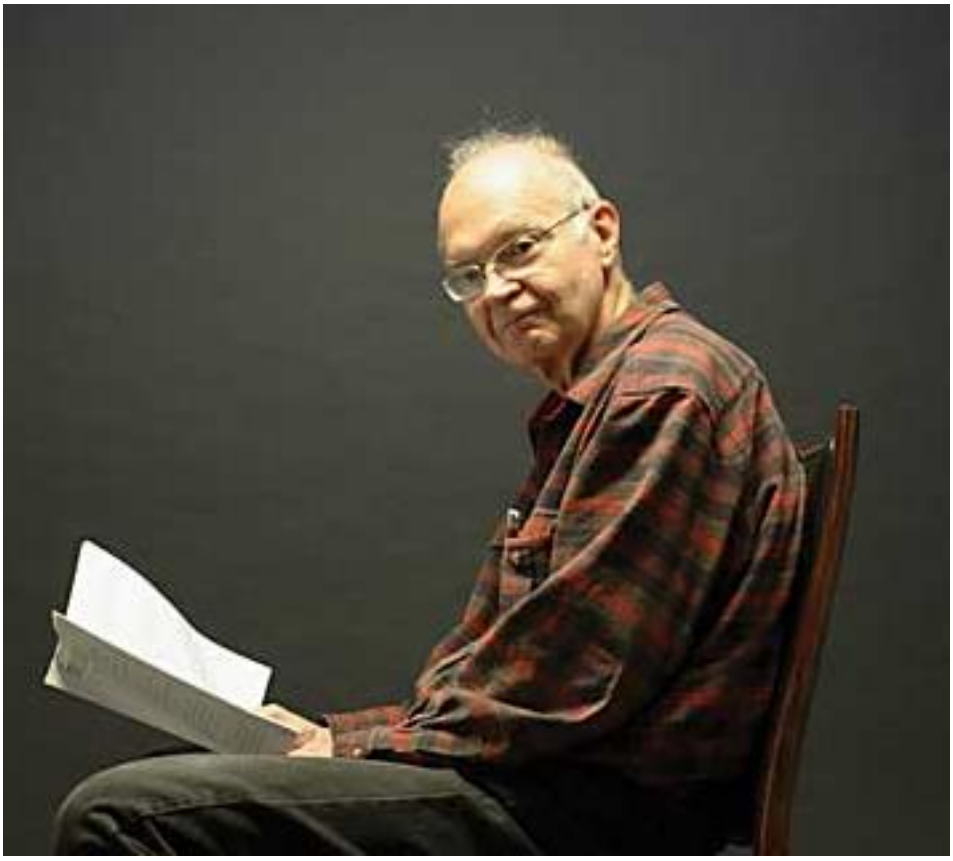
Data Structures & Algorithms in Java (2017)

Tony Mullins

Algorithms + Data Structures = Programs
Nicklaus Wirth (1934-)

The order in which the operations shall be performed in every particular case is a very interesting and curious question, on which our space does not permit us fully to enter. In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selection amongst them for the purposes of a Calculating Engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.
Ada Byron 1842 (notes on the analytical engine)

Donald Knuth (1938-), the father of data structures and the analysis of algorithms.



Some scientists are like explorers who go out and plant the flag in new territory; others irrigate and fertilize the land and give it laws and structure. Knuth

Contents

Introduction

All programs that we write and execute must meet certain contractual obligations. Every program has a purpose no matter how trivial. One way to think about the relationship between a program and its user is in terms of a contract between a client and a supplier. In this case the contract is between the user of the program, the client, and the program itself, the supplier. A programmer, the client, uses an editor, the supplier, to type and edit code for a program. The editor, if correct, carries out the instructions issued by the programmer typing and editing a text file. The editor program has a purpose and guarantees to supply a correct service to the programmer editing a text. Of course the client must also know how to request services from its supplier and in the case of using an editor this means knowing how to issue instructions to it. In reality we don't restrict the client supplier relationship to persons and the programs they use. A client may also be another program or an instance of an object that provides a service to another object. This means that in the object-oriented world the client supplier contract extends to define the relationship between class instances. The `main` method in a program will use the services of objects that it creates. Therefore, the relationship between a class and its clients becomes a formal agreement that commits each party to a set of rights and obligations. The supplier has an obligation to provide the services it makes public correctly and has the right to request clients to provide correct information when using its services. The client has the right to expect that services provided by suppliers are correct and an obligation to make itself familiar with the requirements of its suppliers. This type of formal contract between clients and suppliers is known as design by contract. For example, a client, withdrawing money from an account should not try to remove more money than is available by making itself aware of the current balance before trying to withdraw money. On the other hand, an Account class must guarantee that all balances are correct and that the rules governing control of the account are enforced. The issue here is one of reliability and its associated issues of correctness and robustness. Software is correct if it performs according to its specification; it is robust if it is capable of handling situations not covered in the specification. (A discussion of a formal approach to issues of correctness is discussed in *An Introductrion to Formal Design Methods* by this author. A discussion of design by contract in relation to object oriented prgoramming can be found in *Programming Paradigms with Scala* or *Programming Paradigms with Java* also by this author). Issues of robustness are typically handled by using exception handlers. (See *Object-Oriented Programming and the Story of Encapsulation*, chapter 9, by this author).

It turns out that in practice it is also necessary for programs, i.e suppliers, to deliver results in a *timely fashion*. For many of us this requirement may come as a surprise because we imagine computers are the fastest things in the modern world and are noted for providing results instantly. While it is entirely reasonable to expect that programs provide services that are correct and meet their specifications it seems a bit unusual to make efficiency of performance

a requirement. However, when we think about it, most of us at one time or another have given up using a program because it is too slow to use. We expect to receive results in a matter of seconds; otherwise we won't like using the program. In the early days Internet browsing was very slow and people often complained about the speed of connections. Some people even gave up browsing! Another good example is the use of WiFi. In the early days many argued that it would never be widely used because it was too slow. It turns out that this requirement is a necessary one because delivery of efficient service is crucial in the provision of services that may save lives. As we are all aware computing is now ubiquitous in the modern world and many crucial services are provided by computers. Computer systems play a critical role in hospitals, aviation, transport services, weather reporting and control systems. A brain scanner must provide results of a brain scan in optimum time if a consultant is to treat a brain injury in time to save the life of the patient. We also have computers that use Geographical Information Sytems to fly drones and missiles. Again these systems must be capable of optimum response times if they are to avoid possible tragedy. In the case of modern cars the airbags must release in time with a force proportional to that of the impact to avoid serious injury to passengers in the event of a crash. If computer systems play a fundamental role in the management and monitoring of critical services then they must respond within time frames that meet certain requirements. At a minimum they must try to provide services in optimum time frames. By optimum we mean that critical processes return results in the shortest time frame possible. This means that we must be able to analyse the performance of algorithms and use the result of this analysis to choose between them. But you may ask how do we define the shortest time possible? And how do we measure and compare the performance of programs? These are some of the questions we will try to answer in this lecture series.

To attempt to deliver optimal solutions to problems two possible avenues will be explored: **Algorithmic Analysis** and **Data Structures**.

In the case of algorithms one approach is to develop different programs to solve a given problem and then compare the resulting solutions in terms of performance. We can do this by using agreed benchmarks to test the performance of the different solutions. This would involve not only the choice of data sets to test but also the computer platform used to do the testing. Alternatively,we could develop a classification system that allows the comparison of algorithms independently of their execution. This approach requires a mathematical model for measuring the performance of algorithms. It has the advantage that it will provide a general classification system for performance analysis. We will develop such a model in the chapter on the Analysis of Algorithms and we will use this model to compare different algorithms that solve a given problem. To illustrate what we mean by this we take a very simple example that computes the sum of the first N natural numbers. Two separate functions that solve the problem are listed below.

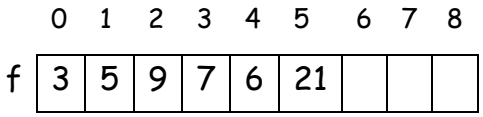
```
static long sumN(long n){
    long s = n * (n + 1)/2;
    return s;
}

static long sumN1(long n){
    long s = 0;
    for(int j=0; j < n; j++) s = s + (j + 1);
    return s;
}
```

Both solutions are correct, assuming $n \geq 0$. But if you had to choose one, which one would it be? To choose, you might consider testing (benchmarking) both functions and check their performance. If you did so, for large values of n, you would find that sumN is much faster than sumN1. In fact it is potentially thousands of times faster. Using a benchmark of n equal to 1000000 and executing it on a standard PC it turned out to be 15700 times faster. But we can also analyse the actual code sequences and use this analysis to compare their performance. A simple inspection of the code shows that sumN calculates the result using a formula and sumN1 does so by iterating over a loop of size n. The iteration involves the execution of at least n instructions and, therefore, requires a greater computation time. A computation time that grows linearly as n increases; whereas, the cost of executing the formula is constant no matter how big n becomes. The greater the value of n, the worse the performance of sumN1; whereas, the computation cost of sumN remains constant. The search for optimal solutions to problems is one of the key goals of algorithmic analsysis.

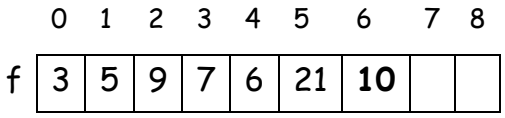
The second avenue to explore is that of designing different ways of structuring data and then examining the performance benefits of the different structures. A data structure provides a particular way of storing and organizing data so that it can be used efficiently. It also provides an ordering between the elements in the collection so that the cost of insertion and retrieval is optimized. The second point is crucial because there is often a trade off between the cost of inserting a new item of data while maintaining the ordering and the cost of retrieving or finding an element in the collection. The goal is to try to optimize both. There are many different types of data structure. Wirth identifies *record* (similar to what we would call a simple class with only public attributes, a constructor and no methods or what is called a case class in Scala) and *array* as two of the basic building blocks of all data structures. The implementation of a data structure usually requires writing a set of procedures and functions that create and manipulate instances of that structure. The choice of data structure to use depends greatly on the data set to be modeled and the performance requirements of the application.

To illustrate this point about trying to optimize both the cost of insertion and retrieval and the choice of data structure we consider the problem of managing a collection of integer values in such a way that we optimise, if possible, the cost of both activities. In the first instance we consider using a linear array to store the numbers. Two cases are considered: an unsorted sequence and a sorted sequence. Given the sequence below we can add a new value to the array by simply appending it to the given sequence. To add 10 we simply write `f[6] = 10`. This statement has a fixed cost that is negligible and, hence, is optimal.

Before



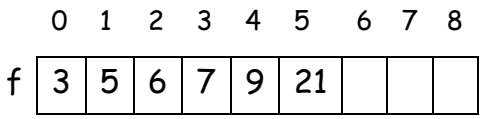
After



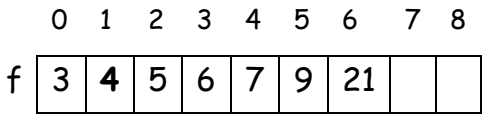
However, when we want to search this sequence for some value, say x, we have to perform a *linear search*. This means that we begin at the start and check every value in sequence until we find x or reach the end of the sequence and fail to find it. The cost of this search is directly proportional to the size of the sequence. Let us say that it has a cost of n where n equals the number of elements in the array. We conclude that the cost insertion is *constant* and the cost of retrieval is *linear*.

Next we consider a sorted sequence. To add, say 4, now is more complex because we have to find where to place 4 in the sequence so that the whole sequence remains sorted after 4 is inserted. This means that we have to shift existing elements one position to the right until we find a value that is less than 4 or we reach the beginning of the sequence. The cost of this insertion is directly proportional to the number of elements that have to be moved. In the worst case this requires n moves.

Before



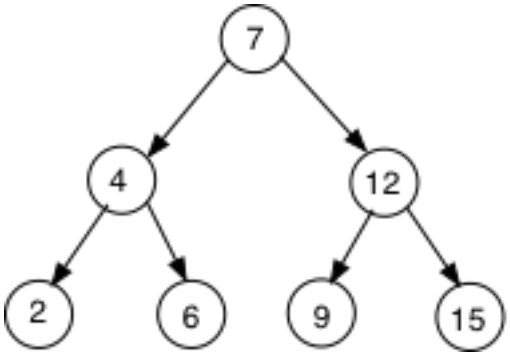
After



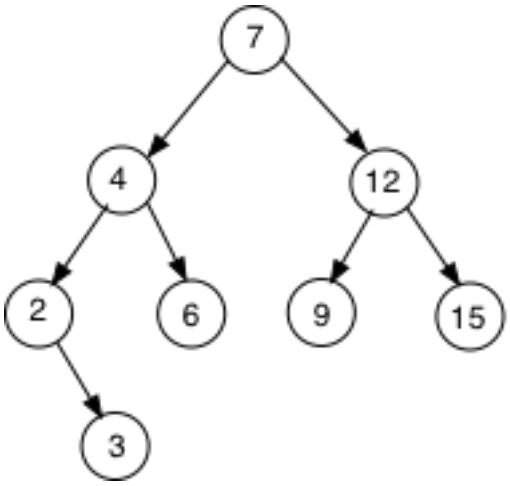
Keeping the values sorted has a big advantage in the case of searching the array for x because we can now use *binary search*ing. This means that every time we check a value we can reduce the search space, i.e. the number of values to check, by half. The cost is $\log_2 n$. This is a very small number. For example, if there are 1Mb of values the cost is only 20 because $\log_2 2^{20} = 20$. In the case of searching this is an optimal value. We conclude that in this cae the cost of inserting a single new value is *linear* and the cost of retrieval is $\log_2 n$, i.e. optimal.

How to decide what arrangement to choose? One possible answer might be to analyse the frequency of the different operations. If the number of additions greatly exceeds the number of searches then the unsorted sequence will perform best. However, if the opposite is the case then the second sorted sequence will provide the best solution.

Is there a data structure that would optimize both? The answer, in fact, is yes. By using a non-linear data structure called a *binary search tree* the cost of both insertion and retrieval can both be reduced to $\log_2 n$. The diagram below shows a balanced binary search tree. Notice that the numbers are arranged so that there is a single root value and that each node in the tree has two descendents. Also notice that the values to the left of any node are always less than it and those to the right of it are always greater than it.



To add a value, say 3, to the tree we start at the root and either go left or right, repeating the movement until we reach the bottom of the tree. 3 is less than 7 so go left; 3 is less than 4 so go left; 3 is greater than 2 so go right. This is the end of the tree so insert here. The diagram below shows the new tree. This has a cost of $\log_2 n$ because each time we check a node we eliminate half the tree.



The cost of searching is also binary because we can eliminate half the values each time we check a node in the tree. Hence, the cost of insertion and retrieval are both optimised. (Note that keeping a tree balanced in this way may incur an overhead that makes insertion more expensive than searching.) We will discuss binary search trees later in the course and explain how to implement them.

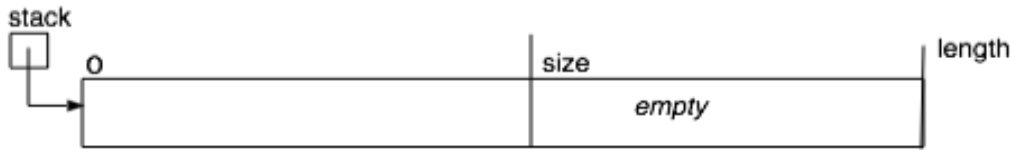
These two approaches to the development of optimal solutions to problems are complimentary. In fact they are intimately bound together so much so that Wirth coined the equation: *Algorithms + Data Structures = Programs* and used it as the title of his book on the programming language Pascal and data structures Therefore, programming without data structures is not possible and vice versa. To quote Wirth:

> *Hoare .. made clear that decisions about structuring data cannot be made without knowledge of the algorithms applied to the data and that, vice versa, the structure and choice of algorithms often strongly depend on the structure of the underlying data. In short, the subjects of program composition and data structures are inseparably intertwined.*
>
> *(Algorithms + Data Structures = Programs, Preface)*

The purpose and goal of our study of algorithms and data structures is to find the most efficient algorithms, in the temporal sense, to solve problems. (We could also study issues of efficiency in relation to memory usage but this is outside the scope of this work.) One must point out, at the outset, that there are some problems for which no efficient solutions are possible. In fact there is a whole class of related problems, known as NP-complete problems, for which no known efficient alforithms exist. This class of problem are unusual in the sense that if an efficient algorithm was ever found to solve one, then a solution to all of them would have been found. Examples are the Travelling Salesman problem and the Bin Packing problem. The travelling Salesman problem is the problem of finding the minimal distance required to travel by a salesman who has to visit n cities and return to his starting point. The solution requires n! combinations and, as a consequence, is intractable. We will return to this again in the chapter on the analysis of algorithms.

Object-Oriented Data Structures

The object-oriented approach to programming is to put all the focus on the encapsulation of data by classes and the provision of public methods that allow communication to take place between instance objects. When it comes to defining data structures in an object-oriented fashion the story is the same one. A data structure is defined by an encapsulating class. The implementation of the data structure involves making a decision about how to store and order the encapsulated data and also the implementation of the public methods that create and manipulate instances of that structure. The client does not necessarily know how the data is stored but it does need to know something about the ordering of data in the data structure and the rules, if any, governing the semantics of the data structure. It also needs to know the semantics of the public methods. To clarify, we take the well understood example of a stack. A stack is a data structure that exhibits last in-first-out behaviour. By this we mean that the last element pushed onto the top of a stack is the first element to be popped from the stack. The usual procedures and functions associated with stack are: `push(x)`, that pushes x onto the top of a non-full stack; `pop()`, that removes the top element of a non-empty stack; `top()`, that returns the top element of a non-empty stack and leaves the stack unchanged. For the purposes of this example we implement a fixed size stack of integer values. The actual stack is implemented using a linear array indexed `0..size-1`. The top of the stack has index value `size-1`. The public methods provide an interface that enforce the stack behaviour and maintain a sequence of integers where *the last element pushed is the next element to be popped*. The actual `stack` is private to the class. A picture of an actual stack might be:



The method `push` adds a new element to the top of the stack, if it is not full. Method `pop` removes an element from the top of a non-empty stack. Method `top` returns the element at the head of the stack, returning `null` if the stack is empty. The methods `full`, `empty` and `size` return information about the current state of the stack.

```
class IntStack{
   private Integer stack[];
   private int size;
   IntStack(int n){ //assume n > 0
      stack = new Integer[n];
      size = 0;
```

```
    }
    boolean push(Integer x){
        if(size == stack.length) return false;
        stack[size] = x;
        size++;
        return true;
    }
    boolean pop(){
        if(size > 0){
            size--;
            return true;
        }
        else return false;
    }
    Integer top(){
        if(size > 0) return stack[size-1];
        else return null;
    }
    boolean empty(){return size == 0;}
    boolean full(){return size == stack.length;}
    int size(){return size;}
}
```

This implementation is type specific and can only be used for stacks of integer values. This has the consequence that we would need to write multiple type specific stacks that all exhibit the same behaviour. This can be avoided by writing a single generic stack that is type parameterized. We will discuss how to use genericity when implementing data structures later in the chapter on generic stacks and queues.

Collection Classes in Java

The Java API provides container classes, called the `Collection` classes, that provide a number of different standard data structures. The main data structures (object containers) supported are `Set`, `List` and `Map`. The Set interface is implemented by both `HashSet` and `TreeSet`. Both implementations enforce the basic rule of sets that duplicate values are not allowed. Both classes support genericity and as a result ensure that all values in a set are the

same type. The List interface is implemented by ArrayList, LinkedList, Stack, Dequeue and Vector. In fact class Stack inherits class Vector and provides methods: push(E item), pop(), empty(), search(Object ob) and peek() that provides the same role as top() defined above . There are also HashMap and TreeMap implementations of mapping functions. (See Java Docs for more details on these and their associated methods.) Some of these classes have been discussed in detail in the book *Object-Oriented Programming and the Story of Encapsulation* (see Chapter 10). It is important to remind ourselves of some key factors when working with these container classes.

1. All classes must implement certain methods if the container classes are to work as intended. The methods are: equals, hashCode and toString. In the case of TreeSet and TreeMap they must also implement the Comparable<E> interface. For example, if a class does not implement an equals method then the Set implementations will allow duplicates. In the case of an ArrayList, if sorting is required then the data instances must implement the Comparable<E> interface. Without it the sorting algorithm has no way to determine a correct ordering of the elements.

2. These data structures do not enforce encapsulation of underlying data. This has consequences for clients because if a client modifies attributes of objects contained in one of the container classes then data can be lost. For example, if a client modifies an attribute in an object that is used to calculate the hashCode then that data item will not be retrieved when next searched for. We will return to this issue again in the chapter on Hashing. The same situation arises for a TreeSet if a client modifies an attribute used to order elements in the tree.

One way to guarantee that both points discussed above are met is to write classes so that at a minimum those attributes used to define equality and, hence, hashCode are immutable. In cases where the Comparable interface is to be implemented then all attributes used in the compareTo method should also be immutable. This means that these attributes should be declared final and no public methods that could be used to modify them should be provided.

In fact there is a school of thought that strongly argues that objects should be immutable. This school argues in favour of what is termed functional object-oriented programming. In this world a class encapsulates state that cannot be modified. All its public methods behave as functions that always return the same values. Hence, the state cannot be modified after the construction phase. All the information or data for the class is fixed at the point of construction and cannot change during the lifetime of the object. We say that instances of such classes are immutable. Examples of immutable objects that we have studied are the

`String` and `Integer` classes. Writing classes in this way satisfies Odersky's requirement that *what we want is programs that transform immutable values instead of stepwise modifications of mutable state*. Both Bloch and Goetz also put forward a similar view. They both argue that classes should be immutable unless there is very good reasons why they should not be.

Classes should be immutable unless there's a very good reason to make them mutable....If a class cannot be made immutable, limit its mutability as much as possible. (Bloch)

We have to adopt the orientation that immutable state is better than mutable state, and seek to increase immutability and decrease sharing. (Goetz)

To make classes immutable we should use the following list of rules.

1. Do not provide any methods that allow a client to modify the class state.

2. Ensure that public methods cannot be overridden by making the class `final`. This prevents subclasses from compromising the immutable behaviour of the class and means that other classes cannot inherit or extend it.

3. Make all attributes `private`.

4. Make all fields `final`

5. Ensure exclusive access to any mutable components. This ensures that clients cannot access references to mutable objects and, hence, violate encapsulation. If access is required then make defensive copies of the mutable object components.

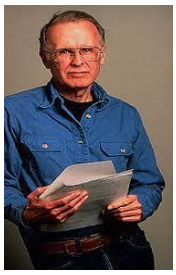
History of Data Structures

The history of the development of data structures is, as Wirth and Hoare state, intimately linked to the history of the development of algorithms. We know that the development of data structures is as old as the first programming language, *FORTRAN*(FORmula TRANslator) (John Backus, 1954). The stack data structure was first proposed in 1955, and then patented in 1957, by the Germans Klaus Samelson and Friedrich L. Bauer. The same concept was developed independently, at around the same time, by the Australian Charles Leonard Hamblin. At a meeting in Darmstadt, Germany, in 1955 a number of mathematicians and computing engineers came together and decided to develop a universal programming language. This language should be machine independent and should be used to allow people to communicate algorithms with each other. This language became know as *Algol60*. The

word *Algol* stands for Algorithmic Language. Dijkstra proposed using a stack to implement recursive procedures as part of this language.(References required here). Algol was the first machine independent programming language.(FORTRAN was machine dependent for a very small subset of its statements.)

An Algol program that computes the mean of an array of positive numbers is given below. Notice the use of the words begin – end to enclose code blocks and data type real for floating point numbers. Notice the use of an array to store real values. Assignment is denoted by := (read as *becomes equal to*). The remainder of the program should be self explanatory to a modern day programmer.

```
begin
  integer N;
  Read Int(N);
  begin
    real array Data[1:N];
    real sum, avg;
    integer j;
    sum:=0;
    for j := 1 step 1 until N do
      begin real val;
        Read Real(val);
        Data[j] := if val < 0 then -val else val
      end;
    for j := 1 step 1 until N do sum:=sum + Data[j];
    avg:=sum/N;
    Print Real(avg)
  end
end
```



One day in the Spring of 1949, a 25 year old young man visited the IBM Computer Centre on Madison Avenue, took the test for programmers and got the job. The name of this young man was John Backus (1924-2007). Backus was assigned to work on the SSEC machine that had no memory and used punch cards. It was a difficult job to work with this machine. He said: *You just read the manual and got the list of the instructions and that was all you knew about programming.* But, nevertheless, he enjoyed it and said it was fun! In December of 1953 Backus headed up a team that were given the task of designing an actual programming language for the IBM 704. This language became known as Fortran (Formula Translation). In actual fact, von Neumann disagreed with this language

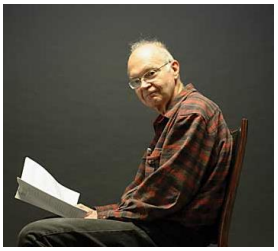
development and did not see the need for high-level languages. But he relented! Backus and his team went on to write the first compiler that mapped a high-level language onto machine code. This language even had a DO loop. Programming, as we know it today, was born. In May 1958, he attended a conference in Zurich whose aim was to improve Fortran and create a standardized computer programming language. This team of people went on to create the International Algebraic Language that became known as Algol. The names of some of the people involved are synonymous with computing: Dijkstra, Hoare, Wirth, McCarthy, Naur, Backus, Perlis, Bauer. As part of his work on this project Backus, together with Naur, went on to develop a notation that is still used to describe the syntax of programming languages. Using the work of Chomsky on context-free grammars they developed what is now called BNF (Backus-Nauer form). The beauty of this formalism is that if you write a program following its rules you always get a syntactically correct piece of code. As an example of the use of BNF we define a constant with the following rules:

```
<constant> ::= <digit>                              rule1
<constant> ::= <constant><digit>                    rule2
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9   rule3
```

Rule1 states that a constant may be composed of a digit. Rule2 says a constant may be composed of a constant followed by a digit. Rule3 says a digit is 0 or 1 or 2 or .. Used together these rules form a grammar for constants. To show that 562 is a constant we can construct it using the given rules as follows:

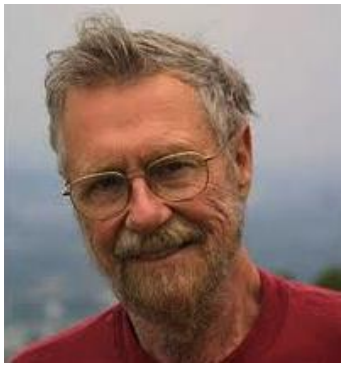
```
<constant>  => <constant><digit>          rule2
            => <constant> 2               rule3
            => <constant><digit>5         rule2
            =><constant>  62              rule3
            => <digit>62                  rule1
            => 562                        rule3
```

In 1977 Backus was given a Turing Award and he used the occasion to criticize the von Neumann fetch-execute model of computing. He stated that the process of fetching data from memory, processing it and returning the result to memory resulted in a loss of clarity. He advocated instead a functional approach and he proposed a language that he called FP. The title of his talk was *Can Programming be Liberated from the von Neumann Style?*



Donald Knuth (1938 -) began working on his multi volume seminal work, The *Art of Computer Programming*, in 1963. To-day, nearly sixty years later, he is still working on it. Volume 4A was published in hardback in 2011 and there are still two further volumes planned. In total the multi-volume book runs to thousands of pages and, is primarily, concerned with

the analysis of algorithms and data structures. Knuth provided a rigorous analysis of the computational complexity of algorithms and developed formal techniques for doing the analysis. In fact he is often referred to as *the father of analysis*. This type of analysis provided a measure of the performance of algorithms and allowed a classification of performance in terms of what are termed asymptotic functions.



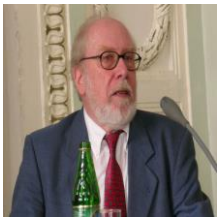
Edsger Dijkstra(1930 – 2002) received a Turing Award in 1972 for his contributions to the development of programming languages. During his lifetime Dijkstra contributed to all fields of computing. He worked on the design of operating systems, invented the semaphore used to synchronize access to shared resources, designed programming languages and wrote many very important algorithms. To-day when you use mapping software to plot a journey between two cities behind the scenes the application uses Dijkstra's shortest path algorithm to calculate the optimal journey in terms of distance. From the 1970's on he developed the theory of axiomatic semantics and applied it to the construction of programs. Dijkstra's methodology was to construct programs and their proofs in parallel. One starts with a mathematical *specification* of what a program is supposed to do and applies mathematical transformations to the specification until it is turned into a program that can be executed. The resulting program is then known to be *correct by construction.* Dijkstra had an unusual method of writing. All his papers (numbered *EWD0, EWD1,* ..) were written free hand with a fountain pen in black ink. They were then copied and distributed to a number of people who in turn copied them and distributed them to more people. More than 1300 papers are now available on the Web.

Tony Hoare (1934 -) is probably best known to students' world wide for his algorithm Quicksort that he developed in 1960 to sort a sequence of values in optimal time. He also developed Hoare logic for verifying the correctness of programs and created the notation CSP (communicating sequential processes) that specifies interactions between concurrent and parallel processes. He received a Turing Award in 1980 for his fundamental contributions to the definition and design of programming languages and in 1982 he was elected Fellow of the Royal Society. In 1977 he became Professor of Computer Science at Oxford University and on retiring from this position in 1999 he became a researcher with Microsoft. Tony Hoare has devoted his life to thinking about software systems. Here are some of his thoughts:

> *There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.*

Almost anything in software can be implemented, sold, and even used given enough determination. There is nothing a mere scientist can say that will stand against the flood of a hundred million dollars. But there is one quality that cannot be purchased in this way — and that is reliability. The price of reliability is the pursuit of the utmost simplicity. It is a price which the very rich find most hard to pay.

My original postulate, which I have been pursuing as a scientist all my life, is that one uses the criteria of correctness as a means of converging on a decent programming language design—one which doesn't set traps for its users, and ones in which the different components of the program correspond clearly to different components of its specification, so you can reason compositionally about it.



Niklaus Wirth(1934 -) a Swiss computer scientist who was given the Turing Award in 1984 for his work on the design of programming languages. He developed Pascal, Modula-2 and the Oberon family of languages. His book *Algorithms + Data Structures = Programs* (1976) provides an analysis of the fundamental data structures in performance terms as part of the development of the imperative programming project using the programming language Pascal.

Conclusion

The primary aim of this text is to provide a detailed introduction to the study of data structures and their associated algorithms. To do so it is also necessary to develop a methodology that allows for the analysis of the performance of algorithms. Our goal in the design of data structures is to provide ways to organise collections of data elements so that the cost of insertion and retrieval are optimised. In different situations we need to understand why one data structure *is better* than another to manage the data.

Another primary goal is to study algorithms and data structures from their foundations up. This means that, like Wirth, we will only use two fundamental building blocks: classes that encapsulate simple data entities and sequences or arrays. We will also use pointers or reference variables to build dynamic data structures. See the chapter on Dynamic Data Structures for a detailed discussion of pointers. These three components will form the basic building blocks of all data structures and their associated algorithms. Of course, it is also essential that we gain an understanding of the use or application of data structures in the development of application programs. Each time we introduce a new data structure we will also show how it can be used to solve domain specific problems.

Finally, we want to deepen our understanding of the `Collection` container classes provided by the Java API. While we will discuss only Java container classes, we should point out that all modern programming languages (Scala, C#, F#, Swift, Go) provide their own versions of these. All of them work in similar ways so that when you understand one library it is easy to work with others. Each time we design and implement a data structure we will also discuss its

equivalent data structure in the Collection classes. This means that you will gain insight into how these class work behind the scenes and better understand how to choose between them when using them as part of your program development tools.

The text begins with a discussion of recursion. It does so because we have not discussed this topic before this and also because many of the algorithms we plan to discuss involve the use of recursive functions. Hence, it is important that you have a basic understanding of how to write recursive functions to solve problems. Chapter 2 outlines two approaches that can be taken to the analysis of algorithms and provides examples of each approach. The chapter on fast sorting provides a detailed study of both QuickSort and MergeSort. Chapter 4 discusses dynamic data structures and provides a detailed study of both ArrayLists and Linkedlists. Generic data structures are introduced in chapter 5 and both generic Stack and Queue classes are developed. Chapter 6 introduces hashing and shows how to implement a hash table. It also provides an implementation of a generic HashSet. In chapter 7 non-linear binary search trees are introduced and a generic binary search tree class is implemented. This chapter also provides a discussion of the TreeSet class. The theory of AVL trees and B-Trees is discussed in chapter 8. Chapter 9 discusses Map classes and chapter 10 provides a simple introduction to Graphs. Finally, chapter 11 discusses external file structures and object serialization.

Bibliography

The literature on data structures is immense and the list below in no way covers a fraction of the works that could be consulted.

Knuth D., *The Art of Computer Programming*, vol 1,2, and 3, Addison-Wesley, 1998.

Cormen, Leiserson, Rivest, Stein, *Introduction to Algorithms*, 3rd edition, MIT Press, 2009.

Mailk D., Nair P, *Data Structures using Java*, Thomson, 2003.

Wirth N. *Algorithms + Data Structures = Programs*, Prenctice-Hall, 1976.

Collins, W. *Data Structures An Object-Oriented Approach*, Addison-Wesley, 1985.

Kruse R., *Data Structures and Program Design*, Prentice-Hall, 1987.

Tremblay J., Cheston G., *Data Structures and Software Development*, Prentice-Hall, 2001.

Harel David, *Algorithms*, Addison-Wesley, 1993.

Wilson Raymond, *An Introduction to Dynamic Data Structures*, McGraw-Hill, 1988.

Goodrich M.T. & Tamassia R, *Data Structures and Algorithms in Java*, Wiley, 2005.

Mullins, Tony, *Object-oriented Programming and the Story of Encapsulation in Java*

Mullins, Tony, *Programming Paradigms with Scala, GCD 2013*

Mullins, Tony, *Programming Paradigms with Java, GCD 2014*

Warburton, Richard, *Java 8 Lambdas*, O'Reilly Press, 2014.
Horstmann, Cay, *Java SE 8 for the Really Impatient*, Addison-Wesley, 2014
Naftalin, M & Wadler, P, *Java Generics and Collections*, O'Reilly Press, 2007.

Bloch, Joshua, *Effective Java* (1st ed) Addison-Wesley, 2001 (available on the Web)
Bloch, Joshua, *Effective Java (2nd ed)* Addison-Wesley, 2008
Ford, Neal, *Functional Thinking*, O'Reilly Press, 2014.

Bloch, Joshua, *Effective Java (2nd ed)* Addison-Wesley, 2008

Odersky, Martin, *Scala by Example*, EPFL Lab, 2010

Odersky M., Spoon L., Venners B., *Programming in Scala (2nd edition*), Artima Press, 2010.

Chapter 1: Recursion

A well-known scientist (some say it was Bertrand Russell) once gave a public lecture on astronomy. He described how the earth orbits around the sun and how the sun, in turn, orbits around the center of a vast collection of stars called our galaxy. At the end of the lecture, a little old lady at the back of the room got up and said: "What you have told us is rubbish. The world is really a flat plate supported on the back of a giant tortoise." The scientist gave a superior smile before replying, "What is the tortoise standing on?" "You're very clever, young man, very clever," said the old lady. "But it's turtles all the way down!"

Stephen Hawking, A Brief History of Time(1988)

What is recursion? In mathematics a definition is said to be recursive if it is defined in terms of itself. In some sense it is paradoxical to say something is defined in terms of itself. It suggests an infinite regress. The very point the little old lady was making! In fact a proper recursive definition never defines something simply in terms of itself, but always in terms of simpler versions of itself. In Mathematics recursion is defined by two properties: a base case and a set of rules that reduce the chain of invocations to the base case. For example, factorial n is defined as:

$$n! = \begin{cases} 1, & if\ n = 0 \\ n * (n-1)!, & \text{if } n > 0 \end{cases}$$

The base case is `n = 0` and the rule that reduces the chain is `n*(n-1)!` Using this definition we can find 5!

```
n! = n*(n-1)!
5! = 5*(5-1)! = 5*4!
   = 5*4*(4-1)! = 5*4*3!
   = 5*4*3*(3-1)! = 5*4*3*2!
   = 5*4*3*2*(2-1)! = 5*4*3*2*1!
   = 5*4*3*2*1*(1-1)! = 5*4*3*2*1*0!
   = 5*4*3*2*1*1
   = 120
```

The case when `n = 0`, base case, terminates the recursion.

In Mathematics recursion is also used to construct infinite sets of values. Consider, the following definition of the natural numbers based on the Peano axioms.

1 is a Natural number (R1)

if *n* is a Natural number , then *n* + 1 is a Natural number (R2)

The set of natural numbers is the smallest set satisfying the previous two properties.
Using this definition we can show that 3 is a member of the set of Natural numbers.

$1 \in N$ (R1)

$\Rightarrow 1+1 \in N$ (R2)

$= 2 \in N$ (addition)

$\Rightarrow 2+1 \in N$ (R2)

$= 3 \in N$(addition)

The power of recursion lies in the possibility of defining a potentially infinite set of objects by a finite piece of text. The equation S = aS defines an infinite sequence of a's

S = aS

= aaS = aaaS = aaaaS = …

When John Backus(1924-2007) and his team designed FORTRAN they had to describe the syntax of the language informally using English. This was very difficult to do and meant that some definitions were ambiguous and incomplete. When Backus joined the Algol team he developed a formal syntactic model that could be used to describe the syntax rules of any language.(*The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference*, IFIP, 1959). This notation was further enhanced by the Danish computer scientist Peter Naur (1928-). The notation is now known as Backus-Naur Form (BNF). This formal system provides a formal notation to specify the syntax of a formal language such as a programming language. The production rules defined in this notation can be used to determine the syntactic correctness of a block of code or a whole program. It can also show that the syntax is unambiguous. The notation consists of two types of terms, called terminal and non-terminal. Rules are written in an equational style where non-terminals appear on the left hand side and terminals or non terminals on the right hand side. Every non-terminal symbol must appear on the left hand side of one syntactic equation. For example, we can define the non-terminal term digit in terms of terminal symbols as follows:

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Read as: *A digit is defined to be a 0 or a 1 or a 2, etc*. The bar (|) denotes Boolean or. The right hand side of a syntactic equation can also contain non-terminals. In fact, a syntactic equation can be defined recursively in terms of a terminal term or the non-terminal term

followed by a terminal or vice versa. An unsigned integer is a sequence of one or more digits. We can define it using syntactic recursion as follows:

<unSignedInt> ::= <digit> | <unSignedInt> <Digit>

Read as: *An unsigned integer is defined to be a digit or an unsigned integer followed by a digit.* There are really two rules here. The terminating condition on the left of the or (|) and a recursive case on the right of it. We could re-write it in terms of two rules. This would give:

<unSignedInt> ::= <digit> (R1)
<unSignedInt> ::= <unSignedInt> <digit> (R2)

We can show, using these two rules, that 256 is a valid unsigned integer.

<unSignedInt> => <unSignedInt> <digit> R2
=> <unSignedInt><digit> 6 R2
=> <digit>56 R1
=> 256

Defining an actual integer value is more complex, however, because we have to take into account possibly a sign, + or -, and also the fact that we do not allow leading zeroes. The number 79 could be written as 00079 but we don't want to allow this format. Also, the number 0 itself is never preceded by a sign. To capture all of these restrictions we need to write a number of different rules. We begin by defining an integer (start symbol) in terms of three non-terminals: zero or signed integer or unsigned integer. Each of these non-terminals is then defined in terms of other non-terminals or terminals. A signed intger begins with a sign (+ or -) followed by an unsigned integer. An unsigned integer is a pDigit (a digit without zero) or a pDigit followed by allDigits. The non-terminal allDigits is defined recursively as a digit or a digit follwed by allDigits. The set of rules is given below:

<integer> ::= <zero> | <signedInt> | <unsignedInt>
<signedInt> ::= <sign><unSignedInt>
<unsignedInt> ::= <pDigit> | <pDigit> <allDigits>
<alldigits> ::= <digit> | <digit> <allDigits>
<pDigit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<digit> ::= <zero> | <pDigit>
<sign> ::= + | -

BNF production rules define the syntax rules for programming languages. From our perspective, they illustrate the power of recursion in formal notations.
Exercise: Show that -2103 is a valid integer and that -023 is not.

Programming with Recursion

Consider a programming language that has no explicit repetitive statement. This hypothetical language would only have input/output statements, assignment and an `if` statement. To construct programs to repeat a task in such a language would necessitate the use of recursive definition. The recursive definition could be implemented as a function, call it `Rec`, where the function body of `Rec` contains an explicit reference to itself. To avoid the possibility of non-terminating recursive calls, each invocation of `Rec` must make *progress towards termination* and its body must contain a *terminating condition*. We illustrate this approach to writing recursive functions with a number of examples.

Example 1

Write a recursive function to compute factorial `n`, `n` ≥ `0`.

The solution is given by implementing the definition of `n!`, as given above.

```
static int fac(int n){
  if(n == 0)
    return 1;
  else
    return(n*fac(n-1));
}
```

The recursion is implemented, in this case, by including a reference to the function name as part of the `return` statement. In fact it uses the right hand side of the definition of n!, i.e. `n*(n-1)!`. In doing so it makes progress toward termination. The terminating condition is: `n == 0`. When this condition is satisfied the function `returns 1` and terminates. This allows all the earlier executions of the `return` statement to terminate in turn. The following example illustrates its semantics.

```
fac(3) = return(3*fac(2))
       = return(3*[return(2*fac(1))])
       = return(3*[return(2*[return(1*fac(0))])])
       = return(3*[return(2*[return(1*1)])])
       = return(3*[return(2*1])
```

```
= return(3*2)
= 6
```

The recursive calls to fac descend until the terminating case, n == 0, is encountered. Each call results in a calculation that involves multiplying n, a local variable, with the invocation of fac(n-1). This may give rise to further invocations or terminate. This pattern is repeated until the termination case is encountered. When this occurs we backtrack because each invocation of fac completes and returns its part of the chain of computations. The final call, fac(0), returns 1. This results in fac(1) completing and returning 1. Then fac(2) completes returning 2 and so on until we reach the top.

The program FacN, listed below, tests the function fac(n).

```
import java.util.Scanner;
class FacN{
  public static void main(String[] args){
    Scanner in = new Scanner(System.in);
    System.out.println("Program to compute factorial n");
    System.out.print("Enter value for n: ");
    int n = in.nextInt(); //assume n>=0
    int x = fac(n);
    System.out.printf("%d! = %d", n,x);
  }
  static int fac(int n){
    if(n == 0)
      return 1;
    else
      return(n*fac(n-1));
  }
}
```

Example 2

Write a recursive procedure that reads a list of integer values from the keyboard, sentinel –1, and prints only the even elements in the list.

A recursive procedure template to read a list from the keyboard is given below. The comment process x denotes the required solution to a given problem.

```
   static void readList(Scanner in){
     int x = in.nextInt();
      if(x != sentinel) {
          // process x
          readList(in);
      }
   }
```

Note

To read a list of n items the following template can be used:

```
   static void readList(int n, Scanner in ){
      if(n != 0){
         int x = in.nextInt();
         // process x
         readList(n-1, in);
      }
   }
```

End note

The program listed below is a solution to the problem given.

```
import java.util.Scanner;
class ProcessList{
  // a recursive procedure to read a list of values
  static final int sentinel = -1;
  public static void main(String[] args){
    Scanner in = new Scanner(System.in);
    System.out.print("Enter list: ");
    readList(in);
 }
 static void readList(Scanner in){
    int x = in.nextInt();
    if(x != sentinel){
      if(x % 2 == 0)
         System.out.printf("%3d",x);
      readList(in);
    }
 }
}
```

Example 3

Write a recursive procedure that reads a list of values, sentinel –1, and prints them in reverse order.

To print the values in reverse order they must first be read into memory. To do this we use the local variables created by each invocation of the procedure to store the values. This is done by calling the procedure `reverseList()` before the output statement. When the *sentinel* is read the terminating condition is satisfied and all the preceding calls to `reverseList()` get to terminate in sequence, thereby printing the values in reverse order. The procedure is:

```
static void reverseList(Scanner in){
  int x = in.nextInt();
  if(x != sentinel){
    reverseList(in);
    System.out.printf("%3d", x);
  }
}
```

As an exercise write a program to test this recursive procedure.

Example 4

Write a program to compute the first `n` terms of the Fibonacci sequence, where the terms of the Fibonacci sequence are defined recursively as:

$$fib(n) = \begin{cases} 1, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ fib(n-1) + fib(n-2), & \text{if } n \geq 2 \end{cases}$$

The function `fib` can be implemented directly using the given definition.

```
static int fib(int n){
  if(n == 0)
    return 1;
  else if(n == 1)
    return 1;
  else
    return(fib(n-1) + fib(n-2));
}
```

To solve the given problem for n terms a loop can be used to generate and print the terms in the sequence as follows:

```
System.out.print("Sequence: ");
 for(int j = 0; j < n; j = j + 1){
      t = fib(j);
      System.out.printf("%4d", t);
  }
```

For n = 10 the output will be:

```
Sequence:   1  1  2  3  5  8 13 21 34 55
```

This solution is correct, however, it is not very efficient because each term is generated independently of all other terms. In fact this recursive solution is computationally very expensive because we calculate many terms multiples of times.(See the Analysis of algorithms chapter for a discussion of this.) From the definition and from observation we see that new terms can be generated from existing terms by adding the two previous terms together. This is a much simpler and more efficient solution. The question is how to implement it and the answer lies in the observation that combining a loop with a recursive function creates the problem. Better to look at the problem as a whole and try to write a recursive function to solve it. Doing so gives an insight into the power of recursion. It doesn't just provide a way to write functions to solve individual tasks, as in previous examples, but a new way of writing general algorithms to solve problems. A heuristic for working with recursion is to see how a problem can be solved assuming that a simpler case of the same problem has already been solved. By reducing the problem to a simpler version of itself a chain of events can take place leading to a base case that terminates the chain. We apply this reasoning to the given problem to illustrate this approach.

We know that the two base cases are: fib(0) = 1, fib(1) = 1, and that fib(2) = fib(0) + fib(1). In general, given two existing Fibonacci terms a, b the next term is a + b. From the base cases the sequence of terms can be generated and printed by recursively invoking the function with the required previous terms until n terms have been generated and printed. The function is:

```
static void fibSequence(int a, int b, int i, int n){ //assumes n >= 1
   if(i == 0) System.out.printf("%4d",a);
   else if(i == 1) System.out.printf("%4d",b);
   else{ // i >= 2
      System.out.printf("%4d",a+b);
```

```
        int temp = a; a = b; b = temp +  b;
    }
    if(i+1 < n) fibSequence(a,b,i+1,n);
 }
```

Notice that each recursive call accumulates the values of successive terms with each recursive invocation. Therefore, there is no backtracking when the recursion terminates. (See tail recursion at the end of this chapter for a further discussion on this.) The terms in this case are printed as they are generated. To generate and store the actual terms of the sequence we could use an array. This is left as an exercise.

The complete program listing is:

```
import java.util.Scanner;
class Fibonacci{
 public static void main(String[] args){
   Scanner in = new Scanner(System.in);
   System.out.println("Fibonacci sequence");
   System.out.print("Enter number of terms: ");
   int n = in.nextInt();
   System.out.print("Sequence: ");
   fibSequence(1,1,0,n);
 }
 static void fibSequence(int a, int b, int i,int n){
        if(i == 0) System.out.printf("%4d",a);
        else if(i == 1) System.out.printf("%4d",b);
        else{
            System.out.printf("%4d",a+b);
            int temp = a; a = b; b = temp+b;
        }
        if(i+1 < n) fibSequence(a,b,i+1,n);
  }
}
```

Output from the program is:

```
Fibonacci sequence
Enter number of terms: 10
Sequence:    1   1   2   3   5   8  13  21  34
55
```

Example 5 Listing files in a Directory

Write a function that lists the names of all files in a folder including its subfolders.

Java provides the package java.io that has a number of classes and associated methods that can be used in programs to access the file system on your disk, create new files, read information stored on file and process it, write new information to a file, etc. The File class contained in this package provides the required methods to solve this problem. (For a detailed discussion of this class see Mullins T. A First Course in Programming with Java, chapter 15.) The methods we require are listed in the table below.

Method	Semantics
File(String pathname)	Constructor that takes a pathname as argument.
boolean isDirectory()	Returns true if the object is a directory; false otherwise.
String getName()	Return the name of the file or directory.
File[] listFiles()	Returns an array of pathnames denoting the files in the current directory

The function displayFiles, listed below, takes the pathname of a directory as argument and recursively iterates over the files in the given directory and all sub-directories contained in it. The method listFiles returns an array of file names in the directory dir. This list may contain actual files and also directory names. If files is not null then the loop iterates over the array. For each file in the list it checks to see if it is a directory. If so, it invokes displayFiles on it and, hence, recursively descends through the sub-directories; otherwise, it prints the name of the file.

```
static void displayFiles(File dir){
   File files[] = dir.listFiles();
   if(files != null){
      for(File f : files)
         if(f.isDirectory()) //recursive call here
            displayFiles(f);
         else //print name of file
            System.out.println(f.getName());
```

```
    }
}
```

A program to test this function is:

```
import java.io.*; import java.util.Scanner;
public class PrintFileNames {
    public static void main(String args[]){
        Scanner in = new Scanner(System.in);
        String path = in.next();
        File d = new File(path);
        listFiles(d);
    }
}
```

Recursion over Sequences

Array processing is a fundamental activity in all programming languages. In the normal course of events we would use a `for` loop or a `while` loop to iterate over a given sequnce of values. In this section we are going to look at an alternative way to do this by showing how to write recursive functions that process elements in an array.

In the case of a sequence there are `N` terms to iterate over. Usually we process all of these terms left-to-right starting at index `0`, the first term. Therefore, the base case or terminating condition is reached when the given `index` value is `N`. The successor of the current term is always `index + 1`. A simple function that displays the values in an array is given below. It takes a reference to the array `f` and an index, `n`, as argument and if `n` is less than the size of the array it prints `f[n]` and then invokes the function again with index equal to `n+1`.

```
static void display(int f[], int n){
    if(n < f.length){
        System.out.print(f[n]+" ");
        display(f,n+1);
    }
}
```

The `sum` function takes an array `f` and an `index` value as arguments. The definition is:

$$sum(f,n) = \begin{cases} 0, & \text{if } n = f.length \\ f[n] + sum(f,n+1), & \text{if } n < f.length \end{cases}$$

A function that implements this definition is:

```
static int sum(int f[],int n){
   if(n == f.length) return 0;
   else return(f[n]+sum(f,n+1));
}
```

This function returns 0 when it terminates. The reason is because the sum over an empty sequence is always defined to be zero. The function uses the stack to store values as it descends or iterates over the sequence. When it terminates it backtracks accumulating the result as it goes. The example below uses an array containing the sequence 2, 4, 7 to illustrate its semantics. The recursive calls iterate over the sequence until the terminating condition is reached when n is 3. This causes the value 0 to be returned and, hence, allows the previous calls to terminate.

```
int f[] ={2,4,7}
sum(f,0)  = return(2 + sum(f,1))
          =  return(2 + return(4 + sum(f,2)))
          = return(2 + return(4 + return(7 + sum(f,3))))
          = return(2 + return(4 + return(7 + 0)))
          = return(2 + return(4 + 7))
          = return(2 + 11)
          = 13
```

A linear `search` function takes three arguments: an array, an index value and the term to search for, `x`. The recursive definition is:

$$search(f,n,x) = \begin{cases} false, & \text{if } n = f.length \\ true, & \text{if } n < f.length \;\&\&\; f[n] = x \\ search(f,n+1,x), & \text{if } n < f.length \;\&\&\; f[n] \neq x \end{cases}$$

A function that implements it is given below. If it terminates without finding `x` it returns `false`. If `f[n]` equals `x` it returns `true`, terminating the search; otherwise, it invokes `search` again incrementing `n` in the process.

```
static boolean search(int f[],int n,int x){
```

```
    if(n == f.length) return false;
    else{
        if(f[n] == x) return true;
        else
            return search(f,n+1,x);
    }
}
```

The following simple program creates an array of integer values and computes the sum of its values and then searches it for a known value and an unknown one.

```
public class ArrayTest {
    public static void main(String args[]){
        int k[] = {2,5,6,7,12,34,6,7,8,9,20,21,2,2};
        int t = sum(k,0);
        System.out.println("Sum = "+t);
        System.out.println(search(k,0,12));
        System.out.println(search(k,0,22));
    }
}
```

Binary Searching

A person picks a number in the range 1..1000000. Your task is to find this number in the quickest time possible by repeatedly asking the same type of question. What would this question be? The answer depends on the strategy you adopt to perform the search. Three approaches suggest themselves:

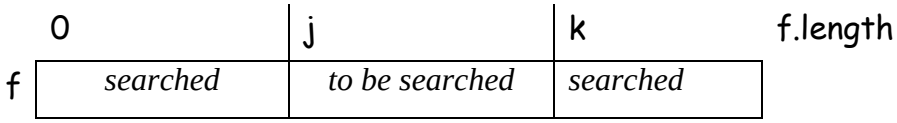
1. You could decide to pick the numbers at random. The question would be: Is it x?
2. You could perform a linear search. The questions would be: Is it 1?, is it 2?, etc.
3. You could decide to exploit the fact that the list of numbers is sorted and try to make strategic moves to reduce the search space. An approach similar to that of searching a telephone directory might work. The type of question to ask might be: Is the number greater than or equal to 500000? If the answer is yes, then the numbers 1..499000 can be eliminated from the search. If the answer is no, then the numbers 500000..1000000 can be eliminated. This type of question can be repeated until the search space is reduced to 1 element. This is the number you are looking for.

Of the three search strategies discussed only 2 and 3 are guaranteed to find the number. The difficulty with the second approach is that it may take a hell of a long time to find the number if it is close to the upper limit. For example, if the number chosen was 1000000 and you were able to ask your question and get a reply in one second it would take, approximately, 11.5 days to find the number! By using the third strategy you will find the number after

approximately 20 questions. This means that you can find it in less than, say, 30 seconds. The third strategy is known as **binary search** because it always halves the size of the search space each time it is applied.

Given a sorted array of integer values and some integer x, write a code fragment to determine if x is an element of the array.

We begin by developing a non-recursive solution based on the algorithm developed by Bottenbruch (a member of the Algol team and the one who gave it the name Algorithmic Language, source Knuth) in 1962. The strategy is to repeatedly start with the middle element and continuously halve the size of the search space until it reaches a segment of size 1. The value in this segment is the outcome of the search. An invariant diagram to describe this strategy is:



initial state: The entire array has to be searched. Therefore, `j = 0;` and `k = f.length`.

guard: The search terminates when the size of the segment *to be searched* is 1, i.e. `j + 1 == k`. Therefore the guard is:

```
j + 1 != k
```

progress: Guaranteed by the `if` statement in the *task to perform* because the upper or lower bound is adjusted.

Task to perform Set `t` equal to the index of the middle element in the current segment and compare `f[t]` with `x`, adjusting the upper or lower bound, as appropriate. The code is:

```
int t;
t = (j + k) / 2;
if(x >= f[t])
   j = t;
else
  k = t;
```

This solution describes the binary search in terms of a loop construct. A static function that defines it's implementation is given below.

```
    static boolean binSearch(int f[],int x){
       int j = 0;
       int k = f.length;
       while(j + 1 != k){
           int i = (j + k)/2;
           if( x >= f[i])
              j = i;
           else
              k = i;
        }
        return(f[j]==x);
    }
```

This is an iterative solution that uses a **while** loop to control the search. We can also write a recursive solution by passing the lower and upper bounds to the search function as arguments. Like the iterative solution above, the recursion terminates when a segment of size 1 is isolated. For all segments of size larger than 1 the middle term is compared with **x** and the appropriate new segment is searched.

```
static boolean binarySearch(int f[], int lb, int ub, int x){
   if(lb+1 == ub) return (f[lb] == x);
   else{
       int mid = (lb+ub)/2;
       if(x >= f[mid])
          return binarySearch(f,mid,ub,x);
       else
          return binarySearch(f,lb,mid,x);
  }
}
```

The following program creates a sorted integer array of random data, asks the user to enter a value to search for, and uses the binary search strategy defined above to find it.

```
import java.util.Scanner;
public class BinarySearch {
  public static void main(String args[]){
      Scanner in = new Scanner(System.in);
```

```
    int f[] = new int[15];
    int t = ((int)(Math.random()*10)) + 1;
    // initialise sorted array
    for(int j = 0; j < f.length; j = j + 1) f[j] = j * t;
    for(int j = 0; j < f.length; j = j + 1)System.out.printf("%4d", f[j]);
    System.out.println();
    System.out.print("Search for: ");
    int x = in.nextInt();
    boolean found = binSearch(f,x);
    if(found)
      System.out.printf("%d is an element of the data\n", x);
    else
      System.out.printf("%d is not an element of the data\n", x);
    //recursive search
    found = binarySearch(f,0,f.length,x);
    if(found)
      System.out.printf("%d is an element of the data", x);
    else
      System.out.printf("%d is not an element of the data", x);
  }
  static boolean binSearch(int f[],int x){
    int j = 0;
    int k = f.length;
    while(j + 1 != k){
      int i = (j + k)/2;
      if( x >= f[i])
        j = i;
      else
        k = i;
     }
   return(f[j]==x);
  }
  static boolean binarySearch(int f[], int lb, int ub, int x){
    if(lb+1 == ub) return (f[lb] == x);
    else{
      int mid = (lb+ub)/2;
      if(x >= f[mid])
          return binarySearch(f,mid,ub,x);
      else
          return binarySearch(f,lb,mid,x);
    }
  }
}
```

Tail Recursion

If you consider the function

```
fac(n : Int) : Int = if(n == 0) 1 else n * fac(n-1)
```

you notice that when it is evaluated the function is repeatedly invoked and this sequence only terminates when the parameter n has a value of 0.

```
fac(3) = 3*fac(2)
       = 3*2*fac(1)
       = 3*2*1*fac(0)
       = 3*2*1*1
       = 6
```

To *remember* where to return the system must store values on the run-time stack. This means that for large values of n your program may run out of stack space, i.e. the run-time stack overflows. If this should happen, then your program will crash. To avoid this you need to write the recursive function so that the last call to the function each time is the recursion itself. A *tail recursive* function is a special case of recursion in which the last instruction executed in the method is the recursive call. This can be done with the use of what is termed an *accumulator* parameter. The idea is to allow the function to accumulate the result as it recursively invokes itself. In this way, the terminating state *knows* the result and can yield or return it.

