Context

When UW switched to online-only instruction in Winter Quarter 2020, CSE 373 cancelled its final and implemented an optional “final assignment” covering topics from before the midterm, for students who wanted to improve their grades using the final exam.

Questions

Question 1: Instructions

All point values are temporary, and will be adjusted later.

You can punch the Submit button below a couple of times to relax. Actually, you can submit your answers as many times as you'd like (and we'd recommend doing it fairly frequently); we will only grade your final submission.

To add people into your group, click Submit below, then scroll to the very bottom of the page and click View Your Submission.

Question 2: Testing and Debugging

You have been working on the following method to determine whether a `String` is a palindrome. After drafting an implementation, you also want to test your code. The following two sub-questions involve this code snippet.

```
/**
 * Determines whether a String is a palindrome.
 *
 * (A palindrome is a string that has the same sequence of chars
when
 * read forwards and backwards. For example, "tacocat" and "11" are
 * palindromes. "Tacocat" ('T' vs 't') and "12" are not.)
 *
 * Arguments: `s` is an input string to be examined.
 *
 * Returns: true if `s` is a palindrome, false otherwise.
 */
public boolean isPalindrome(String s) {
  // (... Implementation omitted ...)
  return true;
```

```
}
```

Q2.1 Formulating Test Cases

Briefly *describe* 5 unique, consistently reproducible test cases that could verify the promised functionality of your implementation.

Q2.2 The Debugging Process

Someone passed all known English palindromic words into your implementation, and noticed that 37.3% of the time, the method does not correctly return `true`. What is a good next step to debug the problem?

Question 3: Heaps

In 2–5 sentences, describe why the runtime for Floyd's `buildHeap` algorithm is O(n), instead of being merely O(n log n). If you prefer, you may submit a proof instead.

Question 4: Hashing

Suppose we have a hash function that uniformly distributes keys over its range, and a hash table using that hash function with a prime table size. Assume that λ (the load factor) is 1.

Q4.1 Separate Chaining

What runtime would we expect `insert(10)` to have, if the hash table is **separate-chaining** and uses a linked list? Describe your answer in 1–2 sentences.

Q4.2 Open Addressing

What runtime would we expect `insert(10)` to have, if the hash table is **open-addressing** and uses linear probing? Describe your answer in 1–2 sentences.

Question 5: Multi-Dimensional Data

Refer to slide 10 from lecture 12 (k-d trees). Although we provided pseudocode for `nearest()`, we deliberately stayed high-level when discussion its helper method, `mightHaveSomethingUseful()`.

Below, please describe the algorithm for `mightHaveSomethingUseful()`. If you prefer, you may submit pseudocode instead.

Question 6: ADTs and Data Structures

For each of the following scenarios, choose the combination of ADT and implementing data structure that best accomplishes the task. When considering "best", you should prioritize efficient runtime first, then efficient memory usage.

Give a brief explanation (2–4 sentences) of how you would use your chosen ADT and data structure to solve the task. Your explanation should include a description of what data is stored in the data structure and which operations you would use.

Q6.1 Two-Dimensional Iterator

You have a list of 2 dimensional points. You need a `getNextClosest()` method that, when called repeatedly, will sequentially return the first, then second, then third, etc. closest points to some FIXED (ie, predetermined) point. You don't necessarily need to sort all the points.

ADT: ○ List ○ Set ○ Map ○ Priority Queue ○ Point Set ○ Disjoint Set

Data Structure: ○ Array ○ Linked List ○ BST (Binary Search Tree)
○ LLRB (Left-leaning Red/Black) Tree ○ Hash Table ○ Binary Heap
○ k-d Tree ○ WeightedQuickUnion (with Path Compression)

Explanation:

Q6.2 Bounded Number Collection

You have a collection of numbers. You want to query for the minimum value in the collection that is greater than some arbitrary input number; the input number is not necessarily in the provided collection.

