SKDAV GOVERNMENT COLLEGE,ROURKELA

DEPARTMENT OF INFORMATION TECHNOLOGY

DATA STRUCTURE

BRANCH-IT

NAME OF THE FACULTY-DIPTI REKHA MISHRA

SEMESTER-3rd

Th-2 DATA STRUCTURE

Common to (CSE/IT)

Theory	**4 Periods per week**	**Internal Assessment**	**20 Marks**
Total Periods	60	**End Sem Exam**	**80 Marks**
Examination	3hours	**Total Marks**	100Marks

A. Topic wise distribution of periods

Sl. No.	Topics	Periods
1	INTRODUCTION	04
2	STRING PROCESSING	03
3	ARRAYS	07
4	STACKS & QUEUES	08
5	LINKED LIST	08
6	TREE	08
7	GRAPHS	06
8	SORTING SEARCHING & MERGING	08
9	FILE ORGANIZATION	08
	TOTAL	**60**

B. RATIONAL: The study of **Data structure** is an essential part of computer science. Data structure is a logical & mathematical model of storing & organizing data in a particular way in a computer. In system programming application programming the methods & techniques of data structures are widely used. The study of data structure helps the students in developing logic & structured programs.

C. OBJECTIVE: After completion of this **course** the student will be able to:

- Understand the concepts of linear data structures, their operations and applications
- Understand the operation in abstract data type like Stack and Queue.
- Understand the concept of pointers and their operations in linked list.
- Know the concepts of non-linear data structures, their operations and applications intree and graph.
- Understand the various sorting and searching techniques.
- Understand file storage and access techniques.

D. DETAIL CONTENT:

INTRODUCTION:

Explain Data, Information, data types

Define data structure & Explain different operations

Explain Abstract data types

Discuss Algorithm & its complexity

Explain Time, space tradeoff

STRING PROCESSING

Explain Basic Terminology, Storing Strings

State Character Data Type,

Discuss String Operations

ARRAYS

Give Introduction about array,

Discuss Linear arrays, representation of linear array In memory

Explain traversing linear arrays, inserting & deleting elements

Discuss multidimensional arrays, representation of two dimensional arrays in memory (row
major order & column major order), and pointers

Explain sparse matrices.

STACKS & QUEUES

Give fundamental idea about Stacks and queues

Explain array representation of Stack

Explain arithmetic expression ,polish notation & Conversion

Discuss application of stack, recursion

Discuss queues, circular queue, priority queues.

LINKED LIST

Give Introduction about linked list

Explain representation of linked list in memory

Discuss traversing a linked list, searching,

Discuss garbage collection.

Explain Insertion into a linked list, Deletion from a linked list, header linked list

TREE

Explain Basic terminology of Tree

Discuss Binary tree, its representation and traversal, binary search tree, searching,

Explain insertion & deletion in a binary search trees

7.0 GRAPHS

7.1 Explain graph terminology & its representation,

7.2 Explain Adjacency Matrix, Path Matrix

8.0 SORTING SEARCHING & MERGING

Discuss Algorithms for Bubble sort, Quick sort,

Merging

Linear searching, Binary searching.

FILE ORGANIZATION

Discuss Different types of files organization and their access method,

Introduction to Hashing, Hash function, collision resolution, open addressing.

Coverage of Syllabus upto Internal Exams (I.A.)
Chapter 1,2,3,4

Book Recommended:-

Sl No.	Name of Authors	Title of Book	Name of Publisher:
1	S. Lipschutz	Data Structure	Schaum Series
2	A.N.Kamthane	Introduction to Data Structure in C	Pearson Education
3	Reema Thereja	Data Structure using C	Oxford University Press

UNIT-1

INTRODUCTION

DATA

Data is a set of values of qualitative or quantitative variables. Data in computing (or data processing) is represented in a structure that is often tabular (represented by rows and columns), a tree (a set of nodes with parent-children relationship), or a graph (a set of connected nodes). Data is typically the result of measurements and can be visualized using graphs or images.

Data as an abstract concept can be viewed as the lowest level of abstraction, from which information and then knowledge are derived.

INFORMATION

Information is that which informs us with some valid meaning, i.e. that from which data can be derived. Information is conveyed either as the content of a message or through direct or indirect observation of some thing. Information can be encoded into various forms for transmission and interpretation. For example, information may be encoded into signs, and transmitted via signals.

DATA TYPE

Data types are used within type systems, which offer various ways of defining, implementing and using the data. Different type systems ensure varying degrees of type safety.

Almost all programming languages explicitly include the notion of data type. Though different languages may use different terminology. Common data types may include:

- Integers,
- Booleans,
- Characters,
- Floating-point numbers,
- Alphanumeric strings.

For example, in the Java programming language, the "int" type represents the set of 32-bit integers ranging in value from -2,147,483,648 to 2,147,483,647, as well as the operations that can be performed on integers, such as addition, subtraction, and multiplication. Colors, on the other hand, are represented by three bytes denoting the amounts each of red, green, and blue, and one string representing that color's name; allowable operations include addition and subtraction, but not multiplication.

Most programming languages also allow the programmer to define additional data types, usually by combining multiple elements of other types and defining the valid operations of the new data type. For example, a programmer might create a new data type named "complex number" that would include real and imaginary parts. A data type also represents a constraint placed upon the interpretation of data in a type system, describing representation, interpretation and structure of values or objects stored in computer memory. The type system uses data type information to check correctness of computer programs that access or manipulate the data.

CLASSES OF DATA TYPES

There are different classes of data types as given below.

- Primitive data type
- Composite data type
- En-numerated data type
- Abstract data type
- Utility data type
- Other data type

PRIMITIVE DATA TYPES

All data in computers based on digital electronics is represented as bits (alternatives 0 and 1) on the lowest level. The smallest addressable unit of datais usually a group of bits called a byte (usually an octet, which is 8 bits). The unit processed by machine code instructions is called a word (as of 2011, typically 32 or 64 bits). Most instructions interpret the word as a binary number, such that a 32-bit word can represent unsigned integer

values from 0 to $2^{32} - 1$or signed integer values from $-2^{31}$to $2^{31} - 1$. Because of two's complement, the machine language and machine doesn't need to distinguish between these unsigned and signed data types for the most part.

There is a specific set of arithmetic instructions that use a different interpretation of the bits in word as a floating-point number. Machine data types need to be *exposed* or made available in systems or low-level programming languages, allowing fine-grained control over hardware. The C programming language, for instance, supplies integer types of various widths, such as short and long. If a corresponding native type does not exist on the target platform, the compiler will break them down into code using types that do exist. For instance, if a 32-bit integer is requested on a 16 bit platform, the compiler will tacitly treat it as an array of two 16 bit integers. Several languages allow binary and hexadecimal literals, for convenient manipulation of machine data.

In higher level programming, machine data types are often hidden or abstracted as an implementation detail that would render code less portable if exposed. For instance, a generic numeric type might be supplied instead of integers of some specific bit-width.

BOOLEAN TYPE

The Boolean type represents the values true and false. Although only two values are possible, they are rarely implemented as a single binary digit for efficiency reasons. Many programming languages do not have an explicit boolean type, instead interpreting (for instance) 0 as false and other values as true.

- The integer data types, or "whole numbers". May be subtyped according to their ability to contain negative values (e.g. unsigned in C and C++). May also have a small number of predefined subtypes (such as short and long in C/C++); or allow users to freely define subranges such as 1..12 (e.g. Pascal/Ada).
- Floating point data types, sometimes misleadingly called reals, contain fractional values. They usually have predefined limits on both their maximum values and their precision. These are often represented as decimal numbers.
- Fixed point data types are convenient for representing monetary values. They are often implemented internally as integers, leading to predefined limits.
- Bignum or arbitrary precision numeric types lack predefined

limits. They are not primitive types, and are used sparingly for efficiency reasons.

COMPOSITE / DERIVED DATA TYPES

Composite types are derived from more than one primitive type. This can bedone in a number of ways. The ways they are combined are called data

structures. Composing a primitive type into a compound type generally results ina new type, e.g. *array-of-integer* is a different type to *integer*.

- An array stores a number of elements of the same type in a specific order. They are accessed using an integer to specify which element is required (although the elements may be of almost any type). Arrays may be fixed- length or expandable.
- Record (also called tuple or struct) Records are among the simplest data structures. A record is a value that contains other values, typically in fixed number and sequence and typically indexed by names. The elements of records are usually called *fields* or *members*.
- Union. A union type definition will specify which of a number of permitted primitive types may be stored in its instances, e.g. "float or long integer". Contrast with a record, which could be defined to contain a float *and* an integer; whereas, in a union, there is only one value at a time.
- A tagged union (also called a variant, variant record, discriminated union, or disjoint union) contains an additional field indicating its current type, for enhanced type safety.
- A set is an abstract data structure that can store certain values, withoutany particular order, and no repeated values. Values themselves are not retrieved from sets, rather one tests a value for membership to obtain a boolean "in" or "not in".
- An object contains a number of data fields, like a record, and also a number of program code fragments for accessing or modifying them. Data structures not containing code, like those above, are called plain old data structure.

Many others are possible, but they tend to be further variations and compoundsof the above.

ENUMERATED TYPE

This has values which are different from each other, and which can be compared and assigned, but which do not necessarily have any particular concrete representation in the computer's memory; compilers and interpreters can represent them arbitrarily. For example, the four suits in a deck of playing cards may be four enumerators named *CLUB*, *DIAMOND*, *HEART*, *SPADE*, belonging to an enumerated type named *suit*. If a variable *V* is declared having *suit* as its data type, one can assign any of those four values to it. Some implementations allow programmers to assign integer values to the enumeration values, or even treat them as type-equivalent to integers.

String and text typesSuch as:

- Alphanumeric character. A letter of the alphabet, digit, blank space, punctuation mark, etc.
- Alphanumeric strings, a sequence of characters. They are typically used to represent words and text.

Character and string types can store sequences of characters from a character set such as ASCII. Since most character sets include the digits, it is possible to have a numeric string, such as "1234". However, many languages would still treat these as belonging to a different type to the numeric value 1234.

Character and string types can have different subtypes according to the required character "width". The original 7-bit wide ASCII was found to be limited and superseded by 8 and 16-bit sets.

ABSTRACT DATA TYPES

Any type that does not specify an implementation is an abstract data type. For instance, a stack (which is an abstract type) can be implemented as an array (a contiguous block of memory containing multiple values), or as a linked list (a set

of non-contiguous memory blocks linked by pointers).

Abstract types can be handled by code that does not know or "care" what underlying types are contained in them. Arrays and records can also contain underlying types, but are

considered concrete because they specify how their contents or elements are laid out in memory.

Examples include:

- A queue is a first-in first-out list. Variations are Deque and Priority queue.
- A set can store certain values, without any particular order, and with norepeated values.
- A stack is a last-in, first out.
- A tree is a hierarchical structure.
- A graph.
- A hash or dictionary or map or Map/Associative array/Dictionary is a more flexible variation on a record, in which name-value pairs can be added anddeleted freely.
- A smart pointer is the abstract counterpart to a pointer. Both are kinds of reference

UTILITY DATA TYPES

For convenience, high-level languages may supply ready-made "real world" data types, for instance *times*, *dates* and *monetary values* and *memory*, even where the language allows them to be built from primitive types.

OTHER DATA TYPES

Types can be based on, or derived from, the basic types explained above. In some languages, such as C, functions have a type derived from the type of their return value. The main non-composite, derived type is the pointer, a data type whose value refers directly to (or "points to") another value stored elsewhere in the computer memory using its address. It is a primitive kind of reference. (In everyday terms, a page number in a book could be considered a piece of datathat refers to another one). Pointers are often stored in a format similar to an integer; however, attempting to dereference or "look up" a pointer whose value was never a valid memory address would cause a program to crash. To ameliorate this potential problem, pointers are considered a separate type to the type of data they point to, even if the underlying representation is the same.

DATA STRUCTURE

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organising data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artifical intelligence, Graphics and many more.

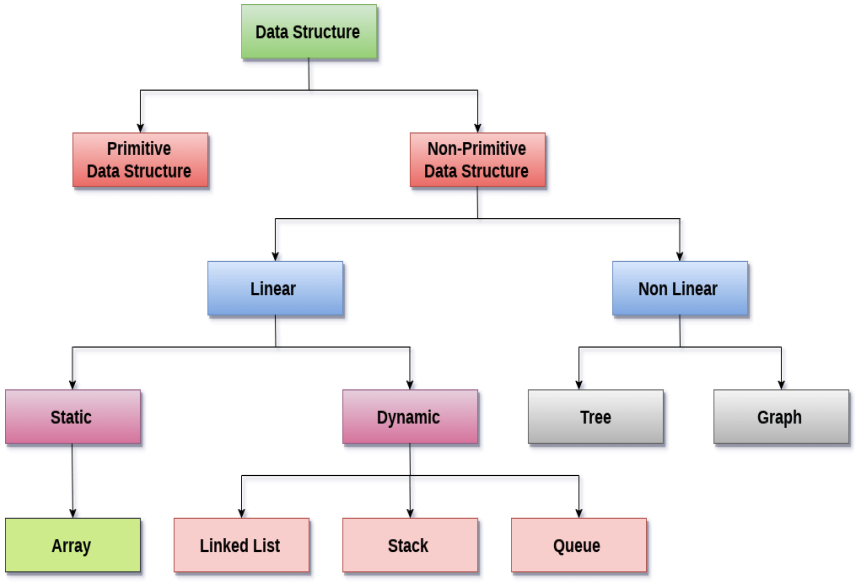
ADVANTAGES OF DATA STRUCTURES

Efficiency: Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a perticular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

Reusability: Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Abstraction: Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

Data Structure Classification



PRIMITIVE DATA STRUCTURE

- Primitive data structures are basic structures and are directly operated upon by machine instructions.
- Primitive data structures have different representations on different computers.
- Integers, floats, character and pointers are examples of primitive data structures.
- These data types are available in most programming languages as built in type.
  - Integer: It is a data type which allows all values without fraction part. We can use it for whole numbers.
  - Float: It is a data type which use for storing fractional numbers.
  - Character: It is a data type which is used for character values.

NON PRIMITIVE DATA TYPE

- These are more sophisticated data structures.
- These are derived from primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.
- Examples of Non-primitive data type are Array, List, and File etc.
- A Non-primitive data type is further divided into Linear and Non-Linear data structure

LINEAR DATA STRUCTURES: A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

Arrays: An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are:

age[0], age[1], age[2], age[3],..... age[98], age[99].

Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

Stack: Stack is a linear list in which insertion and deletions are allowed only at one end, called top.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

The basic concept can be illustrated by thinking of your data set as a stack of plates or books where you can only take the top item off the stack in order to remove things from it.

This structure is used all throughout programming.

The basic implementation of a stack is also called a —Last In First Out‖ structure; however there are different variations of stack implementations.

There are basically three operations that can be performed on stacks. They are:

- inserting (—pushing‖) an item into a stack
- deleting (—popping‖) an item from the stack
- displaying the contents of the top item of the stack (—peeking)

Queue: Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.

It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

NON LINEAR DATA STRUCTURES:

This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non Linear Data Structures are given below:

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the herierchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node. Trees can be classfied into many categories which will be discussed later in this tutorial.

Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree can not have the one.

A graph data structureconsists of a finte (and possibly mutable) set of ordered pairs, called edges or arcs, of certain entities called nodes or vertices. As in mathematics, an edge (*x*,*y*) is said to point or go from *x* to *y*. The nodes may be part of the graph structure, or may be external entities represented by integer indices or references. A graph data structure may also associate to each edge some edge value, such as a symbolic label or a numeric attribute.

OPERATIONS ON DATA STRUCTURE

1) **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will devide that sum by the number of subjects i.e. 6, in order to find the average.

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:**The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then **underflow** occurs.

4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

5) **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

ABSTRACT DATA TYPE

In computer science, an abstract data type (ADT) is a mathematical model for a certain class of data structures that have similar behavior; or for certain data types of one or more programming languages that have similar semantics. An abstract data type is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects (and possibly cost) of those operations.

For example, an abstract stack could be defined by three operations:

1. push, that inserts some data item onto the structure,
2. pop, that extracts an item from it (with the constraint that each pop always returns the most recently pushed item that has not been popped yet), and
3. peek, that allows data on top of the structure to be examined without removal.

When analyzing the efficiency of algorithms that use stacks, one may also specify that all operations take the same time no matter how many items have been pushed into the stack, and that the stack uses a constant amount of storagefor each element.

Abstract data types are purely theoretical entities, used (among other things) to simplify the description of abstract algorithms, to classify and evaluate data structures, and to formally describe the type systems of programming languages. However, an ADT may be implemented by specific data types or data structures, in many ways and in many programming languages; or described in a formal specification language. ADTs are often implemented as modules: the module's interface declares procedures that correspond to the ADT operations, sometimes with comments that describe the constraints. This information hiding strategy allows the implementation of the module to be changed without disturbing the client programs.

ALGORITHM

In mathematics and computer science, an algorithm is a step-by-step procedure for calculations. Algorithms are used for calculation, data processing, and automated reasoning.An algorithm is an effective method expressed as a finite list of well-

defined instructions for calculating a function. Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, proceeds through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input.

COMPLEXITY OF ALGORITHM AND TIME, SPACE TRADEOFF

In computer science, the **algorithms** are evaluated by the determination of the amount of resources (such as time and storage) necessary to execute them. Most algorithms are designed to work with inputs of arbitrary length. Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (spacecomplexity).

Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms.

In theoretical analysis of algorithms it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. Big O notation, Big-omega notation and Big-theta notation are used to this end. For instance, binary search is said to run in a number of steps proportionalto the logarithm of the length of the list being searched, or in O(log(n)), colloquially "in logarithmic time". Usually asymptotic estimates are used because different implementations of the same algorithm may differ in efficiency. However the efficiencies of any two "reasonable" implementations of a given algorithm are related by a constant multiplicative factor called a *hidden constant*.

Exact (not asymptotic) measures of efficiency can sometimes be computed but they usually require certain assumptions concerning the particular implementation of the algorithm, called model of computation. A model of computation may be defined in terms of an abstract computer, e.g., Turing machine, and/or by postulating that certain operations are executed in unit time. For example, if the sorted list to which we apply binary search has *n* elements, and we can guarantee that each lookup of an element in the list can be done in

unit time, then at most $\log_2 n + 1$ time units are needed to return an answer.

BEST, WORST AND AVERAGE CASE COMPLEXITY

The best, worst and average case complexity refer to three different ways of measuring the time complexity (or any other complexity measure) of different inputs of the same size. Since some inputs of size *n* may be faster to solve than others, we define the following complexities:

- Best-case complexity: This is the complexity of solving the problem for the best input of size *n*.
- Worst-case complexity: This is the complexity of solving the problem for the worst input of size *n*.
- Average-case complexity: This is the complexity of solving the problem on an average. This complexity is only defined with respect to a probability distribution over the inputs. For instance, if all inputs of the same size are assumed to be equally likely to appear, the average case complexity can be defined with respect to the uniform distribution over all inputs of size *n*.

TIME COMPLEXITY

In computer science, the **time complexity** of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the string representing the input. The time complexity of an algorithm is commonly expressed using **big O** notation, which excludes coefficients and lower order terms. When expressed this way, the time complexity is said to be described *asymptotically*, i.e., as the input size goes to infinity.

For example, if the time required by an algorithm on all inputs of size *n* is at most $5n^3 + 3n$, the asymptotictime complexity is $O(n^3)$.

Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform. Thus the amount of time taken and the number of elementary operations performed by the algorithm differ by at most a constant factor.

Since an algorithm's performance time may vary with different inputs of the same size, one

commonly uses the worst-case time complexity of an algorithm, denoted as ***T(n)***, which is defined as the maximum amount of time taken on any input of size *n*. Time complexities are classified by the nature of the function *T*(*n*).For instance, an algorithm with *T*(*n*) = O(*n*) is called a linear time algorithm, and an algorithm with *T*(*n*) = O(2^n) is said to be an exponential time algorithm.

SPACE COMPLEXITY

The way in which the amount of storage space required by an algorithm varies with the size of the problem it is solving. Space complexity is normally expressed as an order of magnitude, e.g. O(N^2) means that if the size of the problem (N) doubles then four times as much working storage will be needed.

ASYMPTOTIC NOTATIONS

The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:

- Big oh Notation (O)
- Omega Notation (Ω)
- Theta Notation (θ)

Big oh Notation (O) It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the longest amount of time, algorithm takes to complete their operation.

Omega Notation (Ω) It is the formal way to represent the lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the best case time complexity.

Theta Notation (?) It is the formal way to express both the upper bound and lower bound of an algorithm running time.

UNIT-II

STRINGPROCESSING

A finite sequence _S‘ of zero or more characters is called a String. The string withzero character is called the empty string or null string.

- The number of characters in a string is called its length.
- Specific string will be denoted by in closing their character in singlequotation mark.

For e.g. “SKDAV”
‘123’

Let $S1$; $S2$ be the string consist of the character of $S1$ followed by the characters of $S2$ is called the concatenation of $S1$ & $S2$. It will be denoted by $S1$, $S2$.

For e.g., $S1$ = ‘XY1‘

$S2$ = ‘ PQR‘

$S1$ || $S2$ = XY1 PQR

$S1$ = _XY1‘, $S2$ = (space), $S3$ = _PQR‘

$S1$ || $S2$ || $S3$ = XY1 (space) PQR

The length of $S1$ || $S2$ || $S3$ is equal to the sum of length string $S1$ & $S2$ & $S3$. A string Y is called a substring of a string _S‘ & if their exits string _S‘ & if their exits string X & Z. Such that S=X || Y || Z. If X is an empty string, then y is called &
initial substring of _S‘ & Z is an empty string then _Y‘ is called a terminal substringof _S‘.

CHARACTER DATA TYPE:-

- The character data type is of two data type. (1) Constant (2) Variable

Constant String:

-> The constant string is fixed & is written in either _ ‘ single quote & — ‖ doublequotation.

Ex:- _SONA‘

—Sonaǁ

Variable String:

String variable falls into 3 categories.

- Static
- Semi-Static
- Dynamic

Static character variable:

Whose variable is defined before the program can be executed & cannot change

throughout the program.

Semi-static variable:

Whose length variable may as long as the length does not exist, a maximum value. A maximum value determine by the program before the program is executed.

Dynamic variable:

A variable whose length can change during the execution of the program.

STRING OPERATION:

There are four different operations.

1. Sub string
2. Indexing
3. Concatenation
4. Length

Sub string:-

Group of conjunctive elements in a string (such as wards, purchases or sentences) called substring.

Accessing substring of a given string required 3 pieces of information.

1. The name of the string or the string itself.
2. The position of the first character of the substring in the given string.
3. The length of the substring of the last character of the substring.

We calledthis operation SUBSTRING.

SUBSTRING (String, initial, length)

To denote the substring of string _S‘ beginning in the position _K‘ having a length _L‘.

