Problem Solver (DI03032031) - Problem Solver

Antigravity AI

June 4, 2026

Contents

CHAPTER 1

Basic Concepts of Data Structures and OOP

Data structures and Object-Oriented Programming (OOP) form the foundation for designing efficient algorithms and modular software. This chapter focuses on the computational aspects of arrays, algorithmic complexity, and object memory calculation.

1.1 Array Address Calculation

An array is a collection of elements stored at contiguous memory locations. Calculating the exact memory address of an element is a fundamental computational task.

1.1.1 One-Dimensional (1D) Arrays

Methodology:

1D Array Address Calculation To find the memory address of an element in a 1D array, use the following formula:

1. Identify the base address (B) of the array (the address of the first element).
2. Identify the size (W) of each element in bytes.
3. Identify the lower bound (LB) of the array index (default is usually 0).
4. Let i be the index of the element whose address is to be found.
5. Apply the formula:

$$\text{Address}(A[i]) = B + W \times (i - LB)$$

Worked Example:

Calculate the address of a 1D array element Consider an array $A[-2 \dots 5]$ consisting of integer elements. The base address is 2000 and each integer requires 4 bytes of memory. Find the memory address of $A[3]$.
Step 1: Identify given parameters.

- Base address $B = 2000$
- Element size $W = 4$ bytes
- Lower bound $LB = -2$
- Target index $i = 3$

Step 2: Apply the 1D address formula.

$$\text{Address}(A[3]) = B + W \times (i - LB)$$

$$\text{Address}(A[3]) = 2000 + 4 \times (3 - (-2))$$

$$\text{Address}(A[3]) = 2000 + 4 \times (3 + 2)$$

$$\text{Address}(A[3]) = 2000 + 4 \times 5$$

$$\text{Address}(A[3]) = 2000 + 20 = 2020$$

The memory address of $A[3]$ is 2020.

1.1.2 Two-Dimensional (2D) Arrays

Methodology:

2D Array Address Calculation Row Major Order Row Major Order stores array elements row by row.

1. Identify Base Address (B), Element Size (W), Total Columns (N).
2. Identify the lower bound of rows (LR) and columns (LC).
3. Let the target element be $A[i][j]$.
4. Apply the formula:

$$\text{Address}(A[i][j]) = B + W \times [N \times (i - LR) + (j - LC)]$$

Worked Example:

Calculate address using Row Major Order An array $M[5][10]$ is stored in memory with each element requiring 2 bytes. The base address is 100. Assume the starting indices (lower bounds) are 0 for both rows and columns. Calculate the address of $M[3][7]$ using Row Major Order.

Step 1: Identify given parameters.

- $B = 100$
- $W = 2$ bytes
- N (Total columns) $= 10$
- $LR = 0$, $LC = 0$
- Target index $i = 3$, $j = 7$

Step 2: Apply the Row Major formula.

$$\text{Address}(M[3][7]) = B + W \times [N \times (i - LR) + (j - LC)]$$

$$\text{Address}(M[3][7]) = 100 + 2 \times [10 \times (3 - 0) + (7 - 0)]$$

$$\text{Address}(M[3][7]) = 100 + 2 \times [30 + 7]$$

$$\text{Address}(M[3][7]) = 100 + 2 \times 37$$

$$\text{Address}(M[3][7]) = 100 + 74 = 174$$

The memory address of $M[3][7]$ is 174.

Methodology:

2D Array Address Calculation Column Major Order Column Major Order stores array elements column by column.

1. Identify Base Address (B), Element Size (W), Total Rows (M).
2. Identify the lower bound of rows (LR) and columns (LC).

3. Let the target element be $A[i][j]$.
4. Apply the formula:

$$\text{Address}(A[i][j]) = B + W \times [(i - LR) + M \times (j - LC)]$$

Worked Example:

Calculate address using Column Major Order An array $X[4][5]$ has a base address of 1000 and element size of 4 bytes. Assuming indices start from 1 (i.e. $LR = 1$ and $LC = 1$), find the address of $X[2][4]$ using Column Major Order.

Step 1: Identify given parameters.

- $B = 1000$
- $W = 4$ bytes
- M (Total rows) $= 4$
- $LR = 1$, $LC = 1$
- Target index $i = 2$, $j = 4$

Step 2: Apply the Column Major formula.

$$\text{Address}(X[2][4]) = B + W \times [(i - LR) + M \times (j - LC)]$$

$$\text{Address}(X[2][4]) = 1000 + 4 \times [(2 - 1) + 4 \times (4 - 1)]$$

$$\text{Address}(X[2][4]) = 1000 + 4 \times [1 + 4 \times 3]$$

$$\text{Address}(X[2][4]) = 1000 + 4 \times [1 + 12]$$

$$\text{Address}(X[2][4]) = 1000 + 4 \times 13$$

$$\text{Address}(X[2][4]) = 1000 + 52 = 1052$$

The memory address of $X[2][4]$ is 1052.

1.2 Time Complexity Calculation

Time complexity determines how the execution time of an algorithm grows with the input size n. We typically use Big-O notation to describe the upper bound of an algorithm's running time.

Methodology:

Determining Big O of Iterative Algorithms Follow these steps to analyze the time complexity of loops:

1. Identify the basic operation (usually inside the innermost loop).
2. Determine how many times the basic operation executes relative to n.
3. For a simple loop from 1 to n, the iterations are proportional to n resulting in $O(n)$.
4. For nested loops iterate independently the total time is the product of loop sizes (e.g. $O(n) \times O(n) = O(n^2)$).
5. Drop constants and lower-order terms to formulate the final Big-O expression.

Worked Example:

Calculate time complexity of a single loop Determine the time complexity of the following pseudo-code:

```
sum = 0
for i = 1 to n:
    sum = sum + i
```

Step 1: Identify the basic operation. The operation `sum = sum + i` is the core execution block.
Step 2: Count iterations. The loop runs from $i = 1$ to n, incrementing by 1. Therefore, it executes exactly n times.
Step 3: Express in Big-O. The time complexity is directly proportional to n. Thus, $T(n) = O(n)$.

