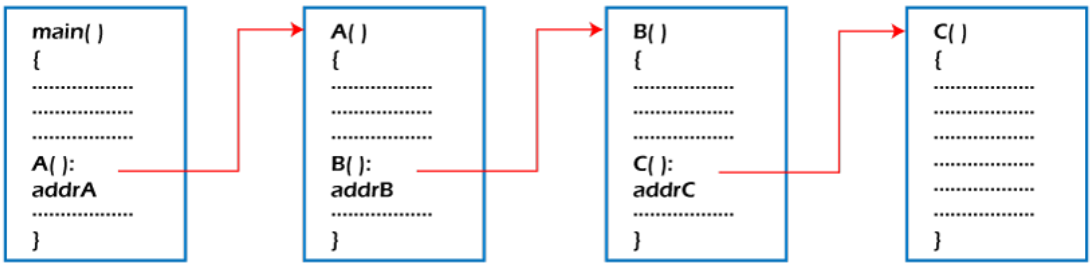
delimiter it places it on a stack. When a closing delimiter is found, it pops up the opening delimiter from the top of the Stack and matches it with the closing delimiter.

Reverse a Data

To reverse a given set of data, we need to reorder the data so that the first and last elements are exchanged, the second and second last element are exchanged, and so on for all other elements.

Processing Function Calls:

Stack plays an important role in programs that call several functions in succession. Suppose we have a program containing three functions: A, B, and C. function A invokes function B, which invokes the function C.



Function call

When we invoke function A, which contains a call to function B, then its processing will not be completed until function B has completed its execution and returned. Similarly for function B and C. So we observe that function A will only be completed after function B is completed and function B will only be completed after function C is completed. Therefore, function A is first to be started and last to be completed. To conclude, the above function activity matches the last in first out behavior and can easily be handled using Stack.

Converting infix expr3ession into either prefix or postfix expressions during evaluation of arithmetic expression:

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#include<string.h>
#include<alloc.h>
```

```
char pop();
void push(char);
int getPriority(char);
void infixToPostfix(char *,char *);
void infixToPrefix(char *,char *);

char stk[40];
int top=-1;
void main()
{
	char inf[40]="a*(b+c)";
	char pre[40],pos[40];
	clrscr();
	printf("infix   : %s\n",inf);
	infixToPostfix(inf,pos);
	printf("postfix : %s\n",pos);

	infixToPrefix(inf,pre);
	printf("prefix  : %s\n",pre);

}

char pop()
{
	char ch=stk[top];
	top--;
	return ch;
}

void push(char ch)
{
	top++;
	stk[top]=ch;
}
```

```
int getPriority(char ch)
{
	switch(ch)
	{
		case '+':
		case '-': return 1;

		case '*':
		case '/': return 2;

		case '^' : return 3;

	}
	return 0;
}

void infixToPostfix(char *inf, char *pos)
{
	int i,ind=0;
	for(i=0 ; i<strlen(inf) ; i++)
	{
		char ch=inf[i];

		if( isalpha(ch))
		{
			pos[ind]=ch;
			ind++;
		}
		else
		{
			if(ch=='(')
				push(ch);

			else if(ch==')')
```

```
                {
                        while(stk[top] !='(')
                        {
                                pos[ind]=pop();
                                ind++;
                        }
                        pop();
                }
                else
                {
                        if(getPriority(stk[top]) < getPriority(ch) )
                                push(ch);
                        else
                        {
                                while(getPriority(stk[top]) > getPriority(ch) )
                                {
                                        pos[ind]=pop();
                                        ind++;
                                }
                                push(ch);

                        }
                }


        }
    }//for

    while(top>-1)
    {
        pos[ind]=pop();
        ind++;
    }
    pos[ind]='\0';
}
```

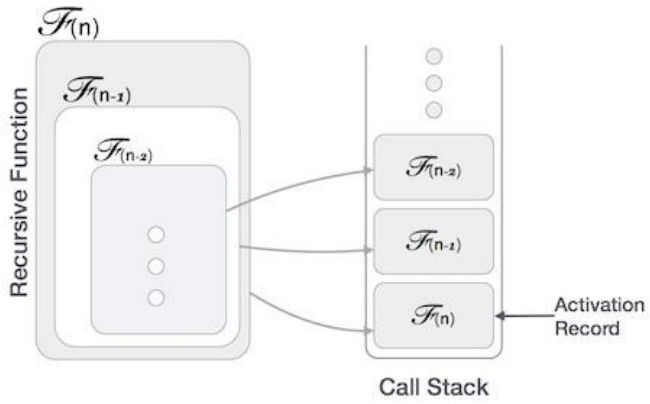
```
void infixToPrefix(char *inf, char *pre)
{
	int i,ind=0;
	for(i=strlen(inf)-1 ; i>-1 ; i--)
	{
		char ch=inf[i];

		if( isalpha(ch))
		{
			pre[ind]=ch;
			ind++;
		}
		else
		{
			if(ch==')')
				push(ch);

			else if(ch=='(')
			{
				while(stk[top] !=')')
				{
					pre[ind]=pop();
					ind++;
				}
				pop();
			}
			else
			{
				if(getPriority(stk[top]) < getPriority(ch) )
					push(ch);
				else
				{
					while(getPriority(stk[top]) > getPriority(ch) )
					{
```

```
                                        pre[ind]=pop();
                                        ind++;
                                }
                                push(ch);

                        }
                }


        }
    }//for

    while(top>-1)
    {
        pre[ind]=pop();
        ind++;
    }
    pre[ind]='\0';
    strrev(pre);
}
```

Stacks and Recursion :

Many programming languages implement recursion by means of stacks. Generally, whenever a function (caller) calls another function (callee) or itself as callee, the caller function transfers execution control to the callee. This transfer process may also involve some data to be passed from the caller to the callee.

This implies, the caller function has to suspend its execution temporarily and resume later when the execution control returns from the callee function. Here, the caller function needs to start exactly from the point of execution where it puts itself on hold. It also needs the exact same data values it was working on. For this purpose, an activation record (or stack frame) is created for the caller function.



This activation record keeps the information about local variables, formal parameters, return address and all information passed to the caller function.

.

The complexity is counted as what amount of extra space is required for a module to execute. In case of iterations, the compiler hardly requires any extra space. The compiler keeps updating the values of variables used in the iterations. But in case of recursion, the system needs to store activation record each time a recursive call is made. Hence, it is considered that space complexity of recursive function may go higher than that of a function with iteration.

But the reason for recursion is, it makes a program more readable and because of latest enhanced CPU systems, recursion is more efficient than iterations.

Ex:

```
int fact(int n){
        if (n==1)
                return 1;
        return n * fact(n-1);
}
```

QUEUES

Introduction:

A queue is a FIFO (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are

added at one end called the REAR and removed from the other end called the FRONT.

Analogies using concept of queues:

People waiting for a bus. The first person standing in the line will be the first one to get into the bus.

Cars lined at a toll bridge. The first car to reach the bridge will be the first to leave.

Taking a Tickets in the Cinema Theater Queue. The First Person will collect the Ticket and Enter into the Hall.

The different operations done on queue data structure are generally called qinsert() or enqueue() and qdelete() or dequeue().

Queue as an Abstract data Type:

A Queue can also be defined as ADT. A queue of elements of a particular data type is a finite (limited/ known count) sequence of elements with below specified operations:

Initialize the queue to be empty

Determine whether the queue is empty or not

Determine whether the queue is full or not

If queue is not full, then add an element / node through the rear pointer of queue. And this operation is called qinsert() / enqueu()

If queue is not empty, then remove an element from front , called qdelete() / deque().

```
procedure Enqueue(element)
    if rear = MAX-1 then      // if queue is full
    call QUEUE_FULL;
    Else                      // queue is not full
    rear=  rear + 1;          //forward rear pointer by 1.
    Q[rear]<- element;        // store element at rear end
    End If;
```

```
procedure Dequeue()
        if front=-1 and rear=-1 then        //if queue empty
        call Queue is Empty;
        else if front==rear                 //if rear and front both are equal
        set rear =-1 and front=-1
        else                                //if queue is not empty
        front=front+1;                       //forward front pointer by 1
        End if;
```

** but a simple queue has a problem that unless it is fully emptied, it cannot be used fully.

Representation of Queues:

Queue implementation with array:

```
int a[5];
int f,r;

void menu();
void qinsert()
void  qdelete()
void disp();

void main()
{
        f=r=0;
        menu();
}
void menu()
{
        int ch;
        printf("MENU\n");
        printf("1  QINSERT\n");
        printf("2  QDELETE\n");

        printf("Enter your choice :");
        scanf("%d",&ch);
```

```
	if(ch==1)
		qinsert();
	else if(ch==2)
		qdelete();

}

void qinsert()
{
	if(r<5)
	{
	printf("Enter a value :");
	scanf("%d",&a[r]);
	r++;
	disp();
	}
	else
	{
		printf("QUEUE IS FULL\n");
	}

}

void qdelete()
{
	if(f!=r)
	{
		f++;
		disp();
	}
	else
	{
		printf("QUEUE is EMPTY\n");
		f=r=0;
		menu();
	}

}
```

```
void disp()
{
	int i;
	for(i=f; i < r  ;i++)
		printf("%d  ", a[i]);
	menu();
}
```

Queue implementation on linked list:

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

struct node{
	int data;
	struct node *n;
};
typedef struct node node;

void qinsert(int);
void qdelete();
void disp();

node *front,*rear;

void main()
{
	front=rear=NULL;
	clrscr();
	qinsert(10);
	qinsert(20);
	qinsert(30);
	qinsert(40);
	disp();
	qdelete();
	qdelete();
	disp();
```

```
}

void qinsert(int data)
{
	node *temp=(node*) malloc( sizeof(node));
	temp->n=NULL;
	temp->data=data;
	printf("inserted %d\n",data);

	if(front==NULL)
	{
		front=temp;
		rear=temp;
	}
	else
	{
		rear->n=temp;
		rear=temp;
	}
}


void qdelete()
{
	node *temp;
	temp=front;
	if(front != rear)
	{
		printf("deleted %d\n",temp->data);
		front=front->n;
	}
	else
		printf("queue is empty");
	 free(temp);
}

void disp(){
	node *temp=front;
	while(temp!=NULL)
	{
```

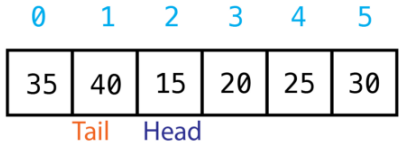
```
            printf("%d ",temp->data);
            temp=temp->n;
        }
        printf("\n");
}
```

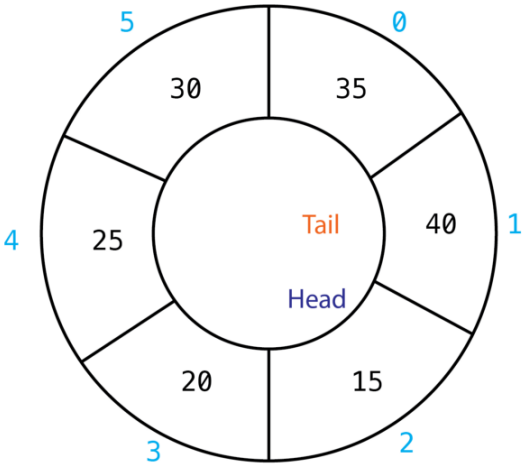
Circular Queues:

A simple queue when implemented, the logic or algorithm is such that, unless the queue is fully emptied, it cannot be used fully. For example if a queue has five elements and 4 elements are stored in that and then 3 are removed, though there are 4 empty places in the queue, the algorithm / logic permits only 1 values to be enqueued.

