VITMEE Resources

Computer Syllabus -

1)Date Structures: Arrays, Stacks, Queues, linked Lists. Sorting techniques, Searching Techniques, Trees and Graph terminology and representation in memory, binary search tree, traversal techniques of graphs and Trees.

2)Computer Networks: Network models, Internet model, OSI model, Physical Layer - Analog and Digital Signals, Analog and Digital Transmission, Coding, Sampling. Data Link Layer - Error detection and correction, Data link control and Protocols, Stop and wait, Go-back-n, Selective repeat. Network Layer - Inter-networks, Addressing, unicast and multicast routing, Presentation Layer.

3)Programming in C: Data types, Declarations, Expressions, statements and symbolic constants, input-Output functions. Operators and expressions: Arithmetic, unary, logical, bit-wise, assignment and conditional operators. Control statements: While, do-while, for statements, nested loops, if else, switch, break, Continue, comma operators. Storage types: Automatic, external, register and static variables. Functions: Defining and accessing, passing arguments, Recursion.

4)Database Management Systems: DBMS architecture, Data models, data independence, E-R model, normalization, Relational Model: concepts, constraints, languages. Data storage, indexing, query processing, design and programming SQL.

5)Operating Systems: Process management, Process States, Process Control Block, Process and Threads, CPU Scheduling, Scheduling algorithm, Process Synchronization and Deadlock, Memory management, Virtual memory concepts paging and segmentation File organization, Blocking and buffering, file descriptor, File and Directory structures, I/O Devices.

6)Computer Architecture: Boolean algebra and computer arithmetic, flip-flops, design of combinational and sequential circuits, instruction formats, addressing modes, interfacing peripheral devices, types of memory and their organization, interrupts and exceptions. Von Neumann Computer, System Bus. Instruction Cycle, Data Representation, Machine instruction and Assembly Language.

---

1) Data Structures

Array - An array in C is a collection of elements of the same data type stored in contiguous memory locations. Arrays are used to store multiple values in a single variable, making It easier to manage and manipulate large amounts of data.

Characteristics of Arrays -

•**Fixed Size:** Once an array is declared, its size cannot be changed. You need to know the number of elements you will store in the array beforehand.

•**Homogeneous Elements:** All elements in an array must be of the same data type, such as all integers, all floats, or all characters.

•**Contiguous Memory:** Elements are stored in consecutive memory locations.

•**Indexed by Integers**: Each element in an array is assigned a unique integer called an index, which identifies its position within the array. Indexing usually starts at 0.

•**Efficient Access:** Because of the way arrays are stored in memory (contiguously), accessing any element by its index is very efficient. This direct access using the index is often referred to as "random access."

Operation	Time Complexity	Description
Access	O(1)	Directly access any element using index
Insertion	O(n)	At a specific position, elements must shift
Deletion	O(n)	Elements shift to fill the gap
Traversal	O(n)	Visiting each element one by one

☑ **Use Cases:**
•Storing a list of numbers, student marks, etc.
•Representing matrices (2D arrays)

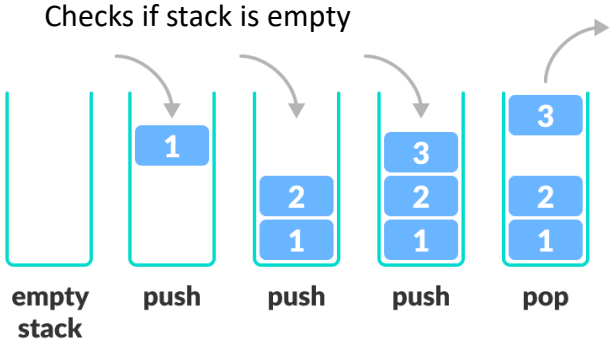
📚 2. Stacks

☑ **Definition:**
A **Stack** is a linear data structure that follows **LIFO (Last In, First Out)** principle. The last element inserted is the first one removed.

☑ **Operations:**

Operation	Description
push()	Inserts element on top of stack
pop()	Removes the top element
peek()	Returns the top element (no removal)
isEmpty()	Checks if stack is empty

☑ **Applications:**
•**Function calls** (call stack)
•**Undo/Redo** operations
•**Parentheses matching**
•**Expression evaluation and conversion**



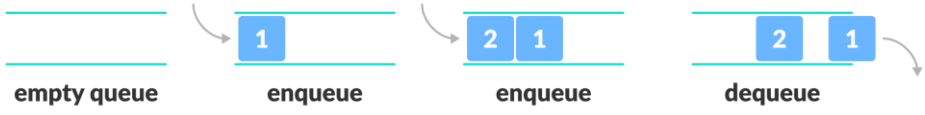
📚 3. Queues

☑ **Definition:**
A **Queue** is a linear data structure that follows **FIFO (First In, First Out)** principle. The first element inserted is the first one removed.

☑ **Operations:**

Operation	Description
enqueue()	Inserts element at the rear
dequeue()	Removes element from the front
peek()	Returns the front element without removing
isEmpty()	Checks if queue is empty

☑ **Types of Queues:**
•**Simple Queue**
•**Circular Queue**
•**Deque (Double-Ended Queue)**
•**Priority Queue** (based on priority, not insertion order)



Refer here for more detail on types of queues - https://www.programiz.com/dsa/types-of-queue

☑ **Applications:**
•CPU scheduling
•Print queues
•Data buffers in networking

4. Linked Lists

☑ **Definition:**
A **Linked List** is a dynamic data structure where each element (node) contains **data** and a **pointer to the next node**.

☑ **Types:**
•Singly Linked List: One pointer to the next node
•Doubly Linked List: Two pointers (next and previous)
•Circular Linked List: Last node points to the first

☑ **Operations:**
•Insertion at beginning, end, or any position
•Deletion of node
•Traversal

☑ **Advantages:**
•Dynamic size
•Efficient insertions/deletions

☑ **Disadvantages:**
•No random access
•More memory per node (pointers)

Representation of Linked list

```
struct node
{
  int data;
  struct node *next;
};
```

•A data item
•An address of another node

Singly Linked List

It is the most common. Each node has data and a pointer to the next node.

Singly linked list

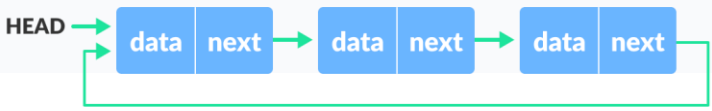
Doubly Linked List

We add a pointer to the previous node in a doubly-linked list. Thus, we can go in either direction: forward or backward.

Circular Linked List

A circular linked list is a variation of a linked list in which the last element is linked to the first element. This forms a circular loop.

Circular linked list



A circular linked list can be either singly linked or doubly linked.
•for singly linked list, next pointer of last item points to the first item
•In the doubly linked list, prev pointer of the first item points to the last item as well.

5. Sorting Techniques

Sorting rearranges elements in a specific order (ascending/descending).

Algorithm	Time Complexity	Stable?	Notes
Bubble Sort	$O(n^2)$	Yes	Simple, compares adjacent pairs
Selection Sort	$O(n^2)$	No	Finds min and swaps
Insertion Sort	$O(n^2)$	Yes	Efficient for small or sorted data
Merge Sort	O(n log n)	Yes	Divide & conquer, uses extra space
Quick Sort	O(n log n)*	No	Fastest in practice, not stable
Heap Sort	O(n log n)	No	Uses heap data structure

6. Searching Techniques

Searching finds the location or existence of a target value in a data structure.

☑ **Linear Search:**

- **Traverse all elements**
- **Time Complexity**: O(n)
- **Used for unsorted data**

Linear Search Applications

- For searching operations in smaller arrays (<100 items).

☑ **Binary Search:**

- Works only on **sorted data**
- Repeatedly divides the array in half
- **Time Complexity**: O(log n)
- Requires start, end, and mid pointers

Binary Search Applications

- In libraries of Java, .Net, C++ STL
- While debugging, the binary search is used to pinpoint the place where the error happens.

Trees

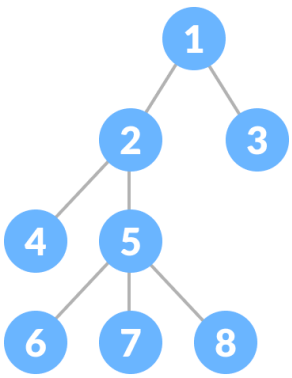
Tree Data Structure

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.

Why Tree Data Structure?

Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.

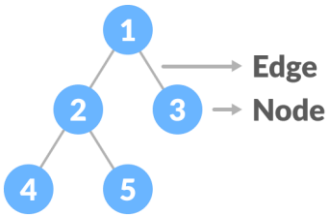
Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.



Tree Terminologies

Node

A node is an entity that contains a key or value and pointers to its child nodes.
The last nodes of each path are called **leaf nodes or external nodes** that do not contain a link/pointer to child nodes.
The node having at least a child node is called an **internal node**.



Edge

It is the link between any two nodes.

Root

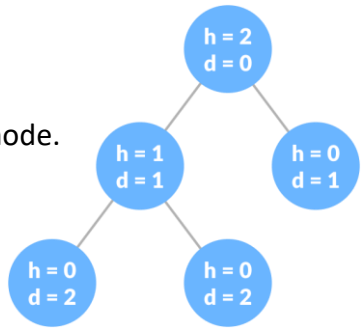
It is the topmost node of a tree.

Height of a Node

The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

Depth of a Node

The depth of a node is the number of edges from the root to the node.



Height of a Tree

The height of a Tree is the height of the root node or the depth of the deepest node.

Degree of a Node

The degree of a node is the total number of branches of that node.

Forest

A collection of disjoint trees is called a forest.

Types of Tree

1. Binary Tree
2. Binary Search Tree
3. AVL Tree
4. B-Tree

Binary Tree

A binary tree is a tree data structure in which each parent node can have at most two children. Each node of a binary tree consists of three items:

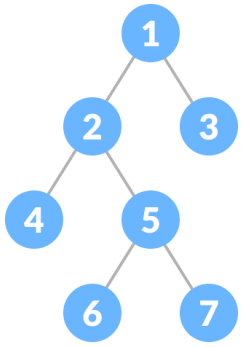
- data item
- address of left child
- address of right child

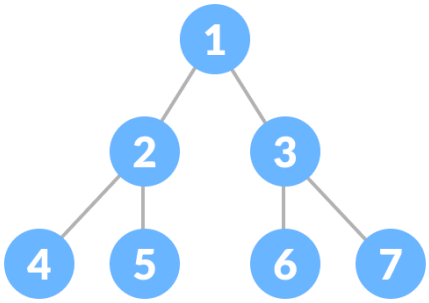
Types of Binary Tree

Types of Binary Tree

1. Full Binary Tree

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.





2. Perfect Binary Tree

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.

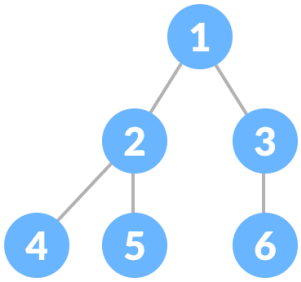
3. Complete Binary Tree

A complete binary tree is just like a full binary tree, but with two major differences

1.Every level must be completely filled

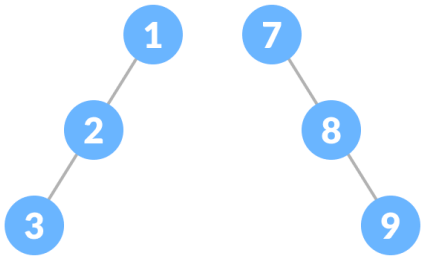
2.All the leaf elements must lean towards the left.

3.The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



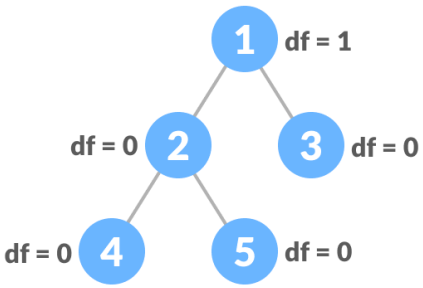
4. Skewed Binary Tree

A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: **left-skewed binary tree** and **right-skewed binary tree**.



5.Balanced Binary Tree

It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.



Binary Tree Applications

•For easy and quick access to data

•In router algorithms

•To implement heap data structure

•Syntax tree

Binary Search Tree(BST)

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

•It is called a binary tree because each tree node has a maximum of two children.

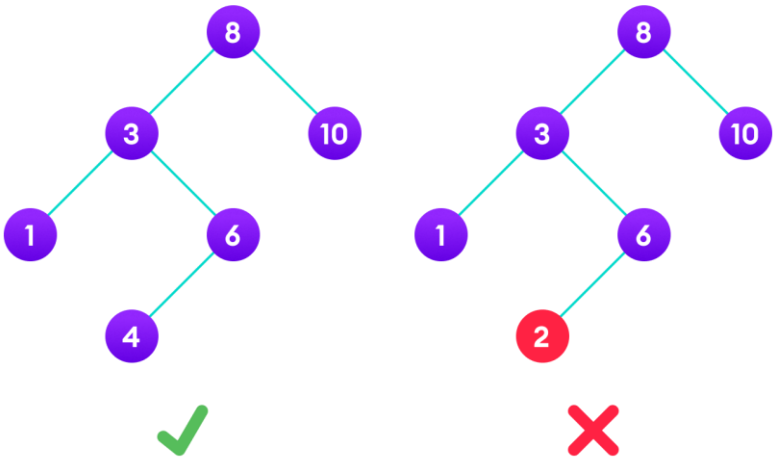
It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

The properties that separate a binary search tree from a regular binary tree is

1.All nodes of left subtree are less than the root node

2.All nodes of right subtree are more than the root node

3.Both subtrees of each node are also BSTs i.e. they have the above two properties



A tree having a right subtree with one value smaller than the root is shown to demonstrate that it is not a valid binary search tree

The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.

