Dsa (1333203) - Problem Solver

Antigravity AI

June 4, 2026

Contents

CHAPTER 1

Basic Concepts of Data Structures and OOP

Data structures provide a logical way to store and organize data, while Object-Oriented Programming (OOP) offers a modular approach to implement these structures. In this chapter, we focus on the fundamental computational techniques to define, allocate, and manipulate basic data representations using OOP concepts in C++.

1.1 Object-Oriented Implementation of Data Structures

The fundamental computational building block in OOP is the Class. Classes allow us to encapsulate data (attributes) and operations (methods) into a single entity called an Abstract Data Type (ADT).

Methodology:

Class Definition and Object Creation To implement a data structure using OOP, follow these computational steps:

1. **Define the Class:** Use the `class` keyword followed by the class name.
2. **Declare Private Data Members:** Define the primitive variables or arrays that store the state (e.g. `int arr[10]; int size;`).
3. **Define Public Member Functions:** Write functions to initialize (constructors), access (getters), modify (setters), and manipulate the data members.
4. **Instantiate Objects:** In the main execution block, create instances of the class (e.g. `MyStruct obj;`) and invoke its methods using the dot operator.

Worked Example:

Implementing a Basic Array Wrapper Class **Problem:** Design an OOP-based wrapper class for a one-dimensional array that initializes the array size and inserts elements sequentially.
Solution:
Step 1: Define the class and data members. We define a class `ArrayWrapper` containing an integer array and a variable `count` to keep track of inserted elements.
Step 2: Define member functions. We need an `insert` method to add elements and a `display` method.
Step 3: Implementation in C++

```
#include <iostream>
using namespace std;

class ArrayWrapper {
private:
    int arr[100];
    int count;
public:
    ArrayWrapper() { // Constructor
```

```
        count = 0;
    }

    void insertElement(int value) {
        if (count < 100) {
            arr[count] = value;
            count++;
        } else {
            cout << "Array is full." << endl;
        }
    }

    void displayElements() {
        for (int i = 0; i < count; i++) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }
};
```

Step 4: Execution In `main()`, we create the object and call the methods:

```
int main() {
    ArrayWrapper aw;
    aw.insertElement(10);
    aw.insertElement(20);
    aw.insertElement(30);
    cout << "Array elements: ";
    aw.displayElements();
    return 0;
}
```

Output: Array elements: 10 20 30

1.2 Dynamic Memory Allocation for Data Structures

Unlike static memory which is allocated at compile time, dynamic memory is allocated at runtime. This is crucial for data structures like linked lists, trees, and dynamic arrays where the size is not known in advance.

Methodology:

Dynamic Memory Allocation using new and delete

1. **Pointer Declaration:** Declare a pointer of the appropriate data type.
2. **Allocate Memory:** Use the `new` operator to request contiguous memory from the heap. For arrays, specify the size (e.g. `int* p = new int[size];`).
3. **Verify Allocation:** Check if the pointer is `NULL` (or `nullptr`) to ensure memory was successfully allocated.
4. **Access and Manipulate:** Use array indexing or pointer arithmetic to process the data.
5. **Deallocate Memory:** Use the `delete` (or `delete[]`) operator to free the memory and prevent memory leaks.

Worked Example:

Dynamic Array Allocation and Summation **Problem:** Write a program to allocate memory for N integers dynamically at runtime, read the integers, and calculate their sum.
Solution:
Step 1: Read the size N. Assume the user inputs $N = 4$.
Step 2: Allocate memory dynamically.

```
int N = 4;
int* ptr = new int[N]; // Dynamic allocation
```

Step 3: Input values and compute sum. Suppose the values provided are 5, 10, 15, and 20.

```
int sum = 0;
ptr[0] = 5;
ptr[1] = 10;
ptr[2] = 15;
ptr[3] = 20;

for(int i = 0; i < N; i++) {
    sum += ptr[i];
}
```

Step 4: Deallocate the memory.

```
delete[] ptr; // Free memory
ptr = nullptr; // Good practice
```

Result: The calculated sum is $5 + 10 + 15 + 20 = 50$.

1.3 Encapsulation and Data Hiding in ADTs

Encapsulation is a core OOP concept heavily used in designing robust data structures. It prevents direct external modification of internal states (such as a stack's top pointer or a queue's front/rear indices).

Methodology:

Implementing Encapsulation for Safe Operations

1. **Restrict Access:** Make all core data storage variables `private`.
2. **Provide Controlled Access:** Create `public` setter and getter methods that include validation logic.
3. **State Management:** Ensure that any method modifying the state updates all dependent variables (e.g. updating size/count) simultaneously to maintain integrity.

Worked Example:

Encapsulated Point Representation **Problem:** Create an encapsulated data structure representing a 2D Cartesian point that prevents negative coordinates.
Solution:
Step 1: Define the Private State.

```
class Point2D {
private:
    int x;
    int y;
```

Step 2: Define Validated Setters.

```
public:
    void setCoordinates(int newX, int newY) {
        if (newX >= 0 && newY >= 0) {
            x = newX;
            y = newY;
        } else {
            x = 0;
            y = 0;
            // Defaults to origin if invalid input
        }
    }
};
```

Step 3: Verification. If a user tries `Point2D p; p.setCoordinates(-5, 10);`, the validation fails, and the internal state defaults to $(0, 0)$. The internal variables `x` and `y` cannot be directly assigned as `p.x = -5`, preventing an invalid state.

1.4 Algorithm Complexity Basics

While understanding basic data structures, it is crucial to analyze the computational efficiency of the operations.

Methodology:

Determining Basic Time Complexity

1. **Identify the Basic Operation:** Find the operation that takes the most time, usually inside the innermost loop.
2. **Count Executions:** Determine how many times this basic operation executes as a function of the input size N.
3. **Drop Constants:** Ignore constant multipliers and lower-order terms to find the Big-O notation $O(f(N))$.

Worked Example:

Analyzing a Sequential Search **Problem:** Determine the time complexity of sequentially searching for an element in an array of size N.
Solution:
Step 1: Identify the basic operation. The basic operation is the comparison: `if (arr[i] == target)`.
Step 2: Count executions. In the worst case (target is at the end or not present), the loop runs exactly N times. Execution count $= N$.
Step 3: Express in Big-O. Since the execution count is directly proportional to N, the worst-case time complexity is $O(N)$.

CHAPTER 2
Stacks and Queues

2.1 Stack Data Structure

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. The operations are performed at only one end, called the `TOP`.

Methodology:

Stack Push and Pop Algorithms Let S be an array representing the stack with a maximum capacity MAX. The variable TOP points to the top element and is initialized to -1.

Algorithm for Push (Insert element X):

1. Check for Overflow: If $TOP = MAX - 1$, print "Stack Overflow" and exit.
2. Increment TOP: $TOP = TOP + 1$.
3. Insert Element: $S[TOP] = X$.

