Worksheet B

Q1. Option C – Compilation error. Due to recursive constructor call in class B, which is not allowed in Java.

Q2.

```java
package Worksheet_B;

public class vowels {
    public static void main(String[] args) {
        String input1 = "Hello";
        String input2 = "nm";

        System.out.println("String: " + input1 + ", Contains Vowel: " + containsVowel(input1));
        System.out.println("String: " + input2 + ", Contains Vowel: " + containsVowel(input2));
    }

    public static boolean containsVowel(String str) {
        String vowels = "aeiouAEIOU";

        for (char ch : str.toCharArray()) {
            if (vowels.indexOf(ch) != -1) {
                return true;
            }
        }

        return false;
    }
}
```

Main Method:

- In the main method, two strings are initialized: input1 with "Hello" and input2 with "nm".
- Each string's vowel presence is determined by calling the containsVowel method, and the results are printed with corresponding descriptive messages.

containsVowel Method:

- The containsVowel method receives a string str as its parameter.
- It declares a string vowels containing all lowercase and uppercase vowels ("aeiouAEIOU").
- Using a for-each loop, it iterates over each character (ch) in the input string str.

- Within the loop, it checks if the character ch exists in the vowels string using indexOf(ch). If ch is found (i.e., its index is not -1), the method returns true, indicating the presence of at least one vowel.
- If no vowels are found after iterating through the entire string, the method returns false.

Q3.

```
package Worksheet_B;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;

public class RemoveDuplicatesArrayList {
   public static void main(String[] args) {
      // Dummy ArrayList
      ArrayList<Integer> listWithDuplicates = new ArrayList<>(
         List.of(1, 2, 3, 1, 4, 2, 5));

      System.out.println("ArrayList with duplicates: " + listWithDuplicates);

      HashSet<Integer> set = new HashSet<>();

      ArrayList<Integer> listWithoutDuplicates = new ArrayList<>();

      for (Integer element : listWithDuplicates) {
         if (!set.contains(element)) {
            set.add(element);
            listWithoutDuplicates.add(element);
         }
      }

      System.out.println("ArrayList without duplicates: " + listWithoutDuplicates);
   }
```

- Initializes an ArrayList listWithDuplicates containing integers with duplicates.

- Creates a HashSet set to store unique elements encountered during iteration.

- Iterates through each element (element) in listWithDuplicates.
- Checks if set already contains element. If not (!set.contains(element)), adds element to both set and listWithoutDuplicates.

- Prints the original ArrayList listWithDuplicates with duplicates.
- Prints the filtered ArrayList listWithoutDuplicates without duplicates, achieved by leveraging HashSet to ensure uniqueness.

Q4.

```
package Worksheet_B;

import java.util.HashSet;

// Definition of a ListNode
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
        this.next = null;
    }
}

public class UnionIntersectionLinkedList {
    public static void main(String[] args) {
        ListNode list1 = createLinkedList(new int[] {1, 2, 3, 4, 5});
        ListNode list2 = createLinkedList(new int[] {3, 4, 5, 6, 7});

        System.out.println("List 1: ");
        printLinkedList(list1);
        System.out.println("List 2: ");
        printLinkedList(list2);

        ListNode union = findUnion(list1, list2);
        ListNode intersection = findIntersection(list1, list2);

        System.out.println("Union of the lists: ");
        printLinkedList(union);
        System.out.println("Intersection of the lists: ");
        printLinkedList(intersection);
    }

    public static ListNode createLinkedList(int[] arr) {
        ListNode dummy = new ListNode(-1);
        ListNode current = dummy;

        for (int num : arr) {
            current.next = new ListNode(num);
            current = current.next;
        }

        return dummy.next;
```

```
}

public static void printLinkedList(ListNode head) {
    ListNode current = head;

    while (current != null) {
        System.out.print(current.val + " ");
        current = current.next;
    }
    System.out.println();
}

public static ListNode findUnion(ListNode head1, ListNode head2) {
    HashSet<Integer> set = new HashSet<>();
    ListNode dummy = new ListNode(-1);
    ListNode current = dummy;

    ListNode ptr1 = head1;
    while (ptr1 != null) {
        if (!set.contains(ptr1.val)) {
            set.add(ptr1.val);
            current.next = new ListNode(ptr1.val);
            current = current.next;
        }
        ptr1 = ptr1.next;
    }

    ListNode ptr2 = head2;
    while (ptr2 != null) {
        if (!set.contains(ptr2.val)) {
            set.add(ptr2.val);
            current.next = new ListNode(ptr2.val);
            current = current.next;
        }
        ptr2 = ptr2.next;
    }

    return dummy.next;
}

public static ListNode findIntersection(ListNode head1, ListNode head2) {
    HashSet<Integer> set = new HashSet<>();
    HashSet<Integer> common = new HashSet<>();

    ListNode ptr1 = head1;
    while (ptr1 != null) {
        set.add(ptr1.val);
```

```
            ptr1 = ptr1.next;
        }

        ListNode ptr2 = head2;
        while (ptr2 != null) {
            if (set.contains(ptr2.val)) {
                common.add(ptr2.val);
            }
            ptr2 = ptr2.next;
        }

        ListNode dummy = new ListNode(-1);
        ListNode current = dummy;

        for (int num : common) {
            current.next = new ListNode(num);
            current = current.next;
        }

        return dummy.next;
    }
}
```

