Name: NetID:

CS2110 SOLUTION Final Exam

Sunday, 21 May 2017, 14:00–16:30

	1	**2**	**3**	**4**	**5**	**6**	**Total**
Question	Short Answer	Object Oriented	Algorithms	Data Structures	Graphs	Concurrency	
Max	20	18	14	28	12	8	100
Score							
Grader							

1. Short Questions (20 points)

(a) Topological sort (4 points)

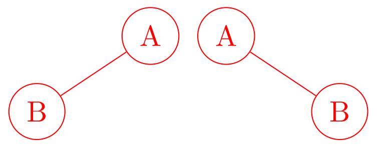
Topological sort of a DAG can be written abstractly as shown below, putting the nodes in topological order into an ArrayList. State the condition under which the topological ordering is unique, i.e. there exists only one topological ordering.

```
Set<Node> g1= a copy of the nodes of graph g (so changing g1 will not change g);
ArrayList<Node> res= new ArrayList<Node>();
while (g1.size() > 0) {
    Let n be a node of g1 with indegree 0;
    res.add(n);
    Remove node n and all edges connected to it from g1;
}
```

Answer: At each iteration, only one node has indegree 0 – a *unique* topological sort means that there can only be one possible ordering. Thus, there can only be one node to pick in each iteration, or one node with no arrows in. Therefore, only one node can have indegree of 0 at each step of the iteration.

(b) Tree traversal (4 points)

Show that you cannot uniquely construct a binary tree from its preorder and postorder traversals by drawing two *different* binary trees that have the same preorder and postorder traversals. Hint: Make the trees as small as possible.



(c) Hashing with linear probing (4 points)

Consider hashing with linear probing using an array $b[0..6]$ to maintain a set of integers. Do not be concerned with the load factor. The hash function to be used is just the integer itself: $hashcode(i) = i$. Draw array b after these values have been added: 11, 13, 20, 14.
Answer – [20, 14, null, null, 11, null, 13]

1. Add 11: 11 % 7 = 4, so hash into index 4 – [null, null, null, null, 11, null, null]
2. Add 13: 13 % 7 = 6, so hash into index 6 – [null, null, null, null, 11, null, 13]
3. Add 20: 20 % 7 = 6, conflict!
Use linear probing to find the next null index, so we use index 0 – [20, null, null, null, 11, null, 13]
4. Add 14: 14 % 7 = 0, conflict again!
Use linear probing again to put 14 into index 1 – [20, 14, null, null, 11, null, 13]

(d) Comparable (4 points)

Function *compareTo*, which is defined in interface *Comparable*, need not return -1, 0, or 1 but can return any integer. This is seen in the specification of *compareTo* in class *Pt*, given below.

(1) Fill in whatever implements clause is required in the class definition below. (2) Complete the two return statements in the body of *compareTo* below. Do not use conditional expressions.

```
/** An instance represents a point in the plane.
  * Points are ordered by x-values, e.g. (5, 8) comes before (6, 3), but on
  * y-values if the x-values are the same, e.g. (5, 3) comes before (5, 4). */

public class Pt implements Comparable <Pt> {
  public int x; // Above, type parameter Pt is needed because the type
  public int y; // of parameter p of function compareTo is Pt.

  /** Return a negative number, 0, or positive number depending on whether
    * this point comes before p, is the same as p, or comes after p. */
  public int compareTo(Pt p) {
     if (x != p.x) return x – p.x ;
     return y – p.y ;
  }
  ...
}
```

(e) Exception handling (4 points)

Below is a scheme for a try-statement, with two catch clauses. $S1$, $S2$, and $S3$, represent sequences of statements. Explain how this try-statement is executed. Execute S1. (1) If S1 does not throw an exception, execution of the try-statement is finished. (2) If S1 throws an ArithmeticException, S2 is executed; if S1 throws some other RuntimeException, S3 is executed. (3) If S1 throws an exception that is not a RuntimeException the exception is thrown out to the call of the method in which the try-statement appears. Note that S2 and S3 are not in the try-block, so if they throw an exception, it is not caught by the catch clauses in this try-statement.

```
try { S1 }
catch (ArithmeticException e) { S2 }
catch (RuntimeException e) { S3 }
```

2. Object-Oriented Programming (18 points)

You and your pals maintain online collections of pics (pictures), each of which is tagged with information like the date and the people in the pictures. Egotistical YOU would like to search through your pals' pictures, their pals' pictures, etc., and get hold of all pictures that have you in them.

You will write a program to get those pics in three steps. Step (b) will use the method from step (a), and step (c) uses (b).

