Prelim 2 Solution

CS 2110, 15 November 2018, 7:30 PM

	1	**2**	**3**	**4**	**5**	**6**	**Total**
Question	Name	Short answer	Heaps	Tree	Sorting	Graph	
Max	1	23	16	20	20	20	100
Score							
Grader							

The exam is closed book and closed notes. Do not begin until instructed.

You have **90 minutes**. Good luck!

Write your name and Cornell **NetID**, **legibly**, at the top of **every** page! There are 6 questions on 9 numbered pages, front and back. Check that you have all the pages. When you hand in your exam, make sure your pages are still stapled together. If not, please use our stapler to reattach all your pages!

We have scrap paper available. If you do a lot of crossing out and rewriting, you might want to write code on scrap paper first and then copy it to the exam so that we can make sense of what you handed in.

Write your answers in the space provided. Ambiguous answers will be considered incorrect. You should be able to fit your answers easily into the space provided.

In some places, we have abbreviated or condensed code to reduce the number of pages that must be printed for the exam. In others, code has been obfuscated to make the problem more difficult. This does not mean that it's good style.

Academic Integrity Statement: I pledge that I have neither given nor received any unauthorized aid on this exam. I will not talk about the exam with anyone in this course who has not yet taken Prelim 2.

(signature)

1. Name (1 point)

Write your name and NetID, **legibly**, at the top of **every** page of this exam.

2. Short Answer (23 points)

(a) True / False (7 points) **Circle** T or F in the table below.

(a)	T	F	Enums implement Comparable and by default method compareTo() uses the order in which the names were declared (and not alphabetical order). True
(b)	T	F	In a GUI, JPanel and Box are the same except that in a JPanel the added components appear horizontal (in a row) while in a Box they are vertical (in a column). False. For example, a Box can be horizontal or vertical.
(c)	T	F	One interface cannot extend another. False: e.g. Java interface List extends interface Collection.
(d)	T	F	$f(n)$ is $O(g(n))$ if and only if there exist constants $c > 0$ and $N \geq 0$ such that $f(N) \leq c * g(N)$. False, see definition of Big-O from the slides.
(e)	T	F	The in-order and post-order representations determine a binary tree uniquely. True.
(f)	T	F	A function $f(m)$ that is $O(m)$ is also $O(5 * m)$ and $O(m^{16})$. True. Look at the definition of Big-O.
(g)	T	F	The worst-case time for adding an element to the end of an ArrayList is amortized time O(1). True. The API documentation for ArrayList states this, and we went over this in lecture. See JavaHyperText: amortize.

(b) Complexity (5 points) For each of the functions f below, state a function g such that f is $O(g)$ where $O(g)$ is as simple and tight as possible. For example, one could say that $f(n) = 2n^2$ is $O(n^3)$ or $O(2n^2)$, but the best answer is $O(n^2)$. Remember that $log(a^b) = b * log(a)$ and n and m are non-negative integers.

1. $f(n) = 2110 + 3n^6$ n^6
2. $f(n) = 6n + 2^p$ n
3. $f(n) = n * log(n) + 10 * log(log(n))$ $n * log(n)$
4. $f(n) = \frac{n^4 - 2n^3}{5n}$ n^3
5. $f(n, m) = \sum_{h=0}^{n} log(m^h)$ $n^2 * log(m)$

(c) Iterable (5 points) Complete the constructor and method next() of private class TableIterator. It enumerates the elements of b in class Table in column-major order. In other words, it enumerates the first column from first to last element, then the second column from first to last element, etc.

```
public class Table<E> implements Iterable {
    private E[][] b; // b is square: number of rows and columns are the same

    /** = an enumerator of elements in this table. */
    public Iterator<E> iterator() {
        return new TableIterator();
    }
```

```
  private class TableIterator implements Iterator<E> {
     int r; // 0 <= r < b.length and 0 <= c <= b.length
     int c; // if c < b.length, b[r][c] is next element to enumerate
            // if c = b.length, there are no more to enumerate

    /** Constructor: an iterator over elements of b, in column-major order */
    public TableIterator() {
         // TODO        r= 0; c= 0;
    }

    /** = there is another element to enumerate */
    public @Override boolean hasNext() { return c < b.length; }

    /** Return the next element to enumerate.
     *  Throw a NoSuchElementException if there is no next element. */
    public @Override E next() {
         if (!hasNext()) throw new NoSuchElementException();
         // TODO
         E res= b[r][c];
         r= r+1;
         if (r == b.length) {
              c= c+1;
              r= 0;
         }
        return res;
    }
  }
}
```

(d) Hashing (6 points) Consider a hash table of size 5, with elements numbered in 0..4. The hash function being used is:

$$H(k) = 2k \bmod 5.$$

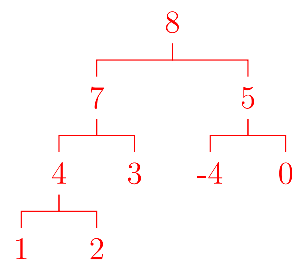
Consider inserting these values into it, in order: 5, 7, 10, 8, 2.

