There will be 5 laboratory sessions of 2 hours each for this course.
The exercises will involve the invention and implementation of algorithms for given computational tasks, and assessing their performance. Your algorithms should, of course, be correct - do what they are designed for - but, in addition, you should consider efficiency of algorithms - how well they perform on various inputs. You will be asked to represent data structures in programming languages and use data abstraction as a program design and structuring tool.
The exercises involve the implementation of algorithms and data structures in Java. You should have your first year material to hand so that you are prepared to use them in the exercises.
For all the exercises you should prepare the material (algorithms, data structures and outline code) before the laboratory session. If not, you will be wasting your own time and your laboratory session, and you are likely to miss deadlines.
The lab exercises are:

Lab exercise 1 List processing in Java. [1 lab session]
Lab exercise 2 Timing analysis of algorithms. [2 lab sessions]
Lab exercise 3 Data abstraction and data structures. [2 lab sessions]
COMP20012 Lab Exercise 1: Linked Lists

DURATION: 1 lab session

A version of this document can be found on the Web at http://www.cs.man.ac.uk/~graham/COMP20012/labs/labs.pdf. It’s worth checking there as it will possibly be more up to date than the paper version.

1.1 Purpose

This is an exercise in programming algorithms for list processing. You will be required to implement Java classes and methods which use a recursively defined Linked List class.

1.2 Background

The Linked List, which was introduced in first year Java courses, is a widely used data structure with the property that data can be inserted at any point in the list. The list grows and diminishes in size as items are added and deleted. Each object in the list consists of a piece of data (an object reference) and a reference (or pointer) to the next object in the list, as shown below. You should define a class ListNode, to implement such objects.

Adding an element to the front of such a list is done as shown below,
Note that an insertion at the front of the list will *always* modify the value of the reference to
the first element in the list (denoted in the diagrams by `front`). This means that, if the method
were to be called `add`, we would have to invoke it as

```java
front = front.add("five");
```

In order to create an object which encapsulates the list we need to introduce a LinkedList object,
which contains a reference to a ListNode, giving the structure shown below:-

If an object of this type were called `list1`, we could then safely write

```java
list1.add("five");
```

since the *value* of `list1` would not be modified by the insertion, merely the value of one its
instance variables.

Insertion of a new ListNode into the middle of a list is done by manipulating pointers as shown
below:-

1.3 Tasks

You are required to implement the classes ListNode and LinkedList together with the following
methods in the LinkedList class, all of which should be suitably tested.

1. `isEmpty` - tests whether a list is empty; returns `boolean`
2. `add` - adds a new entry at the front of the list, containing a reference to the Object \( x \); return type `void`.
3. `makeEmpty` - makes an existing list empty; return type `void`.
4. `printList` - prints the contents of the list. You can choose the exact format to be used; return type `void`.
5. `remove` - remove the first occurrence in the list of an entry containing a reference to an object equal to the Object \( x \); return type `void`.
6. **removeAll** - remove all occurrences in the list of entries containing a reference to an object equal to the Object \( x \); return type void.

7. **addLast** - adds a new entry at the end of the list, containing a reference to the Object \( x \); return type void. (You may want to consider a modification of the class definition to optimise this operation.)

8. **addAfter** - adds a new entry containing a reference to the Object \( x \) after the first occurrence of an element containing a reference to the Object \( y \) (or at the end of the list if no such occurrence exists); return type void.

9. **append** - appends the contents of the LinkedList \( \text{other} \) to the end of the list; return type void.

10. **printListRev** - prints the contents of the list in reverse order; return type void. This must be implemented without modifying the internal structure of the list (Hint: think recursion)

### 1.4 Marking scheme

<table>
<thead>
<tr>
<th>Class structure</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty, makeEmpty</td>
<td>2</td>
</tr>
<tr>
<td>add, addLast, addAfter</td>
<td>3</td>
</tr>
<tr>
<td>remove, removeAll</td>
<td>2</td>
</tr>
<tr>
<td>append</td>
<td>1</td>
</tr>
<tr>
<td>printList, printListRev</td>
<td>2</td>
</tr>
<tr>
<td>Suitable tests</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>15</td>
</tr>
</tbody>
</table>
COMP20012 Lab Exercise 2: Timing and Complexity

DURATION: 2 lab sessions

A version of this document can be found on the Web at http://www.cs.man.ac.uk/~graham/COMP20012/labs/labs.pdf. It’s worth checking there as it will possibly be more up to date than the paper version.

