CS61BL: Data Structures & Programming Methodology Summer 2014

Instructor: Edwin Liao

Final Exam

August 13, 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 3 pages of cheat sheets.
- You have 170 minutes to complete this exam.
- Your exam should contain 10 problems over 17 pages, including the reference sheet.
- This exam comprises 25% of the points on which your final grade will be based (75 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 Sometimes Sort of Sorted (8 points)

Let n be some large integer, $k < \log_2 n$. For each of the input arrays described below, explain which sorting algorithm you would use to sort the input array the fastest and why you chose this sorting algorithm. Make sure to state any assumptions you make about the implementation of your chosen sorting algorithm. Also specify the big-Oh running time for each of your algorithms in terms of k and n. Your big-Oh bounds should be simplified and as tight as possible.

(a) Input: An array of n `Comparable` objects in completely random order

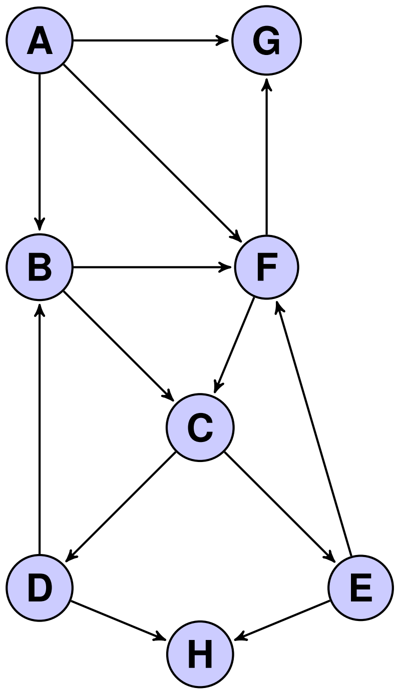
(b) Input: An array of n `Comparable` objects that is sorted except for k randomly located elements that are out of place (that is, the list without these k elements would be completely sorted)

(c) Input: An array of n `Comparable` objects that is sorted except for k randomly located pairs of adjacent elements that have been swapped (each element is part of at most one pair).

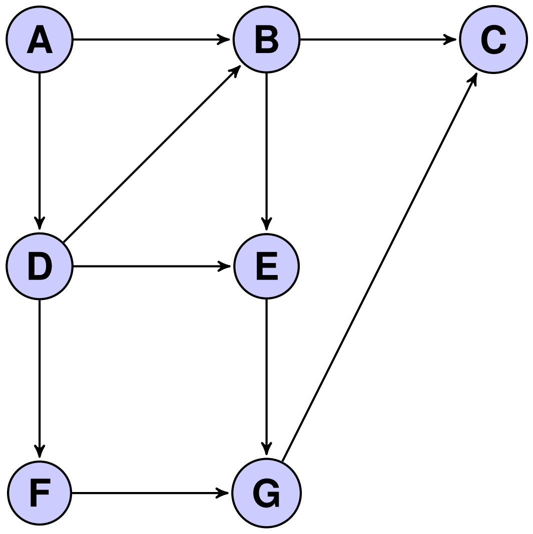
(d) Input: An array of n elements where all of the elements are random `int`s between 0 and k

2 The Unsocial Network (4 points)

(a) For each strongly connected component in the directed graph below, draw a circle around all of the vertices in that strongly connected component.



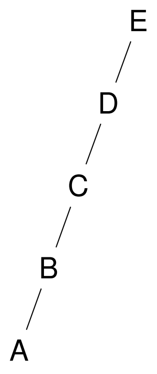
(b) Give a topological sort ordering of the vertices in the following directed acyclic graph (in other words, give a linearized ordering of the vertices in the graph).



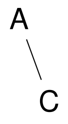
3 Balanced Search Trees (7 points)

For each part below, assume that nodes are ordered alphabetically by their letter (i.e. the value of "A" is less than the value of "B", which is less than the value of "C", etc...)

(a) Draw the splay tree that results after calling `find("A")` on the splay tree below.



(b) Add a "B" node to the AVL tree below by drawing it below. Then draw the tree that results after the AVL tree balances itself. Show all intermediary steps, if any.



(c) Draw the 2-3 tree that results after inserting the following elements in the given order:

```
1 2 3 4 5 6
```

(d) Draw the 2-3 tree that results after inserting the following elements in the given order:

```
4 2 1 3 6 5
```

4 Huffman Tree (6 points)

A table of characters and corresponding frequencies is provided below. Next to the table, draw a valid Huffman encoding tree. Then fill out the codemap table with each character's encoding according to your Huffman encoding tree.

