Data Structures and Algorithms - Table of Contents

9.6 Optimal Triangulation
Graphs
10.1 Minimum Spanning Tree
10.2 Dijkstra's Algorithm
Huffman Encoding
FFT
Hard or Intractable Problems
13.1 Eulerian or Hamiltonian Paths
13.2 Travelling Salesman's Problem
Games
Appendices
ANSI C
Source code listings
Getting these notes
Slides
Slides from 1998 lectures (PowerPoint).
Course Management
Key Points from Lectures
Workshops
Past Exams
Tutorials
Texts
Texts available in UWA library
Other on-line courses and texts
Algorithm Animations

Data Structures and Algorithms

John Morris,
Electrical and Electronic Engineering,
University of Western Australia

These notes were prepared for the Programming Languages and System Design course in the BE(Information Technology) course at the University of Western Australia. The course covers:
Algorithm Complexity
Polynomial and Intractible Algorithms
Classes of Efficient Algorithms
Divide and Conquer
Dynamic
Greedy
Searching
Lists
Trees
Binary
Red-Black
AVL

B-trees and other m-way trees
Optimal Binary Search Trees
Hash Tables
Queues
Heaps and Priority Queues
Sorting
Quick
Heap
Bin and Radix
Graphs
Minimum Spanning Tree
Dijkstra's Algorithm
Huffman Encoding
Fast Fourier Transforms
Matrix Chain Multiplication
Intractible Problems
Alpha-Beta search
The algorithm animations were mainly written by Woi Ang with contributions by Chien-Wei Tan, Mervyn Ng, Anita Lee and John Morris.

PLDS210
Programming Languages and Data Structures
Course Synopsis
This course will focus on data structures and algorithms for manipulating them. Data structures for storing information in tables, lists, trees, queues and stacks will be covered. Some basic graph and discrete transform algorithms will also be discussed.

You will also be introduced to some basic principles of software engineering: good programming practice for "long-life" software.

For a full list of topics to be covered, view the table of contents page for the lecture notes.
Lectures - 1998
There are two lectures every week:
Monday 12 pm E273
Tuesday 12 pm AG11
Lecture Notes
A set of notes for this course is available on the Web. From the table of contents page you can jump to any section of the course.

There is a home page set up for student information:
http://www.ee.uwa.edu.au/internal/ug.courses.html
On which, you will find an entry for course information; you can follow the links to this page and the notes themselves. You can also go directly to the PLSD210 page:
http://www.ee.uwa.edu.au/~plsd210/ds/plds210.html
Note that the Web pages use the string plds (programming languages and data structures) - a historical accident, which we retain because this label describes the content more accurately!
Printed Notes
For a ridiculously low price, you can obtain a pre-printed copy of the notes from the

bookshop. You are strongly advised to do so, as this will enable you to avoid laboriously taking notes in lectures and concentrate on understanding the material. (It will also save you a large amount of time printing each page from a Web browser!)

The printed notes accurately represent the span of the course: you will be specifically advised if examinable material not appearing in these notes is added to the course. (But note that anything appearing in laboratory exercises and assignments is automatically considered examinable: this includes the feedback notes!)

However, the Web notes are undergoing constant revision and improvement (comments are welcome!) so you are advised to browse through the Web copies for updated pages. You'll be advised in lectures if there is a substantial change to any section.

Textbooks

The material on data structures and algorithms may be found in many texts: lists of reference books in the library are part of the Web notes. The Web notes are, of necessity, abbreviated and should not be considered a substitute for studying the material in texts.

Web browsers

Web browsers have varying capabilities: the notes were checked with Netscape 2 - but should read intelligently with other browsers. If you have problems, I would be interested to know about them, but please note that updating these notes, adding the animations, tutoring and marking your assignments for this course have priority: problems with other browsers, your home computer, etc, will only be investigated if time permits.

Using the notes

The notes make use of the hypertext capabilities of Web browsers: you will find highlighted links to subsidiary information scattered throughout the text. Occasionally these links will point to Web resources which may be located off campus and take some time to download: you may find it productive to use the "Stop" facility on the browser to abort the current fetch - you can try again later when the Net is less heavily loaded.

In all cases, the browser's "Back" command should take you back to the original page.

Program source

Example source code for programs will sometimes pop up in a separate window. This is to enable you to scan the code while referring to the notes in the main page. You will probably need to move the source code page out of the way of the main page. When you have finished with the source code page, select File:Close to close the window. Selecting File:Exit will close the window and exit from Netscape - possibly not your intention!

Tutorials - 1997

Exercises for the tutorials and laboratory sessions are also found in the Web pages.

Tutorial Times

Weeks	Time	Location	Groups
4-13	Thursday 9 am	E273	2ic,2it1-3

Thursday
2 pm E269 rest

The first tutorial will be in the fourth week of semester. As long as one tutorial group does not become ridiculously overloaded, you may go to whichever tutorial suits you best.

Laboratory Sessions - 1998
There will be two formal introductory laboratory sessions early in the semester - watch these pages for the final details. These sessions will be in laboratory G.50. After the first two laboratories, a tutor will be available in G.50 every week at times to be advertised. The tutor will advise on any problems related to the whole course: assignments, lecture material, etc.

You will be able to complete the assignment on any machine which has an ANSI C compiler. Assignments will be submitted electronically: submit programs on the SGI machines and on the NT systems in 1.51 may be used - refer to the submission instructions. Note that you are expected to write ANSI standard C which will run on any machine: programs which won't run on our SGI's risk failure! In 1998, Java programs written to an acceptable standard will also be accepted. (The standard required for C is set out explicitly: ensure that you understand how to translate the important elements of this to Java before starting work in Java. Seek feedback if uncertain!)
Assessment
Assignments 20%
Written Exam (3hrs) 80%
As with all other courses with a practical component, the practical assignments are compulsory. Failure to obtain a satsifactory grade in the practical component of the course may cause you to be given a 0 for this component of PLSD210. Since this will make it virtually impossible to obtain more than a faculty pass for the year, failure to do the practical assignments will not only cause you to miss some feedback which may well be useful to you in the written exam, but may cause you to fail the whole unit.

A "satisfactory" grade in assignments is more than 40% overall. Any less will put your whole year at risk. A much safer course is to do the assignments conscientiously, making sure that you understand every aspect of them: assume that the effort put into them will improve your examination mark also.
Assignments - 1998
Four assignment exercises will be set for the semester. You should be able to complete most of the first two assignments during the initial laboratory sessions. The 3rd and 4th are more substantial. Completed assignments (which should include a summary report, the program code and any relevant output) should be submitted by following the submission instructions at the end of the Web page.

Performance on the assignments will be 20% of your overall assessment for the unit.
Assignments 1 & 2
These will be relatively short and should require only 1 or 2 hours extra work to complete. They contribute 6% of your final assessment. These assignments will provide some feedback on what is expected for the remaining two assignments. You

may even find that you can use the (corrected) code from these assignments in the later assignments.

Assignments 3 & 4

For these two assignments, you will be expected to implement one algorithm and test another. You will be assigned an algorithm to implement as assignment 3. You may obtain from one of your class colleagues an implementation of any other algorithm and test it for assignment 4. You must submit them by the dates shown on the assignment sheets. They will constitute the remaining 14% of your assignment assessment.

A minimum standard must be obtained in the assignments to pass the unit as a whole.

Failure to attempt the assignments will put you at a severe disadvantage in the exam.

Assignment reports

Each assignment submission should be accompanied by a summary report. The report should be clear and concise: it is unlikely that you will need to write more than 2 A4 pages (or about 120 lines of text).

Report Format

The report should be in plain ASCII text. The 'native form' of any wordprocessor will be rejected. If you prefer to use a word processor to prepare your report, then ensure that you export a plain text file for submission when you have finished: all word-processors have this capability.

This allows you to concentrate on the content of the report, rather than the cosmetics of its format. However, the general standards for report structure and organisation (title, authors, introduction, body grouped into related paragraphs, conclusion, etc) expected for any other unit apply here also.

Communication

This course attempts to be "paperless" as much as possible! Assignments will be submitted electronically and comments will be emailed back to you. Please ensure that your reports include email addresses of all authors.

The preferred method for communication with the lecturer and tutor(s) is, at least initially, email. All routine queries will be handled this way: we will attempt to respond to all email messages by the next day. If you have more complex problems, email for an appointment (suggest a few times when you will be free). You may of course try to find me in my office at any time (but early in the morning is likely to be a waste of time), but emailing for an appointment first ensures you some priority and enables you to avoid wasting a trip to the 4th floor when there may be zero probability of success!

Continue on the lecture notes.

Data Structures and Algorithms

1. Introduction

This course is designed to teach you how to program efficiently. It assumes that

you know the basics of programming in C,
can write, debug and run simple programs in C, and
have some simple understanding of object-oriented design.
An introduction to object-oriented programming using ANSI standard C may be found in the companion Object First course.

Good Programs
There are a number of facets to good programs: they must
run correctly
run efficiently
be easy to read and understand
be easy to debug and
be easy to modify.
What does correct mean?

We need to have some formal notion of the meaning of correct: thus we define it to mean
"run in accordance with the specifications".
The first of these is obvious - programs which don't run correctly are clearly of little use. "Efficiently" is usually understood to mean in the minimum time - but occasionally there will be other constraints, such as memory use, which will be paramount. As will be demonstrated later, better running times will generally be obtained from use of the most appropriate data structures and algorithms, rather than through "hacking", i.e. removing a few statements by some clever coding - or even worse, programming in assembler!

This course will focus on solving problems efficiently: you will be introduced to a number of fundamental data structures and algorithms (or procedures) for manipulating them.

The importance of the other points is less obvious. The early history of many computer installations is, however, testimony to their importance. Many studies have quantified the enormous costs of failing to build software systems that had all the characteristics listed. (A classic reference is Boehm's text.) Unfortunately, much recent evidence suggests that these principles are still not well understood! Any perusal of Risks forum will soon convince you that there is an enormous amount of poor software in use. The discipline of software engineering is concerned with building large software systems which perform as their users expected, are reliable and easy to maintain. This course will introduce some software engineering principles but we will concentrate on the creation of small programs only. By using well-known, efficient techniques for solving problems, not only do you produce correct and fast programs in the minimum time, but you make your programs easier to modify. Another software engineer will find it much simpler to work with a well-known solution than something that has been hacked together and "looks a bit like" some textbook algorithm.

Key terms
correct
A correct program runs in accordance with its specifications
algorithm

A precisely specified procedure for solving a problem.

. Programming Strategies

It is necessary to have some formal way of constructing a program so that it can be built efficiently and reliably. Research has shown that this can be best done by decomposing a program into suitable small modules, which can themselves be written and tested before being incorporated into larger modules, which are in turn constructed and tested. The alternative is create what was often called "sphaghetti code" because of its tangled of statements and jumps. Many expensive, failed projects have demonstrated that, however much you like to eat sphaghetti, using it as a model for program construction is not a good idea!

It's rather obvious that if we split any task into a number of smaller tasks which can be completed individually, then the management of the larger task becomes easier. However, we need a formal basis for partitioning our large task into smaller ones. The notion of abstraction is extremely useful here. Abstractions are high level views of objects or functions which enable us to forget about the low level details and concentrate on the problem at hand.

To illustrate, a truck manufacturer uses a computer to control the engine operation - adjusting fuel and air flow to match the load. The computer is composed of a number of silicon chips, their interconnections and a program. These details are irrelevant to the manufacturer - the computer is a black box to which a host of sensors (for engines speed, accelerator pedal position, air temperature, etc) are connected. The computer reads these sensors and adjusts the engine controls (air inlet and fuel valves, valve timing, etc) appropriately. Thus the manufacturer has a high level or abstract view of the computer. He has specified its behaviour with statements like:

"When the accelerator pedal is 50% depressed, air and fuel valves should be opened until the engine speed reaches 2500rpm".

He doesn't care how the computer calculates the optimum valve settings - for instance it could use either integer or floating point arithmetic - he is only interested in behaviour that matches his specification.

In turn, the manager of a transport company has an even higher level or more abstract view of a truck. It's simply a means of transporting goods from point A to point B in the minimum time allowed by the road traffic laws. His specification contains statements like:

"The truck, when laden with 10 tonnes, shall need no more than 20l/100km of fuel when travelling at 110kph."

How this specification is achieved is irrelevant to him: it matters little whether there is a control computer or some mechanical engineer's dream of cams, rods, gears, etc.

There are two important forms of abstraction: functional abstraction and structural abstraction. In functional abstraction, we specify a function for a module, i.e.

"This module will sort the items in its input stream into ascending order based on an ordering rule for the items and place them on its output stream."

As we will see later, there are many ways to sort items - some more efficient than others. At this level, we are not concerned with how the sort is performed, but simply that the output is sorted according to our ordering rule.

The second type of abstraction - structural abstraction - is better known as object orientation. In this approach, we construct software models of the behaviour of real world items, i.e. our truck manufacturer, in analysing the performance of his vehicle, would employ a software model of the control computer. For him, this model is abstract - it could mimic the behaviour of the real computer by simply providing a behavioural model with program statements like:

```
if ( pedal_pos > 50.0 ) {
set_air_intake( 0.78*pedal_pos);
set_fuel_valve( 0.12 + 0.32*pedal_pos);
}
```

Alternatively, his model could incorporate details of the computer and its program.

However, he isn't concerned: the computer is a "black box" to him and he's solely concerned with its external behaviour. To simplify the complexity of his own model (the vehicle as a whole), he doesn't want to concern himself with the internal workings of the control computer; he wants to assume that someone else has correctly constructed a reliable model of it for him.

Key terms
hacking
Producing a computer program rapidly, without thought and without any design methodology.

2.1 Objects and ADTs

In this course, we won't delve into the full theory of object-oriented design. We'll concentrate on the pre-cursor of OO design: abstract data types (ADTs). A theory for the full object oriented approach is readily built on the ideas for abstract data types.

An abstract data type is a data structure and a collection of functions or procedures which operate on the data structure.

To align ourselves with OO theory, we'll call the functions and procedures methods and the data structure and its methods a class, i.e. we'll call our ADTs classes. However our classes do not have the full capabilities associated with classes in OO theory. An instance of the class is called an object . Objects represent objects in the real world and appear in programs as variables of a type defined by the class. These terms have exactly the same meaning in OO design methodologies, but they have additional properties such as inheritance that we will not discuss here.

It is important to note the object orientation is a design methodology. As a consequence, it is possible to write OO programs using languages such as C, Ada and

Pascal. The so-called OO languages such as C++ and Eiffel simply provide some compiler support for OO design: this support must be provided by the programmer in non-OO languages.

2.2 An Example: Collections

Programs often deal with collections of items. These collections may be organised in many ways and use many different program structures to represent them, yet, from an abstract point of view, there will be a few common operations on any collection. These might include:

create	Create a new collection
add	Add an item to a collection
delete	Delete an item from a collection
find	Find an item matching some criterion in the collection
destroy	Destroy the collection

2.2.1 Constructors and destructors

The create and destroy methods - often called constructors and destructors - are usually implemented for any abstract data type. Occasionally, the data type's use or semantics are such that there is only ever one object of that type in a program. In that case, it is possible to hide even the object's `handle' from the user. However, even in these cases, constructor and destructor methods are often provided.

Of course, specific applications may call for additional methods, e.g. we may need to join two collections (form a union in set terminology) - or may not need all of these.

One of the aims of good program design would be to ensure that additional requirements are easily handled.

2.2.2 Data Structure

To construct an abstract software model of a collection, we start by building the formal specification. The first component of this is the name of a data type - this is the type of objects that belong to the collection class. In C, we use typedef to define a new type which is a pointer to a structure:

```
typedef struct collection_struct *collection;
```

Note that we are defining a pointer to a structure only; we have not specified details of the attributes of the structure. We are deliberately deferring this - the details of the implementation are irrelevant at this stage. We are only concerned with the abstract behaviour of the collection. In fact, as we will see later, we want to be able to substitute different data structures for the actual implementation of the collection, depending on our needs.

The typedef declaration provides us with a C type (class in OO design parlance), collection. We can declare objects of type collection wherever needed. Although C forces us to reveal that the handle for objects of the class is a pointer, it is better to take an abstract view: we regard variables of type collection simply as handles to objects of the class and forget that the variables are actually C pointers.

2.2.3 Methods

Next, we need to define the methods:

```
collection ConsCollection( int max_items, int item_size );
void AddToCollection( collection c, void *item );
void DeleteFromCollection( collection c, void *item );
void *FindInCollection( collection c, void *key );
```

Note that we are using a number of C "hacks" here. C - even in ANSI standard form - is not exactly the safest programming language in the sense of the support it provides for the engineering of quality software. However, its portability and extreme popularity mean that it is a practical choice for even large software engineering projects. Unfortunately, C++, because it is based on C, isn't much better. Java, the latest fad in the software industry, shows some evidence that its designers have learned from experience (or actually read some of the literature in programming language research!) and has eliminated some of the more dangerous features of C.

Just as we defined our collection object as a pointer to a structure, we assume that the object which belong in this collection are themselves represented by pointers to data structures. Hence in AddToCollection, item is typed void *. In ANSI C, void * will match any pointer - thus AddToCollection may be used to add any object to our collection. Similarly, key in FindInCollection is typed void *, as the key which is used to find any item in the collection may itself be some object. FindInCollection returns a pointer to the item which matches key, so it also has the type void *. The use of void * here highlights one of the deficiencies of C: it doesn't provide the capability to create generic objects, cf the ability to define generic packages in Ada or templates in C++.

Note there are various other "hacks" to overcome C's limitations in this area. One uses the pre-processor. You might like to try to work out an alternative approach and try to convince your tutor that it's better than the one set out here!

2.2.4 Pre- and post-conditions

No formal specification is complete without pre- and post-conditions. A useful way to view these is as forming a contract between the object and its client. The pre-conditions define a state of the program which the client guarantees will be true before calling any method, whereas the post-conditions define the state of the program that the object's method will guarantee to create for you when it returns.

Again C (unlike Eiffel, for example) provides no formal support for pre- and post-conditions. However, the standard does define an assert function which can (and should!) be used to verify pre- and post-conditions [man page for assert]. We will see how this is used when we examine an implementation of our collection object. Thus pre- and post-conditions should be expressed as comments accompanying the method definition.

Adding pre- and post-conditions to the collection object would produce:
Select to load collection.h

Aside

In order to keep the discussion simple at this stage, a very general specification of a collection has been implied by the definitions used here. Often, we would restrict our specification in various ways: for example, by not permitting duplicates (items with

the same key) to be added to the collection. With such a collection, the pre- and post-conditions can be made more formal:
Select to load ucollection.h

Note how the pre- and post-conditions now use the FindInUCollection function to more precisely define the state of the object before and after the method has been invoked. Such formal pre- and post-conditions are obviously much more useful than the informal English ones previously specified. They are also easier to translate to appropriate assertions as will be seen when the implementation is constructed.