To make use of the simple queue fully, each time the dequeue (delete queue) is done, all the elements are to be moved to its previous locations. This “moving elements” each time to its previous locations give burden on system.

To overcome this problem, the circular queues are implemented. In this the logic is implemented such that, the rear pointer moves to first location if first is empty so that the queue is logically formed as a circle.






```
Q = circularQueue(6)
Q.Enqueue(5)
Q.Enqueue(10)
Q.Enqueue(15)
Q.Enqueue(20)
Q.Enqueue(25)
Q.Enqueue(30)
Q.Dequeue()
Q.Dequeue()
Q.Enqueue(35)
Q.Enqueue(40)
```

Implementation:

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

void disp();
int que[5], front,rear,status;

void menu();
void qinsert();
void qdelete();



void main()
{
    clrscr();
    menu();
}


void menu()
{
  int ch;
  printf("\n\nMENU");
  printf("\n1 qinsert");
  printf("\n2.qdelete");
  printf("\nEnter any remaing key to exit");
  printf("\nEnter your choice : ");
  scanf("%d",&ch);


  if(ch==1)
        qinsert();
  else if(ch==2)
        qdelete();
}

void qinsert()
{
```

```
        if(rear==5)
                if(front!=0)
                        rear=0;

  if(rear<5 && rear!=front || front==0 && rear==0 )
  {
            printf("Enter element : ");
            scanf("%d",&que[rear]);
            rear++;
            status=1;
            disp();
  }
   else
   {
      printf("\nQUEUE is FULL");
      menu();
   }
}

void qdelete()
{
    if(front>=5)
          front=0;
     if(front!=rear || status==1)
     {
          front++;
          status=0;
     }

     if(front==rear)
     {
          printf("\nQUEUE is empty");
          front=rear=0;
          menu();
     }
     else
         disp();
}

void disp()
```

```
{
  int i;

   printf("\nElements in QUEUE : ");
   if(front>=rear)
   {
        for(i=front; i<5; i++)
           printf("%d  ",que[i]);
        for(i=0; i<rear; i++)
           printf("%d  ",que[i]);
   }
   else
        for(i=front; i<rear; i++)
          printf("%d  ",que[i]);

   menu();
}
```

Applications of queues :

Queue is used when things don't have to be processed immediately, but have to be processed in First In First Out order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

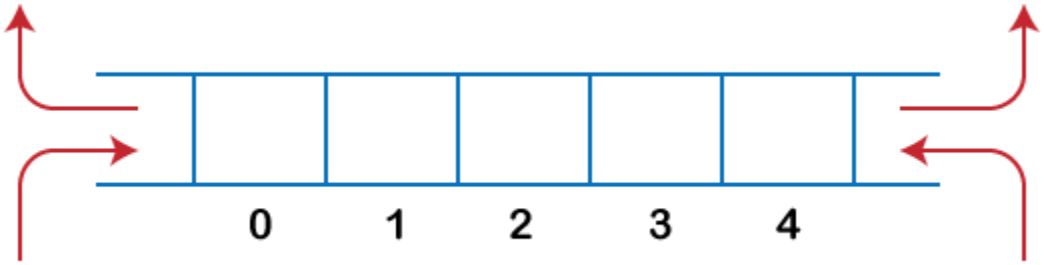
1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.
3) In Operating systems:
    a) Semaphores
    b) FCFS (first come first serve) scheduling, example: FIFO queue
    c) Spooling in printers
    d) Buffer for devices like keyboard
4) In Networks:
    a) Queues in routers/ switches
    b) Mail Queues
5) Variations: (Deque, Priority Queue, Doubly Ended Priority Queue)

6) Memory Management: The unused memory locations in the case of ordinary queues can be utilized in circular queues.
7) Traffic system: In computer controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
8) CPU Scheduling: Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

Double Ended Queues-Deques:

In the queue, the insertion takes place from one end while the deletion takes place from another end. The end at which the insertion occurs is known as the rear end whereas the end at which the deletion occurs is known as front end.

The dequeue stands for Double Ended Queue.



Deque is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

In DEQUEUEs, the insert can be done either from front or from rear and also the deleting can be done either from front or rear.

But the DEQUEUE operation is not considered as a standard QUEUE operation, as the dequeues break the basic FIFO rule of queues.

The following are the six functions that we have used in the below program:

- enqueue_front(): It is used to insert the element from the front end.
- enqueue_rear(): It is used to insert the element from the rear end.
- dequeue_front(): It is used to delete the element from the front end.
- dequeue_rear(): It is used to delete the element from the rear end.
- getfront(): It is used to return the front element of the deque.
- getrear(): It is used to return the rear element of the deque.

Priority Queues:

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

The characteristics are :

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

If the values of the priority queue are 1, 3, 4, 8, 14, 22

All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

- poll(): This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- add(2): This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- poll(): It will remove '2' element from the priority queue as it has the highest priority queue.

- add(5): It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

Implementation of Priority Queue

The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient way of implementing the priority queue, so we will implement the priority queue using a heap data structure in this topic. Now, first we understand the reason why heap is the most efficient way among all the other data structures.

Applications of priority queues:

When the graph is stored in the form of adjacency list or matrix, priority queue can be used to extract minimum efficiently when implementing Dijkstra's algorithm.

Prim's algorithm: It is used to implement Prim's Algorithm to store keys of nodes and extract minimum key node at every step.

Data compression : It is used in Huffman codes which is used to compresses data.

Artificial Intelligence : A* Search Algorithm : The A* search algorithm finds the shortest path between two vertices of a weighted graph, trying out the most promising routes first. The priority queue (also known as the fringe) is used to keep track of unexplored routes, the one for which a lower bound on the total path length is smallest is given highest priority.

Heap Sort : Heap sort is typically implemented using Heap which is an implementation of Priority Queue.

Operating systems: It is also use in Operating System for load balancing (load balancing on server), interrupt handling.

UNIT –IV
BINARY TREES

Introduction to Non- Linear Data Structures:

The data structures are 2 types, linear and non-linear. The arrays, linked lists are linear data structures and the trees and graphs are non-linear data structures.

The trees are also implemented as doubly linked lists, but the nodes here are related with each other in a parent – child relationship.

trees.....

tree is also a data structure.... like array / linked list etc....

arrays VS LL & TREE :

in array when we store data, each entity is refered with a term "ELEMENT".

in LL the same data storage locaiton is not refered as element, rather it is called a NODE.

node is something used to store data and have link to next / previous node.

in arrays, as all elements have sequenential memory and have index numbers to refer each element.

but in case of LL or a tree, the memory is not sequntial, and have no indexing , so each node will have a link to next node.

LL VS tree :

LL is linear and tree is non-linear

a LL can be uni-directional(single LL) or bi directional (doubly LL)
but trees are always unidirectional....

in LL, each NODE will have link to "next" or "previous" node.
in a tree... each node can have link to its "CHILD" node.

the first node in LL is called HEAD
the first node in tree is called ROOT

---

general terminalogy in trees :

the first node is called ROOT and it can have child nodes....

one child node can have further child nodes. if it has further child nodes, then it is a child node and also a parent node.

-> root node is always.... a parent node....
but not all parents are roots.

the child node that FURTHER has no child nodes is called a LEAF.

the leaf nodes (that have no further child nodes) are called external nodes.....

the nodes that have child nodes are called internal nodes.....

EDGE : the link / path between two nodes is called EDGE

PATH : sequence of edges between given nodes

DEGREE : no. of immediate children to a parent is called degree....

SIBLINGS : another child node of its parent node.

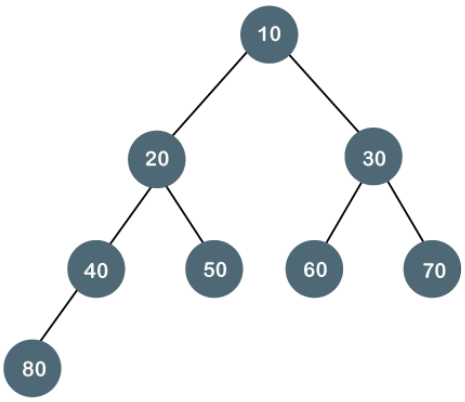
LEVEL : the nodes at same depth from root are said to be at same level, considering the root is at LEVEL 0.

Introduction Binary Trees,

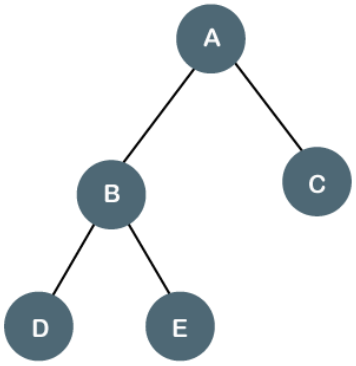
Tree : is a DS, developed using structure nodes that hold data and links.
it is something like a double linked list... where the double LL is a linear DS and tree is non-linear.

In Double Linked List, each node has two pointers, the previous and next. the previous pointer points to previous node and the next pointer points to next node. whereas in trees, the pointers are called left and right, and these pointer point to left child and right child. In a tree a root / parent node can have any number of children. But in DS, we use a binary tree. A tree can have any children, but the BT can have a maximum of 2 children.

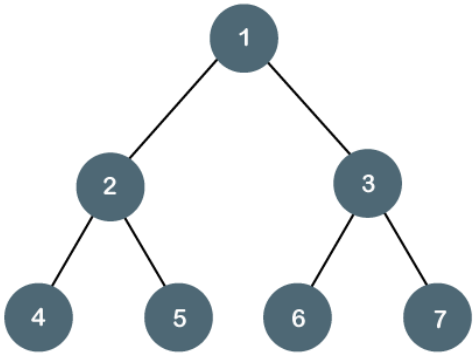
the binary trees are classified into following.....
Types of Trees,
-> full BT: the parent can have either 0 child or 1 child or 2 children
ex :



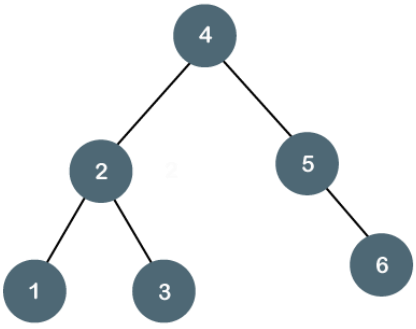
-> complete BT : (is also a FBT) the parent can have either 0 child or 2 children.... and all children filled from LEFT....
ex :



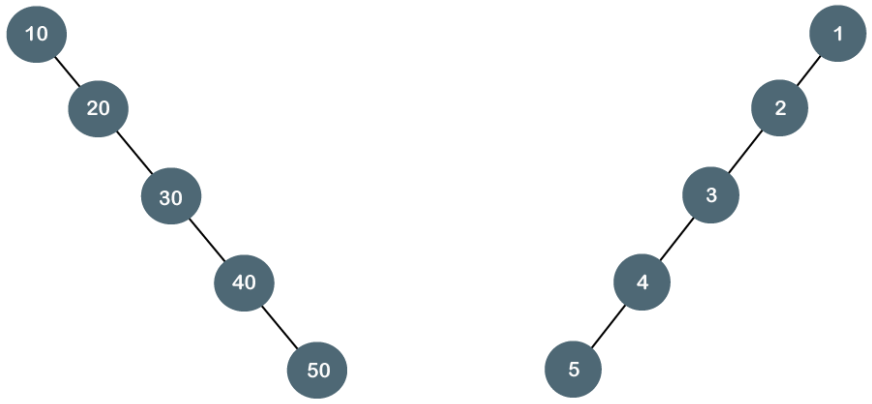
-> perfect BT : (is a FBT), the parent can have either 0 or 2 children, all children filled from LEFT and all LEAVES (leaf nodes) must be at one level.
ex :



-> balanced BT : the deapth / height / level difference of two leaves must not be more than 1.



-> degenerated BT : each parent has only 1 child or 0 child. it can be either left skewed or right skewed or un-skewed



-> Binary Search Tree : (BST) : it need not be a perfect BT or CBT.... but if teh values are stored as SORTED in the BT, then it is caleld BST. it is used for fast searching of data. the data is stored sorted (in inorder traversal). we dont sort and store into BST. while storing itslef, the data is sorted and stored.

-> Threaded Binary Tree (TBT) and heap trees, B- trees, B+ trees, red-black trees, AVL trees etc.

Properties of Binary Trees:

A binary tree is a hierarchal data structure in which each node has at most two children. The child nodes are called the left child and the right child. The linked list representation of a binary tree in which each node has three fields:

- Pointer to store the address of the left child
- Data element
- Pointer to store the address of the right child

1. A binary tree can have a maximum of 2^n nodes at level n if the level of the root is zero.
2. When each node of a binary tree has one or two children, the number of leaf nodes (nodes with no children) is one more than the number of nodes that have two children.
3. There exists a maximum of $(2^h - 1)$ nodes in a binary tree if its height is h.
4. If there exist L leaf nodes in a binary tree, then it has at least L=1 levels.
5. A binary tree of “n” nodes has log(n+1) minimum number of levels or minimum height.

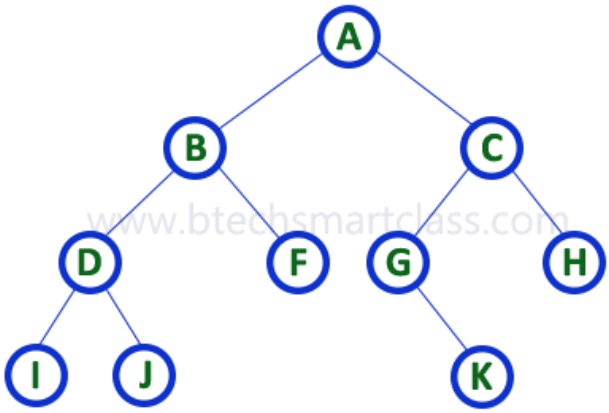
6. The minimum and the maximum height of a binary tree having “n” nodes are $[\log_2 n]$ and “n” respectively.
7. A binary tree of “n” nodes has (n+1) null references

Representation of Binary Trees:

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 of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

Consider the above example of a binary tree and it is represented as follows...

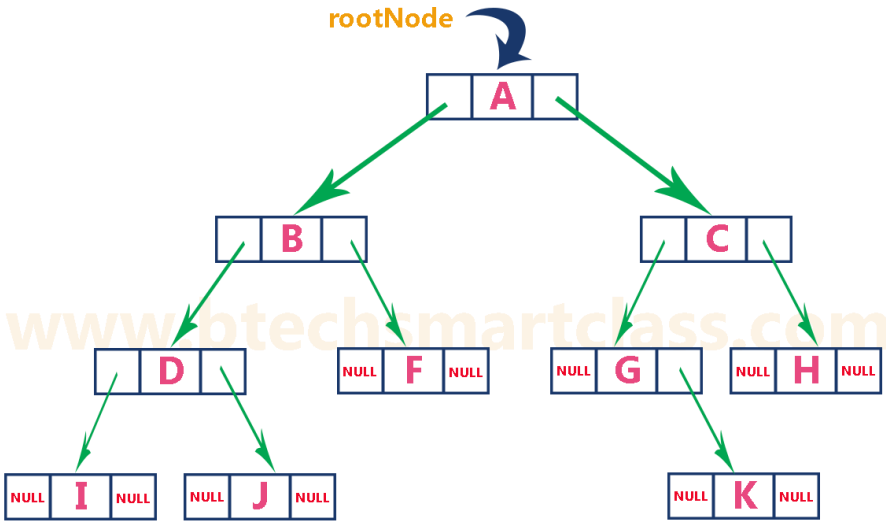
A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of 2n + 1.

2. Linked List Representation of Binary Tree

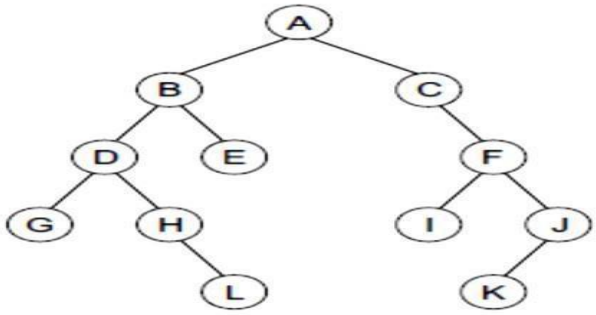
We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address. In this linked list representation, a node has the following structure...

The above example of the binary tree represented using Linked list representation is shown as follows...



Binary Tree Traversal:

Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way. Unlike linear data structures in which the elements are traversed sequentially, tree is a on linear data structure in which the elements can be traversed in many different ways. There are different algorithms for tree traversals. These algorithms differ in the order in which the nodes are visited.



(a)

For above binary tree…

Pre-order Traversal :
To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node. The algorithm works by:

1. Visiting the root node,
2. Traversing the left sub-tree, and finally
3. Traversing the right sub-tree.

First , the left sub-tree next, and then the right sub-tree. Pre-order traversal is also called as depth-first traversal. The word 'pre' in the pre-order specifies that the root node is accessed prior to any other nodes in the left and right sub-trees. Pre-order traversal algorithms are used to extract a prefix notation from an expression tree.

Algorithm

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:          Write TREE -> DATA
Step 3:          PREORDER(TREE -> LEFT)
Step 4:          PREORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```

PRE ORDER TRAVERSAL:

A, B, D, G, H, L, E, C, F, I, J, and K

In-order Traversal :
To traverse a non-empty binary tree in in-order, the following operations are performed recursively at each node. The algorithm works by:

1. Traversing the left sub-tree,
2. Visiting the root node, and finally
3. Traversing the right sub-tree.

Algorithm

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            INORDER(TREE -> LEFT)
Step 3:            Write TREE -> DATA
Step 4:            INORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```

IN ORDER TRAVERSAL:
G, D, H, L, B, E, A, C, I, F, K, and J

Post-order Traversal :
To traverse a non-empty binary tree in post-order, the following operations are performed recursively at each node. The algorithm works by:

1.Traversing the left sub-tree,
2.Traversing the right sub-tree, and finally
3.Visiting the root node.

The word ‘post’ in the post-order specifies that the root node is accessed after the left and the right sub-trees. Post

ALGORITHM

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            POSTORDER(TREE -> LEFT)
Step 3:            POSTORDER(TREE -> RIGHT)
Step 4:            Write TREE -> DATA
       [END OF LOOP]
Step 5: END
```

