Q1 **A. Completeness** → **iii**: Is the strategy guaranteed to find the solution when there is one?

- **B. Time Complexity** → **i**: How long does it take to find a solution?
- **C. Space Complexity** → **ii**: How much memory is needed to perform the search?
- **D. Efficient Complexity** → **v**: O(n2).

Q2 Explain linear and non-linear data structures with examples5M

Linear and Non-Linear Data Structures

1. Linear Data Structures:

- Data elements are arranged sequentially or in a linear order.
- Each element is connected to its previous and next element (except for the first and last elements).

Examples:

- **Array:** A collection of elements stored at contiguous memory locations. Example: `[10, 20, 30, 40]`.
- **Linked List:** A series of nodes where each node points to the next node in the sequence. Example: `10 -> 20 -> 30 -> NULL`.
- **Stack:** Follows the **Last In, First Out (LIFO)** principle. Example: A stack of books.
- **Queue:** Follows the **First In, First Out (FIFO)** principle. Example: A line at a ticket counter.

2. Non-Linear Data Structures:

- Data elements are not arranged sequentially. They are connected hierarchically or in a graph-like structure.
- More complex relationships exist between elements.

Examples:

- **Tree:** A hierarchical structure where each node has a parent and may have child nodes. Example: A binary tree.
- **Graph:** A collection of nodes (vertices) connected by edges. Example: A social network graph.
- **Heap:** A special tree-based structure where the parent node is either greater (max heap) or smaller (min heap) than its children.

Key Difference:

- **Linear Structures** are simpler and suitable for scenarios where data is processed in order (e.g., arrays, queues).
- **Non-Linear Structures** are better for representing complex relationships (e.g., family trees, networks).

Q3 **Practical Applications of Linked Lists 5M**

1. **Dynamic Memory Allocation:**
    - Linked lists are used in applications requiring dynamic memory allocation, as they allow flexible resizing without the need for contiguous memory.

    - Example: Implementing memory management in operating systems.

2. **Implementation of Stacks and Queues:**
    - Linked lists are used to implement stacks and queues efficiently, where insertion and deletion operations can occur at either end in O(1) time.
3. **Graph Representation:**
    - Linked lists are used to represent graphs as adjacency lists. Each node points to a list of connected vertices, making it memory-efficient for sparse graphs.
4. **Handling Polynomials:**
    - Linked lists can be used to represent and manipulate polynomials where each node contains a term with its coefficient and exponent.
    - Example: 3x2+4x+5 can be stored as a linked list.
5. **Undo/Redo Functionality in Applications:**
    - Applications like text editors use linked lists to implement undo and redo operations by maintaining a history of operations as nodes.

These applications highlight the flexibility and efficiency of linked lists in scenarios requiring dynamic and efficient memory management.

Q4 **d. Calculate the time complexity of the code 5M**

```
c
Copy code
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + rand();
}
for (j = 0; j < M; j++) {
    b = b + rand();
}
```

ANS Code Analysis:

First `for` Loop (i = 0 to N):

- The loop executes **N** times.
- Inside the loop, the statement `a = a + rand();` is a constant-time operation, i.e., O(1).
- Therefore, the time complexity of this loop is O(N).

2. **Second `for` Loop (j = 0 to M):**
    - The loop executes **M** times.
    - Inside the loop, the statement `b = b + rand();` is also a constant-time operation, i.e., O(1).
    - Therefore, the time complexity of this loop is O(M).
3. **Overall Time Complexity:**
    - The two loops are independent (i.e., they do not nest inside each other), so their time complexities are **added**, not multiplied.
    - Total time complexity = O(N)+O(M).

Final Answer:

- The **time complexity** of the code is O(N+M).

Q5 What is stack data structure and what are its applications? 5M

Stack Data Structure

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle. This means that the last element added to the stack is the first one to be removed. It can be implemented using arrays or linked lists.

- **Basic Operations in Stack:**
  - **Push:** Adds an element to the top of the stack.
  - **Pop:** Removes and returns the top element of the stack.
  - **Peek/Top:** Retrieves the top element without removing it.
  - **IsEmpty:** Checks if the stack is empty.

