1. Design and Develop Java Program to implement Polynomial addition using Arrays.
Aim:- To design and develop Java Program to implement Polynomial addition using Arrays.

Description:

Define the Polynomial Representation: We'll use arrays where the index represents the power of the polynomial term, and the value at that index represents the coefficient.
Add Polynomials: Implement the logic to add two polynomials.
Handle Edge Cases: Ensure the program can handle polynomials of different degrees.

Algorithm:

Input Collection

- **Step 1**: Read the degree of the first polynomial.
- **Step 2**: Read the coefficients for the first polynomial and store them in `poly1`.
- **Step 3**: Read the degree of the second polynomial.
- **Step 4**: Read the coefficients for the second polynomial and store them in `poly2`.

Addition Logic

- **Step 5**: Compute the length of the result polynomial array, which is the maximum of the lengths of `poly1` and `poly2`.
- **Step 6**: Initialize the `result` array with zeros.
- **Step 7**: For each index in the `result` array:

Add the coefficients from `poly1` and `poly2` if they exist, otherwise use zero for **Output Results**

- **Step 8**: Format and print `poly1`.
- **Step 9**: Format and print `poly2`.
- **Step 10**: Format and print the `result` polynomial.

2. Design and Develop menu driven Java Program to implement Stack and Queue operations using Single Linked List

Aim: To design and develop menu driven Java Program to implement Stack and Queue operations using Single Linked List.

Description:-

Node Class: Represents an individual node in the linked list, holding data and a reference to the next node.

Stack Class: Implements a stack using a linked list where `top` is the top of the stack. Supports `push`, `pop`, `peek`, and `print Stack` operations.

Queue Class: Implements a queue using a linked list where `front` and `rear` represent the front and rear of the queue. Supports `enqueue`, `dequeue`, `peek`, and `printQueue` operations.

Main Class: Contains the `main` method to test stack and queue operations.

Algorithm:-

1. Stack Operations

1.1 Push Operation

Algorithm:

Step 1. Create a New Node:

1.1 Create a new node with the given data.

Step 2. Link the New Node:

2.1 Set the `next` pointer of the new node to the current top node.

Step 3. Update the Top:

3.1 Update the top pointer to the new node.

Step 4. Print Confirmation:

4.1 Output a message indicating that the data has been pushed onto the stack.

Step 5. Input: Data to push (e.g., 10).

Step 6. Create a New Node

6.1 newNode = new Node(data)`

Step 7 Set `next` Pointer:

7.1 newNode.next = top`

Step 8. Update Top Pointer:

8.1 `top = newNode`

Step 9. Print Message:

9.1 `System.out.println("Pushed " + data + " onto stack.");`

1.2 Pop Operation

Algorithm:

Step 1. Check if Empty:

1.1 If the stack is empty, raise an exception or error.

Step 2. Remove Top Node:

2.1 Retrieve the data from the top node.

2.2 Update the top pointer to the next node in the stack.

Step 3. Print Confirmation:

3.1 Output a message indicating that the data has been popped from the stack.

Step 4. Return Data:

4.1 Return the retrieved data.

Step 5. Check Empty:

5.1 `if (isEmpty()) throw new RuntimeException("Stack is empty. Cannot pop.");`

Step 6. Retrieve Data:

6.1 `data = top.data`

Step 7. Update Top Pointer:

7.1 `top = top.next`

Step 8. Print Message:

8.1 `System.out.println("Popped " + data + " from stack.");`

Step 9 Return Data:

9.1 `return data`

1.3 Peek Operation

Algorithm:

Step 1. Check if Empty:

1.1 If the stack is empty, raise an exception or error.

Step 2. Retrieve Top Data:

2.1 Return the data from the top node without removing it.

Step3 . Check Empty:

3.1 `if (isEmpty()) throw new RuntimeException("Stack is empty. Cannot peek.");`

Step 4. Return Top Data:

4.1 `return top.data`

2. Queue Operations

2.1 Enqueue Operation

Step 1. Create a New Node:

1.1 Create a new node with the given data.

Step 2. Check if Queue is Empty:

2.1 If the queue is empty (i.e., `rear` is `null`), set both `front` and `rear` to the new node.

Step 3. Link the New Node:

3.1 If the queue is not empty, set the `next` pointer of the current rear node to the new node, and then update the rear pointer to the new node.