Worked Example:

Calculate time complexity of nested loops Determine the time complexity of the following pseudo-code:

```
count = 0
for i = 1 to n:
    for j = 1 to n:
        count = count + 1
```

Step 1: Analyze outer loop. The outer loop runs n times (for $i = 1 \dots n$).
Step 2: Analyze inner loop. For every iteration of the outer loop, the inner loop also runs n times (for $j = 1 \dots n$).
Step 3: Calculate total executions. Total operations = (Iterations of outer loop) $\times$ (Iterations of inner loop) $= n \times n = n^2$.
Step 4: Express in Big-O. The time complexity is bounded by n^2. Thus, $T(n) = O(n^2)$.

1.3 Object Memory Footprint in OOP

In Object-Oriented Programming, determining the memory required for an object involves summing up the memory required for all its member variables. Member functions generally do not occupy per-object memory (they are stored once for the class).

Methodology:

Calculating Object Size To compute the memory required by a single object of a class:

1. List all the data members (instance variables) defined in the class.
2. Determine the memory size of each primitive data type in the target system (e.g. integer = 4 bytes, character = 1 byte, float = 4 bytes).
3. Sum the sizes of all data members.
4. (Note: For simplicity in introductory calculations, memory alignment and padding are typically ignored unless specified.)

Worked Example:

Calculate memory size of an object Consider a class `Student` defined with the following data members on a 32-bit system:

- `int rollNumber` (4 bytes)
- `char grade` (1 byte)
- `float percentage` (4 bytes)

- An array `int marks[5]`

Find the total memory size of one object of the `Student` class.
Step 1: Calculate the size of simple data members.

- `rollNumber` = 4 bytes
- `grade` = 1 byte
- `percentage` = 4 bytes

Step 2: Calculate the size of the array data member. The `marks` array has 5 integers. Size = 5×4 bytes (size of int) = 20 bytes.
Step 3: Sum the sizes. Total Size = 4 (int) + 1 (char) + 4 (float) + 20 (array) = 29 bytes.
An object of the `Student` class requires 29 bytes of memory.

CHAPTER 2

Stacks and Queues

2.1 Introduction to Stacks

A stack is a Last-In-First-Out (LIFO) data structure. Elements are inserted and removed from only one end called the **Top**. This chapter focuses on the computational aspects and implementations of stack and queue structures.

Methodology:

Stack Operations Push and Pop Let S be a stack of maximum size N. Let TOP be the index of the top element (initially initialized to -1).
Push (Insert an element X):

1. Check for overflow condition: If $TOP = N - 1$, then print "Stack Overflow" and exit.
2. Increment Top pointer: $TOP = TOP + 1$.
3. Insert the new element: $S[TOP] = X$.

Pop (Remove an element):

1. Check for underflow condition: If $TOP = -1$, then print "Stack Underflow" and exit.
2. Read the top element: $X = S[TOP]$.
3. Decrement Top pointer: $TOP = TOP - 1$.
4. Return X.

Worked Example:

Trace Stack Operations Consider a stack of size $N = 4$. Trace the following sequence of operations and show the stack state and TOP pointer at each step: 1. Push 10 2. Push 20 3. Pop 4. Push 30 5. Push 40 6. Push 50
Solution:

Step	Operation	Stack Content (Bottom to Top)	TOP
0	Initial State	Empty	-1
1	Push 10	10	0
2	Push 20	10 20	1
3	Pop	10	0
4	Push 30	10 30	1
5	Push 40	10 30 40	2
6	Push 50	10 30 40 50	3

Note: A subsequent operation like "Push 60" would cause a Stack Overflow because the stack is full at $TOP = 3$ $(N - 1)$.

2.2 Applications of Stacks

Methodology:

Infix to Postfix Conversion To convert a mathematical infix expression to a postfix expression using a stack:

1. Add a right parenthesis ‘)’ to the end of the given infix expression and push a left parenthesis ‘(’ onto the stack.
2. Read the expression sequentially from left to right.
3. If the scanned symbol is an operand (letter or digit), append it to the postfix output.
4. If the scanned symbol is a left parenthesis ‘(’, push it onto the stack.
5. If the scanned symbol is an operator (e.g. +, -, *, /):
   - Repeatedly pop operators from the stack and append them to the output until an operator with strictly lower precedence or a ‘(’ is at the top of the stack.
   - Push the current operator onto the stack.
6. If the scanned symbol is a right parenthesis ‘)’, repeatedly pop operators from the stack and append to the output until a left parenthesis ‘(’ is popped. Do not add ‘(’ or ‘)’ to the postfix output.

Operator Precedence Levels: $\hat{}$ (highest), *, / (medium), +, - (lowest).

Worked Example:

Convert Infix to Postfix Convert the infix expression `A + B * C - D` to postfix.
Solution: First, append ‘)’ to the expression and push ‘(’ to the stack. Expression to process: `A + B * C - D )`

Symbol Scanned	Stack Status (Bottom to Top)	Postfix Expression
-	(	Empty
A	(	A
+	(+	A
B	(+	A B
*	(+ *	A B
C	(+ *	A B C
-	(-	A B C * +
D	(-	A B C * + D
)	Empty	A B C * + D -

The final resulting postfix expression is `A B C * + D -`. Notice that when ‘-‘ is scanned, the stack has ‘*‘ and ‘+‘ which both have higher or equal precedence, so they are popped before ‘-‘ is pushed.

Methodology:

Evaluation of Postfix Expression To evaluate a postfix expression numerically:

1. Add a special delimiter (e.g. ‘#’) to the end of the postfix expression.
2. Scan the expression from left to right.
3. If an operand is encountered, push it onto the stack.
4. If an operator is encountered, pop the top two elements from the stack (let them be A and B, where A is the top element and B is the second top).
5. Evaluate the expression B [operator] A and push the numerical result back onto the stack.