Algorithm for Pop (Remove and return element):

1. Check for Underflow: If $TOP = -1$, print "Stack Underflow" and exit.
2. Read Element: $X = S[TOP]$.
3. Decrement TOP: $TOP = TOP - 1$.
4. Return X.

Worked Example:

Simulating Push and Pop Operations Consider a stack of maximum capacity 4. Simulate the following operations and show the stack status and TOP value after each operation: Push(10), Push(20), Pop(), Push(30), Push(40), Push(50).

Solution: Initially, an empty stack has $TOP = -1$. $MAX = 4$.

Operation	TOP	Stack Content (Bottom to Top)
Initial State	-1	Empty
Push(10)	0	[10]
Push(20)	1	[10 20]
Pop()	0	[10] (Popped 20)
Push(30)	1	[10 30]
Push(40)	2	[10 30 40]
Push(50)	3	[10 30 40 50]

If we now perform Push(60), TOP is equal to $MAX - 1$ (which is $4 - 1 = 3$), so a "Stack Overflow" error occurs.

Methodology:

Stack Peep and Change Algorithms **Algorithm for Peep (Find I-th element from top):**

1. Validate Index: If $(TOP - I + 1) < 0$, print "Invalid Index" and exit.
2. Read Element: Return $S[TOP - I + 1]$.

Algorithm for Change (Change I-th element from top to X):

1. Validate Index: If $(TOP - I + 1) < 0$, print "Invalid Index" and exit.
2. Update Element: $S[TOP - I + 1] = X$.

Worked Example:

Simulating Peep and Change Given a stack containing elements [15, 25, 35, 45] (where 15 is bottom and 45 is top). The TOP is 3. Find the element at $I = 2$ using Peep and change the element at $I = 3$ to 99.
Solution: Current Stack: $S[0] = 15$, $S[1] = 25$, $S[2] = 35$, $S[3] = 45$. $TOP = 3$.
1. Peep with $I = 2$:

- Check index: $TOP - I + 1 = 3 - 2 + 1 = 2$.
- Index 2 is valid.
- Return $S[2]$, which is 35.

2. Change with $I = 3$ to $X = 99$:

- Check index: $TOP - I + 1 = 3 - 3 + 1 = 1$.
- Index 1 is valid.
- Update $S[1] = 99$.

The updated stack is [15, 99, 35, 45].

2.2 Applications of Stack: Expression Conversion

Expressions can be represented in Infix, Postfix, or Prefix forms. A stack is heavily used to convert between these forms without altering the order of operations dictated by precedence.

Methodology:

Infix to Postfix Conversion Algorithm To convert an infix expression to postfix notation:

1. Initialize an empty stack for operators and an empty list for the postfix output.
2. Append a right parenthesis ‘)‘ to the end of the infix expression and push a left parenthesis ‘(‘ onto the stack.
3. Scan the infix expression from left to right:
   (a) If the scanned character is an operand, add it to the postfix output.
   (b) If the scanned character is a left parenthesis ‘(‘, push it onto the stack.
   (c) If the scanned character is an operator (e.g. ‘+‘, ‘-‘, ‘*‘, ‘/‘, ‘^‘), repeatedly pop operators from the stack and add them to the postfix output until an operator with strictly lower precedence is at the top of the stack. Then push the current operator onto the stack.
   (d) If the scanned character is a right parenthesis ‘)‘, repeatedly pop operators and add them to the postfix output until a left parenthesis ‘(‘ is encountered. Pop and discard the left parenthesis.

4. Repeat step 3 until the infix expression is fully processed and the stack is empty.

Precedence rules: `^` (Highest) > `*`, `/` > `+`, `-` (Lowest).

Worked Example:

Convert Infix to Postfix Convert the infix expression `A * (B + C) / D - E` to postfix notation.
Solution: Add ')' to expression: `A * (B + C) / D - E )` Push '(' to Stack.

Scanned Token	Stack	Postfix Output
Initial State	(	
A	(	A
*	(*	A
(	(* (	A
B	(* (	A B
+	(* (+	A B
C	(* (+	A B C
)	(*	A B C +
/	(/	A B C + *
D	(/	A B C + * D
-	(-	A B C + * D /
E	(-	A B C + * D / E
)	Empty	A B C + * D / E -

The resulting postfix expression is: **`A B C + * D / E -`**

2.3 Applications of Stack: Expression Evaluation

Once an expression is converted to postfix or prefix, it can be evaluated easily using a stack without needing to know precedence rules.

Methodology:

Evaluation of Postfix Expression Algorithm

1. Initialize an empty stack.
2. Append a special symbol (like '#' or ')') to the end of the postfix expression.
3. Scan the postfix expression from left to right:
   (a) If the scanned character is an operand, push its value onto the stack.
   (b) If the scanned character is an operator (let's say O), pop the top two elements from the stack. Let A be the top element and B be the second top element.
   (c) Evaluate $R = B\ O\ A$.
   (d) Push the result R back onto the stack.
4. When the special symbol is reached, the result of the expression is the only element remaining in the stack. Pop and print it.

Worked Example:

Evaluate Postfix Expression Evaluate the postfix expression `5 3 + 8 2 / * 4 -`.
Solution:

Scanned Token	Stack	Operation Performed
5	5	Push 5
3	5 3	Push 3
+	8	Pop 3, Pop 5. $5+3=8$. Push 8.
8	8 8	Push 8
2	8 8 2	Push 2
/	8 4	Pop 2, Pop 8. $8/2=4$. Push 4.
*	32	Pop 4, Pop 8. $8*4=32$. Push 32.
4	32 4	Push 4
-	28	Pop 4, Pop 32. $32-4=28$. Push 28.

The final evaluated result is **28**.

2.4 Queue Data Structure

A Queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. Elements are added at the `REAR` end and removed from the `FRONT` end.

Methodology:

Linear Queue Insert and Delete Algorithms Let Q be an array representing the queue with maximum capacity N. Two variables, $FRONT$ and $REAR$, are both initialized to -1.
Algorithm for Enqueue (Insert element X):

1. Check Overflow: If $REAR = N-1$, print "Queue Overflow" and exit.
2. Update Front: If $FRONT = -1$ (Queue is empty), set $FRONT = 0$.
3. Update Rear: $REAR = REAR + 1$.
4. Insert Element: $Q[REAR] = X$.

Algorithm for Dequeue (Remove and return element):

1. Check Underflow: If $FRONT = -1$, print "Queue Underflow" and exit.
2. Read Element: $X = Q[FRONT]$.
3. Update Front: If $FRONT = REAR$, the queue becomes empty. Set $FRONT = -1$ and $REAR = -1$. Otherwise, increment front: $FRONT = FRONT + 1$.
4. Return X.

Worked Example:

Simulating Linear Queue Operations Simulate a linear queue of maximum size 4. Perform the operations: Insert(10), Insert(20), Delete(), Insert(30), Insert(40), Delete().
Solution: $N = 4$. Initial State: $FRONT = -1$, $REAR = -1$.

Operation	FRONT	REAR	Queue Elements
Initial State	-1	-1	Empty
Insert(10)	0	0	[10]
Insert(20)	0	1	[10 20]
Delete()	1	1	[20] (Deleted 10)
Insert(30)	1	2	[20 30]
Insert(40)	1	3	[20 30 40]
Delete()	2	3	[30 40] (Deleted 20)

If we try to Insert(50) now, even though there is empty space at index 0 and 1, $REAR = 3 = N-1$. It will

result in "Queue Overflow", which is a major disadvantage of linear queues.

2.5 Circular Queue

A circular queue connects the last position back to the first position to form a circle, eliminating the empty space wastage seen in linear queues.

Methodology:

Circular Queue Insert and Delete Algorithms Let CQ be an array of size N. $FRONT$ and $REAR$ are initially -1.

Algorithm for Circular Enqueue (Insert element X):

1. Check Overflow: If $(REAR + 1) \pmod N = FRONT$, print "Queue Overflow" and exit.
2. Update Front: If $FRONT = -1$ (Empty), set $FRONT = 0$ and $REAR = 0$.
3. Update Rear: If queue is not empty, set $REAR = (REAR + 1) \pmod N$.
4. Insert Element: $CQ[REAR] = X$.

Algorithm for Circular Dequeue (Remove element):

1. Check Underflow: If $FRONT = -1$, print "Queue Underflow" and exit.
2. Read Element: $X = CQ[FRONT]$.
3. Update Front:
   (a) If $FRONT = REAR$ (Only one element was present), set $FRONT = -1$ and $REAR = -1$.
   (b) Else, $FRONT = (FRONT + 1) \pmod N$.
4. Return X.

Worked Example:

Simulating Circular Queue Operations Consider a circular queue of size $N = 4$. Initial State: $FRONT = -1, REAR = -1$. Perform operations: Insert A, Insert B, Insert C, Insert D, Delete, Delete, Insert E, Insert F.

Solution:

Operation	FRONT	REAR	Array Content [0, 1, 2, 3]
Insert A	0	0	[A, -, -, -]
Insert B	0	1	[A, B, -, -]
Insert C	0	2	[A, B, C, -]
Insert D	0	3	[A, B, C, D]
Delete (Returns A)	1	3	[-, B, C, D]
Delete (Returns B)	2	3	[-, -, C, D]
Insert E	2	0	[E, -, C, D] $(REAR = (3+1)\%4 = 0)$
Insert F	2	1	[E, F, C, D] $(REAR = (0+1)\%4 = 1)$

If we try to Insert G, $REAR = (1 + 1)\%4 = 2$. This equals $FRONT$, so the overflow condition is triggered correctly.

2.6 Double-Ended Queue (Deque) and Priority Queue

Methodology:

Deque Classification A Double-Ended Queue allows insertion and deletion operations at both ends (Front and Rear). It does not follow strict FIFO. There are two restricted variants:

1. **Input Restricted Deque:** Insertion is restricted to one end (Rear), but deletion can be performed at both ends (Front and Rear).
2. **Output Restricted Deque:** Deletion is restricted to one end (Front), but insertion can be performed at both ends (Front and Rear).

Methodology:

Priority Queue Concepts A Priority Queue is a special type of queue in which each element is associated with a priority value. Elements are served on the basis of their priority.

1. Elements with higher priority are dequeued before elements with lower priority.
2. If two elements have the same priority, they are served according to their order in the queue (FIFO).

Implementation is typically done using Heaps or sorted/unsorted Arrays and Linked Lists.

CHAPTER 3
Linked List

3.1 Singly Linked List Operations

A singly linked list is a linear data structure where each element (node) contains a data part and a reference (link) to the next node in the sequence.

Methodology:

Traversal of a Singly Linked List Traversal involves visiting every node in the linked list exactly once.

1. Initialize a pointer variable `PTR` to the starting node (`START`).
2. Repeat Steps 3 and 4 while `PTR` is not `NULL`.
3. Process the data at the current node (`PTR.INFO`).
4. Update the pointer to the next node: `PTR = PTR.LINK`.
5. Exit.

Worked Example:

Trace Singly Linked List Traversal Given a singly linked list starting at address 1000 with the following nodes in memory:

Address	INFO	LINK
1000	15	1008
1004	42	NULL
1008	20	1004

Trace the traversal algorithm and find the sequence of values processed. The list begins at `START` = 1000.
Solution:

1. **Initialization:** Set `PTR` = `START` = 1000.
2. **Iteration 1:**
   - `PTR` (1000) $\neq$ `NULL`.
   - Process `PTR.INFO`: Value at 1000 is **15**.
   - Update pointer: `PTR` = `PTR.LINK` = 1008.
3. **Iteration 2:**
   - `PTR` (1008) $\neq$ `NULL`.
   - Process `PTR.INFO`: Value at 1008 is **20**.
   - Update pointer: `PTR` = `PTR.LINK` = 1004.
4. **Iteration 3:**

- PTR (1004) $\neq$ NULL.
- Process PTR.INFO: Value at 1004 is **42**.
- Update pointer: PTR = PTR.LINK = NULL.

5. **Termination:** PTR is NULL. Loop terminates.

The processed sequence of values is: 15 $\rightarrow$ 20 $\rightarrow$ 42.

Methodology:

Insertion at the Front of a Singly Linked List To insert a new element at the beginning of the list:

1. Allocate memory for the new node NEW.
2. Assign data: NEW.INFO = VAL.
3. Link the new node to the current first node: NEW.LINK = START.
4. Update the start pointer: START = NEW.

Methodology:

Insertion at the End of a Singly Linked List To insert a new element at the end of the list:

1. Allocate memory for NEW. Assign NEW.INFO = VAL and NEW.LINK = NULL.
2. If START == NULL (list is empty) set START = NEW and Exit.
3. Else set PTR = START.
4. Repeat while PTR.LINK is not NULL: PTR = PTR.LINK.
5. Set PTR.LINK = NEW.

