Algorithm Archive

(Work in progress)

Scott Gasch, scott@wannabe.guru.org

$Id: alg.tex, v1.9 1999/01/03 00:54:49 scott Exp scott$

Contents

0.1 Introduction . . . 6
0.1.1 What's New . . . 6
0.1.2 What needs work . . . 7
0.1.3 Conventions . . . 7
0.1.4 Example Code . . . 7
0.2 Data Searching and Storage . . . 9
0.2.1 Linear Search . . . 9
0.2.2 Binary Search . . . 11
0.2.3 Binary Search Trees . . . 14
0.2.4 Red-Black Trees . . . 18
0.2.5 B+-Trees . . . 21
0.2.6 Boyer-Moore String Searching . . . 21
0.2.7 Hashing . . . 24
0.2.8 Skip lists . . . 38
0.2.9 Tries . . . 44
0.2.10 Quadtrees and Octrees . . . 48
0.3 Sorting Algorithms . . . 53
0.3.1 The Quicksort . . . 54
0.3.2 The Mergesort . . . 61
0.3.3 Heapsort . . . 64
0.3.4 Benchmarking the Quicksort and the Heapsort . . . 68
0.3.5 Insertion Sort . . . 76

0.3.6 Shellsort . . . 77
0.3.7 Selection Sort . . . 78
0.3.8 Bubble Sort . . . 79
0.3.9 Bucket and Radix Sorting . . . 80
0.3.10 The nth Largest . . . 81
0.4 Graph Algorithms . . . 83
0.4.1 Graph Representation . . . 83
0.4.2 Graph Traversal . . . 84
0.4.3 Floyd's Algorithm - Shortest Paths . . . 86
0.4.4 Dijkstra's Algorithm - Shortest Path . . . 87
0.4.5 Spanning Trees . . . 92
0.4.6 Transitive Closure . . . 96
0.5 Miscellaneous Algorithms . . . 96
0.5.1 Maximum Consecutive Subsequence . . . 96
0.5.2 Permutations . . . 101
0.5.3 Combinations . . . 103
0.5.4 Exponentiation . . . 106
0.5.5 Julian Calendar Algorithms . . . 107
0.5.6 Greatest Common Divisor, Least Common Multiple . . . 110
0.5.7 Addition Chaining . . . 111
0.5.8 Fibonacci Calculation . . . 112
0.5.9 Cartesian and Polar Coordinates . . . 114
0.5.10 Soundex English word-sounding Algorithm . . . 115
0.5.11 Metaphone Algorithm . . . 117
0.5.12 A Pseudo-Random Number Generator . . . 124
0.5.13 Horner's Rule . . . 127
0.5.14 Chinese Remainder Theorem . . . 127
0.5.15 Large Prime Number Generation . . . 127

0.5.16 Fast Fourier Transform 127
0.6 Formal Language Parsing Algorithms 127
0.6.1 Recursive Descent Parsing 127
0.6.2 Cheatham Sattley Method 127
0.6.3 Samuelson-Bauer xpression analysis 128
0.6.4 Shift-reduce bottom-up parsing 148
0.7 Operating Systems Algorithms 148
0.7.1 Dijkstra's Banker's Algorithm for Deadlock Prevention 148
0.7.2 Dekker's Algorithm for Mutual Exclusion 174
0.7.3 Peterson's Algorithm for Mutual Exclusion 174
0.8 Geometric Algorithms 174
0.8.1 Convex Hull Problem 174
0.8.2 Closest Pair Problem 174
0.8.3 Determining Whether a Point is Inside a Polygon 174
0.8.4 Knapsack Problem 175
0.9 Data Encryption Algorithms 178
0.10 Data Compression Algorithms 178
0.10.1 Run-Length Encoding 179
0.10.2 Integer Coding 189
0.10.3 Huffman Compression 208
0.10.4 Adaptive Huffman Compression 227
0.10.5 Sliding Window Compression 229
0.10.6 Lempel-Ziv-Walsh Compression 237
0.10.7 Arithmetic Compression 237
0.11 Game-Playing Algorithms 237
0.11.1 Minimax Search 237
0.11.2 Alpha-Beta pruning 237
0.12 Data Integrity 237
0.12.1 Checksums 237
0.12.2 Weighted Checksums 244
0.12.3 Cyclic Redundancy Checks 245

0.1 Introduction

This document is a collection of common computing algorithms and their implementations. It was not originally meant to be a formal study in the efficiency or analysis of algorithms but recently I have been augmenting the discussion of the presented algorithms with formal analysis. This is by no means an all-encompassing collection of algorithms. Rather it is intended to be a cookbook of popular algorithms, a discussion of how they work, and a collection of useful implementations.

It is far from complete and contains many bugs. I am actively working on this collection in my spare time so, now and then, you will notice changes. Being that it is a work-in-progress, however, please excuse any rough edges. The wording of my descriptions needs polishing in many places, the implementations given may be buggy, and the overall content is not (and may never be) as complete as I would like.

This document is available as a bunch of HTML files and in Adobe Postscript format. This document is also available in LaTeX format but to run it through LaTeX it you will need the LaTeX2HTML style files included with that translator. If you want the original LaTeX, e-mail me.

All pages in this collection (with the exception of those which were written by someone else and are thus labeled) are protected by the following copyright: Copyright ©1996-1998 by Scott Gasch. Except as otherwise indicated elsewhere in this document, with respect to a particular portion, file, or document, any person is hereby authorized to view, copy, print, and distribute this document subject to the following condition: The above copyright notice must appear in any copy of the document or portion thereof. Sections of code donated to this document are protected by the original copyright of their author(s).

All brands or product names appearing in this document are trademarks, registered trademarks, or service marks of their respective owners.

If you have a suggestion about the direction of the document, find a mistake, have an algorithm you would like to see included, or have an implementation that you would like to donate, please e-mail me at scott@wannabe.guru.org

0.1.1 What's New

I'm working on the data compression section and intend to work on the data encryption section after that. I just bought two great books: *The Data Compression Book* by Nelson and Gailly and *Applied Cryptography* by Schneier. I'd really recommend them both. See the references sections in the appropriate algorithms for more details.

Added implementation of Dijkstra's shortest path algorithm. Fixed many mistakes. Cleaning up LaTeXcode.

I've been working on the back-end LaTeXto make it translate better and cause the translator to complain less.

I've just gotten re-connected to the net so the biggest new item is that the page now has a home again. Other than that, I am working on getting the conversion of the document from LaTeXto HTML working better. If you are interested, I use a cool program called LaTeX2HTML.

0.1.2 What needs work

The searching, sorting, misc, and data integrity sections are all pretty complete. The comression alrogithms are almost complete. It would be nice to have a section about JPEG or MPEG but I don't know enough to write them.

The graph algorithms section needs a total re-write. The implementation are all buggy. Some of the algorithms at the end of the Misc section need sections written and code. The language parsing section needs to be written. The OS section is missing sections about Dekker and Peterson. The geometric section is very sparse. The encryption section is not written at all. The game playing section sucks.

0.1.3 Conventions

Algorithms here discussed are broken down into what I believe to be a logical outline. Each algorithm presented is first discussed at a conceptual level. Where appropriate an analysis of the complexity of an algorithm usually follows. Finally, in most cases, an implementation concludes the section dealing with a particular algorithm.

Words in **boldface** are key ideas or terms. These are also, generally, the words included in the document's index. Words in `terminal` type are either system commands or source code.

0.1.4 Example Code

The example code in this document may or may not compile as is. The reason I include it is to give a concrete implementation of an angorithm or idea. You may, of course, use the code however you want. The code that is intended to compile was built on a FreeBSD machine with `gcc 2.7.2`. In order to compile certain examples you may need this header file, called `global.h`:

```
#ifndef TYPEDEFS_
#define TYPEDEFS_

#ifndef FAILURE
#define FAILURE -1
#endif

#ifndef DWORD
typedef unsigned int DWORD;
#endif

#ifndef ULONG
typedef unsigned long ULONG;
#endif

#ifndef BOOL
```

```
typedef int BOOL;
#endif

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

#ifndef VOID
typedef void VOID;
#endif

#ifndef CHAR
typedef unsigned char CHAR;
#endif

#ifndef UCHAR
typedef unsigned char UCHAR;
#endif

#ifndef BYTE
typedef unsigned char BYTE;
#endif

#ifndef HANDLE
typedef int HANDLE;
#endif

#define IN
#define OUT
#define INOUT

#endif
```

Also, here is `debug.h`:

```
#ifndef _DEBUG_H_
#define _DEBUG_H_

#ifdef DEBUG
VOID _assert(IN const DWORD dwFile, IN const DWORD dwLine);
#define DPRINT((x)) perror(x)
```

```
#define ASSERT((x))      if (x)    \
                         { ; }     \
                         else      \
                         { _assert(__FILE__, __LINE__); }
#else
#define DPRINT
#define ASSERT
#endif
#endif
```

0.2 Data Searching and Storage

0.2.1 Linear Search

This method of searching for data in an array is straightforward and easy to understand. To find a given item, begin your search at the start of the data collection and continue to look until you have either found the target or exhausted the search space. Clearly to employ this method you must first know where the data collection begins and the size of the area to search. Alternatively, a unique value could be used to signify the end of the search space. This method of searching is most often used on an array data structure whose upper and lower bounds are known.

The complexity of this type of search is $O(N)$ because, in the worst case all items in the search space will be examined. This type of search is $\Theta(n/2)$ as, in the average case, one-half of the items in the search space will be examined before a match is found. As we will see in later sections, there are many algorithms for improving search time that can be used in place of a linear search. For instance, the **binary search** algorithm operates much more efficiently than a linear search but requires that the data being searched be in sorted order. Because there are faster ways of searching a memory space, the linear search is sometimes referred to as a **brute force** search.

Source Code

Below is an implementation of a linear search of an array of integers written C. Since this is the first program in the document I will preface it by saying that many of the examples will not compile and are simply illustrations of a concept. This one, and others, will compile, however. But to build most of the programs in this document you will need the `global.h` file from the introduction.

```
#include <stdio.h>
#include <stdlib.h>

//
// See introduction
//
```

```
#include "global.h"

#define NUM_ELEMENTS               100

DWORD g_rgdwSearchSpace[NUM_ELEMENTS];
DWORD dwLcv;
BOOL fFound = NO;
DWORD dwTarget;

int main(VOID)
{
  srand(1);

  //
  // initialize the array
  //
  for (dwLcv = 0; dwLcv < NUM_ELEMENTS; dwLcv++)
  {
    rgdwSearchSpace[dwLcv] = rand();
    printf(" %d:   %d\n", dwLcv, rgdwSearchSpace[dwLcv]);
  }

  //
  // ask which one they want to search for
  //
  printf("Search for: ");
  scanf("%d", &dwTarget);

  //
  // now find it -- linear search
  //
  dwLcv = 0;
  while ((dwLcv < NUM_ELEMENTS) && !fFound)
  {
    if (rgdwSearchSpace[i] == dwTarget)
    {
      printf("Found at element %d!\n", dwLcv);
      fFound = TRUE;
    }
    dwLcv++;
  }

  if (FALSE == fFound)
  {
    printf("Sorry, not found.");
  }
}
```

0.2.2 Binary Search

When it is known that a data set is in sorted order it is possible to drastically increase the speed of a search operation in most cases. The following section deals with an array-based implementation of the **binary search** algorithm.

In a later section we will look more closely at the **binary search tree** data structure and associated algorithm.

Both of these methods take advantage of the fact that the search space is in some order to limit the area in which the target item is known to reside.

