Algorithms and Data Structures for Games Programming

Jonathan G. Campbell
Department of Computing,
Letterkenny Institute of Technology,
Co. Donegal, Ireland.

email: jonathan dot campbell (at) gmail.com, jonathan.campbell@lyit.ie
URL:http://www.jgcampbell.com/adsgp/adsgp.pdf
Report No: jc/09/0002/r

Revision 2.0, Chapter 13 on sets and maps replaces old chapter 13.
2.1 minor typos fixed + loading current versions of Array and BST
current BST uses only operator less-than

10th January 2009
Contents

1 Introduction and Overview .............................................................. 1
  1.1 Aims and Objectives .................................................................... 1
  1.2 Syllabus ....................................................................................... 2
  1.3 Assessment .................................................................................. 2
    1.3.1 Continuous Assessment ........................................................ 2
  1.4 Reading List .................................................................................. 2
  1.5 Outline ......................................................................................... 4

2 Array Containers ............................................................................. 1
  2.1 Introduction ................................................................................... 1
  2.2 An Array class ............................................................................. 1
    2.2.1 Major points to note in Array.h .............................................. 8
    2.2.2 Iterators ................................................................................. 10
  2.3 A Simple Client Program for Array ............................................. 12
  2.4 The Big-Three .............................................................................. 16
    2.4.1 Defence against naive defaults ............................................. 16
  2.5 Overloading the Stream Output Operator .................................... 17
  2.6 std::vector ................................................................................. 17
    2.6.1 Points to note in vectort1.cpp .............................................. 20

3 Analysis of Algorithms ................................................................. 1
  3.1 O Notation (Big-oh) .................................................................... 2
  3.2 Estimating the Growth Rate of an Algorithm ............................. 3
    3.2.1 Simple Statements ................................................................ 3
    3.2.2 Decision ................................................................................. 3
    3.2.3 Counting Loop ....................................................................... 4

4 Sorting and Searching .................................................................... 1
  4.1 Bubble-Sort .................................................................................. 1
    4.1.1 Running Time ....................................................................... 2
  4.2 Selection Sort ............................................................................... 4
    4.2.1 Running Time ....................................................................... 4
  4.3 Insertion Sort ............................................................................... 5
    4.3.1 Running Time ....................................................................... 6
  4.4 Merge Sort .................................................................................... 7
    4.4.1 Running Time ....................................................................... 7
  4.5 Quicksort ....................................................................................... 8
    4.5.1 Running Time ....................................................................... 8
    4.5.2 Standard Library Sort ........................................................... 9
  4.6 Comparison: Bubble, Selection, Insertion and Merge and Quick Sorts ................................. 9
8.5 Proving Termination of Recursive and Other Non-deterministic Algorithms ........ 9
8.6 Divide-and-Conquer ........................................ 10
  8.6.1 Maximum of an Array using Divide and Conquer ................. 10
  8.6.2 Merge Sort ........................................... 13
  8.6.3 Towers of Hanoi ..................................... 14
  8.6.4 Drawing a Ruler .................................... 16
8.7 Trees and Recursion ......................................... 21
  8.7.1 Recursive drawing of a ruler ................................ 21
  8.7.2 Recursive maximum of an array ............................ 21
  8.7.3 Recursive evaluation of a prefix expression ................. 23
8.8 Elimination of Recursion ................................... 23

9 Trees Miscellany .............................................. 1
  9.1 Introduction .............................................. 1
  9.2 n-ary Trees .............................................. 1
  9.3 Game Trees ............................................... 3
    9.3.1 Nim .................................................. 3
    9.3.2 Minimax Algorithm ................................ 6
    9.3.3 Recursive Minimax Algorithm ......................... 7
    9.3.4 Minimax Applied to Tic-tac-toe ....................... 8

10 Simple Pathfinding .......................................... 1
  10.1 Introduction ............................................ 1
  10.2 Deque Data Structure .................................... 1
    10.2.1 Implementation .................................. 1
    10.2.2 Discussion of sDeque.h ............................. 4
  10.3 Path-finding in a Maze — Depth-first and Breadth-first Searching .... 4
    10.3.1 Maze implementation ............................... 5
    10.3.2 Depth-first pathfinding solution ................... 8
    10.3.3 Backtracking ..................................... 11
    10.3.4 Absolute dead end ................................ 12
    10.3.5 Breadth-first pathfinding solution ................. 13
  10.4 Graphs .................................................. 15

11 Graphs ...................................................... 1
  11.1 Introduction ........................................... 1
  11.2 Examples of Graphs ..................................... 2
    11.2.1 General ........................................... 2
    11.2.2 Games and Path-finding ............................ 2
  11.3 Implementation ......................................... 3
  11.4 Graph Traversal ......................................... 3
    11.4.1 Depth-first Traversal ............................... 4
    11.4.2 Breadth-first Traversal ............................ 5
  11.5 Software Implementation ................................ 6
    11.5.1 Basics of Graph.h ................................ 6
    11.5.2 Depth-first Traversal ............................... 10
    11.5.3 Use of Stack for Depth-first? ....................... 11
    11.5.4 Animated Display of the Traversal ................. 12
    11.5.5 Breadth-first Traversal ............................ 13
12 Pathfinding

12.1 Introduction ............................................. 1
12.2 Pathfinding as a graph search problem .................. 1
12.3 Graph search — informed or uninformed? ............. 4
12.4 Graph search pathfinding ............................... 4
   12.4.1 Uninformed graph search ....................... 4
   12.4.2 Breadth-first pathfinding — basic algorithm .... 4
   12.4.3 Informed graph search — add a heuristic .......... 6
   12.4.4 Depth-first pathfinding ......................... 6
12.5 Practical examples of graph-search algorithms .......... 8
   12.5.1 Silly Breadth-first ............................ 9
   12.5.2 Breadth-first .................................. 10
   12.5.3 Distance-first .................................. 11
   12.5.4 ‘Simple’ Heuristic ................................ 12
   12.5.5 Distance Heuristic ............................. 14
   12.5.6 AStar Heuristic .................................. 16
12.6 Algorithms Applied to a Maze problem ................. 17
12.7 Performance Measures .................................. 19
   12.7.1 Performance Measures for Breadth first ......... 19
   12.7.2 Performance Measures for Depth first ........... 21
12.8 Pathfinding in non-tile-based environments .......... 21
12.9 Software Implementations of the Algorithms .......... 22

13 Sets and Maps

13.1 Introduction ............................................. 1
13.2 Implementations ......................................... 2
13.3 Bitvector .................................................. 2
13.4 Hashing and Hash Tables ................................ 2
13.5 Example Use of set ....................................... 3
13.6 Sets need to be able to order elements ............... 5
13.7 Example Use of multiset ................................ 5
13.8 A Name class whose objects we want to insert in set .. 8
13.9 Set of Names ............................................... 11
13.10A Telephone Directory using a map .................... 13
13.11A Telephone Directory using a multimap ............. 16
Chapter 1

Introduction and Overview

This document contains a one semester module on algorithms and data structures, with an emphasis on games programming applications.

1.1 Aims and Objectives

Module Aims To develop an understanding of algorithms and data structures needed by computer games.

Module Learning Outcomes A student who successfully completes this module will be able to:

1. Describe, implement, and apply stack, queue, list, tree, hash, and graph data structures;
2. Describe, implement, and apply common algorithms on the data structures identified in item 1.
3. Analyse the space and time complexity of data structures and algorithms using Big-Oh notation.
4. Explain, implement, and apply recursive algorithms on game trees and graphs, including minimax algorithms.
5. Explain, implement, and apply path finding algorithms.
6. Interpret and modify demonstration computer programs which implement techniques based on learning outcomes 1 to 5.
7. Implement modifications (including design and coding of program fragments) to demonstration programs to appreciably alter their behaviour.
8. Choose and apply appropriate data structures to solve particular application problems.
1.2 Syllabus

Section A. Basic Data Structures and their Computer Representation and Related Algorithms (20%)

Arrays and vectors, stacks; queues; lists; searching; sorting. Binary search tree. Hashing. Big-Oh notation and analysis. Cache issues.

Section B. Further Data Structures and their Computer Representation and Related Algorithms (30%)


Section C. Data Structures for 3D Objects and Scenes (25%)

Software implementation of data structures and algorithms (and object classes) for object representation and game scene management.

Section D. Path Finding in 2D and 3D Environments (25%)


1.3 Assessment

Continuous assessment 60%. Final examination 40%.

1.3.1 Continuous Assessment

Three assignments worth a total of 40% of the module. Class Test(s). 20%

1.4 Reading List

These notes form the essential reading. We will use examples and software demonstrations from Penton’s book (Data Structures for Games Programmers) (Penton 2003). Sherrod’s book on Data Structures and Algorithms for Games Programmers (Sherrod 2007) has a similar objective, but is more elementary and has less games related demonstrations and graphics than has Penton.

For the basics of data structures and algorithms, (Budd 1997) is hard to beat; see also (Horstmann & Budd 2005) and (Budd 1999).
The classic book on Algorithms is (Cormen, Leiserson, Rivest & Stein 2001). Other highly regarded books on algorithms are (Sedgewick 1997) (note that Sedgewick has also a C++ series and a Java series) and (Weiss 1996). See also Knuth’s series (Knuth 1997a) (Knuth 1997b) (Knuth 1998). A recent book that you might like is *Algorithms in a Nutshell* (Heineman, Pollice & Selkow 2008); it has plenty of code examples.

Harel’s book (Harel 2004) is lighter reading than some of those, but an extremely good introduction to the study of algorithms.

For general C++ programming related to games, see (Dickheiser 2007), and (McShaffry 2005).

If you are rusty on C++, the following will help: (Koenig & Moo 2000), (Lippman 2005), (Stroustrup 1997) (all three invaluable reference books on C++), (Meyers 2005), (Meyers 1996) (Eckel 2000) (Eckel 2003), (Wilson 2004), (Cline, Lomow & Girou 1999). You could also look at my C++ notes (Campbell 2007 :).

Fortunately for us, the C++ *Standard Library* (formerly called the *Standard Template Library* (STL)) provides very efficient implementations of the most common data structures and algorithms.

When I’m using *Standard Library* (STL) features on something I’m not too sure about, I have (Josuttis 1999), (Reese 2007), (Meyers 2001) and (Stroustrup 1997) by my right hand.

For both STL and C++, (Stroustrup 1997) and (Lischner 2003) are extremely useful reference books.

Regarding some games specific algorithms, see (Penton 2003) (a poor enough book, but it has actual game examples based on the SDL API), (Dickheiser 2007), (Sherrod 2007); although the coding is in Java, Brackeen’s book that we used in first year (Brackeen, Barker & Vanhelsuwe 2004) will give a good introduction.

If I was asked to recommend one book for games programmers to learn C++, I’d recommend (Dickheiser 2007) (which, incidentally, is a second edition of (Llopis 2003)).

Other books are listed in the bibliography.
1.5 Outline

- First we look at some data structures in detail; we will examine implementation of arrays, lists, stacks, queues and trees; these implementations will provide roughly similar functionality to the STL equivalents; these days, you would rarely ever choose to develop your own in preference to using the standard library, but there are good reasons for knowing the details; for a start, knowing how the data structures are implemented is the key to understanding which one to use to meet a specific requirement; next, you never know when you would end up using a programming language in which you would need to construct your own. And finally, employers are very likely to ask about implementation detail in interviews.

- In parallel with implementing our own data structures, we will examine and demonstrate how to use STL containers; also, we’ll look at some of Penton’s (Penton 2003) implementations;

- Mixed in with this, we will touch on recursion;

- We will implement some simple algorithms such as sorting;

- We will study how to use big-Oh notation to analyse algorithm speed performance; big-Oh is an important way of stating unambiguously how an algorithm performs — rather than something vague like merge-sort is faster than bubble-sort, or even merge-sort is twice as fast as bubble-sort; will the twice as fast be true for an array of length 100, of length 1,000,000? Hardly; the performance will depend on some function of \( N \), the length: \( O(f(N)) \). If we specify the function \( f(.) \), e.g. \( f(N) = N^2 \), \( f(N) = \log N \), then we have something that we can use to predict how long an algorithm will take to complete for an arbitrary data size \( N \).

- Once we have those basics completed, we will start on learning objectives 4 to 7; we will need to study a graph data structure; we will mostly use Penton’s book and code for all this and we will start to concentrate more on the application, i.e. using and applying data structures, than on the detailed implementation.
Chapter 2

Array Containers

2.1 Introduction

We call this chapter array containers for want of a better term. We describe sequence containers which remedy some of the shortcomings of the basic C++ array (e.g. int a[50];).

The two most common sequence containers that we encounter in C++ are std::vector and std::list; we use the general term sequence to signify that the elements are held in strict sequential (linear) order.

The main difference between array-like sequence containers (e.g. std::vector and Array that we develop here) and linked (list-like) sequence containers are the two related characteristics: (i) array-like have random access, e.g. std::cout<< a[i];, whereas lists must be sequentially accessed; and (ii) array-like use contiguous storage, whereas lists used linked storage (e.g. singly and doubly linked lists).

Roughly speaking, you can do anything with an array-like sequence that you can with a linked sequence and vice-versa, but the allocation of storage and random access issues mean that there are major performance drawbacks if you choose the wrong type for your application — i.e. the wrong type may work, but work comparatively very slowly.

Java programmers are aware of the distinction, they have ArrayList and LinkedList.

In this chapter we will comment on the inadequacy of the C++ basic array; we will develop an Array class that performs rather like std::vector; in doing that we will develop some understanding of what goes on inside std::vector, so that when you use std::vector you will have some sympathy for performance issues. Then we will examine use of std::vector itself.

In developing Array, we will identify some inadequacies of contiguous storage that lead to the need for linked storage. We will cover (linked) lists in Chapter 5.

The Array class here is more or less identical to the vector class in (Budd 1997).

2.2 An Array class

In the lecture, we will discuss the inadequacies of plain-vanilla arrays (int a[25];) for the task of sequence container. For example, just look in Array.h at what needs to be done when we
insert (add) an element into the middle of a list, i.e. moving everything else up one. To ask a programmer who is concentrating on doing *sequence operations* to remember this every time is to ask for increased errors and poor productivity.

Figure 2.1 shows the interface of *Array*. Figures 2.2 to 2.5 give the implementation.
#ifndef ARRAYH
#define ARRAYH

#include <iostream>
#include <cassert>

using std::ostream; using std::endl;
typedef unsigned int uint;

template <class T> class Array{
public:
    typedef T* Iterator;
    Array(uint len = 0);
    Array(uint len, const T& val);
    Array(const Array<T>& source);
    ~Array();
    void reserve(uint cap);
    void resize(uint sz);
    Array& operator=(const Array<T>& source);
    void copy(const Array<T>& source);
    T& operator[](uint i) const;
    void push_back(const T& val);
    void pop_back();
    T back() const;
    void insert1(uint pos, uint n, const T& val);
    uint erase(uint pos);
    Iterator begin(){ return dat_;}
    Iterator end(){ return dat_ + sz_;}
    void insert(Iterator itr, uint n, const T& e);
    uint size() const;
    bool empty() const;
    uint capacity() const;
private:
    uint sz_, cap_;
    T* dat_;
};

template <class T>
ostream& operator<<(ostream& os, const Array<T>& a);

Figure 2.1: Declaration of Array.
template <class T>
Array<T>::Array(uint sz) : sz_(sz), cap_(sz) {
    if(cap_ == 0) {dat_ = 0; return;}
    dat_ = new T[cap_];
    assert(dat_ != 0);
    T zero = T();
    for(uint i = 0; i < sz_; ++i) dat_[i] = zero;
}

template <class T>
void Array<T>::reserve(uint cap) {
    //std::cout<< "*reserve* cap = "<< cap<< " cap_ = "<< cap_<< std::endl;
    if(cap <= cap_) return;
    T* newdat = new T[cap];
    assert(newdat != 0);
    for(uint i = 0; i < sz_; ++i) newdat[i] = dat_[i];
    delete [] dat_;
    dat_ = newdat;
    cap_ = cap;
}

template <class T>
void Array<T>::resize(uint sz) {
    assert(sz <= cap_);
    if(sz <= sz_) return;
    else if (sz <= cap_){
        T zero = T();
        for(uint i = sz_; i < sz; ++i) dat_[i] = zero;
        sz_ = sz;
    }
    else {
        T* newdat = new T[sz];
        assert(newdat != 0);
        for(uint i = 0; i < sz_; ++i) newdat[i] = dat_[i];
        T zero = T();
        for(uint i = sz_; i < sz; ++i) newdat[i] = zero;
        delete [] dat_;
        dat_ = newdat;
        cap_ = sz_ = sz;
    }
}

Figure 2.2: Array implementation, part 1
```cpp
template <class T>
Array<T>::Array(uint sz, const T& val)
    : sz_(sz), cap_(sz) {
    if(cap_ == 0) {dat_ = 0; return;}
    dat_ = new T[cap_];
    assert(dat_ != 0);
    for(uint i = 0; i < sz_; ++i) dat_[i] = val;
}

template <class T>
Array<T>::Array(const Array<T>& source) {
    copy(source);
}

template <class T>
Array<T>::~Array() {
    delete[] dat_; 
    dat_ = 0;
}

template <class T>
Array<T>& Array<T>::operator=(const Array& source) {
    //std::cout << "copy ctor" << std::endl;
    if(this != &source) {
        delete[] dat_; 
        copy(source);
    }
    return *this;
}

template <class T>
void Array<T>::copy(const Array<T>& source) {
    sz_ = cap_ = source.size();
    if(cap_ == 0) {dat_ = 0; return;}
    dat_ = new T[cap_];
    assert(dat_ != 0);
    for(uint i = 0; i < sz_; ++i) dat_[i] = source.dat_[i];
}

template <class T>
void Array<T>::insert1(uint pos, uint n, const T& val) {
    uint sz = sz_ + n;
    assert(cap_ >= sz);
    uint id = sz_ - 1; // dest
    uint is = sz_ - 1; // source
    for(; is > pos - 1; --is, --id) dat_[id] = dat_[is];
    for(id = pos; id < pos + n; id++) dat_[id] = val;
    sz_ = sz;
}
```

Figure 2.3: Array implementation, part 2
template <class T>
uint Array<T>::erase(uint pos) {
    for(uint i = pos + 1; i < sz_; ++i) dat_[i-1] = dat_[i];
    --sz_;
    return pos;
}

template <class T>
void Array<T>::insert(Iterator pos, uint n, const T& e) {
    uint sz = sz_ + n;
    assert(cap_ >= sz);
    Iterator itrd = dat_ + sz - 1; // dest
    Iterator itrds = dat_ + sz_ - 1; // source
    for(; itrds != pos-1; --itrds, --itrd) *itrd = *itrds;
    for(itrd = pos; itrd != pos + n; ++itrd) *itrd = e;
    sz_ = sz;
}

template <class T>
void Array<T>::push_back(const T& val) {
    //std::cout<< "*push_back* cap_ = "<< cap_<< " sz_ = "<< sz_<< std::endl;
    if(!(cap_ > sz_)) reserve(2*cap_);
    dat_[sz_] = val;
    ++sz_;
}

template <class T>
void Array<T>::pop_back() {
    assert(sz_ > 0);
    --sz_;
}

template <class T>
T Array<T>::back() const {
    assert(sz_ > 0);
    return dat_[sz_ - 1];
}

template <class T>
T& Array<T>::operator[](uint i) const {
    assert(i < sz_);
    return dat_[i];
}

template <class T>
uint Array<T>::size() const {
    return sz_;
}
template <class T>
bool Array<T>::empty() const {
    return sz_ == 0;
}

template <class T>
uint Array<T>::capacity() const {
    return cap_; 
}

// notice no Array<T>:: -- *not* a member
template <class T>
ostream& operator<<(ostream& os, const Array<T>& a) {
    uint sz = a.size();
    uint cap = a.capacity();
    os << "[" << sz << ", " << cap << "]" << ";
    for(uint i = 0; i < sz; i++) {
        os << a[i];
        if(i != sz - 1) os <<", ";
    }
    os << "])" << endl;
    return os;
}
#endif

Figure 2.5: Array implementation, part 4
2.2.1 Major points to note in Array.h

1. Note the difference between size(), szₜ and capacity(), capₜ; size(), szₜ is the used size of the sequence, while capacity(), capₜ is what it can grow to before we need to allocate more memory.

2. When using Array (and std::vector) it is always a good idea to use reserve to allocate either the size that you know you will need, or a decent chunk at the beginning.

3. Notice the difference between reserve and resize.

4. Notice that when push_back detects that we are at full capacity, it reserves double the current capacity; this might work well or badly, depending on the application. I’m not sure how std::vector handles this.

[Here we repeat some messages from Chapter 9 of (Campbell 2007).]

5. Destructors. A destructor is called, automatically, when control reaches the end of the block in which the object was declared, i.e. when the object goes out of scope. The compiler will always provide adequate destruction of stack-based objects, but, for heap-based objects, proper destructor memory management must be provided if garbage and memory-leaks (or worse, dangling pointers) are to be avoided.

6. Copy constructor. A copy constructor is called when the object is passed (by value) to and from functions. Again, for classes which use stack memory, the compiler will always provide an adequate copy constructor. For heap-based objects the case is quite analogous to that of the destructor: proper constructor memory management must be provided.

7. Assignment. Assignment operator (the ‘=’ operator) needs treatment similar to the copy constructor.

8. The Big-Three. The C++ FAQs (Cline et al. 1999) uses the term the Big-Three for these three functions: copy constructor, assignment operator, and destructor. More on this in section 2.4.

This is because, for classes that use heap storage, it is almost certainly necessary to explicitly program all three. If they are not programmed, the compiler will provide default versions which will probably not do what you would wish; moreover the inadequacies of these defaults may be most subtle, and may require quite determined and skilled testing to detect.

9. We use assert to report errors if allocations were unsuccessful; new returns a null pointer if it is unsuccessful, e.g. due to resources of free memory having become exhausted. This may not be ideal, but the error message that assert issues is a lot more helpful than what will happen if we charge on and ignore the fact that we have run out of memory.

10. Function copy.

```cpp
void Array<T>::copy(const Array<T>& source){
    szₜ = capₜ = source.size();
    if(capₜ== 0){datₜ = 0; return;} // capₜ == 0
    datₜ = new T[capₜ];
    assert(datₜ!= 0);
    for(uint i= 0; i< szₜ; ++i)datₜ[i] = source.datₜ[i];
}
```
(a) This is very similar to the default constructor — except that this time we have passed another object to be copied.

(b) Notice that we have passed a reference (Array& source). If we don’t we’ll end up constructing many multiple copies, each of which must also be destroyed. And when the object becomes large, as may be the case for an Array the performance drain can be considerable.

(c) Of course, we guarantee the safety of the referenced object (in the caller) by making the reference const.

11. Copy constructor.

    Array::Array(const Array& source){
        copy(source);
    }

Here, all the work is done by copy. Again notice the use of reference and const.

12. Destructor.

    Array::~Array(){
        delete [] dat_;
    }

(a) Here we use delete to release the allocated memory.

(b) Owing to the fact that dat_ points to an array, we must use delete [].

13. Assignment operator ‘=’.

    Array& Array::operator=(const Array& source){
        if(this!= &source){ // beware a= a;
            delete [] dat_;
            copy(source);
        }
        return *this;
    }

(a) Let us say we have two Array objects, x, y and x = y. Then this assignment is exactly equivalent to

        x.operator=(y);

(b) Again notice the use of a reference and const.

(c) Since we can envisage x = x, however improbable, we have to be careful to check whether this is the case — and if it is, do nothing.

(d) this is an implicit pointer variable which points at the object itself. Thus if(this!= &source) checks if the calling object and the source object share the same memory location — and so are the same object!

(e) return *this; returns the object (actually a reference, see next comment).
Why does operator= return Array&?

Answer. In C and C++ it is standard for an assignment to have the value of the object assigned, e.g.

```cpp
int x = y = 10;
```

we want the same for objects.

And, as usual, we want the efficiency of reference passing.

14. If you are unsure about operator overloading, refer to (Campbell 2007)

15. Iterators are described in section 2.2.2.

### 2.2.2 Iterators

Because of the fact that the internal representation is a C++ built-in array, we can do everything we want to using array subscripting, overloading the array subscript operator [], e.g. `a[index] = 10;` and integer subscripts.

But the STL collections use the more general concept of an *iterator*. We will see that iterators are made to behave like pointers, i.e. you can dereference an iterator, and you can do pointer-like arithmetic on them.

1. Normally an iterator is implemented as a class, but in the case of *Array* it is easy to declare an iterator as

   ```cpp
typedef T* Iterator;
```

2. Because *Iterator* is declared within the scope of *Array*, when we define an iterator we use the scope resolution operator `::`

   ```cpp
   Array<int>::Iterator itr;
   ```

3. We need an iterator that points to the beginning of the array

   ```cpp
   Iterator begin(){ return dat_;}
   ```

   It simply returns a pointer to the first element of data array.

4. We need an iterator that points **one element past the end of the array**;

   ```cpp
   Iterator end(){ return dat_ + sz_;}
   ```

5. One element past the end of the array is the convention; you *never* dereference this iterator value. The usual pattern for iterating over a collection is as follows:
Array<int>::Iterator itr;
for(itr = c5.begin(); itr!= c5.end(); ++itr){
    cout<< *itr<< ' ';
}

itr!= c5.end() is our way of checking that we are not at the end — really one element past the end.
This is very like the normal array pattern:

    int a[n];
    for(int i = 0; i< n; ++i){
        cout<< a[i]<< ' ';
    }

which could be written:

    int a[n];
    for(int i = 0; i!= n; ++i){
        cout<< a[i]<< ' ';
    }

or even:

    int a[n];
    for(int* pos = &a[0]; pos!= &a[0]+ n; ++pos){
        cout<< *pos<< ' ';
    }

or:

    int a[n];
    for(int* pos = a; pos!= a + n; ++pos){
        cout<< *pos<< ' ';
    }

6. Here is insert written using subscripts

    template <class T>
    void Array<T>::insert1(uint pos, uint n, const T& val) {
        uint sz = sz_ + n;
        assert(cap_>= sz);
        uint id= sz - 1; // dest
        uint is= sz_ - 1; // source
        for(; is> pos-1; --is, --id)dat_[id] = dat_[is];
        for(id = pos; id< pos+n; id++)dat_[id] = val;
        sz_ = sz;
    }
7. And here it is using Iterator

template <class T>
void Array<T>::insert(Iterator pos, uint n, const T& e) {
    uint sz = sz_ + n;
    assert(cap_ >= sz);
    Iterator itrd = dat_ + sz - 1; // dest
    Iterator itrs = dat_ + sz_ - 1; // source
    for (; itrs != pos - 1; --itrs, --itrd)*itrd = *itrs;
    for (itrd = pos; itrd != pos + n; ++itrd)*itrd = e;
    sz_ = sz;
}

We’ll have a good deal more to say about iterators when we get to Chapter 5.

2.3 A Simple Client Program for Array

Figures 2.6 and 2.7 give a program, ArrayT1.cpp which uses Array.
#include "Array.h"
using std::cout; using std::endl; typedef unsigned int uint;

int main()
{
    Array <int> c1(5);
    cout << "Array <int> c1(5): " << c1;

    Array <int> c2(5, 127);
    cout << "Array <int> c2(5, 127): " << c2;

    Array <double> c3(3, 3.14159);
    cout << "Array <double> c3(3, 3.14159): " << c3;

    Array <Array<int> > c4(3);
    cout << "Array <Array<int> > c4(3): " << c4;

    c4[0] = c1; c4[1] = c2;
    cout << "c4[0] = c1, c4[1] = c2: " << c4; cout << endl;

    c1.reserve(20);
    cout << "c1.reserve(20): " << c1;

    c1.resize(10);
    cout << "c1.resize(10): " << c1;

    c1.push_back(22);
    cout << "c1.push_back(22): " << c1;

    c1.insert1(2, 5, 33);
    cout << "c1.insert1(2, 5, 33): " << c1;

    c1.push_back(44);
    cout << "c1.push_back(44): " << c1;

    c1.pop_back();
    cout << "c1.pop_back(): " << c1;

    uint j = c1.erase(4);
    cout << "j = c1.erase(4): " "j = " "j" << " c1 = " "c1;

    for(uint i = 0; i < 20; ++i){
        c1.push_back(i);
        cout << "c1.push_back(i = " "i" "): " << c1;
    }
}

cout << "c1.back(): " "c1.back(); cout << endl;
// continued ....}
Array<int> c5;
c5.reserve(10);
for(uint i = 0; i < 5; ++i){
    c5.push_back(i);
    cout << "c5.push_back(i = " << i << ": " << c5;
}

cout << "using Iterator" << endl;
Array<int>::Iterator itr;
for(itr = c5.begin(); itr != c5.end(); ++itr){
    cout << *itr << ', '
}
cout << endl;

cout << "Array<int>::Iterator itr1 = c5.begin() + 2 " << endl;
cout << "c5.insert(itr1, 5, 33) " << endl;
c5.insert1(2, 5, 33);
cout << c5;

cout << endl;
return 0;
}

Figure 2.7: ArrayT1.cpp part 2
And here is the output from ArrayT1.

```cpp
Array <int> c1(5): [5, 5]{ 0, 0, 0, 0, 0}
Array <double> c3(3, 3.14159): [3, 3]{ 3.14159, 3.14159, 3.14159}
Array <Array<int>> c4(3): [3, 3]{ [0, 0]{  }, [0, 0]{  }, [0, 0]{  }  
  c4[0] = c1, c4[1] = c2: [3, 3]{ [5, 5]{ 0, 0, 0, 0, 0}, [5, 5]{ 127, 127, 127, 127, 127}  
  , [5, 5]{ 127, 127, 127, 127, 127}, [0, 0]{  }  }  
}
c1.reserve(20): [5, 20]{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}  
c1.resize(10): [10, 20]{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}  
c1.push_back(22): [11, 20]{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 22, 22}  
c1.insert(2, 5, 33): [16, 20]{ 0, 0, 33, 33, 33, 33, 33, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 22}  
c1.push_back(44): [17, 20]{ 0, 0, 33, 33, 33, 33, 33, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 22, 44}  
c1.pop_back(): [16, 20]{ 0, 0, 33, 33, 33, 33, 33, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 22}  
j = c1.erase(4): j = 4  c1 = [15, 20]{ 0, 0, 33, 33, 33, 33, 33, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 22}  
c1.push_back(i = 0): [16, 20]{ 0, 0, 33, 33, 33, 33, 33, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 22, 0}  
c1.push_back(i = 1): [17, 20]{ 0, 0, 33, 33, 33, 33, 33, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 22, 0, 1}  
  ... etc. ...  
c1.push_back(i = 19): [35, 40]{ 0, 0, 33, 33, 33, 33, 33, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 22, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19}  
c1.back(): 19

```
c5.push_back(i = 0): [1, 10]{ 0}  
c5.push_back(i = 1): [2, 10]{ 0, 1}  
c5.push_back(i = 2): [3, 10]{ 0, 1, 2}  
c5.push_back(i = 3): [4, 10]{ 0, 1, 2, 3}  
c5.push_back(i = 4): [5, 10]{ 0, 1, 2, 3, 4}  
using Iterator
0 1 2 3 4
Array<int>::Iterator itr1 = c5.begin() + 2  
c5.insert(itr1, 5, 33)  
[10, 10]{ 0, 1, 33, 33, 33, 33, 33, 33, 0, 3, 4}  
```
2.4 The Big-Three

The C++ FAQs (Cline et al. 1999) introduced the term the Big-Three for the three functions: copy constructor, assignment operator, and destructor.

This is because, for classes that use heap storage, it is almost certainly necessary to explicitly program all three. If they are not programmed, the compiler will provide default versions which will probably not meet the requirements of client programs. Nevertheless, the inadequacy of these defaults may be most subtle, and may require very detailed testing to detect.

The lack of a proper destructor would be particularly difficult to detect — it simply causes garbage, whose effect is to leak memory, which may not be detected until the class is used in some application which runs for a long time, e.g. an operating system, or an embedded control program.

Other than the Big-Three care must be exercised with comparison, e.g. equality check ==. If left to its own devices the compiler will provide a shallow compare, which compares just the explicit member(s).

2.4.1 Defence against naive defaults

If the developer of a heap-based class is quite sure that assignment, =, will never be required, it still may be quite dangerous to trust that client programmers will never be tempted to use it; and, normally, as we have said above, the compiler will provide a naive default – but silently, with no warning of its possible inadequacy.

Fortunately, there is a simple and sure defence; this is to declare a stub of the function, and to make it private, i.e.