POST ORDER TRAVERSAL:

G, L, H, D, E, B, I, K, J, F, C, and A

Code for BT traversal:

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

struct node
{
        int data;
        struct node *left,*right;
};

typedef struct node node;

node * create();
void preorder(node *);
void inorder(node *);
void postorder(node *);

void main()
{
        clrscr();
        node * root;
        root=create();
        printf("Pre order :");
        preorder(root);
```

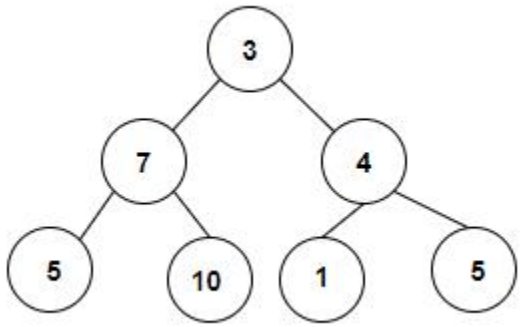
```
	printf("\nin order :");
	inorder(root);
	printf("\npost order :");
	postorder(root);

}

node* create()
{
	int data;
	node * newnode;

	printf("\nEnter a value (press -1 to stop) : ");
	scanf("%d",&data);
	if(data== -1)
		return NULL;


	newnode=(node*) malloc(sizeof(node));
	newnode->left=newnode->right=NULL;


	newnode->data=data;

	printf("Enter value for left child for %d :",data);
	newnode->left=create();

	printf("Enter value for right child for %d :",data);
	newnode->right=create();

	return newnode;
}

void preorder(node *root)
{
	if(root==NULL)
		return;

	printf("%d ",root->data);
	preorder(root->left);
```

```
        preorder(root->right);
}
void postorder(node *root)
{
        if(root==NULL)
                return;
        postorder(root->left);
        postorder(root->right);
        printf("%d ",root->data);