ADT: ○ List ○ Set ○ Map ○ Priority Queue ○ Point Set ○ Disjoint Set

Data Structure: ○ Array ○ Linked List ○ BST (Binary Search Tree)
○ LLRB (Left-leaning Red/Black) Tree ○ Hash Table ○ Binary Heap
○ k-d Tree ○ WeightedQuickUnion (with Path Compression)

Explanation:

Q6.3 WeirdSet

You're implementing `WeirdSet`, which implements the set ADT but allows users to mutate items after adding them to the set. As usual, the set should not allow users to add duplicate items, but users may mutate items already in the set such that they become duplicates. For example:

```
WeirdSet<MutableInt> set =
        new WeirdSet<>();          // {}
MutableInt a = new MutableInt(5);
set.add(a);                        // {5}
a.value = 6;                       // {6}
set.add(new MutableInt(6));        // {6} (duplicate, so no change)
MutableInt b = new MutableInt(5);
```

```
set.add(b);                          // {5, 6}
b.value = 6;                         // {6, 6}
```

What ADT is best for storing the items in your `WeirdSet` implementation?

ADT: ○ List ○ Set ○ Map ○ Priority Queue ○ Point Set ○ Disjoint Set

Data Structure: ○ Array ○ Linked List ○ BST (Binary Search Tree) ○ LLRB (Left-leaning Red/Black) Tree ○ Hash Table ○ Binary Heap ○ k-d Tree ○ WeightedQuickUnion (with Path Compression)

Explanation:

Q6.4 Reverse Image Searching

You're implementing a naive reverse-image search: you have an array of pictures (randomly ordered), each including an image (with arbitrary size) and a name. You want to store them in a data structure that can quickly look up the name of any picture from its exact image.

Your image and picture classes look like this:

```
public class Picture {
  public String name;
  public Image image;
}

public class Image {
  /** a flattened array of pixel values; has length = width*height
*/
  private int[] pixelValues;

  public int compareTo(Image o) {
    int lim = Math.min(this.pixelValues.length,
                       o.pixelValues.length);
    for (int k = 0; k < lim; k++) {
      int p1 = this.pixelValues[k];
      int p2 = o.pixelValues[k];
      if (c1 != c2) {
        return c1 - c2;
      }
    }
    return this.pixelValues.length - o.pixelValues.length;
  }
```

```
  public boolean equals(Object o) {
    if (o instanceOf Image) {
      Image o1 = (Image) o;
      return this.compareTo(o1) == 0;
    }
    return false;
  }

  public int hashCode() {
    int output = 0;
    for (int pixel : this.pixelValues) {
      output = 31 * output + pixel;
    }
    return output;
  }
}
```

What ADT is best for storing the items in your `WeirdSet` implementation?

ADT: ○ List ○ Set ○ Map ○ Priority Queue ○ Point Set ○ Disjoint Set

Data Structure: ○ Array ○ Linked List ○ BST (Binary Search Tree) ○ LLRB (Left-leaning Red/Black) Tree ○ Hash Table ○ Binary Heap ○ k-d Tree ○ WeightedQuickUnion (with Path Compression)

Explanation:

Q6.5 Vertex CLUSTERing

You are given a graph (represented as an adjacency list) and want to identify the number of CLUSTERS of vertices that are connected by at least one edge with a weight of 10 or less.

For example, given a graph containing nodes W, X, Y, and Z; and edges:

- (W, X) with weight 7
- (X, Y) with weight 8
- (Y, Z) with weight 12

The answer returned should be 1, because there is 1 CLUSTER matching the criteria (W, X, and Y).

ADT: ○ List ○ Set ○ Map ○ Priority Queue ○ Point Set ○ Disjoint Set

Data Structure: ○ Array ○ Linked List ○ BST (Binary Search Tree)
○ LLRB (Left-leaning Red/Black) Tree ○ Hash Table ○ Binary Heap
○ k-d Tree ○ WeightedQuickUnion (with Path Compression)

Explanation:

Q6.6 The Data Structural Lottery

You are managing a raffle. First, you print out 500 million tickets. Next, you offer the tickets for sale; customers may purchase more than one. You plan on randomly selecting the winning ticket's ID from the 500 million once all the tickets have been purchased.

Your raffle is so popular that many people — returning customers as well as new ones — continue to buy tickets every day!

After a while, you get curious about which customer is likely to win. Since you're the manager, you know which tickets each participant has purchased. You decide to keep track of the current top-100 most likely winners (tie-breaking doesn't matter). How would you do so efficiently?

ADT: ○ List ○ Set ○ Map ○ Priority Queue ○ Point Set ○ Disjoint Set

Data Structure: ○ Array ○ Linked List ○ BST (Binary Search Tree)
○ LLRB (Left-leaning Red/Black) Tree ○ Hash Table ○ Binary Heap
○ k-d Tree ○ WeightedQuickUnion (with Path Compression)

Explanation:
