Instructor: Edwin Liao

Midterm 2

July 30, 2014

Name:	
Student ID Number:	
Section Time:	
TA:	
Course Login: cs61bl-??	
Person on Left: Possibly “Aisle” or “Wall”	
Person on Right: Possibly “Aisle” or “Wall”	

- Fill out ALL sections on this page. (1 point)
- Do NOT turn this page until the beginning of the exam is announced.
- Once the exam begins, write your SID at the top of each page.
- You should not be sitting directly next to another student.
- You may not use outside resources other than your 2 pages of cheat sheets.
- You have 110 minutes to complete this exam.
- Your exam should contain 7 problems over 12 pages, including the reference sheet.
- This exam comprises 15% of the points on which your final grade will be based (45 points).
- Make sure to check for exam notes added to the board at the front of the room.
- Best of luck. Relax – this exam is not worth having a heart failure over.

1 Hash Functions (6 points)

For this problem, assume that the `String` class has the following `.equals` method:

```
public boolean equals(Object obj) {
    String str = (String) obj;
    if(obj == null || str.length() != this.length()) {
        return false;
    }
    for(int i = 0; i < str.length(); i++) {
        if(str.charAt(i) != this.charAt(i)) {
            return false;
        }
    }
    return true;
}
```

For each of the hash functions provided, explain whether the hash function could be a valid hash function for the `String` class. A hash function is considered invalid if it violates any of Java's specified `hashCode` rules (these rules were also explained in lab) that all hash functions are **required** to follow. If the hash function is valid, explain one flaw or disadvantage with the hash function. If the hash function is invalid, explain why. Note: the `String` class directly extends the `Object` class.

(a)
```
public int hashCode() {
    return 3;
}
```

(b)
```
public int hashCode() {
    return super.hashCode();
}
```

(c)
```
public int hashCode() {
    int h = 0;
    int len = length();
    for (int i = 0; i < len; i++) {
        h = 31 * h + charAt(i);
    }
    return h;
}
```

2 Heaps of Fun (3 points)

(a) Draw the binary tree that the binary min heap array below represents.

X	2	3	4	8	5	10	11
0	1	2	3	4	5	6	7

(b) Fill out the array below such that it represents the binary min heap from part (a) after removing the root node.

X						
0	1	2	3	4	5	6

3 National Heritage (8 points)

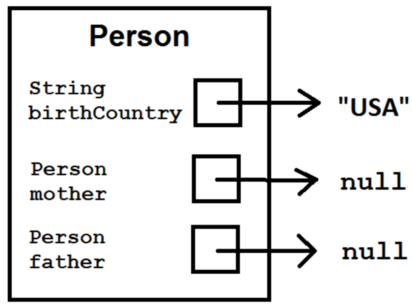
Consider the following `Person` class. Each person has a country of birth, a mother, and a father:

```
public class Person{
    String birthCountry;
    Person mother;
    Person father;
}
```

We're interested in knowing a given person's national heritage. For example, this person's national heritage is USA:

(Person object)

```
birthCountry = "USA";
mother = null;
father = null;
```



However, we might say something different if we knew more about that person's mother and father:

```
Germany        Argentina
       \      /
         USA
```

If we knew that the person's mother is from Germany and that the person's father is from Argentina (as shown above), we would instead say that the root person is half German and half Argentinian. We would then report the root person's heritage as .5 Germany and .5 Argentina.

However, if we knew even more about the family tree we might say something different again:

```
Germany        Brazil
      \        /
       Germany        Argentina
              \      /
                USA
```

Here, we would say the root person's national heritage is .25 Germany, .25 Brazil, and .5 Argentina. Notice that we do NOT say the root person is half German because in this example, the person's mother is actually only half German.

Fill in the method on the next page, which calculates the national heritage of a person. This method is written in the `Person` class. It returns a `HashMap`, where the keys in the map are the countries of national heritage and the values are the corresponding percentages, written as decimals (`Double` is a type of number in Java that can have decimals). Notice that this `HashMap` maintains the invariant that the sum of all of its values always equals 1.

You may assume for simplicity that every `Person` object either has both a mother and a father, or has neither (both are `null`). If a person's mother and father are both `null`, that means we do not know who that person's mother and father are, and so we say the person's national heritage is is entirely their own `birthCountry` (see the first example where the person's national heritage is USA).

You must use recursion to solve this problem. Follow the skeleton below.

```
public HashMap<String, Double> nationalHeritage() {
    HashMap<String, Double> heritages = new HashMap<String, Double>();

    // ---- base case here ----

    // ---- rest of code here ----

    return heritages;
}
```

4 Tree Order (6 points)

(a) Draw a **full** binary tree that has the following preorder and postorder. Each node should contain exactly one letter.

Preorder: C T U W X S A Z O
Postorder: W X U S T Z O A C

(b) What is the inorder of this tree?