Worked Example:

Performing Multiple Insertions Starting with an empty singly linked list, trace the state of the list after each of the following operations: 1. Insert 10 at the front. 2. Insert 20 at the end. 3. Insert 5 at the front.
Solution:
Initial State: START = NULL
Operation 1: Insert 10 at the front

- Create node NEW with INFO = 10.
- NEW.LINK = START $\Rightarrow$ NEW.LINK = NULL.
- START = NEW.
- **List State:** START $\rightarrow$ [10 | NULL]

Operation 2: Insert 20 at the end

- Create node NEW with INFO = 20 and LINK = NULL.
- START is not NULL. Set PTR = START. (Node 10).
- PTR.LINK is NULL. Loop skips.
- PTR.LINK = NEW.
- **List State:** START $\rightarrow$ [10 | $\rightarrow$] $\rightarrow$ [20 | NULL]

Operation 3: Insert 5 at the front

- Create node `NEW` with INFO = 5.
- `NEW.LINK = START`. (Points to Node 10).
- `START = NEW`.
- **List State:** `START` → [5 | →] → [10 | →] → [20 | NULL]

Methodology:

Deletion of the First Node in a Singly Linked List

1. If `START == NULL` print "Underflow" and Exit.
2. Set `PTR = START`.
3. Update start pointer to second node: `START = START.LINK`.
4. Deallocate memory for `PTR`.

Methodology:

Deletion of a Specific Node by Value To delete the first occurrence of a node containing `VAL`:

1. If `START == NULL` print "Underflow" and Exit.
2. If `START.INFO == VAL` then `PTR = START`, `START = START.LINK`, free `PTR` and Exit.
3. Initialize `PTR = START` and `PREV = NULL`.
4. Repeat while `PTR` is not `NULL` and `PTR.INFO` $\neq$ `VAL`:
   - `PREV = PTR`
   - `PTR = PTR.LINK`
5. If `PTR == NULL` print "Node not found" and Exit.
6. Update link: `PREV.LINK = PTR.LINK`.
7. Deallocate memory for `PTR`.

Worked Example:

Tracing Node Deletion Given the linked list: `START` → 11 → 22 → 33 → 44 → `NULL`. Trace the steps to delete the node with value 33.
Solution:

1. `START` is not `NULL`. `START.INFO` (11) $\neq$ 33.
2. Initialize `PTR` = Node(11), `PREV` = `NULL`.
3. **Iteration 1:** `PTR.INFO` = 11. Condition holds.
   - `PREV` = Node(11)
   - `PTR` = `PTR.LINK` = Node(22)
4. **Iteration 2:** `PTR.INFO` = 22. Condition holds.
   - `PREV` = Node(22)

- PTR = PTR.LINK = Node(33)

5. **Iteration 3:** PTR.INFO = 33. Condition PTR.INFO ≠ 33 fails. Loop terminates.
6. Update link: PREV.LINK = PTR.LINK.
   - Node(22)'s link now points to Node(33)'s link, which is Node(44).
7. Free PTR (Node(33)).

Final List State: START → 11 → 22 → 44 → NULL

3.2 Doubly Linked List Operations

A doubly linked list contains nodes with three parts: a left link (LPTR), the data (INFO), and a right link (RPTR). This allows traversal in both directions.

Methodology:

Insertion at the End of a Doubly Linked List

1. Allocate memory for NEW. Set NEW.INFO = VAL and NEW.RPTR = NULL.
2. If START == NULL: Set NEW.LPTR = NULL, START = NEW and Exit.
3. Initialize PTR = START.
4. Repeat while PTR.RPTR ≠ NULL: PTR = PTR.RPTR.
5. Set PTR.RPTR = NEW.
6. Set NEW.LPTR = PTR.

Worked Example:

Building a Doubly Linked List Trace the creation of a doubly linked list by inserting values 5 and 10 at the end.
Solution:
Operation 1: Insert 5

- Create NEW: LPTR=?, INFO=5, RPTR=NULL.
- START is NULL.
- NEW.LPTR = NULL, START = NEW.
- **State:** NULL ← [5] → NULL

Operation 2: Insert 10

- Create NEW: LPTR=?, INFO=10, RPTR=NULL.
- PTR = Node(5). PTR.RPTR is NULL, loop terminates immediately.
- PTR.RPTR = NEW.
- NEW.LPTR = PTR.
- **State:** NULL ← [5] ↔ [10] → NULL

3.3 Circular Linked List

In a circular singly linked list, the last node's link points back to the `START` node instead of `NULL`.

Methodology:

Traversal of a Circular Linked List

1. If `START == NULL` print "Empty List" and Exit.
2. Initialize `PTR = START`.
3. Repeat Steps 4 and 5.
4. Process `PTR.INFO`.
5. `PTR = PTR.LINK`.
6. Until `PTR == START`.

Worked Example:

Circular Linked List Element Count Given a circular linked list starting at Node(10), containing nodes [10] → [20] → [30] → (points back to [10]). Apply the traversal algorithm to count the number of nodes.
Solution:

1. `START` is not `NULL`. Initialize `COUNT` = 0.
2. `PTR = START` (Node with 10).
3. **Loop execution:**
   - **Pass 1:** Process node (10). `COUNT` = 1. `PTR` = Node(20). `PTR` ≠ `START`, continue.
   - **Pass 2:** Process node (20). `COUNT` = 2. `PTR` = Node(30). `PTR` ≠ `START`, continue.
   - **Pass 3:** Process node (30). `COUNT` = 3. `PTR` = `PTR.LINK` = Node(10).
4. `PTR == START` (Node(10) == Node(10)). Loop terminates.
5. **Final Result:** Total nodes counted is 3.

CHAPTER 4

Searching and Sorting

4.1 Searching Techniques

4.1.1 Linear Search

Methodology:

Linear Search Algorithm Linear Search is a sequential search algorithm that finds an element in a list by checking every element one by one until a match is found.
Algorithm Steps:

1. Start from the first element (index $i = 0$) of the array A of size N.
2. Compare the target value X with $A[i]$.
3. If $X == A[i]$, the search is successful. Return index i.
4. If $X \neq A[i]$, increment i by 1.
5. Repeat steps 2-4 until the end of the array is reached ($i = N$).
6. If the end of the array is reached without a match, the search is unsuccessful. Return -1.

Worked Example:

Linear Search on an Array Perform Linear Search to find the target value $X = 35$ in the array $A = [10, 25, 40, 35, 50, 15]$.
Solution:

1. **Initial state:** $X = 35$, Array size $N = 6$.
2. **Iteration 1 ($i = 0$):** Compare X with $A[0]$. $35 \neq 10$.
3. **Iteration 2 ($i = 1$):** Compare X with $A[1]$. $35 \neq 25$.
4. **Iteration 3 ($i = 2$):** Compare X with $A[2]$. $35 \neq 40$.
5. **Iteration 4 ($i = 3$):** Compare X with $A[3]$. $35 == 35$. Match found!

Conclusion: The target value 35 is found at index 3.

4.1.2 Binary Search

Methodology:

Binary Search Algorithm Binary Search is an efficient searching algorithm that finds the position of a target value within a **sorted** array. It works by repeatedly dividing the search interval in half.
Algorithm Steps:

1. Ensure the array A of size N is sorted in ascending order.
2. Initialize $low = 0$ and $high = N - 1$.
3. Loop while $low \leq high$:
   (a) Calculate the middle index: $mid = \lfloor \frac{low+high}{2} \rfloor$.
   (b) If $A[mid] == X$, the target is found. Return mid.
   (c) If $A[mid] < X$, the target must be in the right half. Update $low = mid + 1$.
   (d) If $A[mid] > X$, the target must be in the left half. Update $high = mid - 1$.
4. If the loop terminates without a match, return -1 (target not found).

Worked Example:

Binary Search on a Sorted Array Perform Binary Search to find the target value $X = 64$ in the sorted array $A = [12, 23, 34, 45, 56, 64, 78, 89, 90]$.
Solution:

1. **Initial state:** $N = 9$, $low = 0$, $high = 8$, $X = 64$.
2. **Iteration 1:**
   - $mid = \lfloor \frac{0+8}{2} \rfloor = 4$.
   - $A[4] = 56$.
   - $56 < 64$, so update $low = mid + 1 = 5$.
3. **Iteration 2:**
   - $low = 5$, $high = 8$.
   - $mid = \lfloor \frac{5+8}{2} \rfloor = 6$.
   - $A[6] = 78$.
   - $78 > 64$, so update $high = mid - 1 = 5$.
4. **Iteration 3:**
   - $low = 5$, $high = 5$.
   - $mid = \lfloor \frac{5+5}{2} \rfloor = 5$.
   - $A[5] = 64$.
   - $64 == 64$. Match found!

Conclusion: The target value 64 is found at index 5.

4.2 Sorting Techniques

4.2.1 Bubble Sort

Methodology:

Bubble Sort Algorithm 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. This pass is repeated until the list is sorted.

Algorithm Steps:

1. Loop i from 0 to $N-1$ (representing the number of passes).
2. Within each pass, 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 pass, the largest element among the unsorted elements "bubbles up" to its correct position.
6. If no two elements were swapped by the inner loop, the array is already sorted, and the algorithm can terminate early.

Worked Example:

Tracing Bubble Sort Sort the array $A = [50, 30, 20, 40, 10]$ using Bubble Sort.

Solution:

- **Pass 1:** $(i = 0)$
  - Compare 50 and 30: $50 > 30$, swap $\rightarrow [30, 50, 20, 40, 10]$
  - Compare 50 and 20: $50 > 20$, swap $\rightarrow [30, 20, 50, 40, 10]$
  - Compare 50 and 40: $50 > 40$, swap $\rightarrow [30, 20, 40, 50, 10]$
  - Compare 50 and 10: $50 > 10$, swap $\rightarrow [30, 20, 40, 10, \mathbf{50}]$ (Largest element is sorted)
- **Pass 2:** $(i = 1)$
  - Compare 30 and 20: $30 > 20$, swap $\rightarrow [20, 30, 40, 10, 50]$
  - Compare 30 and 40: $30 \not> 40$, no swap $\rightarrow [20, 30, 40, 10, 50]$
  - Compare 40 and 10: $40 > 10$, swap $\rightarrow [20, 30, 10, \mathbf{40}, \mathbf{50}]$
- **Pass 3:** $(i = 2)$
  - Compare 20 and 30: $20 \not> 30$, no swap $\rightarrow [20, 30, 10, 40, 50]$
  - Compare 30 and 10: $30 > 10$, swap $\rightarrow [20, 10, \mathbf{30}, \mathbf{40}, \mathbf{50}]$
- **Pass 4:** $(i = 3)$
  - Compare 20 and 10: $20 > 10$, swap $\rightarrow [\mathbf{10}, \mathbf{20}, \mathbf{30}, \mathbf{40}, \mathbf{50}]$

Conclusion: The sorted array is $[10, 20, 30, 40, 50]$.

4.2.2 Selection Sort

Methodology:

Selection Sort Algorithm Selection Sort divides the array into a sorted and an unsorted region. It repeatedly selects the smallest element from the unsorted region and swaps it with the leftmost unsorted element.

Algorithm Steps:

1. Loop i from 0 to $N-2$. This represents the boundary of the sorted region.
2. Assume $A[i]$ is the minimum element. Set $min\_index = i$.
3. Loop j from $i+1$ to $N-1$ to find the actual minimum element in the unsorted region.
4. If $A[j] < A[min\_index]$, update $min\_index = j$.
5. After finding the minimum element in the unsorted region, swap $A[i]$ with $A[min\_index]$.

Worked Example:

Tracing Selection Sort Sort the array $A = [64, 25, 12, 22, 11]$ using Selection Sort.
Solution:

- **Pass 1 ($i = 0$):**
  - Unsorted region: $[64, 25, 12, 22, 11]$. Minimum is 11 at index 4.
  - Swap $A[0]$ (64) with $A[4]$ (11).
  - Array becomes: $[\mathbf{11}, 25, 12, 22, 64]$.
- **Pass 2 ($i = 1$):**
  - Unsorted region: $[25, 12, 22, 64]$. Minimum is 12 at index 2.
  - Swap $A[1]$ (25) with $A[2]$ (12).
  - Array becomes: $[\mathbf{11}, \mathbf{12}, 25, 22, 64]$.
- **Pass 3 ($i = 2$):**
  - Unsorted region: $[25, 22, 64]$. Minimum is 22 at index 3.
  - Swap $A[2]$ (25) with $A[3]$ (22).
  - Array becomes: $[\mathbf{11}, \mathbf{12}, \mathbf{22}, 25, 64]$.
- **Pass 4 ($i = 3$):**
  - Unsorted region: $[25, 64]$. Minimum is 25 at index 3.
  - Swap $A[3]$ (25) with $A[3]$ (25) (No change).
  - Array becomes: $[\mathbf{11}, \mathbf{12}, \mathbf{22}, \mathbf{25}, \mathbf{64}]$.

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

4.2.3 Insertion Sort

Methodology:

Insertion Sort Algorithm Insertion Sort builds the sorted array one element at a time by repeatedly taking the next unsorted element and inserting it into its correct position within the already sorted portion of the array.
Algorithm Steps:

1. Assume the first element ($A[0]$) is already sorted.
2. Loop i from 1 to $N-1$.
3. Let $key = A[i]$. This is the element to be inserted.
4. Initialize a variable $j = i - 1$ to traverse the sorted portion backwards.

