Algorithmics:
Data Structures and Data Types

Danny Alexander

Reading List

Russel Winder and Graham Roberts, “Developing Java Software”, J. Wiley & Sons.

Much of the material covered in the course is also covered in here. The emphasis is on software engineering, however, which we won’t be quite as concerned about in the course. It won’t do you any harm to be exposed to this though.

You should read most of part 2 of this text. Some chapters won’t be covered in the lectures, however – I won’t talk about heaps, priority queues or sorting. Also, there is some material at the end of the course on graphs, which isn’t covered in the text. The lecture notes will cover this and you can find more on them in the books listed below.

Richard Wiener and Lewis J. Pinson, “Fundamentals of OOP and Data Structures in Java”, Cambridge University Press.

Similar content to Winder and Roberts. Perhaps more concisely written.

Robert Kruse, Bruce Leung and Clouis Tondo, “Data Structures and Program Design in C”, Prentice Hall International.

The emphasis of this book is on practical working tools, but the coverage is extensive. Unfortunately the language is C, so object oriented concepts don’t get a look in.

J.H. Kingston, “Algorithms and Data Structures: design correctness, analysis”, Addison-Wesley.

Quite theoretical, with the emphasis on mathematical analysis – may be more appropriate for the complexity part of the course. Language is Modula 2. Good exercises.

Robert Sedgewick, “Algorithms”, Addison-Wesley.

Contains little discussion of data structures, but many illustrations without mathematical rigour. Language is Pascal.

D. Harel, “Algorithmics: the spirit of computing”, Addison-Wesley.

An excellent overview of theoretical Computer Science. Not particularly relevant to a data structures course, but well worth reading if you have the time.

D.E. Knuth, “The Art of Computer Programming”, Addison-Wesley.

Comprehensive reference to programming and algorithms.

Coursework

The key to understanding this type of course is hands-on experience. It is vital to get as much programming done as you can.

During the course I will set a number of unassessed exercises, which I strongly recommend you try. I will be happy to look at any work you have done and give my comments, as I'm sure will your personal tutors.

Near reading week I will hand out an assessed exercise of some complexity that you will be given the rest of term to complete.

Website

Material related to the course will appear on the course web site at:

http://www.cs.ucl.ac.uk/teaching/courseinfo/D0b3_info.htm

Aims and Objectives of the Course

Major Aims

A Toolkit of Data Structures

Your prior experience will have shown you that there is a need for data structures to hold together logically related values being manipulated by your program. Primitive types, such as

`boolean, char, int, float,`

are *atomic* data structures that contain just one value. *Aggregate* data structures, such as

`arrays, linked lists, hash tables, trees,`

contain multiple values arranged in some suitable way.

There is a range of data structures that are broadly applicable and are found to be useful in all sorts of different applications. Every programmer needs to be familiar with these and they should be regarded as "tools of the trade" for good programmers. This course is designed to equip you with the most important of these tools. It will give you more experience of some data structures that you have already met and will introduce you to a number of new ones.

Data Abstraction and Abstract Data Types

The idea of an *Abstract Data Type* (ADT) is a simple unifying way of looking at data and what operations can be performed upon it. Most modern programming languages provide facilities for defining our own *types* of *things*, which can act in certain meaningful ways that correspond to intuitive descriptions of the way a computer program works. Designing programs in this way helps to structure data and programs in a nice organised way – one that enhances *correctness*, *readability*, *robustness* and *portability* of your code.

A further aim of this course is to teach you to think in terms of ADTs and to make informed decisions about which data structures are the most appropriate for implementing particular ADTs within particular applications.

Minor Aims

Some Java

Most of the concepts we will be dealing with are not specific to any particular programming language and, in general, data structures and ADTs should be thought about in a *language-independent way*. However, it is hard to understand these concepts without seeing the actual implementation. Since Java is the main course language, all the examples are given in Java. We may come across a few features of the language that you haven't met yet, which I'll have to introduce you too, but this is *not* specifically a course on Java.

Object Oriented Design

Similarly, this isn't really a course on object oriented design, although the subject matter is strongly related so we will consider some aspects. It isn't very helpful to think about data structures or even ADTs in isolation. Inevitably, we have to look at how they interact with each other and the rest of the program. We will also need to consider reusability of our code and commonality of our data types, which are things that object oriented design deals with.

1 Abstract Data Types

1.1 Abstraction

Abstraction is the process of generalisation by suppression of irrelevant detail.

You are familiar with the concept of abstracting *computations* through the use of *functions*. Suppose I want to know how many degrees Fahrenheit are equivalent to 20 degrees centigrade. To find out, I compute `20.0*(9.0/5.0)+32.0` to obtain my answer: 68°F. Rather than having to remember this formula every time we want to perform such a conversion however, we can wrap it up in a function:

```
public static float CtoF(float x)
{
      return x*(9.0/5.0) + 32.0;
}
```

and subsequently use the abstraction `CtoF`, as in `CtoF(20.0)`, instead. The name of the function argument, x in this example, replaces specific values in an expression.

But what does it mean to abstract a type of data. How can we "generalise" or "suppress the irrelevant detail" of a data value (like 17)? The answer is less obvious than it is for the case of computation. This probably accounts for the fact that, although even the earliest assemblers (late 1940s) had some form of functional or procedural abstraction built in, data abstraction is a relatively late idea. It was first seen, in primitive form in the language Simula67, and only widely used since the advent of commercially available object-oriented languages from the mid 1980s onwards.

1.2 Data Abstraction

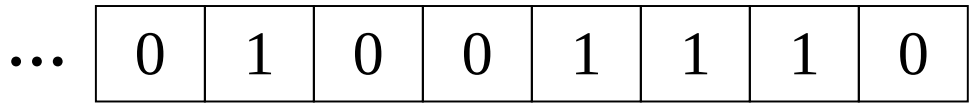
A *data type* is defined by

- The set of values it can take, and
- The set of operations that can be performed upon it.

Data abstraction is performed by defining new types, in terms of their set of possible values and the set of operations we wish to perform on that data. Languages like Java strongly encourage this kind of practice, because it forces programs to be organised in a rational way. The "irrelevant detail" that we want to suppress is the actual representation of the particular values taken by the data and the corresponding implementation of the operations. We would like to be able to apply operations to our data in a single command without having to be concerned about the mechanics of how that operation is actually performed.

1.3 Example

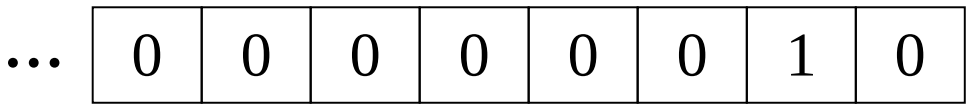
A simple way to understand data abstraction is to go right down to the machine code level. At the machine level, as you know, all data are stored as bit patterns. Consider the primitive Java type `int`. An `int` with a value of say 78, is represented, using 2's complement, by the bit pattern ending in:



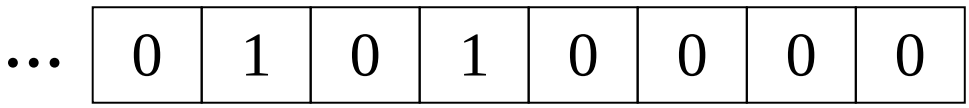
But what actually is an `int`?

It is a data type representing the mathematical concept of an integer. An integer is something that has a whole number value and upon which a number of operations, such as addition, multiplication, etc, can be performed. In order to perform such operations on an actual `int` in a Java program, the machine has to manipulate these bit patterns in an appropriate way.

Suppose we want to add another `int`, say 2, to our existing one. 2 is represented by the bit pattern ending in:



In order to add these two together some code needs to be written that takes these two bit patterns and returns a new bit pattern:



which we can interpret as an `int` of value 80.

As programmers, however, we don't want to be concerned with fiddling around with bit patterns, we are only interested in manipulating integer values. The mechanics of the operations we can perform on an `int` are hidden away – these "irrelevant details" are "suppressed". Thus our programming language provides an "abstraction" from the actual representation and implementation of the type `int` allowing us to concentrate on manipulating integer values.

Note also that there are several ways to represent integer values in bit patterns, 2's complement is the most common choice, but there are others, for example, IBM's binary coded decimal (BCD) representation. As a programmer we don't care which representation is used at the machine code level – we just want to add 2 and 2 and be assured of getting 4. The internal representation of the values as well as the implementation of the operations performed on those values are irrelevant details that are suppressed. The primitive Java type `int` is thus, in some sense, an *abstract* data type. We are concerned about *what* you can do not *how* those things are done.

Many advantages follow from thinking about programming in this way. In particular, we can have a variety of possible representations for the values of some ADT but programs that use things of that type need not be concerned about which representation is actually being used. Use of ADTs leads to a way of programming based on identifying the sorts of things a program manipulates and the operations that are performed on them.

1.4 Data Structures

A *data structure* is a container or a set of cells into which values can be placed.

It is distinct from the concept of an abstract data type. An abstract data type usually makes use of a data structure to hold the values of the data type. The data structure *contains* the *representation* of the data type. In the `int` example given above, the data structure used to hold the representation of an integer value is the set of 32-bits in the computer memory.

1.5 ADTs and Java

In Java, ADTs are defined (like everything else) in classes. Generally, the representation of the ADT, i.e., the data structure that stores its values, is hidden from view. These constitute the `private` data members of the class. The operations that are performed on the ADT are coded in the `public` methods of the class and constitute the *public interface* to the class.

In general, it is good practice to construct ADTs in a "top-down" manner, consisting of the following steps:

- Define ADT values and operations.
- Define the public interface to the class.
- Choose an appropriate data structure to hold the values.
- Implement class.

1.6 Example ADT in Java

Suppose we have a program that needs to manipulate complex numbers. An ideal opportunity to define an abstract data type!

What is a complex number?

A complex number has the form a+bi, where a and b are both real valued numbers and i is the imaginary number $\sqrt{-1}$. All of the usual numerical operations, such as addition, multiplication, etc, can be performed on complex numbers.

A complex number can be represented by a pair of floating point numbers – one for the real part of the number, a above, and one for the imaginary part, b. A pair of `double` values thus constitutes a suitable data structure in which to hold the values.

The public interface to a complex number ADT should include all the operations one is likely to want to perform on a complex number. There are lots of these. For the purposes of this illustration we will just consider two: addition and multiplication. The public interface for our class will thus have two methods, one which implements complex number addition and one which implements complex number multiplication.

Let's first define what these operations are:

Addition: (a+bi) + (c+di) = (a+c) + (b+d)i

Multiplication: (a+bi)×(c+di) = (ac-bd) + (ad+bc)i

Now we are in a position to implement the `ComplexNumber` class.

```
/**
 * <dl>
 * <dt>Purpose: Complex number class.
 * <dd>
 *
 * <dt>Description:
 * <dd>Class representing complex values allowing simple
 * numerical operations to be performed.
 * </dl>
 *
 * @author  Danny Alexander
 * @version $Date: 2000/11/26$
 *
 */
public class ComplexNumber {

    //Private data items – the data structure, which constitutes
    //the internal representation of the data type.

    private double real;
    private double imaginary;

    /**
     * Constructs ComplexNumber objects from doubles
     * representing the real and imaginary parts.
     */
    public ComplexNumber(double r, double i)
    {
        real = r;
        imaginary = i;
    }

    /**
     * Returns the real part.
     */
    public double real()
    {
        return real;
    }

    /**
     * Returns the imaginary part.
     */
    public double imaginary()
    {
        return imaginary;
    }
```

```
    /**
     * Adds complex number c and returns the result.
     */
    public ComplexNumber add(ComplexNumber c)
    {
        double newr = real + c.real();
        double newi = imaginary + c.imaginary();
        return new ComplexNumber(newr, newi);
    }

    /**
     * Multiplies by complex number c and returns the result.
     */
    public ComplexNumber multiply(ComplexNumber c)
    {
        double newr = real*c.real() - imaginary*c.imaginary();
        double newi = real*c.imaginary() + imaginary*c.real();
        return new ComplexNumber(newr, newi);
    }

}
```

Now we can do operations on complex numbers without having to worry about figuring out exactly how to do it every time:

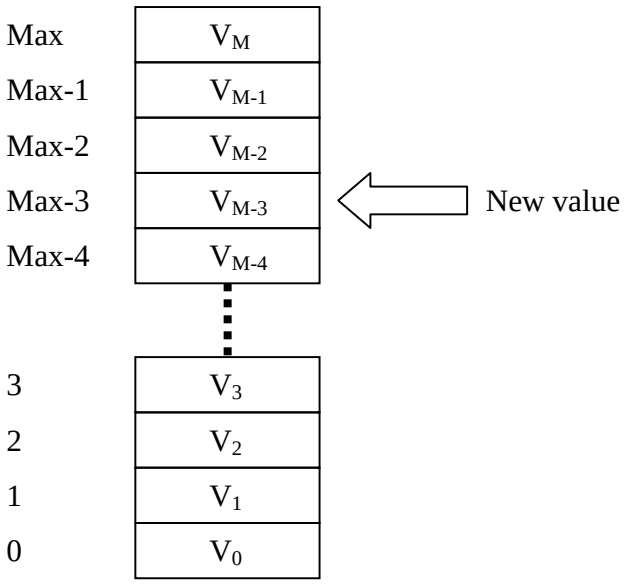
```
ComplexNumber c1 = new ComplexNumber(1.0, 2.0);
ComplexNumber c2 = new ComplexNumber(-1.0, 0.5);

ComplexNumber c1plusc2 = c1.add(c2);
ComplexNumber c1timesc2 = c1.multiply(c2);
```

Exercise: What other methods should we put in the ComplexNumber class? Try implementing the class and include methods for some other operations that can be performed on complex numbers, e.g., powers, sin, cos, etc. Include a `toString` *method and test the correctness of each operation.*

2 Some Language Facilities

In this section we are going to review some aspects of Java, and programming languages in general, which should be somewhat familiar to you. We will need to rely heavily on these things later so it's worth making sure you understand.

Any data structure we construct must be put together using the basic facilities provided by an imperative language. Most modern imperative languages (like Java) share a common semantic view of the way in which data storage is organised, based on the way in which modern computers are built. The fundamental idea is that storage consists of a linear sequence of *cells*, each of which can be thought of as a container for some storable value.



The value stored in a cell can be changed by *assignment* (using the assignment operator, =, in Java) of a new value:

Cell ← Value.

Each cell can be identified by an *address* or *index*. *Linearity* of the sequence of cells, or of a data structure in general, means that cells are identified by a single address number. A cell is similar to the concept of a *word* of storage in RAM, but the two should be distinguished. Whereas a word of storage in RAM is a fixed length chunk of bits, cells can contain any kind of object, which is considered a *storable value* in a higher level language. These objects may be very different sizes.

Cells should be thought of as boxes that we can put things in. Variable names reference different cells and distinguish them from one another so that we can keep track of what is in which box. Java is a strongly typed language, which means that cells can only contain particular things – the boxes have a shape and size and only things of corresponding shape and size will fit inside.

2.1 L-values and R-values

Rather than using the term *address*, which has hardware connotations, we refer to a cell identifier as an *l-value*. To see why this is appropriate, consider the archetypal assignment:

```
a = a + 1;
```

which means, "take the value stored in the cell associated with the variable `a`, add one to it, and store in back in the cell associated with `a`". The name `a` is used in two distinct ways: on the right hand side of the = it refers to the value stored in the cell; on the left hand side, it refers to the cell itself. Hence the name L-(left)-value. The value stored in the cell is the R-(right)-value associated with the variable `a`. Every cell has an L-value, but cells do not have R-values until they are initialised – Java won't allow us to access a cell until it has been initialised. The following won't compile in Java, but the equivalent in some languages will and when it is executed, the contents of the cell corresponding to `a` contains junk left over from last time that piece of memory was used.

```
int a;
System.out.println(a);
```

In Java, an l-value can always be an r-value as long as it has been initialised – any cell that we can put something into, we can also look at what's inside. The converse is not true, however, and there are some cells whose contents we cannot change. When a variable name is put on the right hand side of an assignment, its r-value is retrieved.

There are some *pure* r-values that can never be used as l-values. For example, we can never write:

```
3 = 3+1;
```

What about something like:

```
f(x) = 3+1;
```

where `f` is some function we have defined? No you can't do this in Java, the set of available return types of methods are such that method calls can only ever be used as r-values.

In Java, the declaration of an l-value imposes a restriction on the type of r-value that can be stored there. For example:

```
int a = 1.34;
```

will produce an error, as the r-value `1.34` is not of type `int`.

2.2 Aggregates

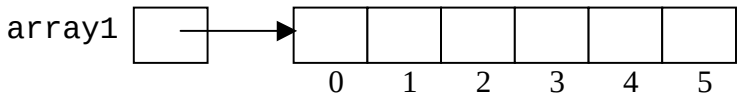
Data structures are useful when they are used to combine several data items. The most important language features used in building data structures are those that provide for *aggregation*, i.e., the mechanism for combining several data items into one more complex structure.

2.2.1 Arrays

The simplest and most common aggregate data structure is the array. An array is a contiguous, ordered, fixed-length sequence of cells, all of the same type. For example,

```
int[] array1 = new int[6];
```

This gives us a set of 6 cells, which contain objects of type `int`. `array1` is an *object reference*, which refers to the group of cells. The individual cells are referenced by *de*referencing `array1` via an index in square brackets: `array1[0], array1[1]`, etc.



These can all be l-values or r-values:

```
int n = array1[4];
array1[2] = n;
array1[3] = array1[5];
```

are all valid operations.

`array1` itself can also be used as an l-value or an r-value. We can do things like:

```
int[] array2 = array1;
```

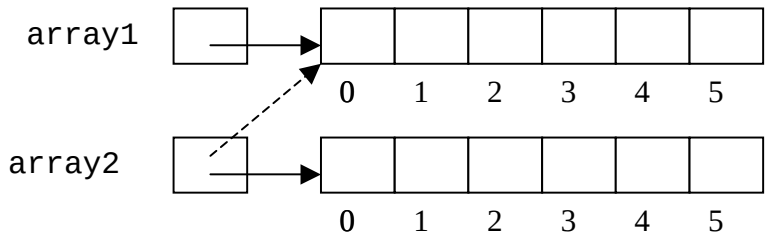
or

```
int[] array2 = new int[6];
array1 = array2;
```

However, these operations might not do quite what you might expect at first. Consider the following sequence of commands:

```
int[] array1 = new int[6];
int[] array2 = new int[6];
array1[0] = 0;
array2 = array1;
array2[0] = 10;
System.out.println(array1[0]);
```

We find that the value of `array1[0]` at this point in the program is, in fact, 10 rather than 0. The assignment `array2 = array1` doesn't copy the whole array, it copies the object reference `array1` into `array2`. So `array2` now references the exact same 6 cells that `array1` references. The 6 cells that `array2` referenced originally are lost and will be reclaimed by the Java garbage collector. This is clear if we stick to the different diagrammatic notation for object references used above, which distinguishes them from names that refer directly to particular cells:



The object references are cells that contain a reference to the object. When `array2` is set equal to `array1` only the object reference contained in the cell `array1` is copied into the `array2` cell, as shown by the dotted arrow in the diagram above.

2.2.2 Vectors

A limitation of arrays is the fact that they are *static*. Once an array has been declared its length, i.e., the number of cells in the array, is fixed. Often we may not know how many data items we want to store in a data structure and so require a more *dynamic* structure, which can expand as we add more elements.

The Java class `Vector` implements just such a data structure. See the Java documentation for `java.util.Vector` for a full description of the vector class.

`Vector` is a linear data structure – accessed by a single number through the `elementAt(int index)` method.

In fact, if you look at the actual implementation of the `Vector` class, you will discover that the underlying data structure is an array. When a vector object is initialised, the array has a certain length. If more elements are required to be stored than the length of the array, a new larger array is declared and the elements of the old array are copied into it. There are more elegant and efficient ways of generating dynamic linear data structures, which we will come across in the next section of the notes.

2.2.3 Structures

A structure is an unordered collection of cells, of arbitrary types, identified by *field names*. An example might be a structure containing employee records for a company. Several different items of data might be required, such as: name, date of birth, sex, salary and job title. We can lump all these things together for a particular employee, by encapsulating them in a class:

```
class Employee
{
    public Employee {…//Constructor…}

    public String name;
    public String dob;
    public char sex;
    public float salary;
    public String jobTitle;
}
```

For particular instances of the `Employee` class we can access all the data members (via the structure field names), as either l-values or r-values, using the dot operator:

```
Employee myEmployee = new Employee();
myEmployee.name = “Jones”;
myEmployee.salary = 20000.00f;

String someString = myEmployee.name;
float someFloat = myEmployee.salary;
```

and so on.

Although Java is perfectly happy to let us do this, it isn’t really the object oriented way. As you know, when we’re defining classes it is good practice to make the data items that make up that class `private` and only allow access to them through public interface methods. The `Employee` class should really look more like this (with appropriate comments, of course):

```
class Employee
{
    public Employee {…//Constructor…}

    public void setName(String newName)
```

```
        {
            name = newName;
        }
        public String getName()
        {
            return name;
        }
        //same for other data members…

        private String name;
        //etc
    }
```

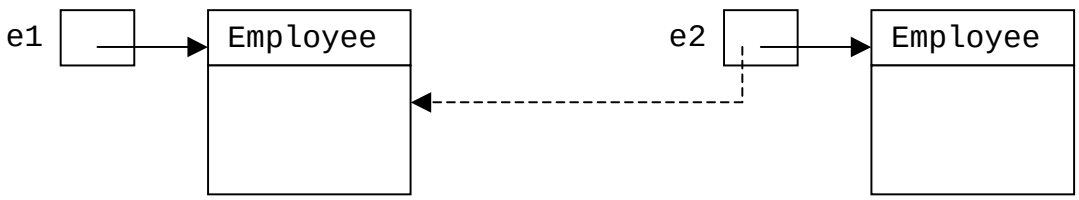
Structures *per se* shouldn't really be used in Java. Although there may often be something resembling a structure at the heart of a Java object, it should be accessed through the public interface to the class rather than directly accessing the data items.

2.2.4 Object References

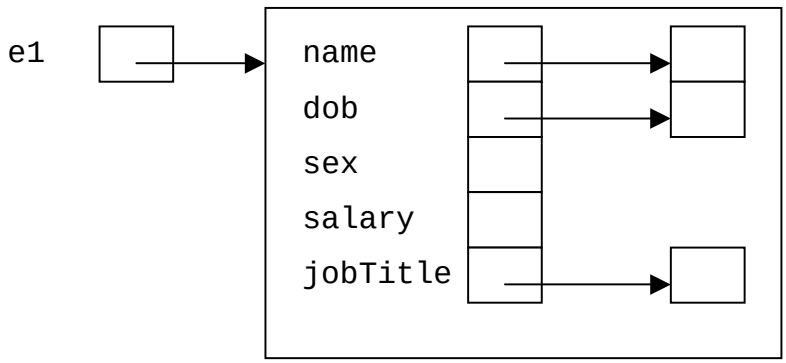
Whenever we create an instance of a class and give it a name, the r-value associated with that name is an object reference, much the same as the names of arrays we discussed above. Suppose we make some instances of the `Employee` class above:

```
Employee e1 = new Employee();
Employee e2 = new Employee();
e1.setName("Smith");
e2 = e1;
e2.setName("Jones");
System.out.println(e1.getName());
```

Exactly as in the case of arrays, we find that `Employee e1` contains the name "Jones" rather than "Smith" even though no operations have been performed directory on `e1`. As with arrays, the assignment `e2=e1` doesn't copy the whole of object `e1`, but causes `e2` to refer to the exact same object as `e1`.



This is true for all objects in Java but not primitive types. It often helps to understand programs, particularly those that use complex data structures, by drawing data diagrams. In order to ensure that we remember how object references behave, we can draw them in the same way we did in the section on Arrays. An object reference refers to a cell that contains a reference to the object itself. A diagram depicting an instance of our `Employee` class in full will look something like this:



We have object references inside the `Employee` object for the non-primitive data types `Date` and `String`.

2.2.5 Data Types and Data Structures Revisited

A data type is a set of values with a collection of operations. A data structure is a collection of cells, somehow related, that can hold data values. Thus, a data structure may serve as the value-holding part of lots of different ADTs – it depends what operations we define on the given structures and there is quite a lot of scope for wrapping them up as data types in different ways. Often there is a conflict between providing a small, compact set of operations required to get a particular job done, and trying to provide generality of interface to give a widely useful data type that can go into a library and be reused.

It is a practical fact that most programmers re-implement the same ADT over and over, with slight variations. Object oriented methodology and languages are supposed to make it easy to reuse code and avoid this. It is debatable whether or nor they have succeeded, but things seem to be improving.

We will illustrate these points in the section 3 by showing how one data structure, the array, can be used to implement a variety of ADTs. Then we will show how a given ADT could be implemented by a variety of different data structures.

2.3 Exceptions and Exception Handling

Exceptions provide a mechanism for identifying and handling unforeseen errors in a Java program. In general, Exceptions are used to flag errors when they occur inside a class due to unexpected input or method parameter values. They avoid the need for the implementer of the class to handle these possibilities explicitly and, rather, allow (in fact, *force*) the user of the class to specify some alternative course of action when the error occurs. Full discussions on the use of Exceptions can be found in Winder and Roberts, or Wiener and Pinson, or in the on-line Java documentation. Here I will just give an example with explanation, there are many more to come.

Suppose we wish to add another method to the `ComplexNumber` class that computes the reciprocal, i.e., 1.0/(a+bi). The reciprocal is simple to compute using the formula: $1.0/(a+bi) = (a/(a^2+b^2)) - (b/(a^2+b^2))i$. But what if a = b = 0? What should we do?

- Return zero?
- Return some other complex value?
- Print out an error?
- Just allow Java to pick up this error in its own way so that any program using our `ComplexNumber` class will fail when this occurs.

A better idea is to have the `reciprocal` method throw an `Exception` to indicate that this has happened and which can be picked up and dealt with appropriately. Here is a `reciprocal` method that uses `Exception`s in this way:

```
public ComplexNumber reciprocal() throws ComplexNumberException
{
    double denominator = real*real + imaginary*imaginary;

    if(denominator == 0) {
        throw new ComplexNumberException("Cannot
                                          reciprocate zero");
    }

    double newr = real/denominator;
    double newi = -imaginary/denominator;

    return new ComplexNumber(newr, newi);
}
```

In the reciprocal method, we detect when the error has occurred and "throw" a `ComplexNumberException` using the keyword `throw` applied to a new object of type `ComplexNumberException`. Notice that we need to declare in the definition of the method that the method is capable of throwing a `ComplexNumberException`. A method can throw many different types of `Exception` and each must be listed in the method definition. If there is more than one, they are separated by commas after the throws keyword: `myMethod() throws Exception1, Exception2, …, ExceptionN`.

`ComplexNumberException` is a new type of object that we need to define before we can use it in our `ComplexNumber` class. Java provides a class `Exception` that can be extended (using inheritance) in order to create `Exception`s for particular situations. This can be done in a very basic way, which essentially just changes the name of the `Exception` so that it's clear to the user what has happened when the `Exception` is thrown, or, if necessary, specialized behaviour can be included. Here we'll do a basic extension of the `Exception` class:

```
class ComplexNumberException extends Exception()
{
    ComplexNumberException() {
        super();
    }
    ComplexNumberException(String s) {
        super(s);
```

```
        }
    }
```

Like all Java classes, the code defining the `ComplexNumberException` class needs to be stored in a separate file called ComplexNumberException.java.

Now that we have declared that our new method can throw this `Exception`, we need to take care of this possibility when making calls to this method. Suppose we try to make a call directly to the `reciprocal` method inside the body of a main method:

```
ComplexNumber c = new ComplexNumber(1.0, 2.0);
ComplexNumber cr = c.reciprocal();
```

The following compiler error appears:

```
Exception ComplexNumberException must be caught, or it must be
decalared in the throws clause of this method.
    ComplexNumber cr = c.reciprocal();
```

The fact that the `reciprocal` method declares that it throws an `Exception`, causes the compiler to insist on being told what to do if the `Exception` should be thrown. The programmer must provide an alternative course of action through the use of a `try`…`catch` construct:

```
ComplexNumber c = new ComplexNumber(1.0, 2.0);
ComplexNumber cr;
try {
    cr = c.reciprocal();
} catch(Exception e) {
    System.out.println(e);
}
```

Java tries to execute the code inside the `try` statement is executed. If an `Exception` is thrown while that code is being executed, the program jumps to the `catch` statement and executes the code therein instead. In this example, the `Exception` is printed in the `catch` statement, so the type of the `Exception`, together with its message appear on the command line:

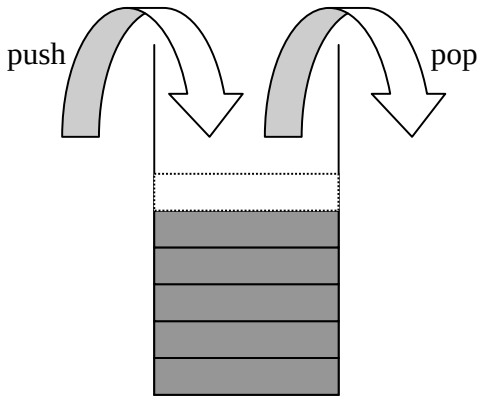
```
Exception in thread "main" ComplexNumberException: Cannot reciprocate
zero…
```

3 Linear Data Structures and ADTs

Much of the material covered in this section consists of program listings. This code is mostly intended to show you how the various data structures and ADTs work; I don't claim that it is necessarily well engineered, robust and adaptable. For a detailed discussion of these issues, read the corresponding chapters in Part 2 of the Winder and Roberts book, or Wiener and Pinson.

