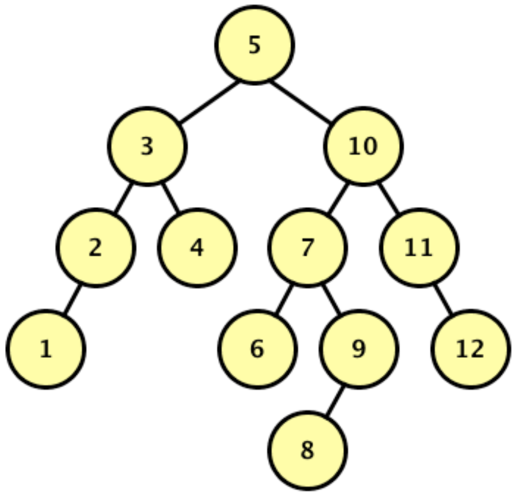
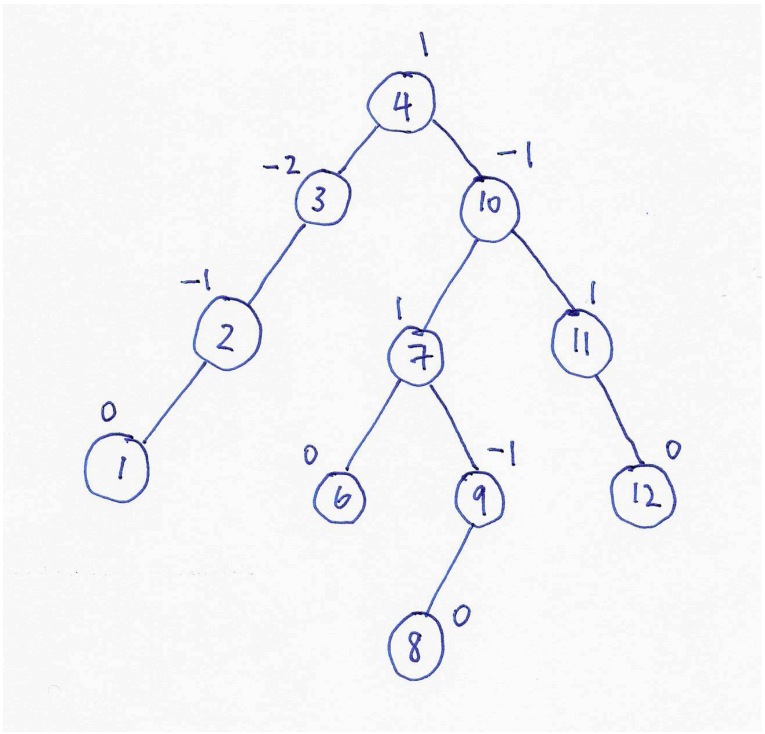
CSE 373 Spring 2012 Final Exam Solution

1. AVL Trees (10 Points)

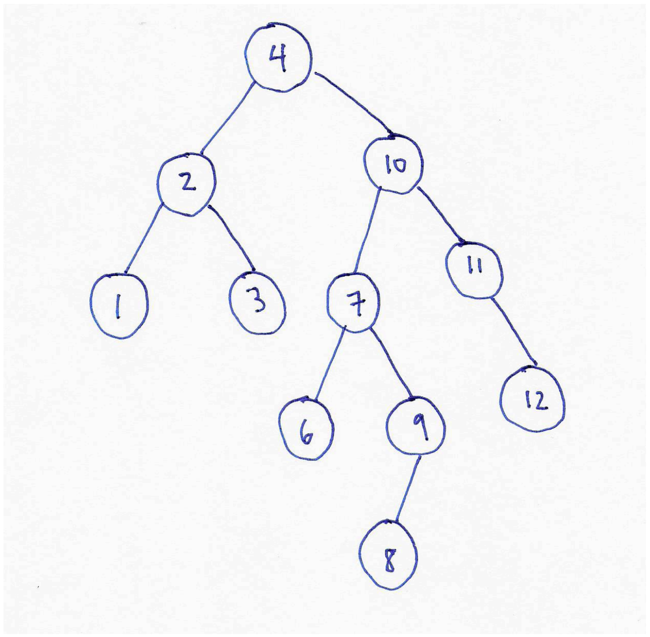
Given the following AVL Tree:



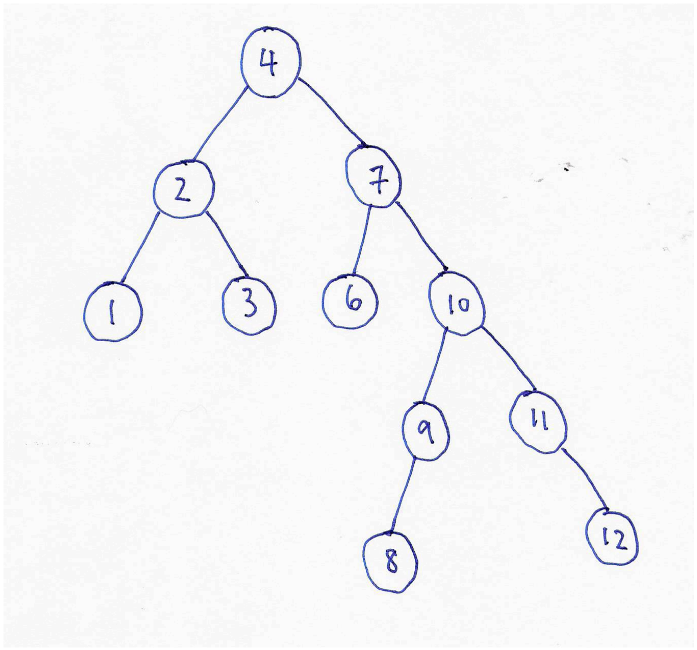
(a) Draw the resulting **BST** after **5** is removed, but *before* any rebalancing takes place. Label each node in the resulting tree with its **balance factor**. Replace a node with both children using an appropriate value from the node's **left** child.

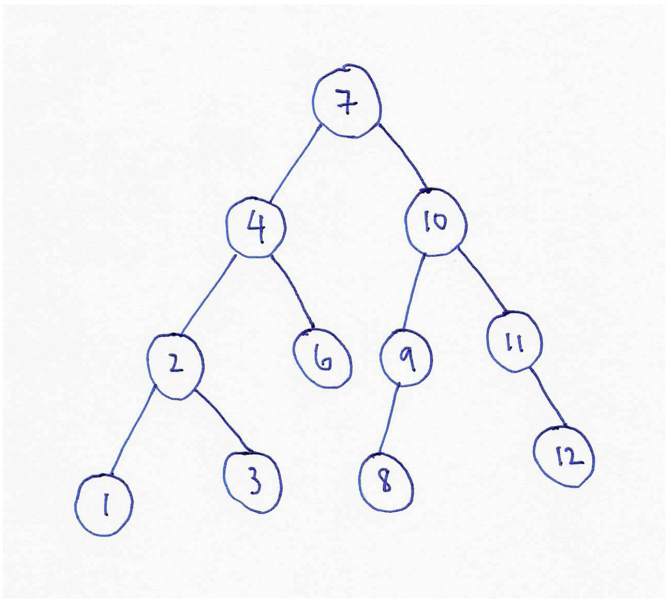


(b) Now rebalance the tree that results from (a). **Draw a new tree** for each rotation that occurs when rebalancing the AVL Tree (you only need to draw one tree that results from an RL or LR rotation). You do not need to label these trees with balance factors.



We show the intermediate rotation from the RL rotation along with the final answer below:





2. Hashing (9 Points)

Simulate the behavior of a **hash set** storing integers given the following conditions:

- The initial array length is **5**.
- The set uses **quadratic probing** for collision resolution.
- The hash function uses the `int` value (plus any probing needed) mod the size of the table.
- When the load factor reaches or exceeds 0.5, the table enlarges to double capacity and rehashes values stored at smaller indices first.
- An insertion **fails** if more than half of the buckets are tried or if the **properties of a set** are violated.

Given the following code:

```
Set<Integer> set = new HashSet<Integer>(5);
set.add(86);
set.add(76);
set.add(16);
set.add(66);
set.add(26);
set.add(76);
```

(Show your work on the next page to get partial credit in case you make a mistake somewhere.)

(a) What are the indices of the values in the final hash set?

16: **17** 26: **6** 66: **7** 76: **16** 86: **10**

(b) Did any values fail to be inserted? If so, which ones and why did the insertion fail?

The second 76. It is a duplicate value.