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

Worked Example:

Tracing Insertion Sort Sort the array $A = [45, 12, 34, 23, 10]$ using Insertion Sort.
Solution:

- **Initial:** Sorted portion is [**45**], unsorted is $[12, 34, 23, 10]$.
- **Pass 1 ($i = 1$):** Insert 12.
  - $key = 12$. Compare with 45. $45 > 12$, shift 45 to the right.
  - Insert 12. Array becomes: [**12**, **45**, 34, 23, 10].
- **Pass 2 ($i = 2$):** Insert 34.
  - $key = 34$. Compare with 45. $45 > 34$, shift 45 right.
  - Compare with 12. $12 \not> 34$, stop shifting.
  - Insert 34. Array becomes: [**12**, **34**, **45**, 23, 10].
- **Pass 3 ($i = 3$):** Insert 23.
  - $key = 23$. Compare with 45, shift 45 right.
  - Compare with 34, shift 34 right.
  - Compare with 12, stop shifting.
  - Insert 23. Array becomes: [**12**, **23**, **34**, **45**, 10].
- **Pass 4 ($i = 4$):** Insert 10.
  - $key = 10$. Compare with $45, 34, 23, 12$, shift all of them to the right.
  - Insert 10 at the beginning.
  - Array becomes: [**10**, **12**, **23**, **34**, **45**].

Conclusion: The sorted array is $[10, 12, 23, 34, 45]$.

4.2.4 Quick Sort

Methodology:

Quick Sort Algorithm Quick Sort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. After partitioning, the pivot is placed in its correct sorted position, with smaller elements to its left and larger elements to its right.
Algorithm Steps for Quick Sort (low, $high$):

1. If $low < high$:
   (a) Find the partition index pi using the Partition algorithm: $pi = Partition(A, low, high)$.
   (b) Recursively call Quick Sort for the left sub-array: $QuickSort(A, low, pi - 1)$.
   (c) Recursively call Quick Sort for the right sub-array: $QuickSort(A, pi + 1, high)$.

Partition Algorithm (low, $high$):

1. Choose the last element as the pivot: $pivot = A[high]$.
2. Initialize $i = low - 1$ (index of smaller element).
3. Loop j from low to $high - 1$:
   (a) If $A[j] < pivot$:
   (b) Increment i.
   (c) Swap $A[i]$ and $A[j]$.
4. Swap $A[i+1]$ and $A[high]$ to place the pivot in the correct position.
5. Return $i + 1$.

Worked Example:

Tracing Quick Sort Partitioning Given the array $A = [30, 80, 10, 90, 40, 50, 70]$. Partition it using the last element as the pivot.
Solution:

1. **Initial state:** $low = 0$, $high = 6$, $pivot = A[6] = 70$, $i = -1$.
2. **Iteration $j = 0$:** $A[0] = 30$. $30 < 70$.
   - $i = 0$, Swap $A[0]$ and $A[0]$. Array: $[30, 80, 10, 90, 40, 50, 70]$.
3. **Iteration $j = 1$:** $A[1] = 80$. $80 \not< 70$. No action.
4. **Iteration $j = 2$:** $A[2] = 10$. $10 < 70$.
   - $i = 1$, Swap $A[1]$ and $A[2]$. Array: $[30, 10, 80, 90, 40, 50, 70]$.
5. **Iteration $j = 3$:** $A[3] = 90$. $90 \not< 70$. No action.
6. **Iteration $j = 4$:** $A[4] = 40$. $40 < 70$.
   - $i = 2$, Swap $A[2]$ and $A[4]$. Array: $[30, 10, 40, 90, 80, 50, 70]$.
7. **Iteration $j = 5$:** $A[5] = 50$. $50 < 70$.
   - $i = 3$, Swap $A[3]$ and $A[5]$. Array: $[30, 10, 40, 50, 80, 90, 70]$.
8. **Final Swap:** Swap $A[i+1]$ ($A[4] = 80$) with the pivot ($A[6] = 70$).
9. **Partitioned Array:** $[30, 10, 40, 50, \mathbf{70}, 90, 80]$.

Conclusion: The pivot 70 is now at index 4. Elements to the left are smaller, elements to the right are larger.

4.2.5 Merge Sort

Methodology:

Merge Sort Algorithm Merge Sort is a Divide and Conquer algorithm that divides the input array into two halves, recursively sorts the two halves, and then merges the sorted halves back together.
Algorithm Steps for Merge Sort (low, $high$):

1. If $low < high$:
   (a) Find the middle point: $mid = \lfloor \frac{low+high}{2} \rfloor$.

(b) Recursively call Merge Sort for the first half: $MergeSort(A, low, mid)$.

(c) Recursively call Merge Sort for the second half: $MergeSort(A, mid + 1, high)$.

(d) Merge the two sorted halves: $Merge(A, low, mid, high)$.

Merge Algorithm (low, mid, $high$):

1. Create temporary arrays L and R to hold the elements of the two halves.
2. Copy data to L (size $n_1 = mid - low + 1$) and R (size $n_2 = high - mid$).
3. Initialize pointers $i = 0$ (for L), $j = 0$ (for R), and $k = low$ (for merged array A).
4. While $i < n_1$ and $j < n_2$:
   (a) If $L[i] \leq R[j]$, set $A[k] = L[i]$ and increment i.
   (b) Else, set $A[k] = R[j]$ and increment j.
   (c) Increment k.
5. Copy any remaining elements of L to A.
6. Copy any remaining elements of R to A.

Worked Example:

Merging Two Sorted Arrays Merge two sorted arrays $L = [10, 30, 50]$ and $R = [20, 40, 60]$ into a single sorted array A.

Solution:

1. **Initial State:** $i = 0, j = 0, k = 0$. $A = []$.
2. **Step 1:** Compare $L[0] = 10$ and $R[0] = 20$.
   - $10 \leq 20 \implies A = [10]$, $i = 1, k = 1$.
3. **Step 2:** Compare $L[1] = 30$ and $R[0] = 20$.
   - $30 > 20 \implies A = [10, 20]$, $j = 1, k = 2$.
4. **Step 3:** Compare $L[1] = 30$ and $R[1] = 40$.
   - $30 \leq 40 \implies A = [10, 20, 30]$, $i = 2, k = 3$.
5. **Step 4:** Compare $L[2] = 50$ and $R[1] = 40$.
   - $50 > 40 \implies A = [10, 20, 30, 40]$, $j = 2, k = 4$.
6. **Step 5:** Compare $L[2] = 50$ and $R[2] = 60$.
   - $50 \leq 60 \implies A = [10, 20, 30, 40, 50]$, $i = 3, k = 5$.
7. **Step 6:** i has reached the end of L. Copy remaining elements of R (which is 60).
   - $A = [10, 20, 30, 40, 50, 60]$.

Conclusion: The merged sorted array is $[10, 20, 30, 40, 50, 60]$.

CHAPTER 5

Trees

Trees are non-linear data structures used to represent hierarchical relationships. In this chapter, we focus on the construction, traversal, and modification of Binary Trees and Binary Search Trees (BST). These concepts are heavily emphasized in computational problems.

