RTB | RWANDA TVET BOARD

RQF LEVEL 4

SWDDA401

SOFTWARE DEVELOMENT

Data Structure and Algorithm Fundamentals

TRAINEE'S MANUAL

October, 2024

RTB | RWANDA TVET BOARD

DATA STRUCTURE AND ALGORITM FUNDAMENTALS

KOICA Korea International Cooperation Agency

TQUM TVET Quality Management Project

2024

AUTHOR'S NOTE PAGE (COPYRIGHT)

- This work has been produced initially with the Rwanda TVET Board with the support from KOICA through TQUM Project
- This work has copyright, but permission is given to all the Administrative and Academic Staff of the RTB and TVET Schools to make copies by photocopying or other duplicating processes for use at their own workplaces.
- This permission does not extend to making of copies for use outside the immediate environment for which they are made, nor making copies for hire or resale to third parties.
- The views expressed in this version of the work do not necessarily represent the views of RTB. The competent body does not give warranty nor accept any liability
- RTB owns the copyright to the trainee and trainer's manuals. Training providers may reproduce these training manuals in part or in full for training purposes only. Acknowledgment of RTB copyright must be included on any reproductions. Any other use of the manuals must be referred to the RTB.

© Rwanda TVET Board

Copies available from:

- *HQs: Rwanda TVET Board-RTB*
- *Web: www.rtb.gov.rw*
- **KIGALI-RWANDA**

Original published version: October 2024

ACKNOWLEDGEMENTS

The publisher would like to thank the following for their assistance in the elaboration of this training manual:

Rwanda TVET Board (RTB) extends its appreciation to all parties who contributed to the development of the trainer's and trainee's manuals for the TVET Certificate IV in Software Development, specifically for the module **" SWDDA401: Data Structure and Algorithm Fundamentals."**

We extend our gratitude to KOICA Rwanda for its contribution to the development of these training manuals and for its ongoing support of the TVET system in Rwanda

We extend our gratitude to the TQUM Project for its financial and technical support in the development of these training manuals.

We would also like to acknowledge the valuable contributions of all TVET trainers and industry practitioners in the development of this training manual.

The management of Rwanda TVET Board extends its appreciation to both its staff and the staff of the TQUM Project for their efforts in coordinating these activities.

This training manual was developed:

Under Rwanda TVET Board (RTB) guiding policies and directives

Under Financial and Technical support of

COORDINATION TEAM

RWAMASIRABO Aimable

MARIA Bernadette M. Ramos

MUTIJIMA Asher Emmanuel

PRODUCTION TEAM

Authoring and Review

ISHIMWE Gilbert

NIZEYIMANA Martin

Validation

KWIZERA Olivier

SEKABANZA Jean de la Paix

Conception, Adaptation and Editorial works

HATEGEKIMANA Olivier

GANZA Jean Francois Regis

HARELIMANA Wilson

NZABIRINDA Aimable

DUKUZIMANA Therese

NIYONKURU Sylvestre

MANIRAKIZA Jean de Dieu

Formatting, Graphics, Illustrations, and infographics

YEONWOO Choe

SUA Lim

SAEM Lee

SOYEON Kim

WONYEONG Jeong

SHYAKA Emmanuel

Financial and Technical support

KOICA through TQUM Project

TABLE OF CONTENT

ACRONYMS

FIFO: First in First Out

JS: JavaScript

KOICA: Korea International Cooperation Agency

LIFO: Last in First out

RTB: Rwanda TVET Board

SWD: Software Development

TQUM: TVET Quality Management Project

TVET: Technical and Vocational Education and Training

INTRODUCTION

This trainee's manual includes all the knowledge and skills required in TVET Certificate IV in Software Development, specifically for the module of **"Data Structure and Algorithm Fundamentals".** Trainees enrolled in this module will engage in practical activities designed to develop and enhance their competencies. The development of this training manual followed the Competency-Based Training and Assessment (CBT/A) approach, offering ample practical opportunities that mirror real-life situations.

The trainee's manual is organized into Learning Outcomes, which is broken down into indicative content that includes both theoretical and practical activities. It provides detailed information on the key competencies required for each learning outcome, along with the objectives to be achieved.

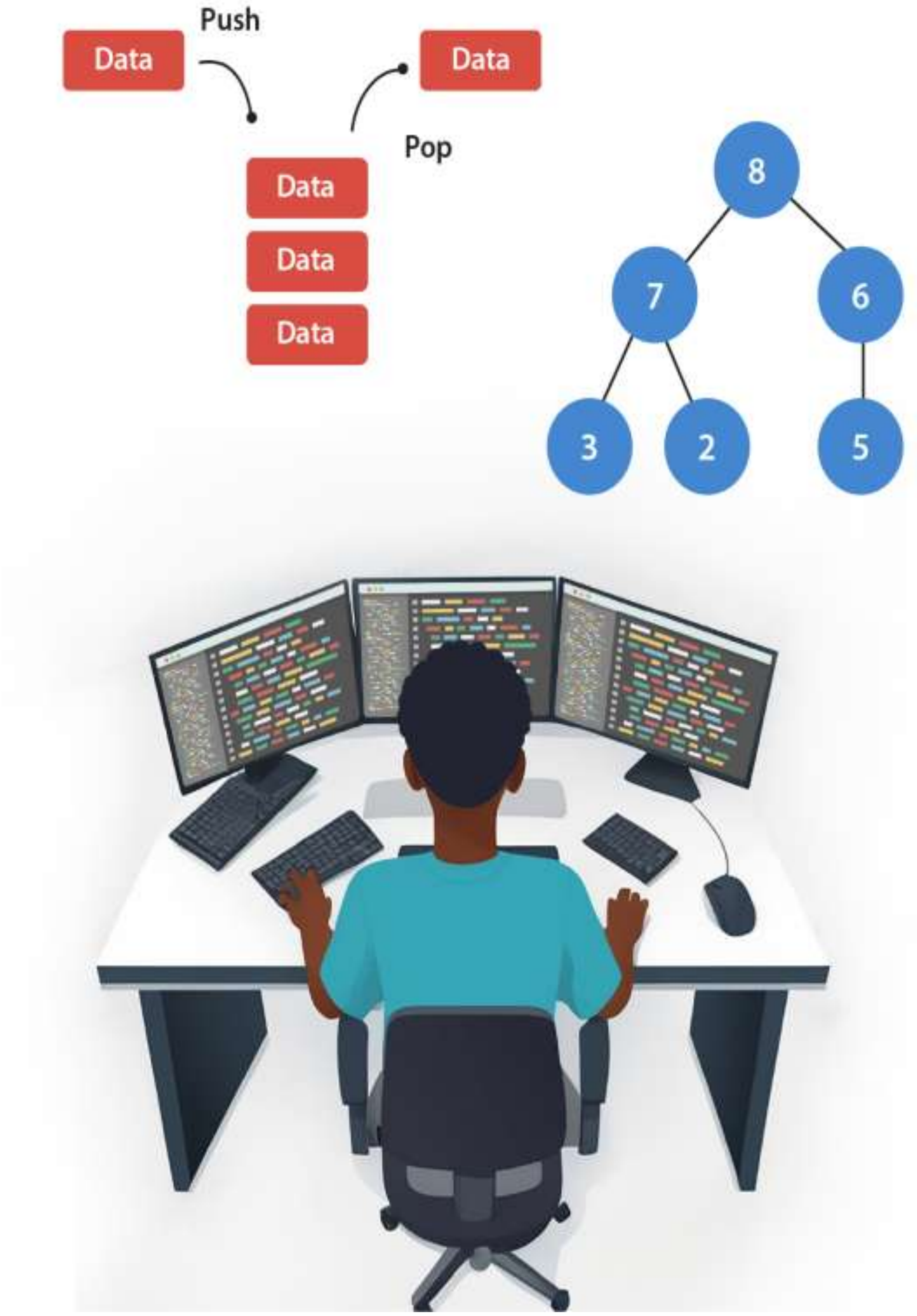
As a trainee, you will start by addressing questions related to the activities, which are designed to foster critical thinking and guide you towards practical applications in the labour market. The manual also provides essential information, including learning hours, required materials, and key tasks to complete throughout the learning process.

All activities included in this training manual are designed to facilitate both individual and group work. After completing the activities, you will conduct a formative assessment, referred to as the end learning outcome assessment. Ensure that you thoroughly review the key readings and the 'Points to Remember' section.

MODULE CODE AND TITLE: SWDDA401 DATA STRUCTURE AND ALGORITHM FUNDAMENTALS

Learning Outcome 1: Apply algorithm Fundamentals
Learning Outcome 2: Apply Data Structure.
Learning Outcome 3: Implement Algorithm using JavaScript.

Learning Outcome 1: Apply algorithm Fundamentals



Indicative contents

1.1 Conversion of number systems

1.2 Description of logic gates and expressions

1.3 Use of data types on variables

1.4 Application of JavaScript operators

1.5 Write an algorithm

Key Competencies for Learning Outcome 1: Apply Algorithm Fundamentals

Knowledge	Skills	Attitudes
● Description of number system ● Description of logic gates and expressions ● Description of the use of data types on variables	● Converting numbers from one numerical system to another ● Applying JavaScript operators ● Developing an algorithm ● Drawing a flowchart	● Being critical thinker ● Being innovative ● Being attentive ● Being creative ● Being practical oriented ● Being Details Oriented ● Having team work spirit

Duration: 30 hrs

Learning outcome 1 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe correctly number system based on number system types
2. Convert correctly Number systems according to the base conversion methods
3. Describe well Logic gates and expressions based on Boolean algebra
4. Apply effectively Data types according to their intended use
5. Apply appropriately Operators based on datatype
6. Write Algorithm properly based on problem to be solved
7. Draw properly program flowchart based on best practice

Resources

Equipment	Tools	Materials
● Computer	● Visual paradigm ● Edraw max ● Lucid chart	● N/A

Indicative content 1.1: Conversion of Number Systems

Duration: 6 hrs

Theoretical Activity 1.1.1: Description of key concepts of Algorithm

Tasks:

1: Answer the following questions:

i. What is mean number system

ii. Discuss about the following concepts of number systems:

a) Decimal base

b) Binary base

c) Hexadecimal base

d) Octal base

iii. Discuss application of number system

2: Write answers on the flipchart/paper.

3: Discuss on provided answers and choose correct answers.

4: Listen attentively, take notes and ask question where is necessary

5: Read the key reading 1.1.1 in the trainee manual

Key readings 1.1.1.: Description of key concepts of Algorithm fundamentals

1. Number system:

A number system is a mathematical notation for expressing numbers of a given set, using digits or other symbols in a consistent manner.

Different number systems are used in various fields, such as mathematics, computer science, and everyday life.

2. Description of some common number systems:

- **Different bases**

B=b numbers are 0, 1, 2, 3, …, b-1

B=2 numbers are 0, 1

B=8 numbers are 0, 1, 2, 3, …,7

B=10 numbers are 0, 1, 2, 3, …,9

B=16 numbers are 0, 1, 2, 3, …,9 , For 10, 11, 12, 13, 14, 15 symbols representing them are respectively A, B, C, D, E, F

- ✓ **Decimal Number System (Base 10):**

The decimal number system is the most commonly used number system. It uses 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. In the decimal system, the position of a digit determines its value, and each position represents a power of 10.

For example, the number 3542 in decimal means $3\times10^3+5\times10^2+4\times10^1+2\times10^0$.

- ✓ **Binary Number System (Base 2):**

The binary number system uses two digits: 0 and 1. It is widely used in computer science for digital representation and processing. Each digit's position represents a power of 2.

For example, the binary number 1011 means $1\times2^3+0\times2^2+1\times2^1+1\times2^0$.

- ✓ **Octal Number System (Base 8):**

The octal number system uses eight digits: 0, 1, 2, 3, 4, 5, 6, and 7. Octal numbers are often used in computer applications, especially in older systems.

- ✓ **Hexadecimal Number System (Base 16):**

The hexadecimal number system uses sixteen digits: 0-9 and A-F, where A stands for 10, B for 11, C for 12, D for 13, E for 14, and F for 15. Hexadecimal numbers are widely used in computer science, especially in programming and digital electronics.

- ✓ **Other Number Systems:**

There are various other number systems, such as base-3 (ternary), base-5 (quinary), base-12 (duodecimal), and so on. These systems are not as commonly used as the ones mentioned above.

Understanding different number systems is essential in computer science, especially in digital arithmetic and data representation. Converting numbers between different bases is a fundamental skill in these fields.

3. Application of number base arithmetic operations

3.1 Number base arithmetic operations

Number base arithmetic involves performing mathematical operations in different numeral systems, such as binary (base 2), octal (base 8), decimal (base 10), and hexadecimal (base 16).

These operations are fundamental in various fields, including computer science, digital electronics, and data communication.

Let's discuss the application of arithmetic operations in different number bases.

1. Binary Arithmetic (Base 2)

Binary arithmetic is the foundation of all digital computing systems. In binary, only two digits are used: 0 and 1.

The primary arithmetic operations in binary are addition, subtraction, multiplication, and division.

- **Addition**: Binary addition follows the same principles as decimal addition, but with a base of 2. The possible sums are:

0 + 0 = 0

1 + 0 = 1

1 + 1 = 10 (which is 2 in decimal, so carry 1)

- **Subtraction**: Binary subtraction uses the concept of borrowing, similar to decimal subtraction. The possible subtractions are:

0 - 0 = 0

1 - 0 = 1

1 - 1 = 0

0 - 1 requires borrowing, resulting in a 1 after borrowing.

- **Multiplication and Division**: Binary multiplication is straightforward, involving simple shifting operations. Division is performed similarly to long division in decimal but with binary digits.

2. Octal Arithmetic (Base 8)

Octal arithmetic is less common but is used in certain computing contexts. The octal system uses digits 0 to 7. Octal arithmetic can be performed similarly to decimal, with carrying and borrowing adapted to base 8.

- **Addition**: Similar rules apply as in decimal, but when sums reach 8, they roll over (carry) to the next higher place.
- **Subtraction**: Borrowing in octal occurs when subtracting a larger number from a smaller one in a given column.
- **Multiplication and Division**: Follows the rules of base 8, with operations often facilitated by conversion from or to decimal or binary.

3. Decimal Arithmetic (Base 10)

Decimal arithmetic is the standard system used in everyday life and most traditional mathematics. Operations in decimal involve the digits 0 through 9.

- **Addition, Subtraction, Multiplication, and Division**: These operations are well-known and form the basis of arithmetic education.

4. Hexadecimal Arithmetic (Base 16)

The hexadecimal system uses sixteen symbols: 0-9 and A-F, where A represents 10, B represents 11, and so on up to F, which represents 15. Hexadecimal is widely used in computing as a compact form of binary representation.

- **Addition**: Similar to decimal and binary, with sums that exceed 15 rolling over (carry) to the next higher place.
- **Subtraction**: Requires borrowing when a larger hexadecimal digit is subtracted from a smaller one.
- **Multiplication and Division**: As with binary and octal, but using base 16 rules.

5. Conversions Between Number Bases

Conversions between different number bases are also important for various applications, especially in computing and digital electronics. Understanding how to convert between binary, octal, decimal, and hexadecimal systems is fundamental in these fields.

3.2 Application

- Binary arithmetic is used in designing digital circuits, where operations are performed using logic gates.
- Many computer algorithms, particularly those involving low-level data manipulation or cryptography, operate directly on binary data.

- Octal was historically used in computer programming of computing systems particularly in older mainframes and minicomputers.
- Decimal arithmetic is important in commerce and finance
- Decimal arithmetic also used in scientific, engineering, and statistical calculations due to its alignment with human numeric understanding.
- Hexadecimal is often used in programming for memory addresses and color codes in web design (e.g., #FFFFFF for white).
- Hexadecimal also used for defining machine-level code and other low-level tasks because it is more readable than binary.
- IP addresses and sub netting often use conversions between binary and decimal.

Practical Activity 1.1.2: Converting a number from one numeral system to another

Task:

1: Read carefully the following questions:

i. Convert 157 from decimal to binary number

ii. Convert 1101.1012 into decimal system.

iii. Convert 100 100 000 from base 2 to octal

iv. Convert 0110 1110 1101 from binary to hexadecimal

v. Convert hexadecimal number (3DE)16 to a binary number

vi. Convert an octal number (345)8 to a binary number

vii. Convert 62010 into hexadecimal

2: Follow carefully as trainer demonstrating to trainees how to convert a number from one numeral system to another.

3: Perform the provided task by applying the knowledge learn from trainer's demonstration

4: Listen carefully and take notes wherever is necessary

5: Ask clarifying question if needed.

6: Ask them to read the Key readings 1.1.2 in their manuals

Key readings 1.1.2: Converting a number from one numeral system to another

1. CONVERSION FROM ONE BASE TO ANOTHER

1.1. Decimal to binary conversion

To convert decimal to binary is very simple, you simply divide the decimal value by 2 and then write down the remainder, repeat this process until you cannot divide by 2 anymore, and for example let’s take the decimal value 157:

157 / 2=78 with a remainder of 1

78 / 2=39 with a remainder of 0

39 / 2=19 with a remainder of 1

19 / 2=9 with a remainder of 1

9 / 2=4 with a remainder of 1

4 / 2=2 with a remainder of 0

2 / 2=1 with a remainder of 0

We stop on 1 which is less than 2

to convert write it first.

$(157)_{10} = (10011101)_2$

1.2. Binary to decimal

1. Write each value of the binary number times two, from right to left, write the power of two starting from0

$\mathbf{1*2^7 + 0*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 157}$.

$(10011101)_2= (157)_{10}$

2. Convert $\mathbf{1101.101_2}$ into decimal system.

$\mathbf{1101.101_2}=1x2^3+1x2^2+0x2^1+1x2^0+1x2^{-1}+0x2^{-2}+1x2^{-3}$

=8+4+0+1+0.5+0+0.125

=13**.625₁₀**

3. Convert the following binary number to their decimal equivalent:

A) 11100.001 B) 110011.10011

1.3. Hexadecimal, octal to binary conversion

To convert a hexadecimal number or octal number into a binary one you take each number of an hexadecimal number and convert it in a binary and put it on the format of four bits. For the octal you put it on the format of three bits.

Examples:

1. Convert an hexadecimal number $(3DE)_{16}$ to a binary number

- ✓ Hexadecimal 3 is equivalent to0011
- ✓ Hexadecimal D is equivalent to 1101
- ✓ Hexadecimal E is equivalent to1110

Thus $(3DE)_{16}$ is equivalent to $(001111011110)_2$

2. Convert an octal number $(345)_8$ to a binary number

Solution

Octal 3 is equivalent to011

Octal 4 is equivalent to100
Octal 5 is equivalent to101
Thus $(345)_8$ is equivalent to $(011100101)_2$

1.4. Binary to hexadecimal and octal conversion

Since the hexadecimal number has its equivalent value on 4 bits, we group the binary number we want to convert in four bits then we search the corresponding number. For the octal we group three bits.

Examples:

a. Convert a binary number $(001111011110)_2$ to an hexadecimal number

Solution

We group these bits by four from right to left as follows:

0011 equivalent to 3

1101 equivalent to D

1110 equivalent to E
Thus $(001111011110)_2$ is equivalent to $(3DE)_{16}$
b. Convert a binary number $(011100101)_2$ to an octal

number

We group these bits by three from right to left as follows:

011 equivalent to 3
100 equivalent to4
101 equivalent to5
Thus $(011100101)_2$ is equivalent to $(345)_8$

1.5. Converting to any base

To convert to any base may be done by division or writing in extended notation. Division is used when converting to a small base. Writing in an extended notation is used when converting to a greater base.

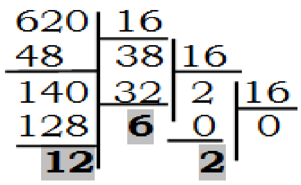
Example:

(a) Convert 620_{10} into hexadecimal

(b) Convert 1110100110_2 into hexadecimal

Solution: (a) $620_{10} = (\quad)_{16}$

⇨ 620



Then, the answer will be written from right to left like this: **620_{10} = $(26C)_{16}$**

(b) **1110100110_2 =($)_{16}$** Here we have to make (form) the groups of 4 digits each, from right to left. If it is necessary, we add the zeros on the last group where the bits are less than 4. Lastly, we translate every quadruple to its equivalent hexadecimal.

The answer will be:

⇨ $\underline{0011}\ \underline{1010}\ \underline{0110} = (\quad)_{16}$

3 A 6

= $3A6_{16}$

Exercises

I. Answer the following questions

a) Convert 0010 0110 from base 2 to decimal

b) Convert 1010 from base 2 to base 10

c) Convert 1100 0110 from binary to base 10

d) Convert 0001 1010 from binary to base 10

e) Convert 0110 1110 1101 from binary to hexadecimal

f) Convert 0100 1011 from base 2 to decimal

g) Convert 100 100 000 from base 2 to octal

h) Convert 101 001 011 from binary to base 8

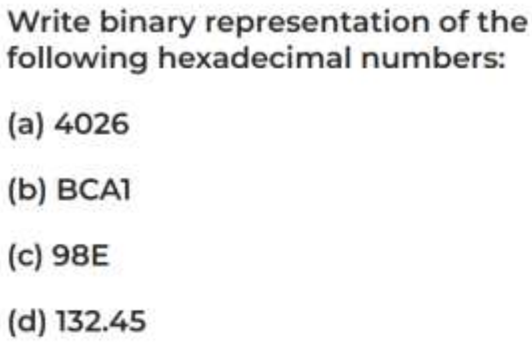
i) Convert 30 from octal to binary

j) Convert 234 from base 10 to binary

II. Convert the following number system to related base

1. Binary number 11001 to decimal.
2. Decimal number 45 to binary
3. Hexadecimal number B2 to binary
4. Binary number 11011 to hexadecimal
5. Decimal number 20 to hexadecimal
6. Hexadecimal number 2C to decimal
7. Binary number 10101100 to its decimal equivalent
8. Decimal number 168 to binary equivalent
9. Hexadecimal number 0x2301 to its binary equivalent

III.



Write binary representation of the following hexadecimal numbers:

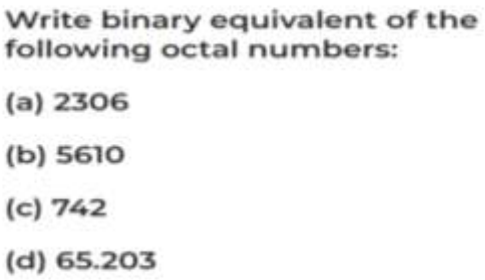
(a) 4026

(b) BCA1

(c) 98E

(d) 132.45

IV.



Write binary equivalent of the following octal numbers:

(a) 2306

(b) 5610

(c) 742

(d) 65.203

Points to Remember

- **Conversions Between Number Bases**

Conversions between different number bases are also crucial for various applications, especially in computing and digital electronics. Understanding how to convert between binary, octal, decimal, and hexadecimal systems is fundamental in these fields.

✓ **Binary to Decimal**: Summing each bit multiplied by 2 raised to its positional power.

✓ **Decimal to Binary**: Repeatedly dividing by 2 and noting remainders.

✓ **Hexadecimal to Binary**: Direct conversion since one hex digit maps to four binary digits.

✓ **Binary to Hexadecimal**: Grouping binary digits in sets of four and converting.

Application of learning 1.1.

RXtc astronomy is a branch of the Rwanda national institute of science. RXtc astronomy deals with celestial objects, space, and the physical universe as a whole. In 2022 the institute have sent spaceship called Ω-spaceship to the moon. And waited for the message that confirm the arrival of the spaceship and coordinates.

After one month the Ω spaceship have sent a series of binary codes which contain a message. You have been recruited to a team of scientists working on deciphering the message sent. Your task is to convert these binary codes into decimal base, binary base, Hexadecimal base and octal decimal base to help the researchers analyze the message.

Notes: The Ω-spaceship provided the following list of binary numbers.

Binary Numbers:

1. 1011001
2. 01001110
3. 11101.001
4. 00110111
5. 10101010
6. 011.11111
7. 10000000
8. 01010101

9. 11001100
10. 00011001

After conversion share your findings with the researchers' team, who will attempt to interpret the message based on the converted numbers.

Indicative content 1.2: Description of Logic Gates and Expressions

Duration: 6 hrs

Theoretical Activity 1.2.1: Description of logic gates and expressions

Tasks:

1: Provide answers on the following questions:

i. Define what is logic gate and discuss about representation of Boolean logic gates

ii. Describe the types of logic gates.

iii. Discuss application of logic gates

2: Present your findings in front of whole class

3: Listen attentively and take notes where is necessary

4: Ask clarifying questions where is necessary

5: Read the key reading 1.2.1 in the trainee manual

Key readings 1.2.1.: Description of logic gates and expressions

representation Of Boolean Logic Gates

1. Logic Gate

Logic gates are fundamental building blocks of digital circuits.

They perform basic logical functions that are essential to digital computation. Each logic gate takes one or more binary inputs (0s and 1s) and produces a single binary output based on a specific logical operation. Common logic gates include:

- **AND Gate:** Outputs 1 only if all inputs are 1.
- **OR Gate:** Outputs 1 if at least one input is 1.
- **NOT Gate:** Outputs the opposite of the input (inverts it).
- **NAND Gate:** Outputs 0 only if all inputs are 1 (opposite of AND gate).
- **NOR Gate:** Outputs 1 only if all inputs are 0 (opposite of OR gate).
- **XOR Gate:** Outputs 1 if an odd number of inputs are 1 (exclusive OR).
- **XNOR Gate:** Outputs 1 if all inputs are the same (opposite of XOR gate).

2. Types of Logic Gates

There are 3 types of logic gates:–

- **Basic Gates:** OR, AND, and NOT Gates.
- **Universal Gates:** NAND, and NOR Gates.
- **Derived Gates:** XOR Gates, and XNOR Gates.

The Universal Logic Gates:

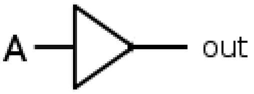
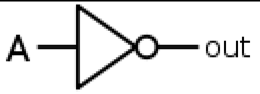
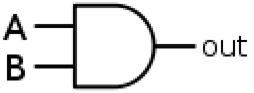
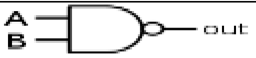
can implement any Boolean expression on its own, this means they don't require any other gate for implementation. A single Universal Logic Gate is capable of building a logic circuit

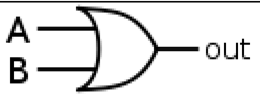
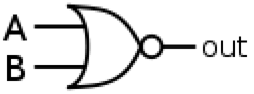
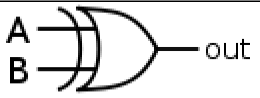
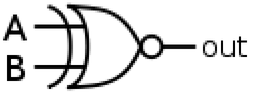
Derived Logic Gates:

The derived or special gates are made for specific applications such as half adders, full adders, and subtractors.

3. Representation of Boolean logic gates.

The table below illustrates basic logic gates, their functions and their truth table

Logic gate	**Illustration**	**Function**	**Truth table**		
Transfer		A	A		Out
			0		0
			1		1
NOT		$\overline{A}$	A		Out
			1		0
			0		1
AND		A .B	A	B	Out
			0	0	0
			0	1	0
			1	0	0
			1	1	1
NAND		$\overline{A \cdot B}$	A	B	Out
			0	0	1
			0	1	1

			1	0	1
			1	1	0
OR		A + B	A	B	Out
			0	0	0
			0	1	1
			1	1	1
			1	1	1
NOR		$\overline{A+B}$	A	B	Out
			0	0	1
			0	1	0
			1	0	0
			1	1	0
XOR		$A \oplus B$	A	B	Out
			0	0	0
			0	1	1
			1	0	1
			1	1	0
XNOR		$\overline{A \oplus B}$	A	B	Out
			0	0	1
			0	1	0
			1	0	0
			1	1	1

Solved Examples on Logic Gates

Question 1: What are Logic gates?

Question 2: What are Universal gates?

Question 3: What is the output of a NOT gate when input 0 is applied?

Question 4: Which logic gate is known as the “invertor”?

Answers:

1. Logic gates are digital circuits that conduct logical operations on the input provided to them and produce appropriate output.
2. To accomplish a specific logical process, universal gates are created by merging two or more fundamental gates. Universal gates are NAND and NOR gates.
3. Because NOT gate is an inverter. As a result, if 0 is used as an input, the output will be 1.
4. An inventor is also known as a NOT gate. The obtained output is the inverse of the input.

- **Application of Boolean logic gates**

Logic gates have a lot of applications, but they are mainly based on their mode of operations or their truth table.

Basic logic gates are often found in circuits such as

✓ safety thermostats,

✓ push-button locks,

✓ automatic watering systems,

✓ Light-activated burglar alarms and many other electronic devices.

Besides, there is no limit to the number of gates that can be used in a single device. In digital **integrated circuits (ICs),** we will find an array of the logic gate area unit.

Uses of Logic Gates

1. Logic gates are utilized in a variety of technologies. These are components of chips (ICs), which are components of computers, phones, laptops, and other electronic devices.
2. Logic gates may be combined in a variety of ways, and a million of these combinations are necessary to make the newest gadgets, satellites, and even robots.
3. Simple logic gate combinations can also be found in burglar alarms, buzzers, switches, and street lights. Because these gates can make a choice to start or stop based on logic, they are often used in a variety of sectors.

4. Logic gates are also important in data transport, calculation, and data processing. Even transistor-transistor logic and CMOS circuitry make extensive use of logic gates.

The Role of Logic Circuits in Computer Science

In computer science, the role of logic circuits is fundamental and profound. They form the core building blocks of digital systems and contribute to the functionality of every type of digital device, from **calculators** and **watches** to **powerful computer processors** and **complex digital networks**.

Practical Activity 1.2.2: Applying Boolean logic gates

Task:

1: Read carefully the following questions:

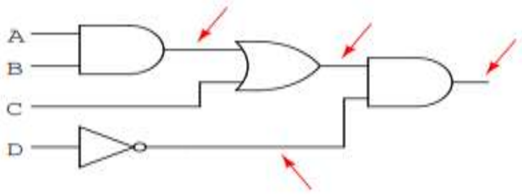
i. Using the above laws, simplify the following expression: (A + B)(A + C)

ii. Convert the following Boolean expression into logic circuit:

a) A(B + CD)

b) $(\overline{AB})B\overline{C}$

iii. Convert the following logic gate circuit into a Boolean expression, writing Boolean sub-expressions next to each gate output in the diagram:



iv. Demonstrate (show) by the truth table the following De Morgan theorems.

a) $\overline{a+b} = \bar{a}.\bar{b}$

b) $\overline{ab} = \bar{a} + \bar{b}$

2: Follow carefully as trainer's demonstration

3: Perform the provided task by applying the knowledge learn from trainer's demonstration

4: Listen carefully and take notes wherever is necessary

5: Ask clarifying question if needed.

6: Ask them to read the Key readings 1.2.2 in their manuals

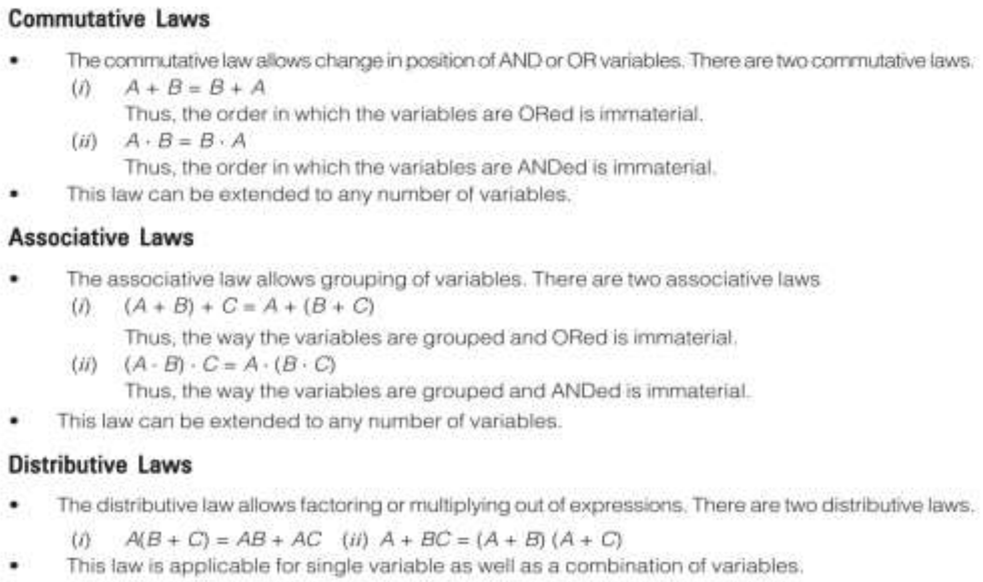
Key readings 1.2.2: Applying Boolean logic gates

1. Laws of Boolean Algebra

The Boolean algebra is governed by certain well-developed rules and laws.

4.1 Types of Boolean Laws

There are six types of Boolean Laws.



Commutative Laws

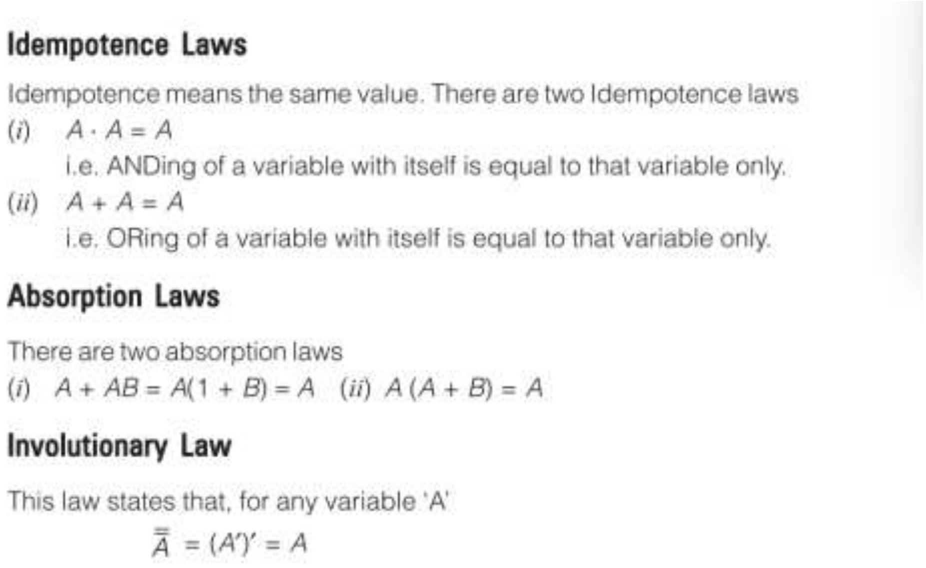
- The commutative law allows change in position of AND or OR variables. There are two commutative laws.
  (*i*) $A + B = B + A$
  Thus, the order in which the variables are ORed is immaterial.
  (*ii*) $A \cdot B = B \cdot A$
  Thus, the order in which the variables are ANDed is immaterial.
- This law can be extended to any number of variables.

Associative Laws

- The associative law allows grouping of variables. There are two associative laws
  (*i*) $(A + B) + C = A + (B + C)$
  Thus, the way the variables are grouped and ORed is immaterial.
  (*ii*) $(A \cdot B) \cdot C = A \cdot (B \cdot C)$
  Thus, the way the variables are grouped and ANDed is immaterial.
- This law can be extended to any number of variables.

Distributive Laws

- The distributive law allows factoring or multiplying out of expressions. There are two distributive laws.
  (*i*) $A(B + C) = AB + AC$ (*ii*) $A + BC = (A + B)(A + C)$
- This law is applicable for single variable as well as a combination of variables.



Idempotence Laws

Idempotence means the same value. There are two Idempotence laws

(*i*) $A \cdot A = A$
i.e. ANDing of a variable with itself is equal to that variable only.

(*ii*) $A + A = A$
i.e. ORing of a variable with itself is equal to that variable only.

Absorption Laws

There are two absorption laws

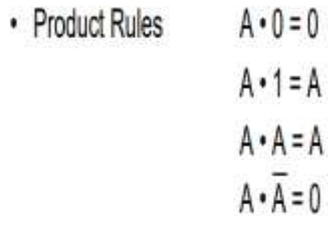
(*i*) $A + AB = A(1 + B) = A$ (*ii*) $A(A + B) = A$

Involutionary Law

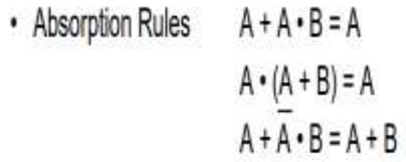
This law states that, for any variable 'A'

$$\overline{\overline{A}} = (A')' = A$$

Summary



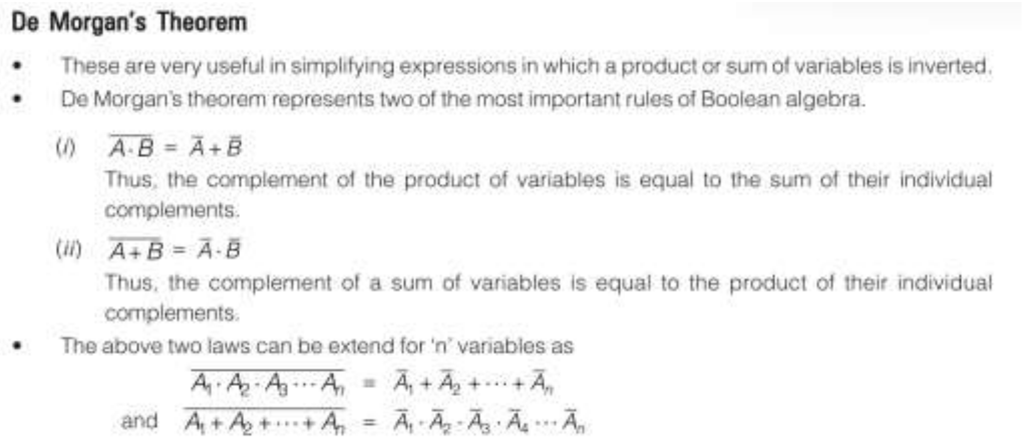
- Product Rules $A \cdot 0 = 0$
  $A \cdot 1 = A$
  $A \cdot A = A$
  $A \cdot \overline{A} = 0$



- Absorption Rules $A + A \cdot B = A$
  $A \cdot (A + B) = A$
  $A + \overline{A} \cdot B = A + B$

4.2 Boolean Algebraic Theorems

Following are few important Boolean Theorems.



De Morgan's Theorem

- These are very useful in simplifying expressions in which a product or sum of variables is inverted.
- De Morgan's theorem represents two of the most important rules of Boolean algebra.

  (*i*) $\overline{A \cdot B} = \overline{A} + \overline{B}$

  Thus, the complement of the product of variables is equal to the sum of their individual complements.

  (*ii*) $\overline{A + B} = \overline{A} \cdot \overline{B}$

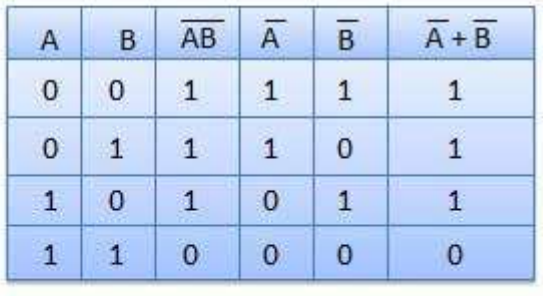
  Thus, the complement of a sum of variables is equal to the product of their individual complements.

- The above two laws can be extend for 'n' variables as

$$\overline{A_1 \cdot A_2 \cdot A_3 \cdots A_n} = \overline{A}_1 + \overline{A}_2 + \cdots + \overline{A}_n$$

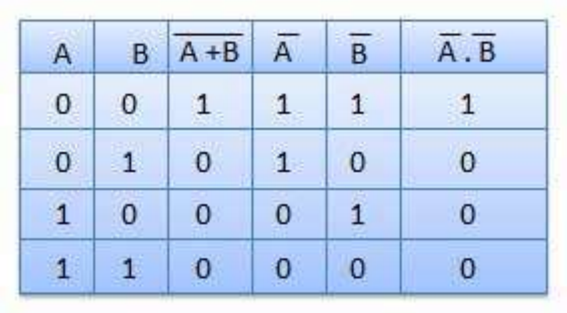
and $$\overline{A_1 + A_2 + \cdots + A_n} = \overline{A}_1 \cdot \overline{A}_2 \cdot \overline{A}_3 \cdot \overline{A}_4 \cdots \overline{A}_n$$

Table showing verification of the De Morgan's first theorem −



A	B	$\overline{AB}$	$\overline{A}$	$\overline{B}$	$\overline{A} + \overline{B}$
0	0	1	1	1	1
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	0	0	0

Table showing verification of the De Morgan's second theorem −



A	B	$\overline{A+B}$	$\overline{A}$	$\overline{B}$	$\overline{A}.\overline{B}$
0	0	1	1	1	1
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	0

Laws of Boolean Algebra Example No1

Using the above laws, simplify the following expression: (A + B)(A + C)

Q = (A + B).(A + C)

A.A + A.C + A.B + B.C → Distributive law

A + A.C + A.B + B.C → Idempotent AND law (A.A = A)

A(1 + C) + A.B + B.C → Distributive law

A.1 + A.B + B.C → Identity OR law (1 + C = 1)

A(1 + B) + B.C → Distributive law

A.1 + B.C → Identity OR law (1 + B = 1)

Q = A + (B.C) → Identity AND law (A.1 = A)

Then the expression: (A + B)(A + C) can be simplified to A + (B.C) as in the Distributive law.

Example 2

Using Boolean algebra techniques, simplify this expression:

AB + A(B + C) + B(B + C)

Solution

Step 1: Apply the distributive law to the second and third terms in the expression, as follows:

AB + AB + AC + BB + BC

Step 2: Apply rule 7 (BB = B) to the fourth term.

AB + AB + AC + B + BC

Step 3: Apply rule 5 (AB + AB = AB) to the first two terms.

AB + AC + B + BC

Step 4: Apply rule 10 (B + BC = B) to the last two terms.

AB + AC + B

Step 5: Apply rule 10 (AB + B = B) to the first and third terms.

B+AC

At this point the expression is simplified as much as possible.

Example 3:

$\overline{A\,\overline{B}.(A + C)} + \overline{A}\,B.\overline{(A + B + \overline{C})}$ Also can be written as **(AB'.(A + C))'+ A'B.(A + B + C')'**

Solution

$\overline{A\,\overline{B}.(A + C)} + \overline{A}\,B.\overline{(A + B + \overline{C})}$

$\overline{A\,\overline{B}.(A + C)} + \overline{A}\,B.\overline{(A + B + \overline{C})} = \overline{A\,\overline{B}} + \overline{A + C} + \overline{A}\,B\,.\,(\overline{A}\,.\,\overline{\overline{B}}\,.\,\overline{\overline{C}})$

$\overline{A\,\overline{B}} + \overline{A + C} + \overline{A}\,B\,.\,(\overline{A}\,.\,\overline{\overline{B}}\,.\,\overline{\overline{C}}) = (\overline{A} + \overline{\overline{B}}) + \overline{A}\,.\,\overline{C} + \overline{A}\,\overline{A}\,B\,B\,C$

$(\overline{A} + \overline{\overline{B}}) + \overline{A}\,.\,\overline{C} + \overline{A}\,\overline{A}\,B\,B\,C = \overline{A} + B + \overline{A}\,\overline{C} + \overline{A}\,B\,C$

$\overline{A} + B + \overline{A}\,\overline{C} + \overline{A}\,B\,C = \overline{A}\,(1 + \overline{C}) + B + \overline{A}\,B\,C$

$\overline{A}\,(1 + \overline{C}) + B + \overline{A}\,B\,C = \overline{A} + B + \overline{A}\,B\,C$

$\overline{A} + B + \overline{A}\,B\,C = \overline{A} + B(1 + \overline{A}\,C)$

$\overline{A} + B(1 + \overline{A}\,C) = \overline{A} + B$

Other typical Examples

Write the following Boolean expressions in their simplest forms

For each question also create a truth table to prove each example.

1. $(\overline{\overline{A}.\overline{B}})$

Using De Morgan's and double negation (Rules 6a and 5) produces $\bar{\bar{A}} + \bar{\bar{B}}$

The double negation rule (Rule 5) leaves us with AB

2. A.B+A.$\bar{B}$

Using distributive Rule 4a A.(B+$\bar{B}$).

Rule 1c states B+$\bar{B}$=1. We are now left with A.1.

Rule 1h states that A.1 = A

3. A.(B+1)

Using Rule 1e B+1 = 1. Therefore, we are left with A.1. Rule 1h is A.1 = A.

