

**KENYA INSTITUTE OF CURRICULUM DEVELOPMENT
STUDY NOTES**

Structured Programming

Contents

CHAPTER 1: INTRODUCTION TO STRUCTURED PROGRAMMING	4
Basic Programming Concept	4
Structured programming and other programming languages	4
Development of Programming Language	5
Programming Paradigms.....	6
Computer hardware and software consideration	6
CHAPTER 2: PROGRAM DEVELOPMENT AND DESIGN	10
Introduction to program development.....	10
Program Development Cycle	10
Programming Techniques	11
Programming Tools	18
CHAPTER 3 PROGRAM STRUCTURE	36
Introduction to program structure	36
Format of a structured programming language	36
Basic C Programs - Variable and I/O instructions	38
Program Structures in Pascal	41
Data types, identifiers and operators	44
Variable Types.....	46
Operator Types	48
CHAPTER 4: PROGRAM WRITING.....	60
Content of structured programming	60
Steps to Develop a Program	60
Error Handling	61
CHAPTER 5: CONTROL STRUCTURES.....	65
Introduction to control structures	65
Sequence structure	65
Selection/Decision Structure	65
Loops/Iterations.....	74
Loop Control Statements	81
CHAPTER 6: DATA STRUCTURES.....	87
Introduction to Data Structures.....	87

Primitive and Non-Primitive data Types	87
Algorithm	88
Array.....	91
Stack.....	95
Queue.....	99
Linked Lists	103
Graph Data Structure	112
Tree	118
Searching and Sorting	137
CHAPTER 7: SUB PROGRAMS	150
Introduction to Subprograms	150
Procedures and Functions	151
Design Issues for Subprograms.....	151
Local Referencing Environment	151
Parameter-Passing Methods.....	152
Writing subprograms - Functions	154
CHAPTER 8: FILE HANDLING.....	158
Introduction to files in programming.....	158
Importance of file handling.....	158
File organization techniques	158
Files Handling (Input/Output) in C programming	159
CHAPTER 9: PROGRAM DOCUMENTATION	169
Define program documentation	169
Importance of programming documentation.....	169
Types of program documentation	169
Writing System/Program documenting	171
CHAPTER 10: EMERGING TRENDS OF STRUCTURED PROGRAMMING	172

CHAPTER 1: INTRODUCTION TO STRUCTURED PROGRAMMING

Basic Programming Concept

Program and Programming

A **computer program** is a series of organized instructions that directs a computer to perform tasks. Without programs, computers are useless.

A program is like a recipe. It contains a list of variables (called ingredients) and a list of statements (called directions) that tell the computer what to do with the variables.

Programming is a creation of a set of commands or instructions which directs a computer in carrying out a task.

A **programming language** is a set of words, symbols and codes that enables humans to communicate with computers.

Examples of programming languages are:

- BASIC (Beginner's All Purpose Symbolic Instruction Code)
- Pascal
- C
- Smalltalk.

Structured programming and other programming languages

Structured programming (sometimes known as *modular programming*) is a subset of procedural programming that enforces a logical structure on the program being written to make it more efficient and easier to understand and modify. Certain languages such as [Ada](#), [Pascal](#), and dBASE are designed with features that encourage or enforce a logical program structure.

Structured programming frequently employs a top-down design model, in which developers map out the overall program structure into separate subsections. Program flow follows a simple hierarchical model that employs looping constructs such as "for," "repeat," and "while." Use of the "Go To" statement is discouraged.

C is called a structured programming language because to solve a large problem, C programming language divides the problem into smaller modules called functions or procedures each of which handles a particular responsibility. The program which solves the entire problem is a collection of such functions. One major drawback of C language is that similar functions cannot be grouped inside a module or class. Also functions cannot be associated to a type or structure. Thus data and functions cannot be bound together. C++

language overcomes these problems by introducing object oriented functionality in its programming capabilities.

Development of Programming Language

FIRST GENERATION OF PROGRAMMING LANGUAGE

The first generation of programming language, or 1GL, is **machine language**. Machine language is a set of instructions and data that a computer's central processing unit can execute directly. Machine language statements are written in binary code, and each statement corresponds to one machine action.

SECOND GENERATION PROGRAMMING LANGUAGE

The second generation programming language, or 2GL, is **assembly language**. Assembly language is the human-readable notation for the machine language used to control specific computer operations. An assembly language programmer writes instructions using symbolic instruction codes that are meaningful abbreviations or mnemonics. An assembler is a program that translates assembly language into machine language.

THIRD GENERATION PROGRAMMING LANGUAGE

The third generation of programming language, 3GL, or **procedural language** uses a series of English-like words, that are closer to human language, to write instructions.

High-level programming languages make complex programming simpler and easier to read, write and maintain. Programs written in a high-level programming language must be translated into machine language by a compiler or interpreter.

PASCAL, FORTRAN, BASIC, COBOL, C and C++ are examples of third generation programming languages.

FOURTH GENERATION PROGRAMMING LANGUAGE

The fourth generation programming language or **non-procedural language**, often abbreviated as 4GL, enables users to access data in a database.

A **very high-level programming language** is often referred to as goal-oriented programming language because it is usually limited to a very specific application and it might use syntax that is never used in other programming languages.

SQL, NOMAD and FOCUS are examples of fourth generation programming languages.

FIFTH GENERATION PROGRAMMING LANGUAGE

The fifth generation programming language or visual programming language is also known as **natural language**. Provides a **visual or graphical interface**, called a visual programming environment, for creating source codes. Fifth generation programming allows people to interact with computers without needing any specialized knowledge. People can talk to computers and the voice recognition systems can convert spoken sounds into written words.

Prolog and Mercury are the best known fifth-generation languages.

EXT : OPEN PROGRAMMING LANGUAGE

The Open Programming Language (OPL) is an embedded programming language found in portable devices that run the Symbian Operating System. For example mobile telephones and PDAs. OPL is an interpreted language that is analogous to BASIC.

Programming Paradigms

A **programming paradigm** is a style or “way” of **programming**. Some languages make it easy to write in some **paradigms** but not others. Never use the phrase “**programming language paradigm**.” A **paradigm** is a way of doing something (like **programming**), and not a concrete thing (like a language).

Some of the more common paradigms are

- **Imperative** — Control flow is an explicit sequence of commands.
- **Declarative** — Programs state the result you want, not how to get it.
- **Structured** — Programs have clean, goto-free, nested control structures.
- **Procedural** — Imperative programming with procedure calls.
- **Functional (Applicative)** — Computation proceeds by (nested) function calls that avoid any global state.
- **Function-Level (Combinator)** — Programs have no variables. No kidding.
- **Object-Oriented** — Computation is effected by sending messages to objects; objects have state and behavior.
 - **Class-based** — Objects get their state and behavior based on membership in a class.
 - **Prototype-based** — Objects get their behavior from a prototype object.
- **Event-Driven** — Control flow is determined by asynchronous actions (from humans or sensors).
- **Flow-Driven** — Computation is specified by multiple processes communicating over predefined channels.
- **Logic (Rule-based)** — Programmer specifies a set of facts and rules, and an engine infers the answers to questions.
- **Constraint** — Programmer specifies a set of constraints, and an engine infers the answers to questions.
- **Aspect-Oriented** — Programs have cross-cutting concerns applied transparently.
- **Reflective** — Programs manipulate their own structures.
- **Array** — Operators are extended to arrays, so loops are normally unnecessary.

Paradigms are **not meant to be mutually exclusive**; you can program in a functional, object-oriented, event-driven style.

Computer hardware and software consideration

A computer system consists of two major elements: *hardware and software*.

Hardware – the physical machines that make up a computer installation (ex. printer, monitor, motherboard)

Software – the collection of programs used by a computer (ex. Word, Excel, Turbo Pascal)

Computer software, or just **software**, is any set of machine-readable instructions (most often in the form of a computer program) that directs a computer's processor to perform specific operations.

On most computer platforms, software can be grouped into a few broad categories:

- System software is the basic software needed for a computer to operate (most notably the operating system);
- Application software is all the software that uses the computer system to perform useful work beyond the operation of the computer itself;
- Embedded software resides as firmware within embedded systems, devices dedicated to a single use. In that context there is no clear distinction between the system and the application software.

Factors of consideration during Choosing hardware and software

Software factors

Factors influencing choice of software includes:

- (i) User requirements: the selected software or package should fit user requirement as closely as possible
- (ii) Processing time: these involves the response time e.g. if the response time is slow the user might consider the software or package as unsuccessful
- (iii) Documentation: the software should be accompanied by manual, which is easy to understand by non-technical person. The manual should not contain technical jargon.
- (iv) User friendliness: the package should be easier to use with clear on screen prompts, menu driven and extensive on screen help facility
- (v) Controls: the software should have in-built controls which may include password options, validation checks, audit trails or trace facilities etc
- (vi) Up-to-date: the software should be up-to-date e.g. should have changes or corrections in line with business procedures
- (vii) Modification: one should consider whether the user could freely change the software without violating copyright.
- (viii) Success in the market: one should consider how many users are using the software and how long it has been in the market
- (ix) Compatibility of the software: how the software integrates with other software particularly the operating system and the user programs
- (x) Portability: one should consider how the software runs on the user computer and whether there will be need for the user to upgrade his hardware
- (xi) Cost: the user company should consider its financial position to establish whether it can afford the software required for efficient operations rather than the least cost package software available.

Software contracts

Software contracts include the costs, purpose and capacity of the software. The following are covered in software contracts:

- Warrant terms
- Support available
- Arrangement for upgrades
- Maintenance arrangements
- Delivery period/time especially for written software
- Performance criteria
- Ownership

Software licensing

Software licensing covers the following:

- Number of users that can install and use the software legally
- Whether the software can be copied without infringing copyrights
- Whether it can be altered without the developers consent
- Circumstances under which the licensing can be terminated
- Limitation of liability e.g. if the user commits fraud using the software
- Obligation to correct errors or bugs if they exist in the software

Hardware factors

Custom-built hardware is a rare necessity. Most hardware is standard, compatible, off-the-shelf components. It is cheaper, easy to maintain, and ensures compatibility with equipment in your organization and your partners and clients.

The system analysis and design should have precisely determined what sort of hardware is needed - down to the make and model.

The decision of hardware choice must consider many factors:

- Future needs - can the equipment be expanded or added to?
- Availability (is it only available overseas?)
- Capacity (e.g. is the hard disk big enough to hold all your data? Is it fast enough?)
- Reliability - can it be depended on?
- Cost - initial cost, running costs, upgrade costs, repair costs, training costs
- Compatibility - with your other equipment, and that of your partners and clients
- Warranty and support - in case of failure or problems
- Ease of use and installation
- Compliance with local conditions (e.g. power supplies must be 240V or compliant with telecommunication systems)

Choosing a supplier

After choosing the hardware equipment and the equipment makers (manufacturers), one must choose a *supplier* or reseller (in other words, once you know what you want to buy, what shop will you choose?)

Factors to consider:

- Reputation for support (e.g. phone support, onsite visits, website help)
- Reputation for reliability, honesty, permanence (very important!)
- Knowledge of the equipment
- Geographic location - can you get to them easily if you need to?
- Ability to offer onsite support or repair
- Prices – cheap, affordable

Installation

Software and hardware installation is done by supplier's technicians or the user organization appointed person to avoid the risks associated with improper installation of the equipment. The system analyst and other development team members may be called to assist where appropriate.

User training

It is important that the system users be trained to familiarize themselves with the hardware and the system before the actual changeover.

The aims of user training are:

- a) To reduce errors arising from learning through trial and error
- b) To make the system to be more acceptable to the users
- c) To improve security by reducing accidental destruction of data
- d) To improve quality of operation and services to the users
- e) To reduce the cost of maintenance by minimizing accidental destruction of data or hardware
- f) To ensure efficiency in system operation when it goes live

The persons to be trained include system operators, senior managers, middle managers and all those affected by the system directly or indirectly. Training should cover current staff and recruited personnel.

CHAPTER 2: PROGRAM DEVELOPMENT AND DESIGN

Introduction to program development

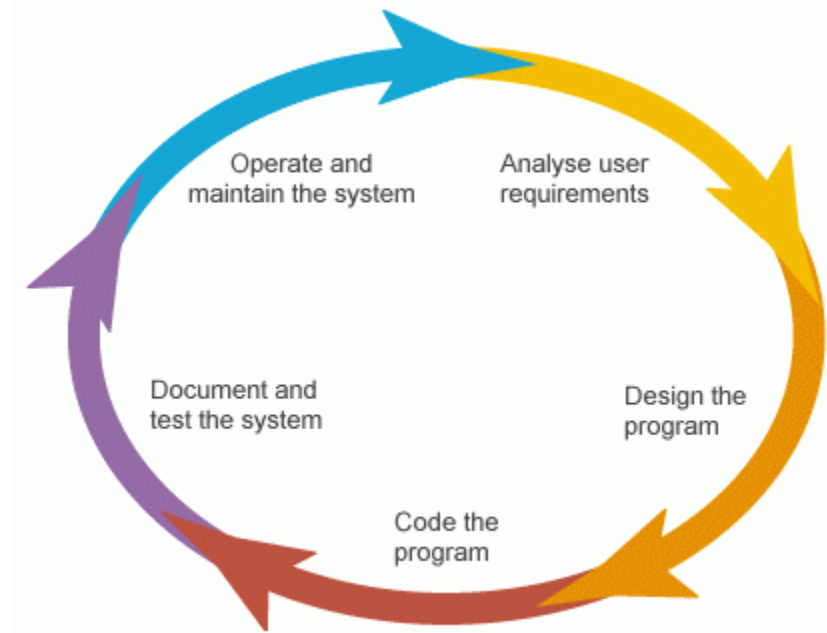
Program development in computing; refer to the coding of an individual software program or to the creation on an entire information system and all related software.

Program design is the activity of progressing from a specification of some required program to a description of the program itself.

It is the process that organizations use to develop a program. Ideally, the process is collaborative, iterative, and tentative—stakeholders work together to repeat, review, and refine a program until they believe it will consistently achieve its purpose.

A program design is also the plan of action that results from that process. Ideally, the plan is developed to the point that others can implement the program in the same way and consistently achieve its purpose.

Program Development Cycle



1. **Analyze – Define the problem.**

You must have a clear idea of what data (or input) is given and the relationship between the input and the desired output.

2. **Design – Plan the solution to the problem.**

Find a logical sequence of precise steps that solve the problem (aka the algorithm). The logical plan may include flowcharts, pseudocode, and top-down charts.

3. **Design the interface – Select objects (text boxes, buttons, etc.).**
Determine how to obtain input and how the output will be displayed. Objects are created to receive input and display output. Appropriate menus, buttons, etc. are created to allow user to control the program.
4. **Code – Translate algorithm into a programming language.**
During this stage that program is written.
5. **Test and debug – Locate and remove errors in program.**
Testing is the process for finding errors. Debugging is the process for correcting errors.
6. **Complete the documentation – Organize all materials that describe the program.**
Documentation is necessary to allow another programmer or non-programmer to understand the program. Internal documentation, known as comments, is created to assist a programmer. An instruction manual is created for the non-programmer. Documentation should be done during the coding stage.

Programming Techniques

Software designing is very anesthetic phase of *software development cycle*. The beauty of heart, skill of mind and practical thinking is mixed with system objective to implement design.

The *designing process* is not simple, but complex, cumbersome and frustrating with many curves in the way of successful design.

Here are some approaches:

- Structural Programming
- Modular Designing
- Top Down Designing
- Bottom Up Designing
- Object Oriented Programming

The objective of Program design are:

(i) **Replace old system:** The new system is used to replace old system because maintenance cost is high in old system and efficiency level low.

(ii) **Demand of Organization:** The new system is developed and installed on the demand of organization and working groups.

(iii) **Productivity:** The new system is installed to increase productivity of company or organization.

(iv) **Competition:** The new system is a matter of status also. In the age of roaring competition, if organization does not cope with modern technology failed to face competitions.

(v) **Maintenance:** The new system is needed to maintain organization status.

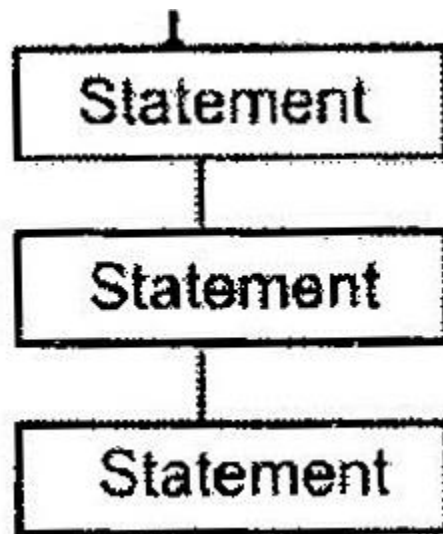
1. Structured Programming

This is the first programming approach used widely in beginning. Professor Edsger Wybe Dijkstra (1960) coins the term Structural Programming. Italian computer scientist C. Bohm and G. Jacopini (1966) give the basic principal that supports this approach. The structured programming movement started in 1970, and much has been written about it. It is often regarded as “*goto-less*” programming, because it is avoided by programmers.

The program is divided into several basic structures. These structures are called building blocks.

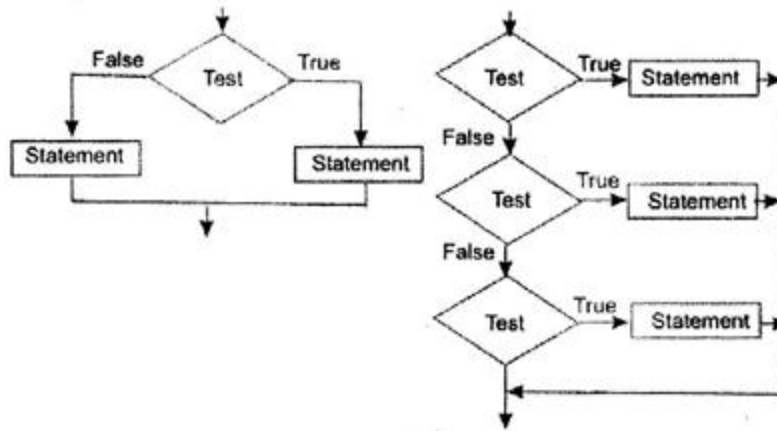
These are following:

(a) **Sequence Structure:** This module contains program statements one after another. This is a very simple module of Structured Programming.



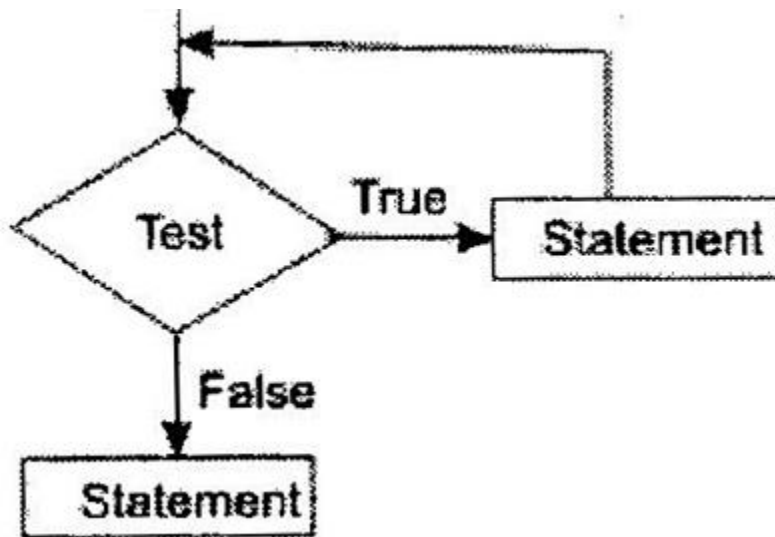
Sequence Structure

(b) **Selection or Conditional Structure:** The Program has many conditions from which correct condition is selected to solve problems. These are (a) if-else (b) else-if, and (c) switch-case



Conditional structure

(c) **Repetition or loop Structure:** The process of repetition or iteration repeats statements blocks several times when condition is matched, if condition is not matched, looping process is terminated. In C, (a) goto, (b) for (), (c) do, (d) do – while are used for this purpose.



Loop Structure

Advantage:

- Problem can be easily described using Flowchart and flowchart can be easily coded into program because the nature of this **technique** is like as flowchart.
- The program is easily coded using modules.
- The testing and debugging is easy because testing and debugging can be performed module-wise.
- Program development cost low.
- Higher productivity, high quality program production.
- Easy to modify and maintain

- It is called “**gotoless**” *programming* because use of *goto* for unconditional branching is strongly avoided. The *goto* is a sign of poor program design, so many designing concepts are not favoring it but it is used in all **programming language** widely. When more *goto* is used in program, program become less readable and its impact is negative in program functionality. So it is saying about *goto*: “Never, ever, ever use *goto*! It is evil”.

Disadvantage:

More memory space is required. When the numbers of modules are out of certain range, performance of program is not satisfactory.

2. Modular Programming

When we study educational philosophy, the concept of modulation can be clear without any ambiguity. Rene Descartes (1596-1650) of France has given concept to reconstruct our knowledge by piece by piece. The piece is nothing, but it is a module of modern *programming* context.

In modular approach, large program is divided into many small discrete components called Modules. In **programming language**, different names are used for it.

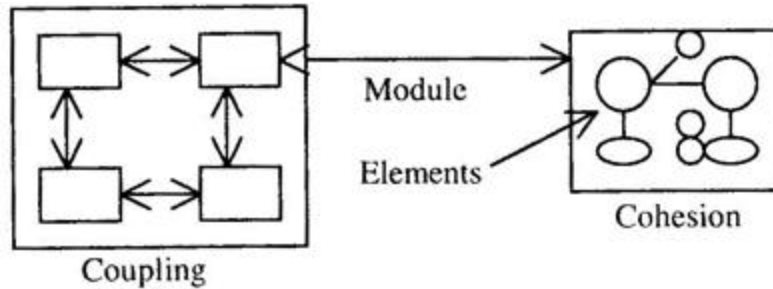
For example:

Q-basic, Fortran	Subroutine
Pascal	Procedure or Function
C, C+, C#, Java	Function

It is logically separable part of program. Modules are independent and easily manageable. Generally modules of 20 to 50 lines considered as good modules when lines are increased, the controlling of module become complex.

Modules are debugged and tested separately and combined to build system. The top module is called root or boss modules which charges control over all sub-modules from top to bottom. The control flows from top to bottom, but not from bottom to top.

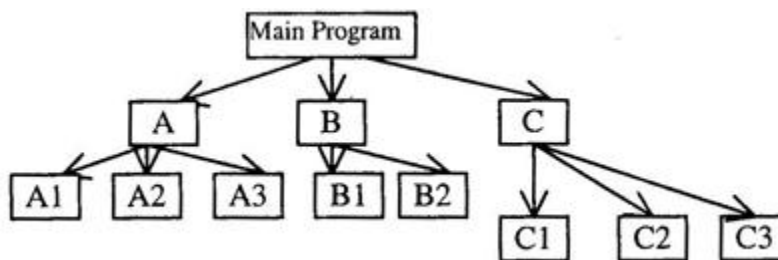
The evaluation of modeling is called coupling and cohesion. The module coupling denotes number of interconnections between modules and module cohesion shows relationship among data or elements within a module.



Modular Programming

3. Top down Approach

- The large program is divided into many small **module** or subprogram or function or procedure from top to bottom.
- At first supervisor program is identified to control other sub modules. Main modules are divided into sub modules, sub-modules into sub- sub- modules. The decomposition of modules is continuing whenever desired module level is not obtained.
- Top module is tested first, and then sub-modules are combined one by one and tested.



Top down Approach

Example: The main program is divided into sub-program A, B, and C. The A is divided into subprogram A1, A2 and A3. The B is into B1, and B2. Just like these subprograms, C is also divided into three subprogram C1, C2 and C3. The solution of Main *program* is obtained from sub program A, B and C.

4. Bottom up Approach

- In this approach designing is started from bottom and advanced stepwise to top. So, this approach is called **Bottom up** approach.
- At first bottom layer modules are designed and tested, second layer modules are designed and combined with bottom layer and combined modules are tested. In this way, designing and testing progressed from bottom to top.

- In software designing, only pure top down or Bottom up approach is not used. The hybrid type of approach is recommended by many designers in which top down and bottom up, both approaches are utilized.

5. Object oriented programming

In the *object-oriented programming*, program is divided into a set of objects. The emphasis given on objects, not on procedures. All the programming activities revolve around objects. An object is a real world entity. It may be airplane, ship, car, house, horse, customer, bank Account, loan, petrol, fee, courses, and Registration number etc. Objects are tied with functions. Objects are not free for walk without leg of functions. One object talks with other through earphone of functions. Object is a boss but captive of functions.

Features of Object oriented Language

- The *program* is decomposed into several objects. In this language, emphasis is given to the objects and objects are central points of programming. All the activities are object centered.
- Objects occupy spaces in memory and have memory address like as records in PASCAL and structure in C language.
- Data and its functions are encapsulated into a single entity.
- **Reusability:** In C++, we create classes and these classes have power of reusability. Other programmers can use these classes.
- It supports bottom up approach of programming. In this approach designing is started from bottom and advanced stepwise to top.

Some technical terms supporting object-oriented languages are:

(i) **Abstraction:** The abstraction is an important property of OOP. The use of essential features over less essential features is called **abstraction**. The following examples will help to understand abstraction.

Example: The computer operators know only to operate computer, but they are unaware to internal organization of computer.

In OOP, there are many devices used for data abstraction such as class, encapsulation, data hiding etc.

(ii) **Class:** A class is a collection of similar objects. Objects are members of class. Once a class is declared, its many members can be easily created in programs. The class binds attributes (data and functions or methods) of member objects.

Examples:

```
Class employee  
{  
  
    char name[30];  
  
    float basic;  
  
    void getdata();  
  
    void show();  
  
};
```

(iii) **Polymorphism:** The ability to find in many forms is called **polymorphism** (Poly: many, **Morphe:** shape / form). For instance, + is mathematical operator, it concatenates two strings and give sum of two digits (numbers). Here, operator + has different behavior for numerical data and strings. Just like it, once declared function has different meaning that is called **function overloading**. If operator has different meaning, it is called *operator overloading*.

(iv) **Encapsulation:** The *encapsulation* is a very striking feature of OOP in which data and function is bound into single unit. Array, records, structure are also example of low level encapsulation but term encapsulation is mostly used in object oriented language. The data and function are *encapsulated* into class. External world or external function cannot access the data. It can be accessed by its own function or method encapsulated with it into class. It hides private elements of objects behind public interface.

(v) **Inheritance:** *Inheritance* is a hierarchy of class in which some properties of base class is transferred to derived class. The main types of inheritance are:

(a) *Single Inheritance:* A derived class (child class or sub class) of single base (super or parent) class is called single Inheritance.

(b) *Multiple Inheritances:* A derived class of multiple base classes is called Multiple Inheritance.

(c) *Multilevel Inheritance:* When derived class is derived from another derived class, such type of inheritance is called Multilevel Inheritance.

Monolithic Design (architectural style or a software development Design pattern), describes a single-tiered software application in which the user interface and data access code are combined into a single program from a single platform.

A monolithic application is self-contained, and independent from other computing applications. The design philosophy is that the application is responsible not just for a particular task, but can perform every step needed to complete a particular function.

Programming Tools

The **programming** is a solution of different problems of our real life. If efficient programming tools are used, problems are effectively solved. We code some instructions to instruct **computer** for problem solving purposes. The choice of tools depends on nature of problems.

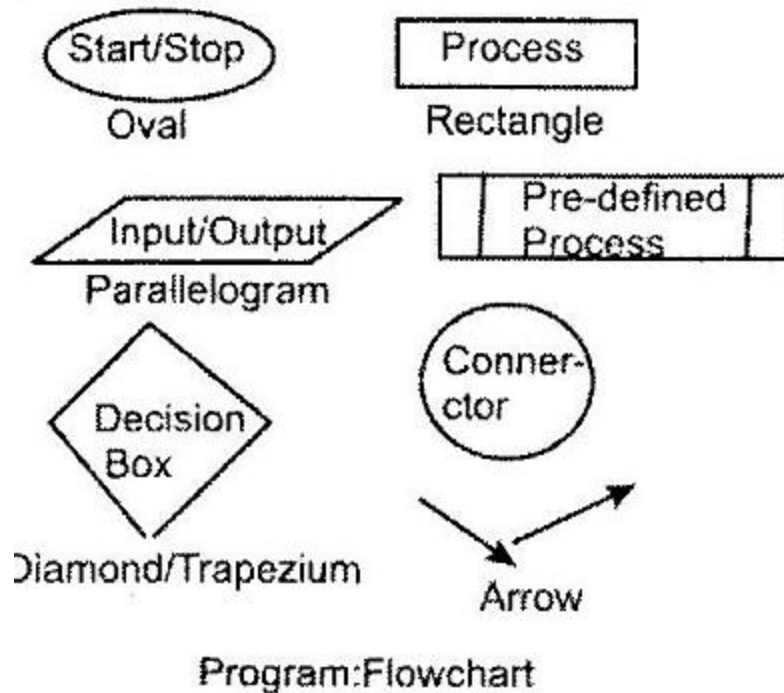
There are many tools for programmers for programming. For instance, **algorithms, flowcharts, pseudocodes, data dictionary, decision table, data flow diagrams** etc are effective tools. Enough and adequate knowledge of programming tools are essential for programming (Software development).

