

Biyani's Think Tank

Concept based notes

Data Structure and Algorithm

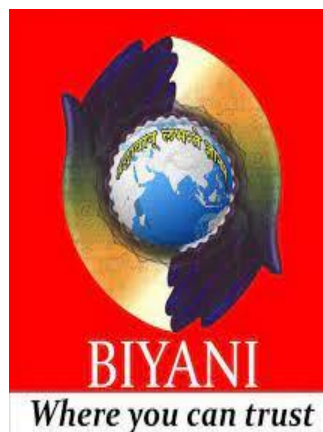
BCA Part-III

Mr. Rahul Agarwal, Ms. Bhavana Sangamnarkar

Asst. Professor

Dept. of IT

Biyani Girls College, Jaipur



Published by :

Think Tanks

Biyani Group of Colleges

Concept & Copyright :

©Biyani Shikshan Samiti

Sector-3, Vidhyadhar Nagar,

Jaipur-302 023 (Rajasthan)

Ph : 0141-2338371, 2338591-95 Fax : 0141-2338007

E-mail : acad@biyanicolleges.org

Website : www.gurukpo.com; www.biyanicolleges.org

ISBN : 978-93-83462-33-9

Edition: 2023

While every effort is taken to avoid errors or omissions in this Publication, any mistake or omission that may have crept in is not intentional. It may be taken note of that neither the publisher nor the author will be responsible for any damage or loss of any kind arising to anyone in any manner on account of such errors and omissions.

Leaser Type Setted by :

Biyani College Printing Department

Preface

I am glad to present this book, especially designed to serve the needs of the students. The book has been written keeping in mind the general weakness in understanding the fundamental concepts of the topics. The book is self-explanatory and adopts the “Teach Yourself” style. It is based on question-answer pattern. The language of book is quite easy and understandable based on scientific approach.

Any further improvement in the contents of the book by making corrections, omission and inclusion is keen to be achieved based on suggestions from the readers for which the author shall be obliged.

I acknowledge special thanks to Mr. Rajeev Biyani, *Chairman* & Dr. Sanjay Biyani, *Director (Acad.)* Biyani Group of Colleges, who are the backbones and main concept provider and also have been constant source of motivation throughout this Endeavour. They played an active role in coordinating the various stages of this Endeavour and spearheaded the publishing work.

I look forward to receiving valuable suggestions from professors of various educational institutions, other faculty members and students for improvement of the quality of the book. The reader may feel free to send in their comments and suggestions to the under mentioned address.

Author

Content

S.No.	Name of Topic
1.	Basics of Algorithms
2.	Data Structure Organisation
3.	Arrays
4.	Strings
5.	C++ : Classes and Objects
6.	Linked List
7.	Stacks
8.	Queue
9.	Sorting and Searching Techniques
10.	Graph
11.	Tree
12.	Unsolved Papers 2011 to 2006
13.	MCQ

Chapter-1

Basics of Algorithms

Q.1. What are the various steps to plan algorithm?

Ans.: Following steps must be followed to plan any algorithm :

- (1) **Device Algorithm :** Creating an algorithm is an art in which may never be fully automated. When we get the problem, we should first analyse the given problem clearly and then write down some steps on the paper.
- (2) **Validate Algorithm :** Once an algorithm is devised , it is necessary to show that it computes the correct answer for all possible legal inputs . This process is known as algorithm validation. The algorithm need not as yet be expressed as a program. It is sufficient to state it in any precise way. The purpose of validation is to assure us that this algorithm will work correctly independently of the issues concerning the programming language it will eventually be written in. Once the validity of the method has been shown, a program can be written and a second phase begins. This phase is referred to as program proving or program verification.
- (3) **Analyse Algorithm :** As an algorithm is executed , it uses the computers central processing unit to perform operations and its memory (both immediate and auxiliary) to hold the program and data. Analysis of algorithm or performance analysis refers to the task of determining how much computing time and storage an algorithm requires. An important result of this study is that it allows you to make quantitative judgments about the value of one algorithm over another. Another result is that it allows you to predict whether the software will meet any efficiency constraints that exist. Analysis can be made by taking into consideration.

- (4) **Test A Program** : Testing a program consists of 2 phases : debugging and performance management. Debugging is the process of executing programs on sample data sets to determine whether results are incorrect if so corrects them. Performance management is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results. These timing figures are useful in that they may confirm a previously done analysis and point out logical places to perform useful optimization.

Q.2. Define Algorithms with suitable example.

Ans.: Consider the following three examples. What do they all have in common?

How to change your motor oil

- (1) Place the oil pan underneath the oil plug of your car.
- (2) Unscrew the oil plug.
- (3) Drain oil.
- (4) Replace the oil plug.
- (5) Remove the oil cap from the engine.
- (6) Pour in 4 quarts of oil.
- (7) Replace the oil cap.

Therefore an algorithm is a set of instructions for solving a problem. Once we have created an algorithm, we no longer need to think about the principles on which the algorithm is based. This means that algorithms are a way of capturing intelligence and sharing it with others. Once you have encoded the necessary intelligence to solve a problem in an algorithm, many people can use your algorithm without needing to become experts in a particular field.

Q.3 Why the algorithm are important to computers. How can we create it in understandable form?

Ans. Algorithms are especially important to computers because computers are really general purpose machines for solving problems. But in order for a computer to be useful, we must give it a problem to solve and a technique for

solving the problem. Through the use of algorithms, we can make computers "intelligent" by programming them with various algorithms to solve problems. Because of their speed and accuracy, computers are well-suited for solving tedious problems such as searching for a name in a large telephone directory or adding a long column of numbers. However, the usefulness of computers as problem solving machines is limited because the solutions to some problems cannot be stated in an algorithm.

Much of the study of computer science is dedicated to discovering efficient algorithms and representing them so that they can be understood by computers.

An informal definition of an algorithm is "a set of instructions for solving a problem" and we illustrated this definition with a recipe, and instructions for changing the oil in a car engine. You also created your own algorithm for putting letters and numbers in order. While these simple algorithms are fine for us, they are much too ambiguous for a computer. In order for an algorithm to be applicable to a computer, it must have certain characteristics. We will specify these characteristics in our formal definition of an algorithm.

An **algorithm** is a well-ordered collection of unambiguous and effectively computable operations that when executed produces a result and halts in a finite amount of time [[Schneider and Gersting 1995](#)].

With this definition, we can identify five important characteristics of algorithms :

- (1) Algorithms are well-ordered.
- (2) Algorithms have unambiguous operations.
- (3) Algorithms have effectively computable operations.
- (4) Algorithms produce a result.
- (5) Algorithms halt in a finite amount of time.

When writing algorithms, we have several choices of how we will specify the operations in our algorithm. One option is to write the algorithm using plain English. Although plain English may seem like a good way to write an algorithm, it has some problems that make it a poor choice. First, plain English is too wordy. When we write in plain English, we must include many words

that contribute to correct grammar or style but do nothing to help communicate the algorithm. Second, plain English is too ambiguous. Often an English sentence can be interpreted in many different ways. Remember that our definition of an algorithm requires that each operation be unambiguous.

Another option for writing algorithms is using programming languages. These languages are collections of primitives (basic operations) that a computer understands. While programming languages avoid the problems of being wordy and ambiguous, they have some other disadvantages that make them undesirable for writing algorithms. Consider the following lines of code from the programming language C++.

```
a = 1;
b = 0;
while (a <= 10)
{
    b = b + a;
    a++;
}
cout << b;
```

This algorithm sums the numbers from 1 to 10 and displays the answer on the computer screen. However, without some special knowledge of the C++ programming language, it would be difficult for you to know what this algorithm does. Using a programming language to specify algorithms means learning special syntax and symbols that are not part of Standard English. For example, in the code above, it is not very obvious what the symbol "++" or the symbol "<<" does. When we write algorithms, we would rather not worry about the details of a particular programming language.

What we would really like to do is combine the familiarity of plain English with the structure and order of programming languages. A good compromise is structured English. This approach uses English to write operations, but groups operations by indenting and numbering lines. Each operation in the algorithm is written on a separate line so they are easily distinguished from

each other. We can easily see the advantage of this organization by comparing the structured English algorithm with the plain English algorithm.

How to change your motor oil	
Plain English	Structured English
First, place the oil pan underneath the oil plug of your car. Next, unscrew the oil plug and drain the oil. Now, replace the oil plug. Once the old oil is drained, remove the oil cap from the engine and pour in 4 quarts of oil. Finally, replace the oil cap on the engine.	<ol style="list-style-type: none"> 1. Place the oil pan underneath the oil plug of your car. 2. Unscrew the oil plug. 3. Drain oil. 4. Replace the oil plug. 5. Remove the oil cap from the engine. 6. Pour in 4 quarts of oil. 7. Replace the oil cap.

For the remainder of this study, we will write our algorithms using the structured English approach.

Q.4. How can we analyse an Algorithm?

Ans.: To analyze an [algorithm](#) is to determine the amount of resources (such as time and storage) necessary to execute it. Most algorithms are designed to work with inputs of arbitrary length. Usually the **efficiency** or [complexity](#) of an algorithm is stated as a function relating the [input length](#) to the number of steps (**time complexity**) or storage locations (**space complexity**).

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 of efficient algorithms.

Q.5. Explain Analysis of Algorithm in terms of different Order Notations.

Ans.: In theoretical analysis of algorithms it is common to estimate their complexity in **asymptotic sense**, i.e., to estimate the complexity function for reasonably large length of input. [Big O notation](#), [omega](#) notation and [theta](#) notation are used to this end. For instance, [binary search](#) is said to run an amount of steps proportional to a logarithm, 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 **hidden constant**.

Asymptotic Notation :

If we want to treat large problems (these are the critical ones), we are interested in the asymptotic behavior of the growth of the running time function.

Thus, when comparing the running times of two algorithms :

Constant factors can be ignored.

Lower order terms are unimportant when the higher order terms are different.

For instance, when we analyze selection sort, we find that it takes $T(n) = n^2 + 3n - 4$ array accesses.

For large values of n , the $3n - 4$ part is insignificant compared to the n^2 part.

An algorithm that takes a time of $100n^2$ will still be faster than an algorithm that n^3 for any value of n larger than 100.

Asymptotically Tight Upper and Lower Bound : Big-Theta " Θ "-Notation :

Definition : Let $f(n)$ and $g(n)$ be real-valued functions of a single non-negative integer argument. We write $f(n) = \Theta(g(n))$ if there exist positive real-valued constants c_1 , c_2 and a positive integer n_0 such that :

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 .$$

In other words : For large inputs (asymptotically) $f(n)$ is "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$.

Examples :

$2n^2+3n$ is $\Theta(n^2)$.

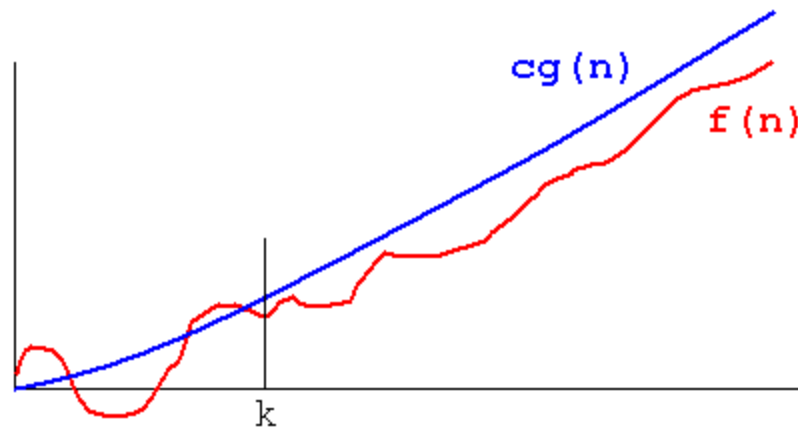
$2n^3+3n$ is not $\Theta(n^2)$.

Asymptotically Tight Upper Bound : Big-Oh "O"-Notation :

Definition : Let $f(n)$ and $g(n)$ be real-valued functions of a single non-negative integer argument. We write $f(n) = O(g(n))$ if there exist a positive real-valued constant c and a positive integer n_0 such that :

$$0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0 .$$

In other words, for large inputs (asymptotically) $f(n)$ is below $c g(n)$:



The " Θ "-Notation is stronger than the O-notation.

Examples :

$3n-6$ is $O(n)$.

$9n^4+12n^2+1234$ is $O(n^4)$.

$n+\log(n)$ is $O(n)$.

$\log(n)+5\log(\log(n))$ is $O(\log(n))$.

1234554321 is $O(1)$.

$3/n$ is $O(1/n)$.

Asymptotically Tight Lower Bound : Big-Omega " Ω "-Notation :

Definition : Let $f(n)$ and $g(n)$ be real-valued functions of a single non-negative integer argument. We write $f(n) = \Omega(g(n))$ if there exist a positive real-valued constant c and a positive integer n_0 such that :

$$0 \leq c g(n) \leq f(n) \text{ for all } n \geq n_0 .$$

In other words : For large inputs (asymptotically) $f(n)$ is above $c g(n)$.

Note :

We have $f(n) = \Theta(g(n))$ iff (if and only if)

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

In other words :

If $f(n)$ is $\Theta(g(n))$, the $f(n)$ is $\Theta(g(n))$.

If $f(n)$ is $\Theta(g(n))$, then $f(n)$ is $O(g(n))$.

Example :

$$2n^3 + 3n \text{ is } \Theta(n^3).$$

Asymptotically Non-Tight Upper Bound : Little-oh "o"-Notation :

Definition : Let $f(n)$ and $g(n)$ be real-valued functions of a single non-negative integer argument. We write $f(n) = o(g(n))$ if **for any** positive real-valued constant c exists a positive integer n_0 such that :

$$0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0 .$$

In other words, the inequality holds for all constants c .

For any constant c we can come up with large enough inputs n so that $f(n)$ is below $c g(n)$.

$$g(n) \text{ grows a lot faster than } f(n) : f(n)/g(n) \rightarrow 0 \text{ as } n \rightarrow \infty.$$

Example :

$$\log(n) \text{ is } o(n)$$

Asymptotically Non-Tight Lower Bound: Little-Omega " ω " Notation

Definition : Let $f(n)$ and $g(n)$ be real-valued functions of a single non-negative integer argument. We write $f(n) = \omega(g(n))$ if **for any** positive real-valued constant c exists a positive integer n_0 such that :

$$0 \leq c g(n) \leq f(n) \text{ for all } n \geq n_0 .$$

As an alternative definition we can say the following :

Definition : Let $f(n)$ and $g(n)$ be real valued functions of an integer variable. We say **$f(n)$ is $\omega(g(n))$** if $g(n)$ is $o(f(n))$. This is pronounced as " $f(n)$ is little-omega of $g(n)$ ".

$f(n)$ grows a lot faster than $g(n)$: $f(n)/g(n) \rightarrow \infty$ as $n \rightarrow \infty$.

Example :

$$n^2 \text{ is } \omega(n \log(n)).$$

Q.6. Define Efficiency of an Algorithm in terms of Time and Space.

Ans.: Efficiency of an algorithm : In [computer science](#), **efficiency** is used to describe properties of an [algorithm](#) relating to how much of various types of resources it consumes. The two most frequently encountered are speed or running time, the time it takes for an algorithm to complete, and space, the memory or non-volatile storage used by the algorithm during its operation. [Optimization](#) is the process of making code as efficient as possible, sometimes focusing on space at the cost of speed, or vice versa.

The speed of an algorithm is measured in various ways. The most common method uses [time complexity](#) to determine the [Big-O](#) of an algorithm: often, it is possible to make an algorithm faster at the expense of space. This is the case whenever you cache the result of an expensive calculation rather than recalculating it on demand. This is a very common method of improving speed, so much so that languages often add special features to support it, such as [C++](#)'s mutable keyword.

The space of an algorithm is actually two separate but related things. The first part is the space taken up by the compiled executable on disk (or equivalent, depending on the hardware and language) by the algorithm. This can often be reduced by preferring run-time decision making mechanisms (such as [virtual functions](#) and [run-time type information](#)) over certain compile-time decision

making mechanisms (such as [macro substitution](#) and [templates](#)). This, however, comes at the cost of speed.

The other part of algorithm space measurement is the amount of temporary memory taken up during processing. For example, pre-caching results, as mentioned earlier, improves speed at the cost of this attribute.

Optimization of algorithms frequently depends on the properties of the machine the algorithm will be executed on. For example, one might optimize code for time efficiency in applications for home computers with sizable amounts of memory, while code to be placed in small, memory-tight devices may have to be made to run slower to conserve space.

One simple way to determine whether an optimization is worthwhile is as follows: Let the original time and space requirements (generally in Big-O notation) of the algorithm be O_1 and O_2 . Let the new code require N_1 and N_2 time and space respectively. If $N_1N_2 < O_1O_2$, the optimization should be carried out. However, as mentioned above, this may not always be true.

One must be careful, in the pursuit of good coding style, not to over-emphasize efficiency. Nearly all of the time, a clean and usable design is much more important than a fast, small design. There are exceptions to this rule (such as [embedded systems](#), where space is tight, and processing power minimal) but these are rarer than one might expect.

Computational complexity theory, as a branch of the [theory of computation](#) in [computer science](#), investigates the problems related to the [amounts of resources](#) required for the execution of [algorithms](#) (e.g., execution time), and the inherent difficulty in providing efficient algorithms for specific [computational problems](#).

A typical question of the theory is, "As the size of the input to an algorithm increases, how do the running time and memory requirements of the algorithm change and what are the implications and ramifications of that change?" In other words, the theory, among other things, investigates the [scalability](#) of computational problems and algorithms. In particular, the theory places practical limits on what [computers](#) can accomplish.

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 set to which we apply [binary search](#) has N elements, and we can guarantee that a single binary lookup can be done in unit time, then at most $\log_2 N + 1$ time units are needed to return an answer.

Exact measures of efficiency are useful to the people who actually implement and use algorithms, because they are more precise and thus enable them to know how much time they can expect to spend in execution. To some people (e.g. game programmers), a hidden constant can make all the difference between success and failure.

Time efficiency estimates depend on what we define to be a step. For the analysis to make sense, the time required to perform a step must be guaranteed to be bounded above by a constant. One must be careful here; for instance, some analyses count an addition of two numbers as a step. This assumption may not be warranted in certain contexts. For example, if the numbers involved in a computation may be arbitrarily large, addition no longer can be assumed to require constant time.

Q.7. Compare the Performance of an Algorithm. Explain Algorithmic Complexity Measures.

Ans.: Our aim is to compare the performance of algorithms.

We can consider the average case or the worst case performance (referring to an instance of a given problem).

Usually we treat only the worst case performance :

The worst case occurs fairly often, example: looking for an entry in a database that is not present.

The result of the worst case analysis is often not different to the average case analysis (same order of complexity).

Running Times : One way to compare running times is simply to run several tests for each algorithm and compare the timings.

Another way is to estimate the time required for an algorithm to solve a problem.

The result of the analysis of an algorithm is usually a formula giving the amount of machine operations (e.g., floating point operations, number of memory accesses, number of comparisons, etc) or some other metric (e.g., number of elements to sort, neurons or connections in a neural network, vertices or edges in a tree, etc.), that the algorithm takes to complete.

In the following we write running times as a function of n .

Space Complexity : Running time is usually the thing we care most about. But it can be as well important to analyze the amount of memory used by a program.

If a program takes a lot of time, you can still run it, and just wait longer for the result.

However if a program takes a lot of memory, you may not be able to run it at all, so this is an important parameter to understand.

We analyze things differently for recursive and iterative programs.

For an **iterative program**, it is usually just a matter of looking at the variable declarations and storage allocation calls, e.g., array of n numbers.

Analysis of **recursive program** space is more complicated: the space used at any time is the total space used by all recursive calls active at that time.

Each recursive call takes a constant amount of space: some space for local variables and function arguments, and also some space for remembering where each call should return to.

Q.8. What is Pseudo Code? Explain how can we use Pseudo Code to design a Program?

Ans.: Pseudo code is a short hand way of describing a computer program. Rather than use the specific syntax of a computer language, more general wording is used. Using pseudocode, it is easier for a non-programmer to understand the general workings of the program.

Pseudo code (pronounced SOO-doh-kohd) is a detailed yet readable description of what a computer program or algorithm must do, expressed in a

Because pseudocode is detailed yet readable, it can be inspected by the team of designers and programmers as a way to ensure that actual programming is likely to match design specifications. Catching errors at the pseudocode stage is less costly than catching them later in the development process. Once the pseudocode is accepted, it is rewritten using the vocabulary and [syntax](#) of a programming language. Pseudocode is sometimes used in conjunction with computer-aided software engineering-based methodologies.

Pseudocode Example :

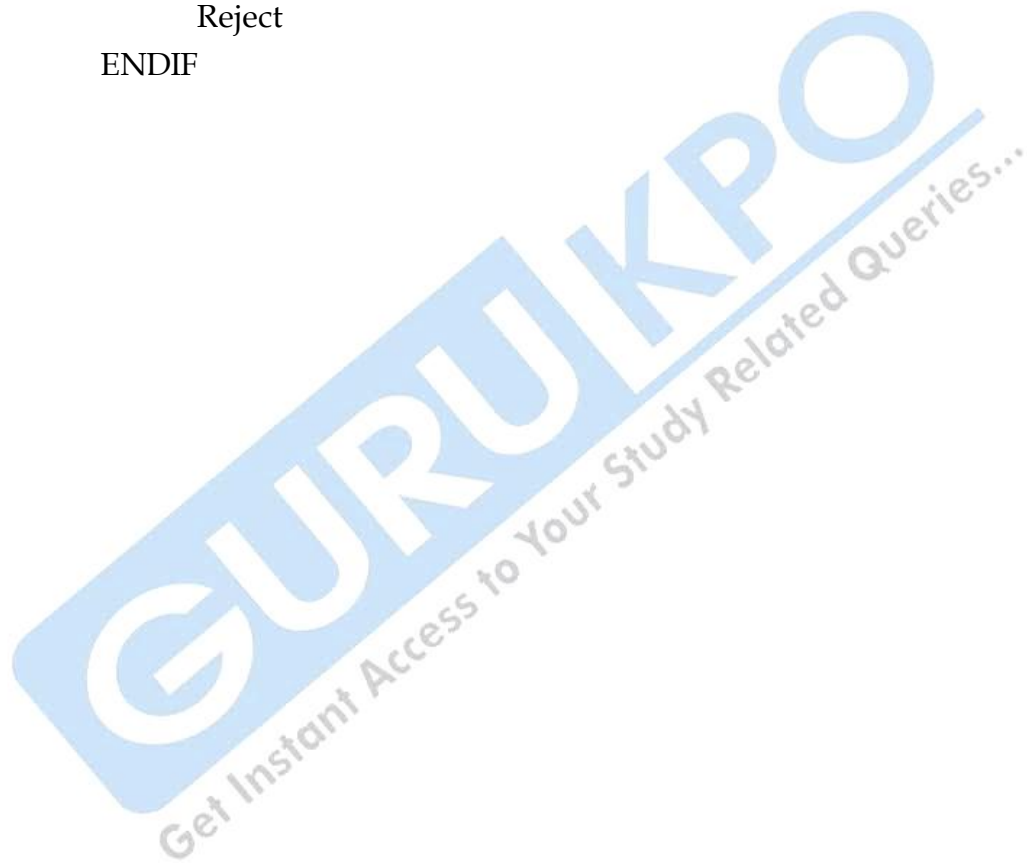
- if** credit card number is valid **then**
 execute transaction based on number and order
else
 show a generic failure message
end if
- A bank will grant loan under the following conditions :

 - If a customer has an account with the bank and had no loan outstanding, loan will be granted.
 - If a customer has an account with the bank but some amount is outstanding from previous loans then loan will be granted if special approval is needed.
 - Reject all loan applications in all other cases.

IF customer has a Bank Account THEN
 IF Customer has no dues from previous account THEN
 Allow loan facility
 ELSE

```
        IF Management Approval is obtained THEN
            Allow loan facility
        ELSE
            Reject
        ENDIF
    ENDIF
ELSE
    Reject
ENDIF
```

□ □ □



Chapter-2

Data Structure Ogranisation

Q.1. Explain Data Types along with their sizes and ranges.

Ans.: Data Types : The *type* of a data object in C determines the range and kind of values an object can represent, the size of machine storage reserved for an object, and the operations allowed on an object. Functions also have types, and the function's return type and parameter types can be specified in the function's declaration

Data Sizes : An object of a given data type is stored in a section of memory having a discreet size. Objects of different data types require different amounts of memory. Following table shows the size and range of the basic data types.

Sizes and Ranges of Data Types :

Type	Size	Range
Integral Types :		
short int , or signed short int	16 bits	-32768 to 32767
unsigned short int	16 bits	0 to 65535
int or signed int	32 bits	-2147483648 to 2147483647
unsigned int	32 bits	0 to 4294967295
long int , or signed long int (OpenVMS)	32 bits	- 2147483648 to 2147483647
long int , or signed long int (Digital UNIX)	64 bits	- 9223372036854775808 to 9223372036854775807

unsigned long int (<i>OpenVMS</i>)	32 bits	0 to 4294967295
unsigned long int (<i>Digital UNIX</i>)	64 bits	0 to 18446744073709551615
signed __int64 (<i>Alpha</i>)	64 bits	-9223372036854775808 to 9223372036854775807
unsigned __int64 (<i>Alpha</i>)	64 bits	0 to 18446744073709551615

Type	Size	Range
Integral Character Types :		
char and signed char	8 bits	-128 to 127
unsigned char	8 bits	0 to 255
wchar_t	32 bits	0 to 4294967295
Floating- Point Types (range is for absolute value) :		
float	32 bits	1.1×10^{-38} to 3.4×10^{38}
double	64 bits	2.2×10^{-308} to 1.7×10^{308}
long double (<i>OpenVMS Alpha</i>)	128 bits	3.4×10^{-49321} to 1.2×10^{49321}
long double (<i>OpenVMS VAX, Digital UNIX</i>)	Same as double	Same as double

Derived types can require more memory space.

Q.2. What is Data Abstraction? Define Abstract Data Types.

Ans.: Abstraction: In [computer science](#), **abstraction** is a mechanism and practice to reduce and factor out details so that one can focus on a few concepts at a time. The major purpose of a database system is to provide users with an **abstract view** of the system. The system hides certain details of how data is stored and created and maintained. Complexity should be hidden from users.

Data Abstraction: **Data abstraction** is the enforcement of a clear separation between the *abstract* properties of a [data type](#) and the *concrete* details of its implementation. The abstract properties are those that are visible to client code that makes use of the data type--the *interface* to the data type--while the

concrete implementation is kept entirely private, and indeed can change, for example to incorporate efficiency improvements over time. The idea is that such changes are not supposed to have any impact on client code, since they involve no difference in the abstract behaviour.

For example, one could define an [abstract data type](#) called *lookup table*, where *keys* are uniquely associated with *values*, and values may be retrieved by specifying their corresponding keys. Such a lookup table may be implemented in various ways: as a [hash table](#), a [binary search tree](#), or even a simple linear list. As far as client code is concerned, the abstract properties of the type are the same in each case.

Of course, this all relies on getting the details of the interface right in the first place, since any changes there can have major impacts on client code. Another way to look at this is that the interface forms a *contract* on agreed behaviour between the data type and client code; anything not spelled out in the contract is subject to change without notice.

Languages that implement data abstraction include [Ada](#) and [Modula-2](#). [Object-oriented](#) languages are commonly claimed to offer data abstraction; however, their [inheritance](#) concept tends to put information in the interface that more properly belongs in the implementation; thus, changes to such information ends up impacting client code, leading directly to the [Fragile binary interface problem](#).

Abstract Data Types :



The Concept of Abstraction :

- Abstraction allows one to collect instances of entities into groups in which their common attributes need not be considered.
- Two kinds of abstractions in programming languages are process abstraction and data abstraction.
- The concept of **process abstraction** is one of the oldest. All subprograms are process abstractions because they provide a way for a program to specify that some process is to be done, without providing the details of how it is to be done.

- Process abstraction is crucial to the programming process.? The ability to abstract away many of the details of algorithms in subprograms makes it possible to construct, read, and understand large programs.
- All subprograms, including concurrent subprograms, and exception handlers, are process abstractions.

**Encapsulation :**

- **Encapsulation** is a grouping of subprograms and the data that they manipulate.
- An encapsulation provides an abstracted system and a logical organization for a collection of related computations.
- They are often placed in libraries and made available for reuse in programs other than those for which they are written.

**Introduction to Data Abstraction :**

- An abstract data type is simply an encapsulation that includes only the data representation of one specific data type and the subprograms that provide the operations for that type.
- An instance of an abstract data type is called an object.
- Object-oriented programming is an outgrowth of the use of **data abstraction**.

**Floating-Point as an Abstract Data Type :**

- All built-in types are abstract data types, even those of FORTRAN I
- Floating-point types employ a key concept in data abstraction: information hiding? The actual format of the data in a floating-point cell is hidden from the user.

**User Defined Abstract Data Types :**

- The concept of user-defined abstract data types is relatively recent.
- They should provide :

- A type definition that allows program units to declare variables of the type but hides the representation of these variables.
- A set of operations for manipulating objects of the type.
- An **abstract data type** is a data type that satisfies two conditions :
 - The representation, or definition, of the type and the operations are contained in a single syntactic unit.
 - The representation of objects of the type is hidden from the program units that use the type, so only direct operations possible on those objects are those provided in the type? Definition.
- Program units that use a specific abstract data type are called **clients** of that type.
- A benefit of information hiding is increased reliability.? This is because clients cannot change the underlying representations of objects directly, either intentionally or by accident, thus increasing the integrity of the object



Design Issues :

- A facility for defining abstract data types in a language must provide a syntactic unit that can encapsulate the type definition and subprogram definitions of the abstraction operations.
- Concurrent Pascal, Smalltalk, C++, and Java directly support abstract data types.
- Some design issues beyond encapsulation are whether the kinds of types that can be abstract should be restricted, whether abstract data types can be parameterized, and what access controls are provided, and how such controls are specified.



Language Examples :

- C++
 - Unlike, Ada and Modula-2, which provide encapsulation that can used to simulate abstract data types, C++

provides the class, which more directly support abstract data types.

- The data defined in a class are called **data members**; the functions defined in a class are called **member functions**.
- Classes may contain both hidden and visible entities.

□ □ □



Chapter-3

Arrays

Q.1. What are the characteristics of Arrays in C?