6. When the delimiter is reached, the final result is the single value remaining on the stack.

Worked Example:

Evaluate Postfix Expression Evaluate the postfix expression `5 3 + 8 2 / *`
Solution:

Symbol Scanned	Stack (Bottom to Top)	Operation Performed
5	5	Push 5
3	5 3	Push 3
+	8	Pop 3, Pop 5, Compute $5+3=8$, Push 8
8	8 8	Push 8
2	8 8 2	Push 2
/	8 4	Pop 2, Pop 8, Compute $8/2=4$, Push 4
*	32	Pop 4, Pop 8, Compute $8*4=32$, Push 32

The final evaluated result is 32.

2.3 Queues

A Queue is a First-In-First-Out (FIFO) data structure. Elements are inserted at one end called the **Rear** and removed from the other end called the **Front**.

Methodology:

Linear Queue Operations Enqueue and Dequeue Let Q be a linear queue of maximum size N. Two pointers are used, $FRONT$ and $REAR$, both initialized to -1.
Enqueue (Insert an element X):

1. Check for overflow condition: If $REAR = N-1$, then print "Queue Overflow" and exit.
2. Check if empty: If $FRONT = -1$, set $FRONT = 0$ (since this is the first element being inserted).
3. Increment Rear pointer: $REAR = REAR + 1$.
4. Insert the element: $Q[REAR] = X$.

Dequeue (Remove an element):

1. Check for underflow condition: If $FRONT = -1$, then print "Queue Underflow" and exit.
2. Read the element to be deleted: $X = Q[FRONT]$.
3. Check if only one element exists: If $FRONT = REAR$, the queue is now empty. Set $FRONT = -1$ and $REAR = -1$.
4. Else, increment Front pointer: $FRONT = FRONT + 1$.
5. Return X.

Worked Example:

Trace Linear Queue Operations Consider a linear queue of size $N = 3$. Trace the following operations sequentially: 1. Enqueue 10 2. Enqueue 20 3. Dequeue 4. Enqueue 30 5. Enqueue 40
Solution:

Step	Operation	Queue Elements	FRONT	REAR
0	Initial State	Empty	-1	-1
1	Enqueue 10	10	0	0
2	Enqueue 20	10 20	0	1
3	Dequeue	20 (10 removed)	1	1
4	Enqueue 30	20 30	1	2
5	Enqueue 40	Overflow (REAR=2, N=3)	1	2

Notice in step 5, even though there is an empty space at index 0 (since element 10 was deleted), the linear queue suffers an overflow condition because $REAR = N - 1$.

2.4 Circular Queues

Circular queues overcome the memory wastage limitation of linear queues by wrapping the REAR pointer around to the beginning of the array if there is empty space available at the front.

Methodology:

Circular Queue Operations Enqueue and Dequeue Let CQ be a circular queue of size N. Initialize pointers $FRONT = -1$ and $REAR = -1$.

Circular Enqueue (Insert element X):

1. Check for overflow condition: If $(REAR + 1) \bmod N = FRONT$, print "Queue Overflow" and exit.
2. If queue is empty ($FRONT = -1$), set $FRONT = 0$ and $REAR = 0$.
3. Else, update rear pointer using modulo arithmetic: $REAR = (REAR + 1) \bmod N$.
4. Insert element: $CQ[REAR] = X$.

Circular Dequeue (Remove element):

1. Check for underflow condition: If $FRONT = -1$, print "Queue Underflow" and exit.
2. Read the element: $X = CQ[FRONT]$.
3. Check if only one element remains: If $FRONT = REAR$, the queue is now empty. Set $FRONT = -1$ and $REAR = -1$.
4. Else, update front pointer using modulo arithmetic: $FRONT = (FRONT + 1) \bmod N$.
5. Return X.

Worked Example:

Trace Circular Queue Operations Consider a circular queue of size $N = 3$. Trace the following operations: 1. Enqueue A, Enqueue B, Enqueue C 2. Dequeue 3. Enqueue D

Solution:

Step	Operation	Queue Elements	FRONT	REAR
0	Initial State	Empty	-1	-1
1	Enqueue A	A at index 0	0	0
2	Enqueue B	A, B at indices 0, 1	0	1
3	Enqueue C	A, B, C at indices 0, 1, 2	0	2
4	Dequeue (removes A)	B, C at indices 1, 2	1	2
5	Enqueue D	D at index 0; B, C at 1, 2	1	0

Because it is a circular queue, element D successfully occupies the freed space at index 0. The rear pointer successfully wrapped around via $(2 + 1) \bmod 3 = 0$.

CHAPTER 3
Linked List

3.1 Introduction

A Linked List is a dynamic data structure consisting of nodes. Each node contains a data element and a reference (pointer) to the next node. Unlike arrays, linked lists do not store elements in contiguous memory.

3.2 Singly Linked List

A Singly Linked List (SLL) allows traversal in only one forward direction.

3.2.1 Node Creation

Methodology:

Creating a Node for SLL

1. Define a node structure containing a data part and a pointer to the same structure type.
2. Allocate memory for the new node dynamically using `malloc()` in C.
3. Assign the data value to the node's data part.
4. Assign `NULL` to the node's next pointer.

Worked Example:

C Code for SLL Node Write the C structure and memory allocation step for creating a node with data 10.
Solution:

```
struct Node {
    int data;
    struct Node* next;
};
// Memory allocation
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = 10;
newNode->next = NULL;
```

3.2.2 Insertion Operations

Methodology:

Inserting at the Beginning of SLL

1. Create a new node `N` with the required data.

2. Set `N.next` to the current `head` pointer.
3. Update the `head` pointer to point to the new node `N`.

Worked Example:

Insert 15 at Head Given a singly linked list: head → 25 → 35 → NULL. Insert 15 at the beginning.
Solution:

1. Create node `N` with data 15 and next as `NULL`.
2. `N.next = head` (so `N.next` points to 25).
3. `head = N`.
4. Final List: head → 15 → 25 → 35 → NULL.