Applications of Stack

1. **Expression Evaluation and Conversion:**
   - Stacks are used to evaluate postfix (Reverse Polish Notation) expressions and convert infix expressions to postfix or prefix.
2. **Function Call Management:**
   - The system stack is used to manage function calls, recursive function execution, and storing local variables.
3. **Undo/Redo Functionality:**
   - Stacks are used to implement undo and redo features in applications like text editors.
4. **Backtracking:**
   - Used in algorithms like maze solving, depth-first search (DFS) in graphs, and parsing.
5. **Browser History:**
   - Browsers use stacks to store the history of visited pages, allowing navigation between previous and next pages.

Stacks are versatile and efficient for scenarios requiring a LIFO structure, making them essential in both software development and operating systems.

Q6 Types of Trees and Applications (10 Marks)

Types of Trees and Their Applications

1. Binary Tree:

- **Definition:** A tree where each node has at most two children (left and right).
- **Applications:**
    - Used in hierarchical data storage.
    - Parsing expressions in compilers.
    - Representation of hierarchical relationships like file directories.

2. Binary Search Tree (BST):

- **Definition:** A binary tree where the left child of a node contains smaller values, and the right child contains larger values.
- **Applications:**
    - Efficient searching, insertion, and deletion (O(logn) on average).
    - Database indexing.
    - Implementation of sets and maps.

3. AVL Tree:

- **Definition:** A self-balancing binary search tree where the height difference (balance factor) between left and right subtrees is at most 1.
- **Applications:**
    - Maintaining sorted data for efficient searches.
    - Used in database systems for faster retrievals.

4. B-Tree:

- **Definition:** A self-balancing search tree where nodes can have more than two children. Commonly used in external storage systems.
- **Applications:**
    - File systems and database indexing.
    - Storing large amounts of sorted data on disk.

    - **5. Heap:**

- **Definition:** A complete binary tree satisfying the heap property (max-heap or min-heap).
- **Applications:**
    - Implementing priority queues.
    - Heap sort algorithm.
    - Efficiently finding the smallest or largest element.

6. Trie (Prefix Tree):

- **Definition:** A tree used to store strings, where each edge represents a character.
- **Applications:**
    - Implementing dictionaries and autocomplete features.
    - Searching for words in a dataset efficiently.
    - IP routing.

7. Red-Black Tree:

- **Definition:** A self-balancing binary search tree where nodes are either red or black, ensuring balanced height.
- **Applications:**
  - Used in the implementation of associative containers in libraries (e.g., `std::map` in C++).
  - Maintaining balanced trees for faster search, insertion, and deletion.

8. Segment Tree:

- **Definition:** A tree used for range queries and updates on an array.
- **Applications:**
  - Range sum or range minimum queries in competitive programming.
  - Efficiently solving interval-related problems.

9. N-ary Tree:

- **Definition:** A tree where each node can have up to N children.
- **Applications:**
  - Representation of hierarchical structures like file systems.
  - Storing game trees (e.g., chess).

10. General Tree:

Definition: A tree with no restrictions on the number of children a node can have.

- **Applications:**
  - Representing organizational structures.
  - Modeling XML/HTML document object models (DOM).

Summary:

Different types of trees are suited for specific applications such as searching, sorting, data storage, and hierarchical modeling. Each type optimizes operations like searching, insertion, or retrieval depending on the use case.

Q7 **b. Linked List Representation of Polynomials (10 Marks)**

Polynomials:

- P1=5X2−4X+2
- P2=−5X−5

ANS Steps:

1. **Create a Node Structure:**
    - Each node contains:
        - Coefficient.
        - Exponent.
        - Pointer to the next node.
2. **Linked List for P1:**
    - First Node: (5,2) → 5X2
    - Second Node: (−4,1) → −4X
    - Third Node: (2,0) → +2
3. **Linked List for P2:**
    - First Node: (−5,1) → −5X
    - Second Node: (−5,0) → −5
4. **Addition of Polynomials (P1 + P2):**
    - Combine nodes with the same exponent.
    - Resultant Polynomial P3: 5X2−9X−3

Sketch (Simplified Representation):

- P1:
  (5,2)→(−4,1)→(2,0)→NULL
- P2:
  (−5,1)→(−5,0)→NULL
- P3:
  (5,2)→(−9,1)→(−3,0)→NULL

Result:

The new polynomial P3=5X2−9X−3.