To illustrate this approach we re-write `fac(n)` so that it uses an *accumulator* parameter. This is difficult to do in Java because it does not allow nested functions. To keep with the original function `fac(n)` we simply write another helper function, `getFac`, that takes three arguments: the accumulator `a`, an integer `n0` and the normal paramter `n`. There is an invariant relation between the first two parameters such that `n0! = a`. When `n0` equals `n`, `a` equals `n!`. This function actually provides the recursive solution by accumulating the result. This means that when it terminates the result can be returned immediately and there is no backtracking. As a consequence, the stack will not overflow and the performance of the function improves because it doesn't have to calculate the solution by backtracking.

```
static int getFac(int a, int n0, int n){
   if(n0 == n) return a;
   else  return getFac(a*(n0+1),n0+1,n);
}
static int fac(int n) {
```

```
    if(n == 0) return 1;
    else
      return getFac(1,1,n);
  }
```

To illustrate this we calculate fac(5) as follows:

fac(5) => getFac(1,1,5) => getFac(2,2,5) => getFac(6,3,5) => getFac(24,4,5) => getFac(120,5,5) => 120

Notice that the stack is not used to store the result term by term as before. Instead each invocation of getFac completes and the result is accumulated in parameter a. Hence, when it terminates the result is just the value of a.

For a second example we consider re-writing the recursive function sum described above that calculates the summ of the value in an array. This function uses the stack to store values as it iterates over the sequence. When backtracking it pops these values and, in doing so, accumulates the sum of the values in the sequence. It is possible to write it using tail recursion. We introduce an additional parameter, total, that is always equal to the summation of the first n values in the array dt. The main function, sum, takes an array as argument returning 0 if it is empty; otherwise it returns the value returned by getSum(0, dt, 0). This helper function implements the recursion by accumulating the final value in its parameter total. The value of total is always equal to the summation of the first j elements in the array dt.

```
static int sum(int dt[]){
  if(dt.length == 0) return 0;
  else
    return getSum(0,dt,0);
}
static int getSum(int j, int dt[], int total){
  if(j == dt.length) return total;
  else
    return getSum(j+1,dt,sum+dt[j]);
}
```

Writing two separate functions to implement efficient tail recursive solutions is not attractive because there is no way to bind them together. One way to do this is to write a class that uses static methods that provide a public interface for the functions. The nested functions are then implemented as private methods of this class. The class TRF given below shows how to do this. It has a private constructor that does nothing. This stops users from creating instances of the class. The static functions provide the public interface and the private methods implement the tail recursive solutions. A simple test program for this class is given.

```
public class TailTest {
   public static void main(String[] args) {
      System.out.println(TRF.fac(3));
      int data[] = {1,2,3,4,5,6,7,8,9};
      System.out.println(TRF.sum(data));
   }
}
class TRF{
   private TRF(){//user cannot construct instance of class}
   static int fac(int n) {
      if(n == 0) return 1;
      else
        return getFac(1,1,n);
   }
   private static int getFac(int a, int n0, int n){
      if(n0 == n) return a;
      else  return getFac(a*(n0+1),n0+1,n);
   }
   static int sum(int dt[]){
      if(dt.length == 0) return 0;
      else
         return getSum(0,dt,0);
   }
   private static int getSum(int j, int dt[], int total){
      if(j == dt.length) return total;
      else
         return getSum(j+1,dt,sum+dt[j]);
   }
}
```

Exercise

Question 1

Write recursive functions to:

1) compute the sum of the elements in a list of 20 integer values;
2) read a list of positive integers, sentinel –1, and print all multiples of 3;
3) read a list of positive integers, sentinel –1, and compute the frequency of the number 5;
4) read a list of positive integers, sentinel –1, and compute the frequency of some given value x.

Write programs to test each of these functions.

Question 2

Write a recursive function that prints a given positive integer value as its binary equivalent. For example, if the value is 11 the output should be 1011.

Question 3

Write recursive functions that compute t(n), n > 0, for each of the following definitions.

1) $t(n) = \begin{cases} 2, & \text{if } n = 1 \\ 2 + t(n-1), & \text{if } n > 1 \end{cases}$

2) $t(n) = \begin{cases} 1, & \text{if } n = 1 \\ n + t(n-1), & \text{if } n > 1 \end{cases}$

3) $t(n) = \begin{cases} 1, & \text{if } n = 1 \\ 1 + 2(n-1) + t(n-1), & \text{if } n > 1 \end{cases}$

Question 4 Euclid's Algorithm

Write a recursive function that implements Euclid's algorithm for finding the greatest common divisor of two positive integer values. The algorithm is defined recursively as follows:

$$\gcd(x,y) = \begin{cases} x, & \text{if } x = y \\ \gcd(x-y,y), & \text{if } x > y \\ \gcd(x,y-x), & \text{if } x < y \end{cases}$$

An alternative definition is given by: $\gcd(x,y) = \begin{cases} x, & if\ y = 0 \\ \gcd(y,x), & if\ x > y \\ \gcd(y,x\%y), & if\ x \le y \end{cases}$

Question 5

Write a recursive function that implements `power(x,n)`, where `x`, `n` are integers and `n ≥ 0`.

$$power(x,n) = \begin{cases} 1, & \text{if } n = 0 \\ power\left(x, \frac{n}{2}\right) * power\left(x, \frac{n}{2}\right), & \text{if } n\%2 = 0 \\ x * power(x, n-1), & \text{if } n\%2 = 1 \end{cases}$$

Write a program to show that $x^m * x^n = x^{m+n}$ and $x^m \div x^n = x^{m-n}$.

Question 6 (Catalan Numbers)

The Catalan numbers form a sequence of natural numbers that can be defined recursively as follows:

$$C_0 = 1, \;\; C_{(n+1)} = \frac{(4 * n + 2) * C_n}{(n+2)}$$

Write a program to:

a) show that $C_{20} = 1767263190$

b) show that a Catalan number can also be calculated by the formula $C_n = \frac{(2n)!}{(n+1)!*n!}$

Question 7

In Mathematics an Arithmetic sequence is given by the terms:

$a, (a+d), \; (a+2d), \; (a+3d), \; .., \; (a+(n-1)d),$

$where\ a = first\ term\ and\ d = common\ difference$

Write a recursive function to compute `t(n)`.

Question 8

A single term of the form cx^n, where c is a constant and n is zero or a positive integer, is called a monomial in `x`. A function that is the sum of a finite number of monomial terms is called a polynomial in `x`. For example, $f(x) = x^3 - 2x^2 + x - 3$ is a polynomial of degree three in `x`. It is degree three because the highest monomial has a power of 3. Design a recursive class hierarchy that represents polynomials of differing degrees. Each polynomial knows how to evaluate itself.

Hint: Begin with a single term or constant, i.e. a monomial of the form cx^0 and use this as a base class. It might have the form,

```
class PolyD0{
    int c;
    public PolyD0(int k){
        c = k;
    }
    public int eval(int x){ // x^0 = 1
        return c;
    }
}
```

PolyD1 extends this class and so on up the hierarchy.

Question 9

Write a recursive algorithm to generate and print the first n terms of the sequence defined by the given recursive function

$$t(n) = \begin{cases} 2, & \text{if } n = 1 \\ 2 + t(n-1), & \text{if } n > 1 \end{cases}$$

Question 10

Write a recursive algorithm to create a set whose values are the first n terms of the sequence defined by the given recursive function

$$t(n) = \begin{cases} 1, & \text{if } n = 1 \\ n + t(n-1), & \text{if } n > 1 \end{cases}$$

Question 11

Given an int array of N values write recursive functions to:

1) count the frequency of a given value x in the array;
2) find the max value in the array, assume N > 0;
3) determine if all values in the array are positive;
4) find the minimum vale and the frequency of its occurrence.

Question 12

You are given a bag of n, where n is an even value, coins containing a single counterfeit coin. Every coin has a weight and counterfeit coins are always lighter or heavier than a valid coin. Write a function to find the counterfeit coin using a binary search strategy.

Hint: think of using a pan balance to compare the weights of two equally sized sets of coins.

Question 13

Given below is a recursive function sum(a,b) that calculates the summation of the numbers:

a + (a + 1) + (a + 2) + .. + b. Write a tail recursive function that provides the same functionality.

```
static int sum(int a, int b){ //assumes a <= b
    if(a == b) return a;
    else
      return (a + sum(a+1,b));
}
```

Question 14

Write tail recursive functions that solve each of the following problems for an array called data;

1) Calculate the product of the values in data;
2) Compute the sum of the even values in data;
3) Find the minimum value in data (note: if data is empty then min should return Integer.MAX_VALUE);
4) Calculate the frequency of x in data;
5) Write a function, filterOdd, that returns an array containing only the odd values in data.
6)

Chapter 2: Analysis of Algorithms

It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one ... We might, for instance, count the number of additions, subtractions, multiplications, divisions, recordings of numbers, ...

Alan Turing(1912-1954)

An essential obligation on all programs is that they be correct. By this we mean that it is essential that every program satisfies its specification and in so doing provides a correct solution to the problem it is intended to solve. A program that gives erroneous results on some occasions cannot be trusted and users would rightly refuse to use it. While correctness is essential it is not the only requirement on programs. It is also very important that programs deliver results in a time that is acceptable to users. In fact, in some cases it is critical that programs meet temporal deadlines. Consider a program that manages the safety bag in a car. On impact it must respond in nanoseconds, otherwise passengers may be injured unnecessarily. A program that monitors weather must produce results in a time frame that makes weather reporting meaningful. There is no point getting today's weather report tomorrow! It is important that we be able to analyze a program and give some guarantees about its performance for different data sets. We need to be able to say how long it will take a program to solve a given problem. We also need to be able to compare performance to help choose between different programs that solve a given problem correctly. Consider, for example, the following functions that compute the sum of the first N natural numbers.

```
static long sumN(long n){
   long s = n*(n+1)/2;
   return s;
}

static long sumN1(long n){
   long s = 0;
   for(int j=0; j < n; j++) s=s+(j+1);
   return s;
}
```

Both solutions are correct, assuming n ≥ 0. But if you had to choose one, which one would it be? To choose, you might consider testing both functions and check their performance. If you

did so, for large values of n, you would find that sumN is much faster than sumN1. In fact it is potentially thousands of times faster. Using a benchmark of n equal to 1000000 and executing it on a standard PC it turned out to be 15700 times faster. A simple inspection of the code shows that sumN calculates the result using a formula and sumN1 does so by iterating over a loop of size n. The iteration involves the execution of at least n instructions and, therefore, requires a greater computation time. A computation time that grows linearly as n increases; whereas, the cost of executing the formula is constant no matter how big n becomes.

There are two ways to measure the quality of the performance of a program. One approach is to benchmark the program against some set of standard inputs. For example, we might benchmark different sorting functions by measuring their performance given the same data set. It is also important that the programs are all tested on the same machine and compiled by the same compiler. By measuring the time taken we could classify the different sorts. Another approach is to analyze the code of the given program and work out a projected time. This would allow the analysis of the potential performance of functions independently of their execution on a given machine. In this chapter, we will develop a method for analyzing the performance of functions based on an analysis of the actual algorithm itself.

Benchmarking

To compare the performance of different algorithms we can try to calculate their actual execution times on an actual machine. One way to do this is to record the start time and the end time and then work out the time elapsed. Because some programs appear to provide their results instantly we need to be able to measure time using very fine granularity. Fortunately, the Java system library provides a way to measure time in nanoseconds (*ns*), where *1ns* equals one billionth of a second. This is an unimaginably small number. To help imagine it we know that *one nanosecond is to one second as one second is to 31.7 years*. To be more precise, we state the following equations: $1sec = 10^3 ms = 10^6 \mu s = 10^9 ns$

The following program provides a benchmark for the two functions given above, where n = 10million.

```
public class AnalysisTest1 {
   public static void main(String args[]) {
      long start = System.nanoTime();
      long k = sumN(10000000);
      long end = System.nanoTime();
      long t1 = end-start;
```

```
        System.out.println("Cost sumN: "+t1);
        start = System.nanoTime();
        k = sumN1(10000000);
        end = System.nanoTime();
        long t2 = end - start;
        System.out.println("Cost sumN1: "+t2);
        System.out.println("Ratio: "+t2/t1);
    }
    static long sumN(long n){
        return n*(n+1)/2;
    }
    static long sumN1(long n){
        long s = 0;
        for(int j=0;j<n;j++)s=s+(j+1);
        return s;
    }
}
```

The results from a sample test show that sumN took *3000ns* and sumN1 took *860800ns*. The first solution is approximately 3000 times faster than the second.

Note that `System.nanoTime()` returns a *64-bit* integer value.

Indices, Logs and Sequences

In this section we revise some basic mathematics that you will require to study the material that we will cover in later sections. The analysis of a certain type of algorithm requires an understanding of logarithms. In mathematics logs and indices form an equivalence relation. This means that indices can be expressed in terms of logs and vice versa. We begin by stating some of the basic laws of indices.

$$a^m * a^n = a^{m+n}$$
$$a^m \div a^n = a^{m-n}$$
$$a^0 = 1$$
$$a^{\frac{1}{n}} = \sqrt[n]{a}$$
$$(a^n)^m = a^{n*m}, \quad \frac{1}{a^n} = a^{-n}$$

These laws can be used to simplify expressions written in index form. The following examples illustrate their use.

$$\frac{2^5 * 2^7}{2^8} = \frac{2^{12}}{2^8} = 2^4 = 1$$

$$\sqrt{\frac{2^5}{2^3}} = \sqrt{2^2} = 2$$

The definition of the log of *x* to the base *b* is given as:

$$log_b x = y \equiv b^y = x$$

Read as: *the log of x to base b equals y is equivalent to b to the power of y equals x.*

A few examples, using common logs, will illustrate its meaning. The log of 100 to the base 10 is 2 because 10^2 is 100, the log of 1000 to base 10 is because 103 is 1000. The log of 165 has to be a number between 2 and 3 and is found by using a logarithm table or by using the log function on your calculater.

$$log_{10}100 = 2 \equiv 10^2 = 100$$
$$log_{10}1000 = 3 \equiv 10^3 = 1000$$
$$log_{10}1 = 0 \equiv 10^0 = 1$$
$$log_{10}165 = 2.2175 \equiv 10^{2.2175} = 165$$

Logs were created by the Scottish mathematician John Napier in 1614 after twenty years of research and hard calculations. They were quickly adopted by astrologers, like Kepler, because they really simplified multiplication and division. To see how, consider multiplying 165 by 1356. Using logs and the rules of indices we simply find the log of each number and then add them using rule 1 for indices above.

$$165 * 1356 = 10^{2.2175} * 10^{3.1323} = 10^{2.2175+3.1323} = 10^{5.3498}$$

Using logs multiplication becomes addition and division becomes subtraction. The laws of logs are:

$$\log(x * y) = \log(x) + \log(y)$$
$$\log(x \div y) = \log(x) - \log(y)$$
$$\log(x^n) = n * \log(x)$$

Example 1

Evaluate each of the following:

$$log_2 16, \qquad log_2 1024, \qquad log_2 \frac{1}{2}$$

Solution

$$log_2 16 = log_2 2^4 = 4 * log_2(2) = 4 * 1 = 4$$
$$log_2 1024 = log_2 2^{10} = 10 * log_2(2) = 10 * 1 = 10$$
$$log_2 \frac{1}{2} = log_2 2^{-1} = -1 * log_2(2) = -1 * 1 = -1$$

or

$$log_2 \frac{1}{2} = log_2 1 - log_2 2 = 0 - 1 = -1$$

Example 2

Show that $\log(n!) = \log(1) + \log(2) + .. + \log(n)$

Solution

$$\log(n!) = \log(n * (n-1) *..* 2 * 1) = \log(1) + \log(2) + .. + \log(n)$$

Example 3

Show $log360 = 3log2 + 2log3 + log5$

$\log(360)$

$$= \log(5 * 8 * 9)$$
$$= \log(5) + \log(8) + \log(9)$$
$$= \log(5) + \log(2^3) + \log(3^2)$$
$$= \log(5) + 3\log(2) + 2\log(3)$$

Changing Base

It is possible to change the base of logs using the law:

$$log_b x = \frac{log_c x}{log_c b}$$

Show that $log_8 128 = \frac{7}{3}$ by converting to log base 2.

Solution

$$log_8 128 = \frac{log_2 128}{log_2 8} = \frac{log_2 2^7}{log_2 2^3} = \frac{7}{3}$$

Logs are relevant in computing because they are used in the analysis of algorithms particularly in relation to what are termed divide and conquer algorithms, e.g. binary searching. The log function grows very slowly and, hence, log scales are often used to compress large scale scientific data. For example, the strength of an earthquake is measured according to the Richter scale. This scale is based on measuring the energy emitted using the

common log of its value. Therefore, a quake of 7.0 emits a 1000 times the energy of a quake measuring 4.0. Using logs we get: $log_{10}x = 7\ and\ log_{10}y = 4$. This gives $10^7 = x\ and\ 10^4 = y$. Therefore, $x = 10^3.10^4 = 1000y$.

Exercise

Question 1

Evaluate each of the following: $log_2 64,\ log_8 512,\ log_{16} 256$.

Question 2

(a) Show that the log to base 2 of 1Mb equals 20.
(b) Show that $log_{10}\sqrt{1000} = 1.5$
(c) Show that $log_{10}(100x) = 2 + log_{10}(x)$
(d) Show that $log_2(1024x^2) = 10 + 2log_2(x)$
(e) Show that $log_b 1 = 0$
(f) Show that $log_b b = 1$

Question 3

Write down an equation in logs that gives the number of digits in any positive integer x.
Hint: $\lfloor log_{10}156 \rfloor = \lfloor 2.1931 \rfloor = 2$

Question 4

(a) Show that $\log(n!) \leq n * \log(n)$
(b) Show that $\frac{n}{2} * \log\left(\frac{n}{2}\right) \leq \log(n!) \leq n * \log(n)$

Sequences and Series

A sequence is an ordered collection of values that are typically related by some given rule.
The sequence *1, 2, 3, 4, ..,n* is given by the rule *2*n-1*, where n is the term number.
Sequences that have a common difference between terms are called Arithmetic Progressions and $t(n) = a + (n-1) * d$, where a equals the first term and d equals the common difference.
A Series is defined to be the sum of the terms in a sequence. The sequence *1,2,3,..,n* becomes the series *1+2+3+..+n* and the sum of the first n terms is given by the equation

$$s(n) = n * (n+1)/2$$

In general, the sum of the first n terms of an arithmetic progression is given by the equation

$$s(n) = \frac{n}{2} * \{2 * a + (n-1) * d\}$$

To calculate the sum for the previous series we get:

$$s(n) = \frac{n}{2} * \{2 * 1 + (n-1) * 1\} = \frac{n}{2} * \{2 + n - 1\} = n * (n+1)/2$$

In Mathematics, we often express the sum of a series using the Greek letter ∑ (sigma) as follows:

$$\sum_{i=1}^{n} term$$

Read as: *the sum from i equals 1 to n of term.*

Examples are:

$$\sum_{i=1}^{10} i = 1 + 2 + 3 + .. + 9 + 10 = 55$$

$$\sum_{i=1}^{n} 1 = 1 + 1 + .. + 1 = n * 1 = n$$

$$\sum_{i=1}^{n} i = 1 + 2 + .. + (n-1) + n = n * (n+1)/2$$

$$\sum_{i=1}^{n} k * i = k * \sum_{i=1}^{n} i = k * n * (n+1)/2$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + .. + n^2 = n * (n+1) * (2 * n + 1)/6$$

Show that $\sum_{i=1}^{n}(n-i) = n * (n+1)/2$

Solution

$$\sum_{i=0}^{n}(n-i) = n + (n-1) + .. + 2 + 1 + 0 = n * (n+1)/2$$

Exercise

Question 1

Find the sum of the first n terms for each of the given series.

(a) *2+4+6+..+2*n*
(b) *1+5+9+..+4*n-3*
(c) *5+11+17+..+6*n-1*

Question 2

Evaluate

$$\sum_{i=1}^{7} i^2$$

$$\sum_{i=3}^{8}(8-i)$$

$$\sum_{i=1}^{2} log_2 i^2$$

Question 3

Show that

$$\sum_{i=1}^{n}(a+b) = \sum_{i=1}^{n} a + \sum_{i=1}^{n} b$$

Question 4

Show that

$$\sum_{i=2}^{n}(i-1) = n*(n-1)/2$$

Question 5

Show that

$$\sum_{j=1}^{w}\left(\sum_{i=k}^{n} 1\right) = (n-k+1)*w$$

Question 6

The series *1 + 3 + 6 + 10 + 20 +..* , is given by

$$\sum_{i=1}^{n}\left(\sum_{j=1}^{i} j\right)$$

Show that

$$\sum_{i=1}^{n}\left(\sum_{j=1}^{i} j\right) = \frac{n}{6}*(n+1)*(n+2)$$

Calculating Running Times on HAL

Machines differ, some are faster than others. Typically, this means that more powerful machines execute instructions quicker than less powerful ones. We measure cost in nanoseconds (*ns*), where *1ns* equals one billionth of a second. A fast machine might execute primitive instructions in *5ns*, whereas a slow machine might take *20ns* to execute the same primitive instruction. Also, different compilers generate different binary instruction sets to implement high level program code. All of this makes the task of calculating the running time of executing programs more complex. To simplify the task we will specify a time cost for executing primitive instructions on an ideal machine called HAL. The real cost, when tested

on an actual machine, will be some multiple of the cost of executing it on HAL. Therefore, if the running time on HAL for a given program is *1000ns*, then the running time on a machine four times faster than HAL will be *250ns*.

We let the cost of executing: `+`, `-`, `*`, `/`, `%`, `<`, `>`, `==`, `>=`, `<=`, `!=`, `=`, be *10ns*. Using these constant values we calculate the cost of the following code segments.

Code	Unit cost (ns)	Total cost (ns)
`x = x + 1`	10,10	20
`x == 2 || x == 7`	10, 10, 10	30

`x = x + y;`
`y = x - y;`
`y = x - y;`
	10, 10
	10, 10
	10, 10
		60

We assign a cost of *50ns* to function invocation and a cost of *10ns* for each parameter, assuming it is a simple pass by value. The cost of executing a `return` statement is also set at *50ns*. This means that the total cost of invoking the following function `sumN` is *150ns.*

```
static long sumN(long n){     50 + 10
   long s = n*(n+1)/2;        10 + 10 + 10 + 10
   return s;                  50
}
```

The time taken to execute an `if` statement of the form `if(b) s1; else s2;` is the cost of `b` plus the max cost of `s1`, `s2`. For example, the cost of executing

```
if(x > 0) x = 1; else x=x*x;
```

is *30ns* because the boolean expression costs *10ns* and the max cost of *(10ns,20ns)* is *20ns*.
Calculating the cost of executing loops is more complex because we have to factor in the number of times the body of the loop is executed. In general, the total cost for a loop is:

totalCost = cost of initialization of variables +
*(n+1) * cost of evaluating guard on loop +*
*n * cost of executing loop body,*
where n equals the number of iterations of the loop.

As an example, we calculate the cost of executing:

```
long s = 0;
for(int j = 0; j < 1000; j = j + 1)
        s = s + (j + 1);
```

The loop is executed 1000 times, therefore, the formula to calculate the total cost is:

*totalCost = (10 + 10) + 1001 * 10 + 1000*(30+20)*
= 20 + 10010 + 50000
= 60030ns

Calculating the cost here is greatly simplified by the fact that the number of iterations is fixed. Usually, we don't know the number of iterations in advance. The original function involved the use of a parameter `n` that allowed the function to work for any given number of terms. We now consider this case. The original function is:

```
static long sumN1(long n){
   long s = 0;
   for(int j=0; j < n; j++) s=s+(j+1);
   return s;
}
```

The cost function is:

totalCost = (50 + 10) - invocation, parameter
+ (10+10) - initialization of variables
*+ (n+1)*10 – evaluation of loop guard*
*+ n *50 - loop body*
+ 50 - return value

This gives

*totalCost = 60 + 20 + (n + 1) * 10 + n * 50 + 50*
*= (140 + 60 * n) ns*

The cost is linearly proportional to the value of n, the bigger the value of n the greater the cost to execute it. For example, if $= 10^6$, the

$$totalCost = 140 + 60 * 10^6 = 140 + 6 * 10^7 \text{ns}$$

Ignoring the constant *140ns*, we get a cost of

$$6 * 10^7 ns = 6 * 10^4 \mu s = 60ms = 0.06 seconds.$$

(Note: $1second = 10^3 ms = 10^6 \mu s = 10^9 ns$)

Nested Loops and Array Indices

The cost of calculating array indices is set to 50ns for both one-dimensional and two-dimensional types and the cost of `new`, for allocating memory, is set to 100ns. This means that the assignment `x = f[10]` has a cost of *10+50*. To illustrate calculating the cost of nested loops we calculate a cost function for the following code fragment.

```
int f[][] = new int[n][n];
int a = 0;
while(a < n){
  int b = 0;
  while(b < n){
    f[a][b] = 1;
    b = b + 1;
   }
   a = a + 1;
}
```

Let `K` equal the total cost of executing the body of the outer loop just once. Then using the formula for calculating the cost of a loop we can write down an equation for the overall cost of the code fragment. The right hand column in the table below lists the costs of the individual components. The formula is:

$$t(n) = 100 + 10 + (n+1) * 10 + n * K$$
$$t(n) = 120 + 10 * n + n * K$$

It remains to calculate `K`. This is a standard loop over a space of size `n`. Therefore,

$K(n) = 10 + (n+1) * 10 + n * (50 + 10 + 20) + n * 10.$

This simplifies to, $K(n) = 20 + 100 * n.$

Substituting this value into the original equation *t(n)* gives:

$$t(n) = 120 + 10 * n + n * (20 + 100 * n)$$
$$t(n) = 120 + 10 * n + 20 * n + 100 * n^2$$
$$t(n) = 120 + 30 * n + 100 * n^2$$

	Cost
int f[][] = new int[n][n];	*100*
int a = 0;	*10*
while(a < n){	*(n+1)*10*
int b = 0;	
while(b < n){	*n*K*
f[a][b] = 1;	
b = b + 1;	
}	
a = a + 1;	
}	

For large values of n this is quite slow. Suppose *n = 10000*.

*t(10000) = 120 + 30 * 10000 + 100 * 10000*10000*

Ignoring *120 + 30 * 10000* gives $t(n) = 10^{10} ns$. This equates to 10 seconds on HAL. For very large values of *n* the computation time would be very long.

Note: If you are testing this in Java on your own machine the max size of any array is `Integer.MAX_VALUE`. Therefore, *n*n* may not exceed this value. However, in practice you may not be able to create arrays this big. It depends on the heap size allocated by the environment.

Optimising Performance

The analysis of the running time of algorithms is useful because it can be used to compare the quality of the coding of a given collection of algorithms that solve some given problem, assuming, of course, that all solutions are deemed correct. By comparing performance we can choose possibly an optimal solution for a given task. In fact, sometimes the search for an optimum solution to a given problem is critical because it may help to save lives. There is a story that Edsger Dijkstra found an optimum algorithm for processing brain scans and, in doing so, provided an instant analysis of a scan. To illustrate this search for optimum solutions we will return to a variation of the problem we started out with at the start of the

chapter. We were asked to choose between two algorithms to compute the sum of the first n integers. One solution had a fixed time and the other a linear time relative to the value n. Now we want to explore solutions to finding the sum of the first n terms in the following series:

1 + 3 + 6 + 10 + 15 + ..

Observe that this series is generated by taking the sum of the sum of the first n terms. That is:

1 + (1 + 2) + (1 + 2 + 3) + (1 + 2 + 3 + 4) + ..

This means that our solutions for the sum of the first n terms can be used to compute the result. Using the function **sumN** we get:

```
static long sumS1(int n){
   long k = 0;
   for(int j = 0; j < n; j++)
      k = k + sumN(j+1);
   return k;
}
```

This solution gives, *t(n) = 140 + 200*n*, given that the running cost of **sumN(n)** is *150ns*

Using the function **sumN1** we get:

```
static long sumS2(int n){
   long k = 0;
   for(int j = 0; j < n; j++)
      k = k + sumN1(j+1);
   return k;
}
```

This solution gives, $t1(n) = 130 + 160 * n + 30 * n^2$, given that the running cost of **sumN1(n)** is *(130 + 30*n) ns*.

Looking at these two running cost functions we can see that even for relatively small values of *n* sumS1 will outperform sumS2. Given a particular value for n, we could work out the actual running times on HAL and give an exact comparison. However, both of these solutions perform in a time relative to the value of n. The question is could we find a solution that has a fixed time? To do so we return to our mathematical work on sequences and series. We can describe this problem mathematically as:

$$\sum_{i=1}^{n}\left(\sum_{j=1}^{i} j\right) = 1 + (1+2) + (1+2+3) + ..$$

By expanding this definition we derive a formula to solve the problem. We know that the sum of *j* from *1* to *i* is *i*(i+1)/2*. That is,

$$\sum_{j=1}^{i} = i * (i+1)/2$$

This means that

$$\sum_{i=1}^{n}\left(\sum_{j=1}^{i} j\right) = \sum_{i=1}^{n} i * \frac{i+1}{2} = \frac{1}{2}\sum_{i=1}^{n}(i^2 + i) = \frac{1}{2}\sum_{i=1}^{n} i^2 + \frac{1}{2}\sum_{i=1}^{n} i \ .$$

Now we can use our formulas for the sum of the first n^2 terms and the sum of the first n terms.

This gives

$$\frac{1}{2}\left[\frac{n*(n+1)*(2n+1)}{6} + \frac{n*(n+1)}{2}\right]$$
$$= n*(n+1)*(n+2)/6$$

Now we have a formula to evaluate any required term in the series. The function sumS3 uses this formula to implement it and it has a running time of *t(n) = 180ns*.

```
static long sumS3(int n){
   long k = n*(n+1)*(n+2)/6;
   return k;
}
```

The running time of sumS3 is **not** relative to the value of *n* and its cost is fixed. If we take, $n = 10^4$, it can be shown that sumS3 is 10,000 times faster than sumS1 and 10 million times faster than sumS2. Hence, we have found an optimum solution for the given problem.

As a second example, we consider the problem of searching an ordered list of values. We know that to search an ordered list we can use linear searching or binary searching. Given that the list is ordered we know that binary searching is the better approach. But, how good is it? To find out we must find its running time function for a given value. Binary search works by continuously discarding half the values in the list until a single value is isolated. The non-recursive solution is:

```
static boolean binSearch(int f[],int x){
    int j = 0; int k = f.length;
    while(j + 1 != k){
      int i = (j + k)/2;
      if( x >= f[i])
        j = i;
      else
        k = i;
    }
    return(f[j]==x);
 }
```

The number of times 2 divides n is given by $log_2 n$. Therefore, the running time of binary search is:

$$t(n) = 200 + 20 * log_2(n + 1) + 100 * log_2 n$$

Ignoring the constant 200 and the additional +1, we get:

$$t(n) = 120 * log_2 n$$

Supposing n = 1Mb we would get a running time of:

$$t(2^{20}) = 120 * log_2(2^{20}) = 120 * 20 * log_2 2 = 120 * 20 * 1 = 2400ns$$

This is *2.4μs = 0.0000024* seconds. Increasing the data size by multiples of 1 Mb has little impact on the overall performance of the search. For example, if n = 1Mb * 1Mb the cost is only increased by a factor of 20.

Exercise

Question 1

Using the statement execution times defined for HAL, calculate the running times for each of the separate code fragments `A`, `B`, `C`. The cost of `Math.random()` is fixed at 200ns per invocation.

```
// A =========================
int x = 100;
int y = x*5+x*x-2;
int z = x + y*y - 56;

// B ==========================
int x = (int)(Math.random()*100);
int y = (int)(Math.random()*100);
if(x>y){
    int temp = x; x=y;y=temp;
}

// C ============================
int k = 0; int s = 0;
while(k < 100){
    s = s + (k+1)*(k+1);
}
```

Question 2

Calculate the running time of the function `term`.

```
static int term(int n){
   int k = n*(n+1)*(2*n+1)/6;
   return k;
}
```

Question 3

Calculate the execution cost of the given block of code on HAL.

```
int f[] = new int[100];
for(int j = 0; j < f.length; j = j + 1)
   f[j] = (int)(Math.random()*100);
int even = 0; int odd = 0;
for(int j = 0; j < f.length; j = j + 1){
   if(f[j] % 2 == 0)
      even = even + 1;
    else
      odd = odd + 1;
}
```

Question 4

Calculate the execution cost for each of the methods in this class.

```
class Q4{
   private boolean f[] = new boolean[20];
   public void negate(int a, int b){
      for(int j = a; j < b; j++)f[j] = !f[j];
   }
   public void swap(int a, int b){
      boolean temp = f[a];
      f[a] = f[b];
      f[b] = temp;
   }
   public void falsifyAll(){
      for(int j = 0; j < f.length; j++) f[j] = false;
   }
}
```

Question 5

Calculate the execution time of the given program.

```
public static void main(String args[]){
   int f[][] = new int[10][5];
   int g[][] = new int[10][5];
   int add[][] = new int[10][5];
   // init both matrices
   for(int i = 0; i < f.length;i++){
      for(int j = 0; j < f[0].length; j++){
         f[i][j] = (int)(Math.random()*10);
         g[i][j] = (int)(Math.random()*10);
       }
   }
   // add corresponding elements on a row by row basis
   for(int i = 0; i < f.length; i++){
```

```
        for(int j = 0;j < f[0].length; j++){
            add[i][j] = f[i][j] + g[i][j];
        }
    }
}
```

Question 6

Find *t(n)* for each of the given functions.

```
static int sum(int f[]){
    int s = 0;
    for(int j = 0; j < f.length; j++)
        s = s + f[j];
    return s;
}

static int freq(int f[], int x){
    int k = 0;
    for(int j = 0; j < f.length; j++){
        if(f[j] == x)
            k = k + 1;
    }
    return k;
}
static boolean unique(int f[]){
    boolean all = true;
    int a = 0;
    while(a < f.length){
        int b = a+1;
        while(b < f.length){
            all = all && f[a] != f[b];
            b = b + 1;
        }
        a = a + 1;
    }
    return all;
}
```

Question 7

Both of the functions given below calculate a^b, for $a \geq 0, b > 0$. Find $t(a^b)$ for both functions and use them to compare their performance. Write a test program to calculate

running times on your machine. Do these times co-relate with the projected times based on your functions.

```
static long power1(int a, int b){
    long z = 1; int k = 0;
    while(k < b){
        z = z * a;
        k = k + 1;
    }
    return z;
}
static long power2(int a, int b){
    int c = 1; int s = b; long z = 1;
    while(b >= c) c=2*c;
    while(c != 1){
        c = c/2; z = z * z;
        if(s >= c){
            s = s - c; z = z * a;
        }
    }
    return z;
}
```

Question 8

Given below are four functions that compute the *quotient* and *remainder* on dividing *a* by *b*, *a >= 0 and b > 0*. The functions intDiv and intDiv1 compute the *quotient* and the other two both compute the *remainder*. Find timing functions for each of the functions and use them to compare their performance. Write a test program to calculate running times on your machine. Do these times co-relate with the projected times based on your functions.

```
static int intDiv(int a, int b){ //assume b > 0, a >=0
    int q = 0; int r = a;
    while(r >= b){
        q = q + 1;
        r = r - b;
    }
    return q;
}
static int intRem(int a, int b){
    //assume b > 0, a >=0
    int q = 0; int r = a;
    while(r >= b){
        q = q + 1;
```

```
            r = r - b;
        }
        return r;
    }
    static int intDiv1(int a, int b){
        int q = 0; int r = a;
        int c = 1;
        while(c*b <= a) c = c * 2;
        while(c > 1){
            c = c/2;
            if(a >= (q + c) * b){
                q = q + c; r = r - c * b;
            }
        }
        return q;
    }
    static int intRem1(int a, int b){
        int q = 0; int r = a;
        int c = 1;
        while(c*b <= a) c = c * 2;
        while(c > 1){
            c = c/2;
            if(a >= (q + c) * b){
                q = q + c; r = r - c * b;
            }
        }
        return r;
    }
```

Big O Notation and the Arithmetic of Infinity

When you read literature on software libraries you often see *big O* mentioned. For example, if you read the Java description of an `ArrayList` you will see the following:

The size, isEmpty, get, set, iterator, and listIterator operations run in ***constant time****. The add operation runs in amortized constant time, that is, adding n elements requires* ***O(n)*** *time. All of the other operations run in linear time (roughly speaking).*

It states that adding elements requires *O(n)* time. What do they mean by this? To explain this we take, as an example, the equation *t(n) = (200 + n)ns* that defines the running time of some function `P`. For large values of *n* the constant *200* may be ignored and we say that: *t(n) is of order n*. Applying this to the function `P`, we say that: `P` *has running time O(n)*. Read as: *P is big-O of n.* Intuitively, this makes sense because in many different contexts we apply this rule without thinking about. For example, suppose you pay 1 million euro for a property and then have to pay twenty thousand euro in fees. You would say the cost of the property is of the order 1 million. The additional twenty thousand may be ignored! As a second example, we consider the case of radioactive clocks that can be used to measure time up to billions of years. Each of these clocks has its own margin of error. If you want to date a rock that is hundreds of millions of years old, you must be satisfied with an error of millions. To date one that is only tens of millions of years old, you must allow for an error of hundreds of thousands of years. But the error in each case is about 1%. The same rule applies when measuring the running times of programs for large values of n. In fact, when we execute multiple functions that solve the same problem we only observe differences in performance when the value of n is very large.

What do you get when you add two infinite numbers together? Why does it make sense to talk of infinite numbers in the context of computing? Well, when you have machines where basic instructions are measured in nanoseconds, then it takes very large, possibly bordering on infinite, sets of instructions to make an impression. Suppose we consider the following functions that define the timings for different programs:

t1(n) = 5 + n, t2(n) = 50 + n, t3(n) = 500 +100 n, t4(n) = 5000 +1000 n.

The impact of the constants is significant for small values of n – say n < 100000. But for very large values of n they have little impact. All of these functions converge, or meet, as *n approaches infinity*. Therefore, all of these functions form a set that are asymptotically dominated by *t(n) = n*.

Formally, we say that:

O($t(n)$) is $O(f(n))$ if $t(n) \leq c * f(n)$ for $c > 0$ and $n \geq n_0$.

Suppose, $t(n) = n^2 + 2 * n + 1$. We want to prove that $t(n)$ is O(n^2).

Proof

We must show that $t(n) \leq c * n^2$ for constant c and $n \geq n_0$.

$$n^2 + 2 * n + 1 \leq n^2 + 2 * n^2 + n^2$$
$$n^2 + 2 * n + 1 \leq 4 * n^2$$

Therefore, c = 4 and n_0 = 1 because, $1^2 + 1 + 1 = 4 * 1^2$.

The values c and n_0 are not unique. We could use c = 2 and n_0=3.

In general, given a polynomial of degree *k* let *big-O* equal $O(n^k)$. For example, if $t(n) = n^4 + 3 * n^2 + 5$, then $t(n)$ is O(n^4).

Using the approach above to show that one function is an order of another function is cumbersome and not necessarily easy to work with. An alternative approach is to use the theory of limits directly.

We can prove that f(n) is O(g(n)) by showing that:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$$

For example, if f(n) = 3*n we prove that f(n) is O(n) as follows:

$$\lim_{n\to\infty} \frac{3*n}{n} = \lim_{n\to\infty} 3 = 3 < \infty.$$

Prove that $f(n) = a + bn + cn^2, where\ a, b, c\ are\ constants$, is $O(n^2)$
Proof:

Let $g(n) = n^2$

$$\lim_{n\to\infty} \frac{a+bn+cn^2}{n^2} = \lim_{n\to\infty} \frac{a}{n^2} + \lim_{n\to\infty} \frac{bn}{n^2} + \lim_{n\to\infty} \frac{cn^2}{n^2} = 0 + 0 + c = c < \infty.$$

Therefore, $O\big(f(n)\big) = O(n^2)$

Note: $\lim_{n\to\infty} \frac{a}{n}$ =0 , for constanct a.

Laws of *big-O*

We state the following laws of *big-O*, without proof.

1. **Summation**

   *O(1)+O(1)+..+O(1) = k * O(1) = O(1)*, where *k* is a constant.
   *O(n) + O(n)+..+O(n) = k * O(n) = O(n)*, where *k* is a constant
   O(n) + O(m) = max(O(n), O(m))
   e.g. $O(n^3) + O(n^5) = O(n^5)$

2. **Product**

   *O(n) * O(n) =* $O(n^2)$
   *n * O(n) =* $O(n^2)$
   *O(n) * O(m) = O(n * m)*
   *O(k * f(n)) = k * O(f(n)) = O(f(n))*, where *k* is a constant
   $O(n^a) * O(n^b) = O(n^{a+b})$

The *big-O* sets of order functions form a chain of sub-sets as follows:

$$O(1) \ll O(\log_2 n) \ll O(n) \ll O(n * \log_2 n) \ll O(n^2) \ll O(n^k, k > 2) \ll O(a^n) \ll (n!)$$

Read as: O(1) *performs better than* $O(\log_2 n)$, etc. Observe the transitive law at play here. If a *performs better than* b and b *performs better than* c, then a *performs better than* c.

Applying Big-Oh to the Analysis of Programs

Using the big-O notation we can give a classification of algorithms in terms of their projected performance. The table below lists the main classifications in order of increasing cost. The optimum solution is O(1) but this is not a realistic goal for problems defined over a data set of size n. Each row lists the big-Oh classification, the running time function and the standard name associated with it. For example, row two lists: $\mathbf{O(\log_2 n)}$ that has a running cost of $t(n) = a + b * log_2 n$, where *a*, *b* are constants. The name given to programs falling in to this category is **logarithmic time**. Programs fitting into the first four categories are incredibly fast even for very large values of n. These solutions are, in general, regarded as optimal. Programs that are $O(n^2)$ are slow for very large values of n. Programs whose time complexity is greater than this can perform very slowly for even small values of n. For example, a program that tries to generate all the possible permutations of an array containing unique values of size n may take many years to complete even for very small values of n. There are n! permutations in a collection of n unique values. For a value as small as 20 this

would give rise to a number of permutations greater than, 10^{19}. Even on the fastest computer possible this would require more than a thousand years of computation time.

big-Oh	t(n)	Name
O(1)	t(n) = k, a constant	**constant time**
$\mathbf{O(\log_2 n)}$	$t(n) = a + b * \log_2 n$	**logarithmic time**
O(n)	t(n) = a + b * n	**linear time**
$\boldsymbol{O(n * \log_2 n)}$	$t(n) = a + b * n * \log_2 n$	**n log n**
$\boldsymbol{O(n^2)}$	$t(n) = a + b * n^2$	**quadratic time**
$\boldsymbol{O(n^3)}$	$t(n) = a + b * n^3$	**cubic time**
$\boldsymbol{O(2^n)}$	$t(n) = a + b * 2^n$	**exponential time**
O(n!)	*t(n) = a + b * n!*	**factorial time**

We can prove the equivalence between the *big-O* representation and the running time cost function using the laws for *big-O* given above. For example, given $t(n) = a + b * n * log_2 n$

$$\begin{aligned} O\big(t(n)\big) &= O(a + b * n * log_2 n) \\ &= O(a) + O(b * n * log_2 n) \\ &= a * O(1) + b * O(n * log_2 n) \\ &= O(1) + O(n * log_2 n) \\ &= O(n * log_2 n) \end{aligned}$$

Using big-O notation we can classify all our constant times as *O(1)*. Therefore, `+`, `-`, `*`, `/`, `%`, `==`, `!=`, `<`, `>`, `>=`, `<=`, function calls, `return` statement and array indexing all fall into the category *O(1)*. The `if(b) s1; else s2;` statement, for `s1`,`s2` consisting of only primitive statements, is also *O(1)*. Loops over a space of size n will be O(n), loops that can reduce the space by half for each iteration will be $O(log_2 n)$, nested loops will, in general, be $O(n^2)$.

As a first example, we evaluate the time complexity of the following program in terms of *big-O*.

```
public static void main(String args[]){
   int f[][] = new int[N][N];
   int g[][] = new int[N][N];
   int add[][] = new int[N][N];
```

O(a)

// init both matrices
for(int i = 0; i < f.length;i++){
for(int j = 0; j < f[0].length; j++){
f[i][j] = (int)(Math.random()*10);
g[i][j] = (int)(Math.random()*10);
}
}
	*O(f.length) * O(b*f[0].length)*

// add corresponding elements on a row by row basis
for(int i = 0; i < f.length; i++){
for(int j = 0;j < f[0].length; j++){
add[i][j] = f[i][j] + g[i][j];
}
}
}
	*O(f.length) * O(c*f[0].length)*

$O(t(n)) = O(a) + O(f.length) * O(b*f[0].length) + O(f.length) * O(c*f[0].length)$

$= O(a) + O(n)*O(b*n) + O(n)*O(c*n)$, where $n = f.length = f[0].length$

$= O(1) + O(n^2) + O(n^2)$

$= O(n^2)$

As a second example, we show that the function doublingUpTo(int n) is $O(log_2 n)$.

static void doublingUpTo(int n){ int p = 0; int c = 1; int s = n;	*O(1)*
while(n >= c) c = c * 2;	$O(log_2 n)$

while(c != 1){
c = c / 2; p = p * 2;
if(s >= c){
p = p + 1; s = s - c;
}
	$O(a + b * log_2 n)$

```
  }
  // p == n
}
```

$$O(\text{doublingUpTo(int n)}) = 0(1) + O(log_2 n) + O(a + b * log_2 n)$$
$$= 0(1) + O(log_2 n) + O(a) + O(b * log_2 n)$$
$$= 0(1) + O(log_2 n) + O(log_2 n)$$
$$= O(log_2 n)$$

This notation provides a way to classify algorithms that can be used for comparison purposes. A program that has *O(k)* will perform better than a program of *O(n)*, in general. We can also extend this idea to analyzing *best case*, *average case* and *worse case* scenarios for given algorithm. Consider, the linear search algorithm given below that searches for some x in an array of size n.

```
static boolean search(int f[], int x){
   boolean found = false;
   int j = 0;
   while(j < f.length && ! found){
      if(f[j] == x) found = true;
      else j++;
   }
   return found;
}
```

The worst case scenario occurs when x is not present. In this case it is *O(n)*, where *n = f.length*. In the event that x occurs in the array it will terminate the search immediately it finds it. This leads to the question what is the best case? Clearly, if x is the first element the result is *O(1)*. However, it might be the second, third, fourth, etc. Suppose the possibility of searching for each element in the array is equally likely. Then the cost of searching for any one of them on average is $(\sum_{i=0}^{n}(i))$/n. This is *(n*(n+1)/2)/n* which is *(n+1)/2*. This is *O(n)* also.

Exercise

Question 1

Using the definition *O(t(n))* is *O(f(n))* if $t(n) \leq c * f(n)$ for $c > 0$ and $n \geq n_0$,

(a) Show that *O(t(n)) = 70+50*n* is *O(n);*
(b) Show that *O*(t*(n)) = 20+45*n+5*n^2* is O(n^2);
(c) Show that *O(t(n)) = (0.001*n)* is *O(n).*

Question 2

Using the laws for *big-O* prove that:

(a) *O(70+50*n) = O(n);*
(b) *O(20+45*n+5*n^2)* = O(n^2);
(c) *O((0.001*n)) = O(n)*;
(d) $O(a + b * n^2 + c * n^3) = O(n^3)$;
(e) $O(a * n^3 + b * 2^n) = O(2^n)$.

Question 3

We can prove that f(n) is O(g(n)) by showing that $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$. For example, if f(n) = 3*n we prove that f(n) is O(n) as follows: $\lim_{n\to\infty} \frac{3*n}{n} = \lim_{n\to\infty} 3 = 3 < \infty$.

Using this method prove that $f(n) = 10 * n^2$ is $O(n^2)$ and k(n) = 5*n is $O(n^2)$.

Question 4

O(n) is $O(n^2)$ but $O(n^2)$ is **not** *O(n).* We can prove this by showing that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$. Prove that $O(n^2)$ is **not** *O(n).*

Question 5

Calculate *big-O* for each of the given functions.

```
static int sum(int f[]){
    int s = 0;
    for(int j = 0; j < f.length; j++)
       s = s + f[j];
    return s;
 }
```

```
static int freq(int f[], int x){
    int k = 0;
    for(int j = 0; j < f.length; j++){
        if(f[j] == x)
            k = k + 1;
     }
    return k;
}
static boolean unique(int f[]){
    boolean all = true;
    int a = 0;
    while(a < f.length){
        int b = a+1;
        while(b < f.length){
            all = all && f[a] != f[b];
            b = b + 1;
        }
        a = a + 1;
    }
    return all;
}
```

Question 6

Both of the functions given below calculate a^b, for $a \geq 0, b > 0$. Find *big-O* for both functions and use them to compare their performance. Write a test program to calculate running times on your machine. Do these times co-relate with the projected times based on your functions.

```
static long power1(int a, int b){
    long z = 1; int k = 0;
    while(k < b){
        z = z * a;
```

```
        k = k + 1;
    }
    return z;
}
static long power2(int a, int b){
    int c = 1; int s = b; long z = 1;
    while(b >= c) c=2*c;
    while(c != 1){
        c = c/2; z = z * z;
        if(s >= c){
            s = s - c; z = z * a;
        }
    }
    return z;
}
```

Question 8

Given below are four functions that compute the *quotient* and *remainder* on dividing *a* by *b*, *a >= 0 and b > 0*. The functions intDiv and intDiv1 compute the *quotient* and the other two both compute the *remainder*. Find *big-O* for each of the functions and use them to compare their performance.

```
static int intDiv(int a, int b){ //assume b > 0, a >=0
    int q = 0; int r = a;
    while(r >= b){
        q = q + 1;
        r = r - b;
    }
    return q;
}
static int intRem(int a, int b){
    //assume b > 0, a >=0
    int q = 0; int r = a;
```

```
    while(r >= b){
       q = q + 1;
       r = r - b;
    }
    return r;
}
static int intDiv1(int a, int b){
 int q = 0; int r = a;
 int c = 1;
 while(c*b <= a) c = c * 2;
 while(c > 1){
    c = c/2;
    if(a >= (q + c) * b){
       q = q + c; r = r - c * b;
    }
 }
 return q;
}
static int intRem1(int a, int b){
 int q = 0; int r = a;
 int c = 1;
 while(c*b <= a) c = c * 2;
 while(c > 1){
    c = c/2;
    if(a >= (q + c) * b){
       q = q + c; r = r - c * b;
    }
 }
 return r;
}
```

Question 9

Show that the function div2Count has a best case performance of *O(1)* and a worst case performance of $O(log_2 n)$.

```
static int div2Count(int n){
  int count = 0;
  while(n % 2 == 0){
    count++;
       n = n/2;
   }
   return count;
}
```

Summary of Rules for Calculating Performance Costs

Calculating Running Times on HAL	
Statement	**Unit cost (ns)**
-, *, /, %, ^, <, >, ==, >=, <=, !=, =	*10ns*
Function invocation	*50ns*
Argument passing	*10ns* per argument
Return	*50ns*
if(b) s1; else s2	the cost of b plus the max cost of s1, s2
for, while loops	*totalCost = cost of initialization of variables* + *(n+1) * cost of evaluating guard on loop* + *n * cost of executing loop body,* *where n equals the number of iterations of the loop.*
New	*100ns*
Calculating array indices	*50ns*
Math.random()	*100ns*

Laws of *big-O*

The laws of *big-O are*:

1. **Summation**

   *O(1)+O(1)+..+O(1) = k * O(1) = O(1)*, where *k* is a constant.
   *O(n) + O(n)+..+O(n) = k * O(n) = O(n)*, where *k* is a constant
   O(n) + O(m) = max(O(n), O(m))
   e.g. $O(n^3) + O(n^5) = O(n^5)$

2. **Product**

   *O(n) * O(n) =* $O(n^2)$
   *n * O(n) =* $O(n^2)$
   *O(n) * O(m) = O(n * m)*
   *O(k * f(n)) = k * O(f(n)) = O(f(n))*, where *k* is a constant
   $O(n^a) * O(n^b) = O(n^{a+b})$

The *big-O* sets of order functions form a chain of sub-sets as follows:

$$O(1) \ll O(\log_2 n) \ll O(n) \ll O(n * \log_2 n) \ll O(n^2) \ll O(n^k, k > 2) \ll O(a^n) \ll (n!)$$

Chapter 3: Fast Sorting

Sorting data is important for many reasons, not least, because it is possible to search a sorted sequence optimally in $O(log_2 n)$ using binary search. The question is what is the optimal solution possible to sort n values? It is clearly at least as bad as *O(n)*, since it has to look at each item at least once. If we analyse the selection sort algorithm given below we see that it is actually $O(n^2)$, where n = f.length, because it involves a nested loop over a space of size f.length. Formally, it is: $\sum_{i=0}^{n}(n-i)$. Selection sort is not impacted in any way by the state of the data, therefore, its worst case and best case performances are both of $O(n^2)$.

```
static void selectionSort(int f[]){
  for(int i = 0; i < f.length; i = i + 1){
    int k = i;
    for(int j = i + 1; j < f.length; j = j + 1){
      if(f[j] < f[k])
        k = j;
    }
    // k = index of smallest element in f[i .. f.length-1]
    // swap f[k] with f[i]
    int temp = f[i]; f[i] = f[k]; f[k] = temp;
  }
}
```

We would like to know if it is possible to write a faster sorting algorithm than selection sort. To answer this question we turn, once again, to mathematics. At the heart of any sorting algorithm is the comparison of two values. This is a *binary* comparison: it produces one of two possible answers. If we take a list of unique values then a sorted list is a particular permutation of the original list. In a list of n unique values there are n! possible permutations. This means that the cost of sorting n unique values equates to the number of comparisons required to pick out one specific permutation out of the n! possible cases. Each comparison, because it is binary, reduces the possibilities by half. Therefore, the cost of sorting n unique values is a number of comparisons, c, such that: $2^c \geq n!$ From this inequality, we find a value for c as follows:

$$2^c \geq n!$$

$$\equiv log_2 2^c \geq log_2 n!$$

$$\equiv c * log_2 2 \geq log_2(n * (n-1) *..* 1$$

$$\equiv c \geq log_2 n + log_2(n-1) + .. + log_2 1$$

The right hand side is of the order n*logn

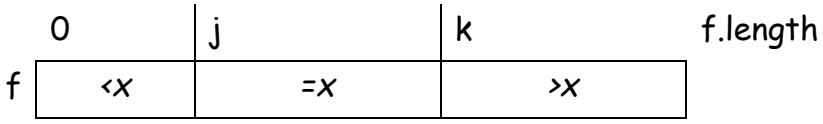
Therefore, $c \geq O(n * log_2 n)$. This means that the optimal performance of any comparison based sorting algorithm is at best $O(n * log_2 n)$.

In this lecture, we examine two comparison based algorithms: QuickSort and MergeSort, that provide $O(n * log_2 n)$ solutions. Before discussing these algorithms we begin by solving two relevant problems. The first is called the Problem of the Dutch National Flag in honour of Edsger Dijkstra (1930-2002). The original problem, as given in *A Discipline of Programming* (Prentice-Hall,1976), asks that a bucket of pebbles, where each pebble is either: red, white or blue, be arranged so that all the pebbles form the colours of the Dutch National flag. We state it here as:

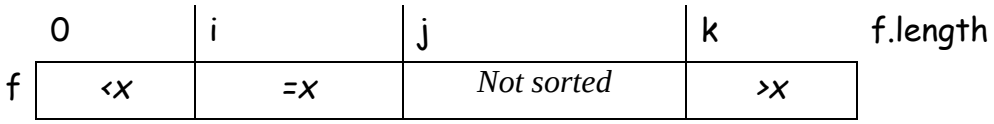
Given an array of size n and some value x, partition it into three segments such that the values in the leftmost segment are less than x, those in the middle segment are equal to x and those in the rightmost segment are greater than x .The solution must be O(n).

