CSE 373 Midterm 1   1/30/06   Sample Solution

Question 1. (10 points) (a) Define what it means to say that a function $f(n)$ is $O(g(n))$. (i.e., give the mathematical definition as described in lecture and in the textbook.)

Function $f(n)$ is $O(g(n)$ if there is a constant c and an integer n_0 such that $f(n) \leq c\, g(n)$ for all $n \geq n_0$

(b) Use the definition from part (a) to prove that $3n + 2n^2 + 12$ is $O(n^2)$.

Pick $c = 3$ (or any number greater than 2). Then for all $n \geq 6$ (or any larger value of n), $3n + 2n^2 + 12 \leq 3n^2$.

Question 2. (8 points) Adding a new item to an array-based list normally takes constant time ($O(1)$), except when the existing array is filled to capacity. In that case we need to allocate a new larger array, copy the existing items from the old array to the new one, then add the new item. The time needed to add an item to the list in this case requires time proportional to the number of items in the list ($O(n)$).

In class, and in the book, we argued that the *total* time needed to add n items to an array-based list was $O(n)$, if, whenever we need to expand the array, we allocated a new one that is **twice** the length of the old one. In other words, with this strategy the *amortized* time needed to add each item was $O(1)$.

Another strategy would be to allocate a new array that is only **one** element larger than the old one whenever the existing array is filled to capacity. For this problem, show that if we only increase the size of the array by 1 each time we need to expand it, then the total time needed to add n items to the list is $O(n^2)$.

Once the initial capacity of the array, say k, has been used, each addition to the list requires allocating a new array and copying all of the existing entries. So the total work needed is proportional to k (for the first k entries) then k+1, k+2, …, n, for the remaining n-k entriees. The sum $k + (k+1) + (k+2) + \ldots + n$ is proportional to n^2, so the total work needed to add n items to the list using the "increase by 1" strategy is $O(n^2)$.

Question 3. (6 points) Recall from lecture and from the book that the operations on a *queue* data structure can include:

- *enqueue*(*item*) – add *item* to the rear of the queue
- *dequeue*( ) – remove and return the front item in the queue.
- *front*( ) – return the front item from the queue but do not remove it
- *isEmpty*( ) – return true if the queue is empty
- *size*( ) – return the number of items currently in the queue

Write down the output that is printed by `System.out.println` when the following code is executed. You should assume that the queue is initially empty. Also assume that the queue `Q` stores and returns `Strings`, so that no casts or other type conversions are needed for this code to compile and run properly. Hint: as you work on the problem, draw a picture to keep track of the contents of the queue.

```
Q.enqueue("Sleepy");
Q.enqueue("Grumpy");
Q.enqueue("Sneezy");
System.out.println(Q.front());
System.out.println(Q.size());
Q.enqueue("Happy");
String s1 = Q.dequeue();
Q.enqueue("Dopey");
String s2 = Q.dequeue();
System.out.println(s2);
System.out.println(Q.size());
Q.enqueue("Bashfull");
String s3 = Q.dequeue();
System.out.println(s3);
System.out.println(Q.size());
```

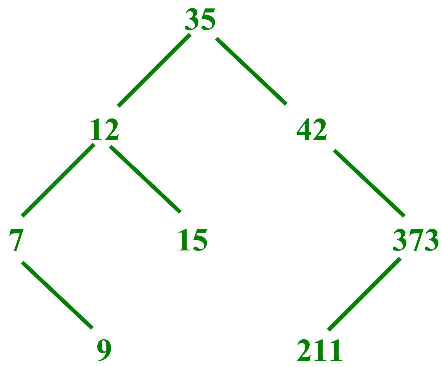
Output produced by `System.out.println`:

```
Sleepy
3
Grumpy
3
Sneezy
3
```

(For movie trivia fans – but not for any extra credit – What is the name of the 7th dwarf?)
Doc

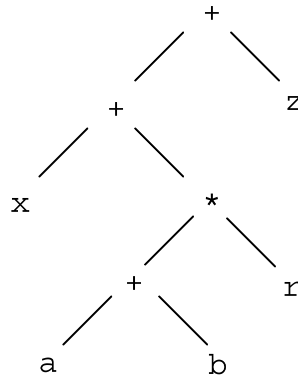
Question 4. (6 points) Draw a diagram of the *binary search tree* (BST) that results when the following numbers are added to the tree *in the given order* without any additional balancing or rearranging of the nodes in the tree.

35 12 7 9 42 373 211 15



Question 5. (6 points) An application of binary trees (*not* binary search trees) is to provide a data structure to represent arithmetic expressions in programs like compilers or symbolic mathematics packages like Mathematica or Matlab.