Q8 Circular Queue and Doubly Ended Queue (10 Marks

1. Circular Queue:

- **Definition:** A linear data structure where the last position is connected back to the first, forming a circle.
- **Properties:**
    - Efficiently utilizes space by reusing empty slots at the front.
    - Operations: Enqueue (add), Dequeue (remove).

- **Applications:** CPU scheduling, memory management.
- **Example:** Initial state: [_, _, _, _]
  After enqueue(1, 2, 3): [1, 2, 3, *]*
  After dequeue(): [, 2, 3, _]
  Enqueue(4): [4, 2, 3, _].

2. Doubly Ended Queue (Deque):

- **Definition:** A linear data structure where elements can be added or removed from both ends.
- **Types:**
  - **Input-restricted:** Add only at one end, remove at both.
  - **Output-restricted:** Remove only at one end, add at both.
- **Applications:** Undo operations, implementing stacks and queues.
- **Example:**
  AddFront(1): [1]
  AddRear(2): [1, 2]
  RemoveFront(): [2].

Q9 Binary Tree Construction (10 Marks)

In-order Traversal: D B E F A G H C

Pre-order Traversal: A B D E F C G H

Steps to Construct the Tree:

1. The **root** is the first element of the Pre-order: A.
2. Find A in the In-order sequence: DBEF | A | GHC.
   - Left subtree: DBEF.
   - Right subtree: GHC.
3. Repeat the process recursively for left and right subtrees:
   - **Left Subtree of A:**
     Pre-order: BDEF, In-order: DBEF.
     Root: B, Left: D, Right: EF.
     Root of EF is E (pre-order).
   - **Right Subtree of A:**
     Pre-order: CGH, In-order: GHC.
     Root: C, Left: GH, Right: ∅.
4. **Post-order Traversal:** D F E B H G C A.

Q10 Five Operations Performed on a Binary Search Tree (BST) 10M

A Binary Search Tree (BST) is a hierarchical data structure that ensures elements are stored in a sorted manner, where the left subtree contains nodes with smaller values, and the right subtree contains nodes with larger values.

---

1. Insertion

- **Operation:**
  - Insert a new element into the BST while maintaining the BST property.
  - Start at the root, and compare the value to be inserted with the current node.
    - If smaller, move to the left subtree.
    - If larger, move to the right subtree.
  - Insert the new node at the correct position when a `NULL` child is encountered.
- **Time Complexity:** O(h), where h is the height of the tree.
- **Example:** Inserting 15 into a BST with values 10,20,5.

2. Deletion

- **Operation:**
  - Remove an element from the BST while maintaining the BST property.
  - Three cases for deletion:
    1. **Node with no children:** Simply delete the node.
    2. **Node with one child:** Replace the node with its child.
    3. **Node with two children:** Replace the node with its **in-order successor** or **in-order predecessor**, then delete the successor or predecessor.
- **Time Complexity:** O(h).
- **Example:** Deleting 20 from a BST with values 10,20,15,5.

3. Searching

- **Operation:**
  - Find whether a given value exists in the BST.
  - Start at the root and compare the key with the current node:
    - If the key is smaller, search the left subtree.
    - If the key is larger, search the right subtree.
    - If the key matches, the element is found.
- **Time Complexity:** O(h).
- **Example:** Searching for 15 in a BST with values 10,20,5,15.

4. Traversal

- **Operation:**
  - Visit all nodes in the BST in a specific order. Common traversal methods include:
    - **In-order Traversal (Left → Root → Right):** Visits nodes in ascending order.
    - **Pre-order Traversal (Root → Left → Right):** Used for creating a copy of the tree.
    - **Post-order Traversal (Left → Right → Root):** Used for deleting or processing subtrees.
- **Time Complexity:** O(n), where n is the number of nodes.
- **Example:** In-order traversal of 10,5,15 results in 5,10,15.

5. Finding Minimum and Maximum

- **Operation:**
    - **Minimum:** Traverse the leftmost path of the tree starting from the root. The last node on this path is the minimum value.
    - **Maximum:** Traverse the rightmost path of the tree starting from the root. The last node on this path is the maximum value.
- **Time Complexity:** O(h).
- **Example:** For a BST with values 10,5,20,15, minimum is 5, and maximum is 20.

Summary Table

Operation	Description	Time Complexity
Insertion	Add a node while maintaining BST properties	O(h)
Deletion	Remove a node while maintaining BST rules	O(h)
Searching	Check if a value exists in the BST	O(h)
Traversal	Visit nodes in a specific order	O(n)
Find Min/Max	Locate the smallest or largest value	O(h)

The height h of the BST significantly impacts performance, so balanced BSTs (e.g., AVL or Red-Black trees) are used to optimize these operations.

Q11 SEARCHING TECHNIQUES 10M

Searching Techniques

Searching is the process of finding a specific element in a dataset (like an array or list). There are two primary categories of searching techniques:

---

1. Linear Search

- **Description:** A straightforward method where each element is checked one by one until the desired element is found.
- **Time Complexity:** O(n) in the worst case, where n is the number of elements.
- **Use Case:** Useful for unsorted datasets.
- **Example:** Finding the number 7 in the array [3,5,7,9,11].

2. Binary Search

- **Description:** A highly efficient search technique that works only on sorted datasets. It repeatedly divides the search interval in half.
- **Steps:**
    1. Compare the middle element of the dataset with the target value.
    2. If the middle element is equal to the target, the search is complete.
    3. If the target is smaller, search the left half; otherwise, search the right half.
    4. Repeat until the target is found or the interval becomes empty.
- **Time Complexity:** O(logn), where n is the number of elements.
- **Use Case:** Searching in sorted arrays or lists.

Binary Search Algorithm (Example)

Algorithm:

```
int binarySearch(int arr[], int n, int target) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target) {
            return mid;  // Target found
        } else if (arr[mid] < target) {
            low = mid + 1;  // Search in the right half
        } else {
            high = mid - 1; // Search in the left half
        }
    }
    return -1;  // Target not found
}
```

Example Walkthrough:

- Dataset: [2,4,6,8,10,12]
- Target: 8

1. **Step 1:** `low = 0`, `high = 5`, `mid = 2`. Middle element is 6. Since 8>6, search in the right half.
2. **Step 2:** `low = 3`, `high = 5`, `mid = 4`. Middle element is 10. Since 8<10, search in the left half.
3. **Step 3:** `low = 3`, `high = 3`, `mid = 3`. Middle element is 8. Target found at index 3.

Other Advanced Searching Techniques

1. **Hashing:**
   - Uses hash functions to index data for fast access.
   - Time Complexity: O(1) for search in ideal conditions.
2. **Depth-First Search (DFS) and Breadth-First Search (BFS):**
   - Used for searching in trees or graphs.
   - Time Complexity: O(V+E), where V is vertices, E is edges.
3. **Interpolation Search:**
   - Used for uniformly distributed sorted datasets.
   - Time Complexity: O(loglogn) in the best case.

Binary search is one of the most efficient techniques for sorted data and is widely used in applications like searching in databases and performing range queries.

Q12 **Application of Huffman Coding with an Example (10 Marks)**

Huffman Coding:

Huffman coding is a lossless data compression algorithm used to reduce the size of data by assigning variable-length binary codes to characters. Characters that occur more frequently are assigned shorter codes, while less frequent characters are given longer codes.

Applications of Huffman Coding:

1. **File Compression:**
   - Commonly used in ZIP, RAR, and GZIP file compression formats.
2. **Multimedia Compression:**
   - Used in image (JPEG) and video (MPEG) compression.
3. **Data Transmission:**
   - Efficient encoding for data communication to minimize transmission bandwidth.
4. **Text Encoding:**
   - Reducing memory usage for large text files like dictionaries.

Steps in Huffman Coding Algorithm:

1. **Frequency Count:**
   - Calculate the frequency of each character in the input data.
2. **Build a Min-Heap:**
   - Create a priority queue (min-heap) of nodes, where each node represents a character and its frequency.
3. **Tree Construction:**
   - Build a binary tree by repeatedly extracting the two nodes with the smallest frequencies, combining them, and inserting the new node back into the heap.
4. **Generate Codes:**
   - Assign binary codes (0 for left, 1 for right) to each character by traversing the tree.

Example:

Input: `"ABRACADABRA"`

Character	Frequency
A	5
B	2
R	2
C	1
D	1

1. **Construct Min-Heap:**

        - Nodes: A (5), B (2), R (2), C (1), D (1).
2. **Build Huffman Tree:**
    - Combine C (1) and D (1) → Node (2).
    - Combine Node (2) and R (2) → Node (4).
    - Combine B (2) and Node (4) → Node (6).
    - Combine A (5) and Node (6) → Root (11).
3. **Assign Codes:**
    - A: `0`
    - B: `101`
    - R: `100`
    - C: `1100`
    - D: `1101`
4. **Result:**
    - Original data: $11 \times 8 = 88$ bits (ASCII, 8 bits per character).
    - Encoded data: $2 \times 5 + 3 \times 2 + 3 \times 2 + 4 \times 1 + 4 \times 1 = 39$ bits.

Huffman coding reduces the size of data efficiently.

---

b. Short Note on Sorting Algorithms (Any One)

Bubble Sort Algorithm (10 Marks)

Definition:

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 process is repeated until the list is sorted. The largest elements "bubble up" to their correct positions after each pass.

Steps in Bubble Sort:

1. Start with the first element in the list.
2. Compare the current element with the next element.
3. If the current element is greater than the next, swap them.
4. Repeat this process for all adjacent elements in the list.
5. After completing one pass, the largest element is placed in its correct position at the end of the list.
6. Repeat the process for the remaining unsorted part of the list until the entire list is sorted.

Algorithm:

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
```

```
            // Swap arr[j] and arr[j + 1]
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
          }
        }
    }
}
```

Example:

Sort the array: [5,1,4,2,8]

1. **Pass 1:**
    - Compare 5 and 1: Swap → [1,5,4,2,8]
    - Compare 5 and 4: Swap → [1,4,5,2,8]
    - Compare 5 and 2: Swap → [1,4,2,5,8]
    - Compare 5 and 8: No Swap → [1,4,2,5,8]
2. **Pass 2:**
    - Compare 1 and 4: No Swap → [1,4,2,5,8]
    - Compare 4 and 2: Swap → [1,2,4,5,8]
    - Compare 4 and 5: No Swap → [1,2,4,5,8]
3. **Pass 3:**
    - Compare 1 and 2: No Swap → [1,2,4,5,8]
    - Compare 2 and 4: No Swap → [1,2,4,5,8]
4. **Result:** Sorted array: [1,2,4,5,8]

Time Complexity:

- **Best Case (Already Sorted):** O(n)
- **Average Case:** O(n2)
- **Worst Case (Reverse Sorted):** O(n2)

Space Complexity:

- **Auxiliary Space:** O(1), as it is an in-place sorting algorithm.

Applications:

1. **Small datasets** where simplicity is preferred over efficiency.
2. **Educational purposes** to teach basic sorting concepts.
3. Sorting nearly sorted data (optimized bubble sort can work efficiently in such cases).

Bubble Sort is easy to implement but is not suitable for large datasets due to its inefficiency compared to other algorithms like Quick Sort or Merge Sort.

Quick Sort Algorithm (10 Marks)

Definition:

Quick Sort is a **divide-and-conquer sorting algorithm** that works by selecting a **pivot element**, partitioning the array into two halves (elements smaller than the pivot and elements larger than the pivot), and recursively sorting the two subarrays.

Steps in Quick Sort:

1. **Choose a Pivot**:
   - Select any element as the pivot (commonly the first, last, or middle element).
2. **Partitioning**:
   - Rearrange the array so that elements smaller than the pivot are on the left, and elements larger than the pivot are on the right.
   - The pivot element is then placed in its correct position in the sorted array.
3. **Recursive Sorting**:
   - Recursively apply the same process to the subarrays on the left and right of the pivot.

Algorithm:

```
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partition the array
        int pi = partition(arr, low, high);