SUBSTRING (S, K, L) **T** **K L**

For e.g.; SUBSTRING (_TO BE OR NOT TO BE‘, 4, 7)

SUBSTRING=BE OR N

SUBSTRING (THE END, 4, 4)SUBSTRING=END.

INDEXING:-

Indexing also called pattern matching which refers to finding the position where a string pattern _P‘. First appears in a given string text _T‘, we called this operation index and write as INDEX (text, pattern)

If the pattern _P‘ does not appear in text _T‘ then index is assign the value 0; the argument & text and pattern can either string constant or string variable.

For e.g.; T contains the text.

_HIS FATHER IS THE PROFESSOR‘Then INDEX (T, _THE‘)

7

INDEX (T, _THEN‘)

0

INDEX (THE‘)

10

Concatenation:-

Let $S1$ & $S2$ in be the string then concatenation of $S1$ & $S2$ is denoted by $S1$ $S2$, $S1$ || $S2$, each the string consist of the character of $S1$ followed by the characters of $S2$.

Ex:- $S1$ = ‗Sonalisa‘ $S2$ = ‗ ‘ $S3$ = ‗Behera‘

$S1$ || $S2$ || $S3$ = Sonalisa Behera

Length operation:-

The number of character in a string is called its length. We will write LENGTH (string).For the length of a given string LENGTH (—Computer‖). The length is 8.

Basic language LEN (STRING)

Strlen (string)

Strupper(string) String upper

Strupr(‗computer‘)

COMPUTER

String lower

Strlwr (‗COMPUTER‘)

COMPUTER

String concatenating

Strcnt String Reverse

Strrev

UNIT-III

ARRAY

Definition

- Arrays are defined as the collection of similar type of data items stored at contiguous memory locations.
- Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc.
- Array is the simplest data structure where each data element can be randomly accessed by using its index number.

For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variable for the marks in different subject.

instead of that, we can define an array which can store the marks in each subject at a the contiguous memory locations. The array marks[10] defines the marks of the student in 10 different subjects where each subject marks are located at a particular subscript in the array i.e. marks[0] denotes the marks in first subject, marks[1] denotes the marks in 2nd subject and so on

Array is a list of finite number of n homogeneous data elements i.e. the elements of same data types Such that:

- The elements of the array are referenced respectively by an index set consisting of n consecutive numbers.
- The elements of the array are stored respectively in the successive memory locations.

The number n of elements is called length or size of array. If not explicitly stated, we will assume the index set consists of integers 1, 2, 3 …n. In general the length or the number of data elements of the array can be obtained from the index set by the formula

Length= UB – LB + 1

Where UB is the largest index, called the upper bound, and LB is the smallestindex, called the lower bound. Note that length = UB when LB = 1.

The elements of an array A are denoted by subscript notation à a_1, a_2, a_3…a_nOr by the parenthesis notation -> A (1), A (2),…., A(n)

Or by the bracket notation -> A[1], A[2],….,A[n].

We will usually use the subscript notation or the bracket notation.

REPRESENTATION OF LINEAR ARRAYS IN MEMORY:

Let LA is a linear array in the memory of the computer. Recall that the memory of computer is simply a sequence of addressed locations.

LOC (LA[k]) = address of element LA[k] of the array LA.

As previously noted, the elements of LA are stored in the successive memory cells. Accordingly, the computer does not need to keep track of the address of every element of LA, but needs to keep track only of the address of the first element of LA, denoted by

Base (LA)

And called the base address of LA. Using base address the computer calculates the address of any element of LA by the following formula:

LOC (LA[k]) = Base (LA) + w (k-lower bound)

Where w is the number of words per memory cell for the array LA.

OPERATIONS ON ARRAYS

Various operations that can be performed on an array

- Traversing
- Insertion
- Deletion
- Sorting
- Searching
- Merging

TRAVERSING LINEAR ARRAY:

Let A be a collection of data elements stored in the memory of the computer. Suppose we want to print the content of each element of A or suppose we wantto count the number of elements of A, this can be accomplished by traversing A, that is, by accessing and processing each element of a exactly ones.

The following algorithm traverses a linear array LA. The simplicity of the algorithm comes from the fact that LA is a linear structure. Other linear structures, such as linked list, can also be easily traversed. On the other hand, traversal of nonlinear structures, such as trees and graph, is considerably more complicated.

Algorithm: (Traversing a Linear Array)

Here LA is a linear array with lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA.

1. [Initialize counter] Set k: =LB.
2. Repeat steps 3 and 4 while k <=UB.
3. [Visit Element] Apply PROCESS to LA [k].
4. [Increase Counter] Set
k: =k + 1.[End of step 2
loop]
5. Exit.

OR

We also state an alternative form of the algorithm which uses a repeat-for loop instead of the repeat-while loop.

Algorithm: (Traversing a Linear Array)

Here LA is a linear array with lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA.

1. Repeat for k = LB to UB:

Apply PROCESS to LA [k]. [End of loop]

2. Exit.

Caution: The operation PROCESS in the traversal algorithm may use certain variables which must be initialized before PROCESS is applied to any of the elements in the array. Accordingly, the algorithm may need to be proceeded by such an initialization step.

INSERTION AND DELETION IN LINEAR ARRAY:

Let A be a collection of data elements in the memory of the computer. —Inserting‖ refers to the operation of adding another element to the collection A, and —deleting‖ refers to the operation of removing one element from A.

Inserting an element at the end of the linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element. On the other hand, suppose we need to insert an element in the middle of the array. Then, on the average, half of the elements must be moved downward to new location to accommodate the new elements and keep the order of the other elements.

Similarly, deleting an element at the end of the array presents no difficulties, but deleting the element somewhere in the middle of the array requires that each subsequent element be moved one location upward in order to fill up the array.

The following algorithm inserts a data element ITEM in to the Kth position in the linear array LA with N elements.

Algorithm for Insertion: (Inserting into Linear Array)INSERT (LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that K<=N. The algorithm inserts an element ITEM into the Kth position in LA.

1. [Initialize counter] Set J: = N.

2. Repeat Steps 3 and 4 while j >= k;
3. [Move jth element downward.] Set LA [J + 1]: =LA [J].
4. [Decrease counter]
Set J: = J-1[End of step
2 loop]
5. [Insert element] Set LA [K]:=ITEM.
6. [Reset N] Set N:=N+1
7. EXIT.

The following algorithm deletes the Kth element from a linear array LA and assigns it to a variable ITEM.

Algorithm for Deletion: (Deletion from a Linear Array)

DELETE (LA, N, K, ITEM)

Here LA is a Linear Array with N elements and K is the positive integer such that K<=N. This algorithm deletes the Kth element from LA.

1. Set ITEM: = LA [k].
2. Repeat for J = K to N – 1.
[Move J + 1st element upward] Set LA [J]: =
LA [J +1].[End of loop]
3. [Reset the number N of elements in LA] Set N: = N-1
4. EXIT

MULTIDIMENSIONAL ARRAY

1. Array having more than one subscript variable is called **Multi-Dimensional array**.
2. Multi Dimensional Array is also called as **Matrix**.

Consider the Two dimensional array -

1. Two Dimensional Array requires **Two Subscript Variables**
2. Two Dimensional Array stores the values in the form of matrix.
3. One Subscript Variable denotes the –**Row**ll of a matrix.
4. Another Subscript Variable denotes the –**Column**ll of a matrix.

Declaration and Use of Two Dimensional Array :

```
int a[3][4];
Use :
for(i=0;i
<row;i+
+)

  for(j=0;j<col;j++)
  {
  printf("%d",a[i][j]);
}
```

Meaning of Two Dimensional Array :

1. Matrix is having 3 rows (i takes value from 0 to 2)
2. Matrix is having 4 Columns (j takes value from 0 to 3)
3. Above Matrix 3×4 matrix will have 12 blocks having 3 rows & 4 columns.
4. Name of 2-D array is **a** and each block is identified by the row & column Number.
5. Row number and Column Number Starts from 0.

Cell Location	Meaning
a[0][0]	0th Row and 0th Column
a[0][1]	0th Row and 1st Column
a[0][2]	0th Row and 2nd Column

a[0][3]	0th Row and 3rd Column
a[1][0]	1st Row and 0th Column
a[1][1]	1st Row and 1st Column
a[1][2]	1st Row and 2nd Column
a[1][3]	1st Row and 3rd Column
a[2][0]	2nd Row and 0th Column
a[2][1]	2nd Row and 1st Column
a[2][2]	2nd Row and 2nd Column
a[2][3]	2nd Row and 3rd Column

Two-Dimensional Array : Summary with Sample Example

Summary Point	**Explanation**
No of Subscript Variables Required	2
Declaration	a[3][4]
No of Rows	3
No of Columns	4
No of Cells	12
No of for loops required to iterate	2

MEMORY REPRESENTATION

1. 2-D arrays are Stored in contiguous memory location **row wise**.

2. X 3 Array is shown below in the first Diagram.
3. Consider **3×3 Array is stored in Contiguous memory** location whichstarts from 4000 .
4. Array element **a[0][0]** will be stored at address **4000** again **a[0][1]** will bestored to next memory location i.e Elements stored

row-wise

5. After **Elements of First Row are stored** in appropriate memory location ,elements of next row get their corresponding mem. locations.
6. This is integer array so each element requires 2 bytes of memory.

Array Representation:

- Column-major
- Row-major

Arrays may be represented in Row-major form or Column-major form. In Row- major form, all the elements of the first row are printed, then the elements of the second row and so on up to the last row. In Column-major form, all the elements of the first column are printed, then the elements of the second column and so on up to the last column. The ‗C' program to input an array of order m x n and print the array contents in row major and column major is given below. The following array elements may be entered during run time to test this program:

Output:

Row Major:

1 2 3
4 5 6
7 8 9

Column Major:

1 4 7
2 5 8
3 6 9

Basic Memory Address Calculation :

a[0][1] = a[0][0] + Size of Data Type

Element	Memory Location
a[0][0]	4000
a[0][1]	4002
a[0][2]	4004
a[1][0]	4006
a[1][1]	4008
a[1][2]	4010
a[2][0]	4012
a[2][1]	4014
a[2][2]	4016

Array and Row Major, Column Major order arrangement of 2 d array

An array is a list of a finite number of homogeneous data elements. The number of elements in an array is called the array length. Array length can be obtained from the index set by the formula

Length=UB-LB+1

Where UB is the largest index , called the upper bound and LB is the smallest index , called the lower bound. Suppose int Arr[10] is an integer array. Upper bound of this array is 9 and lower bound of this array is 0 ,so the length is 9- 0+1=10.

In an array ,the elements are stored successive memory cells. Computer doesnot need to keep track of the address of every elements in memory. It will keep the address of the first location only and that is known as base **address of an array**. Using the base address , address of any other location of an array can be calculated by the computer. Suppose Arr is an array whose base address is Base(Arr) and w is the number of memory cells required by each elements of the array Arr.The address of Arr[k] — k being the index value can be obtained by

Using the formula

Address(Arr[k])=Base(Arr)+w(k-LowerBound)

2 d Array :- Suppose Arr is a 2 d array. The first dimension of Arr contains the index set 0,1,2, . . . row-1 (the lower bound is 0 and the upper bound is row-1) and the second dimension contains the index set 0,1,2,… col-1(with lower bound0 and upper bound col-1.)

The length of each dimension is to be calculated .The multiplied result of both thelengths will give you the number of elements in the array.

Let‘s assume Arr is an two dimensional 2 X 2 array .The array may be stored in memory one of the following way :-

- Column by columni,e column major order
- Row by row , i,e in row major order. The following figure shows bothrepresentation of the above array.

By row-major order, we mean that the **elements in the array** are so arrangedthat the subscript at the extreme right varies fast than the subscript at it‘s left., while in column-major order , the subscript at the extreme left changes rapidly ,

Then the subscript at it‘s right and so on.

1,1

2,1

1,2

2,2

Column MjorOrder

1,1

1,2

2,1

2,2

Row major order

Now we know that computer keeps track of only the base address. So the address of any **specified location of an array** , for example Arr[j,k] of a 2 d array Arr[m,n] can be calculated by using the following formula :- (Column major order) Address(Arr[j,k])= base(Arr)+w[m(k-1)+(j-1)] (Row major order)

Address(Arr[j,k])=base(Arr)+w[n(j-1)+(k-1)] For example Arr(25,4) is an array with base value 200.w=4 for this array. The address of Arr(12,3) can be calculated using row-major order as

Address(Arr(12,3))=200+4[4(12-1)+(3-1)]
=200+4[4*11+2]
=200+4[44+2]
=200+4[46]
=200+184
=384

Again using column-major order
Address(Arr(12,3))=200+4[25(3-1)+(12-1)]
=200+4[25*2+11]
=200+4[50+11]

=200+4[61]
=200+244
=444

SPARSE MATRIX

Matrix with relatively a high proportion of zero entries are called sparse matrix. Two general types of n-square sparse matrices are there which occur in various

applications are mention in figure below(It is sometimes customary to omit blocksof zeros in a matrix as shown in figure below)

```
4
3  -5
1   0   6
-7  8  -1  3
5  -2   0   -8
```

Triangular matrix

```
5  -3
1   4   3
    9  -3  6
        2   4  -7
            3   0
```

Tridiagonal matrix

Triangular matrix

This is the matrix where all the entries above the main diagonal are zero or equivalently where non-zero entries can only occur on or below the main diagonal is called a (lower)Triangular matrix.

Tridiagonal matrix

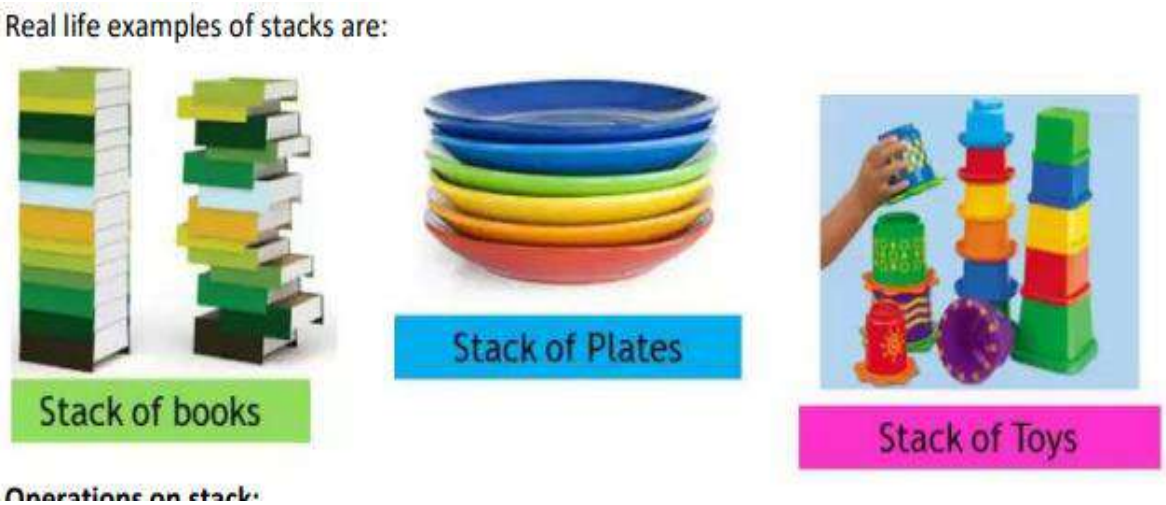
This is the matrix where non-zero entries can only occur on the diagonal or on elements immediately above or below the diagonal is called a Tridiagonal matrix. The natural method of representing matrices in memory as two-dimensionalarrays may not be suitable for sparse matrices i.e. one may save space bystoring only those entries which may be non-zero.

UNIT-IV

STACKS & QUEUES

A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (**front and rear**). It contains only one pointer **top pointer** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a ***stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.***

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed. Real life examples of stacks are:



Operations on stack: The two basic operations associated with stacks are:

1. Push
2. Pop

While performing push and pop operations the following test must be conducted on the stack. a) Stack is empty or not b) stack is full or not

1**. PUSH**: Push operation is used to add new elements in to the stack. At the time of addition first check the stack is full or not. If the stack is full it generates an error message "stack overflow".

2. **POP:** Pop operation is used to delete elements from the stack. At the time of deletion first check the stack is empty or not. If the stack is empty it generates an error message "stack underflow". All insertions and deletions take place at the same end, so the last element

added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

REPRESENTATION OF STACK (OR) IMPLEMENTATION OF STACK:

The stack should be represented in two ways:

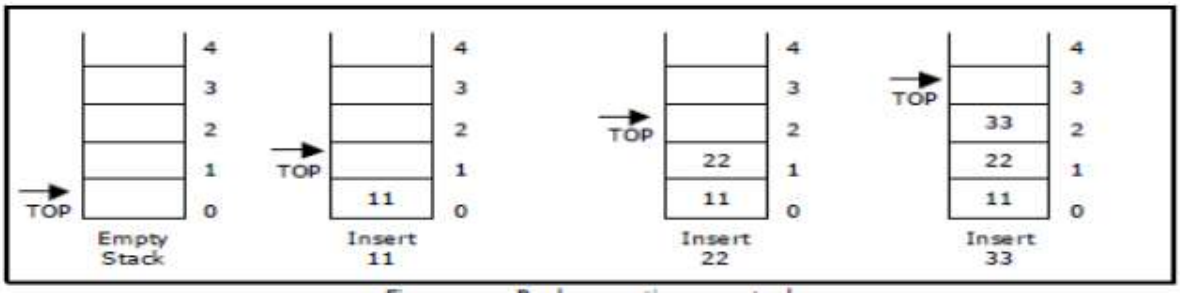
1. Stack using array
2. Stack using linked list

Array

The **array implementation** aims to create an array where the first element (usually at the zero-offset) is the bottom. That is, array[0] is the first element pushed onto the stack and the last element popped off. The program must keep track of the size, or the length of the stack.

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a stack overflow condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a stack underflow condition.

1.push(): When an element is added to a stack, the operation is performed by push(). Below Figure shows the creation of a stack and addition of elements using push().



Initially top=-1, we can insert an element in to the stack, increment the top value i.e top=top+1. We can insert an element in to the stack first check the condition is stack is full or not. i.e top>=size-1. Otherwise add the element in to the stack.

Algorithm 1: PUSH (STACK, TOP, MAXSTK, ITEM)

This procedure pushes an item on to a stack.

1. [Stack already filled]?

If TOP = MAXSTK, then: Print: OVERFLOW, and Return.

2. Set TOP: = TOP + 1. [Increase TOP by 1].
3. Set STACK [TOP]: = ITEM. [Inserts ITEM in new TOP position].
4. Return.

Algorithm 2: POP (STACK, TOP, ITEM)

This procedure deletes the TOP element of STACK and assigns it to the variable ITEM.

1. [Stack has an item to be removed]

If TOP = 0, then: Print: UNDERFLOW and Return.

2. Set ITEM: =STACK [TOP]. [Assign TOP element to ITEM].
3. Set TOP: = TOP – 1 [Decrease TOP by 1].
4. Return.

If we use a dynamic array, then we can implement a stack that can grow or

shrink as much as needed. The size of the stack is simply the size of the dynamic array. A dynamic array is a very efficient implementation of a stack, since adding items to or removing items from the end of a dynamic array is dynamically with respect to time.

APPLICATIONS OF STACK:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to evaluate a postfix expression.
3. Stack is used to convert an infix expression into postfix/prefix form.
4. In recursion, all intermediate arguments and return values are stored on the processor's stack.
5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off. Converting and evaluating Algebraic expressions

CONVERTING AND EVALUATING ALGEBRAIC EXPRESSIONS:

An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, *, /, ^ etc.

An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:

INFIX: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands. Example: A + B

PREFIX: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation. Example: + A B

POSTFIX: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as

suffix notation and is also referred to reverse polish notation.

Example: A B + Conversion from infix to postfix:

POLISH NOTATION

Polish notation, also known as Polish prefix notation or simply prefix notation is a form of notation for logic, arithmetic, and algebra. Its distinguishing feature is that it places operators to the left of their operands. If the arity of the operators is fixed, the result is a syntax lacking parentheses or other brackets that can still be parsed without ambiguity. The Polish logician Jan Łukasiewicz invented this notation in 1924 in order to simplify sentential logic.

The term *Polish notation* is sometimes taken (as the opposite of *infix notation*) to also include Polish *postfix* notation, or **Reverse Polish** notation, in which the operator is placed *after* the operands.

When Polish notation is used as a syntax for mathematical expressions by interpreters of programming languages, it is readily parsed into abstract syntax trees and can, in fact, define a one-to-one representation for the same.

CONVERSION BETWEEN INFIX, PREFIX AND POSTFIX NOTATION

We are accustomed to write arithmetic expressions with the operation between the two operands: a+b or c/d. If we write a+b*c, however, we have to apply precedence rules to avoid the ambiguous evaluation (add first or multiply first?). There's no real reason to put the operation between the variables or values. They can just as well precede or follow the operands. You should note the advantage of prefix and postfix: the need for precedence rules and parentheses are eliminated.

Example of expression in three notations.

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +
(a + b) * (c - d)	* + a b - c d	a b + c d - *
b * b - 4 * a * c		
40 - 3 * 5 + 1		

Postfix expressions are easily evaluated with the aid of a stack.

CONVERSION FROM INFIX TO POSTFIX:

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.

   b) If the scanned symbol is an operand, then place directly in the postfix expression (output).

   c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.

   d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or greater than or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

The three important features of postfix expression are:

1. The operands maintain the same order as in the equivalent infix expression.
2. The parentheses are not needed to designate the expression unambiguously.
3. While evaluating the postfix expression the priority of the operators is no longer relevant. We consider five binary operations: +, -, *, / and $ or ↑ (exponentiation).

For these binary operations, the following in the order of precedence (highest to lowest):

OPERATOR	PRECEDENCE	VALUE
Exponentiation ($ or ↑ or ^)	Highest	3
*, /	Next highest	2
+, -	Lowest	1

Example 1:

Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

EVALUATION OF POSTFIX EXPRESSION:

The postfix expression is evaluated easily by the use of a stack.

1. When a number is seen, it is pushed onto the stack;
2. When an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.
3. When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage.

Example 1:

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

RECURSION

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily.

Example − a function calling itself.

```
int function(int value) {
   if(value < 1)
      return;
   function(value - 1);

   printf("%d ",value);
}
```

Example − a function that calls another function which in turn calls it again.

```
int function1(int value1) {
   if(value1 < 1)
      return;
   function2(value1 - 1);
   printf("%d ",value1);
}
int function2(int value2) {
   function1(value2);
```

```
}
```

Properties

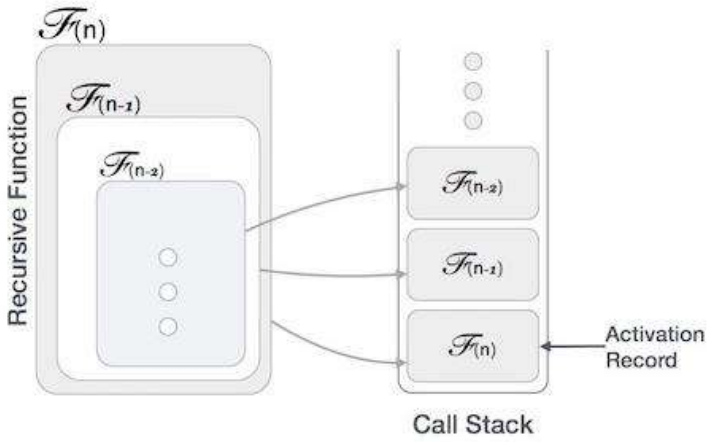
A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have −

- **Base criteria** − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- **Progressive approach** − The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

Implementation

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.