1. Flowchart

The pictorial presentation of program is called *Flowchart*. It is a tool and technique to find out solution of programming problems through some special symbols. It is a way to represent program using geometrical patterns. Prior to 1964, every manufactures use different types of symbols, there was no uniformity and standards of flowcharting. The Standard symbols were developed by American Standard National Institute (ANSI).

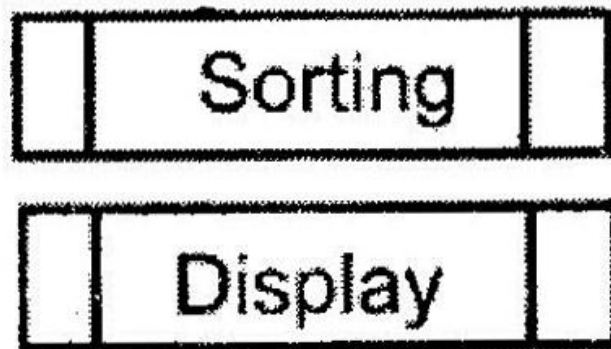
There are two types of Flow Chart:

- (a) Program Flow Chart and
- (b) System Flow Chart.



Program Flow Chart

- (i) **Start / Stop**: This oval is used to represent **START** and **STOP** of program.
- (ii) **Input/Output**: Parallelogram is used to denote input and output of data. We take data through it and display result also using this symbol.
- (iii) **Process**: Rectangle is used to denote process, formula, increment, and decrement and assigned value also.
- (iv) **Pre-defined Process**: The predefined process is denoted by this symbol. *Example*: Sorting and Display is pre-defined process and represented as:



Pre-defined Process

(iv) **Decision box:** It is a symbol of decision. All type of decisions is written in it.

(v) **Connector:** It is used to link to segment of flowchart to complete as one.

(vi) **Data flow:** It is used to show data flow from one component to other component, one process to other and one symbol to other symbol.

The **John Von Neumann** used flowchart to solve problem in 1945 and explained its importance in program designing.

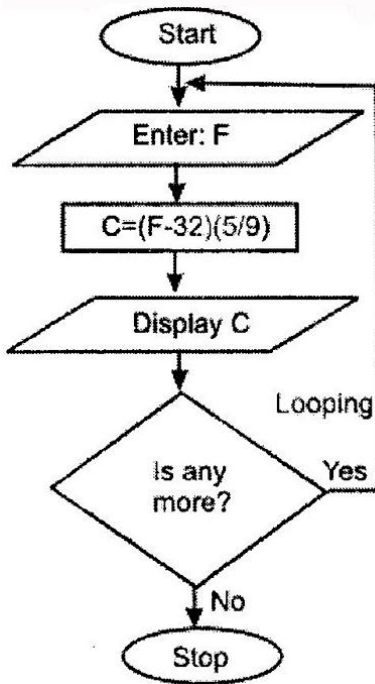
Illustration: In our everyday life, there is a sequence of works. These sequences of works are called routine. We all are tied with it. Suppose, we arise early in the morning at 5 0' clock, go to toilet, after coming from toilet, clean hands with soap and water, brush our teeth then take bath. Here, one sequence of work is formed.

In computer world, programming language is used to solve certain problems. The solution of problem is a sequence of processes. In one problem, there may be many processes, the flowchart help to arrange processes in definite order. Suppose, we have to convert Fahrenheit temperature into Celsius, it has following five steps:

- **Step-1:** Start
- **Step-2:** Enter Fahrenheit temperature (F)
- **Step-3:** Application of formula: $C = (F - 32) \times (5/9)$
- **Step-4:** Displaying the result (C)
- **Step-5:** Stop

These types of processes involved in solution of problem are called *Algorithm*. It is a stepwise presentation of problem solution in simple English without using confusing words and phrases.

When we draw flowchart, it is only conversion of steps or algorithm in special symbols.



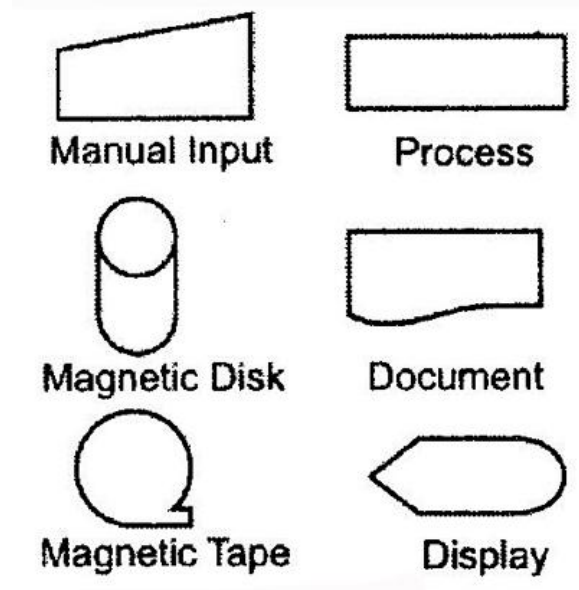
Flow Chart

Flowchart is like as city map. Travelers or tourists search places of historical importance through city map. Buildings are built on the basis of building-design (map) drawn by Architect engineers.

Just like it, program is coded on the basis of flowchart. Flowchart is a language free concept, if it is prepared, any language can be used for program coding. In software development cycle, flowcharting is one essential phase.

System Flowchart:

System flowchart is a pictorial representation of procedure flows inside and outside of the systems. It is also called **Data Flow Chart** or **Procedure Chart**.



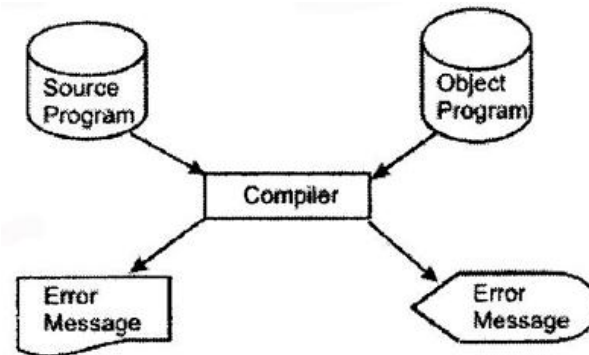
System Flowchart Symbols

- (i) **Manual Input:** It is used to enter data manually. For example, keyboard is a manual input device.
- (ii) **Process:** All type of processes is denoted by Rectangle.
- (iii) **Magnetic Disk:** Mass storage device (Hard Disk).
- (iv) **Magnetic Tape:** Magnetic tape is also storage device.
- (v) **Display:** This is a symbol for online display, Example: Monitor (VDU).
- (vi) **Document:** The print out document is denoted by this symbol.

Program Flowchart and System Flowchart:

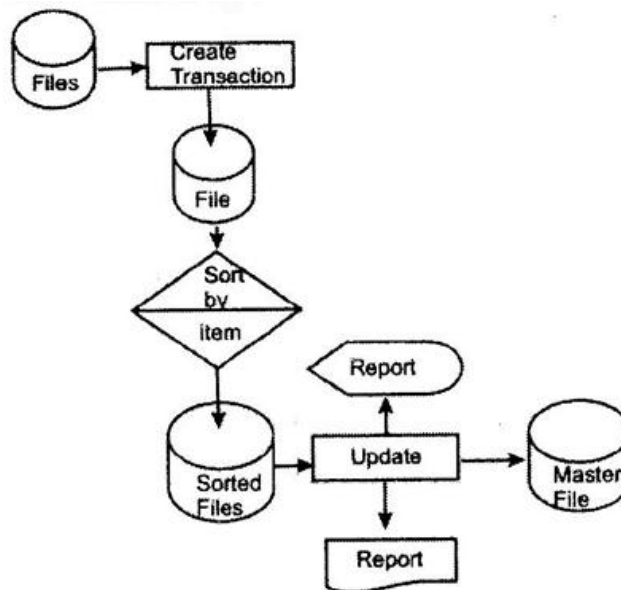
Sn.	Program Flow Chart	System Flow Chart
1.	The pictorial presentation of program is called Flowchart. It is a tool and technique to find out solution of programming problems through some special symbols. It is a way to represent program using geometrical patterns.	System flowchart is a pictorial representation of procedure flows inside and outside of the systems. It is also called Data Flow Chart or procedure Chart. The special type of symbols is used to represent system flowchart.
2.	It is tool of programmers. The flowchart designing to solve any problem is called program flowchart. The candidate system is designed using program flowchart.	This is a flowchart to show data flow inside or out to designed system. The system designers draw it to show data flow inside the current system.

Problem-1: The source program is stored in hard disk and after compilation; the object program (Machine code) is also stored in hard disk. The rectangle represents compilation process. The error messages are printed out and displayed on VDU also.



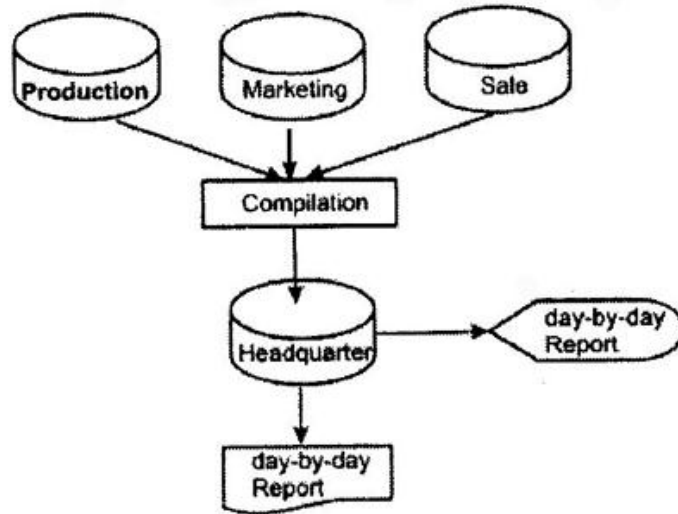
Compilation Process

Problem-2: Draw a system flow chart to update master-file.



System Flowchart to update master file

Problem-3: The headquarter of ABC Private Ltd. is linked with production, marketing and sale department. The day-by-day information of each department is compiled at headquarter, stored in master file and report is given to higher authority of company.



2. Algorithm

Algorithm is a stepwise presentation of procedures of program in simple and phraseless English. It is also a **programming tools** used by programmer to develop software. It is a logical representation of program procedures in order to obtain result. So, sometimes, it is called **Procedure** also. An **algorithm**, named after the ninth Centaury scholar Abu Jafar Muhammad Ibn Musa Al-Khowarizmi of Bagdad (Iraq).

A program is an expression of an idea in any programming language. A programmer starts with a general idea of a task for the computer to perform. The problem is flesh out the outline into a complete, unambiguous, step-by-step procedure for carrying out the task. Such a procedure is called an "**Algorithm**". An *algorithm* is not same as program. An algorithm is more like the idea behind the program.

- Language independent
- Simple, complete, unambiguous, step-by-step program flow
- no standard format or syntax for the algorithm
- helpful to understand problems and plan out solution

Example: Following algorithm is written to find out roots of quadratic equations:

- **Step-1:** Start
- **Step-2:** Read a,b and c
- **Step-3:** $d = b^2 - 4 * a * c$
- **Step-4:** if $D < 0$, root is imaginary
- **Step-5:** $x1 = (-b + \text{sqrt}(D)) / (2 * a)$

$$X2 = (-b - \text{sqrt}(D)) / (2 * a)$$

- **Step-6:** Display x1 and x2

- **Step-7:** Stop

Example: Write an algorithm to find out area of triangle when sides are given.

Algorithm:

Step-1: start

Step-2: read a,b and c

Step-3: $S=(a+b+c)/2$

Step-4: $A=\sqrt{(s-a)(s-b)(s-c)}$

Step-5: Display A

Step-6: Stop

Example: Write an algorithm to find out slop and mid point of coordinate.

Algorithm:

Step-1: Start

Step-2: Read x1, x2, y1 and y2

Step-3: $Slop=(y2-y1)/(x2-x1)$

Step-4: $x=(x1+x2)/2$

Step-5: $y=(y1 + y2)/2$

Step-6: Stop

There are no standard rule for algorithm designing, but we can use some common rules which are not ambiguous to another computer literates:

(a) **Value to variable:** variable (Variable or Expression)

(b) **Conditional statements:** if -endif and if –else-end-if are used for conditions.

Example:

If age \geq 18 then print “Case vote”

Else print “Wait until not 18”

Endif

(c) looping statements:

i. do while (condition)

Statement(s)

endo

ii. for variable=value1 to final value step stepvalue

statement(s)

endfor

Example: Write an algorithm to display factorial of given number.

Fact = 1

Print “Enter number”

Read number

For count =1 to count number step 1

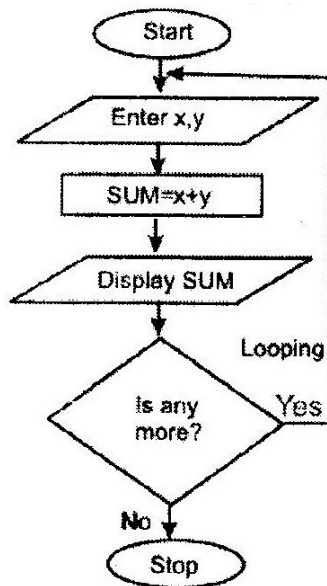
Fact = fact * count

Endfor

Print “Factorial:”, fact

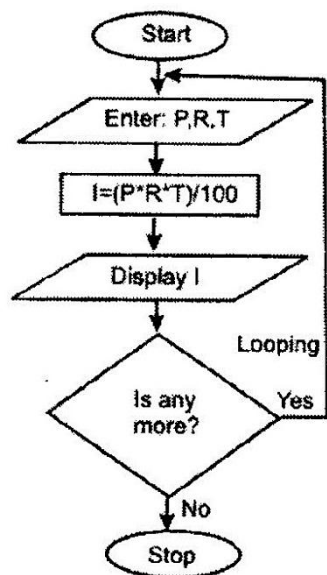
End

Problem-1: Draw a flow chart to input two numbers and display sum.



Flow Chart to display sum of two numbers

Problem-2: Draw a flowchart to enter principal,rate and time and display simple interest.

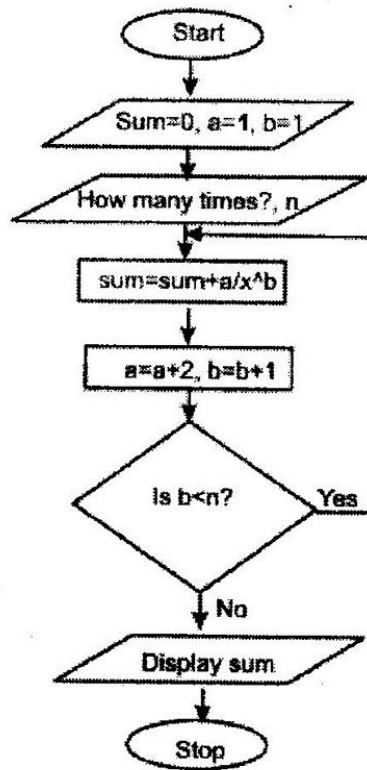


Flowchart to Calculate Simple Interest

Problem-3: Draw a flowchart to find the sum of the given series.

$$\frac{1}{x} + \frac{3}{x^2} + \frac{5}{x^3} + \dots$$

Solution,



Flow Chart to find sum of series

3. Pseudo-code or Structured English

A logical construction, which has

- (a) No strict rules like as programming language,
- (b) Unambiguous statements,
- (c) Phrase less statements, contains noun and simple verb of English , and
- (d) No adjective and adverbs structure is called **Pseudo-code** or *Structured English*.

It is a **Program Design Language (PDL)** provides skeleton for program design and it can replace flowchart used to describe system design.

General rules used in writing *Pseudo-codes* are as follow:

(i) **Imperative Sentence:** The imperative sentences are used to show actions.

Example: Add x to sum, Display result, Sort list etc.

(ii) **Operators:** Mostly arithmetic operators (+, -, *, / etc) and relational operators (=, >, <, > etc.) are used.

(iii) **Decision:** IF, THEN, ELSE, ENDIF, CASE, ENDCASE are used for decision-making.

Example:

CASE (choice)

(Choice = 1):Display Employee, Department

(Choice = 2): Display Employee, BasicSal, Allowance, Gross

(Choice = 3): Terminate Program

ENDCASE.

(iv) **Looping:** The looping or repetition of statements is shown by FOR, FOR DO, ENDFOR, DO, DO WHILE, DO UNTIL, ENDDO etc.

Example:

Display “Selected for National scholarship Examination”

FOR roll1 TO roll=10 DO

Display roll, Name, Address

ENDFOR.

Example:

IF age >=18

THEN CASTE vote

ELSE

WAIT until age is not 18

ENDIF

Example: Mailing a letter

BEGIN

WRITE a letter

IF bicycle is available

THEN GO to Post Office by Bicycle

ELSE

GO to Post Office on foot

ENDIF

BUY a stamp

STICK a stamp on the letter

PUT into letterbox

END.

4. Decision Table

A decision table defines a logical procedure by means of a set of conditions and related actions. It is used for communicating and documenting complex decision procedures.

Procedure: The decision table is divided into four quadrants: Condition stub, Condition entry, Action stub, and action entry. The conditions are answered as Y (yes) or N (No). The blank space shows that the condition involved has not been tested. X (or check marks) represents response to the answer.

Condition Stub	C1	R1 R2 R3 R4
	C2	Condition entry
	-	
	Cn	
Action Stub	A1	Action entry
	A2	
	-	
	An	

Problem: The policy followed by an ABC Ltd. to process agents' orders is given by the following rules:

- (i) If the customer order \leq that in stock and his credit is OK, supply his requirement.
- (ii) If the customer credit is not OK, do not supply. Send him intimation.
- (iii) If the customer credit is OK but items in stock are less than his order, supply what is in stock. Enter the balance to be sent in a back-order file.

Decision Table:

Order \leq Stock	Y	Y	N	N
Credit OK	y	N	Y	N
A1: Supply order	X	-	-	-
A2: Credit not OK	-	X	-	X
	-			
Pay cash	-			

A3: Supply Stock	-	-	-	-
A4: Enter (order-Stock) in back order file		-	-	-

Problem: Create a decision table to decide whether on a given data a student in an University has completed 3 years of enrolment or not.

Decision Table:

Let data enrolment $D_j - M_j - Y_j$ (day-month-year) and current date is $D_t - M_t - y_t$ (say – Month –Year)

If ($Y_j - Y_t$)	>3	$=3$	$=3$	Else
If ($M_j - M_t$)	-	>0	$=0$	
If ($D_j - D_t$)	-	-	≥ 0	
3 years enrolment?	Y	Y	Y	N

Problem: The Delta gas company bills its customers according to the following rate schedule:

- (i) First 500liters Rs. 50 (*flat*)
- (ii) Next 300 Liters Rs. 5.25 per 100 liter
- (iii) Next 30,000 liters Rs. 2.50 per 100 liters
- (iv) Above this Rs.2.00 per 100 liters.

The input record has customer identification, name and address, new meter reading, past and present. Create decision table to obtain bill for customers.

Decision Table:

Consumption= old meter reading-new meter reading

Charge table:

Consumption	<=500	501 to 800	801 to 30800	>30801
Go To	C1	C2	C3	C4

C1: Charge=50

C2: Charge=50+ (consumption-500)*0.0525

C3: Charge=65.75+ (consumption-800)*0.0250

C4: Charge=815.75+ (consumption-30800)*0.02

Problem. A bank has the following policy on deposits: on deposits of Rs. 25000 and above and for three years or above the interest is 10%. On the same deposit for a period less than three years it is 8%. On deposits below Rs. 25000 the interest is 6% regardless of the period of deposit. Write the above process using

(a) Structural English

(b) A decision Table.

Solution

(a) for each deposit do

If deposit \geq 25000

Then if period \geq 3 years

Then interest=10%

Else interest=8%

Endif

Else interest 6%

Endfor

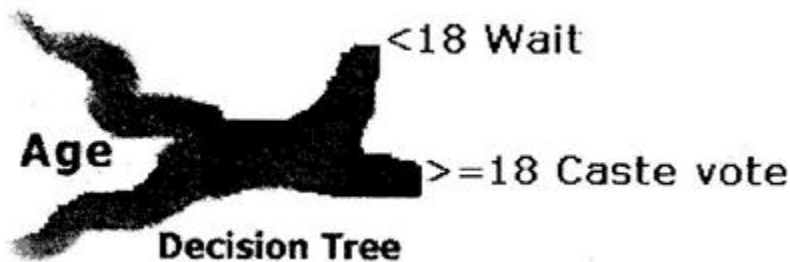
(b) Decision table:

Deposit \geq 25000	Y	Y	N
	Y	N	-
Interest %	10	8	6

5. Decision Tree

The tree like presentation of condition and actions are called **decision tree**. Every node denotes conditions. It is used for logical variations and problems involving few complex decisions with complex branching routines.

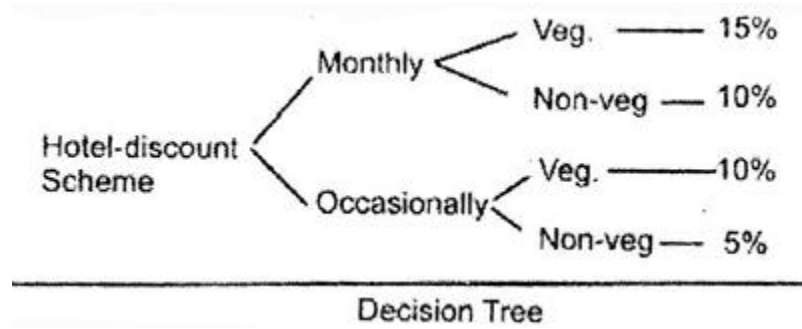
Example: The person which age is greater or equal to 18 can caste vote in the favour of his leader otherwise has to wait until age of 18. There is two conditions (a)age<18, and (b)age \geq 18, both the conditions have actions.



Decision Tree

Example: The hotel ABC offers new discount scheme foe university students. It provides 15% and 10% discount to monthly customers for vegetarian and non-vegetarian meal respectively. The student attend hotel occasionally have discount rate 10% and 5% for vegetarian and non-vegetarian meal.

Draw decision tree to show discount scheme:



Example of Decision tree

CHAPTER 3 PROGRAM STRUCTURE

Introduction to program structure

Program structure The overall form of a program, with particular emphasis on the individual components of the program and the interrelationships between these components. Programs are frequently referred to as either *well-structured* or *poorly structured*.

Format of a structured programming language

A C program basically has the following form:

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

The following program is written in the C programming language. Open a text file *hello.c* using vi editor and put the following lines inside that file.

```
#include <stdio.h>

int main()
{
    /* My first program */
    printf("Hello, World! \n");

    return 0;
}
```

Preprocessor Commands: These commands tell the compiler to do preprocessing before doing actual compilation. Like *#include <stdio.h>* is a preprocessor command which tells a C compiler to include *stdio.h* file before going to actual compilation. You will learn more about C Preprocessors in [C Preprocessors](#) session.

Functions: are main building blocks of any C Program. Every C Program will have one or more functions and there is one mandatory function which is called *main()* function. This function is prefixed with keyword *int* which means this function returns an integer value when it exits. This integer value is returned using *return* statement.

The C Programming language provides a set of built-in functions. In the above example *printf()* is a C built-in function which is used to print anything on the screen.

Variables: are used to hold numbers, strings and complex data for manipulation.

Statements & Expressions : Expressions combine variables and constants to create new values. Statements are expressions, assignments, function calls, or control flow statements which make up C programs.

Comments: are used to give additional useful information inside a C Program. All the comments will be put inside `/*...*/` as given in the example above. A comment can span through multiple lines.

Note:

- There should be a `main ()` function somewhere in the program to determine where to start the executions.
- Usually all C statements are entered in small case letters.
- The group of statements in `main ()` are executed sequentially.
- The left brace indicates the program opening.
- The right brace indicates the program closing.
- In C language Comments are enclosed with `/* -- */` means these statements won't execute when the program is compiled.

A C program consists of one or more functions. Each function performs a specific task. A function is a group or sequence of C statements that are executed together.

Every C program starts with a function called **main()**. This is the place where program execution begins. Hence, there should be **main()** function in every C program. The functions are building blocks of C program. Each function has a name and a list of parameters.

The following are some rules to write C programs

1. All C statements must end with semicolon.
2. C is case – sensitive. That is, upper case and lower case characters are different. Generally the statements are typed in lower case.
3. A C statement can be written in one line or it can split into multiple lines.
4. Braces must always match upon pairs, i.e., every opening brace '{' must have a matching closing brace '}'.
5. A comment can be split into more than one line.

Basic C Programs - Variable and I/O instructions

1) Print Hello Word

Let us look at a simple code that would print the words "Hello World":

```
#include <stdio.h>

int main()
{
    /* my first program in C */
    printf("Hello, World! \n");

    return 0;
}
```

Let us look various parts of the above program:

1. The first line of the program *#include <stdio.h>* is a preprocessor command, which tells a C compiler to include *stdio.h* file before going to actual compilation.
2. The next line *int main()* is the main function where program execution begins.
3. The next line */*...*/* will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.
4. The next line *printf(...)* is another function available in C which causes the message "Hello, World!" to be displayed on the screen.
5. The next line **return 0;** terminates *main()* function and returns the value 0.

Note: Compile & Execute C Program:

Let's look at how to save the source code in a file, and how to compile and run it. Following are the simple steps:

1. Open a text editor and add the above-mentioned code.
2. Save the file as *hello.c*
3. Open a command prompt and go to the directory where you saved the file.
4. Type *gcc hello.c* and press enter to compile your code.
5. If there are no errors in your code the command prompt will take you to the next line and would generate *a.out* executable file.
6. Now, type *a.out* to execute your program.
7. You will be able to see "Hello World" printed on the screen

2) Declaring Variable Inputing and Outputting Value

A variable is nothing but a name given to a storage area that our programs can manipulate.

Each variable in C has a specific *type*, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

- a)** Working with variables - Let's look at an example of how to declare an integer variable in the C language, assign value and read the value in the variable to output.

For example:

```
int age;
```

In this example, the variable named *age* would be defined as an int.

Below is an example C program where we declare this variable:

```
#include <stdio.h>

int main()
{
    int age;

    age = 10;
    printf("TechOnTheNet.com is over %d years old.\n", age);

    return 0;
}
```

This C program would print "TechOnTheNet.com is over 10 years old."

Conversion specifiers are made up of two characters: % and a special character.

The special character tells the program how to convert the data.

Conversion Specifier Description

%d	Displays integer value
%f	Displays floating-point numbers
%c	Displays character

Format specifiers are the operators used in printf() function to print the data which is referred by an object or a variable. when a value is stored in a particular variable, Then we cannot print the value stored in the variable directly with out the using format specifiers. We can retrieve the data stored in the variables and can print them on to the console screen by using these format specifiers in a printf function. Format specifiers start with a percentage(%) symbol and follows a special character to identify the type of the data. There are basically six types of format specifiers are available in c they are

%d - represent integer values

%f - represent float values

%c - represent single character values

%s - represent string values

%u - represent the address of a variable

%ld – represent Long integer values

More examples: Use the printf function with formatting options.

```
#include <stdio.h>

main() {

    int x;
    float y;
    char c;

    x = -4443;
    y = 554.21;
    c = 'M';

    printf("\nThe value of integer variable x is %d", x);
    printf("\nThe value of float variable y is %f", y);
    printf("\nThe value of character variable c is %c\n", c);

}
The value of integer variable x is -4443
    The value of float variable y is 554.210022
    The value of character variable c is M
```

More examples: Use the scanf function with formatting options assign to assign value to variables.

This C program would print "*Your Name is Kim and you age is over 10 yrs old.*"

```
#include <stdio.h>

main() {

    int Age;
    char Name[10];

    printf("\nPlz enter your Name ");
    scanf("%s", &Name);
    printf("\nPlz enter your Age", y);
    scanf("%d", &Age);
    printf("\nYour Name is %s and your age is %d yrs old\n", Name, Age);

}
```


Program Structures in Pascal

Before we study basic building blocks of the Pascal programming language, let us look a bare minimum Pascal program structure so that we can take it as a reference in upcoming chapters.