3.1 Stacks

The first ADT we will investigate is the *stack*, which you may have met before. A stack is a simple, but extremely useful ADT, which consists of a linear sequence of items to which you can add new items and remove old ones. All insertions and deletions occur at one end – the top of the stack. The only item in the stack that can be examined is the one at the top of the stack. A stack is also known as a LIFO, which stands for Last In First Out, since the item at the top of the stack must be the one most recently added. A stack is like a pile of books on a desk or plates on a shelf. Another common analogy is that of a spring loaded plate holder in a cafeteria; this analogy has given rise to the *push* and *pop* terminology, which refer to the operations of adding an item to the top of the stack and taking the item from the top of the stack, respectively.



Stacks are vitally important in operating systems for storing temporary data during function calls. For a simple illustration of the utility of stacks, look up the man pages for the UNIX commands pushd and popd.

Let's follow the "top-down" approach to ADT construction and write down a definition of a stack ADT.

- The set of values that can be stored in a stack is the set of linear sequences of objects.
- The minimal set of operations we wish to perform on a stack consists of

  (a) to add a new item to the stack (push) and

  (b) to take the top item from the stack (pop).

Now we are in a position to define the public interface to a class implementing our stack ADT. We will include some additional methods beyond the basic minimum, which make the implementation more elegant and efficient. The public interface will consist of the following operational methods:

```
/**
 * Adds a new item to the top of the stack.
 */
public void push(Object o)

/**
 * Removes the item from top of the stack and returns it.
 */
public Object pop()

/**
 * Returns the item at the top of the stack but leaves it there.
 */
//This operation could be performed using pop and push, but
//it is more efficient to have a separate method.
public Object top()

/**
 * Tests whether there are any items in the stack.
 */
public boolean empty()

/**
 * Removes all the items from the stack.
 */
//This could be done using pop, but again, efficiency is
//increased by having a separate method.
public void clear()
```

We also need to choose an appropriate data structure to hold the values comprising our ADT. The stack consists of a linear sequence of values and so the obvious choice is to use an array. This is what we will do first of all. There are obvious drawbacks of this approach (*like what?*) and we will see some alternative implementations of the stack ADT that use different data structures later.

For our first implementation, our data structure will be an array of type `Object[]`. We need to keep track of the number of elements contained in the stack so that we know what index of the array corresponds to the head of the stack. So we store an additional private data member, `stackSize`, which is maintained as the (integer) number of elements in the stack. We will implement two constructors, one that allows the user to specify the size of the array used in the `StackArray`and a second one that assigns a default size to the array.

Now we are ready to implement the `StackArray` class. Here is a basic outline of the class:

```
Public class StackArray
{
    //The data structure is stored in the private data members
```

```
    //of the class.
    private Object[] objectArray;
    private int stackSize = 0;

    /**
     * Constructs a stack with a maximum capacity of 50 items.
     */
    public StackArray() {
        objectArray = new Object[50];
        stackSize = 0;
    }

    /**
     * Constructs a stack with specified maximum capacity.
     */
    public StackArray(int size) {
        objectArray = new Object[size];
        stackSize = 0;
    }

    //Public interface methods
    public void push(Object o) {//Method body}
    public Object pop() {//Method body}
    public Object top() {//Method body}
    public boolean empty() {//Method body}
    public void clear() {//Method body}
}
```

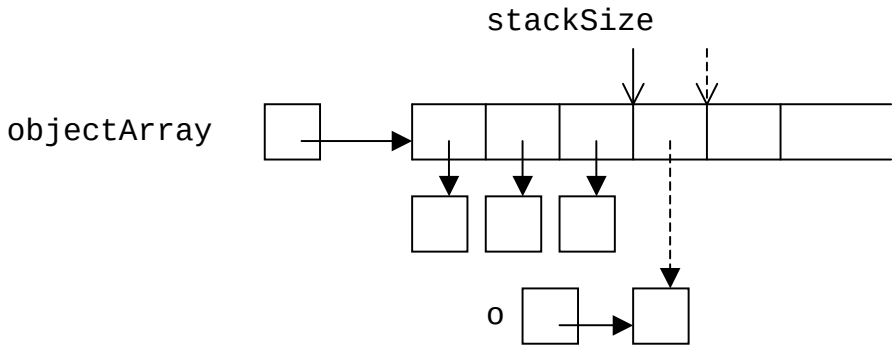
Now we'll look at the implementations of the public member functions one by one. The stack is implemented in such a way that `stackSize` always indexes the next empty element of the array, as well as keeping track of the number of elements contained in the stack. The top of the stack is at the higher indices of the array and the bottom of the stack is at array index zero – `objectArray[0]`. The top of the stack, assuming the stack is not empty, is indexed by `objectArray[stackSize-1]`.

Exercise: Why this way round? Could we implement the stack in such a way that the top of the stack is at the front of the array (`myArray[0]`*)? What is the problem with this approach?*

Here is the public interface method `push`:

```
public void push(Object o) throws StackArrayException {
    if(stackSize < objectArray.length) {
        objectArray[stackSize] = o;
        stackSize += 1;
    } else {
        throw new StackArrayException("Stack overflow");
    }
```

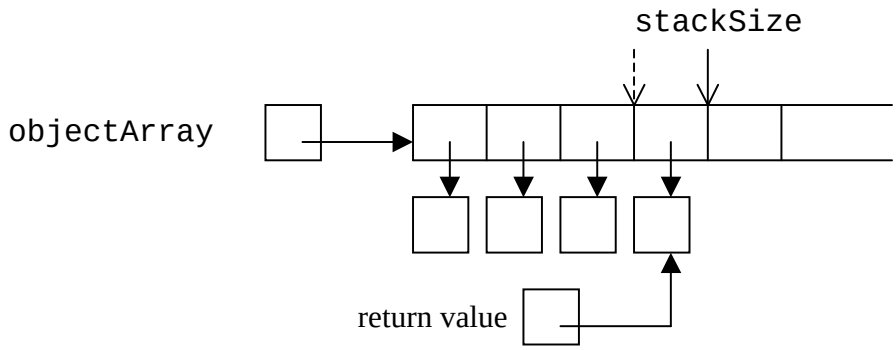
```
}
```



We check to see if the array holding the stack is full. If not the new object is placed in the next available cell of the array, which is the new head of the stack, and `stackSize` is incremented. If the stack is already full, an exception is thrown.

Now the public interface method `pop`:

```
public Object pop() throws StackArrayException {
    if(stackSize != 0) {
        stackSize -= 1;
        return objectArray[stackSize];
    } else {
        throw new StackArrayException("Stack underflow");
    }
}
```



In this method, we need to check that the stack is not empty. If it is not, we return the current head of the stack and decrement `stackSize`. Notice that we do not need to explicitly remove the object at the top of the stack from the array. When a new object is added to the stack, it will be replaced – draw another data diagram to help you understand this.

The public interface method `top` is similar to `pop`, the only difference is that `stackSize` is not decremented as we do not want to remove the top element of the stack.

```
public Object top() throws StackArrayException {
```

```
        if(stackSize != 0) {
            return objectArray[stackSize-1];
        } else {
            throw new StackArrayException("Stack underflow");
        }
    }
```

The final two methods, `empty` and `clear`, are very simple. Notice again that we do not need to explicitly remove elements from the stack in the `clear` method.

```
    public boolean empty() {
        return (stackSize == 0);
    }

    public void clear() {
        stackSize = 0;
    }
```

There is one thing left to implement, which is the `StackArrayException` class that is thrown by several of the public methods above when the stack overflows or underflows. `StackArrayException` is inherited directly from `Exception`, as follows:

```
    public class StackArrayException extends Exception {
        StackArrayException() {
            super();
        }
        StackArrayException(String s) {
            super(s);
        }
    }
```

That completes the implementation of `StackArray`.

Here is a short program, which illustrates how we can create an instance of and use our new abstract data type:

```
    public class StackArrayTest {
        public static void main(String[] args) {

            StackArray myStack = new StackArray(25);

            //Add some elements to the stack.
            for(int i=0; i<5; i++) {
                try {
                    myStack.push(new Integer(10*i));
                } catch (Exception e) {
                    System.out.println(e);
                }
            }
```

```
            //Remove all the elements of the stack and print them.
            while(!myStack.empty()) {
                try {
                    System.out.println(myStack.pop());
                } catch (Exception e) {
                    System.out.println(e);
                }
            }
        }
    }
```

Exercise: what is the output of the program above?

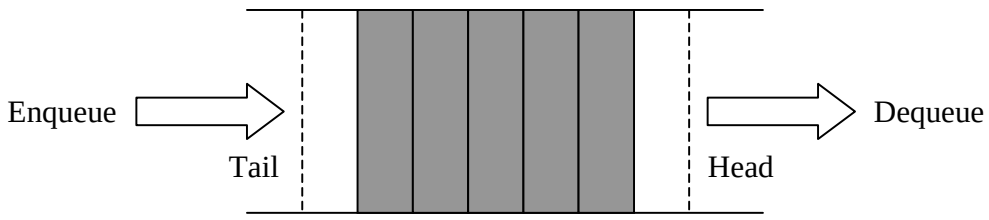
Exercise: extend the above program to test the other methods, `top` *and* `clear`. *Also test the overflow and underflow handling.*

Exercise: A serious limitation of the `StackArray` *class above is that it has fixed size. We could overcome this problem by using a* `Vector` *in place of the array* `objectArray`. *Implement and test a class* `StackVector` *with the same public interface as* `StackArray`, *but using a* `Vector` *as the underlying aggregate data structure.*

3.2 Queues

Another useful ADT is the *queue*, which, like the stack, also consists of a linear sequence of elements. Unlike the stack, however, which is a LIFO sequence, a queue is a FIFO, First In First Out, sequence. The idea is that elements are removed from the queue in exactly the same order that they are put into it. The obvious analogy is that of, unsurprisingly, a queue, for example at the supermarket check out, where we join the queue at the *tail* and leave at the *head*. The elements of the queue (shoppers) leave the queue in the same order that they join it. The general term queue, in a programming context, encompasses many different variations, e.g., priority queues, multilevel queues, but all of these in some way adhere to the general FIFO philosophy.

The operation of making insertions at the tail of a queue is known as *enqueueing*. The operation of taking the element from the head of the queue is known as *dequeueing.*



Queues are vitally important in many computer systems, both in applications – such as airport simulation (see Kruse) – and in systems – such as a time-sharing Operating System.

3.2.1 Queue ADT

The queue ADT is defined as follows:

- The set of values that can be stored in a queue, as for a stack, is the set of linear sequences of objects.
- The minimal set of operations we wish to perform on a queue consists of

  (a) to add a new item to the queue (enqueue), and

  (b) to take the item from the head of the queue (dequeue).

3.2.2 Public Interface

As with the stack ADT, we will include some additional methods in the public interface for our queue class, beyond the basic minimum, to improve the elegance and efficiency of the implementation. The public interface will consist of the following methods:

```
/**
 * Adds a new item to the tail of the queue.
 */
public void enQueue(Object o)

/**
 * Returns and removes the item at the head of the queue.
 */
public Object deQueue()

/**
 * Returns the item at the head of the queue, without removal.
 */
public Object head()

/**
 * Tests whether the queue contains any items.
 */
public boolean empty()

/**
 * Empties the queue.
 */
public void clear()
```

3.2.3 Data Structure

Once again we will use a fixed length array to implement the first version of the queue ADT. Clearly this approach has the same limitations as it did for `StackArray` and we will investigate some alternative approaches later in this section.

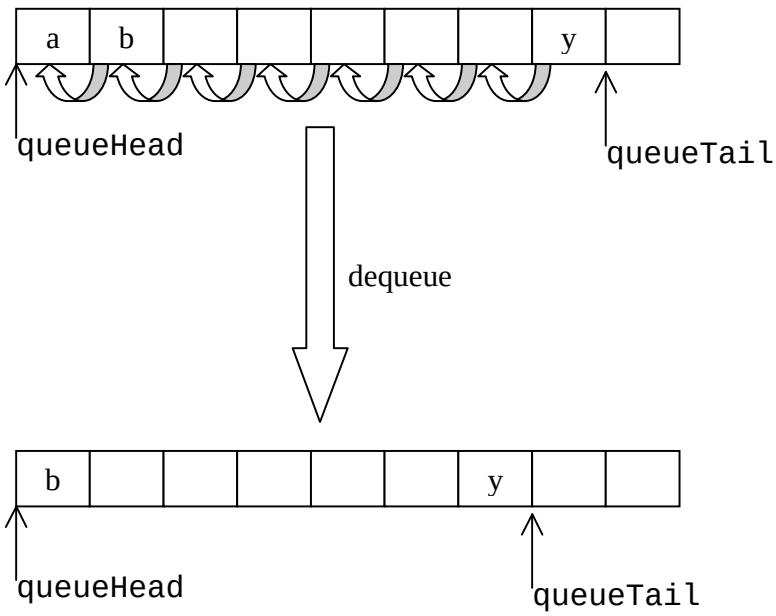
It turns out that to implement the queue in the most effective way, we will require three additional private data members, as well as the array of `Object`s as usual:

```
//The array containing the queue.
private Object[] objectArray;

//The number of items currently stored in the queue.
private int queueSize;

//The array index corresponding to the head of the queue.
private int queueHead;

//The array index corresponding to the tail of the queue.
private int queueTail;
```

3.2.4 Representation

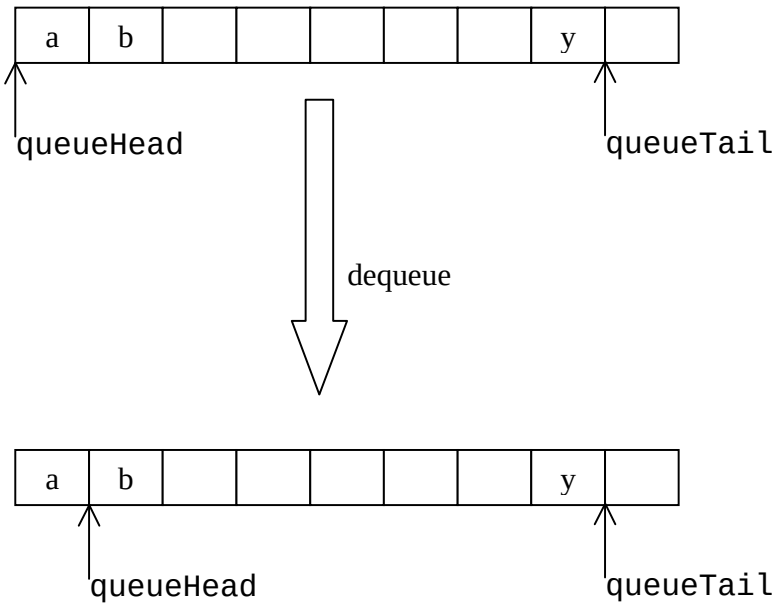
We need to think quite carefully about how we will represent our queue within a fixed length array, i.e., which parts of the array correspond to the head and tail of the queue.

Suppose, first of all, that we use an approach similar to the one we adopted for the stack. So the array is filled from element 0, contiguously, up to element `queueSize - 1`. As with the stack, in order to avoid having to shift all the elements of the array every time we add a new item to the queue, we store an index to the next available cell in the array and always add elements at the far end. The problem with this approach for the queue, is that we are now removing items (dequeueing) from the front of the array – `objectArray[0]`. In order to get the new head of the queue positioned correctly after a dequeue operation, the whole array has to be shifted down one element:

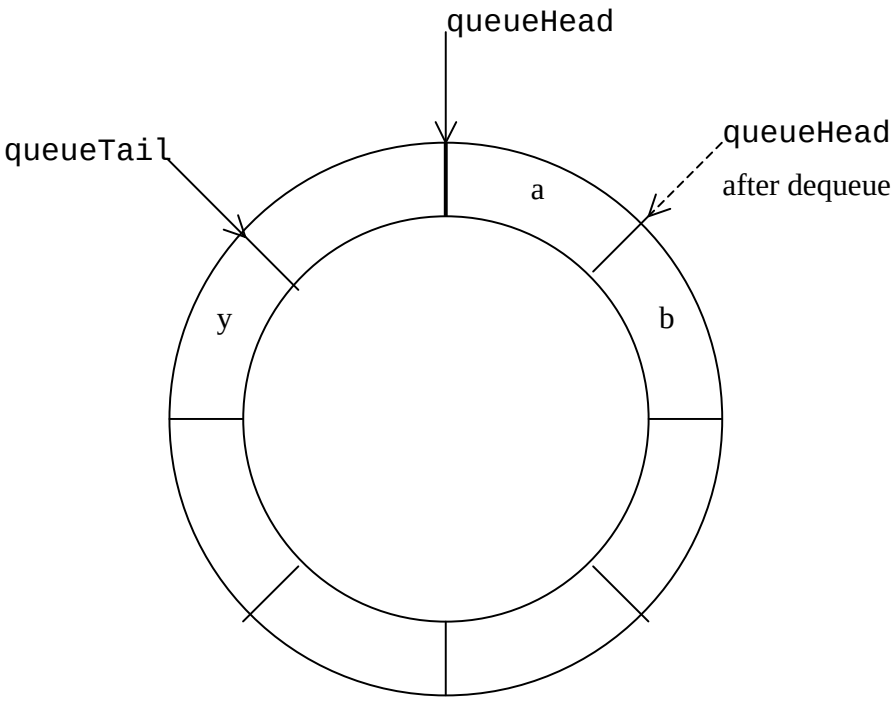


This strategy is analogous to a physical queue, in which everyone shuffles forward after a "dequeue", but it is clearly expensive. We would like to find an approach that doesn't involve this *array shifting*.

Another option is to increment the index `queueHead` whenever an item is removed from the queue:



`deQueue` is much more efficient using this strategy, but we now have the problem that the queue gradually moves down the array. The front of the array is discarded with every `deQueue` operation and so the size of the array actually used to contain the queue data is decreased by one every time. The queue gradually moves up the array and will eventually reach the end.



In fact it is quite straightforward to overcome this problem, by allowing the tail of the queue to wrap around to the beginning of the array, once it reaches the end. In effect we create a *circular array*, which `queueHead` and `queueTail` "chase" each other around, as shown above.

The queue occupies different portions of the array at different times, but we never run out of space unless the array is really full. The wrap around effect is achieved through the use of the *modulo* (or *remainder*) operator, %. The following expression is used to increment the tail index during an `enQueue` operation:

```
queueTail = (queueTail + 1)%objectArray.length;
```

After indexing the last element of the array (`queueTail == objectArray.length – 1`), this increment expression sets `queueTail` to zero.

Here is a complete listing of the class `QueueArray` implemented using a circular array. `queueHead` always indexes the cell of the array that contains the head of the queue, whilst `queueTail` always indexes the next free cell of the array, i.e., the one *after* the cell containing the item at the tail of the queue. Notice the use of the additional private data member, `queueSize`, which keeps track of the number of elements in the queue. We could use comparisons of `queueHead` and `queueTail` to do the job of `queueSize`, but the implementation is neater if we include this extra variable.

```
public class QueueArrayException extends Exception {
    QueueArrayException() {
        super();
    }
    QueueArrayException(String s) {
        super(s);
    }
}

public class QueueArray {

    //Private data members holding the values in the queue and
    //encoding the representation.
    private Object[] objectArray;
    private int queueTail;
    private int queueHead;
    private int queueSize;

    public QueueArray() {
        objectArray = new Object[50];
        queueHead = 0;
        queueTail = 0;
        queueSize = 0;
    }

    public QueueArray(int size) {
        objectArray = new Object[size];
        queueHead = 0;
        queueTail = 0;
        queueSize = 0;
```

```
    }

    public void enQueue(Object o) throws QueueArrayException {
        if(queueSize < objectArray.length) {
            objectArray[queueTail] = o;
            queueTail = (queueTail + 1) % objectArray.length;
            queueSize += 1;
        } else {
            throw new QueueArrayException("Queue Overflow");
        }
    }

    public Object deQueue() throws QueueArrayException {
        if(queueSize != 0) {
            queueSize -= 1;
            int oldQueueHead = queueHead;
            queueHead = (queueHead + 1) % objectArray.length;
            return objectArray[oldQueueHead];
        } else {
            throw new QueueArrayException("Queue Underflow");
        }
    }

    public Object head() throws QueueArrayException {
        if(queueSize != 0) {
            return objectArray[queueHead];
        } else {
            throw new QueueArrayException("Queue Underflow");
        }
    }

    public boolean empty() {
         return (queueSize == 0);
    }

    public void clear() {
        queueTail = 0;
        queueHead = 0;
        queueSize = 0;
    }
}
```

Exercise: As with the class `StackArray`*, the circular array implementation of the queue ADT has the drawback that it has fixed limited size, since it uses an array as its underlying aggregate data structure. In the case of the stack, it is simple to overcome this drawback by using a* `Vector` *in place of the array. How can we do the same thing for the queue ADT?*

3.3 Generalised Linear Sequences

The ADT *sequence* consists of linear sequences of objects, like an array, but, unlike an array, a sequence can grow and shrink by the insertion and removal of elements at arbitrary positions. The minimal set of operations on a sequence are

- Insert an item at a specified position in the sequence.
- Delete the item at a specified position.
- Return the item at a specified position.

The sequence is a generalisation of both the stack and queue ADTs and could be used in such a way as to emulate either – we will return to this later and investigate the use of inheritance to relate these three ADTs.

We can build a sequence ADT using an array as the underlying aggregate data structure, as we did above for the stack and queue ADTs. It turns out that the circular array, used to implement `QueueArray`, provides the neatest representation for a sequence. The code for class `SequenceArray` will appear on the D0b3 web site soon, but we won't go through it in detail here. A few extra methods are included in the public interface to the class; here is the complete listing:

```
/**
 * Constructs a sequence of fixed capacity.
 */
public SequenceArray()

/**
 * Constructs a sequence of specified capacity.
 */
public SequenceArray(int size)

/**
 * Inserts a new item at a specified position.
 */
public void insert(Object o, int index)

/**
 * Inserts an item at the beginning of the sequence
 */
public void insertFirst(Object o)

/**
 * Inserts an item at the end of the sequence
 */
public void insertLast(Object o)

/**
 * Returns the item at the indexed position.
 */
```

```
public Object element(int index)

/**
 * Returns the item at the beginning of the sequence.
 */
public Object first()

/**
 * Returns the item at the end of the sequence.
 */
public Object last()

/**
 * Removes the item at the indexed position.
 */
public void delete(int index)

/**
 * Removes the item at the beginning of the sequence.
 */
public void deleteFirst()

/**
 * Removes the item at the end of the sequence.
 */
public void deleteLast()

/**
 * Return the number of elements in the sequence.
 */
public int size()

/**
 * Empties the sequence.
 */
public void clear()
```

The insertion and deletion member functions, `insert` and `delete`, need to use array shifting to create space for the new element or use up the space vacated by a deleted element, since the sequence is stored contiguously in the array. Here is the code for the `delete` method:

```
public void delete(int index) throws SequenceArrayException {
    if(index >= 0 && sequenceSize > index) {

        //Move all the elements that are further down the
        //sequence back one place in the array.
        for(int i=index+1; i<sequenceSize; i++) {
            int oldIndex = (sequenceHead+i)%objectArray.length;
```

```
            int newIndex = (objectArray.length+oldIndex-
                                        1)%objectArray.length;
            objectArray[newIndex] = objectArray[oldIndex];
        }

        //Update size and tail index.
        sequenceSize -= 1;
        sequenceTail = (objectArray.length+sequenceTail-
                                        1)%objectArray.length;
    }
    else {
        throw new SequenceArrayException("Indexed element is out
                                                of range");
    }
}
```

Array shifting is a lengthy process. The additional methods for insertion and deletion at the beginning and end of the sequence are included, because, using the circular array, these methods can be implemented much more efficiently, by simply incrementing or decrementing the indexes to the beginning and end of the sequence.

Notice that we have very nearly arrived at an ADT with equivalent functionality to the `Vector` class in Java. The only difference is that we have no mechanism for dealing with overflow. As you know, `Vector` also uses an array as its underlying aggregate data structure. When the array becomes full, a new, larger array is allocated and the elements are copied into it from the old array.

Exercise: Try implementing the class `SequenceArray` *without looking at my code, you can check it against that afterwards. If you find that straightforward, adapt your class so that it copes with overflow in a similar way to* `Vector`.

Also think about how we could use the `SequenceArray` *class to implement stacks or queues. Is inheritance the answer?*

3.4 Linked List Implementations

The use of an array in the previous implementations of stack, queue and sequence ADTs is straightforward but has fundamental problems:

- The array has fixed size and so can overflow.
- Some insertion and deletion operations require array shifting and so can be inefficient.

We can overcome the first limitation in two obvious ways:

- Within particular applications of our ADTs, ensure that the allocated array is large enough so that overflow will not occur.
- Include some mechanism in the class for reallocating larger arrays when the original one becomes full.

The first method is generally wasteful in terms of memory, since in most cases large portions of the array will remain unused. The second method can be costly in terms of computational operations.

The second drawback, array shifting, we are stuck with, although we can carefully choose the representation of a particular ADT to optimise insertion and deletion operations in particular applications. For example, the use of the circular array to implement the queue ADT.

There are some advantages of the array implementations of linear ADTs:

- Retrieval is fast – the operation of indexing an array to find a particular item is very efficient.
- Implementation is simple and intuitive.

However, a more elegant way to implement linear ADTs is through the use of *linked lists*. The advantages of linked lists are

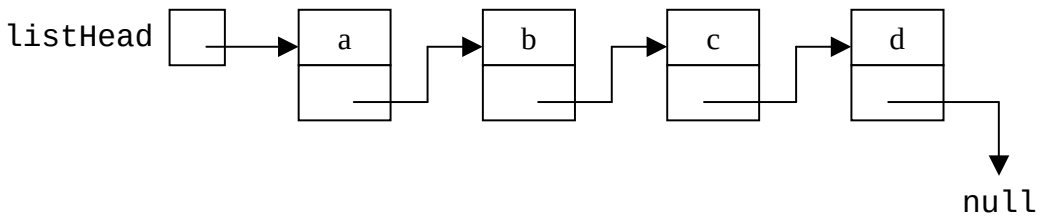
- They are truly dynamic data structures that grow and shrink with the number of items stored. There is no "wasted" memory and ADTs will not overflow until the machine runs out of memory.
- No array shifting is required for insertion and deletion operations at any point in the list.

There are some minor drawbacks associated with the use of linked lists in place of arrays:

- There is some additional memory required to store each item.
- Increased code complexity (possibly).
- Slightly reduced efficiency for some operations, in particular, indexing.

3.4.1 Linked Lists

A linked list is a linear data structure, like an array or a vector, and so requires just a single index to identify individual items stored within. The idea of a linked list is that, with every item stored in the list, a link to the next item in the list is also stored:



The list is built up of *nodes*, each of which contains both a data item – the object we want to store – and a link to the next node in the list. To find individual items in the list, we follow the links starting at the head of the list until we reach the node with the index we are interested in. Common analogies for linked lists are treasure hunts, where we start with a clue, which leads us to a location where we find another clue and so on until eventually we find what we are looking for; or trying to trace a book that has been lent from friend to friend.

To implement a linked list, we encapsulate the idea of a node in a class. Class `Node` will contain two data items: a piece of data, which will be an object reference of type `Object`, and a link to the next node of the list, which is an object reference of type `Node`.

3.4.2 Stack as a Linked List

The implementation of the stack using a linked list is particularly simple, so we will look at this example first. A change in the underlying representation or implementation of an ADT should not affect its appearance from outside the class and the definition of the stack ADT is exactly as it was before. We will have the same set of methods in the public interface to our new class, `StackList`, as we did previously for `StackArray`. Here is an outline of the class implementation:

```
public class StackList {

    //Linked list nodes encapsulated in member class.
    protected class Node {
        public Node(Object o, Node n) {
            datum = o;
            next = n;
        }

        protected Object datum;
        protected Node next;
    }

    //The only private data member required is a reference to
    //the head of the linked list, i.e. a reference to the
    //first node.
    private Node listHead;

    //Constructor
    public StackList() {
        listHead = null;
    }

    //Public interface methods…
    //:
    //:
}
```

The class `Node` is a *member class* of the *top-level class* `StackList`. The idea of a member class is to encapsulate an object that is only used within one top-level class. In the case of a linked list, for example, it does note make sense to create instances of a linked list node outside the scope of the linked list itself. The `Node` class is part of the representation of the list and not something we want other parts of the program to know about or use. Thus `Node` is hidden within the implementation of the class.

A member class has access to all the variables of the enclosing class and the enclosing class has access to all the `public` and `protected` members of the member class (but not the data member declared `private`). In the example above, we can access the protected data members of member class `Node` from the enclosing class `StackList` and we could, although we don't here, access all the data members of the enclosing class `StackList`, such as `listHead`, from within any instance of the

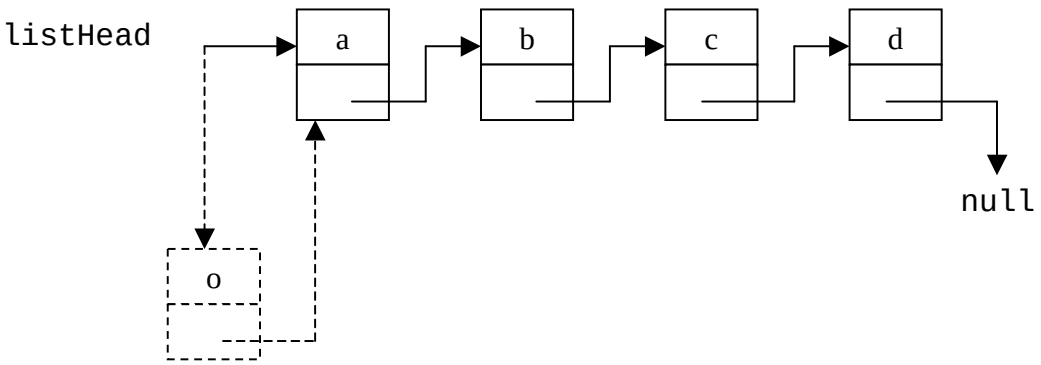
class `Node`. Member classes are described in more detail in chapter 7 and 30 of the Winder and Roberts book or in the online Java documentation.

