Data Structures and Algorithms

Andrew Ensor and Seth Hall
School of Computer and Mathematical Sciences

Autumn 2021

Course Information

Aim To further develop program design skills by studying data structures and contemporary algorithms.

Content Software development, set collections, linear collections, trees, hashing and databases, graphs.

Classes Thursday 8am-10am in room WG404.

Computer Labs Friday 8am-11am in room WS313 and Friday 1pm-4pm in room WS313. Please note that only lab material should be worked on during this time (i.e. no working on assignments, surfing the web, e-mail, nor games). Students who need extra assistance are more than welcome to turn up to multiple labs.

Instructor Dr Molood Barati, room WT134.

Contacts E-mail: mbarati@aut.ac.nz

Blackboard Students are encouraged to regularly check the course web site on Blackboard at `http://blackboard.aut.ac.nz/`. This web site contains class announcements, discussion forums, assignment information, class resources, as well as updated class marks.

Assessment The assessment will be measured through course work, a mid-semester test, and a comprehensive final examination.

Course Work The course work grade will be based equally on two practical computer assignments and is worth a total of 40% of the final grade. These assignments are to be completed in the student's own time and assignments should be submitted in the correct assignment folder on AUT online by midnight on the due date. There is also 10% allocated for lab work each week which will be ticked off the following week.

Late Assignments The policy with late assignments is that for each day an assignment is late it has one fifth of its marks deducted.

Final Examination There is a three hour controlled practical asssessment worth 50% of the final grade at the end of the semester.

Textbook The textbook for this course is *Java Software Structures - designing and using data structures* by Lewis and Chase.

Collaboration Students are welcome to discuss their assignment work with their instructor and with other students. However, unless otherwise stated, all the student's work must be their own. Hence no student may receive any part of an assignment from another person, whether in printed or electronic form. Failure to abide by this will result in the assignment not being accepted. Any form of cheating during the mid-semester test or the final examination is not acceptable.

Timetable for Autumn 2021

Week	Thursday	Thursday
1	Class 1 4 March Software Engineering	Class 2 4 March Analysis of Algorithms
2	Class 3 11 March Threads	Class 4 11 March Collections and Data Structures
3	Class 5 18 March Implementing Sets using Arrays	Class 6 18 March Implementing Sets using Links
4	Class 7 25 March Stacks	Class 8 25 March Queues
	Mid-Semester Break (29 March to 9 April)	
5	Class 9 15 April Lists	Class 10 15 April Searching
6	Class 11 22 April Sorting	Class 12 22 April Specialized Sorting Algorithms
7	Class 13 29 April Trees	Class 14 29 April Binary Search Trees
[1]8	Class 15 6 May Balancing Binary Trees	Class 16 6 May Heaps
9	Class 17 13 May Hash Tables	Class 18 13 May Maps
10	Class 19 20 May Data Structures for Graphs	Class 20 20 May Graph Algorithms
[2]11	Class 21 27 May Revision	Class 22 27 May Revision
[3]	*Controlled practical assessment* (31 May to 18 June)	

[1]Assignment one due: 3 May
[2]Assignment two due: 28 May
[4]Controlled Practical Assessment: TBC

Contents

Chapter 1

Software Development

1.1 Software Engineering

Reading: pp2-19

The discipline of *software engineering* is the study of techniques and theory for the development of high-quality software, which solves the right problem, on time and within budget. The overall quality of software has various characteristics:

Correctness It is essential that the software solves the correct problem, addressing the needs of the client and carefully following the requirements specification.

Reliability The software should be reliable, in the sense that the mean time between failures (unacceptable behaviour) is high, when operating within acceptable conditions. Using common, well-tested libraries, software components, and algorithms saves development time and effort, as well as producing more reliable systems than software prepared entirely from scratch.

Robustness Robust software should gracefully cope with problems, such as unexpected user interaction with the software. It should minimize the effects of a failure, and do no harm in the event of a failure. Important here is the propagation of exceptions - the design decision as to where various exceptions should be handled in the software.

Usability The user interface should facilitate interactions between the user and the software, following the guidelines of good human-computer interaction (HCI).

Maintainability The software developers must create software that is maintainable by others, being well-structured, well written, carefully documented, simple to understand, and which follows common software practices.

Reusability The software should be designed as modular components so that it or parts of it can be easily incorporated into new systems without the need for redesigning or recoding. If it follows standard software design patterns then it is more readily reusable in other projects.

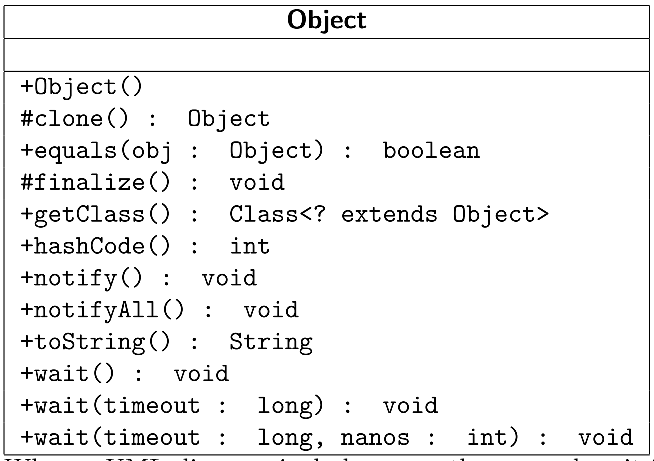
Portability Software is considered *portable* if it can be moved from one computer platform to another with comparatively little effort. This stops the software from being locked in to a single platform and eases the client's transition between platforms.

Efficiency The software should be designed to efficiently use resources such as CPU time, memory, storage, and network resources. The algorithms and data structures used by the software must be carefully chosen.

In order to achieve high-quality software it is essential that an organized strategy for its development (called a *software development model*) is followed from the initial establishment of requirements, through the creation of the design, implementation of the design, and to the testing of the software.

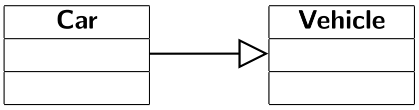
The de facto standard for describing and communicating software design is the *Unified Modeling Language* (UML). A *UML class diagram* is a diagram showing one or more classes, each divided into three sections, with the class name at the top, followed by the attributes (fields) of the class, and then the operations (methods) of the class. The class name appears in bold and might be annotated by a *stereotype* to indicate further information about the class, such as `<<abstract>>` for abstract classes (which contain some method declarations which are not defined in code, and used to organize common features in a class hierarchy), `<<final>>` for final classes (which cannot be subclassed, so none of its methods can be overridden), or `<<interface>>` for interfaces (which declare methods that must be defined in code by any implementing class, and used to describe some required behaviour). An alternative to the `<<abstract>>` stereotype is to display the class name in italics.

Each attribute (field) of the class that is considered relevant for the UML diagram is described on its own line by specifying its visibility, followed by its name, a colon, and its type. The visibility is given symbolically by a `-` for private visibility (only visible to objects of that class), `#` for protected visibility (only visible to objects of that class or subclasses of it), or `+` for public visibility (visible to any objects to which that class is already visible). Similarly, any operation (method) of the class considered relevant is described by specifying its visibility, followed by its name, its parameter list in brackets, a colon, and its return type. The parameter list is given by specifying each parameter name with a colon followed by its type, and parameters are separated by commas. For example, the UML diagram for the `Object` class shows that it has no fields, one constructor and eleven methods.

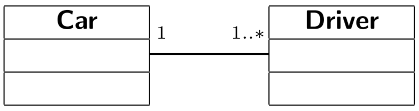


```
Object

+Object()
#clone() : Object
+equals(obj : Object) : boolean
#finalize() : void
+getClass() : Class<? extends Object>
+hashCode() : int
+notify() : void
+notifyAll() : void
+toString() : String
+wait() : void
+wait(timeout : long) : void
+wait(timeout : long, nanos : int) : void
```

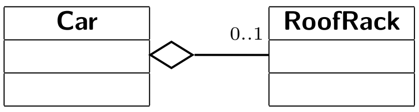
When a UML diagram includes more than one class it is usual to use lines to illustrate the relationships between the classes. Some of the more common types of relationship are:



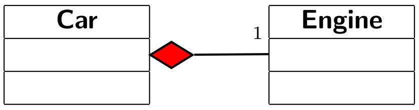
Inheritance where a subclass extends a superclass, such as a `Car` class extending a more general `Vehicle` class.
This is often described as an *is-a* relationship, and is indicated on a UML diagram as a line with an open arrowhead pointing from the subclass to the superclass.



Association where there is a relationship between the instances (objects) created from two classes, such as instances of a `Car` class with instances of a `Driver` class.
The associations that are generally of interest are those that persist for a non-negligible period of time compared to the life of the objects. An association is indicated as a line connecting the two classes, optionally adorned with the number of objects from each class (where an annotation such as 1..* indicates one or more objects from that class).

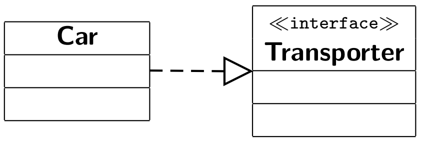


Aggregation where objects from one class are comprised, at least in part, from objects created from another class, such as a `Car` object is an aggregation of various other objects including a `RoofRack` object.
This is often describes as a *has-a* relationship, and is indicated by a line with an open diamond on the aggregate end of the relationship.

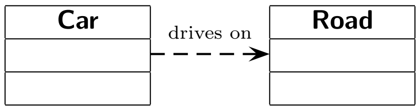


Composition which is a special type of aggregation which emphasizes that the part can belong to only one whole, such as a `Car` class and an `Engine` class.

In such a case a filled diamond is used on the aggregate end.



Implementation where one class implements the methods declared in an interface, such as a `Car` class and a `Transporter` interface.
If a class implements an interface a dashed line is drawn with an open arrowhead at the interface end.



Dependency where one class uses another for an operation (generically called a *service*) it provides, such as a `Car` class and a `Road` class.
One class using another is denoted on a UML diagram by a dotted line with an arrow, and is usually annotated by a brief description of the service.

Exercise 1.1 (Drawing UML Class Diagrams) *Suppose a library program is being designed that will use classes* `Book`*,* `Journal`*,* `Magazine`*,* `Library`*,* `BookIndex`*,* `Customer`*,* `Author`*,* `Librarian` *as well as interfaces* `Loanable` *and* `RestrictedLoanable`*. Draw a UML class diagram which indicates some of the relationships between the classes and interfaces.*

1.2 Analysis of Algorithms

Reading: pp19-24

Often when designing software there might be various alternative algorithms to choose between. Part of the choice depends on a trade off between the efficiency of the algorithm and its simplicity to code and understand. For example, there are various algorithms for sorting values, such as selection sort (which finds the smallest value, then the second smallest etc) and quick sort (which partitions the values into two lists and recursively sorts each). If the quantity of values is guaranteed to be small and the program source needs to be clear then the intuitive but less efficient selection sort might be chosen. If however there are many values then the more efficient quick sort would probably be chosen (in fact many languages such as Java have sorting APIs based on variations of quick sort).

The efficiency of an algorithm can be measured based on various factors (such as memory usage, storage, or network resources), but most commonly is measured in terms of CPU time, giving an indication of how fast the program will execute.

One might consider that given the rapid increase in processing power of computers, the efficiency of an algorithms becomes less important but this is not the case. To illustrate this, consider the common computing task of solving a system of n linear equations in n variables. Two widely taught methods for solving (invertible) systems are Cramer's rule and Gaussian elimination. One can calculate the number of operations required via each technique:

To solve an $n \times n$ system	Cramer's rule	Gaussian elim.
Number of required divisions	n	$\frac{n(n+1)}{2}$
Number of required multiplications	$(n+1)!\,(n-1)$	$\frac{n(n-1)(2n+5)}{6}$
Number of required subtractions	$(n+1)! - (n+1)$	$\frac{n(n-1)(2n+5)}{6}$

Hence, to solve a system of two linear equations in two variables via Cramer's rule would require eleven operations (two divisions, six multiplications, and three subtractions), whereas via Gaussian elimination would require nine operations (three divisions, three multiplications, and three subtractions). Given that division is typically the most complex of the three operations one might consider that Cramer's rule would be suitably efficient but for larger values of n this is not the case. For $n = 10$ (still relatively small), Cramer's rule would require almost 400 million operations (10 divisions, 359251200 multiplications, and 39916789 subtractions), whereas Gaussian elimination requires 805 operations (55 divisions, 375 multiplications, and 375 subtractions). The problem with Cramer's rule is that the number of operations grows very quickly as a function of n, quickly becoming infeasible even on the most powerful processors. Although Gaussian elimination is far preferred and widely used in practice, it too requires significant processing power — doubling the value of n requires approximately eight times the number of operations. Put another way, an increase in processing power by a factor of eight only allows a system of roughly double the size to be solved in the same time as before. This demonstrates that the choice of processor for a task is often far less significant than the choice of algorithm, and greatly improving the performance of a processor might only give modest increases in capabilities.

A function such as $f(n) = \frac{n(n-1)(2n+5)}{6}$ which give the relationship between the complexity $f(n)$ of an algorithm (such as time or number of operations) and the size n of the problem is called a *growth function*. Usually, the precise growth function for an algorithm is not required, but instead it is sufficient to just know its dominant term. The *asymptotic complexity* or *order* of an algorithm refers to the behaviour of the dominant term (the term that increases the quickest for large values of n) in the growth function as n increases. For example, the function $f(n) = \frac{n(n-1)(2n+5)}{6} = \frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n$ is a cubic in n and so is said to have cubic order, denoted by $O\left(n^3\right)$ and often referred to as the *Big-O* notation. Note that the coefficient of the dominant term is not included in the notation, since other factors (such as processor performance or the type of operation counted by the growth function) would be just as significant as the coefficient.

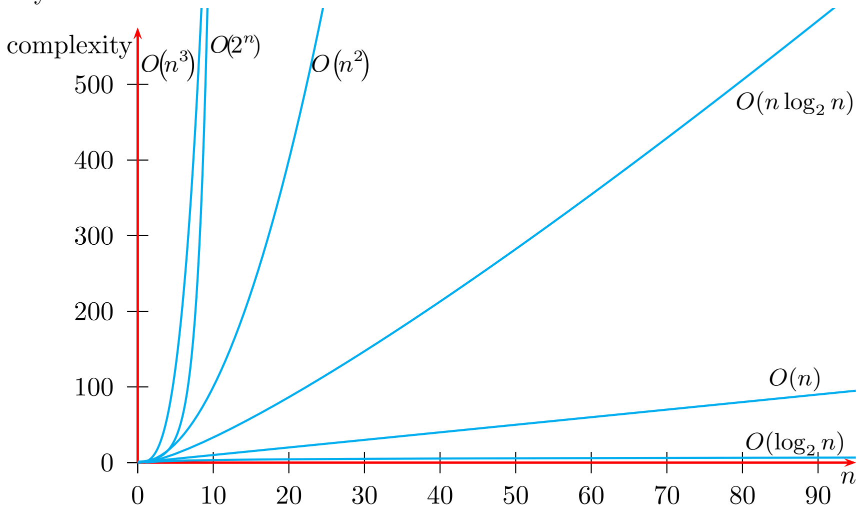
In general, the smaller the order of an algorithm, the more efficient it is considered, and two algorithms are considered equivalent if they have the same order (although in practice one coded algorithm might execute quicker than another of the same order). Some of the orders commonly encountered in computer algorithms listed in decreasing efficiency are:

- $O(1)$ for algorithms that execute in constant time regardless of n, such as appending items to a list,
- $O(\log_2 n)$ for algorithms that execute in logarithmic time (either $\log_2 n$ or $\log n$), such as a binary search of an ordered list of values,
- $O(n)$ for algorithms that execute in linear time, such as a linear search of

an unordered list of values,

- $O(n \log_2 n)$ for algorithms that execute in linear-logarithmic time, such as the faster sorting algorithms like quick sort or heap sort,
- $O(n^2)$ for algorithms that execute in quadratic time, such as the slower sorting algorithms like selection sort or insertion sort,
- $O(n^3)$ for algorithms that execute in cubic time, such as Gaussian elimination,
- $O(2^n)$ for algorithms that execute in exponential time, such as algorithms that require checking possible combinations.

Algorithms that execute in exponential time, or worse in factorial time $O(n!)$ such as Cramer's rule, are generally consider too inefficient to be computationally feasible.



It is important to realize that the Big-O notation only indicates the behaviour of an algorithm's efficiency for large values of n. For example, the growth functions $f(n) = 10n^2$ and $g(n) = n^2 - 5$ are both $O(n^2)$, although if they both count operations that take roughly equal times then the algorithm for g would execute about ten times faster than the algorithm for f. Furthermore, although the growth function $h(n) = 1000n$ is $O(n)$ and so its algorithm would be considered more efficient asymptotically than the other two, for $n < 10$ it actually requires more time.

An operation that does not depend (neither directly nor indirectly) on n has order $O(1)$, since it executes in constant time, such as:

```
System.out.println("Some operation");
```

Likewise, a conditional statement such as:

```
if (n==0)
   System.out.println("Some operation");
else
   System.out.println("Another operation");
```

which results in two operations (a comparison and a `println`), is also considered $O(1)$.

However, a loop that executes some $O(1)$ operation n times, is $O(n)$:

```
for (int i=0; i<n; i++)
   System.out.println("Some operation");
```

as is a loop that executes $\frac{n}{2}$ times, such as:

```
for (int i=0; i<n; i+=2)
   System.out.println("Some operation");
```

A loop that has a more complex increment, such as:

```
for (int i=1; i<n; i*=2)
   System.out.println("Some operation");
```

requires the determination of the number of iterations. In this example the loop iterates with values $i = 1, 2, 4, 8, \ldots$, approximately $\log_2 n$ iterations, so the loop has order $O(\log_2 n)$.

Nested loops can be analyzed in a similar manner. For example, the nested loop:

```
for (int i=0; i<n; i++)
{  for (int j=0; j<n; j++)
      System.out.println("Some operation");
}
```

has order $O\left(n^2\right)$, since the operation inside the loop is executed n^2 times. The nested loop:

```
for (int i=0; i<n; i++)
{  for (int j=0; j<i; j++)
      System.out.println("Some operation");
}
```

executes the operation $0 + 1 + 2 + \ldots + (n-1) = \frac{1}{2}n(n-1)$ times, which is also order $O\left(n^2\right)$.

Exercise 1.2 (Analyzing Selection Sort Algorithm) *The selection sort algorithm works by first finding the smallest value in a list and swapping the first value with it, then finding the second smallest value and swapping the second value with it, continuing until all the values are in order. Implement this algorithm, then determine its growth function, and hence the order of the algorithm.*

1.3 Threads

Reading: none

A computer's ability to have more than one process working (apparently) simultaneously is referred to as *multitasking*. Unless a computer is truly multiprocessor, the operating system achieves this by repeatedly allocating brief periods of the central processor unit's time to each process in turn, giving the impression of parallel (concurrent) activity.

A *multithreaded* program is a program which appears to have more than one task working simultaneously, where each task is known as a *thread*. The essential difference between a process and a thread is that while each process has its own complete set of variables, threads within a program share the same variables. Although multithreading can be rather complicated and requires special care it has several important benefits:

- some tasks are performed quicker in parallel using multiple threads, while one thread might be waiting for a response across a network connection another thread could be performing some other task,
- a program can seem more responsive if the user interface deals with user initiated tasks in a separate thread,
- some algorithms are simpler to implement with multiple threads,
- some tasks require multithreading, such as a program that runs a web server, which must handle multiple connections to it concurrently.

The extra complexity of multithreaded programs arises because several threads can access and manipulate the same data and execute the same statements simultaneously. Also, it is the operating system, not the program, which controls when each thread is allocated processor time.

Every Java program that uses Swing components is actually a multithreaded application. When the Virtual Machine starts up there is a single thread called the *main* thread created, that typically calls some `main` method (for a Swing application this usually just creates and shows a frame). The main thread stays alive until the `main` method is finished. However, the program does not terminate while there is still a (non-daemon) thread alive, and when a GUI window is shown another thread is started, called the *event dispatch* thread, so the Virtual Machine does not yet terminate. All event notifications, such as calls to `actionPerformed` or `paintComponent` actually run in this thread. Calling a component's `repaint` method results in the event dispatch thread calling `paintComponent` on that component when it is given an opportunity to have some processor time. One advantage this provides is for the program to do other tasks while the event handling code or painting takes place. There are also other threads running behind the scenes, such as a thread which posts events notified by the operating system into the event dispatch queue, and a garbage collection thread. The class `MainAndEventQueue` demonstrates how the main thread and the event dispatch thread run simultaneously. Actually, this class also creates another thread, created by the Swing `Timer` object, which is responsible for generating the `ActionEvent` events every 1000 milliseconds.

The `Thread` class in the `java.lang` package is used to create and control threads. Every thread in a program is assigned an `int` priority between the constants `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY`, with default `Thread.NORM_PRIORITY`. Threads with higher priority that are not in a blocked state are executed in preference to threads with lower priority. Each thread can optionally be marked as a daemon, and the Virtual Machine continues to run until all non-daemon threads have died.

A new thread is usually created in a Java program by first writing a class that implements the `Runnable` interface from the `java.lang` package, with a `run` method that contains the code to be executed in its own thread:

```
/**
   A class which demonstrates how a Swing application is multithreaded
   with the main thread and the event dispatch thread
   @author Andrew Ensor
*/
import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.Timer;

public class MainAndEventQueue extends JPanel implements ActionListener
{
   private JLabel label;
   private int counter;

   public MainAndEventQueue()
   {  super();
      setPreferredSize(new Dimension(300, 200));
      label = new JLabel();
      add(label);
      counter = 0;
      // call actionPerformed method every 1000ms using Swing timer
      Timer timer = new Timer(1000, this);
      timer.start();
   }

   public void actionPerformed(ActionEvent e)
   {  label.setText("Event thread counting: " + counter++);
   }

   public static void main(String[] args)
   {  JFrame frame = new JFrame("Main and Event Queue Thread Example");
      // kill all threads when frame closes
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.getContentPane().add(new MainAndEventQueue());
      frame.pack();
      // position the frame in the middle of the screen
      Toolkit tk = Toolkit.getDefaultToolkit();
      Dimension screenDimension = tk.getScreenSize();
      Dimension frameDimension = frame.getSize();
      frame.setLocation((screenDimension.width-frameDimension.width)/2,
         (screenDimension.height-frameDimension.height)/2);
      frame.setVisible(true);
```

cont-

-cont

```
      // now display something while the main thread is still alive
      for (int i=0; i<20; i++)
      {  System.out.println("Main thread counting: " + i);
         try
         {  Thread.sleep(500); // delay for 500ms
         }
         catch (InterruptedException e)
         {}
      }
      System.out.println("Main thread about to die");
   }
}
```

```
public class SomeTask implements Runnable
{
   ...

   public void run()
   {  // code to be executed in its own thread
      ...
   }

   ...
}
```

Then a new `Thread` object is created using this `Runnable` object as a parameter and is started by using the `Thread` method `start`:

```
SomeTask myTask = new SomeTask(...);
Thread myThread = new Thread(myTask);
myThread.start();
```

The `start` method makes the thread eligible for processor time, whereas simply calling the `run` method directly would only execute the `run` method in the current thread (as usual) without creating a new thread (hence the code is always executed by calling the `Thread` method `start`, the `run` method is not called directly). Whenever the new thread is permitted processor time it will continue its execution of the code in the `run` method, resuming from where it was previously (and unaware that it might be sharing processor time with other threads). Note that after calling the `start` method, the original thread then immediately continues its task, while the new thread starts performing its own task concurrently. The new thread continues to be alive until the `run` method has completed, or else there is an uncaught exception. In either case the thread is dead and cannot be restarted.

The `Thread` class itself implements the `Runnable` interface, so occasionally one might see a class extend the `Thread` class, and override its `run` method with the code to be executed by the thread:

```
public class SomeTask extends Thread
{
   ...

   public void run()
   {  // code to be executed in its own thread
      ...
   }

   ...
}
```

Then the `start` method is called by:

```
SomeTask myThread = new SomeTask(...);
myThread.start();
```

Another alternative which is convenient for tasks that should be executed either once or repeatedly at some future time is provided by the `Timer` class in the `java.util` package (a more general-purpose version of the class with the same name in the `javax.swing` package). First a class is written extending the abstract class `TimerTask` (from `java.util`, and which also implements `Runnable`) with a `run` method containing the code to be executed:

```
public class SomeFutureTask extends TimerTask
{
   ...

   public void run()
   {  // code to be executed later in its own thread
      ...
   }

   ...
}
```

Then a `Timer` object is created, and is used to schedule when the future task should be executed:

```
SomeFutureTask myTask = new SomeFutureTask(...);
Timer timer = new Timer();
timer.schedule(myTask,1000,5000); //start in 1s,then every 5s
```

The `schedule` method starts a thread which waits until the specified time or delay before executing the `run` method. The `Timer` method `cancel` can be used to cancel any remaining scheduled tasks, after which they cannot be rescheduled.

For example, the class `MultipleCounters` demonstrates how three simultaneous counters can be set up by implementing the `Runnable` interface. Note from the sample output that the thread for counter `A` paused running part way through one iteration so that the thread for counter `C` could run. Since this thread had higher priority, it was given all the processor time until it completed (some platforms might prevent such starvation of the other threads, giving lower priority threads some time), after which it died. Then the thread for counter `A` continued where it left off (unaware that it had been temporarily paused), but eventually the thread for counter `B` is given some processor time (since it

```
/**
   A class which demonstrates how threads are created
   @author Andrew Ensor
*/

public class MultipleCounters
{
   public static void main(String[] args)
   {  // create three threads
      Counter counterA = new Counter("A", 20);
      Counter counterB = new Counter("B", 20);
      Counter counterC = new Counter("C", 20);
      Thread threadA = new Thread(counterA);
      Thread threadB = new Thread(counterB);
      Thread threadC = new Thread(counterC);
      threadC.setPriority(Thread.NORM_PRIORITY+2); // higher priority
      // start the three threads
      threadA.start();
      threadB.start();
      threadC.start();
   }
}

// a class that represents a counter
class Counter implements Runnable
{
   private String name;
   private int maxCounts;

   public Counter(String name, int maxCounts)
   {  this.name = name;
      this.maxCounts = maxCounts;
   }

   public void run()
   {  for (int i=0; i<maxCounts; i++)
      {  System.out.print("Counter " + name + " is " + i + ", ");
         System.out.println("Counter " + name + " about to loop");
      }
   }
}
```

Sample Output From `MultipleCounters`:

```
Counter A is 0, Counter A about to loop
Counter A is 1, Counter A about to loop
Counter A is 2, Counter A about to loop
Counter A is 3, Counter A about to loop
Counter A is 4, Counter A about to loop
Counter A is 5, Counter C is 0, Counter C about to loop
Counter C is 1, Counter C about to loop
Counter C is 2, Counter C about to loop
Counter C is 3, Counter C about to loop
Counter C is 4, Counter C about to loop
Counter C is 5, Counter C about to loop
Counter C is 6, Counter C about to loop
Counter C is 7, Counter C about to loop
Counter C is 8, Counter C about to loop
Counter C is 9, Counter C about to loop
Counter C is 10, Counter C about to loop
Counter C is 11, Counter C about to loop
Counter C is 12, Counter C about to loop
Counter C is 13, Counter C about to loop
Counter C is 14, Counter C about to loop
Counter C is 15, Counter C about to loop
Counter C is 16, Counter C about to loop
Counter C is 17, Counter C about to loop
Counter C is 18, Counter C about to loop
Counter C is 19, Counter C about to loop
Counter A about to loop
Counter A is 6, Counter A about to loop
Counter A is 7, Counter A about to loop
Counter A is 8, Counter A about to loop
Counter A is 9, Counter A about to loop
Counter A is 10, Counter A about to loop
Counter A is 11, Counter B is 0, Counter B about to loop
Counter B is 1, Counter B about to loop
Counter B is 2, Counter B about to loop
Counter B is 3, Counter B about to loop
Counter B is 4, Counter B about to loop
Counter A about to loop
Counter A is 12, Counter A about to loop
Counter A is 13, Counter A about to loop
Counter A is 14, Counter A about to loop
Counter A is 15, Counter A about to loop
Counter A is 16, Counter A about to loop
Counter A is 17, Counter A about to loop
Counter A is 18, Counter A about to loop
Counter A is 19, Counter A about to loop
Counter B is 5, Counter B about to loop
Counter B is 6, Counter B about to loop
Counter B is 7, Counter B about to loop
Counter B is 8, Counter B about to loop
Counter B is 9, Counter B about to loop
Counter B is 10, Counter B about to loop
Counter B is 11, Counter B about to loop
Counter B is 12, Counter B about to loop
Counter B is 13, Counter B about to loop
Counter B is 14, Counter B about to loop
Counter B is 15, Counter B about to loop
Counter B is 16, Counter B about to loop
Counter B is 17, Counter B about to loop
Counter B is 18, Counter B about to loop
Counter B is 19, Counter B about to loop
```

has the same priority as the thread for counter `A`). These two threads then alternate taking processor time (provided the operating system performs *time-slicing*, sharing time between threads of equal priority, otherwise each thread should periodically call `yield` to give the other thread a chance to execute), each being paused at arbitrary points in their loop so that the other has an opportunity to execute.

At any time a thread is in one of four possible states:

new when a new thread has been created but before its `start` method has been called,

runnable once the thread's `start` method has been called so the thread is eligible for processor time, and usually will be given slices of processor time unless threads of higher priority are using up all the available processor time or it becomes blocked,

blocked when a thread is not currently eligible for processor time, either because its `sleep` method has been called (to put it asleep for a fixed time period), or it is *blocking on i/o* (meaning that it is waiting until some input/output operation is complete), or its `wait` method has been called (so it is waiting for a notification from another thread), or else it is trying to execute a block of synchronized code which another thread has locked,

dead when the code in the `run` method has finished being executed or else an uncaught exception occurred.

Whenever a thread is blocked it is not eligible for processor time. In order for it to again be in a runnable state it must be unblocked, so if sleeping then its sleep time must expire, if blocking on i/o then the i/o operation must complete, if waiting it must receive notification, if locked out of code then the thread with that lock must first release the lock. The `sleep` method is used to put the thread into the blocked state for a fixed period, not wasting processor time that other threads might want to use. It is a good practice to put a thread to sleep periodically (such as inside a loop in the `run` method) to give other (lower priority) threads an opportunity to have processor time. Since threads are automatically placed in a blocked state when they are waiting for i/o, they don't waste any processor time while waiting for operations such as user input or the establishment of a network connection.

The `MultipleCounters` example used independent, asynchronous threads — where each of the threads ran using variables and methods of different `Counter` objects without concern for the state or activities of the other threads. However, there are many cases where concurrently running threads do share data and need to consider the state and activities of the other threads using that data.

Consider for example the class `MoneyMover`. This class has two accounts and simulates the transfer of money back and forth between the accounts via two threads. Each thread repeatedly transfers one dollar randomly between the same two accounts, maintaining a fixed total amount of money in the accounts. Due to the loop in the `run` method, this task should continue indefinitely, unless a thread finds that money has either gone missing or there is too much, which one would expect should never occur. However, if the operating system performs time slicing and both threads run concurrently, the program does typically report a problem. This reveals one of the subtleties of multithreading, since the

```
/**
   A class which demonstrates a race condition and the need
   for code synchronization
   @author Andrew Ensor
*/
import java.util.Random;

public class MoneyMover implements Runnable
{
   private int accountA, accountB; // dollar amounts between 0 and 10
   private final int SUM = 10; // total amount of money
   private Random generator;

   public MoneyMover()
   {  accountA = 5;
      accountB = SUM-accountA;
      generator = new Random();
      Thread threadA = new Thread(this);
      Thread threadB = new Thread(this);
      threadA.start();
      threadB.start();
   }

   public void run()
   {  // should loop forever unless money disappears or appears
      while (isSumCorrect())
      {  // randomly either transfer from A to B or from B to A
         if (generator.nextInt(2)==0)
            transferAToB();
         else
            transferBToA();
      }
      System.out.println("Problem: account A = " + accountA +
         ", account B = " + accountB);
   }

   // transfer one dollar from account A to account B
   // note this method needs to be synchronized
   private void transferAToB()
   {  if (accountA>0)
      {  accountA--;
         accountB++;
      }
   }
```

cont-

-cont

```
   // transfer one dollar from account B to account A
   // note this method needs to be synchronized
   private void transferBToA()
   {  if (accountB>0)
      {  accountB--;
         accountA++;
      }
   }

   // checks whether the total amount is correct
   // note this method needs to be synchronized
   private boolean isSumCorrect()
   {  return (accountA+accountB==SUM);
   }

   public static void main(String[] args)
   {  new MoneyMover();
   }
}
```

program does not control when each thread is paused there is a chance that one thread is paused just after it has taken money out of one account but before it is put into the other account, then the other thread continues its transfers and notices that there is money missing, such as:

```
Problem: account A = 0, account B = 9
```

Notice also that sometimes one of the threads terminates but the total amount is reported correctly, such as:

```
Problem: account A = 4, account B = 6
```

This can occur since although the total amount might have been incorrect during the `while` testing in one thread, by the time it got to its `println` statement the other thread had resumed and finished its transfer.

This example demonstrates a *race condition*, where it has been presumed that once a thread has started a portion of code (transferring money), it will have finished before another thread starts code that might affect it (transferring that same money). Although it is not serious here, it would not be good for a real bank to loose money, and in many situations this can cause a program to crash or produce erroneous results at random times (which can make debugging a real headache).

Code segments within a program that access the same variables from separate concurrent threads are called *critical sections* of code. In Java, such blocks of code or even entire methods can be *synchronized*, which indicate that the code can be locked to prevent more than one thread at a time from getting inside it. The lock can be any object (called a *monitor*) but is typically the object (`this`) whose code is being executed. When the first thread enters a synchronized section, it automatically obtains the lock, then although this thread can still be paused while inside the synchronized code, no other thread can enter any section of code that has been locked with the same object.

The `MoneyMover` class can be synchronized, and thus avoid the race condition, by placing the reserved word `synchronized` in the declaration (before the return type) of the three methods `transferAToB`, `transferBToA`, and `isSumCorrect`. Whenever either thread enters one of these synchronized methods, it obtains the lock (the `MoneyMover` object), which prevents any other thread from entering any of the three methods for this `MoneyMaker` object. Note that synchronization requires careful consideration, if say the `isSumCorrect` were not made synchronized then the error could still occur, but infrequently (and it is worse for an error to occur infrequently, since its likely to go unnoticed until some critical moment, and be difficult to track down).

Sometimes, a thread inside synchronized code might not be able to continue with its task until another thread has done something. In such a case there is no point for that thread to waste processor time that the other could be using, and so the thread can temporarily release its lock by calling the monitor's `wait` method (either indefinitely or for a fixed time). This places the thread in a blocked state, and enables another thread to obtain the lock and enter the synchronized code. Once the other thread has accomplished what was required it can indicate this to waiting thread(s) by calling the monitor's `notifyAll` method. After the other thread has exited the synchronized block, the lock becomes available once again.

Exercise 1.3 (Bouncing Balls) *Design a program which displays several balls bouncing in a box, where the movement of each ball is controlled by its own thread (you might like to make a* `Ball` *class implementing* `Runnable` *and use* `Thread.sleep(delay)` *in its* `run` *method to control the speed of the ball). Try to include buttons for increasing and decreasing the number of balls that are drawn.*

1.4 Collections and Data Structures

Reading: p70,pp84-92

A *collection* is an object that gathers and organizes a group of objects, which are called the *elements* of the collection. Examples of collections include sets, stacks, queues, lists, various types of trees such as binary search trees and heaps, as well as directed, undirected, and weighted graphs.

The implementation of a collection, such as how it stores its elements, is usually hidden or *abstracted* by using private instance variables. Separating how the collection is used from how it is implemented is important in software engineering, since it enables a programmer to focus on how to interact with the collection, rather than its internal details which might involve multidimensional arrays or complicated linked structures. A collection for which the details of the implementation are not defined is often called an *abstract data type*, and the underlying programming details used to implement it are called a *data structure*.

For example, the class `ArrayList` provides an implementation of a collection of objects. The abstract data type is the list of elements together with methods for interacting with the elements, such as `add`, `get`, `remove`, `isEmpty`, and `size`. The actual data structure that is used by the `ArrayList` class to store the elements is an array, but these details are kept private, so that a programmer using the `ArrayList` class can focus on how to use an `ArrayList` rather than details of how it works.

Version 1.5 of Java introduced the concept of *generics* to the programming language, which are parameters that can appear in the place of data types, so that the `ArrayList` class is actually now declared as `ArrayList<E>`, with a generic type `E` specified between angled brackets when an `ArrayList` object is created. To see why generics are so useful when dealing with collections, suppose an `ArrayList` collection is to be used to store a list of `String` names. Prior to generics this would have been achieved by creating an `ArrayList` that can store any *raw type* `Object`, being careful to only add `String` objects to it, and typecasting any object obtained from the `ArrayList` back to the type `String`, such as:

```
ArrayList list = new ArrayList();
list.add("Jack");
list.add("Janet");
String firstName = (String)list.get(0);
```

If an object other than a `String` were in the `ArrayList` collection, the program might generate a runtime exception.

Since the `ArrayList<E>` class now has a generic type, the actual type of elements that will be stored is specified when the `ArrayList` is created, and the methods of this class deal with this specific type rather than treating the elements as arbitrary objects:

```
ArrayList<String> list = new ArrayList<String>();
list.add("Jack");
list.add("Janet");
String firstName = list.get(0); // no typecast required
```

This enables the compiler to check the type correctness of code involving an `ArrayList` at compile-time rather than risking a `ClassCastException` at run-time.

By convention classes that are declared with their own generic types denote the generic types by single character capital letters, such as `E` (for element) or `T` (for type). The local variables and non-static methods (but not static methods nor static fields since the type of `E` can vary between different instances) of the class can use the generic type `E` as though it were a real type for declaring variables, parameter types, typecasting, or method return types. For example, the `ArrayList<E>` class can use an `E[]` array as its data structure to store its elements, the `add` method can require its parameter to be of type `E`, and the `get` method can have return type `E`. However, the generic type cannot be used in the class for invoking constructors or methods, such as:

```
E obj = new E();
```

since it is not known at compile-time what type(s) `E` will be at run-time (and the constructors vary from one class to another). Nor (surprisingly) is the statement:

```
E[] array = new E[10];
```

valid, instead the following cast can be used:

```
E[] array = (E[])(new Object[10]);
```

(which the compiler will warn is an unchecked cast). Constructors and methods can also be declared with their own generic types that are local to that constructor or method (see the `AdjacencyListGraph` constructor and the `removeVertex` method of the `AdjacencyListGraph` class in Section 6.1 for examples).