Types of Recursion

There are two types of Recursion

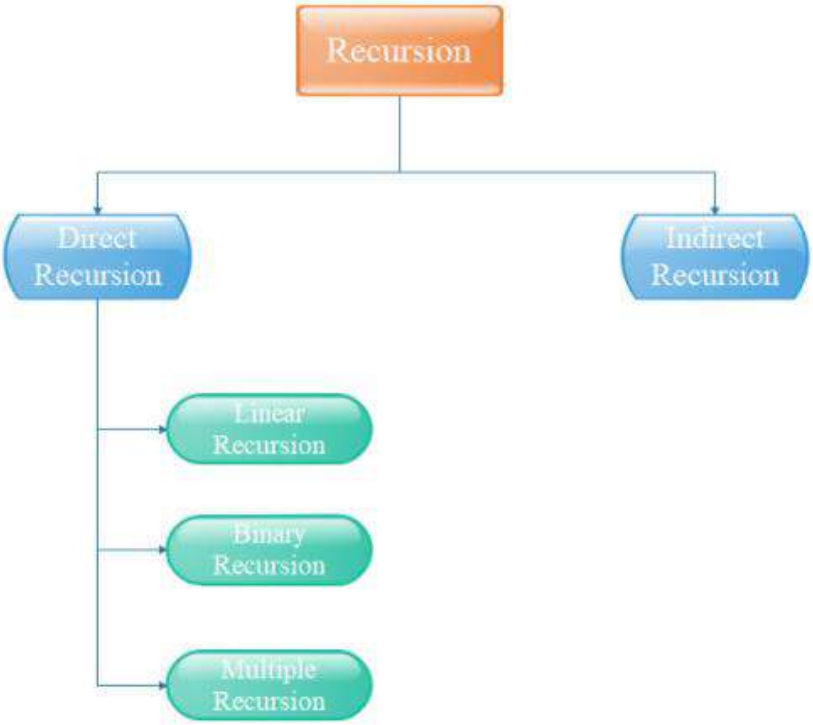
- Direct recursion
- Indirect recursion

Direct Recursion

When in the body of a method there is a call to the same method, we say that the method is **directly recursive**.

There are three types of Direct Recursion

- Linear Recursion
- Binary Recursion
- Multiple Recursion



Linear Recursion

- Linear recursion begins by testing for a set of base cases there should be at least one.

In Linear recursion we follow as under :

- Perform a single recursive call. This recursive step may involve a test that decides which of several possible recursive calls to make, but it should ultimately choose to make just one of these calls each time we perform this step.
- Define each possible recursion call, so that it makes progress towards a base case.

Binary Recursion

- Binary recursion occurs whenever there are two recursive calls for each non base case.

Multiple Recursion

- In multiple recursion we make not just one or two but many recursive calls.

EXAMPLE 1: CALCULATING THE FACTORIAL OF A NUMBER

Calculating the factorial of a number is a common problem that can be solved recursively. As a reminder, a factorial of a number, n, is defined by n! and is the result of multiplying the numbers 1 to n. So, 5! is equal to 5*4*3*2*1, resulting in 120.

```
function factorial(n) {
    if(n === 1 || n === 0) { // base case
        return 1;
    }
    return n * factorial(n - 1); // recursive call
}
```

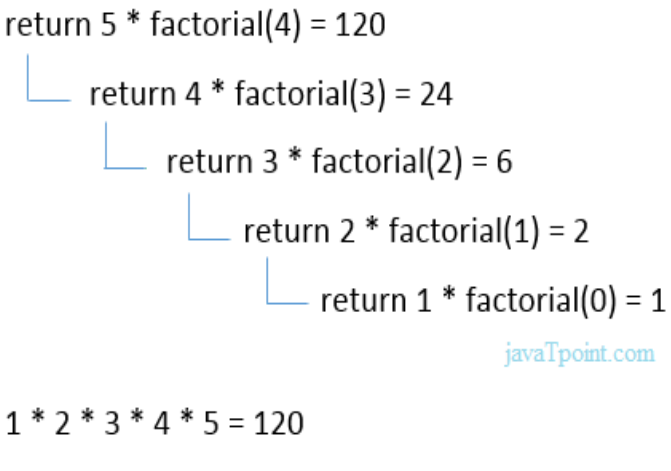
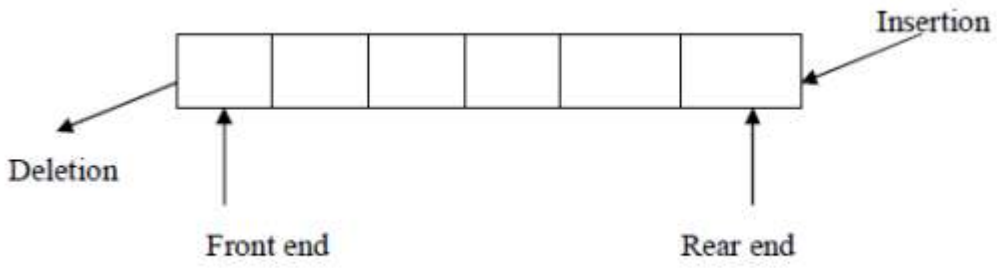


Fig: Recursion

QUEUE ADT

A queue is an ordered collection of data such that the data is inserted at one end and deleted from another end. The key difference when compared stacks is that in a queue the information stored is processed first-in first-out or FIFO. In other words the information receive from a queue comes in the same order that it was placed on the queue.

Queue is a linear list of elements in which deletions can take place only at one end, called the front and insertions can take place only at the other end, called the rear. The terms“front” and “rear” are used in describing a linear list only when it is implemented as a queue.
Queue are also called first-in first-out (FIFO) lists, since the first elements enter a queue will be the first element out of the queue. In other words, the order in which elements enter a queue is the order in which they leave. This contrasts with stacks, which are last-in first-out (LIFO) lists.
Queues abound in everyday life. The automobiles waiting to pass through an intersection form a queue. In which the first car in line is the first car through; the people waiting in line at a bank form a queue, where the first person in line is the
first person to be waited on; and so on. An important example of a queue in computer science occurs in a timesharing system, in which programs with the same priority form a queue while waiting to be executed



REPRESENTATION OF QUEUES:

Queues may be represented in the computer in various ways, usually by meansat one-way lists or linear arrays

Unless otherwise stated or implied, each of our queues will be maintained by a linear array QUEUE and two pointer variables: FRONT, containing the location of the front element of the queue; and REAR, containing the location of the rear element of the queue. The condition FRONT = NULL will indicate that the queue is empty. Following figure shows the way the array in Figure will be stared in memory using an array QUEUE with N elements.

Figure also indicates the way elements will be deleted from the queue and the way new elements will be added to the queue. Observe that whenever an element is deleted from the queue, the value of FRONT is increased by 1; this can be implemented by the assignment

FRONT: = Rear + 1

Similarly, whenever an element is added to the queue, the value of REAR is increased by 1; this can be implemented by the assignment

REAR: = Rear +1

This means that after N insertion, the rear element of the queue will occupy QUEUE [N] or, in other words; eventually the queue will occupy the last part of the array. This occurs even though the queue itself may not contain many elements.

Suppose we want to insert an element ITEM into a queue will occupy the last part of the array, i.e., when REAR=N. One way to do this is to simply move the entire queue to thee beginning of the array, changing FRONT and REAR accordingly, and then inserting ITEM as above. This procedure may be very expensive. The procedure we adopt is to assume that the array QUEUE is circular, that is, that QUEUE [1] comes after QUEUE [N] in the array. With this assumption, we insert ITEM into the queue by assigning ITEM to QUEUE [1]. Specifically, instead of Increasing REAR to N+1, we reset REAR=1 and then assign QUEUE [REAR]: = ITEM

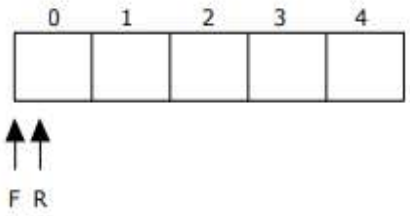
Similarly, if FRONT = N and an element of QUEUE is deleted, we reset FRONT = 1 instead of increasing FRONT to N +1

Suppose that our queue contains only one element, i.e.,

suppose that FRONT = REAR # NULL

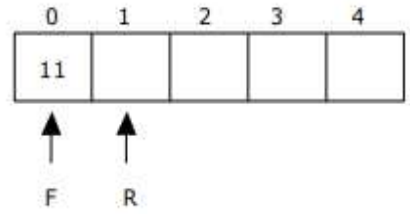
And suppose that the element is deleted. Then we assign FRONT: = NULL

and REAR: = NULL



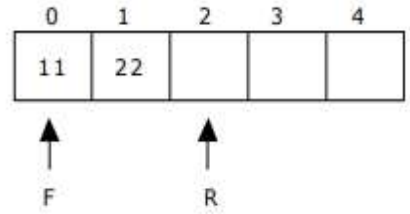
Queue Empty
FRONT = REAR = 0

Now, insert 11 to the queue. Then queue status will be:



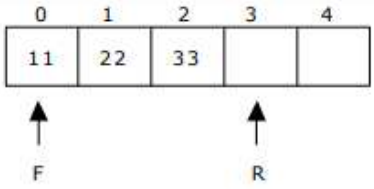
REAR = REAR + 1 = 1
FRONT = 0

Next, insert 22 to the queue. Then the queue status is:



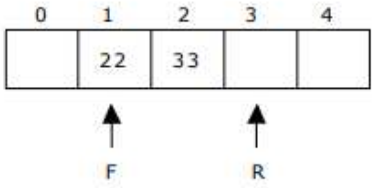
REAR = REAR + 1 = 2
FRONT = 0

Again insert another element 33 to the queue. The status of the queue is:



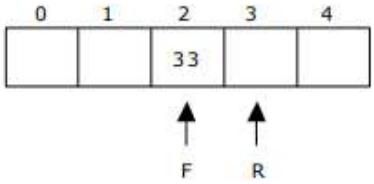
REAR = REAR + 1 = 3
FRONT = 0

Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



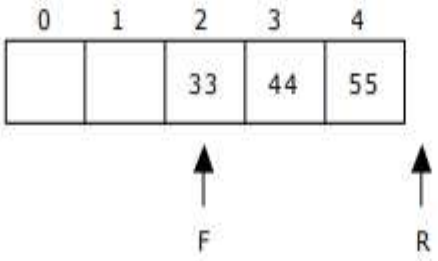
REAR = 3
FRONT = FRONT + 1 = 1

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



REAR = 3
FRONT = FRONT + 1 = 2

Now, insert new elements 44 and 55 into the queue. The queue status is:



REAR = 5
FRONT = 2

OPERATIONS ON QUEUE:

A queue have two basic operations:

a)adding new item to the queue

b) removing items from queue.

The operation of adding new item on the queue occurs only at one end of the queue called the rear or back. The operation of removing items of the queue occurs at the other end called the front. For insertion and deletion of an element from a queue, the array elements begin at 0 and the maximum elements of the array is maxSize. The variable front will hold the index of the item that is considered the front of the queue, while the rear variable will hold the index of the last item in the queue. Assume that initially the front and rear variables are initialized to -1.

Like stacks, underflow and overflow conditions are to be checked before operations in a queue. Queue empty or underflow condition is

Queue empty or underflow condition is

```
if((front>rear)||front= =-1)
    cout<"Queue is empty";
```

Queue Full or overflow condition is

```
if((rear==max)
    cout<"Queue is full";
```

ENQUEUE OPERATION

Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks. The following steps should be taken to enqueue (insert) data into a queue –

Step 1 − Check if the queue is full.

- Step 2 − If the queue is full, produce overflow error and exit
- Step 3 − If the queue is not full, increment rear pointer to point the next empty space.
- Step 4 − Add data element to the queue location, where the rear is pointing.
- Step 5 − return

PROCEDURE ENQUEUE(DATA)

if queue is full return overflow

endif

rear ← rear + 1

queue[rear] ← data

return true

end procedure

DEQUEUE OPERATION

Accessing data from the queue is a process of two tasks − access the data where front is pointing and remove the data after access.

The following steps are taken to perform dequeue operation –

Step 1 − Check if the queue is empty.

- Step 2 − If the queue is empty, produce underflow error and exit
- Step 3 − If the queue is not empty, access the data where front is pointing
- Step 4 − Increment front pointer to point to the next available data element.
- Step 5 − Return success.

ALGORITHM FOR DEQUEUE OPERATION

procedure dequeue

if queue is empty

return underflow

end

if data = queue[front]

front ← front + 1

return true

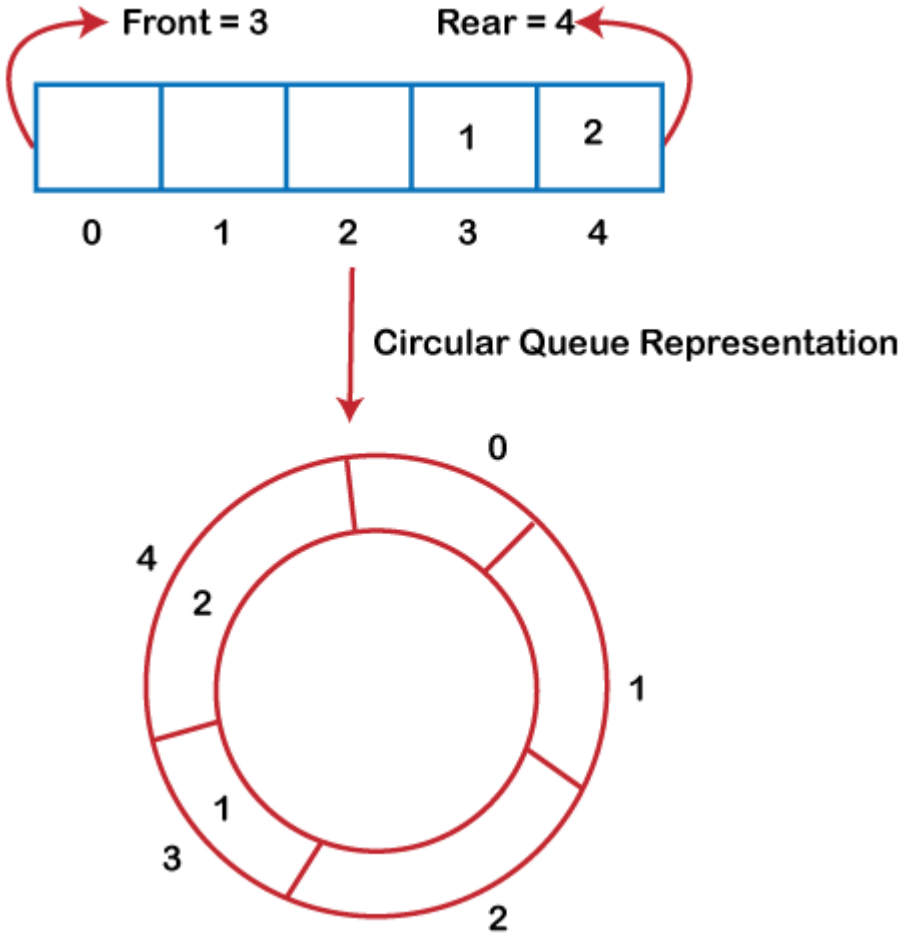
end procedure

Circular Queue

Why was the concept of the circular queue introduced?

There was one limitation in the array implementation of Queue

. If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.



As we can see in the above image, the rear is at the last position of the Queue and front is pointing somewhere rather than the 0^{th} position. In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue. There is one solution to avoid such wastage of memory space by shifting both the elements at the left and adjust the front and rear end accordingly. It is not a practically

good approach because shifting all the elements will consume lots of time. The efficient approach to avoid the wastage of the memory is to use the circular queue data structure.

What is a Circular Queue?

A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a ***Ring Buffer***.

Operations on Circular Queue

The following are the operations that can be performed on a circular queue:

- **Front:** It is used to get the front element from the Queue.
- **Rear:** It is used to get the rear element from the Queue.
- **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
- **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.

Applications of Circular Queue

The circular Queue can be used in the following scenarios:

- **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.
- **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.
- **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of

time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Enqueue operation

The steps of enqueue operation are given below:

- First, we will check whether the Queue is full or not.
- Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
- When we insert a new element, the rear gets incremented, i.e., ***rear=rear+1***.

Scenarios for inserting an element

There are two scenarios in which queue is not full:

- **If rear != max - 1**, then rear will be incremented to **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
- **If front != 0 and rear = max - 1**, it means that queue is not full, then set the value of rear to 0 and insert the new element there.

There are two cases in which the element cannot be inserted:

- When **front ==0** && **rear = max-1**, which means that front is at the first position of the Queue and rear is at the last position of the Queue.
- front== rear + 1;

Algorithm to insert an element in a circular queue

Step 1: IF (REAR+1)%MAX = FRONT
Write " OVERFLOW "
Goto step 4
[End OF IF]

Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE IF REAR = MAX - 1 and FRONT ! = 0
SET REAR = 0
ELSE

SET REAR = (REAR + 1) % MAX
[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT

Dequeue Operation

The steps of dequeue operation are given below:

- First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset to -1.

Algorithm to delete an element from the circular queue

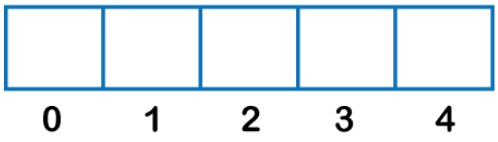
Step 1: IF FRONT = -1
Write " UNDERFLOW "
Goto Step 4
[END of IF]

Step 2: SET VAL = QUEUE[FRONT]

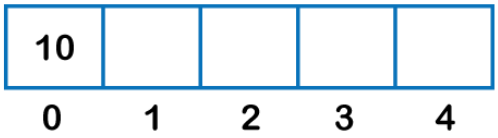
Step 3: IF FRONT = REAR
SET FRONT = REAR = -1
ELSE
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = FRONT + 1
[END of IF]
[END OF IF]

Step 4: EXIT

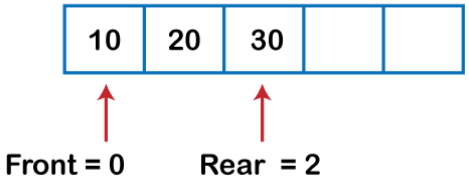
et's understand the enqueue and dequeue operation through the diagrammatic representation.

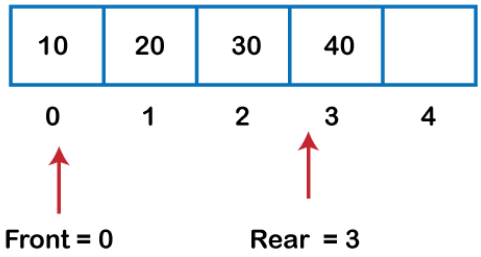


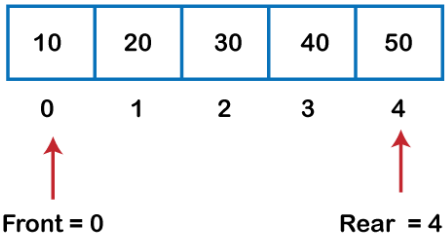
Front = -1
Rear = -1

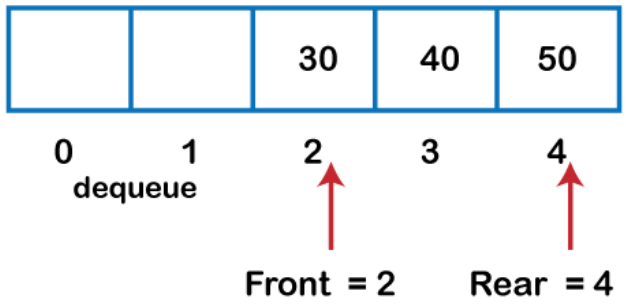


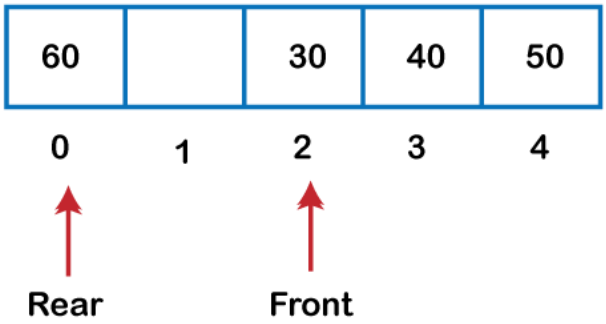
Front = 0
Rear = 0

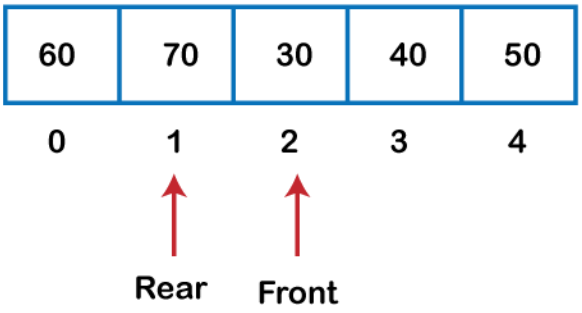






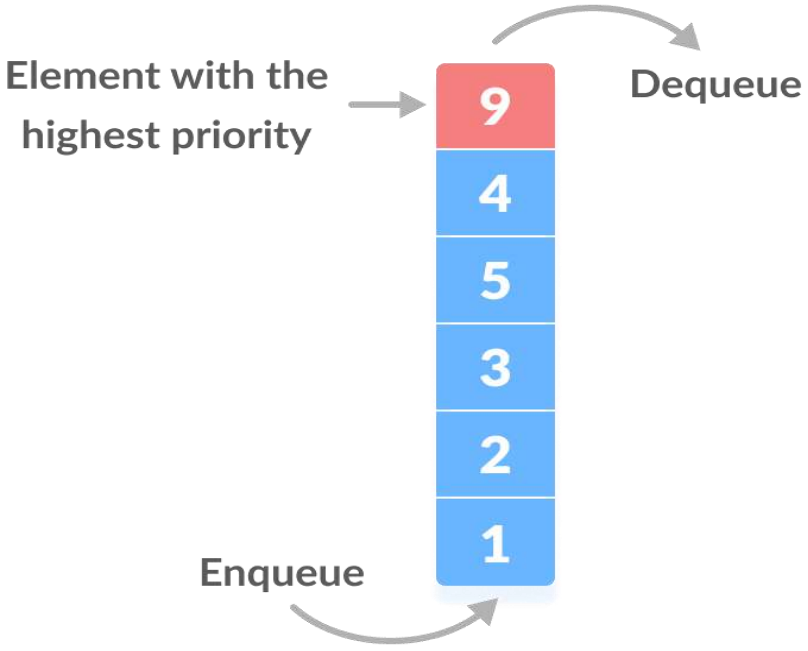

30
40
50
0
1
2
3
4
dequeue
Front = 2
Rear = 4


60
30
40
50
0
1
2
3
4
Rear
Front


60
70
30
40
50
0
1
2
3
4
Rear
Front

PRIORITY QUEUE

- A priority queue is a **special type of queue** in which each element is associated with a **priority value**. And, elements are served on the basis of their priority. That is, higher priority elements are served first.
- However, if elements with the same priority occur, they are served according to their order in the queue.
- **Assigning Priority Value**
- Generally, the value of the element itself is considered for assigning the priority. For example,
- The element with the highest value is considered the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element.
- We can also set priorities according to our needs.