Step 4. Print Confirmation:

Step 5. Output a message indicating that the data has been enqueued into the queue.

Step 6 . Create a New Node:

6.1 `newNode = new Node(data)`

Step 7. Check if Empty:

7.1 `if (rear == null) { front = rear = newNode; }`

Step 8. Link New Node:

8.1 `else { rear.next = newNode; rear = newNode; }`

Step 9. Print Message:

9.1 `System.out.println("Enqueued " + data + " into queue.");`

2.2 Dequeue Operation

Algorithm:

Step 1. Check if Empty:
1.1 - If the queue is empty, raise an exception or error.
Step 2. Remove Front Node:
2.1 Retrieve the data from the front node.
2.2 Update the front pointer to the next node in the queue.
Step 3. Check if Queue is Now Empty:
3.1 If the front is `null` after removal, set the rear to `null` as well.
Step 4. Print Confirmation:
4.1 Output a message indicating that the data has been dequeued from the queue.
Step 5. Return Data:
5.1 Return the retrieved data.
Step 6. Check Empty:
6.1 `if (isEmpty()) throw new RuntimeException("Queue is empty. Cannot dequeue.");`
Step 7. Retrieve Data:
7.1 `data = front.data`
Step 8. Update Front Pointer:
8.1 `front = front.next`
Step 9. Check Empty:
9.1 `if (front == null) rear = null;`
Step 10. Print Message:
10.1 `System.out.println("Dequeued " + data + " from queue.");`
Step 11. Return Data:
11.1 `return data`

2.3 Peek Operation

Algorithm:

Step 1. Check if Empty:
1.1 If the queue is empty, raise an exception or error.
Step 2. Retrieve Front Data:
2.1 Return the data from the front node without removing it.
Step 3. Check Empty:
3.1 `if (isEmpty()) throw new RuntimeException("Queue is empty. Cannot peek.");`
Step 4. Return Front Data:
4.1 `return front.data`
Step 5. End.

3. Design and Develop Java Program to implement for conversion expressions from infix to postfix and infix to prefix

Aim: To design and develop Java Program to implement for conversion expressions from infix to postfix and infix to prefix.

Description :

1. **Precedence Function**:
   - Determines the precedence level of operators.
2. **Infix to Postfix Conversion**:
   - Uses a stack to handle operators and parentheses, ensuring proper precedence and associativity.
3. **Reverse Function**:
   - Used to reverse the string for infix-to-prefix conversion.
4. **Infix to Prefix Conversion**:
   - First reverses the infix expression and swaps parentheses, converts this modified expression to postfix, and then reverses the result to get the prefix notation.
5. **Main Method**:
   - Demonstrates usage with a sample infix expression.

This Java program efficiently converts expressions between infix, postfix, and prefix notations and provides a clear understanding of the below algorithms.

Algorithm:

1. **Infix to Postfix Conversion**:
   - **Input**: `A+B*(C-D)/E`
   - **Process**:
     Step 1: Read `A`: Output → `A`
     Step 2: Read `+`: Stack → `+`
     Step 3: Read `B`: Output → `AB`
     Step 4: Read `*`: Stack → `*` (Pop `+` from stack to output because `*` has higher precedence) → Output → `AB+`
     Step 5: Read `(`: Stack → `*(`
     Step 6: Read `C`: Output → `AB+C`
     Step 7: Read `-`: Stack → `*(-`
     Step 8: Read `D`: Output → `AB+CD`
     Step 9: Read `)`: Stack → `*` (Pop `-` from stack to output, then pop `(`) → Output → `AB+CD-`
     Step 10: Read `/`: Stack → `/` (Pop `*` from stack to output because `/` has lower precedence) → Output → `AB+CD-*`
     Step 11: Read `E`: Output → `AB+CD-*E`
   - **Output**: `ABCD-*E/+`
2. **Infix to Prefix Conversion**:

- **Input**: `A+B*(C-D)/E`
- **Reverse and Swap Parentheses**: `E/(D-C)*B+A`
- **Convert Reversed Infix to Postfix**:
  - **Reversed Input**: `E/(D-C)*B+A`
  - Process similarly to postfix conversion:
    Step 1: Read `E`: Output → `E`
    Step 2: Read `/`: Stack → `/`
    Step 3: Read `(`: Stack → `/(`
    Step 4: Read `D`: Output → `ED`
    Step 5: Read `-`: Stack → `/(-`
    Step 6: Read `C`: Output → `EDC`
    Step 7: Read `)`: Stack → `/` (Pop `-` to output, then pop `(`) → Output → `EDC-`
    Step 8: Read `*`: Stack → `*` (Pop `/` to output because `*` has higher precedence) → Output → `EDC-/`
    Step 9: Read `B`: Output → `EDC-/B`
    Step 10: Read `+`: Stack → `+` (Pop `*` to output) → Output → `EDC-/B*`
    Step 11: Read `A`: Output → `EDC-/B*A`
  - **Postfix of Reversed**: `EDC-/B*A+`
- **Reverse Postfix to Get Prefix**:
  - Reverse `EDC-/B*A+` → `+A*B/-CDE`
- **Output**: `+A*B/-CDE`

Step 12: End.

4. Design and Develop Java Program to implement Binary Tree traversals.

Aim:- To design and develop Java Program to implement Binary Tree traversals.

Description:

- **Tree Node Class**: Represents each node of the tree. It contains `value`, `left`, and `right` pointers.
- **BinaryTree Class**:
  - **inOrderTraversal(TreeNode node)**: Recursively traverses the tree in an in-order fashion.
  - **preOrderTraversal(TreeNode node)**: Recursively traverses the tree in a pre-order fashion.
  - **postOrderTraversal(TreeNode node)**: Recursively traverses the tree in a post-order fashion.
  - **levelOrderTraversal(TreeNode node)**: Uses a queue to traverse the tree level by level.
  - **Main Method**: Constructs a sample binary tree and performs all four types of traversals.

Algorithm:

1. In-order Traversal (Left, Root, Right)
Step 1. Start at the root node.
Step 2. Go Left: Move to the left child of the current node.
Step 3. Visit Node: Print or process the value of the current node.
Step 4. Go Right: Move to the right child of the current node.
Step 5. Repeat steps 2-4 until you have processed all nodes.
Step 6. If the current node is `null`, stop.
Step 7. Traverse the left subtree (apply the same steps).
Step 8. Print the current node's value.
Step 9. Traverse the right subtree (apply the same steps).

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

Step 1. Start at the root node.
Step 2. Visit Node: Print or process the value of the current node.
Step 3. Go Left: Move to the left child of the current node.
Step 4. Go Right: Move to the right child of the current node.
Step 5. Repeat steps 2-4 until you have processed all nodes.
Step 6. If the current node is `null`, stop.
Step 7. Print the current node's value.
Step 8. Traverse the left subtree (apply the same steps).
Step 9. Traverse the right subtree (apply the same steps).

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

Step 1. Start at the root node.
Step 2. Go Left: Move to the left child of the current node.
Step 3. Go Right: Move to the right child of the current node.
Step 4. Visit Node: Print or process the value of the current node.
Step 5. Repeat steps 2-4 until you have processed all nodes.
Step 6. If the current node is `null`, stop.
Step 7. Traverse the left subtree (apply the same steps).
Step 8. Traverse the right subtree (apply the same steps).
Step 9. Print the current node's value.

4. Level-order Traversal (Breadth-First Search)

Step 1. Start with the root node in a queue.
Step 2. Process Node: Remove the front node from the queue and print or process its value.
Step 3. Enqueue Children: Add the left and right children of the node to the queue.
Step 4. Repeat steps 2-3 until the queue is empty.
Step 5; End.

5. Design and Develop menu driven Java Program to implement Graph traversal techniques.

Aim: To design and develop menu driven Java Program to implement Graph traversal techniques.

Description:

- **Define the Graph Class**: This will include the adjacency list and methods for adding edges.
- **Implement DFS and BFS**: These methods will perform the respective graph traversals.
- **Create a Menu System**: To let the user choose the traversal technique and input graph data.

