|  |  |
| --- | --- |
|  | **University of Maryland College Park** |
| **Dept of Computer Science** |
| **CMSC132 Summer 2013** |
| **Midterm II Key** |

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 | (50) |  |
| #2 | Generics | (20) |  |
| #3 | Hashing | (20) |  |
| #4 | Linear Data Structures | (110) |  |
| **Total** | Total | (200) |  |

**Problem #1 (50 pts) Algorithmic Complexity**

1. (18 pts) Calculate the asymptotic complexity of the code snippets below (using big-O notation) with

respect to the problem size **n**.

* 1. (8 pts) f(**n**) = O( **nlog(n)**  )

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

**k = 1;**

**while (k < n) {**

**k = 2 \* k;**

**System.out.println(i);**

**}**

**m = 1;**

**while (m < n) {**

**m = 2 \* m;**

**System.out.println(i);**

**}**

**}**

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

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

**System.*out*.println("Additional Computing");**

* 1. (6 pts) f(**n**) = O( **n** )

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

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

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

**}**

**}**

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

1. log(**n**) + **n + n**log(**n**) f(**n**) = O( **nlog(n)** )
2. **n**k + 6**n + 3**k  f(**n**) = O( **nk** )
3. (4 pts) How can we show the best way to grow an array is by doubling its size?
   1. By using amortized analysis **√**
   2. By using best case analysis
   3. By using a recursive solution
   4. By experimenting with an array
   5. b. and c.
   6. None of the above
4. (4 pts) Give the complexity of an algorithm for problem size **n** whose running time increases by a constant when **n** doubles: O( log(n) )
5. (4 pts) Big-O results are valid
6. For any values associated with **n**
7. Only for large values of **n √**
8. Only for small values of **n**
9. It depends on the algorithm
10. None of the above
11. (4 pts) Assume we have two functions: f(n) = n / 4 and g(n) = 7n + 10. From algorithmic complexity (Big O) point of view:
12. They are equivalent. **√**
13. g(n) is better than f(n)
14. f(n) is better than g(n)
15. None of the above
16. (4 pts) An algorithm can have:
17. Only one critical section
18. At most two critical sections
19. Any number of critical sections **√**
20. None of the above
21. (4 pts) Briefly describe what is the critical section of an algorithm.

**Answer:** The portion of the algorithm that dominates the overall execution time.

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

Given the following interface/class definitions, indicate which of the method calls in main are valid or invalid (circle your answers).

**public interface PortableDevice {…}**

**public class Computer {…}**

**public class Laptop extends Computer implements PortableDevice {…}**

**public class Utilities {**

**public static void printComputers(Collection<Computer> cl) {…}**

**public static void printObjects(ArrayList<Object> cl) {…}**

**public static void printAny(ArrayList<?> cl) {…}**

**public static void printAnyComputer(ArrayList<? extends Computer> cl) {…}**

**public static void main(String[] args) {**

**Laptop laptop1 = new Laptop("Delly", 10, "BatA");**

**Computer computer1 = new Computer("ICM", 20);**

**ArrayList<Computer> computerList = new ArrayList<Computer>();**

**ArrayList<PortableDevice> portableList = new ArrayList<PortableDevice>();**

**portableList.add(laptop1);**

**computerList.add(laptop1);**

**computerList.add(computer1);**

**/\* CIRCLE VALID OR INVALID \*/**

**printComputers(computerList); // VALID**

**printObjects(computerList); // INVALID**

**printAny(computerList); // VALID**

**printAnyComputer(computerList); // VALID**

**printAnyComputer(portableList); // INVALID**

**}**

**}**

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

1. (6 pts) According to the Java Hash Code contract given two objects a and b, if a.equals(b) is true then:
   1. a.hashCode() == b.hashCode() √
   2. a.hashCode() < b.hashCode()
   3. a.hashCode() > b.hashCode()
   4. a.hashCode() != b.hashCode()
   5. Either a. or b.
   6. None of the above
2. (3 pts) By using hashing we have a guaranteed complexity of:
   1. O(1)
   2. O(log(n))
   3. O(nlog(n))
   4. None of the above √
3. (3 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
4. (3 pts) The following function compiles, but it has a bug. Identify the bug and fix it.

