# CSE 373, Winter 2011
## Final Key

## 1. Sorting (15 Points)

<table>
<thead>
<tr>
<th>Part</th>
<th>Answer</th>
</tr>
</thead>
</table>
| a    | `{6, 7, 4, 8, 11, 1, 10, 3, 5, 9}` 0  
      | `{1, 7, 4, 8, 11, 6, 10, 3, 5, 9}` 1  
      | `{1, 3, 4, 8, 11, 6, 10, 7, 5, 9}` 2  
      | `{1, 3, 4, 5, 11, 6, 10, 7, 8, 9}` 3  
      | `{1, 3, 4, 5, 6, 11, 10, 7, 8, 9}` 4  |
| b    | `{6, 7, 4, 8, 11, 1, 10, 3, 5, 9}  initial  
      | `{9, 7, 4, 8, 11, 1, 10, 3, 5, 6}  swap pivot to end  
      | `{5, 7, 4, 8, 11, 1, 10, 3, 9, 6}  ^             ^  
      | `{5, 3, 4, 8, 11, 1, 10, 7, 9, 6}  ^             ^  
      | `{5, 3, 4, 1, 11, 8, 10, 7, 9, 6}  ^^          
      | `{5, 3, 4, 1, 6, 8, 10, 7, 9, 11}  swap pivot back; end of initial partition  
      | `{5, 3, 4, 1}  {8, 10, 7, 9, 11}  swap pivots to end  
      | `{1, 3, 4, 5}  {11, 10, 7, 9, 8}  ^             ^             ^  
      | `{1, 3, 4, 5}  {7, 10, 11, 9, 8}  ^             ^  
      | `{1, 3, 4, 5}  {7, 8, 11, 9, 10}  overall  
      | `{1, 3, 4, 5, 6, 7, 8, 11, 9, 10}  overall |
## 2. AVL Trees and Heaps (15 Points)

<table>
<thead>
<tr>
<th>Part</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>![AVL Tree Diagram]</td>
</tr>
</tbody>
</table>
| a | imbalance at 6 (case 1): Right rotation  
   imbalance at 6 (case 3): Right-Left rotation  
   imbalance at 3 (case 3): Right-left rotation |
bubble up 3 to 6; bubble up 2 to 3; bubble up 1 to 2; bubble up 7 to 10; bubble up 5 to 6;
bubble up 4 to 5

remove 1:
  2
   / \        4  3
   / \ / \  6  5  9 11
  / \ / \ 10 7 8

bubble down 9 with 2; bubble down 9 with 3

remove 2:
  3
   / \         4  8
   / \ / \  6  5  9 11
  / \ 10 7

bubble down 8 with 3

remove 3:
  4
   / \   5  8
   / \ / \  6  7  9 11
  / 10

bubble down 7 with 4; bubble down 7 with 5
3. **Hashing (12 Points)**

**Initial hash table just before rehashing occurs:**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>34</td>
</tr>
</tbody>
</table>

**Final state of the hash table:**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
<th>size: 5</th>
<th>capacity: 10</th>
<th>load factor: .5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>33</td>
<td>Kona</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>R</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>Tina</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>R</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>Meghan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>17</td>
<td>Daisy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>Tina</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
map.put(7, "Jessica") --- AFTER: loadFactor = 1/5
map.put(34, "Tyler") --- AFTER: loadFactor = 2/5
map.put(17, "Ryan") --- DURING: collision with 7; AFTER: loadFactor = 3/5
map.put(15, "Tina") --- BEFORE: loadFactor > 0.5; DURING: REHASH! (collision on 17); AFTER loadFactor = 4/10
map.put(84, "Saptarshi") --- DURING: collision with 4 and 5; AFTER: loadFactor = 5/10
remove(34) --- AFTER: loadFactor = 4/10
map.put(7, "Meghan") --- DURING: the key 7 is already in the table, so value "Jessica" gets overwritten; AFTER: loadFactor = 4/10
map.put(33, "Kona") --- AFTER: loadFactor = 5/10
map.remove(8) --- DURING: no effect, key is not in table; loadFactor = 5/10
map.put(6, "Tina") --- BEFORE: assuming no rehashing DURING: collision with 84, 7, 17; AFTER: loadFactor = 6/10
map.remove(84) --- AFTER: loadFactor = 5/10
map.put(17, "Daisy") --- DURING: the key 17 is already in the table, so value "Ryan" gets overwritten; AFTER: loadFactor = 5/10
4. HashSet Implementation (12 Points)

(a)

```java
public void addAll(StringHashSet otherSet) {
    for (int i = 0; i < otherSet.table.length; i++) {
        StringHashEntry temp = otherSet.table[i];
        while (temp != null) {
            if (!this.contains(temp.data)) {
                this.add(temp.data);
            }
            temp = temp.next;
        }
    }
}
```

(b) O(N) based on the size of the "other" hash table
5. Graphs (12 points)

<table>
<thead>
<tr>
<th>Part</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>directed</td>
</tr>
<tr>
<td>b</td>
<td>unweighted</td>
</tr>
<tr>
<td>c</td>
<td>unconnected (counterexample: Jennifer can't reach anybody) (Technically, the right way to describe a graph like this is &quot;weakly connected&quot; because the vertices are all touching, but not necessarily by edges in the right direction.)</td>
</tr>
<tr>
<td>d</td>
<td>cyclic (example cycle: Carmelita, Marty, Steve, Carmelita)</td>
</tr>
<tr>
<td>e</td>
<td>in-degree of Louann: 2 (from Donald, Carmelita); out-degree of Louann: 4 (to Isabelle, Kris, Marty, Moshe)</td>
</tr>
</tbody>
</table>
### 6. Minimum Spanning Trees (12 points)

<table>
<thead>
<tr>
<th>Part</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a, c, b, d, g, h, f, j, i, e edges: AC = 1, AB = 6, BD = 3, BG = 10, GH = 8, HF = 4, GJ = 9, JI = 5, IE = 2</td>
</tr>
<tr>
<td>b</td>
<td>AC = 1, EI = 2, BD = 3, HF = 4, JI = 5, AB = 6, GH = 8, GJ = 9, BG = 10</td>
</tr>
</tbody>
</table>
7. Graph Implementation (12 points)

The first four solutions shown are correct solutions. The final two solutions shown were also graded as "correct" but do not always produce a correct answer.