Solution

The final state of the sorted array is:



The invariant diagram below introduces a fourth segment denoting the values that are not sorted yet.



Initial state:	At the start the entire array of values has to be sorted. This means the segments containing *<x, =x and >x* are empty. This is established by setting i = 0, j = 0 and k = f.length.
Guard:	The sorting is finished when j == k. Therefore, the guard is: j != k
progress:	Guaranteed by the if statement in the *task to perform*

because j is incremented or k is decremented.

Task to perform

Focus on f[j].

```
if(f[j] < x)
    // swap f[j] with f[i] and
    // increment both i and j
else (f[j] == x)
    // increment j
else // f[j] > x
  // swap f[j] with f[k-1]
  // and decrement k
}
```

This gives:

```
int i, j, k;
i = 0; j = 0; k = f.length;
while(j != k){
   if(f[j] == x)
     j = j + 1;
   else if(f[j] < x){ //swap f[j] with f[i]
     int temp = f[j];
     f[j] = f[i]; f[i] = temp;
     j = j + 1; i = i + 1;
   }
   else{ // swap f[j] with f[k-1]
     int temp = f[j];
     f[j] = f[k-1]; f[k-1] = temp;
     k = k - 1;
   }
}
```

The solution is *O(n)* because the sort is completed in a single iteration of the loop. We will return to this problem again when we discuss QuickSort below.

The second problem will be relevant for our discussion of MergeSort.

Given two sequences sorted in the same order write an algorithm to merge them in a third sequence that preserves the ordering. The merge must be O(n).

Solution

The function mergeLists takes two integer arrays as arguments and returns an integer array. It assumes the lists are sorted in the same order. The array temp is used to store the new sorted merged list. The main loop iterates over both sequences copying individual elements to temp using an if statement to preserve the ordering of the values. When the loop terminates i == a.length or j == b.length . Hence, it only remains to tag on any remaining elements from either a or b.

```
static int[] mergeLists(int a[], int b[]){
  int temp[] = new int[a.length+b.length];
  int i = 0; int j = 0; int k = 0;
  while(i < a.length && j < b.length){
     if(a[i] <= b[j]){
        temp[k] = a[i]; i++;
     }
     else{
        temp[k] = b[j]; j++;
     }
     k++;
  }
  // i == a.length or j == b.length
  //tag on remaining elements
  while(i < a.length){
     temp[k] = a[i]; i++; k++;
  }
  while(j < b.length){
     temp[k] = b[j]; j++;k++;
  }
  return temp;
}
```

This function is *O(a.length+b.length)*. A simple program to test this function is:

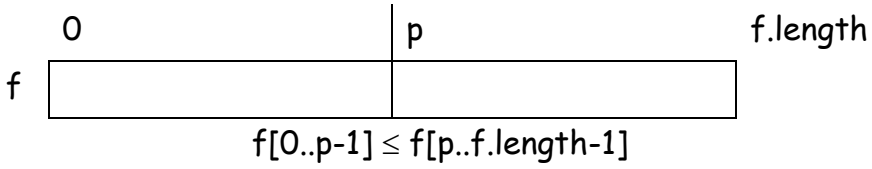
```
  public static void main(String args[]){
     int f[] = {1,3,5,7,9,11,13};
     int g[] = {2,4,6,8,10,12};
     int c[] = mergeLists(f,g);
     for(int x : c) System.out.printf("%3d",x);
     System.out.println();
   }
```

QuickSort

Implement the *Quicksort* algorithm to sort an array of n, n ≥0, integer values.

Solution

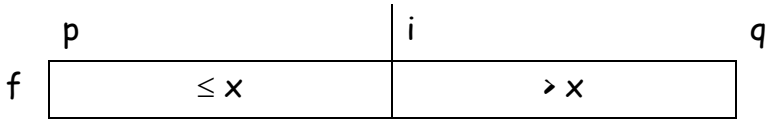
The strategy adopted in *quicksort* is to shuffle the elements in the array into two non- empty partitions such that all the elements in the left partition are less than or equal to those in the right partition. The elements in each partition are not necessarily sorted. The following diagram describes this situation.



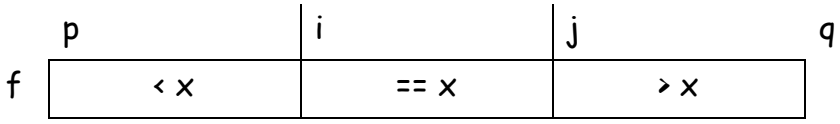
It now only remains to quickSort f[0..p-1] and then quickSort f[p..f.length-1], and the whole array is sorted. An outline solution is:

```
static void quickSort(int f[], int p, int q){
  if(q-p <= 1)
     ; //skip
  else{
    // re-arrange f[p..q-1] into two non-empty partitions
    // f[p..i-1] and f[i..q-1] (p ≤ i ≤ q) and each element in
    // f[p..i-1] ≤ each element in f[i..q-1]
    quickSort(f,p,i);
    quickSort(f,i,q);
  }
}
```

The task re-arrange f[p..q-1] ... must now be solved. The strategy is to choose some x in the segment and partition the elements about x, i.e. establish



But if x happens to be the largest element in the array, then the right hand partition would be empty and the condition that both segments be non-empty is violated. To avoid this situation we choose to arrange the array into three segments as follows:



The final two calls now become:

```
quickSort(f,p,i);

quickSort(f,j,q);
```

The problem of partitioning the array into three segments is the problem of the Dutch national flag discussed above.

```
int x;
int i, j, k;
// let x = middle element in f[p..q-1]
x = f[(p+q)/2];
i = p; j = p; k = q;
while(j != k){
  if(f[j] == x)
     j = j + 1;
  else if(f[j] < x){
    //swap f[j] with f[i]
  }
  else{ // f[j] > x
     // swap f[j] with f[k-1]
   }
 }
```

The derivation of *Quicksort* is now complete.

The code is:

```
static void quickSort(int f[], int p, int q){
 if(q-p <= 1)
    ; //skip
 else{
   int x; int i, j, k;
   // let x = middle element in f[p..q-1]
   x = f[(p+q)/2];
   i = p; j = p; k = q;
   while(j != k){
    if(f[j] == x)
      j = j + 1;
    else if(f[j] < x){ //swap f[j] with f[i]
      int temp = f[j];
      f[j] = f[i]; f[i] = temp;
      j = j + 1; i = i + 1;
    }
    else{ // f[j] > x
      // swap f[j] with f[k-1]
      int temp = f[j];
      f[j] = f[k-1]; f[k-1] = temp;
      k = k - 1;
    }
   }
   quickSort(f,p,i);
   quickSort(f,j,q);
 }
}
```

Analysis

Analysis of the algorithm suggests that it is at best $O(n * log_2 n)$ and at worst $O(n^2)$. The reason for this variation in performance is due in the main on the choice of value given x

during the partition phase. In the solution given above, we chose to give x the middle value. To my knowledge, this choice was suggested by Nicklaus Wirth. In the original solution given by Tony Hoare the value chosen was the first value in the partition. Depending on the given ordering of the data in the original array this could greatly impact on performance. If the original data sequence is sorted then the performance is $O(n^2)$ because the size of the leftmost partition is typically only 1. See the second table listed below. The first row of the table listed below shows the initial state of an array and the last row lists the sorted sequence. The intermediate rows show permutations quicksort generated in mapping the initial state to its final sorted state. The two shaded regions show the initial divide brought about by partitioning the sequence about the middle value 6. This has a depth of 4. Similarly, the right hand segment gives rise to a depth of 4. There are an initial 13 values and the $\lceil log_2 13 \rceil = 4$ (Read as the ceiling of log 13 to base 2 is 4). This gives an $O(n * log_2 n)$ solution.

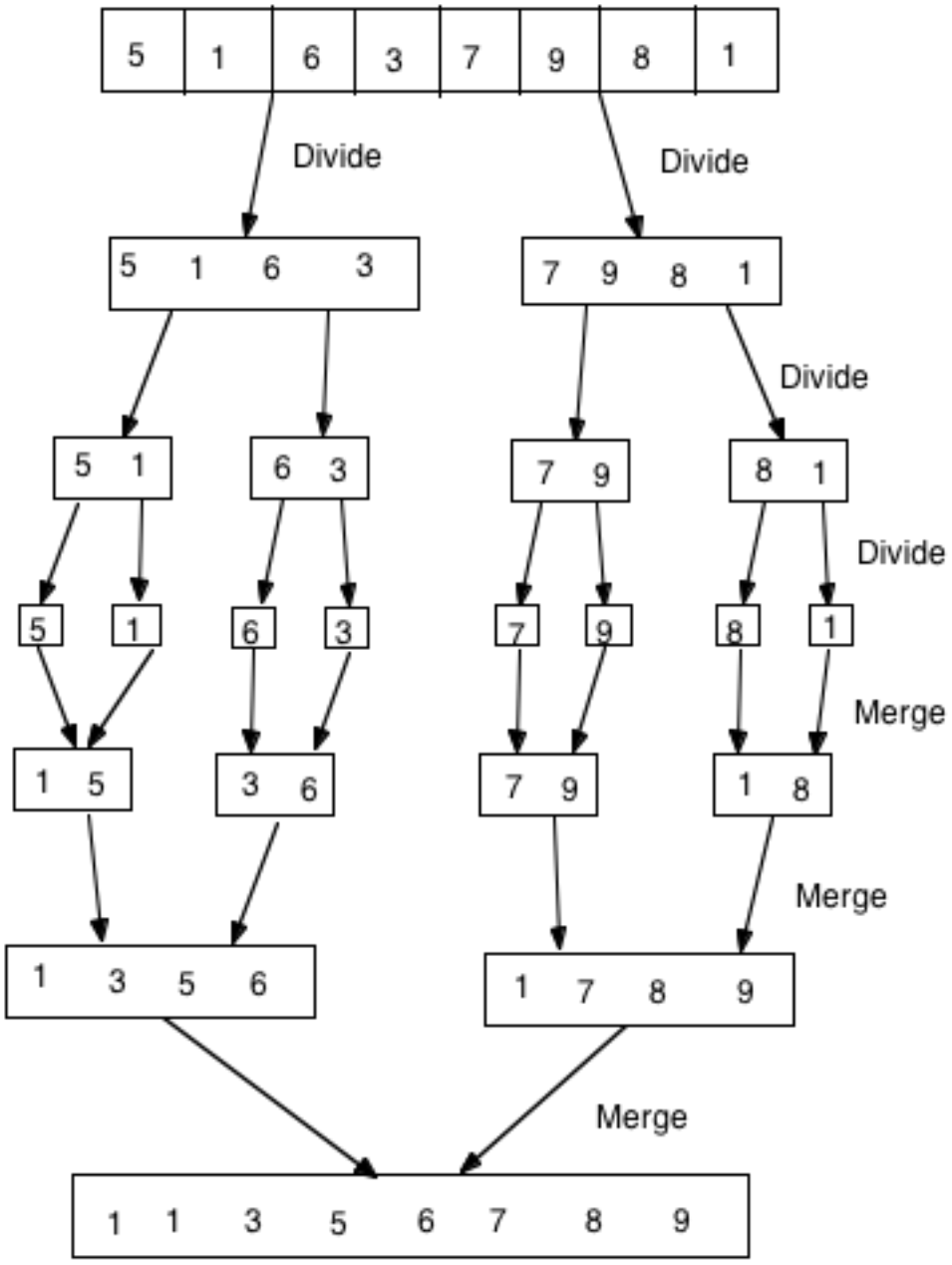
3	2	1	9	5	7	**6**	11	4	1	12	13	8
Left							Right					
x = 6												
3	2	**1**	1	5	4	6	11	7	12	13	8	9
x = 1												
1	1	2	5	4	3	6	11	7	12	13	8	9
x = 4												
1	1	2	3	**4**	5	6	11	7	12	13	8	9
x = 3												
1	1	2	**3**	4	5	6	11	7	12	**13**	8	9
							x = 13					
1	1	2	3	4	5	6	11	7	**12**	8	9	13
							x = 12					
1	1	2	3	4	5	6	11	7	**8**	9	12	13
							x = 8					
1	1	2	3	4	5	6	7	8	9	**11**	12	13
							x = 11					
1	1	2	3	4	5	6	7	8	9	11	12	13

Choosing a sorted list of values and always taking x equal to the first element in the segment to be partitioned gave the following table. Notice that it is $O(n^2)$.

1	2	3	4	5	6	7	8	9	10	11	12	13	
x = 1													
1	**3**	4	5	6	7	8	9	10	11	12	13	2	
x = 3													
1	2	3	**6**	7	8	9	10	11	12	13	5	4	
x= 6													
1	2	3	**4**	5	6	10	11	12	12	9	8	7	
x = 4													
1	2	3	4	5	6	**10**	11	12	13	9	8	7	
x = 10													
1	2	3	4	5	6	**7**	8	9	10	13	12	11	
x=7													
1	2	3	4	5	6	7	8	**9**	10	13	12	11	
x=9													
1	2	3	4	5	6	7	8	9	10	**13**	12	11	
x = 13													
1	2	3	4	5	6	7	8	9	10	**12**	11	13	
x = 12													
1	2	3	4	5	6	7	8	9	10	11	**12**	13	

MergeSort (John vonNeumann,1945)

The merge sort closely follows the divide and conquer paradigm by recursively dividing a given sequence of values until the size of the sequence to be sorted is 1. It then merges the sequences. The division involves no comparisons. The key operation then becomes the merging of sorted subsequences to form the sorted sequence. The following tree like diagram illustrates the process of dividing and merging a given data set.


5 1 6 3 7 9 8 1
Divide
Divide
5 1 6 3
7 9 8 1
Divide
5 1
6 3
7 9
8 1
Divide
5
1
6
3
7
9
8
1
Merge
1 5
3 6
7 9
1 8
Merge
1 3 5 6
1 7 8 9
Merge
1 1 3 5 6 7 8 9

The division continues until segments of size 1 are reached. Then the merging begins from the bottom up until the complete data set is re-assembled. The recursive function is:

```
static void mergeSort(int f[], int lb, int ub){
      //termination reached when a segment of size 1 reached - lb+1 = ub
      if(lb+1 < ub){
         int mid = (lb+ub)/2;
         mergeSort(f,lb,mid);
         mergeSort(f,mid,ub);
         merge(f,lb,mid,ub);
      }
   }
```

The value mid forms the upper bound on the first invocation of mergeSort and the lower bound in the second. Each division gives rise to a single merge that takes all three arguments as parameters. The function merge uses a temporary array to store the merging data. Its size is always r - p, where p represents the lower bound and r the upper bound. When merging is complete the merged data is copied back to the original array f.

```
   static void merge(int f[], int p, int q, int r){
      //p<=q<=r
      int i = p; int j = q;
      //use temp array to store merged sub-sequence
      int temp[] = new int[r-p]; int t = 0;
      while(i < q && j < r){
         if(f[i] <= f[j]){
            temp[t]=f[i];i++;t++;
         }
         else{
            temp[t] = f[j]; j++; t++;
         }
      }
      //tag on remaining sequence
```

```
        while(i < q){
            temp[t]=f[i];i++;t++;
        }
        while(j < r){
            temp[t] = f[j]; j++; t++;
        }
        //copy temp back to f
        i = p; t = 0;
        while(t < temp.length){
            f[i] = temp[t];
            i++; t++;
        }
    }
}
```

A simple program to test this sorting algorithm is given below.

```
public class MergeSortTest {
    public static void main(String args[]){
        int d[] = {2,5,1,2,3,6,7,8,4,2,5,3,7,9,1};
        mergeSort(d,0,d.length);
        for(int x : d) System.out.print(x+" ");
        System.out.println();
    }
```

Analysis

Merge sort always performs in $O(n * log_2 n)$ because the divide component is $O(log_2 n)$ and merging is *O(n)*. This would suggest that it should outperform QuickSort on average. However, this is not the case because there is a hidden overhead. Every time we merge we require a new temporary data array to store the values. This is then copied back to the original array. This is an expensive overhead and for large data sets QuickSort outperforms it.

There are other sorting algorithms not based on comparison testing that will perform at best in O(n). An example is *BucketSort*. But these sorting algorithms require more complex data structures to implement and we will defer discussion of them to later.

Sorting Collections

The `Collection` classes in Java provide a sorting function that takes a list as argument and sorts the data in the list based on the natural ordering of the elements defined by the `Comparable` interface. For example, in the case of `String` the ordering is based on the alphabetical order of characters.

The `sort` operation, it states, uses a slightly optimized *merge sort* algorithm that is fast and stable:

- **Fast**: It is guaranteed to run in n*log(n) time and runs substantially faster on nearly sorted lists. Empirical tests showed it to be as fast as a highly optimized quicksort. A quicksort is generally considered to be faster than a merge sort but isn't stable and doesn't guarantee n*log(n) performance.
- **Stable**: It doesn't reorder equal elements. This is important if you sort the same list repeatedly on different attributes. If a user of a mail program sorts the inbox by mailing date and then sorts it by sender, the user naturally expects that the now-contiguous list of messages from a given sender will (still) be sorted by mailing date. This is guaranteed only if the second sort was stable.

The program listed below demonstrates the use of `Collections.sort`. Two methods are provided: the standard `sort` orders elements based on the natural ordering provided by the `compareTo` method; the second sort takes an additional `Comparator` method that orders elements based on its `compare` method. For example, `Collections.sort(acLst, Account.nameCompare())` orders elements based on account name. The method `nameCompare()` is implemented as a static method on the class.
Exercise: Write a program to test this sorting algorithm against our Quicksort solution above. Use as large a list of numbers as your system will allow.

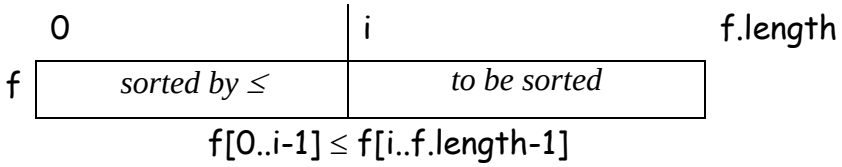
```
import java.util.*;
public class CollectionSort{
   public static void main(String args[]){
      ArrayList<Integer> lst = new ArrayList<>();
      for(int j = 0; j < 1000;j++) lst.add((int)(Math.random()*1000));
      Collections.sort(lst);
```

```
        ArrayList<Account> acLst = new ArrayList<>(Arrays.asList(
                                                    new Account("001","Joe",200.0),
                                                    new Account("003","Mary",200.0),
                                                    new Account("002","Pat",400.0),
                                                    new Account("006","Noel",300.0),
                                                    new Account("004","Sheila",200.0),
                                                   new Account("005","Joe",100.0)));
        Collections.sort(acLst);
        System.out.println(acLst);
        Collections.sort(acLst,Account.nameCompare());
        System.out.println(acLst);
    }
}
final class Account implements Comparable<Account>{
    private String number;
    private String name;
    private double balance;
    Account(String n, String nm, double b){
        number = n; name = nm; balance = b;
    }
    String name(){return name;}
    String number(){return number;}
    double balance(){return balance;}
    void setBalance(double b){balance = b;}
    public String toString(){return number+":"+name+":"+balance;}
    public boolean equals(Object ob){
        if(!(ob instanceof Account)) return false;
        Account ac = (Account)ob;
        return ac.number().equals(number);
    }
    public int hashCode(){
        return number.hashCode()*31;
    }
    public int compareTo(Account ac){
        if(ac == null) return -1;
        return number.compareTo(ac.number);
    }
    public static Comparator<Account> nameCompare(){
        return new Comparator<Account>(){
                public int compare(Account ac1, Account ac2){
                    return ac1.name.compareTo(ac2.name);
                }
        };
    }
}
```

Exercise

Question 1

Sort an array of values using the *Bubblesort* algorithm. The strategy is to always start at the rightmost position in the array and drag the current smallest value leftward, dropping it if a new smallest element is found. The invariant diagram for the outer loop is the same as for selection sort. The difference between the two algorithms is in the strategy used to find the next smallest element. The invariant is:

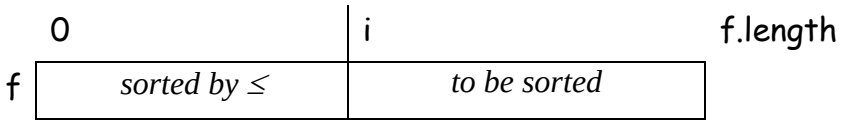


Show that *Bubblesort* has a *best case* and *worse case* performance of $O(n^2)$.

Suppose you have two different arrays of data. In one case, the data is randomly ordered and you use *Bubblesort* to sort it in ascending order. In the second case, the data is initially ordered in descending order and again *Bubblesort* is used to re-order it so that the final state is in ascending order. Reason would suggest that the first case should outperform the second. In practice, however, the opposite might be the case. Can you give a reason why this might be so?

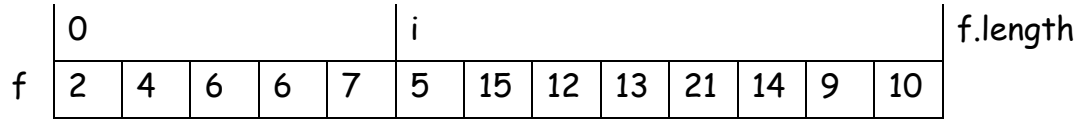
Question 2

Sort an array of values using the *Insertion sort* algorithm. The invariant for the outer loop is:

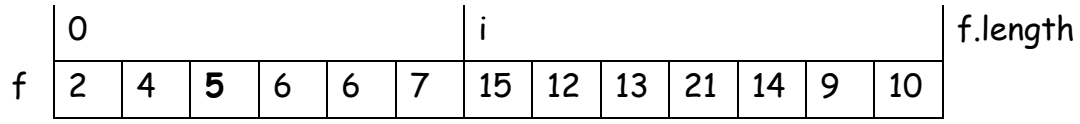


The strategy to make progress is to swap f[i] with f[i-1] while f[i] < f[i-1]

In the diagram below f[0..i-1] is sorted.



After applying the insertion sort strategy the state of the array is:



What is the difference between the invariant in the selection sort algorithm and the invariant used here?

Show that insertion sort has a *best case* performance of $O(n)$ and a *worst case* performance of $O(n^2)$.

Question 3

Write a program that bench marks *QuickSort* and *MergeSort*. You should use different data sets and record the performance of each algorithm on the same sets under the same conditions. Write a short report that compares the two algorithms based your results. It is important to include a discussion of the initial states of the data sets in your report.

Question 4

Write a program that bench marks *QuickSort* and *InsertionSort*. You should different data sets and record the performance of each algorithm on the same sets under the same conditions. Write a short report that compares the two algorithms based your results. It is important to include a discussion of the initial states of the data sets in your report.

Question 5

In the *Quicksort* algorithm we chose to give × the middle value of the segment during the partition phase. An alternative is to assign × the average of the first element, the middle element and the last element in the segment. Re-write it using this approach and write a program to test its performance. Compare this with the performance of the approach discussed above.

Question 6

MergeSort continuously divides the data into segments until segments of size 1 are reached. It then begins the merging phase. This is the expensive part. Improvements could be made if we could reduce the cost of merging. It turns out that *InsertionSort* is very efficient for small data sequences (say sequences of 32 or 64 values) where the data is partially ordered in the correct order and the displacement is small.

The idea is to combine *MergeSort* and *InsertionSort* to reduce the over head of merging. To do this we terminate the *MergeSort* division when segments of some given size are reached, use *InsertionSort* to sort the segments and then do the merging as before.

Your task is to implement this solution.

Question 7

Quicksort can be optimized somewhat by terminating the recursion when the segments are small (say less than 32, instead of less than 2 as in the basic algorithm). This gives a final array consisting of multiple sub-sequences all of which are partially sorted segments. We can then complete the sort very efficiently by performing an insertion sort on the entire array. This last insertion sort is *O(n).*

Chapter 4: Testing Data Structures

One view of classes is that they provide services to clients. Each class encapsulates a single data object or a collection of data objects and its public methods provide a service to a user of the class. It is important that the service is correct and meets the expected requirements of the user. Hence, it is crucial that we be able to test public methods of our classes for correctness of behaviour. One way to do this is to write a test program with a `main` method that constructs an instance of a class and sequentially tests each of its public methods. Usually we do this by printing the results of our tests on the screen. This works fine but it is hit and miss and it would be nice if we could write actual test programs that would automate the testing and let us know if the test results were correct or not. To take this approach we can use a testing framework to ensure that our class under testing meets its contract. Designing test routines helps the design of classes because it forces us to think about *how an object is used* and not just about *how it is implemented.* Focusing on the use of an object greatly improves our confidence on its ability to meet its stated requirements and on the correctness of our code. One of the most widely used testing frameworks is *JUnit* mainly because it is easy to use. Hence, in this section we will introduce JUnit as a frameowrk for testing our classes.

Note that this chapter repeats some of the material contained in the section Testing Classes with Junit in *Object-Oriented Programming and the Story of Encapsulation, Mullins Tony, 2016,* pp 63-69.

Downloading JUnit

JUnit can be found on www.junit.org. Download two *jar* files called *junit* and *hamcrest-core*. You should either put these two files in your class path or else copy them to the directory where you store your java files.

To compile your program from the command line on a Mac machine use:

```
javac -cp .:junit-4.12.jar <filename>.java
```

To run it:

```
java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar org.junit.runner.JUnitCore <className>
```

On Windows use the following two lines. To compile use:

```
javac -cp .;junit-4.12.jar <fileName>.java
```

To run it:

```
java -cp .;junit-4.12.jar:hamcrest-core-1.3.jar org.junit.runner.JUnitCore <className>
```

In both cases above we assume the current directory on your terminal is where your files are stored. It is also possible to set up JUnit with Eclipse. Please see the documentation on how to do this, if you want to use Eclipse.

Creating a Test Class

In this section we are going to create test class examples to show you how to set up simple tests for the classes that you write. To set up simple tests we will use the following template:

```
import org.junit.Test;
import static org.junit.Assert.*;
public class <ClassName>Test{
 @Test
 public void test1() {
   …
   assertEquals(a,b);
 }
 @Test
 public void test2() {
   …
   assertEquals(a,b);
 }
 @Test
 public void test…(){ … }
}
```

You can create as many tests as you like in this class.
To illustrate we write a test class for our Book class above. There are three observer methods and a toString method. In our class we will write a test for each of these using a single book instance. The class is called BookTest and it creates a single instance of a book and then provides four public test methods that test the current state of each of the attributes and the value of the string returned by the toString method. In each case it uses the method assertEquals that tests the given strings for equality. This method is a public method contained in the class Assert that we import from the junit jar file. For now we will use methods from this class only for testing.

```
import org.junit.Test;
import static org.junit.Assert.*;
public class BookTest{
 Book b = new Book("Happy Days","Beckett","fiction");
 @Test
 public void testTitle() {
   assertEquals(b.title(),"Happy Days");
 }
 @Test
 public void testAuthor() {
   assertEquals(b.author(),"Beckett");
```

```
  }
  @Test
  public void testCategory() {
    assertEquals(b.category(),"fiction");
  }
  @Test
  public void testtoString() {
    assertEquals(b.toString(),"Happy Days Beckett fiction");
  }
}
```

To compile and execute this class use:

```
javac -cp .:junit-4.12.jar BookTest.java
java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar org.junit.runner.JUnitCore BookTest
```

When it executes the output is:

```
JUnit version 4.12
....
Time: 0.004

OK (4 tests)
```

Note: It prints 1 dot per test and finally outputs the time taken and simple message indicating that the 4 tests were OK.

In the event that a test fails it prints a message telling you the number of tests that passed and for each test that fials it prints a separate message indicating what it expected and what it found. For example, suppose we modify `testCategory` by replaing the lower case 'f' with an upper case

```
  public void testCategory() {
    assertEquals(b.category(),"Fiction");
  }
```

the ouput is:

```
org.junit.ComparisonFailure: expected:<[f]iction> but was:<[F]iction>
…
FAILURES!!!
Tests run: 4,  Failures: 1
```

Our second example will build a test class that contains 8 tests and illustrates how to use some of the methods in the `Assert` class. (A summary of the main methods in this class is given in the next section below. The complete list of methods are available on the JUnit

website in the JavaDocs section.) The method assertEquals takes two argument values and tests them for equality. There are a number of different versions of this method that support different types of argument values. Three initial examples are given below: testExpression() tests two int values for equality; testBoolExp() tests two boolean expressions for equality and testIntegerExp() tests two Integer instances for equality. The method testDouble() tests two decimal numbers for equality to within a given positive delta value. In the given case below the difference is exactly 0.001 and, hence, is at the max difference allowed. There are also a number of different versions of the method assertArrayEquals() that can be used to test two arrays for equality. Because arrays are ordered data structures we say that: *two arrays are equal if and only if they have the same dimension and all corresponding elements in the given sequences are equal*. Two examples are given that test two arrays of primitive int values and two arrays of Integer values. Finally, we include an example of testing two Sets for equality and one of testing two ArrayLists that are not equal.

```
import org.junit.Test;
import static org.junit.Assert.*;
public class GeneralTest{
   @Test
   public void testExpression(){
      int a = 8; int b = 6; int c = 100;
      assertEquals(a*a+b*b,c);
   }
   @Test
   public void testBoolExp(){
      boolean a = true; boolean b = false;
      assertEquals(a && !b, (b || true) && a);
   }
   @Test
   public void testIntegerExp(){
      Integer k = 21; Integer p = 21;
      assertEquals(k,p);
   }

   @Test
   public void testdouble(){
      double a =9.64;
      assertEquals(a,9.641,0.001);
   }
   @Test
   public void testArray(){
      int f1[] = {2,5,3,9,6};
      int f2[] = {2,5,3,9,6};
```

```
        assertArrayEquals(f1,f2);
    }
    @Test
    public void testArrayIntegerArray(){
        Integer f1[] = new Integer[10];
        for(int j = 0; j < 10; j++) f1[j] = j+1;
        Integer f2[] = new Integer[10];
        for(int j = 0; j < 10; j++) f2[j] = j+1;
        assertArrayEquals(f1,f2);
    }
    @Test
    public void testSet(){
     Set<Integer> s1 = new HashSet<>(Arrays.asList(2,4,7,9,1));
     Set<Integer> s2 = new HashSet<>(Arrays.asList(1,4,2,7,9));
     assertEquals(s1,s2);
    }
    @Test
    public void testArrayList(){
     List<Integer> l1 = new ArrayList<>(Arrays.asList(1,2,4,7,9));
     List<Integer> l2 = new ArrayList<>(Arrays.asList(1,2,5,7,9));
     assertNotEquals(l1,l2);
    }
}
```

Our third example illustrates testing an object collection for a user-defined class. The class LibBook given below encapsulates an immutable class that has a title and an author. The class has an all argument constructor, two observer methods, a toString method, and also equals and hashCode methods. It also implements the Comparable interface. Two LibBook instances are equal if they have the same title and author. The hashCode is based on the product of the hashcodes for title and author times 31. LibBooks are sorted by author and within author by title. Including these methods in the definition of the class is necessary to make it Collection compliant.

```
public final class LibBook implements Comparable<LibBook>{
  private final String title;
  private final String author;
  LibBook(String t, String a){ title = t; author = a;}
  public String title(){return title;}
  public String author(){return author;}
  public String toString(){ return title+" by "+author;}
  public boolean equals(Object ob){
    if(!(ob instanceof LibBook)) return false;
    LibBook b = (LibBook)ob;
```

```
    return(title.equals(b.title) && author.equals(b.author));
  }

  public int compareTo(LibBook b){
    if(b == null) return -1;
    if(this.equals(b)) return 0;
    if(!author.equals(b.author))
      return author.compareTo(b.author);
    else
      return title.compareTo(b.title);
  }
  public int hashCode(){ return 31 * title.hashCode() * author.hashCode();}
}
```

We test this class by creating a class that encapsulates a collection of LibBook objects. The class LibBookTest, listed below, tests a TreeSet of LibBook instances. The constructor creates a small set of LibBook instances and this set is then used for testing purposes. Four tests are provided. The first two tests are quite simple and should be self explanatory. The third test extracts the list of authors in lib and checks that they are correct. The fourth test creates a sorted list of titles by Joyce and then checks that these are indeed those contained in the collection.

```
import org.junit.Test;
import static org.junit.Assert.*;
import java.util.*;
public class LibBookTest{
    TreeSet<LibBook> lib = new TreeSet<>();
    public LibBookTest(){
       lib.addAll(Arrays.asList(new LibBook("Ulysses","Joyce"),
                                new LibBook("Dubliners","Joyce"),
                                new LibBook("Murphy","Beckett"),
                                new LibBook("Finnegan's Wake","Joyce"),
                                new LibBook("Waiting for Godot","Beckett")
                                ));
    }
    @Test
    public void testContains(){
      assertEquals(lib.contains(new LibBook("Ulysses","Joyce")),true);
    }
   @Test
    public void testNoDuplicates(){
      int size = lib.size();
      lib.add(new LibBook("Murphy","Beckett"));
      assertEquals(lib.size(),size);
```

```
  }
  @Test
  public void testAuthor(){
   Set<String> ls = new HashSet<>();
   for(LibBook lb : lib) ls.add(lb.author());
   assertEquals(ls,new HashSet<String>(Arrays.asList("Joyce","Beckett")));
  }
  @Test
  public void testGetBooksBy(){
   //Test titles by Joyce
   List<String> titles = new ArrayList<>(
                              Arrays.asList("Ulysses","Dubliners","Finnegan's Wake")
                            );
   Collections.sort(titles); // necessary because TreeSet sorted
   List<String> bks = new ArrayList<>();
   for(LibBook lb : lib)
     if(lb.author().equals("Joyce")) bks.add(lb.title());
   assertEquals(bks, titles);
  }
}
```

Assert Statement

Java also has an **assert** statement that can be used for testing. This statement takes a boolean expression as argument and throws an **Exception** in your program if the expression is **false**. This statement can be inserted at any time in your code and provides a simple way to test program state at different points in the execution of a program. The program listed below gives 5 examples of using the assert statement to test the state of the relationship between instances of LibBook classes. The first shows that **lb1** equals **lb2**, the second that **lb2** is not equal to **lb3**, the third that **lb3** precedes **lb1** in the ordering, the fourth tests the **contains** method and the fifth the **size** method.

Note: When executing this program you must use the *enable assertion* checking flag:

```
javac –ea AssertTest
```

```
import java.util.*;
public class AssertTest{
 public static void main(String args[]){
  LibBook lb1 = new LibBook("Dubliners","Joyce");
  LibBook lb2 = new LibBook("Dubliners","Joyce");
  LibBook lb3 = new LibBook("Murphy","Beckett");
  assert(lb1.equals(lb2));
  assert(!lb2.equals(lb3));
```

```
  assert(lb3.compareTo(lb1) <= 0);

  List<LibBook> lib = new ArrayList<>();
  lib.addAll(Arrays.asList(new LibBook("Ulysses","Joyce"),
                           new LibBook("Dubliners","Joyce"),
                           new LibBook("Murphy","Beckett")
                           ));

  assert(lib.contains(new LibBook("Dubliners","Joyce")));
  assert(lib.size() == 3);
  System.out.println("5 Tests correct");
 }
}
```

Methods for Assert Class

Method Name	Semantics
assertArrayEquals(boolean[] expecteds, boolean[] actuals)	Asserts that two boolean arrays are equal.
assertArrayEquals(byte[] expecteds, byte[] actuals)	Asserts that two byte arrays are equal.
assertArrayEquals(char[] expecteds, char[] actuals)	Asserts that two char arrays are equal.
assertArrayEquals(double[] expecteds, double[] actuals, double delta)	Asserts that two double arrays are equal.
assertArrayEquals(float[] expecteds, float[] actuals, float delta)	Asserts that two float arrays are equal.
assertArrayEquals(int[] expecteds, int[] actuals)	Asserts that two int arrays are equal.
assertArrayEquals(long[] expecteds, long[] actuals)	Asserts that two long arrays are equal.
assertArrayEquals(Object[] expecteds, Object[] actuals)	Asserts that two object arrays are equal.
assertEquals(double expected, double actual, double delta)	Asserts that two doubles are equal to within a positive delta.
assertEquals(float expected, float actual, float delta)	Asserts that two floats are equal to within a positive delta.
assertEquals(long expected, long actual)	Asserts that two longs are equal.
assertEquals(Object expected, Object actual)	Asserts that two objects are equal.
assertNotEquals(float unexpected, float actual, float delta)	Asserts that two floats are **not** equal to within a positive delta.
assertNotEquals(long unexpected, long actual)	Asserts that two longs are **not** equals.

Chapter 5: Dynamic Data Structures

When you declare an array in a program you must declare its size as part of the declaration. This means that you must know the max size of the array at the point of declaration. Once the memory for the array is allocated at run time it cannot be modified. This restriction is difficult for programmers because they do not always know in advance what the max size of the data set is going to be. If you have studied the Collection classes in *Java* you will have noticed that it is possible to declare an ArrayList without specifying a maximum size for the data set. This allows programmers to work with unbounded arrays that grow on demand. In this lecture we are going to study how to write unbounded data structures that grow on demand. Of course, it should be pointed out that there is always a limit based on the amount of memory in the machine and, in the case of Java the fact that arrays are limited by the amount of memory allocated for the heap by the JVM. In theory the max size is Integer.MAX_VALUE but in practice this is usually not possible. There are two ways to implement dynamic linear data structures. One approach is to use dynamic arrays and the other is to use pointers to construct chains of nodes. In the second case, nodes are created on demand and linked together using memory addresses, i.e. pointers or reference variables.

Dynamic Arrays

We begin by implementing the interface List defined below. For this example we will only use integers as data to store in our data structure. Of course, we need to be able to write data structures so that they can be used to store many different types of data. To make this possible we also need to make them generic. However, this is an unnecessary distraction at this stage because we want to focus exclusively on the issues surrounding the dynamic allocation of memory for data structures. In the next chapter on Stack and Queue we will address the issue of genericity.

We want to implement the following interface.

```
interface IntList{
    public int size(); //return current size of the array
    public void add(int x); //append x to the array
    public int get(int ind); // return value at index ind
    public void set(int ind, int x); // assign x to array at index ind
    public boolean contains(int x); // search the array for x
    public boolean remove(int x); // remove leftmost occurrence of x
                                  // from array, if present
```

```
    public String toString(); // return string representation of data
}
```

The class IntArray implements the IntList interface. The implementation is for type int only and the default constructor creates a new array of size increment. This is chosen as the base size for an array and when additional space is required the size of the array will be increased in increments. There are two constructors, the default one and one that takes a size n as argument.

```
public IntArray(){
    data = new int[increment];
    size = 0;
}
public IntArray(int n){
    data = new int[n]; size = 0;
}
```

Method add takes a value x as argument and automatically increases the size of the array, if necessary. To do so, it creates a new array and then copies the old array data to the new one incrementing the attribute size accordingly.

```
public void add(int x){
    if(size < data.length){
        data[size] = x; size++;
    }
    else{ //increase size of array
        int f[] = new int[data.length+increment];
       // copy data
        for(int j = 0; j < data.length; j++) f[j] = data[j];
        f[size] = x;
        // change data reference
        data = f; size++;
    }
}
```

The method remove finds the leftmost occurrence of x, if any, and removes it by copying all elements to the right of it one position left. The attribute size is then decremented. It returns a boolean indicating success or failure.

```
public boolean remove(int x){ //remove leftmost occurrence only
    boolean found = false;
```

```
        int j = 0;
        while(j < size && !found){
            if(data[j] == x) found = true;
            else j++;
        }
        if(found){
            while(j < length){ data[j] = data[j+1]; j++;}
            size--;
            return true;
        }
        else
            return false;
    }
```

The complete class is given below. The method `contains` performs a standard linear search, `set` assigns its argument `x` to the array at index `ind`, hence, overwriting the existing value. It assumes that `ind` is a value in the range `0..size-1`. The method `get` takes an index as argument and returns the value at position `ind`, again, it assumes `ind` is a value in the range `0..size-1`. Finally, the method `toString` returns a formatted `String` representation of the data in the array.

```
public class IntArray implements IntList{
    private int data[];
    private int size;
    private int increment = 20;
    public IntArray(){
        data = new int[increment];
        size = 0;
    }
    public IntArray(int n){
        data = new int[n]; size = 0;
    }
    public int size(){return size;}
    public void add(int x){
        if(size < data.length){
            data[size] = x; size++;
        }
        else{ //increase size of array
            int f[] = new int[data.length+increment];
            for(int j = 0; j < data.length; j++) f[j] = data[j];
            f[size] = x;
            data = f; size++;
        }
```

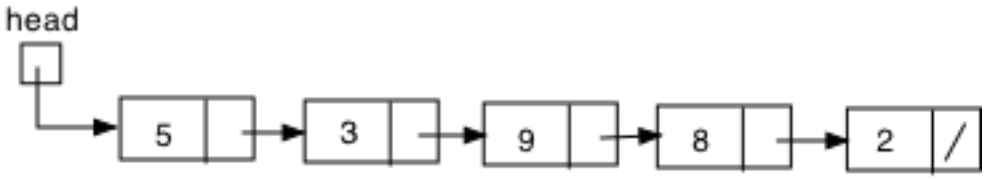
```
    }
    public boolean contains(int x){
        boolean found = false;
        int j = 0;
        while(j < size && !found){
          if(data[j] == x) found = true;
          else j++;
        }
        return found;
    }
    public boolean remove(int x){ //remove leftmost occurrence only
        boolean found = false;
        int j = 0;
        while(j < size && !found){
            if(data[j] == x) found = true;
            else j++;
        }
        if(found){
            while(j < size){ data[j] = data[j+1]; j++;}
            size--;
                return true;
        }
        else
            return false;
    }
    public int get(int ind){
         return data[ind];
    }
    public void set(int ind, int x){
        data[ind] = x;
    }
    public String toString(){
        String s = "[";
        int j = 0;
        while(j < size-1){
            s = s + data[j]+", ";
            j++;
        }
        s = s + data[j]+"]";
        return s;
    }
}
```

A program that tests some of these public methods is given below. Its meaning should be obvious from the context.

```
public class DynamicArrayTest {
   public static void main(String[] args) {
      IntArray f = new IntArray();
      for(int j = 0; j < 10;j++) f.add(j+1);
      System.out.println(f);
      f.remove(8);
      System.out.println(f);
      for(int j = 0; j < f.size(); j++){
         f.set(j,j+5);
      }
      System.out.println(f);
      System.out.println(f.contains(4));
   }
}
```

Dynamic Linked Lists

A linked list consists of zero or more nodes which together form a sequence. A node is a class consisting of a data attribute and a reference (pointer) attribute that is used to hold the address of the next node in the linked list, if any. The diagram below shows a list of 5 nodes linked by pointers. The variable head stores the address of the first element. The last element has no successor.



Historical note: *Linked lists were developed in 1955-56 by Newell, Shaw and Simon at Rand Corporation as the primary data structure in the programming language IPL that was used for implementing solutions to artificial intelligence problems. They also formed the primary data structure in LISP designed by McCarthy in 1958.* End note.

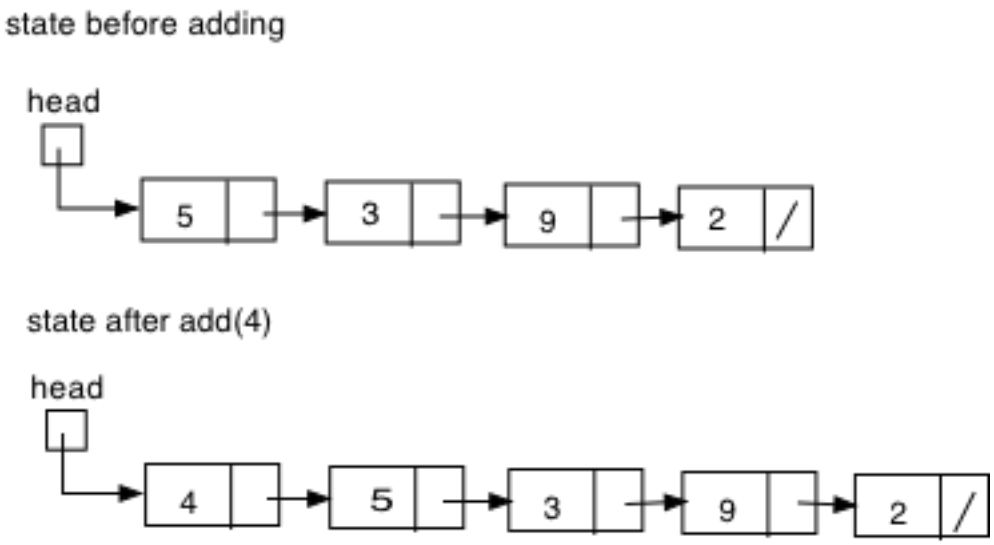
We represent a node as a class with two private attributes: data and next. The attribute next typically holds the address of the next node in the list. Initially it has a default value of null meaning that it has no successor node. The constant null will be used as a sentinel that

signifies the end of the list. The public method next() returns the reference to the next node in the list or null and setNext(Node p) is used to change the value of next. Finally, set(int x) modifies the value of the data attribute and data() returns its value. This class will be private to the IntLinkedList class (see below).

```
private class Node{
    int data;
    Node next;
    public Node(int x){
        data = x; next = null;
    }
    public Node next(){return next;}
    public void setNext(Node p){
        next = p;
    }
    public void set(int x){data = x;}
    public int data(){return data;}
}
```

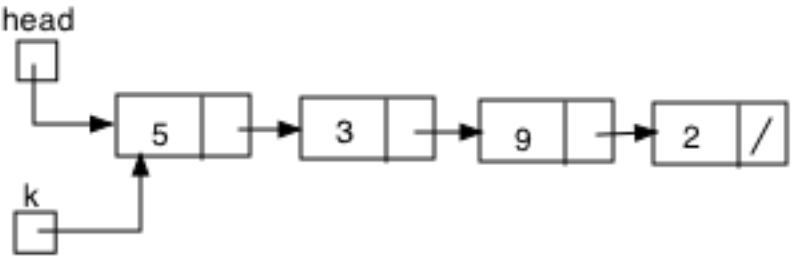
The class IntLinkedList uses instances of the Node class to construct a list. It has a single attribute head that stores the address of the first node in the chain and it has a default value of null indicating, in this instance, an empty list. The add method takes an integer x as argument, creates a new node and inserts it at the head of the list. It is *O(1)*. The code is:

```
public void add(int x){ //add at head
    Node nw = new Node(x);
    nw.setNext(head);
    head = nw;
}
```

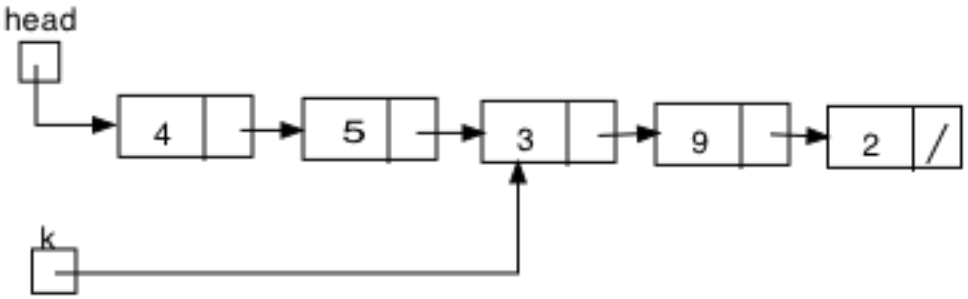


To search the list for a given value we perform a linear search starting at the head of the list. This is the only search method we can use because there are no indices. Hence, searching is always *O(n)*. We begin the search by assigning `k` the reference value of `head` and then continuously change it until we find `x` or reach the end of the list. It is crucial that we do not modify `head`. Doing so results in the loss of data and we cannot retrieve it. The diagram shows a snapshot of the state of the list in the middle of the search.

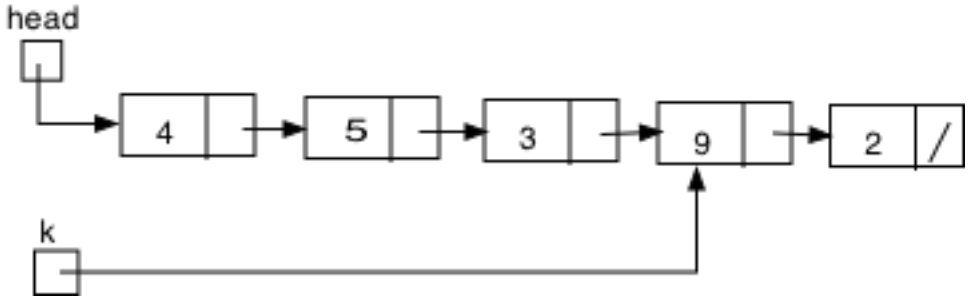
state at start of search for 9



state in middle of search for 9



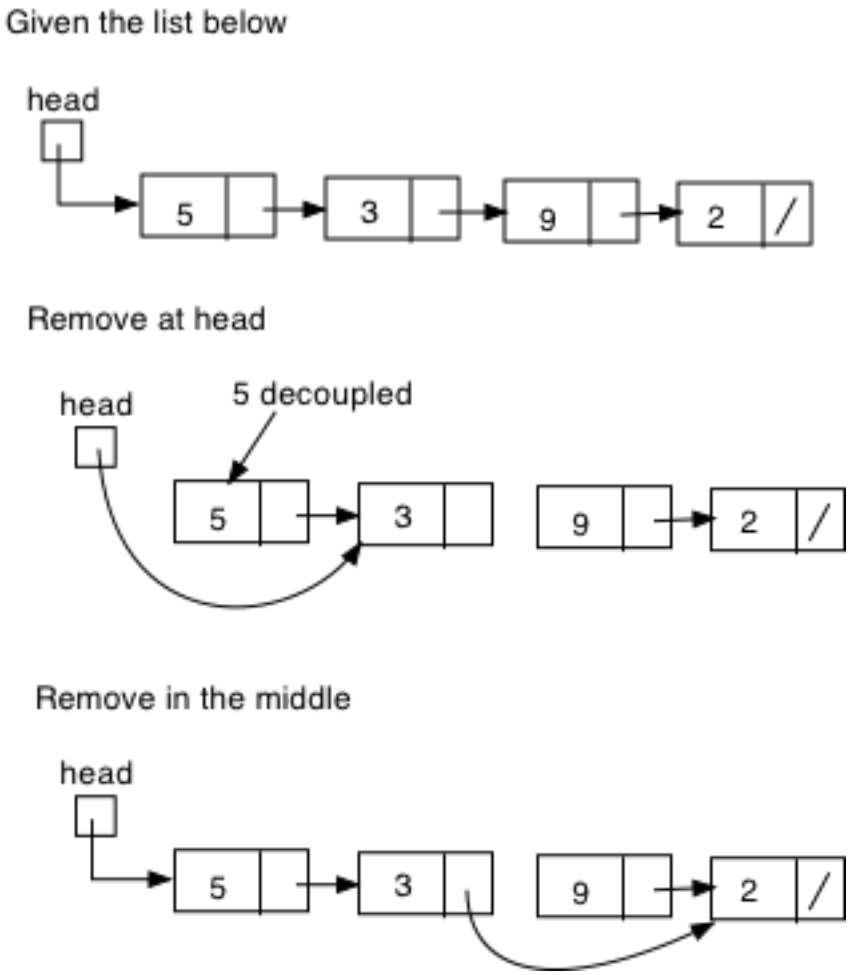
state when 9 found



The code for the methods is:

```
public boolean contains(int x){
    Node k = head;
    boolean found = false;
    while(k != null && !found){
        if(k.data() == x) found = true;
        else k = k.next();
    }
    return found;
}
```

To remove or delete an element in the list we first search the list to find it and then remove the node by re-directing the pointer preceding it to the next node in the list. The diagram below shows a list with 4 nodes. There are two cases. Deleting a node at the head of the list requires that we change the value of head – this is illustrated in the second part of the diagram. The third part illustrates removing a node in the middle of the list.



The method remove takes an integer x as argument and begins by searching for x. Notice that as it does so it records the address of the element preceding the node it is searching. This is necessary because the deletion of x requires the removal of a node from the chain. This is done by re-directing the pointer in the node preceding the one containing the value to be deleted to refer to its successor node. There are two cases to deal with.

```
if(k == head)
    head = k.next();
else
```

```
bk.setNext(k.next());
```

The code for the complete method is:

```
public void remove(int x){
    Node k = head; Node bk = head;
    boolean found = false;
    while(k != null && !found){
        if(k.data() == x) found = true;
        else{ bk = k; k = k.next();}
    }
    if(found)
        if(k == head)
            head = k.next();
        else
          bk.set(k.next());
}
```

Method delHead simply removes the node at the head of a list, if any. The code is:

```
public void delHead(){
    if(head != null)
    head = head.next();
}
```

Methods: size() returns the number of nodes or elements in the list; modify takes two arguments, the existing value to change and its new value, and searches the array until it finds and modifies x and returns true or fails to find it and returns false; toString returns a String representastion of the list of elements.

The complete listing for the class is:

```
public class IntLinkedList{
  private Node head = null;
  public void add(int x){ //add at head
    Node nw = new Node(x);
    nw.set(head);
    head = nw;
  }
  public boolean contains(int x){
    Node k = head;
    boolean found = false;
```

```
    while(k != null && !found){
        if(k.data() == x) found = true;
        else k = k.next();
    }
    return found;
}
public void delHead(){
    if(head != null)
        head = head.next();
}
public void remove(int x){
    Node k = head; Node bk = head;
    boolean found = false;
    while(k != null && !found){
        if(k.data() == x) found = true;
        else{ bk = k; k = k.next();}
    }
    if(found)
        if(k == head)
            head = k.next();
        else
          bk.set(k.next());
}
public int size(){
    Node k = head;
    int len = 0;
    while(k != null){
          len++; k = k.next();
    }
    return len;
}
public boolean modify(int x, int nx){
   //find and replace x with nx
   Node k = head;
   boolean found = false;
   while(k != null && !found){
        if(k.data() == x){
            k.set(nx);
            found = true;
        }
        else k = k.next();
   }
   return found;
}
```

```
    public String toString(){
      if(head == null) return "[]";
        String s = "[";
        Node k = head;
        while(k.next() != null){
            s = s + k.data()+", ";
            k = k.next();
        }
        s = s + k.data()+"]";
        return s;
    }
    private class Node{
        int data;
        Node next;
        public Node(int x){
            data = x; next = null;
        }
        public Node next(){return next;}
        public void set(Node p){
            next = p;
        }
        public int data(){return data;}
    }
}
```

Some additional methods that may prove useful are the ability to add a list to a list and to retrieve an array that replicates the contents of the list. The method `toArray` creates a dynamic array of size length of list, copies the data to it and returns it to the caller. The code is:

```
 public int[] toArray(){
    int f[] = new int[this.size()];
    Node k = head; int j = 0;
    while(k != null){
        f[j] = k.data();
        k = k.next(); j = j + 1;
    }
    return f;
 }
```

The method `add` takes a list as argument, replicates it as an array and then iterates over the sequence invoking `add` on itself as it goes. This is an expensive operation in terms of

memory allocation but it simplifies the task and will suffice for now. We will look at alternative approaches later.

```
public void add(IntLinkedList lst){
   int f[] = lst.toArray();
   for(int j = 0; j < f.length;j++) this.add(f[j]);
}
```

It is always useful to be able to sort a data list and the method below sorts the list in non decreasing order. It is possible to sort lists by copying data to a new sorted list but this approach is always $O(n^2)$. An alternative solution is to copy the elemets to an array and use a static method that sorts integer values from the class Arrays. This guarantees a fast sorting algorithm and removes all the complexity of managing pointers.

```
  public void sort(){
       int f[] = this.toArray();
       Arrays.sort(f);
       head = null;
       //insert largest first to create ascending list
       for(int j = f.length; j > 0; j--) this.add(f[j-1]);
 }
```

A sample program that tests some of these methods is given below. Again, its meaning should be obvious from the context.