Methodology:

Inserting at the End of SLL

1. Create a new node `N` with the required data and `N.next = NULL`.
2. If the `head` is `NULL` (empty list) set `head = N` and return.
3. Otherwise initialize `temp = head`.
4. Traverse the list: `while (temp.next != NULL)` do `temp = temp.next`.
5. Set `temp.next = N`.

Worked Example:

Insert 45 at End Given a singly linked list: head → 15 → 25 → NULL. Insert 45 at the end.
Solution:

1. Create node `N` with data 45 and next as `NULL`.
2. `temp = head` (15).
3. Traverse: `temp` moves to 25. The condition `temp.next != NULL` becomes false.
4. `temp.next = N` (node 25 points to node 45).
5. Final List: head → 15 → 25 → 45 → NULL.

3.2.3 Deletion Operations

Methodology:

Deleting from the Beginning of SLL

1. Check for underflow: If `head == NULL` deletion is not possible.
2. Store `head` in a temporary pointer `temp`.
3. Update `head = head.next`.
4. Deallocate memory of the node pointed by `temp`.

Worked Example:

Delete First Node Given list: head → 10 → 20 → 30 → NULL. Delete from beginning.
Solution:

1. `temp = head` (node with data 10).
2. `head = head.next` (`head` becomes node with data 20).
3. Free `temp`.
4. Final List: head → 20 → 30 → NULL.

3.3 Doubly Linked List

A Doubly Linked List (DLL) has nodes with three components: data, a `prev` pointer to the previous node, and a `next` pointer to the next node. This allows traversal in both directions.

Methodology:

Inserting at the End of DLL

1. Create a new node `N` with the required data, `N.prev = NULL`, and `N.next = NULL`.
2. If `head == NULL` set `head = N` and return.
3. Traverse to the last node using a pointer `temp`.
4. Set `temp.next = N`.
5. Set `N.prev = temp`.

Worked Example:

Insert 100 at End of DLL Given DLL: head → 50 ↔ 75 → NULL. Insert 100 at the end.
Solution:

1. Create node `N` with data 100.
2. Traverse to node 75 using `temp`.
3. Set `temp.next = N` (node 75 points to node 100).
4. Set `N.prev = temp` (node 100 points back to node 75).
5. Final List: head → 50 ↔ 75 ↔ 100 → NULL.

Methodology:

Deleting from the End of DLL

1. If `head == NULL` return (underflow).
2. If `head.next == NULL` (only one node) free `head` and set `head = NULL`.
3. Otherwise traverse to the last node using `temp`.
4. Update the previous node's next pointer: `temp.prev.next = NULL`.
5. Deallocate memory for `temp`.

Worked Example:

Delete Last Node of DLL Given DLL: head → 10 ↔ 20 ↔ 30 → NULL. Delete from the end.
Solution:

1. Traverse to the last node (30) with `temp`.
2. Node 30's `prev` is node 20.
3. Set node 20's `next = NULL`.
4. Free `temp` (node 30).
5. Final List: head → 10 ↔ 20 → NULL.

3.4 Circular Linked List

In a Circular Linked List (CLL), the last node's `next` pointer stores the address of the first node forming a circle.

Methodology:

Inserting at the Beginning of CLL

1. Create a new node `N` with required data.
2. If `head == NULL` set `head = N` and `N.next = head`.
3. Otherwise traverse to the last node (say `temp`) where `temp.next == head`.
4. Set `N.next = head`.
5. Set `temp.next = N`.
6. Update `head = N`.

Worked Example:

Insert 5 at Beginning of CLL Given CLL: head → 15 → 25 → (back to 15). Insert 5 at head.
Solution:

1. Create node `N` with data 5.
2. Traverse to last node (25) using `temp`.
3. `N.next = head` (node 5 points to node 15).
4. `temp.next = N` (node 25 points to node 5).
5. `head = N` (`head` now points to 5).
6. Final List: head → 5 → 15 → 25 → (back to 5).

CHAPTER 4
Searching and Sorting

Searching and sorting are fundamental operations in data structures. Searching involves finding the location of a specific element in a collection, while sorting is the process of arranging elements in a particular order (ascending or descending).

4.1 Linear Search

Linear search is the simplest searching algorithm. It sequentially checks each element of the list until a match is found or the whole list has been searched.

Methodology:

Linear Search Algorithm **Input:** An array A of n elements and a target value x. **Output:** The index of x if found, otherwise -1.

1. Start from the first element of the array ($i = 0$).
2. Compare the target element x with the current element $A[i]$.
3. If $A[i] == x$, return the index i.
4. If $A[i] \neq x$, move to the next element ($i = i + 1$).
5. Repeat steps 2-4 until the end of the array is reached.
6. If the end of the array is reached and x is not found, return -1.

Worked Example:

Perform Linear Search for 34 Given the array $A = [12, 45, 23, 56, 34, 89, 10]$ and a search target $x = 34$. Trace the steps to find the target.
Solution:

1. **Step 1** ($i = 0$)**:** $A[0] = 12$. Compare 12 with 34. $12 \neq 34$.
2. **Step 2** ($i = 1$)**:** $A[1] = 45$. Compare 45 with 34. $45 \neq 34$.
3. **Step 3** ($i = 2$)**:** $A[2] = 23$. Compare 23 with 34. $23 \neq 34$.
4. **Step 4** ($i = 3$)**:** $A[3] = 56$. Compare 56 with 34. $56 \neq 34$.
5. **Step 5** ($i = 4$)**:** $A[4] = 34$. Compare 34 with 34. $34 == 34$. Match found!

Result: The element 34 is found at index 4.

