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. (6 points) (a) Prove that $3n^2 + 25n + 17$ is $O(n^2)$.

(b) Prove that $3n^2 + 25n + 17$ is $O(n^3)$.

(c) Which of the two proofs in parts (a) and (b) of this question provides the most useful information and why?

Question 2. (2 points) When defining a new class in Java, the standard advice is that if we override the inherited `equals(Object)` method we should also override the inherited `hashCode()` method. What relationship should be true between the methods `equals` and `hashCode`?

Question 3. (4 points) A good implementation of a hash table (`HashMap` or `HashSet`, for example) provides $O(1)$ (constant time) insert and contains operations. Give *two* ***distinct*** reasons why these operations might be significantly slower, possibly proportional to the number of items in the hash table.

Question 4. (4 points) Suppose we have two packages `Cowboy` and `Graphics`, both of which contain a class named `Draw`.

(a) (2 points) Will the following code compile? Why or why not?

```
import Cowboy.*;
import Graphics.*;

public class Test {
   Draw d;

   /** Constructor */
   public Test() {
      d = new Draw();
   }
}
```

(b) (2 points) If the code in part (a) won't compile, how could we fix it so it will? [If the code in part (a) does compile, leave this part of the question blank for 2 free points!]

Question 5. (2 points) One of your colleagues is implementing a simple list class, which contains the following method specification.

```
/** Return the object at the given position
 * @param pos position of the desired object in the list
 * @return the selected object
 * @throws IndexOutOfBoundsException if pos is invalid
 */
public Object get(int pos)throws IndexOutOfBoundsException { … }
```

Is the "`throws IndexOutOfBoundsException`" clause in the method heading required, or can it be omitted? Why?

The next few questions involve single-linked lists of integers. The nodes in the linked lists are instances of class `Link`, defined as it was in lecture and in section.

```
public class Link {      // one node in a linked list
   public int item;      // data associated with this link
   public Link next;     // next node in the list; null if none

   /** Construct a new node referring to the given object */
   public Link(int item, Link next) { … }
}
```

The state of a `SimpleLinkedList` is represented by these instance variables:

```
Link first;      // first node in the list; null if empty
Link last;       // last node in the list, null if empty
```

Question 6. (5 points) You've been hired to work on Java 6 (or 1.6, or whatever marketing will eventually call it) and have been asked to write a method to sum up the values in a list, but for whatever reason your boss says that you can't use a loop. However, it's fine if you use recursion.

To compute the sum of the list we call the following method:

```
/** return the sum of the ints in the list */
public int sum() {
   return sumFrom(first);
}
```

Complete the definition of the following method so it returns the sum of the list starting at the given node. For full credit, you **may not** use iteration – use recursion instead.

```
/** return the sum of the ints in the list starting at node p */
public int sumFrom(Link p) {









}
```

Question 7 (9 points) Our simple array-based list implementation included an iterator class that provided the standard `hasNext()` and `next()` operations. For this question, complete the following methods to provide an iterator for the `SimpleLinkedList` class implemented as described on the previous page. You should assume that this iterator class definition is *nested inside* the `SimpleLinkedList` class definition, so it has direct access to any instance variables and methods in that class that it needs.

```
/** Iterator class for a SimpleLinkedList */
public class SimpleLinkedListIterator {

   // declare any instance variables you need here



   /** Construct a new SimpleLinkedListIterator */
   public SimpleLinkedListIterator() {




   }

   /** Return true if there are more items in this iteration */
   public boolean hasNext() {




   }

   /** Return the next item in this iteration.
    * @throws NoSuchElementException if no more elements */
   public int next() {










   }
}
```

Question 8 (8 points) Another possible use of linked data structures is to implement stacks (the same behavior as an array-based implementation, but with a different underlying data structure). For reference, the definition of a link is repeated here.

