

T. Jones Daniel, J. Bercy Miraclin, and Kiruba V

# C Program and IOT - Live Exercise



# C Program and IOT - Live Exercise

**T. Jones Daniel**

Department of CSE-IoT at CITY Engineering College,  
Bengaluru, India

**J. Bercy Miraclin**

Department of Robotics and Automation at Rajalakshmi  
Engineering College, Chennai, Tamil Nadu

**Kiruba V**

Computer Science and Engineering at Jayaraj Annapackiam  
CSI College of Engineering, Nazareth, Thoothukudi, Tamil  
Nadu, India



*Published, marketed, and distributed by:*

Deep Science Publishing, 2025  
USA | UK | India | Turkey  
Reg. No. MH-33-0523625  
www.deepscienceresearch.com  
editor@deepscienceresearch.com  
WhatsApp: +91 7977171947

ISBN: 978-93-7185-205-0

E-ISBN: 978-93-7185-249-4

<https://doi.org/10.70593/978-93-7185-249-4>

Copyright © T. Jones Daniel, J. Bercy Miraclin, Kiruba V, 2025.

**Citation:** Daniel, T. J., Miraclin, J. B., & Kiruba, V. (2025). *C Program and IOT - Live Exercise*. Deep Science Publishing. <https://doi.org/10.70593/978-93-7185-249-4>

This book is published online under a fully open access program and is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License (CC BY-NC 4.0). This open access license allows third parties to copy and redistribute the material in any medium or format, provided that proper attribution is given to the author(s) and the published source. The publishers, authors, and editors are not responsible for errors or omissions, or for any consequences arising from the application of the information presented in this book, and make no warranty, express or implied, regarding the content of this publication. Although the publisher, authors, and editors have made every effort to ensure that the content is not misleading or false, they do not represent or warrant that the information-particularly regarding verification by third parties-has been verified. The publisher is neutral with regard to jurisdictional claims in published maps and institutional affiliations. The authors and publishers have made every effort to contact all copyright holders of the material reproduced in this publication and apologize to anyone we may have been unable to reach. If any copyright material has not been acknowledged, please write to us so we can correct it in a future reprint.

# Preface

This book invites you on a journey into the world of technology — a journey created especially for higher secondary students who believe that education is both a source of power and a means to promote equality. In today's era, where technology drives our future, learning the fundamentals of C Programming and IoT with practical experiments for the IoT, Information Technology, Computer Science, and AI & ML departments is not just an advantage — it is essential. This book is designed to provide that knowledge, opening pathways to innovation and new opportunities for all learners.

Emphasis is placed on practical learning throughout this book. Built on the belief that real understanding comes from doing rather than only reading, it offers clear explanations along with hands-on examples. These examples are intended to simplify complex ideas, making them easier to grasp while encouraging interactive and meaningful learning.

Whether your goal is to design intelligent systems, develop robotic solutions, or simply better understand the digital world, this book will serve as a strong foundation. Through these pages, we hope readers will not only acquire valuable skills but also ignite their curiosity and passion for technology. Ultimately, the aim is to empower students to contribute to a more inclusive, innovative, and technologically advanced future for everyone.

T. Jones Daniel  
J. Bercy Miraclin  
Kiruba V

# Biography

## Author 1:



**Mr. T. Jones Daniel**, M.Th, M.Tech, (Ph.D.), (9.5years experience) is currently serving as an Assistant Professor in the Department of CSE-IoT at CITY Engineering College, Bengaluru. He is also pursuing his Ph.D. in the AI & ML discipline at Kalasalingam Academy of Research and Education (NAAC A++ Grade, NBA, nirf ). Beyond academia, he holds the role of Secretary at Light Social Welfare Trust, an organization committed to supporting destitute elderly individuals and those suffering from mental illness.

His research focuses on leveraging Artificial Intelligence and Machine Learning to extend the lifespan and enhance the quality of life for bedridden elderly individuals.

Mr. Jones journey began during his final year of B.Tech, when he was selected through campus recruitment as a website designer and contributed to projects via the “Design Crowd” platform, collaborating with clients in Australia, the Philippines, and the United States. However, in 2013, he made a pivotal career shift, enrolling in an M.Tech program with a vision of mentoring students and shaping young minds.

A life-altering experience occurred during his final year of M.Tech at Trichy Railway Station, where he witnessed an elderly man attempting suicide. The older man was rescued—though severely injured—after bystanders, including Mr. Jones, stopped the train. The older man’s quiet despair left a profound impact on Mr. Jones, inspiring him to dedicate his life to serving the elderly, mentally ill, physically challenged, visually impaired, and other disabled people.

He strongly believes that “Education imparts essential life skills such as problem-solving and logical thinking, which are crucial for personal growth and development, and it has the power to break the cycle of poverty.” In alignment with this philosophy, he has helped over 26 students from underprivileged urban and rural areas secure employment opportunities since 2016.

Mr. Jones has authored and published numerous research papers and launched Deep Learning book, AI ML & DL book, Applied Industrial Automation with Delta PLC, Data Structure & IOT for students. After seven years of dedicated research, he recently

published a scopus indexed noteworthy paper titled “Optimized AI Bed for Maim and Bedridden Elder Person Based on Mobile Application” (Springer Link below). He presented this paper at an international conference in Singapore (organized by Springer Nature), through online while his mother underwent surgery at CMC Hospital. His work earned him the Best Paper Award.

In addition, he developed a mobile application titled “Deaf, Dumb, and Maim Assistant Application”, published on the Google Play Store. This application, created in collaboration with his former students Er. Munish and Er. Mahendran, has been widely used by people from various countries around the world, free of charge.

As of 2025, the Light Social Welfare Trust shelters and cares for 158 abandoned individuals. With over nine and half years of teaching experience and five years as a Research & Development Coordinator, Mr. Jones continues to guide students in innovative research endeavors. He also runs a free educational website:

FSWD – PYTHON – C PROGRAM <https://jodaeducation.blogspot.com>.

### **Various links:**

YouTube:

<https://youtube.com/@lightsocialwelfaretrust5522?si=IMsPs4NYG98WTml5> Research Old Age people:

[https://link.springer.com/chapter/10.1007/978-981-97-5862-3\\_5](https://link.springer.com/chapter/10.1007/978-981-97-5862-3_5) Education Web:

<https://lightsocialwelfare.blogspot.com>

App:<https://play.google.com/store/apps/details?id=com.lightoldagehomeapplication.lig>  
htoldagehome

### **Author 2:**



**Ms. J. Bercy Miraclin**, is currently pursuing her Bachelor of Engineering in the Department of Robotics and Automation at Rajalakshmi Engineering College. Like many students entering a technical field, she faced early challenges—academic pressure, self-doubt, and uncertainty about her future. At times, she questioned whether she had made the right choice.

She holds a quiet belief that “In the moments we doubt our path, every step is still preparing us for a greater purpose.” This belief kept her moving forward even when things were unclear.

Everything changed when a mentor encouraged her and recognized her potential. That moment gave her clarity, confidence, and a deep interest in her field. With a stronger mindset, she moved forward with focus and determination.

Slowly, her struggles became her strength. Her consistent effort and positive attitude led her to a major personal milestone: becoming the author of her book. This achievement reflects her determination and growth, marked by the release of "Applied Industrial Automation with Delta PLC Programming, Ladder Logic, and Control Systems." (ISBN: 978-93-7185-160-2). This achievement stands as a symbol of her growth, resilience, and commitment.

This success would not have been possible without the unwavering support of her parents. Their emotional and financial strength helped her move through difficult times. Even when she doubted herself, they stood by her with unconditional belief.

Her book is not just a personal victory but also a heartfelt tribute to her parents' sacrifices and support. Bercy's journey is a reminder that with faith, persistence, and the right people beside you, anything is possible.

### Author 3:



#### **Ms. Kiruba V – A Journey of Resilience and Purpose**

Ms. Kiruba is a determined and inspiring student of Computer Science and Engineering at Jayaraj Annapackiam CSI College of Engineering, an institution affiliated with Anna University, located in Nazareth, Thoothukudi, Tamil Nadu.

Raised without the support of parents from a very young age, Kiruba's life began with significant challenges. She completed her schooling in the Tamil

medium, facing not just academic hurdles but also the emotional weight of societal criticisms and hardships. Rather than allowing these obstacles to break her spirit, she transformed them into stepping stones, building a path toward a brighter future with unwavering determination.

Kiruba firmly believes that education is the key to changing one's destiny. Her passion for learning and self-betterment reflects her inner strength and vision. Inspired by the selfless nature of the banyan tree, she aspires to grow strong and wide in her capacity to help others—offering shade, support, and encouragement to those in need.

Her story stands as a powerful testimony to the fact that with resilience, belief, and the right purpose, even the most difficult beginnings can lead to a meaningful and impactful life.



# Table of Contents

MODULE 1 .....1

MODULE 2 .....34

MODULE 3 .....52

MODULE 4 .....67

MODULE 5 .....82

MODULE 6 .....100

MODULE 7 .....129

## MODULE 1

### **Introduction To Data Structures:**

Data Structures, Classifications (Primitive & Non-Primitive), Data structure Operations Review of pointers and dynamic Memory Allocation

**ARRAYS And STRUCTURES:** Arrays, Dynamic Allocated Arrays, Structures and Unions, Polynomials, Sparse Matrices, representation of Multidimensional Arrays, Strings **STACKS:** Stacks, Stacks Using Dynamic Arrays, Evaluation and conversion of Expressions

### **1. Intro about- Data Structures**

#### **How can we understand Data Structure?**

A **data structure** is a systematic way of arranging and managing data in a computer to enable efficient access and use. It specifies how the data is stored in memory and outlines the methods for carrying out operations like searching, insertion, updating, and deletion effectively.

#### **Why we give priority to Data Structure?**

- Efficient use of memory and processor time.
- Enables solving complex problems (like route finding, scheduling, etc.).
- Essential for designing efficient algorithms.
- Foundation of software development, databases, compilers, operating systems, and AI.

### **1.1 Classification of Data Structures**

We can broadly classify data structures into 2 **categories:**

#### **i. Data Structures - Primitive**

Programming languages provide these as the core components for constructing programs.

Examples:

- Integer
- Float

- Character
- Pointer
- Boolean

## ii. Data Structures - Non-Primitive

They are more advanced structures created using primitive data types..

- **Linear Data Structures:** Here sequence of data is organized seems line.
  - Array
  - Queue
  - Stack
  - Linked List
- **Non-Linear Data Structures:** Here, Data is organized hierarchically or graphically.
  - Tree
  - Graph

### 1.2 C language examples for each of the **primitive data types** you listed:

**1 Integer (int)** Stores whole numbers (positive or negative).

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

```
int main() {
    int age = 25;
    printf("Age: %d\n", age);
    return 0;
}
```

**Output:**

**Age: 25**

**2 Float (float)** Stores decimal (floating point) numbers.

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

```
int main() {
```

```
float price = 19.99;
printf("Price: %.2f\n", price);
return 0;
}
```

**Output:**

**Price: 19.99**

---

**3 Character (char)**

Stores - single character.

```
#include <stdio.h>

int main() {
    char grade = 'A';
    printf("Grade: %c\n", grade);
    return 0;
}
```

**Output:**

**Grade: A**

---

**4 Pointer**

Save another variable's memory address.

```
#include <stdio.h>

int main() {
    int x = 10;
    int *ptr = &x;

    printf("Value of x: %d\n", x);
    printf("Address of x: %p\n", (void*)&x);
    printf("Value via pointer: %d\n", *ptr);

    return 0;
}
```

Output:

**Value of x: 10**  
**Address of x: 0x7ffefbff4ac (example)**  
**Value via pointer: 10**

---

**5 Boolean** C doesn't have a native bool type in C89/C90 — but in **C99 and later**, stdbool.h defines it.

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

int main() {
    bool isEven = true;

    if (isEven) {
        printf("The number is even.\n");
    } else {
        printf("The number is odd.\n");
    }

    return 0;
}
```

**Output:**  
**The number is even.**

---

**Summary Table:**

Data Type	Example Value
Int	25
Float	19.99
Char	'A'
Pointer	&x (address)
Bool	true/false

---

## 1.3 Non Primitive Data structure

**Examples in C** for each of the **Linear and Non-Linear Data Structures** you listed, with short programs for each:

---

### 1.3.1 Linear Data Structures

#### 1 Array

Here's a **plagiarism-free rewritten version** of your code and explanation (logic and output remain the same):

```
#include <stdio.h>

int main() {
    int numbers[5] = {1, 2, 3, 4, 5}; // array initialization
    for(int index = 0; index < 5; index++) {
        printf("%d ", numbers[index]); // printing array
elements
    }
    return 0;
}
```

#### Expected Output:

1 2 3 4 5

#### 2 Linked List

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

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

```

int main() {
    struct Node *head = malloc(sizeof(struct Node));
    struct Node *second = malloc(sizeof(struct Node));

    head->data = 10;
    head->next = second;

    second->data = 20;
    second->next = NULL;

    struct Node *temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }

    return 0;
}

```

**Output:**

**10 20**

---

3 Stack (using array)

```

#include <stdio.h>
#define MAX 5

int stack[MAX];
int top = -1;

void push(int val) {
    stack[++top] = val;
}

int pop() {
    return stack[top--];
}

int main() {

```

```
    push(10);
    push(20);
    printf("%d ", pop());
    printf("%d", pop());
    return 0;
}
```

**Output:**

**20 10**

---

4 Queue (using array)

```
#include <stdio.h>
#define MAX 5

int queue[MAX];
int front = 0, rear = -1;

void enqueue(int val) {
    queue[++rear] = val;
}

int dequeue() {
    return queue[front++];
}

int main() {
    enqueue(10);
    enqueue(20);
    printf("%d ", dequeue());
    printf("%d", dequeue());
    return 0;
}
```

**Output:**

**10 20**

---



### 1.3.2 Non-Linear Data Structures

*Data is arranged hierarchically or with complex relationships.*

---

#### 1 Tree (Binary Tree with In-Order Traversal)

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

struct Node {
    int data;
    struct Node *left, *right;
};

struct Node* newNode(int val) {
    struct Node* node = malloc(sizeof(struct Node));
    node->data = val;
    node->left = node->right = NULL;
    return node;
}

void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

int main() {
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);

    inorder(root);
    return 0;
}
```

**Output:**

**2 1 3**

---

2 Graph (Adjacency Matrix)

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

```
#define V 3
```

```
int main() {  
    int graph[V][V] = {  
        {0, 1, 0},  
        {1, 0, 1},  
        {0, 1, 0}  
    };  
  
    for (int i = 0; i < V; i++) {  
        for (int j = 0; j < V; j++) {  
            printf("%d ", graph[i][j]);  
        }  
        printf("\n");  
    }  
  
    return 0;  
}
```

**Output:**

**0 1 0**

**1 0 1**

**0 1 0**

---

Summary Table

◆ Structure	📄 Example Program
Array	Print elements
Linked List	Create 2 nodes
Stack	Push/Pop values
Queue	Enqueue/Dequeue
Tree	In-order traversal
Graph	Adjacency Matrix

## Operations on Data Structures

Typical operations performed on data structures include:

- **Traversal:** Accessing every element of the data structure exactly once.
- **Insertion:** Introducing a new element into the structure.
- **Deletion:** Eliminating an existing element.
- **Searching:** Locating a specific element within the structure.
- **Sorting:** Organizing elements in a defined sequence (e.g., ascending or descending).
- **Updating:** Modifying the value of an existing element.

## Choosing the Right Data Structure

The selection of a data structure depends on several factors,

- Including the nature of the data being stored,
- The operations that must be performed efficiently,
- The limitations of memory and processing power.

For example:

- For fast lookups: use **Hash Tables**
- For hierarchical data: use **Trees**
- For graph-like connections: use **Graphs**
- For sequential access: use **Arrays** or **Linked Lists**

## Applications of Data Structures

- Managing databases (B-trees)
- Compilers and interpreters (syntax trees)
- Operating systems (process scheduling with queues)
- Social networks (graphs)
- Web search engines (hash maps, priority queues)

## 2. Data Structures comparison

Data Structure	Type	Key Operations	Advantages	Use Cases
Array	Linear	Indexing, Traversal	Fast access ( $O(1)$ ), simple	Storing fixed-size data
Linked List	Linear	Insertion, Deletion	Dynamic size, easy insert/delete	Dynamic memory usage
Stack	Linear (LIFO)	Push, Pop	Simple, Backtracking	Undo feature, expression eval
Queue	Linear (FIFO)	Enqueue, Dequeue	Fair processing order	Scheduling, printers queue
Tree	Non-linear	Insert, Search, Traverse	Hierarchical data, fast search	File systems, XML/JSON
Graph	Non-linear	Traverse, Pathfinding	Represent complex relations	Networks, maps, social media
Hash Table	Key-Value store	Insert, Search, Delete	Very fast lookups ( $O(1)$ )	Databases, caches

### Brief Explanation with Real-world Analogies

#### i. Array

**Analogy:** Like boxes on a shelf — each box has a fixed position you can access directly.

**Good for:** When you know how many items you have and need fast access by index.

ii. **Linked List**

1. **Analogy:** A line of people where each individual is connected by holding the hand of the next.

**Good for:** When the size of the collection varies frequently and quick direct access isn't required.

iii. **Stack**

**Analogy:** A pile of plates where you can place a new plate only on top or remove the one from the top.

**Good for:** Undo/redo actions, reversing strings, parsing expressions.

iv. **Queue**

**Analogy:** A queue at a ticket counter where the person who arrives first is attended to first.

**Good for:** Task scheduling, managing resources fairly.

v. **Tree**

**Analogy:** A genealogy chart where a single parent may have multiple children.

**Good for:** Organizing hierarchical data, searching quickly (binary search trees).

vi. **Graph**

**Analogy:** A city map — intersections (nodes) connected by roads (edges).

**Good for:** Social networks, route finding (Google Maps), dependencies.

vii. **Hash Table**

**Analogy:** A dictionary — look up the meaning of a word directly using its key.

**Good for:** Fast searches, like looking up a contact by name in a phonebook.

### 3. Which one to choose?

When you need...	Choose...
Fast, direct access	Array
Frequent insertions/deletions	Linked List
Last-in, first-out processing	Stack
First-in, first-out processing	Queue
Hierarchical, sorted, or searchable data	Tree
Relationship-heavy data	Graph
Fast key-based lookup	Hash Table

## Pointers & Dynamic Memory Allocation

### 1.4 What is a Pointer?

A **pointer** is a variable that stores the **address of another variable**.