Note that in terms of inheritance, although a class such as `Car` might be a subclass of a class such as `Vehicle`, `ArrayList<Car>` is not a subtype of `ArrayList<Vehicle>`. Hence the second line of the following code fragment:

```
ArrayList<String> list1 = new ArrayList<String>();
ArrayList<Object> list2 = list1;
```

will not compile (since otherwise `list2` could then allow any `Object` to be added to the collection it references, but `list1` is presumed to reference a collection of only `String` objects). In order to cope with this complication, particularly when passing collections as parameters, the wildcard type `?` can be used. For example, the following method accepts an `ArrayList` collection of any type:

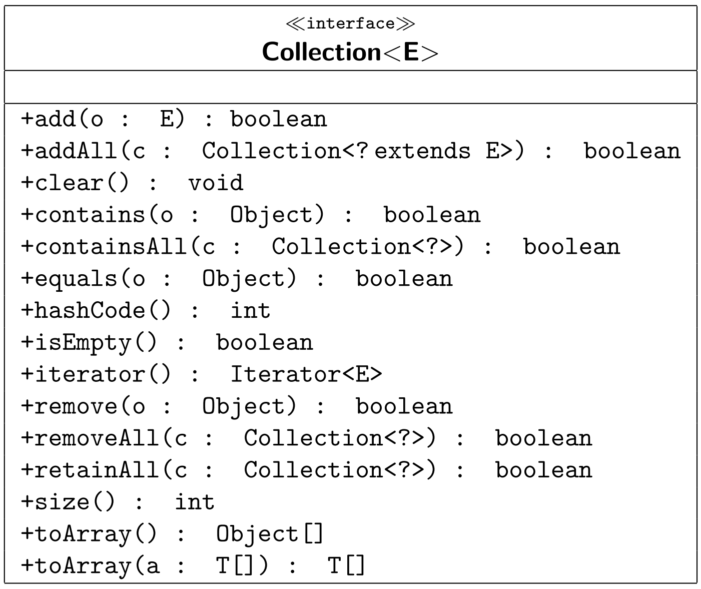
```
public void printList(ArrayList<?> list)
{  for (int i=0; i<list.size(); i++)
   {  Object element = list.get(i);
      System.out.println("Element " + i + " is " + element);
   }
}
```

If the parameter were declared of type `ArrayList<Object>` then only collections of this type could be used, not other collections such as `ArrayList<String>`. Wildcard types can also be bounded, such as `ArrayList<? extends Vehicle>`, to ensure that the collection parameter does contain elements of a desired type or a subtype of it.

The Java Collections Framework API in the `java.util` package provides interfaces for some of the simpler abstract data types, such as `Collection`, `Set`, `SortedSet`, and `List`, as well as classes such as `ArrayList` and `LinkedList` which implement common data structures. The choice of the most suitable abstract data type, and the choice of an appropriate data structure to implement it depends on the nature of the data and how it will be used, and has a great impact on the efficiency of the program.

To facilitate the separation of how a collection is used from its implementation, an interface is either designed or is chosen from the Collections API to describe the operations that may be performed upon the collection. The `Collection` interface is used to represent any group of objects, that may or may not allow duplication of its elements. Since it is a generic interface that takes a type parameter, a collection can be restricted to contain a particular data type (rather than arbitrary objects). The Collections API does not provide any classes that directly implement this interface, but rather its more specific subinterfaces `Set`, `SortedSet`, and `List`. Instead the `Collection` interface is typically used to pass collections around and manipulate them where maximum generality is desired.

The `Collection` methods `add` and `addAll` are for adding one element (of type `E`) or an entire collection of elements (of a subtype of `E`) to the collection and returning whether the collection was changed, `clear` removes all the elements from the collection or throws a `UnsupportedOperationException` exception if



```
≪interface≫
Collection<E>

+add(o : E) : boolean
+addAll(c : Collection<? extends E>) : boolean
+clear() : void
+contains(o : Object) : boolean
+containsAll(c : Collection<?>) : boolean
+equals(o : Object) : boolean
+hashCode() : int
+isEmpty() : boolean
+iterator() : Iterator<E>
+remove(o : Object) : boolean
+removeAll(c : Collection<?>) : boolean
+retainAll(c : Collection<?>) : boolean
+size() : int
+toArray() : Object[]
+toArray(a : T[]) : T[]
```

this is not allowed, whereas `remove`, `removeAll`, and `retainAll` are used for removing any element or any collection (of whatever type, hence the wildcard `<?>`). The `Collection` method `size` returns the number of elements in the collection, whereas `contains` and `containsAll` are used to check whether the collection contains a specified element or all the elements in a specified collection.

The `Collection` method `iterator` returns an `Iterator` object of the appropriate type. The `Iterator` interface (which also allows generic types) provides a means to move through the collection one element at a time. The `Iterator` interface has three methods, `hasNext` which should return whether there is another element in the collection yet to be processed, `next` which should return the next element in the collection, and the method `remove` which should either remove the element returned by the most-recent `next` call or throw a `UnsupportedOperationException` if removing elements is not allowed.

```
public interface Iterator<E>
{
   public boolean hasNext();
   public E next() throws NoSuchElementException;
   public void remove() throws UnsupportedOperationException,
      IllegalStateException;
}
```

Iterators are typically used in a loop to iterate through all the elements in the collection, such as:

```
Collection<Vehicle> collection = ...;
Iterator<Vehicle> iterator = collection.iterator();
while (iterator.hasNext())
{  Vehicle vehicle = iterator.next();
   System.out.println(vehicle);
}
```

Actually, the `Collection` interface is a subinterface of another interface called

`Iterable` in the `java.lang` package. This enables a convenient *for-each* loop syntax to be used to iterate through the elements of any collection. Using this syntax the previous code fragment could be written simply as:

```
Collection<Vehicle> collection = ...;
for (Vehicle vehicle : collection)
{  System.out.println(vehicle);
}
```

The class `Collections` contains many convenient static methods for manipulating collections. After all the necessary elements have been added to some collection, the collection can be made read-only to prevent accidental modification by using the static method `unmodifiableCollection` (there are also methods `unmodifiableSet`, `unmodifiableSortedSet`, and `unmodifiableList` for sets, sorted sets, and lists), such as:

```
Collection<Vehicle> collection = ...;
collection.add(...);
collection.add(...);
collection = Collections.unmodifiableCollection(collection);
```

Any reference to the original modifiable collection should be removed, otherwise it could be used to modify the collection. Any attempt to modify a read-only collection would result in an `UnsupportedOperationException` exception.

Finally, classes in the Collections API that implement data structures, such as `ArrayList` and `LinkedList`, are not thread-safe by default. This was done since unsynchronized code executes much faster than synchronized code. If however a collection might be used by multiple threads in a program, then modifications to the collection need to be synchronized. The `Collections` class provides the ability to wrap an existing collection into a synchronized collection suitable for use by multiple threads with the methods `synchronizedCollection`, `synchronizedSet`, `synchronizedSortedSet`, or `synchronizedList`, such as:

```
List<Vehicle> vehicles
   = Collections.synchronizedList(new ArrayList<Vehicle>());
```

Any reference to the original unsynchronized collection should be removed, otherwise it could be used to access the collection in an unsynchronized manner.

Exercise 1.4 (Implementing an ADT) *Suppose an interface called* `RandomObtainable` *has been written for collections from which a random element should be obtainable. It has a method* `getRandom` *which should return an element randomly selected from the collection, and a method* `removeRandom` *that should try to remove a random element from the collection.*

```
public interface RandomObtainable<E>
{
   public E getRandom() throws NoSuchElementException;
   public boolean removeRandom() throws UnsupportedOperationException;
}
```

Extend the `ArrayList` *class so that it implements this interface, and test it with a driver* `main` *method.*

Chapter 2

Set Collections

2.1 Implementing Sets using Arrays

Reading: pp96-114

A *set* is a collection of elements which does not allow any duplicates (so an element cannot appear several times in a single set). The Collections API has an interface called `Set` which extends the `Collection` interface (actually it has the same methods) and describes the abstract data type for a set.

```
≪interface≫
Set<E>
----------------------------------------------------
----------------------------------------------------
+add(o : E) : boolean
+addAll(c : Collection<? extends E>) : boolean
+clear() : void
+contains(o : Object) : boolean
+containsAll(c : Collection<?>) : boolean
+equals(o : Object) : boolean
+hashCode() : int
+isEmpty() : boolean
+iterator() : Iterator<E>
+remove(o : Object) : boolean
+removeAll(c : Collection<?>) : boolean
+retainAll(c : Collection<?>) : boolean
+size() : int
+toArray() : Object[]
+toArray(a : T[]) : T[]
```

To write a class with a data structure for implementing sets it is convenient to extend the abstract class `AbstractSet`, which implements most of the methods in the `Set` interface, and just provide implementations of the methods `add`, `remove`, `iterator`, `size`, and `clear`.

One implementation strategy for sets is to use an array as the underlying data structure for storing the elements of the set. For example, the class `ArraySet` (similar to the class of the same name in the textbook) implements the `Set` interface (since it extends `AbstractSet`), and uses an array to store the elements of the set. It has a generic type `E` for the type of elements that it can store, so that a line such as:

```
Set<String> bookTitles = new ArraySet<String>();
```

would create a set collection for `String` objects, which actually get stored in a private `E[]` array. Since an array is a *fixed data structure* (the number of elements it holds can't be changed) and it is unknown how many elements will eventually be stored in the set, the array is initially given a default size `INITIAL_CAPACITY`. The variable `numElements` keeps track of how many elements are stored in the array, which get stored at indices `0`, `1`, ..., `numElements-1`, the remaining entries of the array are unused.

The `ArraySet` class has a default (no-parameter) constructor which creates an empty `E[]` array, and an alternative constructor which creates a set from an existing collection.

The inherited method `contains` works by obtaining an iterator via the `iterator` method, and repeatedly calling `next` until either the element is found, or else the iterator has no more elements. This will happen as least once, at most n times for a set of n elements, and on average requires $n/2$ calls of the `next` method for elements that are found in the set. Hence the `contains` method has time complexity $O(n)$.

The `add` method first checks whether the element to be added already exists in the set (since a set should never contain the same element twice) by using the method `contains`. Then it checks whether the array `elements` has the capacity to store another element:

```
if (numElements >= elements.length)
   expandCapacity();
```

If the array is already full, then a helper method called `expandCapacity` is used to allocate a new array with twice the capacity, necessary since the size of an array cannot be modified once created, copy the elements to the new array, and replace the old array by the new one. Since the order of the elements in the array is not relevant for a set, the new element is then just conveniently placed at the unused index `numElements` of the array. Although when the array is replaced by a new array of twice the size not all of the capacity might be needed, this ensures that the time consuming array allocation and copying is performed infrequently, which is an $O(n)$ process for a set with n elements. If instead, a new array of size only one larger were created, then every time an element were added the array would require re-allocation and copying, which is $O(n)$. Overall the `add` method has time complexity $O(n)$, due to its necessary use of the `contains` method.

The `remove` method first checks to see whether the element to be removed actually is in the set, taking care in the special case if it might be `null`:

```
int index = 0;
boolean found = false;
for (int i=0; i<numElements && !found; i++)
{  if ((elements[i]==null && o==null)
      || (elements[i]!=null && elements[i].equals(o)))
      {  index = i;
         found = true;
```

```
/**
   A class that implements a set collection using an
   array as the underlying data structure
   @author Andrew Ensor
*/
import java.util.AbstractSet;
import java.util.Collection;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class ArraySet<E> extends AbstractSet<E>
{
   private final int INITIAL_CAPACITY = 20;
   protected int numElements;
   protected E[] elements;

   // default constructor that creates a new set
   // that is initially empty
   public ArraySet()
   {  super();
      numElements = 0;
      elements = (E[])(new Object[INITIAL_CAPACITY]); // unchecked
   }

   // constructor for creating a new set which
   // initially holds the elements in the collection c
   public ArraySet(Collection<? extends E> c)
   {  this();
      for (E element : c)
      {  add(element);
      }
   }

   // adds the element to the set provided that it
   // is not already in the set, and returns
   // true if the set did not already contain the element
   public boolean add(E o)
   {  if (!(contains(o)))
      {  if (numElements >= elements.length)
            expandCapacity();
         elements[numElements++] = o;
         return true;
      }
      else
         return false;
   }
```

cont-

-cont

```
// remove the element from the set and returns true if the
// element was in the set
public boolean remove(Object o)
{  // search for the index of the element o in the set
   int index = 0;
   boolean found = false;
   for (int i=0; i<numElements && !found; i++)
   {  if ((elements[i]==null && o==null)
         || (elements[i]!=null && elements[i].equals(o)))
      {  index = i;
         found = true;
      }
   }
   if (found) // replace the element at index by last element
   {  elements[index] = elements[numElements-1];
      elements[numElements-1] = null; //removes reference to element
      numElements--;
   }
   return found;
}

// returns an iterator for iterating through the elements in the set
public Iterator<E> iterator()
{  return new ArrayIterator<E>(elements, numElements);
}

// returns the number of elements in the set
public int size()
{  return numElements;
}

// removes all elements from the set
public void clear()
{  for (int i=0; i<numElements; i++)
      elements[i] = null;
   numElements = 0;
}

// helper method which doubles the current size of the array
private void expandCapacity()
{  E[] largerArray =(E[])(new Object[elements.length*2]);//unchecked
   // copy the elements array to the largerArray
   for (int i=0; i<numElements; i++)
      largerArray[i] = elements[i];
   elements = largerArray;
}
```

cont-

-cont

```
   // inner class which represents an iterator for an array
   private class ArrayIterator<E> implements Iterator<E>
   {
      private int numElements; // number of elements in array to use
      private E[] elements; // the array to use
      private int nextIndex; // index of next element for the iterator

      // constructor which accepts an array of elements
      // and prepares an iterator which will iterate through the
      // first numElements elements of the array
      public ArrayIterator(E[] elements, int numElements)
      {  if (numElements > elements.length)
            numElements = elements.length;
         this.numElements = numElements;
         this.elements = elements;
         nextIndex = 0; // start with index of first element in array
      }

      // returns whether there is still another element
      public boolean hasNext()
      {  return (nextIndex<numElements);
      }

      // returns the next element or throws a NoSuchElementException
      // it there are no further elements
      public E next() throws NoSuchElementException
      {  if (!hasNext())
            throw new NoSuchElementException();
         return elements[nextIndex++];
      }

      // remove method not supported by this iterator
      public void remove() throws UnsupportedOperationException
      {  throw new UnsupportedOperationException();
      }
   }
}
```

```
/**
   A simple class which demonstrates how a set can be used
*/
import java.util.Set;

public class SetExample
{
   public static void main(String[] args)
   {  Set<String> bookTitles = new ArraySet<String>();
      bookTitles.add("Java Software Structures");
      bookTitles.add("Computer Graphics Using OpenGL");
      bookTitles.add("Java Software Structures");
      bookTitles.add("Introduction to Algorithms");
      bookTitles.add(null);
      bookTitles.remove("Computer Graphics Using OpenGL");
      System.out.println(bookTitles);
   }
}
```

```
         }
   }
```

At best the element is found after one iteration of the loop, at worst it is found after n (`numElements`) iterations, with on average $n/2$ iterations. Hence finding the element has time complexity $O(n)$. If the element is found then it could be removed by shifting all the elements after it in the array back one, but since this is $O(n)$ and the order of elements is not important in a set, the `remove` method simply replaces the removed element by the last element in the array:

```
if (found) // replace the element at index by last element
{  elements[index] = elements[numElements-1];
   elements[numElements-1] = null;
   numElements--;
}
```

which is an $O(1)$ process. Overall then the `remove` method operates in $O(n)$, due to the searching for the element in the set.

The `iterator` method returns an object which implements the `Iterator` interface with the assistance of an inner class called `ArrayIterator`. This inner class takes an array and the number of elements to iterate over, and uses a counter variable `nextIndex` to iterate through the elements of the array in order starting at index `0`, returning one element each time its `next` method is called.

The small class `SetExample` demonstrates how a set can be implemented using `ArraySet`. It is important to note that all the details of how the abstract set data type is implemented as an array are hidden from a user of the class. Also note that an attempt to add the same element (as determined by the object's `equals` method) several times to the set only results in it being added once.

If there are multiple threads that might access the same set and one of those threads might modify the set then the `add`, `remove`, and `clear` methods need to be synchronized (the `expandCapacity` would also cause difficulties but it is only

```
/**
   A class that demonstrates the problems with using unsynchronized
   add and remove operations of an ArraySet when there are multiple
   threads manipulating the elements
   @author Andrew Ensor
*/
import java.util.Set;

public class ArraySetJumbler implements Runnable
{
   private Set<Integer> set;

   public ArraySetJumbler()
   {  set = new ArraySet<Integer>();
      Thread threadA = new Thread(this);
      Thread threadB = new Thread(this);
      threadA.start();
      threadB.start();
   }

   // method which continually adds and removes a specific integer
   public void run()
   {  while (true) // infinite loop
      {  set.add(new Integer(1));
         set.remove(new Integer(1));
      }
   }

   public static void main(String[] args)
   {  new ArraySetJumbler();
   }
}
```

called from within the add method, so as long as that method is synchronized the expandCapacity shouldn't cause problems).

To illustrate what can go wrong, consider the example ArraySetJumbler. This class creates two threads which repeatedly add and remove the same element from an ArraySet. One might consider that although one thread might have already added or removed the element when the other thread also tries should not cause problems since the add and remove methods each first check whether the element is in the set. However, the problem is severe, typically the program suffers from a run-time exception such as:

```
Exception in thread "Thread-1" java.lang.ArrayIndexOutOfBoundsException: -1
        at ArraySet.remove(ArraySet.java:62)
        at ArraySetJumbler.run(ArraySetJumbler.java:25)
        at java.lang.Thread.run(Thread.java:595)
```

This is the result of one thread manipulating the array while the other was accessing it, the methods add and remove were written under the assumption that the array was not going to be changed while a thread was inside the method. This problem can be avoided by synchronizing these methods, either by using the SyncArraySet class or by using the Collections class:

```
/**
   A class which extends the ArraySet class so that its
   add, remove, and clear methods are synchronized
   @author Andrew Ensor
*/
import java.util.Collection;

public class SyncArraySet<E> extends ArraySet<E>
{
   public SyncArraySet()
   {  super();
   }

   public SyncArraySet(Collection<? extends E> c)
   {  super(c);
   }

   public synchronized boolean add(E o)
   {  return super.add(o);
   }

   public synchronized boolean remove(Object o)
   {  return super.remove(o);
   }

   public synchronized void clear()
   {  super.clear();
   }
}
```

```
set = Collections.synchronizedSet(new ArraySet<Integer>());
```

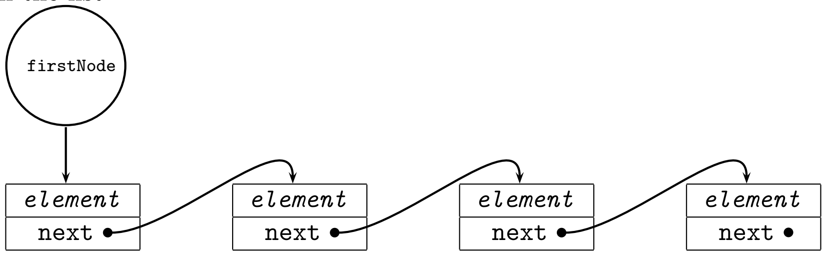
Exercise 2.1 (Testing The Time Complexity of `ArraySet`) *Write a class that creates a set holding the* `Integer` *elements* `0`*,* `1`*, ...,* `n-1` *for some constant value of* `n`*. Time how long the* `ArraySet` *class takes to add these* `n` *elements, how long it takes to find the element* `n/2`*, and how long it takes to remove this element. Try this for some different values of* `n` *and plot a graph of time versus* `n` *for each of the* `add`*,* `contains`*, and* `remove` *operations. The* `System` *method* `nanoTime` *might assist in timing the methods.*

2.2 Implementing Sets using Links

Reading: pp120-137

An alternative to using an array-based data structure is to use object references, or *links*, to join the elements of a collection together. Links can be used to create *dynamic data structures*, which can grow or shrink the number of elements that they hold.

Perhaps the simplest dynamic data structure is a *linked list*, which is commonly used to store a list of elements that can change in size. A linked list is a collection of *nodes*. Each node holds a reference to the element it stores as well as a link (object reference) to the next node in the list (or `null` if there are no more elements). Also a reference such as `firstNode` must be kept to the first node in the list



In order to implement a linked list data structure a class such as `Node` must be created to represent the nodes. In keeping with the philosophy of hiding how a collection is implemented, typically the `Node` class is made a private inner class with a generic type for the type of elements it links together. That way the enclosing class can access the `element` field and the `next` link, which are not seen outside that class, without the need to use accessor methods.

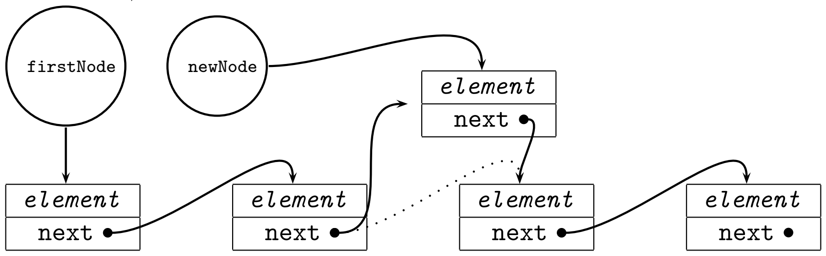
```
private class Node<E>
{
   public E element;
   public Node<E> next;

   public Node(E element)
   {  this.element = element;
      next = null;
   }
}
```

In order to create an initially empty linked list one simply initializes the reference to the start of the list:

```
firstNode = null;
```

There are three common algorithms associated with linked lists, inserting new nodes, searching nodes, and removing nodes.

To insert a new node in a list at a certain position requires that two links get modified. The new node must link to the node that will come immediately after it in the list, and the node before the new node must link to it.



Inserting a new node at the start of the list requires that the `firstNode` reference gets modified, and so is treated as a special case. The following typical code fragment shows how to insert a new node in a singly-linked list immediately after a node called `previous`:

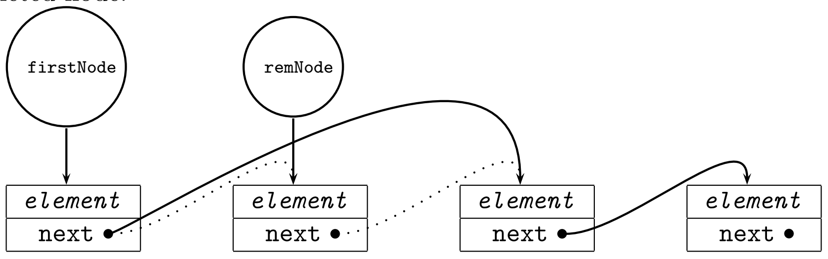
```
if (previous == null)
{  // insert newNode at start of list
   newNode.next = firstNode;
   firstNode = newNode;
}
else
{  // insert newNode after previous
   newNode.next = previous.next;
   previous.next = newNode;
}
```

Note the order to the steps in each case, it is important that the new node is linked to the node after it before that reference is lost. If the position to insert the new node is known, the above code has complexity $O(1)$. Also note that if there are multiple threads manipulating a linked list the above code fragment needs to be synchronized (otherwise one thread might be accessing the list while another thread is part way through modifying the links).

To search a singly-linked list for a node that holds a particular element such as `target` requires an $O(n)$ loop:

```
boolean found = false;
Node<E> current = firstNode;
while (current!=null && !found)
{  if (target.equals(current.element))
      found = true;
   else
      current = current.next; // move along to next node
}
```

To remove a node from the list requires that the link from the node immediately before the deleted node be made to refer to the node immediately after the deleted node.



Removing a new node at the start of the list also requires that the `firstNode` reference gets modified, and so is treated as a special case. The following typical code fragment shows how to remove the node after `previous` in a singly-linked list (presuming there is such a node):

```
if (previous == null)
{  // remove node at start of list
   firstNode = firstNode.next;
}
else
{  // remove the node after previous (node previous.next)
```

```
      previous.next = previous.next.next;
   }
```

Typically however, removing a node also requires searching for the node to be removed, as is done in the `remove` method of the following example, `LinkedSet`.

The class `LinkedSet` (similar to the class of the same name in the textbook) implements the `Set` interface (since it extends `AbstractSet`), similarly to the `ArraySet` example, but it uses a singly-linked list rather than an array to store the elements of the set. It too has a generic type `E` for the type of elements that it can store, and is used by a line such as:

```
   Set<String> bookTitles = new LinkedSet<String>();
```

Apart from the name of the class, a set created from the `LinkedSet` class should behave the same as a set created from the `ArraySet` class, even though the data structures used are quite distinct.

Since the order of elements in a set is not important, the `add` method of `LinkedSet` simply adds a new node to the start of the list (this saves having to maintain a reference to the end of the list, or finding the end when adding). One consequence of this is that the `toString` method for the set shows the elements in the reverse order to which they were added to the set. Since the `add` method must first check whether the element already exists in the set, the method has $O(n)$.

The `remove` method first checks whether the element to be removed is at the start of the list, otherwise it searches for the element by following the links of the list, keeping track of the current and previous node as it progresses, and removes the appropriate node if it is found:

```
   Node<E> previous = firstNode;
   Node<E> current = firstNode.next;
   while (current!=null && !found)
   {  if ((current.element==null && o==null) ||
         (current.element!=null && current.element.equals(o)))
      {  found = true;
         previous.next = current.next;
         numElements--;
      }
      else
      {  previous = current;
         current = current.next;
      }
   }
```

Due to this loop the `remove` method in `LinkedSet` is $O(n)$.

The Collections API actually includes two classes, `HashSet` and `TreeSet`, which provide two further data structure implementations of sets. The class `HashSet` makes use of a hash code for each element (see Section 5.1), and offers constant time performance for the basic operations `add`, `contains`, and `remove` (assuming the hash function properly disperses the hash codes). The class `TreeSet` uses a tree data structure (see Section 4.2) for elements which implement the `Comparable` interface. This implementation provides guaranteed $O(\log_2 n)$ time for the basic operations `add`, `contains`, and `remove`.

```
/**
   A class that implements a set collection using a
   singly-linked list as the underlying data structure
   @author Andrew Ensor
*/
import java.util.AbstractSet;
import java.util.Collection;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class LinkedSet<E> extends AbstractSet<E>
{
   protected int numElements;
   protected Node<E> firstNode;

   // default constructor that creates a new set
   // that is initially empty
   public LinkedSet()
   {  super();
      numElements = 0;
      firstNode = null;
   }

   // constructor for creating a new set which
   // initially holds the elements in the collection c
   public LinkedSet(Collection<? extends E> c)
   {  this();
      for (E element : c)
      {  add(element);
      }
   }

   // adds the element to the set provided that it
   // is not already in the set, and returns
   // true if the set did not already contain the element
   public boolean add(E o)
   {  if (!(contains(o)))
      {  Node<E> newNode = new Node<E>(o);
         // add the new node to the front of the list
         newNode.next = firstNode;
         firstNode = newNode;
         numElements++;
         return true;
      }
      else
         return false;
   }
```

cont-

-cont

```
   // remove the element from the set and returns true if the
   // element was in the set
   public boolean remove(Object o)
   {  // search for the node of the element o in the set
      boolean found = false;
      if (firstNode!=null)
      {  // check if the element is first in list
         if ((firstNode.element==null && o==null) ||
            (firstNode.element!=null && firstNode.element.equals(o)))
         {  found = true;
            firstNode = firstNode.next;
            numElements--;
         }
         else
         {  // check the other nodes in the list
            Node<E> previous = firstNode;
            Node<E> current = firstNode.next;
            while (current!=null && !found)
            {  if ((current.element==null && o==null) ||
                  (current.element!=null && current.element.equals(o)))
               {  found = true;
                  previous.next = current.next;
                  numElements--;
               }
               else
               {  previous = current;
                  current = current.next;
               }
            }
         }
      }
      return found;
   }

   // returns an iterator for iterating through the elements in the set
   public Iterator<E> iterator()
   {  return new LinkedIterator<E>(firstNode);
   }

   // returns the number of elements in the set
   public int size()
   {  return numElements;
   }
```

cont-

-cont

```
   // removes all elements from the set
   public void clear()
   {  firstNode = null;
      numElements = 0;
   }

   // inner class which represents a node in a singly-linked list
   protected class Node<E>
   {
      public E element;
      public Node<E> next;

      public Node(E element)
      {  this.element = element;
         next = null;
      }
   }
```

cont-

To compare the various implementations of sets, one can time say the `add` method on a typical processor (1600MHz Pentium). For a set with $n = 1000$ elements, adding a new element to an `ArraySet` takes approximately 37 microseconds (this number can vary at times depending on the operating system load, and whether a garbage collection occurs during timing), whereas a `LinkedList` takes slightly more, 38 microseconds (it takes a bit more time to pass through the links in a linked list than an array, but a linked list does not waste memory on extra capacity). Both classes show linear time complexity $O(n)$, so adding a new element to a set with $n = 2000$ elements takes about double the time. On the other hand, the Collection API class `TreeSet` took just 3.9 microseconds to add a new element to a set with $n = 1000$ elements, and this time grows slowly, $O(\log_2 n)$, as n increase. Best of all, a `HashSet` took just 2.2 microseconds, a time which did not alter significantly as n was increased.

Exercise 2.2 (Inserting in a Linked List) *Extend the* `LinkedSet` *class to a class called* `LinkedSortedSet`*, whose* `add` *method inserts a* `Comparable` *element in the correct position to maintain the linked list sorted (using each element's* `compareTo` *method).*

-cont

```
   // inner class which represents an iterator for a singly-linked list
   private class LinkedIterator<E> implements Iterator<E>
   {
      private Node<E> nextNode; // next node to use for the iterator

      // constructor which accepts a reference to first node in list
      // and prepares an iterator which will iterate through the
      // entire linked list
      public LinkedIterator(Node<E> firstNode)
      {  nextNode = firstNode; // start with first node in list
      }

      // returns whether there is still another element
      public boolean hasNext()
      {  return (nextNode!=null);
      }

      // returns the next element or throws a NoSuchElementException
      // it there are no further elements
      public E next() throws NoSuchElementException
      {  if (!hasNext())
            throw new NoSuchElementException();
         E element = nextNode.element;
         nextNode = nextNode.next;
         return element;
      }

      // remove method not supported by this iterator
      public void remove() throws UnsupportedOperationException
      {  throw new UnsupportedOperationException();
      }
   }
}
```

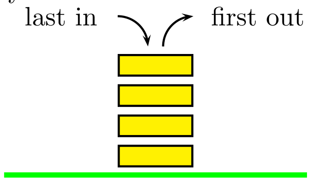
Chapter 3

Linear Collections

3.1 Stacks

Reading: pp178-205

A *linear collection* is a collection in which the elements are placed in some total ordering, typically either by the order in which they were added to the collection, or alternatively by an order determined by the elements themselves.



A *stack* is a linear collection whose elements are added and removed so that the most recent element added is the first to be removed, such as a stack of magazines on a table. Often stacks are described as *last in, first out* lists and the elements are visualized arranged vertically with the most recent addition on top.

The operations that are typically permitted on a stack are:

push for adding an element to the top of the stack,

pop for removing the element that is on top of the stack,

peek (or top) for examining the element that is on top of the stack,

isEmpty for determining whether the stack is empty,

size for determining the number of elements on the stack.

Usually a stack can only be manipulated by pushing elements on or popping elements off the top of the stack (there are no general add or remove operations). The abstract data type for a stack can be described by the `StackADT` interface (note there is no interface for a stack abstract data type included in the Collections API).

As with sets there are two common alternative implementation strategies for stacks, either implemented using an array which gets replaced by a larger array as required, or using a linked list.

The example `ArrayStack` uses an array as its data structure and is quite similar to the `ArraySet` class, using a counter to keep track of the portion of

```
/**
   An interface that defines the abstract data type
   for a stack collection of elements with type E
*/
import java.util.NoSuchElementException;

public interface StackADT<E>
{
   // adds one element to the top of this stack
   public void push(E element);
   // removes and returns the top element from this stack
   public E pop() throws NoSuchElementException;
   // returns without removing the top element of this stack
   public E peek() throws NoSuchElementException;
   // returns true if this stack contains no elements
   public boolean isEmpty();
   // returns the number of elements in this stack
   public int size();
}
```

the array that currently holds elements. However, `ArrayStack` does not provide an iterator for iterating through all the elements in the stack, since in keeping with the abstract data type for a stack only the top element should be accessible (but it does provide a `toString` method which returns the contents of the stack from top to bottom). Note in this example the choice of placing the bottom of the stack at index `0`, which is far more efficient than placing the top at index `0` (otherwise all the elements in the array would need to be shifted up one index whenever a new element were added - an $O(n)$ operation).

The `push` method first checks whether there is enough capacity to add another element to the array, and if not it creates a new array of double the capacity and copies the original array to the new array (an $O(n)$ operation when required). Then it places the new element into the array at the next available index. The `pop` method simply sets the array reference to the last element in the array to `null` and decreases the counter (one should always release the unused reference to the element popped, otherwise it might remain in the array and prevent the element from being marked for garbage collection until the array itself is garbage collected). Apart from the infrequent calls to increase the capacity of the data structure, the array implementation has $O(1)$ for both its `push` and `pop` operations.

The example `LinkedStack` uses a singly-linked list as its data structure and is similar to `LinkedSet` in design. Here the choice was made to put the top of the stack at the start of the list, which saves having to keep a reference to the end of the list and (more importantly) saves having to traverse through the entire list to the penultimate element when an element is popped.

With the linked list implementation of a stack, when an element is pushed on the stack it is simply placed at the front of the linked list, so the `push` method is $O(1)$. Similarly, to pop the top element off the stack just requires changing the reference to the front of the list, so the `pop` method is also $O(1)$.

Note that for both implementations of the stack abstract data type, the `push` and `pop` operations are $O(1)$, so their efficiency is independent of the number of

```
/**
   A class that implements a stack collection using an
   array as the underlying data structure
   @author Andrew Ensor
*/
import java.util.Collection;
import java.util.NoSuchElementException;

public class ArrayStack<E> implements StackADT<E>
{
   private final int INITIAL_CAPACITY = 20;
   protected int numElements;
   protected E[] elements;

   // default constructor that creates a new stack
   // that is initially empty
   public ArrayStack()
   {  numElements = 0;
      elements = (E[])(new Object[INITIAL_CAPACITY]); // unchecked
   }

   // constructor for creating a new stack which
   // initially holds the elements in the collection c
   public ArrayStack(Collection<? extends E> c)
   {  this();
      for (E element : c)
         push(element);
   }

   // Adds one element to the top of this stack
   public void push(E element)
   {  if (numElements >= elements.length)
         expandCapacity();
      elements[numElements++] = element;
   }

   // removes and returns the top element from this stack
   public E pop() throws NoSuchElementException
   {  if (numElements > 0)
      {  E topElement = elements[numElements-1];
         elements[numElements-1] = null;
         numElements--;
         return topElement;
      }
      else
         throw new NoSuchElementException();
   }
```

cont-

-cont

```
   // returns without removing the top element of this stack
   public E peek() throws NoSuchElementException
   {  if (numElements > 0)
         return elements[numElements-1];
      else
         throw new NoSuchElementException();
   }

   // returns true if this stack contains no elements
   public boolean isEmpty()
   {  return (numElements==0);
   }

   // returns the number of elements in this stack
   public int size()
   {  return numElements;
   }

   // returns a string representation of the stack from top to bottom
   public String toString()
   {  String output = "[";
      for (int i=numElements-1; i>=0; i--)
      {  output += elements[i];
         if (i>0)
            output += ",";
      }
      output += "]";
      return output;
   }

   // helper method which doubles the current size of the array
   private void expandCapacity()
   {  E[] largerArray =(E[])(new Object[elements.length*2]);//unchecked
      // copy the elements array to the largerArray
      for (int i=0; i<numElements; i++)
         largerArray[i] = elements[i];
      elements = largerArray;
   }
}
```

```
/**
   A class that implements a stack collection using a
   singly-linked list as the underlying data structure
   @author Andrew Ensor
*/
import java.util.Collection;
import java.util.NoSuchElementException;

public class LinkedStack<E> implements StackADT<E>
{
   protected int numElements;
   protected Node<E> firstNode; // front of list is top of stack

   // default constructor that creates a new stack
   // that is initially empty
   public LinkedStack()
   {  numElements = 0;
      firstNode = null;
   }

   // constructor for creating a new stack which
   // initially holds the elements in the collection c
   public LinkedStack(Collection<? extends E> c)
   {  this();
      for (E element : c)
         push(element);
   }


   // Adds one element to the top of this stack
   public void push(E element)
   {  Node<E> newNode = new Node<E>(element);
      // add the new node to the front of the list
      newNode.next = firstNode;
      firstNode = newNode;
      numElements++;
   }

   // removes and returns the top element from this stack
   public E pop() throws NoSuchElementException
   {  if (firstNode != null)
      {  E topElement = firstNode.element;
         firstNode = firstNode.next;
         numElements--;
         return topElement;
      }
      else throw new NoSuchElementException();
   }
```

cont-

-cont

```
   // returns without removing the top element of this stack
   public E peek() throws NoSuchElementException
   {  if (numElements > 0)
         return firstNode.element;
      else throw new NoSuchElementException();
   }

   // returns true if this stack contains no elements
   public boolean isEmpty()
   {  return (numElements==0);
   }

   // returns the number of elements in this stack
   public int size()
   {  return numElements;
   }

   // returns a string representation of the stack from top to bottom
   public String toString()
   {  String output = "[";
      Node currentNode = firstNode;
      while (currentNode != null)
      {  output += currentNode.element;
         if (currentNode.next != null)
            output += ",";
         currentNode = currentNode.next;
      }
      output += "]";
      return output;
   }

   // inner class which represents a node in a singly-linked list
   protected class Node<E>
   {
      public E element;
      public Node<E> next;

      public Node(E element)
      {  this.element = element;
         next = null;
      }
   }
}
```

elements on the stack. The array implementation does not require the additional space per element for a link to the next element, but might potentially be wasteful of memory since it could allocate up to twice the amount of memory required for the collection.

Stacks are useful for evaluating arithmetic expressions. Arithmetic expressions are typically written by humans using *infix* notation, where a binary operator such as addition is written between its two operands. This notation requires the use of parentheses to alter the usual precedence rules, such as:

$$(1+2)\times(3-4\times 5)$$

which evaluates to -51. The *postfix* notation is an alternative notation where each operator is written after its operands, such as:

$$1 \quad 2 \quad + \quad 3 \quad 4 \quad 5 \quad \times \quad - \quad \times$$

It is usually easier to evaluate expressions written in postfix notation since rules of operator precedence don't apply and parentheses are not required. Postfix notation is widely used by compilers and runtime environments, and in computer languages such as postscript (a graphics language used by most laser printers).

An expression written in postfix notation can be easily evaluated with the assistance of a stack. Parsing the expression from left to right, each operand is pushed onto the stack as it is encountered. Whenever an operator is encountered the correct number of operands are popped off the stack (two for binary operators such as addition, one for unary operators such as inverse) and used by the operator, with the result pushed back on the stack. If there are insufficient elements on the stack then the expression is invalid, otherwise when completed the final result is on the top of the stack. For example, the expression:

$$1 \quad 2 \quad + \quad 3 \quad 4 \quad 5 \quad \times \quad - \quad \times$$

would result in the following stack being formed:

					5			
				4	4	20		
	2		3	3	3	3	−17	
1	1	3	3	3	3	3	3	−51

The class `PostfixEvaluator` demonstrates how a stack collection is used to evaluate a postfix expression obtained from the keyboard. This class also uses a set collection to store the allowable operators, which are represented by instances of the `Operator` class and its subclasses `BinaryOperator` and `UnaryOperator`.

