Name: ________________________ Student ID: __________________________

Unannounced Quiz #3

1. We have discussed how to implement a complete binary tree (e.g., a heap) using an array as the underlying storage. It is also possible, however, to implement a heap using linked nodes:

```
class Node<T> {
  Node<T> _parent, _leftChild, _rightChild;
  T _data;
}
```

A node-based heap would maintain a pointer to a single node, **`_root`**. Every time the user **`add`**s a new object **`o`**, the implementor of this **`Node`**-based ADT would have to enforce the *completeness* constraint by making sure it adds the new node as the left-most child of the lowest level of the heap. Implement your **`add (T o)`** method below; it may take *O*(log *n*) time and O(log *n*) space (in addition to the memory used to store the nodes themselves). You may *not* use an array as the underlying heap storage.

2. (From Cormen, Leiserson, and Rivest 1999): Suppose that we have numbers between 1 and 1000 in a binary search tree and want to search for the number 363. Which of the following sequences could *not* be sequences of nodes examined?
   (a) 2, 252, 401, 398, 330, 344, 397, 363.
   (b) 924, 220, 911, 244, 898, 258, 362, 363.
   (c) 925, 202, 911, 240, 912, 245, 363.
   (d) 2, 399, 387, 219, 266, 382, 381, 378, 363.
   (e) 935, 278, 347, 621, 299, 392, 358, 363.

3. (From Cormen, Leiserson, and Rivest 1999): Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A$, $b \in B$, $c \in C$ must satisfy $a \leq b \leq c$. Give a smallest possible counter-example to the professor's claim.

4. For my implementation of P3, I wrote a method called **`wrapIndex`** which “wrapped” the specified index around the capacity of the ring buffer:

```
int wrapIndex (int index) {
  return index % capacity();
}
```

However, this method only works if **`index >= 0`**. In Java, the modulus operator `%` is defined so that **`a % b`** can be any number in the set { -(*b*-1), -(*b*-2), ..., -2, -1, 0, 1, 2, ..., *b*-2, *b*-1 }. For example, if *b* is 4, then *a* % *b* can be any number in { -3, -2, -1, 0, 1, 2, 3 }. In particular, 7 % 4 is 3, and -7 mod 4 is -3. For ring buffers, this isn’t very useful, as a negative index is never valid.

To fix this problem, write an *improved* implementation of wrapIndex whose range (set of possible return values) is always { 0, 1, 2, ..., **`capacity()`**-1 }. For instance, **`-7 % 4`** should be 1, and **`7 % 4`** should still be 3. *Your solution may not contain any if/else statements.*

5. For the following imbalanced BST, say what kind of configuration (LL, LR, RL, or RR) it is. Label the height and balance of each node. Finally, re-balance the tree by performing the necessary AVL rotation(s).