MAA OMWATI DEGREE COLLEGE HASSANPUR

(PALWAL)

NOTES

BCA 2nd SEM

DATA AND FILE STRUCTURES

Unit-1

1. Elementary Data Organization

Elementary data organization refers to how basic data elements are arranged, stored, and accessed in memory. In computing, **data organization** is critical because it directly affects how efficiently a system performs operations such as retrieval, insertion, deletion, and searching.

At the most fundamental level, data is represented using **bits and bytes**. These are grouped into **data items** such as:

- **Characters** (e.g., 'A', '3')
- **Numbers** (integers, floats)
- **Boolean values** (true/false)

These elementary data items are then structured into **data types** and **data structures** for efficient manipulation.

---

2. Data Structure: Definition

A **data structure** is a specialized format for organizing, processing, and storing data in a computer so that it can be used efficiently. It defines the **relationship** between the data elements and provides **operations** that can be performed on the data.

Formal Definition:

A data structure is a logical and mathematical model that describes the relationship between data elements, their organization, and the operations that can be performed on them.

---

3. Data Type vs Data Structure

Aspect	Data Type	Data Structure
Definition	Classification that specifies type of data (int, float, char)	Way to organize multiple data elements
Size	Single value	Group of related data elements
Operations	Basic (arithmetic, logical)	Complex (traversal, insertion, deletion)
Examples	int, float, char, bool	array, stack, queue, tree, graph
Use Case	Stores individual data items	Organizes collections of data

---

4. Categories of Data Structures

1. **Primitive Data Structures**
    - Integer, Float, Character, Boolean
2. **Non-Primitive Data Structures**
    - **Linear**: Array, Linked List, Stack, Queue
    - **Non-Linear**: Tree, Graph
3. **Static vs Dynamic**
    - **Static**: Fixed size (e.g., array)
    - **Dynamic**: Size can grow/shrink during execution (e.g., linked list)

---

5. Data Structure Operations

Each data structure supports specific operations, which include:

- **Traversal**: Access each element exactly once.
- **Insertion**: Add an element at a specific position.
- **Deletion**: Remove an element.
- **Searching**: Find a particular element.
- **Sorting**: Arrange elements in a certain order.
- **Updating**: Modify the value of an element.

---

6. Applications of Data Structures

- **Arrays**: Used in databases, spreadsheets, and lookup tables.
- **Stacks**: Expression evaluation, backtracking, recursion handling.
- **Queues**: CPU scheduling, printers, network buffers.
- **Linked Lists**: Dynamic memory allocation, polynomial representation.
- **Trees**: File systems, databases (B-Trees), compilers.
- **Graphs**: Social networks, Google Maps, communication networks.

---

7. Arrays: Introduction

An **array** is a linear data structure that stores elements of the **same data type** in **contiguous memory locations**.

Key Features:

- Fixed size
- Direct access via indexing
- Efficient for searching and sorting

---

8. Linear Arrays

Also known as **one-dimensional arrays**, they represent a list of elements arranged in a single line.

Declaration (C/C++ example):

c

CopyEdit

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

Indexing:

- Starts from 0: arr[0] = 10, arr[1] = 20, etc.

---

9. Representation in Memory

Arrays are stored in **contiguous memory blocks**. If Base_Address is the memory address of the first element and i is the index:

Address of arr[i]:

text

CopyEdit

Address(arr[i]) = Base_Address + i × Size_of_element

For example, if Base_Address = 1000 and Size_of_element = 4 bytes, then arr[3] is at 1000 + 3×4 = 1012.

---

10. Array Operations

a. Traversal

Visit each element in order.

c

CopyEdit

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

b. Insertion

Insert an element at index pos. Elements from pos onward must be shifted right.

c. Deletion

Delete an element from index pos. Elements from pos+1 onward are shifted left.

Note: Insertion and deletion operations in arrays are costly (O(n)) due to the shifting of elements.