Stacks are also fundamental in other areas of computer science. For example, the undo feature in an application typically uses a stack to keep track of changes. When changes needs to be undone each is popped off the stack and reversed (in the opposite order to which they were pushed on the stack), so that the most recent change is reversed first.

Also, whenever a program is running a *program stack* is used to keep information called an *activation record* for each method that has been called (holding information such as the current values of local variables). When one method finishes its activation record is popped off the program stack and execution returned to the method whose activation record is now on top (the calling method). The *call stack trace* is just a string representation of this stack, and

```
/**
   A class which demonstrates how a stack is used to
   parse a postfix expression
   @author Andrew Ensor
*/
import java.util.Iterator;
import java.util.HashSet;
import java.util.NoSuchElementException;
import java.util.Scanner; // Java 1.5 equivalent of cs1.Keyboard
import java.util.Set;
import java.util.StringTokenizer;

public class PostfixEvaluator
{
   private Set<Operator> operators;

   public PostfixEvaluator()
   {  // create the valid operators for this evaluator
      operators = new HashSet<Operator>();
      Operator addition = new BinaryOperator("+")
         {  public double evaluate(double op1, double op2)
            {  return op1+op2;
            }
         };
      Operator subtraction = new BinaryOperator("-")
         {  public double evaluate(double op1, double op2)
            {  return op1-op2;
            }
         };
      Operator multiplication = new BinaryOperator("*")
         {  public double evaluate(double op1, double op2)
            {  return op1*op2;
            }
         };
      Operator division = new BinaryOperator("/")
         {  public double evaluate(double op1, double op2)
            {  return op1/op2;
            }
         };
      Operator power = new BinaryOperator("^")
         {  public double evaluate(double op1, double op2)
            {  return Math.pow(op1, op2);
            }
         };
```

cont-

-cont

```
      Operator inverse = new UnaryOperator("inv")
         {  public double evaluate(double op)
            {  return 1.0/op;
            }
         };
      operators.add(addition);
      operators.add(subtraction);
      operators.add(multiplication);
      operators.add(division);
      operators.add(power);
      operators.add(inverse);
   }

   // parse the string expression which is presumed to
   // be an expression in postfix notation with double value
   // operands and string operators all separated by spaces
   public double parse(String expression)
      throws NoSuchElementException, NumberFormatException
   {  StringTokenizer tokenizer = new StringTokenizer(expression);
      StackADT<Double> operandStack = new ArrayStack<Double>();
      while (tokenizer.hasMoreTokens())
      {  String token = tokenizer.nextToken();
         // check whether token is the text for an operator
         Operator operator = getOperator(token);
         if (operator instanceof UnaryOperator)
         {
            double op = operandStack.pop().doubleValue();
            double result = ((UnaryOperator)operator).evaluate(op);
            operandStack.push(new Double(result));
         }
         else if (operator instanceof BinaryOperator)
         {  double op2 = operandStack.pop().doubleValue();
            double op1 = operandStack.pop().doubleValue();
            double result
               = ((BinaryOperator)operator).evaluate(op1, op2);
            operandStack.push(new Double(result));
         }
         else if (operator == null)
         {  // try to push a double value onto stack
            operandStack.push(new Double(Double.parseDouble(token)));
         }
      }
      return operandStack.pop().doubleValue();
   }
```

cont-

-cont

```
   // helper method which searches through the operators set for an
   // operator with the specified text and returns null if none found
   private Operator getOperator(String text)
   {  Iterator<Operator> iterator = operators.iterator();
      while (iterator.hasNext())
      {  Operator nextOperator = iterator.next();
         if (nextOperator.getText().equalsIgnoreCase(text))
            return nextOperator;
      }
      return null; // no operator found for this text
   }

   public static void main(String[] args)
   {  // obtain postfix expression from keyboard
      Scanner keyboardScanner = new Scanner(System.in);
      System.out.print("Please enter a postfix expression:");
      String expression = keyboardScanner.nextLine();
      PostfixEvaluator evaluator = new PostfixEvaluator();
      try
      {  double result = evaluator.parse(expression);
         System.out.println("The result is " + result);
      }
      catch (NoSuchElementException e)
      {  System.out.println("Insufficient operands: "+e.getMessage());
      }
      catch (NumberFormatException e)
      {  System.out.println("Illegal value: " + e.getMessage());
      }
   }
}
```

```
/**
  An abstract class which represents an Operator
  @see PostfixEvaluator.java
*/

public abstract class Operator
{
   private String text; // text representation of operator

   public Operator(String text)
   {  if (text == null)
         this.text = "unknown";
      else
         this.text = text;
   }

   public final String getText()
   {  return text;
   }
}
```

```
/**
   A class which represents a binary operator
   @see Operator.java
*/

public abstract class BinaryOperator extends Operator
{
   public BinaryOperator(String text)
   {  super(text);
   }

   public abstract double evaluate(double op1, double op2);
}
```

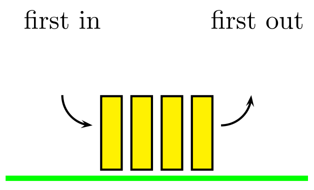
```
/**
   A class which represents a unary operator
   @see Operator.java
*/

public abstract class UnaryOperator extends Operator
{
   public UnaryOperator(String text)
   {  super(text);
   }

   public abstract double evaluate(double op);
}
```

when the stack becomes empty the program has completed. The program stack is what makes recursion possible as one method can have several activation records on the stack at a time. In languages that did not support recursion stacks were used in this way to simulate recursion.

Exercise 3.1 (Inefficient Stack Implementation) *Rewrite the* `ArrayStack` *class so that the top rather than the bottom of the stack is maintained at index* `0` *(this will involve shifting elements along whenever an element is pushed or popped). Compare the performance of your class with the* `ArrayStack` *class.*

3.2 Queues

Reading: pp212-222,227-242



A *queue* is a linear collection whose elements are added and removed so that the first element added is the first to be removed, such as a queue of waiting people. Often queues are described as *first in, first out* lists and the elements are visualized arranged horizontally with the first addition at the front.

The operations that are typically permitted on a queue are:

enqueue (or offer) for adding an element to the rear of the queue,

dequeue (or remove) for removing the element that is at the front of the queue,

first (or peek) for examining the element that is at the front of the queue,

isEmpty for determining whether the queue is empty,

size for determining the number of elements in the queue.

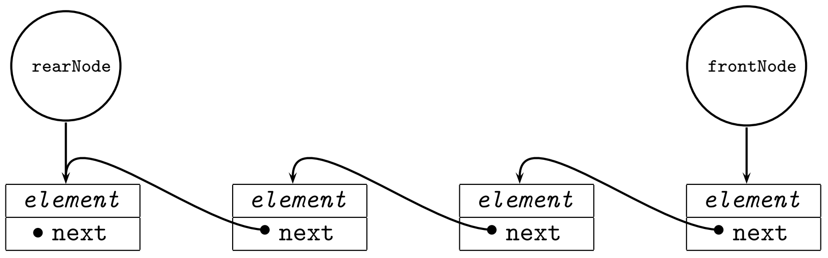
Note the differences in processing with a queue as opposed to with a stack. With a stack the most recent element added to the stack is available first, and then the second most recently added, etcetera, so that elements are obtained in the reverse order to which they were added. Instead, with a queue the elements are obtained in the same order in which they were added to the queue.

The abstract data type for a queue can be described by the `QueueADT` interface (note there is no interface for a queue abstract data type in the Collections API).

A queue can be easily implemented using a singly-linked list, analogously to stacks. Since elements are added to the linked list at one end and removed from the other, it is convenient (and efficient) to keep a reference to each end of the linked list, which was not required for the stack implementation. The choice as to whether to link the nodes so that each node references the one ahead of it in the queue (so that the front of the queue is at the end of the linked list) or alternatively the one behind it (so that the front of the queue is at the start of the linked list) has important implications for the efficiency of the queue. If each node were to reference the one ahead in the queue then the `enqueue` operation could be simply performed by adding a node to the start of the linked list (order

```
/**
   An interface that defines the abstract data type
   for a queue collection of elements with type E
*/
import java.util.NoSuchElementException;

public interface QueueADT<E>
{
   // adds one element to the rear of this queue
   public void enqueue(E element);
   // removes and returns the front element of the queue
   public E dequeue() throws NoSuchElementException;
   // returns without removing the front element of this queue
   public E first() throws NoSuchElementException;
   // returns true if this queue contains no elements
   public boolean isEmpty();
   // returns the number of elements in this queue
   public int size();
}
```

$O(1)$). However, the `dequeue` would have the complication of having to find the penultimate node in the linked list, requiring an $O(n)$ loop. The alternative, linking each node to the node behind it in the queue avoids this requirement.



This approach is used in the `LinkedQueue` class, so that both the `enqueue` and `dequeue` methods are $O(1)$. To enqueue an element a new node is created and placed at the end of the linked list by using the `rearNode` reference. To dequeue an element the `frontNode` reference is used and advanced to the next node in the linked list.

Implementing a queue with an array data structure is more problematic. If the front of the queue were maintained at index `0` of the array then whenever an element were dequeued the remaining elements would each need to be shifted down one index, making `dequeue` an $O(n)$ operation. If instead the rear of the queue were maintained at index `0` then the remaining elements would each need to be shifted up one index whenever an element is enqueued. One solution would be to allow the front and rear of the queue to move progressively forward through the array as elements are dequeued and enqueued. However, one consequence of this design strategy is that the array would eventually be expanded to have capacity to hold all the elements that have passed through the queue, wasting memory resources, even though the queue might only hold a few elements at a time.

One solution to this problem is to use a *circular array.* A circular array is really just an array where conceptually the array is considered to have no end,

```
/**
   A class that implements a queue collection using a
   singly-linked list as the underlying data structure
   @author Andrew Ensor
*/
import java.util.Collection;
import java.util.NoSuchElementException;

public class LinkedQueue<E> implements QueueADT<E>
{
   private int numElements;
   private Node<E> frontNode; // front of list is front of queue
   private Node<E> rearNode; // rear of list is rear of queue

   // default constructor that creates a new queue
   // that is initially empty
   public LinkedQueue()
   {  super();
      numElements = 0;
      frontNode = null;
      rearNode = null;
   }

   // constructor for creating a new queue which
   // initially holds the elements in the collection c
   public LinkedQueue(Collection<? extends E> c)
   {  this();
      for (E element : c)
         enqueue(element);
   }


   // Adds one element to the rear of this queue
   public void enqueue(E element)
   {  Node<E> newNode = new Node<E>(element);
      // add the new node to the end of the list
      if (rearNode==null)
      {  frontNode = newNode;
         rearNode = newNode;
      }
      else
      {  rearNode.next = newNode;
         rearNode = newNode;
      }
      numElements++;
   }
```

cont-

-cont

```
   // removes and returns the front element from this queue
   public E dequeue() throws NoSuchElementException
   {  if (frontNode != null)
      {  E frontElement = frontNode.element;
         frontNode = frontNode.next;
         numElements--;
         if (numElements == 0)
            rearNode = null;
         return frontElement;
      }
      else
         throw new NoSuchElementException();
   }

   // returns without removing the front element of this queue
   public E first() throws NoSuchElementException
   {  if (numElements > 0)
         return frontNode.element;
      else
         throw new NoSuchElementException();
   }

   // returns true if this queue contains no elements
   public boolean isEmpty()
   {  return (numElements==0);
   }

   // returns the number of elements in this queue
   public int size()
   {  return numElements;
   }

   // returns a string representation of the queue from front to rear
   public String toString()
   {  String output = "[";
      Node currentNode = frontNode;
      while (currentNode != null)
      {  output += currentNode.element;
         if (currentNode.next != null)
            output += ",";
         currentNode = currentNode.next;
      }
      output += "]";
      return output;
   }
```

cont-

-cont

```
   // inner class which represents a node in a singly-linked list
   private class Node<E>
   {
      public E element;
      public Node<E> next;

      public Node(E element)
      {  this.element = element;
         next = null;
      }
   }
}
```

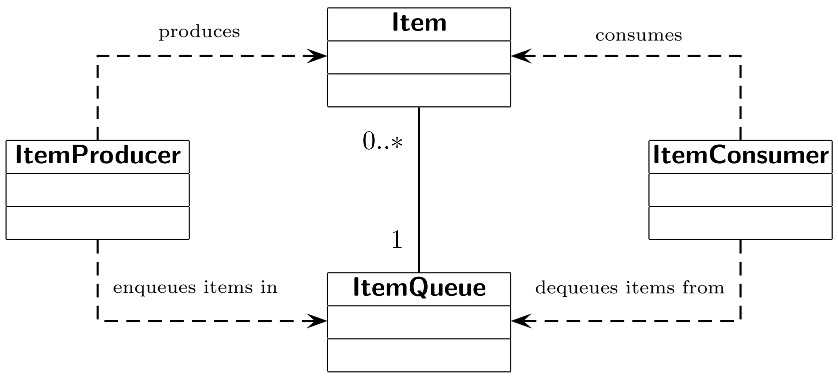
with indices that wrap around rather than just start at `0` and end at the highest index. If an element is placed at the last index in the array then the next element is placed back at index `0`, provided that index `0` is not currently used. To treat an array as a circular array two indices are required, one to represent the start of where elements are stored in the array, and the other to represent the end of the elements. The start index is incremented by one every time an element is dequeued and the end index is incremented every time an element is enqueued, but because one end of the array wraps around to the other end both indices need to be taken modulo the length of the array (using the remainder operator `%`) when incremented. Thus it is possible for the start index to be greater than the end index (with the elements straddling both ends of the actual array).

A common application of queues is to temporarily store items while waiting for them to be processed. This is often called the *producer/consumer problem*, where a producer is producing items and placing them at the rear of a queue, and one or more consumers obtain the items from the front of the queue. Using a queue enables the producer and consumers to work independently of one another. This technique is actually used by the event dispatch thread, which receives events from the user and stores them in a queue until it is able to consume (process) them (a queue is used rather than a stack so that the event that has waited the longest is the next to be processed).

The class `ProducerConsumer` demonstrates the producer/consumer problem using one producer and several consumers, each working independently of one another in their own thread. A single `ItemProducer` object produces `Item` objects at regular intervals and places them in a queue represented by a `CloseableQueue` object. When the producer has completed production, it closes the queue by telling it not to accept any further items, and the producer thread then dies. Each `ItemConsumer` object runs in its own thread, repeatedly checking whether there is an item at the front of the queue, either consuming it or else sleeping for a short period while waiting for an item to appear. Note that the `CloseableQueue` class has synchronized `enqueue` and `dequeue` methods to avoid the possibility of a race condition in the queue between two consumers or between the producer and a consumer.

A slightly more sophisticated approach to the producer/consumer example is to have the consumer threads wait when there are no items in the queue (putting each of them in a blocked state rather than having them check the queue

```
/**
   A class which starts a producer and some consumers of items
   @author Andrew Ensor
*/

public class ProducerConsumer
{
   private static final int NUM_CONSUMERS = 3;

   public static void main(String[] args)
   {  CloseableQueue<Item> queue = new CloseableQueue<Item>();
      // create the producer and consumers
      ItemProducer producer = new ItemProducer(queue);
      ItemConsumer[] consumers = new ItemConsumer[NUM_CONSUMERS];
      for (int i=0; i<NUM_CONSUMERS; i++)
         consumers[i] = new ItemConsumer(queue);
      // create the threads for producer and each consumer
      Thread producerThread = new Thread(producer);
      Thread[] consumerThreads = new Thread[NUM_CONSUMERS];
      for (int i=0; i<NUM_CONSUMERS; i++)
         consumerThreads[i] = new Thread(consumers[i]);
      // start all the threads
      producerThread.start();
      for (int i=0; i<NUM_CONSUMERS; i++)
         consumerThreads[i].start();
   }
}
```



```
/**
   A class which extends the LinkedQueue class so that its
   enqueue and dequeue methods are synchronized
   and which can be closed to prevent it from further
   enqueues of elements
   @author Andrew Ensor
*/
import java.util.NoSuchElementException;
import java.nio.ReadOnlyBufferException;

public class CloseableQueue<E> extends LinkedQueue<E>
{
   private boolean accepting; //whether queue currently allows enqueues

   public CloseableQueue()
   {  super();
      accepting = true;
   }

   public synchronized void enqueue(E element)
      throws ReadOnlyBufferException
   {  if (accepting)
         super.enqueue(element);
      else
         throw new ReadOnlyBufferException();
   }

   public synchronized E dequeue() throws NoSuchElementException
   {  return super.dequeue();
   }

   public boolean isAccepting()
   {  return accepting;
   }

   public void stopAccepting()
   {  accepting = false;
   }
}
```

```
/**
   A class which represents a producer of some items
   that it enqueues in a queue
   @see CloseableQueue.java, Item.java
*/

public class ItemProducer implements Runnable
{
   private CloseableQueue<Item> queue;
   private int idNumber;
   private final int PRODUCTION_QUANTITY = 10; // num items to produce
   private final int PRODUCTION_TIME = 150; // time to produce an item

   public ItemProducer(CloseableQueue<Item> queue)
   {  this.queue = queue;
      idNumber = 0;
   }

   // produce an item that takes time PRODUCTION_TIME
   private Item produceItem()
   {  Item item = new Item("ID number: " + idNumber);
      try
      {  Thread.sleep(PRODUCTION_TIME);
      }
      catch (InterruptedException e)
      {}
      idNumber++;
      return item;
   }

   public void run()
   {  System.out.println("Producer starting");
      for (int i=0; i<PRODUCTION_QUANTITY; i++)
      {  Item item = produceItem();
         System.out.println("Item " + item + " produced");
         // put item in queue
         queue.enqueue(item);
      }
      queue.stopAccepting();
      System.out.println("Producer stopping");
   }
}
```

```
/**
   A class which represents a consumer of some items
   that it dequeues from an item queue
   @see CloseableQueue.java, Item.java
*/
import java.util.NoSuchElementException;
import java.util.Random;

public class ItemConsumer implements Runnable
{
   private CloseableQueue<Item> queue;
   private Random generator;
   private String consumerName; // letter starting at A
   private static int numConsumers = 0; // total number of consumers
   private final int MAX_CONSUMPTION_TIME = 1000;

   public ItemConsumer(CloseableQueue<Item> queue)
   {  this.queue = queue;
      generator = new Random();
      consumerName = ""+(char)('A'+numConsumers);
      numConsumers++;
   }

   // consume an item that takes a random length of time
   private void consumeItem(Item item)
   {  // determine how long this item takes to consume
      int duration = generator.nextInt(MAX_CONSUMPTION_TIME);
      try
      {  Thread.sleep(duration);
      }
      catch (InterruptedException e)
      {}
   }
```

cont-

-cont

```
   public void run()
   {  System.out.println("Consumer " + consumerName + " starting");
      // continue while either the queue is still accepting items
      // or there are still items in the queue to consume
      while (queue.isAccepting() || queue.size()>0)
      {  if (queue.size()>0)
         {  // attempt to get the item at front of queue
            try
            {  Item item = queue.dequeue();
               System.out.println("Item " + item
                  + " consumed by consumer " + consumerName);
               consumeItem(item);
            }
            catch (NoSuchElementException e)
            {  // just in case the item is no longer in the queue
               System.out.println
                  ("Element dequeued by another consumer");
            }
         }
         else
         {  // sleep for 100ms so another item has chance to appear
            try
            {  Thread.sleep(100); // sleep 100ms
            }
            catch (InterruptedException e)
            {}
         }
      }
      System.out.println("Consumer " + consumerName + " stopping");
   }
}
```

```
/**
   A class which represents an item which can be produced,
   placed in an item queue, and consumed
   @see Producer.java, ItemQueue.java, Consumer.java
*/

public class Item
{
   private String name;

   public Item(String name)
   {  this.name = name;
   }

   public String toString()
   {  return name;
   }
}
```

Sample Output From `ProducerConsumer`:

```
Producer starting
Consumer A starting
Consumer B starting
Consumer C starting
Item ID number: 0 produced
Item ID number: 0 consumed by consumer A
Item ID number: 1 produced
Item ID number: 1 consumed by consumer B
Item ID number: 2 produced
Item ID number: 2 consumed by consumer C
Item ID number: 3 produced
Item ID number: 3 consumed by consumer B
Item ID number: 4 produced
Item ID number: 5 produced
Item ID number: 4 consumed by consumer A
Item ID number: 6 produced
Item ID number: 7 produced
Item ID number: 8 produced
Item ID number: 5 consumed by consumer A
Item ID number: 6 consumed by consumer C
Item ID number: 9 produced
Producer stopping
Item ID number: 7 consumed by consumer B
Item ID number: 8 consumed by consumer C
Item ID number: 9 consumed by consumer A
Consumer B stopping
Consumer C stopping
Consumer A stopping
```

periodically) and have the producer thread notify them whenever it places items in the queue (putting a consumer back in the active state). See the interface `BlockingQueue` in the `java.util.concurrent` package for more details.

If there is an event in the event dispatch queue, such as a considerable numerical computation or some input/output operation that takes a long time to process, to a user the program would seem sluggish or unresponsive as further events are piled up in the queue behind the demanding task. To avoid this problem, any demanding task is usually placed in a separate thread which is started by the event dispatch thread, allowing it to proceed immediately with the next event in its queue. Thus a user could initiate a numerical computation or a download across a network connection, starting a separate thread and freeing the event dispatch thread to process further user events, and so enabling the user to continue their work.

Because many events result in changes to a GUI, the event dispatch thread often modifies GUI components. However, a complication arises here due to the fact that, for efficiency, most Swing class are not synchronized, so any modifications to components should only be made in the event dispatch thread to avoid race conditions. For example, instead of some thread changing the text in a `JLabel` by:

```
label.setText(percentage+"% complete");
```

it should be sent to the event dispatch queue for changing when it has the opportunity. This can be conveniently achieved by using the static method `invokeLater` of the class `EventQueue` together with an anonymous inner class, such as by:

```
EventQueue.invokeLater(new Runnable()
   {  public void run()
```

```
        {  label.setText(percentage+"% complete");
        }
     });
```

Exercise 3.2 (Implementing Queues with Circular Arrays) *Design and prepare a class called* `ArrayQueue` *for implementing a queue using a circular array (the* `expandCapacity` *method will need some careful consideration). Prepare a simple driver* `main` *method to test your class.*

3.3 Lists

Reading: pp250-284

A *list* (also called an *indexed list*) is a linear collection of elements which has *positional indexing*, where the elements can be referenced using a numeric index, beginning at `0` for the first element in the list. If an element is added to or removed from the list the indices of other elements are adjusted to maintain the contiguity of the indices.

The Collections API includes an interface called `List` which extends the `Collection` interface and describes the abstract data type for a list. Note that a `List` is not a `Set`, since lists allow duplicate elements. The usual `add`, `addAll`,

```
≪interface≫
List<E>

+add(o : E) : boolean
+add(index : int, element : E) : void
+addAll(c : Collection<? extends E>) : boolean
+addAll(index : int, c : Collection<? extends E>) : boolean
+clear() : void
+contains(o : Object) : boolean
+containsAll(c : Collection<?>) : boolean
+equals(o : Object) : boolean
+get(index : int) : E
+hashCode() : int
+indexOf(o : Object) : int
+isEmpty() : boolean
+iterator() : Iterator<E>
+lastIndexOf(o : Object) : int
+listIterator() : ListIterator<E>
+listIterator(index : int) : ListIterator<E>
+remove(index : int) : E
+remove(o : Object) : boolean
+removeAll(c : Collection<?>) : boolean
+retainAll(c : Collection<?>) : boolean
+set(index : int, element : E) : E
+size() : int
+subList(fromIndex : int, toIndex : int) : List<E>
+toArray() : Object[]
+toArray(a : T[]) : T[]
```

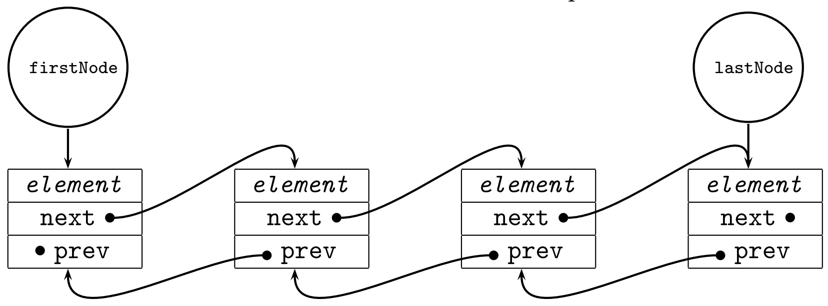
```
public interface ListIterator<E> extends Iterator
{
   public void add(o E);
   public boolean hasNext();
   public boolean hasPrevious();
   public E next() throws NoSuchElementException;
   public int nextIndex();
   public E previous() throws NoSuchElementException;
   public int previousIndex();
   public void remove() throws UnsupportedOperationException,
      IllegalStateException;
   public void set(o E) throws UnsupportedOperationException,
      ClassCastException, IllegalArgumentException,
      IllegalStateException;
}
```

and `remove` methods also have variants which take an index within the list. Also there are `get`, `remove`, and `set` methods for obtaining, removing, and modifying the element at a specified index. Depending on the implementation of the list, accessing elements using indices might typically be $O(1)$ or the less-efficient $O(n)$. Hence, if the implementation of a list is not known it is preferable to use an iterator to iterate through the elements than to access them all using their indices. Besides providing an `Iterator` object, the `List` interface also requires implementations to provide a `ListIterator` object. A `ListIterator` is an iterator which allows bidirectional traversal through the elements in the collection with its methods `next` and `previous`. Like an `Iterator`, a `ListIterator` is considered to be positioned between elements (not at elements) in the collection, and so can have any index between `0` (immediately before the first element) and `list.size()` (immediately after the last element) inclusive. The following code fragment demonstrates how a list can be traversed in reverse order:

```
List<String> list = ...;
// start iterator just beyond last element
ListIterator<String> iterator=list.listIterator(list.size());
while (iterator.hasPrevious())
{  String line = iterator.previous();
   System.out.println(line);
}
```

The Collections API includes two classes, `ArrayList` and `LinkedList`, which provide general-purpose implementations of the `List` interface. The `ArrayList` class is much like the `ArraySet` class from Section 2.1, using an array as its underlying data structure, which gets replaced by another larger array when its capacity is insufficient. If the `add` method specifies an index before the end of the list then all the elements with indices above that index will need to be shifted up one index, using an $O(n)$ loop. Likewise, the `remove` method requires that all elements above the specified index be shifted down one index. However, the `ArrayList` implementation has an efficient `get` method, which can obtain any element in the list, given its index, in constant time $O(1)$.

The alternative common implementation of lists, `LinkedList`, uses a *doubly linked list*. A doubly linked list is similar to a singly linked list, except that each node also has a link to the previous node in the list. This makes traversing the elements with a `ListIterator` more efficient, since each call to `previous` is $O(1)$ thanks to the link to the previous node, rather than requiring an $O(n)$ loop to find the previous element. If the `add` method requires adding the element at either the start or end of the list then this can be performed efficiently in $O(1)$. If instead the element is to be inserted elsewhere, then although elements are not shifted there is no direct access to the particular insertion point, so a loop is required to find the insertion point. Hence, the `add` method is $O(n)$, except in the case of adding at the start or end of the list. For this reason the `LinkedList` class includes the $O(1)$ methods `addFirst`, `addLast`, `removeFirst`, `removeLast`. Similarly, the `get` method must traverse the elements, starting at one end of the list until the element with the specified index is obtained, requiring an $O(n)$ loop, unless dealing with the start or end indices, which can be obtained by `getFirst` and `getLast`. In order to decrease the amount of traversing in the `add`, `remove`, and `get` methods, the `LinkedList` class always starts a traverse from the end that is closer to the specified index.



Since the `get` method of `ArrayList` is always $O(1)$, this implementation of a list is typically chosen when the elements of the list need to be accessed frequently, particularly for random access of elements, but elements are not inserted nor removed frequently (with the exception being insertions and removals from the end of the list, which can be performed $O(1)$). The alternative `LinkedList` has an $O(n)$ `get` method, and so is less efficient for obtaining elements from the list, (except the first and last elements). The `LinkedList` implementation is considered more efficient when elements are frequently added or removed near either end of the list, and either the elements are typically obtained from the ends, or else the elements are obtained sequentially using an iterator (since both implementations use $O(n)$ iterators).

The *Josephus problem* illustrates how indices are convenient for referencing the elements in a list. This problem starts with n soldiers arranged in a circle and a designated first person, then every m-th solder is removed (where $m \leq n$) around the circle. After these solders are removed, the counting continues around the circle that remains. This process continues until all n soldiers have been removed. The order in which soldiers are removed from the circle defines the (n, m)-Josephus permutation of the integers $1, 2, \ldots, n$. For example, the $(7, 3)$-Josephus permutation is $(3, 6, 2, 7, 5, 1, 4)$, since:

$$\begin{array}{l}
1 \quad 2 \quad \boxed{3} \quad 4 \quad 5 \quad 6 \quad 7 \\
1 \quad 2 \quad 4 \quad 5 \quad \boxed{6} \quad 7 \\
1 \quad \boxed{2} \quad 4 \quad 5 \quad 7 \\
1 \quad 4 \quad 5 \quad \boxed{7}
\end{array}$$

$1 \quad 4 \quad \boxed{5}$

$\boxed{1} \quad 4$

$\boxed{4}$

The textbook example `Josephus` uses an `ArrayList` to implement the list collection. Note that an `ArrayList` is more suitable for this problem than a `LinkedList` since the algorithm frequently accesses elements non-sequentially in the list using their indices.

If a linear collection is required which maintains its elements in a particular order, but it is not important to have an index for each element in the collection, then instead of a list a sorted set might be used. A *sorted set* (also called an *ordered list*) is a set where the iterator always gives the elements in ascending order. The order is typically determined by each element's `compareTo` method or else provided by a `Comparator` object, so all the elements in a sorted set must either implement the `Comparable` interface or are accepted by a specified `Comparator`. The Collections API provides the interface `SortedSet`, which is implemented by its `TreeSet` class. This implementation has the advantage that its `add`, `remove`, `get` operations are all $O(\log_2 n)$.

Exercise 3.3 (Queue Implementation of Josephus Problem) *Prepare a method that solves the Josephus problem using a queue and compare the performance of its algorithm to the* `ArrayList` *implementation.*

3.4 Searching

Reading: pp336-341

Searching is the process of finding a particular element (sometimes called the *target element*) if it exists, within some collection of elements (sometimes called the *search pool*). Since searching is a very common computing task, it often needs to be performed as efficiently as possible, minimizing the number of comparisons required to find the element.

Searching involves repeatedly checking whether an element in the collection is equal in some sense to the target element. Elements that are both of a primitive type can be tested for equality (using `==`), whereas an object is tested for equality with another by using its `equals` method. The `equals` method, inherited from the `Object` class, only returns `true` if the objects are actually the same object (so the variables used to reference them are simply aliases of one object). Often the `equals` method is overridden in a class so that distinct objects that might have the same value for some property can be treated as equal. This is done in the `String` class, where two `String` objects are considered equal if they both contain exactly the same characters in the same order. If the `equals` method is overridden it should be carefully written, since there are many algorithms besides searching algorithms that make assumptions about how this method behaves. It should give an equivalence relation on objects (a reflexive, symmetric, and transitive relation), be consistent in the sense that multiple calls to `equals` using the same objects consistently give the same result, and should return `false` if an object is tested with the `null` reference.

Primitive types such as `double`, and `int` can be ordered by their numeric values. Likewise, `char` values can be ordered by comparing their Unicode val-

```
/**
   Class which demonstrates how a list can be used to solve
   the Josephus problem. Starting with numPeople soldiers
   in a circle, every gap-th soldier is removed until
   there are no soldiers remaining
   @author Lewis/Chase (adapted)
*/
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner; // Java 1.5 equivalent of cs1.Keyboard

public class Josephus
{
   public static void main(String[] args)
   {  int numPeople, gap, counter;
      List<Integer> list = new ArrayList<Integer>();
      Scanner in = new Scanner(System.in);
      // get the initial number of soldiers
      System.out.print("Enter the number of soldiers: ");
      numPeople = in.nextInt();
      in.nextLine();
      // get the gap between soldiers
      System.out.print("Enter the gap between soldiers: ");
      gap = in.nextInt();
      // load the initial list of soldiers
      // note that index i holds soldier i+1
      for (int count=1; count<=numPeople; count++)
         list.add(new Integer(count));
      counter = gap-1; // first soldier to remove in list
      // treating the list as circular, remove every nth element
      // until the list is empty
      System.out.print("The order is: ");
      while (!list.isEmpty())
      {  // remove soldier at index counter
         System.out.print(list.remove(counter));
         numPeople--;
         if (numPeople>0)
         {  counter = (counter-1+gap)%numPeople;//note -1 since removal
            System.out.print(", ");
         }
      }
      System.out.println();
   }
}
```

ues. For objects to be ordered they should implement the `Comparable` interface and provide a `compareTo` method. This method returns an `int` value

```
public interface Comparable<T>
{
   public int compareTo(T o);
}
```

that is negative, zero, or positive depending on whether the object is (respectively) less than, equal to, or greater than the object parameter. For example, `"A".compareTo("B")` returns a negative value. In implementing this interface, the `compareTo` method should be written so that it gives a total ordering on objects, and return zero if and only if the objects are equal in the sense of the `equals` method.

For a linear collection there are two common alternative approaches to searching, linear search and binary search, depending whether or not the elements in the collection are ordered in some way.

A *linear search* starts at one end of the search pool and checks the elements sequentially one after another until either the target is found or else there are no further elements to check. For any collection the `contains` method determines whether an element is in the collection, and typically uses an iterator to perform a linear search:

```
Collection<String> collection = ...;
String target = ...;
Iterator<String> iterator = collection.iterator();
boolean found = false;
while (!found && iterator.hasNext())
{  if (target.equals(iterator.next()));
      found = true;
}
```

If it is an array being searched then it is often useful to have the linear search return the index of the (first occurrence of the) target in the array, or a value such as `-1` if not found:

```
String[] lines = ...;
String target = ...;
boolean found = false;
int index = 0;
while (!found && index<lines.length)
{  if (target.equals(lines[index]))
      found = true;
   else
      index++;
}
if (!found)
   index = -1;
```

For a collection of n elements, a linear search has linear time complexity. Unless something more is known about the collection, such as that its elements are sorted in order, one cannot achieve better than $O(n)$.

If the elements in a linear collection are sorted, then a more efficient searching algorithm is possible. Instead of starting the search at one end of the collection a *binary search* starts in the middle of a linear collection and compares the target with the element in the middle. Since the collection is sorted only half of the collection needs to be searched further, so this process is repeated with that half of the collection, repeating until either the element is found or else there is no more of the collection to check. Since the number of elements to search is halved after each comparison, a sorted collection of $n = 2^k$ elements would require at most $k + 1 = \log_2 n + 1$ comparisons, and so the binary search algorithm is $O(\log_2 n)$.

For example, suppose a linear collection consists of the strings:

ant, bat, cat, cow, dog, eel, fly, fox, owl, pig, rat

sorted in ascending (alphabetical) order. A binary search for a string such as "dog" would start by comparing it with the element "eel" in the middle of the collection. Since in alphabetical ordering "dog" < "eel", the second half of the list can be ignored. Then taking the element "cat" in the middle of the remaining collection, since "dog" > "cat", the first half of the remaining elements can be ignored, leaving just two elements. Considering "cow" to be the middle element, since "dog" > "cow", only one element remains to check.

ant bat cat cow dog $\boxed{\text{eel}}$ fly fox owl pig rat

ant bat $\boxed{\text{cat}}$ cow dog

$\boxed{\text{cow}}$ dog

$\boxed{\text{dog}}$

The class `BinarySearch` demonstrates how a binary search can be written to search for an element in a sorted array. The class is declared as `BinarySearch<E extends Comparable>` to ensure the generic type `E` extends `Comparable` (so elements of type `E` do implement the `Comparable` interface). Note that the binary search algorithm is most conveniently coded as a recursive algorithm — the `search` method calls itself with a more narrow range of indices to search — hence there is a base case when `start>=end`, indicating that there are no elements to search.

Comparing the two common searching methods for linear collections, one notes that the binary search algorithm is $O(\log_2 n)$, and so much more efficient than the $O(n)$ linear search (particularly for large values of n). One caution however, since the binary search algorithm uses random access to the elements of a collection, it does perform at $O(\log_2 n)$ for a list such as `ArrayList` (which has $O(1)$ access to elements). However, a list such as `LinkedList` has $O(n)$ access to elements, and so, depending on how the search is implemented, a binary search is actually $O(n)$ (or if not implemented carefully is worse yet, $O(n \log_2 n)$). The `Collections` class has methods for performing binary searches on `List` collections, and the `Arrays` class has miscellaneous methods such as binary searches for arrays.

The binary search algorithm requires the elements in the collection to be comparable, and the collection to be sorted, which if needed typically requires an $O(n \log_2 n)$ sorting process. If the target appears several times in a linear collection there is no guarantee which of the occurrences will be found by a

```
/**
   A class which demonstrates binary search of an array
   @author Andrew Ensor
*/

public class BinarySearch<E extends Comparable>
{
   private E[] elements;

   public BinarySearch(E[] elements)
   {  this.elements = elements;
   }

   public int search(E target)
   {  if (target == null)
         throw new NullPointerException("search target is null");
      return search(target, 0, elements.length);
   }

   // recursive method which searches through the elements array
   // between start index (inclusive) and end index (exclusive) for the
   // index of specified target, or returns -(insertion)-1 if not found
   private int search(E target, int start, int end)
   {  if (start >= end)
         return -start-1; // negative value
      else
      {  int midpoint = (start+end)/2; // midpoint of search region
         int comparison = target.compareTo(elements[midpoint]);
         if (comparison == 0)
            return midpoint;
         else if (comparison < 0)
            return search(target, start, midpoint);
         else // comparison > 0
            return search(target, midpoint+1, end);
      }
   }

   public static void main(String[] args)
   {  String[] list = {"ant", "bat", "cat", "cow", "dog", "eel",
         "fly", "fox", "owl", "pig", "rat"};
      BinarySearch<String> bin = new BinarySearch<String>(list);
      String target = "dog";
      int index = bin.search(target);
      if (index >= 0)
         System.out.println(target + " found at index " + index);
      else
         System.out.println(target + " not at index " + (-index-1));
   }
}
```

binary search, whereas a linear search always returns the first occurrence. Typically with a search of a sorted linear collection the algorithm is written to return a negative value for the index if the target is not found in the collection. As a by-product of the search algorithm, the place where the target should have occurred is found (where it would be inserted if added), so often this information is returned in the negative index as $-i-1$, where i would have been the insertion point.

Exercise 3.4 (Binary Search with a List Iterator) *Design and implement a recursive method that uses a* `ListIterator` *to perform a binary search of a* `List` *collection (which is not presumed to have efficient random access to elements). The method should have $O(n)$ link traversals and $O(\log_2 n)$ element comparisons. The method will need to calculate how many traversals to move the iterator back or forth at each step of the recursion.*

3.5 Sorting

Reading: pp341-356

Sorting is the process of arranging a linear collection of elements in order, based on each object's implementation of the `Comparable` interface, or the numeric ordering for primitive types such as `double` and `int`. There are many common sorting algorithms which vary in their complexity, efficiency, and suitability for various tasks.