Pascal Program Structure

A Pascal program basically consists of the following parts –

- Program name
- Uses command
- Type declarations
- Constant declarations
- Variables declarations
- Functions declarations
- Procedures declarations
- Main program block
- Statements and Expressions within each block
- Comments

Every pascal program generally has a heading statement, a declaration and an execution part strictly in that order. Following format shows the basic syntax for a Pascal program –

```
program {name of the program}
uses {comma delimited names of libraries you use}
const {global constant declaration block}
var {global variable declaration block}

function {function declarations, if any}
{ local variables }
begin
...
end;

procedure { procedure declarations, if any}
{ local variables }
begin
...
end;

begin { main program block starts}
...
end. { the end of main program block }
```

Pascal Hello World Example

Following is a simple pascal code that would print the words "Hello, World!":

```
program HelloWorld;
uses crt;

(* Here the main program block starts *)
```

```
begin
    writeln('Hello, World!');
    readkey;
end.
```

This will produce following result –

```
Hello, World!
```

Let us look various parts of the above program –

- The first line of the program **program HelloWorld;** indicates the name of the program.
- The second line of the program **uses crt;** is a preprocessor command, which tells the compiler to include the crt unit before going to actual compilation.
- The next lines enclosed within begin and end statements are the main program block. Every block in Pascal is enclosed within a **begin** statement and an **end** statement. However, the end statement indicating the end of the main program is followed by a full stop (.) instead of semicolon (;).
- The **begin** statement of the main program block is where the program execution begins.
- The lines within (*...*) will be ignored by the compiler and it has been put to add a **comment** in the program.
- The statement **writeln('Hello, World!');** uses the writeln function available in Pascal which causes the message "Hello, World!" to be displayed on the screen.
- The statement **readkey;** allows the display to pause until the user presses a key. It is part of the crt unit. A unit is like a library in Pascal.
- The last statement **end.** ends your program.

Pascal - Basic Syntax

You have seen a basic structure of pascal program, so it will be easy to understand other basic building blocks of the pascal programming language.

Variables

A variable definition is put in a block beginning with a **var** keyword, followed by definitions of the variables as follows:

```
var
A_Variable, B_Variable ... : Variable_Type;
```

i.e. A, b, c:integer

Pascal variables are declared outside the code-body of the function which means they are not declared within the **begin** and **end** pairs, but they are declared after the definition of the procedure/function and before the **begin** keyword. For global variables, they are defined after the program header.

Functions/Procedures

In Pascal, a **procedure** is set of instructions to be executed, with no return value and a **function** is a procedure with a return value. The definition of function/procedures will be as follows –

```
Function Func Name(params...) : Return Value;
```

```
Procedure Proc_Name(params...);
```

Comments

The multiline comments are enclosed within curly brackets and asterisks as { * ... * }. Pascal allows single-line comment enclosed within curly brackets { ... }.

```
{* This is a multi-line comments
   and it will span multiple lines. *}

{ This is a single line comment in pascal }
```

Case Sensitivity

Pascal is a case non-sensitive language, which means you can write your variables, functions and procedure in either case. Like variables A_Variable, a_variable and A_VARIABLE have same meaning in Pascal.

Pascal Statements (I/O)

Pascal programs are made of statements. Each statement specifies a definite job of the program. These jobs could be declaration, assignment, reading data, writing data, taking logical decisions, transferring program flow control, etc.

For example –

```
readln (a, b, c);
s := (a + b + c)/2.0;
area := sqrt(s * (s - a)*(s-b)*(s-c));
writeln(area);
```

Note: All pascal functions and control structures start with name and follow with Begin..End in place of curly bracket “{ }” used in C programs.

1. Function Example

```
function name(argument(s): type1; argument(s): type2; ...): function_type;
local declarations;
begin
    ...
    < statements >
    ...
    name:= expression;
end;

program exFunction;
var
    a, b, ret : integer;
```

2. Function and control structure Example

```
(*function definition *)
function max(num1, num2: integer): integer;
var
    (* local variable declaration *)
    result: integer;
```

```

begin
    if (num1 > num2) then
        begin
            result := num1;
            writeln(result);
        end;
    else
        begin
            result := num2;
            max := result;
            writeln(max);
        end;
    end;

begin
    a := 100;
    b := 200;
    (* calling a function to get max value *)
    ret := max(a, b);

    writeln( 'Max value is : ', ret );
end.

```

Data types, identifiers and operators

A C identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore _ followed by zero or more letters, underscores, and digits (0 to 9).

C does not allow punctuation characters such as @, \$, and % within identifiers. C is a case sensitive programming language. Thus, Manpower and manpower are two different identifiers in C. Here are some examples of acceptable identifiers:

Mohd, zara, abc, move_name, a_123, myname50, _temp, j, a23b9, retVal

Keywords reserved words in C. These reserved words may not be used as constant or variable or any other identifier names.

Following list shows the reserved words in C.

auto	else	Long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_packed
double			

Whitespace - A line containing only whitespace, possibly with a comment, is known as a blank line, and a C compiler totally ignores it.

Whitespace is the term used in C to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as `int`, ends and the next element begins. Therefore, in the following statement:

Data type

In the C programming language, data types refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The types in C can be classified as follows:

S.N.	Types and Description
1	Basic Types: They are arithmetic types and consists of the two types: (a) integer types and (b) floating-point types.
2	Enumerated types: They are again arithmetic types and they are used to define variables that can only be assigned certain discrete integer values throughout the program.
3	The type void: The type specifier <i>void</i> indicates that no value is available.
4	Derived types: They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

Basic Data types

C has a concept of 'data types' which are used to define a variable before its use. The definition of a variable will assign storage for the variable and define the type of data that will be held in the location.

The value of a variable can be changed any time.

C has the following basic built-in datatypes.

- `int`
- `float`
- `double`
- `char`

Please note that there is not a boolean data type. C does not have the traditional view about logical comparison, but thats another story.

Int - data type

`int` is used to define integer numbers.

```
{
    int Count;
    Count = 5;
}
```

Float - data type

float is used to define floating point numbers.

```
{  
    float Miles;  
    Miles = 5.6;  
}
```

Double - data type

double is used to define BIG floating point numbers. It reserves twice the storage for the number. On PCs this is likely to be 8 bytes.

```
{  
    double Atoms;  
    Atoms = 2500000;  
}
```

Char - data type

char defines characters.

```
{  
    char Letter;  
    Letter = 'x';  
}
```

Variable Types

A variable is just a named area of storage that can hold a single value (numeric or character). The C language demands that you declare the name of each variable that you are going to use and its type, or class, before you actually try to do anything with it.

The Programming language C has two main variable types

- Local Variables
- Global Variables

Local Variables

- Local variables scope is confined within the block or function where it is defined. Local variables must always be defined at the top of a block.
- When a local variable is defined - it is not initialised by the system, you must initialise it yourself.
- When execution of the block starts the variable is available, and when the block ends the variable 'dies'.

Check following example's output

```
main()
{
    int i=4;
    int j=10;

    i++;

    if (j > 0)
    {
        /* i defined in 'main' can be seen */
        printf("i is %d\n",i);
    }

    if (j > 0)
    {
        /* 'i' is defined and so local to this block */
        int i=100;
        printf("i is %d\n",i);
    } /* 'i' (value 100) dies here */

    printf("i is %d\n",i); /* 'i' (value 5) is now visible.*/
}

This will generate following output
i is 5
i is 100
i is 5
```

Here ++ is called incremental operator and it increase the value of any integer variable by 1. Thus **i++** is equivalent to $i = i + 1$;

You will see -- operator also which is called decremental operator and it decrease the value of any integer variable by 1. Thus **i--** is equivalent to $i = i - 1$;

Global Variables

Global variable is defined at the top of the program file and it can be visible and modified by any function that may reference it.

Global variables are initialised automatically by the system when you define them!

Data Type	Initialser
int	0
char	'\0'
float	0
pointer	NULL

If same variable name is being used for global and local variable then local variable takes preference in its scope. But it is not a good practice to use global variables and local variables with the same name.

```
int i=4;          /* Global definition */

main()
{
    i++;          /* Global variable */
    func();
    printf( "Value of i = %d -- main function\n", i );
}

func()
{
    int i=10;      /* Local definition */
    i++;           /* Local variable */
    printf( "Value of i = %d -- func() function\n", i );
}

This will produce following result
Value of i = 11 -- func() function
Value of i = 5 -- main function
```

i in **main** function is global and will be incremented to 5. **i** in **func** is internal and will be incremented to 11. When control returns to **main** the internal variable will die and any reference to **i** will be to the global.

Operator Types

What is Operator? Simple answer can be given using expression *4 + 5 is equal to 9*. Here 4 and 5 are called operands and + is called operator. C language supports following type of operators.

- Arithmetic Operators
- Logical (or Relational) Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

Lets have a look on all operators one by one.

Arithmetic Operators:

There are following arithmetic operators supported by C language:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiply both operands	A * B will give 200

/	Divide numerator by denominator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

Example

Assume variable A holds 10 and variable B holds 20 then:

Try following example to understand all the arithmetic operators. Copy and paste following C program in test.c file and compile and run this program.

```
main()
{
    int a = 21;
    int b = 10;
    int c ;

    c = a + b;
    printf("Line 1 - Value of c is %d\n", c );
    c = a - b;
    printf("Line 2 - Value of c is %d\n", c );
    c = a * b;
    printf("Line 3 - Value of c is %d\n", c );
    c = a / b;
    printf("Line 4 - Value of c is %d\n", c );
    c = a % b;
    printf("Line 5 - Value of c is %d\n", c );
    c = a++;
    printf("Line 6 - Value of c is %d\n", c );
    c = a--;
    printf("Line 7 - Value of c is %d\n", c );
}
```

This will produce following result

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2
Line 5 - Value of c is 1
```

```
Line 6 - Value of c is 21
Line 7 - Value of c is 22
```

Logical (or Relational) Operators:

There are following logical operators supported by C language

Operator	Description	Example
==	Checks if the value of two operands is equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the value of two operands is equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.
&&	Called Logical AND operator. If both the operands are non zero then then condition becomes true.	(A && B) is true.
	Called Logical OR Operator. If any of the two operands is non zero then then condition becomes true.	(A B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is false.

Examples

Assume variable A holds 10 and variable B holds 20 then:

Try following example to understand all the Logical operators. Copy and paste following C program in test.c file and compile and run this program.

```
main()
{
    int a = 21;
    int b = 10;
    int c ;

    if( a == b )
    {
        printf("Line 1 - a is equal to b\n" );
    }
    else
    {
        printf("Line 1 - a is not equal to b\n" );
    }
    if ( a < b )
    {
        printf("Line 2 - a is less than b\n" );
    }
    else
    {
        printf("Line 2 - a is not less than b\n" );
    }
    if ( a > b )
    {
        printf("Line 3 - a is greater than b\n" );
    }
    else
    {
        printf("Line 3 - a is not greater than b\n" );
    }
    /* Lets change value of a and b */
    a = 5;
    b = 20;
    if ( a <= b )
    {
        printf("Line 4 - a is either less than or euqal to b\n" );
    }
    if ( b >= a )
    {
        printf("Line 5 - b is either greater than or equal to b\n" );
    }
    if ( a && b )
    {
        printf("Line 6 - Condition is true\n" );
    }
    if ( a || b )
    {
        printf("Line 7 - Condition is true\n" );
    }
}
```

```

/* Again lets change the value of a and b */
a = 0;
b = 10;
if ( a && b )
{
    printf("Line 8 - Condition is true\n" );
}
else
{
    printf("Line 8 - Condition is not true\n" );
}
if ( !(a && b) )
{
    printf("Line 9 - Condition is true\n" );
}
}

```

This will produce following result

```

Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either less than or euqal to b
Line 5 - b is either greater than or equal to b
Line 6 - Condition is true
Line 7 - Condition is true
Line 8 - Condition is not true
Line 9 - Condition is true

```

Bitwise Operators:

Bitwise operator works on bits and perform bit by bit operation.

Assume if A = 60; and B = 13; Now in binary format they will be as follows:

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

There are following Bitwise operators supported by C language

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Examples

Try following example to understand all the Bitwise operators. Copy and paste following C program in test.c file and compile and run this program.

```
main()
{
    unsigned int a = 60;          /* 60 = 0011 1100 */
    unsigned int b = 13;         /* 13 = 0000 1101 */
    int c = 0;

    c = a & b;                    /* 12 = 0000 1100 */
    printf("Line 1 - Value of c is %d\n", c );

    c = a | b;                    /* 61 = 0011 1101 */
    printf("Line 2 - Value of c is %d\n", c );

    c = a ^ b;                    /* 49 = 0011 0001 */
    printf("Line 3 - Value of c is %d\n", c );

    c = ~a;                       /* -61 = 1100 0011 */
    printf("Line 4 - Value of c is %d\n", c );

    c = a << 2;                   /* 240 = 1111 0000 */
    printf("Line 5 - Value of c is %d\n", c );
}
```

```

c = a >> 2;      /* 15 = 0000 1111 */
printf("Line 6 - Value of c is %d\n", c );
}

```

This will produce following result

```

Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15

```

Assignment Operators:

There are following assignment operators supported by C language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	$C = A + B$ will assign value of $A + B$ into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	$C += A$ is equivalent to $C = C + A$
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	$C -= A$ is equivalent to $C = C - A$
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	$C *= A$ is equivalent to $C = C * A$
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	$C /= A$ is equivalent to $C = C / A$

%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
=	bitwise inclusive OR and assignment operator	C = 2 is same as C = C 2

Examples

Try following example to understand all the Assignment Operators. Copy and paste following C program in test.c file and compile and run this program.

```
main()
{
    int a = 21;
    int c ;

    c = a;
    printf("Line 1 - = Operator Example, Value of c = %d\n", c );

    c += a;
    printf("Line 2 - += Operator Example, Value of c = %d\n", c );

    c -= a;
    printf("Line 3 - -= Operator Example, Value of c = %d\n", c );

    c *= a;
    printf("Line 4 - *= Operator Example, Value of c = %d\n", c );

    c /= a;
    printf("Line 5 - /= Operator Example, Value of c = %d\n", c );

    c = 200;
    c %= a;
    printf("Line 6 - %= Operator Example, Value of c = %d\n", c );

    c <<= 2;
    printf("Line 7 - <<= Operator Example, Value of c = %d\n", c );

    c >>= 2;
    printf("Line 8 - >>= Operator Example, Value of c = %d\n", c );
}
```

```

c &= 2;
printf("Line 9 - &= Operator Example, Value of c = %d\n", c );

c ^= 2;
printf("Line 10 - ^= Operator Example, Value of c = %d\n", c );

c |= 2;
printf("Line 11 - |= Operator Example, Value of c = %d\n", c );
}

```

This will produce following result

```

Line 1 - = Operator Example, Value of c = 21
Line 2 - += Operator Example, Value of c = 42
Line 3 - -= Operator Example, Value of c = 21
Line 4 - *= Operator Example, Value of c = 441
Line 5 - /= Operator Example, Value of c = 21
Line 6 - %= Operator Example, Value of c = 11
Line 7 - <<= Operator Example, Value of c = 44
Line 8 - >>= Operator Example, Value of c = 11
Line 9 - &= Operator Example, Value of c = 2
Line 10 - ^= Operator Example, Value of c = 0
Line 11 - |= Operator Example, Value of c = 2

```

Short Notes on L-VALUE and R-VALUE:

$x = 1$; takes the value on the right (e.g. 1) and puts it in the memory referenced by x . Here x and 1 are known as L-VALUES and R-VALUES respectively L-values can be on either side of the assignment operator where as R-values only appear on the right.

So x is an L-value because it can appear on the left as we've just seen, or on the right like this: $y = x$; However, constants like 1 are R-values because 1 could appear on the right, but $1 = x$; is invalid.

Misc Operators

There are few other operators supported by C Language.

Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is interger, will return 4.
&	Returns the address of a variable.	&a; will give actaul address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable.
?:	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

Examples

sizeof Operator:

Try following example to understand sizeof operators. Copy and paste following C program in test.c file and compile and run this program.

```
main()
{
    int a;
    short b;
    double double c;
    char d[10];

    printf("Line 1 - Size of variable a = %d\n", sizeof(a) );
    printf("Line 2 - Size of variable b = %d\n", sizeof(b) );
    printf("Line 3 - Size of variable c= %d\n", sizeof(c) );
    printf("Line 4 - Size of variable d= %d\n", sizeof(d) );
    /* For character string strlen should be used instead of sizeof */
    printf("Line 5 - Size of variable d= %d\n", strlen(d) );
}
```

This will produce following result

```
Line 1 - Size of variable a = 4
Line 2 - Size of variable b = 2
Line 3 - Size of variable c= 8
Line 4 - Size of variable d= 10
Line 5 - Size of variable d= 10
```

& and * Operators:

Try following example to understand & operators. Copy and paste following C program in test.c file and compile and run this program.

```
main()
{
    int i=4;           /* variable declaration      */
    int* ptr;          /* int pointer          */

    ptr = &i;          /* 'ptr' now contains the
                        address of 'i'          */

    printf(" i is  %d.\n", i);
    printf("*ptr is %d.\n", *ptr);
}
```

This will produce following result

```
i is 4.
*ptr is 4.
```

? : Operator

Try following example to understand ? : operators. Copy and paste following C program in test.c file and compile and run this program.

```
main()
{
    int a , b;

    a = 10;
    b = (a == 1) ? 20: 30;
    printf( "Value of b is %d\n", b );

    b = (a == 10) ? 20: 30;
    printf( "Value of b is %d\n", b );
}
```

This will produce following result

```
Value of b is 30
Value of b is 20
```

Operators Categories:

All the operators we have discussed above can be categorised into following categories:

- Postfix operators, which follow a single operand.
- Unary prefix operators, which precede a single operand.
- Binary operators, which take two operands and perform a variety of arithmetic and logical operations.
- The conditional operator (a ternary operator), which takes three operands and evaluates either the second or third expression, depending on the evaluation of the first expression.
- Assignment operators, which assign a value to a variable.
- The comma operator, which guarantees left-to-right evaluation of comma-separated expressions.

Precedence of C Operators:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example $x = 7 + 3 * 2$; Here x is assigned 13, not 20 because operator * has higher precedence than + so it first gets multiplied with $3*2$ and then adds into 7.

Here operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

CHAPTER 4: PROGRAM WRITING

Content of structured programming

The terms that follow will be used frequently throughout the C. you should be completely familiar with them.

- Source Code: The text of a program that a user can read commonly thought of as the program. The source code is input into the C compiler.
- Object Code: Translation of the source code of a program into machine code, which the computer can read and execute directly. Object code is input to the linker.
- Linker: A program that links separately compiled modules into one program. It also combines the functions in the Standard C library with the code that you wrote. The output of the linker is an executable program.
- Library: The file containing the standard functions that your program can use. These functions include all I/O operations as well as other useful routines.
- Compile Time: The time during which your program is being compiled.
- Run Time: The time during which your program is executing.

Compiling a C program - Creating an executable form of your C program consists of these three steps:

1. Creating your program
2. Compiling your program
3. Linking your program with whatever function needed from the library.

Compilers only accept standard text files for input.

- The first step in executing a C program is to compile the Source program. Compilation can be accomplished by selecting the compile menu or by pressing 'Alt + c'.
- A small window is displayed which gives the information of the lines compiled and the Errors & Warnings in the program. If there are any Errors, they are to be detected. After compiling the source file, an object file with .obj extension is created.

Once the program is error free then we can execute the program by selecting the 'Run' menu or by pressing 'Alt + R'. After executing a program an executable file with .exe extension is created.

Steps to Develop a Program

The following steps are used in sequence for developing an efficient program:

- Specifying the problem statement
- Designing an algorithm
- Coding
- Debugging
- Testing and Validating
- Documentation and Maintenance.

Specifying the Problem: The Problem which has to be implemented into a program must be thoroughly understood before the program is written. Problem must be analyzed to determine the input and output requirements of the program. A problem is created with these specifications.

Designing an Algorithm: With the problem statement obtained in the previous step, various methods available for obtaining the required solution are analyzed and the best suitable method is designed into algorithm

To improve clarity and understandability of the program flow charts are drawn using the algorithms.

Coding: The actual program is written in the required programming language with the help of information depicted in flow charts and algorithms.

Debugging: There is a possibility of occurrence of errors in programs. These errors must be removed to ensure proper working of programs. Hence error check is made. This process is known as “Debugging”.

Types of errors that may occur in the program are:

- **Syntactic Errors:** These errors occur due to the usage of wrong syntax for the statements.

Syntax means rules of writing the program.

Example: $x = z * / b;$

There is syntax error in this statement. The rules of binary operators state that there cannot be more than one operator between two operands.

- **Runtime Errors:** These Errors are determined at the execution time of the program.

Example: Divide by zero

Range out of bounds

Square root of a negative number

- **Logical Errors:** These Errors occur due to incorrect usage of the instruction in the program. These errors are neither displayed during compilation or execution nor cause any obstruction to the program execution. They only cause incorrect outputs. Logical Errors are determined by analyzing the outputs for different possible inputs that can be applied to the program. By this way the program is validated.

Testing and Validating: Testing and Validation is performed to check whether the program is producing correct results or not for different values of input.

Documentation and Maintenance: Documentation is the process of collecting, organizing and maintaining, in written the complete information of the program for future references.

Maintenance is the process of upgrading the program according to the changing requirements.

For writing up the instructions as a program in the way that a computer can understand, we use programming languages.

Error Handling

As such, C programming does not provide direct support for error handling but being a system programming language, it provides you access at lower level in the form of return values. Most of the C or even Unix function calls return -1 or NULL in case of any error and set an error code **errno**. It is set as a global variable and indicates an error occurred during any function call. You can find various error codes defined in <error.h> header file.

So a C programmer can check the returned values and can take appropriate action depending on the return value. It is a good practice, to set `errno` to 0 at the time of initializing a program. A value of 0 indicates that there is no error in the program.

errno, perror(), and strerror()

The C programming language provides **perror()** and **strerror()** functions which can be used to display the text message associated with **errno**.

- The **perror()** function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current `errno` value.
- The **strerror()** function, which returns a pointer to the textual representation of the current `errno` value.

Let's try to simulate an error condition and try to open a file which does not exist. Here I'm using both the functions to show the usage, but you can use one or more ways of printing your errors. Second important point to note is that you should use **stderr** file stream to output all the errors.

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

extern int errno ;

int main () {

    FILE * pf;
    int errnum;
    pf = fopen ("unexist.txt", "rb");

    if (pf == NULL) {

        errnum = errno;
        fprintf(stderr, "Value of errno: %d\n", errno);
        perror("Error printed by perror");
        fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
    }
    else {

        fclose (pf);
    }

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Value of errno: 2
Error printed by perror: No such file or directory
Error opening file: No such file or directory
```

Divide by Zero Errors

It is a common problem that at the time of dividing any number, programmers do not check if a divisor is zero and finally it creates a runtime error.

The code below fixes this by checking if the divisor is zero before dividing –

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

main() {

    int dividend = 20;
    int divisor = 0;
    int quotient;

    if( divisor == 0){
        fprintf(stderr, "Division by zero! Exiting...\n");
        exit(-1);
    }

    quotient = dividend / divisor;
    fprintf(stderr, "Value of quotient : %d\n", quotient );

    exit(0);
}
```

When the above code is compiled and executed, it produces the following result –

```
Division by zero! Exiting...
```

Program Exit Status

It is a common practice to exit with a value of EXIT_SUCCESS in case of program coming out after a successful operation. Here, EXIT_SUCCESS is a macro and it is defined as 0.

If you have an error condition in your program and you are coming out then you should exit with a status EXIT_FAILURE which is defined as -1. So let's write above program as follows –

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

main() {

    int dividend = 20;
    int divisor = 5;
    int quotient;

    if( divisor == 0) {
        fprintf(stderr, "Division by zero! Exiting...\n");
        exit(EXIT_FAILURE);
    }

    quotient = dividend / divisor;
    fprintf(stderr, "Value of quotient : %d\n", quotient );
}
```

```
    exit(EXIT_SUCCESS);  
}
```

When the above code is compiled and executed, it produces the following result –

```
Value of quotient: 4
```


CHAPTER 5: CONTROL STRUCTURES

Introduction to control structures

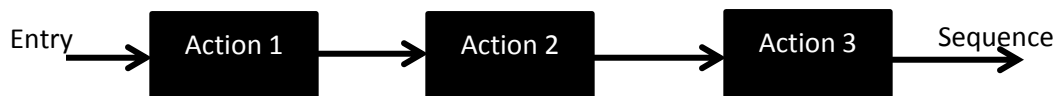
Control structures form the basic entities of a “**structured programming language**“. We all know languages like C/C++ or Java are all structured programming languages. **Control structures are used to alter the flow of execution of the program.** Why do we need to alter the program flow ? The reason is “**decision making**“! In life, we may be given with a set of option like doing “Electronics” or “Computer science”. We do make a decision by analyzing certain conditions (like our personal interest, scope of job opportunities etc). With the decision we make, we alter the flow of our life’s direction. This is exactly what happens in a C/C++ program. We use control structures to make decisions and alter the direction of program flow in one or the other path(s) available.

There are **three types** of control structures available in C and C++

- 1) **Sequence structure (straight line paths)**
- 2) **Selection structure (one or many branches)**
- 3) **Loop structure (repetition of a set of activities)**

Sequence structure

Sequence structure control structure and its flow of execution is represented in the flow charts given below.

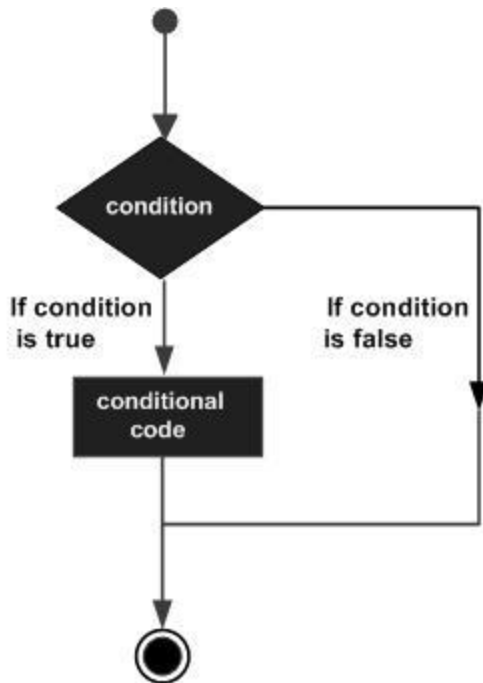


Execution of instructions flow in order from the first to the last in a linear manner

Selection/Decision Structure

Decision making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Show below is the general form of a typical decision making structure found in most of the programming languages –



C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

C programming language provides the following types of decision making statements.

if statement

An **if** statement consists of a Boolean expression followed by one or more statements.

Syntax

The syntax of an 'if' statement in C programming language is –

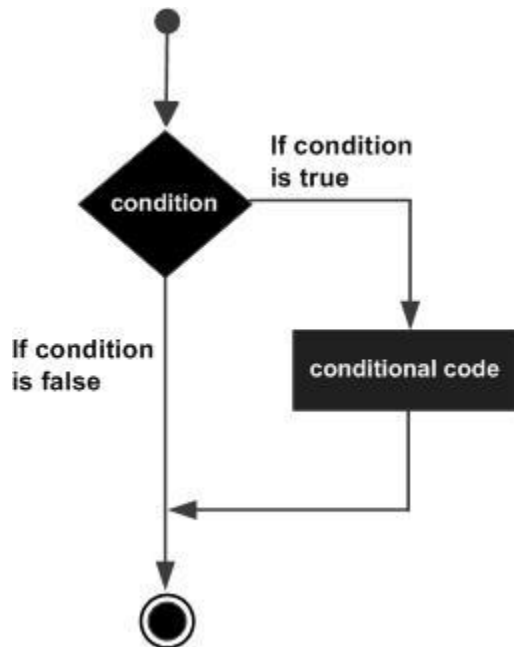
```

if(boolean_expression) {
    /* statement(s) will execute if the boolean expression is true */
}
  
```

If the Boolean expression evaluates to **true**, then the block of code inside the 'if' statement will be executed. If the Boolean expression evaluates to **false**, then the first set of code after the end of the 'if' statement (after the closing curly brace) will be executed.

C programming language assumes any **non-zero** and **non-null** values as **true** and if it is either **zero** or **null**, then it is assumed as **false** value.

Flow Diagram



Example

```
#include <stdio.h>

int main () {

    /* local variable definition */
    int a = 10;

    /* check the boolean condition using if statement */

    if( a < 20 ) {
        /* if condition is true then print the following */
        printf("a is less than 20\n" );
    }

    printf("value of a is : %d\n", a);

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
a is less than 20;
value of a is : 10
```

if...else statement

An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

Syntax

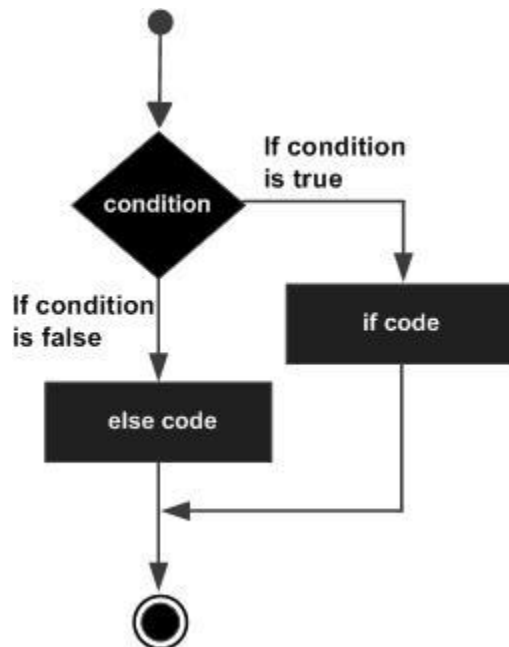
The syntax of an **if...else** statement in C programming language is –

```
if(boolean_expression) {  
    /* statement(s) will execute if the boolean expression is true */  
}  
else {  
    /* statement(s) will execute if the boolean expression is false */  
}
```

If the Boolean expression evaluates to **true**, then the **if block** will be executed, otherwise, the **else block** will be executed.