Every item in the stack is stored in a separate instance of the `Node` class. When an instance of `StackList` is first constructed, there are no items in the stack and so `listHead` is set to `null`, which tells the Java compiler that the object reference, `listHead`, is unitialised (it doesn't reference anything). New items are added at the head of the list and so `listHead` always refers directly to the top of the stack.

We will see how the member class Node is used in the implementation of the public interface methods, which we'll go through now one by one:

```
public void push(Object o) {
    listHead = new Node(o, listHead);
}
```

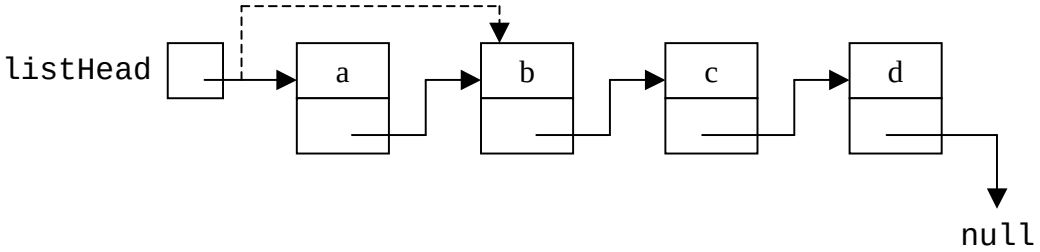
A new `Node` is added at the head of the list, which contains the new item for the stack, `Object o`, and a reference to the `Node` that was previously at the head of the list, which is currently referenced by `listHead`:



The new `pop` method is also quite simple, but we still have to handle the possibility of stack underflow:

```
public Object pop() throws StackListException {
    if(listHead != null) {
        Object top = listHead.datum;
        listHead = listHead.next;
        return top;
    } else {
        throw new StackListException("Stack Underflow");
    }
}
```

A reference to the object contained in the first `Node` of the list is returned and `listHead` is changed to reference the `Node` that is currently second, i.e., the one that is referenced by the `Node` currently first:



Notice that we can use the keyword `null`, which represents an "empty" object reference, in boolean comparisons to test whether `listHead` has been initialised, i.e., whether it references anything.

`top` is similar to `pop`, except that the first element in the list is not unlinked:

```
public Object top() throws StackListException {
    if(listHead != null) {
        return listHead.datum;
    } else {
        throw new StackListException("Stack Underflow");
    }
}
```

The remaining methods, `empty` and `clear`, are straightforward as before:

```
public boolean empty() {
    return (listHead == null;
}

public void clear() {
    listHead = null;
}
```

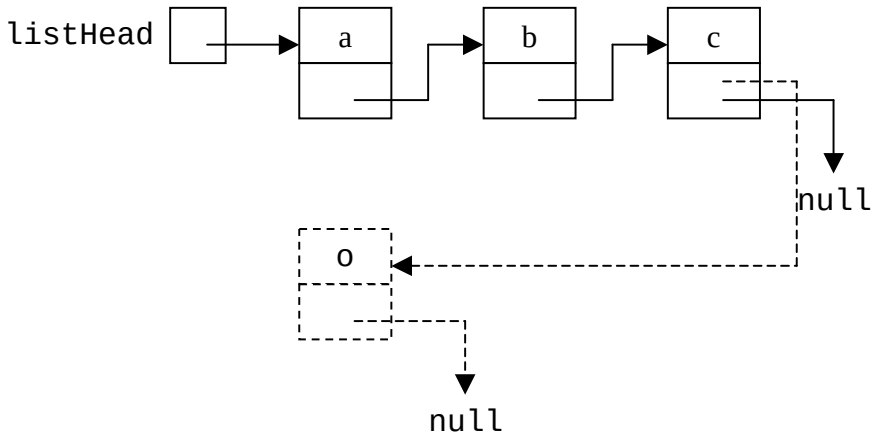
In method `clear`, we do not need to explicitly remove the `Node` objects in the list. Once the object reference `listHead` has been removed, there is no reference to the first `Node` in the list, which can thus be reclaimed by the garbage collector. This removes the only remaining object reference to the second `Node` in the list, which can thus be reclaimed, and so on. To speed up the garbage collection process, we could chain down the linked list, setting each link to `null`.

This implementation of the stack ADT is extremely elegant and in fact the code is probably even simpler than the array implementation, although the subtleties of the implementation are perhaps a little harder to understand.

3.4.3 Queue as a Linked List

We can use the idea of a linked list in a similar way to implement the queue ADT outlined in section 3.2.1. The public interface to the new class, `QueueList`, is unchanged from the `QueueArray` class public interface outlined in section 3.2.2.

The first approach we will try is to use exactly the same data structure that we used in the implementation of `StackList` in the previous section. Suppose the head of the queue is stored in the first node of our linked list and the tail is in the last. The `deQueue` operation is then exactly the same as the `pop` operation in the `StackList` class. However, `enQueue` requires a little more work, as the new `Node` needs to be added at the end of the list rather than the beginning:



In order to add a new item at the end of the list, we need to find the `Node` that is currently last in the linked list and link it to a new `Node` containing the new item. The only way to get to the last `Node` is to follow the links from the head of the list until we reach the end, which we can identify because the link from the final `Node` in the list doesn't point anywhere – it is `null`. This process of following the links in a linked list is known as *chaining*. If we adopt this approach, the `enQueue` method would look something like this:

```
public void enQueue(Object o) {
    if(listHead == null) {
        listHead = new Node(o, null);
    }
    else {
        Node nodePointer = listHead;
        while(nodePointer.next != null) {
            nodePointer = nodePointer.next;
        }
        nodePointer.next = new Node(o, null);
    }
}
```

First of all notice that there is a *special case* when the queue currently contains no elements, which we identify by the fact that `listHead` points nowhere. We will come back to this after looking at the general case when the queue is not empty.

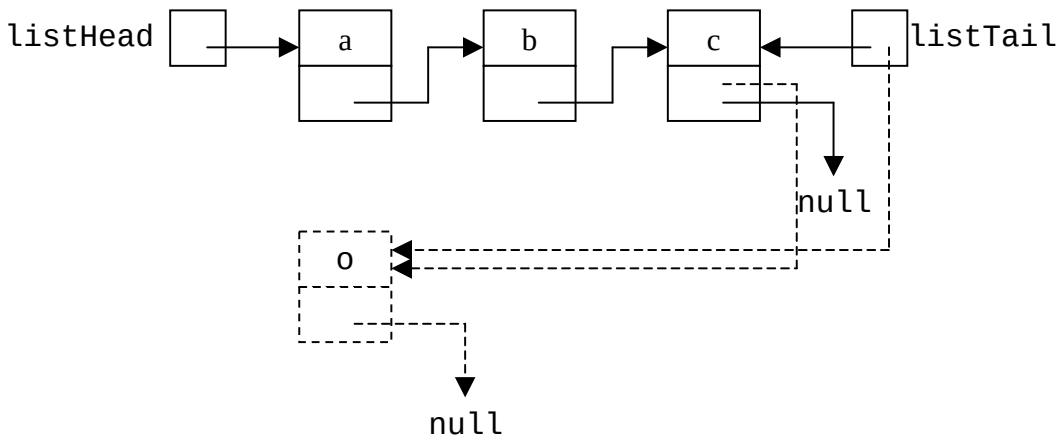
In the general case, we need to find a reference to the last `Node` in the list so that we can change the link in that `Node` to point to a new `Node` containing the new item. A new object reference,

`nodePointer`, of type `Node` is initialised to point to the first `Node` of the list. We then chain down the list, by setting `nodePointer` equal to the `next` field of the `Node` it currently references, which contains a link to the subsequent `Node` in the list. Each of these operations results in `nodePointer` referencing the `Node` one place further down the linked list. This process is terminated when the `next` field of the `Node` that `nodePointer` references is `null`, which tells us that that `Node` is the last one in the list. Finally, we update the link of the `Node` at the end of the list so that it references a new `Node` containing the newly `enQueue`'d `Object`. The link in the new Node is set to `null`, since it is at the end of the list.

When the queue is empty, our chaining operation fails, because it relies on the fact that we can have a reference to at least one `Node` – if `listHead` is `null`, then the predicate in the `while` loop fails. So we have to deal with this special case separately, which we do by setting `listHead` equal to the new `Node` directly.

Chaining is the general purpose method for locating elements of a linked list. However, it is an expensive process, particularly when compared to the equivalent operation on an array. Often we cannot avoid using chaining to perform operations on linked lists, but sometimes it can be avoided for very specific operations. The operation of adding an element to the end of a list turns out to be just such an operation. One extra private data member can be added to the `QueueList` class in order to avoid having to chain down the list and make the `enQueue` operation as efficient as the `push` operation in the `StackList` class.

The trick is to store an extra object reference, `listTail`, that always references the *last* element in the list – this object reference is known as a *tail pointer*. The tail pointer provides a short cut, which we can use to perform operations, such as `enQueue`, that occur at the end of the linked list.



Some extra work needs to be done in the implementation of the class to ensure that `listTail` is maintained, but the computational efficiency of the `enQueue` method is greatly increased. Here is a shortened version of the `QueueList` class, which implements the queue ADT using a linked list with a tail pointer – you can download the full listing from the D0b website.

```
public class QueueList {
    //Member class to hold nodes of the linked list – as for
    //StackList.
```

```
protected class Node {…}

//Private data members – just need head and tail pointers for
//the linked list.
private Node listHead;
private Node listTail;

//Constructor
public QueueList() {
    listHead = null;
    listTail = null;
}

public void enQueue(Object o) {
    //Special case for empty list
    if(listHead == null) {
        listHead = new Node(o, null);
        listTail = listHead
    }
    else {
        listTail.next = new Node(o, null);
        listTail = listTail.next;
    }
}

public Object deQueue() throws QueueListException {
    //Check that the queue isn't empty
    if(listHead == null) {
        throw new QueueListException("Queue Underflow");
    }

    //There is a special case if there is just one element.
    if(listHead.next == null) {
        Object head = listHead.datum;
        listHead = null;
        listTail = null;
        return head;
    }
    else {
        Object head = listHead.datum;
        listHead = listHead.next;
        return head;
    }
}

public Object head() throws QueueListException {
    if(listHead != null) {
       return listHead.datum;
```

```
        }
        else {
            throw new QueueListException("Queue Underflow");
        }
    }

    public boolean empty() {
        return (listHead == null;
    }

    public void clear() {
        listHead = null;
        listTail = null;
    }
}
```

Notice that there are special cases in the `enQueue` and `deQueue` methods. In the case of `enQueue`, there is a special case when the first element is added to the list, as we need to initialise `listTail`, reference the new item, which is the tail of the list as well as the head. For `deQueue`, the special case occurs when the `deQueue` operation results in an empty list. `listTail` must be explicitly set to `null` so that the last `Node` removed from the list can be reclaimed by the garbage collector.

Exercise: one way to avoid these special cases in the implementation of `QueueList` *is to use a dummy* `Node` *at the start of the list. When a new* `QueueList` *is initialised, the constructor of the class must create this dummy* `Node` *and initialise* `listHead` *and* `listTail` *to point to it. The tests for emptiness of the* `QueueList` *will need to be changed accordingly. Adapt the* `QueueList` *class so that it uses a dummy element and avoids these special cases. Write a small test program that uses all the public interface methods of* `QueueList` *to test that the new queue implementation behaves the same way as the original* `QueueList` *class.*

3.4.4 Sequence as a Linked List

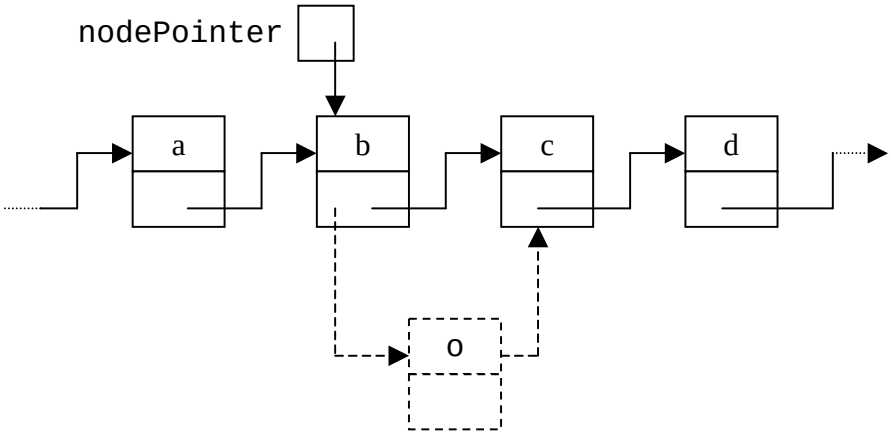
For the sequence ADT, we need to implement the same set of public interface methods listed in section 3.3. We will use the linked list with tail pointer that we used to implement the queue ADT above, as it seems likely that operations at the tail of the list might be important. I won't include the full listing for class `SequenceList`, but you can get it from the D0b web site. The `insertFirst`, `insertLast` and `deleteFirst` methods are similar to the equivalent methods in the `StackList` and `QueueList` classes. We will look at the general `insert` and `delete` methods as well as the `deleteLast` method, which doesn't have an equivalent in either `StackList` or `QueueList`.

Here is the `insert` method:

```
public void insert(Object o, int index) throws
                                    SequenceListException {
    //Check the index is positive
    if(index < 0) {
        throw new SequenceListException("Indexed element is out
```

```
                                                        of range");
        }

        //Special case when the sequence is empty
        if(listHead == null && index == 0) {
            listHead = new Node(o, listHead);
            listTail = listHead;
        }

        //Special case for insertion at the beginning.
        else if(index == 0) {
            listHead = new Node(o, listHead);
        }

        //General case
        else {
            Node nodePointer = listHead;
            int i=1;
            while(i < index) {
                nodePointer = nodePointer.next;
                i += 1;
                if(nodePointer == null) {
                    throw new SequenceListException("Indexed element
                                                    is out of range");
                }
            }
            nodePointer.next = new Node(o, nodePointer.next);

            //If insertion occurs at the end we need to update the
            //tail pointer.
            if(nodePointer == listTail) {
                listTail = listTail.next;
            }
        }
    }
```

Lets consider the general case first. We need to chain down the list until `nodePointer` references the `Node` before the point at which the insertion will occur. Once we have located this position in the list we "hook in" a new `Node` containing the new sequence item.



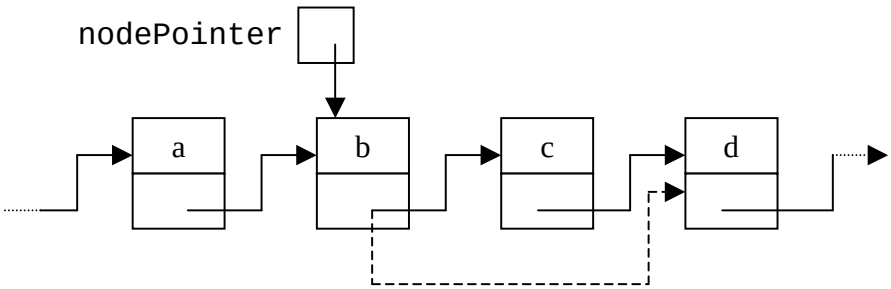
Insertion at the end of the sequence is a special case, because we need to update the tail pointer so that it references the new `Node`, which is the new end of the sequence. Other special cases occur for insertion at the beginning of the sequence and for insertion into an empty sequence.

Here is the `delete` method:

```
public void delete(int index) throws SequenceListException {
    //Check the sequence is not empty
    if(listHead == null) {
        throw new SequenceListException("Sequence Underflow");
    }

    //Check the index is positive
    if(index < 0) {
        throw new SequenceListException("Indexed element is out
                                                    of range");
    }

    //Special case if there is just one item in the sequence.
    if(listHead.next == null && index == 0) {
        listHead = null;
        listTail = null;
    }

    //Special case for deletion of first element.
    else if(index == 0) {
        deleteFirst();
    }

    //General case.
    else {
        Node nodePointer = listHead;
        int i = 1;
        while(i < index) {
            nodePointer = nodePointer.next;
```

```
            i += 1;
            if(nodePointer.next == null) {
                throw new SequenceListException("Indexed element
                                                is out of range");
            }
        }

        //Unhook the node and update the tail pointer if it
        //was the last one in the sequence.
        if(nodePointer.next == listTail) {
            listTail = nodePointer;
        }
        nodePointer.next = nodePointer.next.next;
    }
}
```

Again we need to chain along the sequence, this time to find the `Node` before the one to be removed. Once we have located that `Node`, we unhook the next one from the list by changing the link to reference the `Node` after:



Special cases occur for deletion at the beginning and end of the sequence and for deletion from a sequence with a single element. As with the queue, some special cases could be avoided by using a dummy element at the head of the list.

The final method for `SequenceList` we will look at is the `deleteLast` method. Here is the method:

```
public void deleteLast() throws SequenceListException {
    if(listHead == null) {
        throw new SequenceListException("Sequence Underflow");
    }

    //Special case for sequence with a single item.
    if(listHead.next == null) {
        listHead = null;
        listTail = null;
    }

    //General case
```

```
        else {
            Node nodePointer = listHead;
            while(nodePointer.next != listTail) {
                nodePointer = nodePointer.next;
            }
            nodePointer.next = null;
            listTail = nodePointer;
        }
    }
```

The thing to notice about this method is that, despite the fact that we are using a linked list *with* a tail pointer, which is used to simplify operations at the end of the sequence, we still need to chain all the way along the list in order to delete the last element.

We will finish this section with a quick note on special cases. As we have seen above, it is often necessary to deal with operations at the beginning and end of linked lists separately, as the mechanism is slightly different to the general case. The best approach to designing methods to perform operations on linked lists and identifying special cases is to draw the data diagrams. Draw the diagram for the general case first and sketch out the approach that you will use. Then try your approach on suspicious configurations where you think special cases might occur – operations on empty lists and operations at the beginning and end are always worth testing. Sometimes operations on the first (or last) two (or more) elements of a list can both require separate special cases.

3.5 Variations of Linked Lists

In this section we will consider some variations of the basic linked list used to implement the stack ADT in section 3.4.2, which can improve the efficiency of some operations for certain linear ADTs.

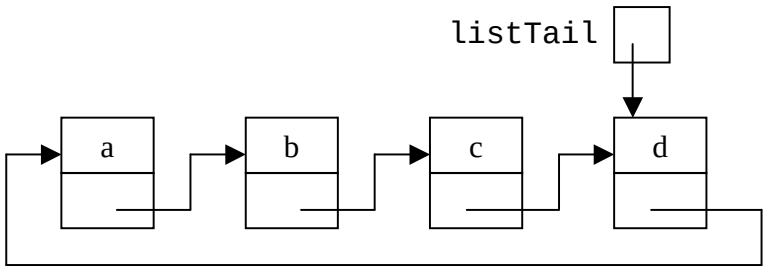
3.5.1 Linked List with Tail Pointer

In order to improve the efficiency of the insertion operation at the end of the list in the queue and sequence ADTs above, we made use of one variation of the basic linked list, the linked list with tail pointer. The tail pointer gives us access directly to the last `Node` in a linked list. This improves the efficiency of the operation of insertion at the end of the sequence, as we saw in the implementation of the queue ADT in section 3.4.3, by avoiding the need to chain all the way down the sequence from the beginning. The operation of accessing the element at the end of a sequence is also reduced to a single operation, as you can see in the full listing of the `SequenceList` class. However, the use of a tail pointer does not help the operation of deletion at the tail of the list, which still requires chaining as we saw in the implementation of the `deleteLast` method of the `SequenceList` class in section 3.4.4.

3.5.2 Circular Linked List

The circular linked list is an alternative to the two pointer approach above. A single pointer is maintained, which references the tail of the list. The link of the last element of the sequence points back

to the beginning of the sequence rather than being set equal to `null`. The list is thus circular in the sense that if you follow the links, you can always get back to where you started from:

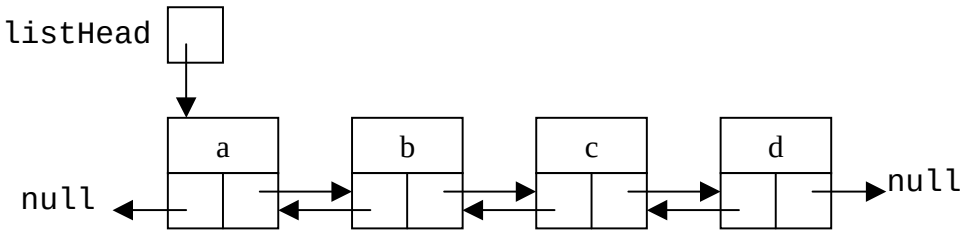


We only need to store the tail pointer, since the head of the list is just `listTail.next`. The circular list is a good choice for the implementation of a queue – try drawing the diagrams to see that both operations can be done efficiently – but we still need to chain around the list to remove the last element in a sequence ADT.

Exercise: sketch out a `clear` *method for the circular linked list. Be careful!*

3.5.3 Double Linked Lists

A double linked list is one in which each node contains two pointers: one as usual to the next element of the list, but also one back to the previous element.



In a double linked list, traversal of the list can be performed in either direction, but there is some sacrifice in terms of used memory space. If combined with a tail pointer or made circular, the operation of deletion at the tail of a sequence can be implemented efficiently with a double linked list, as well as the other operations at the beginning and end of the sequence. Draw the data diagrams to convince yourself of this.

3.6 Other List ADTs

There are some other linear ADTs that crop up from time to time.

3.6.1 Deque

Deque is short for “double entry queue”. A deque is a list that can be added to or deleted from at either the first or last positions, but nowhere else. It has more general utility than a stack or a queue, in as

much as it can perform the job of a stack, but less than a sequence. Despite the appeal of the symmetry of this ADT, there don't seem to be many applications for it.

Exercise: what sort of ADT would be appropriate for a simple time-sharing operating system on a dual processor machine?

3.6.2 Scroll

A scroll can be added to at one end, but deleted from at either the front or the back. It is intermediate between a queue and a deque.

Exercise: Think about what kind of linked lists are most appropriate for deques and scrolls in order to optimise the efficiency of the important operations on these ADTs, while keeping the complexity of the implementation to a minimum.

3.7 Inheritance and Association

We have noted several times the connection between stacks, queues and sequences. The sequence ADT has more general utility than queues and stacks (deques and scrolls too), because it could be used in place of either. In the implementations of the three ADTs there is a lot of code repetition. We should be able to avoid this repetition by use of the object oriented design features of Java. We will consider this in this final part of this section of the course.

3.7.1 Inheritance

Inheritance is the first thing that springs to mind. Can we implement a sequence ADT and then derive the restricted ADTs queue and stack? In fact this is how the `Stack` class that comes in the standard Java libraries is implemented – it is derived directly from the `Vector` class. The listing below uses a similar approach to implement a "stack" ADT via inheritance from a previously implemented `Sequence` class with all the public interface methods listed in section 3.3:

```
//Bad Stack implementation – don't do this!
public class Stack extends Sequence {
    public Stack() {
        super();
    }

    public void push(Object o) {
        insertFirst(o);
    }

    public Object pop() throws SequenceException {
        Object top = first();
        deleteFirst();
        return top;
    }

    public Object top() throws SequenceException {
```

```
            return first();
        }
    }
```

The `empty` and `clear` methods are inherited directly from the parent class. A queue class could be derived in a similarly simple way.

Despite the simplicity of this implementation and the fact that the repetition of code has been avoided, the object oriented purist would be very unhappy about this `Stack` class, as they are about the `Stack` class in the standard Java libraries. And rightly so! The problem with this class is that the inheritance relationship allows access to all the public interface methods of the `Sequence` class. For any instance of the `Stack` class above, we can call any of the sequence methods so we can perform insertion, deletion or access operations at any point in the list of elements. This is highly undesirable and in fact means that the ADT implemented by the class above isn't a stack at all!

This is really not the way inheritance should be used. Recall that the idea of inheritance is *extension* not *restriction*. Classes that define objects with increased functionality are generally further down the inheritance tree than classes with fewer operations. So perhaps we could use inheritance the other way around and derive a sequence ADT from a stack or a queue?

This is a better idea and is proper use of inheritance. We can certainly extend an implementation of a queue or a stack ADT to a sequence, by including the additional set of public interface methods. Other inheritance relationships exist between queues and stacks and the intermediate linear ADTs, deques and scrolls. For example, it would be straightforward to implement a deque by inheritance from a queue. This is proper use of inheritance, since the deque ADT is an extension of a queue – it is a queue with the additional operations of deletion from the tail and addition at the head.

Unfortunately, if we try to use inheritance to implement the whole suite of linear ADTs, we cannot avoid having some code repetition. Although sequences extend either stacks or queues, there is no inheritance relationship between stacks and queues themselves. We need to implement both of these from scratch and then choose one to extend for implementations of the more general ADTs. This is not a major issue, as only a small amount of code repetition occurs (deletion at the start of the list).

3.7.2 Association

An alternative way to exploit the similarity of our set of linear ADTs and avoid any repetition of code is not to use inheritance at all, but *association*. We associate the ADTs by using a sequence to store the values contained in a stack or a queue and write a new set of public interface methods. An instance of the `Sequence` class is declared as a private data member of the `Stack` or `Queue` class. Here is such an implementation of a stack:

```
public class Stack {
    //The sequence is a private data member
    private Sequence s;

    public Stack() {
        s = new Sequence();
    }
```

```
    public void push(Object o) {
        s.insertFirst(o);
    }

    public Object pop() throws SequenceException {
        Object top = s.first();
        s.deleteFirst();
        return top;
    }

    public Object top() throws SequenceException {
        return s.first();
    }

    public boolean empty() {
        return s.empty();
    }

    public void clear() {
        s.clear();
    }
}
```

One drawback of this approach is that it may be wasteful in terms of storage space. An implementation of a sequence ADT will typically use a more complex type of linked list, in order to optimise common operations, than is necessary for simpler ADTs, like stack and queues. This is, however, the correct way to implement one class that is a *restriction* of another and NOT through the use of inheritance.

Exercise: implement a queue ADT by association with one of the sequence classes we've discussed in this section and test its functionality.

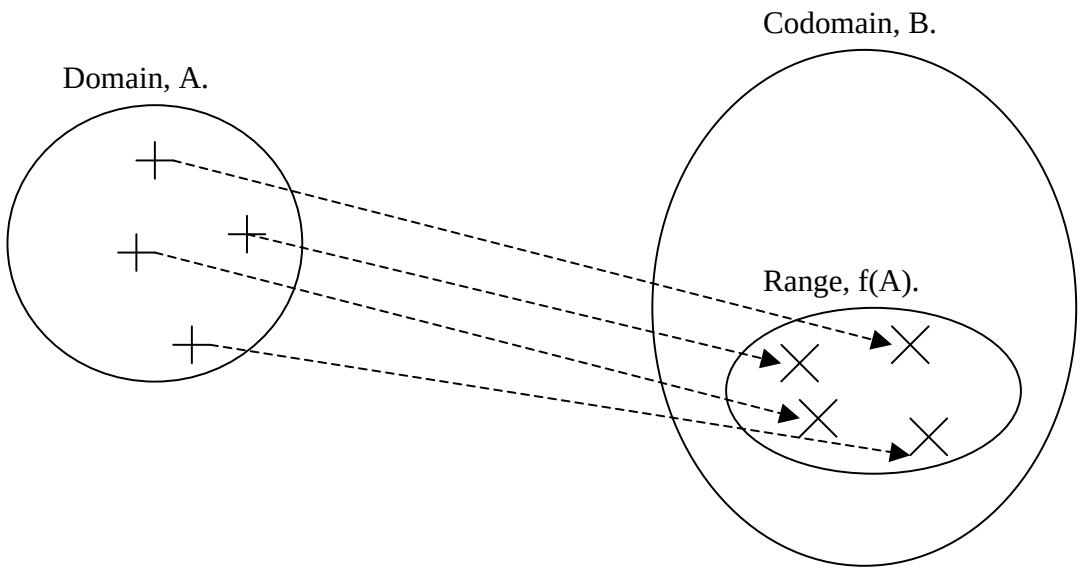
4 Look-Up Tables and Searching

In this section of the course we will investigate *look-up tables*, also known as associative arrays, dictionaries or symbol tables, and consider some different ways of accessing data from these structures, or *searching* look-up tables (LUTs).

For extra information on the topic of this section, I recommend D. E. Knuth, *Sorting and Searching. Vol. 3 of the Art of Computer Programming*. It is a little unfortunate in its choice of presentation of algorithms, but contains more information about searching than anything else. The material used to carry out elementary analysis of the performance of the searching algorithms in this section will be expanded and done properly in the Complexity lectures.

We often need to store data values together with some identification. For example, personnel records identified by a name or by a payroll number; or in a dictionary, French words identified by their English translation, or vice versa. In general, we want to store values of a function f(x), which are identified by x.