---

11. Multidimensional Arrays

A **multidimensional array** is an array of arrays.

Example:

c

CopyEdit

```
int matrix[3][3] = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9}
};
```

Memory is still stored linearly (row-major or column-major order).

---

12. Parallel Arrays

Two or more arrays that store **related data**, where the **index** connects the relationship.

Example:

c

CopyEdit

string studentNames[3] = {"John", "Alice", "Bob"};

int studentMarks[3] = {85, 90, 78};

Here, studentNames[i] corresponds to studentMarks[i].

---

13. Sparse Arrays

A **sparse array** is a type of array in which most of the elements are **zero or empty**.

Problem:

Storing all zeroes wastes memory.

Solution:

Use **compact representations**, like:

- Coordinate list (row, column, value)
- Compressed sparse row (CSR)

Useful in:

- Machine learning (sparse matrices)
- Graph algorithms (adjacency matrices)

---

14. Searching: Introduction

Searching is the process of locating a specific element in a data structure.

Types of Searching:

1. **Sequential (Linear) Search**
2. **Binary Search**

---

15. Sequential (Linear) Search

Checks each element one by one until the desired element is found or the end is reached.

Algorithm:

c

CopyEdit

```
for (int i = 0; i < n; i++) {
   if (arr[i] == key)
      return i;
}
```

Time Complexity:

- **Best Case:** O(1) (first element)
- **Worst Case:** O(n) (last or not found)
- **Average Case:** O(n/2)

Advantages:

- Works on unsorted data

- Easy to implement

---

16. Binary Search

Efficient search method for **sorted arrays**. Repeatedly divides the search interval in half.

Algorithm:

c

CopyEdit

```
while (low <= high) {
   mid = (low + high) / 2;
   if (arr[mid] == key)
      return mid;
   else if (key < arr[mid])
      high = mid - 1;
   else
      low = mid + 1;
}
```

Time Complexity:

- **Best Case:** O(1)
- **Worst Case:** O(log n)
- **Average Case:** O(log n)

---

17. Prerequisites for Binary Search

- The array must be **sorted**.

- Binary search works only on **random access** structures like arrays (not on linked lists efficiently).

---

18. Efficiency Comparison: Linear vs Binary Search

Criteria	Linear Search	Binary Search
Data Type	Unsorted/Sorted	Sorted Only
Time (Worst)	O(n)	O(log n)
Ease	Very Easy	Slightly complex
Memory	In-place	In-place

Unit-2

1. Sorting

Sorting is the process of arranging elements in a particular order – usually either ascending or descending. Sorting plays a vital role in efficient searching, data analysis, and data presentation.

Classification of Sorting Algorithms:

- **Internal Sorting:** When sorting is done in main memory (RAM).
- **External Sorting:** Used when data is too large for memory and sorting is done using external storage.

1.1 Bubble Sort

Bubble Sort is one of the simplest sorting algorithms. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

Algorithm Steps:

1. Compare adjacent elements.
2. Swap them if they are in the wrong order.
3. Repeat the process for all elements.
4. After each pass, the largest unsorted element "bubbles up" to its correct position.

c

CopyEdit

```
for (i = 0; i < n-1; i++)
   for (j = 0; j < n-i-1; j++)
```

```
    if (arr[j] > arr[j+1])
        swap(arr[j], arr[j+1]);
```

Time Complexity:

- **Best Case:** O(n) [when array is already sorted]
- **Average and Worst Case:** $O(n^2)$

Space Complexity: O(1)

Stable: Yes

In-place: Yes

1.2 Insertion Sort

Insertion Sort builds the final sorted array one item at a time. It's efficient for small datasets and nearly sorted data.