5.1 Binary Search Tree (BST) Construction

A Binary Search Tree is a binary tree where every node follows a specific ordering property: the value of the left child is strictly less than the parent node, and the value of the right child is strictly greater than or equal to the parent node.

Methodology:

Constructing a Binary Search Tree To build a BST from a sequence of elements, follow these steps:

1. **Initialize**: Make the first element of the sequence the **root** of the tree.
2. **Process remaining elements**: For each subsequent element in the sequence, start at the root and compare the element's value with the current node's value.
3. **Traverse Left**: If the element is less than the current node, move to the left child.
4. **Traverse Right**: If the element is greater than or equal to the current node, move to the right child.
5. **Insert**: When you reach an empty spot (a NULL reference), insert the new element as a new leaf node at that position.

Worked Example:

Construct a BST from a sequence of numbers Construct a Binary Search Tree for the following sequence of numbers: 50 30 20 40 70 60 80.
Solution: We will insert each element one by one.
Step 1: Insert 50

- The tree is empty. 50 becomes the **Root**.

Step 2: Insert 30

- Compare 30 with 50. Since $30 < 50$, move left.
- Left child of 50 is empty. Insert 30 as the left child of 50.

Step 3: Insert 20

- Compare 20 with 50. Since $20 < 50$, move left to 30.
- Compare 20 with 30. Since $20 < 30$, move left.
- Left child of 30 is empty. Insert 20 as the left child of 30.

Step 4: Insert 40

- Compare 40 with 50 ($40 < 50$), move left to 30.
- Compare 40 with 30 ($40 > 30$), move right.
- Insert 40 as the right child of 30.

Step 5: Insert 70

- Compare 70 with 50 ($70 > 50$), move right.
- Insert 70 as the right child of 50.

Step 6: Insert 60

- Compare 60 with 50 ($60 > 50$), move right to 70.
- Compare 60 with 70 ($60 < 70$), move left.
- Insert 60 as the left child of 70.

Step 7: Insert 80

- Compare 80 with 50 ($80 > 50$), move right to 70.
- Compare 80 with 70 ($80 > 70$), move right.
- Insert 80 as the right child of 70.

Final Tree Structure:

- Root: 50
- Left Subtree of 50: Root is 30 with Left Child 20 and Right Child 40.
- Right Subtree of 50: Root is 70 with Left Child 60 and Right Child 80.

5.2 Tree Traversals

Traversal is the process of visiting each node in the tree exactly once in a specific order. The three most common depth-first traversal methods are Preorder, Inorder and Postorder.

Methodology:

Binary Tree Traversal Algorithms **1. Preorder Traversal (Node-Left-Right):**

1. Visit the root node.
2. Recursively perform Preorder traversal on the left subtree.
3. Recursively perform Preorder traversal on the right subtree.

2. Inorder Traversal (Left-Node-Right):

1. Recursively perform Inorder traversal on the left subtree.
2. Visit the root node.
3. Recursively perform Inorder traversal on the right subtree.

3. Postorder Traversal (Left-Right-Node):

1. Recursively perform Postorder traversal on the left subtree.
2. Recursively perform Postorder traversal on the right subtree.
3. Visit the root node.

Worked Example:

Find Traversals of a Given Binary Tree Consider the BST constructed in the previous example with root 50, left child 30 (which has children 20 and 40) and right child 70 (which has children 60 and 80). Find its Preorder, Inorder and Postorder traversals.

Solution:

1. Preorder Traversal (Node-Left-Right):

- Start at Root 50. Output: **50**
- Go to left subtree of 50 (Root 30). Output: **30**
- Go to left subtree of 30 (Root 20). Output: **20**
- Right subtree of 20 is empty. Go back to 30 and visit its right subtree (Root 40). Output: **40**
- Left subtree of 50 is fully visited. Go to right subtree of 50 (Root 70). Output: **70**
- Go to left subtree of 70 (Root 60). Output: **60**
- Right subtree of 60 is empty. Go back to 70 and visit its right subtree (Root 80). Output: **80**

Preorder Sequence: 50 30 20 40 70 60 80

2. Inorder Traversal (Left-Node-Right):

- Go to leftmost node: 20. Output: **20**
- Visit parent of 20: 30. Output: **30**
- Visit right child of 30: 40. Output: **40**
- Left subtree of 50 is fully visited. Visit root: 50. Output: **50**
- Go to leftmost node in right subtree: 60. Output: **60**
- Visit parent of 60: 70. Output: **70**
- Visit right child of 70: 80. Output: **80**

Inorder Sequence: 20 30 40 50 60 70 80 *(Notice that the Inorder traversal of a BST always yields a sorted sequence.)*

3. Postorder Traversal (Left-Right-Node):

- Go to leftmost node: 20. Output: **20**
- Go to right sibling of 20 (right child of 30): 40. Output: **40**
- Visit parent of 20 and 40: 30. Output: **30**
- Go to right subtree of 50, find its leftmost node: 60. Output: **60**
- Go to right sibling of 60 (right child of 70): 80. Output: **80**
- Visit parent of 60 and 80: 70. Output: **70**
- Visit root: 50. Output: **50**

Postorder Sequence: 20 40 30 60 80 70 50

5.3 Deleting a Node from a Binary Search Tree

Deletion in a BST requires restructuring the tree to preserve the BST properties after a node is removed.

Methodology:

BST Node Deletion Algorithm To delete a node N from a BST, identify which of the following three cases applies:

1. **Case 1: Node N is a leaf node (no children).** Simply remove the node from the tree by setting its parent's pointer to NULL.
2. **Case 2: Node N has exactly one child.** Remove N and connect N's parent directly to N's single child.
3. **Case 3: Node N has two children.**
   (a) Find the **Inorder Successor** of N (the node with the smallest value in N's right subtree).
   (b) Replace the value of N with the value of the Inorder Successor.
   (c) Delete the original Inorder Successor node from the right subtree. Since the Inorder Successor is the minimum value in the right subtree, it will have at most one child (a right child), meaning its deletion will fall under Case 1 or Case 2.

Worked Example:

Delete nodes from a BST Given a BST with nodes {20 30 40 50 60 70 80} structured as before (Root: 50, Left Subtree: 30(20 and 40), Right Subtree: 70(60 and 80)), perform the following deletions successively: (a) Delete 20 (b) Delete 30 (c) Delete 50.

Solution:

(a) Delete 20 (Leaf Node - Case 1)

- Node 20 has no left or right children.
- It is the left child of 30.
- **Action:** Detach 20. The left child of 30 becomes NULL.

(b) Delete 30 (One Child - Case 2)

- In the updated tree, Node 30 has only one child: the right child 40 (its left child 20 was deleted).
- Node 30 is the left child of the root 50.
- **Action:** Bypass 30. Connect 30's parent (50) directly to 30's child (40). Now, 40 is the left child of 50.