The laboratory exercise 2 for COMP20012 consist of TWO laboratory sessions, Sessions 2 and 3 of your laboratory times.

The exercises aim to develop your insight into the performance of algorithms. We shall look at both the actual running time of algorithms (in milliseconds of CPU usage) and the complexity of algorithms (in terms of operations performed), and compare and contrast these.

The main emphasis is not so much on coding but is on understanding the performance of the algorithms and why they perform well and poorly on different inputs. Marks are allocated not just for running code but for your understanding and explanation of the performance of algorithms. So be prepared to give clear explanations to the demonstrator. To understand the performances you may need to consult textbooks in the area.

We shall deal with the process of sorting of lists of integers into ascending order and shall compare the performance of three sorting algorithms in Java: Insertsort, Mergesort and Quicksort.

We are interested in two aspects of performance:

1. How the performance varies as the size of the input increases, i.e. the rate of growth of the performance measures.

2. How the algorithms perform on special kinds of lists, e.g. those already ascending, or descending order, and those in which there are few items repeated many times.

For the rate of growth we will display the results as graphs of performance as input size varies.

You should produce your methods in a file Complexity.java in class Complexity.

Task 1.

In order to assess the performance as the size of the input lists increases, you are asked to provide a routine for generating lists of numbers of arbitrary size whose items are randomly generated.

To do so you need to generate random numbers. Random number generators can be built using arithmetic operations. Strictly speaking we should call these pseudo-random number generators. There is a considerable literature on them: You can consult the course textbook Data Structures and Problem Solving using Java by Weiss, Chapter 9. Other accounts are, for example, Sedgewick’s Algorithms or Knuth’s books, where techniques for generating pseudo-random numbers and their properties are discussed.

In Java, to generate random numbers you may either use the class Random in the API and its methods, or the method random in package Math (see API documentation for details). The latter is a wrapper for instantiating Random. You call it as

```java
double d = Math.random();
```

and this returns a (pseudo)random number greater than or equal to 0.0 and less than 1.0.

Your task is to write a method to generate a list (represented as an array) of random integers (in the range 0...9999 say, ie 4-digit integers). The length of the list should be a parameter that we can change to assess the running time of the various sorting algorithms.

1.4 Marking scheme
Task 2.
We shall also need particular types of lists (as arrays). Write a method to generate ascending and descending lists of integers (simply consecutive integers will do).
Also write a method to generate lists which consists of a few items many times repeated. To do this you may use the random list generator from Task 1, but restrict the possible integers to a small range, say 0-9.

Task 3.
Provide code for the three sorting methods Insertsort, Mergesort and Quicksort. You may write these from scratch, or get them from textbooks, or from Java libraries or from websites dealing with algorithms.
Versions of these sorting algorithms may be found in the directory

/opt/info/courses/COMP20012/labs/lab2.

However you come by them, you must understand how they work. For each method, there are various ways of coding it, for example Insertsort may be iterative or recursive; there are various ways of choosing the pivot elements in Quicksort; etc. For Quicksort your choice of pivot should include at least the first item in the list (you may wish to implement other choices too).

Task 4.
Perform timing analysis on the three sorting algorithms on the various types of lists above (random, ascending, descending and few items).
To find the running time of a program you should use one of the Java timing classes. One suggestion is to use the class Duration which we supply. This class provides a timer for algorithms returning the CPU time in milliseconds. You should be aware that the actual CPU time can be affected not just by the algorithm running, but by other processes which may interrupt, so try to minimize other processes running at the same time.
The class Duration is available in /opt/info/courses/COMP20012/labs/lab2.
You can access it (as usual) by

javac -classpath /opt/info/courses/COMP20012/labs/lab2 Complexity.java

The documentation for this class is in this directory in file Duration.html.
You should run each sorting algorithm on the various types of lists of increasing lengths. The actual lengths of the lists for realistic timing in milliseconds depend (of course) on the speed of your processor. A maximum length of 5,000 or 10,000 may be appropriate, but on slower processors this may be too large. Experiment! Create about 10 lists of lengths evenly spaced up to the maximum.
Store each set of results in a file which consists of pairs of lengths and running times, one pair per line, such as:

400 830
800 1510
... etc

Since you need to run various algorithms on various types of list with various sizes, I suggest (for no extra marks), that you provide command-line arguments to determine which of these cases to run.