Algorithm Steps:

1. Assume the first element is sorted.
2. Pick the next element and compare with elements in the sorted part.
3. Shift the larger elements to the right.
4. Insert the picked element in the correct position.

c

CopyEdit

```
for (i = 1; i < n; i++) {
   key = arr[i];
   j = i - 1;
   while (j >= 0 && arr[j] > key) {
```

```
        arr[j + 1] = arr[j];
        j--;
    }
    arr[j + 1] = key;
}
```

Time Complexity:

- **Best Case:** O(n)
- **Average and Worst Case:** $O(n^2)$

Space Complexity: O(1)

Stable: Yes

In-place: Yes

1.3 Quick Sort

Quick Sort is a divide-and-conquer algorithm. It picks a "pivot" and partitions the array into two parts: one with elements smaller than the pivot and the other with greater elements.

Algorithm Steps:

1. Choose a pivot.
2. Partition the array: elements < pivot go left, > pivot go right.
3. Recursively apply the same steps to sub-arrays.

c

CopyEdit

```
quickSort(arr, low, high) {
    if (low < high) {
```

```
        pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Time Complexity:

- **Best and Average Case:** O(n log n)
- **Worst Case:** O(n^2) [when array is already sorted or pivot is poorly chosen]

Space Complexity: O(log n) [due to recursion]

Stable: No

In-place: Yes

1.4 Merge Sort

Merge Sort is another divide-and-conquer algorithm. It divides the array into halves, recursively sorts them, and merges them back.

Algorithm Steps:

1. Divide the array into two halves.
2. Recursively sort each half.
3. Merge the two sorted halves.

c

CopyEdit

```
mergeSort(arr, left, right) {
    if (left < right) {
```

```
        mid = (left + right) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid+1, right);
        merge(arr, left, mid, right);
    }
}
```

Time Complexity:

- **Best, Average, Worst Case:** O(n log n)

Space Complexity: O(n) [for temporary arrays]

Stable: Yes

In-place: No

1.5 Comparison Table

Algorithm	Best Case	Avg Case	Worst Case	Space	Stable	In-Place
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes	Yes
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes	Yes
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	No	Yes
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes	No

---

2. Stack

2.1 Introduction

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

Operations on Stack:

- **Push:** Add an element to the top.
- **Pop:** Remove the top element.
- **Peek/Top:** View the top element without removing it.
- **isEmpty():** Check if the stack is empty.
- **isFull():** Check if the stack is full (for array implementation).

2.2 Array Representation of Stack

c

CopyEdit

```
#define MAX 100
int stack[MAX];
int top = -1;

void push(int x) {
   if (top >= MAX - 1) printf("Overflow");
   else stack[++top] = x;
}

int pop() {
   if (top == -1) printf("Underflow");
   else return stack[top--];
```

```
}
```

2.3 Linked List Representation of Stack

c

CopyEdit

```
struct Node {
   int data;
   Node* next;
};
Node* top = NULL;

void push(int x) {
   Node* temp = new Node();
   temp->data = x;
   temp->next = top;
   top = temp;
}
```

- No size limit.
- More dynamic and memory efficient.

2.4 Applications of Stacks

1. **Recursion:** Maintains function call stack.
2. **Expression Evaluation:**
    - **Polish (Prefix) Notation**

    - **Reverse Polish (Postfix) Notation**
3. **Undo operations in editors**
4. **Syntax parsing in compilers**
5. **Backtracking algorithms (e.g., maze solving)**

---

3. Queue

3.1 Introduction

A **Queue** is a linear structure that follows **FIFO** (First In, First Out). The first element inserted is the first to be removed.

Basic Operations:

- **Enqueue:** Add to rear
- **Dequeue:** Remove from front
- **Front:** Return front element
- **isEmpty/isFull**

3.2 Array Representation of Queue

c

CopyEdit

```
#define SIZE 100

int queue[SIZE];

int front = -1, rear = -1;


void enqueue(int x) {

   if (rear == SIZE - 1) printf("Overflow");
```

```
    else queue[++rear] = x;
    if (front == -1) front = 0;
}

int dequeue() {
    if (front == -1 || front > rear) printf("Underflow");
    else return queue[front++];
}
```

3.3 Linked List Representation of Queue

c

CopyEdit

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

Node *front = NULL, *rear = NULL;

void enqueue(int x) {
    Node* temp = new Node();
    temp->data = x;
    temp->next = NULL;
    if (rear == NULL) front = rear = temp;
```

```
    else rear->next = temp, rear = temp;
}
```

- Efficient memory usage.
- No overflow if memory is available.