(c) What is the size of the final `set`? **5**

(d) What is the capacity of the final `set`? **20**

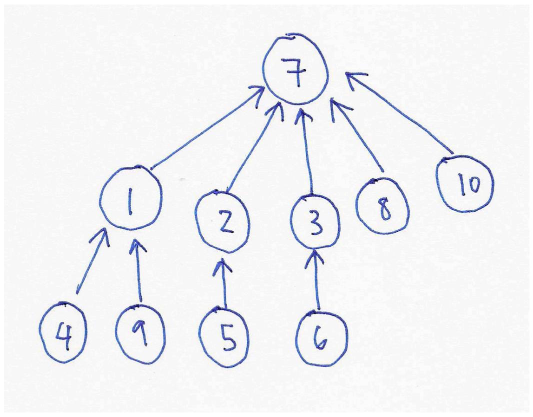
(e) What is the load factor of the final `set`? **0.25**

3. Disjoint Sets (9 Points)

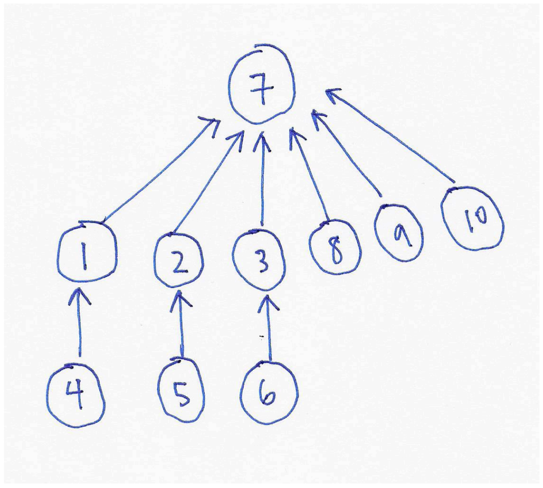
Trace the sequence of operations below, using a disjoint set with union by size (without path compression). If there is a tie when performing a union, the tree with the smaller-valued root should be the root of the union. Assume there are initially 10 sets `{1},{2},{3},…,{10}`.

```
union(7, 8);
union(3, 6);
union(2, 5);
union(7, 10);
union(1, 4);
union(3, 7);
union(2, 7);
union(1, 9);
x = find(7);
y = find(9);
union(x, y);
```

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



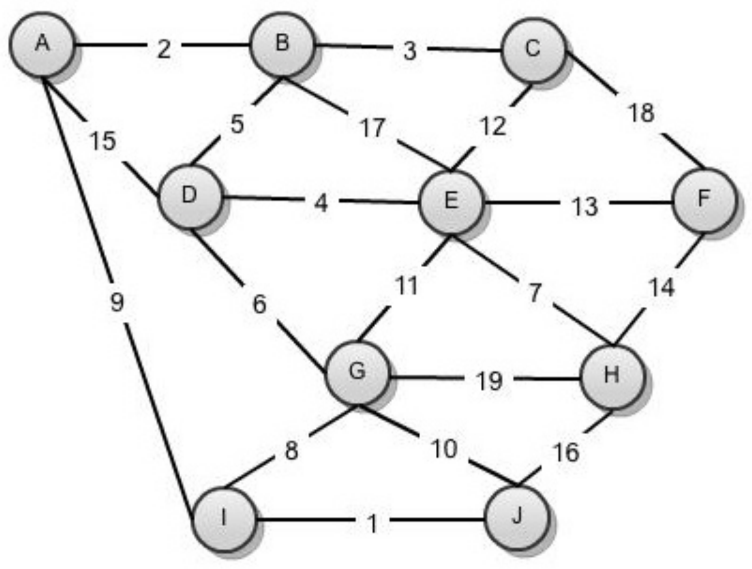
(b) Draw the new forest of up-trees that results from doing a `find(9)` with path compression on your forest of up-trees from (a).



4. Maximum Spanning Trees (10 Points)

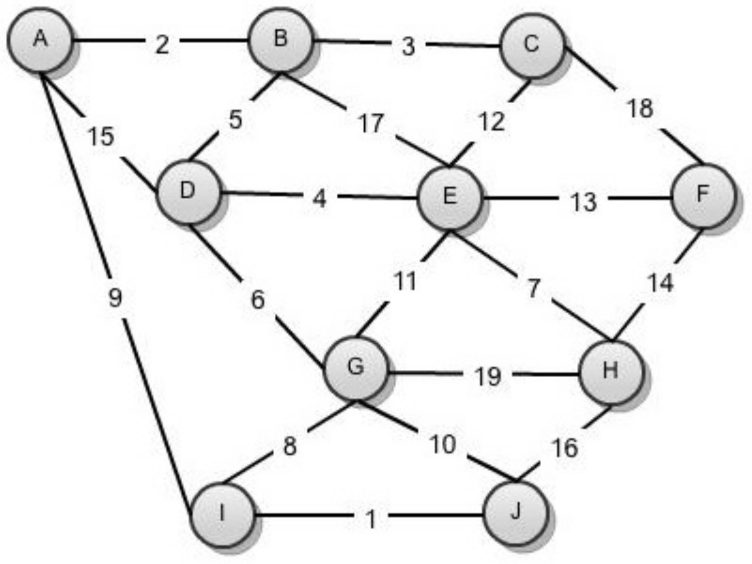
For this problem, you will be finding the **maximum** spanning tree. You can still use Prim's and Kruskal's algorithms if you just switch "smallest" with "biggest" when examining edges.