```
private:
  Array& operator = (const Array & rhs){};
```

This means that any client program which unwittingly invokes an assignment will be stopped by a compiler error; client programs are not allowed to access private members.

This would work also for compare, ==.

Nevertheless, it is hard to envisage a class which can operate successfully without a proper copy constructor; likewise destructor. These will have to be programmed.
2.5 Overloading the Stream Output Operator

The stream output operator << is overloaded as follows:

```cpp
// notice no Array<T>:: -- *not* a member
template <class T>
ostream& operator<<(ostream& os, const Array<T>& a){
    uint sz = a.size();
    uint cap = a.capacity();
    os << "[ " << sz << ", " << cap << "]\n";
    for(uint i=0; i<sz; i++){
        os << a[i];
        if(i!=sz-1)os<<", ";
    }
    os<<"]" << endl;
    return os;
}
```

ostream is a output-stream, so that the parameter os is an output-stream object; the familiar cout is one such object.

It is important, to make it conform to its pattern of operation for the built-in types, that << returns a reference to the ostream object. This allows concatenated calls to it, e.g.:

```cpp
cout<< x<< y<< z<< ";"<< endl;
```

If you are feeling confused about the distinction between member functions and non-member functions such as ostream& operator<< above, please consult Chapter 10 of (Campbell 2007).

2.6 std::vector

As stated previously, the chief reason for developing Array is as an educational experience. std::vector will do everything that Array does only faster and with much less chance of a bug lurking somewhere in its implementation.

If you read games programming books of more than five years ago, you may find ambivalence to the standard library (STL) — for a start, not all compilers included it, and in addition there was suspicion about its efficiency.

That’s in the past. I cannot imagine any reason why anyone would ever roll their own array class — except for educational or some very special reasons.

Figures 2.8 and 2.9 show a program which uses std::vector in almost the same way as Figure 2.6 uses Array.
/* ----- vectorT1.cpp ---------------------------------- 
from Arrayt1.cpp, j.g.c. 2008-01-15 
----------------------------------------------------*/

#include <vector> #include <iostream>
#include <ostream> #include <iterator>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> c1(5);
    cout << "vector <int> c1(5): ";
    copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " ")); cout << endl;

    vector<int> c2(5, 127);
    cout << "vector <int> c2(5, 127): ";
    copy(c2.begin(), c2.end(), ostream_iterator<int>(cout, " ")); cout << endl;

    vector<double> c3(3, 3.14159);
    cout << "vector <double> c3(3, 3.14159): ";
    copy(c3.begin(), c3.end(), ostream_iterator<double>(cout, " ")); cout << endl;

    c1.reserve(20); cout << "c1.reserve(20): ";
    copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " ")); cout << endl;

    c1.resize(10); cout << "c1.resize(10): ";
    copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " ")); cout << endl;

    c1.push_back(22); cout << "c1.push_back(22): ";
    copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " ")); cout << endl;

    c1.insert(c1.begin() + 2, 5, 33);
    cout << "c1.insert(2, 5, 33): ";
    copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " ")); cout << endl;

    c1.push_back(44); cout << "c1.push_back(44): ";
    copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " ")); cout << endl;

    c1.pop_back(); cout << "c1.pop_back(): ";
    copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " ")); cout << endl;

    c1.erase(c1.begin()); cout << endl;
    cout << "c1.erase(c1.begin()): " << "c1 = ";
    copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " ")); cout << endl;

    for(size_t i = 0; i < 20; ++i){
        c1.push_back(i); cout << "c1.push_back(i = " << i << ": ");
        copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " ")); cout << endl;
    }
}

Figure 2.8: Use of std::vector
2–18
Figure 2.9: Use of std::vector

copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " "));
cout<< endl;

uint n = c1.size();
for(uint i = 0; i!= n; ++i){
    cout<< c1[i]<< " ";
}  
cout<< endl;

vector<int>::iterator it;
for(it = c1.begin(); it!= c1.end(); ++it){
    cout<< *it<< " ";
}  
cout<< endl;

cout<< "\nsort(c1.begin(), c1.end())"<< endl;
sort(c1.begin(), c1.end());
copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " "));
cout<< endl;

return 0;
}
2.6.1 Points to note in vectort1.cpp

1. `#include <vector>`.

2. Unless you want to fully qualify the type as `std::vector`, you must include `using namespace std`.

3. You can *randomly access* elements of vector using [] notation, for example `cout<< c1[2]`, see Figure 2.9.

4. You can access elements of vector using and *iterator*, for example

   ```cpp
   vector<int>::iterator it;
   for(it = c1.begin(); it!= c1.end(); ++it){
       cout<< *it<< " ";
   }
   ```

5. We declare the *iterator* as `vector<int>::iterator it;` because that *iterator* is declared within the scope of vector; recall `Array< T >::Iterator` above; `::` is the scope resolution operator.

6. When we come to using `std::list`, we will have to use an *iterator* because `std::list` does not support *random access*.

7. An *iterator* is meant to behave rather like a pointer: (a) `++it` and `--it` move the *iterator* forwards and backwards; (b) `*it` access the element that it refers to.

8. `c1.begin()` is an *iterator* that references the first element of `c1`.

9. `c1.end()` is an *iterator* that references *one element past the last element* of `c1`. Hence `it!= c1.end()`.

10. Although we show explicit use of *iterator* and [] indexed random access, the standard library gives us alternatives, for example

    ```cpp
    copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " "));
    ```

11. The standard library algorithms have been written in such a manner that they can be applied to *pointers* as well as *iterators*.

12. Notice how to sort a vector, see also Chapter 4:

    ```cpp
    sort(c1.begin(), c1.end());
    ```
Chapter 3

Analysis of Algorithms

This chapter is a slight modification of notes provided by Robert Lyttle of Queen’s University Belfast.

In considering the trade-offs among alternative solutions to problems, an important factor is the efficiency. Efficiency, in this context, is measured in terms of memory use and time taken to complete the task. Time is measured by the number of elementary actions carried out by the processor in such an execution. In the interests of brevity, this course discusses only time efficiency.

It should be noted that there is often a trade-off between time and memory; often, you can buy time performance (speed) by using extra memory.

It is difficult to predict the actual computation time of an algorithm without knowing the intimate details of the underlying computer, the object code generated by the compiler, and other related factors. But we can measure the time for a given algorithm, language compiler and computer system by means of some carefully designed performance tests known as benchmarks.

It is also helpful to know the way the running time will vary or grow as a function of the problem size — a function of the number of elements in an array, the number of records in a file, and so forth. Programmers sometimes discover that programs that have run in perfectly reasonable time for the small test sets they have used, take extraordinarily long when run with real world sized data sets or files. These programmers were deceived by the growth rate of the computation.

For example, it is common to write programs whose running time varies with the square of the problem size. Thus a program taking, say, 1 second to sort a list of 1000 items will require not two (2), but four (4) seconds for a list of 2000 items. Increasing the list size by a factor of 10, to 10,000 items, will increase the run-time to $10^2 = 100$ seconds. A list 100,000 items will require 10,000 ($10^4$) seconds, or about 3 hours, to complete. Finally, 1,000,000 items (e.g. a telephone directory for a small country) will need $10^6$ seconds (almost two weeks) to finish! This is a long time compared to the one second taken by the 1000 item test.

This example shows that it makes sense to be able to analyse growth rates and to be able to predict (even roughly) what will happen when the problem size gets large.
3.1 O Notation (Big-oh)

Algorithmic growth rates are expressed as formulae which give the computation time in terms of the problem size \( N \). It is usually assumed that system dependent factors, such as the programming language, compiler efficiency and computer speed, do not vary with the problem size and so can be factored out.

Discussions of growth rate normally use the Big-oh notation (growth rate, order of magnitude).

The most common growth rates we will encounter are the following:

- \( O(1) \), or constant;
- \( O(\log N) \), or logarithmic (logarithm usually taken to the base 2);
- \( O(N) \), or linear (directly proportional to \( N \));
- \( O(N \log N) \), pronounced \( N \log N \);
- \( O(N^2) \), or quadratic (proportional to the square of \( N \)).

The table that follows shows the value of each of these functions for a number of different values of \( N \). It shows that as \( N \) grows, \( \log N \) remains quite small with respect to \( N \) and \( N \log N \) grows fairly rapidly, but not nearly as large as \( N^2 \).

In Chapter 4 we will see that simple searching grows as \( O(N) \) (linear), but a binary search grows as \( O(\log N) \). We also see that most good sorting algorithms have a growth rate of \( O(N \log N) \) and that the slower, more obvious, ones are \( O(N^2) \).

<table>
<thead>
<tr>
<th>log2(N)</th>
<th>N</th>
<th>N log2 N</th>
<th>N^2</th>
<th>2^N</th>
<th>N!</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>20.000E-1</td>
<td>10.000E-1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>20.000E-1</td>
<td>4</td>
<td>40.000E-1</td>
<td>20.000E-1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>80.000E-1</td>
<td>16</td>
<td>16.000E+0</td>
<td>24.000E+0</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>24.000E+0</td>
<td>64</td>
<td>25.600E+1</td>
<td>40.320E+3</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>64.000E+0</td>
<td>256</td>
<td>65.536E+3</td>
<td>20.923E+12</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>16.000E+1</td>
<td>1024</td>
<td>42.950E+8</td>
<td>26.313E+34</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>38.400E+1</td>
<td>4096</td>
<td>40.960E+2</td>
<td>18.447E+18</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>89.600E+1</td>
<td>16384</td>
<td>34.028E+7</td>
<td>12.689E+88</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
<td>20.480E+2</td>
<td>65536</td>
<td>65.536E+3</td>
<td>38.562E+214</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
<td>46.080E+2</td>
<td>262144</td>
<td>42.950E+8</td>
<td>*</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
<td>10.240E+3</td>
<td>104857</td>
<td>13.408E+153</td>
<td>*</td>
</tr>
<tr>
<td>11</td>
<td>2048</td>
<td>22.528E+3</td>
<td>4194304</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>12</td>
<td>4096</td>
<td>49.152E+3</td>
<td>16777216</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>13</td>
<td>8192</td>
<td>10.650E+4</td>
<td>67108864</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>14</td>
<td>16384</td>
<td>22.938E+4</td>
<td>268435456</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>15</td>
<td>32768</td>
<td>49.152E+4</td>
<td>1073741824</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>16</td>
<td>65536</td>
<td>10.486E+5</td>
<td>4294967296</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

* too large to be computed in a double variable

aEp means \( a \times 10^p \)
3.2 Estimating the Growth Rate of an Algorithm

In estimating performance we can take advantage of the fact that algorithms are developed in a structured way — that is, they combine simple statements into complex blocks in four useful ways:

- **sequence**, or writing one statement below the other;
- **selection**, or the well known if-then or if-then-else;
- **repetition**, including counting loops, while loops, etc.;
- **method calls**.

In the rest of this section, some typical algorithm structures are considered and their $O()$ estimated. The problem size is denoted by $N$ throughout.

### 3.2.1 Simple Statements

An assignment statement is an example of a simple statement. If we assume that the statement contains no method calls (whose execution time may, of course, vary with problem size), the statement takes a fixed amount of time to execute. This type of performance is denoted by $O(1)$ because when we factor out the constant execution time we are left with one.

**Sequence of Simple Statements**

A sequence of simple statements obviously takes an amount of time equal to the sum of the times it takes each individual statement to execute. If the performances of the individual statements are $O(1)$, then so is that of the sum.

### 3.2.2 Decision

When estimating performance, the *then* clause and the *else* clause of a conditional structure are considered to be independent, arbitrary structures in their own right. Then the larger of the two individual **big-Ohs** is taken to be the **big-Oh** of the decision.

A variation of the decision structure is the *switch* structure, really just a multi-way if-then-else. Thus in estimating performance of a *switch*, we just take the largest **big-Oh** of all of the switch alternatives.

Performance estimation can sometimes get a bit tricky. For example, the condition tested in a decision may involve a method call, and the timing of the method call may itself vary with problem size.


### 3.2.3 Counting Loop

A counting loop is a loop in which the counter is incremented (or decremented) each time the loop is iterated. This is different from some loops we shall consider later, where the counter is multiplied or divided by a given value.

If the body of a simple counting loop contains only a sequence of simple statements, then the performance of the loop is just the number of times the loop iterates. If the number of times the loop iterates is constant — i.e. independent of the problem size — then the performance of the whole loop is $O(1)$. An example of such a loop is:

```java
def (int i = 0; i < 5; i++)
  statements with $O(1)$
}
```

On the other hand if the loop is something like:

```java
def (int i = 0; i < N; i++)
  statements with $O(1)$
}
```

the number of times the loop iterates depends on $N$, so the performance is $O(N)$.

**Discussion of Single Loop – $O(n)$** Consider the following code in which the individual statements/instructions are numbered:

```java
int sum=0; //s1
  //s2 //s3 //s4
for (int i = 0; i < N; i++)
  //s5
}
```

Here, $s_1$, and $s_2$ are performed only once; $s_3$, $s_4$ and $s_5$ are performed $N$ times. Hence, associating a time $t_j$ with instruction $j$, we have:

$$t_{tot} = t_1 + t_2 + N(t_3 + t_4 + t_5)$$

Normally, it will be the case that $s_5$ will be the most expensive; however, just to be fair, let us assume that all instructions take the same time — 1 unit, e.g. 1 microsecond. Let us see how $t_{tot}$ behaves as $N$ get large.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$T_{tot}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>100</td>
<td>302</td>
</tr>
<tr>
<td>1000</td>
<td>3002</td>
</tr>
</tbody>
</table>
Hence, we see that it is the term \( N(t_3 + t_4 + t_5) \) which becomes dominant for large \( N \). In other words, we can write:
\[
t_{tot} = cN \text{ + negligible terms}
\]
and that the algorithm has a running time of \( O(n) \).

Consider the double counting loop:

```java
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        statements with \( O(1) \)
```

The outer loop is iterated \( N \) times. But the inner loop iterates \( N \) times for each time the outer loop iterates, so the body of the inner loop will be iterated \( N \times N \) times, and the performance of the entire structure is \( O(N^2) \).

The next structure:

```java
for (int i = 0; i < N; i++)
    for (int j = 0; j < i; j++)
        statements with \( O(1) \)
```

looks deceptively similar to that of the previous loop. Again, the outer loop iterates \( N \) times. But this time the inner loop depends on the value of \( i \) (which depends on \( N \)): if \( i = 1 \), the inner loop will be iterated once, if \( i = 2 \) it will be iterated twice, and so on, so that, in general, if \( i = N \), the inner loop will iterate \( N \) times. How many times, in total, will the body of the inner loop be iterated? The number of times is given by the sum:
\[
0 + 1 + 2 + 3 + \ldots + (N - 2) = \sum_{i=1}^{N-2} i
\]

Noting that \( \sum_{i=1}^{N} i = \frac{(N+1)N}{2} \), the summation above is equivalent to \( \frac{(N-2)(N-1)}{2} = \frac{N^2 - 3N + 2}{2} \). The performance of such a structure is said to be \( O(N^2) \), since for large \( N \) the contributions of the \( \frac{3N}{2} \) and \( \frac{2}{2} \) terms are negligible.

As a general rule: a structure with \( k \) nested counting loops — loops where the counter is just incremented (or decremented) by 1 — has performance \( O(N^k) \) if the number of times each loop is iterated depends only on the problem size. A growth rate of \( O(N^k) \) is called polynomial.

In section 4.7 we will show that simple divide-and-conquer algorithms like binary search grow as \( \log N \).
Chapter 4

Sorting and Searching

4.1 Bubble-Sort

For the sake of the description, let us visualise the array as stored as follows:

\[ a[0] \ a[1] \ a[2] \quad \ldots \quad a[size-2] \ a[size-1] \]

Bubble sort has two loops. The outer loop iterates down from the overall top of the array. Conceptually, there are two sub-arrays:

1. \( a[size - 1] \) down to \( a[top + 1] \) which is in a finished state;
2. \( a[top] \) down to \( a[0] \) which is unsorted – though maybe in a better state than when we started.

\[
\begin{align*}
\text{top for outer loop iteration 2} \\
| & \quad \text{top for outer loop iteration 1} \\
| & \\
a[0] \ a[1] \ a[2] \quad \ldots \quad a[size-2] \ a[size-1]
\end{align*}
\]

The purpose of the inner loop is to place the maximum value of the unsorted sub-array in its correct position. It does this by pairwise comparison/swap, i.e. the maximum \textit{bubbles up} to the top. Here is the code.
void bubbleSort(vector<int>& b) {
    uint len = b.size(), ccount = 0, top, i;
    for (top = len-1; top > 0; top--) {
        for (i = 0; i < top; i++) {
            ccount++;
            if (b[i+1] < b[i]) {
                xchg(b, i+1, i);
            }
        }
    }
}

void xchg(vector<int>& c, uint first, uint second) {
    int hold = c[first];
    c[first] = c[second];
    c[second] = hold;
}

Here is the result of each outer loop – for the input array shown at the beginning. This shows that after outer iteration \( m \), the top \( m \) data are in correct (final) order. But note that this printout does not show the bubbling-up process that takes place in the inner loop.

3 10 98 36 37 2 71 61 initial array
3 10 36 37 2 71 61 98
3 10 36 2 37 61 71 98
3 10 2 36 37 61 71 98
3 2 10 36 37 61 71 98
2 3 10 36 37 61 71 98
2 3 10 36 37 61 71 98
2 3 10 36 37 61 71 98

Count of comparisons 28

4.1.1 Running Time

A common way of analysing the running time performance of sorting algorithms is in terms of the number of comparisons required. If the length of the array is \( n \), the outer loop iterates \( n-1 \), \( n-2 \), \ldots, 2, 1 and each corresponding inner loop does \( n-1 \), \( n-2 \), \ldots, 2, 1 comparisons (0...\( n-1 \) for the first, 0 ...\( n-2 \) for the second etc. .... Therefore the total number of comparisons is \((n-1) + (n-2) + \ldots + 2 + 1\), which is equal to \(\frac{(n-1)n}{2}\). Check. In the table above, \( n = 8 \) and \(\frac{(n-1)n}{2} = \frac{7\times8}{2} = 28\).

Exercise. Use sorts.cpp to verify the number of comparisons; for \( n = 8 \), \(\frac{(n-1)n}{2} = 28\), for \( n = 16, 120 \), etc.
Running Time – Measured  The following is an empirical verification that bubble sort is $O(n^2)$. Once the array size is significant, we see that the time increases roughly by 4 for each doubling of $n$, i.e. $O(n^2)$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>0.10</td>
</tr>
<tr>
<td>512</td>
<td>0.38</td>
</tr>
<tr>
<td>1024</td>
<td>1.54</td>
</tr>
<tr>
<td>2048</td>
<td>6.12</td>
</tr>
</tbody>
</table>
| 4096   | 24.56
| 8192   | 98.56 |

Ex. Use sorts.cpp to produce a similar table on your machine, or a laboratory machine.

Ex. Modify sorts.cpp to produce a comparison counts for three cases:

- random data — as before;
- data already sorted – use 

```cpp
sort(a.begin(), a.end());
```
- data in reverse order, use 

```cpp
sort(a.begin(), a.end());
reverse(a.begin(), a.end());
```

Ex. Bubble sort sometimes does unneeded work; if the array is already sorted, there is no need to continue. If there has been no swaps in the previous loop, that indicates that the sort is finished. Modify sorts.cpp to take advantage of this situation. Ans.

```cpp
for (top = len-1; top> 0; top-- ){
    boolean swapped = false;
    for (i = 0; i< top ; i++ ){
        ccount++;
        if(b[i+1]< b[i]){
            swapped = true;
            xchg(b, i+1, i);
        }
        if(!swapped) break;
    }
}
4.2 Selection Sort

In bubble-sort, the only purpose of the exchanging (swapping — xchg) in the (lower) unsorted sub-array is to find the maximum value in it; consequently, we may avoid some swaps by simply finding the maximum, but with no reduction in comparisons.

```c
void selectSort(vector<int> & b) {
    uint top, j, maxPos, len= b.size(), ccount= 0;
    for (top= len-1; top > 0; top--) {
        // find maximum element in the index range 0 to top-1
        maxPos = 0;
        for (j = 1; j < top; j++) {
            ccount++;
            if (b[maxPos] < b[j]) maxPos = j;
        }
        xchg(b, top, maxPos);
    }
    cout << "Count of comparisons " << ccount << endl;
}
```

Here is the result of each outer loop — for the input array shown at the beginning. This shows that, as for bubble sort, after outer iteration \( m \), the top \( m \) data are in correct (final) order.

3 10 98 36 37 2 71 61
3 10 61 36 37 2 71 98
3 10 61 36 37 2 71 98
3 10 2 36 37 61 71 98
3 10 2 36 37 61 71 98
3 10 2 36 37 61 71 98
3 2 10 36 37 61 71 98
2 3 10 36 37 61 71 98

Count of comparisons 28

4.2.1 Running Time

The analysis of number of comparisons is the same as that for bubble sort; i.e. \( \frac{(n-1)n}{2} \).
Running Time – Measured  The following is an empirical verification that selection sort is $O(n^2)$. Once the array size is significant, we see that the time increases roughly by 4 for each doubling of $n$, i.e. $O(n^2)$.

<table>
<thead>
<tr>
<th>n</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>0.05</td>
</tr>
<tr>
<td>512</td>
<td>0.20</td>
</tr>
<tr>
<td>1024</td>
<td>0.76</td>
</tr>
<tr>
<td>2048</td>
<td>3.05</td>
</tr>
<tr>
<td>4096</td>
<td>12.11</td>
</tr>
<tr>
<td>8192</td>
<td>48.35</td>
</tr>
</tbody>
</table>

Ex. Use sorts.cpp to produce a similar table on your machine, or a laboratory machine.

Ex. Modify sorts.cpp to produce a comparison counts and trace of the array for three cases:

- random data, see above;
- data already sorted, see above;
- data in reverse order, see above.

### 4.3 Insertion Sort

Insertion sort works rather like people sort a hand of playing cards. Again, there are two sub-arrays:

1. $a[0]$ up to $a[top - 1]$ which is internally sorted (amongst itself);
2. $a[top]$ up to $a[n-1]$ which is in whatever (unsorted) condition it started in.

The purpose of the inner loop is to take the first element of the unsorted sub-array, and place it in the correct position in the sorted array.

```cpp
void insertSort(vector<int>& b) {
    int top, j, len = b.size(), ccount = 0;
    for (top = 1; top < len; top++) {
        // b[0..top-1] are sorted
        // now put b[top] in correct position
        for (j = top - 1; j >= 0 && b[j+1] < b[j]; --j) {
            xchg(b, j, j+1);
        }
    }
}
```

Here is the result of each outer loop — for the input array shown at the beginning. Notice that for after loop $m$, the bottom $m + 1$ elements are (internally) sorted, and the top $n - m - 1$ are in the same state as when they entered the sort.
4.3.1 Running Time

For the worst case for insertion sort, i.e. where the data are in a maximally unsorted state, the analysis of number of comparisons is the same as that for bubble sort; i.e. \(1 + 2 + \ldots + (n - 2) + (n - 1) = \frac{(n-1)n}{2}\). It can also be shown that even in the average case, the running time is still \(O(n^2)\).

**Ex.** Use sorts.cpp to compute the number of comparisons for \(n = 8, n = 16, n = 32\), and for:

- random data, see above;
- data already sorted, see above
- data in reverse order.

**Measured Running Time** The following is an empirical verification that insertion sort is \(O(n^2)\).

<table>
<thead>
<tr>
<th>n</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>0.08</td>
</tr>
<tr>
<td>512</td>
<td>0.27</td>
</tr>
<tr>
<td>1024</td>
<td>1.13</td>
</tr>
<tr>
<td>2048</td>
<td>4.49</td>
</tr>
<tr>
<td>4096</td>
<td>17.99</td>
</tr>
<tr>
<td>8192</td>
<td>71.82</td>
</tr>
</tbody>
</table>

**Ex.** Use sorts.cpp to produce a similar table on your machine, or a laboratory machine.

**Ex.** Modify sorts.cpp to produce a similar table for three cases:

- random data;
- data already sorted, see above;
- data in reverse order, see above.
4.4 Merge Sort

Merge sort is an example of a sort that take $O(n \log n)$ running time. It is also interesting because of the recursive implementation that we see below.

Merge-sort does array sorting by divide-and-conquer. The algorithm proceeds as follows (recursively):

- If the array is of length 1, the answer is the single element;
- Otherwise, divide the array in two, and sort those;
- Combine the results obtained from the two smaller arrays by merging, i.e. an interleaving the two arrays.

Note that merge-sort needs to create a workspace array the length of the array to be sorted.

```cpp
void merge(vector<int>& a, vector<int>& work, uint l, uint m, uint r) {
    uint i, j, k;
    for (i = m + 1; i > l; i--) work[i - 1] = a[i - 1];
    for (j = m; j < r; j++) work[r + m - j] = a[j + 1];
    for (k = l; k <= r; k++) {
        if (work[i] < work[j]) a[k] = work[i++];
        else a[k] = work[j--];
    }
}

void recMergeSort(vector<int>& a, vector<int>& work, uint l, uint r) {
    if (l == r) return;
    else {
        uint m = (l + r) / 2;
        recMergeSort(a, work, l, m);
        recMergeSort(a, work, m + 1, r);
        merge(a, work, l, m, r);
    }
}

void mergeSort(vector<int>& b) {
    uint len = b.size();
    vector<int> work(len);
    recMergeSort(b, work, 0, len - 1);
}
```

4.4.1 Running Time

The running time for merge-sort is $O(n \log n)$.
**Measured Running Time**  The following is an empirical verification that merge sort is $O(n \log n)$.

If the algorithm is $O(n \log n)$ then the increase factor is

$$2 \times (\log n / \log n - 1)$$

<table>
<thead>
<tr>
<th>n</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>0.01</td>
</tr>
<tr>
<td>512</td>
<td>0.02</td>
</tr>
<tr>
<td>1024</td>
<td>0.05</td>
</tr>
<tr>
<td>2048</td>
<td>0.10</td>
</tr>
<tr>
<td>4096</td>
<td>0.22</td>
</tr>
<tr>
<td>8192</td>
<td>0.45</td>
</tr>
<tr>
<td>16384</td>
<td>1.00</td>
</tr>
</tbody>
</table>

$$2 \times 12/11 = 2.2$$

$$2 \times 13/12 = \text{approx} 2.15$$

$$2 \times 14/13 = \text{approx} 2.15$$

### 4.5 Quicksort

Just for completeness, we show, without any analysis, one of the quickest sorting algorithms available, Quicksort — discovered by C.A.R. Hoare in 1962; Hoare was Professor of Computer Science in Queen’s University Belfast from 1968 to 1977.

```c
int partition(vector<int>& a, int left, int r){
    int i = left-1, j=r;
    int v = a[r], temp;
    while(true){
        while(a[++i]<v);
        while(v < a[--j])if(j==left)break;
        if(i >= j)break;
        temp = a[j]; a[j] = a[i]; a[i] = temp;
    }
    temp = a[r]; a[r] = a[i]; a[i] = temp;
    return i;
}

void quickSort(vector<int>& a, int left, int right){
    int i;
    if(right <= left)return;
    i = partition(a, left, right);
    quickSort(a, left, i-1);
    quickSort(a, i+1, right);
}
```

### 4.5.1 Running Time

The following is an empirical verification that Quicksort is $O(n \log n)$. Notice that in $n \log n$ the log $n$ increases by a very small amount for each $n$ step.
n  time
256, 0.00
512, 0.01
1024, 0.02
2048, 0.04
4096, 0.08
8192, 0.21
16384, 0.45

4.5.2 Standard Library Sort

I’m pretty sure the standard library sort uses quicksort; here are the times for it:

n  time
256, 0.00
512, 0.00
1024, 0.00
2048, 0.01
4096, 0.04
8192, 0.06
16384, 0.12

4.6 Comparison: Bubble, Selection, Insertion and Merge and Quick Sorts

The tables given above show that in order of decreasing running time, we have:

1. Bubble sort; for $n = 4096$, it takes 24.56 seconds;
2. Insertion sort; for $n = 4096$, it takes 12.11 seconds;
3. Selection sort; for $n = 4096$, it takes 17.99 seconds;
4. Merge sort; for $n = 4096$, it takes 0.22 seconds.
5. Our Quicksort; for $n = 4096$, it takes 0.08 seconds.
6. Std. Quicksort; for $n = 4096$, it takes 0.04 seconds.

We still say that each of the slowest three take $O(n^2)$ running time; it is only the the constant of proportionality is different, but that does not matter to in big-Oh terms. $t = a + cf(n)$, where $a, c$ are constants and $f(n) = n^2$ for bubble, insertion, and selection sorts, and $f(n) = n \log n$ for merge sort and quicksort.

