Birthday: Month: _____ Day: _____

Question 1: (3 points)

When you modify a key that has been inserted into a HashMap will you be able to retrieve that entry again?

☐ Always ☒ Sometimes ☐ Never

Explain:

Solution: It is possible that the new Key will end up colliding with the original Key. Only in this rare situation will we be able to retrieve the value. It is a big no-no to modify the Key in a Map. How can we expect the data structure to find the object again for us if we change it?

When you modify a value that has been inserted into a HashMap will you be able to retrieve that entry again?

☒ Always ☐ Sometimes ☐ Never

Explain:

Solution: You can safely modify the value without any trouble. If you reference the value that you put in the tree, the changes will be reflected. Some people misunderstood the question, and indicated in their answer that the value in the hashMap had changed so we can't get the old unchanged version back. This is correct and a sensible interpretation and typically led people to select "NEVER".

RUBRIC:

Part 1: (out of 2)

+ 2	If they say "Sometimes" and explain that it is possible that they have the same hashCode still and that in that case it will find the key.
+ 0	If they say "Sometimes" without an explanation
+ 0	If they say anything other than "Sometimes".
+1	If they say "Never" – b/c changing the key changes the hash code.

Part 2: (out of 1)

+ 1	Correct answer for the second part of the question. "Always" or "Never" with the explanation described above.
+ 1	Some people made the point that you can't get the original "value" back because you've changed it, but they recognize that you'll be able to access the relevant entry.

Question 2 - Hashing (3 points)

a) What are the strengths of the `Car class`? Write "None" if there are no strengths related to hashing.

No strengths needed to be listed. This question is all about the flaw in the design discussed below.

b) What are the weaknesses of the `Car class`? Write "None" if there are no weaknesses related to hashing.

3 points	Explains that .equals objects don't return the same hashCode. This violates the requirement of using a HashMap and will not work.
2 points	Explains that .equals objects "should" return the same hashCode but describe the consequence which is a completely broken HashMap.
1 point	Mentions the discrepancy between equals and hashCode and discusses why this is an advantage. Or The answer suggests that equals isn't relevant, only the hashcode matters.

Question 3 (8 points):

You are provided the classes `Person`, `Address` and `AddressBook` in the handout. In the space on this and the next page, add code to any of the three classes or provide new versions of existing methods as necessary to implement the `getResidents` method for `AddressBook`. `getResidents` should return an `ArrayList` of `Person` objects that are associated with a given address. The `getResidents` method takes a single argument of an `Address` and must run in constant time. There are no restrictions on the time complexity of any other methods. java.util methods are outlined in the attached handout.

Please make it very clear in which class each method would appear.

A	-1 points	Not providing a hashCode method in Address.java
B	-1 points	Not providing an equals method in Address.java
J	-0.5	Compile time issues

Add these to Address.java

```
public int hashCode() {
     return this.myAddress.hashCode();
}
public boolean equals(Object obj){
     Address otherAddress = (Address) obj;
     return otherAddress.myAddress.equals(this.myAddress);
}
```

Add/write over these in AddressBook.java

F	-3 points	The second hashMap is <Address, Person>. And doesn't accomplish the goal of mapping an address to an array of people.

```
private HashMap<Address, ArrayList<Person>> residents = new
                                    HashMap<Address, ArrayList<Person>>();
```

C	-1 points	Doesn't check if put has written over an old address in a call to put
D	-1 point	In add tries to add a Person to an ArrayList that doesn't exist. They need to check if the result of "get" from their new HashMap is not null. If it is null they need to create a new array.
H	-0.5	Calling a method on an object to see if it is empty, when really that object might be null (causing a null pointer exception to try to call a method.)
N	-1	Not putting something into the hashMap or ArrayList.
O	-2	Doesn't add the person if it exists in the hashMap.