DIFFERENCE BETWEEN PRIORITY QUEUE AND NORMAL QUEUE

In a queue, the **first-in-first-out rule** is implemented whereas, in a priority queue, the values are removed **on the basis of priority**. The element with the highest priority is removed first.

DIFFERENCE BETWEEN STACK AND QUEUE

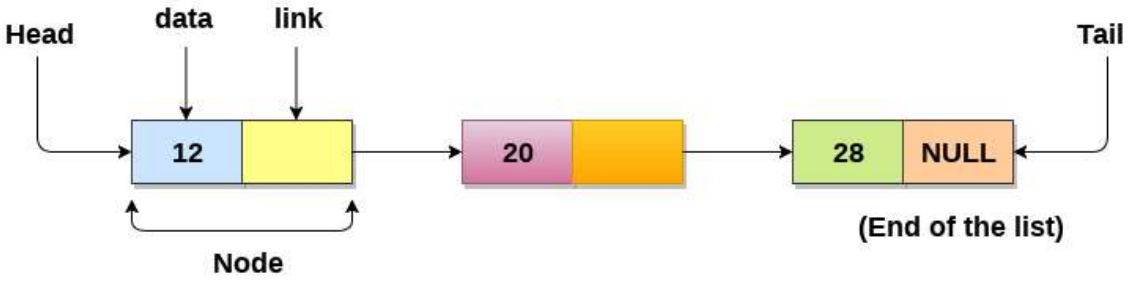
Stack	**Queue**
A Linear List Which allows insertion or deletion of an element at one end only is called as Stack	A Linear List Which allows insertion at one end and deletion at another end is called as Queue
Since insertion and deletion of an element are performed at one end of the stack, the elements can only be removed in the opposite order of insertion.	Since insertion and deletion of an element are performed at opposite end of the queue, the elements can only be removed in the same order of insertion.
Stack is called as Last In First Out (LIFO) List.	Queue is called as First In First Out (FIFO) List.
The most and least accessible elements are called as TOP and BOTTOM of the stack	Insertion of element is performed at FRONT end and deletion is performed from REAR end
Example of stack is arranging plates in one above one.	Example is ordinary queue in provisional store.
Insertion operation is referred as PUSH and deletion operation is referred as POP	Insertion operation is referred as ENQUEUE and deletion operation is referred as DQUEUE
Function calling in any languages uses Stack	Task Scheduling by Operating System uses queue

UNIT-V

LINKED LIST

LIST

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.



USES OF LINKED LIST

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

WHY USE LINKED LIST OVER ARRAY?

Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.

Array contains following limitations:

1. The size of array must be known in advance before using it in the program.
2. Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.

3. All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

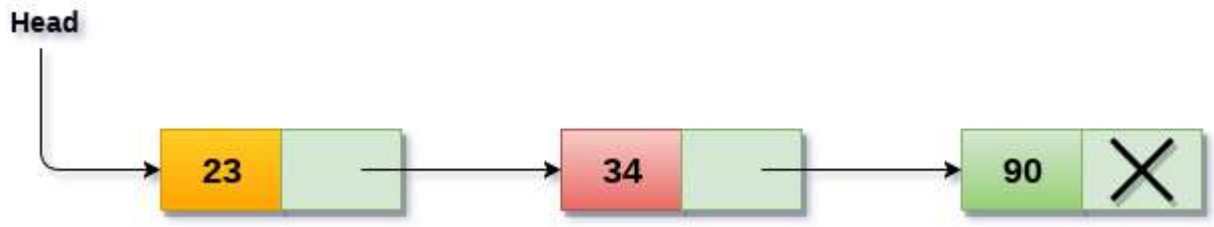
1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

SINGLY LINKED LIST OR ONE WAY CHAIN

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.



In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Advantages of linked list:-

Dynamic data structure that can grow an string. Efficient memory utilization (exact amount of data storage). Insertion deletion & pupation

are easy & efficient. Data stored in RAM but not sequential .

Disadvantages of linked list:-

More memory space is needed if no. of filed are more. Logical & physical ordering of node are different. Searching is solve . Difficult to program because pointer manipulation is required.

Types of linked list:-

Linear linked list or one way linked list or single list. Double linked list or two way linked list are two way linked list. Circular linked list is two types i.e.

(1) Single circular list

(2) Double circular list.

- Linear linked list:- It is a one way collection of nodes where the linear order is maintained by pointers. Nodes are not in sequence, each node implemented in comp. by a self referential structure . Each node is divided in two parts. First part contain the information of the element (INFO). Second part is linked field contains the add. Of next node in the list (LINK) field or next pointer filed .In c linked list is created using structured pointer and Mallow (Allocation of memory). The structure of a node is strict node.

```
{
Into info;
Strict node*link;
};
```

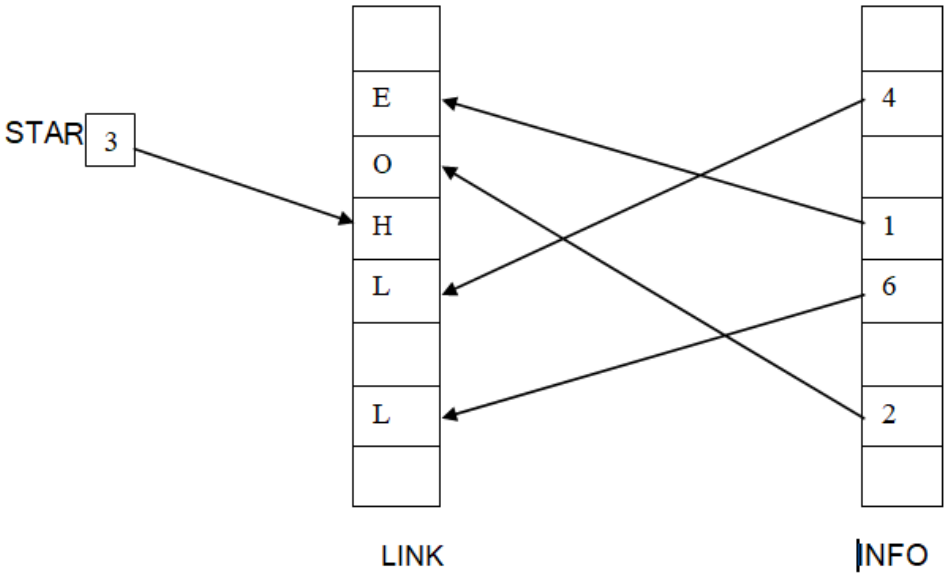
The new node is created and addresses of the new node is assigned to stackhas start.

{START =(Strict node*) mallow(size of (strict node))}

Representation of the linked list in memory:-

Let list be a linked list. Then list required to linear arrays called INFO and LINK.

Such that INFO[k] and LINK[k] contain the information part & next pointerdenoted by a NULL which indicates the end of the LIST.



Memory allocation of the linear linked list:-

The computer maintains a special list which consist of a list of all free memory calls & also has its own pointer is called the list of available space or the free storage list or the free pool .

Suppose insertion are to be performed on linked list then unused memory calls I the array will also be linked to gather to form a linked list using AVAIL. As its list pointer variable such a data structure will be denoted by writing LIST(INFO,LINK,START,AVAIL)

Garbage collection definition:-

The operating system of a computer may periodically collect all the deleted space on to the free storage list. Any technique which does these collections is called garbage collection.

When we delete a particular note from an existing linked list or delete the linked list the space occupied by it must be given back to the free pool. So that the

memory can be the used by some other program that needs memory space.

To the free pool is done.

The operating system will perform this operation whenever it finds the CPU idle or whenever the programs are falling short of memory space. The OS scans through the entire memory cell & marks those cells. That are being by some program then it collects the entire cell which are not being used & add to the free pool. So that this cells can be used by other programs. This process is called garbage collection. The garbage collection is invisible to the programmer.

Traversing of linked list:-

Algorithm:-

Let list be a linked list in memory, this algorithm traverse LIST applying & operation PROCESS to each element of LIST. The variable PTR to point to the nodes currently being processed.

Step 1:-set PTR=START [initialize pointer PTR] Step 2:-repeat step 3 & step 4 while PTR! = NULLStep 3:-apply PROCESS to INFO[PTR]

STEP 4:-SET PTR=LINK [PTR]

[PTR now points to the next node]End of step 2 loop

Step 5:-exit

Algorithm for searching linked list:- SEARCH (INFO,LINK,START,ITEM,LOC)

LIST is a linked list in memory , this algorithm finds the location LOC of the node where ITEM first appears in LIST or sets , LOC=NULL.

Step 1:-set PTR=START[initialize pointer PTR]

Step 2:-repeat step 3 while PTR ! = NULL

Step 3:-if ITEM = INFO[PTR]

Then set LOC = PTR & exit

Else

Set PTR = LINK[PTR]

[PTR now points to next node][End of if structure]

End of step 2 loop

Step 4:-[Search is unsuccessful]Set LOC = NULL

Step 5:-Exit

Inserting the node at the beginning of the list:

-INSERT (INFO, LINK, START, AVAIL, ITEM)

Step 1:-[over flow?]

If AVAIL = NULL, then write over flow & exit

Step 2:-[REMOVE first node from AVAIL LIST]

Set NEW = AVAIL & AVAIL = LINK [AVAIL]

Step 3:-set INFO [NEW] = ITEM

[Copy is new data into new node]

Step 4:-set LINK [NEW] = START

[New node now points to original first node]

Step 5:-set START = NEW

[changes start so its point

to new node]Step 6:-exit

INSERTING AFTER A GIVEN NODE:-

INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM)

This algorithm inserts ITEM. so that ITEM follows the node with location[LOC] or insert ITEM as the first node when LOC = NULL

Step 1:-[over flow] if AVAIL = NULL, then write overflow & exit. Step 2:-[Remove first node from AVAIL list]

Set NEW = AVAIL & AVAIL = LINK [AVAIL]

Step 3:-set INFO [NEW] = ITEM

[Copy is new data into new node]

Step 4:-if LOC = NULL, then [insert as first node] Set LINK [NEW] = START & START = NEW

Else

[INSERT after node with location LOC]

Set LINK [NEW] = LINK [LOC] and LINK [LOC] = NEW

[End of if]

Step 5:-Exit

DELETION FROM A LINKED LIST:-

DEL (INFO, LINK, START, AVAIL, LOC, LOCP)

This algorithm deletes the node N with location LOC, LOCP is the locationof the node LOCP = NULL.

Step 1:-if LOCP = NULL, then set = START=LINK [START]

[Delete first node]

Else

Set = LINK [LOCP] = LINK [LOC]

[Deletes node N][End of if]

Step 2:-[Return deleted node to the AVAIL LIST] Set = LINK [LOC] = AVAIL & AVAIL = LOC

Step 3:-Exit

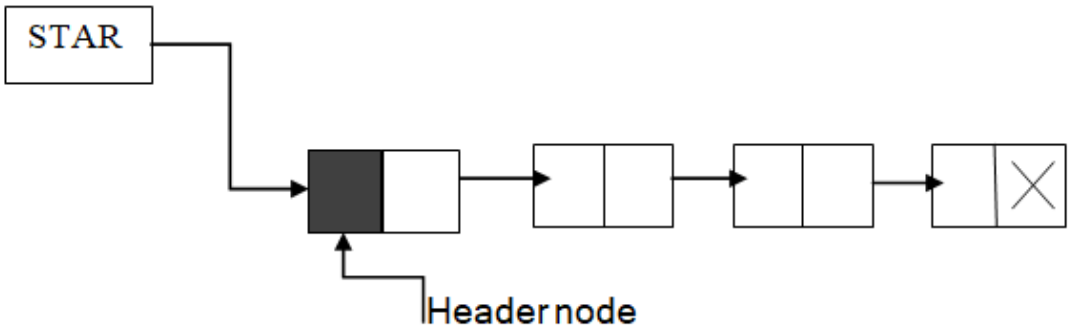
HEADER LINKED LIST:-

A header linked list is a special type of linked list. Which always contains a special node called header node at the beginning of the list so in a header linked list will not point to first node of the list. But start will contain the address of the header node.

There are two types of header linked list i.e. Grounded header linked list & Circular header linked list.

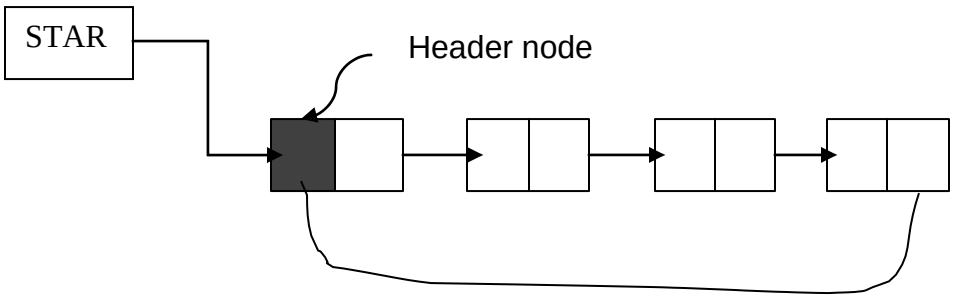
Grounded header linked list:-

It is a header linked list where last node contains the NULL pointer. LINK [start] = NULL indicates that a grounded header linked list is empty.



CIRCULAR HEADER LINKED LIST:-

It is a header linked list where the last node points back to the header node.



LINK [START] = START is indicates that a circular linked list is empty.

Circular header list are frequently used instead of ordinary linked list because

many operation are much easier to implement header list.

This comes from the following two properties, all circular header lists.

- The NULL pointer is not used & hence contains valid address.

Every ordinary node has a predecessor. So the first node may notrequired a special case

UNIT-VI

TREE

Tree - Terminology

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

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

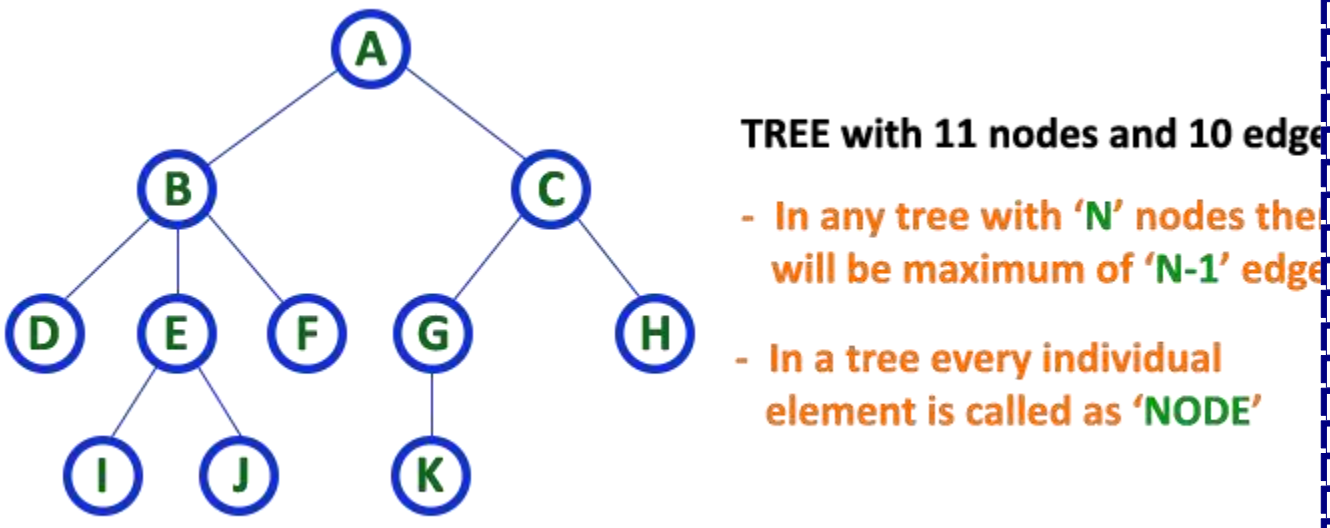
A tree data structure can also be defined as follows...

Tree data structure is a collection of data (Node) which is organized in hierarchical structure recursively

In tree data structure, every individual element is called as **Node**. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure.

In a tree data structure, if we have **N** number of nodes then we can have a maximum of **N-1** number of links.

Example

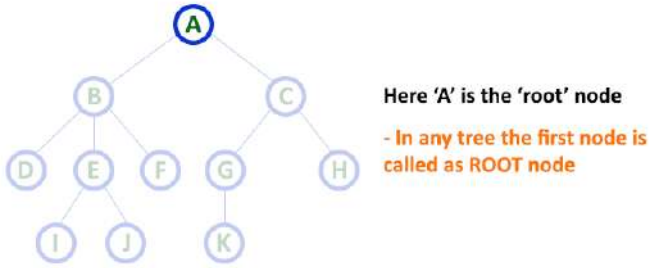


Terminology

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

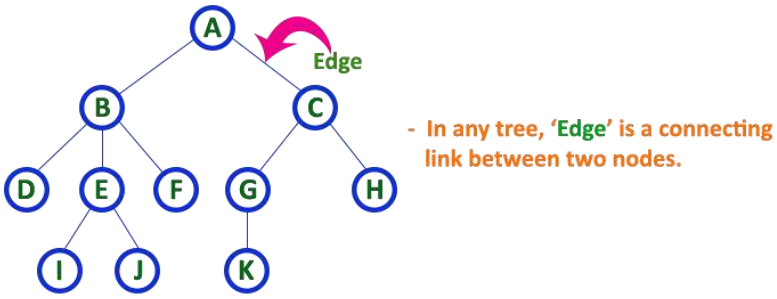
1. Root

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



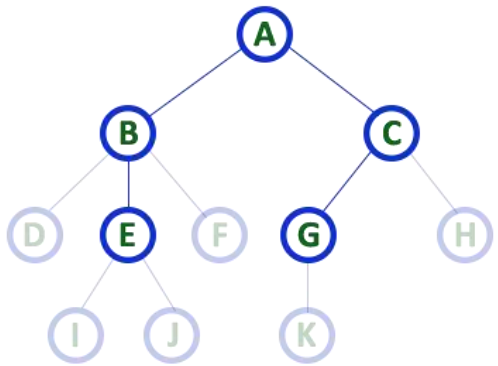
2. Edge

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



3. Parent

In a tree data structure, the node which is a predecessor of any node is called as **PARENT NODE**. In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "**The node which has child / children**".

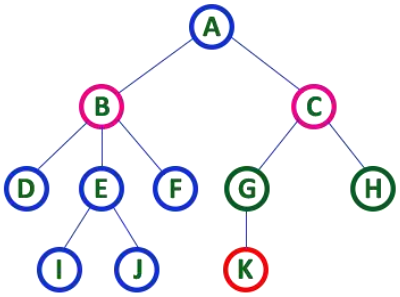


Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

4. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.

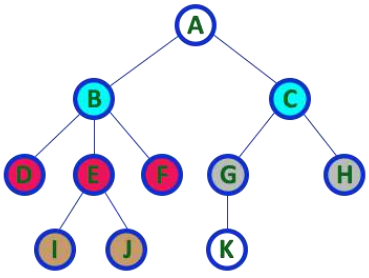


Here B & C are Children of A
Here G & H are Children of C
Here K is Child of G

- descendant of any node is called as CHILD Node

5. Siblings

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



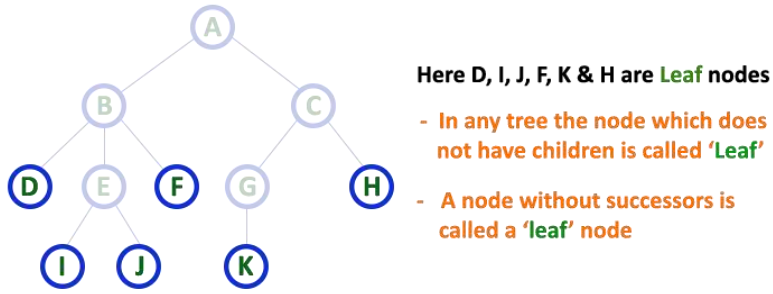
Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

6. Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

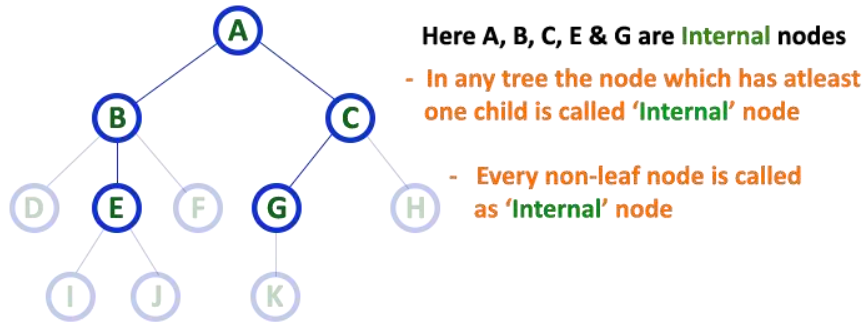
In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.



7. Internal Nodes

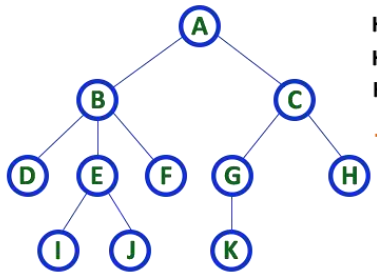
In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes.



8. Degree

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'

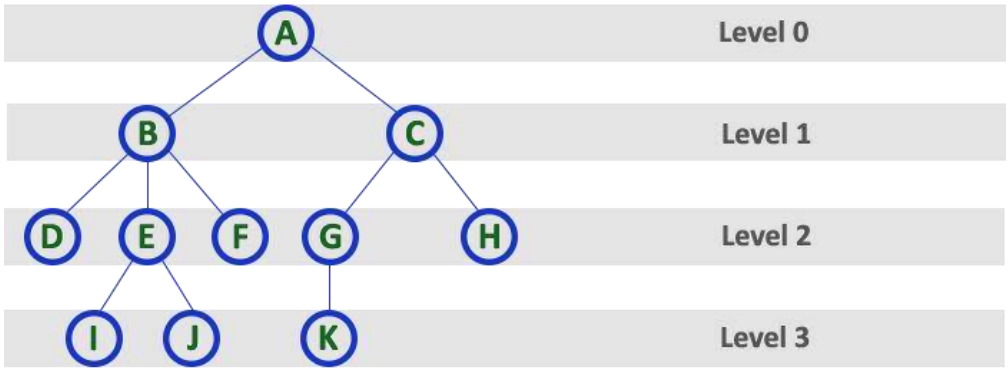


Here Degree of B is 3
Here Degree of A is 2
Here Degree of F is 0

- In any tree, 'Degree' of a node is total number of children it has.