We note, however, that merge sort needs to create a workspace array the length of the array to be sorted. In cases where memory is in short supply, that could be make it unappealing.

Ex. You have to sort an array of $n = 16$ million (take it as $2^{24}$). At $n = 4096$,
(i) Bubble sort takes 20 seconds;
(ii) Quicksort takes 0.04 seconds.

Do a rough estimate of how long each will take for \( n = 16 \text{ million} \).

4.7 Searching

4.7.1 Sequential Search

How would you determine whether the key (value) 55 is in \texttt{vector<int> a}? Let’s say the vector is of size \( N \). The best we can do is iterate through the vector doing comparisons:

```cpp
int lSearch(int key, vector<int>& a) {
    int n = a.size();
    if (n == 0) return -1;
    for (int i = 0; i < n; ++i) {
        if (key == a[i]) return i;
    }
    return -1; // fell off end before found
}
```

Question. What is the average number of comparisons for (i) a key which is in the vector? Answer. \( \frac{N}{2} \); (ii) one which is not in the vector? Answer. \( N \). \texttt{lSearch} is an example of a sequential or linear search. Linear search is fine for small array sizes, but we often need better when the size is large.

For example, think how you use a telephone directory, looking for, say, \textit{J. Murphy}. You would never start at page 1 and examine every entry until you get to \textit{Murphy}; that is a linear search and would take forever.

You take advantage of the fact that the directory is sorted, and you jump straight to the middle; if you have gone too far, you hop back a little, etc. This takes almost no time at all and is strongly related to the very efficient binary search that we cover later.

Using the notation of Chapter 3, we say that linear search grows as \( O(N) \).
4.7.2 Binary Search

If the array is already sorted, we may employ a binary search.

```cpp
int bSearch1(int key, vector<int>& a, uint left, uint right){
    int mid = (left + right)/2;
    if(left > right) return -1; // not found
    else if(key == a[mid]) return mid;
    else if(key > a[mid]) return bSearch1(key, a, mid+1, right);
    else return bSearch1(key, a, left, mid-1);
}

int bSearch(int key, vector<int>& a){
    uint n = a.size();
    if(n == 0) return -1;
    else return bSearch1(key, a, 0, n-1);
}
```

In this case the dependence of time on \( N \) is said to be logarithmic; if you plot time versus \( N \) on a graph you get a curve which starts off almost as a straight line but gradually curves towards the horizontal — meaning that the effect of very large \( N \) is greatly diminished. This is the same effect as finding that the time to search for a name in a telephone directory is almost independent of the size of the directory.

4.7.3 Logarithms

Why is binary search, in big-Oh terms, \( \log(n) \)? For big-Oh, see Chapter 3. If we take the definition of \( \log(n) \) as the inverse of power — \( \log_2(2^x) = x \), or log of a number is the power to which the base must be raised in order to equal that number, we are not much further ahead.

However, sticking with \( \log_2 \) for a moment, let’s look at powers of 2: \( 2^0 = 1 \), \( 2^1 = 2 \), \( 2^2 = 4 \), \( 2^3 = 8 \), \( 2^4 = 16 \), ... Then, of course, we have \( \log_2(16) = 4 \).

What happens if we divide 16 by two, and keep dividing the result of such divisions until we get to 1?

\[
\frac{16}{2} = 8, \quad \frac{8}{2} = 4, \quad \frac{4}{2} = 2, \quad \frac{2}{2} = 1. \]

That is, it took four (4) steps. This computation hints at a definition of logarithm for integer values — the count the number of successive divisions to get to 1 as the opposite of raising to a power. This works for any integer, e.g. 10. If, in the divide-and-conquer strategy in bSearch1, we divided the array into 10 equal chunks — a denary or decimal search — then the dependency would be \( O(\log_{10}(n)) \).
**Logarithm to base** $b$ **in terms of** $\log_{10}$

If you want to take logs-to-base-two, and your calculator does only logs-to-base-10, you need the following:

$$\log_b(x) = \frac{\log_{10}(x)}{\log_{10}(b)},$$

therefore

$$\log_{10}(2) = 0.3010,$$

$$\log_2(x) = \frac{\log_{10}(x)}{0.3010} = 3.322 \times \log_{10}(x).$$

When dealing with logarithms in the context of algorithms, we normally require $\lceil \log(n) \rceil$ – the so-called ceiling of $\log(n)$; in other words, the fractional logarithm, rounded-up. Why? Recall our analysis of binary search for $n = 16$ – we required 4 successive divisions. What if we had $n = 17$, or $n = 21$, right up to $n = 32$? We would have required one more sub-division, i.e. 5; that’s why we need to round up.

### 4.7.4 Is Binary Search Really Faster than Linear?

We can ask the question *Is Binary Search Really Faster than Linear?*. Figure 4.1 shows the code we use to evaluate sorting and searching algorithms.

We know that $\text{lSearch}$ grows as $O(N)$. But $\text{lSearch}$ does not require us to sort the data.

Let’s say $\text{sort}$ takes $O(N \log N)$; so our first $\text{bSearch}$ takes $O(N \log N) + O(\log N)$, which is already worse than $\text{lSearch}$’s $O(N)$.

So which is better?

The answer is that if you are going to do a great many searches on the same data set, then it makes sense to sort (once) at the beginning and take advantage of the fast binary search. But if you know that you will not do any more than $\log N$ searches there is no point in sorting, simply use linear search on the unsorted array.

We could ensure that the way we create the array in the first place ensures that it is sorted; i.e. develop a method $\text{insertInOrder(value)}$; however, from Chapter 2, we know that inserting at a random index takes $O(N)$ (because the data above the new value need to be shifted up one place). So if there are $N$ data to be inserted, this creation procedure takes $O(N^2)$, so that’s no good, you’ve lost out before you even start.

There is a data structure however in which it is easy ($O(\log N)$) to insert in order, that is a binary search tree; and searching a binary search tree also takes $O(\log N)$. Therefore, if you know that you will search a collection a lot, a binary search tree is your choice. Binary search trees are covered in Chapter 7. You will not find the word *tree* in the standard library, but `std::set` and `std::map` both use variations on binary search tree.
int main(int argc, char** argv)
{
    timer timer;
    RNG rng(13131131);

    uint step= 2, min= 256, max= 16384, nReps = 100000;
    //min= 8, max= 8, nReps = 1;
    cout<< " n time "<< endl;
    for(uint n = min; n<= max ; n*=step){
        vector<int> a(n);
        timer.restart(0);

        for(uint i = 0; i < nReps; ++i){
            timer.stop();
            for(uint j = 0; j < n; ++j)a[j] = rng.rand<int32_t>()%100;
            int keyIndex = rng.rand<int32_t>();
            keyIndex = keyIndex%n;
            int key = a[keyIndex];
            key = 200;

            //sort(a.begin(), a.end());
            //reverse(a.begin(), a.end());
            //timer.start();
            //bubbleSort(a);
            //selectSort(a);
            //insertSort(a);
            //mergeSort(a);
            //quickSort(a, 0, n-1);
            sort(a.begin(), a.end());
            timer.start();
            int indx = lSearch(key, a);
            //int indx = bSearch(key, a);
            //copy(a.begin(), a.end(), ostream_iterator<int>(cout, " "));
            //cout<< endl;
            //cout<< "key, index "<< key<< " , "<< indx<< endl;
        }
        cout<<" "<< n<< " , "<< timer<< endl;
    }
}

Figure 4.1: Code to test sorting and searching
Chapter 5

Linked Lists

5.1 Introduction

The two most common form of sequence data structures are (a) contiguous array-based as covered in Chapter 2 and (b) linked-list-based that we cover in this Chapter. In what follows, when comparing them, we’ll use the term array for the former and linked-list for the latter. Apart from sequence data structures, we will encounter other forms of collection, e.g. trees and graphs; some of these use linked representations, some use arrays, and there are other implementations.

std::vector (and likewise Array) that we covered in Chapter 2 do a fine job for many applications. However, they run into performance difficulties in two major circumstances: (i) you need to insert one or more elements into the array at other than the back; in fact std::vector has no push_front function because it is reckoned that to use push_front on an array would be rather silly; (ii) you get it wrong when you declare the initial size of the array (or when you use reserve).

These difficulties can be seen by examining Figure 5.1.

First, push_back works fine until we run out of space, but then we must reserve more memory (an new array), and, (not shown) copy all data from the old array to the new array; that makes push_back an $O(N)$ operation in the worst case, and if, unlike the example, we reserve only what extra space that is immediately needed, then $N$ push_backs will grow as $O(N^2)$.

Next, if we look at insert, we see that the first thing that has to be done is to copy all data that are above the insert position, $n$ memory positions up the array to make space for the $n$ inserts. This is an $O(N)$ operation, so that if we want to insert all $N$ array elements in this manner, constructing the array will be $O(N^2)$.

A linked-list operates by treating every element in a collection as a semi-independent entity but linked to other elements in the collection so that the sequence can be maintained. Using this representation, most insert and delete operations take constant time, i.e. $O(1)$. The one downside is that accessing the $i$-th item in the list (sequence) takes $O(N)$ — sequential access, whilst an array allows random access and the same can be done in $O(1)$.

As we have noted earlier, the purpose of developing our own list classes, just as for the array class in Chapter 2, is mostly to develop some sympathy with how and why they work. When it comes to application development, we will almost always use STL collections and algorithms.
template <class T>
void Array<T>::insert(Iterator pos, uint n, const T& e) {
    uint sz = sz_ + n;
    assert(cap_ >= sz);
    Iterator itrd = dat_ + sz - 1; // destination
    Iterator itrs = dat_ + sz_ - 1; // source
    for (; itrs != pos; --itrs, --itrd) *(dat_ + itrd) = *(dat_ + itrs);
    for (itrd = pos; itrd != pos + n; ++itrd) *(dat_ + itrd) = e;
    sz_ = sz;
}

template <class T>
void Array<T>::push_back(const T& val) {
    if (!(cap_ > sz_)) reserve(2*cap_);
    dat_[sz_] = val;
    ++sz_;
}

Figure 5.1: Extracts from Array.h showing drawback of Array and std::vector
5.2 A Singly Linked List

Here we develop the simplest form of linked list — a singly linked list. This works fine for most list applications, but suffers from the disadvantage that inserting and deleting at the back (push_back, pop_back) take $O(N)$. Later, we will see how a doubly linked list can remedy this particular drawback.

5.2.1 Class Declaration

The declaration of the singly linked SList class is shown in Figure 5.2.
```cpp
#ifndef LISTTH
#define LISTTH

#include <cassert>
#include <iostream>

template <class T>
class List;

template <class T>
std::ostream& operator<<(std::ostream& os, const List<T>& l);

template <class T> class Link;

template <class T> class List{
    friend std::ostream& operator<<(T& os, const List<T>& l);
public:
    List();
    List(const List<T> & other);
~List();
    List & operator = (const List<T> & rhs);
    T front() const;
    void push_front(T e);
    void pop_front();
    T back() const;
    void push_back(T e);
    void pop_back();
    void clear(); // empty the list and delete all elements in it
    bool empty() const;
    int size() const;
private:
    void copy(const List<T> & other); // local utility only, users use =
    Link<T>* first_;
};

template <class T> class Link{
    friend class List<T>;
    friend std::ostream& operator<<(T& os, const List<T>& l);
private:
    Link(T e, Link *next=0);
    T elem_; 
    Link<T>* next_; 
};

Figure 5.2: Declaration of Singly Linked List.
```
5.2.2 Dissection of List

[You know all this already, but it may be worth repeating.]

1. The type parameter $T$. As already mentioned, $\text{List}$ is a parameterised class – the type / class of its element is given as a template class parameter. Just as in declaration of a function with parameters (variables — formal parameters), we must announce this fact, and give the (as yet unknown) parameter an identifier. This is done as follows:

```cpp
template <class T> class List
```

Here we are saying: $\text{List}$ uses an as yet unspecified type / class $T$.

2. When we want to define an actual $\text{List}$ in a program, we do so as, e.g.:

```cpp
List<int> s;
List<float> t;
List<string> x;
```

Thus, as with (parameterised) functions, we can 'use' the abstraction with any number of (different) parameters.

3. Likewise, if we ever need to declare a $\text{List}$ object as a parameter, we use the form:

```cpp
List(const List<T> & other);
```

i.e. `other` is a reference to a $\text{List}$ object of class or type $T$ elements.

4. Instantiation. Just as variables and objects are instantiated (created), so are template classes. In the case of template classes and functions, this is done at compile time.

5. In GNU C++, the $\text{List<int>}$ class, for example, is instantiated, at compile time, by a form of macro-expansion. This is the reason that function implementations must also be in the .h file.

6. The only data member for a $\text{List}$ is $\text{Link<T>* first_}$, a pointer to a $\text{Link}$.

7. When first constructed a $\text{List}$ is empty, in which case $\text{first_ == NULL}$.

8. $\text{List}$ has the Big-Three: copy constructor, assignment operator, and destructor.

9. Ordinarily, e.g. in the copy constructor and the assignment operator, we pass Lists by reference, e.g.

```cpp
List<T>& List<T>::operator = (const List<T>& rhs)
```

This is because we don’t want to make unnecessary copies.

Figure 5.3 gives a brief indication of the operation of $\text{List}$. 
Figure 5.3: Operation of a singly linked list.
5.2.3 Class Implementation

The implementations of selected List class functions are shown in Figures 5.4 to 5.6. Most of these are straightforward, but it will be useful to ensure that we discuss them properly in lectures; diagrams will do a lot to aid your understanding. You should also read appropriate sections of (Penton 2003) and execute his programs which have interactive graphics representations of singly linked and doubly linked lists.
template <class T>
Link<T>::Link(T e, Link<T> *next) {
    elem_ = e;
    next_ = next;
}

template <class T>
List<T>::List(){
    first_ = 0;
}

template <class T>
void List<T>::copy(const List<T> & other){
    if(other.empty())first_ = 0;
    else{
        Link<T> *pp = other.first_; //cursor to other
        Link<T> *pt = new Link<T>(pp->elem_, 0);
        first_ = pt;
        while(pp->next_ != 0){
            pp = pp->next_;;
            pt->next_ = new Link<T>(pp->elem_, 0);
            pt = pt->next_;;
        }
    }
}

template <class T>
List<T>::List(const List<T>& other){
    copy(other);
}

template <class T>
List<T> & List<T>::operator = (const List<T>& rhs){
    if(this != &rhs){ //beware of listA=listA;
        clear();
    }
    copy(rhs);
    return *this;
}

template <class T>
List<T>::~List(){ clear();
}

template <class T>
void List<T>::push_front(T e){
    Link<T> *pt = new Link<T>(e, first_);
    assert(pt != 0);
    first_ = pt;
} // continued ...
template <class T>
T List<T>::front() const{
    assert(!empty());
    return first_->elem_;
}

template <class T>
void List<T>::pop_front(){
    Link<T>* pt = first_;  
    first_ = pt->next_;    
    delete pt; pt = 0;
}

template <class T>
void List<T>::push_back(T e){
    //std::cout<< "push_back"<< std::endl;
    if(empty()){
        first_ = new Link<T>(e, 0);
        assert(first_ != 0);
    }  
    else{  
        Link<T>* pp = first_;  
        // walk to the back  
        while(pp->next_ != 0)pp = pp->next_;  
        // and add a new Link with e in it and next = null  
        pp->next_ = new Link<T>(e, 0);
        assert(pp->next_ != 0);
    }
}

template <class T>
void List<T>::pop_back(){
    //std::cout<< "pop_back"<< std::endl;
    assert(!empty());
    if(first_->next_ == 0){  /*kludge for one element */
        delete first_;  
        first_ = 0;  
        return;
    }
    Link<T>* pp(first_), *prev(first_);
    // walk to the back  
    while(pp->next_ != 0){
        prev = pp; pp = pp->next_;  
    }
    // delete the last Link and set prev->next = null  
    delete pp;  
    pp = 0;
    prev->next_ = 0;
}  // continued ...

Figure 5.5: Implementation code for Singly Linked List, part 2.
template <class T>
T List<T>::back() const{
//std::cout<< "back"<< std::endl;
assert(!empty());
Link<T>* pp (first_);
// walk to the back
while(pp->next_ != 0) pp = pp->next_;  
return pp->elem_;  
}

template <class T>
void List<T>::clear(){
Link<T> *next,*pp(first_);
while(pp != 0){
    next = pp->next_;  
    pp->next_ = 0; // why did Budd include this?
    delete pp;  
    pp = next;  
}
first_ = 0;
}

template <class T>
bool List<T>::empty() const{
    return (first_ == 0);
}

template <class T>
int List<T>::size() const{
    int i = 0;
    Link<T> *pt = first_;  
    while(pt != 0){
        pt = pt->next_;  
        ++i;  
    }
    return i;
}

template <class T>
std::ostream& operator<< (std::ostream& os, const List<T>& lst){
    os<<"f[ ";
    Link<T> *pp = lst.first_;  //cursor to lst
    while(pp != 0){
        if(pp != lst.first_)os<<", ";
    os<< pp->elem_;  
        pp = pp->next_;  
    }
    os<< " ]b"<<std::endl;
    return os;
}
5.3 A simple List client program

As we have seen, there is little to templates.

The program ListT1.cpp in Figures 5.7 and 5.8 demonstrate the use of the List class.

Figure 5.9 shows the output of ListT1.cpp. Please note that if you make minor modifications to the beginning of ListT1.cpp and change occurrences of List to list (std::list), the program performs the same. Although std::list is a doubly-linked-list, the interface functions hide that fact.

In the next section, we develop a doubly linked list.
/* --- ListT1.cpp ---------------------------------------------
 j.g.c. 31/12/96//j.g.c. 1/1/97, 5/1/97, 2007-12-28, 2008-01-18
 changed j.g.c. 2007-12-30 to use new SList.h (std::list compatible)
 -------------------------------------------------------- */
#include "SList.h"
#include <string>
#include <list>
#include <iostream>
using namespace std;

typedef List<double> ListD;
typedef List<int> ListI;
typedef List<string> ListS;

int main(){
    ListD x;
    x.push_front(4.4); x.push_front(3.3); x.push_front(2.2); x.push_front(1.1);
    ListD y(x);
    ListD z = x; //NB. equiv. to ListD z(x); see prev. line
    cout<< "x.front = "<< x.front()<< endl;

    //note that the following destroys the List
    cout<< "List x ="<<endl;
    cout<< "x.size() ="<< x.size()<< endl;
    while(!x.empty()){ 
        cout<< x.front()<< endl;
        x.pop_front();
    }
    cout<< "x.size() now = "<< x.size()<< endl;

    cout<< "List y ="<<endl;
    cout<< y<< endl;

    cout<< "List z ="<<endl;
    cout<< z<< endl;
    // continued ...
ListD v; v = y;
v.pop_front();
cout << "List v (v = y; v.pop_front();) =" << endl;
cout << v << endl;

ListI li; li.push_front(3); li.push_front(2); li.push_front(1);
cout << "List li via operator <<=" << endl;
cout << li << endl;
li.push_back(22); li.push_back(33);

cout << "li.push_back(22), li.push_back(33)" << endl;
cout << li << endl;

cout << "back(), pop.back()" << endl;
while (!li.empty()) {
    cout << li.back() << endl;
    li.pop_back();
}

ListS ls;
ls.push_front("abcd");
ls.push_front("cdefgh");
ls.push_back("back");
cout << ls << endl;

return 0;
}

Figure 5.8: ListT1.cpp, part 2
List x =
x.size() = 4
1.1
2.2
3.3
4.4
x.size() now = 0
List y =
f[ 1.1, 2.2, 3.3, 4.4 ]b

List z =
f[ 1.1, 2.2, 3.3, 4.4 ]b

List v (v = y; v.pop_front();) =
f[ 2.2, 3.3, 4.4 ]b

List li via operator <<
f[ 1, 2, 3 ]b
li.push_back(22), li.push_back(33)
f[ 1, 2, 3, 22, 33 ]b
back(), pop.back()
33
22
3
2
1
f[ cdefgh, abcd, back ]b

Figure 5.9: Output of ListT1.cpp.
5.4 Doubly Linked List

In this section, we develop a *doubly linked list*. The declaration is shown in Figures 5.10, 5.11 (Link) and 5.12 (ListIterator).
```cpp
#ifndef LISTTH
#define LISTTH

#include <cassert>
#include <iostream>

template <class T>
class List;

template <class T>
std::ostream& operator<<(std::ostream& os, const List<T>& l);

template <class T>
class Link;

template <class T>
class ListIterator;

template <class T> class List{
  friend std::ostream& operator<<(std::ostream& os, const List<T>& l);
public:
  typedef ListIterator<T> Iterator;
  List() : first_(0), last_(0) {};
  List(const List<T>& other);
  ~List();
  List & operator = (const List<T>& rhs);
  T& front() const;
  void push_front(const T& e);
  void pop_front();
  T& back() const;
  void push_back(const T& e);
  void pop_back();
  void clear(); // empty the list and delete all elements in it
  bool empty() const;
  int size() const;
  Iterator begin();
  Iterator end();
  Iterator insert(Iterator& itr, const T& val);
  void insert(Iterator& itr, int n, const T& val);
  void erase(Iterator& itr);
  void erase(Iterator& start, Iterator& stop);
private:
  void copy(const List<T>& other); // private utility only, users use =
  Link<T>* first_;          
  Link<T>* last_;           
};

Figure 5.10: Doubly linked list declaration, part 1
```
template <class T> class Link{
    friend class List<T>;
    friend class ListIterator<T>;
    friend std::ostream& operator<< <T>(std::ostream& os, const List<T>& l);
private:
    Link(const T& e) : elem_(e), next_(0), prev_(0){}
    T elem_;  
    Link<T>* next_; 
    Link<T>* prev_; 
};

Figure 5.11: Doubly linked list declaration, part 2, the Link
template <class T> class ListIterator {
    friend class List<T>;
    typedef ListIterator<T> Iterator;

public:
    ListIterator(List<T>* list = 0, Link<T>* cLink = 0) :
        list_(list), cLink_(cLink) {}

    T& operator *(){
        return cLink_->elem_;}

    bool operator == (Iterator rhs) {
        return cLink_ == rhs.cLink_;}

    bool operator != (Iterator rhs) {
        return cLink_ != rhs.cLink_;}

    Iterator& operator ++ (int){
        cLink_ = cLink_->next_;  
        return *this;            }

    Iterator operator ++ ()
        Iterator& operator -- (int){
        cLink_ = cLink_->prev_;  
        return *this;            }

    Iterator operator -- ()
}

private:
    List<T>* list_;  
    Link<T>* cLink_;};

Figure 5.12: Doubly linked list declaration, part 3, the ListIterator
5.4.1 Brief Discussion of the Doubly Linked List

The doubly linked list in Figures 5.10 to 5.12 is very similar to the singly linked list described earlier in the chapter. The chief differences are:

1. There are now two List data members (the singly linked list had just a pointer to the front; now we have pointers to front and back):

   ```cpp
   Link<T>* first_;
   Link<T>* last_;  
   ```

   This means that when we want to push_back or pop_back, we can go directly there, via last_, rather than having to sequentially walk there as in the singly linked list example.

2. Link now has three data members, first the element; then, as before, one pointing to the next link, and now a new pointer pointing to the previous link:

   ```cpp
   T elem_;
   Link<T>* next_;
   Link<T>* prev_;  
   ```

   In the singly linked list, we could traverse the list (iterate) only in the front towards back direction via next_; now we can traverse in both directions, back towards front, using the prev_ pointer.

3. The other major difference is that we have equipped the List with an iterator. This iterator has the same effect as the Array iterator in Chapter 2, i.e. it looks the same to client programs, but it is slightly more complicated to implement.

4. ListIterator has two data members:

   ```cpp
   List<T>* list_;
   Link<T>* cLink_;  
   ```

   a pointer to the List itself, and a pointer one of the Links.
5.4.2 Simple Test Program, ListT1.cpp

Figures 5.13 and 5.14 show a simple test program. The only difference between this program and the one above in Figures 5.7 and 5.8 is that we have added code to exercise the additional functions in the doubly linked list, and the iterator.

As pointed out before, we note that this doubly linked list, the singly linked list above, and std::list appear identical in client programs; the only differences (and there should be none) is that in the lists here, we have chosen to implement only a subset of the functions of std::list and in the singly linked list, the subset is even smaller. As we keep saying, the objectives of the list classes here is not to replace std::list, but to get some feeling how std::list might be implemented.

The output of ListT1.cpp is shown in Figure 5.15
```cpp
#include "List.h"
#include <string>
#include <iostream>
using namespace std;

typedef List<double> ListD;
typedef List<int> ListI;
typedef List<string> ListS;

int main(){
    ListD x;
    x.push_front(4.4); x.push_front(3.3); x.push_front(2.2); x.push_front(1.1);

    ListD y(x);
    ListD z = x; //NB. equiv. to ListD z(x); see prev. line

cout<< "x.front = "<< x.front()<< endl;

    //note that the following destroys the list
    cout<< "List x ="<<endl;
    cout<< "x.size() ="<< x.size()<< endl;
    while(!x.empty()){
        cout<< x.front()<< endl;
        x.pop_front();
    }
    cout<< "x.size() now = "<< x.size()<< endl;

cout<< "List y ="<<endl;
cout<< y<< endl;

cout<< "List z ="<<endl;
cout<< z<< endl;

    ListD v;
    v = y;
    v.pop_front();
    cout<< "List v (v = y; v.pop_front();) ="<<endl;
cout<< v<< endl;  // continued ...
```

Figure 5.13: Simple Test Program for Doubly Linked List, ListT1.cpp, part 1
ListI li; li.push_front(3); li.push_front(2); li.push_front(1);
cout << "List li via operator " << endl;
cout << li << endl;

li.push_back(22);
li.push_back(33);

cout << "li.push_back(22), li.push_back(33)" << endl;
cout << li << endl;

cout << "back(), pop_back()" << endl;
while(!li.empty()){
    cout << li.back() << endl;
    li.pop_back();
}

ListS ls;
ls.push_front("abcd");
ls.push_front("cdefgh");
ls.push_back("back");
cout << ls << endl;

ListI c5;
for(uint i = 0; i < 5; ++i){
    c5.push_back(i);
    cout << "c5.push_back(i = " << i << ") : " << c5;
}

cout << "using Iterator" << endl;
ListI::Iterator itr = c5.begin();
ListI::Iterator itrb = c5.begin();
ListI::Iterator itre = c5.end();

if(itr == itrb) cout << "itr == itrb" << endl;
else cout << "itr != itrb" << endl;

if(itr != itrb) cout << "itr != itrb" << endl;
else cout << "itr == itrb" << endl;

ListI::Iterator it;
for(it = c5.begin(); it != c5.end(); ++it){
    cout << *it << ' ';
}

cout << "ListI::Iterator itr2 = c5.begin(), ++, ++ " << endl;
cout << "c5.insert(itr2, 5, 33) " << endl;
ListI::Iterator itr2 = c5.begin();
itr2++; itr2++;
c5.insert(itr2, 5, 33);
cout << c5;
x.front = 1.1
List x =
x.size() = 4
1.1
2.2
3.3
4.4
x.size() now = 0
List y =
f[ 1.1, 2.2, 3.3, 4.4 ]b

List z =
f[ 1.1, 2.2, 3.3, 4.4 ]b

List v (v = y; v.pop_front();) =
f[ 2.2, 3.3, 4.4 ]b

List li via operator <<
f[ 1, 2, 3 ]b
li.push_back(22), li.push_back(33)
f[ 1, 2, 3, 22, 33 ]b

back(), pop.back()
33
22
3
2
1
f[ cdefgh, abcd, back ]b

c5.push_back(i = 0): f[ 0 ]b
c5.push_back(i = 1): f[ 0, 1 ]b
c5.push_back(i = 2): f[ 0, 1, 2 ]b
c5.push_back(i = 3): f[ 0, 1, 2, 3 ]b
c5.push_back(i = 4): f[ 0, 1, 2, 3, 4 ]b
using Iterator
itr == itrb
itr == itrb
0 1 2 3 4 ListI::Iterator itr2 = c5.begin(), ++, ++
c5.insert(itr2, 5, 33)
f[ 0, 1, 33, 33, 33, 33, 33, 33, 33, 2, 3, 4 ]b

Figure 5.15: Output of ListT1.cpp (double linked version)
5.4.3 Doubly Linked List Implementation

Here we give the complete implementation of the doubly linked list. Since the principles involved are the same as for the singly linked list, we provide no discussion. However, it will be worthwhile to spend some time in class on the implementation, especially that of ListIterator.