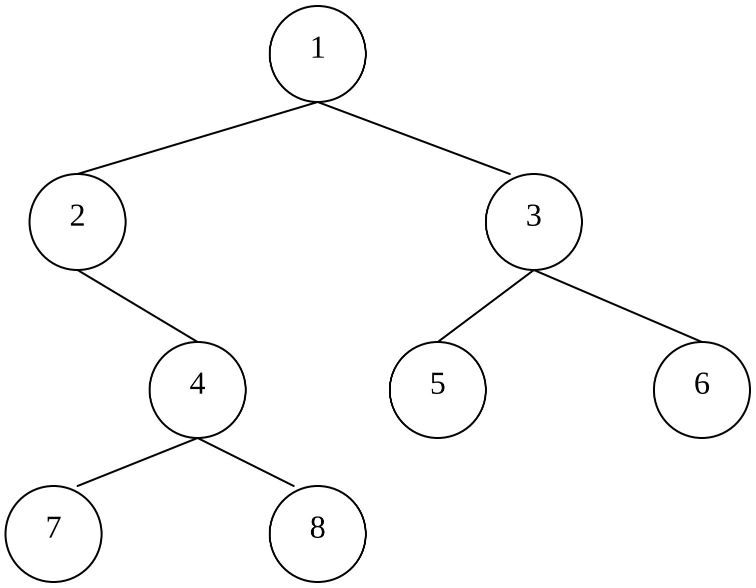
```
public class Link {        // one node in a linked list
   public int item;        // data associated with this link
   public Link next; // next node in the list; null if none

   /** Construct a new node referring to the given object */
   public Link(Object item, Link next) { … }
}
```

Complete the definitions of the constructor and methods `push` and `pop` below to implement a stack. The constructor and one instance variable are provided for you. You can add additional instance variables and modify the constructor if you need to.

```
public class SimpleStack {
   // instance variables
   private Link top;          // top of stack or null if the
                              //   stack is empty


   /** Construct a new empty stack */
   public SimpleStack() {



   }

   /** Push item onto the top of the stack */
   public void push(int item) {



   }

   /** Return the top item on the stack and delete it.
    * @throws NoSuchElementException if the stack is empty. */
   public int pop() throws NoSuchElementException {










   }
}
```

Question 9 (8 points) The tree traversals that we have looked at are known as *depth-first* traversals. The essential idea is that we completely visit a subtree of a node, including all the nodes in the subtree, before we visit the other subtrees of that node.

Another possible strategy is a *breadth-first* traversal. In this strategy, we first visit the root, then we visit all of the root's immediate children, then all the children at the next level of the tree, and so forth. For example, the nodes in the following tree are numbered in the order they would be reached in a breadth-first traversal.



Assume that we have a binary tree containing String data values whose nodes are represented as follows.

```
public class BTNode {      // one node in a binary tree
   public String value;    // value stored in this node
   public BTNode left;     // left subtree or null if empty
   public BTNode right;    // right subtree or null if empty
}
```

Complete the definition of method `btraversal` on the next page so it does a breadth-first traversal of the tree with the given root and prints out all the strings contained in the tree in breadth-first order. If you need additional helper methods, feel free to declare them. You can declare any instance variables that you need outside the `btraversal` method.

Hint: One useful strategy for doing this is to keep a queue of nodes that you have visited, but whose children have not been visited. Initially put the root in this queue, then visit the nodes in the queue one by one. Each time you visit a node, print out its contents and add its children to the end of the queue.

Second hint: Recursion may or may not be your friend. Take a couple of minutes to think about your solution strategy before you start coding.

Question 9 (cont).

```
   // Declare any instance variables you need here







   /** Perform a breadth-first traversal of tree t and
    *  print all of the strings in the nodes in the order
    *  they are reached */
   public void btraversal(BTNode t) {













}
```

Question 10. (10 points) This question involves processing a text stream to accumulate some statistics. The input data consists of one or more lines that each contain a last name, a first name, and the letter M or F to indicate the person's sex. For example:

```
Smith  Sue   F
Jones  Ralph   M
Moose Bullwinkle  M
Bird  Tweety  F
```

Complete the method `percentFemale`, below, so it returns the percentage of the number of lines in the stream where the sex is `F`. You can assume that the sex is always capitalized so you only need to check for `M` or `F`. You can also assume that there are no extra leading or trailing blanks or other characters at the beginning or end of the input lines, and you can assume that there is at least one line in the input stream.

Hint: The last couple of pages of the exam contain some summary information about Java stream and string classes that you might find useful.

```
/** Return the percentage of lines in the input stream where
 *  the third entry on the line is the string “F”.
 * @param in the input stream to read from */
public double percentFemale(BufferedReader in) {






}
```

Question 11. (8 points) We looked at several implementations of collections – arrays, linked lists, trees, binary search trees, and hash tables. Different operations on these data structures had different expected and worst-case times. Complete the tables below with the expected and worst case times (using O()-notation) for the different operations on the given data structures. You should assume that suitable instance variables are used to make operations reasonably fast instead of having to, for example always traverse the complete data structure to count the number of items in it.