(c) Delete 50 (Two Children - Case 3)

- Node 50 is the root and has two children: 40 (left) and 70 (right).
- **Action:** Find the Inorder Successor of 50. This is the minimum value in the right subtree (rooted at 70). The minimum value in {60 70 80} is 60.
- Replace 50's value with 60. The root is now 60.
- Now, recursively delete the original node 60 from the right subtree. Node 60 was a leaf node (Case 1), so detach it from its parent 70.

Final Tree Structure: Root is 60. Left child is 40. Right child is 70. Node 70 has a right child 80.

5.4 Constructing a Tree from Given Traversals

A single traversal sequence (like Preorder or Postorder) is not enough to uniquely determine a binary tree. However, a tree can be uniquely constructed if the Inorder traversal is given along with either the Preorder or the Postorder traversal.

Methodology:

Construct Tree from Inorder and Preorder To construct a binary tree given its Inorder and Preorder traversal sequences:

1. **Identify the Root:** The first element in the Preorder sequence is always the root of the current tree (or subtree).
2. **Partition Inorder Sequence:** Locate the root element in the Inorder sequence.
   - All elements to the left of the root in the Inorder sequence belong to the **Left Subtree**.
   - All elements to the right of the root in the Inorder sequence belong to the **Right Subtree**.
3. **Recursive Construction:**
   - Count the number of nodes in the left subtree (say k).
   - The next k elements in the Preorder sequence will form the Left Subtree.
   - The remaining elements in the Preorder sequence will form the Right Subtree.
   - Recursively apply these steps for both subtrees.

Worked Example:

Construct Tree from Traversals Construct a binary tree from the following traversals:

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

Solution:
Step 1: Find the Root

- The first node in Preorder is **A**. So, **A** is the root of the tree.

Step 2: Partition the Inorder Sequence

- Search for **A** in the Inorder sequence: {D B E} **A** {F C}.
- Left subtree nodes: {D B E} (3 nodes)
- Right subtree nodes: {F C} (2 nodes)

Step 3: Construct Left Subtree {D B E}

- The next 3 nodes in Preorder are: B D E.
- First node is **B**, so **B** is the root of the left subtree.
- Find **B** in Inorder: {D} **B** {E}.
- Left child of B is **D**. Right child of B is **E**.

Step 4: Construct Right Subtree {F C}

- The remaining nodes in Preorder are: C F.
- First node is **C**, so **C** is the root of the right subtree.

- Find **C** in Inorder: {F} **C** {}.
- Left child of C is **F**. Right child of C is empty.

Final Tree Structure:

- Root: A
- Left child of A: B (which has left child D, right child E)
- Right child of A: C (which has left child F, no right child)

5.5 Expression Trees

An Expression Tree is a special type of binary tree used to represent arithmetic expressions.

- **Leaf nodes** represent operands (e.g. constants or variables like $A, B, 3, x$).
- **Internal nodes** represent operators (e.g. $+, -, *, /$).

Methodology:

Evaluation and Traversal of Expression Trees

1. **Infix Expression:** Performing an **Inorder traversal** on an expression tree yields the infix expression. Parentheses must be added for subtrees to maintain the correct order of operations.
2. **Prefix Expression:** Performing a **Preorder traversal** yields the prefix (Polish) notation.
3. **Postfix Expression:** Performing a **Postorder traversal** yields the postfix (Reverse Polish) notation.

Worked Example:

Construct and Traverse an Expression Tree Construct an expression tree for the postfix expression: $A\ B\ +\ C\ D\ -\ *$ and find its infix notation.
Solution:
Step 1: Constructing the tree from postfix Read the postfix expression from left to right.

- **A**: Operand. Create a single-node tree.
- **B**: Operand. Create a single-node tree.
- **+**: Operator. Create a new tree with '+' at the root. Pop the last two trees (B and A). B becomes the right child, A becomes the left child. (Subtree 1: A + B)
- **C**: Operand. Create a single-node tree.
- **D**: Operand. Create a single-node tree.
- **-**: Operator. Create a new tree with '-' at the root. Pop D and C. D becomes the right child, C becomes the left child. (Subtree 2: C - D)
- ***: Operator. Create a root node '*'. Pop Subtree 2 and Subtree 1. Subtree 2 becomes the right child, Subtree 1 becomes the left child.

Step 2: Finding the Infix Notation Perform an Inorder traversal (Left, Node, Right) and enclose subexpressions in parentheses:

- Left subtree (Inorder): $(A + B)$
- Root: $*$

- Right subtree (Inorder): $(C - D)$

Infix Expression: $(A + B) * (C - D)$

Appendix: Key Formulae

This appendix provides a quick reference to the key formulae and mathematical relationships discussed in this book. The equations have been categorized by unit for ease of reference.

Unit 2: Queues and Stacks

Linear Queues

In a standard linear queue, elements are inserted at the rear and removed from the front.

- **Initial Condition (Empty Queue):**

$$\text{FRONT} = -1, \quad \text{REAR} = -1$$

- **Enqueue Operation:** To insert an element X, the `REAR` pointer is incremented.

$$\begin{aligned} \text{REAR} &= \text{REAR} + 1 \\ Q[\text{REAR}] &= X \end{aligned}$$

Circular Queues

In a circular queue of size N, the pointers wrap around upon reaching the end, utilizing modulo arithmetic.

- **Enqueue Operation:** To insert an element X, the `REAR` pointer is updated using the modulo operator.

$$\begin{aligned} \text{REAR} &= (\text{REAR} + 1) \pmod{N} \\ CQ[\text{REAR}] &= X \end{aligned}$$

Stacks & Expression Evaluation

During the evaluation of a postfix expression, operands are popped from the stack and an operation is applied.

$$R = B \ \mathcal{O} \ A$$

where A is the first popped operand (top of stack), B is the second popped operand, $\mathcal{O}$ is the operator, and R is the evaluated result to be pushed back onto the stack.

Unit 4: Searching and Sorting

Quick Sort

Quick Sort is a divide-and-conquer algorithm that operates by selecting a pivot and partitioning the array around it.

- **Pivot Selection:** A common approach is to pick the last element of the current subarray (bounded by `low` and `high`) as the pivot.

$$\text{pivot} = A[\text{high}]$$

- **Partitioning Pointer Initialization:** The pointer i, which tracks the boundary of elements smaller than the pivot, is initialized to one position before the `low` index.

$$i = \text{low} - 1$$

Merge Sort

Merge Sort divides the array into halves, sorts them, and merges them back. During the merge step, elements from temporary subarrays (such as the left subarray L or the right subarray R) are copied back into the original array A.

$$A[k] = R[j] \quad (\text{or } A[k] = L[i])$$

where k is the current index in the main array A, and j (or i) is the current index in the temporary subarray.