// -------- ListIterator ---------------------------------
// postfix form of increment
template<class T>
ListIterator<T> ListIterator<T>::operator ++ (){
    // clone, then increment, return clone
    ListIterator<T> clone (list_, cLink_);
    cLink_ = cLink_->next_;
    return clone;
}

// postfix form of decrement
template<class T>
ListIterator<T> ListIterator<T>::operator -- (){
    // clone, then increment, return clone
    ListIterator<T> clone (list_, cLink_);
    cLink_ = cLink_->prev_;
    return clone;
}

Figure 5.16: Doubly Linked List Implementation, part 1, ListIterator
template <class T>
void List<T>::copy(const List<T> & other){
    if(other.empty())first_ = 0;
    else{
        Link<T> *pp = other.first_; //cursor to other
        Link<T> *pt = new Link<T>(pp->elem_);
        first_ = pt;
        while(pp->next_ != 0){
            pp = pp->next_; 
            pt->next_ = new Link<T>(pp->elem_); 
            pt = pt->next_; 
        }
    }
}

template <class T>
List<T>::List(const List<T> & other){
    copy(other);
}

template <class T>
List<T> & List<T>::operator = (const List<T> & rhs){
    if(this != &rhs){ //beware of listA=listA;
        clear();
    }
    copy(rhs);
    return *this;
}

template <class T>
List<T>::~List(){
    clear();
}

Figure 5.17: Doubly Linked List Implementation, part 2
template <class T>
void List<T>::copy(const List<T> & other){
    if(other.empty())first_ = 0;
    else{
        Link<T>* pp = other.first_; //cursor to other
        Link<T>* pt = new Link<T>(pp->elem_);
        first_ = pt;
        while(pp->next_ != 0){
            pp = pp->next_;            
            pt->next_ = new Link<T>(pp->elem_);
            pt = pt->next_;            
        }
    }
}

template <class T>
List<T>::List(const List<T> & other){
    copy(other);
}

template <class T>
List<T> & List<T>::operator = (const List<T> & rhs){
    if(this != &rhs){  //beware of listA=listA;
        clear();
    }
    copy(rhs);
    return *this;
}

template <class T>
List<T>::~List(){
    clear();
}

Figure 5.18: Doubly Linked List Implementation, part 3
template <class T>
void List<T>::push_front(const T& e){
    Link<T>* newLink = new Link<T>(e);
    assert(newLink != 0);
    if(empty()) first_ = last_ = newLink;
    else {
        first_->prev_ = newLink;
        newLink->next_ = first_;  
        first_ = newLink;
    }
}

template <class T>
T& List<T>::front() const{
    assert(!empty());
    return first_->elem_;  
}

template <class T>
void List<T>::pop_front(){
    assert(!empty());
    Link<T>* tmp = first_;  
    first_ = first_->next_;  
    if(first_ != 0) first_->prev_ = 0;
    else last_ = 0;
    delete tmp;
}

Figure 5.19: Doubly Linked List Implementation, part 4
template <class T>
void List<T>::push_back(const T& e){
    Link<T>* newLink = new Link<T>(e); assert(newLink != 0);
    if(empty()) first_ = last_ = newLink;
    else{
        last_->next_ = newLink;
        newLink->prev_ = last_; last_ = newLink;
    }
}

template <class T>
void List<T>::pop_back(){
    assert(!empty());
    Link<T>* tmp = last_; last_ = last_->prev_; if(last_ != 0) last_->next_ = 0;
    else first_ = 0;
    delete tmp;
}

template <class T>
T& List<T>::back() const{
    assert(!empty());
    return last_->elem_;}

template <class T>
void List<T>::clear(){
    Link<T>* next, *first(first_);
    while(first_ != 0){
        next = first_->next_; delete first_; first_ = next;
    }
    first_ = 0;
}

template <class T>
bool List<T>::empty() const{
    return (first_ == 0);
}

template <class T>
int List<T>::size() const{
    int i = 0;
    Link<T>* pt = first_;
    while(pt != 0){
        pt = pt->next_; ++i;
    }
    return i; }

Figure 5.20: Doubly Linked List Implementation, part 5
template <class T>
ListIterator<T> List<T>::begin() {
    return Iterator(this, first_);
}

template <class T>
ListIterator<T> List<T>::end() {
    return ListIterator<T>(this, 0);
}

template <class T>
void List<T>::erase(ListIterator<T>& itr){
    erase(itr, itr);
}

// insert a new element into the middle of a linked list
template <class T>
ListIterator<T> List<T>::insert (ListIterator<T> & itr, const T& value){
    Link<T>* newLink = new Link<T>(value);
    Link<T>* current = itr.cLink_;

    newLink->next_ = current;
    newLink->prev_ = current->prev_;
    current->prev_ = newLink;
    current = newLink->prev_;
    if (current != 0)current->next_ = newLink;

    return ListIterator<T>(this, newLink);
}

// insert n new elements into the middle of a linked list
// note: iterator changed, but I think this is okay as
// iterators are known to be invalidated by insertion
template <class T>
void List<T>::insert (ListIterator<T> & itr, int n, const T& value){
    for(int i = 0; i<n; i++){
        itr = insert(itr, value);
    }
}

Figure 5.21: Doubly Linked List Implementation, part 6
// remove values from the range of elements

```
template <class T>
void List<T>::erase (ListIterator<T> & start, ListIterator<T> & stop)
{
    Link<T> * first = start.cLink_;  
    Link<T> * prev = first->prev_;   
    Link<T> * last = stop.cLink_;    
    if (prev == 0) { // removing initial portion of list
        first_ = last;  
        if (last == 0) last_ = 0;  
        else last->prev_ = 0;  
    }
    else {
        prev->next_ = last;  
        if (last == 0) last_ = prev;  
        else last->prev_ = prev;  
    }

    // now delete the values
    while (start != stop) {  
        ListIterator<T> next = start;  
        ++next;  
        delete start.cLink_;  
        start = next;  
    }
}
```

```
template <class T>
std::ostream& operator<< (std::ostream& os, const List<T>& lst)
{
    os<"f[ ";
    Link<T> * pp = lst.first_; //cursor to lst
    while (pp != 0) {
        if (pp != lst.first_) os<", ";
        os<pp->elem_;  
        pp = pp->next_;  
    }
    os<" ]b"<<std::endl;
    return os;
}
```

Figure 5.22: Doubly Linked List Implementation, part 7
5.5 Arrays versus Linked List, Memory Usage

On a 32-bit machine, int and pointer typed variables normally occupy four 8-bit bytes.

**Ex. 1.** If we have an `Array<int>` such as that in Chapter 2 (and `std::vector` will be the same) whose size and capacity are 1000, how many memory bytes will be used?

**Ex. 2.** Do the same calculation for a singly linked list which contains 1000 elements.

**Ex. 3.** Do the same calculation for a doubly linked list.

**Ex. 4.** If we define efficiency as actual useful data memory divided by total memory used, what can we say about the efficiency obtained in **Ex. 1.**, **Ex. 2.** and **Ex. 3.** above.

**Ex. 5.** As \(N\), the number of elements, increases to a very large number, what can we say about the efficiency in the three cases: (i) array, (ii) singly linked list, (iii) doubly linked list.

5.6 Arrays versus Linked List, Cache issues

All CPU chips these days have cache memory; cache memory is extra-fast memory close to the CPU. Cache memory has access times one tenth to one twentieth of main memory.

Usually, there are two separate caches, data cache, and instruction cache.

In the case of data cache, when you access a memory location, 4560, say, the cache system will bring memory locations 4560 to 4581 (32 bytes — it could be more, depends on the CPU) into cache; the first memory access will be at the speed of main memory; however, if you access memory 4564, it will already be in cache and this memory access will be at the much faster cache speed.

Hence on machines that have cache, it makes sense to access memory in a orderly manner: e.g. 4560, 4564, 4568, . . .

If you hop about through memory: e.g. 4560, 200057, 26890, . . ., you will lose the speed of the cache memory.

**Ex.** In connection with cache memory, what performance penalty might a program incur when using a linked list instead of an array?
Chapter 6

Stacks and Queues

6.1 Introduction

This chapter introduces Stacks and Queues. Both these may be thought of as specialised lists. Typically, they are implemented using linked data structures.

A Stack is a last-in-first-out (LIFO) container, i.e. the chief methods are push, which operates like to list.push_front, pop (list.pop_front), top (list.front). Stacks are used for the sort of application that requires us to save something that we are working on now when we need to interrupt that and do something else, only to return back to the interrupted task. In the intervening time, the interrupting task can itself be interrupted — so that just one space for saving interrupted tasks is insufficient. LIFO ensures that interrupted tasks are retrieved in the correct order.

The sprung plate stack used in canteens is an often used analogy.

A Queue is a first-in-first-out (FIFO) container, i.e. the chief methods are push, which operates like to list.push_back, pop (list.pop_front), front (list.front). Queues are used for the sort of application that requires us to process things in the same order that they arrive or are encountered. Thus, a Queue operates like a queue of people.

Because of the relative correspondence of behaviour, we find that we can implement a Stack using a List, i.e. a Stack has a List as it’s data member, and we call push_front, front etc. in implementations of Stack interface functions. Likewise Queue.

We could of course make a copy of the List class and do appropriate renaming, but I hope that you are aware of the evils of code duplication.

The standard library provides both containers though the implementations are slightly different from what we describe here.

Since the implementations of these classes have so much in common with the List classes in Chapter 5, our descriptions can be kept brief. Another reason for brevity is existence the standard library versions.
6.2 Queue

6.2.1 Class

/* --- Queue.h ----------------------------------------------*/
#ifndef QUEUE_H
#define QUEUE_H
#include <iostream> #include <list>

template <class T>
class Queue{
  friend std::ostream& operator<< <T>(std::ostream& os, Queue<T> theQ);
public:
  /* not needed if built on std::list
  Queue(); Queue(const Queue<T>& other); Queue();
  Queue& operator = (const Queue<T>& rhs);
  */
  T front() const { return list_.front();};
  T back() const { return list_.back();};
  void push(const T& e){ list_.push_back(e);};
  void pop(){ list_.pop_front();};
  bool empty() const { return list_.empty();};
  int size() const { return list_.size();};
private:
  std::list<T> list_;
};

template <class T>
std::ostream& operator<<(std::ostream& os, Queue<T> theQ){
  os<<"Queue: ("<< theQ.size()<< ") f[ ";
  while(!theQ.empty()){
  os<< theQ.front()<< ","); theQ.pop();
  }
  os<<" ]r"; return os;
}

Figure 6.1: Queue class
6.2.2 A simple client program

Figure 6.2 shows a simple client program. Just to show the similarity, Figure 6.4 shows a program that uses std::queue; note that in Figure 6.4 we provide an implementation of the operator << — this is because the standard library does not provide one. The output from Figure 6.2 is shown in Figure 6.3.

```cpp
/* -- QueueT1.cpp ----------------------------------------
--------------------------------------------------------*/
#include "Queue.h"
#include <iostream>
using namespace std;

int main(){
    Queue<double> x,w;
    x.push(4.4); x.push(3.3); x.push(2.2); x.push(1.1);
    Queue<double> y(x);
    Queue<double> z = x;
    w = x;
    Queue<int> i; i.push(3); i.push(2); i.push(1);

    cout<< "Front of x = "<< x.front()<< endl;

    // the following destroys the queue
    cout<< "Queue x ="<<endl;
    while(!x.empty()){
        cout<< x.front()<< endl;
        x.pop();
    }
    cout<< "Queue y ="<<endl; cout<< y<< endl;
    cout<< "Queue z ="<<endl; cout<< z<< endl;
    cout<< "Queue w ="<<endl; cout<< w<< endl;
    Queue<double> v;
    y.pop(); v = y;
    cout<< "Queue v (y.pop();) ="<<endl;
    cout<< v<< endl;
    cout<< "Queue i ="<<endl; cout<< i<< endl;

    Figure 6.2: QueueT1.cpp
```

6-3
Front of x = 4.4
Queue x =
4.4
3.3
2.2
1.1
Queue y =
Queue: (4) [4.4 3.3 2.2 1.1]r
Queue z =
Queue: (4) [4.4 3.3 2.2 1.1]r
Queue w =
Queue: (4) [4.4 3.3 2.2 1.1]r
Queue v (y.pop();) =
Queue: (3) [3.3 2.2 1.1]r
Queue i =
Queue: (3) [3 2 1]r

Figure 6.3: Output from QueueT1.cpp
#include <queue>
#include <iostream>

using namespace std;

template <class T>
std::ostream& operator<<(std::ostream& os, queue<T> q){
  os << "f[ ";
  while(!q.empty()){
    os << q.front() << ' ';
    q.pop();
  }
  os << " ]b";
  return os;
}

int main(){
  queue<double> x,w;
  x.push(4.4); x.push(3.3); x.push(2.2); x.push(1.1);

  queue<double> y(x);
  queue<double> z = x;
  w = x;

  queue<int> i; i.push(3); i.push(2); i.push(1);
  cout << "Front of x = " << x.front() << endl;
  // the following destroys the queue
  cout << "queue x =" << endl;
  while(!x.empty()){
    cout << x.front() << endl;
    x.pop();
  }
  cout << "queue y =" << endl; cout << y << endl;
  cout << "queue z =" << endl; cout << z << endl;
  cout << "queue w =" << endl; cout << w << endl;

  queue<double> v;
  y.pop();
  v = y;
  cout << "queue v (y.pop()); =" << endl; cout << v << endl;

  cout << "queue i =" << endl; cout << i << endl;
}

Figure 6.4: QueueT2.cpp
6.3 Stack

6.3.1 Class

/* --- Stack.h -------------------------------------------------
---------------------------------------------------------------*/

#ifndef STACKH #define STACKH
#include "../ch05slist/SList.h"

template <class T>
class Stack{
  friend std::ostream& operator<< <T>(std::ostream& os, const Stack<T>& s);
public:
  /* none of these is necessary if we implement using List
     Stack(); Stack(const Stack<T>& other); ~Stack();
     Stack& operator = (const Stack<T>& rhs);
     also, std::stack has no clear
     */
  T top() const {return data_.front();};
  void push(T e){ data_.push_front(e);};
  void pop() { data_.pop_front();};
  bool empty() const{ return data_.empty(); };  
  int size() const{ return data_.size();};
private:
  List<T> data_;  
};

template <class T>
std::ostream& operator<<(std::ostream& os, const Stack<T>& stk){
  os<< "Stack, t. ";
  os<< stk.data_;  
  return os;
}
#undef STACKH

Figure 6.5: Stack.h
6.3.2 A simple client program

Figure 6.6 shows a simple client program. Just to show the similarity, Figure 6.8 shows a program that uses std::stack; note that in Figure 6.8 we provide an implementation of the operator << — this is because the standard library does not provide one. The output from Figure 6.6 is shown in Figure 6.7.

/* --- StackT1.cpp ---------------------------------------
--------------------------------------------------------*/
#include "Stack.h"using namespace std;

int main()
{
    Stack<float> x,w;
x.push(4.4); x.push(3.3); x.push(2.2); x.push(1.1);

    Stack<float> y(x);
    Stack<float> z=x;
    w = x;

    Stack<int> i; i.push(3); i.push(2); i.push(1);
cout<< "top of x = "<< x.top()<< endl;
cout<< "size of x = "<< x.size()<< endl;
cout<< "Stack x "<<endl; cout<< x<< endl;
    x.pop();
cout<< "x popped ...size of x = "<< x.size()<< endl;
cout<< "Stack x "<<endl; cout<< x<< endl;
cout<< "x.empty(): " << x.empty()<< endl;
    
cout<< "Stack y "<<endl; cout<< y<< endl;
cout<< "Stack z "<<endl; cout<< z<< endl;
cout<< "Stack w "<<endl; cout<< w<< endl;
cout<< "Stack i "<<endl; cout<< i<< endl;

Figure 6.6: StackT1.cpp
top of x = 1.1
size of x = 4
Stack x <<
Stack, t. f[ 1.1, 2.2, 3.3, 4.4 ]b

x popped ...size of x = 3
Stack x <<
Stack, t. f[ 2.2, 3.3, 4.4 ]b

x.empty(): 0
Stack y <<
Stack, t. f[ 1.1, 2.2, 3.3, 4.4 ]b

Stack z <<
Stack, t. f[ 1.1, 2.2, 3.3, 4.4 ]b

Stack w <<
Stack, t. f[ 1.1, 2.2, 3.3, 4.4 ]b

Stack i <<
Stack, t. f[ 1, 2, 3 ]b

Figure 6.7: StackT1.cpp output
/* --- StackT2.cpp ---------------------------------------
   j.g.c. 2008-01-31
   exercises std::stack
   --------------------------------------------------------*/
#include <stack>
#include <iostream>
#include <iterator>
using namespace std;

template <class T>
std::ostream& operator<< (std::ostream& os, stack<T> s){
  os << "t[ ";
  while(!s.empty()){
    os << s.top() << ' '; 
    s.pop();
  }
  os << " ]b" << std::endl;
  return os;
}

int main(){
  stack<float> x, w;

  x.push(4.4); x.push(3.3); x.push(2.2); x.push(1.1);

  stack<float> y(x);
  stack<float> z = x;
  w = x;

  stack<int> i; i.push(3); i.push(2); i.push(1);

  cout << "top of x = " << x.top() << endl;
  cout << "size of x = " << x.size() << endl;
  cout << "stack x <<""endl; cout<< x<< endl;

  x.pop();
  cout << "x popped ...size of x = " << x.size() << endl;
  cout << "stack x <<""endl; cout<< x<< endl;
  cout << "x.empty(): " << x.empty() << endl;

  cout << "stack y <<""endl; cout<< y<< endl;
  cout << "stack z <<""endl; cout<< z<< endl;
  cout << "stack w <<""endl; cout<< w<< endl;
  cout << "stack i <<""endl; cout<< i<< endl;

  return 0;
}

Figure 6.8: StackT2.cpp
6.4 Stack Application — RPN Calculator

In the early days of handheld calculators in the 1970s, Hewlett-Packard (HP) was a market leader. Their early calculators used an arithmetic expression representation scheme called reverse Polish notation (RPN). RPN is postfix notation with some additional wrinkles. There are three schemes for representing arithmetic and algebraic expressions: infix, prefix, and postfix.

**Infix** Infix is the one we are used to for plain arithmetic, for example $3 + 2$, the operator $+$ is *in* between the operands. A more complex one is $6/3 + 2*4$; because of operator precedence order $\times$, $\div$, $+$, $-$, this evaluates to $2 + 8 = 10$. In the examples here, we avoid operator precedence by expressing expressions as *fully bracketed infix*. In fully bracketed infix the expression $6/3 + 2*4$ becomes $((6/3) + (2*4))$ — there is a left bracket and a right bracket corresponding to each operator and so there is no ambiguity about which calculation gets done in which order.

**Prefix** Prefix is the one we use to for most mathematical and programming functions. Thus, let’s say we have a binary add function / method. We write its application as $\text{add}(3, 2)$, the function add is *pre* (before) the operands. A more complex one is $\text{add}(\text{div}(6, 3), \text{mult}(2*4))$.

**Postfix** Postfix is the one associated with RPN calculators. We write the addition above as $3 \ 2 +$, the operator $+$ is placed *post* (after) the operands. A more complex one is $6 \ 3 / \ 2 \ 4 * +$; in other words,

- take 6 and 3 and apply $\div$; save the result (2);
- take 2 and 4 and apply $*$; save the result (8);
- now apply $+$ to the two previously saved results.

The mention of *save the result* and *the two previously saved results* should make you think of a stack.

### 6.4.1 RPN Algorithm

A very rough statement of the algorithm is given in Figure 6.9. We use a stack, and we assume that the operands and operators are stored in a queue in the order the user typed them, for example, (front) $6 \ 3 / \ 2 \ 4 * +$ (back)

Figure 6.10 shows parts of a C++ program which implements the algorithm in Figure 6.9. The complete program is available in the programs directory as EvalRPN.cpp. You will notice that we use strings as general purpose operator / operand variable; we could have use a polymorphic expression-element type hierarchy, but here we want to keep things as simple as possible.

The output of Figure 6.10 is given in Figure 6.11.
Input: queue, Q, containing postfix expression
Define empty stack, S.

while(Q is not empty){
    remove the front of Q into z;
    if (z is an operand (number)) push z onto S;
    else if (z is an operator (+, -, *, /) op ) {
        remove the top two elements of S into x, y;
        apply op(x, y) to get res;
        push res onto S;
    } // else
} // while

The overall answer is at the top of S.

Figure 6.9: RPN evaluation
string eval(string op2, string op1, string opr){
    char opch= opr[0];
    istringstream is1(op1);
    int v1; is1 >> v1;
    istringstream is2(op2);
    int v2; is2 >> v2;
    int res;
    if(opch=='*')res = v1*v2;
    else if(opch== '/')res = v1/v2;
    else if(opch== '+')res = v1+v2;
    else if(opch== '-')res = v1-v2;
    else res = 0; // effectively ignore
    ostringstream os1;
    os1<< res;
    return os1.str();
}

string evalPostFixQueue(queue<string> in){
    // assumes that there is at least one element in the queue
    stack<string> s;
    string oq, i1, i2, res;

    while(!in.empty()){  
        oq= in.front(); in.pop();
        if(isNumeric(oq))s.push(oq);
        else if(isOperator(oq)){
            i1= s.top(); s.pop();
            i2= s.top(); s.pop();
            res = eval(i2, i1, oq);
            s.push(res);
        }
        }
        i1= s.top();
        return i1;
    }

int main(int argc, char **argv){
    string s("1 2 1 + 3 2 + * ");

    queue<string> qPostfix= fromString(s);
    cout<< "Postfix queue = " << toString(qPostfix) << endl;

    cout<< "Evaluation of postfix queue " << toString(qPostfix) << " = " << endl;
    cout<< evalPostFixQueue(qPostfix) << endl;
}

Figure 6.10: EvalRPN.cpp
String = 1 21 + 3 2 + *
Postfix queue = 1, 21, +, 3, 2, +, *,

Evaluation of postfix queue 1, 21, +, 3, 2, +, *,
= 110

Figure 6.11: Output of EvalRPN.cpp
6.4.2 Conversion from Infix to Postfix

Okay then, postfix notation makes our calculator very easy to implement, but what if we want to enter the calculation in the more familiar infix? We need to be able to convert an infix expression to postfix. To keep think simple, we limit ourselves to *fully bracketed infix*.

This can be done using Dijkstra’s *shunting algorithm*. It gets the name *shunting* because of its action looks like the shunting action (into a siding) that they used to use when transferring railway carriages from from one train to another; for train think queue; for siding, think stack.

A very rough statement of the algorithm is given in Figure 6.12. We use a stack, and we assume that the infix expression is stored in an input queue containing the operands and operators in the order the user typed them, for example,

(front) ( (6/3) + (2*4) ) (back)

A pictorial example is given in appendix 6.6.

**Input:** queue, Qin, containing infix expression

Define empty operator stack, S.

Define empty output queue, Qpost.

while(Qin is not empty){
    remove the front of Qin into z;
    if (z is an operand (number)) push z into Qpost;
    else if (z is an operator (+, -, *, /)) push z onto S;
    else if (z is a right-bracket ‘)’) {
        pop the operator at the top of S and push it into Qpost;
    } // else
} // while

The postfix version of the expression is now in Qpost.

Figure 6.12: Dijkstra’s Shunting Algorithm, simplified to fully-bracketed-infix.

Figure 6.13 shows parts of a C++ program which implements the algorithm in Figure 6.12. The complete program is available in the programs directory as EvalInfix.cpp. The code for evalPostfixQueue is given above in Figure 6.10.

The output of Figure 6.10 is given in Figure 6.14.
queue<string> inToPostfix(queue<string> in) {
    queue<string> post;
    stack<string> s;
    string oq, os;

    while (!in.empty()) {
        oq = in.front(); in.pop();
        if (isNumeric(oq)) post.push(oq); // main line
        else if (isOperator(oq)) s.push(oq); // siding
        else if (isRightBracket(oq)) { // remove top operator to output
            post.push(s.top());
            s.pop();
        }
        else if (isLeftBracket(oq)) {} // ignore
        else {} // ignore
    }
    return post; // result postfix queue
}

int main(int argc, char **argv) {
    string s("((1+21)*(3+2)) ");
    // String s = "((10+3)*(5+1)) ";
    // String s = "21 "; // end space necessary to stop
    // fromString falling off end
    cout << "String = " << s << endl;

    queue<string> qInfix = fromString(s);
    cout << "Infix queue = " << toString(qInfix) << endl;

    queue<string> qPostfix = inToPostfix(qInfix);
    cout << "Postfix queue = " << toString(qPostfix) << endl;

    cout << "Evaluation of postfix queue " << toString(qPostfix) << " = " << endl;
    cout << evalPostfixQueue(qPostfix) << endl;
}

Figure 6.13: EvalInfix.cpp

String = ((1+21)*(3+2))
Infix queue = (, (, 1, +, 21, ), *, (, 3, +, 2, ), ),
Postfix queue = 1, 21, +, 3, 2, +, *
Evaluation of postfix queue 1, 21, +, 3, 2, +, * = 110

Figure 6.14: Output of EvalInfix.cpp

6–15
6.5 Stack Application — Balancing Brackets

A simpler stack application is that of balancing brackets in programs or text. When the algorithm encounters a left bracket such as one of (, {, [ it pushes it onto a stack; when it encounters a right bracket such as one of ), }, ] it removes the bracket from the top of the stack and compares it to the just encountered right bracket; if they do not match, an error is indicated.

The program in Figure 6.15 implements this algorithm.
```cpp
int indexOf(int c, const char *s){
    int p = 0;
    while(s[p] != '0'){
        if(s[p] == c) return p;
        ++p;
    }
    return -1;
}

bool match(int cr, int cl, const char *rs, const char *ls){
    return (indexOf(cr, rs) == indexOf(cl, ls) && indexOf(cr, rs) != -1);
}

bool isIn(int c, const char *s){
    return (indexOf(c, s) != -1);
}

int main(int argc, char *argv[]){
    stack<char> stk;
    const char *lBrack = "{{", const char *rBrack = "})";
    bool more = true; int nline = 0, nchar = 0;
    while(more){
        char c; cin.get(c); ++nchar;
        if(c == '\n'){
            ++nline; nchar = 0;
        }
        cout << c;
        //cout << "<" << c << "">" << endl;
        if(isIn(c, lBrack)) stk.push(c);
        else if(isIn(c, rBrack)){
            if(!stk.empty()){
                int ctop = stk.top(); stk.pop();
                //System.out.println(c, ctop = "+(char)c+(char)ctop);
                if(!match(c, ctop, rBrack, lBrack)){
                    cout << "Error: line " << nline << " char " << nchar << ', ' << char(c) << endl;
                    return 0;
                }
            }
            else { // stk empty, nothing to compare to c
                cout << "Error: line " << nline << " char " << nchar << ', ' << char(c) << endl;
                return 0;
            }
        }
        // just ignore other characters
        if(c == '#') more = false; //kludge for EOF
    } // while
    if(!stk.empty()){
        cout << "Error: missing a right bracket" << endl; return 0;
    }
    cout << "Successful!" << endl;
}
```

Figure 6.15: Balancing Brackets, Brack.cpp
6.6 Appendix. Illustration of Dijkstra’s Shunting Algorithm

The diagram below is from http://www.allisons.org/ll/AlgDS/Stack/ with modifications by me; the example there does not restrict itself to fully bracketed and so gives a more complete description of the shunting algorithm.

The shunting algorithm uses a stack of operators.

Operands go straight through on the main line but operators go into the siding.

A right bracket is a signal that we must associate the last pushed operator (*) with the last two operands pushed onto the postfix queue; i.e. we pop it off the stack and push it onto the output queue.
Apply the right bracket rule again --- twice

And we end up with   ab*cd++
Chapter 7

Trees

7.1 Introduction

Note: modified on 2008-02-24 to reflect reorganisation of BST.h; but this is only reorganisation — there is no change to the structure or interface of BST.

Here we will cover only binary trees. However, once you see how to implement a binary tree, it is relatively straightforward to implement an n-ary tree. You should read the relevant chapter of (Penton 2003) to see how general n-ary trees are implemented there.

In Chapters 2 and 5 we have considered arrays and linked list implementations of lists as collection data structures. Each has its advantages . . . speed of insertion, speed of searching, etc.

In Chapter 4 we have seen that an ordered array can be searched quickly, i.e. $O(\log N)$ for binary search. If we start with an unordered array, we can sort it in $O(N \log N)$ time (using merge sort or quick sort) and thereafter we can use our $O(\log N)$ binary search. But it means that the overall task takes $O(N \log N)$, which is more expensive than the $O(N)$ of plain old linear search. On the other hand, if we need to do lots of searches, say $M$ of them, then the cost of the $O(N \log N)$ sort can be amortised over the $M$ searches. If $M$ is very large, the cost of each searches will tend to $O(\log N)$.

Another possibility is that we create an ordered array from the outset, i.e. we insert items in their proper order. Consider a partially filled array that is in order; we want to insert a new element. There are two steps: (i) determining the position in which to insert; this is an $O(\log N)$ operation (like search — check this?); (ii) insert, which is an $O(N)$ operation. That means that the overall insert is an $O(N)$ operation. There are $N$ elements to insert, so creating the complete array is $O(N^2)$. Hence, the sorting option above is better.

Could we use a linked list? A linked list handles insertion and deletion inexpensively, but searching requires starting at the beginning and visiting every element until a match is found, i.e. $O(N)$; even if the list is sorted, you cannot use binary search because you don’t have random access.

A binary search tree gives us the best of all worlds. Searching performance is of the same order as for ordered arrays, $O(\log N)$; determining where to insert takes $O(\log N)$. Actual insertion and deletion can be done with the efficiency of a linked list, i.e. $O(1)$.

The simplest binary search tree retains the efficiency mentioned above only as long as the data inserted into it arrives in random order; in this case the tree remains balanced; if the data do not
arrive in random order, the tree becomes unbalanced and performance deteriorates — leading to the tree becoming merely an expensive linked list. There are more elaborate tree structures which overcome this shortcoming, for example AVL trees, Red-Black trees; but we do not need to cover these.

You will notice that the C++ standard library has no collection called tree or BST. This is because the standard library names the data structure for its use, i.e. set, rather than its implementation, i.e. tree. std::set is implemented using some form of binary search tree.

Set  A set is a collection of elements (often called keys); the chief operations are: (i) insert a new key, (ii) search for a key. Like mathematical sets, duplication of keys is not allowed.

Map  A map (or dictionary) is a set of key, value pairs. Insertion and search is based on the key. Maps have numerous applications: symbol table in a compiler, key is a string (symbol name), value is address; telephone directories — what is the key? what is the value?; word dictionaries; etc. etc.

7.2 Binary Search Tree

A binary tree can be defined recursively as a collection of nodes:

- Either, an empty tree;
- Or, a root node which consists of:
  - a left (sub)tree; also called left child;
  - a right (sub)tree; also called right child;

Each of the latter is itself a binary tree.

In addition, each node contains a value (key). A node whose sub-trees are empty is called a leaf.

It is easy to generalise to n-ary trees – here each non-leaf node consists of n sub-trees.

File system directories is a good everyday example of an n-ary tree. Incidentally, think of the difficulty of a non-recursive (deterministic) definition of a file system directory structure.

7.2.1 Binary Search Tree Code

Figures 7.1 to 7.5 show the binary search tree class. Figure 7.6 shows an small test program, and Figure 7.7 shows the output.
/* --- BST.h -----------------------------------------
Binary Search Tree
Ch 9 Data Structures for Game Developers, Allen Sherrod
mods. j.g.c. 2008-01-11, recursive methods, DisplayAsTree
2008-02-01 methods renamed to be compatible with std::set
2008-02-24 function defs separated from class declarations
2008-11-16, need for > and == removed
------------------------------------------------------------ */
#ifndef BST_H
#define BST_H
#include <iostream>
#include <stack>

using namespace std;

template<typename T> class BST;

template<typename T>
class Node{
friend class BST<T>;
public:
    Node(T key) : m_key(key), m_left(NULL), m_right(NULL){}
    ~Node();
    T getKey(){ return m_key; }

private:
    T m_key;
    Node *m_left, *m_right;
};

// ------------- member functions of Node -------------

template<typename T>
Node<T>::~Node(){
    if(m_left != NULL){ delete m_left; m_left = NULL; }
    if(m_right != NULL){ delete m_right; m_right = NULL;}
}

// continued

Figure 7.1: Binary Search Tree, BST.h, part 1, Node

7-3
template<typename T>
class BST{
public:
    BST() : m_root(NULL){}
    ~BST();
    void insert(T key);
    bool find(T key);
    void erase(T key);
    void DisplayPreOrder();
    void DisplayPostOrder();
    void DisplayInOrder();
    void DisplayAsTree();

private:
    Node<T> *m_root;

    Node<T>* insert(T key, Node<T>* n);
    Node<T>* erase(T key, Node<T>* n);
    T disconnectSucc(Node<T>* n);
    bool find(T key, Node<T>* n);
    void DisplayPreOrder(Node<T> *node);
    void DisplayPostOrder(Node<T> *node);
    void DisplayInOrder(Node<T> *node);
};

template<typename T>
BST<T>::~BST<T>(){
    if(m_root != NULL){
        delete m_root;
        m_root = NULL;
    }
}

template<typename T>
void BST<T>::insert(T key){
    m_root = insert(key, m_root);
}

template<typename T>
bool BST<T>::find(T key){
    return find(key, m_root);
}

template<typename T>
void BST<T>::erase(T key){
    m_root = erase(key, m_root);
}

template<typename T>
void BST<T>::DisplayPreOrder(){
    DisplayPreOrder(m_root);
} // continued ...

