NAME: _____ NETID: _____

CS2110 Fall 2013 Prelim 2
November 21, 2013

Write your name and Cornell netid. *There are 5 questions plus one extra-credit question on 10 numbered pages. Check now that you have all the pages. Write your answers in the boxes provided. Use the back of the pages for workspace. Ambiguous answers will be considered incorrect. The exam is closed book and closed notes. Do not begin until instructed. You have 90 minutes. Good luck!*

	1	2	3	4	5	ext	Total
Score	/20	/20	/15	/20	/25	/5	
Grader							

Below, we have reproduced the list of methods defined in class Set<E>. We use Set<E> a few times in the exam and thought you might find this helpful.

Methods of Interface Set<E>

Modifier and Type	Method and Description
boolean	**add**(**E** e) Adds the specified element to this set if it is not already present (optional operation).
boolean	**addAll**(**Collection**<**E**> c) Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	**clear**() Removes all of the elements from this set (optional operation).
boolean	**contains**(**E** o) Returns true if this set contains the specified element.
boolean	**containsAll**(**Collection**<**E**> c) Returns true if this set contains all of the elements of the specified collection.
boolean	**equals**(**E** o) Compares the specified object with this set for equality.
int	**hashCode**() Returns the hash code value for this set.
boolean	**isEmpty**() Returns true if this set contains no elements.
Iterator<**E**>	**iterator**() Returns an iterator over the elements in this set.
boolean	**remove**(**E** o) Removes the specified element from this set if it is present (optional operation).
boolean	**removeAll**(**Collection**<**E**> c) Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(**Collection**<**E**> c) Retains only the elements in this set that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this set (its cardinality).
E[]	**toArray**() Returns an array containing all of the elements in this set.
<**T**> **T** []	**toArray**(**E** [] a) Returns an array containing all of the elements in this set converted to type **T**. The type of the Set elements, **E**, must be a subtype of the desired Array element type, **T**.

1. (20 points) Suppose that you are given a deck of $0 \leq N \leq 52$ poker cards (objects of class **Card**), stored in a collection that implements the Java Interface **Set<Card>,** and are asked to implement an instance method "**public static Card pickACardAnyCard()".** The goal is to select a card *at random* within the set.

(a) [5 points] The Java Interface **Set<E>** lacks methods to extract elements from the set, but a **Set<E>** is iterable. Provide an implementation for **pickACardAnyCard(Set<Card> s)** that uses a Java "for-each" loop and returns either the first element of the set or null if the set is empty.

```
/** Return either the first element of s or null if s is empty */
public static Card pickACardAnyCard(Set<Card> s) {
    for(Card c: s)
        return c;
    return null;
```

(b) [5 points] Explain very briefly why the element returned by (a) might actually not be a randomly selected element. *Hint: Remember that **Set<E>** is an interface, not an implementation. In your answer, give an example of an implementation in which the element returned in (a) might not really be a randomly selected member from the entire set of cards. We'll give more credit if your answer explicitly talks about interfaces and implementations, and less credit if you don't discuss this point.*

There are some implementations of Set<E> for which an iterator visits the items in a fixed order. For example, a priority queue (such as a min-heap) will return values in increasing order. Thus the implementation shown above would always be certain to return the card considered to have minimum value within the deck, which isn't random at all.

(c) [15 points] Assume that a variable **rnum** of type **Random** is available and that **rnum** was initialized correctly. Further, assume that **rnum** supports an instance method **nextInt(int max).** Each call to **nextInt** returns a new random integer in the range 0..**max**. Write a one-line implementation of **pickACardAnyCard(Set<Card> s)** that will return a randomly selected member of the set. Note that unlike part (a), we are not requiring you to use a for-each loop (but you can use one)! We will give partial credit for solutions that require several lines of code, but for full credit, your solution must (1) be a single-line of code (not counting the curly braces of course), (2) be correct, and (3) if called often enough, would return every card in the deck at least once. Note: we realize that the one-line solution might not be the most efficient from a performance perspective.

```
/** Return either a random element of s, or null if s is empty */
public static Card pickACardAnyCard(Set<Card> s) {

  // If the Set isn't empty, convert it to an array and return a random element.
  return s.size() == 0? null:  s.toArray()[ rnum.nextInt(s.size() - 1) ];









}
```

2. (20 points) True or false?

