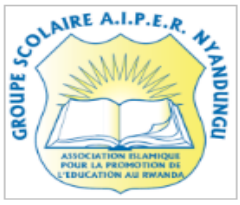
NOTES FOR L4 SOD



SFDAF401

ALGORITHM FUNDAMENTALS

Apply algorithm fundamentals

Competence

REQF Level: 4

Credits: 10

Learning hours

L: 100

Sector: ICT

Sub-sector: Software Development

Issue date: February, 2019

Purpose statement

The aim of this module is to gain skills, knowledge and attitude required to apply algorithm concepts based on programming. At the end of this module, the learner will be able to describe basic concepts of algorithm, apply programming structures, iterative constructs and structured programming and introduce data structure and elementary sorting algorithm.

By the end of this module, the learner will be able to apply algorithm fundamentals.

Learning assumed to be in place

You should have a basic understanding in basic mathematics

Elements of competence and performance criteria

Learning units describe the essential outcomes of a competence.
Performance criteria describe the required performance needed to demonstrate achievement of the learning unit.

By the end of the module, the trainee will be able to:

Elements of competence	Performance criteria
1. Describe basic concept of algorithm	1.1. Proper conversion of number system from one base to another 1.2. Adequate description of the key concepts of algorithm in programming 1.3. Effective revision of relevant basic data types and operators in programming 1.4. Appropriate description of basic input-output of algorithm pseudo-codes in programming
2. Apply programming structures, iterative constructs and structured programming	2.1. Accurate use of conditional statements in programming 2.2. Appropriate use of sequential statements in line with programming pseudo-codes 2.3. Appropriate use of iterative statements in line with programming
3. Describe data structure	3.1. Effective description of lists data structure in line with programming 3.2. Effective description of array in line with programming 3.3. Effective description of data structure searching and sorting techniques in line with programming

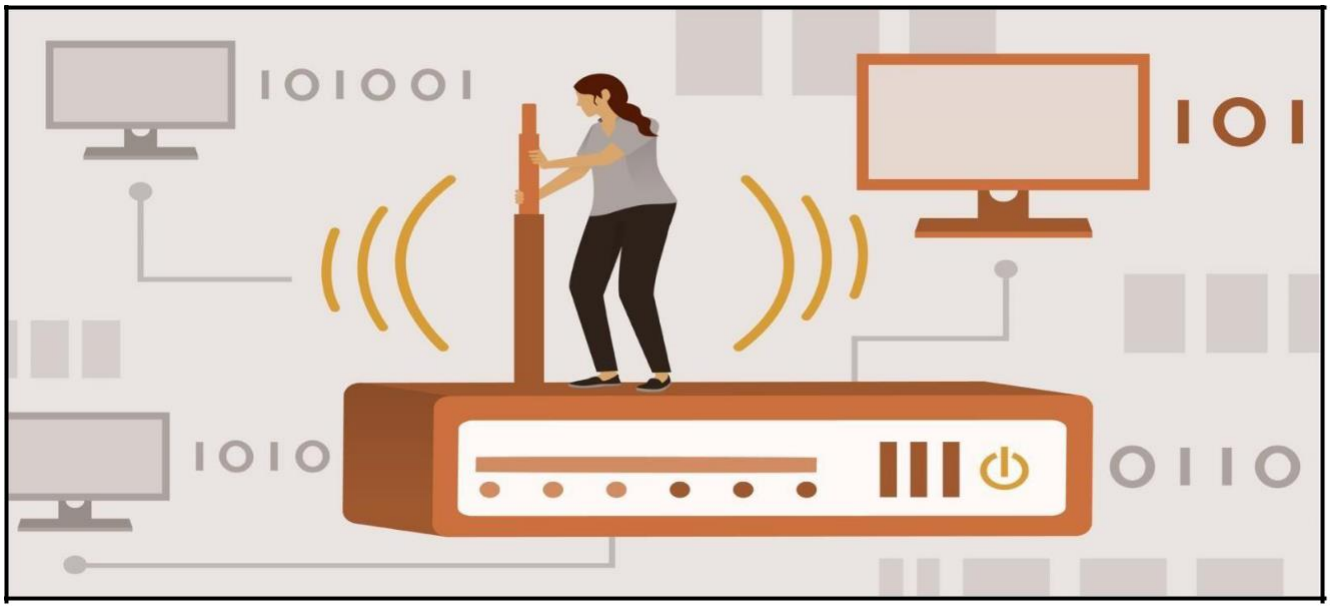
INTRODUCTION TO ALGORITHM FUNDAMENTALS

Topic 1.1: Convert number system from one base to another

Key Competencies:

Knowledge	Skills	Attitudes
1. Describe algorithm concepts	**1.** Identify algorithm concepts	**1.** Be Critical thinker
2. Describe logic concepts	**2.** Apply logic concepts	**2.** Attentive
3. Describe base conversion types	**3.** Perform base conversion types	**3.** Team Work spirit

Getting Started: Look the picture below and answer the following questions.



A. What do you see on the above diagram?

B. Are they relationship between the picture and the topic?

Activity 1: Problem Solving

Task: SJITC Nyamirambo has to recruit a trainer who will train in Software development. Now, to be sure that the trainer is on the requiered standard, they gave him some questions to solve related to base conversion that can be applied in algorithm.

1. Describe base conversion types
2. Describe algorithm concepts
3. Describe logic concepts

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages. i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms

- **Search** − Algorithm to search an item in a data structure.
- **Sort** − Algorithm to sort items in a certain order.
- **Insert** − Algorithm to insert item in a data structure.
- **Update** − Algorithm to update an existing item in a data structure.
- **Delete** − Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

Let's try to learn algorithm-writing by using an example.

Problem − Design an algorithm to add two numbers and display the result.

Step 1 − START

Step 2 − declare three integers **a**, **b** & **c**

Step 3 − define values of **a** & **b**

Step 4 − add values of **a** & **b**

Step 5 − store output of step 4 to **c**

Step 6 − print **c**

Step 7 − STOP

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as −

Step 1 − START ADD

Step 2 − get values of **a** & **b**

Step 3 − c ← a + b

Step 4 − display c

Step 5 – STOP

Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following −

- ***A Priori*** **Analysis** − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- ***A Posterior*** **Analysis** − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

Algorithm Complexity

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components

- A fixed part that is a space required to store certain data and variables, which are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursions stack space, etc.

Algorithm: SUM(A, B)

Step 1 - START

Step 2 - C ← A + B + 10

Step 3 - Stop

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes **n** steps. Consequently, the total computational time is $T(n) = c * n$, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

1. Algorithm logic concepts

1.1. Boolean gates

The table used to represent the boolean expression of a logic gate function is commonly called a **Truth Table**. A logic gate truth table shows each possible input combination to the gate or circuit with the resultant output depending upon the combination of these input(s).

For example, consider a single **2-input** logic circuit with input variables labelled as A and B. There are "four" possible input combinations or 2^2 of "OFF" and "ON" for the two inputs.

However, when dealing with Boolean expressions and especially logic gate truth tables, we do not general use “ON” or “OFF” but instead give them bit values which represent a logic level “1” or a logic level “0” respectively.

Then the four possible combinations of A and B for a 2-input logic gate is given as:

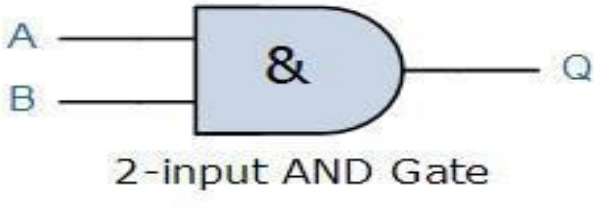
- Input Combination 1. – “OFF” – “OFF” or (0, 0)
- Input Combination 2. – “OFF” – “ON” or (0, 1)
- Input Combination 3. – “ON” – “OFF” or (1, 0)
- Input Combination 4. – “ON” – “ON” or (1, 1)

Therefore, a 3-input logic circuit would have 8 possible input combinations or 2^3 and a 4-input logic circuit would have 16 or 2^4, and so on as the number of inputs increases. Then a logic circuit with “n” number of inputs would have 2^n possible input combinations of both “OFF” and “ON”.

So in order to keep things simple to understand, in this tutorial we will only deal with standard **2-input** type logic gates, but the principals are still the same for gates with more than two inputs.

Then the Truth tables for a 2-input AND Gate, a 2-input OR Gate and a single input NOT Gate are given as: **2-input AND Gate**

For a 2-input AND gate, the output Q is true if BOTH input A “AND” input B are both true, giving the Boolean Expression of: (Q = A and B).

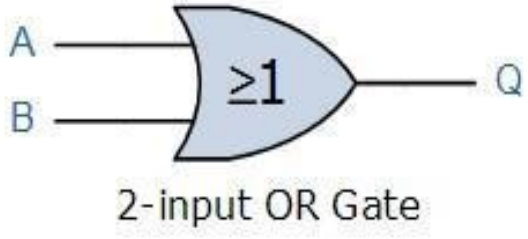
<table>
<tr><th>Symbol</th><th colspan="3">Truth Table</th></tr>
<tr><td rowspan="5"> 

2-input AND Gate</td><td>A</td><td>B</td><td>Q</td></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
<tr><td>Boolean Expression Q = A.B</td><td colspan="3">Read as A AND B gives Q</td></tr>
</table>

Note that the Boolean Expression for a two input AND gate can be written as: A.B or just simply AB without the decimal point.

2-input OR (Inclusive OR) Gate

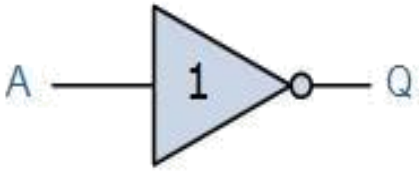
For a 2-input OR gate, the output Q is true if EITHER input A "OR" input B is true, giving the Boolean Expression of: (Q = A or B).

<table>
<tr><th>Symbol</th><th colspan="3">Truth Table</th></tr>
<tr><td rowspan="5"> 

2-input OR Gate</td><td>A</td><td>B</td><td>Q</td></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
<tr><td>Boolean Expression Q = A+B</td><td colspan="3">Read as A OR B gives Q</td></tr>
</table>

NOT Gate (Inverter)

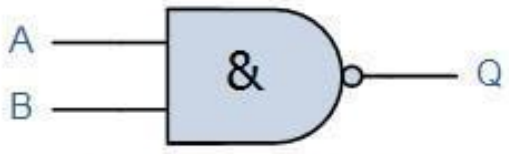
For a single input NOT gate, the output Q is ONLY true when the input is "NOT" true, the output is the inverse or complement of the input giving the Boolean Expression of: (Q = NOT A).

Symbol	Truth Table	
 Inverter or NOT Gate	A	Q
	0	1
	1	0
Boolean Expression Q = NOT A or $\overline{A}$	Read as inversion of A gives Q	

The NAND and the NOR Gates are a combination of the AND and OR Gates respectively with that of a NOT Gate (inverter).

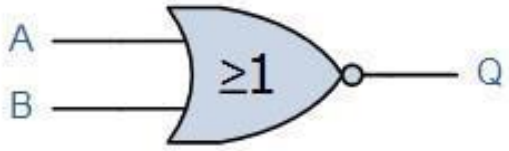
2-input NAND (Not AND) Gate

For a 2-input NAND gate, the output Q is true if BOTH input A and input B are NOT true, giving the Boolean Expression of: (Q = not(A AND B)).

Symbol	Truth Table		
 2-input NAND Gate	A	B	Q
	0	0	1
	0	1	1
	1	0	1
	1	1	0
Boolean Expression Q = $\overline{A.B}$	Read as A AND B gives NOT-Q		

2-input NOR (Not OR) Gate

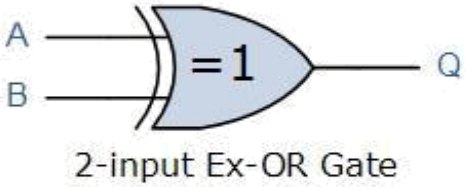
For a 2-input NOR gate, the output Q is true if BOTH input A and input B are NOT true, giving the Boolean Expression of: (Q = not(A OR B)).

Symbol	Truth Table		
 2-input NOR Gate	A	B	Q
	0	0	1
	0	1	0
	1	0	0
	1	1	0
Boolean Expression Q = $\overline{A+B}$	Read as A OR B gives NOT-Q		

As well as the standard logic gates there are also two special types of logic gate function called an ExclusiveOR Gate and an Exclusive-NOR Gate. The Boolean expression to indicate an Exclusive-OR or Exclusive-NOR function is to a symbol with a plus sign inside a circle, ($\oplus$). The switching actions of both of these types of gates can be created using the above standard logic gates. However, as they are widely used functions they are now available in standard IC form and have been included here as reference.

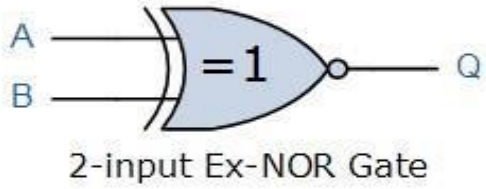
2-input EX-OR (Exclusive OR) Gate

For a 2-input Ex-OR gate, the output Q is true if EITHER input A or if input B is true, but NOT both giving the Boolean Expression of: (Q = (A and NOT B) or (NOT A and B)).

Symbol	Truth Table		
 2-input Ex-OR Gate	A	B	Q
	0	0	0
	0	1	1
	1	0	1
	1	1	0
Boolean Expression $Q = A \oplus B$			

2-input EX-NOR (Exclusive NOR) Gate

For a 2-input Ex-NOR gate, the output Q is true if BOTH input A and input B are the same, either true or false, giving the Boolean Expression of: (Q = (A and B) or (NOT A and NOT B)).

Symbol	Truth Table		
 2-input Ex-NOR Gate	A	B	Q
	0	0	1
	0	1	0
	1	0	0
	1	1	1
Boolean Expression $Q = \overline{A \oplus B}$			

Summary of 2-input Logic Gates

The following Truth Table compares the logical functions of the 2-input logic gates above.

Inputs		Truth Table Outputs For Each Gate					
A	B	AND	NAND	OR	NOR	EX-OR	EX-NOR
0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	0	1	0	0	1

The following table gives a list of the common logic functions and their equivalent Boolean notation.

Logic Function	Boolean Notation
AND	A.B
OR	A+B
NOT	$\overline{A}$
NAND	$\overline{A.B}$
NOR	$\overline{A+B}$
EX-OR	$(A.\overline{B}) + (\overline{A}.B)$ or $A \oplus B$
EX-NOR	$(A.B) + (\overline{A}.\overline{B})$ or $\overline{A \oplus B}$

1.2. Decimal numeration system

All number systems have the same three characteristics: **digits**, **base** and **weight**

i. Decimal numeration system

- The decimal numeration system uses only ten ciphers (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) used in "Weighted" positions to represent very large and very small numbers.
- Each cipher represents an integer quantity, and each place from right to left in the notation represents a multiplying constant, or weight, for each integer quantity.

1.3. Binary numeration system

a) The binary numeration system uses only two ciphers instead of ten as the decimal numeration system.

   Those two ciphers are “0” and “1”. In binary system of numeration, ciphers are called bit (Binary Digit).

b) Cipher are arranged right to left in doubling values of weight (instead of multiplying the weight by 10 as in the case of decimal system).

Remark: With n bits we can represent **2*n*** different binary numbers. The higher H number is given using the following formula. **H =2n – 1**

1.4. Octal Number System

The octal number system has a base or radix of eight, meaning that it has eight possible digits or symbols: 0, 1,2,3,4,5,6,7. How do you count in Octal number system?

1.5. Hexadecimal Number System

The hexadecimal system uses base 16. Thus, it has 16 possible digit symbols. It uses the digits

0, 1, 2, 3… 9 plus the letters A, B, C, D, E, and F.

How do you count in Hexadecimal number system?

2. Perform base conversion

2.1. Decimal to binary conversion

- In the most popular method called the dibble-dabble method, the given decimal number is successively divided by 2 giving a succession of remainders of **0** or **1**.
- The remainders read in the reverse order give the binary equivalent of the given decimal number.

2.2. Decimal to octal conversion

Decimal to octal conversion can be done by the dibble-dabble method by successively dividing by 8, giving succession of remainders lying between 0 and 7.

The remainders written in reverse order give the octal equivalent of given decimal number.

2.3. Decimal to hexadecimal conversion

Decimal to hexadecimal conversion can be done by the dibble-dabble method of successively dividing by 16 giving succession of remainders lying between 0 and 15. Then the remainders between 10 and 15 are renamed in hex as follows 10→A, 11→B, 12→C, 13→D, 14→E,

15→F.

The remainders written in reverse order give the hexadecimal equivalent of the given decimal number.

2.4. Binary to Decimal conversion

Convert 1011.1012 to decimal

1011.1012 = (1 x 23) + (0 x 22) + (1 x 21) + (1 x 20) + (1 x 2-1) + (0 x 2-2) + (1 x 2-3) =

(1 x 8) + (0 x 4) + (1 x 2) + (1 x 1) + (1 x 1/2) + (0 x 1/4) + (1 x 1/8) = 8 + 0 + 2 + 1 +

0.5 + 0 + 0.125 = 11.625

2.5. Binary-To-Octal / Octal-To-Binary Conversion

- To convert a binary number into octal simply divide the number into groups of three bits each starting at the binary point.
- Then express each group by its decimal (or octal) equivalent.

2.6. Binary-To-Hexadecimal / Hexadecimal-To-Binary Conversion

Binary-ToHexadecimal

Steps:

1. Write down the full binary number

2. Split the number into 4 bit groups starting from the right.

3. Substitute the equivalent hexadecimal digit for each group.

People use the decimal number system to perform arithmetic operations. Computers, on the other hand, use the binary system, which contains only two digits: 0 and 1. We need a way to convert numbers from one system to another.

Activity 2: Guided Practice

Task: The table below is in the **wrong order.**

1. Draw a line from the term on the left, to the correct base on the right.

binary		base 2
decimal		base 8
hexadecimal		base 10
octal		base 16

Activity 3: Application

Task: 1. Perform the following question

An alarm clock is controlled by a microprocessor. It uses the 24 hour clock. The hour is represented by an 8-bit register, **A**, and the number of minutes is represented by another 8-bit register, **B**.

(a) Identify what time is represented by the following two 8-bit registers.

A									B								
128	64	32	16	8	4	2	1		128	64	32	16	8	4	2	1
0	0	0	1	0	0	1	0	:	0	0	1	1	0	1	0	1

Hours ..... Minutes .....

[2]

(b) An alarm has been set for 07:30. Two 8-bit registers, **C** and **D**, are used to represent the hours and minutes of the alarm time.

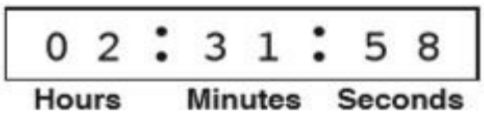
Show how 07:30 would be represented by these two registers:

C									D							
								:								
Hours									Minutes							

2. Perform the following question

A stopwatch uses six digits to display hours, minutes and seconds.

The stopwatch is stopped at:



An 8-bit register is used to store each pair of digits.

(a) Write the 8-bit binary numbers that are currently stored for the **Hours**, **Minutes** and **Seconds**.

Hours								
Minutes								
Seconds								

Points to Remember

- **Don't forget that decimal system each digit has a value ten times greater than its previous number**
- **Each integer number column has values of units, tens, hundreds, thousands, etc.**
- **The Binary Numbering System is the most fundamental numbering system in all digital and computer based systems**

Formative Assessment

1. Convert 85e8 from base 16 to binary

2. Convert 1010 1000 from base 2 to decimal

3. Convert 58bf from hexadecimal to binary

4. Convert 121 from base 10 to base 2

5. Convert 111 110 000 from base 2 to base 8

6. Convert 101 111 110 from base 2 to octal

7. Convert 193 from base 10 to binary

8. Convert 140 from base 10 to base 2

9. Convert 110 from decimal to binary

10. Convert 1011 1110 0000 1001 from base 2 to base 16

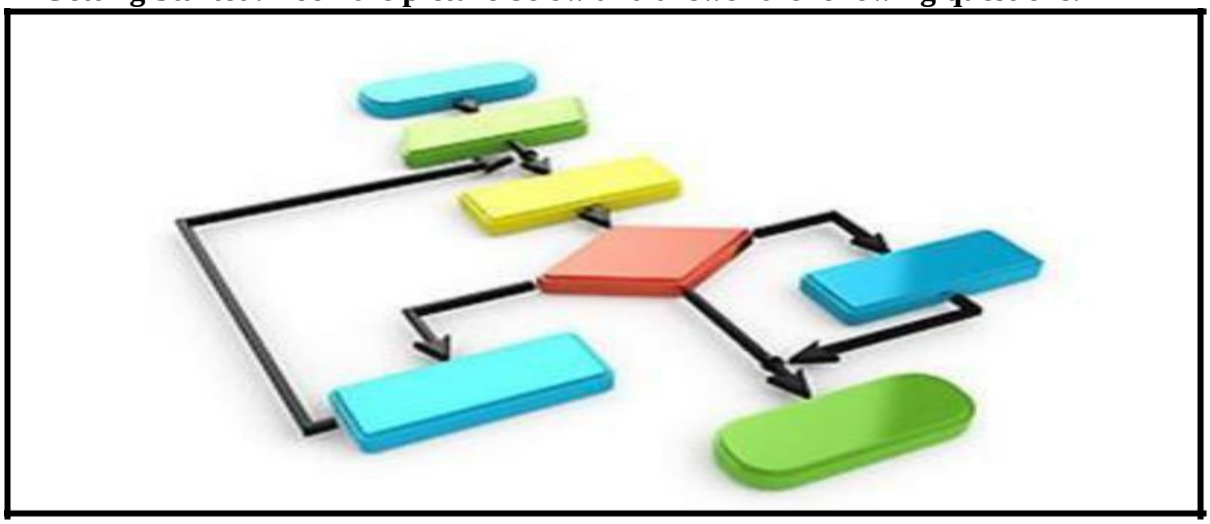
1.2 Key concepts of algorithm description

Topic 1.2: Describe the key concepts of algorithm in programming

Key Competencies:

Knowledge	Skills	Attitudes
1. Describe Algorithm fundamentals	**1.** Use Datatype	**1.** Be Critical thinker
2. Describe qualities of good algorithm	**2.** Use qualities of good algorithm	**2.** Attentive
3. Describe algorithm steps and flowcharts	**3.** Draw algorithm steps and flowcharts	**3.** Team Work spirit

🔁 Getting Started: Look the picture below and answer the following questions.