C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

Flow Diagram



Example

```
#include <stdio.h>  
  
int main () {  
  
    /* local variable definition */  
    int a = 100;  
  
    /* check the boolean condition */  
    if( a < 20 ) {  
        /* if condition is true then print the following */  
        printf("a is less than 20\n" );  
    }  
    else {  
        /* if condition is false then print the following */  
        printf("a is not less than 20\n" );  
    }  
}
```

```

        printf("a is not less than 20\n" );
    }

    printf("value of a is : %d\n", a);

    return 0;
}

```

When the above code is compiled and executed, it produces the following result –

```

a is not less than 20;
value of a is : 100

```

If...else if...else Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

When using if...else if..else statements, there are few points to keep in mind –

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax

The syntax of an **if...else if...else** statement in C programming language is –

```

if(boolean_expression 1) {
    /* Executes when the boolean expression 1 is true */
}
else if( boolean_expression 2) {
    /* Executes when the boolean expression 2 is true */
}
else if( boolean_expression 3) {
    /* Executes when the boolean expression 3 is true */
}
else {
    /* executes when the none of the above condition is true */
}

```

Example

```

#include <stdio.h>

int main () {

    /* local variable definition */
    int a = 100;

    /* check the boolean condition */
    if( a == 10 ) {
        /* if condition is true then print the following */
        printf("Value of a is 10\n" );
    }
    else if( a == 20 ) {
        /* if else if condition is true */

```

```

        printf("Value of a is 20\n" );
    }
    else if( a == 30 ) {
        /* if else if condition is true */
        printf("Value of a is 30\n" );
    }
    else {
        /* if none of the conditions is true */
        printf("None of the values is matching\n" );
    }

    printf("Exact value of a is: %d\n", a );

    return 0;
}

```

When the above code is compiled and executed, it produces the following result –

```

None of the values is matching
Exact value of a is: 100

```

nested if statements

It is always legal in C programming to **nest** if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

Syntax

The syntax for a **nested if** statement is as follows –

```

if( boolean_expression 1) {

    /* Executes when the boolean expression 1 is true */
    if(boolean_expression 2) {
        /* Executes when the boolean expression 2 is true */
    }
}

```

You can nest **else if...else** in the similar way as you have nested *if* statements.

Example

```

#include <stdio.h>

int main () {

    /* local variable definition */
    int a = 100;
    int b = 200;

    /* check the boolean condition */
    if( a == 100 ) {

        /* if condition is true then check the following */
        if( b == 200 ) {
            /* if condition is true then print the following */
            printf("Value of a is 100 and b is 200\n" );
        }
    }
}

```

```

    }
}

printf("Exact value of a is : %d\n", a );
printf("Exact value of b is : %d\n", b );

return 0;
}

```

When the above code is compiled and executed, it produces the following result –

```

Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200

```

Switch statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

Syntax

The syntax for a **switch** statement in C programming language is as follows –

```

switch(expression) {

    case constant-expression :
        statement(s);
        break; /* optional */

    case constant-expression :
        statement(s);
        break; /* optional */

    /* you can have any number of case statements */
    default : /* Optional */
        statement(s);
}

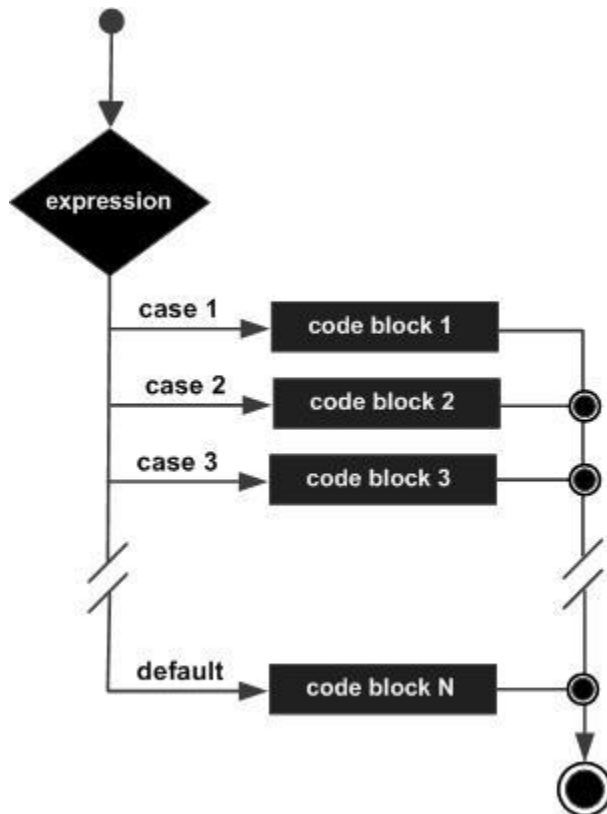
```

The following rules apply to a **switch** statement –

- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

- Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

Flow Diagram



Example

```

#include <stdio.h>

int main () {

    /* local variable definition */
    char grade = 'B';

    switch(grade) {
        case 'A' :
            printf("Excellent!\n" );
            break;
        case 'B' :
        case 'C' :
            printf("Well done\n" );
            break;
        case 'D' :
            printf("You passed\n" );
            break;
    }
}
  
```



```

        case 'F' :
            printf("Better try again\n" );
            break;
        default :
            printf("Invalid grade\n" );
    }

    printf("Your grade is  %c\n", grade );

    return 0;
}

```

When the above code is compiled and executed, it produces the following result –

```

Well done
Your grade is B

```

Nested switch statements

It is possible to have a switch as a part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

Syntax

The syntax for a **nested switch** statement is as follows –

```

switch(ch1) {

    case 'A':
        printf("This A is part of outer switch" );

        switch(ch2) {
            case 'A':
                printf("This A is part of inner switch" );
                break;
            case 'B': /* case code */
        }

        break;
    case 'B': /* case code */
}

```

Example

```

#include <stdio.h>

int main () {

    /* local variable definition */
    int a = 100;
    int b = 200;

    switch(a) {

        case 100:
            printf("This is part of outer switch\n", a );
            switch(b) {

```

```

        case 200:
            printf("This is part of inner switch\n", a );
        }

    printf("Exact value of a is : %d\n", a );
    printf("Exact value of b is : %d\n", b );

    return 0;
}

```

When the above code is compiled and executed, it produces the following result –

```

This is part of outer switch
This is part of inner switch
Exact value of a is : 100
Exact value of b is : 200

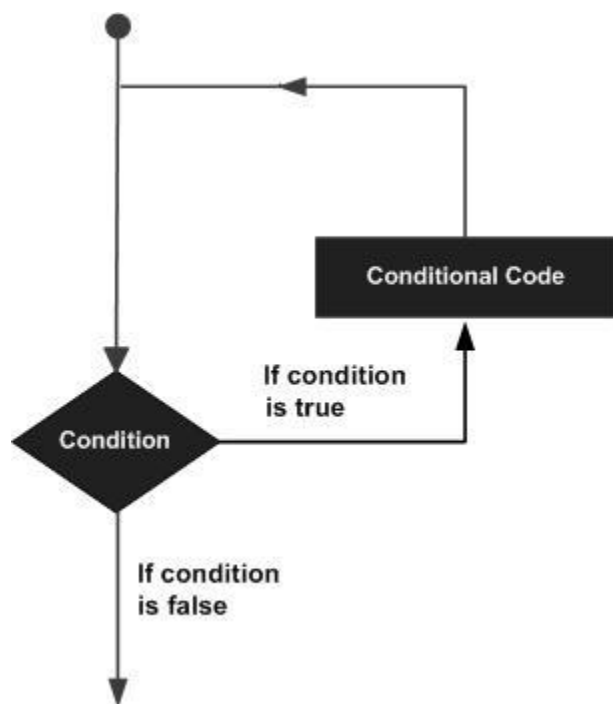
```

Loops/Iterations

You may encounter situations, when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages –



C programming language provides the following types of loops to handle looping requirements.

While loop

A **while** loop in C programming repeatedly executes a target statement as long as a given condition is true.

Syntax

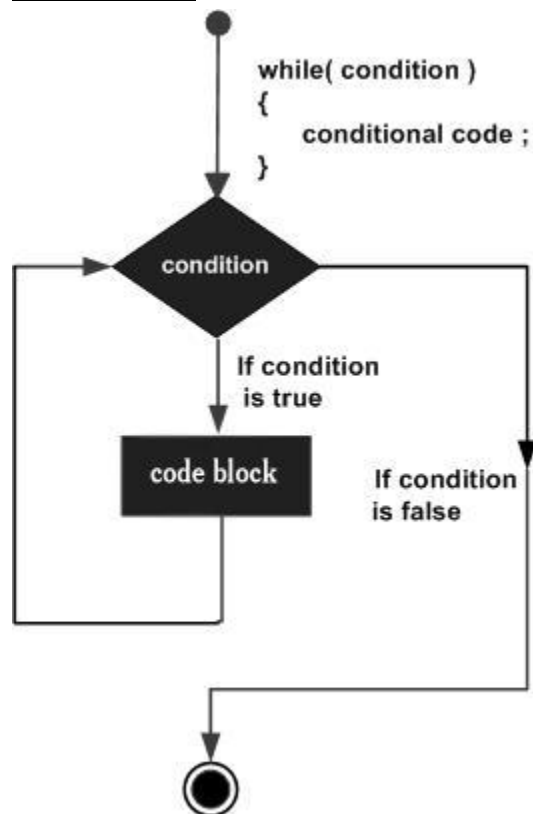
The syntax of a **while** loop in C programming language is –

```
while(condition) {  
    statement(s);  
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true.

When the condition becomes false, the program control passes to the line immediately following the loop.

Flow Diagram



Here, the key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
#include <stdio.h>

int main () {

    /* local variable definition */
    int a = 10;

    /* while loop execution */
    while( a < 20 ) {
        printf("value of a: %d\n", a);
        a++;
    }

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

for loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The syntax of a **for** loop in C programming language is –

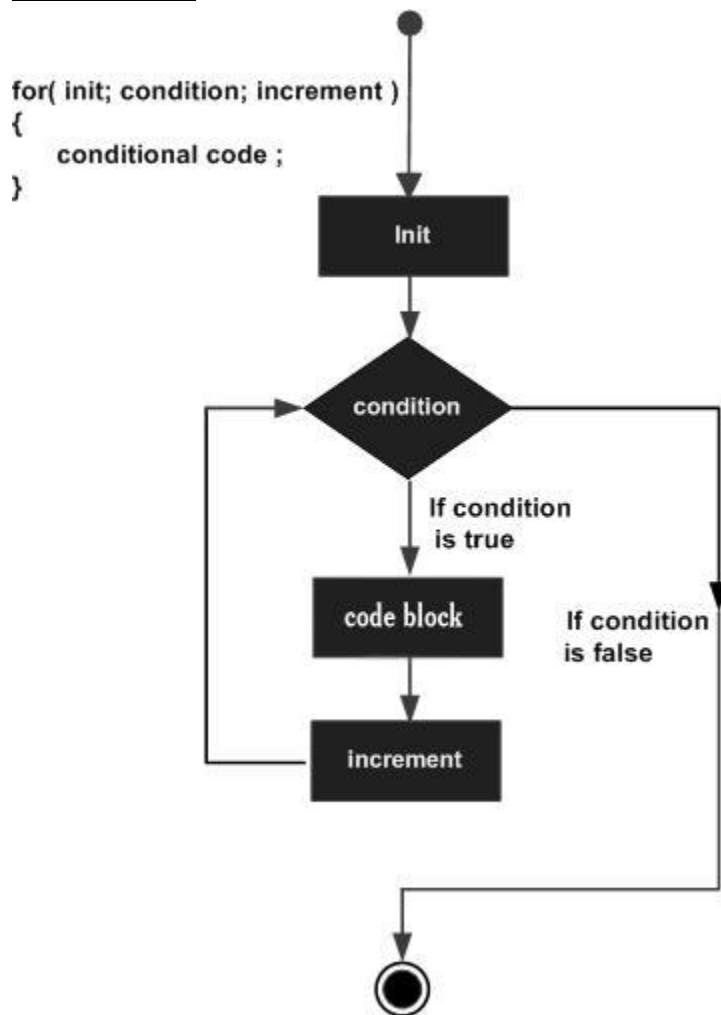
```
for ( init; condition; increment ) {
    statement(s);
}
```

Here is the flow of control in a 'for' loop –

- ✓ The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

- ✓ Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.
- ✓ After the body of the 'for' loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- ✓ The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.

Flow Diagram



Example

```
#include <stdio.h>

int main () {

    int a;

    /* for loop execution */
    for( a = 10; a < 20; a = a + 1 ){
        printf("value of a: %d\n", a);
    }

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

do...while loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

Syntax

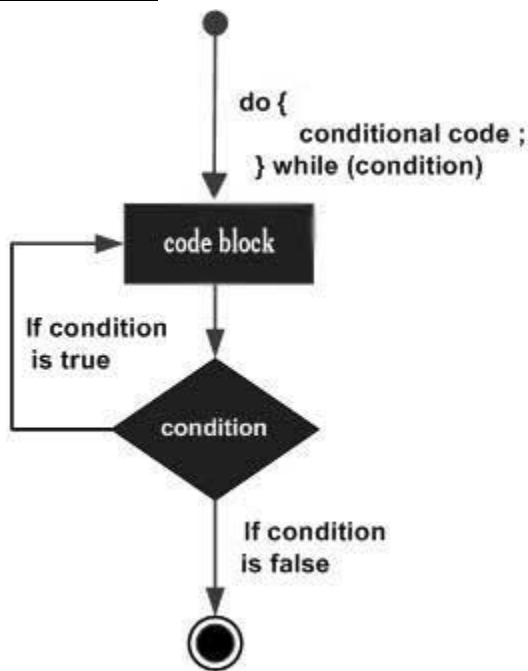
The syntax of a **do...while** loop in C programming language is –

```
do {
    statement(s);
} while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

Flow Diagram



Example

```
#include <stdio.h>

int main () {

    /* local variable definition */
    int a = 10;

    /* do loop execution */
    do {
        printf("value of a: %d\n", a);
        a = a + 1;
    }while( a < 20 );

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Nested loops

C programming allows to use one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax

The syntax for a **nested for loop** statement in C is as follows –

```
for ( init; condition; increment ) {  
  
    for ( init; condition; increment ) {  
        statement(s);  
    }  
  
    statement(s);  
}
```

The syntax for a **nested while loop** statement in C programming language is as follows –

```
while(condition) {  
  
    while(condition) {  
        statement(s);  
    }  
  
    statement(s);  
}
```

The syntax for a **nested do...while loop** statement in C programming language is as follows –

```
do {  
  
    statement(s);  
  
    do {  
        statement(s);  
    }while( condition );  
}while( condition );
```

A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example, a 'for' loop can be inside a 'while' loop or vice versa.

Example

The following program uses a nested for loop to find the prime numbers from 2 to 100 –

```
#include <stdio.h>  
  
int main () {  
  
    /* local variable definition */  
    int i, j;
```



```

for(i = 2; i<100; i++) {

    for(j = 2; j <= (i/j); j++)
        if(!(i%j)) break; // if factor found, not prime
    if(j > (i/j)) printf("%d is prime", i);
}

return 0;
}

```

When the above code is compiled and executed, it produces the following result –

```

2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime

```

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements.

Break statement

The **break** statement in C programming has the following two usages –

- ✓ When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- ✓ It can be used to terminate a case in the **switch** statement (covered in the next chapter).

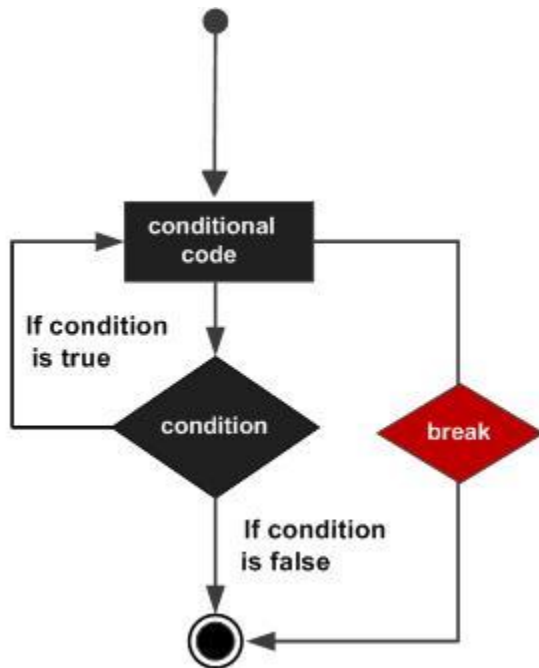
If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for a **break** statement in C is as follows –

```
break;
```

Flow Diagram



Example

```
#include <stdio.h>

int main () {

    /* local variable definition */
    int a = 10;

    /* while loop execution */
    while( a < 20 ) {

        printf("value of a: %d\n", a);
        a++;

        if( a > 15) {
            /* terminate the loop using break statement */
            break;
        }

    }

}
```

```
    return 0;  
}
```

When the above code is compiled and executed, it produces the following result –

```
value of a: 10  
value of a: 11  
value of a: 12  
value of a: 13  
value of a: 14  
value of a: 15
```

Continue statement

The **continue** statement in C programming works somewhat like the **break** statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between.

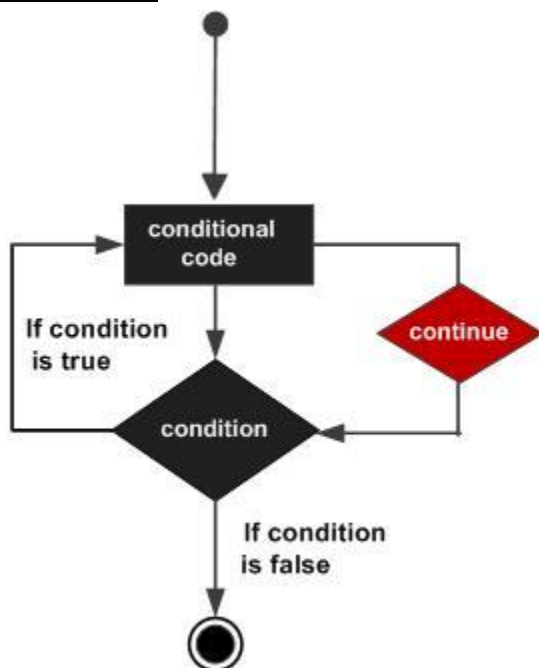
For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, **continue** statement causes the program control to pass to the conditional tests.

Syntax

The syntax for a **continue** statement in C is as follows –

```
continue;
```

Flow Diagram



Example

```
#include <stdio.h>

int main () {

    /* local variable definition */
    int a = 10;

    /* do loop execution */
    do {

        if( a == 15) {
            /* skip the iteration */
            a = a + 1;
            continue;
        }

        printf("value of a: %d\n", a);
        a++;

    } while( a < 20 );

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

goto statement

A **goto** statement in C programming provides an unconditional jump from the 'goto' to a labeled statement in the same function.

NOTE – Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten to avoid them.

Syntax

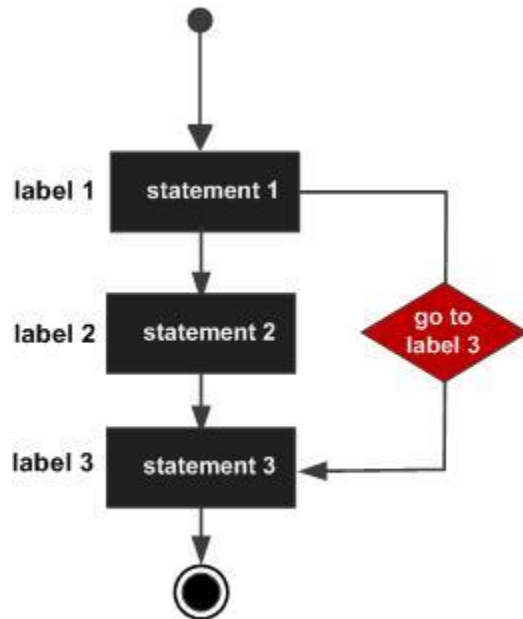
The syntax for a **goto** statement in C is as follows –

```
goto label;
..
.
```

```
label: statement;
```

Here **label** can be any plain text except C keyword and it can be set anywhere in the C program above or below to **goto** statement.

Flow Diagram



Example

```
#include <stdio.h>

int main () {

    /* local variable definition */
    int a = 10;

    /* do loop execution */
    LOOP:do {

        if( a == 15) {
            /* skip the iteration */
            a = a + 1;
            goto LOOP;
        }

        printf("value of a: %d\n", a);
        a++;

    }while( a < 20 );

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

The Infinite Loop

A loop becomes an infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.

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

```
int main () {

    for( ; ; ) {
        printf("This loop will run forever.\n");
    }

    return 0;
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(;;) construct to signify an infinite loop.

NOTE – You can terminate an infinite loop by pressing Ctrl + C keys.

CHAPTER 6: DATA STRUCTURES

Introduction to Data Structures

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way.

A data structure is a specialized format for organizing and storing data. A **data structure** is a particular way of organizing data in a computer so that it can be used efficiently.

Data structures can implement one or more particular **abstract data types**., which are the means of specifying the contract of operations and their complexity. In comparison, a data structure is a concrete implementation of the contract provided by an ADT.

an **abstract data type (ADT)** is a mathematical model for data types where a data type is defined by its behavior (semantics) from the point of view of a *user* of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. This contrasts with data structures, which are concrete representations of data, and are the point of view of an implementer, not a user. **Note:** ADTs are a theoretical concept in computer science, used in the design and analysis of algorithms, data structures, and software systems, and do not correspond to specific features of computer languages

Primitive and Non-Primitive data Types

Data type specifies the type of data stored in a variable. The data type can be classified into two types Primitive data type and Non-Primitive data type

PRIMITIVE DATATYPE The primitive data types are the basic data types that are available in most of the programming languages. The primitive data types are used to represent single values.

- ✓ **Integer:** This is used to represent a number without decimal point.
Eg: 12, 90
- ✓ **Float and Double:** This is used to represent a number with decimal point.
Eg: 45.1, 67.3
- ✓ **Character :** This is used to represent single character
Eg: 'C', 'a'
- ✓ **String:** This is used to represent group of characters.
Eg: "M.S.P.V.L Polytechnic College"
- ✓ **Boolean:** This is used represent logical values either true or false.

NON-PRIMITIVE DATATYPES The data types that are derived from primary data types are known as non-Primitive data types. These data types are used to store group of values.

The non-primitive data types are

- ✓ Arrays
- ✓ Structure
- ✓ Union
- ✓ linked list
- ✓ Stacks
- ✓ Queue etc

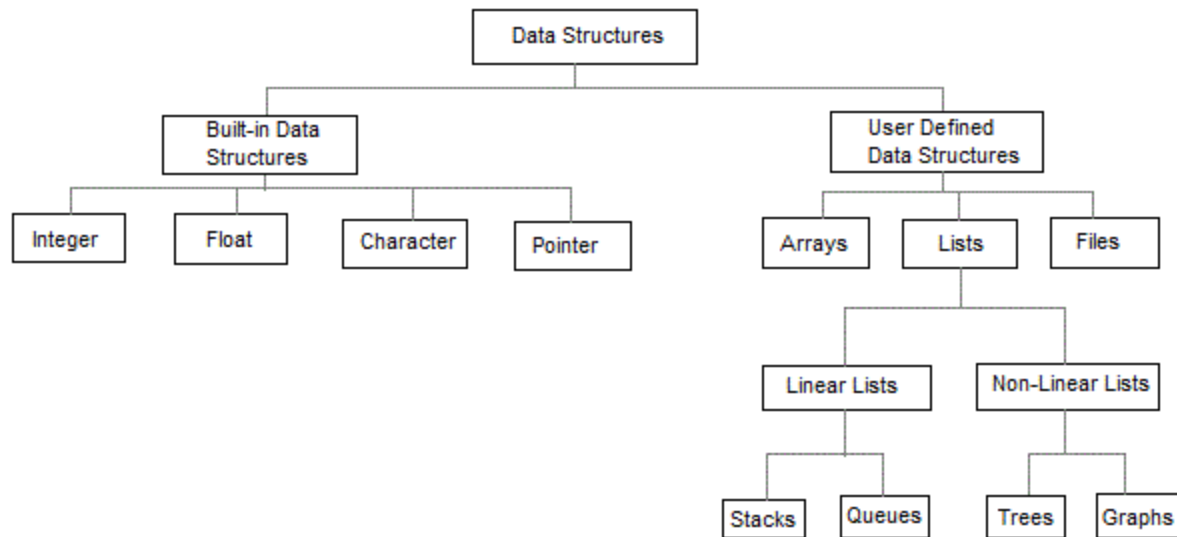


Figure: data structures

Algorithm

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as **pseudocode** or using a **flowchart**.

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties :

1. Time Complexity
2. Space Complexity

Space Complexity

Its the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

An algorithm generally requires space for following components :

- **Instruction Space** : Its the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
- **Data Space** : Its the space required to store all the constants and variables value.
- **Environment Space** : Its the space required to store the environment information needed to resume the suspended function.

Time Complexity

Time Complexity is a way to represent the amount of time needed by the program to run to completion. We will study this in details in our section.

Time Complexity of Algorithms

Time complexity of an algorithm signifies the total time required by the program to run to completion. The time complexity of algorithms is most commonly expressed using the **big O notation**.

Time Complexity is most commonly estimated by counting the number of elementary functions performed by the algorithm. And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

Calculating Time Complexity

Now lets tap onto the next big topic related to Time complexity, which is How to Calculate Time Complexity. It becomes very confusing some times, but we will try to explain it in the simplest way.

Now the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity. In general you can think of it like this :

```
statement;
```

Above we have a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to N.

```
for(i=0; i < N; i++)
{
    statement;
}
```

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for(i=0; i < N; i++)
{
    for(j=0; j < N; j++)
    {
        statement;
    }
}
```

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by $N * N$.

```

while(low <= high)
{
    mid = (low + high) / 2;
    if (target < list[mid])
        high = mid - 1;
    else if (target > list[mid])
        low = mid + 1;
    else break;
}

```

This is an algorithm to break a set of numbers into halves, to search a particular field(we will study this in detail later). Now, this algorithm will have a **Logarithmic** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here). This is because the algorithm divides the working area in half with each iteration.

```

void quicksort(int list[], int left, int right)
{
    int pivot = partition(list, left, right);
    quicksort(list, left, pivot - 1);
    quicksort(list, pivot + 1, right);
}

```

Taking the previous algorithm forward, above we have a small logic of Quick Sort(we will study this in detail later). Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times(where N is the size of list). Hence time complexity will be $N \cdot \log(N)$. The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

NOTE : In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic.

Types of Notations for Time Complexity

Now we will discuss and understand the various notations used for Time Complexity.

1. **Big Oh** denotes "*fewer than or the same as*" <expression> iterations.
2. **Big Omega** denotes "*more than or the same as*" <expression> iterations.
3. **Big Theta** denotes "*the same as*" <expression> iterations.
4. **Little Oh** denotes "*fewer than*" <expression> iterations.
5. **Little Omega** denotes "*more than*" <expression> iterations.

Understanding Notations of Time Complexity with Example

O(expression) is the set of functions that grow slower than or at the same rate as expression.

Omega(expression) is the set of functions that grow faster than or at the same rate as expression.

Theta(expression) consist of all the functions that lie in both $O(\text{expression})$ and $\Omega(\text{expression})$.

Suppose you've calculated that an algorithm takes $f(n)$ operations, where,

$f(n) = 3*n^2 + 2*n + 4.$ // n^2 means square of n

Since this polynomial grows at the same rate as n^2 , then you could say that the function f lies in the set **Theta(n^2)**. (It also lies in the sets **O(n^2)** and **Omega(n^2)** for the same reason.)

The simplest explanation is, because **Theta** denotes *the same as* the expression. Hence, as $f(n)$ grows by a factor of n^2 , the time complexity can be best represented as **Theta(n^2)**.

Array

C Array is a collection of variables belonging to the same data type. You can store group of data of same data type in an array.

- Array might be belonging to any of the data types
- Array size must be a constant value.
- Always, Contiguous (adjacent) memory locations are used to store array elements in memory.
- It is a best practice to initialize an array to zero or null while declaring, if we don't assign any values to array.

Example for C Arrays:

- `int a[10];` // integer array
- `char b[10];` // character array i.e. string

Types of C arrays:

There are 2 types of C arrays. They are,

1. One-dimensional array
2. Multi-dimensional array
 1. Two dimensional array
 2. Three dimensional array, four dimensional array etc...