2.2.5 C conventions

This specification - which all a user or client of this object needs to see (he isn't interested in the implementation details) - would normally be placed in a file with a .h (h = header) suffix to its name. For the collection, we would place the specifications in files called collection.h and Ucollection.h and use the C #include facility to import them into programs which needed to use them. The implementation or body of the class is placed in a file with a .c suffix.

2.7 Error Handling

No program or program fragment can be considered complete until appropriate error handling has been added. Unexpected program failures are a disaster - at the best, they cause frustration because the program user must repeat minutes or hours of work, but in life-critical applications, even the most trivial program error, if not processed correctly, has the potential to kill someone.

If an error is fatal, in the sense that a program cannot sensibly continue, then the program must be able to "die gracefully". This means that it must
inform its user(s) why it died, and
save as much of the program state as possible.

2.7.1 Defining Errors

The first step in determining how to handle errors is to define precisely what is considered to be an error. Careful specification of each software component is part of this process. The pre-conditions of an ADT's methods will specify the states of a system (the input states) which a method is able to process. The post-conditions of each method should clearly specify the result of processing each acceptable input state. Thus, if we have a method:

```
int f( some_class a, int i )
/* PRE-CONDITION: i >= 0 */
/* POST-CONDITION:
   if ( i == 0 )
      return 0 and a is unaltered
   else
      return 1 and update a's i-th element by .... */
```

This specification tells us that i==0 is a meaningless input that f should flag by returning 0 but otherwise ignore.
f is expected to handle correctly all positive values of i.
The behaviour of f is not specified for negative values of i, ie it also tells us that
It is an error for a client to call f with a negative value of i.

Thus, a complete specification will specify
all the acceptable input states, and
the action of a method when presented with each acceptable input state.
By specifying the acceptable input states in pre-conditions, it will also divide responsibility for errors unambiguously.
The client is responsible for the pre-conditions: it is an error for the client to call the method with an unacceptable input state, and
The method is responsible for establishing the post-conditions and for reporting errors which occur in doing so.

2.7.2 Processing errors
Let's look at an error which must be handled by the constructor for any dynamically allocated object: the system may not be able to allocate enough memory for the object.

A good way to create a disaster is to do this:

```
X ConsX( .... )
   {
   X x = malloc( sizeof(struct t_X) );
   if ( x == NULL ) {
      printf("Insuff mem\n"); exit( 1 );
      }
   else
     .....
   }
```

Not only is the error message so cryptic that it is likely to be little help in locating the cause of the error (the message should at least be "Insuff mem for X"!), but the program will simply exit, possibly leaving the system in some unstable, partially updated, state. This approach has other potential problems:
What if we've built this code into some elaborate GUI program with no provision for "standard output"? We may not even see the message as the program exits!
We may have used this code in a system, such as an embedded processor (a control computer), which has no way of processing an output stream of characters at all.
The use of exit assumes the presence of some higher level program, eg a Unix shell, which will capture and process the error code 1.
As a general rule, I/O is non-portable!
A function like printf will produce error messages on the 'terminal' window of your modern workstation, but if you are running a GUI program like Netscape, where will the messages go?

So, the same function may not produce useful diagnostic output for two programs running in different environments on the same processor! How can we expect it to be useful if we transport this program to another system altogether, eg a Macintosh or a Windows machine?

Before looking at what we can do in ANSI C, let's look at how some other languages tackle this problem.

2.7 Programming Languages

This section contains some notes on capabilities of programming languages. The first sub-section discusses the ability to pass a function as an argument to another function - an important capability which enables us to create flexible generic ADTs in ANSI C. The remaining sub-sections give brief overviews of the object-oriented capabilities of C++, Java and Ada - three of the more important programming languages.
Functions as data types in C
C++ classes
Java classes
ADTs in Ada

Data Structures

In this section, we will examine some fundamental data structures: arrays, lists, stacks and trees.
3.1 Arrays
The simplest way to implement our collection is to use an array to hold the items. Thus the implementation of the collection object becomes:

```
/* Array implementation of a collection */
#include <assert.h>/* Needed for assertions */
#include "collection.h"/* import the specification */

struct t_collection {
int item_cnt;
int max_cnt;/* Not strictly necessary */
int item_size;/* Needed by FindInCollection */
void *items[];
};
```

Points to note:

We have imported the specification of this object into the implementaton - this enables the compiler to verify that the implementation and the specification match. Although it's not necessary to include the specification (cf function prototypes), it is much safer to do so as it enables the compiler to detect some common errors and ensures that the specification and its implementation remain consistent when the object is changed.

items is typed as an array of void * in the struct. It is an array of item's which happen to be pointers - but remember that we are trying to hide this from users of the class. Many C programmers would write the equivalent void ** here.

A question:

Why is the attribute max_cnt not strictly necessary?
Hint: it's related to the pre- and post-conditions specified for methods on this object.

The implementations of the methods are:

Select here to load collection.c

Points to note:

ConsCollection uses the memory allocator calloc to dynamically allocate memory off the program's heap for the collection. Two calls are necessary - one to allocate space for the "header" structure itself and one to allocate space for the array of item pointers.

assert calls have been added for the pre-conditions (cf full description of assert). Note that the pre-conditions here are expressed as a number of conditions linked by &&. Since assert requires a single boolean expression as its argument, one assert would suffice. However, we have chosen to implement each individual condition as a separate assert. This is done to assist de-bugging: if the pre-conditions are not satisfied, it is more helpful to know which one of multiple conditions has not been satisfied!

memcmp is a standard function which compares blocks of memory byte by byte [man page for memcmp].

The use of memcp and ItemKey severely constrain the form of the key - it must be in a contiguous string of characters in the item. There are ways of providing more flexible keys (eg ones having multiple fields within item or ones calculated from item. These rely on C capabilities which will be discussed in a later section.

There is no treatment of errors, e.g. if no memory is available on the heap for calloc.

This is a serious shortcoming.

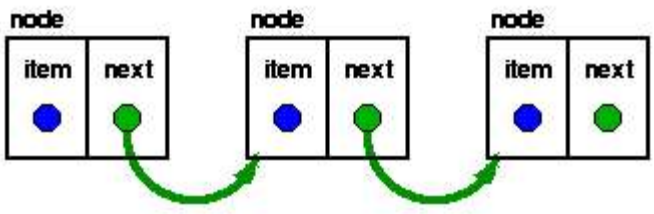
No software without a consistent strategy for detecting, reporting and recovering from errors can be considered well engineered. It is difficult to debug, prone to crashes from faults which are difficult to correct because there is no indication of the source of the error.

Error handling is addressed in a later section.

3.2 Lists

The array implementation of our collection has one serious drawback: you must know the maximum number of items in your collection when you create it. This presents problems in programs in which this maximum number cannot be predicted accurately when the program starts up. Fortunately, we can use a structure called a linked list to overcome this limitation.

3.2.1Linked lists



The linked list is a very flexible dynamic data structure: items may be added to it or deleted from it at will. A programmer need not worry about how many items a program will have to accommodate: this allows us to write robust programs which require much less maintenance. A very common source of problems in program maintenance is the need to increase the capacity of a program to handle larger collections: even the most generous allowance for growth tends to prove inadequate over time!

In a linked list, each item is allocated space as it is added to the list. A link is kept with each item to the next item in the list.

Each node of the list has two elements
the item being stored in the list and
a pointer to the next item in the list
The last node in the list contains a NULL pointer to indicate that it is the end or tail of the list.

As items are added to a list, memory for a node is dynamically allocated. Thus the number of items that may be added to a list is limited only by the amount of memory available.
Handle for the list
The variable (or handle) which represents the list is simply a pointer to the node at the head of the list.
Adding to a list
The simplest strategy for adding an item to a list is to:
allocate space for a new node,
copy the item into it,
make the new node's next pointer point to the current head of the list and
make the head of the list point to the newly allocated node.
This strategy is fast and efficient, but each item is added to the head of the list.

An alternative is to create a structure for the list which contains both head and tail pointers:

```
struct fifo_list {
struct node *head;
struct node *tail;
};
```

The code for AddToCollection is now trivially modified to make a list in which the item most recently added to the list is the list's tail.

The specification remains identical to that used for the array implementation: the max_item parameter to ConsCollection is simply ignored [7]

Thus we only need to change the implementation. As a consequence, applications which use this object will need no changes. The ramifications for the cost of software maintenance are significant.

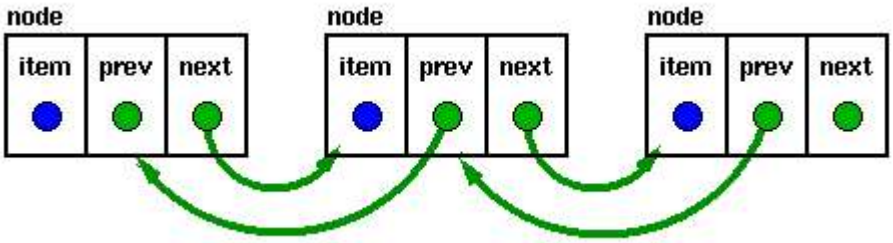
The data structure is changed, but since the details (the attributes of the object or the

elements of the structure) are hidden from the user, there is no impact on the user's program.

Select here to load collection_ll.c

Points to note:
This implementation of our collection can be substituted for the first one with no changes to a client's program. With the exception of the added flexibility that any number of items may be added to our collection, this implementation provides exactly the same high level behaviour as the previous one.
The linked list implementation has exchanged flexibility for efficiency - on most systems, the system call to allocate memory is relatively expensive. Pre-allocation in the array-based implementation is generally more efficient. More examples of such trade-offs will be found later.

The study of data structures and algorithms will enable you to make the implementation decision which most closely matches your users' specifications.
3.2.2 List variants
Circularly Linked Lists
By ensuring that the tail of the list is always pointing to the head, we can build a circularly linked list. If the external pointer (the one in struct t_node in our implementation), points to the current "tail" of the list, then the "head" is found trivially via tail->next, permitting us to have either LIFO or FIFO lists with only one external pointer. In modern processors, the few bytes of memory saved in this way would probably not be regarded as significant. A circularly linked list would more likely be used in an application which required "round-robin" scheduling or processing.
Doubly Linked Lists



Doubly linked lists have a pointer to the preceding item as well as one to the next.

They permit scanning or searching of the list in both directions. (To go backwards in a simple list, it is necessary to go back to the start and scan forwards.) Many applications require searching backwards and forwards through sections of a list: for example, searching for a common name like "Kim" in a Korean telephone directory would probably need much scanning backwards and forwards through a small region of the whole list, so the backward links become very useful. In this case, the node structure is altered to have two links:

```
struct t_node {
    void *item;
    struct t_node *previous;
```

```
    struct t_node *next;
    } node;
```

Lists in arrays

Although this might seem pointless (Why impose a structure which has the overhead of the "next" pointers on an array?), this is just what memory allocators do to manage available space.

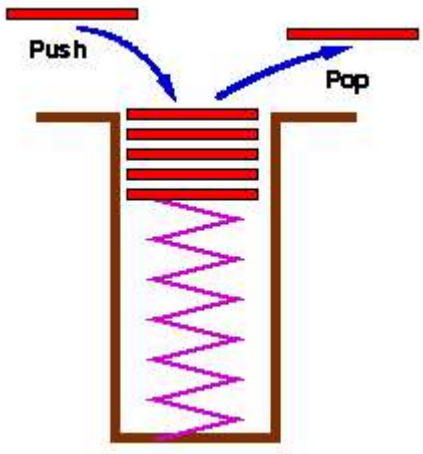
Memory is just an array of words. After a series of memory allocations and de-allocations, there are blocks of free memory scattered throughout the available heap space. In order to be able to re-use this memory, memory allocators will usually link freed blocks together in a free list by writing pointers to the next free block in the block itself. An external free list pointer pointer points to the first block in the free list. When a new block of memory is requested, the allocator will generally scan the free list looking for a freed block of suitable size and delete it from the free list (re-linking the free list around the deleted block). Many variations of memory allocators have been proposed: refer to a text on operating systems or implementation of functional languages for more details. The entry in the index under garbage collection will probably lead to a discussion of this topic.

Key terms

Dynamic data structures

Structures which grow or shrink as the data they hold changes. Lists, stacks and trees are all dynamic structures.

3.3Stacks



Another way of storing data is in a stack. A stack is generally implemented with only two principle operations (apart from a constructor and destructor methods):

push adds an item to a stack

pop extracts the most recently pushed item from the stack

Other methods such as

top returns the item at the top without removing it [9]

isempty determines whether the stack has anything in it

are sometimes added.

A common model of a stack is a plate or coin stacker. Plates are "pushed" onto

to the top and "popped" off the top.

Stacks form Last-In-First-Out (LIFO) queues and have many applications from the parsing of algebraic expressions to ...

A formal specification of a stack class would look like:

```
typedef struct t_stack *stack;

stack ConsStack( int max_items, int item_size );
/* Construct a new stack
   Pre-condition: (max_items > 0) && (item_size > 0)
   Post-condition: returns a pointer to an empty stack
*/

void Push( stack s, void *item );
/* Push an item onto a stack
   Pre-condition: (s is a stack created by a call to ConsStack) &&
                  (existing item count < max_items) &&
                  (item != NULL)
   Post-condition: item has been added to the top of s
*/

void *Pop( stack s );
/* Pop an item of a stack
   Pre-condition: (s is a stack created by a call to
                  ConsStack) &&
                  (existing item count >= 1)
   Post-condition: top item has been removed from s
*/
```

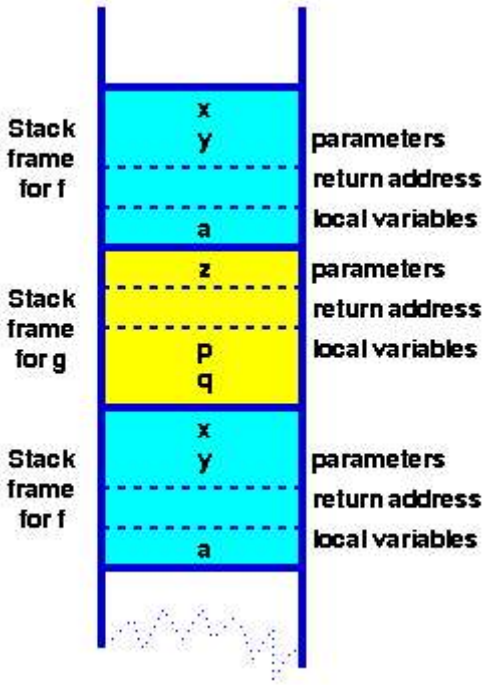
Points to note:
A stack is simply another collection of data items and thus it would be possible to use exactly the same specification as the one used for our general collection. However, collections with the LIFO semantics of stacks are so important in computer science that it is appropriate to set up a limited specification appropriate to stacks only. Although a linked list implementation of a stack is possible (adding and deleting from the head of a linked list produces exactly the LIFO semantics of a stack), the most common applications for stacks have a space restraint so that using an array implementation is a natural and efficient one (In most operating systems, allocation and de-allocation of memory is a relatively expensive operation, there is a penalty for the flexibility of linked list implementations.).

3.3.1 Stack Frames

Almost invariably, programs compiled from modern high level languages (even C!) make use of a stack frame for the working memory of each procedure or function invocation. When any procedure or function is called, a number of words - the stack frame - is pushed onto a program stack. When the procedure or function returns, this frame of data is popped off the stack.

As a function calls another function, first its arguments, then the return address and

finally space for local variables is pushed onto the stack. Since each function runs in its own "environment" or context, it becomes possible for a function to call itself - a technique known as recursion. This capability is extremely useful and extensively used - because many problems are elegantly specified or solved in a recursive way.



Program stack after executing a pair of mutually recursive functions:

```
function f(int x, int y) {
   int a;
   if ( term_cond ) return ...;
   a = .....;
   return g(a);
   }

function g(int z) {
   int p,q;
   p = ...; q = ...;
   return f(p,q);
   }
```

Note how all of function f and g's environment (their parameters and local variables) are found in the stack frame. When f is called a second time from g, a new frame for the second invocation of f is created.

Key terms
push, pop
Generic terms for adding something to, or removing something from a stack
context
The environment in which a function executes: includes argument values, local variables and global variables. All the context except the global variables is stored in a stack frame.

stack frames
The data structure containing all the data (arguments, local variables, return address, etc) needed each time a procedure or function is called.

3.4 Recursion

Many examples of the use of recursion may be found: the technique is useful both for the definition of mathematical functions and for the definition of data structures. Naturally, if a data structure may be defined recursively, it may be processed by a recursive function!
recur
From the Latin, re- = back +
currere = to run
To happen again, esp at repeated intervals.

3.4.1 Recursive functions

Many mathematical functions can be defined recursively:
factorial
Fibonacci
Euclid's GCD (greatest common denominator)
Fourier Transform
Many problems can be solved recursively, eg games of all types from simple ones like the Towers of Hanoi problem to complex ones like chess. In games, the recursive solutions are particularly convenient because, having solved the problem by a series of recursive calls, you want to find out how you got to the solution. By keeping track of the move chosen at any point, the program call stack does this housekeeping for you! This is explained in more detail later.

3.4.2 Example: Factorial

One of the simplest examples of a recursive definition is that for the factorial function:

```
factorial( n ) = if ( n = 0 ) then 1
                 else n * factorial( n-1 )
```

A natural way to calculate factorials is to write a recursive function which matches this definition:

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

Note how this function calls itself to evaluate the next term. Eventually it will reach the termination condition and exit. However, before it reaches the termination condition, it will have pushed n stack frames onto the program's run-time stack.

The termination condition is obviously extremely important when dealing with recursive functions. If it is omitted, then the function will continue to call itself until the program runs out of stack space - usually with moderately unpleasant results!

Failure to include a correct termination condition in a recursive function is a recipe for disaster!
Another commonly used (and abused!) example of a recursive function is the calculation of Fibonacci numbers. Following the definition:

```
fib( n ) = if ( n = 0 ) then 1
         if ( n = 1 ) then 1
         else fib( n-1 ) + fib( n-2 )
```

one can write:

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

Short and elegant, it uses recursion to provide a neat solution - that is actually a disaster! We shall re-visit this and show why it is such a disaster later.

Data structures also may be recursively defined. One of the most important class of structure - trees - allows recursive definitions which lead to simple (and efficient) recursive functions for manipulating them.

But in order to see why trees are valuable structures, let's first examine the problem of searching.

Key terms
Termination condition
Condition which terminates a series of recursive calls - and prevents the program from running out of space for stack frames!

4 Searching

Computer systems are often used to store large amounts of data from which individual records must be retrieved according to some search criterion. Thus the efficient storage of data to facilitate fast searching is an important issue. In this section, we shall investigate the performance of some searching algorithms and the data structures which they use.

4.1 Sequential Searches

Let's examine how long it will take to find an item matching a key in the collections we have discussed so far. We're interested in:

the average time
the worst-case time and
the best possible time.