C. What do you see on the above diagram?

D. Are they relationship between the picture and the topic?

Activity 1: Problem Solving

Task: RP Finance wanted to perform the slary calculation by making sum of the salary from January to december in order to know induvidual sarary and the overall salaries of all employess.

1. Describe algorithm steps and flowcharts

2. Describe qualities of good algorithm

Key Facts 1.2

KEY CONCEPTS OF ALGORITHM DESCRIPTION

1.INTRODUCTION TO ALGORITHM

1.1. Definition of Algorithms

The word Algorithm means "***a process or set of rules to be followed in calculations or other problem-solving operations"***. Therefore, Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed upon in order to get the expected results.

It can be understood by taking an example of cooking a new recipe. To cook a new recipe, one reads the instructions and steps and execute them one by one, in the given sequence. The result thus obtained is the new dish cooked perfectly. Similarly, algorithms help to do a task in programming to get the expected output.

The Algorithm designed are language-independent, i.e. they are just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

1.2. How to Design an Algorithm?

In order to write an algorithm, following things are needed as a pre-requisite:

1. The **problem** that is to be solved by this algorithm.
2. The **constraints** of the problem that must be considered while solving the problem.
3. The **input** to be taken to solve the problem.
4. The **output** to be expected when the problem is solved.
5. The **solution** to this problem, in the given constraints.

Then the algorithm is written with the help of above parameters such that it solves the problem.

Example: Consider the example to add three numbers and print the sum.

Step 1: Fulfilling the pre-requisites

As discussed above, in order to write an algorithm, its pre-requisites must be fulfilled.

1. **The problem that is to be solved by this algorithm**: Add 3 numbers and print their sum.
2. **The constraints of the problem that must be considered while solving the problem**: The numbers must contain only digits and no other characters.
3. **The input to be taken to solve the problem:** The three numbers to be added.
4. **The output to be expected when the problem the is solved:** The sum of the three numbers taken as the input.
5. **The solution to this problem, in the given constraints:** The solution consists of adding the 3 numbers. It can be done with the help of '+' operator, or bit-wise, or any other method.

Step 2: Designing the algorithm

Now let's design the algorithm with the help of above pre-requisites:

Algorithm to add 3 numbers and print their sum:

- START
- Declare 3 integer variables num1, num2 and num3.
- Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.
- Declare an integer variable sum to store the resultant sum of the 3 numbers.
- Add the 3 numbers and store the result in the variable sum.
- Print the value of variable sum
- END

Step 3: Testing the algorithm by implementing it.

In order to test the algorithm, let's implement it in C language.

1.3. Source code in algorithm

Source code is the list of human-readable instructions that a programmer writes often in a word processing program when he is developing a program.

The source code is run through a compiler to turn it into machine code, also called object code, that a computer can understand and execute. Object code consists primarily of 1s and 0s, so it isn't human-readable.

Source Code Example

Source code and object code are the before and after states of a computer program that is compiled. Programming languages that compile their code include C, C++, Delphi, Swift, Fortran, Haskell, Pascal and many others. Here is an example of C language source code:

```
/* Hello World program */
#include<stdio.h> main()
{
printf("Hello World")
}
```

You don't have to be a computer programmer to tell that this code has something to do with printing "Hello World." Of course, most source code is much more complex than this example. It is not unusual for software programs to have millions of lines of code. Windows 10 operating system is reported to have about 50 million lines of code.

Source Code Licensing

Source code can be either proprietary or open. Many companies closely guard their source code. Users can use the compiled code, but they cannot see or modify it. Microsoft Office is an example of proprietary source code. Other companies post their code on the internet where it is free to anyone to download. Apache Open Office is an example of open source software code

1.4. Machine code

Machine code, also called machine language, is a computer language that is directly understandable by a computer's CPU (central processing unit), and it is the language into which all programs must be converted before they can be run. Each CPU type has its own machine language, although they are basically fairly similar.

After the source code for a program has been written by one or more humans in a programming language (e.g., C or C++), it is compiled (i.e., converted) into machine code by a specialized program called a compiler, or by an assembler in the case of assembly language. This machine code is then stored as an executable file (i.e., a ready-to-run program) and can be executed (i.e., run) by the operating system any time it is instructed to do so by another program or by a user.

Machine code is extremely difficult for humans to read because it consists merely of patterns of bits (i.e., zeros and ones). Thus, programmers who want to work at the machine code level instead usually use assembly language, which is a human-readable notation for the machine language in which the instructions represented by patterns of zeros and ones are replaced with alphanumeric symbols (called mnemonics) in order to make it easier to remember and work with them (including reducing the chances of making errors). In contrast to highlevel languages (e.g., C, C++, Java, Perl and Python), there is a nearly one to one correspondence between a simple assembly language and its corresponding machine language.

Programs for the first electronic computers were written directly in machine code. However, the development of assembly language from the 1950s led to a large increase in programmer productivity. Initially, programs written in assembly language programs were hand-translated into machine code, but this tedious task was later eliminated by the development of assemblers to automate the translations

1.5. Editing

Editors or text editors are software programs that enable the user to create and edit text files. In the field of programming, the term editor usually refers to source code editors that include many special features for writing and editing code. Notepad, Wordpad are some of the common editors used on Windows OS and vi, emacs, Jed, pico are the editors on UNIX OS. Features normally associated with text editors are — moving the cursor, deleting, replacing, pasting, finding, finding and replacing, saving etc.

1.5.1. Types Of Editors

There are generally five types of editors as described below:

1. **Line editor**: In this, you can only edit one line at a time or an integral number of lines. You cannot have a free-flowing sequence of characters. It will take care of only one line.

   Ex : Teleprinter, edlin, teco

2. **Stream editors**: In this type of editors, the file is treated as continuous flow or sequence of characters instead of line numbers, which means here you can type paragraphs.

Ex : Sed editor in UNIX

3. **Screen editors**: In this type of editors, the user is able to see the cursor on the screen and can make a copy, cut, paste operation easily. It is very easy to use mouse pointer.

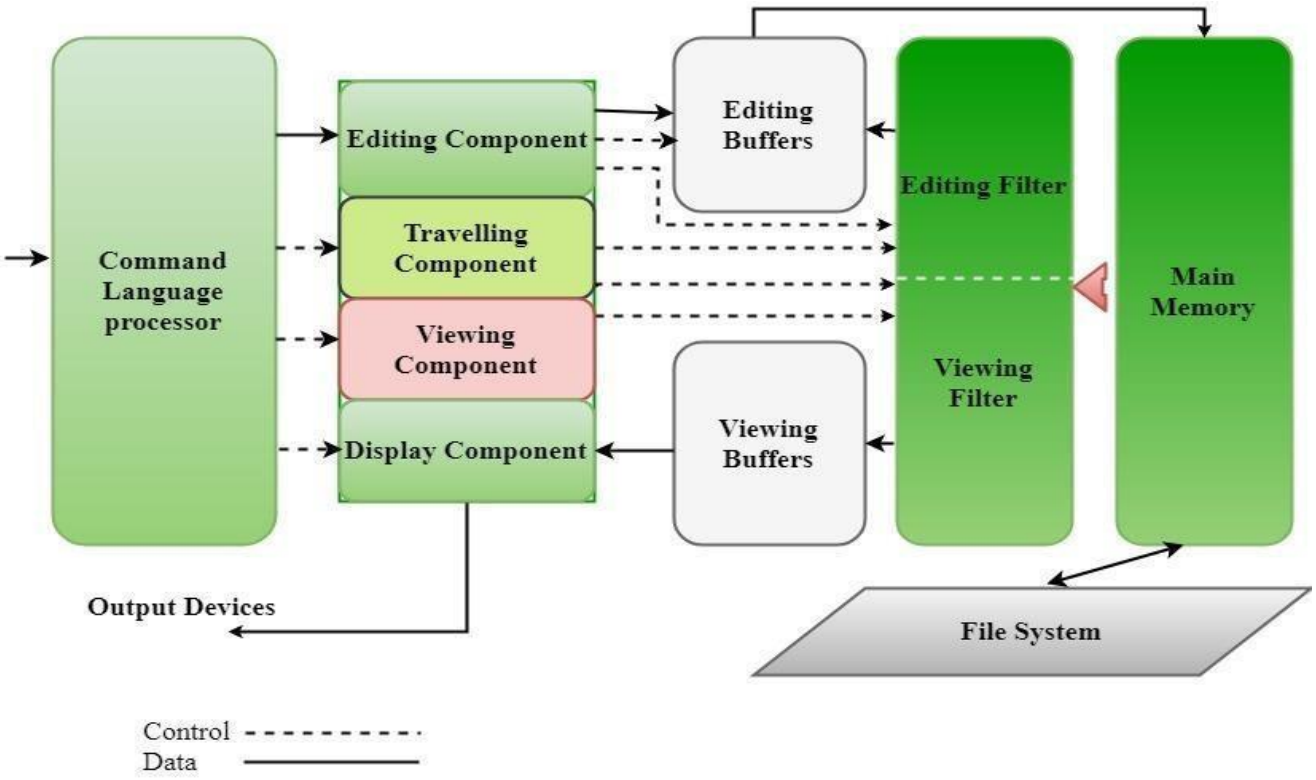
Ex : vi, emacs, Notepad

4. **Word Processor**: Overcoming the limitations of screen editors, it allows one to use some format to insert images, files, videos, use font, size, style features. It majorly focuses on Natural language.

5. **Structure Editor**: Structure editor focuses on programming languages. It provides features to write and edit source code.

Ex : Netbeans IDE, gEdit.

1.5.2 Editor Structure



The command language processor accepts commands, performs functions such as editing and viewing.

It involves traveling, editing, viewing and display.

Editing operations are specified by the user and display operations are specified by the editor.

Traveling and viewing components are invoked by the editor or the user itself during the operations.

Editing component is a module dealing with editing tasks.

The current editing area is determined by the current editing pointer associated with the editing component.

When editing command is made, the editing component calls the editing filter, generates a new editing buffer.

Editing buffer contains the document to be edited at the current editor pointer location. In viewing a document, the start of the area to be viewed is determined by the current viewing pointer.

Viewing component is a collection of modules used to see the next view. Current viewing can be made to set or reset depending upon the last operation.

When display needs to be updated, the viewing component invokes the viewing filter, generates a new buffer and it contains the document to be viewed using the current view buffer.

Then the viewing buffer is pass to the display component which produces the display by buffer mapping.

The editing and viewing buffers may be identical or completely disjoint.

The editing and viewing buffers can also partially overlap or can be contained one within the another.

The component of the editor interacts with the document from the user on two levels: main memory and the disk files system.

1.6. Compiling vs interpreting

We generally write a computer program using a high-level language. A high-level language is one which is understandable by us humans.

It contains words and phrases from the English (or other) language. But a computer does not understand highlevel language.

It only understands program written in 0's and 1's in binary, called the machine code. A program written in high-level language is called a source code. We need to convert the source code into machine code and this is accomplished by compilers and interpreters.

Hence, a compiler or an interpreter is a program that converts program written in high-level language into machine code understood by the computer.

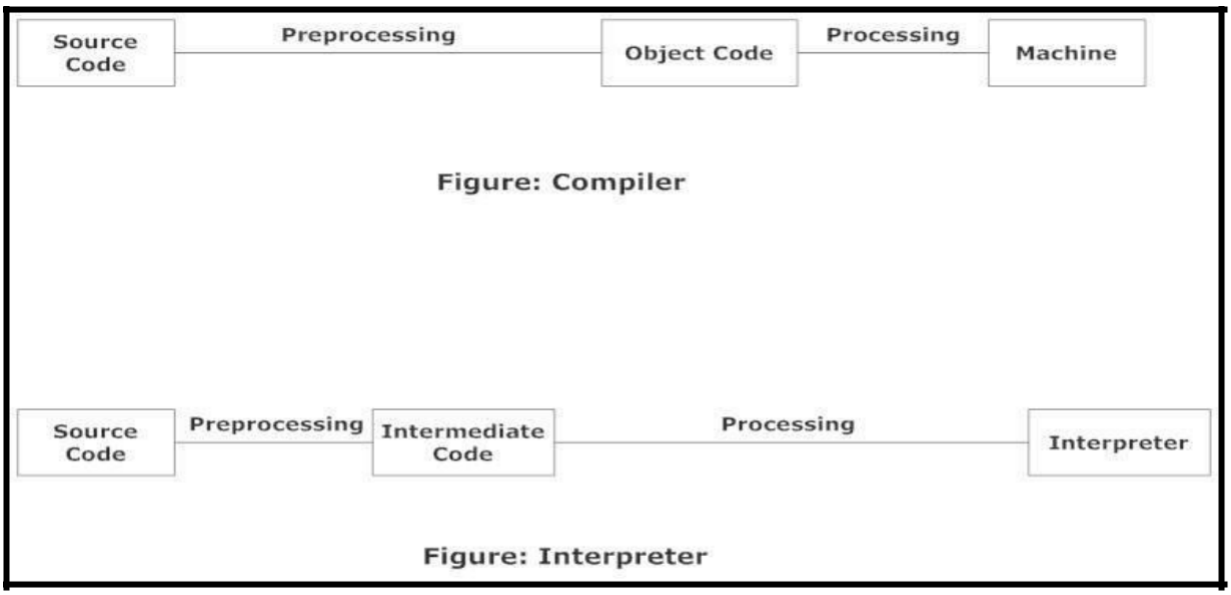
The difference between an interpreter and a compiler is given below:

Interpreter	**Compiler**
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.

Continues translating the program until the first it generates the error message only after scanning the error is met, in which case it stops. Hence debugging whole program. Hence debugging is comparatively is easy. hard.

Programming language like Python, Ruby use Programming language like C, C++ use compilers.

interpreters.



1.7 linking vs debugging

1.7.1 Debugging

It is a systematic process of spotting and fixing the number of bugs, or defects, in a piece of software so that the software is behaving as expected.

Debugging is harder for complex systems in particular when various subsystems are tightly coupled as changes in one system or interface may cause bugs to emerge in another.

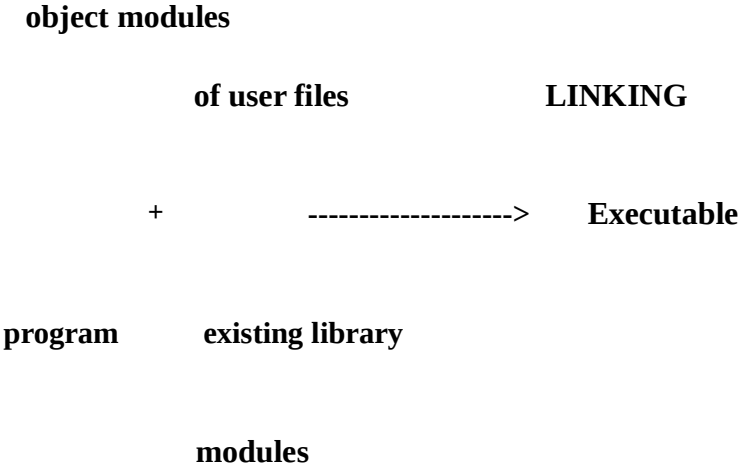
Debugging is a developer activity and effective debugging is very important before testing begins to increase the quality of the system. Debugging will not give confidence that the system meets its requirements completely but testing gives confidence.

1.7.2 Program Linking

A whole program usually is not written in a single file. Apart from code and data definitions in multiple files, a user code often makes references to code and data defined in some "libraries".

Linking is the process in which references to "externally" defined objects (code and data) are processed so as to make them operational. Traditionally linking used to be performed as a task after basic translation of the user program files and the output of this stage is a single executable program file.

This is known as static linking. A more versatile technique is more commonly used these days which is called - dynamic linking.



Two important aspects in linking are - locating the individual object modules in the combined executable program image, and adjusting the addresses used for external references in the various places in the program.

2. Description of qualities of a good algorithm

a. **Finiteness:** the algorithm stops after a finite number of instructions are executed.

b. **Definiteness:** Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.

c. **Input:** the algorithm receives input. An algorithm has zero or more inputs, i.e, quantities which are given to it initially before the algorithm begins.

d. **Output:** the algorithm produces output. An algorithm has one or more outputs i.e, quantities which have a specified relation to the inputs.

e. **Effectiveness:** An algorithm is also generally expected to be effective. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time.

f. **Precision** – the steps are precisely stated (defined).

g. **Uniqueness** – results of each step are uniquely defined and only depend on the input and the result of the preceding steps.

h. **Generality** – the algorithm applies to a set of inputs.

3. Algorithm and flowchart explained with examples

3.1 Description of algorithm and flowchart?

Algorithm and flowchart are programming tools. A Programmer uses various programming languages to create programs.

Before actually writing a program in a programming language, a programmer first needs to find a procedure for solving the problem which is known as planning the program.

The program written without proper pre-planning has higher chances of errors. The tools that are used to plan or design the problem are known as programming tools. Algorithm and flowchart are widely used programming tools.

3.2 Algorithm

The word "algorithm" relates to the name of the mathematician Al-Khwarizmi, which means a procedure or a technique. Programmer commonly uses an algorithm for planning and solving the problems.

An algorithm is a specific set of meaningful instructions written in a specific order for carrying out or solving a specific problem.

3.2.1Types of Algorithm

The algorithm and flowchart are classified into three types of control structures.

1. *Sequence*
2. *Branching(Selection)*
3. *Loop(Repetition)*

According to the condition and requirement, these three control structures can be used.

In the sequence *structure, statements are placed one after the other and the execution takes place starting from up to down.*

Whereas in branch control, there is a condition and according to a condition, a decision of either TRUE or FALSE is achieved. In the case of TRUE, one of the two branches is explored; but in the case of FALSE condition, the other alternative is taken. Generally, the 'IF-THEN' is used to represent branch control.

Example:

Write an algorithm to find the smallest number between two numbers *Step1: Start*

Step2: Input two numbers, say A and B

Step3: If A<B then small = A

Step4: If B<A then Small = B

Step5: Print Small

Step 6: End

Write an algorithm to check odd or even number

Step1: Start

Step2: Read/Input a number and store in A Step3: Is A<0?

If YES then C="ODD"

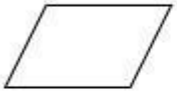
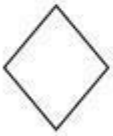
If NO then c="even"

Step4: Display C

Step5: Stop

3.3 Flowchart

A flowchart is the graphical or pictorial representation of an algorithm with the help of different symbols, shapes and arrows in order to demonstrate a process or a program. With algorithms, we can easily understand a program. The main purpose of a flowchart is to analyze different processes. Several standard graphics are applied in a flowchart:

The symbols	*Name*	*Use*
	Terminal	Indicates the beginning and of a programs
	Process	A calculation or assigning of a value to a variable.
	Input/output (I/O)	Any statement that causes date to be input to a program (INPUT, READ) or output from the program, such as printing on the display screen or printer.
	Decision	Program decisions. Allows alternate courses of action-based condition. A decision indicates a question that can be answered yes or no (true or false)
	Connector	Can be used to eliminate lengthy floweriness. Its used indicates that one symbol is connected to another.
	Floweriness Arrowheads	Used to connect symbols and indicate the sequence of operations. The flow is assumed to go from top to bottom and from left to right. Arrowheads are only required when the flow violates standard directions

The graphics above represent different part of a flowchart. The process in a flowchart can be expressed through boxes and arrows with different sizes and colors. In a flowchart, we can easily highlight a certain element and the relationships between each part.

3.3.1 How to Use Flowcharts to Represent Algorithms

Now that we have the definitions of algorithm and flowchart, how do we use a flowchart to represent an algorithm?

Algorithms are mainly used for mathematical and computer programs, whilst flowcharts can be used to describe all sorts of processes: business, educational, personal and of course algorithms. So flowcharts are often used as a program planning tool to visually organize the step-by-step process of a program. Here are some examples:

Example 1: Print 1 to 20:

Algorithm:

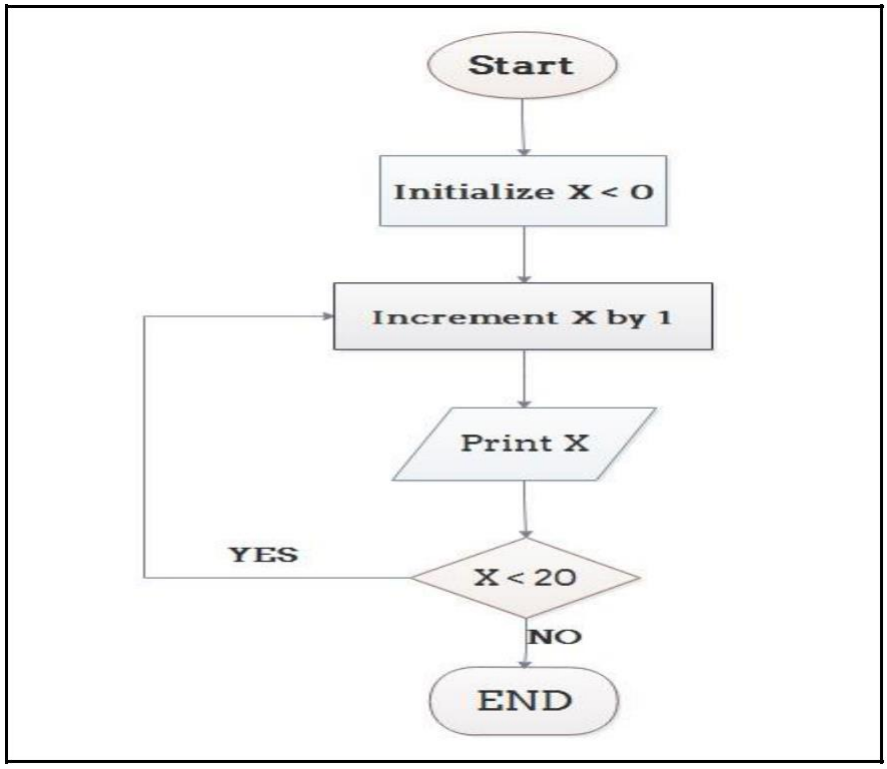
Step 1: Initialize X as 0,

Step 2: Increment X by 1,

Step 3: Print X,

Step 4: If X is less than 20 then go back to step 2.