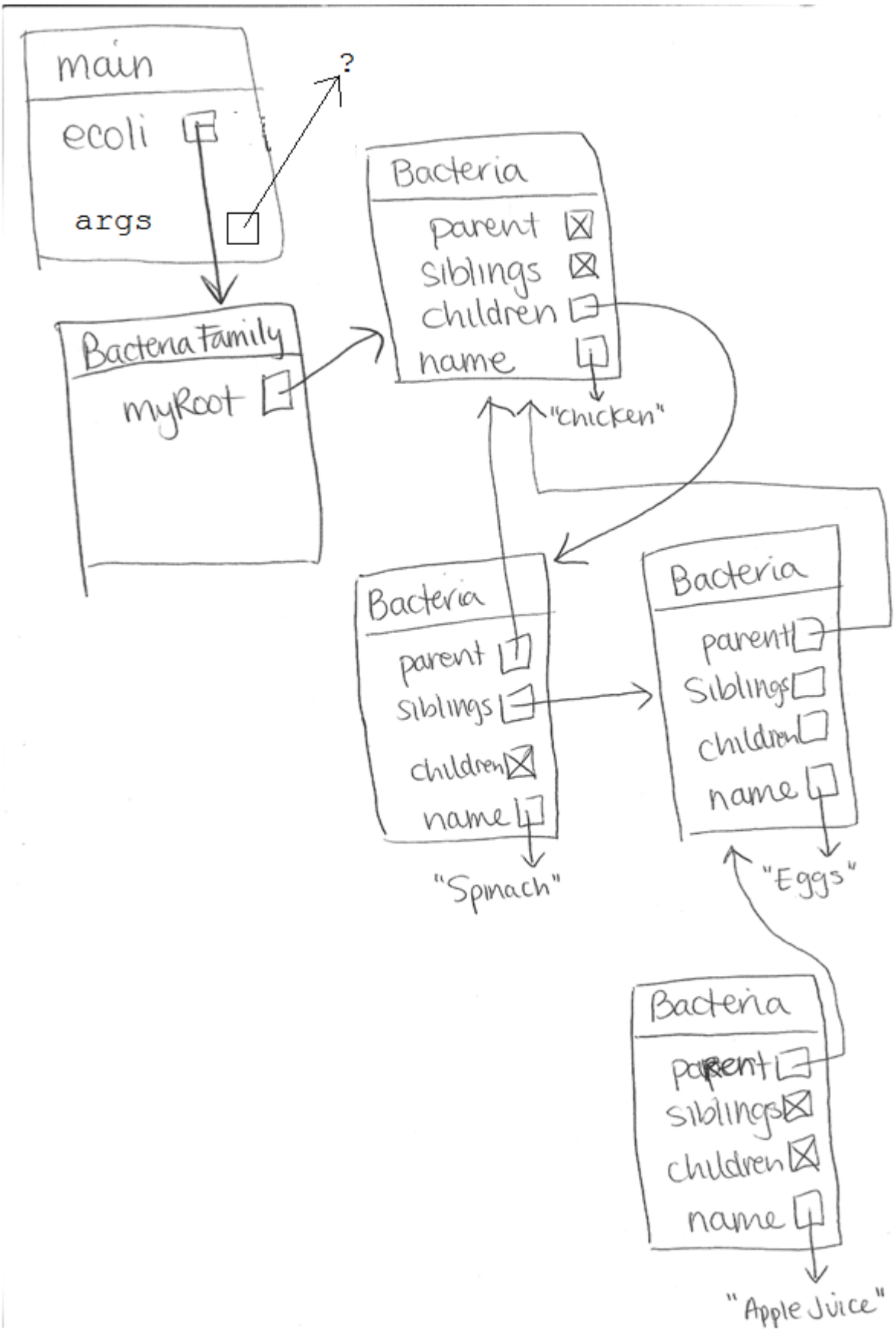
```
public void addEntry(Person personToAdd, Address
                                                addressToAdd){
     this.removeEntry(personToAdd);
     book.put(personToAdd, addressToAdd);
     ArrayList<Person> currentResidents =
                              residents.get(addressToAdd);
     if (currentResidents == null){
          currentResidents = new ArrayList<Person>();
     }
     currentResidents.add(personToAdd);
     residents.put(addressToAdd, currentResidents);
}
```

E	-2 points	Doesn't provide additional code for removeEntry()
P	-1	Does't remove from one hashMap/ArrayList
Q	-2	Doesn't implement getResidents correctly.

```
public void removeEntry(Person personToRemove){
     Address oldAddress = book.remove(personToRemove);
     if (oldAddress != null){
          ArrayList<Person> currentResidents =
                              residents.get(oldAddress);
          if (currentResidents != null){
               currentResidents.remove(personToRemove);
          }
     }
}
public ArrayList<Person> getResidents(Address address){
     return residents.get(address);
}
```