To the left, draw the table after inserting the values, using open addressing with linear probing. To the right, draw the table after inserting the values (into an empty table) using chaining. Draw the linked list as simply and as clearly as possible.

```
  5   10   8   2   7            *   *   *    *    *   (order of values in linked
                                |   |             |         lists does not matter)
                               10   8             7
                                |                 |
                                5                 2
```

3. Heaps (16 Points)

(a) (8 points) Draw the max-heap formed by inserting the following integers in order: [1, 7, -4, 3, 4, 0, 5, 2, 8]



(b) (8 points) Implement method `change` in class `MaxHeap` below. Assume that `bubbleUp` and `bubbleDown` are implemented according to the specifications. You can use only ONE method call to `bubbleUp` OR `bubbleDown`, i.e. you cannot call both or call one several times. Make sure the class invariant is maintained.

```
   /** A max-heap for integers. */
public class MaxHeap {
      /** c[0..size-1] represents a complete binary tree. c[0] is the root.
      *   For each h, c[2h+1] and c[2h+2] are the left and right children of c[h],
      *   and c[h] >= max(c[2h+1], c[2h+2]).
      *   If h != 0, c[(h-1)/2] (using integer division) is the parent of c[h]. */
      protected int[] c;
      protected int size;

      /** Bubble c[k] up the heap to its right place.*/
      void bubbleUp(int k) { ... }

      /** Bubble c[k] down the heap to its right place.*/
      void bubbleDown(int k) { ... }

      /** Call the largest value in the heap lv.
        * Replace lv in the heap by v and return lv.  */
      public int change(int v) {
            // TODO
            int lv= c[0];
            c[0]= v;
            bubbleDown(0);
            return lv;
       }
}
```

4. Trees (20 Points)

(a) (2 points) What is the worst case time complexity for inserting a value into a balanced Binary Search Tree with n nodes? O(log n)

(b) (4 points) To the left, draw the BST created by inserting the following values, in that order, into an empty BST. To the right, draw the max-heap created by inserting the values, in that order, into an empty max-heap.

72, 51, 83, 81, 52

```
    72               83
   /  \             /  \
51      83        81     72
  \    /         /  \
  52  81       51    52
```

(c) (4 points) Suppose a tree has this postorder sequence and inorder sequence:

postorder: 2 3 1 9 7 10 8 4
inorder: 2 1 3 4 7 9 8 10

What is the root of the tree? Which values are in its left subtree, and which values are in its right subtree?

Postorder tell us that the root is 4. Inorder then tells us that the left subtree contains 2, 1, 3 and the right subtree contains 7, 8, 9, and 10

Name: NetID:

(d) (10 points)

Class Node, to the right, is the typical class for a node of a binary tree. Method sum is declared, but we don't show its implementation.

A binary tree is a *sumTree* if either

- It is a leaf (i.e. its size is 1) OR
- (1) It is not a leaf,
  (2) Its left and right subtrees, if they exist, are sumTrees, and
  (3) The root's val field is the sum of all the val fields in its left and right subtrees (if they exist).

The first tree to the right is not a sumTree (because the subtree with root 3 is not a sumTree). The second tree *is* a sumTree.

Complete function isSumTree, below. It is declared within class Node. Don't be concerned with efficiency.

```
public class Node {
  private int val;
  private Node left;
  private Node right;
  ...
  /** Return the sum of the
    * values in the tree
    * whose root is this node. */
  public int sum() { ... }
  ...
}
```

```
     7            8
    / \          / \
   2   3        2   3
        \            \
         2            3
```

```
/** Return true iff the tree whose root is this Node is a SumTree. */
public boolean isSumTree() {
   if (left == null  &&  right == null) return true;
   if (left != null  &&  !left.isSumTree()) return false;
   if (right != null  &&  !right.isSumTree()) return false;
   int v= (left == null ? 0 : left.sum()) +      // OR do simply
          (right == null ? 0 : right.sum());     //    int v= sum() - val;
   return val == v;
}
```

5. Sorting (20 Points)

(a) (4 points) Consider the following class Author. An online literary database requires authors to be listed in decreasing popularity, as measured by the numbers of books they have **sold**. If two authors have sold the same number of copies, then list the one who has written more books first.