Flowchart:



Example 2: Convert Temperature from Fahrenheit (℉) to Celsius (℃)

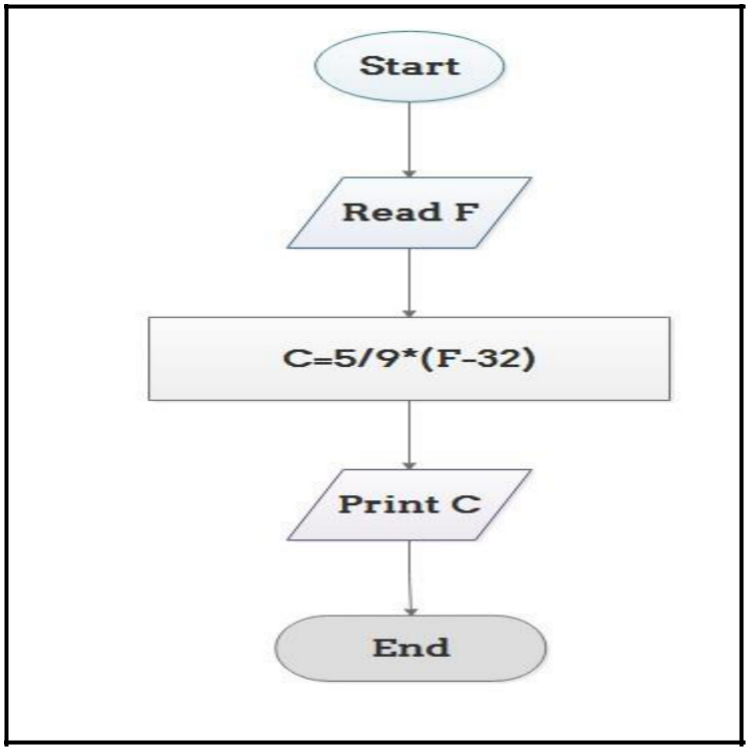
Algorithm:

Step 1: Read temperature in Fahrenheit,

Step 2: Calculate temperature with formula C=5/9*(F-32),

Step 3: Print C,

Flowchart:



3.3.2 Conclusion

From the above we can come to a conclusion that a flowchart is pictorial representation of an algorithm, an algorithm can be expressed and analyzed through a flowchart.
An algorithm shows you every step of reaching the final solution, while a flowchart shows you how to carry out the process by connecting each step. An algorithm uses mainly words to describe the steps while a flowchart uses the help of symbols, shapes and arrows to make the process more logical.

Activity 2: Guided Practice

Task: Mr Serge has a task from his school to compare numbers to find the largest among three different numbers entered by user.

13. Draw flowcharts to find the largest among three different numbers entered by user

Activity 3: Application

Task: Mr. HABUMUGISHA has a work to find all the roots of a quadratic equation $ax^2+bx+c=0$

1. Draw a flowchart to find all the roots of a quadratic equation

Points to Remember

- **Use Consistent Design Elements**
- **Keep Everything on One Page**
- **Flow Data from Left to Right**
- **Use a Split Path Instead of a Traditional Decision Symbol**
- **Place Return Lines Under the Flow Diagram**

Formative Assessment

1. What do you understand by flowchart and list all the standard graphics applied in a flowchart?
2. What is the difference between flowchart and Algorithm?
3. What is the difference between linking and debugging?
4. How can u represent an algorithm into flowcharts?
5. What we mean by source code?
6. Define machine code.

1.3 Revision of basic data types and operators

Topic 1.3: Revision of basic data types and operators

Key Competencies:

Knowledge	Skills	Attitudes
1. Describe elements of data	**1.** Identify elements of data type	**1**. Be a Critical thinker
2. Describe elements of operators	**2.** Identify elements of operators	**2**. Be a detail oriented
3. Describe process of applying data types and operators	3. Implement process of applying data types and operators	**3**. Being organized to achieve the required result.

🔁 Getting Started: What do we know and where are we going?

Task: Form small groups and discuss about these questions:

1.How can you do the addition of the following items

- 10 COWS + 10 FISHES =?
- 4 BOOKS + 2 CARS =?
- 5 HOUSES + 500 Francs =?

2.How can you solve the following Mathematical equations:

- Evaluate [(3 - 5) / (7 x 2)] + 2
- Evaluate 8 – 2 + 7 x 3 – (2 x 6)
- Suppose you are given the equation: y = 7x 3 – 14 and you are asked to solve for x.

3.What do you understand by set in mathematics?

 **Activity 1: Problem Solving**

 **Task:**

SYT ltd is hiring an application developer to develop an attendance application, as a developer you must developer an algorithm to be used in that attendance application, Where the attendance application will receive employee's details, total worked hours of a month and the amount he received per hour. Print the employee's ID and salary (with two decimal places) of a particular month.

Form small groups to discuss about:

1. How they should identify the data types to be used for the required application?
2. What operation to be applied on the data type for the application?
3. How they should apply the data type and operators to perform the requested output?

KEY FACTS 1.3

1. DATA

TYPES 1.1

introduction

Programming uses a number of different data types. A data type determines what type of value an object can have and what operations can be performed.

Data types as its name indicates, a data type represents a type of the data which you can process using your computer program. It can be numeric, alphanumeric, decimal, etc.

Let's keep Computer Programming aside for a while and take an easy example of adding two whole numbers 10 & 20, which can be done simply as follows −

10+20

Let's take another problem where we want to add two decimal numbers 10.50 & 20.50, which will be written as follows −

10.50 + 20.50

The two examples are straightforward. Now let's take another example where we want to record student information in a notebook. Here we would like to record the following information:

Name:

Class:

Section:

Age:

Sex:

Now, let's put one student record as per the given requirement −

Name: Zara Ali

Class: 6th

Section: J

Age: 13

Sex: F

The first example dealt with whole numbers, the second example added two decimal numbers, whereas the third example is dealing with a mix of different data. Let's put it as follows −

- Student name "Zara Ali" is a sequence of characters which is also called a string.
- Student class "6th" has been represented by a mix of whole number and a string of two characters. Such a mix is called alphanumeric.
- Student section has been represented by a single character which is 'J'.
- Student age has been represented by a whole number which is 13.
- Student sex has been represented by a single character which is 'F'.

This way, we realized that in our day-to-day life, we deal with different types of data such as strings, characters, whole numbers (integers), and decimal numbers (floating point numbers).

Similarly, when we write a computer program to process different types of data, we need to specify its type clearly; otherwise the computer does not understand how different operations can be performed on that given data. Different programming languages use different keywords to specify different data types.

For example, C and Java programming languages use **int** to specify integer data, whereas **char** specifies a character data type.

Subsequent chapters will show you how to use different data types in different situations. For now, let's check the important data types available in C, Java, and Python and the keywords we will use to specify those data types.

1.2. List of data type

1.2.1 List of primitive and derived data type in C

Data type	Size	Range	Description
Char	1 byte	-128 to 127	A character
signed char			
unsigned char	1 byte	0 to 255	A character
Short signed short signed short int	2 bytes	−32,767 to 32,767	Short signed integer of minimum 2 bytes
unsigned short	2 bytes	0 to 65,535	Short unsigned integer of minimum 2 bytes
unsigned short int			
Int	2 or 4 bytes	-32,768 to 32,767 or - 2,147,483,648 to 2,147,483,647	An integer (Both positive as well as negative)
signed int			

unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295	An unsigned integer (Positive integer)
Long signed long signed long int	4 bytes	-2,147,483,648 to 2,147,483,647	Long signed integer of minimum 4 bytes
unsigned long unsigned long int	4 bytes	0 to 4,294,967,295	Long unsigned integer of minimum 4 bytes
long long long long int signed long long signed long long int	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Integer with doubled capacity as of long
unsigned long long unsigned long long int	8 bytes	0 to 18,446,744,073,709,551,615	Unsigned integer with doubled capacity as of long
Float	4 bytes	1.2E-38 to 3.4E+38	Single precision floating point number
Double	8 bytes	2.3E-308 to 1.7E+308	Double precision floating point number
long double	12 bytes	3.4E-4932 to 1.1E+4932	Double precision floating point number

Here's a table containing commonly used types in C programming for quick access.

Type	Size (bytes)	Format Specifier
Int	at least 2, usually 4	%d
Char	1	%c
Float	4	%f
Double	8	%lf
short int	2 usually	%hd
unsigned int	at least 2, usually 4	%u
long int	at least 4, usually 8	%li
long long int	at least 8	%lli
unsigned long int	at least 4	%lu
unsigned long long int	at least 8	%llu
signed char	1	%c
unsigned char	1	%c
long double	at least 10, usually 12 or 16	%Lf

1.2.1.1 int

Integers are whole numbers that can have both zero, positive and negative values but no decimal values. For example, 0, -5, 10

We can use int for declaring an integer variable.

```
int id;
```

Here, id is a variable of type integer.

You can declare multiple variables at once in C programming. For example,

```
int id, age;
```

The size of int is usually 4 bytes (32 bits). And, it can take 2^{32} distinct states from 2147483648 to 2147483647.

1.2.1.2 float and double

float and double are used to hold real numbers.

```
float salary;
double price;
```

In C, floating-point numbers can also be represented in exponential. For example,

```
float normalizationFactor = 22.442e2;
```

What's the difference between float and double?

The size of float (single precision float data type) is 4 bytes. And the size of double (double precision float data type) is 8 bytes.

1.2.2.3 char

Keyword char is used for declaring character type variables. For example,

char test = 'h'; The size of the character variable is 1 byte.

1.2.2.4 void

void is an incomplete type. It means "nothing" or "no type". You can think of void as **absent**.

For example, if a function is not returning anything, its return type should be void.

Note that, you cannot create variables of void type.

1.2.1.5 short and long

If you need to use a large number, you can use a type specifier long. Here's how:

```
long a;

long long b;
long double c;
```

Here variables a and b can store integer values. And, c can store a floating-point number.

If you are sure, only a small integer ([−32,767, +32,767] range) will be used, you can use short.

```
short d;
```

You can always check the size of a variable using the sizeof() operator.

```
#include <stdio.h>

int main() {

  short a;

  long b;

  long long c;

  long double d;

  printf("size of short = %d bytes\n", sizeof(a));

  printf("size of long = %d bytes\n", sizeof(b));

  printf("size of long long = %d bytes\n", sizeof(c));

  printf("size of long double= %d bytes\n", sizeof(d));

  return 0;

}
```

1.2.1.6 signed and unsigned

In C, signed and unsigned are type modifiers. You can alter the data storage of a data type by using them. For example,

```
1. unsigned int x;

2.  int y;
```

Here, the variable x can hold only zero and positive values because we have used the unsigned modifier.

Considering the size of int is 4 bytes, variable y can hold values from -2^{31} to $2^{31}-1$, whereas variable x can hold values from 0 to $2^{32}-1$.

Other data types defined in C programming are:

- bool Type
- Enumerated type
- Complex types

1.3.Derived Data Types

Data types that are derived from fundamental data types are derived types. For example:

arrays, pointers, function types, structures, etc.

1.4. User Defined Data Types in C

These type of data type are being define by the user before using them.

A. Typedef:

Typedef, an abbreviation for type definition is a user-defined data type.
Which means, it defines an identifier that can represent an existing data type.
This data type increases the readability of codes with greater complexity.

Example:

```
typedef int numbers;
numbers a=1;
```

The above code states that numbers can be used to declare variables of type int.

B. Enumerated:

It is another user-defined data type which does the job of creating a data type that can be assigned a value from a specific set of values. It is declared by using the keyword 'enum'.

Example:

```
enum prime (2,3,5,7,11,13,17,19); enum prime a,b; a=5; b=19;
```

2. VARIABLE

A variable is a name assign to a storage area that the program can manipulate. A variable type determines the size and layout of the variable's memory.

It also determines the range of values which need to be stored inside that memory and nature of operations that can be applied to that variable.

2.1 Variables in programming

The variable and its counterpart – the constant, now play a huge part in computer programming. Since computer programming plays a fundamental role on the Internet, it is the area where we could all sense the importance of both variables and constants.

Variables empower developers to quickly create a variety of simple and complex programs that tell the computer to behave in a pre-defined manner. Each program contains a list of variables and a list of directions where the variables are used to represent changeable data. The directions provide instructions how the variables should be used by the computer. The greatest advantage of the variables is that they enable one and the same program to execute various sets of data.

In the light of the afore-stated, a variable refers to a symbol for a varying value, which is stored in the system's memory. Each variable has a name (variable name) and a data type, which indicates what type of value it represents. The represented data can contain both numerical (including integers and floating-point numbers) and non-numerical values, such as letters and the '_' symbol.

The process of specifying and modifying variables in a computer program is called variable manipulation. It is mainly used for generating dynamic content through HTML (Hyper Text Mark-up Language) and other widely used programming languages. Some languages have strict data type requirements.

3. OPERATORS

Operator in a programming language is a symbol that tells the compiler or interpreter to perform specific mathematical, relational or logical operation and produce final result. This chapter will explain the concept of operators and it will take you through the important arithmetic and relational operators available in C

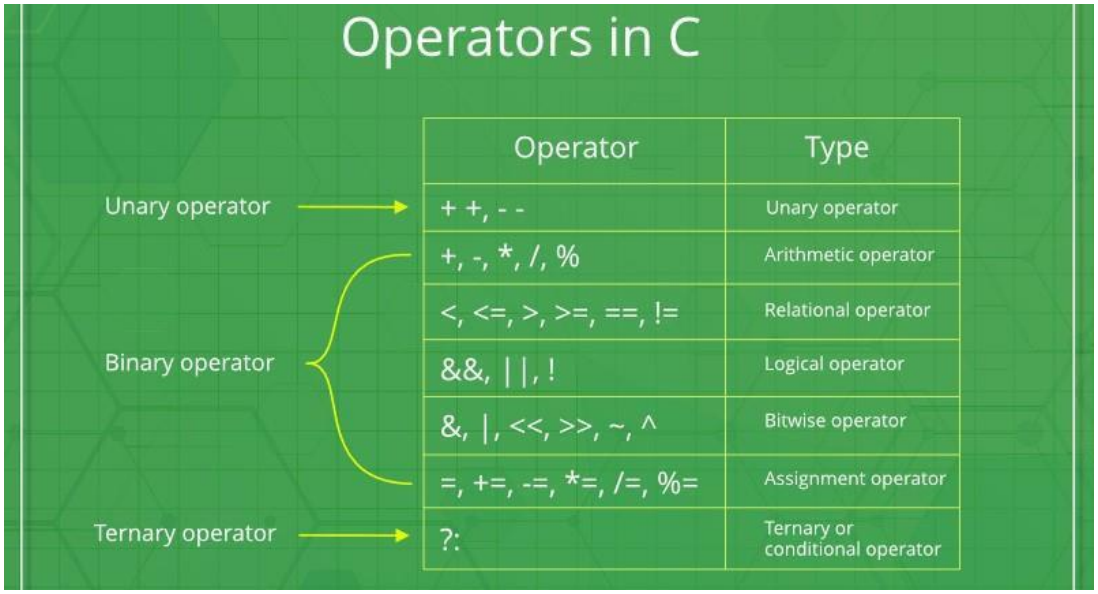
3.1 Operators in C

Operators are the foundation of any programming language. Thus the functionality of C/C++ programming language is incomplete without the use of operators. We can define operators as symbols that help us to perform specific mathematical and logical computations on operands. In other words we can say that an operator operates the operands.

For example, consider the below

```
statement: c = a + b;
```

Here, '+' is the operator known as *addition operator* and 'a' and 'b' are operands. The addition operator tells the compiler to add both of the operands 'a' and 'b'.



C/C++ has many built-in operator types and they can be classified as:

1. **Arithmetic Operators**: These are the operators used to perform arithmetic/mathematical operations on operands. Examples: (+, -, *, /, %,++,–).

Arithmetic operator are of two types:

I. **Unary Operators**: Operators that operates or works with a single operand are unary operators. For example: (++ , –)

II. **Binary Operators**: Operators that operates or works with two operands are binary operators. For example: (+ , – , * , /)

2. **Assignment Operators**: Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value. The value on the right side must be of the same data-type of variable on the left side otherwise the compiler will raise an error.
   Different types of assignment operators are shown below:

- **“=”**: This is the simplest assignment operator. This operator is used to assign the value on the right to the variable on the left.

  For example:

```
a = 10;
b = 20;
ch = 'y';
```

- **“+=”**: This operator is combination of ‘+’ and ‘=’ operators. This operator first adds the current value of the variable on left to the value on right and then assigns the result to the variable on the left.

  Example:

```
(a += b) can be written as (a = a + b)
```

  If initially value stored in a is 5. Then (a += 6) = 11.

- **“-=”**:This operator is combination of ‘-‘ and ‘=’ operators. This operator first subtracts the value on right from the current value of the variable on left and then assigns the result to the variable on the left.

  Example:

```
(a -= b) can be written as (a = a - b)
```

  If initially value stored in a is 8. Then (a -= 6) = 2.

- **“*=”**:This operator is combination of ‘*’ and ‘=’ operators. This operator first multiplies the current value of the variable on left to the value on right and then assigns the result to the variable on the left.

  Example:

```
(a *= b) can be written as (a = a * b)
```

  If initially value stored in a is 5. Then (a *= 6) = 30.

- **“/=”**:This operator is combination of ‘/’ and ‘=’ operators. This operator first divides the current value of the variable on left by the value on right and then assigns the result to the variable on the left.

  Example:

- (a /= b) can be written as (a = a / b)

If initially value stored in a is 6. Then (a /= 2) = 3.

3. **Relational Operators**: Relational operators are used for comparison of the values of two operands. For example: checking if one operand is equal to the other operand or not, an operand is greater than the other operand or not etc. Some of the relational operators are (==, >= , <=).

4. **Logical Operators**: Logical Operators are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operator is a boolean value either true or false.

5. **Bitwise Operators**: The Bitwise operators is used to perform bit-level operations on the operands. The operators are first converted to bit-level and then calculation is performed on the operands. The mathematical operations such as addition , subtraction , multiplication etc. can be performed at bit-level for faster processing.

6. **Other Operators**: Apart from the above operators there are some other operators available in C or C++ used to perform some specific task. Some of them are discussed here:

- **sizeof operator**: sizeof is a much used in the C/C++ programming language. It is a compile time unary operator which can be used to compute the size of its operand. The result of sizeof is of unsigned integral type which is usually denoted by size_t.
  Basically, sizeof operator is used to compute the size of the variable.

- **Comma Operator**: The comma operator (represented by the token **,**) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type). The comma operator has the lowest precedence of any C operator. Comma acts as both operator and separator.

- **Conditional Operator**: Conditional operator is of the form *Expression1 ? Expression2 : Expression3* . Here, Expression1 is the condition to be evaluated. If the condition (Expression1) is *True* then we will execute and return the result of Expression2 otherwise if the condition(Expression1) is *false* then we will execute and return the result of Expression3. We may replace the use of if..else statements by conditional operators.

3.2 Operator precedence chart

The below table describes the precedence order and associativity of operators in C / C++ .

Precedence of operator decreases from top to bottom.

OPERATOR	DESCRIPTION	ASSOCIATIVITY
()	Parentheses (function call)	left-to-right
[]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++/–	Postfix increment/decrement	
++/–	Prefix increment/decrement	right-to-left
+/-	Unary plus/minus	
!~	Logical negation/bitwise complement	
(type)	Cast (convert value to temporary value of type)	
*	Dereference	
&	Address (of operand)	
Sizeof	Determine size in bytes on this implementation	
*,/,%	Multiplication/division/modulus	left-to-right
+/-	Addition/subtraction	left-to-right
<<,>>	Bitwise shift left, Bitwise shift right	left-to-right
<,<=	Relational less than/less than or equal to	left-to-right
>,>=	Relational greater than/greater than or equal to	left-to-right

==,!=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
?:	Ternary conditional	right-to-left
=	Assignment	right-to-left
+=,-=	Addition/subtraction assignment	
*=,/=	Multiplication/division assignment	
%=,&=	Modulus/bitwise AND assignment	
^=,|=	Bitwise exclusive/inclusive OR assignment	
<>=	Bitwise shift left/right assignment	
,	expression separator	left-to-right

Activity 2: Guided Practice

Task:

Save Primary School is hiring a software developer to develop a registration form to collect all the student bio information of student, as a developer you must develop an algorithm to be used on this form with the appropriate data types and apply operators where needed, this form must include also the academics transcripts for the student

In Small groups, do the following :

1. Elaborate the information needed on that form with their data type
2. Design a transcript form to hold the academic record of the student and specify the data type used and operators
3. Use all operators on the academic transcript

Activity 3: Application

Task:

Ministry of health is hiring a software developer to develop a Mitiweli Software to collect all the activities of patient and doctors, as a developer you must developer an algorithm to be used in this Mituweli software.

Form small groups and do the following:

1. Develop a registration of the patient and doctor in applying the appropriate data type
2. Apply all the data type and operators in those forms
3. Develop a patient card with a table showing the number that he has visited certain hospital

Points to Remember

- Use appropriate data type
- Mention the Size of data type in advance
- Focus on the quality of data required

Formative Assessment

1. What is a Variable?
2. What is the uses of operators?
3. What do you understand by User Defined Data Types

1.4 Basic input-output of algorithm description

Topic 1.4: Basic input-output of algorithm description

Key Competencies:

Knowledge	Skills	Attitudes
1.Describe a read and write function	**1.** Analyze a read and write function	**1.** Be detail oriented
2.Explain a read and write function	**2.**Illustrate a read and write function	**2**. Be Critical thinker
3. Interpret a read and write function	**3.**Apply a read and write function	**3.** Be Critical thinker

Getting Started: What do we know and where are we going?