However, we will generally be most concerned with the worst-case time as calculations based on worst-case times can lead to guaranteed performance predictions. Conveniently, the worst-case times are generally easier to calculate than average times.

If there are n items in our collection - whether it is stored as an array or as a linked list - then it is obvious that in the worst case, when there is no item in the collection with the desired key, then n comparisons of the key with keys of the items in the collection will have to be made.

To simplify analysis and comparison of algorithms, we look for a dominant operation and count the number of times that dominant operation has to be performed. In the case of searching, the dominant operation is the comparison, since the search requires n comparisons in the worst case, we say this is a O(n) (pronounce this "big-Oh-n" or "Oh-n") algorithm. The best case - in which the first comparison returns a match - requires a single comparison and is O(1). The average time depends on the probability that the key will be found in the collection - this is something that we would not expect to know in the majority of cases. Thus in this case, as in most others, estimation of the average time is of little utility. If the performance of the system is vital, i.e. it's part of a life-critical system, then we must use the worst case in our design calculations as it represents the best guaranteed performance.

4.2 Binary Search

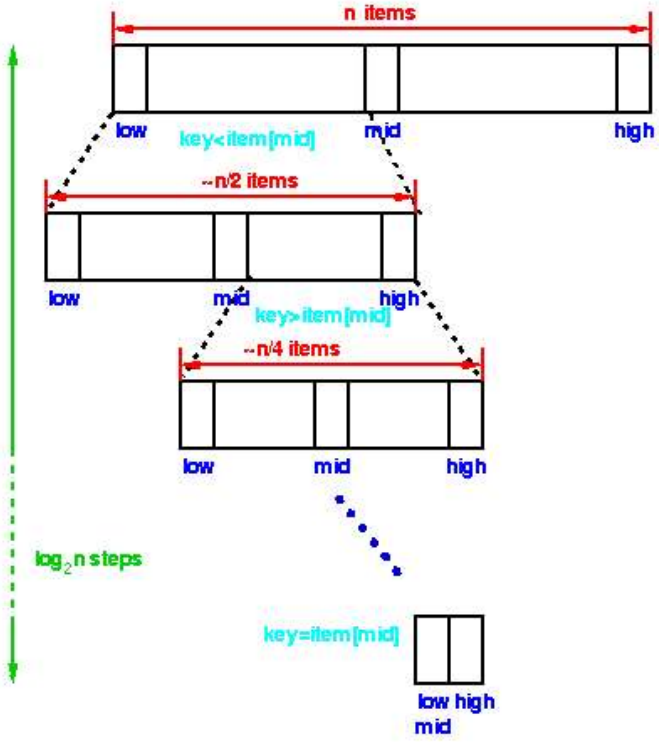
However, if we place our items in an array and sort them in either ascending or descending order on the key first, then we can obtain much better performance with an algorithm called binary search.

In binary search, we first compare the key with the item in the middle position of the array. If there's a match, we can return immediately. If the key is less than the middle key, then the item sought must lie in the lower half of the array; if it's greater then the item sought must lie in the upper half of the array. So we repeat the procedure on the lower (or upper) half of the array.

Our FindInCollection function can now be implemented:

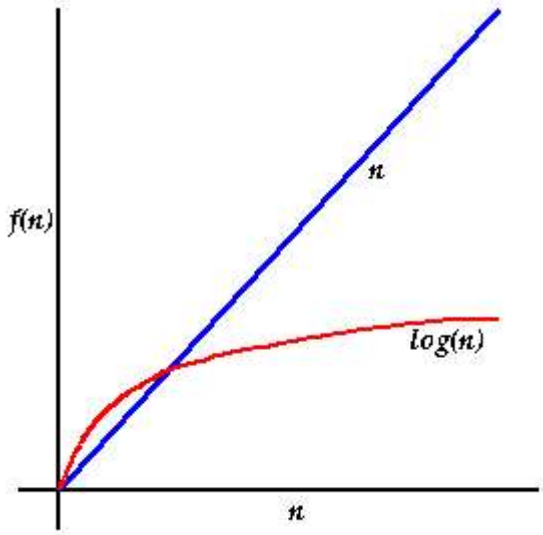
```
static void *bin_search( collection c, int low, int high, void *key ) {
int mid;
/* Termination check */
if (low > high) return NULL;
mid = (high+low)/2;
switch (memcmp(ItemKey(c->items[mid]),key,c->size)) {
/* Match, return item found */
case 0: return c->items[mid];
/* key is less than mid, search lower half */
case -1: return bin_search( c, low, mid-1, key);
/* key is greater than mid, search upper half */
case 1: return bin_search( c, mid+1, high, key );
default : return NULL;
}
}

void *FindInCollection( collection c, void *key ) {
/* Find an item in a collection
```

Pre-condition:
c is a collection created by ConsCollection
c is sorted in ascending order of the key
key != NULL
Post-condition: returns an item identified by key if
one exists, otherwise returns NULL
*/
int low, high;
low = 0; high = c->item_cnt-1;
return bin_search(c, low, high, key);
}
Points to note:
bin_search is recursive: it determines whether the search key lies in the lower or upper half of the array, then calls itself on the appropriate half.
There is a termination condition (two of them in fact!)
If low > high then the partition to be searched has no elements in it and
If there is a match with the element in the middle of the current partition, then we can return immediately.
AddToCollection will need to be modified to ensure that each item added is placed in its correct place in the array. The procedure is simple:
Search the array until the correct spot to insert the new item is found,
Move all the following items up one position and
Insert the new item into the empty position thus created.
bin_search is declared static. It is a local function and is not used outside this class: if it were not declared static, it would be exported and be available to all parts of the program. The static declaration also allows other classes to use the same name internally.
static reduces the visibility of a function an should be used wherever possible to control access to functions!
Analysis



Each step of the algorithm divides the block of items being searched in half. We can divide a set of n items in half at most log2 n times.

Thus the running time of a binary search is proportional to log n and we say this is a O (log n) algorithm.
Binary search requires a more complex program than our original search and thus for small n it may run slower than the simple linear search. However, for large n,



Thus at large n, log n is much smaller than n, consequently an O(log n) algorithm is much faster than an O(n) one.

Plot of n and log n vs n .

We will examine this behaviour more formally in a later section. First, let's see what we can do about the insertion (AddToCollection) operation.

In the worst case, insertion may require n operations to insert into a sorted list.

We can find the place in the list where the new item belongs using binary search in O (log n) operations.
However, we have to shuffle all the following items up one place to make way for the new one. In the worst case, the new item is the first in the list, requiring n move operations for the shuffle!

A similar analysis will show that deletion is also an O(n) operation.

If our collection is static, ie it doesn't change very often - if at all - then we may not be concerned with the time required to change its contents: we may be prepared for the initial build of the collection and the occasional insertion and deletion to take some time. In return, we will be able to use a simple data structure (an array) which has little memory overhead.

However, if our collection is large and dynamic, ie items are being added and deleted

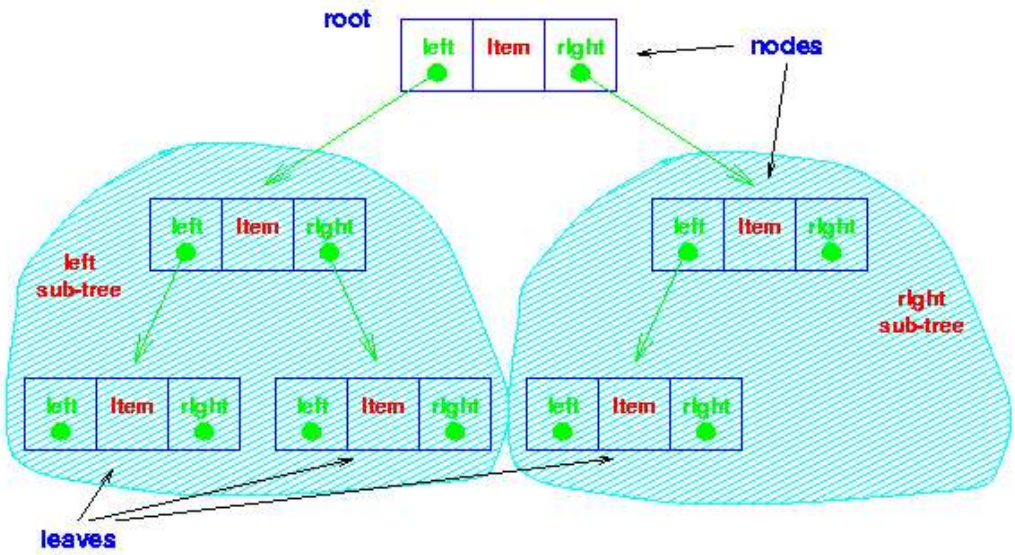
continually, then we can obtain considerably better performance using a data structure called a tree.

Key terms
Big Oh
A notation formally describing the set of all functions which are bounded above by a nominated function.
Binary Search
A technique for searching an ordered list in which we first check the middle item and - based on that comparison - "discard" half the data. The same procedure is then applied to the remaining half until a match is found or there are no more items left.

4.3 Trees

4.3.1 Binary Trees
The simplest form of tree is a binary tree. A binary tree consists of
a node (called the root node) and
left and right sub-trees.
Both the sub-trees are themselves binary trees.

You now have a recursively defined data structure. (It is also possible to define a list recursively: can you see how?)



A binary tree

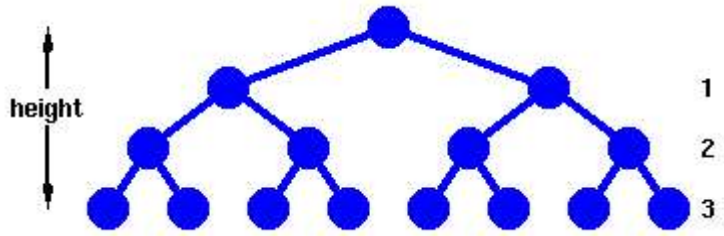
The nodes at the lowest levels of the tree (the ones with no sub-trees) are called leaves.

In an ordered binary tree,
the keys of all the nodes in the left sub-tree are less than that of the root,
the keys of all the nodes in the right sub-tree are greater than that of the root,
the left and right sub-trees are themselves ordered binary trees.
Data Structure
The data structure for the tree implementation simply adds left and right pointers in

place of the next pointer of the linked list implementation. [Load the tree struct.]

The AddToCollection method is, naturally, recursive. [Load the AddToCollection method.]

Similarly, the FindInCollection method is recursive. [Load the FindInCollection method.]
Analysis
Complete Trees



Before we look at more general cases, let's make the optimistic assumption that we've managed to fill our tree neatly, ie that each leaf is the same 'distance' from the root.

A complete tree
This forms a complete tree, whose height is defined as the number of links from the root to the deepest leaf.

First, we need to work out how many nodes, n, we have in such a tree of height, h.

Now,

n = 1 + 21 + 22 + ... + 2h

From which we have,
n = 2h+1 - 1

and
h = floor(log2n)

Examination of the Find method shows that in the worst case, h+1 or ceiling(log2n) comparisons are needed to find an item. This is the same as for binary search.

However, Add also requires ceiling(log2n) comparisons to determine where to add an item. Actually adding the item takes a constant number of operations, so we say that a binary tree requires O(logn) operations for both adding and finding an item - a considerable improvement over binary search for a dynamic structure which often requires addition of new items.

Deletion is also an O(logn) operation.
General binary trees
However, in general addition of items to an ordered tree will not produce a complete tree. The worst case occurs if we add an ordered list of items to a tree.

What will happen? Think before you click here!

This problem is readily overcome: we use a structure known as a heap. However, before looking at heaps, we should formalise our ideas about the complexity of algorithms by defining carefully what O(f(n)) means.

Key terms
Root Node
Node at the "top" of a tree - the one from which all operations on the tree commence. The root node may not exist (a NULL tree with no nodes in it) or have 0, 1 or 2 children in a binary tree.
Leaf Node
Node at the "bottom" of a tree - farthest from the root. Leaf nodes have no children.
Complete Tree
Tree in which each leaf is at the same distance from the root. A more precise and formal definition of a complete tree is set out later.
Height
Number of nodes which must be traversed from the root to reach a leaf of a tree.

5. Complexity

Rendering mathematical symbols with HTML is really painful!
Please don't suggest latex2html .. its tendency to put every symbol in an individual GIF file makes it equally painful!

Please load the postscript file instead - you will need a postscript viewer.

6 Queues

Queues are dynamic collections which have some concept of order. This can be either based on order of entry into the queue - giving us First-In-First-Out (FIFO) or Last-In-First-Out (LIFO) queues. Both of these can be built with linked lists: the simplest "add-to-head" implementation of a linked list gives LIFO behaviour. A minor modification - adding a tail pointer and adjusting the addition method implementation - will produce a FIFO queue.

Performance
A straightforward analysis shows that for both these cases, the time needed to add or delete an item is constant and independent of the number of items in the queue. Thus we class both addition and deletion as an O(1) operation. For any given real machine+operating system+language combination, addition may take c1 seconds and deletion c2 seconds, but we aren't interested in the value of the constant, it will vary from machine to machine, language to language, etc. The key point is that the time is not dependent on n - producing O(1) algorithms.

Once we have written an O(1) method, there is generally little more that we can do from an algorithmic point of view. Occasionally, a better approach may produce a lower constant time. Often, enhancing our compiler, run-time system, machine, etc will produce some significant improvement. However O(1) methods are already very

fast, and it's unlikely that effort expended in improving such a method will produce much real gain!

5.1 Priority Queues

Often the items added to a queue have a priority associated with them: this priority determines the order in which they exit the queue - highest priority items are removed first.

This situation arises often in process control systems. Imagine the operator's console in a large automated factory. It receives many routine messages from all parts of the system: they are assigned a low priority because they just report the normal functioning of the system - they update various parts of the operator's console display simply so that there is some confirmation that there are no problems. It will make little difference if they are delayed or lost.

However, occasionally something breaks or fails and alarm messages are sent. These have high priority because some action is required to fix the problem (even if it is mass evacuation because nothing can stop the imminent explosion!).

Typically such a system will be composed of many small units, one of which will be a buffer for messages received by the operator's console. The communications system places messages in the buffer so that communications links can be freed for further messages while the console software is processing the message. The console software extracts messages from the buffer and updates appropriate parts of the display system. Obviously we want to sort messages on their priority so that we can ensure that the alarms are processed immediately and not delayed behind a few thousand routine messages while the plant is about to explode.

As we have seen, we could use a tree structure - which generally provides O(logn) performance for both insertion and deletion. Unfortunately, if the tree becomes unbalanced, performance will degrade to O(n) in pathological cases. This will probably not be acceptable when dealing with dangerous industrial processes, nuclear reactors, flight control systems and other life-critical systems.

Aside

The great majority of computer systems would fall into the broad class of information systems - which simply store and process information for the benefit of people who make decisions based on that information. Obviously, in such systems, it usually doesn't matter whether it takes 1 or 100 seconds to retrieve a piece of data - this simply determines whether you take your coffee break now or later. However, as we'll see, using the best known algorithms is usually easy and straight-forward: if they're not already coded in libaries, they're in text-books. You don't even have to work out how to code them! In such cases, it's just your reputation that's going to suffer if someone (who has studied his or her algorithms text!) comes along later and says "Why on earth did X (you!) use this O(n2) method -
there's a well known O(n) one!"
Of course, hardware manufacturers are very happy if you use inefficient algorithms - it drives the demand for new, faster hardware - and keeps their profits high!

There is a structure which will provide guaranteed O(logn) performance for both

insertion and deletion: it's called a heap.

Key terms
FIFO queue
A queue in which the first item added is always the first one out.
LIFO queue
A queue in which the item most recently added is always the first one out.
Priority queue
A queue in which the items are sorted so that the highest priority item is always the next one to be extracted.
Life critical systems
Systems on which we depend for safety and which may result in death or injury if they fail: medical monitoring, industrial plant monitoring and control and aircraft control systems are examples of life critical systems.
Real time systems
Systems in which time is a constraint. A system which must respond to some event (eg the change in attitude of an aircraft caused by some atmospheric event like wind-shear) within a fixed time to maintain stability or continue correct operation (eg the aircraft systems must make the necessary adjustments to the control surfaces before the aircraft falls out of the sky!).

6.2 Heaps

Heaps are based on the notion of a complete tree, for which we gave an informal definition earlier.

Formally:

A binary tree is completely full if it is of height, h, and has 2h+1-1 nodes.

A binary tree of height, h, is complete iff
it is empty or
its left subtree is complete of height h-1 and its right subtree is completely full of height h-2 or
its left subtree is completely full of height h-1 and its right subtree is complete of height h-1.
A complete tree is filled from the left:
all the leaves are on
the same level or
two adjacent ones and
all nodes at the lowest level are as far to the left as possible.
Heaps
A binary tree has the heap property iff
it is empty or
the key in the root is larger than that in either child and both subtrees have the heap property.
A heap can be used as a priority queue: the highest priority item is at the root and is trivially extracted. But if the root is deleted, we are left with two sub-trees and we must efficiently re-create a single tree with the heap property.
The value of the heap structure is that we can both extract the highest priority item

and insert a new one in O(logn) time.

How do we do this?

Let's start with this heap.

A deletion will remove the T
at the root.

To work out how we're going to maintain the heap property, use the fact that a complete tree is filled from the left. So that the position which must become empty is the one occupied by the M.

Put it in the vacant root position.

This has violated the condition that the root must be greater than each of its children.

So interchange the M with the larger of its children.

The left subtree has now lost the heap property.

So again interchange the M with the larger of its children.

This tree is now a heap again, so we're finished.

We need to make at most h interchanges of a root of a subtree with one of its children to fully restore the heap property. Thus deletion from a heap is O(h) or O(logn).
Addition to a heap
To add an item to a heap, we follow the reverse procedure.

Place it in the next leaf position and move it up.

Again, we require O(h) or O(logn) exchanges.

Storage of complete trees
The properties of a complete tree lead to a very efficient storage mechanism using n sequential locations in an array.
If we number the nodes from 1 at the root and place:
the left child of node k at position 2k
the right child of node k at position 2k+1

Then the 'fill from the left' nature of the complete tree ensures that the heap can be stored in consecutive locations in an array.
Viewed as an array, we can see that the nth node is always in index position n.

The code for extracting the highest priority item from a heap is, naturally, recursive. Once we've extracted the root (highest priority) item and swapped the last item into its

place, we simply call MoveDown recursively until we get to the bottom of the tree.

Click here to load heap_delete.c

Note the macros LEFT and RIGHT which simply encode the relation between the index of a node and its left and right children. Similarly the EMPTY macro encodes the rule for determining whether a sub-tree is empty or not.

Inserting into a heap follows a similar strategy, except that we use a MoveUp function to move the newly added item to its correct place. (For the MoveUp function, a further macro which defines the PARENT of a node would normally be added.)