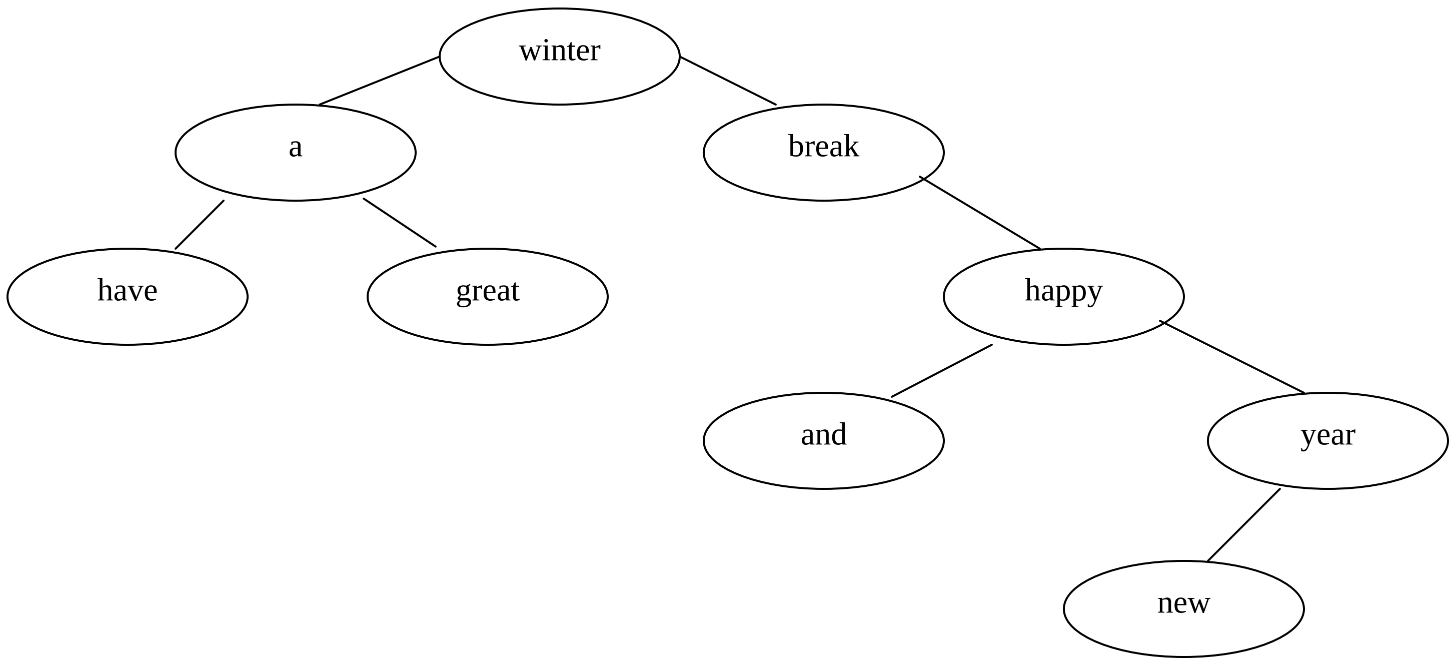
(a) **Expected times**

Operation	Sorted array-based list	Unsorted Linked list	Binary search tree	Hash set
Add item to the collection				
Search for item				
Return size of the collection				

(b) **Worst-case times**

Operation	Sorted array-based list	Unsorted Linked list	Binary search tree	Hash set
Add item to the collection				
Search for item				
Return size of the collection				

Question 12. (2 points) Write down the words in the following tree in the order they are reached in an ***inorder*** traversal.



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 other classes.)

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

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

Additional methods in ***LinkedList***

```
getFirst(), getLast(), addFirst(), addLast(), removeFirst(),
     removeLast()
// retrieve, add, and remove items at either end of the list.
// remove returns the item removed from the list
```

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).

Exceptions

Some standard exceptions that might be useful: `IllegalArgumentException`, `IndexOutOfBoundsException`, `NoSuchElementException`, `NullPointerException`.