4. $\overline{(A.B)}$+$\overline{(\mathrm{A} + \bar{\mathrm{B}})}$

Using De Morgan’s Rule 6a $(\bar{A} + \bar{B}) + (\ \bar{A} + B)$

Then apply rule 1a $\bar{A} + \bar{A} = \bar{A}$

Which gives $\bar{A} + B + \bar{B}$ but Rule 1c makes this $\bar{A} + 1$

We can then apply rule 1e to give $\bar{A}$ + 1 = 1

5. $(X + Y).(X + \bar{Y})$

Using distributive Rule 4b we get X + (Y + $\bar{Y}$)

Using Rule 1d this gives us X + 0

Rule 1f says that X + 0 = X

6. $\overline{\bar{A} + \overline{(B.A)}}$

Using De Morgan’s Law $\overline{\bar{A} + \bar{B} + \bar{A}}$

Using Rule 1a $\overline{\bar{A} + \bar{B}}$

Reapplying De Morgan’s Law gives $\bar{\bar{A}} + \bar{\bar{B}}$, and using double negation (Rule 5) gives A+B

7. $A.B.\bar{C} + A.\bar{C}$

Using the distributive law 4a this gives A.($B.\bar{C} + \bar{C}$))

We can then use the distributive rule again to give A.($\bar{C}(B + 1)$)

Because of Rule 1e we get A.$\bar{C}(1)$)

The answer is therefore A.$\bar{C}$

8. $B.(A + \bar{A})$

Using Rule 1c, B.(1) = B

9. A.B+B

Using distributive law B.(A+1)

Using Rule 1e B.(1) = B

10. $\overline{B}\,.\overline{\overline{A}+\overline{B}}$

Using De Morgan's Law $\overline{B}\,.\,(A.B)$

Using Rule 1d $\overline{B}$.B.A = 0.A

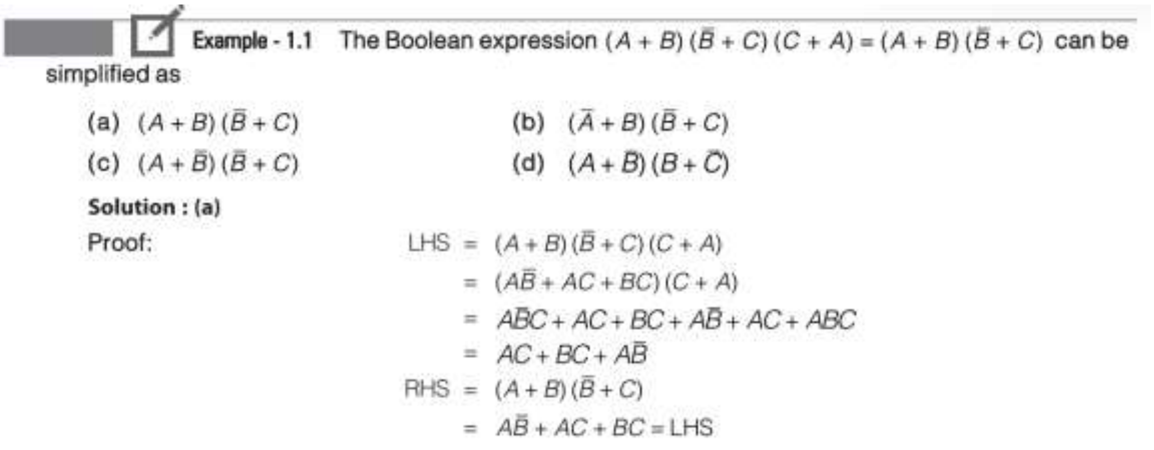
Using Rule 1g 0.A = 0

EXERCISES

I. Simplify the Boolean expressions:

1. AB + A(B + C) + B(B + C).
2. [AB(C + BD) + A B]C
3. ABC + ABC + A B C + ABC + ABC

Other Exercises



Example - 1.1 The Boolean expression $(A+B)(\overline{B}+C)(C+A) = (A+B)(\overline{B}+C)$ can be simplified as

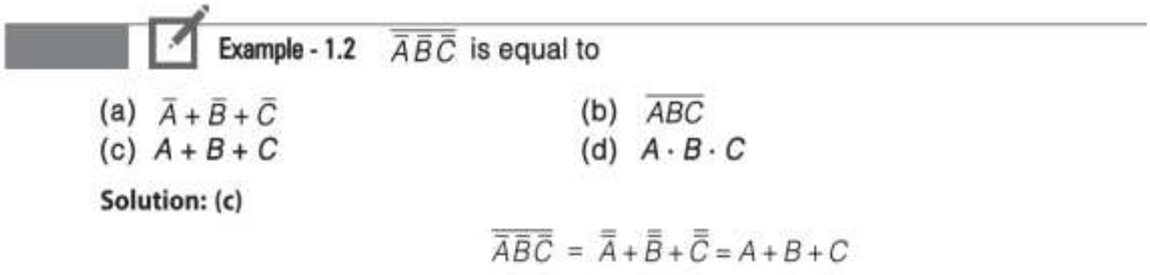
(a) $(A+B)(\overline{B}+C)$ (b) $(\overline{A}+B)(\overline{B}+C)$

(c) $(A+\overline{B})(\overline{B}+C)$ (d) $(A+\overline{B})(B+\overline{C})$

Solution : (a)

Proof:

$$\begin{aligned} \text{LHS} &= (A+B)(\overline{B}+C)(C+A) \\ &= (A\overline{B}+AC+BC)(C+A) \\ &= A\overline{B}C+AC+BC+A\overline{B}+AC+ABC \\ &= AC+BC+A\overline{B} \\ \text{RHS} &= (A+B)(\overline{B}+C) \\ &= A\overline{B}+AC+BC = \text{LHS} \end{aligned}$$



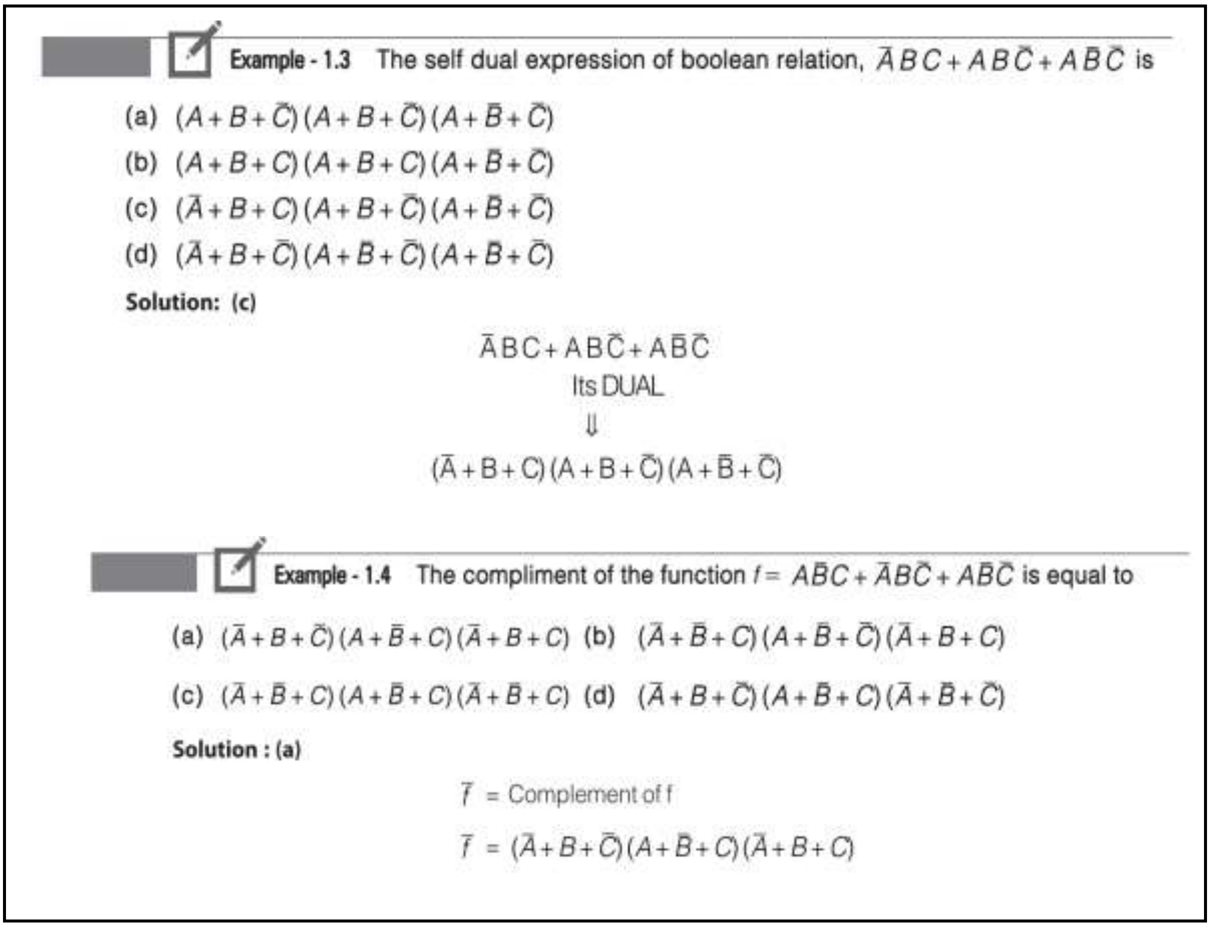
Example - 1.2 $\overline{\overline{A}\,\overline{B}\,\overline{C}}$ is equal to

(a) $\overline{A}+\overline{B}+\overline{C}$ (b) $\overline{ABC}$

(c) $A+B+C$ (d) $A\cdot B\cdot C$

Solution: (c)

$$\overline{\overline{A}\,\overline{B}\,\overline{C}} = \overline{\overline{A}}+\overline{\overline{B}}+\overline{\overline{C}} = A+B+C$$



Example - 1.3 The self dual expression of boolean relation, $\bar{A}BC + AB\bar{C} + A\bar{B}\bar{C}$ is

(a) $(A+B+\bar{C})(A+B+\bar{C})(A+\bar{B}+\bar{C})$

(b) $(A+B+C)(A+B+C)(A+\bar{B}+\bar{C})$

(c) $(\bar{A}+B+C)(A+B+\bar{C})(A+\bar{B}+\bar{C})$

(d) $(\bar{A}+B+\bar{C})(A+\bar{B}+\bar{C})(A+\bar{B}+\bar{C})$

Solution: (c)

$$\bar{A}BC + AB\bar{C} + A\bar{B}\bar{C}$$

Its DUAL

$$\Downarrow$$

$$(\bar{A}+B+C)(A+B+\bar{C})(A+\bar{B}+\bar{C})$$

Example - 1.4 The compliment of the function $f = A\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C}$ is equal to

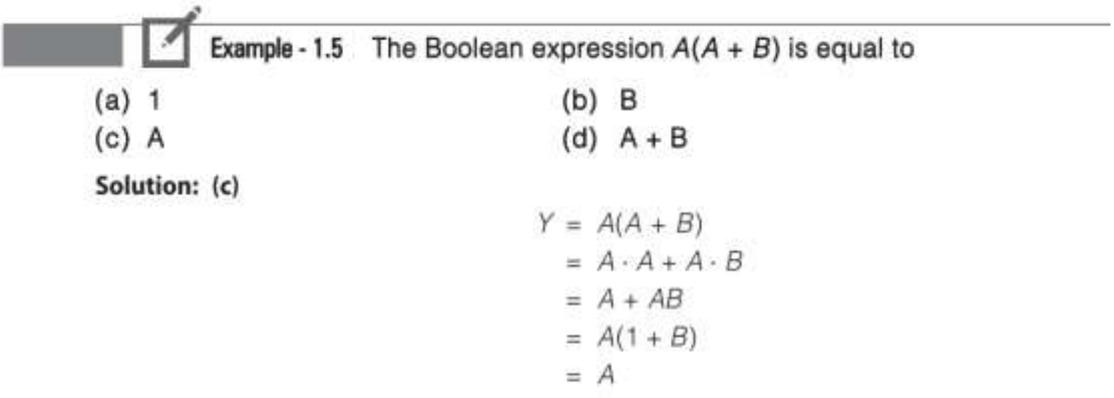
(a) $(\bar{A}+B+\bar{C})(A+\bar{B}+C)(\bar{A}+B+C)$ (b) $(\bar{A}+\bar{B}+C)(A+\bar{B}+\bar{C})(\bar{A}+B+C)$

(c) $(\bar{A}+\bar{B}+C)(A+\bar{B}+C)(\bar{A}+\bar{B}+C)$ (d) $(\bar{A}+B+\bar{C})(A+\bar{B}+C)(\bar{A}+\bar{B}+\bar{C})$

Solution : (a)

$\bar{f}$ = Complement of f

$$\bar{f} = (\bar{A}+B+\bar{C})(A+\bar{B}+C)(\bar{A}+B+C)$$



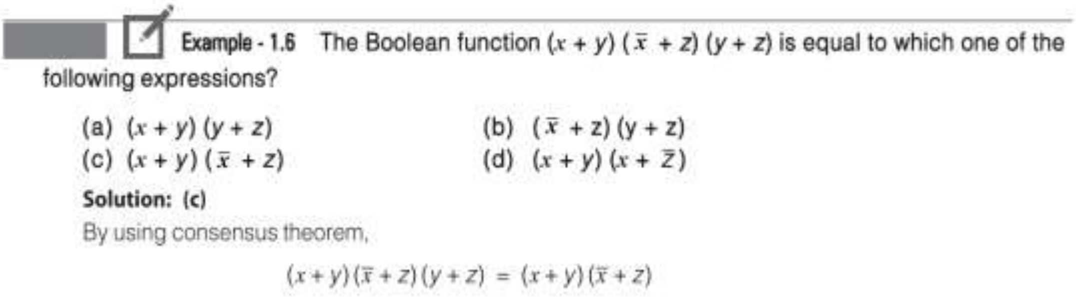
Example - 1.5 The Boolean expression $A(A + B)$ is equal to

(a) 1 (b) B

(c) A (d) A + B

Solution: (c)

$$\begin{aligned} Y &= A(A+B) \\ &= A\cdot A + A\cdot B \\ &= A + AB \\ &= A(1+B) \\ &= A \end{aligned}$$



Example - 1.6 The Boolean function $(x + y)(\bar{x} + z)(y + z)$ is equal to which one of the following expressions?

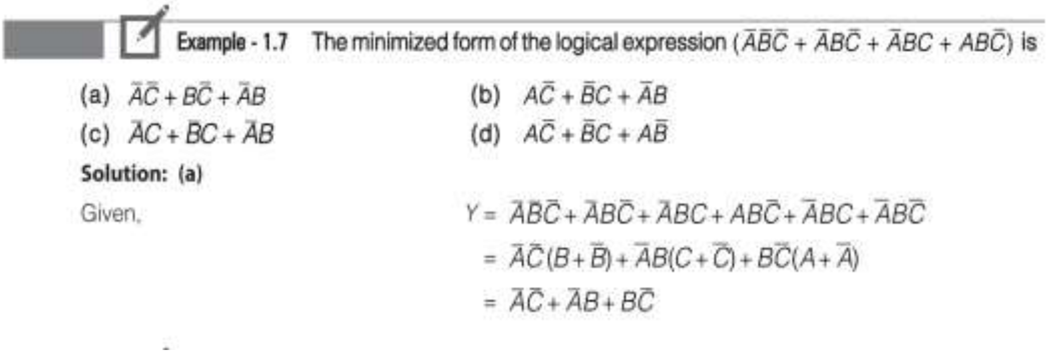
(a) $(x + y)(y + z)$ (b) $(\bar{x} + z)(y + z)$

(c) $(x + y)(\bar{x} + z)$ (d) $(x + y)(x + \bar{z})$

Solution: (c)

By using consensus theorem,

$$(x+y)(\bar{x}+z)(y+z) = (x+y)(\bar{x}+z)$$



Example - 1.7 The minimized form of the logical expression $(\bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}BC + AB\bar{C})$ is

(a) $\bar{A}\bar{C} + B\bar{C} + \bar{A}B$ (b) $A\bar{C} + \bar{B}C + \bar{A}B$

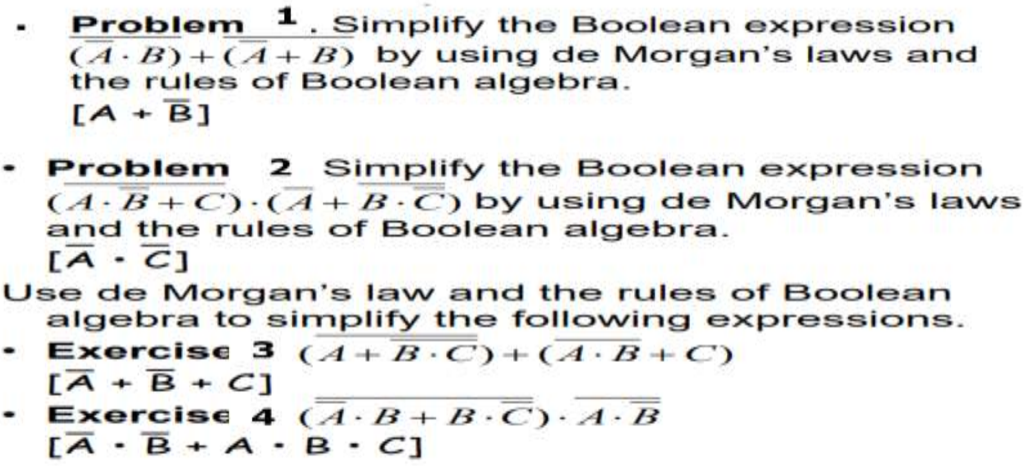
(c) $\bar{A}C + \bar{B}C + \bar{A}B$ (d) $A\bar{C} + \bar{B}C + A\bar{B}$

Solution: (a)

Given,

$$\begin{aligned} Y &= \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}BC + AB\bar{C} + \bar{A}BC + \bar{A}B\bar{C} \\ &= \bar{A}\bar{C}(B+\bar{B}) + \bar{A}B(C+\bar{C}) + B\bar{C}(A+\bar{A}) \\ &= \bar{A}\bar{C} + \bar{A}B + B\bar{C} \end{aligned}$$

Home Work



- **Problem 1**. Simplify the Boolean expression $\overline{(\overline{A}\cdot B)}+\overline{(\overline{A}+B)}$ by using de Morgan's laws and the rules of Boolean algebra.
  $[A + \overline{B}]$
- **Problem 2** Simplify the Boolean expression $(\overline{A\cdot\overline{B}+C})\cdot(\overline{A}+\overline{B\cdot\overline{C}})$ by using de Morgan's laws and the rules of Boolean algebra.
  $[\overline{A}\cdot\overline{C}]$

Use de Morgan's law and the rules of Boolean algebra to simplify the following expressions.

- **Exercise 3** $(\overline{A+\overline{B\cdot C}})+(\overline{A\cdot B}+C)$
  $[\overline{A} + \overline{B} + C]$
- **Exercise 4** $(\overline{\overline{A}\cdot B+B\cdot\overline{C}})\cdot\overline{A\cdot\overline{B}}$
  $[\overline{A}\cdot\overline{B} + A\cdot B\cdot C]$

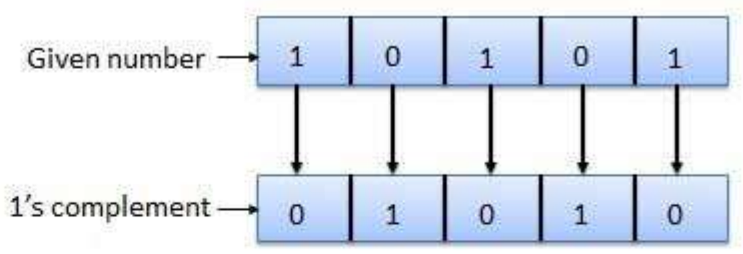
COMPLEMENT FOR BINARY NUMBER

Complements are used in the digital computers in order to simplify the subtraction operation and for the logical manipulations. There are two types of complements.

The two types of complements for the binary system are 1's complement and 2's complement.

1's complement

The 1's complement of a number is found by changing all 1's to 0's and all 0's to 1's. This is called as taking complement or 1's complement. Example of 1's Complement is as follows.

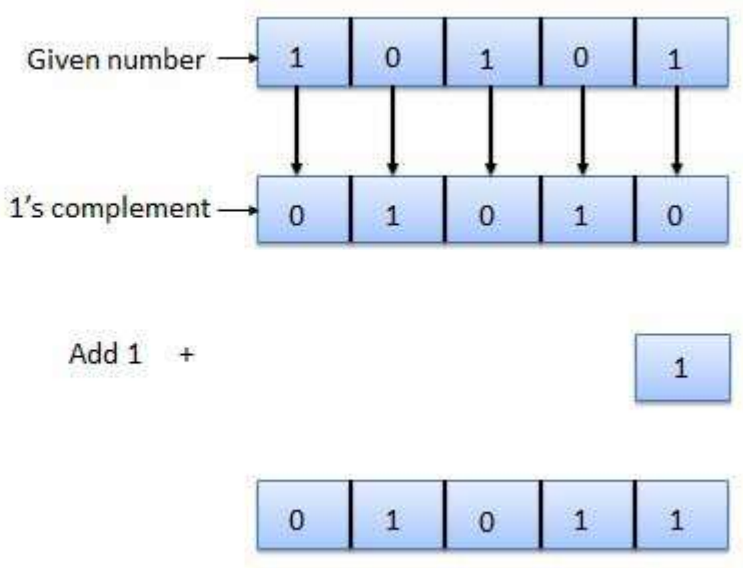


2's complement

The 2's complement of binary number is obtained by adding 1 to the Least Significant Bit (LSB) of 1's complement of the number.

2's complement = 1's complement + 1

Example of 2's Complement is as follows.



In **two's complement** form, a negative **number** is the **2's complement** of its positive **number** with the **subtraction** of two **numbers** being A − B = A + (**2's complement** of B)**. Two's complement** is one's **complement** + 1.

- Circuits

A logic circuit can be defined as an electrical circuit that executes logical operations on one or more binary inputs to produce a single binary output.

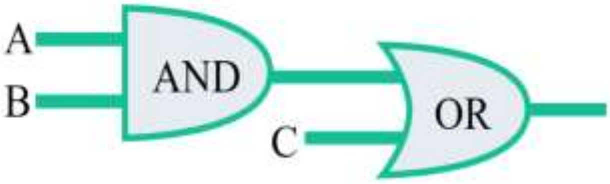
They operate based on Boolean algebra principles, which are grounded in the truth values true and false, often represented as 1 and 0, respectively.

Logic circuits are critical in CPU microarchitecture and cores to undertake tasks such as computation, encoding, multiplexing, and memory interfacing. They also play a significant role in developing hardware like ALUs (Arithmetic Logic Units).

Circuits enables computers to do more complex operations than they could accomplish with just a single gate.

The smallest circuit is a chain of 2 logic gates.

Consider this circuit:



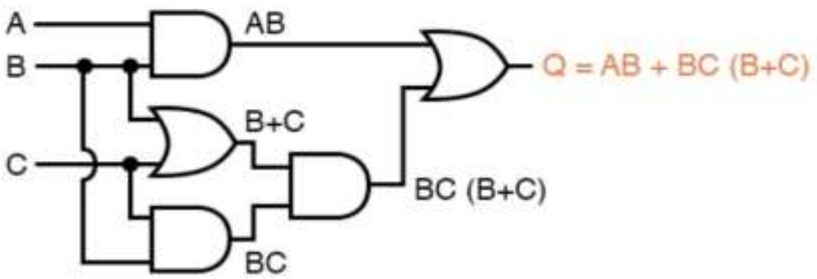
Writing Boolean Expression to Simplify Circuits

Our first step in simplification must be to write a Boolean expression for this circuit.

This task is easily performed step by step if we start by writing sub-expressions at the output of each gate, corresponding to the respective input signals for each gate.

Remember that OR gates are equivalent to Boolean addition, while AND gates are equivalent to Boolean multiplication.

For example, I'll write sub-expressions at the outputs of the first three gates:



Exercises

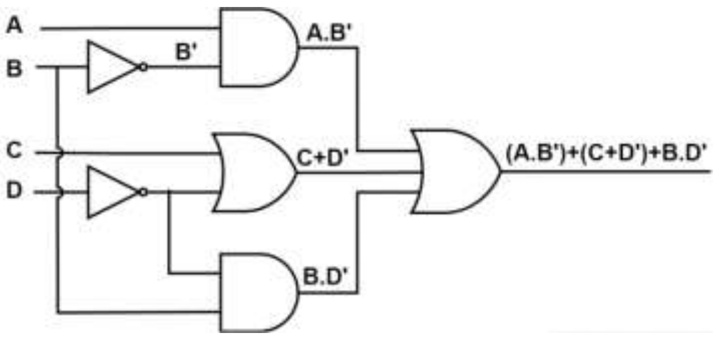
1. Draw a logic circuit of the following Boolean expression:

(i) (A.B) + (C + D) + (B.D)

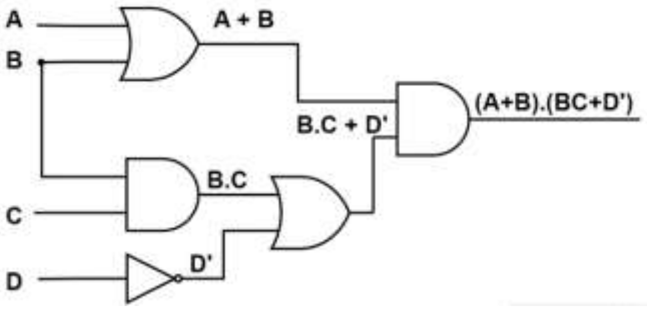
(ii) (A+B).(BC+D)

Answers:

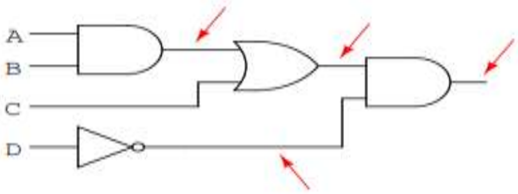
i.

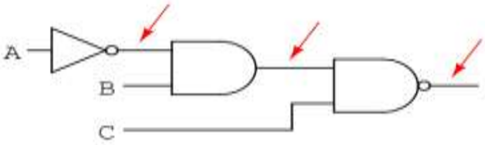


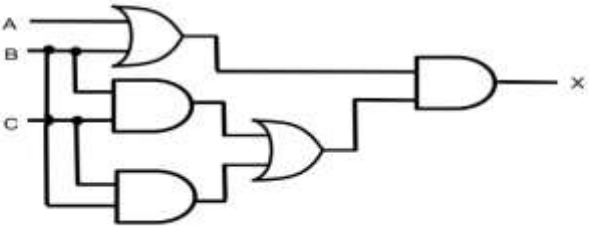
ii.



2.Convert the following logic gate circuit into a Boolean expression, writing Boolean sub-expressions next to each gate output in the diagram:







- **Truth table**

A **truth table** is a mathematical table used in logic to compute the functional values of logical expressions on each of their functional arguments, that is, on each combination of values taken by their logical variables.

When you study truth tables for logic gates or logic circuits, you'll find that each column corresponds to an input or output. Each row in the truth table provides a unique combination of input values, mapping these values to their corresponding output values.

In the world of digital electronics, truth tables are vital in the following ways:

- They can validate the operation of a logic gate or circuit in all possible conditions.
- They can provide a concise and complete summary of all the possible states of a digital system.
- They are beneficial when designing a logic circuit as they give insight into how the circuit will perform.

The procedure to detect output states using a truth table consists of the following steps:

1. Create a truth table with a number of input columns equal to the number of inputs in your logic circuit. Add one more column to denote the output.
2. The number of rows in the truth table would equal 2n, where n is the number of inputs.

For example, if you have 2 inputs, you'll have 22=4 rows. Two zeros and two ones

3. Fill in the table with all possible combinations of inputs.

4. Based on the logic gate or combination of gates in your logic circuit, fill in the output for each row.

Example

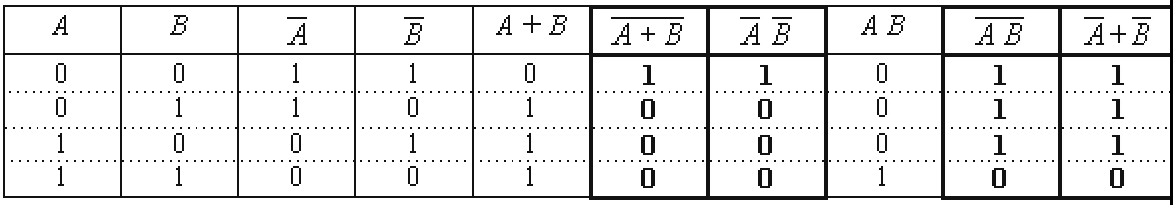
Demonstrate (show) by the truth table the following De Morgan theorems.

a) $\overline{a+b} = \overline{a}.\overline{b}$

b) $\overline{ab} = \overline{a}+\overline{b}$

Solution:

$\overline{A + B} = \overline{A}\overline{B}$ and $\overline{AB} = \overline{A} + \overline{B}$



A	B	$\overline{A}$	$\overline{B}$	$A+B$	$\overline{A+B}$	$\overline{A}\,\overline{B}$	AB	$\overline{AB}$	$\overline{A}+\overline{B}$
0	0	1	1	0	**1**	**1**	0	**1**	**1**
0	1	1	0	1	**0**	**0**	0	**1**	**1**
1	0	0	1	1	**0**	**0**	0	**1**	**1**
1	1	0	0	1	**0**	**0**	1	**0**	**0**

Points to Remember

- A Logic gate is an essential component of digital circuits. It takes one or more binary inputs and produces an output based on the logic rule it operates.
- There are 3 types of logic gates: **Basic Gates** :(OR, AND, and NOT Gates), **Universal Gates** :(NAND, and NOR Gates) and **Derived Gates**: (XOR Gates, and XNOR Gates).
- In computer science, the role of logic circuits is fundamental and profound. They form the core building blocks of digital systems and contribute to the functionality of every type of digital device, from **calculators** and **watches** to **powerful computer processors** and **complex digital networks**
- The procedure to detect output states using a truth table consists of the following steps:

1. Create a truth table with a number of input columns equal to the number of inputs in your logic circuit. Add one more column to denote the output.
2. The number of rows in the truth table would equal 2n, where n is the number of inputs.

For example, if you have 2 inputs, you'll have 22=4 rows. Two zeros and two ones

3. Fill in the table with all possible combinations of inputs.

4. Based on the logic gate or combination of gates in your logic circuit, fill in the output for each row.

Application of learning 1.2.

A team of software engineers is developing a smart home automation system for a new high-tech residential complex. The system aims to manage various home functions, such as lighting, heating, security, and entertainment, through a centralized control hub. To achieve a seamless and efficient operation, the team needs to design an algorithm that decides when and how these functions should be activated based on multiple input conditions.

To optimize the decision-making process, the team decides to use logic gates and Boolean expressions to define the conditions under which different actions will be taken. By applying the fundamentals of logic gates, they can create a simplified model that efficiently controls the automation system. The engineers need to decide which logic gates (AND, OR, NOT, NAND, NOR, XOR, XNOR) to use and how to combine them to represent complex logical expressions.

Indicative content 1.3: Use of Data Types on Variables

Duration: 6 hrs

Theoretical Activity 1.3.1: Description of data types in JavaScript

Tasks:

1: Answer the following questions:

i. Discuss on the following terms as used in JavaScript:

a) Variable

b) Data types

2: Write answers on the flipchart/paper.

3: Discuss on provided answers and choose correct answers.

4: Listen attentively, take notes and ask question where is necessary

5: Read the key reading 1.3.1 in the trainee manual

Key readings 1.3.1.: Description of data types in JavaScript

1. Variables and Data types in JavaScript

- **Variables**

In JavaScript, Variable is a container for Storing Data. Also we can say that variable is a memory zone used to store and manage data.

JavaScript Variables can be declared in 4 ways:

- ✓ Automatically
- ✓ Using var
- ✓ Using let
- ✓ Using const

In this first example, x, y, and z are undeclared variables.

They are automatically declared when first used:

Example

```
x = 5;
y = 6;
```

```
z = x + y;
```

Notes: It is considered good programming practice to always declare variables before use.

From the examples you can guess:

✓ x stores the value 5

✓ y stores the value 6

✓ z stores the value 11

Example using var

```
var x = 5;

var y = 6;

var z = x + y;
```

Note:

The **var** keyword was used in all JavaScript code from 1995 to 2015.

The **let** and **const** keywords were added to JavaScript in 2015.

The **var** keyword should only be used in code written for older browsers.

Example using let

```
let x = 5;

let y = 6;

let z = x + y;
```

Example using const

```
const x = 5;

const y = 6;

const z = x + y;
```

Mixed Example

```
const price1 = 5;

const price2 = 6;

let total = price1 + price2;
```

The two variables price1 and price2 are declared with the const keyword.

These are constant values and cannot be changed.

The variable total is declared with the let keyword.

The value total can be changed.

A variable to be used must first be declared. Declare a variable; means create it by giving it a **name** and a **data type**.

- **Advantages of a variable in an algorithm**

✓ Variable used to store information to be referenced and manipulated in a computer program.

✓ Provide a way of labeling data with a descriptive name

✓ It is a container that hold or store data in memory

✓ Tells compiler how much space is required for the data.

1. Datatype

Data type is **an attribute associated with a data that tells a computer system how to interpret its value**.

Data types are primarily used in computer programming, in which variables are created to store data. Each variable is assigned a data type that determines what type of data the variable may contain.

Data type is a classification of data, which can store a specific type of information.

There are two types of data types in JavaScript:

➢ Primitive or (Built-in) Data Type

➢ Non-primitive or (Derived or reference) Data type

- **Primitive Data Type**

Primitive data types are predefined types of data, which are supported by the programming language.

Primitive Data Types:

- **Number:** Represents numeric values, both integers and floating-point numbers.

Example: let age = 25; let pi = 3.14159;

- **String:** Represents textual data, enclosed in single or double quotes.

Example: let name = "Alice"; let greeting = 'Hello, world!';

- **BigInt:** Represents arbitrarily large integers (introduced in ES2020).

Example: let largeNumber = 9007199254740991n;

- **Boolean**: Represents logical values, either true or false.

Example: let isLoggedIn = true; let isAvailable = false;

- **Undefined:** Represents a variable that has been declared but not assigned a value.

Example: let x; console.log(x); // Output: undefined

- **Null**: Represents the intentional absence of any object value.

Example: let person = null;

- **Symbol:** Represents unique and immutable identifiers (introduced in ES2015).

Example: let mySymbol = Symbol("uniqueKey");

- **Derived or Non- Primitive Data type**

The data type that are derived from primary data types are known as non-primitive data type.

Non-primitive data types are not defined by the programming language, but are instead created by the programmer.

They are sometimes called “**reference variables,"** or "**object references,"** since they reference a memory location, which stores the data.

NB: The non-primitive data types are used to store the group of values.

✓ Object: A collection of key-value pairs, used to store complex data structures.

Example: let person = { name: "John", age: 30 };

✓ Array: An ordered collection of values, accessed by numerical indices.

Example: let numbers = [1, 2, 3, 4, 5];

Is it a must to specify a data type in JavaScript?

No, it's not mandatory to explicitly specify data types when declaring variables in JavaScript. Here's why:

JavaScript is dynamically typed:

✓ It infers the data type of a variable based on the value assigned to it at runtime.

✓ This means you can declare a variable without specifying a type, and its type will be determined by the first value you assign to it.

Example:

```
let myVar = 10;   // myVar is inferred as a number
myVar = "hello"; // Now myVar is a string
```

JavaScript is a **dynamic type language**, means you don't need to specify type of the variable because it is dynamically used by JavaScript engine. You need to use **var** here to specify the data type. It can hold any type of values such as numbers, strings etc. For

Example:

```
1.  var a=40;//holding number
2. var b="Rahul";//holding string
```

2. Application of datatypes

JavaScript is a dynamically typed language. It means that a variable doesn’t associate with a type. In other words, a variable can hold a value of different types.

For example:

```
let counter = 120; // counter is a number
counter = false;   // counter is now a boolean
counter = "foo";   // counter is now a stringCode language: JavaScript (javascript)
```

To get the current type of the value that the variable stores, you use the typeof operator:

```
let counter = 120;
console.log(typeof(counter)); // "number"
counter = false;
console.log(typeof(counter)); // "boolean"
counter = "Hi";
console.log(typeof(counter)); // "string"Code language: JavaScript (javascript)
```

Output:

```
"number"
"boolean"
"string"
```

- **The undefined type**

The undefined type is a primitive type that has only one value undefined. By default, when a variable is declared but not initialized, it is assigned the value of undefined.

Consider the following example:

```
let counter;
console.log(counter);       // undefined
console.log(typeof counter); // undefinedCode language: JavaScript (javascript)
```

In this example, the counter is a variable. Since counter hasn't been initialized, it is assigned the value undefined. The type of counter is also undefined.

It's important to note that the typeof operator also returns undefined when you call it on a variable that hasn't been declared:

```
console.log(typeof undeclaredVar); // undefinedCode language: JavaScript (javascript)
```

- **The null type**

The null type is the second primitive data type that also has only one value null. For example:

```
let obj = null;
console.log(typeof obj); // objectCode language: JavaScript (javascript)
```

The typeof null returns object is a known bug in JavaScript. A proposal to fix this was proposed but rejected. The reason was the that fix would break a lot of existing sites.

JavaScript defines that null is equal to undefined as follows:

```
console.log(null == undefined); // trueCode language: JavaScript (javascript)
```

- **The number type**

JavaScript uses the number type to represent both integer and floating-point numbers.

The following statement declares a variable and initializes its value with an integer:

```
let num = 100;Code language: JavaScript (javascript)
```

To represent a floating-point number, you include a decimal point followed by at least one number. For example:

```
let price= 12.5;
let discount = 0.05;Code language: JavaScript (javascript)
```

Note that JavaScript automatically converts a floating-point number into an integer number if the number appears to be a whole number.

The reason is that Javascript always wants to use less memory since a floating-point value uses twice as much memory as an integer value. For example:

```
let price = 200.00; // interpreted as an integer 200Code language: JavaScript (javascript)
```

To get the range of the number type, you use Number.MIN_VALUE and Number.MAX_VALUE. For example:

```
console.log(Number.MAX_VALUE); // 1.7976931348623157e+308
console.log(Number.MIN_VALUE); // 5e-324Code language: JavaScript (javascript)
```

Also, you can use Infinity and -Infinity to represent the infinite number. For example:

```
console.log(Number.MAX_VALUE + Number.MAX_VALUE); // Infinity
console.log(-Number.MAX_VALUE - Number.MAX_VALUE); // -InfinityCode language: JavaScript (javascript)
```

NaN

NaN stands for Not a Number. It is a special numeric value that indicates an invalid number. For example, the division of a string by a number returns NaN:.

```
console.log('a'/2); // NaN;Code language: JavaScript (javascript)
```

The NaN has two special characteristics:

1) Any operation with NaN returns NaN.
2) The NaN does not equal any value, including itself.

Here are some examples:

```
console.log(NaN/2); // NaN
console.log(NaN == NaN); // falseCode language: JavaScript (javascript)
```

- **The string type**

In JavaScript, a string is a sequence of zero or more characters. A string literal begins and ends with either a single quote(') or a double quote (").

A string that begins with a double quote must end with a double quote. Likewise, a string that begins with a single quote must also end with a single quote:

```
let greeting = 'Hi';
let message  = "Bye";Code language: JavaScript (javascript)
```

If you want to single quote or double quotes in a literal string, you need to use the backslash to escape it. For example:

```
let message = 'I\'m also a valid string'; // use \ to escape the single quote (')Code language: JavaScript (javascript)
```

JavaScript strings are immutable. This means that it cannot be modified once created. However, you can create a new string from an existing string. For example:

```
let str = 'JavaScript';
str = str + ' String';Code language: JavaScript (javascript)
```

In this example:

1) First, declare the str variable and initialize it to a string of 'JavaScript'.
2) Second, use the + operator to combine 'JavaScript' with ' String' to make its value as 'Javascript String'.

Behind the scene, the JavaScript engine creates a new string that holds the new string 'JavaScript String' and destroys the original strings 'JavaScript' and ' String'.

The following example attempts to change the first character of the string JavaScript:

```
let s = 'JavaScript';
    s[0] = 'j';
    console.log(s)Code language: JavaScript (javascript)
```

The output is:

```
'JavaScript'Code language: JavaScript (javascript)
```

But not:

```
'javaScript'Code language: JavaScript (javascript)
```

- **The boolean type**

The boolean type has two literal values: true and false in lowercase. The following example declares two variables that hold the boolean values.

```
let inProgress = true;
let completed = false;
console.log(typeof completed); // booleanCode language: JavaScript (javascript)
```

JavaScript allows values of other types to be converted into boolean values of **true or false**.

Points to Remember

- In JavaScript, Variable is a container for Storing Data. Also, we can say that variable is a memory zone used to store and manage data.
- Data type is an attribute associated with a data that tells a computer system how to interpret its value.
- Data types are primarily used in computer programming, in which variables are created to store data. Each variable is assigned a data type that determines what type of data the variable may contain.

Application of learning 1.3.

Movie Ticket Booking Form

The application based on where a user is interacting with a simple JavaScript-based webpage to fill out a form for a movie ticket booking. The form captures basic information like the user's name, age, contact number, whether they want snacks, and their preferred movie format. In this scenario, we can apply only JavaScript primitive types to handle the form data.

The webpage contains a form with the following fields:

1. **Name** (String)
2. **Age** (Number)
3. **Contact Number** (String)
4. **Snacks** (Boolean)
5. **Preferred Format** (String)
6. **Booking Reference ID** (Symbol)
7. **Total Cost** (Number)
8. **Coupon Code** (String, optional)
9. **Booking Status** (Null or String, initially null indicating no status)
10. **Customer email** (String)
11. **Address** (String)

Indicative content 1.4: Application of JavaScript Operators

Duration: 6 hrs

Practical Activity 1.4.1: Applying JavaScript Operators

Task:

1: Read carefully the following task:

Write a program that will prompts the user to enter two numbers and an operation (+, -, *, /) and then performs the calculation based on the input.

2: Follow carefully as trainer demonstrating to trainees how to apply datatypes

3: Perform the provided task by applying the knowledge learn from trainer's demonstration

4: Listen carefully and take notes wherever is necessary

5: Ask clarifying question if needed.

6: Ask them to read the Key readings 1.4.1 in their manuals

Key readings 1.4.1: Applying JavaScript Operators

- **Application of JavaScript operators**

What is an Operator?

In JavaScript, an **operator** is a special symbol used to perform operations on operands (values and variables). For example, 2 + 3; // 5. Here + is an operator that performs addition, and 2 and 3 are operands.

Let us take a simple expression **4 + 5 is equal to 9**. Here 4 and 5 are called **operands** and '+' is called the **operator**. JavaScript supports the following types of operators.

- Arithmetic Operators
- Comparison Operators
- Logical (or Relational) Operators
- Assignment Operators
- Conditional (or ternary) Operators

Let's have a look on all operators one by one.

Arithmetic Operators

JavaScript supports the following arithmetic operators −

Assume variable A holds 10 and variable B holds 20, then −

SN	Operator & Description
1	**+ (Addition)** Adds two operands **Ex:** A + B will give 30
2	**- (Subtraction)** Subtracts the second operand from the first **Ex:** A - B will give -10
3	*** (Multiplication)** Multiply both operands **Ex:** A * B will give 200
4	**/ (Division)** Divide the numerator by the denominator **Ex:** B / A will give 2
5	**% (Modulus)** Outputs the remainder of an integer division **Ex:** B % A will give 0
6	**++ (Increment)** Increases an integer value by one **Ex:** A++ will give 11
7	**-- (Decrement)** Decreases an integer value by one **Ex:** A-- will give 9

Note − Addition operator (+) works for Numeric as well as Strings. e.g. "a" + 10 will give "a10".