Task: Form small groups of 3 and observe carefully this image. Discuss about it.

a) What do you think about the picture?

b) What is the relationship between given picture with the topic?

Activity 1: Problem Solving

Task: Form small groups of 4, observe the image below and read carefully the scenario below.

Discuss about it.

Shema is a new unexperienced staff recruited as accountant in a business of supermarket.

His boss assigned him a task of preparing a daily report of sold products and their income.

He has a problem to know where to start from

After reading the the text above, discuss to the questions below:

a) What data must be recorded by the accountant?
b) Which calculations the accountant must do on the entered data?
c) What data will is produced and presented to his boss in the report?

Key facts 1.4

Basic input –output of algorithm description

To write a logical step-by-step method to solve the problem is called algorithm, in other words, an algorithm is a procedure for solving problems. In order to solve a mathematical or computer problem, this is the first step of the procedure. An algorithm includes calculations, reasoning and data processing.

1. Read function (input)

A read function is a function which is used for inputs. It helps to receive the value entered by a user and assign it to a variable.

Syntax of read function

Read(variable)

Example:

An algorithm which receives a number entered by a user.

Answer:

S

t

a

r

t

v

a

r

A

a

s

I

n

t

e

g

e

r

Read(A)

End

Explanation:

A is a variable of integer data type

Read() is a function for input of value into variable A

2. Write function

Write function is used for Inputs; it displays the content of a variable OR displays text messages.

Syntax of write function:

write(variable) OR write(“text message”)

Example:

An algorithm which displays a value stored in a variable.

Answer:

S

t

a

r

t

v

a

r

B

a

s

I

n

t

e

g

e

r

B←5
write(“The content of the variable is: ”) //displaying text

message write(B) //displaying text message //displaying

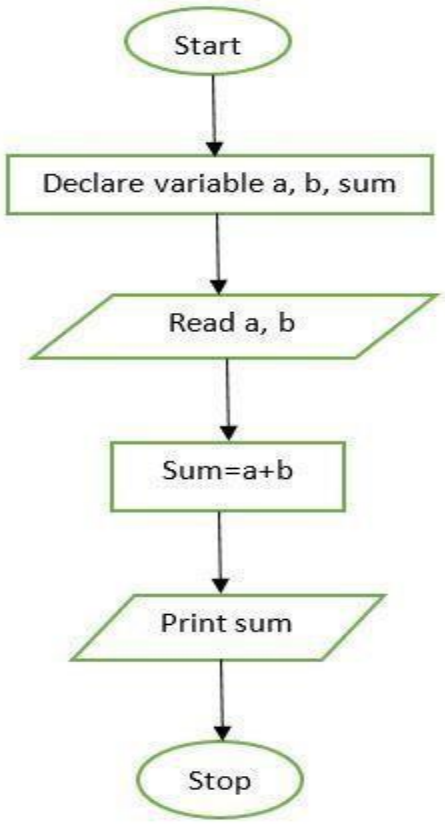
variable’s value

End

Explanation:

B is a variable of integer data type
Write() is a function for write of value(‘5’) into variable B

1.3. Flowchart of read function and write function



- The flow chart above describe a simple program to add two numbers
- The read function read value for a and b with their data types
- The write function print or displays the value of a and b and assign the sum of those values to the variable sum that was declared.
- The flow chart also print sum

Activity 2: Guided Practice

Task: Form groups of 4 and discuss the questions below:

1) A program that takes hours and minutes as input, calculates and displays the total number of minutes.
   a) Write its algorithm pseudocodes
   b) Analyze it and complete the following table

Needed data for input	Process or calculation	Output

c) Illustrate it with a flowchart

2) Compile the program above using C language

Activity 3: Application

Task: This is an indidual work

Write an algorithm pseudocodes and C codes of a:

1) program that asks the user to type 3 integers and writes (displays) the average of the 3 integers.
2) program that displays your name and your age
3) program that prints the perimeter of a rectangle to take its height and width as input.

Points to Remember

- **Entered data as input can be displayed after calculation or not**
- **A value or data entered into a variable, that variable must be declared before it is used**

Formative Assessment

Question 1: What will be the output of the following program?

```
#
includ
e
<stdio
.h>
main(
)
{
  printf("
Main
Menu");
printf("\n 1.
Kenya");
printf("\n 2.
Rwanda");
```

```
printf("\n 3.
Tanzania");

printf("\n 4.
Uganda");
            printf("\n 5. Exit");
}
```

Question 2: In a flowchart an input or output instruction is represented by

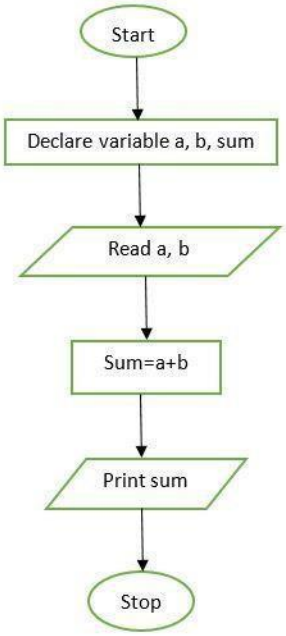
a) A rectangle

b) A rhombus

c) A parallelogram

d) A circle

Question 3: Predict output of following program



If the value of a=20, b=3 what will be the value of the sum.

Question 4: Other keywords to display in algorithm pseudocodes

a) Print, do, display

b) Print or write or display

c) Print or output or write

d) Print or read or output

Self-Reflection

Areas of strength	Areas for improvement	Actions to be taken to improve
1. **2.**	**1.** **2.**	**1.** **2.**

UNIT 2: APPLY PROGRAMMING STRUCTURES, ITERATIVE CONSTRUCTS AND STRUCTURED PROGRAMMING

Illustration of Learning Unit

Draw a standing camera, two cars moving on the road, one driven by a woman and another by a man talking on phone while driving, not wearing a car seatbelt and has exceeded speed, where it has been flashed by a camera plugged on the right side of the road checking speed above 60 km/h.

Topics

2.1. Use of conditional statements in programming

2.2. Use of sequential statements in programming

2.3. Use of iterative statements in programming

Unit Summary:

This unit describes knowledge, Attitudes and skills required to use conditional statements, sequential statements and iterative statements in programming

Topic 2.1: Use of conditional statements in programming
Key Competencies:

Knowledge	Skills	Attitudes
1. Describe statements	**1.** Identify conditional statements	**1.** Be detail oriented
2.Explain statement conditional	**2.** Illustrate conditional	**2**. Be Critical thinker statements
3. Describe the conditional application	**3.** Apply conditional process of statement	**3.** Be Critical thinker statements

Getting Started: What do we know and where are we going?

Task: Form small groups of 2 and observe carefully this image. Discuss about it.

c) What do you think about the picture?

d) What is the relationship between the picture and the topic?

Activity 1: Problem Solving

Task: Form small groups of 4 and read carefully the scenario below. Discuss about it.

At Rwanda National library, librarian records all daily transactions about borrowed and returned books of its subscribed clients. The transactions are recorded on papers. The librarian has a problem manage papers data when searching for returned the book data.

You are requested to disscuss all about these questions :

a) Describe the conditions that can be set to reduce the number of people who delay with books.
b) How can the librarian discover the lost or delayed books?
c) Explain the process of borrowing a book.

Key Facts 2.1

1. USE OF CONDITIONAL STATEMENTS IN PROGRAMMING

Conditional statements help you to make a decision based on certain conditions. Very often when you write code, you want to perform different actions for different decisions. You can use conditional statements in your code to do this.

In all programming languages we have the following conditional statements:

a. If statement or simple IF statement
b. If/else statement
c. Nested if statement
d. If/Else If statement
e. Switch statement (will be seen later)

1.1 Simple IF statement

Use **IF** to specify a block of code (statements) to be executed or a decision made, if a specified condition is true and the block does not execute otherwise.

1.1.1. Syntax

Start

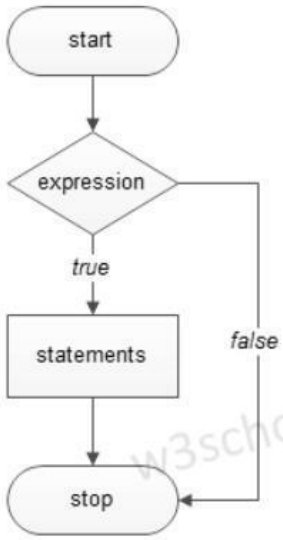
If (condition) **then**

Instructions

End if

End

1.1.2. Flowchart



1.2 IF/ELSE statement

Use **if**/**else** to specify a block of code to be executed, if the same condition is false

1.2.1. Syntax

Start

If (condition) **then**

Instructions

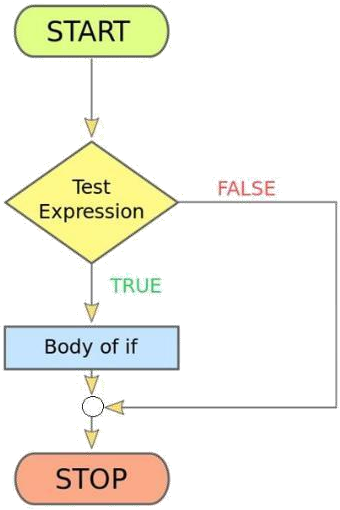
Else

Instructions

End if

End

1.2.2. Flowchart



1.3 ELSE/ IF statement

Use **else/if** to specify a new condition to test, if the first condition is false.

1.3.1. Syntax

Start

If (condition) **then**

Instructions

Else if (condition) **then**

Instructions

Else if (condition) **then**

Instructions

…

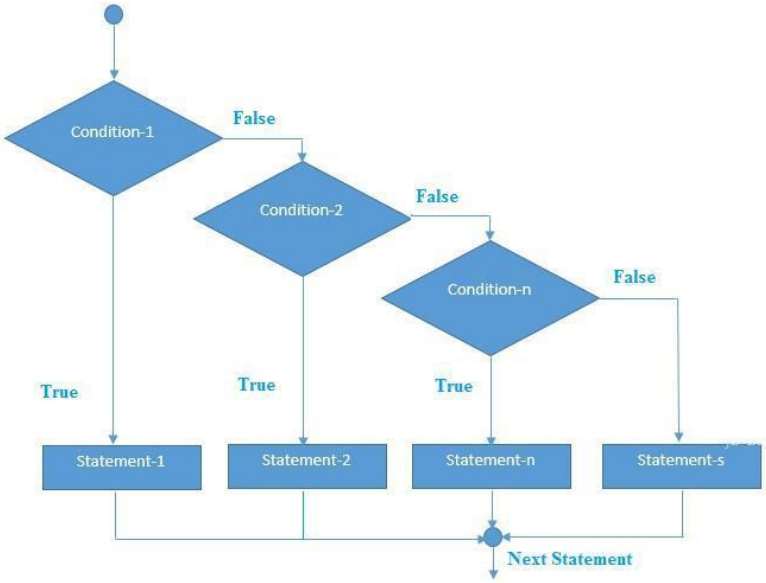
Else [optional]

Instructions

End if

End

1.3.2. Flowchart



1.4 NESTED IF statement

A **NESTED IF** is an IF statement that is the target of another if statement. Nested if statements means an IF statement inside another IF statement.

1.4.1 Syntax

Start

If (condition)

If (condition) **then**

Instructions

…

Else

Instructions

End if

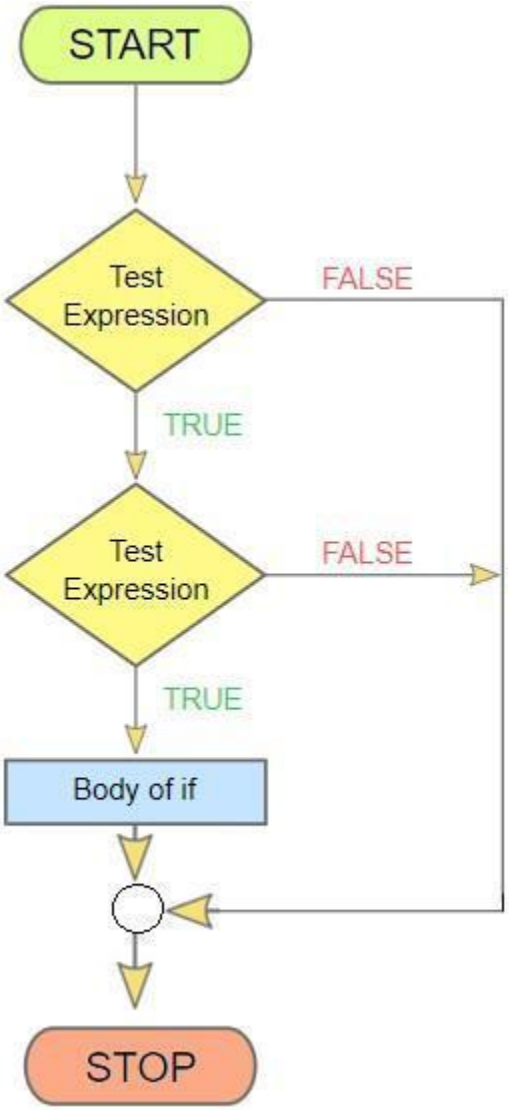
Else [optional]

Instructions

End if

End

1.4.2 Flowchart



Activity 2: Guided Practice

Task: Form groups of 4 and discuss the questions below:

1) Write algorithm pseudocodes the following programs:

a. A program that checks the age and display a decision if someone is eligible to be hired.

b. A program that checks the age and displays a decision if someone is eligible to be hired or not.

c. A program that takes or read three numbers from the user and print the greatest number.

d. A program that helps a trainer to grade students according to their percentage marks results. Follow the table below:

Results %	Grade
0-10	F
11-30	E
31-50	D
51-70	C
71-90	B
91-100	A

2) Illustrate each of the programs above with a flowchart

3) Identify the type of a conditional statement used in each of the programs above(in question 1)

Activity 3: Application

Task: This is an indidual exercise, write algorithms that resolve the questions given below:

1) Write algorithm pseudocodes the following programs:

a. A program to solve quadratic equations (use if, else if and else). Constants values must be entered by the user from the keyboard.

b. A program that records an integer between 1 and 7 from the user and displays the name of the weekday.

c. A program that takes a year from user and print whether that year is a leap year or not.

d. A program to input electricity unit charges and calculate total electricity bill according to the given condition:

For	first	50	units	Rwfs	0.50/unit
For	next	100	units	Rwfs	0.75/unit

For	next Rwfs	100	units	1.20/unit
For	unit Rwfs	above	250	1.50/unit

2) Illustrate each of the programs above with a flowchart
3) Identify the type of a conditional statement used in each of the programs above(in question 1)

Points to Remember

- **Without conditional statements, programs would be unable to react to any changes.**
- **Insist on writing programs pseudocodes in parallel with flowchart**
- **If/else if statement can be replaced by switch case statement**
- **Be careful of use of logical and comparison operators in conditions**

Formative Assessment

Question 1:

a) What will be the output of following program?

```
Start
  var a 1 as character
   if(a != 0)

Wri
te("
Hell
o")
else
      Write ("Bye")
  End if
End
```

b) What is the type of IF statement used?
c) Rewrite this program using a flowchart

Question 2:

a) What is the output of the given code?

Start
var num□10 as integer
if(num>7)then

num□num+2*5

Write(num)

End if

Write("You're right.")

End

i.

52

Y

ou'

re

rig

ht.

ii.

Y

ou'

re

rig

ht.

iii.

60

Y

ou'

re

rig

ht.

iv.

nu

m

10

b) What is the type of IF statement used?
c) Rewrite this program using a flowchart

Question 3: Analyze the expression provided in the first column and complete the second column of the table below.
Assume that a=5, b=10, c=15 and d=0.

EXPRESSION	**TRUE OR FALSE**
(c == a+b)	
(a != 7)	
(b <=a)	
! ((a+d) >= (c-b+1)) (d/a < c*b)	
((c>=15)||(d>a))	
((c>b)&&(b>a))	

Topic 2.2: Use of conditional sequential statement in programming

Key Competencies:

Knowledge	**Skills**	**Attitudes**
1. Describe the cases and statements of a switch	**1.** Differentiate the cases and statements of a switch	**1.** Be detail oriented
	2. Illustrate switch case statement	**2.** Be Critical thinker
2. Interpret switch case statement		
3. Explain the process of switch case statement application	**3.** Apply switch case statements	**3.** Be Critical thinker

Getting Started: What do we know and where are we going?

Task: Form small groups of 3 and observe carefully this image. Discuss about it.

a) What do you think about the picture?
b) What is the relationship between the picture and the topic?

Activity 1: Problem Solving

Task: Form groups of 4 and discuss on the following scenario:

Ramos is a new visitor who has just arrived in Rwanda. She is interested in visiting different touristic places but she has a problem to know all the expenses she has to paid based on her hollidays budget.

You are requested to disscuss all about these questions :

d) Identify the information that Ramos can search for to calculate visiting expenses
e) How can you help her to know the least expensive touristic place?

Key Facts 2.2

1. SEQUENTIAL STATEMENT IN PROGRAMMING

1.1. What is a Switch Statement?

A sequential statement or switch case statement or simply switch statement tests the value of a variable and compares it with multiple cases. Once the case match is found, a block of statements associated with that particular case is executed.

Each case in a block of a switch has a different name which is referred to as an identifier. The value provided by the user or initialized is compared with all the cases inside the switch block until the match is found.

1.1. 1. Syntax

A general syntax of how switch-case is implemented is as follows:

```
switch( expression )
{
        case value-1:

    Block-1;

    Break;
        case value-2:
                    Block-2;
                    Break;
        case value-n:

    Block-n;

    Break;
        default:
```

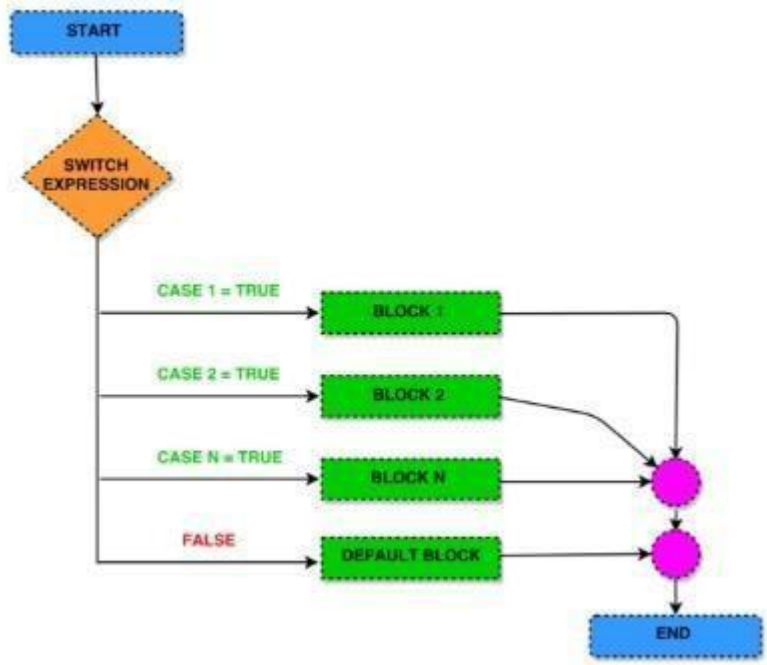
Block-1;

Break;

}

Statement-x;

1.1.2. Flow Chart Diagram of Switch Case:



Activity 2: Guided Practice

Task: Form groups of 4 and discuss the questions below:

Write algorithm pseudocodes and flowchart of the following programs: 1) Write algorithm pseudocodes of a program that accepts a **grade** and displays the equivalent **description**:

Grade	**Description**
E	Excellent
V	Very Good
G	Good
A	Average
F	Fail

2) Illustrate the programs above with a flowchart
3) List the cases along with their statements (in question 1)

Activity 3: Application

Task: This is an indidual exercise, resolve questions given below:

3) Write a program that will ask the user to enter two integer numbers and a simple 4-cases calculator program that might prompt the user by providing the following menu:

Please enter:

+ to add two numbers.

- to subtract two numbers.

* to multiple two numbers.

/ to divide two numbers.

E to exit the program.

This menu will display the result according to the choice of the user.

4) Illustrate the programs above with a flowchart
5) List the cases along with their statements (in question 1)

Points to Remember

- **In switch case, DEFAULT statement represent ELSE in IF/ELSE IF/ELSE**
- **Each case should have BREAK statement at its end, otherwise the statement of other cases below it will be executed.**
- **A switch statement is usually more efficient than a set of nested ifs because fof it facilitate codes readability**

Formative Assessment

Question 1:

What is the output of the following switch statement program? switchval 3 as integer

```
switch(switchval)
{
case 1:
        write("case 1");
 break;
case 2:
        write("case 2");
break;
case 3:
        write("case 3");
 break;
default:
        write("default output");
}
```

a) case 1
b) case 2
c) case 3
d) default output

Question 2:

What will be the output of the following switch statement?

```
switchval2 as integer
switch(switchval)

{
case1:
case2:
case3:
```

```
write(
“Here
I am”)

break;
defaul
t:
        write(“default output”)
}
```

a) Nothing will be printed
b) error in code
c) Here I am
d) default output

Question 3:

How many cases values in Question 2? List them

Question 4:

Which command in a switch statement is referred to for values that don't meet the criteria? a) return

b) default
c) exit
d) break

Question 5: In a switch statement what must come after every case?

a) Curly braces
b) Read
c) Break

Topic 2.3: Use of iterative statements in programming

Key Competencies:

Knowledge	Skills	Attitudes
1. Describe iterative statements	1. Identify iterative statements	1. Skillful
2. Explain iterative statements	**2.** Illustrate iterative statements	**2.** Detail oriented
3. Interpret iterative statement	**3.** Apply iterative statement	**3.**Critical thinking

Getting Started: What do we know and where are we going?