An array-based binary search selects the median (middle) element in the array to and compares its value to that of the target value. Because the array is known to be sorted, if the target value is less than the middle value then the target is known to be in the first half of the array. Likewise if the value of the target item is greater than that of the middle value in the array, it is known that the target lies in the second half of the array. In either case we can, in effect, "throw out" one half of the search space with only one comparison.

Now, knowing that the target must be in one half of the array or the other, the binary search examines the middle value of the half in which the target must lie. The algorithm thus narrows the search area by half at each step until it has either found the target data or the search fails.

The algorithm is easy to remember if you think about a child's guessing game. Imagine I told you that I am thinking of a number between 1 and 1000 and asked you to guess the number. Each time you guessed I would tell you "higher" or "lower." Of course you would begin by guessing 500, then either 250 or 750 depending on my response. You would continue to refine your choice until you got the number correct.

Analysis

A binary search on an array is $O(log_2 n)$ because at each test you can "throw out" one half of the search space. If we assume n, the number of items to search, is 2^x then, given that n is cut in half at every comparisons, the most comparisons needed to find a single item in n is x. It is noteworthy that for very small arrays a **linear search**can prove faster than a binary search. However as the size of the array to be searched increases the binary search is the clear victor in terms of number of comparisons and overall speed.

Still, the binary search has some drawbacks. First of all, it requires that the data to be searched be in sorted order. If there is even one element out of order in the data being searched it can throw off the entire process. When presented with a set of unsorted data the efficient programmer must ponder whether to sort the data and apply a binary search or simply apply the less-efficient linear search. Even the best sorting algorithm is a complicated process. You must decide if the cost of sorting the data is worth the increase in search speed gained with the binary search. If you are searching only once, it is probably better to do a linear search.

Once the data is sorted it can also prove very expensive to add or delete items. In a sorted array, for instance, such operations require a **ripple-shift** of array elements to open or close a "hole" in the array. This is an expensive operation as it requires, in worst case, n comparisons and n item moves.

The binary search assumes easy random-access to the data space it is searching. An array is the data structure that is most often used because it is easy to jump from one index to another in an array. It is difficult, on the other hand, to efficiently compute the midpoint of a **linked list** and then traverse there inexpensively. The **binary search tree** data structure and algorithm, which we discussed later, attempt to solve these array-based binary search weaknesses.

Source Code

Below is an iterative implementation of the binary search in C. The binary search algorithm can be implemented as a recursive algorithm also.

```
#include <stdio.h>
#include <stdlib.h>

#include "global.h"
#include "debug.h"

typedef int KEY;
#define NOTFOUND -1
extern int NumElements(KEY *x);  // to return the # of elements in an array


//
// Search for the target value in an array.  Return the index on success and
// NOTFOUND, or -1, on failure.
//
DWORD bsearch(KEY kTarget, KEY *pkSearchspace) {

  BOOL fFound = NO;                          // have we found it yet?
  DWORD dwFirst = 0;                         // search space starts at index zero
  DWORD dwLast = NumElements(searchspace); // # of elements in search space
  DWORD dwMidpoint;                          // current midpoint

  //
  // Check return value of NumElements and make precondition assertions
  //
  ASSERT(dwLast >= dwFirst);
  ASSERT(pkSearchspace);

  //
  // iterative implementation -- could also be done recursively
  //
```

```
  while ((dwFirst <= dwLast) && (!fFound))
  {

    //
    // calculate the dwMidpoint of the current [sub]range
    //
    dwMidpoint = (dwFirst + dwLast) / 2;

    //
    // compare the target with the value of dwMidpoint element
    //
    if (pkSearchspace[dwMidpoint] == kTarget)
    {
      fFound = YES;
    }
    else
    {
      //
      // if the dwMidpoint is not the target adjust the range accordingly
      //
      if (target < searchspace[dwMidpoint])
      {
dwLast = dwMidpoint - 1;
      }
      else
      {
dwFirst = dwMidpoint + 1;
      }

  }

  //
  // Return value
  //
  if (fFound)
  {
    return(dwMidpoint);
  }
  else
  {
    return(NOTFOUND);
  }
}
```

References

1. Brassard, Gilles and Bratley, Paul. 1988, *Algorithmics: Theory and Practice.* (Englewood Cliffs, NJ: Prentice-Hall), pp. 109-114.

2. Manber, Udi. 1989, *Introduction to Algorithms: A Creative Approach.* (Reading, MA: Addison-Wesley), pp. 120-125.

0.2.3 Binary Search Trees

A **binary search tree** builds on the concept of the **binary search algorithm**. Binary search trees, often called "BSTs," retain the excellent search speed of the binary search of an array but, because of their dynamic structure, also enjoy more efficient insertion and deletion operations. Whereas to insert or delete an item from a sorted array requires an expensive **ripple-shift** operation, to add or delete an item from a BST requires only a few simple pointer operations and a call to the dynamic memory allocator.

A BST is a special type of **binary tree**; this means that every **node** in the tree has a value and zero, one, or two **children nodes**. BSTs have the added property that given a node, X, all children nodes to X's left have smaller values than X whereas all children nodes to X's right have a larger value than X.

In order to search for a particular value in a BST, traverse the tree starting from the **root node**.By comparing each node's value with the one for which you are searching, and then moving in the correct direction (depending on the result of the comparison) it is possible to quickly determine whether a target value is in the tree or not.

Each comparison and subsequent traversal results in reducing the number of items yet to be searched by one-half in a **balanced binary search tree**.A balanced BST is one where every node has either two children or none; this is the ideal shape for a BST. A **degenerate binary search tree** is one in which every node has only one child. This undesirable type of BST looks and behaves more like a **linked-list** than a tree and suffers from lackluster search performance. Such a degenerate tree could be built, for example, building a BST out of data already in sorted order.

The complexity of the average search operation for data in a BST is $\Theta(log_2 n)$ which, you will note, is the same complexity as a search operation in an array using the binary search procedure. The worst case search performance of the BST data structure is $O(n)$ because of the possibility of searching a degenerate tree.

Inserting and Deleting Nodes

To create a BST begin with a null pointer to the non-existent **root node**. Insert data values by examining the value of the root node and moving left or right down the tree until you have reached the proper insertion point. If there is no root node, (i.e. the tree is empty) the very first node inserted becomes the new root node. Ideally this first value should be the median value in the data set as this will cause the number of nodes on each side of the root to be equal.

To delete a node from a BST you must not only find the node in the tree by traversing in the usual manner, but also ensure that the BST retains the "binary search tree property." If the node to be deleted, node d, has no children, it can be deleted outright. If node d has only one child, promote the child and use it to overwrite the contents of node d. If node d has two children, however, things get a bit tricky.

Deleting a node, d, that has two children requires that we find the node, e, in the tree with the next higher or lower value. To do so, traverse in one direction to one of d's children and then continually traverse in the opposite direction until you can go no further. For example, traverse to d's left child and then continue to traverse to right children as far as possible. Swap the last node with d then delete d. In essence, make d's value that of the last node and then delete the last node.

Source Code

The following double pointer implementation of a BST was written by Thomas Niemann and is available on his algorithm collection webpages.

```
#include <stdio.h>
#include <stdlib.h>

/* modify these lines to establish data type */
typedef int T;
#define CompLT(a,b) (a < b)
#define CompEQ(a,b) (a == b)

typedef struct Node_ {
    struct Node_ *Left;         /* left child */
    struct Node_ *Right;        /* right child */
    struct Node_ *Parent;       /* parent */
    T Data;                     /* data stored in node */
} Node;

Node *Root = NULL;               /* root of binary tree */

Node *InsertNode(T Data) {
    Node *X, *Current, *Parent;


   /**********************************************
    *  allocate node for Data and insert in tree  *
    **********************************************/

    /* setup new node */
    if ((X = malloc (sizeof(*X))) == 0) {
        fprintf (stderr, "insufficient memory (InsertNode)\n");
        exit(1);
```

```
    }
    X->Data = Data;
    X->Left = NULL;
    X->Right = NULL;

    /* find X's parent */
    Current = Root;
    Parent = 0;
    while (Current) {
        if (CompEQ(X->Data, Current->Data)) return (Current);
        Parent = Current;
        Current = CompLT(X->Data, Current->Data) ?
            Current->Left : Current->Right;
    }
    X->Parent = Parent;

    /* insert X in tree */
    if(Parent)
        if(CompLT(X->Data, Parent->Data))
            Parent->Left = X;
        else
            Parent->Right = X;
    else
        Root = X;

    return(X);
}
void DeleteNode(Node *Z) {
    Node *X, *Y;

   /*****************************
    *  delete node Z from tree  *
    *****************************/

    /* Y will be removed from the parent chain */
    if (!Z || Z == NULL) return;

    /* find tree successor */
    if (Z->Left == NULL || Z->Right == NULL)
        Y = Z;
    else {

        Y = Z->Right;
        while (Y->Left != NULL) Y = Y->Left;
    }

    /* X is Y's only child */
```

```
    if (Y->Left != NULL)
        X = Y->Left;
    else
        X = Y->Right;

    /* remove Y from the parent chain */
    if (X) X->Parent = Y->Parent;
    if (Y->Parent)
        if (Y == Y->Parent->Left)
            Y->Parent->Left = X;
        else
            Y->Parent->Right = X;
    else
        Root = X;

    /* Y is the node we're removing */
    /* Z is the data we're removing */
    /* if Z and Y are not the same, replace Z with Y. */
    if (Y != Z) {
        Y->Left = Z->Left;
        if (Y->Left) Y->Left->Parent = Y;
        Y->Right = Z->Right;
        if (Y->Right) Y->Right->Parent = Y;
        Y->Parent = Z->Parent;
        if (Z->Parent)
            if (Z == Z->Parent->Left)
                Z->Parent->Left = Y;
            else
                Z->Parent->Right = Y;
        else
            Root = Y;
        free (Z);
    } else {
        free (Y);
    }
}

Node *FindNode(T Data) {

   /*******************************
    *  find node containing Data  *
    *******************************/

    Node *Current = Root;
    while(Current != NULL)
        if(CompEQ(Data, Current->Data))
            return (Current);
```

```
        else
            Current = CompLT (Data, Current->Data) ?
                Current->Left : Current->Right;
    return(0);
}
```

References

1. Manber, Udi. 1989, *Introduction to Algorithms: A Creative Approach.* (Reading, MA: Addison-Wesley), pp. 70-74.

2. Neimann, Thomas. 1998, *Sorting and Searching Algorithms: A Cookbook* (http://www.geocities.com/SoHo/2167/book.html).

0.2.4 Red-Black Trees

Binary search trees are superior to array-based **binary searches** because they make addition and deletion from the data storage structure less costly. However, as discussed in the binary search tree section, it is possible to create **degenerate trees** which suffer from poor search operation performance. These degenerate trees look like linked lists; each node in the tree has one child causing the data lookup to be a linear operation. As said in the binary search tree section, the optimal shape for a search tree data structure is a **balanced tree** or one in which each node in the tree has either two or no child nodes.

The red-black tree algorithm is a method for maintaining balanced search trees. The algorithm is actually based on AVL trees which are data structures that were proposed by G. M. Adel'son-Vel'skii and E. M. Landis. As you might expect, in a red-black tree every node is given a color – either red or black. Further, in a red-black tree, unlike in a binary search tree, data is only stored at the **leaf nodes** of the data structure. **Internal nodes** are used as guides or reference points.