a	**T**	F	Even when working in a language other than Java, like Matlab or Python, no comparison-based sorting algorithm can achieve worst-case complexity better than O(n log n).
b	**T**	F	If A is a superclass of B, variable myObj is declared to be of type A, and myObj = new B() is executed, the dynamic type of myObj is B but the static type is A.
c	**T**	F	A heap data structure would be a fantastic choice for implementing a priority queue. We could use a min-heap if we want to extract elements in order smallest to largest, and a max-heap if we prefer to extract them from largest to smallest.
d	**T**	F	Items pop from a stack in "last in, first out" or LIFO, order
e	**T**	F	A Java class can extend at most one class but can implement many interfaces.
f	**T**	F	If the same element is inserted multiple times into a Java set, the set will contain only a single instance of that element.
g	T	**F**	If the same element is inserted multiple times into a Java list, the list will contain only a single instance of that element.
h	**T**	F	If a GUI includes a JFrame, you can add a large number of components to that JFrame.
i	**T**	F	Java won't garbage collect an object if there is any way to reference that object from some active thread in your program.
j	**T**	F	A cache typically contains some limited set of previously computed results and is used to avoid recomputing those results again and again.
k	T	F	If a method always takes exactly 10 minutes to compute something, we would say that it has complexity O(10 mins).
l	T	**F**	The worst-case complexity of looking up an item in a BST containing N items is O(log(N))
m	**T**	F	If method m is declared: **`static void m(ArrayList<Object> arg),`** you will get a compile-time error if you declare **`X`** to have type **`ArrayList<String>()`** and try to invoke **`m(X);`**
n	**T**	F	If you use a very badly chosen hash function, then looking up an item in a HashMap might have complexity much worse than O(1)
o	T	**F**	A hashcode() method must return a prime integer larger than 7.
p	T	**F**	If a method does something that requires N^2 operations and then does something else that requires N log N operations but does it N/2 times, the complexity of the method is $O(N^2)$
q	T	**F**	Nested for loops always result in complexity $O(N^2)$
r	T	**F**	If classes Dog and Cat extend class Animal and cleopatra is an instance of Cat, (Dog)((Animal) cleopatra).bark() will upcast cleopatra to Animal, then downcast to Dog, and then run its bark() method. *Assume that only class Dog defines the bark method.*
s	T	**F**	In a recursive procedure the base case does all the work.
t	T	**F**	One difference between Java and other languages is that in Java, type checking is used to create documentation, also known as Java Doc.

3. [15 points] Suppose that you are given two objects **b** and **c**, both of type **Set<E>,** and assume that that developer who defined type **E** overroad the hashcode, compareTo and equals methods, providing her own definitions for them**.** See the first page for methods that you can use on Sets.

In part (b) of this question, we will ask you to write a function that checks whether **b** and **c** contain exactly the same values. *Hint: be careful to check by value. We want to know if the lists have the same sets of values, not whether they contain the same objects.*

(a) [5 points]. In a sentence or two, explain why we needed to stress that we are testing for identical values. Would **areSame** need to be different if we *did* want to test that the lists of objects were the same?

In Java, distinct objects can still have the same value: objects are considered to be equal if the equals method returns true. For example suppose we have x != y, but x.equals(y) is true. If we put x and y into two sets, the sets would be the same if we use value equality, and yet they contain different objects.

(b) [10 points]. Provide the code for the method **areSame**.

```
/** Return true if b and c contain exactly the same set of values. */
public static boolean areSame(Set<E> b, Set<E> c) {

    return b.containsAll(c) && c.containsAll(b);

                    . . . or

    return b.size() == c.size() && b.containsAll(c);









}
```

4. (20 points). For each of the following short-answer questions about sorting and priority-queue algorithms, circle the correct answer.

(4 points) When using the **MergeSort** algorithm

a. The algorithm needs a temporary array of size N so that it will have a place to merge the two sorted subsets of the input data.
b. The length of the input vector must be a power of 2, so that we can repeatedly subdivide it in halves.
c. The worst-case complexity will be $O(N^2 \log (2/3\ N))$

(4 points) The **QuickSort** algorithm…

a. Has worst-case complexity $O(N^2)$, but if the pivot value is selected properly, would have complexity O(N log N)
b. Needs a temporary array of size N, like MergeSort, but only uses part of it.
c. Is probably the best algorithm to use if the input arrays often have large numbers of identical values in them (for example, if you expect to often see arrays that are mostly zeros, or that have other big blocks of repeated numbers in them).