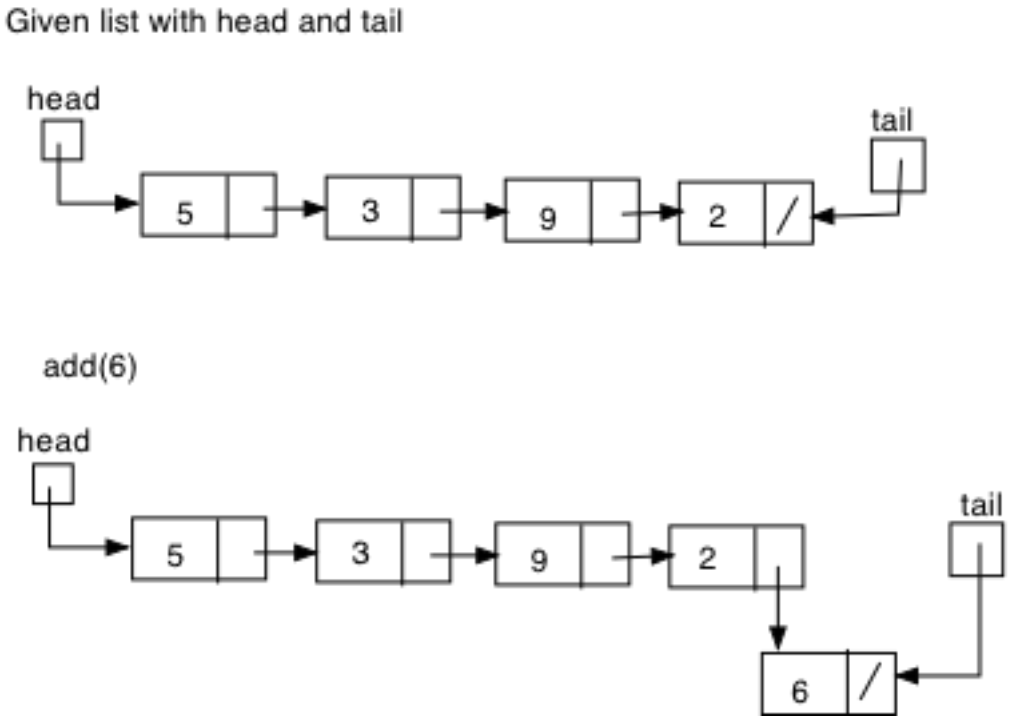
```
public class IntLinkedListTest{
   public static void main(String args[]){
       IntLinkedList lst = new IntLinkedList();
       lst.add(2); lst.add(6); lst.add(4);
       System.out.println(lst);
       System.out.println(lst.contains(4));
       lst.sort();
       System.out.println(lst);
       lst.remove(6);
       System.out.println(lst);
       lst.remove(4);
       System.out.println(lst);
       lst.remove(2);
       System.out.println(lst);
       for(int j = 0; j < 10; j++) lst.add(j);
       System.out.println(lst);
       lst.modify(5,12);
       lst.modify(11,0);
```

```
        System.out.println(lst);
        lst.sort();
        System.out.println(lst);
    }
}
```

Typically, we might like to store elements in **chronological order**. To do this we insert new elements at the end or tail of the list. To ensure that the implementation of this is optimised – *O(1)* – we maintain a pointer, called tail, that holds the address of the last element in the list. An empty list is given by:

```
Node head = null; Node tail = null; //empty list
```

A diagram illustrating the addition of a new node to the list is given below.

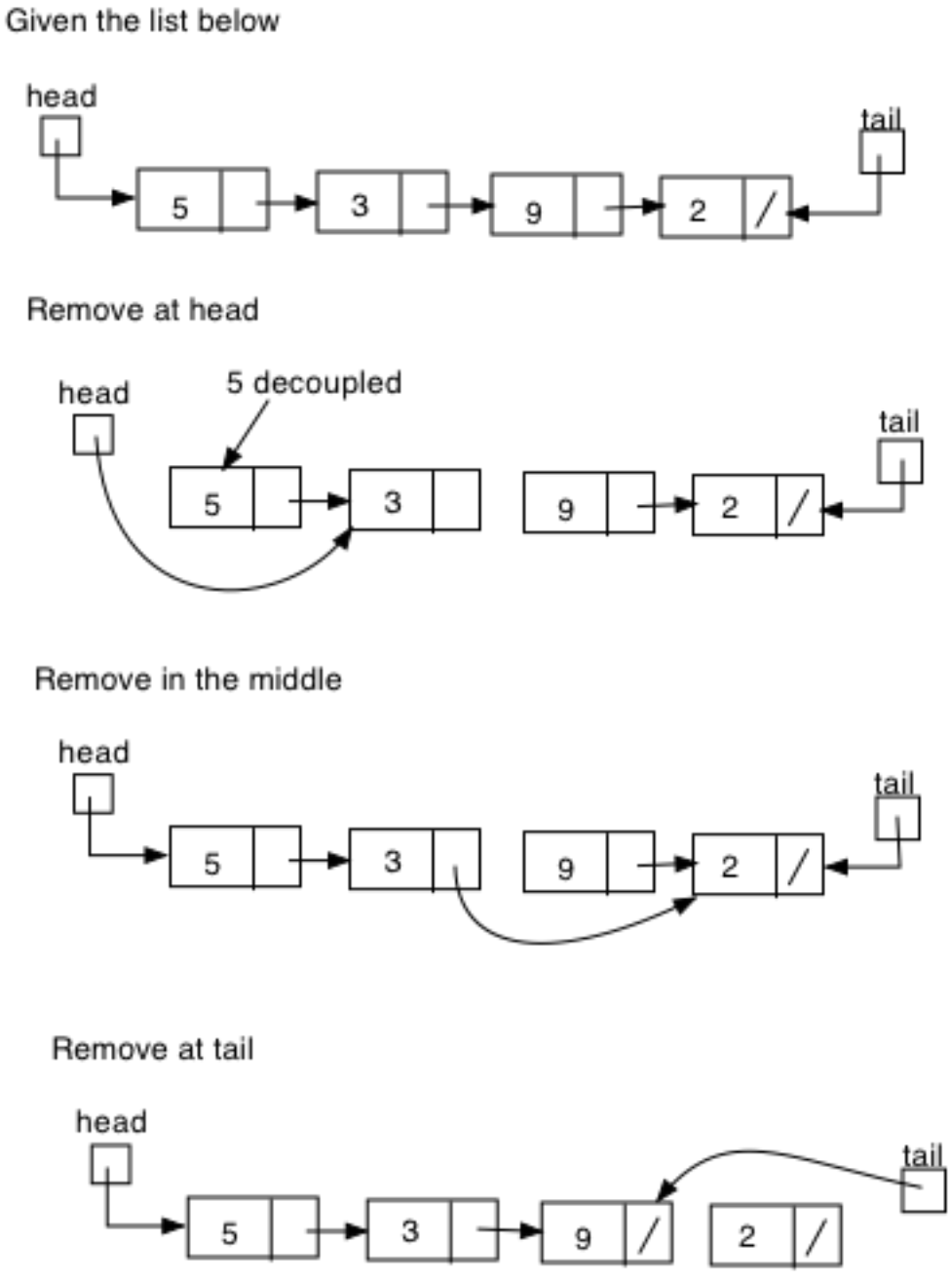


The code to implement this is:

```
Node head = null; Node tail = null; //empty list
public void add(int x){ //add at head
    Node nw = new Node(x);
    if(head == null){
        head = nw; tail = nw;
    }
```

```
    else{
        tail.setNext(nw);
        tail = nw;
    }
}
```

Removing an element becomes more complex now because we have to take into account three possible scenarios: removing the element at the head, an element in the middle and an element at the tail. The following diagram illustrates these scenarios.



The code to remove an element is:

```
public void remove(int x){
   Node k = head; Node bk = head;
   boolean found = false;
   while(k != null && !found){
      if(k.data() == x) found = true;
      else{ bk = k; k = k.next();}
   }
   if(found){
      if(k == head)
         head = k.next();
      else if(k == tail){
        bk.setNext(null);
        tail = bk;
      }
      else
         bk.setNext(k.next());
   }
}
```

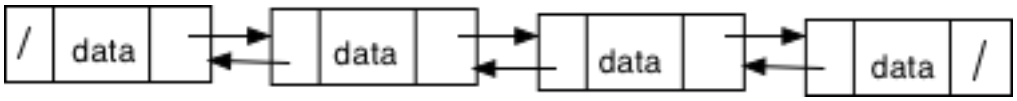
Exercise: Given below is asketch of the class LinkedListTail that always adds new elements to the tail of the list. Your task is to complete the methods listed and write a test program.

```
import java.util.*;
public class LinkedListTail{
   Node head = null; Node tail = null; //empty list
   public void add(int x){ //add at head
      Node nw = new Node(x);
      if(head == null){
         head = nw; tail = nw;
      }
      else{
      tail.setNext(nw);
        tail = nw;
      }
   }
   public void add(LinkedListTail lst){..}
   public int[] toArray(){..}
   public boolean contains(int x){..}
   public void delHead(){..}
   public void remove(int x){
```

```
        Node k = head; Node bk = head;
        boolean found = false;
        while(k != null && !found){
            if(k.data() == x) found = true;
            else{ bk = k; k = k.next();}
        }
        if(found){
            if(k == head)
                head = k.next();
            else if(k == tail){
              bk.setNext(null);
              tail = bk;
            }
            else
                bk.setNext(k.next());
        }
    }
    public void sort(){..}
    public int size(){..}
    public String toString(){..}
    private class Node{
        int data;
        Node next;
        public Node(int x){
            data = x; next = null;
        }
        public Node next(){return next;}
        public void setNext(Node p){
            next = p;
        }
        public void set(int x){data = x;}
        public int data(){return data;}
    }
}
```

Doubly Linked Lists

In a doubly linked list each node contains the address of both its predecessor and its successor. A diagram to illustrate it might be:



Each node holds the address of its predecessor and also its successor. These are represented by `prev` and `next` below. The first element has no predecessor and the last no successor, denoted by the constant `null`. A class to represent a node, therefore, has three attributes: two pointers and a single data attribute. Staying with the use of integers for our data we get the following implementation of a node. The public methods either return the values of the attributes or change them. Their meainings should be obvious from the context.

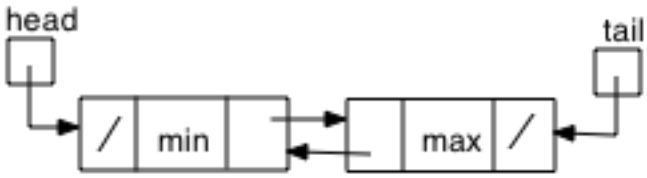
```
class DNode{
 int data;
 DNode prev = null;
 DNode next = null;
 public DNode(int x){ data = x;}
 public DNode prev(){return prev;}
 public DNode next(){return next;}
 public void setNext(DNode p){next = p;}
 public void setPrev(DNode p){prev = p;}
 public void set(int nx){x = nx;}
 public int data(){return data;}
}
```

To simplify the problem of insertion we define an empty list as one consisting of two sentinel nodes that each point to each other. The minimum and maximum integer values are used as sentinel values. The code for the default constructor is:

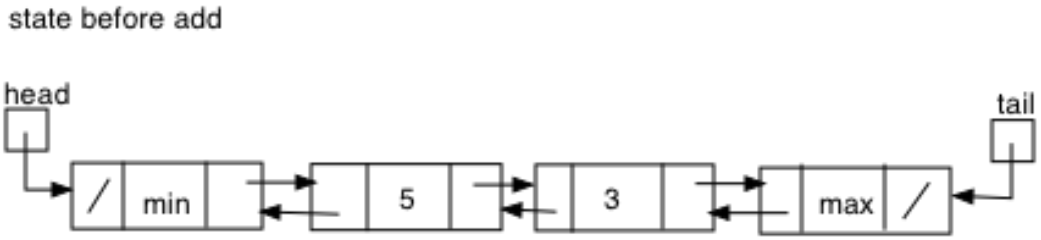
```
public DLinkedList(){
 head = new DNode(Integer.MIN_VALUE);
 tail = new DNode(Integer.MAX_VALUE);
```

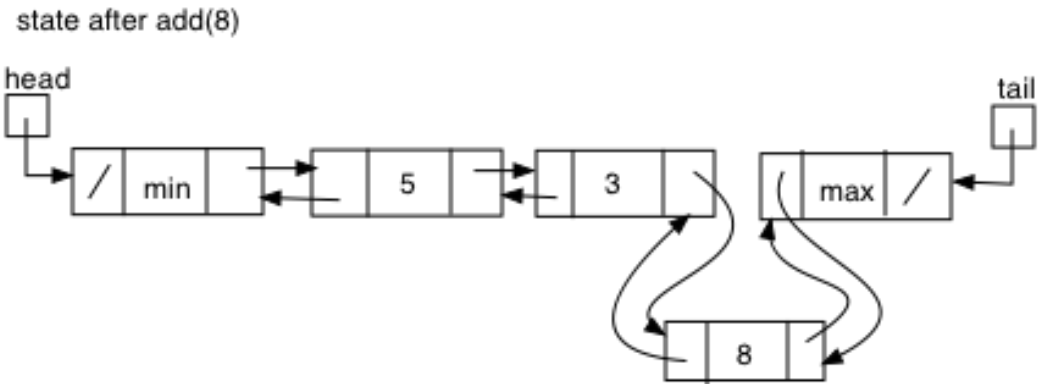
```
    head.setNext(tail); tail.setPrev(head);
  }
```

The reason for using this definition for the empty list is that it means insertion is always in the *middle*. This greatly simplifies the task of adding and removing elements. The diagram below illustrates the state of an empty list.



We add new elements at the tail of the list. To do so we have to modify four pointers. The diagram below illustrates this.



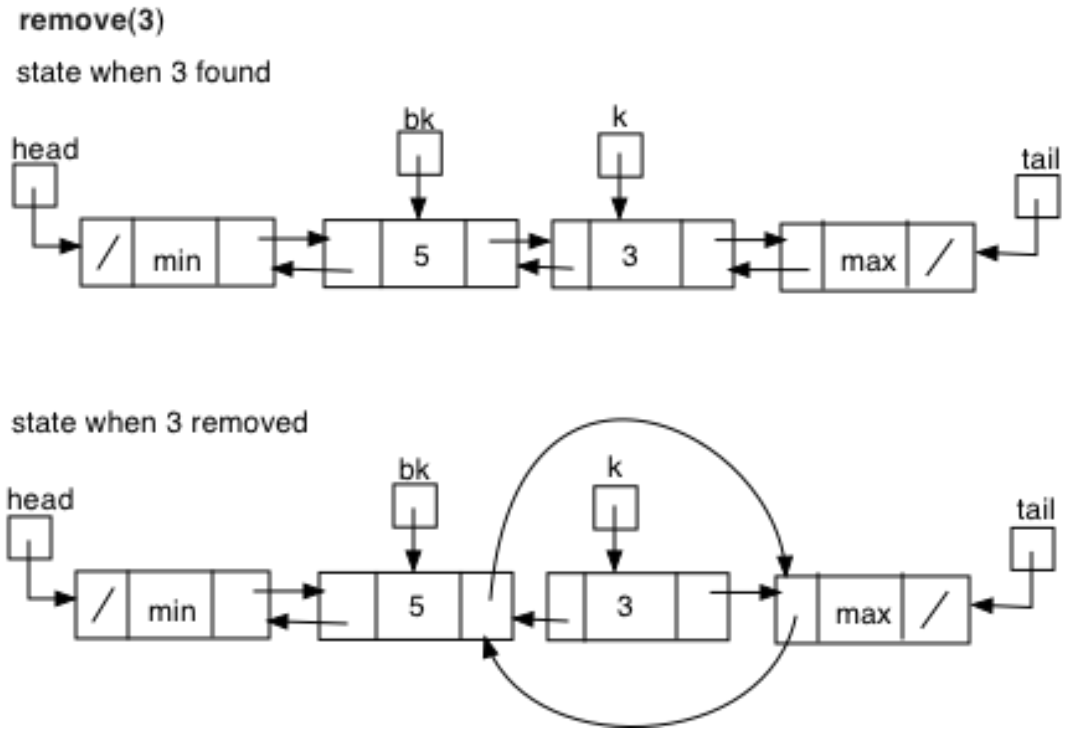


Elements are inserted chronologically, i.e. they are inserted before the tail sentinel. The code is:

```
    public void add(int x){
     //insert elements chronologically
```

```
    DNode p = new DNode(x);
    p.setNext(tail); p.setPrev(tail.prev());
    tail.prev().setNext(p);tail.setPrev(p);
  }
```

To remove an element we search the list until it is found and then modify the relevant pointers. The strategy used is to keep a record of the address of the predecessor of the present node that we are evaluating. Remember that removal is always in the *middle*, there are no special cases to examine. The diagram below illustrates the state when x is found and the state after the deletion of the node.



The code for the method remove is:

```
public void remove(int x){
   DNode k = head.next(); DNode bk = head;
   boolean found = false;
   while(k != tail && !found){
      if(k.data() == x) found = true;
      else{ bk = k; k = k.next();}
   }
   if(found){
      bk.setNext(k.next());
```

```
            k.next().setPrev(bk);
        }
    }
```

The complete class DlinkedList is:

```
class DLinkedList{
  DNode head; DNode tail;
  public DLinkedList(){
     //define empty list as list containing two empty nodes
     // This means that insertion is always in the middle.
    head = new DNode(Integer.MIN_VALUE);
    tail = new DNode(Integer.MAX_VALUE);
    head.setNext(tail); tail.setPrev(head);
  }
  public void add(int x){
    //insert elements chronologically
    DNode p = new DNode(x);
    p.setNext(tail); p.setPrev(tail.prev());
    tail.prev().setNext(p);tail.setPrev(p);
  }
  public void display(){
    DNode k = head.next();
    System.out.print('[');
    while(k != tail){
        if(k.next != tail)
          System.out.print(k.data()+", ");
        else
          System.out.print(k.data());
        k = k.next();
    }
    System.out.println(']');
  }
  public void displayReverse(){
    DNode k = tail.prev();
    System.out.print('[');
    while(k != head){
        if(k.prev != head)
          System.out.print(k.data()+", ");
        else
          System.out.print(k.data());
       k = k.prev();
     }
     System.out.println(']');
```

```
    }
    public int size(){
       DNode k = head.next();
       int len = 0;
       while(k != tail){
          len++; k = k.next();
       }
       return len;
    }
    public int[] toArray(){
      int f[] = new int[this.size()];
      DNode k = head.next(); int j = 0;
      while(k != tail){
         f[j] = k.data();
         k = k.next(); j = j + 1;
      }
      return f;
   }
   public boolean contains(int x){
       DNode k = head.next();
       boolean found = false;
       while(k != tail && !found){
         if(k.data() == x) found = true;
         else k = k.next();
       }
       return found;
    }
    public void remove(int x){
       DNode k = head.next(); DNode bk = head;
       boolean found = false;
       while(k != tail && !found){
         if(k.data() == x) found = true;
         else{ bk = k; k = k.next();}
       }
       if(found){
         bk.setNext(k.next());
         k.next().setPrev(bk);
       }
    }
}
```

An obvious benefit of linked lists over arrays is that elements can easily be inserted or removed without reorganising the elements in the existing list because the data is not stored contiguously in memory. Therefore, the cost of insertion or deletion is *O(1)* once the relevant

node is found. The search cost is *O(n)* even for sorted lists. However, it is easy to maintain a sorted list because shuffling is not required for either insertion or deletion. Doubly linked lists have the advantage that they can be traversed in both directions or from either end. On the down side, linked lists do not allow indexing and, therefore, random access to data is not possible. As a consequence, binary searching is not possible. Therefore, retrieval is *O(n).* But the real benefit in the use of dynamic linked lists lies in the ability to use them to design different data structures on demand to meet the needs of specific problems. We will also see later that they can be used to implement non-linear data structures that offer optimum performance in the case of insertion and retrieval.

Exercise

Question 1

Write a class `SortedIntArray` that implements the `List` interface and keeps the data sorted in non decreasing order.

Question 2

Amend the class `LinkedList` by adding public methods that:

(a) return the frequency of occurrence of a given x;

(b) return maximum value in the list;

(c) delete all occurrences of a given value x in the list;

(d) reverse the order of the elements in the list.

Write a program to test your amendments. Also include a static print function that prints only the positive values in a list.

Question 3

Write a class `LinkedListTail` that always adds new elements to the tail of the list. Your class should include a remove method as given above and methods that: display the list, return the number of nodes in the list, search the list for a given value and removeAll that removes all occurrences of a given value.

Question 4

A circular list is one in which the tail node always holds the address of the head node. Write a class that implements such a list always inserted new elements in chronological order. Your class should have a method to search the list for a given value, one to that returns the sum of

the values in the list and one to delete all occurrences of a given value. A display method should also be included.

Question 5

Write a class called `SortedLinkedList` that always inserts new elements such that the order of the values in the list is always non-decreasing. To simplify insertion of new values define the empty list as one with two nodes, head and tail, such that the value of the head node is `Integer.MIN_VALUE` and that of tail is `Integer.MAX_VALUE`. Your class should implement the `List` interface.

Question 6

Write a class called `SortedDLinkedList` that manages a doubly linked list of values that are always sorted in non-decreasing order.

Chapter 6: Generic Stacks and Queues

In the previous chapter on dynamic data structures we restricted the data type to that of integer values only. This meant that if we wanted to write a data structure to manage a list of books we would have to write a new class. This gives rise to lots of unnecessary duplication of code. It would be better if we could write a data structure with a type parameter that could be specified by the user of the class. This would simplify things greatly and would remove code duplication. From an object-oriented perspective what we want is a class that manages a collection of a given data type and that provides public methods that allow users to interact with the underlying data. One way to write these classes that avoids duplication is to design them so that they only store instances of the base type of all classes, namely the Object class. This works but it allows a user to store lots of different types of object in a given class making it difficult to retrieve elements and gives rise to lots of type casting. A better approach is to allow the user to choose the type of object and then restrict the data structures to instances of the given type only. This is possible by making the class generic at design time. The term used to describe this type of parameterization is *generics*. Both interfaces and classes can have type parameters. The formal type parameters are declared between angle brackets usually with a capital letter, e.g. `List<E>`. A generic class is a type pattern covering an infinite set of possible types. It only becomes an actual type when it is generically derived. That is, when it is given an actual generic type. **Type patterns once derived may only contain elements of the actual generic type**. Users of the class when creating instances simply substitute an actual type for the formal type variable, e.g. `List<Integer>`, `List<String>`, `List<Book>`, etc. Specifying the type in this way allows the compiler to verify (at compile-time) that the type of object you put into the collection is correct, thus reducing errors at runtime. This approach to writing data structures was pioneered by Ada in 1983, and is used in Eiffel, Java, C#, F#, and Visual Basic .NET; parametric polymorphism in ML, Scala, Swift and *templates* in C++. All the core collection interfaces in Java are generic. For example, the declaration of the `Collection` interface is: `public interface Collection<E>`.

We begin our study of generic data structures by implementing a generic array class. The name of the class includes the parameter `T` as an argument in its header and the private attribute data is declared as array of type `T`. However, in Java we cannot dynamically create an array with a deferred type `T`. To get around this restriction we create an Object array of size 50 and type cast it to array `T`. The default constructor is: `data = (T[])(new Object[50])`. This line of code causes the compiler to throw an unchecked type conversion warning. We simply suppress this and choose to ignore it because the type conversion is good. There is also a second constructor that takes creates an initial array of length n. The attribute size simply records the current number of elements curently in the data array. Its initial value is 0. Method add takes a value x of type `T` and assigns it to the data array. Note that this method

throws an exception if the array is full. However, we choose to ignore this problem for now because we want to focus exclusively on how to work with genericity.

```
class Array<T>{
   private T[] data;
   int size = 0;
   @SuppressWarnings("unchecked")
   public Array(){
      data = (T[])(new Object[50]);
   }
   @SuppressWarnings("unchecked")
   public Array(int n){
      data = (T[])(new Object[n]);
   }
   public int size(){
      return size;
   }
   public void add(T x){
      data[size] =x;
      size++;
   }
}
```

To use this class in a program we must substitute an actual type for the parameter T when declaring an instance of the Array<T> class. The following program illustrates its use by creating two different instances of Array<T>. The derived type must always be an object class because genericity is not defined for primitive types.

```
public class GenArrayTest {
   public static void main(String[] args) {
        Array<Integer> k = new Array<Integer>();
        for(int j = 0; j < 20; j++)
           k.add(j);
        System.out.println(k.size());
        Array<String> st = new Array<String>(10);
        st.add("Monday");
   }
}
```

This approach is simple to write and easy to use and it eliminates the need for duplicating code. However, a problem arises when you want to search the array for a given value or sort the data. To do so we need to use the comparison methods: compareTo or equals. The method equals is automatically inherited from the base class Object and can always be used

in the implementation of a search method. However, it only works if the actaul class has overriden this method in its implementation. If not it simply returns false. But these are not defined for our parameterized type T. In Java the interface Comparable imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's compareTo method is referred to as its *natural comparison method*. The interface has only one method compareTo.

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

In our generic class the parameterized type T will have to inherit this interface. We write this as:

```
class Array <T extends Comparable<T>>
```

The constructor is also amended to take into account this feature of T and is written:

```
data = (T[])(new Comparable[50]);
```

The public method contains performs a standard linear search. The code is:

```
    public boolean contains(T x){
     int j = 0;
     boolean found = false;
     while(j < size && !found)
        if(data[j].compareTo(x) == 0) found = true;
        else j++;
      return found;
   }
```

The amended class now is:

```
class Array<T extends Comparable<T>>{
      T[] data;
      int size = 0;
      @SuppressWarnings("unchecked")
      public Array(){
         data = (T[])(new Comparable[50]);
      }
      @SuppressWarnings("unchecked")
```

```
        public Array(int n){
            data = (T[])(new Comparable[n]);
        }
    public void add(T x){..}
     public int size(){return size;}
    public boolean contains(T x){//as above}
}
```

A simple program to test it is:

```
public class GenericTesting {
   public static void main(String args[]) {
    Array<String> s = new Array<String>();
    s.add("Hello"); s.add("now");
    boolean b = s.contains("now");
    System.out.println(b);
   }
}
```

Iteration

An iterator is an object that allows traversal over a given data structure. It is private to the class and knows how the data in the class is ordered. Iterators are useful from both the perspective of the user and the designer. From the users point of view they can traverse the elements in the collection without knowing how the actual collection is internally managed. From the designers point of view they can implement the data structure independently of a users view. This allows them to modify or change the implementation while preserving a consistent view of the data for users. The standard Iterator interface class in *Java* has three methods: next() that returns the next element in the data structure and advances the iterator to the next value, if any; hasNext() that returns true if there are more elements in the collection; the optional method remove() that removes the most recently visited element. A user-defined iterator must implement the first two of these methods, implementation of remove is optional. To use the iterator class you must import java.util.* in your program. To illustrate its use we implement an iterator for the Array<T> class above. The class ArrayIterator<T> is private in the Array<T> class.

```
   private static class ArrayIterator<T> implements Iterator<T>{
      private T[] d;
      private int index = 0;
      private int size;
      ArrayIterator(T[] dd, int s){
```

```
        d = dd; size = s;
    }
    public boolean hasNext(){
      return index < size;
    }
    public T next(){
      if(index == size) throw new NoSuchElementException();
      T item = (T) d[index]; index++;
      return item;
    }
    public void remove(){}
}
```

We add the public method `Iterator<T>` that returns a reference to an instance of the class `ArrayIterator<T>`. This reference variable is then used to iterate over the elements in the data structure. The code for the method is:

```
public Iterator<T> iterator(){
   return new ArrayIterator<T>(data, size);
}
```

A code fragment to use an instance of our iterator is:

```
Array<String> data = new Array<String>();
 ...
Iterator<String> it = data.iterator();
while(it.hasNext()){
    System.out.print(it.next()+" ");
}
```

By implementing an `Iterator` it is possible to use the `for` loop to traverse the elements in your `Array` class. For example, the code fragment above could be written as:

```
for(String s : data) System.out.print(s+" ");
```

The completed class with the addition of some exception handling is listed below. Note: the class is not complete and the addition of extra methods will be explored in the exercises at the end of this chapter.

```
import java.util.*;
class Array<T extends Comparable<T>> implements Iterable<T>{
   T[] data;
```

```
    int size = 0;
    private static final int DEFAULT_SIZE = 50;
    @SuppressWarnings("unchecked")
    public Array(){
        data = (T[])(new Comparable[DEFAULT_SIZE]);
    }
    @SuppressWarnings("unchecked")
    public Array(int n){
        if(n < 0) throw new IllegalArgumentException("Size cannot be negative");
        data = (T[])(new Comparable[n]);
    }
    public int size(){return size;}
    public void add(T x){
        data[size] = x;
        size++;
    }
    public boolean full(){return size == data.length;}
    public Iterator<T> iterator(){
        return new ArrayIterator<T>(data,size);
    }
    private class ArrayIterator<T> implements Iterator<T>{
        private T[] d;
        private int index = 0;
        private int size;
        ArrayIterator(T[] dd, int s){
            d = dd;size = s;
        }
        public boolean hasNext(){
            return index < size;
        }
        public T next(){
            if(index == size) throw new NoSuchElementException();
            T item = (T) d[index]; index++;
            return item;
        }
        public void remove(){}
    }
}
```

Generic Stack

A stack is a *last in, first out* linear data structure. It is characterized by two main operations: push and pop. The push operation adds a new item to the top of the stack, or initializes the stack if it is empty. If the stack has a fixed size and if add fails because it is full, then it is said

to be in overflow state. The pop operation removes the element at the top of the stack, if not empty. This means that elements are removed in inverse order to their insertion. Those last in get to leave first. To inspect the current element at the head of the stack a method top is provided. This method does not change the state of the stack. The methods empty and full simply return boolean values that indicate whether the stack is empty or full. For an unbounded stack full will always returns false.

```
interface Stack<E>{
    public boolean push(E x);
    public boolean pop();
    public E top();
    public boolean empty();
    public boolean full();
}
```

Historical note: The stack was first proposed in 1955, and then patented in 1957, by the Germans Klaus Samelson and Friedrich L. Bauer. The same concept was developed independently, at around the same time, by the Australian Charles Leonard Hamblin. **End note**

A generic stack can be implemented using an array or a linked list. Typically, an array is used for a fixed size stack and a linked list for unbounded stacks. The linked list requires additional memory to hold addresses of nodes. We provide an implementation using linked lists and the array implementation is left as an exercise. A generic node is implemented as follows:

```
class Node<E>{
    E data;
    Node<E> next;
    public Node(E x){
        data = x; next = null;
    }
    public Node<E> next(){return next;}
    public void setNext(Node<E> p){
        next = p;
    }
    public E data(){return data;}
}
```

The class StackList<E> implements the Stack<E> interface. New elements are inserted at the head of the list and pop removes elements from the head of the list. The code should be clear

from our discussion in the previous chapter.

```
class StackList<E> implements Stack<E>, Iterable<E>{
   Node<E> head;
   int size = 0;
   public StackList(){
      head = null;
   }
   public boolean push(E x){//add at head
      Node<E> nw = new Node<E>(x);
      nw.setNext(head);
      head = nw; size++;
      return true;
   }
   public boolean pop(){
    if(size > 0){ //remove at head
        head = head.next();
        size--;
        return true;
    }
    else return false;
   }
   public E top(){
      if(size > 0) return head.data();
      else return null;
   }
   public boolean empty(){
      return size == 0; // or head == null
   }
   public boolean full(){return false;}
   public Iterator<E> iterator(){
      return new StIterator<E>(head, size);
   }
   private static class StIterator<E> implements Iterator<E>{
      private Node<E> d;
      private int size;
      private int index = 0;
      StIterator(Node<E> dd, int s){
         d = dd; size = s;
      }
      public boolean hasNext(){
         return  index < size;
      }
      public E next(){
```

```
        if(index == size) throw new NoSuchElementException();
        E item = (E) d.data();
        d = d.next();index++;
        return item;
      }
      public void remove(){}
   }
   class Node<E>{ //code here}
  }
```

Given below is a simple test program for class StackList.

```
  public class StackTest {
  public static void main(String[] args) {
      StackArray<Integer> st = new StackArray<Integer>();
      for(int j = 0; j < 10; j++) st.push(j);
      while(!st.empty()){
          System.out.print(st.top()+" ");
          st.pop();
      }
      System.out.println();
      StackList<Integer> st1 = new StackList<Integer>();
      for(int j = 0; j < 10; j++) st1.push(j);
      while(!st1.empty()){
          System.out.print(st1.top()+" ");
          st1.pop();
      }
   }
  }
```

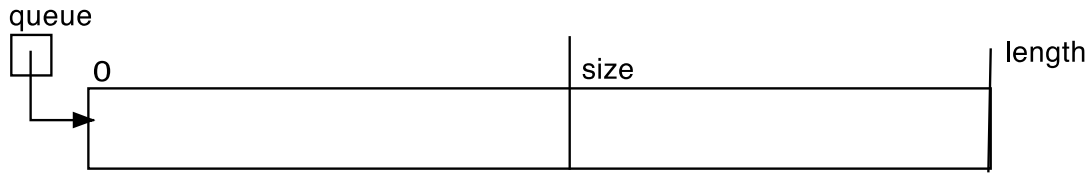
Generic Queue

A queue is *first in, first out* linear data structure. Elements are stored in chronological order and once en-queued may only be removed from the head of the queue. A queue is characterized by two main operations that change its state: join that appends new elements to the end of the queue, if not full and leave that removes the element at the head of a non-empty queue. The interface for a generic queue is:

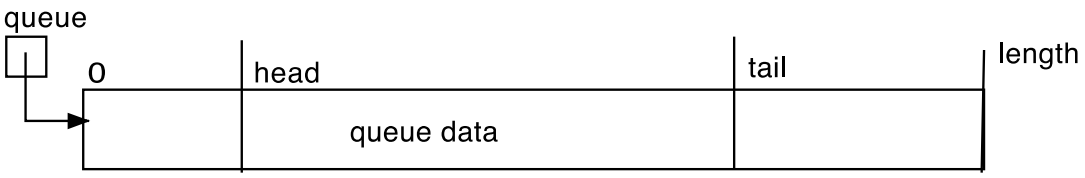
```
interface Queue<T>{
   public boolean join(T x);
   public T top();
```

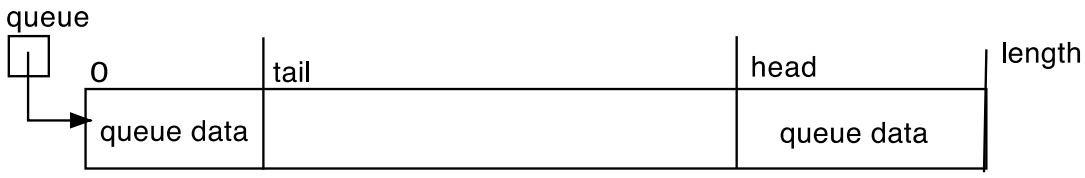
```
    public boolean leave();
    public boolean full();
    public boolean empty();
}
```

A generic queue can be implemented using an array or a linked list. Typically, an array is used for a fixed size queue and a linked list for unbounded queues. Again the linked list requires additional memory to hold addresses of nodes. However, the use of linked lists has a cost factor *O(1)* for both join and leave when the list maintains a pointer to both the head and tail of the queue. Implementing a linked list queue is left as an exercise. We provide, instead, an implementation based on arrays. The motivation for using arrays is that we can deliver an optimal solution using what are called **circular arrays** based on modulo arithmetic. Consider the following diagram of a linear queue.



A new element, x, can join the queue by simply assigning it to queue[size] and incrementing the value of size. However, removing an element from the queue is a problem because all the values to the right of queue[0] have to shift one position to the left. This has cost *O(size-1)*. To avoid this overhead we allow both the head and tail of the list to rotate around the array. The following diagrams illustrate this approach. The first diagram shows the queue with head < tail and the actual queue is given by the array queue[head .. tail-1]. In the second diagram tail < head and the queue data is contained in the arrays queue[head .. queue.length-1] and queue[0 .. tail-1].





New elements join the queue at the tail and leave at the head. The problem is how to update the values of head and tail when change occurs. It turns out that modulo arithmetic greatly simplifies this task. The new value of `tail` after a new element joins the queue is given by `tail = (tail+1)%queue.length` and `leave` modifies head such that `head = (head+1)%queue.length`. The method `join` takes a single argument `x`, checks that the queue is not full and assigns `x` to `queue[tail]` and then adjusts the value of `tail` based on the formula above. Variable `size` records the current size of the queue. It returns a `boolean` denoting success (`true`) or failure (`false`).

```
public boolean join(T x){
    if(size < queue.length){
        queue[tail] = x;
        tail = (tail+1)%queue.length;
        size++;
        return true;
    }
    else return false;
}
```

The method `leave` simply adjusts the value of `head` based on the formula above and updates the `size` of the queue. It returns `false` if the queue is empty; `true` otherwise. The code is:

```
public boolean leave(){
    if(size == 0) return false;
    else{
```

```
            head = (head+1)%queue.length;
            size--;
            return true;
        }
    }
```

The class CircularQueue<T> implements the Queue<T> interface by modeling the queue as a circular array. There are two constructors a default one that sets the size to an arbitrary value of 20 and one that takes the maximum size of the queue as an argument n. In both cases head and tail are set to zero.

```
import java.util.*;
public class CircularQueue<T> implements Queue<T>,Iterable<T>{
    private T queue[];
    private int head, tail, size;
    @SuppressWarnings("unchecked")
    public CircularQueue(){
        queue = (T[])new Object[20];
        head = 0; tail = 0;size = 0;
    }
    @SuppressWarnings("unchecked")
    public CircularQueue(int n){ //assume n >=0
        queue = (T[])new Object[n];
        size = 0; head = 0; tail = 0;
    }
    public boolean join(T x){
        if(size < queue.length){
            queue[tail] = x;
            tail = (tail+1)%queue.length;
            size++;
            return true;
        }
        else return false;
    }
    public T top(){
        if(size > 0)
            return queue[head];
        else
            return null;
    }
    public boolean leave(){
        if(size == 0) return false;
```

```
        else{
            head = (head+1)%queue.length;
            size--;
            return true;
        }
    }
    public boolean full(){
        return (size == queue.length);
    }
    public boolean empty(){
        return (size == 0);
    }
    public String toString(){
        String s = "[";
        int index = head;
        for(int k = 0; k < size-1; k++){
            s = s + queue[index]+", ";
            index = (index+1)%queue.length;
        }
        return s + queue[index]+"]";
    }
    public Iterator<T> iterator(){
        return new QIterator<T>(queue, head, size);
    }
    private static class QIterator<T> implements Iterator<T>{
        private T[] d;
        private int index;
        private int size;
        private int returned = 0;
        QIterator(T[] dd, int head, int s){
            d = dd; index = head; size = s;
        }
        public boolean hasNext(){
            return returned < size;
        }
        public T next(){
            if(returned == size) throw new NoSuchElementException();
            T item = (T)d[index];
            index = (index+1) % d.length;
            returned++;
            return item;
        }
        public void remove(){}
    }
}
```

A simple program that demonstrates the use of a queue is given below.

```
import java.util.*;
public class QueueTesting {
  public static void main(String args[]) {
    CircularQueue<Integer> cq = new CircularQueue<Integer>(6);
    for(int j = 0; j < 6; j++) cq.join(j);
    System.out.println(cq);
    cq.leave(); cq.leave(); cq.leave();
    System.out.println(cq);
    cq.join(8);
    System.out.println(cq);}}}
```

Applications of Stacks and Queues

Stacks have many applications both in the real world and in the world of computing systems. We create stacks of plates, stacks of books, etc, in real life. They are also used to implement recursive calls at run time on machines. Each time a recursive call is invoked the current state of the values is pushed onto a run-time stack to be popped when termination is reached and unfolding begins. Back tracking search algorithms use a stack to record the history of the search. They are used in recording decisions when searching a maze to allow back tracking to a given point. Queues are familiar to everyone and we try to avoid them as much as possible. They are used in the implementation of operating systems to implement fairness and to control access to resources based on a first come first served policy. Operating systems often use what are termed priority queues that order elements in the queue by priority value. We now look at two examples of using these data structures to solve specific problems.

Example 1

Write a program that reads a positive integer value and outputs it in binary form.

Solution

The solution is got by continuously dividing by 2 until the quotient is 0 keeping the remainder on each division. Using k as the integer value, this gives the following loop:

```
while(k > 0){
    int rem = (k % 2);
    //store rem
    k = k / 2;
```

```
   }
```

We cannot print the rem, as we go, because this will give the wrong result. For example, if k equals 23 it would output 11101 instead of the correct answer 10111. However, if we push rem onto a stack each time we will get the correct order when we finally pop the values. The complete solution is given below:

```
import java.util.*;
public class Converter {
   public static void main(String args[]){
    Scanner in = new Scanner(System.in);
    int x = in.nextInt();
    StackList<Integer> st = new StackList<Integer>();
    int k = x;
    while(k > 0){
       st.push(k % 2);
       k = k / 2;
    }
    while(!st.empty()){
       System.out.print(st.top());
       st.pop();
    }
  }
}
```

Example 2 Reverse Polish Notation Problem

In the 1920's the Polish logician Jan Lukasiewicz invented what became known as prefix Polish notation where the operators are written before the operands. For example, the expression 3 + 4 would be written as + 3 4. In the early 1960's Dijkstra and Bauer proposed what they termed reverse Polish notation (RPN). The motivation was to utilize the stack to evaluate expressions. In this schema the operands are written before the operators. The infix expression 3 + 4 becomes 3 4 + in RPN. A compound expression such as $(4 \times 5) - 6$ is be written as 4 5 × 6 − in RPN. Notice that brackets are removed in RPN. In RPN the operands belonging to a particular operator can be expressed without the need for brackets or operator precedence. Expressions are evaluated left-to-right, replacing an operator (when one occurs) with the result of that operator applied to the two previous operands. Therefore, the evaluation of 4 5 * 6 – gives:

4 5 * 6 - = 20 6 - = 14

As a second example, we evaluate 3 * (9 – 4). In RPN this becomes 3 9 4 - *. The evaluation is:

3 9 4 - * = 3 5 * = 15

Given a correctly formed RPN expression a fast algorithm to evaluate it can be written using a stack to store operands. *As operands are read from left-to-right, they are pushed onto the stack. When an operator is reached, its two operands are popped from the stack (the second operand is the left one and the first one the right one), the result of the operation is calculated, and the result is pushed back onto the stack. Processing all tokens in this way will result in a stack with a single value. This value is the result of the evaluation.*

The problem of converting an infix expression, containing possibly bracketed sub-expressions, to RPN still remains. To solve this problem we turn, once again, to Dijkstra. His solution is called the *Shunting-Yard Algorithm* based on the use of a shunting side-track, used to re-arrange the order of carriages in a train. Dijkstra, it would appear, had a great interest in railways. His famous Semaphore concept, used to control access to shared limited resources, in concurrent programming was named after the signal controlling access to a single track joining two sections of double rails.

The algorithm reads an infix expression as a list of tokens ands outputs them in a queue. It uses a stack to store left brackets, and operators that are to occur later in the output. Operands are output to the queue in exactly the same order as they occur in the infix expression. The algorithm is:

```
while(more tokens){
  t = next token
  if(t is an operand) append to output queue
  else if(t is an operator){
    while(priority of operator k at top of stack >= priority of t){
      pop k and append it to output queue
    }
    push t on stack
}
  else if(t is left bracket) push t on stack
  else if(t is right bracked){
```

```
            pop operstors at top of stack appending them to output queue
            until left bracket reached
          }
        }
        pop remaining operators from stack and append to output queue
```

The implementation of this algorithm is:

```
import java.util.*;
public class PolishNotationTest {
   public static void main(String args[]){
   //Read expression in infix notation and output
    //in reverse Polish notation
   StackList<String> stOp = new StackList<String>();
   CircularQueue<String> rpn = new CircularQueue<String>(100);
   Scanner in = new Scanner(System.in);
   String infix = in.nextLine();
   //System.out.println(infix);
   StringTokenizer tk = new StringTokenizer(infix);
   while(tk.hasMoreTokens()){
      String op = tk.nextToken();
      //System.out.print(op);
      if(isOperator(op)){
        if(stOp.empty())
         stOp.push(op);
        else{
          while(stOp.empty() == false && priority(stOp.top()) >= priority(op)){
         String temp = stOp.top();
         stOp.pop();
         rpn.join(temp);
          }
           stOp.push(op);
       }
      }
      else if(op.equals("(")){ stOp.push(op);}
      else if(op.equals(")")){
          while(!(stOp.top().equals("("))){
            rpn.join(stOp.top());
            stOp.pop();
          }
          stOp.pop();
      }
```

```
        else //op == operand
            rpn.join(op);
    }
    while(!stOp.empty()){
        rpn.join(stOp.top());
        stOp.pop();
    }

    System.out.println(rpn);

     //Now apply Polish algorithm to calculate value of the expression
    StackList<Integer> result = new StackList<Integer>();
    while(!rpn.empty()){
        String op = rpn.top(); rpn.leave();
        if(isOperator(op)){
            int a = result.top(); result.pop();
            int b = result.top(); result.pop();
            int x;
            if(op.equals("*")) x = b * a;
            else if(op.equals("/")) x = b / a;
            else if(op.equals("+")) x = b + a;
            else x = b - a;
            result.push(x);
        }
        else{
            Integer x = Integer.parseInt(op);
            result.push(x);
        }
    }
    System.out.println();
    System.out.println(result.top());

    }
    static boolean isOperator(String op){
    if(op.equals("*"))
        return true;
        else if(op.equals("/"))
        return true;
    else if(op.equals("+"))
        return true;
    else if(op.equals("-"))
        return true;
    else
        return false;
    }
```

```
    static int priority(String k){
    if(k.equals("*") || k.equals("/"))
        return 2;
    else if(k.equals("+") || k.equals("-"))
        return 1;
    else
        return 0;
    }
}
```

Implementations of the List Interface in the Collection Classes

In this chapter and the previous one we have discussed ways to implement dynamic arrays, linked lists, stacks and queues. This is a very necessary exercise because it gives insight into how to design generic classes that grow on demand. The Java Collection classes also provide Lists of different types. The most widely used are ArrayList and LinkedList. These two implementations provide similar public interfaces that allow a user to model a linear collection of objects. The main difference is that the ArrayList class uses a dynamic array to store elements, whereas, the LinkedList class uses a list of nodes connected by pointers. It maintains both a head and tail pointer making insertion at head and tail both optimal, i.e. O(1). However, it is only possible to search a list in linear time, O(n), because all searching must begin at the head. Linkedlists are useful when you want to implement queues and stacks. The table below lists the main methods for both classes. A list of all methods can be found in Java Docs.

Constructor	ArrayList<E>() ArrayList<E>(Collection) LinkedList<E>() LinkedList<E>(Collection)
Insert item	add(E elem)
Insert list	addAll(Collection<? extends E> lst)
Remove item	remove(Object ob)
Contains item	Boolean contains(Object ob)
Number of elements	int size()

Convert to string	toString()
Empty set	Boolean isEmpty()
Remove elements	clear()
Retrieve element given index value	E get(int index);
Insert element at index	add(int index, E elem);
Change element at index	E set(int index, E elem);
Remove element at index	E remove(int index)
Get index of object	int indexOf(E elem);
Additional Methods for LinkedList class	
Add new element at head of list	addFirst(E elem)
Return element at head of list	E getFirst()
Remove element at head of list	E removeFirst()
Returns an array containing all of the elements in this list in proper sequence; the runtime type of the returned array is that of the specified array. If the list fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this list.	<T> T[] toArray(T[] a) An example is: ArrayList<Integer> lst = new ArrayList<>(Arrays.asList(3,2,6,9,1)); Integer f[] = new Integer[lst.size()]; f = lst.toArray(f);

The List collection also has an implementation of a stack but it has been superseded by a new class interface called Deque (pronounced *deck*) that provides an optimal solution. There are two implementations of this interface: ArrayDeck and LinkedList. The ArrayDeck implementation uses a circular array that provides the same optimal performance as the CircularQueue class we implemented earlier.

Constructor	ArrayDeque<E>() ArrayDeque<E>(Collection) ArrayDeque(int numElements)
Insert item	addFirst(E elem) addLast(E elem)
Get element without removing it – throws exception if queue empty	E getFirst() E getLast()
Get element without removing it – returns null is queue empty	E peekFirst() E peekLast()
Contains item	Boolean contains(Object ob)
Number of elements	int size()
Returns true if queue empty	Boolean isEmpty()
Convert to string	toString()
Empty set	Boolean isEmpty()
Remove elements	clear()
Retrieve head or tail element, returning null if queue empty	E pollfirst() E pollLast()
Returns an array containing all of the elements in this list in proper sequence; the runtime type of the returned array is that of the specified array. If the list fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this list.	<T> T[] toArray(T[] a) An example is: ArrayDeque<Integer> dlst = new ArrayDeque<>(Arrays.asList(3,2,6,9,1)); Integer f[] = new Integer[dlst.size()]; f = dlst.toArray(f);

Working with this class is left as an exercise.

Exercise

Question 1

Write the following public methods for the Array<T> class given above:

(a) `public void sort()` that sorts the data in non-decreasing order using any sorting algorithm of your choice;

(b) `public int freq(T x)` that returns the frequency of occurrence of `x` in the data sequence.

Write a program to test your amendments that also uses the iterator class defined above.

Question 2

A class Book is given with all appropriate methods defined.

```
class Book implements Comparable<Book>{
   private String title;
   Book(String t){ title = t;}
   public String get(){return title;}
   public String toString(){ return title;}
   public boolean equals(Object p){
      Book b = (Book)p;
      return(title.equals(b.title));
   }
   public int compareTo(Book b){
      return title.compareTo(b.title);
   }
}
```

Write a program that creates an Array of Book.

Question 3

Implement the `Stack<E>` interface using an array to store elements. The stack, in this instance, is bound. Write a test program for your Stack class.

Question 4

Implement the `Queue<E>` interface using a linked list of nodes to store elements. The queue, in this instance, is unbounded. Write a test program for your queue class.

Question 5

Write a program that reads a java file on disk and checks if the number of opening curly brackets matches the number of closing curly brackets.

Question 6

Write a program that creates a stack of integer values such that the value at the top of the stack is always the smallest value.

Question 7

A priority queue is one where elements with the highest priority are stored at the head of the queue and where priorities are equal stored in chronological order. Write a class `PriorityQueue` that implements the `Queue<T>` interface.

Chapter 7: Genericity, Sub-Typing and Bounded Wildcards

The previous chapter provided an introduction to the use of genericity when writing data structures. In this chapter we want to go into more detail on issuesc arising when designing data structures that use generic typing. The design of the Java programming language has always been a work in progress and maintaining compatibility with earlier versions of the language has given rise to many problems. It turns out that implementing generic data structures is one of those areas where backward compatibility is a problem and if the designers were doing it right now from scratch they might do it quite differently. We should point out here that this chapter repeats some of the topics covered in the previous chapter for purposes of clarifying the issues involved. However, the examples will not be repeated.

We begin by implementing a linear list class that we will call MyList to distinguish it from the interface List<E> in the Collection Framework. As a general rule, we will prefix the names of our own implementations of generic data structures in the Framework with the two letter word *My*. The class MyList will provide a generically typed implementation of the LinkedList class developed for type int only in an earlier chapter. The one difference will be that in that case we inserted new elements at the head of the list, whereas, in this case, we will insert them at the tail of the list.

The name of the class includes the formal parameter E as an argument in its header and the private attributes head and tail hold references to the first node and the last node, respectively. The list is constructed using instances of the private class Node<E>. Again this class has a generic formal parameter as part of its argument header. An empty list is denoted by head == null and tail == null. No actual constructor is given for the class. The attribute size records the current number of elements in the list.

The Node<E> class has, as before, two private attributes: data of generic type E and next of type Node<E> that references the next element in the list or has value null signifying that it has no successor. Notice that the constructor makes no reference to E in its name, it simply takes as argument an instance of type E. Using formal generic parameters in constructor names is prohibited. The implementation here allows the possibility that x may be null. In other words, nodes in our list may contain null as actual data values. Method next returns a reference to the next node in the list or null. The set method changes the current data reference for a given node and method data returns a reference to the current data element whose reference is stored in the node. The class is private to the class MyList<E> because using a linked list to implement the class is only one of many possible solutions.

```
private static class Node<E>{
```

```
   E data;
   Node<E> next;
   public Node(E  x){
      data = x; next = null;
   }
   public Node<E> next(){return next;}
   public void setNext(Node<E> p){
      next = p;
   }
   public void set(E x){data = x;}
   public E data(){return data;}
 }
```

The version of MyList<E> listed below has five methods: add(E x), contains(E x), remove(E x), size() and toString(). The method add takes as argument x an instance of type E or null, creates a new Node and appends it to the current list, that may be empty, updating size accordingly. Method contains performs a linear search of the list and returns true if x found; false otherwise. The linear search starts at the head of the list and terminates when x.equals(kk) is true or when it reaches the end of the list. In the process it also checks for cases where both x and kk are both null. Note that all generic types are objects and, hence extend the base class Object. This means that all actual instances of E inherit the equals method. Of course the class in question must over-ride this method and provide its own definition of equality if it is to work properly. It then returns the value of found. The search must begin at head because the list is singly linked, where all pointers reference from left to right. This means that searching is always *O(N),* where N equals the current size of the list. The remove method begins by searching to find the node containing x and, if found, deletes the node by adjusting the appropriate pointers. There are three cases: delete at the head, delete at tail or delete in the middle. All three are covered by the given if statement. Method size simply returns the current number of nodes or elements in the list. The toString method overrides the toString method and provides a String representation of the elements in the list. Again this method assumes that the toString method is defined for instances of class E. If not, then it generates and returns a String of junk! Note that the string representation for an empty list is [].

```
class MyList<E>{
       private Node<E> head = null;
       private Node<E> tail = null;
   private int size = 0;
   public void add(E x){ //add at tail
       Node<E> nw = new Node<E>(x);
```

```
        if(head == null){
            head = nw; tail = nw;
        }
        else{
            tail.setNext(nw); tail = nw;
        }
        size++;
    }
    public boolean contains(E x){
        if(x == null) return false;
        Node<E> k = head;
        boolean found = false;
        while(k != null && !found){
            E kk = k.data();
            if(x.equals(kk)) found = true;
            else k = k.next();
        }
        return found;
    }
    public void remove(E x){
        Node<E> k = head; Node<E> bk = head;
        boolean found = false;
        while(k != null && !found){
            if(x.equals(k.data())) found = true;
            else{ bk = k; k = k.next();}
        }
        if(found){
            size--;
            if(k == head) head = k.next();
            else if(k == tail){
                bk.setNext(null); tail = bk;
            }
            else
              bk.setNext(k.next());
        }
    }
    public int size(){ return size; }
    public String toString(){
        if(head == null) return("[]"); //string for empty list
        String s = "[";
        Node<E> k = head;
        while(k.next() != null){
            s = s + k.data().toString()+", ";
            k = k.next();
        }
```

```
        s = s + k.data().toString()+"]";
        return s;
    }
    private static class Node<E>{
        E data;
        Node<E> next;
        public Node(E  x){
            data = x; next = null;
        }
        public Node<E> next(){return next;}
        public void setNext(Node<E> p){next = p;}
        public void set(E x){data = x;}
        public E data(){return data;}
    }
}
```

To use this class in a program we must substitute an actual type for the parameter E when declaring an instance of the MyList<E> class. The following program illustrates its use by creating two different instances of MyList<E>. This is exactly the same as we have done when using data structures from the Collection Framework. The program creates a list of Integer, adds some elements, prints the list to the screen, removes some values and then prints the updated list. The second instance creates a list of String, adds some names of famous philosophers, prints the list to the screen and finally illustrates an example of using the contains method.

```
class MyListTest{
  public static void main(String args[]){
    MyList<Integer> lst = new MyList<Integer>();
    for(int j = 0; j < 10;j++) lst.add(j);
    System.out.println(lst);
    lst.remove(0);lst.remove(9);
    lst.remove(5);
    System.out.println(lst);
    MyList<String> phlst = new MyList<String>();
phlst.add("Hegel");phlst.add("Fichte");phlst.add("Nietzsche"); phlst.add("Husserl");
    System.out.println(slst);
    System.out.println(slst.contains("Husserl"));
  }
}
```

Iteration

We can add elements, remove elements and check if our list contains a value but we cannot filter the values in the list. For example, we may want to print only the even values in a list of

integers or we may want to select only those strings that begin with a capital letter. To do this, for our generic class, we provide what is called an iterator. An iterator is an object that allows traversal over a given data structure. It is private to the class and knows how the data in the class is ordered. Iterators are useful from both the perspective of the user and the designer. From the users point of view they can traverse the elements in the collection without knowing how the actual collection is internally managed. From the designers point of view they can implement the data structure independently of a users view. This allows them to modify or change the implementation while preserving a consistent view of the data. The standard Iterator interface in *Java* has three methods: next() that returns the next element in the data structure and advances the iterator to the next value, if any; hasNext() that returns true if there are more elements in the collection; the optional method remove() that removes the most recently visited element. A user-defined iterator must implement the first two of these methods. It is not necessary to implement the third. However, if you do not do so you should not try to remove elements while iterating over the list. Doing so will modify the list and may result in a NoSuchElement exception. The only reliable way to delete or remove elements from a data structure while iterating is to implement remove. To use the iterator class you must import java.util.* in your program. To illustrate its use we implement an iterator method for our MyList<E> class by implementing an anonymous inner class. The public method iterator() returns an instance of Iterator<E> and implements: next() and hasNext. This anonymous class has two attributes: h that is initialized with the reference variable head, index that is set to 0. The method hasNext returns true if index < size; false otherwise. Method next returns the next item in the list throwing an exception if none available. It also advances the pointer and increments index. The method remove simply throws an UnSupportedOperationException.

```
public Iterator<E> iterator(){
   return new Iterator<E>(){
      private Node<E> h = head;
      private int index = 0;
      public boolean hasNext(){
         return index < size;
      }
      public E next(){
         if(index == size) throw new NoSuchElementException();
         E item = h.data();
         h = h.next(); index++;
         return item;
      }
      public void remove(){
```

```
            throw new UnSupportedOperationException();
        }
    };
```

We also amend the name of our class by stating that it implements the Iterable<E> interface. It now becomes class MyList<E> implements Iterable<E>. Making this change means that we can now iterate over the data using a for *each* loop. This is similar to the code for traversing an ArrayList. To print the philosophers whose name begins with the capital letter 'H' we could write:

```
for(String nm : phlst )
  if(nm.charAt(0) == 'H') System.out.print(nm+" ");
System.out.println();
```

Next we consider the issue of adding an existing list of elements or removing a list of elements. At present our class only allows a user to add or remove a single occurrence of an element at the time. Suppose a user has a list of values and they want to add all of them to another list the only way to do it is add them one at a time. To simplify this process we provide a new method called addAll. As a first pass at writing this method we could write the following solution:

```
public void addAll(MyList<E> ls){
   for(E x : ls) this.add(x);
}
```

To test this new method we create a new list of philosophers and then add the list to the existing one.

```
MyList<String> phlst1 = new MyList<String>();
phlst1.add("Camus");phlst1.add("Foucault");
phlst.addAll(phlst1);
System.out.println(phlst);
```

This will give as output:

[Hegel, Fichte, Nietzsche, Husserl, Camus, Foucault]

This is correct. However, it imposes restrictions on the user. Problems will arise if an end user tries to use this method where class inheritance relations are involved. Suppose we have two classes: Person and Employee, where class Employee extends class Person. Let the classes be implemented as given below.

```
class Person{
 protected String name;
 Person(String n){name = n;}
 String name(){return name;}
 public String toString(){return name;}
 public boolean equals(Object ob){
   if(!(ob instanceof Person)) return false;
   else{
      Person p = (Person)ob;
      return name.equals(p.name);
   }
 }
}
class Employee extends Person{
 private String empNumber;
 Employee(String n, String num){super(n); empNumber = num;}
 String employeeNumber(){return empNumber;}
 public String toString(){
   return name+" : "+ empNumber;
 }
 public boolean equals(Object ob){
   if(!(ob instanceof Employee)) return false;
   else{
      Employee e =(Employee)ob;
      return empNumber.equals(e.empNumber);
   }
 }
```

A Person instance has a name and an Employee instance is a Person with an employee number. Both classes have equal methods, toString methods and an observer method. Now imagine that we have three separate lists, call them lst1, lst2 and lst3 that are declared as follows:

```
MyList<Person> lst1= new MyList<Person>();
MyList<Person> lst2 = new MyList<Person>();
MyList<Employee> lst3 = new MyList<Employee>();
```

We can add all the persons in lst2 to lst1 using lst1.addAll(lst2) but if we try to do the same thing using lst3 we get the following compiler error.

```
error: incompatible types: MyList<Employee> cannot be converted to MyList<Person>
            lst1.addAll(lst3);
                        ^
```

This is surprising because we can certainly add individual employees to lst1. The statement pl.add(new Employee("Eoin","0001")) compiles and executes correctly. This makes absolute sense because employee instances are person instances due to inheritance. Of course the opposite is not true: person instances are not employee instances. The simple answer to our problem is that in Java while it is okay to assign *subtypes to super types* it is not okay to assign *lists of sub types to lists of super types*. This means, in this case, that while Employee is a sub type of Person MyList<Employee> is **not** a sub type of MyList<Person>. This is why the compiler states that MyList<Employee> cannot be converted to MyList<Person>. Our method only permits adding when both types are exactly the same. It does not accommodate inheritance relationships between MyList types. The reasons for this are quite technical and we defer discussion of this topic for now (see The Theory of Generics in Java below). The way Java solves this problem is through the use of what are called bounded wildcards. There are two types of bounded wildcards: <? extends E> and <? super E>. The former is used when we want to *get or read* elements from a collection and the later is used when we want to *put or write* elements to a collection. The method addAll wants to *get* elements from its argument list. Hence, we modify the argument type to MyList<? extends E>. The method addAll becomes:

```
public void addAll(MyList<? extends E> ls){
  for(E x : ls) this.add(x);
}
```

This compiles correctly and allows lst1.addAll(lst3) to execute correctly. Of course, we now end up with a list that contains both Person instances and Employee instances. A programmer may not always want this to be the case but that is up to the programmer. From our perspective, as an implementer of a data structure, we want to provide the maximum service possible. If we can add individual employees to a person list then we should be able to add a collection of them. But could we improve on this further? Suppose you have an ArrayList<Emlpoyee> and you want to add all of them to our list lst1 using method addAll. The following lines of code will generate a similar error to the one we got above. The code sequence:

```
List<Employee> alst = new ArrayList<Employee>();
alst.add(new Employee("Mike","001"));
lst1.addAll(alst);
```

gives rise to:

```
error: incompatible types: List<Employee> cannot be converted to MyList<? extends
Person>
```

```
            lst1.addAll(alst);
                    ^
```

To remove this error we note that `MyList extends Iterable<E>` and if we change the argument type of our `addAll` method to `Iterable<? extends E>` it works fine. The latest and final implementation of addAll is:

```
public void addAll(Iterable<? extends E> ls){
   for(E x : ls) this.add(x);
}
```

This solution allows copying from any class that implements the `Iterable<E>` interface. This covers all classes from the Collection Framework and any user defined data structures, such as `MyList<E>`, that implement the `Iterable<E>` interface.