A node's color is determined by the following heuristics:

1. All leaf nodes in a red-black tree must be black.

2. On any path from the root of the tree to a leaf, two consecutive red colored nodes are not allowed.

3. All the leaves of the tree must have the same number of black nodes on the path from the root of the red-black tree to that leaf. This number of black nodes is called the **black depth** of a node.

Insertion Operation

The insertion of a new node into a red-black tree can be a complicated procedure.

In order to insert a new element it is mandatory to first determine the point of insertion. To do this we traverse the tree as if it were a binary search tree. Nodes less than or equal to the current node in

value cause us to traverse to the left child (whereas nodes greater than the current node in value cause us to traverse right). As we move down the tree we perform what is known as a **color flip** operation. Every time a black node with two red children is encountered we make the parent node red and the two children black. This operation does not change the black height of the tree and helps ensure easier insertion in most cases. Such a color flip, however, can produce two red nodes in a row. Recall that two consecutive red nodes on any path from root to leaf is not allowed in this structure. If two consecutive red nodes are produced a tree-rebalancing operation must ensue.

In order to eliminate pairs of consecutive red nodes from a red-black tree a rotation operation is used. A rotation operation is performed in a series of steps:

1. Make the current node's right child equal to the current node's right child's left child.
2. Make the current node's right child's parent the current node's parent.
3. Make the current node's right child's left child the current node.

That's a mouthful. Here's the source code to a rotate left function; you may find it easier to read:

```
void rotate_left(node *current_node)
{
    node *child = current_node->right;

    /* establish current_node->right link */
    current_node->right = child->left;
    child->left->parent = current_node;

    /* establish child->parent link */
    child->parent = current_node->parent;
    if (current_node->parent)
    {
        if (current_node == current_node->parent->left)
            current_node->parent->left = child;
        else
            current_node->parent->right = child;
    }
    else
    {
        root = child;
    }

    /* link child and current_node */
    child->left = current_node;
    current_node->parent = child;
}
```

The code for a `rotate_right` operation is what you would expect and is given in the full red-black source code in the ensuing section. There are only a few cases in which two red nodes exist on a path from root to leaf after a color flip operation. See the code for `insert_fix` for a case-by-case study.

When inserting a node in a leaf-with-black-parent situation simply create a new red node between the parent and leaf. The leaf is now a child of this new red node and the node you wanted to insert is the other child.

It is not always possible to arrive at a black-leaf, black-parent situation, though. In this case the node insertion happens the same way (i.e. create a new red node between the leaf and its parent; insert as children of this new red node) but the insertion process is followed by an immediate tree re-balancing as it created two red nodes in a row.

Insertion is a difficult operation to verbalize; the best way to get a handle on what is going on is to make sure you understand the rules of a red-black tree, make sure you understand the rotation operation, and look at the source code below.

Deletion

Deletion is about as complicated as insertion. In fact, as you might expect, deletion is, in essence, a nearly an exact reciprocal operation. We would always like to delete nodes with red parents and to do so we are willing to manipulate the tree via rotations and re-coloring operations.

When looking for a node to delete begin at the root of the structure. If neither of the root's children is red, color the root red. Then, at each node on the path to the node to be deleted, force the node to become red.

Source Code

The below red-black tree implementation of was written by Thomas Niemann and is available on his algorithm collection webpages. This code is not subject to copyright restrictions.

```
-black/red-black.c
```

References

1. Binstock, Andrew and Rex, John. 1995, *Practical Algorithms for Programmers* (Reading, MA: Addison-Wesley), pp. 281-287.

2. Neimann, Thomas. 1998, *Sorting and Searching Algorithms: A Cookbook* (http://www.geocities.com/SoHo/2167/book.html).

0.2.5 B+-Trees

A B+-tree is a data structure to store vast amounts of information. Typically B+-trees are used to store amounts of data that will not fit in main system memory. To do this, secondary storage (usually disk) is used to store the leaf nodes of the tree. Only the internal nodes of the tree are stored in computer memory. In a B+-tree the leaf nodes are the only ones that actually store data items. All other nodes are called **index nodes** or **i-nodes** and simply store "guide" values which allow us to traverse the tree structure from the root down and arrive at the leaf node containing the data item we seek. Because disk I/O is very slow in comparison to memory access these leaf nodes store more than one data item each. In fact, the data structure will perform best when the size of the leaf nodes closely approximates the size of a disk sector under most operating systems. Thus, when we search a B+-tree (by traversing from the root node down to the proper data node) we still must read that data node from the disk and search its contents. Another way to improve the speed of a query operation is to keep a memory cache of recently read leaf nodes.

The ancestor of the B+-tree is a structure known as a B-tree in which data items can be stored in any node on the tree. A more complicated and slightly more robust varient of the B-tree is called a B*-tree.

While conceptually simple in nature, the challenging aspect of B+-trees is effecting their growth and collapse. When data nodes in the tree become too full they split into two nodes. This process demands that a new guide value be added to the parent index node so that future queries can arbitrate between the two new nodes. However, adding this new guide value to the parent may cause it, in turn, to split. In fact, all the nodes on a path from a leaf to the root may split when you add a new value to a leaf node. If the root node splits, a new leaf node is created and your tree grows by one level.

Conversely, the deletion of data items in a leaf node may cause that node to become too empty. When a data node becomes too empty, neighboring nodes are examined and merged into underfull node. This causes the deletion of a guide value in the parent index node which, in turn, may cause it to become too empty. Much like the ripple caused by an insertion operation, a key deletion may cause a merge-delete wave to run from a leaf node all the way up to the root. When the children of the root are merged into one and the root node's only value is deleted the root index node is deallocated, its child node becomes the new root, and the tree shrinks by one level.

As you can see, insertion and deletion processes are recursive in nature and can cascade up or down the B+-tree affecting its shape dramatically.

Source Code

```
// Look in Rex & Binstock ``Practical Algorithms for Programmers'' for
// B-Tree implementation.  Send me a working B+-Tree implementation
// please!
```

0.2.6 Boyer-Moore String Searching

The Boyer-Moore searching algorithm, described in R. S. Boyer and J. S. Moore's 1977 paper *A Fast String Searching Algorithm* is among the best ways known for finding a substring in a search space.

Using their method it is possible to search a data space for a known pattern without having to examine all the characters in the search space. Boyer-Moore search algorithms are based on two search heuristics.

The first of these rule tells us how to search for substrings without repeats in a data space. Keep a pointer into the data space at the current search location; initialize this pointer to the start of the space plus $n-1$ characters where n is the number of characters in the target string.

Compare the character in the data space pointed to by this pointer with the characters in the target string. If this character does not occur in the target string, advance the pointer by n places.

If the character does occur in the target string, advance the pointer by $n-p$ places where p is the position that the character in question first occurs in the target string.

This process repeats until either a match is found or we have shifted over past the end of the search space.

The second search heuristic applies to searching for targets with repeating patterns. Using only the rules set forth in the first heuristic will work for targets with repeating patterns but the search will not be as efficient as possible. By examining partial matches and repeats in the target string, though, it is possible to make more drastic pointer jumps and arrive at the match more rapidly. This type of jump is based on a table which is computed before the search begins.

Source Code

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>

#define alphabet_size (UCHAR_MAX + 1)

#ifndef max
  #define max(a,b) ((a) > (b) ? (a) : (b))
#endif

char *boyer_moore_search (char *target, char *text, size_t length) {
  unsigned char_jump[alphabet_size];
  unsigned *match_jump;
  unsigned *backup;
  size_t pat_len;
  unsigned u, u_text, u_pat, ua, ub;

  pat_len = strlen(target);
  match_jump = (unsigned *) malloc (2 * sizeof(unsigned) * (pat_len + 1));
  backup = match_jump + pat_len + 1;

  /* heuristic #1 setup, simple text search */
```

```
  memset (char_jump, 0, alphabet_size * sizeof(unsigned));
  for (u = 0; u < pat_len; u++)
    char_jump[((unsigned char) target[u])] = pat_len - u - 1;



  /* heuristic #2 setup, repeating pattern search */

  for (u = 1; u <= pat_len; u++)
    match_jump[u] = 2 * pat_len - u;

  u = pat_len;
  ua = pat_len + 1;
  while (u > 0) {
    backup[u] = ua;
    while (ua <= pat_len && target[u - 1] != target[ua - 1]) {
      if (match_jump[ua] > pat_len - u) match_jump[ua] = pat_len - u;
      ua = backup[ua];
    }
    u--; ua--;
  }

  for (u - 1; u <= ua; u++)
    if (match_jump[u] > pat_len + ua - u) match_jump[u] = pat_len + ua - u;

  ub = backup[ua];
  while (ua <= pat_len) {
    while (ua <= ub) {
      if (match_jump[ua] > ub - ua + pat_len)
match_jump[ua] = ub - ua + pat_len;
      ua++;
    }
    ub = backup[ub];
  }


  /* perform search */
  u_pat = pat_len;
  u_text = pat_len - 1;
  while (u_text < text_len && u_pat != 0) {
    if (text[u_text] == target[u_pat - 1]) {
      u_text--;
      u_pat--;
    } else {
      ua = char_jump[((unsigned char) text[u_text])];
      ub = match_jump[u_pat];
      u_text += max(ua, ub);
```

```
      u_pat = pat_len;
    }
  }

  if (u_pat == 0) {
    return((char *) (text + (u_text + 1)));
  } else {
    return(NULL);
  }
}
```

References

1. Binstock, Andrew and Rex, John. 1995, *Practical Algorithms for Programmers* (Reading, MA: Addison-Wesley), pp. 102-111

0.2.7 Hashing

A **hash table** is a data structure used to maximize the speed with which a stored data item can be accessed. Operations typically defined for hash tables are item insertion, lookup, and deletion. The complexity of an average search operation is $\Theta(1)$ – constant time. In many hash schemes a worst case bound for search operation complexity is $O(n)$. The price for such speed, as we will see, is space.

A hashing table data structure is usually implemented as a large array. Whenever an item must be added to the hash table it is **not** simply added at the end of the structure. Instead a function called a **hash function** determines the location at which a given item is stored based, somehow, on some attribute of the item in to be stored. This computed location is called an item's **hash value**. The value being hashed is sometimes called the **pre-image**.

In order to insert a new item into a hash table it is necessary to compute the hash value of the item. Once the hash value of the item is known, the insertion routine looks in the table at the location specified by the hash value. If this table location is empty the new item can be inserted outright. However if this location in the table is already in use by some other data item an alternate strategy must be used. This occurrence is known as a **collision** and, as we will see, there are many ways of handling collisions.

Looking up an item in the hash table entails computing where the item would be stored by finding its hash value then checking the table at that location. Deleting an item is just as straightforward.

Selecting a Hash Function

Choosing a good hash function is of the utmost importance. An **uniform** hash function is one that equally distributes data items over the whole hash table data structure. If the hash function is poorly chosen data items may tend to **clump** in one area of the hash table and many collisions will ensue. A non-uniform dispersal pattern and a high collision rate cause an overall data structure performance

degradation. There are several strategies for maximizing the uniformity of the hash function and thereby maximizing the efficiency of the hash table.

One method, called the **division method** , operates by dividing a data item's key value by the total size of the hash table and using the remainder of the division as the hash function return value. This method has the advantage of being very simple to compute and very easy to understand.

Selecting an appropriate hash table size is an important factor in determining the efficiency of the division method. If you choose to use this method, avoid hash table sizes that simply return a subset of the data item's key as the hash value. For instance, a table one-hundred items large will result put key value 12345 at location forty-five, which is undesirable. Further, an even data item key should not always map to an even hash value (and, likewise, odd key values should not always produce odd hash values). A good rule of thumb in selecting your hash table size for use with a division method hash function is to pick a prime number that is not close to any power of two (2, 4, 8, 16, 32...).

