Algorithms and Data Structures
Week 1: Recap, and an overview of what's to come

Abstract

Summary: A recap of simple Java, and what you need to get started. Looking over some basic data structures (Linked lists) and algorithmic implementation (recursion). A discussion of why we care about the theory of algorithms and how to use this knowledge in the design and implementation of large resource-intensive applications.

· **Book reference**: Drozdek chapters 1 and 3;

· **Resources**: Program bundle containing sample programs discussed in lectures.

· **Practical and tutorial exercises**: There are no mixed classes in Week 1.

· **What you need to submit by Monday Week 2**: There is nothing to hand in this week. Do make sure however that you are up to speed with your practical programming in Java using Eclipse. If you are in doubt, please try some of the small programming exercises at the end of these notes and contact your tutor if you have difficulty completing them.

Quick revision of Java

Although different programming languages appear to be very different, they often share some basic core principles. C++ and Java for example share some basic structure (and even syntax), so if you are already familiar with one of them (eg C++ that you learned in previous years) it's worthwhile reviewing what those core principles are: it will make learning the new language (i.e. Java) much simpler!

Two of the key ideas to remember when programming in Java is that first, whenever you write a program, you will be required to create a class; second is that all variables (apart from basic types) are actually references, and so a bit like C++'s "pointer" variables. This means in method calls, for example, the variables are passed by value (it's just that the value is a reference to memory where the object is located).

These ideas are actually present in C++, the only difference is that *in Java they are fundamental*. Once you have understood these concepts and their implications however, many of the basic programming features in C++ transfer quite simply to Java.

The table below sets out a crib for some of these core principles shared by both Java and C++. For both C++ and Java, the basic libraries and examples of syntax are given. The first tutorial exercises (for completion by the end of Week 2) will reinforce these core language principles so that, if this is your first time programming Java, you should be able to use what you already know to help your learning.

Linked lists

In this week's resources you'll find a zip file containing the material we will be using for this week's lectures, tutorials and practicals. Download it and open in eclipse. In it you will find `SingleLinks.java`, `LinkedTests.java` and `StopWatch.java`. In this section we discuss `LinkedTests.java`, with the other files discussed below.

Concepts

A Linked list is a list-like data structure useful for storing a whole set of information. Arrays are simple and easy to use because of the "indirect addressing". But this convenience comes with the price of inflexibility: arrays are static, cannot be extended or reduced in size and data cannot easily be inserted or removed from the middle of an array. However, it's nice to be able to write something like;

```
for(int i=0; i< N; i++)
    A[i]= SOMETHING;
```

and this pattern is familiar with array-based programming.

Linked lists don't have the inflexibility problem, but there is an overhead on the programmer to keep track of its necessary *direct* addressing, and we can't use the above pattern directly. (However in the Java Library you'll find an "iterator" class which does indeed allow us to use something that's almost like it!)

Below we'll be programming with a special linked list class based on the one used in the standard text for this course. This is so that you can understand exactly what is involved in using linked lists, so that when later if you need to use the standard library class you'll have the understanding to enable to get the best out of it in your programming.

Basics

A linked list is made up of a sequence of *nodes*, where a node contains two pieces of information: (a) the data item to be stored in the linked list, and (b) the address of the next item in the list. To make all of this work out there has to be an explicit way to say "this is the last node of the list"; remember also that the first node in the list is (normally) the only way to access any of the other nodes in the linked list.