A similar situation arises if you want to write to a list. Consider the method `copy` that takes an instance of `MyList` as argument and copies the current list to the given list.

```
public void copy(MyList<E> ls){
   for(E x : this) ls.add(x);
}
```

This works fine if the given list and the argument list have exactly the same type. But if you try to copy from a list of `Employee` to a list of `Person` it fails. The code sequence:

```
List<Employee> alst = new ArrayList<Employee>();
alst.add(new Employee("Mike","001"));
MyList<Person> perlst = new MyList<Person>();
alst.copy(perlst);
```

when compiled, gives the following output:

```
error: no suitable method found for copy(MyList<Person>)
            emplst.copy(perlst);
                ^
  method MyList.copy(MyList<Employee>) is not applicable
    (argument mismatch; MyList<Person> cannot be converted to MyList<Employee>)
  method MyList.<T>copy(MyList<? super T>,MyList<? extends T>) is not applicable
    (cannot infer type-variable(s) T
      (actual and formal argument lists differ in length))
 where T is a type-variable:
  T extends Object declared in method <T>copy(MyList<? super T>,MyList<? extends T>)
```

This is quite a mouthful! Without going into all of the technical terms listed the problem can be resolved by using the bounded wildcard `<? super E>`. What we want to do is copy the

given list of employees to a list of person. In our Person family type Person is a super type of type Employee. To convey this information to the compiler for instances of MyList we amend the signature of our method to MyList<? super E>. This gives:

```
public void copy(MyList<? super E> ls){
  for(E x : this) ls.add(x);
}
```

This method permits copying from a given instance of MyList to another intance of MyList for types in the same ancestral family. If you want to copy a given list to a list in the Collection Framework you could add the following copy method.

```
public void copy(Collection<? super E> ls){
   for(E x : this) ls.add(x);
}
```

Note that we use Collection<? super E> and not Iterable<? super E> because Iterable does not have an add method.

Another way to solve the problem of copying one list to another list might be to write a static method as part of the class MyList to do so. This turns out to be a nice problem because its solution illustrates the use of both type of bounded wildcard as part of the signature of the static method. We need <? extends E> for the source list and <? super E> for the destination list. The code is:

```
public static <T> void copy(MyList<? super T> dst, MyList<? extends T> src){
   for(T x : src) dst.add(x);
}
```

This permits copying within the same homogenous ancestral family. The destination list, dst, must be an instance of the current generation T or an ancestor of T and the source, src, must be a member of the current generation T or a successor of T. The source, src, **produces or supplies** values and the destination, dst, **consumes** values. To quote Bloch: *if a parameterized type represents a T producer, use <? extends T>; if it represents a T consumer, use <? super T>* (Bloch, 2008).

The idea that we can write to lists of the current or previous generations and read from lists of the current generation or successor generations is also captured nicely by Naftler and Wadler when they state that we should use the *Get* and *Put* principle to decide which type of wildcard to use. This principle is stated by them as follows:

> *The Get and Put Principle*: use an extends wildcard when you only *get* values out of a structure, use a super wildcard when you only *put* values into a structure, and don't use a wildcard when you both *get* and *put*.
>
> *(Naftler&Wadler, 2007)*

The static copy method above belongs to the class MyList and not to an instance of the class. This means that it is independent of any type parameters for actual instances of the class. Hence, we are not permitted to refer to the formal parameter E. To make the method generic we declare a type parameter representing the element type of the two MyList classes and use the type parameter in the method. The type parameter is declared after the word static and before any return type. Hence, the signature of copy is static <T> void copy(…). If we wanted to write a join method that returned a new instance of MyList that took two lists as arguments we would use the signature public static <T> MyList<T> join(MyList<? extends T> l1, MyList<? extends T> l2). Note that the return type is MyList<T>. We do not use bounded wildcards as return types. Bloch points out that the use of wildcards should be invisible to users of a class and, as a consequence, should not be used as return types. The code for join is:

```
public static <T> MyList<T> join(MyList<? extends T> l1, MyList<? extends T> l2){
   MyList<T> r = new MyList<T>();
   r.addAll(l1);
   r.addAll(l2);
   return r;
}
```

A code segment that tests both of these two methods might be as follows:

```
MyList<Person> plst1 = new MyList<Person>();
MyList.copy(plst1,el);
System.out.println(plst1);

MyList<Person> plst2 = MyList.join(plst1,el);
System.out.println(plst2);
```

Now we turn our attention to defining an equals method for our class MyList<E>. Two lists are said to be equal if they both have the same number of elements and if all consecutive corresponding pairs of values are equal. Implementing an equals method requires that we use the instanceof operator to check that type of Object passed as an argument is correct. But in Java generic type information is actually erased at runtime and, hence, it is illegal to use the instanceof operator on paramerteized types. There is no problem with this because the parameterized type is actually replaced with what is called a raw type. In our case the raw type name for MyList<E> is MyList. The equals method begins by checking that argument ob is an instance of MyList. If not, or if it is null, it returns false. It then type casts the Object ob corrctly to MyList<?>. The unbounded wildcard <?> is just shorthand for <? extends Object>. Once it has established that both lists have the same number of elements it creates an iterator for one of the lists and uses a for each loop to traverse the other list in sequence returning

false if a pair of values differ. Finally, if no values differ it returns true. The code for the method is:

```
public boolean equals(Object ob){
   if(!(ob instanceof MyList)) return false;
   MyList<?> ls = (MyList<?>)ob;
   if(this.size != ls.size) return false;
   Iterator it = ls.iterator();
   for(E x : this){
      if(!x.equals(it.next())) return false;
   }
   return true;
}
```

A simple test sequence is given below:

```
MyList<Integer> eqlst = new MyList<Integer>();
for(int j = 0; j < 10;j++) eqlst.add(j);
MyList<Integer> eqlst1 = new MyList<Integer>();
for(int j = 0; j < 10;j++) eqlst1.add(j);
System.out.println("Lists equal: "+eqlst.equals(eqlst1));
MyList<Integer> eqlst2 = new MyList<Integer>();
for(int j = 0; j < 10;j++) eqlst2.add(j+1);
System.out.println("Lists equal: "+eqlst.equals(eqlst2));
```

Finally, we look at how to write methods that use the interfaces `Comparable<T>` and `Comparator<T>`. We studied applications of these interfaces when we used the ArrayList class to manage linear collections of object instances. There we saw that they could be used to order elements in a collection using the natural ordering given by `compareTo`. Additional orderings were given by using a `Comparator` for a given class. The Collections class provides `sort` methods that use both of these to order elements in a Collection. These methods can also be used to find the max or min elements in a collection and to find the frequency of occurrence of a given element. In this instance, we want to explore adding such methods to our `MyList` class. To illustrate the technique we develop a single static method `max`. Firstly, we deal with the case where `T` implemennts the `Comparable` interface. This returns the maximum value in a given list based on the natrual ordering of the values defined by the `compareTo` method of the class. The signature of the method we propose is:

```
public static <T extends Comparable<T>> T max(MyList<? extends T> lst)
```

The type parameter uses a recursive bounded wildcard because type T must implement the Comparable interface. The return type is T. The argument list, `lst`, simply produces values and, hence, under the *Get* principle for generics is written as `<? extends T>`. In fact we could argue that the `compareTo` method is a consumer of values and, hence, the recursive type parameter should be `<T extends Comparable<? super T>>`. This is how Naftler and Wadler implement it (p 36). The implementation uses the iterator from the `lst` class to initialize the

candidate max value k. It then uses a for each loop to traverse the values updating k based on a comparison of its value with the current element, el, in the list. The code is:

```
public static <T extends Comparable<T>> T max MyList<? extends T> lst){
   T k = lst.iterator().next();
   for(T el:lst) if(k.compareTo(el) < 0) k = el;
   return k;
}
```

The second implementation of max takes two arguments: the list, lst, and an instance of the Comparator class. This class has a single method compare that takes two arguments of type <? super T>. The compare method is a **consumer** and, hence, by the *Put* rule for generics it uses <? super T>. The actual type parameter for the method is just <T>. The implementation is similar to the first one, except that it uses comp.compare(k, el) to do the comparisons. The code is:

```
public static <T> T max(MyList<? extends T> lst, Comparator<? super T> comp){
   T k = lst.iterator().next();
   for(T el:lst) if(comp.compare(k,el) < 0) k = el;
   return k;
}
```

Exercise: Implement a min method that returns the smallest element in a list. Also write a frequency(MyList<? extends T> lst, Object ob) method that counts the frequency of occurrence of ob in lst.

To test these new methods we amend the Person class so that it implements the Comparable<Person> interface. Two additional methods are required. The compareTo method uses the compareTo method for the String class. The nameLenCompare method returns an anonymous instance of the Comparator<Person> class that compares two Person instances based on the length of each name. The code for both methods is given below.

```
public int compareTo(Person p){
   if(p == null) throw new NullPointerException();
   return name.compareTo(p.name);
}
public static Comparator<Person> nameLenCompare
 = new Comparator<Person>(){
         public int compare(Person p1, Person p2){
            return
               p1.name.length() < p2.name.length() ? -1 :
               p1.name.length() > p2.name.length() ? 1 : 0;
      }
 };
```

The following code fragment gives an example of testing these methods with an instance of MyList<Person> pl.

```
System.out.println("Max name: "+ MyList.max(pl));
System.out.println("Longest name: "+MyList.max(pl,Person.nameLenCompare));
```

The class MyList<E> and a test program are given in the listings at the end of this chapter.

Theory of Generics in Java

Generics were introduced in Java 1.5 and are supported by all versions of the language since then. However, the Collection Framework appeared in version 1.2 and did not support generics. This earlier Framework used raw types for all of its data structures or object containers. The base type of all of these was type Object. One of the difficulties encountered in using these classes was that one had to do type casting when reading values from a collection type. For example, if an ArrayList contained instances of the Integer class then to traverse the list you would have to typecast the values to Integer. Suppose lst is an ArrayList of integer values, the code to iterate over the list would be:

```
Iterator it = lst.iterator();
while(it.hasNext()){
   Integer k = (Integer)it.next();
   // process k
}
```

A consequence of this approach is that if the typecast fails you get run-time errors in your code. With the advent of generics all type checking can be done at compile them and, hence, typing errors are trapped before the program is executed. This guarantees that the type system of your program is correct. However, it posed a problem for the Java designers because they had to ensure backward compatibility. The way they resolved this problem was to *erase* the type information at run-time and replace it with Object references. As a result, *the runtime representation contains less information than the compile time representation* (Bloch, p120). Therefore, raw type of List<String>, List<Person>, List<Integer> is just List.

In Java, through inheritance we establish ancestral relationships between classes. Given two classes A and B such that class B extends A we say that B is a sub-type of A and that A is a super-type of B. We also know that all classes extend the Object class and that the Object class is a super-type of all classes. The class hierarchy is bounded below by null. Therefore, the type range in Java is: Object .. null. Every type is a sub-type of itself. There is also a transitive relation between types: if C is a sub-type of B and B is a sub-type of A, then C is a sub-type of A. However, the relationships are not symmetric. If A is a sub-type of B, then B is not a sub-type of A. A consequence, of this is that a variable of a given type may be

assigned a value of any of its sub-types. This is known as the *Substitution Principle*. Because Employee is a sub-type of Person the assignment Person p = new Employee(..); is valid. A similar situation holds for methods in classes. Given an ArrayList<Person>, called lst, the invocation lst.add(new Employee(..)) is valid.

However, this sub-typing hierarchy and its substitution rule do not extend to Collections of classes. While Employee may be a sub-type of Person, it is not the case that List<Employee> is a sub-type of List<Person>. This makes the sub-typing relation for generics *invariant*. The assignment of variable elst, below, to perlst generates a compiler error because List<Employee> cannot be converted to List<Person>.

```
List<Employee> elst = new ArrayList<Employee>();
List<Person> perlst = elst;
```

This makes sense because if it was valid we could use the add method to add a person to a list of employees and, this is clearly not allowed by the rule of substitution because an instance of Person is not an instance of Employee. (Note that the situation in relation to Arrays is different and we will discuss this in the next section).

But it is reasonable to be able to add lists of elements that are sub-types of a given type E. This is what the bounded wildcards capture. Writing <? extends E> means that it is possible to *get* elements from such a list because they are guaranteed to be of type E or a sub-type of E. Argument lists of this type cannot be modified by the invoking method. Similarly, when we want to write to a list we use <? super E> because it means that all the values in the given list are super types of E. It also means that all you can do is *put* but not *get* values of that type. The use of wildcards introduces what is called *covariant* sub-typing for generics: List<B> is considered a subtype of List<? extends A> if B is a sub-type of A. In relation to super typing they are considerd to be *contravariant*: List<A> is a super type of List<? super B> if B is a sub-type of A. We have seen examples of these when developing the class MyList<E> above.

Arrays

The situation with arrays is quite different. In Java the type of an array is *reified* with its component type. This is the opposite of Collections where the type information is *erazed* at run time. An array of persons has the actual type Person[], whereas an ArrayList<Person> has the raw type ArrayList. The term *reified* simply means that the type is completely represented at run-time. Reification also means that array B[] is a subtype of array A[], if B extends A. The code fragment given below compiles but it will crash, throw an ArrayStoreException, at runtime. From the compilers point of view an array Employee[] is a subtype of array Person[]. Hence, the assignment perarr = emparr is valid. But the assignment on the subsequent line causes a run time exception because an instance of Person is not an instance of Employee.

```
Employee[] emparr = new Employee[10];
Person[] perarr = emparr;
```

```
perarr[0] = new Person("John");
```

To avoid this type of error the programmer has to do type checking at runtime. With generics this is not necessary because it is trapped by the compiler.

Another consequence of reification is that it is not possible to create an instance of a generic array. The statement `E[] f = new E[100]` will fail at compile time because the compiler needs to know the reified (concrete) type of E. Type variables are not reifiable. Reifiable types are any primitive type, non parameterized classes and interfaces, parameterized types where all types are wildcards, raw types or an array where the type is reifiable. All the following are valid: `Integer[] f = new Integer[10]; List<?>[] ls = new List<?>[10]; List[] ls1 = new List[10]`. But `List<Number>[] ln = new List<Number>[10]` is not because erasure removes the type information.

If generic arrays cannot be created then how do we implement the generic `ArrayList` class that uses an array to store reference variables? It turns out that to implement this class we have to create an `Object` array and type cast it to the named generic type parameter. This will cause the compiler to report an unchecked cast warning. We cannot avoid this but it is not a fatal error and we can suppress it in the knowledge that the array will only contain `E` instances. This information is sufficient to ensure type safety. In the class `MyArrayList` we want to be able to sort elements in the array and also find the max and min elements. To allow for this we use the type parameter `<E extends Comparable<E>`. The class also implements the `Iterable` interface. The class has two constructors. A default constructor that creates an array of length 50 and one that takes an argument n denoting the required length of the array. A variable size is initialized to zero and denotes the current size of the array data. The number of elements in the array is given by the current value of size that may not exceed the length of the array. The signature of the class and the two constructors are:

```
class MyArrayList<E extends Comparable<E>> implements Iterable<E>{
  private E[] data;
  int size = 0;
  private static final int DEFAULT_SIZE = 50;
  @SuppressWarnings("unchecked")
  public MyArrayList(){
    data = (E[])new Comparable[DEFAULT_SIZE];
  }
  @SuppressWarnings("unchecked")
  public MyArrayList(int n){
    if(n < 0)
       throw new IllegalArgumentException("Size cannot be negative");
    data = (E[])new Comparable[n];
  }
```

The method add appends a new element to the array allocating additional space when size == data.length. The private method allocateSpace is used to do this. It simply creates a new Comparable array that has space for a number of additional elements. It uses the DEFAULT_SIZE constant for this additional number of spaces. It then uses System.arraycopy to copy the elements from the existing data array to the new one and then updates the reference variable data. The code is:

```
@SuppressWarnings("unchecked")
 private void allocateSpace(){
   E[] ndata = (E[])new Comparable[size + DEFAULT_SIZE];
   System.arraycopy(data,0,ndata,0,size);
   data = ndata;
 }
```

The method remove takes an index, n, as argument, checks that it is within the valid range of indices and stores a reference to it in reference variable el. It uses System.arraycopy to shift elements one position to the left and then sets the rightmost element to null. It updates variable size and returns the reference to the object removed. The code is:

```
public E remove(int n){
   if(n < 0 || n >= size)
      throw new IndexOutOfBoundsException();
   E el = data[n];
   System.arraycopy(data,n+1,data,n,size-n+1); size--;
   data[size] = null;
   return el;
 }
```

There are two sort methods. A default sort that uses the quickSort algorithm to sort the elements based on their natural ordering given by the compareTo method. The second method takes a Comparator as argument and again uses quickSort algorithm to generate an ordered permutation of the values based on the compare method provided by the Comparator.

Listings

The following sections contain listings of both MyList and MyArrayList classes. The test classes for these called MyListTest and MyArrayListTest are both available on Moodle.

Listing of class MyList

```
import java.util.*;
class MyList<E> implements Iterable<E>{
 private Node<E> head = null;
 private Node<E> tail = null;
 private int size = 0;
 public void add(E x){ //add at tail
```

```
    Node<E> nw = new Node<E>(x);
    if(head == null){
        head = nw; tail = nw;
    }
    else{
        tail.setNext(nw); tail = nw;
    }
    size++;
  }
  public boolean contains(E x){
    Node<E> k = head;
    boolean found = false;
    while(k != null && !found){
        E kk = k.data();
        if(x == null && kk == null) found = true;
        else if(x.equals(kk)) found = true;
        // spec in java docs states: o.equals(e)
        else k = k.next();
    }
    return found;
  }
  public void remove(E x){
    Node<E> k = head; Node<E> bk = head;
    boolean found = false;
    while(k != null && !found){
        E kk = k.data();
        if(x == null && kk == null) found = true;
        else if(x.equals(kk)) found = true;
        else{ bk = k; k = k.next();}
    }
    if(found){
        size--;
        if(k == head)
            head = k.next();
        else if(k == tail){
            bk.setNext(null);
            tail = bk;
        }
        else
          bk.setNext(k.next());
    }
  }
  public int size(){
    return size;
  }
```

```
public String toString(){
  if(head == null) return("[]"); //string for empty list
  String s = "[";
  Node<E> k = head;
  while(k.next() != null){
      s = s + k.data().toString()+", ";
      k = k.next();
  }
  s = s + k.data().toString()+"]";
  return s;
}
public void addAll(MyList<? extends E> ls){
  for(E x : ls) this.add(x);
}

public void addAll(Iterable<? extends E> ls){
  for(E x : ls) this.add(x);
}

public void removeAll(MyList<E> ls){
  for(E x : ls) this.remove(x);
}
public void copy(MyList<? super E> ls){
  for(E x : this) ls.add(x);
}

public void copy(Collection<? super E> ls){
  for(E x : this) ls.add(x);
}
public List<E> copy(){
  List<E> ls = new ArrayList<E>();
  for(E x : this) ls.add(x);
  return ls;
}
//Static methods for MyList
//permit copying within the same homogenous ancestral family
public static <T> void copy(MyList<? super T> dst, MyList<? extends T> src){
  for(T x : src) dst.add(x);
}
public static <T> MyList<T> join(MyList<? extends T> l1, MyList<? extends T> l2){
  MyList<T> r = new MyList<T>();
  r.addAll(l1);
  r.addAll(l2);
  return r;
}
```

```
 public static <T extends Comparable<T>> T max(MyList<? extends T> lst){
   // Could use public static <T extends Comparable<? super T>> T max(MyList<?
extends T> lst)
   // recursive bound wildcard because bound on T itself depends on T
   T k = lst.iterator().next();
   for(T el:lst) if(k.compareTo(el) < 0) k = el;
   return k;
 }
 public static <T> T max(MyList<? extends T> lst, Comparator<? super T> comp){
  T k = lst.iterator().next();
   for(T el:lst) if(comp.compare(k,el) < 0) k = el;
   return k;
 }

 public boolean equals(Object ob){
   if(!(ob instanceof MyList)) return false;
   MyList<?> ls = (MyList<?>)ob;
   if(this.size() != ls.size) return false;
   Iterator it = ls.iterator();
   for(E x : this){
      if(!x.equals(it.next())) return false;
   }
   return true;
 }
 public Iterator<E> iterator(){
   return new Iterator<E>(){
      private Node<E> h = head;
      private int index = 0;
      private Node<E> bh = h;
      private int curSize = size;
      public boolean hasNext(){
         return index < curSize;
      }
      public E next(){
         if(index == curSize) throw new NoSuchElementException();
         E item = h.data();
         bh = h; h = h.next(); index++;
         return item;
      }
      public void remove(){
        throw new UnsupportedOperationException();
      }
   };
 }
 private static class Node<E>{
```

```
    E data;
    Node<E> next;
    public Node(E  x){
        data = x; next = null;
    }
    public Node<E> next(){return next;}
    public void setNext(Node<E> p){
        next = p;
    }
    public void set(E x){data = x;}
    public E data(){return data;}
  }
}
```

Listing for class MyArrayList

```
import java.util.*;
class MyArrayList<E extends Comparable<E>> implements Iterable<E>{
  private E[] data;
  int size = 0;
  private static final int DEFAULT_SIZE = 50;
  /*
   *  The MyArrayList only contains instances from add(E)
   *  This ensures type safety.
   *  But runtime of data array will be Object[] not E[]
   */
  @SuppressWarnings("unchecked")
  public MyArrayList(){
    data = (E[])new Comparable[DEFAULT_SIZE];
  }
  @SuppressWarnings("unchecked")
  public MyArrayList(int n){
    if(n < 0)
        throw new IllegalArgumentException("Size cannot be negative");
    data = (E[])new Comparable[n];
  }
  public void add(E x){
    if(size == data.length)
        allocateSpace();
    data[size] = x;
    size++;
  }
  public int size(){return size;}
  public E get(int n){
```

```
  if(n < 0 || n >= size) throw new IndexOutOfBoundsException();
  return data[n];
}
public E remove(int n){
  if(n < 0 || n >= size)
     throw new IndexOutOfBoundsException();
  E el = data[n];
  System.arraycopy(data,n+1,data,n,size-n+1); size--;
  data[size] = null;
  return el;
}
public boolean contains(E x){
  boolean found = false;
  int j = 0;
  while(j < size && !found){
     if(x.equals(data[j])) found = true;
     else j++;
  }
  return found;
}
public void sort(){
  quickSort(data,0,size);
}
public void sort(Comparator<? super E> comp){
  quickSort(data,0,size,comp);
}
public String toString(){
  if(size == 0) return("[]"); //string for empty list
  String s = "[";
  int index = 0;
  while(index < size - 1){
     s = s + data[index].toString()+", ";
     index++;
  }
  s = s + data[index].toString()+"]";
  return s;
}
@SuppressWarnings("unchecked")
private void allocateSpace(){
  E[] ndata = (E[])new Comparable[size + DEFAULT_SIZE];
  System.arraycopy(data,0,ndata,0,size);
  data = ndata;
}
public Iterator<E> iterator(){
  return new Iterator<E>(){
```

```
      private int index = 0;
      public boolean hasNext(){
         return index < size;
      }
      public E next(){
         if(index == size) throw new NoSuchElementException();
         E item = data[index];
         index++;
         return item;
      }
      public void remove(){
        throw new UnsupportedOperationException();
      }
  };
}
public static <T> T max(MyArrayList<? extends T> lst, Comparator<? super T> comp){
  T k = lst.iterator().next();
  for(T el:lst) if(comp.compare(k,el) < 0) k = el;
  return k;
}
private void quickSort(E f[], int p, int q){
    if(q-p <= 1)
       ; //skip
    else{
      int i, j, k;
      // let x = middle element in f[p..q-1]
      E x = f[(p+q)/2];
      i = p; j = p; k = q;
      while(j != k){
        if(f[j].compareTo(x) == 0)
          j = j + 1;
        else if(f[j].compareTo(x) < 0){ //swap f[j] with f[i]
          E temp = f[j];
          f[j] = f[i]; f[i] = temp;
          j = j + 1;i = i + 1;
        }
        else{ // f[j] > x
          // swap f[j] with f[k-1]
          E temp = f[j];
          f[j] = f[k-1]; f[k-1] = temp;
          k = k - 1;
        }
      }
      quickSort(f,p,i);
      quickSort(f,j,q);
```

```
    }
  }
  private void selectSort(E f[], Comparator<? super E> cmp){
  for(int i = 0; i < size; i++){
     int j = i; int k = i + 1;
     while(k < size){
        if(cmp.compare(f[j],f[k]) < 0) j = k;
        k++;
     }
     E temp = f[i]; f[i] = f[j]; f[j] = temp;
  }
  }
  private void quickSort(E f[], int p, int q, Comparator<? super E> cmp){
    if(q-p <= 1)
       ; //skip
    else{
       int i, j, k;
       // let x = middle element in f[p..q-1]
       E x = f[(p+q)/2];
       i = p; j = p; k = q;
       while(j != k){
         if(cmp.compare(f[j], x) == 0)
            j = j + 1;
         else if(cmp.compare(f[j], x) < 0){ //swap f[j] with f[i]
            E temp = f[j];
            f[j] = f[i]; f[i] = temp;
            j = j + 1;i = i + 1;
         }
         else{ // f[j] > x
            // swap f[j] with f[k-1]
            E temp = f[j];
            f[j] = f[k-1]; f[k-1] = temp;
            k = k - 1;
         }
       }
       quickSort(f,p,i,cmp);
       quickSort(f,j,q,cmp);
    }
  }
}
```

Chapter 8: Lambda Expressions, Functions and Predicates

With the release of Java8 in 2014 Oracle followed the lead taken by other modern programming languages by making it possible to write java programs in a functional style. Prior to this release Java supported only imperative state based programming and object-oriented programming. To support this change they introduced two libraries: the functional library and the streams library. Both of these libraries complement each other and together make it possible to write programs in a functional or declarative style. In this chapter we only deal with a brief introduction to the functional library. (For a complete discussion of these libraries see *Programming Paradigms with Java*, 2014 by this author. You may also consult other texts listed in the bibliography at the start of this book.)

Broadly speaking we can view functional programming as a style or paradigm of programming that views computation as the evaluation of mathematical functions. In contrast to the imperative paradigm it avoids the notion of state and mutable data. In practice, the difference between a mathematical function and the notion of a function used in imperative programming is that imperative functions can have side effects that may change the value of program state. Because of this, they lack referential transparency, i.e. the same language expression can result in different values at different times depending on the state of the executing program. Conversely, in functional code, the output value of a function depends only on the arguments that are input to the function. A function always returns the same result when supplied with the same argument values. Eliminating side effects can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Lambda Expressions

A function is a definition that associates a set of input parameters with a single output parameter. It consists of two parts: a signature and an equation that defines its semantics. We write functions using what are called Lambda Expressions (The Lambda Calculus was developed by Alonzo Church in the 1930's to define computable functions). These types of expression are also called *function literals* or *anonymous functions*. Java uses typed lambda expressions that have their origin in functional languages such as Haskell and ML. This approach differs from the untyped lambda calculus usied in Lisp and Scheme. Using typed lambda calculus allows for the possibility of pattern matching (Scala) and the use of strong compile time checking making programs more reliable. Also the use of type inference, where the compiler infers the type from the context, frees the programmer from the need to manually declare types.

The syntax for a lambda expression is a list of parameters, in parentheses, a right arrow, and then the body of the function that may or may not be enclosed in curly brackets. An example of a function literal is:

```
(Integer x) -> x + 1
```

The part preceding the arrow is the parameter list and the part following the arrow is the body of the function. The result type of the expression in the body of the function is its result type. It is inferred from the context and is never specified. Function literals may be defined by a formula or algorithm that tells how to compute the output for a given input. The example given simply uses an arithmetic expression that increments the value of the given argument x. Additional examples of lambda expressions are:

```
(Integer x) -> x % 2 == 0
(Integer x, Integer y) -> x + y
(Integer x) -> x >= 0 ? x : -x
(Integer n) -> {
        int s = 0; for(int j = 0; j < n; j++) s = s +(j+1);
        return s;
}
```

The first one takes a single argument x and compares the remainder on dividing x by 2 with 0 returning one of the boolean values, true or false. The second takes two integer arguments and returns their sum. The third uses a conditional expression to calculate the absolute value of argument x and, the fourth calculates the sum of the first n natural numbers. Notice that the fourth example is enclosed by curly brackets and has an explicit `return` statement. When the definition of a lambda expression requires more than a single expression it must be enclosed by curly brackets. If it evaluates to a given value then this must be explicitly returned. Furthermore, if a lambda expression requires the use of an `if` statement to determine its result then it must return a value for each branch of the `if` statement. For example, the expression `(Integer x) -> {if(x > 0) return true;}` is invalid. It must be written as: `(Integer x) -> {if(x > 0) return true; else return false;}`. It is also possible to write an expression that takes no arguments and may or may not return a value. For example, the expression

```
() -> {for(int j = 0; j < 10; j++) System.out.println(j);}
```

takes no argument and simply executes the action; print the numbers 0 to 9 inclusive. It does not return a value.

In Java to implement lambda expressions we use what are called functional interfaces. In pure functional languages function types are structural but because Java is not a functional programming language they are implemented using specific interfaces where the interface declares its intent (called nominal typing). There are a number of different functional interfaces defined in the package java.util.function and we will discuss the different ones in detail later. For now we use the two most general cases: Function<T,R>, that takes a single argument of type T and has return type R, and BiFunction<T,U,R> that takes two arguments of type T, U and has return type R. Each of these has a single abstract method called apply that is implemented by a given lambda expression.

Using these defined functional interfaces we can give aliases to our function literals by assigning them to variables. The following code block assigns each of our lambda expressions to appropriately named variables. Each one has a single argument and a specific return type.

```
Function<Integer,Integer> inc = x -> x + 1;
Function<Integer,Boolean> even = x -> x % 2 == 0;
Function<Integer,Boolean> pos = x -> x > 0;
Function<Integer, Integer> abs = x -> x >= 0 ? x : -x;
Function<Integer,Integer> sum = n -> {
  int s = 0; for(int j = 0; j < n; j++) s = s +(j+1);
  return s;
};
BiFunction<Integer,Integer,Integer> add = (x,y) -> x + y;
```

At compile time each of these function literals is compiled into a class that when instantiated at run-time is a function value. A function value is an object just like any other object. We invoke the function instance by referencing its apply method as you would do for any other object instance.

The code fragment given below illustrates how to apply these function aliases to actual argument values. The values output on execution of this code are: *8*, *false*, *false*, *7*, *55*, *8*.

```
System.out.println(inc.apply(7));
System.out.println(even.apply(9));
System.out.println(pos.apply(-9));
System.out.println(abs.apply(-7));
System.out.println(sum.apply(10));
System.out.println(add.apply(3,5));
```

We can also write functions that take data structures as arguments and that return information about the state of the given data. The following three functions give examples of how to do

this. The function sumLst takes a list of integer values and calculates the sum of the elements in the list. Each of the functions uses a for each loop to process the data and is coded in an imperative style. (Note that in this chapter all coding of function bodies will be done in an imperative style. We will see in the next chapter that we can re-write many of them in a declarative or functional style using what are called streams.) The function, sumLst, returns 0, if the list is empty. This is correct because from a mathematical perspective the sum of an empty list is defined to be 0. The function allPos takes a list of integer values as argument and returns true if all the elements are positive values; false otherwise. Note that it returns true if the list is empty. The third function, freq, takes both a list of integer values and an integer value as arguments and calculates the frequency of occurrence of the number in the list.

```
Function<List<Integer>,Integer> sumLst = lst ->{
   int s = 0;
   for(Integer x : lst) s = s + x;
   return s;
};
Function<List<Integer>,Boolean> allPos = lst ->{
   for(Integer x : lst) if(x <= 0) return false;
   return true;
};
BiFunction<Integer,List<Integer>,Integer> freq = (k,lst) ->{
   int count = 0;
   for(Integer x : lst) if(k.equals(x)) count++;
   return count;
};
```

Finally, the function getLst demonstrates how to write a function that returns a data structure. It takes an integer n as argument and returns either an empty list, n <= 0, or a list of n values initialized to numbers in the range 1..n.

```
Function<Integer,List<Integer>> getLst = n ->{
   if(n < 0) return new ArrayList<Integer>();
   List<Integer> lst = new ArrayList<Integer>(n);
   for(int j = 0;j < n; j++) lst.add(j+1);
   return lst;
 };
```

The following code fragment uses an assert statement to test each of these functions.

```
List<Integer> dt = new ArrayList<Integer>(Arrays.asList(1,2,3,4,5));
assert sumLst.apply(dt) == 15;
assert allPos.apply(dt) == true;
```

```
assert freq.apply(6,dt) == 0;
assert getLst.apply(5).equals(dt);
```

There is also an identity mapping for Function<T,R>. An identity mapping simply returns its argument value and it is equivalent to the lambda expression x -> x. This method is a static method implemented as part of the interface for Function<T,R>. You can create an alias for it, called id below, or you can use it directly by invoking Function.identity(). The code fragment below shows how to do this.

```
Function<Integer,Integer> id = Function.identity();
assert id.apply(7) == 7;
assert Function.identity().apply(7).equals(7);
assert Function.identity().apply("happy").equals("happy");
```

Special Functions

The package java.util.function provides a number of specialized function types or, more correctly, functional interfaces that when implemented can be used for specific purposes. The table below lists all of these specialized functions and the examples that follow illustrate their application.

The Supplier function takes no arguments and returns a value of type T. The method name for application of this function type is: get. Two examples are given: ran that returns a random decimal value and newLst that returns an empty ArrayList of type integer.

```
Supplier<Double> ran = ()-> Math.random();
Supplier<List<Integer>> newLst = () -> new ArrayList<Integer>();
```

A code fragment to test them might be:

```
Double k = ran.get());
assert newLst.get().size() == 0;
```

The Consumer function takes an argument of type T and has no return value. This means that it effectively accepts a value that it consumes. The method name for application of this function type is: accept. Two examples are given: printS that takes a String as argument and simply prints it on the screen and printSum that takes a list of integers as argument, calculates their sum and outputs the result to the screen.

```
Consumer<String> printS = s -> System.out.println(s);
```

```
Consumer<List<Integer>> printSum = lst ->{
   int s = 0;
   for(Integer x : lst) s = s + x;
   System.out.println("Sum: "+s);
};
```

A code fragment to test them might be:

```
List<Integer> dt = new ArrayList<Integer>(Arrays.asList(1,2,3,4,5));
printS.accept("Happy days are here again");
printSum.accept(dt);
```

The functions UnaryOperator<T> and BinaryOperator<T> both take arguments of the same type T and return instances of T. The method name for application of these functions is: apply. Two examples are given: copy that takes a list of integer values as argument and returns a new list and concat that takes two strings as arguments and returns a concatenated string.

```
UnaryOperator<List<Integer>> copy = lst ->{
   List<Integer> ls = new ArrayList<Integer>();
   for(Integer x : lst) ls.add(x);
   return ls;
};

BinaryOperator<String> concat = (s1,s2) -> s1 + s2;
```

The BinaryOperator function has two static methods called maxBy that returns the greater of two elements according to a specified Comparator and minBy that returns the lesser of two elements according to a specified Comparator. Examples of both are given below. The function minComp takes a two valued lambda expression as argument. This expression uses the Comparator from the Integer class and returns the lesser element of its argument values. The assert statement is used to check the result. The second example firstly creates a Comparator instance defined as a lambda expression and then function maxComp uses it as argument to the maxBy method. Again, the assert statement is used to check the result.

```
BinaryOperator<Integer> minComp = BinaryOperator.minBy(
                                                    (x,y) -> Integer.compare(x,y)
                                                );
 assert minComp.apply(4,7) == 4;

Comparator<Integer> cmp = (x,y) -> Integer.compare(x,y);
```

```
BinaryOperator<Integer> maxComp = BinaryOperator.maxBy(cmp);
System.out.println(maxComp.apply(4,7));
assert maxComp.apply(4,7) == 7;
```

A function Predicate takes a single argument T and a BiPredicate takes two arguments, not necessarily of the same type. Both return a boolean value. The method name for application is: test. Three examples are given:

odd that takes an integer as argument and returns true if it is an odd value, false otherwise;

existsOdd that takes a list of integers returning true if it contains an odd value, false otherwise;

contains that takes an integer and a list of integer as argument and returns true if the given value is contained in the list, false otherwise.

```
Predicate<Integer> odd = x -> x % 2 != 0;
Predicate<List<Integer>> existsOdd = lst ->{
  for(Integer x : lst) if(odd.test(x)) return true;
  return false;
};
BiPredicate<Integer,List<Integer>> contains = (x,lst) -> lst.contains(x);
```

We test these predicates with the following code fragment that uses the assert statement.

```
List<Integer> dt = new ArrayList<Integer>(Arrays.asList(1,2,3,4,5));
assert odd.test(7) == true;
assert existsOdd.test(dt) == true;
assert contains.test(7,dt) == false;
```

Table of Specialized Functions

Function Name	Argument Type	Return Type	Abstract Method Name	Purpose
Supplier<T>	None	T	get	Takes no argument and return a value of type T
Consumer<T>	T	void	accept	Consumes a value of type T
BiConsumer<T,U>	T, U	void	accept	Consumes values of type T and U
UnaryOperator<T>	T	T	apply	A function that takes a

				value of type T as argument and returns a value of type T
BinaryOperator<T>	T, T	T	apply	A function that takes two values of type T as argument and returns a value of type T
Predicate<T>	T	boolean	test	A function that takes a value of type T and returns a boolean value.
BiPredicate<T, U>	T, U	boolean	test	A function that takes two arguments of type T and U and returns a boolean value.

Listing of SimpleFunctionTests

This listing is provided here to allow the reader to see how an application using functions is written. In future cases the completed program files will be made available on Moodle.

```
import java.util.*;
import java.util.function.*;
public class SimpleFunctionTests {
 public static void main(String[] args) {
  //Function aliases with Lambda expressions
  Function<Integer,Integer> inc = x -> x + 1;
  Function<Integer,Integer> dec = x -> x - 1;
  Function<Integer,Boolean> even = x -> x % 2 == 0;
  Function<Integer,Boolean> pos = x -> x > 0;
  Function<Integer, Integer> abs = x -> x >= 0 ? x : -x;
  Function<Integer,Integer> sum = n -> {
     int s = 0; for(int j = 0; j < n; j++) s = s +(j+1);
     return s;
  };
  Function<List<Integer>,Integer> sumLst = lst ->{
     int s = 0;
     for(Integer x : lst)s = s + x;
     return s;
  };
  Function<List<Integer>,Boolean> allPos = lst ->{
     for(Integer x : lst) if(x < 0) return false;
     return true;
  };
  Function<Integer,List<Integer>> getLst = n ->{
     if(n < 0) return new ArrayList<Integer>();
     List<Integer> lst = new ArrayList<Integer>(n);
```

```
   for(int j = 0;j < n; j++) lst.add(j+1);
   return lst;
};
BiFunction<Integer,List<Integer>,Integer> freq = (k,lst) ->{
   int count = 0;
   for(Integer x : lst) if(k.equals(x)) count++;
   return count;
};
//Testing simple functions
System.out.println(inc.apply(7));
System.out.println(even.apply(9));
System.out.println(abs.apply(-7));
System.out.println(sum.apply(10));
List<Integer> dt = new ArrayList<Integer>(Arrays.asList(1,2,3,4,5));
System.out.println(sumLst.apply(dt));

assert sumLst.apply(dt) == 15;
assert allPos.apply(dt) == true;
assert freq.apply(6,dt) == 0;
assert getLst.apply(5).equals(dt);

// End simple function examples ============================================================
//=========================================================================================
//Special types of functional interface
//Supplier takes no argument and returns instance of type T
// method name for application is: get
Supplier<Double> ran = ()-> Math.random();
Supplier<List<Integer>> newLst = () -> new ArrayList<Integer>();

//Consumer takes a single argument type T and has no return type
// method name application is: accept
Consumer<String> print = s -> System.out.println(s);
Consumer<List<Integer>> printSum = lst ->{
int s = 0;
for(Integer x : lst) s = s + x;
System.out.println("Sum: "+s);
};

//BiFunction - a function that takes two arguments of types T, U and returns type R
// T,U,R may be the same or different actual types
//method name for application is: apply
BiFunction<Integer,Integer,Integer> add = (x,y) -> x + y;


//Predicate - a function that takes a single argument T and returns a boolean
// method name for application is: test
Predicate<Integer> odd = x -> x % 2 != 0;
Predicate<List<Integer>> existsOdd = lst ->{
   for(Integer x : lst) if(odd.test(x)) return true;
   return false;
};

BiPredicate<Integer,List<Integer>> contains = (x,lst) -> lst.contains(x);

//UnaryOperator<T> and BinaryOperator<T>
```

```
  //Both take arguments of the same type T and return instances of T
  // method name for application is: apply
  UnaryOperator<List<Integer>> copy = lst ->{
    List<Integer> ls = new ArrayList<Integer>();
    for(Integer x : lst) ls.add(x);
    return ls;
  };

  BinaryOperator<String> concat = (s1,s2) -> s1 + s2;

  //testing special functional interfaces
  System.out.println(ran.get());
  System.out.println(newLst.get().size());
  print.accept("Happy days are here again");
  printSum.accept(dt);

  System.out.println(add.apply(4,6));
  List<Integer> dtlst = new ArrayList<Integer>(Arrays.asList(1,2,2,2,5,2));
  System.out.println("Frequency of 2: "+freq.apply(2,dtlst));

  System.out.println(odd.test(35));
  System.out.println(existsOdd.test(dtlst));
  System.out.println(contains.test(5,dtlst));

  List<Integer> ndt = copy.apply(dtlst);
  System.out.println(ndt);
  System.out.println(concat.apply("Big", "Deal"));
  System.out.println(contains.test(5,ndt));
}
```

Higher-Order Functions

Functions that take other functions as parameters or that return them as results are called *higher-order* functions. These types of functions provide a very flexible mechanism for program composition. In fact, Hughes, in *Why Functional programming Matters*, argues that *higher-order* functions are one of the fundamental types of *glue* that functional programming provides.

In Java function values are objects and, as such, may be passed as arguments to functions. To illustrate this we begin by defining a general purpose higher-order function that takes two arguments: a function that itself takes an integer argument and returns an integer, and an integer argument. This higher-order function is a `BiFunction` because it takes two arguments: the `Function<Integer,Integer>` and its argument `Integer`. The function is aptly called `hFunc` and its definition is given as:

```
BiFunction<Function<Integer,Integer>,Integer,Integer> hFunc =
   (f,x)-> f.apply(x);
```

The function simply returns the result of invoking function f with argument x. The result returned will depend on the meaning of function f. To see how it works we write some functions that take an integer argument and return an integer. These functions are written as lambda expressions simplifying their definition. It is also possible to define the functions separately and pass the function variable as a parameter. This is illustrated in the case of intSqrt given below. The meaning of each of these functions should be self explanatory from the context and the assert statement is used to check their correctness.

```
assert hFunc.apply(x -> x + 1, 3) == 4;
assert hFunc.apply(x -> x + 1, 10) == 11;
assert hFunc.apply(x -> 2 * x + 10, 3) == 16;
assert hFunc.apply(x -> x < 0 ? -x : x, -5) == 5;
assert hFunc.apply(intSqrt,15) == 3;
```

The function intSqrt takes an integer as argument and returns its integer square root. It avoids throwing an exception in the case where x is negative by converting it to its absolute value. The function is defined as follows:

```
Function<Integer,Integer> intSqrt = x ->{
      int y = x <= 0 ? -x : x;
      int sq = 0;
      while((sq+1)*(sq+1) <= y) sq++;
      return sq;
  };
```

ArrayList Methods that use Functions as Arguments

In Java 8 four new methods have been added to the ArrayList class. These methods are higher order methods because they take specialized functions as arguments and apply them automatically to all the elements in the given list. The methods together with a brief description of their semantics are listed in the table below.

Return type	Method	Meaning
void	forEach(Consumer<? super E> action)	Applies the given action function to all the elements in the list in order.
boolean	removeIf(Predicate<? super E> filter)	Removes all values that satisfy the given predicate filter
void	replaceAll(UnaryOperator<E> op)	Replaces each element of this list with the result of applying the

		operator function op to that element.
void	sort(Compaparator<? super E> cmp)	Sorts this list according to the order specified by the given Comparator cmp.

The method forEach provides an internal iterator that takes a Consumer function as argument and applies this function to each element in the sequence provided by the internal iterator for the class. It removes the necessity to write an external loop. The externally coded loop for(Integer x : lst) System.out.print(x+" ") can be replaced with lst.forEach(x->System.out.print(x+" ")). Because this method takes a Consumer function as argument it frees the user from writing the iteration code and allows the programmer to focus on what they want the consumer to do. The program listed below gives an example of doing just this. We write a consumer function, printEven, that takes an integer as argument and prints it only if it is an even number. This function is then passed as argument to the forEach method.

```
Consumer<Integer> printEven = x->{
    if(x % 2 == 0) System.out.print(x+" ");
};
lst.forEach(printEven);
```

The removeIf method is also a higher order method because it takes a Predicate function as argument and removes from the given list only those values that satisfy the given predicate. To remove all values greater than 10 we simply write lst.removeIf(x -> x > 10).

Method replaceAll takes a UnaryOperator function as argument and applies it to each element in the list. This method has the side effect of modifying each value in the given list.

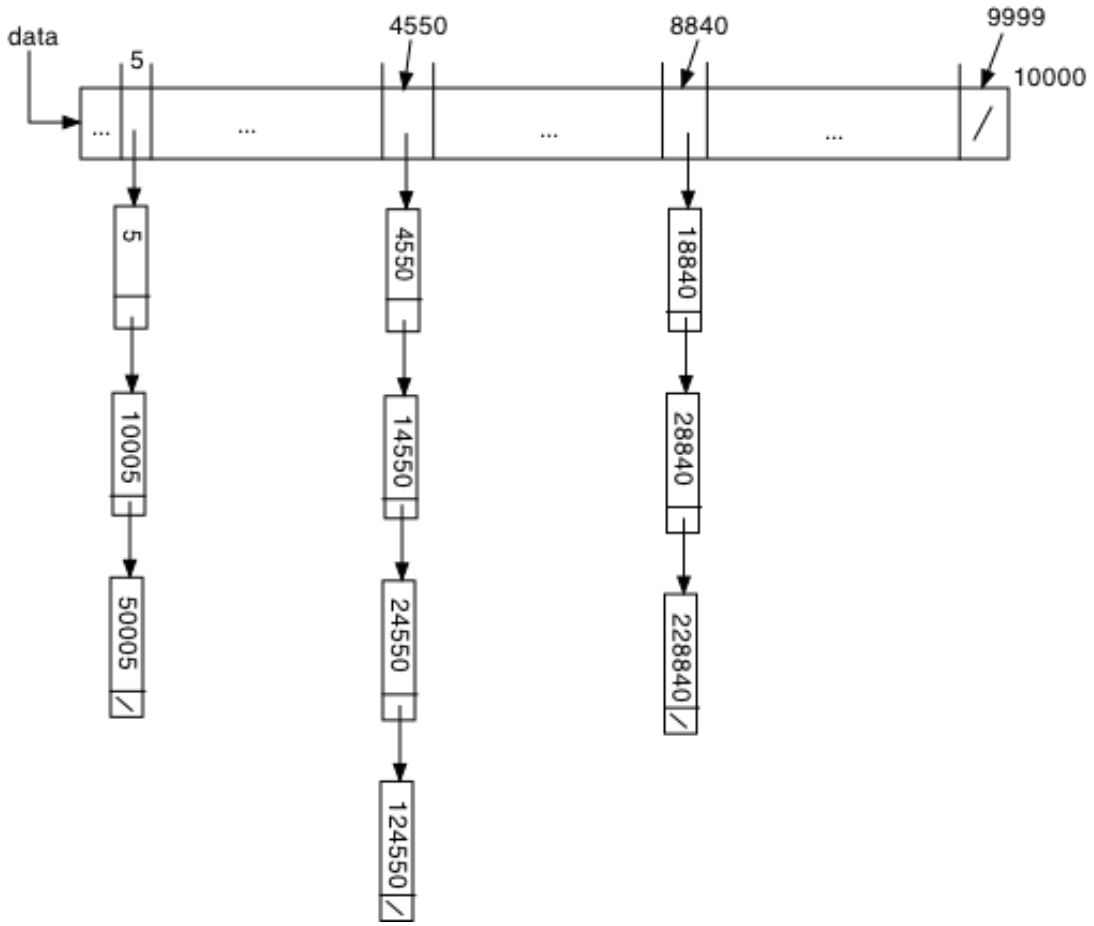
Finally, the sort method takes a Comparator and sorts the elements in order defined by the compare method provided by it. The listing below gives two examples of sorting a list in *natural order* (<=) and by >=.

```
import java.util.*;
import java.util.function.*;
class ArrayListTest{
 public static void main(String args[]){
   ArrayList<Integer> lst = new ArrayList<>(Arrays.asList(2,3,5,7,8,10,21,11));
   //examples of forEach
   lst.forEach(x->System.out.print(x+" "));
```

```
    System.out.println();
    Consumer<Integer> printEven = x->{
        if(x % 2 == 0) System.out.print(x+" ");
    };
    lst.forEach(printEven);
    System.out.println();

    //example of removeIf
    lst.removeIf(x->x>10);
    System.out.println(lst);

    //example of replaceAll
    lst.replaceAll(x-> x*x);
    System.out.println(lst);

    //examples of sort with comparator
    lst.sort((x,y)->Integer.compare(x,y));
    System.out.println(lst);

    Comparator<Integer> gt = (x,y) -> -1*Integer.compare(x,y);
    lst.sort(gt);
    System.out.println(lst);
  }
}
```

Chapter 9: Hashing

Suppose you have a very large collection of random positive integer values and you want to store them in a data structure so that the cost of insertion and retrieval is optimal. By optimal we mean *O(1)*. We could store the values in a linear array chronologically. This would mean that insertion is *O(1)* but all searching would be linear and, hence, *O(n).* We could explore keeping the data sorted. This would mean that searching is *O(log n)* but insertion is now *O(n)* for each value giving an overall performance of $O(n^2)$. We seem to be stuck on the horns of a dilemma. By optimising insertion we make searching expensive and vice versa. To try to get out of this dilemma we explore the possibility of constructing a data structure that uses an array of buckets where each bucket contains a very small number of elements. The idea is to choose an array of size N, where N is a factor of the total number of values in the collection. Suppose the number of integer values is 100,000 we might choose N to be 10,000. This would mean that on average each bucket would contain 10 elements, assuming we can distribute the values evenly over the 10,000 buckets. As a first attempt to do this we consider using *modulo* arithmetic by observing that, given integer x an element of the collection, x % 10,000 always gives a value in the range 0..9,999. These are the indices in the array of buckets. From this we conclude that x % 10,000 is the index of the bucket to store x. All values mapping to the same index will be stored in the same bucket. A snapshot of the array of buckets might be:



From the diagram you can see that all values that have a remainder of 5 are stored in a list whose reference value is data[5]. Similarly, values that have a remainder of 4550 are stored in list data[4550], those with a remainder of 8840 in list data[8840], etc. Notice that the value at data[9999] is null indicating that none of the elements in the collection had a remainder 9999. Using this data structure will optimise the cost of both insertion and retrieval and give an *O(1)* solution if we can guarantee that the size of any of the sublists does not grow linearly. The cost of insertion is always *O(1)* but the cost of retrieval could become *O(n)* if all the values had the same remainder. We have to ensure that this cannot happen. Given the guarantee that the numbers in the collection are randomly distributed in value and using an array of size 10000 gives, in the optimum case, a bucket(list) size of at least 10. As we can see from the example it is possible that some lists may be empty. Therefore, in practice we would be happy with a maximum bucket size of some multiple of the base number 10. The ideal would be to get 100% bucket usage and a maximum bucket size of some small multiple of 10.