Task: Form groups of 2 and observe carefully this image. Discuss about it.

a) What do you think about the picture?

b) What is the relationship between the picture and the topic?

Activity 1: Problem Solving

Task: Form small groups of 4 and read carefully the scenario below. Discuss about it.

In a school there is 4500 students. The information about all the students is recorded in a notebook. Recorded STUDENT information is first name, last name, gender, trade and level. The secretary has to provide the following reports to her/his headmaster.

Describe the process that will facilitate the secretary to produce a report of:

a) Information of all girls?
b) Number of students whose trade is SOFTWARE DEVELOPMENT?
c) List of names of boys?
d) Total number of students per trade

Key Facts 2.3

1. ITERATIVE STATEMENTS IN PROGRAMMING

A loop is a sequence that gets executed several times. A complete execution of a sequence is called an iteration of the loop.

In all programming languages we have 3 main types of loops statements as follows:

a) FOR loop statement
b) WHILE loop statement
c) DO WHILE statement

1.1. FOR loop statement

A for-loop has two parts: a header specifying the iteration, and a body which is executed once per iteration. The header often declares an explicit loop counter or loop variable, which allows the body to know which iteration is being executed. For-loops are typically used when the number of iterations is known before entering the loop.

1.1.1. Syntax in algorithm pseudocodes

for (<initialize counter> **to** <end of repetitions desired>) **do**

Instructions or block of instructions

END FOR

1.1.2. Syntax in C for

(initializationStatement; testExpression;

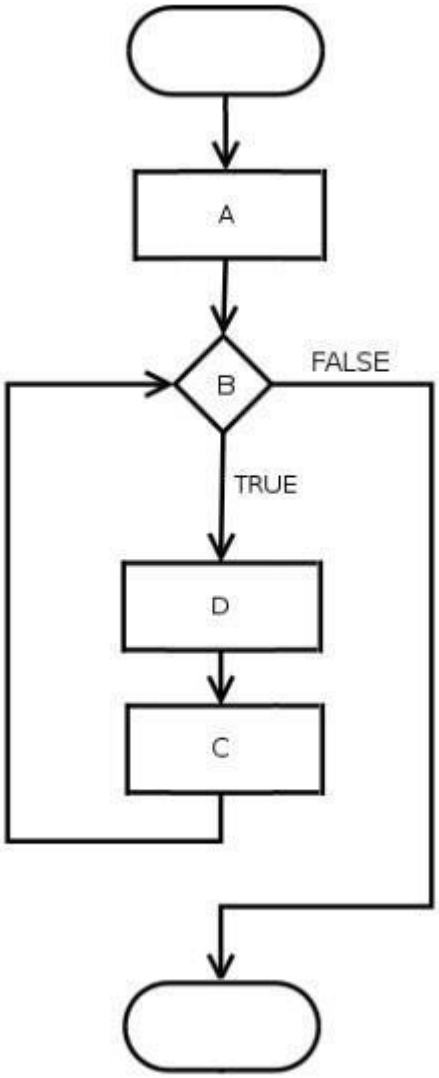
updateStatement)

{

statements inside the body of loop

}

1.1.3. Flowchart



1.2. WHILE loop statement

The *while* construct consists of a block of code and a condition/expression. The condition/expression is evaluated, and if the condition/expression is *true*, the code within the block is executed. This repeats until the condition/expression becomes false.

Because the *while* loop checks the condition/expression before the block is executed, the control structure is often also known as a **pre-test loop**.

1.2.1. Syntax in algorithm pseudocodes:

Variable □start value ***while***

<variable><comparison

operator><end value>

Instruction or block of instructions

Variable □Variable+1

End while

1.2.2. Syntax in C

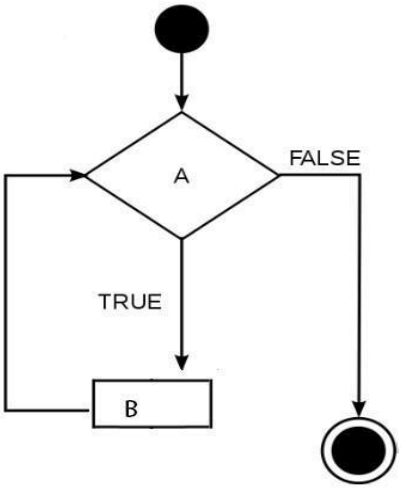
Initial value; while (testExpression)

{

// statements inside the body of the loop

}

1.2.3. Flowchart



1.3. DO WHILE loop statement

In most computer programming languages, a **do while loop** is a control flow statement that executes a block of code at least once, and then repeatedly executes the block, or not, depending on a given boolean condition at the end of the block.

First, the code within the block is executed, and then the condition is evaluated. If the condition is true the code within the block is executed again. This repeats until the condition becomes false.

Because do while loops check the condition after the block is executed, the control structure is often also known as a **post-test loop**.

1.3.1. Syntax in algorithm pseudocodes:

Variable □ start value ***do***

Instruction or block of instructions

Variable □Variable+1 while

<variable><comparison

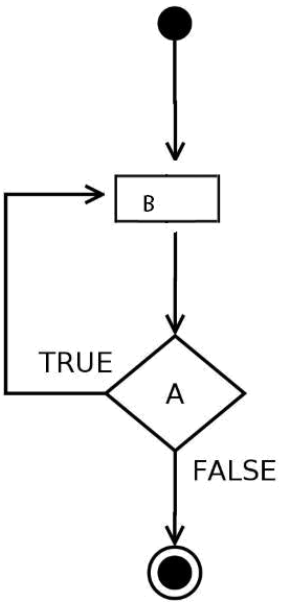
operator><end value>

End do while

1.3.2. Syntax in C

```
Starting value; do
{
  // statements inside the body of the loop
}
while (testExpression);
```

1.3.3. Flowchart



Note: <variable><comparison operator><end value> is a condition

1.4. NESTED LOOP

1.4.1. Types of nested loops

- Nested while loop
- Nested do-while loop
- Nested for loop

1.4.2. Syntax of Nested for loop:

```
for (initialization; condition; increment/decrement)
{
   statement(s);          for (initialization; condition; increment/decrement)
   {
      statement(s);
      ... ... ...
   }
   ... ... ...
}
```

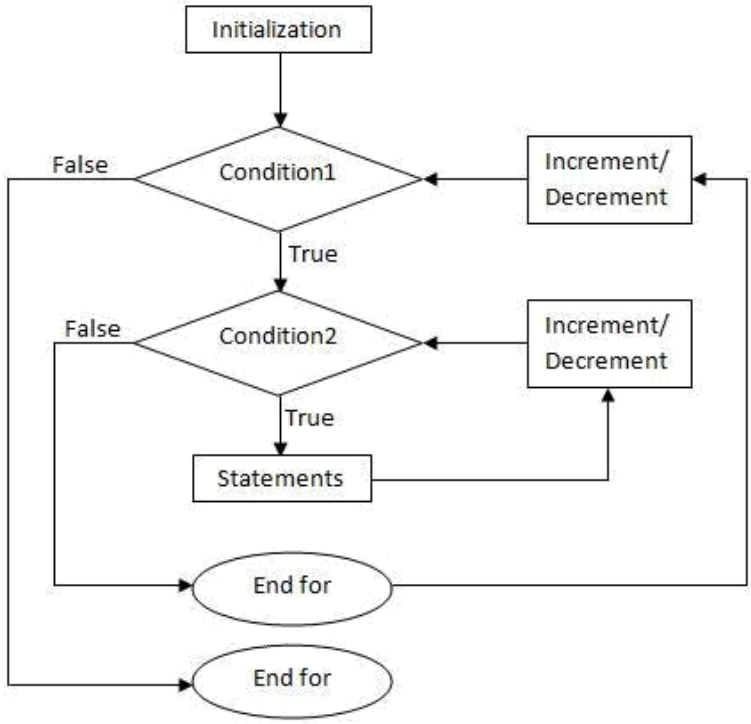


Fig: Flowchart for nested for loop

1.4.3. Syntax of Nested while loop:

```
initialization;
while (condition1)
{
   statement(s);        while (condition2)
   {
      statement(s);
      ... ... ...
   }
   ... ... ...
}
```

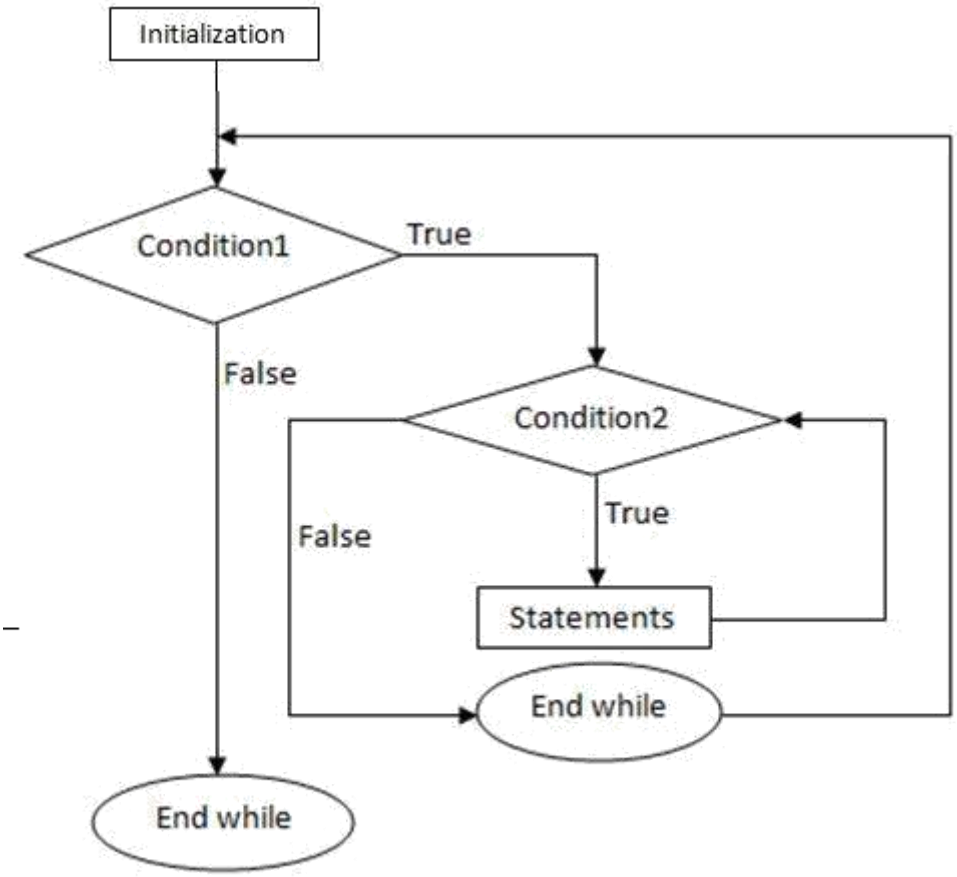


fig: Flowchart for nested while loop

1.4.4. Syntax of Nested do-while loop:

```
initialization;
do
{
   statement(s);
 do
   {
      statement(s);
      ... ... ...
   }while (condition2);
   ... ... ...
}while (condition1);
```

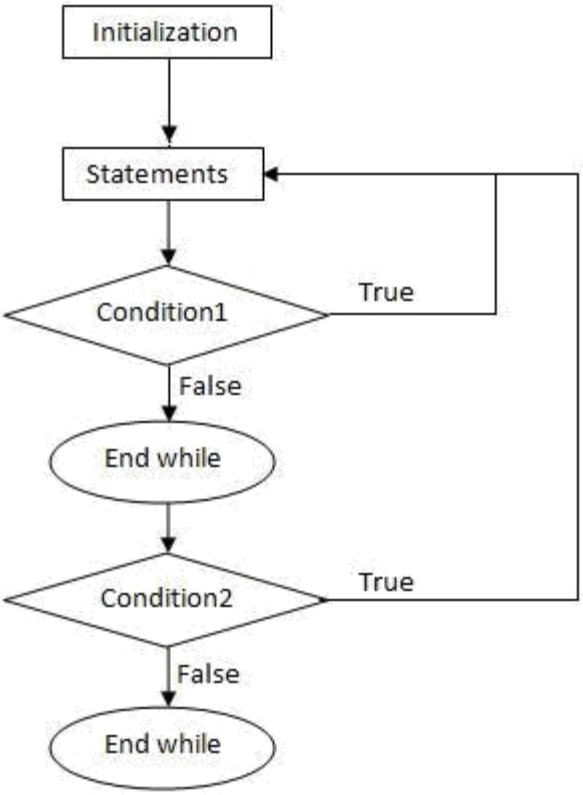


Fig: Flowchart for nested do-while loop

Activity 2: Guided Practice

Task: Form groups of 4 and discuss the questions below::

1. Use a WHILE loop:

a) To write an algorithm pseudocode that displays positive numbers in a range of 20 to 100.

b) Identify the loop parts used – starting value, condition and loop counter

2. Use a FOR loop to write an algorithm pseudocodes program to display integer numbers from 1 to 500, five per line.

3. Use a do while loop to:

a) Display the following triangle made of numbers

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

b) Illustrate with a flowchart the process used to display that triangle

Activity 3: Application

Task: This is an indidual exercise, write algorithms that resolve the questions given below:

1) Use for loop and:

a) Display the following triangle

```
******
 *****
  ****
   ***
    **
     *
```

b) Illustrate the programs above with a flowchart

2) Use a DO WHILE loop to find the factorial of a number using for loop. The number must be entered by the user

3) Illustrate the programs above with a flowchart

4) Use a FOR loop to write a program to produce the first 100 Fibonacci numbers. A Fibonacci number is one that is the product of the prior two generated numbers. The sequence is seeded with the values of 0 and 1, so that the sequence starts off as 0 1 1 2 3 ...

5) Write a program to print the first 100 prime numbers. A prime number any number that can be divided by itself and by 1. Use a do while loop and C language codes.

Points to Remember

- **Be aware of the use of CONTINUE and BREAK in loops**

- **Nested loop is the key for arrays**
- **With Nested loop, the inner is given the priority to execute all its iterations and when it termites, the outer loop changes its counter variable.**

Formative Assessment

Question 1: How many times is a do while loop guaranteed to loop? a) 0
b) Infinitely
c) 1
d) Variable

Question 2: What loops needs a semi colon after? a) for
b) do while
c) while

Question 3: Which for loop is correct?

a) for {i=0,i<10,i++}
b) for (i=0; i<10; i++);
c) for (i=0; i<10; i++)

Question 4: which for loop will not work? Why?
a) for (i=0; i<5; i++)
b) for (i=5; i<=10; i++)
c) for (i=5; i=10; i++)

Question 5: Does every loop require curly braces?
a) true
b) false

Question 6: what is the output of the following program? for(j 10; j > 5; j--) do write(j) end for
a) 10 11 12 13 14 15
b) 9876543210
c) 1098765

d) 109876

Question 7: What must the change be so that the following fragment prints out the even integers 0 2 4 6 8 10?

```
for ( j □0;      j
<=              10;
_______         )
do     write( j);
end for a) j+2
```

b) j□j+2

c) j++++

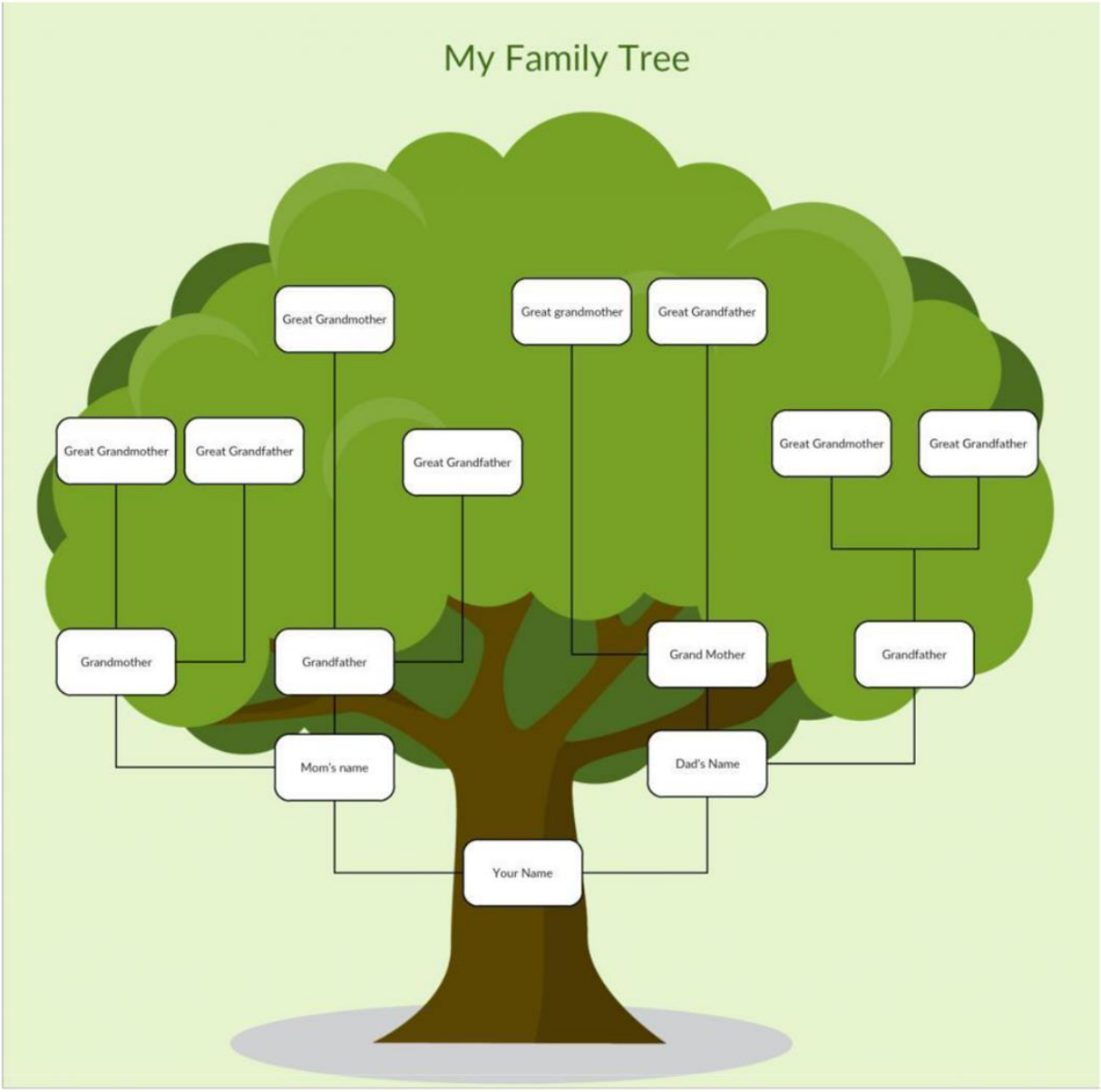
d) ++j++

Self-Reflection

Areas of strength	**Areas for improvement**	**Actions to be taken to improve**
1. **2.**	**1.** **2.**	**1.** **2.**

UNIT 3: DESCRIBE DATA STRUCTURE

Illustration of Learning Unit



Picture brief:

Draw a tree showing the hierarchy of data, it means this tree represent the hierarchy of interconnect event which are linked from the source. on the above you can name it data structure example of family hierarchy.

Topic: 3.1 DESCRIPTION OF LISTS IN DATA STRUCTURE

Unit Summary:

This unit describes skills, knowledge and attitudes required to describe data structure in line with programming, work with array in line with programming, sort and search techniques in line with programming

Topic 3.1: Description of lists in data structure

Key Competencies:

Knowledge	**Skills**	**Attitudes**
1. Describe elements of representing the data structure list	1.identify elements of representing the data structure list	1.Be a Critical thinker
2. Describe concept of manipulating the data structure list	2. Apply concept of manipulating the data structure list	2. Be a detail oriented
3.Describe elements of applying data structure list	Implement elements of applying data structure list	3. Being organized to achieve the required result.

Getting Started: What do we know and where are we going?

Task:

Form small groups and discuss on the following:

1. Have you ever went to cash a cheque in bank?
2. Have you ever used a bus?
3. What do you understand by a queue?

Activity 1: Problem Solving

Task:

Synertech ltd is hiring an application developer to develop a music streaming application, as a developer you must developer a data structure algorithm to be used in that music streaming application, this data structure algorithm will schedule the music flow where the user can play a song and manipulate the song.

Do the following task in groups:

1.Discuss how you will represent the data structure list of required application.

KEY FACTS 3.1

1. DATA STRUCTURE AND ALGORITHMS - LINKED LIST

1.1 Introduction

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.

1.2 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.

1.3 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 th e last element as previous.

