W5 - Data Structures & Algorithms

Problem 1: Sorting Large Array of Random Integers

Overview: Generate an array of 1,000,000 random integers ranging from 1 to 1,000. Sort this array using both a custom counting sort algorithm and Java's built-in Arrays.sort method. Compare the performance of these two sorting methods in terms of running time.

Algorithm Steps:

1. **Generate Random Integers**:
    - Create an array to hold 1,000,000 random integers.
    - Populate the array with random integers ranging from 1 to 1,000.
2. **Counting Sort**:
    - Implement a counting sort algorithm to sort the array.
    - Measure the time taken to complete the sorting.
3. **Arrays.sort Method (2A)**:
    - Use Java's built-in Arrays.sort method to sort the array.
    - Measure the time taken to complete the sorting.
4. **Compare Running Times (2B)**:

- Compare the running times of both sorting algorithms.

5. **Code**:

```
import java.util.Arrays;
import java.util.Random;

public class SortingComparison {

   // Method to generate an array of random integers
   private static int[] generateRandomArray(int size, int maxValue) {
       Random random = new Random();
       int[] array = new int[size];
       for (int i = 0; i < size; i++) {
           array[i] = random.nextInt(maxValue) + 1;
       }
       return array;
   }

   // Method to perform counting sort
   private static void countingSort(int[] array, int maxValue) {
       int[] count = new int[maxValue + 1];
       int[] output = new int[array.length];

       // Count each element
       for (int value : array) {
           count[value]++;
       }

       // Modify count array
       for (int i = 1; i <= maxValue; i++) {
           count[i] += count[i - 1];
       }

       // Build the output array
       for (int i = array.length - 1; i >= 0; i--) {
           output[count[array[i]] - 1] = array[i];
           count[array[i]]--;
       }
```

```
        // Copy the sorted elements back into the original array
        System.arraycopy(output, 0, array, 0, array.length);
    }

    public static void main(String[] args) {
        int size = 1_000_000;
        int maxValue = 1_000;

        // Generate random integers
        int[] arrayForCountingSort = generateRandomArray(size, maxValue);
        int[] arrayForBuiltInSort = Arrays.copyOf(arrayForCountingSort, arrayForCountingSort.length);

        // Sort using counting sort and measure time
        long startTime = System.nanoTime();
        countingSort(arrayForCountingSort, maxValue);
        long countingSortTime = System.nanoTime() - startTime;

        // Sort using Java's built-in method and measure time
        startTime = System.nanoTime();
        Arrays.sort(arrayForBuiltInSort);
        long builtInSortTime = System.nanoTime() - startTime;

        // Output the time taken by both sorting algorithms
        System.out.println("Time taken by Counting Sort: " + countingSortTime / 1000000 + " ms");
        System.out.println("Time taken by Arrays.sort: " + builtInSortTime / 1000000 + " ms");
    }
}
```

Problem 2: Implement Hash Table for RMIT Student Information

Overview: Implement a hash table to manage RMIT student information using a custom hash function. The hash table will store RMITStudent objects with student IDs as keys. Implement two collision resolution strategies: Separate Chaining (2A) and Linear Probing (2B).

Algorithm Steps:

1. **Hash Function**:
    - Design a hash function h(S) that converts a string key into an integer. It sums the individual character hashes modulo the hash table size N.
    - Map characters 'A' to 'Z' and digits '0' to '9' to integers 0 to 35. The hash table size N is 13.
2. **Class Definitions**:
    - RMITStudent: Class representing an RMIT student with properties studentId, fullName, major, and GPA.
    - RMITStudentCollection: Class representing the hash table with put and get methods.
3. **Separate Chaining**:
    - Implement collision handling using separate chaining, where each hash table entry is a linked list of collided keys.
4. **Linear Probing**:
    - Implement collision handling using linear probing, where collisions are resolved by probing subsequent indices until an empty slot is found.
5. **Code**:

```
class RMITStudent {
    String studentId;
    String fullName;
    String major;
    double GPA;
```

```
    // Constructor and other methods
}

 class RMITStudentCollection {
    private static final int SIZE = 13;
    private RMITStudent[] students; // For linear probing
    private LinkedList<RMITStudent>[] studentLists; // For separate chaining

    // Constructor and methods for linear probing and separate chaining

    // Custom hash function
    private int hash(String key) {
        int hashValue = 0;
        for (char c : key.toCharArray()) {
            if (c >= 'A' && c <= 'Z') {
                hashValue += c - 'A';
            } else if (c >= '0' && c <= '9') {
                hashValue += 26 + c - '0';
            }
        }
        return hashValue % SIZE;
    }

    // Put method for linear probing
    boolean putLinear(RMITStudent student) {
        int index = hash(student.studentId);
        while (students[index] != null && !students[index].studentId.equals(student.studentId)) {
            index = (index + 1) % SIZE;
        }
        if (students[index] != null) {
            return false; // Duplicate student ID
        }
        students[index] = student;
        return true;
    }

    // Get method for linear probing
    RMITStudent getLinear(String studentId) {
        int index = hash(studentId);
        while (students[index] != null) {
            if (students[index].studentId.equals(studentId)) {
                return students[index];
            }
            index = (index + 1) % SIZE;
        }
        return null;
    }

    // Put method for separate chaining
    boolean putChain(RMITStudent student) {
        int index = hash(student.studentId);
        if (studentLists[index] == null) {
            studentLists[index] = new LinkedList<>();
        } else {
            for (RMITStudent s : studentLists[index]) {
                if (s.studentId.equals(student.studentId)) {
                    return false; // Duplicate student ID
                }
            }
```

```
        }
        studentLists[index].add(student);
        return true;
    }

    // Get method for separate chaining
    RMITStudent getChain(String studentId) {
        int index = hash(studentId);
        if (studentLists[index] != null) {
            for (RMITStudent student : studentLists[index]) {
                if (student.studentId.equals(studentId)) {
                    return student;
                }
            }
        }
        return null;
    }
}
```

Problem 3: Extended - Implement Remove Operation in Hash Table for RMIT Student Information

Overview: Enhance the previously implemented hash table for RMIT student information by adding a remove operation. This operation will remove a student object based on the provided student ID. Implement this for both Separate Chaining (2A) and Linear Probing (2B) collision resolution strategies.

Algorithm Steps:

1. Remove Operation for Separate Chaining:
   - Search for the student in the linked list at the hashed index. If found, remove the student object and return true. If not found, return false.
2. Remove Operation for Linear Probing:
   - Search for the student in the array starting from the hashed index. If found, remove the student object and return true. If not found, return false.
3. **Code**:

```
class RMITStudentCollection {
    // Existing fields and methods

    // Remove method for separate chaining
    boolean removeChain(String studentId) {
        int index = hash(studentId);
        if (studentLists[index] != null) {
            Iterator<RMITStudent> iterator = studentLists[index].iterator();
            while (iterator.hasNext()) {
                RMITStudent student = iterator.next();
                if (student.studentId.equals(studentId)) {
                    iterator.remove();
                    return true;
                }
            }
        }
        return false;
    }

    // Remove method for linear probing
    boolean removeLinear(String studentId) {
        int index = hash(studentId);
```

```
        while (students[index] != null) {
            if (students[index].studentId.equals(studentId)) {
                students[index] = null; // Remove the student
                // Rehash all students in the same cluster
                rehashCluster(index);
                return true;
            }
            index = (index + 1) % SIZE;
        }
        return false;
    }

    // Helper method for rehashing in linear probing
    private void rehashCluster(int startIndex) {
        int index = (startIndex + 1) % SIZE;
        while (students[index] != null) {
            RMITStudent rehashedStudent = students[index];
            students[index] = null;
            putLinear(rehashedStudent); // Re-insert the student
            index = (index + 1) % SIZE;
        }
    }
}
```