This type of data structure is called a ***HashTable*** that uses ***buckets or chaining***. The function used to map values to their index position is called a ***hashing function***. The hashing function

that we are using for this example is x % 10000. The other important point to note is that we do not allow duplicate values in the table. This will be enforced by the add method.
(Note: there is also an alternative approach called *open addressing*. We will not discuss this approach here.)

To test out our analysis we begin by implementing a generic hash table that uses generic lists to store elements that map to the same index value.

The class has a single private attribute data array of type GLinkedList<E>.
The constructor for the class HashList<E extends Comparable<E>> takes an integer n, denoting the size of the table, as argument. We assume n positive. It creates an array of empty LinkedList<E> lists. The coding of the constructor gets around the problem of creating a generic array with the use of type casting. The code is:

```
private LinkedList<E> data[];
@SuppressWarnings("unchecked")
public HashList(int n){
    data = (LinkedList<E>[])(new LinkedList[n]);
    for(int j = 0; j < data.length;j++)
        data[j] = new LinkedList<E>();
}
```

The private method hashC takes an instance of type E as argument and uses its hashCode method to calculate the index of the list to store x in. The code is:

```
private int hashC(E x){
    int k = x.hashCode();
    int h = Math.abs(k % data.length);
    return(h);
}
```

For this to work type E must override the hashCode method in the base class Object. In Java the classes Integer, Double, Character, Byte, String override the base method and provide their own hash code. In all cases the number returned is an integer value in the range Integer.MIN_VALUE .. Integer.MAX_VALUE. This means that it can be negative. Hence, the need to use Math.abs to return a positive value. (**Note** that Math.abs(Integer.MIN_VALUE) returns -2147483648, i.e. Integer.MIN_VALUE. We avoid this possibility by calculating the remainder first). See below for a discussion of hashing functions in *Java*.

The public method add takes an instance of type E as argument, uses the private method hashC to calculate its index in the table and then adds it to the list. We do not allow null values in our hash table. (The Java specification for hash codes returns 0 for null values but we choose to ignore them.)The code for add is:

```
public void add(E x){
  if(x != null){
        int index = hashC(x);
        if(!data[index].contains(x))
         data[index].add(x);
    }
}
```

The method contains uses the hashC function to determine the index for argument x and returns data[index].contains(x). Similarly, remove returns data[index].remove(x). The code for these methods is:

```
public boolean contains(E x){
    if(x == null) return false;
    int index = hashC(x);
    return(data[index].contains(x));
}
public boolean remove(E x){
 if(x == null) return false;
 int index = hashC(x);
 return data[index].remove(x);
}
```

The class has a toString method with the usual semantics and also method displayLists that prints a fomatted list of each sub-list in the table. The class implements the Iterable interface and, hence, provides an iterator method that uses an ArrayList to make a copy of the data references in the table and then returns an iterator for it. This simplifies the problem of iterating over an iteration. The code is:

```
public Iterator<E> iterator(){
  ArrayList<E> items = new ArrayList<E>();
   int ind = 0;
   while(ind < data.length){
      Iterator<E> it = data[ind].iterator();
      while(it.hasNext())
         items.add(it.next());
```

```
            ind++;
        }
        return items.iterator();
    }
```

(The complete class is given in the listings section at the end of this chapter)

We can now solve our original problem by creating an integer hashlist and adding 100000 values to it. To ensure that the data is random we use a random number generator. The code is:

```
HashList<Integer> list = new HashList<Integer>(10000);
for(int j = 0; j < 100000; j++){
  int x = (int)(Math.random()*1000000);
  list.add(new Integer(x));
}
```

But there is no guarantee that a high percentage of the buckets are used nor is there a guarantee that all lists have a small number of elements. To check this we now insert some additional methods that provide information about the table. The method percentUsed calculates the percentage of used buckets by counting the number of buckets of length greater than zero. The code is:

```
public double percentUsed(){
    int count = 0;
    for(int j = 0; j < data.length; j++){
        if(data[j].size() > 0)
          count++;
    }
    double p = count *100.0 / data.length;
    return p;
}
```

The method largestBucket returns the length of the longest list in the table.

```
public int largestBucket(){
    int max = 0;
    for(int j = 0; j < data.length; j++)
        if(data[j].size() > max) max = data[j].size();
    return max;
}
```

Similarly, the method smallestBucket returns the length of the smallest list.

```
public int smallestBucket(){
    int min = data[0].size();
    for(int j = 1; j < data.length; j++)
        if(data[j].size() < min) min = data[j].size();
    return min;
}
```

The method listSizes calculates the frequency of all the different list lengths from 0 up to, and including, the largest one.

```
public int[] listSizes(){
    int n = this.largestBucket();
    int d[] = new int[n+1];
    for(int j = 0; j < d.length; j++) d[j] = 0;
    for(int j = 0; j < data.length; j++){
        int m = data[j].size();
        d[m] = d[m] + 1;
    }
    return d;
}
```

Method empty simply counts the number of empty lists.

```
public int empty(){
    int count = 0;
    for(int j = 0; j < data.length; j++)
        if(data[j].size() == 0) count++;
    return count;
}
```

Incorporating these test methods into our HashList class we get the following test program.

```
public static void main(String[] args) {
    HashList<Integer> list = new HashList<Integer>(10000);
    for(int j = 0; j < 100000; j++){
    int x = (int)(Math.random()*1000000);
    list.add(new Integer(x));
    }
    //list.displayLists();
    System.out.println("Percentage of buckets used: "+list.percentUsed());
    System.out.println("Largest bucket size =  "+list.largestBucket());
    System.out.println("Smallest bucket size =  "+list.smallestBucket());
    int lSizes[] = list.listSizes();
```

```
    System.out.println("Frequency list");
    for(int j = 0; j < lSizes.length; j++){
      if(lSizes[j] > 0)
        System.out.printf("Buckets with %d elements = %d\n",j,lSizes[j]);
    }
    System.out.println("Empty buckets = "+list.empty());
}
```

Typical output from a test run of this program is:

Percentage of buckets used: 100.0
Largest bucket size = 24
Smallest bucket size = 1
Frequency list
Buckets with 1 elements = 7
Buckets with 2 elements = 19
Buckets with 3 elements = 75
Buckets with 4 elements = 179
Buckets with 5 elements = 373
Buckets with 6 elements = 612
Buckets with 7 elements = 917
Buckets with 8 elements = 1148
Buckets with 9 elements = 1250
Buckets with 10 elements = 1285
Buckets with 11 elements = 1134
Buckets with 12 elements = 949
Buckets with 13 elements = 702
Buckets with 14 elements = 536
Buckets with 15 elements = 335
Buckets with 16 elements = 196
Buckets with 17 elements = 132
Buckets with 18 elements = 75
Buckets with 19 elements = 33
Buckets with 20 elements = 21
Buckets with 21 elements = 11
Buckets with 22 elements = 5
Buckets with 23 elements = 5
Buckets with 24 elements = 1
Empty buckets = 0

We see that all the buckets were used and that the max list size is 24. However, the really interesting result is the almost perfect bell curve shape of the frequency of bucket size. The modal list size is 10 and 92.4% of list sizes have between 5 and 15 elements. Lists of this size guarantee that retrieval is *O(1)*.
Testing with a small list of 100 random values distributed over 10 buckets yielded similar results. In this case the actual sub-lists are printed.

```
[170, 760, 0, 700, 940, 380, 970, 290, 780]
[511, 471, 391, 191, 881, 81, 161]
[892, 32, 322, 272, 282, 782, 102, 602, 192]
[223, 723, 253, 873, 533, 743, 113, 213, 813, 63, 363, 363, 573, 473]
[404, 614, 4, 94, 614, 144, 754, 184, 434, 514, 654, 784]
[385, 375, 165, 315, 795, 385, 385, 715, 915, 975, 665, 235]
[976, 26, 436, 226, 866, 636, 896, 246, 966, 776, 346]
[497, 397, 457, 817, 947, 937, 197, 197, 187, 197, 847]
[698, 178, 148, 198]
[49, 379, 429, 759, 429, 19, 979, 959, 719, 309, 639]

Percentage of buckets used: 100.0
Largest bucket size =  14
Smallest bucket size =  4
Frequency list
Buckets with 4 elements = 1
Buckets with 7 elements = 1
Buckets with 9 elements = 2
Buckets with 11 elements = 3
Buckets with 12 elements = 2
Buckets with 14 elements = 1
Empty buckets = 0
```

Finally, to find the maximum or minimum element in a hash table requires iteration over the entire collection. This can be accomplished using the iterator method to retrieve a reference to all elements in sequence or by writing public methods that return the actual min or max values.

It is also important to keep in mind that hash tables require additional memory to hold pointers that link elements in the individual lists. This overhead can double the memory required if the actual data elements have byte sizes similar to that required to hold a memory address.

The class developed here uses a fixed sized array where the size is passed as an argument to the constructor. If they should choose a small value and then add a large number of elements the performance would be degraded because the individual lists would become large. This would make `add` O(n). To avoid this situation the class should use a dynamic array that grows when thelargest list reaches some critical value. This means re-distributing all the values in the table by re-calculating their hashCodes. Implementing such a solution is left as an exercise and will be explored further in the exercises.

Hashing Functions in Java

There are three primary requirements that must be met by any implementation of a good hashing function for a given type. These are:

1. It must be *deterministic*. This means that equal elements must return the same hash value. In Java this means that given two instances of the same type `b1`, `b2` such that if `b1.equals(b2)`, then `b1.hashCode()` must equal `b2.hashCode()`. It is **not** a requirement that if two hash codes are equal both instances are equal.

2. It should be implemented so that it is *efficient to compute*.

3. It should *uniformly distribute the keys* so that data is distributed over the whole table, hence, minimizing the size of lists. The use of modulo arithmetic in the implementation of the hash function guarantees this as long as hash values are randomly distributed.

Every class in Java inherits a `hashCode` from class `Object`. However, this code is based on the memory address of the object and not on its content and, hence, is not very useful. All the primitive wrapper classes – `Integer`, `Double`, `Character`, `Boolean` – override this inherited method and provide their own. This is also the case for class `String`. The `hashCode` method for class `Integer` returns the encapsulated integer value, e.g. if `x = new Integer(-100)`, then `x.hashCode()` returns `-100`. The hash code for class Double is *the exclusive OR of the two halves of the long integer bit representation, exactly as produced by the method doubleToLongBits(double), of the primitive double value represented by this Double object.* The implementation is:

```
int hashCodeDouble(Double d){
   long v = d.doubleToLongBits(d.doubleValue());
   int h = (int)(v^(v>>>32));
```

```
    return h;
}
```

The hash code for class Character is the ordinal value of the encapsulated character, e.g. if `ch = new Character('B')`, then `ch.hashCode()` yields `66`.

The implementation for class `String` is:

```
int hashCode(String s) {
    int hash = 0;
   for (int j = 0; j < s.length(); j++)
       hash = (hash * 31) + s.charAt(j);
    return hash;
 }
```

This function takes the position of the characters in the string `s` into account when it is calculating a hash code. The constant `31` is a Mersenne prime – a prime number that is one less than a power of 2 – and it has been shown that it satisfies the distribution constraint listed above very well. We should point out that the return value may be negative. For example, if `s = "Happy days are here again"`, then `s.hashCode() = -266512891` and the string *polygenelubricants* yields a hash value of `Integer.MIN_VALUE`.

Class `Boolean` is implemented as follows:

```
int hashCode(boolean value) {
    return value ? 1231 : 1237;
}
```

The constants 1231 and 1237 are arbitrary prime numbers.

All of these primitive wrapper classes meet the equality constraint listed above (requirement 1). This is crucial from a search perspective.

When writing our own `hashCode` methods for user defined types it is, in practice, a good idea to use these primitive wrapper hash code methods in the calculation because they guarantee the distribution constraint listed above. However, it is essential that the *equality* constraint is not ignored. Consider the following Book class.

```
class Book implements Comparable<Book>{
   private String title;
```

```
    private String author;
    private double price;
    public Book(String a, String t, double p){
        title = t; author = a; price = p;
    }
    public String title(){return title;}
    public String author(){return author;}
    public int compareTo(Book bk) throws ClassCastException{
        if(bk == null){return -1;}
        if(author.compareTo(bk.author) == 0)
                return(title.compareTo(bk.title));
        else
            return(author.compareTo(bk.author));
        }
    }
    public boolean equals(Object ob){
        if(!(ob instanceof Book)) return false;
        Book b = (Book)ob;
        return title.equals(b.title)&& author.equals(b.author));
    }
    public int hashCode(){
        Double p = new Double(price);
        int h = title.hashCode() + author.hashCode()+p.hashCode();
        return(h);
    }
    public String toString(){
        return title+" by "+ author;
    }
}
```

The class implements both the **equals** method and the **hashCode** method. The **hashCode** method combines the primitive wrapper class hash codes when calculating its own hash code. This satisfies the distribution requirement. However, it fails to satisfy the equality requirement that insists that if two instances of a type are equal then their respective hash codes must also be equal. To illustrate this we consider the following code fragment:

```
public class BookTestHash {
    public static void main(String[] args){
        Book b1 = new Book("Doyle","Paddy Clarke", 10.0);
        Book b2 = new Book("Doyle","Paddy Clarke", 6.50);
        System.out.println(b1.equals(b2));
        System.out.println(b1.hashCode() + " " + b2.hashCode());
    }
}
```

On executing this code you will find that books b1 and b2 are deemed equal but they have different hash codes: -1671985325 and -1672640685. This occurs because our implementation fails to satisfy the equality requirement. To satisfy it simply change the hashCode method to:

```
public int hashCode(){
    int h = title.hashCode() + author.hashCode();
    return(h);
}
```

Implementing Sets

A set is a collection of things that does not allow duplicates and imposes no ordering of items. Two sets are said to be equal if and only if they contain the same values. In Mathematics there are a small number of operations that we apply to sets.

Given two sets A, B:

$A \cup B$ (union) = the set containing all elements of A and B, duplicates removed.

$A \cap B$ (intersection) = the set containing those elements common to A and B.

$A - B$ (difference) = the set containing those elements in A not contained in B.

For example,

$A = \{1, 3, 5, 6\}$, $B = \{2, 3, 6, 8\}$

$A \cup B = \{1, 3, 5, 6, 2, 8\}$

$A \cap B = \{3, 6\}$

$A - B = \{1, 5\}$

We also define a subset of a set A as a set containing only elements of A. For example,

$\{2, 6\} \subseteq B$.

An implementation of sets must support all of these operations.

Because the collections contain a class HashSet<E> we will implement a class called MyHashSet<E> to store elements for a set and implement the set operations defined above. This class will not attempt to implement all the methods implemented by HashSet<E>. An instance of the HashList developed above is used to store elements in the set. There are three constructors. The default constructor imposes a limit of Integer.MAX_VALUE on the size of the set. The choice of 100 lists for the default set is arbitrary. The second constructor takes the max size of the set as argument and constructs a HashList based on the size of n. The

third constructor makes it possible to make new sets based on an given Collection from the Collection classes.

```
public class MyHashSet<E extends Comparable<E>> implements Iterable<E> {
   private HashList<E> data;
   private int maxSize = Integer.MAX_VALUE;
   private int size = 0;
   public MyHashSet() {
       data = new HashList<E>(100);
   }
   public MyHashSet(int n){
       if(n > 1000) //divide n by 10
          data = new HashList<E>(n/10);
       else
          data = new HashList<>(100);
       maxSize = n;
   }
   public MyHashSet(Collection<? extends E> lst){
       data = new HashList<E>(lst.size());
       for(E x : lst) data.add(x);
   }
…
}
```

Method add first checks the size and then adds it. It returns a boolean indicating success or failure. It implements that rule that duplicate elements are not allowed.

Methods contains, remove, size, toString and iterator should all be self explanatory at this stage.

The intersection of two sets is a set containing those elements common to both. The method intersection, defined below, takes an instance of HashSet<E> as argument and returns a new set that is the intersection of itself and its argument set. It uses a set, in, that is assigned elements common to both sets. An for loop is used to iterate over the elements in set st and if an element is contained in the given data set it is added to the return set, in. The code is:

```
   public MyHashSet<E> intersection(MyHashSet<E> st){
     MyHashSet<E> in = new MyHashSet<>();
     for(E x : st)
         if(data.contains(x)) in.add(x);
     return in;
   }
```

The union of two sets is a set containing all elements in both sets. The method union, defined below, takes an instance of HashSet<E> as argument and returns a new set that is the union of itself and its argument set. It uses a set, un, that is assigned elements from both sets. A for loop is used to iterate over the elements in both set st and the data set, in sequence, adding elements to un as they iterate. The code is:

```
public MyHashSet<E> union(MyHashSet<E> st){
   MyHashSet<E> un = new MyHashSet<>();
   for(E x : st) un.add(x);
   for(E x : data) un.add(x);
   return un;
}
```

Completing methods that implement set difference and subset are left as exercises and solutions are provided in the listings given below.

A simple program to test the HashSetTest class is given below.

```
import java.util.*;
public class HashSetTest{
   public static void main(String args[]){
      MyHashSet<Integer> s1 = new MyHashSet<>(Arrays.asList(2,4,6,8));
      System.out.println(s1);
      MyHashSet<Integer> s2 = new MyHashSet<>(Arrays.asList(2,3,6,9));
      System.out.println(s2);
      MyHashSet<Integer> s3 = s1.intersection(s2);
      System.out.println(s3);
      MyHashSet<Integer> s4 = s1.union(s2);
      System.out.println(s4);
      MyHashSet<Integer> s5 = s1.difference(s2);
      System.out.println(s5);
      System.out.println(s1.subSet(s3));

   }
}
```

Appendix

This appendix lists the code for both the HashList class and the MyHashSet class.

```
import java.util.*;
public class HashList<E extends Comparable<E>> implements Iterable<E>{
    private LinkedList<E> data[];
    @SuppressWarnings("unchecked")
    public HashList(int n){
        data = (LinkedList<E>[])(new LinkedList[n]);
        for(int j = 0; j < data.length;j++)
            data[j] = new LinkedList<E>();
    }
    @SuppressWarnings("unchecked")
    public HashList(Collection<? extends E> cl){
        data = (LinkedList<E>[])(new LinkedList[cl.size()]);
        for(int j = 0; j < data.length;j++)
            data[j] = new LinkedList<E>();
        for(E x : cl) this.add(x);
    }
    private int hashC(E x){
        int k = x.hashCode();
        //an alternative is to mask the minus using
        //int k = x.hashCode() & 0x7fffffff;

        int h = Math.abs(k % data.length);
        return(h);
    }
    public void add(E x){
      if(x != null){
            int index = hashC(x);
            if(!data[index].contains(x))
             data[index].add(x);
        }
    }
    public boolean contains(E x){
        if(x == null) return false;
        int index = hashC(x);
        return(data[index].contains(x));
    }
    public boolean remove(E x){
     if(x == null) return false;
     int index = hashC(x);
     return data[index].remove(x);
```

```
}
public void displayLists(){
    for(LinkedList<E> k : data){
     if(k.size() > 0)
          System.out.println(k);
    }
}
public String toString(){
    StringBuffer s = new StringBuffer(this.size());
    s.append('<');
    int ind = 0;
    while(ind < data.length){
        Iterator<E> it = data[ind].iterator();
        while(it.hasNext())
            s.append(it.next()+", ");
        ind++;
    }
    s.deleteCharAt(s.length()-1);
    s.setCharAt(s.length()-1,'>');
    return s.toString();
}
public int size(){
    int j = 0;
    for(LinkedList<E> lst : data) j += lst.size();
    return j;
}
public int empty(){
    int count = 0;
    for(int j = 0; j < data.length; j++)
        if(data[j].size() == 0) count++;
    return count;
}

public double percentUsed(){
    int count = 0;
    for(int j = 0; j < data.length; j++){
        if(data[j].size() > 0)
          count++;
    }
    double p = count *100.0 / data.length;
    return p;
}
public int largestBucket(){
    int max = 0;
    for(int j = 0; j < data.length; j++)
```

```
            if(data[j].size() > max) max = data[j].size();
        return max;
    }
    public int smallestBucket(){
        int min = data[0].size();
        for(int j = 1; j < data.length; j++)
            if(data[j].size() < min) min = data[j].size();
        return min;
    }
    public int[] listSizes(){
        int n = this.largestBucket();
        int d[] = new int[n+1];
        for(int j = 0; j < d.length; j++) d[j] = 0;
        for(int j = 0; j < data.length; j++){
            int m = data[j].size();
            d[m] = d[m] + 1;
        }
        return d;
    }

    public Iterator<E> iterator(){
      ArrayList<E> items = new ArrayList<E>();
      int ind = 0;
      while(ind < data.length){
            Iterator<E> it = data[ind].iterator();
            while(it.hasNext())
                items.add(it.next());
            ind++;
       }
       return items.iterator();
    }
}
```

```
import java.util.*;
public class MyHashSet<E extends Comparable<E>> implements Iterable<E> {
   private HashList<E> data;
   private int maxSize = Integer.MAX_VALUE;
   private int size = 0;
   public MyHashSet() {
      data = new HashList<E>(100);
   }
   public MyHashSet(int n){
      //divide n by 10
      if(n > 1000)
         data = new HashList<E>(n/10);
      else
         data = new HashList<>(100);
      maxSize = n;
   }
   public MyHashSet(Collection<? extends E> lst){
      data = new HashList<E>(lst.size());
      for(E x : lst) data.add(x);
   }
   public boolean add(E x){
      if(size == maxSize) return false;
      data.add(x);
      size++;
      return true;
   }
   public boolean contains(E x){
      return data.contains(x);
   }
   public boolean remove(E x){
      if(size == 0) return false;
      if(data.remove(x)){
         size--;
         return true;
      }
      else return false;
   }
   public int size(){ return size;}
   public String toString(){
      return data.toString();
   }
   public Iterator<E> iterator(){
      return data.iterator();
   }
   public MyHashSet<E> intersection(MyHashSet<E> st){
```

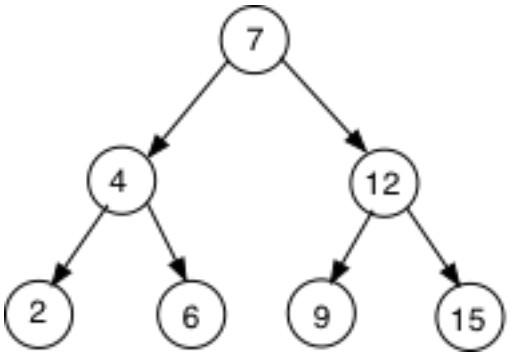
```
      MyHashSet<E> in = new MyHashSet<>();
      for(E x : st)
           if(data.contains(x)) in.add(x);
      return in;
   }
   public MyHashSet<E> union(MyHashSet<E> st){
       MyHashSet<E> un = new MyHashSet<>();
       for(E x : st) un.add(x);
       for(E x : data) un.add(x);
       return un;
   }
   public MyHashSet<E> difference(MyHashSet<E> st){
      MyHashSet<E> diff = new MyHashSet<>();
      for(E x : data)
       if(!st.contains(x))
            diff.add(x);
      return diff;
   }
   public boolean subSet(MyHashSet<E> st){
       for(E x : st)
          if(!data.contains(x)) return false;
       return true;
   }
}
```

Chapter 10: Binary Search Trees

Given an ordered linked list of items the only way to search for a given item is to perform a linear search. This is *O(n)*, where n is the number of items in the list. One might ask the question: is it possible to rearrange the nodes so that a binary search of the data list is possible? This would give a search time of *O(log n)*. Consider the following sorted list of values.

```
list = [ 2,4,6,7,9,12,15]
```

Applying a binary search, for a given value `x`, to this list we can model the possible comparisons as a tree rooted at the middle value `7`. If `x <= 7`, then we compare `x` with `4`; else compare `x` with `12`. This is `O(log n)`. To keep this optimal cost of searching we could store the items in the structure of the comparison tree itself. Building trees that organize, or structure, data in this way is the topic for discussion.



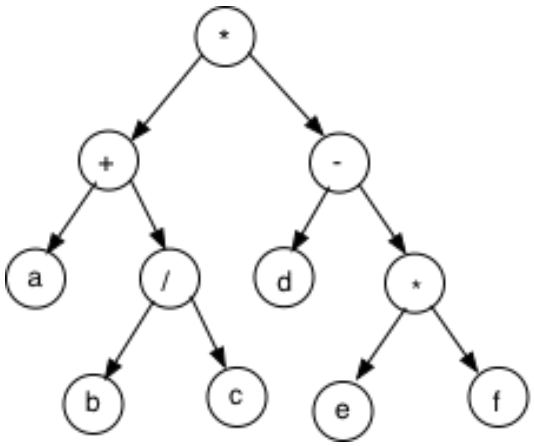
Definitions

A **binary tree** is a finite set of nodes that either is *empty* or consists of a *root* node with two disjoint binary trees called the *left* and *right* sub-trees of the root.

A *node* is an element in the tree. Each node is connected to its parent and descendents by arcs called *edges*. Nodes are either terminal, in which case they have no descendents (called *leaves*), or not terminal, in which case they have descendents and are called interior nodes. The number of direct descendents of an interior node is called its *degree*. In the case of binary trees the degree is always 2. The number of edges that have to be traversed from the root node to a node k is called the *path length* or *height* of k. In general, a node at level j has path length j-1, assuming path length of the root node is 0. The *height* of a tree is the maximum path length associated with any of its nodes. The tree given above has `4` *leaf nodes* (`2,6,9,15`), 2 *interior nodes* (`4, 12`) and *root* `7`. Its *height* is `2` and the *height* of `12` is `1`, given that *root* has a *height* of `0`.

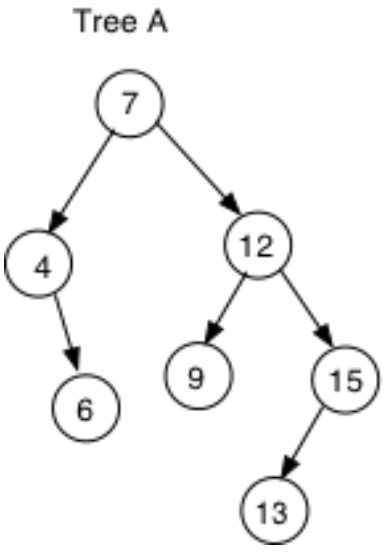
Examples of binary trees are:

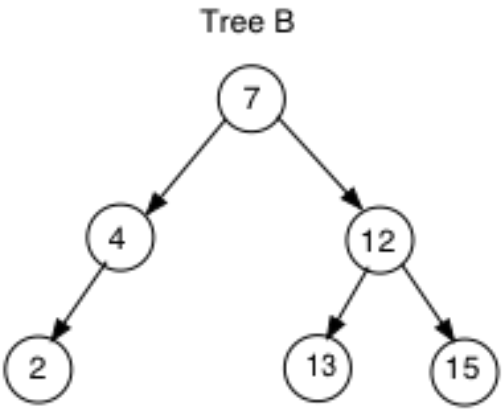
- A family tree inverted, where parents are viewed as descendents;
- The history of a soccer tournament where the winner is the root;
- An arithmetic expression with binary operators, where operands appear as *leaf* nodes, e.g. the tree representation of `(a + b/c) * (d - e*f)` is



Binary Search Tree

A binary search tree is a binary tree with the additional constraint that the *value in every node must be greater than all values in its left sub-tree, and less than all values in its right sub-tree*. Duplicate values are not allowed. In the diagrams below the tree on the Tree A satisfies this condition and Tree B does not.





Traversing Binary Search Trees

Thee are three natural ways in which we can visit each node in a binary tree. These are based on the recursive nature of the tree structure. The order of possible visitations is:

(1) Preorder

process root;

processs left sub-tree;

process right sub-tree

(2) Inorder

process left sub-tree;

processs root;

process right sub-tree

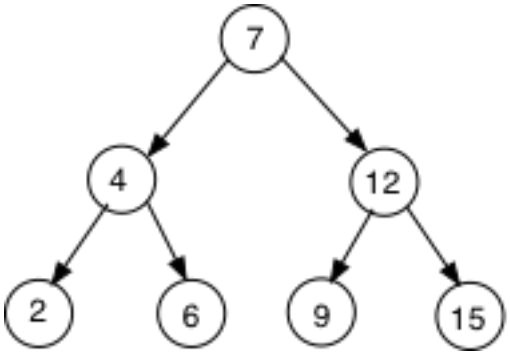
(3) Postorder

process left sub-tree;

process right sub-tree;

processs root

Consider the following tree:



Traversal using

preorder is: 7, 4, 2, 6, 12, 9, 15

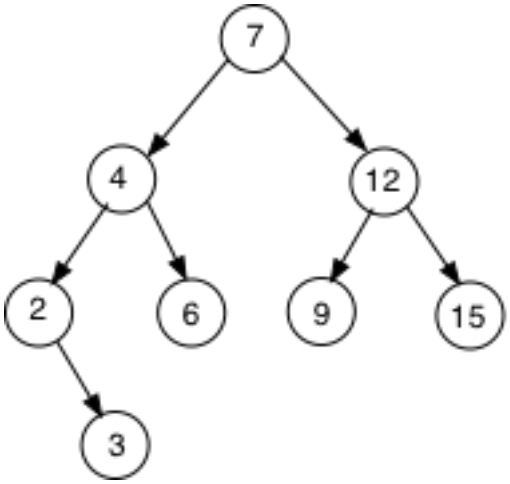
inorder is: 2, 4, 6, 7, 9, 12, 15

postorder is: 2, 6, 4, 9, 15, 12, 7

Observe that an **inorder** traversal of the tree gives a sorted list of values.

Adding a new value to a Binary Search Tree

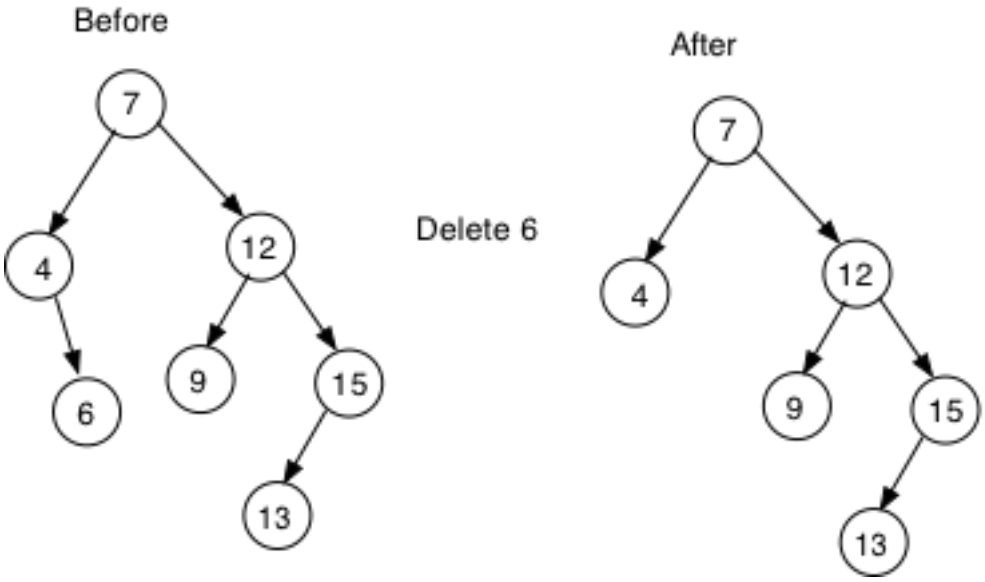
When adding a new value we must ensure that the *value in every node must be greater than all values in its left sub-tree, and less than all values in its right sub-tree* is invariant. A new value is added as a leaf node in such a way that this rule is preserved. To insert the value 3 in the tree given above search the tree until we find the correct node to attach it to. In this tree the only possible position is the right sub-tree of node 2. This gives



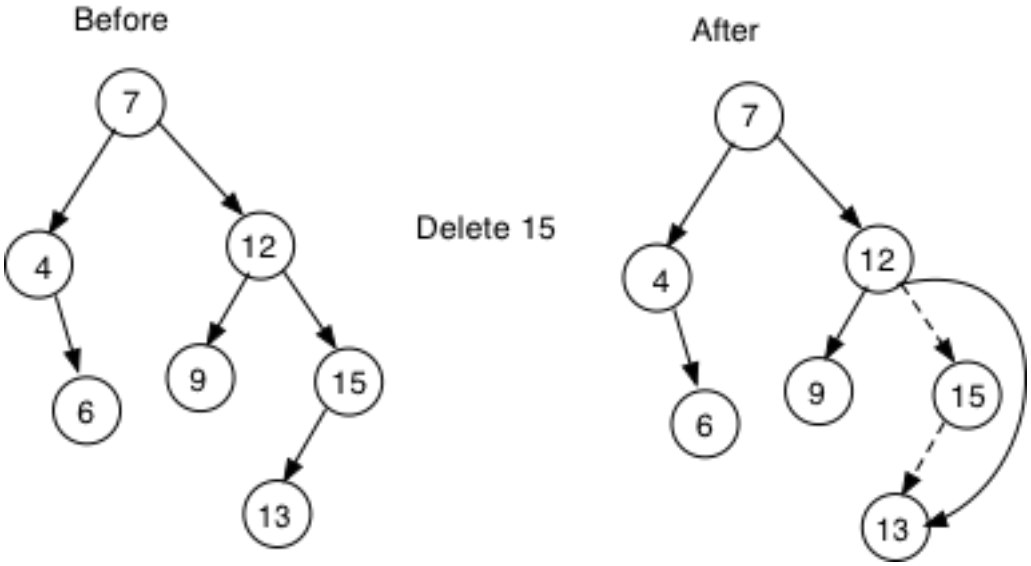
To construct a binary search tree from a given list of values the first item on the list becomes the root node.

Deleting an element from the tree

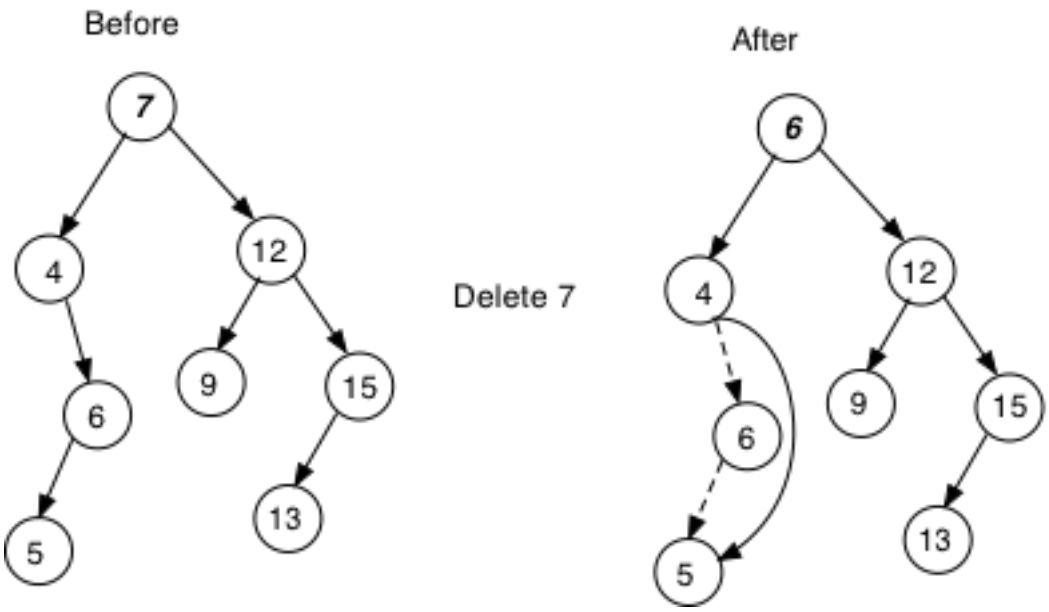
Deleting an element is relatively simple if the element is a leaf node or a node with a single descendent. The diagram below shows a binary search tree prior to deleting the leave node 6.



If the node to be deleted has a single descendent then the parent node simply points to the descendent of the node to be deleted. The diagram below illustrates the deletion of the node containing the value 15. It has a single descendent and its parent node points to it after the delete operation.



However, removing a node with two descendent sub-trees proves a problem. To remove such a node we cannot point its parent node in two directions at once. Hence, we must replace the data item with some other element in the tree such that its invariant property is preserved. One possible solution is to replace the value in the node to be deleted by the rightmost element in its left sub-tree and then delete this node. The diagram below shows that state of the tree before the root node containing 7 is removed. The value 6 is the rightmost element of its left sub-tree and it is chosen as the candidate to replace it. This node has a single descendent and is removed by pointing its parent to this descendent node.

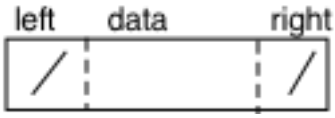


Implementing a Generic Binary Search Tree

A binary search tree has to maintain a sorted list of data elements and, hence, its generic type must supply a compareTo method. Each node has *degree* 2 and, hence, has to have a `left` and `right` reference to possible descendents. A new instance of `BNode` below always initializes both descendents to `null`. The constructor takes a single instance of type `E` as argument and initializes the `data` attribute. This value may also be modified using `set` and retrieved with the `data` method. The methods `setLeft` and `setRight` are used to connect child nodes. The code is:

```
private class BNode<E extends Comparable<E>>{
   private E data;
   private BNode<E> left;
   private BNode<E> right;
   public BNode(E d){
      data = d; left = null; right = null;
   }
   public E data(){return data;}
   public void set(E x){data = x;}
   public BNode<E> left(){ return left;}
   public BNode<E> right(){return right;}
   public void setLeft(BNode<E> k){left = k;}
   public void setRight(BNode<E> k){right =k;}
}
```

A picture of a node might be:



The constructor simply initializes `root` to `null` and sets attribute `count` to `0`, where `count` equals the number of nodes in the tree.

```
class BinarySearchTree<E extends Comparable<E>>, implements Iterable<E>{
   private BNode<E> root;
   private int count;
   public BinarySearchTree(){
       root = null; count = 0;
  }
   …
}
```

Method `add` takes an instance of `E` as argument and invokes a private method `add` that is defined recursively on the structure of the tree. It takes the root reference and the element to add to the tree as arguments. It terminates the recursion when its argument `rt` is `null`. In this case it creates a new `BNode` and returns a reference to it. The other two possible cases traverse the tree going left or right depending on the comparison of `x` with the data value in the given node `rt`. The code is:

```
public void add(E x){
  root = add(root,x);
}
private BNode<E> add(BNode<E> rt, E x){
   if(rt == null){
      count++;
      return new BNode<E>(x);
   }
   else if(x.compareTo(rt.data()) < 0){
      BNode<E> p = add(rt.left(), x);
      rt.setLeft(p);
      return rt;
   }
   else if(x.compareTo(rt.data()) > 0){
```

```
            BNode<E> p = add(rt.right(), x);
            rt.setRight(p);
            return rt;
        }
        else // x present, so no change
            return rt;
    }
```

The method **contains** searches the tree for a given value **x**. It uses a private method to recursively iterate over the tree.

```
    public boolean contains(E x){
        return contains(root, x);
    }
    private boolean contains(BNode<E> rt, E x){
        if(rt == null) return false;
        else{
            if(rt.data().equals(x)) return true;
            else if(x.compareTo(rt.data()) < 0)
                return contains(rt.left(),x);
            else
                return contains(rt.right(), x);
        }
    }
```

Three different traversals of a binary tree were discussed above. Given that we are implementing a binary search tree then the most relevant of these is the *in order* traversal because it visits the nodes in sorted order, least first. This algorithm processes the left sub-tree, followed by its root, followed by the right sub-tree. The method **inOrder** returns an **ArrayList** containing the data values in order. It uses a private method **inOrder** that recursively iterates over the tree. The implementation simply uses the definition above inserting elements in the array, **lst**, as it goes.

```
    public ArrayList<E> inOrder(){
        ArrayList<E> lst = new ArrayList<E>();
        inOrder(root,lst);
        return lst;
    }
    private void inOrder(BNode<E> rt, ArrayList<E> lst){
     if(rt != null){
         inOrder(rt.left(), lst);
```

```
        lst.add(rt.data());
        inOrder(rt.right(),lst);
    }
}
```

The implementations of *pre order* and *post order* also use the definitions given in their definitions.

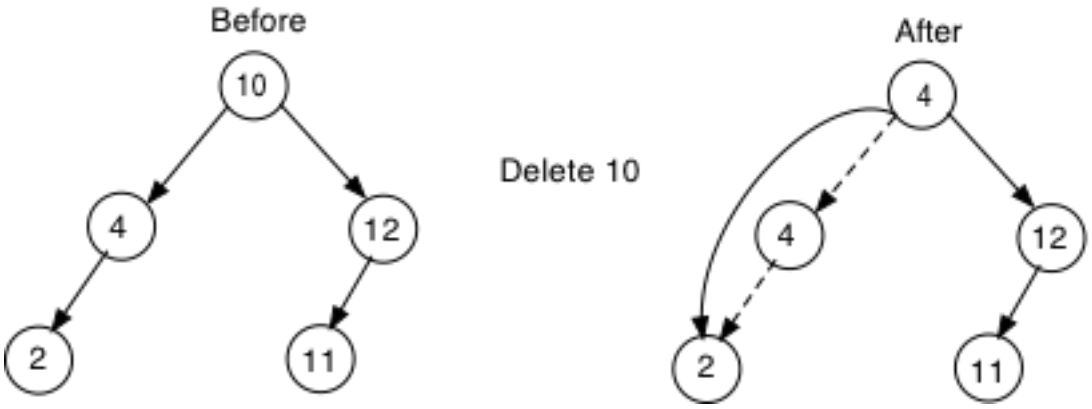
```
public ArrayList<E> preOrder(){
    ArrayList<E> lst = new ArrayList<E>();
    preOrder(root,lst);
    return lst;
}
private void preOrder(BNode<E> rt, ArrayList<E> lst){
    if(rt != null){
        lst.add(rt.data());  //process root
        preOrder(rt.left(), lst); //process left sub-tree
        preOrder(rt.right(),lst);  //process right sub-tree
    }
}

public ArrayList<E> postOrder(){
    ArrayList<E> lst = new ArrayList<E>();
    postOrder(root,lst);
    return lst;
}
private void postOrder(BNode<E> rt, ArrayList<E> lst){
    if(rt != null){
        postOrder(rt.left(), lst); //process left sub-tree
        postOrder(rt.right(),lst); //process right-subtree
        lst.add(rt.data());  //process root
    }
}
```

Method remove implements the remove algorithm described above. It begins by searching the tree for the given value `x`. There are two local variables `ptr` and its parent named `parentPtr`. When the search is complete `ptr` holds the reference to the node to be deleted and `parentPtr` references its immediate ancestor, if the search is successful. If the node to delete happens to be the root node then `root` may change. Hence, the assignment `root = removeNode(root)`. If x is less than its parent node data the parent left sub-tree will change; otherwise its right sub-tree changes.

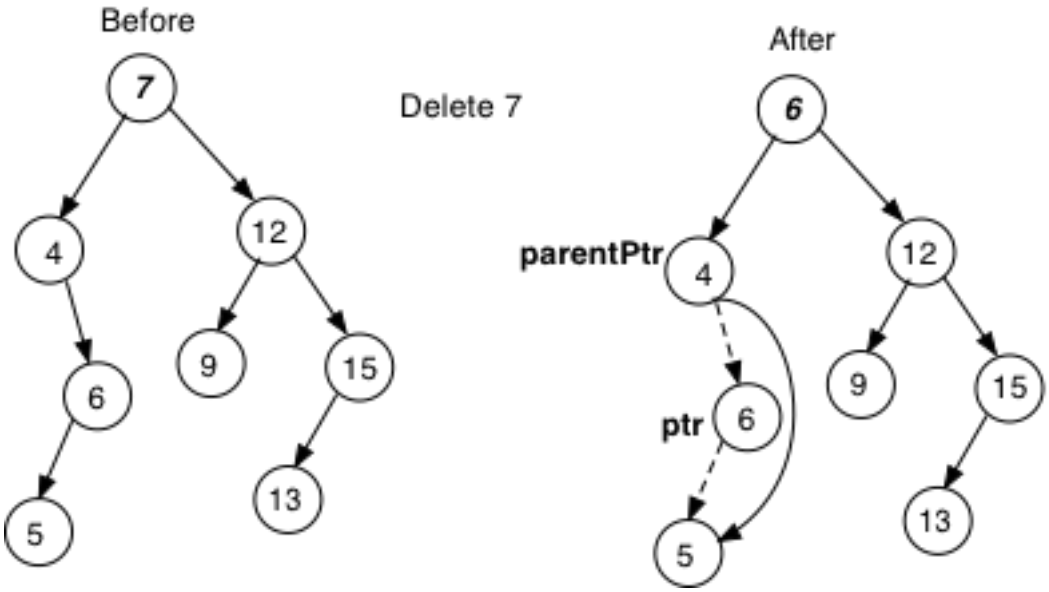
```
public void remove(E x){
   BNode<E> ptr = null; BNode<E> parentPtr = null;
   if(root != null){
      ptr = root; parentPtr = root;
      boolean found = false;
      while(ptr != null && !found){
      if(ptr.data().equals(x)) found = true;
      else{
        parentPtr = ptr;
        if(x.compareTo(ptr.data()) < 0)
         ptr = ptr.left();
        else
         ptr = ptr.right();
      }
      }
      if(found){
      if(ptr == root){
         root = removeNode(root);
      }
      else{
        if(x.compareTo(parentPtr.data()) < 0){
          BNode<E> n = removeNode(parentPtr.left());
          parentPtr.setLeft(n);
        }
        else{
          BNode<E> n = removeNode(parentPtr.right());
          parentPtr.setRight(n);
        }
         }
      }
  }
}
```

The method removeNode returns a pointer to the parent node. If the node is a leaf node it returns null. If its left node reference is null then it returns the right node reference. If its right node reference is null then it returns the left node reference. In the case where both sub-trees are not null, it finds the rightmost node of the left sub-tree and uses its value to replace the element to be deleted. It then adjusts the appropriate pointer. There are two possible cases. If the replacement node is the child of the parent then set parents, rt, left pointer to its left sub-tree, given by rt.setLeft(ptr.left()). This occurs because the left node has no right sub-tree. Consider the diagram where the value to be deleted is 10 and the replacement candidate

value is 4. In this case the left pointer of the parent node has to point to node with 2.



If the replacement node is the rightmost node of the left sub-tree that is not a child of the parent that is changed then parentPtr.setRight(ptr.left()).



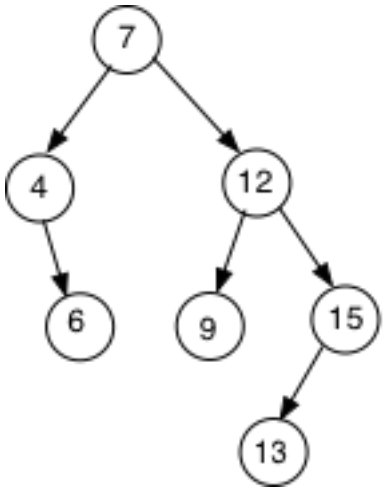

```
private BNode<E> removeNode(BNode<E> rt){
    if(rt == null) return null;
    else if (rt.left() == null && rt.right() == null)
        return null;
    else if(rt.left() == null){
        return rt.right();
    }
    else if(rt.right() == null){
        return rt.left();
    }
    else{
```

```
        BNode<E> ptr = rt.left();
        BNode<E> parentPtr = null;
        while(ptr.right() != null){
           parentPtr = ptr; ptr = ptr.right();
        }
        rt.set(ptr.data());
        if(parentPtr == null) // parent did not move
           rt.setLeft(ptr.left());
        else
           parentPtr.setRight(ptr.left());
        return rt;
     }
  }
```

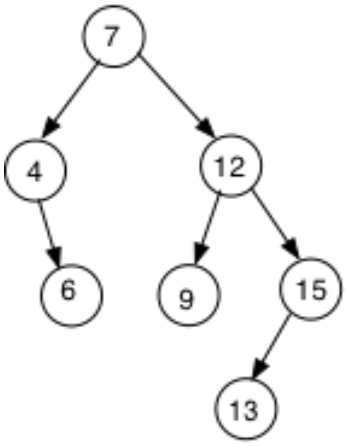
The *height* of a binary tree is defined to the maximum path length associated with any of its nodes. To calculate the height of a tree we provide a public method that returns an integer value called height. This method uses a method that recursively iterates over the tree starting at its root. For a tree with just one node, the root node, the height is defined to be 0, if there are 2 levels of nodes the height is 1 and so on. A null tree (no nodes except the null node) is defined to have a height of –1. Therefore, height of an empty tree is -1; otherwise it is `1 + max(height(rt.left(), height(rt.right())` where `max` returns the larger of the two values. The code is:

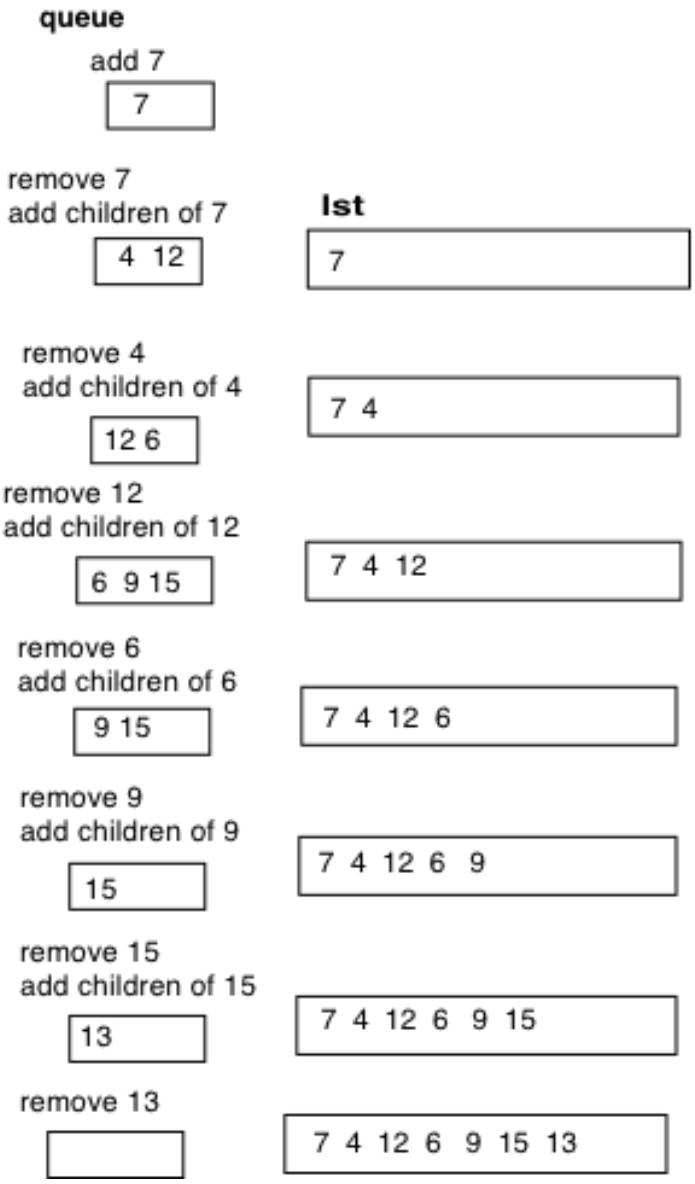
```
public int height(){
   return height(root);
}
private int height(BNode<E> rt){
   if(rt == null) return -1;
   else{
      return 1 + max(height(rt.left()),height(rt.right()));
   }
}
private int max(int a,int b){ return a >= b? a:b;}
```

The three iterations `inoOrder`, `preOrder` and `postOrder` each provide a depth first traversal of the tree. A breadth first traversal of the tree would visit each node in the tree level by level starting at the root. For example, the nodes in the given tree would be visited in the order: 7,4,12,6,9,15,13



The method breadthFirst returns an ArrayList containing the values in the tree ordered by level left to right. To visit the nodes level by level it turns out that a queue provides the best data manage the nodes in the tree. The queue stores references to the nodes. To see how this works we consider the following tree. The root 7 is appended to the queue. It is then removed from the queue, added to the list and its children join the queue. Then the head of the queue is removed (4), added to the list and its children (6) join the queue. This process continues until the queue is empty. The resultant list is the list of elements breadth first left to right.


7
4
12
6
9
15
13


queue
add 7
7
remove 7
add children of 7
4 12
lst
7
remove 4
add children of 4
12 6
7 4
remove 12
add children of 12
6 9 15
7 4 12
remove 6
add children of 6
9 15
7 4 12 6
remove 9
add children of 9
15
7 4 12 6 9
remove 15
add children of 15
13
7 4 12 6 9 15
remove 13
7 4 12 6 9 15 13

```
    public ArrayList<E> breadthFirst(){
        ArrayList<E> lst = new ArrayList<E>();
        CircularQueue<BNode<E>> queue = new
                          CircularQueue<BNode<E>>(count);
        if(root == null)
            return lst;
        else{
            queue.join(root);
            while(!queue.empty()){
            BNode<E> n = queue.top(); queue.leave();
            lst.add(n.data());
              if(n.left() != null) queue.join(n.left());
              if(n.right() != null) queue.join(n.right());
               }
               return lst;
        }
     }
   }
```

A test program that constructs a binary search tree of integer values is given. It creates a tree and then illustrates the use of the various methods defined for the class.

```
import java.io.*;
import java.util.*;
public class BinarySearchTreeTest {
    public static void main(String[] args) {
        BinarySearchTree<Integer> t1 = new BinarySearchTree<>(
                                                  Arrays.asList(4,1,7,6,3,2)
                                              );
    System.out.println(t1);
    System.out.println(t1.preOrder());
    System.out.println(t1.postOrder());
    System.out.println(t1.size());
    System.out.println(t1.height());
    t1.remove(4);
    System.out.println(t1);
    t1.remove(2); t1.remove(6);
    System.out.println(t1);
    System.out.println(t1.height());
    t1.remove(1);t1.remove(7);
    System.out.println(t1);
    System.out.println(t1.height());
    t1.remove(3);
    System.out.println(t1.height());
```

```
    for(int j = 0; j < 6;j++) t1.add((int)(Math.random()*10));
    System.out.println(t1.inOrder1());
    System.out.println(t1.preOrder1());
    System.out.println(t1.postOrder1());
    System.out.println(t1.breadthFirst()); }
}
```

Exercise

Question 1

Create a binary search tree using the following list of values: 9,12,4,7,15,2,6,10,19,23,1,3,11. Delete the following nodes from your tree re-drawing it each time: 1, 15, 9, 19.

Question 2

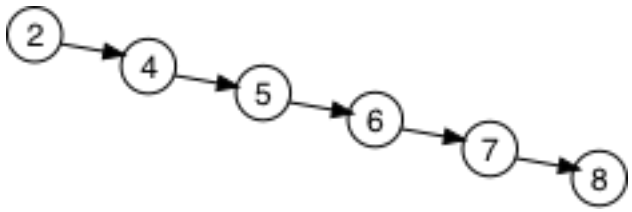
Create a binary search tree using the following list of values: 8.3.6.10.11.12.13.14.15. Delete the following nodes from your tree re-drawing it each time: 8,12,10.

Question 3

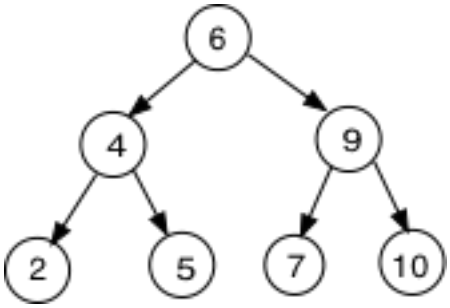
If you create a binary search tree with a sorted list of values what do you get?

Chapter 11: AVL Trees

A binary search tree is designed to optimize the cost of insertion and retrieval. At best they offer *O(log n)* insertion and retrieval performance. However, at worst they can degenerate to singly linked lists and only offer *O(n)* performance. To see how this can happen, consider creating a binary search tree with an ordered list of values. Creating a binary search tree with the list: 2,4,5,6,7,8 gives the following tree. This tree is a singly linked list where the cost of retrieval is *O(n)*.



The opposite of this type of *degenerate* binary search tree is what is termed a *complete* binary tree. A complete binary tree is one where the nodes are equitably distributed between the two sub-trees of a given node. Such a tree would always have minimal height. The tree below is a complete binary search tree.

