Prelim 2 Solution

CS 2110, 15 November 2018, 5:30 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 the compareTo() method uses the alphabetic ordering of the constant names. False: The ordering is the declaration order.
(b)	T	F	The Border Layout Manager has four component placements named top, bottom, left, and right. False: There are five components: North, East, South, West, Center
(c)	T	F	Interface I1 can extend interface I2 without implementing all the methods in I2. True: Subinterfaces do not need to implement superinterface methods.
(d)	T	F	$f(n)$ is $O(g(n))$ if and only if there exist $c > 0$ and $N \geq 0$ such that $f(N) \leq c * g(N)$. False, see definition of Big-O from the slides.
(e)	T	F	An algorithm's time complexity can be both $O(1)$ and $O(n^2)$. True. Any algorithm that is $O(1)$ is also $O(n^2)$.
(f)	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: ?amortization?.
(g)	T	F	The pre-order and post-order representations determine a binary tree uniquely. False: Counter-example A – B where B can be the left or right node

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

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

```
public class Matrix<E> implements Iterable {
    private E[][] b;

    /** = an enumerator of the matrix elements. */
    public Iterator<E> iterator() {
        return new MatrixIterator();
    }
    private class MatrixIterator implements Iterator<E> {
       int i; // 0 <= i <= b.length and 0 <= j < b[i].length
       int j; // if i < b.length, b[i][j] is next element to enumerate
              // if i = b.length, there are no more to enumerate

      /** Constructor: an iterator over elements of b, in row-major order */
      public MatrixIterator() {
           // TODO        i= 0; j= 0;
      }

      /** = there is another element to enumerate */
      public @Override boolean hasNext() { return i < 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[i][j];
           j= j+1;
           if (j == b[i].length) {
                i= i+1;
                j= 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:

$$H(k) = 3k \text{ mod } 5.$$

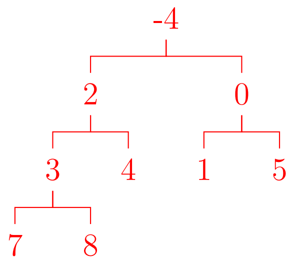
Consider inserting these values into it, in order: 5, 7, 10, 8, 2.

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

3. Heaps (16 Points)

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



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

```
   /** A min-heap for integers. */
public class MinHeap {
      /** 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] <= min(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) { ... }

      /** Poll the next element from the heap, add e to the heap, and
        * return the polled element.  */
      public int pollAndInsert(int e) {
            // TODO
            int ret= c[0];
            c[0]= e;
            bubbleDown(0);
            return ret;
       }
}
```

4. Trees (20 Points)

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

76, 51, 88, 86, 66

```
   76                 88
  /  \               /  \
51    88          86     76
 \   /           /  \
 66 86        51     66
```

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

postorder: 3 5 4 8 7 10 9 6
inorder: 3 4 5 6 7 8 9 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 6. Inorder then tells us that the left subtree contains 3, 4, and 5 and the right subtree contains 7, 8, 9, and 10

(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 static 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 n is a SumTree.
  * Precondition: n is not null. */
public static boolean isSumTree(Node n) {
   if (n.left == null  &&  n.right == null) return true;
   if (n.left != null  &&  !isSumTree(n.left)) return false;
   if (n.right != null  &&  !isSumTree(n.right)) return false;
   int v= (n.left == null ? 0 : n.left.sum()) +      // OR do simply
          (n.right == null ? 0 : n.right.sum());     //    int v= n.sum - n.val;
   return n.val == v;
}
```

5. Sorting (20 Points)

(a) (4 points) Consider the following class File. A file processing system requires files to arrive for processing in decreasing order of priority. If two files have the same priority, process the larger one first. Complete method compareTo(...).

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

(b) (8 points) Using method compareTo(), complete method insertionSort(), below. Be careful, this is insertion 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.

We deducted points for an inner loop that always iterates up to k = b.length-1. The loop MUST stop as soon as the initial b[i] is in its final position. No regrade request on this will get points back.

```
   public void insertionSort(File[] b) {
        // invariant: b[i+1..] is sorted
        for ( int i= b.length-1; 0 <= i; i= i-1 ) {
            // Push b[i] up to its sorted position in b[i..]
            int k= i;
            // inv: b[i..] is sorted except that b[k] could be > b[k+1]
            while (k < b.length-1  &&  b[k].compareTo(b[k+1]) > 0) {
                File t=  b[k+1];  b[k+1]= b[k];  b[k]= t;
                k= k+1;
            }
       }
   }
```

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

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

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

6. Graphs (20 Points)

(a) (3 points) Topological sort of a DAG g can be written abstractly as shown below (changing g), putting the nodes in topological order into ArrayList res. State the condition under which the topological ordering is unique, i.e. there exists only one topological ordering.

At each iteration, exactly one node has indegree 0.

```
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), (Socorro, 14), (Fort Bliss, 3), (Horizon City, 20)

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 Fort Bliss is 3.

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

```
/** Visit all nodes reachable along unvisited paths from n.
 *  Precondition: n is unvisited. */
public static void dfs(Node n) {
   visit n;
   for each neighbor v of n
      if (v is unvisited) dfs(v);
 }
```

(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 (1), we wrote this abstract algorithm for creating a spanning tree:

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

(i) (3 points) Below, write the abstract algorithm that results from using property (2):

```
(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.`

(ii) (4 points) Each of the abstract algorithms (1) and (2) 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.