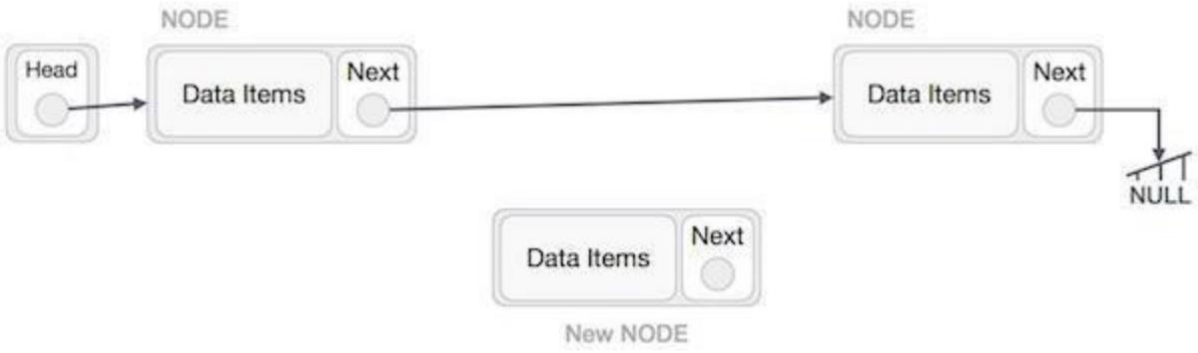
1.4 Basic Operations

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 l ist.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

1.4.1. Insertion Operation

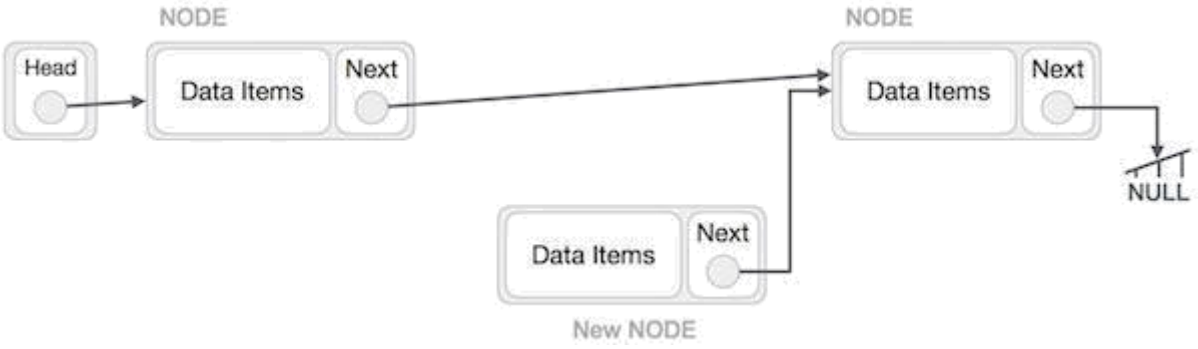
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



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

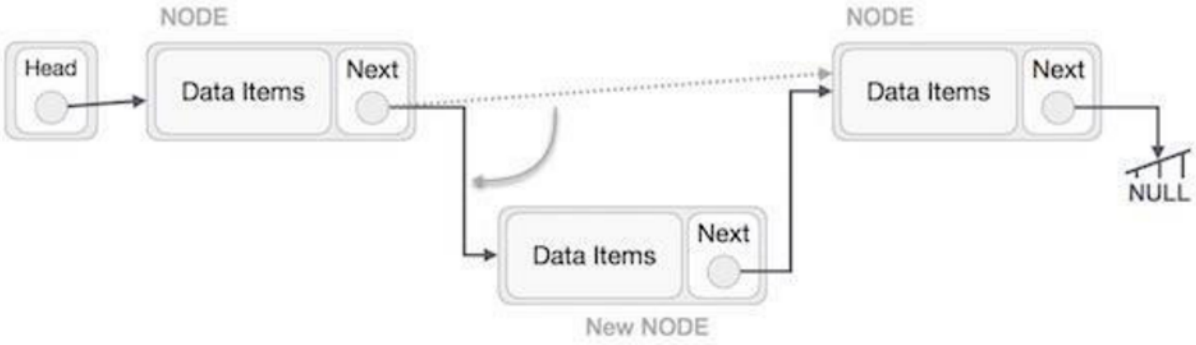
NewNode.next −> RightNode;

It should look like this −



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

LeftNode.next –> NewNode;

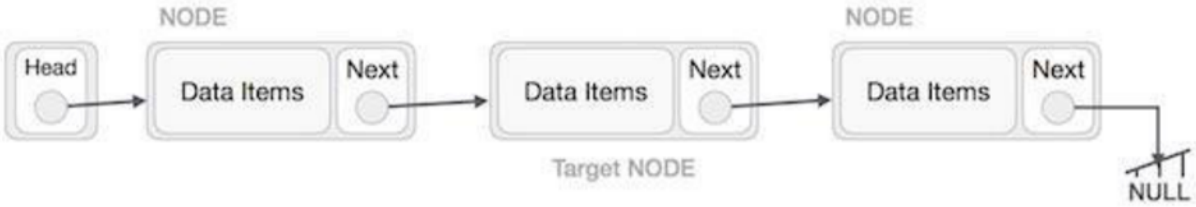


This will put the new node in the middle of the two. The new list should look like this –

Similar steps should be taken if the node is being inserted at the beginning of th e 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.

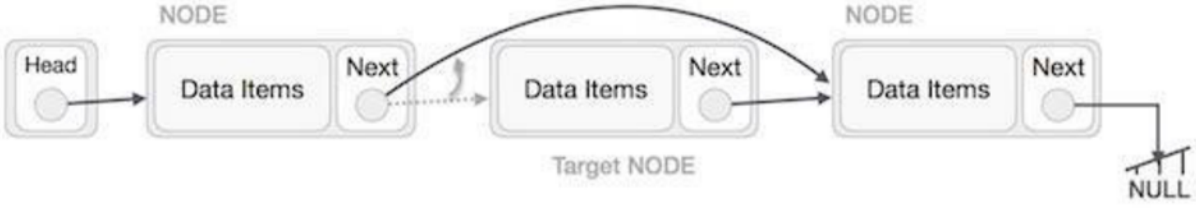
1.4.2 Deletion Operation

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.



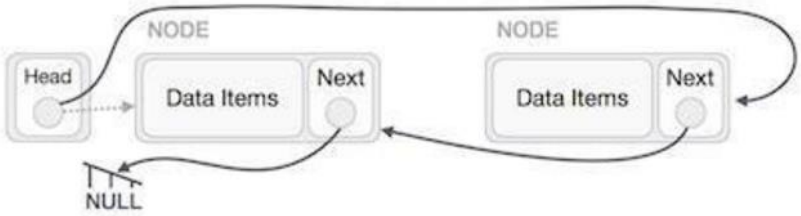
The left (previous) node of the target node now should point to the next node of the target node –

LeftNode.next –> TargetNode.next;

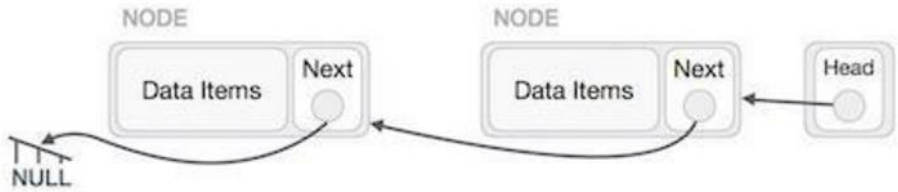


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

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.



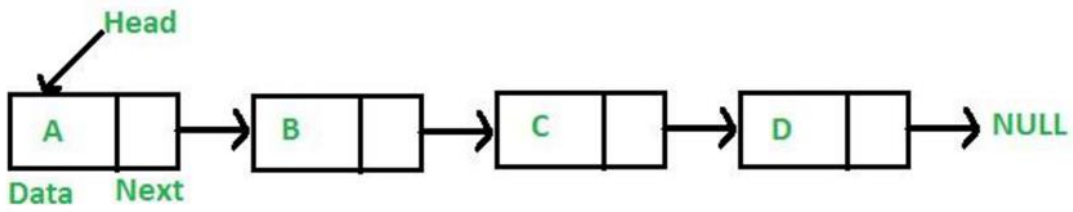
We'll make the head node point to the new first node by using the temp node.



The linked list is now reversed. To see linked list implementation in C programming language, please click here.

1.5 Applications of linked list data structure

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



Applications of linked list in computer science

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

Applications of linked list in real world

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

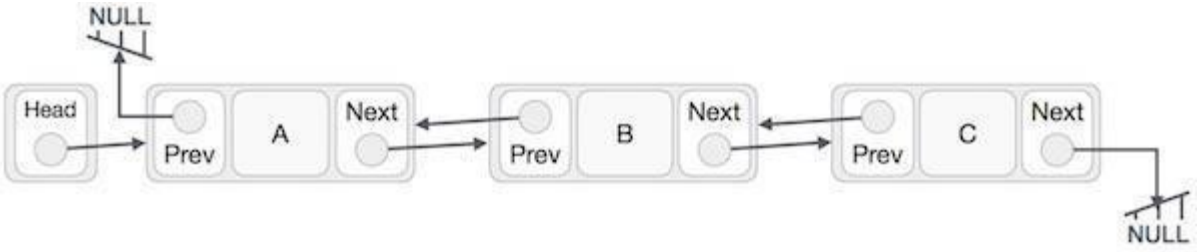
2. Data Structure - Doubly Linked List

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.

2.1 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.

2.2 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.

2.2.1 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;

}
```

2.2.2 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;
}
```

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

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

}
```

2.3 Applications/Uses of doubly linked list in real life

There are various application of doubly linked list in the real world. Some of them can be listed as:
Doubly linked list can be used in navigation systems where both front and back navigation is required.

It is used by browsers to implement backward and forward navigation of visited web pages i.e. **back** and **forward** button.

It is also used by various application to implement **Undo** and **Redo** functionality.

It can also be used to represent deck of cards in games.

It is also used to represent various states of a game.

3. Data Structure - Circular Linked List

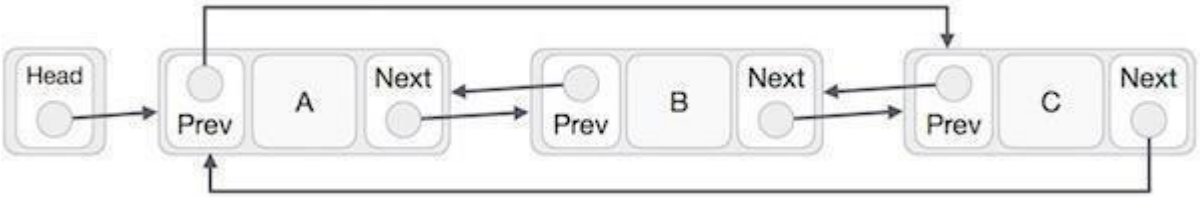
3.1 Introduction

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.

3.2 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.

3.2.1. 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()) {
                                h
  ead =
link;          head->next =
head;                                  link->next =
  } else {
    //point it to old first node
head;
```

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

3.2.2 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;
}
```

3.2.3. 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(" ]");
}
```

3.4 Applications/Uses of Circular linked list in real life

Circular lists are used in applications where the entire list is accessed one-by-one in a loop.

Example: Operating systems may use it to switch between various running applications in a circular loop.

It is also used by Operating system to share time for different users, generally uses RoundRobin time sharing mechanism.

Multiplayer games uses circular list to swap between players in a loop.

Activity 2: Guided Practice

Task:

Photo Studio ltd is hiring an application developer to develop an image viewer to slideshow picture image of the clients, as a developer you must developer a data structure algorithm to be used in that image viewer application, this data structure algorithm will schedule the music flow where the user can manipulate the pictures as needed.

In Small groups, do the followings :

1.Develop a data structure algorithm to add a picture at the end of the pictures to be viewed in that image application

2. Develop a data structure algorithm to add a picture at the beginning of the pictures to be viewed in that image application
3. Develop a data structure algorithm to display all the pictures in that image application

Activity 3: Application

Task:

Rumba Music ltd is hiring an application developer to develop a video streaming application, as a developer you must developer a data structure algorithm to be used in that video streaming application, this data structure algorithm will schedule the videos flow where the user can play a song and manipulate

the videos **Form small groups and** stage **a gathering and analysis of information and do the following:**

1. Apply all the linked list operation on the video streaming application to be developer
2. Represent all the data structure list for the application to be developed
3. Develop a data structure algorithm to delete all the video in that video streaming application

 **Points to Remember**

- Follow the steps while developing the data structure
- Always simplify the algorithm
- Focus on the output required

 **Formative Assessment**

1. Elaborate all the operation in Circular Linked List
2. What is the difference between Linked List and Double Linked List?
3. What are the uses of doubly linked list in real life?

Topic 3.2: Array description

Key Competencies:

Knowledge	**Skills**	**Attitudes**
1.Describe an array	**1.**Examine an array	**1.** Be detail oriented
2.Explain an array	**2.**Illustrate an array	**2**. Be Critical thinker
3. Interpret an array	**3.**Apply an array	**3.** Be Critical thinker

 **Getting Started: What do we know and where are we going?**

 **Task:** Form small groups of 2 and observe carefully this image. Discuss about it.

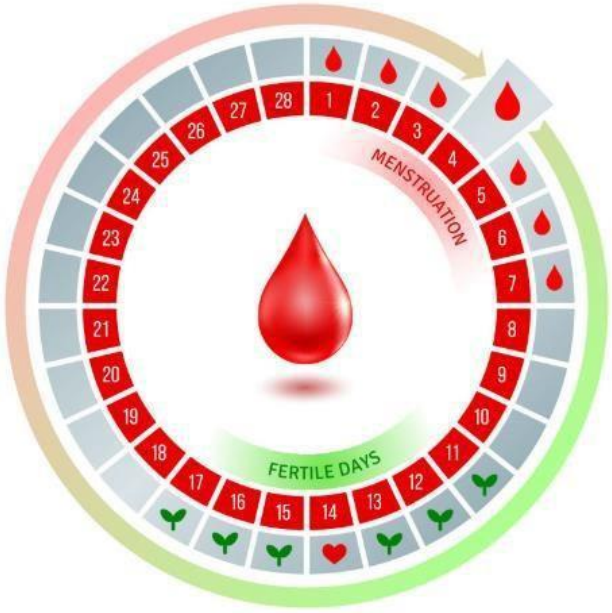
	Monday	Tuesday	Wednesday	Thursday	Friday
9:00 AM	Math	Science	English	Math	English
10:00 AM	(S.E Lecture Theatre)			(S.E Lecture Theatre)	
11:00 AM					
12:00 PM	Lunch	Lunch	Lunch	Lunch	Lunch
1:00 PM	Science	Math	History	History	Break
2:00 PM		(S.E Lecture Theatre)			Science
3:00 PM				English	
4:00 PM		History	Break		
5:00 PM	Sport		Sport		Sport

e) What do you think about the picture?

f) What is the relationship between given picture with the topic?

Activity 1: Problem Solving

Task: Form small groups of 4, observe the image below and read carefully the scenario below. Discuss about it.



Kevine is an ignorant girl about periodical menstruation calculation. She wants to know the exact dates of her menstruation, fertile and non-fertile periods. The last menstruation period started on 30^{th} of last month and her menstruation cycle is of 28 days.

After observing the image above and readng the text below it, discuss to the questions below:

f) What is the next date of her *first day of menstruation* period on calendar?

g) From the question above, draw a table of her next menstruation cycle showing MENSTRUATION period, FERTILE and non-FERTILE days.

h) Draw one table (3x28) of 3 next menstruation cycles highlighting MENSTRUATION period, FERTILE and non-FERTILE days on calendar.

Key Facts 3.2

ARRAY DESCRIPTION

An array is a variable that can store multiple values of the same data type.

Array is used to store data of similar data type. Arrays are used when we want to store data in large quantities, e.g. if we want to store 100 numbers we have to declare 100 variables and remembering the name of each variable is a very a difficult task, here comes the array we can declare a single variable int num[100] now this variable can store 100 different or same numbers and can be accessed by referencing as num[0], num[1], num[2] and so on. So, we can say that to reduce efforts we use arrays.

1. Example where arrays are used

- to store list of Employee or Student names,
- to store marks of students,
- or to store list of numbers or characters etc.

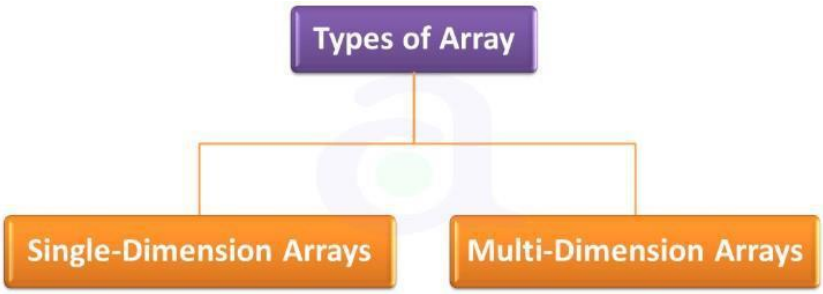
Since arrays provide an easy way to represent data, it is classified amongst the data structures. Other data structures are **structure**, **lists**, **queues**, **trees** etc. Array can be used to represent not only simple list of data but also table of data in two or three dimensions.

2. Why do we need arrays?

We can use normal variables (v1, v2, v3...) when we have a small number of objects, but if we want to store a large number of instances, it becomes difficult to manage them with normal variables. The idea of an array is to represent many instances in one variable.

3. Types of arrays

There are 2 types of arrays.



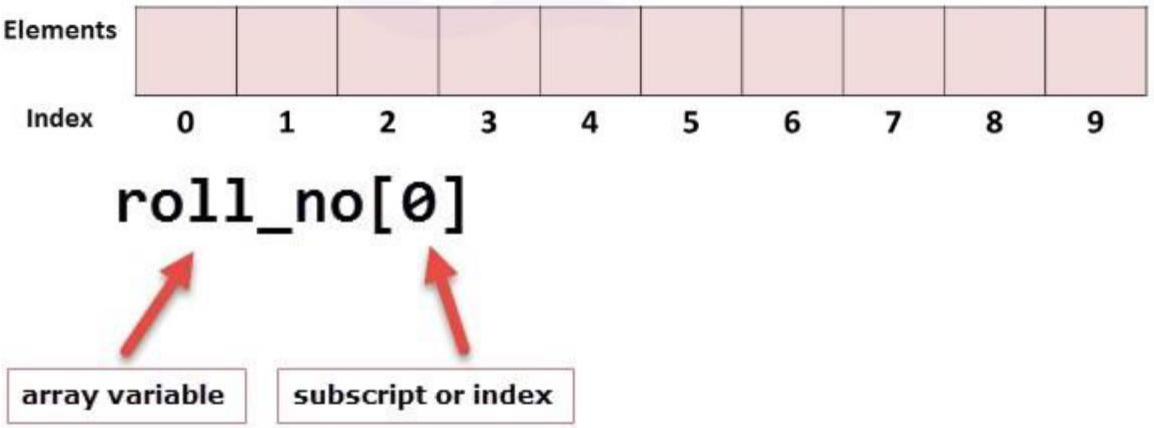
a) One dimensional array

b) Multi-dimensional array

- Two dimensional array
- Three dimensional array
- Four dimensional array etc…

3.1. One dimensional array

A one-dimensional array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript which can either represent a row or column index.



3.1.1. Declaration

We can declare an array by specifying its type and size or by initializing it or by both.

Using algorithm pseudocodes
arr1[10] as integer

// With recent C/C++ versions, we can also // declare an array of user specified size int n 10 int arr2[n]

// Array declaration by initializing elements **arr[] □ { 10, 20, 30, 40 } as integer**

/ Compiler creates an array of size 4.
/ above is same as "int arr[4] = {10, 20, 30, 40}"

/ Array declaration by specifying size and initializing elements **arr[6] □ { 10, 20, 30, 40 } as integer**

/ Compiler creates an array of size 6, initializes first
/ 4 elements as specified by user and rest two elements as 0.
/ above is same as "int arr[] = {10, 20, 30, 40, 0, 0}"

In C

/ Array declaration by specifying size **int arr1[10];**

/ With recent C/C++ versions, we can also // declare an array of user specified size int n = 10;
int arr2[n];

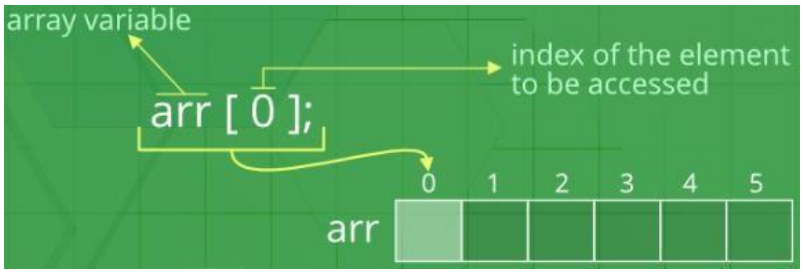
/ *Array declaration by initializing elements* **int arr[] = { 10, 20, 30, 40 }**

/ Compiler creates an array of size 4.
/ above is same as "int arr[4] = {10, 20, 30, 40}"

/ Array declaration by specifying size and initializing elements **int arr[6] = { 10, 20, 30, 40 };**
/ Compiler creates an array of size 6, initializes first
/ 4 elements as specified by user and rest two elements as 0.
/ above is same as "int arr[] = {10, 20, 30, 40, 0, 0}"

3.1.2. Accessing one dimensional array elements

Array elements are accessed by using an integer index. Array index or indice or subscript starts with 0 and goes till size of array minus 1.



Note: Array elements are accessed with the purpose of initialization or input or output.

3.1.2.1. Array Initialization and output syntax

Using algorithm //INITIALIZATION
array_name[size] { value1, value 2, value 3, ... value N } as datatype **pseudocodes** OR
array_name[size] as datatype
array_name[index0] value1 as datatype array_name[index1] value2 as datatype
array_name[indexN] value3 as datatype

// OUTPUT
write(array_name[index0]) write(array_name[index1]) write(array_name[indexN])
OR(using a loop)

for(i 0 as integer i<size i++) do write(array_name[i]) end for

In C datatype array_name[size] = { value 1, value 2, value 3, ... valueN }; OR
datatype array_name[size] ; array_name[index0] = value1;
array_name[index1] = value2; array_name[indexN] = valueN;
// OUTPUT
write(array_name[index0])
printf("format specifier", array_name[index0]); printf("format specifier",
array_name[index1]); printf("format specifier", array_name[indexN]);
OR(using a loop) for(int i=0; i<size; i++) {
printf("format specifier", array_name[i]);
}

4. Dynamically allocated arrays

3.1.2.2. Array input and output

Note: Refer to dynamic allocated arrays

3.1.3. Rules for declaring one dimensional array

- An array variable must be declared before being used in a program.
- The declaration must have a data type (int, float, char, double, etc.), variable name, and subscript.
- The subscript represents the size of the array. If the size is declared as 10, programmers can store 10 elements.
- An array index always starts from 0. For example, if an array variable is declared as s[10], then it ranges from 0 to 9.
- Each array element stored in a separate memory location

3.2. Two dimensional array

An array of arrays is known as 2D array. The two dimensional (2D) array is also known as matrix. A matrix can be represented as a table of rows and columns.