}
void inorder(node *root)
{
        if(root==NULL)
                return;

        inorder(root->left);
        printf("%d ",root->data);
        inorder(root->right);
}
```

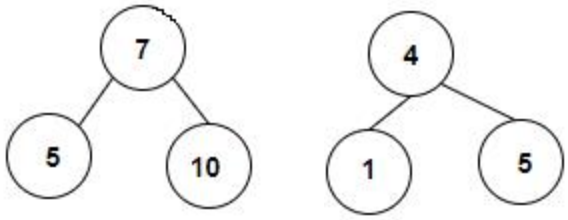
Counting Number of Binary Trees:

We are given a binary tree as input. The goal is to find the number of binary search trees (BSTs) present as subtrees inside it. A BST is a binary tree with left child less than root and right child more than the root.

The tree which will be created after inputting the values is given below

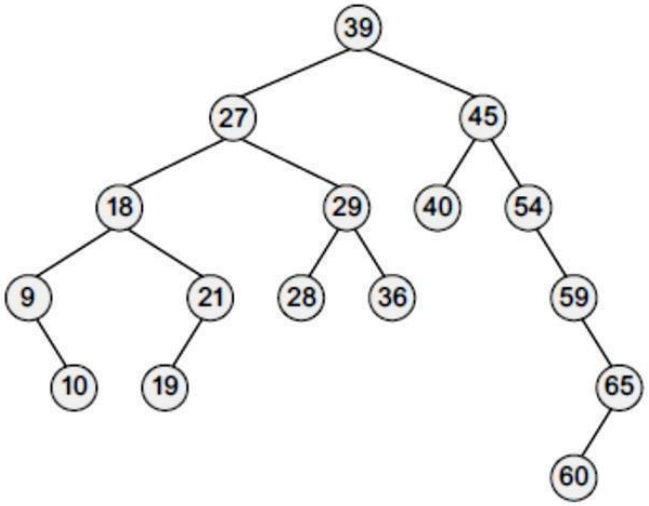


Count of the Number of Binary Search Trees present in above Binary Tree are: 2



Operations on a Binary Search Tree:

In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node. Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node. The same rule is applicable to every sub-tree in the tree.



The root node is 39. The left sub-tree of the root node consists of nodes 9, 10, 18, 19, 21, 27, 28, 29, and 36. All these nodes have smaller values than the root node. The right sub-tree of the root node consists of nodes 40, 45, 54, 59, 60, and 65. Recursively, each of the sub-trees also obeys the binary search tree constraint. Since the nodes in a binary search tree are ordered, the time needed to search an element in the tree is greatly reduced. Whenever we search for an element, we do not need to traverse the entire tree. At every node, we get a hint regarding which sub-tree to search in.

For example, in the given tree, if we have to search for 29, then we know that we have to scan only the left sub-tree. If the value is present in the tree, it will only be in the left sub-tree, as 29 is smaller than 39 (the root node's value). Binary search trees also speed up the insertion and deletion operations.

OPERATIONS ON BINARY SEARCH TREES:

Inserting a New Node in a Binary Search Tree

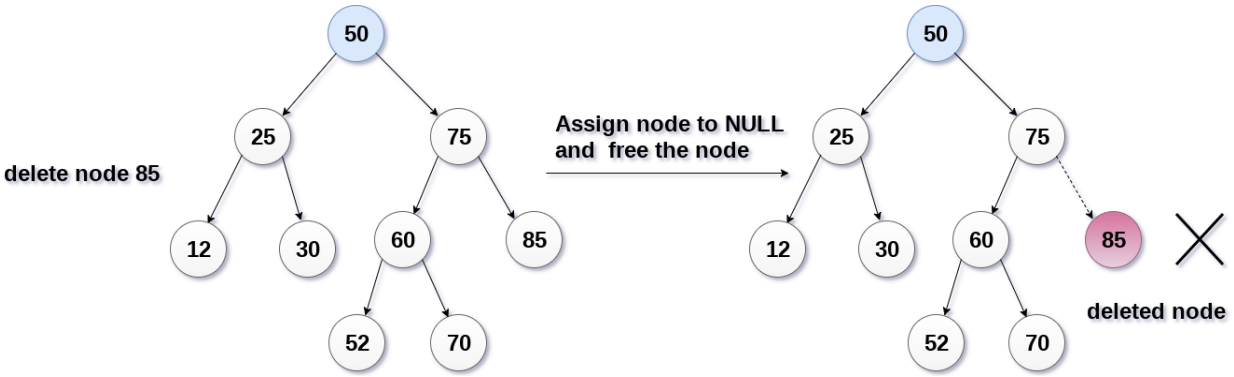
The insert function is used to add a new node with a given value at the correct position in the binary search tree. First find the correct position where the insertion has to be done and then add the node at that position. In Step 1 of the algorithm, the insert function checks if the current node of TREE is NULL. If it is NULL, the algorithm simply adds the node else If the current node's value is less than that of the new node, then the right sub-tree is traversed, else the left sub-tree is traversed. The insert function continues moving down the levels of a binary tree until it reaches a leaf node.

```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
            IF VAL < TREE -> DATA
                  Insert(TREE -> LEFT, VAL)
            ELSE
                  Insert(TREE -> RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: END
```

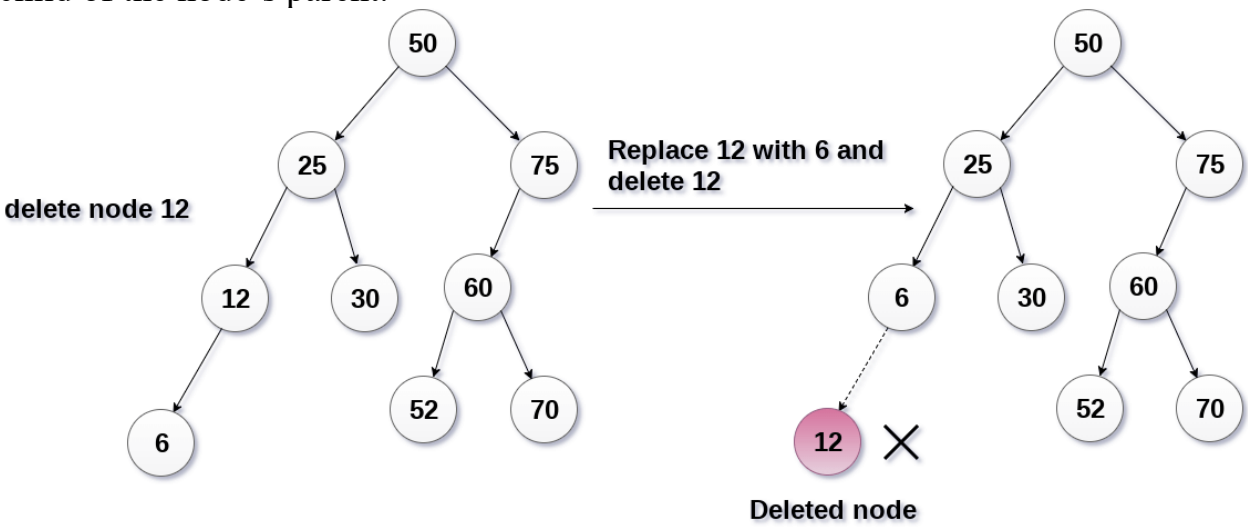
Deleting a Node from a Binary Search Tree

The delete function deletes a node from the binary search tree.

Case 1: Deleting a Node that has No Children If we have to delete node that has no child , we can simply remove this node without any issue. This is the simplest case of deletion.



Case 2: Deleting a Node with One Child .To handle this case, the node's child is set as the child of the node's parent. In other words, replace the node with its child. Now, if the node is the left child of its parent, the node's child becomes the left child of the node's parent.



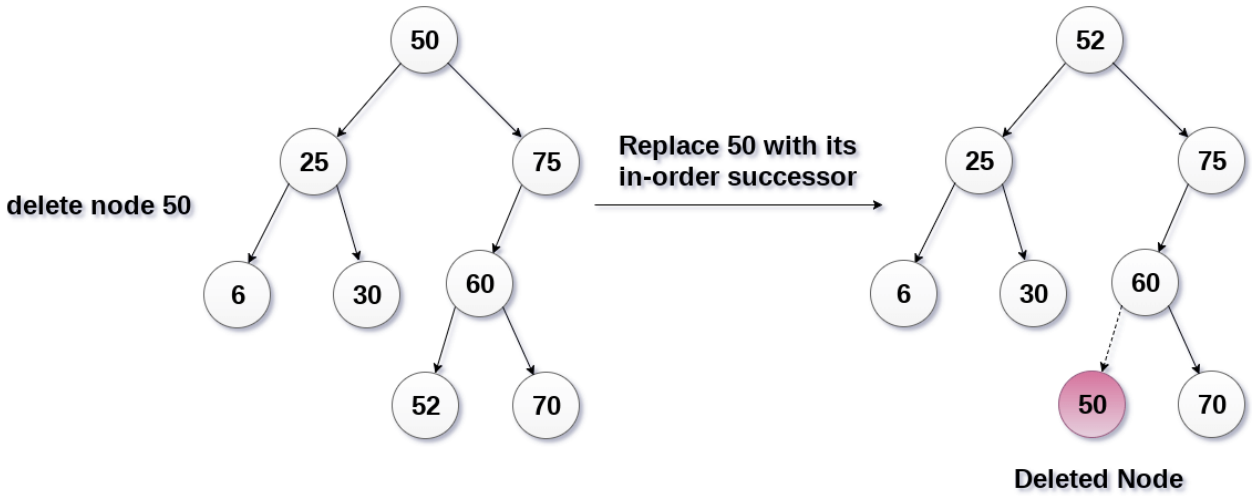
Case 3: The node to be deleted has two children.

It is a bit complexed case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.

In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.

6, 25, 30, 50, 52, 60, 70, 75.

replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



Algorithm

Delete (TREE, ITEM)

- Step 1: IF TREE = NULL

  Write "item not found in the tree" ELSE IF ITEM < TREE -> DATA

  Delete(TREE->LEFT, ITEM)

  ELSE IF ITEM > TREE -> DATA

  Delete(TREE -> RIGHT, ITEM)

  ELSE IF TREE -> LEFT AND TREE -> RIGHT

  SET TEMP = findLargestNode(TREE -> LEFT)

  SET TREE -> DATA = TEMP -> DATA

  Delete(TREE -> LEFT, TEMP -> DATA)

  ELSE

  SET TEMP = TREE

  IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL

  SET TREE = NULL

  ELSE IF TREE -> LEFT != NULL

  SET TREE = TREE -> LEFT

  ELSE

  SET TREE = TREE -> RIGHT

  [END OF IF]

  FREE TEMP

  [END OF IF]

- Step 2: END

Code for BST traversal and inserting value and deleting value:

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

struct btnode
{
   int value;
   struct btnode *l;
   struct btnode *r;
}*root = NULL, *temp = NULL, *t2, *t1;

void delete1();
void insert();
void delete();
void inorder(struct btnode *t);
void create();
void search(struct btnode *t);
void preorder(struct btnode *t);
void postorder(struct btnode *t);
void search1(struct btnode *t,int data);
int smallest(struct btnode *t);
int largest(struct btnode *t);

int flag = 1;

void main()
{
   int ch;

   printf("\nOPERATIONS ---");
   printf("\n1 - Insert an element into tree\n");
   printf("2 - Delete an element from the tree\n");
   printf("3 - Inorder Traversal\n");
   printf("4 - Preorder Traversal\n");
   printf("5 - Postorder Traversal\n");
   printf("6 - Exit\n");
   while(1)
   {
        printf("\nEnter your choice : ");
```

```
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            insert();
            break;
        case 2:
            delete();
            break;
        case 3:
            inorder(root);
            break;
        case 4:
            preorder(root);
            break;
        case 5:
            postorder(root);
            break;
        case 6:
            exit(0);
        default :
            printf("Wrong choice, Please enter correct choice  ");
            break;
        }
    }
}

/* To insert a node in the tree */
void insert()
{
    create();
    if (root == NULL)
        root = temp;
    else
        search(root);
}

/* To create a node */
void create()
{
```

```
    int data;

    printf("Enter data of node to be inserted : ");
    scanf("%d", &data);
    temp = (struct btnode *)malloc(1*sizeof(struct btnode));
    temp->value = data;
    temp->l = temp->r = NULL;
}

/* Function to search the appropriate position to insert the new node */
void search(struct btnode *t)
{
    if ((temp->value > t->value) && (t->r != NULL))      /* value more than root
node value insert at right */
        search(t->r);
    else if ((temp->value > t->value) && (t->r == NULL))
        t->r = temp;
    else if ((temp->value < t->value) && (t->l != NULL))    /* value less than root
node value insert at left */
        search(t->l);
    else if ((temp->value < t->value) && (t->l == NULL))
        t->l = temp;
}

/* recursive function to perform inorder traversal of tree */
void inorder(struct btnode *t)
{
    if (root == NULL)
    {
        printf("No elements in a tree to display");
        return;
    }
    if (t->l != NULL)
        inorder(t->l);
    printf("%d -> ", t->value);
    if (t->r != NULL)
        inorder(t->r);
}

/* To check for the deleted node */
```

```
void delete()
{
   int data;

   if (root == NULL)
   {
        printf("No elements in a tree to delete");
        return;
   }
   printf("Enter the data to be deleted : ");
   scanf("%d", &data);
   t1 = root;
   t2 = root;
   search1(root, data);
}

/* To find the preorder traversal */
void preorder(struct btnode *t)
{
   if (root == NULL)
   {
        printf("No elements in a tree to display");
        return;
   }
   printf("%d -> ", t->value);
   if (t->l != NULL)
        preorder(t->l);
   if (t->r != NULL)
        preorder(t->r);
}

/* To find the postorder traversal */
void postorder(struct btnode *t)
{
   if (root == NULL)
   {
        printf("No elements in a tree to display ");
        return;
   }
   if (t->l != NULL)
```

```
        postorder(t->l);
    if (t->r != NULL)
        postorder(t->r);
    printf("%d -> ", t->value);
}

/* Search for the appropriate position to insert the new node */
void search1(struct btnode *t, int data)
{
    if ((data>t->value))
    {
        t1 = t;
        search1(t->r, data);
    }
    else if ((data < t->value))
    {
        t1 = t;
        search1(t->l, data);
    }
    else if ((data==t->value))
    {
        delete1(t);
    }
}

/* To delete a node */
void delete1(struct btnode *t)
{
    int k;

    /* To delete leaf node */
    if ((t->l == NULL) && (t->r == NULL))
    {
        if (t1->l == t)
        {
            t1->l = NULL;
        }
        else
        {
            t1->r = NULL;
```

```
    }
    t = NULL;
    free(t);
    return;
}

/* To delete node having one left hand child */
else if ((t->r == NULL))
{
    if (t1 == t)
    {
       root = t->l;
       t1 = root;
    }
    else if (t1->l == t)
    {
       t1->l = t->l;

    }
    else
    {
       t1->r = t->l;
    }
    t = NULL;
    free(t);
    return;
}

/* To delete node having right hand child */
else if (t->l == NULL)
{
    if (t1 == t)
    {
       root = t->r;
       t1 = root;
    }
    else if (t1->r == t)
       t1->r = t->r;
    else
       t1->l = t->r;
```

```
        t = NULL;
        free(t);
        return;
    }

    /* To delete node having two child */
    else if ((t->l != NULL) && (t->r != NULL))
    {
        t2 = root;
        if (t->r != NULL)
        {
            k = smallest(t->r);
            flag = 1;
        }
        else
        {
            k =largest(t->l);
            flag = 2;
        }
        search1(root, k);
        t->value = k;
    }

}

/* To find the smallest element in the right sub tree */
int smallest(struct btnode *t)
{
    t2 = t;
    if (t->l != NULL)
    {
        t2 = t;
        return(smallest(t->l));
    }
    else
        return (t->value);
}

/* To find the largest element in the left sub tree */
int largest(struct btnode *t)
```

```
{
   if (t->r != NULL)
   {
        t2 = t;
        return(largest(t->r));
   }
   else
        return(t->value);
}
```

