DDS 1-4 Unit

Design of Data Structures

Unit 1: Introduction to Data Structures

1.1 What are Data Structures?

A data structure is a way of organizing and storing data so that it can be accessed and worked with efficiently.

It’s not just about the data itself, but also about the relationships between data items.

Example: Imagine a library ▯. The books are the data. How they are organized on shelves (by genre, author, or a unique ID) is the data structure. This organization makes it easy to find a specific book.

1.2 Classification

Data structures are broadly categorized into two types:

Primitive Data Structures:

The most basic building blocks of data. They hold a single value.

Examples: int, float, char, boolean.

Real-life Example: Your age, 20, is a single integer value.

Non-Primitive Data Structures:

Built from primitive data structures. They can store multiple values.

Linear: Elements arranged sequentially.

Examples: Array, Stack, Queue, Linked List.

Non-Linear: Elements not in a sequence.

Examples: Tree, Graph.

1.3 Operations on Data Structures

Traversal: Visiting every element once.

Example: Going through every student’s roll number in a class list.

Insertion: Adding a new element.

Example: Adding a new contact to your phonebook.

Deletion: Removing an existing element.

Example: Deleting an old email from your inbox.

Searching: Finding a specific element.

Example: Searching for a song in your playlist.

Sorting: Arranging elements in a specific order.

Example: Sorting student marks from highest to lowest.

1.4 Arrays

An Array is a collection of elements of the same data type stored at contiguous memory locations.

Real-life Example: The seats in a cinema hall are arranged in a fixed order. Each seat has a unique number (index).

Algorithm: Traversing an Array

```
Algorithm Traverse_Array(A, n)
1. For i ← 0 to n-1 do
```

```
2.    Print A[i]
End
```

1.5 Structures & Unions

Structure: A user-defined data type that groups related data of different types under a single name.

Example: A Student structure could contain Name (string), Roll (integer), and Marks (float).

Union: Similar to a structure, but all members share the same memory location. Only one member can be accessed at a time.

Example: A storage box that can hold either a book or clothes, but not both at once.

1.6 Pointers & Dynamic Memory

Pointer: A variable that stores the memory address of another variable. It "points" to data.

Dynamic Memory Allocation: Allocating memory during program execution (runtime) using functions like malloc, calloc.

Example: An ATM dynamically allocates space for your receipt based on transaction length.

1.7 Performance Analysis

Algorithms are analyzed based on:

Time Complexity: How the running time grows with input size.

Space Complexity: Amount of memory required.

Big-O Notation Examples:

O(1) → Constant time (Accessing an array element).

O(n) → Linear time (Traversing a list).

O(log n) → Logarithmic (Binary search).

$O(n^2)$ → Quadratic (Bubble sort).

Unit 2: Stacks, Recursion and Queues

2.1 Stack

A Stack is a linear data structure that follows the LIFO (Last In First Out) principle.

Real-life Example: A stack of plates ⍰. You add/remove from the top only.

Stack Operations (Array Implementation)

PUSH (Stack, item):

```
1. If TOP = MAX-1 → Overflow
2. Else
   TOP ← TOP + 1
   Stack[TOP] ← item
```

POP (Stack):

```
1. If TOP = -1 → Underflow
```

```
2. Else
    item ← Stack[TOP]
    TOP ← TOP - 1
    Return item
```

2.2 Applications of Stack

Reversing a string

Undo/Redo in software

Expression evaluation (Infix → Postfix)

2.3 Recursion

Recursion is when a function calls itself.

Every recursive function must have a base case.

Factorial:

```
FACT(n):
1. If n = 0 return 1
2. Else return n * FACT(n-1)
```

Fibonacci Series:

```
FIB(n):
1. If n = 0 return 0
```

```
2. If n = 1 return 1
3. Else return FIB(n-1) + FIB(n-2)
```

Tower of Hanoi:

```
TOH(n, source, temp, dest):
1. If n = 1 → Move disk from source → dest
2. Else
    TOH(n-1, source, dest, temp)
    Move disk from source → dest
    TOH(n-1, temp, source, dest)
```

2.4 Queue

A Queue follows FIFO (First In First Out).

Real-life Example: People waiting in line ⍰.