Time Complexity

Operation	Best Case Complexity	Average Case Complexity	Worst Case Complexity
Search	O(log n)	O(log n)	O(n)
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)

Binary Search Tree Applications

1.In multilevel indexing in the database
2.For dynamic sorting
3.For managing virtual memory areas in Unix kernel

AVL Tree

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

▨ Traversal Techniques

☑ Tree Traversals:

Traversal	Order	Use Case
Inorder	Left → Root → Right	BST gives sorted data
Preorder	Root → Left → Right	Copy tree, prefix expressions
Postorder	Left → Right → Root	Delete tree, postfix expressions

▨ Graph Terminology & Representation

☑ Terminology:

•**Vertex (Node)**: Basic unit
•**Edge**: Connection between vertices
•**Degree**: Number of edges of a node
•**Directed/Undirected Graph**
•**Weighted Graph**: Edges have weights

☑ Representation in Memory:

1. Adjacency Matrix:
•2D array of size V×V
•matrix[i][j] = 1 if edges exists
•**Space Complexity**: $O(V^2)$
2. Adjacency List:
•Array of lists
•Each list contains neighbors
•**Space Efficient**: O(V + E)
3. Edge List:
•List of all edges: (u, v)

☑ **Graph Traversals:**

1. BFS (Breadth-First Search):

•Uses a **queue**

•Visits nodes level by level

•Best for finding shortest path in unweighted graphs

2. DFS (Depth-First Search):

•Uses a **stack** or recursion

•Visits as deep as possible before backtracking

Data Structure or Algorithm Name	Best Case	Average Case	Worst Case
Arrays	O(1)	O(n)	O(n)
Linked Lists	O(1)	O(n)	O(n)
Stacks	O(1)	O(1)	O(1)
Queues	O(1)	O(1)	O(1)
Hash Tables	O(1)	O(1)	O(n)
Binary Search Trees (BST)	O(1)	O(log n)	O(n)
AVL Trees	O(log n)	O(log n)	O(log n)
Red-Black Trees	O(log n)	O(log n)	O(log n)
Heaps	O(1)	O(log n)	O(n)
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)
Bubble Sort	O(n)	O(n^2)	O(n^2)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	O(n^2)
Heap Sort	O(n log n)	O(n log n)	O(n log n)

2) Computer Networks

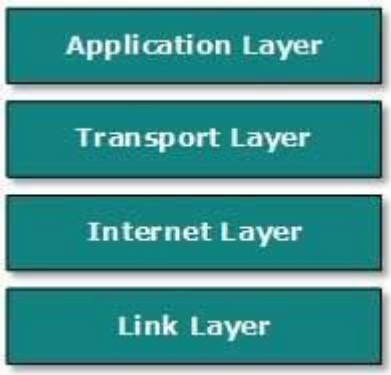
Network Models -

In the world of networking, two models play a vital role in describing the communication and data transfer processes between hosts: the OSI/ISO model and the TCP/IP model. These models provide a framework for understanding how information flows through a network and enable efficient data transmission. In this article, we will explore the fundamental concepts of these models and their significance in networking.

- **OSI/ISO**
- **TCP/IP**

Internet Model

Internet uses TCP/IP protocol suite, also known as Internet suite. This defines Internet Model which contains four layered architecture. OSI Model is general communication model but Internet Model is what the internet uses for all its communication. The internet is independent of its underlying network architecture so is its Model. This model has the following layers:



•**Application Layer**: This layer defines the protocol which enables user to interact with the network. For example, FTP, HTTP etc.

•**Transport Layer**: This layer defines how data should flow between hosts. Major protocol at this layer is Transmission Control Protocol (TCP). This layer ensures data delivered between hosts is in-order and is responsible for end-to-end delivery.

•**Internet Layer**: Internet Protocol (IP) works on this layer. This layer facilitates host addressing and recognition. This layer defines routing.

•**Link Layer**: This layer provides mechanism of sending and receiving actual data. Unlike its OSI Model counterpart, this layer is independent of underlying network architecture and hardware.

OSI Model

Open System Interconnect is an open standard for all communication systems. OSI model is established by International Standard Organization (ISO). This model has seven layers:

•**Application Layer**: This layer is responsible for providing interface to the application user. This layer encompasses protocols which directly interact with the user.

•**Presentation Layer**: This layer defines how data in the native format of remote host should be presented in the native format of host.

•**Session Layer**: This layer maintains sessions between remote hosts. For example, once user/password authentication is done, the remote host maintains this session for a while and does not ask for authentication again in that time span.

•**Transport Layer**: This layer is responsible for end-to-end delivery between hosts.

•**Network Layer**: This layer is responsible for address assignment and uniquely addressing hosts in a network.

•**Data Link Layer**: This layer is responsible for reading and writing data from and onto the line. Link errors are detected at this layer.

•**Physical Layer**: This layer defines the hardware, cabling wiring, power output, pulse rate etc.

3. OSI Model (Open Systems Interconnection Model)

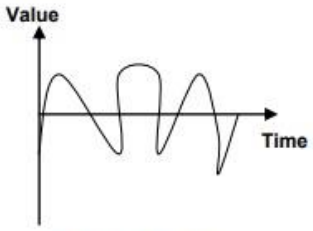
The **OSI model** is a conceptual framework that standardizes communication functions into seven layers:

- **Layer 1: Physical Layer**: Deals with the physical connection between devices (e.g., cables, switches).
- **Layer 2: Data Link Layer**: Ensures reliable data transfer over the physical medium (e.g., Ethernet).
- **Layer 3: Network Layer**: Handles routing, addressing, and forwarding packets (e.g., IP).
- **Layer 4: Transport Layer**: Provides end-to-end communication and error control (e.g., TCP, UDP).
- **Layer 5: Session Layer**: Manages sessions or connections between applications.
- **Layer 6: Presentation Layer**: Ensures data is in a usable format (e.g., encryption, compression).
- **Layer 7: Application Layer**: Provides application-level services (e.g., HTTP, FTP).

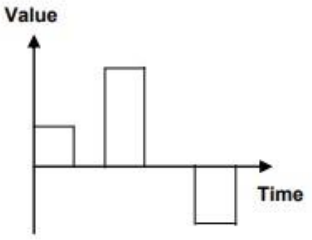
4. Physical Layer - Analog and Digital Signals

The **Physical Layer** deals with the transmission of raw data over a physical medium. This includes the use of analog and digital signals:

- **Analog Signals**: These are continuous signals that can take any value within a range (e.g., sound waves, electrical voltages). They are often used in traditional telephone lines and radio transmission.
- **Digital Signals**: These signals represent data as discrete values, usually binary (0s and 1s). Digital transmission is more reliable and efficient than analog because it is less susceptible to noise.



a. Analog Signal



b. Digital Signal

5. Analog and Digital Transmission

- **Analog Transmission**: Involves the transmission of continuous signals. Analog systems are subject to noise, which can distort the signal, reducing the quality of transmission.
- **Digital Transmission**: Involves transmitting data in discrete forms, which is more resilient to noise and can be amplified without degradation.

6. Coding and Sampling

- **Coding**: Refers to the conversion of data into a specific format that is suitable for transmission. For example, binary encoding uses 0s and 1s to represent data.
- **Sampling**: In analog-to-digital conversion, **sampling** refers to the process of taking periodic measurements of an analog signal at discrete intervals to convert it into a digital signal. The more frequent the sampling, the more accurately the digital signal will represent the original analog signal. This is governed by the **Nyquist theorem**, which states that the sampling rate should be at least twice the frequency of the analog signal.

7. Data Link Layer

The **Data Link Layer** is responsible for the reliable transmission of data across a physical link. It handles error detection and correction, framing, and flow control. The primary protocols include Ethernet and PPP.

Error Detection and Correction

- **Error Detection**: Mechanisms like **parity bits**, **checksum**, and **Cyclic Redundancy Check (CRC)** are used to detect errors in transmitted data.
- **Error Correction**: This involves techniques to correct errors without the need for retransmission, such as **Hamming code** and **Reed-Solomon codes**.

Data Link Control and Protocols

•**Stop-and-Wait**: A simple protocol where the sender transmits one frame and waits for an acknowledgment from the receiver before sending the next frame.

•**Go-Back-N**: A protocol in which the sender can send multiple frames, but if an error is detected, the receiver asks for retransmission of all frames from the erroneous one onward.

•**Selective Repeat**: A more efficient version of Go-Back-N, where only the frames with errors are retransmitted, and the receiver can accept out-of-order frames.

8. Network Layer

The **Network Layer** is responsible for routing packets across the network, logical addressing, and packet forwarding. It determines how data is routed from the source to the destination across multiple networks.

Internetworks

•**Internetworks** refer to interconnected networks, such as how the Internet is formed by connecting various local and wide area networks.

Addressing

•**IP Addressing**: Devices on a network are identified by IP addresses. There are two versions of IP addresses: IPv4 (32-bit) and IPv6 (128-bit).

•**Subnetting**: Divides an IP network into smaller sub-networks for efficient management and security.

Unicast and Multicast Routing

•**Unicast**: Refers to one-to-one communication between two devices on a network (e.g., a client-server connection).

•**Multicast**: Refers to one-to-many or many-to-many communication, where data is sent to multiple devices at once (e.g., streaming video, conference calls).

9. Presentation Layer

The **Presentation Layer** is responsible for translating, encrypting, and compressing data for the application layer. It ensures that data is in a usable format for the receiving system. Some key tasks include:

•**Data Encoding**: Converting data into a standard format for transmission (e.g., ASCII, EBCDIC).

•**Data Compression**: Reducing the size of data for more efficient transmission (e.g., ZIP, JPEG).

•**Encryption**: Securing data by converting it into an unreadable format that can only be deciphered with the appropriate decryption key (e.g., SSL/TLS encryption for secure web browsing).

3) Programming in C

1. Data Types in C

Data types define the type of data that a variable can store. In C, data types can be categorized into:

•**Basic Data Types:**

- **int**: Represents integer values. Size depends on the system (usually 4 bytes on most systems).
- **char**: Represents a single character or a small integer (1 byte).
- **float**: Represents single-precision floating-point numbers (usually 4 bytes).
- **double**: Represents double-precision floating-point numbers (usually 8 bytes).
- **void**: Represents an absence of type, typically used for functions that do not return a value.

Derived Data Types:

•**Arrays:** A collection of elements of the same data type.

•**Pointers:** A variable that stores the memory address of another variable.

•**Structures:** A user-defined data type that groups variables of different types under a single name.

•**Unions:** Similar to structures but share memory for all members, saving space.

Enumeration Data Types (enum):

•An enum is a user-defined type that consists of a set of named integer constants.

Declarations in C

•**Variable Declaration:**

- Variables must be declared before they can be used. A declaration provides the type and the name of the variable.

```
int age;      // Declares an integer variable called age
float salary; // Declares a floating-point variable called salary
```

3. Expressions and Statements

•**Expression:**

- An expression is a combination of variables, constants, operators, and functions that evaluates to a value.

•Example:

```
int result = a + b * c; // Expression that evaluates to a value
```

Statement:

•A statement is a single line of code that performs an action, such as variable assignment, function call, or control flow change.

Example:

```
x = 10;  // Assignment statement
```

4. Symbolic Constants

•**Define Symbolic Constants:**

•In C, symbolic constants are defined using **#define** or **const** for constants that do not change during the execution.

Example:

```
#define MAX 100    // Macro constant
const float PI = 3.14; // Constant variable
```

5. Input and Output Functions

Input Function: The **scanf()** function is used to take input from the user.
Example:

```
int num;
printf("Enter a number: ");
scanf("%d", &num);
```

Output Function:
The **printf()** function is used to display output to the console.
Example:

```
printf("The number is: %d", num);
```

6. Operators and Expressions
Operators are symbols that perform operations on variables and values. C has several types of operators:

1. Arithmetic Operators:
- Used for basic mathematical calculations.
- Examples:
  - + (addition)
  - − (subtraction)
  - * (multiplication)
  - / (division)
  - % (modulo/remainder)
- Example: `int result = a + b;`

2. Relational Operators:
- Used to compare two values and return a boolean result (0 or 1).
- Examples:
  - == (equal to)
  - != (not equal to)
  - > (greater than)
  - < (less than)
  - >= (greater than or equal to)
  - <= (less than or equal to)
- Example: `if (a > b) { ... }`

3. Logical Operators:
- Used to combine or modify boolean expressions.
- Examples:
  - && (AND)
  - || (OR)
  - ! (NOT)
- Example: `if (a > 0 && b < 10) { ... }`

4. Bitwise Operators:
- Used to manipulate individual bits of a variable.
- Examples:
  - & (AND)
  - | (OR)
  - ^ (XOR)
  - ~ (NOT)
  - << (left shift)
  - >> (right shift)

•Example: `int result = a & b;`

5. Assignment Operators:

•Used to assign values to variables.

•Examples:

- = (assignment)
- += (add and assign)
- -= (subtract and assign)
- *= (multiply and assign)
- /= (divide and assign)
- %= (modulo and assign)

•Example: `a = 5;`

6. Conditional Operator:

•A ternary operator that evaluates a condition and returns one of two values based on the result.

