Imperial College of Science, Technology and Medicine	Department of Computing
Computing Science (CS) / Software Engineering (SE)	BEng and MEng
Examinations Part I	Integrated Laboratory Course

Laboratory work is a continuously assessed part of the examinations

and is a required part of the degree assessment.

Laboratory work must be handed in for marking by the due date.

Late submissions may not be marked.

Exercise: 16	Working: Individual
Title: Java Database	
Issue date: 9th March 2009	Due date: 16th March 2009
System: Linux	Language: Java

Aims

- To gain further experience of *abstract data types* (ADTs) in Java.
- To familiarise yourselves further with Java 1.6's API and Generic Types.
- To illustrate OO concepts such as polymorphism and highlight the advantages of programming with interfaces.
- To practice the design principles taught in the Abstract Data Type course.
- To introduce unit testing using **junit** in Java.

The Problem

- Your task is to *implement* two abstract data types (**ADT**s) for use in a database. The database stores information about the number of cats living in a cattery; in particular, the name of each cat is associated with the number of tins of cat food each cat eats per week. The database is ordered alphabetically by cat name with the pre-condition that there are no duplicate names. The ADT implementations allow the database of cats to be managed using an ordered linked list and a binary search tree. You can then *compare* the efficiencies of the two implementations using the *benchmarking* tool provided.
- You are provided with a number of classes, interfaces and packages that your classes support. These are:
- The main program class **CatAdmin** which is the menu driven interface to the database and provides a simple menu-driven facility for performing operations on the database (where NAME is a String and TINS is an integer):

```
add NAME TINS  // Add a cat to the database
               // If the cat is already present
               // change the number of tins to TINS

lookup NAME    // Print the number of tins eaten by NAME

clear          // Remove all entries

count          // Print the total number of cats and tins

delete NAME    // Remove the cat NAME from the database

print          // Print out the database

help           // Print the help menu

quit           // Quit the database


switch         // Toggle the database type used from
               // BinarySearchTree to OrderedLinkedList and back
               // Note the BinarySearchTree is used initially

debug          // Print the debug menu

load NUMBER    // Reload the database with NUMBER cats
               // Where NUMBER is a non-negative integer

benchmark      // Benchmark the current implementation
```

- You are provided with the class **Database** that contains an encapsulated **Dictionary** together with the methods called by **CatAdmin** needed to modify it. The **Database** class calls the methods provided by the **Dictionary** class to modify the **Dictionary**. It also catches and processes the exceptions thrown by the implementations of **Dictionary**.
- The **Dictionary** is an **interface** containing headers for the methods that you need to write. This class is generic so you should call it using a **String** and **Integer** as parameters. As the **Dictionary** extends the **Iterable** interface any implementation of this class needs to provide an **interator()** method that returns an **Iterator**.
- Also provided is the **DictionaryEntry interface** which contains methods that allow the program to get the key and value associated with each entry (i.e. cat name and number of tins eaten) in the case of its usage by **Database** on behalf of **CatAdmin**.
- You are required to write four implementation classes for the interfaces **Dictionary.java** and **DictionaryEntry.java**. You should implement these interfaces using a Binary Search Tree in **BinarySearchTree.java** and **BinarySearchTreeEntry.java**, and using an Ordered Linked List in **OrderedLinkedList.java** and **OrderedLinkedListEntry.java**. Each of these classes should be in the **database** package.

- Once you have written these implementations you should test the relative efficiency of your implementations using the benchmarking tools provided (see below).
- Your **BinarySearchTree** and **OrderedLinkedList** classes should provide the operations needed to process the database. These include an **Iterator** so that the user can step through the database to print out the database and count the total number of tins needed by the cattery each week.
- You are also provided with a partial **junit** test file for the **OrderedLinkedList** class - **TestOrderedLinkedList**. This contains some simple tests that can be carried out on each of the methods you will be writing to implement the **OrderedLinkedList** class. You should design and add further tests and comment them so that your PPT/UTA can understand how you designed them.

