Reference information about many standard Java classes appears at the end of the test. You might want to tear off those pages to make them easier to refer to while solving the programming problems.

Question 1. (8 points) (a) Draw the binary search tree that is created if the following numbers are inserted in the tree in the given order.

12 15 3 35 21 42 14

(b) Draw a *balanced* binary search tree containing the same numbers given in part (a).

Question 2. (3 points) Show that $3n^4 + 100\ n^2 + 42n$ is $O(n^4)$.

Question 3. (2 points) Methods `hashCode()` and `equals()` are defined in class `Object` and are inherited by all Java classes. If a Java class overrides `equals()` it should normally also override `hashCode()` to be sure that `equals()` and `hashCode()` work together properly. Given two objects `o1` and `o2`, what should be true about the `hashCode()` function if `o1.equals(o2)`?

Question 4. (8 points) During the quarter we've looked at several different ways of implementing a collection of objects, in particular:

- Lists based on arrays
- Linked lists
- Trees, particularly binary search trees
- Hash tables

For each of the following operations, circle the expected time required for the operation. Your answers should assume that the data structure is performing well, i.e., binary search trees are reasonably well balanced and hash tables have a low load factor (i.e., your answer should be the *expected* time, not the *worst case* time if those two are different).

(a) ***contains***: return true or false depending on whether a particular object is a member of a collection.

Sorted list implemented with an array:

$O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

Sorted list implemented a with linked list:

$O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

Hash table: $O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

Binary search tree: $O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

(b) ***insert***: add a new item to the collection in the appropriate place

Unsorted list implemented with an array:

$O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

Sorted list implemented with an array:

$O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

Hash table: $O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

Binary search tree: $O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

The following class implements an *n* by *n* square 2-D array with several operations. It is used in the next two questions.

```
public class Grid {

   private double[][] grid;

   public Grid(int n) {
      grid = new double[n][n];   // array elements are all 0 here
   }                             // (this takes constant, O(1) time)

   public void set(int row, int col, double value) {
      grid[row][col] = value;
   }

   public double get(int row, int col) {
      return grid[row][col];
   }

   public void smash(int row, int col) {
      double sum = 0.0;
      for (int r = row-1; r <= row+1; r++) {
         for (int c = col-1; c <= col+1; c++) {
            sum = sum + grid[r][c];
         }
      }
      grid[row][col] = sum/9.0;
   }

   public void reverse(int row) {
      int L = 0;
      int R = grid.length-1;
      while (L < R) {
         double temp = grid[row][L];
         grid[row][L] = grid[row][R];
         grid[row][R] = temp;
         L++;
         R--;
      }
   }

   public void flip() {
      for (int r = 0; r < grid.length; r++) {
         reverse(r);
      }
   }

}
```

Question 5. (8 points) For each of the following code fragments, circle the smallest complexity class that gives the running time of the code as a function of the grid size *n*. You should assume that the parameter `n` is large enough so that no `IndexOutOfBounds` exceptions will occur.

(a)
```
public void parta(int n) {
   Grid g = new Grid(n);
   g.flip();
}
```

$O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

(b)
```
public void partb(int n) {
   Grid g = new Grid(n);
   for (int k = -10000; k <= 10000; k++) {
      g.smash(1, 1);
   }
}
```

$O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

(c)
```
public void partc(int n) {
   Grid g = new Grid(n);
   for (int r = 1; r < n; r = 2*r) {
      g.reverse(r);
   }
}
```

$O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

(d)
```
public void partd(int n) {
   Grid g = new Grid(n);
   for (int x = 1; x < n-1; x++) {
      g.smash(x,x);
   }
}
```

$O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

Question 7. (4 points) (a) What is the *expected* time needed by Quicksort to sort a list with n items? (circle)

$O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

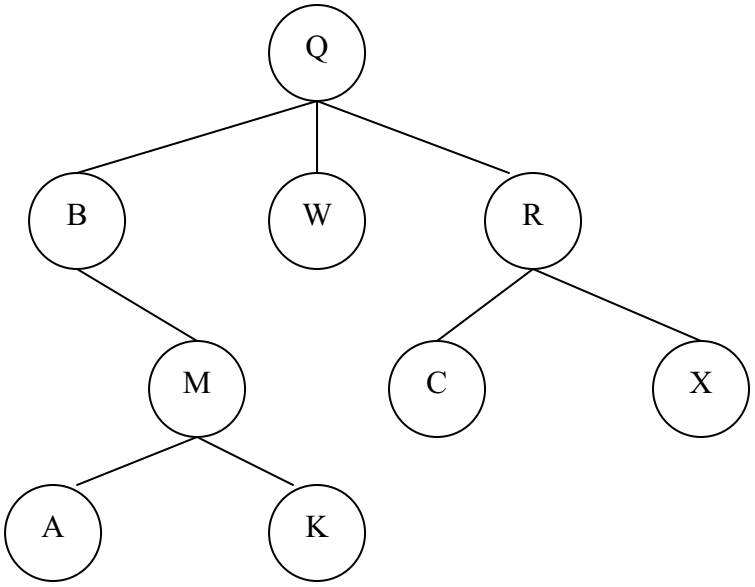
(b) What is the *worst-case* time needed by Quicksort to sort a list with n items? (circle)

$O(1)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

(c) What needs to be true about the implementation of Quicksort to ensure that the sort only requires the expected time instead of the worst-case time?