7-4
template<typename T>
void BST<T>::DisplayPostOrder()
{
    DisplayPostOrder(m_root);
}

template<typename T>
void BST<T>::DisplayInOrder()
{
    DisplayInOrder(m_root);
}

// -------- private member functions of BST -------------

template<typename T>
Node<T>* BST<T>::insert(T key, Node<T>* n){
    if (n == NULL) n = new Node<T>(key);
    else if(key < n->m_key) n->m_left = insert(key, n->m_left);
    else if(n->m_key < key) n->m_right = insert(key, n->m_right);
    else {} // equal, do nothing
    return n;
}

template<typename T>
Node<T>* BST<T>::erase(T key, Node<T>* n){
    if (n == NULL) return n;
    else if(key < n->m_key) n->m_left = erase(key, n->m_left);
    else if(n->m_key < key) n->m_right = erase(key, n->m_right);
    else if(n->m_left != NULL && n->m_right != NULL)
        n->m_key = disconnectSucc(n);
    else if(n->m_left == NULL) n = n->m_right;
    else n = n->m_left;
    return n;
}

template<typename T>
T BST<T>::disconnectSucc(Node<T>* n){
    Node<T>* succParent = n;
    Node<T>* succ = n;
    Node<T>* curr = n->m_right;
    // locate successor
    while(curr != NULL){
        succParent = succ;
        succ = curr;
        curr = curr->m_left;
    }
    if (succ == succParent->m_right) succParent->m_right = succ->m_right;
    else succParent->m_left = succ->m_right;

    return succ->m_key;
} // continued ...
template<typename T>
bool BST<T>::find(T key, Node<T>* n) {
    if (n == NULL) return false;
    //else if(key == n->m_key) return true;
    else if(n->m_key < key) return find(key, n->m_right);
    else if(key < n->m_key) return find(key, n->m_left);
    else return true; // ==
}

template<typename T>
void BST<T>::DisplayPreOrder(Node<T> *node) {
    if(node != NULL) {
        cout << node->m_key << "", ";
        DisplayPreOrder(node->m_left);
        DisplayPreOrder(node->m_right);
    }
}

template<typename T>
void BST<T>::DisplayPostOrder(Node<T> *node) {
    if(node != NULL) {
        DisplayPostOrder(node->m_left);
        DisplayPostOrder(node->m_right);
        cout << node->m_key << "", ";
    }
}

template<typename T>
void BST<T>::DisplayInOrder(Node<T> *node) {
    if(node != NULL) {
        DisplayInOrder(node->m_left);
        cout << node->m_key << "", ";
        DisplayInOrder(node->m_right);
    }
} // continued

Figure 7.4: Binary Search Tree, BST.h, part 4
template<typename T>
void BST<T>::DisplayAsTree() {
    std::stack<Node<T>*> global;
    global.push(m_root);
    int nsp = 32;
    bool isRowEmpty = false;
    std::cout << "............................................................\n";
    while(!isRowEmpty) {
        std::stack<Node<T>*> local;
        isRowEmpty = true;
        for(int j = 0; j < nsp; j++) std::cout << ' ';
        while(!global.empty()) {
            Node<T>* temp = global.top();
            global.pop();
            if(temp != NULL) {
                std::cout << temp->m_key;
                local.push(temp->m_left);
                local.push(temp->m_right);
                if(temp->m_left != NULL || temp->m_right != NULL) isRowEmpty = false;
            } else {
                std::cout << "--";
                local.push(NULL);
                local.push(NULL);
            }
            for(int j = 0; j < nsp*2 - 2; j++) std::cout << ' ';
        }
        std::cout << '\n';
        nsp /= 2;
        while(!local.empty()) {
            global.push(local.top());
            local.pop();
        }
    }
    std::cout << "............................................................\n";
    std::cout << std::endl;
}

Figure 7.5: Binary Search Tree, BST.h, part 5
### 7.2.2 Notes on BST.h

1. If you understand *linked lists*, binary trees are straightforward. In place of `Link`, we have `Node`. While a `Link` has an element and a pointer to the next `Link`, a `Node` has an element (`key`) and pointers to left and right (next) `Nodes`.

   ```
   T m_key;
   Node *m_left, *m_right;
   ```

2. Recall that a *singly linked list* has just a pointer to the next link, but a *doubly linked list* has a pointer to the previous link as well. Some trees need a node that has a pointer to the parent (previous); for example, think of a file directory tree and `cd ..` (connect to the parent directory). But a *binary search tree* does not need such a pointer.

3. The last two trees in Figure 7.7 show highly unbalanced trees, i.e. the result of keys arriving for insertion in far from random order. In both cases, the tree becomes an expensive *linked list* and search takes $O(N)$ instead of $O(\log N)$.

4. The most difficult part of BST is *erase* of an element; I leave it here for completeness, but note that there are many applications where erasure is never needed or where it is so infrequent that some workaround may be used in its place. Therefore, we will not spend time understanding method `erase`.

5. `operator <`. If you want to use a BST for `key` variable/object values then that type or class will have to have defined an `operator <`, see //A, //B, where `T key` is the variable/object to be inserted.

   ```
   template<typename T> Node<T>* BST<T>::insert(T key, Node<T>* n) { 
   if (n == NULL) n = new Node<T>(key);
   else if(key < n->m_key) n->m_left = insert(key, n->m_left); //A
   else if(n->m_key < key) n->m_right = insert(key, n->m_right); //B
   else {} // equal, do nothing. Why?
   return n;
   }
   ```

   The same is the case for `std::set`, `std::map` etc., see Chapter 13. Attempting create a BST of `key value` type `T` which has no `operator <` will result in a compiler error.

6. Notice that in //A, //B above we deliberately use only `operator <` and never `operator ==` (or for that matter operator `==`).

7. In `find`, we are checking whether `key` is in the container. At //C we test if `key` is greater than the value in that node (i.e. `key greater than n->m_key` is tested by `n->m_key` less than `key`) and if that is true, then we search the right-hand subtree. At //D we test if `key` is less than the value in that node and if that is true, then we search the left-hand subtree.

   And if neither `key < n->m_key` nor `n->m_key < key`, then they must be equal and we have found `key`.

   ```
   bool BST<T>::find(T key, Node<T>* n) { 
   if (n == NULL) return false;
   else if(n->m_key < key) return find(key, n->m_right); //C
   else if(key < n->m_key) return find(key, n->m_left); //D
   else return true; // ==
   ```
7.2.3 Traversal of Trees

In many applications involving trees, the major distinction is according to how you iterate through the tree, visiting and processing nodes – tree traversal. Traversal algorithms are defined recursively from three steps:

- Process a node;
- Recursively visit and process the left child;
- Recursively visit and process the right child;

With these subtasks, there are six possible arrangements:

1. Process the node, then left child, then right;
2. Process left child, then node, then right;
3. Process left child, then right, then node;
4. Process the node, then right, then left;
5. Process right child, then node, then left;
6. Process right child, then left, then node;

In most cases, left and right are of equal standing, or if not, the tree is populated accordingly; thus, we can consider just the first three possibilities:

1. Process the node, then left, then right; this is **pre-order** traversal;
2. Process left, then node, then right; this is **in-order** traversal;
3. Process left, then right, then node; this is **post-order** traversal;

If you look at the section on **pre-fix**, **in-fix**, and **post-fix** expressions, you will see the connection.

7.2.4 Exercising Program, BSTT1.cpp
```cpp
#include<iostream>   #include"BST.h"   using namespace std;

int main(int argc, char **argv){
    BST<int> t;
    t.insert(20); t.insert(10); t.insert(12); t.insert(27);
    t.insert(9);  t.insert(50); t.insert(33); t.insert(6);

    if(t.find(20))cout << "The key 20 found!" << endl;
    else cout << "The key 20 NOT found!" << endl;

    if(t.find(14)) cout << "The key 14 found!" << endl;
    else cout << "The key 14 NOT found!" << endl;

    if(t.find(27))cout << "The key 27 found!" << endl;
    else cout << "The key 27 NOT found!" << endl;

    cout << " Pre-order: "; t.DisplayPreOrder(); cout << endl;
    cout << "Post-order: "; t.DisplayPostOrder(); cout << endl;
    cout << " In-order: "; t.DisplayInOrder(); cout << endl;
    cout << "As Tree\n "; t.DisplayAsTree(); cout << endl;
    cout << "t.erase(27);\n"; t.erase(27);

    if(t.find(20))cout << "The key 20 found!" << endl;
    else cout << "The key 20 NOT found!" << endl;

    if(t.find(14)) cout << "The key 14 found!" << endl;
    else cout << "The key 14 NOT found!" << endl;

    if(t.find(27))cout << "The key 27 found!" << endl;
    else cout << "The key 27 NOT found!" << endl;

    cout << "Pre-order (depth first): "; t.DisplayPreOrder(); cout << endl;
    cout << "Post-order: "; t.DisplayPostOrder(); cout << endl;
    cout << " In-order: "; t.DisplayInOrder(); cout << endl;
    cout << "As Tree\n "; t.DisplayAsTree(); cout << endl;

    BST<int> t2;
    for(int i = 0; i< 5; ++i){ t2.insert(i); }
    cout << "t2\n "; t2.DisplayAsTree(); cout << endl;

    BST<int> t3;
    for(int i = 20; i > 10; i-= 2){ t3.insert(i);}
    cout << "t3\n "; t3.DisplayAsTree(); cout << endl;
```

Figure 7.6: BST Test program, BSTT1.cpp
The key 20 found!
The key 14 NOT found!
The key 27 found!

Pre-order: 20 10 9 6 12 27 50 33
Post-order: 6 9 12 10 33 50 27 20
In-order: 6 9 10 12 20 27 33 50

As Tree

```
   20
   / \
  10  27
 /  / \
9  12 --
/  /  /  \
6 -- -- -- -- 33
```

```
t.erase(27);
The key 20 found!
The key 14 NOT found!
The key 27 NOT found!
```

Pre-order (depth first): 20 10 9 6 12 50 33
Post-order: 6 9 12 10 33 50 20
In-order: 6 9 10 12 33 50

As Tree

```
   20
   / \
  10  50
 /  / \
9  12 33
/  /  /  \
6 -- -- -- -- 33
```

```
t2
```
```
  0
 / \
-- -- 1
 /  /  \
-- -- -- -- -- -- -- 3
```
```
t3
```
```
  20
  / \
18 --
 /  \
16 --
 /  \
14 --
 /  \
12 --
```

Figure 7.7: Output from BSTT1.cpp
7.3  n-ary Trees

Recalling the recursive definition of *binary tree* in section 7.2, an *n-ary* tree may be defined recursively as a collection of nodes:

- Either, *an empty tree*;
- Or, a *root* node which consists of:
  - zero or more (sub)trees; also called *children*;
  - Each of the children is itself an *n-ary tree*.

In addition, each node contains a value (key). A node whose subtrees are empty is called a *leaf*.

The zero or more subtrees is often implemented as a *list* of some sort.

The robot example that we covered in graphics was implemented using a crude *n-ary tree*. 
Chapter 8

Recursion

8.1 Introduction

The notion of recursion is fundamental in computer science and mathematics. Already, we have seen recursion in Chapter 4, merge sort, quick sort, and binary search. Then, in Chapter 7, we saw a recursive definition of a binary tree.

A binary tree may be defined recursively as a collection of nodes:

- Either, an empty tree;
- Or, a root node which consists of:
  - a left (sub)tree; also called left child;
  - a right (sub)tree; also called right child;
Each of the latter is itself a binary trees.

Definition of Natural Numbers and Arithmetic using Recursion In a similar vein, the natural numbers, \( \mathbb{N} \), where \( x \) is a general member of \( \mathbb{N} \), may be defined by:

\[
\begin{align*}
x &= 0, \text{ an initial number, zero} \\
or s(x), \text{ where } s \text{ is the successor function.}
\end{align*}
\]

Arithmetic functions – the methods of \( \mathbb{N} \), may be specified using recursion:

1. Addition, \( + \):

\[
\begin{align*}
x + 0 &= x, \text{ the basis part} \\
x + s(y) &= s(x + y), \text{ the recursive part.}
\end{align*}
\]

2. Multiplication, \( \times \):

\[
\begin{align*}
x \times 0 &= 0, \text{ the basis part} \\
x \times s(y) &= x + x \times y, \text{ the recursive part.}
\end{align*}
\]
3. Exponentiation, (power):

\[ x^0 = 1, \text{ the basis part} \]  \hspace{1cm} (8.7)
\[ x^{s(y)} = x \times x^y, \text{ the recursive part.} \]  \hspace{1cm} (8.8)

Thus, through recursion we can achieve elegant and natural expressions of certain concepts that would otherwise require much more lengthy description. From our purposes, that is from a practical point of view, recursion may be used in the development of elegant and efficient data structures and algorithm to solve a wide variety of problems.

### 8.2 Mathematical Induction

Recursion and mathematical induction are two sides of the same coin.

If you know that some predicate, \( P \), say, is true of zero, and you know that, if it is true of an arbitrary natural number, \( n \), then it is also true for \( n + 1 \), then you can conclude that it is true for any natural number.

In other words, if the following two statements are true:

1. \( P \) is true of 0. This is the \textit{basis};
2. If, given that \( P \) is true of \( n \), then \( P \) is true of \( n + 1 \), i.e. the successor of \( n \). This is the \textit{induction}.

then \( P \) is true for every natural number.

Example. In Chapter 3, we used the fact that the sum from 1 to \( n \) is \( \frac{n(n+1)}{2} \). Let’s call the formula \( \frac{n(n+1)}{2} \), \( F(n) \).

\[ \sum_{i=1}^{n} i = F(n) = \frac{n(n+1)}{2}. \]  \hspace{1cm} (8.9)

Proof. Basis.

\[ \sum_{i=0}^{0} i = F(0) = 0(1) = 0. \]

Induction. Assume,

\[ \sum_{i=1}^{n} i = F(n) = \frac{n(n+1)}{2} \]
Examine,
\[
\sum_{i=1}^{n+1} i = (1 + 2 + \ldots + n) + n + 1,
\]
\[
= \sum_{i=1}^{n} i + n + 1,
\]
\[
= \frac{n(n + 1)}{2} + n + 1, \text{ by our induction assumption},
\]
\[
= \frac{n(n + 1)}{2} + \frac{2(n + 1)}{2},
\]
\[
= \frac{n(n + 1) + 2(n + 1)}{2},
\]
\[
= \frac{n^2 + 3n + 2}{2},
\]
\[
= \frac{(n + 1)(n + 2)}{2},
\]
which ends our proof – we have shown that:

1. \(F(0)\) is true – the basis;

2. And, given that \(F(n)\) is true, then \(F(n + 1)\) is true – the induction.

### 8.2.1 Deduction versus Induction

A brief note of caution against confusing mathematical induction with logical induction.

**Deduction in Logic** In logic, deduction is the strongest form of reasoning. From the premises:

- All people are mortal.
- Socrates is a person.

we can deduce:

- Therefore, Socrates is mortal.

*Induction* is a weaker form of reasoning – reasoning based on generalising from a few particular cases.

On my holidays in Bundoran, I observe a number of sea-birds — all of which are white, and I conclude, using induction, that all sea-birds are white. Of course, the weakness of this form of reasoning becomes obvious as soon as I encounter a cormorant or a puffin.

On the other hand, mathematical induction is in no way approximate — and the proof steps given above: \(P\) is true of 0 \ldots If, given that \(P\) is true of \(n\) \ldots then \(P\) is true of \(n + 1\) \ldots then \(P\) is true for every natural number \ldots are always valid.
8.3 Recursive Algorithms

A recursive algorithm is one which solves its problem by restating it in terms on one or smaller versions of the same problem. When the algorithm is expressed in a Java method/function, this normally means that the function calls itself — it is a recursive function.

Example. Factorial. \( f(n) \equiv n! = n(n-1)(n-2)\ldots1 \), and \( f(0) \) is defined to be 0:

\[
\begin{align*}
n! & = n \times (n-1)!, & \text{for } n \geq 1, & \text{the recursive part,} \\
n! & = 1, & \text{for } n = 0, & \text{the basis part.}
\end{align*}
\]

Equations 8.10 and 8.11 lead immediately to a recursive algorithm and the recursive C++ function shown in Figure 8.1.

```cpp
/* ----------- fact.cpp ---------------------------- 
* recursive factorial 
* j.g.c. 18/02/00, C++ 2008-02-03 
-------------------------------- */ 
#include <iostream>
using namespace std;

int fact(int n){
    cout<< "fact "<< n<< endl;
    if(n == 0)return 1;
    else return n*fact(n-1);
}

int main(int argc, char *argv[]){
    int f= fact(6);
    cout<< "fact(6) = "<< f<< endl;
}
```

Figure 8.1: Recursive factorial.

If you want to see how the recursive sequence of calls proceeds, Figure 8.2. shows the execution of fact(3).
Figure 8.2: Recursive factorial of 3.
Of course, recurrence relations, such as equations 8.10 and 8.11, have been around long before computer programs; they can in fact be expressed in algorithms not involving recursive functions. Moreover, it is always possible to transform a recursive algorithm into one involving loops; for example, a loop version of factorial is shown in Figure 8.3.

```c
factloop(int n){
    for(int i=1, fact=1; i<=n; i++)fact= fact*i;
    return fact;
}
```

Figure 8.3: Factorial using a loop.

If you compare the recursive version `fact` with the loop version `factloop()`, whatever your initial impressions of recursion, I think you will have to agree that `fact` is easier to read and comprehend. In addition, we note that `fact` effectively contains nothing but mathematical equations, whilst `factloop` contains local variables and assignment — two things which make (imperative) programming difficult to understand.

**Termination of Recursive functions** In order for it to terminate, rather than recurse forever, a recursive algorithm must have a basis or base case; in addition, the recursive part must get us closer to the basis. `fact` satisfies this latter requirement by reducing by one the argument for each recursive call, i.e. `else return n*fact(n-1);`. We have more to say later about proof of termination.

Euclid’s algorithm for computing the greatest common divisor (GCD) (also called highest common factor) of two integers is one of the oldest known algorithms. It is based on the fact that the GCD of $x$ and $y$, for $x > y$ is the same as the GCD for $y$ and $x \mod y$; $x \mod y$ is the remainder after $x$ is divided by $y$; in C++, `x%y`. This is because an integer $z$ divides both $x$ and $y$ if and only if $z$ divides both $x$ and $x \mod y$, because $x = x \mod y + k.y$, where $k$ is an integer; actually, $k = x/y$, where the division is integer division.

The recursive GCD function is shown in Figure 8.4; Figures 8.5 and 8.6 show how it progresses. In the case of `gcd(1024, 4)`, we require just two steps. In the case of `gcd(1719, 131)`, in spite of the fact that 1719, 131 are relatively prime, we get a result in just five steps. If two integers are relatively prime, the largest common multiplier is 1.
/* ---------------- Gcd.cpp ------------------------*/
#include <iostream>
using namespace std;

int gcd(int m, int n){
    cout << "gcd " << m << " " << n << endl;
    if(n > m) return gcd(n, m);
    else if(n==0) return m;
    else return gcd(n, m%n);
}

int main(int argc, char *argv[]){
    int g = gcd(1719, 131);
    cout << "gcd(1719, 131)= " << g << endl;
}

Figure 8.4: Recursive GCD, Gcd.cpp

gcd 1719 131
gcd 131 16
gcd 16 3
gcd 3 1
gcd 1 0

gcd(1719, 131)= 1

Figure 8.5: Recursive GCD(1719, 131), output

gcd 1024 4
gcd 4 0

gcd(1024, 4)= 4

Figure 8.6: Recursive GCD(1024, 4), output
8.4 Recursion in Compilers and Calculators

The following method performs recursive evaluation of prefix expressions held in a queue. A queue is a good model for a file stream; in other words we could replace the code `q.front(); q.pop()`, which consumes one item from the front of the queue, with a `char ch = readFromFile()`. Note: we have already covered evaluation of a postfix (RPN) expression in section 6.4 and the evaluation of an infix expression in Figure 6.10 — which uses Dijkstra’s shunting algorithm to convert infix to postfix and then uses the RPN evaluation algorithm in section 6.4.

We can define the recursive prefix evaluation as:

1. If the front of the queue is an operator, then operate on `eval(q)` and `eval(q)`; note that `eval()` consumes the queue; thus, `eval()` on "* 4 6" is 24; this is the recursion case;
2. If the front of the queue is a number, `n`, evaluate to `n`; this is the base case; thus, `eval()` on "2" is 2.

The code to perform recursive evaluation is given in Figure 8.7 — full program in `eval.cpp` in the progs directory for this chapter.

```cpp
int eval(queue< string >& q) {
    string s = q.front(); q.pop();
    if (isAdd(s)) return eval(q) + eval(q);
    else if (isSub(s)) return eval(q) - eval(q);
    else if (isMult(s)) return eval(q) * eval(q);
    else if (isDiv(s)) return eval(q) / eval(q);
    else return numericValue(s);
}
```

Figure 8.7: Recursive evaluation of prefix expressions.

In Figure 8.7, we restrict ourselves to binary operators; again, it would be simple to alter `eval()` to handle, for example, unary -. Also, it is clear that prefix expressions are a lot easier to understand when punctuated with brackets — again it would be simple to cater for them.

The example `* + 7 * * 4 6 + 8 9 5` is much easier to understand if we insert punctuating brackets, here we use different brackets to make the situation more obvious: `*<+{7, *[+(4, 6), +(8, 9)]}, 5>;` thus, translating the operators to function names: `mult(plus(7, mult(mult(4, 6), plus(8, 9)) ), 5);` i.e. `mult(plus(7, mult( 24, 17 ) ), 5).` and, evaluating further, `mult(plus(7, 408 ), 5);` finally, `7 + 408 = 415,` and, `415 * 5 = 2075.`

The tree nature of this evaluation algorithm is described in section sec:expression-tree.

8.4.1 Prefix, infix, postfix

Recall section 6.4; here we repeat some of the material there.

Prefix, infix and postfix are three ways of writing applications of functions; application = call.
Prefix  Prefix is how we call functions in C++; thus, \( z = \text{plus}(x, y) \), where,

```cpp
int plus(int a, int b){
    return a + b;
}
```

Infix  Infix is how we use operators in C++; thus, \( z = x + y \); which has the same meaning as the prefix expression above;

Postfix  In postfix – so-called *reverse Polish* notation, after the Polish mathematician Jan Lukasiewicz – the operands are given first, then the operator; thus, \( z = + x y \);

We shall not delve into it here, but we just mention that if one required to develop an interactive *in-fix* (normal) calculator, the task is greatly simplified by converting first the *in-fix* to *post-fix* – via a so-called *shunting* algorithm, in which operators are pushed (shunted) onto the stack and held there until the operands are fetched.

### 8.5 Proving Termination of Recursive and Other Non-deterministic Algorithms

Most of this section is taken from (Budd 1997), (Sedgewick 1997).

Unlike the case of deterministic looping algorithms, e.g. \( \text{for(int i=1, fact=1; i<=n; i++)fact=fact*i;} \), where we know that the algorithm will perform at most \( n \) iterations, there is some challenge in proving that a non-deterministic iteration or recursive algorithm will actually terminate.

The common approach to proving the termination of such algorithms is to seek some property of the algorithm whose value indicates the progress towards termination. This value should satisfy *three* characteristics:

- The value is, or can be related to, an integer;
- It remains non-negative;
- It decreases steadily as the algorithm proceeds.

Thus, in *fact*, \( n \) is the immediate choice.

```cpp
private static int fact(int n){
    if(n == 0)return 1; // base
    else return n*fact(n-1); // recursion
}
```
We note, of course, the crucial importance of the base case in the termination of recursive algorithms — if you have no base case, you never terminate; in the case of fact the base case stops n ever going negative — if it did go negative, n would set off for $-\infty$ and the algorithm would never terminate.

Euclid’s algorithm is less obvious, but still its termination is easy to prove. All we need to do is assure ourselves that, in the recursion, else return $\gcd(n, m \% n)$; $m \% n$ is not only less than $n$, but also less than $m$.

The example below, puzzle, which, incidentally does nothing except demonstrate this point, is an example of an algorithm whose termination is impossible to prove. According to (Sedgewick 1997), it is known to terminate for all cases of 32-bit unsigned integers, it cannot be proved for all integers.

```cpp
int puzzle(int n) {
    cout << "puzzle " << n << endl;
    if (n == 1) return 1;
    else if (n % 2 == 0) return puzzle(n / 2);
    else return puzzle(3 * n + 1);
}
```

8.6 Divide-and-Conquer

We now come to algorithms which solve the problem by: (a) dividing the problem into two or more smaller parts, (b) solving the smaller problems, and, (c) then somehow combining the results of the smaller solutions to obtain the complete solution.

Quite often, owing to the strategy thus formulated, the solution is naturally expressed using recursion.

8.6.1 Maximum of an Array using Divide and Conquer

A classical example is finding the maximum of an array. The algorithm proceeds as:

- If the array is of length 1, the answer is the single element;
- Otherwise, divide the array in two, and find the maximum of those;
- Combine the results obtained from the two smaller arrays.
The coded version is shown in Figure 8.8; full code is in arrayMax1.cpp.

The progression of the algorithm is shown in Figure 8.9.

```cpp
int maxRec(vector<int>& a, int left, int right){
    if(left==right)return a[left];
    else{
        int mid= (left + right)/2;
        cout<< left<< "":"<< mid<< " ";
        int maxLeft= maxRec(a, left, mid);
        cout<< (mid+1)<< "":"<< right<< " ";
        int maxRight= maxRec(a, mid+1, right);
        cout<< endl;
        if(maxLeft> maxRight)return maxLeft;
        else return maxRight;
    }
}

int max(vector<int>& x){
    cout<< 0<< "":"<< x.size()-1<< endl;
    return maxRec(x, 0, x.size()-1);
}

int main(){
    srand(131131);
    int n= 11;
    vector<int> a;
    a.reserve(n);
    for(int i = 0; i< n; ++i){
        a.push_back(rand()%100);
    }
    copy(a.begin(), a.end(), ostream_iterator<int>(cout, " "));
    int amax= max(a);
    cout<< "max= "<< amax<< endl;
}
```

Figure 8.8: Maximum of array using divide-and-conquer

The straightforward linear-search version of max takes $n - 1$ comparisons, i.e $O(n)$, the divide-and-conquer version takes the same.
35 54 93 61 61 51 75 27 62 12 50 0:10
0:5 0:2 0:1 0:0 1:1
2:2
3:5 3:4 3:3 4:4
5:5
6:10 6:8 6:7 6:6 7:7
8:8
9:10 9:9 10:10

max= 93

Figure 8.9: Progression of arrayMax1.cpp
8.6.2 Merge Sort

[Repeated from section 4.4.]

Merge-sort does array sorting by divide-and-conquer. The algorithm proceeds in almost the same manner as maxRec:

- If the array is of length 1, the answer is the single element;
- Otherwise, divide the array in two, and sort those;
- Combine the results obtained from the two smaller arrays by merging, i.e. an interleaving the two arrays.

Note that merge-sort needs to create a workspace array the length of the array to be sorted. The code is given in Figure 8.10.

```cpp
void merge(vector<int>& a, vector<int>& work, uint l, uint m, uint r){
  uint i, j, k;
  for(i = m+1; i > l; i--) work[i-1] = a[i-1];
  for(j = m; j < r; j++) work[r+m-j] = a[j+1];
  for(k = l; k <= r; k++){
    if(work[i] < work[j]) a[k] = work[i++];
    else a[k] = work[j--];
  }
}

void recMergeSort(vector<int>& a, vector<int>& work, uint l, uint r){
  if(l == r) return;
  else{
    uint m = (l + r)/2;
    recMergeSort(a, work, l, m);
    recMergeSort(a, work, m+1, r);
    merge(a, work, l, m, r);
  }
}

void mergeSort(vector<int>& b){
  uint len = b.size();
  vector<int> work(len);
  recMergeSort(b, work, 0, len-1);
}
```

Figure 8.10: Merge sort — sorting using divide-and-conquer
8.6.3 Towers of Hanoi

Towers of Hanoi is the classical introduction to recursion and divide-and-conquer. The problem is: given a set of three towers (actually, more like spindles) with a number of disks with holes in the middle that allow them to be fitted onto the spindles. The problem is to move the complete stack of disks on position to the right using only the following movements: (a) only one disk may be shifted at a time; (b) no disk may be placed on top of a smaller one.

\[
\begin{array}{ccc}
\text{----} & \text{disk 1} & \text{---} \\
\text{---} & \text{---} & \text{---} \\
\text{====} & \text{====} & \text{====} (\text{base})
\end{array}
\]

Tower A B C

If we start off with \(n\) disks to move from A to B, the divide-and-conquer solution is as follows:

Procedure: move \(n\) from X to Y using Z (Z is used as an intermediate):

1. If \(n\) is 1, then move disk 1 from X to Y; (the base case)
2. Otherwise:
   (a) move \(n - 1\) from X to Z using Y;
   (b) move \(n - 1\) from Z to Y using X;
3. return;

Thus, after solving the first \(n - 1\) problem:

\[
\begin{array}{ccc}
\text{-----} & \text{-----} & \text{-----} \\
\text{----} & \text{---} & \text{---} \\
\text{====} & \text{====} & \text{====} (\text{base})
\end{array}
\]

Tower A B C

Figure 8.11 shows C++ code (complete program in Towers.cpp; Figure 8.12 shows how the algorithm proceeds.

When the monks in Tibet devised the Towers of Hanoi puzzle, it was posed for 40 disks, and it was reckoned that the world would end before it was solved. They were not far wrong — in the recursive solution given, \(2^n - 1\) steps are required for an \(n\) disk problem. In other words, the problem has running time \(O(2^n)\) — this is an example of the dreaded exponential growth rate.