1. One dimensional array in C:

- Syntax : data-type arr_name[array_size];

Array declaration	Array initialization	Accessing array
Syntax: data_type arr_name [arr_size];	data_type arr_name [arr_size]=(value1, value2, value3,...);	arr_name[index];
int age [5];	int age[5]={0, 1, 2, 3, 4};	age[0]; /*0 is accessed*/age[1]; /*1 is accessed*/age[2]; /*2 is accessed*/
char str[10];	char str[10]={'H','a','i'}; (or)char str[0] = 'H';char str[1] = 'a'; char str[2] = 'i';	str[0]; /*H is accessed*/str[1]; /*a is accessed*/str[2]; /* i is accessed*/

Example program for one dimensional array in C:

```
#include<stdio.h>int main() {  
  
int i;  
  
int arr[5] = {10,20,30,40,50};  
  
// declaring and Initializing array in C  
  
//To initialize all array elements to 0, use int arr[5]={0};  
  
/* Above array can be initialized as below also  
  
arr[0] = 10;  
  
arr[1] = 20;  
  
arr[2] = 30;  
  
arr[3] = 40;  
  
arr[4] = 50; */
```

```

for (i=0;i<5;i++)

{

// Accessing each variable

printf("value of arr[%d] is %d \n", i, arr[i]);

}

}

```

Output:

```

value of arr[0] is 10
value of arr[1] is 20
value of arr[2] is 30
value of arr[3] is 40
value of arr[4] is 50

```

2. Two dimensional array in C:

- Two dimensional array is nothing but array of array.
- syntax : data_type array_name[num_of_rows][num_of_column]

S.no	Array declaration	Array initialization	Accessing array
1	Syntax: data_type arr_name [num_of_rows][num_of_column];	data_type arr_name[2][2] = {{0,0},{0,1},{1,0},{1,1}};	arr_name[index];
2	Example:int arr[2][2];	int arr[2][2] = {1,2, 3, 4};	arr [0] [0] = 1; arr [0] [1] = 2;arr [1][0] = 3; arr [1] [1] = 4;

Example program for two dimensional array in C:

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

int i,j;

// declaring and Initializing array

int arr[2][2] = {10,20,30,40};

/* Above array can be initialized as below also

arr[0][0] = 10; // Initializing array
arr[0][1] = 20;
arr[1][0] = 30;
arr[1][1] = 40; */

for (i=0;i<2;i++)

{

for (j=0;j<2;j++)

{

// Accessing variables

printf("value of arr[%d] [%d] : %d\n",i,j,arr[i][j]);

}

}

}
```

Output:

value of arr[0] [0] is 10
value of arr[0] [1] is 20
value of arr[1] [0] is 30
value of arr[1] [1] is 40

Stack

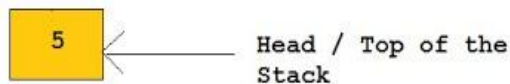
Stack is a specialized data storage structure (Abstract data type). Unlike, arrays access of elements in a stack is restricted. It has two main functions push and pop. Insertion in a stack is done using push function and removal from a stack is done using pop function. Stack allows access to only the last element inserted hence, an item can be inserted or removed from the stack from one end called the top of the stack. It is therefore, also called Last-In-First-Out (LIFO) list. Stack has three properties: capacity stands for the maximum number of elements stack can hold, size stands for the current size of the stack and elements is the array of elements.

The Stack: Last In-First Out (LIFO)

The Empty Stack:
(null)

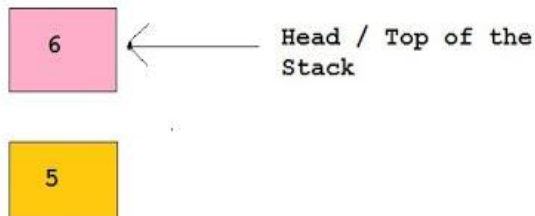
Push 5 onto the
Stack:

The Stack Now Looks Like :



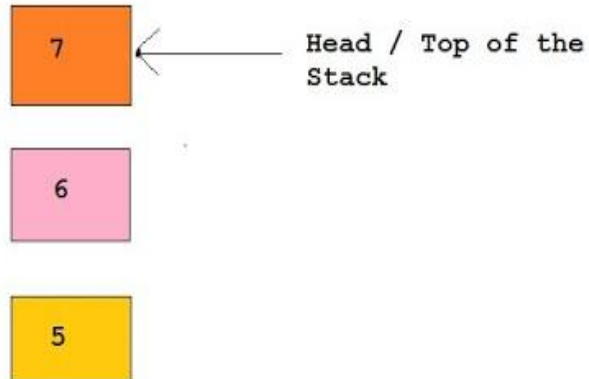
Push 6 onto the
Stack:

The Stack Now Looks Like :



Push 7 onto the Stack:

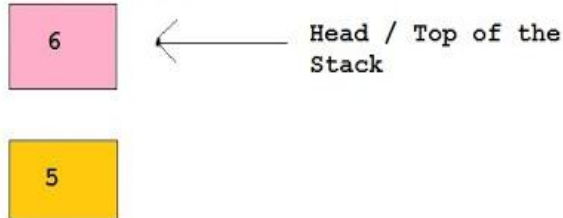
The Stack Now Looks Like :



Pop Whatever is on top of the Stack :

The Stack Now Looks Like :

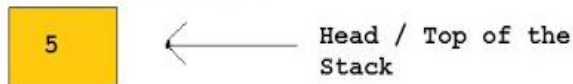
Value Popped Out



Pop Whatever is on top of the Stack :

The Stack Now Looks Like :

Value Popped Out



Algorithm:

Stack structure is defined with fields capacity, size and *elements (pointer to the array of elements).

The stack Methods/Functions – (explained in greater detail in the coding)

1. **createStack function**– This function takes the maximum number of elements (maxElements) the stack can hold as an argument, creates a stack according to it and returns a pointer to the stack. It initializes Stack S using malloc function and its properties.

2. **push function** - This function takes the pointer to the top of the stack S and the item (element) to be inserted as arguments. Check for the emptiness of stack

3. **pop function** - This function takes the pointer to the top of the stack S as an argument.

4. **top function** – This function takes the pointer to the top of the stack S as an argument and returns the topmost element of the stack S.

Properties of stacks:

1. Each function runs in $O(1)$ time.
2. It has two basic implementations

Array-based implementation – It is simple and efficient but the maximum size of the stack is fixed.

Singly Linked List-based implementation – It's complicated but there is no limit on the stack size, it is subjected to the available memory.

Stacks - C Program source code

```
#include<stdio.h>
#include<stdlib.h>
/* Stack has three properties. capacity stands for the maximum number of
elements stack can hold.
Size stands for the current size of the stack and elements is the array of
elements */
typedef struct Stack
{
    int capacity;
    int size;
    int *elements;
}Stack;
/* crateStack function takes argument the maximum number of elements the
stack can hold, creates
a stack according to it and returns a pointer to the stack. */
Stack * createStack(int maxElements)
{
    /* Create a Stack */
    Stack *S;
    S = (Stack *)malloc(sizeof(Stack));
    /* Initialise its properties */
    S->elements = (int *)malloc(sizeof(int)*maxElements);
    S->size = 0;
    S->capacity = maxElements;
    /* Return the pointer */
    return S;
}
void pop(Stack *S)
{
    /* If stack size is zero then it is empty. So we cannot pop */
    if(S->size==0)
```

```

    {
        printf("Stack is Empty\n");
        return;
    }
    /* Removing an element is equivalent to reducing its size by one */
    else
    {
        S->size--;
    }
    return;
}
int top(Stack *S)
{
    if(S->size==0)
    {
        printf("Stack is Empty\n");
        exit(0);
    }
    /* Return the topmost element */
    return S->elements[S->size-1];
}
void push(Stack *S,int element)
{
    /* If the stack is full, we cannot push an element into it as there
is no space for it.*/
    if(S->size == S->capacity)
    {
        printf("Stack is Full\n");
    }
    else
    {
        /* Push an element on the top of it and increase its size by
one*/
        S->elements[S->size++] = element;
    }
    return;
}
int main()
{
    Stack *S = createStack(5);
    push(S,7);
    push(S,5);
    push(S,21);
    push(S,-1);
    printf("Top element is %d\n",top(S));
    pop(S);
    printf("Top element is %d\n",top(S));
    pop(S);
    printf("Top element is %d\n",top(S));
    pop(S);
    printf("Top element is %d\n",top(S));
}


```

Queue




Queue is a specialized data storage structure (Abstract data type). Unlike, arrays access of elements in a Queue is restricted. It has two main operations enqueue and dequeue. Insertion in a queue is done using enqueue function and removal from a queue is done using dequeue function. An item can be inserted at the end ('rear') of the queue and removed from the front ('front') of the queue. It is therefore, also called First-In-First-Out (FIFO) list. Queue has five properties - capacity stands for the maximum number of elements Queue can hold, size stands for the current size of the Queue, elements is the array of elements, front is the index of first element (the index at which we remove the element) and rear is the index of last element (the index at which we insert the element).

Queues : First In - First Out (FIFO)

Initializing an Empty Queue : [null]

Enqueue 5 in the Queue :  (<- head is 5)

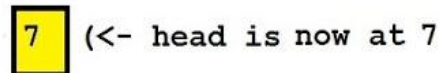
Enqueue 6 in the Queue :  ->  (<-head is still 5)

Enqueue 7 in the Queue :
 ->  ->  (<- head is still 5)

De-Queue Whatever is at the Head of the Queue. 5 will be de-queued, and now the Queue looks like :



Again, De-Queue Whatever is at the Head of the Queue. 6 will now be de-queued and the Queue now looks like :



Again, de-queue whatever is at the head of the queue. 7 will be de-queued and the Queue is now empty. So we are back to [null]

Algorithm:

Queue structure is defined with fields capacity, size, *elements (pointer to the array of elements), front and rear.

The Queue methods/Functions explained in Details in the coding

1. createQueue function– This function takes the maximum number of elements (maxElements) the Queue can hold as an argument, creates a Queue according to it and returns a pointer to the Queue.

2. enqueue function - This function takes the pointer to the top of the queue Q and the item (element) to be inserted as arguments. Check for the emptiness of queue

3. dequeue function - This function takes the pointer to the top of the stack S as an argument and will then dequeue an element.

4. front function – This function takes the pointer to the top of the queue Q as an argument and returns the front element of the queue Q.

Properties:

1. Each function runs in $O(1)$ time.
2. It has two basic implementations

Array-based implementation – It's simple and efficient but the maximum size of the queue is fixed.

Singly Linked List-based implementation – It's complicated but there is no limit on the queue size, it is subjected to the available memory.

Queues - C Program source code

```
#include<stdio.h>
#include<stdlib.h>
/*Queue has five properties. capacity stands for the maximum number of
elements Queue can hold.
    Size stands for the current size of the Queue and elements is the array of
elements. front is the
    index of first element (the index at which we remove the element) and rear
is the index of last element
    (the index at which we insert the element) */
typedef struct Queue
{
    int capacity;
    int size;
    int front;
    int rear;
    int *elements;
}Queue;
/* crateQueue function takes argument the maximum number of elements the
Queue can hold, creates
    a Queue according to it and returns a pointer to the Queue. */
Queue * createQueue(int maxElements)
{
    /* Create a Queue */
    Queue *Q;
    Q = (Queue *)malloc(sizeof(Queue));
    /* Initialise its properties */
    Q->elements = (int *)malloc(sizeof(int)*maxElements);
    Q->size = 0;
    Q->capacity = maxElements;
    Q->front = 0;
    Q->rear = -1;
    /* Return the pointer */
    return Q;
}
void Dequeue(Queue *Q)
{
    /* If Queue size is zero then it is empty. So we cannot pop */
    if(Q->size==0)
    {
        printf("Queue is Empty\n");
    }
}
```

```

        return;
    }
    /* Removing an element is equivalent to incrementing index of front
by one */
    else
    {
        Q->size--;
        Q->front++;
        /* As we fill elements in circular fashion */
        if(Q->front==Q->capacity)
        {
            Q->front=0;
        }
    }
    return;
}
int front(Queue *Q)
{
    if(Q->size==0)
    {
        printf("Queue is Empty\n");
        exit(0);
    }
    /* Return the element which is at the front*/
    return Q->elements[Q->front];
}
void Enqueue(Queue *Q,int element)
{
    /* If the Queue is full, we cannot push an element into it as there
is no space for it.*/
    if(Q->size == Q->capacity)
    {
        printf("Queue is Full\n");
    }
    else
    {
        Q->size++;
        Q->rear = Q->rear + 1;
        /* As we fill the queue in circular fashion */
        if(Q->rear == Q->capacity)
        {
            Q->rear = 0;
        }
        /* Insert the element in its rear side */
        Q->elements[Q->rear] = element;
    }
    return;
}
int main()
{
    Queue *Q = createQueue(5);
    Enqueue(Q,1);
    Enqueue(Q,2);
    Enqueue(Q,3);
    Enqueue(Q,4);
    printf("Front element is %d\n",front(Q));
    Enqueue(Q,5);
}

```

```

    Dequeue(Q) ;
    Enqueue(Q,6) ;
    printf("Front element is %d\n",front(Q)) ;
}

```

Linked Lists

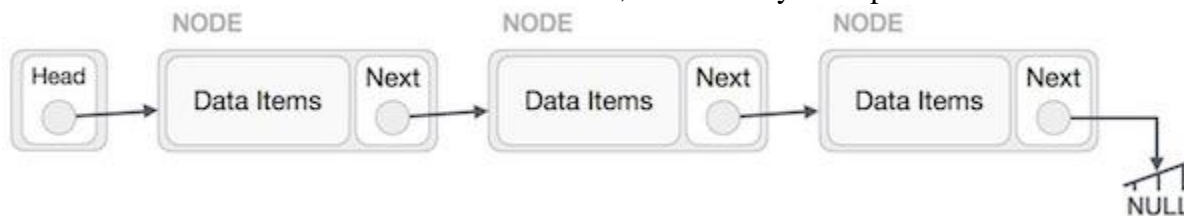
A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **LinkedList** – A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



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

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

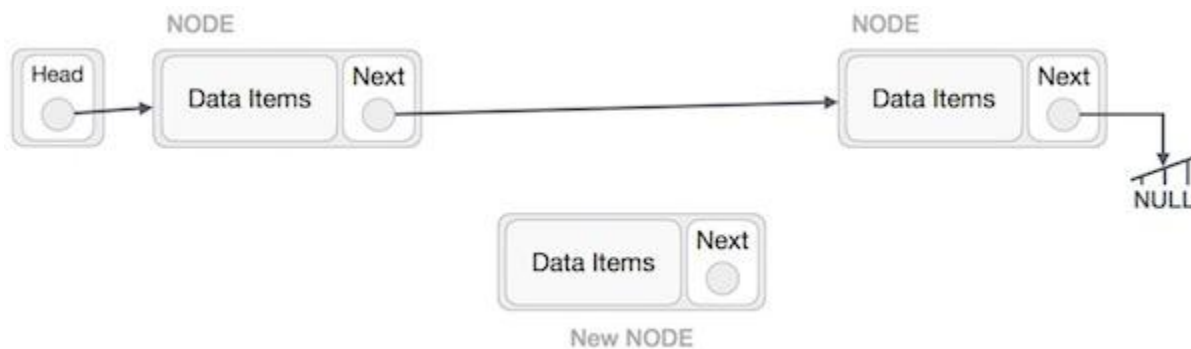
Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.
- **Delete** – Deletes an element using the given key.

Insertion Operation

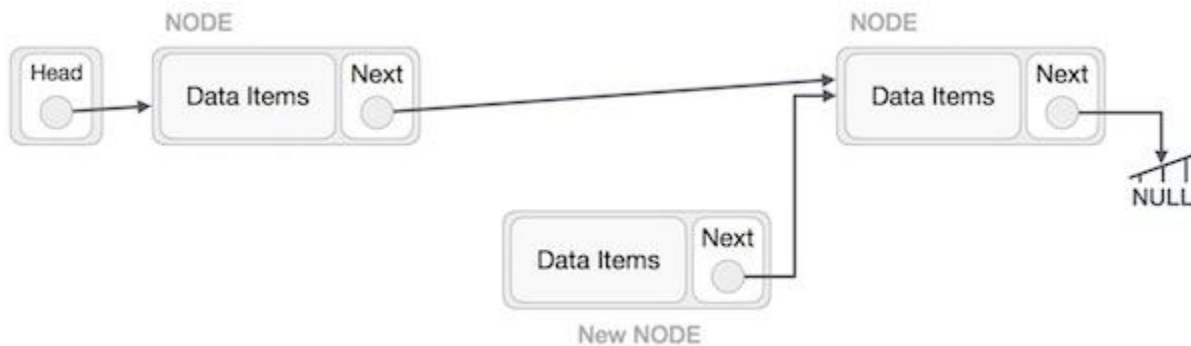
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C –

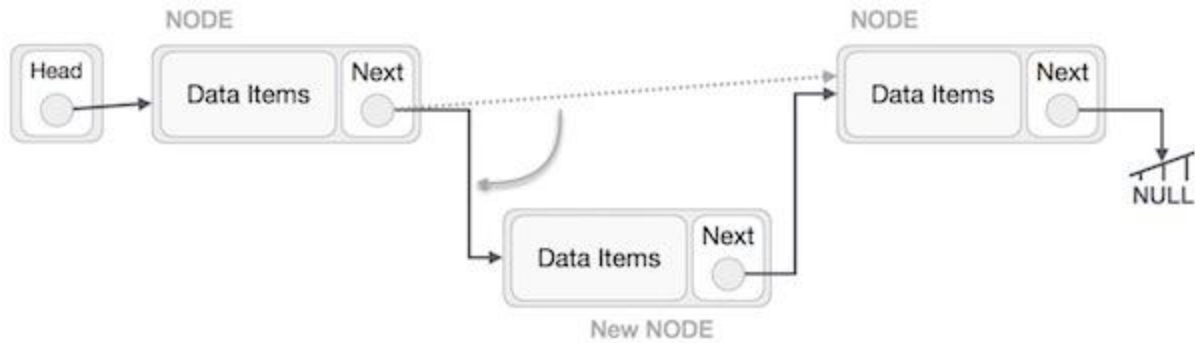
```
NewNode.next -> RightNode;
```

It should look like this –



Now, the next node at the left should point to the new node.

```
LeftNode.next -> NewNode;
```

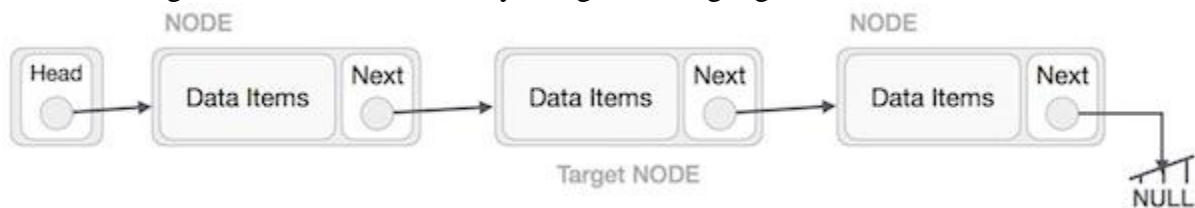
This will put the new node in the middle of the two. The new list should look like this –



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

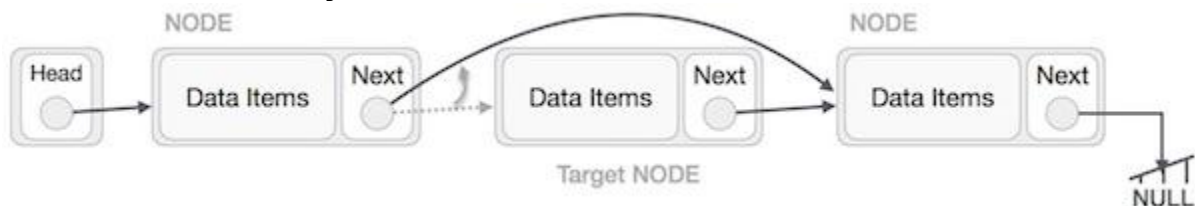
Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



The left (previous) node of the target node now should point to the next node of the target node –

```
LeftNode.next -> TargetNode.next;
```

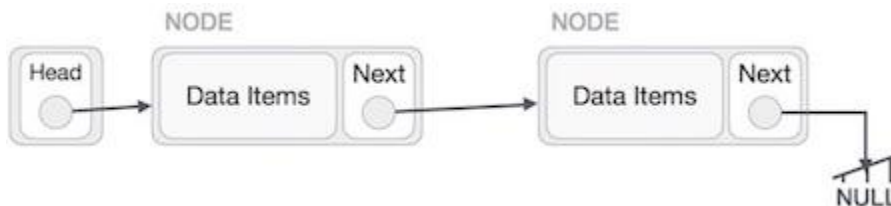


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

```
TargetNode.next -> NULL;
```

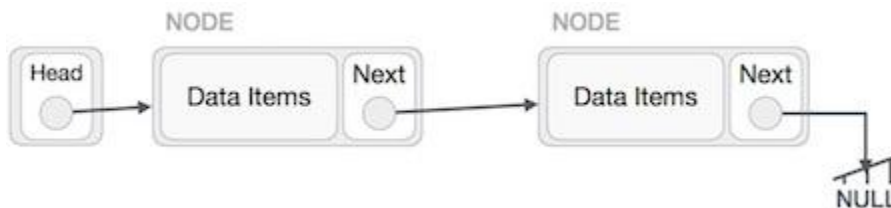


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

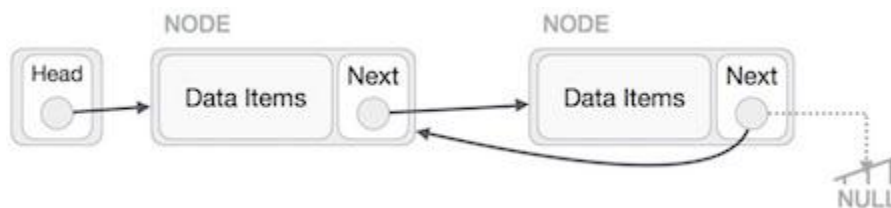


Reverse Operation

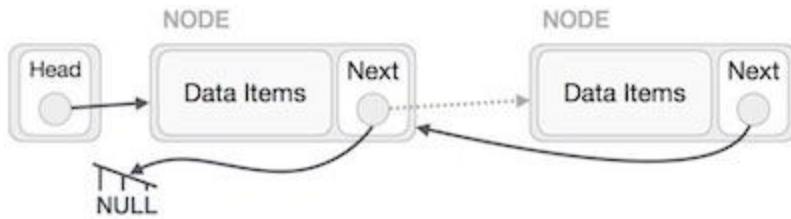
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



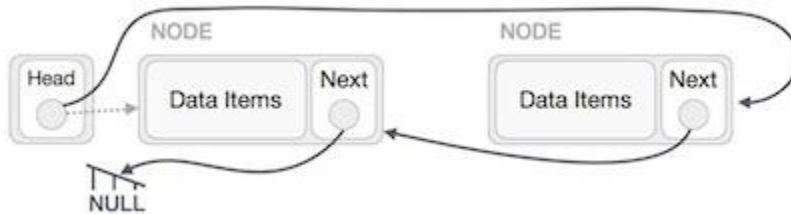
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node –



We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



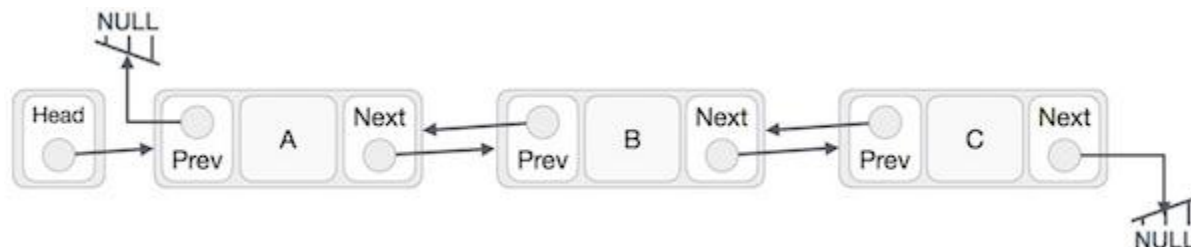
We'll make the head node point to the new first node by using the temp node.

Doubly Linked List

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **Prev** – Each link of a linked list contains a link to the previous link called Prev.
- **LinkedList** – A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation



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

- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.

- The last link carries a link as null to mark the end of the list.

Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Insert Last** – Adds an element at the end of the list.
- **Delete Last** – Deletes an element from the end of the list.
- **Insert After** – Adds an element after an item of the list.
- **Delete** – Deletes an element from the list using the key.
- **Display forward** – Displays the complete list in a forward manner.
- **Display backward** – Displays the complete list in a backward manner.

Insertion Operation

Following code demonstrates the insertion operation at the beginning of a doubly linked list.

Example

```
//insert link at the first location
void insertFirst(int key, int data) {

    //create a link
    struct node *link = (struct node*) malloc(sizeof(struct node));
    link->key = key;
    link->data = data;

    if(isEmpty()) {
        //make it the last link
        last = link;
    } else {
        //update first prev link
        head->prev = link;
    }

    //point it to old first link
    link->next = head;

    //point first to new first link
    head = link;
}
```

Deletion Operation

Following code demonstrates the deletion operation at the beginning of a doubly linked list.

Example

```
//delete first item
struct node* deleteFirst() {

    //save reference to first link
    struct node *tempLink = head;
```

```

//if only one link
if(head->next == NULL) {
    last = NULL;
} else {
    head->next->prev = NULL;
}

head = head->next;

//return the deleted link
return tempLink;
}

```

Insertion at the End of an Operation

Following code demonstrates the insertion operation at the last position of a doubly linked list.

Example

```

//insert link at the last location
void insertLast(int key, int data) {

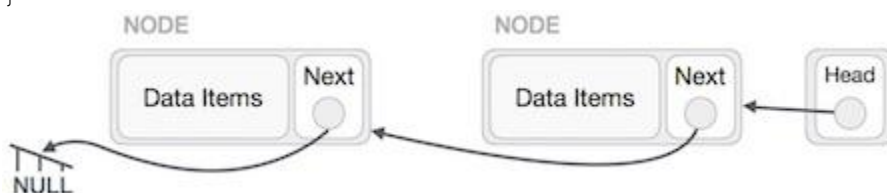
    //create a link
    struct node *link = (struct node*) malloc(sizeof(struct node));
    link->key = key;
    link->data = data;

    if(isEmpty()) {
        //make it the last link
        last = link;
    } else {
        //make link a new last link
        last->next = link;

        //mark old last node as prev of new link
        link->prev = last;
    }

    //point last to new last node
    last = link;
}

```

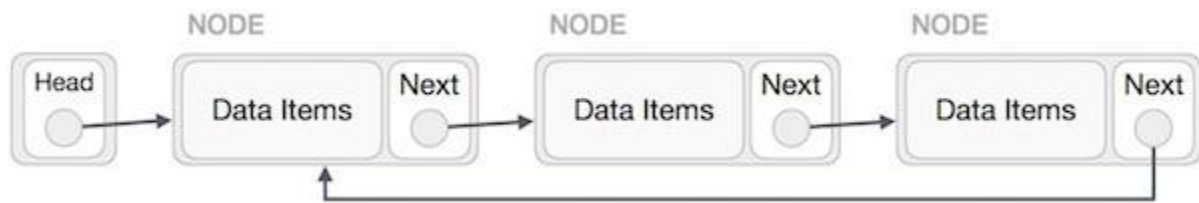


Circular Linked List

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

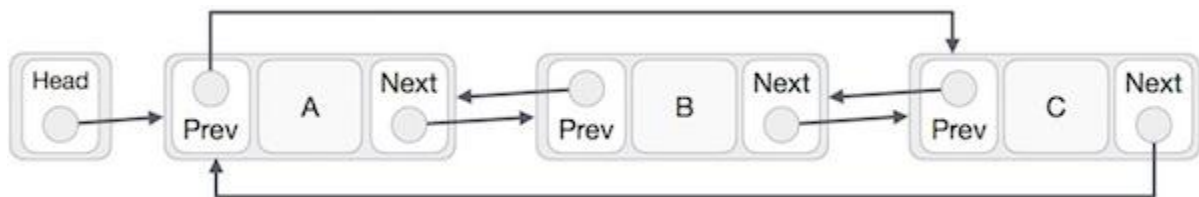
Singly Linked List as Circular

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



Doubly Linked List as Circular

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



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

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

Basic Operations

Following are the important operations supported by a circular list.

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

Insertion Operation

Following code demonstrates the insertion operation in a circular linked list based on single linked list.

Example

```
//insert link at the first location
void insertFirst(int key, int data) {
    //create a link
    struct node *link = (struct node*) malloc(sizeof(struct node));
    link->key = key;
```

```

link->data= data;

if (isEmpty()) {
    head = link;
    head->next = head;
} else {
    //point it to old first node
    link->next = head;

    //point first to new first node
    head = link;
}
}

```

Deletion Operation

Following code demonstrates the deletion operation in a circular linked list based on single linked list.

```

//delete first item
struct node * deleteFirst() {
    //save reference to first link
    struct node *tempLink = head;

    if(head->next == head) {
        head = NULL;
        return tempLink;
    }

    //mark next to first link as first
    head = head->next;

    //return the deleted link
    return tempLink;
}

```

Display List Operation

Following code demonstrates the display list operation in a circular linked list.