The interface classes that you need to implement

- The **Dictionary** interface which provides the top level methods for operations on the database is given below. You should note that this interface uses generics.

```
public interface Dictionary<K extends Comparable<K>, V>
extends Iterable<DictionaryEntry<K, V>> {

        // Returns the number of key-value objects stored in this dictionary.
        public int size();

        // Tests if this Dictionary contains key-value objects.
        public boolean isEmpty();

        /*
         * Returns the value associated with the key. There
         * can be at most one value associated with each key.
         *
         * throws NoSuchElementException if the specified key does not exist in
         * the dictionary.
         */
        public V get(K key) throws NoSuchElementException;

        /*
         * Creates a dictionary entry associating the given key and
         * value. If the dictionary previously contained an entry for
         * this key, the old value is updated with the given value.
         */
        public void put(K key, V value);

        /*
         * Removes the entry for the key from the dictionary if it is present.
         *
         * throws NoSuchElementException if it is not present
         */
        public void remove(K key) throws NoSuchElementException;
```

```
        // Removes all entries from the dictionary.
        public void clear();
}
```

- Although these access methods contain no specific information about the data structure implementing the dictionary, some constraints are present. The Dictionary should be instantiated with two kinds of objects, representing key and value respectively; and instances of the key-type, `K`, must be `java.lang.Comparable` [1] objects - this is reflected in the interface declaration. You do not need to implement the **Comparable** interface in the **OrderedLinkedListEntry** and **BinarySearchTreeEntry** classes by providing the code for **compareTo** as this method is already available for the **String** class.
- Each concrete implementation of this interface is required to have an `iterator()` method that returns a `java.util.Iterator` [2] object. Your implementation of the iterators should traverse the ADT returning `DictionaryEntry` objects in ascending order - by key. You should implement the Iterator as an *inner class* to the classes **BinarySearchTree** and **OrderedLinkedList** as the iterator needs to access the data structure in the outer class.
- The **DictionaryEntry** interface provides the methods for operations on a database entry. You should note that this interface uses generics.

```
public interface DictionaryEntry<K, V> {

        // return the key stored in this entry.
        public K getKey();

        // return the value stored in this entry.
        public V getValue();
}
```

Submit by Monday 16th March 2009

What to Do

- Copy the test harness and main program files **CatAdmin.java**, **Database.java**, **Dictionary.java**, **DictionaryEntry.java** and **TestOrderedLinkedList.java** in the package **database**; **Benchmark.java** and **Counter.java** in the package **database.benchmark**; and the file **benchmark.jar** in the top level directory, using the command **exercise 16**.
- This should give you the files you need to write your program in the appropriate directory structure.
- Write **OrderedLinkedList.java** and **OrderedLinkedListEntry.java** in the **database** package to implement the interface **Dictionary.java** as an Ordered Linked List.
- Design and add some further **junit** tests in **TestOrderedLinkedList.java** while you are writing **OrderedLinkedListEntry.java** and add comments on your tests.

[1] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html
[2] http://java.sun.com/j2se/1.5.0/docs/api/java/util/Iterator.html

- Write **BinarySearchTree.java** and **BinarySearchTreeEntry.java** in the **database** package to implement the interface **Dictionary.java** as a binary search tree.
- Launch the test harness with the command:

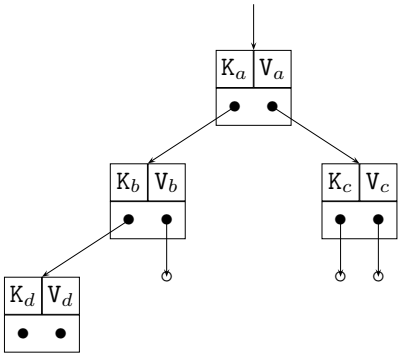
```
java -javaagent:benchmark.jar database.CatAdmin
```

- Using the `debug` menu command from the Cat Administrator menu for each ADT find the complexity of the lookup ('get') operation using the Cat Administrator's `load` and `benchmark` commands. Include your findings as a comment at the top of each implementation class.