The simplest form of a multidimensional array is the two-dimensional array. Both the row's and column's index begins from 0. x[3][4]:

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Here, x is a two-dimensional (2d) array. The array can hold 12 elements.

Note: x[1][3] is the 7 element in the array which means that 1 is the index representing the second row in the array and 3 is the index representing the forth column in the array.

3.2.1. 2D array declaration

Syntax

Two Dimensional array requires two subscript-index variables. In the following syntax size1 represents number of rows whereas size2 represents number of columns.

Algorithm syntax: **var** array_name **[size1][size2] as data-type**

C language syntax: data_type array_name[rows][columns];

3.2.2. Initialization of 2D Array and Output

There are two ways to initialize a two Dimensional arrays during declaration.

Using algorithm pseudocodes
//INITIALIZATION var arr[2][4] □ {

{10, 11, 12, 13},

{14, 15, 16, 17}

} as integer

OR

var arr[2][4] = { 10, 11, 12, 13, 14, 15, 16, 17} as integer

//OUTPUT

Output with a loop var i, j as integer

for(i□0 i<2 i++)do for(j □0 j<4 j++)do

write(arr[i][j]); end for end for

Output without a loop write (arr[0][0]) write (arr[0][1]) write (arr[0][2]); write (arr[0][3]); write (arr[1][0]); write (arr[1][1]); write (arr[1][2]); write (arr[1][3]);

In C //INITIALIZATION int arr[2][4] = {

{10, 11, 12, 13},

{14, 15, 16, 17}

};

OR

int arr[2][4] = { 10, 11, 12, 13, 14, 15, 16, 17};

//OUTPUT

Output with a loop for(int i=0; i<2; i++){ for(int j=0; j<4;

j++){ printf("%d", arr[i][j]);

}

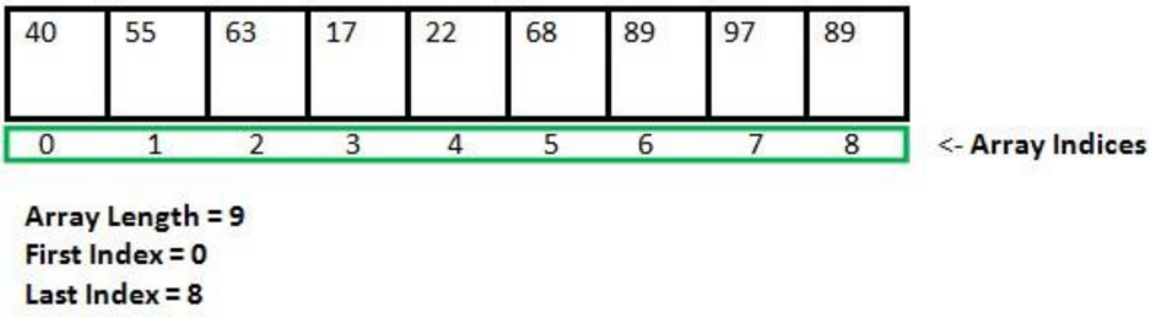
}

	Output without a loop **printf("%d", arr[0][0]);** **printf("%d", arr[0][1]);** **printf("%d", arr[0][2]);** **printf("%d", arr[0][3]);** **printf("%d", arr[1][0]);** **printf("%d", arr[1][1]);** **printf("%d", arr[1][2]);** **printf("%d", arr[1][3]);**

Why Using Multi-Dimensional Array?

Consider a specific example. Suppose that you have an extreme interest in the weather, and you are intent on recording the temperature each day at 2 separate geographical locations throughout the year. After you have sorted out the logistics of actually collecting this information, you can use an array of 2 elements corresponding to the number of locations, where each of these elements in an array of 4 element s to store the temperature values.

4. Dynamically allocated arrays



As it can be seen that the length (size) of the array above made is 9. But what if there is a requirement to change this length (size). For Example, if there is situation where only 5 elements are needed to be entered in this array. In this case the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.

Take another situation. In this there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

This procedure is referred as **Dynamic Memory Allocation**.

Therefore, **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

Unlike a fixed array, where the array size must be fixed at compile time, dynamically allocating an array allows us to choose an array length at runtime.

4.1 malloc ( )

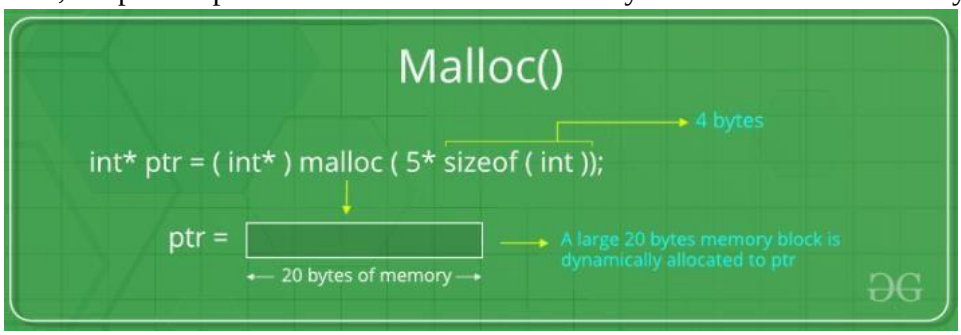
"malloc" or **"memory allocation"** method is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form.

Syntax:

ptr = (cast-type*) malloc(byte-size)

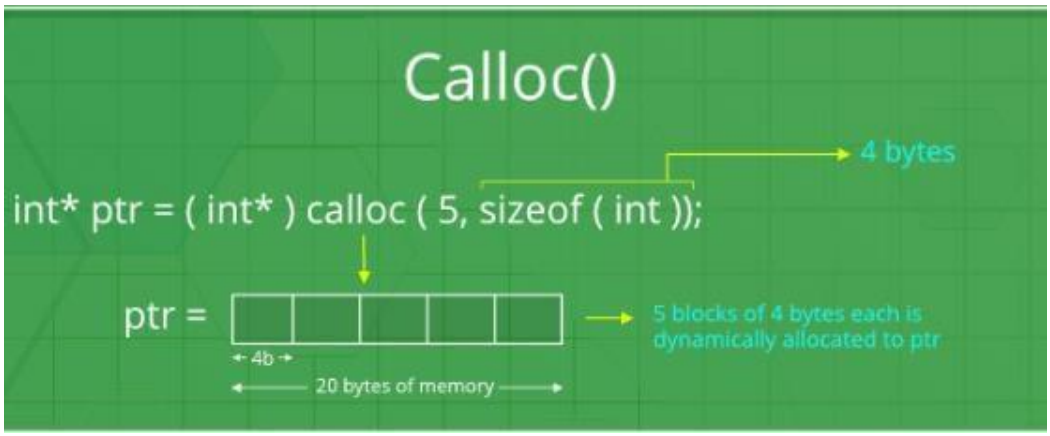
For Example:
ptr = (int*) malloc(100 * sizeof(int));
Since the size of int is 4 bytes,this statement will allocate 400 bytes of memory.

And, the pointer ptr holds the address of the first byte in the allocated memory.



4.2 calloc ( )

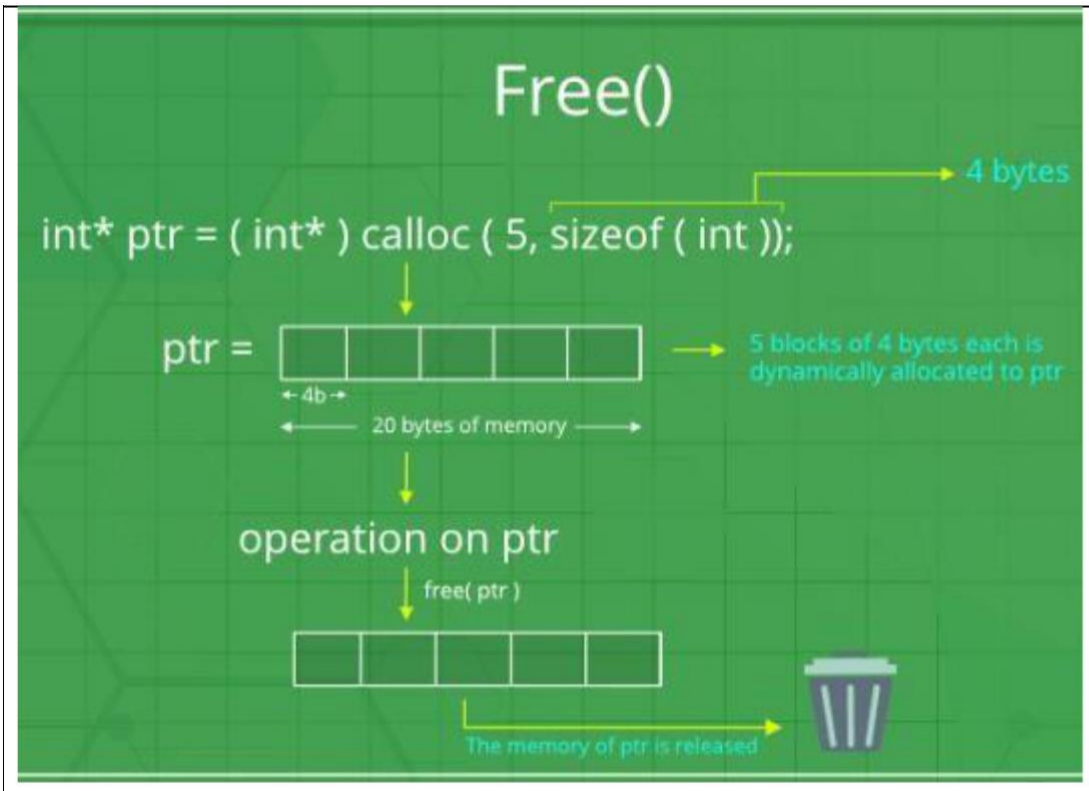
"calloc" or **"contiguous allocation"** method is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value '0'.
Syntax: ptr = (cast-type*)calloc(n, element-size);
For Example:
ptr = (float*) calloc(25, sizeof(float));
This statement allocates contiguous space in memory for 25
elements each with the size of float.



4.3 free()

"free" method is used to dynamically de-allocate the memory. The memory allocated using functions malloc() and calloc() are not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax: free(ptr);



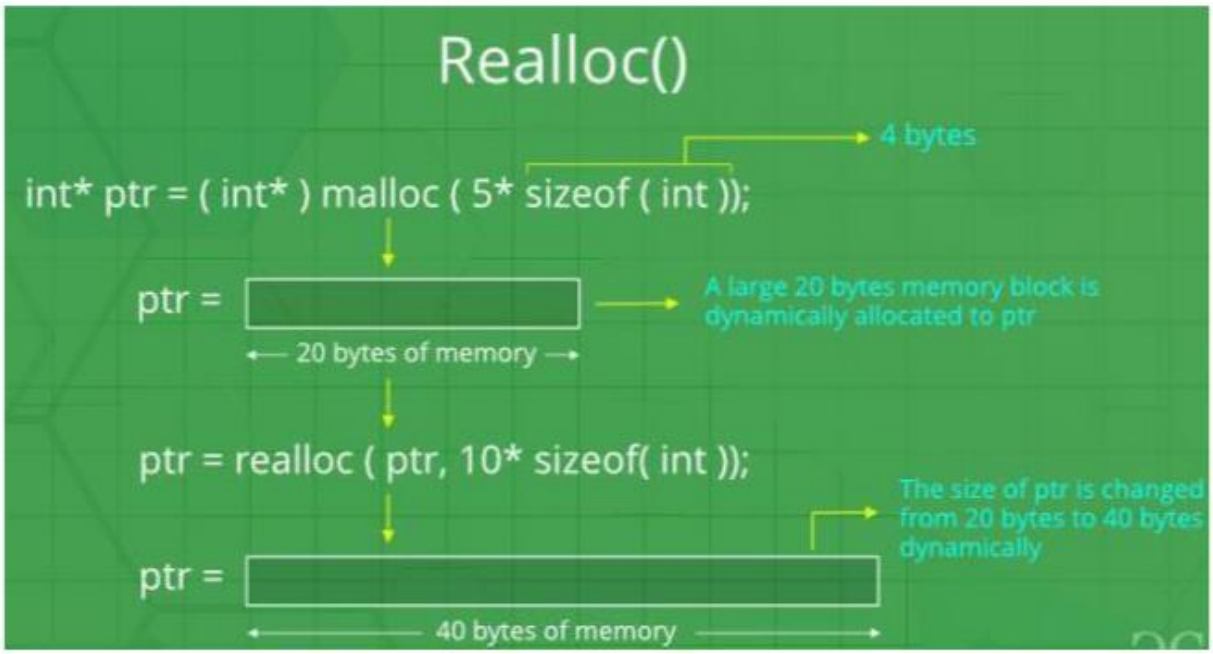
4.3 realloc()

“realloc” or **“re-allocation”** method is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**.

Syntax:

ptr = realloc(ptr, newSize);

where ptr is reallocated with new size 'newSize'.



Activity 2: Guided Practice

Task: Form groups of 4 and discuss the questions below:

6) Write a program in C to store elements in an array and print it.

   Test Data :

   Input 10 elements in the

   array : element - 0 : 1

   element - 1 : 1

   element - 2 : 2

   .....

7) What type of an array is required in the question above?
8) Illustrate the program above with algorithm pseudocodes.

9) Write a program in C for a 2D array of size 3x3 and print the matrix.
Test Data :
Input elements in the matrix :

element - [0],[0] : 1 element
- [0],[1] : 2 element - [0],[2]
: 3 element - [1],[0] : 4
element - [1],[1] : 5 element
- [1],[2] : 6 element - [2],[0]
: 7 element - [2],[1] : 8
element - [2],[2] : 9
Expected Output :
The matrix is :

1 2 3
4 5 6
7 8 9

Activity 3: Application

Task: This is an indidual work

1) Write a program in C to find the sum of all elements of the array.
Test Data :
Input the number of elements to be stored in the array :3 Input 3
elements in the array :
element - 0 : 2 element
- 1 : 5 element - 2 : 8
Expected Output :
Sum of all elements stored in the array is : 15

2) Write a C program to input elements in array and count negative elements in array. C program to find all negative elements in array.
Example
Input
Input array elements: 10, -2, 5, -20, 1, 50, 60, -50, -12, -9
Output
Total number of negative elements: 5

3) By using two-dimensional array, write C program to initialize and display a matrix as shown below:

23	36	99	12	7
11	15	17	17	90
14	32	54	27	50
2	43	6	23	60

Points to Remember

- **A two-dimensional array requires the use of nested loop. Inner loop deals with columns while outer loop deals with rows.**
- **A 2D is popular than other multi-dimensional arrays.**
- **When array is initialized, each set of curly braces represents a row, or a single dimensional array.**

Formative Assessment

Question 1: Predict output of following program

```
#include<stdi
o.h>int
main()
{          int i;       int
arr[5]        =    {1};
for (i = 0; i < 5;
i++){
printf("%d            ",
arr[i]);}          return
0;
}
```

a) 1 followed by four garbage values

b) 10000

c) 11111

d) 00000

Question 2: Examine the following:

double[][] values = { {1.2, 9.0, 3.2},{9.2, 0.5, 1.5, -1.2},{7.3, 7.9, 4.8} } ; What is in values[2][1] ? a) 7.3

b) 7.9

c) 9.2

d) There is no such array element. **Question 3:** Examine the following:

double[][] values = { {1.2, 9.0, 3.2}, {9.2, 0.5, 1.5, -1.2}, {7.3, 7.9, 4.8} } ; What is in values[3][0] ?

a) 7.3

b) 7.9

c) 9.2

d) There is no such array element.

Question 4: How would you get the value 6 out of the following array int[][] a = {{2, 4, 6, 8}, {1, 2, 3, 4}};? a) a [0][3]

b) a [1][3]

c) a[0][2]

d) a[2][0]

e) a[3][1]

Question 5: Which of the following header files must necessarily be included to use dynamic memory allocation functions? a) stdlib.h

b) stdio.h

c) memory.h

d) dos.h

Question 6: Can I increase the size of statically allocated array?

a) Yes

b) No

Question 7: When we dynamically allocate memory is there any way to free memory during run time? a) Yes

b) No

TOPIC 3.3: DESCRIBE DATA STRUCTURE SEARCHING AND SORTING TECHNIQUES

Topic 3.3: Describe data structure searching and sorting techniques

Key Competencies:

Knowledge	Skills	Attitudes

8. Describe stack and queue techniques	**9.** Use stack and queue techniques	**10.** Be Critical thinker
11. Describe searching techniques	**12.** Use searching techniques	**13.** Attentive
14. Describe sorting techniques	**15.** Use sorting techniques	**16.** Team Work spirit

Getting Started: Look the picture below and answer the following questions.

Illustration: Draw a diagram like the above showing someone search for an application to be used and another sorted sequence from A to Z and 1 to 5 … as shown above

E. What do you see on the above diagram?
F. Are they relationship between the picture and the topic?

Activity 1: Problem Solving

Task: NIDA has a task of arange the Rwandan people from the first to the last according to their identity card number, and after arrangement they want to look for information of the following ID number:
1199080070378397 so that they put his ID card to other IDs from his District according to the Name.

1. Describe the sorting techniques that can be used?
2. Describe searching techniques for these issues?
3. Describe steps of stack techniques used to put the ID in its position due to the name?

Key Facts 3.3

1. **Stack & Queue**

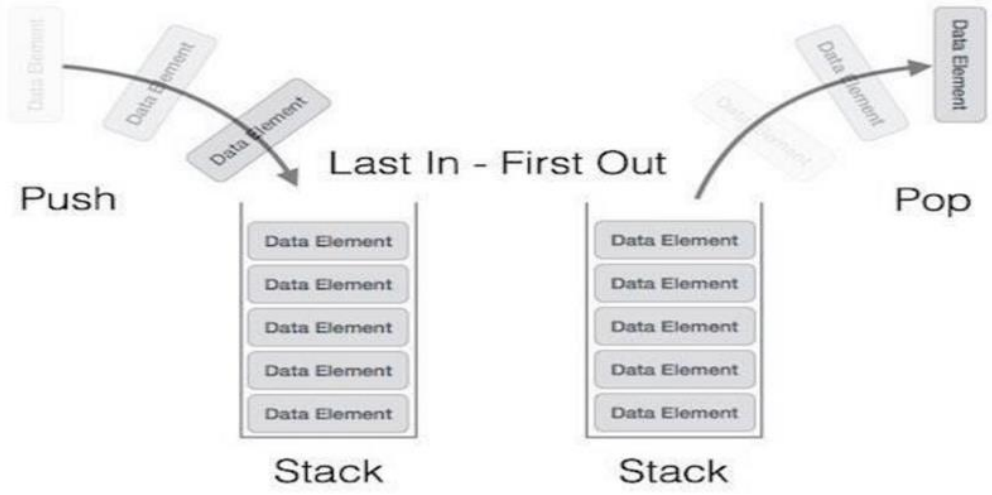
1.1. Stack: A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real -world stack, for example – a deck of cards or a pile of plates, etc.

A real -world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

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

Stack Representation

The following diagram depicts a stack and its operations −



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

Basic Operations

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

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

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.

