



# University of Maryland College Park

## Department of Computer Science

### CMSC132 Fall 2021

### Exam #3

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 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 (Sets from JCF)	14
Problem #2 (Applications of Stack)	20
Problem #3 (Miscellaneous)	6
Problem #4 (Hash Tables and Linked Lists)	60
Total	100

## **Problem #1 (Sets from JCF) – 14 pts**

Assume the necessary import statements and consider the code below:

```
public class SetQuestion {

    public static Object[] notUnique (Integer [] arr) {

        HashSet <Integer> h = new HashSet <Integer>();
        TreeSet <Integer> t = new TreeSet<Integer>();

        return (notUniqueAux (arr, h, t, 0).toArray());

    }

    private static Set<Integer> notUniqueAux (Integer [] arr, HashSet <Integer> h,
        TreeSet <Integer> t, int index) {

        // YOU WRITE THIS METHOD

    }

    public static void main(String[] args) {

        System.out.println(Arrays.toString(notUnique(new Integer[]
            {7,5,4,3,7,4,32,12,54,54,12,12})));
        System.out.println(Arrays.toString(notUnique(new Integer[] {7,5,4})));

    }

}
```

### **Output of main**

```
[4, 7, 12, 54]
[]
```

Write the code for notUniqueAux. Its job is to return an ordered set having only values appearing more than once from arr. You **cannot** add any fields, create any local variables (using the four parameters are ok), or have any loops. How you use the 4 parameters is up to you. Your code must be tail recursive. As for methods of the sets from JCF, you can only use the add method. Remember that if the set already contains the element, the call to add leaves the set unchanged and returns false.

**Write the answer on the back**

```
private static Set<Integer> notUniqueAux (Integer [] arr, HashSet <Integer> h,
    TreeSet <Integer> t, int index) {
    if (index == arr.length) //finished with array elements
        return (t); //return tree set

    if(h.add(arr[index]) ==false)
    {
        //already in hashset, so repeated value added to tree set
        t.add(arr[index]);
    }

    return notUniqueAux (arr, h, t, index+1);
}
```

**Directory ID:**

## **Problem #2 (Applications of Stack) - 20 pts**

Assume the necessary import statements and consider the code below:

```
public class StackQuestion {  
  
    public static boolean matchingPar(String arg) {  
        Stack <Character> s = new Stack<Character>();  
  
        // YOU WRITE THIS METHOD  
    }  
  
    public static void main(String[] args) {  
  
        System.out.println(matchingPar("ab")); //true  
        System.out.println(matchingPar("(ab)")); //true  
        System.out.println(matchingPar("a((ab))")); //false  
        System.out.println(matchingPar("v((vv))")); //true  
        System.out.println(matchingPar(")z((xc))(")); //false  
        System.out.println(matchingPar("()e((*&))()")); //true  
    }  
}
```

Write the code for `matchingPar`. Its job is to return true only if all the open and closing parenthesis are correctly paired. All other characters in the string can be ignored. Use the local Stack variable, `s`, to help you with the logic as you loop through the string and look at one character at a time using the string method: `charAt`.

As for Stack methods you only need to use:

`push(Character item)` - Pushes an item onto the top of this stack.

`pop()` - Removes the object at the top of this stack and returns that object as the value of this function.  
Throws: `EmptyStackException` - if this stack is empty.

`empty()` - Returns true if and only if this stack contains no items; false otherwise.

**Write the answer on the back**

```

public static boolean matchingPar(String arg) {
    Stack <Character> s = new Stack<Character>();
    for(int i =0; i <arg.length(); i++)
    {

        if(arg.charAt(i) == '(')
            s.push('(');
        else if (arg.charAt(i) == ')')
        {
            try {
                s.pop();
            }
            catch (EmptyStackException e)
            {
                return false; //there is ) but no ( in stack
            }
        }

    }

    if (s.empty())
        return true;
    return false; //there is ( in stack but no ) found
}

```

### **Problem #3 (Miscellaneous) – 6 pts**

1. What is the Big-O complexity of `matchingPar` method, as described in problem 2, in terms of `n` being the number of characters in the parameter. You can assume that the stack methods are all  $O(1)$  operations. **Explain your answer. No more than 2 sentences.**

**`O(n)` – we iterate through each character linearly**

2. Does the `StackQuestion` class from problem 2 satisfy the java hash code contract as it is or does something additional have to be added ? **Explain your answer. No more than 3 sentences.**