---

4. Double-Ended Queue (Deque)

A **Deque (Double Ended Queue)** allows insertion and deletion from **both front and rear ends**.

Types:

- **Input-restricted deque:** Insertion at one end only.
- **Output-restricted deque:** Deletion at one end only.

Applications:

- Task scheduling
- Palindrome checking
- Sliding window algorithms

---

5. Priority Queue

A **Priority Queue** is a special queue in which each element has a priority. Elements are dequeued based on **priority**, not position.

Features:

- Higher priority = served before
- Can be implemented using:
    - Arrays
    - Linked lists

- **Heaps** (for efficiency)

Applications:

- CPU scheduling
- Dijkstra’s algorithm
- Emergency room systems

---

6. Applications of Queues

1. **CPU and Disk Scheduling**
2. **Print Spooling**
3. **Call Center Systems**
4. **Breadth-First Search in Graphs**
5. **Handling Buffers in Networking**

Unit-3

🌿 Linked List

🔹 Introduction

A **linked list** is a linear data structure where each element is a separate object, called a **node**. Each node contains:

- **Data**
- A pointer/reference to the next node

Unlike arrays, **linked lists** are not stored in contiguous memory locations. They can grow or shrink in size dynamically.

🔹 Representation in Memory

Each node in memory consists of:

c

CopyEdit

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

- data holds the value.
- next holds the address of the next node.

The linked list starts with a special pointer called head, which points to the first node. If the list is empty, head = NULL.

🔹 Traversal of Linked List

Traversal means visiting every node in the list from the start to the end.

c

CopyEdit

```
void traverse(Node* head) {
    Node* current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
}
```

🔹 Insertion in Linked List

Insertion can happen at:

1. **Beginning**
2. **End**
3. **After a specific node**

Insert at Beginning:

c

CopyEdit

```
Node* newNode = new Node();
newNode->data = value;
newNode->next = head;
head = newNode;
```

Insert at End:

Traverse to the last node and update its next to new node.

- **Deletion in Linked List**

Deletion can also occur at the beginning, end, or a specific position.

Delete at Beginning:

c

CopyEdit

```
Node* temp = head;
head = head->next;
free(temp);
```

Delete Specific Node:

Find the previous node, change its next pointer to skip the deleted node.

- **Searching in Linked List**

Linear search is used:

c

CopyEdit

```
Node* current = head;
while (current != NULL) {
    if (current->data == key) return current;
    current = current->next;
}
```

🔹 Header Linked List

A **header node** is a dummy node at the beginning of the list that does not contain meaningful data but simplifies insertions and deletions by avoiding special cases for the first node.

🔹 Circular Linked List

- The last node's next pointer points to the **first** node instead of NULL.
- Useful in applications where looping through the list indefinitely is required (e.g., CPU scheduling).

🔹 Doubly Linked List (Two-Way)

Each node contains three fields:

- **Data**
- **Previous Pointer**
- **Next Pointer**

c

CopyEdit

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

- Can be traversed in both directions.

- Insertion and deletion are more flexible but use more memory.

🔹 Threaded List

Used in **binary trees**, a threaded list is where the NULL pointers are replaced with pointers to **in-order predecessor or successor** to speed up traversals.

🔹 Garbage Collection

Garbage collection automatically reclaims memory no longer in use. In **linked lists**, when nodes are deleted, the memory must be freed explicitly (in languages like C/C++). Languages like Java handle this automatically.