```

//display the list
void printList() {
    struct node *ptr = head;
    printf("\n[ ");

    //start from the beginning
    if(head != NULL) {
        while(ptr->next != ptr) {
            printf("(%d,%d) ",ptr->key,ptr->data);
            ptr = ptr->next;
        }
    }

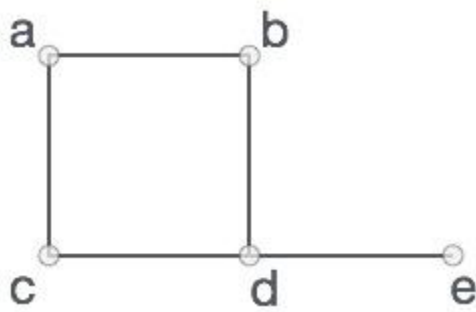
    printf(" ]");
}

```

Graph Data Structure

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets (V, E) , where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

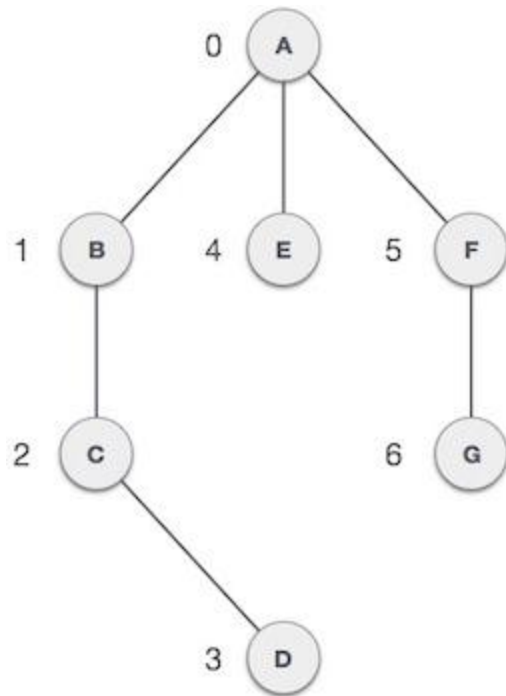
$$V = \{a, b, c, d, e\}$$

$$E = \{ab, ac, bd, cd, de\}$$

Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

- **Vertex** – Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** – Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** – Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



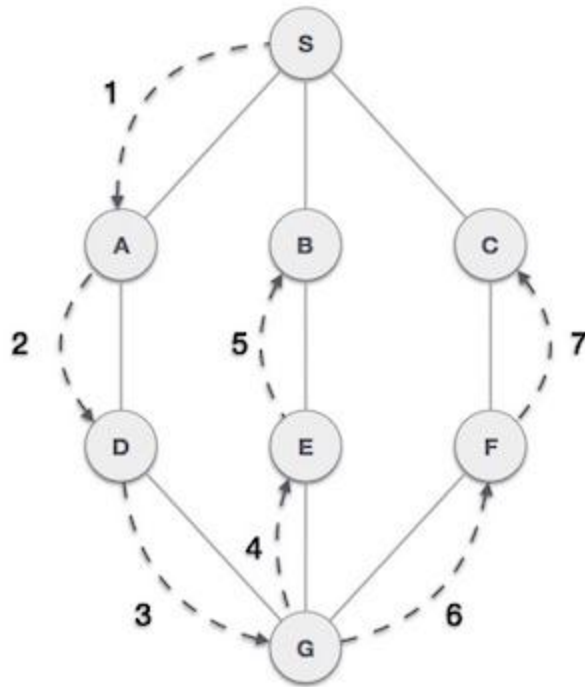
Basic Operations

Following are basic primary operations of a Graph –

- **Add Vertex** – Adds a vertex to the graph.
- **Add Edge** – Adds an edge between the two vertices of the graph.
- **Display Vertex** – Displays a vertex of the graph.

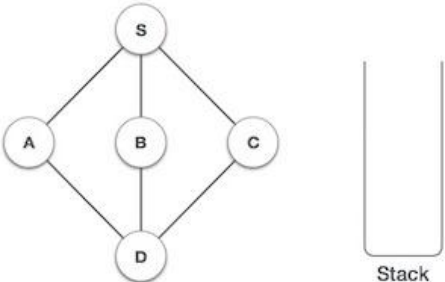
Graph Traversal

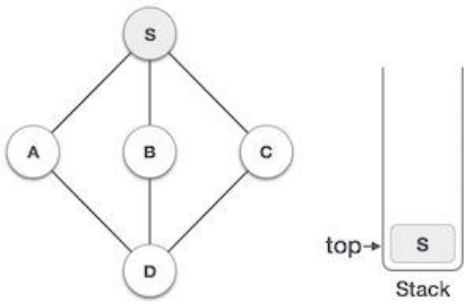
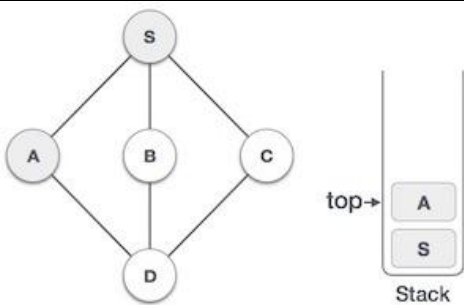
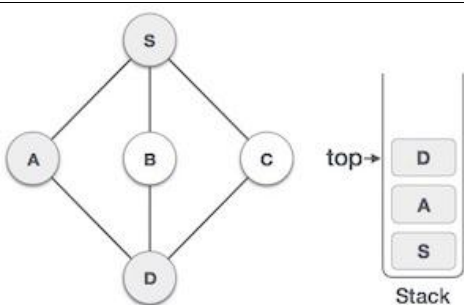
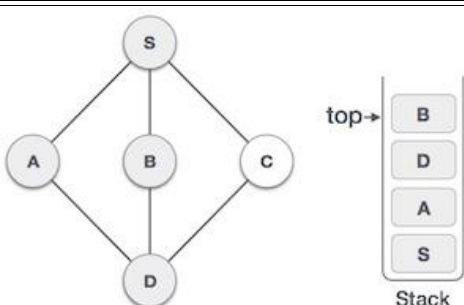
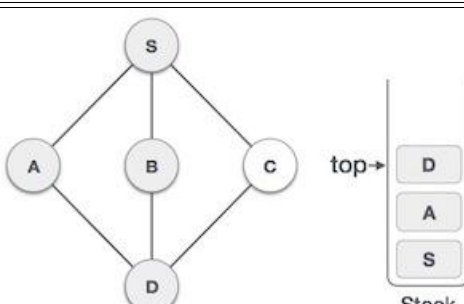
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

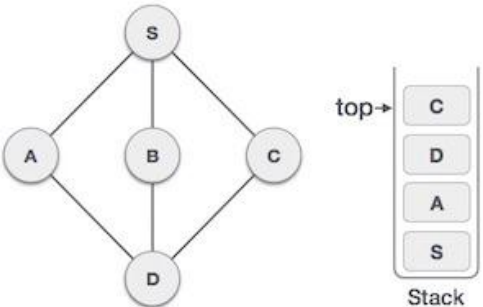


As in the example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

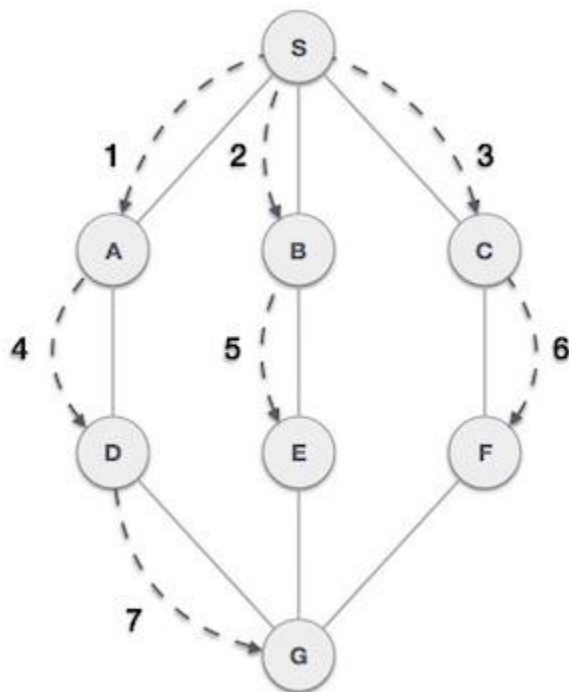
Step	Traversal	Description
1.		Initialize the stack.

2.		<p>Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.</p>
3.		<p>Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.</p>
4.		<p>Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.</p>
5.		<p>We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.</p>
6.		<p>We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.</p>

7.		<p>Only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.</p>
----	---	---

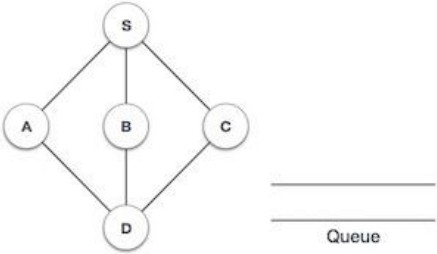
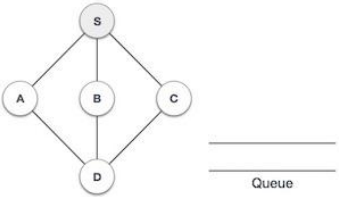
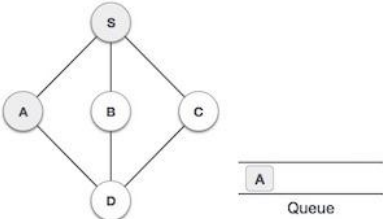
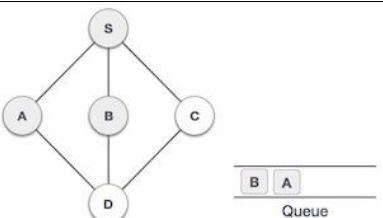
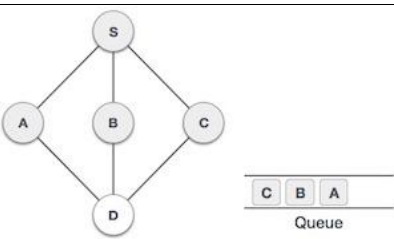
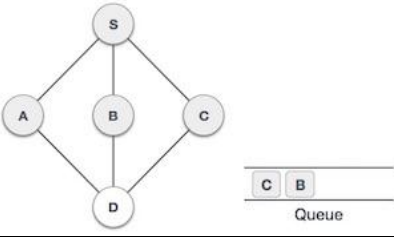
As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

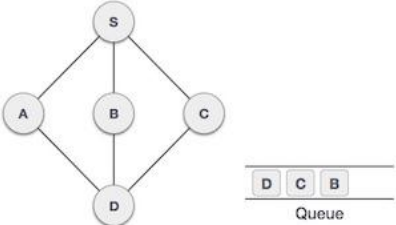
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1.		Initialize the queue.
2.		We start from visiting S (starting node), and mark it as visited.
3.		We then see an unvisited adjacent node from S . In this example, we have three nodes but alphabetically we choose A , mark it as visited and enqueue it.
4.		Next, the unvisited adjacent node from S is B . We mark it as visited and enqueue it.
5.		Next, the unvisited adjacent node from S is C . We mark it as visited and enqueue it.
6.		Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A .

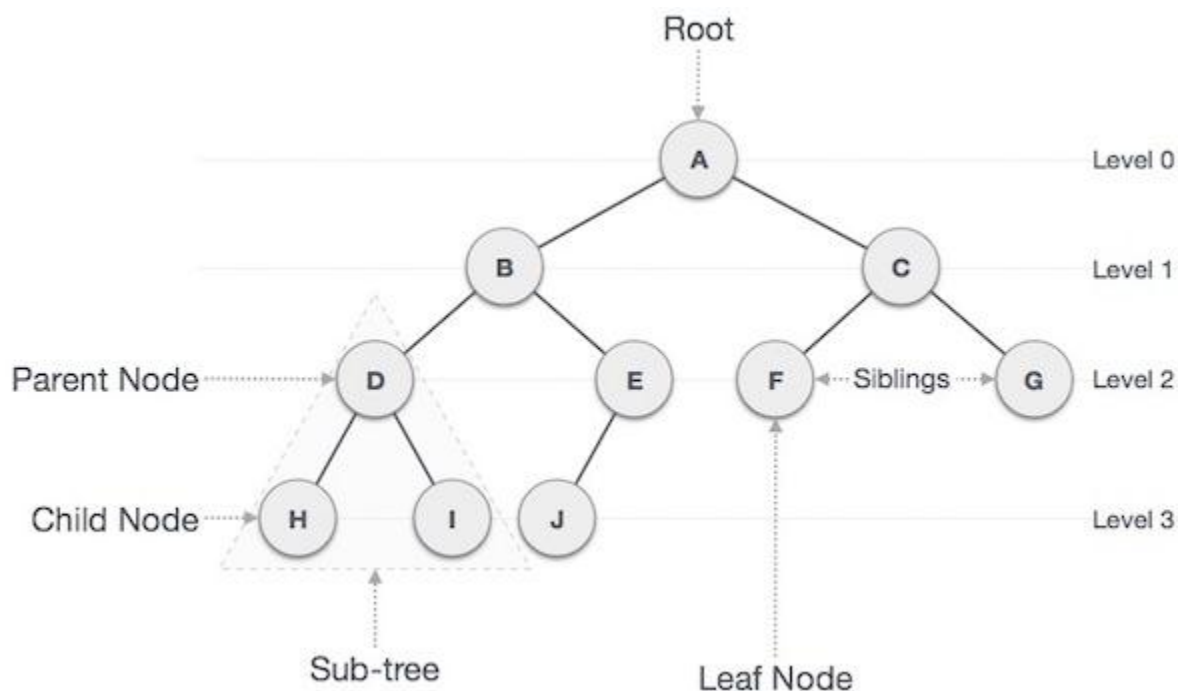
7.		From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.
----	---	--

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Tree

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special datastructure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



Important Terms

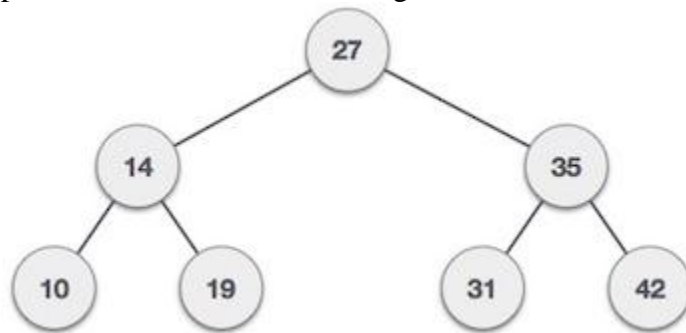
Following are the important terms with respect to tree.

- **Path** – Path refers to the sequence of nodes along the edges of a tree.

- **Root** – The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** – Any node except the root node has one edge upward to a node called parent.
- **Child** – The node below a given node connected by its edge downward is called its child node.
- **Leaf** – The node which does not have any child node is called the leaf node.
- **Subtree** – Subtree represents the descendants of a node.
- **Visiting** – Visiting refers to checking the value of a node when control is on the node.
- **Traversing** – Traversing means passing through nodes in a specific order.
- **Levels** – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** – Key represents a value of a node based on which a search operation is to be carried out for a node.

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



We're going to implement tree using node object and connecting them through references.

Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```

struct node {
    int data;
    struct node *leftChild;
    struct node *rightChild;
};
  
```

In a tree, all nodes share common construct.

BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following –

- **Insert** – Inserts an element in a tree/create a tree.
- **Search** – Searches an element in a tree.

- **Preorder Traversal** – Traverses a tree in a pre-order manner.
- **Inorder Traversal** – Traverses a tree in an in-order manner.
- **Postorder Traversal** – Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```

If root is NULL
    then create root node
return

If root exists then
    compare the data with node.data

    while until insertion position is located

        If data is greater than node.data
            goto right subtree
        else
            goto left subtree

    endwhile

    insert data

end If

```

Implementation

The implementation of insert function should look like this –

```

void insert(int data) {
    struct node *tempNode = (struct node*) malloc(sizeof(struct node));
    struct node *current;
    struct node *parent;

    tempNode->data = data;
    tempNode->leftChild = NULL;
    tempNode->rightChild = NULL;

    //if tree is empty, create root node
    if(root == NULL) {
        root = tempNode;
    } else {
        current = root;
        parent = NULL;

```



```

while(1) {
    parent = current;

    //go to left of the tree
    if(data < parent->data) {
        current = current->leftChild;

        //insert to the left
        if(current == NULL) {
            parent->leftChild = tempNode;
            return;
        }
    }

    //go to right of the tree
    else {
        current = current->rightChild;

        //insert to the right
        if(current == NULL) {
            parent->rightChild = tempNode;
            return;
        }
    }
}
}
}
}
}

```

Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```

If root.data is equal to search.data
    return root
else
    while data not found

        If data is greater than node.data
            goto right subtree
        else
            goto left subtree

        If data found
            return node

    endwhile

    return data not found

end if

```

The implementation of this algorithm should look like this.

```
struct node* search(int data) {
    struct node *current = root;
    printf("Visiting elements: ");

    while(current->data != data) {
        if(current != NULL)
            printf("%d ", current->data);

        //go to left tree

        if(current->data > data) {
            current = current->leftChild;
        }
        //else go to right tree
        else {
            current = current->rightChild;
        }

        //not found
        if(current == NULL) {
            return NULL;
        }

        return current;
    }
}
```

Tree Traversal

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

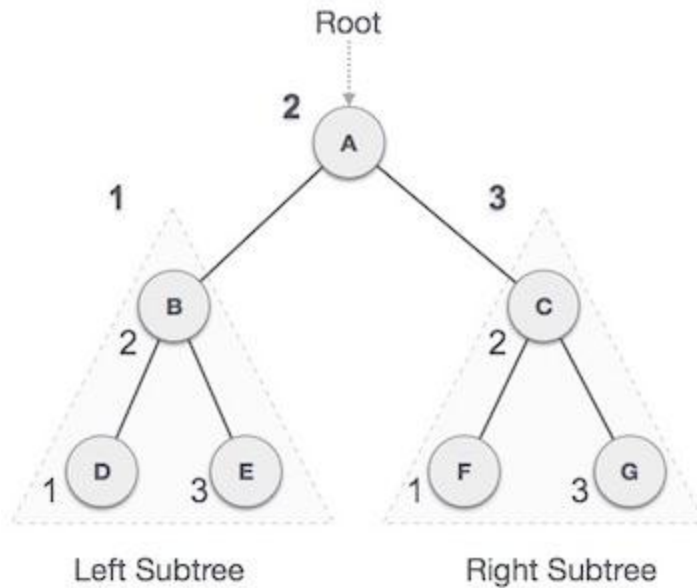
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

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

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be –

$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$

Algorithm

Until all nodes are traversed –

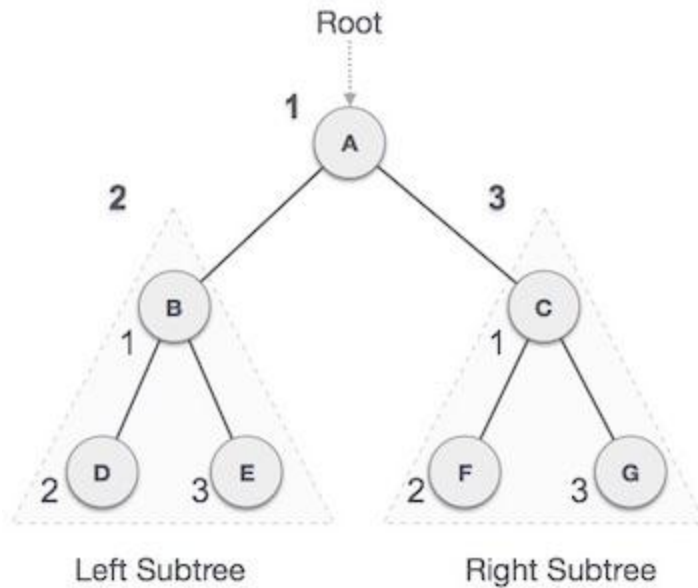
Step 1 – Recursively traverse left subtree.

Step 2 – Visit root node.

Step 3 – Recursively traverse right subtree.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Algorithm

Until all nodes are traversed –

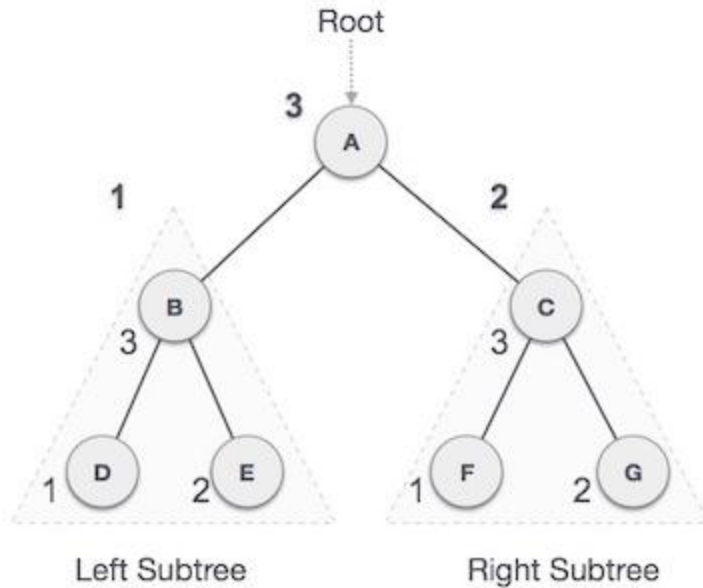
Step 1 – Visit root node.

Step 2 – Recursively traverse left subtree.

Step 3 – Recursively traverse right subtree.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following pre-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$

Algorithm

Until all nodes are traversed –

- Step 1** – Recursively traverse left subtree.
- Step 2** – Recursively traverse right subtree.
- Step 3** – Visit root node.

Types of trees

We have three basic types as listed below

- 1) Binary Search Tree
- 2) AVL Trees
- 3) Spanning Tree
- 4) Heap Data Structures

Binary Search Tree

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.

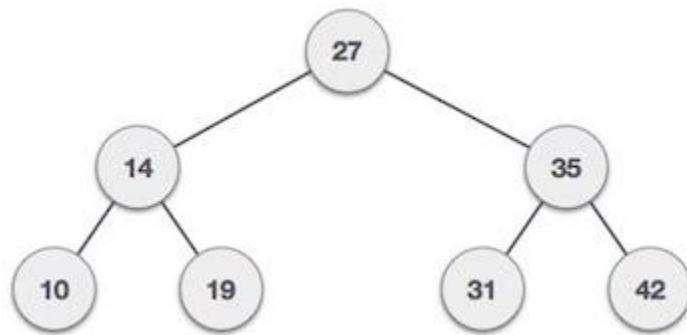
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as –

$$\text{left_subtree (keys)} \leq \text{node (key)} \leq \text{right_subtree (keys)}$$

Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST –



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations

Following are the basic operations of a tree –

- **Search** – Searches an element in a tree.
- **Insert** – Inserts an element in a tree.
- **Pre-order Traversal** – Traverses a tree in a pre-order manner.
- **In-order Traversal** – Traverses a tree in an in-order manner.
- **Post-order Traversal** – Traverses a tree in a post-order manner.

Node

Define a node having some data, references to its left and right child nodes.

```
struct node {  
    int data;  
    struct node *leftChild;  
    struct node *rightChild;  
};
```

Search Operation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
struct node* search(int data){
    struct node *current = root;
    printf("Visiting elements: ");

    while(current->data != data){

        if(current != NULL) {
            printf("%d ", current->data);

            //go to left tree
            if(current->data > data){
                current = current->leftChild;
            } //else go to right tree
            else {
                current = current->rightChild;
            }

            //not found
            if(current == NULL){
                return NULL;
            }
        }
    }
    return current;
}
```

Insert Operation

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```
void insert(int data) {
    struct node *tempNode = (struct node*) malloc(sizeof(struct node));
    struct node *current;
    struct node *parent;

    tempNode->data = data;
    tempNode->leftChild = NULL;
    tempNode->rightChild = NULL;

    //if tree is empty
    if(root == NULL) {
        root = tempNode;
    } else {
        current = root;
        parent = NULL;
    }
```

```

while(1) {
    parent = current;

    //go to left of the tree
    if(data < parent->data) {
        current = current->leftChild;
        //insert to the left

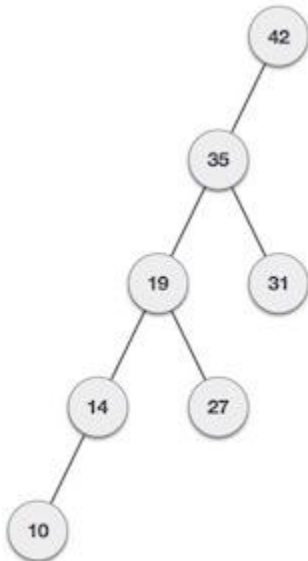
        if(current == NULL) {
            parent->leftChild = tempNode;
            return;
        }
    } //go to right of the tree
    else {
        current = current->rightChild;

        //insert to the right
        if(current == NULL) {
            parent->rightChild = tempNode;
            return;
        }
    }
}
}
}
}
}

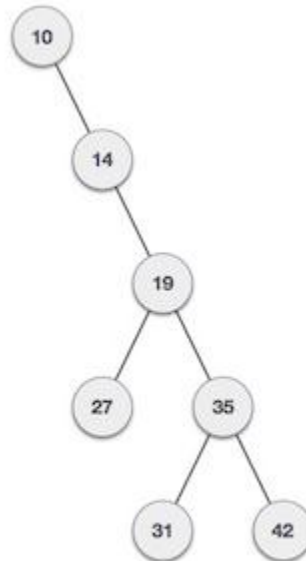
```

AVL Trees

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this –



If input 'appears' non-increasing manner

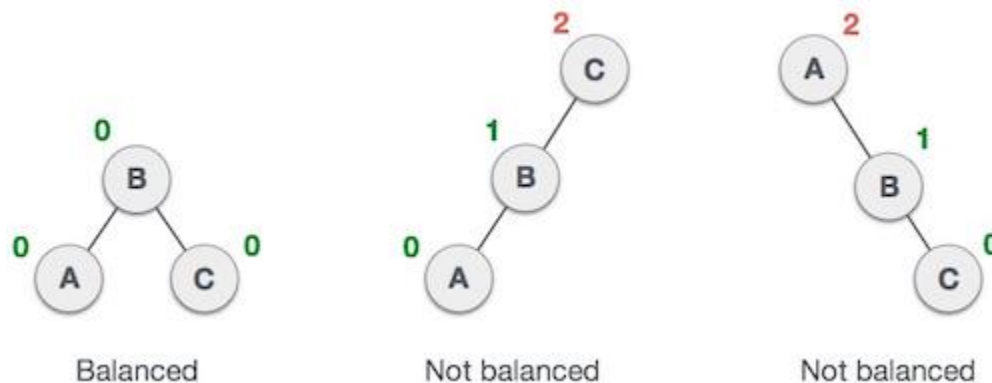


If input 'appears' in non-decreasing manner

It is observed that BST's worst-case performance is closest to linear search algorithms, that is $O(n)$. In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson, Velski & Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced and the next two trees are not balanced –



In the second tree, the left subtree of C has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of A has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

BalanceFactor = $\text{height}(\text{left-sutree}) - \text{height}(\text{right-sutree})$

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

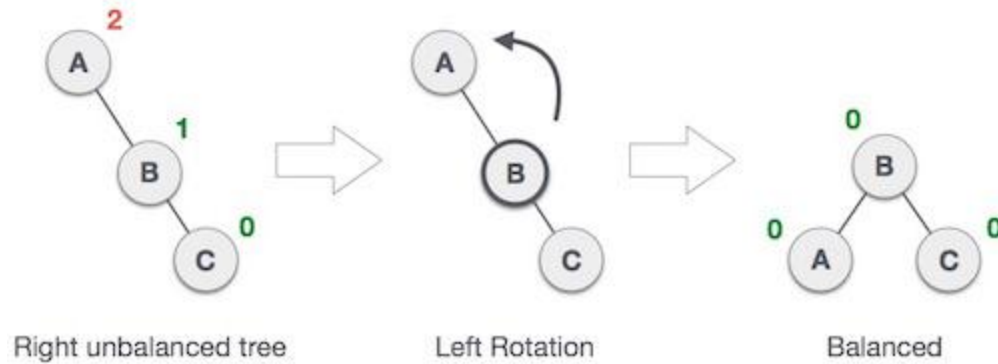
To balance itself, an AVL tree may perform the following four kinds of rotations –

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

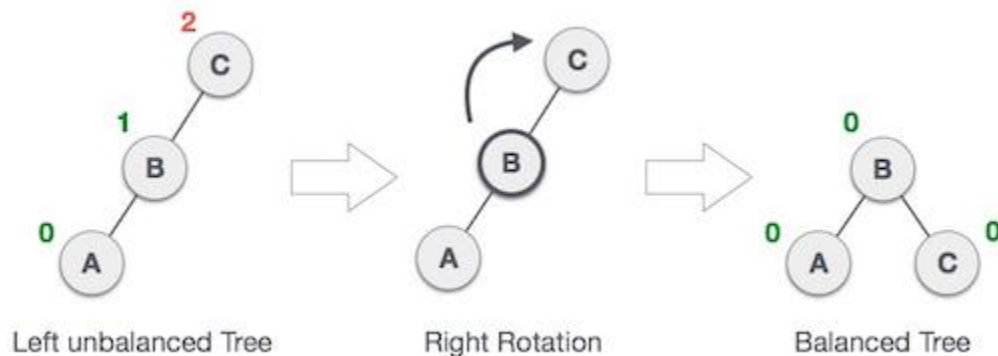
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