```
int hash_function(data_item item)
{
  return item.key % hash_table_size;
}
```

Sometimes it is inconvenient to have the hash table size be prime. In certain cases only a hash table size which is a power of two will work. A simple way of dealing with table sizes which are powers of two is to use the following formula to computer a key: $k = (xmodp)modm$. In the above expression x is the data item key, p is a prime number, and m is the hash table size. Choosing p to be much larger than m improves the uniformity of this key selection process.

Yet another hash function computation method, called the **multiplication method**, can be used with hash tables with a size that is a power of two. The data item's key is multiplied by a constant, k and then bit-shifted to compute the hash function return value.

A good choice for the constant, k is $N * (sqrt(5) - 1)/2$ where N is the size of the hash table.

The product $key * k$ is then bitwise shifted right to determine the final hash value. The number of right shifts should be equal to the $log_2 N$ subtracted from the number of bits in a data item key. For instance, for a 1024 position table (or 2^{10}) and a 16-bit data item key, you should shift the product $key * k$ right six (or 16 - 10) places.

```
int hash_function(data_item item)
{
  extern int constant;
  extern int shifts;

  return (int)((constant * item.key) >> shifts);
}
```

Note that the above method is only effective when all data item keys are of the same, fixed size (in bits). To hash non-fixed length data item keys another method is **variable string addition** so named because it is often used to hash variable length strings. A table size of 256 is used. The hash function works by first summing the ASCII value of each character in the variable length strings. Next, to determine the hash value of a given string, this sum is divided by 256. The remainder of this division will be in the range of 0 to 255 and becomes the item's hash value.

```
int hash_function (char *str)
{
  int total = 0;

  while (*str) {
    total += *str++;
  }
  return (total % 256);
}
```

Yet another method for hashing non fixed-length data is called **compression function** and discussed in the one-way hashing section.

Dealing with Collisions

No matter how good your hash function is, or how carefully you choose the hash table size, sometimes data collisions are bound to occur. Recall that a collision is when two distinct data items produce the same hash value and, thus, want to be stored in the same table location. For instance, suppose you have an item, A, already in your hash table at location 144. A's hash function value, therefore, is 144. Another item, B, with a totally different key than A, is to be added to the table. Suppose, however, that B's hash value is 144 also. B cannot be stored at table location 144 because A is already there... or can it?

One way of dealing with this situation is to make each location in the hash table the head of a **linked-list** data structure. If you find a collision has occurred, traverse down the linked list at the hash value and add the new data item to the list's tail (or head). Of course, when you search for an item in a table using this insertion scheme you must be mindful of the fact that such an item may not be at the head of the linked list residing in the hash table at the hash location. This method is sometimes known as **open hashing** because multiple data items sharing the same hash value are stored "outside" the hash table.

The following methods all store collided data inside the hash table at different locations than their computed hash value. This practice is known as **closed hashing**.

The classic way to deal with collisions is to simply increment an item's hash value by one until a finding an unoccupied hash table location then store the item a location or two away from it's computed hash value. This method is known as **linear probing**. In querying a table employing such an insertion

scheme you have to not only check at the hash location of the item for which you are looking, but, if it contains some item other than the one you seek, continue to search in adjacent hash table locations until you either find your goal item or encounter an empty table location. Linear probing, while simple to understand and implement, leads to data clumping and is not the ideal way of handling collisions.

Different spins on the concept of probing are known as **quadratic probing** and **random probing**. In quadratic probing instead of simply moving one address down in the hash table the number of spaces moved is somehow dependent on the number of times that we have moved so far. For example, consider the following hash value offset function:

$$o(i) = 2^i - 1$$

The first collision adds one address to the base hash address, the second three, the third seven, and so on. When quadratic probing is employed, collisions tend not to produce the **clusters** of full areas in the hash table which are linear probing's drawback.

Random probing uses the address of the collision as the seed for a pseudo-random number generator and computes the next address to try with a function which takes a random element into account. Both quadratic and random probing are usually slower than linear probing but produce more uniform hash data distributions.

One final way of dealing with collisions is called **rehashing** or **dualhashing**. If the hash address produced is a collision, the address is reused as input to the hash function in order to compute a new address. This process, as you might expect, complicates lookups and deletes.

Choosing a Table Size

When choosing a table size you are, again, weighing options. The smaller the table you use, the faster the access times will be. However, if you use too small a table the probability of a collision occurring increases. As mentioned in the previous section, if you are using the division method for calculating hash function value, the hash table size should be a prime number. The multiplication method relies on a hash table size that is a power of two.

Source Code

The following code was written by Thomas Niemann and is available on his algorithm collection web-pages.

```
#include <stdlib.h>
#include <stdio.h>

/* modify these lines to establish data type */
typedef int T;
#define CompEQ(a,b) (a == b)
```

```
typedef struct Node_ {
    struct Node_ *Next;         /* next node */

    T Data;                     /* data stored in node */
} Node;

typedef int HashTableIndex;

Node **HashTable;
int HashTableSize;

HashTableIndex Hash(T Data) {
    /* division method */
    return (Data % HashTableSize);
}

Node *InsertNode(T Data) {
    Node *p, *p0;
    HashTableIndex bucket;

    /* insert node at beginning of list */
    bucket = Hash(Data);
    if ((p = malloc(sizeof(Node))) == 0) {
        fprintf (stderr, "out of memory (InsertNode)\n");
        exit(1);
    }
    p0 = HashTable[bucket];
    HashTable[bucket] = p;
    p->Next = p0;
    p->Data = Data;
    return p;
}

void DeleteNode(T Data) {
    Node *p0, *p;
    HashTableIndex bucket;

    /* find node */
    p0 = 0;
    bucket = Hash(Data);
    p = HashTable[bucket];
    while (p && !CompEQ(p->Data, Data)) {

        p0 = p;
        p = p->Next;
    }
```

```
    if (!p) return;

    /* p designates node to delete, remove it from list */
    if (p0)
        /* not first node, p0 points to previous node */
        p0->Next = p->Next;
    else
        /* first node on chain */
        HashTable[bucket] = p->Next;

    free (p);
}

Node *FindNode (T Data) {
    Node *p;
    p = HashTable[Hash(Data)];
    while (p && !CompEQ(p->Data, Data))
        p = p->Next;
    return p;
}
```

One-way hashing

One-way hashing is used to generate a digital fingerprint of data. Such fingerprints are commonly used in digital signatures. For instance, if you electronically sign a contract, part of the signtaure affixed to the document includes a fingerprint of what you signed. This way someone cannot change the terms of the contract after the fact. Likewise, your digital signtaure cannot be lifted from one document and affixed to another, different document.

The "fingerprint" generated is a hash value. One-way hashing functions take input streams, called **pre-text**, and map them to hash values. A hashing algorithm, H, is "one-way" if it is computationally difficult to arrive at x such that $H(x) = h$. That is, if you have a known hash value you cannot reverse the process of computation and arrive at a document that has that hash value.

One of the challenges involved with creating one-way hash functions is that they must operate on input data streams of variable size. It should be possible to obtain a digital fingerprint of short and long input messages alike. In most traditional hashing systems input data is of a fixed length.

A common way for one-way hash functions to deal with the variable length input problem is called a **compression function**. Compression functions work by viewing the data being hashed as a sequence of n fixed-length blocks. To compute the hash value of a given block, the algorithm needs two things: the data in the block and an input seed. To begin, the input seed is set to some constant value, c, and the algorithm computes the hash value h_1 of the first block. Next, the hash value of the first block, h_1 is used as the seed for the second block. The function proceeds to compute the hash value of the second block based on the data in the second block and the hash value of the first block, h_1. So, the hash value for block n is related to the data in block n and the hash value h_{n-1} (for $n > 1$). The hash value of the entire input stream is the hash value of the last block.

Another problem for one-way hashing functions is to minimize the number of collisions. The logic being that if there are many collisions in the system then it will be easier to find two documents that produce a the same fingerprint. In order to accomplish this some one-way hash algorithms encode the length of the input stream at the tail end of the last block. This reduces the chances that two messages of different lengths will have the same hash value.

None of the techniques discussed thus far have anything to do with the "one-way" requirement, though. All of the techniques described can be reversed fairly easily. Popular one-way hashing algorithms handle this requirement in different ways. For example, the MD5 algorithm, designed by Ron Rivest, operates on 512 bit blocks and uses four **chaining variables**. The variables are initialized to different values. Then, for each 512 bit blocks, four rounds of operations are performed. Each round consists of sixteen operations. These operations involve computing a value using three of the four chaining variables as operands. This result is then added to the value of the fourth chaining variable, a subrange of bits in the current block, and a constant. Which three chaining variables take place in the operation rotates as the algorithm progresses.

The source code for the MD5 algorithm is included below:

```
/*
 * MD5C.C - RSA Data Security, Inc., MD5 message-digest algorithm
 *
 * Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All
 * rights reserved.
 *
 * License to copy and use this software is granted provided that it
 * is identified as the "RSA Data Security, Inc. MD5 Message-Digest
 * Algorithm" in all material mentioning or referencing this software
 * or this function.
 *
 * License is also granted to make and use derivative works provided
 * that such works are identified as "derived from the RSA Data
 * Security, Inc. MD5 Message-Digest Algorithm" in all material
 * mentioning or referencing the derived work.
 *
 * RSA Data Security, Inc. makes no representations concerning either
 * the merchantability of this software or the suitability of this
 * software for any particular purpose. It is provided "as is"
 * without express or implied warranty of any kind.
 *
 * These notices must be retained in any copies of any part of this
 * documentation and/or software.
 *
 * $Id: md5c.c,v 1.9 1998/03/27 10:22:01 phk Exp $
 *
 * This code is the same as the code published by RSA Inc.  It has been
 * edited for clarity and style only.
 */
```

```
#include <sys/types.h>

#ifdef KERNEL
#include <sys/systm.h>
#else
#include <string.h>
#endif

#include <sys/md5.h>

static void MD5Transform __P((u_int32_t [4], const unsigned char [64]));

#ifdef KERNEL
#define memset(x,y,z) bzero(x,z);
#define memcpy(x,y,z) bcopy(y, x, z)
#endif

#ifdef i386
#define Encode memcpy
#define Decode memcpy
#else /* i386 */

/*
 * Encodes input (u_int32_t) into output (unsigned char). Assumes len is
 * a multiple of 4.
 */

static void
Encode (output, input, len)
unsigned char *output;
u_int32_t *input;
unsigned int len;
{
unsigned int i, j;

for (i = 0, j = 0; j < len; i++, j += 4) {
output[j] = (unsigned char)(input[i] & 0xff);
output[j+1] = (unsigned char)((input[i] >> 8) & 0xff);
output[j+2] = (unsigned char)((input[i] >> 16) & 0xff);
output[j+3] = (unsigned char)((input[i] >> 24) & 0xff);
}
}

/*
 * Decodes input (unsigned char) into output (u_int32_t). Assumes len is
 * a multiple of 4.
```

```
 */

static void
Decode (output, input, len)
u_int32_t *output;
const unsigned char *input;
unsigned int len;
{
unsigned int i, j;

for (i = 0, j = 0; j < len; i++, j += 4)
output[i] = ((u_int32_t)input[j]) | (((u_int32_t)input[j+1]) << 8) |
    (((u_int32_t)input[j+2]) << 16) | (((u_int32_t)input[j+3]) << 24);
}
#endif /* i386 */

static unsigned char PADDING[64] = {
  0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};