•Syntax: `(condition) ? value_if_true : value_if_false`

•Example: `int max = (a > b) ? a : b;`

7. Increment and Decrement Operators:

•Used to increase or decrease the value of a variable by 1.

•Examples:

- ++ (increment)
- -- (decrement)

•Example: `a++;`

8. Comma Operator:

•Evaluates multiple expressions in a single statement and returns the value of the last expression.

•Example: `(a = 1, b = 2);`

9. Sizeof Operator:

•Returns the size of a data type or variable in bytes.

•Example: `sizeof(int);`

7. Control Statements

Control statements manage the flow of execution in a program:

•**While Loop:**

- Executes a block of code as long as the condition is true.

•Example:

```
while (x < 10) {
   printf("%d\n", x);
   x++;
}
```

Do-While Loop:

•Executes a block of code at least once and then repeats as long as the condition is true.

Example:

```
do {
   printf("%d\n", x);
   x++;
} while (x < 10);
```

For Loop:

•Used when the number of iterations is known beforehand.

Example:

```
for (int i = 0; i < 10; i++) {
   printf("%d\n", i);
}
```

If-Else Statement:

•Used for decision-making.

Example:

```
if (x > 5) {
   printf("x is greater than 5");
} else {
   printf("x is not greater than 5");
}
```

Switch Statement:

•Used for selecting one of many code blocks to be executed based on the value of an expression.

Example:

```
switch (day) {
   case 1:
      printf("Monday");
      break;
   case 2:
      printf("Tuesday");
      break;
   default:
      printf("Invalid day");
}
```

Break Statement:

•Exits a loop or switch statement prematurely.

Example:

```
while (x < 10) {
   if (x == 5) break;
   x++;
}
```

Continue Statement:

•Skips the current iteration of a loop and moves to the next one.

Example:

```
for (int i = 0; i < 10; i++) {
   if (i == 5) continue;
   printf("%d\n", i);
}
```

C Storage Class

Every variable in C programming has two properties: type and storage class.

Type refers to the data type of a variable. And, storage class determines the scope, visibility and lifetime of a variable.

There are 4 types of storage class:

1.automatic
2.external
3.static
4.Register

Local Variable

The variables declared inside a block are automatic or local variables. The local variables exist only inside the block in which it is declared.

Let's take an example.

```
#include <stdio.h>

int main(void) {

 for (int i = 0; i < 5; ++i) {
    printf("C programming");
 }
// Error: i is not declared at this point
 printf("%d", i);
 return 0;
}
```

Global Variable

Variables that are declared outside of all functions are known as external or global variables. They are accessible from any function inside the program.

```
#include <stdio.h>
void display();

int n = 5;  // global variable

int main()
{
   ++n;
   display();
   return 0;
}

void display()
{
   ++n;
   printf("n = %d", n);
}
```

Output
N = 7

Register Variable

The register keyword is used to declare register variables. Register variables were supposed to be faster than local variables.
However, modern compilers are very good at code optimization, and there is a rare chance that using register variables will make your program faster.
Unless you are working on embedded systems where you know how to optimize code for the given application, there is no use of register variables.

Static Variable

A static variable is declared by using the static keyword for example;

```
static int i;
```

The value of a static variable persists until the end of the program i.e will be fixed unitl the end of the program.

Functions in C

A function is a block of code which only runs when it is called.
You can pass data, known as parameters, into a function.
Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

Predefined Functions

So it turns out you already know what a function is. You have been using it the whole time while studying this tutorial!
For example, **main()** is a function, which is used to execute code, and **printf()** is a function; used to output/print text to the screen.

Create a Function

To create (often referred to as *declare*) your own function, specify the name of the function, followed by parentheses ( ) and curly brackets { }
Syntax:

```
void myFunction() {
  // code to be executed
}
```

Call a Function

Declared functions are not executed immediately. They are "saved for later use", and will be executed when they are called.

To call a function, write the function's name followed by two parentheses () and a semicolon ;

In the following example, **myFunction()** is used to print a text (the action), when it is called:

```
// Create a function
void myFunction() {
  printf("I just got executed!");
}

int main() {
  myFunction(); // call the function
  return 0;
}

// Outputs "I just got executed!“
```

A function can be called multiple times

Passing Arguments:

•Arguments can be passed to functions either by value (default) or by reference using pointers.

1. Pass by Value

When you pass a variable to a function **by value**, a copy of the variable is passed, and any modifications made to the parameter inside the function will not affect the original variable.

Example:

```
#include <stdio.h>

// Function to add two integers
void add(int a, int b) {
   int sum = a + b;
   printf("Sum inside function: %d\n", sum);
}

int main() {
   int x = 5;
   int y = 10;

   printf("Before function call: x = %d, y = %d\n", x, y);

   // Passing by value
   add(x, y);

   printf("After function call: x = %d, y = %d\n", x, y);

   return 0;
}
```

Output

Before function call: x = 5, y = 10
Sum inside function: 15
After function call: x = 5, y = 10

2. Pass by Reference (using Pointers)

When you pass a variable to a function **by reference** (using pointers), the function can modify the actual value of the variable in the calling function. This is because you pass the **address** of the variable rather than its value.

Example:

```
#include <stdio.h>

// Function to swap two integers by passing by reference (using pointers)
void swap(int *a, int *b) {
    int temp = *a;  // Dereferencing to get the value of a
    *a = *b;        // Dereferencing to set the value of a to b
    *b = temp;      // Dereferencing to set the value of b to the old value of a
}

int main() {
    int x = 5;
    int y = 10;

    printf("Before swap: x = %d, y = %d\n", x, y);

    // Passing by reference (using pointers)
    swap(&x, &y);  // Pass the address of x and y

    printf("After swap: x = %d, y = %d\n", x, y);

    return 0;
}
```

Output

Before swap: x = 5, y = 10

After swap: x = 10, y = 5

Explanation:

In this case, the function **swap()** accepts pointers (int *a, int *b) as arguments.

The addresses of **x** and **y** are passed to the function using the & operator.

Inside the **swap()** function, the values at the addresses of **x** and **y** are modified, effectively swapping the original values of x and y.

4) Database Management System

A **DBMS** is a software system designed to manage databases, enabling users to store, retrieve, and manipulate data efficiently. DBMS architecture defines how various components of the system interact and how data is managed.

DBMS Architecture Levels:

1.Internal Level (Physical Level):

1. Deals with the physical storage of data on hardware. It includes the data structures, storage devices, and file systems.
2. This level is concerned with how data is stored and organized on disk.

2.Conceptual Level:

1. The logical view of the entire database.
2. It abstracts the details of the physical storage and focuses on the logical structure and relationships between data.
3. Users and developers interact with this level.

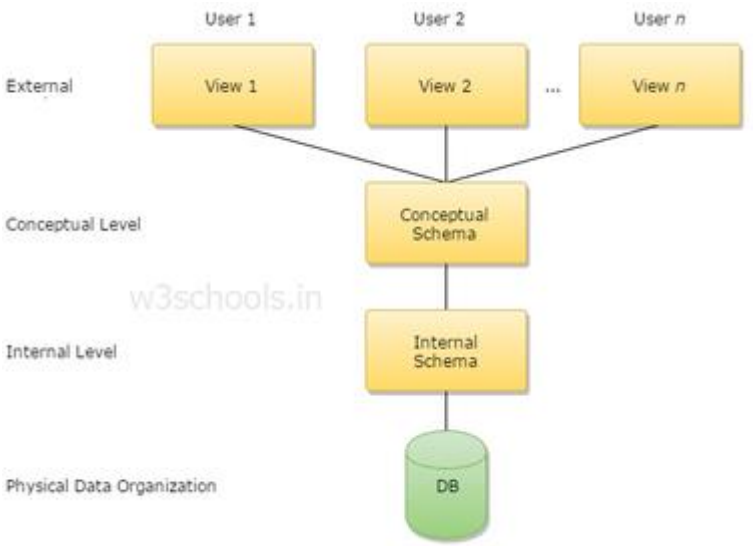
3.External Level (View Level):

1. The highest level of abstraction, where individual users or user groups see the data in a way that is relevant to them.
2. Multiple views of the database can exist for different users, each showing only the data that is important to them.

Three-Level Architecture:

•The DBMS is typically designed in a **three-tier** architecture:

- **External**: Different views for different users.
- **Conceptual**: A unified, logical view of the entire database.
- **Internal**: Physical storage and how data is actually stored on disks.



2. Data Models

A **data model** defines the structure of data, the relationships among different data elements, and the operations allowed on the data. Common data models include:

•**Hierarchical Model**:

- Data is organized in a tree-like structure, where each record has a single parent.
- This model is less flexible as it requires a predefined hierarchy.

•**Network Model**:

- Similar to the hierarchical model, but a record can have multiple parent records, forming a graph structure.

•**Relational Model** (most widely used):
- Data is stored in tables (relations), and the relationships between data are established using keys.
- Tables contain rows (tuples) and columns (attributes).

•**Object-Oriented Model**:
- Data is represented as objects, similar to objects in object-oriented programming languages. It supports encapsulation, inheritance, and polymorphism.

•**Entity-Relationship (E-R) Model**:
- A high-level conceptual model used to design databases. It defines entities (objects) and relationships between them.

Entity-Relationship Model

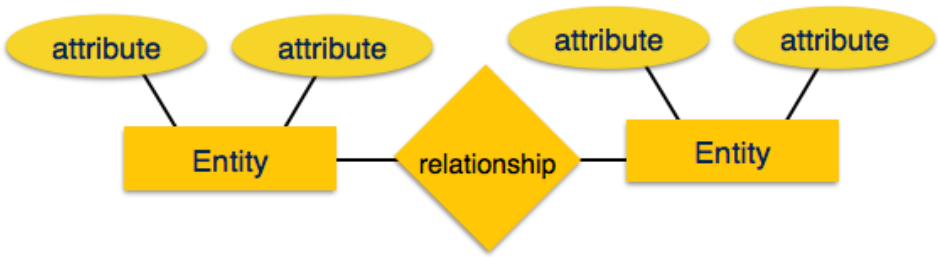
Entity-Relationship (ER) Model is based on the notion of real-world entities and relationships among them. While formulating real-world scenario into the database model, the ER Model creates entity set, relationship set, general attributes and constraints.

ER Model is best used for the conceptual design of a database.

ER Model is based on −

•**Entities** and their *attributes.*

•**Relationships** among entities.



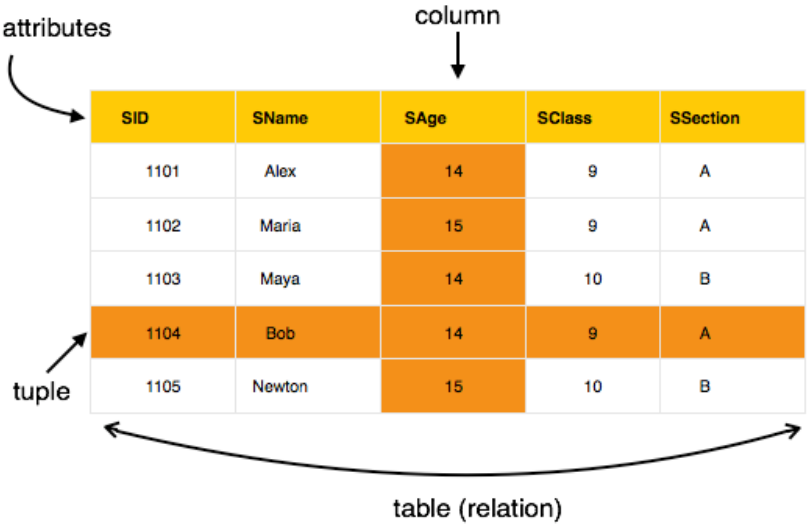
•**Entity** − An entity in an ER Model is a real-world entity having properties called **attributes**. Every **attribute** is defined by its set of values called **domain**. For example, in a school database, a student is considered as an entity. Student has various attributes like name, age, class, etc.

•**Relationship** − The logical association among entities is called ***relationship***. Relationships are mapped with entities in various ways. Mapping cardinalities define the number of association between two entities.

•Mapping cardinalities −
- one to one
- one to many
- many to one
- many to many

Relational Model

The most popular data model in DBMS is the Relational Model. It is more scientific a model than others. This model is based on first-order predicate logic and defines a table as an **n-ary relation**.

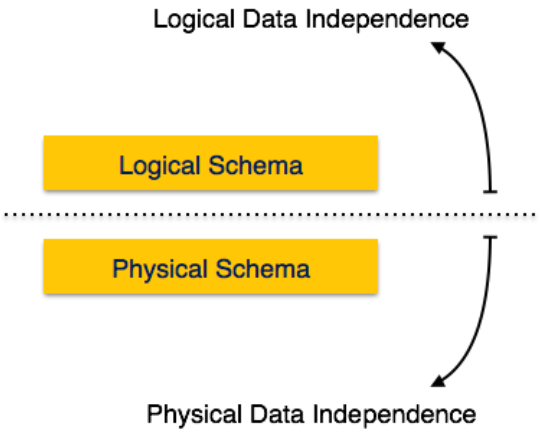


SID	SName	SAge	SClass	SSection
1101	Alex	14	9	A
1102	Maria	15	9	A
1103	Maya	14	10	B
1104	Bob	14	9	A
1105	Newton	15	10	B

The main highlights of this model are −
•Data is stored in tables called **relations**.
•Relations can be normalized.
•In normalized relations, values saved are atomic values.
•Each row in a relation contains a unique value.
•Each column in a relation contains values from a same domain.