Applications of Binary Tree

Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:
   file system.

2. We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Log n) for insertion/deletion.
3. Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes as nodes are linked using pointers.

Other Applications :

1. Store hierarchical data, like folder structure, organization structure, XML/HTML data.
2. Binary Search Tree is a tree that allows fast search, insert, delete on a sorted data. It also allows finding closest item
3. Heap is a tree data structure which is implemented using arrays and used to implement priority queues.
4. B-Tree and B+ Tree : They are used to implement indexing in databases.
5. Syntax Tree: Used in Compilers.
6. K-D Tree: A space partitioning tree used to organize points in K dimensional space.
7. Trie : Used to implement dictionaries with prefix lookup.
8. Suffix Tree : For quick pattern searching in a fixed text.

9. Spanning Trees and shortest path trees are used in routers and bridges respectively in computer networks
10. As a workflow for compositing digital images for visual effects.

UNIT V

Sorting – An Introduction

Sorting is the process of arranging things in a specific order. It becomes easy to identify required data if the data is sorted. There are different sorting algorithms existing each used in a specific situation.

The sorting are classified into two types. Internal sorting and External sorting. The sorting in which the data is swapped and stored within the reserved memory are called internal sorting and those that cannot accommodate data within the reserved memory during sorting are called external sorting.

Bubble Sort :

In this the heavier data values are moved down and the lighter values move up as an air bubble in the water, so it is called bubble sort.

In this each element is compared with its next element and the heavier element moved to later indexes and lighter elements move to previous indexes.

Algorithm:

```
begin BubbleSort(list)

  for all elements of list
    if list[i] > list[i+1]
      swap(list[i], list[i+1])


end BubbleSort
```

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

```
void bubbleSort(int[],int);
void main()
{
	int a[]={5,12,7,4,1,3};
	int i;
	int n=sizeof(a)/sizeof(int);
	clrscr();
	bubbleSort(a,n);
	for(i=0;i<n;i++)
		printf("%d ",a[i]);

}

void bubbleSort(int a[],int n)
{
	int i,j,t;
	for(i=0;i<n-1;i++)
	for(j=0;j<n-i-1;j++)
		if(a[j]>a[j+1])
		{
			t=a[j];
			a[j]=a[j+1];
			a[j+1]=t;
		}
}
```

Selection sort:

In this sorting, each element is selected and that selected element is compared with every element from its next element till end, and the smallest element is stored into selected indexed element.

```
begin selectionSort(list)

        select each element of list
        compare with all elements from next element
    if list[selected index] > list[comparing index]
          swap(list[selected index], list[comparing index])
```

```
end SelectionSort
```

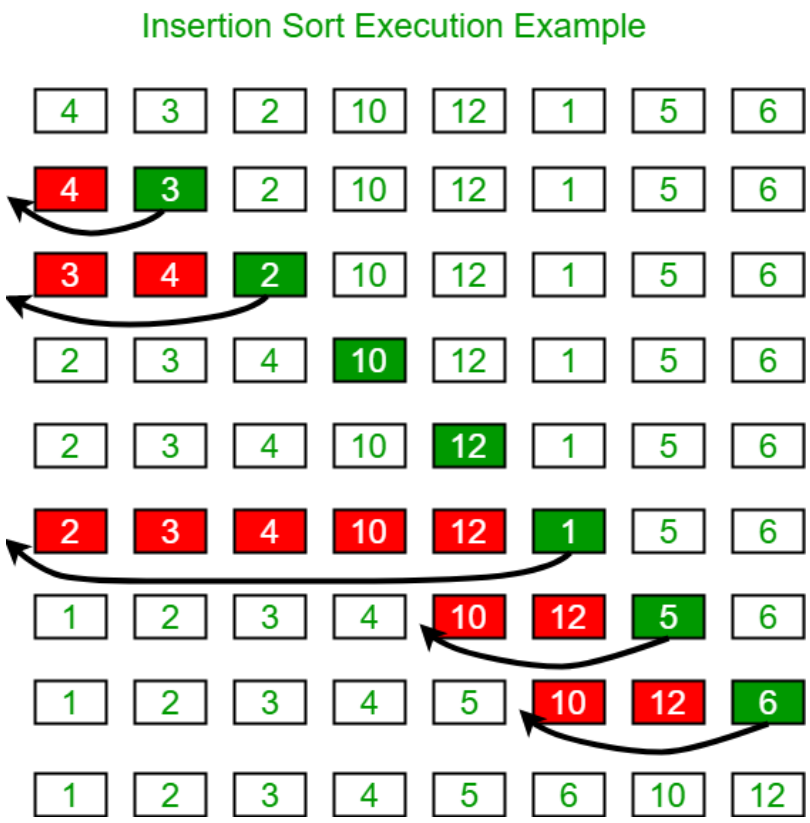
```
#include<stdio.h>
#include<conio.h>

void selectionSort(int[],int);

void main()
{
	int a[]={5,12,7,4,1,3};
	int i;
	int n=sizeof(a)/sizeof(int);
	clrscr();
	selectionSort(a,n);
	for(i=0;i<n;i++)
		printf("%d ",a[i]);

}


void selectionSort(int a[],int n)
{
	int i,j,t;
	for(i=0;i<n-1;i++)
	for(j=i+1;j<n;j++)
		if(a[i]>a[j])
		{
			t=a[i];
			a[i]=a[j];
			a[j]=t;
		}
}
```

Insertion Sort :

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and

an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm
To sort an array of size n in ascending order:
1: Iterate from arr[1] to arr[n] over the array.
2: Compare the current element (key) to its predecessor.
3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.




```
// C program for insertion sort
#include <math.h>
#include <stdio.h>

/* Function to sort an array using insertion sort*/

void insertionSort(int arr[], int n)
{
```

```
        int i, key, j;
        for (i = 1; i < n; i++)
         {
                key = arr[i];
                j = i - 1;

                /* Move elements of arr[0..i-1], that are
                greater than key, to one position ahead
                of their current position */

                while (j >= 0 && arr[j] > key) {
                        arr[j + 1] = arr[j];
                        j -- ;
                }
                arr[j + 1] = key;
        }
}

// A utility function to print an array of size n
void printArray(int arr[], int n)
{
        int i;
        for (i = 0; i < n; i++)
                printf("%d ", arr[i]);
        printf("\n");
}

/* Driver program to test insertion sort */
int main()
{
        int arr[] = { 12, 11, 13, 5, 6 };
        int n = sizeof(arr) / sizeof(arr[0]);

        insertionSort(arr, n);
        printArray(arr, n);
        return 0;
}
```

Quick sort :

