EECS-214 Practice Quiz 1

Answer the following questions as best you can. When writing code, you may not use any built-in sequence types other than arrays. In other words: if you need to use linked-lists, then you should build the lists yourself rather than assuming the presence of a pre-made linked list class, such as the C# LinkedList<T> class.

Note: the real quiz will not be this long.

Question 1

Consider the following pseudocode:

```
PrintSorted( list of numbers) {
    make a new, empty, binary search tree
    for each number in the list
        add it to the binary search tree
    node = minimum node of the BST
    while node != null {
        print node.key
        node = successor(node)
    }
}
```

A. What is the worst case time complexity (i.e. O(1), O(n), etc.) of this algorithm if we use a normal (i.e. not self-balancing) binary search tree?

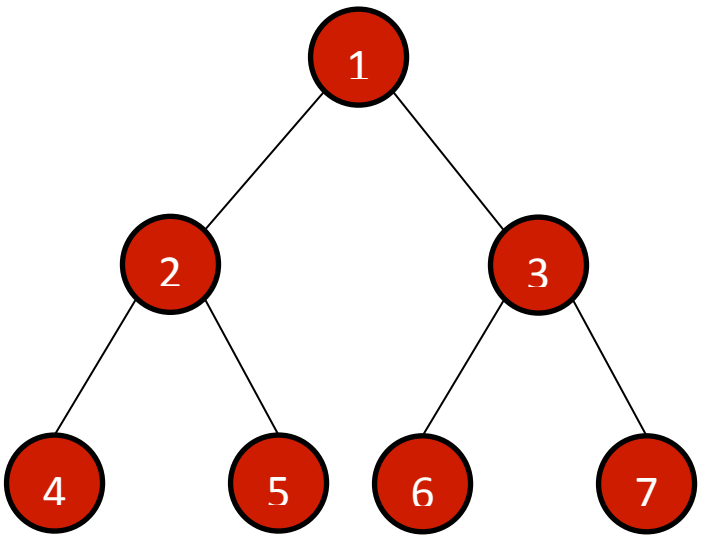
Answer: $O(n^2)$ (this is all you'd have to answer on the test)
Explanation: Both the add and successor operations are $O(h)$, but for unbalanced binary trees, h, the height of the tree is $O(n)$ in the worst case, so the individual add and successor operations end up being $O(n)$ also. Since they get executed n times, we have $O(n^2)$ for the complete algorithm.

B. What if we use a red/black tree instead of a normal binary tree?

Answer: O(*n* log *n*)
Explanation: same as above, except that the individual operations are O(log n) in the worst case, so the total time is O(n log n).

Question 2

Give the order in which the nodes of the tree below would be printed by an in-order traversal



Answer: 4 2 5 1 6 3 7

Question 3

Give an algorithm for walking the tree above that, if run on the tree above, would walk the nodes in the order 1, 2, 3, 4, 5, etc. You may assume that stack and queue data types have already been defined and that you can use them for your answer without having to write code for them.

Answer: This is just a breadth-first traversal, so we use a queue:

```
Make a new, empty queue
Add root to queue
While queue not empty
    node = queue.Dequeue()
    print node
    for each child c of node
        add c to queue
```

Question 4

Consider the following linked-list queue implementation:

```
class Cell {
   Cell next;
   int value;
}
```

```
class Queue {
   Cell head;
   Cell Tail() {
       Cell c = head;
       while (c.next != null) c = c.next;
      return c;
   }

   void Enqueue(value) {
      Cell t = new Cell();
       t.value = value;
       t.next = null;


       if (head==null)
           head = t;
      else
           Tail().next = t;
   }
}
```

This implementation has the unfortunate consequence that building a queue with n items takes $O(n^2)$ time (assuming we don't dequeue anything inbetween). Explain how to fix this so that it takes only O(n) time.

Answer: The problem is that we're always storing at the tail (end) of the queue, but we only have a pointer to the head (beginning) of the queue. So each time we enqueue, we have to find the tail using a linear search, which is O(n) time. Calling that n times is $O(n^2)$. The solution is to add a second field to the queue that holds the tail of the queue. Then the Enqueue operation looks like:

```
   void Enqueue(value) {
      Cell t = new Cell();
       t.value = value;
       t.next = null;


       if (head==null)
           head = tail = t;  // empty queue so new node is both head and tail
      else {
           tail.next = t;  // just update the tail
           tail = t;
      }
```

}

Question 5

Write a class definition for a tree node for a tree. Assume the tree can have an arbitrary branching factor (i.e. no limit on the number of children). You need only give the code for the fields; you do not need to define any methods for the class.

Answer: There are several possible reasonable answers for this. One possibility it to use the left-child/right sibling representation:

```
class TreeNode {
   TreeNode parent;
   TreeNode leftChild;
   TreeNode rightSibling;
}
```

Another is to use an array to hold the children, if for example, children aren't expected to be added or removed:

```
class TreeNode {
   TreeNode parent;
   TreeNode[] children;
}
```

Question 6

Given the following definition for a doubly-linked list:

```
class DLLCell {
   DLLCell previous;
   DLLCell next;

}

class DLList {
   DLLCell head;    // first cell in the list
}
```

Write a procedure, AddAfter(DLList list, DLLCell c) that adds a new cell to the list *list*. It should add it immediately after the cell *c*, which you may assume is already in the list *list*. If *c* is null, you should add the cell at the beginning of the list.

Answer: I'm just writing this as pseudocode, which you should feel free to do also.
Note: it occurs to me as I'm writing the solution to this, that in my attempt to simplify the question

and remove distracting details, I've made this a somewhat odd question. I didn't add any fields to the Cell datatype to hold keys or other data, so the Cells are kind of useless. It simplifies the question, but it also makes adding a cell to the list kind of meaningless. My apologies if that made the question more confusing rather than simpler.

```
AddAfter(list, c) {
    newCell = new Cell()
    if (c == null) {     // Add to beginning
       newCell.previous = null
       newCell.next = list.head
       if (list.head != null)  // If list is non-empty
          list.head.previous = newCell
       list.head = newCell
   }  else {  // Splice inbetween existing cell c and whatever comes after it
       newCell.next = c.next            // c's old successor is newCell's successor
       if (c.next != null)              // Of course it might be null if c is at the end
         c.next.previous = newCell   // If not, update the successor's previous field
       newCell.previous = c
       c.next = newCell
    }
}
```

Question 7

Here's a random algorithm that operates on an array. Yes, I know it doesn't do anything useful, but tell me what its time complexity is anyway (i.e. is it linear, quadratic, log n, or what?).

```
BlaBlaBla(array, start, end)
   if (start!=end)
      for each i between start and end
         sum += array[i]
      BlaBlaBla(array, start, (start+end)/2)
      BlaBlaBla(array, (start+end)/2, end)
```

Answer: The recursion runs $O(\log n)$ level deep. At each level, the different calls to BlaBlaBla run their for each loops for different numbers of iterations, but the total number of iterations is always n, so the total number of iterations per level of recursion is n, so the total time is $O(n \log n)$.