Main Method:

- Creates two linked lists (list1 and list2) using createLinkedList method and prints them using printLinkedList.
- Finds the union and intersection of list1 and list2 using findUnion and findIntersection methods respectively.
- Prints the union and intersection results using printLinkedList.

createLinkedList Method:

- Converts an integer array (arr) into a linked list of ListNode objects.
- Returns the head of the linked list.

printLinkedList Method:

- Prints the values of nodes in a linked list starting from the given head (head).

findUnion Method:

- Finds the union of two linked lists by iterating through both lists and using a HashSet (set) to track unique values.
- Constructs a new linked list (dummy) containing unique values found in both lists.

findIntersection Method:

- Finds the intersection of two linked lists by iterating through one list (head1) and checking for common elements in the other list (head2).
- Constructs a new linked list (dummy) containing nodes with values present in both lists.

Q5.

```
package Worksheet_B;

public class MiddleSum {
   public static void main(String[] args) {
      int[][] matrix = {
            { 1, 2, 3 },
            { 4, 5, 6 },
            { 7, 8, 9 },
            { 10, 11, 12 }
      };

      int middleRow = matrix.length / 2;
      int middleColumn = matrix[0].length / 2;

      int sumMiddleRow = calculateSumOfRow(matrix, middleRow);
      int sumMiddleColumn = calculateSumOfColumn(matrix, middleColumn);

      System.out.println("Matrix:");
      printMatrix(matrix);

      System.out.println("Sum of middle row (row " + middleRow + "): " + sumMiddleRow);
      System.out.println("Sum of middle column (column " + middleColumn + "): " +
sumMiddleColumn);
   }

   public static int calculateSumOfRow(int[][] matrix, int rowIndex) {
      int sum = 0;
      for (int col = 0; col < matrix[rowIndex].length; col++) {
         sum += matrix[rowIndex][col];
      }
      return sum;
   }

   public static int calculateSumOfColumn(int[][] matrix, int colIndex) {
      int sum = 0;
      for (int row = 0; row < matrix.length; row++) {
         sum += matrix[row][colIndex];
      }
      return sum;
   }
```

```
    public static void printMatrix(int[][] matrix) {
        for (int[] row : matrix) {
            for (int num : row) {
                System.out.print(num + " ");
            }
            System.out.println();
        }
    }
}
```

Matrix Initialization:

- Defines a 2D array matrix containing integer values.

Finding Middle Indices:

- Determines middleRow as the index of the middle row (matrix.length / 2).
- Determines middleColumn as the index of the middle column (matrix[0].length / 2).

Calculating Sum of Middle Row:

- Utilizes calculateSumOfRow method to compute the sum of elements in the middle row identified by middleRow.

Calculating Sum of Middle Column:

- Utilizes calculateSumOfColumn method to compute the sum of elements in the middle column identified by middleColumn.