- Ans.:**
- 1) An array holds elements that have the same data type.
 - 2) Array elements are stored in subsequent memory locations.
 - 3) Two-dimensional array elements are stored row by row in subsequent memory locations.
 - 4) Array name represents the address of the starting element.
 - 5) Array size should be mentioned in the declaration. Array size must be a constant expression and not a variable.

Q.2. How do I know how many elements an Array can hold?

Ans.: The amount of memory an array can consume depends on the data type of an array. In DOS environment, the amount of memory an array can consume depends on the current memory model (i.e. Tiny, Small, Large, Huge, etc.). In general an array cannot consume more than 64 kb. Consider following program, which shows the maximum number of elements an array of type int, float and char can have in case of Small memory model.

```
main()  
{  
    int i[32767];  
    float f[16383];  
    char s[65535];  
}
```

Q.3. How can we declare an Array?

Ans.: Declaring Arrays : Arrays are declared with the bracket punctuators [], as shown in the following syntax :

storage-class-specifier(opt) type-specifier declarator [constant-expression-list(opt)];

The following example shows a declaration of a 10-element array of integers, a variable called table_one :

```
int table_one[10];
```

The *type-specifier* shows the data type of the elements. The elements of an array can be of any scalar or aggregate data type. The identifier table_one specifies the name of the array. The constant expression 10 gives the number of elements in a single dimension. Arrays in C are zero-based; that is, the first element of the array is identified with a 0 subscript, such as the one shown in the following example :

```
int x[5];
```

```
x[0] = 25; /* The first array element is assigned the value 25 */
```

The expression between the brackets in the declaration must be an integral constant expression with a value greater than zero. Omitting the constant expression creates an incomplete array declaration, which is useful in the following cases :

- If the array is declared external and its storage is allocated by a definition in another place, you can omit the constant expression for convenience when the array name is declared, as in the following example :

```
extern int array1[];
```

```
int first_function(void)
```

```
{
```

```
.
```

```
.
```

```
.
```

```
}
```

In a separate compilation unit :

```
int array1[10];
```



```

int second_function(void)
{
    .
    .
    .
}

```

The array size specifier may only be omitted from the first pair of brackets in a multidimensional array declaration. This is because an array's elements must have complete types, even if the array itself has an incomplete type.

- If the declaration of the array includes initializers, you can omit the size of the array, as in the following example :

```

char array_one[] = "Shemps";
char array_two[] = { 'S', 'h', 'e', 'm', 'p', 's', '\0' };

```

The two definitions initialize variables with identical elements. These arrays have seven elements: six characters and the null character (`\0`), which terminates all character strings. The size of the array is determined from the number of characters in the initializing character-string constant or initialization list. Initializing an incomplete array completes the array type. An array is completed at the end of its initializer list.

- If you use the array as a function parameter, the array must be defined in the calling function. However, the declaration of the parameter in the called function can omit the constant expression within the brackets. The address of the first element of the array is passed. Subscripted references in the called function can modify elements of the array. The following example shows how to use an array in this manner :

```

main()
{
    /* Initialize array */
    static char arg_str[] = "Thomas";

```

```

int sum;
sum = adder(arg_str); /* Pass address of first array element */
.
.
.
}
/* adder adds ASCII values of letters in array */

int adder( char param_string[])
{
    int i, sum = 0;    /* Incrementer and sum */
    /* Loop until NULL char */
    for (i = 0; param_string[i] != '\0'; i++)
        sum += param_string[i];
    return sum;
}

```

After the function `adder` is called, parameter `param_string` receives the address of the first character of argument `arg_str`, which can then be accessed in `adder`. The declaration of `param_string` serves only to give the type of the parameter, not to reserve storage for it.

Array members can also be pointers. The following example declares an array of floating-point numbers and an array of pointers to floating-point numbers :

```
float fa[11], *afp[17];
```

When a function parameter is declared as an array, the compiler treats the declaration as a pointer to the first element of the array. For example, if `x` is a parameter and is intended to represent an array of integers, it can be declared as any one of the following declarations :

```
int x[];
```

```
int *x;  
int x[10];
```

Note that the specified size of the array does not matter in the case of a function parameter, since the pointer always points to only the first element of the array.

C supports arrays declared as an array of arrays. These are sometimes called multidimensional arrays. Consider the following example, where variable `table_one` is a two-dimensional array containing 20 integers :

```
int table_one[10][2];
```

Arrays are stored in row-major order, which means the element `table_one[0][0]` (in the previous example) immediately precedes `table_one[0][1]`, which in turn immediately precedes `table_one[1][0]`.

Q.4. What are the steps to initialize an Array?

Ans.: Initializing Arrays : Arrays are initialized with a brace-enclosed list of constant expressions. A list of initializers for an incomplete array declaration completes the array's type and completely defines the array size. Therefore, when initializing an array of unknown size, the number of initializers in the initializer list determines the size of the array. For example, the following declaration initializes an array of three elements :

```
int x[] = { 1, 2, 3 };
```

If the array being initialized has a storage class of `static`, the initializers must be constant expressions.

Initializers for an array of a given size are assigned to array members on a one-to-one basis. If there are too few initializers for all members, the remaining members are initialized to 0. Listing too many initializers for a given size array is an error. For example :

```
int x[5] = { 0, 1, 2, 3, 4, 5 }; /* error */
```

String literals are often assigned to a `char` or `wchar_t` array. In this case, each character of the string represents one member of a one-dimensional array, and the array is terminated with the null character. When an array is initialized by

a pointer to a string literal, the string literal cannot be modified through the pointer.

When initializing an array with a string literal, use quotation marks around the initializing string. For example :

```
char string[26] = { "This is a string literal." };  
/* The braces above are optional here */
```

The terminating null character is appended to the end of the string if the size permits, as it does in this case. Another form for initializing an array with characters is the following :

```
char string[12] = {'T', 'h', 'i', 's', ' ', 'w', 'a', 'y' };
```

The preceding example creates a one-dimensional array containing the string value "This way ". The characters in this array can be freely modified. Remaining uninitialized array members will be automatically initialized to zero.

If the size of the array used for a string literal is not explicitly stated, its size is determined by the number of characters in the string (including the terminating null character). If the size of the array is explicitly stated, initializing the array with a string literal longer than the array is an error.

Note : There is one special case where the null character is not automatically appended to the array. This case is when the array size is explicitly specified and the number of initializers completely fills the array size. For example :

```
char c[4] = "abcd";
```

Here, the array c holds only the four specified characters, a , b , c , and d . No null character terminates the array.

Using the following rules, you can omit braces when initializing the members of a multidimensional arrays:

- When initializing arrays, you can omit the outermost pair of braces.
- If the initializer list includes all of the initializers for the object being initialized, you can omit the inner braces.
-

Consider the following example :

```
float x[4][2] = {  
    { 1, 2 }  
    { 3, 4 }  
    { 5, 6 }  
};
```

In this example, 1 and 2 initialize the first row of the array `x`, and the following two lines initialize the second and third rows, respectively. The initialization ends before the fourth row is initialized, so the members of the fourth row default to 0. Here is the result :

```
x[0][0] = 1;  
x[0][1] = 2;  
x[1][0] = 3;  
x[1][1] = 4;  
x[2][0] = 5;  
x[2][1] = 6;  
x[3][0] = 0;  
x[3][1] = 0;
```

The following declaration achieves the same result :

```
float x[4][2] = { 1, 2, 3, 4, 5, 6 };
```

Here, the compiler fills the array row by row with the available initial values. The compiler places 1 and 2 in the first row (`x[0]`), 3 and 4 in the second row (`x[1]`), and 5 and 6 in the third row (`x[2]`). The remaining members of the array are initialized to zero.

Q.5. Compare and relate Pointers with Arrays.

Ans.: Pointers and Arrays : Data objects in an array can be referenced through pointers instead of using array subscripts. The data type of such a pointer is referred to as "pointer to array of *type*". The array name itself behaves like a

pointer, so there are several alternative methods to accessing array elements. For example :

```
int x[5] = { 0, 1, 2, 3, 4 };    /* Array x declared with five elements */
int *p = x;                     /* Pointer declared and initialized to point */
                                /* to the first element of the array x */

int a, b;
a = *(x + 3);                   /* Pointer x incremented by twelve bytes */
                                /* to reference element 3 of x */
b = x[3];                       /* b now holds the same value as a */
```

In the previous example, a receives the value 3 by using the dereferencing operator (*). b receives the same value by using the subscripting operator. For more information on the different unary operators.

Note that the assignment of a was a result of incrementing the pointer to x. This principle, known as *scaling*, applies to all types of pointer arithmetic. In scaling, the compiler considers the size of an array element when calculating memory addresses of array members. For example, each member of the array x is 4 bytes long, and adding three to the initial pointer value automatically converts that addition to 3 * (the size of the array member, which in this case is 4). Therefore, the intuitive meaning of $z = *(y + 3)$; is preserved.

When passing arrays as function arguments, only a pointer to the first element of the array is passed to the called function. The conversion from array type to pointer type is implicit. Once the array name is converted to a pointer to the first element of the array, you can increment, decrement, or dereference the pointer, just like any other pointer, to manipulate data in the array. For example :

```
int func(int *x, int *y)        /* The arrays are converted to pointers */
{
    *y = *(x + 4);               /* Various elements of the arrays are
                                accessed */
}
```


Remember that a pointer is large enough to hold only an address; a pointer into an array holds the address of an element of that array. The array itself is large enough to hold all members of the array.

When applied to arrays, the size of operator returns the size of the entire array, not just the size of the first element in the array.

Q.6. How will you explain Record and Record Structure?

Record and record structures : The `::struct::record` package provides a mechanism to group variables together as one data structure, similar to a 'C' structure. The members of a record can be variables or other records. However, a record can not contain circular record, i.e. records that contain the same record as a member.

This package was structured so that it is very similar to how Tk objects work. Each record definition creates a record object that encompasses that definition. Subsequently, that record object can create instances of that record. These instances can then be manipulated with the **cget** and **configure** methods.

The package only contains one top level command, but several sub commands (see below). It also obeys the namespace in which the record was define, hence the objects returned are fully qualified.

record define *recordName* *recordMembers* ? *instanceName1* *instanceName2* ... ?

Defines a record. *recordName* is the name of the record, and is also used as an object command. This object command is used to create instances of the record definition. *recordMembers* are the members of the record that make up the record definition. These are variables and other record. If optional *instanceName* args are given, then an instance is generated after the definition is created for each *instanceName*.

record show *record*

Returns a list of records that have been defined.

record show instances *recordName*

Returns the instances that have been instantiated by *recordName*.

record show members *recordName*

Returns the members that are defined for record *recordName*. It returns the same format as how the records were defined.

record show *values instanceName*

Returns a list of values that are set for the instance *instanceName*. The output is a list of key/value pairs. If there are nested records, then the values of the nested records will itself be a list.

record exists *record recordName*

Tests for the existence of a *record* with the name *recordName*.

record exists *instance instanceName*

Tests for the existence of a *instance* with the name *instanceName*.

record delete *record recordName*

Deletes *recordName*, and all instances of *recordName*. It will return an error if the record does not exist.

record delete *instance instanceName*

Deletes *instance* with the name of *instanceName*. It will return an error if the instance does not exist.

Record Members : Record members can either be variables, or other records, However, the same record can not be nested within itself (circular). To define a nested record, you need to specify the record keyword, along with the name of the record, and the name of the instance of that nested record. For example, it would look like this :

```
# this is the nested record
record define mynestedrecord {
    nest1
    nest2
}
# This is the main record
record define myrecord {
    mem1
```

```
mem2
{record mynestedrecord mem3}
}
```

□ □ □

GURUKPO
Get Instant Access to Your Study Related Queries...

Chapter-4

Strings

Q.1. What is the difference between a String and an Array?

Ans.: An array is an array of anything. A string is a specific kind of an array with a well-known convention to determine its length.

There are two kinds of programming languages: those in which a string is just an array of characters, and those in which it's a special type. In C, a string is just an array of characters (type char), with one wrinkle: a C string always ends with a NULL character. The "value" of an array is the same as the address of (or a pointer to) the first element; so, frequently, a C string and a pointer to char are used to mean the same thing. An array can be of any length. If it's passed to a function, there's no way the function can tell how long the array is supposed to be, unless some convention is used. The convention for strings is NULL termination; the last character is an ASCII NULL (") character.

Q.2. What is the difference between Strings and Character Arrays?

Ans.: A major difference is: string will have static storage duration, whereas as a character array will not, unless it is explicitly specified by using the static keyword.

Actually, a string is a character array with following properties :

- The multi-byte character sequence, to which we generally call string, is used to initialize an array of static storage duration. The size of this array is just sufficient to contain these characters plus the terminating NULL character.
- It not specified what happens if this array, i.e., string, is modified.

So the value of a string is the sequence of the values of the contained characters, in order.

Q.3. What is the difference between "calloc(...)" and "malloc(...)"?

Ans.: (1) `calloc(...)` allocates a block of memory for an array of elements of a certain size. By default the block is initialized to 0. The total number of memory allocated will be (number_of_elements * size).

`malloc(...)` takes in only a single argument which is the memory required in bytes. `malloc(...)` allocated bytes of memory and not blocks of memory like `calloc(...)`.

(2) `calloc(...)` allocates an array in memory with elements initialized to 0 and returns a pointer to the allocated space. `calloc(...)` calls `malloc(...)` in order to use the C++ `_set_new_mode` function to set the new handler mode.

`malloc(...)` allocates memory blocks and returns a void pointer to the allocated space, or NULL if there is insufficient memory available.

Q.4. What is the purpose of `realloc()`?

Ans.: The function `realloc(ptr,n)` uses two arguments. the first argument `ptr` is a pointer to a block of memory for which the size is to be altered. The second argument `n` specifies the new size. The size may be increased or decreased. If `n` is greater than the old size and if sufficient space is not available subsequent to the old region, the function `realloc()` may create a new region and all the old data are moved to the new region.

Q.5 What is difference between `malloc()/free()` and `new/delete`?

Ans.: `malloc` allocates memory for object in heap but doesn't invoke object's constructor to initialize the object.

`new` allocates memory and also invokes constructor to initialize the object.

`malloc()` and `free()` do not support object semantics

Does not construct and destruct objects

`string * ptr = (string *) (malloc (sizeof(string)))`

Are not safe

Does not calculate the size of the objects that it construct

Returns a pointer to void

```
int *p = (int *) (malloc(sizeof(int)));
```

```
int *p = new int;
```

Are not extensible

new and delete can be overloaded in a class

"delete" first calls the object's termination routine (i.e. its destructor) and then releases the space the object occupied on the heap memory. If an array of objects was created using new, then delete must be told that it is dealing with an array by preceding the name with an empty []:-

```
Int_t *my_ints = new Int_t[10];
```

...

```
delete []my_ints;
```

Q.6. What is the difference between "new" and "operator new" ?

Ans.: "operator new" works like malloc.

Q.7. What is the difference between realloc() and free()?

Ans.: The free subroutine frees a block of memory previously allocated by the malloc subroutine. Undefined results occur if the Pointer parameter is not a valid pointer. If the Pointer parameter is a null value, no action will occur. The realloc subroutine changes the size of the block of memory pointed to by the Pointer parameter to the number of bytes specified by the Size parameter and returns a new pointer to the block. The pointer specified by the Pointer parameter must have been created with the malloc, calloc, or realloc subroutines and not been deallocated with the free or realloc subroutines. Undefined results occur if the Pointer parameter is not a valid pointer.

Q.8. Is it better to use malloc() or calloc()?

Ans.: Both the malloc() and the calloc() functions are used to allocate dynamic memory. Each operates slightly different from the other. malloc() takes a size and returns a pointer to a chunk of memory at least that big:


```
void *malloc( size_t size );
```

calloc() takes a number of elements, and the size of each, and returns a pointer to a chunk of memory at least big enough to hold them all:

```
void *calloc( size_t numElements, size_t sizeOfElement );
```

There's one major difference and one minor difference between the two functions. The major difference is that malloc() doesn't initialize the allocated memory. The first time malloc() gives you a particular chunk of memory, the memory might be full of zeros. If memory has been allocated, freed, and reallocated, it probably has whatever junk was left in it. That means, unfortunately, that a program might run in simple cases (when memory is never reallocated) but break when used harder (and when memory is reused). calloc() fills the allocated memory with all zero bits. That means that anything there you're going to use as a char or an int of any length, signed or unsigned, is guaranteed to be zero. Anything you're going to use as a pointer is set to all zero bits. That's usually a null pointer, but it's not guaranteed. Anything you're going to use as a float or double is set to all zero bits; that's a floating-point zero on some types of machines, but not on all.

The minor difference between the two is that calloc() returns an array of objects; malloc() returns one object. Some people use calloc() to make clear that they want an array.

Q.9. Explain differences between eg. new() and malloc()

- Ans.1:** 1.) "new and delete" are preprocessors while "malloc() and free()" are functions. [we don't use brackets with calling new or delete].
- 2.) No need to allocate the memory while using "new" but in "malloc()" we have to use "sizeof()".
- 3.) "new" will initialize the new memory to 0 but "malloc()" gives random value in the new allotted memory location [better to use calloc()]

Ans.2: new() allocates continuous space for the object instance
malloc() allocates distributed space.

new() is castless, meaning that it allocates memory for this specific type, malloc(), calloc() allocate space for void * that is casted to the specific class type pointer.

Q.10. String Processing -- Write out a function that prints out all the permutations of a String. For example, abc would give you abc, acb, bac, bca, cab, cba.

Ans.: void PrintPermu (char *sBegin, char* sRest) {
 int iLoop;
 char cTmp;
 char cFLetter[1];
 char *sNewBegin;
 char *sCur;
 int iLen;
 static int iCount;
 iLen = strlen(sRest);
 if (iLen == 2) {
 iCount++;
 printf("%d: %s %s\n", iCount, sBegin, sRest);
 iCount++;
 printf("%d: %s %c%c\n", iCount, sBegin, sRest[1], sRest[0]);
 return;
 } else if (iLen == 1) {
 iCount++;
 printf("%d: %s %s\n", iCount, sBegin, sRest);
 return;
 } else {
 // swap the first character of sRest with each of
 // the remaining chars recursively call debug print
 sCur = (char*) malloc(iLen);

```

sNewBegin = (char*)malloc(iLen);
for (iLoop = 0; iLoop < iLen; iLoop++) {
    strcpy(sCur, sRest);
    strcpy(sNewBegin, sBegin);
    cTmp = sCur[iLoop];
    sCur[iLoop] = sCur[0];
    sCur[0] = cTmp;
    sprintf(cFLetter, "%c", sCur[0]);
    strcat(sNewBegin, cFLetter);
    debugprint(sNewBegin, sCur+1);
}
}
}

void main() {
    char s[255];
    char sIn[255];
    printf("\nEnter a string:");
    scanf("%s%c", sIn);
    memset(s, 0, 255);
    PrintPermu(s, sIn);
}

```

Q.11. What is the difference between a string copy (strcpy) and a memory copy (memcpy)? When should each be used?

Ans.: The strcpy() function is designed to work exclusively with strings. It copies each byte of the source string to the destination string and stops when the terminating null character () has been moved. On the other hand, the memcpy() function is designed to work with any type of data. Because not all data ends with a null character, you must provide the memcpy() function with the number of bytes you want to copy from the source to the destination.

Q.12. How can I convert a String to a Number?

Ans. : The standard C library provides several functions for converting strings to numbers of all formats (integers, longs, floats, and so on) and vice versa.

The following functions can be used to convert strings to numbers :

Function Name	Purpose
atof()	Converts a string to a double-precision floating-point value.
atoi()	Converts a string to an integer.
atol()	Converts a string to a long integer.
strtod()	Converts a string to a double-precision floating-point value and reports any leftover numbers that could not be converted.
strtol()	Converts a string to a long integer and reports any leftover numbers that could not be converted.
strtoul()	Converts a string to an unsigned long integer and reports any leftover numbers that could not be converted.

Q13. How can I convert a Number to a String?

Ans.: The standard C library provides several functions for converting numbers of all formats (integers, longs, floats, and so on) to strings and vice versa. The following functions can be used to convert integers to strings :

Function Name	Purpose
itoa()	Converts an integer value to a string.
ltoa()	Converts a long integer value to a string.
ultoa()	Converts an unsigned long integer value to a string.

The following functions can be used to convert floating-point values to strings :

Function Name	Purpose
ecvt()	Converts a double-precision floating-point value to a string without an embedded decimal point.

fcvt()	Same as ecvt(), but forces the precision to a specified number of digits.
gcvt()	Converts a double-precision floating-point value to a string with an embedded decimal point.

Q.14. Write a Program to interchange two variables without using the third one.

Ans.: a=7;

b=2;

a = a + b;

b = a - b;

a = a - b;

Q.15. How will you define String Processing along with the different String Operations?

Ans.: String Processing (Storing Strings and String Operations) : In C, a string is stored as a null-terminated char array. This means that after the last truly usable char there is a null, hex 00, which is represented in C by '\0'. The subscripts used for the array start with zero (0). The following line declares a char array called *str*. C provides fifteen consecutive bytes of memory. Only the first fourteen bytes are usable for character storage, because one must be used for the string-terminating null.

```
char str[15];
```

The following is a representation of what would be in RAM, if the string "Hello, world!" is stored in this array.

Characters: H e l l o , w o r l d !

Hex values: 48 65 6C 6C 6F 2C 20 77 6F 71 6C 64 21 00

Subscripts: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

The name of the array is treated as a pointer to the array. The subscript serves as the offset into the array, i.e., the number of bytes from the starting memory location of the array. Thus, both of the following will save the address of the 0th character in the pointer variable *ptr*.

```
ptr = str;  
ptr = &str[0];
```

strlen()

Syntax : `len = strlen(ptr);`

where `len` is an integer and `ptr` is a pointer to `char`

`strlen()` returns the length of a string, excluding the null. The following code will result in `len` having the value 13.

```
int len;  
char str[15];  
strcpy(str, "Hello, world!");  
len = strlen(str);
```

strcpy()

Syntax: `strcpy(ptr1, ptr2);`

where `ptr1` and `ptr2` are pointers to `char`

`strcpy()` is used to copy a null-terminated string into a variable. Given the following declarations, several things are possible.

```
char S[25];
```

```
char D[25];
```

- Putting text into a string:
`strcpy(S, "This is String 1.");`
- Copying a whole string from `S` to `D`:
`strcpy(D, S);`
- Copying the tail end of string `S` to `D`:
`strcpy(D, &S[8]);`

Strncpy()

Syntax: `strncpy(ptr1, ptr2, n);`

where `n` is an integer and `ptr1` and `ptr2` are pointers to `char`

`strncpy()` is used to copy a portion of a possibly null-terminated string into a variable. Care must be taken because the `'\0'` is put at the end of destination string *only* if it is within the part of the string being copied. Given the following declarations, several things are possible.

```
char S[25];
```

```
char D[25];
```

Assume that the following statement has been executed before each of the remaining code fragments.

- Putting text into the source string:
 - `strcpy(S, "This is String 1.");`
 - Copying four characters from the beginning of S to D and placing a null at the end:
 - `strncpy(D, S, 4);`
 - `D[4] = '\0';`
 - Copying two characters from the middle of string S to D:
 - `strncpy(D, &S[5], 2);`
 - `D[2] = '\0';`
 - Copying the tail end of string S to D:
 - `strncpy(D, &S[8], 15);`
 - which produces the same result as `strcpy(D, &S[8]);`
- strcat()**

Syntax: `strcat(ptr1, ptr2);`

where `ptr1` and `ptr2` are pointers to char

`strcat()` is used to concatenate a null-terminated string to end of another string variable. This is equivalent to pasting one string onto the end of another, overwriting the null terminator. There is only one common use for `strcat()`.

```
char S[25] = "world!";
```

```
char D[25] = "Hello, ";
```

- Concatenating the whole string S onto D:

- `strcat(D, S);`

strncat()

Syntax: `strncat(ptr1, ptr2, n);`

where `n` is an integer and `ptr1` and `ptr2` are pointers to `char`

`strncat()` is used to concatenate a portion of a possibly null-terminated string onto the end of another string variable. Care must be taken because some earlier implementations of C do *not* append the `'\0'` at the end of destination string. Given the following declarations, several things are possible, but only one is commonly used.

```
char S[25] = "world!";
```

```
char D[25] = "Hello, ";
```

- Concatenating five characters from the beginning of `S` onto the end of `D` and placing a null at the end:
- `strncat(D, S, 5);`
- `strncat(D, S, strlen(S) - 1);`
- Both would result in `D` containing "Hello, world".

N.B. If you fail to ensure that the source string is null-terminated, very strange and sometimes very ugly things may result.

strcmp()

Syntax : `diff = strcmp(ptr1, ptr2);`

where `diff` is an integer and `ptr1` and `ptr2` are pointers to `char`

`strcmp()` is used to compare two strings. The strings are compared character by character starting at the characters pointed at by the two pointers. If the strings are identical, the integer value zero (0) is returned. As soon as a difference is found, the comparison is halted and if the ASCII value at the point of difference in the first string is less than that in the second (e.g. `'a' 0x61` vs. `'e' 0x65`) a negative value is returned; otherwise, a positive value is returned. Examine the following examples.

```
char s1[25] = "pat";
```

```
char s2[25] = "pet";
```

diff will have a *negative* value after the following statement is executed.

```
diff = strcmp(s1, s2);
```

diff will have a *positive* value after the following statement is executed.

```
diff = strcmp(s2, s1);
```

diff will have a value of *zero* (0) after the execution of the following statement, which compares *s1* with itself.

```
diff = strcmp(s1, s1);
```

strncmp()

Syntax: `diff = strncmp(ptr1, ptr2, n);`

where *diff* and *n* are integers *ptr1* and *ptr2* are pointers to `char`

`strncmp()` is used to compare the first *n* characters of two strings. The strings are compared character by character starting at the characters pointed at by the two pointers. If the first *n* strings are identical, the integer value zero (0) is returned. As soon as a difference is found, the comparison is halted and if the ASCII value at the point of difference in the first string is less than that in the second (e.g. 'a' 0x61 vs. 'e' 0x65) a negative value is returned; otherwise, a positive value is returned. Examine the following examples.

```
char s1[25] = "pat";
```

```
char s2[25] = "pet";
```

diff will have a *negative* value after the following statement is executed.

```
diff = strncmp(s1, s2, 2);
```

diff will have a *positive* value after the following statement is executed.

```
diff = strncmp(s2, s1, 3);
```

diff will have a value of *zero* (0) after the following statement.

```
diff = strncmp(s1, s2, 1);
```

Q.16. How can we replace single character in a String?

Ans.: Single characters can be replaced in a string. Given the following declarations, several things are possible.

```
char str[25] = "cot";
```

```
char ch = 'u';
```

```
char D[25] = "pat";
```

- Replacing a single character using a char variable:

- `D[1] = ch;`

This would result in D containing "put".

- Replacing a single character using a char literal:

- `D[1] = 'e';`

This would result in D containing "pet".

- Replacing a single character using a single character from a string variable:

- `D[1] = str[1];`

This would result in D containing "pot".

Q.17. What do you mean by Pattern Matching?

Ans.: In [computer science](#), **pattern matching** is the act of checking for the presence of the constituents of a given [pattern](#). In contrast to [pattern recognition](#), the pattern is rigidly specified. Such a pattern concerns conventionally either [sequences](#) or [tree structures](#). Pattern matching is used to test whether things have a desired structure, to find relevant structure, to retrieve the aligning parts, and to substitute the matching part with something else. Sequence (or specifically text string) patterns are often described using [regular expressions](#) (i.e. [backtracking](#)) and matched using respective algorithms. Sequences can also be seen as trees branching for each element into the respective element and the rest of the sequence, or as trees that immediately branch into all elements.

Chapter-5

C++ : Classes and Objects

Q.1. What is C++?

Ans.: Released in 1985, C++ is an object-oriented programming language created by Bjarne Stroustrup. C++ maintains almost all aspects of the C language, while simplifying memory management and adding several features - including a new datatype known as a class (you will learn more about these later) - to allow object-oriented programming. C++ maintains the features of C which allowed for low-level memory access but also gives the programmer new tools to simplify memory management.

C++ used for : C++ is a powerful general-purpose programming language. It can be used to create small programs or large applications. It can be used to make CGI scripts or console-only DOS programs. C++ allows you to create programs to do almost anything you need to do. The creator of C++, Bjarne Stroustrup, has put together a partial list of applications written in C++.

Q.2. What is difference between C & C++?

Ans.:

- C does not have a class/object concept.
- C++ provides data abstraction, data encapsulation, Inheritance and Polymorphism.
- C++ supports all C syntax.
- In C passing value to a function is "Call by Value" whereas in C++ its "Call by Reference".
- File extension is .c in C while .cpp in C++. (C++ compiler compiles the files with .c extension but C compiler can not!).

- In C structures can not have contain functions declarations. In C++ structures are like classes, so declaring functions is legal and allowed.
- C++ can have inline/virtual functions for the classes.
- c++ is C with Classes hence C++ while in c the closest u can get to an User defined data type is struct and union.

Q.3. What is an Object?

Ans.: Object is a software bundle of variables and related methods. Objects have state and behavior.

Q.4. What is a Class?

Ans.: Class is a user-defined data type in C++. It can be created to solve a particular kind of problem. After creation the user need not know the specifics of the working of a class.

Q.5. What is the difference between Class and Structure?

Ans.: **Structure :** Initially (in C) a structure was used to bundle different type of data types together to perform a particular functionality. But C++ extended the structure to contain functions also. The major difference is that all declarations inside a structure are by default public.

Class : Class is a successor of Structure. By default all the members inside the class are private.

Q.6. What is the difference between an Object and a Class?

Ans.: Classes and objects are separate but related concepts. Every object belongs to a class and every class contains one or more related objects.

- A Class is static. All of the attributes of a class are fixed before, during, and after the execution of a program. The attributes of a class don't change.
- The class to which an object belongs is also (usually) static. If a particular object belongs to a certain class at the time that it is created then it almost certainly will still belong to that class right up until the time that it is destroyed.

- An Object on the other hand has a limited lifespan. Objects are created and eventually destroyed. Also during that lifetime, the attributes of the object may undergo significant change.

Q.7. What is Virtual Class and Friend Class?

Ans.: Friend classes are used when two or more classes are designed to work together and need access to each other's implementation in ways that the rest of the world shouldn't be allowed to have. In other words, they help keep private things private. For instance, it may be desirable for class DatabaseCursor to have more privilege to the internals of class Database than main() has.