Example

The following code shows how to use arithmetic operators in JavaScript.

```
<html>
  <body>
```

```
<script type = "text/javascript">
  <!--
    var a = 33;
    var b = 10;
    var c = "Test";
    var linebreak = "<br />";

    document.write("a + b = ");
    result = a + b;
    document.write(result);
    document.write(linebreak);

    document.write("a - b = ");
    result = a - b;
    document.write(result);
    document.write(linebreak);

    document.write("a / b = ");
    result = a / b;
    document.write(result);
    document.write(linebreak);

    document.write("a % b = ");
    result = a % b;
    document.write(result);
    document.write(linebreak);

    document.write("a + b + c = ");
    result = a + b + c;
    document.write(result);
    document.write(linebreak);
```

```
        a = ++a;
        document.write("++a = ");
        result = ++a;
        document.write(result);
        document.write(linebreak);

        b = --b;
        document.write("--b = ");
        result = --b;
        document.write(result);
        document.write(linebreak);
      //-->
    </script>

    Set the variables to different values and then try...
  </body>
</html>
```

Output

```
a + b = 43
a - b = 23
a / b = 3.3
a % b = 3
a + b + c = 43Test
++a = 35
--b = 8
Set the variables to different values and then try...
```

Comparison Operators

JavaScript supports the following comparison operators –

Assume variable A holds 10 and variable B holds 20, then –

Sr.No.	Operator & Description
1	**= = (Equal)** Checks if the value of two operands are equal or not, if yes, then the condition becomes true. **Ex:** (A == B) is not true.
2	**!= (Not Equal)** Checks if the value of two operands are equal or not, if the values are not equal, then the condition becomes true. **Ex:** (A != B) is true.
3	**> (Greater than)** Checks if the value of the left operand is greater than the value of the right operand, if yes, then the condition becomes true. **Ex:** (A > B) is not true.
4	**< (Less than)** Checks if the value of the left operand is less than the value of the right operand, if yes, then the condition becomes true. **Ex:** (A < B) is true.
5	**>= (Greater than or Equal to)** Checks if the value of the left operand is greater than or equal to the value of the right operand, if yes, then the condition becomes true. **Ex:** (A >= B) is not true.
6	**<= (Less than or Equal to)** Checks if the value of the left operand is less than or equal to the value of the right operand, if yes, then the condition becomes true. **Ex:** (A <= B) is true.

Example

The following code shows how to use comparison operators in JavaScript.

```
<html>
  <body>
    <script type = "text/javascript">
      <!--
        var a = 10;
        var b = 20;
        var linebreak = "<br />";
        document.write("(a == b) => ");
        result = (a == b);
        document.write(result);
        document.write(linebreak);

        document.write("(a < b) => ");
        result = (a < b);
        document.write(result);
        document.write(linebreak);

        document.write("(a > b) => ");
        result = (a > b);
        document.write(result);
        document.write(linebreak);

        document.write("(a != b) => ");
        result = (a != b);
        document.write(result);
        document.write(linebreak);

        document.write("(a >= b) => ");
        result = (a >= b);
        document.write(result);
```

```
         document.write(linebreak);


         document.write("(a <= b) => ");
         result = (a <= b);
         document.write(result);
         document.write(linebreak);
       //-->
     </script>
     Set the variables to different values and different operators and then try...
   </body>
</html>
```

Output

```
(a == b) => false
(a < b) => true
(a > b) => false
(a != b) => true
(a >= b) => false
a <= b) => true
Set the variables to different values and different operators and then try...
```

Logical Operators

JavaScript supports the following logical operators −

Assume variable A holds 10 and variable B holds 20, then −

Sr.No.	Operator & Description
1	**&& (Logical AND)** If both the operands are non-zero, then the condition becomes true. **Ex:** (A && B) is true.

2	**|| (Logical OR)** If any of the two operands are non-zero, then the condition becomes true. **Ex:** (A || B) is true.
3	**! (Logical NOT)** Reverses the logical state of its operand. If a condition is true, then the Logical NOT operator will make it false. **Ex:** ! (A && B) is false.

Example

Try the following code to learn how to implement Logical Operators in JavaScript.

```
<html>
  <body>
    <script type = "text/javascript">
      <!--
        var a = true;
        var b = false;
        var linebreak = "<br />";

        document.write("(a && b) => ");
        result = (a && b);
        document.write(result);
        document.write(linebreak);

        document.write("(a || b) => ");
        result = (a || b);
        document.write(result);
        document.write(linebreak);

        document.write("!(a && b) => ");
        result = (!(a && b));
        document.write(result);
        document.write(linebreak);
      //-->
    </script>
    <p>Set the variables to different values and different operators and then try...</p>
  </body>
</html>
```

Output

```
(a && b) => false
(a || b) => true
!(a && b) => true
Set the variables to different values and different operators and then try...
```

Bitwise Operators

JavaScript supports the following bitwise operators −

Assume variable A holds 2 and variable B holds 3, then −

Sr.No.	Operator & Description
1	**& (Bitwise AND)** It performs a Boolean AND operation on each bit of its integer arguments. **Ex:** (A & B) is 2.
2	**| (BitWise OR)** It performs a Boolean OR operation on each bit of its integer arguments. **Ex:** (A | B) is 3.
3	**^ (Bitwise XOR)** It performs a Boolean exclusive OR operation on each bit of its integer arguments. Exclusive OR means that either operand one is true or operand two is true, but not both. **Ex:** (A ^ B) is 1.

4	**~ (Bitwise Not)** It is a unary operator and operates by reversing all the bits in the operand. **Ex:** (~B) is -4.
5	**<< (Left Shift)** It moves all the bits in its first operand to the left by the number of places specified in the second operand. New bits are filled with zeros. Shifting a value left by one position is equivalent to multiplying it by 2, shifting two positions is equivalent to multiplying by 4, and so on. **Ex:** (A << 1) is 4.
6	**>> (Right Shift)** Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand. **Ex:** (A >> 1) is 1.
7	**>>> (Right shift with Zero)** This operator is just like the >> operator, except that the bits shifted in on the left are always zero. **Ex:** (A >>> 1) is 1.

Example

Try the following code to implement Bitwise operator in JavaScript.

```
<html>
  <body>
    <script type = "text/javascript">
      <!--
        var a = 2; // Bit presentation 10
        var b = 3; // Bit presentation 11
        var linebreak = "<br />";

        document.write("(a & b) => ");
        result = (a & b);
        document.write(result);
        document.write(linebreak);

        document.write("(a | b) => ");
        result = (a | b);
        document.write(result);
        document.write(linebreak);

        document.write("(a ^ b) => ");
        result = (a ^ b);
        document.write(result);
        document.write(linebreak);

        document.write("(~b) => ");
        result = (~b);
        document.write(result);
        document.write(linebreak);

        document.write("(a << b) => ");
        result = (a << b);
        document.write(result);
        document.write(linebreak);

        document.write("(a >> b) => ");
        result = (a >> b);
        document.write(result);
        document.write(linebreak);
      //-->
    </script>
    <p>Set the variables to different values and different operators and then try...</p>
```

```
  </body>
</html>
```

```
(a & b) => 2
(a | b) => 3
(a ^ b) => 1
(~b) => -4
(a << b) => 16
(a >> b) => 0
```

Set the variables to different values and different operators and then try...

Assignment Operators

JavaScript supports the following assignment operators –

Sr.No.	Operator & Description
1	**= (Simple Assignment)** Assigns values from the right side operand to the left side operand **Ex:** C = A + B will assign the value of A + B into C
2	**+= (Add and Assignment)** It adds the right operand to the left operand and assigns the result to the left operand. **Ex:** C += A is equivalent to C = C + A
3	**−= (Subtract and Assignment)** It subtracts the right operand from the left operand and assigns the result to the left operand. **Ex:** C -= A is equivalent to C = C - A
4	***= (Multiply and Assignment)** It multiplies the right operand with the left operand and assigns the result to the left operand. **Ex:** C *= A is equivalent to C = C * A
5	**/= (Divide and Assignment)**

	It divides the left operand with the right operand and assigns the result to the left operand. **Ex:** C /= A is equivalent to C = C / A
6	**%= (Modules and Assignment)** It takes modulus using two operands and assigns the result to the left operand. **Ex:** C %= A is equivalent to C = C % A

Note − Same logic applies to Bitwise operators so they will become like <<=, >>=, >>=, &=, |= and ^=.

Example

Try the following code to implement assignment operator in JavaScript.

```
<html>
<body>
<script type = "text/javascript">
<!--
var a = 33;
var b = 10;
var linebreak = "<br />";

document.write("Value of a => (a = b) => ");
result = (a = b);
document.write(result);
document.write(linebreak);

document.write("Value of a => (a += b) => ");
result = (a += b);
document.write(result);
document.write(linebreak);

document.write("Value of a => (a -= b) => ");
```

```
result = (a -= b);
document.write(result);
document.write(linebreak);

document.write("Value of a => (a *= b) => ");
result = (a *= b);
document.write(result);
document.write(linebreak);

document.write("Value of a => (a /= b) => ");
result = (a /= b);
document.write(result);
document.write(linebreak);

document.write("Value of a => (a %= b) => ");
result = (a %= b);
document.write(result);
document.write(linebreak);
//-->
</script>
<p>Set the variables to different values and different operators and then try...</p>
  </body>
</html>
```

Output

```
Value of a => (a = b) => 10

Value of a => (a += b) => 20

Value of a => (a -= b) => 10

Value of a => (a *= b) => 100

Value of a => (a /= b) => 10

Value of a => (a %= b) => 0

Set the variables to different values and different operators and then try...
```

Miscellaneous Operator

We will discuss two operators here that are quite useful in JavaScript: the **conditional operator** (? :) and the **typeof operator**.

Conditional Operator (? :)

The conditional operator first evaluates an expression for a true or false value and then executes one of the two given statements depending upon the result of the evaluation.

Sr.No.	Operator and Description
1	**? : (Conditional)** If Condition is true? Then value X : Otherwise value Y

Example

Try the following code to understand how the Conditional Operator works in JavaScript.

```
<html>
<body>
<script type = "text/javascript">
<!--
var a = 10;
var b = 20;
var linebreak = "<br />";

document.write ("((a > b) ? 100 : 200) => ");
result = (a > b) ? 100 : 200;
document.write(result);
document.write(linebreak);

document.write ("((a < b) ? 100 : 200) => ");
result = (a < b) ? 100 : 200;
document.write(result);
document.write(linebreak);
//-->
</script>
<p>Set the variables to different values and different operators and then try...</p>
</body>
</html>
```

Output

```
((a > b) ? 100 : 200) => 200
((a < b) ? 100 : 200) => 100
Set the variables to different values and different operators and then try...
```

typeof Operator

The **typeof** operator is a unary operator that is placed before its single operand, which can be of any type. Its value is a string indicating the data type of the operand.

The typeof operator evaluates to "number", "string", or "boolean" if its operand is a number, string, or boolean value and returns true or false based on the evaluation.

Here is a list of the return values for the **type of** Operator.

Type	String Returned by typeof
Number	"number"
String	"string"
Boolean	"boolean"
Object	"object"
Function	"function"
Undefined	"undefined"
Null	"object"

Example

The following code shows how to implement **typeof** operator.

```
<html>
<body>
<script type = "text/javascript">
<!--
var a = 10;
var b = "String";
var linebreak = "<br />";

result = (typeof b == "string" ? "B is String" : "B is Numeric");
document.write("Result => ");
document.write(result);
document.write(linebreak);

result = (typeof a == "string" ? "A is String" : "A is Numeric");
document.write("Result => ");
document.write(result);
document.write(linebreak);
//-->
</script>
<p>Set the variables to different values and different operators and then try...</p>
  </body>
</html>
```

Output

```
Result => B is String
Result => A is Numeric
```

Points to Remember

- JavaScript operators are special symbols used in scripts to perform operations on operands, such as arithmetic calculations, logical comparisons, or value assignments. It plays a crucial role in controlling the flow and processing of data within the language.
- There are different types of JavaScript operators:
  - ✓ Arithmetic Operators
  - ✓ Assignment Operators
  - ✓ Comparison Operators
  - ✓ String Operators
  - ✓ Logical Operators

- ✓ Bitwise Operators
- ✓ Ternary Operators

Application of learning 1.4.

An application based on Creating a simple calculator application that performs basic arithmetic operations (addition, subtraction, multiplication, and division) on numbers. To do so represents numerical values, including integers and floating-point numbers, Used to store the operands and results of calculations. Represents textual data, used to display the input and output values, as well as any error messages. Represents true or false values, used to control conditional logic (e.g., checking for invalid input or handling division by zero).

Indicative content 1.5: Write an Algorithm

Duration: 6 hrs

Theoretical Activity 1.5.1: Description of key concepts of Algorithm fundamentals

Tasks:

1: Answer the following questions:

i. Define the following terms:

a) Language

b) Programming language

c) Source code

d) Machine code

e) Algorithm

f) IDE

ii. Differentiate compiler from interpreter

iii. Discuss about types of algorithms

iv. Describe characteristics/qualities of a good algorithm

2: Write answers on the flipchart/paper.

3: Discuss on provided answers and choose correct answers.

4: Listen attentively, take notes and ask question where is necessary

5: Read the key reading 1.5.1 in the trainee manual

Key readings 1.5.1.: Description of key concepts of Algorithm fundamentals

1. Key concepts of algorithm fundamentals

- **Language** is a mode of communication that is used **to share ideas, opinions with each other**. For example, if we want to teach someone, we need a language that is understandable by both communicators.

To communicate with computers, programmers also need a language called Programming language.

- **Programming language:** Programming language is a formal language used by computer programmer to create software programs and instruct a computer

A programming language is a computer language that is **used by programmers (developers) to communicate with computers.**

Natural languages are the languages that people speak, such as English, Kinyarwanda or French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications.

- **Computer Programmer:** A programmer is a person who writes computer programs.
- **Source code:** Source code is the set of instructions written by a programmer using a computer programming language. This code is later translated into machine language by a compiler.
- **Machine code:** Also called **machine language**, is a computer language that is directly understandable by a computer's CPU. It is composed of digital binary numbers (zeros and ones). and it is the language into which all programs must be converted before they can be run.

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 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.

- **Algorithm:** It is a step-by-step procedure (well-defined instructions) to solve a given problem in order to get an expected result.

It can be translated into a programming language in order to produce the result.

Algorithm is not the complete code or program; it is just the core logic (solution) of a problem.

- **IDE (Integrated Development)**

An IDE is a set of tools like text editor, compiler, debugger etc., that all work together to facilitate software programming.

An editor is simply one tool among tools of IDE that is designed to edit text.

Editor

Editors or **Text editors** are software programs that enable the user to create and edit text files.

Ex: Dev++, Code-blocks, Dreamweaver, Notepad++, Sublime

In the field of programming, the term editor usually refers to source code editors that include many special features for writing and editing code.

Features normally associated with text editors are moving the cursor, deleting, replacing, pasting, finding and replacing, saving etc.

- **High level language and Low-level language**

Generally, we 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 high-level 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.

After the source code for a program has been written by a Programmer, 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 (run) by the operating system any time.

- **Compiler and Interpreter**

Compiler: A compiler is a computer program that translates source code into machine code

Interpreter: It is a computer program that executes instructions written in a programming/scripting language without requiring to be compiled into machine language.

A compiler translates the entire source code into machine code before execution, resulting in faster execution since no translation is needed during runtime. On the other hand, an interpreter translates code line by line during execution, making it easier to detect errors but potentially slowing down the program.

A **compiler** is defined as a software that transforms an entire set of source code into object code and saves it as a file before executing it. Conversely, an interpreter converts and executes source code line by line without saving it and points out errors along the way.

2. The difference between an interpreter and a compiler is given bellow

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	It takes large amount of time to analyze the source code
Overall execution time is slow	Overall execution time is comparatively faster
No intermediate object code is generated	Generates intermediate object code
Execution requires low memory	Execution requires more memory
Continues translating the program until the first error is meet, in which case it stops. Hence debugging is easy	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard
Programming language like python, Ruby, php and Perl use interpreters	Programming language like C,C++, C# use compiler

3. Types of algorithms

There are many types of algorithms but the most important and fundamental algorithms that you must are discussed in this article.

- **Brute Force Algorithm:**

This is the most basic and simplest type of algorithm.

A Brute Force Algorithm is the straightforward approach to a problem i.e., the first approach that comes to our mind on seeing the problem.

- **Recursive Algorithm:**

In recursion, a problem is solved by breaking it into subproblems of the same type and calling own self again and again until the problem is solved with the help of a base condition.

Examples of a problem that is solved using recursive algorithms are Factorial of a Number.

a) Divide and Conquer Algorithm:

- **Purpose:** To break down a problem into smaller, easier-to-solve subproblems, solve those subproblems, and then combine their solutions to solve the original problem.

Some common problem that is solved using Divide and Conquers Algorithms are

- Binary Search
- Merge Sort
- Quick Sort
- Strassen's Matrix Multiplication, etc.

b) Dynamic Programming Algorithms:

Its idea is to solve problems by breaking them down into overlapping subproblems and storing the solutions to those subproblems to avoid recalculation.

The following problems can be solved using the Dynamic Programming algorithm Fibonacci Sequence calculation, Longest Common Subsequence problem

Knapsack Problem, Weighted Job Scheduling, Floyd War shall Algorithm, etc.

c) Greedy Algorithm:

- Purpose: To make locally optimal choices at each step, hoping to reach a global optimal solution.
- Examples:

- ✓ Dijkstra's Algorithm for finding the shortest path in a graph:
- ✓ Huffman Coding for data compression
- ✓ Prim's Algorithm for finding a minimum spanning tree:

d) Backtracking Algorithm:

In Backtracking Algorithm, the problem is solved in an incremental way i.e. it is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time. Some common problems that can be solved through the Backtracking Algorithm are the Hamiltonian Cycle, M-Coloring Problem, N Queen Problem, Rat in Maze Problem, etc.

•Randomized Algorithm:

In the randomized algorithm, we use a random number.it helps to decide the expected outcome. The decision to choose the random number so it gives the immediate benefit Some common problems that can be solved through the Randomized Algorithm are Quicksort: In Quicksort we use the random number for selecting the pivot.

•Sorting Algorithm:

The sorting algorithm is used to sort data in maybe ascending or descending order. It’s also used for arranging data in an efficient and useful manner. Some common problems that can be solved through the sorting Algorithm are Bubble sort, insertion sort, merge sort, selection sort, and quick sort are examples of the Sorting algorithm.

•Searching Algorithm:

The searching algorithm is the algorithm that is used for searching the specific key in particular sorted or unsorted data. Some common problems that can be solved through the Searching Algorithm are Binary search or linear search is one example of a Searching algorithm.

• Hashing Algorithm:

Hashing algorithms work the same as the Searching algorithm but they contain an index with a key ID i.e a key-value pair. In hashing, we assign a key to specific data. Some common problems can be solved through the Hashing Algorithm in password verification.

4. Characteristics/qualities of a good algorithm

- Efficiency: A good algorithm should perform its task quickly and use minimal resources.
- Correctness: It must produce the correct and accurate output for all valid inputs.
- Clarity: The algorithm should be easy to understand and comprehend, making it maintainable and modifiable.
- Scalability: It should handle larger data sets and problem sizes without a significant decrease in performance.
- Reliability: The algorithm should consistently deliver correct results under different conditions and environments.
- Optimality: Striving for the most efficient solution within the given problem constraints.
- Robustness: Capable of handling unexpected inputs or errors gracefully without crashing.
- Adaptability: Ideally, it can be applied to a range of related problems with minimal adjustments.
- Simplicity: Keeping the algorithm as simple as possible while meeting its requirements, avoiding unnecessary complexity.

Practical Activity 1.5.2: Developing an algorithm using structured English

Task:

1: Read carefully the following task:

By using structured English, write an algorithm of the program that will help a user to calculate the sum and the average of three numbers.

2: Follow carefully as trainer demonstrating to trainees how to Develop an algorithm using structured English

3: Perform the provided task by applying the knowledge learn from trainer's demonstration

4: Listen carefully and take notes wherever is necessary

5: Ask clarifying question if needed.

6: Read the Key readings 1.5.2 in your manuals

Key readings 1.5.2: Developing an algorithm using structured English

Structured English is the use of the **English language** with the **syntax** of **structured programming** to communicate the design of a computer program to non-technical users by breaking it down into logical steps using straightforward English words. Structured English gives aims to get the benefits of both the programming logic and natural language: program logic helps to attain precision, whilst natural language helps with the familiarity of the spoken word.

Structure English is derived from structured programming language which gives more understandable and precise description of process. It is based on procedural logic that uses construction and imperative sentences designed to perform operation for action.

- It is best used when sequences and loops in a program must be considered and the problem needs sequences of actions with decisions.
- It does not have strict syntax rule. It expresses all logic in terms of sequential decision structures and iterations.

For example, see the following sequence of actions −

```
if customer pays advance
then
Give 5% Discount
else
if purchase amount >=10,000
then
if the customer is a regular customer
then Give 5% Discount
```

```
else No Discount
end if
else No Discount
end if
end if
```

Following are the algorithm of the sum and the average of three numbers is given below.

Explanation:

Step 1: Start

Step 2: Read the three number suppose "a","b","c" form the user.

Step 3: Declared a variable "sum" and "Avg".

Step 4 : sum=a+b+c;

Step 5: Avg=sum/3

Step 6: Display "sum " and "Avg".

Step 7: End.

Find the Largest Number Using if-else Statements

The idea is to use the compound expression where each number is compared with the other two to find out which one is the maximum of them.

Algorithm

- Check if A is greater than or equal to both B and C, A is the largest number.
- Check if B is greater than or equal to both A and C, B is the largest number.
- Check if C is greater than or equal to both A and B, C is the largest number.

Practical Activity 1.5.3: Developing an algorithm using pseudocode

Task:

1: Read carefully the following task:

By using pseudocode, write an algorithm of the program that will help a user to calculate the sum and the average of three numbers.

2: Follow carefully as trainer demonstrating to trainees how to Develop an algorithm using pseudo code

3: Perform the provided task by applying the knowledge learn from trainer's demonstration

4: Listen carefully and take notes wherever is necessary

5: Ask clarifying question if needed.

6: Read the Key readings 1.5.3 in your manuals

Key readings 1.5.3: Developing an algorithm using pseudocode

1. Structure of an algorithm

An algorithm is made mainly of the following parts:

⇨ The beginning of an algorithm
⇨ The variable declaration line.
⇨ The instructions part
⇨ The end

Example:

```
Var A as Integer
Start
A<- 5
End
```

Explanations:

Var A as Integer is the variable declaration line

Start	marks the beginning of an algorithm
A←5	the instructions part
End	marks the end of an algorithm

2. Declaration of variables, data types and operators

2.1 Variable

A variable is a memory zone which is used to store data. It is characterized by a name, an address and a data type.

✓ **Variable name**: a variable name helps to distinguish it from other variables,
✓ **Variable address**: helps to locate it in the memory,

✓ **Variable data type**: helps to know the operations allowed to be performed on it and the size it occupies in the memory.Variable declaration

A variable to be used must first be declared. Declare a variable; means create it by giving it a name and a data type.

2.2 Advantages of a variable in an algorithm

- Variable used to store information to be referenced and manipulated in a computer program.
- Provide a way of labeling data with a descriptive name
- It is a container that hold or store data in memory
- Tells compiler how much space is required for the data.

2.3 Read function (Inputs) and Write function (Outputs)

a. Read function (Inputs)

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( )

Example:

Write an algorithm which receives a number entered by a user.

Answer:

```
Var A as Integer Start
read(A)
End
```

b. Write function

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

Syntax of write function:

Write( )

Example:

Write an algorithm which displays a value stored in a variable.

Answer:

```
Var B as Integer Start
B←5
Write("The content of the variable is: ")
Write(B)
```

End

Notes: Here is another way of writing an algorithm

Algorithm to add two numbers

We will need

```
Input: Two numbers, num1 and num2
Output: Sum of num1 and num2
1. Start
2. Read num1 from the user
3. Read num2 from the user
4. Add num1 and num2: sum = num1 + num2
5. Display the sum
```

3. Flow of control in algorithm or control structure of algorithm

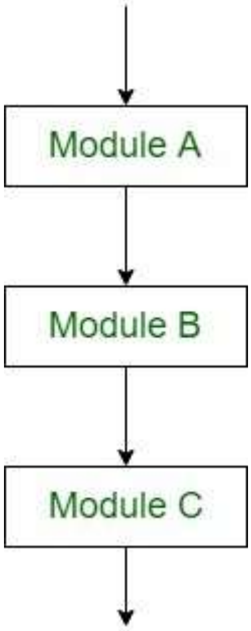
Algorithms use control flow to make decisions about which order to do things. They can repeat actions or start new actions based on new information. Computer programs use sequence, selection and iteration to control the flow of the program.

There are three basic types of control structure, or flow of control or control structure, known as:

1. **Sequential structure** (Sequence logic, or sequential flow)
2. **Conditional structure** (Selection logic, or conditional flow)
3. **Iterating/Looping structure** (Iteration logic, or repetitive flow)

3.1 Sequential structure

In algorithms, a sequential structure refers to a basic control flow where actions or instructions are executed one after the other in the order they are written. This structure does not involve any decision-making or loops;



Example of a Sequential Structure:

Consider an algorithm that calculates the sum of two numbers and prints the result:

1. Start
2. Input number1
3. Input number2
4. sum = number1 + number2
5. Output sum
6. End

3.2 Conditional Structure

Very often when you write code, you want to perform different actions for different conditions. You can use conditional statements in your algorithm/code to do this.

In Algorithm, we have the following conditional statements:

- **If statement -** executes some code if one condition is true
- **If...else statement** - executes some code if a condition is true

and another code if that condition is false

- **If...elseif... statement** - executes different codes for more than two conditions
- **Switch statement** - selects one of many blocks of code to be executed

a) If statement
Syntax

```
If (condition)then
            Block of
         code
             End if.
```

NB: **If statement** is used to make a decision. The block of instructions following **If**, executes if the decision is true, and the block does not execute otherwise.

Example 3:
Write an algorithm which receives a number and informs the user when it is positive.

Answer:

```
Var a as integer Start
Write("enter a number") Read(a)
If (a>0) then
Write("the number is positive")
End if
End
```

To this question when the condition evaluates true it displays the instruction: **The number is positive** but when it evaluates for false **it displays nothing**

b. If…. else statement
The **if…else statement** is used to make a decision and gives the alternative when the condition evaluates to false. The block of instructions following the **if** executes if the decision is true, and the block after **else** when the condition evaluates to false.

Syntax

```
If(condition) then
Block of code
Else
Block of code
End if
```

```
End
```

Example 1:

Write an algorithm which receives a number and informs the user whether it is positive or negative.

Answer:

```
Start
Var a as integer
Write ("enter a number")
Read(a)
If (a>0) then
Write ("the number is positive")
else
Write ("the number is negative")
End if
End
```

To this question when the condition evaluates true it displays the message the number is positive when it evaluates to false it displays the message the number is negative.

a. If …ElseIf….

If statement may be used inside another if statement, in such case we call it a nested if.

Example:

Write an algorithm which receives student note and it displays the grade as follows: Note form 16 and above: Grade A

Note14-16 : Grade B

Note12-14 : Grade C

Note below 12 : Grade D

Answer:

```
Start
Var Note as integer
Write("Enter the Note")
```

```
Read(Note)
If(Note>=16) then
Write("Grade A")
Elseif(Note>=14) then
Write("GradeB")
Else if (Note>=12)then
Write("GradeC")
Else Write("GradeD")
End if
End
```

b. Multiple choice using 'switch'

A multiple-choice using **switch** helps to solve the problem caused by nested if statement in case there are many conditions to be tested. Switch receives a variable then it evaluates it using several **Case**.

Syntax:

```
Menu explaining to the user how to make a choice
Switch(variable)
Case 1
Instructions
Case 2
Instructions
   …….
  ……..
Case n
Instructions
Default
Instruction
End switch
```

Example:

Write an algorithm which receives note and displays the student's grade.

16 and above: Grade A

14-16: Grade B
12-14: Grade C
Less than 12: Grade D

Answer:

```
Var Note as integer
Start
Write("Use number to chose the range of your note")
Write("enter 1 for note ranging from 16 and above")
Write("enter 2 for note ranging from 14 to16")
Write("enter 3 for note ranging from 12 to14")
Write("enter 4 for note less than 12")
Write("enter your choice now using number:")
Read(Note)
Switch(Note)
Case1
Write("You have grade A")
Case2
Write("You have grade B")
Case3
Write("You have grade C")
Case4
Write("You have grade D")
Default
Write("Your choice is not listed, try again")
End switch
End
```

1.3 Iterative/Looping structure

What is a loop?

A **loop** is a sequence of instructions that is continually repeated until a certain condition is reached.

A loop helps to repeat instruction or block of instructions. It assists in the algorithm where you want to carry out an activity for a certain number of times.

Often when you write code, you want the same block of code to run over and over again in a row. Instead of adding several almost equal code-lines in a script, we can use loops to perform a task like this.

In Algorithm, we have the following looping statements:

while - loops through a block of code as long as the specified condition is true

do...while - loops through a block of code once, and then repeats the loop as long as the specified condition is true

for - loops through a block of code a specified number of times

foreach - loops through a block of code for each element in an array

a. Do while loop

The **do-While loop** executes its statements at least once even if the condition fails for the first time.

Syntax

```
Variable=<start value>
Do while <variable><comparison operator><end value>
Instruction or block of instructions
Variable=variable+1
Loop
```

Example:

Write an algorithm which use do while loop and displays numbers from 1 to 10

Answer:

```
Var A as integer Start
A=1
while A<=10
Write(A)
A=A+1
Loop
End
```

b. Do… while loop

Syntax:

```
<Variable>=<start value>
          do
Instruction or block of instructions
Variable=variable+1
          while<variable><comparison operator><end value>
```

Example:

Write an algorithm which use do while loop and displays numbers from 1 to 10

Answer:

```
Start
Var A as integer
A=1

Do
Write(A)
A=A+1
while A>10
End
```

c. For loop

The **for loop** is an iterative loop, it specifies some elements about the loop in one single line.

a. Setting a loop counter to an initial value.

b. End which determines whether its value has reached the number of repetitions desired.

The value of the loop counter will be increased each time (iteration), and segment within the loop will be executed.

Syntax:

```
for(<initialize counter>to <end of repetitions desired>) do
  //bloc of instructions
        End for
```

Example:

Write an algorithm which ask a user to enter a number and it displays the 10 next numbers.

Solution:

```
Start
Var I,A as integer
  Write (“Enter a number”)
   Read(I)
   For (A=1 to 10) do
   I<-I+1
   Write(I)
 End for
 End
```

Loops in Loops

Loops in loops refer to what we call nested loops. These are loops that are such that when one increment by one the other continues inside the first.

```
start
Var I, J As Integer
 For I = 1 To 9 do
 For J = 1 To I do
       Write(J)
 End for
 End for
End
```

Theoretical Activity 1.5.1.2: Description of program flowchart

Tasks:

1: Read and answer the following questions:

i. What is flowchart

ii. Talk about advantages of flowchart

iii. Describe elements of flowchart

iv. Tools used to draw flowchart

v. Discuss flowchart best practices

2: Present their findings in front of the class.

3: Ask clarifying question where is needed

4: Read the key reading 1.5.1.2 in the trainee manual

Key readings 1.5.1.2: Description of program flowchart

- **Designing a Flowchart**

1. Introduction to Flowchart

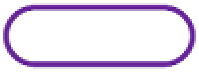
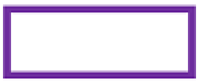
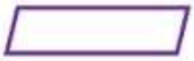
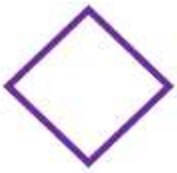
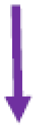
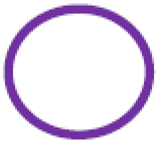
Flowchart is diagrammatic /Graphical representation of sequence of steps to solve a problem.

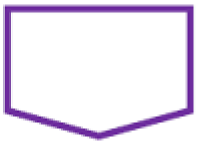
2. Advantages of flowchart:

- ✓ Flowchart is an excellent way of communicating the logic of a program.
- ✓ Easy and efficient to analyze problem using flowchart.
- ✓ During program development cycle, the flowchart plays the role of a blueprint, which makes program development process easier.
- ✓ After successful development of a program, it needs continuous timely maintenance during the course of its operation. The flowchart makes program or system maintenance easier.
- ✓ It is easy to convert the flowchart into any programming language code.

3. Description of Elements of Flowchart/ Flowchart Symbols

Here is a chart for some of the common symbols used in drawing flowcharts.

Symbol	Symbol Name	Purpose
	Start/Stop	Used at the beginning and end of the algorithm to show start and end of the program.
	Process	Indicates processes like mathematical operations.
	Input/ Output	Used for denoting program inputs and outputs.
	Decision	Stands for decision statements in a program, where answer is usually Yes or No.
	Arrow	Shows relationships between different shapes.
	On-page Connector	Connects two or more parts of a flowchart, which are on the same page.

	Off-page Connector	Connects two parts of a flowchart which are spread over different pages.

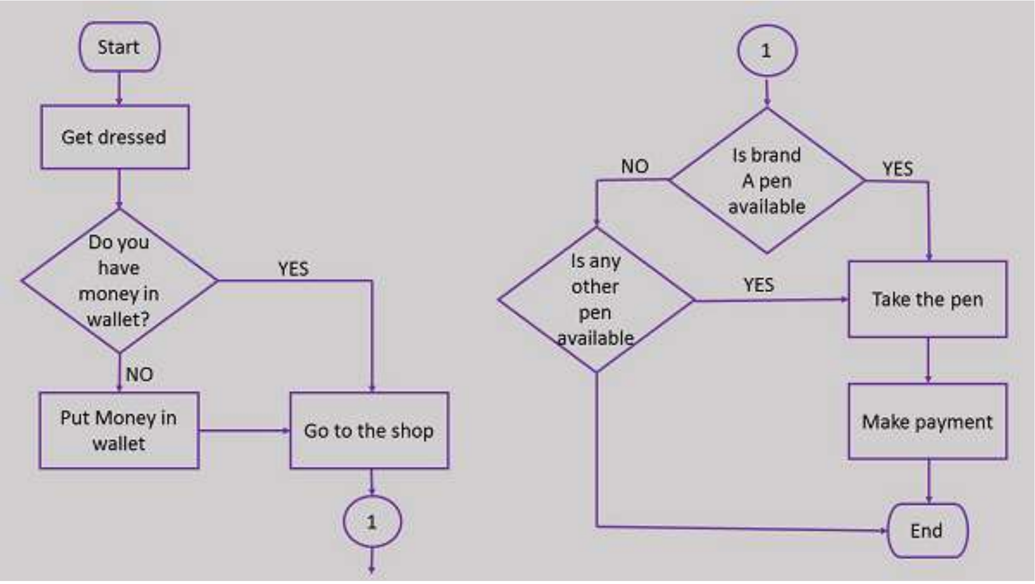
2. Guidelines for Developing Flowcharts

These are some points to keep in mind while developing a flowchart −

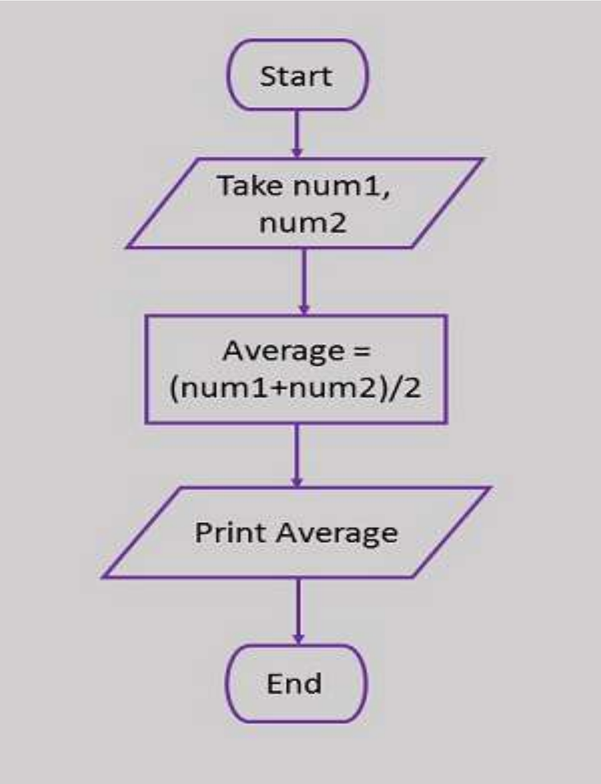
- Flowchart can have only one start and one stop symbol
- On-page connectors are referenced using numbers
- Off-page connectors are referenced using alphabets
- General flow of processes is top to bottom or left to right
- Arrows should not cross each other

Example Flowchart

Here is the flowchart for going to the market to purchase a pen.



Here is a flowchart to calculate the average of two numbers.



1. Selection/conditional structures flowchart

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

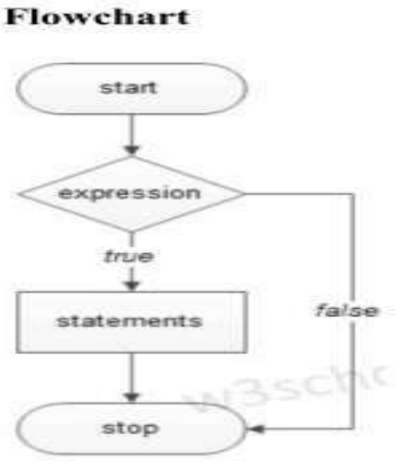
e. Switch statement (will be seen later)

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.

Syntax

```
Start
  If (condition) then
      Instructions
End if
End
```

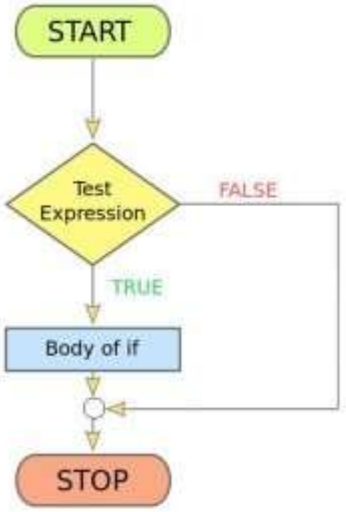


1.1. IF/ELSE statement

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

Syntax

```
Start
  If (condition) then
      Instructions
Else
Instructions
End if
End
```



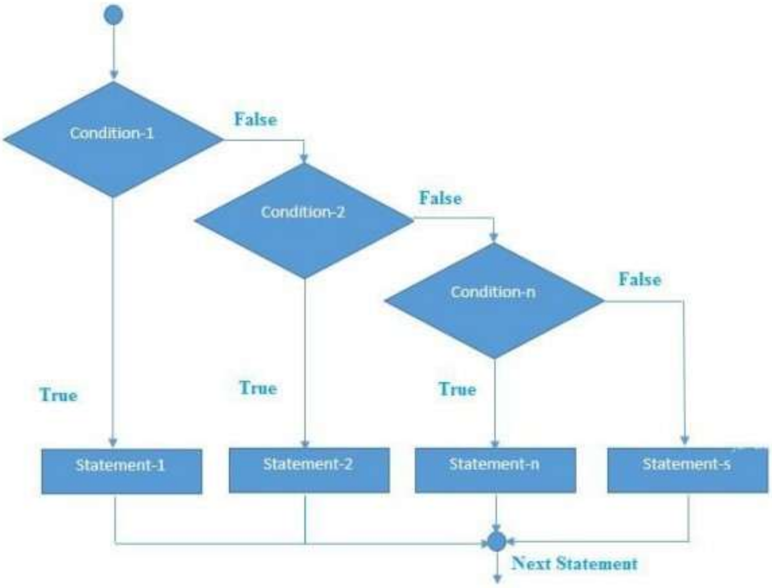
1.2. ELSE/ IF statement

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

Syntax

```
 Start
```

```
If (condition) then
    Instructions
Else if (condition) then
Instructions
Else if (condition) then
Instructions
…
Else [optional]
Instructions
End if
End
```

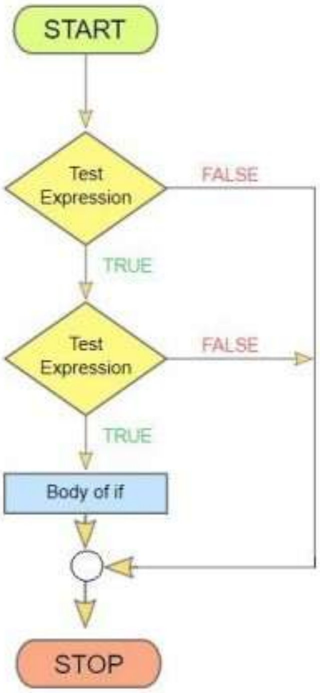


3. NESTED IF statement

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.

Syntax

```
Start
    If (condition)
        If (condition) then
        Instructions
        …
         Else
         Instructions
         End if
    Else [optional]
    Instructions
    End if
End
```



Arrange the steps in a logical order, typically from top to bottom or left to right. Ensure that the flow of the process is easy to follow and that decision points are clearly indicated.

6. Use Connectors:

For complex flowcharts, use connectors (small circles or labeled arrows) to indicate continuation points, especially when the flowchart spans multiple pages.

7. Label Decision Paths:

Clearly label the paths branching from decision points to indicate the conditions or outcomes leading to each path.

8. Provide a Key or Legend:

If your flowchart uses specialized symbols or colors, include a key or legend to explain their meanings.

Practical Activity 1.5.4: Draw a basic flow chart in draw.io

Task:

1: Read carefully the following task:

By using draw.io, draw a flowchart of the program that will help a user to calculate the sum and the average of three numbers.

2: Follow carefully as trainer demonstrating to trainees how to draw a flowchart of the program

3: Perform the provided task by applying the knowledge learn from trainer's demonstration

4: Listen carefully and take notes wherever is necessary

5: Ask clarifying question if needed.

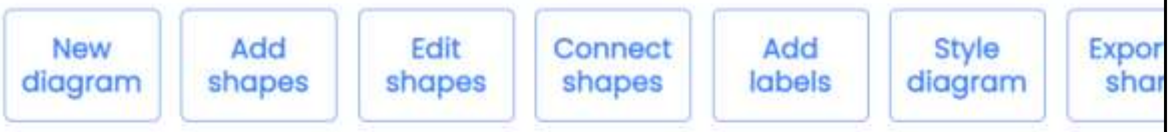
6: Ask them to read the Key readings 1.5.5 in their manuals

Key readings 1.5.5: Draw a basic flow chart in draw.io

- **Draw a basic flow chart in draw.io**

  You can create many different types of diagrams with draw.io and our online diagram editor. To learn how to use the editor, let's start with a basic flowchart to document a process.

✓ **Steps to draw a basic flow chart in draw.io:**

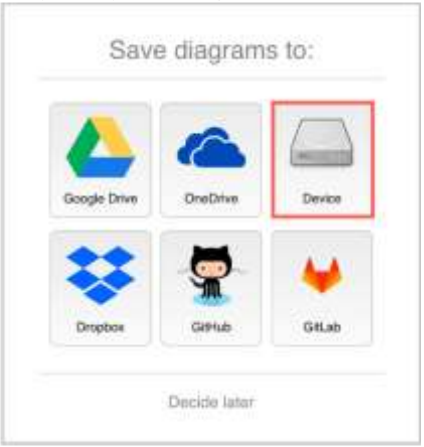


1. Create a new blank diagram
2. Add shapes to the drawing canvas
3. Move, resize, rotate, and delete shapes
4. Connect shapes
   - Draw a floating connector
   - Draw a fixed connector
   - Change the path of a connector
5. Add labels
6. Style your flow chart
7. Export and share your flow chart

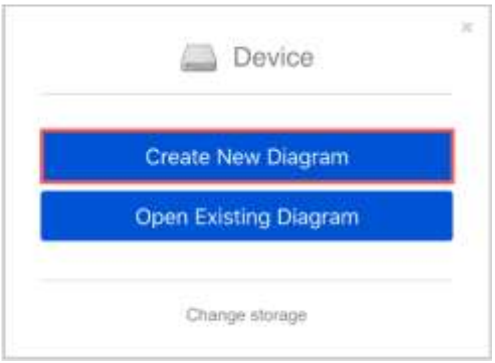
Details:

Step 1: Create a new blank diagram

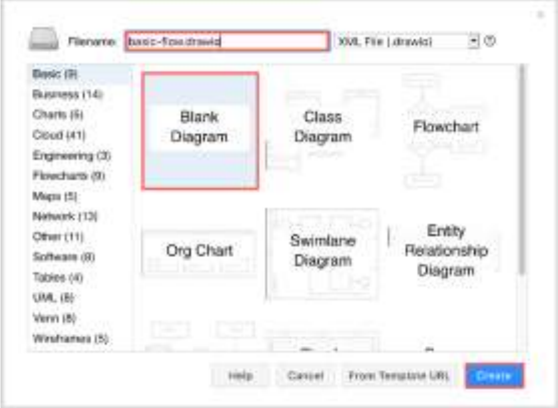
1. Go to the online draw.io editor.
2. Select *Device* to save your flow chart to your device. **Note:** You can select another location to store your diagram file, but you may be prompted to grant the editor permission if you select a cloud storage platform.



3. Click on *Create New Diagram*.



4. In the template manager, enter a filename for your flow chart, ensure *Blank Diagram* is selected, then click *Create*.


Filename: basic-flow.drawio

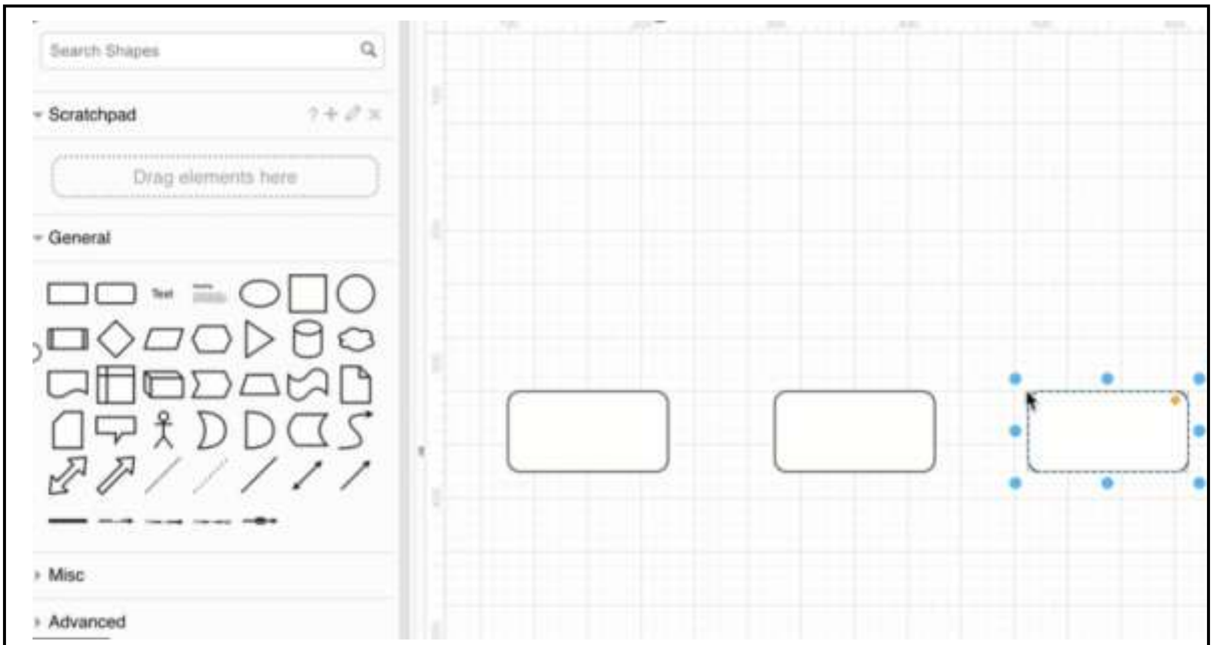
XML File (.drawio)

Step 2: Add shapes to the drawing canvas

There are a number of different ways to add shapes to the drawing canvas in draw.io.

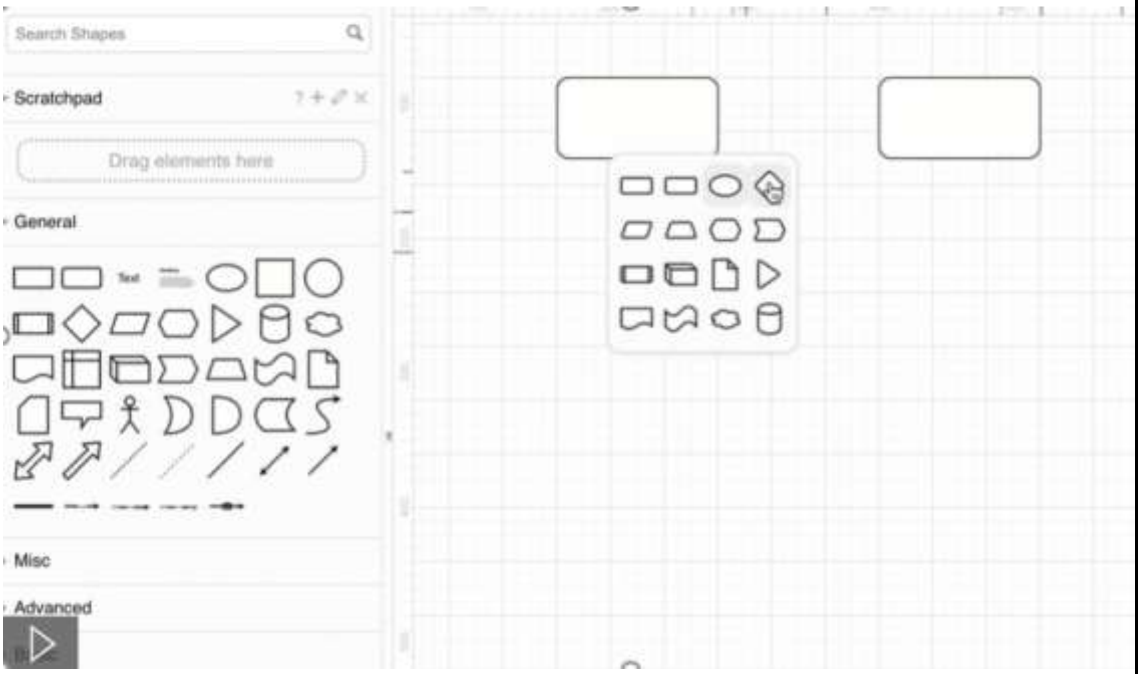
Add the first step - Use one of the following methods to add a rectangle to the drawing canvas. Rectangles represent the steps in your process.

- Click on a rectangle in the *General* shape library to add it the drawing canvas.
- Double-click on an empty area on the drawing canvas and select a rectangle shape.
- Drag a rectangle from the *General* shape library to a specific position on the drawing canvas.

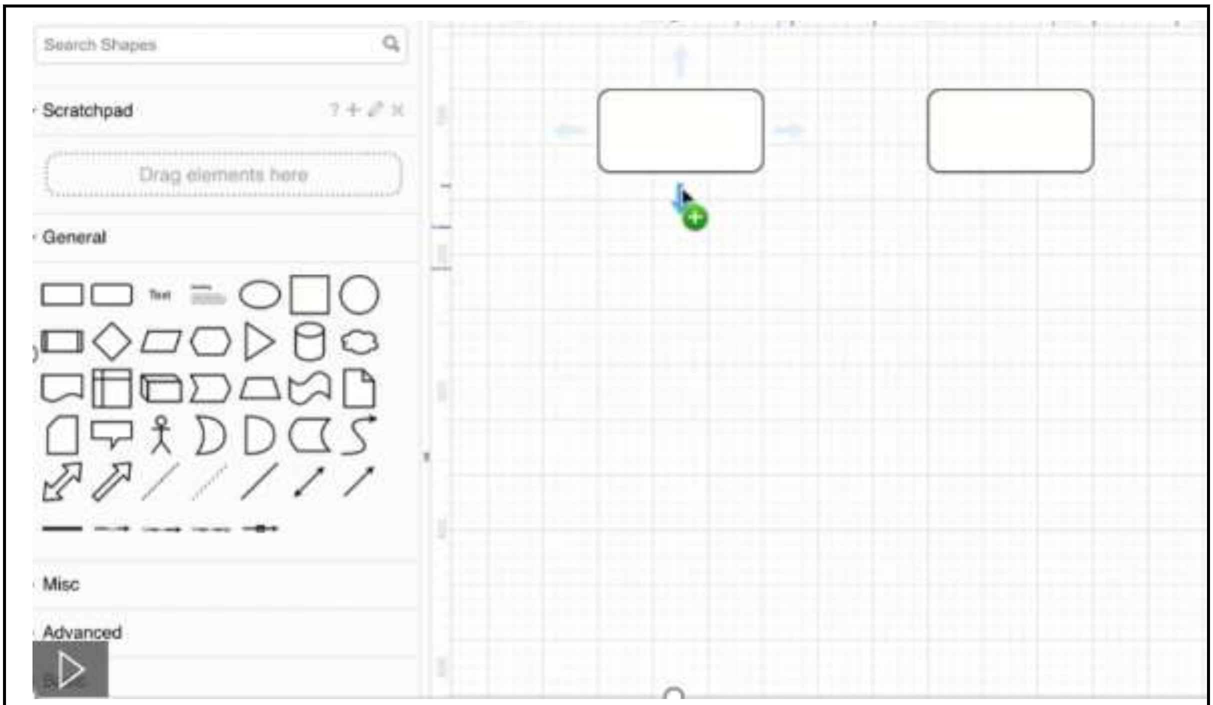


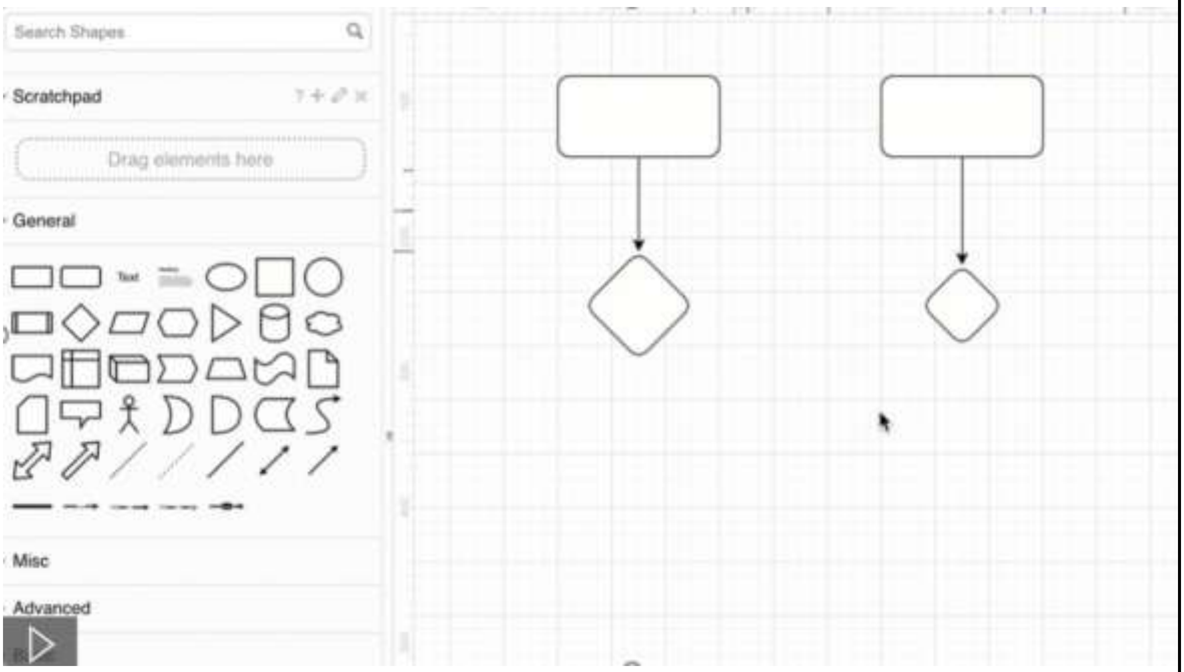
Add more shapes - Use one of the following methods to add extra shapes to represent the next steps in your process.

- Hover over the first shape you placed to see the four direction arrows. Click on one of the shapes, then select a shape to add and connect it in that direction.



- Drag a shape from the shape library and hover over an existing shape until you can see the four direction arrows. Move over one of these direction arrows, and drop the shape you have dragged. It will be added to the drawing canvas and connected in that direction.

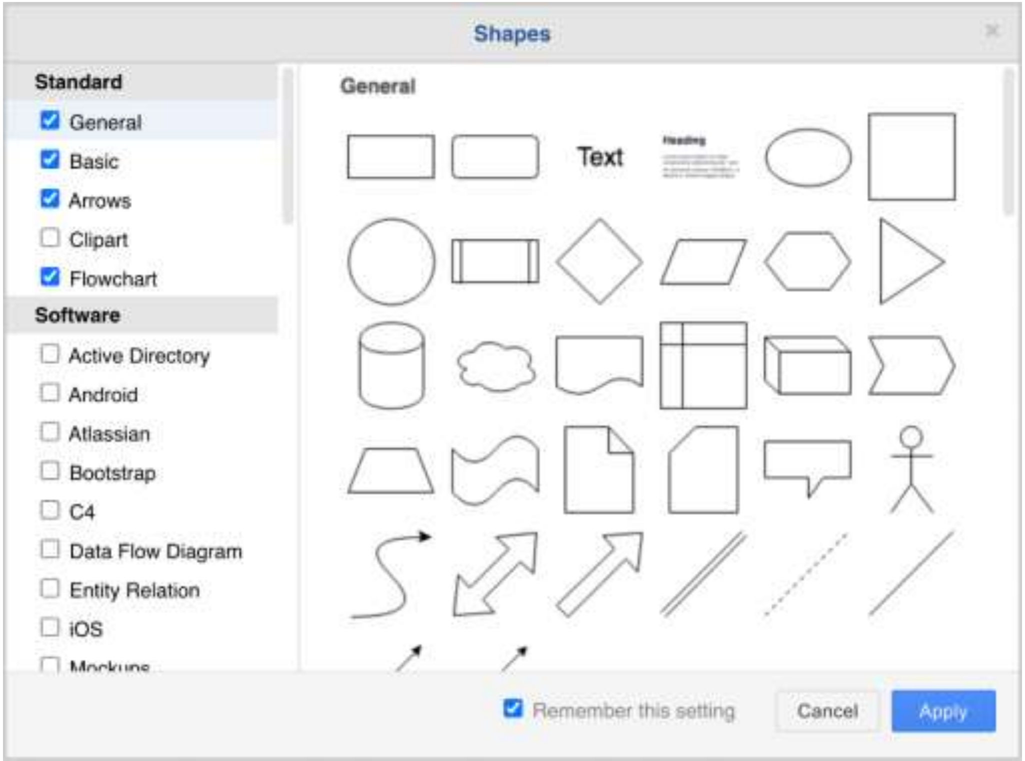




Notes: If you added multiple shapes by simply clicking on them in the shape library, you can connect them later.

In addition to the shapes in the *General* and *Advanced* shape libraries, you can use shapes from the *Flowchart* shape library.

1. Click on *More Shapes* at the bottom of the left panel.
2. Click on the checkbox next to the *Flowchart* shape library's name to enable it.



3. Click *Apply*. The *Flowchart* shape library is now available in the left panel.

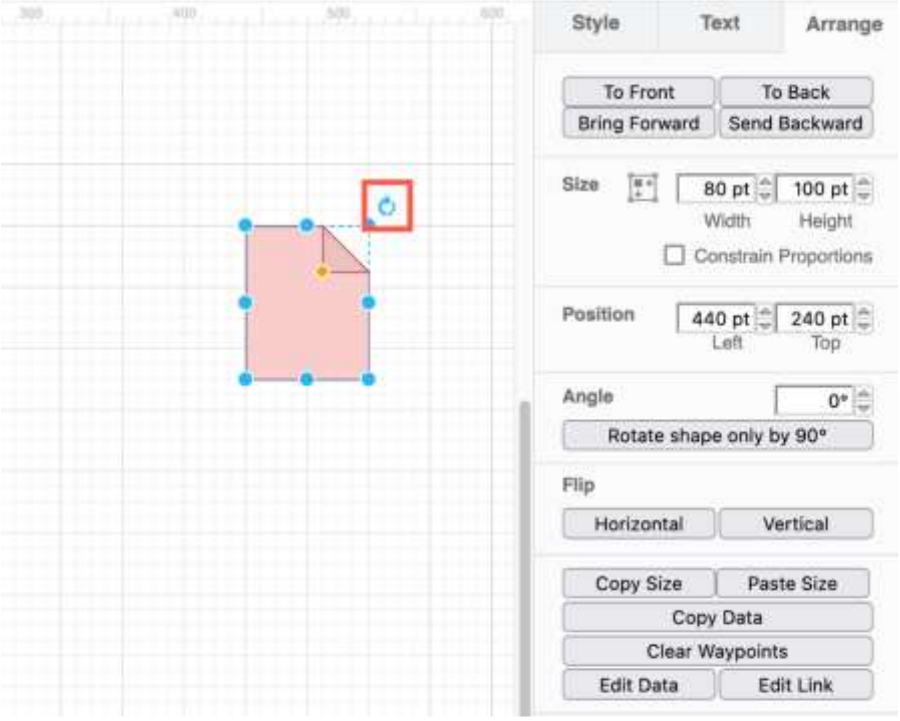
Step 3: Move, resize, rotate, and delete shapes

To select a shape, click on it. To select multiple shapes, hold down Shift or Cmd and click on them.

Move - Select and drag a shape that is on the drawing canvas to another position.

Resize - Select a shape. Drag any of the round 'grab' handles to make the shape smaller or larger. Hold down Control when you resize shapes to keep them centred. See how to resize groups of shapes

Rotate - Select a shape. Drag the rotate grab handle (the round arrow) at the top right corner of the shape to rotate the shape around its center point.



Delete - Select a shape, then press Backspace or Delete, or click on the *Delete* tool in the toolbar.

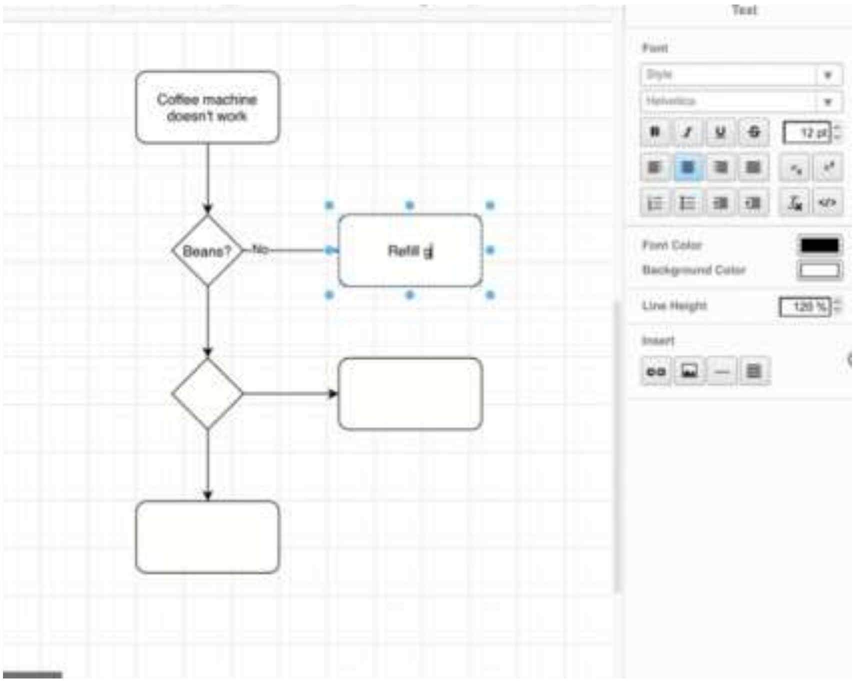
Step 4: Connect shapes

Connectors are lines that connect your shapes together and may or may not have arrows at one or both ends.

Step 5: Add labels

Short labels on shapes make it easier to understand a diagram quickly.

1. Double click on a shape. Start typing to replace the label with your own text. Alternatively, single click on a shape and start typing to add or edit the label.
2. Press *Enter* to save the label text.



Connector labels

You can add more than one label to a connector - at the source end, the target end, and in the middle.

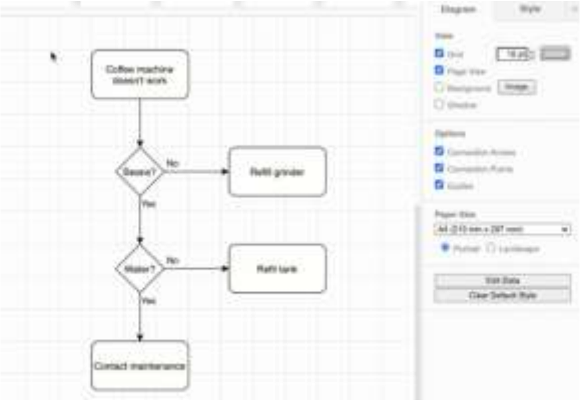
- Double click in the position you want to add a text label there.
- To reposition the connector label text, click on the label, then drag the small yellow diamond to a new position.

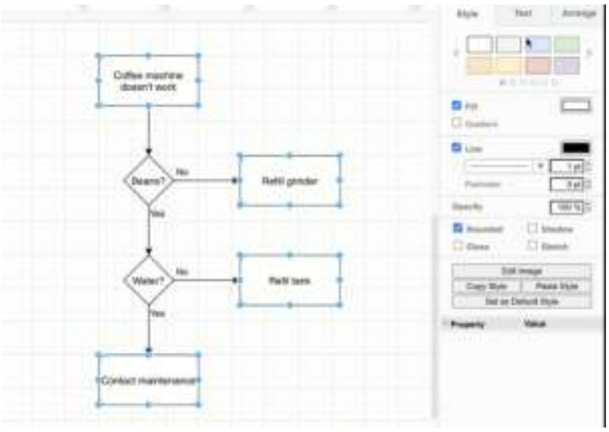
If you reroute the connector, you may need to update any labels you had manually positioned, but usually the label will move intelligently with the connector.

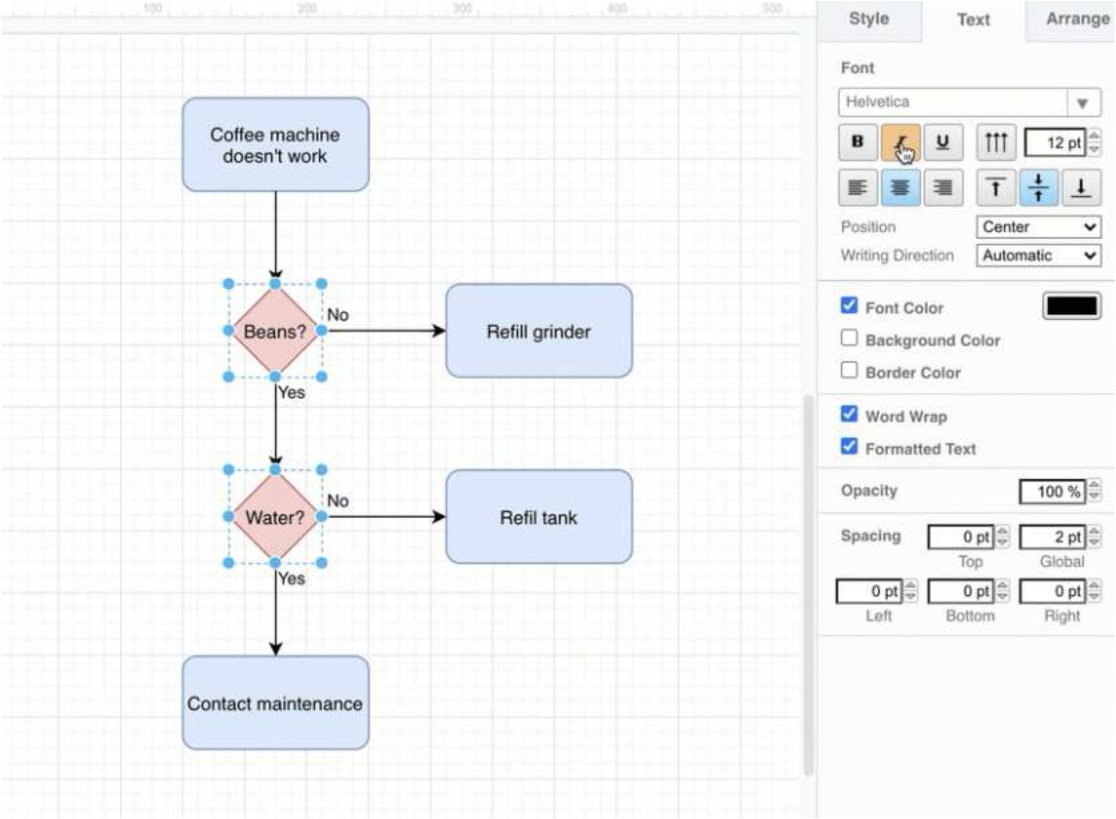
Step 6: Style your flow chart

Once you have finished adding all the shapes, connectors and labels, you can style your flow chart.

1. Select a shape, or hold Shift down and click on multiple shapes and connectors to select many.







Font: Helvetica, 12 pt

Position: Center

Writing Direction: Automatic

☑ Font Color

☐ Background Color

☐ Border Color

☑ Word Wrap

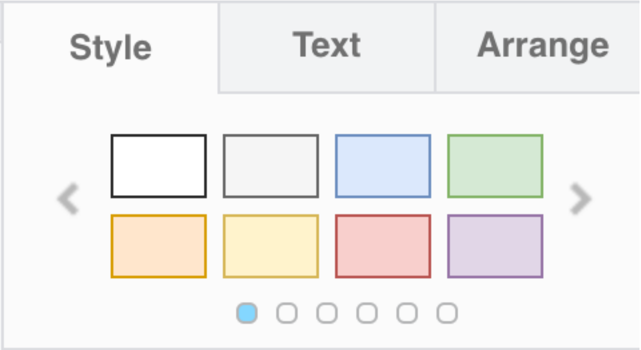
☑ Formatted Text

Opacity: 100 %

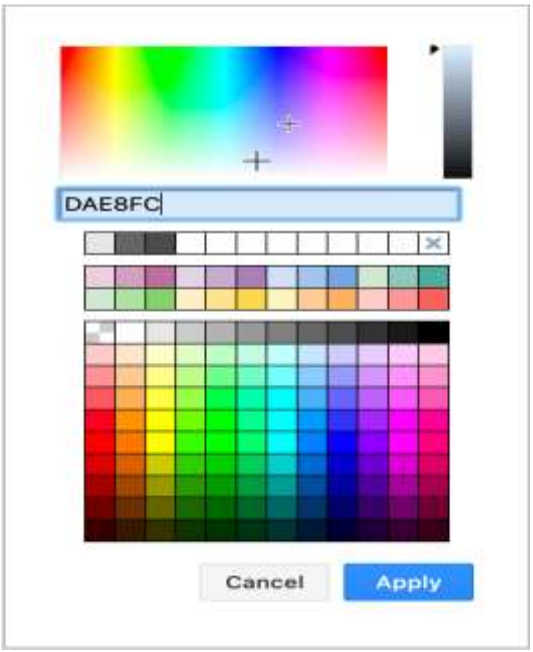
Spacing: 0 pt Top, 2 pt Global, 0 pt Left, 0 pt Bottom, 0 pt Right

2. Add colours and style your shapes and connectors via the *Style* tab.

- The style palette at the top of the *Style* tab changes both the fill and outline colour. Click the left or right arrows to view more styles.

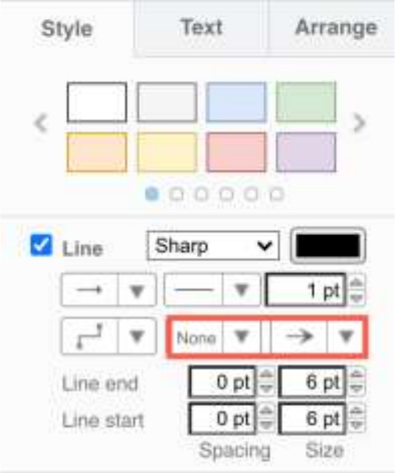


- To set your own colour, click on the colour button next to *Fill* or *Line*, select a a new colour or enter a hex colour code.



DAE8FC

- Style a selected connector to have arrows on both ends or no arrows.



☑ Line Sharp

1 pt

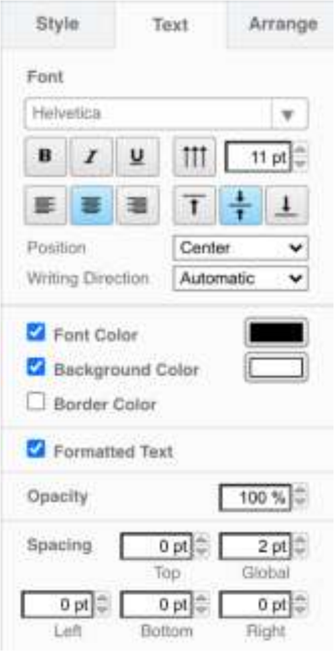
None

Line end 0 pt 6 pt

Line start 0 pt 6 pt

Spacing Size

- Change the text style of labels on the *Text* tab.
  - ✓ Select a different *Font* from the list.
  - ✓ Add *Bold*, *Italics*, or *Underline*.
  - ✓ Change the justification to be left, right or centred.
  - ✓ Click on the colour button next to *Font Color*.



Font

Helvetica

11 pt

Position: Center

Writing Direction: Automatic

☑ Font Color

☑ Background Color

☐ Border Color

☑ Formatted Text

Opacity: 100 %

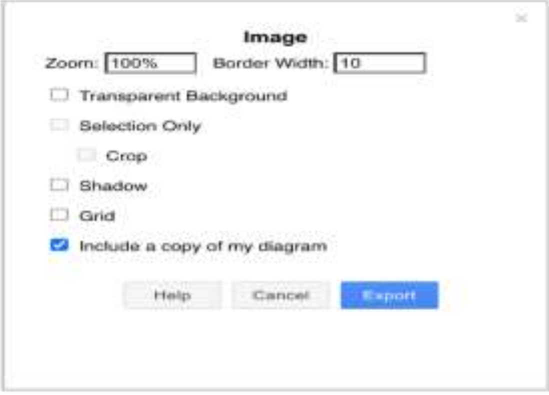
Spacing: 0 pt Top, 2 pt Global

0 pt Left, 0 pt Bottom, 0 pt Right

Step 7: Export and share your flow chart

You can share your diagram in a number of different ways via the *File > Export as* menu. The most common export formats are as images or as a URL.

- Export as a *PNG*, *JPEG* or *SVG* to convert your diagram to an image that you can paste into a website or email.



Image

Zoom: 100% Border Width: 10

☐ Transparent Background

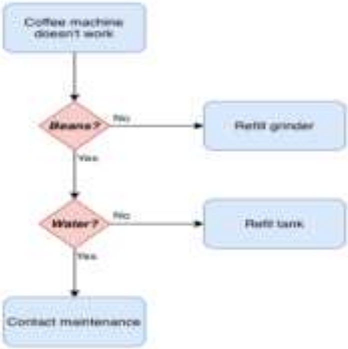
☐ Selection Only

☐ Crop

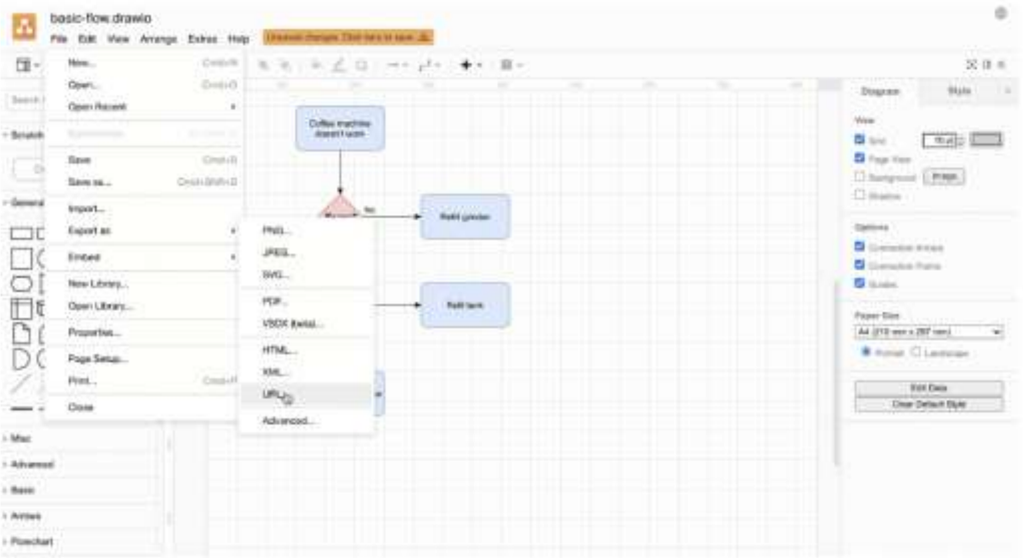
☐ Shadow

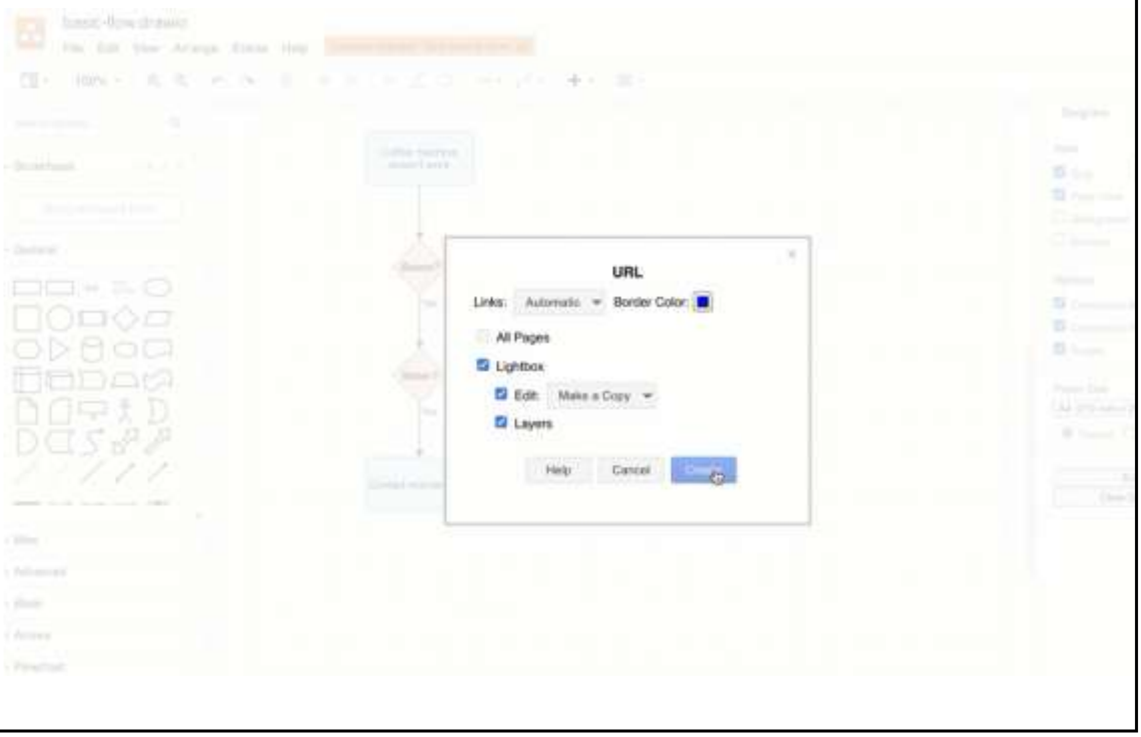
☐ Grid

☑ Include a copy of my diagram



- Export as a *URL* to encode your entire diagram in a URL. When you share this (very long) URL, the person viewing the diagram will see a copy - they don't open or edit your original diagram.




URL

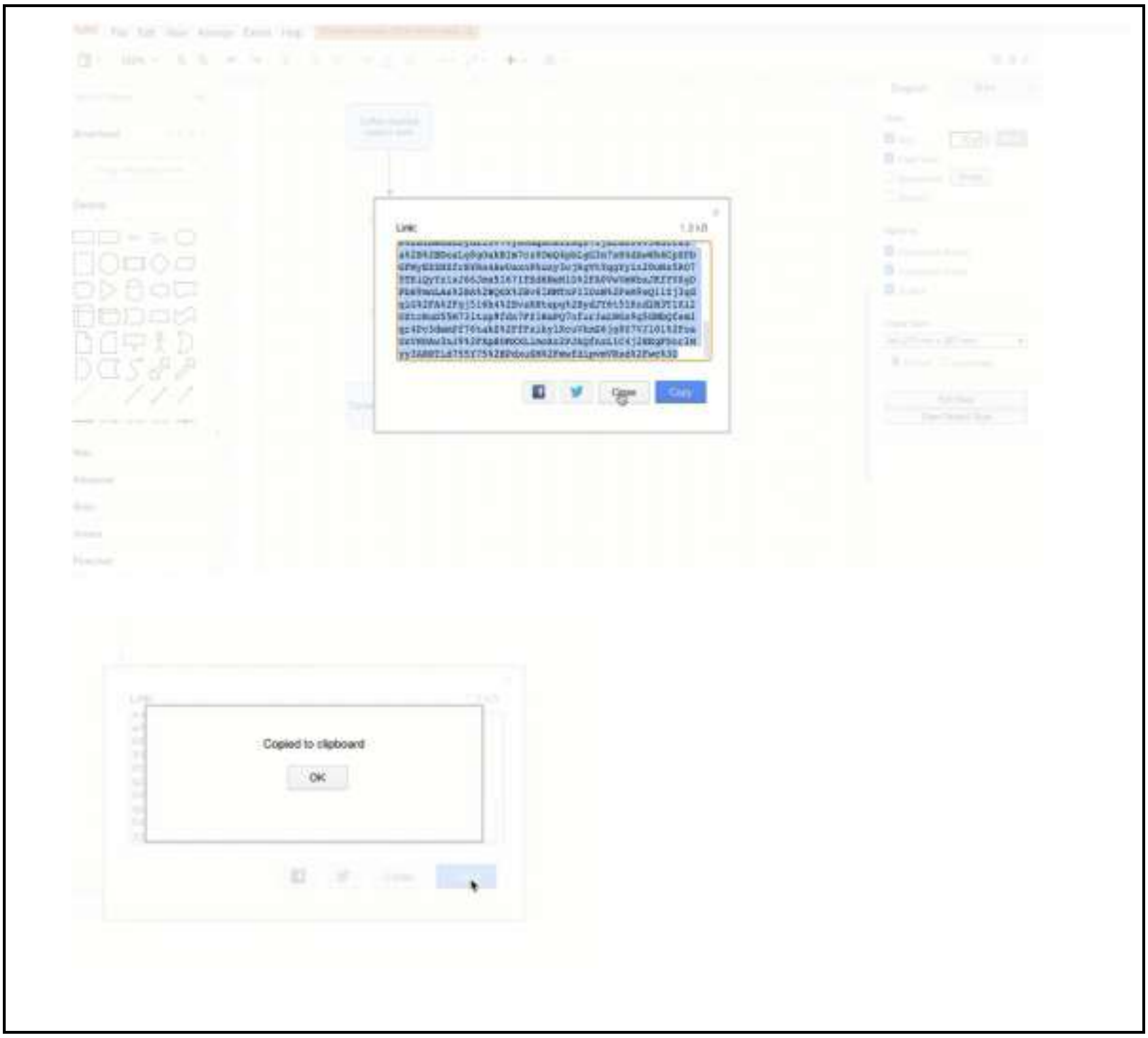
Links: Automatic   Border Color: ■

□ All Pages

☑ Lightbox

☑ Edit: Make a Copy

☑ Layers



Copied to clipboard

Points to Remember

- Steps to draw a basic flow chart in draw.io:
  1. Create a new blank diagram
  2. Add shapes to the drawing canvas
  3. Move, resize, rotate, and delete shapes
  4. Connect shapes
  5. Draw a floating connector
  6. Draw a fixed connector
  7. Change the path of a connector
  8. Add labels
  9. Style your flow chart

10.Export and share your flow chart

Application of learning 1.5.

Drawing a Flowchart for a Simple ATM Withdrawal Program

You have been asked to document the logic of a basic ATM (Automated Teller Machine) withdrawal program. The goal is to create a flowchart that visually represents how a user interacts with the ATM to withdraw money. The ATM has features for PIN authentication, balance checking, and processing a withdrawal.

Learning outcome 1 end assessment

Written assessment

Section 1: Read the Read the following statement related to data structure and algorithm fundamentals, and choose the letter corresponding to the correct answer

1. **Convert the binary number 1101 to its decimal equivalent.**
   a) 11
   b) 12
   c) 13
   d) 14
2. **Convert the decimal number 29 to its binary equivalent.**
   a) 11101
   b) 10101
   c) 11011
   d) 10011
3. **What is the hexadecimal equivalent of the binary number 10110110?**
   a) B6
   b) 76
   c) A6
   d) C6
4. **Convert the octal number 345 to its decimal equivalent.**
   a) 229
   b) 229
   c) 221
   d) 231
5. **What is the result of converting the hexadecimal number 2A to binary?**
   a) 101010
   b) 110010
   c) 100101
   d) 111000
6. **Which logic gate has an output of 1 only when both of its inputs are 1?**
   a) OR
   b) AND
   c) NOT
   d) XOR
7. **The output of an OR gate is 1 if:**
   a) Both inputs are 0
   b) At least one input is 1
   c) Both inputs are 1
   d) Both inputs are different
8. **If you want to build a circuit that outputs 1 when either input A is 1 or both inputs B and C are 1, which combination of gates would you use?**
   a) OR and NOT
   b) AND and OR
   c) AND and NAND
   d) XOR and NOR

9. **Which of the following is a universal gate?**
   a) AND
   b) OR
   c) XOR
   d) NAND
10. **What is the decimal equivalent of the hexadecimal number 7F?**
   a) 127
   b) 126
   c) 124
   d) 121
11. **Which logic gate can be used to invert a binary input?**
   a) AND
   b) OR
   c) NOT
   d) XOR

Section 2: Read the following statement related to data structure and algorithm fundamentals, and answer by True if the statement is correct and False if the statement is wrong

1. In JavaScript, the keyword var is used to declare a variable.
2. JavaScript variables are case-sensitive, meaning myVar and myvar are considered the same variable.
3. A JavaScript variable name cannot start with a number.
4. In JavaScript, a variable declared with const can be reassigned later in the code.
5. JavaScript supports both let and const for block-level variable declarations.
6. The + operator in JavaScript can be used for both addition and concatenation.
7. The === operator checks for equality without type coercion in JavaScript.
8. The assignment operator in JavaScript is represented by ==.
9. In JavaScript, the && operator returns true only if both operands are true.
10. JavaScript has a ternary operator represented by?: that can be used for conditional expressions.

Practical assessment

In an educational setting, teachers and administrative staff often need to calculate students' performance based on their marks in different subjects. This calculation involves determining the total marks obtained and the average marks to assess whether a student has passed or failed according to predefined criteria.

However, performing these calculations manually for a large number of students can be time-consuming and prone to errors. To streamline this process and minimize errors, developing a simple program or algorithm becomes essential. This scenario requires designing an algorithm using pseudocode and a flowchart to automate the calculation of total and average marks for a student based on their scores in five different subjects.

END

References

Books

Karumanchi, N. (2011 (2nd Edition)). *Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles.* Hyderabad, India: CareerMonk Publications.

Robert Sedgewick, K. W. (2011 (4th Edition)). *Algorithms.* Boston, MA, USA: Addison-Wesley.

Skiena, S. S. (2020 (3rd Edition)). *The Algorithm Design Manual.* New York, USA: Springer.

Thomas H. Cormen, C. E. (2009 (3rd Edition)). *Introduction to Algorithms.* Cambridge, MA, USA: The MIT Press.

Web links

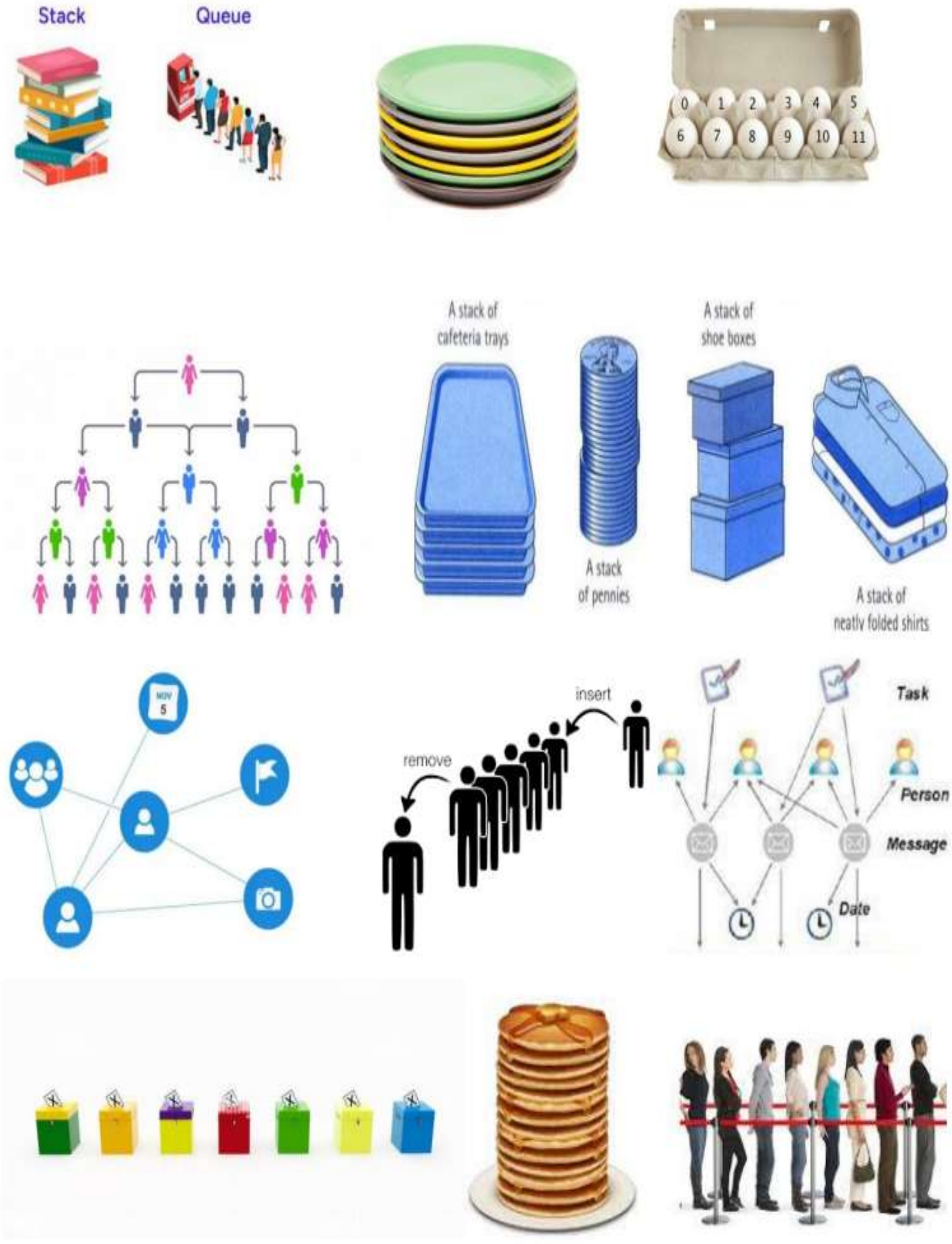
drawio. (2023). *doc/getting-started-basic-flow-chart*. Retrieved from drawio: https://www.drawio.com/doc/getting-started-basic-flow-chart

geeksforgeeks. (2023). *javascript-data-types*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/javascript-data-types/

tutorialspoint. (2024). *programming_methodologies/programming_methodologies_flowchart_elements.htm*. Retrieved from tutorialspoint: https://www.tutorialspoint.com/programming_methodologies/programming_methodologies_flowchart_elements.htm

Sw3schools. (n.d.). */js/js_datatypes.asp*. Retrieved from w3schools: https://www.w3schools.com/js/js_datatypes.asp

Learning Outcome 2: Apply Data Structure



Indicative contents

2.1 Identification of data structure concepts

2.2 Application of linear data structures and their operations

2.3 Application of non-linear data structure and their operations

Key Competencies for Learning Outcome 2: Apply Data Structure

Knowledge	**Skills**	**Attitudes**
● Description of data structure concepts. ● Description of algorithm design ● Classification of sorting algorithms ● Classification of sorting algorithms Searching techniques	● Applying search technics ● Applying sorting techniques ● Applying linear data structure and their operations ● Applying non-linear data structure and their operations	● Being attentive. ● Having sprit of time management

Duration: 45 hrs.

Learning outcome 2 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe clearly Data structure concepts based on intended use.
2. Apply properly Linear Data Structures based on their operational complexity.
3. Apply properly Non-Linear Data Structures based on their operational complexity.

Resources

Equipment	**Tools**	**Materials**
● Computer	● Python tutor ● VisuAlgo	● N/A

Indicative content 2.1: Identification of Data Structure Concepts

Duration: 25 hrs.

Theoretical Activity 2.1.1.1: Description of data structure concepts

Tasks:

1: Answer the following questions:

i.Explain deeply what is data structure?

ii.Describe Characteristics of Data structures?

iii. Discuss about types of Data structures?

iv.Explore classifications of data structures?

v.Discuss list representation

vi.Describe what structure is in programming.

2: Present your findings

3: Ask question

4: Read the Key readings 2.1.1.1 in your manual

Key readings 2.1.1.1: Description of data structure concepts

- **Data Structure Concepts**

1. DATA STRUCTURE

Data Structure is a way to store and organize data so that it can be used efficiently. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

Every application, piece of software, or programs foundation consists of two components: **algorithms** and **data**. Data is information, and algorithms are rules and instructions that turn the data into something useful to programming.

Put another way, remember these two simple equations:

Related data + Permissible operations on the data = Data Structures

Data structures + Algorithms = Programs

Every program relies on data and algorithms. Algorithms are sets of instructions that dictate how data is processed to produce meaningful results. Data, on the other hand, represents the information that algorithms operate on. Together, they form the foundation on which all software systems are built. Data structures play a crucial role in the interaction between algorithms and data.

The data structures have the following importance for programming:

- Data structures study how data are stored in a computer so that operations can be implemented efficiently
- Data structures are especially important when there is a large amount of information to deal with.
- Data structures are conceptual and concrete ways to organize data for efficient storage and manipulation

2. CHARACTERISTICS OF DATA STRUCTURES

Data Structure is the systematic way used to organize the data. The characteristics of Data Structures are:

✓ **Linear or Non-Linear**

This characteristic arranges the data in sequential order, such as arrays, graphs etc.

✓ **Static and Dynamic**

Static data structures have fixed formats and sizes along with memory locations. The static characteristic shows the compilation of the data.

✓ **Time Complexity**

The time factor should be very punctual. The running time or the execution time of a program should be limited. The running time should be as less as possible. The less the running time, the more accurate the device is.

✓ **Correctness**

Each data must definitely have an interface. Interface depicts the set of data structures. Data Structure should be implemented accurately in the interface.

✓ **Space Complexity**

The Space in the device should be managed carefully. The memory usage should be used properly. The space should be less occupied, which indicates the proper function of the device.

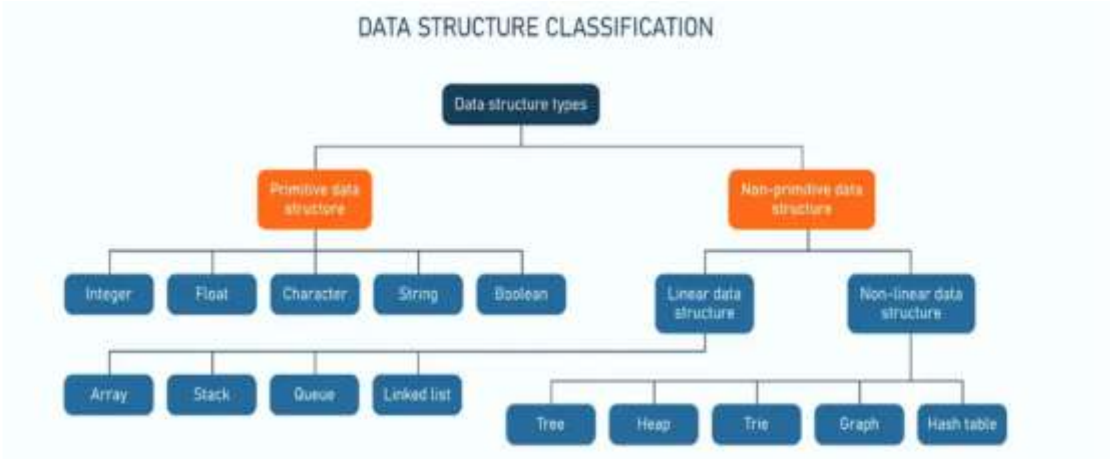
4. TYPES OF DATA STRUCTURES

There are two types of data structures:

- Primitive data structure

- Non-primitive data structure

Check out the graph below to better understand how data structure types are categorized.



3.1 Primitive Data structure

Primitive data structures or Data types are the most basic data units available in programming languages. They include:

- ✓ **integers:** whole numbers—positive, negative, or zero, for example, 1, -30, 100, etc.;
- ✓ **floats:** decimal numbers with a fractional part, like 3.14, -0.5, 7.0;
- ✓ **characters:** individual characters, for example, “A,” “d,” and “1”;
- ✓ **strings:** sequences of characters; and
- ✓ **Booleans:** binary values that can be either true or false, commonly used for conditional expressions.

3.2 Non-Primitive Data structure

Non-primitive data structures are created with primitive data structures as their building blocks to efficiently organize and manage a collection of data. They can handle different data types and complex operations like searching, sorting, insertion, deletion, and more.

Non-primitive data structures fall into two large categories:

- ✓ Linear data structure
- ✓ Non-linear data structure

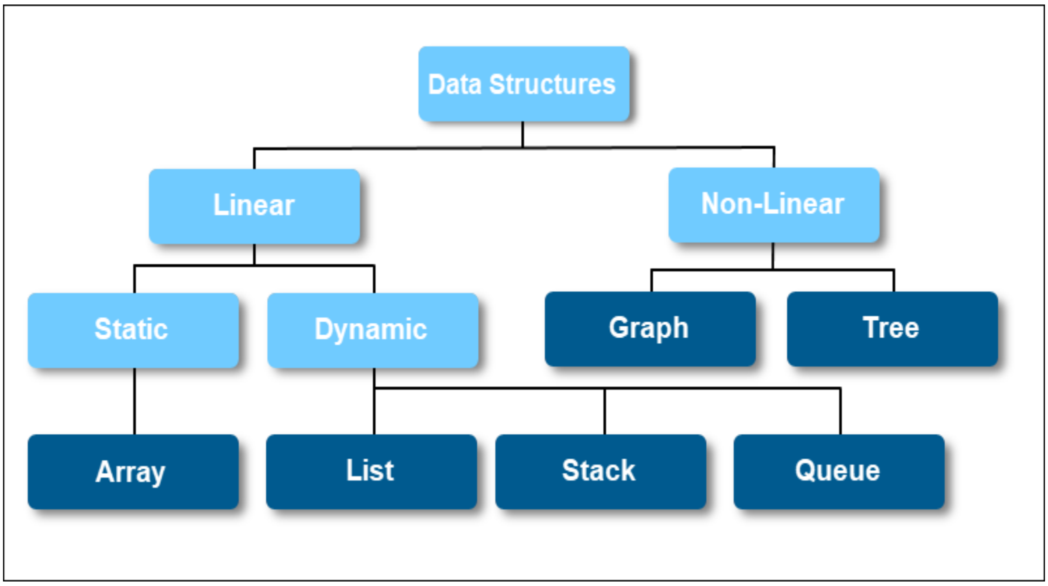
Notes: This section will deal with non-primitive data structure. When we delve into the study of data structures, much of the focus tends to be on non-primitive data types because they offer more sophisticated ways to manage and organize data. So the term Data structure have much fucus on non-primitive types.

4. CLASSIFICATIONS OF DATA STRUCTURES

As mentioned above data structures are divided into two categories:

1. Linear data structure
2. Non-linear data structure

Check out the graph below to better understand how data structure types are categorized.



4.1 Linear Data Structures

In linear data structure, data elements are linked to one another in a sequential arrangement. In this manner, a single run can traverse the structure. Linear data structures consist of four types. They are:

- Stack
- Array
- Queue
- Linked list

Linear Data structures can also be classified as:

✓ **Static data structure**

Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.

An example of this data structure is an array.

✓ **Dynamic data structure**

In the dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.

4.2 Non-Linear Data Structures

In linear data structure data elements are randomly arranged. The elements are non-arranged sequentially.

In Non-linear data structures, there are different paths for an element to reach the other element.

The data elements in the non-linear data structures are connected to one or more elements.

There are two types of non-linear data structures. They are:

- Tree Data Structure
- Graph Data Structure
- Table Data Structure

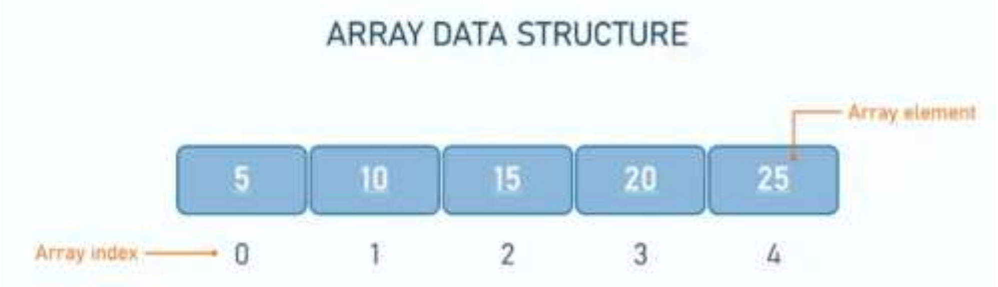
Let's learn about each type in detail.

Popular linear data structures are:

1. Array Data Structure

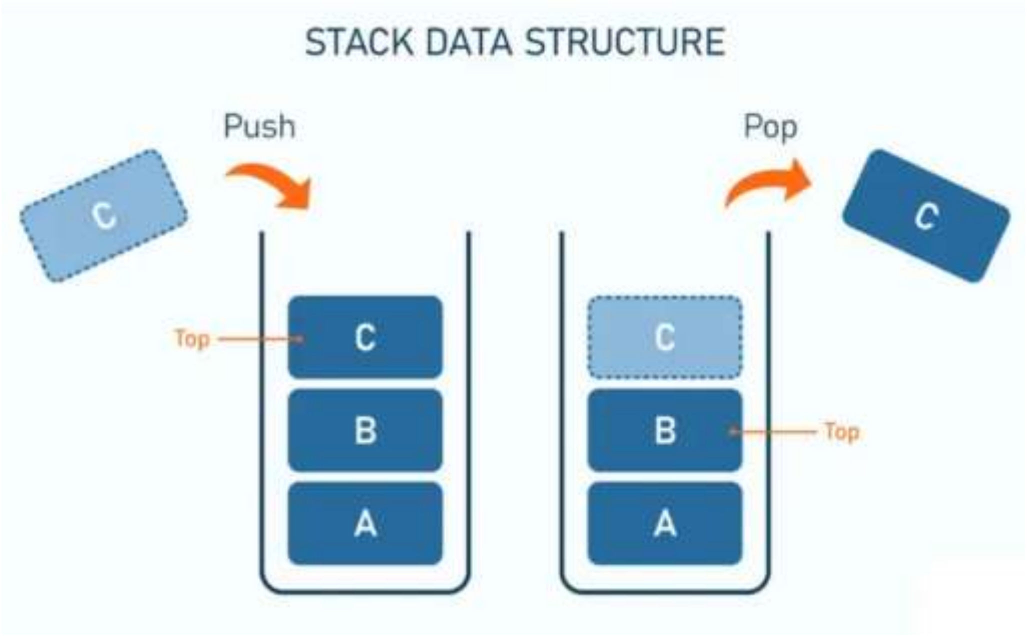
In an array, elements in memory are arranged in continuous memory. All the elements of an array are of the same type. And, the type of elements that can be stored in the form of arrays is determined by the programming language.

In an array, each element is associated with an index, a numerical identifier that indicates the element's position. Indexing begins at 0 for the first element and increments sequentially up to the array size minus one. For example, in an array of size 5, the indices range from 0 to 4, like in the picture below.



2. Stack Data Structure

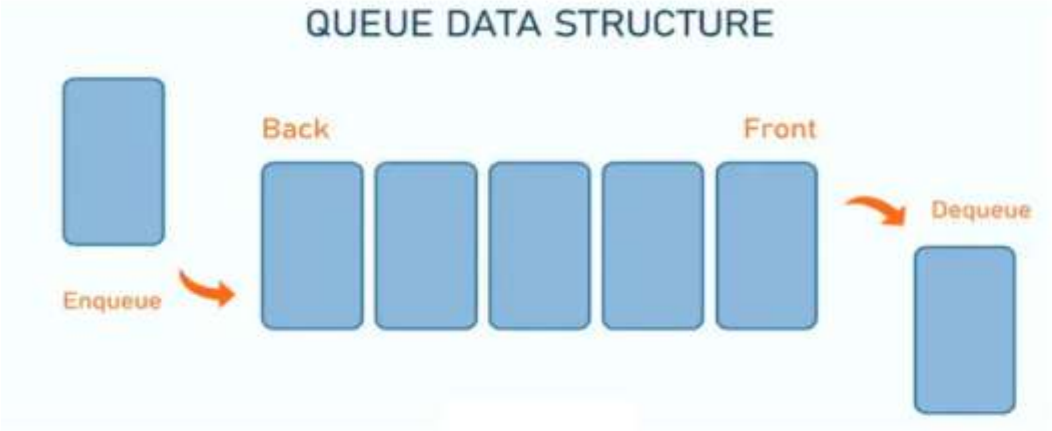
In stack data structure, elements are stored in the LIFO principle. That is, the last element stored in a stack will be removed first.



3. Queue Data Structure

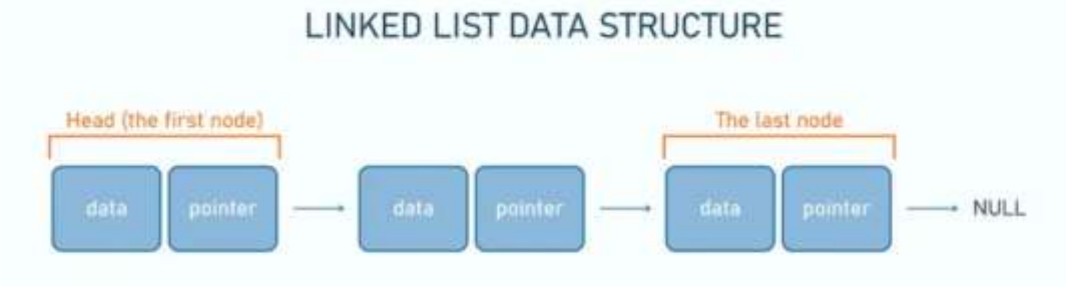
Unlike stack, the queue data structure works in the FIFO principle where first element stored in the queue will be removed first.

It works just like a queue of people in the ticket counter where first person on the queue will get the ticket first.



4. Linked List Data Structure

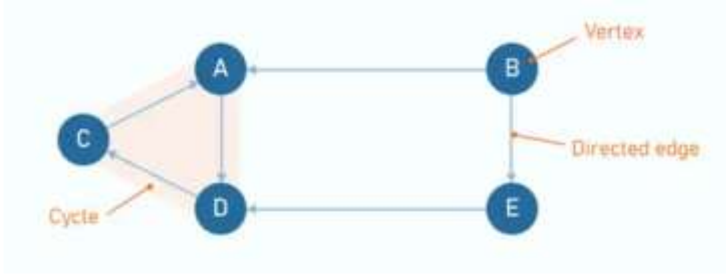
In linked list data structure, data elements are connected through a series of nodes. And, each node contains the data items and address to the next node.

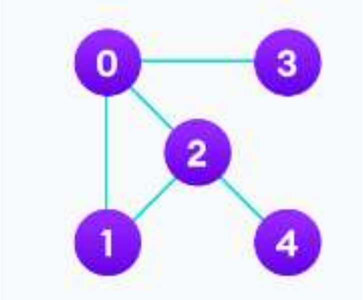


Popular Non-linear data structures are:

1. Graph Data Structure

In graph data structure, each node is called vertex and each vertex is connected to other vertices through edges.

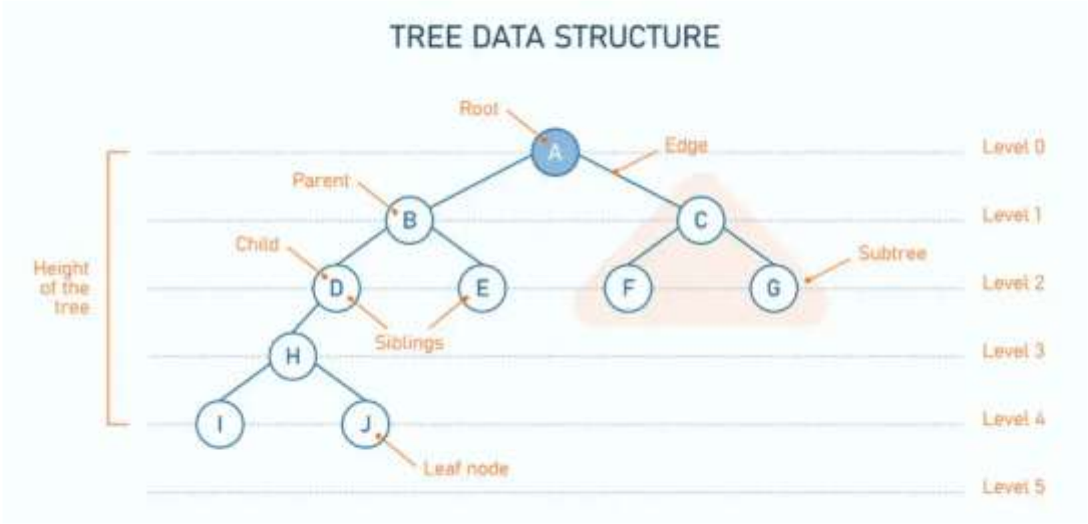


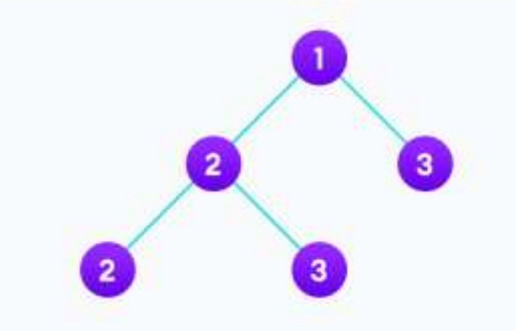


2. Trees Data Structure

A tree is a collection of elements (nodes) connected with edges that reflect parent-child relationships. Each node (except the root—the topmost node) has a single parent. Nodes without children are known as leaves. You've definitely seen this structure when looking at a family tree or organizational chart.

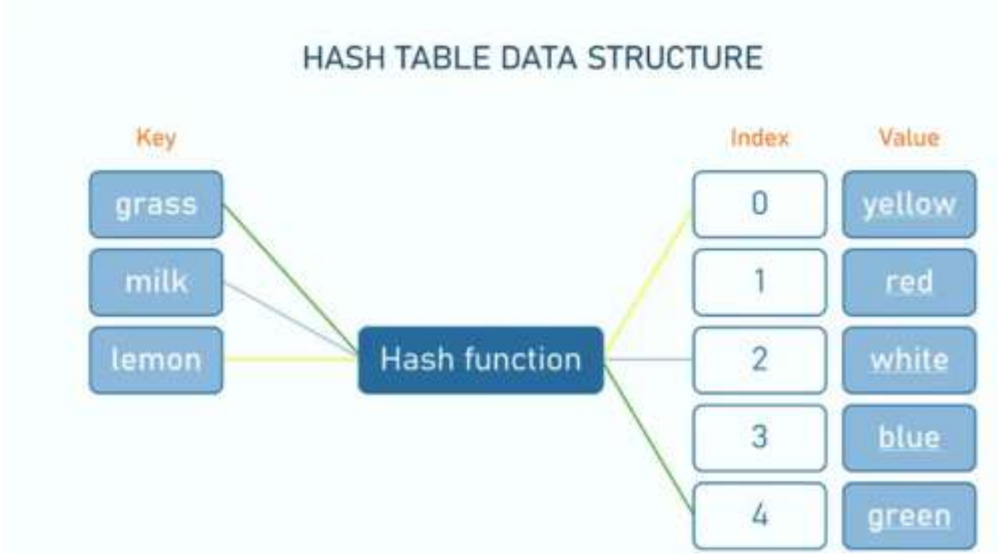
Similar to a graph, a tree is also a collection of vertices and edges. However, in tree data structure, there can only be one edge between two vertices.





3. Hash table data structure

A hash table or hash map stores key-value pairs in an array. The key is a unique identifier to access or retrieve the value, which is the associated data or information.



The hash function takes a key as input and produces a unique numerical value called a hash code that serves as an index for a particular data element. This mapping process is deterministic, meaning the same key will always be hashed to the same index, facilitating rapid lookup and retrieval of values.

We use hash tables to quickly map keys to specific locations in the array. The goal of the hash function is to distribute keys appropriately across the array indices, minimizing collisions (situations where multiple keys map to the same index).

Hash tables are common in scenarios where efficient storage and retrieval of key-value pairs are required — for example, caching systems. Additionally, we employ hash tables in data processing tasks such as indexing and deduplication.

5. MAJOR DATA STRUCTURE OPERATIONS

The major or the common operations that can be performed on the data structures are:

- **Searching:** We can search for any element in a data structure.
- **Sorting:** We can sort the elements of a data structure either in an ascending or descending order.
- **Insertion:** We can also insert the new element in a data structure.
- **Update:** We can also update the element, i.e., we can replace the element with another element.
- **Deletion:** We can also perform the delete operation to remove the element from the data structure.

6. LIST REPRESENTATION

2.1 **List representation:** List presentation in data structures refers to how lists are organized and represented in a programming environment. There are several ways to implement lists, each with its advantages and use cases:

✓ **Arrays:** An array is a collection of elements identified by index or key. In a one-dimensional array, elements are stored sequentially in memory.

✓ **Linked Lists:** A linked list is a linear collection of elements called nodes. Each node contains data and a reference (or pointer) to the next node in the sequence.

- **Types of Linked list.**

✓ **Singly Linked List**: Each node points to the next node.

✓ **Doubly Linked List:** Each node points to both the next and the previous node.

✓ **Circular Linked List:** The last node points back to the first node, creating a circular structure

2.2 List operations

List operations refer to the various actions that can be performed on lists, whether they are implemented as arrays, linked lists, or other data structures. Here we have common list **operations and their typical implementations:**

- **Insertion operation:**

Refers to add a new element to the list.

✓ **Types Insertion operation in list**

- **At the End:** Adding an element to the end of the list (often referred to as appending).
- **At a Specific Position:** Adding an element at a specific index or location in the list.

✓ **Implementation in list**

- **Arrays:** Inserting at the end is usually straightforward, but inserting at a specific position requires shifting elements**.**
- **Linked Lists**: Inserting is generally efficient as it involves updating pointers. For singly linked lists, the insertion might require traversal to find the correct position.

- **Deletion operation:**

Refers to removing an element from the list.

✓ **Types in list**

- **By Value:** Removing the first occurrence of a specific value**.**
- **By Index:** Removing the element at a specific index or position.

✓ **Implementation in list**

- **Arrays:** Removing an element requires shifting elements to fill the gap.
- **Linked Lists:** Deleting involves updating pointers. In a doubly linked list, deletion is efficient as both previous and next pointers are available.

- **Traversal Operation**

Refers to visit each element of the list to perform some operation.

✓ **Implementation in list**

- **Arrays:** Simple iteration through indices.
- **Linked Lists:** Requires starting from the head and following pointers to visit each node.
- **Sorting:** Arranging the elements of the list in a specific order (e.g., ascending or descending).
- **Update:** refers to modify the value of an element at a specific index or position.

7. STRUCTURES

Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a member of the structure.

Unlike an array, a structure can contain many different data types (int, float, char, etc.).

7.1Characteristics of Structures in Programming

•**Composite Data Type**: Structures allow you to create complex data types by combining simpler data types, such as integers, floats, and strings.

•**Member Variables**: Each structure can contain multiple member variables. These can be of different types and are accessed using a dot notation.

•**Encapsulation**: Structures encapsulate related data, making it easier to manage and pass around as a single unit. This is especially useful for grouping related data fields together.

•**Data Alignment**: In some languages, structures may have specific alignment requirements or padding to ensure efficient access. This can affect the size of the structure.

•**User-defined**: Structures are user-defined types, meaning that programmers can create them according to their needs.

Theoretical Activity 2.1.1.2: Description of data structure algorithm

Tasks:

1: Answer the following questions:

i. Explain searching techniques

ii. Describe sorting techniques

iii. Describe two types of algorithm complexity

2: Present your findings

3: Ask clarifying question if any

4: Read the Key readings 2.1.1.2

Key readings 2.1.1.2: Description of data structure algorithm

An algorithm in the context of data structures (DS) is a step-by-step procedure or formula for solving a problem or performing a task. Algorithms manipulate data structures, which are organized formats for storing and managing data

1. **Searching techniques**

Searching in data structure refers to the process of finding the required information from a collection of items stored as elements in the computer memory. These sets of items are in different forms, such as an array, linked list, graph, or tree.

Another way to define searching in the data structures is by locating the desired element of specific characteristics in a collection of items. Different Searching Methods Searching in the data structure can be done by applying searching algorithms to check for or extract an element from any form of stored data structure.

These algorithms are classified according to the type of search operation they perform, such as:

- **Sequential search**

The list or array of elements is traversed sequentially while checking every component of the set. For example – **Linear Search.**

- **Interval Search**

The interval search includes algorithms that are explicitly designed for searching in sorted data structures. In terms of efficiency, these algorithms are far better than linear search algorithms. **Example- Logarithmic Search, Binary search.**

2. Classification of Sorting Algorithms

Sorting is an algorithm that arranges the elements of a given list in a particular order [ascending or descending].

- Sorting algorithms are categorized on the following basis

✓ **By number of comparisons:** Comparison-based sorting algorithms check the elements of the list by key comparison operation and need at least O(n log n) comparisons for most inputs. In this method, algorithms are classified based on the number of comparisons. For comparison-based sorting algorithms, best case behavior is O(n log n) and worst case behavior is O(n2). For example – Quick Sort, Bubble Sort, Insertion Sort etc.

✓ **By Number of Swaps:** In this method, sorting algorithms are categorized by the number of swaps (interchanging of position of two numbers, also called inversion).

✓ **By Memory Usage:** Some sorting algorithms are “in place” and they need O(1) or O(log n) memory to create auxiliary locations for sorting the data temporarily**.**

- ✓ **By Recursion**: Sorting algorithms are either recursive (for example – quick sort) or non-recursive (for example – selection sort, and insertion sort), and there are some algorithms which use both (for example – merge sort).

- ✓ **By Stability:** Sorting algorithm is stable if two elements with equal values appear in the same order in output as it was in the input. The stability of a sorting algorithm can be checked with how it treats equal elements. Stable algorithms preserve the relative order of equal elements, while unstable sorting algorithms don't. In other words, stable sorting maintains the position of two equals elements similar to one another. For example – Insertion Sort, Bubble Sort , and Radix Sort.

- ✓ **By Adaptability**: In a few sorting algorithms, the complexity changes based on pre-sorted input i.e. pre-sorted array of the input affects the running time. The algorithms that take this adaptability into account are known to be adaptive algorithms. For example – Quick sort is an adaptive sorting algorithm because the time complexity of Quick sort depends on the initial input sequence. If input is already sorted then time complexity becomes O(n^2) and if input sequence is not sorted then time complexity becomes O(n logn). Some adaptive sorting algorithms are: Bubble Sort, Insertion Sort and Quick Sort. On the other hand, some non-adaptive sorting algorithms are: Selection Sort, Merge Sort, and Heap Sort.

- ✓ **Internal Sorting**: Sorting algorithms that use main memory exclusively during the sort are called internal sorting algorithms. This kind of algorithm assumes high-speed random access to all memory. Some of the common algorithms that use this sorting feature are: Bubble Sort, Insertion Sort., and Quick Sort.

- ✓ **External Sorting**: Sorting algorithms that use external memory, during the sorting come under this category. They are comparatively slower than internal sorting algorithms. For example, merge sort algorithm. It sorts chunks that each fit in RAM, then merges the sorted chunks together.

Practical Activity 2.1.2: Selecting appropriate data structure algorithm

Task:

1: perform the task described below:

In a bank, a Teller has role of saving the clients. He/she has organize the banknotes of same values together and coins of same values together that he/she has serve to clients. When a client arrives in the bank, he/she chooses a line to join in the front of a Teller and the line is formed according to the arrival of clients.

Mr.one wants to pay 28450rwf as school fees to the bank and he must organize banknotes and coins before deposit money to facilitate the teller to not mix the banknotes and coins.

1. Observe the above figure and describe what you see
2. Where a new arrived client will line up
3. Using above figure who is going to be served first by teller 1or teller 2
4. Discuss how one is going to organize banknotes and coins before depositing;
5. Discuss how the teller is going to keep money deposited by one

2: Select an appropriate algorithm to an existing problem

3: Perform a given task above

4: Ask assistance where is needed

5: Present your findings

6: Ask question and take notes

7: Read key readings 2.1.2 in trainees manual

Key readings 2.1.2:Selecting appropriate data structure

Selecting appropriate data structure algorithm

1. Sorting

Sorting is the process of arranging elements in some logical order either in ascending or

Descending order.

2. Methods for sorting

There are various methods for sorting in this section that we are going to discuss:

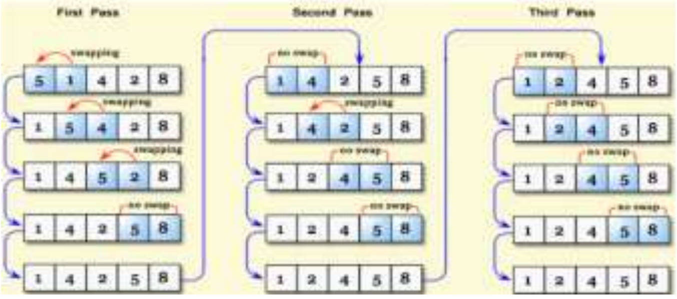
Bubble sort,

Insertion sort and Selection sort.

1. **Bubble sort**

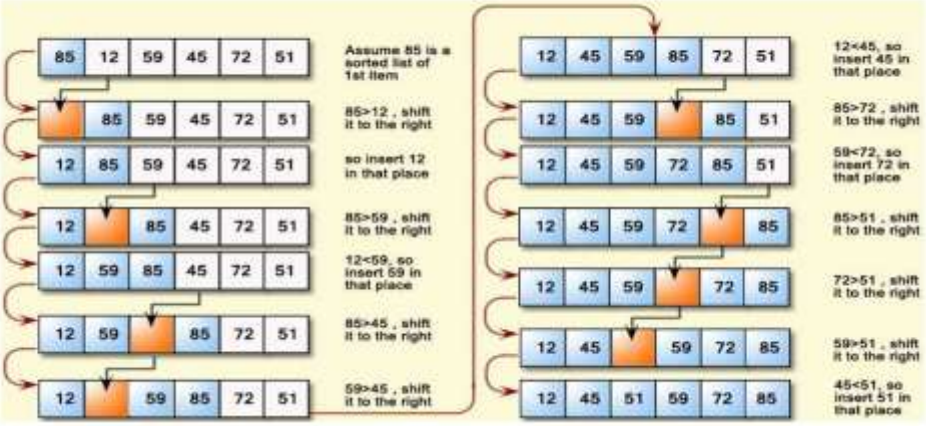
Bubble Sort is a simple-minded algorithm based on the idea that we look at the list, and wherever we find two consecutive elements out of order, we swap them. This is done as follows: Repeatedly traverse the unsorted part of the array by comparing consecutive elements, and interchange them when they are out of order

See below demonstration:



2. **Insertion sort**

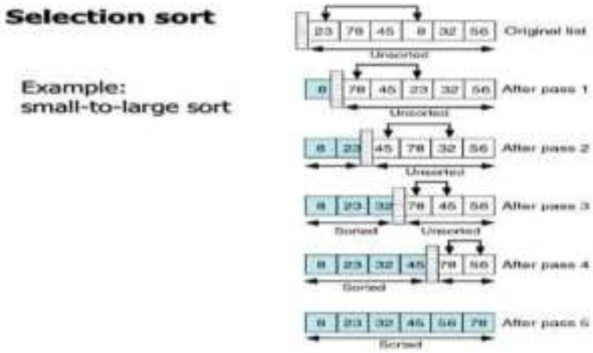
The Insertion Sort is a comparison-based algorithm that builds a final sorted array one element at a time. It iterates through an input array and removes one element per iteration, finds the place if the element belongs in the array, and then places it there. The illustration is given here below:


Assume 85 is a sorted list of 1st item
85>12 , shift it to the right
so insert 12 in that place
85>59 , shift it to the right
12<59, so insert 59 in that place
85>45 , shift it to the right
59>45 , shift it to the right
12<45, so insert 45 in that place
85>72 , shift it to the right
59<72, so insert 72 in that place
85>51 , shift it to the right
72>51 , shift it to the right
59>51 , shift it to the right
45<51, so insert 51 in that place

The Insertion Sort traverses the array and inserts each element into the sorted part of the list where it belongs. It involves pushing down the larger elements in the sorted part.

3. **Selection sort**

The Selection Sort is a simplicity sorting algorithm. Here are basic steps of selection sort algorithm:



The idea of Selection Sort is that we repeatedly find the smallest element in the unsorted part of the array and swap it with the first element in the unsorted part of the array.

4. **Merge Sort**

Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller sub arrays and sorting those sub arrays then merging them back together to obtain the sorted array.

Points to Remember

- **Merge sort is** a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller sub arrays and sorting those sub arrays then merging them back together to obtain the sorted array**.**

- **Quicksort** is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array**.**
- **Heap Sort** is a comparison-based sorting algorithm that uses a binary heap data structure.
- **Radix Sort** is a non-comparison-based sorting algorithm that sorts numbers by processing individual digits.
- **Counting Sort** is a non-comparison-based sorting algorithm suitable for sorting integers within a specific range.
- **Bucket Sort** is a non-comparison-based sorting algorithm that distributes elements into several buckets and then sorts each bucket individually (using a different sorting algorithm, like Insertion Sort).
- **Shell sort** is a generalized version of the insertion sort algorithm. It first sorts elements that are far apart from each other and successively reduces the interval between the elements to be sorted.

Application of learning 2.1.

In a market inventory system, you need to arrange a list of products by their prices in ascending order. Describe how you would apply the bubble sort technique to accomplish this. What would be the steps involved?

Indicative content 2.2: Application of Linear Data Structures and their Operations

Duration: 8 hrs

Theoretical Activity 2.2.1: Descriptions of operations of linear

Tasks:

1: Answer the following questions:

i. Define a linear data structure.

ii. Identify Data Structure Operations

iii. Explain application of linear data structure and examples.

2: Present your findings

3: Ask question

4: Read the Key readings 2.2.1 in your manual

Key readings 2.2.1.: Descriptions of operations of linear

1. **A linear data structure** is a type of data structure where elements are arranged sequentially or linearly, one after another. In linear data structures, every element has a unique predecessor and a unique successor, except for the first and last elements.

2. **The following are some of the most frequently used operations:**

- **Traversing:** Accessing each record exactly once so that certain items in the record may be processed.
- **Searching:** Finding the location of the record with a given key value or finding the locations of all records which satisfy one or more conditions
- **Inserting:** Adding a new record to the structure.
- **Deleting:** Removing a record from the structure.
- **Sorting:** Arranging the records in some logical order (e.g., in numerical order according to some NUMBER key, such as social security number or account number).
- **Merging:** Combining the records of two different sorted lists into a single sorted list.

3. Examples of a linear data structure we have: **Arrays, Linked Lists, Stacks, and Queues**

I. ARRAYS

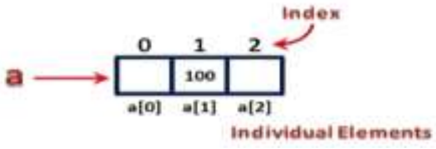
•**An array** is a collection of elements identified by index or key, where each element is of the same data type and stored at contiguous memory locations

• **Arrays dimensions**:

A dimension of array is a direction in which you can vary the specification of an array's elements. An array can be of one or many dimensions.

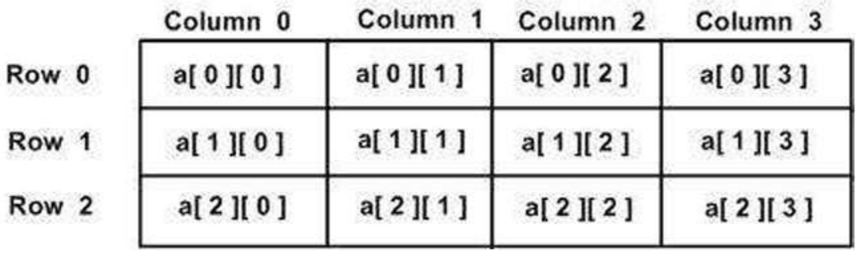
a) One-dimensional array.

Example: a [3] It is an array called a with 3 elements. The elements of the array are logically represented by the name of the array with in brackets its index or positions. For the one dimensional array a with 3 elements, elements are written like a[0], a[1] and a[2]. The indexes start always by 0. Graphically, it is



b) Two-dimensional array.

A Two-dimensional Array is a collection of a fixed number of elements (components) of the same type arranged in two dimensions. The two-dimensional array is also called a matrix or a table. The intersection of a column and a row is called a cell. The numbering of rows and columns starts by 0 Example: The array a[3][4] is an array of 3 rows and 4 columns. In matrix representation, it looks like the following:



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

a[i][j] means that it is an element located at the intersection of row i and column j.

Each element is identified by its row and column.

II. LINKED LISTS

Simply a **list** is a sequence of data, and **linked list** is a sequence of data linked with each other.

Types of linked lists

Linked list is classified into three types as shown below:

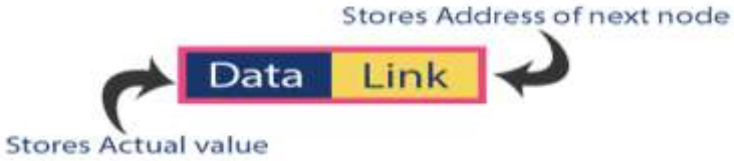
a) Single linked lists

b) Double linked lists

c) Circular linked list

1) Single linked lists

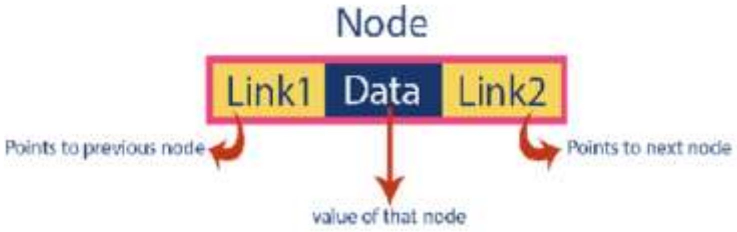
Single linked list is a sequence of elements in which every element has link to its next element in the sequence. In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data and next. The data field is used to store actual value of that node and next field is used to store the address of the next node in the sequence.



2) Double linked lists

Double linked list is a sequence of elements in which every element has links to its previous

Element and next element in the sequence. So, we can traverse forward by using next field and can traverse backward by using previous field. Every node in a double linked list contains three fields and they are shown in the following figure.



Double linked lists representation

- **Algorithm in Double Linked List**
  - ✓ **Inserting at Specific location in the list (After a Node)**

We can use the following steps to insert a new node after a node in the double linked list

Step 1: Create a new Node with given value.

Step 2: Check whether list is Empty (head == NULL)

Step 3: If it is Empty then, assign NULL to newNode => previous&newNode => next and New Node to head.

Step 4: If it is not Empty then, define two node pointers temp1&temp2 and initialize temp1 With head.

Step 5: Keep moving the temp1 to its next node until it reaches to the node after which we want to insert the newNode (until temp1 => data is equal to location, here location is the Node value after which we want to insert the newNode).

Step 6: Every time check whether temp1 is reached to the last node. If it is reached to the last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and Terminate the function. Otherwise move the temp1 to next node.

Step 7: Assign temp1 => next to temp2, newNode to temp1=> next, temp1 to newNode => Previous, temp2 to newNode => next and newNode to temp2 => previous.

- ✓ **Deleting a Specific Node from the list**

We can use the following steps to delete a specific node from the double linked list

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3: If it is not Empty, then define a Node pointer 'temp' and initialize with head.

Step 4: Keep moving the temp until it reaches to the exact node to be deleted or to the last Node.

Step 5: If it is reached to the last node, then display 'Given node not found in the list!

Deletion not possible!!!' and terminate the fuction.

Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7: If list has only one node and that is the node which is to be deleted then set head to NULL and delete temp (free (temp)).

Step 8: If list contains multiple nodes, then check whether temp is the first node in the list (Temp == head).

Step 9: If temp is the first node, then move the head to the next node (head = head => next), set head of previous to NULL (head => previous = NULL) and delete temp.

Step 10: If temp is not the first node, then check whether it is the last node in the list (temp => Next == NULL).

Step 11: If temp is the last node then set temp of previous of next to NULL (temp =>
Previous => next = NULL) and delete temp (free (temp)).

Step 12: If temp is not the first node and not the last node, then set temp of previous of next to temp of next (temp => previous => next = temp => next), temp of next of previous to temp of previous (temp => next => previous = temp => previous) and delete temp (Free (temp)).

- ✓ **Displaying a Double Linked List**

We can use the following steps to display the elements of a double linked list

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty, then display 'List is Empty!!!' and terminate the function.

Step 3: If it is not Empty, then define a Node pointer 'temp' and initialize with head.