/* F, G, H and I are basic MD5 functions. */
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
#define H(x, y, z) ((x) ^ (y) ^ (z))
#define I(x, y, z) ((y) ^ ((x) | (~z)))

/* ROTATE_LEFT rotates x left n bits. */
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32-(n))))

/*
 * FF, GG, HH, and II transformations for rounds 1, 2, 3, and 4.
 * Rotation is separate from addition to prevent recomputation.
 */
#define FF(a, b, c, d, x, s, ac) { \
(a) += F ((b), (c), (d)) + (x) + (u_int32_t)(ac); \
(a) = ROTATE_LEFT ((a), (s)); \
(a) += (b); \
}
#define GG(a, b, c, d, x, s, ac) { \
(a) += G ((b), (c), (d)) + (x) + (u_int32_t)(ac); \
(a) = ROTATE_LEFT ((a), (s)); \
(a) += (b); \
}
#define HH(a, b, c, d, x, s, ac) { \
(a) += H ((b), (c), (d)) + (x) + (u_int32_t)(ac); \
```

```
(a) = ROTATE_LEFT ((a), (s)); \
(a) += (b); \
}
#define II(a, b, c, d, x, s, ac) { \
(a) += I ((b), (c), (d)) + (x) + (u_int32_t)(ac); \
(a) = ROTATE_LEFT ((a), (s)); \
(a) += (b); \
}

/* MD5 initialization. Begins an MD5 operation, writing a new context. */

void
MD5Init (context)
MD5_CTX *context;
{

context->count[0] = context->count[1] = 0;

/* Load magic initialization constants.  */
context->state[0] = 0x67452301;
context->state[1] = 0xefcdab89;
context->state[2] = 0x98badcfe;
context->state[3] = 0x10325476;
}

/*
 * MD5 block update operation. Continues an MD5 message-digest
 * operation, processing another message block, and updating the
 * context.
 */

void
MD5Update (context, input, inputLen)
MD5_CTX *context;
const unsigned char *input;
unsigned int inputLen;
{
unsigned int i, index, partLen;

/* Compute number of bytes mod 64 */
index = (unsigned int)((context->count[0] >> 3) & 0x3F);

/* Update number of bits */
if ((context->count[0] += ((u_int32_t)inputLen << 3))
    < ((u_int32_t)inputLen << 3))
context->count[1]++;
context->count[1] += ((u_int32_t)inputLen >> 29);
```

```
partLen = 64 - index;

/* Transform as many times as possible. */
if (inputLen >= partLen) {
memcpy((void *)&context->buffer[index], (const void *)input,
    partLen);
MD5Transform (context->state, context->buffer);

for (i = partLen; i + 63 < inputLen; i += 64)
MD5Transform (context->state, &input[i]);

index = 0;
}
else
i = 0;

/* Buffer remaining input */
memcpy ((void *)&context->buffer[index], (const void *)&input[i],
    inputLen-i);
}

/*
 * MD5 padding. Adds padding followed by original length.
 */

void
MD5Pad (context)
MD5_CTX *context;
{
unsigned char bits[8];
unsigned int index, padLen;

/* Save number of bits */
Encode (bits, context->count, 8);

/* Pad out to 56 mod 64. */
index = (unsigned int)((context->count[0] >> 3) & 0x3f);
padLen = (index < 56) ? (56 - index) : (120 - index);
MD5Update (context, PADDING, padLen);

/* Append length (before padding) */
MD5Update (context, bits, 8);
}

/*
 * MD5 finalization. Ends an MD5 message-digest operation, writing the
```

```
 * the message digest and zeroizing the context.
 */

void
MD5Final (digest, context)
unsigned char digest[16];
MD5_CTX *context;
{
/* Do padding. */
MD5Pad (context);

/* Store state in digest */
Encode (digest, context->state, 16);

/* Zeroize sensitive information. */
memset ((void *)context, 0, sizeof (*context));
}

/* MD5 basic transformation. Transforms state based on block. */

static void
MD5Transform (state, block)
u_int32_t state[4];
const unsigned char block[64];
{
u_int32_t a = state[0], b = state[1], c = state[2], d = state[3], x[16];

Decode (x, block, 64);

/* Round 1 */
#define S11 7
#define S12 12
#define S13 17
#define S14 22
FF (a, b, c, d, x[ 0], S11, 0xd76aa478); /* 1 */
FF (d, a, b, c, x[ 1], S12, 0xe8c7b756); /* 2 */
FF (c, d, a, b, x[ 2], S13, 0x242070db); /* 3 */
FF (b, c, d, a, x[ 3], S14, 0xc1bdceee); /* 4 */
FF (a, b, c, d, x[ 4], S11, 0xf57c0faf); /* 5 */
FF (d, a, b, c, x[ 5], S12, 0x4787c62a); /* 6 */
FF (c, d, a, b, x[ 6], S13, 0xa8304613); /* 7 */
FF (b, c, d, a, x[ 7], S14, 0xfd469501); /* 8 */
FF (a, b, c, d, x[ 8], S11, 0x698098d8); /* 9 */
FF (d, a, b, c, x[ 9], S12, 0x8b44f7af); /* 10 */
FF (c, d, a, b, x[10], S13, 0xffff5bb1); /* 11 */
FF (b, c, d, a, x[11], S14, 0x895cd7be); /* 12 */
FF (a, b, c, d, x[12], S11, 0x6b901122); /* 13 */
```

```
FF (d, a, b, c, x[13], S12, 0xfd987193); /* 14 */
FF (c, d, a, b, x[14], S13, 0xa679438e); /* 15 */
FF (b, c, d, a, x[15], S14, 0x49b40821); /* 16 */

/* Round 2 */
#define S21 5
#define S22 9
#define S23 14
#define S24 20
GG (a, b, c, d, x[ 1], S21, 0xf61e2562); /* 17 */
GG (d, a, b, c, x[ 6], S22, 0xc040b340); /* 18 */
GG (c, d, a, b, x[11], S23, 0x265e5a51); /* 19 */
GG (b, c, d, a, x[ 0], S24, 0xe9b6c7aa); /* 20 */
GG (a, b, c, d, x[ 5], S21, 0xd62f105d); /* 21 */
GG (d, a, b, c, x[10], S22,  0x2441453); /* 22 */
GG (c, d, a, b, x[15], S23, 0xd8a1e681); /* 23 */
GG (b, c, d, a, x[ 4], S24, 0xe7d3fbc8); /* 24 */
GG (a, b, c, d, x[ 9], S21, 0x21e1cde6); /* 25 */
GG (d, a, b, c, x[14], S22, 0xc33707d6); /* 26 */
GG (c, d, a, b, x[ 3], S23, 0xf4d50d87); /* 27 */
GG (b, c, d, a, x[ 8], S24, 0x455a14ed); /* 28 */
GG (a, b, c, d, x[13], S21, 0xa9e3e905); /* 29 */
GG (d, a, b, c, x[ 2], S22, 0xfcefa3f8); /* 30 */
GG (c, d, a, b, x[ 7], S23, 0x676f02d9); /* 31 */
GG (b, c, d, a, x[12], S24, 0x8d2a4c8a); /* 32 */

/* Round 3 */
#define S31 4
#define S32 11
#define S33 16
#define S34 23
HH (a, b, c, d, x[ 5], S31, 0xfffa3942); /* 33 */
HH (d, a, b, c, x[ 8], S32, 0x8771f681); /* 34 */
HH (c, d, a, b, x[11], S33, 0x6d9d6122); /* 35 */
HH (b, c, d, a, x[14], S34, 0xfde5380c); /* 36 */
HH (a, b, c, d, x[ 1], S31, 0xa4beea44); /* 37 */
HH (d, a, b, c, x[ 4], S32, 0x4bdecfa9); /* 38 */
HH (c, d, a, b, x[ 7], S33, 0xf6bb4b60); /* 39 */
HH (b, c, d, a, x[10], S34, 0xbebfbc70); /* 40 */
HH (a, b, c, d, x[13], S31, 0x289b7ec6); /* 41 */
HH (d, a, b, c, x[ 0], S32, 0xeaa127fa); /* 42 */
HH (c, d, a, b, x[ 3], S33, 0xd4ef3085); /* 43 */
HH (b, c, d, a, x[ 6], S34,  0x4881d05); /* 44 */
HH (a, b, c, d, x[ 9], S31, 0xd9d4d039); /* 45 */
HH (d, a, b, c, x[12], S32, 0xe6db99e5); /* 46 */
HH (c, d, a, b, x[15], S33, 0x1fa27cf8); /* 47 */
HH (b, c, d, a, x[ 2], S34, 0xc4ac5665); /* 48 */
```

```
/* Round 4 */
#define S41 6
#define S42 10
#define S43 15
#define S44 21
II (a, b, c, d, x[ 0], S41, 0xf4292244); /* 49 */
II (d, a, b, c, x[ 7], S42, 0x432aff97); /* 50 */
II (c, d, a, b, x[14], S43, 0xab9423a7); /* 51 */
II (b, c, d, a, x[ 5], S44, 0xfc93a039); /* 52 */
II (a, b, c, d, x[12], S41, 0x655b59c3); /* 53 */
II (d, a, b, c, x[ 3], S42, 0x8f0ccc92); /* 54 */
II (c, d, a, b, x[10], S43, 0xffeff47d); /* 55 */
II (b, c, d, a, x[ 1], S44, 0x85845dd1); /* 56 */
II (a, b, c, d, x[ 8], S41, 0x6fa87e4f); /* 57 */
II (d, a, b, c, x[15], S42, 0xfe2ce6e0); /* 58 */
II (c, d, a, b, x[ 6], S43, 0xa3014314); /* 59 */
II (b, c, d, a, x[13], S44, 0x4e0811a1); /* 60 */
II (a, b, c, d, x[ 4], S41, 0xf7537e82); /* 61 */
II (d, a, b, c, x[11], S42, 0xbd3af235); /* 62 */
II (c, d, a, b, x[ 2], S43, 0x2ad7d2bb); /* 63 */
II (b, c, d, a, x[ 9], S44, 0xeb86d391); /* 64 */

state[0] += a;
state[1] += b;
state[2] += c;
state[3] += d;

/* Zeroize sensitive information. */
memset ((void *)x, 0, sizeof (x));
}
```

References

1. Binstock, Andrew and Rex, John. 1995, *Practical Algorithms for Programmers* (Reading, MA: Addison-Wesley), pp. 63-93.

2. Manber, Udi. 1989, *Introduction to Algorithms: A Creative Approach.* (Reading, MA: Addison-Wesley), pp. 79-80.

3. Neimann, Thomas. 1998, *Sorting and Searching Algorithms: A Cookbook* (http://www.geocities.com/SoHo/2167/book.html).

4. Schneier, Bruce. 1996, *Applied Cryptography* (New York, NY: John Wiley & sons), pp. 429-441.

0.2.8 Skip lists

In a paper entitled *Skip Lists: A Probabilistic Alternative to Balanced Trees* William Pugh at the University of Maryland proposed a linked-list like data structure with some additional features which improved traversal speed. Binary trees work well when the elements which they are to contain are inserted in a random order. As we have seen in the previous sections, BSTs perform very poorly when data is inserted in order. Red-Black trees, which do not suffer from degenerate performance manifested by BSTs, must constantly rotate and re-balance as data is inserted in order. A skip list is an ordered linked list in which every node contains a variable number of links to other nodes. The nth link of a given node points to subsequent nodes in the list skipping over some number of intermediary nodes. These skipped nodes are common in that they have fewer than n links associated with them.

Because most nodes have a variable number of links, a skip list is really a collection of linked lists of different levels. In order to quickly traverse the structure looking for some target key we search on the upper level list until either the target data is encountered or we find a node with a key smaller than the target which links to a subsequent node with a larget value than the target. In the second case we continue by repeating the same procedure beginning at the node with the smaller value than the target and continuing on the list one level down.

In the following quotation William Pugh describes the performance of his skip list data structure:

> Skip lists are a probabilistic alternative to balanced trees. Skip lists are balanced by consulting a random number generator. Although skip lists have bad worst-case performance, no input sequence consistently produces the worst-case performance (much like quicksort when the pivot element is chosen randomly). It is very unlikely a skip list data structure will be significantly unbalanced (e.g., for a dictionary of more than 250 elements, the chance that a search will take more than 3 times the expected time is less than one in a million). Skip lists have balance properties similar to that of search trees built by random insertions, yet do not require insertions to be random.
>
> Balancing a data structure probabilistically is easier than explicitly maintaining the balance. For many applications, skip lists are a more natural representation than trees, also leading to simpler algorithms. The simplicity of skip list algorithms makes them easier to implement and provides significant constant factor speed improvements over balanced tree and self-adjusting tree algorithms. Skip lists are also very space efficient. They can easily be configured to require an average of 1 1/3 pointers per element (or even less) and do not require balance or priority information to be stored with each node.

The above article appeared in the June 1990 issue of the Communications of the ACM. A Postscript format version is available for anonymous ftp from ftp.cs.umd.edu. The author, William Pugh, can be contacted at pugh@cs.umd.edu.

Source Code

This example skip list source code for C was written by William Pugh at the University of Maryland. His contact information if above.