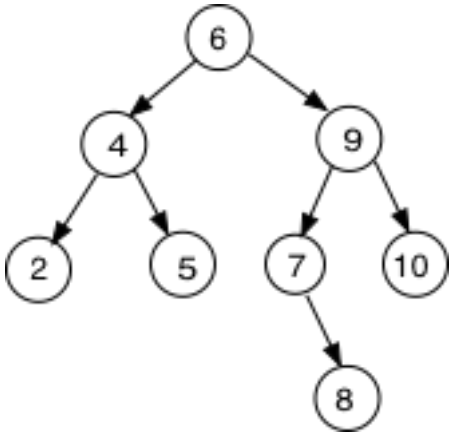


It can be shown that the average search cost of a binary search tree is *2 log 2 * log n*. This is approximately *1.39* times the cost of searching a completely balanced tree. This ideal model is not feasible for random sets of data. To address this problem two Russian computer scientists Adelson-Velski and Landis (1962) defined what became known as an *AVL tree*. This type of tree is a self- balancing binary search tree such that the number of left nodes and the number of right nodes of a given root node do not differ by more than 1. Each node of an AVL binary search tree has an associated balance factor that is left high, equal or right high according, respectively, as the left sub-tree has height greater than, equal to, or less than that of the right sub-tree. With this property, it can be shown that the depth of the tree will not exceed $1.44 \log_2 n$.

We begin our study of *AVL binary search trees* by testing some trees to determine if they satisfy the definition of an *avl* binary search tree. All leaf nodes have a balance factor of 0 and the accompanying table, for each tree, lists the balance factor of each node in the tree. The balance factor is always equal to the difference in height between the left sub-tree and the right sub-tree of a given node. This value will always be 1, 0, -1, if the tree is an *avl tree*. This value is got by subtracting the height of the right sub-tree from the height of the left sub-tree. The height of a node is measured starting with its deepest leaf node, tracing its path up the tree.

Example 1

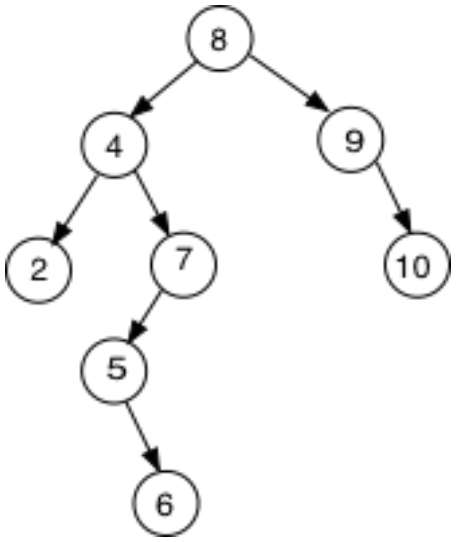
nodes	**balance**
2, 5,8, 10 (leaf nodes)	0
4	1-1 = 0
7	0-1 = -1
9	2-1 =1
6	2 –3 = -1



This tree satisfies the conditions for an *avl tree* because each node has a balance factor of 1, 0, or -1. Node 4 has a balance factor of 0 because the heights of its left sub-tree and right sub-tree are both 1. Node 7 has a balance factor of -1 because the height of its left sub-tree is 0 and that of its right sub-tree is 1. 6 has a balance factor of -1 because the height of its left sub-tree is 2 and that of its right sub-tree is 3.

Example 2

nodes	**balance**
2, 6, 10 (leaf nodes)	0
5	0-1 = -1
7	2-0 = 2
4	1-3 =-2
9	0 –1 = -1
8	4-2 = 2



This tree fails the test because not all nodes have a balance factor of 1, 0, or -1. The first offending node is 7 because the height of the left sub-tree is 2 and that of the right sub-tree is 0.

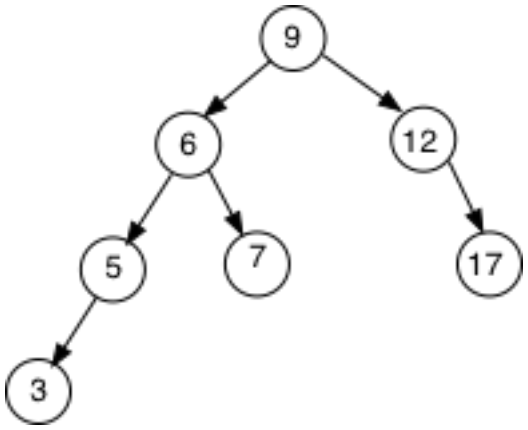
Exercise

Test the given trees to see if they pass the avl test.

Inserting an element in an AVL Tree

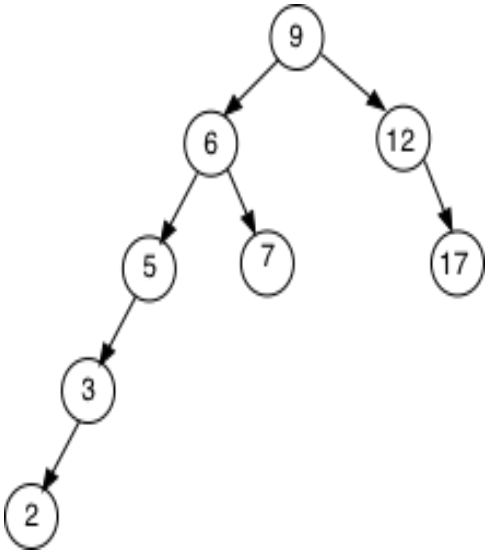
Given an *avl* tree we want to insert a new element in the tree while keeping its *avl* property invariant. Inserting a new element proceeds as for insertion in an ordinary binary search tree. This can cause problems because the tree now may become unbalanced and, hence, may need to re-balance itself. Given the following avl binary search tree to start with.

nodes	balance
3, 7, 17 (leaf nodes)	0
5	1-0 = 1
6	2-1 = 1
12	0-1 = -1
9	3 -2 = 1



Inserting 2 in the tree gives the unbalanced tree.

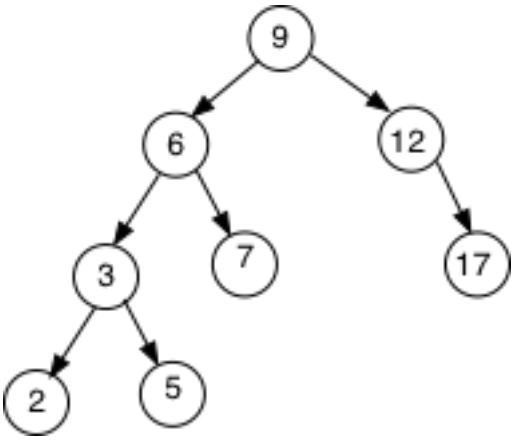
nodes	old balance	New balance
2, 7, 17 (leaf nodes)	0	0
3		1-0 = 1
5	1	2-0 = 2
6	1	3-1 = 2
12	-1	-1



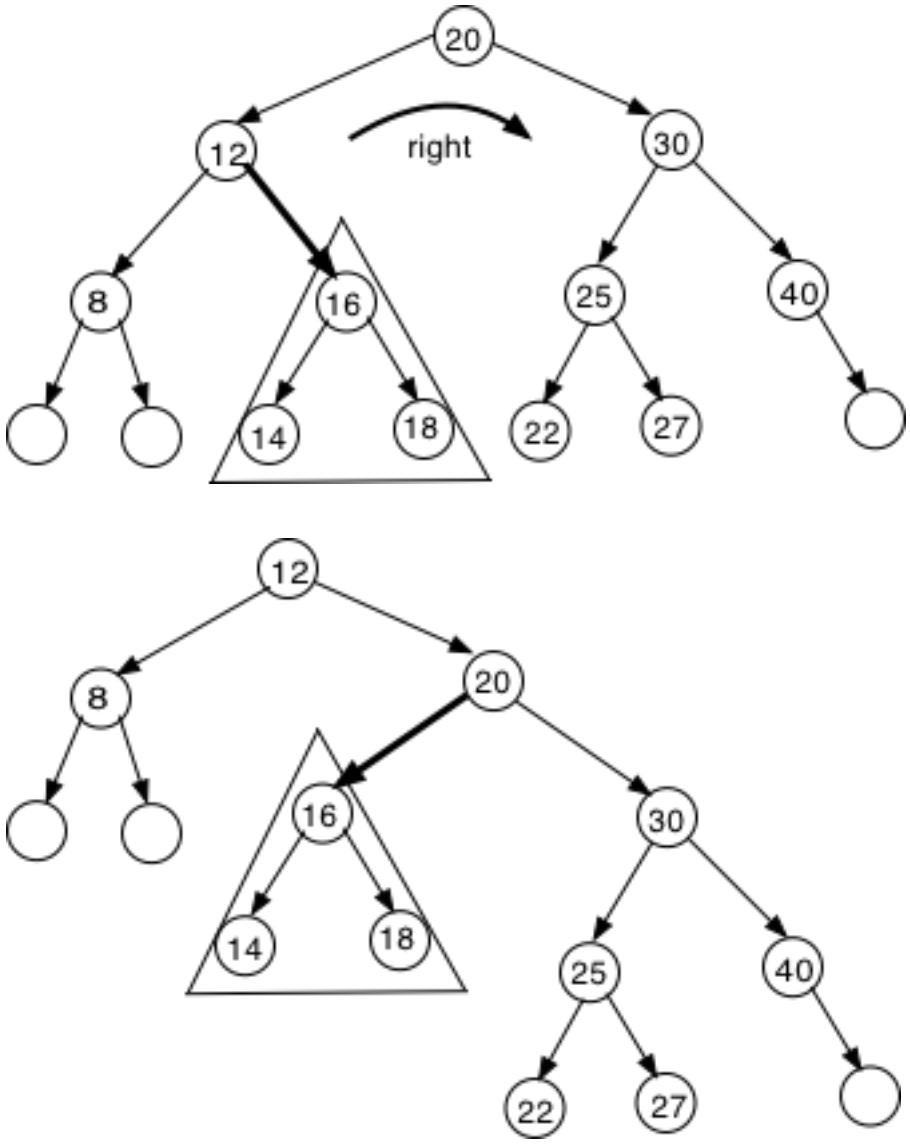
9	1	4-2 = 2

The first node ascending from the root node 2 that violates the *avl* property is 5. Hence, the tree must be re-balanced. To do this we re-balance the tree by placing the grandparent as the right child of its child node 3, and then connecting 3 to its own grandparent 6. Now each node in the tree has a balance factor satisfying the *avl* property and the new tree remains a binary search tree.

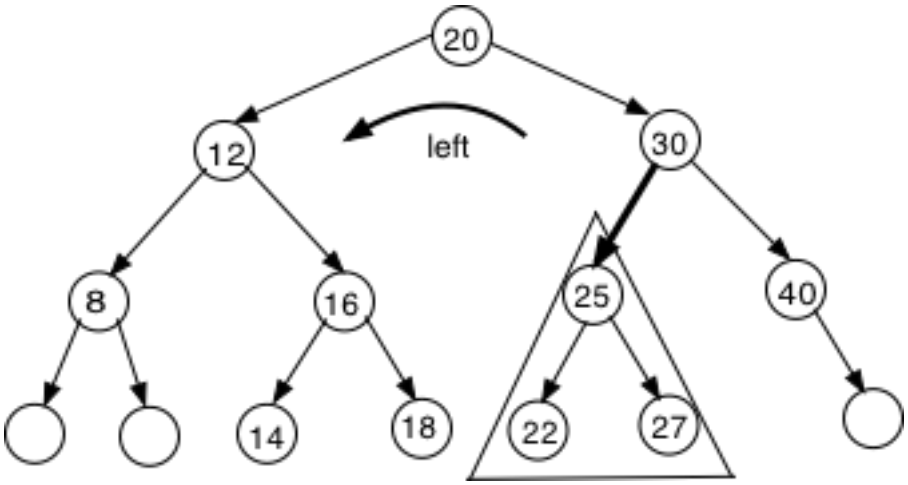
nodes	**balance**
2, 5, 7, 17 (leaf nodes)	0
3	1-1 = 0
6	2-1 = 1
12	0-1 = -1
9	3 –2 = 1

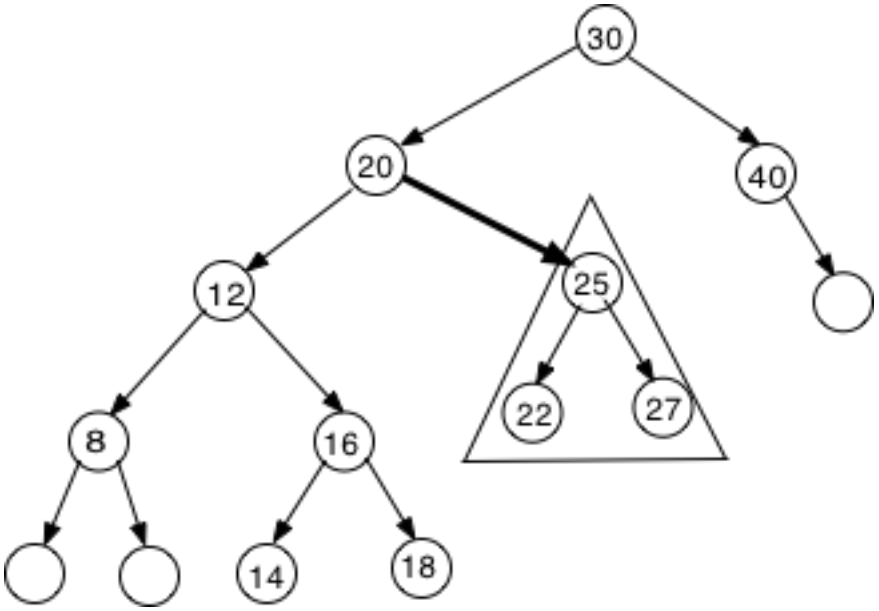


The task of re-balancing binary search trees keeping their ordering invariant is done by using different rotations of the nodes of the tree. There are two basic rotations used called left rotation and right rotation. Both of these are single rotations. To describe the mechanics of both rotations two examples are given. The tree listed below is a binary search tree. The plan is to perform a right rotation about the root 20 keeping invariant the ordering for a binary search tree. This rotation will make 12 the new root. The triangle enclosing nodes 14, 16 and 18 has to shift to the right side of the new root. This means that the block of nodes will be placed as the left child of 20 in the new tree. This shift is illustrated in the second diagram below. The rotation only requires a change of pointers – the left pointer of 20 points to node 16 and the right pointer of node 12 points to node 20. This has a fixed time cost and, hence, is *O(1).* This is called a ***right rotation***.

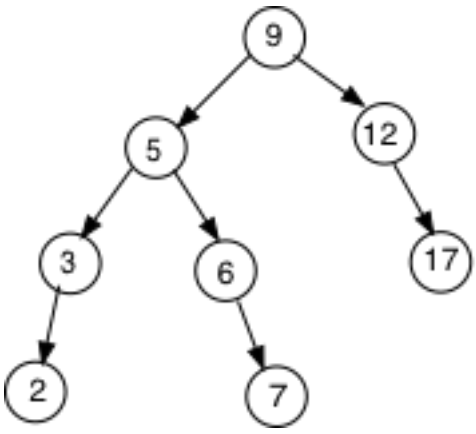


The second rotation is called a ***left rotation***. It is the mirror image of the *right rotation* described above. The pair of diagrams, given below, show the state of a tree before a ***left rotation*** and after it has taken place. To keep the ordering the right pointer of node 20 points to node 25 and the left pointer of node 30 points to node 20.





A left or a right rotation can be used to re-balance a binary search tree keeping its *avl* property invariant. The first example, given above, illustrated a right rotation necessary to balance the tree when the number 2 was inserted in the tree. We now give an example of a ***left rotation***. Given, below, is a balanced *avl* tree.

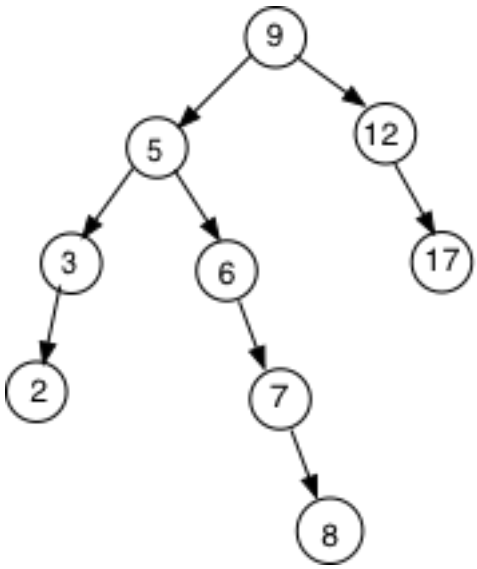


nodes	balance
2, 7, 17 (leaf nodes)	0
3	1-0 = 1
6	0-1 = -1
5	2-2 = 0

12	0-1 = -1
9	3 –2 = 1

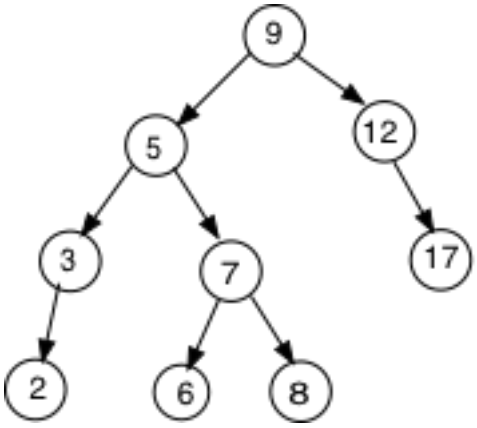
Inserting the number 8 to the right of 7 gives an unbalanced tree.

nodes	balance
2, 8, 17 (leaf nodes)	0
3	1-0 = 1
7	0-1 = -1
6	0-2 = -2
5	2-3 = -1
12	0-1 = -1
9	3 –2 = 1



Node 6 is the grandparent of node 8 and to re-balance the tree we place 6 as the left child of its child node 7 and 7 becomes the root node of the sub-tree. This re-balances the tree as the table below shows.

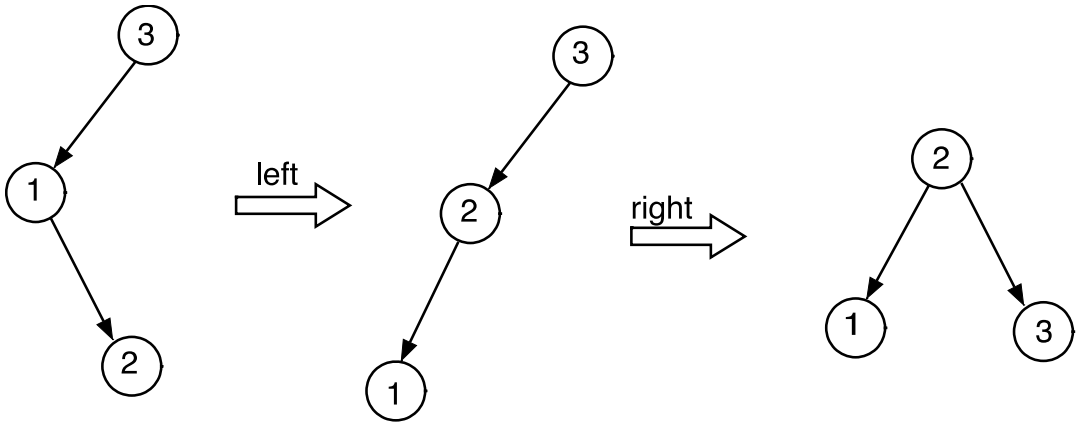
nodes	balance
2, 6, 8, 17 (leaf nodes)	0
3	1-0 = 1
7	1-1 = 0
5	2-2 = 0
12	0-1 = -1
9	3 –2 = 1



Sometimes when a new value is inserted the task of re-balancing the tree requires more than a single *left* or *right* rotation.

The example, below, illustrates what is called a ***left-right rotation***. We consider the simple case of creating a balanced tree from the list 3, 1, 2. When we add the values to the tree we

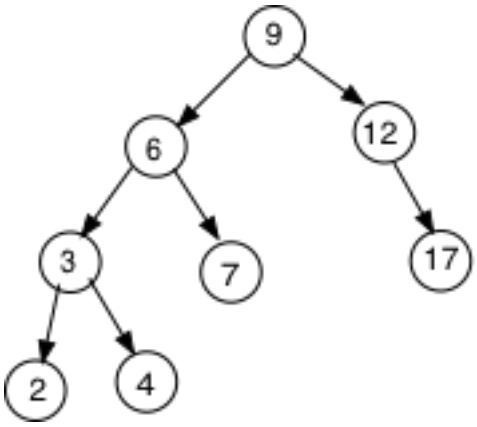
get an unbalanced tree on the left hand side of the diagram below. To balance this tree we cannot do a simple *left* or *right* rotation. To re-balance it requires two rotations: *left* about the number 1 giving the middle tree; followed by a right rotation at 3 giving the perfectly balanced tree on the right hand side.



This is a simple example to illustrate the technique but the real question is how do you know when to do this. The answer is that when the balnce goes from -1 to +2 we must do a left-right rotation. In the previous diagram the balance at 1 is -1 and at 3 is +2, hence a left-right rotation works. The next example, illustrates a more complex situation.

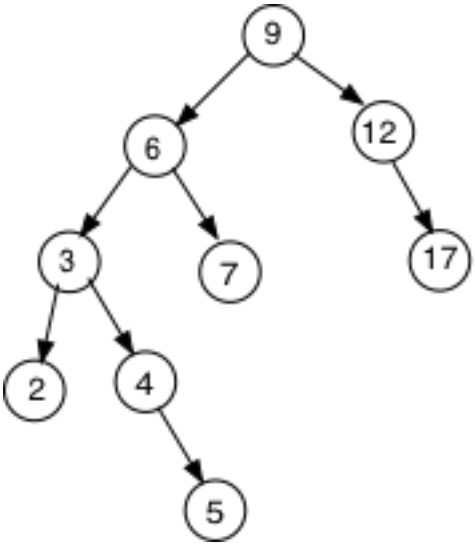
Given the balanced avl tree, below.

nodes	**balance**
2, 4, 7, 17 (leaf nodes)	0
3	1-1 = 0
6	2-1 = 1
12	0-1 = -1
9	3 –2 = 1



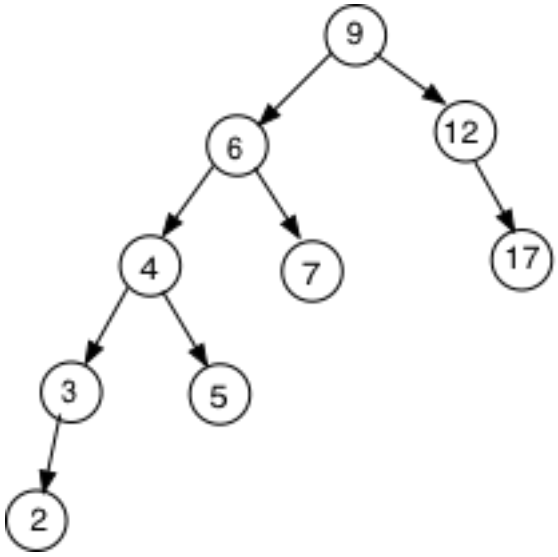
Inserting 5 to the right of 4 unbalances the tree as the table shows.

nodes	balance
2, 5, 7, 17 (leaf nodes)	0
4	0-1 = -1
3	1-2 = -1
6	3-1 = 2
12	0-1 = -1
9	4-2 = 2



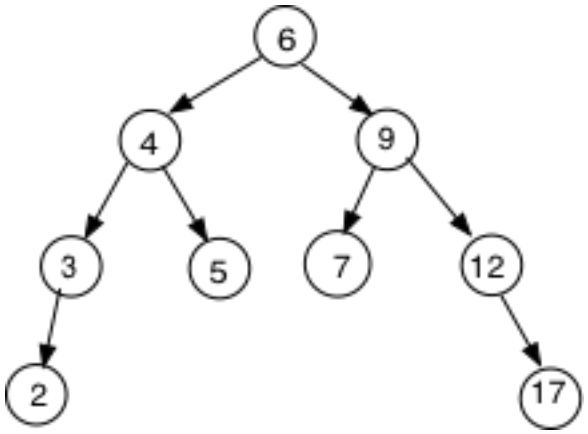
Re-balancing the tree requires two steps: a ***left*** rotation followed by a ***right*** rotation. The ***left*** rotation gives an intermediate tree that is still unbalanced.

nodes	balance
2, 5, 7, 17 (leaf nodes)	0
3	1-0 = 1
4	2-1 = 1
6	3-1 = 2
12	0-1 = -1
9	4-2 = 2



Taking this intermediate tree and rotating ***right*** around node 6 gives a balanced tree. Node 7 becomes a child of its original grandparent.

nodes	balance
2, 5, 7, 17 (leaf nodes)	0
3	1-0 = 1
4	2-1 = 1
6	3-3 = 0
9	1-2 = -1
12	0–1 = -1



The fourth, and final case, involves what is called a ***right-left rotation***. This is a mirror image of a ***left-right rotation***. The rule is: if the balance goes from +1 to -2 perform a right followed by a left rotation.

Deleting a node is as for binary search trees, except that depth re-calculations and sub-tree re-balancing along the path from the deletion point to the root must be carried out. This entails more work than insertion because the sub-tree re-balancing must proceed right up to the root.

Exercise
Create an *avl* binary search tree using the list: 4,1,6,9,12,5,15,18,19,2,3,7,8,10.

B-Tree

Consider a huge database where the index file is too large to store in memory. On such a database the indexing system will have to be built on a secondary storage device. The basic idea is to use a tree structure such that the nodes of the tree are stored on disk and the pointers to child nodes are represented by disk addresses and not memory addresses. Because disk access is very slow and costly the use of a binary search tree would be inefficient – at best $\log_2 n$ disk accesses – assuming the tree is well balanced. A better idea would be to use a multi-way tree where access is to an entire group of items. This gives rise to the idea of a page, where the page size corresponds to a multiple of the block size of the underlying storage device. This choice simplifies and optimizes disk access. Each disk access would read a single page. This can give incredibly fast retrieval times. For example, if the page size was 1000, then, given N items on disk, the cost of retrieval is $\log_{1000} N$. This is an incredibly small number even for huge values of N.

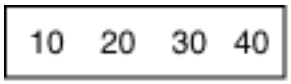
$$\log_{1000} 1000^3 = 3 * \log_{1000} 1000 = 3 * 1 = 3$$

This means that a tree of height 3 and a branching factor of 1000 can manage a billion keys but only requires at most 3 disk accesses to retrieve a key.

However, this degree of efficiency is only possible if the tree is not allowed to grow at random. The optimum would be to have a perfectly balanced tree. However, this must be ruled out because the balancing overhead is too great. In 1972 Bayer and McCreight came up with a solution that they called a ***B-tree***. The etymology of the name is uncertain. Bayer and McCreight worked as researchers for Boeing and some say the *B* stands for *Boeing*. However, most say it stands for *Bayer*. Their suggestion was to define a binary search tree that grows form the bottom up. Letting `n` = order of the B-tree, the rules for the tree are:

(1) Every page contains at most `2*n` items.
(2) Every page, except root, contains at least `n` items.
(3) Every page is either a leaf page, i.e. has no descendents, or has `m+1` descendents where `m` = the number of keys in the page.
(4) All leaf pages appear at the same level.

These laws force the tree to grow from the leaf nodes upwards in the following manner. Let the order of the tree be 2. This means that a page may contain at most 4 items. Adding four numbers to the root node gives:

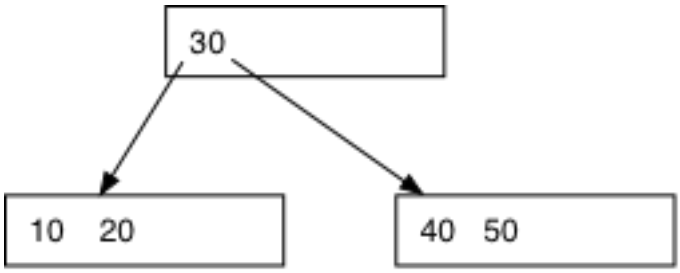


Observe that the numbers are sorted.

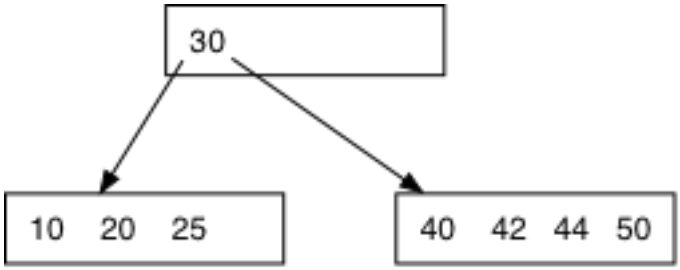
Inserting, say 50, now causes this node to split in the following way:

(a) Split node into two nodes
(b) Put smallest two elements in the left node
(c) Put largest two elements in the right node
(d) Create new root node and insert the middle value in it.

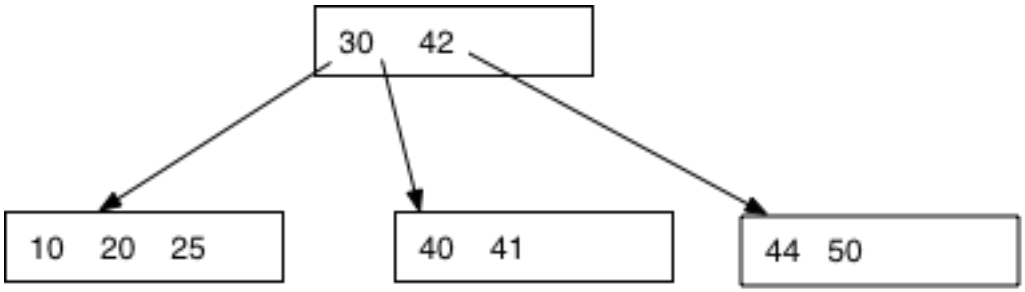
The new tree is:



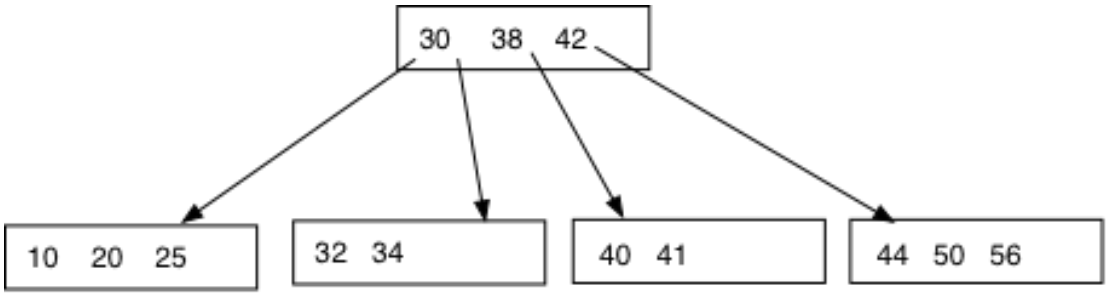
Inserting 25, 42 and 44 gives:



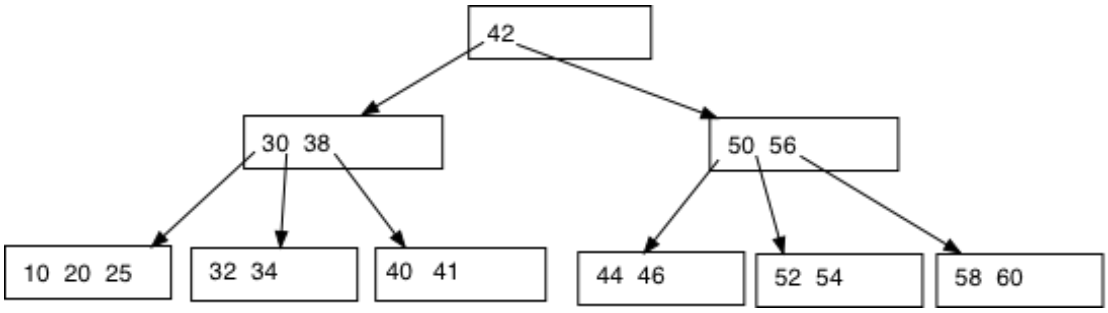
Adding 41 will now force the tree to split again.



Adding 32, 38, 56 and 34 gives



Finally, adding 58, 60, 52, 54 and 46 results in a tree of height 3.


42
30 38
50 56
10 20 25
32 34
40 41
44 46
52 54
58 60

Chapter 12: Maps

A map is a collection of items where each item has two parts called a key and a value. All the keys in a map must be unique and each key can only map to at most a single value. Keys should be immutable and the java documentation states that: *the behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map*. The Map interface does not inherit from Collection but the public methods provide a similar role to those found in the Collection framework. They fall into four groups: methods to add elements, methods to remove elements, methods that can be used to query the map and methods that provide different views of the contents of the map. In total there are 8 different implementations and we will look at three of these: HashMap, TreeMap and EnumMap. (The remaining implementations and additional constructors can be found in Java Docs.) The following table lists their methods:

Constructor	HashMap<K,V>() HashMap <K,V>(Map<? extends K, ? extends V> mp) TreeMap<K,V>() TreeMap <K,V>(Map<? extends K, ? extends V> mp) EnumMap(Class<K> keyType)
Add or replace a key-value pair	put(K key, V value) putAll(Map<? extends K, ? extends V> mp)
If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.	V putIfAbsent(K key, V value)
Remove key-value pair and returns value associated with key, or null	V remove(Object key)
Replaces the entry for the specified key only if it is currently mapped to some value.	V replace(K key, V value)
Replaces the entry for the specified key only if currently mapped to the	boolean replace(K key, V oldValue, V newValue)

specified value.	
Contains key	boolean containsKey(Object key)
Contains value	boolean containsValue(Object value);
Number of elements	int size()
Convert to string	toString()
Empty set	boolean isEmpty()
Remove elements	clear()
Retrieve value	V get(Object key);
Retrieve the key set	Set <K> keySet();
Retrieve values	Collection<V> values();

It is important to note that if you are using a HashMap to contain instances of a particular class then the attribute or attributes used to calculate the equals method and, hence, the hashCode method must be immutable. Similarly, for the TreeMap, the attributes used for calculateding compareTo should be immutable.

The following program creates a TreeMap that associates the names of people with a profession. Because it is a TreeMap the key-value pairs are sorted by key value. Hence, the ouptut from the first print is:

{Anne=Doctor, Carmel=Lecturer, Fran=Lecturer, Jim=Doctor, Mary=Dentist}

We then delete *Jim* and re-print giving: {Anne=Doctor, Carmel=Lecturer, Fran=Lecturer, Mary=Dentist}

Next we extract the key set and the set of values and finally modify the professions of existing key-value pairs.

```
import java.util.*;
class MapTest{
   public static void main(String args[]){
      //Create a map of names to occupations
      Map<String,String> map = new TreeMap<String,String>();
      map.put("Fran","Lecturer");
      map.put("Jim", "Doctor");
      map.put("Mary","Dentist");
      map.put("Carmel","Lecturer");
      map.put("Anne","Doctor");
      System.out.println(map);
      map.remove("Jim");
      System.out.println(map);
```

```
        // retrieve the set of keys
        HashSet<String> keys = new HashSet<String>(map.keySet());
        System.out.println(keys);
        TreeSet<String> prof = new TreeSet<>(map.values());
        System.out.println(prof);

        //change profession for Mary
        map.replace("Mary","Dentist","Lecturer");
        System.out.println(map);
        //change profession of Fran
        map.put("Fran","Doctor");
        System.out.println(map); }
}
```

In this next example we provide a program that builds a word concordance by computing the frequency of each word in a given list. A word is defined to be a sequence of characters with no spaces or punctuation. The list, ls, contains a short sequence of words involving duplicates. We use a TreeMap of String-Integer pairs to calculate the frequency of each word in the list. The ouptut from part one of the program is: {hat=4, heat=2, hope=1, sad=2}
In the second part of the program we find the word, or words, with the highest frequency. This is done in two steps: find the highest frequency and using it construct a list of words. The output is: Words with max freq 4 are: [hat]

```
import java.util.*;
class WordFrequency{
  public static void main(String args[]){
    //Compute word frequencies from a list of words
    List<String> ls = new ArrayList<>(Arrays.asList("hat","hope","heat","hat",
                                                    "heat","sad","sad","hat","hat"));
    Map<String,Integer> wFreq = new TreeMap<String,Integer>();
    for(String wd : ls)
       if(wFreq.containsKey(wd))
            wFreq.put(wd,wFreq.get(wd)+1);
       else
            wFreq.put(wd,1);
     System.out.println(wFreq);
     //find word or words with highest frequency
     //solution - find highest frequency and then return words that match this value.
     int maxFreq = 0;
     for(String wd : wFreq.keySet())
        if(maxFreq < wFreq.get(wd)) maxFreq = wFreq.get(wd);
      List<String> wds = new ArrayList<>();
```

```
    for(String wd : wFreq.keySet())
      if(maxFreq == wFreq.get(wd)) wds.add(wd);
    System.out.println("Words with max freq "+maxFreq+" are: "+wds);   }
  }
}
```

We want to make a list of authors grouped by nationality. This means we need a data structure that associates a nationality with a list of authors. A graph of such a data structure might be similar to the one given below. Each nation listed on the left hand side has an associated list of authors. The key value in each case is the nationality.

Using a **Map** we can easily build such a data structure. The code fragment below creates a **TreeMap** that associates a **String** instance with a **List<String>**, where key values will be represented by the names of nationalities. Two approaches to adding new vlaue pairs to the list are given. In the first part we simply construct a simple list. In the second part we create a separate list of nation-author pairs and proceed to add them to the existing list. Notice that if the key already exists then we extract the actual value list and append the author to that list (**natAuthors.get(p.x()).add(p.y());**). Alternatively, if the nation key does not exist we add a new pair to the map as follows:

```
List<String> temp = new ArrayList<>();
temp.add(p.y());
natAuthors.put(p.x(),temp);
```

To extract or view the actual list of authors two solutions are provided. In the first case we simply extract the list of lists as a Collection using:

```
Collection autLst1 = natAuthors.values();
```

This gives an Object list of list and is useful just to view the actual value lists.

The second and better solution is to flatten the list of lists to a single list. To do this we create a new list and use the addAll method to append the next list of authors to the flattened list.

```
List<String> autLst = new ArrayList<>();
for(String n : natAuthors.keySet())
     autLst.addAll(natAuthors.get(n));
```

The completed code fragment together with a listing of the Pair class is given below.

```
Map<String, List<String>> natAuthors = new TreeMap<>();
natAuthors.put("Irish", new ArrayList<>(Arrays.asList("Banville","Doyle","Barry")));
natAuthors.put("French",new ArrayList<>(Arrays.asList("Sartre","Flaubert")));
natAuthors.put("English",new ArrayList<>(Arrays.asList("Faulks","Dickens","Dahl")));
System.out.println(natAuthors);
//print in order of nationality one per row
for(String n : natAuthors.keySet())
     System.out.println(n+" => "+natAuthors.get(n));

// add author to natAuthors
List<Pair<String,String>> nls = new ArrayList<>(Arrays.asList(
                                        new Pair<String,String>("French","Balzac"),
                                        new Pair<String,String>("German","Mann"),
                                        new Pair<String,String>("Irish","Joyce"),
                                        new Pair<String,String>("German","Hesse"),
                                        new Pair<String,String>("French","Voltaire"),
                                        new Pair<String,String>("Scottish","Burns")
                                    ));


for(Pair<String,String> p : nls)
   if(natAuthors.containsKey(p.x()))
      natAuthors.get(p.x()).add(p.y());
    else{
      List<String> temp = new ArrayList<>();
      temp.add(p.y());
      natAuthors.put(p.x(),temp);
    }
System.out.println("Updated list:");
for(String n : natAuthors.keySet())
     System.out.println(n+" => "+natAuthors.get(n));

//List all authors - flatten lists to a single list
List<String> autLst = new ArrayList<>();
for(String n : natAuthors.keySet())
```

```
    autLst.addAll(natAuthors.get(n));
System.out.println(autLst);

//alternatively we could simply write - this gives a list of list view
System.out.println("Output collection======");
Collection autLst1 = natAuthors.values();
System.out.println(autLst1);
for(Object s : autLst1) System.out.println(s);


class Pair<A,B>{
   private A x; private B y;
   Pair(A x1, B y1){x = x1; y = y1;}
   public A x(){return x;}
   public B y(){return y;}
   public boolean equals(Object ob){
      if(!(ob instanceof Pair)) return false;
      Pair<?,?> p = (Pair)ob;
      return x.equals(p.x) && y.equals(p.y);
   }
}
```

Suppose we are only interested in writers from England, Scotland, Ireland and Wales. We could create an enumerated type and construct the map using an EnumMap implementation. The following code fragment shows how to do this. We first create the EnumMap using the class Nationality that is an enumerated type containing just four values. We then iterate over these values and create a map that associates each nationality with an empty list. Finally, we iterate over the list prs building the correct associations as we go.

```
List<Pair<Nationality,String>> prs;
prs = new ArrayList<>(Arrays.asList(
                        new Pair<>(Nationality.ENGLISH, "Faulks"),
                        new Pair<>(Nationality.SCOTTISH, "Burns"),
                        new Pair<>(Nationality.SCOTTISH, "Welsh"),
                        new Pair<>(Nationality.WELSH, "Thomas"),
                        new Pair<>(Nationality.IRISH, "Beckett"),
                        new Pair<>(Nationality.ENGLISH, "Dickens"),
                        new Pair<>(Nationality.ENGLISH, "Shakespeare"),
                        new Pair<>(Nationality.IRISH, "Doyle"),
                        new Pair<>(Nationality.IRISH, "Dunne"),
                        new Pair<>(Nationality.ENGLISH, "Dahl")
                      ));
```

```
Map<Nationality,List<String>> map1 = new EnumMap<>(Nationality.class);
for(Nationality n : Nationality.values())
    map1.put(n,new ArrayList<>());
for(Pair<Nationality,String> p : prs) map1.get(p.x()).add(p.y());
System.out.println(map1);


enum Nationality{IRISH, SCOTTISH, WELSH, ENGLISH}
```

Map Methods that use Functions as Arguments

The Map class has also new additional higher order methods that take specialized functions as arguments and apply them to some or all keys in the map. The methods together with a brief description of their semantics are listed in the table below.

Return type	Method	Meaning
void	forEach(Consumer<? super E> action)	Applies the given action function to all the elements in the list in order.
V	compute(K key, BiFunction<?super K,? super V, ? extends V> mapFunc)	Attempts to compute a mapping for the specified key and its current mapped value
V	computeIfAbsent(K key, Function<?super K,? extends V> mapFunc)	If key not present, it computes its value with the mapFunc and enters it in the map
V	computeIfPresent(K key, BiFunction<?super K,? super V, ? extends V> mapFunc)	If key is present, it computes a new value with the mapFunc.
void	replacAll(BiFunction<?super K,? super V, ? extends V> func)	Replaces each key's value with result of applying func to the key.

The program listed below gives examples of using these methods. The first part creates a Map<String,Integer> called mp1 that associates words with their lengths. It then uses a BiConsumer function that takes a pair, <String, Integer>, as argument and prints the string only if it has length 3. The forEach method from the map class is then used to print words that satisfy the function print3. The second part creates a Map<Integer,String> where key values are all mapped to null. It then iterates over all the keys applying mp2.compute to each key value. The BiFunction is written as the lambda expression (k,v)->(v==null) ? "Empty" : v). This function replaces all occurrences of null with the word Empty. In this particular case

we could also have used: mp2.replaceAll((k,v)->(v==null) ? "Empty" : v)). The third part gives examples of computeIfAbsent and computeIfPresent. The computeIfAbsent example simply adds a new key, value pair to the map. A lambda expression is passed as argument for the required function. It maps the key, 6, to its associated value, Six. The computeIfPresent method takes both a key and a BiFunction as argument. The example firstly creates the BiFunction remap that takes an integer and string as arguments and returns a string. It is defined by the expression: (k,s)-> s.equals("Empty") ? k.toString() : s. A for loop is then used to iterate over the set of keys and apply remap to each key.

```
import java.util.*;
import java.util.function.*;
public class MapTest{
  public static void main(String args[]){
    Map<String,Integer> mp1 = new TreeMap<String,Integer>();
    mp1.put("box",3); mp1.put("house",5); mp1.put("book",4);
    mp1.put("hat",3); mp1.put("mushroom",8); mp1.put("horse",5);

    BiConsumer<String,Integer> print3 = (k,x) ->{
       if(x == 3) System.out.println(k);
    };
    System.out.println("Three letter words");
    mp1.forEach(print3);

    //example for compute
    //Suppose you have a map with null values and you want to replace all of them
   //with "Empty"
    Map<Integer,String> mp2 = new TreeMap<>();
    for(int j = 0; j < 5;j++) mp2.put(j+1,null);
    System.out.println(mp2);
    for(Integer key : mp2.keySet())
       mp2.compute(key,(k,v)->(v == null) ? "Empty" : v);
    System.out.println(mp2);

    //computeIfAbsent
    mp2.computeIfAbsent(6, k -> "Six");
    System.out.println(mp2);

    //computeIfPresent
    BiFunction<Integer,String,String> remap =
       (k,s)-> s.equals("Empty") ? k.toString() : s;
    for(Integer key : mp2.keySet())
        mp2.computeIfPresent(key,remap);
```

```
    System.out.println(mp2);
  }
}
```

Generic Bag Problem

In Mathematics, a bag (multi-set) is a generalization of the notion of a set in which elements may appear more than once. The number of times an element occurs in a bag is called its multiplicity. Two bags are said to be equal if and only if they contain the same elements with the same multiplicities (frequencies). For example, if bag A = [2,2,3,4,4,4] and bag B = [2,4,4,3,2,4], then A == B. The total number of elements in a bag including repeated elements is its cardinality.

Like sets, for bags we define bag union, intersection and difference.
Given two bags A, B:

A ∪ B (union) = the bag containing those elements common to A and B, where for each common element x the multiplicity of x in A ∪ B is the greater of the multiplicity of x in A, the multiplicity of x in B.

A ∩ B (intersection) = the bag containing those elements common to A and B, where for each common element x the multiplicity of x in A ∩ B is the lesser of the multiplicity of x in A, the multiplicity of x in B.

A – B (difference) = the bag containing those elements in A not contained in B and for those x common to A and B the multiplicity of x in A – B is the greater of the multiplicity of x in A minus the multiplicity of x in B, 0. (The reason for this rule is that the multiplicity of x in B may exceed the multiplicity of x in A in which case the multiplicity is 0.)

There is also an operator called multi-set sum (+) that joins two existing bags to form a new bag that contains all the values in both bags.

For example,

A = [1, 1, 3, 3, 3, 6], B = [2, 3, 3, 6, 6, 8]
A ∪ B = [1, 1, 2, 3, 3, 3, 6, 6, 8]

$A \cap B = [3, 3, 6]$

$A - B = [1, 1, 3]$

$A + B = [1, 1, 2, 3, 3, 3, 3, 3, 6, 6, 6, 8]$

We also define inclusion (subset) for bag A as any bag B containing only elements of A where for each x in B, the multiplicity of x in B is less than or equal to the multiplicity of x in A. For example, $[2, 3, 3, 6] \subseteq B$.

We provide an implementation of a bag using a Map<E,Integer>, where E is the generic type used as key for the element and Integer value represents the frequency of occurrence of key instances in the bag. The generic class is called TreeBag<E extends Comparable<E>> implements Iterable<E>. All instances of the generic type E must implement the Comparable interface. An implementation decision is that no null values are allowed and attempts to add them are just ignored. The entire class is listed below. The implementation of the basic methods should be clear from the work we have completed during the previous sections and chapters. However, some methods do deserve discussion. The methods that were difficult to write were union, ..

The union of two bags is the bag containing those elements common to A and B, where for each common element x the multiplicity of x in $A \cup B$ is the greater of the multiplicity of x in A, the multiplicity of x in B. This means that all the values in A and B are included and the frequency of each one equates to the max of A.count(x), B.count(x). Because we need to include all elements we create a set of all key values called dom. We then iterate over this set and add the pair with the highest count() value. Note that if x is not contained in one of the bags count() returns 0. This is given by

```
TreeBag<E> union(TreeBag<E> bg){
   //defined: (A U B)(x) = {bag x : max(A.count(x), B.count(x))}
   TreeBag<E> tmp = new TreeBag<E>();
   TreeSet<E> dom = new TreeSet<>(this.ran());
   dom.addAll(bg.ran());
   for(E x : dom){
      int a = this.count(x); int b = bg.count(x);
      if(a < b)
         tmp.add(x,b);
      else
         tmp.add(x,a);
```

```
    }
    return tmp;
  }
```

Each of the methods: retainAll, removeAll and containsAll use a similar approach.

TreeBag Listing

```
import java.util.*;
public class TreeBag<E extends Comparable<E>> implements Iterable<E> {
    private Map<E,Integer> bag;
    public TreeBag(){bag = new TreeMap<>();}
    public TreeBag(List<? extends E> ls) {
        bag = new TreeMap<E,Integer>();
        for(E x : ls) this.add(x);
  }
  public TreeBag(TreeBag<? extends E> bg) {
        bag = new TreeMap<E,Integer>();
        for(E x : bg) this.add(x);
  }
  public void add(E x){
    if(x == null) return;
    if(bag.containsKey(x))
        bag.put(x,bag.get(x)+1);
    else
        bag.put(x,1);
  }
  public void add(E x, int k){
    if(x == null || k <= 0) return;
    if(bag.containsKey(x))
        bag.put(x,bag.get(x)+k);
    else
        bag.put(x,k);
  }
  public void add(List<? extends E> ls){
    if(ls == null) return;
    for(E x : ls) this.add(x);
  }
  public Integer count(E x){
    if(bag.containsKey(x)) return bag.get(x);
    return 0;
  }
  public boolean contains(E x){return bag.containsKey(x);}

  public void remove(E x, int k){
```

```
  if(!bag.containsKey(x) || k <= 0) return;
  if(bag.get(x) > k)
     bag.put(x,bag.get(x) - k);
  else
     bag.remove(x);
}
public void removeAll(E x){
 if(bag.containsKey(x)) bag.remove(x);
}
public int size(){
  int s = 0;
  for(E x : bag.keySet()) s += bag.get(x);
  return s;
}
TreeSet<E> ran(){
     TreeSet<E> ls = new TreeSet<>();
     for(E x : bag.keySet()) ls.add(x);
     return ls;
}

TreeBag<E> addAll(TreeBag<E> bg){
  //defined: (A + B)(x) = {bag x : (A+B).count(x) = A.count(x) + B.count(x)}
  TreeBag<E> tmp = new TreeBag<E>(bg);
  for(E x : bag.keySet())
     tmp.add(x,bag.get(x));
  return tmp;
}

TreeBag<E> union(TreeBag<E> bg){
  //defined: (A U B)(x) = {bag x : max(A.count(x), B.count(x))}
  TreeBag<E> tmp = new TreeBag<E>();
  TreeSet<E> dom = new TreeSet<>(this.ran());
  dom.addAll(bg.ran());
  for(E x : dom){
    int a = this.count(x); int b = bg.count(x);
     if(a < b)
        tmp.add(x,b);
     else
        tmp.add(x,a);
  }
  return tmp;
}
TreeBag<E> retainAll(TreeBag<E> bg){
   //defined: (A * B)(x) = {bag x : min(A.count(x), B.count(x))}
  TreeBag<E> tmp = new TreeBag<E>();
```

```
  TreeSet<E> dom = new TreeSet<>(this.ran());
  dom.addAll(bg.ran());
  for(E x : dom){
    int a = this.count(x); int b = bg.count(x);
    if(a > 0 && b > 0)
          if(a < b)
             tmp.add(x,a);
          else
             tmp.add(x,b);
  }
  return tmp;
}
TreeBag<E> removeAll(TreeBag<E> bg){
  //defined: (A - B)(x) = {bag x : max(0, A.count(x) - B.count(x))}
  TreeBag<E> tmp = new TreeBag<E>();
  TreeSet<E> dom = new TreeSet<>(this.ran());
  dom.addAll(bg.ran());
  for(E x : dom){
    int a = this.count(x); int b = bg.count(x);
    if(a - b > 0)
             tmp.add(x,a-b);
  }
  return tmp;
}
boolean containsAll(TreeBag<E> bg){
  //defined: (B << A)(x) = forAll x in B.ran() : B.count(x) <= A.count(x))}
  TreeBag<E> tmp = new TreeBag<E>();
  TreeSet<E> dom = new TreeSet<>(bg.ran());
  for(E x : dom){
    int a = this.count(x); int b = bg.count(x);
    if(a < b) return false;
  }
  return true;
}
public Iterator<E> iterator(){
  List<E> ls = new ArrayList<E>();
    for(E x : bag.keySet()){
      int fr = bag.get(x);
      for(int j = 0; j < fr; j++) ls.add(x);
    }
    return ls.iterator();
}
public String toString(){
  List<E> ls = new ArrayList<E>();
    for(E x : bag.keySet()){
```

```
      int fr = bag.get(x);
      for(int j = 0; j < fr; j++) ls.add(x);
    }
    return ls.toString();
  }
}
```

A short program that tests this class is listed below.

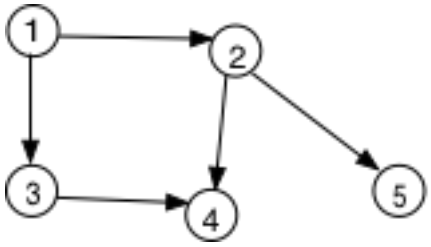
```
import java.util.*;
public class TreeBagTest {
  public static void main(String[] args) {
    TreeBag<Integer> bg1 = new TreeBag<>(Arrays.asList(2,3,4,3,5,1,1,5,3));
    TreeBag<Integer> bg2 = new TreeBag<>(Arrays.asList(2,3,3,3,1,7,3));
    System.out.println(bg1);
    System.out.println(bg2);
    System.out.println("Union: "+ bg2.union(bg1));
    System.out.println("Union: "+ bg1.union(bg2));
    System.out.println("Intersection: "+ bg1.retainAll(bg2));
    System.out.println("Intersection: "+ bg2.retainAll(bg1));
    System.out.println("Diff: "+ bg1.removeAll(bg2));
    System.out.println("Diff: "+ bg2.removeAll(bg1));
    System.out.println(bg1.addAll(bg2));
    System.out.println(bg1.containsAll(bg2));
    int s = 0;
    for(Integer x : bg1) s += x;
    System.out.println("Sum "+bg1+" = "+s);
  }
}
```

This concludes our discussion of the Map interface implementations. The exercises to follow will explore other issues.

Chapter 13: Graphs

A graph G comprises a set V of vertices and a set E of edges. Each edge in E is a pair (x,y) of vertices in V.

Given the set of vertices $V = \{1,2,3,4,5\}$ and the set $E = \{(1,2),(1,3),(2,5),(3,4),(2,4)\}$ we can draw a graph of $G = (V,E)$ as follows:



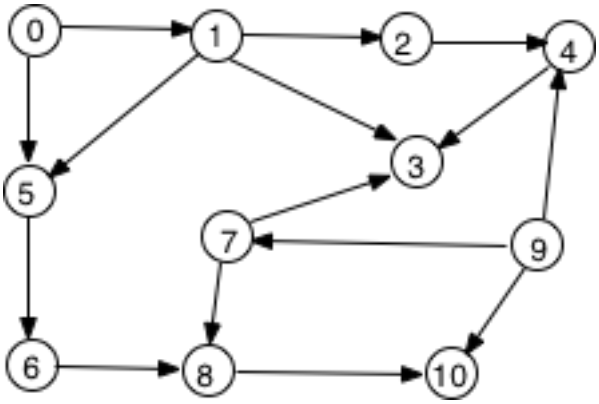
The **size** of G is the number of vertices in V. The **order** of G is the number of edges in E. The minimum order of G is 0 (empty graph) and the maximum number is $n*(n-1)/2$ (complete graph).

The **degree** d_a of vertex a is the number of vertices to which a is linked in E. The min degree of a is 0 and the max degree is $n-1$.

A path from vertex a to vertex b is an ordered sequence of distinct vertices $v_0, v_1, \ldots v_m$ in which each adjacent pair v_{j-1},v_j is linked by an edge. The length of the path is m.

A ***cycle*** is an ordered sequence of vertices $a = v0, v1, \ldots, vm = a$ in which each adjacent pair $(vj-1,vj)$ of vertices is linked by an edge, and $v0, v1, \ldots, vm-1$ are distinct . The **length** of the cycle is *m*

Given below is graph G.