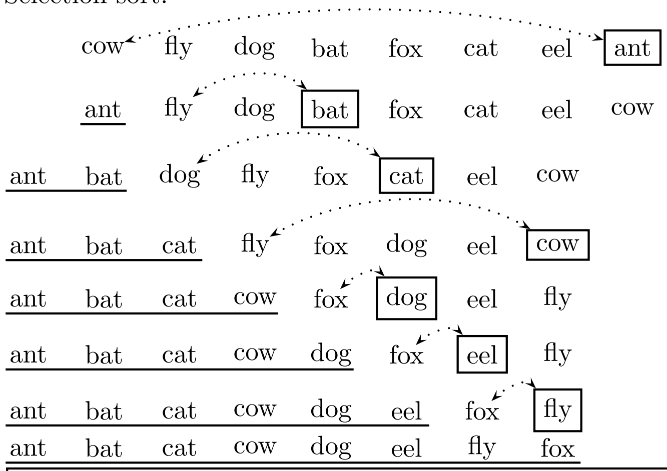
One of the simplest and most intuitive algorithms for sorting is the *selection sort* algorithm. This algorithm works by first finding the smallest value in a list and swapping the first value with it, then finding the second smallest value and swapping the second value with it, continuing until all the values are in order. For example, to sort the linear collection of strings:

"cow", "fly", "dog", "bat", "fox", "cat", "eel", "ant",

the selection sort algorithm first finds the smallest string "ant" (using the `compareTo` method of the `String` class), and swaps it with the string "cow" at the start of the list. Then ignoring the first position in the list, the next smallest value is "bat" which gets swapped with the string "fly" in the second position in the list. This continues until the entire list is sorted. The example `ArraySorter` contains a method `selectionSort` which uses this algorithm to sort an array called `list` of type `E[]`, where `E` is a generic subtype of `Comparable` (hence this method can use the `compareTo` method of the array elements). Note that an outer loop iterates between `0` (the first index in the array) and one less than the length of the array (since the one remaining element must be the largest), and an inner loop is used to find the smallest value. For an array of length n this method requires $(n-1)+(n-2)+\ldots+1=\frac{1}{2}n(n-1)$ comparisons and $n-1$ swaps, and so is considered an $O\left(n^2\right)$ algorithm.

Another simple sorting algorithm is the *insertion sort* algorithm. This algorithm works by progressively enlarging a sorted subset of the list and inserting each value one at a time into the correct position in the subset, moving larger values over to accommodate it. For example, once again consider the list "cow", "fly", "dog", "bat", "fox", "cat", "eel", "ant". The insertion sort algorithm starts with the sorted subset containing only the string "cow", and inserts the

Selection sort:


cow fly dog bat fox cat eel ant
ant fly dog bat fox cat eel cow
ant bat dog fly fox cat eel cow
ant bat cat fly fox dog eel cow
ant bat cat cow fox dog eel fly
ant bat cat cow dog fox eel fly
ant bat cat cow dog eel fox fly
ant bat cat cow dog eel fly fox

```
public class ArraySorter<E extends Comparable>
{
   ...
   public void selectionSort(E[] list)
   {  int indexMin; // index of least element
      E temp; // temporary reference to an element for swapping
      for (int i=0; i<list.length-1; i++)
      {  // find the least element that has index>=i
         indexMin = i;
         for (int j=i+1; j<list.length; j++)
         {  if (list[j].compareTo(list[indexMin])<0)
               indexMin = j;
         }
         // swap the element at indexMin with the element at i
         temp = list[indexMin];
         list[indexMin] = list[i];
         list[i] = temp;
      }
   }
   ...
}
```

string "fly" into the subset at the end. Then the string "dog" is inserted into this subset at the second position and "fly" is shifted along to accommodate it. Next the string "bat" is considered and inserted in the correct position. This algorithm, demonstrated in the `insertionSort` method, also uses nested loops, the outer loop starting at index `1` of the array (since the subset with just the first element is already sorted), and finishing at the last index of the array. It requires somewhere between $n-1$ (for a list already in ascending order) and $\frac{1}{2}n(n-1)$ (for a list in descending order) comparisons and anywhere up to $\frac{1}{2}n(n-1)$ shifts. Although also an $O\left(n^2\right)$ algorithm it is quite efficient for collections that are already almost sorted.

A third $O\left(n^2\right)$ sorting algorithm is *bubble sort.* This algorithm makes repeated passes through the collection, comparing adjacent elements and swapping them if they are out of order. In this way after one iteration the largest element has bubbled (swapped) its way to the top (end of the collection) and other large values have bubbled part of the way. The `bubbleSort` method uses $\frac{1}{2}n(n-1)$ comparisons and up to $\frac{1}{2}n(n-1)$ swaps (it requires a significant number of swaps). One small efficiency that can be made with this algorithm is to have it terminate immediately if some iteration of the outer loop results in no swapping of elements in the inner loop. This can improve the efficiency when the original list is almost sorted.

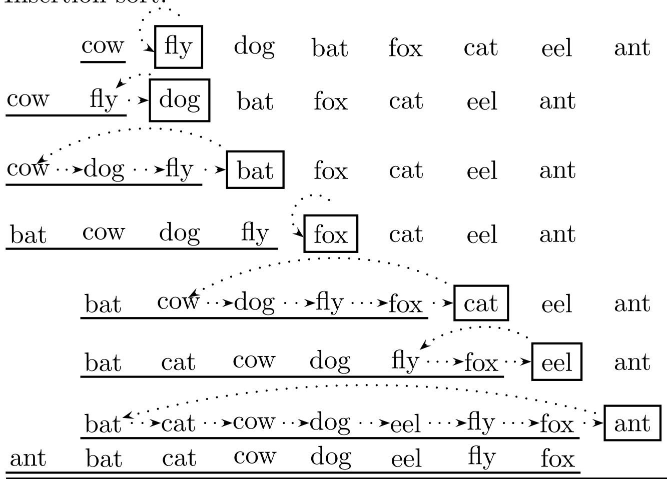
One of the preferred sorting algorithms is *quick sort* (variants of which are often implemented in API libraries). The quick sort algorithm sorts a linear collection by partitioning it using an arbitrarily chosen *partition element.* The collection is partitioned by placing all the elements less than the partition element on the left side of the collection, all the elements greater than it on the right, with it in between. The `partition` method demonstrates how this is done with $k-1$ comparisons and up to $\frac{1}{2}k+1$ swaps for an array of length k (a `LinkedList` would require a slightly modified algorithm to achieve this efficiency). Then the quick sort algorithm is recursively applied separately to the left side and to the right side, thus sorting the entire collection. The `quickSortSegment` method chooses the partition element to be the first element in the segment of the list that is to be sorted, which works best if the list is not already partially sorted (so both sides have roughly half the elements of the collection). In such a case the algorithm requires:

$$(n-1)+2\left(\tfrac{1}{2}n-1\right)+4\left(\tfrac{1}{4}n-1\right)+\ldots+\tfrac{n}{2}\left(2-1\right)\approx n\log_2 n-(n-1)$$

comparisons, and up to about $\frac{n}{2}\log_2 n+(n-1)$ swaps. Hence the quick sort algorithm is typically $O\left(n\log_2 n\right)$, although in the worst case (a collection already in descending order and always choosing the first element as the partition element) there can be $O\left(n^2\right)$ comparisons.

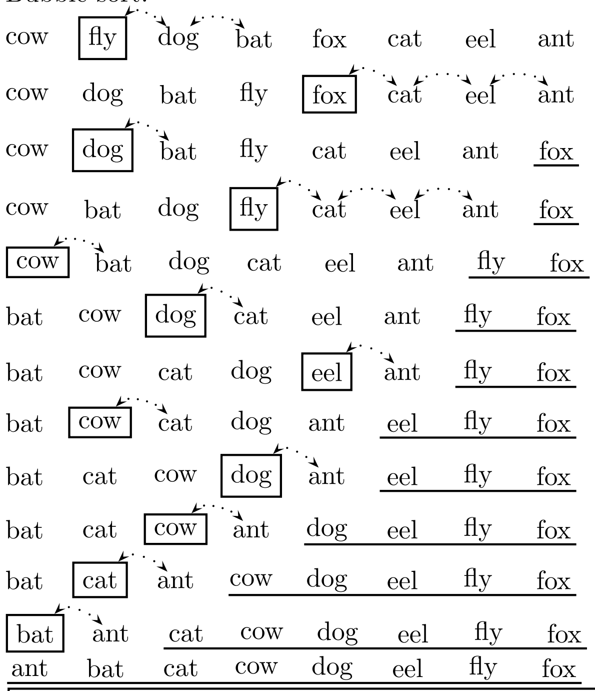
Another efficient $O\left(n\log_2 n\right)$ sorting algorithm that is inherently recursive is *merge sort.* This algorithm is based on the fact that two lists that are each already sorted can be merged together into one sorted list in $O(n)$ by a loop which repeatedly picks the lesser of the smallest element remaining in each of the two lists. The merge sort algorithm uses this fact by recursively dividing a linear collection in half until each part has only one element and then merging the parts together a pair at a time. In total it requires between $\frac{1}{2}n\log_2 n$ and $n\log_2 n$ comparisons. The example `mergeSort` method, to keep the code simple, allocates $n-1$ arrays to facilitate the merging process.

Insertion sort:


cow fly dog bat fox cat eel ant
cow fly dog bat fox cat eel ant
cow dog fly bat fox cat eel ant
bat cow dog fly fox cat eel ant
bat cow dog fly fox cat eel ant
bat cat cow dog fly fox eel ant
bat cat cow dog eel fly fox ant
ant bat cat cow dog eel fly fox

```
public class ArraySorter<E extends Comparable>
{
   ...
   public void insertionSort(E[] list)
   {  E elementInsert;
      for (int i=1; i<list.length; i++)
      {  // get the element at index i to insert at some index<=i
         elementInsert = list[i];
         // find index where to insert element to maintain 0..i sorted
         int indexInsert = i;
         while (indexInsert>0 &&
            list[indexInsert-1].compareTo(elementInsert)>0)
         {  // shift element at insertIndex-1 along one to make space
            list[indexInsert] = list[indexInsert-1];
            indexInsert--;
         }
         // insert the element
         list[indexInsert] = elementInsert;
      }
   }
   ...
}
```

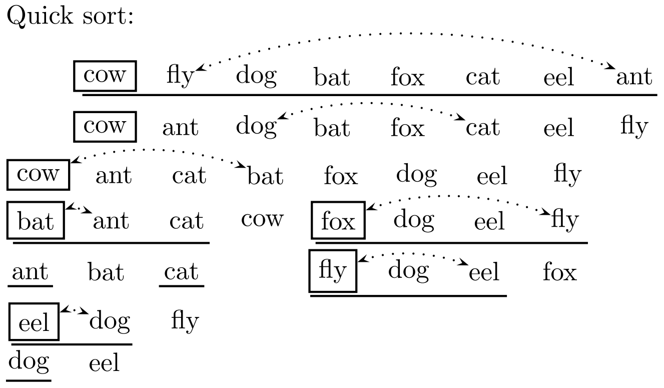
Bubble sort:



cow fly dog bat fox cat eel ant
cow dog bat fly fox cat eel ant
cow dog bat fly cat eel ant fox
cow bat dog fly cat eel ant fox
cow bat dog cat eel ant fly fox
bat cow dog cat eel ant fly fox
bat cow cat dog eel ant fly fox
bat cow cat dog ant eel fly fox
bat cat cow dog ant eel fly fox
bat cat cow ant dog eel fly fox
bat cat ant cow dog eel fly fox
bat ant cat cow dog eel fly fox
ant bat cat cow dog eel fly fox

```
public class ArraySorter<E extends Comparable>
{
   ...
   public void bubbleSort(E[] list)
   {  E temp; // temporary reference to an element for swapping
      for (int i=list.length-1; i>=0; i--)
      {  // pass through indices 0..i and bubble (swap) adjacent
         // elements if out of order
         for (int j=0; j<i; j++)
         {  if (list[j].compareTo(list[j+1])>0)
            {  // swap the elements at indices j and j+1
               temp = list[j+1];
               list[j+1] = list[j];
               list[j] = temp;
            }
         }
      }
   }
   ...
}
```

Quick sort:


cow fly dog bat fox cat eel ant
cow ant dog bat fox cat eel fly
cow ant cat bat fox dog eel fly
bat ant cat cow fox dog eel fly
ant bat cat fly dog eel fox
eel dog fly
dog eel

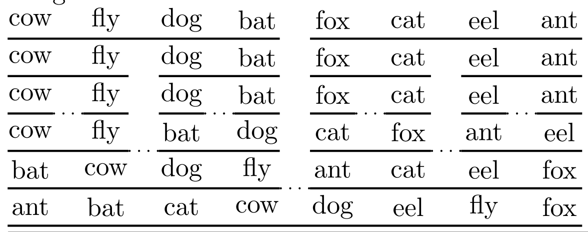
```
public class ArraySorter<E extends Comparable>
{
   ...
   public void quickSort(E[] list)
   {  quickSortSegment(list, 0, list.length);
   }

   // recursive method which applies quick sort to the portion
   // of the array between start (inclusive) and end (exclusive)
   private void quickSortSegment(E[] list, int start, int end)
   {  if (end-start>1) // then more than one element to sort
      {  // partition the segment into two segments
         int indexPartition = partition(list, start, end);
         // sort the segment to the left of the partition element
         quickSortSegment(list, start, indexPartition);
         // sort the segment to the right of the partition element
         quickSortSegment(list, indexPartition+1, end);
      }
   }
```

cont-

-cont

```
   // use the index start to partition the segment of the list
   // with the element at start as the partition element
   // separating the list segment into two parts, one less than
   // the partition, the other greater than the partition
   // returns the index where the partition element ends up
   private int partition(E[] list, int start, int end)
   {  E temp; // temporary reference to an element for swapping
      E partitionElement = list[start];
      int leftIndex = start; // start at the left end
      int rightIndex = end-1; // start at the right end
      // swap elements so elements at left part are less than
      // partition element and at right part are greater
      while (leftIndex<rightIndex)
      {  // find element starting from left greater than partition
         while (list[leftIndex].compareTo(partitionElement) <= 0
            && leftIndex<rightIndex)
            leftIndex++; // this index is on correct side of partition
         // find element starting from right less than partition
         while (list[rightIndex].compareTo(partitionElement) > 0)
            rightIndex--; // this index is on correct side of partition
         if (leftIndex<rightIndex)
         {  // swap these two elements
            temp = list[leftIndex];
            list[leftIndex] = list[rightIndex];
            list[rightIndex] = temp;
         }
      }
      // put the partition element between the two parts at rightIndex
      list[start] = list[rightIndex];
      list[rightIndex] = partitionElement;
      return rightIndex;
   }
   ...
}
```

Merge sort:



cow	fly	dog	bat	fox	cat	eel	ant
cow	fly	dog	bat	fox	cat	eel	ant
cow	fly	dog	bat	fox	cat	eel	ant
cow	fly	bat	dog	cat	fox	ant	eel
bat	cow	dog	fly	ant	cat	eel	fox
ant	bat	cat	cow	dog	eel	fly	fox

```
public class ArraySorter<E extends Comparable>
{
   ...
   public void mergeSort(E[] list)
   {  mergeSortSegment(list, 0, list.length);
   }

   // recursive method which applies merge sort to the portion
   // of the array between start (inclusive) and end (exclusive)
   private void mergeSortSegment(E[] list, int start, int end)
   {  int numElements = end-start;
      if (numElements>1)
      {  int middle = (start+end)/2;
         // sort the part to the left of middle
         mergeSortSegment(list, start, middle);
         // sort the part to the right of middle
         mergeSortSegment(list, middle, end);
         // copy the two parts elements into a temporary array
         E[] tempList = (E[])(new Comparable[numElements]); //unchecked
         for (int i=0; i<numElements; i++)
            tempList[i] = list[start+i];
         // merge the two sorted parts from tempList back into list
         int indexLeft = 0; // current index of left part
         int indexRight = middle-start; // current index of right part
```

cont-

-cont

```
         for (int i=0; i<numElements; i++)
         {  // determine which element to next put in list
            if (indexLeft<(middle-start))//left part still has elements
            {  if (indexRight<(end-start))// right part also has elem
               {  if (tempList[indexLeft].compareTo
                        (tempList[indexRight])<0) // left element smaller
                        list[start+i] = tempList[indexLeft++];
                  else // right element smaller
                     list[start+i] = tempList[indexRight++];
               }
               else // take element from left part
                  list[start+i] = tempList[indexLeft++];
            }
            else // take element from right part
               list[start+i] = tempList[indexRight++];
         }
      }
   }
   ...
}
```

Exercise 3.5 (Measuring Sorting Efficiency) *Modify the algorithms in the* `ArraySorter` *class so that the total number of comparisons and execution time for each algorithm is calculated. Then compare the algorithms for some linear collections, such as a random collection of strings, a collection that is almost ordered, and a collection that is almost in decreasing order.*

3.6 Specialized Sorting Algorithms

Reading: pp222-227

The $O(n \log_2 n)$ sorting algorithms such as quick sort and merge sort are generally considered the most efficient general purpose sorting algorithms. However, in some specialized circumstances, if more is known about the particular collection to be sorted and its elements then an even more efficient algorithm might be possible.

For instance, if all the values to be sorted have a particular structure which can be used part at a time to sort the values, then a radix sort might be appropriate. A *radix sort* sorts values based upon individual digits or letters that comprise the values, rather than comparing the entire values with each other. For example, to sort three-digit numbers the sort can be broken down into three separate sorts, one for each digit, of which there are 10 possibilities. Instead, to sort two-letter uppercase postal codes the sort can be broken down into two sorts, one for each of the two letters of the code, of which there are 26 possibilities.

One way to implement a radix sort is to use a queue for each of the possible digits or letters. Intuitively, one might expect to sort the values starting with the most-significant digit first, however, this would require further queues for each of the other digits. Instead, radix sort starts by sorting on the least-significant digit first, combining the values back into a single list by dequeuing from each queue in order, and then repeating with all the values again but using the next digit (reusing the same queues), repeating the process for each digit, with sorting on the most-significant digit last.

For example, radix sort can be used to sort the three-digit values:

073, 521, 135, 291, 550, 153, 913, 910, 770, 321, 039, 125.

Ten queues are used three times over to store the values. First the queues are used to sort the values according to their one's-digit. Then the values are dequeued from each queue in turn and reassembled into a single collection (being careful to maintain the order in which values were obtained). Then the values are taken one at a time from the reassembled collection and sorted according to their ten's-digit. After reassembling the values from each queue in turn back into the collection the process is repeated with the hundred's-digit. Once reassembled back into the collection for the third time the values are fully sorted. This example is demonstrated by the `RadixSort` class.

Note that a radix sort of n elements requires n enqueues and n dequeues for each parse through the elements. If the elements have approximately $\log n$ digits then radix sort is an $O(n \log_2 n)$ sorting algorithm, and so appears to rival quick sort's average time of $O\left(n \log_2 n\right)$. However, it requires that the elements have a special structure, and requires more memory for the required queues (whereas algorithms such as quick sort work *in place* and so do not require extra memory). Furthermore, in practice the constant factors hidden in the $O(n \log_2 n)$ complexity are comparatively quite high for radix sort.

If there are comparatively few possible values for the elements in a collection then there is a very simple and efficient $O(n)$ algorithm for sorting the collection. A *bin sort* allocates a bin for each of the possible values, and in a single parse through the collection assigns each element to a bin. If the collection does not contain any duplicates (such as a set collection) then each bin can just hold a boolean indicating whether there is an element with this value. This is demonstrated in the class `BinSort`, which can be easily modified to allow repetitions of elements.

Related to sorting is the *selection problem*. This problem requires finding the i-th smallest element in a linear collection of n elements. Special cases are $i = 1$ for the minimum, $i = \frac{1}{2}(n + 1)$ for the median, and $i = n$ for the maximum. The minimum and maximum can each be found using a simple $O(n)$ loop, and if the collection is already sorted then the selection problem can also be solved in $O(n)$. However, if starting with an unsorted collection, first sorting the collection would generally require an $O\left(n \log_2 n\right)$ algorithm, but this is not necessary to solve the selection problem for a given value of i.

One efficient solution to the selection problem is based on the fact that to know that the $i + 1$-th smallest element in a linear collection is at index i does not require that the elements of the list all be sorted in order, but instead just that the elements on the left of this index all be smaller than the element at i, and the elements on the right all be larger. In other words, if a partition of the collection can be made with partition element at i, then this element is the $i + 1$-th smallest element in the collection. The `partition` method from the quick sort algorithm of Section 3.5 is used by the class `SelectionProblem` to give an algorithm for solving the selection problem, typically in $O(n)$ time. It achieves this by randomly selecting an element from the list and partitioning the

Radix sort:

queue 9:	queue 7:	queue 5:	queue 3:	queue 2:	queue 1:	queue 0:
039	empty	125,135	913,153,073	empty	321,291,521	770,910,550

queue 9:	queue 7:	queue 5:	queue 3:	queue 2:	queue 1:	queue 0:
291	073,770	153,550	039,135	125,321,521	913,910	empty

queue 9:	queue 7:	queue 5:	queue 3:	queue 2:	queue 1:	queue 0:
913,910	770	550,521	321	291	153,135,125	073,039

```
/**
   Class which demonstrates how radix sort can be used to
   sort fixed digit non-negative integer numbers
*/
public class RadixSort
{
   public static void main(String[] args)
   {  int[] list = {73, 521, 135, 291, 550, 153, 913, 910, 770, 321,
            39, 125}; // three-digit non-negative integers
      final int NUM_DIGITS = 3;
      // create the 10 queues
      LinkedQueue<Integer>[] radixQueues
         = (LinkedQueue<Integer>[])(new LinkedQueue[10]); // unchecked
      for (int queueNum=0; queueNum<10; queueNum++)
         radixQueues[queueNum] = new LinkedQueue<Integer>();
      // use radix sort on each of the digits
      int pwrOfTen = 1; // pwrOfTen = 10^digit
      int queueNum; // determines which queue to use for a value
      for (int digit=0; digit<NUM_DIGITS; digit++)
      {  // parse through all the values in the list and enqueue each
         // into the appropriate queue
         for (int i=0; i<list.length; i++)
         {  // extract digit from value to determine the queue to use
            queueNum = (list[i]/pwrOfTen)%10;
            radixQueues[queueNum].enqueue(new Integer(list[i]));
         }
         // dequeue each queue in turn and put values back into list
         int nextIndex = 0;
         for (queueNum=0; queueNum<10; queueNum++)
         {  while (!radixQueues[queueNum].isEmpty())
               list[nextIndex++]
                  = radixQueues[queueNum].dequeue().intValue();
         }
         pwrOfTen *= 10;
      }
      // output the results
      for (int i=0; i<list.length; i++)
         System.out.print(((i>0)?", ":"") + list[i]);
      System.out.println();
   }
}
```

```
/**
   Class which demonstrates how bin sort can be used to
   sort distinct integer numbers between 0 and MAX_VALUE
*/

public class BinSort
{
   public static void main(String[] args)
   {  int[] list = {17, 2, 23, 7, 41, 29, 19, 43, 31, 5, 11, 47, 13,
         3, 37}; // distinct integer values between 0 and MAX_VALUE
      final int MAX_VALUE = 50;
      boolean[] flags = new boolean[MAX_VALUE+1]; //initially all false
      // determine which bins should be set to true
      for (int i=0; i<list.length; i++)
         flags[list[i]] = true;
      // use the flags to put the numbers back in the list sorted
      int flagNo = 0;
      for (int i=0; i<list.length; i++)
      {  // find the next flag that is true
         while (flagNo<flags.length && !flags[flagNo])
            flagNo++;
         list[i] = flagNo++;
      }
      // output the results
      for (int i=0; i<list.length; i++)
         System.out.print(((i>0)?", ":"") + list[i]);
      System.out.println();
   }
}
```

list using that element (using a random element rather than the first element each time avoids a drop in efficiency if the list is already nearly sorted in some way). If the index of the partition is the same as i then the sought $i+1$-th smallest element must be the partition element, if the index is greater than i then it is known that the sought element is somewhere in the left side of the partition, whereas if it is less than i then the sought element is on the right side. In either case only one side of the partition needs to be partitioned further (whereas quick sort would continue to partition both sides). Repeating this process recursively eventually gives the $i+1$-th smallest element.

Exercise 3.6 (Bin Sort with Repetitions) *Modify the bin sort algorithm so it can be used with integers in the range* `0` *to* `100` *(inclusive) and allows for repetitions of elements.*

```
/**
   Class which demonstrates how the selection problem
   (finding the i+1-th smallest element in an unsorted collection)
   can be solved by repeatedly using randomized partitions
   @author Andrew Ensor
*/
import java.util.Random;

public class SelectionProblem<E extends Comparable>
{
   private Random generator;

   public SelectionProblem()
   {  generator = new Random();
   }

   // returns the (i+1)-th smallest element in the list, i.e. the
   // element which would appear at index i if the list were sorted
   public E find(E[] list, int i)
   {  return find(list, i, 0, list.length);
   }

   // recursive method which returns the (i+1)-th smallest element
   // in the list, which is known to be between indices start
   // (inclusive) and end (exclusive)
   private E find(E[] list, int i, int start, int end)
   {  // base case when only one element in range
      if (start+1>=end)
         return list[start];
      else
      {  // get a random index between start and end and
         // swap element there with the element at start
         // so that the random element is used as partition element
         int randomIndex = generator.nextInt(end-start) + start;
         E temp = list[start];
         list[start] = list[randomIndex];
         list[randomIndex] = temp;
         // randomly partition the list between startIndex and endIndex
         int indexPartition = partition(list, start, end);
         if (indexPartition == i) // partition element is i-th smallest
            return list[indexPartition];
         else if (i<indexPartition) // i-th smallest in left side
            return find(list, i, start, indexPartition);
         else // i>indexPartition so i-th smallest in right side
            return find(list, i, indexPartition+1, end);
      }
   }
```

cont-

-cont

```
   // use the index start to partition the segment of the list
   // with the element at start as the partition element
   // separating the list segment into two parts, one less than
   // the partition, the other greater than the partition
   // returns the index where the partition element ends up
   private int partition(E[] list, int start, int end)
   {  E temp; // temporary reference to an element for swapping
      E partitionElement = list[start];
      int leftIndex = start+1; // start at the left end
      int rightIndex = end-1; // start at the right end
      // swap elements so elements at left part are less than
      // partition element and at right part are greater
      while (leftIndex<rightIndex)
      {  // find element starting from left greater than partition
         while (list[leftIndex].compareTo(partitionElement) <= 0
            && leftIndex<rightIndex)
            leftIndex++; // this index is on correct side of partition
         // find element starting from right less than partition
         while (list[rightIndex].compareTo(partitionElement) > 0)
            rightIndex--; // this index is on correct side of partition
         if (leftIndex<rightIndex)
         {  // swap these two elements
            temp = list[leftIndex];
            list[leftIndex] = list[rightIndex];
            list[rightIndex] = temp;
         }
      }
      // put the partition element between the two parts at rightIndex
      list[start] = list[rightIndex];
      list[rightIndex] = partitionElement;
      return rightIndex;
   }

   public static void main(String[] args)
   {  SelectionProblem<String> selector=new SelectionProblem<String>();
      String[] list = {"cow", "fly", "dog", "bat", "fox", "cat", "eel",
         "ant"};
      int i=4; // 5-th smallest element in list
      String element = selector.find(list, i);
      System.out.println("The " + (i+1) + "-th smallest element is "
         + element);
   }
}
```

Chapter 4

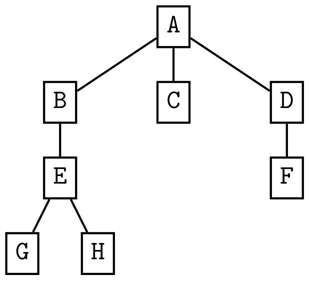
Trees

4.1 Trees

Reading: pp362-380

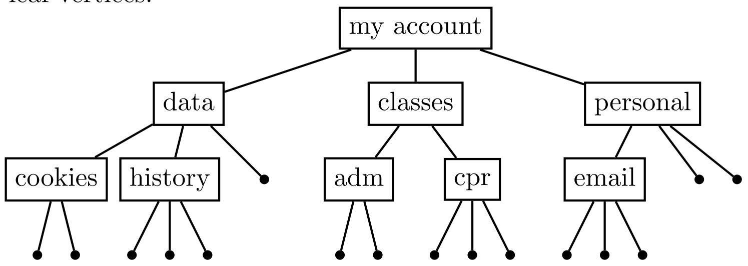
A *tree* is a non-linear collection comprised of *vertices* (or *nodes*) in which the elements are stored, and *edges* which connect pairs of vertices together (technically, a tree is a connected graph which contains no closed paths).

Often one vertex of a tree is singled out and called the *root* of the tree, and the tree is visualized with the root at the top. The *level* of a vertex is the number of edges that must be followed to get from the root to that vertex, with the root at level 0. If a vertex at some level i is joined to a vertex at level $i + 1$ by an edge, then the first vertex is called the *parent* and the other its *child*, with any two children of the same parent referred to as *siblings*. A vertex that does not have any children is called a *leaf*. The *height* of a tree is the maximum level in the tree, and the tree is said to be *balanced* if the child subtrees of any vertex always have height within one of each other. If a tree limits each vertex to have no more than k children then it is called a k*-ary* tree. In particular, a *binary tree* is a 2-ary tree (where each vertex has at most two children).



For example, in the graph alongside node `A` is the root and is the parent of three child nodes. Nodes `B`, `D`, and `E` are also parent nodes, whereas nodes `C`, `F`, `G`, and `H` are leaf nodes. This tree has height 3, but since one leaf is at level 1, yet others are at level 3, it is not balanced (were node `C` to have one or more child nodes it would be a balanced tree). If further nodes could only be added under the proviso that no parent could have more than three children then this tree would be considered a 3-ary tree.

Rooted trees are widely used for the organization of information. For example, computer files are usually stored in a directory structure using folders. Such a hierarchy is actually a tree. The main directory (commonly called the *root directory*) is the root and each folder is a vertex in the tree, with files being the leaf vertices:



The `javax.swing.tree` package contains an interface `TreeNode` which can be used to describe a vertex (node) in a tree (actually this interface was specifically written to support the `JTree` GUI component, but is quite suitable for a general tree collection - rather surprisingly it lacks a method `getUserObject` for obtaining the element held in the node). This interface describes meth-

```
≪interface≫
TreeNode
-----------------------------------------------------
+children() : Enumeration
+getAllowsChildren() : boolean
+getChildAt(childIndex : int) : TreeNode
+getChildCount() : int
+getIndex(node : TreeNode) : int
+getParent() : TreeNode
+isLeaf() : boolean
```

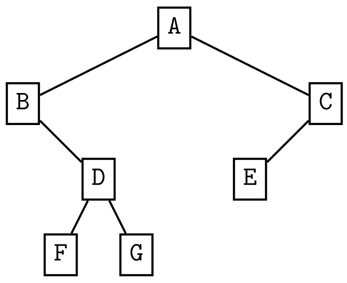
ods that a typical node in a tree would support, allowing access to the parent of the node as well as its children, either one at a time using `getChildAt` or all together using `children` (note that an `Enumeration` is an older version of the preferred `Iterator` interface, with just two methods `hasMoreElements` and `nextElement`).

A tree collection is often *mutable* in the sense that nodes can be added and removed from the tree, so `TreeNode` has a subinterface called `MutableTreeNode`. This subinterface allows child nodes to be added and removed, the node to be

```
≪interface≫
MutableTreeNode
-----------------------------------------------------
+insert(child : MutableTreeNode, index : int) : void
+remove(index : int) : void
+remove(node : MutableTreeNode) : void
+removeFromParent() : void
+setParent(newParent : MutableTreeNode) : void
+setUserObject(object : Object) : void
```

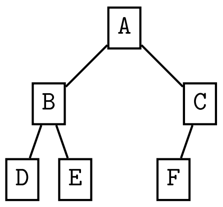
removed from its parent, and the object stored in the node to be changed.

Most array implementations of trees make some restriction on the tree, such as being a k-ary tree. The element for the root is placed at index 0 of the array, and for each element at some index i of the array its children are placed at the indices $ki+1, ki+2, \ldots, ki+k$ as required, with `null` references where there is no child. For example, the illustrated binary tree:

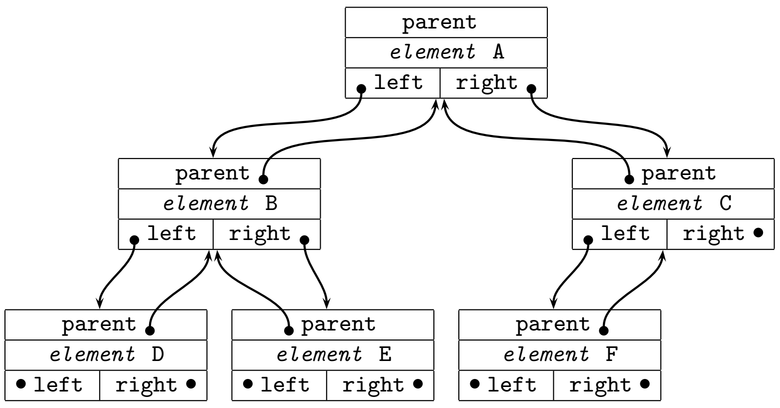


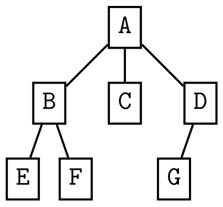
0	A
1	B
2	C
3	null
4	D
5	E
6	null
7	null
8	null
9	F
10	G

can be represented as an array of length `11`, where the children of a node at index i appear at indices $2i+1$ (for the child on the left) and $2i+2$ (for the child on the right). The root holding `A` is placed at index `0`. Its two children, holding `B` and `C` are placed at indices `1` and `2` respectively. Since `B` does not have a left child there is no entry at index `3`, but its right child `D` is placed at index `4`. This array currently has capacity to hold a right child of `C`, and a left child of `B` with two children. But note that if a node as added as a left child of `F` then the array would need expanded capacity to hold it at index `19`. This demonstrates that such an array implementation can be quite wasteful of memory if the tree is not balanced.

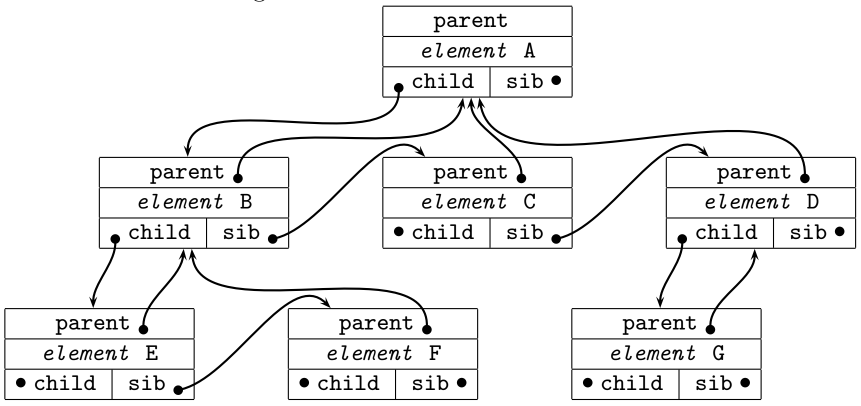


There are various linked data structure implementations of trees. For a binary tree a popular strategy is to have a node hold not only the element, but also references to its parent node and each of its child nodes.





An alternative linked data structure for implementing trees which is more general in that it allows an arbitrary number of children from any parent, is to have each node hold a link to its parent, a link to just one of its children, and a link to its next sibling.



Yet another alternative is to have each node store links to its children in a `List` collection. The example class `ListTreeNode` shows how a tree collection can be implemented using a list of links to the child nodes. The driver `main` method demonstrates how a tree of `MutableTreeNode` nodes is built, then it removes a subtree (consisting of `nodeC` and its only child `nodeF`) and inserts it elsewhere in the tree (as a child of `nodeB`).

The class `DefaultMutableTreeNode` in the `javax.swing.tree` package provides an implementation of a general purpose tree node. It implements the `MutableTreeNode` interface but also provides other features, such as methods for traversing the tree in various orders, or for following the path between two nodes. This class (and the `ListTreeNode` class) are not thread safe, and so require synchronization if used by multiple threads (a convention is to synchronize using the root of the tree as the monitor).

Exercise 4.1 (Linked Implementation of a Binary Tree) *Design and prepare a class called* `LinkedBinaryTreeNode` *for implementing a binary tree comprised of* `MutableTreeNode` *nodes using a linked data structure, where each node is linked to its parent and to its children.*

4.2 Binary Search Trees

Reading: pp396-426

The Collections API has an interface called `SortedSet` which extends the `Set` interface and describes the abstract data type for a sorted set (a set where the iterator always gives the elements in ascending order). It includes methods `headSet`, `subSet`, and `tailSet` for creating sorted sets which are subsets of the original sorted set (referred to as *views* of the collection). Modifications to either the original sorted set or any of its views also affects the others.

```
/**
   A class that implements a tree node using a list
   to store references to the child nodes
   @author Andrew Ensor
*/
import java.util.ArrayList;
import java.util.Enumeration; // old version of an Iterator
import java.util.Iterator;
import java.util.List;
import javax.swing.tree.MutableTreeNode;
import javax.swing.tree.TreeNode;

public class ListTreeNode<E> implements MutableTreeNode
{
   private E element;
   private MutableTreeNode parent;
   private List<MutableTreeNode> children;

   public ListTreeNode()
   {  this(null);
   }

   public ListTreeNode(E element)
   {  this.element = element;
      parent = null;
      children = new ArrayList<MutableTreeNode>();//no initial children
   }

   // returns an Enumeration of the child nodes
   public Enumeration<E> children()
   {  return (Enumeration<E>)
         (new Enumerator(children.iterator())); // unchecked
   }

   // returns that this node allows children
   public boolean getAllowsChildren()
   {  return true;
   }

   // return the child at specified index
   public TreeNode getChildAt(int childIndex)
      throws IndexOutOfBoundsException
   {  return children.get(childIndex);
   }
```

cont-

-cont

```
   // returns the number of children of this node
   public int getChildCount()
   {  return children.size();
   }

   // returns the index of node or -1 if node not found
   public int getIndex(TreeNode node)
   {  if (node==null)
         throw new IllegalArgumentException();
      else
         return children.indexOf(node);
   }

   // return the parent node or null if this node is the root
   public TreeNode getParent()
   {  return parent;
   }

   // returns whether this node is a leaf
   public boolean isLeaf()
   {  return (getChildCount()==0);
   }

   // add the child node at specified index, updating this node
   // and child node to reflect the change
   public void insert(MutableTreeNode child, int index)
      throws IllegalArgumentException
   {  if (child==null)
         throw new IllegalArgumentException("null child");
      child.removeFromParent(); //remove child from its existing parent
      children.add(index, child); // update list of child nodes
      child.setParent(this); // update the child node
   }

   // removes the child at index from this node, updating this
   // node and child node to reflect the change
   public void remove(int index)
   {  MutableTreeNode child = children.get(index); // update list
      if (child!=null)
         remove(child);
   }
```

cont-

-cont

```
   // remove the specified child from this node, updating this
   // node and child node to reflect the change
   public void remove(MutableTreeNode node)
   {  if (children.remove(node)) // node found and removed
         node.setParent(null);
   }

   // remove this node from its parent, updating this
   // node and its parent node
   public void removeFromParent()
   {  if (this.parent!=null)
      {  parent.remove(this);
         this.parent = null;
      }
   }

   // sets the parent of this node to be newParent
   // but does not update newParent
   public void setParent(MutableTreeNode newParent)
   {  removeFromParent(); // remove this node from its existing parent
      this.parent = newParent;
   }

   // set the element held in this node
   public void setUserObject(Object object)
   {  this.element = (E)object; // unchecked, could throw exception
   }

   public E getUserObject()
   {  return element;
   }

   // returns a string representation of the node and its child nodes
   // in preorder notation
   public String toString()
   {  String output = "" + this.element;
      if (!isLeaf())
      {  output += "[ ";
         for (MutableTreeNode childNode : children)
            output += childNode + " ";
         output += "]";
      }
      return output;
   }
```

cont-

-cont

```
   public static void main(String[] args)
   {  // create some sample nodes
      MutableTreeNode root = new ListTreeNode<String>("A");
      MutableTreeNode nodeB = new ListTreeNode<String>("B");
      MutableTreeNode nodeC = new ListTreeNode<String>("C");
      MutableTreeNode nodeD = new ListTreeNode<String>("D");
      MutableTreeNode nodeE = new ListTreeNode<String>("E");
      MutableTreeNode nodeF = new ListTreeNode<String>("F");
      // build the tree
      nodeB.insert(nodeD, 0);
      nodeB.insert(nodeE, 1);
      nodeC.insert(nodeF, 0);
      root.insert(nodeB, 0);
      root.insert(nodeC, 1);
      System.out.println("Original Tree: " + root);
      root.remove(nodeC);
      nodeB.insert(nodeC, 1);
      System.out.println("Modified Tree: " + root);
   }
}

// utility class to wrap an Iterator object as an Enumeration object
class Enumerator<E> implements Enumeration<E>
{
   private Iterator<E> iterator;

   public Enumerator(Iterator<E> iterator)
   {  this.iterator = iterator;
   }

   public boolean hasMoreElements()
   {  return iterator.hasNext();
   }

   public E nextElement()
   {  return iterator.next();
   }
}
```

```
                    ≪interface≫
                    SortedSet<E>

+comparator() : Comparator<? super E>
+first() : E
+headSet(toElement : E) : SortedSet<E>
+last() : E
+subSet(fromElement : E, toElement : E) : SortedSet<E>
+tailSet(fromElement : E) : SortedSet<E>
```

Tree data structures can conveniently and efficiently be used to implement sorted sets. A *binary search tree* (also called a *sort tree*) is a binary tree for which the element held in each node is greater than that held in any element in its left subtree but less than or equal to all those in its right subtree (hence in a binary search tree there must be a distinction between a left child and a right child). With this property the elements in a binary tree can be efficiently searched, typically requiring an $O(\log_2 n)$ binary search algorithm, rather than the $O(n)$ linear search required for unsorted linear collections.