(4 points) The **HeapSort** algorithm

a. Works by first putting all N items into a min-heap, then extracting them again
b. Sorts data in place by calling the Heapify method on the original data in the vector where it was originally located.
c. Turns the data into a fancy kind of binary tree but can be used only for TreeNode objects since the tree requires child pointers for each node.

(4 points) When using a **min-heap** structure to implement a priority queue

a. The worst case cost of inserting one item or removing one item is O(N), and it arises if the item is the new smallest item for insert or if you removed the previous smallest item.
b. We think of the data as being organized into a binary tree but in reality the tree takes the form of a one-dimensional array.
c. The Heap invariants ensure that the values in the subtree to the left of any node will be smaller than the value of the node itself, which in turn must be a value smaller than any values stored in the subtree to the right of that node.

(4 points) Suppose a min-heap contains N items and is represented as a 1-dimensional array, as discussed in class. Now consider the item stored in the location with index k. Which of the following prints a list of the indices of the nodes on the path from k to the root, not including k itself?

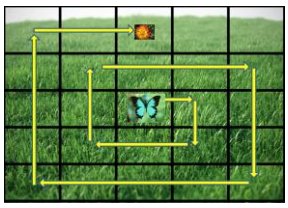
a. int i = k; while(i > 0) { i = (i-1)/2; console.println("The next node is at index " + i); }
b. int i = k; while(i < N) { i = i*2+1; console.println("The next nodes are at index " + I + " and " + (i+1)); }
c. Neither a nor b will work: to print the nodes on that path, one *must* use recursion.

5. (25 points) In our Butterfly project we focused on building a map of the flyable part of the park and finding every flower within it. Suppose that our goal is different and that we instead want to have the butterfly visit a flower that is nearest to its starting location (if two flowers are at the same distance, either will be fine). Moreover, assume that we want to solve this problem *without using flower aroma information* and that we *know nothing about the park when the program starts running.*

a. [5 points] In about a number of graph algorithms: DFS, BFS, Dijkstra's, Prim's, Kruskal's algorithm, etc. For each of these, give a very short (one line) definition of what it does (for DFS and BFS you should tell us what the initials stand for and *also* tell us what the method does).

DFS	**Depth first search. The search recursively visits the children of each vertex before visiting (searching) the vertex itself.**
BFS	**Breadth first search. The search visits a vertex, then the child vertices that were reachable in one hop from it, then the vertices reachable in one hop from those, etc.**
Dijkstra's	**This isn't a search algorithm. Dijkstra's algorithm computes the length of the shortest paths from a given vertex to each of the other vertices in a graph for which the edges have positive weights. (It can also keep track of what those shortest paths actually are.)**
Prim's	**A greedy algorithm for finding a spanning tree in a weighted graph. Prim's is an additive algorithm: it forms a component by picking any vertex, then repeatedly grows the spanning tree by adding the minimum-weight edge that runs from it to some vertex not already in the spanning tree.**
Kruskal's	**A greedy algorithm for finding a spanning tree in a weighted graph. Kruskal's is also an additive algorithm, but it works differently from Prim's. Looking at all edges in the entire graph in order of weight, smallest first, it takes each edge that doesn't introduce a cycle, ignoring edges that would create a cycle when added to spanning tree.**

b. [10 points] One way to solve this problem is to have the butterfly fly in a spiral around the point at which it starts, as in the example to the right. This is a form of BFS: BFS permits a wide range of possible search orders, and it is easy to create a version of BFS that will use a spiral order. Assume we've done this. Here's the specification of S-BFS:



```
/** S-BFS does a BFS search starting at location [x][y], visiting  cells in a spiral pattern.
  *  For d>0, S-BFS visits all cells at distances d' < d before visiting any cell at distance d.
 */
```

*Let's say that the shortest path to the flower is of length **d** from where the butterfly starts and let's also assume the map contains no unflyable tiles***.** For example, in our picture, there is one flower at distance 2. Obviously, with S-BFS the butterfly will fly quite a bit further (22 hops) before finding it.