```
/*

This file contains source code to implement a dictionary using
skip lists and a test driver to test the routines.

A couple of comments about this implementation:
  The routine randomLevel has been hard-coded to generate random
  levels using p=0.25. It can be easily changed.

  The insertion routine has been implemented so as to use the
  dirty hack described in the CACM paper: if a random level is
  generated that is more than the current maximum level, the
  current maximum level plus one is used instead.

  Levels start at zero and go up to MaxLevel (which is equal to
        (MaxNumberOfLevels-1).

The compile flag allowDuplicates determines whether or not duplicates
are allowed. If defined, duplicates are allowed and act in a FIFO manner.
If not defined, an insertion of a value already in the file updates the
previously existing binding.

BitsInRandom is defined to be the number of bits returned by a call to
random(). For most all machines with 32-bit integers, this is 31 bits
as currently set.

The routines defined in this file are:

  init: defines NIL and initializes the random bit source

  newList: returns a new, empty list

  freeList(l): deallocates the list l (along with any elements in l)

  randomLevel: Returns a random level

  insert(l,key,value): inserts the binding (key,value) into l. If
        allowDuplicates is undefined, returns true if key was newly
        inserted into the list, false if key already existed

  delete(l,key): deletes any binding of key from the l. Returns
        false if key was not defined.

  search(l,key,&value): Searches for key in l and returns true if found.
        If found, the value associated with key is stored in the
        location pointed to by &value
```

```
*/


#define false 0
#define true 1
typedef char boolean;
#define BitsInRandom 31

#define allowDuplicates

#define MaxNumberOfLevels 16
#define MaxLevel (MaxNumberOfLevels-1)
#define newNodeOfLevel(l) (node)malloc(sizeof(struct nodeStructure)+(l)*sizeof(node *))

typedef int keyType;
typedef int valueType;

#ifdef allowDuplicates
boolean delete(), search();
void insert();
#else
boolean insert(), delete(), search();
#endif

typedef struct nodeStructure *node;

typedef struct nodeStructure{
    keyType key;
    valueType value;
    node forward[1]; /* variable sized array of forward pointers */
    };

typedef struct listStructure{
   int level;     /* Maximum level of the list
                        (1 more than the number of levels in the list) */
   struct nodeStructure * header; /* pointer to header */
   } * list;

node NIL;

int randomsLeft;
int randomBits;

init() {
  NIL = newNodeOfLevel(0);
  NIL->key = 0x7fffffff;
  randomBits = random();
```

```
  randomsLeft = BitsInRandom/2;
};

list newList() {
  list l;
  int i;

  l = (list)malloc(sizeof(struct listStructure));
  l->level = 0;
  l->header = newNodeOfLevel(MaxNumberOfLevels);
  for(i=0;i<MaxNumberOfLevels;i++) l->header->forward[i] = NIL;
  return(l);
  };

freeList(l)
  list l;
  {
  register node p,q;
  p = l->header;
  do {
    q = p->forward[0];
    free(p);
    p = q; }
    while (p!=NIL);
  free(l);
  };

int randomLevel()
  {register int level = 0;
   register int b;
   do {
    b = randomBits&3;
    if (!b) level++;
    randomBits>>=2;
    if (--randomsLeft == 0) {
        randomBits = random();
        randomsLeft = BitsInRandom/2;
        };
    } while (!b);
    return(level>MaxLevel ? MaxLevel : level);
    };

#ifdef allowDuplicates
void insert(l,key,value)
#else
boolean insert(l,key,value)
#endif
```

```
register list l;
register keyType key;
register valueType value;
  {
  register int k;
  node update[MaxNumberOfLevels];
  register node p,q;

  p = l->header;
  k = l->level;
  do {
        while (q = p->forward[k], q->key < key) p = q;
        update[k] = p;
        } while(--k>=0);

#ifndef allowDuplicates
  if (q->key == key) {
        q->value = value;
        return(false);
        };
#endif

    k = randomLevel();
    if (k>l->level) {
        k = ++l->level;
        update[k] = l->header;
        };
    q = newNodeOfLevel(k);
    q->key = key;
    q->value = value;
    do {
        p = update[k];
        q->forward[k] = p->forward[k];
        p->forward[k] = q;
        } while(--k>=0);
#ifndef allowDuplicates
    return(true);
#endif
    }

boolean delete(l,key)
register list l;
register keyType key;
  {
  register int k,m;
  node update[MaxNumberOfLevels];
```

```
  register node p,q;

  p = l->header;
  k = m = l->level;
  do {
        while (q = p->forward[k], q->key < key) p = q;
        update[k] = p;
        } while(--k>=0);

  if (q->key == key) {
        for(k=0; k<=m && (p=update[k])->forward[k] == q; k++)
          p->forward[k] = q->forward[k];
        free(q);
        while( l->header->forward[m] == NIL && m > 0 )
             m--;
        l->level = m;
        return(true);
        }
    else return(false);
    }

boolean search(l,key,valuePointer)
register list l;
register keyType key;
valueType * valuePointer;
  {
  register int k;
  register node p,q;
  p = l->header;
  k = l->level;
  do while (q = p->forward[k], q->key < key) p = q;
      while (--k>=0);
  if (q->key != key) return(false);
  *valuePointer = q->value;
  return(true);
  };

#define sampleSize 65536
main() {
  list l;
  register int i,k;
  keyType keys[sampleSize];
  valueType v;

  init();

  l= newList();
```

```
  for(k=0;k<sampleSize;k++) {
        keys[k]=random();
        insert(l,keys[k],keys[k]);
        };


  for(i=0;i<4;i++) {
        for(k=0;k<sampleSize;k++) {
            if (!search(l,keys[k],&v)) printf("error in search #%d,#%d\n",i,k);
            if (v != keys[k]) printf("search returned wrong value\n");
            };
        for(k=0;k<sampleSize;k++) {
            if (! delete(l,keys[k])) printf("error in delete\n");
            keys[k] = random();
            insert(l,keys[k],keys[k]);
            };
        };

  freeList(l);

  };
\end{verbatim}
\nopagebreak\hrule
\addvspace{\medskipamount}
```

References

1. ftp://ftp.cs.umd.edu/pub/skipLists/

0.2.9 Tries

A **trie** is a special kind of tree data structure in which items are stored and accessed based upon their key value alone, not based upon their key value in relation to others in the tree. In contrast, recall that a **binary search tree**uses a greater-than or less-than relationship between keys in the tree to determine where a new insertion would be placed. In tries, the new insertion's place is predetermined based on its key value. For this reason a trie is somewhat of a cross between a tree and a **hash table**.

Tries can only be used when the range of valid keys to be stored is known up-front. To store a new item in a trie, that item's key value is somehow broken down into components. If, for example, the item to be stored is a string, a logical way to break its value down is by letter. For a number, perhaps a good way to break down the key value is by digits in its binary representation. Every internal node in a trie can have as many child nodes as the number of items in the alphabet you are storing. That is, if you

are traversing on English letters each node can have up to 26 children. If you are traversing on binary digits, each node can have two children, 0 or 1.

Imagine we have a trie in which we are storing strings. We wish to store the new item “dog” in the trie. From the root node we follow the “d” edge and arrive at a child node. The only data stored in leaves under this child node are words that begin with the letter “d.”

Next, we traverse along the “o” link from the “d” node. We reach yet another internal node under which only words which start with the prefix “do” are stored. Finally suppose we find that there is no “g” link off the “do” node. We create one and store “dog” at this new leaf node.

Likewise, if we want to store the number six (6) in a numeric trie, we might convert six to it’s binary representation (110). From the first node we follow the “1” edge. Next we traverse down the “1” edge again. And finally we store the value six in a leaf node that is lies along the “110” edges from the root node.

Any path from the root to a leaf in a trie corresponds to the value of the item stored at the leaf node reached.

Huffman trees, the data structures behind Huffman data compression algorithms, are a popular use of tries. **Huffman coding**is discussed in the data compression section of this document.

Source Code

This code, in C++, implements an alphabetic trie class.

```
struct trie_node {

    // how many nodes are beneath this one?
    int count;

    // array of pointers to children
    trie_node *child_list[ALPH_SIZE];

    // initializer...
    trie_node();
};

trie_node::trie_node() {

    // there is nothing below us...
    count = 0;

    // all child pointers start out pointing nowhere...
    for(int index = 0; index < ALPHABET_SIZE; index++)
        child_list[index] = NULL;
```

```
}



class Trie {

public:

    // methods

    Trie(int level, int chars);        // create it
    ~Trie();                           // destroy trie nodes
    int node_count;                    // the number of nodes in whole trie
    void AddToTrie(const string \&s);   // add string to trie

private:

    // methods

    void recursive_delete (trie_node * t);     // recursive delete helper function



    // data

    int node_count;                      // number of nodes in the trie
    trie_node * Root;                  // the root of the trie
};




Trie::Trie() {
    node_count = 0;
    Root(new trie_node);
}


Trie::~Trie() {

    // delete whole trie
    recursive_delete(myRoot);
}


void Trie::recursive_delete(trie_node * t) {
    int index;
```

```
    if (t != NULL) {

        // delete all children
        for(index = 0; index < ALPHABET_SIZE; index++)
           recursive_delete(t->child_list[index]);

// delete self
        delete t;
    }
}




int Trie::node_count() {

    return (node_count);

}



void Trie::AddToTrie(const string \&s) {
    int lcv, index;

    trie_node *t = Root;

    // there is one more string stored somewhere under the root...
    t->count++;


    // loop over the length of the string to add and traverse the trie...
    for(lcv=0; lcv < s.length(); lcv++) {

        index = s[lcv]; // the character in s we are processing...