Virtual class is used for run time polymorphism when object is linked to procedure call at run time. or is an inner class that can be overridden by derived classes of the outer class.

Q.8. What is the word you will use when defining a function in base class to allow this function to be a Polymorphic Function?

Ans.: Virtual

Q.9. What do you mean by Binding of Data and Functions?

Ans.: Encapsulation.

Q.10. What is Friend Function?

Ans.: As the name suggests, the function acts as a friend to a class. As a friend of a class, it can access its private and protected members. A friend function is not a member of the class. But it must be listed in the class definition.

Q.11. What is a Scope Resolution Operator?

Ans.: A scope resolution operator (::), can be used to define the member functions of a class outside the class. Example of SRO

```
e.g.  
....  
....  
{  
    int x = 10;  
    ....  
    ....  
    {  
        int x = 20;  
        .....  
        ....  
    }  
    ....  
}
```

The declaration of the inner block hides the declaration of same variable in outer block. This means, within the inner block, the variable x will refer to the data object declared therein. To access the global version of the variable, C++ provides scope resolution operator.

Q.12. What do you mean by Inheritance?

Ans.: Inheritance is the process of creating new classes, called derived classes, from existing classes or base classes. The derived class inherits all the capabilities of the base class, but can add embellishments and refinements of its own.

Q.13. What are the advantages of Inheritance?

Ans.: It permits code reusability. Reusability saves time in program development. It encourages the reuse of proven and debugged high-quality software, thus reducing problem after a system becomes functional.

Q.14. Does c++ support Multilevel and Multiple Inheritance?

Ans.: Yes.

Q.15. What do you mean by Inline Function?

Ans.: The idea behind inline functions is to insert the code of a called function at the point where the function is called. If done carefully, this can improve the application's performance in exchange for increased compile time and possibly (but not always) an increase in the size of the generated binary executables. Example:

Q.16. What are Templates?

Ans.1. C++ Templates allow u to generate families of functions or classes that can operate on a variety of different data types, freeing you from the need to create a separate function or class for each type. Using templates, u have the convenience of writing a single generic function or class definition, which the compiler automatically translates into a specific version of the function or class, for each of the different data types that your program actually uses. Many data structures and algorithms can be defined independently of the type of data they work with. You can increase the amount of shared code by separating data-dependent portions from data-independent portions, and templates were introduced to help you do that.

Ans.2. Templates allow to create generic functions that admit any data type as parameters and return value without having to overload the function with all the possible data types. Until certain point they fulfill the functionality of a macro. Its prototype is any of the two following ones :

```
template <class indetifier> function_declaration; template <typename  
indetifier> function_declaration;
```

The only difference between both prototypes is the use of keyword class or typename, its use is indistinct since both expressions have exactly the same meaning and behave exactly the same way.

Q.17. What is the difference between Declaration and Definition?

Ans.: The declaration tells the compiler that at some later point we plan to present the definition of this declaration.

E.g.: void stars () //function declaration

The definition contains the actual implementation.

E.g.: void stars () // declarator

```
{  
for(int j=10; j > =0; j--) //function body  
cout << *;  
cout << endl; }
```

Q.18. What is Operator Overloading?

Ans.: When an operator is overloaded, it takes on an additional meaning relative to a certain class. But it can still retain all of its old meanings.

Examples :

- 1) The operators >> and << may be used for I/O operations because in the header, they are overloaded.
- 2) In a stack class it is possible to overload the + operator so that it appends the contents of one stack to the contents of another. But the + operator still retains its original meaning relative to other types of data.

Q.19. What is Function Overloading and Operator Overloading?

Ans.: Function overloading: C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters (at least as far as their types are concerned). This capability is called function overloading. When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call. Function overloading is commonly used to create several functions of the same name that perform similar tasks but on different data types.

Operator overloading allows existing C++ operators to be redefined so that they work on objects of user-defined classes. Overloaded operators are syntactic sugar for equivalent function calls. They form a pleasant facade that doesn't add anything fundamental to the language (but they can improve understandability and reduce maintenance costs).

Q.20. Define a Constructor. What it is and how it might be called (2 methods)?

Ans.1: constructor is a member function of the class, with the name of the function being the same as the class name. It also specifies how the object should be initialized.

Ways of calling constructor :

- 1) Implicitly : automatically by compiler when an object is created.
- 2) Calling the constructors explicitly is possible, but it makes the code unverifiable.

Example class Point2D{

int x; int y;

```
public Point2D() : x(0) , y(0) {} // default (no argument) constructor  
};
```

```
main(){
```

Point2D MyPoint; // Implicit Constructor call. In order to allocate memory on stack, the default constructor is implicitly called.

Point2D * pPoint = new Point2D(); // Explicit Constructor call. In order to allocate memory on HEAP we call the default constructor.

Q.21. What are Virtual Constructors/Destructors?

Ans.1: Virtual Destructors : If an object (with a non-virtual destructor) is destroyed explicitly by applying the delete operator to a base-class pointer to the object, the base-class destructor function (matching the pointer type) is called on the object.

There is a simple solution to this problem declare a virtual base-class destructor.

This makes all derived-class destructors virtual even though they don't have the same name as the base-class destructor. Now, if the object in the hierarchy is destroyed explicitly by applying the delete operator to a base-class pointer to a derived-class object, the destructor for the appropriate class is called.

Virtual constructor: Constructors cannot be virtual. Declaring a constructor as a virtual function is a syntax error.

Ans.2: Virtual Destructors : If an object (with a non-virtual destructor) is destroyed explicitly by applying the delete operator to a base-class pointer to the object, the base-class destructor function (matching the pointer type) is called on the object.

There is a simple solution to this problem - declare a virtual base-class destructor. This makes all derived-class destructors virtual even though they don't have the same name as the base-class destructor. Now, if the object in the hierarchy is destroyed explicitly by applying the delete operator to a base-class pointer to a derived-class object, the destructor for the appropriate class is called.

Virtual Constructor : Constructors cannot be virtual. Declaring a constructor as a virtual function is a syntax error.

Q.22. What is Encapsulation?

Ans.: Packaging an object's variables within its methods is called encapsulation.

Q.23. Explain term Polymorphism and give an example using eg. SHAPE object. If I have a base class SHAPE, how would I define DRAW methods for two objects CIRCLE and SQUARE?

Ans.1: Polymorphism : 'Polymorphism' is an object oriented term. Polymorphism may be defined as the ability of related objects to respond to the same message with different, but appropriate actions. In other words, polymorphism means taking more than one form. Polymorphism leads to two important aspects in Object Oriented terminology - Function Overloading and Function Overriding. Overloading is the practice of supplying more than one definition for a given function name in the same scope. The compiler is left to pick the appropriate version of the function or operator based on the arguments with which it is called. Overriding refers to the modifications made in the sub class to the inherited methods from the base class to change their behavior.

In other words Polymorphism is a phenomenon which enables an object to react differently to the same function call.

In C++ it is attained by using a keyword virtual.

Example :

```
public class SHAPE
{
    public virtual void SHAPE::DRAW()=0;
}
```

Note here the function DRAW() is pure virtual which means the sub classes must implement the DRAW() method and SHAPE cannot be instantiated

```
public class CIRCLE::public SHAPE
{
    public void CIRCLE::DRAW()
    {
        // TODO drawing circle
    }
}
```

```
public class SQUARE::public SHAPE
{
    public void SQUARE::DRAW()
    {
        // TODO drawing square
    }
}
```

now from the user class the calls would be like

globally

```
SHAPE *newShape;
```

When user action is to draw

```
public void MENU::OnClickDrawCircle(){
```

```
newShape = new CIRCLE();
```

```
}
```

```
public void MENU::OnClickDrawCircle(){
```

```
newShape = new SQUARE();
```

```
}
```

the when user actually draws

```
public void CANVAS::OnMouseOperations(){
```

```
newShape->DRAW();
```

```
}
```

Ans.3: class SHAPE{
 public virtual Draw() = 0; // abstract class with a pure virtual method
};
 class CIRCLE{
 public int r;
 public virtual Draw() { this->drawCircle(0,0,r); }
};
 class SQUIRE
 public int a;

```
public virtual Draw() { this->drawRectangular(0,0,a,a); }
};
```

Each object is driven down from SHAPE implementing Draw() function in its own way.

Q.24. Describe Private, Protected and Public - the differences and give examples.

Ans.1: Public, protected and private are three access specifier in C++.

Public data members and member functions are accessible outside the class.

Protected data members and member functions are only available to derived classes.

Private data members and member functions can't be accessed outside the class. However there is an exception can be using friend classes.

Write a function that swaps the values of two integers, using int* as the argument type.

```
void swap(int* a, int*b) {
int t;
t = *a;
*a = *b;
*b = t;
}
```

Ans.2: class Point2D{

```
int x; int y;
public int color;
protected bool pinned;
public Point2D() : x(0) , y(0) {} // default (no argument) constructor
};
Point2D MyPoint;
```

You cannot directly access private data members when they are declared (implicitly)

private:

```
MyPoint.x = 5; // Compiler will issue a compile ERROR
```

```
//Nor you can see them:
```

```
int x_dim = MyPoint.x; // Compiler will issue a compile ERROR
```

On the other hand, you can assign and read the public data members:

```
MyPoint.color = 255; // no problem
```

```
int col = MyPoint.color; // no problem
```

With protected data members you can read them but not write them:

```
MyPoint.pinned = true; // Compiler will issue a compile ERROR
```

```
bool isPinned = MyPoint.pinned; // no problem
```

Q.25. Write a Program that ask for user input from 5 to 9 then calculate the average.

Ans.: #include "iostream.h"

```
int main() {
```

```
int MAX = 4;
```

```
int total = 0;
```

```
int average;
```

```
int numb;
```

```
for (int i=0; i<MAX; i++) {
```

```
cout << "Please enter your input between 5 and 9: ";
```

```
cin >> numb;
```

```
while ( numb<5 || numb>9) {
```

```
cout << "Invalid input, please re-enter: ";
```

```
cin >> numb;
```

```
}
```

```
total = total + numb;
}
average = total/MAX;
cout << "The average number is: " << average << "\n";
return 0;
}
```

Q.26. Write a short code using C++ to print out all odd number from 1 to 100 using a for loop.

Ans.: `for(unsigned int i = 1; i <= 100; i++)`
`if(i & 0x00000001)`
`cout << i << "\n";`

□ □ □

Chapter-6

Linked List

Q.1. What is the difference between an Array and a List?

Ans.1: Array is collection of homogeneous elements.

List is collection of heterogeneous elements.

For **Array**, memory allocated is static and continuous.

For **List**, memory allocated is dynamic and random.

Array : User need not have to keep in track of next memory allocation.

List : User has to keep in Track of next location where memory is allocated.

For Example: Array uses direct access of stored members, list uses sequential access for members.

//With Array you have direct access to memory position 5

Object x = a[5]; // x takes directly a reference to 5th element of array

//With the list you have to cross all previous nodes in order to get the 5th node:

```
list myList;
```

```
list::iterator it;
```

```
for( it = list.begin() ; it != list.end() ; it++ )
```

```
{
```

```
if( i==5)
```

```
{
```

```
x = *it;
```

```
break;
```



```
}  
i++;  
}
```

Q.2. How can I search for data in a Linked List?

Ans.: Unfortunately, the only way to search a linked list is with a linear search, because the only way a linked list's members can be accessed is sequentially. Sometimes it is quicker to take the data from a linked list and store it in a different data structure so that searches can be more efficient.

Q.3. What is Static Memory Allocation and Dynamic Memory Allocation?

Ans.: Static Memory Allocation : The compiler allocates the required memory space for a declared variable. By using the address of operator, the reserved address is obtained and this address may be assigned to a pointer variable. Since most of the declared variables have static memory, this way of assigning pointer value to a pointer variable is known as static memory allocation. Memory is assigned during compilation time.

Dynamic Memory Allocation : It uses functions such as `malloc()` or `calloc()` to get memory dynamically. If these functions are used to get memory dynamically and the values returned by these functions are assigned to pointer variables, such assignments are known as dynamic memory allocation. Memory is assigned during run time.

Q.4. Linked List-Compilation. How to reduce a final size of executable?

Ans.: Size of the final executable can be reduced using dynamic linking for libraries.

Q.5. What is a NULL Pointer?

Ans.: There are times when it's necessary to have a pointer that doesn't point to anything. The macro `NULL`, defined in `<stddef.h>`, has a value that's guaranteed to be different from any valid pointer. `NULL` is a literal zero, possibly cast to `void*`

or `char*`. Some people, notably C++ programmers, prefer to use 0 rather than NULL.

The null pointer is used in three ways :

- 1) To stop indirection in a recursive data structure
- 2) As an error value
- 3) As a sentinel value

Q.6 Why should we assign NULL to the elements (pointer) after freeing them?

Ans.: This is paranoia based on long experience. After a pointer has been freed, you can no longer use the pointed-to data. The pointer is said to dangle; it doesn't point at anything useful. If you NULL out or zero out a pointer immediately after freeing it, your program can no longer get in trouble by using that pointer. True, you might go indirect on the null pointer instead, but that's something your debugger might be able to help you with immediately. Also, there still might be copies of the pointer that refer to the memory that has been deallocated; that's the nature of C. Zeroing out pointers after freeing them won't solve all problems.

Q.7. Is NULL always defined as 0?

Ans.: NULL is defined as either 0 or `(void*)0`. These values are almost identical; either a literal zero or a void pointer is converted automatically to any kind of pointer, as necessary, whenever a pointer is needed (although the compiler can't always tell when a pointer is needed).

Q.8. What is the difference between NULL and NUL?

Ans.: NULL is a macro defined in for the null pointer.

NUL is the name of the first character in the ASCII character set. It corresponds to a zero value. There's no standard macro NUL in C, but some people like to define it.

The digit 0 corresponds to a value of 80, decimal. Don't confuse the digit 0 with the value of " (NUL)! NULL can be defined as `((void*)0)`, NUL as `"`.

Q.9. How can I sort a Linked List?

Ans.: Both the merge sort and the radix sort are good sorting algorithms to use for linked lists.

Q.10. Tell how to check whether a Linked List is circular?

Ans.: Create two pointers, and set both to the start of the list. Update each as follows :

```
while (pointer1) {  
    pointer1 = pointer1->next;  
    pointer2 = pointer2->next;  
    if (pointer2) pointer2=pointer2->next;  
    if (pointer1 == pointer2) {  
        print ("circular");  
    }  
}
```

If a list is circular, at some point pointer2 will wrap around and be either at the item just before pointer1, or the item before that. Either way, its either 1 or 2 jumps until they meet.

□ □ □

Chapter-7

Stacks

Q.1. What is Stack?

Ans.: A **stack** is a limited version of an array. New elements, or **nodes** as they are often called, can be added to a stack and removed from a stack only from one end. For this reason, a stack is referred to as a LIFO structure (Last-In First-Out).

Stacks have many applications. For example, as processor executes a program, when a function call is made, the called function must know how to return back to the program, so the current address of program execution is pushed onto a stack. Once the function is finished, the address that was saved is removed from the stack, and execution of the program resumes. If a series of function calls occur, the successive return values are pushed onto the stack in LIFO order so that each function can return back to calling program. Stacks support recursive function calls in the same manner as conventional non recursive calls.

Stacks are also used by compilers in the process of evaluating expressions and generating machine language code. They are also used to store return addresses in a chain of method calls during execution of a program.

Q.2. How can we declare a Stack using an Array?

Ans.: **Stack Declaration Using an Array :** Suppose elements of the stack are of type int and the stack can store a maximum of 10 elements.

```
#define MAX 10
```

```
typedef struct
{
    int top;
    int elements[MAX];
}stack;
stack s;
```

- Here we have defined our own data type named stack.
- The first element "top" will be used to index the topmost element
- Array "elements" holds the elements of the stack
- The last line declares a variable "s" of type stack

Representation of Stack in Memory :

	0	1	2	3	4	5	6	7	8	9
4 top	8	10	12	-5	6					

	0	1	2	3	4	5	6	7	8	9
2 top	8	10	12							

	0	1	2	3	4	5	6	7	8	9
6 top	8	10	12	-5	6	9	55			

Q.3. Tell the different operations used in Stacks.

Ans.1: Operations : An abstract data type (ADT) consists of a data structure and a set of **primitive operations**. The main primitives of a stack are known as :

Push adds a new node

Pop removes a node

Additional primitives can be defined :

IsEmpty reports whether the stack is empty

IsFull reports whether the stack is full

Initialise creates/initialises the stack

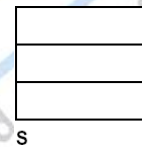
Destroy deletes the contents of the stack (may be implemented by re-initialising the stack)

Initialise creates the structure - i.e. ensures that the structure exists but contains no elements

e.g. *Initialise(S)* creates a new empty stack named S

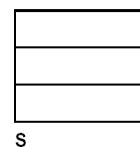
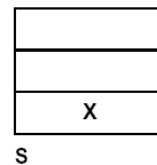
Push

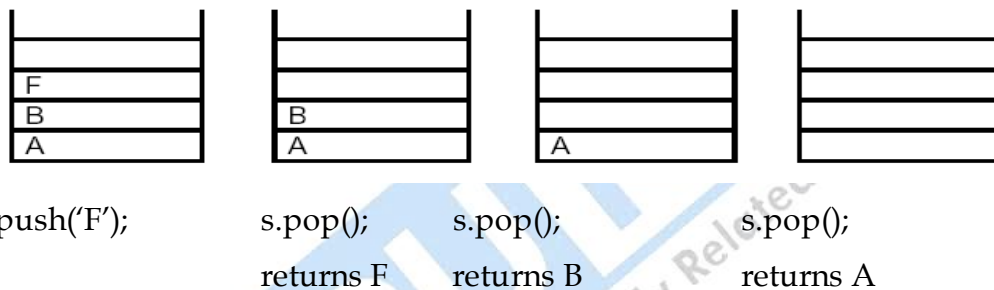
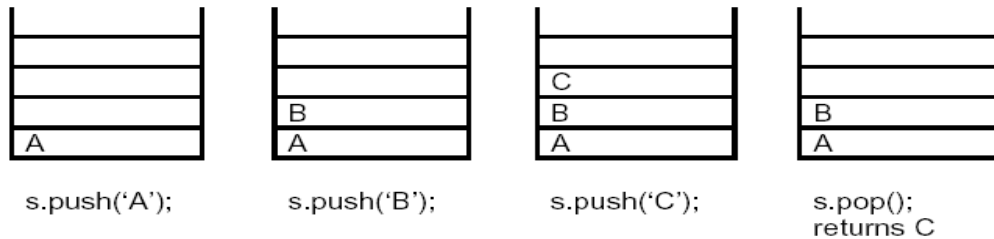
e.g. *Push(X,S)* adds the value X to the TOP of stack S



Pop

e.g. *Pop(S)* removes the TOP node and returns its value



Example

We could try the same example with actual values for A, B and C.

A = 1 B = 2 C = 3

Ans.2: The STACK Data Structure :**Exercise : Stack Operations**

- What would the state of a stack be after the following operations :
 - create stack
 - push A onto stack
 - push F onto stack
 - push X onto stack
 - pop item from stack
 - push B onto stack
 - pop item from stack
 - pop item from stack

2. Show the state of the stack and the value of each variable after execution of each of the following statements :

$A = 5$ $B = 3$ $C = 7$

- (a) create stack
push A onto stack
push $C * C$ onto stack
pop item from stack and store in B
push $B + A$ onto stack
pop item from stack and store in A
pop item from stack and store in B
- (b) create stack
push B onto stack
push C onto stack
push A onto stack
 $A = B * C$
push $A + C$ onto stack
pop item from stack and store in A
pop item from stack and store in B
pop item from stack and store in C

Stack Implementation : The Java Collections Framework includes a set of ready made data structure classes, including a *Stack* class. However, you will create your own stack class in order to learn how a stack is implemented. Your class will be a bit simpler than the Collections Framework one but it will do essentially the same job.

Q.4. Differentiate between Static and Dynamic Data Structures.

Ans.: Static and Dynamic Data Structures :

A stack can be stored in :

- a static data structure

OR

- a dynamic data structure

Static Data Structures : These define collections of data which are fixed in size when the program is compiled.

An **array** is a static data structure.

Dynamic data structures : These define collections of data which are variable in size and structure. They are created as the program executes, and grow and shrink to accommodate the data being stored.

Q.5. Define Stacks & Lists in terms of implementing algorithms with the various operations.

Ans.: Stacks & Lists : **Stacks** and **Lists** are basic data structures that you need to know about when implementing algorithms.

A stack can be regarded as an array of data elements, that needs to be filled. There are 2 different kinds of stacks. FIFO stack and LIFO stack.

FIFO is short for First in, First out . That is, the first element that is added to the array will also be the first to be removed.

LIFO is short for Last in, First out . That is, the last element being added to the array will be the first to be removed.

The Stack Pointer . Another important thing is, that all stacks, be it FIFO or LIFO have a so called stack pointer. The stack pointer marks the last element of the array.

Push and Pop : Push and pop are the basic operations executed on a stack.

Push means, to add another item to the stack. **Pop** means, to remove another item from the stack.

Push and **Pop** are usually implemented as functions but they could also be implemented as a loop in the main program.

Implementing a LIFO stack. Now comes the fun. We are going to implement a LIFO stack. Before starting coding we may need gather our basic informations about the stack.

Since its a LIFO stack, we may implement it as an array. Remember that LIFO means Last in, First out, so we need to assure that the last element being added will be the first to be released. This is a rather straightforward operation, since all arrays usually grow from bottom to top and we can simply remove the last element being added.

```
/* allocating an array with 10 elemets used as stack */
int* stack;
int StackInit(int i)
{
    stack = malloc(i*(sizeof(int)));
    return stack;
}
```

In this code snippet, the stack is defined as global variable, because it needs to be accessible to all other functions.

```
void push(int elem)
{
    stack[++top] = elem;
}
int pop(int elem)
{
    return stack[elem--];
}
```

The code above shows possible implementations of **push** and **pop** . The implementation is hold easy to demonstrate their functionality. In a real-world program you would have to check if the stack is empty or already filled, if the stack is full etc.

Push and **pop** could than be used as follows :

```
int main()
{
```

```
int j = 10;
int l;
/* allocating a stack */
stackinit(j);
/* fill the stack with 10 items */
for(l=0;l<=10;l++){
    push(l);
    /* we have 10 items in the stack, so quit */
    if(l >= 10)
    {
        break;
        exit(1);
    }
}
/* pop out all items of the stack */
int k;
for(k = 10;k>=0;k--){
    pop(k);
    if(k <= 0)
    {
        break;
        exit(1);
    }
}
return 0;
}
```

This program is rather simple. First it allocates space for the stack and then it uses push and pop operations to fill and empty the stack.

Lists Lists are another basic and popular data structure. That's why some High Level programming languages like C++ and Java have implemented them in their libraries. In C you need to build your own List. This is usually done with a struct.

```
typedef struct LinkedList
{
    struct LinkedList* head;
    struct LinkedList* next;
};
struct LinkedList list;
```

Having defined this list as a new data type you can use it like you would use any other elementary data type of C.

Basic List operations: Basic List operations are: **insert an item**, **delete an item**, **move an item**. However, before doing any operation on lists, you need to allocate memory for the list.

```
struct* LinkedList InitList(struct LinkedList* p)
{
    if(p =
        malloc(x*sizeof(LinkedList))
        == NULL)
        printf("Error! Unable to allocate memory");
    else
    {
        p = p->next;
        p->next = NULL;
```



```
    }  
    return p;  
}
```

This function is rather simple. It allocates memory for the list dynamically as we have done with the stack. But it does a little bit more. It also sets the pointer `p` as the one and only element, that is, it is the first and the last element of the list.

Note : This code may not work when compiled with ANSI-C conformity because ANSI-C does not allow use of reference operator on the left hand side of an assignment .

□ □ □

Chapter-8

Queue

Q.1. What is Queue?

Ans.: In general, a queue is a line of people or things waiting to be handled, usually in sequential order starting at the beginning or top of the line or sequence. In computer technology, a queue is a sequence of work objects that are waiting to be processed. The possible factors, arrangements, and processes related to queues is known as [queueing theory](#).

Queue is a linear data structure in which data can be added to one end and retrieved from the other. Just like the queue of the real world, the data that goes first into the queue is the first one to be retrieved. That is why queues are sometimes called as **First-In-First-Out** data structure.

In case of queues, we saw that data is inserted both from one end but in case of Queues; data is added to one end (known as REAR) and retrieved from the other end (known as FRONT).

The data first added is the first one to be retrieved while in case of queues the data last added is the first one to be retrieved.

A few points regarding Queues:

Queues: It is a linear data structure; linked lists and arrays can represent it. Although representing queues with arrays have its shortcomings but due to simplicity, we will be representing queues with arrays in this article.

Rear: A variable stores the index number in the array at which the new data will be added (in the queue).

Front: It is a variable storing the index number in the array where the data will be retrieved.

Let us have look at the process of adding and retrieving data in the queue with the help of an example.

Suppose we have a queue represented by an array queue [10], which is empty to start with. The values of front and rear variable upon different actions are mentioned in {}.

queue [10]=EMPTY {front=-1, rear=0}

add (5)

Now, queue [10] = 5 {front=0, rear=1}

add (10)

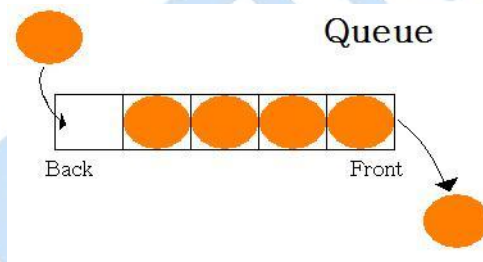
Now, queue [10] = 5, 10 {front=0, rear=2}

retrieve () [It returns 5]

Now, queue [10] = 10 {front=1, rear=2}

retrieve () [now it returns 10]

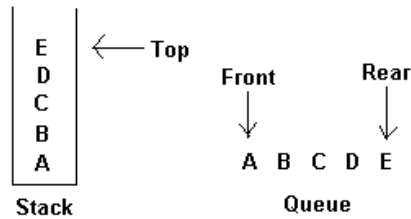
Now, queue [10] is again empty {front=-1, rear=-1}



Q.2. Compare Stacks with Queue.

Ans.: In [programming](#), a queue is a [data structure](#) in which elements are removed in the same order they were entered. This is often referred to as FIFO (first in, first out). In contrast, a [stack](#) is a data structure in which elements are removed in the reverse order from which they were entered. This is referred to as LIFO (last in, first out).

A stack is an ordered list in which all insertions and deletions are made at one end, called the top. A queue is an ordered list in which all insertions take place at one end, the rear, while all deletions take place at the other end, the front. Given a stack $S=(a[1], a[2], \dots, a[n])$ then we say that $a[1]$ is the bottommost element and element $a[i]$ is on top of element $a[i-1]$, $1 < i \leq n$. When viewed as a queue with $a[n]$ as the rear element one says that $a[i+1]$ is behind $a[i]$, $1 < i \leq n$.



The restrictions on a stack imply that if the elements A,B,C,D,E are added to the stack, in that order, then the first element to be removed/ deleted must be E. Equivalently we say that the last element to be inserted into the stack will be the first to be removed. For this reason stacks are sometimes referred to as Last In First Out (LIFO) lists. The restrictions on queue imply that the first element which is inserted into the queue will be the first one to be removed. Thus A is the first letter to be removed, and queues are known as First In First Out (FIFO) lists. Note that the data object queue as defined here need not necessarily correspond to the mathematical concept of queue in which the insert/ delete rules may be different.

Q.3. Describe the different operations on Queue.

Ans.:

Queue constructor	construct a new queue
back	returns a reference to last element of a queue
empty	true if the queue has no elements
front	returns a reference to the first element of a queue
pop	removes the first element of a queue
push	adds an element to the end of the queue
size	returns the number of items in the queue

Priority Queue :

A Priority queue is a collection of zero or more elements. Each element has a **priority** or a value. The operations performed on a priority queue are :

- 1) Find an element
- 2) Insert a new element
- 3) Delete an element

In "**Min Priority Queue**" the find operation finds the element with minimum priority, while the delete operation deletes it.

In "**Max Priority Queue**" the find operation finds the element with maximum priority, while the delete operation deletes it.

Unlike the general queues which are FIFO structures , the order of deletion from a priority queue is determined by the element priority. Elements are deleted either in increasing or decreasing order of priority from a priority queue.