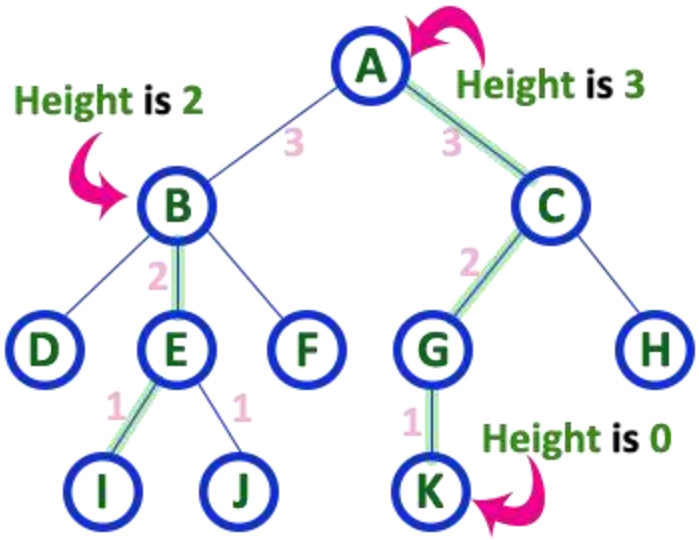
9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



10. Height

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

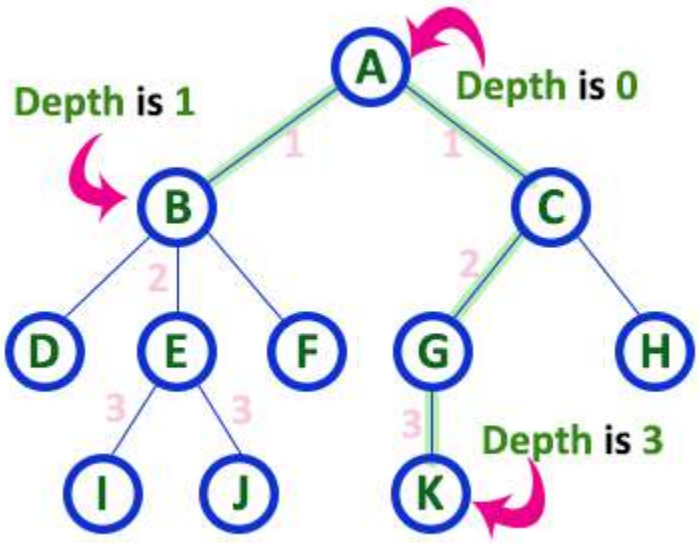


Here Height of tree is 3

- In any tree, 'Height of Node' is total number of Edges from lea to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

11. Depth

In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**

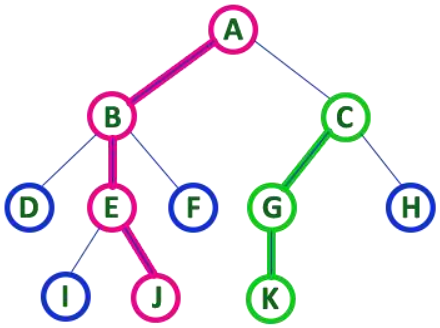


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from roo to that node.
- In any tree, 'Depth of Tree' is total number of edges from roo to leaf in the longest path.

12. Path

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



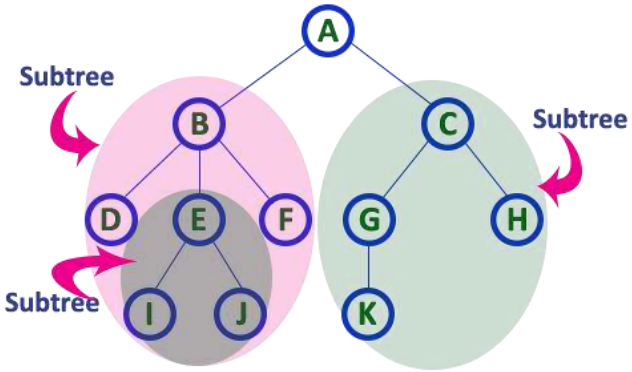
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K

13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



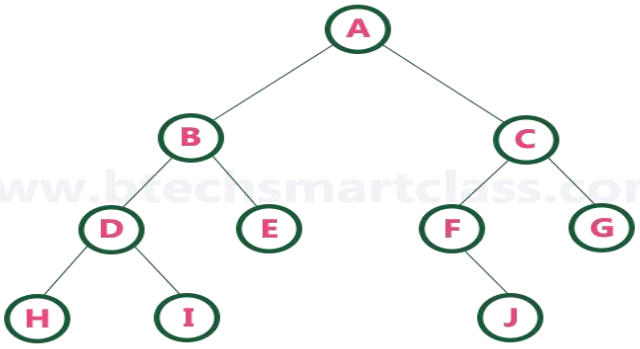
Binary Tree Datastructure

In a normal tree, every node can have any number of children. A binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**. One is known as a left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

Example



PROPERTIES OF BINARY TREE: Some of the important properties of a binary tree are as follows:

1. If h = height of a binary tree, then

a. Maximum number of leaves = 2h

b. Maximum number of nodes = 2h + 1 – 1

2. If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l + 1.

3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2l node at level l.

4. The total number of edges in a full binary tree with n node is n - 1

There are different types of binary trees and they are...

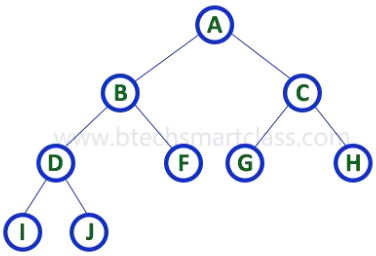
1. Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...

A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree

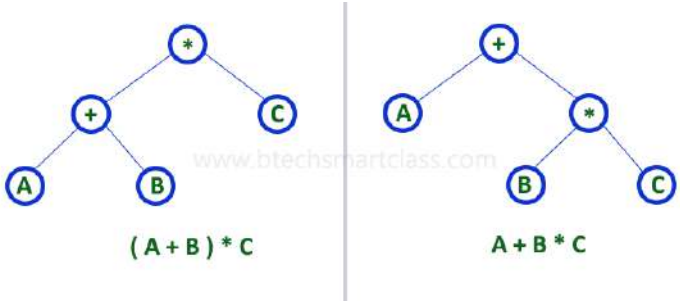
---

Strictly binary tree is also called as **Full Binary Tree** or **Proper Binary Tree** or **2-Tree**



Strictly binary tree data structure is used to represent mathematical expressions.

Example

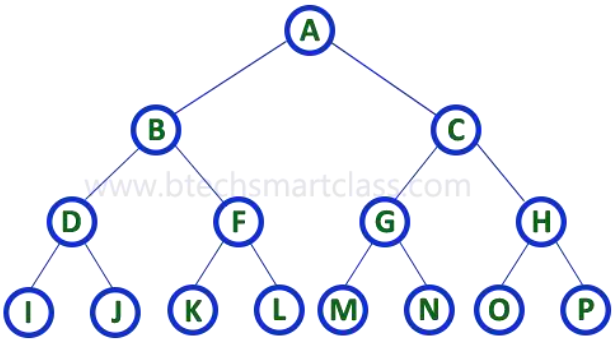


2. Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2^{level} number of nodes. For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes.

A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.

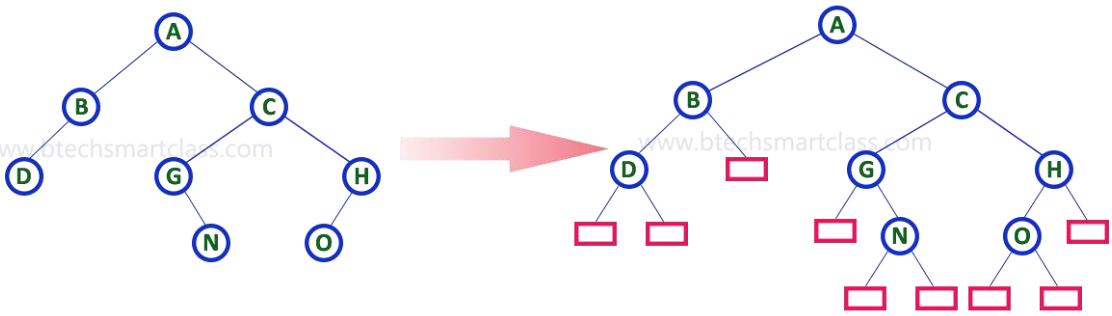
Complete binary tree is also called as **Perfect Binary Tree**



3. Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.



In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink colour).

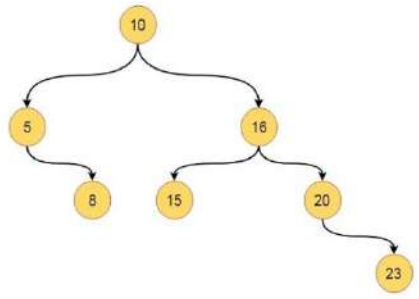
Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation**
2. **Linked List Representation**

Array Representation

Suppose we have one tree like this −



The array representation stores the tree data by scanning elements using level order fashion. So it stores nodes level by level. If some element is missing, it left blank spaces for it. The representation of the above tree is like below −

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	5	16	-	8	15	20	-	-	-	-	-	-	-	23

The index 1 is holding the root, it has two children 5 and 16, they are placed at location 2 and 3. Some children are missing, so their place is left as blank.

In this representation we can easily get the position of two children of one node by using this formula −

child1=2∗parentchild1=2∗parent

child2=(2∗parent⍰+1child2=(2∗parent⍰+1

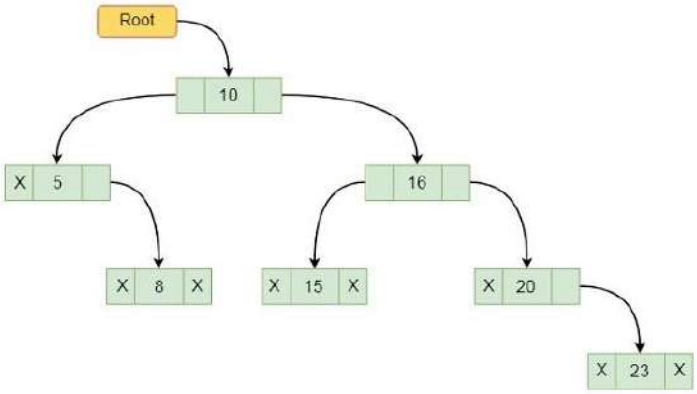
To get parent index from child we have to follow this formula −

parent=[child2]parent=[child2]

This approach is good, and easily we can find the index of parent and child, but it is not memory efficient. It will occupy many spaces that has no use. This representation is good for complete binary tree or full binary tree.

Linked List Representation

Another approach is by using linked lists. We create node for each element. This will be look like below −



Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges links we always start from the root head node. That is, we cannot random access a node in tree. There are three ways which we use to traverse a tree –

In-order Traversal

Pre-order Traversal

Post-order Traversal

Generally we traverse a tree to search or locate given item or key in the tree or to print all the values it contains.

. Preorder Traversal-

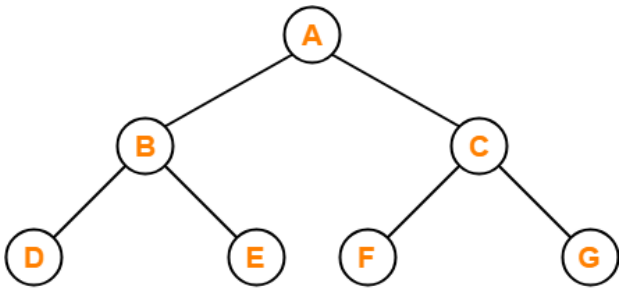
Algorithm-

1. Visit the root
2. Traverse the left sub tree i.e. call Preorder (left sub tree)
3. Traverse the right sub tree i.e. call Preorder (right sub tree)

Root → Left → Right

Example-

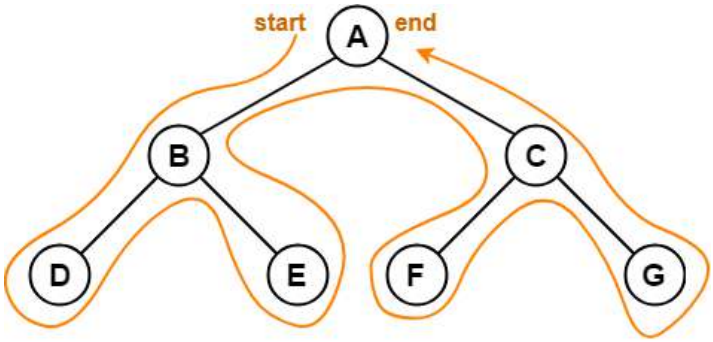
Consider the following example-



Preorder Traversal : A , B , D , E , C , F , G

Preorder Traversal Shortcut

Traverse the entire tree starting from the root node keeping yourself to the left.



Preorder Traversal : A , B , D , E , C , F , G

Applications-

- Preorder traversal is used to get prefix expression of an expression tree.
- Preorder traversal is used to create a copy of the tree.

2. Inorder Traversal-

Algorithm-

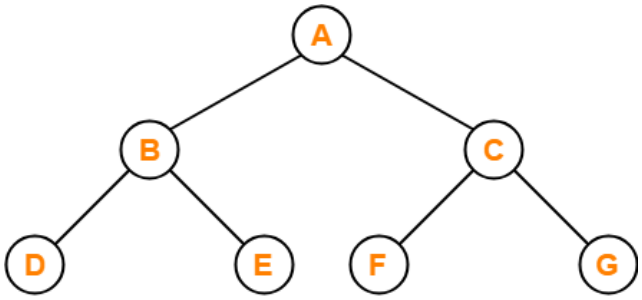
1. Traverse the left sub tree i.e. call Inorder (left sub tree)

2. Visit the root
3. Traverse the right sub tree i.e. call Inorder (right sub tree)

Left → Root → Right

Example-

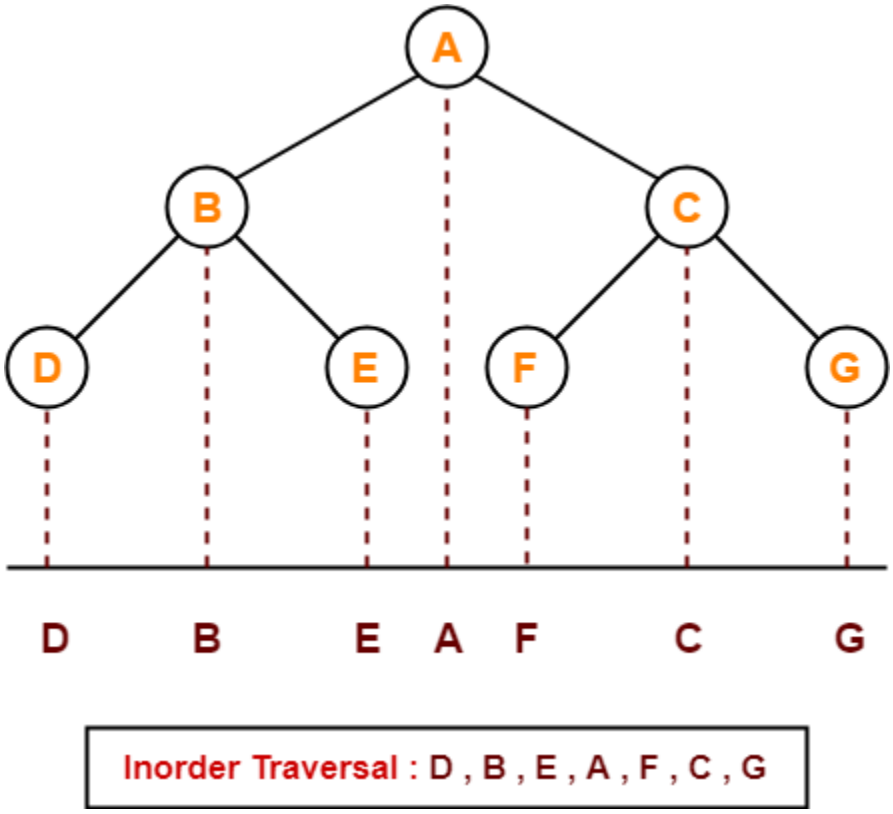
Consider the following example-



Inorder Traversal : D , B , E , A , F , C , G

Inorder Traversal Shortcut

Keep a plane mirror horizontally at the bottom of the tree and take the projection of all the nodes.



Application-

- Inorder traversal is used to get infix expression of an expression tree.

3. Postorder Traversal-

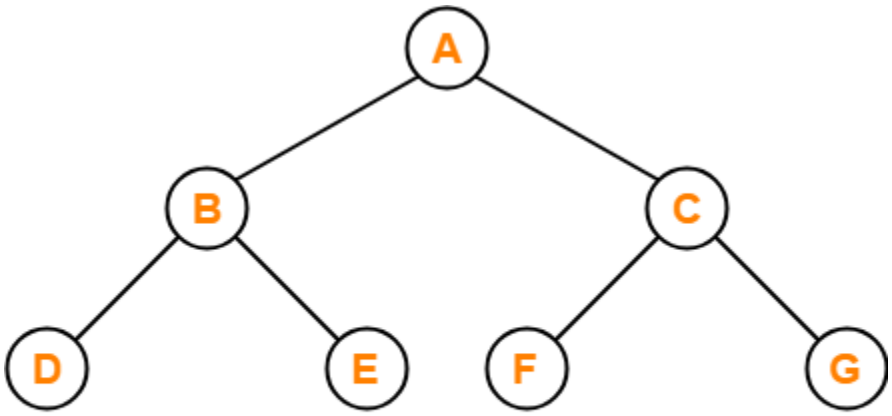
Algorithm-

1. Traverse the left sub tree i.e. call Postorder (left sub tree)
2. Traverse the right sub tree i.e. call Postorder (right sub tree)
3. Visit the root

Left → Right → Root

Example-

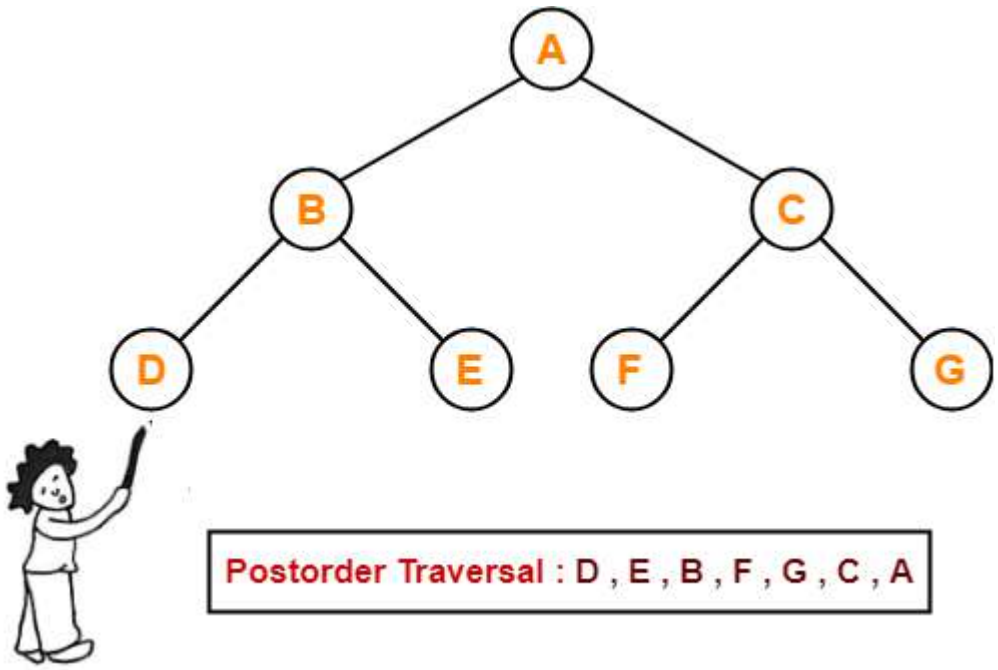
Consider the following example-



Postorder Traversal : D , E , B , F , G , C , A

Postorder Traversal Shortcut

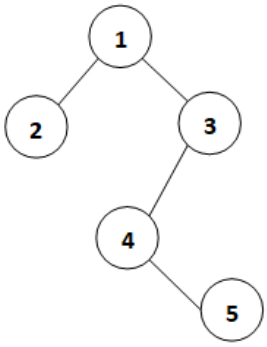
Pluck all the leftmost leaf nodes one by one.



Applications-

- Postorder traversal is used to get postfix expression of an expression tree.
- Postorder traversal is used to delete the tree.
- This is because it deletes the children first and then it deletes the parent.

Give traversal order of following tree into Inorder, Preorder and Postorder.

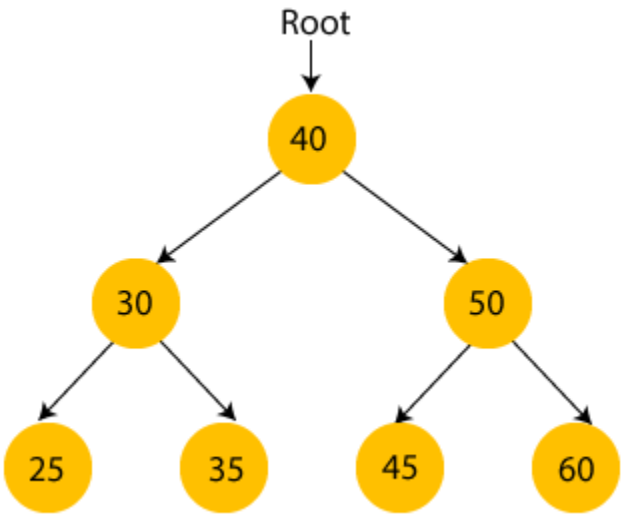


Inorder: 2 1 4 5 3
Preorder: 1 2 3 4 5
Post order: 2 5 4 3 1

hat is a Binary Search tree?