As an example, the class BinarySearchTree implements a binary tree using a data structure formed from linked BinaryTreeNode nodes. Each node holds an element of generic type E as well as links to its left child and its right child (or null if the node does not have such a child). The algorithms employed by the methods of BinarySearchTree only need to traverse downward through the tree, so it is unnecessary for each node to keep a reference link to its parent node. Besides the four constructors recommended in the documentation for the SortedSet interface, there is a private constructor (used by headSet, subSet, and tailSet) which creates a view of the tree restricted between fromElement (inclusive) and toElement (exclusive).

The contains method inherited from AbstractSet has been overridden to provide a more-efficient searching algorithm:

```
boolean found = false;
BinaryTreeNode currentNode = rootNode;
while (!found && currentNode!=null)
{  int comparison = compare(currentNode.element, element);
   if (comparison==0)
      found = true;
   else if (comparison<0)
      currentNode = currentNode.rightChild;
   else // comparison>0
      currentNode = currentNode.leftChild;
}
```

One complication is that besides allowing elements to be compared using their natural ordering (that given to the elements by their compareTo methods) the class also allows the comparison to be determined by a Comparator object (if specified). To avoid repeated code fragments throughout the class checking whether there is a Comparator, a helper method compare has been written.

The add method uses a loop to traverse down the tree starting from the root to find the node where the new node should be added as a leaf (this method fails to check whether adding a new node might create a closed path, destroying the tree structure and creating problems for other methods). Similarly, the `remove` method uses a loop to traverse down the tree, but it has an extra complication. When a node is removed, care must be taken if that node has children, since they should be retained in the tree. This is accomplished by the `makeReplacement` method. If the node to be removed has no children then it can simply be removed without repercussions. If the node has only one child, then that child can take the place of the node, and the tree keeps its binary search tree property. However, if the node has two children, then possibly neither could take the place of the removed node. To handle this case the `makeReplacement` method searches the descendants on the right subtree of the node for its *successor* (the next larger element).

The right child of the successor node (if there is one) is made the left child of the successor's parent, and the successor is used as the node to replace the removed node, adopting the replaced node's two children as its own.

Note that there are several recursive methods, which is quite commonly found when dealing with tree algorithms. For example, the `countNodes` method:

```
private int countNodes(BinaryTreeNode node)
{    if (node==null)
        return 0;
     else
        return countNodes(node.leftChild) + 1
             + countNodes(node.rightChild);
}
```

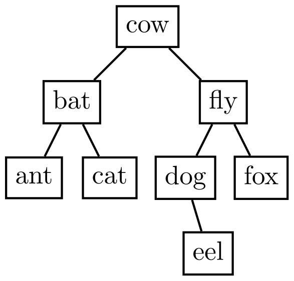
counts the number of descendant nodes of any specified node (including itself). This is straight-forward using recursion, since if the node is null (the base case) then there are no nodes below it, otherwise the number of nodes is the sum of the number of nodes in the left subtree plus that node plus the number of nodes in the right subtree. Thus, the total number of nodes in a binary tree with root `rootNode` can be easily found using `countNodes(rootNode)`.

Unlike one-dimensional *linked lists* and other linear data structures, which are traversed in contiguous order, trees may be traversed in multiple ways. There are several useful routines for navigating trees. Most common are the *in-order*, *pre-order*, and *post-order* tree traversal algorithms. These tree traversal algorithms can be implemented using recursion starting off usually at the root. The general recursive pattern for traversing a (non-empty) binary tree at node N is to do the following steps:

(L) Recursively traverse left subtree of N

(R) Recursively traverse right subtree of N

(N) Process N, if not null.



For example, if elements:

> "cow", "fly", "dog", "bat", "fox",
> "cat", "eel", "ant",

are added in this order then the illustrated binary tree will be built with root holding the element "cow".

An appropriate traversal of this tree can be performed using a combination of the above recursive steps. The L and R steps can be done in any order either a left-to-right traversal or a right-to-left traversal. The following tree traversals assume a left-to-right traversal.

pre-order(NLR) Check if N is not empty. Process or display N. Recursively visit left subtree calling pre-order method passing N's left child. Recursively visit left subtree calling pre-order method passing N's right child. This would navigate the elements in the illustrated tree in the order: cow, bat, ant, cat, fly, dog, eel, fox.

in-order(LNR) Check if N is not empty. Recursively visit left subtree calling pre-order method passing N's left child. Process or display N. Recursively visit left subtree calling pre-order method passing N's right child. This would navigate the elements in the illustrated tree in the order: ant, bat, cat, cow, dog, eel, fly, fox.

post-order(LRN) Check if N is not empty. Recursively visit left subtree calling pre-order method passing N's left child. Recursively visit left subtree calling pre-order method passing N's right child. Process or display N. This would navigate the elements in the illustrated tree in the order: ant, cat, bat, eel, dog, fox, fly, cow.

The `BinaryTreeIterator` in `BinarySearchTree` provides an implementation for an `Iterator`. It's constructor calls the private recursive helper method, `traverseInOrder` to populate a `List` of elements obtained from the tree *in order* (and within view, if subsets are used). Therefore the iterators `next` method returns elements one at a time from this list in a sorted order.

A *level-order* traversal is another tree traversal algorithm, but isn't as simple to implement recursively. The algorithm navigates the tree starting on level 0, the root. The algorithm then moves down the tree, processing elements level by level going left to right. The furthest right-most leaf node being visited last. For example using the same illustrated tree above, a level-order traversal would give the elements back in the following order: cow, bat, fly, ant, cat, dog, fox, eel.

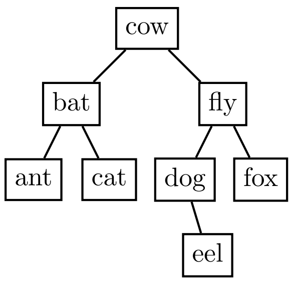
A level-order traversal can be easily implemented using a `Queue`. The algorithm starts off with the root being placed in the queue. Then using a loop, repeatedly dequeue the front element and process or display it. If the currently dequeued element has a left child enqueue that left child, and if the currently dequeued element has a right child enqueue that right child. Process repeats until the queue is empty. The following *pseudo-code* describes a level order traversal.

```
LEVEL ORDER TRAVERSAL($N$)
 1  if N is null
 2      then return .
 3  let $Q$ be a queue
 4  $Q.enqueue(N)$
 5  while $Q.isEmpty() = $ FALSE
 6      then ▷ Dequeue front element $C$
 7              $C \leftarrow Q.dequeue()$
 8              PRINT($C$)
 9              if $C.leftChild \neq null$
10                 then $Q.enqueue(C.leftChild)$
11              if $C.rightChild \neq null$
12                 then $Q.enqueue(C.rightChild)$
13  ▷ level order traversal complete, queue empty
```

Exercise 4.2 (Navigating trees) *Modify the* `BinaryTreeIterator` *in* `BinarySearchTree` *to instead provide elements obtained using alternative tree traversal techniques pre-order, post-order and level-order.*

4.3 Balancing Binary Trees

If a binary tree has n elements and is balanced then there are approximately $\log_2 n$ levels, so the `add`, `remove`, and `contains` methods each require about $\log_2 n$ comparisons (one per level), making them $O(\log_2 n)$ processes. However, the order in which elements are added to a binary search tree affects the shape of the tree, and thus the performance of these methods might drop if there are more levels in the tree than necessary.



For example, if elements:

> "cow", "fly", "dog", "bat", "fox", "cat", "eel", "ant",

are added in this order then the illustrated binary tree will be built with root holding the element "cow". Searching for an element in this balanced tree would require at most $4 \approx \log_2 n$ comparisons.

```
/**
   A class that implements a sorted set collection using a binary
   search tree. Note this implementation of a binary tree does not
   have duplicate (equal) elements.
   This class allows a restricted view of the tree, between
   fromElement (inclusive) and toElement (exclusive)
   @author Andrew Ensor
*/
import java.util.AbstractSet;
import java.util.Collection;
import java.util.Comparator;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.NoSuchElementException;
import java.util.SortedSet;

public class BinarySearchTree<E> extends AbstractSet<E>
   implements SortedSet<E>
{
   private int numElements;
   protected BinaryTreeNode rootNode;
   private Comparator<? super E> comparator;//null for natural ordering
   private E fromElement, toElement; // bounds for visible view of tree

   public BinarySearchTree()
   {  super();
      numElements = 0;
      rootNode = null;
      comparator = null;
      fromElement = null;
      toElement = null;
   }

   public BinarySearchTree(Collection<? extends E> c)
   {  this();
      for (E element : c)
         add(element);
   }

   public BinarySearchTree(Comparator<? super E> comparator)
   {  this();
      this.comparator = comparator;
   }
```

cont-

-cont

```
public BinarySearchTree(SortedSet<E> s)
{  this();
   this.comparator = s.comparator();
   for (E element : s)
      add(element);
}

// private constructor used to create a view of a portion of tree
private BinarySearchTree(BinaryTreeNode rootNode,
   Comparator<? super E> comparator, E fromElement, E toElement)
{  this(comparator);
   this.rootNode = rootNode;
   this.fromElement = fromElement;
   this.toElement = toElement;
   // calculate the number of elements
   this.numElements = countNodes(rootNode);
}

// recursive helper method that counts number of descendants of node
private int countNodes(BinaryTreeNode node)
{  if (node==null)
      return 0;
   else
      return countNodes(node.leftChild) + 1
         + countNodes(node.rightChild);
}

// helper method that determines whether an element is within the
// specified view
private boolean withinView(E element)
{  boolean inside = true;
   if (fromElement!=null && compare(element, fromElement)<0)
      inside = false;
   if (toElement!=null && compare(element, toElement)>=0)
      inside = false;
   return inside;
}
```

cont-

-cont

```
   // adds the element to the sorted set provided it is not already in
   // the set, and returns true if the sorted set did not already
   // contain the element
   public boolean add(E o)
   {  if (!withinView(o))
         throw new IllegalArgumentException("Outside view");
      BinaryTreeNode newNode = new BinaryTreeNode(o);
      boolean added = false;
      if (rootNode==null)
      {  rootNode = newNode;
         added = true;
      }
      else
      {  // find where to add newNode
         BinaryTreeNode currentNode = rootNode;
         boolean done = false;
         while (!done)
         {  int comparison = compare(o, currentNode.element);
            if (comparison<0) // newNode is less than currentNode
            {  if (currentNode.leftChild==null)
               {  // add newNode as leftChild
                  currentNode.leftChild = newNode;
                  done = true;
                  added = true;
               }
               else
                  currentNode = currentNode.leftChild;
            }
            else if (comparison>0)//newNode is greater than currentNode
            {  if (currentNode.rightChild==null)
               {  // add newNode as rightChild
                  currentNode.rightChild = newNode;
                  done = true;
                  added = true;
               }
               else
                  currentNode = currentNode.rightChild;
            }
            else if (comparison==0) // newNode equal to currentNode
               done = true; // no duplicates in this binary tree impl.
         }
      }
      if (added) numElements++;
      return added;
   }
```

cont-

-cont

```
   // performs a comparison of the two elements, using the comparator
   // if not null, otherwise using the compareTo method
   private int compare(E element1, E element2)
   {  if (comparator!=null)
         return comparator.compare(element1, element2);
      else if (element1!=null && element1 instanceof Comparable)
         return ((Comparable)element1).compareTo(element2); //unchecked
      else if (element2!=null && element2 instanceof Comparable)
         return -((Comparable)element2).compareTo(element1);//unchecked
      else return 0;
   }

   // remove the element from the sorted set and returns true if the
   // element was in the sorted set
   public boolean remove(Object o)
   {  boolean removed = false;
      E element = (E)o; // unchecked, could throw exception
      if (!withinView(element))
         throw new IllegalArgumentException("Outside view");
      if (rootNode!=null)
      {  // check if root to be removed
         if (compare(element, rootNode.element)==0)
            rootNode = makeReplacement(rootNode);
         else
         {  // search for the element o
            BinaryTreeNode parentNode = rootNode;
            BinaryTreeNode removalNode;
            // determine whether to traverse to left or right of root
            if (compare(element, rootNode.element)<0)
               removalNode = rootNode.leftChild;
            else // compare(element, rootNode.element)>0
               removalNode = rootNode.rightChild;
            while (removalNode!=null && !removed)
            {  // determine whether the removalNode has been found
               int comparison = compare(element, removalNode.element);
               if (comparison==0)
               {  if (removalNode==parentNode.leftChild)
                     parentNode.leftChild
                        = makeReplacement(removalNode);
                  else // removalNode==parentNode.rightChild
                     parentNode.rightChild
                        = makeReplacement(removalNode);
                  removed = true;
               }
               else // determine whether to traverse to left or right
               {  parentNode = removalNode;
                  if (comparison<0) removalNode =removalNode.leftChild;
                  else removalNode = removalNode.rightChild;
               }
```

cont-

-cont

```
            }
         }
      }
      if (removed) numElements--;
      return removed;
   }

   // helper method which removes removalNode (presumed not null) and
   // returns a reference to node that should take place of removalNode
   private BinaryTreeNode makeReplacement(BinaryTreeNode removalNode)
   {  BinaryTreeNode replacementNode = null;
      // check cases when removalNode has only one child
      if (removalNode.leftChild!=null && removalNode.rightChild==null)
         replacementNode = removalNode.leftChild;
      else if (removalNode.leftChild==null
         && removalNode.rightChild!=null)
         replacementNode = removalNode.rightChild;
      // check case when removalNode has two children
      else if (removalNode.leftChild!=null
         && removalNode.rightChild!=null)
      {  // find the inorder successor and use it as replacementNode
         BinaryTreeNode parentNode = removalNode;
         replacementNode = removalNode.rightChild;
         if (replacementNode.leftChild==null)
            // replacementNode can be pushed up one level to replace
            // removalNode, move the left child of removalNode to be
            // the left child of replacementNode
            replacementNode.leftChild = removalNode.leftChild;
         else
         {  //find left-most descendant of right subtree of removalNode
            do
            {  parentNode = replacementNode;
               replacementNode = replacementNode.leftChild;
            }
            while (replacementNode.leftChild!=null);
            // move the right child of replacementNode to be the left
            // child of the parent of replacementNode
            parentNode.leftChild = replacementNode.rightChild;
            // move the children of removalNode to be children of
            // replacementNode
            replacementNode.leftChild = removalNode.leftChild;
            replacementNode.rightChild = removalNode.rightChild;
         }
      }
      // else both leftChild and rightChild null so no replacementNode
      return replacementNode;
   }
```

cont-

-cont

```
   public Iterator<E> iterator()
   {  return new BinaryTreeIterator(rootNode);
   }

   // returns the number of elements in the tree
   public int size()
   {  return numElements;
   }

   // removes all elements from the collection
   public void clear()
   {  rootNode = null; // all nodes will be garbage collected as well
   }

   // overridden method with an efficient O(log n) search algorithm
   // rather than the superclasses O(n) linear search using iterator
   public boolean contains(Object o)
   {  boolean found = false;
      E element = (E)o; // unchecked, could throw exception
      if (!withinView(element))
         return false;
      BinaryTreeNode currentNode = rootNode;
      while (!found && currentNode!=null)
      {  int comparison = compare(currentNode.element, element);
         if (comparison==0)
            found = true;
         else if (comparison<0)
            currentNode = currentNode.rightChild;
         else // comparison>0
            currentNode = currentNode.leftChild;
      }
      return found;
   }

   // returns the Comparator used to compare elements or null if
   // the element natural ordering is used
   public Comparator<? super E> comparator()
   {  return comparator;
   }
```

cont-

-cont

```
// returns the first (lowest) element currently in sorted set that
// is at least as big as fromElement, returns null if none found
public E first()
{  if (rootNode==null)
      throw new NoSuchElementException("empty tree");
   // find the least descendant of rootNode that is at least
   // as big as fromElement by traversing down tree from root
   BinaryTreeNode currentNode = rootNode;
   BinaryTreeNode leastYetNode = null; // smallest found so far
   while (currentNode!=null)
   {  if (compare(currentNode.element, fromElement)>=0)
      {  if (compare(currentNode.element, toElement)<0)
            leastYetNode = currentNode;
         // move to the left child to see if a smaller element okay
         // since all in right subtree will be larger
         currentNode = currentNode.leftChild;
      }
      else // compare(currentNode.element, fromElement)<0
      {  // move to the right child since this element too small
         // so all in left subtree will also be too small
         currentNode = currentNode.rightChild;
      }
   }
   if (leastYetNode==null) // no satisfactory node found
      return null;
   else
      return leastYetNode.element;
}

public SortedSet<E> headSet(E toElement)
{  return subSet(null, toElement);
}

// returns the last (highest) element currently in sorted set that
// is less than toElement, return null if none found
public E last()
{  if (rootNode==null)
      throw new NoSuchElementException("empty tree");
   // find the greatest descendant of rootNode that is less than
   // toElement by traversing down tree from root
   BinaryTreeNode currentNode = rootNode;
   BinaryTreeNode greatestYetNode = null; // greatest found so far
```

cont-

-cont

```
      while (currentNode!=null)
      {  if (compare(currentNode.element, toElement)<0)
         {  if (compare(currentNode.element, fromElement)>=0)
               greatestYetNode = currentNode;
            // move to the right child to see if a greater element okay
            // since all in left subtree will be smaller
            currentNode = currentNode.rightChild;
         }
         else // compare(currentNode.element, toElement)>=0
         {  // move to the left child since this element too large
            // so all in right subtree will also be too large
            currentNode = currentNode.leftChild;
         }
      }
      if (greatestYetNode==null) // no satisfactory node found
         return null;
      else
         return greatestYetNode.element;
   }


   public SortedSet<E> subSet(E fromElement, E toElement)
   {  return new BinarySearchTree<E>(rootNode, comparator, fromElement,
         toElement);
   }

   public SortedSet<E> tailSet(E fromElement)
   {  return subSet(fromElement, null);
   }

   // outputs the elements stored in the full binary tree (not just
   // the view) using inorder traversal
   public String toString()
   {  return "Tree: " + rootNode;
   }

   public static void main(String[] args)
   {  // create the binary search tree
      SortedSet<String> tree = new BinarySearchTree<String>();
      // build the tree
      tree.add("cow");tree.add("fly");tree.add("dog");tree.add("bat");
      tree.add("fox");tree.add("cat");tree.add("eel");tree.add("ant");
      System.out.println("Original Tree: " + tree);
      tree.remove("owl");
      tree.remove("cow");
      tree.add("owl");
      System.out.println("Modified Tree: " + tree);
```

cont-

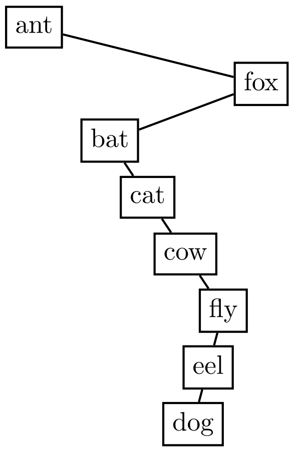
-cont

```
      SortedSet<String> subtree = tree.subSet("cat", "fox");
      System.out.print("Subtree iteration: ");
      Iterator<String> i = subtree.iterator();
      while (i.hasNext())
      {  System.out.print(i.next());
         if (i.hasNext()) System.out.print(", ");
      }
      System.out.println();
      System.out.println("first element in subtree: "+subtree.first());
      System.out.println("last element in subtree: " + subtree.last());
   }

   // inner class that represents a node in the binary tree
   // where each node consists of the element and links to
   // left child and right child (no need for link to parent)
   protected class BinaryTreeNode
   {
      public BinaryTreeNode leftChild, rightChild;
      public E element;

      public BinaryTreeNode(E element)
      {  this.element= element;
         leftChild = null;
         rightChild = null;
      }

      // returns a string representation of the node and
      // its children using inorder (left-this-right) traversal
      public String toString()
      {  String output = "[";
         if (leftChild!=null)
            output += "" + leftChild;
         output += "" + element;
         if (rightChild!=null)
            output += "" + rightChild;
         output += "]";
         return output;
      }
   }
```

cont-

-cont

```
   // inner class that represents an Iterator for a binary tree
   private class BinaryTreeIterator implements Iterator<E>
   {
      private LinkedList<E> list;
      private Iterator<E> iterator;

      public BinaryTreeIterator(BinaryTreeNode rootNode)
      {  // puts the elements in a linked list using inorder traversal
         list = new LinkedList<E>();
         traverseInOrder(rootNode);
         iterator = list.iterator();
      }

      // recursive helper method that traverses the subtree from node
      // adding the elements to the list collection
      private void traverseInOrder(BinaryTreeNode node)
      {  if (node!=null)
         {  traverseInOrder(node.leftChild);
            if (withinView(node.element))
               list.add(node.element);
            traverseInOrder(node.rightChild);
         }
      }

      public boolean hasNext()
      {  return iterator.hasNext();
      }

      public E next()
      {  return iterator.next();
      }

      public void remove()
      {  throw new UnsupportedOperationException();
      }
   }
}
```



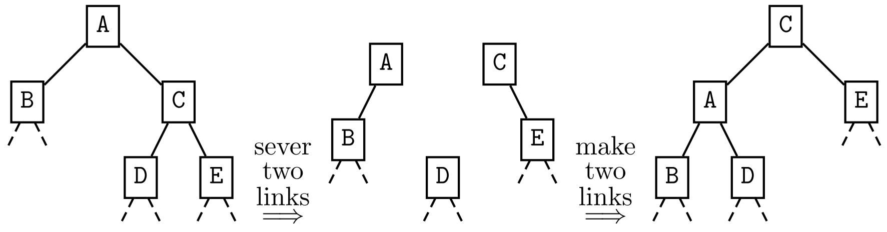
If instead the elements were added in an order such as:

> "ant", "fox", "bat", "cat", "cow", "fly", "eel", "dog",

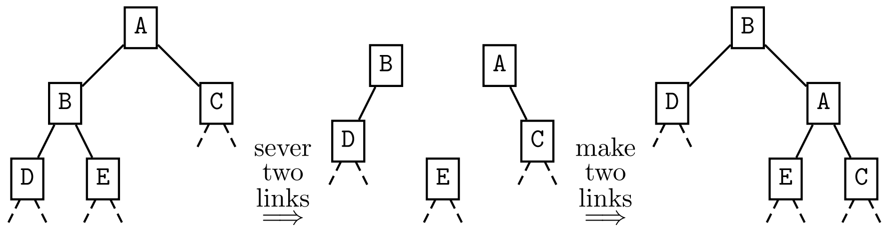
then the binary tree is built with "ant" as its root. In this case the tree is called *degenerate* in the sense that each parent has only one child, with no branching, and resembles more a linked list than a tree. To search this tree for an element requires up to $8 = n$ comparisons. It thus becomes important when adding or removing nodes that the tree be kept balanced, or nearly balanced, with as few levels as practically possible, in order to avoid the tree's performance dropping to $O(n)$.

If after an addition or removal of a node it is determined that the tree needs to be balanced then there are two *rotations* of the tree that can be used individually or in conjunction to make the tree more balanced.

A *left rotation* is used to promote the right child of a node up a level and demote the left child down one level. It does this in two steps. First it breaks the link between the node to be rotated left and its right child, and between that child and its own left child (if present). Then it makes the right child the parent of the original node and reattaches its detached child as the right child of the original node. Following these steps ensures that the required ordering of a binary tree is maintained.



A *right rotation* reverses this process by promoting the left child up a level and demoting the right child down a level (the mirror image of a left rotation), first severing two links and then making two other links.



Typically a left rotation is used around a node whose right subtree has several more levels than its left subtree, particularly when the right child itself has a long right subtree. If instead the right child has a long left subtree then first a right rotation around that child should be performed to balance it better, followed by a left rotation around the node. Collectively, this is known as a *rightleft* rotation.

Similarly, a right rotation is used around a node whose left child itself has a long left subtree. If the left child has a long right subtree then a *leftright* rotation is used, consisting of a left rotatation around the left child followed by a right rotation around the node.

There are several alternative algorithms that are widely used to determine when a tree needs balancing. An *AVL tree* is a binary tree where each node is assigned a *balance factor*, which is the height of its left subtree minus the height of its right subtree. Meaning that each `Node` needs to keep a field, `height` to represent the maximum distance of a path from its left or right subtree to a leaf node. After the addition or removal of a node the balance factor of the ancestor nodes is recalculated (so it is convenient for each node in an AVL tree to keep a link to its parent). If the balance factor is not -1, 0, or 1 for a node, then checking the balance factor for it and it's child nodes determines which type of rotation is required to balance the tree.

The basic steps for an AVL insert can be described as follows:

- Let w be a newly inserted node, perform normal binary search tree insertion of w adding it as a leaf node.
- Starting from node w, traverse back up to its parent x.
- Repeatedly move up the tree recalculating each nodes balance factor towards the root.
- Keep following parent links until we reach the root or find a node that is unbalanced z with balance factor $+2$ or -2. Care must be taken as z may be the root or have a parent link.
- Perform the approprate rotation of z by looking at the balance factor of its child y on the path back to w. There are four possible rotations:
  1. If $z = +2$ and $y > 0$, meaning the outer left subtree is too heavy, then a **right rotation** of z performed.
  2. If $z = +2$ and $y < 0$, meaning the inner left subtree is too heavy, then a **left rotation** of y followed by a **right rotation** of z performed
  3. If $z = -2$ and $y < 0$, meaning the outer right subtree is too heavy, then a **left rotation** of z performed.
  4. If $z = -2$ and $y > 0$, meaning the inner right subtree is too heavy, then a **right rotation** of y followed by a **left rotation** of z performed.

For example, the following illustrated diagrom below shows z unbalanced with a balance factor of $+2$. Its child y on the path to w is checked and found to be a positive number. So only a right rotation of z is required.

```
      z(+2)                                  y(0)
       /   \                                /    \
    y(+1)  N2                            x(+1)  z(0)
    /   \         -Right Rotate(z)->     /       /   \
 x(+1)  N1                              w       N1   N2
 /
w
```

Another example in the next illustrated diagram shows z unbalanced with a balance factor of $+2$. However this time its child y on the path to w is found to be a negative number. So a leftright rotation of y and then z is required.

```
    z(+2)                    z(+2)                 x(0)
    /   \                   /    \                /   \
  y(-1)  N2               x(+2)   N2            y(0)  z(-1)
  /   \       -LR(y)->    /           -RR(z)->  / \      \
N1   x(+1)             y(0)                   N1   w     N2
     /                  / \
    w                  N1  w
```

The two diagrams and weights can be mirrored to show the left rotation and rightleft rotaion of z.

Instead of each node keeping reference to its parent, we can use a modified recursive binary search tree `insert` method. After insertion we get all references to ancestor nodes one by one in a bottom up manner as activation records are popped off the stack when recursive calls finish. After insertion of a new node, each nodes balance factor is recalculated working back up the tree. One of the four appropriate rotations is then used when a node with a balance factor < -1 or > 1 is encountered.

A recursive insert using an AVL tree can be described with the following psuedo-code:

```
INSERT(Node:n, E:data)
 1  if $n$ is $null$
 2      then return new Node(data)
 3  if $data < n.data$
 4      then $n.left \leftarrow INSERT(n.left, data)$
 5      else  if $data > n.data$
 6      then $n.right \leftarrow INSERT(n.right, data)$
 7      else
 8      then return $n$ ▷ duplicates not allowed
 9  $n.height \leftarrow \mathrm{MAX}(n.left.height, n.right.height) + 1$
10  $balance \leftarrow (n.left.height - n.right.height)$ ▷ use 0 if left or right null
11  if $balance > 1$ AND $data < node.left.data$
12      then return RIGHTROTATE(n)
13  if $balance < -1$ AND $data > node.right.data$
14      then return LEFTROTATE($n$)
15  if $balance > 1$ AND $data > node.left.data$
16      then $n.left \leftarrow$ LEFTROTATE($n.left$)
17           return RIGHTROTATE($n$)
18  if $balance < -1$ AND $data < node.right.data$
19      then $n.right \leftarrow$ RIGHTROTATE($n.right$)
20           return LEFTROTATE($n$)
21  return n
```

A common alternative to using an AVL tree is to use a *red/black* tree. In a red/black tree each node is assigned one of the two colours subject to the following restrictions:

- the root is black,
- children of a red node must be black,
- there must be the same number of black nodes along any path from the root to a leaf.

The maximum height in a red/black tree is less than $2 \log_2 n$, so unlike an AVL tree it is not required to be actually balanced. Adding or removing a node follows an algorithms which recolours the ancestor nodes as it progressively moves up the tree (so the nodes in a red/black tree should also keep links to their parents), using the colours to determine when a rotation is required. The Collections API class called `TreeSet` actually uses a red/black binary search tree for its implementation of a sorted set to ensure the basic operations are $O(\log_2 n)$.

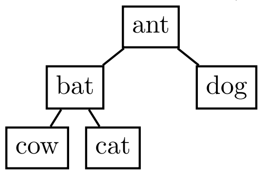
Exercise 4.3 (Rotations in a Binary Tree) *Write methods that take a node in a binary tree as a parameter and perform a left, right, rightleft, or a leftright rotation around that node, returning the node that is now uppermost.*

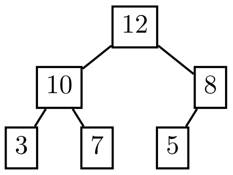
4.4 Heaps

Reading: pp472-492

```
/**
   An interface that defines the abstract data type
   for an ascending heap collection of elements with type E
*/
import java.util.Comparator;

public interface HeapADT<E>
{
   // adds one element to the heap in correct position to maintain heap
   // and returns true if element successfully added
   public boolean add(E element);
   // removes and returns smallest element in heap and maintain heaps
   public E removeMin();
   // returns without removing the smallest element in heap
   public E getMin();
   // returns the Comparator used to order collection, or null if
   // collection is sorted according to elements' natural order
   public Comparator<? super E> comparator();
   // returns true if this heap contains no elements
   public boolean isEmpty();
   // returns the number of elements in this heap
   public int size();
}
```

An *(ascending) heap* (also called a *minheap*) is a binary tree in which the element held in each node is less than or equal to that held in either of its child nodes, and which is balanced so that all the leaf nodes at the bottom level are as far left as possible. If the ordering is instead reversed so that each parent is greater than or equal to its child nodes then the heap is a *descending heap* (also called a *maxheap*). For example, of the following two binary trees, the one on the left is an ascending heap and the other a descending heap (using the elements' natural order):





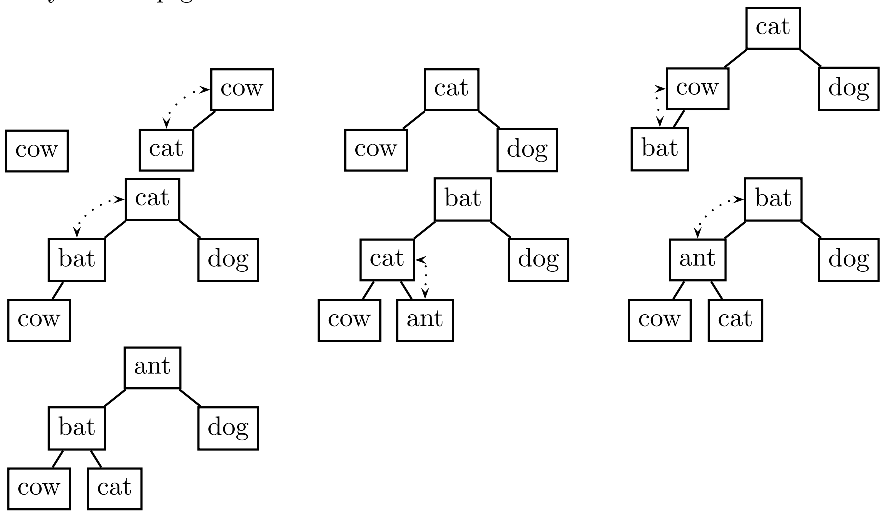
A heap is a binary tree, but due to the requirements on the order of elements and the shape of the tree its abstract data type typically would not permit elements to be inserted at arbitrary nodes of the tree. Instead, like binary search trees, when an element is added the heap itself determines a suitable position for the element. Unlike binary search trees, only the root of a heap is usually allowed to be accessed or removed, rather than arbitrary elements (since the position of elements in a heap does not greatly assist when searching for an element). The interface `HeapADT` describes a suitable abstract data type for a heap.

Typically, when an element is added to a heap, it is first placed as a leaf node in the required position so as to maintain the shape of the heap. To reestablish the required order the new element is repeatedly compared with its parent, and swapped if smaller than its parent, moving up the heap until either a smaller parent is encountered or else it becomes the root. Since a heap is balanced it has approximately $\log_2 n$ levels, so the process of adding a new element could involve up to about $\log_2 n$ comparisons and swaps.

For example, suppose a heap is to be built by adding the strings:

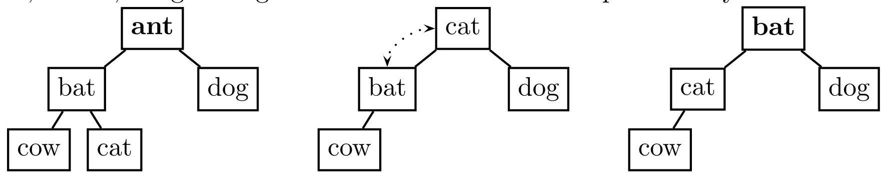
"cow", "cat", "dog", "bat", "ant".

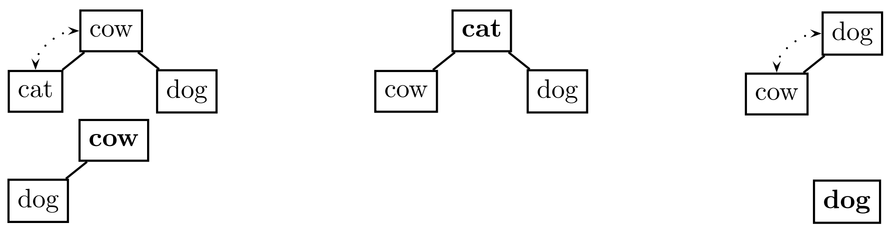
The heap is started using the string "cow". Then "cat" is added as the left child of "cow". Since "cat" < "cow" these two vertices need to be swapped to maintain the ordering of the heap. Next, "dog" is added as the right child of "cat", and does not need to be swapped with "cat" as the ordering has been maintained. Then the next node "bat" is added as the left child of "cow". Continuing in this way the heap grows as follows:



Due to the ordering of an ascending heap the smallest element is always at the root. When the root is removed typically it is replaced with the last leaf node so as to retain the shape of the heap. To reestablish the required order the former leaf element is repeatedly compared with the smaller of its children, and swapped if greater than the smaller child (using the smaller child avoids further swaps down the other subtree), moving down the heap until either the ordering is reestablished or else it becomes a leaf. The process of removing the root could involve up to about $\log_2 n$ comparisons and swaps, so it too is $O(\log_2 n)$.

For example, considering the previous heap the least element "ant" is at the root. Removing the root and replacing it with the last leaf node "cat" retains the shape of the heap but destroys the required ordering. To restore the ordering "cat" is swapped with its child "bat" (which is smaller than the other child "dog"). Once the ordering is restored, the new smallest value "bat" is in the root. If this root is removed it gets replaced with "cow" (to maintain the shape of the heap). Continuing in this way results in the elements "ant", "bat", "cat", "cow", "dog" being removed in order as the heap is destroyed:





Since adding and removing a single element are both $O(\log_2 n)$ operations, the entire process of adding n elements to a heap and then removing them one at a time in order gives an efficient $O(n \log_2 n)$ sorting algorithm, commonly known as *heap sort*.

Like other types of trees, a heap can be implemented using linked nodes. Since addition of an element requires a traversal up the tree toward the root, each node should have a link to its parent (hence the `ListTreeNode` class from Section 4.1 would need modifications), as well as links to its children.

Although most trees are more conveniently implemented using linked nodes, heaps provide an interesting exception. As a heap is a binary tree, if an element is stored at index i, then its left child can be placed at index $2i + 1$ and its right child at $2i + 2$. Due to the shape of a heap, the elements would be stored contiguously in an array from index 0 (the root) to index $n - 1$ (the last leaf).