(a) Using Prim's algorithm starting with vertex "A", list the vertices of the graph below in the order they are added to the **maximum** spanning tree.



A, D, I, G, H, J, F, C, E, B

(b) Using Kruskal's algorithm, list the edges of the **maximum** spanning tree of the graph below in the order that they are added.



(G, H)

(C, F)

(B, E)

(H, J)

(A, D)

(F, H)

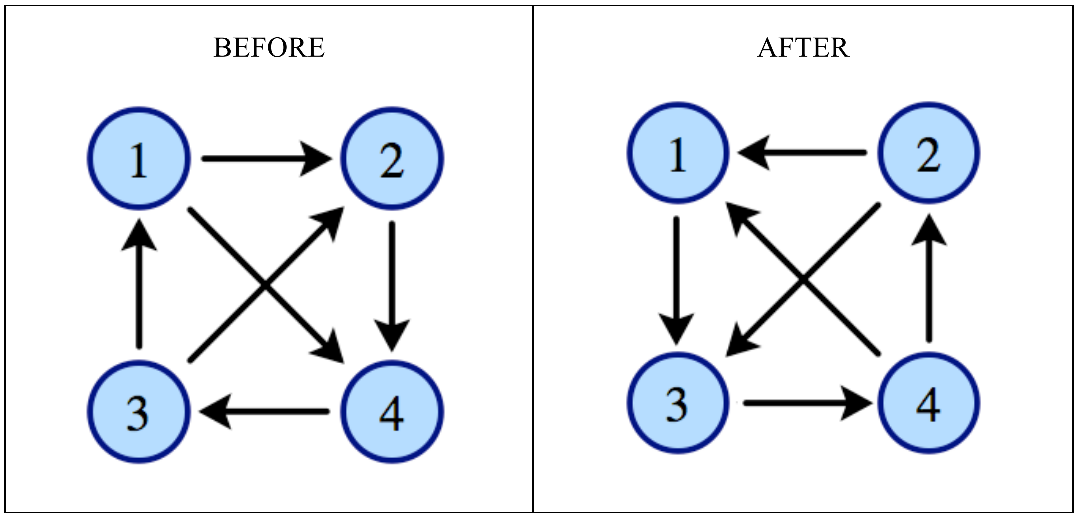
(E, F)

(A, I)

(G, I)

5. Graphs (15 Points)

(a) Suppose that the `Graph` class in Homework #7 (Kevin Bacon) was a **directed** graph. Write a method named `reverse` that would replace all edges (*v*, *w*) with (*w*, *v*). The picture below shows a graph before and after a call to `reverse`.



```
public void reverse() {
   Map<V, Map<V, EdgeInfo<E>>> newAdjacencyMap
      = new HashMap<V, Map<V, EdgeInfo<E>>>();

   for (V source : vertices()) {
      for (V target : adjacencyMap.get(source).keySet()) {
         EdgeInfo<E> e = adjacencyMap.get(source).get(target);

         if (!newAdjacencyMap.containsKey(target)) {
            newAdjacencyMap.put(target, new HashMap<V, EdgeInfo<E>>());
         }

         Map<V, EdgeInfo<E>> targetMap = newAdjacencyMap.get(target);
         targetMap.put(source, e);
      }

   }

   adjacencyMap = newAdjacencyMap;
}
```

(b) What is the running time of your method? Explain your answer.

The method processes each edge exactly once while accessing each vertex on the vertex list: O(|V| + |E|)

O(|E|) is also acceptable.

6. What's the Point? (4 Points)

(a) What is the purpose of AVL trees?

AVL trees are binary search trees that are guaranteed to be balanced. They guarantee O(log n) operations on the tree.