A function, f, maps elements from a set A to elements of a set B. Set A is called the *domain* of f and set B is called the *codomain*. The *range* of f, f(A), is the subset of the codomain, B, into which the domain, A, maps.



The idea of a LUT is that we maintain a number of *entries* in the table, each of which comprises an identifying *key* together with a *value*. Given a particular key we can retrieve the associated value from the LUT. This relates to the diagram above in the following way:

- The set of keys forms the domain, A.
- The set of all possible values forms the codomain, B.
- The set of actual values we have stored forms the range, f(A).
- The LUT implements the function f. So that value = f(key).

If we refer back to the examples given above, and suppose we have a bunch of employee records (values), which we wish to index via payroll number (keys). Then

- The domain A is the set of keys, which is the set of *assigned* payroll numbers.
- The codomain B is the set of *all possible* values. So B is a very large set that contains all possible employee records, i.e., all conceivable records with every combination of name, salary, etc.
- The range, f(A), is the set of actual employee records – one for each payroll number.
- A LUT that holds these employee records would implement the function that maps payroll numbers to employee records. If you pass the LUT a payroll number, it gives you back the corresponding employee record.

If a Mr. Jones has payroll number 44100033, then we would like to be able to access Mr. Jones' Employee record through the use of a command like:

```
Employee jonesRecord = lut.retrieve(44100033);
```

where `lut` is an object of type look-up table, which has a method `retrieve`, to which we pass a payroll number and receive back the instance of the `Employee` class (see section 2) that contains the information on Mr. Jones.

A LUT is thus a similar idea to an array, or other linear data structure, in that we access each stored value via a single index. In an array, however, elements are distinguished by an integer index that corresponds to the position in the ordered structure, whereas a LUT is an unordered set where elements are distinguished by labels (or keys), which can be of any type, in general.

Similarly for the English to French dictionary

- The domain is the set of English words that are in the dictionary (probably not all).
- The codomain is the set of all French words.
- The range is the set of French translations for the English words in the dictionary.
- The LUT implements the translation function.

In this example, the keys would be strings. If `eToFlut` is a LUT that implements the English to French dictionary, we would like to be able to use commands like:

```
String frenchHouse = eToFlut.retrieve("house");
```

to set the string `frenchHouse` to an appropriate value, like `"maison"`.

Actually, the dictionary example isn't quite appropriate, because the function it implements may not be one to one. There may be some English words that have more than one equivalent in French, which means that some keys refer to more than one value, although we could, of course, store a number of words in a single value. The LUTs that we will consider in this section only implement one to one functions, but there are techniques to cope with one to many functions. We will see some of these later when we look at hash tables.

4.1 Look-Up Table ADT

We will start by defining a look-up table abstract data type. In order to do this we will also define a new type `Entry`, which consists of a key and a value.

- The set of values taken on by a LUT ADT is: the set of sets of entries or (key, value) pairs.

The set of operations we wish to perform on a LUT are the following:

- *Retrieve* – take a key and return the associated value, f(key).
- *Update* – take a key and a value and change the value associated for the key to the new value.
- *Insert* – take a new {key, value} pair and add them to the table.
- *Delete* – take a key and remove the corresponding entry from the table.

Notice that the update operation is redundant, because we could just delete an existing entry to the table and insert a new one with the same key but different value.

It is a mistake to think of a LUT as a list of entries, although this is a possible *implementation*. Lists have an implicit order, which LUTs, in general, do not. LUTs are based on sets and thus have no imposed ordering. We can (and will) envision ordered LUTs, but ordering is not part of the ADT, it is an implementational decision that does not affect the external behaviour.

There are several possibilities for implementing a LUT. Despite the observation in the previous paragraph, we will begin by considering the implementation in terms of a list, before going on to some more elaborate and efficient implementations.

4.2 Searching Unordered LUTs – Linear Search

The simplest strategy to adopt for a LUT implementation is to use a list to store all the entries. Since there is no concept of ordering in a LUT, we can just insert new entries at the easiest place – at the end of an array or the beginning of a linked list. In order to retrieve elements from a LUT like this, given a key, we go through each entry in the list starting from the beginning until we find a key that matches or reach the end of the list. This retrieval strategy is known as *linear search*.

4.2.1 Implementation

We have defined the look-up table ADT in terms of its set of values and operations above. Now, we can outline the public interface for a LUT class:

```
/**
 * Inserts a new entry into the table.
 */
public void insert(String key, Object value)

/**
 * Removes the entry associated with the specified key.
 */
public void remove(String key)

/**
 * Updates the value associated with the specified key.
 */
public void update(String key, Object value)
```

```
/**
 * Retrieves the value associated with the specified key.
 */
public Object retrieve(String key)

/**
 * Returns a string containing all the key-value pairs in the
 * table.
 */
public String toString()
```

Notice that the type of the `key` parameter is `String`. In general, a key can be any object that can be tested for equality with another object of the same type. For the purposes of the illustrations in this section, however, the type of all keys is restricted to `String`. In the next subsection we will also need to compare keys and decide whether one is less than another. It is simple to adapt the code shown in this section so that the keys are of an alternative type that allows this kind of comparison, for example, `Integer` or `Float`. For a completely general implementation of the look-up table class, we would like the keys to be of any type that allows ordering comparisons to be made – less than, greater than, etc. Implementation of this gets a little involved so I've sidestepped it here. If you are interested in how this can be done, look at the implementations in Winder and Roberts or Wiener and Pinson. This generality is achieved through the use of `Interface`'s, which are described in both books, or in the online Java documentation.

In fact, using `String`'s as keys provides considerable generality, as it is simple to represent `Integer`'s, `Float`'s, etc, as `String`'s. In languages that contain associative arrays as standard data structures, such as Perl, it is common practice only to allow keys to be strings.

The data structure we will use to store the look-up table is a `Sequence`, in fact we will use an instance of the class `SequenceArray` that we implemented in the previous section. We could equally well have used a `SequenceList`, but since many indexing operations occur in the middle of the sequence the use of the array based sequence is more efficient. A `Vector` would also be a good choice. Each element of the sequence will be an `Entry` containing a {key, value} pair.

Here is an outline of a simple look-up table class, `LinearLUT`:

```
public class LinearLUT {
    protected class Key {
        public Key() {
           kString = null;
        }
        public Key(String s) {
           kString = s;
        }

        public boolean equals(Key k) {
           return kString.equals(k.toString());
        }
```

```
        public boolean lessThan(Key k) {
            return (kString.compareTo(k.toString()) < 0);
        }

        public boolean greaterThan(Key k) {
            return (kString.compareTo(k.toString()) > 0);
        }

        public String toString() {
            return kString;
        }

        private String kString;
    }

    protected class Entry {
        public Entry(Key k, Object v) {
            key = k;
            value = v;
        }

        protected Key key;
        protected Object value;
    }

    //Single protected data member
    protected SequenceArray seq;

    public LinearLUT() {
        seq = new SequenceArray();
    }

    //Public Interface methods
    //:
    //:
}
```

There are two member classes, `Key` and `Entry`, of the top-level class `LinearLUT`. Although the keys of our table are just strings, we have encapsulated them in a class, `Key`, because we want to implement the additional comparative tests, `lessThan` and `greaterThan`. In fact we won't use these methods in this implementation of the look-up table, but we will use them in the slightly more sophisticated implementations in the next subsection. The `compareTo` method of the `String` class returns an integer less than zero if the string whose method is called is lexicographically *before* the argument to the method, and a positive integer if it is lexicographically after. Encapsulating `Key` in a class in this way provides a general framework, which can be used for implementations in which keys are of types other than `String`.

The class `Entry` encapsulates the idea of a {key, value} pair. Every entry in our LUT will be stored in an object of type `Entry`. Thus every object that is stored in the sequence is an instance of the `Entry` class.

Here are the public interface methods of the class `LinearLUT`:

```
public void insert(String key, Object value) throws
                                        SequenceArrayException {
    Entry newEntry = new Entry(new Key(key), value);
    seq.insertLast(newEntry);
}
```

The simplest insertion operation on an array based sequence is insertion at the end of the sequence, because no array shifting is required. So we create a new `Entry` object containing the new `Key` and the corresponding value, which is of general type, `Object`. This new `Entry` is placed at the end of the sequence. We need to be aware of the possibility that the sequence holding our LUT can fill up and the `seq.insertLast` method can throw a `SequenceArrayException` if sequence overflow occurs. Rather than try to handle this occurrence in the `insert` method above, we simply declare that this method too can throw the same type of `Exception`, leaving it to the user of our LUT class to decide what alternative action should be taken.

```
public void remove(String key) throws KeyNotFoundInTableException
{
    Key searchKey = new Key(key);
    int index = findPosition(searchKey);
    if(index >= 0) try {
        seq.delete(index);
    } catch (Exception e) {
        System.out.println(e);
    }
    else {
        throw new KeyNotFoundInTableException();
    }
}
```

The method `findPosition` implements the searching strategy for our LUT, which is linear search here. The listing for `findPosition` is below, but for now, it returns the index of the element of the sequence that contains the key equal to `searchKey`. If none of the `Key`'s in the `Entry`'s in the sequence match the search key, `findPosition` returns `–1`. We create an instance of the `Key` class from the `String` that is passed to the `remove` method, so that we can compare it directly with the keys in the `Entry` objects in the sequence.

There is a lot of code in this method and the ones below that deals with `Exception` handling. Many of the uses of the underlying sequence object that are made within this class are "safe", in that we know they will never cause `Exception`s from the sequence class, for example, we know that this code will never try and access an element beyond the end of the sequence. We thus do not require a user of the LUT to handle the `Exception`s that are thrown by the `SequenceArray` class explicitly (using `try` and `catch`), because we know they will never arise. That requirement is removed by putting the `try`

and `catch` construct inside these methods, rather than including `throw SequenceArrayException` in the definitions of the methods. The only time a `SequenceArrayException` can occur is when the sequence overflows. We cannot guarantee that this won't happen. It can only occur in the `insert` method, which, as we saw above, must be allowed to throw this `Exception`.

```
public Object retrieve(String key) throws
                                        KeyNotFoundInTableException {
    Key searchKey = new Key(key);
    int index = findPosition(searchKey);
    if(index >= 0) try {
        Entry searchEntry = (Entry)seq.element(index);
        return searchEntry.value;
    } catch (Exception e) {
        System.out.println(e);
    }
    else {
        throw new KeyNotFoundInTableException();
    }
    return null;
}
```

For the `retrieve` method, again we call `findPosition` to locate the element of the sequence corresponding the entry of the table that is sought. If there is an `Entry` with `key` corresponding to the `String` passed to the `retrieve` method, the `value` part of that `Entry` is returned. Notice that we need to cast the return type of `seq.element` to `Entry`. The return type of the method `SequenceArray.element` is `Object`, but we know we can perform this cast, because we only ever store objects of type `Entry` in the sequence.

```
public void update(String key, Object value) throws
                                        KeyNotFoundInTableException {
    Key searchKey = new Key(key);
    int index = findPosition(searchKey);
    if(index >= 0) try {
        Entry searchEntry = (Entry)seq.element(index);
        searchEntry.value = value;
    } catch (Exception e) {
        System.out.println(e);
    }
    else {
        throw new KeyNotFoundInTableException();
    }
}
```

This method is similar again. Once the `Entry` corresponding to the key provided has been found, the `value` of that `Entry` is changed to the new specified `value`.

```
public String toString() {
```

```
        String output = “”;
        for(int i=0; i<seq.size(); i++) try {
            Entry tableEntry = (Entry)seq.element(i);
            output = output + tableEntry.key.toString();
            output = output + “:”;
            output = output + tableEntry.value.toString();
            output = output + “, “;
        } catch(Exception e) {
            System.out.println(e);
        }
        return output;
    }
```

This method creates a `String` that lists all the {key, value} pairs in the table, in the order that they are stored in the sequence.

Finally, there are some non-public member functions to this class that implement the searching of the look-up table to retrieve values. We have encapsulated the search strategy in these separate methods for two reasons:

- because the searching mechanism is required in several different public interface methods, so we use procedural abstraction and encapsulate it in a separate method.
- so that we can use inheritance to override them with implementions of other more efficient searching algorithms.

```
    protected int findPosition(Key k) {
        return linearSearch(k);
    }

    protected int linearSearch(Key k) {
        for(int i=0; i<seq.size(); i++) {
            if(k.equals(keyAt(i))) {
                return i;
            }
        }
        return –1;
    }

    protected Key keyAt(int index) {
        try {
            Entry entryAt = (Entry)seq.element(index);
            return entryAt.key;
        } catch(Exception e) {
            System.out.println(e);
        }
        return null;
    }
```

The `findPosition` method returns the result of `linearSearch` directly. `linearSearch` goes through the sequence from the beginning comparing the key of each `Entry` to the `searchKey`, using the `keyAt` method. If it finds a key equal to the `searchKey` the corresponding index is returned. If the end of the table is reached without finding a match, `-1` is returned to indicate that the search failed.

4.2.2 Simple Test Program

Here is a simple test program to illustrate how to create an instance of and use a look-up table. A `linearLUT` is used to store a set of peoples ages indexed by their first names.

```
class LUTTest {
    public static void main(String[] args) {

        LinearLUT myLUT = new LinearLUT();

        myLUT.insert("Priscilla", new Integer(41));
        myLUT.insert("Steven", new Integer(34));
        myLUT.insert("Samuel", new Integer(28));
        myLUT.insert("Helena", new Integer(39));
        myLUT.insert("Andrew", new Integer(14));
        myLUT.insert("Kay", new Integer(24));
        myLUT.insert("Hristo", new Integer(67));

        System.out.println(myLUT);
        System.out.println(myLUT.retrieve("Hristo"));

        myLUT.update("Samuel", new Integer(29));
        myLUT.remove("Andrew");

        System.out.println(myLUT);
    }
}
```

Exercise: What is the output of the above program?

4.2.3 Analysis of Linear Search

The retrieval process is dominated by the `for` loop in the `linearSearch` method. At each position in the sequence we perform a key comparison – `searchKey.equals(keyAt(i))`. We will make this comparison the *unit of cost* and analyse this search strategy in terms of the number of these comparisons that have to be performed.

Suppose we have a LUT that contains n entries.

- *Unsuccessful search:* the search key is compared with every key in the table.

cost = n

- *Successful search:* if the entry is at position k in the sequence, we make k comparisons
  - *Best case:* k = 1, the entry is at the start of the sequence.

    cost = 1
  - *Worst case:* k = n, the entry is at the end of the sequence.

    cost = n
  - *Average case:* Suppose p(k) is the probability that the entry is at position k.

$$\text{average cost} = \sum_{k=1}^{n} p(k) \times (\text{cost when entry is at position k}) = \sum_{k=1}^{n} p(k).k$$

The cost when the entry is at position k is obviously k, since we make comparisons at this entry and each of the (k-1) preceding ones, but what about p(k)? It is possible to go into some very detailed analyses about expected probability distributions (especially if some of the tricks discussed in the next subsection are used), but in the absence of any detailed knowledge, we will assume that all positions in the sequence are equally likely.

Thus, p(k) = B, a constant. Since probabilities must sum to 1, we know that

$$\sum_{k=1}^{n} p(k) = 1$$

$$\Rightarrow nB = 1 \Rightarrow B = \frac{1}{n}.$$

So,

$$\text{average cost} = A(n) = \sum_{k=1}^{n} \frac{k}{n} = \frac{1}{n}(1 + 2 + ... + n) = \frac{1}{2}(n+1)$$

In summary then, we have that the worst case performance, W(n), and the average case performance, A(n) are both “order n”, O(n).

4.2.4 Self Organising Lists

There are various heuristic techniques that can be used to speed up linear search.

Experience shows that, in many applications, entries obey *locality of reference*, i.e., the same small set of entries tends to be accessed in bunches. Thus, we can often speed things up by storing the most recently accessed elements at the front of the list, where the search begins. This is a similar idea to caching in operating systems. There are a number of strategies that are used, some examples are:

- *Move-to-front:* whenever an entry is accessed, move it to the front of the sequence. The sequence ends up ordered by recentness of access, with the most recently accessed entries at the front.

- *Migrate-to-front:* whenever an entry is accessed, swap it with its predecessor in the sequence and so move it one place closer to the front of the sequence. Popular entries gradually migrate to the front of the sequence, where they can be accessed more quickly.
- The most sophisticated techniques store a frequency access table and order the sequence with the most frequently accessed at the front.

Note that linear search remains O(n) in the worst case when any of these techniques are adopted.

4.3 Searching Ordered LUTs

The linear search described in the previous section only takes into account whether two keys are equal or not equal. This reflects the idea of the LUT ADT, which is an unordered set. However, just because the LUT represents an unordered set, doesn't mean we can't use ordering in the *implementation* of the LUT in order to make it more efficient. Intuitively, it seems that we could make the searching process easier if we stored the entries in the LUT so that the keys are in order, i.e., for any `i` and `j`, such that `i < j < seq.size()`, `keyAt(i)` "is less than" `keyAt(j)`. Consider a telephone directory. It's a look-up table in which the keys are names and the values are telephone numbers. The fact that the names are stored in alphabetical order allows us to find the key we are interested in much more quickly than if they were stored unordered.

In fact, in our ordered LUTs we will order entries in the opposite sense so that the "largest" key comes first and the smallest last. To do this we require some way of comparing keys to decide which of two keys is the larger and which is the smaller. In other words, our Key ADT requires the comparative operations: `lessThan` and `greaterThan`. This restricts the set of objects that we can use as keys to those of a type that can be compared in this way. In practice, this is not a major restriction and, in fact, we have already restricted ourselves further than this by allowing only `String`'s as the underlying type of our `Key` class. With a little care, we could be more general than this by defining a key ADT that encompasses all objects that can be compared in this way (see Winder and Roberts or Wiener and Pinson). We won't get into this here, however, and we'll stick with our `String` based keys, which, as we noted before, can be adapted quite simply to use other types that support comparisons. We have already included `lessThan` and `greaterThan` operations in our `Key` class, but we didn't use them in the implementation of `LinearLUT`. In this section we will look at some implementations of the LUT that use ordered storage of the entries to improve the efficiency of searching the LUT.

4.3.1 Linear Search in an Ordered LUT

In order to implement a LUT in which the entries are stored in order of their key value, we need to write a new `insert` method that it puts new entries in the sequence in order. We also need to adapt the `linearSearch` method so that it exploits the new ordering. We can do this very easily by inheritance from our `linearLUT` class. `insert` and `linearSearch` are overridden, but everything else is left unchanged.

Here is the complete listing:

```
public class OrderedLinearLUT extends LinearLUT {
    public OrderedLinearLUT() {
        super();
```

```
    }

    public void insert(String key, Object value) throws
                                            SequenceArrayException {
        Entry newEntry = new Entry(new Key(key), value);
        int index = 0;
        while(index < seq.size() &&
                          newEntry.key.lessThan(keyAt(index))) {
            index += 1;
        }

        seq.insert(newEntry, index);
    }

    protected int linearSearch(Key k) {
        int i=0;
        while(i < seq.size() && k.lessThan(keyAt(i))) {
            i += 1;
        }

        if(i < seq.size() && k.equals(keyAt(i))) {
            return i;
        }
        return –1;
    }
}
```

So the `insert` method goes along the sequence until either the end of the sequence is reached, or an `Entry` is found with a `Key` that the new `Key` is not less than, i.e., a `Key` that is less than or equal to the new `Key`. The new `Entry` is the inserted into the sequence at the index of that `Entry`, which means that it goes just before it. Notice that this means that the entry with the *largest* key is at the beginning of the sequence and the keys get smaller, in order, as we go along the sequence.

If we adapt the short test program in section 4.2.2 to use an `OrderedLinearLUT` rather than a `LinearLUT`, we find that the entries are stored in the sequence in the following order: {Steven: 34}, {Samuel: 28}, {Priscilla: 41}, {Kay: 24}, {Hristo: 67}, {Helena: 39}, {Andrew: 14}. Rather than being in the order in which they were added to the LUT, as they were in `LinearLUT`, they are now stored in reverse alphabetical order of the keys.

In the `linearSearch` method, we still start at the beginning of the sequence and compare each key in turn – that is linear search. Now however, we know that if we encounter a key that is less than the search key, we can stop and the search has failed. This is the only advantage of ordering the sequence if we adopt a linear search strategy. The complexity of *successful* searches is unchanged from the unordered sequence, it is still O(n), but for unsuccessful searches, we no longer have to check every entry in the sequence.

If we assume that an unsuccessful search is equally likely to be terminated at any of the positions 1…n, the average number of comparisons is:

$$A_{\text{unsuccsessful}}(n) = \frac{1}{n}\sum_{k=1}^{n}(k+1) = \frac{1}{2}(n+3)$$

which is just over half the cost of an unsuccessful search in an unordered table, but still O(n). Notice that the number of comparisons for position k is k+1, since we have k `lessThan` comparisons and one `equals` comparison.

4.3.2 Binary Search

We can exploit the ordering of our LUT entries in the sequence properly if we take note of whether a failed comparison was "less than" or "greater than". This is a familiar idea. If you were looking up a telephone number, you wouldn't scan all the entries in a directory from the beginning of the directory, you start looking roughly where you expect to find that entry in the (ordered) directory. For example, if you are looking for someone called Smith, you open the directory about three quarters of the way through. If you find yourself among the U's, you flick back a few pages, if you're among the P's you go forward a bit. Then you look again and decide which way to turn the pages depending on whether the names you see are lexicographically less than or greater than the name you are looking for. Eventually, you narrow things down so that you either find the number you're looking for or decide that it's not there.

We can use a similar strategy to search LUTs and it is known as *binary search*. In the case of a telephone directory, you know roughly where to start looking, because you know something about the distribution of surnames in the alphabet. In a general LUT, you have no prior knowledge about the distribution of keys so you might as well start your search in the middle. If we compare a search key with the key in the middle of the sequence storing our LUT, we know that if our search key is less than that key, we only need to search the top half of the sequence. If our search key is greater, we only need to search the bottom half. Having cut down the set of keys we need to search in this way, we repeat the process and compare our search key to the middle one of the new range – the top or bottom half of the table – to find which *quarter* of the table our key is in. The range of the table that we are searching rapidly gets smaller, and will eventually reduce to a single entry in the sequence. If the range reduces to zero, we know that the key we are looking for isn't there so the search fails.

Here is an outline of the binary search algorithm:

```
find(key, lut)
begin
    if (lut has size 0) then fail

    let mk = key in the middle of the table
    if (key == mk) then succeed
    else
        if (key < mk) then
            find(key, top half of table)
        else
            find(key, bottom half of table)
        end
    end
end
```

Notice that the algorithm is recursively defined: it calls itself with smaller and smaller portions of the sequence holding the LUT. We will implement the algorithm using recursion, which Java allows us to do. Generally it is more efficient not to use recursion and it is possible to implement this algorithm without doing so.

For a LUT with binary search, we need to ensure that the entries are stored in order, as in the `OrderedLinearLUT` class in the previous section. In fact the only thing we need to change from `OrderedLinearLUT` is the search strategy, which we encapsulated in the `protected` method, `findPosition`. So we can use inheritance again and we only need to override the `findPosition` method, which currently uses `linearSearch`, to call a new `binarySearch` method instead.

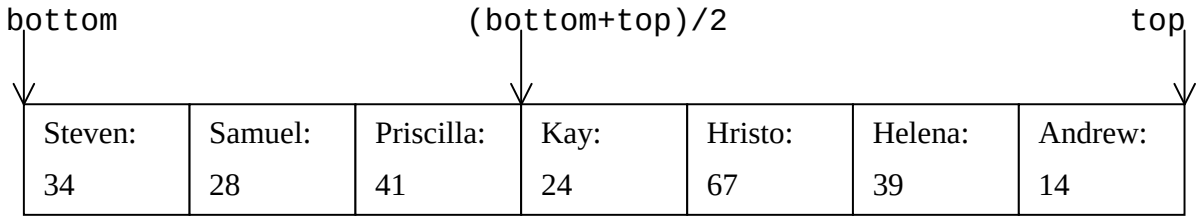
Here is the complete listing:

```
public class BinaryLUT extends OrderedLinearLUT {
    public BinaryLUT() {
        super();
    }

    protected int findPosition(Key k) {
        return binarySearch(k, 0, seq.size());
    }

    protected int binarySearch(Key k, int bottom, int top) {

        //If the range is empty, the search failed.
        if(bottom == top) {
            return –1;
        }

        Key lutKey = keyAt((bottom+top)/2);

        //If this is the one return its index.
        if(k.equals(lutKey)) {
            return (bottom+top)/2;
        }

        //Otherwise search the half of the range contains the key
        else if(k.lessThan(lutKey)) {
            return binarySearch(k, (bottom+top)/2+1, top);
        }
        else {
            return binarySearch(k, bottom, (bottom+top)/2);
        }
    }
}
```

In the `binarySearch` method, the range of the table being investigated is specified by the two sequence indexes `top` and `bottom`. `bottom` indexes the first entry in the range and `top` indexes the first entry in the sequence *after* the range being considered. Suppose we use the same simple test program listed in section 4.2.2, but using a `BinaryLUT`. Let's include an additional operation:

```
myLUT.retrieve("Hristo")
```

Here is the sequence holding our LUT:



When `binarySearch` is first called, `bottom` is initially zero and `top` is `seq.size()`– indexing the next free space in the sequence (=7). The midpoint is then `(bottom+top)/2 = (0+7)/2 = 3`, and so indexes the entry containing the key "Kay". Our search key, "Hristo", is less than "Kay", since it is lexicographically before, so we call:

```
binarySearch(k, (bottom+top)/2+1, top);
```

I.e., we search the range 4 to 7 – we don't need to include entry 3 here, because we have already established that our search key is less than it. In the next recursion then, the midpoint is (4+7)/2, which equals 5 – indexing "Helena". When we do the comparison, we find that our search key is not less than the indexed key, so we call:

```
binarySearch(k, bottom, (bottom+top)/2);
```

Our range is now 4 to 5. The midpoint is now (4+5)/2, which is 4 and we find that our search key equals the key at position 4, so we return the index 4.

The entry is retrieved in three steps rather than the 5 that would be required using linear search. In fact, this is the largest number of steps that is required to retrieve any entry in the table. There is some overhead attached to the use of this recursive procedure and so for small tables like this, we probably wouldn't notice any difference in performance between the two search strategies. For larger tables, however, it can make a huge difference.

Try following the `binarySearch` method through by hand for some other retrieval operations to understand the intricacies of the algorithm. See what happens when the search key is not in the LUT, for example, "Thomas", "Ian", or "Adam".

4.3.3 Analysis of Binary Search

It is fairly easy to see that the number of comparisons in a binary search is $O(\log_2 n)$ for a LUT with n entries.

You will do some proper analysis of this type of process in the Complexity part of the course, but for now, think if it in the following way. The worst case successful search occurs when we end up with a search range of size 1, which contains the entry with the matching key. Suppose this search required k comparisons. Each time one of these comparisons was made, the size of the search range was

(approximately) halved. Thus, for the last comparison, the size of the search range was 1 (= 2^0). For the previous one, the size was approximately 2 (= 2^1) – 3 in fact. For the third to last, approximately 4 (= 2^2); fourth to last 8 (= 2^3), etc. The first comparison is the k-th to last and is performed on the whole range of the LUT whose size must thus have been approximately 2^{k-1}. Thus, if $n \approx 2^{k-1}$, we require k comparisons for binary search in the worst case. So, $k - 1 \approx \log_2(n)$, and so $k = O(\log_2 n)$ and thus $W(n) = O(\log_2 n)$.

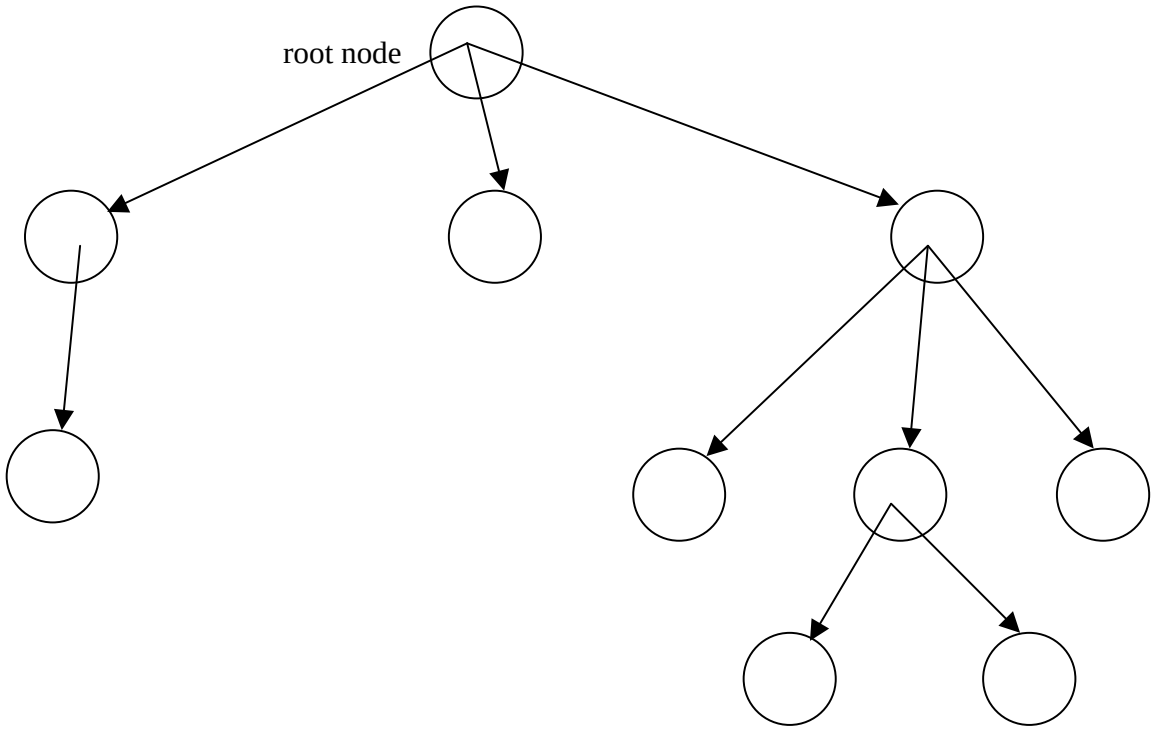
For unsuccessful searches, we require one more recursion than the worst case successful search, as the range has to go down to zero. The complexity is the same - $O(\log_2 n)$.