Heaps provide us with a method of sorting, known as heapsort. However, we will examine and analyse the simplest method of sorting first.

Animation
In the animation, note that both the array representation (used in the implementation of the algorithm) and the (logical) tree representation are shown. This is to demonstrate how the tree is restructured to make a heap again after every insertion or deletion.

Sorting

Sorting is one of the most important operations performed by computers. In the days of magnetic tape storage before modern data-bases, it was almost certainly the most common operation performed by computers as most "database" updating was done by sorting transactions and merging them with a master file. It's still important for presentation of data extracted from databases: most people prefer to get reports sorted into some relevant order before wading through pages of data!

7.1Bubble, Selection, Insertion Sorts

There are a large number of variations of one basic strategy for sorting. It's the same strategy that you use for sorting your bridge hand. You pick up a card, start at the beginning of your hand and find the place to insert the new card, insert it and move all the others up one place.

```
/* Insertion sort for integers */

void insertion( int a[], int n ) {
/* Pre-condition: a contains n items to be sorted */
   int i, j, v;
   /* Initially, the first item is considered 'sorted' */
   /* i divides a into a sorted region, x<i, and an
      unsorted one, x >= i */
   for(i=1;i<n;i++) {
      /* Select the item at the beginning of the
         as yet unsorted section */
      v = a[i];
      /* Work backwards through the array, finding where v
```

```
        should go */
      j = i;
      /* If this element is greater than v,
           move it up one */
      while ( a[j-1] > v ) {
        a[j] = a[j-1]; j = j-1;
        if ( j <= 0 ) break;
        }
      /* Stopped when a[j-1] <= v, so put v at position j */
      a[j] = v;
      }
   }
```

Insertion Sort Animation

Bubble Sort

Another variant of this procedure, called bubble sort, is commonly taught:

```
/* Bubble sort for integers */
#define SWAP(a,b)   { int t; t=a; a=b; b=t; }

void bubble( int a[], int n )
/* Pre-condition: a contains n items to be sorted */
   {
   int i, j;
   /* Make n passes through the array */
   for(i=0;i<n;i++)
      {
      /* From the first element to the end
         of the unsorted section */
      for(j=1;j<(n-i);j++)
         {
         /* If adjacent items are out of order, swap them */
         if( a[j-1]>a[j] ) SWAP(a[j-1],a[j]);
         }
      }
   }
```

Analysis

Each of these algorithms requires n-1 passes: each pass places one item in its correct place. (The nth is then in the correct place also.) The ith pass makes either ior n - i comparisons and moves. So:

or O(n2) - but we already know we can use heaps to get an O(n logn) algorithm. Thus these algorithms are only suitable for small problems where their simple code makes them faster than the more complex code of the O(n logn) algorithm. As a rule of thumb, expect to find an O(n logn) algorithm faster for n>10 - but the exact value depends very much on individual machines!.

They can be used to squeeze a little bit more performance out of fast sort algorithms - see later.

Key terms
Bubble, Insertion, Selection Sorts
Simple sorting algorithms with O(n2) complexity - suitable for sorting small numbers of items only.

7.2 Heap Sort

We noted earlier, when discussing heaps, that, as well as their use in priority queues, they provide a means of sorting:
construct a heap,
add each item to it (maintaining the heap property!),
when all items have been added, remove them one by one (restoring the heap property as each one is removed).
Addition and deletion are both O(logn) operations. We need to perform n additions and deletions, leading to an O(nlogn) algorithm. We will look at another efficient sorting algorithm, Quicksort, and then compare it with Heap sort.
Animation
The following animation uses a slight modification of the above approach to sort directly using a heap. You will note that it places all the items into the array first, then takes items at the bottom of the heap and restores the heap property, rather than restoring the heap property as each item is entered as the algorithm above suggests. (This approach is described more fully in Cormen et al.)

Note that the animation shows the data
stored in an array (as it is in the implementation of the algorithm) and also
in the tree form - so that the heap structure can be clearly seen.
Both representations are, of course, equivalent.

7.3Quick Sort

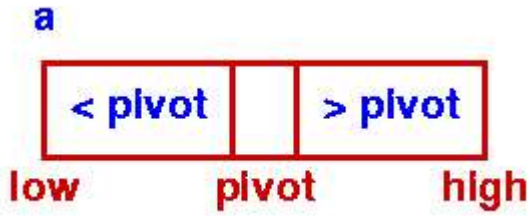
Quicksort is a very efficient sorting algorithm invented by C.A.R. Hoare. It has two phases:
the partition phase and
the sort phase.
As we will see, most of the work is done in the partition phase - it works out where to divide the work. The sort phase simply sorts the two smaller problems that are generated in the partition phase.

This makes Quicksort a good example of the divide and conquer strategy for solving problems. (You've already seen an example of this approach in the binary search procedure.) In quicksort, we divide the array of items to be sorted into two partitions and then call the quicksort procedure recursively to sort the two partitions, ie we divide the problem into two smaller ones and conquer by solving the smaller ones. Thus the conquer part of the quicksort routine looks like this:

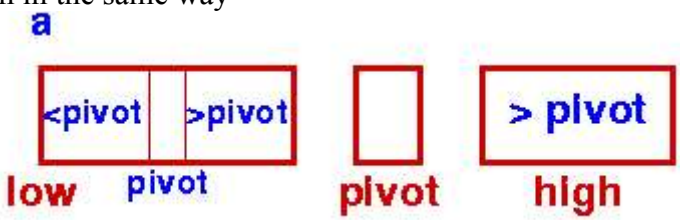
```
quicksort( void *a, int low, int high )
  {
  int pivot;
  /* Termination condition! */
  if ( high > low )
```

```
  {
  pivot = partition( a, low, high );
  quicksort( a, low, pivot-1 );
  quicksort( a, pivot+1, high );
  }
}
```

Initial Step - First Partition



Sort Left Partition in the same way



For the strategy to be effective, the partition phase must ensure that all the items in one part (the lower part) and less than all those in the other (upper) part.

To do this, we choose a pivot element and arrange that all the items in the lower part are less than the pivot and all those in the upper part greater than it. In the most general case, we don't know anything about the items to be sorted, so that any choice of the pivot element will do - the first element is a convenient one.

As an illustration of this idea, you can view this animation, which shows a partition algorithm in which items to be sorted are copied from the original array to a new one: items smaller than the pivot are placed to the left of the new array and items greater than the pivot are placed on the right. In the final step, the pivot is dropped into the remaining slot in the middle.
QuickSort Animation
This animation was based on a suggestion made by Jeff Rohl;
it was written by Woi Ang.

Observe that the animation uses two arrays for the items being sorted: thus it requires O(n) additional space to operate. However, it's possible to partition the array in place. The next page shows a conventional implementation of the partition phase which swaps elements in the same array and thus avoids using extra space.

Key terms
Divide and Conquer Algorithms
Algorithms that solve (conquer) problems by dividing them into smaller sub-problems until the problem is so small that it is trivially solved.

in place
In place sorting algorithms don't require additional temporary space to store elements as they sort; they use the space originally occupied by the elements.

7.4 Bin Sort

Assume that
the keys of the items that we wish to sort lie in a small fixed range and
that there is only one item with each value of the key.
Then we can sort with the following procedure:
Set up an array of "bins" - one for each value of the key - in order,
Examine each item and use the value of the key to place it in the appropriate bin.
Now our collection is sorted and it only took n operations, so this is an O(n) operation. However, note that it will only work under very restricted conditions.

Constraints on bin sort
To understand these restrictions, let's be a little more precise about the specification of the problem and assume that there are m values of the key. To recover our sorted collection, we need to examine each bin. This adds a third step to the algorithm above,
Examine each bin to see whether there's an item in it.
which requires m operations. So the algorithm's time becomes:
$T(n) = c1n + c2m$
and it is strictly O(n + m). Now if m <= n, this is clearly O(n). However if m >> n, then it is O(m).

For example, if we wish to sort 104 32-bit integers, then m = 232 and we need 232 operations (and a rather large memory!).
For n = 104:

nlogn ~ 104 x 13 ~ 213 x 24 ~ 217
So quicksort or heapsort would clearly be preferred.

An implementation of bin sort might look like:

```
#define EMPTY-1 /* Some convenient flag */
void bin_sort( int *a, int *bin, int n ) {
   int i;
   /* Pre-condition: for 0<=i<n : 0 <= a[i] < M */
   /* Mark all the bins empty */
   for(i=0;i<M;i++) bin[i] = EMPTY;
   for(i=0;i<n;i++)
      bin[ a[i] ] = a[i];
   }

main() {
   int a[N], bin[M];    /* for all i: 0 <= a[i] < M */
   .... /* Place data in a */
   bin_sort( a, bin, N );
```

If there are duplicates, then each bin can be replaced by a linked list. The third step then becomes:
Link all the lists into one list.
We can add an item to a linked list in O(1) time. There are n items requiring O(n) time. Linking a list to another list simply involves making the tail of one list point to the other, so it is O(1). Linking m such lists obviously takes O(m) time, so the algorithm is still O(n+m).

In contrast to the other sorts, which sort in place and don't require additional memory, bin sort requires additional memory for the bins and is a good example of trading space for performance.

Although memory tends to be cheap in modern processors -
so that we would normally use memory rather profligately to obtain performance,
memory consumes power
and in some circumstances, eg computers in space craft,
power might be a higher constraint than performance.

Having highlighted this constraint, there is a version of bin sort which can sort in place:

```
#define EMPTY-1 /* Some convenient flag */
void bin_sort( int *a, int n ) {
   int i;
   /* Pre-condition: for 0<=i<n : 0 <= a[i] < n */
   for(i=0;i<n;i++)
if ( a[i] != i )
        SWAP( a[i], a[a[i]] );
   }
```

However, this assumes that there are n distinct keys in the range 0 .. n-1. In addition to this restriction, the SWAP operation is relatively expensive, so that this version trades space for time.

The bin sorting strategy may appear rather limited, but it can be generalised into a strategy known as Radix sorting.

7.5 Radix Sorting

The bin sorting approach can be generalised in a technique that is known as radix sorting.

An example
Assume that we have n integers in the range (0,n2) to be sorted. (For a bin sort, m = n2, and we would have an O(n+m) = O(n2) algorithm.) Sort them in two phases:
Using n bins, place ai into bin ai mod n,
Repeat the process using n bins, placing ai into bin floor(ai/n), being careful to append to the end of each bin.
This results in a sorted list.

As an example, consider the list of integers:

36 9 0 25 1 49 64 16 81 4
n is 10 and the numbers all lie in (0,99). After the first phase, we will have:

Bin	0	1	2	3	4	5	6	7	8	9
Content		0	1							
81	-	-	64							
4	25	36								
16	-	-	9							
49										

Note that in this phase, we placed each item in a bin indexed by the least significant decimal digit.

Repeating the process, will produce:

Bin	0	1	2	3	4	5	6	7	8	9
Content		0								
1										
4										
9	16	25	36	49	-	64	-	81	-	

In this second phase, we used the leading decimal digit to allocate items to bins, being careful to add each item to the end of the bin.

We can apply this process to numbers of any size expressed to any suitable base or radix.

7.5.1 Generalised Radix Sorting

We can further observe that it's not necessary to use the same radix in each phase, suppose that the sorting key is a sequence of fields, each with bounded ranges, eg the key is a date using the structure:

```
typedef struct t_date {
   int day;
   int month;
   int year;
} date;
```

If the ranges for day and month are limited in the obvious way, and the range for year is suitably constrained, eg 1900 < year <= 2000, then we can apply the same procedure except that we'll employ a different number of bins in each phase. In all cases, we'll sort first using the least significant "digit" (where "digit" here means a field with a limited range), then using the next significant "digit", placing each item after all the items already in the bin, and so on.

Assume that the key of the item to be sorted has k fields, fi|i=0..k-1, and that each fi has si discrete values, then a generalised radix sort procedure can be written:

```
radixsort( A, n ) {
   for(i=0;i<k;i++) {
      for(j=0;j<si;j++) bin[j] = EMPTY;
O(si)

      for(j=0;j<n;j++) {
         move Ai
         to the end of bin[Ai->fi]
         }
O(n)

      for(j=0;j<si;j++)
         concatenate bin[j] onto the end of A;
      }
}
O(si)
Total
```

Now if, for example, the keys are integers in (0,bk-1), for some constant k, then the keys can be viewed as k-digit base-b integers.
Thus, si = b for all i and the time complexity becomes O(n+kb) or O(n). This result depends on k being constant.

If k is allowed to increase with n, then we have a different picture. For example, it takes log2n binary digits to represent an integer <n. If the key length were allowed to increase with n, so that k = logn, then we would have:
.

Another way of looking at this is to note that if the range of the key is restricted to (0,bk-1), then we will be able to use the radixsort approach effectively if we allow duplicate keys when n>bk. However, if we need to have unique keys, then k must increase to at least logbn. Thus, as n increases, we need to have logn phases, each taking O(n) time, and the radix sort is the same as quick sort!

Sample code
This sample code sorts arrays of integers on various radices: the number of bits used for each radix can be set with the call to SetRadices. The Bins class is used in each phase to collect the items as they are sorted. ConsBins is called to set up a set of bins: each bin must be large enough to accommodate the whole array, so RadixSort can be very expensive in its memory usage!
RadixSort.h
RadixSort.c
Bins.h
Bins.c

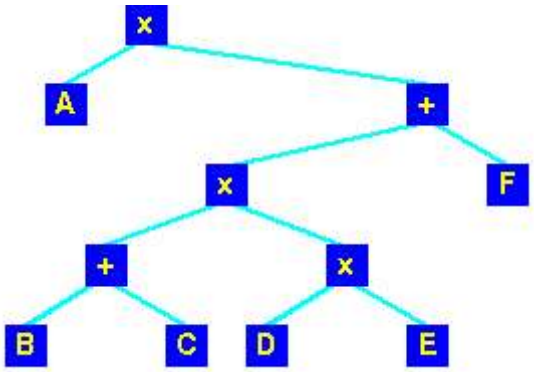
8 Searching Revisited

Before we examine some more searching techniques, we need to consider some operations on trees - in particular means of traversing trees.
Tree operations

A binary tree can be traversed in a number of ways:
pre-order
Visit the root
Traverse the left sub-tree,
Traverse the right sub-tree
in-order
Traverse the left sub-tree,
Visit the root
Traverse the right sub-tree
post-order
Traverse the left sub-tree,
Traverse the right sub-tree
Visit the root

If we traverse the standard ordered binary tree in-order, then we will visit all the nodes in sorted order.

Parse trees



If we represent the expression:

A*(((B+C)*(D*E))+F)

as a tree:

then traversing it post-order will produce:
A B C + D E * * F + *
which is the familiar reverse-polish notation used by a compiler for evaluating the expression.
Search Trees
We've seen how to use a heap to maintain a balanced tree for a priority queue. What about a tree used to store information for retrieval (but not removal)? We want to be able to find any item quickly in such a tree based on the value of its key. The search routine on a binary tree:

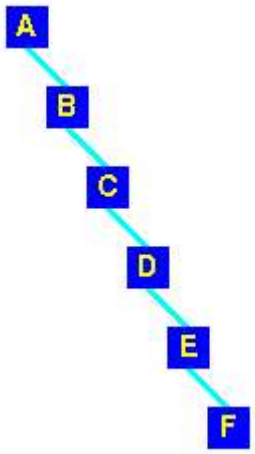
```
tree_search(tree T, Key key) {
   if (T == NULL) return NULL;
   if (key == T->root) return T->root;
   else
```

```
    if (key < T->root) return tree_search( T->left, key );
    else               return tree_search( T->right, key );
}
```

is simple and provides us with a O(log n) searching routine as long as we can keep the tree balanced. However, if we simply add items to a tree, producing an unbalanced tree is easy!
This is what happens if
we add the letters

A B C D E F

in that order to a tree:



Not exactly well balanced!

Key terms
Pre-order tree traversal
Traversing a tree in the order: root | left | right
In-order tree traversal
Traversing a tree in the order: left | root | right
Post-order tree traversal
Traversing a tree in the order: left | right | root

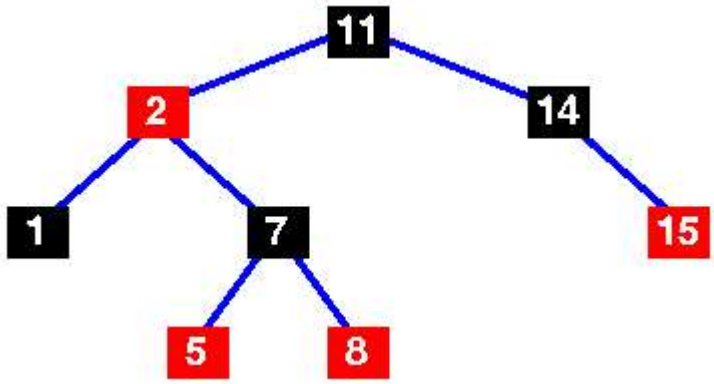
8.2 Red-Black Trees

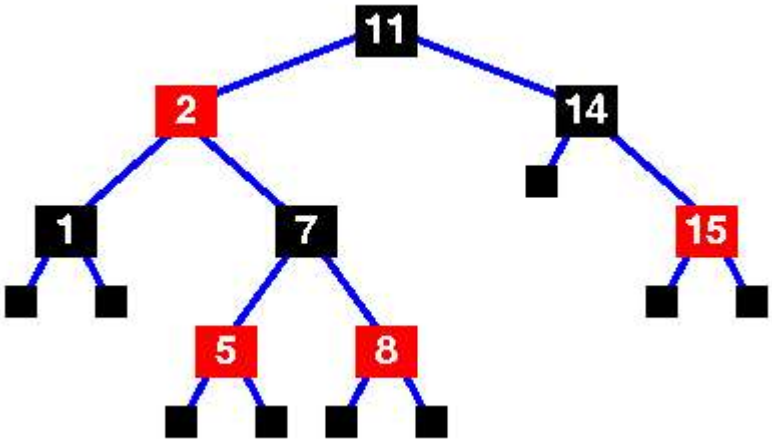
A red-black tree is a binary search tree with one extra attribute for each node: the colour, which is either red or black. We also need to keep track of the parent of each node, so that a red-black tree's node structure would be:

```
struct t_red_black_node {
   enum { red, black } colour;
   void *item;
   struct t_red_black_node *left,
                  *right,
                  *parent;
   }
```

For the purpose of this discussion, the NULL nodes which terminate the tree are considered to be the leaves and are coloured black.

Definition of a red-black tree





A red-black tree is a binary search tree which has the following red-black properties:
Every node is either red or black.
Every leaf (NULL) is black.
If a node is red, then both its children are black.
Every simple path from a node to a descendant leaf contains the same number of black nodes.
implies that on any path from the root to a leaf, red nodes must not be adjacent.
However, any number of black nodes may appear in a sequence.