4.2 Binary Search

Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item until you have narrowed down the possible locations to just one.

Methodology:

Binary Search Algorithm **Input:** A **sorted** array A of n elements and a target value x. **Output:** The index of x if found, otherwise -1.

1. Set two pointers: $low = 0$ and $high = n - 1$.
2. While $low \leq high$:
   (a) Calculate the middle index: $mid = \lfloor \frac{low+high}{2} \rfloor$.
   (b) If $A[mid] == x$, return mid.
   (c) If $A[mid] < x$, the target is in the right half. Update $low = mid + 1$.
   (d) If $A[mid] > x$, the target is in the left half. Update $high = mid - 1$.
3. If the loop terminates without finding the target, return -1.

Worked Example:

Perform Binary Search for 56 Given the sorted array $A = [10, 12, 23, 34, 45, 56, 89]$ and a search target $x = 56$. Trace the binary search algorithm.
Solution: Initial state: $low = 0$, $high = 6$. Target $x = 56$.

- **Iteration 1:**
  - $mid = \lfloor (0 + 6)/2 \rfloor = 3$.
  - $A[3] = 34$.
  - Compare 34 with 56. Since $34 < 56$, update $low = mid + 1 = 3 + 1 = 4$.
- **Iteration 2:**
  - Current bounds: $low = 4$, $high = 6$.
  - $mid = \lfloor (4 + 6)/2 \rfloor = 5$.
  - $A[5] = 56$.
  - Compare 56 with 56. Since $56 == 56$, the target is found.

Result: The element 56 is found at index 5.

4.3 Bubble Sort

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

Methodology:

Bubble Sort Algorithm **Input:** An array A of n elements. **Output:** The array A sorted in ascending order.

1. Loop i from 0 to $n - 1$ (representing the pass number).
2. Inside the loop, loop j from 0 to $n - i - 2$.

3. Compare adjacent elements $A[j]$ and $A[j+1]$.
4. If $A[j] > A[j+1]$, swap them.
5. After each full inner loop, the largest unsorted element "bubbles up" to its correct position at the end of the array.
6. Repeat until no swaps are needed (or $n-1$ passes are completed).

Worked Example:

Sort Array using Bubble Sort Sort the array $A = [5, 1, 4, 2, 8]$ using Bubble Sort.
Solution:
Pass 1 ($i = 0$)**:**

- Compare 5 and 1: $5 > 1$, swap $\rightarrow [1, 5, 4, 2, 8]$
- Compare 5 and 4: $5 > 4$, swap $\rightarrow [1, 4, 5, 2, 8]$
- Compare 5 and 2: $5 > 2$, swap $\rightarrow [1, 4, 2, 5, 8]$
- Compare 5 and 8: $5 < 8$, no swap $\rightarrow [1, 4, 2, 5, 8]$

After Pass 1, the largest element (8) is in its correct position.
Pass 2 ($i = 1$)**:**

- Compare 1 and 4: $1 < 4$, no swap $\rightarrow [1, 4, 2, 5, 8]$
- Compare 4 and 2: $4 > 2$, swap $\rightarrow [1, 2, 4, 5, 8]$
- Compare 4 and 5: $4 < 5$, no swap $\rightarrow [1, 2, 4, 5, 8]$

After Pass 2, the next largest element (5) is in its correct position.
Pass 3 ($i = 2$)**:**

- Compare 1 and 2: $1 < 2$, no swap $\rightarrow [1, 2, 4, 5, 8]$
- Compare 2 and 4: $2 < 4$, no swap $\rightarrow [1, 2, 4, 5, 8]$

The array is now sorted. A smart implementation would stop here since no swaps were made in Pass 3.
Result: The sorted array is $[1, 2, 4, 5, 8]$.

4.4 Selection Sort

Selection sort divides the input list into two parts: a sorted sublist of items which is built up from left to right, and a sublist of the remaining unsorted items. It proceeds by finding the smallest element in the unsorted sublist and exchanging (swapping) it with the leftmost unsorted element.

Methodology:

Selection Sort Algorithm **Input:** An array A of n elements. **Output:** The array A sorted in ascending order.

1. Loop i from 0 to $n-2$.
2. Assume the element at index i is the minimum: $min\_idx = i$.
3. Loop j from $i+1$ to $n-1$ to find the actual minimum element in the unsorted part.
4. If $A[j] < A[min\_idx]$, update $min\_idx = j$.

5. After the inner loop, if $min\_idx \neq i$, swap $A[i]$ with $A[min\_idx]$.
6. The sorted boundary expands by one element to the right.

Worked Example:

Sort Array using Selection Sort Sort the array $A = [64, 25, 12, 22, 11]$ using Selection Sort.
Solution:
Step 1 ($i = 0$): Unsorted part is $[64, 25, 12, 22, 11]$.

- Find minimum: The minimum element is 11 at index 4.
- Swap 64 (at index 0) with 11 (at index 4).
- Array becomes: $[11, 25, 12, 22, 64]$.

Step 2 ($i = 1$): Unsorted part is $[25, 12, 22, 64]$.

- Find minimum: The minimum element is 12 at index 2.
- Swap 25 (at index 1) with 12 (at index 2).
- Array becomes: $[11, 12, 25, 22, 64]$.

Step 3 ($i = 2$): Unsorted part is $[25, 22, 64]$.

- Find minimum: The minimum element is 22 at index 3.
- Swap 25 (at index 2) with 22 (at index 3).
- Array becomes: $[11, 12, 22, 25, 64]$.

Step 4 ($i = 3$): Unsorted part is $[25, 64]$.

- Find minimum: The minimum element is 25 at index 3.
- Since 25 is already at the first position of the unsorted part, no swap is needed.
- Array remains: $[11, 12, 22, 25, 64]$.

Result: The sorted array is $[11, 12, 22, 25, 64]$.

4.5 Insertion Sort

Insertion sort builds the final sorted array one item at a time. It iterates through the input elements and grows a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there.

