Midterm 2 Solutions

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;
}
```

Solution: This is a valid hash function, since `String`s that are `.equals` to each other will have the same hash code, and the hash function returns the same thing when called multiple times on an unchanged object. However, one flaw is there will be a lot of collisions (there will be a collision between **any pair** of `String`s).

Comments: Although many students stated that it is inefficient for all objects to map to the same hashcode (as a flaw), many did not explain the reasoning behind this. We were looking for: all objects would map to the same bucket, and thus would make searching for a given element similar to searching through a linked list, which is $O(n)$ where n is the total number of elements in the hash map.

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

Solution: This is not a valid hash function because `String`s that are `.equals` to each other will not have the same hash code. Instead, this hash function returns some integer corresponding to the `String` object's location in memory.

Comments: The `Object` class `hashCode` method is deterministic, since it depends on an object's location in memory, which doesn't change. Many students incorrectly stated that this `hashCode` method is nondeterministic.

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

Solution: This is the same as Java's `String` class `hashCode` function (minus memoization). It is a valid hash code because it satisfies the requirements of a hash function (see part a). However, one disadvantage of this hash function (in this case, compared to the hash function from part a) is that it takes longer to run for `String` objects that are longer. An (impossibly) ideal hash function would run in constant time.

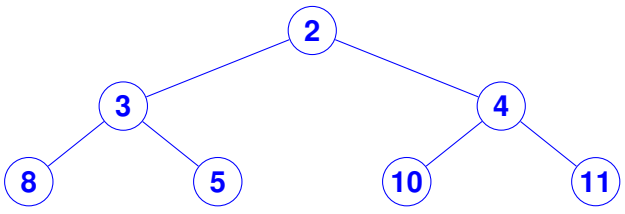
Comments: Adding a `char` to an `int` is perfectly valid. Also, integer overflow causes the integer value to go from `Integer.MAX_VALUE` all the way down to `Integer.MIN_VALUE`, which is fine. The return type of a hashcode is always an integer, which is stored with a fixed amount of space in memory (32 bits). Many students thought that a flaw was that more space was needed to store larger hashcode values; this isn't true because the size of a hashcode is the same regardless of the actual hashcode value. "There will be collisions" was not an accepted disadvantage because this is true of all hash functions.

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

Solution:



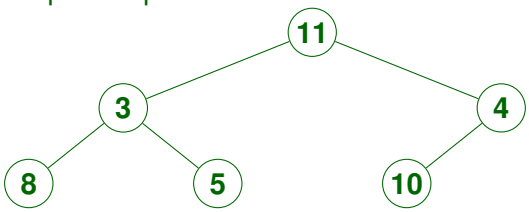
(b) Fill out the array below such that it represents the binary min heap from part (a) after removing the root node.

Solution:

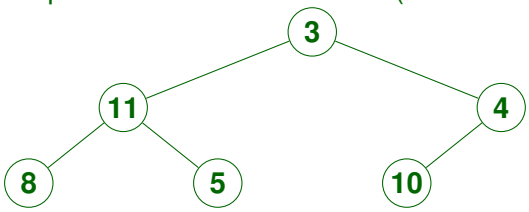
X	3	5	4	8	11	10
0	1	2	3	4	5	6

Comments:

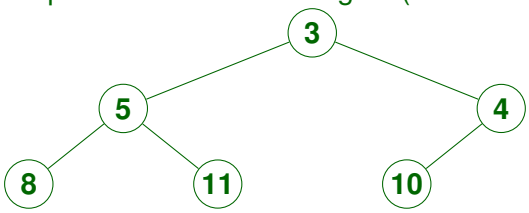
Step 1: Replace root node with last node



Step 2: Bubble down 11 node (switch with minimum of children)



Step 3: Bubble down 11 again (can't bubble down any more after this, so we're done)



3 National Heritage (8 points)

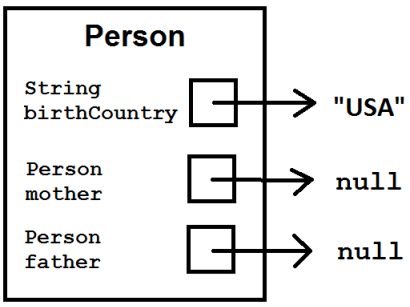
Consider the following `Person` class. Each person has a country of birth, a mother, and a father:

```
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.

Solution:

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

    // ---- base case here ----
    if(mother == null || father == null) {
        heritages.put(birthCountry, 1);
    }

    // ---- rest of code here ----
    else {
        HashMap<String, Double> motherHeritages = mother.nationalHeritage();
        HashMap<String, Double> fatherHeritages = father.nationalHeritage();
        Set<String> motherCountries = motherHeritages.keySet();
        Set<String> fatherCountries = fatherHeritages.keySet();