(b) What is the purpose of heaps?

Heaps are how most priority queues are implemented. They allow you to quickly find the minimum value from the values stored in the heap without costly adds.

7. More Graphs (7 Points)

Assume a graph that represents all the web pages (vertices) and the hyperlinks (edges) between them on the Internet.

Is the graph…

(a) **directed**

(b) **cyclic**

(c) **unweighted**

(d) What kinds of web pages have a high in-degree?

Examples:

- Popular web sites like YouTube
- Popular news articles that other pages link to

(e) What kinds of web pages have a high out-degree?

Examples:

- Directory pages like Yahoo!
- Personalized pages like MyUW
- News sites like New York Times

8. Sorting (4 Points)

In the `Arrays` class, there is a static sort method:

```
Arrays.sort(int[] a)
```

The method sorts the specified array into ascending numerical order. The sorting algorithm is a tuned quicksort.

Quicksort has a worst case running time of $O(n^2)$ whereas mergesort has a worst case running time of O(n log n), but quicksort is generally chosen over mergesort. Why? Explain your answer.

Quicksort generally runs in O(n log n) time. The constant factor is better than mergesort, because it does the sort with simple swaps, whereas mergesort has to create a new array to store the merged data.

9. Hash Functions (9 Points)

Override the `hashCode` method of the following (poorly-designed) class. Your method should follow proper hashing principles and minimize collisions. You may assume that any non-primitive field in `RentalCar` has an `equals` methods and a good implementation of `hashCode`.

```java
public class RentalCar {
   private int year;
   private String color;
   private String model;
   private Person renter;  // null if available

   ...

   public boolean equals(Object other) {
      if (!(other instanceof RentalCar)) {
         return false;
      }

      RentalCar o = (RentalCar)other;

      return
         (year / 10 == o.year / 10) // car model year in same decade
         && model.equals(o.model)
         && ((renter == null && o.renter == null) ||
             (renter != null && renter.equals(o.renter)));

   }


   public int hashCode() {
         int result = 17;
         result = 37 * result + (year / 10);
         result = 37 * result + model.hashCode();

         if (renter == null) {
            result = 37 * result + 1;
         } else {
            result = 37 * result + renter.hashCode();
         }

         return result;
   }

}
```

10. B-Trees (9 Points)

(a) What is the purpose of B-Trees?

B-Trees are trees that take into account that most data is not stored in main memory. It's designed to minimize the number of disk accesses that must be performed when operating on the tree.

(b) Describe how to compute M and L in terms of variables and constraints. Define your variables.

For example:

A = number of apples

B = weight of bananas

C = size of carrots

Find smallest integer M where $M \geq 10 \cdot A + \frac{B}{C}$. Equivalently, $M = \left\lceil 10 \cdot A + \frac{B}{C} \right\rceil$

B = size of page block

K = size of key

P = size of branch pointer

R = size of record (including key)

$$M = \left\lfloor \frac{B + K}{K + P} \right\rfloor$$

$$L = \left\lfloor \frac{B}{R} \right\rfloor$$

11. Miscellaneous (10 Points)

(a) Write a method that given an array of integers will print out a list of duplicates (separated by spaces). The contents of the array do not have to be preserved. The duplicates should appear exactly once, though they do not have to be in any particular order. It is OK to have an extra space at the end of the output. The method may use any auxiliary data structures.

Array	Possible Output
{ 1, 2, 3, 4, 1, 2, 3 }	1 2 3
{ 1, 2, 3, 4, 4, 3, 2, 1 }	2 4 3 1
{ 1, 2, 3, 4 }	
{ 1, 1, 1, 2, 2, 1, 1, 1 }	2 1

```
public static void printDuplicates(int[] array) {
      Arrays.sort(array);

      boolean foundDuplicate = false;
      int lastDuplicate = 0;
      for (int i = 1; i < array.length; i++) {
         if (array[i] != array[i-1]) {
            if (foundDuplicate) {
              System.out.print(lastDuplicate + " ");
            }
            foundDuplicate = false;
         } else {
            lastDuplicate = array[i];
            foundDuplicate = true;
        }
      }

      if (foundDuplicate) {
         System.out.print(lastDuplicate);
      }

      System.out.println();
   }

   public static void printDuplicates(int[] array) {
      Map<Integer, Integer> counts = new HashMap<Integer, Integer>();
      for (int num : array) {
         if (counts.containsKey(num)) {
            counts.put(num, counts.get(num)+1);
         } else {
            counts.put(num, 1);
         }
      }

      for (int num : counts.keySet()) {
         if (counts.get(num) > 1) {
            System.out.print(num + " ");
         }
      }

      System.out.println();
   }
```