Question 8. (2 points) Two properties of a collection are its *size* and *capacity*. What is the difference between these two properties?

Question 9. (7 points) Consider the following tree, which is not a binary tree.



(a) Which node(s) is(are) the roots of this tree?

(b) Which node(s) is(are) the leaves of this tree?

(c) Write down the nodes in the order they are reached if we perform a *postorder* traversal of this tree starting with node Q.

Question 10. (8 points) If we want to create a binary tree whose nodes contain integer values, we can represent the nodes using instances of the following Java class.

```
/** binary tree node with integer node values */
public class BTNode {
   public int value;          // value contained in this node
   public BTNode left;        // left subtree; null if empty
   public BTNode right;       // right subtree; null if empty
}
```

Complete the definition of the following method so it returns the sum of the values contained in all of the nodes of the binary tree with root `n`. Hint: Recursion *is* your friend.

```
/** Return the sum of the values in a binary tree with root n */
public int sum(BTNode n) {




}
```

Question 12. (8 points) One common way to implement a list is a single-linked list containing a collection of nodes that refer to the data objects in the list. We can define the link nodes as follows:

```
/** One link in the list */
public class Link {
   public Object value;  // data value associated with this node
   public Link   next;   // next node in list or null if none

   /** Construct a new link with value v and next node n */
   public Link(Object v, Link n) {
      this.value = v;  this.next = n;
   }
}
```

A class using these links to implement a list would have an instance variable of type `Link` referring to the list data.

```
/** A simple list implemented with a linked list */
public class SimpleLinkedList {
   private Link head;    // first link in the list or null if
                         //    the list is empty
```

Complete the definition of method `addToEnd` below of class `SimpleLinkedList` so it adds a new value to the *end* of the list. You **may not** assume there are any additional instance variables in class `SimpleLinkedList`, and you **may not** add any.

```
   /** Add value v to the end of this SimpleLinkedList */
   public void addToEnd(Object v) {









   }
```

Question 13. (2 points) The infamous bouncing ball program contained code similar to the following to set up the buttons at the bottom of the window.

```
JButton stop = new JButton("stop");
JButton go = new JButton("go");
JPanel buttons = new JPanel();
buttons.add(stop);
buttons.add(go);
add(buttons, BorderLayout.SOUTH);
buttonListener = new SimButtonListener(world);
stop.addActionListener(buttonListener);
go.addActionListener(buttonListener);
```

How would the program's behavior change if the last two lines of code (the `addActionListener` method calls) were omitted?

Question 14. (4 points) A hash set or hash map can perform many operations like insert, delete, search (contains) and others very efficiently, this can only happen if the hash function and load factor have certain properties.

(a) What is a *hash function*? What must be true about the hash function for these operations to be very efficient?

(b) What is the *load factor* of a hash table? What must be true about the load factor for operations to be very efficient?

Java Reference Information

Feel free to detach these pages and use them for reference as you work on the exam. This information is identical to the reference information on the 2nd midterm. You may not need most (or even all) of it to answer the questions on this exam.

`class` *BufferedReader*

`String readline()`	Return next line from input stream, or `null` if no more input. Can throw `IOException`.

`class` *PrintWriter*

`void print(arg)`	Print arg to the `PrintWriter` stream. The parameter can be any type
`void println()`	Terminate the current output line and move to the beginning of the next. line
`void println(arg)`	Print `arg`, then advance to the beginning of the next line

`class` *String*

All of the search methods in class `String` return -1 if the item is not found

`int length()`	length of this string
`int indexOf(char ch)`	first position of `ch`
`int indexOf(char ch, int start)`	first position of `ch` starting from `start`
`int indexOf(String str)`	first position of `str`
`int indexOf(String str, int start)`	first position of `str` starting from `start`
`int lastIndexOf(char ch)`	last position of `ch`
`int lastIndexOf(char ch, int start)`	last position of `ch` searching backward from `start`
`int lastIndexOf(String str)`	last position of `str`
`int lastIndexOf(String str, int start)`	last position of `str` searching backward from `start`
`String substring(int start)`	substring of this string from position `start` to the end
`String substring(int start, end)`	substring of this string from position `start` to `end-1`
`String trim()`	copy of this string with leading and trailing whitespace deleted

All ***Collection*** interfaces (***List***, ***Set***) and classes (***ArrayList***, ***LinkedList***, ***HashSet***, ***TreeSet***)

```
boolean add(Object obj)
boolean addAll(Collection other)
void clear()
boolean contains(Object obj)
Iterator iterator()
boolean remove(Object obj)
int size()
Object[] toArray()   // return an array containing all the
                     // elements in this collection
```

In addition, all `Collection` classes provide a constructor that takes another `Collection` as a parameter and creates a new collection whose initial contents are copied from that parameter. (i.e., `public ArrayList(Collection c)`, and similarly for the other classes.)

Additional methods in ***List***, ***ArrayList***, ***LinkedList***

```
add(int position, Object obj)
remove(int position)
```

Map, ***HashMap***, ***TreeMap***

```
Object put(Object key, Object value)
Object get(Object key)
Object remove(Object key)
Set keySet()
Collection values()
int size()
```

arrays

If `a` is a Java array, `a.length` is the number of elements in that array.

If `m` is a 2-dimensional Java array, `m[k]` refers to row `k` of the array, and `m[k].length` is the length of that row (which is the same for all rows in a normal, rectangular array)..