Task 5.

1.4 Marking scheme
Plot graphs of the timing of each algorithm against the size of the input list.
You may use any plotting method. A simple one is to use the Linux utility `gnuplot`. Type `gnuplot` and at the prompt, type `plot "datafile"` where `datafile` is a file that you have created with a list length and running time on each line (as above). To obtain a printed copy of such a plot, do not just grab a screen image of the `gnuplot` output, this creates enormous files which block up the printer queue. You should (in `gnuplot`)

```plaintext
set terminal postscript
set output "myfile.ps"
plot ..... 
```
and then print the file using `lpr`, after using `gv` to check that it looks right.

**Task 6.**
This task measures the performance of algorithms by counting how many operations are executed during processing. This is the time complexity of an algorithm. We compare this to the running time as computed in the previous tasks.
Can we know simply by looking at an algorithm how long it is going to take to run? If so then we may be able to design algorithms to have better performances. The answer is “yes” and the idea is to choose operations to count, operations which we consider will take a lot of processing time, and count them! That is, count how many times they will be executed whilst the algorithm runs. In the lectures, we show how we can count them *without running the algorithm*! Here we support this analysis by explicitly counting operations whilst the algorithm runs and comparing it to the running time of the previous exercise. We use the three sorting algorithms of the previous exercise.

We count the number of operations in the sorting algorithms. Which operations? There are several possibilities (and it turns out it doesn’t matter much). The standard is to count the order operation by which we compare elements.

Modify your sorting algorithms to count the number of comparison operations performed.
Run the modified algorithms on the same lists as for the timing analysis and plot the count of the number of operations against the length of the lists. How does it compare to the timing analysis?

**Task 7.**
Be prepared to explain to the demonstrator all the graphs of timing and complexity that you produce.
For marking, you need to show the demonstrator your code for the algorithms, show that they work and explain why. You will also need to explain the shape of the graphs: Look at the timing graphs for random lists. Clearly some algorithms grow slower than others as the size of the input increases. Simple algorithms are not always the best! Can you see what sorts of rates of growth are involved? For example: are they linear - a straight line, quadratic - grow as \( N^2 \) where \( N \) is the length of the list, or cubic, or linear-logarithmic \( N \times \log(N) \), etc.? Hint: a linear algorithm doubles its running time as the size of the input doubles. A quadratic one quadruples (times 4) its running time as the size of the input doubles, and a cubic is times 8 etc. Also note that for short lists the rate of growth may differ from that of longer lists, and note that some graphs are more regular than others. Why is this? Consider also the behaviour on the special forms of lists. Explain each of these and compare them to the behaviour on the random lists.

Examine the graphs of time complexity and explain how they compare to the timing analysis.
You need not write down your explanations but you should be able to explain clearly to the laboratory demonstrator.
Marking, out of 15 marks:

- Code for generating lists and for timing: 6 marks
- Code for time complexity via counting operations: 1 mark
- Explanations of timing and complexity on random lists: 4 marks
- Explanation of timing and complexity on special lists: 4 marks

You should labmail your file Complexity.java.
COMP20012 Lab Exercise 3: A simple spell checker

DURATION: 2 lab sessions

A version of this document can be found on the Web at http://www.cs.man.ac.uk/~graham/COMP20012/labs/labs.pdf. It’s worth checking there as it will possibly be more up to date than the paper version.

3.1 Purpose

This lab is an exercise in implementing some common datatypes, together with a framework in which they can be used.

3.2 Background

The exercise is to implement a very simple spelling checker which uses a set in which to store all the words in a dictionary. A text is then read and words which are in the text which can not be found in the dictionary are reported to the user, together with the line numbers on which these words occur in the text. The dictionary is to be contained in a collection which implements the following very simplified collection interface

```java
public interface Coll {
    public boolean add(Object o);
    public boolean contains(Object o);
    public void printColl();
}
```

The behaviour of the `add` and `contains` methods should be the same as the corresponding methods in `Set`. `printColl` should print the contents of collection to standard output in a suitable manner.

3.3 Tasks

Your tasks for this lab are to provide two different implementations of this interface, both using `Hash Tables`, one using `Open Hashing` and the other using `Closed Hashing`. Although these classes are to be used to hold `String` data, they should be general enough to contain any `Object`. You do not need to implement any hash functions, you should use the method `hashCode` provided in the JDK.