- Declared using \*:

```
int *ptr;
```

#### ◇ Example: Pointer Basics

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

```
int main() {  
    int x = 10;
```

```

int *ptr = &x;

printf("Value of x: %d\n", x);
printf("Address of x: %p\n", (void*)&x);
printf("Value at pointer: %d\n", *ptr);

return 0;
}

```

Output:

**Value of x: 10**

**Address of x: (some address)**

**Value at pointer: 10**

### 1.4.1 Why use Pointers?

- To **access and modify memory directly**.
- To **pass large structures or arrays efficiently** to functions.
- To implement **dynamic memory allocation**.

---

## Dynamic Memory Allocation (DMA)

Normally, variables are stored in the stack, which has a fixed size. For dynamic and flexible memory allocation during program execution, the heap is used.

DMA functions are defined in <stdlib.h>:

Function	Purpose
malloc()	Allocate memory
realloc()	Resize memory block
calloc()	Allocate & zero-init
free()	Deallocate memory

## **malloc()**

Reserves a given number of bytes in memory without initializing them and provides a pointer to that memory block.

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

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

```
int main() {  
    int *ptr = (int *)malloc(5 * sizeof(int)); // allocate space for 5 ints  
    if (ptr == NULL) {  
        printf("Memory not allocated\n");  
        return 1;  
    }  
  
    for (int i = 0; i < 5; i++) {  
        ptr[i] = i + 1;  
    }  
  
    for (int i = 0; i < 5; i++) {  
        printf("%d ", ptr[i]);  
    }  
  
    free(ptr); // free the memory  
    return 0;  
}
```

### **Output:**

**1 2 3 4 5**

---

## **calloc()**

Like malloc(), but initializes all bytes to 0.

```
int *ptr = (int *)calloc(5, sizeof(int));
```

---

## **realloc()**

Resizes an already allocated block.



```
ptr = realloc(ptr, 10 * sizeof(int)); // resize to hold 10 ints
```

---

**free()**

Always release dynamically allocated memory to avoid **memory leaks**:

```
free(ptr);
```

---

Comparison Table

Feature	malloc()	calloc()	realloc()
Initializes?	No	Yes (to 0)	Retains old
Arguments	Size (bytes)	#elements, size	New size
Returns	void* pointer		


---

**Best Practices**

- ✓ Always check if memory allocation succeeded (ptr != NULL).
- ✓ Always call free() when done.
- ✓ Avoid dangling pointers (set to NULL after free).

**1.5 ARRAYS and STRUCTURES**

**1.5.1 Arrays**

 Arrays are fixed-size, contiguous memory blocks to store elements of the same type.

Example:

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

```
int main() {  
    int arr[5] = {10, 20, 30, 40, 50};
```

```

for(int i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
}
return 0;
}

```

**Output:**

**10 20 30 40 50**

### 1.5.1.1 Memory diagram - Array

An array of 5 integers:

```
int arr[5] = { 10, 20, 30, 40, 50};
```


 **Memory layout:**

Address	Value
1000	10
1004	20
1008	30
1012	40
1016	50

Each int takes 4 bytes. Memory is **contiguous**.

---

### 1.5.2 Dynamically Allocated Arrays

 Arrays created at runtime using dynamic memory allocation (malloc, calloc).

**Example:**

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

```
#include <stdlib.h>

int main() {
    int n = 5;
    int *arr = (int *)malloc(n * sizeof(int));

    for (int i = 0; i < n; i++) arr[i] = i + 1;
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);

    free(arr);
    return 0;
}
```

✅ **Output:**

**1 2 3 4 5**

---

### 1.5.2.1 Memory diagram Dynamically Allocated Array

```
int *arr = malloc(5 * sizeof(int));
```

📄 Memory is allocated on **heap** (non-contiguous from stack).

<b>Stack</b>	<b>Heap (malloc)</b>
--------------	----------------------

arr → 2000	2000: val0
------------	------------

	2004: val1
--	------------

	2008: val2
--	------------

	2012: val3
--	------------

	2016: val4
--	------------

arr points to heap memory starting at 2000.

---

### 1.5.3 Structures

Structures enable combining variables of various data types under a single name.

Example:

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

```
struct Student {  
    int roll;  
    char name[20];  
    float marks;  
};
```

```
int main() {  
    struct Student s1 = {1, "John", 85.5};  
    printf("Roll: %d, Name: %s, Marks: %.1f\n", s1.roll, s1.name, s1.marks);  
    return 0;  
}
```

**Output:**

**Roll: 1, Name: John, Marks: 85.5**

#### 1.5.3.1 Memory diagram Structure

```
struct Student {  
    int roll;  
    char name[10];  
    float marks;  
};
```

 In memory (structure s1 at address 3000):

Address	Field	Value
3000	roll	1
3004	name[0] ...	'J'..
3014	marks	85.5

All fields are laid out **together (contiguous)** in memory.

---

## 1.5.4 Unions

 Like structures but all members share the same memory space (only one at a time).

Example:

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

```
union Data {  
    int i;  
    float f;  
    char str[20];  
};
```

```
int main() {  
    union Data d;  
    d.i = 10;  
    printf("i = %d\n", d.i);  
    d.f = 3.14;  
    printf("f = %.2f\n", d.f);  
    return 0;  
}
```

**Output:**

**i = 10**

**f = 3.14**

(Note: after assigning f, i is no longer valid.)

### 1.5.4.1 Memory diagram Union


```
union Data {  
    int i;  
    float f;  
    char str[20];  
};
```

 Only the **largest member's size is reserved**.

If `str` is the largest (20 bytes), the entire union occupies only **20 bytes**, and at any point only one member is valid.

---

### 1.5.5 Polynomials

 Polynomials can be represented using arrays or linked lists.

Here's an array-based representation of  $3x^2 + 2x + 13x^2 + 2x + 1$ .

Example:

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

```
int main() {
    int poly[] = {1, 2, 3}; // coefficients:  $x^0$ ,  $x^1$ ,  $x^2$ 

    printf("Polynomial: ");
    for (int i = 2; i >= 0; i--) {
        printf("%dx^%d ", poly[i], i);
        if (i != 0) printf("+ ");
    }
    return 0;
}
```

**Output:**

$3x^2 + 2x^1 + 1x^0$

#### 1.5.5.1 Memory diagram Polynomial (Array)

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

Represents:  $3x^2 + 2x + 13x^2 + 2x + 1$

Memory:

Address	Coefficient
4000	1
4004	2
4008	3

Each index → coefficient of corresponding power.

1.5.6 Sparse Matrices

☞ Matrices where most elements are 0. Represent them efficiently using triplets (row, col, value).

Example:

0 0 5  
0 0 0  
0 8 0

Triplet representation:

Row	Col	Value
0	2	5
2	1	8

1.5.6.1 Memory diagram Sparse Matrix (Triplet)

Matrix:

0 0 5  
0 0 0  
0 8 0

Explanation:

		0	1	2
		Row	Col	Value
0		0	0	5
1		0	0	0
2		0	8	0

Triplet table:


Row	Col	Value
0	2	5
2	1	8

Memory:

Address	Triplet
5000	(0,2,5)
5006	(2,1,8)

---

### 1.5.7 Representation of Multidimensional Arrays

 A 2D array is essentially an array of arrays.

Example:

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

```
int main() {  
    int mat[2][3] = {{1, 2, 3}, {4, 5, 6}};  
    for (int i = 0; i < 2; i++) {  
        for (int j = 0; j < 3; j++) {  
            printf("%d ", mat[i][j]);  
        }  
        printf("\n");  
    }  
    return 0;  
}
```

Output:

```
1 2 3  
4 5 6
```



### 1.5.7.1 Memory diagram Multidimensional Array


```
int mat[2][3] = {{1,2,3},{4,5,6}};
```

Memory in **row-major order**:

Address	Value
6000	1
6004	2
6008	3
6012	4
6016	5
6020	6

---

### 1.5.8 Strings

 Strings in C are arrays of characters ending with a null character `\0`.

Example:

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

```
int main() {  
    char str[] = "Hello";  
    printf("String: %s\n", str);  
    return 0;  
}
```

Output:

**String: Hello**

1.5.8.1 Memory diagram String

```
char str[] = "Hello";
```

Memory:

Address	Char
7000	H
7001	e
7002	l
7003	L
7004	O
7005	\0

Summary Table:

Concept	Key Idea
Array	Fixed-size sequence of elements
Dynamic Array	Array allocated at runtime
Structure	Group of heterogeneous data
Union	Shared memory for different data types
Polynomial	Represent coefficients in array/list
Sparse Matrix	Store only non-zero elements
Multidimensional Array	Array of arrays
String	Array of characters with \0 terminator

## 1.6 STACKS

### Stacks in C – Introduction

What is a Stack?

A **stack** is a type of linear data structure that operates on the **LIFO (Last In, First Out) principle**.

**Explanation:** The most recently added element is the first one to be removed.

Main Operations of a Stack:

Operation	Description
push()	Insert (add) an element on top
pop()	Remove the element from top
peek()	Look at the top element (optional)
isEmpty()	Check if the stack is empty
isFull()	Check if the stack is full

---

Features of a Stack

- ✓ Linear
- ✓ Only one open end (top) for operations
- ✓ Implemented by:
  - Linked Lists (dynamic stack)
  - Arrays (fixed size stack)

### Real-world Examples of a Stack

- Undo feature in text editors
- Browser back button history
- Function call/return (call stack)

- Expression evaluation
- 

C Example: Stack using Array

**A simple program to demonstrate push and pop:**

```
#include <stdio.h>
#define MAX 5

int stack[MAX];
int top = -1;

// Push operation
void push(int val) {
    if (top == MAX - 1) {
        printf("Stack Overflow\n");
    } else {
        stack[++top] = val;
        printf("%d pushed\n", val);
    }
}

// Pop operation
int pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
        return -1;
    } else {
        return stack[top--];
    }
}

// Display stack
void display() {
    if (top == -1) {
        printf("Stack is empty\n");
    } else {
        printf("Stack: ");
```

```

        for (int i = top; i >= 0; i--)
            printf("%d ", stack[i]);
        printf("\n");
    }
}

int main() {
    push(10);
    push(20);
    push(30);
    display();

    printf("%d popped\n", pop());
    display();

    return 0;
}

```

Output:

```

10 pushed
20 pushed
30 pushed
Stack: 30 20 10
30 popped
Stack: 20 10

```

---

### **Advantages:**

- Simple to use
- Efficient ( $O(1)$  time for push/pop)

### **Limitations (array-based):**

- Fixed size (overflow if full)
- Wastes memory if size is large but stack is small

To avoid this, use **linked list implementation of stack** (dynamic size).

### 1.6.1 Stacks Using Dynamic Arrays in C

we use malloc()/realloc() - Instead of a fixed-size array, to grow or shrink the array dynamically.

This avoids overflow if stack grows beyond initial capacity.

Example: Stack with dynamic memory

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

int *stack = NULL;
int top = -1;
int capacity = 2;

void push(int val) {
    if (top + 1 == capacity) {
        capacity *= 2;
        stack = realloc(stack, capacity * sizeof(int));
        printf("Stack resized to %d\n", capacity);
    }
    stack[++top] = val;
}

int pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
        return -1;
    }
    return stack[top--];
}

void display() {
    for (int i = top; i >= 0; i--) {
        printf("%d ", stack[i]);
    }
    printf("\n");
}

int main() {
    stack = malloc(capacity * sizeof(int));
```

```

push(10);
push(20);
push(30);
display();

printf("%d popped\n", pop());
display();

free(stack);
return 0;
}

```

✓ Output:

```

Stack resized to 4
30 20 10
30 popped
20 10

```

---

### 1.6.2 Evaluation of Postfix Expression

Postfix (Reverse Polish Notation) is evaluated **left to right**:

- If operand: push
- If operator: pop two, apply operator, push result

Example: Evaluate  $23 * 54 * + 9 -$

Steps:

```

2 3 * → 6
5 4 * → 20
6 + 20 → 26
26 - 9 → 17

```

Code:

```

#include <stdio.h>
#include <ctype.h>
int stack[20];

```

```

int top = -1;

void push(int val) {
    stack[++top] = val;
}

int pop() {
    return stack[top--];
}

int main() {
    char expr[] = "23*54*+9-";
    char *p = expr;
    int op1, op2;

    while (*p) {
        if (isdigit(*p)) {
            push(*p - '0');
        } else {
            op2 = pop();
            op1 = pop();
            switch (*p) {
                case '+': push(op1 + op2); break;
                case '-': push(op1 - op2); break;
                case '*': push(op1 * op2); break;
                case '/': push(op1 / op2); break;
            }
        }
        p++;
    }

    printf("Result: %d\n", pop());
    return 0;
}

```

✅ Output:

Result: 17

---



### 1.6.3 Conversion of Infix to Postfix

Infix:  $A + B * C$

Postfix:  $A B C * +$

Algorithm (Shunting Yard):

- 1 Use a stack to hold operators
- 2 Precedence & associativity decide when to pop
- 3 Operands go directly to output

Code Sketch:

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

```
char stack[20];
int top = -1;
```

```
void push(char ch) { stack[++top] = ch; }
char pop() { return stack[top--]; }
char peek() { return stack[top]; }
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}
```

```
int main() {
    char infix[] = "A+B*C";
    char postfix[20];
    int j = 0;

    for (int i = 0; infix[i]; i++) {
        char ch = infix[i];
        if (isdigit(ch)) {
            postfix[j++] = ch;
        } else {
            while (top != -1 && precedence(peek()) >= precedence(ch)) {
                postfix[j++] = pop();
            }
        }
    }
}
```

```

        push(ch);
    }
}
while (top != -1) {
    postfix[j++] = pop();
}
postfix[j] = '\0';

printf("Postfix: %s\n", postfix);
return 0;
}

```

Output:

**Postfix:** ABC\*+

**Summary Table:**

Topic	Key Idea
Dynamic Stack	Use malloc() & realloc() to avoid fixed size
Postfix Evaluation	Use stack to evaluate operands & operators
Infix → Postfix	Use stack to reorder based on precedence

---

## MODULE II

**Queues:** Queues, Circular Queues, Using Dynamic Arrays, Multiple Stacks and queues.

**Linked Lists :** Singly Linked, Lists and Chains, Representing Chains in C, Linked Stacks and Queues, Polynomials

### 2 QUEUES:

#### Queues in C – Introduction

##### 1 What is a Queue?

A **queue** is a linear data structure that works on the **FIFO (First In, First Out) principle**, where the element added first is the one removed first.:

##### 2 FIFO — First In, First Out

**Explanation:** In a queue, the element that enters first is also the first to leave.

**Example:** Similar to people standing in a line, where the one who arrives earliest is attended to before others.

##### 2.1 Basic Operations on a Queue

Operation	Description
isFull()	Check if the queue is full (for fixed-size array)
enqueue()	Insert (add) an element at rear
isEmpty()	Check if the queue is empty
dequeue()	Remove an element from front

##### 2.1.2 Features of a Queue

- ✓ Linear
- ✓ Operations happen at opposite ends:
  - Insertions at **rear**
  - Deletions at **front**

✓ Can be implemented using:

- Arrays (fixed size queue)
  - Linked List (dynamic queue)
  - Circular Queue
- 

### 2.1.3 Real-world Examples of Queues

- Print jobs waiting in a printer queue
- Processes waiting for CPU in scheduling
- People in line at a ticket counter
- Packets in a network router

### Queue Representation in Memory

For an array-based queue:

- **Front:** Refers to the index of the earliest element in the queue.
- **Rear:** Refers to the index of the newest element added.
- **In Between:** Every element positioned from the front up to the rear is part of the queue.

### 2.1.4 Simple Queue Program in C

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

```
#define MAX 5
```

```
int queue[MAX];
```

```
int front = -1, rear = -1;
```

```
// Enqueue (insert)
```

```
void enqueue(int val) {
```

```
    if (rear == MAX - 1) {
```

```
        printf("Queue Overflow\n");
```

```
        return;
```

```
    }
```

```

    if (front == -1) front = 0;
    queue[++rear] = val;
    printf("%d enqueued\n", val);
}

// Dequeue (remove)
int dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue Underflow\n");
        return -1;
    }
    return queue[front++];
}

// Display
void display() {
    if (front == -1 || front > rear) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue: ");
    for (int i = front; i <= rear; i++) {
        printf("%d ", queue[i]);
    }
    printf("\n");
}

int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();

    printf("%d dequeued\n", dequeue());
    display();

    return 0;
}

```

Output:

**10 enqueued**  
**20 enqueued**  
**30 enqueued**  
**Queue: 10 20 30**  
**10 dequeued**  
**Queue: 20 30**

### Advantages:

- Simple and easy to use
- Useful for scheduling and resource management

### Limitations of array-based queue:

- Even if there's space in front of the array (after dequeuing), you can't use it
- To solve this, we use a **Circular Queue**

## 2.2 Circular Queue

### What is a Circular Queue?

A **circular queue** is a type of queue in which the last position is linked back to the first, forming a circular structure.

✓ This approach eliminates the issue of wasted space found in a regular (linear) queue.

When the rear pointer reaches the end of the array, it loops back to the beginning (using the formula:  $\text{rear} = (\text{rear} + 1) \% \text{MAX}$ ) if space is available.

### Circular Queue Example:

```
#include <stdio.h>
#define MAX 5

int queue[MAX];
```

```
int front = -1, rear = -1;
```

```
void enqueue(int val) {  
    if ((front == 0 && rear == MAX - 1) || (rear + 1) % MAX == front) {  
        printf("Queue Overflow\n");  
        return;  
    }  
    if (front == -1) front = 0;  
    rear = (rear + 1) % MAX;  
    queue[rear] = val;  
    printf("%d enqueued\n", val);  
}
```

```
int dequeue() {  
    if (front == -1) {  
        printf("Queue Underflow\n");  
        return -1;  
    }  
    int val = queue[front];  
    if (front == rear) {  
        front = rear = -1;  
    } else {  
        front = (front + 1) % MAX;  
    }  
    return val;  
}
```