Character	Frequency
`a`	5
`b`	2
`c`	8
`d`	7
`e`	3
`f`	7
`g`	1

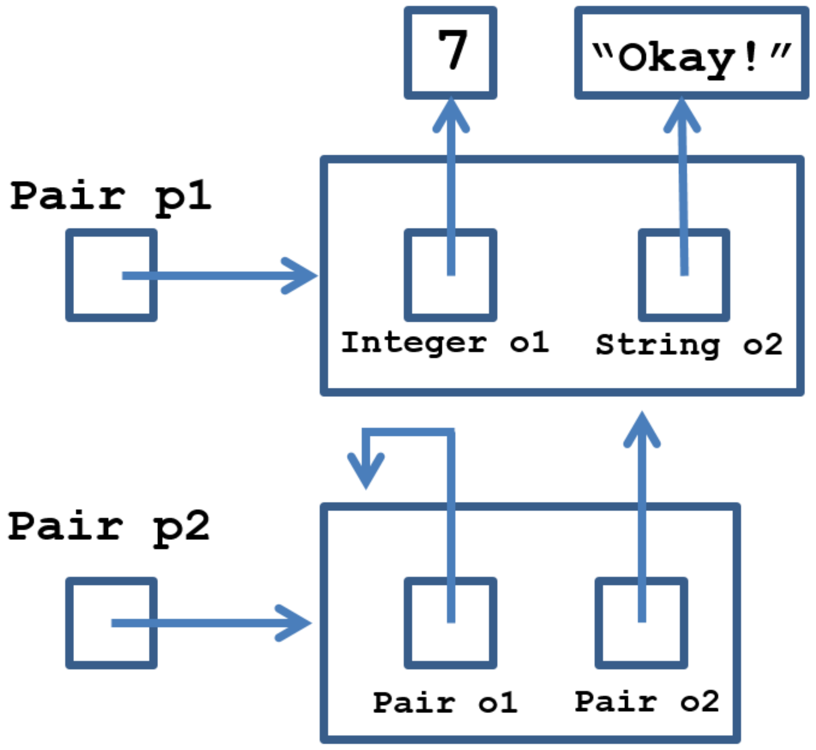
Draw Huffman encoding tree below:

Fill out the codemap below:

Character	Encoding
`a`	
`b`	
`c`	
`d`	
`e`	
`f`	
`g`	

5 `Pair` (5 points)

Write a program, `Pair.java`, that generates the box-and-pointer diagram shown below when run. Your program should include a class with a `main` method. In the diagram below, each object's static type is labeled next to the corresponding variable name. Each object's dynamic type is not shown.



6 Poorly Named Variables (6 points)

```java
public class V {
    private int VVVVV;

    public V(int VVVV) {
        VVVVV = VVVV;
    }

    private int VV() {
        return 0;
    }

    public double OOOO() {
        return 0;
    }
}
```

```java
public class O extends V {
    private int OOO;
}
```

For each of the following methods, explain whether the `O` class would still compile if the method is added to it. If the `O` class wouldn't compile, explain why. The `O` class may have additional methods / constructors. The `V` class does not.

(a) `public O() { }`

(b) `void OO() { OOO = VV(); }`

(c) `public boolean OOOO() { return false; }`

7 Dijkstra Analysis (8 points)

For each part below, all solutions given in terms of big-Oh must be simplified and as tight of an upper bound as possible. $|E|$ is the number of edges in the graph and $|V|$ is the number of vertices in the graph.

(a) Recall that Dijkstra's algorithm finds the shortest paths from some starting vertex s to all other vertices in the graph. In terms of big-Oh, $|E|$, and $|V|$, **how many times** does each of the following priority queue operations get called in one complete run of Dijkstra's algorithm?

i) `enqueue`:

ii) `dequeue`:

iii) `isEmpty`:

iv) `containsKey`:

v) `update` (updates a vertex's priority value in the priority queue):

(b) In lab, we analyzed the running time of Dijkstra's algorithm using a priority queue implemented with a binary min-heap. Now let's use a priority queue implemented with Java's `HashMap` that maps vertices to their priority values. Assuming that the hash map operations `put`, `get`, and `remove` take constant time, what are the new big-Oh running times (in terms of $|E|$ and $|V|$) of a single call to each of the following operations:

i) `enqueue`:

ii) `dequeue`:

iii) `isEmpty`:

iv) `containsKey`:

v) `update`:

(c) With our changes above, what is the new big-Oh running time of Dijkstra's algorithm (in terms of $|V|$ and $|E|$)? Show your work.

