CS21, Swarthmore College, Spring 2008, Practice Quiz 7

1. A `Queue` is a collection of items supporting two basic operations: `enqueue` and `dequeue`. The `Queue` data structure can be thought of as representing a line of people waiting to buy a concert ticket: the first person in the queue will be the first person to get a chance to buy the ticket; the last person in the queue will not get a chance to buy a ticket until everyone in front of them has bought their ticket. When a new person arrives to buy tickets, we use the `enqueue` method to add them to the end of the line; when a person is helped at the ticket counter, we use the `dequeue` method to remove them from the front of the queue.

   One way to implement a queue is to use a Python list to store the data, storing the most recently added item at the end of the list. The `enqueue` operation is accomplished by using list's `append` method and the `dequeue` operation is accomplished by using list's `pop` method. Below is a simple `Queue` implementation using Python lists:

```
class Queue(object):
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if self.queue == []:
            raise IndexError("dequeue from empty queue")
        else:
            return self.queue.pop(0)
```

   - What is the output of `main`, shown below?

```
def main():
    s = Queue()

    for i in range(5):
        s.enqueue(i)

    for i in range(5):
        print s.dequeue()
```

   - The implementation of the `Queue` shown above is not algorithmically efficient. What is the run time in terms of n, the number of items in the `Queue`, for both the `enqueue` and `dequeue` methods?

   - We could rewrite the `Queue` class so that when we enqueued an item, we added it to the front of the list, and when we dequeued an item, we removed it from the end of the list. What is the run time of this new implementation? Is this more efficient, less efficient, or equally efficient to the original implementation?

2. Below is part of the interface for the `LinkedList` class:

```
class LinkedList(object)
 |  Methods defined here:
 |
 |  __init__(self)
 |      Create an empty linked list.
 |
 |  append(self, value)
 |      Add an item to the end of a linked list.
 |
 |  insert(self, index, value)
 |      Insert a value into the list at the position specified by index.
 |
 |  pop(self, index)
 |      Remove and return the value at the position specified by index.
```

   - Write a new implementation of the `Queue` class that uses a `LinkedList` to store the items, instead of a Python list. Your solution should be more efficient than either of the two proposed solutions that used Python lists.
   - What is the run time of the `enqueue` and `dequeue` methods you wrote?

3. Explain why the runtime of finding a key in a balanced binary search tree is proportional to $\log_2 n$, where n is the number of items in the tree.

4. The binary search tree we developed in class sorts items in the tree by key, not by value. This means that if we wanted to find a node in the tree that stored a particular value, we could not take advantage of the ordered property of the binary search tree.

   - Using the implementation we developed in class, add a new method called `findValue`:

     ```
     |  findValue(self, value)
     |      Search for the BinaryNode containing this value beginning from the
     |      root of the tree.  Returns the BinaryNode if found; None otherwise.
     |      If there are multiple BinaryNodes with this value, return the node
     |      with the smallest key.
     ```

   - What is the run time of the `findValue` method you wrote?