CSE 373 Winter 2011 Final

1. Sorting

Given the following array of int elements:

```
// index        0  1  2  3   4  5   6  7  8  9
int numbers = {6, 7, 4, 8, 11, 1, 10, 3, 5, 9};
```

a. Show the state of the elements after 5 passes of the outermost loop of **selection sort**.

b. Show a trace of 2 levels deep of the **quicksort** algorithm. Show partitioning of the overall array, plus one level deep of the recursive calls' partitioning. Assume that the first element of a given range of the array is used as the pivot.

2. AVL Trees and Heaps

Given the following Integer elements:

```
6, 3, 2, 10, 8, 1, 11, 7, 5, 4, 9
```

Draw the tree that results when all of the above elements are added (in the given order) to each of the following initially empty data structures:

a. An **AVL tree**. (No removals are performed here.)

b. A minimum binary **heap**. After adding all the elements, perform **3 removes** on the heap.

Please show your work. You do not have to draw an entirely new tree after each element is added or removed, but since the final answer depends on every add/remove being done correctly, you may wish to show the tree at various important stages to help earn partial credit in case of an error.

3. Hashing

Simulate the behavior of a hash map as described in lecture. Assume the following:

- the hash table array has an initial size of 5
- the hash table uses **linear probing** for collision resolution
- the hash function returns the Integer key's `int` value, mod (remainder) the size of the table, plus any linear probing needed
- rehashing occurs **at the start of an add** where the load factor is 0.5 and causes the capacity of the hash map to double
- the table uses lazy removal and stores a special character "R" for an array entry that has been used but then later its entry is removed

Given the following set of calls:

```
HashMap map = new HashMap();

map.put(7, "Jessica");

map.put(34, "Tyler");

map.put(17, "Ryan");
```

```
map.put(15, "Tina");

map.put(84, "Saptarshi");

map.remove("Tyler");

map.put(7, "Meghan");

map.put(33, "Kona");

map.remove(17);

map.put(6, "Tina");

map.remove(84);

map.put(15, "Daisy");
```

Fill in the diagrams on the next page to show the state of the hash table right before rehashing occurs and the final state of the hash table. Leave a box empty if an array element is null or is unused. Also write the size, capacity, and load factor of the final hash table.

Initial hash table just before rehashing occurs:

```
    key             value
    +---------------+---------------+
0   |               |               |
    +---------------+---------------+
1   |               |               |
    +---------------+---------------+
2   |               |               |
    +---------------+---------------+
3   |               |               |
    +---------------+---------------+
4   |               |               |
    +---------------+---------------+
```

Final state of the hash table:

```
    key             value
    +---------------+---------------+
0   |               |               |          size:        _______________________
    +---------------+---------------+
1   |               |               |
    +---------------+---------------+
2   |               |               |          capacity:    _______________________
    +---------------+---------------+
3   |               |               |
    +---------------+---------------+
4   |               |               |          load factor: _______________________
    +---------------+---------------+
5   |               |               |
    +---------------+---------------+
6   |               |               |
    +---------------+---------------+
7   |               |               |
    +---------------+---------------+
8   |               |               |
    +---------------+---------------+
9   |               |               |
    +---------------+---------------+
```

4. Hash Set Implementation

Part A: Implementation

In lecture we wrote a class named `StringHashSet` that was an implementation of the Set ADT for `String`s. Our implementation used separate chaining as its collision resolution strategy. Add a method named `addAll` to this class that takes another `StringHashSet` as a parameter and adds all of the elements from the given `StringHashSet` to this `StringHashSet`. For example, the following sequence of calls below would result in `set` containing `{"Alan", "Meghan", "Martin"}`.

```
StringHashSet set = new StringHashSet();
set.add("Alan");
set.add("Meghan");
StringHashSet set2 = new StringHashSet();
set2.add("Alan");
set2.add("Martin");
set.addAll(set2);
```

`StringHashSet` stores its entries as `StringHashEntry` objects (shown below) and already has the methods `add`, `contains`, and `size` implemented (so use them!). If a bucket in `StringHashSet`'s `table` is empty, it contains a `null` value.

Hints: Recall that private fields are visible to their entire class, including other objects of that same class so it legal for one `StringHashSet` object to access fields of another. Our solution is 11 lines long.

