Final Exam
CS 1102 Computer Science 2

Spring 2018

Thursday May 10, 2018
Instructor Muller

KEY

Before reading further, please arrange to have an empty seat on either side of you. Now that you are seated, please write your name **on the back** of this exam.

This is a closed-notes and closed-book exam. Computers, calculators, and books are prohibited.

- Partial credit will be given so be sure to show your work.
- Feel free to write helper functions if you need them.
- **Please write neatly.**

Problem	Points	Out Of
1		5
2		4
3		8
4		10
Total		27

Final scores will be scaled to 40.

Part 1 (5 Points): Short Answer

For true/false questions, please circle the correct answer.

1. (1 Point) The implementation details of an ADT should be hidden from clients of the ADT. In a sentence, describe one reason why this is good practice.

   Answer:

   ```
   The implementation should be hidden

   1. so that the client can ignore it;

   2. so that the owner of the ADT is free to make changes to the implementation
      in the future.
   ```

2. (1 Point) In a sentence, what is the main risk of using mutable keys for ordered data structures?

   Answer:

   ```
   Mutation of a key can violate the invariants of the data structure,
   thereby destroying the data structure.
   ```

3. (1 Points) Let $A = \{a, b, c\}$. Show any total order on A.

   Answer:

   $$\{(a, a), (b, b), (c, c), (a, b), (b, c), (a, c)\}$$

4. (2 Points) Consider a hash table of size $m = 10$ using *double hashing* with hash function

   $$h(k, i) = (h_1(k) + ih_2(k)) \bmod 10$$

   and where $h_1(k) = k \bmod 10$ and $h_2(k) = k \bmod 6$. Is this a reasonable hash function? Why or why not?

   Answer: This is a bad hash function because it doesn't produce a complete probe sequence. E.g., for $k = 11$:

   $$\begin{aligned} h(11, 0) &= 1 \\ h(11, 1) &= (1 + 5) \bmod 10 = 6 \\ h(11, 2) &= (1 + 10) \bmod 10 = 1 \\ h(11, 3) &= (1 + 15) \bmod 10 = 6 \end{aligned}$$

Part 2 (4 Points): Binary Heaps

Let t be a *max binary heap*. Show **all** of the binary trees arising in using the max binary heap for an ascending-order heapsort.

```
      W
    /   \
   S     R  =>
  / \   /
 E   N A
```

Answer:

```
      W             A             S             S             A             R
    /   \         /   \         /   \         /   \         /   \         /   \
   S     R  =>   S     R  =>   A     R  =>   N     R  =>   N     R  =>   N     A
  / \   /       / \   .       / \   .       / \   .       / .   .       / .   .
 E   N A       E   N W       E   N W       E   A W       E   S W       E   S W

      E             N             A             E             A
    /   \         /   \         /   .         /   .         .   .
   N     A  =>   E     A  =>   E     N  =>   A     N  =>   E     N
  . .   .       . .   .       . .   .       . .   .
 R   S W       R   S W       R   S W       R   S W       R   S W
```

Part 3 (8 Points): ADTs

Let's say we have a *min priority queue* ADT in Java. Its specification might look like:

```
public interface MinPQ<T extends Comparable<T>> {
  boolean isEmpty();
  void insert(T item);
  T removeMin();
}
```

and its implementation might be something like

```
public class MinBinHeap<T extends Comparable<T>> implements MinPQ<T> { ... }
```

The priority queue ADT can be used to implement a last-in-first-out `Stack<T>` ADT.

```
public interface Stack<T> {
  boolean isEmpty();
  void push(T item);
  T pop();
}
```

Provide an implementation.

