



# University of Maryland College Park

## Department of Computer Science

### CMSC132 Fall 2022

### Exam #3

FIRSTNAME, LASTNAME (PRINT IN UPPERCASE):

KEY

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 100 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.
- 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

Problem #1 (Short Answer)	15
Problem #2 (Stacks)	20
Problem #3 (Linked Lists and Sets)	65
Total	100

## **Problem #1 (Short Answers) – 15 pts – 3 pts each**

1. Define what is meant by tail recursion (no more than 2 sentences)?

**A recursive function in which the recursive call is the last statement that is executed by the function.**

2. Define what is meant by the Java Hash Code contract (no more than 2 sentences)?

**if  $a.equals(b) == true$ , then we must guarantee  $a.hashCode() == b.hashCode()$**

3. Given the code pseudocode below:

```
//some code A
for (int i = 0; i < n; i++)
{
    //some code B
    for (int j = i+1; j < n; j++)
    { //some code C }

    //some code D
}
```

Assume input size  $n$ . How many times does **//some code C** execute on the iteration  $i=k$ ,  $0 \leq k < n$ ? Provide the answer with an expression in terms of  $n$  and  $k$ . What is the overall asymptotic complexity of the code above (in terms of Big O)?

**//some code C execute  $(n-(k+1))$  times  
Algorithm is  $O(n^2)$ .**

4. Define what is meant by Big-Omega (no more than 2 sentences)?

**Function  $f(n)$  is in  $\Omega(g(n))$  if  
For some positive constants  $M, N_0$   
 $M * g(n) \leq f(n)$ , for all  $n \geq N_0$ . It is a lower bound.**

5. In class, we spoke at great length about the linear probing strategy to resolve hashing collision. In no more than 2 sentences, explain how the insertion algorithm works. Remember that duplicates are not allowed and there are 3 states in the table: occupied, never used, and removed.

**Insertion → searches the probe sequence, keeping track of the first element that is in the *Removed* or *NeverUsed* state. If the key is not found it is placed in the first element that was in the *Removed* or *NeverUsed* state**

## **#2 (Stacks) - 20 pts**

1. Matthew, one of our lead TAs, provided the following code to find the max value in a stack as part of his review:

```
/*
 * Given a stack of comparable elements, return the largest. Assume elements
 * are non-null. Return null if empty.
 */
public static <E extends Comparable<E>> E max(Stack<E> stack) {
    if (stack.empty()) {
        return null;
    }

    E max = stack.pop();

    while (!stack.empty()) {
        E item = stack.pop();

        if (item.compareTo(max) > 0) {
            max = item;
        }
    }

    return max;
}
```

He suggested you try to make it recursive. Sounds like a good test question! Finish `maxR` by making one call to `maxRAux`. `maxRAux` can have any 2 parameters you want, the code must be fully recursive (no loops), and you can only use library method calls shown in the iterative code above.

```
public static <E extends Comparable<E>> E maxR(Stack<E> stack) {
    if (stack.empty()) {
        return null;
    }

    return maxRAux(stack, stack.pop()); //make your call to maxRAux
}
```

**//write maxRAux on next page**

**Directory ID:**

```
public static <E extends Comparable<E>> E maxRAux(Stack<E> stack, E max) {  
    if (!stack.empty())  
    {  
        E item = stack.pop();  
        max= item.compareTo(max) > 0 ? maxRAux(stack, item) :  
            maxRAux(stack, max);  
    }  
    return max;  
}
```

### #3 (Linked List and Sets) - 65 pts

Given the code below, you are asked to complete the missing methods:

**/\* Assume import statements need to use the 3 JCF Sets and the Set Interface\*/**

```
public class MyLinkedList<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;
    private Node tail;

    public MyLinkedList() {
        head =tail= null;
    }

    /* Adding to the end of the list in O(1) time*/
    public MyLinkedList<T> add(T data) {
        // YOU WRITE THIS METHOD
    }

    /* Iterative method that will return a set of repeats and modify the list to have unique values*/
    public Set <T> makeUniqueIter(){
        // YOU WRITE THIS METHOD
    }

    /* Will make one call to the recursive aux method*/
    public void makeUniqueRec(){
        // YOU WRITE THIS METHOD
    }

    /*Recursive method that will modify the list to have unique values*/
    // YOU WILL WRITE makeUniqueRecAux (with a maximum of 3 parameters)

    public String toString() {
        String result = "\" " ";
        Node curr = head;

        while (curr != null) {
            result += curr.data + " ";

            curr = curr.next;
        }

        return result + "\"";
    }
}
```