The class `ArrayHeap` demonstrates how a heap can be implemented using an array data structure. Its `add` method first places the new element at the last position in the array (expanding capacity if necessary), and calls the helper method `restoreHeapAfterAdd` to restore the ordering, which uses a loop to traverse up the tree:

```
int currentIndex = numElements-1; // start with added element
int parentIndex = (currentIndex-1)/2;
while (currentIndex>0 && compare(elements[currentIndex],
   elements[parentIndex])<0)
{  // swap this node with its parent node
   temp = elements[currentIndex];
   elements[currentIndex] = elements[parentIndex];
   elements[parentIndex] = temp;
   // move up to the parent node
   currentIndex = parentIndex;
   parentIndex = (currentIndex-1)/2;
}
```

The `removeMin` method first places the element that was at the last position in the array at index 0 (the root). Then it calls the helper method `restoreHeapAfterRemove` to reestablish the ordering, which uses a loop to traverse down the tree starting at the root:

```
int currentIndex = 0; // start with root element
int smallerChild = getSmallerChildIndex(currentIndex);
while (smallerChild>=0 && compare(elements[currentIndex],
   elements[smallerChild])>0)
{  // swap this node with its smaller child node
   temp = elements[currentIndex];
   elements[currentIndex] = elements[smallerChild];
```

```
/**
   A class that implements a heap collection using an array data
   structure, where a node at index i has its left child stored
   at index 2i+1 and its right child stored at index 2i+2,
   so its parent is at index (i-1)/2
   @author Andrew Ensor
*/
import java.util.Collection;
import java.util.Comparator;
import java.util.NoSuchElementException;

public class ArrayHeap<E> implements HeapADT<E>
{
   private final int INITIAL_CAPACITY = 15; // initial heap height is 3
   private int numElements;
   private E[] elements;
   private Comparator<? super E> comparator;

   public ArrayHeap()
   {  this(null);
   }

   public ArrayHeap(Comparator<? super E> comparator)
   {  this.comparator = comparator;
      numElements = 0;
      elements = (E[])(new Object[INITIAL_CAPACITY]); // unchecked
   }

   public ArrayHeap(Collection<? extends E> c,
      Comparator<? super E> comparator)
   {  this(comparator);
      for (E element : c)
         add(element);
   }

   // adds one element to the heap in correct position to maintain heap
   public boolean add(E element)
   {  if (numElements >= elements.length)
         expandCapacity();
      elements[numElements++] = element;
      if (numElements>1)
         restoreHeapAfterAdd();
      return true;
   }
```

cont-

-cont

```
   // helper method which increase the current size of the array to
   // handle another level of the heap
   private void expandCapacity()
   {  E[] largerArray
         = (E[])(new Object[elements.length*2+1]); // unchecked
      // copy the elements array to the largerArray
      for (int i=0; i<numElements; i++)
         largerArray[i] = elements[i];
      elements = largerArray;
   }

   // helper method that restores the heap ordering after an element
   // has been added to index numElements-1, moving the element upward
   // toward the root until order is restored
   private void restoreHeapAfterAdd()
   {  int currentIndex = numElements-1; // start with added element
      int parentIndex = (currentIndex-1)/2;
      while (currentIndex>0 && compare(elements[currentIndex],
         elements[parentIndex])<0)
      {  // swap this node with its parent node
         E temp = elements[currentIndex];
         elements[currentIndex] = elements[parentIndex];
         elements[parentIndex] = temp;
         // move up to the parent node
         currentIndex = parentIndex;
         parentIndex = (currentIndex-1)/2;
      }
   }

   // removes and returns smallest element in heap and maintain heap
   // or throws a NoSuchElementException if heap is empty
   public E removeMin() throws NoSuchElementException
   {  if (numElements==0)
         throw new NoSuchElementException();
      E minElement = elements[0];
      // replace the root of heap with its last element
      elements[0] = elements[--numElements];
      elements[numElements] = null;
      restoreHeapAfterRemove();
      return minElement;
   }
```

cont-

-cont

```
// helper method that restores the heap ordering after an element
// has been removed, moving the root element downward until order
// is restored
private void restoreHeapAfterRemove()
{  int currentIndex = 0; // start with root element
   int smallerChild = getSmallerChildIndex(currentIndex);
   while (smallerChild>=0 && compare(elements[currentIndex],
      elements[smallerChild])>0)
   {  // swap this node with its smaller child node
      E temp = elements[currentIndex];
      elements[currentIndex] = elements[smallerChild];
      elements[smallerChild] = temp;
      // move down to the smaller child node
      currentIndex = smallerChild;
      smallerChild = getSmallerChildIndex(currentIndex);
   }
}

// helper method which returns index of smaller child of element
// at specified index, or returns -1 if no children
private int getSmallerChildIndex(int parentIndex)
{  int smallerChild = -1;
   int leftChild = 2*parentIndex+1;
   int rightChild = 2*parentIndex+2;
   if (leftChild<numElements)
   {  // the node has a left child
      if (rightChild<numElements)
      {  // the node also has a right child
         if (compare(elements[leftChild], elements[rightChild])<0)
            smallerChild = leftChild;
         else
            smallerChild = rightChild;
      }
      else // only a left child
         smallerChild = leftChild;
   }
   return smallerChild;
}
```

cont-

-cont

```
// returns without removing the smallest element in heap
// or throws a NoSuchElementException if heap is empty
public E getMin() throws NoSuchElementException
{  if (numElements>0)
      return elements[0];
   else
      throw new NoSuchElementException();
}

// returns the Comparator used to order collection, or null if
// collection is sorted according to elements' natural order
public Comparator<? super E> comparator()
{  return comparator;
}

// returns true if this heap contains no elements
public boolean isEmpty()
{  return numElements==0;
}

// returns the number of elements in this heap
public int size()
{  return numElements;
}

// returns a string representation of the elements in the heap
public String toString()
{  String output = "[";
   for (int i=0; i<numElements; i++)
   {  output += ""+elements[i];
      if (i+1<numElements)
         output += ",";
   }
   output += "]";
   return output;
}
```

cont-

-cont

```
   // performs a comparison of the two elements, using the comparator
   // if not null, otherwise using the compareTo method
   private int compare(E element1, E element2)
   {  if (comparator!=null)
         return comparator.compare(element1, element2);
      else if (element1!=null && element1 instanceof Comparable)
         return ((Comparable)element1).compareTo(element2); //unchecked
      else if (element2!=null && element2 instanceof Comparable)
         return -((Comparable)element2).compareTo(element1);//unchecked
      else return 0;
   }

   public static void main(String[] args)
   {  HeapADT<String> heap = new ArrayHeap<String>();
      heap.add("cow");
      heap.add("cat");
      heap.add("dog");
      heap.add("bat");
      heap.add("ant");
      System.out.println("Elements sorted using heap sort");
      while (!heap.isEmpty())
      {  System.out.print(heap);
         System.out.println(" (smallest = " + heap.removeMin() + ")");
      }
   }
}
```

```
        elements[smallerChild] = temp;
        // move down to the smaller child node
        currentIndex = smallerChild;
        smallerChild = getSmallerChildIndex(currentIndex);
    }
```

Besides using heaps for sorting, they are also convenient for implementing priority queues. A *priority queue* is a collection whose elements are each assigned a priority (either implementing the `Comparable` interface or using a `Comparable` object), which has methods similar to a queue, but which always returns the element with highest priority first (unlike the usual first in, first out for a queue). Priority queues are often used in the implementation of algorithms. Typically the problem to be solved consists of a number of subtasks and the solution strategy involves prioritizing the subtasks and then performing those subtasks in the order of their priorities. One implementation of priority queues called `PriorityQueue` is available in the `java.util` package.

Exercise 4.4 (Implementing a Priority Queue) *Write a class called* `Task` *which holds an element of generic type* `E` *and an integer priority. Then design and implement a priority queue for* `Task` *objects which has methods* `enqueue`*,* `dequeueMin`*, and* `findMin`*.*

Chapter 5

Hashing

5.1 Hash Tables

Reading: pp516-530

A *hashing function* is a function which assigns an `int` value, called a *hash code*, to each object that is an instance of some class. The `Object` method `hashCode` uses a hashing function to return a hash code for any object, which typically converts the internal memory address of the object into an `int` value. However, classes that make use of hash codes usually override this method and provide their own hashing function using some attribute(s) of the object (known as the *key* for the hashing function). For example, the `String` class overrides the `hashCode` method so that a string "$c_0c_1c_2 \ldots c_{n-1}$" has the hash code:

$$c_0 \cdot 31^{n-1} + c_1 \cdot 31^{n-2} + c_2 \cdot 31^{n-3} + \ldots + c_{n-1},$$

where each character is converted to its Unicode equivalent, and the empty string is assigned the hash code 0. Hence the string `"AB"` has hash code $2081 = 65 * 31 + 66$.

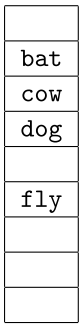
Note that a hashing function is not required to be one-to-one (injective), so it is possible for two distinct objects to share the same hash code. Ideally though the hash codes should be approximately uniformly distributed, to avoid the situation where certain hash codes are shared by many objects but most potential codes are not used at all. There are many techniques commonly used to make hashing functions from keys, but most are based on some combination of extraction, division, and folding.

The *extraction* method only uses a certain part of the key for the calculation of hash codes. For example, if an object holds the name of a person then one possible hashing function might just extract the first letter of the surname starting with hash code 0 for the character `'A'`. Then the name `Joseph Chase` would be assigned the hash code 2, whereas `John Lewis` would be assigned the hash code 11, as would the name `William Loftus`.

Another technique is the *division* method, which uses the integer remainder operator `%` to ensure that the hash code is within certain bounds. Using a prime number as the divisor helps to distribute the hash codes more uniformly. For example, the `String` hashing function could give hash values as large as $2^{31} - 1$. By taking the remainder modulo 101, one obtains a value between 0 and 100 inclusive, so for example the string `"AB"` would be assigned hash code 61 =`"AB".hashCode()%101`.

If a key can be broken into parts, then those parts could be *folded* (combined) together to form a hashing function. For example, if the key is a six digit number, then it could be broken into three two digit numbers. These numbers could then be added together to give a hash code between 0 and 297 inclusive. Alternatively, one or more of the two digit numbers could have its digits reversed before adding, it could be squared, or perhaps rather than adding an exclusive-or might be used.

A *hash table* is a data structure for a set collection where elements are stored in an array at indices determined by their hash code. In a hash table each index in the array is called a *bucket* or a *bin*.



As a simple example suppose the strings:

“cow”, “fly”, “dog”, “bat”

are to be stored in a hash table with 9 buckets, using a hashing function which takes the first letter of the string and subtracts off `'a'`. Since “cow” would hash to 2, it is placed at this index in the hash table array. To then add “fly” to the hash table, its hash code would be calculated to be 5, so it would be placed at that index. Similarly, “dog” would be placed at index 3, and “bat” at index 1, leaving the indices 0, 4, 6, 7, 8 available for further elements.
If an element is to be added which has a hash code larger than the capacity of the hash table the usual strategy is to take the remainder modulo the capacity to obtain a valid index. Hence an element such as “rat”, with hash code 17, would be stored at index 8 = 17%9.

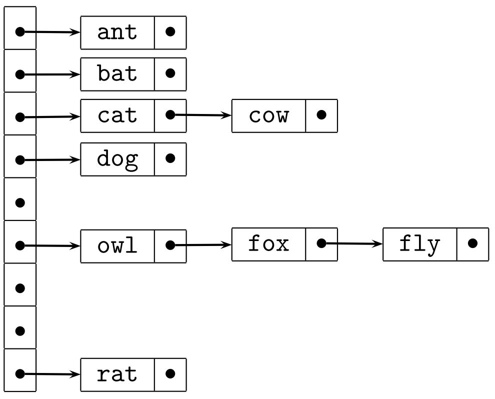
A hash table can provide very efficient access to the elements in its collection, since to determine whether a specified element is in the hash table, its hash code is calculated, and then the hash table is checked to see if the element is stored at that index. This can be a constant time $O(1)$ process, avoiding the need for a linear or binary search, as long as the hashing is one-to-one for the elements in the collection, termed *perfect hashing*, so that no two elements are allocated the same index.

A *collision* is said to occur when a hashing function assigns the same index to two distinct elements. For example, attempting to add “fox” or “cat” to the previous hash table would cause a collision, since the buckets at indices 5 and 2 are already occupied. While it might sometimes be possible to devise a perfect hashing function so that collisions are avoided, a hashing function that results in an occasional collision is not a significant problem if dealt with correctly.

The initial capacity for a hash table is typically chosen to be about 150% of the expected number of elements. While this is wasteful of memory, it does reduce the frequency of collisions, which even for a reasonable hashing function get more and more frequent as the table fills up, reducing the efficiency of the table. For this reason a hash table usually expands its capacity (called *dynamic* *resizing*) well before it reaches capacity. The percentage that the hash table will fill up before it resizes is called its *load factor*, and is usually taken to be about 75%. Hash tables represent a sacrifice of memory for the sake of speed - they are certainly not the most memory efficient means of storing data, but they can provide very fast lookup times.

Unless a hash table is using a perfect hashing function, it must have a strategy for resolving collisions when they occur. Three common alternative strategies that a hash table data structure might use are: chaining the colliding elements together, allocating overflow areas for extra elements, or using open addressing. Whichever strategy is used, the hash table must have an algorithm for finding elements that are not placed at the desired index due to collisions.

One simple strategy called *chaining* uses a singly linked list in each bucket to store all the elements whose hash code allocates them that index in the array. When an element is added to the hash table it is just added to the linked list at that bucket, typically being added at the head before any other elements with the same hash code that have already been added. Searching for an element requires computing the element's hash code and then traversing the linked list at the corresponding bucket. Similarly, it is straight-forward to remove elements from the hash table. So long as the hash table does not degenerate into a few very long linked lists the performance of a hash table that uses chaining is approximately $O(1)$ for adding, searching, and removing elements, and there is no limit on the number of collisions that can be handled.

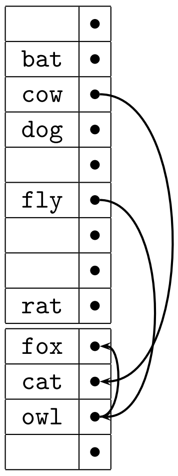


For example if the strings:

"cow", "fly",
"dog", "bat",
"fox", "cat",
"rat", "owl"

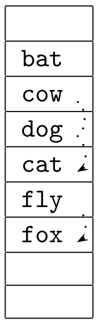
are added to the previous hash table, and if it uses chaining then the hash table adds the first four elements, creating a new linked list for each. When the string "fox" is added, it is added to the head of the linked list that already contains "fly".

Another strategy for resolving collisions uses an *overflow area.* The array is divided into two separate sections, the primary area and an overflow area. If adding an element does not result in a collision, it is just added to the primary area as usual. If however, it results in a collision with an element that already occupies that bucket, then the new element is placed in the overflow area, and a link from the bucket in the primary area is made to it in the overflow area. If there has already been a previous collision at that bucket, then links are adjusted to insert it in the resulting linked list.



This strategy is quite similar to chaining, except that each bucket has space to store an element as well as a link, and the overflow area is pre-allocated and so possibly faster to access. However, it does require dynamic resizing when either the primary area or the overflow area reaches the loading factor, and the hash table needs to keep track of which buckets in the overflow area are available. Using the previous example, the first four elements are added as before. When "fox" is added, its bucket has already been allocated, so it is placed at the start of the overflow area and a link made to it so that it can be later retrieved if needed. Similarly, "cat" is placed in the overflow area and a link made to it from the primary area. Since "owl" hashes to the same index as "fly" it too must be placed in the overflow area, and for simplicity, rather than traversing all the links and placing it at the end, it is inserted before the "fox" link.

The *open addressing* strategy for resolving collisions uses a second hashing operation (rehashing) when there is a collision to find an available bucket in which to put the element.



One of the simplest rehashing operations is to use the original hash function plus (or minus) one, checking whether a neighbouring bucket at $i + 1$ is available. If it is not available then the next bucket at $i + 2$ is tried, repeating until there is an available bucket. This is called *linear probing* and can be performed very quickly, although it tends to create *clusters*, contiguous blocks of filled buckets, which tend to enlarge and cause further collisions, thus affecting the performance of the table. Note how linear probing quickly forms a cluster when the strings are added, "fox" must be put in the bucket after "fly", and several probes are required before "cat" finds an available bucket.

To reduce the formation of clusters an alternative open address strategy is to use *quadratic probling*, which first checks the neighbour $i + 1$, then $i - 1$, $i + 4$, $i - 4$, $i + 9$ etcetera until an available bucket is found. The problem of clusters can be reduced still further by computing a second hashing function and adding its hash code j as an offset to the first when there is a collision, trying the index $i + j$, then $i + 2j$, $i + 3j$, and so on. While open addressing can provide an efficient solution to collisions if clustering can be avoided it does require care when elements are removed from the table. If an element is removed, then once the bucket is free it may not be apparent that certain other elements are still in the hash table. Considering for instance the previous linear probing example, if the string “dog” were actually removed, then a search for “cat” would first check the bucket at index 2 and find it occupied by “cow”. Using its linear probing it would find the bucket at index 3 available and so erroneously presume that “cat” were not in the table. This complication can be handled by using a boolean flag in the bucket that marks an element for removal, so that the bucket can be used by new additions, but that is not considered removed for the purpose of a search.

The Collections API contains classes `HashSet` and `LinkedHashSet` which provide general-purpose hash table implementations of collections. Both classes use a generic type for the elements they store, and since the elements are stored in a hash table, they should have a suitable `hashCode` method. The `HashSet` class implements the `Set` interface and uses a hash table with chaining to resolve collisions. By default it has an initial capacity of 16 elements with dynamic resizing, and a load factor of 75%. This class offers constant time performance for the basic operations `add`, `remove`, `contains` and `size`, assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the number of elements and the capacity of the table. Since the elements are hashed, the iterator returns the elements in no particular order, and the order might change over time if elements are added or the table gets resized. The subclass `LinkedHashSet` differs from its superclass `HashSet` in that in addition to the table with chained lists it also maintains doubly-linked list links running through all of the elements, in the order in which the elements were added to the table. Due to the added expense of maintaining the linked list the performance of a `LinkedHashSet` is likely to be just slightly below that of a `HashSet` for the basic operations. However, iterators in a `LinkedHashSet` can efficiently use the doubly-linked list and so only require time proportional to the size of the set (independent on the capacity of the table), and consistently give the elements in the order in which they were added to the table.

Exercise 5.1 (Hash Table with Chaining) *Prepare a class called* `Student` *that represents information about a student. Its* `hashCode` *method should uses the letters of the student’s name added together. Then implement a hash table for storing some* `Student` *objects that uses chaining to resolve collisions and a load factor of* 75%*.*

5.2 Maps

Reading: pp531-539

A *map* is a function from one collection, called the *keys*, to another collection, called the *values*. A map does not allow duplicate keys (the domain must be a set), and each key must map to only one value (the map must be a function), although two keys can map to the same value (the map might not be one-to-one).

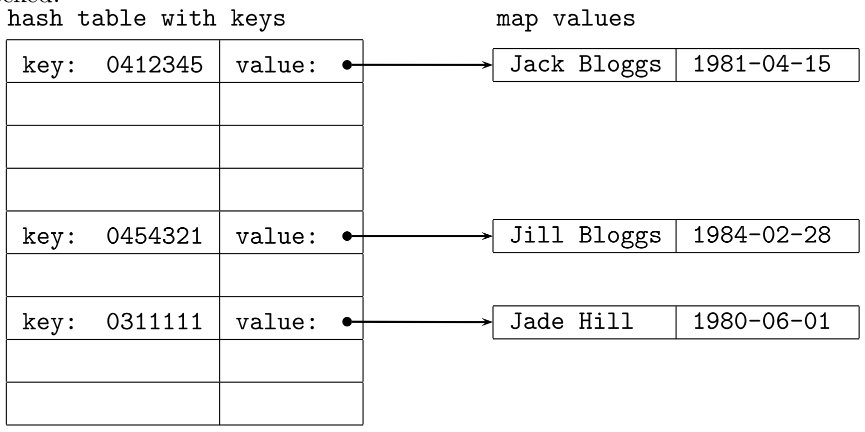
For example, the keys might consist of student id numbers, and the values might consist of student information such as the name and birthdate of a student. Each id number key should map to some student information, although (theoretically) two id numbers might map to identical information (students with identical names and birthdates). Conceptually, an (indexed) list can be considered a special case of a map, where the keys are the integer indices, the values are the elements stored in the list, and the map assigns each index to the element at that index.

The Collections API has an interface called `Map` which describes the abstract data type for a map (it does not extend the `Collection` interface, since a map is conceptually considered a function between collections rather than a collection itself). The `Map` interface takes two type parameters, `K` for the type of keys and `V` for the type of values.

```
≪interface≫
Map<K,V>
------------------------------------------------------------

------------------------------------------------------------
+clear() : void
+containsKey(key : Object) : boolean
+containsValue(value : Object) : boolean
+entrySet() : Set<Map.Entry<K,V>>
+equals(o : Object) : boolean
+get(key : Object) : V
+hashCode() : int
+isEmpty() : boolean
+keySet() : Set<K>
+put(key : K, value : V) : V
+putAll(t Map<? extends K, ? extends V>) : void
+remove(key : Object) : V
+size() : int
+values() : Collection<V>
```

A key-value pair (or *mapping*) is placed in a map by using the `put` method, and the `remove` method uses a specified key to remove a key-value pair from the map, or throws an `UnsupportedOperationException` exception if removals are not permitted. Likewise the `get` method is used to obtain the value with the specified key. The `size` method returns the number of key-value pairs in the mapping, and `isEmpty` returns `true` if there are no pairs. The usual way of checking whether a value is part of a key-value pair (that is, whether it is in the image of the map) is to use the `containsKey` method, specifying a key for the value, although the `containsValue` method can be used to search for a particular value if its key is unknown. There are three view of a map, `keySet` returns all the keys used in the map as a `Set` collection, `values` returns all the values as a `Collection` (it might not be a set since there could be duplicate values), and `entrySet` returns a set consisting of all the key-value pairs (wrapped as `Map.Entry` objects).

Most general-purpose implementations of the `Map` interface use a hash table as their data structure. The keys are stored in the hash table (so the keys need to be equipped with appropriate hashing functions) along with references to the values that form each key-value pair for the map (the values themselves are not stored in the hash table). Because of the hash table data structure, key-based methods such as `containsKey` would run in $O(1)$, whereas methods such as `containsValue` would require $O(m)$, where m is the capacity of the table, since the entire hash table would need to be searched for keys and each key-value pair checked.



The Collection classes `HashMap` and `LinkedHashMap` are general-purpose map implementations analogous to their hash set equivalents, with `LinkedHashMap` using a doubly-linked list of the keys to keep track of the order in which key-value pairs are put in the map. They provide constant-time performance for the basic operations `get` and `put`, assuming the hash function disperses the elements properly among the buckets. The `LinkedHashMap` class uses a more efficient iterator which returns the key-value pairs in the order they were added to the map.

The `IdentityHashMap` class is a special-use implementation of maps that is similar to the `HashMap` class except that it uses reference-equality (`k1==k2`) instead of the more usual object-equality (`k1.equals(k2)`) when comparing keys (and values).

The `WeakHashMap` class provides another implementation of maps that uses a hash table as its data structure. The difference with a `WeakHashMap` is that its hash table uses weak references to the keys it stores rather than the usual strong references. A *weak reference* is a reference to an object that is not strong enough by itself to keep the object alive, avoiding it from being collected by the garbage collector. The garbage collector periodically removes those objects that are no longer referenced, but it also collects objects that are only referenced by weak references. This allows keys in a `WeakHashMap` to be collected when the user of a hash table no longer has a reference to the key, and so is unable to obtain the key-value pair. If the user of a `HashMap` looses all references to a key in the table, that key, and the key-value pair are not garbage collected, since the hash table itself keeps a reference to each key. However, when this happens in a `WeakHashMap`, the key-value pair is eventually garbage collected.

The `ConcurrentHashMap` class provides an advanced implementation of maps using a hash table that provides support for concurrency. All its operations are thread-safe, unlike the other implementations such as `HashMap` which requires synchronization (such as by the `Collections` method `synchronizedMap`). The retrieval operations of a `ConcurrentHashMap` do not use synchronization locking but do allow several threads to safely retrieve key-value pairs concurrently without having to wait for a synchronization monitor.

Hash table implementations of maps provide efficient $O(1)$ `put`, `get`, and `containsKey` operations, but they do not keep the key-value pairs sorted according to the keys (since the keys are hashed and placed at various indices in the table depending on their hash code, not their `compareTo` method). If a map is required where the key-value pairs are sorted according to the keys, such as described by the `SortedMap` interface (analogous to the `SortedSet` interface),

```
≪interface≫
SortedMap<K,V>
-----------------------------------------------------
+comparator() : Comparator<? super K>
+firstKey() : K
+headMap(toKey : K) : SortedMap<K,V>
+lastKey() : K
+subMap(fromKey : K, toKey : K) : SortedMap<K,V>
+tailMap(fromKey : K) : SortedMap<K,V>
```

then a binary search tree would typically be used to store the keys rather than a hash table. The Collections API has a class called `TreeMap` which provides an implementation of maps, and which uses a red/black tree implementation of the `SortedMap` interface, to guarantee that the map will be in ascending key order. Its `put`, `get`, and `containsKey` operations have $O(\log_2 n)$ performance.

As an example, the class `OccurrenceCounter` uses a GUI for opening, displaying, and editing a text file (or a text file with HTML markup tags) in a `JEditorPane` component. It includes a menu bar prepared by the `OCMenuBar` class, which demonstrates an elegant, rather than the simplest, way of including menu options by representing each option as an `Action` object. When the user selects the menu option to count words a hash map is used, with `String` keys (for the words) and `Integer` values (for the counting), to count the occurrences of the words. Each word is treated as a key in the map to efficiently find and increment the current occurrence count of that word, kept as the value in each key-value pair (each value must be stored as an `Integer` object rather than an `int` since the values in a map must always be objects). Once the counting has completed, the hash map is then transferred to a tree map, which sorts the key-value pairs, placing the keys in a binary search tree. This approach is more efficient than using a tree map from the outset, since the map requires a `get` operation for each occurrence of a word.

Exercise 5.2 (Implementing an Enrollments Program) *Prepare a program which allows a user to enroll students, collecting information such as their name, address and birthdate, and which allocates a unique student id to each student as they are enrolled. The program also have a find feature which if given the student id can locate the information about that student. It should also have a feature for listing all the students in the order in which they enrolled.*

```
/**
   A class which demonstrates how a hash map can be used to count the
   number of occurrences of a word in an editable document
   @author Andrew Ensor
*/
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.Toolkit;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.TreeMap;
import java.util.StringTokenizer;
import javax.swing.JEditorPane;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;

public class OccurrenceCounter extends JPanel
{
   private JEditorPane displayArea; // editable text component

   public OccurrenceCounter()
   {  super();
      displayArea = new JEditorPane();
      displayArea.setPreferredSize(new Dimension(600,450));
      add(new JScrollPane(displayArea));
   }

   // clears the display in displayArea
   public void clear()
   {  displayArea.setText("");
   }
```

cont-

-cont

```
   // open the specified text file and display in displayArea
   public void openFile(File file)
   {  try
      {  BufferedReader br = new BufferedReader(new FileReader(file));
         if (file.getName().toLowerCase().endsWith(".html"))
            displayArea.setContentType("text/html");
         else
            displayArea.setContentType("text/plain");
         displayArea.read(br, null);
      }
      catch (IOException event)
      {  JOptionPane.showMessageDialog(this, event.getMessage(),
            "Error Opening File", JOptionPane.ERROR_MESSAGE);
      }
   }

   // save current text in displayArea to the specified text file
   public void saveFile(File file)
   {  try
      {  PrintWriter pw = new PrintWriter(
            new BufferedWriter(new FileWriter(file)));
         String text = displayArea.getText();
         pw.print(text);
         pw.close();
      }
      catch (IOException event)
      {  JOptionPane.showMessageDialog(this, event.getMessage(),
            "Error Saving File", JOptionPane.ERROR_MESSAGE);
      }
   }

   // display the occurrences of each word in the displayPanel
   public void countWords()
   {  // first count the occurrences of each word using a hash map
      Map<String,Integer> occurrenceMap
         = new HashMap<String,Integer>();
      Integer one = new Integer(1); // frequently used count
      StringTokenizer tokenizer
         = new StringTokenizer(displayArea.getText(), " ;,.\n\t()[]");
      while (tokenizer.hasMoreTokens())
      {  String word = tokenizer.nextToken().trim().toLowerCase();
         Integer frequency = occurrenceMap.get(word);
         if (frequency==null)
            frequency = one;
         else
            frequency = new Integer(frequency.intValue()+1);
         occurrenceMap.put(word, frequency);
      }
```

cont-

-cont

```
      // now transfer the map to a tree map so it gets sorted
      TreeMap<String,Integer> sortedMap
         = new TreeMap<String,Integer>(occurrenceMap);
      // Prepare the results in a panel with two columns
      JPanel resultsPanel = new JPanel(new GridLayout(0,2));
      Iterator<Map.Entry<String,Integer>> iterator
         = sortedMap.entrySet().iterator();
      while (iterator.hasNext())
      {  Map.Entry<String,Integer> pair = iterator.next();
         resultsPanel.add(new JLabel(pair.getKey()));
         resultsPanel.add(new JLabel(""+pair.getValue()));
      }
      // put the results in a frame
      JFrame resultsFrame = new JFrame("Summary");
      resultsFrame.getContentPane().add(new JScrollPane(resultsPanel));
      resultsFrame.setSize(new Dimension(400,300));
      // position the frame in the middle of the screen
      Toolkit tk = Toolkit.getDefaultToolkit();
      Dimension screenDimension = tk.getScreenSize();
      Dimension frameDimension = resultsFrame.getSize();
      resultsFrame.setLocation(
         (screenDimension.width-frameDimension.width)/2,
         (screenDimension.height-frameDimension.height)/2);
      resultsFrame.setVisible(true);
   }

   public static void main(String[] args)
   {  JFrame frame = new JFrame("Occurrence Counter");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      OccurrenceCounter panel = new OccurrenceCounter();
      frame.getContentPane().add(panel);
      frame.setJMenuBar(new OCMenuBar(panel));
      frame.pack();
      // position the frame in the middle of the screen
      Toolkit tk = Toolkit.getDefaultToolkit();
      Dimension screenDimension = tk.getScreenSize();
      Dimension frameDimension = frame.getSize();
      frame.setLocation((screenDimension.width-frameDimension.width)/2,
         (screenDimension.height-frameDimension.height)/2);
      frame.setVisible(true);
   }
}
```

```
/**
   A class that prepares a menu bar for controlling an
   OccurrenceCounter panel
   @see OccurrenceCounter.java
*/
import java.awt.event.ActionEvent;
import java.awt.event.KeyEvent;
import java.io.File;
import javax.swing.AbstractAction;
import javax.swing.Action;
import javax.swing.KeyStroke;
import javax.swing.JFileChooser;
import javax.swing.JMenu;
import javax.swing.JMenuBar;

public class OCMenuBar extends JMenuBar
{
   public OCMenuBar(final OccurrenceCounter ocPanel)
   {  super();
      // create the actions that can be performed
      Action newAction = new AbstractAction("New")
         {  public void actionPerformed(ActionEvent e)
            {  ocPanel.clear();
            }
         };
      Action openAction = new AbstractAction("Open")
         {  public void actionPerformed(ActionEvent e)
            {  JFileChooser chooser = new JFileChooser(new File("."));
               int status = chooser.showOpenDialog(ocPanel);
               if (status == JFileChooser.APPROVE_OPTION)
                  ocPanel.openFile(chooser.getSelectedFile());
            }
         };
      Action saveAction = new AbstractAction("Save")
         {  public void actionPerformed(ActionEvent e)
            {  JFileChooser chooser = new JFileChooser(new File("."));
               int status = chooser.showSaveDialog(ocPanel);
               if (status == JFileChooser.APPROVE_OPTION)
                  ocPanel.saveFile(chooser.getSelectedFile());
            }
         };
      Action exitAction = new AbstractAction("Exit")
         {  public void actionPerformed(ActionEvent e)
            {  System.exit(0);
            }
         };
```

cont-

-cont

```
      Action countAction = new AbstractAction("Count Words")
         {  public void actionPerformed(ActionEvent e)
            {  ocPanel.countWords();
            }
         };
      // add keyboard mnemonics for each action
      newAction.putValue(Action.MNEMONIC_KEY,
         new Integer(KeyEvent.VK_N));
      openAction.putValue(Action.MNEMONIC_KEY,
         new Integer(KeyEvent.VK_O));
      saveAction.putValue(Action.MNEMONIC_KEY,
         new Integer(KeyEvent.VK_S));
      exitAction.putValue(Action.MNEMONIC_KEY,
         new Integer(KeyEvent.VK_X));
      countAction.putValue(Action.MNEMONIC_KEY,
         new Integer(KeyEvent.VK_C));
      // add accelerators for some actions
      newAction.putValue(Action.ACCELERATOR_KEY,
         KeyStroke.getKeyStroke(KeyEvent.VK_N, ActionEvent.CTRL_MASK));
      openAction.putValue(Action.ACCELERATOR_KEY,
         KeyStroke.getKeyStroke(KeyEvent.VK_O, ActionEvent.CTRL_MASK));
      saveAction.putValue(Action.ACCELERATOR_KEY,
         KeyStroke.getKeyStroke(KeyEvent.VK_S, ActionEvent.CTRL_MASK));
      // create the file menu
      JMenu fileMenu = new JMenu("File");
      fileMenu.setMnemonic(KeyEvent.VK_F);
      fileMenu.add(newAction);
      fileMenu.add(openAction);
      fileMenu.add(saveAction);
      fileMenu.addSeparator();
      fileMenu.add(exitAction);
      // create the view menu
      JMenu viewMenu = new JMenu("View");
      viewMenu.setMnemonic(KeyEvent.VK_V);
      viewMenu.add(countAction);
      // add both menus to this menu bar
      add(fileMenu);
      add(viewMenu);
   }
}
```

Chapter 6

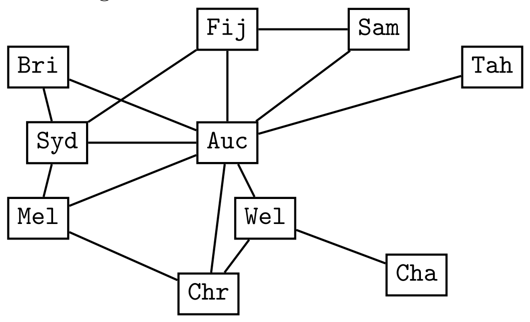
Graphs

6.1 Data Structures for Graphs

Reading: pp546-550,560-563

An (undirected) *graph* is a non-linear collection comprised of a set of *vertices* (or *nodes*) and a set of *edges* which connect pairs of vertices. Usually the vertices store elements and the edges have no direction, but in a *directed graph* each edge is given a direction, and in a *weighted graph* the edges are assigned numerical weights. A tree is actually a connected graph (where it is possible to get from any vertex to any other by following adjacent edges) which contains no closed paths, and a linked list is a connected graph where each vertex is adjacent to at most two other vertices.

Graphs have many applications, such as mappings in geographic information systems, transportation networks, electrical circuits, and computer networks. They also involve many interesting algorithms, such as for searching, constructing spanning trees, and finding minimal paths. For example, an application might involve a graph whose vertices represent airports and edges represent available flight routes:



If some routes were not available in both directions, the graph would be made a directed graph, whereas if the costs, times, or the distances of the flights were of interest the edges could be adorned with the values, making it a weighted graph.

One possible abstract data type for a graph is described by the `GraphADT` interface, which is appropriate for an undirected, non-weighted graph whose vertices hold elements of a generic type `E`. This interface permits two views of a

```
/**
   An interface that defines the abstract data type for an undirected
   graph whose vertices hold elements of type E
*/
import java.util.Set;

public interface GraphADT<E>
{
   // removes all vertices and edges from the graph
   public void clear();
   // returns true if the graph has no vertices nor edges
   public boolean isEmpty();
   // returns a view of the vertices as a Set
   public Set<Vertex<E>> vertexSet();
   // returns a view of the edges as a Set
   public Set<Edge<E>> edgeSet();
   // method which adds the graph as a subgraph into this graph
   public <F extends E> void addGraph(GraphADT<F> graph);
   // adds and returns a new isolated vertex to the graph
   public Vertex<E> addVertex(E element);
   // adds and returns a new undirected edge between two vertices
   public Edge<E> addEdge(Vertex<E> vertex0, Vertex<E> vertex1);
   // removes the specified vertex from the graph
   public <F> boolean removeVertex(Vertex<F> vertex);
   // removes the specified undirected edge from the graph
   public <F> boolean removeEdge(Edge<F> edge);
   // returns whether the specified vertex is in the graph
   public boolean containsVertex(Vertex<?> vertex);
   // returns whether the specified edge is in the graph
   public boolean containsEdge(Edge<?> edge);
}
```

graph, as a set of vertices using `vertexSet`, and as a set of edges using `edgeSet` (if one of the views can be edited it should also alter the graph itself). Since both these views are sets, they can be used for iterating through the vertices and edges of the graph. The `addGraph` method allows a copy of any graph of some type `F` which extends `E` to be added to the graph. Since a graph consists of vertices and edges there are two further methods for building the graph, `addVertex` for creating a new vertex in the graph that initially has no incident edges, and `addEdge` for adding an edge together with its two end vertices (if not already in graph). Likewise, there are two remove methods, and two methods for searching the graph (either for a particular vertex or for a particular edge).

The interfaces `Vertex` and `Edge` are used by the `GraphADT` interface to describe requirements of its vertices and edges respectively. The `Vertex` interface has an accessor method `getUserObject` and a mutator method `setUserObject` for dealing with the element it holds. It also has the methods `incidentEdges` for obtaining the set of edges that connect with the vertex, `adjacentVertices` which returns the set of all vertices that are adjacent (connected by an edge) with the vertex, and `isAdjacent` for checking whether the vertex is adjacent to some specified vertex. The `Edge` interface only contains two methods, `endVertices` which returns an array of length two holding the end vertices for the edge (which

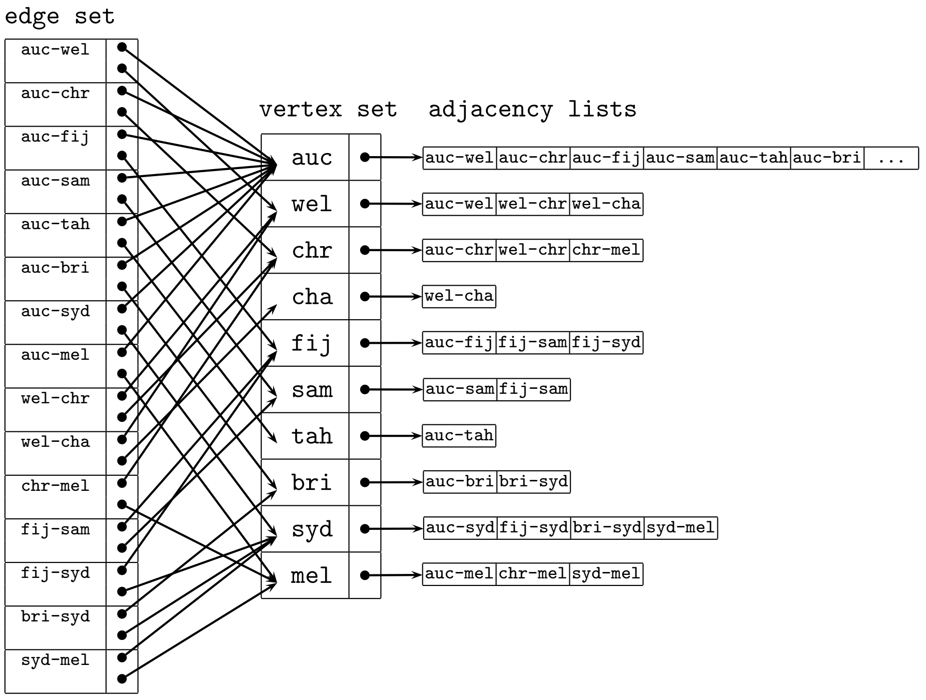
```
/**
   An interface that represents a vertex in a graph which holds
   elements of type E in its vertices
   @see GraphADT.java
*/
import java.util.Set;

public interface Vertex<E>
{
   // returns the element held in the vertex
   public E getUserObject();
   // sets the element held in the vertex
   public void setUserObject(E element);
   // returns the edges connecting with this vertex as a Set
   public Set<Edge<E>> incidentEdges();
   // returns the vertices that are adjacent to this vertex as a Set
   public Set<Vertex<E>> adjacentVertices();
   // returns whether specified vertex is adjacent to this vertex
   public boolean isAdjacent(Vertex<?> vertex);
}
```

```
/**
   An interface that represents an undirected and unweighted edge
   in a graph which holds elements of type E in its vertices
   @see GraphADT.java
*/
import java.util.Set;

public interface Edge<E>
{
   // returns the two end vertices (poss same) for this edge as array
   public Vertex<E>[] endVertices();
   // returns the end vertex opposite the specified vertex
   public Vertex<E> oppositeVertex(Vertex<E> vertex);
}
```

is more convenient than returning them in a set), and `oppositeVertex` which, for convenience, gives the other end vertex of the edge than the specified vertex.