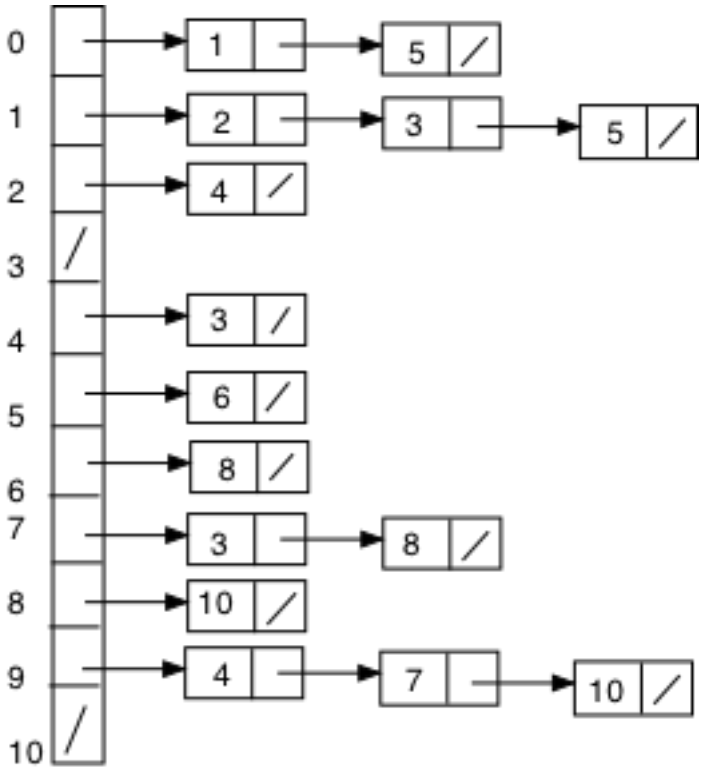


V = {0,1,2,3,4,5,6,7,8,9,10}

E = {(0,1),(0,5), (1,2),(1,3),(1,5),(2,4),(4,3),(5,6),(6,8),(7,3)(7,8),(8,10),
(9,4),(9,7),(9,10)}

The size of *G* = 11, the order of *G* = 15, the degree of 0 = 2, the degree of 1 = 3, etc. There are no cycles in the graph and the path from vertex 0 to vertex 3 is 0-1-2-4-3.

Graphs can be represented by what are called **adjacency lists**. An adjacency list is an array of linked lists where the indices correspond to the vertices in the graph. The size of the array is the number of vertices or nodes in the graph. The adjacency list for the graph G above is:



Graph Traversals

Typically there are two different traversals: a **depth first traversal** and a **breadth first traversal**. Both traversals visit all vertices in the graph. We look at each in turn.
Formally, depth first progresses by expanding the first vertex of the search tree going deeper and deeper until a goal vertex is found, or until I finds a vertex that has no edges. Then the search backtracks, returning to the most recent vertex it hasn't visited. In a non-recursive implementation, all vertices are added to a stack so that additional adjacent vertices may be visited, if any. If none, then the vertex is popped from the stack.
The iterative algorithm is:

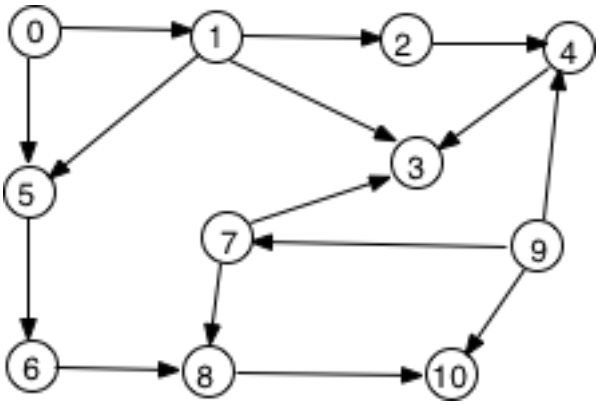
```
depthFirst(Vertex v){
  mark v visited;
  process v;
  push v on stack st;
  while (!st.isEmpty() ){
        let u = next unvisited adjacent vertex of st.top();
        mark u visited;
        process u;
        push u on stack st;
        if (all vertices of st.top() visited)
          st.pop();
  }
}
```

To use this function on a graph of size `gSize` use the following loop:

```
for(Vertex v = 0; v < gSize; v++){
  if(! marked[v])
    depthFirst(v);
}
```

We assume the existence of a `boolean` array `marked` of length `gSize`.

A depth first traversal of the given graph gives: 0, 1, 2, 4, 3, 5, 6, 8, 10, 7, 9.



On the web I found the following apt description of depth first traversal.

The search is similar to searching maze of hallways, edges, and rooms, vertices, with a string and paint. We fix the string in the starting we room and mark the room with the paint as visited we then go down the an incident hallway into the next room. We mark that room and go to the next room always marking the rooms as visited with the paint. When we get to a dead end or a room we have already visited we follow the string back a room that has a hall way we have not gone through yet.

A **breadth first traversal** visits all the vertices at a given level before visiting the vertices at the next level. An iterative algorithm for breadth first uses a queue to store next level vertices while processing higher-level vertices.
The iterative algorithm is:

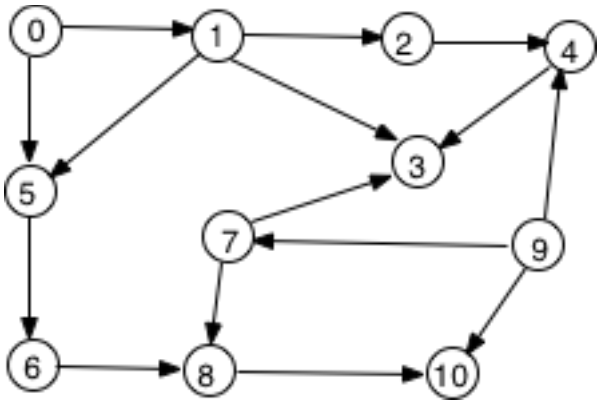
```
breadthFirst(Vertex v){
  mark v visited;
  process v;
  for(u : v.list()){
    if(!marked(u)){
      process u;
      mark u visited; q.add(u);
  }
  while (!q.isEmpty() ){
       u = q.front(); q.leave();
       for(t : u.list())
         if(!marked(t)){
           process t;
```

```
                mark(t); q.add(t);
            }
        }
    }
```

To use this function on a graph of size gSize use the following loop:

```
for(Vertex v = 0; v < gSize; v++){
  if(! marked[v])
    breadthFirst(v);
}
```

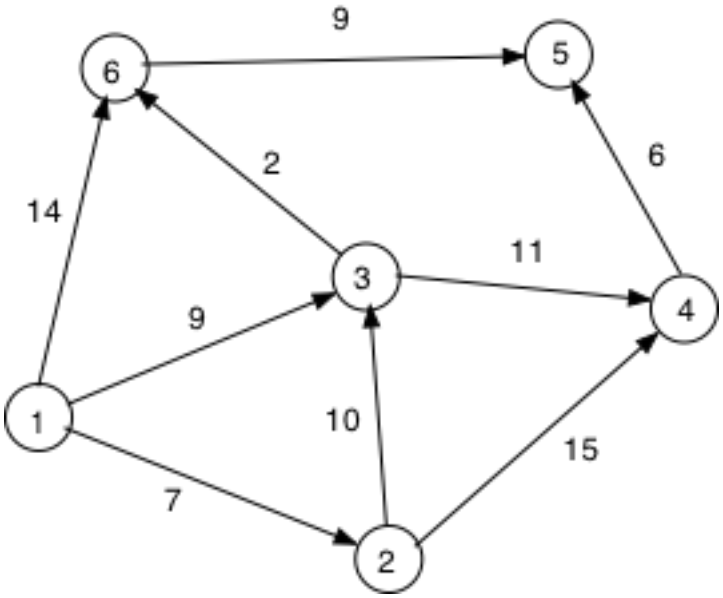
A breadth first traversal of the given graph gives: 0, 1, 5, 2, 3, 6, 4, 8, 10, 7, 9.



Dijkstra's Algorithm: The Shortest Path Problem

Given a weighted graph comprising a set of vertices V, a set of edges E and a set of weights C specifying weights $c_{i,j}$ for edgers (i, j) in E. We are also given a starting vertex v in V.
The one-to-all shortest path problem is the problem of determining the shortest path from node v to all the other nodes in the graph.

Given below is a directed graph with 6 vertices. Each edge has an associated weight that represents distance.



Dijkstra's algorithm starts by assigning some initial values for the distances from vertex v and to every other vertex in the graph. It operates in steps, where at each step the shortest distance from node v to another vertex is determined. Each vertex has a state that consists of two features: *distance value* and *status label*. The distance value of a node is a scalar representing an estimate of its distance from vertex v. The status label is an attribute specifying whether the distance value of a vertex is equal to the shortest distance to vertex v or not. Its status label is *Permanent* (**p**) if its distance value is equal to the shortest distance from vertex v; otherwise, its status is *Temporary* (***t)*** At each step one node is designated as *current*.

There are three steps or stages in the algorithm.

Step 1. Initialization

Label each vertex as follows:

(1) Starting vertex v is given zero distance and status permanent, represented by (0,p).
(2) All other vertices have status (∞, t)

Designate the vertex v as the current vertex.

Step 2. Distance Value Update and Current Node Designation Update

Let i be the index of the current vertex.

Find the set J of vertices with temporary labels that can be reached from the current vertex i by a link (i, j). Update the distance values of these vertices using the formula:

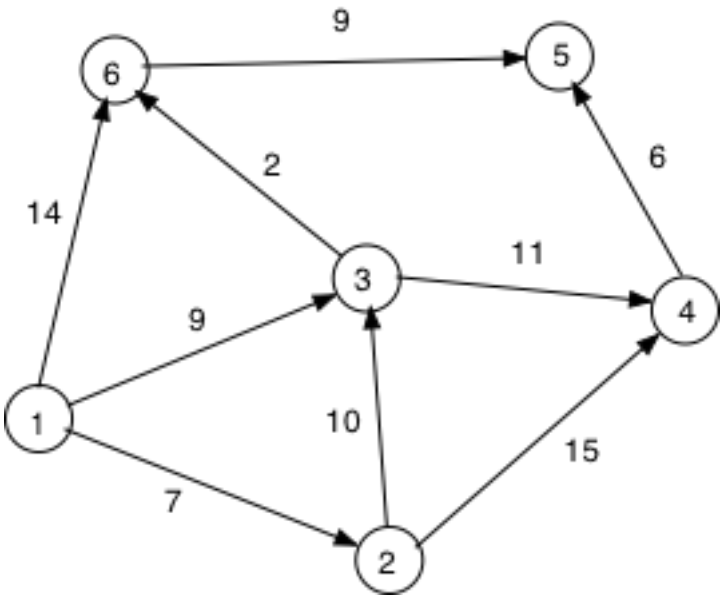
for each $j \in J$, the distance value $dj = \min\{dj, di + c_{ij}\}$, where c_{ij} is the cost of link (i, j), as given in the graph.

Taking vertex j that has the smallest distance value dj among all nodes $j \in J$, change its status to permanent and designate it as the current node.

Step 3. Termination

If all nodes that can be reached from starting vertex v have been labeled permanent then stop. If any temporary labeled vertex cannot be reached from the current vertex change all temporary labels to permanent. Otherwise, go to **Step 2**.

Applying Dijkstra's algorithm to the given graph we find the shortest path to all vertices starting at vertex 1.



Chapter 14: External Data Structures

In this chapter we look at ways of organizing and managing data on external storage devices. Storing data on disk provides persistence and allows different programs access to the same data set. For example, program A might write to the data set and program B might read from it. The important point is that the data persists over time and is not lost when the program terminates. Before we discuss the different file types we look at how to access the external file system on disks. This is covered under the heading files.

Files

The operating system provides a set of commands that allow an end user to create and maintain an electronic filing system on their computer. Using these commands an end user can create a directory structure, copy files from one directory to another or from a hard disk to a floppy disk, etc. They can delete files, rename files, move files from one directory to another or copy files. On Windows systems all of these actions are usually done by means of a program such as *Explorer* that provides a view of the file system with commands listed on dropdown menus or provided by right mouse clicks. Each file on disk has a set of properties or attributes that uniquely identify it, describe its type and control access rights to its contents. It is identified by its pathname, it can be a directory or an ordinary file, it can be hidden or visible, it can be read only or read-write. These properties can be accessed and set directly by using system calls provided by the operating system or by using commands provided by programming languages. Java provides the package java.io that has a number of classes and associated methods that can be used in programs to access the file system on your disk, create new files, read information stored on file and process it, write new information to a file, etc.

The File class provides a number of methods that can be used to check the properties of a file. It is possible to determine if a particular file exists on disk, what its size in bytes is, when it was last modified, whether it is a file or a directory, whether it can be written to, etc. The following table lists the methods for this class together with their semantics.

Method	Semantics
File(String pathname)	Constructor that takes a pathname as argument.
File(String name)	Constructor that takes a relative pathname as argument. In this case it works relative to the current directory.
boolean exists()	Returns `true` if the file or directory

	exists; false otherwise. Neither of the constructor methods throw exceptions if the named file does not exist. This method can be used to determine this.
boolean canRead()	Returns true if the file is readable; false otherwise.
boolean canWrite()	Returns true if the file can be written to; false otherwise.
boolean isDirectory()	Returns true if the object is a directory; false otherwise.
boolean isFile()	Returns true if the object is a file; false otherwise.
boolean delete()	Delete the file and return true; otherwise false.
boolean createNewFile()	Create a new file with zero bytes.
deleteOnExit()	When the Java runtime system exits the file will be deleted.
createTempFile(String pfx, String sfx))	Create a temporary file with specified prefix and suffix in the temp directory.
String getName()	Return the name of the file or directory.
String getParent()	Returns the name of the parent directory.
String getPath()	Returns the path of the file or directory.
String getAbsolutePath()	Returns the absolute pathname of the file.
length()	Returns the length of the file in bytes.
String[] list()	Returns a String array of files in the current directory
boolean renameTo(File f)	Renames the file or directory.
boolean setReadOnly()	Sets the file to read only status.
long lastModified()	Returns the last modification time of the file or directory.

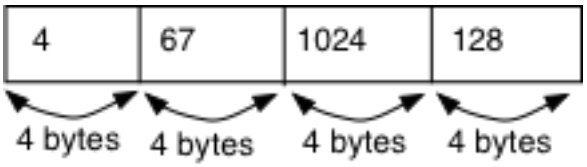
The following program illustrates how to use some of these methods. It requests an end user to enter a file name and returns information about the file. If it exists and is a normal file it checks its readability status and prints its pathname. If the file is a directory it lists its contents.

```
import java.io.*;
import java.util.Scanner;
class FileTest{
  public static void main(String args[]){
    Scanner in = new Scanner(System.in);
    File f1;
    System.out.print("File name: ");
    String fName = in.nextLine();
    f1 = new File(fName);
    if(f1.exists()){
      // output some properties
      if(f1.canRead())
        System.out.println("Readable file");
      else
        System.out.println("Not readable file");
      if(f1.isDirectory()){
        // list file contents
        System.out.println("Directory listing:");
        String files[] = f1.list();
        for(int j= 0; j < files.length; j = j + 1)
          System.out.println(files[j]);
       }
       else{
         // display absolute path
        String path = f1.getAbsolutePath();
        System.out.println("Path: "+path);
        }
    }
       else
      System.out.println("File does not exist");
     }
}
```

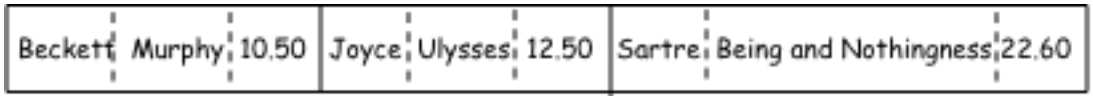
Types of File

Information in a file may be encoded as either text or binary data. The text in a text file is organized into a sequence of lines separated by a line separator. This separator is a character

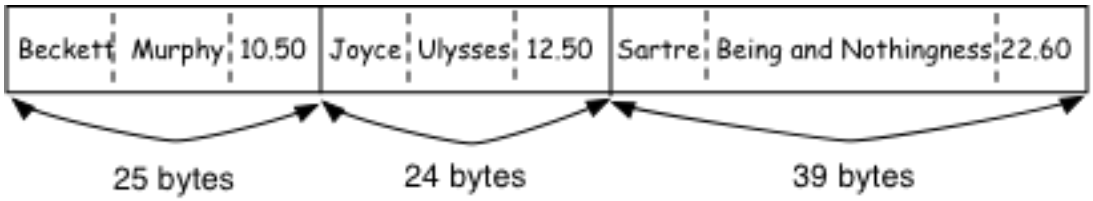
determined by the actual operating system you are using. When the file is subsequently opened the data can be read line by line. All data is stored in ASCII format. A binary file, on the other hand, is a file in which data is stored in the same format as it is held in the memory of the computer. Binary files are simply memory dumps. They store both numeric data and strings in binary format. Binary files are created and managed directly by end user programs and are not readable by general-purpose programs such as text editors. Opening a binary file in a text editor produces strange characters interspersed with pieces of legible text. The data in binary files must be retrieved in the order and format in which it was written to make sense. This means that writer and reader programs must use the same data format when processing the information on file. Either file format can be used to store any collection of data. However, there are differences. Text files can be viewed using any text editor and they are highly portable between machines and different programs. Their weakness is that they use a lot of additional memory to store non-text data, such as integers, doubles and booleans, and they can only be processed in sequence. A further limitation is that non-text data has to be converted to the correct format as it is read. Binary files are not portable nor can they be processed with editors. They require specialized programs to read and write the data. But the storage cost is small: an integer always occupies exactly 32 bits, whereas, its' textual representation might require 100 bits. A binary file stores data in chunks or records and this allows for easy access to individual records. This means that individual records on file can be organized so that the cost of searching is minimized. We can picture a binary file as an array of items where each item occupies a number of bytes on the disk. A file of integer values containing the numbers 4, 67, 1024, 128 would be:



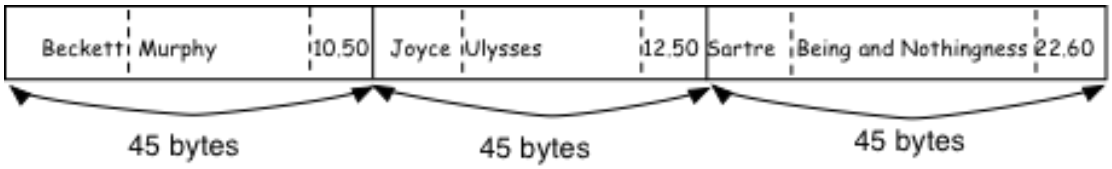
A file storing data that models a book, where each book has an author, a title and a cost would be:



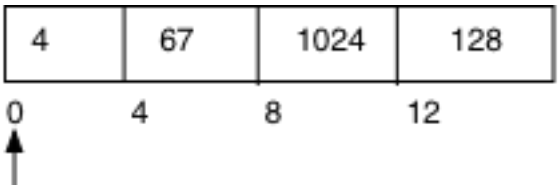
In this file there are 3 *records* each consisting of three *fields* or *attributes*. The number of bytes required to store integer values is 4, based on the fact that integers on 32-bit machines use 32 bits or 4 bytes. The number of bytes required to store decimal values (type double) is 64 bits or 8 bytes. However, the required space for storing strings is based on the number of characters in the string. If we use UTF format then the required storage in bytes is the number of characters in the string plus 2. The additional two bytes are used to store the length of the given string. Each record in the file of books has a different length based on this encoding of strings.



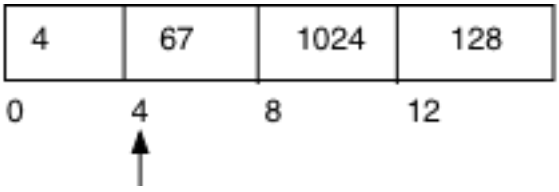
The fact that individual records have different storage requirements makes it virtually impossible to access individual records by index. This means that we would have to process a file sequentially to find an individual book. An alternative approach is to give each string attribute a fixed dimension that sets a maximum length for the string. String attributes whose actual lengths are less than their dimensions are padded with spaces and strings whose length is greater than their dimension are truncated. Using this approach each record will be the same length. The additional cost in storage is worth it because it makes access to individual records possible. The file can be treated as an indexed sequence of records just like an array. Each record in the file has a fixed size of 45 bytes making it possible to calculate the position of a given record.



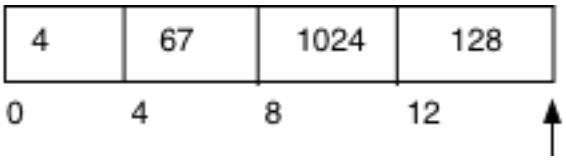
Every file has associated with it a *file pointer* that is maintained by the run-time system. This is an integer variable (of type `long`) containing a `byte` offset in the file. When a file is opened for reading the file pointer indexes the start of the file:



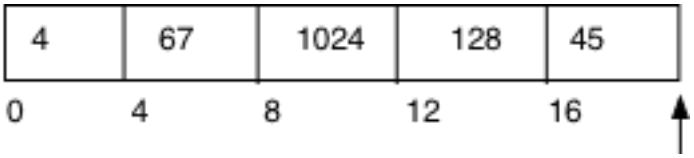
In this case the file stores integers and each integer is 4 bytes in size. All reading or writing always takes place at the location indexed by the *file pointer*. When an item is read from a file it is retrieved from the position indexed by the *file pointer*, and the *file pointer* is advanced by the length, in bytes, of the item. Reading a file has no affect on its contents. After a single integer read operation on the above file, the value 4 is returned and the *file pointer* is positioned as follows:



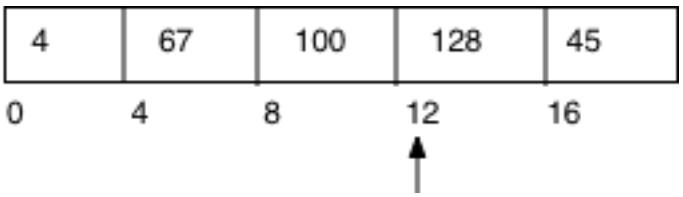
Three more successive reads will leave the *file pointer* positioned at the end of the file.



Should a write operation take place when the *file pointer* is at the end of the file, then new items are appended to the file. If 45 is written to the file depicted above the result is:



If the *file pointer* is positioned at a given position then the item at that position is overwritten. Suppose the file pointer is indexed at byte 8 and 100 is written to the file. Then the new state of the file is:



Text Files

The package java.io is used to handle all input and output in a uniform way. Fundamental to this package is the concept of a stream. A stream represents a flow of data in one direction from a source to a sink. The source might be a file on disk and the sink a variable in your program. Data travels from the source to the sink. In the java.io package there are numerous classes that can be used for reading and writing streams of data. The FileReader class can be used to open a text-based stream for reading. It has two main constructor methods: public FileReader(String fileName) and public FileReader(File file). Both of these constructors throw a FileNotFoundException and, consequently, can only be used within a try-catch block. To read the actual data stream, an instance of the BufferedReader class can be used. This class takes an instance of the FileReader class as an argument in its constructor and creates a buffer to facilitate processing of the input stream. It is a type of filter stream that reads ahead and buffers a certain amount of text. One of its most useful methods is the readLine() method because it returns a string representing the next line of text in the buffer. It does not return any line termination characters. In this way it can be used to read a text file line by line. When the end of the stream is reached it returns null. Because readLine() throws an IOException it must also be used within a try-catch block. When processing is complete the input stream is closed by the close() method. An alternative

approach is to use the Scanner class from java.util. Both approaches are illustrated in the sample program listed below that simply displays its own code.

```
import java.io.*;
import java.util.*;
class DisplayText {
  public static void main(String[] args) {
    Scanner keyIn = new Scanner(System.in);
    System.out.print("File name: ");
    String fname = keyIn.nextLine();
    try {
      File f1 = new File(fname);
      FileReader fr = new FileReader(f1);
      BufferedReader in = new BufferedReader(fr);
      String s = in.readLine();
      while (s != null) {
        System.out.println(s);
        s = in.readLine();
      }
      in.close();
    }
    catch (IOException e) {
      System.out.println(e.getMessage());
    }
    //Alternative approach
    try {
      File f1 = new File(fname);
      FileReader fr = new FileReader(f1);
      Scanner ins = new Scanner(fr);
      while(ins.hasNextLine()){
       String s = ins.nextLine();
       System.out.println(s);
      }
      ins.close();
    }
    catch (IOException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

For a second example we write a program that reads a text file and computes the number of words in the file. It may be assumed that words are separated by white space. A solution is

given by using the StringTokenizer class to process each line of text read. The method countTokens() can be used to return the number of words in the string.

```
import java.io.*;
import java.util.*;
class WordCount{
  public static void main(String[] args){
    String s;
    int wCount = 0; // number of words so far in file
    StringTokenizer t;
    try{
      File f1 = new File("keyboardData.txt");
      FileReader fr = new FileReader(f1);
      BufferedReader in = new BufferedReader(fr);
      s = in.readLine();
      while (s != null) {
        t = new StringTokenizer(s);
        wCount = wCount + t.countTokens();
        s = in.readLine();
      }
      in.close();
    }
    catch (IOException e) {
      System.out.print(e.getMessage());
    }
    System.out.println("Word count: "+wCount);
  }
}
```

For a final example we process a file, called *Results.dta*, of student examination results for a class of students. Each line of the file contains a student number followed by individual marks for each of four subjects. The first few lines might be:

```
H0001 40 34 48 40
H0002 30 89 40 67
H0003 24 78 47 54
H0004 45 46 46 89
```

```
import java.io.*;
import java.util.*;
class Report{
  public static void main(String args[]){
    ArrayList<Result> results = new ArrayList<>();;
    try{
      File f1 = new File("Results.dta");
```

```
      FileReader fr = new FileReader(f1);
      BufferedReader in = new BufferedReader(fr);
      String s = in.readLine();
      while (s != null) {
        StringTokenizer t = new StringTokenizer(s);
        String c = t.nextToken();
        int tMark = 0; int res = 0;
        while(t.hasMoreTokens()){
          Integer mrk = new Integer(t.nextToken());
          tMark += mrk; res += 1;
        }
        results.add(new Result(c,tMark/res));
        s = in.readLine();
      }
      in.close();
   }

    catch (Exception e) {
      System.err.println(e);
    }
    System.out.println("Averages: "+results);
  }
}

class Result{
   private String code;
   private int average;
   public Result(String c, int a){
     code = c;
     average = a;
   }
   String code(){return code;}
    int average(){return average;}
    public String toString(){
   return "("+code + "," + average+")";
    }
}
```

The FileWriter class can be used to open a character-based stream for writing. It has three constructor methods:

Method	Semantics
FileWriter(String fName)	Open a new file for writing with the name fName or truncate fName if it

	already exists.
FileWriter(File file)	Open a new file or truncate file if it already exists.
FileWriter(String fName, boolean append)	Open fName for appending if append has the value true.

All of these constructors throw IOExceptions and, hence, must be used within a try-catch block. To actually write text to the stream an instance of the PrintWriter class is used. This class has a number of print methods that output formatted data to the stream. To write line separated text the most useful of these methods is println(). It takes any primitive type as argument, outputs its value and finishes the line.

```
import java.io.*;
import java.util.Scanner;
class KeyboardReader{
  public static void main(String[] args){
   Scanner in = new Scanner(System.in);
   System.out.println("Enter test:");
   try{
     FileWriter f1 = new FileWriter("keyboardData.txt");
     PrintWriter out = new PrintWriter(f1);
     String text = in.nextLine();
     while (text.length() > 0) {
       out.println(text);
       text = in.nextLine();
     }
     out.close();
   }
   catch(Exception e){e.printStackTrace();}
  }
}
```

To append new text to this file simply open it using:

```
FileWriter f1 = new FileWriter("keyboardData.txt", true);
```

Exercise

Question 1

Write a program that prompts an end user to enter the name of a text file and displays the contents of the file on the screen.

Question 2

Using an editor create a text file of words. Write a program that reads this file and computes the number of occurrences of the word *fun* in the text.

Question 3

Using an editor create a text file of integer values. Write a program that reads the file and computes the number of even values in the file.

Question 4

Write a program that generates 100 random integer values and writes them to a file called *RandomInt.txt*.

Question 5

Write a program that reads the data in the file *RandomInt.txt* and computes its sum.

Binary Files in Java

In Java there are a number of different binary types and we will focus on just one of these that supports both sequential and random access to records on file. To class RandomAccessFile supports both reading and writing of data on file. The following table lists its methods and their semantics.

Method	Semantics
RandomAccessFile(File name, String mode)	Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument. Mode stands for *read/write* or *read* only. These are represented by the strings: "rw", "r". It throws FileNotFoundException.
RandomAccessFile(String name, String mode)	Creates a random access file stream to read from, and optionally to write to, a file with the specified name. Mode is as above. It throws FileNotFoundException.
void close() throws IOException	Closes file. Files must be closed when processing of data ends to avoid loss of data.
void writeInt(int x) throws IOException	Writes x to the file and advances the file pointer by 4.
void writeDouble(double x) throws IOException	Writes x to the file and advances the file pointer by 8.
void writeBoolean(boolean x) throws IOException	Writes x to the file and advances the file pointer by 1.
void writeChar(char x) throws IOException	Writes x to the file and advances the file pointer by 2.
void writeUTF(String x) throws IOException	Writes x to the file and advances the file pointer by x.length() + 2. Explained above.
int readInt() throws	Reads and returns an integer from the

IOException, EOFException	position of the *file pointer* in the file. The *file pointer* is advanced by 4 bytes.
double readDouble() throws IOException, EOFException	Reads and returns a decimal number from the position of the *file pointer* in the file. The *file pointer* is advanced by 8 bytes.
boolean readBoolean() throws IOException, EOFException	Reads and returns a Boolean value from the position of the *file pointer* in the file. The *file pointer* is advanced by 1 byte.
char readChar() throws IOException, EOFException	Reads and returns a character from the position of the *file pointer* in the file. The *file pointer* is advanced by 2 bytes.
String readUTF() throws IOException, EOFException	Reads and returns a string written to the file using writeUTF from the position of the *file pointer* in the file. The *file pointer* is advanced by the number of bytes inserted by writeUTF.
long length() throws IOException	Returns the current size of the file in bytes.
long getFilePointer() throws IOException	Returns the current value of the *file pointer*.
long setLength(int n) throws IOException	Truncates the file to the first n bytes. Really only used to clear the file by setting its length to zero using setLength(0).
void seek(long n) throws IOException	Sets the file-pointer n bytes offset, measured from the beginning of this file, at which the next read or write occurs. For file f, f.seek(f.length()) sets the file pointer just after the end of the file, ready to append new records. f.seek(0) sets the *file pointer* to the start of the file.

There is no end of file marker. Therefore, when iterating over the file while reading we use the guard f.getFilePointer < f.length().

Our first example creates a file of 1000 integer values in a file called *integers.dat*, calculates the sum of the items on file, reads the 100th number and then appends a sequence of numbers, 0..9. The primary purpose of this example is to illustrate the methods listed in the table above. The comments in the code should suffice to explain the semantics.

```
import java.io.*;
import java.util.*;
public class IntegerFile {
   public static void main(String[] args) {
   try{
         //create new file or open existing one
      RandomAccessFile f1 = new RandomAccessFile("integers.dat","rw");
      int j = 0;
      while(j < 1000){
        int x = (int)(Math.random()*100);
        f1.writeInt(x);
        j++;
      }
      System.out.println("File length in bytes = "+f1.length());

    //=========================================================
      //read file sequentially calculating the sum of its elements
      f1.seek(0); int sum = 0;
      while(f1.getFilePointer() < f1.length()){
       int x = f1.readInt();
       sum = sum + x;
      }
      System.out.println("Sum of numbers = "+sum);
     //=========================================================
       //get the 100th number.
     //Multiplication by 4 necessary because each integer is 4 bytes long
      f1.seek(99*4);
      int x = f1.readInt();
      System.out.println("100th number = "+x);
    //=========================================================
      //append some numbers
      f1.seek(f1.length());
      for(j = 0; j < 10; j++) f1.writeInt(j);
      //print out these numbers
      f1.seek(f1.length() - 10*4);
      for(j = 0; j < 10; j++){
          x = f1.readInt();
         System.out.printf("%3d",x);
      }
```

```
      f1.close();
    }
    catch(IOException e){e.printStackTrace();}
  }
}
```

A Sequential File of Records

To demonstrate how to manage a sequential file of complex data we create a file of Person where each person has a name, an age and a salary. This example also shows how to implement a class that *knows* how to write its state to file. The two important methods in this class defined below are write and read. Both methods take a RandomAccessFile reference as argument. The write method uses writeUTF to write the name attribute to file and read retrieves it with readUTF. The order of writing and reading must be the same. The code is:

```
public void write(RandomAccessFile f){
    try{
        f.writeUTF(name);
        f.writeInt(age);
        f.writeDouble(salary);
    }
    catch(IOException e){e.printStackTrace();}
}
public void read(RandomAccessFile f){
    try{
        name = f.readUTF();
        age = f.readInt();
        salary = f.readDouble();
    }
    catch(IOException e){e.printStackTrace();}
}
```

A collection class, called Group, manages an ArrayList of Person. This class assumes that the memory available is sufficient to store all known persons. The list of persons in memory can be written to, and read from, a named file by a user program through methods write and read that both take a file name as argument. This means that the list of persons can persist over time and can be used by any program that *knows* its' encoding and that has access to sufficient memory to store all persons. The write method creates a new file or opens an existing file setting the *file pointer* to the start of the file. Any data on file will be over-written

by this method. An iterator is used to traverse the people ArrayList and each record is written to file using its write method – p.write(fh). The code is:

```
public void write(String fName){
   try{
    RandomAccessFile fh = new RandomAccessFile(fName,"rw");
    Iterator<Person> it = people.iterator();
    while(it.hasNext()){
       Person p = it.next();
       p.write(fh);
    }
    fh.close();
  }
  catch(IOException e){e.printStackTrace();}
}
```

The read method uses an instance of the File class to check if the file fName exists. If it does, it clears the data from the array, opens the file for reading and then iterates over the file using p.read(fh) to read successive records. The code is:

```
public void read(String fName){
   File fn = new File(fName);
   try{
    if(!fn.exists()){
     System.out.println("File does not exist");
    }
    else{
       people.clear();
       RandomAccessFile fh = new RandomAccessFile(fn,"r");
       while(fh.getFilePointer() < fh.length()){
      Person p = new Person();
      p.read(fh);
      people.add(p);
       }
       fh.close();
    }
  }
    catch(IOException e){e.printStackTrace();}
}
```

Both classes together with a simple test program are given below. The test program actually creates two Group instance variables. The first group, gr, is used to create the data for the

file *painters.dat*. This file is then read into the second group, ngr, and updated in memory before copying it to disk. It is important to note that the records on file are not all the same length. The size of a record is name.length+2+4+8 bytes. The actual size of a record is, therefore, the number of characters in the name plus 14 bytes. As a consequence, this data file must be processed sequentially from left to right.

```
import java.io.*;
import java.util.*;
public class SequentialTesting {
   public static void main(String[] args) {
   Group gr = new Group();
   gr.add(new Person("Matisse",30,1000.0));
     gr.add(new Person("Picasso",20,1900.0));
     gr.add(new Person("Michelangelo",90,9000.0));
     gr.add(new Person("Cardavaggio",60,9001.0));
     gr.write("painters.dat");
     System.out.println(gr.toString());
     Group ngr = new Group();
     ngr.read("painters.dat");
     System.out.println(ngr.toString());
     ngr.add(new Person("Bacon",70,20000.0));
     ngr.write("painters.dat");
     ngr.read("painters.dat");
     System.out.println(ngr.toString());
   }
}
class Group{
   ArrayList<Person> people;
   public Group(){
      people = new ArrayList<Person>();
   }
   public void add(Person p){
      people.add(p);
   }
   public boolean contains(Person p){
      return people.contains(p);
   }
   public String toString(){
      return people.toString();
   }
   public void write(String fName){
      try{
        RandomAccessFile fh = new RandomAccessFile(fName,"rw");
        Iterator<Person> it = people.iterator();
```

```
        while(it.hasNext()){
            Person p = it.next();
            p.write(fh);
        }
        fh.close();
     }
     catch(IOException e){e.printStackTrace();}
  }
  public void read(String fName){
    File fn = new File(fName);
       try{
        if(!fn.exists()){
         System.out.println("File does not exist");
        }
        else{
            people.clear();
            RandomAccessFile fh = new RandomAccessFile(fn,"r");
            while(fh.getFilePointer() < fh.length()){
          Person p = new Person();
          p.read(fh);
          people.add(p);
           }
           fh.close();
        }
     }
     catch(IOException e){e.printStackTrace();}
  }
}
class Person{
    private String name;
    private int age;
    private double salary;
    public Person(String n, int a, double s){
        name = n; age = a; salary = s;
    }
    public Person(){name = ""; age = 0; salary = 0.0;}
    public String toString(){
        return name+" "+age+" "+salary;
    }
    public boolean equals(Object ob){
        Person p = (Person)ob;
        return(name.equals(p.name) && age == p.age);
    }
    public void write(RandomAccessFile f){
        try{
```

```
            f.writeUTF(name);
            f.writeInt(age);
            f.writeDouble(salary);
        }
        catch(IOException e){e.printStackTrace();}
    }
    public void read(RandomAccessFile f){
        try{
            name = f.readUTF();
            age = f.readInt();
            salary = f.readDouble();
        }
        catch(IOException e){e.printStackTrace();}
    }
}
```

Random Access File of Records

To create a random access file we must ensure that all records have the same length. In the case of `String` attributes this means that we have to give each string a dimension denoting its maximum length. To demonstrate this we re-write the class `Person` so that a person's name is limited in length to `40` characters. All names whose length is greater than 40 are truncated when written to file. Using our formula, given above, we also encode the record length as part of the class definition. Both are delimited `static`. The `write` method is the same as above, with the exception of `name` that is either truncated or padded with spaces using the command `String.format("%-40s",name)`. The `read` method is unchanged, except for the use of `trim` to remove any padding that `name` may have acquired during writing. In the event that `name` was truncated during writing, the additional characters are lost.

The class `GroupFile` manages a random access file of Person. All records are stored on file and individual records are only read into memory variables when required. The attribute `length` is used to keep track of the number of records on file. When an existing file is opened, the number of records on file is given by `fh.length()/Person.recordLen`. The constructor takes a file name as argument creates a new file or opens an existing one and sets the *file pointer* to the start of the file. The method `add` appends a new `Person` record to the file by setting the *file pointer* to the end of file and using `p.write(fh)`. The attribute length is also incremented. Method `contains` performs a linear search of the file for a given person `p`. There are two versions of `get`, both return an instance of `Person` or `null`, based on the information provided. Version `get(String name)` searches for a match based on the name of

the person and the other method - get(int ind) – takes a record index as argument and returns the record at ind*Person.recordLen offset in the file by setting the *file pointer* to this position. The code is:

```
public Person get(int ind){//assume >= 0
  try{
     if(ind < length){
        fh.seek(ind*Person.recordLen);
        Person p = new Person();
        p.read(fh);
        return p;
     }
     else return null;
    }catch(IOException e){e.printStackTrace();return null;}
}
```

Method replace takes two Person instances as arguments, searches for record p on file and, if found, replaces it with np. When p is found the *file pointer* has to be reset to the start of the record p to allow over-writing to take place. This is essential to avoid over-writing the successor of p or possibly appending a new record to the file, should p be the last record. This is done using fh.seek(j*Person.recordLen), where j equals the index of p. The code is:

```
public boolean replace(Person p, Person np){
    try{
      fh.seek(0);
      int j = 0; boolean found = false;
      while(j < length && !found){
         Person per = new Person();
         per.read(fh);
         if(per.equals(p)) found = true;
         else j++;
      }
      if(found){
         fh.seek(j*Person.recordLen);
         np.write(fh);
         return true;
      }
      else
        return false;
    }catch(IOException e){e.printStackTrace();return false;}
}
```

The code for each of the classes together with a simple test program is given below.

```
import java.io.*;
import java.util.*;
public class IndexFileTest {
   public static void main(String[] args) {
   GroupFile gr = new GroupFile("painters1.dat");
   gr.add(new Person("Matisse",30,1000.0));
     gr.add(new Person("Picasso",20,1900.0));
     gr.add(new Person("Michelangelo",90,9000.0));
     gr.add(new Person("Cardavaggio",60,9001.0));
     gr.display();
     System.out.println(Person.recordLen);
     System.out.println(gr.length());
     Person p = gr.get(2);
     System.out.println();
     System.out.println(p.toString());
     System.out.println();
     gr.replace(p,new Person("Bacon",50,1234.0));
     gr.display();
     System.out.println();
     p = gr.get("Cardavaggio");
     p.setAge(61);
     gr.replace(p,p);
     gr.display();
   }
}
class GroupFile{
   RandomAccessFile fh;
   private long length;
   public GroupFile(String fName){
      try{
          fh = new RandomAccessFile(fName,"rw");
          length = fh.length()/Person.recordLen;
          fh.seek(0);
      }catch(IOException e){e.printStackTrace();}
   }
   public void add(Person p){
      try{
      fh.seek(fh.length());
      p.write(fh);
      length++;
      }catch(IOException e){e.printStackTrace();}
   }
```

```
public boolean contains(Person p){
    try{
      fh.seek(0);
      int j = 0; boolean found = false;
      while(j < length && !found){
          Person per = new Person();
          per.read(fh);
          if(per.equals(p)) found = true;
          else j++;
         }
         return found;
    }
    catch(IOException e){e.printStackTrace();return false;}
}
public Person get(String name){
    Person p = new Person();
    p.setName(name);
    Person per = null;
    try{
        fh.seek(0);
        int j = 0; boolean found = false;
        while(j < length && !found){
            per = new Person();
            per.read(fh);
            if(per.equals(p)) found = true;
            else j++;
        }
    }
    catch(IOException e){e.printStackTrace();}
    return per;
}
public Person get(int ind){//assume >= 0
  try{
      if(ind < length){
          fh.seek(ind*Person.recordLen);
          Person p = new Person();
          p.read(fh);
          return p;
      }
      else return null;
    }catch(IOException e){e.printStackTrace();return null;}
}
public boolean replace(Person p, Person np){
    try{
      fh.seek(0);
```

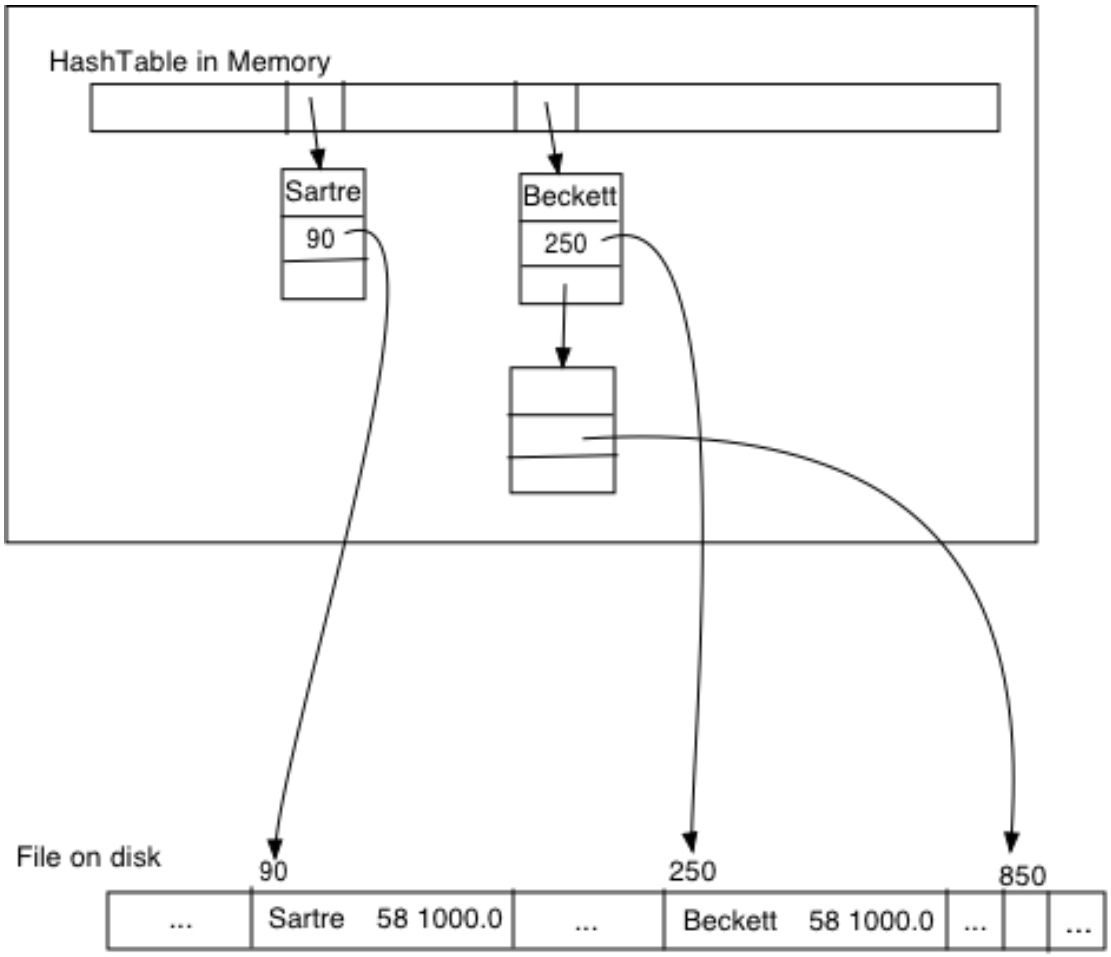
```
        int j = 0; boolean found = false;
        while(j < length && !found){
            Person per = new Person();
            per.read(fh);
            if(per.equals(p)) found = true;
            else j++;
        }
        if(found){
            fh.seek(j*Person.recordLen);
            np.write(fh);
            return true;
        }
        else     return false;
      }catch(IOException e){e.printStackTrace();return false;}
   }
   public long size(){
      return length;
   }
   public long length(){
      long l = 0;
      try{
         l = fh.length();
      } catch(IOException e){e.printStackTrace();}
      return l;
   }
   public void close(){
      try{
       fh.close();
      }catch(IOException e){e.printStackTrace();}
   }
   public void display(){
      try{
        fh.seek(0); int j = 0;
        while(j < length){
            Person p = new Person();
            p.read(fh);
            System.out.println(p.toString());
            j++;
        }
      }catch(IOException e){e.printStackTrace();}
   }
}
class Person{
   static int nameDim = 40;
   static int recordLen = 2+nameDim+4+8;
```

```
    private String name;
    private int age; private double salary;
    public Person(String n, int a, double s){
        name = n; age = a; salary = s;
    }
    public Person(){
        name = ""; age = 0; salary = 0.0;
    }
    public void setAge(int a){ age = a;}
    public void setName(String n){name = n;}
    public void setSalary(double s){salary = s;}
    public String name(){return name;}
    public int age(){return age;}
    public double salary(){return salary;}
    public String toString(){
        return name+" "+age+" "+salary;
    }
    public boolean equals(Object ob){
        Person p = (Person)ob;
        return(name.equals(p.name));
    }
    public int hashCode(){
        return name.hashCode();
    }
    public void write(RandomAccessFile f){
        try{
            String n = String.format("%-40s",name);
            f.writeUTF(n);
            f.writeInt(age);
            f.writeDouble(salary);
        }
        catch(IOException e){e.printStackTrace();}
    }
    public void read(RandomAccessFile f){
        try{
             name = f.readUTF().trim();
             age = f.readInt();
             salary = f.readDouble();
        }
        catch(IOException e){e.printStackTrace();}
    }
}
```

Indexed Random Access Filing System

In the previous example the `get` method, that takes an index as argument, simply calculates the address of the record and returns the record at the given offset. This provides *O(1)* retrieval because it avoids the need to perform a linear search of the file. This is a big advantage because it is extremely slow to search a file on disk record by record. In fact reading from or writing to a disk is orders of magnitude slower than accessing data held in variables in memory. Of course, this kind of performance is only possible when we know the index of the record we are searching for. Given the frequency with which files are searched for information it is important to optimize the cost of retrieval. In the case of files keeping them sorted on disk is not really an option because the cost of doing so is too expensive. Appending new records to an existing file is relatively inexpensive and should not be sacrificed. Therefore, the goal is to find a way to optimize retrieval times. If the file is small we could always copy it to memory and process it there. (Our sequential file example demonstrates this.) But many files are much too large to fit in memory.

It is common, when designing data records to include a *key* field that will be used to uniquely identify that record. Every passport has a unique passport number, every person has a unique social security number and every bank account has a unique bank account number. One of the most common operations on a filing data system is to display the details of a record identified by its unique key value supplied by an end user. To optimize searching we construct what is called a *directory* or *index* for each field on which we may need to search the file. For example, if we intend to search a passport file for records with a given passport number, then we maintain a directory of passport numbers. The *directory* consists of a collection of records where each record has two components: the *key* value for the record, and the *position* of the record on file. The actual byte offset of the record is calculated from the *position*. (An alternative is to store the byte offset of the record directly.) The actual *directory* is constructed from the file of records on disk and is, usually, stored in memory. In typical industrial applications, the size of a directory will be less than 1% of the size of the file and should fit in memory. The directory may be stored in an array or a hash table or any other appropriate data structure. To describe this model we use the `name` field of our `Person` class as the *key* value to search the file with. The diagram below shows two records on file with their entries in a directory using a hash table to store the indexing records. *Sartre* has index *90* and *Beckett* index *250*. To find *Sartre* we look up the entry in the *directory* and then calculate its address on disk by multiplying the index value by `Person.recordLen`. Retrieving the actual record only requires a single disk read.



A class IndexKey encapsulates the relevant data to store in our directory table. The class satisfies the requirement for hash coding and has two constructors. One constructor takes only a name as argument and is used purely for search purposes in the hash table. The code is:

```
class IndexKey{
   String key;
   long index;
   public IndexKey(String s, long k){
      key = s; index = k;
   }
   public IndexKey(String s){
      key = s; index = -1;
   }
   public int hashCode(){
      return key.hashCode();
   }
   public boolean equals(Object ob){
      IndexKey ky = (IndexKey)ob;
      return(key.equals(ky.key));
```

```
    }
    public long getFileIndex(){ return index;}
}
```

The class IndexedGroupFile is similar to the class GroupFile discussed above and provides a similar public interface. It constructs a directory using a hash table to store keys and their indices. The constructor opens the file and reads the file record by record adding new instances of IndexKey to the hTable for each record. The code is:

```
public IndexedGroupFile(String fName){
        try{
          fh = new RandomAccessFile(fName,"rw");
          length = fh.length()/Person.recordLen;
          fh.seek(0);
          //create hash table
          if(fh.length() == 0) //file empty
                hTable = new HashList<IndexKey>(1000);
          else{
            //create index table from file of existing data
            hTable = new HashList<IndexKey>(1000);
            int j = 0;
            while(j < length){
             Person per = new Person();
             per.read(fh);
             IndexKey ky = new IndexKey(per.name(),j);
             hTable.add(ky);
             j++;
            }
         }
      }catch(IOException e){e.printStackTrace();}
    }
```

To retrieve the record for a given name we look up the table and use the index to calculate the byte address of the record on file. The code is:

```
public Person get(String name){
        Person per = null;
        IndexKey ky = hTable.get(new IndexKey(name));
         if(ky == null) return null;
         else{
         try{
            fh.seek(ky.getFileIndex()*Person.recordLen);
            per = new Person();
            per.read(fh);
```

```
        }
        catch(IOException e){e.printStackTrace();}
        return per;
      }
    }
```

Finally, the add method is amended to update the directory when a new person is appended to the file. The code for the class together with a simple test program is given below.

```
class IndexedGroupFile{
   RandomAccessFile fh = null;
   private long length = 0;
   private HashList<IndexKey> hTable = null;
   public IndexedGroupFile(String fName){
      try{
         fh = new RandomAccessFile(fName,"rw");
         length = fh.length()/Person.recordLen;
         fh.seek(0);
         //create hash table
         if(fh.length() == 0)
               hTable = new HashList<IndexKey>(1000);
         else{
               //create index table from file of existing data
               hTable = new HashList<IndexKey>(1000);
            int j = 0;
            while(j < length){
             Person per = new Person();
             per.read(fh);
             IndexKey ky = new IndexKey(per.name(),j);
             hTable.add(ky);
             j++;
            }
      }
     }catch(IOException e){e.printStackTrace();}
   }
   public void add(Person p){
   try{
      fh.seek(fh.length());
      p.write(fh);
      length++;
      hTable.add(new IndexKey(p.name(),length-1));
   }catch(IOException e){e.printStackTrace();}
    }
    public boolean contains(Person p){
```

```
      //just check for key in internal table
      IndexKey ky = hTable.get(new IndexKey(p.name()));
      if(ky != null) return true;
      else return false;
    }
    public Person get(String name){
       Person per = null;
       IndexKey ky = hTable.get(new IndexKey(name));
        if(ky == null) return null;
        else{
        try{
            fh.seek(ky.getFileIndex()*Person.recordLen);
            per = new Person();
            per.read(fh);
        }
        catch(IOException e){e.printStackTrace();}
        return per;
       }
     }
       public void close(){
       try{
         fh.close();
       }catch(IOException e){e.printStackTrace();}
   }
}


import java.util.*;
import java.io.*;
public class HashFileTest {
   public static void main(String[] args) {
    IndexedGroupFile f1 = new IndexedGroupFile("writers.dat");
    f1.add(new Person("Joyce",58,1000.0));
    f1.add(new Person("Sartre",58,1000.0));
    f1.add(new Person("Beckett",58,1000.0));
    f1.add(new Person("Maupassant",58,1000.0));
    f1.add(new Person("Voltaire",58,1000.0));
    System.out.println();
    Person p = f1.get("Beckett");
    System.out.println(p.toString());
   }
}
```

Object Serialization

Serialization is the process of converting a data structure or object state into a binary format that can be stored on file or sent over a network. A *serialized* object is an object represented as a sequence of bytes that includes all data as well as information about its type and the types of data stored in the object (its methods are not included). It is a binary stream of bytes that represent the state of a complete object. This object may be a single instance of a simple class or one that instantiates a collection of items. A complete data structure, such as an ArrayList, can be serialized. A serialized object may be written to a file using what is called an ObjectOutputStream. Once written to file it can be read and *deserialized*, i.e. it can be re-constructed in memory. Several object-oriented programming languages support object serialization. Python, PHP, Java, and the .NET family of languages all support it. Java provides automatic serialization as long as the class implements the java.io.Serializable interface. All classes that implement this interface may be copied to file using object streams. The primary advantage of serialization in relation to persistence is that it makes it very simple to store collections of objects on file. The program given below demonstrates how to do this.

```
import java.io.*;
import java.util.*;
public class ObjectFiles {
   public static void main(String[] args) {
      Library lib = new Library();
      Book b = new Book("Doyle", "The Committments", 10.00);
      lib.add(b);
      b = new Book("Beckett", "Murphy", 8.00);
      lib.add(b);
      b = new Book("Beckett", "Malone Dies", 8.50);
      lib.add(b);
      Iterator<Book> it = lib.iterator();
      while(it.hasNext()){
        b = it.next();
        System.out.println(b.toString());
      }
      lib.save("books.dat");
      Library nLib = new Library("books.dat");
      System.out.println();
      it = lib.iterator();
      while(it.hasNext()){
        b = it.next();
        System.out.println(b.toString());
      }
   }
```

```
}
class Library{
   private ArrayList<Book> bks;
   public Library(){
      bks = new ArrayList<Book>();
   }
   public Library(String fName){
      this.read(fName);
   }
   public void add(Book b){
      bks.add(b);
   }
   public boolean contains(Book b){
      return bks.contains(b);
   }
   public Iterator<Book> iterator(){
      return bks.iterator();
   }
   public void save(String fName){
      try{
         FileOutputStream outStr = new FileOutputStream(fName);
         ObjectOutputStream out = new ObjectOutputStream(outStr);
         out.writeObject(bks);
         out.flush();
         out.close();
      }
      catch (FileNotFoundException e) {e.printStackTrace();}
      catch (IOException e){e.printStackTrace();}
   }
   public void read(String fName){
      try{
       FileInputStream inStr = new FileInputStream(fName);
         ObjectInputStream in = new ObjectInputStream(inStr);
         try{
            bks = (ArrayList<Book>)in.readObject();
         }catch(ClassNotFoundException e){e.printStackTrace();}
         in.close();
       }
      catch (FileNotFoundException e) {e.printStackTrace();}
      catch (IOException e){e.printStackTrace();}
   }
}
class Book implements Comparable<Book>, java.io.Serializable{
   private String title;
   private String author;
```

```
    private double price;
    public Book(String a, String t, double p){
        title = t; author = a; price = p;
    }
    public String getTitle(){return title;}
    public String getAuthor(){return author;}
    public int compareTo(Book bk){
        if(bk == null){return -1;}
        else{
            if(author.compareTo(bk.author) == 0)
                return(title.compareTo(bk.title));
            else
                return(author.compareTo(bk.author));
        }
    }
    public boolean equals(Object ob){
        Book b = (Book)ob;
        if(title.equals(b.title)&& author.equals(b.author))
            return true;
        else
            return false;
    }
    public int hashCode(){
        Double p = new Double(price);
        int h = title.hashCode() + author.hashCode();
        return(h);
    }
    public String toString(){
        String p = Double.toString(price);
        return (author.toString()+" "+title.toString()+" "+p);
    }
}
```