A priority queue can be **implemented by using**

- 1) **Heap**
- 2) **Height and Weight balanced leftist trees.**

```
#include<iostream.h>

class searching
{
private:
    double *array;
    int n;
public:
    void input();
    void bubblesort();
    void binarysearch();
};

void searching::input()
{
    cout<<"*****\n"
    <<"This program is to implement binary search algorithm\n"
    <<"*****\n";
    cout<<"Enter how many numbers you are going to enter::";
```

```
cin>>n;
array=new double[n+1];
cout<<"Now enter your elements ::\n";
for(int i=1;i<=n;i++)
    cin>>array[i];
}
void searching::bubblesort()
{
    for(int i=1;i<=n-1;i++)
    {
        for(int j=1;j<=n-i;j++)
            if(array[j]>=array[j+1])
                array[j]+=array[j+1],
                array[j+1]=array[j]-array[j+1],
                array[j]=array[j]-array[j+1];
    }
}
void searching::binarysearch()
{
    cout<<"Enter the element to be searched ::";
    double x;
    cin>>x;
    int low=1,high=n;
    while(low<=high)
    {
        int middle=(low+high)/2;
        if(x<array[middle])
```



```

        high=middle-1;
    else if(x>array[middle])
        low=middle+1;
    else
    {
        cout<<"found \n";
        return;
    }
    cout<<"search unsuccessful\n";
}
int main()
{
    searching obj;
    obj.input();
    obj.bubblesort();
    obj.binarysearch();
    return 0;
}
/*****

```

SAMPLE OUTPUT ::

This program is to implement binary search algorithm

Enter how many numbers you are going to enter::5

Now enter your elements ::

1.3

1.2

1.6

1.5

1.4

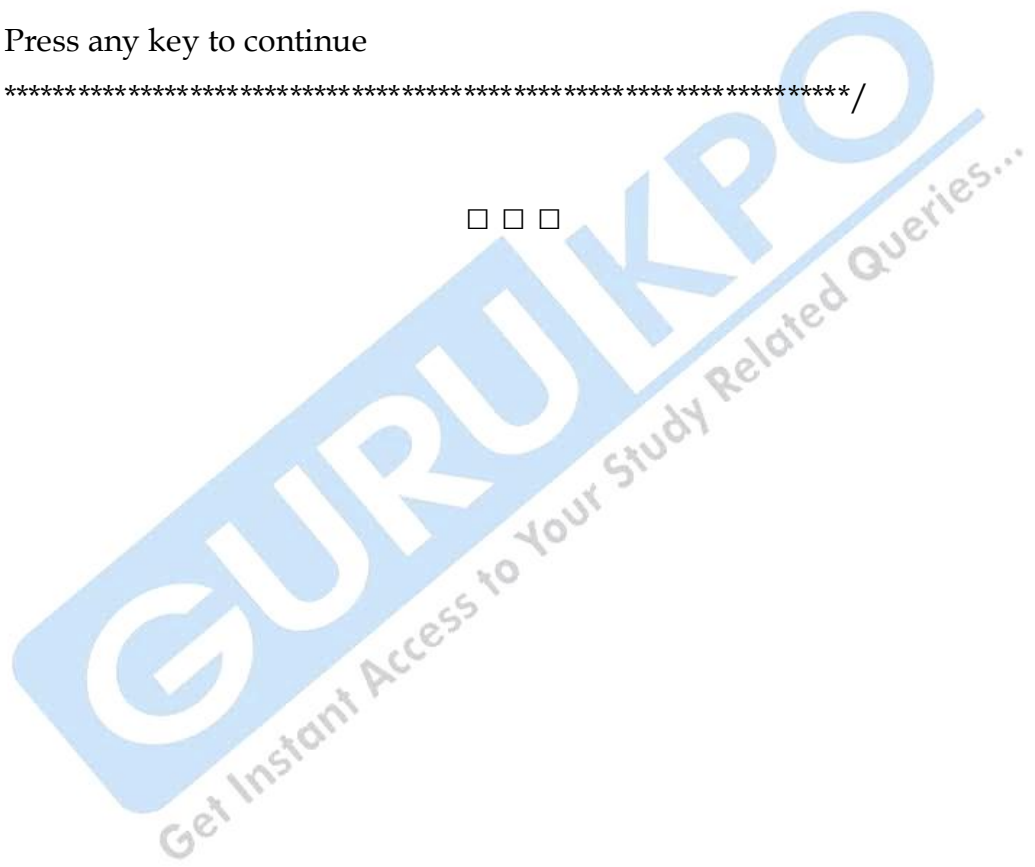
Enter the element to be searched ::1.4

found

Press any key to continue

***** /

□ □ □



Chapter-9

Sorting & Searching Techniques

Q.1. Explain the concept of Bubble Sort along with Algorithm.

Ans.: Bubble sort is a simple [sorting algorithm](#). It works by repeatedly stepping through the list to be sorted, comparing two items at a time and [swapping](#) them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a [comparison sort](#).

Step-by-Step Example : Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort algorithm. In each step, elements written in **bold** are being compared.

First Pass :

(**5** **1** 4 2 8) (**1** **5** 4 2 8) Here, algorithm compares the first two elements, and swaps them.

(**1** **5** 4 2 8) (**1** **4** **5** 2 8)

(**1** **4** **5** 2 8) (**1** **4** **2** **5** 8)

(**1** **4** **2** **5** 8) (**1** **4** **2** **5** 8) Now, since these elements are already in order, algorithm does not swap them.

Second Pass :

(**1** **4** 2 5 8) (**1** **4** 2 5 8)

(**1** **4** 2 5 8) (**1** **2** **4** 5 8)

(**1** **2** **4** 5 8) (**1** **2** **4** 5 8)

(**1** **2** **4** **5** 8) (**1** **2** **4** **5** 8)

Now, the array is already sorted, but our algorithm does not know if it is completed.

Algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass :

(1 2 4 5 8) (1 2 4 5 8)

(1 2 4 5 8) (1 2 4 5 8)

(1 2 4 5 8) (1 2 4 5 8)

(1 2 4 5 8) (1 2 4 5 8)

Finally, the array is sorted, and the algorithm can terminate.

procedure bubbleSort(A : list of sortable items) **defined as:**

for each i **in** 1 **to** length(A) **do:**

for each j **in** length(A) **downto** i + 1 **do:**

if A[j - 1] > A[j] **then**

 swap(A[j - 1], A[j])

end if

end for

end for

end procedure

Q.2. Explain the concept of Selection Sort along with Algorithm.

Ans.: The algorithm works as follows :

- (1) Find the minimum value in the list.
- (2) Swap it with the value in the first position.
- (3) Repeat the steps above for remainder of the list (starting at the second position).

Effectively, we divide the list into two parts: the sublist of items already sorted, which we build up from left to right and is found at the beginning, and the sublist of items remaining to be sorted, occupying the remainder of the array.

Here is an example of this sort algorithm sorting five elements :

64 25 12 22 11

11 25 12 22 64

11 12 25 22 64

11 12 22 25 64

Selection sort can also be used on list structures that make add and remove efficient, such as a [linked list](#). In this case it's more common to *remove* the minimum element from the remainder of the list, and then *insert* it at the end of the values sorted so far. For example :

64 25 12 22 11

11 64 25 12 22

11 12 64 25 22

11 12 22 64 25

11 12 22 25 64

Pseudo-code :

A is the set of elements to sort, **n** is the number of elements in **A** (the array starts at index 0)

```

for i ← 0 to n-2 do
    min ← i
    for j ← (i + 1) to n-1 do
        if A[j] < A[min]
            min ← j
    swap A[i] and A[min]

```

Q.3. Explain the Algorithm of Insertion Sort?

Ans.: **Insertion sort** is a simple sorting algorithm, a comparison sort in which the sorted array (or list) is built one entry at a time. It is much less efficient on large lists than more advanced algorithms such as quick sort, heap sort, or merge sort, but it has various advantages :

- (1) Simple to implement.
- (2) Efficient on (quite) small data sets.
- (3) Efficient on data sets which are already substantially sorted: it runs in $O(n + d)$ time, where d is the number of inversions.
- (4) stable(does not change the relative order of elements with equal keys)
- (5) In- place (only requires a constant amount $O(1)$ of extra memory space)
- (6) It is an online algorithm , in that it can sort a list as it receives it.

A simple **procedure for Insertion Sort** is :

insertionSort(array A)

 for $i = 1$ to $\text{length}[A]-1$ do

 begin

 value = $A[i]$

$j = i-1$

 while $j \geq 0$ and $A[j] > \text{value}$ do

 begin

$A[j + 1] = A[j]$

$j = j-1$

 end

$A[j+1] = \text{value}$

 end

Q.4. Explain the Merge Sort?

Ans.: Conceptually, a Merge Sort works as follows :

- If the list is of length 0 or 1, then it is sorted. Otherwise;
- Divide the unsorted list into two sublists of about half the size;
- Sort each sublist recursively by re-applying merge sort;
- Merge the two sublists back into one sorted list.

In a simple pseudocode form, the algorithm could look something like this:

```
function mergesort(m)
  var list left, right, result
  if length(m) ≤ 1
    return m
  var middle = length(m) / 2
  for each x in m up to middle
    add x to left
  for each x in m after middle
    add x to right
  left = mergesort(left)
  right = mergesort(right)
  result = merge(left, right)
  return result
```

Q.5. Explain the Radix Sort?

Ans.: In [computer](#) science, **radix sort** is a [sorting algorithm](#) that sorts integers by processing individual digits. Because integers can represent strings of characters (e.g., names or dates) and specially formatted floating point numbers, radix sort is not limited to integers.

Most digital computers internally represent all of their data as electronic representations of binary numbers, so processing the digits of integer representations by groups of binary digit representations is most convenient. Two classifications of radix sorts are [least significant digit](#) (LSD) radix sorts and [most significant digit](#) (MSD) radix sorts. LSD radix sorts process the integer representations starting from the least significant digit and move towards the most significant digit. MSD radix sorts work the other way around.

The integer representations that are processed by sorting algorithms are often called "keys," which can exist all by themselves or be associated with other data. LSD radix sorts typically use the following sorting order: short keys come before longer keys, and keys of the same length are sorted lexicographically. This coincides with the normal order of integer

representations, such as the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. MSD radix sorts use lexicographic order, which is suitable for sorting strings, such as words, or fixed-length integer representations. A sequence such as "b, c, d, e, f, g, h, i, j, ba" would be lexicographically sorted as "b, ba, c, d, e, f, g, h, i, j". If lexicographic ordering is used to sort variable-length integer representations, then the representations of the numbers from 1 to 10 would be output as 1, 10, 2, 3, 4, 5, 6, 7, 8, 9, as if the shorter keys were left-justified and padded on the right with blank characters to make the shorter keys as long as the longest key for the purpose of determining sorted order.

Q.6. Explain the Quick Sorting?

Ans.: Quicksort sorts by employing a [divide and conquer](#) strategy to divide a [list](#) into two sub-lists.

The steps are :

- Pick an element, called a [pivot](#), from the list.
- Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
- [Recursively](#) sort the sub-list of lesser elements and the sub-list of greater elements.

The [base case](#) of the recursion are lists of size zero or one, which are always sorted.

In simple [pseudocode](#), the algorithm might be expressed as:

```
function quicksort(array)
    var list less, greater
    if length(array) ≤ 1
        return array
    select a pivot value pivot from array
    for each x in array
        if x < pivot then append x to less
```

```
    if x > pivot then append x to greater
return concatenate(quicksort(less), pivot, quicksort(greater))
```

Q.7. Explain the concept of Binary Search?

Ans.: A **binary search algorithm** (or **binary chop**) is a technique for finding a particular value in a [sorted list](#). It makes progressively better guesses, and closes in on the sought value by selecting the [median](#) element in a list, comparing its value to the target value, and determining if the selected value is greater than, less than, or equal to the target value. A guess that turns out to be too high becomes the new top of the list, and a guess that is too low becomes the new bottom of the list. Pursuing this strategy iteratively, it narrows the search by a factor of two each time, and finds the target value.

The algorithm : The most common application of binary search is to find a specific value in a [sorted list](#). To cast this in the frame of the guessing game (see Example below), realize that we are now guessing the *index*, or numbered place, of the value in the list. This is useful because, given the index, other data structures will contain associated information. Suppose a data structure containing the classic collection of name, address, telephone number and so forth has been accumulated, and an array is prepared containing the names, numbered from one to N . A query might be: what is the telephone number for a given name X . To answer this the array would be searched and the index (if any) corresponding to that name determined, whereupon it would be used to report the associated telephone number and so forth. Appropriate provision must be made for the name not being in the list (typically by returning an *index* value of zero), indeed the question of interest might be only whether X is in the list or not.

low = 0

high = N

```
while (low < high) {
    mid = (low + high)/2;
    if (A[mid] < value)
        low = mid + 1;
```

```
    else
        // can't be high = mid-1: here A[mid] >= value,
        // so high can't be < mid if A[mid] == value
        high = mid;
    }

    if (low < N) and (A[low] == value)
        return low
    else
        return not_found
```

This algorithm has two other advantages. At the end of the loop, *low* points to the first entry greater than or equal to *value*, so a new entry can be inserted if no match is found. Moreover, it only requires one comparison; which could be significant for complex keys in languages which do not allow the result of a comparison to be saved.

Q.8. What is Internal & External Sorting techniques?

Ans.: An **internal sort** is any data sorting process that takes place entirely within the [main memory](#) of a computer. This is possible whenever the data to be sorted is small enough to all be held in the main memory. For sorting larger datasets, it may be necessary to hold only a chunk of data in memory at a time, since it won't all fit. The rest of the data is normally held on some larger, but slower medium, like a hard-disk. Any reading or writing of data to and from this slower media can slow the sorting process considerably. This issue has implications for different [sort algorithms](#).

Consider a [Bubblesort](#), where adjacent records are swapped in order to get them into the right order, so that records appear to 'bubble' up and down through the dataspace. If this has to be done in chunks, then when we have sorted all the records in chunk 1, we move on to chunk 2, but we find that some of the records in chunk 1 need to 'bubble through' chunk 2, and vice versa (i.e., there are records in chunk 2 that belong in chunk 1, and records in chunk 1 that belong in chunk 2 or later chunks). This will cause the chunks to be read and written back to disk many times as records cross over the

boundaries between them, resulting in a considerable degradation of performance. If the data can all be held in memory as one large chunk, then this performance hit is avoided.

On the other hand, some algorithms handle [external sorting](#) rather better. A [Merge sort](#) breaks the data up into chunks, sorts the chunks by some other algorithm (maybe bubblesort or [Quick sort](#)) and then recombines the chunks two by two so that each recombined chunk is in order. This approach minimises the number of reads and writes of data-chunks from disk, and is a popular external sort method.

Q.9. What is the quickest Sorting Method to use?

Ans.: The answer depends on what you mean by quickest. For most sorting problems, it just doesn't matter how quick the sort is because it is done infrequently or other operations take significantly more time anyway. Even in cases in which sorting speed is of the essence, there is no one answer. It depends on not only the size and nature of the data, but also the likely order. No algorithm is best in all cases.

There are three sorting methods in this author's toolbox that are all very fast and that are useful in different situations. Those methods are quick sort, merge sort, and radix sort.

The Quick Sort : The quick sort algorithm is of the divide and conquer type. That means it works by reducing a sorting problem into several easier sorting problems and solving each of them. A dividing value is chosen from the input data, and the data is partitioned into three sets: elements that belong before the dividing value, the value itself, and elements that come after the dividing value. The partitioning is performed by exchanging elements that are in the first set but belong in the third with elements that are in the third set but belong in the first. Elements that are equal to the dividing element can be put in any of the three sets; the algorithm will still work properly.

The Merge Sort : The merge sort is a divide and conquer sort as well. It works by considering the data to be sorted as a sequence of already-sorted lists (in the worst case, each list is one element long). Adjacent sorted lists are merged into larger sorted lists until there is a single sorted list containing all the elements. The merge sort is good at sorting lists and other data structures that

are not in arrays, and it can be used to sort things that don't fit into memory. It also can be implemented as a stable sort.

The Radix Sort : The radix sort takes a list of integers and puts each element on a smaller list, depending on the value of its least significant byte. Then the small lists are concatenated, and the process is repeated for each more significant byte until the list is sorted. The radix sort is simpler to implement on fixed-length data such as ints.

Q.10. What is the quickest Searching Method to use?

Ans.: A binary search, such as `bsearch()` performs, is much faster than a linear search. A hashing algorithm can provide even faster searching. One particularly interesting and fast method for searching is to keep the data in a digital trie. A digital trie offers the prospect of being able to search for an item in essentially a constant amount of time, independent of how many items are in the data set.

A digital trie combines aspects of binary searching, radix searching, and hashing. The term digital trie refers to the data structure used to hold the items to be searched. It is a multilevel data structure that branches N ways at each level.

□ □ □

Chapter-10

Graph

Q.1. What is Graph?

Ans.: In [computer science](#), a graph is a kind of [data structure](#), specifically an [abstract data type](#) (ADT), that consists of a [set](#) of nodes (also called vertices) and a set of edges that establish relationships (connections) between the nodes. The graph ADT follows all connects from the [graph](#) theory of [mathematics](#).

Informally, $G=(V,E)$ consists of vertices, the elements of V , which are connected by edges, the elements of E . Formally, a graph, G , is defined as an ordered pair, $G=(V,E)$, where V is a set (usually [finite](#)) and E is a set consisting of two element subsets of V .

Q.2. How can we represent a Graph?

Ans.: Choices of Representation : Two main data structures for the representation of graphs are used in practice. The first is called an [adjacency list](#), and is implemented by representing each node as a data structure that contains a list of all adjacent nodes. The second is an [adjacency matrix](#), in which the rows and columns of a two-dimensional array represent source and destination vertices and entries in the array indicate whether an edge exists between the vertices. Adjacency lists are preferred for [sparse graphs](#); otherwise, an adjacency matrix is a good choice. Finally, for very large graphs with some regularity in the placement of edges, a [symbolic graph](#) is a possible choice of representation.

Q..3 Describe the various list structure of Graphs.

Ans.: List Structures :

Incidence List : The edges are represented by an [array](#) containing pairs (ordered if directed) of vertices (that the edge connects) and possibly weight and other data. Vertices connected by an edge are said to be adjacent.

Adjacency List : Much like the incidence list, each vertex has a list of which vertices it is adjacent to. This causes redundancy in an undirected graph: for example, if vertices A and B are adjacent, A's adjacency list contains B, while B's list contains A. Adjacency queries are faster, at the cost of extra storage space.

Matrix Structures : There are so many matrices by which we can show the sequential representation of a graph, some of them are (i) Incidence matrix and (ii) Adjacency matrix :

(i) **Incidence Matrix** : Let G be a graph with n vertices, e edges and no self loops. Define an $n \times e$ matrix $M = a_{ij}$, whose n rows corresponds to the n vertices and these columns correspond to the edges, as follows :

The matrix element

$a_{ij} = 1$ if j^{th} edge e_j is incident on i^{th} vertex v_i , and

$= 0$ otherwise

(ii) **Adjacency Matrix** : Adjacency matrix is an alternative approach to represent the graph by incidence matrix. The adjacency matrix of a graph G with n vertices and no parallel edges is an n by n symmetric binary matrix denoted by $A = [a_{ij}]$

where $a_{ij} = 1$, if there is an edge between i^{th} and j^{th} vertices and $= 0$, if there is no edge between them.

Q.4. Define Undirected , Directed and Multi Graphs.

Ans.: Undirected Graph : An [undirected graph](#) G has two kinds of incidence matrix: unoriented and oriented. The incidence matrix (or unoriented incidence matrix) of G is a $p \times q$ [matrix](#) (b_{ij}), where p and q are the numbers of [vertices](#) and [edges](#) respectively, such that $b_{ij} = 1$ if the vertex v_i and edge x_j are incident and 0 otherwise.

Multigraph : A multigraph with multiple edges (red) and a loop (blue). Not all authors allow multigraphs to have loops.

A multigraph or pseudograph is a [graph](#) which is permitted to have [multiple edges](#), (also called "parallel edges"), that is, edges that have the same end nodes. Thus two vertices may be connected by more than one edge. Formally, a multigraph G is an [ordered pair](#) $G:=(V, E)$ with

- V a [set](#) of vertices or nodes,
- E a [multiset](#) of unordered pairs of vertices, called edges or lines.

Multigraphs might be used to model the possible flight connections offered by an airline. In this case the pseudograph would be a [directed graph](#) with pairs of directed parallel edges connecting cities to show that it is possible to fly both to and from these locations.

Some authors also allow multigraphs to have [loops](#), that is, an edge that connects a vertex to itself.

A multidigraph is a [directed graph](#) which is permitted to have multiple arcs, i.e., arcs with the same source and target nodes. A multidigraph G is an ordered pair $G:=(V, A)$ with

- V a [set](#) of vertices or nodes,
- A a multiset of ordered pairs of vertices called directed edges, arcs or arrows.

A mixed multigraph $G:=(V, E, A)$ may be defined in the same way as a [mixed graph](#).

Q.5. How will you Explain Matrix Structures of Graph?

Ans.: [Incidence Matrix](#) : The graph is represented by a [matrix](#) of size $|V|$ (number of vertices) by $|E|$ (number of edges) where the entry [vertex, edge] contains the edge's endpoint data (simplest case: 1 - connected, 0 - not connected).

The incidence matrix of a [directed graph](#) D is a $p \times q$ matrix $[b_{ij}]$ where p and q are the number of vertices and edges respectively, such that $b_{ij} = -1$ if the edge x_j leaves vertex v_i , 1 if it enters vertex v_i and 0 otherwise. (Note that many authors use the opposite sign convention.)

An oriented incidence matrix of an undirected graph G is the incidence matrix, in the sense of directed graphs, of any orientation of G . That is, in the column of edge e , there is a +1 in the row corresponding to one vertex of e and a -1 in

the row corresponding to the other vertex of e , and all other rows have 0. All oriented incidence matrices of G differ only by negating some set of columns. In many uses, this is an insignificant difference, so one can speak of the oriented incidence matrix, even though that is technically incorrect.

The oriented or unoriented incidence matrix of a graph G is related to the [adjacency matrix](#) of its [line graph](#) $L(G)$ by the following theorem:

$$A(L(G)) = B(G)TB(G) - 2I_q$$

where $A(L(G))$ is the adjacency matrix of the line graph of G , $B(G)$ is the incidence matrix, and I_q is the [identity matrix](#) of dimension q .

[Adjacency Matrix](#): This is the n by n matrix A , where n is the number of vertices in the graph. If there is an edge from some vertex x to some vertex y , then the element $a_{x,y}$ is 1 (or in general the number of xy edges), otherwise it is 0. In computing, this matrix makes it easy to find subgraphs, and to reverse a directed graph.

Q.6. How do we compare Graph with other Data Structures?

Ans.: Graph data structures are non-[hierarchical](#) and therefore suitable for data sets where the individual elements are interconnected in complex ways. For example, a [computer network](#) can be modeled with a graph.

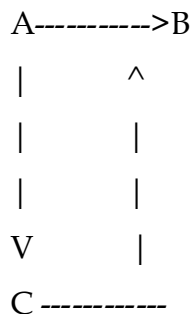
Hierarchical data sets can be represented by a [binary](#) or [nonbinary tree](#). It is worth mentioning, however, that trees can be seen as a special form of graph.

Operations : Graph algorithms are a significant field of interest within computer science. Typical operations associated with graphs are: finding a path between two nodes, like [depth-first search](#) and [breadth-first search](#) and finding the shortest path from one node to another, like [Dijkstra's algorithm](#). A solution to finding the shortest path from each node to every other node also exists in the form of the [Floyd-Warshall algorithm](#).

A directed graph can be seen as a [flow network](#), where each edge has a capacity and each edge receives a flow. The [Ford-Fulkerson algorithm](#) is used to find out [the maximum flow](#) from a source to a sink in a graph.

The graphs can be represented in two ways. One is adjacency matrix and adjacency list.

For example, let us consider the following graph



Adjacency Matrix :

```

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

```

Adjacency List :

```

A ----> | B | ----> | C | ---- NULL
B ----> ---- NULL
C ----> | B | ---- NULL

```

Q.7. Explain Depth-First Search.

Ans.: Depth-First Search (DFS) is an [algorithm](#) for traversing or searching a [tree](#), [tree structure](#), or [graph](#). One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before [backtracking](#).

The depth first search is well geared towards problems where we want to find any solution to the problem (not necessarily the shortest path), or to visit all of the nodes in the graph

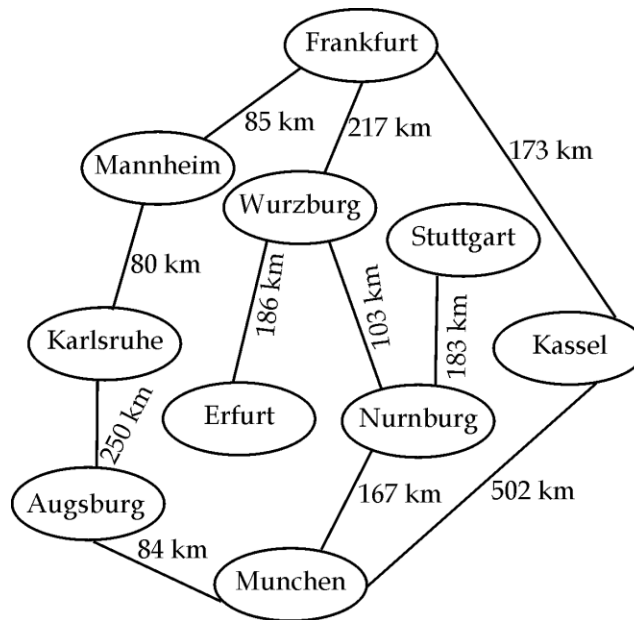
This concept maps extremely well to a Depth First search. The basic concept is to visit a node, then push all of the nodes to be visited onto the stack. To find the next node to visit we simply pop a node of the stack, and then push all the nodes connected to that one onto the stack as well and we continue doing this until all nodes are visited. It is a key property of the Depth First search that we not visit the same node more than once, otherwise it is quite possible that we will recurse infinitely. We do this by marking the node as we visit it, then unmarking it after we have finished our recursions. This action allows us to visit all the paths that exist in a graph; however for large graphs this is mostly infeasible so we sometimes omit the marking the node as not visited step to just find one valid path through the graph (which is good enough most of the time).

Q.8. Explain Breadth-First Search.

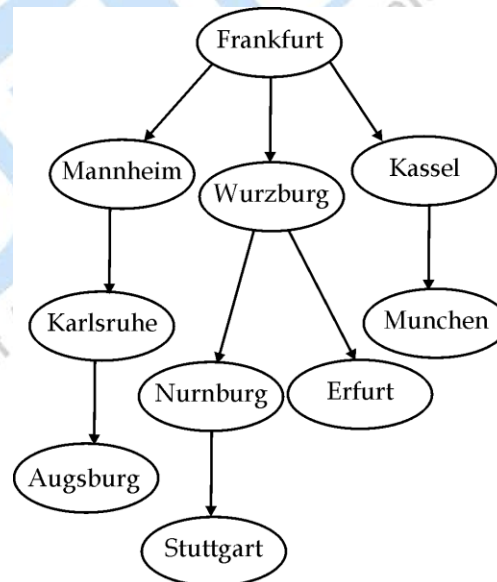
Ans.: In [graph theory](#), breadth-first search ([BFS](#)) is a [graph search algorithm](#) that begins at the root [node](#) and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbour nodes, and so on, until it finds the goal.

How it works : BFS is a [uninformed search](#) method that aims to expand and examine all nodes of a [graph](#) systematically in search of a solution. In other words, it exhaustively searches the entire graph without considering the goal until it finds it. It does not use a [heuristic](#).

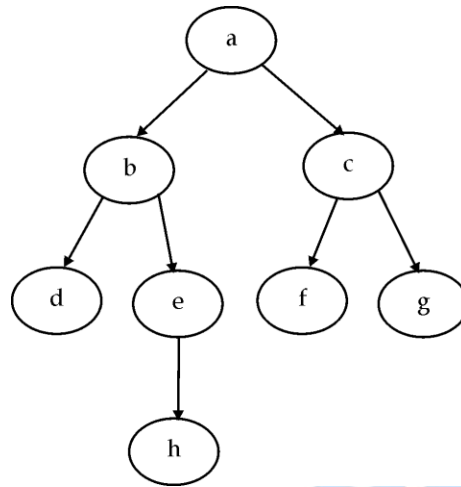
From the standpoint of the [algorithm](#), all child nodes obtained by expanding a node are added to a [FIFO queue](#). In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue or [linked list](#)) called "open" and then once examined are placed in the container "closed".



An example map of [Germany](#) with some connections between cities.



The breadth-first tree one gets when running BFS on the given map and starting in [Frankfurt](#).



Animated example of a breadth-first search

Algorithm (Informal) :

- (1) Put the ending node (the root node) in the queue.
- (2) Pull a node from the beginning of the queue and examine it.
 - If the searched element is found in this node, quit the search and return a result.
 - Otherwise push all the (so-far-unexamined) successors (the direct child nodes) of this node into the end of the queue, if there are any.
- (3) If the queue is empty, every node on the graph has been examined -- quit the search and return "not found".
- (4) Repeat from Step 2.

C Implementation :

Algorithm of Breadth-First Search :

```

void BFS(VLink G[], int v) {
    int w;
    VISIT(v);          /*visit vertex v*/
    visited[v] = 1;    /*mark v as visited : 1 */
    ADDQ(Q,v);
    while(!QMPTYQ(Q)) {
        v = DELQ(Q);   /*Dequeue v*/
    }
}
  
```



```

w = FIRSTADJ(G,v); /*Find first neighbor, return -1 if no neighbor*/
while(w != -1) {
    if(visited[w] == 0) {
        VISIT(w); /*visit vertex v*/
        ADDQ(Q,w); /*Enqueue current visited vertex w*/
        visited[w] = 1; /*mark w as visited*/
    }
    W = NEXTADJ(G,v); /*Find next neighbor, return -1 if no neighbor*/
}
}
}

```

Main Algorithm of apply Breadth-first search to graph $G=(V,E)$

```

void TRAVEL_BFS(VLink G[], int visited[], int n) {
    int i;
    for(i = 0; i < n; i++) {
        visited[i] = 0; /* Mark initial value as 0 */
    }
    for(i = 0; i < n; i++)
        if(visited[i] == 0)
            BFS(G,i);
}

```

C++ implementation

This is the implementation of the above informal algorithm, where the "so-far-unexamined" is handled by the parent array.

Suppose we have a struct:

```

struct Vertex {
    ...
    std::vector<int> out;
    ...
};

```


and an array of vertices: (the algorithm will use the indexes of this array, to handle the vertices)

```
std::vector<Vertex> graph(vertices);
```

the algorithm starts from start and returns true if there is a directed path from start to end:

```
bool BFS(const std::vector<Vertex>& graph, int start, int end) {
    std::queue<int> next;
    std::map<int,int> parent;
    parent[start] = -1;
    next.push(start);
    while (!next.empty()) {
        int u = next.front();
        next.pop();
        // Here is the point where you can examine the u th vertex of graph
        // For example:
        if (u == end) return true;
        for (std::vector<int>::const_iterator j = graph[u].out.begin(); j !=
graph[u].out.end(); ++j) {
            // Look through neighbors.
            int v = *j;
            if (parent.count(v) == 0) {
                // If v is unvisited.
                parent[v] = u;
                next.push(v);
            }
        }
    }
    return false;
}
```

It also stores the parents of each node, from which you can get the path.

□ □ □

Chapter-11

Tree

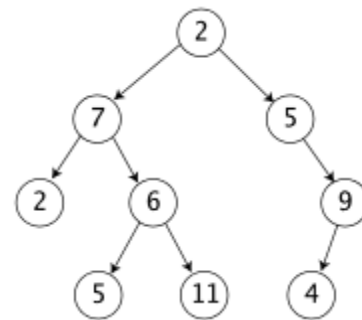
Q.1. What is Tree?

Ans.: A tree is a non-empty collection of vertices & edges that satisfies certain requirements. A vertex is a simple object (node) that can have a name and carry other associated information. An edge is a connection between two vertices.