We would like you to provide an inductive proof that S-BFS is guaranteed to find a closest flower. Note: If you are unable to do an inductive proof, for a maximum of 5 points, simply tell us in English why S-BFS will find that flower. To help, we've structured what you must do into three subparts:

[3 points] Write down the base case of the induction:

d=0. (The butterfly gets lucky and was born in the cell containing the flower.)

[3 points] Write down the inductive hypothesis:

I.H: Assume that for values of d from 0..k the algorithm will find the closest flower.

[4 points] Write down the inductive step and prove it. Then tell us why this implies that BFS will find a closest flower:

From the I.H. we know that if the flower was at distance ≤k, S-BFS would have found it. All cells at distance k+1 are one hop from cells at distance k. Therefore, S-BFS will search all the cells at distance k+1 next, in a clockwise order as shown in the figure. If the flower is at distance d=k+1, it will find the flower. Otherwise, we can conclude that there is no flower within distance k+1.

From this, and from the assumption that there exists a closest flower at distance d, we can conclude that S-BFS will find that flower after searching at distances 0…d.

c. [10 points] Derive the O() worst-case complexity of finding a closest flower using S-BFS. Again, you may assume that there are no unflyable tiles within distance d of the butterfly origin. In your big O() analysis, count "fly-to" operations. Note: If you are unable to derive a O() worst-case complexity, we will award a maximum of 2 points if you simply tell us, in English, precisely what the worst-case scenario will be for a S-BFS search of this kind (drawing a picture can help us understand your English description).

[2 points] Worst case scenario:

The worst case arises if the flower is the very last cell searched at distance d. In the clockwise search shown, this would be the flower immediately above the cell at distance d to the right of the origin location.

[8 points] O() complexity of finding a flower at distance d from the origin

[Option a] We observe that to search the cells at distance d, the butterfly visits all cells in a box. A box of side-length K will have K^2 cells in it. So what is K? For distance d, it is trivial to see that a box would have sides of length 2d+1. Squaring this we have $(2d+1)^2$ which is $O(d^2)$

[Option b] We need to compute the path length flown by the butterfly. Some examples will help:

Distance 0: 0
Distance 1: (Distance 0) + 8 (1 hop to the right, then around 4 sides of length 2)
Distance 2: (Distance 1) + 16 (1 more hop to the right, then around 4 sides of length 4)

Clearly the general formula is this: to search the cells at distance k, we first search from 0…k-1, then fly outwards one cell to reach the square consisting of cells at distance k, and then we search a square of size 4*(2*k). We can write this as Cost(k) = $\sum_{i=0}^{k} 8i$. As we saw in class, this is 8n*(n-1)/2. Ignoring the constant 8, since constants don't matter in a O() computation, and the linear term, this is $O(k^2)$.

[5 points extra credit] Suppose that an algorithm does 2^0 operations for an input of length 0, $2^1 + 2^0$ operations for an input of length 1, $2^2 + 2^1 + 2^0$ for an input of length 2. Thus $count(n) = \sum_{k=0}^{n} 2^k$. Derive a simple closed-form formula for the O() complexity of the algorithm for an input of length n. If any step in your derivation might not be completely obvious, give a very brief English-language explanation of that step. *Hint: It may help to think about the binary representation of a positive integer. It can also be useful to write down the formula opcount(n) for n= 0, 1, 2, 3, … and see whether the pattern helps you figure out the closed-form formula. We'll give 2 points if your closed form formula is correct, and 3 more points if your explanation is rigorous enough to convince us. But we're not insisting on any particular <u>style</u> of proof, just a <u>convincing</u> proof.*

If we think of a positive integer represented in binary form, we're being asked to compute this sum:

Compute….		Binary Representation									
		2^{n+1}	2^n	2^{n-1}	…	…	2^4	2^3	2^2	2^1	2^0
	2^0	0	0	0	0	0	0	0	0	0	1
	$+2^1$	0	0	0	0	0	0	0	0	1	0
	$+2^2$	0	0	0	0	0	0	0	1	0	0
	…										
Total	=	0	1	1	1	1	1	1	1	1	1

Here I've simply made a table showing the binary representation of each of the numbers we're asked to sum. Each power of two has just one bit set, so the sum, shown in the bottom row, has all bits set from 0 to bit n. This number is clearly 2^{n+1}-1 by the properties of binary arithmetic.

Thus the closed form is 2^{n+1} - 1