|  |  |
| --- | --- |
|  | **University of Maryland College Park** |
| **Dept of Computer Science** |
| **CMSC132 Summer 2014** |
| **Midterm II** |

First Name (PRINT): **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Last Name (PRINT): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

University Directory ID (e.g., testudoJr) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

I pledge on my honor that I have not given or received any unauthorized assistance on this examination.

Your signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Instructions**

* This exam is a closed-book and closed-notes exam.
* Total point value is 200 pts.
* The exam is a 50 minutes exam.
* Please use a pencil to complete the exam.
* **WRITE NEATLY**.

***Grader Use Only***

|  |  |  |  |
| --- | --- | --- | --- |
| #1 | Algorithmic Complexity | (44) |  |
| #2 | Generics | (20) |  |
| #3 | Hashing | (20) |  |
| #4 | Exceptions | (20) |  |
| #5 | Linear Data Structures | (96) |  |
| **Total** | Total | (200) |  |

**Problem #1 (44 pts) Algorithmic Complexity**

1. (14 pts) Calculate the asymptotic complexity of the code snippets below (using big-O notation) with

respect to the problem size **n**.

* 1. (4 pts) f(**n**) = O( **1** )

**if (n > y) {**

**System.*out*.println("Hello");**

**} else {**

**System.*out*.println("Goodbye");**

**}**

* 1. (5 pts) f(**n**) = O( **n** )

**for (k = 0; k < n/2; k++) {**

**for (i = 0; i <= 1000; i++) {**

**System.*out*.println(k + " " + i);**

**}**

**}**

* 1. (5 pts) f(**n**) = O( **n2log(n)** )

**for (int i = 1; i <= (n \* n); i++)**

**for (int k = 1; k <= n; k = k\*2)**

**System.out.println("Processing");**

1. (8 pts) Give the asymptotic bound of the following functions:

1. log(**n**) + 300**n**4 + **n -** 2 f(**n**) = O( **n**4 )
2. **n + n**2 + 200 f(**n**) = O( **n**2 )
3. 125 f(**n**) = O( **1** )
4. (6 pts) List the following big-O expressions in order of asymptotic complexity (lowest complexity first)

O(**n**log(**n**)) O(**n**!) O(**nk**) O(log(**n**))

Answer: O(log(**n**)), O(**n**log(**n**)), O(**nk**), O(**n**!)

1. (4 pts) What is the big-O associated with array indexing?

Answer: O(**1**)

1. (6 pts) Give the complexity of an algorithm for problem size **n** whose running time:
   1. Increases by a constant when **n** doubles O( **log(n)** )
   2. Does not change when **n** quadruples O( **1** )
2. (6 pts) Briefly describe what is the critical section of an algorithm.

Answer: The section of the algorithm that dominates the overall execution time

**Problem #2 (20 pts) Generics**

Modify the following class (feel free to edit/cross out) so it becomes a generic Bag class that allow us to define bags not only of strings (e.g., Bag<String>), but also of any class (e.g., Bag<Integer>).

|  |  |
| --- | --- |
| /\* Original code \*/  public class Bag {  private int maxSize;  private String[] data;  int numberElements;    public Bag(int maxSize) {  this.maxSize = maxSize;  data = new String[maxSize];  numberElements = 0;  }    public void addElement(String element) {  if (numberElements < maxSize) {  data[numberElements++] = element;  }  }    public String returnLastElementAdded() {  if (numberElements > 0) {  return data[numberElements - 1];  }  return null;  }    public String toString() {  String answer = "";  for (String elem : data) {  answer += elem;  }    return answer;  }    } | /\* Answer \*/  public class Bag<T> {  private int maxSize;  private T[] data;  int numberElements;  public Bag(int maxSize) {  this.maxSize = maxSize;  data = (T[]) new Object[maxSize];  numberElements = 0;  }  public void addElement(T element) {  if (numberElements < maxSize) {  data[numberElements++] = element;  }  }  public T returnLastElementAdded() {  if (numberElements > 0) {  return data[numberElements - 1];  }  return null;  }  public String toString() {  String answer = "";  for (T elem : data) {  answer += elem;  }  return answer;  }  } |

**Problem #3 (20 pts) Hashing**

1. (4 pts) A hashCode() method that generates unique values for each object will:
   1. Reduce collisions
   2. Increase collisions
   3. Not make any difference (the same as generating different values)
   4. None of the above

Answer: a.