Right Rotation

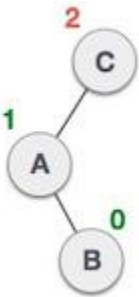
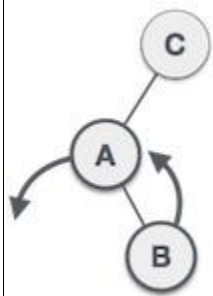
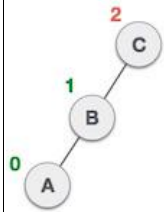
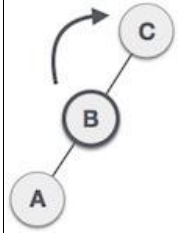
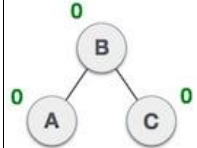
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

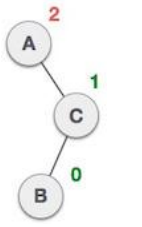
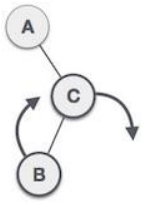
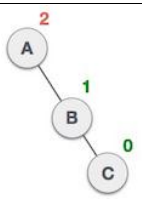
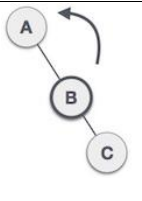
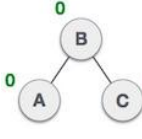
Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

State	Action
	<p>A node has been inserted into the right subtree of the left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.</p>
	<p>We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.</p>
	<p>Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.</p>
	<p>We shall now right-rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree.</p>
	<p>The tree is now balanced.</p>

Right-Left Rotation

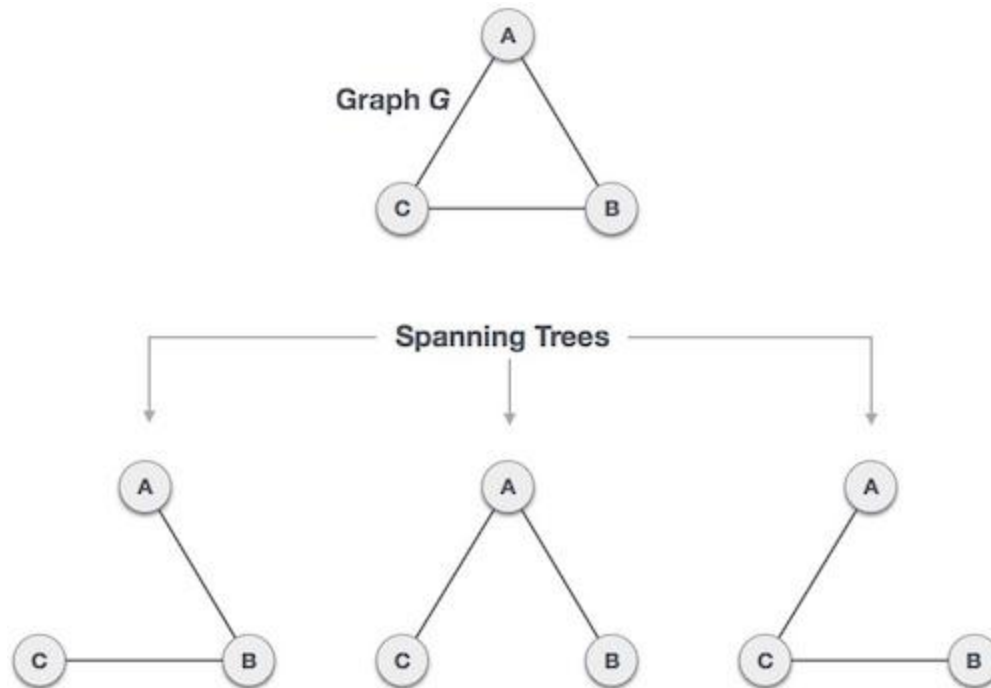
The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes A , an unbalanced node with balance factor 2.
	First, we perform the right rotation along C node, making C the right subtree of its own left subtree B . Now, B becomes the right subtree of A .
	Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making B the new root node of the subtree. A becomes the left subtree of its right subtree B .
	The tree is now balanced.

Spanning Tree

A spanning tree is a subset of Graph G , which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. In the above addressed example, $3^{3-2} = 3$ spanning trees are possible.

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G –

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Mathematical Properties of Spanning Tree

- Spanning tree has $n-1$ edges, where n is the number of nodes (vertices).
- From a complete graph, by removing maximum $e - n + 1$ edges, we can construct a spanning tree.
- A complete graph can have maximum n^{n-2} number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are –

- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Heap Data Structures

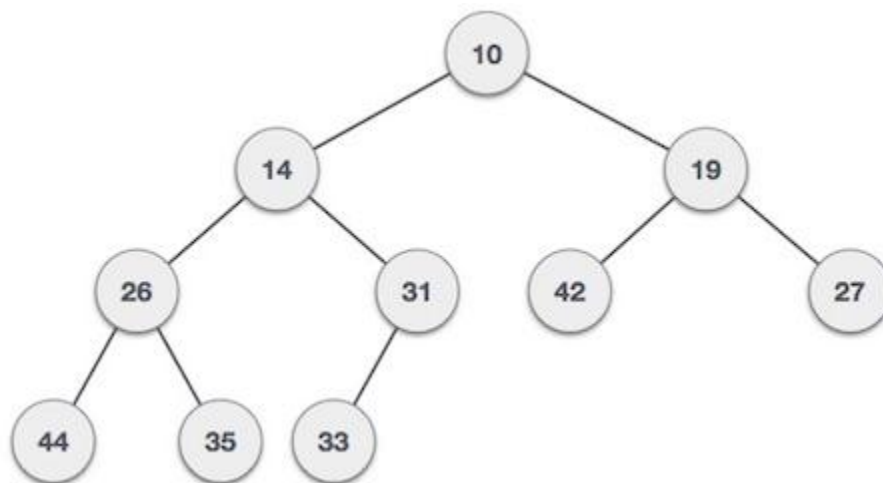
Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If α has child node β then –

$$\text{key}(\alpha) \geq \text{key}(\beta)$$

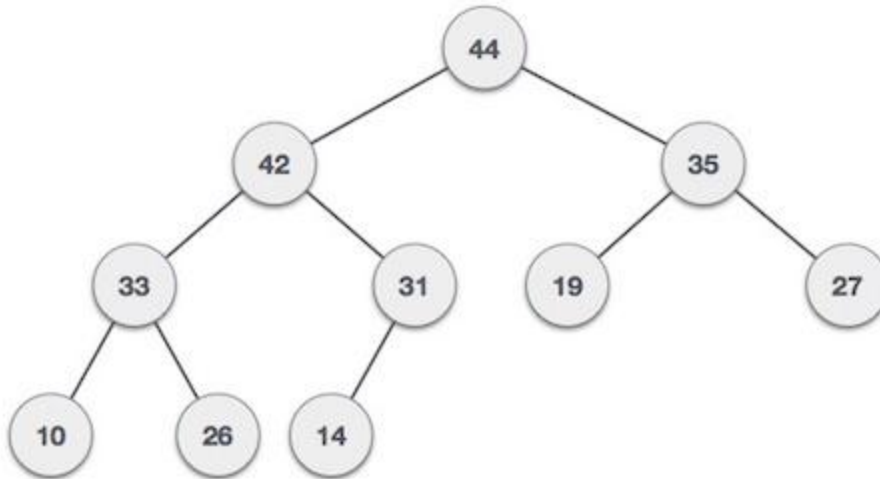
As the value of parent is greater than that of child, this property generates **Max Heap**. Based on this criteria, a heap can be of two types –

For Input → 35 33 42 10 14 19 27 44 26 31

Min-Heap – Where the value of the root node is less than or equal to either of its children.



Max-Heap – Where the value of the root node is greater than or equal to either of its children.



Both trees are constructed using the same input and order of arrival.

Max Heap Construction Algorithm

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

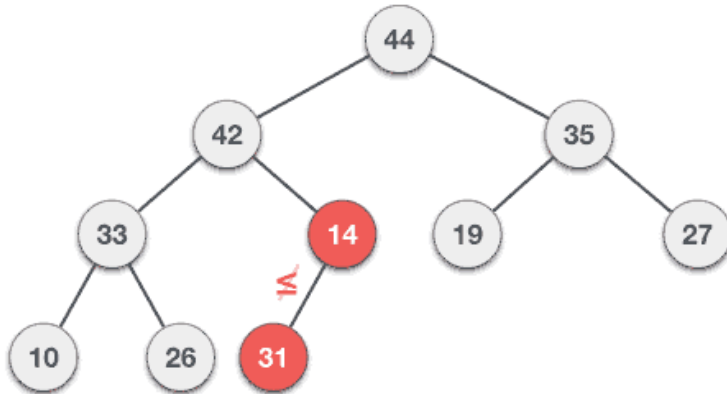
We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

- Step 1** - Create a new node at the end of heap.
- Step 2** - Assign new value to the node.
- Step 3** - Compare the value of this child node with its parent.
- Step 4** - If value of parent is less than child, then swap them.
- Step 5** - Repeat step 3 & 4 until Heap property holds.

Note – In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

Let's understand Max Heap construction by an animated illustration. We consider the same input sample that we used earlier.

Input 35 33 42 10 14 19 27 44 26 31



Max Heap Deletion Algorithm

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

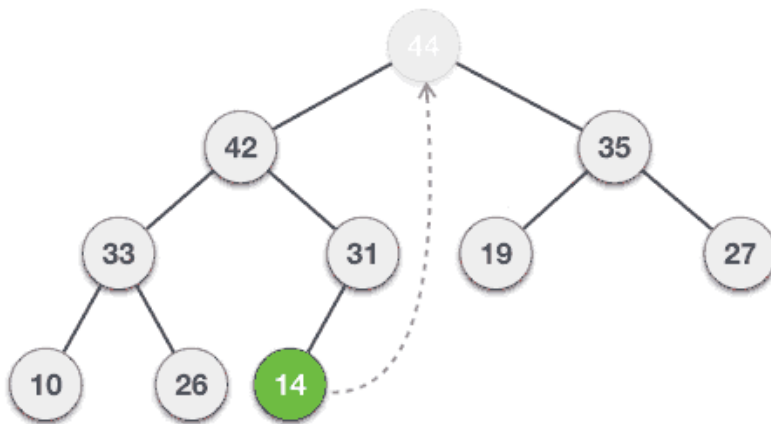
Step 1 - Remove root node.

Step 2 - Move the last element of last level to root.

Step 3 - Compare the value of this child node with its parent.

Step 4 - If value of parent is less than child, then swap them.

Step 5 - Repeat step 3 & 4 until Heap property holds.



Searching and Sorting

Objective:

- Learn how to search arrays.
- Learn how to sort an array.

1. Searching

A common problem encountered in programming is the need to search an array in order to find the location of the desired value. Suppose you are looking for a student with ID 995203 in a list of ID's, then you search for this ID in the list from the beginning till you find it or reach the end of the array and the ID is not there.

The searching algorithms we will use are the linear search and the binary search.

Linear search:

This is a general search strategy. Here, we search for the element from the beginning of the array. If we find the target element in the array, we stop searching and return the position of the element in the array; otherwise we continue till the end of the array. If we reach the end of the array and did not find the target, the search returns -1 as an indication that it is not in the array.

The following figure is an example of Linear search:

Suppose we wish to write a method that takes in a one-dimensional array and some value x, and returns either the position (i.e., "index") of x in the array, or -1 if x does not appear in the array.

One option would be to use a **linear search**. In a linear search, we compare x (which we call the "key") with each element in the array list, starting at one end and progressing to the other.

Graphically, we can imagine the following comparisons being made:

Key	List
3	6 4 1 9 7 3 2 8
3	6 4 1 9 7 3 2 8
3	6 4 1 9 7 3 2 8
3	6 4 1 9 7 3 2 8
3	6 4 1 9 7 3 2 8
3	6 4 1 9 7 3 2 8

This is the code of linear search:

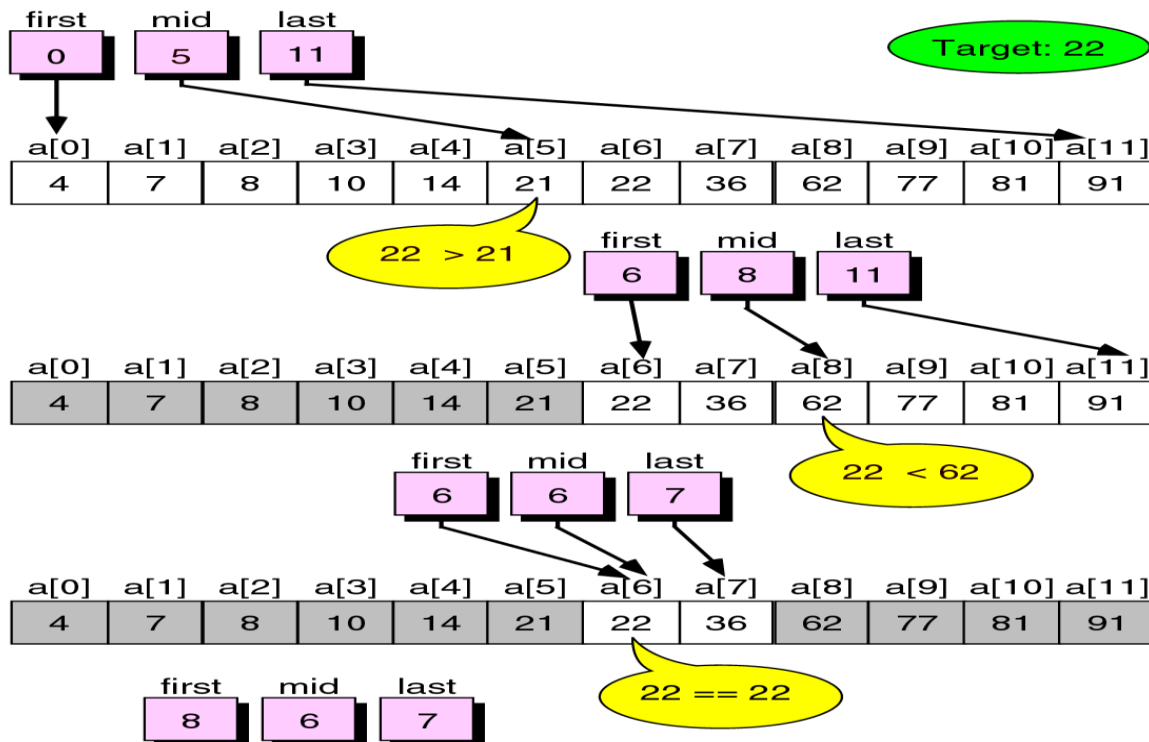
```
int linear_search(int a[], int n, int key){  
    int i;  
    for(i = 0; i < n; i++){  
        if(key == a[i])  
            return i ;  
    }  
    return -1;  
}
```

Binary search:

If the array we are searching is sorted and the sorting order is known, then we can use a better strategy than linear search to look for the target element. The binary search makes use of the fact that the elements are ordered and does not scan the whole array. The steps of binary search, for an array that is sorted in increasing order, are:

1. if the middle element m is equal to the target, then return its position and we are done.
2. else:
 - a. if the target $<$ middle element, then search the first half of the array and repeat step 1.
 - b. if the target $>$ middle, then search second half of the array and repeat step 1.
 - c. if the array is exhausted, then the target is not in the array and the search returns -1.

The following figure is an example of binary search:



In the implementation, we use two variables (`first` and `last`) that indicate the start and the end of the sub-array being searched. Through `first` and `last` we can get the middle element and specify the boundary of the array for the next pass if we do not find the target in the current pass.

This is code for binary search:

```
int binary_search(int x[], int first, int last, int key) {
    int middle;

    if(last<first)
        return -1; // key not in array

    middle=(first+last)/2;

    if(key==x[middle])
        return middle;

    else if (key<x[middle])
        return binary_search(x, first,middle-1, key);
}
```

```
else  
  
    return binary_search(x, middle+1, last, key);  
  
}
```

2. Sorting

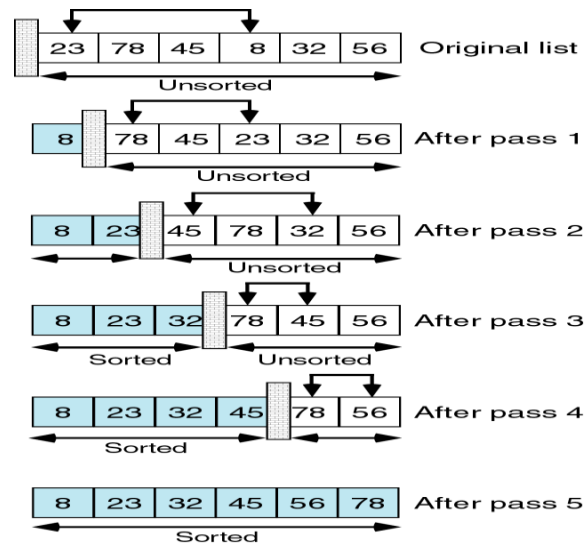
Another interesting problem in programming is to sort the elements of an array in increasing or decreasing order. The use of sorted arrays is very obvious. Having sorted records makes it easier to locate a particular element. Also if you want the output of these records (for example students ID's along with their grades) to be displayed on screen sorted, then using a sort strategy becomes necessary.

Sorting in general refers to various methods of arranging or ordering things based on criteria's (numerical, chronological, alphabetical, hierarchical etc.). In Computer Science, due to obvious reasons, Sorting (of data) is of immense importance and is one of the most extensively researched subjects. It is one of the most fundamental algorithmic problems. So much so that it is also fundamental to many other fundamental algorithmic problems such as search algorithms, merge algorithms etc. It is estimated that around 25% of all CPU cycles are used to sort data. There are many approaches to sorting data and each has its own merits and demerits. This article discusses some of the common sorting algorithms.

In the discussion, we will discuss one sorting algorithm: Selection Sort.

Selection Sort:

Selection sort is a simple straightforward algorithm to sort an array of numbers and it works as follows. First, we find the minimum element and store it in $A[0]$. Next, we find the minimum of the remaining elements and store it in $A[1]$. We continue this way until the second largest element is stored in $A[\text{size} - 1]$. The following figure explains the process of selection sort:



This is the code for selection sort:

```
void selection_sort(int x[], int size){
    int k,j,minpos,temp;
    for (k=0; k < size - 1; k++){
        minpos = k; // initialize location of min value
        // go over the elements to find location of minimum value
        for(j = k+1; j < size; j++){
            if(x[j] < x[minpos])
                minpos = j;
        }
        // bring minimum value which is at minpos at index k
        temp = x[minpos];
        x[minpos] = x[k];
        x[k] = temp;
    }
}
```

Other Sorting techniques.....

Bubble Sort

Bubble Sort is probably one of the oldest, most easiest, straight-forward, inefficient sorting algorithms. It is the algorithm introduced as a sorting routine in most introductory courses on Algorithms. Bubble Sort works by comparing each element of the list with the element next to it and swapping them if required. With each pass, the largest of the list is "bubbled" to the end of the list whereas the smaller values sink to the bottom. It is similar to selection sort although not as straight forward. Instead of "selecting" maximum values, they are bubbled to a part of the list. An implementation in C.

```
void BubbleSort(int a[], int array_size)
{
    int i, j, temp;
    for (i = 0; i < (array_size - 1); ++i)
    {
        for (j = 0; j < array_size - 1 - i; ++j )
        {
            if (a[j] > a[j+1])

                {
                    temp = a[j+1];
                    a[j+1] = a[j];
                    a[j] = temp;
                }
        }
    }
}
```

A single, complete "bubble step" is the step in which a maximum element is bubbled to its correct position. This is handled by the inner for loop.

```
for (j = 0; j < array_size - 1 - i; ++j )
{
    if (a[j] > a[j+1])
    {
        temp = a[j+1];
        a[j+1] = a[j];
        a[j] = temp;
    }
}
```

Examine the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting)

8 6 10 3 1 2 5 4	} pass 0
6 8 3 1 2 5 4 10	} pass 1
6 3 1 2 5 4 8 10	} pass 2
3 1 2 5 4 6 8 10	} pass 3
1 2 3 4 5 6 8 10	} pass 4
1 2 3 4 5 6 8 10	} pass 5
1 2 3 4 5 6 8 10	} pass 6
1 2 3 4 5 6 8 10	} pass 7

The above tabulated clearly depicts how each bubble sort works. Note that each pass results in one number being bubbled to the end of the list.

Selection Sort

The idea of Selection Sort is rather simple. It basically determines the minimum (or maximum) of the list and swaps it with the element at the index where its supposed to be. The process is repeated such that the nth minimum (or maximum) element is swapped with the element at the n-1th index of the list. The below is an implementation of the algorithm in C.

```
void SelectionSort(int a[], int array_size)
{
    int i;
    for (i = 0; i < array_size - 1; ++i)
    {
        int j, min, temp;
        min = i;
        for (j = i+1; j < array_size; ++j)
        {
            if (a[j] < a[min])
                min = j;
        }

        temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}
```

Consider the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting)

8 6 10 3 1 2 5 4 } pass 0
1 6 10 3 8 2 5 4 } pass 1
1 2 10 3 8 6 5 4 } pass 2
1 2 3 10 8 6 5 4 } pass 3
1 2 3 4 8 6 5 10 } pass 4
1 2 3 4 5 6 8 10 } pass 5
1 2 3 4 5 6 8 10 } pass 6
1 2 3 4 5 6 8 10 } pass 7

At pass 0, the list is unordered. Following that is pass 1, in which the minimum element 1 is selected and swapped with the element 8, at the lowest index 0. In pass 2, however, only the sublist is considered, excluding the element 1. So element 2, is swapped with element 6, in the 2nd lowest index position. This process continues till the sub list is narrowed down to just one element at the highest index (which is its right position).

Insertion Sort

The Insertion Sort algorithm is a commonly used algorithm. Even if you haven't been a programmer or a student of computer science, you may have used this algorithm. Try recalling how you sort a deck of cards. You start from the beginning, traverse through the cards and as you find cards misplaced by precedence you remove them and insert them back into the right position. Eventually what you have is a sorted deck of cards. The same idea is applied in the Insertion Sort algorithm. The following is an implementation in C.

```
void insertionSort(int a[], int array_size)
{
    int i, j, index;
    for (i = 1; i < array_size; ++i)
    {
        index = a[i];
        for (j = i; j > 0 && a[j-1] > index; j--)
            a[j] = a[j-1];

        a[j] = index;
    }
}
```


Examine the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting)

8 6 10 3 1 2 5 4 } pass 0
6 8 10 3 1 2 5 4 } pass 1
6 8 10 3 1 2 5 4 } pass 2
3 6 8 10 1 2 5 4 } pass 3
1 3 6 8 10 2 5 4 } pass 4
1 2 3 6 8 10 5 4 } pass 5
1 2 3 5 6 8 10 4 } pass 6
1 2 3 4 5 6 8 10 } pass 7

The pass 0 is only to show the state of the unsorted array before it is given to the loop for sorting. Now try out the deck-of-cards-sorting algorithm with this list and see if it matches with the tabulated data. For example, you start from 8 and the next card you see is 6. Hence you remove 6 from its current position and "insert" it back to the top. That constituted pass 1. Repeat the same process and you'll do the same thing for 3 which is inserted at the top. Observe in pass 5 that 2 is moved from position 5 to position 1 since its $< (6,8,10)$ but > 1 . As you carry on till you reach the end of the list you'll find that the list has been sorted. It didn't take a course to tell you how to sort a deck of cards, did it; you prolly figured it out on your own. Amazed at the computer scientist in you ? ;)

Heap Sort

Heap sort algorithm, as the name suggests, is based on the concept of heaps. It begins by constructing a special type of binary tree, called heap, out of the set of data which is to be sorted. Note:

- A Heap by definition is a special type of binary tree in which each node is greater than any of its descendants. It is a complete binary tree.
- A semi-heap is a binary tree in which all the nodes except the root possess the heap property.
- If N be the number of a node, then its left child is $2*N$ and the right child $2*N+1$.

The root node of a Heap, by definition, is the maximum of all the elements in the set of data, constituting the binary tree. Hence the sorting process basically consists of extracting the root node and reheapng the remaining set of elements to obtain the next largest element till there are no more elements left to heap. Elementary implementations usually employ two arrays, one for the heap and the other to store the sorted data. But it is possible to use the same array to heap the unordered list and compile the sorted list. This is usually done by swapping the root of the heap with the end of the array and then excluding that element from any subsequent reheapng.

Significance of a semi-heap - A Semi-Heap as mentioned above is a Heap except that the root does not possess the property of a heap node. This type of a heap is significant in the discussion of Heap Sorting, since after each "Heaping" of the set of data, the root is extracted and replaced by an element from the list. This leaves us with a Semi-Heap. Reheapng a Semi-Heap is particularly easy since all other nodes have already been heaped and only the root node has to be

shifted downwards to its right position. The following C function takes care of reheapifying a set of data or a part of it.

```
void downHeap(int a[], int root, int bottom)
{
    int maxchild, temp, child;
    while (root*2 < bottom)
    {
        child = root * 2 + 1;
        if (child == bottom)
        {
            maxchild = child;
        }
        else
        {
            if (a[child] > a[child + 1])
                maxchild = child;
            else
                maxchild = child + 1;
        }

        if (a[root] < a[maxchild])
        {
            temp = a[root];
            a[root] = a[maxchild];
            a[maxchild] = temp;
        }
        else return;

        root = maxchild;
    }
}
```

In the above function, both root and bottom are indices into the array. Note that, theoretically speaking, we generally express the indices of the nodes starting from 1 through size of the array. But in C, we know that array indexing begins at 0; and so the left child is

```
child = root * 2 + 1
/* so, for eg., if root = 0, child = 1 (not 0) */
```

In the function, what basically happens is that, starting from root each loop performs a check for the heap property of root and does whatever necessary to make it conform to it. If it does already conform to it, the loop breaks and the function returns to caller. Note that the function assumes that the tree constituted by the root and all its descendants is a Semi-Heap.

Now that we have a downheaper, what we need is the actual sorting routine.

```
void heapsort(int a[], int array_size)
{
    int i;
    for (i = (array_size/2 - 1); i >= 0; --i)
    {
        downHeap(a, i, array_size-1);
    }

    for (i = array_size-1; i >= 0; --i)
    {
        int temp;
        temp = a[i];
        a[i] = a[0];
        a[0] = temp;
        downHeap(a, 0, i-1);
    }
}
```

Note that, before the actual sorting of data takes place, the list is heaped in the for loop starting from the mid element (which is the parent of the right most leaf of the tree) of the list.

```
for (i = (array_size/2 - 1); i >= 0; --i)
{
    downHeap(a, i, array_size-1);
}
```

Following this is the loop which actually performs the extraction of the root and creating the sorted list. Notice the swapping of the ith element with the root followed by a reheaping of the list.

```
for (i = array_size-1; i >= 0; --i)
{
    int temp;
    temp = a[i];
    a[i] = a[0];
    a[0] = temp;
    downHeap(a, 0, i-1);
}
```

The following are some snapshots of the array during the sorting process. The unodered list -

8 6 10 3 1 2 5 4

After the initial heaping done by the first for loop.

10 6 8 4 1 2 5 3

Second loop which extracts root and reheaps.

```
8 6 5 4 1 2 3 10 } pass 1
6 4 5 3 1 2 8 10 } pass 2
5 4 2 3 1 6 8 10 } pass 3
4 3 2 1 5 6 8 10 } pass 4
3 1 2 4 5 6 8 10 } pass 5
2 1 3 4 5 6 8 10 } pass 6
1 2 3 4 5 6 8 10 } pass 7
1 2 3 4 5 6 8 10 } pass 8
```

Heap sort is one of the preferred sorting algorithms when the number of data items is large. Its efficiency in general is considered to be poorer than quick sort and merge sort.

Lab Work:

Question 1:

Write a program that will generate an array of 10 integer numbers from 0 to 30 using the function **rand()** present in **stdlib.h** library. The function **rand()** will return a pseudo random number from 0 to **RAND_MAX** which is the maximum value in the **int** type. In order to limit your number from 0 to **n**, make the returned **value** to be **value % (n+1)**. By using **rand()** only, your program will generate the same sequence during each execution. To avoid this problem insert the following statement at the beginning of your main function:

```
srand(time(NULL)) ;
```

Ask the user to enter a **target value** (between 0 and 30) to check if it is present in the array by using linear search. If present your program should display its position, otherwise print a message saying that the value is not in the array.

Question 2:

The binary search function shown above is recursive; rewrite the same function using a loop instead of recursion. To test your iterative binary search function repeat **question 1** by using binary search instead of linear search.