3. Data Independence

A database system normally contains a lot of data in addition to users data. For example, it stores data about data, known as metadata, to locate and retrieve data easily. It is rather difficult to modify or update a set of metadata once it is stored in the database. But as a DBMS expands, it needs to change over time to satisfy the requirements of the users. If the entire data is dependent, it would become a tedious and highly complex job.



- **Logical Data Independence**

Logical data is data about database, that is, it stores information about how data is managed inside. For example, a table (relation) stored in the database and all its constraints, applied on that relation.
Logical data independence is a kind of mechanism, which liberalizes itself from actual data stored on the disk. If we do some changes on table format, it should not change the data residing on the disk.

- **Physical Data Independence**

All the schemas are logical, and the actual data is stored in bit format on the disk. Physical data independence is the power to change the physical data without impacting the schema or logical data.
For example, in case we want to change or upgrade the storage system itself − suppose we want to replace hard-disks with SSD − it should not have any impact on the logical data or schemas.

4. Relational Model: Concepts, Constraints, and Languages

Relational Model Concepts:
•**Relation**: A table with rows (tuples) and columns (attributes).
•**Tuple**: A row in a table, representing a single record.
•**Attribute**: A column in a table, representing a data field.
•**Domain**: The set of permissible values for an attribute.

Constraints in Relational Model:
•**Primary Key**: A unique identifier for each record in a table.
•**Foreign Key**: A field that links two tables together. It refers to the primary key of another table.
•**Unique Constraint**: Ensures that all values in a column are unique.
•**Not Null Constraint**: Ensures that a column cannot have a NULL value.
•**Check Constraint**: Ensures that a column only accepts values that satisfy a specific condition.
•**Default Constraint**: Assigns a default value to a column when no value is specified.

Relational Query Languages:

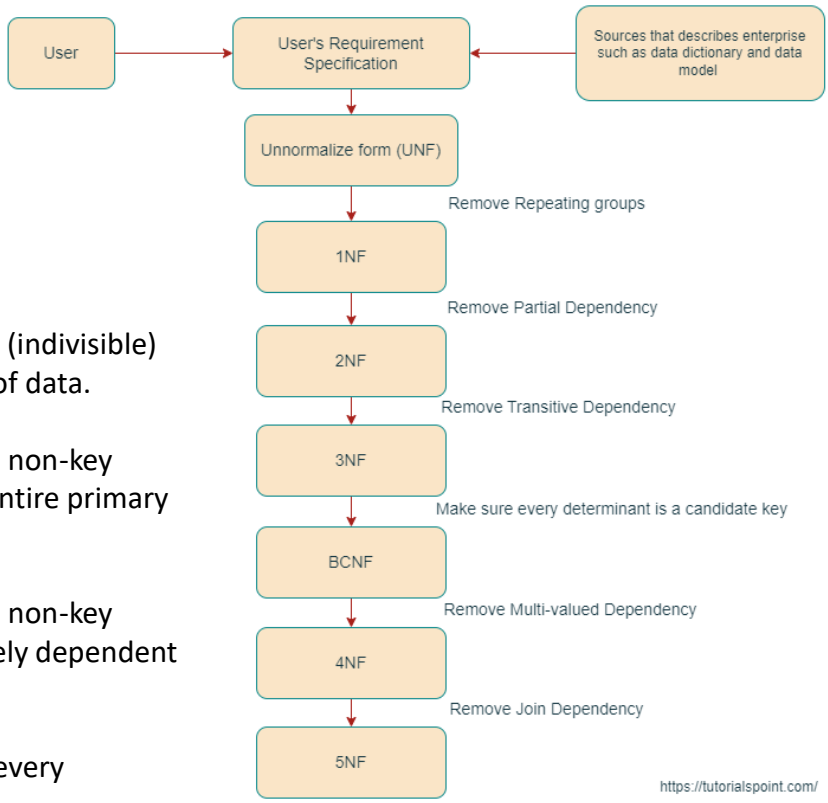
•**SQL (Structured Query Language)**: The most widely used language for interacting with relational databases. It includes:

- **DDL (Data Definition Language)**: Used to define and modify database structure (e.g., CREATE,ALTER,DROP)
- **DML (Data Manipulation Language)**: Used for querying and modifying data (e.g., SELECT, INSERT, UPDATE, DELETE)
- **DCL (Data Control Language)**: Used for managing permissions (e.g., GRANT, REVOKE)
- **TCL (Transaction Control Language)**: Used for managing transactions (e.g., COMMIT, ROLLBACK)

5.Normalization

Normalization is a process of organizing the data in database to avoid data redundancy, insertion anomaly, update anomaly & deletion anomaly.

Normalization is the process of organizing data in a database to minimize redundancy and dependency. In database design, there are different normal forms based on the primary keys of a table. These include ?



Normal Forms:

•**1st Normal Form (1NF)**:

- The table must have only atomic (indivisible) values and no repeating groups of data.

•**2nd Normal Form (2NF)**:

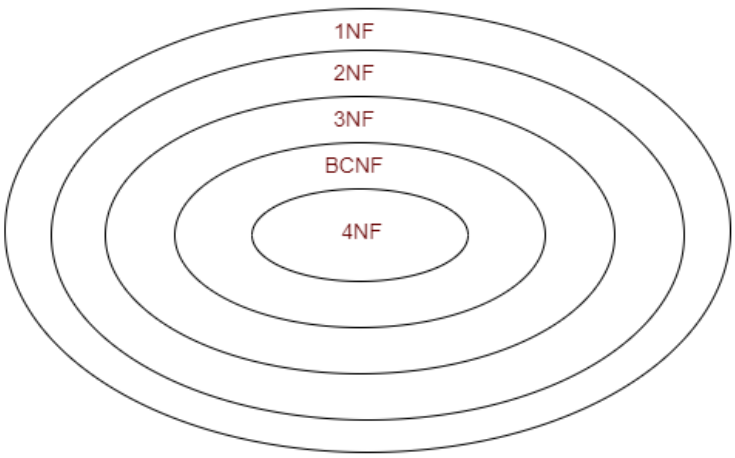
- The table must be in 1NF, and all non-key attributes must depend on the entire primary key.

•**3rd Normal Form (3NF)**:

- The table must be in 2NF, and all non-key attributes must be non-transitively dependent on the primary key.

•**Boyce-Codd Normal Form (BCNF)**:

- A stricter version of 3NF, where every determinant is a candidate key.



Advantages and Disadvantages of Normalization

Advantages of Normalization

•Reduced Data Redundancy
•Improved Data Consistency
•Simplified Database Maintenance
•Improved Query Performance

Disadvantages of Normalization

•Increased Complexity
•Decreased Read Performance
•Increased Write Performance
•Increased Storage Space
•Over-Normalization

6. Data Storage, Indexing, and Query Processing

Data Storage:

•**Storage Hierarchy**: The storage system typically includes memory (RAM), disk storage, and backup storage.

•**Tables**: Data is stored in tables (relations). Each table corresponds to a file or a set of files in the storage system.

Indexing:

•**Index**: A data structure that speeds up data retrieval operations.

•**Types of Indexes**:

- **Primary Index**: Automatically created for the primary key.
- **Secondary Index**: Created for other columns to speed up search operations.
- **B-tree Index**: A balanced tree structure commonly used in DBMS for efficient searching and sorting.
- **Hash Index**: Uses a hash function to map data to fixed-size buckets.

Query Processing:

•**Query Optimization**: DBMS uses query optimization techniques to select the most efficient way to execute a query.

•**Execution Plan**: The DBMS generates an execution plan, which includes operations like table scans, joins, and indexes.

•**Cost-Based Optimization**: The DBMS estimates the cost of different query execution plans and selects the one with the lowest cost.

7. Design and Programming in SQL

•**SQL Syntax**: SQL queries follow a structured syntax for interacting with databases.

- **SELECT Statement**: Retrieves data from one or more tables.

```
SELECT name, age FROM employees WHERE department = 'Sales';
```

INSERT Statement: Adds new rows to a table.

```
INSERT INTO employees (id, name, department) VALUES (1, 'John Doe', 'Sales');
```

UPDATE Statement: Modifies existing data in a table.

```
UPDATE employees SET department = 'Marketing' WHERE id = 1;
```

DELETE Statement: Removes rows from a table.

```
DELETE FROM employees WHERE id = 1;
```

JOIN Operations: Combine data from multiple tables.

```
SELECT employees.name, departments.name FROM employees
JOIN departments ON employees.department_id = departments.id;
```

GROUP BY and Aggregates: Group data for summarization.

```
SELECT department, COUNT(*) FROM employees GROUP BY department;
```

5) Operating Systems

1. Process Management in Operating Systems

Process management is one of the key functionalities of an operating system (OS), ensuring that processes are created, scheduled, executed, and terminated effectively. It involves the management of both processes and threads.

Process:

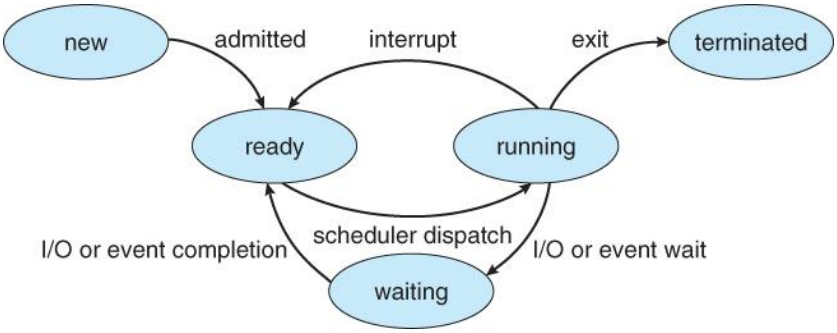
•A **process** is a program in execution, which includes the program code, data, and the state of the program.

•A process may contain multiple threads, which are the smallest unit of execution.

Process States:

A process goes through different states during its execution, which are managed by the OS:

1.New: The process is being created.
2.Ready: The process is ready to be executed, waiting for CPU time.
3.Running: The process is currently being executed on the CPU.
4.Waiting (Blocked): The process is waiting for some event (e.g., I/O completion).
5.Terminated: The process has completed its execution.



Process Control Block (PCB):

The **Process Control Block** is a data structure that stores information about the process, such as:

•Process state (new, ready, running, waiting, terminated).

•Program counter (the address of the next instruction to be executed).

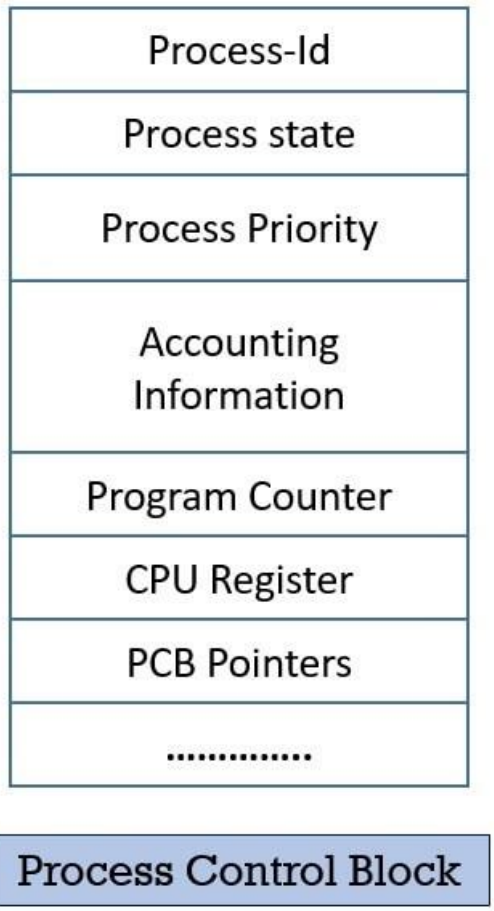
•CPU registers (used by the process).

•Memory management information (e.g., base and limit registers).

•Process ID (PID).

•I/O status information (e.g., open files).

The PCB is maintained by the OS to manage the process and is used when context switching occurs.



Processes and Threads:

•A **thread** is a smaller unit of execution within a process. Multiple threads within the same process share resources, such as memory and open files.

•Threads are more lightweight compared to processes because they don't require their own memory space. They share the memory of the parent process.

CPU Scheduling:

CPU scheduling in OS is a method by which one process is allowed to use the CPU while the other processes are kept on hold or are kept in the waiting state. This hold or waiting state is implemented due to the unavailability of any of the system resources like I/O etc. Thus, the purpose of CPU scheduling in OS is to result in an efficient, faster and fairer system.

As soon as the CPU becomes idle, the operating system selects one of the processes that are in the ready queue, waiting to be executed. The short-term scheduler also known as the CPU scheduler carries out the task of selecting a **process** from among the processes in memory that are waiting to be executed and allocates the CPU to one of these.

Important Terms Related to CPU Scheduling in OS

1.Arrival time in OS: It is the time of arrival of a process in the ready queue.

2.Completion Time in OS: It is the time at which the selected process completes its execution.

3.Burst Time in OS: This is the time required by a process for its execution by the CPU.

4.Turn Around Time in OS: Turnaround time for each process is the difference in completion time and the time of arrival of that process. Turn around time = Completion time – Arrival time

5.Waiting time in OS: The waiting time for each and every process is the difference between the turnaround time and burst time of that process.

Waiting time = Turnaround time – Burst time.

The types of scheduling in OS are primarily categorized into two main categories:

•Preemptive Scheduling

•Non-Preemptive Scheduling

Preemptive Scheduling

Preemptive scheduling is implemented when a process switches from its running state to a ready state or from its waiting state to the ready state.