Notes: Just take one method at a time, read its specification carefully, and implement it. The only loops you need to write are for-each loops. Here are two classes that you will use.

```
/** An instance represents a person,
  * their pals, and their pics. */
public class Person {
    public String name;
    public Set<Person> pals;
    public Set<Pic> pics;
}
```

```
/** An instance is a picture with tags. */
public class Pic {
    public JPG image; // you won't need this
    public Set<String> tags;
}
```

(a) Gathering who's pics (6 points) Write the body of the following procedure:

```
/** Add to setPics all of p's pics that have tag who and are not already in setPics.
  * Precondition: p, who, and setPics are not null. */
public static void gatherPics(Person p, String who, Set<Pic>setPics) {
    for (Pic pic: p.pics)
        if (pic.tags.contains(who)) // Note that since setPics is a Set,
            setPics.add(pic);       // duplicate adds are ignored!
}
```

(b) Recursive gather (6 points) Write method *gather*, below, using method *gatherPics* of part (a).

```
/** If p is not in set seen, then:
  * (1) Add p to seen and add any of p's pics that are tagged with who to setPics,
  * (2) Recursively do the same with all of p's pals. */
public static void gather(Person p, String who, Set<Pic> setPics, Set<Person> seen) {
    if (seen.contains(p)) return; // check if p has been seen
    seen.add(p); // add p to seen
    gatherPics(p, who, setPics); // process p's pictures using part (a)
    for (Person pal: p.pals)
```

```
        gather(pal, who, setPics, seen); // recursively call on each of p's pals
    }
```

(c) (6 points) Write the body of method *getMINE*, below, which solves the overall problem for which we want a method: Find all pictures tagged with a person's name that that user's pals, their pals, etc. have. Use the method of part (b).

Note: Person p's own photos shouldn't be included!

```
  /** Return the set of all pics that p's pals, their pals, their
   * pals, etc. have that are tagged with ps name. */
  public static Set<Pic> getMINE(Person p) {
      Set<Pic> setPics= new HashSet<Pic>(); // instantiate necessary data structures
      Set<Person> seen= new HashSet<Person>();
      seen.add(p); // add p to seen (don't gather p's pictures! See the note)
      for (Person pal: p.pals)
          gather(pal, p.name, setPics, seen); // gather all photos from p's pals using part (b)
      return setPics; // don't forget to return your results!
  }
```

3. Algorithms (14 points)

(a) Sorting (4 points)

We want to sort an array of (x, y) pairs in lexicographic order, i.e. sort by the x dimension and break ties with the y dimension. E.g.: these pairs are in order: $(1, 4), (1, 5), (2, 3)$.

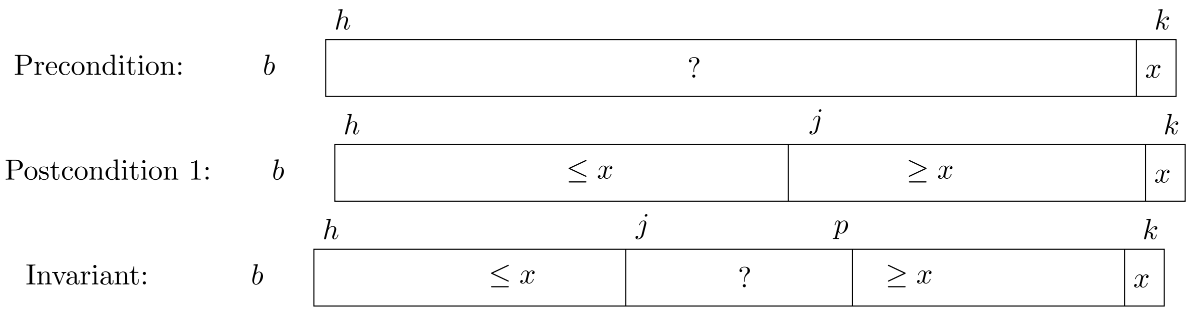
We do this by making two calls to existing function *DimSort*(*pair-array*, *dimension*), which sorts a pair-array in place by an indexed dimension:

$DimSort(pairs, 1)$; // sort by y dimension
$DimSort(pairs, 0)$; // sort by x dimension

What assumption about *DimSort* is needed to make this work? *DimSort* must be stable.
Remember that a stable sort keeps the relative positions of equal values the same. Hypothetically, assume DimSort is not stable. If the relative order is subject to change, then the second sort can change the order from the first sort.
By example, suppose that we have three points: (1, 2), (1, 3), and (2, 1). According to the specification, we should end up with with the above order. Now, after the first sort, we will get the ordering [(2, 1), (1, 2), (1, 3)] by sorting the y-coordinate. Next, we sort by the x values. If the sorting algorithm is not stable, then this sort can terminate with *pairs* equal to [(1, 3), (1, 2), (2, 1)] – this sort does accurately sort by x-values, but not correctly for this specific specification. Thus, DimSort must be stable.