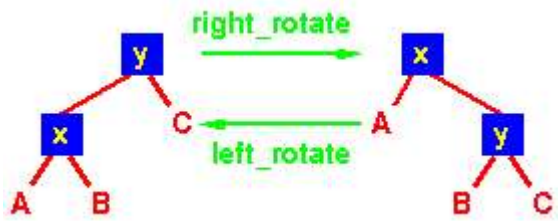
A basic red-black tree

Basic red-black tree with the sentinel nodes added. Implementations of the red-black tree algorithms will usually include the sentinel nodes as a convenient means of flagging that you have reached a leaf node.

They are the NULL black nodes of property 2.
The number of black nodes on any path from, but not including, a node x to a leaf is called the black-height of a node, denoted bh(x). We can prove the following lemma:
Lemma
A red-black tree with n internal nodes has height at most 2log(n+1).
(For a proof, see Cormen, p 264)

This demonstrates why the red-black tree is a good search tree: it can always be searched in O(log n) time.

As with heaps, additions and deletions from red-black trees destroy the red-black property, so we need to restore it. To do this we need to look at some operations on red-black trees.
Rotations



A rotation is a local operation in a search tree that preserves in-order traversal key ordering.

Note that in both trees, an in-order traversal yields:

A x B y C

The left_rotate operation may be encoded:

```
left_rotate( Tree T, node x ) {
   node y;
   y = x->right;
   /* Turn y's left sub-tree into x's right sub-tree */
   x->right = y->left;
   if ( y->left != NULL )
      y->left->parent = x;
   /* y's new parent was x's parent */
   y->parent = x->parent;
   /* Set the parent to point to y instead of x */
   /* First see whether we're at the root */
   if ( x->parent == NULL ) T->root = y;
   else
      if ( x == (x->parent)->left )
         /* x was on the left of its parent */
         x->parent->left = y;
      else
         /* x must have been on the right */
         x->parent->right = y;
```

```
   /* Finally, put x on y's left */
   y->left = x;
   x->parent = y;
   }
```

Insertion

Insertion is somewhat complex and involves a number of cases. Note that we start by inserting the new node, x, in the tree just as we would for any other binary tree, using the tree_insert function. This new node is labelled red, and possibly destroys the red-black property. The main loop moves up the tree, restoring the red-black property.

```
rb_insert( Tree T, node x ) {
   /* Insert in the tree in the usual way */
   tree_insert( T, x );
   /* Now restore the red-black property */
   x->colour = red;
   while ( (x != T->root) && (x->parent->colour == red) ) {
     if ( x->parent == x->parent->parent->left ) {
        /* If x's parent is a left, y is x's right 'uncle' */
        y = x->parent->parent->right;
        if ( y->colour == red ) {
           /* case 1 - change the colours */
           x->parent->colour = black;
           y->colour = black;
           x->parent->parent->colour = red;
           /* Move x up the tree */
           x = x->parent->parent;
           }
        else {
           /* y is a black node */
           if ( x == x->parent->right ) {
              /* and x is to the right */
              /* case 2 - move x up and rotate */
              x = x->parent;
              left_rotate( T, x );
              }
           /* case 3 */
           x->parent->colour = black;
           x->parent->parent->colour = red;
           right_rotate( T, x->parent->parent );
           }
        }
     else {
        /* repeat the "if" part with right and left
           exchanged */
        }
     }
   /* Colour the root black */
   T->root->colour = black;
   }
```

Here's an example of the insertion operation.

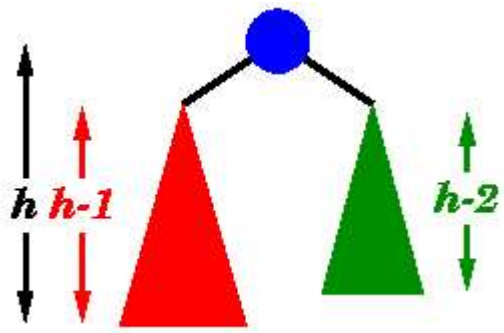
Animation
Red-Black Tree Animation

Examination of the code reveals only one loop. In that loop, the node at the root of the sub-tree whose red-black property we are trying to restore, x, may be moved up the tree at least one level in each iteration of the loop. Since the tree originally has O(log n) height, there are O(log n) iterations. The tree_insert routine also has O(log n) complexity, so overall the rb_insert routine also has O(log n) complexity.

Key terms
Red-black trees
Trees which remain balanced - and thus guarantee O(logn) search times - in a dynamic environment. Or more importantly, since any tree can be re-balanced - but at considerable cost - can be re-balanced in O(logn) time.

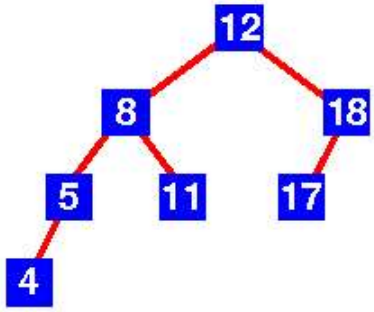
8.3 AVL Trees

An AVL tree is another balanced binary search tree. Named after their inventors, Adelson-Velskii and Landis, they were the first dynamically balanced trees to be proposed. Like red-black trees, they are not perfectly balanced, but pairs of sub-trees differ in height by at most 1, maintaining an O(logn) search time. Addition and deletion operations also take O(logn) time.
Definition of an AVL tree

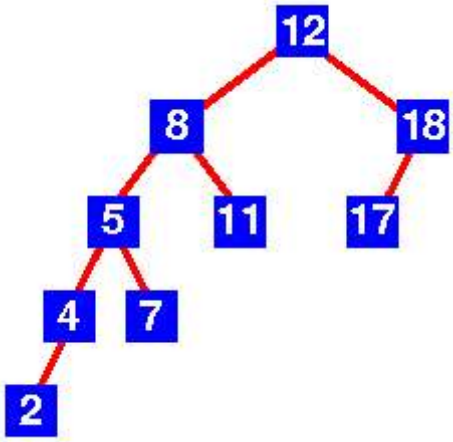


An AVL tree is a binary search tree which has the following properties:
The sub-trees of every node differ in height by at most one.
Every sub-tree is an AVL tree.

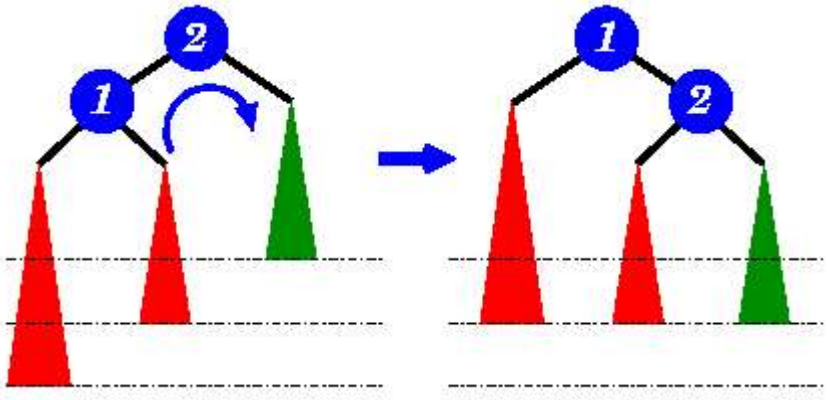
Balance requirement for an AVL tree: the left and right sub-trees differ by at most 1 in height.
You need to be careful with this definition: it permits some apparently unbalanced trees! For example, here are some trees:
Tree AVL tree?



Yes
Examination shows that each left sub-tree has a height 1 greater than each right sub-tree.



No
Sub-tree with root 8 has height 4 and sub-tree with root 18 has height 2
Insertion



As with the red-black tree, insertion is somewhat complex and involves a number of cases. Implementations of AVL tree insertion may be found in many textbooks: they rely on adding an extra attribute, the balance factor to each node. This factor indicates whether the tree is left-heavy (the height of the left sub-tree is 1 greater than the right sub-tree), balanced (both sub-trees are the same height) or right-heavy (the height of the right sub-tree is 1 greater than the left sub-tree). If the balance would be destroyed by an insertion, a rotation is performed to correct the balance.

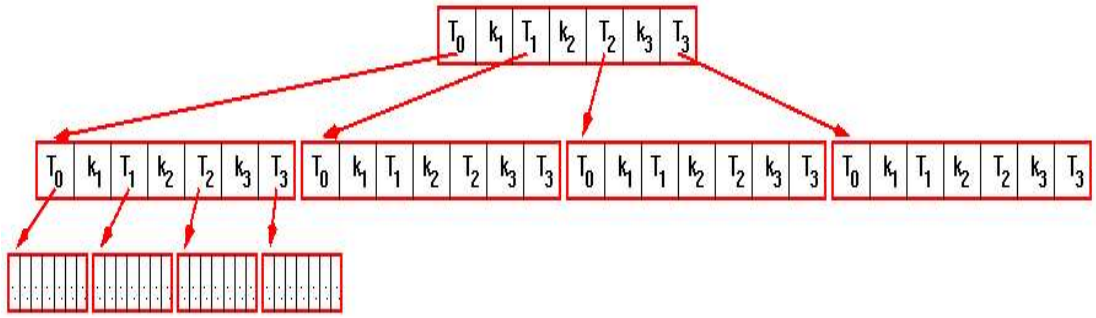
A new item has been added to the left subtree of node 1, causing its height to

become 2 greater than 2's right sub-tree (shown in green). A right-rotation is performed to correct the imbalance.

Key terms
AVL trees
Trees which remain balanced - and thus guarantee O(logn) search times - in a dynamic environment. Or more importantly, since any tree can be re-balanced - but at considerable cost - can be re-balanced in O(logn) time.

8.2 General n-ary trees

If we relax the restriction that each node can have only one key, we can reduce the height of the tree.
An m-way search tree
is empty or
consists of a root containing j (1<=j<m) keys, kj, and
a set of sub-trees, Ti, (i = 0..j), such that
if k is a key in T0, then k <= k1
if k is a key in Ti (0<i<j), then ki <= k <= ki+1
if k is a key in Tj, then k > kj and
all Ti are nonempty m-way search trees or all Ti are empty
Or in plain English ..
A node generally has m-1 keys and m children.
Each node has alternating sub-tree pointers and keys:
sub-tree | key | sub-tree | key | ... | key | sub_tree



All keys in a sub-tree to the left of a key are smaller than it.
All keys in the node between two keys are between those two keys.
All keys in a sub-tree to the right of a key are greater than it.
This is the "standard" recursive part of the definition.

The height of a complete m-ary tree with n nodes is ceiling(logmn).

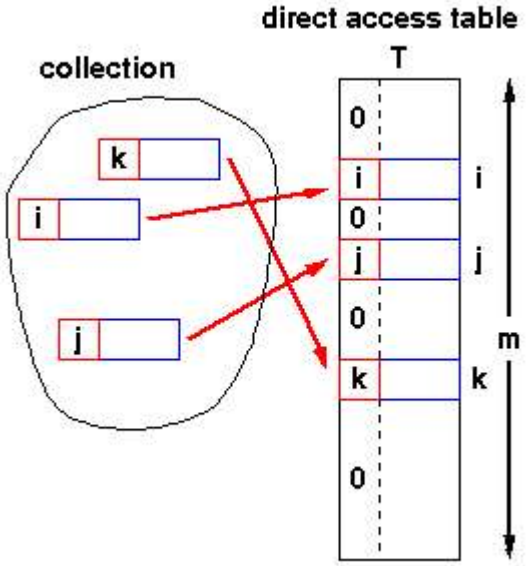
A B-tree of order m is an m-way tree in which
all leaves are on the same level and
all nodes except for the root and the leaves have at least m/2 children and at most m children. The root has at least 2 children and at most m children.
A variation of the B-tree, known as a B+-tree considers all the keys in nodes except the leaves as dummies. All keys are duplicated in the leaves. This has the advantage that is all the leaves are linked together sequentially, the entire tree may be scanned

without visiting the higher nodes at all.

Key terms
n-ary trees (or n-way trees)
Trees in which each node may have up to n children.
B-tree
Balanced variant of an n-way tree.
B+-tree
B-tree in which all the leaves are linked to facilitate fast in order traversal.

8.3 Hash Tables

8.3.1 Direct Address Tables
If we have a collection of n elements whose keys are unique integers in (1,m), where m >= n,
then we can store the items in a direct address table, T[m],
where Ti is either empty or contains one of the elements of our collection.

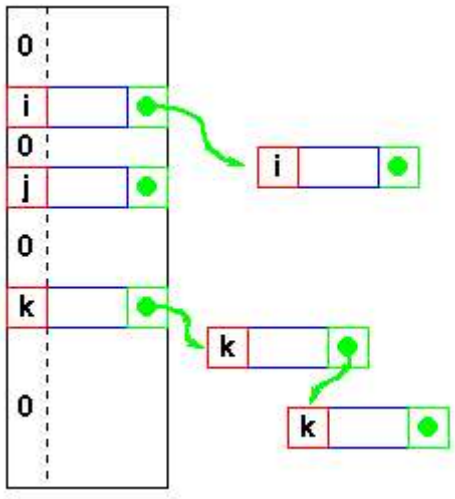


Searching a direct address table is clearly an O(1) operation:
for a key, k, we access Tk,
if it contains an element, return it,
if it doesn't then return a NULL.
There are two constraints here:
the keys must be unique, and
the range of the key must be severely bounded.

If the keys are not unique, then we can simply construct a set of m lists and store the heads of these lists in the direct address table. The time to find an element matching an input key will still be O(1).

However, if each element of the collection has some other distinguishing feature (other than its key), and if the maximum number of duplicates is ndupmax, then searching for a specific element is O(ndupmax). If duplicates are the exception rather

than the rule, then ndupmax is much smaller than n and a direct address table will provide good performance. But if ndupmax approaches n, then the time to find a specific element is O(n) and a tree structure will be more efficient.



The range of the key determines the size of the direct address table and may be too large to be practical. For instance it's not likely that you'll be able to use a direct address table to store elements which have arbitrary 32-bit integers as their keys for a few years yet!

Direct addressing is easily generalised to the case where there is a function,

h(k) => (1,m)

which maps each value of the key, k, to the range (1,m). In this case, we place the element in T[h(k)] rather than T[k] and we can search in O(1) time as before.

8.3.2 Mapping functions
The direct address approach requires that the function, h(k), is a one-to-one mapping from each k to integers in (1,m). Such a function is known as a perfect hashing function: it maps each key to a distinct integer within some manageable range and enables us to trivially build an O(1) search time table.

Unfortunately, finding a perfect hashing function is not always possible. Let's say that we can find a hash function, h(k), which maps most of the keys onto unique integers, but maps a small number of keys on to the same integer. If the number of collisions (cases where multiple keys map onto the same integer), is sufficiently small, then hash tables work quite well and give O(1) search times.
Handling the collisions
In the small number of cases, where multiple keys map to the same integer, then elements with different keys may be stored in the same "slot" of the hash table. It is clear that when the hash function is used to locate a potential match, it will be necessary to compare the key of that element with the search key. But there may be more than one element which should be stored in a single slot of the table. Various techniques are used to manage this problem:

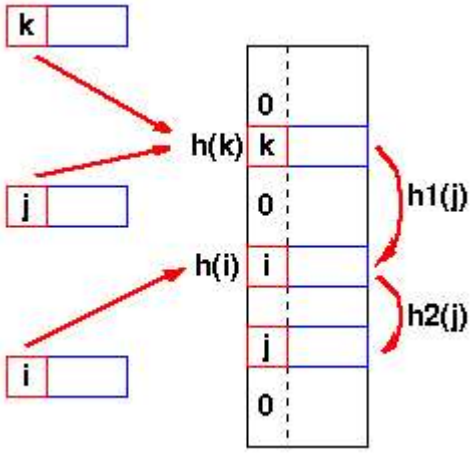
chaining,
overflow areas,
re-hashing,
using neighbouring slots (linear probing),
quadratic probing,
random probing, ...

Chaining
One simple scheme is to chain all collisions in lists attached to the appropriate slot. This allows an unlimited number of collisions to be handled and doesn't require a priori knowledge of how many elements are contained in the collection. The tradeoff is the same as with linked lists versus array implementations of collections: linked list overhead in space and, to a lesser extent, in time.
Re-hashing
Re-hashing schemes use a second hashing operation when there is a collision. If there is a further collision, we re-hash until an empty "slot" in the table is found.

The re-hashing function can either be a new function or a re-application of the original one. As long as the functions are applied to a key in the same order, then a sought key can always be located.
Linear probing
One of the simplest re-hashing functions is +1 (or -1), ie on a collision, look in the neighbouring slot in the table. It calculates the new address extremely quickly and may be extremely efficient on a modern RISC processor due to efficient cache utilisation (cf. the discussion of linked list efficiency).

The animation gives you a practical demonstration of the effect of linear probing: it also implements a quadratic re-hash function so that you can compare the difference.

h(j)=h(k), so the next hash function,
h1 is used. A second collision occurs,
so h2 is used.



Clustering
Linear probing is subject to a clustering phenomenon. Re-hashes from one location occupy a block of slots in the table which "grows" towards slots to which other keys hash. This exacerbates the collision problem and the number of re-hashed can become large.
Quadratic Probing
Better behaviour is usually obtained with quadratic probing, where the secondary hash function depends on the re-hash index:
address = h(key) + c i2
on the tth re-hash. (A more complex function of i may also be used.) Since keys which are mapped to the same value by the primary hash function follow the same sequence of addresses, quadratic probing shows secondary clustering. However, secondary clustering is not nearly as severe as the clustering shown by linear probes.

Re-hashing schemes use the originally allocated table space and thus avoid linked list overhead, but require advance knowledge of the number of items to be stored.

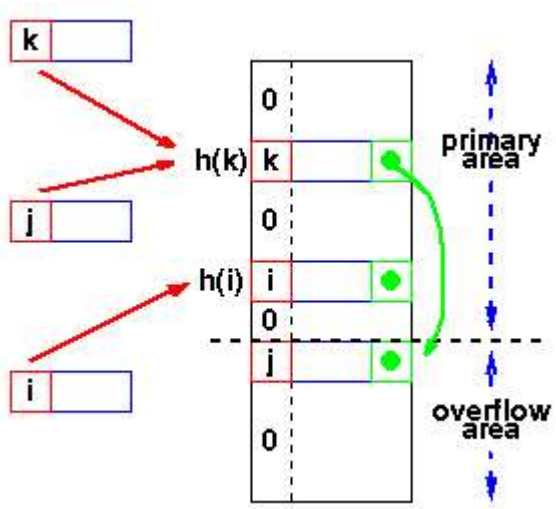
However, the collision elements are stored in slots to which other key values map directly, thus the potential for multiple collisions increases as the table becomes full.
Overflow area
Another scheme will divide the pre-allocated table into two sections: the primary area to which keys are mapped and an area for collisions, normally termed the overflow area.