Exercise. Given \(n = 40\), and that it takes one second to move a disk, how long will it take to solve the problem? Hint: You could start with \(n = 32\) and proceed from there. \(2^{32} \approx 4.2 \times 10^9\); there are approx. \(31.5 \times 10^6\) seconds in a year.
int ccount = 0;
void towers(int n, char from, char inter, char to){
    ++ccount;
    if(n==1)cout<< "disk 1 from " " from " to " to " endl;
    else {
        towers(n-1, from, to, inter);
        cout<< "disk " n " from " " from " to " to " endl;
        towers(n-1, inter, from, to);
    }
}

int main(int argc, char *argv[]){
    int n;
    if(argc < 2)n = 3;
    else n = atoi(argv[1]);
    towers(n, 'a', 'b', 'c');
    cout<< "number of operations = " ccount<< endl;
}

Figure 8.11: Recursive Towers-of-Hanoi

disk 1 from a to c
disk 2 from a to b
disk 1 from c to b
disk 3 from a to c
disk 1 from b to a
disk 2 from b to c
disk 1 from a to c

number of operations = 7

Figure 8.12: Recursive Towers-of-Hanoi — output
8.6.4 Drawing a Ruler

The code in Figure 8.13 shows how to construct a ruler (with marks of height varying according to whether at half, quarter etc.) using divide-and-conquer. Complete code is available as Ruler.cpp.

The tree nature of the algorithm is emphasised by giving in-order, pre-order and post-order versions; see Chapter 7. Traces of the algorithm are in Figures 8.14, 8.15, and 8.16.
void mark(int p, int h)
{
    cout << "mark at " << p << " ht:" << h << endl;
}

void ruleIn(int l, int r, int h)
{
    cout << "rule " << l << " " << r << " " << h << "\n" << endl;
    if(h==0) return;
    else
    {
        int m = (l+r)/2;
        ruleIn(l, m, h-1);
        mark(m, h);
        ruleIn(m, r, h-1);
    }
}

void rulePost(int l, int r, int h)
{
    cout << "rule " << l << " " << r << " " << h << "\n" << endl;
    if(h==0) return;
    else
    {
        int m = (l+r)/2;
        rulePost(l, m, h-1);
        rulePost(m, r, h-1);
        mark(m, h);
    }
}

void rulePre(int l, int r, int h)
{
    cout << "rule " << l << " " << r << " " << h << "\n" << endl;
    if(h==0) return;
    else
    {
        int m = (l+r)/2;
        mark(m, h);
        rulePre(l, m, h-1);
        rulePre(m, r, h-1);
    }
}

int main()
{
    cout << "--- Inorder ---" << endl;
    ruleIn(0, 8, 3);
    cout << "--- Postorder ---" << endl;
    rulePost(0, 8, 3);
    cout << "--- Preorder ---" << endl;
    rulePre(0, 8, 3);
}

Figure 8.13: Drawing a Ruler — recursive algorithms
--- Inorder ---
rule 0 8 <3>
rule 0 4 <2>
rule 0 2 <1>
rule 0 1 <0>
mark at 1 ht:1
rule 1 2 <0>
mark at 2 ht:2
rule 2 4 <1>
rule 2 3 <0>
mark at 3 ht:1
rule 3 4 <0>
mark at 4 ht:3
rule 4 8 <2>
rule 4 6 <1>
rule 4 5 <0>
mark at 5 ht:1
rule 5 6 <0>
mark at 6 ht:2
rule 6 8 <1>
rule 6 7 <0>
mark at 7 ht:1
rule 7 8 <0>

Figure 8.14: Ruler algorithm — inorder
--- Postorder ---
rule 0 8 <3>
rule 0 4 <2>
rule 0 2 <1>
rule 0 1 <0>
rule 1 2 <0>
mark at 1 ht:1
rule 2 4 <1>
rule 2 3 <0>
rule 3 4 <0>
mark at 3 ht:1
mark at 2 ht:2
rule 4 8 <2>
rule 4 6 <1>
rule 4 5 <0>
rule 5 6 <0>
mark at 5 ht:1
rule 6 8 <1>
rule 6 7 <0>
rule 7 8 <0>
mark at 7 ht:1
mark at 6 ht:2
mark at 4 ht:3

Figure 8.15: Ruler algorithm — postorder
--- Preorder ---
rule 0 8 <3>
mark at 4 ht:3
rule 0 4 <2>
mark at 2 ht:2
rule 0 2 <1>
mark at 1 ht:1
rule 0 1 <0>
rule 1 2 <0>
rule 2 4 <1>
mark at 3 ht:1
rule 2 3 <0>
rule 3 4 <0>
rule 4 8 <2>
mark at 6 ht:2
rule 4 6 <1>
mark at 5 ht:1
rule 4 5 <0>
rule 5 6 <0>
rule 6 8 <1>
mark at 7 ht:1
rule 6 7 <0>
rule 7 8 <0>

Figure 8.16: Ruler algorithm — preorder
8.7 Trees and Recursion

8.7.1 Recursive drawing of a ruler

The drawing-a-ruler recursive solution follows a remarkably similar pattern to the Towers of Hanoi. In the case of towers-of Hanoi, look at the disk-moved identifiers (1, 2, 1, 3, 1, 2, 1), and, in the case of the ruler problem, Figure 8.14 (inorder version), the lengths of the ticks drawn: (1, 2, 1, 3, 1, 2, 1); i.e. they are the same. Both can be considered to solve their problem by traversing a tree.

In the case of ruler drawing we have the tree in Figure 8.17, where the value at the node is the value of the parameter $h$ at the call:

```
3
/    \
2     2
/     /\  \
1     1 1
/    /  /\  \
0  0  0 0 0 0 0
```

Figure 8.17: Tree for ruler algorithm

8.7.2 Recursive maximum of an array

Let us now examine the performance of recursive maximum on an vector of length 8; granted, 8 does not show the full generality of the divide-and-conquer algorithm, but it does allow us to see the similarity with the ruler problem and the towers problem. See Figure 8.18.

```c++
int maxRec(vector<int>& a, int left, int right)
{
    if(left==right)return a[left];
    else{
        int mid= (left + right)/2;
        cout<< left<< "":"<< mid<< " ";
        int maxLeft= maxRec(a, left, mid);
        cout<< (mid+1)<< "":"<< right<< " ";
        int maxRight= maxRec(a, mid+1, right);
        cout<< endl;
        if(maxLeft> maxRight)return maxLeft;
        else return maxRight;
    }
}
```

Figure 8.18: Recursive maximum of array

Let us start with the array \{-9, 8, 4, 1, 7, 6, 5, 3\} (these numbers are kept to one digit only to allow easier fit in the subsequent graphics).
Figure 8.19: Sub-array lengths in recursive maximum.

In terms of sub-array lengths, the problem divides as shown in Figure 8.19.

Next, in terms of sub-array range, Figure 8.20.

Figure 8.20: Sub-array ranges in recursive maximum.

And, finally, in terms of the maximum value returned, Figure 8.21.

Figure 8.21: Recursive maximum, values returned.
8.7.3 Recursive evaluation of a prefix expression

A tree is also a very natural way of viewing an arithmetic expression.

Recall the prefix expression: $*+\{7, *[*(4, 6), +(8, 9)]}, 5\}$ in section 8.4. This can be expressed as the expression tree in Figure 8.22.

```
  *
 / \  
+ 5  
/ \  
7 *  
/ \  
+ *  
/ \ / \  
8 9 4 6
```

Figure 8.22: Expression tree.

8.8 Elimination of Recursion

In general, any recursive algorithm may be expressed iteratively — and vice-versa. However, for many problems, recursion provides elegant and natural and easy to understand solutions.

The problem is there may be a price to pay for recursion; as we have discussed before, a call to a subprogram requires the provision of a new stack-frame (environment); this is none the less true for recursive calls and ff the recursion is very deep — i.e. one or more leaves are very far from the root, then the memory consumed by the stack may become excessive.

One form of recursion that is easily (mechanically) transformed into iteration is emphtail-recursion. Tail recursion is when the last step in a subprogram is a recursive call to itself. In such a case, the recursion may be replaced by a loop, and the recursive call by some assignments, followed by a return to the beginning of the loop.

Figure 8.23 shows transformation to iteration of the GCD algorithm in Figure 8.4 — which, apart from the first call to gcd which merely swaps the arguments, is tail-recursive.

However, as we have seen, not all recursive algorithms are tail-recursive. In these cases, the transformation is more difficult, but may be done with the aid of a (software) stack. In (Penton 2003), when he covers graphs, and graph searching algorithms such as depth-first-search, and path-finding, Penton mentions that, though these can be done quite naturally using recursion, practical applications often use a stack implementation (where we mean a stack in the algorithm, not that provided by the hardware).
// tail-recursive
int gcd(int m, int n) {
    cout << "gcd " << m << " " << n << endl;
    if (n > m) return gcd(n, m);
    else if (n == 0) return m;
    else return gcd(n, m % n);
}

// transformed to iterative
int gcdNonRec(int m, int n) {
    while (true) {
        if (n > m) { // swap, instead of reversed call
            int temp = n; n = m; m = temp;
        }
        else if (n == 0) return m;
        else {
            int rem = m % n; m = n; n = rem;
        }
    }
}

Figure 8.23: Transforming GCD from tail-recursive to iterative.
Chapter 9

Trees Miscellany

9.1 Introduction

In Chapter 7 we covered binary trees. Here we mention two topics: (i) the implementation of $n$-ary trees and some applications; (ii) the specific application of game trees in the context of game theory.

9.2 n-ary Trees

As we said in Chapter 7, once you know how to implement a binary tree, it is relatively straightforward to implement an $n$-ary tree. You should now read the Chapter 11 of (Penton 2003) to see how general $n$-ary trees are implemented there; implementation of an n-ary tree class is discussed in page 338 onwards.

We defined a binary tree recursively as a collection of nodes:

- Either, an empty tree;
- Or, a root node which consists of:
  - a left (sub)tree; also called left child;
  - a right (sub)tree; also called right child;

Each of the latter is itself a binary tree.

Each node may contain some value (a key). A node whose sub-trees are empty is called a leaf.
Analogously, we define an (n-ary) tree.

- Either, an empty tree;
- Or, a root node which points to: zero or more trees (children). Each of the children is itself a tree.

Each node may contain some value (a key). A node whose sub-trees are empty is called a leaf.

The obvious implementation of an n-ary tree is to replace the Node representation from (binary tree), see Figure 7.2,

```c
T m_key; // value / key
Node *m_left, *m_right; // pointers to left and right sub-trees
```

to

```c
T m_key;
list <Node *> children; // zero or more children trees
```

Recall the robot example that we covered in Graphics 2; Figure 9.1; each treenode represents a moving body part and the overall robot body is a pointer to a node, i.e. it is a tree, because of the recursive nature of treenode. We had the torso as the overall root.

```c
typedef struct treenode{
  GLfloat m[16]; // modelview transformation for this node
  void (*f)(void); // drawing callback function
  struct treenode *sibling; /* pointer to sibling (of child); this can recurse, i.e. child, sibling form a list */
  struct treenode *child; // pointer to child
}treenode;
```

typedef treenode* t_ptr; // tree

Figure 9.1: Tree representation of a connected game structure (robot).

In Figure 9.1, the list of zero or more trees is represented by child (first child) and then a pointer to that latter child’s sibling — this forms a list, because that sibling has a pointer to its sibling, and so on, recursively. The end of the list is indicated by a NULL sibling pointer.

Note: although Figure 9.1 works, it is always better to develop proper abstract data types for trees and lists and to hide the internals in a class; having the internals open to view works but is a lot more difficult and error prone to use.

Structures like this are very important in computer graphics, computer games, and general virtual reality; the form the basis of scene graphs.

Now read the Chapter 11 of (Penton 2003).

For examinations, you should be able to describe (in outline) the implementation of a general n-ary tree, see above, and, again in outline, how you would implement such methods as:
• size (count the number of nodes);
• empty (boolean, is tree empty?);
• pre-order traversal;
• in-order traversal;
• post-order traversal.

9.3 Game Trees

Now we switch direction and discuss game trees. Game trees are a branch of game theory; game theory originated in 1944 in a book by John von Neumann and Oskar Morgenstern, The Theory of Games in Economic Behaviour. The particular strategy we discuss here, the minimax algorithm, was describe by von Neumann in a paper in 1928, On the Theory of Games of Strategy.

The minimax algorithm gives an optimum strategy for playing certain combinatorial games, such as we might wish to program into the artificial intelligence (AI) of a game agent.

We use two simple games to discuss game trees: (i) Nim as discussed in Chapter 15 of (Penton 2003), Penton calls it Rocks, and (ii) noughts-and-crosses (tic-tac-toe).

9.3.1 Nim

Here is a grossly simplified form of Nim.

The game starts with two piles of matchsticks, see Figure 9.2; there are two players and they take turns in removing one or more matches from one of the piles — as many as they desire, but from only one pile. The person who removes the last match loses.

\[
\begin{array}{c|c}
Pile 1 & Pile 2 \\
\hline
+-----------+\
| * | * | * |
|---|---|
+-----------+
\end{array}
\]

Figure 9.2: Simple Nim, starting state.
Figure 9.3 shows the three possible *game states* after the three possible *game moves* by Player 1:

1.1 Remove one stick from pile 1;
1.2 Remove two sticks from pile 1;
1.3 Remove one stick from pile 2.

\[ +----------+ +----------+ +----------+ \]
\[ | * | * | * | Player 1’s turn \]
\[ | | | | \]
\[ | | | | edges = moves \]
\[ +----------+ +----------+ +----------+ \]
\[ | * | | * | Player 2’s turn \]
\[ | | | | \]
\[ +----------+ +----------+ +----------+ \]
\[ 1.1 1.2 1.3 \]

Figure 9.3: Simple Nim, first two levels of the game tree.

**Nodes and Edges** Up to now we have neglected to use the term *edge*; *edge* will become very important when we reach *graphs*. In the tree shown in Figure 9.3, the *nodes* represent possible game states; *edges*, the connections between nodes, represent game moves.

In graphs, for example a network of computers, nodes are the computers, edges are the connections. In graphs, a node is often called *vertex* (plural *vertices*). Note: this use of vertex has little or no link to the use of vertex in computer graphics, where vertex means *point*.

A graph is very similar to a tree; the difference is that in a tree, there is only one *path* of edges linking one node to another; in a graph, there may be *loops*. You can travel to Dublin via Strabane (etc.) or via Sligo, or via Enniskillen, etc.

Back to Figure 9.3. We see the three possible game states after Player 1’s move. What can Player 2 do in each case, assuming he/she wants to win and he/she knows that Player 1 will do his/her best to win on the move after?

1.1.1 Left. Remove one stick from pile 1;
1.1.2 Left. Remove one stick from pile 2;
1.2.1 Middle. Remove one stick from pile 2; *loses*;
1.3.1 Right. Remove one stick from pile 1;
1.3.2 Right. Remove two sticks from pile 1; *loses*;

You can imagine the (game ending) consequences.

Figure 9.4 now shows the complete game tree.
Figure 9.4: Simple Nim, complete game tree.
9.3.2 Minimax Algorithm

We now discuss the minimax algorithm for computing the best move; see Figure 9.5.

We start at the leaf nodes (terminal game states) and score according to Player 1’s point of view; we call Player 1 Max, and Player 2 Min.

A loss for Max score 0. A loss for Min score 1. The score of an edge (branch) is given in brackets, e.g. (0), (1).

We then move up to the next level. It is Max’s turn; if the state is an end, the node (a leaf) gets a score corresponding to the result; otherwise we take the maximum of the scores of the children; in each case, the nodes have just one child, so computing the maximum is trivial.

Finally, we move up to the level just below the start. It is Min’s turn; if the state is an end, the node (a leaf) gets a score corresponding to the result (there are none of these); otherwise we take the minimum of the scores of the children; hence, we end up with 0 for the left move, 1 for the middle move, and 0 for the right.

This tells Max that he/she should take the middle move; if Max was to choose either the righthand or lefthand moves, in order to win from the resultant state, he/she has to depend on Min responding with some equally (very) silly move at Min’s next turn.

\[
\begin{array}{c|c|c|c|c}
\text{Max} & \text{Player 1's turn} \\
\hline
| * * | * | \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
\text{Min} & \text{Player 2's turn} \\
\hline
| * | * | \\
\hline
| * | * | \\
\hline
| * * | \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c}
\text{Max} & \text{Player 1 turn} \\
\hline
| * | * | \\
\hline
| * | * | \\
\hline
| * * | \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c}
\text{Player 2 loses} & \text{Player 2 loses} \\
\hline
(0) & (0) & (1) & (0) & (1) \\
\hline
\text{Player 1 loses} & \text{Player 1 loses} \\
\hline
\end{array}
\]

Figure 9.5: Simple Nim, minimax.
9.3.3 Recursive Minimax Algorithm

As described informally above, we proceed *bottom-up* and for that we require a view of the complete game tree.

Like most tree algorithms, the minimax algorithm yields to the neat recursive definition in Figure 9.6; from http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html.

```c
int Minimax(player, game) {
    if (game ended) return score; //A
    else if (max’s turn)
        return maximum of Minimax applied to all child nodes //B
    else /*Min’s turn*/
        return minimum of Minimax applied to all child nodes //C
}
```

Figure 9.6: Recursive minimax.

The reference given above discusses a method called *alpha-beta pruning* which allows the algorithm to avoid evaluating some branches — hence pruning. Although it is quite simple in concept, we do not cover *alpha-beta pruning* in this module; there is decent description of *alpha-beta pruning* in the website mentioned above and also in (Heineman et al. 2008).

It might help to think of taking the minimum of all child scores //C as evaluating what is the worst Min can do at his/her move (and that Min will look ahead and pick the best branch for him/her).

Likewise, taking maximum of all child scores //B as thinking what is the best Max can do at his/her move, including looking ahead picking the best branch for him/her).
9.3.4 Minimax Applied to Tic-tac-toe

Figure 9.7 shows minimax applied to a late stage of a noughts-and-crosses game; taken from http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html.

In noughts-and-crosses we have scores of +1 for a win for Max (x), −1 win for Min (o), and 0 for a draw.

\[
\begin{array}{c}
\text{o\ o\ x} \\
\hline
\text{o\ o\ x} \\
\hline
\text{o\ x\ x} \\
\hline
(-1) \ . \ (-1) . \ (0) .
\end{array}
\]

\[
\begin{array}{c}
\text{o\ o\ x} \\
\hline
\text{o\ o\ x} \\
\hline
\text{o\ x\ o} \\
\hline
(0) \ . \ (+1) . \ (-1) . \ (0).
\end{array}
\]

\[
\begin{array}{c}
\text{o\ x\ x} \\
\hline
\text{x\ x\ x} \\
\hline
\text{o\ x\ o} \\
\hline
(+1) . \ (0) . \ (0) . \ (+1).
\end{array}
\]

\[
\begin{array}{c}
\text{o\ o\ x} \\
\hline
\text{o\ o\ x} \\
\hline
\text{o\ x\ o} \\
\hline
\text{o\ o\ x} \\
\hline
\text{o\ o\ x} \ 
\end{array}
\]

\[
\begin{array}{c}
\text{x\ x\ x} \\
\hline
\text{x\ x\ o} \\
\hline
\text{o\ x\ o} \\
\hline
\text{o\ o\ x} \ 
\end{array}
\]

Figure 9.7: Tic-tac-toe game tree

Limitations of Minimax Minimax provides a good theoretical introduction to optimal game playing for certain sorts of games (and many problem solving applications); however, it’s applications are limited:

- Works only for games with a finite number of states;
- Games need to be able to be described using a game tree;
- Need to be zero-sum games: Min’s loss is Max’s gain, and vice versa;
- etc. see (Penton 2003).
Chapter 10

Simple Pathfinding

10.1 Introduction

This chapter introduces pathfinding using a simple example of pathfinding in a maze from (Budd 1997). This brings us to considerations of depth-first searching, breadth-first searching, and backtracking. In addition, we need to introduce the deque data structure.

10.2 Deque Data Structure

A deque, pronounced "deck" or "Dee-Queue", is so called because it is a double-ended queue. As such, a deque behaves like a combination of stack and queue; but not only that, it also has some of the behaviour of vector and to a lesser extent list. Thus, the following operations are relatively efficient (relatively because there may be slight performance deficits compared vector, stack, queue and list on their home turf, i.e. using their best matched operations):

- insertion, deletion and access at the front: push_front, pop_front, and front;
- insertion, deletion and access at the back: push_back, pop_back, and back;
- random access, i.e. c.at(index), c[index].

Quite often, e.g. in STL, stacks and queues are implemented using a deque.

Because our main use here of deque is as a hybrid stack/queue, and because we are already familiar with the operations of stack and queue, we will not dwell too much on the implementation

10.2.1 Implementation

The typical implementation of deque is as two vectors. Figure 10.1 shows part of the implementation and Figure 10.2 shows part of the implementation of deque::iterator.
# include <vector>
using std::vector;

template <class T> class deque {
public:
    typedef dequeIterator<T> iterator;
    typedef T value_type;

    // constructors
    deque () : vecOne(), vecTwo() { }
    deque (unsigned int sz, T & initial) : vecOne (sz/2, initial),
    vecTwo (sz - (sz / 2), initial) { }
    deque (deque<T> & d) : vecOne(d.vecOne), vecTwo(d.vecTwo) { }

    // operations
    T & operator [] (unsigned int);
    T & front ();
    T & back ();
    bool empty () { return vecOne.empty () && vecTwo.empty (); }
    iterator begin () { return iterator(this, 0); }
    iterator end () { return iterator(this, size ()); }
    void erase (iterator);
    void erase (iterator, iterator);
    void insert (iterator, T &);
    int size () { return vecOne.size () + vecTwo.size (); }
    void push_front (T & value) { vecOne.push_back(value); }
    void push_back (T & value) { vecTwo.push_back(value); }
    void pop_front ();
    void pop_back ();

protected:
    vector<T> vecOne;
    vector<T> vecTwo;
};

template <class T> T & deque<T>::front (){ 
    if (vecOne.empty ())
        return vecTwo.front ();
    else
        return vecOne.back ();
}

template <class T> void deque<T>::pop_front (){ 
    if (vecOne.empty ())
        vecTwo.erase(vecTwo.begin ());
    else
        vecOne.pop_back ();
}

Figure 10.1: Deque
template <class T> class dequeIterator {
    friend class deque<T>;
    typedef dequeIterator<T> iterator;

public:
    // constructors
    dequeIterator (deque<T> * d, int i) : theDeque(d), index(i) {
    }
    dequeIterator (dequeIterator<T> & d) : theDeque(d.theDeque), index(d.index) {
    }

    // iterator operations
    T & operator * () { return (*theDeque)[index]; }
    iterator & operator ++ (int) { ++index; return * this; }
    iterator operator ++ () { // prefix change
        iterator & operator -- (int) { --index; return * this; }
    iterator operator -- (); // postfix change
    bool operator == (iterator & r) {
        return theDeque == r.theDeque && index == r.index;
    }
    bool operator ¡ (iterator & r) {
        return theDeque == r.theDeque && index ¡ r.index;
    }
    T & operator [ ] (unsigned int i) {
        return (*theDeque) [index + i];
    }
    void operator = (iterator & r) {
        theDeque = r.theDeque; index = r.index;
    }
    iterator operator + (int i) {
        return iterator(theDeque, index + i);
    }
    iterator operator - (int i) {
        return iterator(theDeque, index - i);
    }

protected:
    deque<T> * theDeque;
    int index;
};

// postfix form of increment
template <class T> dequeIterator<T> dequeIterator<T>::operator ++ (){
    // clone, update, return clone
    dequeIterator<T> clone(theDeque, index);
    index++;
    return clone;
}

Figure 10.2: deque::iterator
10.2.2 Discussion of sDeque.h

First notice the implementation — two vectors:

```cpp
protected:
    vector<T> vecOne;
    vector<T> vecTwo;
```

Figure 10.3 shows the roles of the two vectors and their connection with the logical view of the deque as a doubly-ended container.

The — front — of the deque is the back of vecOne, while the — back — of the deque is the back of vecTwo; as we know, deletion at the back of a vector is very efficient (O(1)); likewise insertion, apart from the case where capacity is exceeded and reallocation is necessary. Hence insertion and deletion at either end of the deque is similarly efficient.

### deque logical view

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| a | b | c | d | e | f | g | h | i | j | k | l | m | n |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

### deque physical implementation

```
vecOne front
+---+---+---+---+---+---+---+---+
| h | g | f | e | d | c | b | a |    deque front
+---+---+---+---+---+---+---+---+

vecTwo front
+---+---+---+---+---+---+
| i | j | k | l | m | n |
+---+---+---+---+---+---+
```

Figure 10.3: Deque logical view, and implementation using two vectors

If you want help in understanding the implementation deque::iterator, refer back to the iterators for vector/Array in Chapter 2 or for List/list (doubly linked list) in Chapter 5.

That’s all we have to say about the implementation of deque.

10.3 Path-finding in a Maze — Depth-first and Breadth-first Searching

We make our introduction to pathfinding using Budd’s maze example. We introduce depth-first and breadth-first search and backtracking.
Figure 10.4 shows a simple maze; the left hand diagram shows the maze showing start cell (S), goal cell (G), and current cell, before solution (*); the right hand diagram shows the cell identification numbering scheme for later reference.

![Maze Diagram](image)

Figure 10.4: Left, maze showing start cell (S), goal cell (G), and current cell, before solution (*); right, cell numbering scheme.

### 10.3.1 Maze implementation

The maze is implemented as a two-dimensional arrangement of cells; each cell can have up to four barrier walls, north south, east and west, or none. The maze arrangement is read in from file, where each cell is represented by an integer code; the coding scheme is given in Figure 10.5. That is, the least significant four bits of the integer are used to code the presence of walls.

Because each cell has information on what neighbours are accessible (no walls) collision detection is not needed.

```plaintext
<table>
<thead>
<tr>
<th>bit</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>south</td>
<td>east</td>
<td>north</td>
<td>west</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 10.5: Maze cell coding

Thus, the maze in Figure 10.4 is described by the file in Figure 10.6.
Figure 10.6: Maze file — 'mazeone'

Cells are implemented as shown in Figure 10.7; the chief point to note is that the connectivity of a cell is given by a list of pointers to cells to which one can move when in that cell. Each cell has a visited marker — this is needed to ensure that the pathfinding search algorithms do not form infinite loops.

```cpp
class cell {
public:
    cell (int c, int n) : code(c), number(n), visited(false) { }
    void addNeighbor (cell * n) { neighbors.push_back(n); }
    bool visit (deque<cell *>&);
    string toString();

protected:
    int code;
    int number;
    bool visited;
    list <cell *> neighbors;
};
```

Figure 10.7: Cell implementation.

The maze is implemented as shown in Figure 10.8, which shows also the constructor which reads from a file; for a detailed explanation of the extraction of the accessible neighbours lists for each cell, see (Budd 1997).

Note: I have added variables and diagnostic code that are not needed for the solution of maze, but are merely used to provide sample output for these notes.

The maze is represented by `cell * start`; this cell and the others contain lists of accessible neighbours.

deque <cell *> path is a deque used in the solution.
class maze {
public:
    maze (istream &);
    void solveMaze ();
    void print();
    int getCode(int r, int c);
protected:
    int numRows;
    int numCols;
    vector<int> codes;
    cell * start;
    bool finished;
    deque <cell *> path;
};

// construct maze by reading from file
maze::maze (istream & infile){
    int counter = 1; cell * current = 0;

    infile >> numRows >> numCols;
    codes.reserve(numRows*numCols);
    // create vector for previous row
    cell * nothing = 0;
    vector <cell *> previousRow (numCols, nothing);

    // now read data values
    for (int r = 0; r < numRows; r++)
        for (int c = 0; c < numCols; c++) {
            int walls;
            infile >> walls;
            current = new cell(walls, counter++);
            codes.push_back(walls);
            // make north connections
            if ((r > 0) & ((walls & 0x04) == 0) &
                (previousRow[c]->code & 0x01) == 0) {
                current->addNeighbor (previousRow[c]);
                previousRow[c]->addNeighbor (current);
            }
            // make west connections
            if ((c > 0) & ((walls & 0x08) == 0) &
                (previousRow[c-1]->code & 0x02) == 0) {
                current->addNeighbor (previousRow[c-1]);
                previousRow[c-1]->addNeighbor (current);
            }
            previousRow[c] = current;
        }
    // most recently created cell is start of maze
    start = current; finished = false;
};
10.3.2 Depth-first pathfinding solution

We now examine the pathfinding solution of the maze in Figure 10.4; first, *depth-first search*. The relevant code is shown in Figure 10.9: `maze::solveMaze` and `cell::visit`.