Algorithm:

Step 1. Define the Graph Structure
Step 2. Create a `Graph` class:
2.1 adjacencyList`: A `Map<Integer, List<Integer>>` to store the graph. The key is a vertex, and the value is a list of adjacent vertices.
Step 3. Methods in the `Graph` Class
3.1 addEdge(int source, int destination)`
3.2 Add `destination` to the adjacency list of `source`.
3.3 Add `source` to the adjacency list of `destination` (for undirected graph).
Step 4`depthFirstSearch(int start)`
4.1 Initialize a `Set<Integer>` to keep track of visited vertices.
4.2 Call `dfsHelper(start, visited)` to perform DFS traversal.
Step 5 dfsHelper(int node, Set<Integer> visited)
5.1 Mark `node` as visited.
5.2 Print the `node`.
5.3 Recursively visit all unvisited neighbors of `node`.
Step 6. `breadthFirstSearch(int start)`
6.1 Initialize a `Set<Integer>` for visited vertices and a `Queue<Integer>` for BFS.
6.2 Mark `start` as visited and add it to the queue.
6.3 While the queue is not empty:
6.4 Dequeue a vertex, print it.
6.5 Add all unvisited neighbors to the queue and mark them as visited.
Step 7`printGraph()`
7.1 Iterate through each vertex in the `adjacencyList`.
7.2 Print the vertex and its adjacent vertices.

2. Implement the Main Method

Step 1. Create a `Scanner` object to read user input.
Step 2. Create an instance of the `Graph` class.

Step 3. Display the Menu and Handle User Choices:
3.1 While Loop (Menu Loop):
3.2 Display Menu Options:
3.3 Add Edge
3.4 Print Graph
3.5 Perform DFS
3.6 Perform BFS
3.7 Exit

3. Complete Program Flow

Step 1. Start the Program :
1.1 Initialize the `Graph` and `Scanner`.
1.2 Enter the menu loop to handle user interactions.
Step 2. Perform Actions Based on User Input :
2.1 Add edges, print the graph, perform traversals as per user choices.
Step 3. Exit :
3.1 Clean up resources and terminate.
Step 4. Create `Graph` class with methods for adding edges, DFS, BFS, and printing the graph.
4.1 Implement the `main` method to handle menu-driven interactions.
Step 5. Menu System
Step 5.1 Display options to the user.
Step 5.2 Based on user input, call appropriate methods on the `Graph` instance.
Step 6.. Handle Choices:
6.1 Add Edge: Update the graph structure.
6.2 Print Graph: Output the current state of the graph.
6.3 DFS: Perform and display DFS traversal.0
6.4 BFS: Perform and display BFS traversal.
6.5 Exit: End the program.
Step 7: End

6. Design and Develop Java Program to implement Binary Search tree operations

Aim:- To design and develop Java Program to implement Binary Search tree operations

Algorithm:

1. Insertion

Step1. Start at the root of the BST.
Step 2. If the root is null , create a new node with the given value and return it. This new node becomes the root of the tree.
Step 3. Otherwise, compare the given value with the value of the current node:
3.1 If the given value is less than the current node's value, move to the left child of the current node.
3.2 If the given value is greater than the current node's value, move to the right child of the current node.
Step 4. Repeat step 2 and 3 until you find a null position to insert the new node.
Step 5. Return the unchanged node pointer after insertion.

2. Deletion

Step 1. Start at the root of the BST.
Step 2. If the root is null , the value is not found, so return null.
Step 3. Compare the value to be deleted with the current node's value:
3.1 If the value is less than the current node's value, recur to the left child .
3.2 If the value is greater than the current node's value, recur to the right child .
3.3 If the value matches the current node's value, perform the following steps:
3.4 Case 1 : Node has no children (leaf node): Simply remove the node and return null.
3.4 Case 2 : Node has one child : Replace the node with its child and return the child.
3.5 Case 3 : Node has two children :
3.6. Find the in order successor (smallest node in the right sub tree).
3.7 Replace the node's value with the in order successor's value.
3.8 Delete the in order successor node from the right sub tree.
Step 4. Return the (possibly updated) node pointer after deletion.

3. Search

Step 1. Start at the root of the BST.
Step 2. Compare the value to be searched with the current node's value:
2.1 If the value matches the current node's value, return true (value found).
2.2 If the value is less than the current node's value, recur to the left child.
2.3 If the value is greater than the current node's value, recur to the right child .
Step 3. If you reach a null node , return false (value not found).

4. Traversal

In-Order Traversal (Left-Root-Right):
Step 1. Recur on the left sub tree.
Step 2. Visit the root node.
Step 3. Recur on the right sub tree.

