



University of Maryland College Park

Department of Computer Science

CMSC132 Fall 2019

Exam #3 Key

FIRSTNAME, LASTNAME (PRINT IN UPPERCASE):

STUDENT ID (e.g. 123456789):

Instructions

- Please print your answers and use a pencil.
- This exam is a closed-book, closed-notes exam with a duration of 50 minutes and 200 total points.
- **Do not remove the exam's staple.** Removing it will interfere with the scanning process (even if you staple the exam again).
- Write your directory id (e.g., terps1, not UID) at the bottom of pages with **DirectoryId**.
- Provide answers in the rectangular areas.
- Do not remove any exam pages. Even if you don't use the extra pages for scratch work, return them with the rest of the exam.
- Your code must be efficient and as short as possible.
- If you continue a problem on the extra page(s) provided, make a note on the particular problem.
- For multiple choice questions you can assume only one answer is expected, unless stated otherwise.
- You don't need to use meaningful variable names; however, we expect good indentation.
- **You must write your name and id at this point (we will not wait for you after time is up).**
- You must stop writing once time is up.

Grader Use Only

#1	Problem #1 (Algorithmic Complexity)	40	
#2	Problem #2 (Hashing)	24	
#3	Problem #3 (Linear Data Structures)	136	
Total	Total	200	

Problem #1 (Algorithmic Complexity)

1. (30 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 (int k = 1; k <= n; k++) {  
    if (k == n / 2) {  
        System.out.println(k);  
        break;  
    }  
}
```

Time → $n / 2$

Big O → $O(n)$

Circling critical section

- b. (10 pts)

```
for (int k = 1; k <= (n * n) / 2; k++) {  
    for (int m = 400; m <= 5000; m += 5000) {  
        System.out.println(m);  
    }  
}
```

Time → $n^2/2$

Big O → $O(n^2)$

Circling critical section

- c. (10 pts)

```
for (int k = 1; k <= n; k *= 2) {  
    for (int m = 1; m <= n / 2; m++) {  
        System.out.println(m);  
    }  
    for (int m = 1; m <= n / 4; m++) {  
        System.out.println(m);  
    }  
}
```

Time → $(\log(n) + 1)(n/2 + n/4)$

Big O → $n \log(n)$

Circle first critical section

Circle second critical section

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

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

Answer: $O(\log(n))$, $O(n \log(n))$, $O(n^3)$, $O(n^n)$

3. (2 pts) Indicate the complexity (Big O) for an algorithm with the following running times:

<u>Size(n)</u>	<u>Running Time</u>
----------------	---------------------

2	8
4	32
8	128

Answer: $O(n^2)$

4. (2 pts) Indicate the complexity (Big O) for an algorithm whose running time does not change when input size doubles.

Answer: $O(1)$

5. (4 pts) Give the asymptotic bound of the following functions:

- | | |
|--------------------------|-------------------------|
| a. $\log(n) + 10n^3 + n$ | $f(n) = O(n^3)$ (2 pts) |
| b. $n + n^4 + 7$ | $f(n) = O(n^4)$ (2 pts) |

Problem #2 (Hashing)

1. (3 pts) Which of the following are true assuming a class satisfies the Java Hash Code contract and two different objects have different hashCode() values? Circle all that apply.

- a. Computation of hashCode() will be faster.
- b. Collisions will be reduced when using the hashCode() method.
- c. Determining whether two objects are equal will take a longer time than usual.
- d. None of the above.

Answer: b.

2. (3 pts) Which properties are associated with a good hash function? Circle all that apply.

- a. Relies on the compareTo method.
- b. The function is slow to compute a hash value, but the probability of collisions is guaranteed to be zero.
- c. The function is fast to compute a hash value, with low probability of collisions, but not guaranteed to be zero.
- d. None of the above.

Answer: c.

3. (3 pts) In linear probing an entry that is removed is updated to the "Removed" state instead of the "NeverUsed" state because:

- a. The hash table will only store half of the possible items otherwise.
- b. An entry that collided might not be found otherwise.
- c. We will not be able to add any other additional entries otherwise.
- d. None of the above.

Answer: b.

4. (3 pts) In hashing the load factor is: (Circle all that apply)

- a. A measure of the cost of collision resolution.
- b. Number of entries in hash table / size of the table.
- c. Number of entries in hash table / average size of the key.
- d. None of the above.

Answer: a. and b.

5. (12 pts) The Elevator class is defined as follows:

```
public class Elevator {
    private String type;
    private int passengers;

    public Elevator(String type) {
        if (type != null && type.length() > 0) {
            this.type = type;
        } else {
            this.type = "A";
        }
    }

    public void increasePassengers() {
        passengers++;
    }

    public boolean equals(Object obj) {
        if (obj == this) return true;
        if (!(obj instanceof Elevator)) return false;
        return type == ((Elevator) obj).type;
    }

    public String toString() { return type + " " + passengers; }
}
```

Which of the following hashCode() methods will satisfy the Java Hash Code contract? Circle all that apply.

- a. public int hashCode() { return type.length(); }
- b. public int hashCode() { return passengers; }
- c. public int hashCode() { return type.charAt(0) * -1; }
- d. public int hashCode() { return type.hashCode(); }
- e. public int hashCode() { return 0; }
- f. public int hashCode() { return type.charAt(0); }

Problem #3 (Linear Data Structures)

A linked list is used to keep track of menu items in a restaurant. There are two kinds of menu items: salads and desserts. Each menu item has a name and a number of calories. For this problem you may not add any instance variables nor static variables to the classes, you may not add any methods to the Node class, and you may not use the Java API LinkedList class. A driver (which you may ignore) illustrating the functionality of methods you need to implement, and a cheat sheet with map methods can be found at the end. You may not use the **add** method you see in the sample driver during the implementation of other methods.

```
public class LinkedList {
    private class Node {
        private MenuItem data;
        private Node next;

        private Node(MenuItem data) {
            this.data = data;
            next = null;
        }
    }

    private Node head;

    public LinkedList() {
        head = null;
    }
}

public class Dessert extends MenuItem {
    private String description;

    public Dessert(String name, int calories, int chocolate) {
        super(name, calories);
        description = name + " " + calories + " " + chocolate;
    }

    public String toString() { return description; }
}

public class Salad extends MenuItem {
    private String description;

    public Salad(String name, int calories, int tomatoes) {
        super(name, calories);
        description = name + " " + calories + " " + tomatoes;
    }

    public String toString() { return description; }
}
```

Map Methods

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

Driver

```
MenuItem[] items = {new Salad("Vegi", 200, 3),
                    new Dessert("Cake", 120, 10),
                    new Salad("CC", 50, 4),
                    new Dessert("IC", 55, 10)};

LinkedList list = new LinkedList();
for (int i = 0; i < items.length; i++) {
    list.add(items[i]);
}
System.out.println("Orig: " + list);
Map<String, Integer> map = list.getSalads(true);
System.out.println("Salads: " + map);
System.out.println("CC first: " + list.moveToFront("CC"));
```

Output

```
Orig: " Vegi 200 3, Cake 120 10, CC 50 4, IC 55 10"
Salads: {CC=50, Vegi=200}
CC first: " CC 50 4, Vegi 200 3, Cake 120 10, IC 55 10"
```

1. (50 pts) Provide a **NON-RECURSIVE** implementation for the **moveToFront** method below. The method moves to the front of the list the first **node** from the list (if any) that has a **data** value corresponding to the **itemName** parameter. If the node is already the first node of the list, there is no work to be done. The method will return a reference to the current object. **IMPORTANT: you must move the node (not just the menu item) to the front of the list.** That means the **head** reference needs to be adjusted so it points to the node being moved, and references to the node/references the node has, need to be updated.

Answer:

```
public LinkedList moveToFront(String itemName) {
    if (head != null) {
        Node prev = null, curr = head;

        while (curr != null) {
            if (curr.data.getName().equals(itemName)) {
                if (head != curr) {
                    prev.next = curr.next;
                    curr.next = head;
                    head = curr;
                }
                break;
            }
            prev = curr;
            curr = curr.next;
        }

        return this;
    }
}
```

2. (60 pts) Provide a **NON-RECURSIVE** implementation for the **getSalads** method below. The method returns a map with the name of a salad (key of the map) and the salad's calories (value of the value) for each salad in the linked list (if any). The method will return a **TreeMap** when the **sorted** parameter is true, and a **HashMap** otherwise. An empty map will be returned if no salads are found.

Answer:

```
public Map<String, Integer> getSalads(boolean sorted) {
    Map<String, Integer> map = sorted ? new TreeMap<String, Integer>()
                                       : new HashMap<String, Integer>();

    Node curr = head;
    while (curr != null) {
        if (curr.data instanceof Salad) {
            Salad salad = (Salad)curr.data;
            map.put(salad.getName(), salad.getCalories());
        }
        curr = curr.next;
    }

    return map;
}
```

3. (26 pts) Provide a **RECURSIVE** implementation for the **twoAdjacentSame** method below. The method returns true if there is at least two menu items in the list with the same name that are adjacent (next) to each other; otherwise the method will return false. For this problem you may only add one auxiliary method. If you use any loop construct (e.g., while, do while, for) you will automatically receive 0 credit.

Answer:

```
public boolean twoAdjacentSame() {
    if (head == null || head.next == null) {
        return false;
    }

    return twoAdjacentSame(head, head.next);
}

private boolean twoAdjacentSame(Node prev, Node curr) {
    if (curr != null) {
        if (prev.data.getName().equals(curr.data.getName())) {
            return true;
        } else {
            return twoAdjacentSame(curr, curr.next);
        }
    }
    return false;
}
```