1. Represent the data structures graphically after each of the following instructions execute:

**LinkedList ls1 = new LinkedList();**

**ls1.add(1);**

**ls1.add(2);**

**ls1.add(3);**

**ls1.add(4);**

**ls1.remove(5);**

**ls1.remove(3);**

**ArrayList ls2 = new ArrayList(2);**

**ls2.add(1);**

**ls2.add(2);**

**ls2.add(3);**

**ls2.add(4);**

**ls2.remove(5);**

**ls2.remove(3);**

2. Represent the following instructions over a hashmap using both collision resolution schemes each in a single drawing. Assume chaining does not resize the underlying data structure, and the initial backing array has size = 4. The mappings are as follows,

Dennis, 4 Joy, 1 Hung, 2 PM, 0 Victoria, 8

**Hashmap hm = new Hashmap();**

**hm.put(“Dennis”, 21);**

**hm.put(“Joy”, 21);**

**hm.put(“PM”, 57);**

**hm.put(“Victoria”, 25);**

**hm.put(“Hung”, 20);**

3. What operations do these data structures share? How do their time complexities compare over these operations?

4. Demonstrate graphically how to sort the following array using both merge and quick sort.

[6, 5, 3, 1, 8, 7, 2, 4]

5. Given the following class signature, arrange the following pseudo-codes to the proper invocation order that would correctly recursively sort an array according to each sorting algorithms. When finished, consider what tasks need to perform for each of the algorithms, and how they differ.

|  |  |
| --- | --- |
| **mergeSort( )**  **mergeSort( )**  **sort( )**  **If (base\_case) return array/index** | **quickSort( )**  **quickSort( )**  **sort( )**  **If (base\_case) return array/index** |
| **function mergeSort(array)** | **function quicksort(array)** |

Tree add/delete

Arraylist

Hashmap Collison resolution

Sequential data structure comparative analysis