// is there a child node for this character?
        if (t->child_list[index] == NULL) {

    // if not, make one!
            t->child_list[index] = new trie_node;
            node_count++;
        }

        // there is another string under this node...
        t->child_list[index]->count++;
```

```
        // move to it this node... and loop
        t = t->child_list[index];
    }
}
```

0.2.10 Quadtrees and Octrees

A quadtree is a hierarchical data structure often used for image representation. Quadtrees encode two-dimensional images by placing subsections of the image into a tree structure much like a **binary tree**. Unlike a binary tree where every tree node has up to two children, in a quadtree every node can have up to four children. Each node stores a particular piece of the overall image.

Quadtrees operate by means of **recursively decomposing space**. While several specialized types of quadtrees exist, the most common type are known as **region quadtrees**. In such trees the image to be encoded and stored is partitioned into four quadrants. Each quadrent is then represented by a node in the tree. The root of the quadtree represents the entire stored image while each of it's four children represent a different fourth of the image. This process continues; each of the quadrants represented by the children of the root is again partitioned into four sub-quadrants and represented by the root's grandchildren nodes. This process continues until a region being stored is sufficiently simple that it can be wholly encoded in one node. Sometimes this means that we continue breaking the image down until every individual pixel is stored at some leaf node. Other times a region larger than a pixel is sufficiently homogeneous that it can be represented at a leaf node and requires no further decomposition. For example, if an entire quadrent is one color it does not make sense to continue to decompose it.

The regions represented by nodes in a quadtree are sometimes named after map directions (northwest, northeast, southwest, southeast).

An octree is essentially the same thing as a quadtree however each node in an octree has eight children instead of four. This makes octrees good candidates for representing three-dimensional objects in memory because instead of breaking the image into four quadrants the octree breaks the image into eight blocks. These blocks are then recursively decomposed into eight sub-blocks. This process, as in the quadtree, continues until a blocks is sufficiently homogenous that it can be represented by one node.

An interesting property of both quadtrees and octrees is that they support a form of data compression. By never traversing below a certain level of the tree structure it is possible to filter out the "details" of the image opting, instead, for a less-precise but smaller version of the data. Such a compression could be accomplished by means of a breadth-first traversal of the tree data structure.

Octree Source Code

Many thanks to Lynn Jones for donating octree code to this collection. The following implementation was written in C++ by Lynn Jones in a research project at the University of South Carolina.

```
// Octree Program -- dataarray.h
// Lynn Jones, Virginia Tech, lwjones@vt.edu

#ifndef D_ARRAY
#define D_ARRAY

#include <stdio.h>
#include "constants.h"
#include "octreeNode.h"

class DataArray {
public:
  uchar data[xVal][yVal][zVal];

  DataArray(){}
  ~DataArray(){}

  void ReadData() {
    for (int i=0; i<zVal; i++)
      for (int j=0; j<yVal; j++)
for (int k=0; k<xVal; k++) {
  cin >> data[k][j][i];
}
  }

  void WriteData() {
    for (int i=0; i<xVal; i++)
      for (int j=0; j<yVal; j++)
for (int k=0; k<zVal; k++)
  cout << (short)data[i][j][k];
  }

  OctreeNode *BuildTree
    (int fromX =0, int fromY =0, int fromZ =0, int length = xVal)
      /*
 Octree subdivides in this order:

 1---5
       /|  /|
      0---4 |
      | 3-|-7
      |/  |/
      2---6

      */
  {
    OctreeNode *parent;
```

```
    if (length == 1) {
      parent = new OctreeNode(LEAF);
      parent->value = data[fromX][fromY][fromZ];
      return parent;
    }
    //else
    parent = new OctreeNode(INTERNAL);
    parent->children[0] = BuildTree(fromX, fromY, fromZ, length/2);
    parent->children[1] = BuildTree(fromX, fromY, fromZ+length/2, length/2);
    parent->children[2] = BuildTree(fromX, fromY+length/2,fromZ, length/2);
    parent->children[3] = BuildTree(fromX, fromY+length/2, fromZ+length/2, length/2);
    parent->children[4] = BuildTree(fromX+length/2, fromY, fromZ, length/2);
    parent->children[5] = BuildTree(fromX+length/2, fromY, fromZ+length/2, length/2);
    parent->children[6] = BuildTree(fromX+length/2, fromY+length/2, fromZ, length/2);
    parent->children[7] = BuildTree(fromX+length/2, fromY+length/2, fromZ+length/2, length/2);
    parent->value = parent->AverageChildren();
    return parent;
  }
};
#endif
```

```
// Octtree Program -- octreenode.h
// Lynn Jones, Virginia Tech, lwjones@vt.edu

#ifndef OCTNODE
```

```
// Octtree Program -- octreenode.h
// Lynn Jones, Virginia Tech, lwjones@vt.edu

#ifndef OCTNODE
#define OCTNODE

#include <iostream.h>
#include "constants.h"


class OctreeNode {

public:
  short index;
  uchar value;
  OctreeNode **children;
```

```
  OctreeNode(int nodeType) {
    index = 0;
    value = (uchar)0;
    if (nodeType == LEAF)
      children = NULL;
    else {
      children = new OctreeNode*[8];
      for (int i=0; i<8; i++)
children[i] = NULL;
    }
  }

  ~OctreeNode() {
    if (children)
      for (int i=0; i<8; i++)
delete children[i];
  }

  void Print() {
    cout << endl << "index = " << index << "\tvalue = " << (unsigned int)value;
  }

  void PruneTree() {
    ushort prevValue, tempValue=0;
    int homogeneous = 1;


    // EXACT MATCH ****************
    for (int i=0; i<8; i++) {
      prevValue = (ushort)children[i]->value;
      tempValue += prevValue;
      //if ((tempValue / (i+1)) != prevValue)
      //homogeneous = 0;
    }
    if ((tempValue / 8) != prevValue)
      homogeneous = 0;


    /*RANGE COLLAPSING ************

    ??????? How to keep range from propagating up the tree ?????????
    this var not declared:firstVal = (ushort)children[0]->value;
    for (int i=0; i<8; i++) {
      prevValue = (ushort)children[i]->value;
      tempValue += prevValue;
      if ((tempValue / (i+1)) != prevValue)
       homogeneous = 0;
```

```
    }
    */

    value = (uchar)(tempValue/8);
    if (homogeneous)
      for (int i=0; i<8; i++){
delete children[i];
children[i]=NULL;
      }
  }

  uchar AverageChildren() {
    if (children) {
      ushort temp = 0;
      for (int i=0; i<8; i++)
temp += (ushort)children[i];
      return (uchar)(temp /= 8);
    }
    //else
    return (uchar)0;
  }

};
#endif
```

```
// Octree Program -- constants.h
// Lynn Jones, Virginia Tech, lwjones@vt.edu

#ifndef CONSTS
#define CONSTS

#define xVal         64
#define yVal         64
#define zVal         64
#define SPHERESIZE   16
#define SHAPEVALUE    1
#define VOIDVALUE     0
#define LEAF        111
#define INTERNAL    222
typedef unsigned char  uchar;
typedef unsigned short ushort;

#endif
```

```
// PROGRAM to build an Octree and output the nodes to file

// PROGRAMMER Lynn W. Jones, Virginia Tech : lwjones@vt.edu
// PROJECT The Dynamic Brain Project, Univ. of South Carolina
// summer 1996

#include <iostream.h>
#include <stdlib.h>
#include <string.h>

#include "octreeNode.h"
#include "dataArray.h"
#include "tree.h"

int main() {
  DataArray myData;
  myData.ReadData();
  Tree_8 dataTree(myData.BuildTree());
  dataTree.DepthTraversal(dataTree.root, "PruneTree");
  dataTree.DepthTraversal(dataTree.root, "PruneTree");
  //call this function as many times as has any effect
  //dataTree.DepthTraversal(dataTree.root, "PruneTree");
  //dataTree.DepthTraversal(dataTree.root, "PruneTree");
  dataTree.DepthTraversal(dataTree.root, "Print");
  return 0;
}
```

0.3 Sorting Algorithms

Several common sorting algorithms are discussed in the following section along with a careful examination of the strengths and weaknesses of each.

Before any discussion of sorting, however, it is useful to define some terms which will be used in describing the process or sorting. A **sorting algorithm** is one orders a data set based on some **key value**. A particular data set comprised of n items can have anywhere between one and n distinct (or "unique") keys. For example, a set in which all the items are identical would only have one key and is said to be a **homogeneous set**. A set with a few identical items might have $(n - x)$ keys where x is the number of identical items in said set.

All sorting algorithms can be divided into one of two categories: those which are **comparison based** and those which are not. A comparison based algorithm is one that orders the data set by weighing the key value of one element against that of one or many other elements. Conversely a **non comparison based** sort puts the target data set into order without consideration of two or more data items.

The efficiency of each presented algorithm is discussed in detail. When considering the efficiency of a given algorithm it is useful to examine its performance in several cases. These include sorting data sets of various sizes, data sets already in sorted order, data sets in reverse sorted order, and data sets in random order. Sometimes the number of comparisons performed by a particular algorithm does not matter but the number of swaps must be minimized because swapping records is **expensive**. Other we need not worry about space but want an algorithm that sorts as quickly as possible. It is *vital* to know something about the data which you are sorting in order to choose the algorithm that best matches your needs.

In the following sections we explore the following algorithms in detail:

0.3.1 The Quicksort

Over the years computer scientists have come up with many different algorithms for sorting data. C.A.R. Hoare's Quicksort, however, is generally regarded as the most efficient and fastest sorting algorithm in the average case. As we will see, though, a worst case data set causes the Quicksort to perform poorly.

A Quicksort operates by selecting a value called the **pivot point** and then arranging the data being sorted such that all data items with a key value less than the that of the pivot point appear at the beginning of the data structure and all data items greater than or equal to the pivot value are moved towards the end of the data structure.

Thus, the data set to be sorted is **partitioned** into two pieces; the k items less in value than the pivot item's value, and the $(n - k)$ items greater than or equal to the pivot value. It is essential to note that neither of these two partitions are sorted as they are built; all we know after the partitioning operation is that every item to the "left" of the pivot is less in value than it, and every item "right" of the pivot point is greater than or equal to it.

The Quicksort continues, at this point, to process the two partitions discussed above. In each sub-partition a new pivot value is selected which allows yet another sub-division of the data set. This process of selecting a new pivot value and then sub-dividing a range of data into two more partitions continues again and again until the size of a resulting partition becomes small enough that it can be easily explicitly sorted. This point usually occurs when there are two or fewer items remaining in a partition because such ranges can be easily put into order with a very little effort. As we will see later, there are other alternatives.

Picking a Pivot Value

As you might imagine, selecting a good pivot value is crucial to the success and performance of this sort. In the ideal case we want to pick the statistical median key in a partition as the pivot value and, thus, split the partition into equal halves.

The simplist way to pick a pivot value is to use the value of first data item. This method has the advantage of being a very fast but, unfortunately, operates on a sometimes faulty supposition. If the data being sorted is in near-sorted order then the first item in a given partition is very likely to have the lowest value of all items in said partition. This leads to unbalanced partitioning of the data set. To avoid infinite recursion usually Quicksort is implemented to partition into a set containing the pivot element and a set containing the $n - 1$ others. However, repeated unbalanced partitioning causes Quicksort to perform very poorly.

In order to eliminate this risk, often the higher of the two first distinct key values in a partition is selected to be the pivot. Other pivot selection schemes add the first and last values in the partition and divide by two. Still others take the middle-of-three approach. While methods such as these must examine slightly more data than one which blindly chooses the first element, normally they speed up the overall algorithm by choosing a value more likely to be near the median of the partition.

Below is an example pivot selecting routine written in C. It uses the greater of the first two distinct values in a partition as the pivot point and returns NONE if the partition does not need to be sorted any further.