Birthday: Month: _____ Day: _____

Question 4 (6 points):



A	**-1 points**	**If Eggs points to spinach instead of Spinach pointing to Eggs.** Or both point to each other.
B	**-1 points**	**If parent pointers aren't correct**
C	**-0.5 point**	**If name is written inside the box for the bacteria object (they need to be their own objects)**
D	**-1 points**	**If they don't have a Bacteria family object with a myRoot**
F	**-0.5**	**Missing variables (no parent, siblings, children or name – up to 1 point)**
G	**-0.5**	**Missing X for null – max 1 point deduction.**
H	**-1**	**If children of chicken points to eggs (applies w/ two pointers)**
I	**-1**	**Missing pointer**
J	**-1**	**Sibling pointers – pointing to "children box in parent.**
K	**-1**	**Making boxes for newParent/newChild**
L	**-1**	**Extra boxes/objects (don't count twice with K).**
M	**-2**	**No boxes for objects (only rough tree sketch)**
N	**-1**	**Missing or incorrect child pointer to apple juice.**

There was a max of -3 on part A.

Question 4 continued:

A	**-0.5 points**	If they don't check if the current node is null.
B	**-1.5**	**Doesn't traverse the entire tree.** Doesn't traverse the siblings (recursively or using the stack) OR Doesn't travers the children (recursively or using the stack) But they still have the recursive calls to the siblings/children
C	**-0.5 point**	If they put nulls on the stack but don't check for them
D	**-0.5 points**	If they don't return something
E	**-1**	Uses helper method
F	**-1**	Infinite loop
G	**-0.5**	Called "find" as a Bacteria static method, rather than as a Bacteria Family
H	**-1**	Returned null w/out checking the entire tree.
J	**-0.5**	Doesn't return null If trying to find is NOT in the tree.
K	**-0.5**	Syntax error with .equals of "==" syntax error or other syntax errors.
L	**-1**	Like b, but if they removed an else they would have traversed the entire tree.
M	**-4**	Doesn't include code that would ever check the whole tree (missing children or siblings)
N	**-0**	Paren error
P	**-1**	Helper method causes infinite loop.

SOLUTION 1

Recursive Solution

```
// Uses Recursion
private static Bacteria find(String tryingToFind, Bacteria current) {
      if (current != null) {
            if (current.name.equals(tryingToFind)) {
                  return current;
            }
            Bacteria inSiblings = BacteriaFamily.find(tryingToFind,
                        current.siblings);
            if (inSiblings != null) {
                  return inSiblings;
            }
            Bacteria inChildren = BacteriaFamily.find(tryingToFind,
                        current.children);
```

```
                if (inChildren != null) {
                        return inChildren;
                }
        }
        return null;
    }
```

SOLUTION 2

```
    // Uses an explicit stack that MAY contain null
    private static Bacteria find2(String tryingToFind, Bacteria current) {
        Stack <Bacteria> fringe = new Stack<Bacteria>();
        if (current != null) {
            fringe.push(current);
            while(!fringe.isEmpty()){
                Bacteria tempBacteria = fringe.pop();
                // This version will put nulls on the list that later need to be
checked for.
                // For example when you get to the end of a sibling chain.
                if (tempBacteria == null) {
                    break;
                }
                if (tempBacteria.name.equals(tryingToFind)){
                    return tempBacteria;
                }
                fringe.push(tempBacteria.siblings);
                fringe.push(tempBacteria.children);
            }
        }
        return null;
    }
```

SOLUTION 3

```
// Uses an explicit stack that never contains null
    private static Bacteria find3(String tryingToFind, Bacteria current) {
        Stack <Bacteria> fringe = new Stack<Bacteria>();
        if (current != null) {
            fringe.push(current);
            while(!fringe.isEmpty()){
                Bacteria tempBacteria = fringe.pop();
                if (tempBacteria.name.equals(tryingToFind)){
                    return tempBacteria;
                }
                // This one makes sure we never put a null on the stack
                if (tempBacteria.siblings != null){
                    fringe.push(tempBacteria.siblings);

                }
                if (tempBacteria.children != null){
                    fringe.push(tempBacteria.children);
                }
            }
        }
        return null;
    }
```