```cpp
void maze::solveMaze () {
    start->visit (path);
    while (!finished && !path.empty ()) {
        cell *current = path.front ();
        path.pop_front ();
        finished = current->visit (path);
    }
    if (!finished)
        cout << "no solution found\n";
}

// visit cell, place neighbors into queue
// return true if solution is found
bool cell::visit (deque<cell *> &path) {
    if (visited) // already been here
        return false;
    visited = true; // mark as visited
    cout << "visiting cell " << number << endl;
    if (number == 1) {
        cout << "puzzle solved\n";
        return true;
    }
    // put neighbors into deque
    list<cell *>::iterator start, stop;
    start = neighbors.begin ();
    stop = neighbors.end ();
    for (; start != stop; ++start)
        if (!(*start)->visited)
            path.push_front (*start); // depth-first
            // path.push_back (*start); // breadth-first
    return false;
}
```

Figure 10.9: Maze pathfinding
Solution

1. We start at cell 25;

2. `solveMaze` immediately calls `cell::visit`, with cell 25 (start) as the argument cell;

3. Now all cell 25’s accessible neighbours are pushed onto the front of the deque; note that `start` is used as an iterator,

```
    start = neighbors.begin ();
    stop = neighbors.end ()
    for ( ; start != stop; ++start)
        if (! (*start)->visited)
            path.push_front (*start); // ------------
```

4. Cell 25 has just one accessible neighbour, namely 20 and it has not yet been `visited`; so 20 is pushed onto the front of the deque and the situation is as shown below, where the deque is shown as `< ... >`

   visiting cell 25   <20 >

5. We now return from `cell::visit` back to the while loop in `solveMaze`

```
    while ((! finished) && (! path.empty ())) {
        cell * current = path.front (); // 1
        path.pop_front (); // 2
        finished = current->visit (path); // 3
    }
```

6. `solveMaze` takes the front of the deque, // 1, (cell 20), removes it (pop_front // 2) and calls `cell::visit` // 3;

7. Cell 20 has accessible neighbours 19 and 15 and as neither has been `visited`, they are pushed onto the front to the deque; the situation is now:

   visiting cell 20   <19 15 >

8. (same step as 5. above)
   We return from `cell::visit` back to the while loop in `solveMaze`

```
    while ((! finished) && (! path.empty ())) {
        cell * current = path.front (); // 1
        path.pop_front (); // 2
        finished = current->visit (path); // 3
    }
```

9. (same as step 6.)
   `solveMaze` takes the front of the deque, // 1, (cell 19), removes it (pop_front // 2) and calls `cell::visit` // 3;
10. (same as step 7.)

Cell 19 has an accessible neighbour 24 and as it has not yet been visited it is pushed onto the front to the deque; the situation is now:

visiting cell 19 <24 15>

Note that because this is depth-first, cell 15 has been not yet been processed.

11. Steps 8., 9., and 10. are repeated until we arrive at the goal, or reach either of the situations:

(a) We have reached a dead end, and backtracking is required. i.e we are in a cell whose only accessible neighbours we have already visited — meaning that at this cell, we pushed no new cells onto the deque and the next cell on the deque is one that was inserted earlier — i.e. backtracking;

(b) Final dead end; the deque is empty, indicating that there is nowhere to backtrack to.
10.3.3 Backtracking

To describe backtracking, we need to go further into the solution. The trace of full solution to Figure 10.4 is shown in Figure 10.10 and the order of cell visits is shown in Figure 10.11 — the numbers in the top left corner.

visiting cell 25 <20 >
visiting cell 20 <19 15 >
visiting cell 19 <24 15 >
visiting cell 24 <23 15 >
visiting cell 23 <22 18 15 >
visiting cell 22 <21 17 18 15 >
visiting cell 21 <16 17 18 15 >
visiting cell 16 <17 18 15 >
visiting cell 17 <18 15 >
visiting cell 18 <13 15 >
visiting cell 13 <12 8 15 >
visiting cell 12 <11 8 15 >
visiting cell 11 <6 8 15 >
visiting cell 6 <1 8 15 >
visiting cell 1 puzzle solved

Figure 10.10: Trace of solution.

Figure 10.11: Indication of order of visits — numbers in the top left corner.

Steps 8., 9., and 10. keep being repeated in a depth-first manner (i.e. where possible we keep visiting a neighbour of a new cell) until we get to cell 16 (visit number 7) where we have the situation:

visiting cell 16 <17 18 15 >

Cell 16 had no accessible neighbours that has not already been visited (cell 21 is the only accessible one, and it has been visited), so backtracking to cell 17 occurs; note that the proper use of the deque means that we need take no specific decision to backtrack — it happens automatically.
So we backtrack to cell 17 which is another dead end, and we have the situation:

visiting cell 17  <18 15 >

We backtrack to cell 18 (visit 9) and we escape into the freedom of cell 13.
Eventually we reach the goal at cell 1; at this stage cells 8 and 8 are still in the deque, but we don’t need to visit them.

### 10.3.4 Absolute dead end

Figure 10.12 shows a maze where the goal just cannot be reached. Figure 10.13 shows a trace of the steps to reach the absolute dead end.

![Maze with no solution](image)

Figure 10.12: Maze with no solution.

visiting cell 35  <28 >
visiting cell 28  <21 >
visiting cell 21  <14 >
visiting cell 14  <13 >
visiting cell 13
no solution found

Figure 10.13: No solution.
10.3.5 Breadth-first pathfinding solution

Remarkably, the only change that is necessary to choose a breadth-first search is to replace comments on the lines in Figure 10.9 (in cell::visit).

```cpp
for ( ; start != stop; ++start)
    if (!(*start)->visited)
        // path.push_front(*start); // depth-first
        path.push_back(*start); // breadth-first
```

That is, new candidates (non-visited accessible neighbours of the current cell), are pushed onto the back of the deque; this means that they have to take their turn after current occupants of the deque. Thus, in breadth-first search, the deque acts as a proper queue, rather than as a stack in the depth-first search usage of it.

Figure 10.14 shows the breadth-first path taken to solve Figure 10.4. Figure 10.15 shows a trace of the steps to reach the solution.

![Figure 10.14: Breadth-first search.](image)

We will explore depth-first search and breadth-first search in more detail in a later chapter on Graphs.

In depth-first search we visit a cell, then its first neighbour, then that cell’s first neighbour, and so on, until we reach a dead end and have to backtrack; recall pre-order traversal of trees in Chapter 7.

In breadth-first search we visit a cell, then each of its neighbours in turn, and so on. There is no equivalent in traversal of trees.
visiting cell 25 <20 >
visiting cell 20 <15 19 >
visiting cell 15 <19 10 14 >
visiting cell 19 <10 14 24 >
visiting cell 10 <14 24 5 >
visiting cell 14 <24 5 >
visiting cell 24 <5 23 >
visiting cell 5 <23 4 >
visiting cell 23 <4 18 22 >
visiting cell 4 <18 22 3 9 >
visiting cell 18 <22 3 9 13 >
visiting cell 22 <3 9 13 17 21 >
visiting cell 3 <9 13 17 21 2 >
visiting cell 9 <13 17 21 2 8 >
visiting cell 13 <17 21 2 8 8 12 >
visiting cell 17 <21 2 8 8 12 >
visiting cell 21 <2 8 8 12 16 >
visiting cell 2 <8 8 12 16 7 >
visiting cell 8 <8 12 16 7 7 > <12 16 7 7 >
visiting cell 12 <16 7 7 11 >
visiting cell 16 <7 7 11 >
visiting cell 7 <7 11 > <11 >
visiting cell 11 <6 >
visiting cell 6 <1 >
visiting cell 1
puzzle solved

Figure 10.15: Breadth-first solution.
10.4 Graphs

Very rough introduction to graphs; much to come in a later chapter.

A graph is more general than a tree. The nodes (vertices) in a tree (binary tree or n-ary tree, it doesn’t matter) are connected such that if we view the edges (pointers) as directed paths, there is only one path from the root to any vertex. Trees are acyclic graphs.

A graph is a more general collection of vertices and edges: (i) there is no root; (ii) there may be more than one path from a vertex to another vertex.

A graph may be implemented as a list of vertices (nodes), together with some implementation of adjacency.

Adjacency means direct connectivity between vertices.

Adjacency is typically implemented either as:

- a matrix $m$ of dimension $\text{numberOfVertices} \times \text{numberOfVertices}$; a 1 in $m[i][j]$ indicates that vertex-$i$ is connected to vertex-$j$.
- an adjacency list for each vertex.

The neighbours list in the maze example above is an example of an adjacency list. Hence, without knowing it, we implemented the maze as a graph data structure.

Later we will find that most sensible pathfinding algorithms demand that we represent our possibilities using a graph.
Chapter 11

Graphs

11.1 Introduction

In Chapter 10 we introduced pathfinding using a tile-based (cell-based) maze. Although the term graph was not explicitly used in our maze solution, we can now describe the maze problem quite generally in terms of a graph.

A graph is a collection of nodes (also called vertices, but note that this use of vertex has little to do with the use of vertex in graphics, where a vertex is a point). Some nodes are connected to other nodes by edges (also called arcs).

Figure 11.1 shows an example of a graph. There are eight nodes (vertices): a, b, c, d, e, f, g, h. There are thirteen edges (arcs): a-b, a-c, a-d, a-e, a-h, b-e, c-g, c-f, d-f, e-f, e-g, f-g, h-a; this is if we count each directed edge; a-h is one edge and h-a another.

![Diagram of a graph](image)

Figure 11.1: Example of a graph.

Edges can be directed or non-directed; in the examples we will deal with, edges will always be directed; nodes a and h share two edges a-h and h-a.
We use the term *adjacency* to describe whether there is an edge between two nodes; if the edges are directed, we can have two *adjacencies* between any two edges. In Figure 11.1, c is adjacent to a; h is adjacent to a and a is adjacent to h.

You will notice similarities between *graphs* and *trees*, Chapter 7. Both have collections of nodes, the nodes are connected by edges.

But a *graph* is more general than a *tree*. The nodes in a tree (binary tree or n-ary tree, it doesn’t matter) are connected such that if we view the edges (pointers) as directed paths, there is only one path from the root to any vertex.

It is possible to construct a graph such that it is a tree; for example, in Figure 11.1, if we remove node h and retain only edges a-b, a-c, a-d, b-e, c-g, c-f, we have a tree — there is only one path from a (root) to any other node. In Figure 11.1 we can travel via directed edges from a to g via a number of paths.

Furthermore, in a tree, we have no cycles; in Figure 11.1, there is a cycle — we can travel from a to d to h— and back again to a.

Trees normally represent some sort of hierarchy, for example, a family tree, or a disk storage directory structure. In graphs, there needs to be no such hierarchy. The nodes in Figure 11.1 are *all equal*; we can think of *readability* between nodes; we can think of node a as the starting point of a graph traversal; but we could have started anywhere. Note, however, that if we start at, say, f, we are quite limited in what nodes we can reach.

In addition to possessing direction, edges can also have *weights* or *costs* which represent the *cost* of travelling along that edge — for example in the Internet, some connections may charge more than others.

### 11.2 Examples of Graphs

#### 11.2.1 General

- A network of computers; nodes = computers; edges = network connections. Internet *routing algorithms* involve finding a path between two computers — normally minimising the *cost* (e.g. sum of weights of the edges used in the path).

- A network of towns (nodes) and roads (edges). Here edge weights could represent *distance*. Google or the AA have software to determine a *minimum distance* path between two towns.

- Etc.

#### 11.2.2 Games and Path-finding

- In a tile game: tiles = nodes, edges = readability from one tile to its neighbours; recall the maze of Chapter 10.
• In a 3D FPS, rooms/spaces = nodes, edges = readability between one and the other. Here we could have directed edges — if there is a locked door between nodes i and j, with the key in i’s side, then we have an edge between i and j, but not the other way round. In this case weights could represent difficulty of a path — across tarmac, low weight, across a swamp, high weight.

• In a 3D FPS, portal engines; here the graph represents visibility. If we are at node x and the region represented by node y is not visible (no arc) from x, then, while the camera is in region a, there is no need to render anything in region y. See portal engines in Penton’s book.

11.3 Implementation

A graph may be implemented as a list or array of nodes, together with some implementation of adjacency.

Adjacency is typically implemented either as:

• an adjacency matrix m of dimension numberOfNodes × numberOfNodes; a 1 in $m[i][j]$ indicates that node-i is connected to node-j. Because there are two entries, $m[i][j]$ and $m[j][i]$, this allows for directed edges. We can have boolean values (no weights) or numerical value (weights).

• an adjacency list for node. Each node has a list of edges. Each edge in the list will, in general, have (i) a pointer to, or some other indicator of the node it points to, (ii) a weight.

The graph software we use from Penton uses an adjacency list.

Adjacency list make more efficient use of memory when the number of connections is sparse, i.e. when a connection between nodes i and j is the exception rather than the rule.

If the nodes are densely connected or if there are just a small number of nodes, then an adjacency matrix may represent a simpler and/or more efficient solution.

11.4 Graph Traversal

By graph traversal, we mean starting at some node, e.g. a in Figure 11.1 and visiting, once, as many nodes as are reachable from the starting point.

The two traversal algorithms we use here are: (i) depth first, and breadth first; we have already encountered both of these in Chapter 10 (pathfinding in the maze). Note: depth first is the equivalent of pre-order traversal of trees.
11.4.1 Depth-first Traversal

Figure 11.2 shows the result of, starting at a, depth-first traversal of the graph in Figure 11.1. The numbers show the visit order.

Node a is the starting point (0), then b is visited, then c, then g; g is a dead end, so it backtracks to e to see if there is anything to visit from there; there is, and f is visited.

g is in f’s adjacent list, but it is marked as visited. Backtrack to e — nothing left to visit; backtrack to b — nothing left to visit.

Now backtrack back to a and visit c. But at c, the two nodes in its adjacency list are marked as visited.

Backtrack back to a; visit d; the visit h.

This traversal is called depth-first because, at any node, we pick the first (*) node in its adjacency list and visit that; that is repeated (recursively) until we have to backtrack.

(*) In the examples first and the general order of the adjacency list is merely a result of the order in which edges were added.
11.4.2 Breadth-first Traversal

Figure 11.3 shows the corresponding breadth-first traversal.

![Breadth-first graph traversal](image)

We see that, starting at a, breadth-first visits those nodes that are immediately connected to a, thus: b, 1, e, 2, c, 3, d, 4, and h, 5.

Then we go to b and attempt to repeat the exercise (recursively). b has no neighbours that are not marked as visited.

Then e, which has g and f as neighbours; g is visited, then f.

Then c, but the nodes in its adjacency list are already marked as visited.

Then d, but the nodes in its adjacency list are also already marked as visited.

Then h, but the only node in its adjacency list, a has been marked.

This traversal is called breadth-first because, at any node, we first visit the nodes in its adjacency list in order; we then proceed to the first node and apply breadth-first to that (recursively).
11.5  Software Implementation

We use the graph implementation, Graph.h, and the demonstration program, GraphDemo.cpp from Penton’s Chapter 17. But note that I have made a significant number of modifications to both files. Notable modifications: (i) refactoring the code using std::list, std::queue, std::vector, instead of Penton’s versions of these; (ii) removal of unnecessary comments and white space; (iii) labelling of nodes and visit order in GD17-024.cpp.

11.5.1  Basics of Graph.h

We now show parts of the graph class. Figure 11.4 shows the node (GraphNode); GraphNode’s data members comprise:

```cpp
NodeType m_data; // data (in a tile-game, coordinates (position
// to draw the node at); necessary only
// for the graphical demonstration
list<Arc> m_arcList; // adjacency list
bool m_marked; // 'visited' mark for depth-first and
// breadth-first traversal
```
template<class NodeType, class ArcType>
class GraphNode{
public:

typedef GraphArc<NodeType, ArcType> Arc;
typedef GraphNode<NodeType, ArcType> Node;

NodeType m_data;
list<Arc> m_arcList;
bool m_marked;

// Description: This finds the arc in the current node that
// points to the node in the parameter.
// Arguments: p_node: the node that the arc connects to.
// Return Value: a pointer to the arc, or 0 if an arc doesn’t
// exist from this to p_node.
Arc* GetArc( Node* p_node ){
    typename list<Arc>::iterator itr = m_arcList.begin();
    typename list<Arc>::iterator end = m_arcList.end();
    for( ; itr!= end; ++itr){
        if( (*itr).m_node == p_node ) return &(*itr);
    }
    return 0;  }

// Description: This adds an arc from the current node pointing
// to p_node, with p_weight as the weight.
// Arguments: p_node: the node to connect the arc to.
// p_weight: the weight of the arc.
void AddArc( Node* p_node, ArcType p_weight ){
    Arc a;
    a.m_node = p_node;
    a.m_weight = p_weight;
    m_arcList.push_back( a );
}

// Description: This finds an arc from this node to p_node and
// removes it.
void RemoveArc( Node* p_node ){  
    typename list<Arc>::iterator itr = m_arcList.begin();
    typename list<Arc>::iterator end = m_arcList.end();
    for( ; itr!= end; ++itr){
        if( (*itr).m_node == p_node ){
            m_arcList.erase( itr );
            return;
        }
    }
}
};
Figure 11.5 shows the arc (edge) (GraphArc). GraphArc’s data members comprise:

```
GraphNode<NodeType, ArcType>* m_node; // node edge points TO
ArcType m_weight;                      // weight (sometimes not used)
```

template<class NodeType, class ArcType>
class GraphArc {
public:
  // Description: pointer to the node that the arc points to
  GraphNode<NodeType, ArcType>* m_node;
  ArcType m_weight;
};

Figure 11.5: Graph Arc (edge) — part of Graph.h
Figure 11.6 shows part of the graph itself. Graph’s data members comprise:

```cpp
vector<Node*> m_nodes; // array of (all) nodes making up the graph
int m_count; // number of nodes
```

The implementation here is a bit kludgy; an array (vector) of a certain maximum number of pointers-to-node is allocated and initialised with null-pointers. If a new nod is to be added, we search for the first null item in the array, create a Node and insert it and update the count. This could probably be done more elegantly by using push_back on a growing vector… maybe we leave that as an exercise for the student :)

```cpp
template<class NodeType, class ArcType>
class Graph {
 public:
  typedef GraphArc<NodeType, ArcType> Arc;
  typedef GraphNode<NodeType, ArcType> Node;

  vector<Node*> m_nodes;
  int m_count;

  Graph( int p_size ) : m_nodes( p_size ){
    int i;
    for( i = 0; i < p_size; i++ ) m_nodes[i] = 0;
    m_count = 0;
  }

  ~Graph(){
    for(int index = 0; index < m_nodes.size(); index++ ){
      if( m_nodes[index] != 0 ) delete m_nodes[index];
    }
  }

  // Description: This adds a node at a given index in the graph.
  // Arguments: p_data: the data to store in the node.
  // p_index: index to store the node.
  // Return Value: true if successful
  bool AddNode( NodeType p_data, int p_index ){
    // find out if a node already exists at that index.
    if( m_nodes[p_index] != 0 ) return false;
    m_nodes[p_index] = new Node;
    m_nodes[p_index]->m_data = p_data;
    m_nodes[p_index]->m_marked = false;
    m_count++;
    return true;
  }
```

Figure 11.6: Graph, part of, (Graph.h)
11.5.2 Depth-first Traversal

Figure 11.7 shows the Graph.h method that implements depth-first traversal.

```c
// Description: Performs a depth-first traversal on p_node
// Arguments: p_node: the starting node
// p_process: the processing function.
void DepthFirst( Node* p_node, void (*p_process)(Node*) ){
    if( p_node == 0 ) return;

    p_process( p_node );
    p_node->m_marked = true;
    // iterate through each connected node
    typename list<Arc>::iterator itr =
        p_node->m.arcList.begin();
    typename list<Arc>::iterator end =
        p_node->m.arcList.end();
    for( ; itr!= end; ++itr ){
        // process the linked node if it isn't already marked.
        if( (*itr).m_node->m_marked == false )
            DepthFirst( (*itr).m_node, p_process );
    }
    return;
}
```

In GraphDemo.cpp, DepthFirst is called as follows, where g_graph is the graph and g_current is the starting node.

```c
  g_graph.DepthFirst( g_current, NodeProcess );
```

NodeProcess is a callback function:

```c
void NodeProcess( GraphNode<Coordinates, int>* p_node ){
  g_queue.push( p_node );
}
```

All that NodeProcess does is push the node onto a globally defined queue called g_queue. Please note that this is not a significant part of the traversal; it is used simply to record the nodes as they are visited, so that a (separate) animated playback can be displayed as described in subsection 11.5.4.

Yes, a queue is used in breadth-first traversal (but not depth-first), so please note that g_queue is not part of the traversal algorithm in either case.
11.5.3 Use of Stack for Depth-first?

In Chapter 10, we used a deque, operating in stack mode, to implement the depth-first search of the maze. If you need to, go back and see this now.

Here no stack. Why not? The answer is that the recursive call to DepthFirst provides and implicit stack — using the C++ language’s subprogram stack.

As Penton points out, the implementation in DepthFirst is nice and simple and he left it that way. However, he points out that, in a game, you would probably want to use an explicit stack as there may be worries about depth of recursion and the amount of stack memory that this would consume.

Figure 11.8 shows an implementation of depth-first traversal using an explicit stack.

```c
// Description: Performs a depth-first traversal on p_node
// j.g.c. 2008-11-17, explicit stack version
void DepthFirstS( Node* p_node, void (*p_process)(Node*) ){
    if( p_node == 0 ) return;
    stack<Node*> s;
    s.push(p_node);
    p_node->m_marked = true;

    while(!s.empty()){
        p_process(s.top());
        // push all of unmarked child nodes onto the stack
        typename list<Arc>::iterator itr = s.top()->m_arcList.begin();
        typename list<Arc>::iterator end = s.top()->m_arcList.end();
        s.pop();
        for( ; itr!= end; ++itr){
            if(!(*itr).m_node->m_marked){
                (*itr).m_node->m_marked = true;
                s.push( (*itr).m_node );
            }
        }
    }
    return;
}
```

Figure 11.8: Depth-first Traversal using explicit stack (Graph.h)
11.5.4 Animated Display of the Traversal

Figure 11.9 shows the part of GraphDemp.cpp which provides an animated display of the traversal. But note that this animation is provided after the traversal has been completed, using the recording of nodes that was placed in the g_queue queue.

```c++
if( g_state == DEPTHFIRST || g_state == BREADTHFIRST ){
    g_current = g_queue.front();
    if( SDL_GetTicks() - g_timer >= 700 ){
        g_number++;
        g_queue.pop();
        if( !g_queue.empty() ){
            g_current = g_queue.front();
            g_current->m_data.data = g_number;
        } else g_state = NOSTATE;
        g_timer = SDL_GetTicks();
    }
}
```

Figure 11.9: Display of Graph Traversal using SDL
11.5.5 Breadth-first Traversal

Figure 11.10 shows the Graph.h method that implements breadth-first traversal.

```cpp
void BreadthFirst( Node* p_node, void (*process)(Node*) ){
    if( p_node == 0 ) return;

    queue<Node*> queue;

    // place the first node on the queue, and mark it.
    queue.push( p_node );
    p_node->m_marked = true;

    // loop through the queue while there are nodes in it.
    while(!queue.empty()){
        // process the node at the front of the queue.
        process( queue.front() );
        // add all of the child nodes that have not been
        // marked into the queue
        typename list<Arc>::iterator itr =
            queue.front()->m_arcList.begin();
        typename list<Arc>::iterator end =
            queue.front()->m_arcList.end();
        for( ; itr!= end; ++itr){
            if( (*itr).m_node->m_marked == false ){
                // mark the node and add it to the queue.
                (*itr).m_node->m_marked = true;
                queue.push( (*itr).m_node );
            }
        }
        // dequeue the current node.
        queue.pop();
    }
}
```

Figure 11.10: Breadth-first Traversal (Graph.h)

In GraphDemo.cpp, BreadthFirst is called as DepthFirst (above), where g_graph is the graph and g_current is the starting node.

```cpp
g_graph.BreadthFirst( g_current, NodeProcess );
```

As before, NodeProcess is a callback function that records the traversal and is not part of the algorithm.

In Figure 11.10, we see explicit use of the queue queue<Node*> queue.
11.5.6 Depth-first and Breadth-first, a Summary

The breadth-first traversal algorithm may be summarised as follows.

BreadthFirst(Node start){
  Queue q;

  mark start as visited;
  push start onto the (back of) q;

  while(q not empty){
    Node current = node from front of q; pop it off q;
    Process current;
    Iterate through all nodes adjacent to current{
      mark the node;
      push it onto q;
    }
  }
}

In contrast the depth-first traversal algorithm may be summarised as follows.

DepthFirst(Node start){
  Stack s;

  mark start as visited;
  push start onto the (top of) s;

  while(s not empty){
    Node current = node from top of s; pop it off s;
    Process current;
    Iterate through all nodes adjacent to current{
      mark the node;
      push it onto s;
    }
  }
}

The significance of the breadth-first and depth-first names may be made more obvious by looking at Figure 11.11 (breadth-first traversal) and Figure 11.12 (depth-first traversal).

In Figure 11.11, the search proceeds along the breadth of the graph, visiting nodes one hop away; only when these are all visited, does it move to nodes two hops away; and so on.

In Figure 11.12, the search proceeds into the depth the graph, visiting the first node one hop away, then one hop away from that, and so on until it reaches a dead end; it then backtracks and explores starting with the second neighbour of the start; and so on.

As drawn in Figure 11.11, breadth-first might be considered to be row-first; and in Figure 11.12, depth-first might be considered to be column-first.
Figure 11.11: Breadth-first traversal

Figure 11.12: Depth-first traversal
Path-finding  In pathfinding, just as in Internet routing, we want to find the least-cost (shortest) route from a starting node to a goal node.

This involves a search, via traversal, of possible paths between start and goal and reporting the shortest (including the list of nodes to visit).

We study pathfinding algorithms in Chapter 12, where we find that breadth-first traversal (breadth-first search) is the basis of the best known algorithms.
Chapter 12

Pathfinding

12.1 Introduction

Already, in Chapter 10, we have seen an example of pathfinding in a tile-based maze. In that example, we saw solutions based on (a) depth-first search, and (b) breadth-first search. At the end of that Chapter, we noted that we were using barely disguised graph-searching methods.

Then in Chapter 11 we looked at graphs and two graph-traversal algorithms: depth-first and breadth-first.

In this Chapter, we take a more detailed look at pathfinding. Two familiar books will be invaluable in helping you understand this topics: Penton (2003, Chapter 23) and Brackeen et al. (2004, Chapter 12). Russell & Norvig (2003, Chapters 3, 4) contains a very complete and accessible coverage of general graph-search methods in artificial intelligence (AI).

We now look at pathfinding in a little more detail.

12.2 Pathfinding as a graph search problem

Figure 12.1 shows a pathfinding problem on a tile-based map. The current position, g, is marked Start and we need to get to Goal. (Only a few tiles are labelled.) There are no obstacles; handling of obstacles will be covered later.

Figures 12.2 and 12.3 show how this problem can be interpreted as a graph search problem.

Figure 12.2(a) shows the cells that are one move away from g. Now assume that we have probed cells h, m, l, k, f, a, b, c and that we have chosen m as the next base for further exploration. Figure 12.2(b) now shows the cells that are one move away from m. (The cells that have already been visited are marked with * — only g in this case.)

Figure 12.3 shows (some of) the cells expressed as a graph; the numbers give an indication of the order of visits in a breadth-first traversal. cells are graph nodes and edges indicate neighbour readability. Since we have no obstacles, all eight neighbours of any cell are reachable. Later we will see how to implement barriers (impassable cells) and path cost (difficulty) using edge weights.
Figure 12.1: Pathfinding on a tile-based map; current position is cell g (Start); the goal is marked Goal.

Figure 12.2: Breadth-first pathfinding in a tile game; (a) starting at cell g, cells h, m, l, k, f, a, b, c are adjacent (reachable neighbours); (b) then from cell m, cells n, s, r, q, l, g, h, i, n are adjacent; * signifies already visited cells.
Because of the simplicity of the situation, all edges are bidirectional; or, recalling Chapter 11, between any two cells, \( i \) and \( j \), we have two edges: an edge from node \( i \) to node \( j \) and another from \( j \) to \( i \).

![Graph representation of some of the cells (nodes) in the tile map; all edges are bidirectional; not all edges are shown. The numbers give an indication of the order of probing in a breadth-first traversal / search.](image)

**Figure 12.3:** Graph representation of some of the cells (nodes) in the tile map; all edges are bidirectional; not all edges are shown. The numbers give an indication of the order of probing in a breadth-first traversal / search.

The search pattern shown in Figures 12.2 and 12.3 is our old friend *breadth-first search.*
12.3 Graph search — informed or uninformed?

Graph search algorithms may be categorised according to whether they use (a) an uninformed strategy (blind search) or (b) an informed strategy through use of a heuristic.

**Uninformed (blind) search**  Given a problem like that shown in Figure 12.1, if we assume that we have no information about the Goal other than that we will know when we arrive at it, a uninformed graph search is the best we can do.

**Informed (heuristic) search**  Given a problem like that shown in Figure 12.1, if we now assume that our player can somehow sense the Goal and, for all possible choices of next cell (next node) to explore, can somehow provide an estimate of the distance from that current cell to the Goal then the search can proceed a lot more intelligently.

12.4 Graph search pathfinding

12.4.1 Uninformed graph search

Here we assume that we have no information about the Goal other than that we will know when we arrive at it.

In the implementation, each cell should contain a pointer to the cell from which it was entered, prev, and, if we want to determine a minimum distance path, the distance from the start.

12.4.2 Breadth-first pathfinding — basic algorithm

The basic algorithm is given in Figure 12.4.

In other texts you will notice that our Queue open; is replaced by something called an Open List and that marking as visited is replaced by adding the node to a Closed List.

Incidentally, you should read Penton (2003, pp. 735–736) to see why a cell/node can be on the queue, yet have been marked as already visited.

If you look at Figure 12.2(a), you will see that once g has been taken from the queue open, its neighbours will be pushed onto open which will then contain

\[ f[m l k f a b c h]b \]

(we assume that is the order that was chosen — for the plain breadth-first algorithm the order is arbitrary).

Next m is taken from open (and marked as visited); then m’s neighbours are pushed onto open, so that it will now contain

\[ f[l k f a b c h s r q 1+ g* h+ i n]b. \]

That is, 1, h are duplicated entries (marked with +) and, of course, g is marked as visited (marked with *).
Function PathFindBF(Graph g, GraphNode start, GraphNode goal){
    GraphNode curr;
    Queue open; // this is the OpenList
    float cost; // distance

    start.cost = 0;
    open.push(start);
    while(not open.empty){
        current = open.front; open.pop;
        if(curr == goal) return success;
        if(curr not already visited){
            mark curr as visited; // same as adding to ClosedList
            for(GraphNode adj from all GraphNodes adjacent to curr){
                if(adj not already visited){
                    adj.prev = curr; // to keep track of path to adj
                    cost = curr.cost + additional cost to get from curr to adj
                    if(adj.prev != null){ // cost to it already calculated
                        adj.cost = Min(adj.cost, cost);
                    }
                    open.push(adj);
                }
            }
        }
    }
    return failure; //q is empty, nowhere else to search;
} //end PathFindBF

Figure 12.4: Breadth-first pathfinding — basic algorithm
12.4.3 Informed graph search — add a heuristic

We can add a heuristic to the basic algorithm with some simple modifications; the modified algorithm is given in Figure 12.5, where the only modifications are marked with an asterisk, *.

Function PathFindHeur(Graph g, GraphNode start, GraphNode goal){
    GraphNode curr;
    PriorityQueue open; //* // this is the OpenList
    // the rest is the same
    // except that in programs you will notice that you must use
    // top instead of front to access the 'front' of a
    // std::priority_queue
}

Figure 12.5: Graph search pathfinding — with heuristic

In this version, when a node is added to the OpenList it takes up a position not necessarily at the back, but in priority queue order according to the node’s heuristic value. Example, if open contains nodes with heuristic values f[5 4 3 3 3 2 1]b and a new node is added with heuristic value 3, it will take up its place as marked by the *: f[5 4 3 3 3* 2 1]b.

This means that, although the algorithm is firmly based on breadth-first, heuristic-based preference can take over and the search evolve very differently from breadth-first.

12.4.4 Depth-first pathfinding

As we have seen in Chapters 10 and 11, the breadth-first algorithm can be changed to depth-first by changing OpenList to be a stack (FIFO) instead of a queue (LIFO); the depth-first algorithm is given in Figure 12.6, where the only modifications are marked with an asterisk, *.

I have added a depth-first pathfinder to the Penton’s program; when you run it you will see that it is not particularly suited to pathfinding in that context.
Function PathFindDF(Graph g, GraphNode start, GraphNode goal) {
    GraphNode curr;
    Stack open;  // this is the OpenList
    float cost;  // distance

    start.cost = 0;
    open.push(start);
    while (not open.empty) {
        current = open.top; open.pop;  // now stack

        // the remainder is the same.
    }
} // end PathFindDF

Figure 12.6: Depth-first pathfinding — basic algorithm
12.5 Practical examples of graph-search algorithms

We’re now going to use an adapted version of the pathfinding program from Penton (2003, Chapter 23) to test our pathfinding algorithms. We start by showing the test case — Figure 12.7.

Figure 12.7: Pathfinding test case. X is the goal; the 'little guy' is the starting point. The black wall facing the goal is impassable; the weights of the other coloured cells are describe in the text.

In Figure 12.7, X is the goal; the little guy is the starting point. The black wall facing the goal is impassable; the weights of the other coloured cells are given Figure 12.8. We have modified the program to have it interpret 9 as an impassable barrier.