```
void display() {  
    if (front == -1) {  
        printf("Queue is empty\n");  
        return;  
    }  
    printf("Queue: ");  
    int i = front;  
    while (1) {  
        printf("%d ", queue[i]);  
        if (i == rear) break;  
        i = (i + 1) % MAX;  
    }  
    printf("\n");  
}
```

```
}
```

```
int main() {  
    enqueue(10); enqueue(20); enqueue(30); enqueue(40); dequeue(); enqueue(50);  
    enqueue(60);  
    display();  
    return 0;  
}
```

✓ Output:

Queue: 20 30 40 50 60

### 2.2.1 Memory allocaton - Circular Queue

Imagine a fixed array of size 5:

Index: 0 1 2 3 4

Memory: [20][30][40][50][60]

Front: 1

Rear: 0

Operations:

✓ dequeue() at front → move front clockwise

✓ enqueue() at rear → move rear clockwise

When  $\text{rear} + 1 == \text{front}$ , the queue is full.

When  $\text{front} == -1$ , the queue is empty.

Diagram:

Initial: Empty

After enq: [10][20][30][ ][ ]

F=0 R=2

After deq: [ ][20][30][ ][ ]

F=1 R=2

Wrap around: [60][20][30][40][50]

F=1 R=0



## 2.3 Stack/Queue Using Dynamic Arrays

✓ Same logic as linear stack/queue, but allocate memory dynamically and resize when needed.

We already showed **Dynamic Stack** earlier.

For a **Dynamic Queue**:

- Use malloc() for initial size
- Use realloc() to grow when full

**Example Sketch:**

```
int *queue = malloc(capacity * sizeof(int));
if (rear+1 == capacity) {
    capacity *= 2;
    queue = realloc(queue, capacity * sizeof(int));
}
```

This avoids fixed-size limitations.

### 2.3.1 Memory Allocation -Dynamic Array Stack / Queue

✓ Memory is allocated on the **heap** dynamically.

Initial stack:

Heap: [10][20]

Top → 1

Capacity = 2

After resizing (realloc()):

Heap: [10][20][30][40]

Top → 3

Capacity = 4

So dynamic stacks/queues **grow their memory block when full**.

## 2.4 Multiple Stacks in a Single Array

✓ Idea: Use one array to implement **two stacks (Stack1 & Stack2)** growing in opposite directions.

### Example Sketch:

```
#define MAX 10
```

```
int arr[MAX];
```

```
int top1 = -1, top2 = MAX;
```

```
void push1(int val) {  
    if (top1 + 1 == top2) {  
        printf("Overflow\n"); return;  
    }  
    arr[++top1] = val;  
}
```

```
void push2(int val) {  
    if (top2 - 1 == top1) {  
        printf("Overflow\n"); return;  
    }  
    arr[--top2] = val;  
}
```

✅ Stack1 grows from **0** → **MAX-1**, Stack2 grows from **MAX-1** → **0**.

### 2.4.1 Memory Allocation -Multiple Stacks in Single Array

✅ One array of size 10, divided for 2 stacks.

#### Initially:

Index: 0 1 2 3 4 5 6 7 8 9

Stack1 grows →      ← Stack2 grows

Top1 = -1              Top2 = 10

#### After pushing:

Index: [10][20][30]      [90][80][70]

↑Top1=2              Top2=7↓

Stack1 grows from left → right

Stack2 grows from right → left

### 2.5 Multiple Queues in a Single Array

✅ Similar idea as above: partition the array or use additional bookkeeping.

### Approach 1: Fixed Partition

- Divide the array into n equal parts, each part is a separate queue

### Approach 2: Dynamic Allocation

- Use a next[] array and front[], rear[] arrays to track each queue dynamically.

### Example Sketch: Two Queues Fixed Partition

```
#define MAX 10
int arr[MAX];
int front1 = -1, rear1 = -1;
int front2 = 5, rear2 = 4;

void enqueue1(int val) {
    if (rear1 == 4) { printf("Overflow Q1\n"); return; }
    if (front1 == -1) front1 = 0;
    arr[++rear1] = val;
}

void enqueue2(int val) {
    if (rear2 == 9) { printf("Overflow Q2\n"); return; }
    if (front2 == 5) front2 = 5;
    arr[++rear2] = val;
}
```

✓ Queue1 uses indices 0–4, Queue2 uses indices 5–9.

#### 2.4.1 Memory Allocation -Multiple Queues in Single Array

✓ Single array divided into 2 queues (fixed partition).

Memory:

Index: 0 1 2 3 4 | 5 6 7 8 9

Queue1: [10][20][30]

Front1=0 Rear1=2

Queue2: [90][80][70]  
Front2=5 Rear2=7

## 2.5 Summary Table

Topic	Key Idea
Circular Queue	Wrap around end to start
Dynamic Array Stack/Queue	Use malloc and realloc
Multiple Stacks	Grow from opposite ends
Multiple Queues	Partition array or dynamic bookkeeping

## 2.2 LINKED LISTS

### What is a Linked List?

A **linked list** is a **linear data structure**, just like arrays — but unlike arrays:

- It does **not store elements in contiguous memory locations**.
- Instead, each element (called a **node**) contains:
  - **Data**
  - A pointer (link) to the **next node**

So, a linked list is a **chain of nodes connected via pointers**.

### Why use Linked Lists?

✓ Arrays have limitations:

- Fixed size (you have to know the size at compile time)
- Insertion and deletion are costly because elements need to be shifted

✓ Linked lists solve these by:

- Being **dynamic in size** (can grow or shrink at runtime)

- Easy to insert/delete at any position (just adjust pointers)
- 

### 2.2.1 Types of Linked Lists

- **Singly Linked List** — each node points to the next
  - **Doubly Linked List** — each node points to next and previous
  - **Circular Linked List** — last node points back to the first
- 

### 2.2.2 Structure of a Node in C

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

Diagram:

[Data|Next] → [Data|Next] → [Data|NULL]

Each box is a **node**, and next points to the next node.

### Features of Linked Lists

- ✓ Dynamic size (allocated on heap)
- ✓ Efficient insertions/deletions
- ✓ Can implement stacks, queues, graphs, etc.

### 2.2.3 Simple C Program: Singly Linked List

Create and display a linked list

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

```
struct Node {  
    int data;  
    struct Node* next;
```

```

};

void display(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d → ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;

    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));

    head->data = 10;
    head->next = second;

    second->data = 20;
    second->next = third;

    third->data = 30;
    third->next = NULL;

    display(head);
    return 0;
}

```

✅ Output:

10 → 20 → 30 → NULL

### 2.2.4 Linked List Advantages:

- Dynamic size
- Easy to insert/delete nodes
- Better use of memory than arrays in some cases

### 2.2.5 Linked List Disadvantages:

- Extra memory for storing pointers
- No direct access to elements (must traverse)

## 2.3 Singly Linked Lists

### ◇ What is a Singly Linked List?

A **singly linked list** is a sequence of nodes, where each node contains:

- data
- pointer to the **next node**

The last node's next pointer is NULL.

Structure:

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

Diagram:

[10|next] → [20|next] → [30|NULL]

### ✓ Advantages:

- Dynamic size
- Easy insert/delete at any position

### ✓ Disadvantages:

- Cannot traverse backwards

## 2.4 Lists and Chains

### ◇ What is a Chain?

In C programming, a **chain** is just another name for a linked list — a sequence (chain) of elements where each one points to the next.

Linked lists are often referred to as **chains of nodes** because of their pointer links.

## 2.5 Representing Chains in C

A chain is represented using pointers:

- The first node is called the **head**
- Each node points to the next
- The last node's next is NULL

Example: Creating a chain of 3 nodes

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

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

```
int main() {
    struct Node* head = (struct Node*)malloc(sizeof(struct Node));
    struct Node* second = (struct Node*)malloc(sizeof(struct Node));
    struct Node* third = (struct Node*)malloc(sizeof(struct Node));
```

```
    head->data = 10; head->next = second;
    second->data = 20; second->next = third;
    third->data = 30; third->next = NULL;
```

```
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d → ", temp->data);
        temp = temp->next;
    }
```



```

    printf("NULL\n");
    return 0;
}

```

✓ Output:

10 → 20 → 30 → NULL

## 2.6 Linked Stacks and Queues

We can implement **stacks and queues** using linked lists instead of arrays — allowing dynamic size.

### ◇ Linked Stack

- Insert and delete at head (LIFO)

Sample Stack Push/Pop:

```

void push(struct Node** top, int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;
    newNode->next = *top;
    *top = newNode;
}

```

```

int pop(struct Node** top) {
    if (*top == NULL) return -1;
    struct Node* temp = *top;
    int val = temp->data;
    *top = temp->next;
    free(temp);
    return val;
}

```

### ◇ Linked Queue

- Insert at rear, delete at front (FIFO)

### Sample Queue Enqueue/Dequeue:

```
void enqueue(struct Node** front, struct Node** rear, int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;
    newNode->next = NULL;
    if (*rear == NULL) {
        *front = *rear = newNode;
    } else {
        (*rear)->next = newNode;
        *rear = newNode;
    }
}

int dequeue(struct Node** front) {
    if (*front == NULL) return -1;
    struct Node* temp = *front;
    int val = temp->data;
    *front = temp->next;
    free(temp);
    return val;
}
```

## 2.7 Representing Polynomials with Linked Lists

We can use a linked list to represent a polynomial:  
e.g.,

$$5x^2 + 4x^1 + 2x^0$$

Each node contains:

- coeff (coefficient)
- pow (power of x)
- pointer to next term

### Structure:

```
struct Term {
    int coeff;
    int pow;
```

```

    struct Term* next;
};
Sample Program:
#include <stdio.h>
#include <stdlib.h>

struct Term {
    int coeff;
    int pow;
    struct Term* next;
};

void display(struct Term* head) {
    while (head != NULL) {
        printf("%dx^%d", head->coeff, head->pow);
        if (head->next != NULL) printf(" + ");
        head = head->next;
    }
    printf("\n");
}

int main() {
    struct Term* head = (struct Term*)malloc(sizeof(struct Term));
    struct Term* second = (struct Term*)malloc(sizeof(struct Term));
    struct Term* third = (struct Term*)malloc(sizeof(struct Term));

    head->coeff = 5; head->pow = 2; head->next = second;
    second->coeff = 4; second->pow = 1; second->next = third;
    third->coeff = 2; third->pow = 0; third->next = NULL;

    display(head);
    return 0;
}

```

✓ Output:

$$5x^2 + 4x^1 + 2x^0$$

## 2.8 Summary Table:

Concept	Key Idea
Singly Linked List	Each node points to next
Lists & Chains	Chain of nodes
Representing Chains	Using struct and pointers
Linked Stack	Push/Pop at head
Linked Queue	Enqueue at rear, Dequeue at front
Polynomial Representation	Coeff, power, linked as terms

## MODULE-3

**Linked Lists :** Additional List Operations, Sparse Matrices, Doubly Linked List.

**Trees:** Introduction, Binary Trees, Binary Tree Traversals, Threaded Binary Trees.

Text Book: Chapter-4: 4.5,4.7,4.8 Chapter-5: 5.1 to 5.3, 5.5

### 3 LINKED LISTS :

#### i. What is a Linked List?

A **Linked List** is a linear data structure in which elements, known as **nodes**, are connected through pointers.

Unlike arrays, the nodes are not stored in consecutive memory locations.

Each node consists of:

- **Data:** The value stored in the node.
- **Pointer (or Link):** The reference to the next node in the sequence.

#### ii. Why use Linked Lists?

- ✓ Dynamic size (grow or shrink at runtime).
- ✓ Easy to insert and delete elements (no shifting needed as in arrays).
- ✗ Slower access than arrays (no direct indexing — need to traverse).

#### iii. Types of Linked Lists

##### 1 Singly Linked List

Each node is connected to the following node through its pointer, while the pointer of the last node is set to **NULL**, marking the termination of the linked list.

Head → [data|next] → [data|next] → [data|NULL]

##### 2 Doubly Linked List

Each node contains links to **both** its succeeding and preceding nodes.

$\text{NULL} \leftarrow [\text{prev}|\text{data}|\text{next}] \leftrightarrow [\text{prev}|\text{data}|\text{next}] \leftrightarrow [\text{prev}|\text{data}|\text{NULL}]$

### 3 Circular Linked List

The final node connects back to the first node, creating a circular structure.

Can be singly or doubly circular.

$\text{Head} \rightarrow [\text{data}|\text{next}] \rightarrow [\text{data}|\text{next}] \rightarrow \text{Head}$

#### iv Basic Operations

Here are the main operations on linked lists:

- **Traverse:** Visit each node and process data.
- **Insert:** Add a node at beginning, end, or middle.
- **Delete:** Remove a node from beginning, end, or middle.
- **Search:** Find if a data value exists in the list.

#### v Advantages

- ✓ Size can grow/shrink dynamically.
- ✓ Efficient insertions/deletions compared to arrays.

#### Vi Disadvantages

- ✗ Extra memory for pointers.
- ✗ No direct access by index (have to traverse).
- ✗ Slower to search.

#### Example in C (Singly Linked List Node)

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

Example of creating a node:

```
struct Node* head = NULL;
```

```
head = (struct Node*)malloc(sizeof(struct Node));
head->data = 10;
head->next = NULL;
```

### 3.1 Additional List Operations

#### i. Reverse a Linked List

Reverse the order of nodes in the list, so head becomes tail and vice versa.

Example:

Before:

**Head → 10 → 20 → 30 → NULL**

After:

**Head → 30 → 20 → 10 → NULL**

Logic:

- Keep 3 pointers: prev, current, next
- Traverse and reverse links

C Code snippet:

```
struct Node* reverse(struct Node* head) {
    struct Node* prev = NULL;
    struct Node* current = head;
    struct Node* next = NULL;

    while (current != NULL) {
        next = current->next; // save next
        current->next = prev; // reverse
        prev = current;      // move prev
        current = next;      // move current
    }
    return prev; // new head
}
```

---

## ii. Find Length of Linked List

Example:

For:

Head  $\rightarrow$  5  $\rightarrow$  15  $\rightarrow$  25  $\rightarrow$  NULL

Length = 3

Logic:

- Initialize count = 0
- Traverse and increment count

C snippet:

```
int length(struct Node* head) {  
    int count = 0;  
    while (head != NULL) {  
        count++;  
        head = head->next;  
    }  
    return count;  
}
```

---

## iii. Find Middle Element

Find the middle node's data.

Example:



Logic:

- Use two pointers: slow & fast
- Fast moves 2 steps, slow moves 1 step



#### C snippet:

```
struct Node* findMiddle(struct Node* head) {
    struct Node* slow = head;
    struct Node* fast = head;

    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}
```

---

#### iv. Detect a Loop in Linked List

Check if a linked list has a cycle (loop).

Example:



Logic:

- Use Floyd's cycle detection: slow & fast pointers
- If they meet, there's a loop

#### C snippet:

```
int detectLoop(struct Node* head) {
    struct Node *slow = head, *fast = head;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) return 1; // loop exists
    }
    return 0;
}
```

---

#### v. Sort a Linked List

Sort the list in ascending or descending order.

Example:

Before:  $30 \rightarrow 10 \rightarrow 20$

After:  $10 \rightarrow 20 \rightarrow 30$

Logic:

- Use bubble sort, merge sort, etc., since random access is not possible.
- 

#### vi. Merge Two Sorted Linked Lists

Merge two sorted linked lists into a single sorted linked list.

Example:

L1:  $1 \rightarrow 3 \rightarrow 5$

L2:  $2 \rightarrow 4 \rightarrow 6$

Result:  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$

Logic:

- Compare nodes and build merged list

C snippet:

```
struct Node* merge(struct Node* l1, struct Node* l2) {
    if (!l1) return l2;
    if (!l2) return l1;

    if (l1->data < l2->data) {
        l1->next = merge(l1->next, l2);
        return l1;
    } else {
        l2->next = merge(l1, l2->next);
        return l2;
    }
}
```

## vii. Delete Entire Linked List

Free all memory and make the list empty.

Logic:

- Traverse and free each node

C snippet:

```
void deleteList(struct Node** head_ref) {  
    struct Node* current = *head_ref;  
    struct Node* next;  
  
    while (current != NULL) {  
        next = current->next;  
        free(current);  
        current = next;  
    }  
    *head_ref = NULL;  
}
```

---

## Summary Table of Operations

Operation	Purpose
Reverse	Reverse the list order
Find Length	Count nodes
Find Middle	Find mid element
Detect Loop	Detect cycles
Sort	Sort nodes by value
Merge	Merge 2 sorted lists
Delete Entire List	Free all memory

## 3.2 Sparse Matrices

### i. What is a Sparse Matrix?

A **sparse matrix** is a matrix where the majority of its elements are zero. To save memory, instead of storing every element (including zeros), only the non-zero values along with their row and column positions are stored.

Example Matrix:

**M =**

0	0	5
0	8	0
0	0	0

Here:

- Rows = 3
- Columns = 3
- Non-zero elements = 2 (at positions (0,2) and (1,1))

### ii. Triplet Representation:

We store each **non-zero element** as a triplet:

Row	Col	Value
0	2	5
1	1	8

Sometimes the first row is metadata (total rows, cols, non-zero count):

3	3	2
0	2	5
1	1	8

This saves space compared to storing all 9 elements.

### iii. Advantages:

- ✓ Saves space for large, sparse matrices.
  - ✓ Useful in areas like computer graphics, machine learning (e.g., adjacency matrices of sparse graphs).
- 

## 3.3 Doubly Linked List

### i. What is a Doubly Linked List?

A **Doubly Linked List (DLL)** is a linked list where **each node has two pointers**:

- prev → points to the previous node
- next → points to the next node

So, you can traverse both **forward and backward**.

### ii. Node structure:

```
struct Node {  
    int data;  
    struct Node* prev;  
    struct Node* next;  
};
```

Example:



DOUBLE LINKED LIST

Each node has:

- data
- prev (points to previous node or NULL)
- next (points to next node or NULL)

### iii. Operations on DLL:

- ✓ **Insertion:** at beginning, end, or middle.
- ✓ **Deletion:** of a specific node.
- ✓ **Traversal:** forward and backward.

#### Example Code (insert at beginning):