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

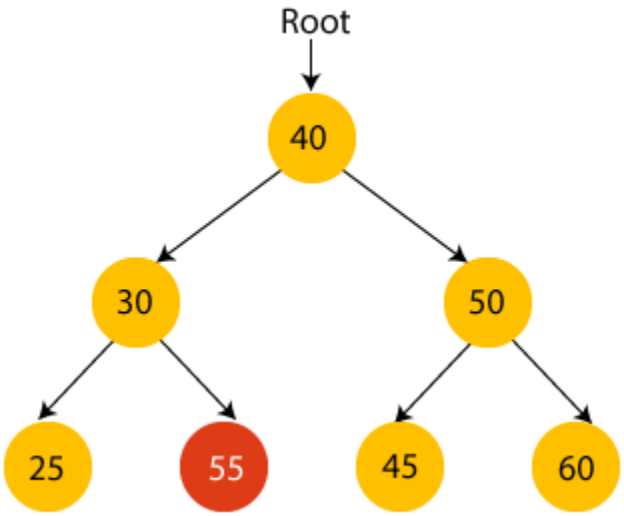
Let's understand the concept of Binary search tree with an example.



In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.



In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree.

Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Example of creating a binary search tree

Now, let's see the creation of binary search tree using an example.

Suppose the data elements are **- 45, 15, 79, 90, 10, 55, 12, 20, 50**

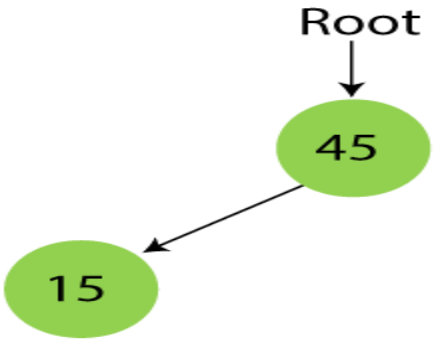
- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -

Step 1 - Insert 45.

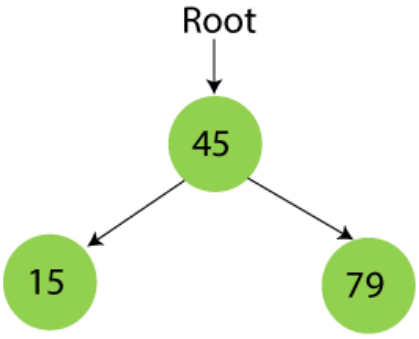
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



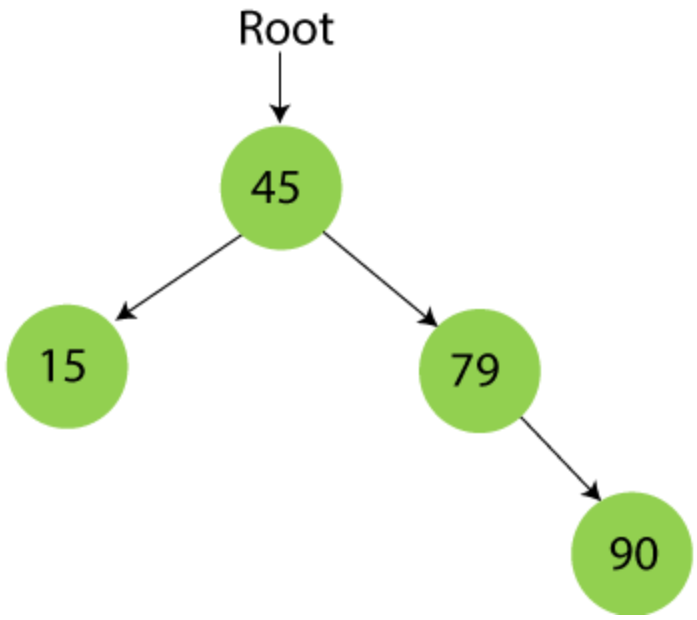
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



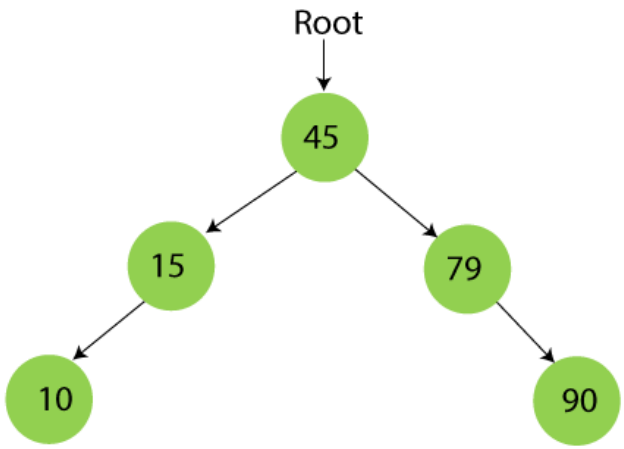
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



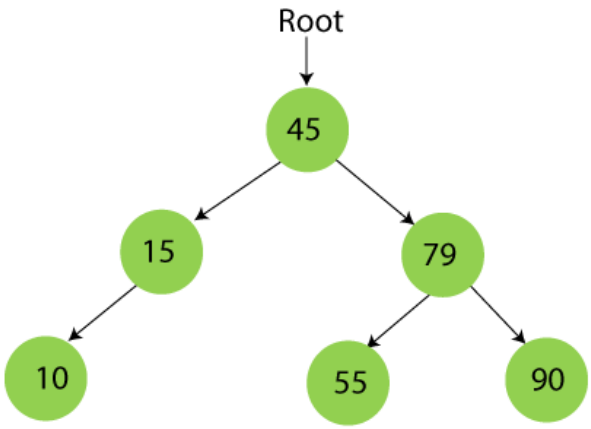
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



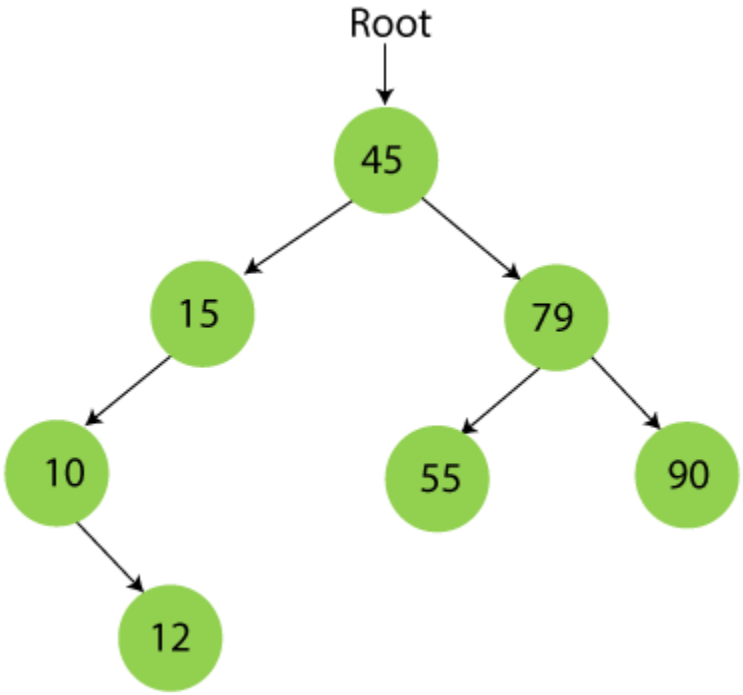
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



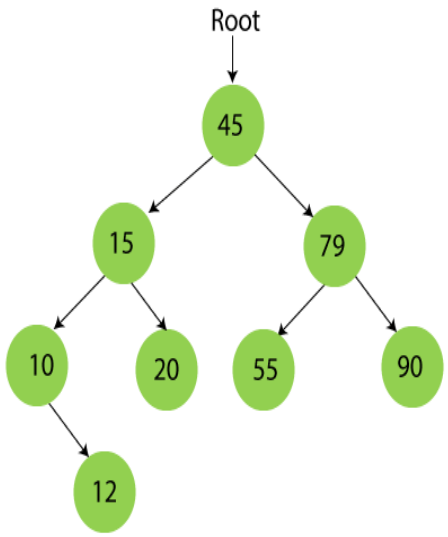
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



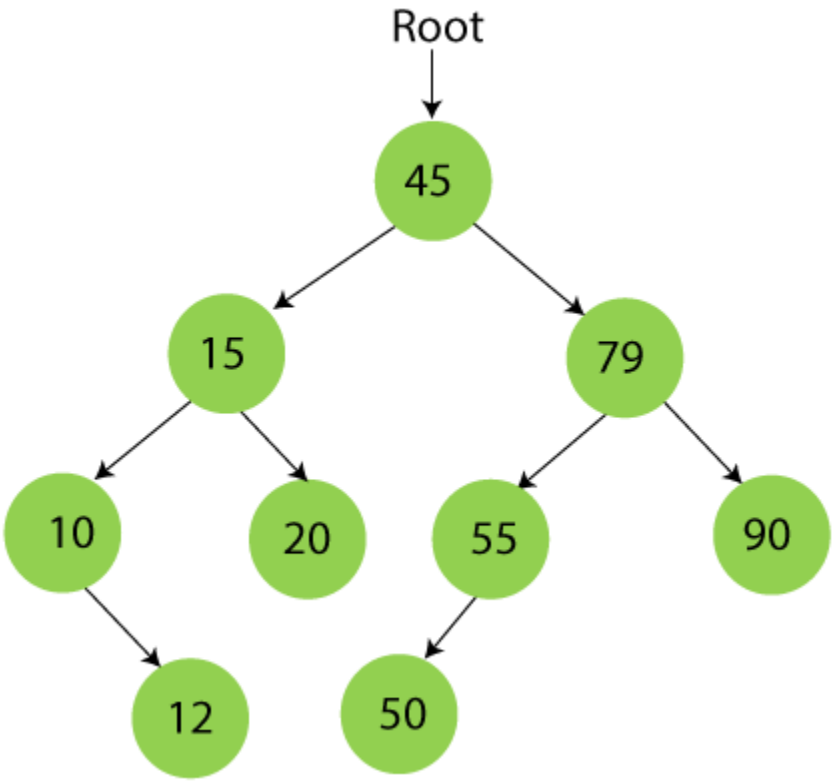
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Now, the creation of binary search tree is completed. After that, let's move towards the operations that can be performed on Binary search tree.

We can perform insert, delete and search operations on the binary search tree.

Let's understand how a search is performed on a binary search tree.

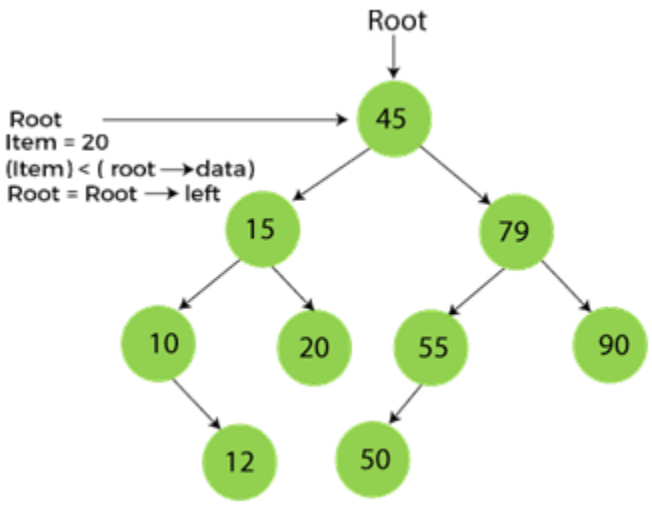
SEARCHING IN BINARY SEARCH TREE

Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -

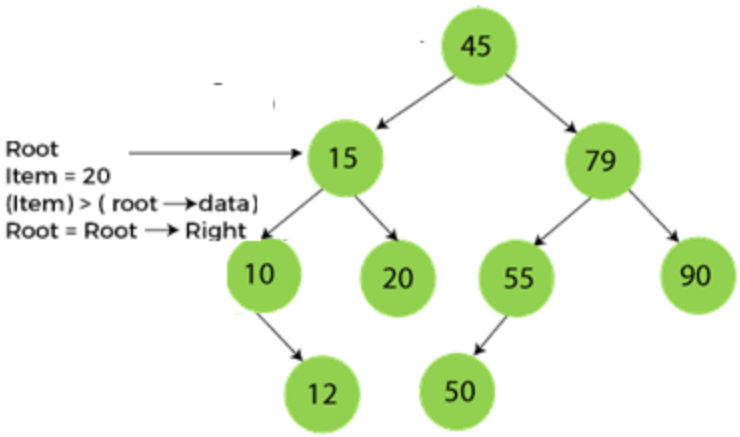
1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4. If it is larger than the root element, then move to the right subtree.
5. Repeat the above procedure recursively until the match is found.
6. If the element is not found or not present in the tree, then return NULL.

Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.

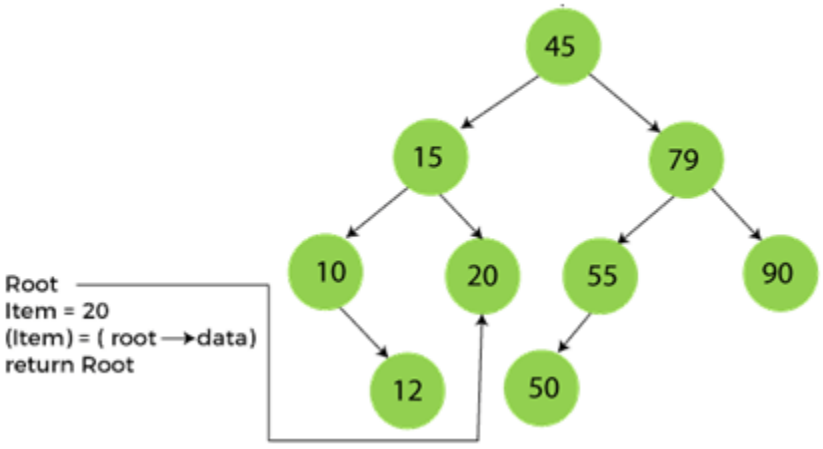
Step1:



Step2:



Step3:



Now, let's see the algorithm to search an element in the Binary search tree.

Algorithm to search an element in Binary search tree

1. Search (root, item)
2. Step 1 - if (item = root → data) or (root = NULL)
3. return root
4. else if (item **< root** → data)
5. return Search(root → left, item)
6. else
7. return Search(root → right, item)
8. END if
9. Step 2 - END

Now let's understand how the deletion is performed on a binary search tree. We will also see an example to delete an element from the given tree.

DELETION IN BINARY SEARCH TREE

In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

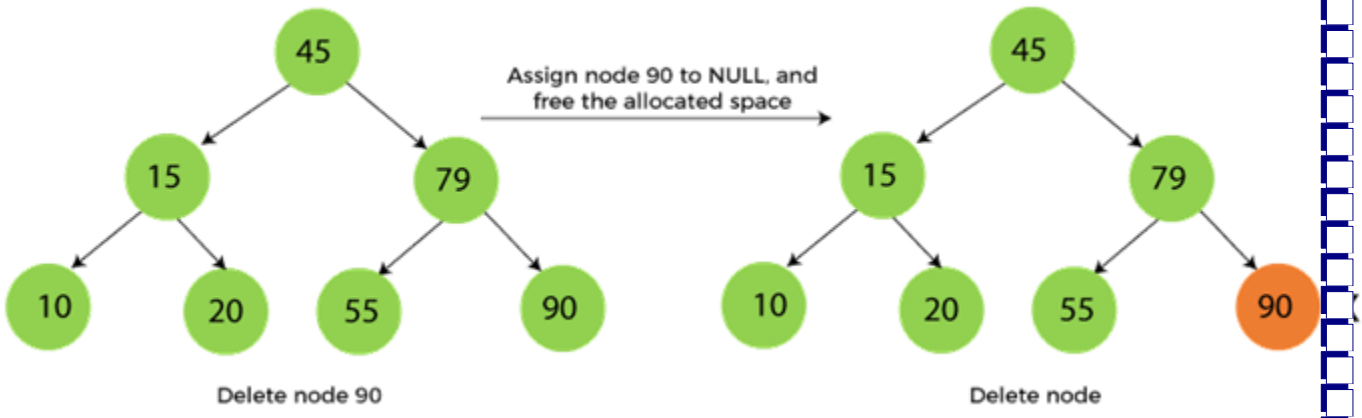
- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

We will understand the situations listed above in detail.

When the node to be deleted is the leaf node

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.

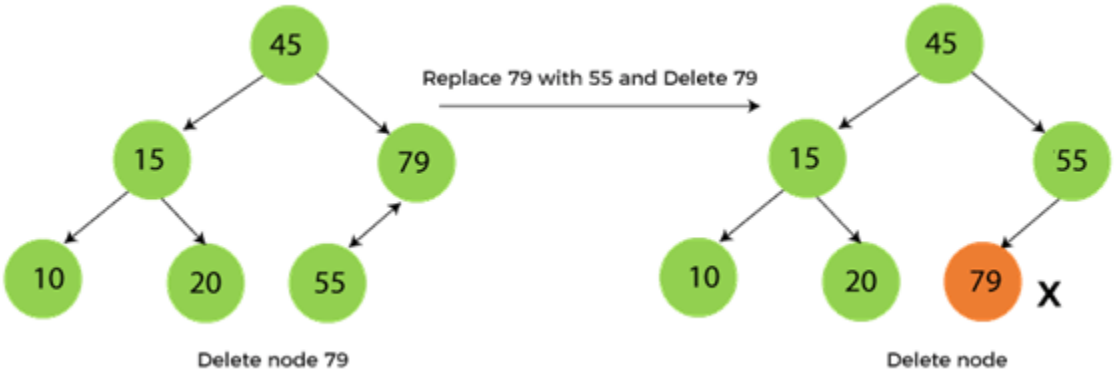


When the node to be deleted has only one child

In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.

We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.

So, the replaced node 79 will now be a leaf node that can be easily deleted.



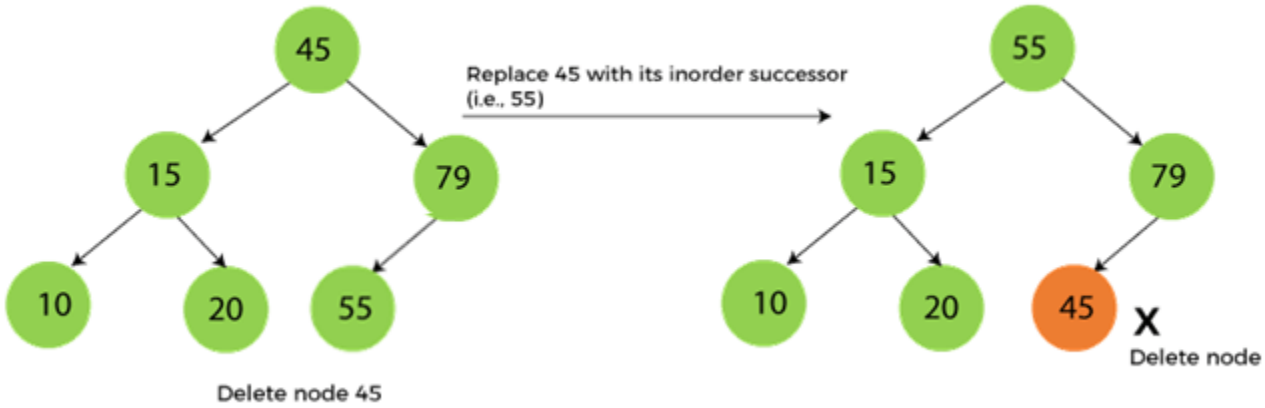
When the node to be deleted has two children

This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -

- First, find the inorder successor of the node to be deleted.
- After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
- And at last, replace the node with NULL and free up the allocated space.

The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.

We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.

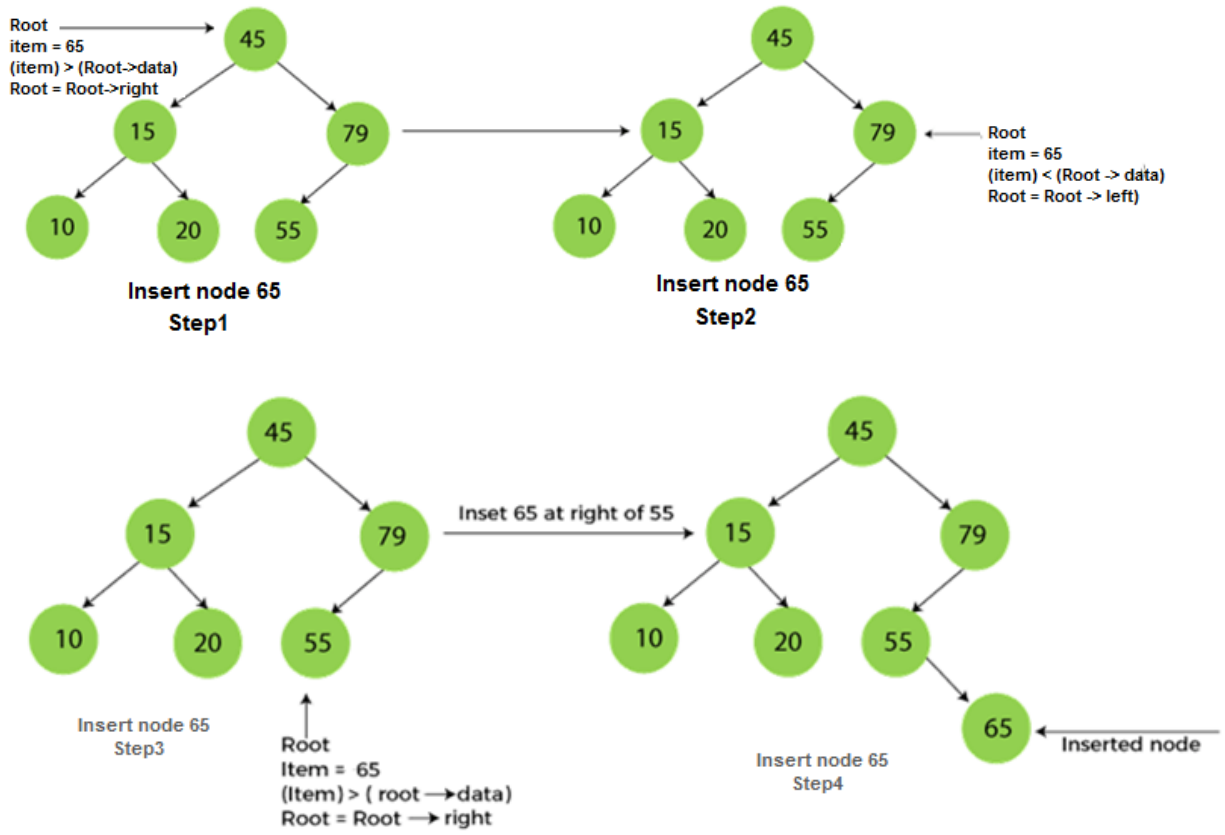


Now let's understand how insertion is performed on a binary search tree.

INSERTION IN BINARY SEARCH TREE

A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.

Now, let's see the process of inserting a node into BST using an example.