Complete method compareTo(...).

```
/** An instance represents a comparable author object*/
public class Author implements Comparable<Author> {
   private String name;
   private int copiesSold;
   private int booksWritten;
    ...
   @Override public int compareTo(Author ob) {  // There are other solutions
     if (ob.copiesSold == copiesSold) return ob.booksWritten - booksWritten;
     return ob.copiesSold - copiesSold;
    }
}
```

(b) (8 points) Using method compareTo, complete method selectionSort(), below. Be careful. This is selection sort, but slightly different from what we showed in lecture. Read the loop invariant carefully. Complete the for-loop header and the implementation of the high-level statement that says what the for-loop repetend does.

```
public void selectionSort(Author[] b) {
     // inv: b[i+1..] is sorted and b[0..i] <= b[i+1..]
     for ( int i= b.length-1; 0 <= i; i= i-1 ) {
         // Swap b[i] with the maximum value of b[0..i]
         int k= 0; int j= 0
         // inv: b[j] is the maximum of b[0..k-1]
         for (k= 1; k <= i; k= k+1) {
             if (b[k].compareTo(b[j] > 0) j= k;
         }
         Author t= b[j]; b[j]= b[i]; b[i]= t;
       }
}
```

(c) (2 points) What is the worse-case time and expected time of selection sort? $O(n^2)$, $O(n^2)$

(d) (6 points) State the tightest expected *space* complexity of mergesort, quicksort, and insertionsort. For quicksort, assume it is the version that reduces the space as much as possible.

mergesort: $O(n)$ quicksort: $O(log\ n)$ insertionsort: $O(1)$

6. Graphs (20 Points)

(a) (3 points) Topological sort of a directed graph g is written abstractly below —it changes g and puts the nodes into ArrayList res in topological order. State the condition under which this algorithm does not produce the nodes in topological order and identify which statement or expression in the algorithm will fail.

Doesn't work if g is cyclic. If that case, at some point a node n with indegree 0 won't be found..

```
ArrayList<Node> res= new ArrayList<Node>();
while (g.size() > 0) {
    Let n be a node of g with indegree 0;
    res.add(n);
    Remove n and all edges connected to it from g;
 }
```

(b) (4 points) Below, state the theorem that is proved from the invariant of Dijkstra's shortest-path algorithm. Then, state what it tells us if the Frontier set contains these pairs of cities and distances: (Austin, 574), (New York, 3000), (Helena, 100), (Dallas, 120)
Theorem. Consider a node n in the Frontier set with minimum distance $d[n]$ over all nodes in the Frontier set. Then $d[n]$ is indeed the shortest distance from the start node to n. For the given Frontier set, the shortest distance from the start node to Helena is 100.

(c) (6 points) Write the body of the following algorithm with pseudo-code. Make it recursive. Leave the notion of "visiting" and "visited" abstract.

```
/** Visit all nodes reachable along unvisited paths from u.  */
public static void dfs(Node u) {
   if (u is already visited) return;
   visit u;
   for each neighbor v of u   // it is alright, but not necessary,
      dfs(v);                 // to call dfs(v) only if v is not visited.
 }
```

(d) (7 points) We stated two properties of a spanning tree:

(1) A spanning tree of a graph is a maximal set of edges that contains no cycle.
(2) A spanning tree of a graph is a minimal set of edges that connects all nodes.

Based on (2), we wrote this abstract algorithm for creating a spanning tree:

```
(A2) Start with all nodes and no edges. While the nodes are not all
connected, add an edge that connects two unconnected components,
i.e. that does not introduce a cycle.
```

(i) (3 points) Below, write the abstract algorithm that results from using property (1):

```
(A1) While there is a cycle, pick an edge of the cycle and throw it out.
```

(ii) (4 points) Each property (1) and (2) inspires an abstract algorithm that has a loop. Write down the number of iterations each loop takes (as a function of the number n of nodes and the number e of edges). Based on these numbers of iterations, state in general which of these two algorithms is preferred.

For a connected undirected graph with n nodes, a spanning tree has $n-1$ edges. Therefore, algorithm A1 has to throw out $e-(n-1)$ edges. So $e-(n-1)$ iterations are performed. Since e can be $O(n*n)$, that's $O(n*n)$ iterations in the worst case, and this algorithm is not preferred. Algorithm A2 has to add $n-1$ edges, so the number of iterations is always $n-1$. Thus, this algorithm is preferred.