Methodology:

Insertion Sort Algorithm **Input:** An array A of n elements. **Output:** The array A sorted in ascending order.

1. Loop i from 1 to $n - 1$.
2. Set $key = A[i]$ (the element to be inserted into the sorted portion).
3. Set $j = i - 1$.
4. While $j \geq 0$ and $A[j] > key$:

(a) Shift $A[j]$ one position to the right: $A[j+1] = A[j]$.

(b) Decrement j by 1.

5. Insert the key at its correct position: $A[j+1] = key$.

Worked Example:

Sort Array using Insertion Sort Sort the array $A = [12, 11, 13, 5, 6]$ using Insertion Sort.
Solution: Start with the first element (12) considered as a sorted sublist.
Pass 1 ($i = 1$): $key = 11$. Compare with sorted portion $[12]$.

- Since $12 > 11$, shift 12 to the right.
- Insert 11 at index 0.
- Array becomes: $[11, 12, 13, 5, 6]$.

Pass 2 ($i = 2$): $key = 13$. Compare with sorted portion $[11, 12]$.

- Since $12 \leq 13$, no shifting is needed.
- 13 remains at its current position.
- Array remains: $[11, 12, 13, 5, 6]$.

Pass 3 ($i = 3$): $key = 5$. Compare with sorted portion $[11, 12, 13]$.

- $13 > 5$: shift 13 right. Array is now $[11, 12, \_, 13, 6]$.
- $12 > 5$: shift 12 right. Array is now $[11, \_, 12, 13, 6]$.
- $11 > 5$: shift 11 right. Array is now $[\_, 11, 12, 13, 6]$.
- Insert 5 at index 0.
- Array becomes: $[5, 11, 12, 13, 6]$.

Pass 4 ($i = 4$): $key = 6$. Compare with sorted portion $[5, 11, 12, 13]$.

- $13 > 6$: shift 13 right.
- $12 > 6$: shift 12 right.
- $11 > 6$: shift 11 right.
- $5 \leq 6$: stop shifting.
- Insert 6 at index 1.
- Array becomes: $[5, 6, 11, 12, 13]$.

Result: The sorted array is $[5, 6, 11, 12, 13]$.

4.6 Merge Sort

Merge sort is a divide-and-conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

Methodology:

Merge Sort Algorithm **Input:** An array A with starting index low and ending index $high$. **Output:** The sorted sub-array.

Algorithm MergeSort(A, low, high):

1. If $low < high$:
   (a) Find the middle point: $mid = \lfloor (low + high)/2 \rfloor$.
   (b) Recursively call MergeSort for the first half: $MergeSort(A, low, mid)$.
   (c) Recursively call MergeSort for the second half: $MergeSort(A, mid + 1, high)$.
   (d) Merge the two sorted halves: $Merge(A, low, mid, high)$.

Merge Procedure(A, low, mid, high):

1. Create temporary arrays for the left half (L) and right half (R).
2. Compare elements from L and R one by one.
3. Copy the smaller element back into the original array A.
4. After one of the halves is exhausted, copy any remaining elements from the other half into A.

Worked Example:

Trace Merge Sort Show the splitting and merging steps of Merge Sort on the array $[38, 27, 43, 3, 9, 82, 10]$.

Solution:

1. Splitting Phase:

- $[38, 27, 43, 3, 9, 82, 10]$ is split into $[38, 27, 43, 3]$ and $[9, 82, 10]$.
- $[38, 27, 43, 3]$ is split into $[38, 27]$ and $[43, 3]$.
- $[38, 27]$ is split into $[38]$ and $[27]$.
- $[43, 3]$ is split into $[43]$ and $[3]$.
- $[9, 82, 10]$ is split into $[9, 82]$ and $[10]$.
- $[9, 82]$ is split into $[9]$ and $[82]$.

2. Merging Phase:

- Merge $[38]$ and $[27] \rightarrow [27, 38]$.
- Merge $[43]$ and $[3] \rightarrow [3, 43]$.
- Merge $[27, 38]$ and $[3, 43] \rightarrow [3, 27, 38, 43]$.
- Merge $[9]$ and $[82] \rightarrow [9, 82]$.
- Merge $[9, 82]$ and $[10] \rightarrow [9, 10, 82]$.
- Finally, merge $[3, 27, 38, 43]$ and $[9, 10, 82]$.
  - Compare 3 and 9 $\rightarrow [3]$
  - Compare 27 and 9 $\rightarrow [3, 9]$
  - Compare 27 and 10 $\rightarrow [3, 9, 10]$
  - Compare 27 and 82 $\rightarrow [3, 9, 10, 27]$
  - Compare 38 and 82 $\rightarrow [3, 9, 10, 27, 38]$

- Compare 43 and 82 $\rightarrow [3, 9, 10, 27, 38, 43]$
- Copy remaining 82 $\rightarrow [3, 9, 10, 27, 38, 43, 82]$

Result: The sorted array is $[3, 9, 10, 27, 38, 43, 82]$.

4.7 Quick Sort

Quick sort is another divide-and-conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways. Here, we choose the last element as the pivot.

Methodology:

Quick Sort Algorithm **Input:** An array A with starting index low and ending index $high$. **Output:** The sorted sub-array.
Algorithm QuickSort(A, low, high):

1. If $low < high$:
   (a) Find the partition index $pi = Partition(A, low, high)$.
   (b) Recursively call QuickSort before partition: $QuickSort(A, low, pi - 1)$.
   (c) Recursively call QuickSort after partition: $QuickSort(A, pi + 1, high)$.

Partition Procedure(A, low, high):

1. Choose the last element as pivot: $pivot = A[high]$.
2. Set index of smaller element: $i = low - 1$.
3. Loop j from low to $high - 1$:
   (a) If $A[j] < pivot$, increment i and swap $A[i]$ with $A[j]$.
4. Swap $A[i + 1]$ with $A[high]$ (the pivot).
5. Return the partition index $i + 1$.