One of the simplest, although not most efficient implementations of graphs is an *edge list* data structure. In this implementation the edges and vertices are each stored in sets and each edge has links to its end vertices. However, if the graph has m vertices and n edges, finding the edges that are adjacent to a specific vertex has $O(n)$ complexity, so traversing the graph along a path is quite inefficient.

To improve the efficiency for traversals, besides using an edge set and a vertex set, an *adjacency list* is maintained for each vertex. An adjacency list (actually often implemented as a set rather than a list) holds references to all the edges that are incident with that vertex. Although this information is redundant (it can be obtained directly from the edge set), it improves the performance of a traversal to an adjacent vertex, which can be important for many graph algorithms.



In some situations, such as when the edges themselves are not of interest, the edge set can be avoided and the adjacency lists can just hold references to the adjacent vertices.

The example `AdjacencyListGraph` class provides an adjacency list implementation of the `GraphADT` interface. It stores the edges and vertices in two sets as the graph is being built, and also maintains an adjacency list for each vertex using a hash map for efficiency. The keys of the hash map are the vertices and the values are the corresponding adjacency lists (each implemented as a hash set rather than a list since the order of the incident edges is unimportant). The two view methods, `vertexSet` and `edgeSet`, both return unmodifiable collections. This prevents a user from attempting to manipulate the graph via `Set` methods, instead the `addVertex`, `addEdge`, `removeVertex`, `removeEdge` must be used (an alternative approach would be to have the `Set` methods call these `AdjacencyListGraph` methods). The sample class `GraphTest` demonstrates how a graph can be built and manipulated.

It is worthwhile noting that wherever possible the `AdjacencyListGraph` class has used interface types such as `Vertex`, `Edge`, `GraphADT`, `Set`, and `Map`, rather than actual class types such as `AdjacencyListVertex`, `AdjacencyListEdge`, `AdjacencyListGraph`, `HashSet`, and `HashMap`. The class types are required for creating objects, but once created the code is much cleaner, simpler, and easier to modify if objects are referenced by the interfaces they implement (this is considered good programming practice in object-oriented programming).

Often restrictions are placed on a graph, such as that there can only be at most one edge between any pair of vertices. Under this restriction, the adjacency list implementation can be improved by replacing the edge set and the collection of adjacency lists by a single matrix, called the *adjacency matrix* of the graph.

```
/**
   A class that implements an undirected graph using an adjacency
   list as the underlying data structure, and whose vertices
   hold elements of type E (which should have a suitable hash function)
   @author Andrew Ensor
*/
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class AdjacencyListGraph<E> implements GraphADT<E>
{
   private Set<Vertex<E>> vertices;
   private Set<Edge<E>> edges;
   // map holding adjacency list (as a set) for each vertex
   private Map<Vertex<E>,Set<Edge<E>>> adjacencyLists;

   public AdjacencyListGraph()
   {  this.vertices = new HashSet<Vertex<E>>();
      this.edges = new HashSet<Edge<E>>();
      this.adjacencyLists = new HashMap<Vertex<E>,Set<Edge<E>>>();
   }

   public <F extends E> AdjacencyListGraph(GraphADT<F> graph)
   {  this();
      addGraph(graph);
   }

   // removes all vertices and edges from the graph
   public void clear()
   {  vertices.clear();
      edges.clear();
      adjacencyLists.clear();
   }

   // returns true if the graph has no vertices nor edges
   public boolean isEmpty()
   {  return vertices.isEmpty();
   }

   // returns a view of the vertices as a Set
   public Set<Vertex<E>> vertexSet()
   {  return Collections.unmodifiableSet(vertices);
   }
```

cont-

-cont

```
   // returns a view of the edges as a Set
   public Set<Edge<E>> edgeSet()
   {  return Collections.unmodifiableSet(edges);
   }

   // generic method which adds a copy of the graph (that has elements
   // of type F which extends type E) into this graph
   public <F extends E> void addGraph(GraphADT<F> graph)
   {  // add all the vertices to this graph and keep track of the
      // correspondence between old vertices in specified graph
      // and new vertices added to this graph
      HashMap<Vertex<F>,Vertex<E>> correspondence
         = new HashMap<Vertex<F>,Vertex<E>>();
      Set<Vertex<F>> oldVertices = graph.vertexSet();
      for (Vertex<F> oldVertex : oldVertices)
      {  Vertex<E> newVertex = addVertex(oldVertex.getUserObject());
         correspondence.put(oldVertex, newVertex);
      }
      // add all the edges to this graph
      Set<Edge<F>> oldEdges = graph.edgeSet();
      for (Edge<F> oldEdge : oldEdges)
      {  // obtain the one or two end vertices for the old edge
         Vertex<F>[] oldEndVertices = oldEdge.endVertices();
         // add an appropriate new edge in this graph
         addEdge(correspondence.get(oldEndVertices[0]),
            correspondence.get(oldEndVertices[1]));
      }
   }

   // adds and returns a new isolated vertex to the graph
   public Vertex<E> addVertex(E element)
   {  AdjacencyListVertex vertex = new AdjacencyListVertex(element);
      addVertex(vertex);
      return vertex;
   }

   // helper method to add a Vertex object to the graph
   private void addVertex(Vertex<E> vertex)
   {  vertices.add(vertex);
      adjacencyLists.put(vertex, new HashSet<Edge<E>>());
   }
```

cont-

-cont

```
// adds and returns a new undirected edge between two vertices
// Note: if the end vertices are not already in the graph
// then copies of them are added as well
public Edge<E> addEdge(Vertex<E> vertex0, Vertex<E> vertex1)
{  // first add the end vertices if not already in graph
   if (!containsVertex(vertex0))
      addVertex(vertex0);
   if (!containsVertex(vertex1))
      addVertex(vertex1);
   // create the new edge
   Edge<E> edge = new AdjacencyListEdge(vertex0, vertex1);
   edges.add(edge);
   // update the adjacency list for each end vertex
   adjacencyLists.get(vertex0).add(edge);
   adjacencyLists.get(vertex1).add(edge);
   return edge;
}

// removes the specified vertex from the graph
public <F> boolean removeVertex(Vertex<F> vertex)
{  if (!containsVertex(vertex))
      return false;
   else
   {  // first remove all incident edges
      Set<Edge<F>> edgesToRemove = vertex.incidentEdges();
      Iterator<Edge<F>> edgeIterator = edgesToRemove.iterator();
      while (edgeIterator.hasNext())
      {  Edge<F> edgeToRemove = edgeIterator.next();
         edgeIterator.remove(); // uses iterator to remove edge
         // remove edge from adjacency lists of both end vertices
         Vertex<F>[] endVertices = edgeToRemove.endVertices();
         adjacencyLists.get(endVertices[0]).remove(edgeToRemove);
         adjacencyLists.get(endVertices[1]).remove(edgeToRemove);
      }
      // remove the vertex
      vertices.remove(vertex);
      return true;
   }
}
```

cont-

-cont

```
// removes the specified undirected edge from the graph
public <F> boolean removeEdge(Edge<F> edge)
{  if (!containsEdge(edge))
      return false;
   else
   {  edges.remove(edge);
      // remove edge from adjacency lists of both its end vertices
      Vertex<F>[] endVertices = edge.endVertices();
      adjacencyLists.get(endVertices[0]).remove(edge);
      adjacencyLists.get(endVertices[1]).remove(edge);
      return true;
   }
}

// returns whether the specified vertex is in the graph
public boolean containsVertex(Vertex<?> vertex)
{  return vertices.contains(vertex);
}

// returns whether the specified edge is in the graph
public boolean containsEdge(Edge<?> edge)
{  return edges.contains(edge);
}

public String toString()
{  String output = "Graph:\n";
   for (Vertex<E> vertex : vertices)
      output += "" + vertex + " has edges:"
         + adjacencyLists.get(vertex) + "\n";
   return output;
}

// inner class that implements a vertex for the AdjacencyListGraph
private class AdjacencyListVertex implements Vertex<E>
{
   private E element;

   public AdjacencyListVertex(E element)
   {  this.element = element;
   }

   // returns the element held in the vertex
   public E getUserObject()
   {  return element;
   }
```

cont-

-cont

```
      // sets the element held in the vertex
      public void setUserObject(E element)
      {  this.element = element;
      }

      // returns the edges connecting with this vertex as a Set
      public Set<Edge<E>> incidentEdges()
      {  return adjacencyLists.get(this);
      }

      // returns vertices that are adjacent to this vertex as a Set
      public Set<Vertex<E>> adjacentVertices()
      {  Set<Edge<E>> adjacencyList = adjacencyLists.get(this);
         Set<Vertex<E>> vertices = new HashSet<Vertex<E>>();
         for (Edge<E> edge : adjacencyList)
            vertices.add(edge.oppositeVertex(this));
         return vertices;
      }

      // returns whether specified vertex is adjacent to this vertex
      public boolean isAdjacent(Vertex<?> vertex)
      {  return adjacentVertices().contains(vertex);
      }

      // overridden method which returns a hash code for this vertex
      public int hashCode()
      {  if (element==null)
            return 0;
         else
            return element.hashCode();
      }

      public String toString()
      {  return "" + element;
      }
   }
```

cont-

-cont

```
   // inner class that implements a vertex for the AdjacencyListGraph
   private class AdjacencyListEdge implements Edge<E>
   {
      private Vertex<E> vertex1, vertex2;

      public AdjacencyListEdge(Vertex<E> vertex1, Vertex<E> vertex2)
      {  this.vertex1 = vertex1;
         this.vertex2 = vertex2;
      }

      // returns the two end vertices for this edge as an array
      public Vertex<E>[] endVertices()
      {  Vertex<E>[] vertices=(Vertex<E>[])(new Vertex[2]);//unchecked
         vertices[0] = vertex1;
         vertices[1] = vertex2;
         return vertices;
      }

      // returns the end vertex opposite the specified vertex
      public Vertex<E> oppositeVertex(Vertex<E> vertex)
      {  if (vertex1.equals(vertex))
            return vertex2;
         else
            return vertex1;
      }

      public String toString()
      {  return "(" + vertex1 + "-" + vertex2 + ")";
      }
   }
}
```

```
/**
   A class which demonstrates how a GraphADT graph is built
*/
public class GraphTest
{
   public static void main(String[] args)
   {  GraphADT<String> graph = new AdjacencyListGraph<String>();
      Vertex<String> auckland = graph.addVertex("Auc");
      Vertex<String> wellington = graph.addVertex("Wel");
      Vertex<String> christchurch = graph.addVertex("Chr");
      Vertex<String> chatham = graph.addVertex("Cha");
      Vertex<String> fiji = graph.addVertex("Fij");
      Vertex<String> samoa = graph.addVertex("Sam");
      Vertex<String> tahiti = graph.addVertex("Tah");
      Vertex<String> brisbane = graph.addVertex("Bri");
      Vertex<String> sydney = graph.addVertex("Syd");
      Vertex<String> melbourne = graph.addVertex("Mel");
      graph.addEdge(auckland, wellington);
      graph.addEdge(auckland, christchurch);
      graph.addEdge(auckland, fiji);
      graph.addEdge(auckland, samoa);
      graph.addEdge(auckland, tahiti);
      graph.addEdge(auckland, brisbane);
      graph.addEdge(auckland, sydney);
      graph.addEdge(auckland, melbourne);
      graph.addEdge(wellington, christchurch);
      graph.addEdge(wellington, chatham);
      graph.addEdge(christchurch, melbourne);
      Edge<String> fijiSamoaEdge = graph.addEdge(fiji, samoa);
      graph.addEdge(fiji, sydney);
      graph.addEdge(brisbane, sydney);
      graph.addEdge(sydney, melbourne);
      System.out.println(graph);
      System.out.println("Graph contains " + fijiSamoaEdge + ":"
         + graph.containsEdge(fijiSamoaEdge));
      System.out.println("Removing edge " + fijiSamoaEdge);
      graph.removeEdge(fijiSamoaEdge);
      System.out.println(graph);
      System.out.println("Graph contains " + fijiSamoaEdge + ":"
         + graph.containsEdge(fijiSamoaEdge));
      GraphADT<Object> graph2 = new AdjacencyListGraph<Object>(graph);
      Vertex<Object> vertex = graph2.vertexSet().iterator().next();
      System.out.println("Removing vertex " + vertex);
      graph2.removeVertex(vertex);
      System.out.println(graph2);
      System.out.println("Graph contains " + vertex + ":"
         + graph2.containsVertex(vertex));
   }
}
```

First, the m vertices of the graph are stored in a list so that their indices in the list can be used to refer to rows and columns of a square $m \times m$ matrix. The i, j-entry of this matrix is the edge connecting the vertex at index i with the vertex at index j, or is `null` if there is no such edge. This allows adjacent vertices to be found in constant $O(1)$ time, which is useful when vertices might have many incident edges (the adjacency list implementation requires a search through the edges incident to a vertex). The drawback however is that the matrix requires more storage space than do adjacency lists.

`vertex list`

0	auc
1	wel
2	chr
3	cha
4	fij
5	sam
6	tah
7	bri
8	syd
9	mel

`adjacency matrix`

index	0	1	2	3	4	5	6	7	8	9
0		auc-wel	auc-chr		auc-fij	auc-sam	auc-tah	auc-bri	auc-syd	auc-mel
1	auc-wel		wel-chr	wel-cha						
2	auc-chr	wel-chr								chr-mel
3		wel-cha								
4	auc-fij					fij-sam			fij-syd	
5	auc-sam				fij-sam					
6	auc-tah									
7	auc-bri								bri-syd	
8	auc-syd				fij-syd			bri-syd		syd-mel
9	auc-mel		chr-mel						syd-mel	

For undirected graphs the adjacency matrix is symmetric, so only about half the entries are necessary, and if the graph does not allow loops (edges between a vertex and itself) then the diagonal entries are not required either. If the edges themselves are not of interest then the entries of the matrix are commonly made `boolean` values to represent whether or not two vertices are adjacent.

Exercise 6.1 (Adjacency Matrix Implementation) *Design an implementation of the* `GraphADT` *interface using an adjacency matrix data structure for a graph with at most one edge between vertices (the* `addVertex` *and* `removeVertex` *methods require careful consideration).*

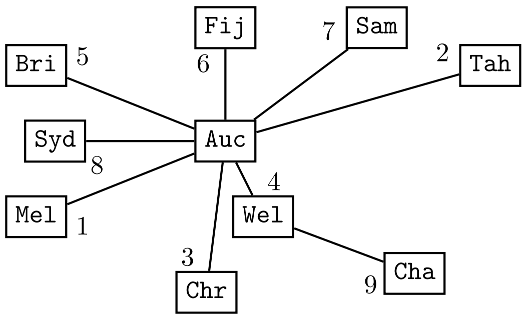
6.2 Graph Algorithms

Reading: pp551-560,570-584

Finding a solution to some problem, such as a way out of a maze or choosing the best possible move in a game of chess, often requires searching a graph of possible situations. There are two alternative strategies for searching a graph, breadth-first searching and depth-first searching. Both strategies use the idea of *paths*, where a path is a sequence of adjacent edges (with no edge repeated), and require that the graph be *connected*, meaning that for every pair of vertices there is a path between them.

A breadth-first search of a connected graph starts from a chosen vertex and visits each of its adjacent vertices. Then the adjacent vertices to each of these is visited in turn, repeating until there are no further adjacent vertices to visit. This process is conveniently handled by using a queue of those vertices whose neighbouring vertices still have to be visited, dequeuing them once their neighbours have all been visited. Since a queue is used, the first vertex placed in the queue is the first to have all its neighbours visited. The example `GraphAlgorithms` class contains a method `breadthFirstSearch` which performed a breadth-first search of any graph that implements the `GraphADT` interface.

To illustrate this, suppose a breadth-first search is to be performed on the graph illustrated on page ??. Choosing to start at the `Auc` vertex, each of its neighbours is visited in turn and placed on the queue.



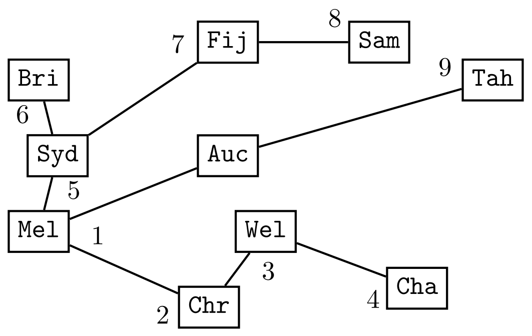
Queue:

```
                                                Auc
Tah    Sam    Fij    Bri    Wel    Chr    Tah    Mel    Auc
                            Tah    Sam    Fij    Bri    Wel
                     Cha    Tah    Sam    Fij    Bri    Wel
                                                        ---
```

As the graph used to demonstrate the algorithm has been created from the `AdjacencyListGraph` class, which uses a hash set to store the incident edges, the adjacent vertices are visited in a seemingly random order, with the vertex `Mel` first (the `Auc-Mel` edge must have a smaller hash code than the other edges in the adjacency list for `Auc`). Once the eight vertices adjacent to `Auc` have been visited and enqueued in the queue, the vertex `Mel` at the front of the queue is dequeued and checked for adjacent vertices. Since all three of its adjacent vertices have already been visited, it is dequeued and the process continues. When the `Wel` vertex is dequeued, its still-unvisited adjacent vertex `Cha` is enqueued, following which all the vertices are dequeued one by one since they don't have any unvisited adjacent vertices.

A depth-first search of a connected graph also starts from a chosen vertex and visits each vertex in the graph, but usually in a different order to a breadth-first search. From the start vertex, one adjacent vertex is chosen and it is visited. But then instead of returning to the start vertex and choosing another vertex adjacent to it, the vertex most-recently visited is used first (rather than last) to find further unvisited vertices. Because of this last-in, first-out policy, a stack can be conveniently used to keep track of the vertices which potentially still have neighbours to visit. A natural alternative to using a stack is to implement a depth-first search using recursion.

For example, using a depth-first search of the graph illustrated on page ??, again starting at the vertex `Auc`, the `depthFirstSearch` method first visits the adjacent vertex `Mel`.



```
        Cha                     Sam
        Wel          Bri        Fij
        Chr          Syd   Syd  Syd
        Mel    Mel   Mel   Mel  Mel         Tah
   Auc  Auc    Auc   Auc   Auc  Auc   Auc   Auc  ___
```

Stack:

But then it visits a vertex `Chr` adjacent to it and pushes it on the stack, followed by a vertex `Wel` adjacent to `Chr`, before visiting the vertex `Cha`. Since the `Cha` vertex has no unvisited neighbours, it is popped off the stack, leaving `Wel` at the top. Since `Wel` has no unvisited neighbours it too is popped, followed by `Chr`, and leaving `Mel` at the top. Finding another unvisited vertex `Syd` adjacent to `Mel` the processing continues, until the vertices `Bri`, `Fij`, and `Sam` have been visited. Since these vertices have no unvisited adjacent vertices they are popped one at a time until only `Auc` remains on the stack. As `Auc` has an unvisited adjacent vertex `Tah` this final vertex is visited and pushed on the stack. Following this, the remaining two vertices on the stack are popped, leaving the stack empty.

Many applications of graphs require finding paths between two vertices. The example `Path` interface represents a path in a graph based on the `Vertex` and `Edge` interfaces. The `LinkedPath` class provides an implementation of this interface using a linked list data structure to store the edges that comprise the path (a linked list was chosen rather than an array list due to the nature of the path operations). Note that this implementation is consistent with the most-common definition of path, where a path is allowed to visit the same vertex several times but can only use each edge at most once.

The method `allPaths` uses recursion to provide a remarkably simple algorithm for finding all possible paths between a specified vertex and all the other vertices in a graph. This method conveniently returns a `Map` object whose keys are the vertices in the graph, and whose values consist of sets of all possible paths from the specified start vertex to the key vertex. Furthermore, if there is no mapping for some key then that key vertex must not be connected to the start vertex. The method works by starting with a single path of length zero at the start vertex. It then calls the recursive method `extendPath` with this path, which first adds the path to the map, and then tries to extend the path using each edge in turn that is incident to the end of the path. If the path can be extended using one of these edges then the extended path is recursively checked to see if it can be extended further. Unless there is an infinite number of edges in the graph this recursive process must eventually terminate.

The *mark-sweep* algorithm is one of the most common algorithms used by garbage collectors to reclaim space in the memory heap that is occupied by objects eligible for collection. The mark-sweep algorithm associates a special mark bit with each object, identifying whether or not the object is alive. When garbage collection is performed all other threads are suspended and all the mark bits are cleared. Then the garbage collector searches the stacks of running

```
/**
   A class which contains some common graph algorithms
   for any graph that implements the GraphADT interface
   @author Andrew Ensor
*/
import java.util.HashSet;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class GraphAlgorithms
{
   // performs a breadth-first search of the specified graph starting
   // at the given vertex
   public static <E> List<Vertex<E>> breadthFirstSearch(
      GraphADT<E> graph, Vertex<E> startVertex)
   {  if (!graph.containsVertex(startVertex))
         throw new IllegalArgumentException();
      // create list to hold vertices as they are encountered
      List<Vertex<E>> visitedVertices = new LinkedList<Vertex<E>>();
      //create queue to keep track of vertices not yet fully processed
      QueueADT<Vertex<E>> processingQueue
         = new LinkedQueue<Vertex<E>>();
      // handle the starting vertex
      visitedVertices.add(startVertex);
      processingQueue.enqueue(startVertex);
      // repeatedly find adjacent vertices and visit them
      while (!processingQueue.isEmpty())
      {  //remove front element
         Vertex<E> frontVertex = processingQueue.dequeue();
         // find all the adjacent vertices that have not been visited
         // and enqueue them
         Iterator<Vertex<E>> iterator
            = frontVertex.adjacentVertices().iterator();
         while (iterator.hasNext())
         {  Vertex<E> adjacentVertex = iterator.next();
            if (!visitedVertices.contains(adjacentVertex))
            {  visitedVertices.add(adjacentVertex);
               processingQueue.enqueue(adjacentVertex);
            }
         }
      }
      return visitedVertices;
   }
```

cont-

-cont

```
// performs a depth-first search of the specified graph starting
// at the given vertex
public static <E> List<Vertex<E>> depthFirstSearch
   (GraphADT<E> graph, Vertex<E> startVertex)
{  if (!graph.containsVertex(startVertex))
      throw new IllegalArgumentException();
   // create list to hold vertices as they are encountered
   List<Vertex<E>> visitedVertices = new LinkedList<Vertex<E>>();
   //create stack to keep track of vertices not yet fully processed
   StackADT<Vertex<E>> processingStack
      = new LinkedStack<Vertex<E>>();
   // handle the starting vertex
   visitedVertices.add(startVertex);
   processingStack.push(startVertex);
   // repeatedly find adjacent vertices and visit them
   while (!processingStack.isEmpty())
   {  Vertex<E> topVertex = processingStack.peek();
      // find an adjacent vertex that has not been visited and push
      // it on stack or else take this top element off stack
      Iterator<Vertex<E>> iterator
         = topVertex.adjacentVertices().iterator();
      boolean found = false;
      while (!found && iterator.hasNext())
      {  Vertex<E> adjacentVertex = iterator.next();
         if (!visitedVertices.contains(adjacentVertex))
         {  visitedVertices.add(adjacentVertex);
            processingStack.push(adjacentVertex);
            found = true;
         }
      }
      if (!found) // pop top vertex as all its adj already visited
         processingStack.pop();
   }
   return visitedVertices;
}
```

cont-

-cont

```
// obtains all possible paths in graph that start at startVertex
// returns a map which for any vertex in the graph gives the set of
// paths that started at startVertex and ended at that vertex
public static <E> Map<Vertex<E>,Set<Path<E>>> allPaths
   (Vertex<E> startVertex)
{  // create a map from vertices in the graph to all the paths
   // found so far that start at startVertex and end at that vertex
   Map<Vertex<E>,Set<Path<E>>> paths
      = new HashMap<Vertex<E>,Set<Path<E>>>();
   // initially there is just one path startVertex (length 0)
   Path<E> initialPath = new LinkedPath<E>(startVertex);
   // start extending initial path adding it to paths as extended
   extendPath(paths, initialPath);
   return paths;
}

// recursive helper method which first adds a path to the map paths
// and then tries to extend a copy of the path by one further edge
// This method presumes that the paths are all distinct
private static <E> void extendPath
   (Map<Vertex<E>,Set<Path<E>>> paths, Path<E> path)
{  Vertex<E> pathEnd = path.getEndVertex();
   // add path to the map paths using path end as the map key
   Set<Path<E>> pathsEndingHere = paths.get(pathEnd);
   if (pathsEndingHere==null) // this is the first path ending here
   {  // create a new set of paths for the vertex
      pathsEndingHere = new HashSet<Path<E>>();
      paths.put(pathEnd, pathsEndingHere);
   }
   pathsEndingHere.add(path);
   // get all the edges that are incident with the end of path
   Set<Edge<E>> incidentEdges = pathEnd.incidentEdges();
   // try to extend path along each of the incident edges
   for (Edge<E> incidentEdge : incidentEdges)
   {  if (!path.containsEdge(incidentEdge))
      {  // extend path along the incident edge
         Path<E> extendedPath = getCopy(path);
         extendedPath.appendEnd(incidentEdge);
         extendPath(paths, extendedPath);
      }
   }
}
```

cont-

-cont

```
   // helper method that makes a copy of an existing path
   private static <E> Path<E> getCopy(Path<E> existingPath)
   {  Path<E> newPath=new LinkedPath<E>(existingPath.getStartVertex());
      Iterator<Edge<E>> iterator = existingPath.iterator();
      while (iterator.hasNext())
         newPath.appendEnd(iterator.next());
      return newPath;
   }

   public static void main(String[] args)
   {  GraphADT<String> graph = new AdjacencyListGraph<String>();
      Vertex<String> auckland = graph.addVertex("Auc");
      Vertex<String> wellington = graph.addVertex("Wel");
      Vertex<String> christchurch = graph.addVertex("Chr");
      Vertex<String> chatham = graph.addVertex("Cha");
      Vertex<String> fiji = graph.addVertex("Fij");
      Vertex<String> samoa = graph.addVertex("Sam");
      Vertex<String> tahiti = graph.addVertex("Tah");
      Vertex<String> brisbane = graph.addVertex("Bri");
      Vertex<String> sydney = graph.addVertex("Syd");
      Vertex<String> melbourne = graph.addVertex("Mel");
      graph.addEdge(auckland, wellington);
      graph.addEdge(auckland, christchurch);
      graph.addEdge(auckland, fiji);
      graph.addEdge(auckland, samoa);
      graph.addEdge(auckland, tahiti);
      graph.addEdge(auckland, brisbane);
      graph.addEdge(auckland, sydney);
      graph.addEdge(auckland, melbourne);
      graph.addEdge(wellington, christchurch);
      graph.addEdge(wellington, chatham);
      graph.addEdge(christchurch, melbourne);
      graph.addEdge(fiji, samoa);
      graph.addEdge(fiji, sydney);
      graph.addEdge(brisbane, sydney);
      graph.addEdge(sydney, melbourne);
      System.out.println("Example Graph:\n" + graph);
      System.out.println("Performing Depth-first search from Auc:");
      List<Vertex<String>> list = depthFirstSearch(graph, auckland);
      System.out.println(list);
      System.out.println("Performing Breadth-first search from Auc:");
      list = breadthFirstSearch(graph, auckland);
      System.out.println(list);
      System.out.print("Finding paths from Cha to Sam: ");
      Map<Vertex<String>,Set<Path<String>>> paths = allPaths(chatham);
      System.out.println("there are " + paths.get(samoa).size()
         + " such paths");
   }
}
```

```
/**
   An interface that represents a path in a graph which holds
   elements of type E between two vertices (with distinct edges)
   @see GraphADT.java
*/
import java.util.Iterator;

public interface Path<E>
{
   // returns the vertex at the start of this path
   public Vertex<E> getStartVertex();

   // returns the vertex at the end of this path
   public Vertex<E> getEndVertex();

   // returns an iterator of the edges in order from the start vertex
   public Iterator<Edge<E>> iterator();

   // returns the number of edges in this path
   public int length();

   // appends specified edge to the start of the path if possible
   public boolean appendStart(Edge<E> edge);

   // appends specified edge to the end of the path if possible
   public boolean appendEnd(Edge<E> edge);

   // removes (and returns) the edge at the start of the path or null
   public Edge<E> removeStart();

   // removes (and returns) the edge at the end of the path or null
   public Edge<E> removeEnd();

   // returns true if specified vertex is in the path
   public boolean containsVertex(Vertex<E> vertex);

   // returns true if specified edge is in the path
   public boolean containsEdge(Edge<E> edge);
}
```

```
/**
   A class that implements a path in a graph (which holds
   elements of type E) between two vertices (with distinct edges)
   using a linked list as the data structure
   @author Andrew Ensor
*/
import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class LinkedPath<E> implements Path<E>
{
   private Vertex<E> startVertex, endVertex;
   private List<Edge<E>> edges;

   // creates a new path with length 0 starting at specified vertex
   public LinkedPath(Vertex<E> startVertex)
   {  if (startVertex==null)
         throw new NullPointerException();
      this.startVertex = startVertex;
      this.endVertex = startVertex;
      this.edges = new LinkedList<Edge<E>>();
   }

   // returns the vertex at the start of this path
   public Vertex<E> getStartVertex()
   {  return startVertex;
   }

   // returns the vertex at the end of this path
   public Vertex<E> getEndVertex()
   {  return endVertex;
   }

   // returns an iterator of the edges in order from the start vertex
   public Iterator<Edge<E>> iterator()
   {  return Collections.unmodifiableCollection(edges).iterator();
   }

   // returns the number of edges in this path
   public int length()
   {  return edges.size();
   }
```

cont-

-cont

```
   // appends specified edge to the start of the path if possible
   public boolean appendStart(Edge<E> edge)
   {  Vertex<E>[] endVertices = edge.endVertices();
      if (edges.contains(edge))
         return false; // edge is already used in the path
      else if (startVertex.equals(endVertices[0]))
      {  edges.add(0, edge);
         startVertex = endVertices[1];
      }
      else if (startVertex.equals(endVertices[1]))
      {  edges.add(0, edge);
         startVertex = endVertices[0];
      }
      else
         return false; // edge was not incident to start vertex
      return true;
   }

   // appends specified edge to the end of the path if possible
   public boolean appendEnd(Edge<E> edge)
   {  Vertex<E>[] endVertices = edge.endVertices();
      if (edges.contains(edge))
         return false; // edge is already used in the path
      if (endVertex.equals(endVertices[0]))
      {  edges.add(edge);
         endVertex = endVertices[1];
      }
      else if (endVertex.equals(endVertices[1]))
      {  edges.add(edge);
         endVertex = endVertices[0];
      }
      else
         return false; // edge was not incident to end vertex
      return true;
   }

   // removes (and returns) the edge at the start of the path or null
   public Edge<E> removeStart()
   {  if (edges.size()==0)
         return null;
      else
      {  Edge<E> edge = edges.remove(0); // first edge in list
         endVertex = edge.oppositeVertex(endVertex);
         return edge;
      }
   }
```

cont-

-cont

```
   // removes (and returns) the edge at the end of the path or null
   public Edge<E> removeEnd()
   {  if (edges.size()==0)
         return null;
      else
      {  Edge<E> edge=edges.remove(edges.size()-1);//last edge in list
         startVertex = edge.oppositeVertex(startVertex);
         return edge;
      }
   }

   // returns true if specified vertex is in the path
   public boolean containsVertex(Vertex<E> vertex)
   {  for (Edge<E> edge : edges)
      {  Vertex<E>[] endVertices = edge.endVertices();
         if (vertex.equals(endVertices[0]) ||
            vertex.equals(endVertices[1]))
            return true;
      }
      return false;
   }

   // returns true if specified edge is in the path
   public boolean containsEdge(Edge<E> edge)
   {  return edges.contains(edge);
   }

   // returns a string representation of the path
   public String toString()
   {  String output = "Path from " + startVertex + " to "
         + endVertex + ": ";
      for (Edge<E> edge : edges)
         output += "" + edge;
      return output;
   }

   // returns whether this path has same edges as specified path
   public boolean equals(Path<E> path)
   {  Iterator<Edge<E>> iterator1 = edges.iterator();
      Iterator<Edge<E>> iterator2 = path.iterator();
      while(iterator1.hasNext() && iterator2.hasNext())
      {  if (!iterator1.next().equals(iterator2.next()))
            return false;
      }
      return !(iterator1.hasNext() || iterator2.hasNext());
   }
}
```

threads, marking all their reachable objects as live, and following all references from one accessible object to another, performing a depth-first search of the (directed) graph whose vertices are all the objects and edges are the object references. Once this *mark phase* has completed the memory heap is scanned and any space occupied by objects that have not been marked is reclaimed (referred to as the *sweep phase*). The mark-sweep garbage collection algorithm reclaims unused space in time proportional to the number of live objects and their references, plus the size of the memory heap.

The Web is another example of a large (directed) graph, whose vertices are the hypertext documents and edges consist of hyperlinks between documents. A web *crawler* or *spider* is the data collection portion of a search engine which traverses this graph examining its vertices. Typically a web crawler collates information about each page, such as its title, keywords, headings and frequently used words, and sends it all to a database which the search engine can quickly search when requested.

A web crawler starts by parsing a specified web page, noting any hypertext links on that page that point to other web pages. Then it in turn parses those pages for new links, recursively traversing the Web. A large search engine would typically have web crawlers constantly traversing the Web to find new pages and revisit its known pages, so that when a user asks the engine to search for a keyword its database is reasonably up to date. Some web sites can notify web crawlers about pages they should not use by including a file named `robots.txt`, written using a format according to the *Robots Exclusion Standard*.

The `WebCrawler` class demonstrates how a simple web crawler could be implemented. First it sets the system proxy host properties to enable it to receive information from outside AUT's firewall. Then it creates a graph that will be built by the web crawler as it traverses the Web. Its vertices represent visited web pages, each holding a `URL` object, where a *uniform resource locator* (URL) is a pointer to some resource on the Web, such as a web page.

Starting with a specified web page, it creates a `WebPageSearcher` whose `run` method first checks that the web site doesn't want to discourage crawlers, and then obtains the content of the page using the `URL` method `openStream`, which it layers with an `InputStreamReader` (to convert the byte stream to a character stream), and then a `BufferedReader` (to buffer the stream so entire lines can be read at a time). This content is then searched for hyperlinks, which for HTML documents are `href` attributes inside anchor tags `<A href="...">`. A SAX parser would be ideal for this task, but unfortunately many web pages still do not conform to XML standards.

Next, a loop is performed for each of the hyperlinks, which is placed inside a synchronized block. This block is synchronized on the graph (the monitor) which is shared by each `WebPageSearcher` instance, meaning that only one thread can access the loop at a time. The reason for this is to avoid a race condition of several threads modifying the graph at the same time (alternatively, the methods of the graph itself could have been synchronized). Each iteration of this loop picks one of the hyperlinks and checks whether its URL is already an element in some vertex of the graph, and adds a new edge to the graph. If this URL is new then a new `WebPageSearcher` is created and placed in a separate thread. Having each new page opened and searched in a separate thread enables many downloads to take place concurrently, rather than sequentially, so one slow connection does not affect the retrieval of the others. Finally, once threads have been started for each hyperlink, the `WebPageSearcher` waits until all the threads it started have finished before finishing itself (by calling each thread's `join` method). To avoid having the web crawler search the entire Web (a rather long process), this example places an upper limit on the number of pages that it will search.