<b>Directory ID:</b>
----------------------

## Sample Driver

```
import java.util.Set;

public class SampleDriver {

    public static void main(String[] args) {

        String answer = "";
        MyLinkedList<String> newList = new MyLinkedList<String>();
        newList.add("Sarah").add("Rose").add("Peter").add("Kelly").add("Albert");
        newList.add("Rose").add("Peter").add("Kelly").add("Joe");
        answer+=newList+"\n";
        Set<String> result =newList.makeUniqueIter();
        answer+=newList+"\n"; //all data is unique
        newList.add("Mark"); // ensures tail is OK
        answer += newList + "\n";

        for (String s: result) //only the repeats in alphabetical order
        {
            answer+= s+" ";
        }

        answer += "\n";

        MyLinkedList<Integer> newList1 = new MyLinkedList<Integer>();
        newList1.add(35).add(37).add(35).add(84);
        newList1.add(16).add(16).add(16).add(84).add(35);
        answer+=newList1+"\n";
        newList1.makeUniqueRec();
        answer+=newList1+"\n"; //all data is unique

        System.out.println(answer);

    }

}
```

## Sample Driver Output

```
" Sarah Rose Peter Kelly Albert Rose Peter Kelly Joe "
" Sarah Rose Peter Kelly Albert Joe "
" Sarah Rose Peter Kelly Albert Joe Mark "
Kelly Peter Rose
" 35 37 35 84 16 16 16 84 35 "
" 35 37 84 16 "
```

1. Write the code for `add` that will add a node to the end of the list in  $O(1)$  time (you need to know what this means). Make sure that the `head` points to the first node and `tail` to the last when the operation is completed. If there is just one node in the list, both `head` and `tail` point to it. Return the current object.

```
public MyLinkedList<T> add(T data) {  
    if (head==null) //first node  
        head=tail = new Node(data);  
    else {  
        tail.next= new Node(data);  
        tail = tail.next;  
    }  
    return this;  
}
```

2. Write the code for `makeUniqueIter` that will return a set of all data values that shows up more than once in the linked list. Adding data to the set using a reference copy is fine. When you iterate through the set (see sample driver), the data should come out in their natural ordering (e.g. if numbers in numeric order). If the linked list is empty, just return an empty set.

In addition to returning this set, the method must modify the current object so that any data value that has already been encountered once in the traversal, will have its node be removed from the list. In other words, after the method is done, the linked list will have only unique data values. **The code must be iterative** (i.e. use a loop) and not recursive. When writing your code, keep in mind that the head will never need to be removed since it cannot be a repeat, but the tail node may be deleted (so you need to make sure that `tail` points to the last node when you have removed all repeats).

You may use as many (or as few) of the 3 JCF sets as you want in the code, but you are only limited to invoking the default set constructors, the `add` method, and the `contains` method. Do not make a new linked list (modify the original by detaching nodes you don't want) and don't use any other library data structures other than the 3 sets.

```
public Set <T> makeUniqueIter() {

    if(head==null) //if empty nothing to do
        return new TreeSet<T>();

    Set <T> mySet = new HashSet<T>();
    Set <T> repeat = new TreeSet<T>();

    Node curr = head;
    Node prev = null;

    while (curr != null) {
        if(!mySet.contains(curr.data)) { //not in set
            mySet.add(curr.data); //add to set
            prev = curr; //change prev to be current
        }
        else{
            //in set, so remove by making prev.next point to next of curr
            repeat.add(curr.data);
            if (curr.next == null)
                tail = prev; // in case tail is removed
            prev.next = curr.next;
        }

        curr = curr.next; //move the next node in either case
    }
    return repeat;
}
```



3. Write the code for `makeUniqueRec`. The end result of calling this method is that the linked list will be modified in the same fashion as is done via a call to `makeUniqueIter`. That is, any data value that has already been encountered once in the traversal of the linked list will have its node removed so that the linked list will have all unique data values. If the linked list is empty, just return. Unlike `makeUniqueIter`, you do not return a set of the repeats.

`makeUniqueRec` can have any non-iterative startup code you feel is necessary, but it must make one call to the private recursive `makeUniqueRecAux` which can have up to 3 arguments of your choice. `makeUniqueRecAux` recursively removes any node from the list that contains data already encountered in the traversal. **The code must be recursive** (i.e. do not use a loop) and not iterative. You cannot make a call to `makeUniqueIter` for this question, nor call this method in question #2. When writing your code, keep in mind that the head will never need to be removed since it cannot be a repeat, but the tail node may be deleted (so you need to make sure that `tail` points to the last node when you have removed all repeats).

You may use as many (or as few) of the 3 JCF sets as you want in the code, but you are only limited to invoking the default set constructors, the `add` method, and the `contains` method. Do not make a new linked list (modify the original by detaching nodes you don't want) and don't use any other library data structures other than the 3 sets.

```
public void makeUniqueRec(){
    //at some point make one call to makeUniqueRecAux that you will define below

    if(head==null) //if empty nothing to do
        return;

    HashSet <T> mySet = new HashSet<T>();
    makeUniqueRecAux(head, null, mySet);
}

private void makeUniqueRecAux(Node headAux, Node prev, HashSet <T> mySet) {
    if (headAux != null) {
        if(!mySet.contains(headAux.data)) { //not in set
            mySet.add(headAux.data); //add to set
            makeUniqueRecAux(headAux.next, headAux, mySet);
        }
        else{
            prev.next = headAux.next;
            if (headAux.next == null)
                tail = prev;
            makeUniqueRecAux(headAux.next, prev, mySet);
        }
    }
}
```

**LAST PAGE**