        // Recursively sort the subarrays
        quickSort(arr, low, pi - 1); // Left subarray
        quickSort(arr, pi + 1, high); // Right subarray
    }
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choosing the last element as pivot
    int i = (low - 1); // Index of smaller element
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            // Swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    // Swap arr[i + 1] and pivot
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return (i + 1);
}
```

Example:

Sort the array: [10,7,8,9,1,5]

1. **Initial Array**: [10,7,8,9,1,5]
    - Pivot = 5
    - Partition: [1,5,8,9,10,7]
    - Pivot 5 is placed in its correct position at index 1.
2. **Left Subarray**: [1] (Already sorted)
3. **Right Subarray**: [8,9,10,7]
    - Pivot = 7
    - Partition: [7,8,9,10]
    - Pivot 7 is placed in its correct position at index 2.
4. **Repeat the process** until all subarrays are sorted.

Final Sorted Array: [1,5,7,8,9,10]

Time Complexity:

- **Best Case (Balanced Partitions):** O(nlogn)
- **Average Case:** O(nlogn)
- **Worst Case (Unbalanced Partitions):** O(n2)

Worst-case occurs when the pivot is the smallest or largest element repeatedly, such as in already sorted or reverse-sorted arrays.

Space Complexity:

- **Auxiliary Space:** O(logn) for recursion stack in the best/average case.
- O(n) in the worst case (deep recursion).

Applications:

1. **Efficient Sorting**:
    - Quick Sort is widely used in sorting large datasets.
2. **System Libraries**:
    - Used in standard library implementations such as Python (`sorted()` function) and C++ (`std::sort`).
3. **Databases**:
    - Sorting records for efficient searching and retrieval.
4. **Embedded Systems**:
    - Useful in memory-limited environments due to its in-place sorting nature.

Advantages:

- Faster than other O(n2) algorithms like Bubble Sort or Insertion Sort.
- Performs well on average and works efficiently on large datasets.

Disadvantages:

- Poor performance in the worst case (if pivot selection is poor).
- Not stable (order of equal elements may change).

Merge Sort Algorithm (10 Marks)

Definition:

Merge Sort is a **divide-and-conquer sorting algorithm** that splits an array into two halves, recursively sorts each half, and then merges the two sorted halves into a single sorted array.

Steps in Merge Sort:

1. **Divide**:
    - Divide the array into two halves until each subarray contains a single element (or is empty).
2. **Conquer (Recursively Sort)**:
    - Sort each of the two halves recursively using Merge Sort.
3. **Combine (Merge)**:
    - Merge the two sorted halves into one sorted array.

Algorithm: void mergeSort(int arr[], int left, int right) {

```
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Recursively sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays
    int L[n1], R[n2];

    // Copy data to temporary arrays
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    // Merge the temporary arrays back into arr
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Example:

Sort [38,27,43,3,9,82,10]:

- Divide into halves until single elements: [38], [27], etc.
- Merge into sorted pairs: [27,38], [3,9], etc.
- Final sorted array: [3,9,10,27,38,43,82].

Complexity:

- **Time**: O(nlogn) (Best, Average, Worst Case).
- **Space**: O(n) (for temporary arrays).

Applications:

1. Sorting large datasets.

2. External sorting (e.g., disk-based data).
3. Computational geometry (e.g., inversion count).

Advantages:

- Stable and consistently O(nlogn).
- Works well for large datasets.

Disadvantages:

- Requires extra memory (O(n)).

Q13 **Use of Hashing, Collisions, and Hash Table Construction (10 Marks)**

Use of Hashing:

Hashing is a technique used for efficient data retrieval and storage. It maps data to a specific index in a hash table using a hash function.

Collision:

A collision occurs when two different keys are mapped to the same index in the hash table.

Hash Function:

The **modulo division method** calculates the index as:

SizeIndex=KeymodTable Size

Dataset:

12,45,67,88,27,78,20,62,36,55

Hash Table Size: 10

Hash Table Entries:

Using Index=Keymod10:

Key	Index (Keymod10)	Collision
12	12mod10=2	No
45	45mod10=5	No
67	67mod10=7	No
88	88mod10=8	No
27	27mod10=7	Yes
78	78mod10=8	Yes
20	20mod10=0	No
62	62mod10=2	Yes
36	36mod10=6	No
55	55mod10=5	Yes

Final Hash Table (Handling Collisions):

Using **chaining** (storing multiple values at one index):

Index	Keys
0	20
1	-
2	12, 62
3	-
4	-
5	45, 55
6	36
7	67, 27
8	88, 78
9	-

Q15 **Tree Traversal Algorithm (10 Marks)**

Tree traversal is a process of visiting each node in a tree data structure exactly once in a systematic way. It is essential for performing operations like searching, updating, or printing the elements of the tree.