// Solution 1: inexplicit base case
public int numReachable(V v, int steps) {
    this.clearVertexInfo();
    return this.numReachableHelper(v, steps);
}

private int numReachableHelper(V v, int steps) {
    int count = 0;
    if (!this.vertexInfo.get(v).visited) {
        this.vertexInfo.get(v).visited = true;
        count = 1;     // the vertex V itself
    }
    if (steps > 0) {   // recursive case
        for (V n : this.neighbors(v)) {
            count += this.numReachableHelper(n, steps - 1);
        }
    }
    return count;
}

// Solution 2: explicit base case
public int numReachable(V v, int steps) {
    this.clearVertexInfo();
    return this.numReachableHelper(v, steps);
}

private int numReachableHelper(V v, int steps) {
    if (steps == 0) {
        if (!this.vertexInfo.get(v).visited) {
            this.vertexInfo.get(v).visited = true;
            return 1;
        } else {
            return 0;
        }
    } else {    // recursive case
        int count;
        if (!this.vertexInfo.get(v).visited) {
            this.vertexInfo.get(v).visited = true;
            count = 1;
        } else {
            count = 0;
        }
        for (V n : this.neighbors(v)) {
            count += this.numReachableHelper(n, steps - 1);
        }
        return count;
    }
}
// Solution 3: using Set as a supporting data structure
public int numReachable(V v, int steps) {
    Set<V> visited = new HashSet<V>();
    visited.add(v);
    numReachable(v, steps, visited);
    return visited.size();
}

private void numReachable(V v, int steps, Set<V> visited) {
    if (steps > 0) {
        for (V n : neighbors(v)) {
            visited.add(n);
            numReachable(n, steps - 1, visited);
        }
    }
}

// Solution 4: counting visited solution
public int numReachable(V v, int steps) {
    this.clearVertexInfo();
    numReachableHelper(v, steps);
    int n = 0;
    for (VertexInfo<V> i : vertexInfo.values()) {
        if (i.visited) {
            n++;
        }
    }
    return n;
}

private void numReachableHelper(V v, int steps) {
    vertexInfo.get(v).visited = true;
    if (steps > 0) {
        for (V n : this.neighbors(v)) {
            numReachableHelper(n, steps - 1);
        }
    }
}

// Solution 5: Nearly correct solution, but stops short on neighbors of a
// visited vertex with that can be reached through multiple
// paths and a longer path is examined before a shorter one
public int numReachable(V v, int steps) {
    this.clearVertexInfo();
    return this.numReachableHelper(v, steps);
}

private int numReachableHelper(V v, int steps) {
    this.vertexInfo.get(v).visited = true;
    int count = 1;     // the vertex V itself
    if (steps > 0) {   // recursive case
        for (V n : this.neighbors(v)) {
            if (!this.vertexInfo.get(n).visited) {
                count += this.numReachableHelper(n, steps - 1);
            }
        }
    }
    return count;
public int numReachable(V v, int steps) {
    this.clearVertexInfo();
    return this.numReachableHelper(v, steps);
}

private int numReachableHelper(V v, int steps) {
    if (steps == 0 || this.vertexInfo.get(v).visited) {
        return 0; // base case
    } else {
        int count = 1; // the vertex V itself
        this.vertexInfo.get(v).visited = true;

        for (V n : this.neighbors(v)) {
            count += this.numReachableHelper(n, steps - 1);
        }

        return count;
    }
}
8. Disjoint Sets (10 points)

<table>
<thead>
<tr>
<th>Part</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>initial disjoint set</td>
</tr>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12</td>
</tr>
<tr>
<td></td>
<td>after first for loop (i.e. union(0, 1), union(2, 3), union(4, 5), union(6, 7), union(8, 9), union(10, 11)):</td>
</tr>
<tr>
<td></td>
<td>0 2 4 6 8 10 12</td>
</tr>
<tr>
<td></td>
<td>1 3 5 7 9 11</td>
</tr>
<tr>
<td></td>
<td>after second for loop (i.e. union(0, 2), union(4, 6), union(8, 10)):</td>
</tr>
<tr>
<td></td>
<td>0 4 8 12</td>
</tr>
<tr>
<td></td>
<td>1 2 5 6 9 10</td>
</tr>
<tr>
<td></td>
<td>3 7 11</td>
</tr>
<tr>
<td></td>
<td>after union(9, 1):</td>
</tr>
<tr>
<td></td>
<td>0 4 12</td>
</tr>
<tr>
<td></td>
<td>1 2 8 5 6</td>
</tr>
<tr>
<td></td>
<td>3 9 10 7</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>after union(12, 6):</td>
</tr>
<tr>
<td></td>
<td>0 4</td>
</tr>
<tr>
<td></td>
<td>1 2 8 5 6 12</td>
</tr>
<tr>
<td></td>
<td>3 9 10 7</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>

|      | 0 4 |
|      | 1 2 8 10 11 5 6 12 |
|      | 3 9 7 |