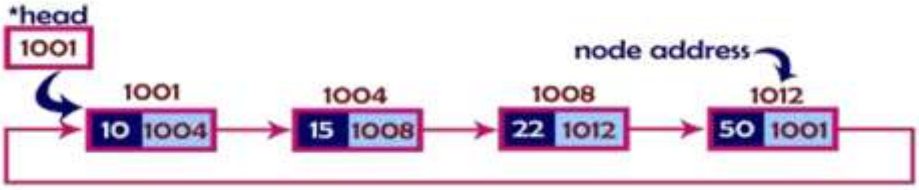
Step 4: Display 'NULL <= '.

Step 5: Keep displaying temp → data with an arrow (⍰) until temp reaches to the last node

Step 6: Finally, display temp → data with arrow pointing to NULL (temp → data --> NULL).

3) Circular Linked List

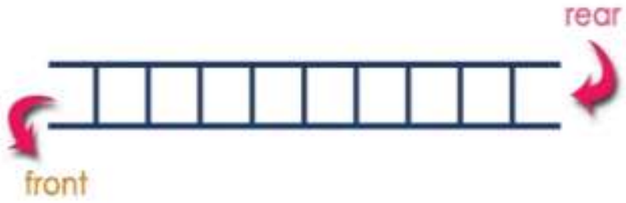
In circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.



Circular Linked List representation

III. QUEUE

A queue is a linear list in which data can only be inserted at one end, called the REAR, and deleted from the other end, called the FRONT. These restrictions ensure that the data is processed through the queue in the order in which it is received. In other words, a queue is a structure in which whatever goes fist comes out first (first in, first out (FIFO) structure).



Queue representation

Types of queues

There exist two types of queues:

a. **Linear queue**
b. **Circular queue**

a. **Linear Queue**

Queue data structure is a linear data structure in which the operations are performed based on FIFO principle. Queue Operations using Array before we implement actual operations, first follow the below steps to create an **empty queue,**

Step 1: Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.

Step 2: Declare all the user **defined functions** which are used in queue implementation.

Step 3: Create a one dimensional array with above defined SIZE **(int queue[SIZE])**

Step 4: Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = 1, rear = -1)

Step 5: Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

enQueue(value) - Inserting value into the queue

Step 1: Check whether queue is FULL. (rear == SIZE-1)

Step 2: If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!"" and

Terminate the function.

Step 3: If it is NOT FULL, then increment rear value by one (rear++) and set queue [rear] = value.

deQueue() - Deleting a value from the Queue

Step 1: Check whether queue is EMPTY. (front == rear)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then increment the front value by one (front ++). Then display queue[front] as deleted element. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue

Step 1: Check whether queue is EMPTY. (front == rear)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.

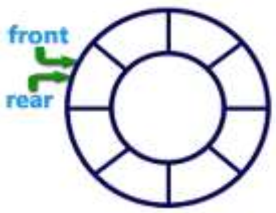
Step 3: If it is NOT EMPTY, then define an integer variable 'I' and set 'i = front+1'.

Step 3: Display 'queue[i]' value and increment 'I' value by one (i++). Repeat the same

until 'I' value is equal to rear (i<= rear)

b. Circular Queue

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.



Circular Queue Operation

To implement a circular queue data structure using array, we first create it.

Step 1: Include all the header files which are used in the program and define a constant **'SIZE'** with specific value.

Step 2: Declare all user defined functions used in circular queue implementation.

Step 3: Create a one dimensional array with above defined SIZE (int cQueue[SIZE])

Step 4: Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int

front = -1, rear = -1)

Step 5: Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

enQueue(value) - Inserting value into the Circular Queue

Step 1: Check whether queue is FULL. **((rear == SIZE-1 && front == 0) || (front == rear+1))**
Step 2: If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.
Step 3: If it is **NOT FULL**, then check **rear == SIZE - 1** && **front != 0** if it is TRUE, then set rear = -1.
Step 4: Increment rear value by one (rear++), set queue[rear] = value and check 'front
== -1' if it is TRUE, then set front = 0.
deQueue() - Deleting a value from the Circular Queue

Step 1: Check whether queue is **EMPTY. (front == -1 && rear == -1)**
Step 2: If it is **EMPTY**, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.
Step 3: If it is **NOT EMPTY,** then display **queue[front]** as deleted element and increment the front value by one (front ++). Then check whether front == SIZE, if it is TRUE, then set 'front = 0'. Then check whether both 'front – 1' and rear are equal (front -1 == rear), if it TRUE, then set both **front** and **rear** to **'-1'** **(front = rear = -1).**

display () - Displays the elements of a Circular Queue

Step 1: Check whether queue is **EMPTY. (front == -1)**
Step 2: If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.
Step 3: If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front'.
Step 4: Check whether 'front <= rear', if it is TRUE, then display 'queue[i]' value and
Increment 'I' value by one (i++). Repeat the same until 'i <= rear' becomes FALSE.
Step 5: If 'front <= rear' is FALSE, then display 'queue[i]' value and increment 'I' value by one (i++). Repeat the same until 'i <= SIZE – 1' becomes FALSE.
Step 6: Set **i** to **0.**
Step 7: Again display **'cQueue[i]'** value and increment **i** value by one **(i++).** Repeat the same until **'i <= rear'** becomes **FALSE.**

Queue operations

There are two main operations related to queues.

- **Enqueue:** the enqueue operation insert an item at the rear of the queue
- **Dequeue:** the dequeue operation deletes the item at the front of the queue**.**

Practical Activity 2.2.1: Selecting appropriate linear data structures

Task:

1: perform the task described below:

A market analyst is reviewing the last few transactions to identify the most recent ones. Which linear data structure would you use to review these transactions in reverse order, and why?

2: Select an appropriate algorithm to an existing problem

3: Perform a given task above

4: Ask assistance where is needed

5: Present your findings

6: Ask question and take notes

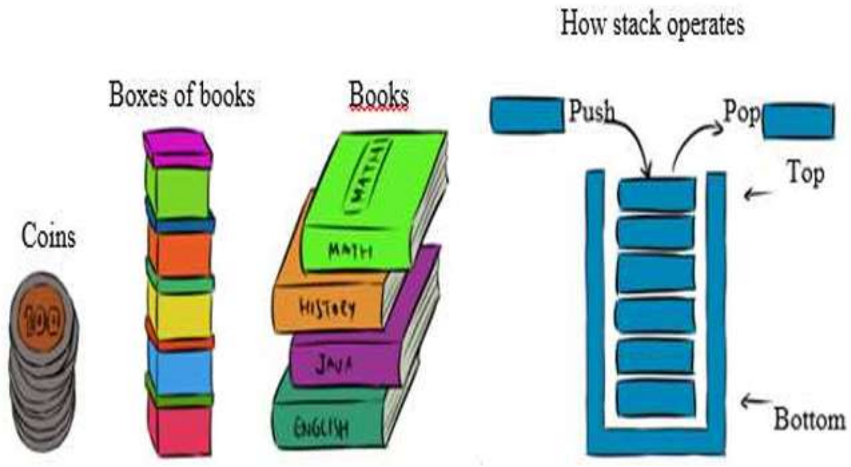
7: Read key readings 2.2.2

Key readings 2.2.1: Selecting appropriate linear data structure

1. Stack

1.1 Basic concepts

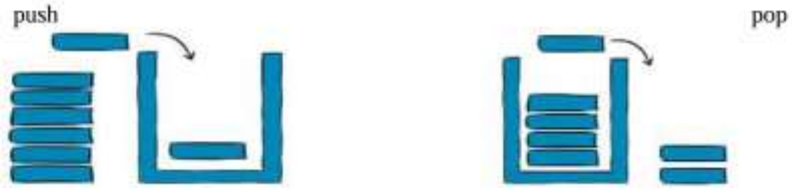
A stack is a restricted linear list in which all additions and deletions are made at one end, the top. If we insert a series of data items into a stack and then remove them, the order of the data is reversed. Data input as 5, 10, 15, 20, for example would be removed as 20, 15, 10, and 5. This reversing attribute is why Stacks are known as Last in, First out (LIFO) data structures. We use many different types of stacks in our daily lives. We often talk of a stack of coins, stack of books on a table and stack of plates in a kitchen. Any situation in which we can only add or remove an object at the top is a stack. If we want to remove an object other than the one at the top, we must first remove all objects above it. The following charts illustrates cases of stack.



Stack presentation

A Stack is a Last in First out (LIFO) dynamic table or data structure. It has the following characteristics:

- List of the same kind of elements;
- Addition and deletion of elements occur only at one end, called the top of the stack;
- Computers use stacks to implement method calls;
- Stacks are also used to convert recursive algorithms into non recursive algorithm.



Stacks operations (Push and Pop)

1.2 Operations performed on stacks are the following

- **Push:** adds an element to the stack
- **Pop:** removes an element from the stack
- **Peek:** display at top element of the stack.

1.3 Stack exceptions

Adding an element to a full stack and removing an element from an empty stack would generate errors or exceptions:

- **Stack overflow exception:** it occurs if you try to add an item to stack or queue which is already full.

- **Stack underflow exception:** it occurs if you try to remove an item from an empty queue or stack.

Points to Remember

- **Linear Data Structures** are simpler and involve direct, sequential relationships (like Arrays, Linked Lists, Stacks, and Queues).
- **Arrays:** An array is a collection of elements identified by index or key, where each element is of the same data type and stored at contiguous memory locations.
- **Linked List:** A linked list is a linear data structure where each element (node) contains a data part and a reference (or link) to the next node in the sequence.
- **Stack:** A stack is a linear data structure that follows the Last In, First Out (LIFO) principle.
- **Queue:** A queue is a linear data structure that follows the First In, First Out (FIFO) principle.
- **A stack underflow** occurs when you try to pop an item from an empty stack. Essentially, it happens when there are no elements to remove.
- **A stack overflow** occurs when you try to push an item onto a stack that is already full. In other words, it happens when there is no more space available to add new elements.

Application of learning 2.2.

A grocery store implements a system where the last product scanned at checkout can be removed if the customer changes their mind. Which linear data structure would be best for this functionality, and how would it operate?

Indicative content 2.3: Application of Non-Linear Data Structures and their Operations

Duration: 7 hrs.

Theoretical Activity 2.3.1: Description of application for non- linear data structure

Tasks:

1: Answer the following questions:

i. Define a non-linear data structure.

ii. Discus to examples of non-linear data structure

2: Present your findings

3: Ask question

4: Read the Key readings 2.3.1

Key readings 2.3.1: Description of application for non-linear data structure

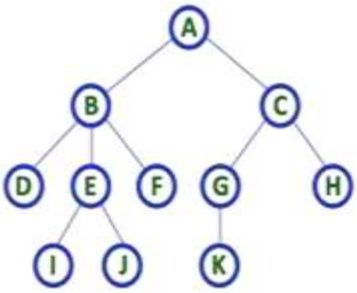
1. Non-linear data structure: A non-linear data structure is a type of data structure where data elements are not arranged in a sequential or linear order. Instead, they are organized in a hierarchical or interconnected manner, which allows for more complex relationships between the elements. Unlike linear data structures (e.g., arrays, linked lists, stacks, and queues) where each element has a single successor and predecessor, non-linear data structures can have multiple relationships, enabling efficient storage and retrieval of data in scenarios where a hierarchical or graph-like structure is needed**.**

- **Key Characteristics of Non-Linear Data Structures**
  - ✓ **Hierarchical Relationships:** Data elements are organized in a hierarchy, such as in trees, where elements are connected in parent-child relationships.
  - ✓ **Multiple Connections:** Elements can be connected to multiple other elements, as seen in graphs, where nodes can be linked to many other nodes.
  - ✓ **Complex Data Access:** Non-linear data structures enable more complex data access patterns, such as searching for data based on a hierarchy or multiple connections.

1. Tree

In linear data structure, data is organized in sequential order and in non-linear data structure; data is organized in random order. Tree is a very popular data structure used in wide range of applications. A tree data structure can be defined as follows. **Tree data structure is a collection of data (Node) which is organized in hierarchical structure, and this is a recursive definition.**

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



TREE with 11 nodes and 10 edges

- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

Tree Terminology

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

a) Root

In a tree data structure, the first node is called as Root Node. Every tree must have root node.

b) Edge

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

c) Parent

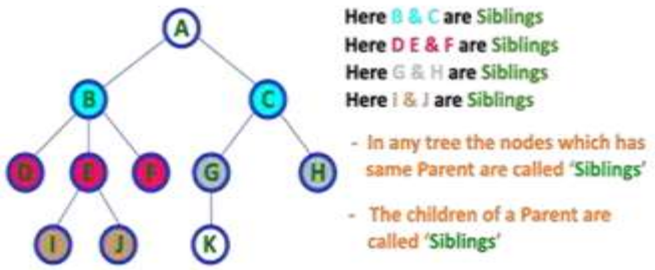
In a tree data structure, the node which is predecessor of any node is called a PARENT NODE. In simple words, the node which has branch from it to any other node is called as parent node.

d) Child

In a tree data structure, the node which is descendant of any node is called a CHILD Node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.

e) Siblings

In a tree data structure, nodes which belong to same Parent are called SIBLINGS.

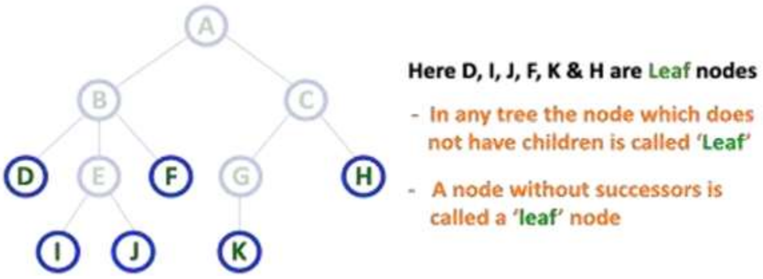


Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

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

f) Leaf

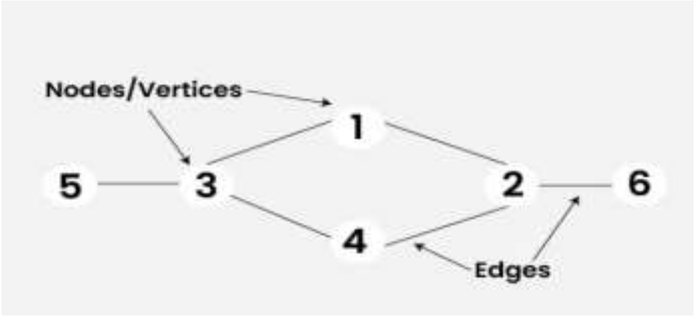
In a tree data structure, the node which does not have a child is called a **LEAF Node**. In a tree data structure, the leaf nodes are also called a External Nodes or Terminal node.



Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

2. **Graph Data Structure** is a collection of nodes connected by edges. It's used to represent relationships between different entities. Graph algorithms are methods used to manipulate and analyse graphs, solving various problems like finding the shortest path or detecting cycles.



- **Applications of Graph**

Following are the real-life applications:

- ✓ If we recall all the previous data structures that we have studied like array, linked list, tree, etc. All these had some restrictions on structure (mostly linear and tree hierarchical which means no loops). Graph allows random connections between nodes which is useful in many real-world problems where do have restrictions of previous data structures.

- ✓ Used heavily in social networks. Everyone on the network is a vertex (or node) of the graph and if connected, then there is an edge. Now imagine all the features that you see, mutual friends, people that follow you, etc can see as graph problems.

- ✓ **Neural Networks:** Vertices represent neurons and edges represent the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^11 neurons and close to 10^15 synapses.
- ✓ **Compilers:** Graph Data Structure is used extensively in compilers. They can be used for type inference, for so-called data flow analysis, register allocation, and many other purposes. They are also used in specialized compilers, such as query optimization in database languages**.**

- ✓ Robot planning: Vertices represent states the robot can be in and the edges the possible transitions between the states. Such graph plans are used, for example, in planning paths for autonomous vehicles.

- ✓ In GPS. The problems like finding the closest route, closest petrol pumps, etc are all soled using graph problems.

- ✓ For optimizing the cost of connecting all locations of a network. For example, minimizing wire length in a wired network to make sure all devices are connected is a standard Graph problem called Minimum Spanning Tree.

- ✓ Can be used to represent the interactions between players on a team, such as passes, shots, and tackles. Analyzing these interactions can provide insights into team dynamics and areas for improvement.

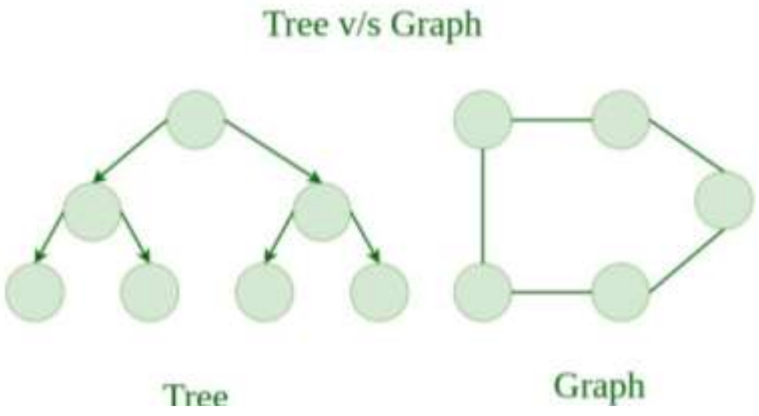
- ✓ Can be used to represent the topology of computer networks, such as the connections between routers and switches.

- ✓ Graphs are used to represent the connections between different places in a transportation network, such as roads and airports.

- ✓ Dependencies in a software project (or any other type of project) can be seen as graphs and generating a sequence to solve all tasks before dependents is a standard graph topological sorting algorithm.

- **Difference Between Graph and Tree**

Graphs and **trees** are two fundamental data structures used in computer science to represent relationships between objects. While they share some similarities, they also have distinct differences that make them suitable for different applications.



•**A graph data structure** is a collection of nodes (also called vertices) and edges that connect them. Nodes can represent entities, such as people, places, or things, while edges represent relationships between those entities. Graphs are used to model a wide variety of real-world systems, such as social networks, transportation networks, and computer networks.

•A **tree data structure** is a hierarchical data structure that consists of nodes connected by edges. Each node can have multiple child nodes, but only one parent node. The topmost node in the tree is called the **root node**. Trees are often used to represent hierarchical data, such as file systems, **XML documents**, and **organizational charts.**

Practical Activity 2.3.1: Selecting an appropriate non-linear data structure to solve problem

Task:

1: Perform the task described below:

You want to ensure that all market locations in a city are connected so that goods can be transferred between any two locations, either directly or indirectly. Describe how you would check if your graph is fully connected.

2: Select an appropriate non-linear data structure to solve problem.

3: Perform a given task above

4: Ask assistance where is needed

5: Present your findings

6: Ask question and take notes

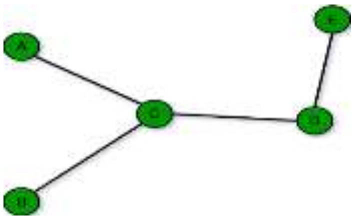
7: Read key readings 2.3.2

Key readings 2.3.2: Selecting an appropriate non-linear data structure to solve problem

Types of Graphs

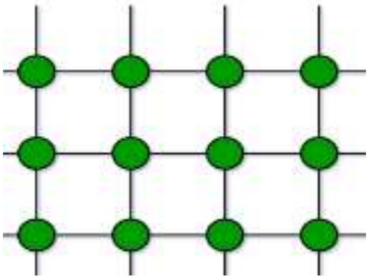
1. Finite Graphs

A graph is said to be finite if it has a finite number of vertices and a finite number of edges. A finite graph is a graph with a finite number of vertices and edges. In other words, both the number of vertices and the number of edges in a finite graph are limited and can be counted. Finite graphs are often used to model real-world situations, where there is a limited number of objects and relationships between them



2. Infinite Graph

A graph is said to be infinite if it has an infinite number of vertices as well as an infinite number of edges.

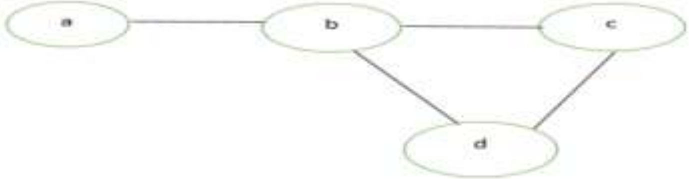


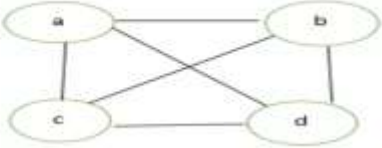
3. **Trivial Graph**

A graph is said to be trivial if a finite graph contains only one vertex and no edge. A trivial graph is a graph with only one vertex and no edges. It is also known as a singleton graph or a single vertex graph.

4. **Simple Graph**

A simple graph is a graph that does not contain more than one edge between the pair of vertices.





5. **Multi Graph**

Any graph that contains some parallel edges but doesn’t contain any self-loop is called a multigraph. For example a Road Map.

Parallel Edges: If two vertices are connected with more than one edge then such edges are called parallel edges that are many routes but one destination.
Loop: An edge of a graph that starts from a vertex and ends at the same vertex is called a loop or a self-loop.

6. **Null Graph**

A graph of order n and size zero is a graph where there are only isolated vertices with no edges connecting any pair of vertices. A null graph is a graph with no edges. In other words, it is a graph with only vertices and no connections between them. A null graph can also be referred to as an edgeless graph, an isolated graph, or a discrete graph

Points to Remember

- **Depth-First Search (DFS)** is a fundamental algorithm used in graph theory to traverse or search through a graph or tree structure.
- **A graph data structure** is a collection of nodes that have data and are connected to other nodes.
- **Undirected Graphs**: A graph in which edges have no direction, i.e., the edges do not have arrows indicating the direction of traversal.
- **Directed Graphs**: A graph in which edges have a direction, i.e., the edges have arrows indicating the direction of traversal.
- **Weighted Graphs:** A graph in which edges have weights or costs associated with them.
- **Unweight Graph**s: A graph in which edges have no weights or costs associated with them.

Application of learning 2.3

You are designing a market distribution network where each main warehouse supplies multiple regional warehouses, and each regional warehouse, in turn, supplies several local stores. However, there are also direct routes between some local stores for emergency supplies. How would you model this system using a combination of graph and tree structures?

Learning outcome 2 end assessment

Written assessment

I. Read the Read the following statement related to data structure and algorithm fundamentals, and choose the letter corresponding to the correct answer

1. Which of the following is an example of a non-linear data structure?

a) Linked List
b) Queue
c) Stack
d) Tree

2. Which sorting algorithm does NOT use recursion?

a) Quick Sort
b) Merge Sort
c) Insertion Sort
d) Heap Sort

3. Which of the following operations is common to both linear and binary search?

a) Sorting the data before searching
b) Comparing elements
c) Splitting the data into two halves
d) Reversing the data

4. What is the best sorting algorithm to use when stability is a requirement?

a) Quick Sort
b) Merge Sort
c) Heap Sort
d) Shell Sort

5. Which classification of sorting algorithms involves comparing elements to one another?

a) By Recursion
b) By Memory Usage
c) By Number of Comparisons
d) By Stability

II. Read the following statement related to data structure and algorithm fundamentals, and answer by True if the statement is correct and False if the statement is wrong

1. A binary search can be used on unsorted data
2. Merge Sort is both a stable and recursive sorting algorithm
3. Arrays and linked lists are examples of non-linear data structures.
4. The number of swaps is a classification criterion for sorting algorithms.
5. Time complexity measures the amount of memory an algorithm uses during execution

Practical assessment

Hasha Ltd is facing challenges in managing its warehouse data due to the use of hard copies for record-keeping. As a solution, you have been tasked with creating an efficient data management system that handles imports, exports, and transit processes for furniture. Use the following data structures: **Stack, Sorting, and Linked List** to develop the system.

For each question below, **write an algorithm** that addresses the respective problem:

1. The company receives shipments daily, and the manager needs to process them in the order they arrive. Sometimes the manager needs to undo the last action (removal of incorrect shipments). Write an algorithm using a **Stack** to manage this process. The stack should allow the manager to undo the last operation efficiently. Define the methods needed to push, pop, and check the top of the stack.
2. The warehouse manager needs to sort furniture items by categories such as Tables, Chairs, Beds, and Dressers to produce a daily report. Write an algorithm using a Sorting Algorithm to organize these categories. Explain which sorting algorithm (e.g., Quick Sort, Merge Sort, Bubble Sort, etc.) you would choose and why it is appropriate for this situation.

3. The number of shipments in the warehouse can vary daily, so a flexible data structure is needed. Write an algorithm using a Linked List to represent the furniture items in the warehouse, allowing dynamic addition and removal of items. Explain how the linked list supports these operations and how you would traverse the list to generate reports

END

References

BOOKS

Algorithms, L. J. (2016). *Loiane Groner.* Birmingham, UK: Packt Publishing.

Bae, S. (2017). *Data Structures and Algorithms with JavaScrip.* UK: Packt Publishing.

Rozentals, N. (2016). *Mastering JavaScript Data Structures and Algorithms.* Birmingham, UK: Packt Publishing.

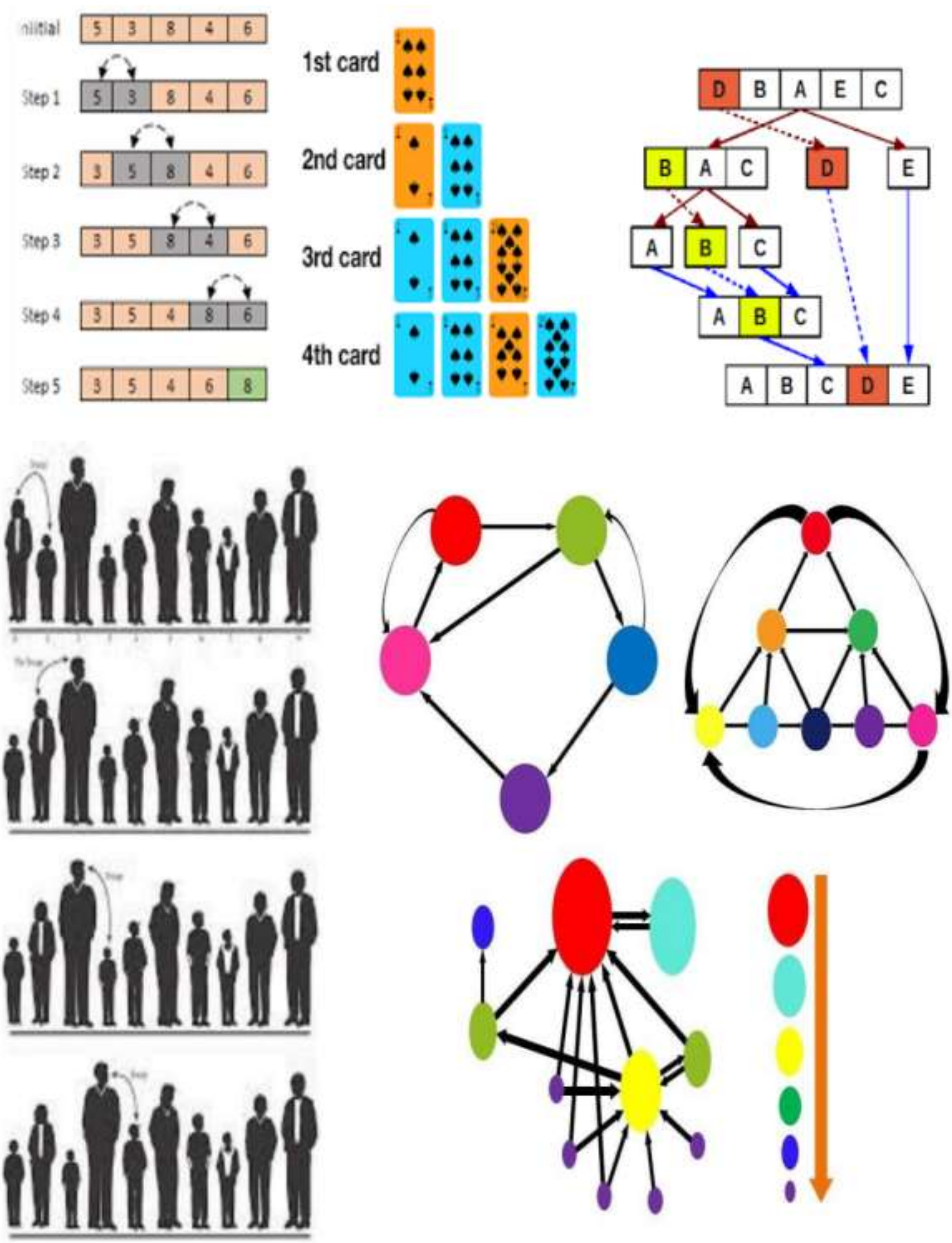
WEB LINKS

programiz. (2024). */dsa/data-structure-types*. Retrieved from programiz: https://www.programiz.com/dsa/data-structure-types

simplilearn. (2024). */tutorials/data-structure-tutorial/what-is-data-structure#:~:text=Data%20structures%20are%20a%20specific,the%20data%20for%20easy%20use.* Retrieved from simplilearn: https://www.simplilearn.com/tutorials/data-structure-tutorial/what-is-data-structure#:~:text=Data%20structures%20are%20a%20specific,the%20data%20for%20easy%20use.

techtarget. (2024). */searchdatamanagement/definition/data-structure*. Retrieved from techtarget: https://www.techtarget.com/searchdatamanagement/definition/data-structure

Learning Outcome 3: Implement Algorithm Using JavaScript



Indicative contents
1.1 Development of JavaScript source code **1.2 Run JavaScript source codes** **1.3 Test Time and space complexity**

Key Competencies for Learning Outcome 3: Implement Algorithm using JavaScript

Knowledge	Skills	Attitudes
● Description of JavaScript running environment ● Description of browser embedded Tools and IDE Terminal ● Description of Key concepts of measuring time and space complexity ● Description of Time and space measurement tools	● Preparing JavaScript running environment ● Writing JavaScript source code ● Appy sorting operations ● Apply search operations ● Using browser-embedded Tools ● Using IDE Terminal ● Documenting test findings	● Being critical thinker ● Being innovative ● Being attentive ● Being creative ● Being practical oriented ● Being details Oriented ● Having team work spirit

Duration: 55 hrs

Learning outcome 2 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe correctly JavaScript running environment in line of its type
2. Prepare properly JavaScript running environment in accordance with computer operating system
3. Write properly JavaScript Source code based on Algorithm
4. Run successfully JavaScript source code in accordance with expected result
5. Describe properly Key concepts of measuring time and space complexity based on properties of developed program
6. Test successfully Time and space complexity based on data structure standards

Resources

Equipment	**Tools**	**Materials**
● Computer	● Benchmarkify.js ● jsPerf ● WebStorm ● Visual Studio Code IDE built-in profiling tool ● Frame graphs ● Blackfire ● YSlow ● Chrome/Microsoft/Firefox DevTools	● Internet

Indicative content 3.1: Development of JavaScript Source Code

Duration: 20 hrs

Practical Activity 3.1.1: Preparing JavaScript running environment

Tasks:

1: Read the given task

To get started with JavaScript programming, especially for implementing data structures like Linked Lists, Arrays, Queues, Stacks, Trees, Graphs, and Tables, you need to set up a development environment. Go in computer LAB and prepare environment to run JavaScript programs.

2: Follow the demonstration of trainer

3: Individually perform the task 3.1.1 referring to the demonstration

4: Ask for assistance where it is needed.

5: Read the key readings 3.1.1 in trainee’s Manual

Key readings 3.1.1.: Preparing JavaScript running environment

- **JavaScript running environment**

The **JavaScript running environment** refers to the platform or environment where JavaScript code is executed.

There are two main types of environments where JavaScript can run: **Browser Environment** and **Node.js Environment (Server-side).**

- **Preparation of JavaScript Running Environment**

Before writing any code, it’s important to set up the environment where you’ll run your JavaScript programs. Here's how to prepare the environment:

- ✓ **Choose an Editor or IDE**: You can use text editors like **Visual Studio Code**, **Sublime Text**, or an Integrated Development Environment (IDE) such as **WebStorm**.
- ✓ **Install Node.js (optional)**: If you want to run JavaScript outside the browser, you can install **Node.js**. This allows you to execute JavaScript programs directly on your system.
    - ✚ Download from **Node.js official website.**
    - ✚ After installation, you can run JavaScript files using the terminal with node filename.js.

- ✓ **Browser Console**: You can also run JavaScript directly in your web browser (e.g., Chrome, Firefox). Open the **Developer Tools** (F12) and use the **Console** to write and execute JavaScript code.

Practical Activity 3.1.2: Implementing Linked List Data Structure

Task:

1: Read the task described below:

As a software developer implement a function to add a new node at the beginning, middle, and end of a linked list.

2: Follow the demonstration of trainer

3: Individually perform the task 3.1.2 referring to the demonstration

4: Ask for assistance where it is needed.

5: Read the key readings 3.1.2 in trainee's Manual

Key readings 3.1.2: Implementing Linked List Data Structure

1. Linked lists

- **Properties of a Linked List**

Linked lists are dynamic, which means they can grow or shrink during the runtime of a program

Linked lists can be accessed only sequentially

The nodes are not stored in contiguous memory locations.

Node contains the data and pointer

A particular node is pointed and can be accessed by the pointer stored in the previous node

The first node of the linked list is pointed by a specific pointer called the head

The last node of the linked list points to null, which specifies the end of the list.

- **Implementation**

A linked list is a data structure that consists of a sequence of elements, each of which contains a reference (or "link") to the next element in the sequence. The first element is called the head and the last element is called the tail.

Linked lists have many advantages over other data structures. Now we shall look at how to implement Linked list using JavaScript.

- ✓ Defining the Node Class and the LinkedList class

This is basically the prerequisite in order to implement a linked list in JavaScript. In this step, 2 classes namely one for the nodes and the other for the linked list need to be created.

A constructor is a special function that creates and initializes an object instance of a class. In JavaScript, a constructor gets called when an object is created using the new keyword. The purpose of a constructor is to create a new object and set values for any existing object properties.

1. The Node class represents a single node in the linked list. It has two properties which are data and next. The data property is used to store the actual data of the node, whereas the next property is a reference to the next node in the list. The Node class consists of a constructor that initializes the data and next property when creating a new Node.

```
class Node {    constructor(data) {      this.data = data;      this.next = null;
  }
}
```

2. The LinkedList class is a representation of the linked list itself. It has a head property that refers to the first node in the list. The LinkedList class also has a constructor that initializes the head property when creating a new LinkedList.

```
class LinkedList {    constructor() {      this.head = null;      this.tail = null;      this.length
= 0;
  }
}
```

3. The LinkedList class also consists of a method that allows you to insert, delete, and search for nodes in the list while simultaneously allowing other operations like printing the list, counting the elements, reversing the list and so on.

- ✓ Printing the linked list

You can print the elements of a linked list by traversing through the list and printing the data of each node.

```
printAll() {
  let current = this.head;
while (current) {
console.log(current.data);
current = current.next;
  }
}
```

- ✓ Adding Node to the Linked List

There are multiple methods to add data to a linked list depending on where the new node has to be inserted, and are as follows –

- Adding node to the beginning of the linked list

To add node/ element at the start of a linked list, once a new node is created with the data, simply set its next property to the current head of the list. Then you can update the head of the list to the new node. This is also known as Insertion at the head of the linked list and is the most basic type of addition of data. It is simply done by calling the add function defined below.

```
add(data) {
const newNode = newNode(data);
  if (!this.head) {
 this.head = newNode;
  this.tail = newNode;
   } else {
this.tail.next = newNode;
  this.tail = newNode;
   }
this.length++;
   return this;
}
```

- Adding node to the end of the linked list

To add node/ element at the end of a linked list, we need to traverse the list and find the last node. After which a new node with data is created and we set the next property of the last node to the new node. This is also known as Insertion at the tail of the linked list and is the second most basic type of addition of data. It is simply done by calling the addToTail function defined below.

```
addToTail(data) {
let newNode = new Node(data);
if (this.head === null) {
this.head = newNode;
return;
}
let current = this.head;
while (current.next !== null) {
current = current.next;
  }
  current.next = newNode;
}
```

- Adding node at a specific position

To add node/ element at a specific position in a linked list, you can traverse the list to find the node at the position before the insertion point, create a new node with the data, set the next property of the new node to the current node at the position, and set the next property of the previous node to the new node.

```
 addAtPosition(data, position) {    let newNode = new Node(data);    if (position === 1)
 {      newNode.next = this.head;      this.head = newNode;      return;    }
 let current = this.head;
let i = 1;    while (i < position – 1 && current) {
 current = current.next;
 i++;
  }
 if (current) {
 newNode.next = current.next;      current.next = newNode;
   } }
```

Example (Adding Nodes to the Linked List)

In the below example, we implement adding nodes at beginning, at end and at a specific position.

```
 // this function is used to iterate over the entire linkedlist and print    it
   printAll()   {                 let   current   =   this.head;                 while   (current)   {
 console.log(current.data);        current = current.next;
      }
   } } const list = new LinkedList(); // add elements to the linkedlist list.add("node1");
 list.add("node2");  list.add("node3");  list.add("node4");  console.log("Initial  List:");
 list.printAll();    console.log("List    after    adding    nodex    at    position    2");
 list.addAtPosition("nodex",2);  list.printAll();  console.log("List  after  adding  nodey  to
 tail"); list.addToTail("nodey"); list.printAll();
```

Output

```
Initial List: node1 node2 node3 node4
List after adding nodex at position 2 node1 nodex node2 node3 node4
List after adding nodey to tail node1 nodex
node2 node3 node4 nodey
```

✓ Removing Node from the Linked List

Removal of data too, can be done via several methods depending upon the requirement.

- Removing a specific node

To remove a specific node from a linked list, we need to traverse the list and find the node before the one you want to remove, update its next property to skip over the node you want to remove, and update the reference to the next node. This removes the node based upon the value.

```
remove(data) {
   if (!this.head) {      return null;
   }     if (this.head.data === data) {        this.head = this.head.next;        this.length--;
 return this;
   }
   let current = this.head;    while (current.next) {       if (current.next.data === data) {
 current.next = current.next.next;        this.length--;        return this;
      }
      current = current.next;
   }   return null;
 }
```

- Removing a node at a Specific Position

To remove a node at a specific position in a linked list, we need to traverse the list and find the node before the one you want to remove, update its next property to skip over the node you want to remove, and update the reference to the next node. This is basically removing the node based upon the index value of it.

```
removeAt(index) {    if (index < 0 || index >= this.length) return null;    if (index === 0)
return this.remove();     let current = this.head;     for (let i = 0; i < index - 1; i++) {
current = current.next;
   }
   current.next = current.next.next;    this.length--;    return this; }
```

Example (Removing Nodes from the Lined List)

In the below example, we implement removing a specific node and a node at a specific position.

```
class Node {    constructor(data) {      this.data = data;      this.next = null;
   } } class LinkedList {   constructor() {     this.head = null;     this.tail = null;     this.length
= 0;
   }
   // function to add data to linked list    add(data) {
    const newNode = new Node(data);      if (!this.head) {        this.head = newNode;
this.tail = newNode;
    }      else {        this.tail.next = newNode;        this.tail = newNode;
    }      this.length++;      return this;
  }
  // function to remove data from linked list     remove(data) {        if (!this.head) {
return null;
    }
    if (this.head.data === data) {          this.head = this.head.next;          this.length--;
return this;
    }      let current = this.head;      while (current.next) {        if (current.next.data ===
data) {          current.next = current.next.next;          this.length--;          return this;
      }
      current = current.next;
    }      return null;
  }
  // function to remove from a particular index       removeAt(index) {      if (index < 0
|| index >= this.length) return null;       if (index === 0) return this.remove();       let
current = this.head;      for (let i = 0; i < index - 1; i++) {        current = current.next;
    }
    current.next = current.next.next;      this.length--;      return this;
  }
  // this function is used to iterate over the entire linkedlist and print it    printAll() {
let current = this.head;     while (current) {       console.log(current.data);       current
= current.next;
    }
  } } const list = new LinkedList(); // add elements to the linkedlist list.add("node1");
list.add("node2");  list.add("node3");  list.add("node4");  console.log("Initial  List:");
list.printAll();
console.log("List after removing node2"); list.remove("node2"); list.printAll();
console.log("List after removing node at index 2"); list.removeAt(2); list.printAll();
```

Output

```
Initial List: node1 node2 node3 node4
List after removing node2 node1 node3 node4
List after removing node at index 2 node1 node3
```

Practical Activity 3.1.3: Implementing Array Data structure

Task:

1: Read the task described below:

Given an array containing numbers 1 through n but with one missing number, find the missing number.

2: Follow the demonstration of trainer

3: Individually perform the task 3.1.3 referring to the demonstration

4: Ask for assistance where it is needed.

5: Read the key readings 3.1.3 in trainee's Manual

Key readings 3.1.3:Implementing Array Data structure

1. Array

The JavaScript **Array** object lets you store multiple values in a single variable. An array is used to store a sequential collection of multiple elements of same or different data types. In JavaScript, arrays are dynamic, so you don't need to specify the length of the array while defining the array.

The size of a JavaScript array may decrease or increase after its creation.

Syntax

Use the following syntax to create an **Array** object in JavaScript −

```
const arr = new Array(val1, val2, val3, ..., valN)
```

Parameters

val1, val2, val3, ..., valN − It takes multiple values as an argument to initialize an array with them.

Return value

It returns an array object.

When you pass the single numeric argument to the Array() constructor, it defines the array of argument length containing the undefined values. The maximum length allowed for an array is 4,294,967,295.

You can add multiple comma separated elements inside square brackets to create an array using the array literal −

```
const fruits = [ "apple", "orange", "mango" ];
```

You will use ordinal numbers to access and to set values inside an array as follows.

fruits[0] is the first element

fruits[1] is the second element

fruits[2] is the third element

Array Properties

Here is a list of the properties of the Array object along with their description −

Sr.No.	Property & Description
1	**constructor** Returns a reference to the array function that created the object.
2	**length** Reflects the number of elements in an array.
3	**prototype** The prototype property allows you to add properties and methods to an object.

Array Methods

Here is a list of the methods of the Array object along with their description −

Array Static methods

These methods are invoked using the Array class itself:

Sr.No.	Method & Description
1	**from()** Creates a shallow copy of the array.
2	**isArray()** Returns boolean values based on the argument is an array.
3	**Of()** Creates an array from multiple arguments.

Example: Creating JavaScript Array Object

In the example below, the array 'strs' is initialized with the string values passed as an Array() constructor's argument.

The 'cars' array contains 20 undefined elements. If you pass multiple numeric values, it defines the array containing those elements but needs to be careful with a single numeric argument to the array() constructor.

```
<html>
<head>
  <title> JavaScript - Array() constructor </title>
</head>
<body>
  <p id = "demo"> </p>
  <script>
    const output = document.getElementById("demo");      let strs = new
Array("Hello", "World!", "Tutorials Point");      output.innerHTML += "strs ==> " + strs
+ "<br>";
    let cars = new Array(20);      output.innerHTML += "cars ==> " + cars + "<br>";
  </script>
</body>
</html>
```

Output strs ==>

```
Hello,World!,Tutorials Point
cars ==> ,,,,,,,,,,,,,,,,,,,
```

Example: Creating Arrays Using Array Literal

In the example below, we have created different arrays. The arr1 array contains the numbers, the arr2 array contains the strings, and the arr3 array contains the boolean values.

```
<html>
<head>
  <title> JavaScript - Array literals </title>
</head>
<body>
  <p id = "output"> </p>
  <script>      const arr1 = [10, 40, 50, 60, 80, 90];  // Array of numbers
     const arr2 = ["Hello", "Hi", "How", "are", "you?"]; // Array of strings
const arr3 = [true, false, true, true]; // Array of booleans
    document.getElementById("output").innerHTML =
        "arr1 ==>  " + arr1 + "<br>" +
    "arr2 ==>  " + arr2 + "<br>" +
    "arr3 ==>  " + arr3;
  </script>
</body>
</html>
```

Output

```
arr1 ==> 10,40,50,60,80,90
arr2 ==>
Hello,Hi,How,are,you?
arr3 ==> true,false,true,true
```

Accessing JavaScript Array Elements

The array index starts from 0. So, you can access the array element using its index.

```
let number = arr[index]
```

In the above syntax, 'arr' is an array, and 'index' is a number from where we need to access the array element.