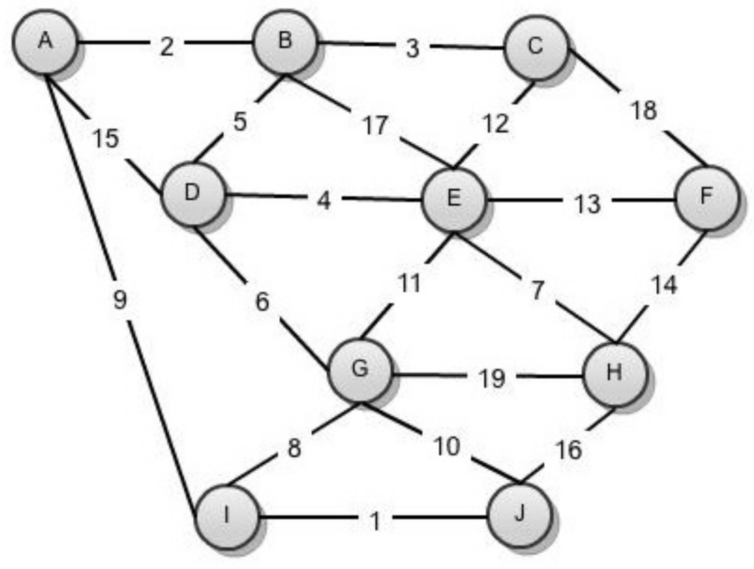
(b) What is the runtime of your implementation? Explain your answer.

The runtime of the first method that uses sorting is dominated by the sort and takes O(n log n) time. Finding the duplicates from a sorted list takes linear time and is subsumed by the runtime of the sort.

The runtime of the second is linear. It goes over the array and accesses the hash per element. Accessing the hash table takes constant time, so the total running time is linear. Printing the elements requires going over the hash values and only printing values that have a count greater than one. That also takes only linear time. O(n)

12. Graph Search (4 Points)

Given the following graph:



For the following problems, assume alphabetical edge order.

(a) Write (in order) the list of vertices that would be visited by running a **breadth**-first search (BFS) starting from G

G D E H I J A B C F

(b) Write (in order) the list of vertices that would be visited by running a **depth**-first search (DFS) starting from G.

G D A B C E F H J I

Grading criteria:

1. AVL Trees

(a) 4 points
- 2 element correctly removed
- 2 balance factors

(b) 6 points
- 3 attempts rotations to get final AVL tree
- 3 correct

2. Hashing
- 1 point per blank/answer
- (e) is ok if it equals (c) / (d)
- -3 if don’t rehash in the right order

3. Disjoint Sets

(a) 6 points
- -3 per mistake

(b) 3 points (all or nothing)

4. Maximum Spanning Tree
- 5 points each (-3 points per swap/extra/missing)

5. Graphs

(a) 12 points
- 1 header
- 3 loops over all edges
  - 1 attempt
  - 2 correct
- 2 creates new map for new source (old target)
- 2 places reversed (w, v) in map
- 1 sets old adjacency map properly
- 3 correct

If not modifying adjacency map (max 8 points):
- 1 header
- 3 loops over all edges
  - 1 attempt
  - 2 correct
- 2 places reversed edges
- 2 removed original edge

(b) 3 points (1 point if bad answer)

6. What’s the Point?
- 2 points per section (1 incomplete answer)

7. More Graphs
- 1 point for (a), (b), (c)
- 2 points each for (d), (e) (answer must be reasonable)

8. Sorting
- 1 point if bad answer
- 2 points if partial answer

9. Hash Function
- 1 correctly ignores color
- 3 correct polynomial accumulation
  - 1 attempt
  - 2 correct
- 1 handles decade
- 1 handles model
- 2 handles null renter
- 1 handles non-null renter

10. B-Trees

(a) 3 points (1 point if bad answer)

(b) 6 points
- 2 states correct variables (-1 per missing variable)
- 3 correct equation for M
- 1 correct equation for L
- -1 for missing floor

11. Miscellaneous

(a) 7 points
- 1 header
- 2 finds all duplicates (no false positive)
- 2 find duplicates exactly once
- 2 correct

(b) 3 points (1 point if bad answer)

12. Graph Search

2 points each (all or nothing)