Pre-Order Traversal (Root-Left-Right):
Step 1. Visit the root node.
Step 2. Recur on the left sub tree.
Step 3. Recur on the right sub tree.

Post-Order Traversal (Left-Right-Root):
Step 1. Recur on the left sub tree.
Step 2. Recur on the right sub tree.
Step 3. Visit the root node.

Summary of Operations

Insertion :
Step 1: Traverse the tree to find the correct null position.
Step 2: Insert a new node.

Deletion :
Step1: Locate the node to be deleted.
Step2: Handle three cases (no children, one child, two children) accordingly.

Search :
Step:1 Traverse the tree based on comparisons to locate the value.

Traversals :
Step 1: Use recursive approaches to visit nodes in the desired order.

7. Design and Develop Java Program to implement AVL Trees.

Aim: To design and develop Java Program to implement AVL Trees.

Description:

AVL Trees are powerful data structures that maintain balance to ensure efficient operations. Understanding and implementing AVL Trees involves grasping the concepts of balance factors, rotations, and maintaining the binary search tree properties.

Algorithm:

1. AVL Tree Structure

Step1 start Node Structure:

Step 1.1: `key`: The value stored in the node.

Step 1.2 `height`: The height of the node in the tree.

Step 1.3 `left`: Reference to the left child node.

Step 1.4 `right`: Reference to the right child node.

Insertion Algorithm

Step 2. Perform Standard BST Insertion :

2.1 If the current node is `null`, create a new `AVLNode` with the given key and return it.

2.2 If the key is less than the current node's key, insert it into the left subtree.

2.3 If the key is greater than the current node's key, insert it into the right subtree.

2.4 If the key is equal to the current node's key, do nothing (since duplicate keys are not allowed).

Step 3 Update Height :

3.1 After inserting the node, update the height of the current node.

3.2 `height(node) = 1 + max(height(left), height(right))`.

Step 4. Balance the Tree :

4.1 Compute the balance factor for the current node.

4.2 If the balance factor is greater than 1, the tree is left-heavy.

4.3 If the balance factor is less than -1, the tree is right-heavy.

4.4 Perform rotations to restore balance.

Step 5. Balancing Algorithm

5.1. Calculate Balance Factor:

5.2 `balance(node) = height(left) - height(right)`.

Step 6. Check Balance:

6.1 If `balance(node) > 1` (left-heavy):

6.2 Left-Left Case : If the balance factor of the left child is `>= 0`, perform a right rotation.

6.3 Left-Right Case : If the balance factor of the left child is `< 0`, perform a left rotation on the left child, then a right rotation on the current node.

Step 7 : If `balance(node) < -1` (right-heavy):

7.1 Right-Right Case : If the balance factor of the right child is `<= 0`, perform a left rotation.

7.2 Right-Left Case : If the balance factor of the right child is `> 0`, perform a right rotation on the right child, then a left rotation on the current node.

Rotation Algorithms

Step1. Setup :
1.1 Let `x` be the left child of `y`.
1.2 Let `T2` be the right child of `x`.

Step 2. Perform Rotation :
2.1 Set `x.right` to `y`.
2.2 Set `y.left` to `T2`.

Step 3. Update Heights :
3.1 Update the height of `y` and `x`.

Left Rotation (on node `x`):

Step 1. Setup :
1.1 Let `y` be the right child of `x`.
1.2 Let `T2` be the left child of `y`.

Step 2. Perform Rotation :
2.1 Set `y.left` to `x`.
2.2 Set `x.right` to `T2`.

Step 3. Update Heights :
3.1 Update the height of `x` and `y`.

Left-Right Rotation :

Step 1. Perform Left Rotation on the left child of the node.
Step 2. Perform Right Rotation on the node.

Right-Left Rotation :

Step 1. Perform Right Rotation on the right child of the node.
Step 2. Perform Left Rotation on the node.
Step 3 End

8. Design and Develop menu driven Java Program to for Quick sort and Insertion sort.

Aim: To design and develop menu driven Java Program to for Quick sort and Insertion sort.

Description:

Quick Sort works by partitioning the array around a pivot and recursively sorting the sub-arrays. Its time complexity is average-case \(O(n \log n)\) and worst-case \(O(n^2)\) but is usually faster in practice due to its divide-and-conquer approach.