(b) Partition algorithm (10 points)

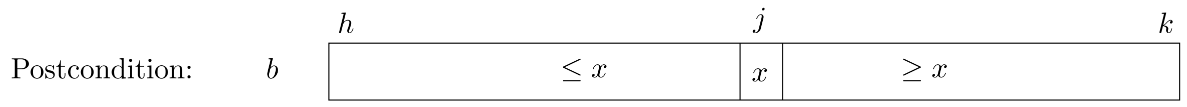
A variant of the loop of the partition algorithm of quicksort is given by the following precondition, postcondition, and loop invariant.



(i) 8 points Write a loop with initialization that swaps the values of $b[h..k]$ and stores a value in j to truthify the postcondition. Your loop must use the given invariant.

```
j= h; p= k-1;
while (j <= p) {
   if (b[j] <= b[k]) j= j+1;                    | if (b[j] <= b[k]) j= j+1;
   else if (b[p] >= b[k]) p= p-1;               | else {Swap b[j] and b[p]; p= p-1;}
   else {Swap b[j] and b[p]; j= j+1; p= p-1;}   |
}
```

(ii) 2 points What statement is needed after the loop to truthify the final postcondition shown below?
Swap b[j] and b[k]; // try converting Postcondition 1 (above) into this postcondition



4. Data Structures (28 points)

(a) Heaps (6 points)

Array segment b[0..size-1] is supposed to contain a heap —a tree with no holes that satisfies certain properties, as discussed in the lecture on priority queues and heaps. Write the body of the following function. Recursion is not necessary and makes it harder.

```
/** Return -1, 0, or 1 depending on whether b[0..size-1] is a min-heap,
  * not a heap, or a max-heap.
  * Precondition: size >1. All the elements of b[0..size-1] are different.
  * b[0..size-1] represents a complete tree, with no holes. */
public static int isHeap(int[] b, int size) {
        boolean maxHeap= b[0] > b[1];
        for (int k= 1; k < size; k= k+1) {                    // Note. Many people tried to compare b[k]
            if (maxHeap != b[(k-1)/2] > b[k]) return 0; // to its children instead of its parent.
        }                                                    // That's harder and much more work!
        return maxHeap ? 1 : -1;
    }
```

(b) Complexity (7 points)

Consider an undirected connected graph with n nodes and e edges. In the implementation, each node is an object of class *Node*, and the neighbors of a node are given as an *ArrayList<Node>*.

Look at method *visit*, below; each statement has a label on it, so we can talk about it.

```
/** Visit all nodes reachable from u along unvisited paths.
  * Precondition. v contains all visited nodes. */
public static void visit(Node u, HashSet<Node> v) {
    a: if (v.contains(u))               // 1 + 2e (we accept 2e, 1 + e, e)
    b:       return;                    // 2e - n (we accept e - n)
    c: v.add(u);                        // n
    d: for (Node node : u.neighbors)    // n
    e:       visit(node, v);            // 2e: once for each node endpoint on each edge in the graph
}
```

Suppose *visit* is called as follows, where *node* is one of the nodes of the graph.

(1) $visit(node,\ new\ HashSet{<}Node{>}());$

1. To the right of each of the 5 statements in the body of *visit*, write the total number of times it will be executed during execution of call (1) above. For (d) we want the number of times the for-each statement is executed, not how many iterations it does.

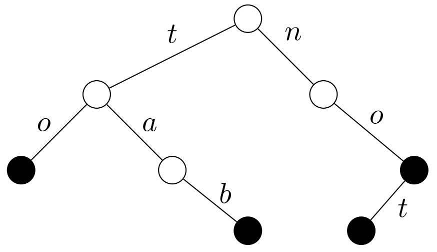
2. What is the expected time to evaluate the condition of statement (a) once? $O(1)$: v is a HashSet

3. What would be the expected time to evaluate the condition of statement (a) once if v were implemented as an *ArrayList*? $O(v.size())$: $O(n)$ is not the tightest bound!

(c) A data structure for words (15 points)

The tree to the right illustrates a neat data structure for maintaining a set of words in the English language. Assume all letters are in lower case. This tree contains the words "to", "tab", "no", and "not". A word in the set is found by reading off the characters on a path going down to a black node.