Example

In the example below, we have created the array of numbers and accessed the elements from the 0th and 2nd index of the array. The element at the 0th index is 1, and the element at the 2nd index is 6.

```
<html>
<head>
  <title> JavaScript - Accessing array elements </title>
</head>
<body>
  <p id = "output"> </p>
  <script>      const nums = [1, 5, 6, 8, 90];
document.getElementById("output").innerHTML =
    "Element at 0th index is : " + nums[0] + "<br>" +
    "Element at 2nd index is : " + nums[2];
  </script>
</body>
</html>
```

Output

```
Element at 0th index is : 1
Element at 2nd index is : 6
```

JavaScript Array length

The 'length' property of the array is used to find the length of the array.

```
let len = arr.length;
```

Example

In the example below, the 'length' property returns 5, as array contains 5 elements.

```
<html>
<head>
  <title> JavaScript - Array length </title>
</head>
<body>
  <p id = "output"> </p>
  <script>      const nums = [1, 5, 6, 8, 90];
document.getElementById("output").innerHTML =
    "Array length is : " + nums.length;
  </script>
</body>
</html>
```

Output

```
Array length is : 5
```

Adding a new element to the array

You can use the push() method to insert the element at the end of the array. Another solution is that you can insert the array at the index equal to the array length.

```
arr.push(ele)
OR
arr[arr.length] = ele;
```

In the above syntax, 'ele' is a new element to insert into the array. Here, if the array length is N, the array contains elements from 0 to N – 1 index. So, we can insert the new element at the Nth index.

Example

In the example below, we insert 6 to the array using the push() method. Also, we used the 'length' property to insert the element at the end.

```
<html>
<body>
  <p id = "output"> </p>
```

```
    <script>
        const output = document.getElementById("output");
        const nums = [1, 2, 3, 4, 5];
        nums.push(6); // Inserting 6 at the end
        output.innerHTML += "Updated array is : " + nums + "<br>";
        nums[nums.length] = 7; // Inserting 7
        output.innerHTML += "Updated array is : " + nums + "<br>"
    </script>
</body>
</html>
```

Output

```
Updated array is : 1,2,3,4,5,6
Updated array is : 1,2,3,4,5,6,7
```

Updating Array Elements

To update any array element, you can access the array index and change its value.

```
    arr[index] = ele;
```

In the above syntax, 'index' is an index where we need to update a value with the 'ele' value.

Example

In the example below, we update the element at the first index in the array.

```
<html>
<body>
  <p id = "output"> </p>
  <script>
      const nums = [1, 2, 3, 4, 5];
      nums[0] = 100; // Updating first element
     document.getElementById("output").innerHTML =
      "Updated array is : " + nums;
  </script>
</body>
</html>
```

Output

```
Updated array is : 100,2,3,4,5
```

Traversing the Arrays

You can use the loop to traverse through each array element. However, some built-in methods exist to traverse the array, which we will see in later chapters.

```
for (let p = 0;
 p < nums.length;
 p++) {
   // Access array using the nums[p]
}
```

Example

In the below code, the array contains 5 numbers. We used the for loop to traverse the array and print each element.

However, while and do-while loops can also be used to traverse the array.

```
<html>
<body>
   <p id = "demo"> </p>
   <script>
      const output = document.getElementById("demo");
       const nums = [1, 2, 3, 4, 5];
       for (let p = 0; p < nums.length; p++) {
          output.innerHTML += "nums[" + p + "] ==> " + nums[p] +
"<br>";
      }
   </script>
</body>
</html>
```

Output

```
nums[0] ==> 1 nums[1] ==> 2 nums[2] ==> 3 nums[3] ==> 4 nums[4] ==> 5
```

Situatuation

XYZ Police Station monitors traffic at four key intersections, and they have collected vehicle count data for different hours of the day. The data is organized in the following 2D array, where rows represent intersections and columns represent hours:

Intersection/Hour	H	H	H	H
I1	5	6	5	4

I2	7	6	8	7
I3	3	4	3	5
I4	9	8	9	8

Based on this data, answer the following:

1. Calculate the total number of vehicles passing through each intersection over the entire day.
2. Determine which hour had the highest total number of vehicles across all intersections.
3. Identify which intersection has the highest average number of vehicles per hour.
4. Recommend a change in traffic light timing to improve traffic flow during peak hours, based on your analysis. Explain your recommendation

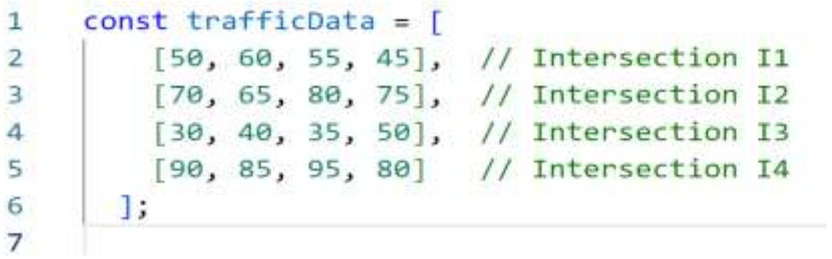
```
// Define the data array
const trafficData = [
  [50, 60, 55, 45],  // Intersection I1
  [70, 65, 80, 75],   // Intersection I2
  [30, 40, 35, 50],  // Intersection I3
  [90, 85, 95, 80]   // Intersection I4
];
// 1. Calculate the total number of vehicles passing through each intersection
const totalVehiclesPerIntersection = trafficData.map(intersection => {
  return intersection.reduce((sum, count) => sum + count, 0);
});
// 2. Determine which hour had the highest total number of vehicles
const totalVehiclesPerHour = [0, 1, 2, 3].map(hourIndex => {
  return trafficData.reduce((sum, intersection) => sum + intersection[hourIndex],
0);
});
const peakHourIndex =
totalVehiclesPerHour.indexOf(Math.max(...totalVehiclesPerHour));
// 3. Identify which intersection has the highest average number of vehicles per
hour
```

```
const averageVehiclesPerIntersection = totalVehiclesPerIntersection.map(total => total / 4);
const highestAverageIntersectionIndex = averageVehiclesPerIntersection.indexOf(Math.max(...averageVehiclesPerIntersection));
// 4. Recommend a change in traffic light timing
// Assuming more green light should be allocated to intersections with high vehicle counts during peak hours
const peakHourVehicles = totalVehiclesPerHour[peakHourIndex];
const vehiclesAtPeakHourPerIntersection = trafficData.map(intersection => intersection[peakHourIndex]);

const highestPeakHourIntersectionIndex = vehiclesAtPeakHourPerIntersection.indexOf(Math.max(...vehiclesAtPeakHourPerIntersection));
console.log('Total vehicles per intersection:', totalVehiclesPerIntersection);
console.log('Total vehicles per hour:', totalVehiclesPerHour);
console.log('Peak hour index (H1=0, H2=1, H3=2, H4=3):', peakHourIndex);
console.log('Intersection with highest average vehicles per hour:', highestAverageIntersectionIndex);
console.log('Recommended adjustment for traffic light timing at peak hour intersection:', highestPeakHourIntersectionIndex);

// For readability, you might also want to map these indices back to meaningful names
const intersectionNames = ['I1', 'I2', 'I3', 'I4'];
const hourNames = ['H1', 'H2', 'H3', 'H4'];

console.log('Peak hour:', hourNames[peakHourIndex]);
console.log('Intersection with highest average vehicles per hour:', intersectionNames[highestAverageIntersectionIndex]);
```

console.log('Recommended adjustment for traffic light timing at intersection:', intersectionNames[highestPeakHourIntersectionIndex]);

Explanation:

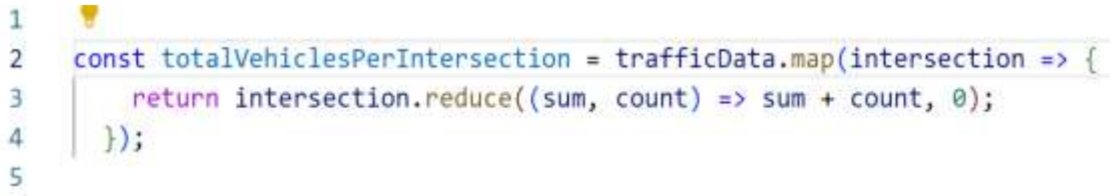
Data Initialization



```
const trafficData = [
    [50, 60, 55, 45],  // Intersection I1
    [70, 65, 80, 75],  // Intersection I2
    [30, 40, 35, 50],  // Intersection I3
    [90, 85, 95, 80]   // Intersection I4
  ];

```

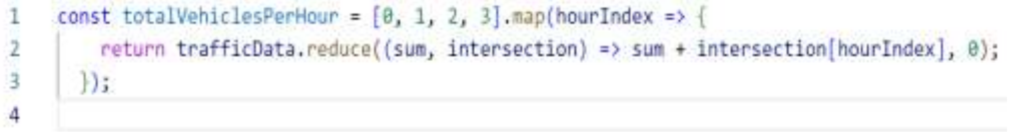
This array represents the number of vehicles passing through four intersections (I1, I2, I3, I4) at four different hours (H1, H2, H3, H4). Each inner array corresponds to one intersection, with each number representing the vehicle count at a specific hour.

1. **Calculate the Total Number of Vehicles Passing Through Each Intersection**



```

const totalVehiclesPerIntersection = trafficData.map(intersection => {
    return intersection.reduce((sum, count) => sum + count, 0);
  });

```

- **trafficData.map(...)** iterates over each intersection.
- **intersection.reduce((sum, count) => sum + count, 0)** calculates the total number of vehicles for each intersection by summing up all counts in the intersection array.
- **totalVehiclesPerIntersection** will be an array where each element represents the total number of vehicles for an intersection



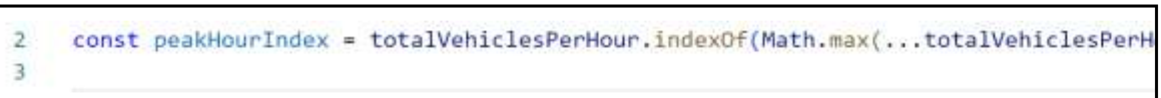
```
const totalVehiclesPerHour = [0, 1, 2, 3].map(hourIndex => {
    return trafficData.reduce((sum, intersection) => sum + intersection[hourIndex], 0);
  });

```

2. Determining Hour Had the Highest Total Number of Vehicles

[0, 1, 2, 3].map(hourIndex => ...) iterates over each hour index (H1 to H4).

trafficData.reduce((sum, intersection) => sum + intersection[hourIndex], 0) sums up the vehicle counts for a specific hour across all intersections.

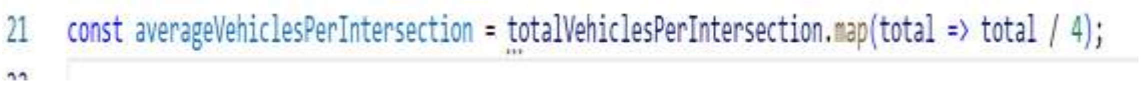
totalVehiclesPerHour will be an array where each element represents the total number of vehicles for an hour.



```
const peakHourIndex = totalVehiclesPerHour.indexOf(Math.max(...totalVehiclesPerH
```

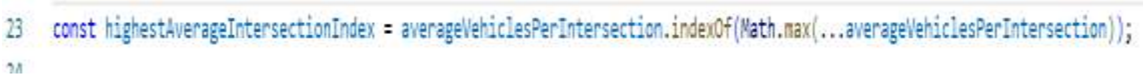
- **Math.max(...totalVehiclesPerHour)** finds the highest value in the **totalVehiclesPerHour** array.
- **totalVehiclesPerHour.indexOf(...)** gets the index of this maximum value, which corresponds to the hour with the highest total vehicle count.

2. **Identification of Intersection has the Highest Average Number of Vehicles per Hour**



```
const averageVehiclesPerIntersection = totalVehiclesPerIntersection.map(total => total / 4);
```

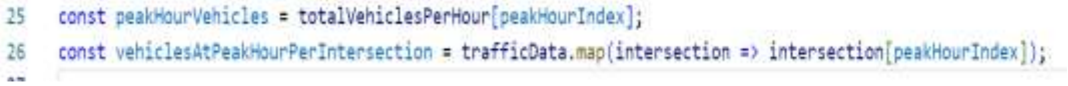
totalVehiclesPerIntersection.map(total => total / 4) calculates the average number of vehicles per hour for each intersection by dividing the total vehicles by the number of hours (4).



```
const highestAverageIntersectionIndex = averageVehiclesPerIntersection.indexOf(Math.max(...averageVehiclesPerIntersection));
```

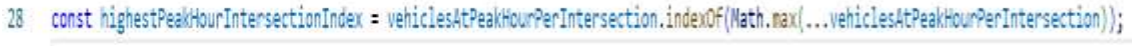
- **Math.max(...averageVehiclesPerIntersection)** finds the highest average number of vehicles per hour.
- **averageVehiclesPerIntersection.indexOf(...)** gets the index of this maximum average, which corresponds to the intersection with the highest average vehicle count.

3. **Recommend a Change in Traffic Light Timing**



```
const peakHourVehicles = totalVehiclesPerHour[peakHourIndex];
const vehiclesAtPeakHourPerIntersection = trafficData.map(intersection => intersection[peakHourIndex]);
```

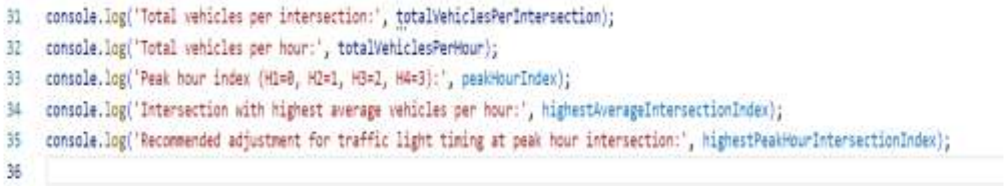
- **totalVehiclesPerHour[peakHourIndex]** retrieves the total number of vehicles at the peak hour.
- **trafficData.map(intersection => intersection[peakHourIndex])** creates an array of vehicle counts at the peak hour for each intersection.



```
const highestPeakHourIntersectionIndex = vehiclesAtPeakHourPerIntersection.indexOf(Math.max(...vehiclesAtPeakHourPerIntersection));
```

- **Math.max(...vehiclesAtPeakHourPerIntersection)** finds the highest vehicle count at the peak hour across all intersections.
- **vehiclesAtPeakHourPerIntersection.indexOf(...)** gets the index of this maximum

value, which corresponds to the intersection with the highest vehicle



```
console.log('Total vehicles per intersection:', totalVehiclesPerIntersection);
console.log('Total vehicles per hour:', totalVehiclesPerHour);
console.log('Peak hour index (H1=0, H2=1, H3=2, H4=3):', peakHourIndex);
console.log('Intersection with highest average vehicles per hour:', highestAverageIntersectionIndex);
console.log('Recommended adjustment for traffic light timing at peak hour intersection:', highestPeakHourIntersectionIndex);
```

Count the peak hour.

❖ **Output the Results**

These console.log statements output the results of the calculations:

- **totalVehiclesPerIntersection** shows the total vehicle count for each intersection.
- **totalVehiclesPerHour shows** the total vehicle count for each hour.
- **peakHourIndex** indicates which hour has the highest total vehicle count.
- **highestAverageIntersectionIndex** shows which intersection has the highest average number of vehicles per hour.
- **highestPeakHourIntersectionIndex** recommends which intersection should get more green light during peak hours

Practical Activity 3.1.4: Implementing Queue Data Structure

Task:

1: Read the task described below:

Create a queue class with enqueue (), dequeue(), and peek() methods using an array.

enqueue('A') adds an element, and dequeue ()

i. Removes the front element

ii.Write a function is Empty() to check if the queue is empty or not, and demonstrate its use

2: Follow the demonstration of trainer

3: Individually perform the task 3.1.4 referring to the demonstration

4: Ask for assistance where it is needed.

5: Read the key readings 3.1.4 in trainee’s Manual

Key readings 3.1.4: Implementing Queue Data Structure

Queue

Implementation of Queue in JavaScript.

A Queue works on the FIFO(First in First Out) principle. Hence, it performs two basic operations which are the addition of elements at the end of the queue and the removal of elements from the front of the queue. Like Stack, Queue is also a linear data structure.

To implement a queue data structure we need the following methods:

- **enqueue** : To add elements at the end of the queue.
- **dequeue:** To remove an element from the front of the queue.
- **peek:** To get the front element without removing it.
- **isEmpty:** To check whether an element is present in the queue or not.
- **printQueue:** To print the elements present in the queue.

Now, we will see the practical implementation of these queue operations, which are as follows:

1) **enqueue ():** The queue operation which is used for adding elements to the queue.

Example:

```
enqueue(ele) {
    if(this.rear < this.size ) {         this.data[this.rear] = ele;         thisthis.rear
= this.rear + 1;
    }
}
}
```

In the above code, we have added elements to the queue using the push function.

2) **dequeue ():** The queue operation which is used for removing or popping out the existing values from the queue.

Example:

```
dequeue() {       if(this.isEmpty() === false) {            thisthis.rear = this.rear-1;
return this.data.shift();
        }
    }
```

In the above code, firstly, we have checked if the queue is already empty or not. It is because if the queue has no value, it will return "Underflow". Else, it will check and return the element.

3) **Length ():** The queue operation is used to return the length of the queue.

Example:

```
length() {

    return this.rear;
 }
```

The statement this.rear will help to fetch the length of the queue.

4) **isEmpty ():** The queue operation is used to check whether the queue is empty or not. If the queue is found empty, it returns true. Otherwise, it returns false.

Example:

```
        isEmpty() {      return this.rear === 0;
        }
```

In the above code, it will check if the value of the rear, i.e., the end, is equal to 0 or not. If it is true, it will return true else false will be returned.

5) **print ():** The queue operation is used to print the elements of the queue from the index value 0 to the rear position of the queue.

Example:

```
    print() {
        for(let i =0; i < this.rear; i++) {        console.log(this.data[i]);
           }
        }
```

In the above code, using for loop and beginning from the 0 indexes to the queue's rear position, it will print the value and put it in the data array.

6) **clear ():** The queue operation is used to clear or delete all the elements of the queue and makes the value of the rear equal to 0.

Example:

```
        clear() {     this.data.length =0;
            this.rear = 0;
        }
```

In the above code, using the clear () operation, the value in the data array becomes 0 and sets the rear value to 0.

Implementing the JavaScript Queue The complete code:

```
class Queue {
     constructor(){
     this.data = [];
     this.rear = 0;
    this.size = 20;
 }
  enqueue(ele) {
     if(this.rear < this.size ) {
           this.data[this.rear] = ele;
           thisthis.rear = this.rear + 1;
   }
   }
   length() {
      return this.rear;
 }
 isEmpty() {
     return this.rear === 0;
 }
   getFront() {
     if(this.isEmpty() === false) {
         return this.data[0];
  }
   }
   getLast() {
      if(this.isEmpty() === false) {
           return this.data[ this.rear - 1 ] ;
   }
 }
 dequeue() {
      if(this.isEmpty() === false) {
           thisthis.rear = this.rear-1;
           return this.data.shift();
   }
   }
   print() {
     for(let i =0; i < this.rear; i++) {
       console.log(this.data[i]);
  }
   }
    clear() {
       this.data.length = 0;
```

```
        this.rear = 0;
  }
}
```

Another example:

```
 class Queue {
    constructor() {
          this.items = {}
          this.frontIndex = 0
          this.backIndex = 0
    }
    enqueue(item) {
          this.items[this.backIndex] = item
          this.backIndex++
          return item + ' inserted'
    }
    dequeue() {
          const item = this.items[this.frontIndex]               delete this.items[this.frontIndex]
          this.frontIndex++
          return item
    }
    peek() {
          return this.items[this.frontIndex]
    }
 get printQueue() {   return this.items;
    }
}
 const queue = new Queue() console.log(queue.enqueue(7)) console.log(queue.enqueue(2))
    console.log(queue.enqueue(6))                                console.log(queue.enqueue(4))
    console.log(queue.dequeue())  console.log(queue.peek())  let str = queue.printQueue;
    console.log(str)
```

Practical Activity 3.1.5: Implementing Stack Data Structure

Task:

1: Read the task described below:

Create a stack class with methods for push (), pop(), and peek().

2: Follow the demonstration of trainer
3: Individually perform the task 3.1.5 referring to the demonstration
4: Ask for assistance where it is needed.
5: Read the key readings 3.1.5 in trainee's Manual

Key readings 3.1.5: Implementing Stack Data Structure

Stack

In this article, we are going to discuss how to create the stack data structure in JavaScript. It is a linear data structure where the push and popping of elements follow the LIFO (last in first out and FILO (first in last out) sequence.

Though Arrays in JavaScript provide all the functionality of a Stack, let us implement our own Stack class. Our class will have the following functions.

- push(element) − Function to push elements on top of the stack.
- pop() − Function that removes an element from the top and returns it.
- peek() − Returns the element on top of the stack.
- isFull() − Checks if we reached the element limit on the stack.
- isEmpty() − checks if the stack is empty.
- clear() − Remove all elements.
- display() − display all contents of the array

Let's start by defining a simple class with a constructor that takes the max size of the stack and a helper function display() that'll help us when we implement the other functions for this class. We have also defined 2 more functions, isFull and isEmpty to check if the stack is full or empty.

The isFull function just checks if the length of the container is equal to or more than maxSize and returns accordingly.

The isEmpty function checks if a size of the container is 0.

These will be helpful when we define other operations. The functions we define from this point onwards will all go inside the Stack class.

Example 1

The following example demonstrates how to create the Stack Data Structure in JavaScript. In this example, we are going to discuss the use of push(), and the pop()

methods. We create a stack and add the elements to it and display the stack elements.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Creating Stack Data Structure</title>
  </head>
  <body>
<script type="text/javascript">
class Stack {
constructor() {
this.stkArr = [];
}
// add element to the stack
add(element) {
return this.stkArr.push(element);
}
// remove element from the stack
remove() {
if (this.stkArr.length > 0) {
document.write("<br>");
return "The Popped element is : " + this.stkArr.pop();
}
}
// view the last element
peek() {
document.write("<br>");
return (
"The Peek element of the stack is : " + this.stkArr[this.stkArr.length - 1] );
}
// check if the stack is empty
isEmpty() {
document.write("<br>");
return this.stkArr.length == 0;
}
// the size of the stack
size() {
document.write("<br>");
return "The size of the stack is : " + this.stkArr.length;
 }
display() {
if (this.stkArr.length !== 0) {
return "The stack elements are : " + this.stkArr + "<br>";
} else {
document.write("The Stack is Empty..! <br>");
}
}
// empty the stack
clear() {
document.write("The stack is cleared..!" + "<br>");
this.stkArr = [];
}
}
let stack = new Stack();
stack.add(1);
stack.add(2);
stack.add(3);
stack.add(4);
stack.add(5);
document.write(stack.display());
document.write(stack.size());
    </script>   </body>
```

```
</html>
```

Example 2

Another example to execute the operations of stack in JavaScript is given as follows −

```
class Stack {
constructor(maxSize) {
if (isNaN(maxSize)) {
maxSize = 10;
    }
    this.maxSize = maxSize;
this.container = [];
  }   display() {
console.log(this.container);
  }
  push(element) {
this.container.push(element);
  } } const obj = new Stack(10);
obj.push(10);
obj.push(44);
obj.push(55);
obj.display();
```

Practical Activity 3.1.6: Implementing Tree Data Structure

Task:

1: Read the task described below:

Write a function to calculate the height of a binary tree.

2: Follow the demonstration of trainer

3: Individually perform the task 3.1.6 referring to the demonstration

4: Ask for assistance where it is needed.

5: Read the key readings 3.1.6 in trainee's Manual

Key readings 3.1.6: Implementing Tree Data Structure

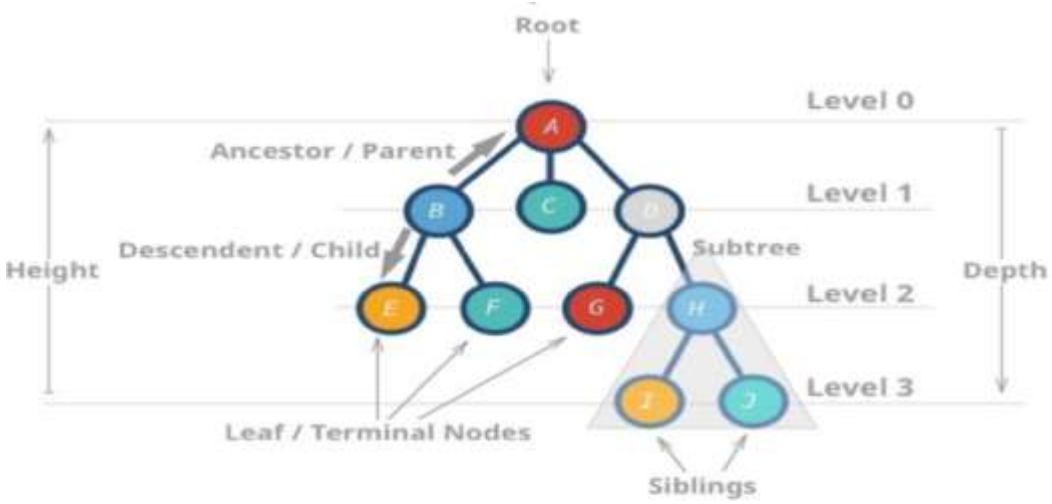
Tree Data Structure

Tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.

Let's understand some key points of the Tree data structure.

- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string.
- Each node contains some data and the link or reference of other nodes that can be called children.



Here are some properties of trees:

- The top-most node is called **root**.
- A node without children is called **leaf** node or **terminal** node.

- **Height** (h) of the tree is the distance (edge count) between the farthest leaf to the root.

A has a height of 3 o I has a height of 0

- **Depth** or **level** of a node is the distance between the root and the node in question.

H has a depth of 2 o B has a depth of 1

Implementing a simple tree data structure

As we saw earlier, a tree node is just a data structure with a value and links to its descendants.

Here's an example of a tree node:

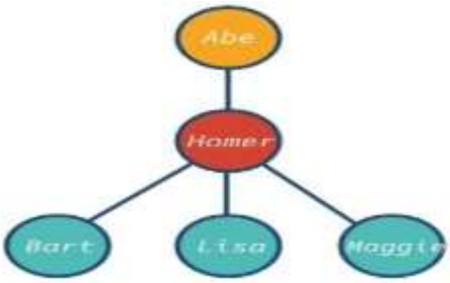
```
 class
TreeNode {
constructo
r(value)    {
this.value
=      value;
this.desce
ndants = [];
   }
 }
```

```
 // create nodes with values
const abe = new
TreeNode('Abe'); const homer
= new TreeNode('Homer');
```

We can create a tree with 3 descendants as follows:

const bart = new TreeNode('Bart'); const lisa = new TreeNode('Lisa'); const maggie = new TreeNode('Maggie'); // associate root with is descendants abe.descendants.push(homer); homer.descendants.push(bart, lisa, maggie);

That's all; we have a tree data structure!

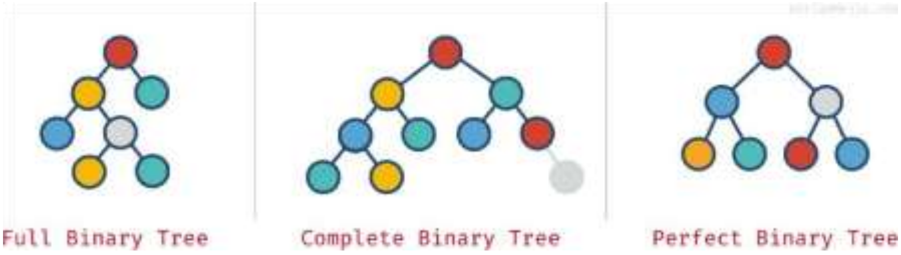


Full, Complete, and Perfect binary trees

Depending on how nodes are arranged in a binary tree, it can be **full**, **complete** and **perfect**:

- **Full binary tree**: each node has exactly 0 or 2 children (but never 1).
- **Complete binary tree**: when all levels except the last one are **full** with nodes.
- **Perfect binary tree**: when all the levels (including the last one) are full of nodes.

Look at these examples:



These properties are not always mutually exclusive. You can have more than one:

- A perfect tree is **always** complete and full.

✓ Perfect binary trees have precisely 2k−12k-1 nodes, where k is the last level of the tree (starting with 1).

- A complete tree is **not** always full.

✓ Like in our "complete" example, since it has a parent with only one child. If we remove the rightmost gray node, then we would have a **complete** and **full** tree but not perfect.

- A full tree is not always complete and perfect.

Each node in a tree data structure must have the following properties:			
		key	: The key of the node : The value of the node : The parent of the node (null if there is none) children: An array of pointers to the node's children
		value parent	

Th e main operations of a tree data structure are:			
		insert	: Inserts a node as a child of the given parent node : Removes a node and its children from the tree Retrieves a given node preOrderTraversal: Traverses the tree by recursively traversing each node followed by its postOrderTraversal: Traverses the tree by recursively traversing each node's children followed by the node
		remove find: children	

	<u>Implementation</u>

```
class TreeNode {
  constructor(key, value = key, parent = null) {
    this.key = key;
    this.value = value;
    this.parent = parent;
    this.children = [];
  }
  get isLeaf() {
    return this.children.length === 0;
  }
  get hasChildren() {
    return !this.isLeaf;
  } }
 class Tree {
  constructor(key, value = key) {
    this.root = new TreeNode(key, value);
  }
  *preOrderTraversal(node = this.root) {
    yield node;
    if (node.children.length) {
      for (let child of node.children) {
        yield* this.preOrderTraversal(child);
      }
    }
  }
  *postOrderTraversal(node = this.root) {
    if (node.children.length) {
      for (let child of node.children) {
        yield* this.postOrderTraversal(child);
      }
    }
   yield node;
  }
  insert(parentNodeKey, key, value = key) {
    for (let node of this.preOrderTraversal()) {
```

Define a remove() method, that uses the preOrderTraversal() method and Array.prototype.filter() to remove a TreeNode from the tree.

Define a find() method, that uses the preorder traversal () method to retrieve the given node in the tree.

Hash Table

Hash table is one of the most important data structures that uses a special function known as a hash function that maps a given value with a key to access the elements faster.

A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., key and value. The hash table can be implemented with the help of an associative array. The efficiency of mapping depends upon the efficiency of the hash function used for mapping.

For example, suppose the key value is John and the value is the phone number, so when we pass the key value in the hash function shown as below:

Hash(key)= index;

When we pass the key in the hash function, then it gives the index.

Hash(john) = 3;

The above example adds the john at the index 3.

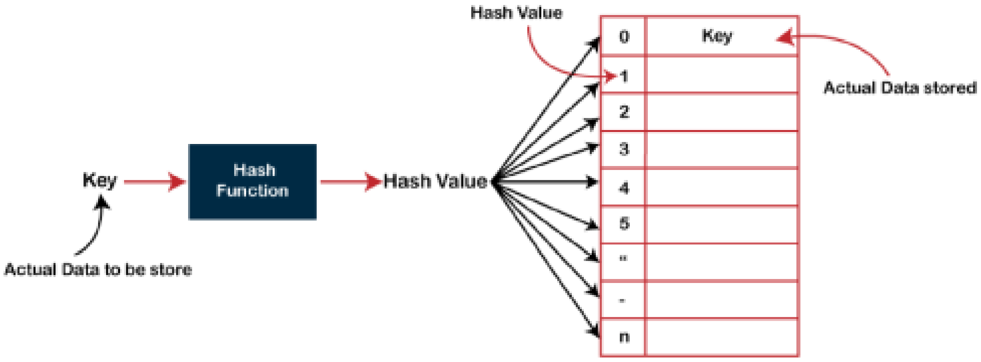
Drawback of Hash function

A Hash function assigns each value with a unique key. Sometimes hash table uses an imperfect hash function that causes a collision because the hash function generates the same key of two different values.

In Hashing technique, the hash table and hash function are used. Using the hash function, we can calculate the address at which the value can be stored.

The main idea behind the hashing is to create the (key/value) pairs. If the key is given, then the algorithm computes the index at which the value would be stored. It can be written as:

Index = hash(key)



Hash tables are made up of two parts:

- **Object:** An object with the table where the data is stored. The array holds all the keyvalue entries in the table. The size of the array should be set according to the amount of data expected.

- **Hash function (or mapping function):** This function determines the index of our key-value pair. It should be a one-way function and produce the a different hash for each key.

To implement a hash table using JavaScript, we will do three things:

⇨ Create a hash table class, add a hash function, and implement a method for adding key/value pairs to our table.

First, let’s create the HashTable class.

```
class HashTable {
  constructor() {
    this.values = {};
    this.length =  0;
    this.size =  0;
 }
}
```

The constructor contains an object in which we’re going to store the values, the length of the values, and the entire size of the hash table: meaning how many buckets the hash table contains.

We will be storing our data in these buckets.

Next, we have to implement a simple hashing function.

```
    calculateHash(key) {
      return key.toString().length % this.size;
    }
```

This function takes the provided key and returns a hash that’s calculated using an arithmetic modulus.

Finally, we need a method to insert key/value pairs. Take a look at the code and see this in action:

```
add(key, value) {
   const hash = this.calculateHash(key);
if(!this.values.hasOwnProperty(hash)) {
     this.values[hash] = {};
  }
  if(!this.values[hash].hasOwnProperty(key)) {
     this.length++;
  }
   this.values[hash][key] = value;
 }
```

The first thing we do here is calculate a hash for our key. If the hash does not already exist, we save it to our object store. The next check we perform is on the hash. If it doesn’t have a key saved, we save the key and value and increment the size of our hash table.

Searching in a hash table goes very fast. Unlike with an array where we have to go through all of the elements until we reach the item we need, with a hash table we simply get the index. I’ll add the complete code for our hash table implementation below.

```
 class
HashTab
le         {
construc
tor()      {
this.valu
es   =   {};
this.leng
th  =    0;
this.size
=  0;
   }
   calculateHash(key) {
return key.toString().length %
this.size;   }   add(key, value) {
     const hash =
this.calculateHash(key);     If
(!this.values.hasOwnProperty(h
ash)) {       this.values[hash] = {};
     }
     if (!this.values[hash].hasOwnProperty(key)) {
this.length++;
     }
     this.values[hash][key] = value;
   }
sear
ch(k
ey) {
      const hash = this.calculateHash(key);    if
(this.values.hasOwnProperty(hash) &&
this.values[hash].hasOwnProperty(key)) {
    return this.values[hash][key];
      }
else {
```

```
return
null;
     }
   }
 }
 //create   object   of
type  hash  table  const
ht  =  new  HashTable();
//add data to the hash
table                    ht
ht.add("Canada",
"300");
ht.add("Germany",
"100");   ht.add("Italy",
"50");
 //search
 console.log(ht.search("Italy"));
```

Practical Activity 3.1.7: Implementing Graph Data Structure

Task:

1: Read the task described below:

Implement a graph using an adjacency list and an adjacency matrix in JavaScript. Write code to add vertices and edges and print both the adjacency list and matrix

2: Follow the demonstration of trainer

3: Individually perform the task 3.1.7 referring to the demonstration

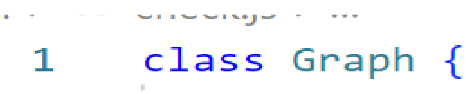
4: Ask for assistance where it is needed.

5: Read the key readings 3.1.7 in trainee's Manual

Key readings 3.1.7: Implementing Graph Data Structure

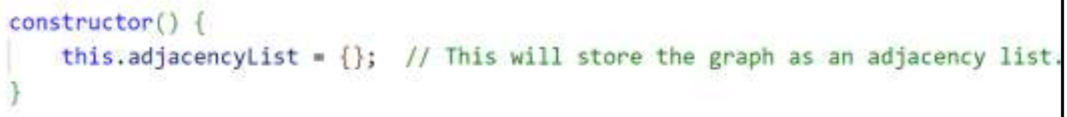
Graph

A graph consists of nodes (vertices) connected by edges. It can be represented in various ways, but a common one is using an Adjacency List



```
class Graph {
```

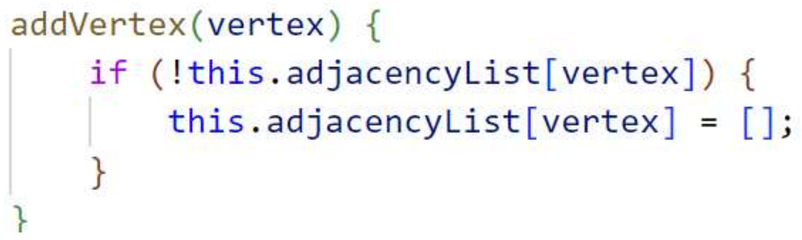
This defines a new class called Graph. In JavaScript, a class is a blueprint for creating objects



```
constructor() {
    this.adjacencyList = {};  // This will store the graph as an adjacency list.
}
```

- **constructor():** This is a special method that is called when an instance of the Graph class is created.
- **this.adjacencyList:** This is an object (a dictionary) where the keys are vertices, and the values are arrays representing the list of neighboring vertices (i.e., the edges of the graph). Initially, it's an empty object.

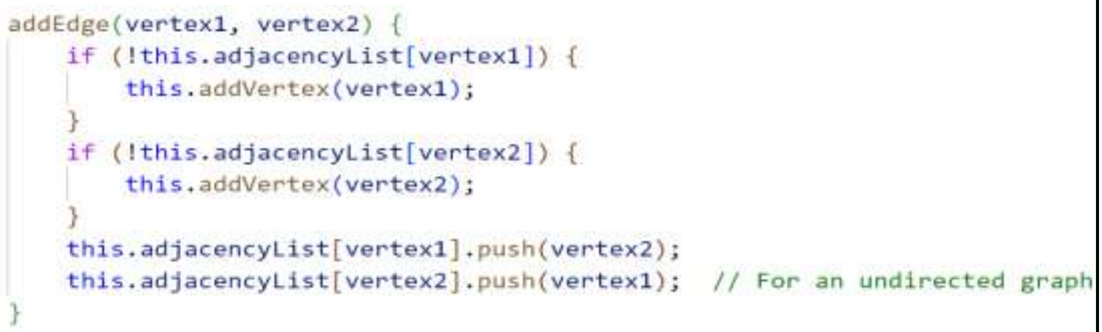
1.Add a Vertex:



```
addVertex(vertex) {
    if (!this.adjacencyList[vertex]) {
        this.adjacencyList[vertex] = [];
    }
}
```

- **addVertex(vertex):** This method adds a new vertex to the graph.
- **if (!this.adjacencyList[vertex]):** It checks if the vertex does not already exist in the adjacency list.
- **this.adjacencyList[vertex] = []:** If the vertex doesn't exist, it creates an empty array for the vertex to store its neighboring vertices (edges).

2.Add an Edge

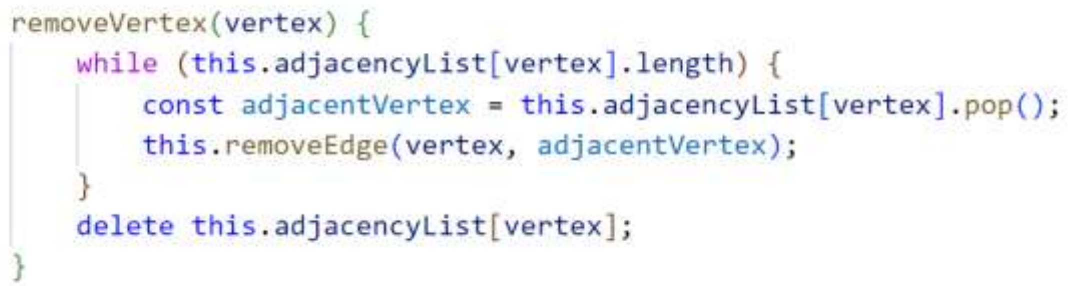


```
addEdge(vertex1, vertex2) {
    if (!this.adjacencyList[vertex1]) {
        this.addVertex(vertex1);
    }
    if (!this.adjacencyList[vertex2]) {
        this.addVertex(vertex2);
    }
    this.adjacencyList[vertex1].push(vertex2);
    this.adjacencyList[vertex2].push(vertex1);  // For an undirected graph
}
```

- **addEdge(vertex1, vertex2)**: This method connects two vertices by adding an edge between them.
- It first checks whether vertex1 and vertex2 exist. If they don't, it adds them by calling addVertex.
- this.adjacencyList[vertex1].push(vertex2): It adds vertex2 to the list of neighbors of vertex1.
- **this.adjacencyList[vertex2].push(vertex1)**: It adds vertex1 to the list of

neighbors of vertex2 (this is for an **undirected** graph; for a directed graph, you would only add one of these).

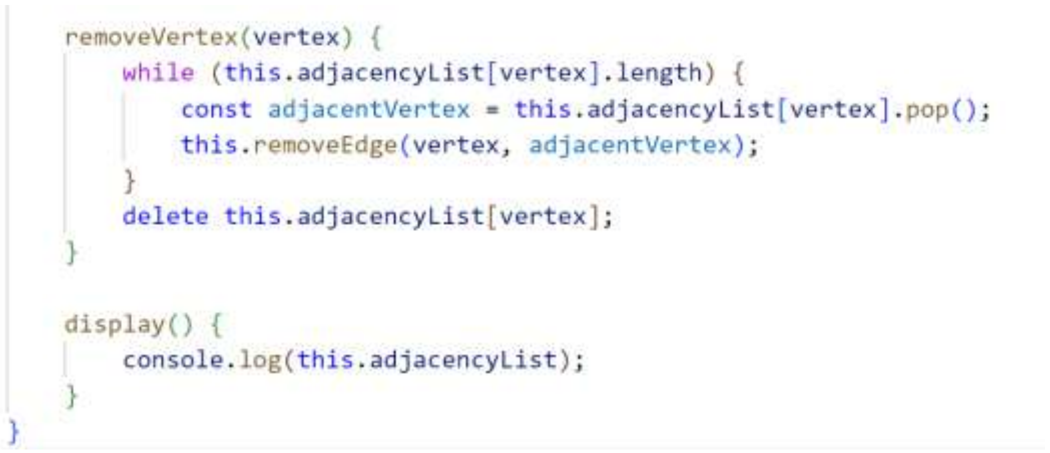
3. Remove an Edge



```
removeVertex(vertex) {
    while (this.adjacencyList[vertex].length) {
        const adjacentVertex = this.adjacencyList[vertex].pop();
        this.removeEdge(vertex, adjacentVertex);
    }
    delete this.adjacencyList[vertex];
}
```

- **RemoveEdge(vertex1, vertex2):** This method removes the edge between vertex1 and vertex2.
- **filter:** The filter() method is used to remove vertex2 from the list of vertex1's neighbors, and vice versa.

4. Remove a Vertex

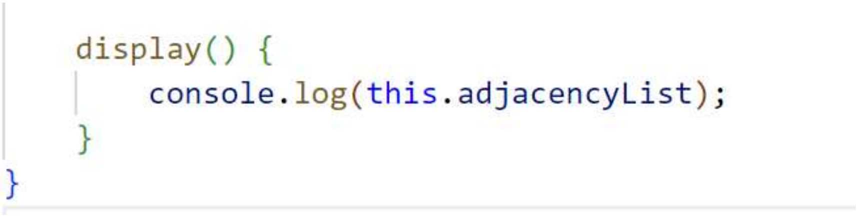


```
removeVertex(vertex) {
    while (this.adjacencyList[vertex].length) {
        const adjacentVertex = this.adjacencyList[vertex].pop();
        this.removeEdge(vertex, adjacentVertex);
    }
    delete this.adjacencyList[vertex];
}

display() {
    console.log(this.adjacencyList);
}
}
```

- **removeVertex(vertex)**: This method removes a vertex and all edges connected to it.
- **while (this.adjacencyList[vertex].length)**: It loops over all the neighbors of vertex.
- **pop()**: It removes the last neighbor from the adjacency list of vertex.
- **this.removeEdge(vertex, adjacentVertex)**: For each neighbor, it removes the edge between vertex and its neighbors.
- **delete this.adjacencyList[vertex]**: Finally, it deletes the vertex from the adjacency list.

4. Display the Graph

This method simply logs the adjacency list to the console to show the current structure of the graph.