🔹 Applications of Linked Lists

- Dynamic memory management
- Implementing stacks, queues, graphs
- Managing polynomials
- Real-time scheduling systems
- Undo features in text editors

🌲 Tree

🔹 Introduction

A **tree** is a hierarchical data structure consisting of nodes. The topmost node is called the **root**, and every node may have children (sub-nodes).

🔹 Binary Tree

A **binary tree** is a tree where each node has at most two children:

- **Left child**
- **Right child**

🔹 Representation of Binary Tree in Memory

1. Array Representation

- For **complete binary trees**.
- Indexing:
  - Root = index 0
  - Left child of i = 2*i + 1
  - Right child of i = 2*i + 2

2. Linked Representation

c

CopyEdit

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

🔹 Tree Traversals

Traversal = visiting each node in a particular order.

1. In-order (Left, Root, Right)

Used in BSTs to get sorted data.

2. Pre-order (Root, Left, Right)

Used to copy the tree.

3. Post-order (Left, Right, Root)

Used to delete the tree.

- **Traversal Using Stack**

Tree traversals can be implemented non-recursively using stacks.

Non-Recursive In-Order Traversal:

c

CopyEdit

```
stack<Node*> s;
Node* curr = root;

while (curr != NULL || !s.empty()) {
    while (curr != NULL) {
        s.push(curr);
        curr = curr->left;
    }
    curr = s.top(); s.pop();
    cout << curr->data << " ";
    curr = curr->right;
}
```

🔗 Graph

🔹 Introduction

A **graph** is a non-linear data structure made up of:

- **Vertices (nodes)**
- **Edges (connections)**

Graphs can be:

- **Directed or Undirected**
- **Weighted or Unweighted**

🔹 Graph Theory Terminology

- **Vertex (Node):** Fundamental unit
- **Edge:** Connection between nodes
- **Degree:** Number of edges connected to a vertex
- **Path:** A sequence of vertices
- **Cycle:** Path that begins and ends at the same vertex
- **Connected Graph:** Every node is reachable
- **Acyclic Graph:** No cycles
- **Bipartite Graph:** Can be colored using two colors such that no adjacent nodes have the same color

---

🔹 Representation of Graphs

1. Adjacency Matrix (Sequential)

- 2D matrix of size V×V
- Entry [i][j] = 1 if there is an edge from i to j

2. Adjacency List (Linked)

- Array of linked lists
- Each vertex stores a list of its adjacent vertices

c

CopyEdit

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

- **Applications of Graphs**
  - Social networks
  - Google Maps (shortest path)
  - Compiler optimizations
  - Network routing
  - Scheduling problems

Unit-4

🌐 Concept Introduction to File Structures

🔹 What is a File?

A **file** is a collection of data or information that is stored on a storage device and treated as a single unit by the operating system. Files are the fundamental building blocks for storing persistent data.

- Files can contain **text**, **binary data**, **structured records**, **unstructured data**, etc.
- Unlike variables in memory (RAM), files store data **permanently**.

📁 Types of Files

Files are classified based on structure, content, and usage:

1. Text Files

- Store data in **human-readable** format (ASCII/Unicode).
- Each line typically ends with a newline character.
- Example: .txt, .csv

2. Binary Files

- Store data in **machine-readable** form.
- Faster access and compact size.
- Example: .bin, executable files

3. Structured Files

- Data is organized in **fixed formats**, often as records (e.g., databases).

- Example: files used by DBMS, configuration files

4. Unstructured Files

- No predefined structure.
- Example: multimedia files (audio, video), documents

📂 Basic File Operations

Operations on files are similar across most programming languages:

1. Open a File

- Establishes a link between the program and the file.

python

CopyEdit

file = open("data.txt", "r") # Modes: r, w, a, rb, etc.

2. Read from a File

- Reads data from the file into memory.

python

CopyEdit

content = file.read()

3. Write to a File

- Writes data from memory into the file.

python

CopyEdit

file.write("Hello World!")