Exercise 6.2 (Recursive Graph Algorithms) *The depth-first search algorithm can be simply implemented using recursion rather than an explicit stack. Prepare such a recursive version of the* `depthFirstSearch` *method.*

```java
/**
   A class that represents a web crawler which starts at a specified
   web address and follows hyperlinks to new pages, forming a graph
   @author Andrew Ensor
*/
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Properties;
import java.util.Set;

public class WebCrawler
{
   private static final String DEFAULT_START = "http://www.aut.ac.nz";

   public WebCrawler()
   {  // set the proxy host and port number for AUT's firewall
      // note this is not needed if no firewall present
      Properties props = new Properties(System.getProperties());
      props.put("http.proxySet", "true"); // true if using proxy
      props.put("http.proxyHost", "cache.aut.ac.nz"); // AUT specific
      props.put("http.proxyPort", "3128"); // AUT specific
      System.setProperties(props);
   }

   public GraphADT<URL> search() throws MalformedURLException
   {  return search(DEFAULT_START);
   }

   public GraphADT<URL> search(String startPage)
      throws MalformedURLException
   {  GraphADT<URL> webGraph = new AdjacencyListGraph<URL>();
      WebPageSearcher.setGraph(webGraph);
      Vertex<URL> startVertex =webGraph.addVertex(new URL(startPage));
      WebPageSearcher startPageSearcher
         = new WebPageSearcher(startVertex);
      startPageSearcher.run(); // first searcher is run in this thread
      return webGraph;
   }

   public static void main(String[] args)
   {  WebCrawler crawler = new WebCrawler();
      try
      {  GraphADT<URL> webGraph
            = crawler.search("http://www.nzherald.co.nz/");
      }
      catch (MalformedURLException e) // not valid protocol
      {  System.out.println("Invalid protocol used:" + e.getMessage());
      }
   }
}
```

```
/**
   A class that handles opening a URL, checking that robots are
   allowed, finding hyperlinks to other URLs and spawning a
   new WebPageSearcher in its own thread for each URL found
   @see WebCrawler.java
*/
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;
import java.util.StringTokenizer;

public class WebPageSearcher implements Runnable
{
   private Vertex<URL> vertex;
   private static GraphADT<URL> webGraph;
   private static int numSearchers = 0;//total num of WebPageSearchers
   private static final int MAX_SEARCHERS = 20;
   private static final String DISALLOW_COMMAND = "Disallow:";

   // constructor for a new WebPageCrawler, the vertex is already
   // presumed to be added to webGraph
   public WebPageSearcher(Vertex<URL> vertex)
   {  this.vertex = vertex;
      System.out.println("New searcher created for "
         + vertex.getUserObject());
      numSearchers++;
   }

   // set the graph that is shared by all the WebPageCrawler objects
   public static void setGraph(GraphADT<URL> graph)
   {  webGraph = graph;
   }
```

cont-

-cont

```
   // open the URL, check if robots are allowed, read contents and
   // spawn a new WebPageSearcher for each hyperlink found
   public void run()
   {  // check valid protocol and robot allowed access
      URL url = vertex.getUserObject();
      System.out.println("Searcher starting for " + url);
      if (crawlerAllowedAccess(url))
      {  try
         {  // open the url and read content, then find all hyperlinks
            String pageContent = getContent(url);
            List<String> hyperlinks = getHyperlinks(pageContent);
            // create a set of threads, one thread per new searcher
            Set<Thread> threads = new HashSet<Thread>();
            // the following code is synchronized with webGraph as its
            // monitor, so that only one WebPageSearcher can be adding
            // to the graph at a time
            synchronized(webGraph)
            {  for (String hyperlink : hyperlinks)
               {  try
                  {  URL linkedURL = new URL(url, hyperlink);
                     // search for this url as element of some vertex
                     Iterator<Vertex<URL>>iterator
                        = webGraph.vertexSet().iterator();
                     Vertex<URL> foundVertex = null;
                     while (iterator.hasNext() && foundVertex==null)
                     {  Vertex<URL> nextVertex = iterator.next();
                        if (nextVertex.getUserObject().equals
                           (linkedURL))
                           foundVertex = nextVertex;
                     }
                     if (foundVertex==null)
                     {  // the hyperlinked page not yet visited
                        foundVertex = webGraph.addVertex(linkedURL);
                        if (numSearchers<MAX_SEARCHERS)
                        {  WebPageSearcher newSearcher
                              = new WebPageSearcher(foundVertex);
                           threads.add(new Thread(newSearcher));
                        }
                     }
                     if (!vertex.equals(foundVertex))//don't add loops
                        webGraph.addEdge(vertex, foundVertex);
                  }
                  catch (MalformedURLException e) // not valid protocol
                  {  System.out.println
                        ("Protocol not valid for " + url);
                  }
               }
            }
```

cont-

-cont

```
         // start all the WebPageSearcher threads
         for (Thread searcherThread : threads)
            searcherThread.start();
         //have this thread wait until each of the threads has died
         for (Thread searcherThread : threads)
         {  try
            {  // have this thread wait for searcherThread to die
               searcherThread.join();
            }
            catch (InterruptedException e)
            {} // ignore
         }
      }
      catch (IOException e)
      {  System.out.println("I/O Exception in " + url);
      }
   }
   else
      System.out.println("Crawler disallowed at " + url);
   System.out.println("Searcher terminating for " + url);
}

// checks the protocol and robots.txt file at the url to see
// whether the web site requests robots not to access it
// note this doesn't check meta tags for such requests
private boolean crawlerAllowedAccess(URL url)
{  // check that the url is using http protocol
   if (url.getProtocol().compareTo("http")!=0)
   {  System.out.println("Invalid protocol for " + url);
      return false;
   }
   // read the contents of the robots.txt file (if exists)
   String robotFileContent = "";
   try
   {  URL robotFileURL = new URL("http://" + url.getHost()
         + "/robots.txt");
      robotFileContent = getContent(robotFileURL);
   }
   catch (IOException e)
   {  return true; // file can't be read or doesn't exist
   }
   // search for the "Disallow:" commands
   String webPageName = url.getFile().toLowerCase();
   int disallowIndex = 0;
```

cont-

-cont

```
      do
      {  // find index of next disallow command
         disallowIndex = robotFileContent.indexOf(DISALLOW_COMMAND,
            disallowIndex);
         if (disallowIndex >= 0)
         {  // a disallow command was found in robots.txt file
            // so obtain the page name that is disallowed
            disallowIndex += DISALLOW_COMMAND.length(); // go to name
            StringTokenizer tokenizer = new StringTokenizer
               (robotFileContent.substring
               (disallowIndex).toLowerCase());
            String disallowedPage = tokenizer.nextToken();
            if (webPageName.startsWith(disallowedPage))
               return false;
         }
      }
      while (disallowIndex >= 0);
      return true; // no mention anywhere of disallowing this page
   }

   // returns the content of the specified url
   private String getContent(URL pageURL) throws IOException
   {  String pageContent = "";
      try
      {  // set connection and read timeouts for the connection
         URLConnection connection = pageURL.openConnection();
         connection.setConnectTimeout(5000); // 5000ms
         connection.setReadTimeout(5000); // 5000ms
         // create an input stream and read all the lines
         InputStream is = pageURL.openStream();
         BufferedReader br = new BufferedReader
            (new InputStreamReader(is));
         String line = br.readLine();
         while (line != null)
         {  pageContent += line;
            line = br.readLine();
         }
         br.close();
      }
      catch (MalformedURLException e) // not valid protocol
      {  System.out.println("Protocol not valid for "+pageURL);
      }
      return pageContent;
   }
```

cont-

-cont

```
   // returns a list of all the hyperlinks found in the page content
   // by searching for "<a" followed by "href" and "="
   // then the hyperlink then eventually a ">"
   private List<String> getHyperlinks(String pageContent)
   {  String remainingContent = pageContent.toLowerCase();
      List<String> hyperlinkList = new LinkedList<String>();
      // declare temporary variables
      int startTagIndex, hrefIndex, endTagIndex;
      do
      {  // find index of the next anchor tag
         startTagIndex = remainingContent.indexOf("<a");
         if (startTagIndex >= 0)
         {  hrefIndex=remainingContent.indexOf("href", startTagIndex);
            if (hrefIndex>0)
               hrefIndex = remainingContent.indexOf("=", hrefIndex);
            endTagIndex=remainingContent.indexOf(">", startTagIndex);
            if (hrefIndex>startTagIndex && hrefIndex<endTagIndex)
            {  // valid anchor tag found with href inside tag
               remainingContent=remainingContent.substring(hrefIndex+1);
               StringTokenizer tokenizer = new StringTokenizer
                  (remainingContent, "\t\n\r\"’>#");
               hyperlinkList.add(tokenizer.nextToken());
            }
            else if (endTagIndex<remainingContent.length())
               remainingContent
                  = remainingContent.substring(endTagIndex);
            else // this anchor tag did not end
               remainingContent = "";
         }
         else // there are no further anchor tags
            remainingContent = "";
      }
      while (remainingContent.length()>0);
      return hyperlinkList;
   }
}
```

Chapter 7

Databases

7.1 Relational Databases

Reading: none

A traditional file system, sometimes called a *flat file*, is a one-dimensional storage system, presenting its information from a single point of view. As computing became more widely used in information management each application tended to be implemented as a separate system with its own collection of data in flat files. As a consequence, much information was duplicated across several files, and related information stored separately. In order to reduce this duplication and integrate related information database systems were introduced. A *database* is a collection of data with links between its entries to make the information accessible from a variety of perspectives.

A typical database application involves several layers of software. The application software handles the communication with the database user, determining the system's external characteristics. Application software does not directly manipulate the database but instead passes requests to the database management system. The *database management system* (DBMS) is software that actually manages and stores data in a particular database.

Separating the software into layers allows applications to treat the database at a more abstract level, ignoring the details of how the database is actually implemented, the hashing functions and maps it might use, or whether the database is spread across several machines. This simplifies the design of application software. The DBMS can control access to the database and ensure the integrity and security of the information. Furthermore, separating the user interface from the actual data manipulation gives *data independence* — the ability to change the organization of the database without having to change the application software.

There are several alternative models used by database management systems to represent data to the application software, hiding the internal complexities of the database. The two most popular models are:

relational database model where data are stored in rectangular tables of mathematical relations with links to show the relationships between the tables,

object-oriented database model where objects are stored with links to show the relationships between the objects.

Each table in a relational database holds data that have the same type of information. Each row of the table is called a *record* or *tuple*, and each column is called an *attribute* because it describes one characteristic (attribute) of the record.

The first step in designing a relational database is to determine what tables will be required and suitable attributes for each table. In a typical complex database such decisions have important repercussions and contain many subtleties.

As an example, suppose details about university enrollments are to be stored in a database, including information about students, their details, courses, and grades. A first attempt might try to place all this information in a single table, but this would create several difficulties: there would be redundancy in the table if a single student were enrolled in more than one course their personal details would appear several times; updating the personal details for a student would require searching the entire table and making changes to each tuple that contained information for that student, but not for other students that might by chance have identical personal information. Instead, the information would be better split across several tables:

- a student table holding student information, with attributes for the student id number, given and surnames, gender, birthdate, and telephone number,
- a course table holding course information, with attributes for the course control number, the name of the course, fees, department responsible, and degree level,
- an enrollment table holding enrollment information, with attributes for the student id number, course control number for the course in which that student is enrolled, the semester and year of enrollment, and the student's grade in that course.

The enrollment table requires an attribute to specify when the enrollment occurred, since a student might be enrolled in the same course several times.

Students					
S_Id	*G_Name*	*S_Name*	*Gender*	*B_Date*	*H_Phone*
0412345	Jack	Bloggs	M	1981-04-15	9179990
0454321	Jill	Bloggs	F	1984-02-28	9179990
0311111	Jade	Hill	F	1980-06-01	null
⋮	⋮	⋮	⋮	⋮	⋮

Courses				
CCN	*C_Name*	*Fee*	*Dept_Code*	*Level*
716180	Data Structures and Algorithms	450	SA	6
716181	Algorithm Design and Analysis	450	SA	6
145612	Communications	420	AD	5
⋮	⋮	⋮	⋮	⋮

Enrollments				
S_Id	*CCN*	*Semester*	*Year*	*Grade*
0412345	145612	1	2004	B
0412345	716180	1	2004	A
0412345	716181	2	2005	null
0454321	145612	1	2004	B
0454321	717180	1	2005	null
0311111	145612	1	2004	D
0311111	145612	2	2004	C
0311111	716180	1	2005	null
0311111	716181	2	2005	null
⋮	⋮	⋮	⋮	⋮

Note that the personal details for each student occur only once in the database, so updating the details of a student involves changing just one record. There are proper mathematical tools for determining whether the design of a database is suitable (called *normal forms*), but roughly speaking the database designer tries to avoid combining more than one concept into a single table, and avoid redundancies of information.

A *candidate key* for a table is a collection of attributes that can be used to uniquely identify each record in the table. Usually one candidate key is chosen for each table and called the *primary key*. Knowing the values of a record for the attributes that comprise the primary key should uniquely determine the remaining attributes for the record. For the enrollment database, a suitable primary key for the student table would be the student id number, since each student is uniquely determined by their id number. Likewise, a suitable primary key for the course table would be the course control number, since no two courses have the same control number. The enrollment table is more complicated, the student id number is not suitable (since a student would probably be enrolled in more than one course while at the university), nor is the course control number (since a course would probably have more than one student ever enrolled). Using both the student id number together with the course control number still is insufficient, since a student might repeat the same course. A suitable primary key would be using the student id number, course control number, together with the semester and year of the enrollment. Given this information for a record there could only ever be one grade for the record, presuming that a student cannot get two different grades for the same course in a given semester and year (hence the grade attribute is said to be *functionally dependent* on the attributes that form the primary key). When a record in some table refers to the primary key of another table, that primary key is often called a *foreign key*. Understanding the concepts of tables and primary keys is central to designing an effective database.

It is the responsibility of the DBMS to ensure that the primary key obeys the following constraints:

entity integrity constraint the primary key cannot have null values and no two records in the table can have the same values for the primary key (so the student id cannot be null and no two students are allowed the same student id),

referential integrity constraint a value in a foreign key can only appear if that value already appears in the primary key for some record (so the enrollment table cannot refer to a student id that does not exist in the student table).

Application software communicate with the DBMS using the *Structured Query Language* (SQL), which is a standardized language for interacting with a DBMS managing a relational database. Desktop databases (such as Access or File-Maker Pro) usually have a graphical user interface for users, so that the user can avoid submitting SQL statements, but commercial server-based databases (such as the proprietary Oracle, DB2, or the open source MySQL) are accessed purely through SQL. The DBMS translates SQL statements into sequences of internal operations on the database that optimize data retrieval.

The Java Application Programming Interfaces (APIs) for relational databases are known as JDBC (often referred to as Java Database Connectivity). Since there are so many databases using vendor specific protocols, whereas Java is platform independent, the JDBC does not communicate directly with a DBMS but instead communicates with a piece of driver software written for a specific database, or with a bridge driver to another standard protocol such as the JDBC-ODBC Bridge driver (which in turn would communicate with an ODBC driver written for a specific database).

For a program to communicate with a database once a suitable JDBC driver has been installed the following steps are used:

Load appropriate driver The driver (or several drivers) for the database is loaded using the `forName` method of the `Class` class. For example, to access a MySQL database via the MySQL driver one uses:

```
String driver = "com.mysql.jdbc.Driver";
Class.forName(driver);
```

(in practice the name of the driver probably would not be hard coded like this in the program). This automatically registers the driver with the `DriverManager`, which is the class responsible for selecting database drivers and creating database connections.

Connect to database The `DriverManager` class (from the package `java.sql`) is used to establish a connection to a specified database URL of the form `jdbc:`*`subprotocol`*`:`*`subname`*. For example, to connect to the AUT database server `raptor` on the port `3306` which is allocated to its MySQL DBMS one uses:

```
String url = "jdbc:mysql://raptor:3306/";
String userName = ...;
String password = ...;
Connection con = DriverManager.getConnection(url,
   userName, password);
```

(where the user name and password would typically be obtained from a GUI).

Create a JDBC statement To execute SQL statements, first a `Statement` object must be created for the database connection. The same `Statement` object can be reused for sending further commands to the database:

```
Statement stmt = con.createStatement();
```

If the same SQL statement is to be executed many times, it might more efficient to use a `PreparedStatement` which gets precompiled by the DBMS.

Execute SQL statements For SQL statements such as `INSERT`, `UPDATE`, or `DELETE`, as well as data definition commands such as `CREATE TABLE` or `DROP TABLE`, or other SQL statements that return nothing one uses the `Statement` method `executeUpdate`:

```
String command = ...;
stmt.executeUpdate(command);
```

For SQL statements that return a result set, typically a `SELECT` statement, one instead uses the `Statement` method `executeQuery`:

```
String command = "SELECT ...";
ResultSet rs = stmt.executeQuery(command);
```

For unknown SQL strings or statements that might return multiple result sets (such as a stored procedure) one can use the more general `Statement` method `execute`.

Process the result Some SQL statements, such as `SELECT` queries return a `ResultSet` object holding a table of data, and a cursor pointing to its current row of data, initially positioned before the first row. The `next` method moves the cursor to the next row, and because it returns `false` when there are no more rows it can be used in a `while` loop to iterate through the result set using `get`*`Xxx`* methods of the appropriate type to retrieve the value in each column:

```
while (rs.next())
{  ...= rs.getXxx(...);
}
```

Release resources `Statement` and `Connection` objects should be closed as soon as one is finished with them to avoid tying up database resources (both are automatically closed when garbage collected):

```
stmt.close();
con.close();
```

When a `Statement` object is closed, its current `ResultSet` object, if one exists, is also closed.

As an example, the class `ShowDatabases` demonstrate how one can obtain a list of databases for AUT's `raptor` database server.

Exercise 7.1 (Show Tables GUI) *Prepare a program that obtains the user's name and password for connection to the database named* `universitydb` *on* `raptor` *(using the URL* `jdbc:mysql://raptor:3306/universitydb`*), and then makes a list in a GUI holding the names of the tables in the database (using the SQL statement* `SHOW TABLES`*). Selecting one of the names from the list should display a description of that table (using the SQL statement* `DESCRIBE` *`TableName`*).*

```
/**
   A class that demonstrates how to obtain a list of databases
   available on a database server
   @author Andrew Ensor
*/
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Scanner; // Java 1.5 equivalent of cs1.Keyboard

public class ShowDatabases
{
   private static final String DRIVER="com.mysql.jdbc.Driver";
   private static final String DB_URL="jdbc:mysql://raptor:3306/";

   public static void main(String[] args)
   {  // obtain user name and password from keyboard
      Scanner keyboardScanner = new Scanner(System.in);
      System.out.print("Please enter user name:");
      String userName = keyboardScanner.nextLine();
      System.out.print("Please enter password:");
      String password = keyboardScanner.nextLine();
      try
      {  Class.forName(DRIVER); // load the database driver for MySQL
         System.out.println("Trying to open connection to raptor");
         Connection con
            = DriverManager.getConnection(DB_URL, userName, password);
         Statement stmt = con.createStatement();
         // obtain result set holding names of current databases
         System.out.println("Listing databases on raptor");
         String command = "SHOW DATABASES"; // SQL command
         ResultSet rs = stmt.executeQuery(command);
         while (rs.next())
         {  // database name string is in first column of result set
            System.out.println(rs.getString(1));
         }
         System.out.println("Closing connection to raptor");
         stmt.close();
         con.close();
      }
      catch (SQLException e)
      {  System.out.println("SQL Exception:" + e);
      }
      catch (ClassNotFoundException e)
      {  System.out.println("ClassNotFoundException:" + e);
      }
   }
}
```

7.2 SQL Statements

Reading: none

The *Structured Query Language* is a standardized language for interacting with a relational database. By convention, one usually types an SQL keyword (such as `SELECT`) in uppercase so that it is easily recognizable, and spacing within a statement is not important. Some databases such as MySQL and Oracle require a semicolon to indicate the end of a statement, whereas other such as Sybase use a word like `GO`, but the JDBC driver automatically supplies the appropriate statement terminator.

SQL statements can be subdivided into four distinct groups:

Data definition language used to manipulate database structures and definitions, such as `CREATE`, `DROP`, `ALTER`, and `TRUNCATE` (for which the JDBC uses the `executeUpdate` method).

Data manipulation language used to change database data, such as `INSERT`, `DELETE`, and `UPDATE` (for which the JDBC uses `executeUpdate`).

Data query language used to get data from the database and impose ordering upon it, such as `SELECT` and `SHOW` (for which the JDBC uses `executeQuery`).

Rights used to give and remove access rights to database objects, such as `GRANT` and `REVOKE` (usually only available to the database administrator).

Once a connection has been established with a particular database a new table can be created by the statement:

```
CREATE TABLE table_name(
   att_name Data_Type,
   att_name Data_Type,
      ⋮
   att_name Data_Type,
   PRIMARY KEY(att_name, ..., att_name))
```

The `CREATE TABLE` can be proceeded by `IF NOT EXISTS`. Specifying a primary key, although recommended, is optional. The attributes that form the primary key should be specified as `NOT NULL`. Other constraints, such as foreign keys, no duplicates, restricted or default values can also be specified in the statement. The table can later be removed by the statement:

```
DROP TABLE table_name
```

or to avoid an exception if it might have already been dropped:

```
DROP TABLE IF EXISTS table_name
```

For example, the tables `Students`, `Courses`, and `Enrollments` can be created using the following statements:

```
CREATE TABLE Students(
   S_Id VARCHAR(20) NOT NULL,
   G_Name VARCHAR(20),
```

SQL Data Type	*Description*	*Java Equivalent*
`INTEGER` or `INT`	typically a 32-bit integer	`int`
`SMALLINT`	typically a 16-bit integer	`short`
`NUMERIC(m,n)` or `DECIMAL(m,n)` or `DEC(m,n)`	decimal with `m` total digits and `n` digits after decimal point	`java.sql.Numeric`
`FLOAT(n)`	decimal with `n` bits of precision	`double`
`REAL`	typically a 32-bit decimal	`float`
`DOUBLE`	typically a 64-bit decimal	`double`
`CHARACTER(n)` or `CHAR(n)`	fixed-length string of length `n`	`String`
`VARCHAR(n)`	variable-length string with maximum length `n`	`String`
`BOOLEAN`	a boolean value	`boolean`
`DATE`	calendar date, impl. dependent	`java.sql.Date`
`TIME`	time of day, impl. dependent	`java.sql.Time`
`TIMESTAMP`	date and time, impl. dependent	`java.sql.Timestamp`
`BLOB`	a binary large object	`java.sql.Blob`
`CLOB`	a character large object	`java.sql.Clob`

```
      S_Name VARCHAR(20) NOT NULL,
      Gender CHAR(1),
      B_Date DATE,
      H_Phone INTEGER,
      PRIMARY KEY(S_Id))

   CREATE TABLE Courses(
      CCN INTEGER NOT NULL,
      C_Name VARCHAR(40) NOT NULL,
      Fee DECIMAL(6,2) DEFAULT '0.00',
      Dept_Code CHAR(2),
      Level SMALLINT,
      PRIMARY KEY(CCN))

   CREATE TABLE Enrollments(
      S_Id VARCHAR(20) NOT NULL,
      CCN INTEGER NOT NULL,
      Semester SMALLINT NOT NULL,
      Year SMALLINT NOT NULL,
      Grade VARCHAR(2),
      PRIMARY KEY(S_Id, CCN, Semester, Year))
```

The `INSERT` statement is used for putting one record into a table by giving the values for the attributes in order, such as:

```
   INSERT INTO Students
      VALUES ('0412345', 'Jack', 'Bloggs', 'M', '1981-04-15',
      '9179990')
```

Existing records can be modified by using the `UPDATE` statement:

```
UPDATE Students SET H_Phone='9179876' WHERE S_Id = '0412345'
```

This updates values in the specified attribute for all records that match the `WHERE` clause. Similarly, records can be removed from a table by using the `DELETE` statement, with `WHERE` specifying the records to remove, such as:

```
DELETE FROM Students WHERE S_Id>'0599999'
```

Note that using the `DELETE` statement without a `WHERE` clause actually removes all the records from the table, and so should be used with caution.

At the heart of SQL is the `SELECT` statement, which is used to query the database, obtaining data from one or more tables. Its general form is:

```
SELECT attributes to select
   FROM tables to use
   WHERE filtering conditions on records
   GROUP BY attributes to group by
   ORDER BY ASC or DESC order
```

In its simplest form the `SELECT` statement can be used to retrieve all the data in a table by using `*` in place of the attributes:

```
SELECT * FROM table_name
```

Particular records can be obtained by using the `WHERE` clause, such as retrieving the students born since `1980`:

```
SELECT * FROM Students WHERE B_Date>='1980-01-01'
```

or the female students with surname `Bloggs`:

```
SELECT * FROM Students WHERE Gender='f' AND S_Name='Bloggs'
```

Besides the `AND` operator there are also `OR` and `NOT` operators which can be combined using round brackets to form complex conditions.

The `SELECT` statement can be made to retrieve only particular attributes by specifying the attributes. Selecting particular attributes from a table is known as a *projection* of the table. For example, the student ids and semester for students that enrolled in the course with control number `716180` in the year `2005` can be retrieved by the statement:

```
SELECT S_Id, Semester FROM Enrollments
   WHERE CCN='716180' AND Year='2005'
```

Note that a query such as:

```
SELECT S_Id FROM Enrollments WHERE CCN='716180'
```

simply returns the student id for each student enrolled in that course, so if a student was enrolled more than once then their id will appear more than once in the result set. To avoid this duplication one can insist that the `SELECT` statement just retrieve each result once by using the `DISTINCT` condition, such as:

```
SELECT DISTINCT S_Id FROM Enrollments WHERE CCN='716180'
```

The results of a query can also be sorted into either ascending (default) or descending order by stating the attribute(s) to order by. For example:

```
SELECT S_Name FROM Students
   WHERE H_Phone IS NULL ORDER BY B_Date
```

will retrieve the surnames of all students that do not have a home telephone number ordered with the youngest student first.

Besides specifying specific attributes for the `SELECT` query to retrieve, one can provide wildcards, where the underscore symbol (_) will match any single character, and the percentage symbol (`%`) will match any number of characters. These wildcards are incorporated in a `SELECT` query by using either the `LIKE` or the `NOT LIKE` comparison operators. For example, the statement:

```
SELECT S_Id, G_Name, S_Name FROM Students
   WHERE G_Name LIKE 'Ja%'
```

will retrieve the ids and names of all the students whose given name starts with the letters `Ja` (such as `Jack` and `Jade`, but not `Jill`).

Part of the power of relational databases is the ability to efficiently perform queries across several tables, temporarily joining the tables together by specifying the relationship between the attributes in each table in the `WHERE` clause. A *join* is a database operation that relates two or more tables by the values they share in common. For example, the student names (not just their ids) and grades for students that not received a D grade in the course with control number `716180` can be obtained by:

```
SELECT Students.G_Name, Students.S_Name, Enrollments.Grade
   FROM Students, Enrollments
   WHERE Students.S_Id = Enrollments.S_Id
   AND Enrollments.Grade<>'D' AND Enrollments.CCN='716180'
```

Note here that the attribute names have been preceded by the table names to avoid ambiguities with identically-named attributes. Similarly, to retrieve the names of students that are or have been enrolled in a course taught by the department with code `SA`, one uses the statement:

```
SELECT DISTINCT Students.G_Name, Students.S_Name
   FROM Students, Courses, Enrollments
   WHERE Students.S_Id = Enrollments.S_Id
   AND Courses.CCN = Enrollments.CCN
   AND Courses.Dept_Code = 'SA'
```

Exercise 7.2 (Retrieving Pet Information) *The* `raptor` *server has a database called* `pets`*. Describe the tables that are in the* `pets` *database and determine suitable SQL queries to retrieve the following information (your statements can be checked by using the* `MySQL-Front` *application available in the lab):*

- *details about pets named* `'Aldo'`*,*
- *the owners of female birds,*
- *the names of the living pets from oldest to youngest,*

- *the names of pets whose species starts with the letter* C*,*
- *the species, date and cost of each trip to the vet.*

Then determine what each of the following MySQL commands retrieves:

```
SELECT COUNT(*) FROM pets

SELECT owner,COUNT(*) FROM pets GROUP BY owner

SELECT species, MAX(birth) AS birth FROM pets
   WHERE death IS NOT NULL GROUP BY species

SELECT name, birth,
   (YEAR(CURRENT_DATE)-YEAR(birth))
   -(RIGHT(CURRENT_DATE,5)<RIGHT(birth,5)) AS age
   FROM pets ORDER BY age

SELECT pets.name,pets.owner,pets.species, SUM(cost) AS total
   FROM pets, vet
   WHERE pets.name = vet.name AND pets.owner = vet.owner
   GROUP BY pets.name, pets.owner

SELECT table1.name, table1.sex, table2.name, table2.sex
   FROM pets AS table1, pets AS table2
   WHERE table1.species = table2.species
   AND table1.sex='f' AND table2.sex='m'
```

7.3 Programming with a Database

Reading: none

Data query language statements such as `SELECT` return a table of data held in a `ResultSet` object, which also holds a reference (called the *cursor*) to the currently accessed row in the table. The `ResultSet` method `next` moves the cursor to the next row, returning whether there is a next row. The first call to the method `next` moves the cursor to the first row and makes it the current row. Once the cursor is positioned on a row, appropriate `ResultSet` methods `get`*`Xxx`* are used to retrieve the values in each column of the current row.

JDBC offers two ways to identify the column from which a `get`*`Xxx`* method gets a value. One way is to specify the attribute name as the parameter to the method, the other is to give the column index, where `1` indicates the first column (note that unlike array indices, database column numbers start at `1`). For example, the code fragment:

```
ResultSet rs=stmt.executeQuery("SELECT * FROM Enrollments");
while (rs.next())
{  String grade = rs.getString("Grade");
   int courseNumber = rs.getInt("CCN");
   System.out.println("Student has grade " + grade
      + " in the course " + courseNumber);
}
```

SQL Data Type	*Recommended get method*
`INTEGER` or `INT`	`getInt`
`SMALLINT`	`getShort`
`NUMERIC(m,n)` or `DECIMAL(m,n)` or `DEC(m,n)`	`getBigDecimal`
`FLOAT(n)`	`getDouble`
`REAL`	`getFloat`
`DOUBLE`	`getDouble`
`CHARACTER(n)` or `CHAR(n)`	`getString`
`VARCHAR(n)`	`getString`
`BOOLEAN`	`getBoolean`
`DATE`	`getDate`
`TIME`	`getTime`
`TIMESTAMP`	`getTimestamp`
`BLOB`	`getBlob`
`CLOB`	`getClob`

outputs all the grades with the corresponding courses. If the attributes `CCN` and `Grade` are in the second and fifth columns respectively of the result set, then `rs.getInt("CCN")` is equivalent to `rs.getInt(2)`, and `rs.getString("Grade")` is equivalent to `rs.getString(5)`.

In order to be able to move both forward and backward through a result set or jump to any position, a *scrolling* result set is required. If scrolling result sets are supported by the database driver then they can be enabled by obtaining a `Statement` object with the method:

```
Statement stmt = con.createStatement(type, concurrency);
```

where the `ResultSet` constant *`type`* is either the default `TYPE_FORWARD_ONLY` (result set is not scrollable), `TYPE_SCROLL_INSENSITIVE` (result set is scrollable but not sensitive to database changes), or `TYPE_SCROLL_SENSITIVE` (result set is scrollable and sensitive to database changes), and *`concurrency`* is either `CONCUR_READ_ONLY` (result set cannot be used to update the database) or `CONCUR_UPDATABLE` (result set can be used to update the database).

Information about the structure of a database or its tables is called *metadata* (as opposed to the actual data that is stored in the database). Such information is known by the person that designed a particular database, but not by a program that might be expected to work with arbitrary databases. The `ResultSet` method `getMetaData` returns a `ResultSetMetaData` object, which can be used to obtain information about the table, such as its number of columns and each column's name and type. For example, the following code fragment uses `ResultSetMetaData` to print the attribute names and rows of the table `Students`, without previously knowing the structure of the table. This code could be improved by incorporating the width of each column (rather than simply using tabs), obtained from `rsmd.getColumnDisplaySize(i)`:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM Students");
ResultSetMetaData rsmd = rs.getMetaData();
int columnCount = rsmd.getColumnCount();
// print the attribute names
for (int i=1; i<=columnCount; i++)
```

```
{  String columnName = rsmd.getColumnLabel(i);
   System.out.print(columnName+"\t");
}
System.out.println();
// print each row of the result set
while (rs.next())
{  for (int i=1; i<=columnCount; i++)
   {  String value = rs.getString(i);
      System.out.print(value+"\t");
   }
   System.out.println();
}
```

If a statement is to be executed many times, it is usually more efficient to use a `PreparedStatement` rather than a `Statement`. A `PreparedStatement` is given an SQL statement when it is created, and in most cases is sent to the DBMS immediately to be compiled and optimized. For example, if there are many fees in the `Courses` table that need to be updated, a `PreparedStatement` might be used:

```
PreparedStatement updateFeesStmt = con.prepareStatement(
   "UPDATE Courses SET Fee = ? WHERE CCN = ?");
```

In order to use the `PreparedStatement`, values must be supplied in place of the question mark placeholders (if there are any). The values are supplied by using an appropriate `set`*Xxx* method, specifying the number of the placeholder (starting at `1`) and the value to put there. Then the `PreparedStatement` is executed using the appropriate execute method. Typically this would appear in a loop, such as the following:

```
PreparedStatement updateFeesStmt = con.prepareStatement(
   "UPDATE Courses SET Fee = ? WHERE CCN = ?");
int[] courses = {716180, 716181, 145612};
double[] newFees = {499.0, 499.0, 449.0};
for (int i=0; i<courses.length; i++)
{  updateFeesStmt.setDouble(1, newFees[i]);
   updateFeesStmt.setInt(2, courses[i]);
   updateFeesStmt.executeUpdate();
}
```

When a connection is made to a database it is in auto-commit mode by default, meaning that each SQL statement is treated individually. A *transaction* is a set of one or more statements that are executed together as a unit, so either all the statements are executed, or none are executed. To group statements into a transaction, auto-commit mode should be first disabled:

```
con.setAutoCommit(false);
```

Once auto-commit mode is disabled, statements are not committed until the `commit` method is used:

```
con.commit();
```

Transactions help preserve the integrity of the data in the database, since it provides some protection against conflicts that can arise when two users access data concurrently. The DBMS uses locks to block access by others to the data that is being accessed by a transaction (whereas when auto-commit mode is enabled, locks are only held for one statement at a time). If while executing a transaction a `SQLException` is thrown, then the method `rollback` can (if supported by the DBMS) be called to abort the entire transaction.

The Swing `JTable` component is used to display a two-dimensional grid of data. A `JTable` either obtains its data from an `Object[][]` array, or else from a `TableModel` object (following the model-view-controller pattern). Swing tables have rich behaviour, such as enabling the user to drag or resize columns, and allow custom cell renderers (so that objects in the table are not just simply displayed as strings). In order to appear, usually the table would be placed inside a scroll pane before being added to a GUI panel.

Creating a table from an array is quite simple and is suitable if the contents of the table are fixed, and not held by another data structure. The content for the tables and the column headings are given to the `JTable` constructor:

```
Object[][] cells =
   {{"Mercury", new Integer(2440), Color.YELLOW},
    {"Venus", new Integer(6052), Color.YELLOW},
    {"Earth", new Integer(6378), Color.BLUE},
    {"Mars", new Integer(3397), Color.RED},
    {"Jupiter", new Integer(71492), Color.ORANGE},
    {"Saturn", new Integer(60268), Color.ORANGE},
    {"Uranus", new Integer(25559), Color.BLUE},
    {"Neptune", new Integer(24766), Color.BLUE},
    {"Pluto", new Integer(1137), Color.BLACK}};
String[] colNames = {"Planet", "Radius", "Colour"};
JTable table = new JTable(cells, colNames);
JScrollPane pane = new JScrollPane(table);
```

If the content for the table are already held by another data structure, if they could change, or the table should be editable then a table model is preferable to using an `Object[][]` array. A table model must implement the interface `TableModel`, but the abstract class `AbstractTableModel` implements most of the required methods, leaving subclasses to implement three methods: `getRowCount`, `getColumnCount`, and `getValueAt`. If the contents of the table should be editable by the user, then the table model should override the methods `isCellEditable` (which by default returns `false` for every row and column) and `setValueAt` (and possibly include custom cell editors).

The `ResultSetDisplayer` demonstrates how a table is created from a table model which uses a result set for its data. Note that a result set is passed to the `ResultSetTableModel` when it is created so that it knows what to return when the table calls its `getValueAt` method. This example also demonstrates using a scrollable cursor which updates the database when the result set is changed, and uses a `WindowListener` to close the database connection when the window is being closed.

Exercise 7.3 (Query GUI) *Enhance Exercise 7.1 so that it also contains a text area, a button, and a table component. SQL query statements should be* *able to be entered into the text area and the results shown in the (non-editable) table when the button is pressed.*

```
/**
   A class that demonstrates how a JTable can be used to display and
   edit a ResultSet from a database and updates the database
   @author Andrew Ensor
*/
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Toolkit;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Scanner; // Java 1.5 equivalent of cs1.Keyboard
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;
import javax.swing.table.TableModel;
import javax.swing.WindowConstants;

public class ResultSetDisplayer extends JPanel
{
   private static final String DRIVER = "com.mysql.jdbc.Driver";
   private static final String DB_URL
      = "jdbc:mysql://raptor:3306/universitydb";
```

cont-

-cont

```
   public static void main(String[] args)
   {  // obtain user name and password from keyboard
      Scanner keyboardScanner = new Scanner(System.in);
      System.out.print("Please enter user name:");
      String userName = keyboardScanner.nextLine();
      System.out.print("Please enter password:");
      String password = keyboardScanner.nextLine();
      try
      {  Class.forName(DRIVER); // load the database driver for MySQL
         System.out.println
            ("Trying to open connection to raptor/universitydb");
         final Connection con
            = DriverManager.getConnection(DB_URL, userName, password);
         // create statement that will be sensitive to database changes
         // if supported by database
         final Statement stmt = con.createStatement
            (ResultSet.TYPE_SCROLL_SENSITIVE,
            ResultSet.CONCUR_UPDATABLE);
         // obtain sample result set holding contents of Students table
         String command = "SELECT * FROM Students"; // SQL command
         ResultSet rs = stmt.executeQuery(command);
         // prepare a ResultSetDisplayer in a JFrame
         JFrame frame = new JFrame("ResultSetDisplayer");
         frame.setDefaultCloseOperation
            (WindowConstants.DISPOSE_ON_CLOSE);
         frame.addWindowListener(new WindowAdapter()
            {  public void windowClosing(WindowEvent we)
               {  // close the database connection when frame closes
                  System.out.println("Closing connection to raptor");
                  try
                  {  if (stmt != null) stmt.close();
                     if (con != null) con.close();
                     System.exit(0);
                  }
                  catch (SQLException e)
                  {  System.out.println("SQL Exception:" + e);
                  }
               }
            });
         frame.getContentPane().add(new ResultSetDisplayer(rs));
         frame.pack();
         // position the frame in the middle of the screen
         Toolkit tk = Toolkit.getDefaultToolkit();
         Dimension screenDimension = tk.getScreenSize();
         Dimension frameDimension = frame.getSize();
         frame.setLocation((screenDimension.width-frameDimension.width)
            /2, (screenDimension.height-frameDimension.height)/2);
         frame.setVisible(true);
      }
```

cont-

-cont

```
      catch (SQLException e)
      {  System.out.println("SQL Exception:" + e);
      }
      catch (ClassNotFoundException e)
      {  System.out.println("ClassNotFoundException:" + e);
      }
   }

   /**
      An inner class that is the table data model for an updateable
      result set from a database query
   */
   private class ResultSetTableModel extends AbstractTableModel
   {
      private ResultSet rs;
      private ResultSetMetaData rsmd;

      public ResultSetTableModel(ResultSet rs)
      {  this.rs = rs;
         try
         {  rsmd = rs.getMetaData();
         }
         catch (SQLException e)
         {  System.out.println(e);
         }
      }

      public String getColumnName(int col)
      {  try
         {  return rsmd.getColumnName(col + 1);
         }
         catch (SQLException e)
         {  System.out.println(e);
            return "";
         }
      }

      public int getColumnCount()
      {  try
         {  return rsmd.getColumnCount();
         }
         catch (SQLException e)
         {  System.out.println(e);
            return 0;
         }
      }
```

cont-

-cont

```
      public int getRowCount()
      {  try
         {  rs.last();
            return rs.getRow();
         }
         catch(SQLException e)
         {  System.out.println(e);
            return 0;
         }
      }

      public Object getValueAt(int row, int col)
      {  try // use scrolling cursor to retrieve value from result set
         {  rs.absolute(row + 1); // position cursor at the row
            return rs.getObject(col + 1);
         }
         catch(SQLException e)
         {  System.out.println(e);
            return null;
         }
      }

      public boolean isCellEditable(int row, int col)
      {  return true; // all cells are presumed editable
      }

      public void setValueAt(Object obj, int row, int col)
      {  try // use scrolling cursor to update value in result set
         {  rs.absolute(row + 1); // position cursor at the row
            rs.updateObject(col + 1, obj); // updates rowset
            rs.updateRow(); // updates database
         }
         catch(SQLException e)
         {  System.out.println(e);
         }
      }
   }
}
```