Queue Operations (Array Implementation):

```
ENQUEUE (Queue, item):

1. If REAR = MAX-1 → Overflow
2. Else
    REAR ← REAR + 1
    Queue[REAR] ← item
```

DEQUEUE (Queue):

```
1. If FRONT > REAR → Underflow
2. Else
    item ← Queue[FRONT]
    FRONT ← FRONT + 1
    Return item
```

2.5 Circular Queue

A Circular Queue solves wasted space by wrapping REAR back to the start.

Example: CPU scheduling (Round Robin).

2.6 Deque (Double Ended Queue)

A Deque allows insertions and deletions at both ends.

Example: A train with doors at both ends.

2.7 Priority Queue

In a Priority Queue, elements are served based on priority, not arrival order.

Example: Hospital emergency room ▯.

Unit 3: Linked Lists

3.1 What is a Linked List?

A Linked List is a linear data structure where elements (nodes) are linked using pointers.

Each node has:

Data (information)

Next (address of next node)

Real-life Example: A train . Each compartment is a node, and the connector is the pointer.

3.2 Types of Linked Lists

Singly Linked List: Each node points to the next.

Doubly Linked List: Nodes point to both previous and next.

Circular Linked List: Last node points back to the first.

Header Linked List: Starts with a special header node.

3.3 Operations on Linked Lists

Traversing

```
TRAVERSE(head):
1. ptr ← head
2. While ptr ≠ NULL do
3.     Print ptr.data
4.     ptr ← ptr.next
```

Insertion at Beginning

```
INSERT_BEGIN(head, item):
1. newnode ← create(item)
2. newnode.next ← head
3. head ← newnode
```

Insertion at End

```
INSERT_END(head, item):
1. newnode ← create(item)
2. ptr ← head
3. While ptr.next ≠ NULL do
4.     ptr ← ptr.next
5. ptr.next ← newnode
```

Deletion

```
DELETE_NODE(head, key):
1. ptr ← head, prev ← NULL
2. While ptr ≠ NULL do
3.     if ptr.data = key then
4.         if prev = NULL → head ← ptr.next
5.         else prev.next ← ptr.next
6.     Move ptr forward
```

3.4 Special Types

Doubly Linked List: Traversal in both directions.

Circular Linked List: Continuous looping (e.g., round-robin CPU scheduling).

Unit 4: Searching and Sorting

4.1 Searching

Linear Search

```
LINEAR_SEARCH(A, n, key):
1. For i ← 0 to n-1 do
2.     If A[i] = key then return i
3. Return -1
```

Binary Search (on sorted list)

```
BINARY_SEARCH(A, low, high, key):
1. While low ≤ high do
2.     mid ← (low+high)/2
3.     If A[mid] = key → return mid
4.     Else if A[mid] > key → high ← mid-1
5.     Else low ← mid+1
6. Return -1
```

4.2 Sorting Techniques

Bubble Sort

```
BUBBLE_SORT(A, n):
```

```
1. For i ← 0 to n-1 do
2.    For j ← 0 to n-i-2 do
3.       If A[j] > A[j+1] → Swap
```

Selection Sort

```
SELECTION_SORT(A, n):
1. For i ← 0 to n-1 do
2.    min ← i
3.    For j ← i+1 to n-1 do
4.       If A[j] < A[min] → min ← j
5.    Swap A[i] & A[min]
```

Insertion Sort

```
INSERTION_SORT(A, n):
1. For i ← 1 to n-1 do
2.    key ← A[i], j ← i-1
3.    While j ≥ 0 and A[j] > key do
4.       A[j+1] ← A[j], j ← j-1
5.    A[j+1] ← key
```

Quick Sort

```
QUICK_SORT(A, low, high):
1. If low < high then
```

```
2.    p ← PARTITION(A, low, high)
3.    QUICK_SORT(A, low, p-1)
4.    QUICK_SORT(A, p+1, high)
```

Merge Sort

```
MERGE_SORT(A, low, high):
1. If low < high then
2.    mid ← (low+high)/2
3.    MERGE_SORT(A, low, mid)
4.    MERGE_SORT(A, mid+1, high)
5.    MERGE(A, low, mid, high)
```

Radix Sort

A non-comparative integer sorting algorithm. It processes numbers digit by digit and is stable (preserves order of equal elements).