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

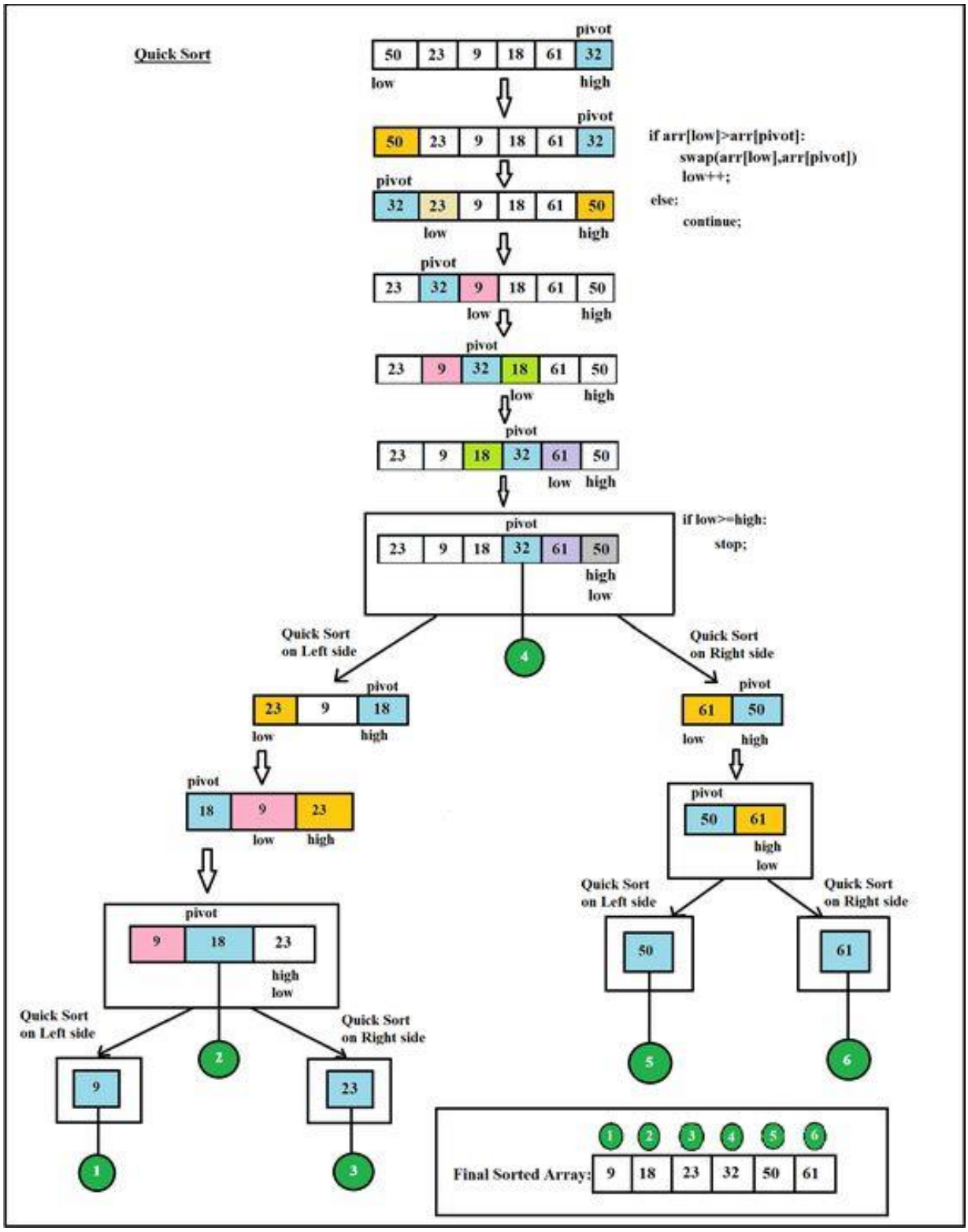
Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are $O(n^2)$, respectively.

Quick sort algorithm:

Step 1 − Make the right-most index value pivot
Step 2 − partition the array using pivot value
Step 3 − quicksort left partition recursively
Step 4 − quicksort right partition recursively

Algorithm for partitioning:

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot


Quick Sort
pivot
50 23 9 18 61 32
low
high
pivot
50 23 9 18 61 32
if arr[low]>arr[pivot]:
swap(arr[low],arr[pivot])
low++;
else:
continue;
pivot
32 23 9 18 61 50
low
high
pivot
23 32 9 18 61 50
low
high
pivot
23 9 32 18 61 50
low
high
pivot
23 9 18 32 61 50
low high
pivot
23 9 18 32 61 50
if low>=high:
stop;
high
low
Quick Sort
on Left side
Quick Sort
on Right side
4
pivot
23 9 18
low
high
pivot
61 50
low
high
pivot
18 9 23
low
high
pivot
50 61
high
low
pivot
9 18 23
high
low
Quick Sort
on Left side
Quick Sort
on Right side
Quick Sort
on Left side
Quick Sort
on Right side
50
61
9
23
2
5
6
1
3
Final Sorted Array:
1 2 3 4 5 6
9 18 23 32 50 61

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

// to swap two numbers
void swap(int* a, int* b)
{
   int t = *a;
   *a = *b;
   *b = t;
}

int partition (int arr[], int low, int high)
{
   int pivot = arr[high];  // selecting last element as pivot
   int i = (low - 1);  // index of smaller element
   int j;

   for (j = low; j <= high- 1; j++)
   {
        // If the current element is smaller than or equal to pivot
        if (arr[j] <= pivot)
        {
           i++;    // increment index of smaller element
           swap(&arr[i], &arr[j]);
        }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}
/*
   a[] is the array, p is starting index, that is 0,
   and r is the last index of array.
*/
void quicksort(int a[], int p, int r)
{
   int q;
   if(p < r)
   {
```

```
        q = partition(a, p, r);
        quicksort(a, p, q-1);
        quicksort(a, q+1, r);
    }
}


// function to print the array
void printArray(int a[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", a[i]);
}

void main()
{
    int arr[] = {9, 7, 5, 11, 12, 2, 14, 3, 10, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    clrscr();
    // call quickSort function
    quicksort(arr, 0, n-1);

    printf("Sorted array: \n");
    printArray(arr, n);
}
```

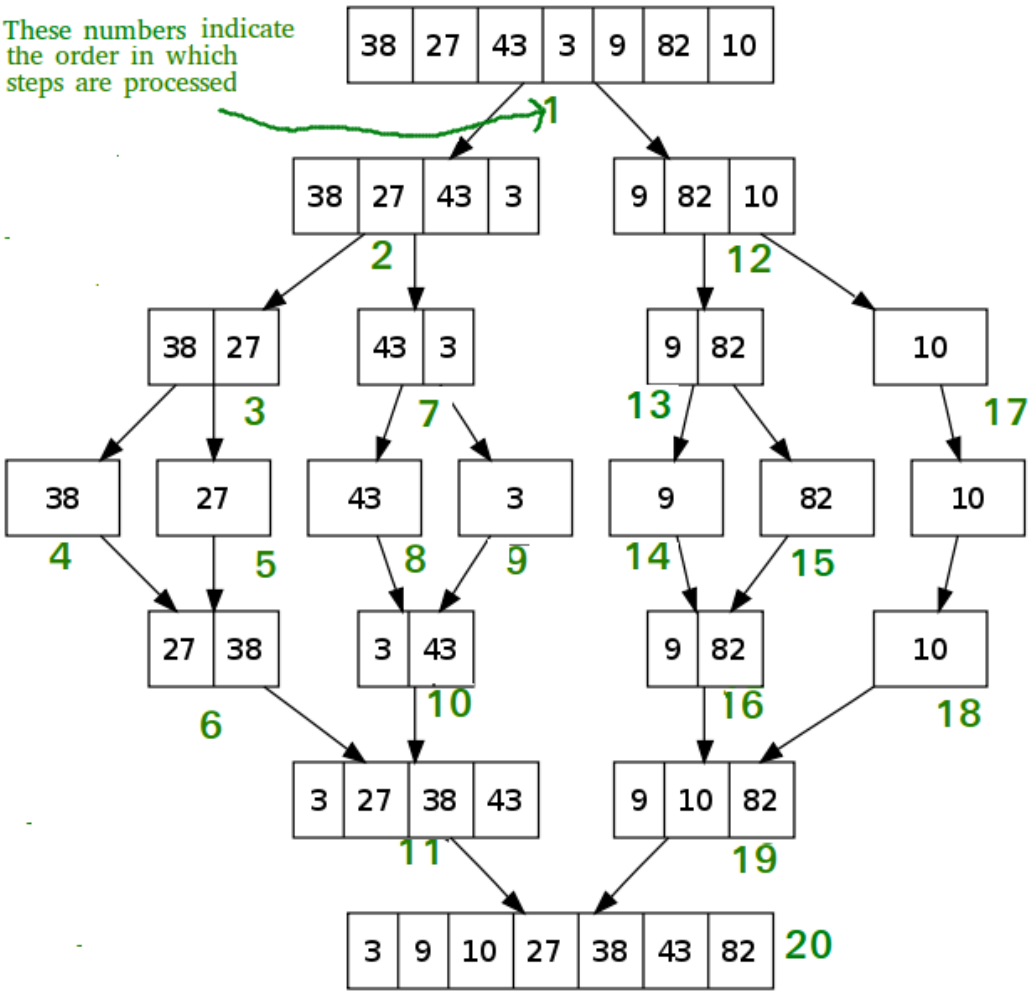
Merge Sort :

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See the following C implementation for details.

Algorithm for merge sort

MergeSort(arr[], l, r)
If r > l

1. Find the middle point to divide the array into two halves:
   middle m = l+ (r-l)/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)



```
/* C program for Merge Sort */
#include <stdio.h>
#include <stdlib.h>

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
	int i, j, k;
	int n1 = m - l + 1;
	int n2 = r - m;

	/* create temp arrays */
	int L[30], R[30];

	/* Copy data to temp arrays L[] and R[] */
	for (i = 0; i < n1; i++)
		L[i] = arr[l + i];
	for (j = 0; j < n2; j++)
		R[j] = arr[m + 1 + j];

	/* Merge the temp arrays back into arr[l..r]*/
	i = 0; // Initial index of first subarray
	j = 0; // Initial index of second subarray
	k = l; // Initial index of merged subarray
	while (i < n1 && j < n2) {
		if (L[i] <= R[j]) {
			arr[k] = L[i];
			i++;
		}
		else {
			arr[k] = R[j];
			j++;
		}
		k++;
	}

	/* Copy the remaining elements of L[], if there
	are any */
```

```
        while (i < n1) {
                arr[k] = L[i];
                i++;
                k++;
        }

        /* Copy the remaining elements of R[], if there
        are any */
        while (j < n2) {
                arr[k] = R[j];
                j++;
                k++;
        }
}


/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
        if (l < r) {
                // Same as (l+r)/2, but avoids overflow for
                // large l and h
                int m = l + (r - l) / 2;

                // Sort first and second halves
                mergeSort(arr, l, m);
                mergeSort(arr, m + 1, r);

                merge(arr, l, m, r);
        }
}

/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
        int i;
        for (i = 0; i < size; i++)
                printf("%d ", A[i]);
```

```
        printf("\n");
}

/* Driver code */
int main()
{
        int arr[] = { 12, 11, 13, 5, 6, 7 };
        int arr_size = sizeof(arr) / sizeof(arr[0]);
        clrscr();
        printf("Given array is \n");
        printArray(arr, arr_size);

        mergeSort(arr, 0, arr_size - 1);

        printf("\nSorted array is \n");
        printArray(arr, arr_size);
        return 0;
}
```

Searching

An Introduction

Searching is the process of finding specific element in given list of values. The searching is said to be either successful of unsuccessful based on whether the searched item is found or not.

There are 2 ways of searching in given list viz. linear search and binary search.

Linear or Sequential Search:

This is the simplest method of searching required data in given list. In this method, each element in the list is traversed and checked whether it is the required element or not. The search starts with the first element and goes on sequentially with next elements till the required element found.