5 Up (10 points)

```java
import java.util.*;

public class Tree {

    private TreeNode root;

    public Tree(Object rootItem) {
        this.root = new TreeNode(rootItem);
    }

    private class TreeNode {
        private Object item;
        private ArrayList<TreeNode> children;

        public TreeNode(Object inputObj) {
            this.item = inputObj;
            children = new ArrayList<TreeNode>();
        }

        public void addChild(Object childObj) {
            children.add(new TreeNode(childObj));
        }
    }
}
```

(a) Write a method in the provided `Tree` class (see previous page) that returns the items of the tree in reverse BFS order. That is, it returns an array list of items such that for all positive i, all items of nodes at depth i+1 come before all items of nodes at depth i in the returned array list. Items of nodes at the same depth level can be in any order in the list. While not necessary, you are allowed to write up to one helper method.

```
public ArrayList<Object> reverseBFS() {
```

(b) What is the big-O running time of your algorithm from part (a)? Your answer should be as tight of a bound as possible.

6 Analyze mergeAll (6 points)

Recall the homework assignment where you wrote a merge method that returned the result of merging two sorted linked lists. Given two sorted linked lists as input (one with l elements and another with r elements), your merge method took $O(l+r)$ time.

You also implemented a mergeAll method that takes in a list of sorted linked lists and outputs a single sorted linked list that contained all elements of all of the input linked lists. Below are two implementations of mergeAll. What are the big-Oh running times of each in terms of n (the number of lists in lists) and m (the number of elements in each list)? **Show your work and/or explain your answer.**

Your answers must be as tight as possible (i.e. $(nm)!^{(nm)!}$ is technically a valid upper bound, but not the one we are looking for) and reduced to simplest terms.

Assume that all linked lists in lists initially have the same number of elements, and that merge is implemented correctly.

(a)

```
public static AbstractListNode mergeAll(ArrayList<AbstractListNode> lists) {
    for (int i = 1; i < lists.size(); i++) {
        AbstractListNode myList = lists.get(i);
        AbstractListNode firstLL = lists.get(0);
        lists.set(0, merge(firstLL, myList));
    }
    return lists.get(0);
}
```

(b)

```
public static AbstractListNode mergeAll(ArrayList<AbstractListNode> lists) {
    while (lists.size() > 1) {
        List<AbstractListNode> newLists = new ArrayList<AbstractListNode>();
        for (int i = 0; i < lists.size(); i++) {
            int numLists = lists.size();
            if (numLists % 2 == 1 && i == numLists - 1) {
                newLists.add(lists.get(i));
            } else {
                newLists.add(merge(lists.get(i), lists.get(i + 1)));
                i++;
            }
        }
        lists = newLists;
    }
    return lists.get(0);
}
```

7 I Don't Even... (5 points)

```java
public class MyLinkedList {

    private ListNode head;

    public MyLinkedList(Object item) {
        head = new ListNode(item);
    }

    private class ListNode {
        private Object item;
        private ListNode next;

        public ListNode(Object inputItem) {
            this(inputItem, null);
        }

        public ListNode(Object inputItem, ListNode next) {
            this.item = inputItem;
            this.next = next;
        }
    }
}
```

Implement a `dontEven` method in the `MyLinkedList` class above that modifies the linked list such that it contains every other node of the original linked list. For example, if a linked list originally contains [5, 4, 2, 3, 1], the linked list after calling `dontEven` should contain [5, 2, 1]. Your method should work destructively and should not create any new objects. The last `ListNode` of a linked list has its `next` instance variable set to `null`.

```java
public void dontEven() {





}
```

Reference Sheet: ArrayList

Here are some methods and descriptions from Java's ArrayList<E> class API.

Return type and signature	Method description
boolean add(E e)	Append the specified element to the end of the list
boolean contains(Object o)	Returns true if this list contains the specified element
E get(int index)	Returns the element at the specified position in this list
Iterator<E> iterator()	Returns an iterator over the elements in this list in proper sequence
E remove(int index)	Removes the element at the specified position in this list
boolean remove(Object o)	Removes the first occurrence of the specified element from this list, if it is present
E set(int index, E element)	Replaces the element at the specified position in this list with the specified element (returns the previous element at this position)
int size()	Returns the number of elements in this list

Reference Sheet: Map<K, V>

Here are some methods and descriptions from Java's Map<K, V> interface API.

Return type and signature	Method description
V get(Object key)	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key
boolean isEmpty()	Returns true if this map contains no key-value mappings
V put(K key, V value)	Associates the specified value with the specified key in this map (returns the previous value associated with key, or null if there was no mapping for key)
V remove(Object key)	Removes the mapping for a key from this map if it is present
Set<K> keySet()	Returns a Set<K> of the keys contained in this map (the Set<K> class implements Iterable<K>)

Reference Sheet: `Stack<E>`

Here are some methods and descriptions from Java's `Stack<E>` class API.

Return type and signature	Method description
`boolean empty()`	Returns whether the stack is empty
`E peek()`	Returns the top element of the stack without removing it
`E pop()`	Removes and returns the top element of the stack
`E push(E item)`	Pushes an item onto the top of the stack and returns it

Reference Sheet: `Queue<E>`

Here are some methods from Java's `Queue<E>` interface API. Note: Java's `LinkedList<E>` class implements the `Queue<E>` interface.

Return type and signature	Method description
`E peek()`	Returns, but does not remove, the first element of the queue, or `null` if the queue is empty
`E poll()`	Returns and removes the first element of the queue, or `null` if the queue is empty
`boolean add(E item)`	Appends the item to the end of the queue and returns `true`

Reference Sheet: `String`

Here are some methods from Java's `String` class API.

Return type and signature	Method description
`char charAt(int index)`	Returns the `char` value at the specified index
`int length()`	Returns the length of this string