For example, here is a binary tree representing the expression $x + (a + b) * r + z$. In this tree, each operator and operand is represented by a node, and each operator node has children that represent its operands.



(a) Write down the order in which the nodes of the above tree are encountered by an *inorder* traversal of the tree. Include all of the nodes, both operators and operands.

x + a + b * r + z

(b) Write down the order in which the nodes of the above tree are encountered by a *postorder* traversal of the tree. Include all of the nodes, both operators and operands.

x a b + r * + z +

Question 6. (12 points) One of the methods we did not implement in our linked-list classes is a method to add an item to the list at a particular location given a reference to the link at that location. For this problem, your are to implement method `add(item,p)` to add `item` to a double-linked list just *before* the node referenced by `p`.

Details: Assume that the links in the list are represented as follows:

```
class DLink {
   public Object item;   // item referred to by this link
   public DLink next;    // next link in the list; null if none
   public DLink prev;    // previous link; null if none

   /** construct new link given item, next, and previous */
   public DLink (Object item, DLink next, DLink prev) { ... }
}
```

You should assume that these three instance variables *only* are available and should be used as needed:

```
private DLink head;      // first link or null if empty list
private DLink tail;      // last link or null if empty list
private int size;        // number of links in the list
```

There are no extra header or trailer nodes in the list. Complete the definition of method `add`, below. You may assume that `p` is not null and refers to a link in the list.

```
/** Add item to the list immediately before link p ... */
public void add(Object item, DLink p) {

   DLink q = new DLink(item, p, p.prev);

   if (p == head) {
      head = q;           // new link at head of list
   } else {
      p.prev.next = q;    // new link in middle of list
   }

   p.prev = q;
   size++;

}
```

The following two questions refer to sorted sets implemented with binary search trees, as in homework 3. Recall that the operations available on a set include the following

- *add*(*item*) – add *item* to the set if not already present.
- *contains*(*item*) – return true if the set contains *item*; false if not
- *size*( ) – return the number of items in the set
- *remove*(*item*) – remove item from the set if present

Question 7. (10 points) Complete the following JUnit test to verify that the operations in the test work properly. Recall that JUnit includes methods names `assertEquals`, `assertTrue`, `assertFalse`, `assertNull`, `assertNotNull`, and `fail`, among others. Your test only needs to verify properties of the set that can be observed with the above functions at the blank places below where there is room to add code. You don't need to add tests between the lines of code in the rest of the test.

```
// check set operations after adding and deleting a few items
public void testAddRemoveEtc( ) {
   private BasicSet s = new BasicOrderedTreeSet();
   s.add("Mickey");
   s.add("Michael");
   s.add("Donald");
   // add any appropriate tests below

   assertTrue(s.contains("Mickey"));
   assertTrue(s.contains("Michael"));
   assertTrue(s.contains("Donald"));
   assertFalse(s.contains("Steven"));
   assertEquals(3, s.size());

   s.remove("Michael");
   s.add("Steven");
   // add any appropriate tests below

   assertTrue(s.contains("Mickey"));
   assertTrue(s.contains("Steven"));
   assertTrue(s.contains("Donald"));
   assertFalse(s.contains("Michael"));
   assertEquals(3, s.size());

}
```

Question 8. (12 points) Several of the Java collection classes provide methods to return a copy of the collection as a list. For this question, implement a method `toList()` for `BasicOrderedTreeSet` that does this. The items in the list should be stored in increasing order.

Details: Assume that the set is represented using a binary search tree whose nodes are defined as follows:

```
class BSTNode {
   public Comparable item;     // item referred to by this node
   public BSTNode left;        // left subtree; null if none
   public BSTNode right;       // right subtree; null if none
   ...
}
```

The `BasicOrderedTreeSet` contains only the following instance variable:

```
private BSTNode root;    // root of the tree or null if the
                         // set (tree) is empty
```

Complete the definition of the method `toList` below. You may define additional methods if you find them helpful. **Restriction**: You may ***not*** use any additional methods belonging to the `BasicOrderedTreeSet`. In particular, you may ***not*** use an iterator to solve the problem.

Hints: An inorder tree traversal might be useful. Use an `ArrayList` or `LinkedList` to accumulate the result, and recall that these lists include an `add(…)` method to add new items to the end of a list.

```
/** Return the contents of this set as an ordered list */
public List toList() {

   ArrayList values = new ArrayList();
   addSubtree(root, values);
   return values;

}

// add contents of subtree with root r to values in order
private void addSubtree(BSTNode r, List values) {
   if (r == null) {
      return;
   }

   addSubtree(r.left, values);
   values.add(r.item);
   addSubtree(r.right, values);
}
```