```
procedure LINEAR_SEARCH (array, key)

   for each item in the array
      if match element == key
         return element's index
      end if
   end for

end procedure
```

```
#include <stdio.h>

int LINEAR_SEARCH(int inp_arr[], int size, int val)
{
    int i;
    for (i = 0; i < size; i++)
        if (inp_arr[i] == val)

            return i;
    return -1;
}


int main(void)
{
    int arr[] = { 10, 20, 30, 40, 50, 100, 70,90,80 };
    int key = 100;
    int size = sizeof(arr)/sizeof(arr[0]);
    int res = LINEAR_SEARCH(arr, size, key);
    if (res == -1)
    printf("ELEMENT NOT FOUND!!");
    else
    printf("Item is present at index %d", res);

    return 0;
}
```

Binary Search

Binary search is the quickest and efficient algorithm for searching in a list. But the binary search requires the all the elements sorted in the list. In this searching, the key to be searched is identified and checked whether it falls in left sub list or right sub list of the middle of list, and further divides that sub list into further sub lists and search for the key.

Binary Search is a search algorithm that is used to find the position of an element (target value) in a sorted array. The array should be sorted prior to applying a binary search. Binary search is also known by these names, logarithmic search, binary chop, half interval search.

Working

The binary search algorithm works by comparing the element to be searched by the middle element of the array and based on this comparison follows the required procedure.

Case 1 − element = middle, the element is found return the index.

Case 2 − element > middle, search for the element in the sub-array starting from middle+1 index to n.

Case 3 − element < middle, search for element in the sub-array starting from 0 index to middle -1.

ALGORITHM

Parameters inital_value , end_value

Step 1 : Find the middle element of array. using ,

middle = initial_value + end_value / 2 ;

Step 2 : If middle = element, return ‘element found’ and index.

Step 3 : if middle > element, call the function with end_value = middle - 1 .

Step 4 : if middle < element, call the function with start_value = middle + 1 .

Step 5 : exit.

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

int iterativeBinarySearch(int array[], int start_index, int end_index, int element)
{
  int middle;
```

```
  while (start_index <= end_index)
  {
    // middle = start_index + (end_index - start_index )/2;
    middle=(start_index+end_index) /2;

    if (array[middle] == element)
        return middle;

    if (array[middle] < element)
        start_index = middle + 1;
    else
        end_index = middle - 1;
  }
  return -1;
}
void main(){
  int arr[] = {1, 4, 7, 9, 16, 56, 70};
  int n = sizeof(arr)/sizeof(arr[0]);
  int element = 16;
  int found_index;
  clrscr();

   found_index = iterativeBinarySearch(arr, 0, n-1, element);

  if(found_index == -1 )
    printf("Element not found in the array ");

  else
    printf("Element %d found at index : %d",element, found_index);


}
```

Binary search using recursion

```
#include <stdio.h>

// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
```

```
// otherwise -1

int binarySearch(int arr[], int l, int r, int x)
{
   if (r >= l) {
      int mid = l + (r - l) / 2;

      // If the element is present at the middle
      // itself
      if (arr[mid] == x)
         return mid;

      // If element is smaller than mid, then
      // it can only be present in left subarray
      if (arr[mid] > x)
         return binarySearch(arr, l, mid - 1, x);

      // Else the element can only be present
      // in right subarray
      return binarySearch(arr, mid + 1, r, x);
   }

   // We reach here when element is not
   // present in array
   return -1;
}

int main(void)
{
   int arr[] = { 2, 3, 4, 10, 40 };
//if array is not sorted, then sort it first.....
   int n = sizeof(arr) / sizeof(arr[0]);
   int x = 10;
   int result = binarySearch(arr, 0, n - 1, x);
   (result == -1) ? printf("Element is not present in array")
                  : printf("Element is present at index %d",
                           result);
   return 0;
}
```

Indexed Sequential Search:

In this searching method, first of all, an index file is created, that contains some specific group or division of required record when the index is obtained, then the partial indexing takes less time cause it is located in a specified group. When the user makes a request for specific records it will find that index group first where that specific record is recorded.
Characteristics of Indexed Sequential Search:

- In Indexed Sequential Search a sorted index is set aside in addition to the array.
- Each element in the index points to a block of elements in the array or another expanded index.
-
- The index is searched 1st then the array and guides the search in the array.

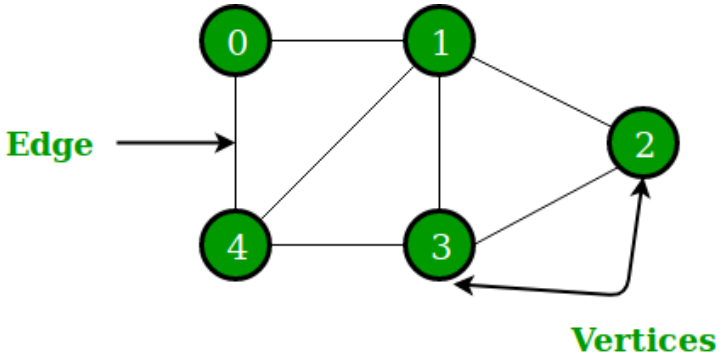
Note: Indexed Sequential Search actually does the indexing multiple times, like creating the index of an index.

Though it is faster for searching, but it takes more memory space for maintaining the index to index. It also can be used only on sorted data.

Graphs

Introduction to Graphs

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.



In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {0-1, 1-2, 2-3, 3-4, 0-4, 1-4, 1-3}, and the graph is represented as G{V,E}, where the V are vertices and the E are edges.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook.

For example, in Face book, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

Terms Associated with Graphs

Path

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.

Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

Loop

An edge that is pointing to the same vertex from which it is started.

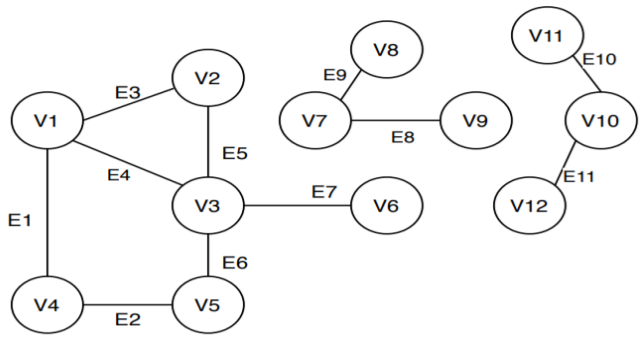
Adjacent Nodes

If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbors or adjacent nodes.

Degree of the Node

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

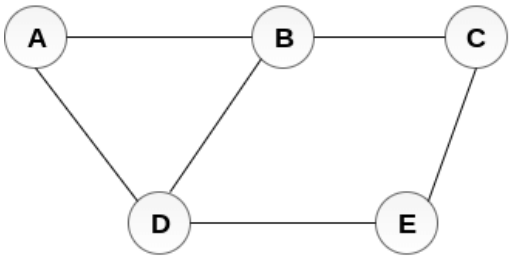
Connected components



Sequential Representation of Graphs

There are two ways to store Graph into the computer's memory. The sequential representation and linked representation.

In sequential representation, we use adjacency matrix to store the mapping represented by vertices and edges. In adjacency matrix, the rows and columns are represented by the graph vertices. A graph having n vertices will have a dimension n x n.



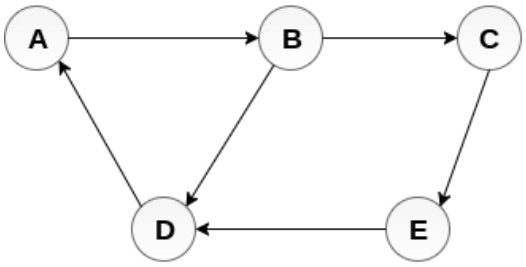
Undirected Graph

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

In the above figure, we can see the mapping among the vertices (A, B, C, D, E) is represented by using the adjacency matrix and it is representation of undirected graph.

But a directed graph and its adjacency matrix representation is shown in the following figure.



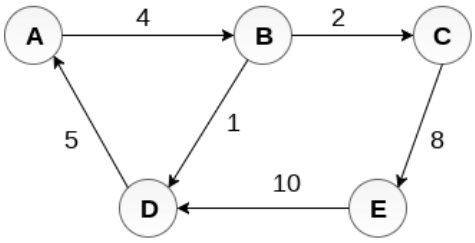
Directed Graph

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

Representation of weighted directed graph is different. Instead of filling the entry by 1, the Non- zero entries of the adjacency matrix are represented by the weight of respective edges.

The weighted directed graph along with the adjacency matrix representation is shown in the following figure



Weighted Directed Graph

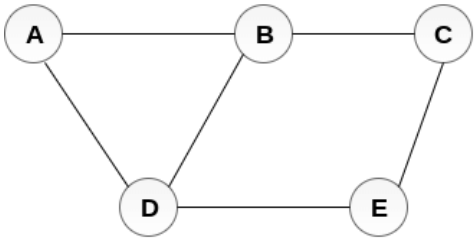
	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

Adjacency Matrix

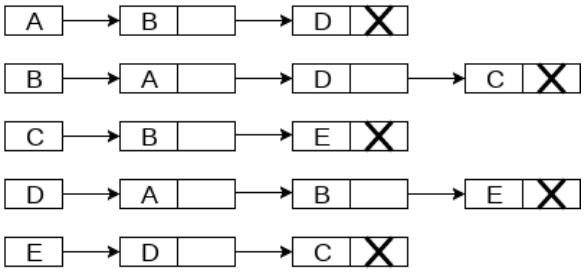
** the above adjacency matrices are represented as SPARSE MATRIX.

Linked Representation of Graphs

In the linked representation, an adjacency list is used to store the Graph into the computer's memory. Consider the undirected graph shown in the following figure and check the adjacency list representation



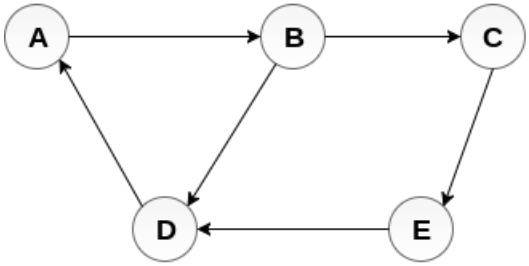
Undirected Graph



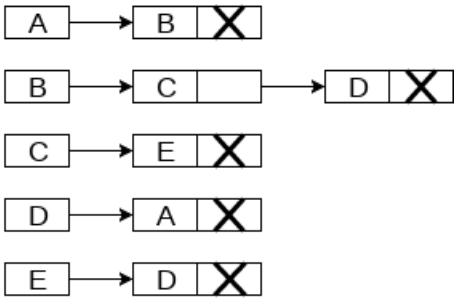
Adjacency List

An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list. The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.

Consider the directed graph shown in the following figure and check the adjacency list representation of the graph.