Insertion Sort works by iterating through the array and inserting each element into its correct position within the sorted portion. Its time complexity is \(O(n^2)\) in the worst case but is very efficient for small arrays or nearly sorted data.

Both algorithms have different use cases and performance characteristics, and choosing the right one depends on the specific requirements of the task at hand.

Algorithm:-

Step 1. Choose a Pivot:

1.1 Select an element from the array to act as a "pivot". Common strategies include choosing the last element, the first element, or the median.

Step 2. Partition the Array :

2.1 Rearrange the elements in the array such that all elements less than the pivot come before it, and all elements greater than the pivot come after it. The pivot is placed in its correct position.

Step 3. Recursively Apply Quick Sort:

3.1 Apply Quick Sort to the sub-array of elements before the pivot.

3.2 Apply Quick Sort to the sub-array of elements after the pivot.

Step 4. Base Case :

4.1 The recursion terminates when the sub-array has one or zero elements, which means it is already sorted.

Step 5: sort the array `[64, 25, 12, 22, 11]` using Quick Sort:

Step 6. Initial Array : `[64, 25, 12, 22, 11]`

Step 7: Choose Pivot and choose the last element, `11`.

Step 8. Partitioning:

8.1 Elements less than `11`: `[11]` (pivot position)

8.2 Elements greater than `11`: `[64, 25, 12, 22]`

8.3 Re-arrange to `[11, 25, 12, 22, 64]` with `11` in the correct position.

Step 9 :Recursively Apply

9.1 Apply Quick Sort to `[25, 12, 22, 64]`:

9.2 Pivot: `64`

9.3 Partitioned: `[25, 12, 22, 64]` (no change needed as `64` is already in place)

9.4 Apply Quick Sort to `[25, 12, 22]`:

9.5 Pivot: `22`
9.6 Partitioned: `[12, 22, 25]`
9.6 No further recursion needed.
Step 10: Result: The fully sorted array is `[11, 12, 22, 25, 64]`.

Insertion Sort Algorithm

Step 1. Start from the Second Element :
1.1 Assume the first element is already sorted. Begin with the second element.
Step 2. Insert the Element :
2.1 Compare the current element with the elements in the sorted portion (left side).
2.2 Shift all elements that are greater than the current element to the right to make space.
2.3 Insert the current element into its correct position.
Step 3. Repeat :
3.1 Move to the next element and repeat the process until the end of the array.
Step 4. Base Case :
4.1 The process terminates when all elements are processed and inserted into their correct positions.
Step 5: End

9. Design and Develop menu driven Java Program to for Heap sort and Merge sort.

Aim: - To design and develop menu driven Java Program to for Heap sort and Merge sort
Description: To create a Java program that allows users to choose between Heap Sort and Merge Sort using a menu-driven approach, you'll need to follow these steps:

1. Implement Heap Sort and Merge Sort algorithms.
2. Create a menu system to allow users to select which sort algorithm they want to use.
3. Take user input to sort an array.
4. Display the sorted output.

Algorithm:
Step 1 : start
Step 2: Collect array data from the user.
Step 3. Print "Enter the number of elements:".
Step 4: Read the number of elements.
Step 5: . Initialize an array of the specified size.
Step 6: Print "Enter the elements:".
Step 7. Read and store elements in the array.
Step 8 Sort the Array Using Heap Sort
8.1 Purpose: Sort the array using Heap Sort algorithm.
8.2. Call `heapSort(array)` method.
Step 9. Heap Sort Implementation
9.1 Purpose: Sort the array using the heap sort technique.
9.2 Build Max Heap
9.3 Start from the last non-leaf node and move upwards.
9.4 Call `heapify` to ensure the max heap property.
Step 10 Extract Elements
10.1 Swap the root (largest element) with the last element.
10.2 Reduce the heap size and call `heapify` to restore the heap property.
10.3 Repeat until the heap size is reduced to zero.
Step 11: Print Sorted Array
11.1. Call `printArray(array)` method.
Step 12: Merge Sort Algorithm:
12.1. Print "Enter the number of elements:".
12.2. Read the number of elements.
12.3. Initialize an array of the specified size.
12.4. Print "Enter the elements:".
12.5. Read and store elements in the array.