When a collision occurs, a slot in the overflow area is used for the new element and a link from the primary slot established as in a chained system. This is essentially the same as chaining, except that the overflow area is pre-allocated and thus possibly faster to access. As with re-hashing, the maximum number of elements must be known in advance, but in this case, two parameters must be estimated: the optimum size of the primary and overflow areas.
Of course, it is possible to design systems with multiple overflow tables, or with a mechanism for handling overflow out of the overflow area, which provide flexibility without losing the advantages of the overflow scheme.
Summary: Hash Table Organization
Organization Advantages Disadvantages
Chaining
Unlimited number of elements

Unlimited number of collisions
Overhead of multiple linked lists
Re-hashing
Fast re-hashing
Fast access through use
of main table space
Maximum number of elements must be known

Multiple collisions may become
probable

Overflow area



Fast access
Collisions don't use primary table space
Two parameters which govern performance
need to be estimated
Animation
Hash Table Animation

Key Terms
hash table
Tables which can be searched for an item in O(1) time using a hash function to form an address from the key.
hash function
Function which, when applied to the key, produces a integer which can be used as an address in a hash table.
collision
When a hash function maps two different keys to the same table address, a collision is said to occur.
linear probing
A simple re-hashing scheme in which the next slot in the table is checked on a collision.
quadratic probing
A re-hashing scheme in which a higher (usually 2nd) order function of the hash index is used to calculate the address.
clustering.
Tendency for clusters of adjacent slots to be filled when linear probing is used.
secondary clustering.
Collision sequences generated by addresses calculated with quadratic probing.
perfect hash function
Function which, when applied to all the members of the set of items to be stored in a hash table, produces a unique set of integers within some suitable range.

9 Dynamic Algorithms

Sometimes, the divide and conquer approach seems appropriate but fails to produce an efficient algorithm.

We all know the algorithm for calculating Fibonacci numbers:

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

This algorithm is commonly used as an example of the elegance of recursion as a programming technique. However, when we examine its time complexity, we find it's far from elegant!

Analysis

If tn is the time required to calculate fn, where fn is the nth Fibonacci number. Then, by examining the function above, it's clear that

tn = tn-1 + tn-2

and

t1 = t2 = c,

where c is a constant.

Therefore

tn = cfn

Now,

thus

tn = O(fn) = O(1.618..n)

So this simple function will take exponential time! As we will see in more detail later, algorithms which run in exponential time are to be avoided at all costs!

An Iterative Solution

However, this simple alternative:

```
int fib( int n ) {
  int k, f1, f2;
  if ( n < 2 ) return n;
  else {
    f1 = f2 = 1;
    for(k=2;k<n;k++) {
      f = f1 + f2;
      f2 = f1;
      f1 = f;
      }
  return f;
}
```

runs in O(n) time.

This algorithm solves the problem of calculating f0 and f1 first, calculates f2 from these, then f3 from f2 and f1, and so on. Thus, instead of dividing the large problem into two (or more) smaller problems and solving those problems (as we did in the divide and conquer approach), we start with the simplest possible problems. We solve them (usually trivially) and save these results. These results are then used to solve slightly larger problems which are, in turn, saved and used to solve larger problems again.

Free Lunch?
As we know, there's never one! Dynamic problems obtain their efficiency by solving and storing the answers to small problems. Thus they usually trade space for increased speed. In the Fibonacci case, the extra space is insignificant - the two variables f1 and f2, but in some more complex dynamic algorithms, we'll see that the space used is significant.

Key terms
Dynamic Algorithm
A general class of algorithms which solve problems by solving smaller versions of the problem, saving the solutions to the small problems and then combining them to solve the larger problem.

9.2 Binomial Coefficients

As with the Fibonacci numbers, the binomial coefficients can be calculated recursively - making use of the relation:
nCm = n-1Cm-1 + n-1Cm
A similar analysis to that used for the Fibonacci numbers shows that the time complexity using this approach is also the binomial coefficient itself.

However, we all know that if we construct Pascal's triangle, the nth row gives all the values,
nCm, m = 0,n:

```
            1
          1   1
        1   2   1
      1   3   3   1
    1   4   6   4   1
  1   5  10  10   5   1
1   6  15  20  15   6   1
1   7  21  35  35  21   7   1
```

Each entry takes O(1) time to calculate and there are O(n2) of them. So this calculation of the coefficients takes O(n2) time. But it uses O(n2) space to store the coefficients.

9.3 Optimal Binary Search Trees

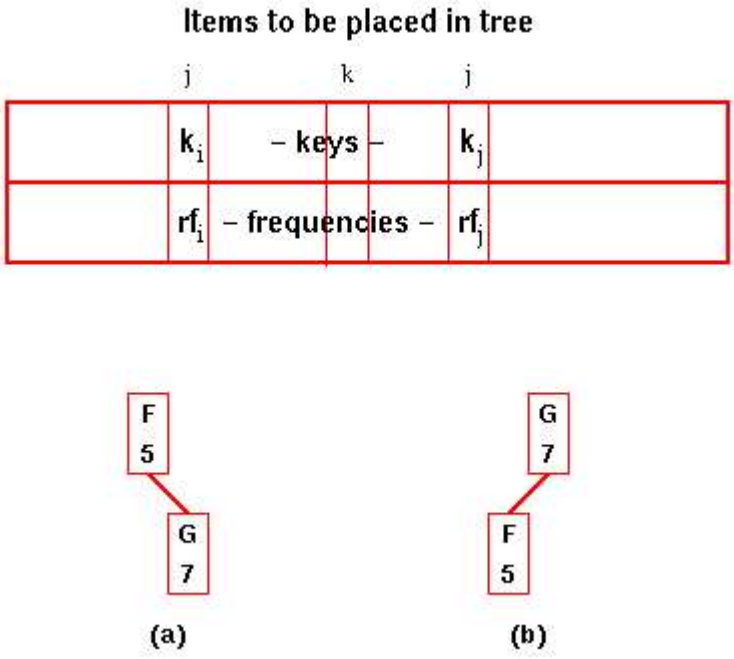
Up to this point, we have assumed that an optimal search tree is one in which the probability of occurrence of all keys is equal (or is unknown, in which case we assume it to be equal). Thus we concentrated on balancing the tree so as to make the cost of finding any key at most log n.

However, consider a dictionary of words used by a spelling checker for English language documents. It will be searched many more times for 'a', 'the', 'and', etc than for the thousands of uncommon words which are in the dictionary just in case someone happens to use one of them. Such a dictionary needs to be large: the average educated person has a vocabulary of 30 000 words, so it needs ~100 000 words in it to be effective. It is also reasonably easy to produce a table of the frequency of

occurrence of words: words are simply counted in any suitable collection of documents considered to be representative of those for which the spelling checker will be used. A balanced binary tree is likely to end up with a word such as 'miasma' at its root, guaranteeing that in 99.99+% of searches, at least one comparison is wasted!

If key, k, has relative frequency, rk, then in an optimal tree,
sum(dkrk)
where dk is the distance of the key, k, from the root (ie the number of comparisons which must be made before k is found), is minimised.

We make use of the property:
Lemma
Sub-trees of optimal trees are themselves optimal trees.
Proof
If a sub-tree of a search tree is not an optimal tree, then a better search tree will be produced if the sub-tree is replaced by an optimal tree.
Thus the problem is to determine which key should be placed at the root of the tree. Then the process can be repeated for the left- and right-sub-trees. However, a divide-and-conquer approach would choose each key as a candidate root and repeat the process for each sub-tree. Since there are n choices for the root and 2O(n) choices for roots of the two sub-trees, this leads to an O(nn) algorithm.



An efficient algorithm can be generated by the dynamic approach. We calculate the O (n) best trees consisting of just two elements (the neighbours in the sorted list of keys).

In the figure, there are two possible arrangements for the tree containing F and G.

The cost for (a) is
5.1 + 7.2 = 19

and for (b)
7.1 + 5.2 = 17

Thus (b) is the optimum tree and its cost is saved as c(f,g). We also store g as the root of the best f-g sub-tree in best(f,g).

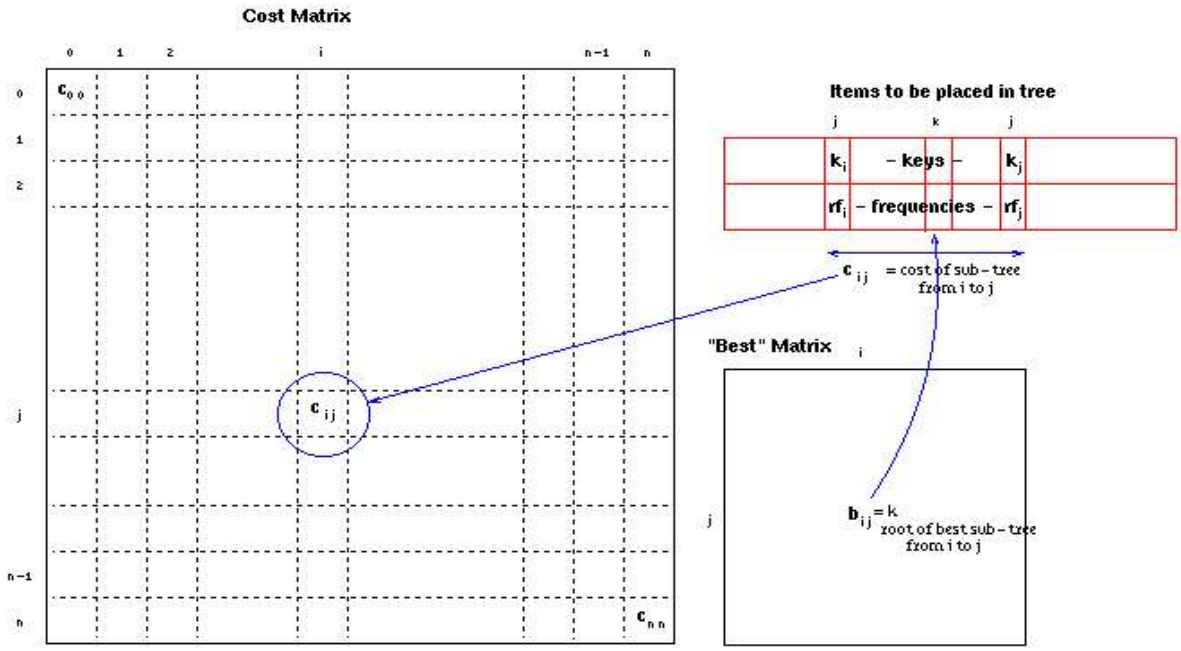
Similarly, we calculate the best cost for all n-1 sub-trees with two elements, c(g,h), c (h,i), etc.
The sub-trees containing two elements are then used to calculate the best costs for sub-trees of 3 elements. This process is continued until we have calculated the cost and the root for the optimal search tree with n elements.

There are O(n2) such sub-tree costs. Each one requires n operations to determine, if the cost of the smaller sub-trees is known.

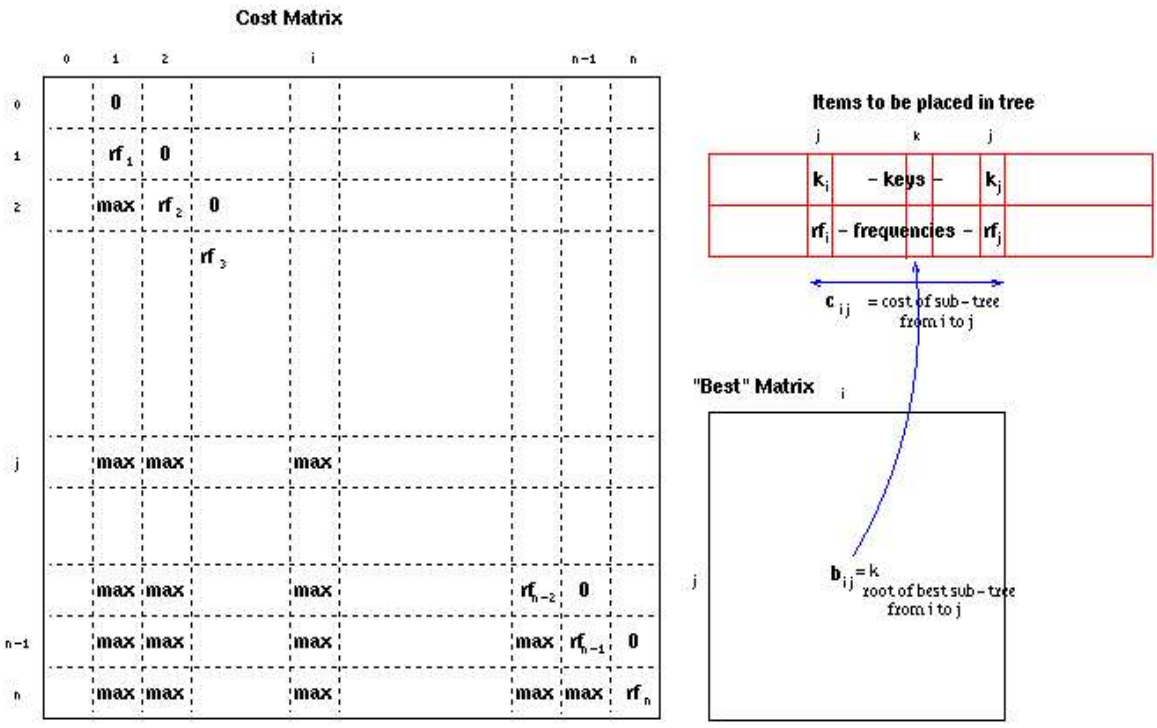
Thus the overall algorithm is O(n3).

Code for optimal binary search tree
Note some C 'tricks' to handle dynamically-allocated two-dimensional arrays using pre-processor macros for C and BEST!
This Java code may be easier to comprehend for some! It uses this class for integer matrices.

The data structures used may be represented:



After the initialisation steps, the data structures used contain the frequencies, rfi, in cii (the costs of single element trees), max everywhere below the diagonal and zeroes in the positions just above the diagonal (to allow for the trees which don't have a left or right branch):



In the first iteration, all the positions below the diagonal (ci,i+1) will be filled in with the optimal costs of two-element trees from i to i+1.

In subsequent iterations, the optimal costs of k-1 element trees (ci,i+k) are filled in using previously calculated costs of smaller trees.

Matrix Chain Multiplication

Problem
We are given a sequence of matrices to multiply:
A1 A2 A3 ... An
Matrix multiplication is associative, so
A1 (A2 A3) = (A1 A2) A3
that is, we can can generate the product in two ways.

The cost of multiplying an nxm by an mxp one is O(nmp) (or O(n3) for two nxn ones). A poor choice of parenthesisation can be expensive: eg if we have

Matrix	Rows	Columns
A1	10	100
A2	100	5
A3	5	50

the cost for (A1 A2) A3 is
A1A2 10x100x5 = 5000 => A1 A2 (10x5)
(A1A2) A3 10x5x50 = 2500 => A1A2A3 (10x50)
Total 7500
but for A1 (A2 A3)
A2A3 100x5x50 = 25000 => A2A3 (100x5)
A1(A2A3) 10x100x50 = 50000 => A1A2A3 (10x50)

Total 75000
Clearly demonstrating the benefit of calculating the optimum order before commencing the product calculation!
Optimal Sub-structure
As with the optimal binary search tree, we can observe that if we divide a chain of matrices to be multiplied into two optimal sub-chains:
(A1 A2 A3 ... Aj) (Aj+1 ... An)
then the optimal parenthesisations of the sub-chains must be composed of optimal chains. If they were not, then we could replace them with cheaper parenthesisations.

This property, known as optimal sub-structure is a hallmark of dynamic algorithms: it enables us to solve the small problems (the sub-structure) and use those solutions to generate solutions to larger problems.

For matrix chain multiplication, the procedure is now almost identical to that used for constructing an optimal binary search tree. We gradually fill in two matrices,
one containing the costs of multiplying all the sub-chains. The diagonal below the main diagonal contains the costs of all pair-wise multiplications: cost[1,2] contains the cost of generating product A1A2, etc. The diagonal below that contains the costs of triple products: eg cost[1,3] contains the cost of generating product A1A2A3, which we derived from comparing cost[1,2] and cost[2,3], etc.
one containing the index of last array in the left parenthesisation (similar to the root of the optimal sub-tree in the optimal binary search tree, but there's no root here - the chain is divided into left and right sub-products), so that best[1,3] might contain 2 to indicate that the left sub-chain contains A1A2 and the right one is A3 in the optimal parenthesisation of A1A2A3.
As before, if we have n matrices to multiply, it will take O(n) time to generate each of the O(n2) costs and entries in the best matrix for an overall complexity of O(n3) time at a cost of O(n2) space.

Key terms
optimal sub-structure
a property of optimisation problems in which the sub-problems which constitute the solution to the problem itself are themselves optimal solutions to those sub-problems. This property permits the construction of dynamic algorithms to solve the problem.

Longest Common Subsequence

Another problem that has a dynamic solution is that of finding the longest common subsequence.
Problem
Given two sequences of symbols, X and Y, determine the longest subsequence of symbols that appears in both X and Y.
Reference
Cormen, Section 16.3
Lecture notes by Kirk Pruhs, University of Pittsburgh
Pseudo-code from John Stasko's notes for CS3158 at Georgia Tech

Key terms
optimal sub-structure

a property of optimisation problems in which the sub-problems which constitute the solution to the problem itself are themselves optimal solutions to those sub-problems. This property permits the construction of dynamic algorithms to solve the problem.

Optimal Triangulation

Triangulation - dividing a surface up into a set of triangles - is the first step in the solution of a number of engineering problems: thus finding optimal triangulations is an important problem in itself.
Problem
Any polygon can be divided into triangles. The problem is to find the optimum triangulationi of a convex polygon based on some criterion, eg a triangulation which minimises the perimeters of the component triangles.
Reference
Cormen, Section 16.4

Key terms
convex polygon
a convex polygon is one in which any chord joining two vertices of the polygon lies either wholly within or on the boundary of the polygon.

10 Graphs

10.1 Minimum Spanning Trees
Greedy Algorithms
Many algorithms can be formulated as a finite series of guesses, eg in the Travelling Salesman Problem, we try (guess) each possible tour in turn and determine its cost. When we have tried them all, we know which one is the optimum (least cost) one. However, we must try them all before we can be certain that we know which is the optimum one, leading to an O(n!) algorithm.

Intuitive strategies, such as building up the salesman's tour by adding the city which is closest to the current city, can readily be shown to produce sub-optimal tours. As another example, an experienced chess player will not take an opponent's pawn with his queen - because that move produced the maximal gain, the capture of a piece - if his opponent is guarding that pawn with another pawn. In such games, you must look at all the moves ahead to ensure that the one you choose is in fact the optimal one. All chess players know that short-sighted strategies are good recipes for disaster!

There is a class of algorithms, the greedy algorithms, in which we can find a solution by using only knowledge available at the time the next choice (or guess) must be made. The problem of finding the Minimum Spanning Tree is a good example of this class.