8 Buggy Priority Queue (10 points)

Joe Cool is implementing a priority queue with a binary min-heap using an array list (his code is provided below). However, his pair programming partner tells him that, with this implementation, calls to `dequeue` will not always work as intended.

```java
import java.util.ArrayList;

public class MyPriorityQueue {

    private ArrayList<Integer> binMinHeap;

    public MyPriorityQueue() {
        binMinHeap = new ArrayList<Integer>();
        binMinHeap.add(null);
    }

    // Removes and returns item in priority queue with smallest priority
    public Integer dequeue() {
        Integer toReturn = binMinHeap.get(1);
        binMinHeap.set(1, binMinHeap.remove(binMinHeap.size() - 1));
        bubbleDown(1);
        return toReturn;
    }

    // Adds item to priority queue
    public void enqueue(Integer item) {
        binMinHeap.add(item);
        bubbleUp(binMinHeap.size() - 1);
    }

    // Swaps the elements at index1 and index2 of the binary min heap
    private void swap(int index1, int index2) {
        int temp = binMinHeap.get(index1);
        binMinHeap.set(index1, binMinHeap.get(index2));
        binMinHeap.set(index2, temp);
    }

    // Bubbles up the element in the binary min heap array list at given index
    private void bubbleUp(int index) {
        while (index / 2 > 0 && binMinHeap.get(index) < binMinHeap.get(index / 2)) {
            swap(index, index / 2);
            index = index / 2;
        }
    }

    // Bubbles down the element in the binary min heap array list at given index
    private void bubbleDown(int index) {
        int n = binMinHeap.size();
        while (index * 2 < n && binMinHeap.get(index) > binMinHeap.get(index * 2)) {
            swap(index, index * 2);
            index = index * 2;
        }
    }
}
```

(a) Draw an example of a binary min-heap with exactly 5 nodes (either tree or array list form is fine) such that calling Joe's `dequeue` method on your binary min-heap produces an invalid binary min-heap.

(b) Which lines of code are buggy? ______________
Explain the bug.

(c) Rewrite the lines of code you specified in part b so that his min-heap will work as intended.

9 Every Other (8 points)

```java
public class MyLinkedList {

    private ListNode head;

    public MyLinkedList(ListNode inputHead) {
        head = inputHead;
    }

    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;
        }
        // There may be other methods not shown here
    }
    // There may be other methods not shown here
}
```

On the **next page**, write an `evenOdd` method in the `MyLinkedList` class above that destructively sets the linked list to contain every other linked list node of the original linked list, starting with the **first** node. Your method must **also** return a linked list that contains every other linked list node of the original linked list, starting with the **second** node.

Your method should work destructively and should not create any new `ListNode` objects. If a `MyLinkedList` contains zero elements or only one element, a call to `evenOdd` should return `null`. The last `ListNode` of each `MyLinkedList` has it's `next` instance variable set to `null`.

Example: If a `MyLinkedList` initially contains the elements [5, 2, 3, 1, 4], then a call to `evenOdd` should return a `MyLinkedList` with the elements [2, 1], and after the call, the original `MyLinkedList` should contain the elements [5, 3, 4]

```
public MyLinkedList evenOdd() {











}
```

10 `MemoryMap` (12 points)

You have been tasked with designing a `MemoryMap` class that implements the `Map<K, V>` interface and supports the following operations:

- `V put(K key, V value)`: Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.
- `V get(K key)`: Returns the value associated with the input `key`, or `null` if there is no mapping for the key.
- `V remove(Object key)`: Removes the mapping for the specified key from this map if present. Returns the previous value associated with the input `key`, or `null` if there was no mapping for `key`.
- `ArrayList<K> recent(int m)`: Returns an `ArrayList` of the `m` unique most recently accessed keys (sorted from most recently accessed to least recently accessed) that still have associated values in the map. A key `k` is considered accessed whenever `put` or `get` is called with `k` as the key. If there are fewer than `m` elements in the map, returns an `ArrayList` with all of the map's keys. Calls to `recent` should not modify the state of the `MemoryMap` in any way (so calling `recent` multiple times in a row without any other method calls in between should result in `recent` returning identical `ArrayList`s).

Example:

```
MemoryMap<String, String> map = new MemoryMap<String, String>();
map.put("A", "1");
map.put("B", "2");
map.recent(2); // Returned list contains: ["B", "A"]
map.get("A"); // Returns "1"
map.recent(2); // Returned list contains: ["A", "B"]
map.put("C", "3");
map.recent(3); // Returned list contains: ["C", "A", "B"]
map.remove("A");
map.recent(2); // Returned list contains: ["C", "B"]
```

(a) Explain in words how you would implement `MemoryMap` (including which data structures you would use) so that the operations listed on the previous page are time efficient. Space efficiency is not a concern. **Do not write any code.** Solutions that are as efficient as possible will receive full credit. Less efficient solutions may receive partial credit.

(b) For each of the methods listed below, explain how you would implement the method for `MemoryMap` and state your planned implementation's average case running time. State any assumptions you make about the average case running times of any data structures you would use in your implementation.

- `put`:

- `get`:

- `remove`:

- `recent`:

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>`)

(Blank page)