```
  2
 22
333   999
44444  9
66666  9
67776  9
S 67776  9 X
67776  9
66666  9
44444  9
333333  9
222222  9999
```

Figure 12.8: Weights in test case; 9 is taken as impassable.
12.5.1 Silly Breadth-first

First we apply what we term the *silly breadth-first* pathfinding algorithm; this algorithm applies a blind breadth-first search and ignores weights entirely, both in its choice of search visits and in its calculation of distance; it calculates distance as number of cells passed through. We show this algorithm merely to emphasise the link with breadth-first. Figure 12.9 shows the search in progress and Figure 12.10 shows the completed search and chosen path.

Like all the examples here, the performance of the algorithm is much more evident when you watch the program running.

![Silly breadth-first pathfinding — in progress.](image1)

![Silly breadth-first pathfinding — completed.](image2)
12.5.2 Breadth-first

Next we apply "plain" breadth-first pathfinding; this version of breadth-first algorithm is still blind, but takes account of weights in its calculation of distance and of the fact that a diagonal step is \( \sqrt{2} = 1.414 \) times longer than a horizontal or vertical step. On the other hand, it ignores this distance differential when choosing the order of cell to visit when it moves one cell deeper.

Figure 12.11 shows the search in progress and Figure 12.12 shows the completed search and chosen path.
12.5.3 Distance-first

Next we add a heuristic to breadth-first pathfinding; this version is still blind (uninformed is the terms used in the AI literature), but it takes account of the diagonal distance factor ($\sqrt{2} = 1.414$) as a heuristic when choosing the order of cells to probe when it moves one cell deeper. However, this heuristic has effect only on the order of probing at the next level — the algorithm still retains an essentially breadth-first character and cells actually cannot jump the queue as they can with other heuristics — the next-level cells with 1.414 are always closer than cells at the next-again-level cells (2 and 2.828).

Penton (2003) calls this algorithm distance-first. Figure 12.13 shows the search in progress and Figure 12.14 shows the completed search and chosen path.

![Distance-first pathfinding — in progress.](image)

![Distance-first pathfinding — completed.](image)
12.5.4 'Simple' Heuristic

This algorithm uses a heuristic which favours cells that get us closer to the goal. Figure 12.15 shows how it works. The probed cells (those with numbers in them) are scored according to how much closer (negative) or further away (positive), or just the same (0). (It computes the heuristic for both x and y axes (-1 for closer, 0 for the same, + 1 for further away) and adds them.)

![Figure 12.15: Simple heuristic.](image)

Figure 12.16 shows the search in progress and Figure 12.17 shows the completed search and chosen path.

As you can see, (i) the search is pretty well focussed on the goal; (ii) when it reaches a barrier, it can relatively quickly dispense with an unfavourable search avenue.

On the other hand, Figure 12.18 shows a case in which it can return a path which is obviously not the shortest.

The problem was that once the cells that got us closer to the goal were exhausted, it had to take off in some other direction; as it turns out, the horizontal path which eventually runs out of space at the left hand edge of the map did in fact seem best to the algorithm — that search avenue’s cells are all zero distance along the y-axis; it is not until these cells are marked, that the algorithm has to try some other cell — further away in x and y, but from that cell a successful new search can be launched.
Figure 12.16: Simple heuristic pathfinding — in progress.

Figure 12.17: Simple heuristic pathfinding — completed.

Figure 12.18: Simple heuristic pathfinding — shortest path not found.
12.5.5 Distance Heuristic

This version bases its heuristic on the distance from the cell to the goal. This means that, unlike the previous simple heuristic pathfinder, it cannot be lured down an increasingly distance search avenue.

Figure 12.19 shows the search in progress and Figure 12.20 shows the completed search and chosen path.

Figure 12.21 shows that it handles the previously difficult case properly.

Figure 12.19: Distance heuristic pathfinding — in progress.

Figure 12.20: Distance heuristic pathfinding — completed.
Figure 12.21: Distance heuristic pathfinding – difficult case, shortest path found.
12.5.6 AStar Heuristic

The A* heuristic is a small but significant variation on the so-called distance heuristic: it adds the current distance to the probed-cell-to-goal distance.

Figure 12.22 shows the search in progress and Figure 12.23 shows the completed search and chosen path.

Figure 12.22: A* heuristic pathfinding — in progress.

Figure 12.23: A* heuristic pathfinding — completed.
12.6 Algorithms Applied to a Maze problem

Just for interest, we apply all five algorithms to a maze problem similar to that described in Chapter 10.

Figure 12.24: Plain breadth-first pathfinding — maze.

Figure 12.25: Distance-first pathfinding — maze.
Figure 12.26: Simple heuristic pathfinding — maze.

Figure 12.27: Distance heuristic pathfinding — maze.

Figure 12.28: A* heuristic pathfinding — maze.
12.7 Performance Measures

Russell & Norvig (2003, p. 71) defines four criteria by which search algorithms may be evaluated:

Completeness Will the algorithm find a solution of one exists?
Optimality Will the algorithm find the optimal solution, in our case, the shortest path.
Time Complexity What is the algorithm’s processing time Big-Oh?
Memory Use Complexity What is the Big-Oh of the algorithm’s memory use?

12.7.1 Performance Measures for Breadth first

In these calculations, $b$ is the branching factor, i.e. when we expand a node, how many new nodes are created; in the case of the tile map, a cell has eight (8) neighbours, hence $b = 8$. $d$ is the maximum depth, i.e. the depth of the goal.

Completeness Yes.
Optimality Yes.
Time Complexity $O(b^{d+1})$, see below, we assume that processing time is linear in $n$, the number of nodes to be stored.
Memory Use Complexity $O(b^{d+1})$. Each node must be saved in memory. The start is at depth $d = 0$, see Figure 12.29.

\[
\begin{array}{cccccccc}
3 & 3 & 3 & 3 & 3 & 3 & \text{Goal} \\
2 & 2 & 2 & 2 & 2 & 3 \\
2 & 1 & 1 & 1 & 2 \\
2 & 1 & 0 & 1 & 2 \\
2 & 1 & 1 & 1 & 2 \\
2 & 2 & 2 & 2 & 2 \\
\end{array}
\]

Figure 12.29: Estimate of number of nodes at depth $d$.

At $d = 0$, the algorithm expands $b = 8$ nodes (Russell & Norvig 2003, p. 74); then at $d = 1$, each those $b$ expands another $b$ nodes, resulting in $b \times b = b^2$ nodes; at $d = 1$, we have $b \times b^2 = b^3$ nodes and so on. The sum of cells, $n$, out to depth $d$ gives eqn. 12.1

\[
n = b + b^2 + b^3 + \cdots + b^d + (b^{d+1} - b). \tag{12.1}
\]
That is, depth \( d \) normally results in \( b^{d+1} \) new nodes; however, the last term \((b^{d+1} - b)\) is because we subtract the nodes that would have been expanded by the goal, but the search stops when the goal is reached.

The big-Oh (growth rate) indicated by eqn. 12.1 is \( O(b^{d+1}) \), that is, the dreaded exponential (see Chapter 3). For \( b = 8 \), we have:

- Depth \( d = 1 \), \( b^{d+1} = 8^2 = 64 \);
- Depth \( d = 2 \), \( b^{d+1} = 8^3 = 512 \);
- Depth \( d = 4 \), \( b^{d+1} = 8^5 = 32768 \);
- Depth \( d = 6 \), \( b^{d+1} = 8^7 \approx 2 \times 10^6 \);
- Depth \( d = 8 \), \( b^{d+1} = 8^9 \approx 0.1 \times 10^9 \);
- Depth \( d = 10 \), \( b^{d+1} = 8^{11} \approx 10 \times 10^9 \).

This demonstrates the unpleasantness of exponential growth; at \( d = 10 \), we already have \( 10 \times 10^9 \); even if a node used only one byte (it would more likely be 100 bytes), we would have already exhausted a 32-bit address space.

**But searching a tile-map is not quite as bad as that!** If we look again at Figure 12.29 and remember that cells are marked as they are processed (they are in the closed list), then we can see that the new cells added at depth \( d \) corresponds only to the fringe of cells that surround the cells at depth \( d \).

Hence, keeping \( b = 8 \), we have

- Depth \( d = 0 \), \( n = b = 8 \);
- Depth \( d = 1 \), \( n = 2b = 16 \);
- Depth \( d = 2 \), \( n = 3b = 24 \);

Etc. Count the fringe cells on Figure 12.29 to see that this is true. Hence eqn. 12.1 reduces to eqn. 12.2

\[
    n = b + 2b + 3b + \cdots + db + (d + 1)b = [1 + 2 + 3 + \cdots + d + (d + 1)]b. \quad (12.2)
\]

Which is

\[
    b \times \sum_{i=1}^{d+1} i. \quad (12.3)
\]

Since

\[
    \sum_{i=1}^{d+1} i = (d + 1)(d + 2)/2 = O(d^2), \quad (12.4)
\]

we arrive at \( b \times O(d^2) = O(d^2) \) — quadratic growth rate.
12.7.2 Performance Measures for Depth first

Completeness No; but maybe yes in a restricted situation like a tile map?

Optimality No.

Time Complexity $O(b^m)$, where $m$ is the maximum depth reached.

Memory Use Complexity $O(bm)$.

12.8 Pathfinding in non-tile-based environments

See Penton (2003, pp. 762–767) and Brackeen et al. (2004, Chapter 12).

• Binary Space Partition (BSP) Trees Brackeen et al. (2004, Chapter 12);
• Quadtrees Penton (2003, p. 764);

All these can use variations on breadth-first search.

Note that in the case of BSP trees, we will have a memory use and time complexity of $O(b^{d+1})$, where $b = 2$ — back to the dreaded exponential growth rate.
12.9 Software Implementations of the Algorithms

We will examine my adapted versions of the pathfinding demonstration in (Penton 2003, Chapter 23). We note that Brackeen et al. (2004, Chapter 12, pp. 657–686) has a very clear description of \textit{breadth-first} and \textit{A*} and their implementations.
Chapter 13

Sets and Maps

13.1 Introduction

From your mathematics classes, you already know what a set is, that is, a collection of things (objects). In the mathematics classes you probably did operations like set union and set intersection and set complement — we are not too interested in those here, just in a means of indicating the fact that a particular object, say X, is in the set (S) so that we can quickly get an answer is X in set S?

In a game for example, we may have a large number (N) of objects; in an update cycle, a few of them may have been modified or moved and hence need updating before rendering. For the sake of argument, assume N is very large and stored in some sort of data structure (let’s say a list); assume also that the objects have no indicator that they have been modified but that when it has been modified it is added to a set S.

When we come to an object (X), enquiring if it needs updating may waste a lot of time. If we use a list or vector then searching for X is $O(N)$. But using a proper implementation of a set S, such as is available in the standard library, we can get $O(\log N)$.

There is also a multiset which allows duplication of entries in the set; in mathematics this is called a bag; duplication is not allowed in a true set.

Another most useful collection is a map (or dictionary) — a lookup table. Here we have a set of pairs where a pair is a <key, value> pair. For example, <name, phone-number>. You want to be able to quickly search on name to get the phone-number.

Like multiset, there is also a multimap collection. Yhe use of a multimap is a bit easier to imagine than the use of a multiset — for example a dictionary where one word has many entries, or a telephone directory where one name has more than one phone-number.

Sets, multisets, maps, and multimaps are called associative containers because we can access members by their names or keys.
13.2 Implementations

In the *standard library*, sets, multisets, maps and multimaps are implemented based on a variation of *binary search trees*, see ch:2:2:2:2. That is, we can insert, find, or remove in $O(\log N)$.

Actually, because plain *binary search trees* can become *unbalanced*, the *standard library* uses a *red-black tree* data structure — but we’ll avoid the details of that.

We have already dealt with trees, so in this chapter we will be more interested in the *use* of the *standard library*, sets, multisets, maps and multimaps.

It is worth knowing that other possible implementations of sets are:

- *Bitvector*; an efficient implementation of a *small* set, where the identity of a member can be coded as an integer;
- *Hashing*.

We mention these implementations briefly before passing on to the use of *standard library*, sets, multisets, maps and multimaps.

13.3 Bitvector

If the range of keys allowed in the set is a relatively small range of *integers*, then *bit vector* can do the job with efficient speed of insert and search, and with efficient use of memory.

Read (Penton 2003, chapter 4), (Budd 1997, chapter 12.3).

13.4 Hashing and Hash Tables

If the range of keys allowed in the set is a relatively small range of *positive integers*, then an array can do the job with efficient speed of insert and lookup.

**Hash function** However, where a key is a long text string or a large data structure (object), then, if we want to use an array, we need some way of converting the object to an integer; the way of converting is called a *hash function*. A very simple hash function for strings would be the following: (i) sum the character values in the string; (ii) take $hash = sum \% N$; now you have a *key* (hash key) in $0 \ldots N-1$ and you can use an array of size $N$. 
Collisions  The problem is that different strings will give the same hash; these are called collisions.

We can solve collisions in a number of ways. The conceptually simplest is via chaining.

Instead of having a just a value, or a boolean in the case of a simple set, in the array, we now have a list of keys or, in the case of a hash table (hash map), we have a list of (key, value) pairs. In other words, a hash table is an array (0 . . . N-1) of lists. The lists are sometime alled buckets — each entry in the array is a bucket which can contain 0, 1, or more keys.

The hash function takes your key and gives you a hash, say h, in 0 . . . N-1, then you examine the list at index h; if the list is empty, that key is not present; otherwise you check each list entry and see if the key is present in any of them. If it is not, then the key is not in the set.

If the key is found, then it is present. In the case of a hash map, you now look up the value in the (key, value) pair.

Read (Penton 2003, chapter 8), (Budd 1997, chapter 17).

13.5  Example Use of set

Figures 13.1 show an example use of set.
#include <iostream>
#include <iterator>
#include <set>
using namespace std;

int main() {
    cout << "setT1 ..." << endl;
    set<int> c;

    c.insert(21); c.insert(22);
    c.insert(24); c.insert(25);
    c.insert(25); c.insert(25);
    c.insert(26); c.insert(22);

    cout << "c.size() = " << c.size() << endl;

    copy(c.begin(), c.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    // same using iterator
    set<int>::iterator it1 = c.begin();
    set<int>::iterator it2 = c.end();
    for(; it1 != it2; ++it1) {
        cout << *it1 << " ";
    }
    cout << endl;

    // try with 90
    int key = 22;
    set<int>::iterator it3 = c.find(key);
    if(it3 != c.end()) {
        cout << key << " is in c" << endl;
    } else {
        cout << key << " is not in c" << endl;
    }

    c.erase(21);
    cout << "c.size() = " << c.size() << endl;
    copy(c.begin(), c.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
    cout << "finishing setT1..." << endl;
}

Figure 13.1: SetT1.cpp
Output from SetT1.cpp

setT1 ...
c.size() = 5
21 22 24 25 26
21 22 24 25 26
22 is in c
... after c.erase(21);
c.size() = 4
22 24 25 26
finishing setT1...

13.6 Sets need to be able to order elements

Because it uses a binary search tree (or a variation) to store elements, set need to be able to order them. It does this using the less-than operator (<). Attempt to create a set of objects that have no less-than operator will fail. Later we will see an example of a class with which we have equipped such an operator.

Can ‘less-than’ be used used to establish equality? — for example in find or erase. Yes, but it is better to call it equivalence.

Let the two objects be X and Y. If !(X < Y) && !(Y < X) is true then they are equivalent.

Note. The sort algorithm when applied to any collection also uses the less-than operator.

13.7 Example Use of multiset

Figures 13.2 and 13.3 show an example use of multiset.
```cpp
#include <iostream>
#include <iterator>
#include <set>
// #include <multiset> no multiset header, all in set
using namespace std;

int main()
{
    cout << "msetT1 ..." << endl;

    multiset<int> c;
    c.insert(21); c.insert(22); c.insert(24); c.insert(25);
    c.insert(25); c.insert(25); c.insert(26); c.insert(22);

    cout << "c.size() = " << c.size() << endl;
    copy(c.begin(), c.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    // same using iterator
    multiset<int>::iterator it1 = c.begin();
    multiset<int>::iterator it2 = c.end();
    for (; it1 != it2; ++it1)
    {
        cout << *it1 << " ";
    }
    cout << endl;

    cout << "lower_bound(25): " << *c.lower_bound(25) << endl;
    cout << "upper_bound(25): " << *c.upper_bound(25) << endl;
    // equal_range has two parts, first = lower_bound
    // .second = upper_bound
    // be careful dereferencing upper_bound;
    // possibly, upper_bound(.) == collection.end()
    // and that is *one item after* the last legally dereferencable item
    cout << "equal_range(25): " << *c.equal_range(25).first << " "
        << *c.equal_range(25).second << endl;
    cout << endl;

    cout << "lower_bound(23): " << *c.lower_bound(23) << endl;
    cout << "upper_bound(23): " << *c.upper_bound(23) << endl;
    cout << "equal_range(23): " << *c.equal_range(23).first << " "
        << *c.equal_range(23).second << endl;

    copy(c.begin(), c.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
    // continued ...
}
```

Figure 13.2: Set example, part 1
//here is the usual for loop pattern for accessing all instances
//of a key in a multiset
int key = 25; // try with 23, then 25
multiset<int>::const_iterator it1 = c.lower_bound(key);
multiset<int>::const_iterator it2 = c.upper_bound(key);
multiset<int>::const_iterator it;
cout << "instances of " << key << endl;
if(it1 == it2){
    cout << "key " << key << " is not present" << endl;
}
else{
    // this would not cause an error if not guarded
    // it1==it2 mans loop does not execute even once
    for(it = it1; it != it2; ++it){
        cout << *it << " ";
    }
}
cout << endl;

// or, can use ...
cout << "same with copy ...
" << endl;
copy(c.lower_bound(key), c.upper_bound(key), ostream_iterator<int>(cout, " "));
cout << endl;

// you may wonder what is the use of having multiple copies
// of a key; not easy to say, but when we get to multimaps
// we’ll see a valid need.
// A map is a set of <key, value> pairs, something like a
// telephone directory (key = name, value = number) or a
// symbol table in an assembler or compiler (<name, address>)
cout << "finishing setT1...
" << endl;
return 0;
}
Output from msetT1.cpp

msetT1 ...
c.size() = 8
21 22 22 24 25 25 25 26
21 22 22 24 25 25 25 26
lower_bound(25): 25
upper_bound(25): 26
equal_range(25): 25 26

lower_bound(23): 24
upper_bound(23): 24
equal_range(23): 24 24
21 22 22 24 25 25 25 26
instances of 25
25 25 25
same with copy ...
25 25 25
finishing msetT1...

13.8 A Name class whose objects we want to insert in set

Now we’ll create a Name class whose objects we want to insert in set. It’s pretty trivial, but we need to equip it with a less-than operator. In addition, we’ll equip it with an ostream << operator so that we can use the normal output patterns.

Figure 13.4 shows the Name class and Figure 13.5 shows a small test program.
/* ----- Name.h ----------------------------------
j.g.c. 2008-10-15
name of a person
----------------------------------------------------*/

#ifndef NAMEH
#define NAMEH

#include <iostream>
#include <string>
#include <cassert>
using std::string;
using std::ostream; using std::endl;

class Name{
public:
    Name(string last = string("f..."), string first = string("l...") )
        : first_(first), last_(last){}
    bool operator< (const Name& rhs) const;
    string first_;
    string last_;}

bool Name::operator< (const Name& rhs) const{
    if(last_ == rhs.last_)return first_ < rhs.first_; else return last_ < rhs.last_;}

ostream& operator<<(ostream& os, const Name& n){
    os<< n.last_" ", "<< n.first_; return os;
}
#endif

Figure 13.4: Name class
/* ----- NameT1.cpp ----------------------------------
   j.g.c. 2008-10-15
   ---------------------------------------------------- */

#include <iostream>
#include "Name.h"
using namespace std;

int main()
{
    cout << "NameT1 ..." << endl;

    Name n1("Bloggs", "Joe");
    cout << n1 << endl;

    Name n2("Jones", "Andy");
    cout << n2 << endl;

    Name n3("Murphy", "Jane");
    cout << n3 << endl;

    cout << "finishing NameT1..." << endl;
    return 0;
}

Figure 13.5: Name example program
13.9 Set of Names

Figure 13.6 shows an example program of a set of Names.
/* ----- setNameT2.cpp ----------------------------------
  j.g.c.  2008-10-15
  ---------------------------------------------------- */

#include <iostream>
#include <iterator>
#include <set>
#include <vector>

// #include <multiset>  no multiset header, all in set
#include "Name.h"

using namespace std;

int main()
{
  cout<< "setNameT2 ...

  const char* lasts[] = {"Bloggs", "Murphy", "DBrien", "Jones", "Jones"};
  const char* firsts[] = {"Joe", "Jane", "Jack", "Andrew", "Sarah"};
  const int nNames = sizeof(lasts)/sizeof(lasts[0]);

  vector<Name> v;
  for(int i = 0; i < nNames; ++i)
  {
    v.push_back(Name(lasts[i], firsts[i]));
  }
  cout<< "vector v = 
  copy(v.begin(), v.end(), ostream_iterator<Name>(cout, "; "));
  cout<< endl;

  set<Name> c(v.begin(), v.end());
  cout<< "set c = 
  copy(c.begin(), c.end(), ostream_iterator<Name>(cout, "; "));
  cout<< endl;

  Name query(Name(lasts[3], firsts[3]));
  cout<< "Searching for "

  set<Name>::const_iterator res = c.find(query);
  if(res == c.end())
  { // one past last, i.e. not found
    cout<< query<< " not found"<< endl;
  }
  else
  {
    cout<< "result = "
  }
  cout<< "finishing setNameT2...

Figure 13.6: Set of Names

13-12
13.10 A Telephone Directory using a map

Figures 13.7 and 13.8 show a simple telephone directory constructed using a map and Name.
#include <iostream>
#include <iterator>
#include <map>
#include <vector>
#include "Name.h"

using namespace std;

int main()
{
    cout << "mapNameT1 ..." << endl;
    
    const char* lasts[] = {"Bloggs", "Murphy", "OBrien", "Jones", "Jones"};
    const char* firsts[] = {"Joe", "Jane", "Jack", "Andrew", "Sarah"};
    const char* nums[] = {"0749312345", "0749154321", "08797654",
                           "04871271127", "019706812"};

    const int nNames = sizeof(lasts)/sizeof(lasts[0]);

    vector<Name> v;
    for(int i = 0; i < nNames; ++i){
        v.push_back(Name(lasts[i], firsts[i]));
    }
    cout << "vector v = " << endl;
    copy(v.begin(), v.end(), ostream_iterator<Name>(cout, "; "));
    cout << endl;

    typedef map<Name, string> PhoneBook; // phone book
    PhoneBook pb;

    pb[v[0]] = nums[0];  pb[v[1]] = nums[1];  pb[v[2]] = nums[2];
    pb[v[3]] = nums[3];  pb[v[4]] = nums[4];

    pb.insert( make_pair(Name("Cowen", "Brian"), "012342345") );

    PhoneBook::iterator pos;
    cout << "Phone book ..." << endl;
    for(pos = pb.begin(); pos != pb.end(); ++pos){
        cout << pos->first << " = " << pos->second << endl;
    }
    cout << "pb[Name("Bloggs", "Joe")] = " << pb[Name("Bloggs", "Joe")]<< endl;
    cout << "pb[Name("Bloggs", "Jim")] = " << pb[Name("Bloggs", "Jim")]<< endl;
    // continued ...
}

Figure 13.7: Telephone directory (mapNameT1.cpp), part 1

13–14
typedef map<string, Name> InvPhoneBook; // inverse phone book
InvPhoneBook ipb;
for(pos = pb.begin(); pos != pb.end(); ++pos){
    ipb.insert(make_pair(pos->second, pos->first));
}

InvPhoneBook::iterator ipos;
for(ipos = ipb.begin(); ipos != ipb.end(); ++ipos){
    cout<< ipos->first << " " => " " << ipos->second<< endl;
}

cout<< "ipb["012342345"] = " << ipb["012342345"]<< endl;

cout<< "finished mapNameT1..."<< endl;
return 0;
}

Figure 13.8: Telephone directory (mapNameT1.cpp), part 2
Discussion

1. Output from mapNameT1.cpp:

```cpp
mapNameT1 ...
vector v =
Bloggs, Joe; Murphy, Jane; OBrien, Jack; Jones, Andrew; Jones, Sarah;
Phone book ...
Bloggs, Joe => 0749312345
Cowen, Brian => 012342345
Jones, Andrew => 04871271127
Jones, Sarah => 019706812
Murphy, Jane => 0749154321
OBrien, Jack => 08797654
pb[Name("Bloggs", "Joe")]) = 0749312345
pb[Name("Bloggs", "Jim")]) =
    => Bloggs, Jim
012342345 => Cowen, Brian
019706812 => Jones, Sarah
04871271127 => Jones, Andrew
0749154321 => Murphy, Jane
0749312345 => Bloggs, Joe
08797654 => OBrien, Jack
ipb["012342345"] = Cowen, Brian
finishing mapNameT1...
```

2. You can save a lot of typing and make the program easier to read by declaring a typename synonym:

```cpp
typedef map<Name, string> PhoneBook;
```

3. Notice how you can index on the key, i.e a non-integer index:

```cpp
pb[v[0]] = nums[0];
```

```cpp
pb[Name("Bloggs", "Joe")]) ...
```

but it is maybe better to use insert and find:

```cpp
pb.insert( make_pair(Name("Cowen", "Brian"), "012342345") );
```

because indexing does not work on multiset. Why do you think that is so?

4. Notice the use of pair and make_pair(.,.).

5. pair has template members first and second and they are public.

13.11 A Telephone Directory using a multimap

Figures 13.9 and 13.10 show a simple telephone directory constructed using a multimap and Name; that is, we allow multiple entries of a Name.
#include <iostream>
#include <iterator>
#include <map>
#include <vector>
#include "Name.h"
using namespace std;

int main()
{
    cout << "mapNameT2 ..." << endl;

    const char* lasts[] = {"Bloggs", "Murphy", "DBrien", "Jones", "Jones"};
    const char* firsts[] = {"Joe", "Jane", "Jack", "Andrew", "Sarah"};
    const char* nums[] = {"0749312345", "0749154321", "08797654",
                          "04871271127", "019706812"};

    const int nNames = sizeof(lasts)/sizeof(lasts[0]);

    vector<Name> v;
    for(int i = 0; i < nNames; ++i) {
        v.push_back(Name(lasts[i], firsts[i]));
    }
    cout << "vector v = " << endl;
    copy(v.begin(), v.end(), ostream_iterator<Name>(cout, "; "));
    cout << endl;

typedef multimap<Name, string> PhoneBook; // phone book
PhoneBook pb;

    // multiset, in contrast to set, does not allow direct element
    // access via []
    // pb[v[0]] = nums[0]; is not allowed; instead, use insert

    for(int i = 0; i < nNames; ++i) {
        pb.insert(make_pair(v[i], nums[i]));
    }
    pb.insert(make_pair(Name("Cowen", "Brian"), "012342345"));
    pb.insert(make_pair(Name("Cowen", "Brian"), "012342346"));
    pb.insert(make_pair(Name("Cowen", "Brian"), "012342347"));

    PhoneBook::iterator pos;
    cout << "Phone book ..." << endl;

    for(pos = pb.begin(); pos != pb.end(); ++pos) {
        cout << pos->first << " => " << pos->second << endl;
    }
    // continued ...

Figure 13.9: Telephone directory (mapNameT2.cpp), part 1

13–17
Name key(Name("Cowen", "Brian"));

PhoneBook::const_iterator it1 = pb.lower_bound(key);
PhoneBook::const_iterator it2 = pb.upper_bound(key);
PhoneBook::const_iterator it;

cout<< "occurrences of "<< key<< endl;
if(it1 == it2){
    cout << "key "<< key << " is not present"<< endl;
}
else{
    // this would not cause an error if not guarded
    // it1==it2 means loop does not execute even once
    for(it = it1; it != it2; ++it){
        cout<< it->first << " => "<< it->second<< endl;
    }
}

cout<< endl;

typedef map<string, Name> InvPhoneBook; // inverse phone book
// here we’ll assume that there is only ever one occurrence
// of a phone number
InvPhoneBook ipb;
for(pos = pb.begin(); pos != pb.end(); ++pos){
    ipb.insert(make_pair(pos->second, pos->first));
}

InvPhoneBook::iterator ipos;
for(ipos = ipb.begin(); ipos != ipb.end(); ++ipos){
    cout<< ipos->first << " => "<< ipos->second<< endl;
}

cout<< "ipb["012342345"] = "<< ipb["012342345"]<< endl;
// if we access something that is not present
// an item with value == default of value-type,
// here Name(), is added.
//cout<< "ipb["012342340"] = "<< ipb["012342340"]<< endl;

for(ipos = ipb.begin(); ipos != ipb.end(); ++ipos){
    cout<< ipos->first << " => "<< ipos->second<< endl;
}

// hence it is probably better to use 'find' when 'reading'
cout<< "using find ...
"<< endl;
string key1("012342345"); // try "012342340" and "012342345"
ipos = ipb.find(key1);
if(ipos == ipb.end()){
    cout<< "key "<< key1<< " not present"<< endl;
}
else{
    cout<< ipos->first << " => "<< ipos->second<< endl;
}
} }

Figure 13.10: Telephone directory (mapNameT2.cpp), part 2
Bibliography