**yes - Default `hashCode()` and `equals()` satisfy contract”**

### **Problem #4 (HashTables and Linked Lists) – 60 pts**

Assume the necessary import statements and consider the code below:

```
public class Exam3HashSet {
    private StringList hashTable[];
    private int capacity; //number of array elements in the table
    private int size; //number of data in the hash table

    public Exam3HashSet(int capacity) {

        this.hashTable = new StringList[capacity];
        for (int i =0; i< hashTable.length; i++)
            hashTable[i] = new StringList ();
        this.capacity = capacity;
        this.size = 0;
    }

    public double loadFactor()
    {
        // YOU WRITE THIS METHOD
    }

    public void insert (String s)
    {
        // YOU WRITE THIS METHOD
    }

    public boolean contains (String s)
    {
        // YOU WRITE THIS METHOD
    }
}
```

```

public void remove (String s)
{
    // YOU WRITE THIS METHOD
}

public int hashFunction(String s)
{
    return Math.abs((s.hashCode() * 104059) % capacity);
}

public void displayHashTable()
{
    for (StringList s: hashTable)
    {
        System.out.println(s+"\n");
    }
}
//private inner StringList class to be used as buckets
private class StringList {

    private class Node {
        private String data;
        private Node next;

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

    private Node head;

    private StringList() {
        head = null;
    }

    private void add(String input) {
        // YOU WRITE THIS METHOD
    }

    private boolean search(String input) {
        // YOU WRITE THIS METHOD
    }

    private void delete(String input) {
        // YOU WRITE THIS METHOD
    }

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

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

        return result + "\n";
    }
}
}

```

## Sample Driver

```
public class SampleDriver {  
    public static void main(String[] args) {  
        Exam3HashSet mySet = new Exam3HashSet (5);  
  
        mySet.insert("trees");  
        mySet.insert("heaps");  
        mySet.insert("lists");  
        mySet.insert("stacks");  
  
        mySet.insert("red");  
        mySet.insert("blue");  
        mySet.insert("green");  
        mySet.insert("yellow");  
  
        mySet.insert("lists"); //duplicates will not get added to set  
        mySet.insert("stacks");  
        mySet.insert("red");  
        mySet.insert("blue");  
  
        System.out.println("After insert: ");  
  
        System.out.println("Load Factor is : " + mySet.loadFactor());  
  
        mySet.displayHashTable();  
  
        System.out.println("-----");  
  
        System.out.println("red in table? " + mySet.contains("red"));  
        System.out.println("map in table? " + mySet.contains("map"));  
  
        System.out.println("-----");  
  
        mySet.remove("trees");  
        mySet.remove("heaps");  
        mySet.remove("lists");  
        mySet.remove("stacks");  
  
        mySet.remove("trees"); //already deleted, no change  
        mySet.remove("heaps");  
        mySet.remove("lists");  
        mySet.remove("stacks");  
  
        System.out.println("After remove: ");  
  
        System.out.println("Load Factor is : " + mySet.loadFactor());  
  
        mySet.displayHashTable();  
  
    }  
}
```



## Sample Driver Output

After insert:

Load Factor is : 1.6

" "

" lists stacks "

" green "

" trees red blue yellow "

" heaps "

-----  
red in table? true

map in table? false  
-----

After remove:

Load Factor is : 0.8

" "

" "

" green "

" red blue yellow "

" "

1. With **one** statement, complete the method below. You should know how to calculate the load factor from lecture.

```
public double loadFactor() {  
    return (double) size / capacity;  
}
```

2. With **one** statement, complete the method below. The hash table is using separate chaining and the add method of the inner `StringList` class has the logic for adding `s` to the linked list if it is not already there.

```
public void insert (String s){  
    hashTable[hashFunction(s)].add(s);  
}
```

3. With **one** statement, complete the method below. The hash table is using separate chaining and the search method of the inner `StringList` class has the logic for finding `s` in the linked list if it is there.

```
public boolean contains (String s) {  
    return hashTable[hashFunction(s)].search(s);  
}
```

4. With **one** statement, complete the method below. The hash table is using separate chaining and the `delete` method of the inner `StringList` class has the logic for removing `s` from the linked list if it there.

```
public void remove (String s) {  
    hashTable[hashFunction(s)].delete(s);  
}
```

5. Implement the `search` method of the inner `StringList` class. It should return `true` if `input` is in the linked list and `false` otherwise.

```
private boolean search(String input) {  
    Node curr = head;  
    while (curr != null) {  
        if (curr.data.equals(input)) //just a linear search, if you find  
            return true;  
  
        curr = curr.next;  
    }  
    return false;  
}
```

6. Implement the add method of the inner StringList class. It should add input to the end of the linked list if it is not already there. Do not forget to update size.

```
private void add(String input) {
    if (head == null) {

        head = new Node(input);    //add first node
        size++;

    }

    else {    //at least one node
        Node curr, prev;
        curr = prev = head;
        while (curr != null) {
            if (curr.data.equals(input))    //already in list
                return;    //just return, don't add

            prev = curr;
            curr = curr.next;

        }
        prev.next = new Node(input);    //add at end new value
        size++;

    }

}
```

7. Implement the `delete` method of the inner `StringList` class. It should delete input from the linked list if it is there. Do not forget to update `size`.

```
private void delete(String input) {
    if (head == null)
        return;

    //not empty list at this point

    if (head.data.equals(input)) {
        head = head.next;    //delete head
        size--;
    }

    else {
        Node curr, prev;
        curr = head.next;
        prev = head;
        while (curr != null) {
            if (curr.data.equals(input))    //found it
            {
                prev.next = curr.next;
                size--;
            }
            prev = curr;
            curr = curr.next;
        }
    }
}
```