Worked Example:

Trace Quick Sort Partition Step Given an array $A = [10, 80, 30, 90, 40, 50, 70]$. Trace the first partition step using the last element as the pivot.
Solution: $low = 0$, $high = 6$, $pivot = A[6] = 70$. Initialize $i = low - 1 = -1$.

- **Loop $j = 0$:** $A[0] = 10$. Since $10 < 70$, i becomes 0. Swap $A[0]$ and $A[0]$ (no change). Array: $[10, 80, 30, 90, 40, 50, 70]$.
- **Loop $j = 1$:** $A[1] = 80$. $80 > 70$. Do nothing. i remains 0.
- **Loop $j = 2$:** $A[2] = 30$. Since $30 < 70$, i becomes 1. Swap $A[1]$ and $A[2]$. Array: $[10, 30, 80, 90, 40, 50, 70]$.
- **Loop $j = 3$:** $A[3] = 90$. $90 > 70$. Do nothing. i remains 1.
- **Loop $j = 4$:** $A[4] = 40$. Since $40 < 70$, i becomes 2. Swap $A[2]$ and $A[4]$. Array: $[10, 30, 40, 90, 80, 50, 70]$.
- **Loop $j = 5$:** $A[5] = 50$. Since $50 < 70$, i becomes 3. Swap $A[3]$ and $A[5]$. Array: $[10, 30, 40, 50, 80, 90, 70]$.

End of loop. Now place the pivot in its correct position: Swap $A[i+1]$ (which is $A[4] = 80$) with $A[high]$ (which is $A[6] = 70$). Final Array after first partition: $[10, 30, 40, 50, 70, 90, 80]$.
Result: The pivot 70 is now at its correct position (index 4). All elements to its left are smaller, and all elements to its right are larger. The function returns index 4.

CHAPTER 5

Trees

5.1 Binary Tree Traversals

Tree traversal refers to the process of visiting each node in a tree data structure exactly once. For binary trees, there are three primary depth-first traversal methods.

Methodology:

Preorder Traversal Algorithm In a preorder traversal, the nodes are visited in the following order:

1. Visit the root node.
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

Methodology:

Inorder Traversal Algorithm In an inorder traversal, the nodes are visited in the following order:

1. Traverse the left subtree in inorder.
2. Visit the root node.
3. Traverse the right subtree in inorder.

Methodology:

Postorder Traversal Algorithm In a postorder traversal, the nodes are visited in the following order:

1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Visit the root node.

Worked Example:

Finding Traversals of a Binary Tree Find the Preorder, Inorder and Postorder traversals for a binary tree where A is the root. A's left child is B and right child is C. B's left child is D and right child is E. C's left child is F and right child is G.

Solution:

1. **Tree Structure:**

/ \/ \
D E F G

2. **Preorder Traversal (Root → Left → Right):**
   - Start at root A.
   - Left subtree of A: B, D, E
   - Right subtree of A: C, F, G
   - Result: A, B, D, E, C, F, G
3. **Inorder Traversal (Left → Root → Right):**
   - Left subtree of A: D, B, E
   - Root: A
   - Right subtree of A: F, C, G
   - Result: D, B, E, A, F, C, G
4. **Postorder Traversal (Left → Right → Root):**
   - Left subtree of A: D, E, B
   - Right subtree of A: F, G, C
   - Root: A
   - Result: D, E, B, F, G, C, A

5.2 Constructing a Binary Tree from Traversals

A binary tree can be uniquely constructed if the inorder traversal is given along with either the preorder or postorder traversal.

Methodology:

Constructing Tree from Inorder and Preorder

1. Read the elements from the preorder traversal one by one from left to right. The first element is the root of the tree.
2. Find this root element in the inorder traversal.
3. The elements to the left of the root in the inorder traversal form the left subtree, and the elements to the right form the right subtree.
4. Repeat this process recursively for the left and right subtrees using the subsequent elements in the preorder sequence.

Worked Example:

Construction using Inorder and Preorder Construct a binary tree given the following traversals:

- Inorder: D, B, E, A, F, C
- Preorder: A, B, D, E, C, F

Solution:

1. **Step 1: Identify Root**
   The first element in Preorder is 'A'. Thus, 'A' is the root.

2. **Step 2: Partition Inorder Sequence**
   Find 'A' in Inorder: (D, B, E) A (F, C).
   Left subtree contains: D, B, E.
   Right subtree contains: F, C.

3. **Step 3: Construct Left Subtree (D B E)**
   The next elements in Preorder are B, D, E.
   The first among them in Preorder is 'B', so 'B' is the root of the left subtree.
   Find 'B' in Inorder of left subtree: (D) B (E).
   Left child of B is D, Right child of B is E.

4. **Step 4: Construct Right Subtree (F C)**
   The remaining elements in Preorder are C, F.
   The first among them in Preorder is 'C', so 'C' is the root of the right subtree.
   Find 'C' in Inorder of right subtree: (F) C.
   Left child of C is F, Right child of C is empty.

5. **Final Tree:**

Methodology:

Constructing Tree from Inorder and Postorder

1. Read the elements from the postorder traversal from right to left. The last element is the root of the tree.
2. Find this root element in the inorder traversal.
3. The elements to the left of the root in the inorder traversal form the left subtree, and the elements to the right form the right subtree.
4. Repeat this process recursively, noting that in reverse postorder, the right subtree elements appear before the left subtree elements.

Worked Example:

Construction using Inorder and Postorder Construct a binary tree given the following traversals:

- Inorder: 4, 8, 2, 5, 1, 6, 3, 7
- Postorder: 8, 4, 5, 2, 6, 7, 3, 1

Solution:

1. **Step 1: Identify Root**
   The last element in Postorder is 1. Root is 1.