Printing Matrix:

- Outputs the original matrix using printMatrix method to display its contents.

Q6.

```
package Worksheet_B;

// Definition of ListNode
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
        this.next = null;
    }
```

```
}

public class MergedLinkedList {
    public static void main(String[] args) {
        ListNode list1 = createLinkedList(new int[] { 1, 3, 5, 7 });
        ListNode list2 = createLinkedList(new int[] { 2, 4, 6, 8, 9 });

        System.out.println("List 1:");
        printLinkedList(list1);
        System.out.println("List 2:");
        printLinkedList(list2);

        ListNode mergedList = mergeSortedLists(list1, list2);

        System.out.println("Merged Sorted List:");
        printLinkedList(mergedList);
    }

    public static ListNode createLinkedList(int[] arr) {
        ListNode dummy = new ListNode(-1);
        ListNode current = dummy;

        for (int num : arr) {
            current.next = new ListNode(num);
            current = current.next;
        }

        return dummy.next;
    }

    public static void printLinkedList(ListNode head) {
        ListNode current = head;

        while (current != null) {
            System.out.print(current.val + " ");
            current = current.next;
        }
        System.out.println();
    }

    public static ListNode mergeSortedLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(-1);
        ListNode current = dummy;

        while (l1 != null && l2 != null) {
            if (l1.val <= l2.val) {
                current.next = l1;
```

```
            l1 = l1.next;
        } else {
            current.next = l2;
            l2 = l2.next;
        }
        current = current.next;
    }

    if (l1 != null) {
        current.next = l1;
    }
    if (l2 != null) {
        current.next = l2;
    }

    return dummy.next;
  }
}
```

Main Method:

- Creates two sorted linked lists (list1 and list2) using the createLinkedList method, initialized with sorted integer arrays.
- Prints both lists using printLinkedList to display their contents.
- Merges the two sorted lists into a single sorted list using mergeSortedLists method.
- Prints the merged sorted list using printLinkedList.

createLinkedList Method:

- Converts an integer array (arr) into a linked list of ListNode objects.
- Returns the head of the linked list.

printLinkedList Method:

- Traverses and prints each element of a linked list starting from the given head (head).

mergeSortedLists Method:

- Combines two sorted linked lists (l1 and l2) into a single sorted linked list.
- Uses a dummy node as the starting point for the merged list to simplify the edge case handling.
- Iterates through both lists, comparing nodes and linking them accordingly in ascending order.
- Appends any remaining nodes from l1 or l2 that were not fully traversed.

Q7.

```
package Worksheet_B;

import java.util.*;

class TreeNode {
   int val;
   TreeNode left;
   TreeNode right;

   TreeNode(int val) {
      this.val = val;
      this.left = null;
      this.right = null;
   }
}

public class BottomViewBinary {
   public static void main(String[] args) {
      TreeNode root = new TreeNode(20);
      root.left = new TreeNode(8);
      root.right = new TreeNode(22);
      root.left.left = new TreeNode(5);
      root.left.right = new TreeNode(3);
      root.right.left = new TreeNode(4);
      root.right.right = new TreeNode(25);
      root.left.right.left = new TreeNode(10);
      root.left.right.right = new TreeNode(14);

      System.out.println("Bottom View of the Binary Tree:");
      printBottomView(root);
   }

   public static void printBottomView(TreeNode root) {
      if (root == null) {
         return;
      }

      Map<Integer, Integer> map = new TreeMap<>();

      Queue<TreeNode> queue = new LinkedList<>();
      Queue<Integer> hdQueue = new LinkedList<>();

      queue.offer(root);
      hdQueue.offer(0);

      while (!queue.isEmpty()) {
```

```
            TreeNode node = queue.poll();
            int hd = hdQueue.poll();

            map.put(hd, node.val);

            if (node.left != null) {
                queue.offer(node.left);
                hdQueue.offer(hd - 1);
            }

            if (node.right != null) {
                queue.offer(node.right);
                hdQueue.offer(hd + 1);
            }
        }

        for (int key : map.keySet()) {
            System.out.print(map.get(key) + " ");
        }
        System.out.println();
    }
}
```

Main Method:

- Creates a binary tree with specified nodes and values.
- Calls printBottomView method to print the bottom view of the tree.

printBottomView Method:

- Uses a TreeMap (map) to store the bottom view nodes based on their horizontal distance (hd).
- Utilizes two queues (queue for nodes and hdQueue for horizontal distances) to perform level-order traversal.
- Updates map with the latest node value for each horizontal distance.
- Prints the values from map in ascending horizontal distance order, representing the bottom view of the binary tree.