Language feature	C++	Java
“main” program	standalone method: `int main() { .. }`	method as part of a user-defined class `public static void main() {.. }`
Output	`cout << "a string" << endl;`	`System.out.println("a string" + ’\n’);`
Simple variables	`int x= 0; char ch= ’a’;`	`int x= 0; char ch= ’a’;`
for-loops	`for(int i= 0; i< 3; i++) {x=x+i;}`	`for(int i= 0; i< 3; i++) {x=x+i;}`
strings: use	Must include a library: `#include<string>`	Library: `import java.lang.String;`

strings: declaration	`string x; x= "abc";`	`String x= new String; x= "abc";`
		OR,
		`String x; x= "abc";`

Arrays: declaration	`int myArray[3];`	`int myArray[]= new int[3];`
Arrays: declaration	`String mySarray[3];`	`String mySarray[]= new String[3];`
Arrays: use	`myArray[0]= 1;`	`myArray[0]= 1;`
Arrays: use	`mySarray[0]= "hello";`	`mySarray[0]= "hello";`
Input/output	Must include a library: `#include<iostream>`	Library: `import java.io.*;`

Conditional	`if (Cond) x= 0;`
	`else x=1;`
		`if (Cond) x= 0;`
		`else x=1;`

Switch	`switch ( Cond ) {`
	`case label1 :  x= 0; break;`
	`...`
	`default:  x= 200; break;`
	`}`
		`switch ( Cond ) {`
		`case label1 :  x= 0; break;`
		`...`
		`default:  x= 200; break;`
		`}`

Method calls	Call-by-reference or call-by-value	All method calls are call-by-value
By-value: basic	`void f(int x) { x=x+1; }`	`void f(int x) { x=x+1; }`
By-reference: basic	`void f(int &x) { x=x+1; }`	We cannot translate this.
Parameters: arrays	`void g(int[] x) { x[0]=2; }`	`void g(int[] x) { x[0]=2; }`

Parameters: objects	`void h(MyObject &x){`
	`...  "update x" ...`
	`}`
		`void h(MyObject x){`
		`...  "update x" ...`
		`}`

Classes	Data and methods	Data and methods
Declaration	`public` or `private`	A range, including `static` Consult Jia 4.4
Constructors	Default and user-defined	Default and user-defined

Constructors:
Implementation
	`myClass::myClass(params) {`
	`"Implementation"`
	`}`
		`myClass(params) {`
		`"Implementation"`
		`}`

Methods: Declaration	`void myMethod (params);`	`void myMethod (params);`

Methods: Implementation	`void myClass::myMethod(params) {`
	`"Implementation..."`
	`}`
		`void myMethod(params){`
		`"Implementation..."`
		`}`

Figure 1: A “one page” crib sheet comparing C++ and Java

What you need to know how to do for COMP2010

Familiarise yourself with the methods in the class `SingleLinks.java` which is part of this week's lecture bundle. In it you'll find the definitions for creating a list node which is the basic building block; you'll also find how these nodes are put together to make a list. Make sure you know how to do the following.

- Create an empty list;
- Add an item to the list;
- Insert an item or items into the list;
- Access any item in the list;
- Delete nodes from the list.

Looking further ahead, we'll be considering data structures implemented using linked lists and, in particular, whether the design of implementations have an impact on the performance and

Other implementations of linked lists

The linked list concept can be implemented in a number of different ways, each having its own advantages and disadvantages. The trick is knowing what those are and choosing whichever implementation most suits your purposes. Eg if an application needs to add items to the top of a list but remove them from the end of the list a different implementation than the one given in `SingleLinks` would be more appropriate.

- The implementation of `SingleLinks.java` uses an arrangement whereby each node only has a single pointer to the next node. A list then just links these nodes together, with the address of the first node accessible in the linked list. This arrangement can handle dynamic list operations, but one drawback is that accessing any particular node in the list other than the top node requires following each node along the chain until the correct node is reached.

  Such an arrangement would be suitable for implementing stacks.

- An embellishment of `SingleLinks.java` would be to add an additional field which records the location of the last node in the list. This information would be useful in the case that the last node is frequently accessed.

  Such an arrangement might be suitable for implementing queues.

- Another variation is a doubly linked list. This time it is an embellishment of the node which determines the variation, where the list is defined as linking the double nodes together.

This is advantageous for applications that need both forward and backwards traversing of the list (think of programming a text editor for example). There is however a space overhead in saving an extra link for each node.

Note that the basic concepts for linked lists are assumed. These notes are for revision.