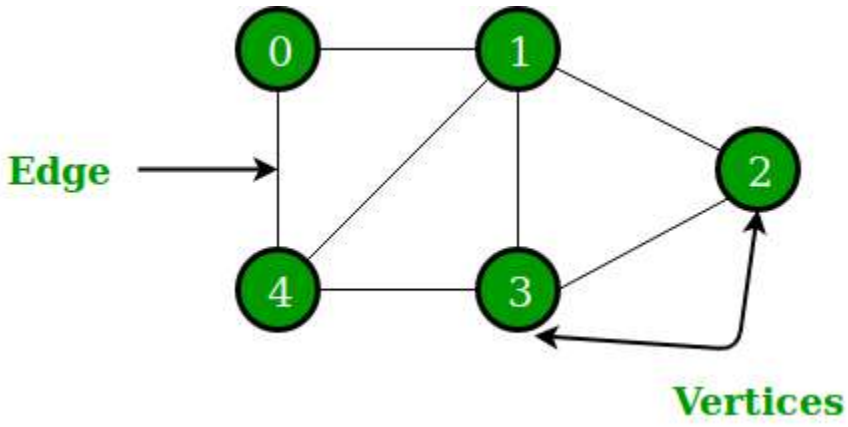


UNIT-VII

GRAPH

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. More formally a Graph can be defined as,

A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.



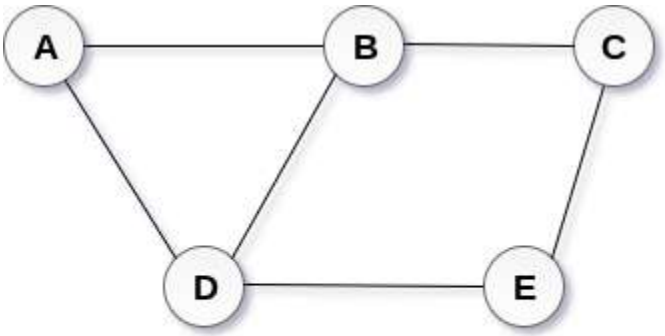
In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

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

Definition

A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



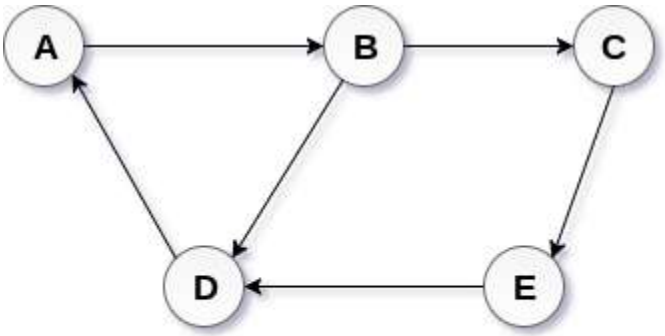
Undirected Graph

DIRECTED AND UNDIRECTED GRAPH

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.

A directed graph is shown in the following figure.



Directed Graph

GRAPH TERMINOLOGY

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.

Closed Path

A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0=V_N$.

Simple Path

If all the nodes of the graph are distinct with an exception $V_0=V_N$, then such path P is called as closed simple path.

Cycle

A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Connected Graph

A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

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 associated with the similar end points can be called as Loop.

Adjacent Nodes

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

Graph Representation

By Graph representation, we simply mean the technique which is to be used in order to store some graph into the computer's memory.

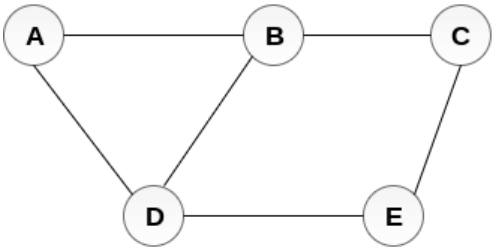
There are two ways to store Graph into the computer's memory. In this part of this tutorial, we discuss each one of them in detail.

1. Sequential 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**.

An entry M_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if there exists an edge between V_i and V_j.

An undirected graph and its adjacency matrix representation is shown in the following figure.



Undirected Graph

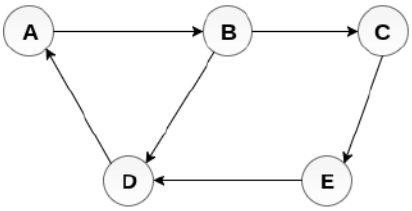
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

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 which is also shown in the figure.

There exists different adjacency matrices for the directed and undirected graph. In directed graph, an entry A_{ij} will be 1 only when there is an edge directed from V_i to V_j.

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



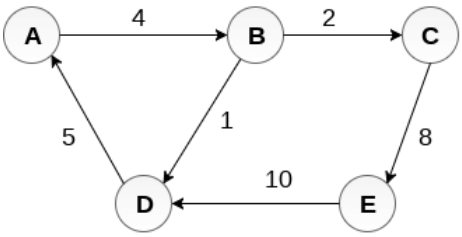
Directed Graph

	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

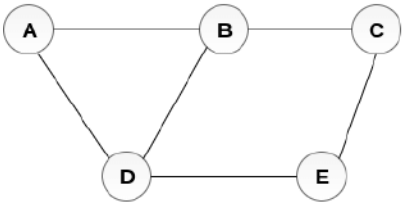
	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

Adjacency Matrix

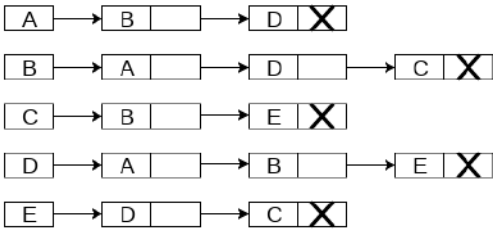
LINKED REPRESENTATION

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.



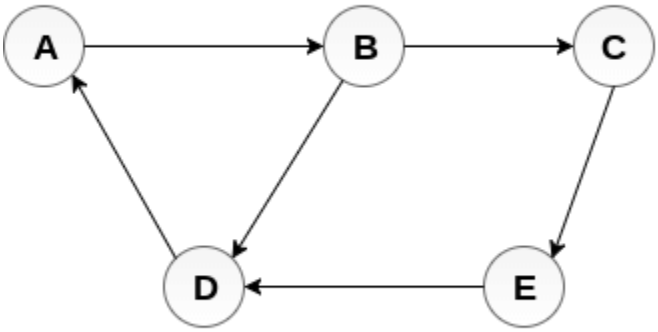
Undirected Graph



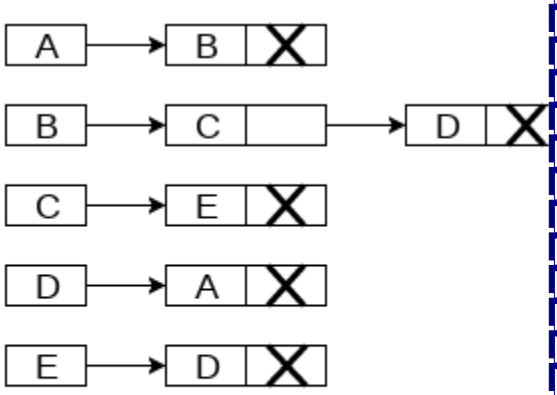
Adjacency List

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.



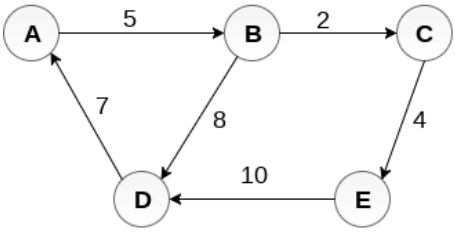
Directed Graph



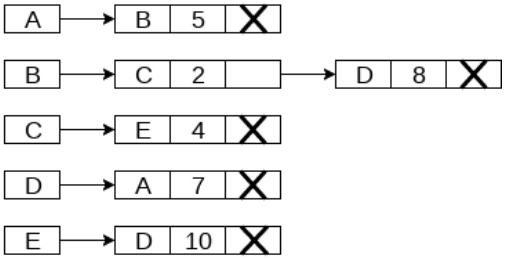
Adjacency List

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

ADJACENCY MATRIX

Adjacency matrix definition

In graph theory, an adjacency matrix is a dense way of describing the finite graph structure. It is the 2D matrix that is used to map the association between the graph nodes.

If a graph has **n** number of vertices, then the adjacency matrix of that graph is **n x n**, and each entry of the matrix represents the number of edges from one vertex to another.

An adjacency matrix is also called as **connection matrix**. Sometimes it is also called a **Vertex matrix**.

Adjacency Matrix Representation

If an Undirected Graph G consists of n vertices then the adjacency matrix of a graph is n x n matrix A = [aij] and defined by -

a_{ij} = 1 {if there is a path exists from V_i to V_j}

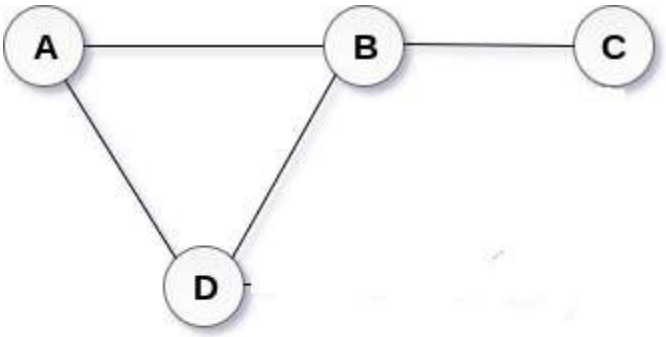
a_{ij} = 0 {Otherwise}

Let's see some of the important points with respect to the adjacency matrix.

- If there exists an edge between vertex V_i and V_j, where i is a row, and j is a column, then the value of a_{ij} = 1.
- If there is no edge between vertex V_i and V_j, then the value of a_{ij} = 0.
- If there are no self loops in the simple graph, then the vertex matrix (or adjacency matrix) should have 0s in the diagonal.
- An adjacency matrix is symmetric for an undirected graph. It specifies that the value in the i^{th} row and j^{th} column is equal to the value in j^{th} row i^{th}

- If the adjacency matrix is multiplied by itself, and if there is a non-zero value is present at the i^{th} row and j^{th} column, then there is the route from V_i to V_j with a length equivalent to 2. The non-zero value in the adjacency matrix represents that the number of distinct paths is present.

Note: In an adjacency matrix, 0 represents that there is no association is exists between two nodes, whereas 1 represents that there is an association is exists between two nodes.



Undirected Graph

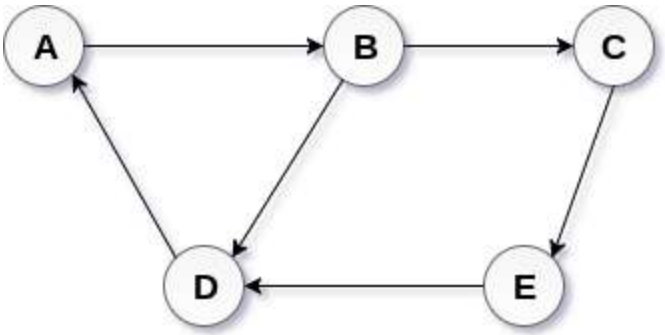
In the graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix will be 0. The adjacency matrix of the above graph will be -

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

ADJACENCY MATRIX FOR A DIRECTED GRAPH

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called the initial node, while node B is called the terminal node.

Let us consider the below directed graph and try to construct the adjacency matrix of it.



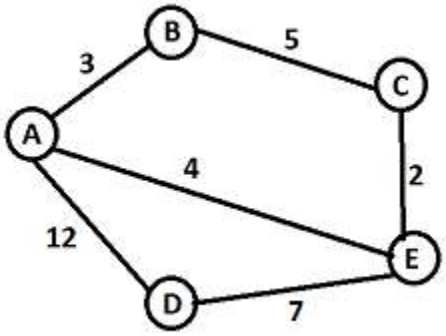
Directed Graph

In the above graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix will be 0. The adjacency matrix of the above graph will be -

	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

NOTE: A graph is said to be the weighted graph if each edge is assigned a positive number, which is called the weight of the edge.

Question 1 - What will be the adjacency matrix for the below undirected weighted graph?

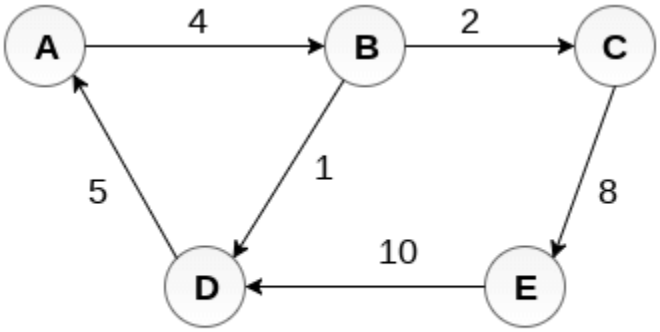


Solution - In the given question, there is no self-loop, so it is clear that the diagonal entries of the adjacent matrix for the above graph will be 0. The above graph is a weighted undirected graph. The weights on the graph edges will be represented as the entries of the adjacency matrix.

The adjacency matrix of the above graph will be -

	A	B	C	D	E
A	0	3	0	12	4
B	3	0	5	0	0
C	0	5	0	0	2
D	12	0	0	0	7
E	4	0	2	7	0

Question 2 - What will be the adjacency matrix for the below directed weighted graph?



Weighted Directed Graph

Solution - In the given question, there is no self-loop, so it is clear that the diagonal entries of the adjacent matrix for the above graph will be 0. The above graph is a weighted directed graph. The weights on the graph edges will be represented as the entries of the adjacency matrix.

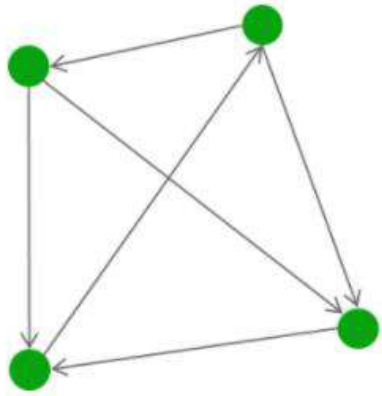
The adjacency matrix of the above graph will be -

	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

Hope this article is beneficial to you in order to understand about adjacency matrix. Here, we have discussed the adjacency matrix along with its creation and properties. We have also discussed the formation of adjacency matrix on directed or undirected graphs, whether they are weighted or not.

WHAT IS PATH MATRIX?

A path matrix is a matrix representing a graph where each value in m'th row and n'th column project whether there is a path from m to n. The path may be direct or indirect. It may have a single edge or multiple edges.



$$P = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Path Matrix for a Graph

Consider the graph:

Adjacency Matrix for this graph is:

$$A = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Derive A^2 : Derive A^2 :

$$A^2 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Derive A^3 :

$$A^3 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 2 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

Derive A^4 :

$$A^4 = \begin{pmatrix} 1 & 0 & 2 & 2 \\ 2 & 1 & 2 & 0 \\ 0 & 0 & 1 & 2 \\ 0 & 2 & 0 & 1 \end{pmatrix}$$

Sum up all four matrix to derive B_4 :

$$B_4 = \begin{pmatrix} 2 & 2 & 4 & 4 \\ 3 & 3 & 4 & 3 \\ 1 & 1 & 2 & 3 \\ 1 & 3 & 2 & 3 \end{pmatrix}$$

Derive Path Matrix P from B_4 by replacing any none zero value with 1:

$$\mathrm{P} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Difference Between Adjacency Matrix & Path Matrix

Key difference between Adjacency Matrix and Path Matrix is that an adjacency matrix is about direct edge where a path matrix is about whether can be traveled or not. Path matrix include both direct and indirect edges.

UNIT-VIII

SORTING SEARCHING & MERGING

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

BUBBLE SORTING:

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2) where n is the number of items.

Algorithm:

BUBBLE
(DATA,N)

Here DATA is an array with N element. This algorithm sorts the element in DATA.Step 1: [Loop]

Repeat step 2 and step 3 for K=1 to N-1

Step 2: [Initialize pass pointer PTR]

Set[PTR]=1

Step 3: [Execute pass]

Repeat while PTR <=N-K

a. If DATA [PTR] > DATA [PTR+1]

Then interchange DATA [PTR] & DATA [PTR+1][End of if structure]

b. Set PTR =PTR+1

[End of

Step 1 Loop] Step

4: Exit

COMPLEXITY OF THE BUBBLE SORT ALGORITHM

The time for a sorting algorithm is measured in terms of the number of comparisons. The number f(n) of comparisons in the bubble sort is easily computed.

There are n-1 comparisons during the first pass, which places the largest element in the last position;

there are n-2 comparisons in the second step, which places the second largest element in the next to last position and so on.

F(n)=(n-1)+(n-2)+….+2+1=n(n-1)/2=n^2/2+0(n)=0(n^2)

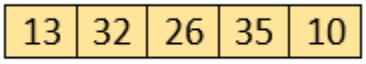
The time required to execute the bubble sort algorithm is proportional to n^2,Where n is the number of input items.

Working of Bubble sort Algorithm

Now, let's see the working of Bubble sort Algorithm.

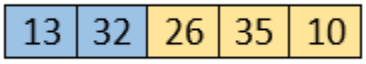
To understand the working of bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the complexity of bubble sort is **O(n^2).**

Let the elements of array are -

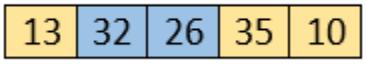


First Pass

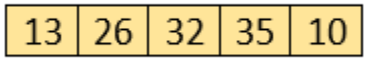
Sorting will start from the initial two elements. Let compare them to check which is greater.



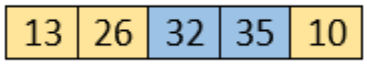
Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.



Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -

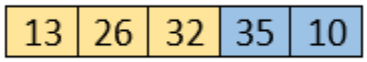


Now, compare 32 and 35.

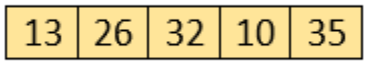


Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.

Now, the comparison will be in between 35 and 10.



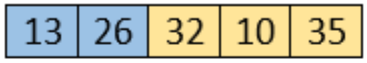
Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -

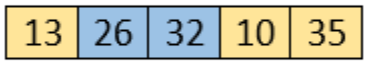


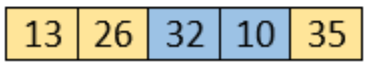
Now, move to the second iteration.

Second Pass

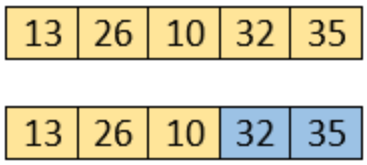
The same process will be followed for second iteration.







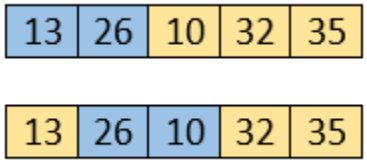
Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -



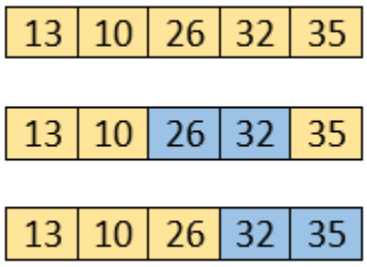
Now, move to the third iteration.

Third Pass

The same process will be followed for third iteration.



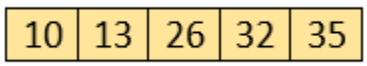
Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -



Now, move to the fourth iteration.

Fourth pass

Similarly, after the fourth iteration, the array will be -



Hence, there is no swapping required, so the array is completely sorted.

Bubble sort complexity

Now, let's see the time complexity of bubble sort in the best case, average case, and worst case. We will also see the space complexity of bubble sort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

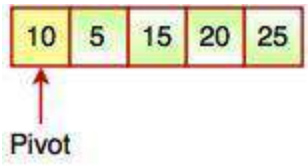
- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is **O(n).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is **$O(n^2)$.**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is **$O(n^2)$.**

QUICK SORT

- Quick sort is also known as **Partition-exchange sort** based on the rule of **Divide and Conquer.**
- It is a highly efficient sorting algorithm.
- Quick sort is the quickest comparison-based sorting algorithm.
- It is very fast and requires less additional space
- Quick sort picks an element as pivot and partitions the array around the picked pivot.

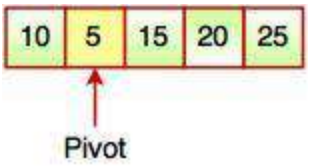
There are different versions of quick sort which choose the pivot in different ways:

1. First element as pivot

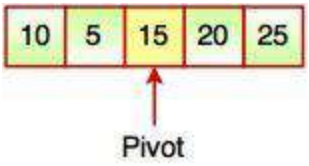


2. Last element as pivot

3. Random element as pivot



4. Median as pivot



Example of Quick Sort:

1. 44 33 11 55 77 90 40 60 99 22 88

Let **44** be the **Pivot** element and scanning done from right to left

Comparing **44** to the right-side elements, and if right-side elements are **smaller** than **44**, then swap it. As **22** is smaller than **44** so swap them.

22 33 11 55 77 90 40 60 99 **44** 88

Now comparing **44** to the left side element and the element must be **greater** than 44 then swap them. As **55** are greater than **44** so swap them.

22 33 11 **44** 77 90 40 60 99 **55** 88

Recursively, repeating steps 1 & steps 2 until we get two lists one left from pivot element **44** & one right from pivot element.

22 33 11 **40** 77 90 **44** 60 99 55 88

Swap with 77:

22 33 11 40 **44** 90 **77** 60 99 55 88

Now, the element on the right side and left side are greater than and smaller than **44** respectively.

Now we get two sorted lists:

22 33 11 40 **44** 90 77 66 99 55 88

Sublist1 **Sublist2**

And these sublists are sorted under the same process as above done.

These two sorted sublists side by side.

22 33 11 40 **44** **90** 77 60 99 55 88

11 33 **22** 40 **44** 88 77 60 99 55 **90**

11 **22** 33 40 **44** 88 77 60 **90** 55 **99**

First sorted list

88 77 60 **55** **90** 99

Sublist3 **Sublist4**

55 77 60 **88** 90 99

Sorted

55 **77** **60**

55 60 77

Sorted

Merging Sublists:

11	22	33	40	44	55	60	77	88	90	99

SORTED LISTS

lgorithm:

```
1. QUICKSORT (array A, start, end)
2. {
3.  1 if (start < end)
4.  2 {
5.  3 p = partition(A, start, end)
6.  4 QUICKSORT (A, start, p - 1)
7.  5 QUICKSORT (A, p + 1, end)
8.  6 }
9.  }
```

Partition Algorithm:

The partition algorithm rearranges the sub-arrays in a place.

```
1. PARTITION (array A, start, end)
2. {
3.  1 pivot = A[end]
4.  2 i = start-1
5.  3 for j = start to end -1 {
6.  4 do if (A[j] < pivot) {
7.  5 then i = i + 1
8.  6 swap A[i] with A[j]
9.  7  }}
10. 8 swap A[i+1] with A[end]
11. 9 return i+1
12.}
```

Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	$O(n^2)$

In **Merge Sort**, the given unsorted array with n elements, is divided into n subarrays, each having **one** element, because a single element is always sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.

