University of Maryland College Park
Dept of Computer Science
CMSC132 Fall 2018
Exam #3

FIRSTNAME, LASTNAME (PRINT IN UPPERCASE):

STUDENT ID (e.g. 123456789):

Instructions

- Please print your answers and use a pencil.
- Do not remove the staple from the exam. Removing it will interfere with the Gradescope scanning process.
- To make sure Gradescope can recognize your exam, print your name, write your directory id at the bottom of pages with the text DirectoryId, provide answers in the rectangular areas provided, and do not remove any exam pages. Even if you use the provided extra pages for scratch work, they must be returned with the rest of the exam.
- This exam is a closed-book, closed-notes exam, with a duration of 50 minutes and 200 total points.
- Your code must be efficient.
- You don't need to use meaningful variable names; however, we expect good indentation.

Grader Use Only

#1	Problem #1 (Algorithmic Complexity)	30	
#2	Problem #2 (Heaps)	16	
#3	Problem #3 (Hashing)	15	
#4	Problem #4 (Linear Data Structures)	65	
#5	Problem #5 (Trees)	74	
Total	Total	200	

1

Problem #1 (Algorithmic Complexity)

1. (20 pts) For the following problems you need to provide the asymptotic complexity using Big O notation. In addition, you need to **identify the critical section (circle it)** and the time function (Time → below). Here is an example:

```
for (j = 1; j <= n; j++) {
    System.out.println(j);
}
System.out.println("Goodbye");
```

Time → $n + 1$
Big O → $O(n)$

   a. (10 pts)

```
for (k = 1; k <= n; k++) {
    System.out.println(k);
    System.out.println(k * k);
    for (t = 1; t <= n / 2; t++) {
        System.out.println(k * t);
    }
}
```

   Time →

   Big O →

   b. (10 pts)

```
for (i = 1; i <= n; i *= 2) {
    for (k = 1; k <= n - 10; k++) {
        System.out.println(k);
    }
    System.out.println(i);
}
```

   Time →

   Big O →

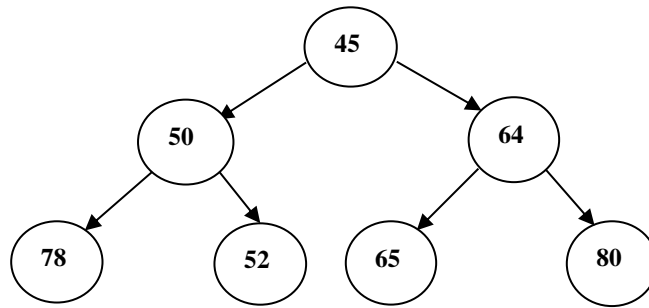2. (6 pts) List the following Big O expressions in order of asymptotic complexity (lowest complexity first).

$$O(n\log(n)) \quad O(k^n) \quad O(n^n) \quad O(n^3) \quad O(n^k)$$

3. (4 pts) Indicate the complexity (Big O) for an algorithm whose running time does not change when input size doubles.

DirectoryId:

3

Problem #2 (Heaps)

Use the following heap to answer the questions that follow.

1. (8 pts) Draw the heap (as a tree) that would result from inserting **32** in the above heap.

2. (8 pts) Draw the heap (as a tree) that would result by deleting **45 from the original heap** (not the heap from step 1.)

Problem #3 (Hashing)

1. (3 pts) Does the default implementation of the equals and hashCode methods for a Java class satisfy the Java Hash Code contract? Yes / No

2. (3 pts) Returning the integer value 10 as the implementation of the hashCode() method:

   a. It is correct, but it is inefficient as it can lead to many collisions.
   b. It is correct, and efficient.
   c. It is incorrect.
   d. None of the above.

3. (3 pts) Which of the following represents the optimal performance for hashing?

   a. O(n)
   b. O(1)
   c. O(log(n))
   d. None of the above.

4. (3 pts) Which of the following is a collision handling strategy that looks for an unused entry in the table (wrapping around if necessary)?

   a. Open Addressing
   b. Separate Chaining
   c. Deep Searching
   d. None of the above.

5. (3 pts) Linear probing can cause:

   a. Primary Clustering
   b. Secondary Clustering
   c. Null Clustering
   d. None of the above.

DirectoryId:

Problem #4 (Linear Data Structures)

Use the following classes to implement the methods below. You may not add any instance variables nor static variables to either class, you may not add any methods to the Node class, and you may not use the Java API LinkedList class. A cheat sheet with set and map methods can be found on the next page. Use **compareTo** and not **equals** for comparisons.

```
public class LinkedList<T extends Comparable<T>> {
      private class Node {
            private T data;
            private Node next;
            private Node(T data) { this.data = data; next = null; }
      }
      private Node head;
      public LinkedList() { head = null; }
}
```

1. (37 pts) Provide a **NON-RECURSIVE** implementation for the **getCount** method below. The method returns a TreeMap with the number of instances found of each unique **data** element in the list. An empty map should be returned if the list is empty.

```
public TreeMap<T, Integer> getCount()
```

2. (28 pts) Provide a **RECURSIVE** implementation for the **removeLastIfEqualFirst** method. The method removes the **last element/node from the list** if the **data** component of the last element from the list is the same as the **data** component of the first element. **If you use any iteration statement (e.g., while loop, do while, for loop) you will get 0 credit.** No processing will take place if the list is empty or it only has one element. You may only add one auxiliary method.

```
public void removeLastIfEqualFirst(){




```

Set methods

boolean **add**(E e)	boolean **remove**(Object o)
boolean **contains**(Object o)	void **clear**()
boolean **isEmpty**()	int **size**()

Map Methods

V **put**(K key, V value)	V **remove**(Object key)
V **get**(Object key)	void **clear**()
boolean **isEmpty**()	int **size**()

DirectoryId:

Problem #5 (Trees)

Implement the methods below based on the following Java class definitions. You may not add any instance variables nor static variables to either class and you may not add any methods to the Node class. Your solutions must be **RECURSIVE** and you may only add one auxiliary method. **If you use any iteration statement (e.g., while loop, do while, for loop) you will get 0 credit.** Use **compareTo** and not **equals** for comparisons.

```
public class BinarySearchTree<K extends Comparable<K>, V> {
      private class Node {
            private K key;
            private V data;
            private Node left, right;
            public Node(K key, V data) { this.key = key; this.data = data;}
      }
      private Node root;
}
```

1. (44 pts) Implement the **RECURSIVE** method **getKeysOnPathToElem** that adds to the ArrayList parameter the **key** component of nodes in the path traversed to determine whether the tree has an element with a **target** (parameter) key value. The method returns the number of nodes that were visited. You can assume the **answer** ArrayList parameter is initially empty.

```
public int getKeysOnPathToElem(K target, ArrayList<K> answer){
```

2. (30 pts) Implement the **RECURSIVE** method **removeLeaves** that removes the leaf nodes from the tree.

```java
public void removeLeaves() {
```

LAST PAGE