Note: you should sort your array before searching

Question 3:

Write a logical function **sorted_inc** that receives an array of int values and **n** representing the number of values. The function will return **1** if the array values are sorted in increasing order, **0** otherwise.

Write a similar function **sorted_dec** to check if the array values are sorted in decreasing order or not.

Write another version of selection_sort; **selection_sort_dec** so that it sorts the array in decreasing order.

Write a program that will generate an array x1 of 20 random integer values from 1 to 100. Use similar procedure as in question 1, but now the lower part is not 0.

Print the array on the screen.

Check if it is by chance sorted in increasing or decreasing order by using the functions sorted_inc and sorted_dec. If the array is not sorted (this is what is expected), sort the array by using the original and modified selection sort functions i.e. increasing and decreasing. Print your array after each sorting.

CHAPTER 7: SUB PROGRAMS

Introduction to Subprograms

Subprograms are named program instruction blocks that can take parameters and be invoked. Programming has two types of subprograms called *procedures* and *functions*. Generally, you use a procedure to perform an action and a function to compute a value. The only difference between a PROCEDURE and a FUNCTION is the former does not return a value, while the later does.

Subprograms have a declarative part, an executable part, and an optional exception-handling part. The declarative part contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These objects are local and cease to exist when you exit the subprogram. The executable part contains statements that assign values, control execution, and manipulate data. The exception-handling part contains exception handlers, which deal with exceptions raised during execution.

Fundamental of Subprograms

Subprograms include the following characteristics:

- Each subprogram has a single entry point
- There is only one subprogram execution at any given time
- Control always returns to the caller when the subprogram execution terminates.

Basic Definitions

- A **subprogram definition** describes the interface to and the actions of the subprogram abstraction
- A **subprogram call** is the explicit request that the subprogram be executed.
- A subprogram is **active** if after having been called, it has begun execution but has not yet completed that execution.
- Two fundamental kinds of subprograms are procedures and functions.
- A **subprogram header**, which is the first line of the definition, serves several purposes. It specifies that the following syntactic unit is a subprogram definition. And it provides the name for the subprogram. And, it may optionally specify list of parameters.
- The **parameter profile** of a subprogram is the number, order and types of its formal parameters.
- The **protocol** of a subprogram is its parameter profile plus, if it is a function, its return types
- Subprograms declarations are common in C programs where they are called **prototypes**.

Parameters

- Subprograms usually describe computations.
- There are 2 ways that a subprogram can gain access to the data that is to process: through direct access to nonlocal variables or through parameter passing.
- Data passed through parameters are accessed through names that are local to the subprogram. Parameter passing is more flexible than direct access to nonlocal variables
- The parameters in the subprogram header are called **formal parameters**
- Subprograms call statements must include the name of the subprogram and a list of parameters to be bound to the formal parameters of the subprogram. These parameters are called **actual parameters**

- The binding of actual parameters to formal parameters – is done by simple position: the first actual parameter is bound to the first formal parameter and so forth. Such parameters are called **positional parameters**.
- When lists are long, it is easy for the program writer to make mistakes in the order of parameters in the list – one solution to this problem is to provide **keyword parameters**, in which the name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter.

Procedures and Functions

- Procedures are collections of statements that define parameterized computations. These computations are enacted by single call statements - procedures define new statements.
- Functions structurally resemble procedures but are semantically modeled on mathematical functions.
- Functions are called by appearances of their names in expressions, along with the required actual parameters. The value produced by a function's execution is returned to the calling code, replacing the call itself.

Design Issues for Subprograms

- Subprograms are complex structures in programming languages
- An overloaded subprogram is one that has the same name as another subprogram in the same referencing environment.
- A generic subprogram is one whose computation can be done on data of different types with different calls

Local Referencing Environment

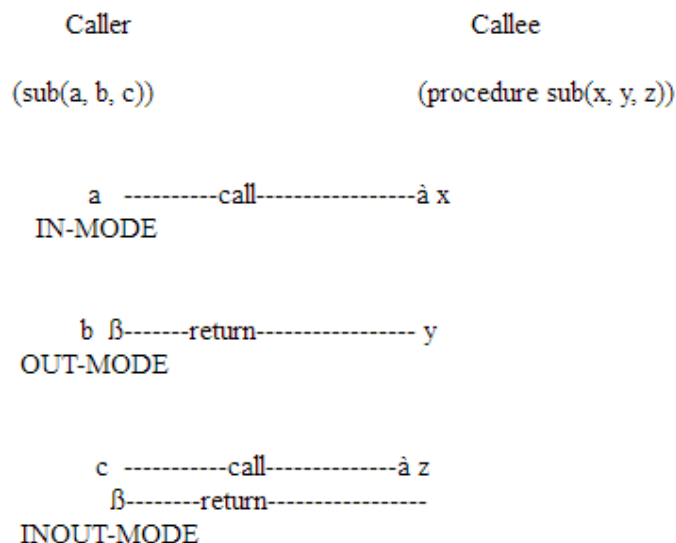
- Subprograms are generally allowed to define their own variables, thereby defining local referencing environments. Variables that are defined inside subprograms are called **local variables** because access to them is usually restricted to the subprogram in which they are defined.
- If local variables are stack dynamic, they are bound to storage when the subprogram begins execution and unbound from storage when that execution terminates. An advantage of this is flexibility.
- It is important that recursive subprograms have stack dynamic local variables.
- Another advantage is that some of the storage for local variables of all subprograms can be shared
- Main disadvantages of stack dynamic local variables are:
 - Cost of time required to allocate, initialize and de-allocate for each activation
 - Accesses of stack dynamic local variables must be indirect, where accesses to static can be direct
 - Stack dynamic local variables, the subprograms cannot retain data values of local variables between calls.
- The primary advantage of static local variables is that they are very efficient because of no indirection

Parameter-Passing Methods

- Parameter-passing methods are the ways in which parameters are transmitted to and / or from called programs
- Formal parameters are characterized by one of three semantics models:
 - They can receive data from the corresponding actual parameter
 - They can transmit data to the actual parameter, OR
 - They can do both.
- These three semantics models are called **in mode**, **out mode** and **inout mode**, respectively.
- There are 2 conceptual models of how data transfers take place in parameter transmission: either an actual value is physically moved (to the caller, to the callee, or both ways), or an access path is transmitted.
- Most commonly the access path is a simple pointer.

Pass by Value

- When a parameter is **passed by value**, the value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable in the subprogram – this implements in-mode semantics.



- Pass-by-value is implemented by actual data transfer
- The main disadvantage of pass by value is that if physical moves are done, the additional storage is required for the formal parameter, either in the called subprogram or in some area outside both the caller and the called subprogram.

Pass by Result

- **Pass by result** is an implementation model for out-mode parameters
- When a parameter is passed by result, no value is transmitted to the subprogram
- One problem with the pass by result is that there can be an actual parameter collision such as the one created with the call

`sub(p1, p2)`

- **sub** here assumes 2 formal parameters with different names, then whichever of the 2 is assigned to their corresponding actual parameter last becomes the value of p1.
- The order in which the actual parameters are assigned determines their value.
- Another problem with pass by result is that the implementer may be able to choose between 2 different times to evaluate the addresses of the actual parameters: at the time of the call or at the time of the return.

Pass by Value Result

- **Pass by value result** is an implementation model for in-out mode parameters in which actual values are moved.
- It is a combination of pass by value and pass by result.

Pass by Reference

- **Pass by reference** is a second implementation of in-out mode parameters
- Rather than transmitting data values back and forth, as in pass by value result, the pass by reference method transmits an access path, usually just an address, to the called subprogram. This provides the access path to the cell storing the actual parameter.
- The advantage of pass by reference is efficiency in both time and space.
- The disadvantages are:
 - Access to formal parameters is slow
 - Inadvertent and erroneous changes may be made to the actual parameter
 - Aliases can be created.

Pass by Name

- **Pass by name** is an in-out mode parameter transmission method that does not correspond to a single implementation model.
- When parameters are passed by name, the actual parameter is textually substituted for the corresponding formal parameter in all its occurrences in the subprogram.

Accessing Nonlocal Environments

- The **non-local variables** of a subprogram are those that are visible within the subprogram but are not locally declared
- **Global variables** are those that are visible in all program units.

Writing subprograms - Functions

Functions are basic building blocks in a program. All C programs are written using functions to improve re-usability, understandability and to keep track on them. You can learn below concepts of C functions in this section in detail.

Introduction to function

A large C program is divided into basic building blocks called C function. C function contains set of instructions enclosed by “{ }” which performs specific operation in a C program. Actually, Collection of these functions creates a C program.

Function declaration, function call and function definition:

There are 3 aspects in each C function. They are,

- Function declaration or prototype - This informs compiler about the function name, function parameters and return value's data type.
- Function call – This calls the actual function
- Function definition – This contains all the statements to be executed.

S.no	C function aspects	syntax
1	function definition	return_type function_name (arguments list) { Body of function; }
2	function call	function_name (arguments list);
3	function declaration	return_type function_name (argument list);

Simple example program for C function:

- As you know, functions should be declared and defined before calling in a C program.
- In the below program, function “square” is called from main function.
- The value of “m” is passed as argument to the function “square”. This value is multiplied by itself in this function and multiplied value “p” is returned to main function from function “square”.

```
#include<stdio.h>
// function prototype, also called function declaration
float square ( float x );

// main function, program starts from here
int main( )
{

    float m, n ;
    printf ( "\nEnter some number for finding square \n");
    scanf ( "%f", &m ) ;
    // function call
```

```

        n = square ( m ) ;
        printf ( "\nSquare of the given number %f is %f",m,n );
    }

float square ( float x )    // function definition
{
    float p ;
    p = x * x ;
    return ( p ) ;
}

```

Output:

```

Enter some number for finding square
2
Square of the given number 2.000000 is 4.000000

```

How to call C functions in a program?

There are two ways that a C function can be called from a program. They are,

1. Call by value
2. Call by reference

1. Call by value:

- In call by value method, the value of the variable is passed to the function as parameter.
- The value of the actual parameter can not be modified by formal parameter.
- Different Memory is allocated for both actual and formal parameters. Because, value of actual parameter is copied to formal parameter.

Note:

- Actual parameter – This is the argument which is used in function call.
- Formal parameter – This is the argument which is used in function definition

Example program for C function (using call by value):

- In this program, the values of the variables “m” and “n” are passed to the function “swap”.
- These values are copied to formal parameters “a” and “b” in swap function and used.

```

#include<stdio.h>
// function prototype, also called function declaration
void swap(int a, int b);

int main()
{
    int m = 22, n = 44;
    // calling swap function by value
}

```

```

        printf(" values before swap  m = %d \nand n = %d", m, n);
        swap(m, n);
    }

void swap(int a, int b)
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
    printf(" \nvalues after swap m = %d\n and n = %d", a, b);
}

```

Output:

```

values before swap m = 22
and n = 44
values after swap m = 44
and n = 22

```

2. Call by reference:

- In call by reference method, the address of the variable is passed to the function as parameter.
- The value of the actual parameter can be modified by formal parameter.
- Same memory is used for both actual and formal parameters since only address is used by both parameters.

Example program for C function (using call by reference):

- In this program, the address of the variables “m” and “n” are passed to the function “swap”.
- These values are not copied to formal parameters “a” and “b” in swap function.
- Because, they are just holding the address of those variables.
- This address is used to access and change the values of the variables.

```

#include<stdio.h>
// function prototype, also called function declaration
void swap(int *a, int *b);

int main()
{
    int m = 22, n = 44;
    // calling swap function by reference
    printf("values before swap m = %d \n and n = %d",m,n);
    swap(&m, &n);
}

void swap(int *a, int *b)
{
    int tmp;
    tmp = *a;

```

```

    *a = *b;
    *b = tmp;
    printf("\n values after swap a = %d \nand b = %d", *a, *b);
}

```

Output:

```

values before swap m = 22
and n = 44
values after swap a = 44
and b = 22

```

Accessing Structure Members with Pointer

To access members of structure with structure variable, we used the dot “.” operator. But when we have a pointer of structure type, we use arrow “->” to access structure members.

```

struct Book
{
    char name[10];
    int price;
}

int main()
{
    struct Book b;
    struct Book* ptr = &b;
    ptr->name = "Dan Brown";    //Accessing Structure Members
    ptr->price = 500;
}

```

The “->” operator lets us access a member of the structure pointed to by a pointer.i.e.:

```

pt_ptr -> x = 1.0;

pt_ptr -> y = pt_ptr -> y - 3.0;

```

CHAPTER 8: FILE HANDLING

Introduction to files in programming

File - a group of related records. Files are frequently classified by the application for which they are primarily used (employee file).

Importance of file handling

File Handling concept in C language is used for store a data permanently in computer. Using this concept we can store, retrieve and change our data in Secondary memory (Hard disk) and related storage locations.

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all. However, if you have a file containing all the data, you can easily access the contents of the file using few commands in C.
- You can easily move your data from one computer to another without any changes.

File organization techniques

Data files are organized so as to facilitate access to records and to ensure their efficient storage. A tradeoff between these two requirements generally exists: if rapid access is required, more storage is required to make it possible.

Access to a record for reading it is the essential operation on data. There are two types of access:

1. **Sequential access** - is performed when records are accessed in the order they are stored.

Sequential access is the main access mode only in batch systems, where files are used and updated at regular intervals.

2. **Direct access** - on-line processing requires direct access, whereby a record can be accessed without accessing the records between it and the beginning of the file. The primary key serves to identify the needed record.

There are three methods of file organization:

1. Sequential organization
2. Indexed-sequential organization
3. Direct organization

Sequential Organization: In sequential organization records are physically stored in a specified order according to a key field in each record.

Advantages of sequential access:

1. It is fast and efficient when dealing with large volumes of data that need to be processed periodically (batch system).

Disadvantages of sequential access:

1. Requires that all new transactions be sorted into the proper sequence for sequential access processing.

2. Locating, storing, modifying, deleting, or adding records in the file require rearranging the file.

3. This method is too slow to handle applications requiring immediate updating or responses.

Indexed-Sequential Organization: In the indexed-sequential files method, records are physically stored in sequential order on a magnetic disk or other direct access storage device based on the key field of each record. Each file contains an index that references one or more key fields of each data record to its storage location address.

Direct Organization: Direct file organization provides the fastest direct access to records. When using direct access methods, records do not have to be arranged in any particular sequence on storage media. Characteristics of the direct access method include:

1. Computers must keep track of the storage location of each record using a variety of direct organization methods so that data can be retrieved when needed.

2. New transactions' data do not have to be sorted.

3. Processing that requires immediate responses or updating is easily performed.

Files Handling (Input/Output) in C programming

Types of Files

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

1. Text files

Text files are the normal .txt files that you can easily create using Notepad or any simple text editors.

When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide least security and takes bigger storage space.

2. Binary files

Binary files are mostly the .bin files in your computer.

Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold higher amount of data, are not readable easily and provides a better security than text files.

File Operations

In C, you can perform four major operations on the file, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and program.

```
FILE *fptr;
```

Opening a file - for creation and edit

Opening a file is performed using the [library function](#) in the "**stdio.h**" header file: `fopen()`.

The syntax for opening a file in standard I/O is:

```
ptr = fopen("fileopen", "mode")
```

For Example:

```
fopen("E:\\cprogram\\newprogram.txt", "w");
```

```
fopen("E:\\cprogram\\oldprogram.bin", "rb");
```

- Let's suppose the file `newprogram.txt` doesn't exist in the location `E:\\cprogram`. The first function creates a new file named `newprogram.txt` and opens it for writing as per the mode 'w'.
The writing mode allows you to create and edit (overwrite) the contents of the file.
- Now let's suppose the second binary file `oldprogram.bin` exists in the location `E:\\cprogram`. The second function opens the existing file for reading in binary mode 'rb'.
The reading mode only allows you to read the file, you cannot write into the file.

Opening Modes in Standard I/O		
File Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, <code>fopen()</code> returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, <code>fopen()</code> returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.

Opening Modes in Standard I/O		
File Mode	Meaning of Mode	During Inexistence of file
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. i.e, Data is added to end of file.	If the file does not exists, it will be created.
ab	Open for append in binary mode. i.e, Data is added to end of file.	If the file does not exists, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exists, it will be created.

Closing a File

The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using library function `fclose()`.

```
fclose(fp); //fp is the file pointer associated with file to be closed.
```

Reading and writing to a text file

For reading and writing to a text file, we use the functions `fprintf()` and `fscanf()`.

They are just the file versions of `printf()` and `scanf()`. The only difference is that, `fprint` and `fscanf` expects a pointer to the structure `FILE`.

Writing to a text file

Example 1: Write to a text file using fprintf()

```
#include <stdio.h>
int main()
{
    int num;
    FILE *fptr;
    fptr = fopen("C:\\program.txt", "w");

    if(fptr == NULL)
    {
        printf("Error!");
        exit(1);
    }

    printf("Enter num: ");
    scanf("%d", &num);

    fprintf(fptr, "%d", num);
    fclose(fptr);

    return 0;
}
```

This program takes a number from user and stores in the file `program.txt`.

After you compile and run this program, you can see a text file `program.txt` created in C drive of your computer. When you open the file, you can see the integer you entered.

Reading from a text file

Example 2: Read from a text file using fscanf()

```
#include <stdio.h>
int main()
{
    int num;
    FILE *fptr;

    if ((fptr = fopen("C:\\program.txt", "r")) == NULL) {
        printf("Error! opening file");

        // Program exits if the file pointer returns NULL.
        exit(1);    }

    fscanf(fptr, "%d", &num);

    printf("Value of n=%d", num);
    fclose(fptr);

    return 0; }
```

This program reads the integer present in the `program.txt` file and prints it onto the screen.

If you successfully created the file from **Example 1**, running this program will get you the integer you entered.

Other functions like `fgetchar()`, `fputc()` etc. can be used in similar way.

Reading and writing to a binary file

Functions `fread()` and `fwrite()` are used for reading from and writing to a file on the disk respectively in case of binary files.

Writing to a binary file

To write into a binary file, you need to use the function `fwrite()`. The function takes four arguments: Address of data to be written in disk, Size of data to be written in disk, number of such type of data and pointer to the file where you want to write.

```
fwrite(address_data, size_data, numbers_data, pointer_to_file);
```

Example 3: Writing to a binary file using fwrite()

```
#include <stdio.h>

struct threeNum
{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;

    if ((fptr = fopen("C:\\\\program.bin", "wb")) == NULL) {
        printf("Error! opening file");

        // Program exits if the file pointer returns NULL.
        exit(1);
    }

    for(n = 1; n < 5; ++n)
    {
        num.n1 = n;
        num.n2 = 5n;
        num.n3 = 5n + 1;
        fwrite(&num, sizeof(struct threeNum), 1, fptr);
    }
    fclose(fptr);

    return 0;
}
```

In this program, you create a new file `program.bin` in the C drive.

We declare a structure `threeNum` with three numbers - *n1*, *n2* and *n3*, and define it in the main function as `num`.

Now, inside the for loop, we store the value into the file using `fwrite`.

The first parameter takes the address of *num* and the second parameter takes the size of the structure `threeNum`.

Since, we're only inserting one instance of *num*, the third parameter is 1. And, the last parameter `*fptr` points to the file we're storing the data.

Finally, we close the file.

Reading from a binary file

Function `fread()` also take 4 arguments similar to `fwrite()` function as above.

```
fread(address_data, size_data, numbers_data, pointer_to_file);
```

Example 4: Reading from a binary file using `fread()`

```
#include <stdio.h>

struct threeNum
{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;

    if ((fptr = fopen("C:\\\\program.bin", "rb")) == NULL) {
        printf("Error! opening file");

        // Program exits if the file pointer returns NULL.
        exit(1);
    }

    for(n = 1; n < 5; ++n)
    {
        fread(&num, sizeof(struct threeNum), 1, fptr);
        printf("n1: %d\\tn2: %d\\tn3: %d", num.n1, num.n2, num.n3);
    }
    fclose(fptr);

    return 0;
}
```

In this program, you read the same file `program.bin` and loop through the records one by one.

In simple terms, you read one `threeNum` record of `threeNum` size from the file pointed by `*fptr` into the structure `num`.

You'll get the same records you inserted in Example 3.

Getting data using fseek()

If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record.

This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using `fseek()`.

As the name suggests, `fseek()` seeks the cursor to the given record in the file.

Syntax of fseek()

```
fseek(FILE * stream, long int offset, int whence)
```

The first parameter `stream` is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

Different Whence in fseek	
Whence	Meaning
SEEK_SET	Starts the offset from the beginning of the file.
SEEK_END	Starts the offset from the end of the file.
SEEK_CUR	Starts the offset from the current location of the cursor in the file.

Example of fseek()

```
#include <stdio.h>

struct threeNum
{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
```

```

FILE *fptr;

if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
    printf("Error! opening file");

    // Program exits if the file pointer returns NULL.
    exit(1);
}

// Moves the cursor to the end of the file
fseek(fptr, sizeof(struct threeNum), SEEK_END);

for(n = 1; n < 5; ++n)
{
    fread(&num, sizeof(struct threeNum), 1, fptr);
    printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);
}
fclose(fptr);

return 0;
}

```

This program will start reading the records from the file `program.bin` in the reverse order (last to first) and prints it.

Examples of files handling in C Programming

Write a C program to read name and marks of n number of students from user and store them in a file

```

#include <stdio.h>
int main(){
    char name[50];
    int marks,i,n;
    printf("Enter number of students: ");
    scanf("%d",&n);
    FILE *fptr;
    fptr=(fopen("C:\\student.txt","w"));
    if(fptr==NULL){
        printf("Error!");
        exit(1);
    }
    for(i=0;i<n;++i)
    {
        printf("For student%d\nEnter name: ",i+1);
        scanf("%s",name);
        printf("Enter marks: ");
        scanf("%d",&marks);
        fprintf(fptr,"\nName: %s \nMarks=%d \n",name,marks);
    }
    fclose(fptr);
    return 0;
}

```

Write a C program to read name and marks of n number of students from user and store them in a file. If the file previously exists, add the information of n students.

```
#include <stdio.h>
int main() {
    char name[50];
    int marks, i, n;
    printf("Enter number of students: ");
    scanf("%d", &n);
    FILE *fptr;
    fptr=(fopen("C:\\student.txt", "a"));
    if(fptr==NULL) {
        printf("Error!");
        exit(1);
    }
    for(i=0; i<n; ++i)
    {
        printf("For student%d\nEnter name: ", i+1);
        scanf("%s", name);
        printf("Enter marks: ");
        scanf("%d", &marks);
        fprintf(fptr, "\nName: %s \nMarks=%d \n", name, marks);
    }
    fclose(fptr);
    return 0;
}
```

Write a C program to write all the members of an array of structures to a file using fwrite(). Read the array from the file and display on the screen.

```
#include <stdio.h>
struct s
{
    char name[50];
    int height;
};
int main() {
    struct s a[5], b[5];
    FILE *fptr;
    int i;
    fptr=fopen("file.txt", "wb");
    for(i=0; i<5; ++i)
    {
        fflush(stdin);
        printf("Enter name: ");
        gets(a[i].name);
        printf("Enter height: ");
        scanf("%d", &a[i].height);
    }
    fwrite(a, sizeof(a), 1, fptr);
    fclose(fptr);
    fptr=fopen("file.txt", "rb");
    fread(b, sizeof(b), 1, fptr);
}
```

```
for(i=0;i<5;++i)
{
    printf("Name: %s\nHeight: %d",b[i].name,b[i].height);
}
fclose(fptr);
}
```


CHAPTER 9: PROGRAM DOCUMENTATION

Define program documentation

In computer hardware and software product development, documentation is the information that describes the product to its users. It consists of the product technical manuals and online information (including online versions of the technical manuals and help facility descriptions). The term is also sometimes used to mean the *source* information about the product contained in design documents, detailed code comments, white papers, and blackboard session notes.

The program documentation/Software Documentation: Documentation is the technical manual, user manual or other instructional manual that facilitate the use of a software product or services. Documentation includes source code, instructional items, error code where ever required. It may exists in any formats like PDF, Word or CDs. i'ts a kind of documentation that gives a comprehensive procedural description of a program. It shows as to how software is written. Program documentation even has the capability to sustain any later maintenance or development of the program. The program documentation describes what exactly a program does by mentioning about the requirements of the input data and the effect of performing a programming task.

Importance of programming documentation

Now we have discussed the meaning of software documentation and its time to see why it is important to have documentation.

1. **Clarify your business goals, requirements and activities:** With a proper documentation, you can share the business goals and requirement with your managers and team mates so that they have a clear vision and goals and the activity they perform will be more towards the success.
2. **Design and Specify your product:** This comes in Architectural/Design documents and it gives you complete overview of how your products look like.
3. **Everything is clearly explained:** When you makes End User documentation of the product of software, you have to explain each and everything about its working. It describes each feature of the program, and assists the user in realizing these features.
4. **Any body can work on other's code:** If you are a developer, it is not sufficient to write good codes only but you also need to take cares about the documentation part, which can be helpful to other developers while working in a team.
5. **Helpful in proper communication:** A good software documentation is helpful in proper communication. The written procedure helps you to make interaction within several departments.

Types of program documentation

Types of documentation include:

1. Requirements – Statements that identify attributes, capabilities, characteristics, or qualities of a system. This is the foundation for what will be or has been implemented.
2. Architecture/Design – Overview of software. Includes relations to an environment and construction principles to be used in design of software components.

3. Technical – Documentation of code, algorithms, interfaces, and APIs.
4. End user – Manuals for the end-user, system administrators and support staff.
5. Marketing – How to market the product and analysis of the market demand.

User Documentation

Also known as software manuals, user documentation is intended for end users and aims to help them use software properly. It is usually arranged in a book-style and typically also features table of contents, index and of course, the body which can be arranged in different ways, depending on whom the software is intended for. For example, if the software is intended for beginners, it usually uses a tutorial approach and guides the user step-by-step. Software manuals which are intended for intermediate users, on the other hand, are typically arranged thematically, while manuals for advanced users follow reference style.

Besides printed version, user documentation can also be available in an online version or PDF format. Often, it is also accompanied by additional documentation such as video tutorials, knowledge based articles, videos, etc.

Requirements Documentation

Requirements documentation, also referred to simply as requirements explains what a software does and shall be able to do. Several types of requirements exist which may or may not be included in documentation, depending on purpose and complexity of the system. For example, applications that don't have any safety implications and aren't intended to be used for a longer period of time may be accompanied by little or no requirements documentation at all. Those that can affect human safety or/and are created to be used over a longer period of time, on the other hand, come with an exhausting documentation.

Architecture Documentation

Also referred to as software architecture description, architecture documentation either analyses software architectures or communicates the results of the latter (work product). It mainly deals with technical issues including online marketing and **seo services** but it also covers non-technical issues in order to provide guidance to system developers, maintenance technicians and others involved in the development or use of architecture including end users. Architecture documentation is usually arranged into architectural models which in turn may be organized into different views, each of which deals with specific issues.

Comparison document is closely related to architecture documentation. It addresses current situation and proposes alternative solutions with an aim to identify the best possible outcome. In order to be able to do that, it requires an extensive research.

Technical Documentation

Technical documentation is a very important part of software documentation and many programmers use both terms interchangeably despite the fact that technical documentation is only one of several types of software documentation. It describes codes but it also addresses algorithms, interfaces and other technical aspects of software development and application. Technical documentation is usually created by the programmers with the aid of auto-generating tools.

Writing System/Program documenting

We can divide documentation into two basic forms:-

- **System documentation:-** Records a detailed information about a system's design specifications, its internal workings, and its functionality. It can further be divided
 - **Internal Documentation-** Part of the program source code or is generated at compile time.
 - **External Documentation - -** Includes the outcome of structured diagramming techniques such as data flow and entity relationship diagrams.
- **User Documentation:-** Consists of written or other visual information about an application system, how it works and how to use it

Most user documentations are now delivered on-line in hypertext format instead of bulky paper manuals.

CHAPTER 10: EMERGING TRENDS OF STRUCTURED PROGRAMMING

Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of subroutines, block structures, for and while loops—in contrast to using simple tests and jumps such as the go to statement which could lead to "spaghetti code" causing difficulty to both follow and maintain.

It emerged in the late 1950s with the appearance of the ALGOL 58 and ALGOL 60 programming languages, with the latter including support for block structures. Contributing factors to its popularity and widespread acceptance, at first in academia and later among practitioners, include the discovery of what is now known as the structured program theorem in 1966, and the publication of the influential "Go To Statement Considered Harmful" open letter in 1968 by Dutch computer scientist Edsger W. Dijkstra, who coined the term "structured programming".

It is possible to do structured programming in any programming language, though it is preferable to use something like a procedural programming language. Some of the languages initially used for structured programming include: ALGOL, Pascal, PL/I and Ada – but most new procedural programming languages since that time have included features to encourage structured programming, and sometimes deliberately left out features – notably GOTO – in an effort to make unstructured programming more difficult. *Structured programming* (sometimes known as modular programming) enforces a logical structure on the program being written to make it more efficient and easier to understand and modify.

Structured programming is most frequently used with deviations that allow for clearer programs in some particular cases, such as when exception handling has to be performed.