- e merge sort algorithm is based on the classical divide-and-conquer paradigm. It operates as follows:
  - **DIVIDE:** Partition the n-element sequence to be sorted into two subsequences of n/2 elements each.
  - **CONQUER:** Sort the two subsequences recursively using the merge sort.
  - **COMBINE:** Merge the two sorted subsequences of size n/2 each to produce the sorted sequence consisting of n elements.
- Note that recursion "**bottoms out**" when the sequence to be sorted is of unit length.
- Since every sequence of length 1 is in sorted order, no further recursive call is necessary.
- The key operation of the merge sort algorithm is the merging of the two sorted sub sequences in the "combine step".
- To perform the merging, we use an auxiliary procedure Merge (A,p,q,r), where A is an array and p,q and r are indices numbering elements of the array such that procedure assumes that the sub arrays A[p..q] and A[q+1...r] are in sorted order.
- It merges them to form a single sorted sub array that replaces the current sub array A[p..r]. Thus finally, we obtain the sorted array A[1..n], which is the solution.

Algorithm

```
rocedure mergesort( var a as array )
  if ( n == 1 ) return a

  var l1 as array = a[0] ... a[n/2]
  var l2 as array = a[n/2+1] ... a[n]

  l1 = mergesort( l1 )
  l2 = mergesort( l2 )

  return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )

  var c as array
  while ( a and b have elements )
    if ( a[0] > b[0] )
      add b[0] to the end of c
      remove b[0] from b
    else
      add a[0] to the end of c
      remove a[0] from a
    end if
  end while

  while ( a has elements )
    add a[0] to the end of c
    remove a[0] from a
  end while

  while ( b has elements )
    add b[0] to the end of c
    remove b[0] from b
  end while

  return c

end procedure
```

To know about merge sort implementation in C programming language, please click here.

How Merge Sort Works?

As we have already discussed that merge sort utilizes divide-and-conquer rule to break the problem into sub-problems, the problem in this case being, **sorting a given array**.

In merge sort, we break the given array midway, for example if the original array had 6 elements, then merge sort will break it down into two subarrays with 3 elements each.

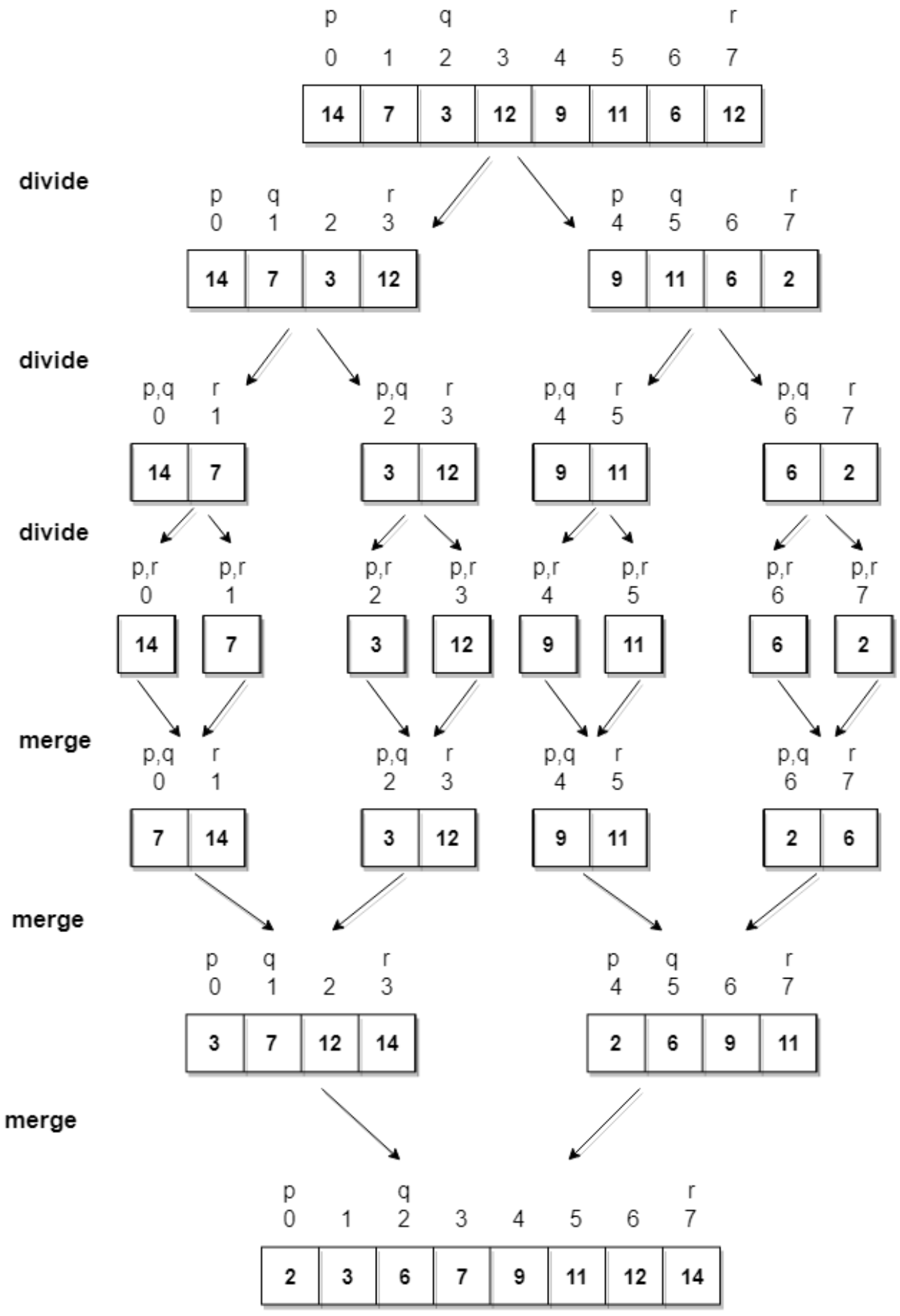
But breaking the orignal array into 2 smaller subarrays is not helping us in sorting the array.

So we will break these subarrays into even smaller subarrays, until we have multiple subarrays with **single element** in them. Now, the idea here is that an array with a single element is already sorted, so once we break the original array into subarrays which has only a single element, we have successfully broken down our problem into base problems.

And then we have to merge all these sorted subarrays, step by step to form one single sorted array.

Let's consider an array with values {14, 7, 3, 12, 9, 11, 6, 12}

Below, we have a pictorial representation of how merge sort will sort the given array.


p q r
0 1 2 3 4 5 6 7
14 7 3 12 9 11 6 12
divide
p q r
0 1 2 3
14 7 3 12
p q r
4 5 6 7
9 11 6 2
divide
p,q r
0 1
14 7
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
6 2
divide
p,r
0
p,r
1
p,r
2
p,r
3
p,r
4
p,r
5
p,r
6
p,r
7
14
7
3
12
9
11
6
2
merge
p,q r
0 1
7 14
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
2 6
merge
p q r
0 1 2 3
3 7 12 14
p q r
4 5 6 7
2 6 9 11
merge
p q r
0 1 2 3 4 5 6 7
2 3 6 7 9 11 12 14

I COMPLEXITY OF THE MERGING ALGORITHM

The input consists of the total number n=r+s of elements in A and B. Each comparison assigns an elements to the array C, which eventually has n elements. Accordingly, the number f(n) of comparisons cannot exceed n:

F(n) <= n = 0(n)

In other words, the merging algorithm can be run in

linear time.WORST CASE : n log= 0[n log n]

AVERAGE CASE : n log n=0[n log n]

SEARCHING:

Searching refers to finding the location i.e LOC of ITEM in an array. The search is said to be successful if ITEM appears the array & unsuccessful otherwise we have two types of searching techniques.

1. linear Search
2. Binary Search

LINEAR SEARCH:

Suppose DATA is a linear array with n elements. No other information about DATA, the most intuitive way to search for a given ITEM in DATA is to compare ITEM with each element of DATA one by one. First we have to test whether DATA [1]=ITEM, nad then we test whether DATA[2] =ITEM , and so on. This method which traverses DATA sequentially to locate ITEM, is called *linear search*or *sequential search*.

ALGORITHM

LINEAR (DATA, N, ITEM, LOC)

Step 1: [Insert ITEM at the

end of data] Set
DATA [N+1] =
ITEM

Step 2: [Initialize counter]
Set LOC=1

Step 3: [Search for ITEM]
Repeat while DATA
[LOC]!= ITEM Step 4:
[Successful]
If LOC=N+1
Then
Set LOC = 0

Step 5: Exit

COMPLEXITY OF THE LINEAR SEARCH ALGORITHM

The complexity of search algorithm is measured by the number f(n) of comparisons required to find ITEM in DATA where DATA contains n elements. Two important cases to consider are the average case and the worst case.

The worst case occurs when one must search through the entire array DATA. In this case, the algorithm requires

F(n)= n+1

Thus, in the worst case, running time is proportional to n.

The running time of the average case uses the probabilistic notion of expectation. The probability that ITEM appears in DATA[K], and q is the probability that ITEM

does not appear in DATA . Since the algorithm uses k comparison when ITEM appears in DATA[K], the average number of comparison is given by

F(n) = 1 . p1 + 2 . p2 +…..+ n . pn + (n+1).q

BINARY SEARCH

Suppose DATA is an array which is sorted in increasing numerical order or equivalently, alphabetically. Then there is an extremely efficient searching algorithm, called *binary search.*

Algorithm

Binary search (DATA, LB, UB,
ITEM, LOC) Step 1: [Initialize
the segment variables]

Set BEG := LB, END := UB and MID := INT ((BEG + END)/2)

Step 2: [Loop]

Repeat Step 3 and Step 4 while BEG <= END and DATA [MID] != ITEMStep 3: [Compare]

If ITEM < DATA [MID]
then set END := MID - 1

Else

Set BEG = MID + 1

Step 4: [Calculate MID]

Set MID := INT ((BEG + END)/2)

Step 5: [Successful search]

If DATA [MID] = ITEM
then set
LOC := MID
Else set
LOC :=
NULL

Step 6: Exit

COMPLEXITY OF THE BINARY SEARCH ALGORITHM

The complexity is measured by the number f(n) of comparison to locate ITEM in DATA where DATA contains n elements. Observe that each comparison reduces the sample size in half. Hence we require at most f(n)

comparison to locate ITEM where2f(n) > n

Or equivalently F(n) = [log2 n] + 1

The running time for the worst case is approximately equal to log2 n and the average case is approximately equal to the running time for the worst case.

UNIY-IX

FILES AND THEIR ORGANIZATION

INTRODUCTION

File is a collection of records related to each other. The file size is limited by the size of memory and storagemedium.

FILE ORGANIZATION

File organization ensures that records are available for processing. It is used to determine an efficient file organization for each base relation.

For example, if we want to retrieve employee records in alphabetical order of name. Sorting the file by employee name is a good file organization. However, if we want to retrieve all employees whose marks are in a certain range, a file is ordered by employee name would not be a good file organization.

Types of File Organization

There are three types of organizing the file:

1. Sequential access file organization

2. Direct access file organization

3. Indexed sequential access file organization

1. SEQUENTIAL ACCESS FILE ORGANIZATION

• Storing and sorting in contiguous block within files on tape or disk is called as sequential access file organization.

• In sequential access file organization, all records are stored in a sequential order. The records are arranged in the ascending or descending order of a key field.

• Sequential file search starts from the beginning of the file and the records can be added at the end of the file.

• In sequential file, it is not possible to add a record in the middle of the file without rewriting the file.

Advantages of sequential file

• It is simple to program and easy to design.

• Sequential file is best use if storage space.

Disadvantages of sequential file

• Sequential file is time consuming process.

• It has high data redundancy.

• Random searching is not possible.

2. DIRECT ACCESS FILE ORGANIZATION

• Direct access file is also known as random access or relative file organization.

• In direct access file, all records are stored in direct access storage device (DASD), such as hard disk. The records are randomly placed throughout the file.

• The records does not need to be in sequence because they are updated directly and rewritten back in the same location.

• This file organization is useful for immediate access to large amount of information. It is used in accessing large databases.

• It is also called as hashing.

Advantages of direct access file organization

• Direct access file helps in online transaction processing system (OLTP) like online railway reservation system.

• In direct access file, sorting of the records are not required.

• It accesses the desired records immediately.

• It updates several files quickly.

• It has better control over record allocation.

Disadvantages of direct access file organization

• Direct access file does not provide back up facility.

• It is expensive.

• It has less storage space as compared to sequential file.

3. INDEXED SEQUENTIAL ACCESS FILE ORGANIZATION

• Indexed sequential access file combines both sequential file and direct access file organization.

• In indexed sequential access file, records are stored randomly on a direct access device such as magnetic disk by a primary key.

• This file have multiple keys. These keys can be alphanumeric in which the records are ordered is called primary key.

• The data can be access either sequentially or randomly using the index. The index is stored in a file and read into memory when the file is opened.

Advantages of Indexed sequential access file organization

• In indexed sequential access file, sequential file and random file access is possible.

• It accesses the records very fast if the index table is properly organized.

• The records can be inserted in the middle of the file.

• It provides quick access for sequential and direct processing.

• It reduces the degree of the sequential search.

Disadvantages of Indexed sequential access file organization

• Indexed sequential access file requires unique keys and periodic reorganization.

• Indexed sequential access file takes longer time to search the index for the data access or retrieval.

• It requires more storage space.

• It is expensive because it requires special software.

• It is less efficient in the use of storage space as compared to other file organizations.

HASHING

There are many possibilities for representing the dictionary and one of the best methods for representing is **hashing**. Hashing is a type of a solution which can be used in almost all situations. Hashing is a technique which uses less key comparisons and searches the element in **O(n)** time in the worst case and in an average case it will be done in **O(1)** time. This method generally used the hash functions to map the keys into a table, which is called a **hash table**.

1) Hash table

Hash table is a type of data structure which is used for storing and accessing data very quickly. Insertion of data in a table is based on a key value. Hence every entry in the hash table is defined with some key. By using this key data can be searched in the hash table by few key comparisons and then searching time is dependent upon the size of the hash table.

2) Hash function

Hash function is a function which is applied on a key by which it produces an integer, which can be used as an address of hash table. Hence one can use the same hash function for accessing the data from the hash table. In this the integer returned by the hash function is called hash key.

Types of hash function

There are various types of hash function which are used to place the data in a hash table,

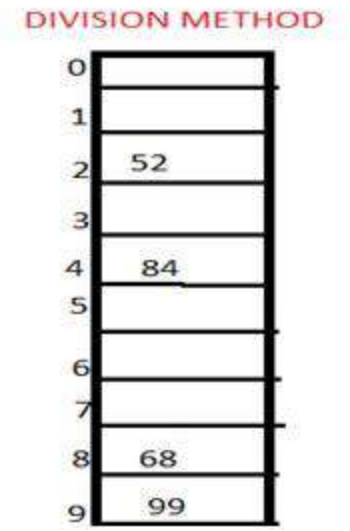
1. Division method

In this the hash function is dependent upon the remainder of a division. For example:-if the record 52,68,99,84 is to be placed in a hash table and let us take the table size is 10.

Then:

h(key)=record% table size.

2=52%10
8=68%10
9=99%10
4=84%10



2. Mid square method

In this method firstly key is squared and then mid part of the result is taken as the index. For example: consider that if we want to place a record of 3101 and the size of table is 1000. So 3101*3101=9616201 i.e. **h (3101) = 162 (middle 3 digit)**

3. Digit folding method

In this method the key is divided into separate parts and by using some simple operations these parts are combined to produce a hash key. For example: consider a record of 12465512 then it will be divided into parts i.e. 124, 655, 12. After dividing the parts combine these parts by adding it.

H(key)=124+655+12
=791

Characteristics of good hashing function

1. The hash function should generate different hash values for the similar string.
2. The hash function is easy to understand and simple to compute.
3. The hash function should produce the keys which will get distributed, uniformly over an array.
4. A number of collisions should be less while placing the data in the hash table.
5. The hash function is a perfect hash function when it uses all the input data.

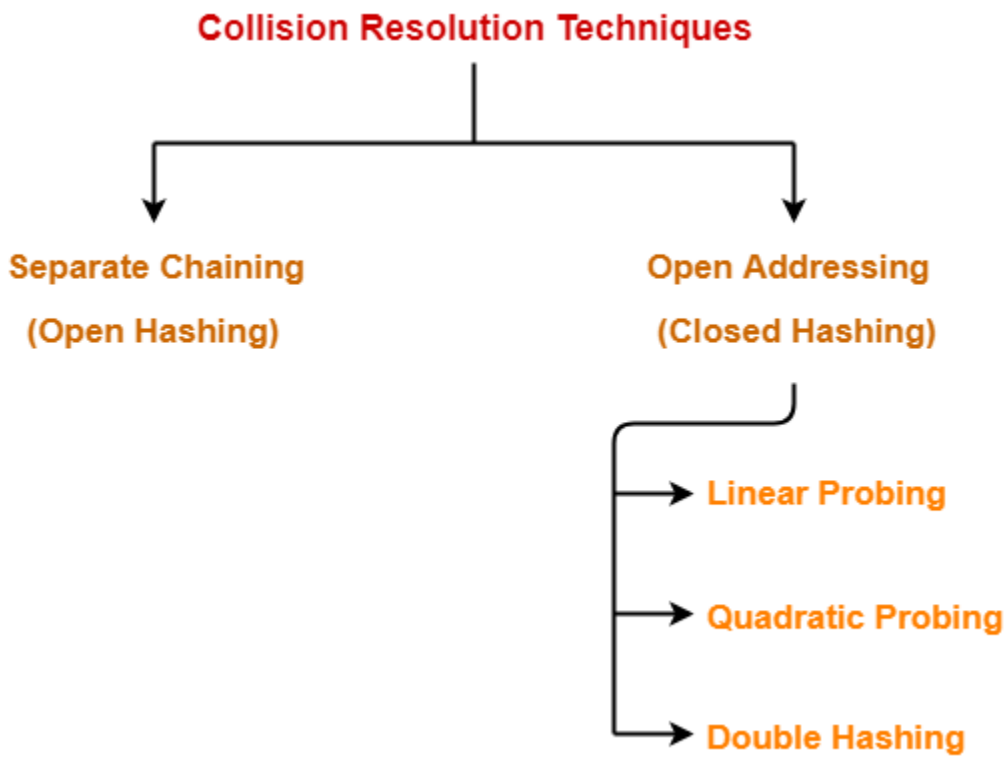
Collision

It is a situation in which the hash function returns the same hash key for more than one record, it is called as collision. Sometimes when we are going to resolve the collision it

may lead to a overflow condition and this overflow and collision condition makes the poor hash function.

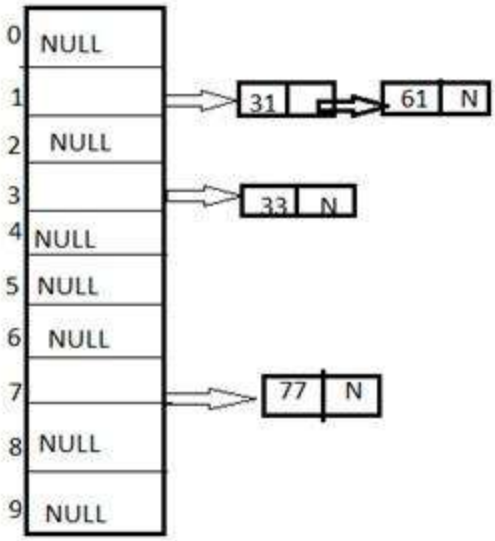
Collision resolution technique

If there is a problem of collision occurs then it can be handled by apply some technique. These techniques are called as collision resolution techniques. There are generally four techniques which are described below.



1) Chaining

It is a method in which additional field with data i.e. chain is introduced. A chain is maintained at the home bucket. In this when a collision occurs then a linked list is maintained for colliding data.

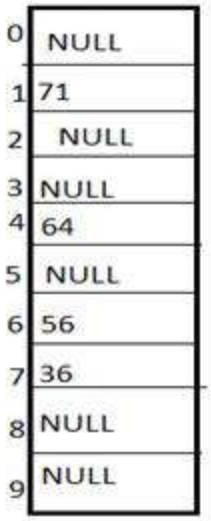


Example: Let us consider a hash table of size 10 and we apply a hash function of H(key)=key % size of table. Let us take the keys to be inserted are 31,33,77,61. In the above diagram we can see at same bucket 1 there are two records which are maintained by linked list or we can say by chaining method.

2) Linear probing

It is very easy and simple method to resolve or to handle the collision. In this collision can be solved by placing the second record linearly down, whenever the empty place is found. In this method there is a problem of clustering which means at some place block of a data is formed in a hash table.

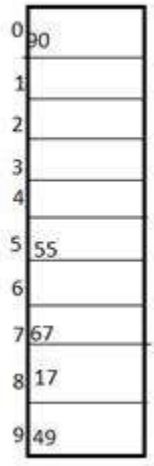
Example: Let us consider a hash table of size 10 and hash function is defined as H(key)=key % table size. Consider that following keys are to be inserted that are 56,64,36,71.



In this diagram we can see that 56 and 36 need to be placed at same bucket but by linear probing technique the records linearly placed downward if place is empty i.e. it can be seen 36 is placed at index 7.

3) Quadratic probing

This is a method in which solving of clustering problem is done. In this method the hash function is defined by the H(key)=(H(key)+x*x)%table size. Let us consider we have to insert following elements that are:-67, 90,55,17,49.



In this we can see if we insert 67, 90, and 55 it can be inserted easily but at case of 17 hash function is used in such a manner that :-(17+0*0)%10=17 (when x=0 it provide the index value 7 only) by making the increment in value of x. let x =1 so (17+1*1)%10=8.in this case bucket 8 is empty hence we will place 17 at index 8.

4) Double hashing

It is a technique in which two hash function are used when there is an occurrence of collision. In this method 1 hash function is simple as same as division method. But for the second hash function there are two important rules which are

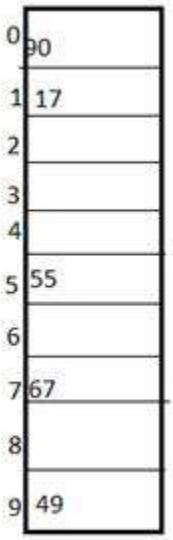
1. It must never evaluate to zero.
2. Must sure about the buckets, that they are probed.

The hash functions for this technique are:

H1(key)=key % table size
H2(key)=P-(key mod P)

Where, **p** is a prime number which should be taken smaller than the size of a hash table.

Example: Let us consider we have to insert 67, 90,55,17,49.



In this we can see 67, 90 and 55 can be inserted in a hash table by using first hash function but in case of 17 again the bucket is full and in this case we have to use the second hash function which is H2(key)=P-(key mode P) here p is a prime number which should be taken smaller than the hash table so value of p will be the 7.

i.e. H2(17)=7-(17%7)=7-3=4 that means we have to take 4 jumps for placing the 17. Therefore 17 will be placed at index 1.