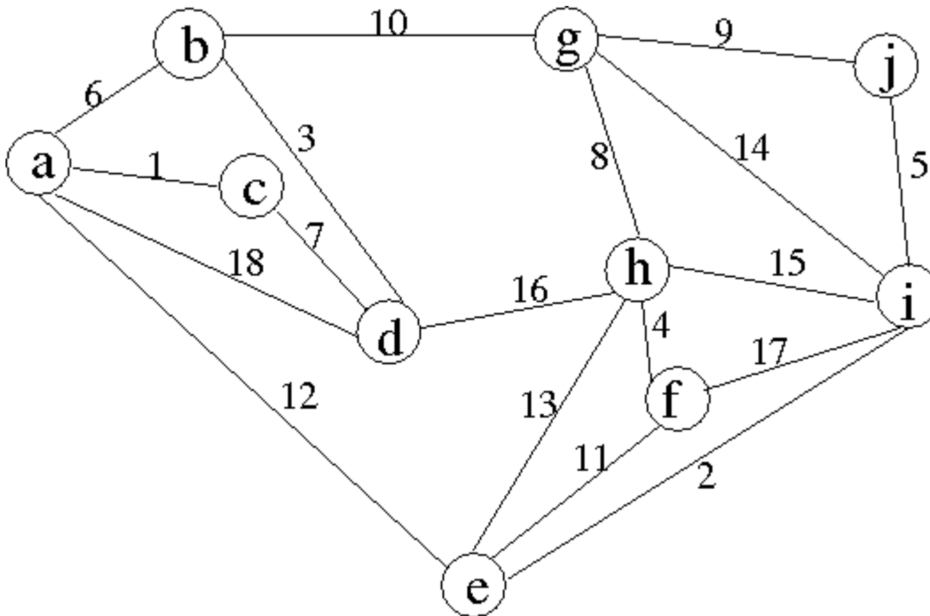
```
public class StringHashEntry {
    public String data;             // data stored at this node
    public StringHashEntry next;    // reference to the next entry
}

public class StringHashSet {
    private StringHashEntry[] table;
    private int size;

    // Adds the specified String to the set if it is not already present.
    // Returns true if the set did not already contain the String; false otherwise.
    public boolean add(String value) { … }

    // Returns true if the set contains the specified String; false otherwise.
    public boolean contains(String value) { … }

    // Returns the number of elements in the set (its cardinality).
    public int size() { … }

    // YOUR METHOD GOES HERE


   }
```

Part B: Analysis

Assuming that `StringHashSet` rehashes appropriately and a good hash function is used to distribute `String` values evenly throughout the table, what is the average case runtime of your `addAll` method. Give your answer in Big Oh notation and explain how you arrived at your answer. Be explicit about where variables used in your Big Oh come from.

5. Question corrupted (taken out)

6. Minimum Spanning Trees



a. Using Prim's Algorithm starting with vertex "a", list the vertices in the order they are added to the MST.

b. Using Kruskal's Algorithm, list the edges of the MST in the order that they are added.

7. Graph Implementation

Assume we modified `Graph.java` from Programming Project #4 to represent a directed graph (i.e. when an edge is added, it is only added to the given source vertex's neighbors list and not to the destination vertex neighbor's list).

Write a method named `numReachable` that could be added to the `Graph` class from Programming Project 4. The method accepts two parameters: a vertex and a number of steps. The method returns the count of how many vertices can be reached starting from the given vertex and walking up to the given number of steps. A given vertex can reach only itself in 0 steps; it can reach itself and its neighbors in 1 step; it can reach itself, its neighbors, and their neighbors in 2 steps; and so on.

For example, in the graph from the previous question, the colored nodes represent the reachable vertices from vertex "Marty" in 1 and 3 steps respectively; the calls of `graph.numReachable("Marty", 1)` and `graph.numReachable("Marty", 3)` would respectively return 4 and 10:

You may assume all arguments are valid. For full credit, your method's runtime should be O(V + E). You may not add any additional data fields or public methods to the graph class, but local variables and private methods are allowed. Hint: You may wish to use recursion to help you write this method.

API Reference for #7:

Map methods:

```
void clear()
boolean containsKey(Object key)
boolean containsValue(Object value)
boolean equals(Object o)
V get(Object key)
int hashCode()
boolean isEmpty()
Set keySet()
V put(K key, V value)
void putAll(Map t)
V remove(Object key)
int size()
Collection values()
```

Graph fields:

```
protected final Map<V, Map<V, EdgeInfo<E>>> adjacencyMap
protected final Map<V, VertexInfo<V>> vertexInfo
protected final List<E> edgeList
```

Graph methods:

```
void addEdge(V v1, V v2, E e)
void addEdge(V v1, V v2, E e, int weight)
void addVertex(V v)
void clearVertexInfo()
boolean containsEdge(V v1, V v2)
boolean containsVertex(V v)
E edge(V v1, V v2)
Collection edges()
int edgeWeight(V v1, V v2)
Collection neighbors(V v)
Collection vertices()
```

VertexInfo fields:

```
V v
V previous
boolean visited
int distance
```

8. Disjoint Sets

Trace the sequence of operations below, using a disjoint set with union by rank (without path compression). Assume there are initially 13 sets `{0},{1},{2},…,{12}`.

```
for (i = 0; i < 12; i += 2) {
    union(i, i + 1);
}
for (i = 0; i < 12; i += 4) {
    union(i, i + 2);
}
union(9, 1);
union(12, 6);
```

a. Draw the final forest of up-trees that result from the operations above.

b. Draw the new forest of up-trees that results from doing a `find(11)` with path compression on your forest of uptrees from (a).