A Tree is a finite set of a zero or more vertices such that there is one specially designated vertex called Root and the remaining vertices are partitioned into a collection of sub-trees, each of which is also a tree. A node may not have children, such node is known as Leaf (terminal node). The line from parent to a child is called a branch or an edge. Children to same parent are called siblings.

Q.2. What is Binary Tree?

Ans.: In computer science, a binary tree is a tree data structure in which each node has at most two children. Typically the child nodes are called left and right. One common use of binary trees is binary search trees; another is binary heaps.



A simple binary tree of size 9 and height 4, with a root node whose value is 2

In other words, A binary tree consists of

(α) a node (called the root node) and

(β) left and right sub-trees.

Both the sub-trees are themselves binary trees.

The nodes at the lowest levels of the tree (the ones with no sub-trees) are called leaves.

In an ordered binary tree,

- (a) the keys of all the nodes in the left sub-tree are less than that of the root,
- (b) the keys of all the nodes in the right sub-tree are greater than that of the root,
- (c) the left and right sub-trees are themselves ordered binary trees.

Q.3. Explain the different definitions for Rooted Tree.

Ans.: Definitions for Rooted Trees :

- A **directed edge** refers to the link from the parent to the child (the arrows in the picture of the tree).
- The root node of a tree is the node with no parents. There is at most one root node in a rooted tree.
- A leaf is a node that has no children.
- The **depth of a node** n is the length of the path from the root to the node. The set of all nodes at a given depth is sometimes called a level of the tree. The root node is at depth zero.
- The **height of a tree** is the length of the path from the root node to its furthest leaf. A tree with only a root node has a height of zero.
- **Siblings** are nodes that share the same parent node.
- If a **path** exists from node p to node q , where node p is closer to the root node than q , then p is an ancestor of q and q is a descendant of p .
- The **size of a node** is the number of descendants it has including itself.

Q.4. Explain about the different types of Binary Tree.

Ans.: Types of Binary Trees :

- A **rooted binary tree** is a rooted [tree](#) in which every node has at most two children.
- A **full binary tree**, or proper binary tree, is a tree in which every node has zero or two children.
- A **perfect binary tree** (sometimes complete binary tree) is a full binary tree in which all leaves are at the same depth.
- A **complete binary tree** may also be defined as a full binary tree in which all leaves are at depth n or $n-1$ for some n . In order for a tree to be the latter kind of complete binary tree, all the children on the last level must occupy the leftmost spots consecutively, with no spot left unoccupied in between any two. For example, if two nodes on the bottommost level each occupy a spot with an empty spot between the two of them, but the rest of the children nodes are tightly wedged together with no spots in between, then the tree cannot be a complete binary tree due to the empty spot.
- A **rooted complete binary tree** can be identified with a [free magma](#).
- An **almost complete binary tree** is a tree in which each node that has a right child also has a left child. Having a left child does not require a node to have a right child. Stated alternately, an almost complete binary tree is a tree where for a right child, there is always a left child, but for a left child there may not be a right child.
- The number of nodes n in a complete binary tree can be found using this formula: $n = 2^{h+1} - 1$ where h is the height of the tree.
- The number of leaf nodes n in a complete binary tree can be found using this formula: $n = 2^h$ where h is the height of the tree.

Q.5. What do you mean by Binary Search Tree?


Ans.: In [computer science](#), a binary search tree (BST) is a [binary tree data structure](#) which has the following properties :

- each node has a value;
- a [total order](#) is defined on these values;

- the left subtree of a node contains only values less than the node's value;
- the right subtree of a node contains only values greater than or equal to the node's value.

The major advantage of binary search trees is that the related [sorting algorithms](#) and [search algorithms](#) such as [in-order traversal](#) can be very efficient. Binary search trees are a fundamental [data structure](#) used to construct more abstract data structures such as [sets](#), [multisets](#), and [associative arrays](#). If a BST allows duplicate values, then it represents a multiset. This kind of tree uses non-strict inequalities. Everything in the left subtree of a node is strictly less than the value of the node, but everything in the right subtree is either greater than or equal to the value of the node. If a BST doesn't allow duplicate values, then the tree represents a set with unique values, like the mathematical set. Trees without duplicate values use strict inequalities, meaning that the left subtree of a node only contains nodes with values that are less than the value of the node, and the right subtree only contains values that are greater. The choice of storing equal values in the right subtree only is arbitrary; the left would work just as well. One can also permit non-strict equality in both sides. This allows a tree containing many duplicate values to be balanced better, but it makes [searching](#) more complex.

Q.6. What are the different Operations of Tree Structure?

Ans.:  **Operations :** All operations on a binary tree make several calls to a [comparator](#), which is a [subroutine](#) that computes the total order on any two values. In generic implementations of binary search trees, a program often provides a [callback](#) to a comparator when it creates a tree, either explicitly or, in languages that support [type polymorphism](#), by having values be of a comparable type.

- **Searching :** Searching a binary tree for a specific value is a process that can be performed recursively because of the order in which values are stored. We begin by examining the root. If the value we are searching for equals the root, the value exists in the tree. If it is less than the root, then it must be in the left subtree, so we recursively search the left subtree in the same manner. Similarly, if it is greater than the root, then it must be in the right subtree, so we recursively search the right subtree. If we reach a leaf and have not found the value, then the item is not where it

would be if it were present, so it does not lie in the tree at all. A comparison may be made with [binary search](#), which operates in nearly the same way but using random access on an array instead of following links. Here is the search algorithm in the [Python programming language](#): `<source lang="python">`

```
def search_binary_tree(node, key):
    if node is None:
        return None # key not found
    if key < node.key:
        return search_binary_tree(node.left, key)
    else if key > node.key:
        return search_binary_tree(node.right, key)
    else: # key is equal to node key
        return node.value # found key
```

`</source>` This operation requires $O(\log n)$ time in the average case, but needs $O(n)$ time in the worst-case, when the unbalanced tree resembles a linked list (degenerate tree).

- **Insertion** : Insertion begins as a search would begin; if the root is not equal to the value, we search the left or right subtrees as before. Eventually, we will reach an external node and add the value as its right or left child, depending on the node's value. In other words, we examine the root and recursively insert the new node to the left subtree if the new value is less than the root, or the right subtree if the new value is greater than or equal to the root. Here's how a typical binary search tree insertion might be performed in [C++](#): `<source lang="cpp">`

```
/* Inserts the node pointed to by "newNode" into the subtree rooted at
"treeNode" */
void InsertNode(struct node *&treeNode, struct node *newNode)
{
    if (treeNode == NULL)
        treeNode = newNode;
```



```

else if (newNode->value < treeNode->value)
    InsertNode(treeNode->left, newNode);
else
    InsertNode(treeNode->right, newNode);
}

```

</source> The above "destructive" procedural variant modifies the tree in place. It uses only constant space, but the previous version of the tree is lost. Alternatively, as in the following [Python](#) example, we can reconstruct all ancestors of the inserted node; any reference to the original tree root remains valid, making the tree a [persistent data structure](#): <source lang="python">

```

def binary_tree_insert(node, key, value):
    if node is None:
        return TreeNode(None, key, value, None)
    if key == node.key:
        return TreeNode(node.left, key, value, node.right)
    if key < node.key:
        return TreeNode(binary_tree_insert(node.left, key, value),
                        node.key,
                        node.value, node.right)
    else:
        return TreeNode(node.left, node.key, node.value,
                        binary_tree_insert(node.right, key, value))

```

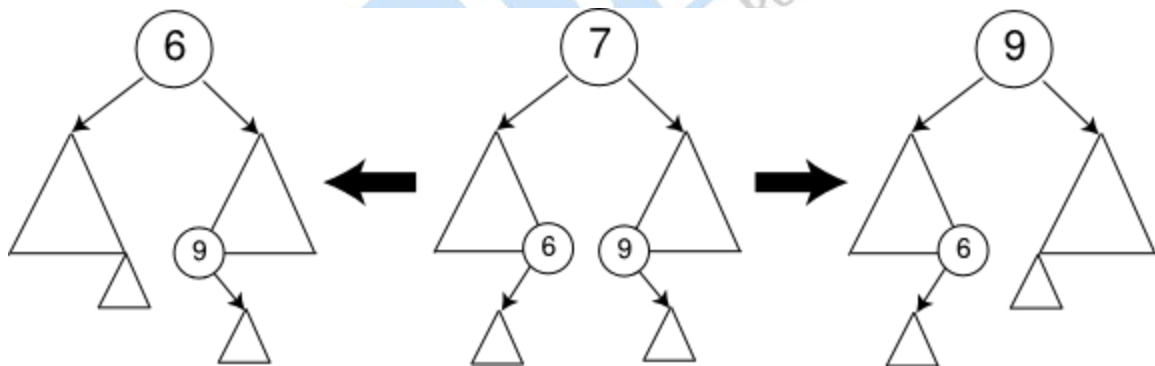
</source> The part that is rebuilt uses $\Theta(\log n)$ space in the average case and $\Omega(n)$ in the worst case (see [big-O notation](#)). In either version, this operation requires time proportional to the height of the tree in the worst case, which is $\mathcal{O}(\log n)$ time in the average case over all trees, but $\Omega(n)$ time in the worst case. Another way to explain insertion is that in order to insert a new node in the tree, its value is first compared with the value of the root. If its value is less than the root's, it is then

compared with the value of the root's left child. If its value is greater, it is compared with the root's right child. This process continues, until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its value. There are other ways of inserting nodes into a binary tree, but this is the only way of inserting nodes at the leaves and at the same time preserving the BST structure.

-

Deletion : There are several cases to be considered:

- Deleting a leaf: Deleting a node with no children is easy, as we can simply remove it from the tree.
- Deleting a node with one child: Delete it and replace it with its child.
- Deleting a node with two children: Suppose the node to be deleted is called N. We replace the value of N with either its in-order successor (the left-most child of the right subtree) or the in-order predecessor (the right-most child of the left subtree).



Once we find either the in-order successor or predecessor, swap it with N, and then delete it. Since both the successor and the predecessor must have fewer than two children, either one can be deleted using the previous two cases. In a good implementation, it is generally recommended to avoid consistently using one of these nodes, because this can [unbalance](#) the tree. Here is [C++](#) sample code for a destructive version of deletion. (We assume the node to be deleted has already been located using search.)

```
<source lang="cpp">
void DeleteNode(struct node * &node) {
```

```
if (node->left == NULL) {
    struct node *temp = node;
    node = node->right;
    delete temp;
} else if (node->right == NULL) {
    struct node *temp = node;
    node = node->left;
    delete temp;
} else {
    // In-order predecessor (rightmost child of left subtree)
    // Node has two children - get max of left subtree
    struct node **temp = &node->left; // get left node of the original
node

    // find the rightmost child of the subtree of the left node
    while ((*temp)->right != NULL) {
        temp = &(*temp)->right;
    }

    // copy the value from the in-order predecessor to the original
node
    node->value = (*temp)->value;

    // then delete the predecessor
    DeleteNode(*temp);
}
}
```

Although this operation does not always traverse the tree down to a leaf, this is always a possibility; thus in the worst case it requires time proportional to the height of the tree. It does not require more even when the node has two children, since it still follows a single path and does not visit any node twice.

Sort

A binary search tree can be used to implement a simple but inefficient [sorting algorithm](#). Similar to [heapsort](#), we insert all the values we wish to sort into a new ordered data structure — in this case a binary search tree — and then traverse it in order, building our result: `<source lang="python">`

```
def build_binary_tree(values):
    tree = None
    for v in values:
        tree = binary_tree_insert(tree, v)
    return tree

def traverse_binary_tree(treenode):
    if treenode is None: return []
    else:
        left, value, right = treenode
        return (traverse_binary_tree(left), [value],
                traverse_binary_tree(right))
```

`</source>` The worst-case time of `build_binary_tree` is $\Theta(n^2)$ — if you feed it a sorted list of values, it chains them into a [linked list](#) with no left subtrees. For example, `build_binary_tree([1, 2, 3, 4, 5])` yields the tree (None, 1, (None, 2, (None, 3, (None, 4, (None, 5, None))))). There are several schemes for overcoming this flaw with simple binary trees; the most common is the [self-balancing binary search tree](#). If this same procedure is done using such a tree, the overall worst-case time is $O(n \log n)$, which is [asymptotically optimal](#) for a [comparison sort](#). In practice, the poor [cache](#) performance and added overhead in time and space for a tree-

based sort (particularly for node [allocation](#)) make it inferior to other asymptotically optimal sorts such as [quicksort](#) and [heapsort](#) for static list sorting. On the other hand, it is one of the most efficient methods of incremental sorting, adding items to a list over time while keeping the list sorted at all times.

Q.7. What do you mean by Tree Traversal?

Ans.: Traversal : Once the binary search tree has been created, its elements can be retrieved [in order](#) by recursively traversing the left subtree of the root node, accessing the node itself, then recursively traversing the right subtree of the node, continuing this pattern with each node in the tree as it's recursively accessed. The tree may also be traversed in [pre-order](#) or [post-order](#) traversals.

```
<source lang="python">
def traverse_binary_tree(treenode):
```

```
    if treenode is None: return
```

```
    left, nodevalue, right = treenode
```

```
    traverse_binary_tree(left)
```

```
    visit(nodevalue)
```

```
    traverse_binary_tree(right)
```

```
</source>
```

Traversal requires $\Omega(n)$ time, since it must visit every node. This algorithm is also $O(n)$, and so it is [asymptotically optimal](#).

Traversing A Binary Tree : Traversing a binary tree comes in handy when you would like to do print out of all the data elements in the tree. We demonstrate three types of traversals in our tutorial.

All traversal descriptions refer to :

These three types are as follows :

- **Pre Order Traversal :** A pre order traversal prints the contents of a sorted tree, in pre order. In other words, the contents of the

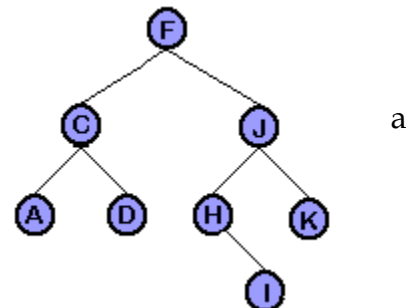


Figure : Sorted Binary Tree

root node are printed first, followed by left subtree and finally the right subtree. So in Figure , an in order traversal would result in the following string: FCADJHIK

```
PreOrder (T)
  If T < > Null
  then print (T.data)
  else print('empty tree')
  If T.lp < > null
  then PreOrder(T.lp)
  If T.rp < > null
  then preorder (T.rp)
  end.
```

- **In Order Traversal :** An in order traversal prints the contents of a sorted tree, in order. In other words, the lowest in value first, and then increasing in value as it traverses the tree. The order of a traversal would be 'a' to 'z' if the tree uses strings or characters, and would be increasing numerically from 0 if the tree contains numerical values. So in Figure , an in order traversal would result in the following string: ACDFHIJK

```
InOrder (T)
  If T < > null
  print ('empty tree')
  If T.lp < > null
  then InOrder(T.lp)
  print (T.data)
  If T.rp < > null
  then InOrder (T.rp)
  end.
```

- **Post Order Traversal :** A post order traversal prints the contents of a sorted tree, in post order. In other words, the contents of the left subtree are printed

first, followed by right subtree and finally the root node. So in Figure , an in order traversal would result in the following string: ADCIHKJF.

```

PostOrder (T)
  If T = null
  then print ('empty tree')
  If T.lp < > null
  then PostOrder(T.lp)
  If T.rp < > null
  then PostOrder(T.rp)
  Print(T.data)
end.

```

Q.8. What is a Heap?

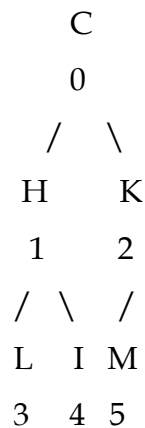
Ans.: Definition : A minimal heap (descending heap) is an almost complete binary tree in which the value at each parent node is less than or equal to the values in its child nodes.

Obviously, the minimum value is in the root node. Note, too, that any path from a leaf to the root passes through the data in descending order.

Here is an example of a minimal heap :



Storage of Heap Data : The typical storage method for a heap, or any almost complete binary tree, works as follows. Begin by numbering the nodes level by level from the top down, left to right. For example, consider the following heap. The numbering has been added below the nodes.



Then store the data in an array as shown below :

C	H	K	L	I	M
0	1	2	3	4	5

The advantage of this method over using the usual pointers and nodes is that there is no wasting of space due to storing two pointer fields in each node. Instead, starting with the current index, CI, one calculates the index to use as follows :

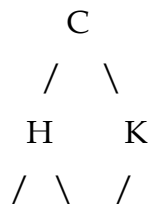
$$\text{Parent}(\text{CI}) = (\text{CI} - 1) / 2$$

$$\text{RightChild}(\text{CI}) = 2 * (\text{CI} + 1)$$

$$\text{LeftChild}(\text{CI}) = 2 * \text{CI} + 1$$

For example, if we start at node H (with index 1), the right child is at index $2 * (1 + 1) = 4$, that is, node I.

Inserting into a Heap : This is done by temporarily placing the new item at the end of the heap (array) and then calling a FilterUp routine to make any needed adjustments on the path from this leaf to the root. For example, let's insert E into the following heap :



L I M

First, temporarily place E in the next available position :

```

      C
     / \
    H   K
   / \ / \
  L  I M  E

```

Of course, the new tree might not be a heap. The FilterUp routine now checks the parent, K, and sees that things would be out of order as they are. So K is moved down to where E was. Then the parent above that, C, is checked. It is in order relative to the target item E, so the C is not moved down. The hole left behind is filled with E, then, as this is the correct position for it.

```

      C
     / \
    H   E
   / \ / \
  L  I M  K

```

For practice, let's take the above heap and insert another item, D. First, place D temporarily in the next available position :

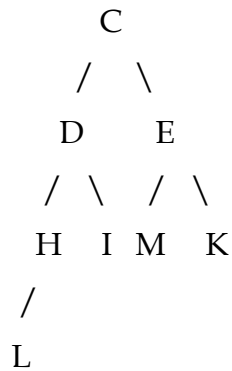
```

      C
     / \
    H   E
   / \ / \
  L  I M  K
 /
D

```

Then the FilterUp routine checks the parent, L, and discovers that L must be moved down. Then the parent above that, H, is checked. It too must be moved

down. Finally C is checked, but it is OK where it is. The hole left where the H had been is where the target D is then inserted.



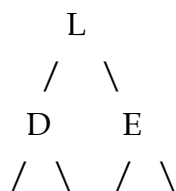
Things have now been adjusted so that we again have a heap!

Removing from a Heap : We always remove the item from the root. That way we always get the smallest item. The problem is then how to adjust the binary tree so that we again have a heap (with one less item).

The algorithm works like this: First, remove the root item and replace it temporarily with the item in the last position. Call this replacement the target. A FilterDown routine is then used to check the path from the root to a leaf for the correct position for this target. The path is chosen by always selecting the smaller child at each node. For example, let's remove the C from this heap :



First we remove the C and replace it with the last item (the target), L :



H I M K

The smaller child of L is D. Since D is out of order compared to the target L, we move D up. The smaller child under where D had been is H. When H is compared to L we see that the H too needs to be moved up. Since we are now at a leaf, this empty leaf is where the target L is put.

```

      D
     / \
    H   E
   / \ / \
  L  I M  K

```

For another example, let's remove the E from the following heap :

```

      E
     / \
    G   K
   / \ / \
  J  N K  X
 / \ /
X  Y P

```

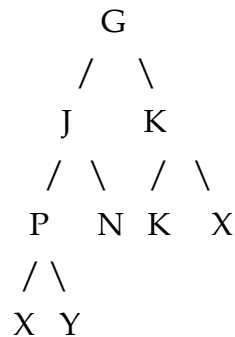
First remove the E and replace it with the target P (the last item) :

```

      P
     / \
    G   K
   / \ / \
  J  N K  X
 / \ /
X  Y

```

Now use the FilterDown routine to filter the P down to its correct position by checking the smaller child, G, which should be moved up, and then the smaller child below that, J, which should also be moved up. Finally, the smaller child, X, under where J had been is checked, but it does not get moved since it is OK relative to the target P. The P is then placed in the empty node above X. We then have the following heap:



Q.9. Explain Heapsort in terms of tree structure.

Ans.: Heapsort is performed by somehow creating a heap and then removing the data items one at a time. The heap could start as an empty heap, with items inserted one by one. However, there is a relatively easy routine to convert an array of items into a heap, so that method is often used. This routine is described below. Once the array is converted into a heap, we remove the root item (the smallest), readjust the remaining items into a heap, and place the removed item at the end of the heap (array). Then we remove the new item in the root (the second smallest), readjust the heap, and place the removed item in the next to the last position, etc.

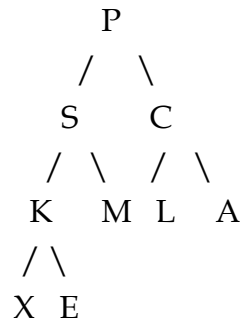
Heapsort is $\Theta(n \cdot \lg(n))$, either average case or worst case. This is great for a sorting algorithm! No appreciable extra storage space is needed either. On average, quicksort (which is also $\Theta(n \cdot \lg(n))$ for the average case) is faster than heapsort. However, quicksort has that bad $\Theta(n^2)$ worst case running time.

Example Trace of Heapsort : Let's heapsort the following array :

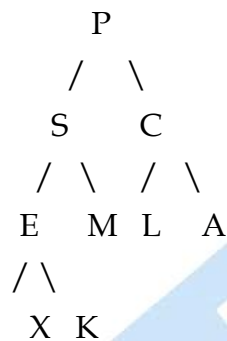
P	S	C	K	M	L	A	X	E
0	1	2	3	4	5	6	7	8

To convert this to a heap, first go to the index of the last parent node. This is given by $(\text{HeapSize} - 2) / 2$. In this case, $(9 - 2) / 2 = 3$. Thus K is the last parent in the tree. We then apply the FilterDown routine to each node from this index down to index 0. (Note that this is each node from 3 down to 0, not just the nodes along the path from index 3 to index 0.)

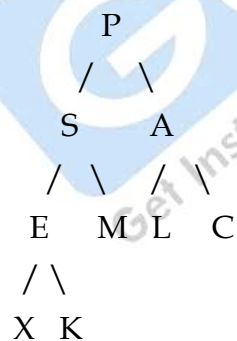
In our example, the array corresponds directly to the following binary tree. Note that this is not yet a heap.



Applying FilterDown at K gives the following. (Note that E is the smaller child under K.)

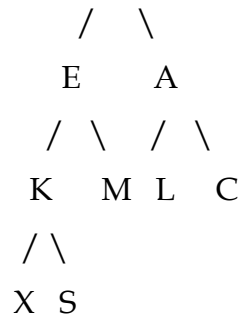


Now apply FilterDown at index 2, that is, at node C. (Under C, A is the smaller child.)

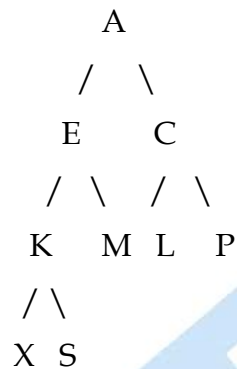


Next, apply FilterDown at index 1, that is, at node S. Check the smaller child, E, and then the smaller child under that, namely K. Both E and K get moved up.

P

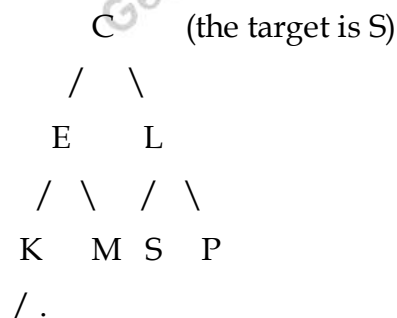


Finally, apply FilterDown at index 0, that is, at the root node. We check the smaller child, A, and then the smaller child, C, relative to the target P. Both A and C get moved up.



Now we have a heap! The first main step of heapsort has been completed. The other main component of heapsort was described earlier: to repeatedly remove the root item, adjust the heap, and put the removed item in the empty slot toward the end of the array (heap).

First we remove the A, adjust the heap by using FilterDown at the root node, and place the A at the end of the heap (where it is not really part of the heap at all and so is not drawn below as connected to the tree).



X A

Of course, all of this is really taking place in the array that holds the heap. At this point it looks like the following. Note that the heap is stored from index 0 to index 7. The A is after the end of the heap.

C	E	L	K	M	S	P	X	A
0	1	2	3	4	5	6	7	8

Next we remove the C, adjust the heap by using FilterDown at the root node, and place the C at the end of the heap:

E (the target is X)

```

  /  \
 K    L
 /  \ /  \
X   M S   P
..
C A

```

Next we remove the E, adjust the heap by using FilterDown at the root node, and place the E at the end of the heap :

K (the target is P)

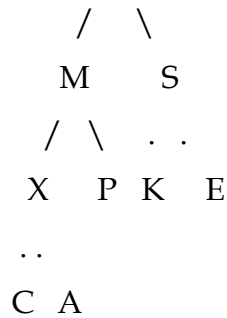
```

  /  \
 M    L
 /  \ /  \
X   P S   E
..
C A

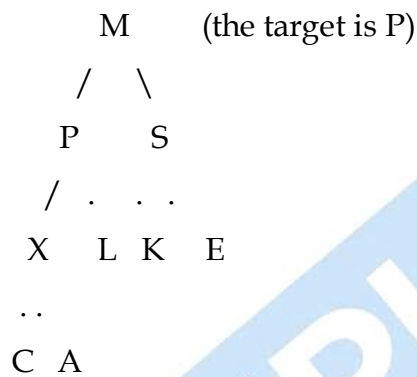
```

Next we remove the K, adjust the heap by using FilterDown at the root node, and place the K at the end of the heap :

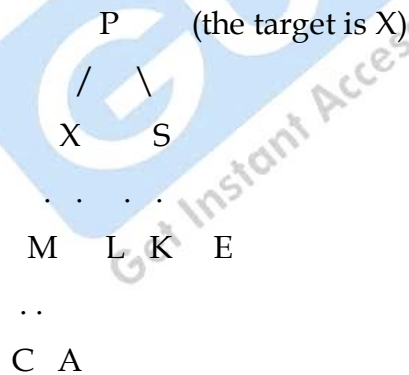
L (the target is S)



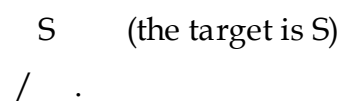
Next we remove the L, adjust the heap by using FilterDown at the root node, and place the L at the end of the heap :



Next we remove the M, adjust the heap by using FilterDown at the root node, and place the M at the end of the heap :



Next we remove the P, adjust the heap by using FilterDown at the root node, and place the P at the end of the heap :



```

      X    P
    . . . .
  M    L K  E
..
C  A

```

Next we remove the S, adjust the heap (now a trivial operation), and place the S at the end of the heap :

```

      X    (the target is X)
    . .
  S    P
    . . . .
  M    L K  E
..
C  A

```

Since only the item X remains in the heap, and since we have removed the smallest item, then the second smallest, etc., the X must be the largest item and should be left where it is. If you now look at the array that holds the above items you will see that we have sorted the array in descending order:

X	S	P	M	L	K	E	C	A
0	1	2	3	4	5	6	7	8

Q.10. Define Hash Tables.

Ans.: In [computer science](#), a hash table, or a hash map, is a [data structure](#) that associates [keys](#) with [values](#). The primary operation it supports efficiently is a lookup: given a key (e.g. a person's name), find the corresponding value (e.g. that person's telephone number). It works by transforming the key using a [hash function](#) into a hash, a number that is used as an index in an array to locate the desired location ("bucket") where the values should be.

Hash tables support the efficient insertion of new entries, in expected [O\(1\)](#) time. The time spent in searching depends on the hash function and the load

of the hash table; both insertion and search approach $O(1)$ time with well chosen values and hashes.

Hashing means the act of taking some value and producing a number from the value. A hash function is a function that does this. Every equivalence predicate e has a set of acceptable hash functions for that predicate; a hash function h is acceptable iff $(e \text{ obj1 obj2}) \rightarrow (= (\text{hash obj1}) (\text{hash obj2}))$.

A hash function h is good for a equivalence predicate e if it distributes the result numbers (hash values) for non-equal objects (by e) as uniformly as possible over the numeric range of hash values, especially in the case when some (non-equal) objects resemble each other by e.g. having common subsequences. This definition is vague but should be enough to assert that e.g. a constant function is not a good hash function.

Basic Operation : A hash table works by transforming the key using a [hash function](#) into a hash, a number that is used as an index in an array to locate the desired location ("bucket") where the values should be. The number is normally converted into the index by taking a [modulo](#), or sometimes [bit masking](#) is used where the array size is a power of two. The optimal hash function for any given use of a hash table can vary widely, however, depending on the nature of the key.

Typical operations on a hash table include insertion, deletion and lookup (although some hash tables are precalculated so that no insertions or deletions, only lookups are done on a live system). These operations are all performed in amortized constant time, which makes maintaining and accessing a huge hash table very efficient.

It is also possible to create a hash table statically where, for example, there is a fairly limited fixed set of input values - such as the value in a single byte (or possibly two bytes) from which an index can be constructed directly (see section below on creating hash tables). The hash table can also be used simultaneously for tests of validity on the values that are disallowed.

Q.11. What do you mean by Threaded Binary Tree?

Ans.: Threaded [binary tree](#) may be defined as follows :

A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node, and all left child

pointers that would normally be null point to the inorder predecessor of the node."

A threaded [binary tree](#) makes it possible to traverse the values in the [binary tree](#) via a linear traversal that is more rapid than a recursive [in-order traversal](#).

It is also possible to discover the parent of a node from a threaded binary tree, without explicit use of parent pointers or a stack, albeit slowly. This can be useful however where stack space is limited, or where a stack of parent pointers is unavailable.

This is possible, because if a node (k) has a right child (m) then m's left pointer must be either a child, or a thread back to k. In the case of a left child, that left child must also have a left child or a thread back to k, and so we can follow m's left children until we find a thread, pointing back to k. The situation is similar for when m is the left child of k

In pseudocode,

function getParent(node : pointer to a node)

begin

if node == tree.root then

begin

return nil

end

x = node

y = node

while true do

begin

if IsThread(Y, Right) then

begin

p = y.right

if (p == nil) or (p.left <> node) then

begin

p = x

while not IsThread(p, Left) do

```

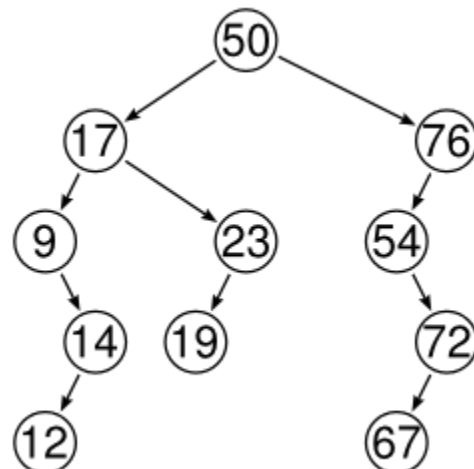
    p = p.left
    p = p.left
end
return p
end
if IsThread(Y, Left) then
begin
    p = x.left
    if (p == nil) or (p.left <> node) then
    begin
        p = y
        while not IsThread(p, Right) do
            p = p.right
        p = p.right
    end
    return p
end
x = x.left
y = y.right
end

```

Q.12. Define AVL Trees.

Ans.: An example of an unbalanced non-AVL tree

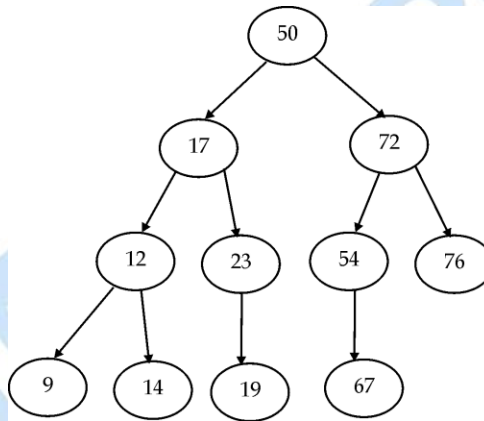
In [computer science](#), an AVL tree is a [self-balancing binary search tree](#), and the first such data structure to be invented[citation needed]. In an AVL tree the [heights](#) of the two [child](#) subtrees of any node differ by at most one, therefore it is also called [height-balanced](#). Lookup, insertion,



and deletion all take $O(\log n)$ time in both the average and worst cases. Additions and deletions may require the tree to be rebalanced by one or more [tree rotations](#).

The AVL tree is named after its two inventors, [G.M. Adelson-Velsky](#) and [E.M. Landis](#), who published it in their 1962 paper "An algorithm for the organization of information."

The balance factor of a node is the height of its right subtree minus the height of its left subtree. A node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees.



AVL trees are often compared with [red-black trees](#) because they support the same set of operations and because red-black trees also take $O(\log n)$ time for the basic operations. AVL trees perform better than red-black trees for lookup-intensive applications.^[1] The AVL tree balancing algorithm appears in many computer science curricula.

The same tree after being height-balanced.

Explanation : First, see [binary tree](#) for an explanation of a normal binary tree. An AVL tree operates exactly like a normal binary tree, except additional work is done after an insertion and a deletion.

The problem with normal binary trees is that if data is added in-order then the tree isn't a tree at all - it's a list. For example, if we had five data to add to a tree where each datum is an integer, and we added the integers 1, 2, 3, 4 and 5, in that order, the tree would not branch at all - it would be a single long line.

This is a problem because the whole point of a binary tree is that searching is fast because when the tree is balanced, each step in the search eliminates half of the tree at that point. (Imagine a very large, fully balanced tree; you start at the root node, and go left or right. In doing so, you've just eliminated half the elements from your search!)

So to solve this problem, an AVL tree can be used, which as mentioned, acts as a normal binary tree but does additional processing after an add or delete, so that the tree remains balanced, no matter what order data is placed (or removed) from the tree.

Now, to perform balancing work on the tree, we have to know when the tree is unbalanced - because we have to know when we need to fix things. One possible way to do this is to keep track in each node of the tree how many nodes exist on the left and right of each node. When we add a node, or delete a node, we travel up the tree from the point the node is added or deleted, updating the node counts. If the number of nodes is different, we know that the tree to the left and right cannot be balanced, because there are a different number of elements on each side.

However, this doesn't quite work - the problem is that the number of elements in a subtree doesn't tell you anything about how they are arranged. They could be properly balanced, or they could be a long list!

In fact what you need to keep track of is the depth of the subtree. A full subtree with a depth of three would have fourteen elements - and it would be perfectly balanced. A subtree with a depth of three which is unbalanced might have only three elements in (a list) - and so be in need of rebalancing.

Each time an element is added, we in fact update a count of the depth of the subtree on the left and right of each node. We travel up the tree from the point of addition, updating those values by adding one. We then travel back up the same path, this time rearranging the tree so that the subtree depths remain equal on each side.

(In fact, subtree depths must be equal or differ only by one - after all, we could have three elements on the left and two on the right, which would still be as balanced as possible. If we rebalanced that tree, all we'd do is then have two elements on the left and three on the right; we wouldn't gain anything).

Rebalancing the tree involves performing left and right rotation operations on unbalanced nodes, which adjusts the physical shape of the tree which keeping it logically valid (e.g. binary search down the tree remains correct).

In fact, the tree is rebalanced every time a node is inserted or deleted, which specifically limits the things that can have to be done to rebalance the tree, because the tree can never be more than one sub-tree depth out of balance.

Operations : The basic operations of an AVL tree generally involve carrying out the same algorithms as would be carried out on an unbalanced [binary search tree](#), but preceded or followed by one or more of the so-called "AVL rotations."

- **Insertion :** Insertion into an AVL tree may be carried out by inserting the given value into the tree as if it were an unbalanced binary search tree, and then retracing one's steps toward the root updating the balance factor of the nodes.[citation needed]

If the balance factor becomes -1, 0, or 1 then the tree is still in AVL form, and no rotations are necessary.

If the balance factor becomes 2 or -2 then the tree rooted at this node is unbalanced, and a [tree rotation](#) is needed. At most a single or double rotation will be needed to balance the tree.[citation needed]

Only the nodes traversed from the insertion point to the root of the tree need be checked, and rotations are a constant time operation, and because the height is limited to $O(\log(n))$, the execution time for an insertion is $O(\log(n))$.

- **Deletion :** If the node is a leaf, remove it. If the node is not a leaf, replace it with either the largest in its left subtree or the smallest in its right subtree, and remove that node. Thus the node that is removed has at most one child. After deletion retrace the path back up the tree to the root, adjusting the balance factors as needed.

The retracing can stop if the balance factor becomes -1 or 1 indicating that the height of that subtree has remained unchanged. If the balance factor becomes 0 then the height of the subtree has decreased by one

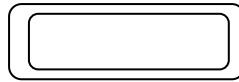
and the retracing needs to continue. If the balance factor becomes -2 or 2 then the subtree is unbalanced and needs to be rotated to fix it. If the rotation leaves the subtree's balance factor at 0 then the retracing towards the root must continue since the height of this subtree has decreased by one. This is in contrast to an insertion where a rotation resulting in a balance factor of 0 indicated that the subtree's height has remained unchanged.

The time required is $O(h)$ for lookup plus $O(h)$ rotations on the way back to the root; so the operation can be completed in $O(\log n)$ time.

□ □ □

**BACHELOR OF COMPUTER APPLICATIONS
(PART-I) EXAMINATION
ALGORITHMS AND DATASTRUCTURES
PAPER – 118**

OBJECTIVE PART- I



Time allowed : One Hour

Maximum Marks : 20

The question paper contains to 40 multiple choice questions with four choices and student will have to pick the correct one (each carrying $\frac{1}{2}$ mark).

1. What is a compact and informal high level description of a computer programming algorithm that use the structural conventions of a programming language, but is intended for human reading rather than machine reading?
(a) Pseudocode
(b) Algorithm
(c) Flowchart
(d) Website ()
2. Which one is the desirable characteristics of an algorithm?
(a) Correctness and ease of understanding
(b) Elegance and efficiency
(c) Both (a) and (b)
(d) None of the above ()
3. An algorithm that calls itself directly or indirectly is known as :
(a) Subalgorithm
(b) Recursion
(c) Polish Notation
(d) Nesting ()
4. Which of the following case does not exist in complexity theory?
(a) Best Case (b) Average Case
(c) Null Case (d) Worst Case ()
5. Two main measures for the efficiency of an algorithm are:

- (a) Processor and memory
(b) Complexity and capacity
(c) Time and space
(d) Data and space ()
6. The memory address of the first element of an array is:
(a) base address
(b) main address
(c) reference
(d) first address ()
7. Which of the following is not a basic data structure that could be used to implement an abstract data type?
(a) Array (b) Linked List
(c) Hash table (d) Heap ()
8. What term is used to describe an $O(n)$ algorithm?
(a) Constant (b) Linear
(c) Logarithmic (d) Quadratic ()
9. What is the time required to search an element in a linked list of length n ?
(a) $O(\log_2 n)$ (b) $O(n)$
(c) $O(1)$ (d) $O(n^2)$ ()
10. What is the time complexity of linear search algorithm over an array of n ?
(a) $O(\log_2 n)$ (b) $O(n)$
(c) $O(n \log_2 n)$ (d) $O(n^2)$ ()
11. Which data structure is needed to convert infix notations to postfix notations?
(a) Linear List (b) Queue
(c) Tree (d) Stack ()
12. Sorting is not possible by using which of the following method?
(a) Insertion (b) Selection
(c) Exchange (d) Deletion ()
13. In tree construction which is the suitable efficient data structure?
(a) Array (b) Linked List
(c) Stack (d) Queue ()

14. Which of the following abstract data types are not used by integer Abstract data type group?
(a) short (b) int
(c) float (d) long ()
15. If the depth of a tree is 3 levels, then what is the size of the tree?
(a) 4 (b) 2
(c) 8 (d) 6 ()
16. A Linked List index isthat represents the position of a node in a linked list:
(a) an integer (b) a variable
(c) a character (d) a boolean ()
17. What happens when you push a new node onto a stack?
(a) the new node is placed at the front of the linked list
(b) the new node is placed at the back of the linked list
(c) the new node is placed at the middle of the linked list
(d) no changes happens ()
18. The maximum number of nodes in a binary tree of depth k is:
(a) 2^{k-1} (b) $2^{(k+1)}$
(c) $2^k - 1$ (d) $2^{(k+1)} - 1$ ()
19. Which of the following is not a property of the binary tree?
(a) There is a specially designated node called the root
(b) The rest of the nodes could be partitioned into t -disjoint sets $t \geq 0$
(c) Any node should be reachable from anywhere in the tree
(d) At most one cycle could be present in the tree ()
20. If there exists at least one path between every pair of vertices in a graph, the graph is known as:
(a) Complete graph
(b) Disconnected graph
(c) Connected graph
(d) Euler graph ()
21. A simple graph with n vertices and k components can have at most:
(a) n edges (b) $n - k$

- (c) ~~$n(n-1)/2$~~ edges (d) ~~$n(n+1)/2$~~ edges ()
22. The element at the root of heap is:
 (a) Largest
 (b) Smallest
 (c) Depends of the type of heap
 (d) None of the above ()
23. Minimum number of queues needed to implement the priority queue is:
 (a) 1 (b) 3
 (c) 2 (d) 0 ()
24. The Queue data structure follows the order:
 (a) LIFO
 (b) Random
 (c) FIFO
 (d) None of the above ()
25. The underflow condition for the circular queue is when:
 (a) Front = rear
 (b) Front = rear + 1
 (c) Rear - front = 1
 (d) None of the above ()
26. Which of the following sorting algorithms has average case and worst case running time of?
 (a) Bubble sort (b) Insertion sort
 (c) Merge sort (d) Quick sort ()
27. A variable P is called pointer if :
 (a) P contains the address of an element in DATA
 (b) P points to the address of the first element in DATA
 (c) P can store only memory addresses
 (d) P contains the DATA and the address of DATA ()
28. Which of the following data structure store the homogeneous data elements?
 (a) Arrays
 (b) Records
 (c) Pointers
 (d) None of the above ()

29. In a graph if $e = (u, v)$ means:
(a) u is adjacent to v but v is not adjacent to u
(b) e begins at u and ends at v
(c) u is predecessor and v is successor
(d) both (b) and (c) ()
30. A connected graph T without any cycles is called:
(a) a tree graph
(b) free tree
(c) a tree
(d) All of the above ()
31. The post order traversal of a binary tree is DEBFCA. Find out the preorder traversal:
(a) ABFCDE (b) ADBFEC
(c) ABDECF (d) ABDCEF ()
32. Binary search algorithm cannot be applied to:
(a) Sorted Linked List (b) Sorted Binary Trees
(c) Sorted Linear Array (d) Pointer Array ()
33. The complexity of Bubble Sort algorithm is:
(a) $O(n)$ (b) $O(\log n)$
(c) $O(n^2)$ (d) $O(n \log n)$ ()
34. The complexity of Merge Sort algorithm is:
(a) $O(n)$ (b) $O(\log n)$
(c) $O(n^2)$ (d) $O(n \log n)$ ()
35. Finding the location of the element with a given value is:
(a) Traversal
(b) Search
(c) Sort
(d) None of the above ()
36. A Huffman tree is a kind of:
(a) Full binary tree
(b) Binary search tree
(c) Complete binary tree

- (d) None of the above ()
37. A linked list object keeps its collection of items:
(a) on the heap
(b) on the stack
(c) at consecutive memory locations
(d) all of the above ()
38. A video game server keeps a list of people waiting to play. The list should be a:
(a) Binary search tree (b) Stack
(c) Queue (d) Static array ()
39. Which one is not a type of Sorting Technique?
(a) Radix Sort (b) Binary Sort
(c) Decimal Sort (d) Both (b) and (c) ()
40. A sort which compares adjacent elements in a list and switches where necessary is:
(a) insertion sort (b) heap sort
(c) quick sort (d) bubble sort ()
-

DESCRIPTIVE PART – II

Year- 2011

Time allowed: 2 Hours

Maximum Marks : 30

Attempt any four descriptive type questions out of the six. All questions carry 7½ marks each.

- Q.1 (a) What is complexity of an algorithm? How is it measured? Discuss time space trade off with an example.
(b) What is Polish Notation? Explain with example.
- Q.2 (a) What do you mean by Sorting and Searching? Explain linear search algorithm and derive its best-case, average-case and worst-case complexity.
(b) What is an array? Discuss limitation of an array.
- Q.3 (a) Design a string manipulation algorithm for duplicating a given character string N times.
(b) What do you mean by traversal of a tree? Explain the difference between preorder and post order traversal with an example.
- Q.4 Differentiate the following :
(a) Graph and Tree;
(b) Internal and External Sorting;
(c) Recursion and Iteration;
(d) Stack and Queue.
- Q.5 (a) Write an algorithm for creation of Singly Linked List. Also write algorithms for insertion and deletion of elements from the singly linked list.
(b) Discuss Huffman encoding scheme with an example.
- Q. 6 Write short notes on the following:
(a) Abstract data type;
(b) Hash Table;
(c) Bubble Sort;
(d) Binary Tree.

ALGORITHMS AND DATA STRUCTURES

OBJECTIVE PART- I

Year - 2010

Time allowed : One Hour

Maximum Marks : 20

The question paper contains to 40 multiple choice questions with four choices and student will have to pick the correct one (each carrying $\frac{1}{2}$ mark).

1. Two main measures for the efficiency of an Algorithm are:
(a) Processor and Memory (b) Complexity and Capacity
(c) Time and Space (d) Data and Space ()
2. The time factor when determining the efficiency of an algorithm is measured by:
(a) Counting microseconds
(b) Counting the number of key operations
(c) Counting the number of statement
(d) Counting the Kilobytes of algorithm ()
3. A data structure is a way of:
(a) arrangement of different data elements
(b) organizing data with consideration of items stored into it along with their relationship with each other
(c) ordering of collected data
(d) None of the above ()
4. The space factor when determining the efficiency of algorithm is measured by:
(a) Counting the maximum memory needed by the algorithm
(b) Counting the minimum memory needed by the algorithm
(c) Counting the average memory needed by the algorithm
(d) Counting the maximum disk space needed by the algorithm ()
5. The de-queue process removes data:
(a) From the front of the queue
(b) From the bottom of the queue
(c) Can not be removed
(d) None of the above ()

6. A queue is a :
(a) Sequential Organization of data
(b) Listing of data
(c) Indexing of data
(d) None of the above ()
7.is a way of grouping things together by placing one thing on top of another and then removing things one at a time from the top.
(a) Array
(b) Stack
(c) Pointer
(d) All of the above ()
8. The complexity of Binary Search algorithm is:
(a) $O(n)$ (b) $O(\log n)$
(c) $O(n^2)$ (d) $O(n \log n)$ ()
9. The complexity of Bubble Sort algorithm is:
(a) $O(n)$ (b) $O(\log n)$
(c) $O(n^2)$ (d) $O(n \log n)$ ()
10. Each array declaration need not give, implicitly or explicitly, the information about:
(a) the name of array
(b) the data type of array
(c) the first data from the set to be stored
(d) the index set of the array ()
11. Which of the following is not linear data structure?
(a) Arrays
(b) Linked lists
(c) Both of the above
(d) None of the above ()
12. Finding the location of the element with a given value is:
(a) Traversal
(b) Search
(c) Sort
(d) None of the above ()
13. The operation of processing each element in the list is known as:

- (a) Sorting (b) Merging
(c) Inserting (d) Traversal ()
14. Arrays are best data structures :
(a) for relatively permanent collections of data
(b) for the size of the structure and the data in the structure are constantly changing
(c) for both of above situation
(d) for none of the above situation ()
15. Linked lists are best suited:
(a) For relatively permanent collections of data
(b) for the size of the structure and the data in the structure are constantly changing
(c) for both of above situation
(d) for none of the above situation ()
16. In tree construction which is the suitable efficient data structure :
(a) array (b) linked lists
(c) stack (d) queue ()
17. By a schematics character variable we mean:
(a) A variable whose length is defined before the program is executed and cannot change through the program
(b) A variable whose length may vary during the execution of the program as long as the length does not exceed a maximum value determined by the program before the program is executed
(c) (a) and (b) both
(d) None of the above ()
18. In this STACKS, "PUSH" is the term used to:
(a) insert an element into a stack
(b) delete an element from a stack
(c) modify the existing element of a stack
(d) None of the above ()
19. Maximum number of queues needed to implement the priority queue:
(a) Three (b) Two
(c) Five (d) One ()
20. Polish notation refers to the notations in which:
(a) the operator symbol is placed after its two operands

- (b) the operator symbol is placed before its two operands
(c) the operator symbol is placed in the middle
(d) None of the above ()
21. A node haselements.
(a) None (b) One
(c) Two (d) Three ()
22. Which of the following abstract data types are not used by Integer Abstract Data Type group?
(a) Short (b) Int
(c) Float (d) Long ()
23. There are.....standard ways of maintaining a graph G in the memory of a computer.
(a) four
(b) three
(c) two
(d) None of the above ()
24. In the sequential representation of Graph G in computer, it may be difficult to:
(a) insert nodes in G
(b) delete nodes in G
(c) (a) and (b) Both
(d) None of the above ()
25. The three standard ways of traversing a binary tree is:
(a) Process the Root R
(b) Traverse the left subtree of R in procedure
(c) Traverse the right subtree of R in preorder
In order is:
(a) (2) (1) (3) (b) (3) (2) (1)
(c) (1) (2) (3) (d) None of the above ()
26. Recursion may be implemented by means of:
(a) Stacks
(b) Binary Tree
(c) Queue
(d) None of the above ()

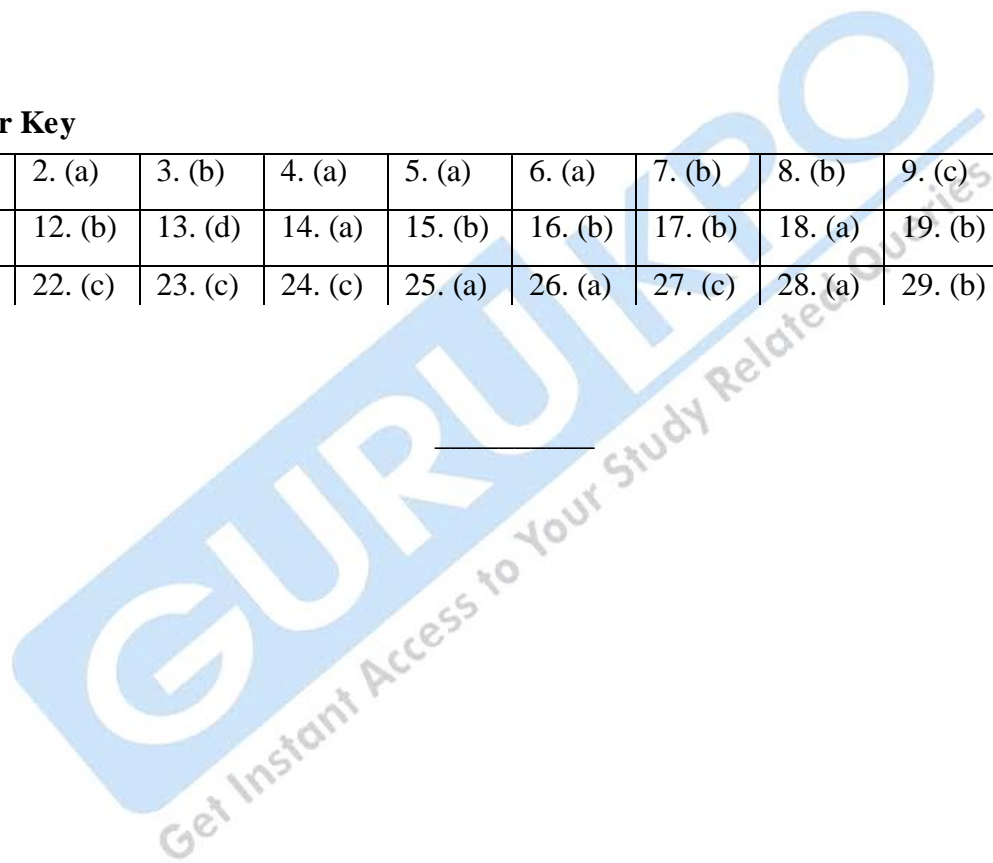
27. An array PTR is called a pointer array if each element of a PTR is a:
(a) Integer
(b) Null element
(c) Pointer
(d) None of the above ()
28. Two condition of Binary Search algorithm are:
(a) the list must be sorted and one must have direct access to the middle element in any sub list
(b) the list must be sorted and one must have direct access to the last element in any sub list
(c) only list is sorted
(d) None of the above ()
29. Variables that can be accessed by all program modules are called:
(a) Private variables
(b) Global variables
(c) Static variables
(d) None of the above ()
30. Liner array is a :
(a) List of finite number n of heterogeneous data elements
(b) List of finite number, n of homogenous data elements
(c) (a) and (b) both
(d) None of the above ()
31. How many null branches are there in a binary tree with 20 nodes:
(a) Zero
(b) Thirty
(c) Twenty one
(d) None of the above ()
32. Equivalent Prefix notations for the expression:
 $((A+B)*C (D-E)^{(F+G)})$ is :
(a) $\wedge_ *+ABC-DE+FG$
(b) $*_ \wedge+ABC-DE+FG$
(c) $\wedge+*_ABC-DE+FG$
(d) None of the above ()
33. Queue are also called:

- (a) LIFO lists
 - (b) FIFO lists
 - (c) Linked lists
 - (d) None of the above ()
34. Quick sort is an algorithm of the :
- (a) Modern type
 - (b) Divide and conquer type
 - (c) Slower efficiency
 - (d) None of the above ()
35. Deletion of elements in a queue can take place at:
- (a) both end
 - (b) one end
 - (c) in the middle
 - (d) None of the above ()
36. An input restricted deque is the one that allows:
- (a) insertion at only one end of the list but allows deletions at both of the list
 - (b) insertions at both ends of the list but allows deletion at only one end of the list
 - (c) uncertain insertion and deletion of elements at both ends
 - (d) None of the above ()
37. A binary tree T is said to be a 2-tree or an extended binary tree if each node N:
- (a) has either 0 or 2 children
 - (b) has maximum number of children
 - (c) has odd number of children
 - (d) None of the above ()
38. Data items that are divided into sub items are called:
- (a) Elementary item
 - (b) Group item
 - (c) (a) and (b) Both
 - (d) None of the above ()
39. DELETE (ABCDEFGH, 4,2) results into:
- (a) ABCDG
 - (b) ABCD
 - (c) ACDEFG
 - (d) None of the above ()

40. Complexity of searching algorithm measured in terms of :
- (a) the number $f(n)$ of comparisons required to find item in DATA where DATA contain n elements
 - (b) $O(\log_2 n)$ comparisons
 - (c) $O(n^2)$ comparisons
 - (d) None of the above
- ()

Answer Key

1. (c)	2. (a)	3. (b)	4. (a)	5. (a)	6. (a)	7. (b)	8. (b)	9. (c)	10. (c)
11. (d)	12. (b)	13. (d)	14. (a)	15. (b)	16. (b)	17. (b)	18. (a)	19. (b)	20. (b)
21. (d)	22. (c)	23. (c)	24. (c)	25. (a)	26. (a)	27. (c)	28. (a)	29. (b)	30. (b)



DESCRIPTIVE PART – II**Year- 2010*****Time allowed: 2 Hours******Maximum Marks : 30*****Attempt any four descriptive type questions out of the six. All questions carry 7½ marks each.**

- Q.1 (a) What is pseudo code? How can we analyze various algorithms?
(b) What do you mean by abstract data type? Explain abstract data types and C++ classes.
- Q.2 (a) Explain string functions for inserting and deleting string from the text.
(b) Describe the pattern matching algorithm with suitable examples.
- Q.3 (a) What is dequeue? Describe the chief characteristics of dequeue.
(b) Explain PUSH and POP operations of the stack with suitable examples. What is over flow condition in it?
- Q.4 (a) What is linear search? Compare the linear search method with Binary Search Method.
(b) Explain Bubble Sort Algorithm. Sort the following list using bubble sort.
61,8,32,53,81,64.
- Q.5 (a) What is a Hash table? Explain the significance of it.
(b) What is priority queue? Explain Huffman's algorithm.
- Q. 6 Write short notes on (any three) of the following:
(a) Representation of Binary tree in memory.
(b) Merge sort (example)
(c) Warshall's algorithm
(d) Quick sort (example)
(e) Linked list
-

ALGORITHMS AND DATA STRUCTURES

OBJECTIVE PART- I

Year - 2009

Time allowed : One Hour

Maximum Marks : 20

The question paper contains to 40 multiple choice questions with four choices and student will have to pick the correct one (each carrying $\frac{1}{2}$ mark).

1. The process of arranging data in increasing order is called:
(a) Sorting (b) Data Arrangement
(c) Merging (d) Indexing ()
2. What is the complexity of linear Search?
(a) $O(\log_2 n)$ (b) $O(n)$
(c) $O(n \log_2 n)$ (d) $O(n^2)$ ()
3. Complexity measures are:
(a) time
(b) speed
(c) both (a) & (b)
(d) None of the above ()
4. Which of the following is LIFO structure?
(a) Stack (b) Queue
(c) Tree (d) Graph ()
5. What is the lower bound of an array in C?
(a) 1 (b) 0
(c) Garbage (d) -1 ()
6. The elements of an array are accessed by:
(a) Accessing function in built – in data structure
(b) Mathematical function
(c) Index
(d) None of the above ()
7. Array is:

- (a) Linear Data Structure
(b) Non Linear Data Structure
(c) Complex Data Structure
(d) None of the above ()
8. A dynamically allocated memory can be returned to the system by using :
(a) malloc () (b) Calloc ()
(c) realloc () (d) free () ()
9. If `char * name = "Dishita";` statement is executed successfully, then what will be the value of `* name`?
(a) D
(b) Dishita
(c) Garbage
(d) None of the above ()
10. String Concatenate means:
(a) Copying one string to another
(b) Adding one string at end of the other
(c) Adding one string at beginning of the other
(d) None of the above ()
11. What is the minimum number of field with each node of doubly linked list?
(a) 1 (b) 2
(c) 3 (d) 4 ()
12. The address field of a linked list:
(a) Contain address of the next node
(b) Contain address of the next pointer
(c) May contain NULL address
(d) Both (a) and (c) ()
13. It is appropriate to represent a queue is:
(a) A circular list (b) Doubly linked list
(c) Linear linked list (d) Array ()
14. NULL pointer is used to tell:
(a) End of linked list
(b) Empty pointer field of a structure
(c) the linked list is empty

- (d) All of the above ()
15. The operations push () and pop () are associated with:
(a) Stack
(b) Queue
(c) Tree
(d) All of the above ()
16. The infix expression $A + (B - C) * D$ is correctly represented in prefix notation is:
(a) $A + B - C * D$ (b) $+A * -BCD$
(c) $ABC - D * +$ (d) $A + BC - D * ()$
17. A linear list of elements in which deletion can be done from one end and insertion can take place only at the other end is known is:
(a) Queue (b) Stacks
(c) Tree (d) Branch ()
18. A binary search tree is generated by inserting in order the following integers:
50,15,62,5,20,58,91,3,8,37,60,24,
The number of nodes in the left subtree and right subtree of the root respectively is:
(a) 4,7 (b) 7,4
(c) 8,3 (d) 3,8 ()
19. In the balanced binary search tree:
(a) Balance factor each node is either -1, 0 or 1
(b) Balance factor of each node is 0
(c) Balance factor of each node is either -2, -1, 0, 1, 2
(d) A binary search tree is always balanced ()
20. In which traversal algorithm, the items are printed in Ascending order?
(a) Preorder
(b) Post order
(c) In order
(d) All of the above ()
21. The heap (represented by an array) constructed from the list of number :
30,10,80,60,15,55,17, is :
(a) 60,80,,55,30,10,17,15
(b) 80,55,60,15,10,30,17
(c) 80,60,30,17,55,15,10,

- (d) None of the above ()
22. Breadth first search :
(a) Scans all incident edges before moving to other vertex
(b) Scans all adjacent unvested vertex as soon as possible
(c) Is same as backtracking
(d) None of the above ()
23. Which method of traversal does not use stack to hold nodes that are waiting to be processed >
(a) Breadth First
(b) Depth first
(c) D-search
(d) None of the above ()
24. A binary tree with nodes has.....number of Null links.
(a) $n + 1$ (b) $2n$
(c) $n + 2$ (d) $n - 1$ ()
25. How many cycles should be contained in a tree?
(a) 0
(b) At least 1
(c) Any number
(d) None of the above ()
26. A balanced binary tree is a binary tree in which the height of the two subtree of every node never differs by more than:
(a) 2
(b) 1
(c) 3
(d) None of the above ()
27. In which of the following tree must balance of each node be either 1, -1 or 0?
(a) Threaded tree
(b) Lexical ordered binary tree
(c) AV tree
(d) None of the above ()
28. Which of the following abstract data types can be used to represent a many to many relation?

- (a) Tree, only (b) Plex, only
(c) Graph, only (d) Both B and A ()
29. Graphic can be implemented using:
(i) Arrays
(ii) Linked list
(iii) Stack
(iv) Queue
(a) (i), (ii) and (iv) (b) (i), (ii) and (iii)
(c) (ii) and (iii) (d) (i) and (ii) ()
30. Adjacency matrix of a graph is:
(a) Identity matrix
(b) Symmetric matrix
(c) Asymmetric matrix
(d) None of the above ()
31. A technique which collects all detected space in free storage list is called:
(a) Static memory allocation
(b) Garbage collection
(c) Dynamic memory
(d) None of the above ()
32. If function DELETE (AAA BBB', 2,2) runs, result will be:
(a) AABB
(b) AB BB
(c) AAAB
(d) None of the above ()
33. Header of linked list is a special node at the :
(a) Middle of the list (b) Beginning of the list
(c) End of the linked list (d) Both (b) and (c) ()
34. Divide and Conquer algorithm may be viewed as a:
(a) Recursive procedure
(b) Iterative procedure
(c) Both of the above
(d) None of the above ()
35. A full binary tree with a non-leaf nodes contains:

- (a) $2n + 1$ nodes (b) $n + 1$ nodes
(c) $2n + 5$ nodes (d) $\log_2 n$ nodes ()
36. A connected graph G is a Euler graph if and only if all vertices are known as:
(a) Same degree (b) Different degree
(c) Odd degree (d) Even degree ()
37. A vertex of degree one is called as:
(a) Isolated vertex (b) Pendant vertex
(c) Colored vertex (d) Null vertex ()
38. Using arrays most efficient implementation of Queue is as:
(a) Linear queue
(b) Circular queue
(c) Priority queue
(d) None of the above ()
39. Traversing means:
(a) Accessing and Processing each record exactly once
(b) Arranging data in some given order
(c) Finding the location of the record with a given key
(d) None of the above ()
40. Stack is:
(a) Static data structure
(b) Inbuilt data structure
(c) Dynamic data structure
(d) None of the above ()

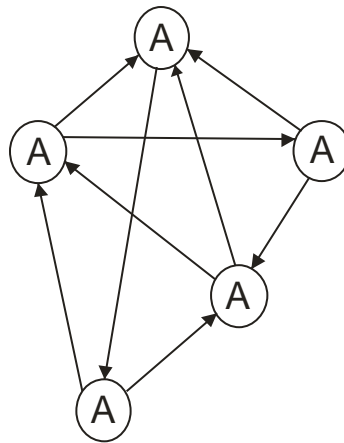
Answer Key

1. (a)	2. (b)	3. (c)	4. (a)	5. (b)	6. (c)	7. (a)	8. (a)	9. (a)	10. (b)
11. (c)	12. (a)	13. (c)	14. (a)	15. (a)	16. (b)	17. (a)	18. (b)	19. (a)	20. (c)
21. (c)	22. (b)	23. (a)	24. (a)	25. (a)	26. (b)	27. (c)	28. (c)	29. (d)	30. (b)
31. (b)	32. (b)	33. (b)	34. (c)	35. (d)	36. (d)	37. (b)	38. (a)	39. (a)	40. (b)

DESCRIPTIVE PART - II**Year- 2009****Time allowed: 2 Hours****Maximum Marks : 30**

Attempt any four descriptive type questions out of the six. All questions carry 7½ marks each.

- Q.1 (a) What do you mean by an algorithm? How do you measure the efficiency of the algorithm? Explain?
(b) What do you understand by multidimensional array? How will you assign the address of an array into a pointer and now using pointer how can you access the values in the array? Explain taking suitable example.
- Q.2 (a) What is a doubly linked list? Why do we need it? Explain the insertion and deletion in a doubly linked list taking suitable example.
(b) What is stack? Explain push and pop operations mentioning the overflow and underflow condition and taking suitable example.
- Q.3 (a) What is Recursion? Explain it by giving the example of binary search algorithm.
(b) Convert the following expression into postfix by using a stack and then evaluate the postfix expression by using another stack?
 $32 / (14 - 6) + 4 * (6 + 16) - 7$
- Q.4 (a) Explain the traversal algorithms for a binary tree by taking suitable example.
(b) Sort the following list by using heap sort algorithm.
52, 7, 41, 72, 23, 92, 48, 15
- Q.5 (a) Explain the linked representation of the following graph:



- (b) What do you mean by Hashing? What is Hash Collision? How do you recover from the hash collision? Explain your answer by giving a suitable example?

Q.6 Write short notes on any three of the following.

- (a) Data Abstraction
- (b) Priority Queue
- (c) Quick Sort (Example)
- (d) Radix Sort (example)
- (e) Threaded Binary Tree (Threads)

ALGORITHMS AND DATA STRUCTURES

OBJECTIVE PART- I

Year - 2008

Time allowed : One Hour

Maximum Marks : 20

The question paper contains to 40 multiple choice questions with four choices and student will have to pick the correct one (each carrying $\frac{1}{2}$ mark).

1. Efficiently of an algorithm is measured by:
(a) time used (b) space used
(c) A and B (d) None ()
2. The pointer of the last node in a linked list contains:
(a) Data item (b) Null value
(c) Address (d) None ()
3. A node contains :
(a) Information and link field
(b) Information and data item
(c) Address and link field
(d) All of the above ()
4. Operating system periodically collect all the deleted space into the free storage list is called:
(a) Fragmentation
(b) Garbage collection
(c) Overflow
(d) Underflow ()
5. Overflow means:
(a) NO empty space available
(b) No item is available
(c) Error
(d) none ()
6. The pointer available in two way list node:
(a) INFO, FORW, BACK

- (b) INFO, FORM, REVE
(c) INFO, FRONT, REAR
(d) None ()
7. Stack is also called:
(a) First in first out (b) Last in first out
(c) First in last out (d) Last in last out ()
8. Data structure that take insertion and deletion only at beginning or the end, not in the middle:
(a) Linked list and linear array
(b) Stack and queues
(c) Stack and linked list
(d) Linked list and queues ()
9. The element insert in stack through:
(a) POP (b) PUSH
(c) FIFO (d) LIFO ()
10. Which one shows that STACK is empty :
(a) $TOP = 0$ or $TOP = N$
(b) $TOP = 0$ or $TOP = NULL$
(c) $TOP = N$ or $TOP = NULL$
(d) None ()
11. Which one is not in infix notation:
(a) $A+B$ (b) $C-D$
(c) G/H (d) $+AB$ ()
12. is based on 'Divide and Conquer' paradigm:
(a) Merge sort
(b) Quick sort
(c) Heap sort
(d) All of the above ()
13. Complexity of quick sort :
(a) $n \log_2 n$ (b) $\log n$
(c) $\log n \cdot n$ (d) $n-1$ ()

14. A is a linear list of elements in which deletion can take place only at one end and insertion at other end?
(a) Stack (b) Linked list
(c) Queue (d) Tree ()
15. Dequeue stands for :
(a) Double –single queue
(b) Double – ended queue
(c) Double – circular queue
(d) None ()
16. Which one is non linear data structure :
(a) Linked (b) Stack
(c) Queue (d) Tree ()
17. Any node N in binary tree has either.....successor:
(a) 0,1,2 (b) 1,2 N
(c) 0,3,5 (d) All ()
18. A tree becomes – tree :
(a) If each node N has either 0 or 2 children
(b) Each node N has N children
(c) Each node has no children
(d) None ()
19. The depth of the complete tree T_n with n nodes is given by:
(a) $D_n = [\log_2 n + 1]$ (b) $D_n = [\log n + 1]$
(c) $D_n = [\log_n 2 + 1]$ (d) None ()
20. Threads are related to:
(a) Linked list (b) Stack
(c) Queues (d) Tree ()
21. Heap is related to:
(a) Linked list (b) Stack
(c) Tree (d) Queue ()
22. Graphs can be represented by:
(a) $G(V, E)$ (b) $G = (B, C)$
(c) $G = (A, B)$ (d) $G = (A, C)$ ()

23. Complexity of bubble sort is:

(a) $O(n^2)$

(c) $\log n$

(b) $O(n \log_2 n)$

(d) $O(n)$

()

24. Complexity of Heapsort is:

(a) $O(n \log_2 n)$

(c) $\log n$

(b) $O(n)$

(d) None

()

25. An edge e is calledif it has identical endpoints.

(a) Loop

(c) Forest

(b) Multigraph

(d) Extended tree

()

26. What will be the length of string 'string':

(a) 6

(c) 8

(b) 7

(d) None

()

27. A class is a :

(a) Abstract data type

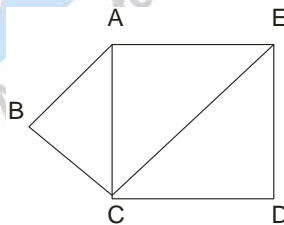
(b) User defined data type

(c) Binding of data and member function

(d) All of the above

()

28. The following is an example of:



(a) Graph

(c) Tree

(b) Multigraph

(d) Weighted graph

()

29. Which one is the path of length 2 (refer Fig- A)

(a) BAD

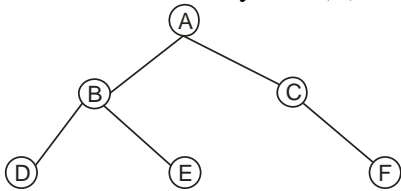
(c) BCDE

(b) BCE

(d) BAED

()

30. A 4 – cycles in the graph (refer Fig A) :

31. Degree of C i.e. $\deg(c)$ is (refer fig A) :
 (a) ABCEA (b) CDEA
 (c) BADC (d) None ()
32. Which one is not a path (refer Fig A) :
 (a) BAD (b) BAC
 (c) ABC (d) CDE ()
33. Consider the binary tree (T) :
- 
- ```

graph TD
 A((A)) --- B((B))
 A --- C((C))
 B --- D((D))
 B --- E((E))
 C --- F((F))

```
- (a) DBEACF (b) ABDECF  
 (c) DEBFCA (d) None ( )
34. The in order traversal of tree (refer Fig B) :  
 (a) DBEACF (b) DBEACF  
 (c) DEBFCA (d) None ( )
35. The post order traversal of tree (tree Fig B)  
 (a) DEBFCA (b) DBEACF  
 (c) ABDECF (d) NONE ( )
36. Which of the following is a graph traversal method:  
 (a) BFS (b) DFS  
 (c) Both (d) None ( )
37. A program is made up of:  
 (a) constants  
 (b) variables  
 (c) instructions  
 (d) all of the above ( )
38. Which of the following is a flowchart symbol:  
 (a) Decision (b) Flow lines

- (c) Both A & B (d) None ( )
39. Recursion means:  
(a) function calling itself (b) Subroutine  
(c) Null function (d) None ( )
40. A binary tree node that has no children is called:  
(a) Leaf node (b) Root node  
(c) Non leaf node (d) None ( )

**Answer Key**

|         |         |         |         |         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| 1. c)   | 2. (b)  | 3. (a)  | 4. (b)  | 5. (a)  | 6. (a)  | 7. (b)  | 8. (b)  | 9. (b)  | 10. (b) |
| 11. (d) | 12. (b) | 13. (a) | 14. (c) | 15. (b) | 16. (d) | 17. (a) | 18. (a) | 19. (a) | 20. (d) |
| 21. (c) | 22. (a) | 23. (a) | 24. (a) | 25. (a) | 26. (a) | 27. (d) | 28. (a) | 29. (b) | 30. (a) |
| 31. (c) | 32. (a) | 33. (b) | 34. (a) | 35. (a) | 36. (c) | 37. (d) | 38. (c) | 39. (a) | 40. (a) |

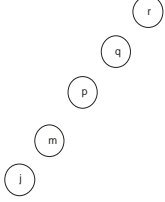
**DESCRIPTIVE PART - II****Year- 2008***Time allowed: 2 Hours**Maximum Marks : 30**Attempt any four descriptive type questions out of the six. All questions carry 7½ marks each.*

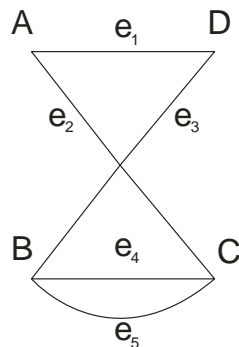
- Q.1 (a) What are the criteria to measure the efficiency of any algorithm ?  
(b) Write the pseudo code with flowchart diagram for calculating factorial?
- Q.2 (a) Explain time analysis space complexity.  
(b) Define the following in brief :  
(i) Sequential logic  
(ii) Selection logic  
(iii) Iteration logic
- Q.3 (a) What are the difference between external and internal sorting?  
(b) Explain briefly using a simple example the logic of the quick sort algorithm. Write a recursive a algorithm for quick sort and sort how quick sort would sort the array.
- Q.4 (a) Discuss linear and non linear implementation of queues.  
(b) Write an algorithm for inserting an item into a linked list.
- Q.5 (a) What is binary tree? Consider a following list and insert an item in order into an empty binary search tree?  
(b) Describe an algorithm for find a minimum spanning tree T of a weighted graph G.
- Q.6 Write short notes on the following :  
(i) Binary search  
(ii) Warshall's Algorithm  
(iii) Hashed searching
-

**ALGORITHMS AND DATA STRUCTURES****OBJECTIVE PART- I****Year - 2007*****Time allowed : One Hour******Maximum Marks : 20***

**The question paper contains to 40 multiple choice questions with four choices and student will have to pick the correct one (each carrying  $\frac{1}{2}$  mark).**

1. Stack is also known as:  
(a) LIFO system  
(b) FIFO system  
(c) FIFO LIFO system  
(d) None of the system ( )
2. A binary tree node that has no children is called:  
(a) Leaf node  
(b) Root Node  
(c) Non leaf node  
(d) None of the above ( )
3. Which of the following sorting algorithm is based on the 'Divide and Conquer' paradigm?  
(a) Quick sort  
(b) Merge Sort  
(c) Heap Sort  
(d) All of the above ( )
4. The collection of same type of data is called:  
(a) A union  
(b) A structure  
(c) A graph  
(d) None of the above ( )
5. The process of accession data stored in a tape is similar to manipulating data on a:  
(a) Stack (b) Queue  
(c) List (d) Heap ( )

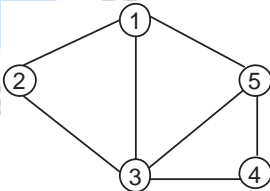
6. The initial configuration of a queue is P,Q, R,S (P is the front end). To the configuration S,R,Q one needs a minimum of:
- 2 addition and 3 deletion
  - 3 addition and 3 deletion
  - 3 addition and 4 deletion
  - 3 addition and 2 deletion
- ( )
7. The depth  $n$  of the complete binary tree in with  $n$  nodes is gives by:
- $\log_2 (n+1) - 1$
  - $\log_2 n+1$
  - $\log_2 (n-1) + 1$
  - $\log_2 n$
- ( )
8. What will be the expression for the following binary tree?
- $E = (a-b)/((c*d)+e)$
  - $E = a-b/c * d + e$
  - $E = a - (b/c * d) + e$
  - $E = (a-b/c) * (d+e)$
- ( )
9. One of the more popular balanced trees was introduced in 1962 by adelson-velski and Landis is known as:
- AVL Tree
  - B Tree
  - M-way search tree
  - None of the above
- ( )
10. The following is an example of:
- 
- Skewed binary search tree
  - Binary tree
  - AVL search tree
  - Binary search tree
- ( )
11. The following figure represents:



- (a) Directed graph  
(b) Multigraph  
(c) AVL tree  
(d) None of the above ( )
12. The operation of finding the location of a given item in a collection of items is called:  
(a) Sorting  
(b) Searching  
(c) Listing  
(d) None of the above ( )
13. Worst case complexity of quick sort algorithm is:  
(a)  $O(n^2)$   
(b)  $O(n \log n)$   
(c)  $O(\log n)$   
(d) None of those ( )
14. Average case complexity of heap sort algorithm is:  
(a)  $O(n \log n)$   
(b)  $O(n^2)$   
(c)  $O(n)$   
(d)  $O(n \log_2 n)$  ( )
15. Hashing or has addressing is a technique of:  
(a) Searching  
(b) Sorting  
(c) Both (a) and (b)  
(d) None of the above ( )
16. The notation in which operation symbol is placed before its two operands, is called:  
(a) Infix notation



- (b) Polish notation  
(c) Suffix notation  
(d) None of the above ( )
17. ....is the term used to delete an element from a stack.  
(a) PUSH (b) POP  
(c) DEL (d) Both B and C ( )
18. When a called function in turn calls another function a process of chaining occurs. A special case of this process, where a function calls itself is called.  
(a) Recursion (b) Deletion  
(c) Insertion (d) Overloading ( )
19. Sparse matrices have:  
(a) Many zero elements  
(b) Many non zero elements  
(c) Higher dimension  
(d) None of the above ( )
20. Length of the string "Manisha" is :  
(a) 7  
(b) 8  
(c) either 7 or 8  
(d) None of the above ( )
21. What will be the results of insert (' ABCDEFG',3 XYZ)?  
(a) ABCDEFGXYZ  
(b) ABXYZCDEFG  
(c) ABCXYZDEFG  
(d) None of the above ( )
22. The string with zero characters is called:  
(a) Empty string (b) Null string  
(c) Full string (d) Both A and B ( )
23. The variables which can be accessed only within a particular program or subprogram are known as:  
(a) Local variables  
(b) Global variable  
(c) Auto variables

- (d) External variables ( )
24. Which of the following is a data structure?  
(a) Array  
(b) Linked list  
(c) Tree  
(d) All of the above ( )
25. A collection of related data-items or fields or attributes is called a:  
(a) Record  
(b) File  
(c) Database  
(d) None of the above ( )
26. Which of the following is a graph traversal method?  
(a) BFS  
(b) DFS  
(c) Both BFS and DFS  
(d) None of the above ( )
27. Previously allocated memory returned to the system by using the function:  
(a) malloc ( ) (b) calloc ( )  
(c) free ( ) (d) realloc ( ) ( )
28. The following figure represents :  
  
(a) Directed graph (b) Undirected graph  
(c) Unconnected graph (d) AVL tree ( )
29. A special list maintained with the linked list in memory, which consists of unused memory cells and has its own pointer is called:  
(a) List of available space  
(b) Free storage list  
(c) Free pool  
(d) All of the above ( )

30. If every edge in the graph is assigned some data, it is called:  
(a) Multi graph (b) Directed graph  
(c) Tree (d) Weighted graph ( )
31. What is the minimum number of fields with each elements of a doubly linked list?  
(a) 1 (b) 2  
(c) 3 (d) 4 ( )
32. Character data types are represented by the word?  
(a) int (b) float  
(c) char (d) ch ( )
33. The running time  $T(n)$ , where 'n' is the input size of recursive algorithm is given as follows:  $T(n) = c + T(n-1)$  ; if  $n > 1$ , )  
 $D = 1$ , if  $n \leq 1$   
The order of algorithm is:  
(a)  $n^2$  (b)  $n$   
(c)  $n^3$  (d)  $n^n$  ( )
34. If we use a 16-bit word length, the maximum size of the integer value is:  
(a)  $2^{16}-1$  (b)  $2^{15}-1$   
(c)  $2^{19}-1$  (d)  $2^{15}$  ( )
35. A linear list of elements in which deletions can take place only at one end and insertions can take place only at other end is called:  
(a) Stack (b) Queue  
(c) Deque (d) Linked list ( )
36. Sometimes new data are to be inserted into a data structure but there is no available space i.e. the free storage list is empty. This situation is usually called:  
(a) Underflow  
(b) Overflow  
(c) Overflow  
(d) None of the above ( )
37. Which of the following is not a sorting technique?  
(a) Bubble (b) Binary  
(c) Radix (d) Insertion ( )
38. The process of memory allocation at run time is known as:

- (a) Dynamic memory allocation  
 (b) Static memory allocation  
 (c) Compaction  
 (d) Fragmentation ( )
39. The following series is known as:  
 0,1,1,2,3,5,8,13,21,34,55.....  
 (a) Fibonacci series (b) Natural number series  
 (c) Compaction (d) Even number series ( )
40. A header list where the last node points back to the header node is called :  
 (a) A grounded header list  
 (b) A circular header list  
 (c) Both (A) and (B)  
 (d) None of the above ( )

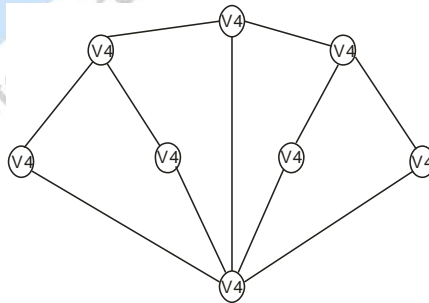
**Answer Key**

|         |         |         |         |         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| 1. (a)  | 2. (a)  | 3. (a)  | 4. (d)  | 5. (b)  | 6. (a)  | 7. (b)  | 8. (a)  | 9. (a)  | 10. (a) |
| 11. (b) | 12. (b) | 13. (a) | 14. (a) | 15. (a) | 16. (b) | 17. (b) | 18. (a) | 19. (a) | 20. (a) |
| 21. (b) | 22. (d) | 23. (a) | 24. (d) | 25. (a) | 26. (c) | 27. (c) | 28. (b) | 29. (d) | 30. (d) |
| 31. (c) | 32. (c) | 33. (a) | 34. (a) | 35. (b) | 36. (b) | 37. (b) | 38. (a) | 39. (a) | 40. (b) |

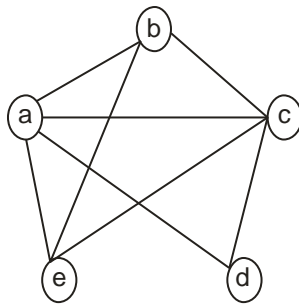
**DESCRIPTIVE PART - II****Year- 2007****Time allowed: 2 Hours****Maximum Marks : 30**

**Attempt any four descriptive type questions out of the six. All questions carry 7½ marks each.**

- Q.1 (a) What is string? Explain various string operations with suitable examples and show how these operations are used in word processing?
- Q.2 (a) Explain array representation of stacks and queues.  
(b) Define infix, postfix and prefix notations giving examples of each.
- Q.3 (a) What is linked list? Give two advantages of linked lists over arrays.  
(b) What is complete binary tree? How is it different from binary tree?
- Q.4 (a) What is merging? Give complexity of the merging algorithm?  
(b) Sort the following array using merge-sort:  
66,33,40,22,55,88,60,11,80,20,50,44,77,30  
Give the complexity of merge-sort algorithm.
- Q.5 (a) What is a graph? Explain depth-first search Algorithm.



- (b) Consider the graph G in the following figure:



Find vertices, edges and degree of each node.

Q.6 Write short notes on any two :

- (a) Sparse matrices;
- (b) Variables;
- (c) Queue and dequeue;
- (d) Hasing technique.

**GURUKPO**  
Get Instant Access to Your Study Related Queries...

---

---

---

**ALGORITHMS AND DATA STRUCTURES**

---

---

**OBJECTIVE PART- I****Year - 2006*****Time allowed : One Hour******Maximum Marks : 20***

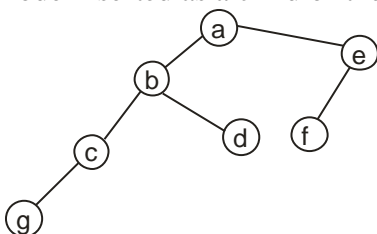
The question paper contains to 40 multiple choice questions with four choices and student will have to pick the correct one. (Each carrying  $\frac{1}{2}$  marks.).

1. FRONT and REAR words are related with:  
(a) Stack  
(b) Queue  
(c) Linked list  
(d) None of the above ( )
2. Queue is also known as:  
(a) LIFO – system  
(b) FIFO system  
(c) LIFO FIFO  
(d) None of the above ( )
3. For a sequential search, the average number of comparisons for a file with records is:  
(a)  $(n+1)/2$  (b)  $\log_2 n$   
(c)  $n^2$  (d)  $n/2$  ( )
4. A data structure, in which an element is added and removed only from one end is known is:  
(a) Queue  
(b) Stack  
(c) Array  
(d) None of the above ( )
5. Header of a linked list a special node at the:  
(a) End of the linked list  
(b) Middle of the list  
(c) Beginning of the list



- (d) None of the above ( )
6. Adjacency matrix for a graph is:  
(a) Unimarix  
(b) Symmetric  
(c) Asymmetric  
(d) None of the above ( )
7. What is the minimum number of fields with each element of a doubly linked list?  
(a) 1  
(b) 2  
(c) 3  
(d) 4 ( )
8. The collection of same type of data is called:  
(a) An array  
(b) A union  
(c) A structure  
(d) None of the above ( )
9. In a linear linked list, a node contains at least:  
(a) Node address field and next pointer field  
(b) Node number and data field  
(c) An information field and next pointer field  
(d) None of the above ( )
10. Which of the following sorting algorithm is based on the idea of "Divide" and conquer?  
(a) Merge sort  
(b) Heap sort  
(c) Both B and A  
(d) None of the above ( )
11. Which of the following is a method of searching?  
(a) Linear search (b) Bubble search  
(c) Insertion search (d) Selection search ( )
12. The element at the root of heap is:  
(a) Largest

- (b) Smallest  
(c) Depends on type of heap  
(d) None of the above ( )
13. Average case time complexity of quick sort algorithm is:  
(a)  $O(n \log n)$   
(b)  $O(\log_2 n)$   
(c)  $O(n^2)$   
(d) None of the above ( )
14. The five items A B C D AND E are pushed in a stack, one after the another starting from A. The stack is popped four times and each elements is inserted in a queue. The two elements are deleted from the queue and pushed back on the stack. Now one item is popped from the stack. The popped items is:  
(a) A (b) B  
(c) C (d) D ( )
15. Malloc function returns a NULL when:  
(a) Memory is successfully allocated  
(b) All memory is cleared  
(c) Space is insufficient to satisfy the request  
(d) None of the above ( )
16. The process memory allocation at run time is known as:  
(a) Dynamic memory allocation (b) Static memory allocation  
(c) Compaction (d) fragmentation ( )
17. Which of the following is not a type of tree?  
(a) Binary (b) Binary Search  
(c) AVL (d) Insertion ( )
18. In which tree for every node the height of its left and right sub tree differ at last by one?  
(a) Binary search tree  
(b) AVL tree  
(c) Complete Tree  
(d) None of the above ( )
19. Which of the following is data structure operation?  
(a) Searching

- (b) Sorting  
(c) Traversing  
(d) All of the above ( )
20. When of the following in turn calls another function a process of 'chaining' occurs. A special case of this process, where a function calls itself is called:  
(a) Recursion (b) Delection  
(c) Insertion (d) Overloading ( )
21. Which of the following abstract data types can be used to represent a many to many relation:  
(a) Tree  
(b) Graph  
(c) Both A and B  
(d) None of the above ( )
22. In the balanced binary tree given below, how many nodes become unbalanced when a node inserted as a child of the node "g"
- 
- (a) 1 (b) 3  
(c) 7 (d) 8 ( )
23. Which of the following is a non linear data structure?  
(a) Tree only (b) Graph only  
(c) Array (d) Both tree and graph ( )
24. If we use a 16- bit word length, the size of the integer value is limited to the range:  
(a)  $-2^6$  to  $2^6-1$   
(b)  $-2^{10}$  to  $2^{10}-1$   
(c)  $2^{15}$   
(d)  $-2^{15}$  ( )
25. Floating point data types are represented by the word:  
(a) int (b) floating  
(c) float (d) char ( )

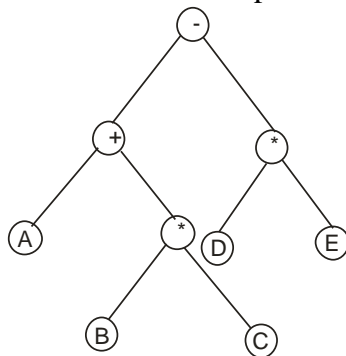
26. If there are  $n$  vertices in the graph then how many edges are needed to construct a minimum spanning tree?