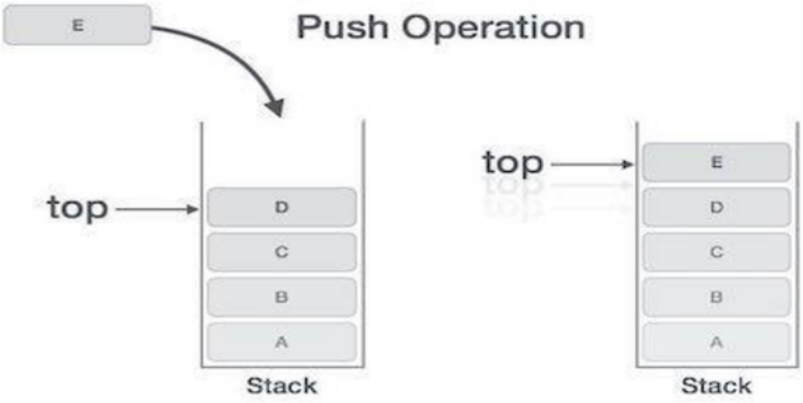
  isEmpty() − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

1.1.1. Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push : stack, data

  if stack is full
    return null
  endif

  top ← top + 1
  stack[top] ← data
```

end procedure

Implementation of this algorithm in C, is very easy. See the following code **Example**

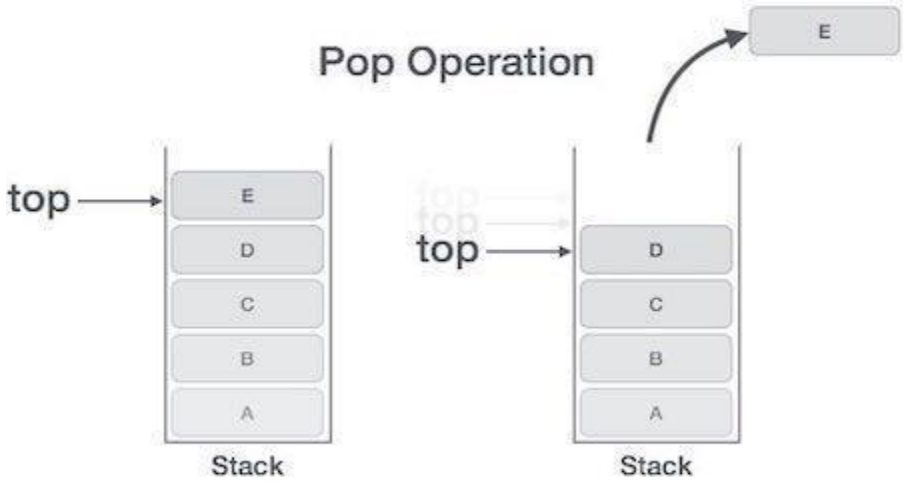
```
void push(int data) {
  if(!isFull())          {
top = top + 1;
stack[top]           =
data;
  } else {
    printf("Could not insert data, Stack is full.\n");
  }
}
```

1.1.2. Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and reallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows
− begin procedure pop: stack

```
  if stack is empty              return null
```

```
   endif

   data ← stack[top]
   top ← top - 1
   return data

end procedure
```

Implementation of this algorithm in C, is as follows –

Example

```
int pop(int data) {

   if (!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf ("Could not retrieve data, Stack is empty.\n");
   }
}
```

1.2. Queue: Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).

Queue follows First -In -First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real -world examples can be seen as queues at the ticket windows and bus-stops.

1.2.1. Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram
given below tries to explain queue representation as data structure –

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

1.2.2. Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

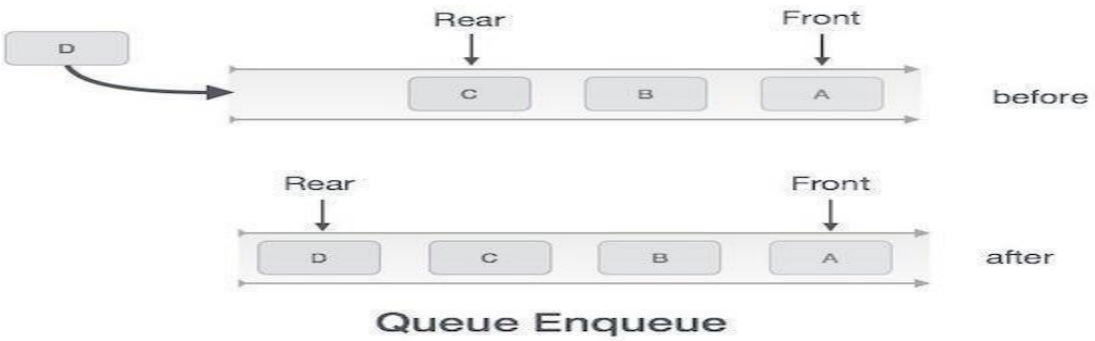
1.2.3. Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

Step 1 − Check if the queue is full.

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



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations. **Algorithm for enqueue operation**

---

procedure enqueue(data)

```
  if queue is full
return overflow
  endif

  rear ← rear + 1
queue[rear] ← data
  return true

end procedure
```

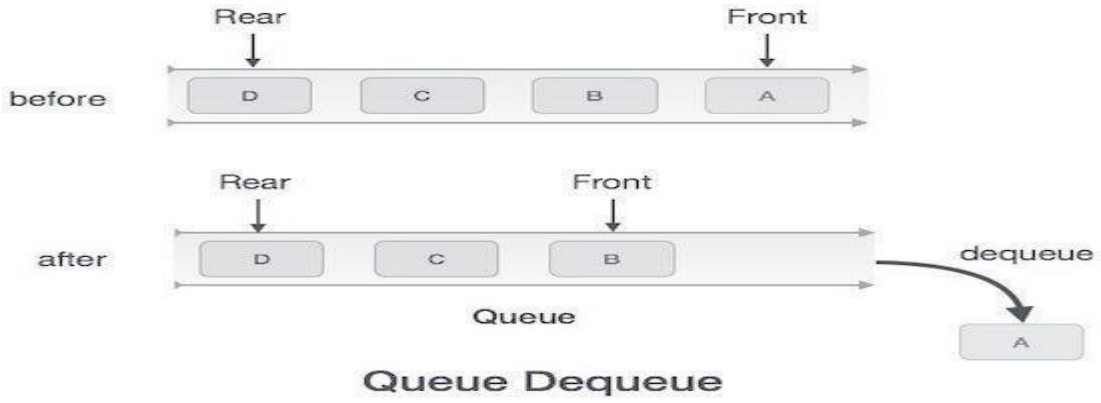
Implementation of enqueue() in C programming language – **Example**

```
int enqueue(int data)
  if(isfull())
    return 0;

  rear = rear + 1;
  queue[rear] = data;

  return 1;
end procedure
```

1.2.4. Dequeue Operation

Accessing data from the queue is a process of two tasks – access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation –

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.
- **Step 5** – Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue

   if queue is empty                return underflow    end if   data = queue[front]    front ← front + 1    return
   true

end procedure
```

Implementation of dequeue() in C programming language −

Example

```
int          dequeue()      {
if(isempty())
     return 0;

   int data = queue[front];
   front = front + 1;

   return data;
}
```

2. Searching Techniques

2.1. Linear search: Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Algorithm

Linear Search (Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found **Step 8:** Exit

Pseudocode

```
procedure linear_search (list, value)

  for each item in the list

    if              item   ==
            ma  return the
    tch               end if
value
item's location
  end for


end procedure
```

2.2. Binary search: Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquers. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the sub array reduces to zero.

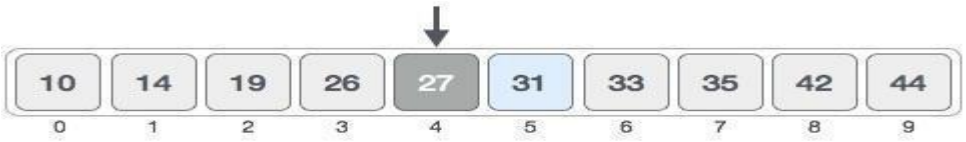
How Binary Search Works?

For a binary search to work, it is mandatory for the target array to be sorted.. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this

formula mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

We change our low to mid + 1 and find the new mid value again.

low = mid + 1 mid
= low + (high -
low) / 2

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.

We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Pseudocode

The Pseudocode of binary search algorithms should look like this:

```
Procedure
binary_search
A ← sorted array
n ← size of array
  x ← value to be searched
   Set lowerBound = 1
  Set upperBound = n
   while x not found          if upperBound < lowerBound          EXIT: x does not exists.
        set midPoint = lowerBound + ( upperBound - lowerBound ) / 2
           if A[midPoint] < x
       set lowerBound = midPoint + 1
            if A[midPoint] > x
       set upperBound = midPoint - 1
     if A[midPoint] = x
       EXIT:          x  found  at  location
midPoint    end while

end procedure
```

3. Sorting Techniques

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

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real- life scenarios −

- **Telephone Directory** − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary** − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

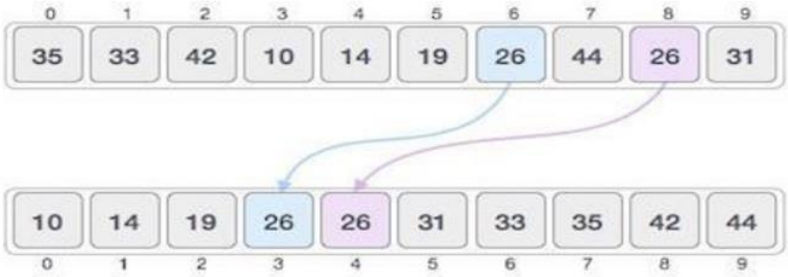
In-place Sorting and Not-in-place Sorting

Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called **in-place sorting**. Bubble sort is an example of in-place sorting.

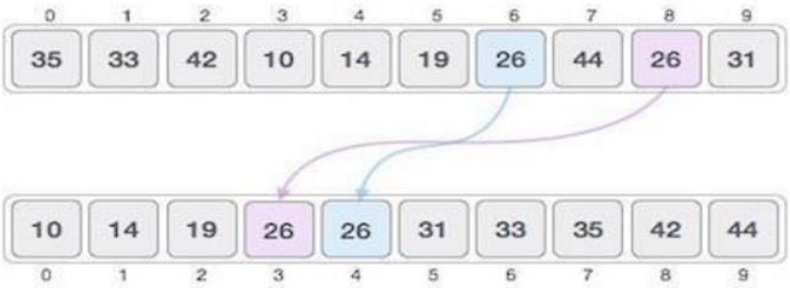
However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called**not-in-place sorting**. Merge-sort is an example of not-in-place sorting.

Stable and Not Stable Sorting

If a sorting algorithm, after sorting the c ontents, does not change the sequence of similar content in which they appear, it is called **stable sorting**.



If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.



Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.

Adaptive and Non-Adaptive Sorting Algorithm

A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them.

A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sortedness.

Important Terms

Some terms are generally coined while discussing sorting techniques, here is a brief introduction to them –

Increasing Order

A sequence of values is said to be in **increasing order**, if the successive element is greater than the previous one. For example, 1, 3, 4, 6, 8, 9 are in increasing order, as every next element is greater than the previous element.

Decreasing Order

A sequence of values is said to be in **decreasing order**, if the successive element is less than the current one. For example, 9, 8, 6, 4, 3, 1 are in decreasing order, as every next element is less than the previous element.

Non-Increasing Order

A sequence of values is said to be in **non-increasing order**, if the successive element is less than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 9, 8, 6, 3, 3, 1 are in non-increasing order, as every next element is less than or equal to (in case of **3**) but not greater than any previous element.

Non-Decreasing Order

A sequence of values is said to be in **non-decreasing order**, if the successive element is greater than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 1, 3, 3, 6, 8, 9 are in non-decreasing order, as every next element is greater than or equal to (in case of 3) but not less than the previous one.

3.1. Bubble sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.

The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

Then we move to the next two values, 35 and 10.

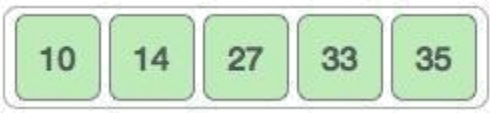
We know then that 10 is smaller 35. Hence they are not sorted.

We swap these values. Wefind that we have reached the end of the array. After one iteration, the array should look like this −

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.

And when there's no swap required, bubble sorts learns that an array is completely sorted. 132



Now we should look into some practical aspects of bubble sort.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
begin BubbleSort(list)

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

   return list

end BubbleSort
```

Pseudocode

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Pseudocode of BubbleSort algorithm can be written as follows

```
procedure bubbleSort( list : array of items )

   loop = list.count;

   for i = 0 to loop-1 do:

swapped   =
false

      for j = 0 to loop-1 do:

         /*   compare      the   adjacent
elements */                  if list[j] >
list[j+1] then             /* swap them
*/
            swap( list[j], list[j+1] )
            swapped = true
```

```
        end if

      end for

      /*if no number was swapped that means

      array is sorted now, break the loop.*/

      if(not
swapped)     then
break       end if


   end for

end procedure return list
```

3.2. Insertion sort

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there.

Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

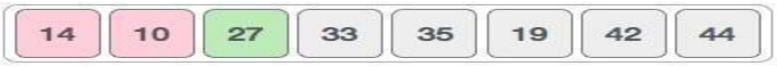
These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 − If it is the first element, it is already sorted. return 1;

Step 2 − Pick next element

Step 3 − Compare with all elements in the sorted sub-list

Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 − Insert the value

Step 6 − Repeat until list is sorted

Pseudocode

```
procedure insertionSort( A : array of
items )    int holePosition
   int valueToInsert
            for              i   =   1   to
length(A) inclusive do:
```

```
    /*      select  value  to  be
inserted        valueToInsert
*/ = A[i]  holePosition = i


    /*locate hole position for the element to be inserted */


    while       holePosition  >  0  and  A[holePosition-1]  >
valueToInsert do: A[holePosition] = A[holePosition-1]
holePosition = holePosition -1 end while

    /* insert the number at hole position */
    A[holePosition] = valueToInsert

  end for

end procedure
```

3.3. Selection sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

How Selection Sort Works?

Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −

Now, let us learn some programming aspects of selection sort.

Algorithm

Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted

Pseudocode

```
procedure   selection
sort     list  : array of
items
   n     : size of list

   for i = 1 to n - 1
   /*      set    current     element     as
minimum*/       min = i

      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
```

```
min = j;
end if
    end for


    /* swap the minimum element with the current
element*/     if indexMin != i then       swap list[min]
and list[i]     end if end for

end procedure
```

3.4. Merge sort

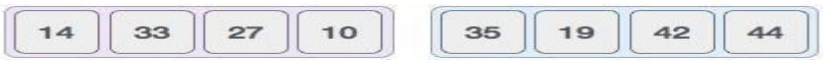
Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

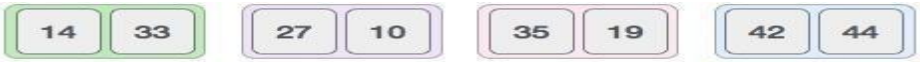
How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following −

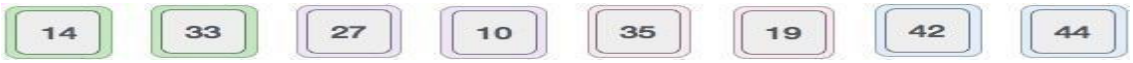
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

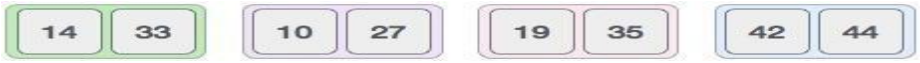


We further divide these arrays and we achieve atomic value which can no more be divided.

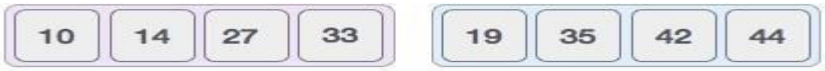


Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this −

Now we should learn some programming aspects of merge sorting.

Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be divided. **Step 3** − merge the smaller lists into new list in sorted order.

Pseudocode

We shall now see the pseudocodes for merge sort functions. As our algorithms point out two main functions − divide & merge.

Merge sort works with recursion and we shall see our implementation in the same way.

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

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

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

   return merge( l1,
l2 ) end procedure

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

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

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

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

```
remove b[0] from b
end while

  return c


end procedure
```

3.5. Shell sort

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as **interval**.

This interval is calculated based on Knuth's formula as:

Knuth's Formula

h = h * 3 + 1

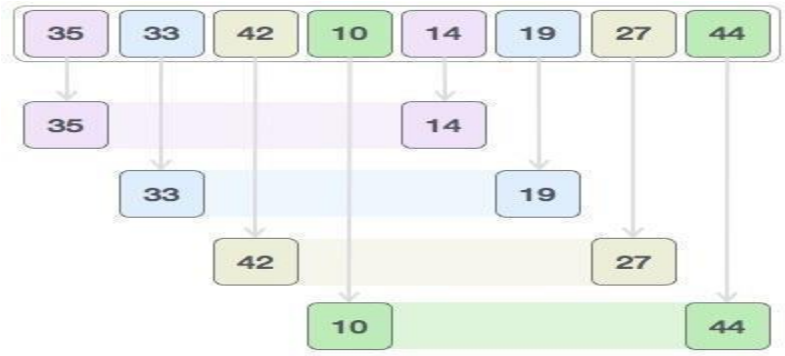
where −

h is interval with initial value 1

This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is O(n), where n is the number of items. And the worst case space complexity is O(n).

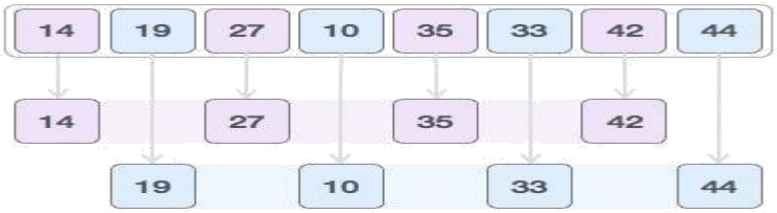
How Shell Sort Works?

Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}

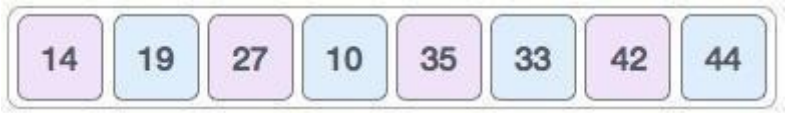


We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this −

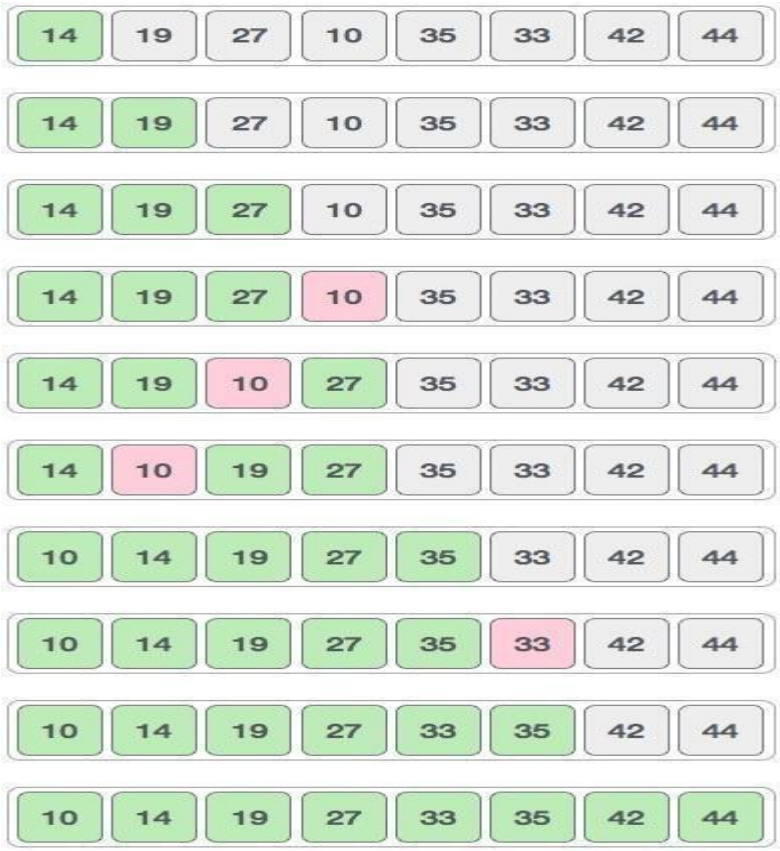
Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this −



Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array. Following is the step-by-step depiction −



We see that it required only four swaps to sort the rest of the array.

Algorithm

Following is the algorithm for shell sort.

Step 1 − Initialize the value of *h*
Step 2 − Divide the list into smaller sub-list of equal interval
h **Step 3** − Sort these sub-lists using **insertion sort Step 3** −
Repeat until complete list is sorted **Pseudocode**

Following is the pseudocode for shell sort.
procedure shellSort()

```
  A : array of items

  /*    calculate  interval*/
while interval < A.length
/3 do:            interval  =
interval * 3 + 1
  end while

  while interval > 0 do:


    for outer = interval; outer < A.length; outer ++ do:

    /*       select  value  to  be  inserted  */
valueToInsert = A[outer]
    inner = outer;

      /*shift element towards right*/ while inner > interval -1
&& A[inner - interval] >= valueToInsert do: A[inner] = A[inner
- interval] inner = inner - interval end while


    /* insert the number at hole position */
    A[inner] = valueToInsert

    end for

  /* calculate interval*/
interval  =  (interval  -1)
/3;

  end while

end procedure
```

3.6. Quick sort is

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

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting sub arrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items. **Partition in Quick Sort**

Following animated representation explains how to find the pivot value in an array.

The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot **Step 3** − left points to the low index

Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot

Quick Sort Pivot Pseudocode

The Pseudocode for the above algorithm can be derived as −

```
function partitionFunc(left, right, pivot)  leftPointer = left
rightPointer = right - 1

  while True do
    while A[++leftPointer] < pivot do
      //do-nothing
    end while

    while rightPointer > 0 && A[--rightPointer] > pivot
      do //do-nothing
    end while

    if leftPointer >= rightPointer break
    else
      swap leftPointer,rightPointer end if

  end while
```

```
   swap leftPointer,right
   return leftPointer

end function
```

Quick Sort Algorithm

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows −

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

Quick Sort Pseudocode

To get more into it, let see the Pseudocode for quick sort algorithm −

```
procedure quickSort(left, right)

   if right-left <=
0 return
   else
      pivot = A[right]
      partition = partitionFunc(left, right, pivot)
quickSort(left,partition-1)
      quickSort(partition+1,right)
   end if

end procedure
```

Activity 2: Guided Practice

Task: Mr MUSABIYINEMA Adrien is a deputy director in charge of TVET a SJITC Nyamirambo, and he was recorded all students from many trades in on excel sheet during registration, Now, he wants to print out the students from each class so that every students will submit their expired student card in the box according to their student number.

1. Use stack methods to arrange the cards in the box so that they are in descending order?
2. Sort all students according to their classes
3. Use binary search to look for the class called L4SFD?

Activity 3: Application

Task: Mrs Jemima NYIRASAFARI is an algirithm fundamentals trainer at SOS Kagugu, She has students marks but arranged according to the alphabet. And she want to see the perfomance of the students. Thelist of student is as follow:

No	1	2	3	4	5	6	7	8	9	10
Marks	50	79	43	90	37	89	66	53	81	73

1. Sort all students according to their marks?
2. Search the position where 50 marks is located

Points to Remember

- **Don't forget that memory usage of a data structure operation should be as little as possible.**
- **Running time or the execution time of operations of data structure must be as small as possible**
- **Processor speed although being very high, falls limited if the data grows to billion records.**

Formative Assessment

1. Describe the following terms:

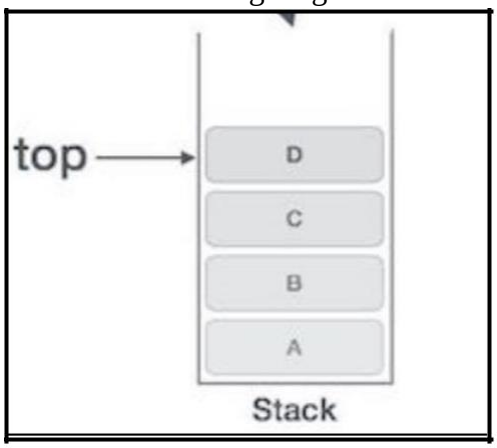
a. Stack

b. Queue

c. Binary search

d. Linear search

2. List the step needed to perform a push operation?
3. Given the following diagram and answer the question related to it:



a. Write all steps needed to perform pop operation the element B 4.

Differentiate enqueue from dequeue operations

5. Using insertion sort, Sort the following array:

Self Reflection

Areas of strength	**Areas for improvement**	**Actions to be taken to improve**
1. **2.**	**1.** **2.**	**1.** **2.**