Recursion

Recursion is used when the definition of a method actually refers to itself. The apparent circularity of this is avoided provided that you follow these rules when you define your methods.

1. Ensure there is a base case.

   The circularity is avoided provided at some point the recursive calls stop and something appropriate is returned or computed by your function. This is called the base case; when your method is called on the base case then the result should be returned without having the need to call on the method.

2. Ensure that recursive calls are made to "strictly smaller" instances of the input values.

   Again this contributes to ruling out the circularity. If you always ensure that the recursive call is made on strictly smaller (in an appropriate sense) input value so that it is "closer to" the base case, then the recursion cannot continue indefinitely since at some point the base case will be reached.

3. Ensure that the result of the recursive call is used to create the solution to the original problem.

   When implementing your method, make sure that the result of the recursive call is used to solve the problem you first started.

A typical example of recursion is given by the Fibonacci numbers, whose mathematical definition is

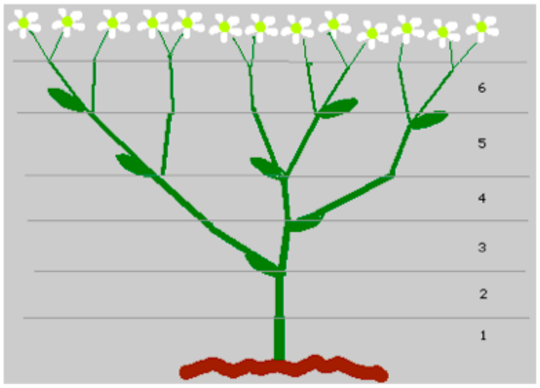
$$\mathit{fib}(n) = \mathit{fib}(n-1) + \mathit{fib}(n-2) \ ,$$

when $n >= 2$. There are actually two base cases: $\mathit{fib}(0) = \mathit{fib}(1) = 1$. Notice how each of the rules above is obeyed by this definition. Of course this can be implemented straightaway, but it isn't the best way to organise things if time or space is an issue. We'll discuss the reasons why later in the course when we look at Dynamic Programming.

Pictorial representation of Recursion and the Fibonacci series

Below [1] is an (ideal) image of a sneezewort plant which has been observed to follow this rule when putting out new growth:

> Each branch much first grow for two months before it is hardy enough to support new growth. At the two month point it grows a new branch, but then puts out a new branch every month after that. Each new branch follows the two month rule first, and ever after puts out a new branch.
>
> Assuming that growth rate is uniform, observe that the number of branches at the cross section of 1 month, 2 months etc. form a Fibonacci sequence.



Observe that the image depicted has a recursive shape: the growth after 7 months is made up of a growth of 6 months combined with a growth of 5 months.

JUnit for correctness testing of your programs

JUnit testing is a way to test key properties of the classes that you implement; it should be used in addition to other kinds of testing. You need to have the JUnit library installed in Eclipse, as well as linked to your Java project.

Like any kind of testing, how well it tests the programs is determined by the quality of the test. You will be writing your own JUnit tests to test the methods that you implement. To get the most use out of the exercise think carefully about what the method is required to achieve, and then formulate a (set of) representative tests that you think will validate its requirements. You will probably need to think up several tests for each method because for complex data structures there are a number of special cases. In the case of lists for example, methods usually behave specially in the case of empty lists, lists

[1]Taken from `www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibnat.html`

with a single item, and where the processing occurs at the beginning, middle or end of the list.

Study the examples given in `LinkedTests` to see examples of JUnit tests for the class `SLList`.

Including JUnit libraries in your project

If you are having trouble getting Eclipse to recognise JUnit, make sure you have the current Eclipse installed; this includes JUnit 4, which we'll be using in this course. You still have to "tell" Eclipse where to find those libraries, so here's what to do.