The BinarySearchTree and OrderedLinkedList classes

- You will not be able to compile the Cat Administrator tool, **CatAdmin.java**, until you have implemented *both* ADTs. We therefore recommended that you create stub classes with dummy methods (see the `database.Dictionary` and `java.lang.Iterable`[3] interfaces) to allow incremental development.
- The node representations for each ADT should implement the `database.DictionaryEntry` interface.
- Java provides a generic `Iterator`⟨`T`⟩ interface for traversing elements in collections (where all elements in the collection are traversed only once!). Note that the method **printDatabase** uses an Iterator to access the contents of the database in ascending order. A concrete implementation of an iterator, `Iterator<DictionaryEntry<K,V>>` provides implementations of the following three methods:
  - `boolean hasNext()`: tests if the iteration has more elements, yet to be traversed.
  - `DictionaryEntry<K,V> next()`: returns the next `DictionaryEntry<K,V>` element in the iteration; for this lab, the elements should be returned in ascending order by `K`.
  - `void remove()`: removes the previously returned element from the underlying data structure. (You should *not* implement this method, rather you should provide a stub so that your program compiles.)
- The binary search tree iterator should perform an in-order traversal. Given that the binary search tree is ordered, the iterator's `next()` method should return the nodes of the binary search tree in the "left, top, right" order for example: (K_d,V_d), (K_b,V_b), (K_a,V_a), (K_c,V_c) in the diagram. Pseudo-code for this algorithm is given below.

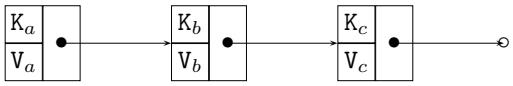
[3] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Iterable.html



globals:
```
Stack workList;
Node current = root of tree;
```

code:
```
while(current!=null){
    push current node on to workList
    current = current.left
    }
if(workList not empty){
    RESULT = pop node off the workList;
    current = RESULT.right
    }
return RESULT
```

- The ordered linked list iterator should perform a linear traversal. The `next()` method should return the nodes of the OrderedLinkedList in the order: (K_a,V_a), (K_b,V_b), (K_c,V_c). Pseudo-code for this algorithm is given below.



globals:
```
Node current = head of list
```

code:
```
if(current!=null){
    RESULT = current
    current = current.next
    }
return RESULT
```

Introducing Unit Testing

- Unit testing is a design method which allows code testing to be done whilst the code is being designed and written. This allows errors to be found and corrected rapidly compared to leaving testing to after the code is written. Unit testing should save development time and encourage correct code to be written. Although it may take time to design and write the tests it should save time in the long run.
- In Java programs the unit tested is the class and individual method. If possible each method should be tested independently of other methods.
- Once the tests are written their existence also helps with program maintainence since the developer can modify existing code and check the modified code with the unit tests.
- Java provides the **junit** framework to support unit testing in Java. This allows the developer to write specialised test programs which when executed test each class and method in the program being developed. Generally each class has a corresponding test class and each method a corresponding test method. We have provided some test code that can be run with **junit** in the class **TestOrderedLinkedList**. This contains tests for the various methods in **OrderedLinkedList**.
- These tests should provide examples of how to write tests and you should design and add further tests as you write your **OrderedLinkedList** class.

- To run the tests compile the **TestOrderedLinkedList** class in the usual way using **javac**. Note that this class imports: `import junit.framework.*;`. Then you should use - **java junit.textui.TestRunner database.TestOrderedLinkedList** to execute the tests and if they were all successful the output should look like:

```
........
Time: 0.037

OK (8 tests)
```

  (The number of tests you can run depends on number of methods you have written and you should comment out any tests you don't want run - see discussion of the **TestSuite** below.) If any tests fail the program prints a stack trace and information about which test has failed so that you can then fix the code before trying the tests again.