In fact, it can be shown that no searching algorithm based on key comparisons can perform better than this, i.e., there is no algorithm that has $W(n) < O(\log n)$.

5 Trees

We have taken care to distinguish between Abstract Data Types, which are defined by their functionality and mathematical properties, and Data Structures, which are fundamental building blocks used for implementing ADTs. So far, we have met two *fundamental* aggregate data structures: arrays, and linked lists. In this section, a new one is introduced: the *tree*.

In general, a tree is a structure that consists of nodes, which contain a value and a set of links to other nodes that are the *children* of that node. There is a *root node* at the top of the tree from which all other nodes are descended:



A tree is either empty or it consists of a root node together with a (possibly empty) set of subtrees. Each child of each node in the tree is the root node of a subtree.

A linked list is a kind of degenerate tree, in which there is only one link in every node so each node has at most one child.

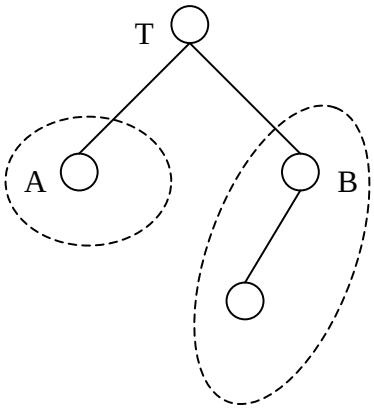
We begin by looking at the simplest type of tree, the *binary tree*.

5.1 Binary Trees

In a binary tree, each node can have at most two children. Binary trees encapsulate the idea of Yes/No decisions very neatly. They are highly recursive structures and concretely illustrate the divide-and-conquer approach to problem solving. Binary trees arise in searching and sorting algorithms, which are among the most studied algorithms in computer science, and are also important in data compression.

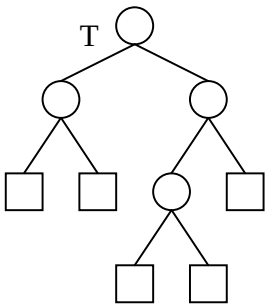
5.1.1 Some Definitions

- A *binary tree* is either *empty* or consists of a *root node* and a pair of trees called the *left subtree* and the *right subtree*.
- Lines connecting nodes to their non-empty subtrees are called *edges.* A non-empty binary tree with n nodes has n-1 edges (*Exercise: prove it).*
- The *size* of a tree is the number of nodes that it contains.



Here is a tree, T, of size 4 and thus with 3 edges. A is the left subtree of T. A is a tree in which the left and right subtrees are empty. B, the right subtree of T, is a tree in which the right subtree is empty.

- It is sometimes convenient to denote empty subtrees by a special symbol – a square – standing for *external nodes.* External nodes are also known as *leaves* or *leaf nodes*. The tree drawn in this way is known as the *extended tree*.



I(T) is defined as the set of all internal nodes of tree T. E(T) is the set of all external nodes. So,

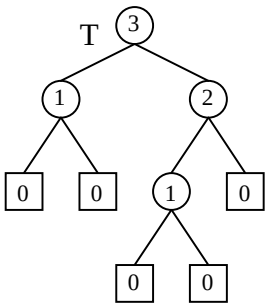
size(T) = Cardinality(I(T)) (= number of elements of I(T)).

A binary tree with n internal nodes has (n+1) external nodes (*Exercise: prove it).*

- Every node except the root node has a unique *parent*.
- The set of *ancestors* of a node x is recursively defined as: x together with the ancestors of its parent (if it has one).
- The set of ancestors form a *path* from x up to the root.
- A *proper ancestor* of x is an ancestor other than x.
- Every node has at most two *children.*
- Two nodes are *siblings* if they share the same parent.
- The set of *descendants* of a node x is recursively defined as: x together with the descendants of its children.
- The set of descendants of x, together with their connecting edges, form a *subtree* rooted at x. This is denoted T_x, if x is a node of tree T.
- A *proper descendant* of x is a descendant other than x.
- $s(x) = size(T_x)$ is the number of descendants of x, i.e., the size of the subtree rooted at x.

5.1.2 Height and Path Length

- The *height*, h(x), of a node x is the number of edges on the *longest* path leading down from x in the extended tree. Equivalently, h(x) is the number of internal nodes on this path, including x if x is internal.
- The height of a tree, h(T), is the height of its root node – 3 in the example to the right.



Nodes labelled by height.

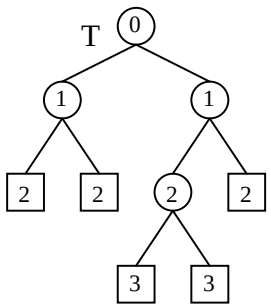
h(T) is a measure of a tree's shape. It is an important factor in the analysis of the complexity of algorithms.

- The *depth*, d(x), of a node x is the number of edges on the path from x to the root. Equivalently, d(x) is the number of internal nodes on this path *excluding* x.
- The *internal path length*, i(T), is the sum of depths of internal nodes of T:

$$i(T) = \sum_{x \in I(T)} d(x).$$

- The *external path length*, e(t), is the sum of depths of external nodes of T:

$$e(T) = \sum_{x \in E(T)} d(x).$$



Nodes labelled by depth.

Theorem:

For any tree, T, of size n,

e(T) = i(T) + 2n.

Proof of this theorem is by induction, see the book by Kingston if you're interested.

5.2 Implementing Trees

Trees are implemented in a similar way to linked lists. We define a class to encapsulate the idea of a node, which contains a value and a set of object references to other nodes. These object references are the edges of the tree. For a binary tree, a node consists of

- a value;
- a link to the left subtree; and
- a link to the right subtree.

so we might define a `BinaryTreeNode` class as follows:

```
protected class BinaryTreeNode {
    protected Object value;
```

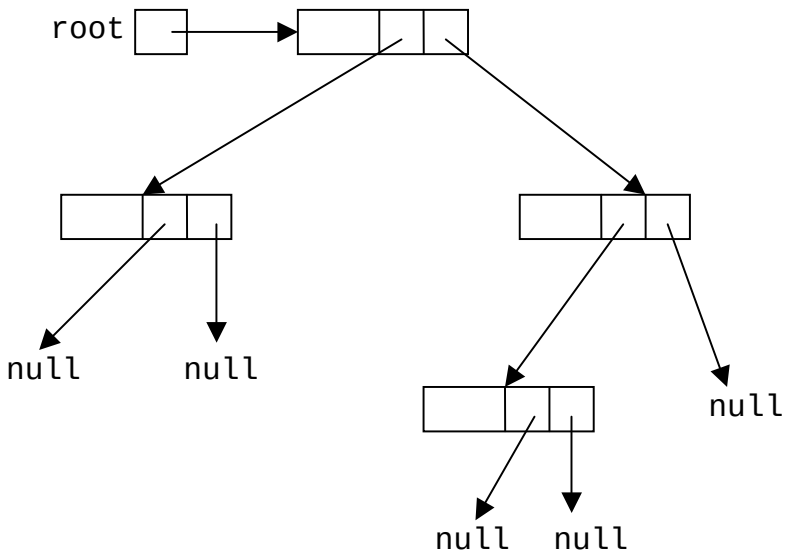
```
        protected BinaryTreeNode left;
        protected BinaryTreeNode right;

        public BinaryTreeNode(Object o) {
            value = o;
            left = null;
            right = null;
        }

        public BinaryTreeNode(Object o, BinaryTreeNode l,
                                              BinaryTreeNode r) {
            value = o;
            left = l;
            right = r;
        }
    }
```

In much the same way as a linked list, the only data item we require in order to implement a binary tree is an object reference to the root node, which we will call `root`. In an empty tree, there are no instances of the `BinaryTreeNode` and so `root` is `null`. Each time a new item is added to the tree, a new instance of `BinaryTreeNode` is constructed, which contains the new item.

So we could implement a general BinaryTree class, that would look something like this:

```
    public class BinaryTree {

        protected class BinaryTreeNode {
            //As above
        }

        public BinaryTree() {
            root = null;
        }

        //Single protected data item.
        protected BinaryTreeNode root;

        //public interface methods, such as insert, delete, retrieve.
    }
```

So the data structure for the tree, T, in the examples above would look like this:



A binary tree is not much use for searching if we don't order it in some way: if we just put new items anywhere in the tree, regardless of how they compare to items already stored in there, we might as well just use an array or a linked list. So it makes sense to use a less general class than `Object` to store the values in nodes of the tree. Much like the type of keys in the LUTs discussed in the previous section, the type of value in a node should be one that has comparative methods, `equals`, `lessThan` and `greaterThan`.

We won't attempt to construct a general binary tree data structure in this way here. Instead, we'll look at an implementation of the LUT ADT that uses a tree structure to store the values and we will see how the binary tree provides a truly dynamic data structure that we can search efficiently.

5.3 Binary Search Trees

All of the LUTs described in section 4 were based on arrays rather than linked lists and if you've attempted part 4 of the unassessed exercises on LUTs, you'll know why. Retrieving elements from the middle of a linked list is a slow process. The array based LUTs are efficient for retrieval operations, but we are stuck with the usual drawback of arrays: they are of fixed size and so the LUT can overflow. A *Binary Search Tree* is a LUT implemented as a binary tree. The binary tree provides a dynamic data structure for the LUT that can be searched efficiently.

In order to exploit the tree structure for key searching, we need to decide upon the way in which items will be ordered in the tree. An item is stored at each node of the tree. We will grow our tree from the root, always adding new items (nodes) at positions that are external nodes (*leaves*) in the current tree. To find the particular external node at which a new item will be added, we start at the root and compare the key of the new entry to the keys of entries already in the table. If an existing key is less than the new key, the new key goes in the left subtree of the tree node containing that key, if not, it goes in the right subtree. We follow a path down to the bottom of the tree, which is determined in this way. When we reach the bottom, i.e., we find an external node (a `null` `BinaryTreeNode` reference), we add a new node containing the new key, value pair.

When we come to search the tree in order to retrieve the item corresponding to a particular key, we can use this ordering to search the tree. We start at the root and compare our key to the key in the root node. If the keys are equal, we are done. If the search key is less than the key in the root node, we know we only have to search the nodes in the right subtree, if it is greater, we only have to search the left subtree. In this way, we find a path down the tree that leads us to the node containing the key that matches the search key (if there is one).

5.3.1 ADT and Public Interface

The definition of the LUT ADT is unchanged from before, since, as we know, it is independent of the underlying representation and implementation, which we are changing from a linear data structure to a tree. Thus we have the same set of public interface methods that we had for the array based LUTs in the previous section:

```
public void insert(String key, Object value);

public void remove(String key);

public void update(String key, Object value);

public Object retrieve(String key);

public String toString();
```

As in the previous section, we will limit ourselves to keys that are `String`'s. We will use the same `Key` and `Entry` classes that we used for our array based LUTs. As usual, these will be member classes in our top-level LUT class. We will use an additional member class, `BSTreeNode`, that encapsulates the idea of a node in a binary tree. The class `BSTreeNode` is slightly less general than the `BinaryTreeNode` class used in the `BinaryTree` class discussed above, because the value it contains is of type `Entry` rather than `Object`. Otherwise it is the same.

Here is an outline of the binary search tree class, `BinaryTreeLUT`:

```
public class BinaryTreeLUT {
    protected class Key {
        //See section 4.2.1
    }

    protected class Entry {
        //See section 4.2.1
    }

    protected class BSTreeNode {
        protected Entry kvPair;
        protected BSTreeNode left;
        protected BSTreeNode right;

        public BSTreeNode(Entry e) {
```

```
            kvPair = e;
            left = null;
            right = null;
        }
    }

    public BinaryTreeLUT() {
        root = null;
    }

    //Single protected data item which reference the root node
    protected BSTreeNode root;

    //Public interface methods.
    //:
    //:
 }
```

The recursive nature of the tree structure makes recursion a natural tool for the implementation of operations on trees. All of our public interface methods make use of recursion.

5.3.2 Insertion

The first public interface method we will look at is the `insert` method:

```
public void insert(String key, Object value) {
    Entry newEntry = new Entry(new Key(key), value));
    BSTreeNode newNode = new BSTreeNode(newEntry);

    addToTree(newNode, root);
}

/**
 * Adds newNode to the binary tree rooted at curNode recursively.
 */
protected void addToTree(BSTreeNode newNode, BSTreeNode curNode)
{
    //Special case for empty tree
    if(root == null) {
        root = newNode;
    }

    //General case
    else if(curNode.kvPair.key.lessThan(newNode.kvPair.key)) {
        //Put the new node in the left subtree
        if(curNode.left == null) {
            curNode.left = newNode;
```

```
            }
            else {
                addToTree(newNode, curNode.left);
            }
        }
        else {
            //Put it in the right subtree
            if(curNode.right == null) {
                curNode.right = newNode;
            }
            else {
                addToTree(newNode, curNode.right);
            }
        }
    }
```

The `insert` method creates a new `BSTreeNode` containing an `Entry` with the new key and value pair. The `left` and `right` pointers are `null`, since we know that the new node will be inserted at the bottom of the tree and so will have no children. The new node is then passed to the recursive method `addToTree`, which finds the ordered position in the tree for the new node, based on the value of the new key, and links it in to the tree data structure.

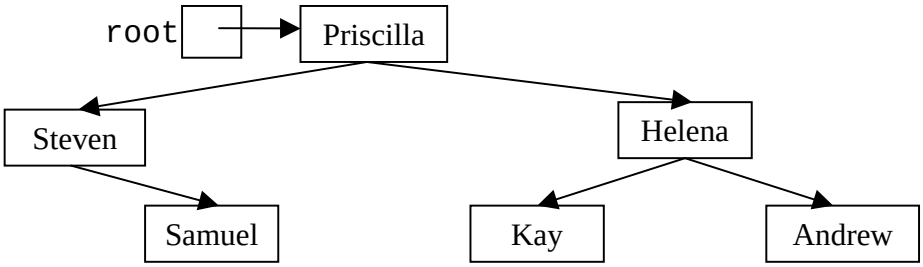
`addToTree` is initially called with the root node of the tree as argument, which is the node currently under investigation – `curNode`. The function compares the key in the new node (`newNode`) to the key at `curNode`. If the key at `curNode` is smaller, then `newNode` is added to the left subtree of `curNode`, otherwise it is added to the right subtree. If the chosen subtree is `null`, then we have found an external node of the tree and so we can link in the new node, otherwise `addToTree` is recursively called again, replacing `curNode` with one of its children.

There is a special case for inserting into an empty tree, as we cannot do the `lessThan` comparison, which we perform in the general case.

Consider our simple test program for LUTs listed in section 4.2.2. It contains the set of `insert` operations:

```
myLUT.insert(“Priscilla”, new Integer(41));
myLUT.insert(“Steven”, new Integer(34));
myLUT.insert(“Samuel”, new Integer(28));
myLUT.insert(“Helena”, new Integer(39));
myLUT.insert(“Andrew”, new Integer(14));
myLUT.insert(“Kay”, new Integer(24));
myLUT.insert(“Hristo”, new Integer(67));
```

Suppose we have performed the first six `insert`’s and we’ll consider the last one. Here is the tree after the first six items have been inserted:



Now we'll try to add the new item with key "Hristo". Initially we call `addToTree` with our new tree node, containing key "Hristo", and a reference to the root of the tree:

```
addToTree(newNode, root);
```

In the `addToTree` method, we perform the test:

```
curNode.kvPair.key.lessThan(newNode.kvPair.key)
```

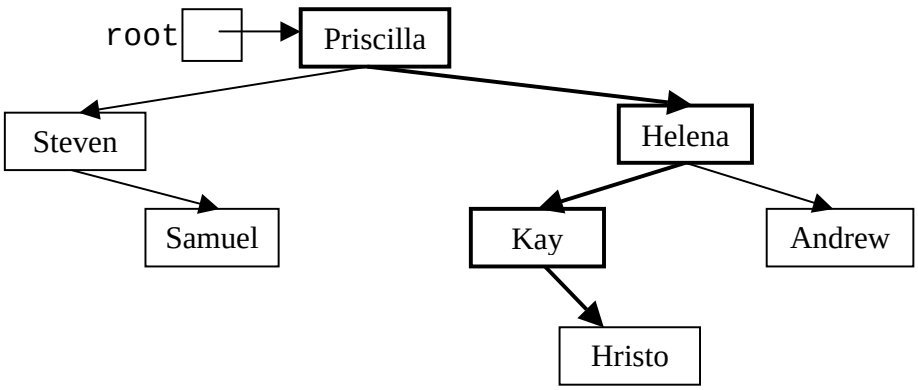
to see if "Priscilla" is less than "Hristo" and we find that it is not. Thus we go to the right subtree, we find that the right subtree is not empty – the test `(curNode.right == null)` fails – so we call:

```
addToTree(newNode, curNode.right);
```

In this new call to `addToTree`, `curNode` becomes `root.right`, which is the node containing key "Helena". Again we compare this to the key in our new node and we find that the "Helena" *is* less than "Hristo", so this time we go to the left subtree. The left subtree is not `null`, so we call:

```
addToTree(newNode, curNode.left);
```

`curNode` now becomes the node containing key "Kay"; "Kay" is not less than "Hristo", so we go to the right subtree. This time we find that the right subtree is empty – `curNode.right` is `null` – and so we can add the new node in by assigning the right subtree of `curNode` to reference `newNode`. The tree finally looks like this:



5.3.3 Retrieval and Update

Here are the `retrieve` and `update` public interface methods:

```
public Object retrieve(String key) throws
                                        KeyNotFoundInTableException {
    Key searchKey = new Key(key);
    BSTreeNode treeNode = getFromTree(searchKey, root);
    return treeNode.kvPair.value;
}

public void update(String key, Object value) throws
                                        KeyNotFoundInTableException {
    Key searchKey = new Key(key);
    BSTreeNode treeNode = getFromTree(searchKey, root);
    treeNode.kvPair.value = value;
}
```

Both methods call the recursive method `getFromTree`, which returns a reference to the node of the tree that contains the search key. Once that node has been found, the retrieval and update operations are trivial: for `retrieve`, we extract the value from the entry in the returned node and return it; for `update` we assign the value to the new value.

Here is the `getFromTree` method:

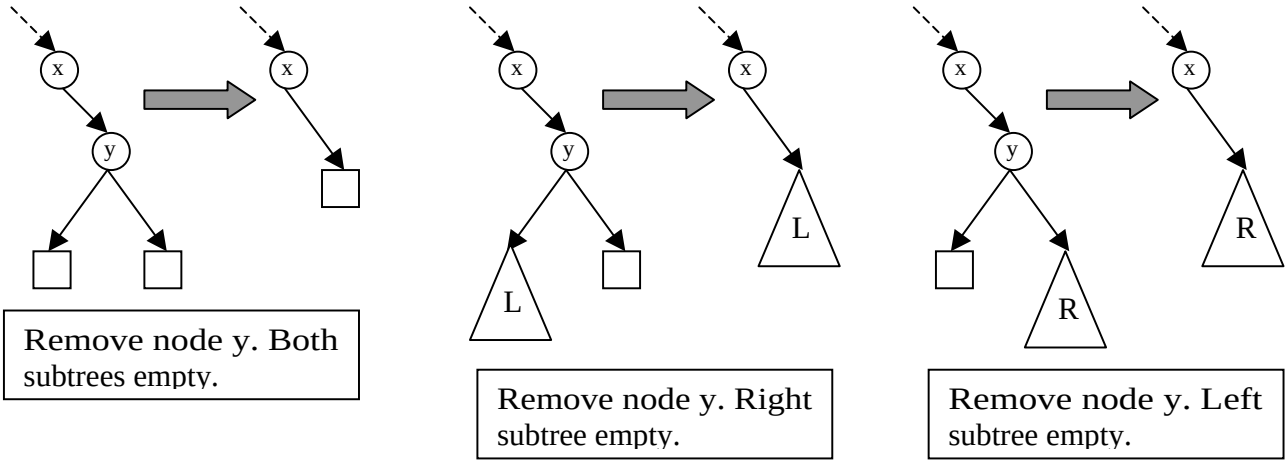
```
/**
 * Returns the node containing k in the tree rooted at node.
 */
protected BSTreeNode getFromTree(Key k, BSTreeNode node)
                            throws KeyNotFoundInTableException {
    if(node == null) {
        throw new KeyNotFoundInTableException();
    }
    else if(node.kvPair.key.equals(k)) {
        return node;
    }
    else if(node.kvPair.key.lessThan(k)) {
        return getFromTree(k, node.left);
    }
    else {
        return getFromTree(k, node.right);
    }
}
```

We start at the root of the tree and compare the search key to the key in that node. If the keys are equal, we return a reference to that node. If the key in the node is smaller, we search the left subtree, otherwise the right subtree. If the key is not in the LUT, we eventually find ourselves calling the method with an empty subtree of a node. If this occurs, we know that the key cannot be in the tree and so we throw an `Exception` to indicate that the search was unsuccessful. Try working through an

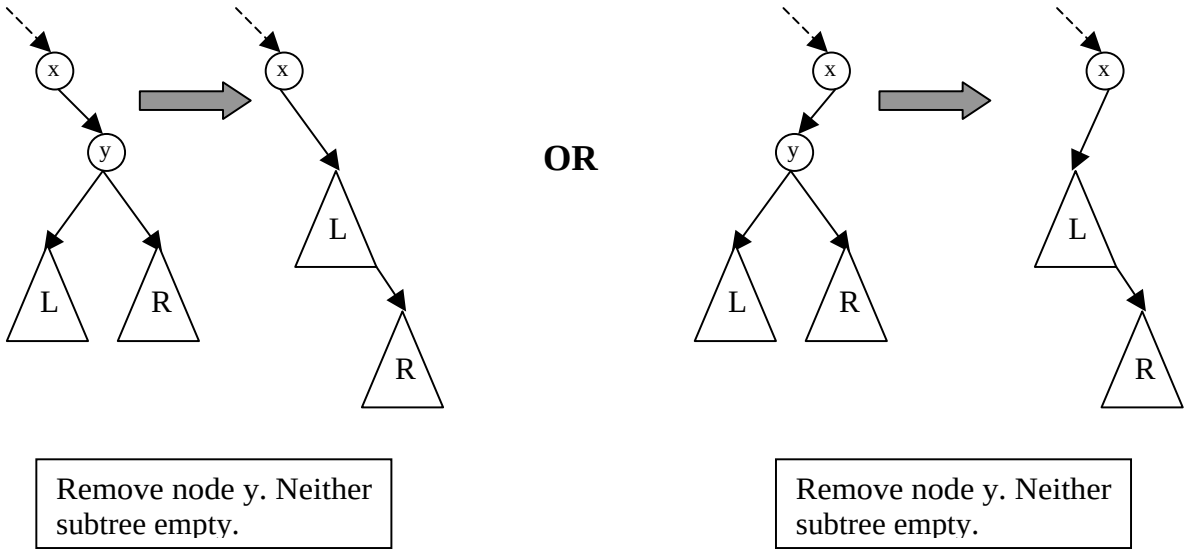
unsuccessful search by hand to understand this, for example, try searching the tree drawn at the end of the last subsection for keys: “Thomas”, “Ian” or “Adam”.

5.3.4 Deletion

Removal of elements from ordered binary trees is a little trickier, because we need to ensure that the tree is still ordered after we perform the removal. In fact, it is quite straightforward if one or both subtrees of the node are empty, as we can simply promote the non-empty subtree to the position originally occupied by the deleted node and the tree will remain ordered:



We have to think a little more carefully when both subtrees, L and R above, are non-empty. We know that all of the values in L, R and y (the right subtree of x) are less than the value in x. This observation tells us that we can merge L and R and make the combined tree the right subtree of x, but we need to know how to merge these two trees. Notice that a similar argument also holds if y is the left child of x – in that case, we know that all of the values in L, R and y are *greater* than the value in x, so L and R can be merged to form a new left subtree of x.



In order to merge trees L and R, we note further that we also know that all the values in L are greater than the value in y and that all the values in R are less than the value in y. So all the values in R are less than any value in L. Thus we can attach R to the bottom right corner of L – where a new item would be added to L if it were less than any item already in L – and be sure that the tree is still ordered correctly. This combined tree is then promoted to the position in the tree that was originally occupied by y as shown above.

We could equally well have attached the left subtree to the bottom left corner of the right subtree and promote the combined tree in the same way.

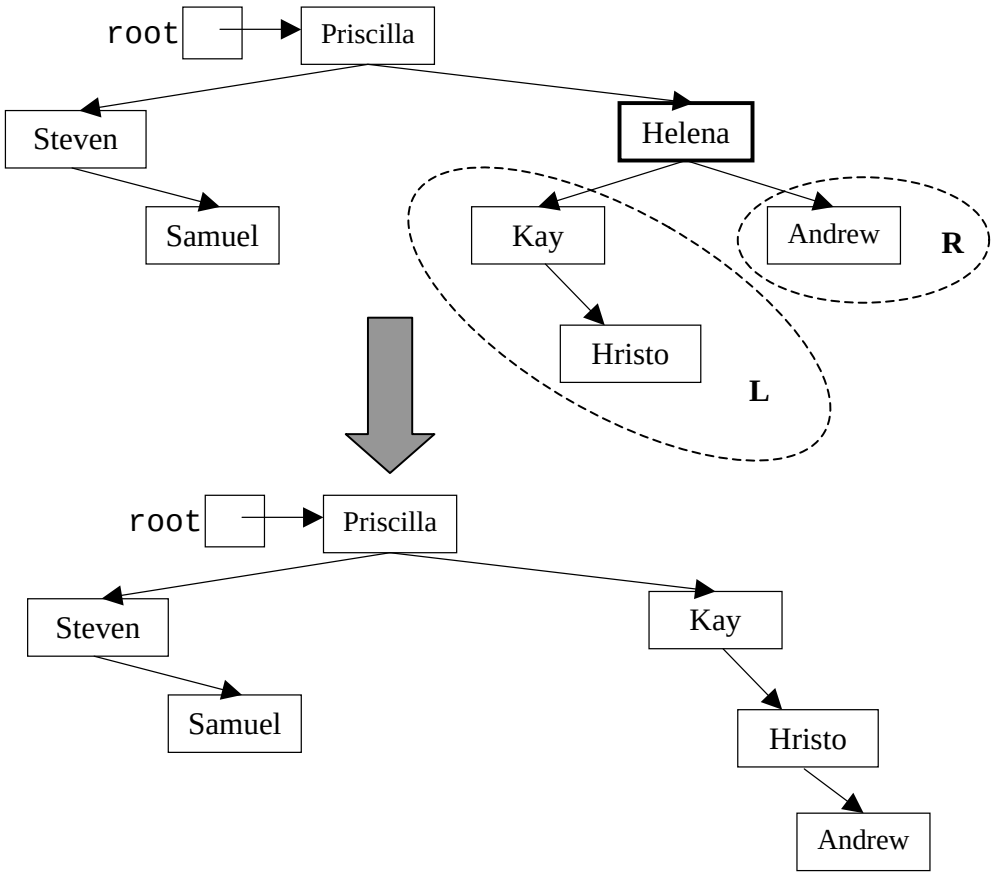
Here are the `remove` method, the recursive method `removeFromTree` that it calls, and the method `lrMerge` that merges the left and right subtrees of a given node:

```
public void remove(String key) throws KeyNotFoundInTableException
{
    Key searchKey = new Key(key);
    removeFromTree(searchKey, root);
}

/**
 * Removes then node containing k from the tree rooted at node.
 */
protected void removeFromTree(Key k, BSTreeNode node) throws
                                      KeyNotFoundInTableException {
    //Special case for empty tree
    if(node == null) {
        throw new KeyNotFoundInTableException();
    }

    //Special case for deleting root node
    else if(root.kvPair.key.equals(k)) {

           //Merge the subtrees of root and set root equal
           //to the resulting tree.
        root = lrMerge(root);
    }

    //General case
    //If the key at the current node is less than the search key
    //go to the left subtree.
    else if(node.kvPair.key.lessThan(k)) {

        //If the left subtree is empty then the key cannot be in
        //the LUT.
        if(node.left == null) {
            throw new KeyNotFoundInTableException();
        }

        //If the current node is the parent of the one being
        //removed then do the removal.
```

```
        else if(node.left.kvPair.key.equals(k)) {
            node.left = lrMerge(node.left);
        }
        //Otherwise recurse down another level.
        else {
            removeFromTree(k, node.left);
        }
    }

    //Otherwise go to the right subtree.
    else {
        //If the right subtree is empty then the key cannot be in
        //the LUT.
        if(node.right == null) {
            throw new KeyNotFoundInTableException();
        }

        //If the current node is the parent of the one being
        //removed then do the removal.
        else if(node.right.kvPair.key.equals(k)) {
            node.right = lrMerge(node.right);
        }
        //Otherwise recurse down another level.
        else {
            removeFromTree(k, node.right);
        }
    }
}

/**
 * Merges the two subtrees of node and returns a reference to
 * the root of the merged tree.
 */
protected BSTreeNode lrMerge(BSTreeNode node) {

    BSTreeNode mergedTrees = null;

    //First cases takes care of when the left only or both
    //subtrees are empty.
    if(node.left == null) {
        mergedTrees = node.right;
    }

    //Second case takes care of when right subtree only is empty.
    else if(node.right == null) {
        mergedTrees = node.left;
    }
```

```
        //General case when neither subtree is empty.
        else {
            addToTree(node.right, node.left);
            mergedTrees = node.left;
        }

        return mergedTrees;
    }
```

The recursive `removeFromTree` method uses the usual search strategy to find the *parent* of the node that is to be removed. We need to find the parent so that it's subtree link can be updated once its child is removed. This gives rise to two special cases: firstly when the tree is empty and secondly when the item to be removed is the `root`. Once the parent of the node to be removed has been located, `lrMerge` is called, which returns an ordered binary tree containing all the elements of the left and right subtrees of the node being removed. The parents link is then updated to point to this new tree and bypass the removed node, which is reclaimed by the Java garbage collector.