Answer:

```
public class StackC<T> implements Stack<T> {

  class Entry implements Comparable<Entry> {

    // fields
    T info;
    Integer count;

    //constructor
    Entry(T info, Integer count) {
      this.info = info;
      this.count = count;
    }

    public int compareTo(Item other) { return this.count.compareTo(other.count); }
  } // end of Entry

  // fields
  private int count;
  private MinPQ<Entry> minPQ;

  // one constructor
  public StackC() {
    this.minPQ = new MinPQ<Entry>();
    this.count = 0;                 // start at 0, decrement on push => LIFO
  }

  public boolean isEmpty() { return this.minPQ.isEmpty(); }
  public T pop() { return this.minPQ.removeMin().info; }
  public void push(T item) {
    this.minPQ.insert(new Entry(item, this.count--));
  }
}
```

Part 4 (10 Points): Full Binary Trees

Consider the ADT for full binary trees specified and partially implemented in Appendix A. A full binary tree is either empty or it is a leaf or it is an interior node with exactly two sub-trees. The implementation of `Empty` trees is completed in the appendix. Implement the following operations for interior `Node`s.

1. (1 Point) Write the function `public boolean isEmpty()`.

   Answer:

   ```
   public boolean isEmpty() { return false; }
   ```

2. (1 Point) Write the function `public boolean isLeaf()`.

   Answer:

   ```
   public boolean isLeaf() {
     return this.getLeft().isEmpty() && this.getRight().isEmpty();
   }
   ```

3. (2 Point) The height of a tree is the maximum depth of any leaf in the tree. Write the function `public int height()`.

   Answer:

   ```
   public int height() {
     if (this.isLeaf())
       return 0;
     return 1 + Math.max(this.getLeft().height(), this.getRight().height());
   }
   ```

4. (2 Point) A full binary tree is *well formed* in this implementation if it contains no null links. Write the function `public boolean wellFormed()`.

Answer:

```
public boolean wellFormed() {
  FullBinaryTree<T>
    left  = this.getLeft(),
    right = this.getRight();
  return left  != null && left.wellFormed() &&
         right != null && right.wellFormed();
}
```

5. (4 Point) Two full binary trees match if they have the same structure and if their corresponding `info` fields are `equal`. Write the function `public boolean matches(FullBinaryTree<T> other)`.

Answer:

```
public boolean matches(FullBinaryTree<T> other) {
  if (other.isEmpty() || !this.getInfo().equals(other.getInfo()))
    return false;

  if (this.isLeaf()) return other.isLeaf();

  boolean
    leftMatch  = this.getLeft().matches(other.getLeft()),
    rightMatch = this.getRight().matches(other.getRight());
  return leftMatch && rightMatch;
}
```

Appendix A: An ADT for Full Binary Trees

```
public interface FullBinaryTree<T> {

  T getInfo();
  FullBinaryTree<T> getLeft();
  FullBinaryTree<T> getRight();

  boolean isEmpty();
  boolean isLeaf();
  int height();
  boolean wellFormed();                      // no null left or right links
  boolean matches(FullBinaryTree<T> other);
}
```

```
public class Empty<T> implements FullBinaryTree<T> {

  public Empty() {}    // constructor

  public T getInfo()                  { throw new RuntimeException(); }
  public FullBinaryTree<T> getLeft()  { throw new RuntimeException(); }
  public FullBinaryTree<T> getRight() { throw new RuntimeException(); }

  public boolean isEmpty()    { return true; }
  public boolean isLeaf()     { return false; }
  public int height()         { return 0; }
  public boolean wellFormed() { return true; }

  public boolean matches(FullBinaryTree<T> other) { return other.isEmpty(); }
}
```

```
public class Node<T> implements FullBinaryTree<T> {

  private T info;
  private FullBinaryTree<T> left, right;

  public Node(T info, FullBinaryTree<T> left, FullBinaryTree<T> right) {
    this.info = info;
    this.left = left;
    this.right = right;
  }
  public Node(T info) { this(info, new Empty(), new Empty()); }

  // getters
  public T getInfo()                  { return this.info; }
  public FullBinaryTree<T> getLeft()  { return this.left; }
  public FullBinaryTree<T> getRight() { return this.right; }

  public boolean isEmpty()    { YOUR CODE }
  public boolean isLeaf()     { YOUR CODE }
  public int height()         { YOUR CODE }
  public boolean wellFormed() { YOUR CODE }
  public boolean matches(FullBinaryTree<T> other) { YOUR CODE }
}
```