1. Inside of your test file at the top you need to have an import statement `import org.junit.Test;` This will allow you to use the various `Assert` facilities supported in the JUnit libraries.
2. Eclipse needs to know where those libraries are, so you'll need to include the JUnit Library in your project settings via the *project properties*; this is found under the *Project menu*.
3. From there, open the *Java Build Path* setting and then look at the *Libraries* tab. By default there should be an entry there for the *JRE System Library*.
4. Next you need to add JUnit via the *Add Library* button, and selecting *JUnit*. In the dialog for the JUnit library it shows the path of the library (e.g. something like .../Eclipse/plugins/org.junit...). Select *JUnit 4*.
5. Finally save the project settings and those pesky error flags (is you had them) should magically disappear!

Performance testing your programs

We have provided for you a class called `StopWatch` which you can use to test the timing of your programs. It contains methods `start` and `stop` which you use to surround the code whose performance you want to investigate and a method `getElapsedTimeSecs` which returns the difference between the most recent start and stop of the stopwatch object.

To use the class `StopWatch` you must declare an instance of the `StopWatch` class in your program. We'll show you examples in the lectures.

Tips for experimental performance testing your code

Note that testing code in this way can only be regarded as an indication of the performance. That is because your computer will be carrying out other tasks unrelated to your Java project and those will have an effect on the results from your testing. However performance testing does give you an idea about how your algorithms are performing for large datasets.

- For methods which do have to do very much this type of testing will almost always report an elapsed time of 0 or 1 seconds.
- If you believe that your method depends on the size of various input, formulate a *hypothesis* and create an experiment to test it.
- If your method relies on the input of parameters whose instances have a measurable size, and you want to test to see the dependency on the method and the size of the input parameter, use a loop to create a new instance of the method, of increasing size, call the method, using the stop watch. You can then build up a set of performance tests for ech side of input and graph your results.
- Sometimes it is difficult to obtain test data because programs run too fast. Sometimes you can use the delay feature in Java to slow down your program at points where you have decided is the most expensive operation. We'll see some examples of that next week when we look at time complexity. Be careful how you draw conclusions, remembering that this should give an indication only of the time it takes to run your program!

1 Exercises

We'll be releasing lecture bundles as zipped archive files. To install these files inside of eclipse do the following:

- First create a new project;
- Choose File → Import, and select Archive file.
- Browse to select the zipped archive file you wish to install;
- Choose Open.

The files should be ready for you to look at.

1. List the advantages and disadvantages of arrays, and explain how linked lists overcome the disadvantages. Are there any disadvantages to linked lists?
2. Use the `StopWatch` class to test experimentally the performance of the *equals* method in the class `SLList`. Can you draw any conclusions about how the lengths of the input lists affects the performance of *equals*?
3. In the class `SLList` add a method `putBefore` which takes two objects and p and q, creates a node containing the information p and inserts it in a list directly before the first occurrence of a node containing information q, or at the beginning of the list if no such item q exists.

```
public void putBefore(Object p, Object q);
// POST: inserts a node containing p immediately before the first
//  occurrence of q, or at the beginning ifq is not in the list.
```

4. In the class `SLList` add a new constructor which takes as an input an array and creates a linked list where the order in the linked list is the same as order of elements in the input array.

   ```
   public SLList(Object[]  A);
   // PRE:  A is an array
   // POST: constructs an instance of an SLList whose nodes
   //   comprise the elements A[i], and in the same order as A.
   ```

   Write a JUnit test to validate your method, and investigate its performance using the `StopWatch` class. What conclusions can you draw about the performance of your method and the length of the input array?

5. Write a recursive algorithms for to implement the following. What are the preconditions and post conditions of your methods? Write JUnit tests to check your programs.

   (a) Given a character and a string, check to see whether it occurs in the string.

   (b) Given character and a string, count the number of times it occurs in the string.

   (c) Given a character and a string, remove all of its occurrences in the string, leaving all the other characters in the same order.

   (d) Given a string, reverse it.

6. Write a recursive method that uses only addition, subtraction and comparison to multiply two non-negative integers.

   [Hint: $a \times b = a + a \times (b - 1)$, when $b > 0$.]