Q8.

```
package Worksheet_B;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
```

```
    TreeNode(int val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}

public class MirrorBinaryTree {
    public static void main(String[] args) {
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.left = new TreeNode(4);
        root.left.right = new TreeNode(5);

        System.out.println("Original Binary Tree:");
        printTree(root);

        TreeNode mirror = convertToMirror(root);

        System.out.println("\nMirror Binary Tree:");
        printTree(mirror);
    }

    public static void printTree(TreeNode root) {
        if (root == null) {
            return;
        }
        printTree(root.left);
        System.out.print(root.val + " ");
        printTree(root.right);
    }

    public static TreeNode convertToMirror(TreeNode root) {
        if (root == null) {
            return null;
        }

        TreeNode left = convertToMirror(root.left);
        TreeNode right = convertToMirror(root.right);

        root.left = right;
        root.right = left;

        return root;
    }
}
```

Main Method:

- Creates an original binary tree with specified nodes and values.
- Prints the original tree structure using printTree.
- Converts the original tree into its mirror image using convertToMirror.
- Prints the mirror tree structure using printTree.

printTree Method:

- Recursively prints the binary tree in an in-order traversal (left subtree, root, right subtree).

convertToMirror Method:

- Recursively converts the given binary tree into its mirror image by swapping left and right child nodes at each node.

Q9.

```
package Worksheet_B;

class TreeNode {
   int val;
   TreeNode left;
   TreeNode right;

   TreeNode(int x) {
      val = x;
   }
}

public class IdenticalTrees {

   public boolean isIdentical(TreeNode p, TreeNode q) {
      if (p == null && q == null) {
         return true;
      }

      if (p == null || q == null) {
         return false;
      }

      return (p.val == q.val) && isIdentical(p.left, q.left) && isIdentical(p.right, q.right);
   }
```

```
    public static void main(String[] args) {
        IdenticalTrees solution = new IdenticalTrees();

        TreeNode root1 = new TreeNode(1);
        root1.left = new TreeNode(2);
        root1.right = new TreeNode(3);

        TreeNode root2 = new TreeNode(1);
        root2.left = new TreeNode(2);
        root2.right = new TreeNode(3);

        boolean identical = solution.isIdentical(root1, root2);
        if (identical) {
            System.out.println("The two binary trees are identical.");
        } else {
            System.out.println("The two binary trees are not identical.");
        }
    }
}
```

The program defines a TreeNode class for a binary tree node. The main class, IdenticalTrees, has a method isIdentical that recursively checks if two binary trees are identical by comparing node values and their structures. If both nodes are null, they're identical. The main method creates two example trees, compares them using isIdentical, and prints whether they are identical or not based on the comparison result.

Q10.

```
package Worksheet_B;

public class PowerOfTwo {

    public boolean isPowerOfTwo(int n) {
        if (n <= 0) {
            return false;
        }
        return (n & (n - 1)) == 0;
    }

    public static void main(String[] args) {
        PowerOfTwo solution = new PowerOfTwo();

        int num1 = 16;
        int num2 = 17;
```

```
        System.out.println(num1 + " is power of two: " + solution.isPowerOfTwo(num1));
        System.out.println(num2 + " is power of two: " + solution.isPowerOfTwo(num2));
    }
}
```

The program defines a class PowerOfTwo with a method isPowerOfTwo(int n) that checks if a given integer n is a power of two. It first handles edge cases by returning false if n is less than or equal to zero since negative numbers and zero cannot be powers of two.

The core logic relies on a bitwise operation: (n & (n - 1)) == 0. This operation checks if n has exactly one '1' bit in its binary representation, which is a defining characteristic of powers of two