The example above uses the tree generated from our simple test program. The LUT entry with key "Helena" is removed from the tree. The two subtrees are merged using the scheme discussed above and the combined tree is promoted to the position originally held by the deleted node.

`lrMerge` uses the `addToTree` method in the general case when both subtrees of the node are non-empty. `addToTree` links the root node of the right subtree (first argument to `addToTree`) into the

appropriate place in the left subtree (second argument to `addToTree`). As discussed above, the appropriate place is the bottom right corner of the left subtree, however, `addToTree` will find this for us and we don't need to do the merging explicitly.

Notice that using this strategy to remove nodes from a tree can result in trees that are highly unbalanced. There are other schemes that can be used to merge the left and right subtrees, which tend to result in more balanced trees after removal. We won't go into them here, but one scheme is described in the Winder and Roberts book, chapter 15, pages 467. See also, Wiener and Pinson, Ch. 15.

5.3.5 Tree traversal – the `toString` method.

The final public interface method to look at is the `toString` method. This method is interesting, because it illustrates the general methods of traversal of a tree structure, which are commonly used and are important to understand. Traversing the tree means visiting every node in the tree and performing some operation, such as printing out the contents of that node. If we want to visit every node in a tree and perform some operation, there are three logical orders in which we can visit the nodes using recursion:

i) Pre-order

```
visit(node);
traverse(node.left);
traverse(node.right);
```

ii) In order

```
traverse(node.left);
visit(node);
traverse(node.right);
```

iii) Post-order

```
traverse(node.left);
traverse(node.right);
visit(node);
```

Each of the three outlines above is a possible (rough) implementation of the (non-existent) method `traverse`, where `visit` is some method that performs an operation on a node (e.g, prints out the value). With pre-order traversal, the root of the tree is the first node to be visited – the traversal then continues (approximately) top to bottom, left to right. With in-order, the first node visited is the left-most node and the other nodes are visited in order of their size. With post-order, the root of the tree is visited last – this goes from (approximately) bottom to top, left to right. We could also switch the order of left and right traversals to reverse the overall order of traversal. There is a nice illustration of the different visiting orders caused by the different methods of tree traversal in the Winder and Roberts book, page 451, as well as further discussion of tree traversal.

For the `toString` method, we have chosen to use an in order traversal. Here is the `toString` method and the recursive method, `treeString`, that it calls:

```
public String toString() {
    return treeString(root);
}
```

```
/**
 * Uses in-order tree traversal to construct a LUT string.
 */
protected String treeString(BSTreeNode node) {
    if(node == null) {
        return “”;
    }

    Entry lutEntry = node.kvPair;
    String thisNode = “”;
    thisNode = lutEntry.key.toString();
    thisNode += “:”;
    thisNode += lutEntry.value;
    thisNode += “, “;

    return treeString(node.left) + thisNode +
                                          treeString(node.right);
}
```

Exercise: Run the simple test program in section 4.2.2 adapted to work with a BinaryTreeLUT. Adapt the toString method so that it uses pre-order instead of in order traversal. What can you say about the order in which the items are output? Try post-order instead. Now what do you notice?

There are other orders in which we sometimes need to traverse trees, which do not fall as naturally into the recursive framework that we have been using to perform operations on trees. Another important order is *level order*, which visits the nodes in order of depth in the tree. The root node is visited first (depth 0), then it’s two children (depth 1), then each child of those two nodes (depth 2), and so on. For the tree generated by our simple test program, see page 72 or page 77, level order traversal, assuming we go from left to right at each level, would visit the nodes in the following order:

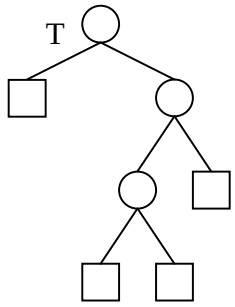
“Priscilla”, “Steven”, “Helena”, “Samuel”, “Kay”, “Andrew”, “Hristo”.

Exercise: level order traversal of a binary tree can be implemented elegantly through the use of a Queue (see Winder and Roberts). Adapt the `toString` *method to output the contents of the binary search tree in level order.*

5.4 Skew Trees and Complete Trees

The idea of a binary tree conjures up the image of a nicely balanced structure, which bifurcates at each level. However, the definition of a binary tree affords some very weird shapes. In this section we’ll look at the two extremes.

5.4.1 Skew Trees



Amongst all binary trees with n nodes, there will be some with internal path lengths that are maximum. These are called *skew trees*. They also have maximum *external* path length. A skew tree is a long thin tree, see example

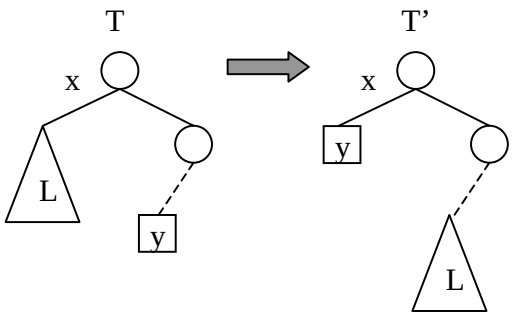
to the right, which shows a skew tree with n=3. It has *maximum height*, h(T) = n (*Exercise: prove it*).

Theorem:

A binary tree is a skew tree iff every node in it has at most one internal node amongst its children.

Sketch Proof: (see Kingston pg. 85-86 for full details)

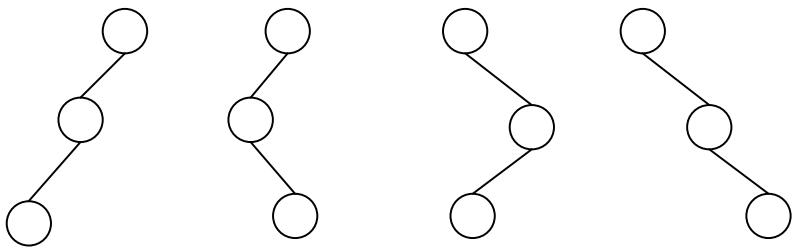
a) prove that if a node has more than one child which is internal, then it is *not* skew.



Suppose that there is a node x in a tree, T, that has non-empty left and right subtrees, L and R. Let y be an external node in R. If we exchange L and y to make a new tree T', then the depth of nodes in L is increased, which means that the internal path-length must be increased. Since i(T') > i(T), the internal path-length of T cannot have been maximal, so T is not skew.

b) prove that all trees, with the property that every node has at most one child that is internal, are skew.

Call this property P and let S_n be the set of trees of size n with property P. Here is S_3:



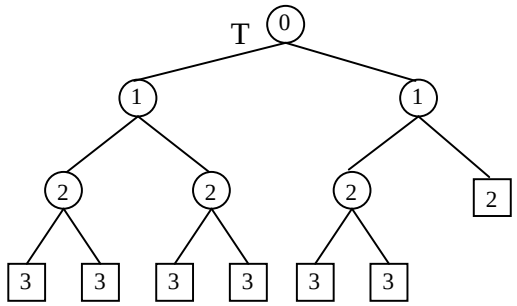
All such trees have the same internal path length:

i(T) = 0 + 1 + … + n-1 = ½n(n-1).

Since all skew trees of size n must be in S, then all trees in S have maximum internal path-length, which tells us that all trees in S are skew.

5.4.2 Complete Trees

Amongst all binary trees with n nodes there will be some with internal path lengths minimum. These are called *complete trees*. They also have minimum *external* path length. A complete tree is highly balanced. All external nodes have a depth differing by at most 1. The example below shows a complete tree, with size six, in which all of the nodes have been labelled by their depth.



Theorem:

A binary tree is a complete tree iff there is a number, q, such that every external node has depth q or q+1.

Sketch Proof: (see Kingston, pages 86-87 for details)

The proof is similar to the skew case: if external nodes x and y have depth differing by more than 1, then exchange the *parent* of the deeper node with the higher node. The resultant tree will have shorter internal path length.

Here are some other results that hold for complete trees:

1) $n = 2^k-1 \Rightarrow h(T) = k$, and $n = 2^k \Rightarrow h(T) = k + 1$. So $h(T) = \lceil \log_2(n+1) \rceil$. Note that $\lceil x \rceil$ means "ceiling(x)", which is the smallest integer larger than x.
2) $h(T) \geq d(x) \geq h(T) - 1$, when x is an external node. If we sum this over all external nodes, we get:

$$(n+1)(\lceil \log_2(n+1) \rceil - 1) \leq e(T) \leq (n+1)\lceil \log_2(n+1) \rceil$$

5.4.3 Results for Binary Trees

We can combine the results for skew and complete trees to get some general results for binary trees:

Height:

$$\lceil \log_2(n+1) \rceil \leq h(T) \leq n$$

Internal Path Length:

$$(n+1)(\lceil \log_2(n+1) \rceil - 1) - 2n \leq i(T) \leq \frac{1}{2}n(n-1)$$

External Path Length:

$$(n+1)(\lceil \log_2(n+1) \rceil - 1) \leq e(T) \leq \frac{1}{2}n(n-1) + 2n$$

The last two results use the theorem on page 66 connecting the internal and external path lengths of a tree.

5.5 Complexity of Binary Search Trees

We will take a moment to consider the complexity of retrieving items from a binary search tree. Recall that the complexity of our searching algorithms on the linear data structures, discussed in section 4, was O(n) for linear search and O(log n) for binary search, where n is the number of entries in the LUT.

For our binary search tree, with every recursion of the main retrieval method, `getFromTree`, see page 73, we move down a level of the tree being searched and two key comparisons are made:

```
node.kvPair.key.equals(k) and
node.kvPair.key.lessThan(k)
```

The worst case scenario is when the corresponding key is not found in the tree so we have to recurse all the way down to the bottom of the tree – we have to make h(T) calls to the recursive method. The worst case number of comparisons required is therefore 2h(T) and so the complexity is O(h(T)).

For a balanced, or complete tree, $h(T) = \lceil \log_2(n+1) \rceil$, as we saw in the last subsection and so the complexity of retrieval from a balanced binary search tree is O(log n), as in the case of binary search on a linear data structure. The advantage we obtain from using the tree is the truly dynamic nature of the data structure.

However, the way we construct our binary search tree, using the `insert` method, does not guarantee that the tree is complete. The worst case scenario occurs when h(T) is maximal, i.e., when the tree is skew and so h(T) = n, which gives us O(n) complexity, as in linear search on an array or linked list based LUT. In fact the search strategy is almost identical to linear search on a linked list based LUT when the tree is skew. Ironically, this worst case scenario occurs when the entries are added to the tree in key order.

There are implementations of binary trees that adjust themselves whenever an insertion or deletion operation is performed on the tree so that the tree remains balanced. These trees are called *AVL Trees*. We won't look at them here, but they are discussed in most textbooks on the subject, including the Winder and Roberts book.

It can be shown however, that if the entries are added to the binary search tree in random order, the height of the tree is O(log n) *on average*. Since the complexity of retrievals from a binary tree is O(h(n)), the complexity of retrievals is also O(log n) in the average case.

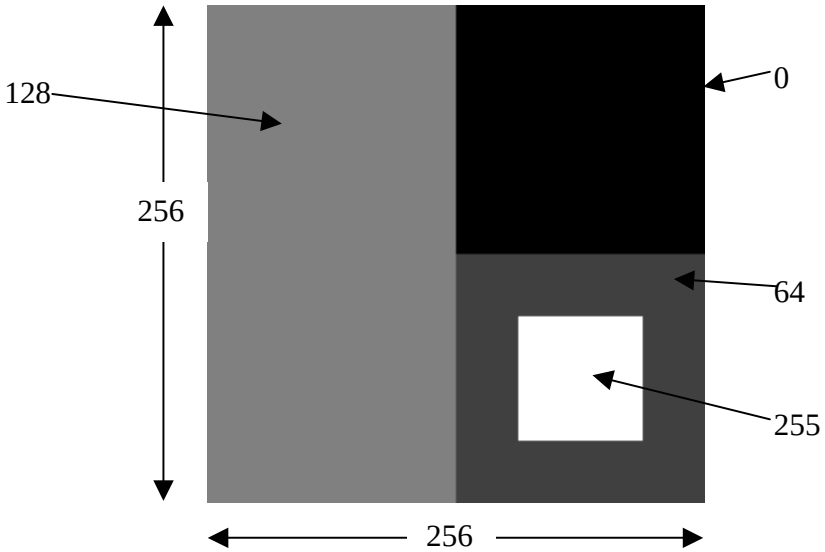
5.6 General Trees and Forests

As we saw at the beginning of this section, the binary tree is a restriction of the general definition of a tree, which allows any number of branches at each node in the tree. In this section, we will look briefly at some other restrictions of the general tree, some applications of general trees and *forests*, which are collections of trees.

5.6.1 Quadtrees and Octrees

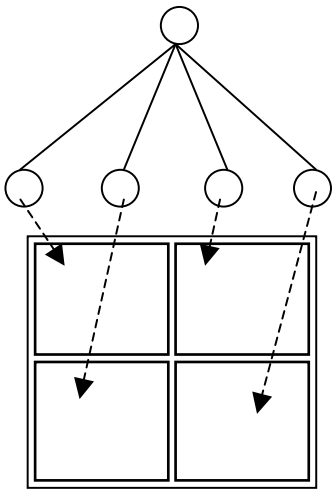
A *quadtree* is a tree that has at most *four* branches at each node. The main application of quadtrees is for storage of 2D information, in particular, images.

A digital image is made up of a large number of tiny squares called *pixels*, each of which has a particular colour or *grey-level value*. In order for an image to be convincing to the human eye, pixels have to be small enough that we cannot see the transition between them. In order to make up a whole image, we need large numbers of pixels, each containing and value that must be stored. In *grey-scale* images, each pixel contains a value between 0 and 255, which represents the brightness of that pixel: 0 represents black, 255 represents white and the values in between represent increasingly light shades of grey. Here is a simple grey-scale image, which is described by a grid of 256x256 pixels and thus



requires 65536 bytes of storage:

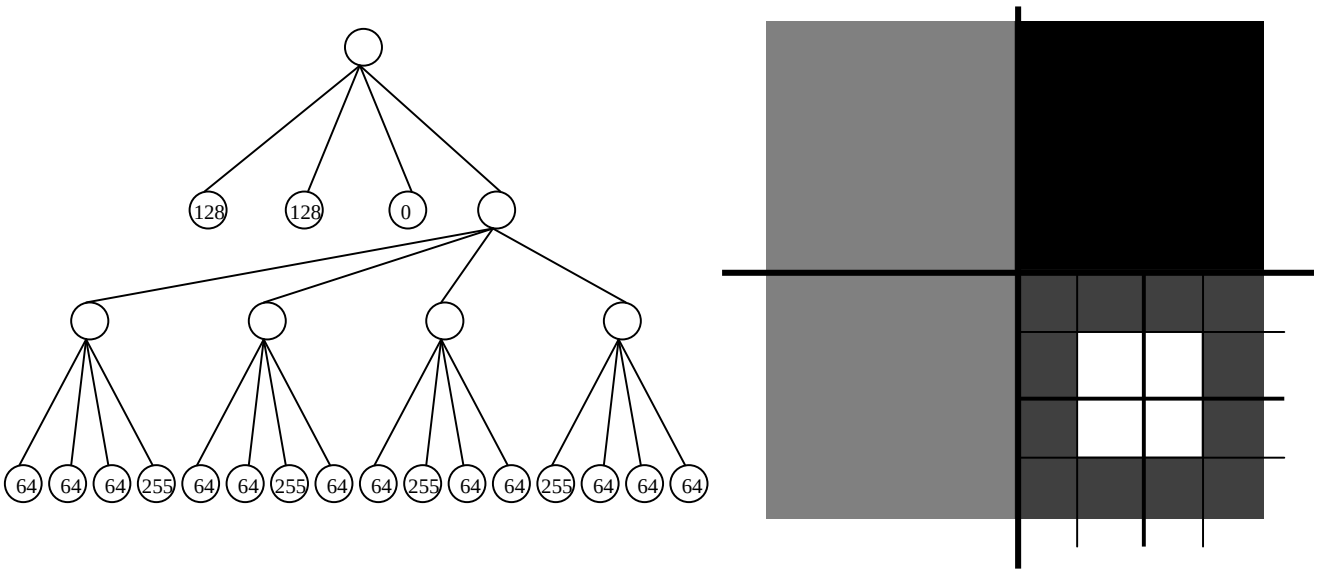
A quadtree provides an efficient way to store images like this. It exploits that fact that images (particularly artificially generated images like the one above) tend to consist of large regions of the same colour. The image above has four distinct regions, which contain pixels of the same grey-level value indicated above. At each level of the tree, the image is divided into four quarters each represented by a child node in the quadtree. We keep dividing the image until all the pixels contained in the image region represented by a node have the same grey-level value.



So the root node of the quadtree represents the entire image. The four children of the root node, represent the four quarters of the image, as shown to the left. The children of these nodes represent the four quarters of each of these regions and so on.

To construct a quadtree representation of an image, the image is divided in this way until regions contain pixels that are all of the same colour. That region can then be represented by a single grey-level value and no further subdivision has to occur.

The diagram below shows the quadtree that represents the example image above and shows the corresponding division of the image that the quadtree represents. Only the leaf nodes in the quadtree need to contain values and they contain the grey-level value of the pixels in the region of the image they represent.



So a tree with 25 nodes is required to store this image. Each node probably requires 17 bytes of storage: four for each object reference and one for the grey-level value, which means the total number of bytes to store the image is reduced from 65536 to 17 x 25 = 425 – a significant saving. The quadtree representation is less effective for images where the values in pixels vary a lot, for example, in real photographs rather than synthetic images generated from computer drawing programs. *Why?*

An *octree* is a similar idea, but with a maximum of 8 children at each node. Octrees are used in a similar way to quadtrees, but are used to store 3D information – a 3D volume of data is divided into 8 *octants* at each successive level of the tree. Octrees also arise frequently in computer graphics and image processing applications.

5.6.2 General Trees

Tree-like data structures arise in many computer applications besides searching: any hierarchy is equivalent to a tree. Examples include the departmental or managerial structure of a large conventional business organisation, a family tree, or the relationships between derived classes in a well structured large object oriented system. In most cases, we cannot assert that trees are binary, or even quad or octrees. There might be more children than any limit that we set and, usually, we will want to accommodate variable numbers of subtrees for different nodes in the tree.

One way that we might extend the data structure of a binary tree to hold general trees is to increase the number of links in the tree node class. Rather than having two link fields, `left` and `right`, in the class, we could have an array of k pointers to other nodes. This is a bad idea, however, firstly because we need to choose a value for k and secondly because most of the pointer fields in such a tree will be `null` and thus be wasted space. There is a simple proof of the latter assertion:

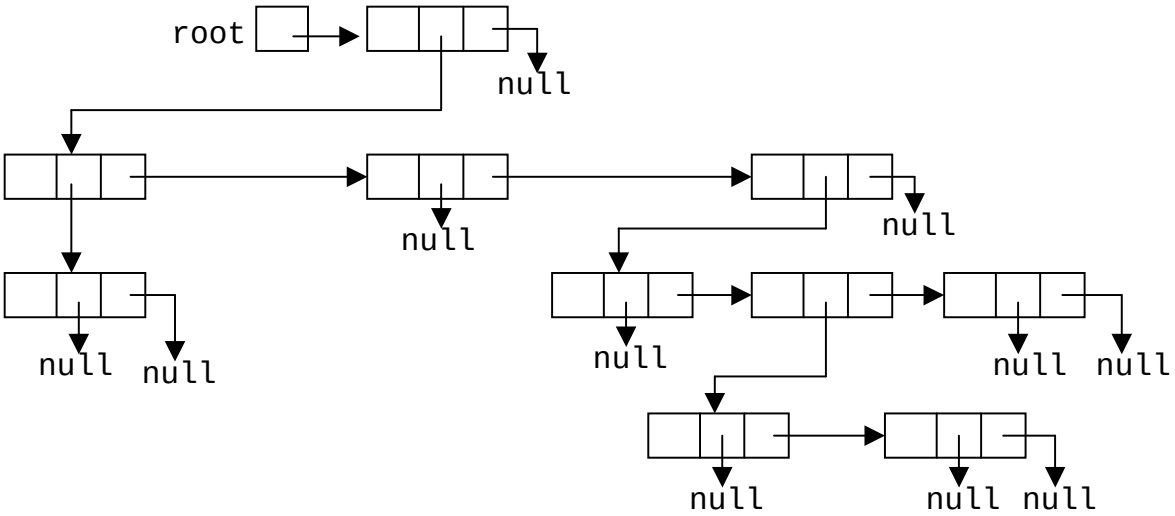
Suppose there are n nodes in a tree. Each node, except the root, has exactly one ancestor and so each node except the ancestor has exactly one link field pointing to it. Thus there are n-1 nodes being pointed to, so n-1 link fields are not `null`. Each node has k link fields and so there are a total of nk

link fields in the whole tree. So nk – (n–1) link fields must be `null`. The proportion of link fields that are `null` is then:

$$\frac{nk-(n-1)}{nk}=1-\frac{n-1}{nk}>1-\frac{1}{k}$$

So for k = 4 (the quadtree), about 75% of the links are wasted space. For k = 10, it is about 90%. Notice that even for the binary tree, 50% of the link fields will be `null`.

A better idea is to use a linked list to store all the links to the descendants of a node. Each node contains a pointer to its leftmost descendant, which has a pointer to its next sibling, etc. Here is a diagram of such a data structure for the general tree drawn at the beginning of section 5 – page 64.



Notice that the nodes in this tree data structure are identical to the nodes in a binary tree: there is a value and two link fields, which reference other nodes. This tree is structurally equivalent to a binary tree. In fact for every general tree there exists a corresponding binary tree. Notice that the right subtree of the root node in the corresponding binary tree is empty and this will always be the case for a binary tree that corresponds to a general tree, which implies that the converse does not hold.

A neat variation of this data structure is to replace the null pointer at the end of a list of siblings to point back to the parent node.

5.6.3 Forests

A *forest* is a collection of trees that need to be considered together as a single entity. The descendants of any node in a tree constitute a forest. One context in which forests arise is when trees are constructed from the leaves up. Parse trees are an example of this, see also Huffman trees, which are used for encoding data in a compact representation – Aho, Hopcroft and Ullman, “Data Structures and Algorithms”, chapter 3, pages 94-102.

In the previous section, we noticed that the data structure for a general tree could equally represent a binary tree. The converse is not true, however, because general trees are only equivalent to binary trees with empty right subtree. A binary tree with non-empty right subtree is equivalent to a forest.

5.6.4 Traversal of Trees and Forests

As for binary trees, a common requirement of general tree structures is to perform some operation at every node in the structure. For a general tree or forest, $\{T_1, T_2, \ldots, T_n\}$, there is no natural analogue of the in-order traversal strategy that we had for binary trees – where do you fit the root between its descendants? We can define pre-order and post-order, however:

i) *Pre-order:*

```
visit(root(T1));
traverse(children(T1));
traverse(T2, T3, …, Tn);
```

ii) *Post-order:*

```
traverse(children(T1));
visit(root(T1));
traverse(T2, T3, …, Tn);
```

So for a tree, we either visit nodes and then traverse their subtrees (pre-order) or traverse subtrees and then visit the node (post-order). For forests, we do the same for each tree in turn. Notice that we can also define level order for trees or forests.

6 Hashing

In the last two sections, we have seen various methods of implementing LUT search by key comparison and we can compare the complexity of these approaches:

Search Strategy	A(n)	W(n)
Linear search	O(n)	O(n)
Binary Search	$O(\log_2 n)$	$O(\log_2 n)$
Binary Tree Search	$O(\log_2 n)$	O(n)

However, key comparison isn't the only way to do table look-up. *Hash tables* are LUTs that potentially offer O(1) complexity, like array indexing; unfortunately, the worst case is still O(n).

Consider first a special case. If we know that all our keys are integers in the range [0, …M-1] for some reasonably small value of M, then we could store the entry with key k at position k in an array of length M. In order to retrieve the value associated with key k, we just go to position k in the array and take out the value. Retrieval is done in a single step: it is O(1). Here is some code that might appear in the body of a LUT class implementing this strategy:

```
public void insert(int key, Object value) {
    objectArray[key] = value;
}

Object retrieve(int key) {
    return objectArray(key);
}

private Object[] objectArray = new Object[M];
```

This presents two significant advantages over the LUTs we have seen so far:

- Space complexity is O(M).
- Time complexity is O(1).

Of course, the first point might not be an advantage if the number of entries is small compared to the range of key values. Consider the payroll numbers in the `Employee` class, what if we have 676 `Employee`'s, but the largest payroll number is greater than 1100000?

Hashing, also known as *key to address translation* or *scatter-storage*, is a generalisation of this approach, where the key is used to compute the address of a place to store a LUT entry. In general, the range of possible key values may be very large, or even infinite, for example, the set of character strings. In such cases, it is not possible to make an array large enough to hold all possible key values. In addition, we probably only require a small subset of all the possible values – i.e., the table is *sparse*. The solution is to introduce a *hash function*.

A hash function, h, is a mapping from the set of all possible keys to the range of indexes of an array, h(k): {all possible keys} → {0, 1, … M-1}. We can use this hash function to compute an array location at which to store any given value. Given a {key, value} pair, we set

```
objectArray[h(key)] = value;
```

i.e., given a key, k, compute a value h(k), such that $0 \leq h(k) < M$, for a table of size M. Retrieval from such a table is independent of the size of the table. Given a key, k, whose corresponding value we want to retrieve, we compute h(k) and return the value stored at index h(k) of our array.

6.1 Choice of Hash Function

In general, the number of possible keys will be much larger that the number of slots in the array, so it is impossible to devise a function h that computes unique values for every k: h is a many-to-one function. Instead, we adopt a strategy of allowing $h(k_1) = h(k_2)$ sometimes, for $k_1 \neq k_2$. This means that we also need to devise a means of dealing with the resulting *collisions*, which occur when the hash function produces the same value for two different keys.

- In the absence of collisions, complexity is O(1).
- In the presence of collisions, performance must inevitably degrade, but complexity depends on how the collisions are handled.

We would like to minimise the occurrence of collisions, since the performance in their absence is so good. So the implementation of a hash table calls for two choices:

- Choice of a suitable hash function, that *minimises collisions* and is *simple and efficient to compute*.
- Choice of a *collision resolution strategy*.

Let the size of the table be M. We require $0 \leq h(k) < M$ for all possible k. This requires that the range of h(k) is an integer between 0 and M-1. The domain of h(k) can be anything in general, but most commonly, k will be a character string, so we need to map strings of characters to integers. This can be done conveniently by using the internal character codes of the characters in the string. In Java, we can get these codes by casting `char` to `int` (a procedure that is not often recommended):

```
char a = 'S';
int b = (int)a;
System.out.println(a + ":" + b);
```

The above program prints out:

```
S:83
```

83 is the UNICODE value of the character '`S`'.

A character string of length n is thus equivalent to an 8n-bit binary number, since each character is represented by a byte – an 8-bit binary number. The job of our hashing function, h, is to cut down the range of values of such a big number.

The distribution of values of h(k) must be well spread in the range 0 to M-1. Without knowledge of the domain of k, it is difficult to predict how well a particular function will perform; the choice is governed by a mixture of some analysis, experiment and experience. A general principal that seems to make

sense is to choose h so that all the digits/characters of k contribute to the value computed. A popular choice is some generalisation of:

```
public static int h1(String key, int M) {
    int n = 0;
    for(int i=0; i<key.length(); i++) {
        n += (int)k.charAt(i);
    }
    return n%M;
}
```

So we add up the UNICODEs of all the characters in the string and take the remainder modulo M, which ensures that we end up with a value in the correct range: 0 to M-1.

This hashing function has the obvious disadvantage that all anagrams hash to the same value. A better choice is to take the full representation of `k` as a large integer, `y`, and return `(y % M)`. Here is the Java method:

```
public static int h2(String key, int M) {
    int factor = 1;
    int n = 0;
    for(int i=key.length()–1; i>=0; i--) {
        n += factor*(int)key.charAt(i);
        factor *= 256;
    }
    return n%M;
}
```

The binary representation of this large integer is obtained by concatenating the bit patterns representing each character in the string. Suppose we have key, “AKEY”. Here are the UNICODEs for these characters in decimal and binary:

$A = 65_{10} = 01000001_2$
$K = 75_{10} = 01001011_2$
$E = 69_{10} = 01000101_2$
$Y = 89_{10} = 01011001_2$

The subscripts on the numbers indicate the *base* of the representation of the number, so a subscript of 10 indicates a decimal number and a subscript of 2 indicates a binary number. The large integer `y` representing “AKEY” is:

$y = 01000001010010110100010101011001_2$
$= \text{int(A)*256*256*256 + int(K)*256)*256 + int(E)*256 + int(Y)}$
$= 65_{10}*256*256*256 + 75_{10}*256*256 + 69_{10}*256 + 89_{10}$
$= 1095451993_{10}$

Suppose we choose a table of size 101, then `(y % M)` gives a hash value of 34. With this method, anagrams are not hashed to the same value, but we still get collisions:

```
h1(“AKEY”) = 96,          h2(“AKEY”) = 34
h1(“YEAK”) = 96,          h2(“YEAK”) = 2
h1(“EAYK”) = 96,          h2(“EAYK”) = 38
```

```
h1("KZFO") = 11,          h2("KZFO") = 34
```

Exercise: What happens to the performance of h1 and h2 when the size of the table, $M=2^n$?

A problem with this approach is that we very quickly reach the limits of accuracy of the primitive type `int`. Four bytes are used to represent an `int` value and so the longest `String` that we can convert to an `int` in this way has only four characters. We could use the primitive type `long` instead, but this also has limited range – in fact a `long` is represented by 8 bytes – so we have similar limitations. Suppose we want to find the hash value of "VERYLONGKEY". The corresponding large integer `y` is well out of range of both int's and long's. The value we are after is equal to:

$$(86\times256^{10} + 69\times256^{9} + 82\times256^{8} + 89\times256^{7} + 76\times256^{6} + 79\times256^{5} + 78\times256^{4} + 71\times256^{3} + 74\times256^{2} + 69\times256^{1} + 89) \% M$$

In order to compute this value, we don't have to compute the very large integer value in the brackets explicitly. Instead we can exploit the multiplicative and additive properties of the % operator (Horner's method):

$(a\times b + c) \% M = ((a \% M)\times b + c) \% M$

Exercise: prove it.

This leads to the following algorithm for computing the hash value of a `String` of arbitrary length:

```
public static int h3(String key, int M) {
    int n = 0;
    for(int i=0; i<key.length(); i++) {
        n = (n*256 + (int)key.charAt(i))%M;
    }
    return n;
}
```

This scheme correctly calculates the hash value of "VERYLONGKEY" as 79, for M = 101.

There are many ways to construct hash functions. Another popular approach is to add the characters in blocks of four, i.e., treat them as 32-bit words. Each of these words is squared and the middle digit(s) are extracted. The middle digits of each squared 32-bit word are concatenated to make a large integer which provides a hash value by taking the remainder modulo M. This technique is not very good for short keys, but it takes away the property of h_1 that anagrams all hash to the same value.

6.2 Table Size

We want the table size to be about the same as the expected number of entries – M~n, so that there is not too much wasted space and there is not an excessive number of collisions.

To work effectively, a general rule of thumb is that M be *prime* and *not too close to a power of 2* (see *Exercise* in the previous section). 1009 is said to be a good number (and far enough away from 1024).

6.3 Object.hashCode()

The idea of a hash table is so important that a special method, `hashCode`, has been included in the general `Object` class – the root of the Java class hierarchy. For general objects, the `hashCode` is computed by looking at the memory address at which the object is stored in the computer – so be aware that these values are consistent within one execution of a program, but not beyond that. The method is overridden in many classes, however, such as `String`, `Integer`, `Float`, etc, so that the hash code is a function of the actual data in the object. You can override this method in any class that you construct, which you want to use as keys in a LUT.

The hash function used for the `String` class in Java is different to any of the ones we've seen so far. It is detailed in the on-line Java documentation.

6.4 Collision Resolution

No matter how carefully the hash function is chosen, there will always be collisions. In this section, we'll look at some ways of handling collisions.

6.4.1 Resolution by Overflow Chaining

The easiest way to deal with collisions is to allow each slot in the table to hold more than one entry. This we can do easily, by making each slot in the table be a reference to a linked list of actual entries. Look up is done by computing h(k) and then doing a linear search in the list stored in the corresponding index of the array. If h is poorly chosen, this will lead to long lists in some locations while other array slots might remain empty. In the worst case, all of the entries get stored in a list at a single slot in the array and retrieval degenerates to linear search: performance is O(n). Ideally, the results should be more like the example shown below with lots of short lists and many entries stored in lists of their own, "singletons".

Suppose we want to use a hash table to store employee records of the academic staff in the department. The keys of the entries in our LUT are the surnames of the staff. We'll just use twelve examples and try to store them in a hash table of size 13. We will use hash function h_3 from section 6.1. Here are the 12 surnames (keys) and their hash values:

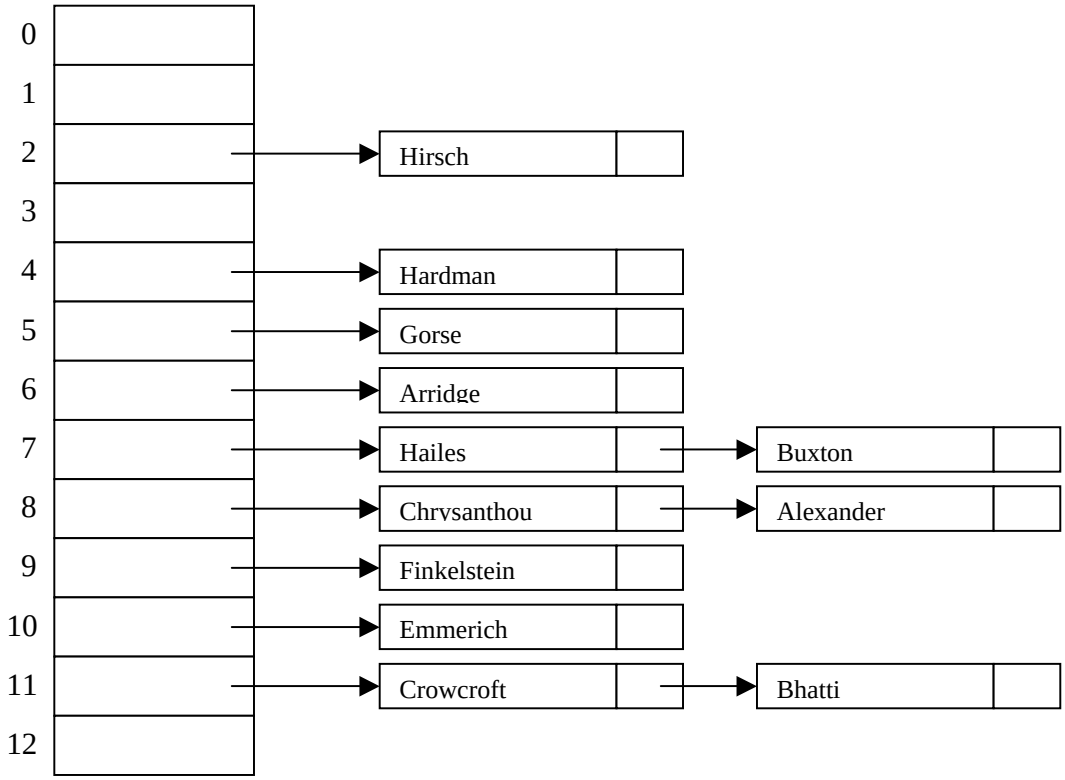
Alexander	8
Arridge	6
Bhatti	11

Emmerich	10
Finkelstein	9
Gorse	5

Buxton	7
Chrysanthou	8
Crowcroft	11

Hailes	7
Hardman	4
Hirsch	2

The data structure holding the hash table that contains these keys using h_3 as the hash function and collision resolution by chaining looks like this:



We will look briefly at an implementation of a LUT that uses hashing in this way. Here is an outline of the class:

```
public class HashTableChain {

    protected class EntryNode {
        protected String key;
        protected Object value;
        protected EntryNode next;

        public EntryNode(String k, Object v, EntryNode n) {
            key = k;
            value = v;
            next = n;
        }
```

```
    }

    //Protected data member holding array of pointers to chains
    protected EntryNode[] entryArray;

    public HashTableChain() {
    entryArray = new EntryNode[101];
    }

    public HashTableChain(int size) {
        entryArray = new EntryNode[size];
    }

    //LUT public interface methods
    //insert, retrieve, update, delete, toString
    //:
}
```

The table is stored in an array of `EntryNode` objects which contain a key, a value and a link to another `EntryNode`, so that these objects can be linked together in a list. `EntryNode` is a member class of the `HashTableChain` class. Here are the `insert` and `retrieve` methods, the other LUT public interface methods: `update`, `remove` and `toString` are omitted, but are similar.

```
public void insert(String key, Object value) {

    //Compute hash value
    int index = hash(key, entryArray.length);

    //Create new list node and link in at the start of the list
    //at the computed index.
    entryArray[index] = new EntryNode(key, value,
                                          entryArray[index]);
}

public Object retrieve(String key)
                          throws KeyNotFoundInTableException {

    //Compute hash value
    int index = hash(key, entryArray.length);

    //Use linear search to chain down list looking for match
    EntryNode entry = entryArray[index];
    while(entry != null) {
        if(entry.key.equals(key)) {
            return entry.value;
        }
        else {
            entry = entry.next;
```

```
            }
        }
        throw new KeyNotFoundInTableException();
    }
```

Notice that we could use the heuristic techniques, move-to-front and migrate-to-front, described in section 4.2.4, which improve the performance of linear search on the linked lists referenced by each entry of the array.

From the diagram above we can observe that there is quite a lot of wasted space when this collision resolution strategy is adopted. Although they are not marked on the diagram, there is a `null` reference at the end of each list – one for each slot in the array. It is well observed that the number of collisions increase as the table gets more full. If we saved the space for link pointers and stored pointers to the table entries directly in the array we could use an array twice as big, which would be less likely to fill up and so should perform better. But then we need an alternative collision resolution strategy.

6.4.2 Open Addressing (sometimes called “closed hashing”)

Open addressing schemes work by attempting to store at the hashed address, and if it is already occupied, *probing* the array to find another empty slot.

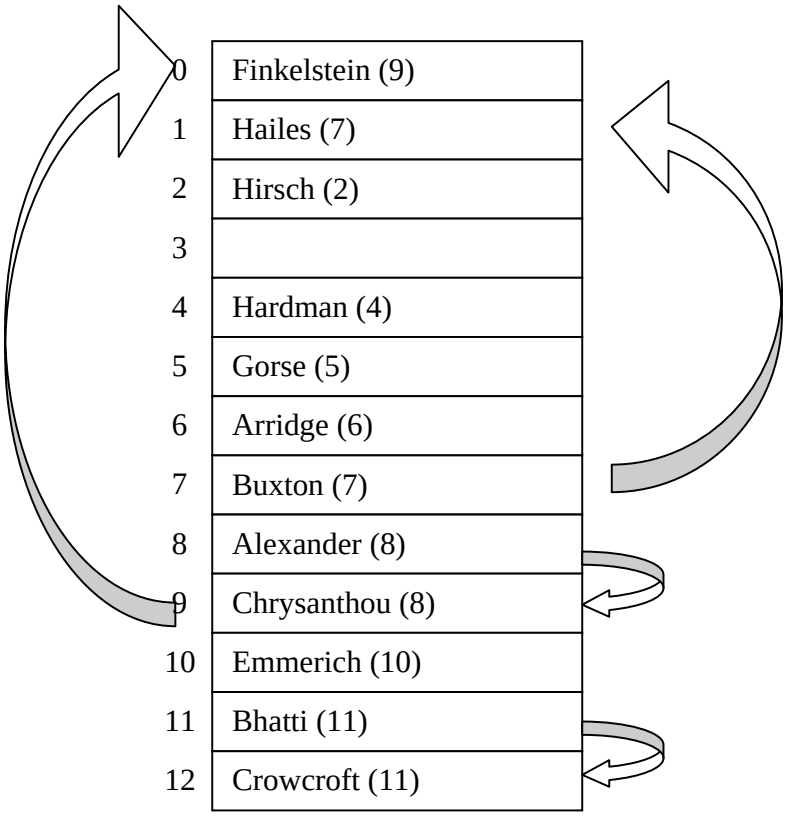
Linear Probing looks at the subsequent slots in the array. If the slot indexed by the hash value of a new key is already occupied, we try to put the new entry in the next slot of the array: `(index+1)%M`. If that slot is also occupied, we try the next one, until an empty location is found. Unlike overflow chaining, this scheme affords the possibility that the table can overflow and we need to check for this in the `insert` method of a hash table class that uses this collision resolution strategy.

Here is an outline of a hash table class that uses linear probing.

```
public class HashTableLProbe {

    protected class Entry {
        protected String key;
        protected Object value;

        public Entry(String k, Object v) {
            key = k;
            value = v;
        }
    }

    //Protected data member holding array of Entry's
    protected Entry[] entryArray;

    public HashTableLProbe() {
        entryArray = new Entry[101];
    }

    public HashTableLProbe(int size) {
```

```
            entryArray = new Entry[size];
        }

        //Public interface methods.
        //:
        //:
    }
```

Suppose we use a table of size 13 and insert the same entries we used previously – the academic staff records indexed by surnames. Here is the table containing all those entries:



A number of collisions occur and some entries in the table are moved some distance from their hashed location – indicated by the arrows above. For example, “Hailes” has hash value 7, but by the time we insert this entry, slots 7-12 and 0 are all occupied and so the first available slot has index 1. Since the table is nearly full, we can expect that some values will be stored well away from their hash positions, however, as we have saved some storage by not using linked lists, we can afford to make the table bigger.

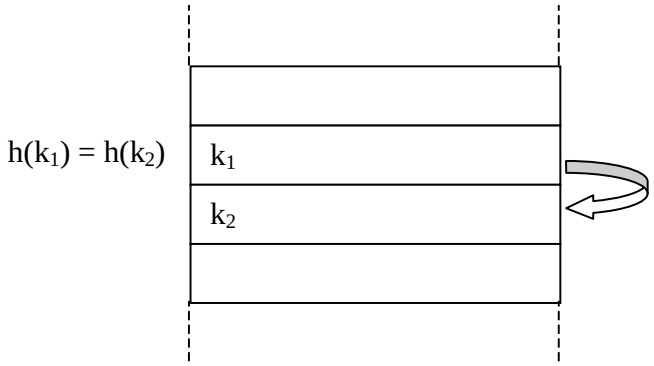
Exercise: draw the hash table of size 19 that contains the above entries (note that the hash values will be different) and uses linear probing. What is the greatest distance between hashed index and actual location of all the entries?

6.4.3 Deletion and Tombstones

We need to be careful when we remove entries from a table that uses open addressing that the removal doesn't result in entries that have been offset from their hashed position being lost.

In the `retrieve` method, we do not want to have to search the whole table for keys, as performance would be very poor for unsuccessful searches. We would like to be able to check the hashed position of a search key and, if it is empty, then we know that the search has failed. If the hashed position is occupied, but the key is not equal to our search key, then we probe until the key is found or an empty location is found, which indicates that the search has failed.

The problem is the following. Suppose we have two different keys k_1 and k_2, such that $h(k_1) = h(k_2)$. If we insert entries with these keys in the table, the second entry will be offset by one position:



Now suppose we remove the entry containing k_1 from the table. Slot $h(k_2)$ of the array is set to `null`. When we try and search for k_2, we go to slot $h(k_2)$ and find that it is empty indicating that the search has failed, so the entry will never be found. The answer is to use a *tombstone*. A tombstone is a dummy entry with a distinctive key (e.g. "Tombstone"), which can be replaced by the `insert` method, but tells the `retrieve` (and other) methods that a deletion has occurred at a particular slot in the array and so probing still needs to be performed.

6.4.4 Quadratic Probing and Double Hashing

One problem with linear probing is that the entries tend to cluster around positions in the table. There can be parts of the table where bunches of entries are stored and collisions cause some entries to be far away from their hashed index, whilst there are other parts of the table that are sparsely populated. An alternative to linear probing is to use *quadratic probing*. Rather than looking for empty slots in positions h(k), h(k)+1, h(k)+2, h(k)+3, etc, the offset is squared so we look in positions h(k), h(k)+1, h(k)+4, h(k)+9, etc. This helps minimise the clustering effect.

Double hashing generalises linear and quadratic probing by using a *second* hash function to get a fixed increment to use for the probe sequence. The offset is some function of the original hash value:

```
index = hash(key);
while(entryArray[index] == null) {
    index = (index + hash2(index))%M;
```

```
}
```

We need to keep a count to ensure that we don't probe forever in the above loop and also to check whether the table is full.

Some basic requirements for the second hash function are:

- `hash2(index)` is never 0 – or we enter an infinite loop.
- M and `hash2(index)` must be relatively prime – if `hash2(index)` = M/2, we won't get very far. This requirement is generally met if M is prime and `hash2(index)` < M.

Here is a simple example of a second hash function:

```
hash2(index) = 8 – (index%8);  //Uses last three bits of index.
```

Notice that the use of quadratic probing and double hashing in general makes it difficult to detect table overflow in the way implemented in the previous section. A simple solution is to store the table size in a separate instance variable of the hash table class and increment and decrement it in the `insert` and `remove` methods.

6.5 Performance of Hash Tables

Empirically, the performance of open addressing degenerates rapidly as the table fills up. This is because collisions breed collisions, which leads to runs of occupied slots. When h(k) is anywhere within a run, the entry for k gets sent to the end of the run, making it even longer. As the table approaches fullness, adjacent runs coalesce leading to a catastrophic loss of performance – you can see this in the example in section 6.4.2 above.

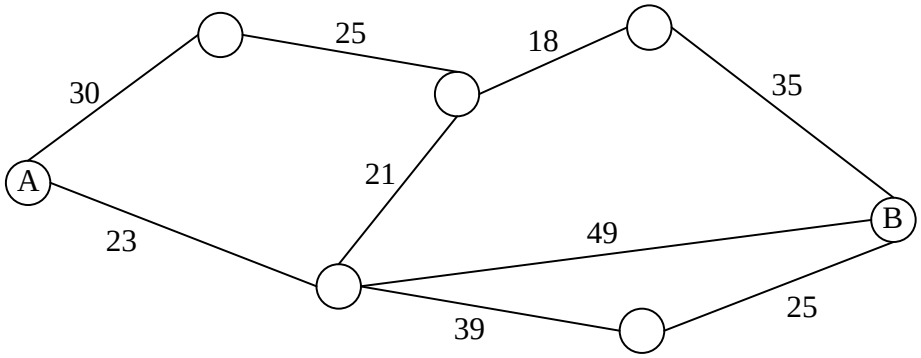
Since there is no use of linked lists in a hash table with open addressing, the table can always fill up. Performance considerations suggest that you should never let the array get entirely full, but should expand it when it gets, say 85-90% full. To expand the table, a bigger array must be allocated and all the entries in the old array must be inserted in the new array – you can't just copy the old array, because h(k) depends on M.

Open addressing uses less than five probes on average for a hash table that is less than 2/3 full – see Sedgewick, pages 236-238, for more details and proofs of this.

7 Directed Graphs

Graphs or, more specifically, *directed graphs* are generalisations of the tree data structure. They represent a large number of real world problems – in particular, those that are concerned with networks of interconnected objects. For example:

- Minimisation of costs of communications networks.
- Designing the layout of connections on a circuit board.
- Generating efficient assembly code for evaluating expressions.
- Flowcharts.
- Road or rail maps.



Suppose we want to find the shortest route between two towns on a map. We can draw a graph, such as the one above, in which the nodes (or *vertices*) of the graph represent towns and the edges (or *arcs*) represent roads between them. We can label each edge with the length or the road – the distance between towns. A classic graph theoretic problem is to find the shortest path between two nodes, say A and B, which could represent the shortest distance by road from town A to town B. It might equally represent the cheapest flight from airport A to airport B if each node represents an airport, the edges are the available flights, and the edge labels are the costs of those flights.

There are many special forms of graph and many algorithms for solving particular problems on them. Many of these algorithms have very complex proofs of correctness. Many problems involving graphs remain unsolved or only have very inefficient solutions. They form the core of "Theoretical Computer Science". We will only scratch the surface here and look at:

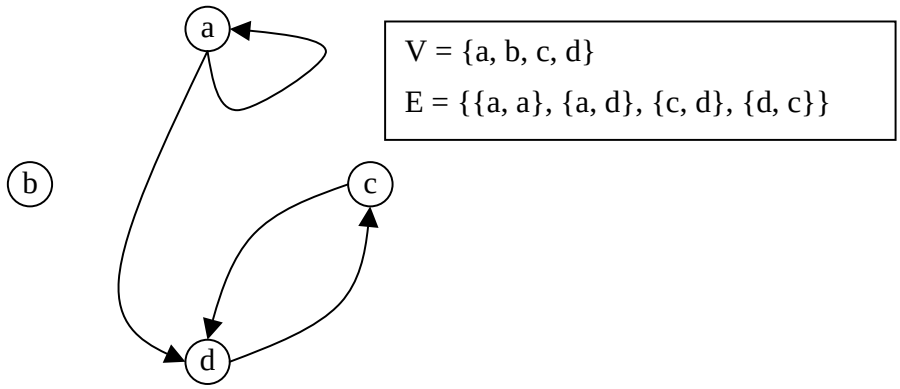
- some definitions concerning graphs and their properties,
- some ways to implement graphs, and
- the solution to some simple problems involving graphs.

7.1 Definitions

- A directed graph (or *digraph*, or just graph) is a set of vertices, V, together with a set of ordered pairs, E, of edges. Thus we write that a graph, G = <V, E>.

- Each edge consists of two vertices in V and is represented diagrammatically by an arrow from the first vertex to the second.

Here is an example of a graph with four vertices in V and four edges in E:



This definition permits self-loops, i.e., edges of the form {v, v}, that begin and end at the same place. Parallel edges, i.e., two identical edges in E, are prohibited however.

7.1.1 Vertices, Edges and Labels

- If $\{v, w\} \in E$, then w is a *successor* of v and v is a *predecessor* of w.
- The *in-degree* of vertex w is the number of predecessors that w has.
- The *out-degree* is the number of successors.

In the graph above, the in-degree of vertex a is 1, while its out-degree is 2. The in-degree of d is 2, while its out-degree is 1.

- A *labelled* digraph is a digraph G = <V, E> together with a pair of functions:

$L_V: V \rightarrow$ vertex labels

$L_E: E \rightarrow$ edge labels

So each vertex and each edge has some value associated with it, which is known as its label. The values of labels can be of any type.

- Labels on vertices are usually names (strings of characters).
- Labels on edges are usually some sort of cost or weight, e.g., the distances on a road map, which have a numerical value.

7.1.2 Paths and Cycles

- A *path* is a sequence of vertices $(v_1, v_2, \ldots, v_k)$, $k \geq 1$, such that $\{v_i, v_{i+1}\} \in E$ for all $1 \leq i < k$.
- A *proper path* is a path for which $k \geq 2$.

- If (v_1, v_2, …, v_k) is a path, then we say that v_k is *reachable* from v_1. Every vertex is reachable from itself.
- Two vertices v and w are *strongly connected* if w is reachable from v *and* v is reachable from w.
- A path (v_1, v_2, …, v_k) is a *cycle* if $v_1 = v_k$.
- A path (v_1, v_2, …, v_k) is *simple* if all vertices, v_1, …, v_{k-1} are distinct. v_1 can be equal (or not equal) to v_k. If $v_1 = v_k$ then the path is a *simple cycle*.
- The *length* of a path is the number of edges it has.

The graph shown above has the following set of simple cycles:

{a}, {a, a}, {c, d, c}, {b}, {c}, {d}, and {d, c, d}.

The proper, simple cycles are:

{a, a}, {c, d, c} and {d, c, d}.

The only strongly connected pair of vertices is c and d.

7.1.3 Subgraphs and Trees

We can think of a tree as a special type of graph <V, E> with a distinguished vertex called the *root*, such that for every vertex $v \in V$ there is exactly one path (root, …, v). This is equivalent to saying that each vertex has only one predecessor (c.f., parent in tree language).

- A *subgraph* of a directed graph G = <V, E> is a graph G' = <V', E'> such that $V' \subset V$ and $E' \subset E$.
- A *spanning tree* of a digraph G = <V, E> is a subgraph T = <V', E'> of G such that T is a rooted tree and V' = V.

Not every graph has a spanning tree – the graph used as an example above doesn't for example, since there are no paths between vertex b and the other vertices.

7.1.4 Undirected Graphs

Undirected graphs differ from directed graphs in that every edge connects two vertices in both directions. An undirected graph can be defined in two ways:

1. A new object, like a graph, but with *unordered* edges. In-degree and out-degree of vertices are replaced by just *degree*, which is the number of edges that have a particular vertex as an endpoint.

The road map example at the start of section 7 is an undirected graph (the roads are assumed to be 2 way streets). The degree of vertex A in that example is 2, while the degree of vertex B is 3.

2. A graph that satisfies the following: if (v_1, v_2) $\in$ E, then (v_2, v_1) $\in$ E. I.e., edges are always added in pairs so that the two vertices in the edge are strongly connected.

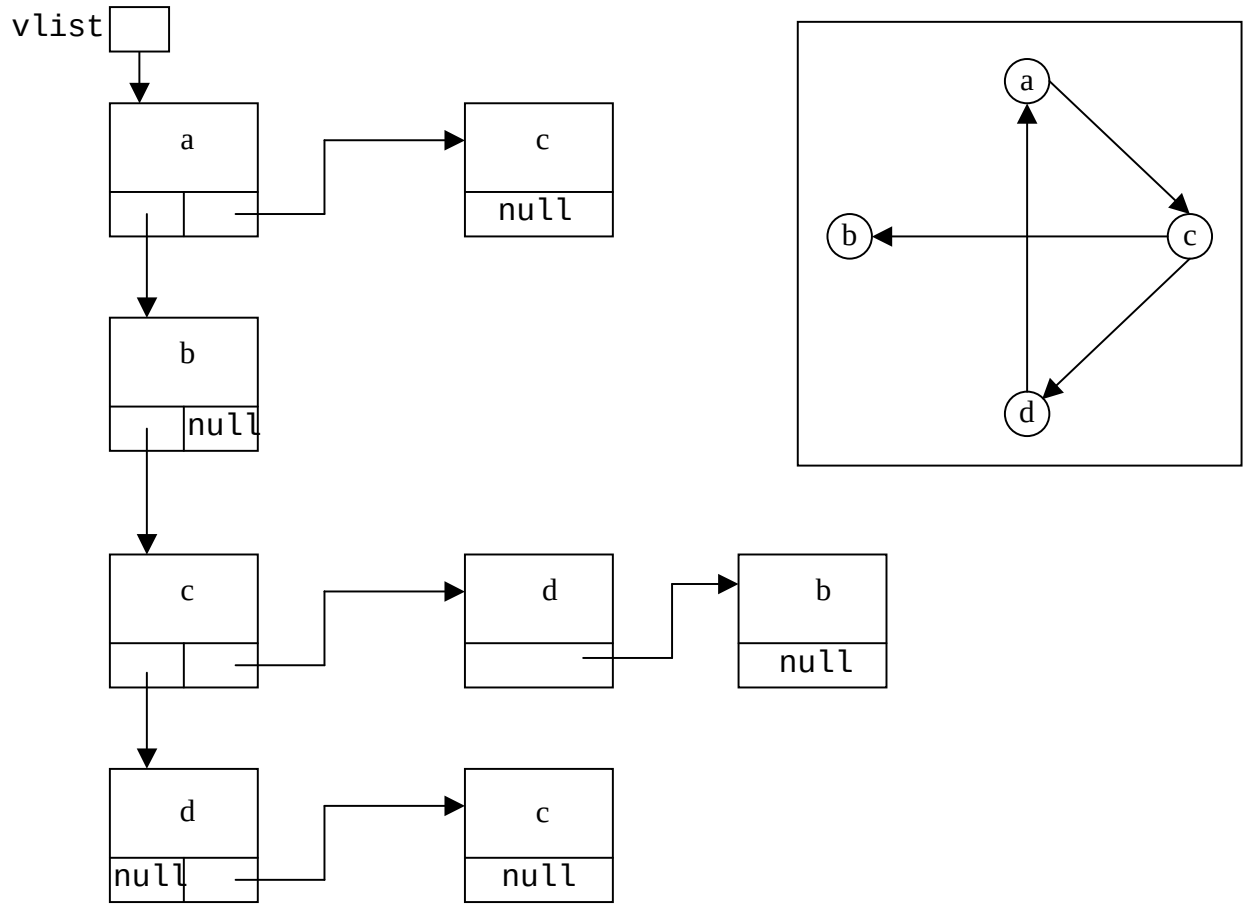
7.2 Implementation

There is a wide diversity of applications of graphs, which all require different operations to be performed on the graph. In much the same way as trees, it is not sensible to try to define a graph ADT that supports all these operations. In most cases the majority of these operations will not be used so the class that implements such an ADT would be overcomplicated and cumbersome. Instead, we will just look at some data structures that can be used to represent graphs. For an actual implementation, a good way to proceed is to define a minimal `DiGraph` class with basic low-level operations for constructing graphs, such as insertion and deletion of vertices and edges. This class can then be extended to include more sophisticated operations and algorithms that are required for particular applications.

There are many ways of representing graphs, but they are generally all variants of two methods:

- Adjacency Lists
- Adjacency Matrices

We will look at how graphs can be stored in both of these data structures.



7.3 Adjacency Lists

For each vertex, v, we keep a linked list of all w's such that {v, w}∈E. The vertices are also stored in a linked list. The diagram above shows a digraph and the data structure that would be used to store it.