Question 5: (5 points)

a) 0.5 points
Solution: YES
Only the node that is inserted is incorrect.
- 0.5 points if they miss this

b) 2 points
Solution: YES

There are 4 important cases. - 0.5 point for each case they miss

		They lose the points if they don't include the case listed below.
A	-0.5	• This half point is assumed unless they say anything incorrect about what happens for leaves.
B	-0.5	• If the node removed had one child, all nodes in that sub-tree are incorrect. (must be explicit about 1 child)
C	-0.5	• If the node removed had two children, ○ All nodes in the sub-tree of the in-order successor are now incorrect. (must be explicit about 2 children)
D	-0.5	• If the node removed had two children, ○ The in-order successor now has the wrong path. (must be explicit about 2 children)

c) 1.5 points

Solution:

A	1.5 point answer	N * H, where n is the number of nodes and h is the height of the BST.
B	1.0 point answer:	N^2, where n is the number of nodes, because the tree might be completely unbalanced.
C	0.5 point answer	If they say N*H but don't define the variables
D	0 point answer	If they say N^2 but they don't define the variable N.
E	0 point answer	Anything else.
F	1 point answer	NLog N where N is the number of nodes
G	0 point answer	N log N but they don't' define the variable N.
H	1.5 point answer	N log N where N is the number of nodes they are explicit about h= log N AND that they are assuming it is a balanced search BST. (This isn't a valid assumption but shows understanding of the general problem.)
I	1.5 point	The number of nodes rooted at T times the depth of each of those nodes

	answer	
J	0.5 point answer	O(n log n) for balanced tree, O(n^2) in the worst case, but n is not definied.

d) 1 point

Solution: YES this is possible.

If we just do one traversal of the tree. On the way down we could update the value in the node. Then when we go to the child it only has to look at its parent to calculate the correct path from room. The version in the handout it traversed all the way back up the tree.

1 point – all or nothing.

Birthday: Month: _____ Day: _____

Question 6:

a) Given the code in the handout, can the body of the method `reduceHelper` be written in each of these ways? Briefly explain.

Iterative:	Recursive:	Constructive:	Destructive:
☐ Yes ☐ No	☐ Yes ☐ No	☐ Yes ☐ No	☐ Yes ☐ No

Solution:

Yes, Yes, No, Yes (2 points)

We can't make it constructive because of how the method is called by reduce. Reduce doesn't set myRoot to point to the result of reduceHelper so we cannot make it constructive.

A	-0.5	If they Say "NO" to iterative
B	-0.5	If they Say "NO" to recursive
C	-1	If they Say "YES" to constructive
D	-0.5	If they Say "NO" to destructive

Code:

E	-2	Not setting next correctly (breaks the chain)
F	-2	Not setting prev correctly (breaks the chain)
G	-3	Not traversing all nodes
H	-1	No null check/Null Pointer stuff, but it is necessary.
I	-1	Not returning an IntListNode
J	-1	No form of exploring
K	-2	Trying to set "this"
L	-1	Compress check is wrong.
M	-2	Doesn't handle the case of needing to compress two nodes in a row.
N	-3	Compress into one node
O	-1	Tries to access head explicitly.

It was pointed out on the Lab Improvement form that the picture given in the problem was misleading. The picture included only forward arrows and therefore it was easy to forget to set the prev pointers. Sorry for this confusion, but in the diagram we tried to make it explicitly not a "real" diagram of the class. For example the class has a header node that was not represented and we didn't use any of the normal box and pointer notation or representations.

SOLUTION 1

```
// Recursive Destructive
        private IntListNode reduce() {
              if (this.next == null) {
              } else if (this.max + 1 == this.next.min) {
                    this.max = this.next.max;
                    this.next = this.next.next;
                    this.reduce();
              } else {
                    this.next.prev = this;
                    this.next.reduce();
              }
              return this;
        }
```

SOLUTION 2

```
        // Iterative Destructive
        private IntListNode reduce2() {
              IntListNode current = this;
              while (current.next != null){
                    if (current.max + 1 == current.next.max){
                          current.max = current.next.max;
                          current.next = current.next.next;
                    }
                    else {
                          // Save a pointer the "current"
                          IntListNode previous = current;
                          // Update the current pointer
                          current = current.next;
                          // Set the current point's prev to point to
                          // the previous "current"
                          current.prev = previous;
                    }
              }
              return this;
        }
```