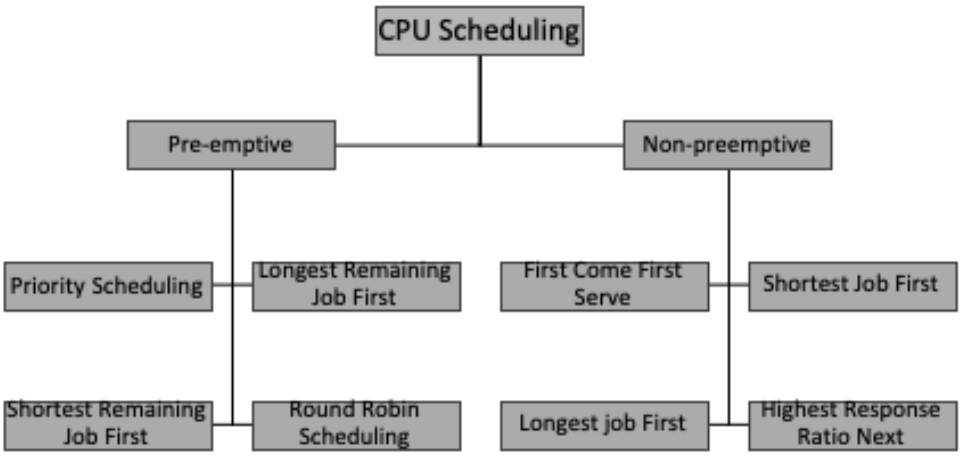
Here the assignment of tasks is often prioritized. If a task has a higher priority than the running process, then this process will be interrupted and resumed only after the completion of the task with a higher priority.

At times the cyclic allocation of the CPU takes place. A process is allocated to the CPU for a limited amount of time and then taken away for the next process, while the earlier process is again placed in the ready queue if the CPU burst time is still left. This process remains in the ready queue till the CPU is allocated again.

Non-Preemptive Scheduling

Non-Preemptive scheduling is used when a process terminates or completes its execution, or when a process switches from its running state to the waiting state.

Here a CPU that has been allocated once to a process is not released until the process has terminated or switched to the waiting state. A running process is not interrupted during its execution, and the next process waits for CPU allocation till the current process completes its CPU burst time.



Types of Scheduling Algorithms

1. First Come First Serve:
Of all the scheduling algorithms it is one of the simplest and easiest to implement. As the name suggests the First Come First Serve scheduling algorithm means that the process that requests the CPU first is allocated the CPU first. It is basically implemented using a First In First Out queue. It supports both non-preemptive and preemptive CPU scheduling algorithms.

2. Shortest Job First (SJF):
Shortest job first (SJF) is a scheduling algorithm that selects the waiting process with the smallest execution time to be executed next. The method followed by the SJF scheduling algorithm may or may not be preemptive. SJF reduces the average waiting time of other waiting processes significantly.

3. Longest Job First (LJF):
The working of the Longest Job First(LJF) scheduling process is the opposite of the shortest job first (SJF), Here, the process with the largest burst time is processed first. Longest Job First is one of the non-preemptive algorithms.

4. Priority Scheduling:
The Priority CPU Scheduling Algorithm is one of the pre-emptive methods of CPU scheduling. Here, the most important process must be done first. In the case there are more than one processes with the same priority value then the FCFS algorithm is used to resolve the situation.

5. Round Robin:
In the Round Robin CPU scheduling algorithm the processes are allocated CPU time in a cyclic order with a fixed time slot for each process. It is said to be a pre-emptive version of FCFS. With the cyclic allocation of equal CPU time, it works on the principle of time-sharing.

6. Shortest Remaining Time First:
The Shortest remaining time first CPU scheduling algorithm is a preemptive version of the Shortest job first scheduling algorithm. Here the CPU is allocated to the process that needs the smallest amount of time for its completion. Here the short processes are handled very fast but the time taking processes keep waiting.

7. Longest Remaining Time First:
The longest remaining time first CPU scheduling algorithm is a preemptive CPU scheduling algorithm. This algorithm selects those processes first which have the longest processing time remaining for completion i.e. processes with the largest burst time are allocated the CPU time first.

8. Highest Response Ratio Next:
The Highest Response Ratio Next is one of the non-preemptive CPU Scheduling algorithms. It has the distinction of being recognized as one of the most optimal scheduling algorithms. As the name suggests here the CPU time is allocated on the basis of the response ratio of all the available processes where it selects the process that has the highest Response Ratio. The selected process will run till its execution is complete.

3. Process Synchronization and Deadlock

Process Synchronization:

Process synchronization involves the coordination and control of concurrent processes to ensure correct and predictable outcomes. Its primary purpose is to prevent race conditions, data inconsistencies, and resource conflicts that may arise when multiple processes access shared resources simultaneously.

•**Synchronization** ensures that multiple processes or threads can work together without causing inconsistent or unpredictable results.

Challenges in Concurrent Execution

Concurrent execution introduces several challenges, including −

•**Race Conditions** − Concurrent processes accessing shared resources may result in unexpected and erroneous outcomes. For example, if two processes simultaneously write to the same variable, the final value may be unpredictable or incorrect.

•**Deadlocks** − Processes may become stuck in a state of waiting indefinitely due to resource dependencies. Deadlocks occur when processes are unable to proceed because each process is waiting for a resource held by another process, creating a circular dependency.

•**Starvation** − A process may be denied access to a shared resource indefinitely, leading to its inability to make progress. This situation arises when certain processes consistently receive priority over others, causing some processes to wait indefinitely for resource access.

•**Data Inconsistencies** − Inconsistent or incorrect data may occur when processes manipulate shared data concurrently. For example, if multiple processes simultaneously update a database record, the final state of the record may be inconsistent or corrupted.

Common synchronization techniques include:

Synchronization Mechanisms

Mutual Exclusion

In order to avoid conflicts when processes need to use a shared resource mutual exclusion plays a crucial role in synchronizing them. Locks, semaphores and similar synchronization primitives are often utilized for ensuring exclusive access. By allowing only one process to access a shared resource at any given time, mutual exclusion prevents data races and ensures data consistency.

Semaphores (Solves dining philosopher's Problem & Producer Consumer Problem)

Semaphores are synchronization objects that maintain a count and allow or restrict access to resources based on the count value. Successful management of shared resources and coordinated process execution often require reliable tools such as semaphores. These flexible mechanisms allow for control over various resource requirements - whether binary (0 or 1) or non binary (greater than 1). Ultimately. This approach ensures efficient use of available resources without compromising other vital processes.

Monitors

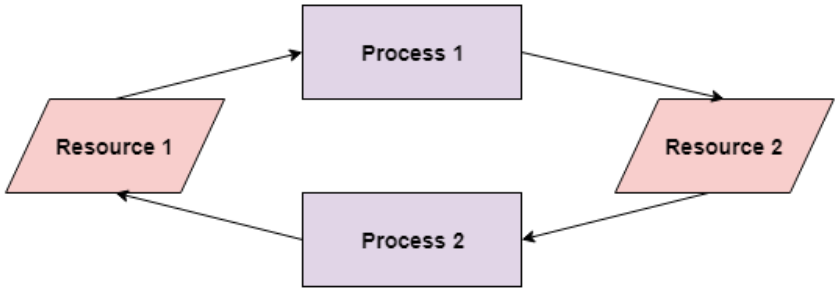
Monitors are higher-level synchronization constructs that encapsulate shared data and the procedures that operate on them. They ensure that only one process can execute a procedure within the monitor at any given time, preventing concurrent access to shared data. Monitors provide a structured and controlled way to synchronize concurrent processes, often using condition variables to manage process coordination.

Condition Variables

Condition variables are synchronization primitives used in conjunction with locks or monitors to enable processes to wait for specific conditions to be satisfied before proceeding. They provide a means for processes to communicate and coordinate their actions. Processes can wait on a condition variable until another process signals or broadcasts that the condition has been met.

Deadlock

A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.



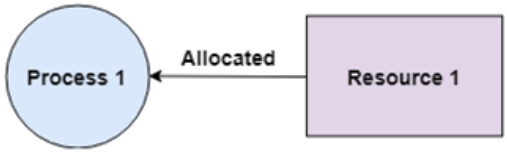
Deadlock in Operating System

In the above diagram, the process 1 has resource 1 and needs to acquire resource 2. Similarly process 2 has resource 2 and needs to acquire resource 1. Process 1 and process 2 are in deadlock as each of them needs the other's resource to complete their execution but neither of them is willing to relinquish their resources.

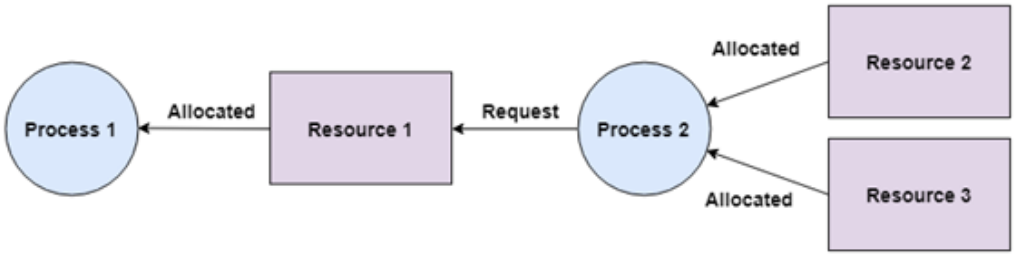
Coffman Conditions

A deadlock occurs if the four Coffman conditions hold true. But these conditions are not mutually exclusive. The Coffman conditions are given as follows

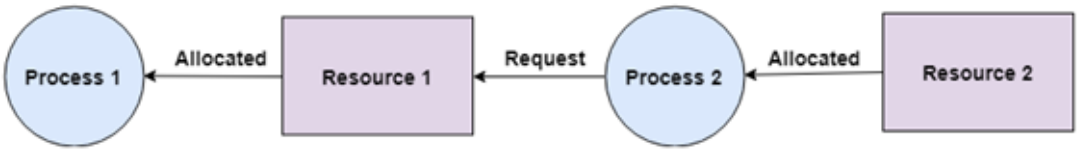
1) Mutual Exclusion- There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



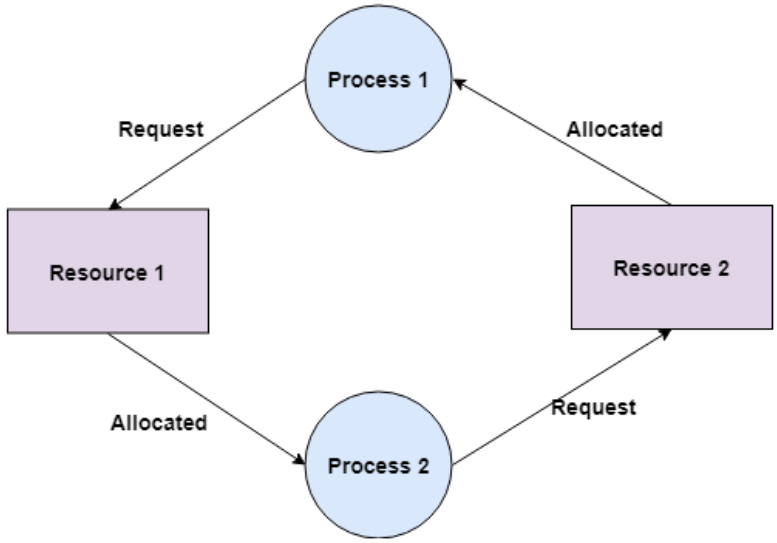
2)Hold and Wait- A process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



3)No Preemption- A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



4)Circular Wait- A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. For example: Process 1 is allocated Resource2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



Deadlock Detection

A deadlock can be detected by a resource scheduler as it keeps track of all the resources that are allocated to different processes. After a deadlock is detected, it can be resolved using the following methods ?

•All the processes that are involved in the deadlock are **terminated**. This is not a good approach as all the progress made by the processes is **destroyed**.

•Resources can be preempted from some processes and given to others till the deadlock is resolved.

Deadlock Avoidance Algorithms

•When resource categories have only single instances of their resources, **Resource- Allocation Graph** Algorithm is used. In this algorithm, a cycle is a necessary and sufficient condition for deadlock.

•When resource categories have multiple instances of their resources**, Banker's Algorithm** is used. In this algorithm, a cycle is a necessary but not a sufficient condition for deadlock.

1) Resource-Allocation Graph Algorithm

Resource Allocation Graph (RAG) is a popular technique used for deadlock avoidance. It is a directed graph that represents the processes in the system, the resources available, and the relationships between them. A process node in the RAG has two types of edges, request edges, and assignment edges. A request edge represents a request by a process for a resource, while an assignment edge represents the assignment of a resource to a process.

To determine whether the system is in a safe state or not, the RAG is analyzed to check for cycles. If there is a cycle in the graph, it means that the system is in an unsafe state, and granting a resource request can lead to a deadlock. In contrast, if there are no cycles in the graph, it means that the system is in a safe state, and resource allocation can proceed without causing a deadlock.

2) Banker's Algorithm

The Banker's algorithm assumes that each process declares its maximum resource requirements upfront. Based on this information, the algorithm allocates resources to each Resource-Allocation Graph process such that the total number of allocated resources never exceeds the total number of available resources. The algorithm does not grant access to resources that could potentially lead to a deadlock situation. The Banker's algorithm uses a matrix called the "allocation matrix" to keep track of the resources allocated to each process, and a "request matrix" to keep track of the resources requested by each process. It also uses a "need matrix" to represent the resources that each process still needs to complete its execution.