```
    display() {
        console.log(this.adjacencyList);
    }
}
```

Work

Kiki ltd is road Construction Company located in Kigali city need to build road network using a graph. Each city intersection is a vertex, and each road is an edge between two intersections. As software development write JavaScript code using Graph to:

a) Add intersections (vertices).

b) Add roads (edges).

c) Remove roads and intersections.

d) Display the network of roads

Practical Activity 3.1.8: Implementing Table Data Structure

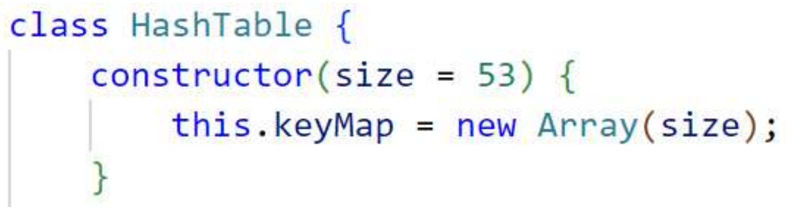
Task:

1: Read the task described below:
   Write a function that checks for duplicates in an array using a hash table
   Test with different arrays to find duplicates.
2: Follow the demonstration of trainer
3: Individually perform the task 3.1.8 referring to the demonstration
4: Ask for assistance where it is needed.
5: Read the key readings 3.1.8 in trainee's Manual

Key readings 3.1.8: Implementing Table Data Structure

Table

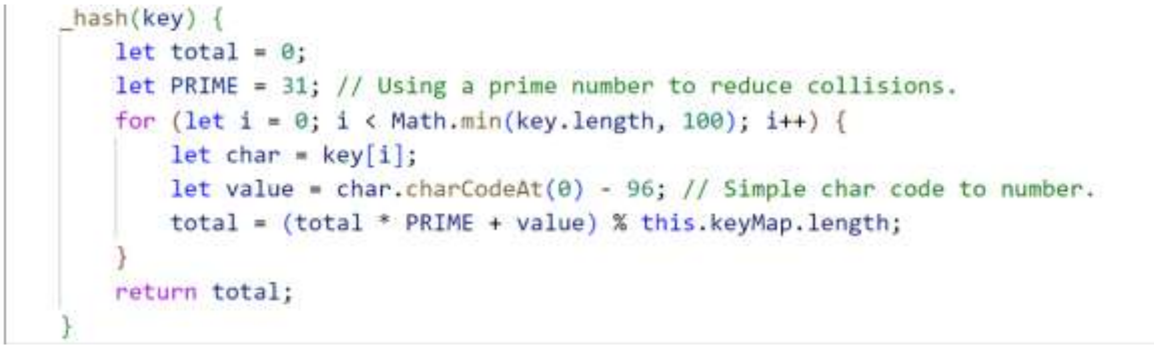
A hash table uses a hash function to map keys to values. In JavaScript, a native object can be used as a simple hash table, but here's how you can implement a basic custom hash table.



```
class HashTable {
    constructor(size = 53) {
        this.keyMap = new Array(size);
    }
```

- ✓ **Hash Table**: This defines a new class for creating a hash table.
- ✓ **Constructor (size = 53)**: The constructor initializes an array keymap with a default size of 53 (this will hold the hashed keys and their values).
- ✓ **New Array(size)**: This creates an empty array of length size to store the key-value pairs

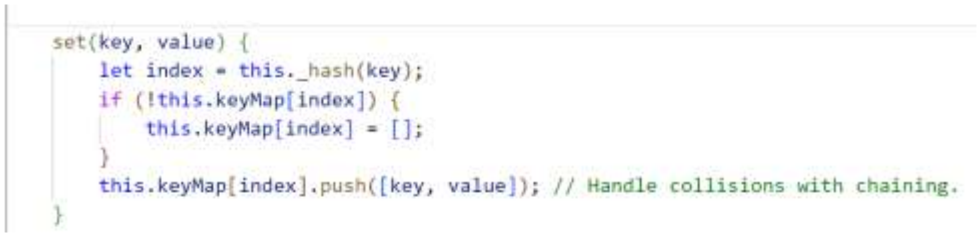
1. **Hash Function**



```
_hash(key) {
    let total = 0;
    let PRIME = 31; // Using a prime number to reduce collisions.
    for (let i = 0; i < Math.min(key.length, 100); i++) {
        let char = key[i];
        let value = char.charCodeAt(0) - 96; // Simple char code to number.
        total = (total * PRIME + value) % this.keyMap.length;
    }
    return total;
}
```

- ✓ **_hash(key)**: This is a private method (indicated by _ prefix) that generates an index in the array for a given key.
- ✓ **let total = 0**: Starts a total that will be the hash result.
- ✓ **PRIME = 31**: Using a prime number (31) to reduce collisions (this is a common technique in hash functions).
- ✓ **char.charCodeAt(0) - 96**: Converts each character to a number by taking its ASCII code minus 96, so 'a' becomes 1, 'b' becomes 2, etc.
- ✓ **total = (total * PRIME + value) % this.keyMap.length**: This combines the current total with the character's value, multiplied by the prime number and then uses the modulus operator to ensure the result is within the bounds of the array.

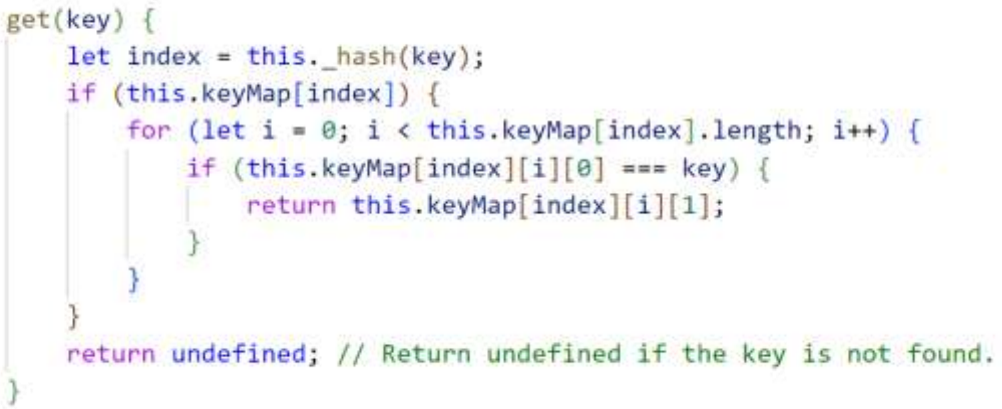
1. Set Method



```
set(key, value) {
    let index = this._hash(key);
    if (!this.keyMap[index]) {
        this.keyMap[index] = [];
    }
    this.keyMap[index].push([key, value]); // Handle collisions with chaining.
}
```

- ✓ **set(key, value)**: This method stores a key-value pair in the hash table.
- ✓ **this._hash(key)**: First, it calculates the index for the key using the _hash function.
- ✓ **if (!this.keyMap[index])**: Checks if the bucket at the calculated index is empty. If so, it initializes an empty array (to handle collisions via chaining).
- ✓ **this.keyMap[index].push([key, value])**: Pushes the key-value pair into the array at the hashed index.

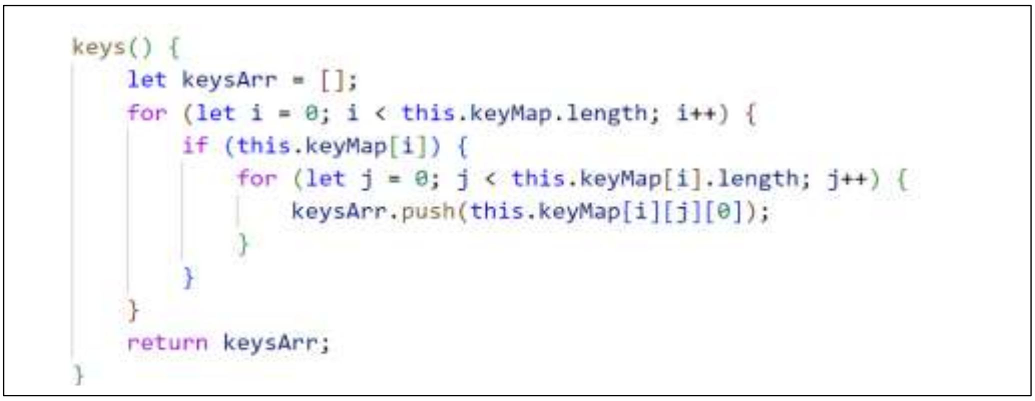
2. Get Method



```
get(key) {
    let index = this._hash(key);
    if (this.keyMap[index]) {
        for (let i = 0; i < this.keyMap[index].length; i++) {
            if (this.keyMap[index][i][0] === key) {
                return this.keyMap[index][i][1];
            }
        }
    }
    return undefined; // Return undefined if the key is not found.
}
```

- ✓ **get(key)**: Retrieves the value associated with a key.
- ✓ **let index = this._hash(key)**: Hashes the key to find its index.
- ✓ **if (this.keyMap[index])**: If there is something at that index, it loops through the key-value pairs to find the matching key.
- ✓ **return this.keyMap[index][i][1]**: Returns the value associated with the key if found.
- ✓ **return undefined**: If the key is not found, it returns undefined.

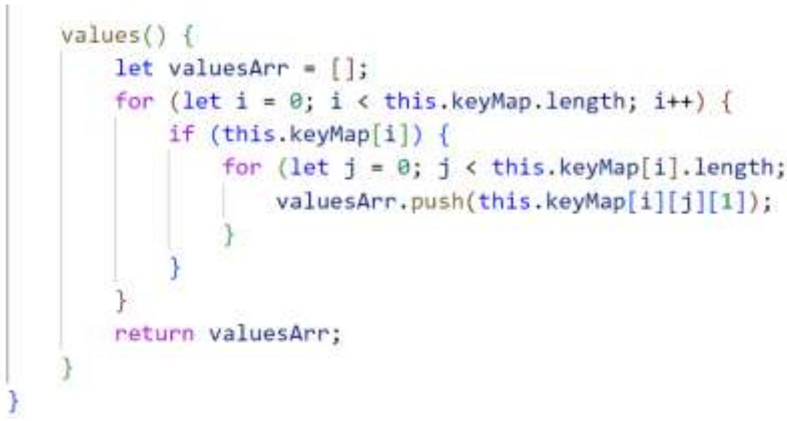
3. Retrieve Keys



```
keys() {
    let keysArr = [];
    for (let i = 0; i < this.keyMap.length; i++) {
        if (this.keyMap[i]) {
            for (let j = 0; j < this.keyMap[i].length; j++) {
                keysArr.push(this.keyMap[i][j][0]);
            }
        }
    }
    return keysArr;
}
```

- ✓ **keys()**: Returns all the keys stored in the hash table.
- ✓ **for (let i = 0; i < this.keyMap.length; i++)**: Loops through all the buckets (arrays).
- ✓ **keysArr.push(this.keyMap[i][j][0])**: Adds each key from the key-value pair to keysArr.
- ✓ **return keysArr**: Returns the array of keys

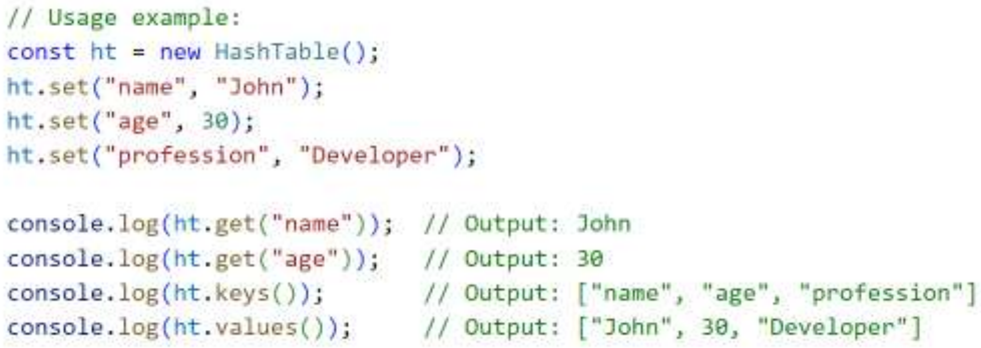
4. Retrieve Keys



```
    values() {
        let valuesArr = [];
        for (let i = 0; i < this.keyMap.length; i++) {
            if (this.keyMap[i]) {
                for (let j = 0; j < this.keyMap[i].length; j++) {
                    valuesArr.push(this.keyMap[i][j][1]);
                }
            }
        }
        return valuesArr;
    }
}
```

- ✓ **values()**: Returns all the values stored in the hash table.
- ✓ Similar to the keys() method, but it retrieves and returns values from the key-value pairs ([j][1])

5. Display method



```
// Usage example:
const ht = new HashTable();
ht.set("name", "John");
ht.set("age", 30);
ht.set("profession", "Developer");

console.log(ht.get("name"));  // Output: John
console.log(ht.get("age"));   // Output: 30
console.log(ht.keys());       // Output: ["name", "age", "profession"]
console.log(ht.values());     // Output: ["John", 30, "Developer"]
```

Practical Activity

Suppose that client asked you to use a hash table to store and retrieve user information like name, age, and profession. Each user's name will be the key and store additional user data like age and profession. By using JavaScript write code to:

a. Add user data (name, age, profession).

b. Retrieve data by name.

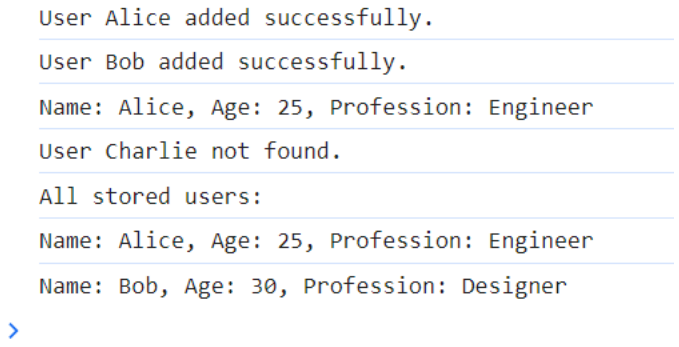
c. Display all stored users

Solution

```
// Hash table (object) to store user data
let users = {};
// Function to add user data
function addUser(name, age, profession) {
    users[name] = {
        age: age,
        profession: profession
    };
    console.log(`User ${name} added successfully.`);
}

// Function to retrieve user data by name
function getUser(name) {
    if (users[name]) {
        console.log(`Name: ${name}, Age: ${users[name].age}, Profession: 
${users[name].profession}`);
        return users[name];
    } else {
        console.log(`User ${name} not found.`);
        return null;
    }
}

// Function to display all stored users
```

```
function displayAllUsers() {
    console.log("All stored users:");
    for (let name in users) {
       console.log(`Name: ${name}, Age: ${users[name].age}, Profession: ${users[name].profession}`);
    }
}

// Example usage
addUser("Alice", 30, "Engineer");
addUser("Bob", 30, "Designer");

getUser("Alice"); // Retrieve Alice's data
getUser("Charlie"); // Try to retrieve non-existent user

displayAllUsers(); // Display all users
```

OUTPUT



```
User Alice added successfully.
User Bob added successfully.
Name: Alice, Age: 25, Profession: Engineer
User Charlie not found.
All stored users:
Name: Alice, Age: 25, Profession: Engineer
Name: Bob, Age: 30, Profession: Designer
>
```

Practical Activity 3.1.9: Performing sort operations

Task:

1: Read the task described below:

A store manager wants to analyse the sales data of different products in their shop. The data is stored as an array of objects, where each object contains the name of a product and the total sales (in USD) for that product.

Write a JavaScript function that takes this array as input and performs the following tasks:

a) Sort the products in descending order of sales (highest sales first).

b) Sort the products alphabetically by name if two or more products have the same sales. Return the sorted array.

2: Follow the demonstration of trainer

3: Individually perform the task 3.1.9 referring to the demonstration

4: Ask for assistance where it is needed.

5: Read the key readings 3.1.9 in trainee's Manual

Key readings 3.1.9.: Performing sorting operations

- **Bubble**

We have an unsorted array arr = [1, 4, 2, 5, -2, 3], and the task is to sort the array using bubble sort in ascending order.

Bubble sort compares the element from index 0 and if the 0th index value is greater than 1st index value, then the values get swapped and if the 0th index value is less than the 1st index value, then nothing happens.

Next, the 1st index value compares to the 2nd index value, and then the 2nd index value compares to the 3rd index value, and so on…

Syntax

```
BubbleSort(array) {    for i -> 0 to arrayLength         for j -> 0 to (arrayLength - i - 1)          if arr[j]
    > arr[j + 1]               swap(arr[j], arr[j + 1])
}
```

```
// Bubble sort Implementation using Javascript
// Creating the bbleSort function
 function bblSort(arr) {
  for (var i = 0; i < arr.length; i++) {

       // Last i elements are already in place
     for (var j = 0; j < (arr.length - i - 1); j++) {           // Checking if the item at present iteration
     // is greater than the next iteration
    if (arr[j] > arr[j + 1]) {
          // If the condition is true
     // then swap them
       var temp = arr[j]               arr[j] = arr[j + 1]               arr[j + 1] = temp
          }
        }
    }
   // Print the sorted array    console.log(arr);
}
// This is our unsorted array var arr = [234, 43, 55, 63, 5, 6, 235, 547]; // Now pass this array to
     the bblSort() function bblSort(arr);
```

Output: Sorted array

```
[5, 6, 43, 55, 63, 234, 235, 547]
```

Note: This implementation is not optimized. We will see the optimized solution next.

Optimized Solution

As we discussed the implementation of bubble sort earlier that is not optimized. Even if the array is sorted, the code will run with O(n^2) complexity. Let's see how to implement an optimized bubble sort algorithm in javascript.

The syntax for Optimized solution BubbleSort(array){   for i -> 0 to arrayLength   isSwapped <- false   for j -> 0 to (arrayLength - i - 1)   if arr[j] > arr[j + 1]   swap(arr[j], arr[j + 1])   isSwapped -> true
}

Implementation function

```
bubbleSort(array) {    const arrayLength = array.length;    let isSwapped;
   for (let i = 0; i < arrayLength; i++) {        isSwapped = false;
      for (let j = 0; j < arrayLength - i - 1; j++) {           if (array[j] > array[j + 1]) {
            // Swap elements
            [array[j], array[j + 1]] = [array[j + 1], array[j]];              isSwapped = true;
         }
      }
      // If no two elements were swapped in the inner loop, array is sorted        if (!isSwapped)
     break;
   }    return array;
}

// Test the function const sortedArray = bubbleSort([45, 23, 3, 5346, 5, 356, 243, 35]);
     console.log("Sorted Array:"); console.log(sortedArray);
```

Output: Sorted Array

```
[3, 5, 23, 35, 45, 243, 356, 5346]
```

- **Quick**

What is Quick Sort Algorithm?

Quick sort is one of the sorting algorithms that works on the idea of divide and conquer. It takes an element as a pivot and partitions the given array around that pivot by placing it in the correct position in the sorted array. The pivot element can be selected in the following ways:

- Select the First element as a pivot
- Select the Last element as a pivot
- Select the Middle element as a pivot
- Select a Random element as a pivot

We will use the Last element as a pivot for this article.

Programmatic Approach for Quick Sort

- First, create a recursive function that takes array, low value and high value as input and calls the partition function for recursive call for partitioned arrays.
- Define a partition function for last element as pivot and give the partition index of array.
- Partition function iterate the given array and compare elements to pivot. If smaller then swap them to a sequential position else no swap is performed.
- At the end for iteration the pivot element is swapped to its correct position stored in i for the exact place.
- Now return the latest position of the pivot element

Example: Here is the complete code for the Quick Sort algorithm using JavaScript.

```
function partition(arr, low, high) {      let pivot = arr[high];      let i = low - 1;      for (let j = low; j <= high - 1; j++) {
    // If current element is smaller than the pivot        if (arr[j] < pivot) {
      // Increment index of smaller element            i++;
      // Swap elements
      [arr[i], arr[j]] = [arr[j], arr[i]];
    }
  }
  // Swap pivot to its correct position
  [arr[i + 1], arr[high]] = [arr[high], arr[i + 1]];      return i + 1; // Return the partition index
}
function quickSort(arr, low, high) {      if (low >= high) return;      let pi = partition(arr, low, high);      quickSort(arr, low, pi - 1);      quickSort(arr, pi + 1, high);
}
let arr = [10, 80, 30, 90, 40];  console.log("Original array: " + arr);  quickSort(arr, 0, arr.length - 1);  console.log("Sorted array: " + arr);
```

Output

```
Original array: 10,80,30,90,40 Sorted array: 10,30,40,80,90
```

Practical Activity 3.1.10: Performing search operations

Task:

1: Read the task described below:

A store manager wants to analyse the sales data of different products in their shop. The data is stored as an array of objects, where each object contains the name of a product and the total sales (in USD) for that product.

Write a JavaScript function that takes this array as input and performs the following tasks:

a) Write a function to search for a product by its name using binary search for faster lookup.

b) If the product is found, return its price. If the product is not found, return "Product not available.

2: Follow the demonstration of trainer

3: Individually perform the task 3.1.10 referring to the demonstration

4: Ask for assistance where it is needed.

5: Read the key readings 3.1.10 in trainee's Manual

Key readings 3.1.10.: Performing search operations

1. Linear search

```
// Product data
const products = [
 { name: "Laptop", price: 1200 },
 { name: "Mouse", price: 25 },
 { name: "Keyboard", price: 50 },
 { name: "Monitor", price: 300 },
 { name: "Headphones", price: 100 },
];

// Linear Search Function
function linearSearch(products, targetName) {
 for (let i = 0; i < products.length; i++) {
  if (products[i].name === targetName) {
   return products[i].price; // Product found
  }
 }
 return "Product not available"; // Product not found
}
```

```
// Example Usage
console.log("Price of Laptop:", linearSearch(products, "Laptop")); // Output: 1200
console.log("Price of Monitor:", linearSearch(products, "Monitor")); // Output: 300
console.log("Price of Tablet:", linearSearch(products, "Tablet")); // Output: Product not available
```

- **const products**:

  This creates a **constant array** called products that stores objects.

  Each object has two properties:

  Name: The name of the product (a string).

  Price: The price of the product (a number).

```
const products = [
  { name: "Laptop", price: 1200 },
  { name: "Mouse", price: 25 },
  { name: "Keyboard", price: 50 },
  { name: "Monitor", price: 300 },
  { name: "Headphones", price: 100 },
];
for (let i = 0; i < products.length; i++) {
```

- **for loop:**
  - Iterates through the products array.
  - let i = 0: Starts the loop with i as 0 (first index).
  - i < products.length: Ensures the loop runs until the last product in the array.
  - i++: Increases the index i by 1 after each iteration

  Purpose: To search for a specific product in the array

- **if statement:**

```
if (products[i].name === targetName) {
```

- Checks if the name of the product at index i matches the targetName.
- products [i]: Accesses the product object at index i in the array.
- .name: Gets the name property of the current product.
- ===: Ensures a strict comparison (type and value must match).

Purpose: To determine if the current product is the one being searched for.

- **return statement:**

```
return products[i].price;
```

- If the if condition is true, this returns the price of the found product.
- products[i].price: Accesses the price property of the matched product.

Purpose: To immediately stop the function and output the price when the product is found.

}

❖ **End of if block:**

Closes the condition that checks if the product is found.

return "Product not available";

return statement:

- If the loop completes without finding the product, this statement is executed.
- Returns a message saying the product is not in the array.

Purpose: To handle cases where the product does not exist in the list.

```
console.log("Price of Laptop:", linearSearch(products, "Laptop")); // Output: 1200
console.log("Price of Monitor:", linearSearch(products, "Monitor")); // Output: 300
console.log("Price of Tablet:", linearSearch(products, "Tablet")); // Output: Product
    not available
```

console.log:

Prints the result of calling linearSearch for different product names to the console.

Examples:

linearSearch(products, "Laptop"): Searches for "Laptop" and prints its price (1200).

linearSearch(products, "Monitor"): Searches for "Monitor" and prints its price (300).

linearSearch(products, "Tablet"): Searches for "Tablet" (not in the list) and prints "Product not available".

Purpose: To test the function with different inputs.

Overall Explanation

- The linear search function goes through each product in the array until it finds the one that matches the name or finishes checking all products.
- **Output:**

  If the product is found, its price is returned.

  If not found, a message ("Product not available") is returned

2. Binary Search

```
// Pre-sorted product data
const products = [
  { name: "Headphones", price: 100 },
  { name: "Keyboard", price: 50 },
  { name: "Laptop", price: 1200 },
  { name: "Monitor", price: 300 },
  { name: "Mouse", price: 25 },
];

// Binary Search Function
function binarySearch(products, targetName) {
  let left = 0; // Start pointer
  let right = products.length - 1; // End pointer

  while (left <= right) {
    const mid = Math.floor((left + right) / 2); // Middle index

    if (products[mid].name === targetName) {
      return products[mid].price; // Product found, return its price
    }

    if (products[mid].name < targetName) {
      left = mid + 1; // Target is in the right half
    } else {
      right = mid - 1; // Target is in the left half
    }
  }

  return "Product not available"; // Target not found
}

// Example Usage
console.log("Price of Laptop:", binarySearch(products, "Laptop")); // Output: 1200
console.log("Price of Monitor:", binarySearch(products, "Monitor")); // Output: 300
console.log("Price of Tablet:", binarySearch(products, "Tablet")); // Output: Product not available
```

Program explanation

Input Data:

- The products array is already **sorted** by name in ascending order.
- Each object has two properties: name and price.

Function:

- **left**: The starting index of the search range.
- **right**: The ending index of the search range.
- **mid**: The middle index of the current range (Math.floor((left + right) / 2)).

Algorithm Steps:

- **Step 1**: Check if the middle product's name matches the target (products[mid].name === targetName).
- **Step 2**: If the target name is alphabetically **after** the middle (products[mid].name < targetName), move the left pointer to mid + 1.
- **Step 3**: If the target name is alphabetically **before** the middle (products[mid].name > targetName), move the right pointer to mid - 1.
- **Step 4**: Repeat until the left pointer exceeds the right pointer or the product is found.

Return Value:

- Returns the price if the product is found.
- Returns "Product not available" if the product is not in the list

Application of learning 3.1.

Hasha Ltd is facing challenges in managing its warehouse data due to the use of hard copies for record-keeping. As a solution, you have been tasked with creating an efficient data management system that handles imports, exports, and transit processes for furniture. Use the following data structures: **Stack, Sorting, and Linked List** to develop the system.

For each question below, **Write JavaScript code** that addresses the respective problem:

1. The company receives shipments daily, and the manager needs to process them in the order they arrive. Sometimes the manager needs to undo the last action (removal of incorrect shipments). Write JavaScript code using a **Stack** to manage this process. The stack should allow the manager to undo the last operation efficiently. Define the methods needed to push, pop, and check the top of the stack.

2. The warehouse manager needs to sort furniture items by categories such as Tables, Chairs, Beds, and Dressers to produce a daily report. Write JavaScript code using a Sorting Algorithm to organize these categories.

3. The number of shipments in the warehouse can vary daily, so a flexible data structure is needed. Write JavaScript code using a Linked List to represent the furniture items in the warehouse, allowing dynamic addition and removal of items.

Indicative content 3.2: Run JavaScript Source Codes

Duration: 20 hrs

Theoretical Activity 3.2.1: Description of browser embedded Tools and IDE Terminal

Tasks:

1: Answer the following questions:

i. Discuss browser-embedded tools, particularly focusing on the rendering engine and web development tools.

ii. Define an IDE and List top 5 JavaScript IDEs

2: Present their findings in front of Class

3: Listen attentively, take notes and ask questions where is needed

4: Read key readings 3.2.1 from trainee's manual

Key readings 3.2.1.: Description of browser embedded Tools and IDE Terminal

Run JavaScript source codes

•Using browser-embedded Tools

Browser-embedded tools are features or functions integrated directly into a web browser, providing additional capabilities beyond the basic browsing experience.

✓ **Rendering engine**

A rendering engine is a software program that interprets and converts the HTML, CSS, and JavaScript code of a web page into visuals that are displayed on the screen. It is the core component of a web browser and plays a crucial role in the overall performance and compatibility of the browser.

Benefits of Rendering Engines

The benefits of Rendering engines are:

- **Speedy Browsing:** Rendering engines are like superchargers for websites, making them load faster and smoother. The quicker the engine, the zippier your browsing experience!
- **Consistent Look:** Imagine a website that stretches and squishes on different devices. Not cool! Rendering engines ensure websites appear exactly as intended, no matter if you're on a phone, laptop, or giant screen.

- **Works Everywhere:** Don't worry about Windows, Mac, or Linux – rendering engines speak the language of all devices, making sure websites work flawlessly regardless of your setup.
- **Modern Goodies:** Think of fancy animations, 3D graphics, and cool new features. Rendering engines are always learning, constantly adding support for the latest and greatest web technologies.
- **Your Browser, Your Way:** Some engines love customization! They let you tweak your browser to your liking, adding extensions, and themes, and even building entirely new things. It's like having a toolbox for your browser! **Tasks performed by Rendering Engines in Browsers** The various task performed by Rendering Engines are:
- **Creating DOM tree:** The rendering engine's parse the HTML code of the web page.

This involves breaking down the HTML elements and their attributes into a the Document Object Model (DOM) tree. The DOM tree is a hierarchical representation of the web page's content.

- **Appling styles:** The rendering engine parses the CSS styles of the web page. CSS rules such as font styles, colors, margins, and positioning define how HTML elements should be visually presented, . The rendering engine applies these styles to the DOM tree, transforming the structural representation into a styled representation.
- **Layout Construction:** Once the DOM tree is styled, the rendering engine enters the layout phase. It calculates the exact position and dimensions of each element on the screen. Some complex layout algorithms are used for calculating the relationships between elements such that the web page's content is displayed as intended by the developer.
- **DOM to pixels:** The rendering engine starts the rasterization stage where it converts the vector-based DOM tree into a grid of pixels. Each pixel is assigned a color with respect to the element. This results in a rasterized image representation of the web page.
- **Pixels to Display:** The compositing is then done where the rasterized image is layered on top of other visual elements such as the background and browser controls. Compositing make sure that the visual elements are correctly blended and displayed on the screen.
- **GPU for Rendering**: To enhance performance, modern rendering engines uses Graphics Processing Unit (GPU). The GPU is a specialized hardware component designed to handle graphics-intensive tasks efficiently. By allowing rasterization and compositing process to the GPU, rendering

engines can significant improve its performance. This feature allows smoother and more responsive web browsing experiences.

- **Optimizing:** Rendering engines are constantly evolving, undergoing continuous optimization to enhance performance, compatibility, and maintain web standards. Developers improve the efficiency of algorithms, reduce memory usage, and ensure compatibility with the latest web technologies.

✓ **Rendering Engines of Web Browsers**

There are three primary rendering engines that power the majority of web browsers today:

- **Blink:** Blink is an open-source rendering engine developed by Google and is the foundation for the Chrome browser. It is also used by several other browsers, including Opera, Brave, and Vivaldi.
- **Gecko:** Gecko is another open-source rendering engine developed by Mozilla and is the engine that powers the Firefox browser. It is known for its adherence to web standards and strict compatibility with various web technologies.
- **WebKit:** WebKit is an open-source rendering engine initially developed by Apple for its Safari browser. It is also used by the iOS version of the Chrome browser and serves as the basis for the Qt WebEngine framework.
- **EdgeHTML:** EdgeHTML, a fork of Trident (Internet Explorer's engine), is Microsoft's

Rising star. It's known for its focus on interoperability and compatibility with older web technologies while embracing modern advancements. Think of it as a bridge builder, connecting the past and present of the web, ensuring smooth transitions for users and developers alike.

Other Engines: Beyond these giants, a constellation of niche engines exists, catering to specific needs and platforms. WebKit variations like Presto (Opera Mini) and Goanna (Vivaldi) offer unique mobile experiences. WebRender (Android WebView) prioritizes efficiency on low-powered devices. Each engine, like a specialist chef, brings its own flavor and expertise to the table, enriching the web's diversity and adaptability

Browser	Rendering Engine
Google Chrome	Blink
Mozilla Firefox	Gecko
Apple Safari	WebKit

Microsoft Edge	Blink
Opera	Blink
Brave	Blink
Vivaldi	Goanna(Fork of WebKit)
Internet Explorer 11	Trident
Android WebView	WebRender
Samsung Internet	WebKit

•Web dev tools

Developer tools (or "development tools" or short "DevTools") are programs that allow a developer to create, test and debug software.

Current browsers provide integrated developer tools, which allow to inspect a website. They let users inspect and debug the page's HTML, CSS, and JavaScript, allow to inspect the network traffic it causes, make it possible to measure its performance, and much more.

✓ Sublime Text
✓ Vim
✓ Notepad++
✓ Atom
✓ Django
✓ Vue
✓ Ruby on Rails
✓ Foundation
✓ Ember
✓ REST Assured
✓ HoppScotch
✓ Svelte
✓ Meteor
✓ Figma
✓ ProtoPie

Top JavaScript IDEs

1. Atom
2. Tabnine
3. GoormIDE
4. Sublime Text
5. PLAYCODE. IO

6. JSFiddle
7. CodePen
8. Observable
9. Aptana
10. Amazon Web Services9,

Points to Remember

- **Browser embedded tools** are features or functions integrated directly into a web browser, providing additional capabilities beyond the basic browsing experience.
- The main component of browser embedded Tools include Rendering engine and Web dev tools

Application of learning 3.2.

As software developer install a browser and JavaScript IDE terminal into your computer and write JavaScript codes that display Hello Word, using browser embedded Tools.

Indicative content 3.3: Test Time and Space Complexity

Duration: 15 hrs

Theoretical Activity 3.3.1: Description of Key concepts of measuring time and space complexity

Tasks:

1: Answer the following questions:

i. Differentiate algorithm time complexity from space complexity?

ii. Describe the following algorithm measurement notation?

a) O(n log n)

b) O(n)

iii. Describe time and space measurement tools?

2: Present their findings in front of Class

3: Ask clarifying question if any.

5: Read key readings 3.3.1 in trainee's Manual

Key readings 3.3.1.: Description of Key concepts of measuring time and space complexity

- **Key Concepts of Measuring Time and Space Complexity in Algorithm Evaluation**

1. Time complexity vs Space complexity

In the context of evaluating algorithms, **time complexity** refers to how the execution time of an algorithm scales with the size of the input, while **space complexity** measures the amount of memory the algorithm uses.

When evaluating algorithms, **time and space complexity** are two important aspects used to assess performance. These complexities help determine how scalable and efficient an algorithm is as input sizes grow.

Below are the key concepts involved:

1.1 Time Complexity

Time complexity refers to how the execution time of an algorithm changes as the size of the input grows. It provides a high-level understanding of the algorithm's efficiency by focusing on how quickly it runs.

Here are major terms used:

- **Big-O Notation**: This is the standard way to express time complexity. It represents the worst-case scenario of an algorithm’s growth rate relative to the input size (n). Common Big-O time complexities include:
  - ✓ **O(1)**: Constant time – the execution time does not change regardless of input size.
  - ✓ **O(log n)**: Logarithmic time – the time increases logarithmically as input size increases.
  - ✓ **O(n)**: Linear time – execution time grows proportionally to input size.
  - ✓ **O(n²)**: Quadratic time – time grows with the square of the input size.
  - ✓ **O(2^n)**: Exponential time – the time doubles with each additional input element.
- **Best-case, Worst-case, and Average-case**:
  - ✓ **Worst-case**: The maximum time the algorithm takes for any input.
  - ✓ **Best-case**: The minimum time, often when the input is already optimized (e.g., sorted).
  - ✓ **Average-case**: The expected time, considering all possible inputs and probabilities.

1.2. Space Complexity

Space complexity measures the amount of memory or storage an algorithm uses as the input size grows. Like time complexity, it is expressed in Big-O notation.

- **Auxiliary Space**: Refers to the extra space or temporary storage used by the algorithm, excluding the input data itself.
- **In-place algorithms**: These algorithms use a minimal amount of extra memory, typically O(1), as they modify the input data directly.

1.3 Trade-offs Between Time and Space

In many cases, optimizing for one complexity may worsen the other:

- **Time-space trade-off**: Some algorithms consume more memory to reduce time, while others take longer but use less memory. For example:
  - ✓ **Merge Sort** (O(n log n) time, O(n) space) uses extra space for sorting.
  - ✓ **Quick Sort** (O(n log n) time, O(log n) space) uses less space but may be slower on certain inputs.

2. Time and Space Measurement Tools

Measuring these complexities is important for optimizing and ensuring the efficiency of code. Various tools are available to aid in profiling and benchmarking code execution.

Below are the key tools for measuring time and space complexities in JavaScript: **Profiling Tools**, **Benchmark.js**, **Benchmarkify** and **jsPerf**

1. **Profiling Tools**:

Profiling tools allow developers to collect detailed data about the runtime performance of an application. They provide insight into function call durations, memory usage, and possible performance bottlenecks.

- ✓ **Chrome DevTools**: One of the most commonly used tools for JavaScript profiling. It helps visualize call stacks, memory allocation, and heap snapshots.
- ✓ **Node.js Profiler**: Offers built-in profiling features to measure both time and space complexity in Node.js environments.

2. **Benchmark.js**:

Benchmark.js is a powerful JavaScript benchmarking library designed to measure the execution time of code with high precision. It can handle asynchronous benchmarks and automatically account for fluctuations in the system load, giving accurate results.

- ✓ Supports comparative benchmarking of multiple code snippets.
- ✓ Can be run both in browsers and Node.js environments.
- ✓ Built-in functions for managing test cases, suites, and statistical analysis.

3. **Benchmarkify**:

Benchmarkify is a minimalistic benchmarking framework designed for Node.js. It simplifies the process of comparing performance between multiple functions or implementations.

- ✓ Provides an easy API for creating and running benchmarks.
- ✓ Capable of comparing different algorithms or code implementations.
- ✓ Supports asynchronous benchmarks, which makes it a good fit for performance testing in complex environments.

4. **jsPerf**:

jsPerf is an online platform that allows developers to create and share JavaScript performance tests. It lets users test the performance of different approaches to solve a problem by comparing execution speeds.

- ✓ Allows for easy performance comparison of different code snippets in a collaborative manner.
- ✓ Provides access to real-world browser environments for accurate measurements.
- ✓ Suitable for testing browser-specific behaviors, which is especially useful for front-end developers.

Practical Activity 3.3.2: Testing the program performance.

Task:

1: Read the task described below:

Develop a simple web page and Test its Time and space complexity, and finally document test findings.

2: Follow the demonstration of trainer

3: Individually perform the task 3.3.2 referring to the demonstration

4: Ask for assistance where it is needed.

5: Read the key readings 3.3.2 in trainee’s Manual

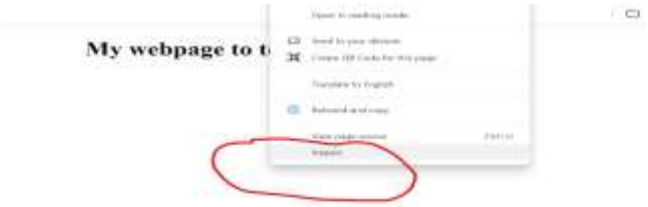
Key readings 3.3.2: Testing the program performance

Time and space measurement tools

There are some libraries and 3rd party tools that could aid you in measuring JavaScript execution time and memory usage.

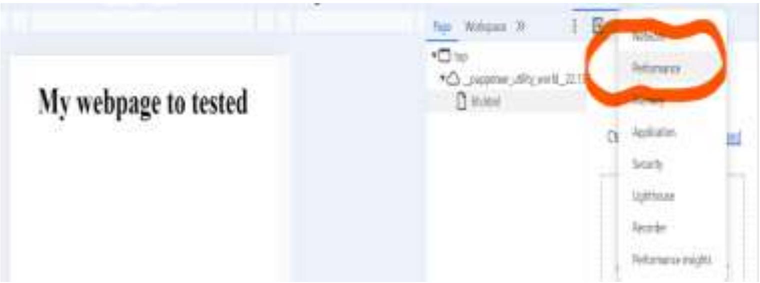
1. Google Chrome DevTools

Step 1: Open DevTools



Access DevTools by right-clicking anywhere on your webpage and selecting "Inspect".
Alternatively, you can press F12 on Windows or Cmd + Opt + I on Mac.

Step 2: Use the Performance Panel



Navigate to the "Performance" panel. You can capture a snapshot of all the actions on the webpage for a certain period by clicking the record button at the top left of the panel. While recording, interact with your web page in a way that triggers the JavaScript operation you're interested in, then stop recording.

After that, DevTools will provide a comprehensive breakdown of everything that happened, including JavaScript execution, painting, layout, system, and loading. Look for long bars in the “Main” row for possible inefficiencies. Additionally, you can click on these bars to get more detailed information.

Step 3: Use the Profiler Panel

Another tool useful for checking JavaScript execution time is the "JavaScript Profiler" panel. In this panel, you can start a new CPU profiling by clicking the Start button. After collecting the profile data, stop recording.

2.Document Test Findings

When documenting your findings, include the following:

- **Purpose**: State the purpose of the benchmark (e.g., comparing data structure performance).
- **Test Setup**: Describe the environment, input sizes, and specific operations tested.
- **Results**: Include the measured time and memory usage. Use tables or graphs for clarity.
- **Analysis**: Interpret the results. Discuss which data structure performed better and why.
- **Conclusions**: Summarize the findings and any recommendations for future implementations.

Example Documentation Structure

Title: Performance Testing of JavaScript Data Structures

Purpose: To compare the performance of arrays vs. linked lists for insertion operations.

Test Setup:

- Environment: Node.js v14
- Input Size: 1,000 elements
- Operations: Insert 1,000 elements into an array and a linked list.

Results

Data Structure	Time (ms)	Memory Usage (MB)
Array	5	2
Linked List	12	3

Analysis: The array performed faster for insertions due to contiguous memory allocation, while the linked list used more memory because of node pointers

Points to Remember

- Steps to measure program performance using Google Chrome DevTools:

1. Open DevTools
2. Use the Performance Panel
3. Use the Profiler Panel

Application of learning 3.3.

DF TSS is a TVET school that has a new school management system. Before its use, the school need software engineer to test for them the developed system and document test findings.

Learning outcome 3 end assessment

Written assessment

I. **Read the following statement related to data structure and algorithm fundamentals, and answer True if the statement is correct and False if the statement is wrong:**

1. The bubble sort algorithm is generally faster than quick sort for large datasets
2. Arrays and linked lists are both linear data structures
3. Binary search can be performed on both sorted and unsorted arrays
4. Benchmark.js and jsPerf are tools used for testing and measuring the performance of JavaScript code.
5. Profiling and benchmarking tools like Benchmark.js and jsPerf help evaluate the performance of JavaScript code.

II. **Read the Read the following statement related to data structure and algorithm fundamentals and choose the letter corresponding to the correct answer.**

1. Which of the following environments can be used to run JavaScript source code?

   a) Browser Developer Tools
   b) IDE Terminal
   c) Text Editor
   d) Both a and b

2. Which of the following is a linear data structure?

   a) Graph
   b) Tree
   c) Linked List
   d) Hash Table

3. Which sorting algorithm generally has better average performance on large datasets?

   a) Bubble Sort
   b) Quick Sort
   c) Insertion Sort
   d) Selection Sort

4. When can binary search be used?

   a) On unsorted data
   b) On sorted data
   c) On linked lists only
   d) On any data structure

5. Which of the following tools is commonly used to profile and measure JavaScript performance?
   a) Chrome DevTools
   b) jsPerf
   c) Benchmark.js
   d) All of the above

Practical assessment

Hasha Ltd is facing challenges in managing its warehouse data due to the use of hard copies for record-keeping. As a solution, you have been tasked with creating an efficient data management system that handles imports, exports, and transit processes for furniture. Use the following data structures: **Stack, Sorting, and Linked List** to develop the system.

For each question below, **Write JavaScript code** that addresses the respective problem:

I. The company receives shipments daily, and the manager needs to process them in the order they arrive. Sometimes the manager needs to undo the last action (removal of incorrect shipments). Write JavaScript code using a **Stack** to manage this process. The stack should allow the manager to undo the last operation efficiently. Define the methods needed to push, pop, and check the top of the stack.

II. The warehouse manager needs to sort furniture items by categories such as Tables, Chairs, Beds, and Dressers to produce a daily report. Write JavaScript code using a Sorting Algorithm to organize these categories.

III. The number of shipments in the warehouse can vary daily, so a flexible data structure is needed. Write JavaScript code using a Linked List to represent the furniture items in the warehouse, allowing dynamic addition and removal of items.

END

References

BOOKS

javatpoint. (2024). */javascript-queue*. Retrieved from javatpoint: https://www.javatpoint.com/javascript-queue

Karim, M. R. (2019). *JavaScript Algorithms and Data Structures.*

tutorialspoint. (n.d.). */Creating-a-Stack-in-Javascript* . Retrieved from tutorialspoint: https://www.tutorialspoint.com/Creating-a-Stack-in-Javascript

WEB LINKS

educative. (2024). */blog/data-strucutres-hash-table-javascript* . Retrieved from educative: https://www.educative.io/blog/data-strucutres-hash-table-javascript

geeksforgeeks. (2024). */rendering-engines-used-by-different-web-browsers/* . Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/rendering-engines-used-by-different-web-browsers/

tutorialspoint. (2024). */javascript/javascript_arrays_object.htm*. Retrieved from tutorialspoint: https://www.tutorialspoint.com/javascript/javascript_arrays_object.htm

RTB | RWANDA TVET BOARD

October 2024