You should then implement a class `SpellCheck` which uses either of these implementations to produce a simple spell checker, as described above.

There is just one deadline for this lab, at the end of the second lab session, when both implementations will be marked together.
Task 1: Open Hashing based implementation

The first task is to create a class HashSetOpen which implements the Coll interface using hash tables based on open hashing, so that collisions are dealt with by using a list of entries. You can probably reuse a modified form of your LinkedList code from lab 1 for list handling purposes. In addition to the methods required for the interface you will need suitable constructors and a main method which includes code to test your methods.

Task 2: Closed Hashing based implementation

The second task is to reimplement the Coll interface using hash tables based on closed hashing, so that collisions are dealt with by using a collision resolution mechanism, such as linear probing. You could alternatively use quadratic probing or double hashing. This class should be called HashSetClosed. For this style of hash table you will also need to implement a rehash method, which is invoked by add when the the table reaches a predetermined load factor (0.5 is a suitable value), otherwise the table risks overfilling. Your implementation should, of course, include suitable testing. For both types of hash table it is preferable to use hash tables whose size is a prime number. A class Primes is provided in /opt/info/courses/COMP20012/labs/lab3/Primes.java, to help you identify suitable prime numbers. In particular the method nextPrime returns the first prime number greater than or equal to its argument.

Task 3: The Spell Checker

The final task is to write a class SpellCheck which uses either of these implementations to produce a simple spell checker, as described above. The text file to be checked should be specified via a command line argument, as should the Coll implementation to be used and the dictionary. One possible example of how this might be used is:-

```
java SpellCheck -o -d /usr/share/dict/words test-text
```

This should use the dictionary /usr/share/dict/words and the open hashing based hash table implementation to spell check the file test-text. The simpler command

```
java SpellCheck test-text
```

might use a default dictionary and text file.

The recommended way to switch between the two implementations is to have a SpellCheck constructor which takes an argument of type Coll which is then used to store the dictionary. This can then be used in conjunction with a method to do the spell check, which can be invoked by that object.

To populate the dictionary and also to read the text to be checked, you need to read a file and identify the words in it. For this purpose we regard a word as a sequence of consecutive alphabetic characters. Any non-alphabetic character can be regarded as a separator. You may find the StringTokenizer class useful here. The dictionary is just a text file containing white space separated words. It may, or may not, have only one word per line.
A sample dictionary and text file to be spell checked can be found in the directory /opt/info/courses/COMP20012/labs/lab3/, together with a Java file containing some examples of command line handling. There is a good tutorial on this subject at http://www.ecs.umass.edu/ece/wireless/people/emmanuel/java/java/cmdLineArgs/cmdLineArgs.html, from where the above Java source was obtained.

The output from the program should be in the form

3: thiss
4: iss
5: ze
7: owtput

where the numbers are the line numbers in the file where the wrongly spelt words occur.

Your spell checker should also deal sensibly with the fact that words are sometimes written in a mixture of upper and lower case, so that, if word is in the dictionary, the checker should not complain about Word, or even wOrD. This aspect should be handled in the spell checker itself, and not in either of the HashSet implementations.

Optional extras

There are many ways in which this program could be enhanced. One optional extra (not assessed) is to give your spelling checker the capability to read from a user’s personal dictionary in addition to the system dictionary, and to add ‘incorrect’ words to this personal dictionary if the user so chooses.

Another is to add extra constructors to the SpellCheck class to use the implementations of Set provided by the JDK, for example HashSet and TreeSet; this would also involve creating small wrapper classes which implemented Coll.

3.4 Marking Scheme

<table>
<thead>
<tr>
<th>Description</th>
<th>Marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>HashSetOpen implementation</td>
<td>6</td>
</tr>
<tr>
<td>HashSetOpen testing</td>
<td>1</td>
</tr>
<tr>
<td>HashSetClosed implementation</td>
<td>6</td>
</tr>
<tr>
<td>HashSetClosed testing</td>
<td>1</td>
</tr>
<tr>
<td>Basic SpellCheck implementation</td>
<td>3</td>
</tr>
<tr>
<td>Command line handling</td>
<td>2</td>
</tr>
<tr>
<td>SpellCheck testing</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>20</strong></td>
</tr>
</tbody>
</table>