4. Memory Management
Memory management is responsible for allocating and deallocating memory to processes efficiently.

Virtual Memory:
•**Virtual memory** allows processes to access a larger address space than is physically available by using disk storage (e.g., paging).
•The OS manages memory as if the system has more RAM than it physically does.

Paging:
•**Paging** divides memory into fixed-size blocks called **pages**.
•The physical memory is divided into fixed-size blocks called **frames**.
•Pages are mapped to frames, and the mapping is stored in a **page table**.
•Paging eliminates the need for contiguous memory allocation and reduces fragmentation.

Segmentation:
•**Segmentation** divides memory into variable-sized segments, such as code, data, stack, etc.
•Unlike paging, segments are logical divisions of the program, and each segment can vary in size.
•Segmentation is more flexible but can lead to fragmentation.

5. File Organization
File organization refers to how files are stored and accessed in a file system.

File Types:
•**Regular Files**: These store user data (text, binary files, etc.).
•**Directory Files**: These store information about files and other directories, allowing hierarchical organization.

Blocking and Buffering:
•**Blocking** refers to storing multiple records in a single disk block. It can optimize storage space and reduce the overhead of reading individual records.
•**Buffering** involves storing data in memory temporarily before writing it to disk or after reading it from disk. It helps improve I/O efficiency by reducing the number of disk accesses.

File Descriptors:
•A **file descriptor** is an integer representing an open file in a program. It is used by the OS to manage file operations such as reading, writing, and closing files.
•File descriptors are obtained when a file is opened using system calls (e.g., open() in UNIX like systems)

6. File and Directory Structures
File systems organize data into files and directories:
File Structures:
•Files can have different internal structures, such as:
- **Sequential**: Data is stored in a linear sequence (e.g., text files).
- **Indexed**: Uses an index to directly access data blocks.
- **Hashed**: Uses a hash function to map data to storage locations.

Directory Structures:
•A **directory** is a container that holds information about files and other directories.
•Directories can be organized in different ways:
- **Single-Level Directory**: All files are in one directory.
- **Two-Level Directory**: Separate directories for each user.
- **Hierarchical Directory**: Directories are organized in a tree-like structure (common in modern file systems).

7. I/O Devices
I/O Devices are physical devices that allow a computer system to communicate with the external world. Examples include:

•**Input Devices**: Keyboard, mouse, microphone.
•**Output Devices**: Monitor, printer, speakers.

I/O Management:
•The OS handles input/output operations, ensuring that I/O devices are efficiently utilized.
•**I/O Scheduling** is necessary to determine the order in which requests for device access are serviced (e.g., disk scheduling algorithms like FCFS, SSTF, and SCAN).
•**Device Drivers** are software that allow the OS to communicate with hardware devices, abstracting the details of the hardware.

6) Computer Architecture

1. Boolean Algebra and Computer Arithmetic
•**Boolean Algebra**: This is a branch of algebra that operates on binary variables and logical operations like AND, OR, NOT, NAND, NOR, XOR, and XNOR. Boolean algebra is fundamental in the design and operation of digital circuits. It simplifies the design of circuits by using binary variables (0 or 1) to represent truth values (false or true).

Logical Operations in Boolean Algebra
The following are the fundamental logical operations that form the basis of Boolean algebra −

AND Operation
In Boolean algebra, a logical operation in which the outcome is true (1) only when all the input values are true (1), otherwise, the output is false (0) is termed as AND operation. The AND operation is represented by a dot (.). For example, A AND B can be represented as A.B in symbolic form.
OR Operation
In Boolean algebra, the OR operation is another logical operation in which the output is false (0) only when all input values are false (0), otherwise the output is true (1). The OR operations is denoted by a plus (+). For example, A OR B can be represented as A + B.
NOT Operation
In Boolean algebra, the NOT operation is performed to obtain the inverted version of the input value. Thus, the result of the NOT operation is false (0), if the input is true (1) and vice-versa. The NOT operation is represented by the symbol "~". For example, NOT A is represented as ~A.

These are the basic operations used in the Boolean algebra. However, there are many more logical operations and rules that used in the Boolean algebra to perform complex tasks.

Logic Gate
A logic gate is a digital circuit that can perform a specific logical operation. There are 7 main logic gates used in digital electronics, they are AND gate, OR gate, NOT gate, NOR gate, NAND gate, XOR gate, and XNOR gate.
These are the basic terms used in Boolean algebra and provides a foundation for its understanding and working.

Laws of Boolean Algebra

All the important laws and rules of Boolean algebra are explained below −

Rules of Logical Operations

There are three basic logical operations namely, AND, OR, and NOT. The following table highlights the rules associated with these three logical operations −

AND Operation	OR Operation	NOT Operation
0 AND 0 = 0	0 OR 0 = 0	NOT of 0 = 1
0 AND 1 = 0	0 OR 1 = 1	NOT of 1 = 0
1 AND 0 = 0	1 OR 0 = 1	
1 AND 1 = 1	1 OR 1 = 1	

Commutative Laws

There are following two commutative laws in Boolean algebra −

Law 1 − According to this law, the operation A OR B produces the same output as the operation B OR A, i.e.,

$$A + B = B + A$$

Hence, the order of the variables does not affect the OR operation.

This law can be extended to any number of variables. For example, for three variables, it will be,

$$\mathbf{A + B + C = C + B + A = B + C + A = C + A + B}$$

Law 2 − According to this law, the output of the A AND B operation is same as that of the B AND A operation, i.e.,

$$A \cdot B = B \cdot A$$

This law states that the order in which the variables are ANDed does not affect the result.

We can extend this law to any number of variables. For example, for three variables, we get,

$$A \cdot B \cdot C = A \cdot C \cdot B = C \cdot B \cdot A = C \cdot A \cdot B$$

Associative Laws

Associative laws define the ways of grouping the variables. There are two associative laws as described below.

Law 1 − The expression A OR B ORed with C results the same as the A Ored with B OR C, i.e.,

$$(A + B) + C = A + (B + C)$$

This law can be extended to any number of variables. For example, for 4 variables, we get,

$$(A + B + C) + D = A + (B + C + D) = (A + B) + (C + D)$$

Law 2 − The expression A AND B ANDed with C results the same as the expression A ANDed with B AND C, i.e.,

$$(A \cdot B) \cdot C = A \cdot (B \cdot C)$$

We can extend this law to any number of variables. For example, if we have 4 variables, then

$$(ABC)D = A(BCD) = (AB)\cdot(CD)$$

Distributive Laws

In Boolean algebra, there are the following two distributive laws that allow for multiplying or factoring out of expressions.

Law 1 − According to this law, we OR several variables and then AND the result with a single variable. It gives the same result as the expression in which the single variable is ANDed with each of the several variables and then ORed the product terms, i.e.,

$$A \cdot (B + C) = AB + AC$$

We can extend this law to any number of variables. For example,

$$A(BC + DE) = ABC + ADE$$

$$AB(CD + EF) = ABCD + ABEF$$

Law 2 − According to this law, if we AND several variables and then the result is ORed with a single variable. It gives the same result as we OR the single variable with each of the several variables and then the sum terms are ANDed together, i.e.,

$$A + BC = (A + B)(A + C)$$

Idempotence Laws

The term "idempotence" is a synonym for "same value". There are two idempotence laws in Boolean algebra. They are,

Law 1 − According to this law, ANDing a variable with itself is equal to the variable, i.e.,

$$A \cdot A = A$$

Law 2 − According to this law, ORing a variable with itself is equal to the variable, i.e.,

$$A + A = A$$

Absorption Laws

There are two absorption laws in Boolean algebra and they are explained below.

Law 1 − According to this law, if we OR a variable with the AND of the that variable and another variable, then it is equal to the variable itself, i.e.,

$$A + A \cdot B = A$$

This can be proved as follows,

$$LHS = A + A \cdot B = A \cdot (1 + B)$$
$$= A \cdot 1 = A = RHS$$

Law 2 − According to this law, the AND of a variable with the OR of that variable and another variable is equivalent to the variable itself i.e.,

$$A(A + B) = A$$

This can also be proved as follows,

$$LHS = A(A + B) = AA + AB$$
$$= A + AB = A(1 + B) = A \cdot 1 = A = RHS$$

Hence, this law proves that if a term appears in another term, then the latter term will become redundant and can be removed from the expression.

Computer Arithmetic: Refers to the methods used by computers to perform arithmetic operations like addition, subtraction, multiplication, and division. This is typically done using binary representation of numbers. For example:

•**Addition and Subtraction**: Computers use **binary addition** and **two's complement representation** to perform subtraction.
•**Multiplication and Division**: Done using algorithms like **shift-and-add** or **Booth's algorithm**.
•**Floating Point Arithmetic**: Involves representing numbers with fractional parts (e.g., IEEE 754 standard)

2) Flip-Flops

•A **flip-flop** is a basic digital storage element used to store one bit of information (0 or 1). They are used in sequential circuits, and they have two stable states (representing binary 0 or 1).
•Types of flip-flops include:

- **SR (Set-Reset) Flip-Flop**
- **D (Data) Flip-Flop**
- **T (Toggle) Flip-Flop**
- **JK Flip-Flop**

•Flip-flops are used in building registers, memory cells, counters, and other sequential circuits.

3. Design of Combinational and Sequential Circuits

•**Combinational Circuits**: The output of these circuits is purely dependent on the current input. Examples include **adders, multiplexers, decoders, and encoders**. The output changes immediately when the input changes.
•**Sequential Circuits**: These circuits depend not only on the current input but also on the previous state (i.e., they have memory). Examples include **flip-flops, registers, counters, and shift registers**. Sequential circuits are typically used for tasks that require storing past information (like timing or sequencing).

COMBINATIONAL CIRCUITS

What is an **Encoder**?

An **encoder** is a digital combinational circuit that converts a human friendly information into a coded format for processing using machines. In simple words, an encoder converts a piece of information normal form to coded form. This process is called encoding.

Encoders are crucial components in various digital electronics applications such as data transmission, controlling and automation, communication, signal processing, etc.

An encoder consists of a certain number of input and output lines. Where, an encoder can have maximum of "2^n" input lines whereas "n" output lines. Hence, an encoder encodes information represented by "2^n" input lines with "n" bits.

What is a **Decoder**?

In digital electronics, a combinational logic circuit that converts an N-bit binary input code into M output channels in such a way that only one output channel is activated for each one of the possible combinations of inputs is known as a **decoder**.

In other words, a combinational logic circuit which converts N input lines into a maximum of 2_N output lines is called a **decoder**.

Therefore, a decoder is a combination logic circuit that is capable of identifying or detecting a particular code. The operation that a decoder performs is referred to as decoding. A general block diagram of a decoder is shown in Figure-1.

Here, the decoder has N input lines and M (2^N) output lines. In a decoder, each of the N input lines can be a 0 or a 1, hence the number of possible input combinations or codes be equal to 2^N. For each of these input combinations, only one of the M output lines will be active, and all other output lines will remain inactive.

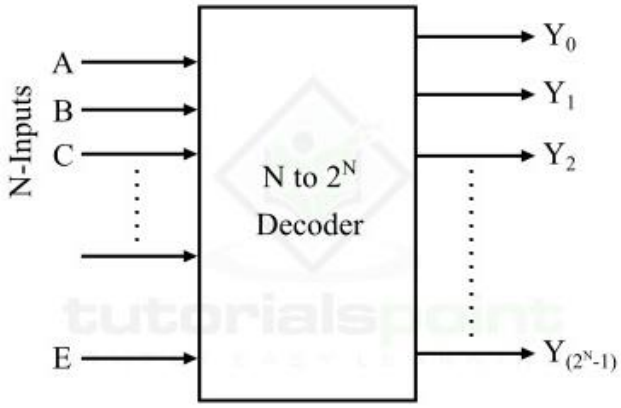


Figure 1 - Decoder

What is a **Multiplexer**?

As already mentioned, a **multiplexer**, also referred to as **MUX**, is a combination logic circuit that is designed to accept multiple input signals and transfer only one of them through the output line. In simple words, a multiplexer is a digital logic device that selects one-out-of-N (N = 2^n) input data sources and transmits the selected data to a single output line.

The multiplexer is also called **data selector** as it selects one from several. The block diagram of a typical **2^n:1** multiplexer is shown in Figure 1.

In the case of multiplexer, the selection of desired data input to flow through the output line is controlled with the help of **SELECT lines**. In the block diagram of mux in Figure 1, I_0, I_1,... I_{n-1}, i.e., (2^n) are the input lines, and **"n"** be the select lines. These select lines will determine which input is to be routed to the output.

Hence, the multiplexer works as a multi-position switch whose operation is controlled by digital signals. These digital control signals are applied to the select lines to determine which data input will be switched to the output line.

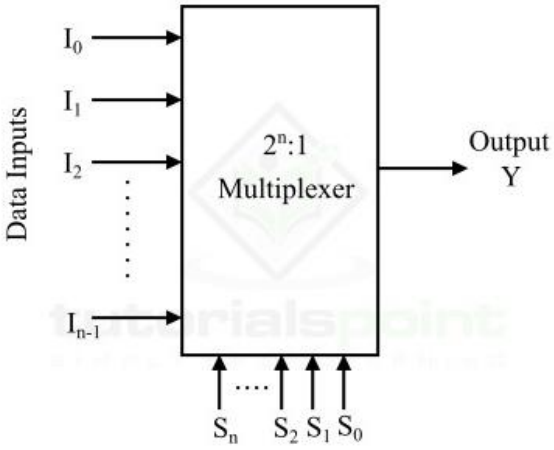


Figure 1 - Digital Multiplexer

What is a **Demultiplexer**?