1. (6 pts) Describe the Java Hash Code contract.

Answer: if a.equals(b) == true, then we must guarantee a.hashCode() == b.hashCode()

1. (4 pts) What is usually returned by the default implementation of the hashCode() method?

Answer: The address of the object.

1. (6 pts) Does the default implementation of equals and hashCode in Java satisfy the Java Hash Code contact? Explain briefly. Yes or No answer will receive no credit.

Answer: Yes. The hashCode() returns the address of the object and the default equals compares addresses.

**Problem #4 (20 pts) Exceptions**

**The function inRange() throws a checked exception named InRangeException.**

1. (10 pts) Modify the following code fragment so the exception InRangeException is handled by a catch clause. Display the message “Invalid value” (using System.out.println) when the exception occurs.

**public static void processRequest(int value) {**

**if (*inRange*(value)) {**

**System.*out*.println(value \* value);**

**}**

**}**

Answer:

**try {**

**if (inRange(value)) {**

**System.out.println(value \* value);**

**}**

**} catch(InRangeException e) {**

**System.out.println("Invalid value");**

**}**

1. (3 pts) Modify the provided code fragment above so we do not need to handle the exception in the function processRequest.

Answer: public static void processRequest(int value) throws InRangeException

1. (7 pts) Modify the following code fragment so it throws an IllegalArgumentException if 0 is provided for **y**.

**public static boolean isDivisible(int x, int y) {**

**return x % y == 0 ? true : false;**

**}**

Answer:

**public static boolean isDivisible(int x, int y) {**

**if (y == 0)**

**throw new IllegalArgumentException();**

**return x % y == 0 ? true : false;**

**}**

**Problem #5 (96 pts) Linear Data Structures**

Implement the methods below based on the following Java class definitions. **You may not modify the Node class and you may not add any instance nor static variables.** In addition, you may not use the Java API LinkedList class.

**public class LinkedList<K extends Comparable<K>, V> {**

**private class Node {**

**private K key;**

**private V value;**

**private Node next;**

**public Node(K key, V value) {**

**this.key = key;**

**this.value = value;**

**next = null;**

**}**

**}**

**private Node head; /\* List head pointer \*/**

**public LinkedList() { head = null; }**

**public TreeMap<K, V> getMap() { /\* YOU MUST IMPLEMENT \*/ }**

**public void removeLargerOrEqualTo(K elem) { /\* YOU MUST IMPLEMENT \*/ }**

**}**

1. Implement the **getMap** method. This method will return a TreeMap with the key and value pairs present tin the linked list. For this problem:
   1. You must implement a **RECURSIVE** solution, otherwise you will not get credit.
   2. **You will get 0 credit for this problem if you use any iteration statement (for, while, do while).**
   3. You may only add one auxiliary method.
   4. You may not modify the original list.
2. Implement the method **removeLargerOrEqualTo.** This method removes from the original list elements that are larger or equal to the parameter. For this problem:
   1. You must implement a **NON-RECURSIVE** solution, otherwise you will not get credit.
   2. You may not add auxiliary methods.
   3. You may not create any nodes.

**You may find the following Map methods helpful:**

* V **get**(Object key) - Returns the value to which this map maps the specified key.
* V **put**(K key,V value) - Associates the specified value with the specified key in this map.
* Set<K> **keySet**() - Returns a set view of the keys contained in this map.
* boolean **isEmpty**() - Returns true if this map contains no key-value mappings.

1. **(40 pts)** getMap()

Answer:

**public TreeMap<K, V> getMap() {**

**TreeMap<K, V> map = new TreeMap<K, V>();**

**return getMapAux(head, map);**

**}**

**public TreeMap<K, V> getMapAux(Node headAux, TreeMap<K, V> map) {**

**if (headAux != null) {**

**map.put(headAux.key, headAux.value);**

**getMapAux(headAux.next, map);**

**}**

**return map;**

**}**

1. **(56 pts)** removeLargerOrEqualTo

Answer:

public void removeLargerOrEqualTo(K elem) {

if (head != null) {

Node curr = head, prev = null;

while (curr != null) {

if (curr.key.compareTo(elem) >= 0) {

if (head == curr) {

head = head.next;

curr = head;

} else {

prev.next = curr.next;

curr = curr.next;

}

} else {

prev = curr;

curr = curr.next;

}

}

}

}