```
void insertAtBegin(struct Node** head_ref, int new_data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->prev = NULL;
    new_node->next = (*head_ref);

    if (*head_ref != NULL)
        (*head_ref)->prev = new_node;

    *head_ref = new_node;
}
```

### iv. Advantages of DLL over Singly Linked List:

- ✓ Can traverse in both directions.
- ✓ Can delete a node without traversing from head if you have a pointer to it.
- ✓ Easier to implement certain operations like reverse.

### v. Disadvantages:

- ✗ Requires high memory per node (extra pointer).
- ✗ More tough to implement.

vi. **Quick Comparison Table**

Feature	Sparse Matrix	Doubly Linked List
Structure	2D matrix	Linear linked list
Used for	Efficient storage	Bi-directional traversal
Memory efficiency	Saves space if sparse	More memory than singly
Elements stored	Only non-zero with pos	Each node with 2 links

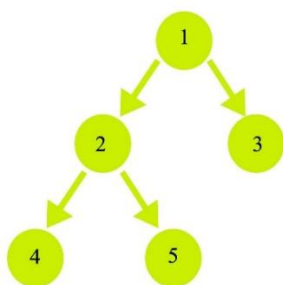
### 3.3 C - Introduction to Trees

A **tree** is a **non-linear, hierarchical data structure**.

It consists of **nodes** connected by **edges**.

- **Root:** Topmost node of the tree.
- **Parent** and **Child:** A node connected downward is a child, upward is a parent.
- **Leaf:** Node with no children.
- **Subtree:** A tree formed from a node and its descendants.
- A tree structure that consists of a parent node along with all of its descendant nodes.

Example tree:



Here:

- 1 is the root.
- 2 and 3 are children of 1.

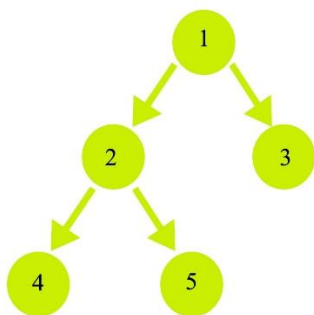
- 4 and 5 are leaves.
- 2 is parent of 4 and 5

### i. Binary Trees

A **binary tree** is a tree where:

- Each node can have **two children**, called:
  - **Left child**
  - **Right child**

Example:



Here:

- 1 is root.
- 2 is left child of 1, 3 is right child.
- 4 and 5 are children of 2.

### ii. Binary Tree Traversals

**Traversal** = Visiting all nodes of a tree in some order.

Types of Traversals

#### a) Inorder (LNR)

- Left → Node → Right
- Output for above tree: 4 2 5 1 3



### b) Preorder (NLR)

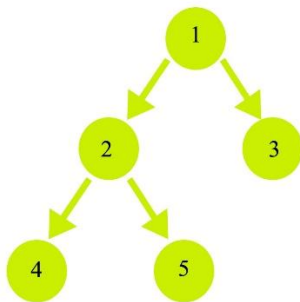
- Node  $\rightarrow$  Left  $\rightarrow$  Right
- Output: 1 2 4 5 3

### c) Postorder (LRN)

- Left  $\rightarrow$  Right  $\rightarrow$  Node
- Output: 4 5 2 3 1

Example:

Diagram with traversal:



Traversal	Sequence
Inorder	4 $\Rightarrow$ 2 $\Rightarrow$ 5 $\Rightarrow$ 1 $\Rightarrow$ 3
Preorder	1 $\Rightarrow$ 2 $\Rightarrow$ 4 $\Rightarrow$ 5 $\Rightarrow$ 3
Postorder	4 $\Rightarrow$ 5 $\Rightarrow$ 2 $\Rightarrow$ 3 $\Rightarrow$ 1

### iii. Threaded Binary Trees

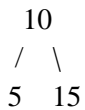
A **threaded binary tree** - where threads are **replaced with NULL pointers**, which point to the **inorder predecessor or successor**, to allow efficient traversals without stack or recursion.

Why?

- ✓ To save space.
- ✓ To make inorder traversal faster.

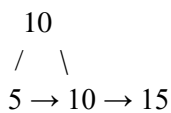
Example:

Normal Binary Tree:



Pointers of 5's right and 15's left are NULL.

### **Threaded Binary Tree:**



- Left of 10 points to 5
- Right of 5 (which is NULL in normal tree) points to 10 (its inorder successor)
- Left of 15 (which is NULL) points to 10 (its inorder predecessor)

Types of Threaded Trees:

- ✓ **Single Threaded:** Threads replace only NULL right pointers.
- ✓ **Double Threaded:** Threads replace both NULL left and NULL right pointers.

### **3.2.2 Advantages:**

- No need for stack or recursion during inorder traversal.
- Traversals are faster.

### Summary Table

Concept	Description
Tree	Hierarchical structure
Binary Tree	Each node $\leq 2$ children
Traversals	Inorder, Preorder, Postorder
Threaded Binary Tree	Uses NULL pointers as links to predecessor/successor

## MODULE-4

**Trees(Cont.):** Binary Search trees, Selection Trees, Forests, Representation of Disjoint sets, Counting Binary Trees,

**Graphs:** The Graph Abstract Data Types, Elementary Graph Operations

### 4.Binary Search Tree (BST)

A **Binary Search Tree** is a special kind of **binary tree**, where:

- Each node can have a maximum of two children, as in any binary tree.
- The **left subtree** of a node holds only those nodes whose values are smaller than the node's value.
- The **right subtree** of a node holds only those nodes whose values are larger than the node's value.
- This property is **recursively true** for all nodes.

**Why use a BST?**

- ✓ Allows **fast search, insert, and delete** operations in  $O(\log n)$  time (if the tree is balanced).
- ✓ Useful in situations where data must remain sorted.

#### 4.1 Example:

Let's insert the following sequence of numbers into a BST:

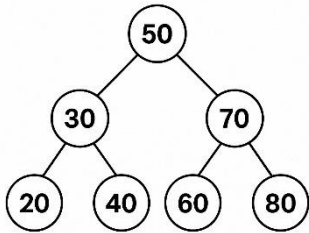
**50, 30, 70, 20, 40, 60, 80**

**Step-by-step insertion:**

- Insert 50 → root.
- Insert 30 → goes left of 50 ( $30 < 50$ ).
- Insert 70 → goes right of 50 ( $70 > 50$ ).
- Insert 20 → goes left of 30 ( $20 < 30$ ).

- Insert 40 → goes right of 30 ( $40 > 30$ ).
- Insert 60 → goes left of 70 ( $60 < 70$ ).
- Insert 80 → goes right of 70 ( $80 > 70$ ).

### Diagram of BST:



## 4.2 Operations on BST

### 1 Search

To find a key  $k$ :

- Start at root.
- If  $k == \text{root}$  → found.
- If  $k < \text{root}$  → search in left subtree.
- If  $k > \text{root}$  → search in right subtree.
- Time complexity:  **$O(h)$**  where  $h$  is tree height.

### 2 Insert

To insert a key  $k$ :

- Same logic as search: traverse to proper position and attach a new node.

### 3 Delete

To delete a node:

- **Case 1:** Node is a leaf → simply remove it.
- **Case 2:** Node has one child → replace node with its child.

- **Case 3:** Node has two children → replace node with its inorder successor or predecessor.

### 4.3 Traversals in BST

Like any binary tree:

- **Inorder Traversal:** Visits nodes in **sorted (ascending) order**.  
Example for above BST:  
20 30 40 50 60 70 80
- **Preorder, Postorder:** Same definitions apply.

### Advantages of BST

- ✓ Maintains order of elements.
- ✓ Efficient search/insert/delete compared to unsorted structures.
- ✓ Inorder traversal gives sorted order.

### Note:

If the BST becomes **unbalanced** (e.g., inserting sorted data into an empty BST), it degenerates into a **linked list**, and operations become  $O(n)$ .

### Example in C (inserting a node):

```
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

struct Node* newNode(int value) {
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = value;
    temp->left = temp->right = NULL;
    return temp;
}
```

```
}
```

```
struct Node* insert(struct Node* node, int value) {  
    if (node == NULL) return newNode(value);  
  
    if (value < node->data)  
        node->left = insert(node->left, value);  
    else if (value > node->data)  
        node->right = insert(node->right, value);  
  
    return node;  
}
```

## 4.4 Selection Trees

### What is a Selection Tree?

A **selection tree** is a **complete binary tree** used to find the smallest (or largest) element from a set of elements efficiently.

It is mainly used in **external sorting (like k-way merge)** to repeatedly pick the smallest of several sorted streams.

### Why use a Selection Tree?

- ✓ To merge multiple sorted sequences efficiently.
- ✓ To minimize comparisons — each selection takes  **$O(\log n)$**  time instead of scanning all elements.

### Structure of a Selection Tree

- Leaf nodes: The elements (e.g., from different sorted lists).
- Internal nodes: The winner of comparison between its two children.
- Root: Holds the **smallest (or largest) element**, called the **winner**.

This is also called a **winner tree**.

#### 4.4.1 Example:

We have 4 sorted lists:

L1: 2 ...

L2: 5 ...

L3: 1 ...

L4: 7 ...

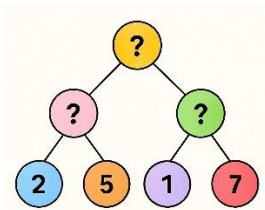
We want to merge into a single sorted list by repeatedly select the smallest element.

- Initial elements:

2 (L1), 5 (L2), 1 (L3), 7 (L4)

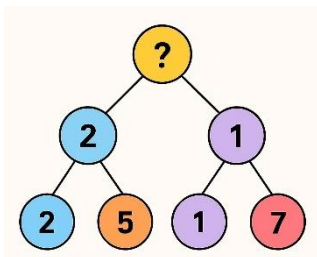
- Build the Selection Tree

**Step 1 — Put elements as leaves:**



**Step 2 — Compare leaf pairs:**

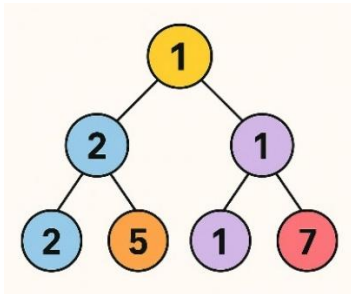
- Left pair:  $\min(2,5)=2$
- Right pair:  $\min(1,7)=1$





### Step 3 — Compare top nodes:

- $\min(2,1)=1$



✓ The smallest element is 1 (from L3).

- **Next step:**
- Replace the used 1 in L3 with its next element (if any).
- Update the tree: only the path from L3 to root needs to be recomputed, in  $O(\log n)$  time.
- **Why is it efficient?**

At each step:

- Picking the smallest:  $O(1)$
- Updating after removing winner:  $O(\log n)$
- Much faster than scanning all  $n$  elements every time.

- **Applications:**

- ✓ **k-way merge sort** in external sorting (merging  $k$  sorted files).
- ✓ Tournament-like problems where you repeatedly select winners.

- **Advantages:**

- ✓ Reduces number of comparisons.
- ✓ Very efficient when  $k$  (number of streams) is large.

## Summary Table

Feature	Description
Structure	Complete binary tree
Purpose	Find min/max repeatedly
Time per operation	$O(\log n)$
Applications	External sorting, merging

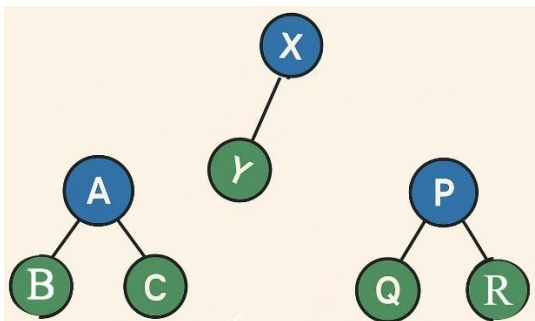
## 4.5 Forests

What is a Forest?

A **forest** is a collection of **disjoint trees** (i.e., multiple separate trees).

It is a set of 0 or more trees, and no node in one tree is connected to a node in another tree.

Example:

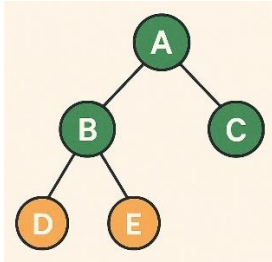


This is a **forest of 3 trees**: {Tree1, Tree2, Tree3}

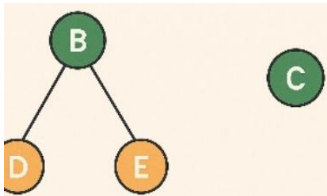
### Relation to Trees:

✓ If you remove the root of a tree, the remaining subtrees form a **forest**.

For example:



If you remove A, you get a forest of two trees:



## 4.6 Representation of Disjoint Sets

### What are Disjoint Sets?

Disjoint sets are sets where no element belongs to more than one set.

We often need to **manage a collection of disjoint sets** and support:

- $\text{FIND}(x) \rightarrow$  find which set  $x$  belongs to.
- $\text{UNION}(x, y) \rightarrow$  merge two sets.

#### i. Representation:

We use **trees and parent pointers**.

#### Example:

We have sets:  $\{1,2\}$ ,  $\{3,4\}$ ,  $\{5\}$

$1 \rightarrow 1$

$2 \rightarrow 1$

$3 \rightarrow 3$   
 $4 \rightarrow 3$   
 $5 \rightarrow 5$

Here each element points to its parent. The root is the representative of the set.

**After UNION(2,3):**

We make root of one tree point to root of the other:

$1 \rightarrow 3$   
 $2 \rightarrow 1$   
 $3 \rightarrow 3$   
 $4 \rightarrow 3$   
 $5 \rightarrow 5$

So now:  $\{1,2,3,4\}, \{5\}$

ii. Optimizations:

✓ **Union by rank/size:** attach smaller tree under bigger tree.

✓ **Path compression:** during FIND, make every node on the path point directly to the root for faster future queries.

## 4.7 Counting Binary Trees

How many binary trees can you form with  $n$  nodes?

The number of **distinct binary trees with  $n$  nodes** is given by the **Catalan number**:

Closed-form formula (Catalan Number):

$$C_n = \frac{(2n)!}{(n+1)! \cdot n!}$$

Recursive formula:

$$C_0 = 1, \quad C_n = \sum_{i=0}^{n-1} C_i \cdot C_{n-1-i}$$

i. Example:

**Example: Counting Binary Trees using Catalan Numbers**

- For  $n = 0$ :

$$C_0 = 1 \quad (\text{Empty tree})$$

- For  $n = 2$ :

$$C_2 = C_0 \cdot C_1 + C_1 \cdot C_0 = 1 \cdot 1 + 1 \cdot 1 = 2$$

- For  $n = 3$ :

$$\begin{aligned} C_3 &= C_0 \cdot C_2 + C_1 \cdot C_1 + C_2 \cdot C_0 \\ C_3 &= 1 \cdot 2 + 1 \cdot 1 + 2 \cdot 1 = 2 + 1 + 2 = 5 \end{aligned}$$

So, there are 5 distinct binary trees with 3 nodes.

ii. Example trees with 3 nodes:

- Root with left subtree of 2 nodes, right subtree empty
- Root with right subtree of 2 nodes, left subtree empty
- Root with 1 node left & 1 node right
- etc.

### iii. Summary Table

Concept	Key Idea
Forests	Collection of disjoint trees
Disjoint Sets	Manage sets with no overlap, support FIND/UNION
Counting Binary Trees	Catalan number counts number of distinct binary trees

## 4.8 Introduction to Graphs

A **graph** holds:

- **Vertices (nodes)**: points which is mentioned in the graph.
- **Edges (links)**: connections between vertices.

Graphs can represent networks, relationships, maps, etc.

 Example graph:

```
  A
 /  \
B---C
 \  /
  D
```

Vertices: {A, B, C, D}

Edges: {(A,B), (A,C), (B,C), (B,D), (C,D)}

## 4.9 Graph Abstract Data Type (ADT)

A **Graph ADT** defines a set of operations to work with a graph, independent of how it's implemented.

We define a graph  $G = (V, E)$  where:

- V is the set of vertices
- E is the set of edges (pairs of vertices)

### ADT Operations:

- ✓ CreateGraph() — create an empty graph
- ✓ AddVertex(v) — add vertex v
- ✓ AddEdge(u,v) — add edge between vertices u and v
- ✓ RemoveVertex(v) — remove vertex and its edges
- ✓ RemoveEdge(u,v) — remove edge
- ✓ Adjacent(u,v) — check if edge between u and v exists
- ✓ Neighbors(v) — return all vertices adjacent to v
- ✓ GetVertices() — return set of vertices
- ✓ GetEdges() — return set of edges

### Graph Representations:

Graphs are commonly implemented as:

#### ✓ Adjacency Matrix:

A  $V \times V$  matrix where  $\text{matrix}[i][j]=1$  if there's an edge between i and j.

Example:

For vertices {A,B,C} and edges {(A,B), (B,C)}:

	A	B	C
A	[ 0	1	0 ]
B	[ 1	0	1 ]
C	[ 0	1	0 ]

#### ✓ Adjacency List:

Each vertex has a list of adjacent vertices.

Example:

A → B  
B → A, C  
C → B

#### ✓ Edge List:

Simply a list of edges:

(A,B), (B,C)

## 4.10 Elementary Graph Operations

These are the basic actions you can perform on a graph:

### a) Adding a Vertex

Add a new vertex E:

Before:

Vertices: {A,B,C,D}

After:

Vertices: {A,B,C,D,E}

### b) Adding an Edge

Add an edge between two vertices:

Before:

Edges: {(A,B), (B,C)}

After adding (C,D):

Edges: {(A,B), (B,C), (C,D)}

### c) Removing a Vertex

Remove a vertex and all edges connected to it:

Before:

Vertices: {A,B,C,D}

Edges: {(A,B), (B,C), (C,D)}

Remove C:

Vertices: {A,B,D}

Edges: {(A,B)}



#### d) Removing an Edge

Remove a specific edge:

Before:

Edges:  $\{(A,B), (B,C), (C,D)\}$

Remove (B,C):

Edges:  $\{(A,B), (C,D)\}$

#### e) Checking Adjacency

Check if two vertices are directly connected:

$\text{Adjacent}(B,C) \rightarrow \text{True}$

$\text{Adjacent}(A,D) \rightarrow \text{False}$

#### f) Finding Neighbors

Find all vertices connected to a given vertex:

$\text{Neighbors}(B) \rightarrow \{A,C\}$

### Summary Table of Graph ADT Operations

Operation	Purpose
CreateGraph()	Initialize an empty graph
AddVertex(v)	Add a vertex
AddEdge(u,v)	Connect two vertices
RemoveVertex(v)	Delete vertex and its edges
RemoveEdge(u,v)	Delete a specific edge
Adjacent(u,v)	Check if u & v both are connected
Neighbors(v)	List all adjacent vertices

### Applications of Graphs:

- ✓ Social networks (friends/followers)
- ✓ Computer networks
- ✓ Maps and navigation
- ✓ Project scheduling (DAGs)
- ✓ Circuit design

## MODULE 5

**Hashing:** Introduction, Static Hashing, Dynamic Hashing

**Priority Queues:** Single and double ended Priority Queues, Leftist Trees

**Introduction To Efficient Binary Search Trees:** Optimal Binary Search Trees

### 5 Introduction to Hashing

#### i. What is Hashing?

**Hashing** is a technique to **store and retrieve data efficiently in constant time** (on average).

We use a **hash function** to map a key (like a number or name) to an index in a table (called a **hash table**).

#### Example:

We have keys: 25, 42, 96, 77

If the hash table has size 10 and we use:

$$h(k) = k \bmod 10$$

then:

- $25 \rightarrow 5$
- $42 \rightarrow 2$
- $96 \rightarrow 6$
- $77 \rightarrow 7$

We store the keys at these indices.

#### ii. Why use hashing?

- ✓ Fast insert/search/delete —  $O(1)$  average time.
- ✓ Useful in databases, compilers, caches, symbol tables, etc.

## 5.1 Static Hashing

### What is Static Hashing?

In **static hashing**, the size of the hash table and the hash function are fixed in advance. We cannot change the table size even if the number of keys grows.

#### Example:

Hash table of size 10, with hash function:

$$h(k) = k \bmod 10$$

We insert keys: 23, 44, 12, 67, 34, 56

Table:

0:

1:

2: 12

3: 23

4: 44, 34

5:

6: 56

7: 67

8:

9:

Here we see **collisions** — e.g., 44 and 34 both map to 4.

#### EXPLANATION:

We're building a **hash table of size 10**, using the hash function:

$$h(k) = k \bmod 10$$

That means:

For any key  $k$ , we compute  $k \bmod 10$  (the remainder when  $k$  is divided by 10).

We store  $k$  at that index in the table.

### Keys to insert:

23, 44, 12, 67, 34, 56

### Hash Table:

It has **10 slots**: from 0 to 9.

Step by step:

#### Insert 23:

Compute:  $23 \bmod 10 = 3$

So, put 23 in slot 3.

3: 23

#### Insert 44:

Compute:  $44 \bmod 10 = 4$

So, put 44 in slot 4.

3: 23

4: 44

#### Insert 12:

Compute:  $12 \bmod 10 = 2$

So, put 12 in slot 2.

2: 12

3: 23

4: 44

#### Insert 67:

Compute:  $67 \bmod 10 = 7$

So, put 67 in slot 7.

2: 12  
3: 23  
4: 44  
7: 67

Insert 34:

Compute:  $34 \bmod 10 = 4$   
 $34 \setminus \bmod 10 = 3$

But slot 4 already has 44.

➡ **Collision happens.**

We store both in slot 4 using **chaining** (linked list at that slot).

**4: 44 → 34**

Insert 56:

Compute:  $56 \bmod 10 = 6$   
 $56 \setminus \bmod 10 = 5$

So, put 56 in slot 6.

**6: 56**

**Final Hash Table:**

Slot	Keys
0	
1	
2	12
3	23
4	44 → 34
5	
6	56

Slot	Keys
7	67
8	
9	

Note:

### What happened at slot 4?

Both 44 and 34 hash to 4 ( $44 \bmod 10 = 4$ ,  $34 \bmod 10 = 4$ ), so we **chain them together at slot 4**.

This is called **collision resolution by chaining**.

- **How to handle collisions?**

✓ **Chaining:** Store colliding keys in a linked list at that index.

✓ **Open Addressing:** Search for the next free slot (linear probing, quadratic probing, double hashing).

- **Disadvantages of Static Hashing:**

✗ If too many keys → table gets overloaded → performance degrades.

✗ If too few keys → wasted space.

We can't resize the table.

## 5.2 Dynamic Hashing

- **What is Dynamic Hashing?**

Dynamic hashing is a technique where the **hash table can grow (or shrink)** automatically as the number of records increases (or decreases).

This solves the problem of **overflow** and **wasted space** in static hashing:

- Static hashing: table size is fixed → can lead to too many collisions or empty slots.
- Dynamic hashing: table adjusts its size and structure as needed.

- **Why use Dynamic Hashing?**

- ✓ Supports unlimited insertions without rehashing the entire table.
- ✓ Reduces collisions.
- ✓ Efficient use of space.

- **How does it work?**

One common dynamic hashing method is **Extendible Hashing**.

We maintain:

- A **directory** of pointers to buckets.
- Buckets contain records (keys).
- Buckets have a local depth, and the directory has a global depth.
- When a bucket overflows, it splits — and the directory may grow.

### 5.2.1 Example:

#### ◆ Setup:

- Each bucket can hold **2 keys**.
- Hash function:  $h(k)$  = binary representation of  $k$ , and we use **the least significant bits** based on depth.

Initial **global depth**: 1 → directory has  $2^1 = 2$  entries.

Buckets:

**Directory:**

0 → **Bucket0**

1 → **Bucket1**



**Insert keys: 5, 7, 1, 3**

◆ **Step 1: Insert 5**

Binary: 101 → last 1 bit = 1

Put in Bucket1.

Bucket0:

Bucket1: 5

◆ **Step 2: Insert 7**

Binary: 111 → last 1 bit = 1

Put in Bucket1.

Bucket0:

Bucket1: 5, 7

◆ **Step 3: Insert 1**

Binary: 001 → last 1 bit = 1

Bucket1 is **full**, so we need to split.

◆ **Split Bucket1:**

- Increase **global depth** to 2 → directory doubles to  $2^2=4^2=4$  entries.

Directory:

00 → Bucket0

01 → NewBucket1

10 → Bucket0

11 → NewBucket1

- Redistribute keys in Bucket1 using **2 bits**:
  - 5 (101) → 01
  - 7 (111) → 11
  - 1 (001) → 01

Bucket0:  
Bucket1 (01): 1, 5  
Bucket3 (11): 7

◆ **Step 4: Insert 3**

Binary: 011 → last 2 bits = 11  
Put in Bucket3:

Bucket0:  
Bucket1 (01): 1, 5  
Bucket3 (11): 7, 3

**Final Table:**

- ✓ Directory points to buckets properly, and no bucket exceeds its capacity.
- ✓ Further insertions may trigger more splits and directory growth.

◆ **Advantages of Dynamic Hashing:**

- ✓ Automatically handles growth.
- ✓ No need to rehash the entire table.
- ✓ Good space utilization.

Feature	Static Hashing	Dynamic Hashing
Table size	Fixed	Grows as needed
Collisions	Must resolve	Fewer due to splits
Space utilization	Poor if badly sized	Better
Complexity	Simple	More complex

## 🔴 Advantages of Dynamic Hashing:

- ✓ Table adjusts to the number of keys.
- ✓ No overflow.
- ✓ Space-efficient.

### 5.3 Priority Queues: Introduction

A **Priority Queue** is a specialized form of queue in which every element is assigned a priority, and removal is determined by priority instead of the standard FIFO (First-In-First-Out) rule.

- The element with the highest (or lowest) priority is removed before others.
- When two elements share the same priority, they are processed in the order they were inserted.

Example:

We insert the elements:

(JobA, priority=3), (JobB, priority=1), (JobC, priority=2)

In a min-priority queue (lower number = higher priority), the removal order is:  
JobB → JobC → JobA

Operations:

- ✓ `insert(x, priority)` → insert an element with priority.
- ✓ `deleteMin()` or `deleteMax()` → remove the element with highest or lowest priority.

#### 5.3.1 Single-Ended vs Double-Ended Priority Queues

##### i. Single-Ended Priority Queue

- You can insert elements anywhere.
- But you can **only remove the element with the highest priority**.

Example:

Queue:

[(JobA, 3), (JobB, 1), (JobC, 2)]

- Insert: new job with priority 4
- Delete: removes JobB (priority 1)

This is the standard Priority Queue.

ii. Double-Ended Priority Queue (Deque or DEPQ)

- You can insert elements anywhere.
- And you can **remove both the minimum and maximum priority elements**.

Example:

Queue:

[(JobA, 3), (JobB, 1), (JobC, 2), (JobD, 5)]

- RemoveMin: removes JobB
- RemoveMax: removes JobD

This is useful when you need to process both extremes.

Applications:

- ✓ Single-Ended → job scheduling, shortest path algorithms.
- ✓ Double-Ended → interval problems, simulation, event scheduling.

---

## 5.4 Leftist Trees

A **Leftist Tree (or Leftist Heap)** is a type of binary tree used to implement **Priority Queues efficiently**.

i. Properties of Leftist Tree:

- ✓ It is a **binary tree**, but not strictly balanced.
- ✓ It satisfies the **heap property**: each parent has lower (or higher, for max-heap) priority than its children.
- ✓ Leftist trees are **skewed to the left**, ensuring the shortest path to a null child is always on the right.

We maintain a property called **null path length (npl)**:

- $npl(x)$  = length of shortest path from  $x$  to a node with no two children.

ii. Why use Leftist Trees?

- ✓ They allow **merge of two priority queues in  $O(\log n)$  time**.
- ✓ Insert and delete operations are also efficient.

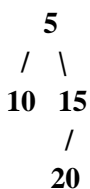
### 5.4.1 Example of Leftist Tree

We want to create a min-priority queue.

We insert: 10, 20, 5, 15

- ✓ Insert 10 → becomes root.
- ✓ Insert 20 → compare with 10, stays as right child.
- ✓ Insert 5 → 5 is smaller, becomes new root.
- ✓ Insert 15 → merged to maintain heap & leftist properties.

Final structure:



✓ Here:

- Parent  $\leq$  children (min-heap)
- Left child's npl  $\geq$  right child's npl

### 5.4.2 EXPLANATION:

#### Leftist Tree: Rules

- Heap property: Parent  $\leq$  children (min-heap).
- Leftist property: **null path length (npl) of left child  $\geq$  npl of right child.**
- Always merge in a way that keeps the tree left-heavy.

#### Step 1: Insert 10

We start with an empty tree.

Insert 10  $\rightarrow$  it becomes the root.

10

✓ Heap property is satisfied (only one element).

✓ Leftist property is satisfied.

---

#### Step 2: Insert 20

Now we merge 20 into the tree.

Compare 10 and 20: since  $10 < 20$ , keep 10 as root, and merge 20 to the right.

10  
  \  
  20

Now check null path lengths (npl):

- Left of 10  $\rightarrow$  null  $\rightarrow$  npl=0
- Right of 10  $\rightarrow$  20  $\rightarrow$  npl=0

✓ Leftist property already satisfied since both children have equal npl.

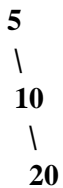
---

### Step 3: Insert 5

Now we merge 5 into the tree.

Compare 10 and 5: since  $5 < 10$ , make 5 the new root, and merge the old tree under it.

So now:

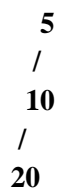


But this violates the leftist property (we prefer left-heavy trees).

So we **swap left and right subtrees at each node if needed to maintain:  $\text{npl}(\text{left}) \geq \text{npl}(\text{right})$** .

We swap 10's children (since left=null and right=20), and also at 5 we swap left & right.

Final after swapping:



- ✓ Heap property:  $5 < 10 < 20$
- ✓ Leftist property: each node's left subtree is at least as heavy as its right.

#### Step 4: Insert 15

Now we merge 15 into the tree.

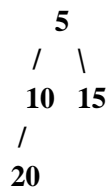
We merge 15 with root 5.

Compare 5 and 15: since  $5 < 15$ , keep 5 as root.

Then merge 15 into 5's right child (which is null).

So 15 becomes the right child of 5.

Intermediate:

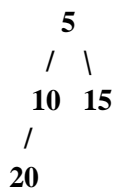


Now we must maintain leftist property.

At 5, the npl of left (10) is 1, and right (15) is 0.

✓ Already left-heavy → no need to swap.

#### Final Leftist Tree:



✓ Heap property:  $5 < 10$ ,  $5 < 15$ ,  $10 < 20$

✓ Leftist property: each node's left subtree's npl  $\geq$  right's npl.



### Summary of each step:

Step	Action
Insert 10	Becomes root
Insert 20	Merged to right of 10
Insert 5	Becomes new root, tree restructured to keep leftist property
Insert 15	Attached as right child of 5

#### 5.4.2 Where they are used?

- ✓ **Priority Queues** → CPU scheduling, Dijkstra's algorithm
- ✓ **Double-Ended PQ** → Simulation, median finding
- ✓ **Leftist Trees** → Efficient PQ implementation

## 5.5 Efficient Binary Search Trees: Optimal Binary Search Trees (OBST)

### Introduction

We know that a **Binary Search Tree (BST)** is a binary tree where:

- Left subtree contains keys  $<$  root.
- Right subtree contains keys  $>$  root.
- Searching in a balanced BST takes  **$O(\log n)$**  time.

However:

- ✓ If the BST is **not balanced**, the search time can degrade to  **$O(n)$** .

Also:

- ✓ In some applications (like compilers, dictionaries), **some keys are searched more frequently than others**.

We want to build a BST that **minimizes the expected search cost**, given the **probability of accessing each key**.

Such a tree is called an:

## Optimal Binary Search Tree (OBST)

### i. Why OBST?

Given:

- $n$  sorted keys  $K_1 < K_2 < \dots < K_n$
- $p_i$ : probability of searching for key  $K_i$
- $q_i$ : probability of searching for a value that is **not in the tree**, between keys

We want to build a BST that minimizes:

#### ✓ Expected search cost

### ii. Cost of a BST

If a key is found at depth  $d$ , and its probability is  $p_i$ , it contributes  $p_i \times (d+1)$  to the expected cost.

Similarly for unsuccessful searches.

Total expected cost  $EE$  is:

$$E = \sum_{i=1}^n p_i (\text{depth of } K_i + 1) + \sum_{i=0}^n q_i (\text{depth of dummy node} + 1)$$

### iii. Dynamic Programming Solution:

We solve it using DP in  $O(n^3)$  time (or optimized to  $O(n^2)$ ).

We compute:

- ✓  $e[i,j]$ : expected cost of OBST containing keys  $K_i, \dots, K_j$
- ✓  $w[i,j]$ : sum of probabilities  $p_i, \dots, p_j$  and dummy keys  $q_{i-1}, \dots, q_j$

We fill a DP table and pick the root of each subtree to minimize cost.

## 5.6 Example:

### Given:

Keys:  $K_1=10$ ,  $K_2=20$ ,  $K_3=30$

Probabilities:

- Successful:  $p_1=0.3$ ,  $p_2=0.2$ ,  $p_3=0.5$
- Unsuccessful:  $q_0=0.1$ ,  $q_1=0.1$ ,  $q_2=0.1$ ,  $q_3=0.1$

### Approach:

We need to decide which key to place at root and recursively which keys to place in left/right subtrees.

We try all possible roots for each subproblem:

- If  $K_1$  is root  $\rightarrow$  left: empty, right:  $K_2, K_3$
- If  $K_2$  is root  $\rightarrow$  left:  $K_1$ , right:  $K_3$
- If  $K_3$  is root  $\rightarrow$  left:  $K_1, K_2$  right: empty

For each, we compute the cost recursively and pick the minimum.

### Result:

Optimal structure for this example:

```
    30
   /
  10
   \
    20
```

Here, 3030 is chosen as root because it has the highest probability (0.5).  
1010 and 2020 are arranged to minimize the expected weighted path length.

## 5.7 Advantages of OBST:

- ✓ Minimum expected search cost
- ✓ Takes into account frequency of searches
- ✓ Useful for applications like:
  - Symbol tables in compilers
  - Dictionary lookups where some words are queried more often

## Time Complexity:

- ✓ DP approach:  $O(n^3)$
- ✓ Knuth's optimization:  $O(n^2)$

## Summary Table

Feature	Description
Input	Sorted keys & probabilities
Output	BST with minimum expected search cost
Technique	Dynamic programming
Time complexity	$O(n^2)O(n^2)$ or $O(n^3)O(n^3)$
Use cases	Dictionaries, compilers

## **MODULE 6**

### **INTERNET OF THINGS (IOT)**

#### **6.1.Arduino**

##### **6.1.1.Introduction to Arduino**

Arduino seems to an open-source microcontroller platform used for construct digital devices and interactive systems. It consists of programmable hardware boards and a user-friendly software IDE. Arduino can read inputs from sensors and control outputs like LEDs, motors, or relays. It is widely used for prototyping, IoT, robotics, and automation applications. It consists of programmable hardware boards and a user-friendly software IDE. Arduino can read inputs from sensors and control outputs like LEDs, motors, or relays. It is widely used for prototyping, IoT, robotics, and automation applications.

##### **6.1.2.Types of Arduino Boards with Pin Configurations**

###### 1. Arduino UNO

The Arduino UNO is familiarly used in board circuit , ideal for beginners. It is based on the ATmega328P microcontroller.

Digital Input & Output Pins: 14

Analog Input Pins: 6

PWM Pins: 6

UART (Serial Communication): 1

I2C: Yes

SPI: Yes

###### 2. Arduino Nano

The Arduino Nano is a compact and breadboard-friendly version of the UNO, using the same microcontroller. It is preferred for space-constrained applications.

Digital I/O Pins: 14

Analog Input Pins: 8

PWM Pins: 6

UART: 1

I2C: Yes

SPI: Yes

### 3. Arduino Mega

This is designed for larger and more complex projects needs multiple actuators and sensors . It uses the ATmega2560 microcontroller.

Digital Input and Output Pins: 54

Analog Input Pins: 16

PWM Pins: 15

UART: 4

I2C: Yes

SPI: Yes

#### **6.1.3.Parts of Arduino UNO**

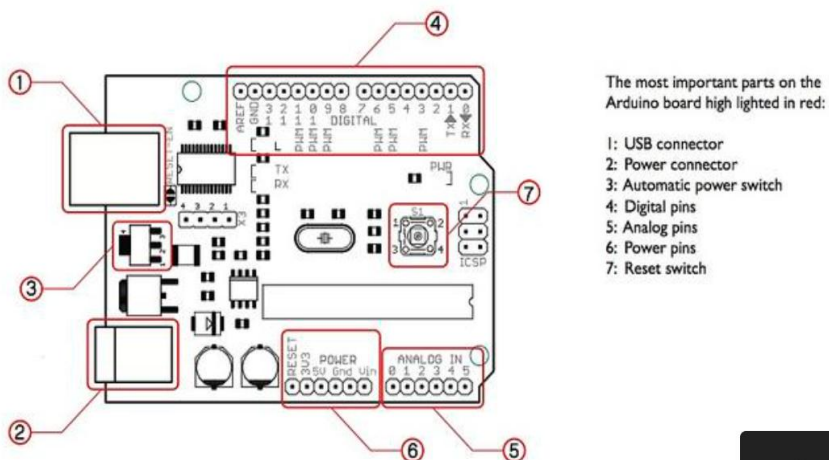


Figure 6.1

#### **6.1.4.Five Steps to program an Arduino:**

- Programs written in Arduino are known as sketches. A basic sketch consists of 3 parts

1. Declaration of Variables

2. Initialization: - written in the void setup () function.
  3. Control code: - written in the void loop () function.
- The sketch is saved with .ino extension. Any operations like verifying, opening a sketch, saving a sketch can be done using the buttons on the toolbar or using the tool menu.
  - The sketch should be stored in the sketchbook directory.
  - Choose the proper board from the tools menu and the serial port numbers.
  - Click on the Upload button or choose Upload from the Tools menu. USB cable helped to install the program code to the microcontroller through bootloader.

#### 6.1.5.Few of basic Adruino functions are:

**digitalRead(pin):** Reads the digital value at the given pin.

**digitalWrite(pin, value):** Writes the digital value to the given pin.

**pinMode(pin, mode):** Sets the pin to input or output mode.

**analogRead(pin):** Reads and returns the value.

**analogWrite(pin, value):** Writes the value to that pin.

**serial.begin(baud rate):** Sets the beginning of serial communication by setting the bit rate.

### 6.2.Some Basic Projects

#### 6.2.1.Train Traffic Light

A train traffic light with ultrasonic sensors and two sensors can be used to detect the presence of trains and control signals. The first sensor detects a train approaching, turning the signal green to allow the train to pass. The second sensor detects when the train has passed, turning the signal red to stop traffic. This setup ensures safe train movement and prevents accidents by controlling the lights based on sensor inputs.

#### Components and its usage:

1. Arduino Board (e.g., Arduino Uno): The central controller that processes sensor data and controls the LEDs.
2. 2 Ultrasonic Sensors (e.g., HC-SR04): Measure the distance to detect an approaching or departing train.

3. Red LED: Represents the stop signal, turning on when the train is detected or has passed.
4. Green LED: Represents the go signal, blinking when the train is approaching.
5. Jumper Wires: Used to make electrical connections between the Arduino, sensors, and LEDs.

### **Circuit Diagram:**

The circuit diagram is illustrated in figure 6.2

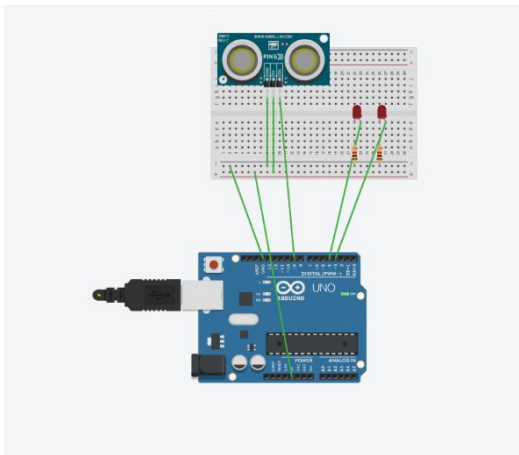


Figure 6.2

### **program:**

```
const int trigPin = 9; // Trigger pin for Ultrasonic sensor
const int echoPin = 10; // Echo pin for Ultrasonic sensor
const int redLed = 3; // Red LED
const int greenLed = 4; // Green LED

void setup() {
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
  pinMode(redLed, OUTPUT);
  pinMode(greenLed, OUTPUT);
}
```



```

    Serial.begin(9600); // Open serial monitor to see distance readings
}

void loop() {
    long duration;
    int distance;

    digitalWrite(trigPin, LOW);
    delayMicroseconds(2);
    digitalWrite(trigPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(trigPin, LOW);
    duration = pulseIn(echoPin, HIGH);
    distance = duration * 0.034 / 2;

    Serial.print("Distance: ");
    Serial.print(distance);
    Serial.println(" cm");

    // Blink Green LED if a train is detected within 50 cm
    if (distance > 0 && distance <= 50) {
        digitalWrite(redLed, LOW); // Turn off red
        blinkLed(greenLed);        // Blink green
    } else {
        digitalWrite(greenLed, LOW); // Turn off green
        blinkLed(redLed);            // Blink red
    }

    delay(500); // Wait for 500 ms before the next reading
}

void blinkLed(int ledPin) {

```

```
digitalWrite(ledPin, HIGH); // Turn LED on
delay(300);                // Wait for 300 ms
digitalWrite(ledPin, LOW); // Turn LED off
delay(300);                // Wait for 300 ms
}
```

### **Hands on work**

The Hands on work is shown in figure 6.3

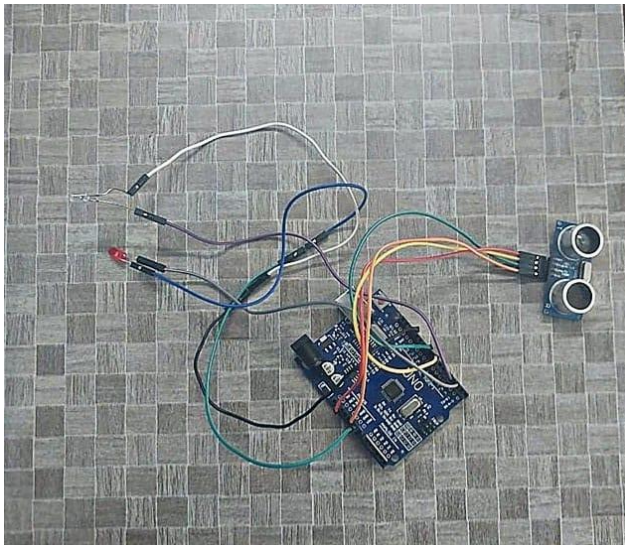


Figure 6.3

### **6.2.2. Tunnel Light Automation**

Tunnel light automation uses sensors to detect ambient light and vehicle movement, adjusting lighting accordingly for safety and energy efficiency. It employs light sensors and motion detectors to control LEDs, ensuring optimal illumination during low visibility. This system conserves energy by reducing lighting when it's not needed. It enhances road safety in tunnels by providing adequate lighting based on real-time conditions.

#### **Components and its usage:**

1. **Arduino Uno:** The main microcontroller board that controls the entire system.
2. **LEDs:** Simulate tunnel lights, turned on or off by the relay based on light conditions.

3. IR sensor: Detects the presence or movement of objects using infrared light, triggering corresponding actions in automation systems.

### Circuit Diagram:

The circuit diagram is illustrated in figure 6.4

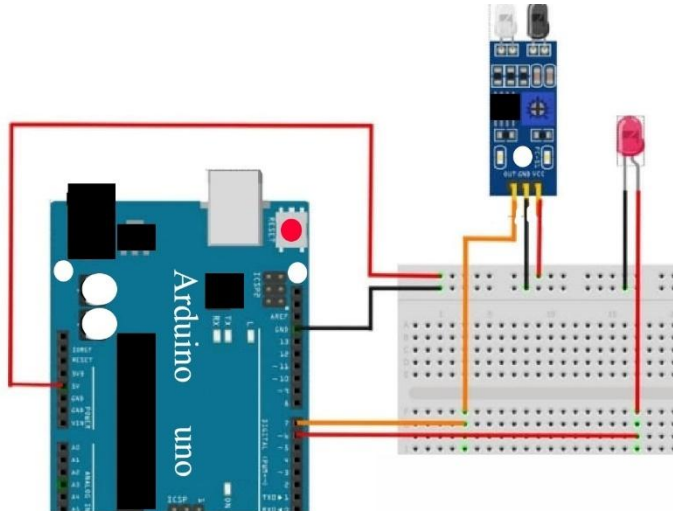


Figure 6.4

### Program:

```
const int ledPin = 6; // LED connected to digital pin 6
const int irPin = 7; // IR sensor connected to digital pin 7

void setup() {
  pinMode(ledPin, OUTPUT); // Set LED pin as output
  pinMode(irPin, INPUT); // Set IR sensor pin as input
}

void loop() {
  int irState = digitalRead(irPin); // Read state of the IR sensor

  if (irState == LOW) { // When the IR sensor detects an object (LOW is common for
    some IR sensors)

    digitalWrite(ledPin, HIGH); // Turn on the LED
  } else {
```

```
digitalWrite(ledPin, LOW); // Turn off the LED  
}  
  
delay(100); // Small delay for stability  
}
```

### **Hands on work:**

The Hands on work is shown in figure 6.5

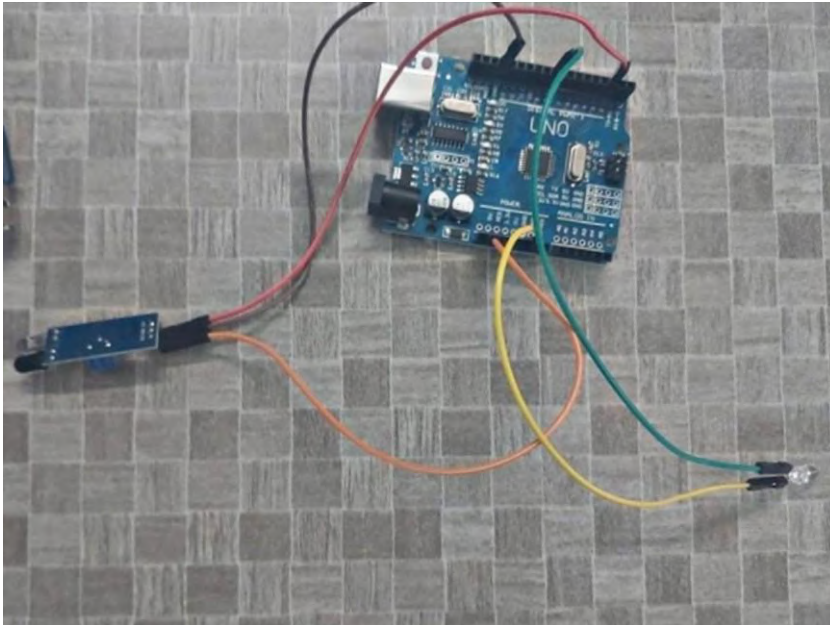


Figure 6.5

### **6.2.3.Flame Detection System**

A flame detection sensor system used to detect a flame or fire by sensing infrared (IR) radiation. When a flame is detected, the system triggers alarms or safety measures to prevent damage. It's commonly used in fire alarms, safety systems, and industrial environments. This system ensures early detection and response to potential fire hazards.

#### **Components and its usage:**

1. Arduino Uno: The main microcontroller that processes sensor data and controls the output devices
2. Flame Sensor: Detects infrared radiation emitted by flames and sends a signal to the Arduino.

3. Buzzer: Sounds an audible alarm when a flame is detected to alert nearby individuals.
4. Jumper Wires: Used to connect all the components together in the circuit.

### Circuit Diagram:

The circuit diagram is illustrated in figure 6.6

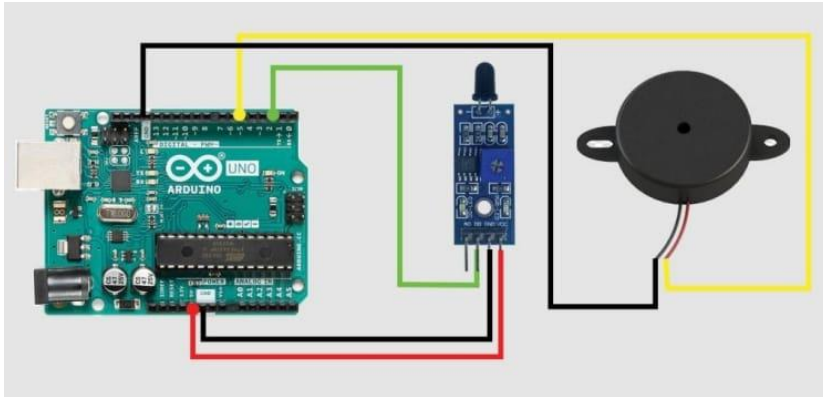


Figure 6.6

### Program:

```
const int flameSensorPin = 2; // Flame sensor connected to digital pin 2
const int buzzerPin = 5;    // Buzzer connected to digital pin 5

void setup() {
  pinMode(flameSensorPin, INPUT);
  pinMode(buzzerPin, OUTPUT);
  Serial.begin(9600);
}

void loop() {
  int flameState = digitalRead(flameSensorPin); // Read flame sensor state
  Serial.println(flameState); // Print the flame sensor state to the Serial Monitor
  if (flameState == LOW) { // Flame detected
    tone(buzzerPin, 1000); // Sound buzzer continuously
    Serial.println("Flame detected!");
  }
}
```

```

} else { // No flame detected

  noTone(buzzerPin); // Stop buzzer

  Serial.println("No flame detected.");
}

delay(500); // Delay for stability
}

```

### **Hands on project:**

The Hands on work is shown in figure 6.7

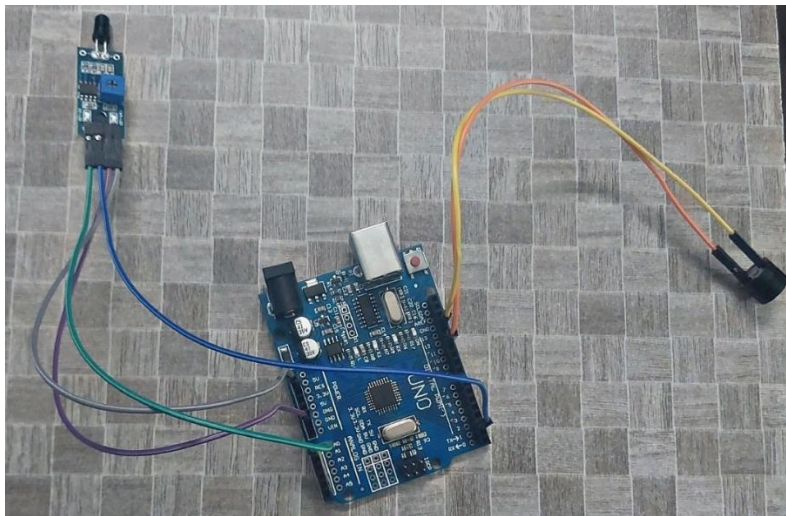


Figure 6.7

#### **6.2.4.Solar Tracker**

A solar tracker using 2 LDRs and Arduino with a servo motor is a device designed to maximize the energy absorbed by solar panels and randomly adjusting their orientation/panel to follow the source like sun's movement across the sky. The system uses Light Dependent Resistors (LDRs) to detect power centre point of sunlight and a servo motor to reposition the solar panel.

#### **Components and its usage:**

1. Arduino Board (UNO): Acts as the brain of the system, controlling the servo motor based on input from the LDRs.
2. Two LDRs (Light Dependent Resistors): Sensors that measure the light intensity from two different directions.

3. Servo Motor: Adjusts the position of the solar panel to align it optimally with the sunlight.

### Circuit Diagram:

The circuit board diagram is illustrated in figure 6.8

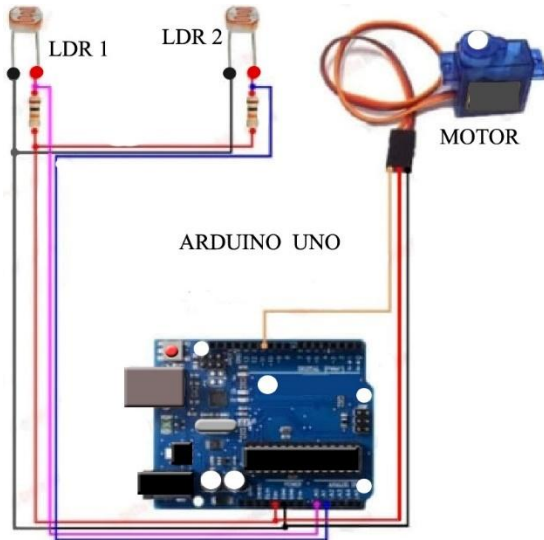


Figure 6.8

### Program:

```
#include<Servo.h>

Servo servo;

int ldr1 = A0;
int ldr2 = A1;

int servoAngle = 90;
int threshold = 90;
int stepSize = 5;

void setup() {
  servo.attach(11);
  servo.write(servoAngle);
  Serial.begin(9600);
```