A **Demultiplexer** is a combinational logic circuit that accepts a single input and distributes it over several output lines. Demultiplexer is also termed as **DEMUX** in short. As Demultiplexer is used to transmit the same data to different destinations, hence it is also known as **data distributor**.

There is another combinational logic circuit named multiplexer which performs opposite operation of the Demultiplexer, i.e. accepts several inputs and transmits one of them at time to the output line.

From the definition, we can state that a Demultiplexer is a 1-to-2^n device. The functional block diagram of a typical 1×2^n Demultiplexer is shown in Figure-1.

It can be seen that the Demultiplexer has only one data input line, 2^n output lines, and n select lines. The logic level applied to select lines of the Demultiplexer determines the output channel to which the input data will be transmitted.

Demultiplexer circuit are the combinational logic circuit widely used in digital decoders and Boolean function generator circuits.

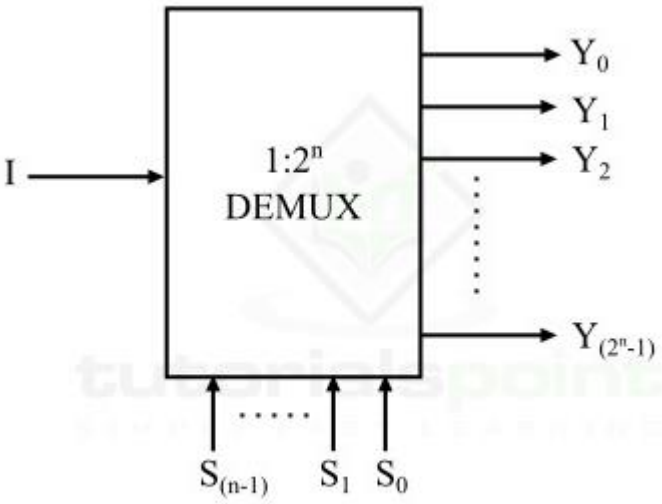


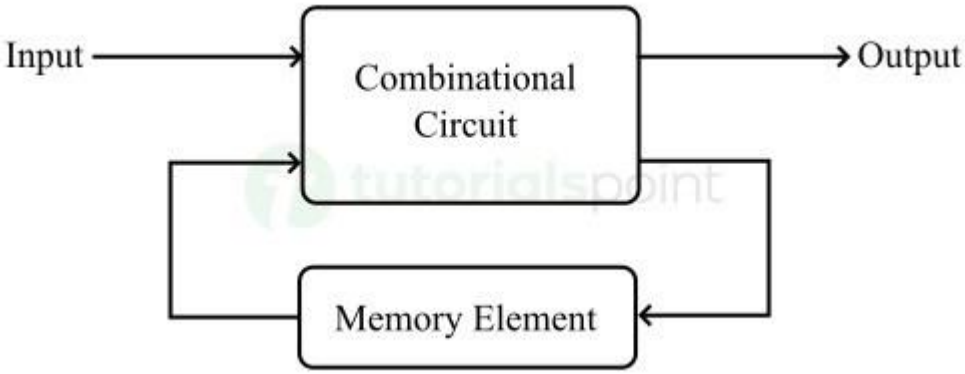
Figure 1 - Demultiplexer

SEQUENTIAL CIRCUITS

What is a Sequential Circuit?

A **sequential circuit** is a logic circuit that consists of a memory element to store history of past operation of the circuit. Therefore, the output of a sequential circuit depends on present inputs as well as past outputs of the circuit.

The **block diagram of a typical sequential circuit** is shown in the following figure −



The sequential circuits are named so because they use a series of latest and previous inputs to determine the new output.

Main Components of Sequential Circuit

A sequential circuit consists of several different digital components to process and hold information in the system. Here are some key components of a sequential circuit explained −

Logic Gates

The logic gates like AND, OR, NOT, etc. are used to implement the data processing mechanism of the sequential circuits. These logic gates are basically interconnected in a specific manner to implement combinational circuits to perform logical operations on input data.

Memory Element

In sequential circuits, the memory element is another crucial component that holds history of circuit operation. Generally, flip-flops are used as the memory element in sequential circuits.

In sequential circuits, a feedback path is provided between the output and the input that transfers information from output end to the memory element and from memory element to the input end.

All these components are interconnected together to design a sequential circuit that can perform complex operations and store state information in the memory element.

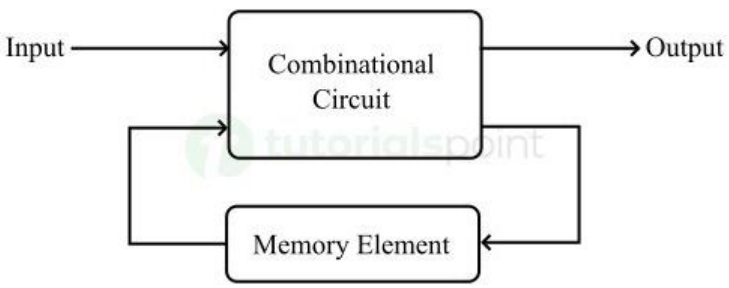
Types of Sequential Circuits

Based on structure, operation, and applications, the sequential circuits are classified into the following two types −

- Asynchronous Sequential Circuit
- Synchronous Sequential Circuit

Asynchronous Sequential Circuit

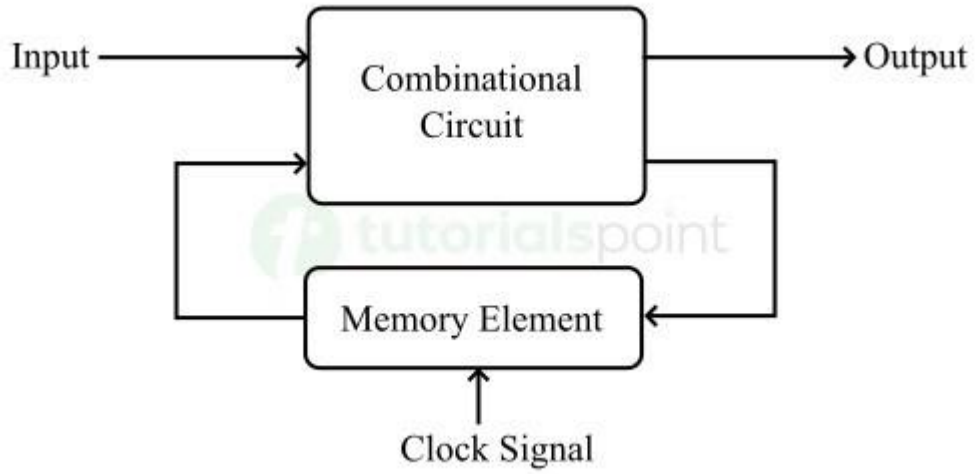
A type of sequential circuit whose operation does not depend on the clock signals is known as an **asynchronous sequential circuit**. This type of sequential circuits operates using the input pulses that means their state changes with the change in the input pulses.



The main components of the asynchronous sequential circuits include un-clocked flip flops and combinational logic circuits. The **block diagram of a typical asynchronous sequential circuit** is shown in the following figure.

From this diagram, it is clear that an asynchronous sequential circuit is similar to a combinational logic circuit with a feedback mechanism.
Asynchronous sequential circuits are mainly used in applications where the clock signals are not available or practical to use. For example, in conditions when speed of the task execution is important.
Asynchronous sequential circuits are relatively difficult to design and sometimes they produce uncertain output.
The **ripple counter** is a common example of asynchronous sequential circuit.

Synchronous Sequential Circuit
A synchronous sequential circuit is a type of sequential circuit in which all the memory elements are synchronized by a common clock signal. Hence, synchronous sequential circuits take a clock signal along with input signals.
In synchronous sequential circuits, the duration of the output pulse is equivalent to the duration of the clock pulse applied. Take a look at the **block diagram of a typical synchronous sequential circuit** −



In this figure, it can be seen that the memory element of the sequential circuit is synchronized by a clock signal.
The major disadvantage of the synchronous sequential circuits is that their operation is quite slow. This is because, every time the circuit has to wait for a clock pulse for the operation to take place. However, the most significant advantage of synchronous sequential circuits is that they have a reliable and predictable operation.
Some common examples of synchronous sequential circuits include **counters, registers, memory units, control units, etc.**

What is a **Latch?**
In digital electronics, a **latch** is an asynchronous sequential circuit that can store 1-bit information. It is used as the fundamental memory element in digital circuits.
A latch can have two stable states namely, **set** and **reset**. The set state is denoted by the logic 1 and the reset state is represented by the logic 0. Due to these two stable states, a latch is also known as a **bistable-multivibrator**. The state of a latch toggles according to the applied input.
The most important thing to be noted about latches is that they do not have a clock signal for synchronization. That is why they are called asynchronous sequential circuits.
The logic gates are the fundamental building blocks of latches. Since there is no synchronization and clock signal used. Hence, the latches operate immediately on the application of input signals.

Characteristics of Latches
Some key characteristics of latches are explained below −
•Latches can store 1-bit of digital information that can be represented using either logic 0 or logic 1. Thus, the latches are mainly used as memory elements in digital circuits.
•Latches have a feedback mechanism that allows them to maintain their current state as it is until the next input is applied.
•The operation of latches is completely controlled by applied inputs that means the output of the latches updates based on the change in the input signals.

FLIP-FLOP

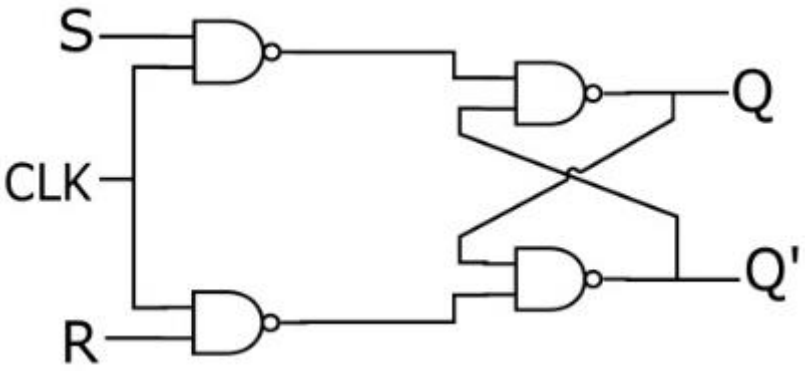
A **flip-flop** is a sequential digital electronic circuit having two stable states that can be used to store one bit of binary data. Flip-flops are the fundamental building blocks of all memory devices.

Types of Flip-Flops

•S-R Flip-Flop
•J-K Flip-Flop
•D Flip-Flop
•T Flip-Flop

S-R Flip-Flop

This is the simplest flip-flop circuit. It has a set input (S) and a reset input (R). When in this circuit when S is set as active, the output Q would be high and the Q' will be low. If R is set to active then the output Q is low and the Q' is high. Once the outputs are established, the results of the circuit are maintained until S or R get changed, or the power is turned off.



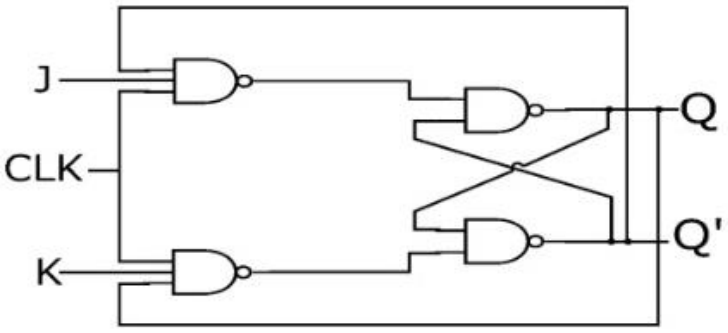
Truth Table of S-R Flip-Flop

S	R	Q	State
0	0	0	No Change
0	1	0	Reset
1	0	1	Set
1	1	X	

J-K Flip-Flop

Because of the invalid state corresponding to S=R=1 in the SR flip-flop, there is a need of another flip-flop. The JK flip-flop operates with only positive or negative clock transitions. The operation of the JK flip-flop is similar to the SR flip-flop. When the input J and K are different then the output Q takes the value of J at the next clock edge.

When J and K both are low then NO change occurs at the output. If both J and K are high, then at the clock edge, the output will toggle from one state to the other.



Truth Table of JK Flip-Flop

J	K	Q	State
0	0	0	No Change
0	1	0	Reset
1	0	1	Set
1	1	Toggles	Toggle

4. Instruction Formats

•Instruction format refers to the structure or layout of a machine instruction. It specifies how the instruction is broken down into different fields such as:

- **Opcode** (operation code): Specifies the operation to be performed (e.g., ADD, SUB).
- **Operand(s)**: Specifies the data or memory location on which the operation is to be performed.
- **Addressing mode**: Specifies how the operands are accessed or interpreted.

•There are different formats:

- **Fixed-length instruction**: All instructions have the same size.
- **Variable-length instruction**: Instructions can vary in size.

5. Addressing Modes

•Addressing modes determine how the operands of an instruction are accessed. Some common addressing modes are:

- **Immediate Addressing**: The operand is specified directly within the instruction.
- **Direct Addressing**: The memory address of the operand is specified directly in the instruction.
- **Indirect Addressing**: The instruction contains the address of a memory location that holds the address of the operand.
- **Register Addressing**: The operand is a register (the instruction specifies the register).
- **Indexed Addressing**: The effective address of the operand is calculated by adding an index value to a base address.