2. **Step 2: Partition Inorder Sequence**
   Find 1 in Inorder: (4, 8, 2, 5) 1 (6, 3, 7).
   Left subtree: 4, 8, 2, 5.
   Right subtree: 6, 3, 7.

3. **Step 3: Construct Right Subtree (6 3 7)**
   Look at Postorder before 1: 3 is the last element among (6, 7, 3). Root is 3.

Find 3 in Inorder: (6) 3 (7).
Left child of 3 is 6, Right child is 7.

4. **Step 4: Construct Left Subtree (4 8 2 5)**
   Look at Postorder elements for left subtree: (8, 4, 5, 2). Last element is 2. Root is 2.
   Find 2 in Inorder: (4, 8) 2 (5).
   Right child of 2 is 5. Left subtree of 2 is (4, 8).
   For (4, 8), Postorder is (8, 4). Root is 4.
   Find 4 in Inorder: 4 (8).
   Right child of 4 is 8, Left child is empty.

5. **Final Tree:**

5.3 Binary Search Trees

A Binary Search Tree (BST) is a binary tree where for each node, all elements in its left subtree are strictly less than the node's value, and all elements in its right subtree are strictly greater than the node's value.

Methodology:

BST Insertion Algorithm To insert a new value X into a BST:

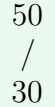
1. If the tree is empty, create a new node with value X as the root.
2. If X is less than the current node's value, move to the left child.
3. If X is greater than the current node's value, move to the right child.
4. Repeat steps 2 and 3 until an empty position is found, and insert the new node there.

Worked Example:

Constructing a BST from a Sequence Construct a Binary Search Tree by inserting the following sequence of numbers: 50, 30, 20, 40, 70, 60, 80.
Solution:

1. **Insert 50:** Tree is empty, so 50 becomes the root.

2. **Insert 30:** $30 < 50$, insert as left child of 50.



3. **Insert 20:** $20 < 50$ (go left), $20 < 30$, insert as left child of 30.

/
30
/
20

4. **Insert 40:** $40 < 50$ (go left), $40 > 30$, insert as right child of 30.

50
/
30
/ \
20 40

5. **Insert 70:** $70 > 50$, insert as right child of 50.

50
/ \
30 70
/ \
20 40

6. **Insert 60:** $60 > 50$ (go right), $60 < 70$, insert as left child of 70.
7. **Insert 80:** $80 > 50$ (go right), $80 > 70$, insert as right child of 70.
8. **Final BST:**

50
/ \
30 70
/ \/ \
20 40 60 80

Methodology:

BST Deletion Algorithm To delete a node N from a BST, three cases arise:

1. **Case 1:** N **is a leaf node.** Simply remove N from the tree.
2. **Case 2:** N **has exactly one child.** Remove N and replace it with its child.
3. **Case 3:** N **has two children.** Find the inorder successor of N (the smallest value in the right subtree) or the inorder predecessor (the largest value in the left subtree). Replace N's value with that of the successor/predecessor, and then recursively delete the successor/predecessor node.

Worked Example:

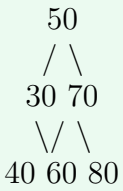
Deleting Nodes from a BST Given the BST constructed in the previous example, show the tree after performing the following operations sequentially: 1. Delete 20 2. Delete 30 3. Delete 50
Solution:

1. **Initial Tree:**

50
/ \
30 70
/ \/ \

20 40 60 80

2. **Delete 20 (Leaf Node):**
   Node 20 has no children. We simply remove it.



3. **Delete 30 (Node with one child):**
   Node 30 has only one child (40). We replace 30 with 40.

4. **Delete 50 (Node with two children):**
   Node 50 has two children. We find its inorder successor (smallest value in the right subtree), which is 60.
   We replace 50's value with 60, and then delete the original node 60 (which is a leaf).

Appendix: Key Formulae

This appendix provides a quick reference to the key formulae and mathematical relationships discussed in this book.

Unit 1: Array Address Calculations

1-Dimensional Arrays

The memory address of an element at index i in a 1D array A is calculated as:

$$\text{Address}(A[i]) = B + W \times (i - LB)$$

where:

- B is the base address of the array.
- W is the word size (size of each element in bytes).
- LB is the lower bound of the array index.

2-Dimensional Arrays

The memory address of an element at row i and column j in a 2D array depends on how the array is stored in memory. Let M be the total number of rows and N be the total number of columns, with LR as the lower bound for rows and LC as the lower bound for columns.

Row-Major Order:

$$\text{Address}(A[i][j]) = B + W \times [N \times (i - LR) + (j - LC)]$$

Column-Major Order:

$$\text{Address}(A[i][j]) = B + W \times [(i - LR) + M \times (j - LC)]$$

Unit 2: Queues and Circular Queues

Linear Queue Operations

When inserting an element X into a linear queue Q:

$$REAR = REAR + 1$$
$$Q[REAR] = X$$

When deleting an element X from a linear queue Q:

$$X = Q[FRONT]$$
$$FRONT = FRONT + 1$$

Initial empty queue conditions are typically:

$$FRONT = -1, \quad REAR = -1$$

or alternatively $FRONT = 0, REAR = -1$ depending on the specific implementation.

Circular Queue Operations

For a circular queue CQ of size N, the indices wrap around using the modulo operator.

When inserting an element X:

$$REAR = (REAR + 1) \bmod N$$
$$CQ[REAR] = X$$

When deleting an element X:

$$X = CQ[FRONT]$$
$$FRONT = (FRONT + 1) \bmod N$$

Unit 4: Searching and Sorting

Binary Search

To find the middle index mid in a binary search on an array bounded by low and $high$:

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

If the target is greater than the middle element, the lower bound is updated:

$$low = mid + 1$$

Sorting Basics

Typical variable assignments involving keys and pointers (e.g., during Insertion Sort or Quick Sort):

$$key = A[i]$$
$$j = i - 1$$
$$i = low - 1$$