- As an example take the test for **public V get(K key)**:

```
public void testget(){
        d.put("Tiddles", 10);
        assertTrue("Gettin on item failed", d.get("Tiddles") == 10);
```

- You will see that the test consists of an assertion and most **junit** tests consist of assertions like:

```
assertEquals([String message], expected, actual)
// where expected and actual can be either primitives
// or classes with an equals method defined
// uses equals() method if defined

assertNull([String message], java.lang.Object object)
assertNotNull([String message], java.lang.Object object)
// Object tested to see if it is null or not null

assertSame([String message], expected, actual)
assertNotSame([String message], expected, actual)
// asserts that expected and actual refer to the same or different objects

assertTrue([String message], boolean condition)
assertFalse([String message], boolean condition)
// asserts that the boolean condition is true/false
```

- The two methods

```
protected void setUp()
protected void tearDown()
```

  are run to initialise and clean up before and after each test.

- You can test that exceptions are thrown as specified:

```
public void testgetException() {
        try{
                d.get("Tiddles");
                fail("Looking up cat not in database should cause exception");
        }
        catch (NoSuchElementException e){
                assertTrue(true);
        }
}
```

- If a test suites is provided (as below) the test methods included are run otherwise all the test methods are run.

```
public static Test suite(){
        TestSuite suite = new TestSuite();
        suite.addTest(new TestOrderedLinkedList("testisEmpty"));
        suite.addTest(new TestOrderedLinkedList("testsize"));
        suite.addTest(new TestOrderedLinkedList("testclear"));
        suite.addTest(new TestOrderedLinkedList("testput"));
        suite.addTest(new TestOrderedLinkedList("testget"));
        suite.addTest(new TestOrderedLinkedList("testremove"));
        suite.addTest(new TestOrderedLinkedList("testgetException"));
        suite.addTest(new TestOrderedLinkedList("testListIterator"));
        return suite;
}
```

- As you write the **OrderedLinkedList** class you should add test methods to the test suite as appropriate.
- You can find further material on **junit** on the web such as:
  http://supportweb.cs.bham.ac.uk/documentation/tutorials/docsystem/build/tutorials/junit/junit.html

Benchmarking

- The benchmarking tool hooks on to your implementation classes and counts, specifically, the number of times the method `DictionaryEntry.getKey(..)` is invoked during the execution of the `Dictionary.get(..)` method. For correct results, you **must** ensure your implementations access a DictionaryEntry's key *only* via the `getKey(..)` method.
- Additional commands for the Cat Administrator tool are available through the `debug` menu. For each ADT implementation, plot a graph of the (average number of lookups) vs. (size of database) for varying database sizes. You may find the following helpful.
  1. The `switch` command can be used to toggle the 'active' ADT implementation used by the Database: BinarySearchTree or OrderedLinkedList.
  2. The `load` command can be used to populate the database with a specified number of entries. e.g. the command `load 10000` will populate the database with 10,000 entries.
  3. The `benchmark` command will calculate the average number of lookups to retrieve an arbitrary DictionaryEntry from your implementation of the active ADT.

4. Using your dataset (or otherwise), deduce the complexity of the lookup operation for each of the ADTs.

- (Optional) Was the average number of lookups for the binary search tree what was expected? If not, explain the discrepancy and amend your implementation based on your findings. You can include your answers to this as a comment at the top of your **BinarySearchTree.java** file.

Submission

- Submit **OrderedLinkedList.java**, **OrderedLinkedListEntry.java**, and **BinarySearchTree.java**, **BinarySearchTreeEntry.java** and **TestOrderedLinkedList.java** in the usual way using the **CATE** system.

Assessment

```
OrderedLinkedList                1.0
OrderedLinkedListEntry           0.5
Iterator for OrderedLinkedList   0.5
Tests for OrderedLinkedList      0.5

BinarySearchTree                 1.0
BinarySearchTreeEntry            0.5
Iterator for BinarySearchTree    1.0

Design, style, readability       5.0

Total                           10.0
```