6. Interfacing Peripheral Devices

•Interfacing involves connecting peripheral devices like keyboards, mice, monitors, printers, etc., to the CPU and ensuring data transfer. The process involves:

- **Input/Output (I/O) ports**: Used for transferring data between peripherals and the CPU.
- **Direct Memory Access (DMA)**: Allows peripherals to access memory directly, bypassing the CPU to improve data transfer efficiency.
- **Interrupts**: Mechanism for peripherals to request CPU attention when needed.

7. Types of Memory and Their Organization

•**Primary Memory**:

- **RAM (Random Access Memory)**: Volatile memory used for temporary storage of data and instructions that are currently being executed.
- **ROM (Read-Only Memory)**: Non-volatile memory used to store firmware and system boot-up instructions.

•**Secondary Memory**: Non-volatile memory for long-term storage (e.g., hard drives, SSDs).

•**Cache Memory**: A small, fast memory located close to the CPU to store frequently accessed data, reducing the time needed to fetch data from main memory.

•**Virtual Memory**: An abstraction that allows programs to use more memory than what is physically available by using disk space as "virtual" RAM.

8. Interrupts and Exceptions

•**Interrupts** are signals to the processor that temporarily halt the current process to allow the CPU to address an event (e.g., I/O request).

- **Hardware Interrupts**: Triggered by external devices (e.g., keyboard press).
- **Software Interrupts**: Triggered by a software program (e.g., system calls).

•**Exceptions** are disruptions caused by errors in the current instruction or program, like **division by zero**, **invalid memory access**, or **invalid opcodes**.

•**Interrupt Vector Table (IVT)**: A table that holds addresses for interrupt service routines (ISRs).

Von-Neumann Model

Von-Neumann proposed his computer architecture design in 1945 which was later known as Von-Neumann Architecture. It consisted of a Control Unit, Arithmetic, and Logical Memory Unit (ALU), Registers and Inputs/Outputs.

Von Neumann architecture is based on the stored-program computer concept, where instruction data and program data are stored in the same memory. This design is still used in most computers produced today.

Von-Neumann Basic Structure:

A Von Neumann-based computer:

•Uses a single processor

•Uses one memory for both instructions and data.

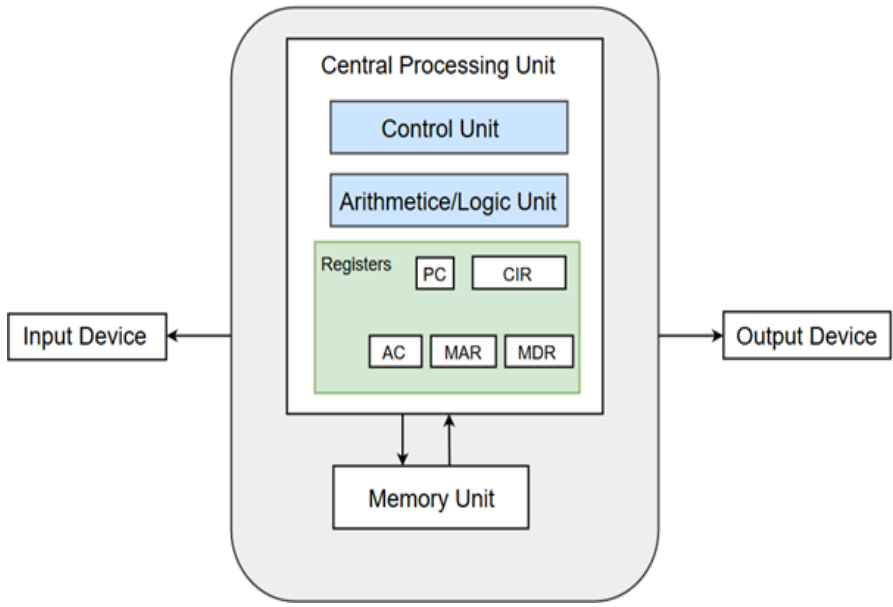
•Executes programs following the fetch-decode-execute cycle

Components of Von-Neumann Model:

•Central Processing Unit

•Buses

•Memory Unit



Central Processing Unit

The part of the Computer that performs the bulk of data processing operations is called the Central Processing Unit and is referred to as the CPU.

The Central Processing Unit can also be defined as an electric circuit responsible for executing the instructions of a computer program.

The CPU performs a variety of functions dictated by the type of instructions that are incorporated in the computer.

The major components of CPU are Arithmetic and Logic Unit (ALU), Control Unit (CU) and a variety of registers.

Arithmetic and Logic Unit (ALU)

The Arithmetic and Logic Unit (ALU) performs the required micro-operations for executing the instructions. In simple words, ALU allows arithmetic (add, subtract, etc.) and logic (AND, OR, NOT, etc.) operations to be carried out.

Control Unit

The Control Unit of a computer system controls the operations of components like ALU, memory and input/output devices.

The Control Unit consists of a program counter that contains the address of the instructions to be fetched and an instruction register into which instructions are fetched from memory for execution.

Registers

Registers refer to high-speed storage areas in the CPU. The data processed by the CPU are fetched from the registers.

Following is the list of registers that plays a crucial role in data processing.

Registers	Description
MAR (Memory Address Register)	This register holds the memory location of the data that needs to be accessed.
MDR (Memory Data Register)	This register holds the data that is being transferred to or from memory.
AC (Accumulator)	This register holds the intermediate arithmetic and logic results.
PC (Program Counter)	This register contains the address of the next instruction to be executed.
CIR (Current Instruction Register)	This register contains the current instruction during processing.

Buses

Buses are the means by which information is shared between the registers in a multiple-register configuration system.

A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer.

Von-Neumann Architecture comprised of three major bus systems for data transfer.

Bus	Description
Address Bus	Address Bus carries the address of data (but not the data) between the processor and the memory.
Data Bus	Data Bus carries data between the processor, the memory unit and the input/output devices.
Control Bus	Control Bus carries signals/commands from the CPU.

Memory Unit

A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of the storage. The memory stores binary information in groups of bits called words. The internal structure of a memory unit is specified by the number of words it contains and the number of bits in each word.

Two major types of memories are used in computer systems:

1.RAM (Random Access Memory)

2.ROM (Read-Only Memory)

10. System Bus

•The **system bus** is a collection of pathways used for communication between the CPU, memory, and peripherals. It typically consists of:

- **Data Bus**: Carries data between components.
- **Address Bus**: Carries addresses used to locate data in memory.
- **Control Bus**: Carries control signals for managing operations between components (e.g., read/write signals).

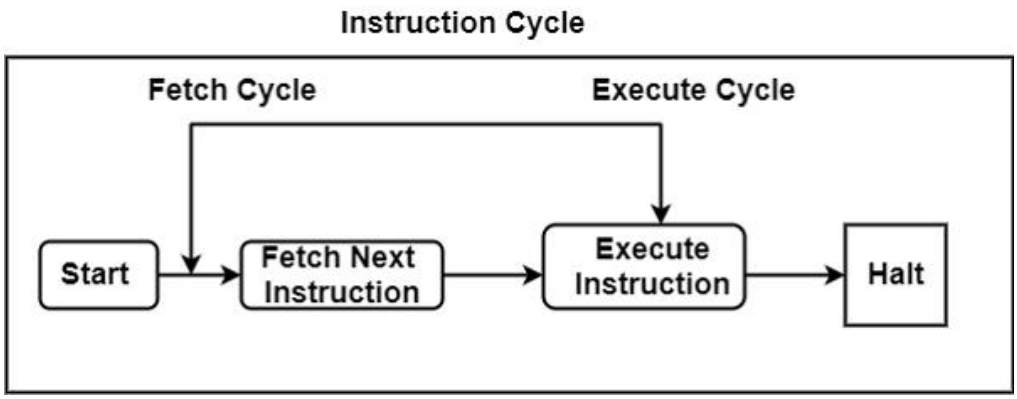
Instruction Cycle

A program consisting of the memory unit of the computer includes a series of instructions. The program is implemented on the computer by going through a cycle for each instruction.

In the basic computer, each instruction cycle includes the following procedures −

•It can fetch instruction from memory.
•It is used to decode the instruction.
•It can read the effective address from memory if the instruction has an indirect address.
•It can execute the instruction.

After the following four procedures are done, the control switches back to the first step and repeats the similar process for the next instruction. Therefore, the cycle continues until a **Halt** condition is met. The figure shows the phases contained in the instruction cycle.



As display in the figure, the halt condition appears when the device receive turned off, on the circumstance of unrecoverable errors, etc.

Fetch Cycle

The address instruction to be implemented is held at the program counter. The processor fetches the instruction from the memory that is pointed by the PC.

Next, the PC is incremented to display the address of the next instruction. This instruction is loaded onto the instruction register. The processor reads the instruction and executes the important procedures.

Execute Cycle

The data transfer for implementation takes place in two methods are as follows −

•**Processor-memory** − The data sent from the processor to memory or from memory to processor.

•**Processor-Input/Output** − The data can be transferred to or from a peripheral device by the transfer between a processor and an I/O device.

In the execute cycle, the processor implements the important operations on the information, and consistently the control calls for the modification in the sequence of data implementation. These two methods associate and complete the execute cycle.

Data Representation

Number System

A computer system considers numbers as data; it includes integers, decimals, and complex numbers. All the inputted numbers are represented in binary formats like 0 and 1. A number system is categorized into four types −

•**Binary** − A binary number system is a base of all the numbers considered for data representation in the digital system. A binary number system consists of only two values, either 0 or 1; so its base is 2. It can be represented to the external world as $(10110010)_2$. A computer system uses binary digits (0s and 1s) to represent data internally.

•**Octal** − The octal number system represents values in 8 digits. It consists of digits 0,12,3,4,5,6, and 7; so its base is 8. It can be represented to the external world as $(324017)_8$.

•**Decimal** − Decimal number system represents values in 10 digits. It consists of digits 0, 12, 3, 4, 5, 6, 7, 8, and 9; so its base is 10. It can be represented to the external world as $(875629)_{10}$.
•**Hexadecimal number** − Hexadecimal number system represents values in 16 digits. It consists of digits 0, 12, 3, 4, 5, 6, 7, 8, and 9 then it includes alphabets A, B, C, D, E, and F; so its base is 16. Where A represents 10, B represents 11, C represents 12, D represents 13, E represents 14 and F represents 15.The below-mentioned table below summarizes the data representation of the number system along with their Base and digits.

Number System

System	Base	Digits
Binary	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

Bits and Bytes

Bits

A bit is the smallest data unit that a computer uses in computation; all the computation tasks done by the computer systems are based on bits. A bit represents a binary digit in terms of 0 or 1. The computer usually uses bits in groups. It's the basic unit of information storage and communication in digital computing.

Bytes

A group of eight bits is called a byte. Half of a byte is called a nibble; it means a group of four bits is called a nibble. A byte is a fundamental addressable unit of computer memory and storage. It can represent a single character, such as a letter, number, or symbol using encoding methods such as ASCII and Unicode.
Bytes are used to determine file sizes, storage capacity, and available memory space. A kilobyte (KB) is equal to 1,024 bytes, a megabyte (MB) is equal to 1,024 KB, and a gigabyte (GB) is equal to 1,024 MB. File size is roughly measured in KBs and availability of memory space in MBs and GBs.

Byte Value	Bit Value
1 Byte	8 Bits
1024 Bytes	1 Kilobyte
1024 Kilobytes	1 Megabyte
1024 Megabytes	1 Gigabyte
1024 Gigabytes	1 Terabyte
1024 Terabytes	1 Petabyte
1024 Petabytes	1 Exabyte
1024 Exabytes	1 Zettabyte

Text Code

A Text Code is a static code that allows a user to insert text that others will view when they scan it. It includes alphabets, punctuation marks and other symbols. Some of the most commonly used text code systems are −
•EBCDIC
•ASCII
•Extended ASCII
•Unicode

•**Binary Representation**: Data is represented as binary numbers (0s and 1s).
•**Character Representation**: Characters are represented using encoding schemes like **ASCII** or **Unicode**.
•**Integer Representation**: Typically represented using **signed** or **unsigned** binary numbers, and methods like **two's complement** for signed integers.
•**Floating-Point Representation**: Represents real numbers, often using the **IEEE 754** standard.

Machine Instructions -
Machine instructions are the binary code (0s and 1s) that a computer's CPU directly understands and executes, while assembly language is a human-readable, symbolic representation of these instructions, using mnemonics and labels, that an assembler translates into machine code.

Machine Instructions (Machine Code):
•**Definition:** Machine instructions are the fundamental commands a computer's CPU (Central Processing Unit) can directly execute.
•**Format:** They are represented as binary code (sequences of 0s and 1s).
•**Execution:** The CPU reads these instructions from memory and performs the corresponding actions.
•**Example:** 10101100 00010001 might represent an instruction to add two numbers.
•**Level:** Machine code is the lowest-level programming language, directly interacting with the hardware

Assembly Language -
•**Definition:**
•Assembly language is a low-level programming language that uses symbolic codes (mnemonics) to represent machine language instructions.

•**Mnemonics:**
Instead of using binary codes, assembly language uses mnemonics (short, easy-to-remember words) like **MOV (move), ADD (add), SUB (subtract), JMP (jump)**, etc.
•**Assembler:**
Assembly language code is converted into machine code by a program called an assembler.
•**Example:**
MOV AX, 10 (move the value 10 into the AX register) is an assembly language instruction that an assembler would translate to a machine instruction.
•**Level:**
Assembly language is a level above machine language, offering a more human-readable way to interact with the hardware.