**public static int hashBucket(Object x, int N) {**

**int h = x.hashCode();**

**return Math.*abs*(h) % N;**

**}**

Answer: **Math.abs(h) % N** needs to be **Math.abs(h % N)**

1. (5 pts) Given the following class, which of the methods below are valid hashCode() methods for the class? Notice we don’t care about how good the hashCode() method is.

**public class Car {**

**private String name;**

**private int tag;**

**public boolean equals(Object obj) {**

**if (obj == this)**

**return true;**

**if (!(obj instanceof Car))**

**return false;**

**return tag == ((Car)obj).tag;**

**}**

**public int hashCode() {**

**// CODE HERE**

**}**

**}**

1. **public int hashCode() { return name.length(); }**
2. **public int hashCode() { return -1; }**
3. **public int hashCode() { return tag - name.length(); }**
4. All of the above. √
5. None of the above.

**Problem #4 (110 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 or static variables.** In addition, you may not use the Java API LinkedList class. We have provided a main() method that illustrates the functionality associated with the methods you need to implement. Feel free to ignore this method if you know what to implement. Notice the main method relies on two methods (add, toString()) that you do not need to implement. The output of the main() method is:

**Members's list: Zoe John**

**List after remove: Zoe Albert Rose**

**Set: [John, Laura, Mike]**

**public class LinkedList {**

**private class Node {**

**private String data;**

**private Node next;**

**public Node(String data) {**

**this.data = data;**

**next = null;**

**}**

**}**

**private Node head;**

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

**public LinkedList membersList(Map<String, Integer> people, int ageCutoff) {**

**/\* YOU MUST IMPLEMENT \*/**

**}**

**public void removeValuesInRange(String lowerBound, String upperBound, Set<String> set) {**

**/\* YOU MUST IMPLEMENT \*/**

**}**

**public static void main(String[] args) {**

**LinkedList list = new LinkedList();**

**list.add("John").add("Rose").add("Mike").add("Albert").add("Zoe").add("Laura");**

**TreeMap<String, Integer> ageInfo = new TreeMap<String, Integer>();**

**ageInfo.put("John", 18);**

**ageInfo.put("Rose", 15);**

**ageInfo.put("Mike", 4);**

**ageInfo.put("Zoe", 25);**

**LinkedList newList = list.membersList(ageInfo, 18);**

**System.out.println("Members's list: " + newList);**

**Set<String> set = new TreeSet<String>();**

**list.removeValuesInRange("John", "Mike", set);**

**System.out.println("List after remove: " + list);**

**System.out.println("Set: " + set);**

**}**

**}**

1. Implement the **membersList** method. This method will return a linked list with the names of people whose age is greater than or equal to the **ageCutoff** value. Age information can be found in the provided map (the map provides the age for a particular person).
   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. Do not include in the new list a person that is in the original list, but not in the map.
   4. You may only add one auxiliary method.
   5. You may not modify the membersList prototype (parameters or return type).
   6. You may not modify the original list (you must return a new list).

**Answer:**

**public LinkedList membersList(Map<String, Integer> people, int ageCutoff) {**

**LinkedList newList = new LinkedList();**

**newList.head = membersListAux(head, people, ageCutoff);**

**return newList;**

**}**

**private Node membersListAux(Node headAux, Map<String, Integer> ageInfo, int ageCutoff) {**

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

**Integer entry = ageInfo.get(headAux.data);**

**if (entry != null && entry >= ageCutoff) {**

**Node newNode = new Node(headAux.data);**

**newNode.next = membersListAux(headAux.next, ageInfo, ageCutoff);**

**return newNode;**

**} else {**

**return membersListAux(headAux.next, ageInfo, ageCutoff);**

**}**

**}**

**return null;**

**}**

1. Implement the method **removeValuesInRange.** This method removes from the original list elements that are in the specified range. 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.
   4. You should place in the set parameter all the elements that were removed.

**Answer:**

**public void removeValuesInRange(String lowerBound, String upperBound, Set<String> set) {**

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

**Node curr=head, prev=null;**

**while (curr != null) {**

**if (curr.data.compareTo(lowerBound) >= 0 &&**

**curr.data.compareTo(upperBound) <= 0) {**

**set.add(curr.data);**

**if (head == curr) {**

**head = head.next;**

**curr = head;**

**} else {**

**prev.next = curr.next;**

**curr = curr.next;**

**}**

**} else {**

**prev = curr;**

**curr = curr.next;**

**}**

**}**

**}**

**}**