The Minimum Spanning Tree Problem
Suppose we have a group of islands that we wish to link with bridges so that it is possible to travel from one island to any other in the group. Further suppose that (as usual) our government wishes to spend the absolute minimum amount on this project (because other factors like the cost of using, maintaining, etc, these bridges will probably be the responsibility of some future government). The engineers are able to

produce a cost for a bridge linking each possible pair of islands. The set of bridges which will enable one to travel from any island to any other at minimum capital cost to the government is the minimum spanning tree.

We will need some definitions first:
Graphs

A graph is a set of vertices and edges which connect them. We write:

G = (V,E)

where V is the set of vertices and the set of edges,

E = { (vi,vj) }
where vi and vj are in V.
Paths
A path, p, of length, k, through a graph is a sequence of connected vertices:

p = <v0,v1,...,vk>
where, for all i in (0,k-1:

(vi,vi+1) is in E.

Cycles

A graph contains no cycles if there is no path of non-zero length through the graph, p = <v0,v1,...,vk> such that v0 = vk.
Spanning Trees

A spanning tree of a graph, G, is a set of |V|-1 edges that connect all vertices of the graph.
Minimum Spanning Tree
In general, it is possible to construct multiple spanning trees for a graph, G. If a cost, cij, is associated with each edge, eij = (vi,vj), then the minimum spanning tree is the set of edges, Espan, forming a spanning tree, such that:

C = sum(cij | all eij in Espan)

is a minimum.
Kruskal's Algorithm
This algorithm creates a forest of trees. Initially the forest consists of n single node trees (and no edges). At each step, we add one (the cheapest one) edge so that it joins two trees together. If it were to form a cycle, it would simply link two nodes that were already part of a single connected tree, so that this edge would not be needed.

The basic algorithm looks like this:

```
Forest MinimumSpanningTree( Graph g, int n, double **costs ) {
  Forest T;
  Queue q;
```

```
  Edge e;
  T = ConsForest( g );
  q = ConsEdgeQueue( g, costs );
  for(i=0;i<(n-1);i++) {
     do {
       e = ExtractCheapestEdge( q );
     } while ( !Cycle( e, T ) );
     AddEdge( T, e );
  }
  return T;
}
```

The steps are:
The forest is constructed - with each node in a separate tree.
The edges are placed in a priority queue.
Until we've added n-1 edges,
Extract the cheapest edge from the queue,
If it forms a cycle, reject it,
Else add it to the forest. Adding it to the forest will join two trees together.
Every step will have joined two trees in the forest together, so that at the end, there will only be one tree in T.

We can use a heap for the priority queue. The trick here is to detect cycles. For this, we need a union-find structure.
Union-find structure
To understand the union-find structure, we need to look at a partition of a set.
Partitions
A partitions is a set of sets of elements of a set.
Every element of the set belong to one of the sets in the partition.
No element of the set belong to more than one of the sub-sets.
or
Every element of a set belongs to one and only one of the sets of a partition.

The forest of trees is a partition of the original set of nodes. Initially all the sub-sets have exactly one node in them. As the algorithm progresses, we form a union of two of the trees (sub-sets), until eventually the partition has only one sub-set containing all the nodes.

A partition of a set may be thought of as a set of equivalence classes. Each sub-set of the partition contains a set of equivalent elements (the nodes connected into one of the trees of the forest). This notion is the key to the cycle detection algorithm. For each sub-set, we denote one element as the representative of that sub-set or equivalence class. Each element in the sub-set is, somehow, equivalent and represented by the nominated representative.

As we add elements to a tree, we arrange that all the elements point to their representative. As we form a union of two sets, we simply arrange that the representative of one of the sets now points to any one of the elements of the other set.

So the test for a cycle reduces to: for the two nodes at the ends of the candidate edge, find their representatives. If the two representatives are the same, the two nodes are

already in a connected tree and adding this edge would form a cycle. The search for the representative simply follows a chain of links.

Each node will need a representative pointer. Initially, each node is its own representative, so the pointer is set to NULL. As the initial pairs of nodes are joined to form a tree, the representative pointer of one of the nodes is made to point to the other, which becomes the representative of the tree. As trees are joined, the representative pointer of the representative of one of them is set to point to any element of the other. (Obviously, representative searches will be somewhat faster if one of the representatives is made to point directly to the other.)

Equivalence classes also play an important role in the
verification of software.

Select diagrams of Kruskal's algorithm in operation.
Greedy operation
At no stage did we try to look ahead more than one edge - we simply chose the best one at any stage. Naturally, in some situations, this myopic view would lead to disaster! The simplistic approach often makes it difficult to prove that a greedy algorithm leads to the optimal solution. proof by contradiction is a common proof technique used: we demonstrate that if we didn't make the greedy choice now, a non-optimal solution would result. Proving the MST algorithm
is, happily, one of the simpler proofs by contradiction!
Data structures for graphs
You should note that we have discussed graphs in an abstract way: specifying that they contain nodes and edges and using operations like AddEdge, Cycle, etc. This enables us to define an abstract data type without considering implementation details, such as how we will store the attributes of a graph! This means that a complete solution to, for example, the MST problem can be specified before we've even decided how to store the graph in the computer. However, representation issues can't be deferred forever, so we need to examine ways of representing graphs in a machine. As before, the performance of the algorithm will be determined by the data structure chosen.

Key terms
Greedy algorithms
Algorithms which solve a problem by making the next step based on local knowledge alone - without looking ahead to determine whether the next step is the optimal one.
Equivalence Classes
The set of equivalence classes of a set is a partition of a set such that all the elements in each subset (or equivalence class) is related to every other element in the subset by an equivalence relation.
Union Find Structure
A structure which enables us to determine whether two sets are in fact the same set or not.
Kruskal's Algorithm
One of the two algorithms commonly used for finding a minimum spanning tree - the other is Prim's algorithm.

10.2 Dijkstra's Algorithm

Djikstra's algorithm (named after its discover, E.W. Dijkstra) solves the problem of finding the shortest path from a point in a graph (the source) to a destination. It turns out that one can find the shortest paths from a given source to all points in a graph in the same time, hence this problem is sometimes called the single-source shortest paths problem.
The somewhat unexpected result that all the paths can be found as easily as one further demonstrates the value of reading the literature on algorithms!

This problem is related to the spanning tree one. The graph representing all the paths from one vertex to all the others must be a spanning tree - it must include all vertices. There will also be no cycles as a cycle would define more than one path from the selected vertex to at least one other vertex. For a graph,

G = (V,E) where
V is a set of vertices and
E is a set of edges.
Dijkstra's algorithm keeps two sets of vertices:
S the set of vertices whose shortest paths from the source have already been determined and
V-S the remaining vertices.
The other data structures needed are:
d array of best estimates of shortest path to each vertex
pi an array of predecessors for each vertex

The basic mode of operation is:
Initialise d and pi,
Set S to empty,
While there are still vertices in V-S,
Sort the vertices in V-S according to the current best estimate of their distance from the source,
Add u, the closest vertex in V-S, to S,
Relax all the vertices still in V-S connected to u

Relaxation
The relaxation process updates the costs of all the vertices, v, connected to a vertex, u, if we could improve the best estimate of the shortest path to v by including (u,v) in the path to v.

The relaxation procedure proceeds as follows:

```
initialise_single_source( Graph g, Node s )
  for each vertex v in Vertices( g )
     g.d[v] := infinity
     g.pi[v] := nil
  g.d[s] := 0;
```

This sets up the graph so that each node has no predecessor (pi[v] = nil) and the estimates of the cost (distance) of each node from the source (d[v]) are infinite, except for the source node itself (d[s] = 0).

Note that we have also introduced a further way to store a graph (or part of a graph - as this structure can only store a spanning tree), the predecessor sub-graph - the list of predecessors of each node,

pi[j], 1 <= j <= |V|

The edges in the predecessor sub-graph are (pi[v],v).

The relaxation procedure checks whether the current best estimate of the shortest distance to v (d[v]) can be improved by going through u (i.e. by making u the predecessor of v):

```
relax( Node u, Node v, double w[][] )
   if d[v] > d[u] + w[u,v] then
      d[v] := d[u] + w[u,v]
      pi[v] := u
```

The algorithm itself is now:

```
shortest_paths( Graph g, Node s )
   initialise_single_source( g, s )
   S := { 0 }        /* Make S empty */
   Q := Vertices( g ) /* Put the vertices in a PQ */
   while not Empty(Q)
      u := ExtractCheapest( Q );
      AddNode( S, u ); /* Add u to S */
      for each vertex v in Adjacent( u )
         relax( u, v, w )
```

Operation of Dijkstra's algorithm

As usual, proof of a greedy algorithm is the trickiest part.

Animation

In this animation, a number of cases have been selected to show all aspects of the operation of Dijkstra's algorithm. Start by selecting the data set (or you can just work through the first one - which appears by default). Then select either step or run to execute the algorithm. Note that it starts by assigning a weight of infinity to all nodes, and then selecting a source and assigning a weight of zero to it. As nodes are added to the set for which shortest paths are known, their colour is changed to red. When a node is selected, the weights of its neighbours are relaxed .. nodes turn green and flash as they are being relaxed. Once all nodes are relaxed, their predecessors are updated, arcs are turned green when this happens. The cycle of selection, weight relaxation and predecessor update repeats itself until all the shortest path to all nodes has been found.

Key terms

single-source shortest paths problem

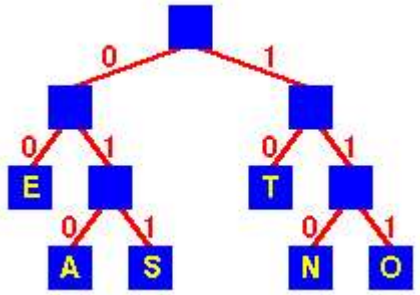
A descriptive name for the problem of finding the shortest paths to all the nodes in a graph from a single designated source. This problem is commonly known by the algorithm used to solve it - Dijkstra's algorithm.

predecessor list

A structure for storing a path through a graph.

11 Huffman Encoding

This problem is that of finding the minimum length bit string which can be used to encode a string of symbols. One application is text compression:
What's the smallest number of bits (hence the minimum size of file) we can use to store an arbitrary piece of text?
Huffman's scheme uses a table of frequency of occurrence for each symbol (or character) in the input. This table may be derived from the input itself or from data which is representative of the input. For instance, the frequency of occurrence of letters in normal English might be derived from processing a large number of text documents and then used for encoding all text documents. We then need to assign a variable-length bit string to each character that unambiguously represents that character. This means that the encoding for each character must have a unique prefix. If the characters to be encoded are arranged in a binary tree:



Encoding tree for ETASNO
An encoding for each character is found by following the tree from the route to the character in the leaf: the encoding is the string of symbols on each branch followed.

For example:

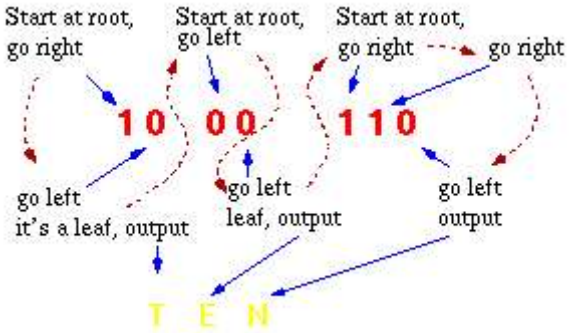
String	Encoding
TEA	10 00 010
SEA	011 00 010
TEN	10 00 110

Notes:
As desired, the highest frequency letters - E and T - have two digit encodings, whereas all the others have three digit encodings.
Encoding would be done with a lookup table.
A divide-and-conquer approach might have us asking which characters should appear in the left and right subtrees and trying to build the tree from the top down. As with the optimal binary search tree, this will lead to to an exponential time algorithm.

A greedy approach places our n characters in n sub-trees and starts by combining the two least weight nodes into a tree which is assigned the sum of the two leaf node weights as the weight for its root node.

Operation of the Huffman algorithm.

The time complexity of the Huffman algorithm is O(nlogn). Using a heap to store the weight of each tree, each iteration requires O(logn) time to determine the cheapest weight and insert the new weight. There are O(n) iterations, one for each item.
Decoding Huffman-encoded Data
Curious readers are, of course, now asking
"How do we decode a Huffman-encoded bit string? With these variable length strings, it's not possible to break up an encoded string of bits into characters!"



The decoding procedure is deceptively simple. Starting with the first bit in the stream, one then uses successive bits from the stream to determine whether to go left or right in the decoding tree. When we reach a leaf of the tree, we've decoded a character, so we place that character onto the (uncompressed) output stream. The next bit in the input stream is the first bit of the next character.
Transmission and storage of Huffman-encoded Data
If your system is continually dealing with data in which the symbols have similar frequencies of occurence, then both encoders and decoders can use a standard encoding table/decoding tree. However, even text data from various sources will have quite different characteristics. For example, ordinary English text will have generally have 'e' at the root of the tree, with short encodings for 'a' and 't', whereas C programs would generally have ';' at the root, with short encodings for other punctuation marks such as '(' and ')' (depending on the number and length of comments!). If the data has variable frequencies, then, for optimal encoding, we have to generate an encoding tree for each data set and store or transmit the encoding with the data. The extra cost of transmitting the encoding tree means that we will not gain an overall benefit unless the data stream to be encoded is quite long - so that the savings through compression more than compensate for the cost of the transmitting the encoding tree also.

Sample Code
A full implementation of the Huffman algorithm is available from Verilib. Currently, there is a Java version there. C and C++ versions will soon be available also.
Other problems
Optimal Merge Pattern
We have a set of files of various sizes to be merged. In what order and combinations should we merge them? The solution to this problem is basically the same as the Huffman algorithm - a merge tree is constructed with the largest file at its root.

12 Fast Fourier Transforms

Fourier transforms have wide application in scientific and engineering problems, for example, they are extensively used in signal processing to transform a signal from the time domain to the frequency domain.

Here, we will use them to generate an efficient solution to an apparently unrelated problem - that of multiplying two polynomials. Apart from demonstrating how the Fast Fourier Transform (FFT) algorithm calculates a Discrete Fourier Transform and deriving its time complexity, this approach is designed to reinforce the following points:
'Better' solutions are known to many problems for which, intuitively, it would not appear possible to find a better solution.
As a consequence, unless you have read extensively in any problem area already, you should consult the literature before attempting to solve any numerical or data processing problem presented to you.

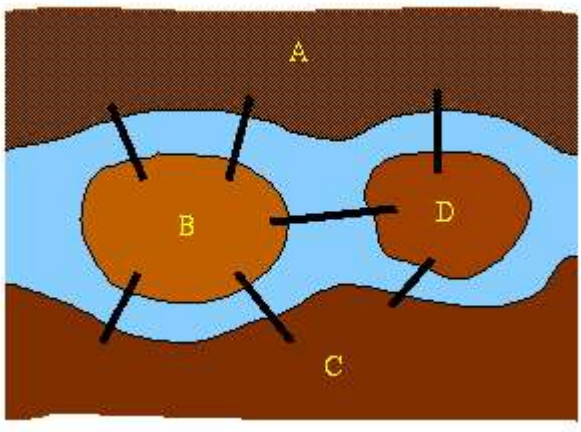
Because of the limitations of HTML in handling mathematical equations, the notes for this section were prepared with LaTeX and are available as a PostScript file.

13 Hard or Intractable Problems

If a problem has an O(nk) time algorithm (where k is a constant), then we class it as having polynomial time complexity and as being efficiently solvable.

If there is no known polynomial time algorithm, then the problem is classed as intractable.

The dividing line is not always obvious. Consider two apparently similar problems:
Euler's problem
(often characterized as the Bridges of Königsberg - a popular 18th C puzzle) asks whether there is a path through a graph which traverses each edge only once.
Hamilton's problem asks whether there is a path through a graph which visits each vertex exactly once.
Euler's problem

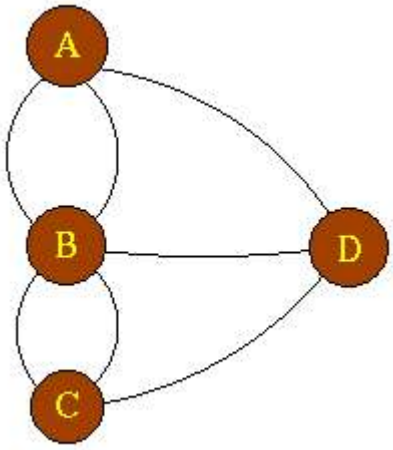


The 18th century German city of Königsberg was situated on the river Pregel. Within a park built on the banks of the river, there were two islands joined by seven bridges.

The puzzle asks whether it is possible to take a tour through the park, crossing each bridge only once.

An exhaustive search requires starting at every possible point and traversing all the possible paths from that point - an O(n!) problem. However Euler showed that an Eulerian path existed iff
it is possible to go from any vertex to any other by following the edges (the graph must be connected) and
every vertex must have an even number of edges connected to it, with at most two exceptions (which constitute the starting and ending points).
It is easy to see that these are necessary conditions: to complete the tour, one needs to enter and leave every point except the start and end points. The proof that these are sufficient conditions may be found in the literature . Thus we now have a O(n) problem to determine whether a path exists.

Transform the map into a graph in which
the nodes represent the "dry land" points
and the arcs represent the bridges.



We can now easily see that the Bridges of Königsberg does not have a solution.

A quick inspection shows that it does have a Hamiltonian path.

However there is no known efficient algorithm for determining whether a Hamiltonian path exists.
But if a path was found, then it can be verified to be a solution in polynomial time: we simply verify that each edge in the path is actually an edge (O(e) if the edges are stored in an adjacency matrix) and that each vertex is visited only once (O(n2) in the worst case).

Classes P and NP
Euler's problem lies in the class P: problems solvable in Polynomial time. Hamilton's problem is believed to lie in class NP (Non-deterministic Polynomial).

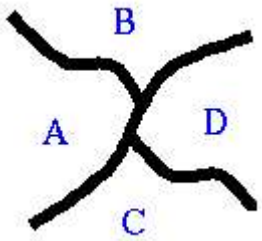
Note that I wrote "believed" in the previous sentence. No-one has succeeded in proving that efficient (ie polynomial time) algorithms don't exist yet!
What does NP mean?
At each step in the algorithm, you guess which possibility to try next. This is the non-deterministic part: it doesn't matter which possibility you try next. There is no information used from previous attempts (other than not trying something that you've already tried) to determine which alternative should be tried next. However, having made a guess, you can determine in polynomial time whether it is a solution or not.

Since nothing from previous trials helps you to determine which alternative should be tried next, you are forced to investigate all possibilities to find a solution. So the only systematic thing you can do is use some strategy for systematically working through all possibilities, eg setting out all permutations of the cities for the travelling salesman's tour.