Each node has a 26-element array $b[0..25]$ for the children, each of which represents a lowercase letter and is either null or a (pointer to) a child. Each node also has a bit to say whether that node ends a word or not (black or white in the tree to the right).



(1) 3 points For the children of a node, $b[0]$ is for character $'a'$, $b[1]$ for $'b'$, etc. Let char variable c contain a lowercase letter. Below, fill in the index so that the array reference returns the value of the b-element for character c: $b[c\ -\ 'a']$ – Keep in mind that characters can be converted to integers and subtracted; thus, the array would have 'a' be in index 0, 'b' in index 1, and so on.

(2) 12 points In the table below, give the tightest expected and worst-case Big-O complexity bounds for searching for a word of length n in a set of size s of Strings when the set is implemented as (1) a tree, as above, (2) a balanced Binary Search Tree (BST) of Strings, and (3) a HashSet<String>, remembering that the hash code for a String depends on the length of the String.

Comments on the table below: (0) To start the search for string st in the data structure above, look in the root's array b, at $b[st.charAt[0] -' a']$. (1) In general, to compare two Strings of length n may take time $O(n)$. (2) The depth of a balanced BST of size s is $O(log\ s)$. (3) We are fairly lenient with the expected time of a balanced BST. (4) The hash function for a String takes time O(length-of-string).

	expected time	worst-case time
(1) Tree explained above	$O(n)$	$O(n)$
(2) balanced Binary Search Tree	$O(\log s)$	$O(n\ log\ s)$
(3) HashSet<String>	$O(n)$	$O(s * n)$

5. Graphs (12 points)

(a) Shortest-path algorithm (6 points)

State the three-part invariant of Dijkstra's shortest-path algorithm, involving a settled set, a frontier set, a far-off set, and an array d.

1. For s in the settled set, d[s] IS the shortest path length from the start node to s.
2. For f in the frontier set, d[f] is the length of the shortest path from the start node to f that consists entirely of settled nodes (except for f).

3. All edges leaving the settled set go to the frontier set; there are no edges from the settled set to the far-off set.

(b) Spanning Trees (6 points)

In introducing the notion of a spanning tree of an undirected connected graph, we stated two properties:

(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 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) 2 points Below, give 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. (You can also say ”that does not introduce a cycle.)”

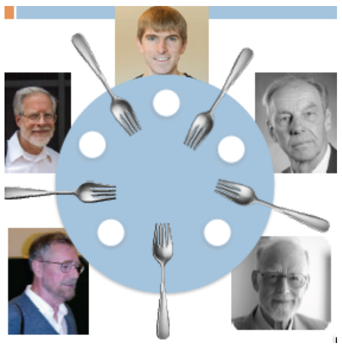
(ii) 4 points Each of the abstract algorithms (1) and (2) has a loop. Write down the number of iterations each loop takes (as a function of the number n of nodes and the number e of edges). Based on these number of iterations, state in general which of these two algorithms is preferred.

For a connected undirected graph with n nodes, each 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 a lot to throw out, and this algorithm is not preferred.

Algorithm A2 has to add $n-1$ edges, so the number of iterations is $n-1$. Thus, this algorithm is preferred.

6. Concurrency (8 points)

(a) Deadlock (4 points)



(1) Define the notion of deadlock. To help you remember, to the right is an image from a lecture slide. Deadlock occurs when two or more threads each has a control of a resource that another thread needs. In the image, suppose it requires two forks (resources) to eat. If each person picks up their left fork, there is deadlock.

(2) State the standard way of avoiding deadlock. Number the resources 1, 2, 3, ... A thread that requires two or more resources must acquire them in the order given by the numbering.

(b) Producer-consumer problem (4 points)

Recall the producer-consumer problem: Producers (e.g. a bakery chef) place products (e.g. bread loaves) at the end of a bounded queue; consumers (e.g. customer buying bread) take loaves of bread from the front of the queue. Problems arise when the queue is full or empty.

Below is the outline of the bounded-buffer class, with method *produce* specified and partially filled in. Complete the method by filling in the two lines numbered (1) and (2).

```
/** An instance maintains a bounded buffer of limited size */
public class BoundedBuffer {
    ArrayQueue aq; // bounded buffer implemented in aq

    /** Put v into the bounded buffer.*/
    public synchronized void produce(Integer v) {
        while (aq.isFull()) { // wait until there is room to add bread to the queue
            (1) wait(); // Actually it needs to be in a try-statement
        }
        aq.put(v);
        (2) notifyAll(); // notify all people waiting that you added more bread to the queue
}
```