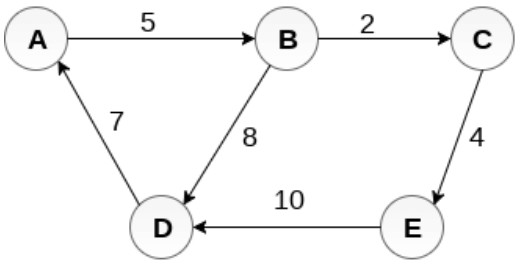
Directed Graph



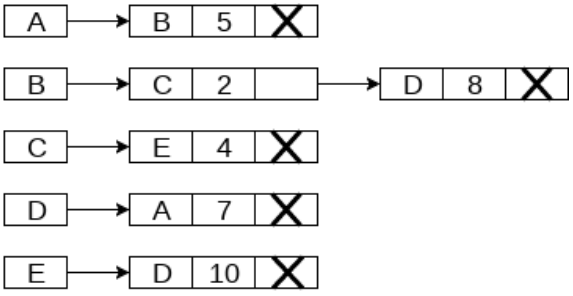
Adjacency List

In a directed graph, the sum of lengths of all the adjacency lists is equal to the number of edges present in the graph.

In the case of weighted directed graph, each node contains an extra field that is called the weight of the node. The adjacency list representation of a directed graph is shown in the following figure.



Weighted Directed Graph



Adjacency List

Program to demonstrate adjacency list representation of graphs:

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

// A structure to represent an adjacency list node
struct AdjListNode
{
   int dest;
   struct AdjListNode* next;
};

// A structure to represent an adjacency list
struct AdjList
{
   struct AdjListNode *head;
};

// A structure to represent a graph. A graph
// is an array of adjacency lists.
// Size of array will be V (number of vertices
// in graph)
struct Graph
{
   int V;
   struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest)
{
   struct AdjListNode* newNode =
    (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
   newNode->dest = dest;
   newNode->next = NULL;
   return newNode;
}

// A utility function that creates a graph of V vertices
```

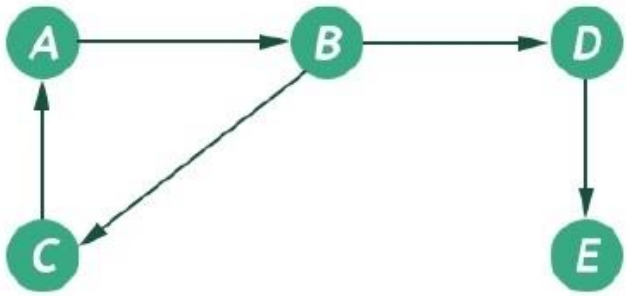
```
struct Graph* createGraph(int V)
{
int i;
    struct Graph* graph =
         (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;

    // Create an array of adjacency lists.  Size of
    // array will be V
    graph->array =
      (struct AdjList*) malloc(V * sizeof(struct AdjList));

    // Initialize each adjacency list as empty by
    // making head as NULL
    for (i = 0; i < V; ++i)
         graph->array[i].head = NULL;

    return graph;
}

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest)
{
    // Add an edge from src to dest.  A new node is
    // added to the adjacency list of src.  The node
    // is added at the beginning
    struct AdjListNode* newNode = newAdjListNode(dest);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;

    // Since graph is undirected, add an edge from
    // dest to src also
    newNode = newAdjListNode(src);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}

// A utility function to print the adjacency list
// representation of graph
void printGraph(srtruct Graph* graph)
```

```
{
    int v;
    for (v = 0; v < graph->V; ++v)
    {
        struct AdjListNode* pCrawl = graph->array[v].head;

        printf("\n Adjacency list of vertex %d\n head ", v);
        while (pCrawl)
        {
            printf("-> %d", pCrawl->dest);
            pCrawl = pCrawl->next;
        }
        printf("\n");
    }
}

// Driver program to test above functions
void  main()
{
    // create the graph given in above figure
    int V = 5;
    struct Graph* graph = createGraph(V);

    addEdge(graph, 0, 1);
    addEdge(graph, 0, 4);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);

    // print the adjacency list representation of the above graph
    printGraph(graph);

}
```

Traversal of Graphs :

The graph is one non-linear data structure. That is consists of some nodes and their connected edges. The edges may be director or undirected. This graph can be

represented as G(V, E). The following graph can be represented as G({A, B, C, D, E}, {(A, B), (B, D), (D, E), (B, C), (C, A)})



The graph has two types of common traversal algorithms. These are called the Breadth First Search and Depth First Search.

Depth First Search (DFS)

The Depth-First Search (DFS) is a graph traversal algorithm. In this algorithm, one starting vertex is given, and when an adjacent vertex is found, it moves to that adjacent vertex first and tries to traverse in the same manner. It moves through the whole depth, as much as it can go, after that it backtracks to reach previous vertices to find the new path. To implement DFS in an iterative way, we need to use the stack data structure. If we want to do it recursively, external stacks are not needed, it can be done internal stacks for the recursion calls.

Algorithm

- Step 1: SET STATUS = 1 (ready state) for each node in G
- Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)
- Step 3: Repeat Steps 4 and 5 until STACK is empty
- Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)
- Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state) [END OF LOOP]
- Step 6: EXIT

Breadth First Search (BFS)

The Breadth First Search (BFS) traversal is an algorithm, which is used to visit all of the nodes of a given graph. In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one. After completing all of the adjacent vertices, it moves further to check another vertex and checks its adjacent vertices

again. To implement this algorithm, we need to use the Queue data structure. All the adjacent vertices are added into the queue when all adjacent vertices are completed, one item is removed from the queue and start traversing through that vertex again.
Algorithm

- Step 1: SET STATUS = 1 (ready state)
  for each node in G
- Step 2: Enqueue the starting node A
  and set its STATUS = 2
  (waiting state)
- Step 3: Repeat Steps 4 and 5 until
  QUEUE is empty
- Step 4: Dequeue a node N. Process it
  and set its STATUS = 3
  (processed state).
- Step 5: Enqueue all the neighbours of
  N that are in the ready state
  (whose STATUS = 1) and set
  their STATUS = 2
  (waiting state)
  [END OF LOOP]

Step 6: EXIT

The differences between BFS and DFS are :

	BFS	DFS
1.	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
2.	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
3.	BFS can be used to find single source shortest path in an unweighted graph, because in BFS, we reach a vertex with	In DFS, we might traverse through more edges to reach a destination vertex from a

	minimum number of edges from a source vertex.	source.
3.	BFS is more suitable for searching vertices which are closer to the given source.	DFS is more suitable when there are solutions away from source.
4.	BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win situation, we stop.
5.	The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.	The Time complexity of DFS is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.

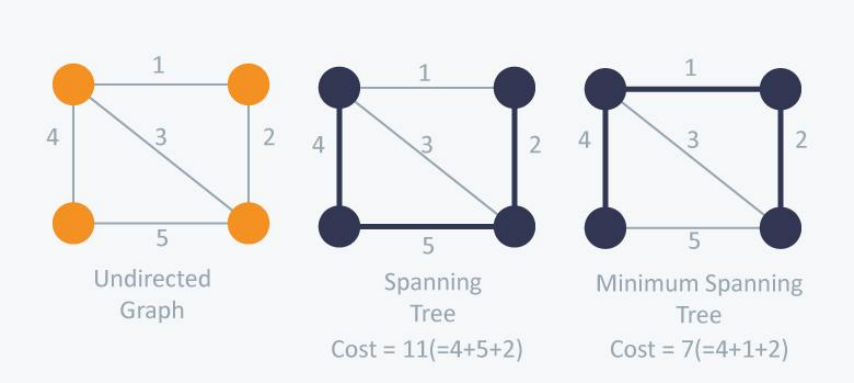
Spanning Trees :

Trees can be defined as special cases of graphs. A tree is a connected graph that has no cycles.

In case of a tree there is one root node and all child nodes are traversed from that root node. But tree representation of a graph has no special root vertex. Each vertex can be treated as root.

Given an undirected and connected graph G=(V,E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a sub graph of G (every edge in the tree belongs to G)

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/low among all the spanning trees. There also can be many minimum spanning trees.



In above example, the thick lines in figure 2 & 3 are the spanning trees. In which we can identify the cost of each span while traversing and we can choose the minimum cost span tree.

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:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation

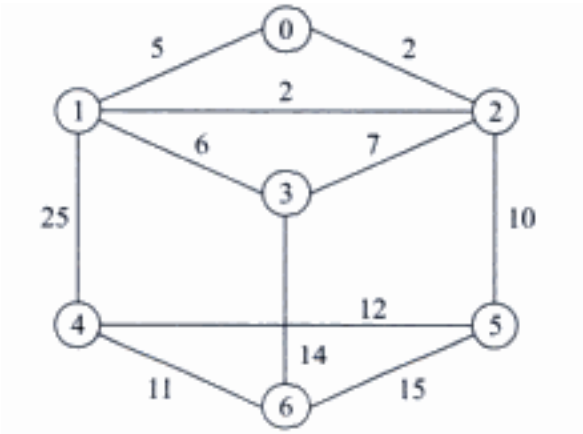
The main advantages of minimal spanning tree is to find shortest path.

Shortest Path :

As a final application of graphs, one requiring somewhat more sophisticated reasoning, we consider the following problem: We are given a directed graph **G** in which every edge has a weight attached, and our problem is to find a path from one vertex **V** to another **W** such that the sum of the weights on the path is as small as possible. We call such a path a **shortest path**.

Length of a path in a weighted graph is defined to be the sum of costs or weights of all edges in that path. In general there could be more than one path between a pair of specified vertices, say "$\mathbf{V_i}$" and "$\mathbf{V_j}$" and a path with the minimum cost or weight is called the **shortest** path from "$\mathbf{V_i}$" to "$\mathbf{V_j}$". Note that the shortest path between two vertices may not be unique.

Consider the weighted undirected graph given in Fig. 12.33.



It can be easily seen that there are more than three paths 0–1, 0–2–1, 0–2–3–1, from the vertex '**0**' to the vertex '**1**'. The path with the shortest path length is 0–2–1. The length of the shortest path is 4.

There are many different variations of the shortest path problem. They vary with respect to the specification of the start vertex (referred to as source) and end vertex (referred to as destination). Some of the commonly known variants are listed below:

- The shortest path from a specified source vertex to a specified destination vertex.
- The shortest path from one specified vertex to all other vertices. This problem is also known as single source shortest path problem.
- The shortest path between all possible source and destination vertices. This problem is also known as all pairs shortest path problem.
- The shortest path can be determined using Kruskal's on Prim's algorithm.

Application of Graphs:

Let us assume one input line containing four integers followed by any number of input lines with two integers each. The first integer on the first line, n, represents number of cities which, for simplicity, are numbered from 0 to n – 1. The second and third integers on that line are between 0 to n-1 and represent two cities. It is desired to travel from the first city to second using exactly nr roads, where nr is the fourth integer on the first input line. Each subsequent input line contains two integers representing two cities, indicating that there is a road from the first city to the second. The problem is to determine whether there is a path of required length by which one can travel from the first of the given cities to the second.

Following is the plan for solution:

Create a graph with the cities as nodes and the roads as arcs. To find the path of length nr from node A to node B, look for a node C such that an arc exists from A to C and a path of length nr – 1 exists from C to B. If these conditions are satisfied for some node C, the desired path exists. If the conditions are not satisfied for any node C, the desired path does not exist.

The traversal of a graph like Depth first traversal has many important applications such as finding the components of a graph, detecting cycles in an undirected graph, determining whether the graph is bi-connected, etc.