Many other problems lie in class NP. Some examples follow.
Composite Numbers
Determining whether a number can be written as the product of two other numbers is the composite numbers problem. If a solution is found, it is simple to verify it, but no efficient method of finding the solution exists.
Assignment
Assignment of compatible room-mates: assume we have a number of students to be assigned to rooms in a college. They can be represented as the vertices on a graph with edges linking compatible pairs. If we have two per room, a class P algorithm exists, but if three are to be fitted in a room, we have a class NP problem.
Boolean satisfiability
Given an arbitrary boolean expression in n variables:
a1 op a2 op ... op an
where op are boolean operators, and, or, ..

Can we find an assignment of (true,false) to the ai so that the expression is true? This problem is equivalent to the circuit-satisfiability problem which asks can we find a set of inputs which will produce a true at the output of a circuit composed of arbitrary logic gates.

A solution can only be found by trying all 2n possible assignments.
Map colouring
The three-colour map colouring problem asks if we can colour a map so that no adjoining countries have the same colour. Once a solution has been guessed, then it is readily proved.



[This problem is easily answered if there are only 2 colours - there must be no point at which an odd number of countries meet - or 4 colours - there is a proof that 4 colours suffice for any map.]

This problem has a graph equivalent: each vertex represents a country and an edge is drawn between two vertices if they share a common border.

Its solution has a more general application. If we are scheduling work in a factory: each vertex can represent a task to be performed - they are linked by an edge if they share a common resource, eg require a particular machine. A colouring of the vertices with 3 colours then provides a 3-shift schedule for the factory.

Many problems are reducible to others: map colouring can be reduced to graph colouring. A solution to a graph colouring problem is effectively a solution to the equivalent map colouring or scheduling problem. The map or graph-colouring problem may be reduced to the boolean satisfiability problem. To give an informal description of this process, assume the three colours are red, blue and green. Denote the partial solution, "A is red" by ar so that we have a set of boolean variables:

ar	A is red
ab	A is blue
ag	A is green
br	B is red
bb	B is blue
bg	B is green
cr	C is red
...	...

Now a solution to the problem may be found by finding values for ar, ab, etc which make the expression true:
((ar and not ab and not ag) and ((bb and (cb and (dg ....

Thus solving the map colouring problem is equivalent to finding an assignment to the variables which results in a true value for the expression - the boolean satisfiability problem.

There is a special class of problems in NP: the NP-complete problems. All the problems in NP are efficiently reducible to them. By efficiently, we mean in polynomial time, so the term polynomially reducible provides a more precise definition.

In 1971, Cook was able to prove that the boolean satisfiability problem was NP-complete. Proofs now exist showing that many problems in NP are efficiently reducible to the satisfiability problem. Thus we have a large class of problems which will are all related to each other: finding an efficient solution to one will result in an efficient solution for them all.

An efficient solution has so far eluded a very large number of researchers but there is also no proof that these problems cannot be solved in polynomial time, so the search continues.

Class NP problems are solvable by non-deterministic algorithms: these algorithms consist of deterministic steps alternating with non-deterministic steps in which a random choice (a guess) must be made. A deterministic algorithm must, given a possible solution,
have at least one set of guessing steps which lead to the acceptance of that solution, and
always reject an invalid solution.

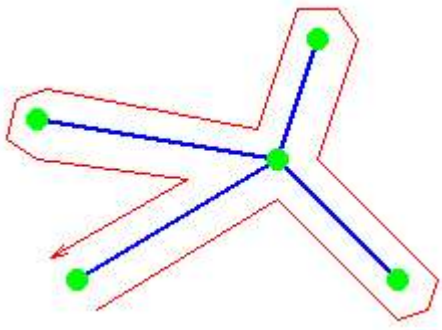
We can also view this from the other aspect: that of trying to determine a solution. At each guessing stage, the algorithm randomly selects another element to add to the solution set: this is basically building up a "game" tree. Various techniques exist for pruning the tree - backtracking when an invalid solution is found and trying another branch, but this is where the exponential time complexity starts to enter!
Travelling salesman
It's possible to cast this problem - which is basically an optimality one, we're looking for the best tour - into a yes-no one also by simply asking:
Can we find a tour with a cost less than x?

By asking this question until we find a tour with a cost x for which the answer is provably no, we have found the optimal tour. This problem can also be proved to be in NP. (It is reducible to the Hamiltonian circuit problem.)

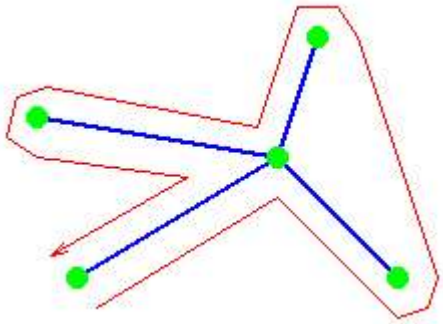
Various heuristics have been developed to find near optimal solutions with efficient algorithms.

One simple approach is the find the minimum spanning tree. One possible tour simple traverses the MST twice. So we can find a tour which is at most twice as long as the optimum tour in polynomial time. Various heuristics can now be applied to reduce this tour, eg by taking shortcuts.

An algorithm due to Christofides can be shown to produce a tour which is no more than 50% longer than the optimal tour.



It starts with the MST and singles out all cities which are linked to an odd number of cities.
These are linked in pairs by a variant of the procedure used to find compatible room-mates.



This can then be improved by taking shortcuts.

Another strategy which works well in practice is to divide the "map" into many small regions and to generate the optimum tour by exhaustive search within those small regions. A greedy algorithm can then be used to link the regions. While this algorithm will produce tours as little as 5% longer than the optimum tour in acceptable times, it is still not guaranteed to produce the optimal solution.

Key terms
Polynomial Time Complexity
Problems which have solutions with time complexity O(nk) where k is a constant are said to have polynomial time complexity.
Class P
Set of problems which have solutions with polynomial time complexity.
Non-deterministic Polynomial (NP)
A problem which can be solved by a series of guessing (non-deterministic) steps but whose solution can be verified as correct in polynomial time is said to lie in class NP.
Eulerian Path
Path which traverses each arc of a graph exactly once.
Hamiltonian Path
Path which passes through each node of a graph exactly once.
NP-Complete Problems
Set of problems which are all related to each other in the sense that if any one of them can be shown to be in class P, all the others are also in class P.

14 Games

Naive Solutions
A naive program attempting to play a game like chess will:
Determine the number of moves which can be made from the current position,
For each of these moves,
Apply the move to the current position,
Calculate a "score" for the new position,
If the maximum search "depth" has been reached, return with this score as the score for this move,
else recursively call the program with the new position.
Choose the move with the best score and return its score and the move generating it to the calling routine.
Because there are usually at least 20 possible moves from any given chess position, to search to a depth of m requires ~20m moves. Since good human players usually look 10 or more moves ahead, the simple algorithm would severely tax the capabilities of

even the fastest modern computer.

However, with a little cunning, the number of moves which needs to be searched can be dramatically reduced - enabling a computer to search deeper in a reasonable time and, as recent events have shown, enable a computer to finally be a match for even the best human players.

Alpha-Beta Algorithm

The Alpha-Beta algorithm reduces the number of moves which need to be explored by "cutting off" regions of the game tree which cannot produce a better result than has already been obtained in some part of the tree which has already been searched.

Appendices

Appendix A: Languages

A.1 ANSI C

Function prototypes

ANSI C Compilers

A.2 C++

A.3 Java

Designed by a group within Sun Microsystems, Java has eliminated some of the more dangerous features of C (to the undoubted disappointment of some hackers - who probably achieve their daily highs from discovering new ways to program dangerously in C!).

A host of texts on Java have now appeared - possibly setting a new record for the rate of textbook production on any one subject!

Appendix B: Source Code Listings

This section collects references to all the source code listings inserted in other parts of the notes in one place.

Listing Description

collection.h Generic collection specification

collection.c Array implementation of a collection

collection_ll.c Linked list implementation of a collection

coll_a.h

coll_at.c

Collection with ordering function set on construction

Implementation for a tree

binsearch.c Binary search

tree_struct.c

tree_add.c

tree_find.c

Trees

heap_delete.c

Heaps

RadixSort.h

RadixSort.c

Bins.h

Bins.c
Radix Sort
optbin.c Optimal binary search tree

Getting these notes

Notes
A gzipped tar file of all the notes and animations is available here. It's about 6 Mbytes, so please don't ask for it to be email'd to you - especially if you're using a free mail host such as hotmail: it won't fit!

If you place these notes on a public (or semi-public) server anywhere, then please leave the attributions to all the authors in the front page. I'd appreciate it if you'd also let me know where they've been placed - it's nice to know that your work is getting lots of exposure ;-).
Animations
The animations alone (including the Java source code used to generate them) are available here.
Problems?
If your decompression program (most will work: gunzip on Linux and WinZip under Windows are known to be fine) complains about a corrupted file, I suggest fetching it again, making doubly sure that the transfer is taking palce in binary mode.

Some browsers try to be too smart about file types and try to decompress everything automatically. If this is happening and causing problems for you, then try a different browser - or try to download the file onto your machine without decompression first.

If you have problems accessing the files, email me giving me as much information about the problem as possible and I will try to help, but don't expect much from a simple "I can't download your files".

John Morris

Slides

PowerPoint Slides
1998 Lectures
The files in the table below are gzipped files of PowerPoint slides. You will need a PowerPoint viewer to look at them. These are the actual slides from the 1998 lectures: expect some improvements, error corrections and changes in the order in which topics are presented. However, the 1999 lectures will mainly use the same material.

Please note that the "information density" on lecture slides is very low: printing out all the slides on single pages will consume a large number of trees for the amount of information thus gained. The lecture notes themselves have a much higher information density. However, running through the slides with a viewer may be a valuable way of refreshing your memory about major points made in lectures. If you must print them out, it is strongly suggested that you use PowerPoint's "6-up" facility!
Lists

Stacks
Searching
Complexity
Sorting
Bin Sort
Searching (2)
Searching (3)
Hash Tables
Dynamic Algorithms
Dynamic Algorithms
Minimum Spanning Trees
Equivalence Classes
Graph Representations
Dijkstra's Algorithm
Huffman Encoding
Fourier Transforms
Hard Problems
Games
Experimental Design
Functions
Key points

Course Management

Course Management

Workshops
Before starting on the assignment exercises, it's worthwhile to consider the design of experiments first.

1999 Workshops
Lab Schedule 1999

There is no assignment 2.
Assignments 3 & 4 - 1999

Submission instructions

1998 Workshops
You might find that browsing through previous years' workshops and the feedback notes helps you to determine what is expected!
Workshop/Assignment 1 - 1998
Workshop/Assignment 2 - 1998
Assignments 3 & 4 - 1998

1997 Workshops
Workshop 1 - Collections

Workshop 2 - Searching

Workshop 3 - QuickSort vs RadixSort

Workshop 4 - Red-Black Trees

1996 Workshops
Workshop 1 - Collections

Workshop 1 - Feedback

Workshop 2 - Searching

Workshop 2 - Feedback

Workshop 3 - Minimum Spanning Trees

Workshop 3 - Feedback

Workshop 4 - Minimum Spanning Trees

Workshop 4 - Feedback

Past Exams
1997

Tutorial Exercises
Arrays or Linked Lists? Overheads, Complexity
Asymptotic behaviour, ADT Design
Sheet 3
B+ trees
stable sorting,
a puzzle,
AVL trees,
dynamic memory allocation,
equivalence classes
Sheet 4
Heap Sort,
Quick Sort,
Radix Sort,
Hash Tables,
Search Trees
Sheet 5
MST,
Sheet 6
Hard problems

Texts

Data Structures and Algorithms
Texts
The following is a (non-exhaustive) list of texts which are in the UWA library which

cover aspects of this course. Not all the texts cover all the material - you will need to search a little for some of the topics.

Since there are many texts here, it's probably simpler to note a few representative catalogue numbers and simply look in the shelves in that area! For instance 005.37 obviously has a decent block of texts.

Texts highlighted in red have been used as sources for some of the material in this course.

Brown, Marc H.
Algorithm animation / Marc H. Brown.
Cambridge, Mass : M.I.T. Press, c1988.
FIZ 006.6 1988 ALG x

Harel, David, 1950-
Algorithmics : the spirit of computing / David Harel.
Wokingham, England ; Reading, Mass : Addison-Wesley, c1987.
FIZ 004 1987 ALG x

Sedgewick, Robert, 1946-
Algorithms / Robert Sedgewick.
Reading, Mass : Addison-Wesley, c1983.
SRR 517.6 1983 ALG DUE 22-11-96 x

Sedgewick, Robert, 1946-
Algorithms / Robert Sedgewick.
Reading, Mass : Addison-Wesley, c1988.
FIZ Reserve 517.6 1988 ALG x
FIZ Reserve 517.6 1988 ALG x

Kingston, Jeffrey H. (Jeffrey Howard)
Algorithms and data structures : design, correctness, analysis /
Sydney : Addison-Wesley, 1990.
FIZ 005.73 1990 ALG DUE 30-08-96 x

Wirth, Niklaus, 1934-
Algorithms + data structures=programs / Niklaus Wirth.
Englewood Cliffs, N.J : Prentice-Hall, c1976.
FIZ 005.1 1976 ALG x
FIZ 005.1 1976 ALG x

Moret, B. M. E. (Bernard M. E.)
Algorithms from P to NP / B.M.E. Moret, H.D. Shapiro.
Redwood City, CA : Benjamin/Cummings, c1991-
FIZ 005.1 1991 ALG

Sedgewick, Robert, 1946-
Algorithms in C / Robert Sedgewick.
Reading, Mass : Addison-Wesley Pub. Co., c1990.
SRR 005.133 1990 ALG

Collected algorithms from ACM.
New York, N.Y : Association for Computing Machinery, 1975-
R 005.1 FIZ Reference x
MICROFICHE MP 430 FIZ Microform x

Moffat, David V., 1944-
Common algorithms in Pascal with programs for reading / David V.
Englewood Cliffs, N.J : Prentice-Hall, c1984.
FIZ 005.133 1984 COM x

Baase, Sara.
Computer algorithms : introduction to design and analysis / Sara
Reading, Mass : Addison-Wesley Pub. Co., c1978.
FIZ 005.1 1978 COM x

Walker, Henry M., 1947-
Computer science 2 : principles of software engineering, data
Glenview, Ill : Scott, Foresman, c1989.
FIZ 005.1 1989 COM

Garey, Michael R.
Computers and intractability : a guide to the theory of NP-
San Francisco : W. H. Freeman, c1979.
FIZ 005.1 1979 COM DUE 03-09-96 x

Aho, Alfred V.
Data structures and algorithms / Alfred V. Aho, John E. Hopcroft,
Reading, Mass : Addison-Wesley, c1983.
FIZ 005.73 1983 DAT x

Aho, Alfred V.

The design and analysis of computer algorithms / Alfred V. Aho,
Reading, Mass : Addison-Wesley Pub. Co., [1974]

FIZ	005.1 1974 DES		x
FIZ	005.1 1974 DES		x

Mehlhorn, Kurt, 1949-
UNIF Effiziente Allgorithmen. English.
Data structures and algorithms / Kurt Mehlhorn.
Berlin ; New York : Springer, 1984.

FIZ	005.73 1984 DAT V. 2		x
FIZ	005.73 1984 DAT V. 3		x
FIZ	005.73 1984 DAT V. 1		x

Brassard, Gilles, 1955-
Fundamentals of algorithmics / Gilles Brassard and Paul Bratley.
Englewood, N.J. : Prentice Hall, c1996.

FIZ	517.6 1996 FUN	DUE 22-11-96	x

Horowitz, Ellis.
Fundamentals of computer algorithms / Ellis Horowitz, Sartaj
Potomac, Md : Computer Science Press, c1978.

FIZ	005.12 1978 FUN	DUE 30-08-96	x

Gonnet, G. H. (Gaston H.)
Handbook of algorithms and data structures : in Pascal and C /
Wokingham, England ; Reading, Mass : Addison-Wesley Pub. Co.,

SRR	005.133 1991 HAN		x

Cormen, Thomas H.
Introduction to algorithms / Thomas H. Cormen, Charles E.
Cambridge, Mass : MIT Press ; New York : McGraw-Hill, c1990.

FIZ Reserve	005.1 1990 INT		x
FIZ 3day	005.1 1990 INT		x

Tremblay, Jean-Paul, 1938-
An Introduction to computer science : an algorithmic approach /
New York : McGraw-Hill, c1979.

FIZ	005.1 1979 INT		x

Machtey, Michael.
An introduction to the general theory of algorithms / Michael

New York : North Holland, c1978.
FIZ 005.13 1978 INT x

Greene, Daniel H., 1955-
Mathematics for the analysis of algorithms / Daniel H. Greene,
Boston : Birkhauser, c1981.
FIZ 517.6 1981 GRE x

Reinelt, G. (Gerhard)
The traveling salesman : computational solutions for TSP
Berlin ; New York : Springer-Verlag, c1994.
FIZ P 004.05 P27 x
Classic data structures in C++ / Timothy A. Budd.
Reading, Mass. : Addison-Wesley Pub. Co., c1994.
FIZ 005.73 1994 CLA x

Standish, Thomas A., 1941-
Data structure techniques / Thomas A. Standish.
Reading, MA : Addison-Wesley, c1980.
FIZ 005.73 1980 DAT x

Data Structures & Algorithms - Online courses
This is a partial list of on-line course material and tutorials for data structures and algorithms.
Thomas Niemann's text on sorting and searching
Updated version of Thomas Niemann's text

Animated Algorithms

The following pages contain animations of some of the algorithms covered in this text. Please note that
Some of the Java classes take a very long time to load!
These animations are currently the result of a major effort to enhance the data structures and algorithms course and are thus subject to continuous enhancement. Comments are most welcome!

UWA animations
Please note that these are under active development!
Sorting algorithms
Woi Ang's Insertion Sort Animation
Woi Ang's QuickSort Animation
Chien Wei Tan's QuickSort Animation
Woi Ang's Bin Sort Animation
Woi Ang's Radix Sort Animation
Woi Ang's Priority Queue Animation
Searching Algorithms

Mervyn Ng's Red Black Tree Animation
Woi Ang's Hash Table Construction Animation
Woi Ang's Optimal Binary Search Tree Animation
Greedy algorithms
Woi Ang's Huffman Encoding & Decoding Animation
Dynamic algorithms
Woi Ang's Matrix Chain Multiplication Animation
Graph algorithms
Mervyn Ng's Minimum Spanning Tree Animation
Mervyn Ng's Animation of Dijkstra's Algorithm
If you find the animations useful, but want them a little closer to home, you can download a file of them all: anim.tar.gz. They are also available by ftp. If you do download them, please don't forget to acknowledge Woi Ang as the author wherever you use them and I'd appreciate it if you'd let me know .. and, of course, if you have any suggestions or comments, they're most welcome: morris@ee.uwa.edu.au.