Types of Tree Traversal

Tree traversal methods are classified into two categories:

1. **Depth-First Search (DFS):**
   - **Inorder Traversal (Left → Root → Right):**
     - Traverses the left subtree first, then visits the root node, and finally traverses the right subtree.
     - Commonly used for **binary search trees (BST)** to retrieve elements in sorted order.
     - Example:
       For the tree:

       ```
       mathematica
       Copy code
           A
          / \
         B   C
        / \
       D   E
       ```

       Inorder: D, B, E, A, C
   - **Preorder Traversal (Root → Left → Right):**
     - Visits the root node first, then traverses the left subtree, followed by the right subtree.
     - Used to create a **copy** of the tree.
     - Example: A, B, D, E, C

        - **Postorder Traversal (Left → Right → Root):**
            - Traverses the left subtree first, then the right subtree, and finally the root node.
            - Useful for **deleting a tree** or evaluating postfix expressions.
            - Example: D, E, B, C, A
2. **Breadth-First Search (BFS):**
    - Also called **Level-Order Traversal**.
    - Visits all nodes level by level from left to right.
    - Example: For the same tree: A, B, C, D, E

Applications of Tree Traversal

1. **Inorder Traversal:**
    - Used in **binary search trees (BST)** for retrieving sorted data.
2. **Preorder Traversal:**
    - Helpful in creating tree **replications** and expressions (prefix notation).
3. **Postorder Traversal:**
    - Used in tree **deletion**, evaluating postfix expressions, and generating a **polish notation**.
4. **Level Order Traversal:**
    - Ideal for **shortest path** problems or printing the tree layer by layer.

Example Tree Traversals with Diagram

Consider the binary tree:

```
mathematica
Copy code
        F
      /   \
     B     G
    / \     \
   A   D     I
      / \   /
     C   E  H
```

- **Inorder Traversal:** A, B, C, D, E, F, G, H, I
- **Preorder Traversal:** F, B, A, D, C, E, G, I, H
- **Postorder Traversal:** A, C, E, D, B, H, I, G, F
- **Level Order Traversal:** F, B, G, A, D, I, C, E, H

Tree Diagram

```
mathematica
Copy code
        F
      /   \
     B     G
    / \     \
   A   D     I
      / \   /
     C   E  H
```

Conclusion

Tree traversal is a fundamental operation in tree-based data structures and plays a crucial role in various computer science applications such as expression evaluation, data retrieval, and hierarchical representation.

Q16 **Graph Traversal Algorithms (Short Note)**

Graph traversal algorithms are used to visit all the vertices and edges of a graph systematically. These techniques help solve problems like pathfinding, network analysis, and connectivity checks. The two primary graph traversal methods are **Breadth-First Search (BFS)** and **Depth-First Search (DFS).**

---

1. Breadth-First Search (BFS):

- **Definition:** BFS explores a graph level by level. It starts at a source vertex, visits all its neighbors, then proceeds to the next level of neighbors.
- **Approach:**
  - Use a **queue** to keep track of vertices to visit.
  - Mark visited vertices to avoid repetition.
- **Steps:**
  - Start at the source vertex and mark it as visited.
  - Enqueue all its unvisited neighbors.
  - Dequeue a vertex, visit it, and enqueue its neighbors.
  - Repeat until the queue is empty.
- **Applications:**
  - Shortest path in an unweighted graph.
  - Solving puzzles like mazes.
- **Example:** For the graph:

```
mathematica
Copy code
      A
     / \
    B   C
   / \
  D   E
```

  BFS (starting from A): **A** → **B** → **C** → **D** → **E**

---

2. Depth-First Search (DFS):

- **Definition:** DFS explores as far as possible along each branch before backtracking. It follows one path to its deepest point and then tries alternate paths.
- **Approach:**
  - Use a **stack** (or recursion) to keep track of the path.
  - Mark visited vertices to avoid repetition.
- **Steps:**
  - Start at the source vertex and mark it as visited.

    - Push it onto the stack (or use recursion).
    - Visit an unvisited neighbor, mark it as visited, and push it onto the stack.
    - Backtrack when no unvisited neighbors remain.
    - Repeat until all vertices are visited.
- **Applications:**
    - Topological sorting.
    - Detecting cycles in graphs.
    - Solving connectivity problems.
- **Example:** For the same graph:

```
mathematica
Copy code
      A
     / \
    B   C
   / \
  D   E
```

  DFS (starting from A): **A** → **B** → **D** → **E** → **C**