To implement a digraph in this way in Java, we use two member classes, `Vertex` and `Edge` – notice that the structure above is built up from two slightly different objects, the link list nodes on the left that represent the vertices and the nodes of the adjacency lists, which represent edges. Here is an outline of the class:

```
public class DiGraphAL {
    protected class Edge {
        public Edge(String succ, Edge n) {
            successor = succ;
            next = n;
        }

        protected String successor;
        protected Edge next;
    }

    protected class Vertex {
        public Vertex(String l, Vertex v) {
            label = l;
            next = v;
            edges = null;
        }

        protected String label;
        protected Vertex next;
        protected Edge edges;
    }

    //Protected data member which points to the top of the list
    //of vertices.
    protected Vertex vlist;

    public DiGraphAL() {
        vlist = null;
    }

    /**
     * Insterts a vertex with label l into the graph.
     */
    public void insertVertex(String l) {

        Include check that vertex isn't already in the graph…
```

```
            //Link in the new vertex
            vlist = new Vertex(l, vlist);
        }

        /**
         * Insterts an edge from the vertex with label
         * pred to that with label succ.
         */
        public void insertEdge(String pred, String succ) {

            Include checks that both Strings represent valid vertices
            and that the edge isn't already in the graph.

            Vertex v = vlist;
            while(!v.label.equals(pred)) {
                v = v.next;
            }

            //Add new edge at start of list.
            v.edges = new Edge(succ, v.edges);
        }
    }
```

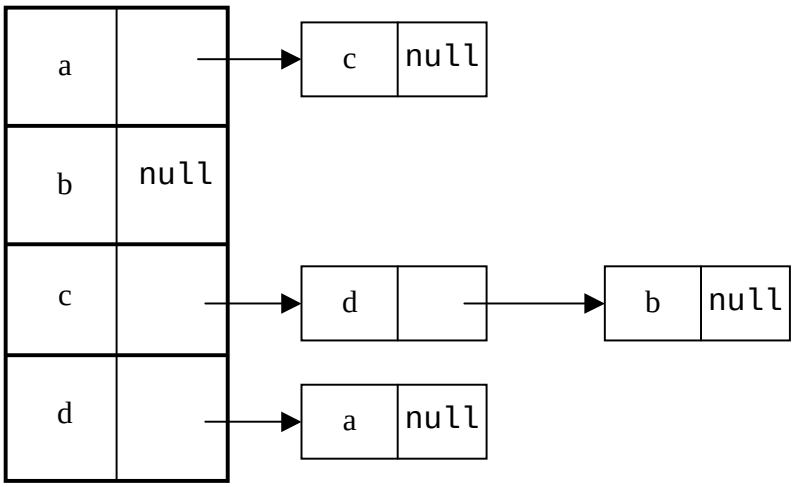
The code above is incomplete and various checks should be included in the insert methods. The delete methods have been omitted too, but these are important in a proper implementation. Basically, new vertices are added at the start of the linked list of vertices (the easiest place to insert into a linked list) and, similarly, new edges are added at the start of the linked list of edges referenced by a vertex.

Notice that the list of vertices is essentially a LUT. In order to perform an operation on a particular vertex, such as add an edge, we need to chain through the list of vertices comparing vertex labels until we find the one we are looking for. We are using a linear search strategy on a LUT whose entries consist of keys that are the vertex labels and values that are lists of edges of that vertex.

If the edges have labels as well as the vertices, we can include an extra `label` field in the edge class. We would also have to include an extra argument to the `insertEdge` method allowing the value of the label to be set when the edge is created, for example:

```
public void insertEdge(String pred, String succ, int label) {
```

7.3.1 Simplifications of the General Adjacency List Structure

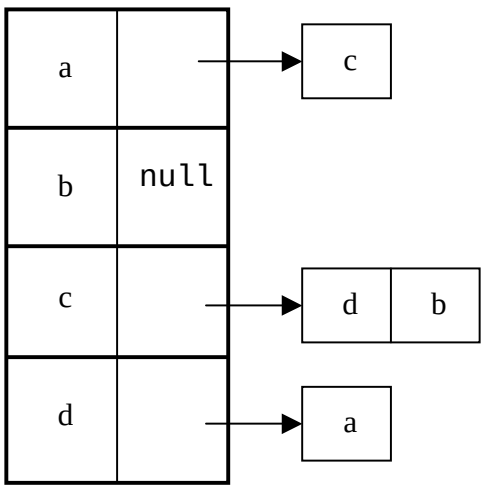


The general adjacency list data structure can be simplified if we have information about the graph in advance. Firstly, if we know how many vertices there are, we can use an array of vertices rather than a linked list. Each element of the array represents a vertex and references a linked list of edges, as shown above.

The `Vertex` class no longer requires the link field and so consists of just a label and a reference to an `Edge` object.

As before, in order to retrieve a particular vertex we need to search for it given its label. However, now that the set of vertices is of predefined length, we can use a more efficient LUT, such as a hash table, to locate the cell of the array that contains a particular vertex.

If, in addition to the number of vertices, we also know how many edges there are in advance for each vertex, then the adjacency lists themselves can also be replaced by arrays. For each slot in the array of vertices, we store an array of edge objects each containing the label of a successor of that vertex and possibly a label for the edge itself.



7.4 Adjacency Matrices

If we know an upper bound on the number of vertices, |V|, in a graph, then we can use a |V| x |V| matrix (a 2D array) of Boolean values to store the graph:

	a	b	c	d
a	F	F	T	F
b	F	F	F	F
c	F	T	F	T
d	T	F	F	F

$$A(v,w) = \begin{cases} \text{TRUE} & \text{if } (v,w) \in E \\ \text{FALSE} & \text{otherwise} \end{cases}$$

If the edges are labelled, we can store the labels in the matrix rather than Boolean values. We need to choose some "null" value to represent that an edge is not present, e.g., an empty string or a negative number.

The vertex labels are stored in a LUT with the label as key and the index of the adjacency matrix as value. In the example above, the LUT would contain the entries: {a, 0}, {b, 1}, {c, 2}, {d, 3}.

Here is an outline implementation of a digraph using an adjacency matrix:

```
import java.util.Hashtable;

public class DiGraphAM {

    //Protected data members
    protected int [][] aMatrix;
    protected hashTable vertLUT;
    protected int nextVertIndex;

    public DiGraphAM(int maxVerts) {
        aMatrix = new boolean[maxVerts][maxVerts];

        //Set all slots to empty
        for(int i=0; i<maxVerts; i++) {
            for(int j=0; j<maxVerts; j++) {
                aMatrix[i][j] = false;
            }
        }

        //Initialise hashtable
        vertLUT = new Hashtable(maxVerts*3/2);
        nextVertIndex = 0;
    }

    public void insertVertex(String l) {

        Check vertex isn't already in there
```

```
            vertLUT.put(l, new Integer(nextVertIndex));
            nextVertIndex += 1;
        }

        public void insertEdge(String pred, String succ)
                                    throws VertexNotFoundException {
            Integer p;
            Integer s;

            //If either vertex is not in the graph, the hashtable
            //will throw an exception.
            try {
                p = (Integer)vertLUT.get(pred);
                s = (Integer)vertLUT.get(succ);
            } catch(Exception e) {
                //We'll catch the hashtable exception and throw
                //our own.
                throw new VertexNotFoundException();
            }

            int pIndex = p.intValue();
            int sIndex = s.intValue();

            aMatrix[pIndex][sIndex] = true;
        }
    }
```

Exercise: can you see any problems with using a hashtable to store the vertex labels in this way? Think about how you might implement a toString method for this class, for example. A solution might be to use an additional protected data item, which is an array storing the vertex labels in order.

The operation of testing for the existence of an edge is generally a little more efficient with an adjacency matrix, but at the expense of larger storage requirements.

7.5 Relations and Digraphs

Graphs can be used to model *binary relations*. Binary relations are things like “less than” or “greater than” comparisons, or “is equal to” or “is not equal to”, etc, which take two values and return a boolean indicating whether the relation is TRUE or FALSE.

Given a set S, the *Cartesian Product* of S with itself, S ⊗ S, is the set of *pairs* of elements of S:

If S = {a, b, c, …}, then,

S ⊗ S = { <a, a >, <a, b >, <a, c > …

<b, a >, <b, b >, <b, c > …

<c, a >, <c, b >, <c, c > …

: …}

A binary relation, R, between elements of a set S can be defined as a subset of S ⊗ S. The relation R on elements a and b of set S is TRUE if and only if <a, b > ∈ R ⊂ S ⊗ S, i.e., the relation holds for a and b if the pair <a, b > is in the set R.

As an example of this, suppose S is the set of non-negative integers, 0, 1, 2, …, and $R_{\leq}$ is the "less than or equal to" relation, ≤. S ⊗ S is the set of all pairs of non-negative integers. $R_{\leq}$ is the subset of S ⊗ S in which the first integer in the pair "is less than or equal to" the second:

S ⊗ S = { <0, 0 >, <0, 1 >, <0, 2 > …
<1, 0 >, <1, 1 >, <1, 2 > …
<2, 0 >, <2, 1 >, <2, 2 > …
: …
}

$R_{\leq}$ = { <0, 0 >, <0, 1 >, <0, 2 > …
<1, 1 >, <1, 2 > …
<2, 2 > …
…
}

Thus,

x ≤ y is equivalent to <x, y > ∈ $R_{\leq}$. In general, we write "xRy" (c.f., x ≤ y) if <x, y > ∈ R for some binary relation R.

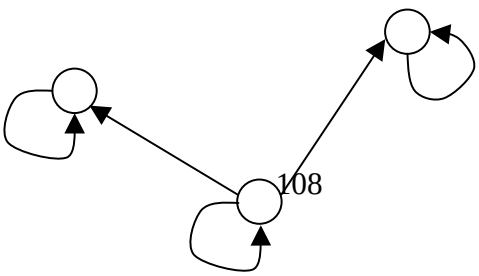
Suppose S is the set of vertices of a graph, G. The set of edges, E, of G is a set of pairs of vertices, i.e., it is a subset of the Cartesian product S ⊗ S and so defines a binary relation, R, on the set, S, of vertices. For every edge <v_1, v_2> ∈ E, we have that $v_1 R v_2$. Graphs and relations are equivalent in this way: any relation R on a set S can be represented as a graph <S, R> and any graph with vertex set S defines a relation on S by its set of edges.

We could express any random collection of vertices and edges as a relation in this way, but there are certain properties that are important in a relation.

7.5.1 Reflexivity

A binary relation R is *reflexive* if xRx for all x in S.

In graph terms, this means that the pair of vertices <x, x > is in the set of edges for all vertices x in S. So every vertex has a self-loop:



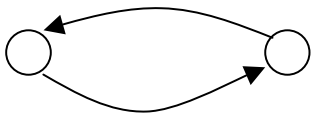
For any integer x, it is always true to say that x ≤ x. So the relation ≤ on the set of integers is reflexive.

7.5.2 Irreflexivity

A binary relation R is *irreflexive* if xRx is FALSE for all x ∈ S. In graph terms there are no self loops. the relation '<' – the "less than" relation, is irreflexive on the integers, since there is no integer x for which x < x.

7.5.3 Symmetry

A relation R is *symmetric* if xRy => yRx for all x and y in S. I.e., if the relation holds in one direction, it must hold in the other direction. So in graph terms it means that all edges (except self-loops) occur in pairs – vertices are either connected in both directions or they are not connected at all:



On the integers, the equality relation, "is equal to", is symmetric, since if x == y then y == x for any integers x and y. The equality relation is also reflexive.

7.5.4 Antisymmetry

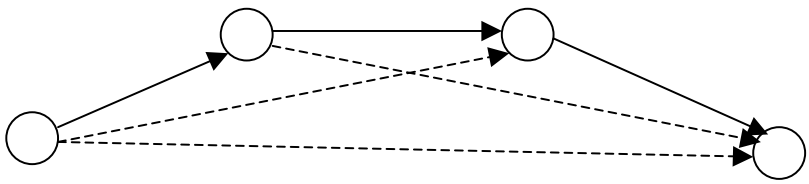
A relation R is antisymmetric if xRy and yRx => x = y. I.e., the relation can only hold in both directions if the two entities being compared are the same. Conversely, if x is not equal to y, then either xRy or yRx (or both) must be FALSE.

In digraph terms, it means that no pair of distinct vertices is connected in both directions.

The relation ≤ on the integers is antisymmetric, since x ≤ y and y ≤ x can only both be true if x == y.

7.5.5 Transitivity

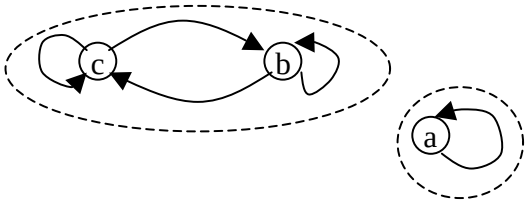
A relation R is *transitive* if (xRy and yRz) => xRz. In digraph terms, the endpoints of all proper paths are connected by a single edge. In the diagram below, if the graph represents a transitive relation then the existence of the solid edges means that the dashed ones must exist too:



The relations ≤ and < are both transitive. If x < y and y < z then x < z.

7.5.6 Equivalence Relations

A relation is an equivalence relation if it is reflexive, symmetric and transitive. The idea of an equivalence relation is to allow a notion of "sameness" without expressly having equality. An equivalence relation R *partitions* the domain S into disjoint subsets E_{xi} of the form



$E_x = \{y \mid yRx\}$ – the set of nodes related to x.

$S = E_{x1} \cup E_{x2} \cup \ldots$ – the union of these subsets is S.

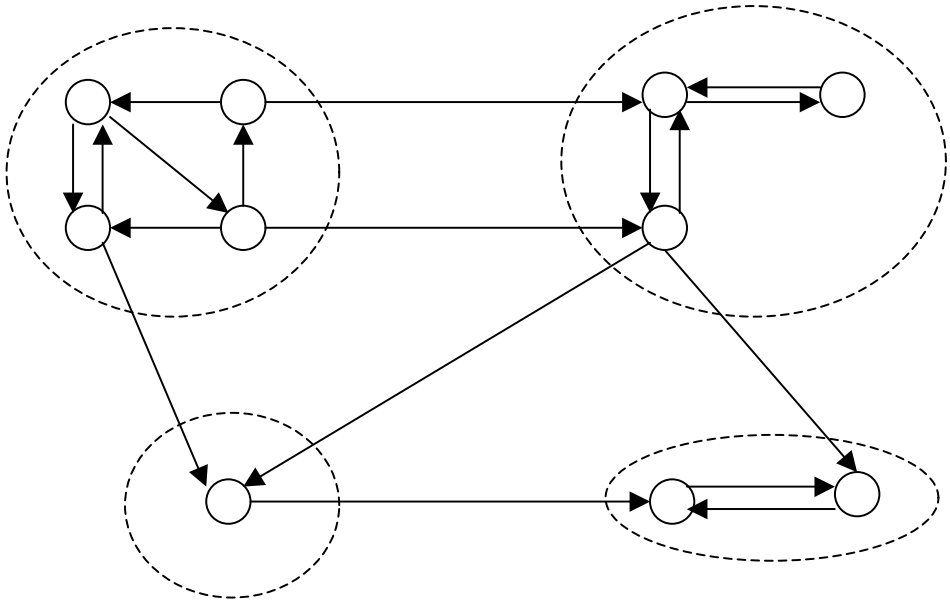
The subsets E_{xi} are the *equivalence classes* of S.

As an example, the relation:

R = {<a, a>, <b, b>, <b, c>, <c, b>, <c, c>}

is an equivalence relation on the set S = {a, b, c}, which partitions S into equivalence classes {a} and {b, c}.

In a directed graph the relation "is strongly connected to" is an equivalence relation. The graph below is partitioned into four equivalence classes by this relation.



In an undirected graph, the relation "is connected to" is an equivalence relation – in fact it is the same as the "is strongly connected to" relation in a digraph.

7.5.7 Partial Orders

A relation is a partial order if it is reflexive, antisymmetric and transitive. An example is ≤ on the integers.

Theorem

Digraphs of partial orders are acyclic

Proof

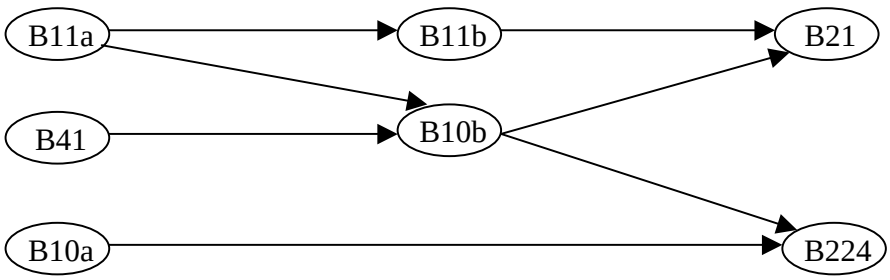
Suppose a graph has a proper cycle (v_1, v_2, …v_k), v_1=v_k, then v_1Rv_2, and, by transitivity, v_2Rv_k. But this means that v_2Rv_1, so, by antisymmetry $v_1 = v_2$, which contradicts the assertion that the cycle was proper.

7.6 Directed Acyclic Graphs

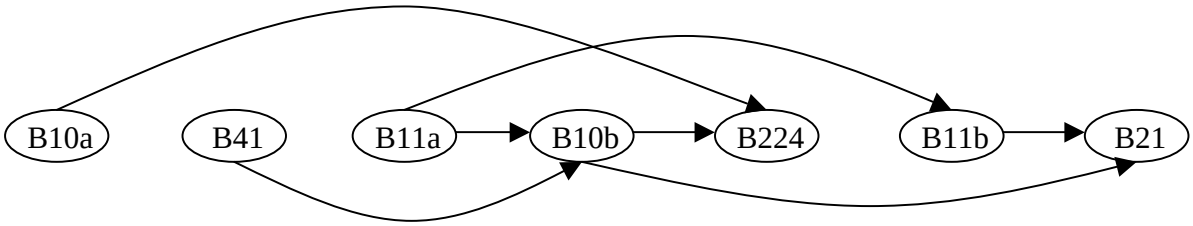
The presence of cycles introduces a lot of complication in processing graphs, because of the danger of infinite loops. Directed acyclic graphs (DAGs) are digraphs in which there are no cycles and these are of special interest. They arise whenever activities need to be performed in some order. That order may not be unique, but some actions need to be performed before others.

An example of this is time-tabling a set of courses in which there are prerequisites. We can express the prerequisites of each course clearly and compactly using a DAG:

The labels of the nodes in the graph below are the names of courses and the edges indicate that one course (the predecessor) is a prerequisite for another (the successor).



A *topological ordering* of the vertices of a DAG is a way to visit all the vertices that satisfies these prerequisites.



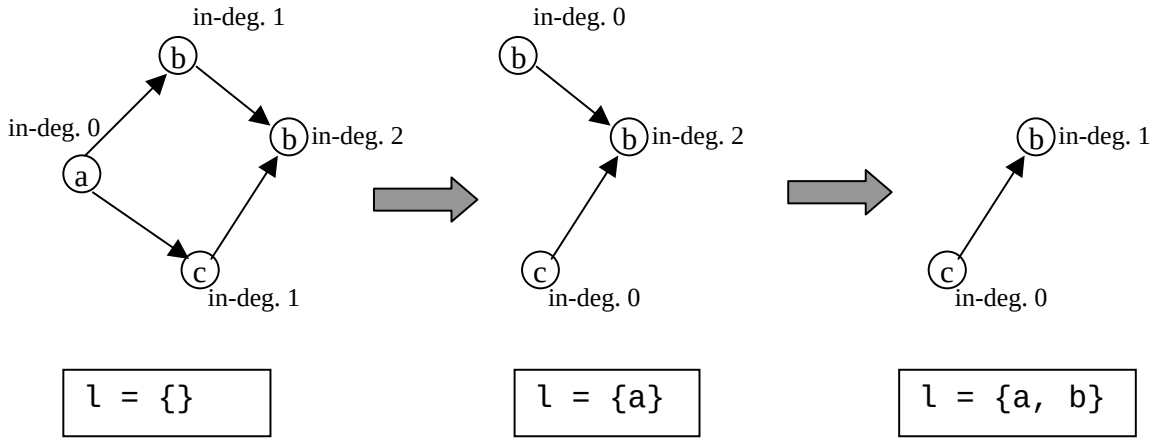
Specifically, we want to find a sequence (v_1, v_2, …, v_n) such that if there is an edge <v_j, v_k > then j < k; equivalently, if v_jRv_k then j < k.

7.6.1 Topological Sort

A topological ordering is found by using a *topological sorting* algorithm. A straightforward way to produce a topological ordering is to search for a vertex of in-degree 0, add it next in the ordering and remove it and all its out-going edges from the graph. Here is an outline of the algorithm:

```
List topologicalSort(DiGraph G) {
    List l = new List();
    Vertex v;
    while(G is not empty) {
        v = a vertex of in-degree 0;
        l.append(v);
        G.delete(v and all its out-going edges);
    }
    return l;
}
```

Here is a simple example:



Eventually, l = {a, b, c, d}.

7.7 Traversal of Digraphs

Traversal of a graph is similar to the idea of tree traversal. It means to visit every vertex in the graph once and perform some operation at that vertex. There are two important ways of traversing graphs.

7.7.1 Depth First Traversal

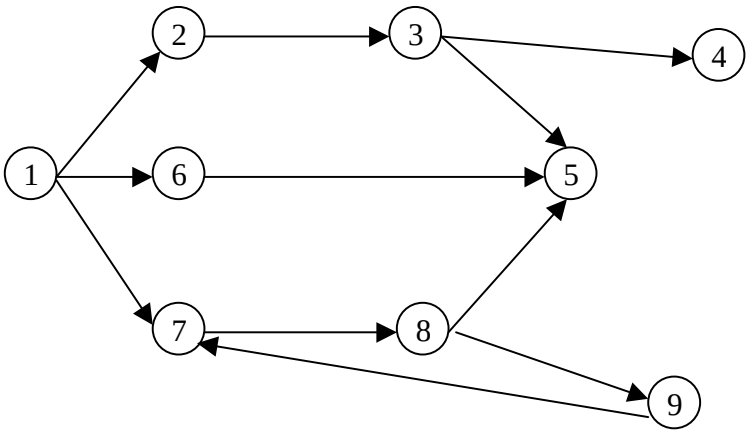
As usual, our “visits” will constitute a print operation, but this could be replaced by anything else. Here is an outline of a *depth* first traversal of a digraph. These would be methods of a `DiGraph` class.

```
public void depthFirstPrint() {
    boolean visited[] = new boolean[number of vertices];

    set the whole array to false.
```

```
    foreach vertex v in V {
        if(!visited[indexOf(v)]) {
            traverse(v, visited);
        }
    }
}

protected void traverse(Vertex v, boolean[] visited) {

    //Visit the vertex
    System.out.println(v.label);
    visited[indexOf(v)] = true;

    foreach w such that (v, w) is in E {
        if(!visited[indexOf(w)]) {
            traverse(w, visited);
        }
    }
}
```

Note that the actual visit can be done in pre or post order. Above it is done in pre-order, since the visit is performed before the traversal. We could move the statement `System.out.println(v.label)` to after the loop over the edges out of the vertex, to make it post-order.

Here is a graph showing the order of traversal using the pre-order depth first traversal above:



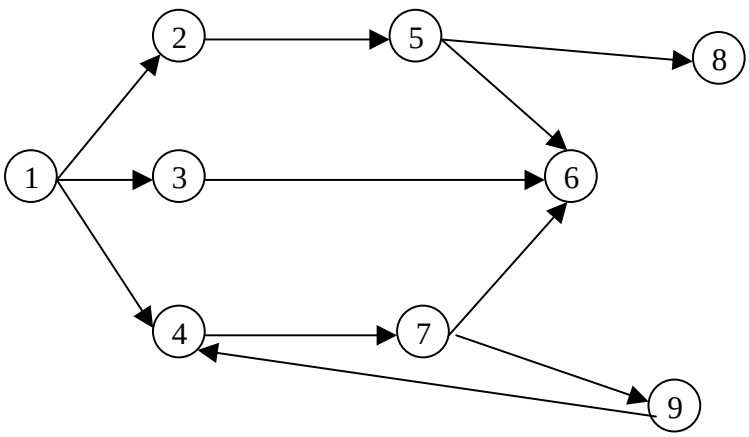
When the visits are performed *after* traversal (post-order) the vertices are visited in *reverse* topological order. *Exercise: try this by hand on the graph above starting with the left-most vertex.*

7.7.2 Breadth First Traversal

Here is an outline of a *breadth* first print method of the `DiGraph` class.

```
    public void breadthFirstPrint() {
        boolean visited[] = new boolean[number of vertices];

        set the whole array to false.

        Queue q = new Queue();

        foreach vertex v in V {
            if(!visited[indexOf(v)]) {

                //Add vertex to queue and mark as visited
                visited[indexOf(v)] = true;
                q.enQueue(v);

                while(!q.empty()) {

                    //Visit vertex at head of queue and add its
                    //successors to queue.
                    Vertex v = q.deQueue();
                    System.out.println(v.label);
                    foreach w such that (v, w) is in E {
                        if(!visited[indexOf(w)]) {

                            //Mark as visited when entering queue
                            q.enQueue(w);
                            visited[indexOf(w)] = true;
                        }
                    }
                }
            }
        }
    }
```

Here is the order of visiting in the same graph as before:



7.8 Some Other Algorithms

In this final section, we'll look at some other algorithms for solving common problems concerning graphs.

7.8.1 Breadth First Topological Sort

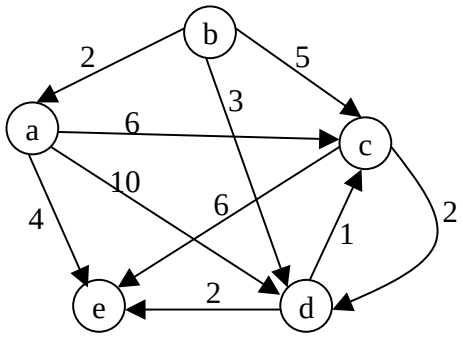
We looked previously at a basic outline of a topological sorting algorithm. Here is an alternative method, based on a breadth first traversal of the graph, which does not require us to destroy the graph in the process of doing the sorting.

```
//Start by counting the number of predecessors of each vertex.
int[] predecessors = new int[number of vertices];
boolean[] visited = new boolean[number of vertices];
foreach v in V {
    foreach w such that (v,w) is in E {
        predecessors[indexOf(w)] += 1;
        visited[indexOf(w)] = false
    }
}

//Produce the ordering
Queue q = new Queue();
foreach v in V {
    if(predecessors[indexOf(v)] == 0 && !visited[indexOf(v)]) {
        q.enQueue(v);
        visited[indexOf(v)] = true;
    }
}
while(!q.empty()) {

    //Head of queue is next in topological order
    Vertex v = q.deQueue();

    //Add vertex to topological ordering.
    System.out.println(v.label);

    //We may now be able to add some of the vertices its
    //connected to.
    foreach w such that (v,w) is in E {
        predecessors[indexOf(w)] -= 1;
        if(predecessors[indexOf(w)] == 0) {
            q.enQueue(w);
        }
    }
```

```
}
```

7.8.2 Shortest Paths

A natural question to ask about directed graphs in which the edges are labelled with costs is: what is the shortest path from one vertex to another, i.e., the path that incurs the least total cost.



Suppose we want to get from vertex b to vertex c in the labelled digraph above.

Path: {(b,c)} = cost of 5

Path: {(b, d), (d, c)} = cost of 3 + cost of 1 = cost of 4.

Path: {(b, a), (a, c)} = cost of 2 + cost of 6 = cost of 8.

There are other even longer paths, the “shortest” path is the one with the smallest cost, which in this case is the second path above – not the most direct path.

To evaluate the shortest path involves incrementally adding vertices to a set S of vertices whose shortest path from S is known. As a corollary, we find the shortest path to all the other vertices too.

Suppose we want to find the shortest paths from vertex a to all the others. A set S contains vertices whose shortest path from a are known (indicated by the boolean array `distanceFound` in the algorithm below). The distance is stored in an array of distances, `distance`.

We consider every vertex w for which the distance is not yet known, but for which it can be determined, as a candidate to add to the set S.

This candidate shortest path must be the one which minimises distance(a, v) + cost(v, w), over the set of all v’s already in S.

Here is an outline of the algorithm:

```
public shortestPath(Vertex v1, Vertex v2)
    boolean[] distanceFound = new boolean[number of vertices];
    float[] distance = new float[number of vertices];

    Vertex v, w;
    float min;
    float infinity = some very large number;
```

```
    foreach w such that (v1, w) is in E {
        distance[indexOf(w)] = cost(v1, w);
    }
    distanceFound[indexOf(v1)] = true;

    v = v1;

    //Main loop.
    for(int i=0; i<no. of verts; i++) {
        min = infinity;
        foreach w in V {
            if(!distanceFound[indexOf(w)] &&
                distance[indexOf(w)] < min) {
                v = w;
                min = distance[indexOf(w)];
            }
        }

        //Add to set S
        distanceFound[indexOf(v)] = true;

        //Update distances
        foreach w such that (v, w) is in E {
            if(min + cost(v,w) < distance[indexOf(w)]) {
                distance[indexOf(w)] = min + cost(v,w);
            }
        }
    }

    return distance[indexOf(v2)];
}
```

This is a version of Dijkstra's algorithm. It provides the cost of the shortest path from the specified vertex to all the others. No computational effort can be saved by just computing the cost to one, specific other vertex. Note that if you need to get the actual paths themselves, an array of predecessors needs to be maintained through the algorithm.

A more detailed description of shortest path algorithms can be found in Kruse, et al; Aho, et al; and Kingston.