4. Close a File

- Releases resources and flushes buffers.

python

CopyEdit

file.close()

💿 External Storage Devices

Files are stored on **external or secondary storage devices**, such as:

- Hard Disk Drives (HDD)
- Solid State Drives (SSD)
- USB Flash Drives
- Optical Media (CD/DVD)
- Magnetic Tapes (for backups)

These devices are **non-volatile** and suitable for storing large volumes of data persistently.

🗂️ Concepts of Record, File, Database, and Database System

🔹 Record

- A **record** is a collection of related data fields, often representing an entity.
- Example: A student record may include ID, Name, Marks.

c

CopyEdit

struct Student {

  int id;

  char name[20];

float marks;

};

- **File**
  - A **file** is a collection of **records** stored in a specific format.
- **Database**
  - A **database** is an **organized collection** of related files and records.
  - Supports **querying, updating, and reporting**.
- **Database System**
  - Includes:
    - **Database**
    - **DBMS software**
    - **Users**
    - **Application programs**

It ensures **data integrity, security, concurrency, and recovery**.

📄 File Organization

File organization determines how records are **arranged and accessed** on the storage medium.

- **1. Sequential File Organization**
  - Records are stored one after another in a **linear sequence**.
  - Typically sorted on a **key field**.

Structure and Processing

- Easy to implement.

- Suitable for **batch processing** and **sequential access** (e.g., payroll system).

Access

- Requires reading from the beginning to the desired record.

plaintext

CopyEdit

[Record 1] → [Record 2] → [Record 3] → ... → [Record N]

Limitations

- Insertion and deletion are inefficient.
- Searching is slow (linear search).

🔹 **Record Structures and Access Methods**

- **Fixed Length Records**: All records have the same size.
- **Variable Length Records**: Record size varies depending on content.

Access Methods:

- **Sequential Access**: Read one by one
- **Direct Access**: Jump to a specific record (requires indexing or hashing)

🔹 **2. Indexed Sequential File Organization**

- Combines features of **sequential** and **direct access**.
- Maintains an **index** to quickly locate blocks of records.

Structure

- Primary index on key field.

- Data file is still sequential.
- May also include **secondary indexes**.

plaintext

CopyEdit

Index:

100 → Block 1

200 → Block 2

Data:

[100, 105, 109] → Block 1

[200, 207, 210] → Block 2

Processing

- **Fast access** using index.
- Maintains order for sequential access.

🔹 **Indexing Techniques**

- **Single-level Index**: One index for all records
- **Multi-level Index**: Indexes over indexes (like book index)
- **Dense Index**: One entry per record
- **Sparse Index**: One entry per block

🔹 **B-Trees for Indexing**

A **B-Tree** is a self-balancing tree data structure that maintains sorted data and allows **searches, sequential access, insertions, and deletions in logarithmic time**.

- Used in databases and file systems (e.g., NTFS, HFS+).
- Maintains balance and minimizes disk I/O.

🔹 **Hashing for Indexed Files**

- Converts key into a **hash value** (index).
- Uses a **hash function** to compute location.

c

CopyEdit

index = hash(key) % table_size;

- Collisions are handled using:
  - **Chaining**
  - **Open Addressing**

🔹 **3. Direct File Organization**

- Records are stored at **calculated locations**.
- No sequential ordering.
- Access via **hashing** or mathematical function on key.

plaintext

CopyEdit

record_address = base + f(key)

- Best for **real-time systems**.

- Fastest form of access.
- No need for index.

🔹 4. Hashed File Organization

A **hash function** is used to determine the storage location of a record.

Implementation

- A **hash table** is created.
- A **hash function** computes the address.

c

CopyEdit

```
int hash(int key) {
   return key % table_size;
}
```

Collision Resolution

1. **Linear Probing**: Search next slot.
2. **Quadratic Probing**: Jump in intervals.
3. **Double Hashing**: Use second hash function.

Advantages

- Fast access
- Ideal for online and transaction systems

Disadvantages

- Not good for range queries or ordered data
- May require frequent resizing