Step 13. Sort the Array Using Merge Sort
13.1. Call `mergeSort(array, 0, array.length - 1)` method.
Step 14:. Merge Sort Implementation
14.1 Find the middle index of the array.
14.2 Recursively sort the left and right halves.
Step15: Merge
15.1 Combine two sorted halves into a single sorted array.
15.2 Use temporary arrays to assist with merging.
Step 16 : Print Sorted Array
16.1. Call `printArray(array)` method.
Step 17 Display Sorted Array
17.1. Iterate through the array.
17.2. Print each element followed by a space.
17.3. Print a newline character.
Step 18: End.

10. Design and Develop menu driven Java Program to implement Linear search and Binary Search.

Aim: - To design and develop menu driven Java Program to implement Linear search and Binary Search.

Description:

1. Linear Search:
    - Prompts the user to enter the number of elements and then the elements themselves.
    - It then prompts for the target value to search.
    - It performs a linear search through the array and displays the result.
2. Binary Search:
    - Prompts the user to enter the number of elements and then the sorted elements themselves.
    - It then prompts for the target value to search.
    - It performs a binary search through the array and displays the result.
3. Input Validation: The `getValidIntInput()` method ensures that only valid integer inputs are accepted.

Algorithm:-

Step 1. Start Program
1.1. Display the menu options.
1.2. Read the user's choice.
Step 2. Process User Choice
2.1 If Choice is Linear Search:**
2.2 Input Array Size and Elements:**
2.3 Prompt the user to enter the number of elements.
2.4 Read the number of elements.
2.5 Initialize an array with the given size.
2.6 Prompt the user to enter each element of the array.
2.7 Store each input into the array.
Step 3 Input Target Element:
3.1 Prompt the user to enter the element to search for.
3.2 Read the target element.
Step 4 : Perform Linear Search:
4.1 Initialize a variable `index` to -1.
4.2 Loop through each element of the array:
4.3 If the current element matches the target, set `index` to the current position and break the loop.

4.4 After the loop, check if `index` is -1:
4.5 If yes, print "Element not found."
4.6 Otherwise, print "Element found at index: `index`."

Step 5: If Choice is Binary Search:
5.1 Input Array Size and Elements
5.2 Prompt the user to enter the number of elements.
5.3 Read the number of elements.
5.4 Initialize an array with the given size.
5.5 Prompt the user to enter each element of the array in sorted order.
5.6 Store each input into the array.

Step 6 : Input Target Element:
6.1 Prompt the user to enter the element to search for.
6.2 Read the target element.

Step 7 : Perform Binary Search:
7.1 Initialize two variables, `left` to 0 and `right` to `array.length - 1`.
7.2 While `left` is less than or equal to `right`:
7.3 Compute `mid` as the middle index `(left + right) / 2`.
7.4 If the element at `mid` equals the target, print "Element found at index: `mid`" and exit.
7.5 If the element at `mid` is less than the target, adjust `left` to `mid + 1`.
7.6 If the element at `mid` is greater than the target, adjust `right` to `mid - 1`.
7.7 If the loop ends without finding the target, print "Element not found."

Step 8. If Choice is Exit:
8.1 Print "Exiting..." and terminate the program.

Linear Search Algorithm

Step 9. Start Linear Search
9.1. Input Array and Target Element:
9.2 Read the array of `n` elements.
9.3 Read the target element.

Step 10 Search Process
10.1 Initialize `index` to -1.
10.2 For each element in the array from index 0 to `n-1`:
10.3 If the current element equals the target element:
10.4 Set `index` to the current index.
10.5 Break the loop.

Step 11 Check Result
11.1 If `index` is -1, the target is not found.
11.2 Otherwise, return `index` where the target is found.

Binary Search Algorithm

Step 1. Start Binary Search
1.1 Input Array and Target Element
1.2 Read the sorted array of `n` elements.
1.3 Read the target element.
Step 2. Search Process:
2.1 Initialize `left` to 0 and `right` to `n-1`.
2.2 While `left` is less than or equal to `right`:
2.3 Compute `mid` as `(left + right) / 2`.
2.4 If the element at `mid` equals the target element:
2.5 Return `mid`.
2.6 If the element at `mid` is less than the target:
2.7 Update `left` to `mid + 1`.
2.8 If the element at `mid` is greater than the target:
2.9 Update `right` to `mid - 1`.
Step 3. Check Result:**
3.1 If the loop ends without finding the target, return -1 indicating the target is not found.
Step 4: End;