        Double heritageAmount;
        for(String motherCountry : motherCountries) {
            heritageAmount = 0.5 * motherHeritages.get(motherCountry);
            heritages.put(motherCountry, heritageAmount);
        }

        for(String fatherCountry : fatherCountries) {
            heritageAmount = 0.5 * fatherHeritages.get(fatherCountry);

            if(heritages.get(fatherCountry) != null) {
                heritageAmount += heritages.get(fatherCountry);
            }

            heritages.put(fatherCountry, heritageAmount);
        }
    }

    return heritages;
}
```

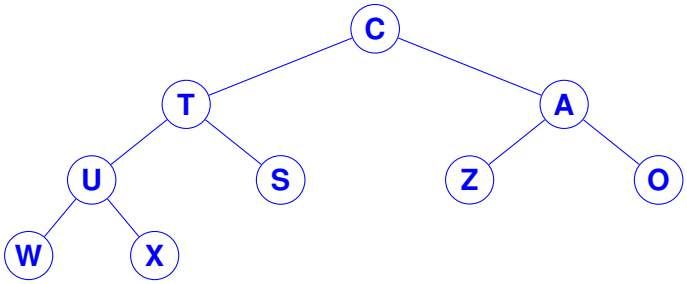
Comments: This problem tested your understanding of tree recursion and whether you knew how to use a `HashMap`. Other than `HashMap` misuse, the most common mistake was not properly handling the case where the mother and father shared heritage countries (you were supposed to add their heritages together).

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

Solution:



(b) What is the inorder of this tree?

Solution: W U X T S C Z A O

5 Up (10 points)

```
1  import java.util.*;
2
3  public class Tree {
4
5      private TreeNode root;
6
7      public Tree(Object rootItem) {
8          this.root = new TreeNode(rootItem);
9      }
10
11     private class TreeNode {
12         private Object item;
13         private ArrayList<TreeNode> children;
14
15         public TreeNode(Object inputObj) {
16             this.item = inputObj;
17             children = new ArrayList<TreeNode>();
18         }
19
20         public void addChild(Object childObj) {
21             children.add(new TreeNode(childObj));
22         }
23     }
24 }
```

(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.

Solution:

```
public ArrayList<Object> reverseBFS() {
    // Using linked list as queue
    LinkedList<TreeNode> fringe = new LinkedList<TreeNode>();
    Stack<Object> nodeItems = new Stack<Object>();

    // Check if root is null
    if(root == null) {
        return new ArrayList<Object>();
    } else {
        fringe.add(root);
    }

    // Add the items to a stack in BFS order
    while(fringe.peek() != null) {
        TreeNode currentNode = fringe.poll();
        nodeItems.push(currentNode.item);
        ArrayList<TreeNode> nodeChildren = currentNode.children;
        for(TreeNode nodeChild : nodeChildren) {
            fringe.add(nodeChild);
        }
    }

    // Remove items from the stack in reverse BFS order
    // And add them to an array list
    ArrayList<Object> returnList = new ArrayList<Object>();
    while(!nodeItems.empty()) {
        returnList.add(nodeItems.pop());
    }

    return returnList;
}
```

(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.

Solution: $O(n)$ where n is the number of nodes in the tree.

Comments: You had to correctly identify your own algorithm's running time. If you did not specify what your variables represent (e.g. n is the number of nodes in the tree), you lost a point.

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);
}
```

Solution:

Iteration	`myList` size	`firstList` size	Approx num steps (this iteration)	Approx num steps (running total)
1	m	m	$2m$	$2m$
2	m	$2m$	$3m$	$5m$
i	m	$i \times m$	$(i+1) \times m$	$(2+3+\cdots+(i+1)) \times m$
$n-1$	m	$(n-1) \times m$	nm	$(2+3+\cdots+n) \times m$ $= \frac{(n+2)(n-1)}{2} \times m$ $= \boxed{O(n^2 m)}$

(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);
}
```

Solution:

While-loop iteration	Number of for-loop iterations	Steps per call to `merge`	Approx num steps (this iteration)	Approx num steps (running total)
1	$\frac{n}{2}$	$2m$	nm	nm
2	$\frac{n}{4}$	$4m$	nm	$2nm$
i	$\frac{n}{2^i}$	$2^i \times m$	nm	$i \times nm$
$\log_2 n$	$\frac{n}{2^{\log_2 n}} = \frac{n}{n} = 1$	$2^{\log_2 n} \times m = nm$	nm	$(\log_2 n) \times nm$ $= \boxed{O(nm \log_2 n)}$

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

```
1  public class MyLinkedList {
2
3      private ListNode head;
4
5      public MyLinkedList(Object item) {
6          head = new ListNode(item);
7      }
8
9      private class ListNode {
10         private Object item;
11         private ListNode next;
12
13         public ListNode(Object inputItem) {
14             this(inputItem, null);
15         }
16
17         public ListNode(Object inputItem, ListNode next) {
18             this.item = inputItem;
19             this.next = next;
20         }
21     }
22 }
```

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`.

Solution:

```
public void dontEven() {
    ListNode currentNode = head;
    while(currentNode != null && currentNode.next != null) {
        currentNode.next = currentNode.next.next;
        currentNode = currentNode.next;
    }
}
```

Comments: You can't reassign `this` (the compiler will give an error). The `dontEven` method is in the `MyLinkedList` class, **not** the `ListNode` class. Lastly, you had to make sure your code would never throw a `NullPointerException` (this was by far the most common mistake for this problem).

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