```

}

void loop() {
  int leftLdr=analogRead(ldr1);
  int rightLdr=analogRead(ldr2);
  Serial.print("Left LDR:");
  Serial.print(leftLdr);
  Serial.print("||Right LDR");
  Serial.print(rightLdr);
  if(leftLdr > rightLdr + threshold)
  {
    servoAngle = constrain(servoAngle - stepSize ,0,180);
    Serial.println("Moving left");
  }
  else if(rightLdr > leftLdr + threshold)
  {
    servoAngle = constrain(servoAngle + stepSize ,0,180);
    Serial.println("Moving right");
  }
  else
  {
    Serial.println("Balanced light");
  }
  servo.write(servoAngle);
  Serial.println("servoAngle:");
  Serial.println(servoAngle);
  delay(200);
}

```



}

### **Hands on work:**

The Hands on work is shown in figure 6.9



Figure 6.9

### **6.2.5. Fire Detection System**

A Fire Detection System using an LCD, Fire Sensor, and Buzzer with 4-line output is designed to monitor and alert users about potential fire hazards. This system typically uses a Fire Sensor (such as the MQ series sensor) to detect the presence of smoke or fire, and an LCD to display the system status. A Buzzer is used to provide an audible alert when fire is detected

#### **Components and its usage:**

1. Arduino: Acts as the microcontroller that processes data from the fire sensor and controls outputs like the LCD and LED.
2. Jumper Wires: Connect components (sensors, LEDs, LCD) to the Arduino without soldering.
3. LCD (Liquid Crystal Display): Displays messages or status like "Fire Detected" or sensor readings.
4. Fire Sensor: Detects fire or flames by sensing infrared radiation (usually a flame sensor module).
5. Buzzer: Emits a loud sound to alert occupants of a potential fire hazard.

### Circuit diagram:

The circuit diagram is illustrated in figure 6.10

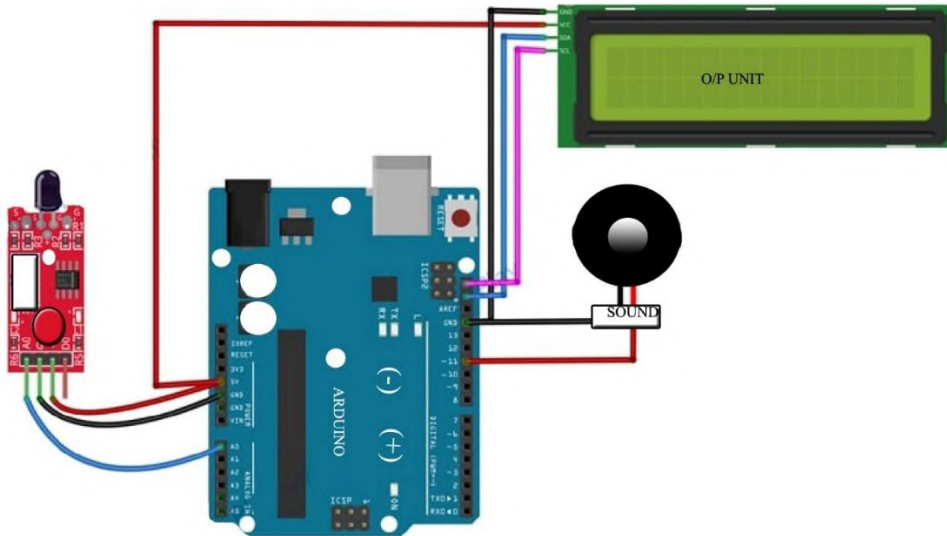


Figure 6.10

### Program :

```
#include <Wire.h>
```

```
#include <LiquidCrystal_I2C.h> // Set the LCD address to the one found using the I2C scanner
```

```
LiquidCrystal_I2C lcd(0x27, 16, 2); // Replace 0x27 with your LCD's I2C address
```

```
const int flameSensorPin = 2;
```

```
const int buzzerPin = 8;
```

```
void setup() {
```

```
  lcd.init();
```

```
  lcd.backlight();
```

```
  pinMode(flameSensorPin, INPUT);
```

```
  pinMode(buzzerPin, OUTPUT);
```

```
  lcd.print("Fire Detection");
```

```

delay(2000); // Display welcome message for 2 seconds

lcd.clear();

}

void loop() {

  int flameStatus = digitalRead(flameSensorPin);

  lcd.setCursor(0, 0);

  lcd.print("Flame Status: ");

  if (flameStatus == LOW) {

    lcd.setCursor(0, 1);

    lcd.print("Fire Detected!");

    tone(buzzerPin, 1000); // Play a 1000Hz tone on the buzzer pin

  } else {

    lcd.setCursor(0, 1);

    lcd.print("No Fire      ");

    noTone(buzzerPin); // Stop playing the tone

  }

  delay(500); // Update every half second

}

```

### **Hands on work :**

The Hands on work is shown in figure 6.11

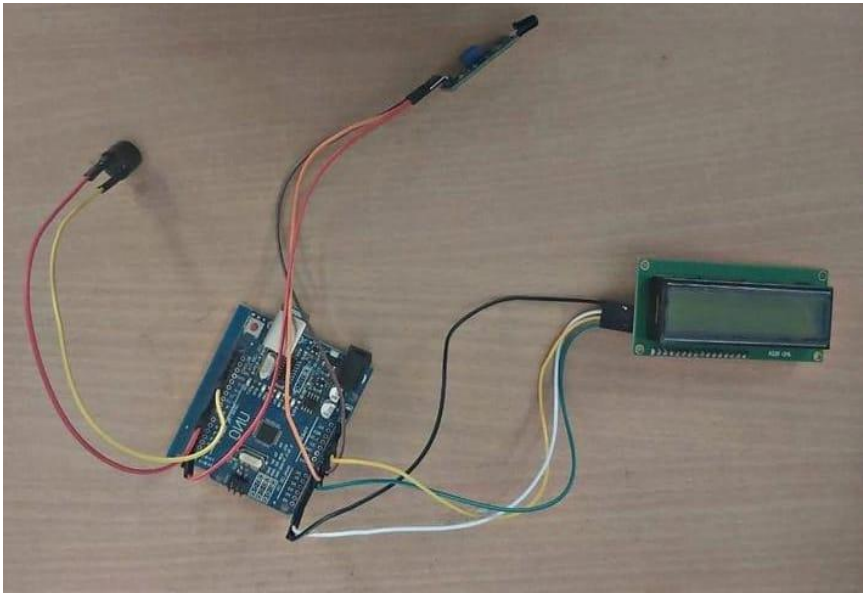


Figure 6.11

### 6.2.6. Temperature And Humidity Sensing System

A temperature and humidity sensing system uses sensors to monitor and record environmental conditions. These systems employ sensors like DHT11 to measure temperature and humidity levels. The data collected can be displayed on an LCD screen, stored for analysis, or used to trigger other devices. It's essential for applications such as climate control, agriculture, and environmental monitoring.

#### Components and its usage:

1. Arduino Uno: The main microcontroller board that processes data from the sensors and controls the display.
2. DHT11 Sensor: Measures temperature and humidity levels in the environment.
3. Liquid Crystal I2C LCD: Displays the temperature and humidity readings for easy monitoring.
4. Jumper Wires: Connect all components together to form the circuit.

#### Circuit Diagram:

The circuit diagram is illustrated in figure 6.12

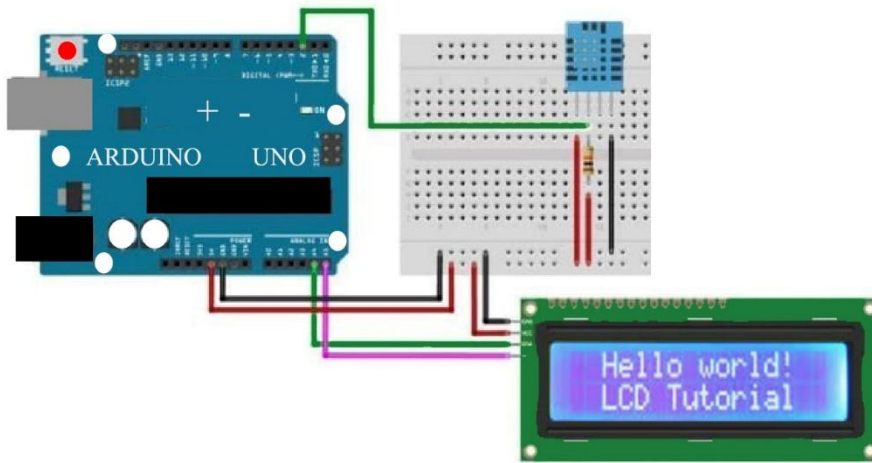


Figure 6.12

**Program:**

```
#include <Wire.h>

#include <LiquidCrystal_I2C.h>

#include <DHT.h>

#define DHTPIN 7    // Pin where the DHT11 is connected
#define DHTTYPE DHT11 // DHT 11

DHT dht(DHTPIN, DHTTYPE);

LiquidCrystal_I2C lcd(0x27, 20, 4); // set the LCD address to 0x27 for a 20x4 display

void setup() {
    lcd.init();           // initialize the lcd
    lcd.backlight();
    dht.begin();
}

void loop() {
    delay(2000); // Wait for 2 seconds between measurements
    float humidity = dht.readHumidity();
```

```

float temp = dht.readTemperature();
if (isnan(humidity) || isnan(temp)) {
    lcd.setCursor(0, 0);
    lcd.print("Sensor error");
    return;
}
lcd.setCursor(0, 0);
lcd.print("Humidity: ");
lcd.print(humidity);
lcd.print(" %");
lcd.setCursor(0, 1);
lcd.print("Temp: ");
lcd.print(temp);
lcd.print(" *C");
}

```

### **Hands on work:**

The Hands on work is shown in figure 6.13

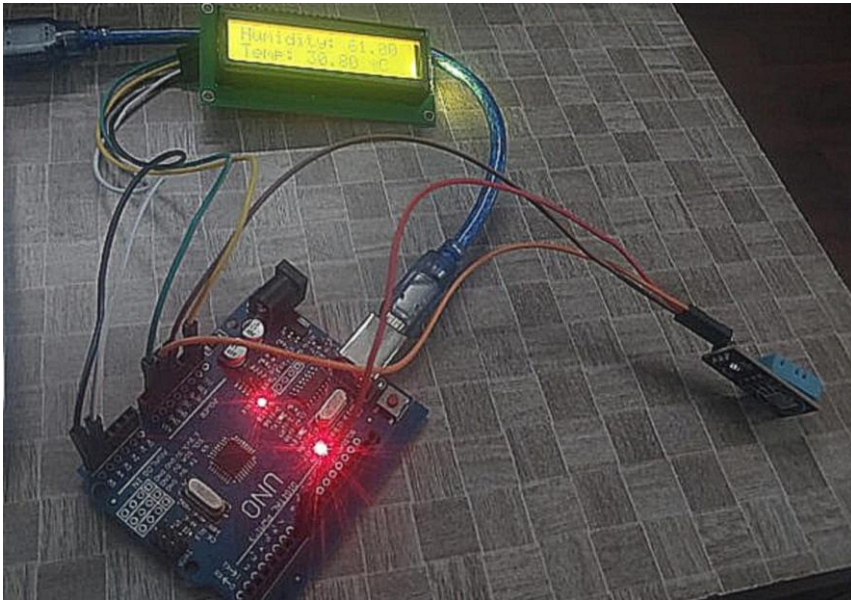


Figure 6.13

### 6.2.7. Flood Monitoring System

A Flood Monitoring System using the Esp8266 is an Iot-based project that uses sensors to detect rising water levels. The ESP8266 processes sensor data and sends alerts via SMS or IoT platforms. Key components include the ESP8266 module, ultrasonic sensor, GSM module, and an LED indicator. The system provides real-time monitoring and early warnings to mitigate flood risks.

#### Components and its usage:

1. ESP8266 NodeMCU: The main controller.
2. HC-SR04 Ultrasonic Sensor: Measures distance.
3. LED: Indicator light.
4. Jumper wires: connect components with microcontroller without soldering.

#### Circuit Diagram:

The circuit diagram is illustrated in figure 6.14

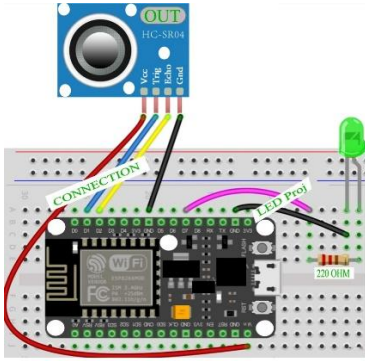


Figure 6.14

**Program:**

```
#define TRIG_PIN 4
#define ECHO_PIN 0
#define LED_PIN 2

void setup() {
  Serial.begin(9600);
  pinMode(TRIG_PIN, OUTPUT);
  pinMode(ECHO_PIN, INPUT);
  pinMode(LED_PIN, OUTPUT);
  Serial.println("Setup complete");
}

void loop() {
  long duration, distance; // Send a pulse to trigger the sensor
  digitalWrite(TRIG_PIN, LOW);
  delayMicroseconds(2);
  digitalWrite(TRIG_PIN, HIGH);
  delayMicroseconds(10);
  digitalWrite(TRIG_PIN, LOW); // Read the echo signal
  duration = pulseIn(ECHO_PIN, HIGH);
```



```

Serial.print("Duration: ");
Serial.println(duration); // Calculate the distance
distance = (duration / 2) / 29.1;
Serial.print("Distance: ");
Serial.print(distance);
Serial.println(" cm"); // Control the LED based on the distance
if (distance > 0 && distance < 10) {
    digitalWrite(LED_PIN, HIGH); // Turn on the LED if the water level is too high
    Serial.println("Flood Alert!");
} else {
    digitalWrite(LED_PIN, LOW); // Turn off the LED if the water level is normal
    Serial.println("Normal Conditions");
}
delay(1000); // Wait for 1 second before measuring again
}

```

### **Hands on work:**

The Hands on work is shown in figure 6.15

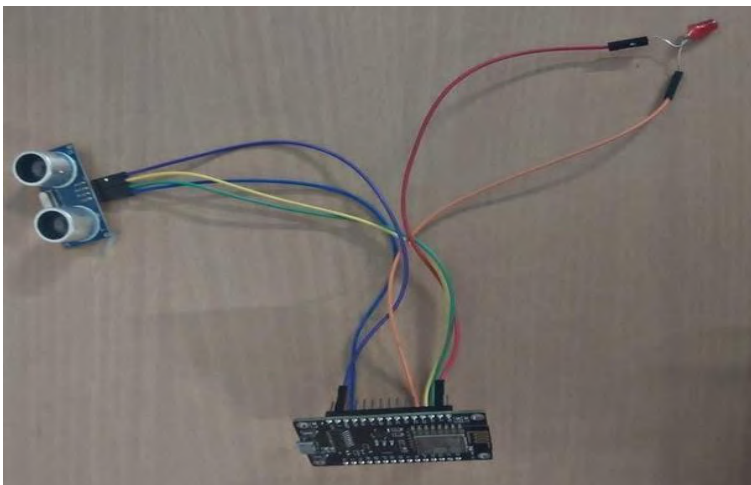


Figure 6.15

### 6.2.8. Smart Rain Shield

A rain sensor-based servo system automatically detects rainfall and activates a servo motor. When rain is detected, the servo motor rotates to deploy a protective cover. When no rain is detected, the servo retracts the cover back to its original position. This system is useful for protecting electronics, clotheslines, or open windows from rain.

#### Components and its usage:

1. Arduino UNO – microcontroller board
2. Rain Sensor Module – to detect rain
3. Servo Motor (e.g., SG90) – to move the cover
4. Jumper Wires – for connections
5. Breadboard or base (optional) – for neat wiring
6. USB Cable – to upload code and power the Arduino

#### Circuit Diagram :

The circuit diagram is illustrated in figure 6.16

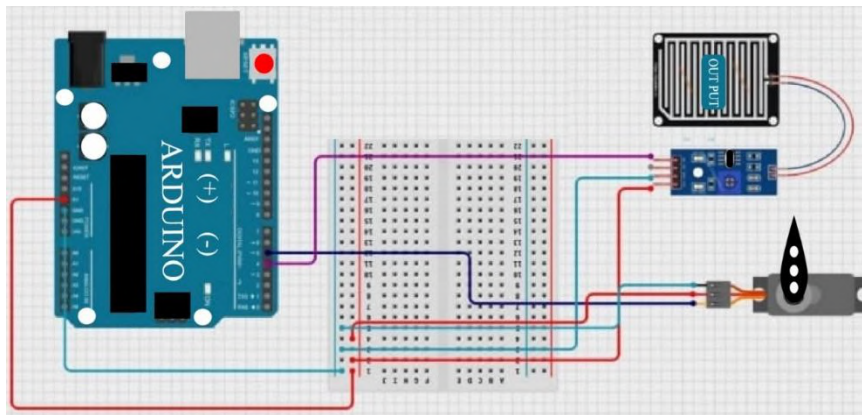


Figure 6.16

#### Program:

```
#include <Servo.h>

Servo myServo;

int rainSensorPin = 2; // Connected to D0 of the rain sensor
int rainStatus = 0;

int servoOpenPosition = 90; // Cover deployed
```

```

int servoClosePosition = 0; // Cover retracted

void setup() {
  pinMode(rainSensorPin, INPUT);
  myServo.attach(9);          // Servo signal pin to D9
  myServo.write(servoClosePosition); // Start with cover retracted
  Serial.begin(9600);
}

void loop() {
  rainStatus = digitalRead(rainSensorPin);
  if (rainStatus == LOW) { // LOW means rain is detected
    Serial.println("Rain detected! Deploying cover...");
    myServo.write(servoOpenPosition);
  } else {
    Serial.println("No rain. Retracting cover...");
    myServo.write(servoClosePosition);
  }
  delay(500);
}

```

### **Hands on work:**

The Hands on work is shown in figure 6.17

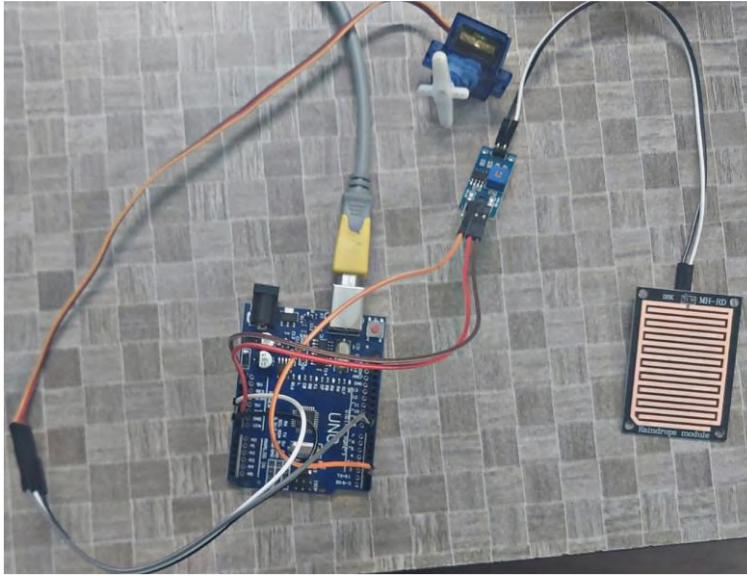


Figure 6.17

### 6.2.9.Edge Detector Robot

vehicle that uses IR sensors to detect edges (like the end of a table). It moves forward until the sensor senses a drop (no surface beneath). When the IR sensor detects no reflection (edge), the robot stops or turns. This helps prevent the robot from falling off elevated platforms.

#### Components Used:

1. Arduino UNO – Main controller
2. L298N Motor Driver Module – Controls the motors
3. IR Sensor Module – Detects the edge (table end)
4. DC Geared Motors (2 pcs) – Drives the robot
5. Wheels (2 pcs) – Connected to the motors
6. Caster Wheel (1 pc) – Balances the robot front
7. Robot Chassis Board – Base to mount all parts
8. Battery Holder (4xAA) – Power supply
9. AA Batteries (4 pcs) – Power source

10. Jumper Wires – For connections

11. USB Cable – Uploads code to Arduino

**Program:**