(a)  $n$  (b)  $n+1$   
 (c)  $n-1$  (d)  $n^2$  ( )

27. Which of the following is an advantage of using pointers:

(a) Pointers reduce the length and complexity of a program  
 (b) They increase the execution speed  
 (c) Pointers are more efficient in handling the data tables  
 (d) All of the above ( )

28. What will be the expression for the following tree?



(a)  $(A + (B * C)) - (D * E)$   
 (b)  $(A+B) * C - DE$   
 (c)  $ABC + *(D * E)$   
 (d) None of the above ( )

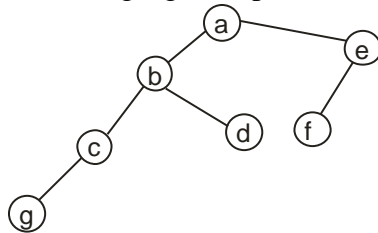
29. A connection between two vertices is called:

(a) edge  
 (b) vertex  
 (c) tree  
 (d) none of the above ( )

30. A node that has no children is called:

(a) lead node  
 (b) root node  
 (c) parent node  
 (d) none of the above ( )

31. Following figure represents:



- (a) Directed Graph (b) Undirected graph  
(c) Unconnected graph (d) AVL tree ( )
32. The fundamental operations used on a stack are:  
(a) PUSH  
(b) POP  
(c) Both A and B  
(d) None of the above ( )
33. A linear list in which elements can be added or removed at either end but not in the middle is called:  
(a) Queue (b) Dequeue  
(c) Stack (d) Linked list ( )
34.  $\text{FRONT} := \text{REAR} := \text{NULL}$ , refers to empty  
(a) Stack (b) Queue  
(c) Array (d) Linked ( )
35. Worst case complexity of heap sort is:  
(a)  $O(n^2)$  (b)  $O(n \log_2 n)$   
(c)  $o(n)$  (d)  $O(\log_2 n)$  ( )
36. Which of the following algorithm have worst case complexity as  $O(n^2)$ ?  
(a) Insertion sort  
(b) Bubble sort  
(c) Quick sort  
(d) All of the above ( )
37. Which of the following is a hashing technique?  
(a) Division method  
(b) Med square method  
(c) Folding method

- (d) All of the above ( )
38. POP is the term used to delete an elements from a :  
 (a) Stack  
 (b) Queue  
 (c) Linked list  
 (d) Tree ( )
39. When new data are to be inserted into a data structure but there is no available space, this situation is called :  
 (a) Overflow (b) Underflow  
 (c) Compaction (d) Fragmentation ( )
40. The variables which can be accessed by all modules in a program, are known as:  
 (a) Local variables (b) Internal variables  
 (c) Global variables (d) Auto variables ( )

**Answer Key**

|         |         |         |         |         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| 1. (b)  | 2. (b)  | 3. (d)  | 4. (b)  | 5. (c)  | 6. (b)  | 7. (c)  | 8. (a)  | 9. (c)  | 10. (d) |
| 11. (a) | 12. (c) | 13. (a) | 14. (d) | 15. (c) | 16. (a) | 17. (d) | 18. (b) | 19. (d) | 20. (a) |
| 21. (b) | 22. (b) | 23. (d) | 24. (a) | 25. (c) | 26. (c) | 27. (d) | 28. (a) | 29. (a) | 30. (d) |
| 31. (a) | 32. (c) | 33. (b) | 34. (b) | 35. (b) | 36. (d) | 37. (d) | 38. (a) | 39. (a) | 40. (c) |

**DESCRIPTIVE PART - II****Year- 2006***Time allowed: 2 Hours**Maximum Marks : 30*

**Attempt any four descriptive type questions out of the six. All questions carry 7½ marks each.**

- Q.1 (a) What is data structure?
- (b) Define array. Discuss the representation of linear arrays in memory and give any two advantage of linked over arrays.
- Q.2 What is a tree? Explain any four types of trees with the help of suitable examples.
- Q.3 (a) Consider the graph of the following figure.  
Perform a breadth first search (BFS) beginning at vertex  $V_1$  list vertices in the order in which they visited.
- (b) Construct a heap H from the following list of number:  
40,30,50,22,60,55,77,55
- Q.4 What is difference between
- (a) Stacks and queues
- (b) local and global variables
- (c) directed and undirected graph
- (d) BFS and DFS.

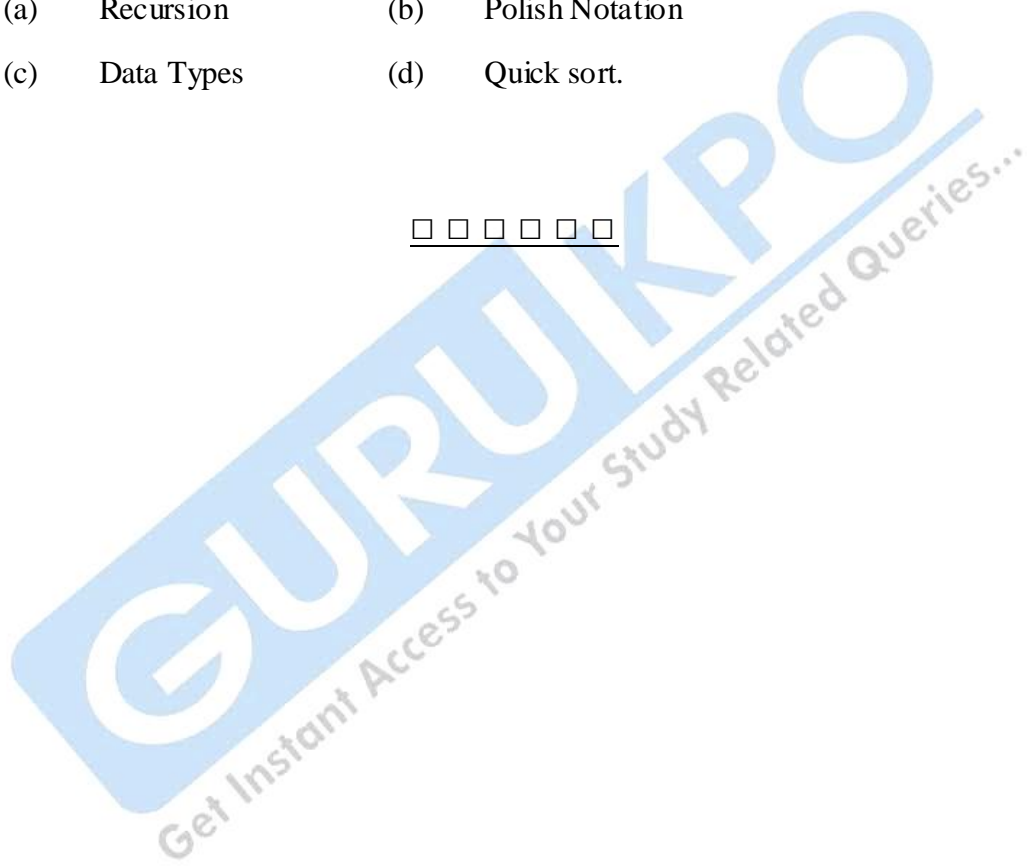


Q.5 Write a short note on various data structure operations and explain binary search algorithm.

Q.6 Write a short note on:

- |                |                     |
|----------------|---------------------|
| (a) Recursion  | (b) Polish Notation |
| (c) Data Types | (d) Quick sort.     |

□ □ □ □ □ □



## MCQ

1. Two main measures for the efficiency of an Algorithm are:  
(a) Processor and Memory (b) Complexity and Capacity  
(c) Time and Space (d) Data and Space (c)
2. The time factor when determining the efficiency of an algorithm is measured by:  
(a) Counting microseconds  
(b) Counting the number of key operations  
(c) Counting the number of statement  
(d) Counting the Kilobytes of algorithm (a)
3. A data structure is a way of:  
(a) arrangement of different data elements  
(b) organizing data with consideration of items stored into it along with their relationship with each other  
(c) ordering of collected data  
(d) None of the above (b)
4. The space factor when determining the efficiency of algorithm is measured by:  
(a) Counting the maximum memory needed by the algorithm  
(b) Counting the minimum memory needed by the algorithm  
(c) Counting the average memory needed by the algorithm  
(d) Counting the maximum disk space needed by the algorithm (a)
5. The de-queue process removes data:  
(a) From the front of the queue  
(b) From the bottom of the queue  
(c) Can not be removed  
(d) None of the above (a)
6. A queue is a :  
(a) Sequential Organization of data  
(b) Listing of data  
(c) Indexing of data  
(d) None of the above (a)

7. ....is a way of grouping things together by placing one thing on top of another and then removing things one at a time from the top.  
(a) Array  
(b) Stack  
(c) Pointer  
(d) All of the above (b)
8. The complexity of Binary Search algorithm is:  
(a)  $O(n)$  (b)  $O(\log n)$   
(c)  $O(n^2)$  (d)  $O(n \log n)$  (b)
9. The complexity of Bubble Sort algorithm is:  
(a)  $O(n)$  (b)  $O(\log n)$   
(c)  $O(n^2)$  (d)  $O(n \log n)$  (c)
10. Each array declaration need not give, implicitly or explicitly, the information about:  
(a) the name of array  
(b) the data type of array  
(c) the first data from the set to be stored  
(d) the index set of the array (c)
11. Which of the following is not linear data structure?  
(a) Arrays  
(b) Linked lists  
(c) Both of the above  
(d) None of the above (d)
12. Finding the location of the element with a given value is:  
(a) Traversal  
(b) Search  
(c) Sort  
(d) None of the above (b)
13. The operation of processing each element in the list is known as:  
(a) Sorting (b) Merging  
(c) Inserting (d) Traversal (d)
14. Arrays are best data structures :  
(a) for relatively permanent collections of data  
(b) for the size of the structure and the data in the structure are constantly changing

- (c) for both of above situation  
(d) for none of the above situation (a)
15. Linked lists are best suited:  
(a) For relatively permanent collections of data  
(b) for the size of the structure and the data in the structure are constantly changing  
(c) for both of above situation  
(d) for none of the above situation (b)
16. In tree construction which is the suitable efficient data structure :  
(a) array (b) linked lists  
(c) stack (d) queue (b)
17. By a schematics character variable we mean:  
(a) A variable whose length is defined before the program is executed and cannot change through the program  
(b) A variable whose length may vary during the execution of the program as long as the length does not exceed a maximum value determined by the program before the program is executed  
(c) (a) and (b) both  
(d) None of the above (b)
18. In this STACKS, "PUSH" is the term used to:  
(a) insert an element into a stack  
(b) delete an element from a stack  
(c) modify the existing element of a stack  
(d) None of the above (b)
19. Maximum number of queues needed to implement the priority queue:  
(a) Three (b) Two  
(c) Five (d) One (a)
20. Polish notation refers to the notations in which:  
(a) the operator symbol is placed after its two operands  
(b) the operator symbol is placed before its two operands  
(c) the operator symbol is placed in the middle  
(d) None of the above (b)
21. A node has .....elements.  
(a) None (b) One

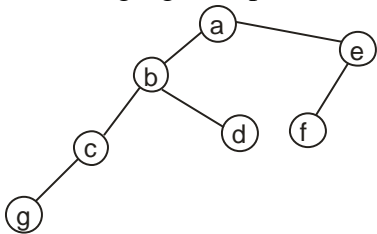
- (c) Two (d) Three (d)
22. Which of the following abstract data types are not used by Integer Abstract Data Type group?  
 (a) Short (b) Int  
 (c) Float (d) Long (c)
23. There are.....standard ways of maintaining a graph G in the memory of a computer.  
 (a) four  
 (b) three  
 (c) two  
 (d) None of the above (c)
24. In the sequential representation of Graph G in computer, it may be difficult to:  
 (a) insert nodes in G  
 (b) delete nodes in G  
 (c) (a) and (b) Both  
 (d) None of the above (c)
25. The three standard ways of traversing a binary tree is:  
 (a) Process the Root R  
 (b) Traverse the left subtree of R in procedure  
 (c) Traverse the right subtree of R in preorder  
 In order is:  
 (a) (2) (1) (3) (b) (3) (2) (1)  
 (c) (1) (2) (3) (d) None of the above (a)
26. Recursion may be implemented by means of:  
 (a) Stacks  
 (b) Binary Tree  
 (c) Queue  
 (d) None of the above (a)
27. An array PTR is called a pointer array if each element of a PTR is a:  
 (a) Integer  
 (b) Null element  
 (c) Pointer  
 (d) None of the above (c)

28. Two condition of Binary Search algorithm are:
- (a) the list must be sorted and one must have direct access to the middle element in any sub list
  - (b) the list must be sorted and one must have direct access to the last element in any sub list
  - (c) only list is sorted
  - (d) None of the above
- (a)
29. Variables that can be accessed by all program modules are called:
- (a) Private variables
  - (b) Global variables
  - (c) Static variables
  - (d) None of the above
- (b)
30. Liner array is a :
- (a) List of finite number n of heterogeneous data elements
  - (b) List of finite number, n of homogenous data elements
  - (c) (a) and (b) both
  - (d) None of the above
- (b)
31. How many null branches are there in a binary tree with 20 nodes:
- (a) Zero
  - (b) Thirty
  - (c) Twenty one
  - (d) None of the above
- (c)
32. Equivalent Prefix notations for the expression:  
 $((A+B)*C (D-E)^{(F+G)})$  is :
- (a)  $^+_*+ABC-DE+FG$
  - (b)  $^+_*+ABC-DE+FG$
  - (c)  $^+_*+ABC-DE+FG$
  - (d) None of the above
- (a)
33. Queue are also called:
- (a) LIFO lists
  - (b) FIFO lists
  - (c) Linked lists
  - (d) None of the above
- (b)

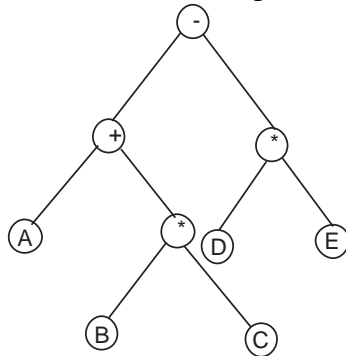
34. Quick sort is an algorithm of the :  
(a) Modern type  
(b) Divide and conquer type  
(c) Slower efficiency  
(d) None of the above (b)
35. Deletion of elements in a queue can take place at:  
(a) both end  
(b) one end  
(c) in the middle  
(d) None of the above (b)
36. An input restricted deque is the one that allows:  
(a) insertion at only one end of the list but allows deletions at both of the list  
(b) insertions at both ends of the list but allows deletion at only one end of the list  
(c) uncertain insertion and deletion of elements at both ends  
(d) None of the above (a)
37. A binary tree T is said to be a 2-tree or an extended binary tree if each node N:  
(a) has either 0 or 2 children  
(b) has maximum number of children  
(c) has odd number of children  
(d) None of the above (a)
38. Data items that are divided into sub items are called:  
(a) Elementary item  
(b) Group item  
(c) (a) and (b) Both  
(d) None of the above (a)
39. DELETE (ABCDEFG, 4,2) results into:  
(a) ABCDG  
(b) ABCD  
(c) ACDEFG  
(d) None of the above (a)
40. Complexity of searching algorithm measured in terms of :  
(a) the number of comparisons required to find item in DATA where DATA contain n elements  
(b)  $O(\log_2 n)$  comparisons



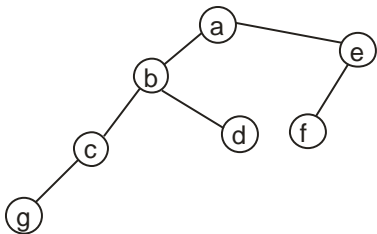
- (c)  $O(n^2)$  comparisons  
(d) None of the above (a)
41. The process of arranging data in increasing order is called:  
(a) Sorting (b) Data Arrangement  
(c) Merging (d) Indexing (a)
42. What is the complexity of linear Search?  
(a)  $O(\log_2 n)$  (b)  $O(n)$   
(c)  $O(n \log_2 n)$  (d)  $O(n^2)$  (b)
43. Complexity measures are:  
(a) time  
(b) speed  
(c) both (a) & (b)  
(d) None of the above (c)
44. Which of the following is LIFO structure?  
(a) Stack (b) Queue  
(c) Tree (d) Graph (a)
45. What is the lower bound of an array in C?  
(a) 1 (b) 0  
(c) Garbage (d) -1 (b)
46. The variables which can be accessed by all modules in a program, are known as:  
(a) Local variables (b) Internal variables  
(c) Global variables (d) Auto variables (c)
47. When new data are to be inserted into a data structure but there is no available space, this situation is called :  
(a) Overflow (b) Underflow  
(c) Compaction (d) Fragmentation (a)
48. POP is the term used to delete an elements from a :  
(a) Stack  
(b) Queue  
(c) Linked list  
(d) Tree (a)
49. Which of the following is a hashing technique?  
(a) Division method  
(b) Med square method  
(c) Folding method  
(d) All of the above (d)
50. Which of the following algorithm have worst case complexity as  $O(n^2)$ ?  
(a) Insertion sort  
(b) Bubble sort  
(c) Quick sort

- (d) All of the above (d)
51. Worst case complexity of heap sort is:  
 (a)  $O(n^2)$  (b)  $O(n \log_2 n)$   
 (c)  $o(n)$  (d)  $O(\log_2 n)$  (b)
52. FRONT := REAR := NULL, refers to empty  
 (a) Stack (b) Queue  
 (c) Array (d) Linked (b)
53. The fundamental operations used on a stack are:  
 (a) PUSH  
 (b) POP  
 (c) Both A and B  
 (d) None of the above (c)
54. Following figure represents:  
  
 (a) Directed Graph (b) Undirected graph  
 (c) Unconnected graph (d) AVL tree (a)
55. A node that has no children is called:  
 (a) lead node  
 (b) root node  
 (c) parent node  
 (d) none of the above (d)
56. A connection between two vertices is called:  
 (a) edge  
 (b) vertex  
 (c) tree  
 (d) none of the above (a)

57. What will be the expression for the following tree?



- (a)  $(A + (B * C)) - (D * E)$   
 (b)  $(A+B) * C - DE$   
 (c)  $ABC + *(D * E)$   
 (d) None of the above (a)
58. Which of the following is an advantage of using pointers:  
 (a) Pointers reduce the length and complexity of a program  
 (b) They increase the execution speed  
 (c) Pointers are more efficient in handling the data tables  
 (d) All of the above (d)
59. Floating point data types are represented by the word:  
 (a) int (b) floating  
 (c) float (d) char (c)
60. If there are  $n$  vertices in the graph then how many edges are needed to construct a minimum spanning tree?  
 (a)  $n$  (b)  $n+1$   
 (c)  $n-1$  (d)  $n^2$  (c)
61. If we use a 16-bit word length, the size of the integer value is limited to the range:  
 (a)  $-2^6$  to  $2^6-1$   
 (b)  $-2^{10}$  to  $2^{10}-1$   
 (c)  $2^{15}$   
 (d)  $-2^{15}$  (a)
62. If we use a 16-bit word length, the size of the integer value is limited to the range:  
 (a)  $-2^6$  to  $2^6-1$   
 (b)  $-2^{10}$  to  $2^{10}-1$   
 (c)  $2^{15}$

- (d)  $-2^{15}$  (a)
63. Which of the following is a non linear data structure?  
 (a) Tree only (b) Graph only  
 (c) Array (d) Both tree and graph (d)
64. In the balanced binary tree given below, how many nodes become unbalanced when a node inserted as a child of the node "g"
- 
- (a) 1 (b) 3  
 (c) 7 (d) 8 (b)
65. Which of the following abstract data types can be used to represent a many to many relation:  
 (a) Tree  
 (b) Graph  
 (c) Both A and B  
 (d) None of the above (b)
66. When of the following in turn calls another function a process of 'chaining' occurs. A special case of this process, where a function calls itself is called:  
 (a) Recursion (b) Delection  
 (c) Insertion (d) Overloading (a)
67. Which of the following is data structure operation?  
 (a) Searching  
 (b) Sorting  
 (c) Traversing  
 (d) All of the above (d)
68. In which tree for every node the height of its left and right sub tree differ at last by one?  
 (a) Binary search tree  
 (b) AVL tree  
 (c) Complete Tree  
 (d) None of the above (d)
69. Which of the following is not a type of tree?  
 (a) Binary (b) Binary Search  
 (c) AVL (d) Insertion (b)

70. The process memory allocation at run time is known as:  
(a) Dynamic memory allocation (b) Static memory allocation  
(c) Compaction (d) fragmentation (a)
71. The collection of same type of data is called:  
(a) An array  
(b) A union  
(c) A structure  
(d) None of the above (a)
72. In a linear linked list, a node contains at least:  
(a) Node address field and next pointer field  
(b) Node number and data field  
(c) An information field and next pointer field  
(d) None of the above (c)
73. Which of the following sorting algorithm is based on the idea of "Divide" and conquer?  
(a) Merge sort  
(b) Heap sort  
(c) Both B and A  
(d) None of the above (d)
74. Which of the following is a method of searching?  
(a) Linear search (b) Bubble search  
(c) Insertion search (d) Selection search (a)
75. The element at the root of heap is:  
(a) Largest  
(b) Smallest  
(c) Depends on type of heap  
(d) None of the above (c)
76. Average case time complexity of quick sort algorithm is:  
(a)  $O(n \log n)$   
(b)  $O(\log_2 n)$   
(c)  $O(n^2)$   
(d) None of the above (a)

77. The five items A B C D AND E are pushed in a stack, one after the another starting from A. The stack is popped four times and each elements is inserted in a queue. The two elements are deleted from the queue and pushed back on the stack. Now one item is popped from the stack. The popped item is:
- (a) A (b) B  
(c) C (d) D (d)
78. Malloc function returns a NULL when:
- (a) Memory is successfully allocated  
(b) All memory is cleared  
(c) Space is insufficient to satisfy the request  
(d) None of the above (c)
79. FRONT and REAR words are related with:
- (a) Stack  
(b) Queue  
(c) Linked list  
(d) None of the above (b)
80. Queue is also known as:
- (a) LIFO – system  
(b) FIFO system  
(c) LIFO FIFO  
(d) None of the above (b)
81. For a sequential search, the average number of comparisons for a file with records is:
- (a)  $(n+1)/2$  (b)  $\log_2 n$   
(c)  $n^2$  (d)  $n/2$  (d)
82. A data structure, in which an element is added and removed only from one end is known as:
- (a) Queue  
(b) Stack  
(c) Array  
(d) None of the above (b)
83. Header of a linked list is a special node at the:
- (a) End of the linked list  
(b) Middle of the list  
(c) Beginning of the list

- (d) None of the above (c)
84. Adjacency matrix for a graph is:  
(a) Unimarix  
(b) Symmetric  
(c) Asymmetric  
(d) None of the above (b)
85. What is the minimum number of fields with each element of a doubly linked list?  
(a) 1  
(b) 2  
(c) 3  
(d) 4 (c)
86. Worst case complexity of quick sort algorithm is:  
(a)  $O(n^2)$   
(b)  $O(n \log n)$   
(c)  $O(\log n)$   
(d) None of those (a)
87. Average case complexity of heap sort algorithm is:  
(a)  $O(n \log n)$   
(b)  $O(n^2)$   
(c)  $O(n)$   
(d)  $O(n \log_2 n)$  (a)
88. Hashing or has addressing is a technique of:  
(a) Searching  
(b) Sorting  
(c) Both (a) and (b)  
(d) None of the above (a)
89. The notation in which operation symbol is placed before its two operands, is called:  
(a) Infix notation  
(b) Polish notation  
(c) Suffix notation  
(d) None of the above (b)
90. ....is the term used to delete an element from a stack.  
(a) PUSH (b) POP  
(c) DEL (d) Both B and C (b)



91. When a called function in turn calls another function a process of chaining occurs. A special case of this process, where a function calls itself is called.
- |               |                 |     |
|---------------|-----------------|-----|
| (a) Recursion | (b) Deletion    |     |
| (c) Insertion | (d) Overloading | (a) |
92. Sparse matrices have:
- |                            |     |
|----------------------------|-----|
| (a) Many zero elements     |     |
| (b) Many non zero elements |     |
| (c) Higher dimension       |     |
| (d) None of the above      | (a) |
93. Length of the string "Manisha" is :
- |                       |     |
|-----------------------|-----|
| (a) 7                 |     |
| (b) 8                 |     |
| (c) either 7 or 8     |     |
| (d) None of the above | (a) |
94. What will be the results of insert (' ABCDEFG',3 XYZ)?
- |                       |     |
|-----------------------|-----|
| (a) ABCDEFGXYZ        |     |
| (b) ABXYZCDEFG        |     |
| (c) ABCXYZDEFG        |     |
| (d) None of the above | ( ) |
95. The string with zero characters is called:
- |                  |                  |     |
|------------------|------------------|-----|
| (a) Empty string | (b) Null string  |     |
| (c) Full string  | (d) Both A and B | ( ) |
96. The variables which can be accessed only within a particular program or subprogram are known as:
- |                        |     |
|------------------------|-----|
| (a) Local variables    |     |
| (b) Global variable    |     |
| (c) Auto variables     |     |
| (d) External variables | ( ) |

## KEY TERMS

**abstract data type** A set of data values and associated operations that are precisely specified independent of any particular implementation.

**Algorithm-**

A computable set of steps to achieve a desired result.

**asymptotic space complexity**

When analyzing the running time or space usage of programs, we usually try to estimate the time or space as function of the input size.

**asymptotic bound**

A curve representing the limit of a function. That is, the distance between a function and the curve tends to zero. The function may or may not intersect the bounding curve.

**Access vector** — A list of pointers providing access to a set of data items.

**Algorithm** — A rule for arriving at an answer in a finite number of steps.

**Ancestor** — A parent of a parent (or an ancestor).

**Arc** — An edge on a directed graph.

**Array** — An elementary data structure that resembles a table; typically, one data element is stored in each array cell and the cells are distinguished by subscripts.

**Attribute** — A property of an entity.

**Binary tree** — A special type of tree in which each node has two branches.

**Branch** — On a tree, a link between a parent and a child.

**Child** — An immediate lower-level node in a tree.

**Circular linked list** — A linked list in which the last node points back to the first node.

**Cycle** — On a graph, a path that leads from a node back to the same node.

**Data element** — An attribute that cannot be logically decomposed; the most basic unit of data that has logical meaning.

**Data structure** — A way of organizing data that considers both the data items and their relationships to each other.

**Descendant** — A child of a child (or a descendant).

**Directed graph (digraph)** — A graph on which each edge (or arc) has a direction.

**Doubly linked list** — A linked list in which each node contains both forward and backward pointers.

**Edge** — On a graph, a link between two nodes.

**Entity** — An object (a person, group, place, thing, or activity) about which data are stored.

**Field** — A data element physically stored on some medium.

**File** — A set of related records.

**Graph** — A set of nodes (or vertexes) linked by a set of edges.

**Indegree** — On a directed graph, the number of arcs entering a given node.

**Key** — The attribute or group of attributes that uniquely distinguishes one occurrence of an entity.

**Leaf (leaf node)** — On a tree, a node with no branches.

**Linked list** — A list in which each node contains data plus a pointer to the next node.

**List** — A series of nodes each of which holds a single data item; the most basic data structure.

**Matrix** — A two-dimensional array.

**Minimum spanning tree** — Within a graph, a subtree or spanning tree for which the sum of arc weights is minimal.

**Multi-linked list** — A linked list in which each node contains two or more pointers, thus providing access to two or more other nodes.

**Multi-way tree** — A tree in which each node holds  $n$  (two or more) values and can have  $(n + 1)$  branches.

**Network (weighted graph)** — A graph on which the edges have values.

**Node** — An entry in a list; often, a single data element or a single record.

**Occurrence** — A single instance of an entity.

**Ordered list** — A list in which the nodes are stored in data value or key order.

**Outdegree** — On a directed graph, the number of arcs exiting from a given node.

**Parent** — The immediate higher-level node in a tree.

**Path** — On a graph, a sequence of edges that links a set of nodes; on a digraph, the path's direction is significant.

**Pointer** — A link to a data item; typically, a key value or an address.

**Pop** — To remove an entry from the top of a stack.

**Push** — To add an entry to the top of a stack.

**Queue** — A special type of linked list in which insertions occur at the rear and deletions occur at the front.

**Record** — The set of fields associated with an occurrence of an entity.

**Recursion** — A subroutine calling itself; a subroutine initiating a circular chain of calls that returns eventually to itself.

**Root (root node)** — A tree's top (or base) node.

**Siblings** — Two or more nodes that share the same level.

**Singly linked list** — A linked list in which each node points only to the next node.

**Sink** — On a directed graph, a node of outdegree 0.

**Source** — On a directed graph, a node of indegree 0.

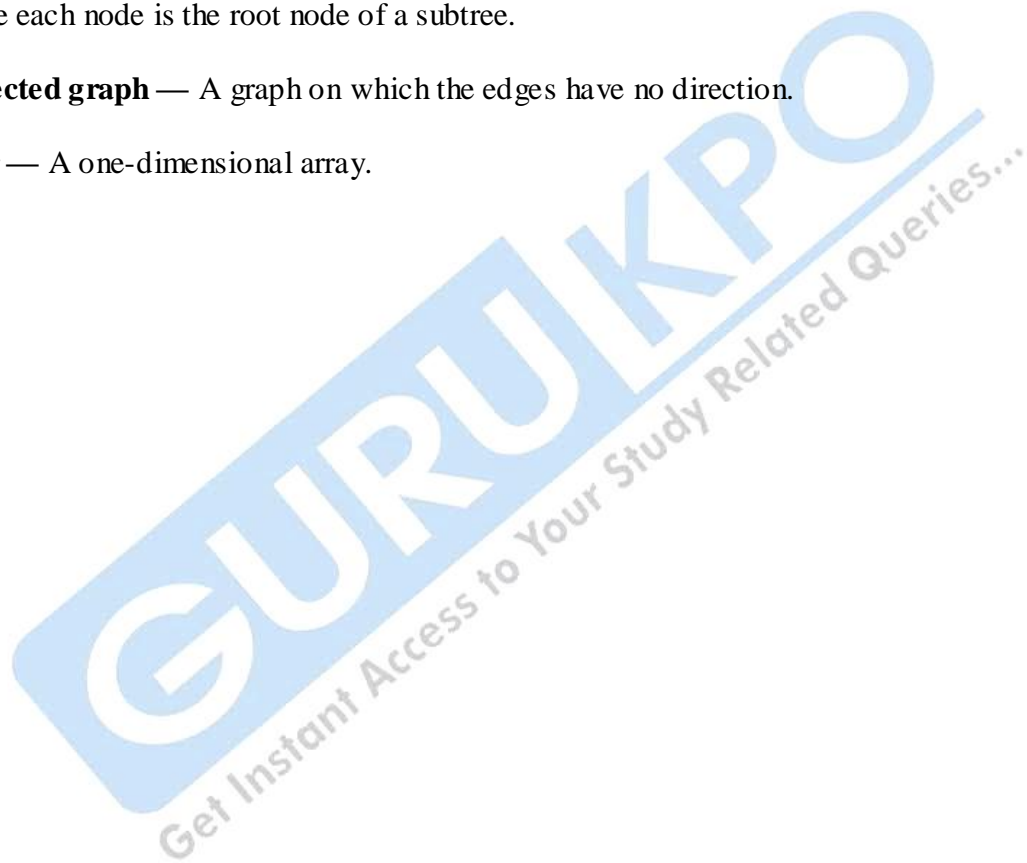
**Stack** — A special type of linked list in which all insertions and deletions occur at the top.

**Subtree (spanning tree)** — A tree within a graph; a subset of a tree that is itself a tree.

**Tree** — A two-dimensional, hierarchical data structure; a tree can be defined recursively because each node is the root node of a subtree.

**Undirected graph** — A graph on which the edges have no direction.

**Vector** — A one-dimensional array.



## References

Data Structures and Algorithms (Addison-Wesley Series in Computer Science and Information Pr)

2) data structure schaum series by lipschutz

3) Data Structure ( Tanenbaum book)

4) Data structure by Balaguruswamy.

Source Site could give you more information :

1) [http://en.wikipedia.org/wiki/Data\\_structure](http://en.wikipedia.org/wiki/Data_structure)

2) <http://www.cplusplus.com/doc/tutorial/structures.html>