```
// Motor A pins
int ENA = 9;
int IN1 = 8;
int IN2 = 7;

// Motor B pins
int ENB = 3;
int IN3 = 5;
int IN4 = 4;
int irSensor = 2; // IR sensor pin

void setup() {
  pinMode(ENA, OUTPUT);
  pinMode(IN1, OUTPUT);
  pinMode(IN2, OUTPUT);
  pinMode(ENB, OUTPUT);
  pinMode(IN3, OUTPUT);
  pinMode(IN4, OUTPUT);
  pinMode(irSensor, INPUT);
  Serial.begin(9600);
}

void loop() {
  int sensorValue = digitalRead(irSensor);
  Serial.println(sensorValue);
  if (sensorValue == HIGH) {
```

```

    moveForward(); // Surface detected
} else {
    stopMotors(); // Edge detected
    delay(1000);
    turnRight(); // Turn to avoid falling
    delay(500);
}
}

void moveForward() {
    digitalWrite(IN1, HIGH);
    digitalWrite(IN2, LOW);
    analogWrite(ENA, 150);
    digitalWrite(IN3, HIGH);
    digitalWrite(IN4, LOW);
    analogWrite(ENB, 150);
}

void stopMotors() {
    digitalWrite(ENA, LOW);
    digitalWrite(ENB, LOW);
}

void turnRight() {
    digitalWrite(IN1, LOW);
    digitalWrite(IN2, HIGH);
    digitalWrite(IN3, HIGH);
    digitalWrite(IN4, LOW);
    analogWrite(ENA, 150);

```

```
    analogWrite(ENB, 150);  
}
```

### **Hands on work:**

The Hands on work is shown in figure 6.18

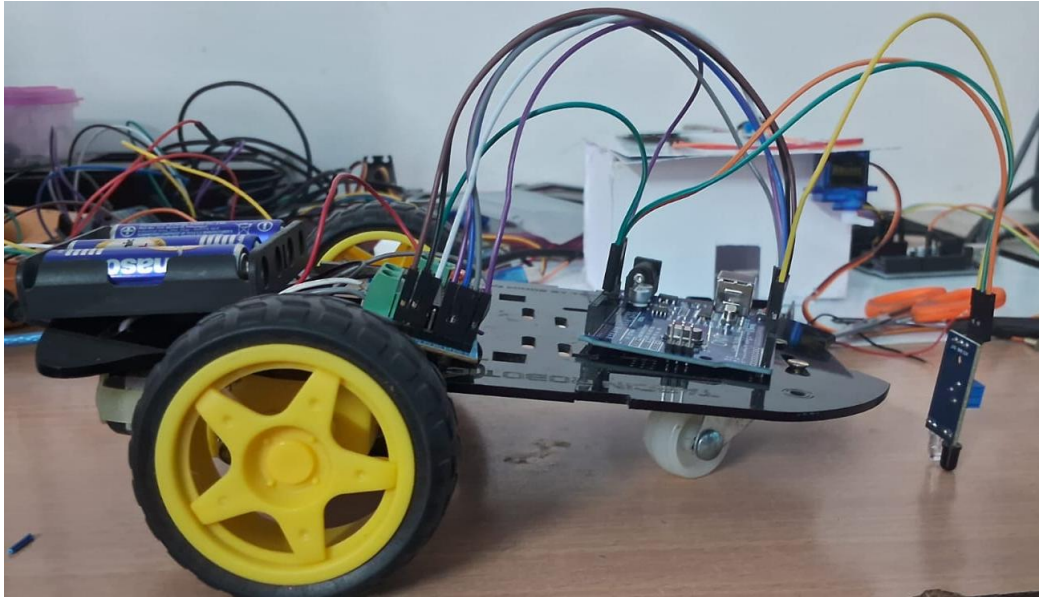


Figure 6.18

## **6.3.Raspberry Pi**

### **6.3.1.What is Raspberry Pi?**

The **Raspberry Pi** is a compact single-board computer. When peripherals such as a keyboard, mouse, and monitor are connected, it functions like a mini personal computer. It is widely used in areas like real-time image and video processing, IoT applications, and robotics. Although not as fast as a traditional laptop or desktop, the Raspberry Pi still delivers the essential features of a computer while consuming very little power.

### **6.3.2.OS for Raspberry Pi**

The **Raspberry Pi Foundation** officially offers the Raspbian OS, which is based on Debian, along with the NOOBS installer for easy setup. In addition, several third-party operating systems such as Ubuntu, Arch Linux, RISC OS, and Windows 10 IoT Core can also be installed. Raspbian, the official

operating system, is available free of cost and is specially optimized for Raspberry Pi. It comes with a graphical user interface (GUI) that provides tools for web browsing, Python programming, office applications, and games. The OS is typically stored on an SD card, with at least 8 GB recommended. Beyond functioning as a computer, Raspberry Pi allows direct access to its onboard hardware through GPIO pins, enabling the connection and control of devices such as LEDs, motors, and sensors for application development.

### **6.3.3.Raspberry Pi processor**

The **Raspberry Pi** is powered by an ARM-based Broadcom processor (SoC) integrated with an on-chip GPU for graphics processing. Its CPU speed ranges between 700 MHz and 1.2 GHz. The board includes on-board SDRAM, with capacity varying from 256 MB to 1 GB. Additionally, it supports hardware modules such as SPI, I<sup>2</sup>C, I<sup>2</sup>S, and UART for interfacing with external devices.

### **6.3.4.Versions of Raspberry pi models**

There are different versions of raspberry pi available as listed below:

1. **Raspberry Pi 1 Model A**
2. **Raspberry Pi 1 Model A+**
3. **Raspberry Pi 1 Model B**
4. **Raspberry Pi 1 Model B+**
5. **Raspberry Pi 2 Model B**
6. **Raspberry Pi 3 Model B**
7. **Raspberry Pi Zero**



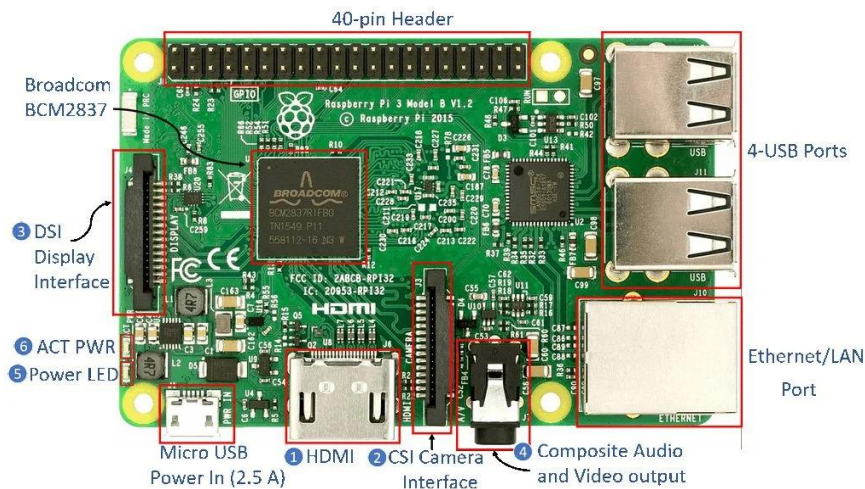


Figure 6.19

### 6.3.5. Hardware Components shown in Figure 6.10 are:

- **HDMI (High-Definition Multimedia Interface):** Used to transmit uncompressed video and digital audio signals to devices like monitors and digital TVs. This port typically connects the Raspberry Pi to a television or display screen.
- **CSI (Camera Serial Interface):** Establishes a connection between the Broadcom processor and the Pi Camera, providing the necessary electrical interface for image and video capture.
- **DSI (Display Serial Interface):** Utilized to connect an LCD screen to the Raspberry Pi via a 15-pin ribbon cable. It offers a high-resolution display connection that sends video data directly from the GPU to the LCD.
- **Composite Video and Audio Output:** Provides combined video and audio signals for connection to audio/video systems.
- **Power LED:** A red LED that indicates power status. It turns on when the Raspberry Pi is powered and blinks if the supply voltage drops below 4.63V, as it is connected directly to the 5V line.
- **ACT PWR LED:** A green LED that indicates SD card activity.

## MODULE 7

### Programming Essentials Using C

#### 1.1 What is C Programming?

A **C program** is a set of instructions written in the C programming language. C is a versatile and powerful general-purpose language developed by **Dennis Ritchie** in the 1970s at **Bell Labs**. It's commonly used to build system-level software like operating systems, embedded systems, and other applications because it is fast and efficient. As a compiled language, C translates code directly into machine instructions, making it highly performant and suitable for low-level programming.

#### 1.2 Main features of C:

- ✓ Easy to understand and write.
- ✓ Executes quickly.
- ✓ Can run on multiple platforms (portable).
- ✓ Supports the use of functions and structured coding.
- ✓ Gives access to memory using pointers.
- ✓ Acts as a foundation for many other languages (e.g., C++, Java, Python).

#### 1.3 Basic format of a C Program:

##### Example 1:

##### Program:

```
#include<stdio.h> // Library for input and output

int main(){ // Main function where execution starts

printf("Hello , World!\n"); //Display message on screen

return 0; // Exit the program successfully

}
```

### Explanation:

1. `#include<stdio.h>`

- It refers the standard library - input-output.
- It gives access to print functions like `printf()` which are used to show messages on the screen.

2. `int main()`

- This is the main entry point of the program.
- All C program starts execution through this function.

3. `printf("Hello, World ! \n");`

- Helps to prints the text "**Hello, World!**" to the display.
- The `\n` is a special character(newline) that moves the cursor to a new line after printing.

4. `return 0;`

- This statement terminates the program and sends the value **0** back to the operating system.
- It signifies that the program has run successfully without any errors.

### Example 2:

#### Add two numbers

#### Program:

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

```
int main(){
```

```
    int a, b, sum;
```

```
    printf("Enter two numbers:");
```

```
    scanf("%d %d", &a, &b);
```

```
    sum = a + b;
```

```
    printf("Sum =%d\n", sum);
```

```
    return 0;
```

```
}
```

## Output:

```
Enter two numbers: 4 5
Sum = 9
```

## Explanation:

1.#include<stdio.h>

- This line tells the compiler about standard input \output library.
- printf() to show messages and scanf() to get input.

2. int main()

- This is the entry point of the program.
- The program starting execution point starts from here.

3.int a, b, sum;

- Declares three integer variables:
- a and b helps to get and hold user input.
- sum will store the result after adding a and b.

4.printf("Enter two numbers:");

- Print a text and display it – “enter two numbers.”

5.scanf(“%d %d”, &a, &b);

- Takes input from the user and stores the values in a and b.
- %d is a placeholder for integers.
- &a and &b are **addresses** of the variables - where the input is stored.

6.sum= a + b;

- Adds a and b, and stores the results into sum.

7.printf(“Sum = %d \n”, sum);

- Shows the result(sum) on the screen.
- %d prints the integer value stored in sum.

8.return 0;

- Tells the computer that the program ended **successfully**.

### **Example 3:**

#### **Check Even or Odd:**

#### **Program:**

```
#include<stdio.h>

int main(){
    int number ;
    printf("Enter a number :");
    scanf("%d", &number);
    if(number % 2 == 0)
        printf("Even\n");
    else
        printf("Odd\n");
    return 0;
}
```

#### **Output:**

```
Enter a number: 4
Even

[Process completed - press Enter]
```

```
Enter a number: 5
Odd

[Process completed - press Enter]
```

**Explanation:**

1.#include<stdio.h>

- Adds the standard input-output header library.
- Needed to use printf() for output and scanf() for input.

2.int main()

- Initiate the main function where the program begins.

3.int number;

- Creates an integer variable named number.
- It will stores the value which is given by the user.

4.printf("Enter a number:");

- Displays the text.

5.scanf("%d" , &number);

- Reads and save.

6.if (number % 2 == 0)

- Checks if the number is even.
- % is the modulus operator, which gives the remainder after the division.
- If the number divided by 2 leaves no remainder, it is even.

7.printf("Even\n");

- If the condition is true, it prints "Even".

8.else

printf("Odd\n");

- If the number is not even, then it prints "Odd".

9. return 0;

- Ends the program and returns 0 to show success.

#### Example 4:

#### Find the Largest Two Numbers

#### Program:

```
#include<stdio.h>

int main(){
    int a, b;
    printf("Enter two numbers:");
    scanf("%d %d",&a , &b);
    if (a > b)
        printf("%d is larger\n", a);
    else if(b > a)
        printf("%d is larger\n", b);
    else
        printf("Both numbers are equal\n");
    return 0 ;
}
```

#### Output:

```
Enter two numbers: 23 45
45 is larger
[Process completed - press Enter]
```

```
Enter two numbers: 45 45
Both numbers are equal
[Process completed - press Enter]
```

**Explanation:**

1.#include<stdio.h>

- Adds the standard input/output library.
- Lets use printf() for printing and scanf() for reading input.

2.int main(){

- The program's starting point.

3.int a , b ;

- Declares two integer variables – a and b.
- These will store the numbers entered by the user.

4.printf(“Enter two numbers:”);

- Prints a message asking the user to enter two numbers.

5.scanf(“%d %d”, &a , &b);

- Reads two integer values from the user.
- Stores them into a and b.

6.if( a > b)

printf(“%d is larger\n”, a);

- If the value of a is greater than b.
- Then it will print that a is larger.

7.else if ( b > a)

printf(“%d is larger\n”, b);

- If b is greater than a, then it will print that b is larger.

8. else

printf(“Both numbers are equal\n”);

- if a and b are equal, Then it will print that both numbers are equal.

9.return 0;

- This line ends the program.



### Example 5:

#### Swap two numbers

#### Program:

```
#include<stdio.h>

int main(){

    int a, b, temp;

    printf("Enter two numbers:");

    scanf("%d %d", &a, &b);

    temp = a;

    a = b;

    b = temp;

    printf("After swapping: a= %d, b=%d\n", a, b);

    return 0;

}
```

#### Output:

```
Enter two numbers: 10 20
After swapping: a = 20, b = 10

[Process completed - press Enter]
```

#### Explanation:

1. #include<stdio.h>

- This line adds the standard input-output library.
- It allows us to use built-in functions like printf() (to show messages) and scanf() (to get input).

2.int main()

- This is the main function –the starting point of the program where execution begins.

3.int a, b, temp;

- Declares three integer variables.
- temp is used to help with swapping values.

4. printf("Enter two numbers:");

scanf("%d %d", &a, &b);

- These lines: Ask the user to type two numbers.
- Store them in variables a and b using scanf().

5.temp= a;

a = b;

b = temp;

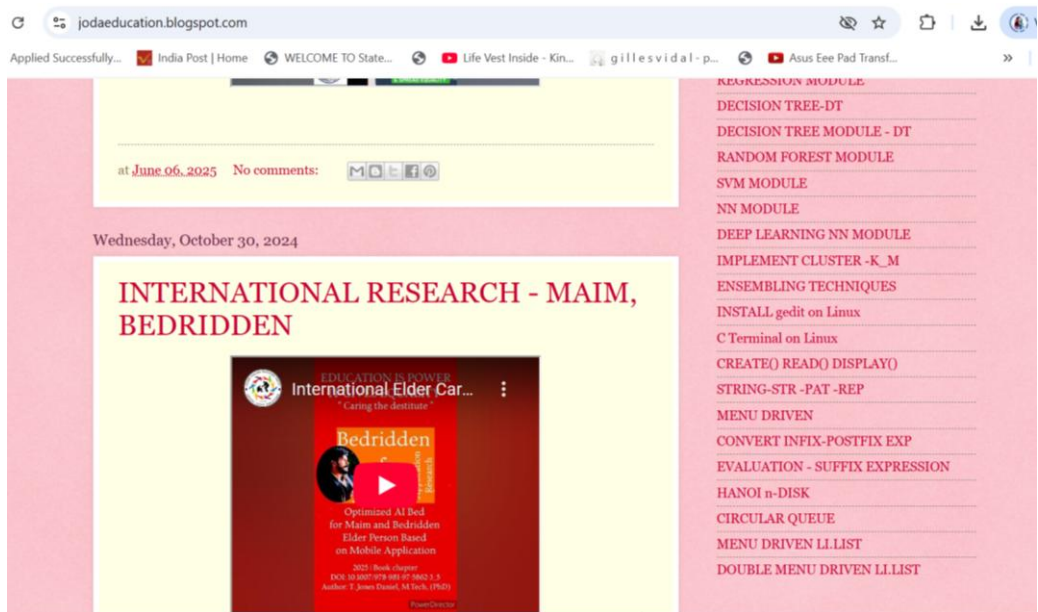
- The value in a is stored temporarily in temp.
- Then, the value in b is moved into a.
- Finally, the original value of a (which is now in temp) is moved into b.

6.printf("After swapping: a = %d, b = %d\n", a, b);

- This line displays the swapped values.

7.return 0;

- Ends the program successfully.



ONLINE WEBSITE REFERENCES

SAMPLE PROGRAM COLLECTION

REACT, ANGULAR, NODE JS, PYTHON, C PROGRAM

<https://jodaeducation.blogspot.com>