```
/* NONE must be a value that cannot occur as a key */
#define NONE -1

/* these are arbitrary */
typedef key int;
typedef data struct {
  key thekey;
  char *therest;
};

/*
 *  Return the pivot value or NONE if the partition does
 *  not need to be sorted any further.
 *
 */

key selectpivot(data *array, int left, int right) {

  key first = value(array[left]);         /* the first key */
  int lcv;                                 /* loop control */

  for (lcv = left + 1; lcv <= right; lcv++) {
    if (value(array[lcv]) > first) return (value(lcv));
    else if (value(array[lcv]) < first) return first;
  }

  /* if we get here the partition is homogeneous */
  return (NONE);
}
```

```
key value(data *item) {
  return (item->thekey);
}
```

Above, the routine selects the value of a pivot point and returns it, or, if all the elements in the partition are equal in value, returns `NONE` to indicate that further sorting of this partition is not necessary. `NONE` must be a value that will not appear in the array. If you cannot predict which values will appear in your data set, it would be better to write the above routine to return the index of the pivot value rather than the value itself. Then it could return an invalid index number to specify a homogeneous partition.

Partitioning

Now, here is code to do the actual work of dividing an input range into two partitions based on a pivot point. Again, it is written in C:

```
int partition (data *array, int left, int right, int pivotval) {
  do {
    swap (&array[left], &array[right]);
    while (value(array[left]) < pivotval) left++;
    while (value(array[right]) >= pivotval) right++;
  } while (right >= left);

  /* this will be the value of the first element in the right part */
  return (left);
}
```

The above routine cleverly moves inward from each end of the input range swapping data values that are on the wrong side of the pivot value until the two inward-moving indices meet in the middle. The initial swap, above, is not necessary; it is included only to make the do-loop more simple. In the worst case, $n-1$ comparisons and $\frac{n}{2}$ swaps are necessary to partition the data set.

Analysis

The performance of the whole Quicksort algorithm depends on how well `selectpivot` does at picking a good pivot point. The worst case for the given `selectpivot` procedure is when the data is already sorted. The higher of the first two elements in a data partition will simply divide the range into an one-element left side and an $(x-1)$ element right side. This causes the Quicksort to run in quadratic time, n^2 to be precise.

Another limitation of the Quicksort is that it tends to be an excellent choice for large arrays but performs badly with very small ones.

The best and average case running efficiency of a Quicksort is $\Theta(nlog_2n)$.

In order to Quicksort one element takes no comparisons. That is:

$$T(1) = 0$$

Now, in order to Quicksort n elements we have to select a pivot point, partition the n elements, and recurse on the two partitions. Assume the i element is chosen to be the pivot point (where $i \in n$). In order to partition the n elements will take, at most, $(n-1)$ comparisons at which point the quicksort will recurse on both of the two partitions. One partition will consist of the first $i-1$ elements and the other will be the contain $n-i$ elements (assuming the pivot value itself goes with the right partition). Thus, to sort n items we need, at most, $(n-1)$ comparisons in the partitioning phase plus however long it takes to sort each resulting partition:

$$T(n) = (n-1) + T(i-1) + T(n-i)$$

However, the above recurrence relation cannot be simplified in the stated form; to continue more information about the selection of a pivot point is needed. Assume that any value of the n range is equally likely to become the pivot value. The above expression can be rewritten in the form below. Note that the terms $T(i-1)$ and $T(n-i)$ have been summed for all n possible values of i and then divided by n:

$$T(n) = (n-1) + \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i))$$

The summation in the above expression can be distributed to both $T(x)$ terms giving:

$$T(n) = (n-1) + \frac{1}{n}\sum_{i=1}^{n}T(i-1) + \frac{1}{n}\sum_{i=1}^{n}T(n-i)$$

Or...

$$T(n) = (n-1) + \frac{2}{n}\sum_{i=0}^{n-1}T(i)$$

This is a full-history recurrence relation and is solved by subtracting $T(n)$ from $T(n+1)$. First, calculate $T(n+1)$ by substituting an $(n+1)$ for every n in the above equation:

$$T(n+1) = ((n+1)-1) + \frac{2}{n+1}\sum_{i=1}^{(n+1)-1}T(i)$$

These summations cannot be subtracted because of each ones the leading fraction. To proceed, multiply each formula by the denominator of its fraction and get:

$$(n+1)T(n+1) = n(n+1) + 2\sum_{i=1}^{n}T(i)$$
$$(n)T(n) = n(n-1) + 2\sum_{i=1}^{n-1}T(i)$$

Now subtract the latter from the former:

$$(n+1)T(n+1) - (n)T(n) = (n+1)n - n(n-1) + 2T(n) = 2n + 2T(n)$$

This implies that:

$$T(n+1) = \frac{n+2}{n+1}T(n) + \frac{2n}{n+1}$$

This relation is still tricky to solve. To proceed substitute a value of 2 for the fraction $\frac{2n}{n+1}$ – a pretty close approximation. Also reverse the terms being added.

$$T(n+1) = 2 + \frac{n+2}{n+1}T(n)$$

So expanding this relation:

$$T(n) = 2 + \frac{n+1}{n}(2 + \frac{n}{n-1} + (2 + \frac{n-1}{n-2}(...\frac{4}{3})...))$$

That is:

$$T(n) = 2(1 + \frac{n+1}{n} + \frac{n+1}{n-1} + ...\frac{n+1}{3}$$

Or:

$$T(n) = 2(n+1)(H(n+1) - 1.5)$$

Where H above is the harmonic series $(1 + 1/2 + 1/3 + 1/4... + 1/n)$. The harmonic series of n can be approximated as:

$$H(n) = ln(n) + O(1/n) + \gamma$$

Where $\gamma = 0.577...$ is **Euler's constant**. The approximate solution to the average case complexity of the Quicksort is:

$$T(n) \leq 2(n+1)(ln(n) + \gamma - 1.5) + O(1)$$

This is an $O(nlog_2n)$ solution.

Improvement Strategies

Some improvements can be made to the Quicksort by bolstering its weaknesses. The Quicksort does not perform well for small data sets. While your first instinct may be to ignore this limitation because you are "always going to be sorting large arrays" you should be remember that, as the algorithm divides your large array into smaller partitions, eventually it will reach a point that is is sorting a small array. The performance of the Quicksort can be enhanced if, for small partitions, it does not call itself recursively. Rather, calling another sort algorithm to handle the small data set can substantially decrease running time.

Another way to accomplish this same optimization is to just stop sorting when your partitions reach a certain (small) size. Your final product will be a data set that is in "almost" sorted order. A few data items will be out of place here and there but never by much. Such an "almost sorted" array can then

be fed through an Insertion Sort and put into final order. Because the Insertion Sort runs in near linear time for "almost sorted" data, this last step will normally prove to be much faster than continuing to sort every little partition recursively with a Quicksort.

Yet another Quicksort improvement strategy which pertains especially to recursive implementations of the algorithm is to always process the smallest partition first. This results in more efficient use of the call stack and overall faster execution.

Source Code

Here is a full implementation of an unoptimized recursive Quicksort in C.

```
/* ------------------------------------------------------------------------ */
/*   get_pivot - return the index of the selected pivot value
 */

int get_pivot (int low, int hi) {

  /* safety net, this should not happen */
  if (low == hi) return(data[low]);

  /* return the greater of the first two items in the range  */
  return( (data[low] > data[low+1]) ? low : (low+1) );
}

/* ------------------------------------------------------------------------ */
/*   swap - given two pointers to integers, swap their contents
 */

void swap (int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
  num_swaps++;
}

/* ------------------------------------------------------------------------ */
/*   q_sort - Quicksort a data range
 */

void q_sort (int low, int hi) {
  int pivot_index;                /* index in the data set of the pivot */
  int pivot_value;                /* the value of the pivot element     */
  int left, right;

  /* select the pivot element and remember its value */
```

```
  pivot_index = get_pivot(low, hi);
  pivot_value = data[pivot_index];

  /* do the partitioning */
  left = low; right = hi;
  do {

    /* move left to the right bypassing elements already on the correct side */
    while ((left <= hi) && (data[left] < pivot_value)) {
      num_comps++;
      left++;
    }
    num_comps++;

    /* move right to the left bypassing elements already on the correct side */
    while ((right >= low) && (pivot_value < data[right])) {
      num_comps++;
      right--;
    }
    num_comps++;

    /*
     *  if the pointers are in the correct order then they are pointing to two
     *  items that are on the wrong side of the pivot value, swap them...
     */
    if (left <= right) {
      swap(&data[left], &data[right]);
      left++;
      right--;
    }

  } while (left <= right);

  /* now recurse on both partitions as long as they are large enough */
  if (low < right) q_sort(low, right);
  if (left < hi) q_sort(left, hi);
}
```

References

1. Manber, Udi. 1989, *Introduction to Algorithms: A Creative Approach.* (Reading, MA: Addison-Wesley), pp. 52-53, 131-137.

2. Press, William H. et al, *Numerical Recipes in C* (Cambridge, UK: Cambridge University Press), pp. 333-335.

0.3.2 The Mergesort

This algorithm is very easy to understand and exhibits good runtime speed for most data sets. Like the previously analyzed Quicksort, the Mergesort reiterates on the dataset, dividing it into ever smaller portions and sorting each subdivision. The Mergesort chops partitions in half by using the formula:

$$m = \frac{l + r}{2}$$

Next, the Mergesort reiterates on both of the newly formed partitions, chopping them in half and continuing the process. This subdivision stops when the partition size reaches one item.

At this point the algorithm has created many one-item data sets. Any one-item set is in "sorted" order by definition. The next step in the process is to merge these data sets together thus creating ever larger sorted sets.

To combine two sorted lists the Mergesort compares successive pairs of elements, one from each list. If the one from list "A" has a smaller key, it it chosen to be appended to the aggregate list. Of course the opposite applies if the element we are examining from list "B" proves to have a smaller key value. In the event that either input list is exhausted, all the remaining elements on the other list are appended to the aggregate list. While this merging algorithm will work with two sorted input lists of unequal size, in order to minimize the number of calls to the routine, Mergesort tries to combine lists with the same (or close to the same) number of elements. This merging process takes at least $\frac{n}{2}$ comparisons and no more than $(n - 1)$ comparisons.

Mergesort repeats the process of combining sorted sublists into ever larger aggregate lists until all have been successfully integrated back into a single, sorted list.

Analysis

One drawback this sorting method is that in order to combine two lists of n elements we either need storage for $4n$ data items in memory, or a clever way of merging lists. Using the most straightforward algorithm for merging two sorted lists we will need n storage cells for each of the two lists to be merged, and then another $2n$ for the aggregate list which we are building from members of the two sorted input lists.

There are some more space-efficient methods of building the aggregate list. One attack on this space problem is to use **linked-lists** to represent the data items and then build the aggregate list by changing pointer values. This way no space for the aggregate list need be allocated as it will be built of the memory already used by members of each sorted input list. There are also other ways of building an aggregate list in the "unused" (or "already used") space of the two input lists but this saved space sometimes comes at the cost of a slight penalty in runtime and an increase in algorithm complexity.

Unlike the Quicksort, the Mergesort does not concern itself with the ordering of its sublists but rather accomplishes the goal of sorting the entire data set by sorting very small subsets and then merging these back together again in order. The Mergesort's performance, therefore, is not greatly affected by

the arrangement of values in the input set. Thus, the worst case performance of the Mergesort does not deteriorate for certain "bad" data orders.

Mergesorts are often used to sort linked lists both because the algorithm does not require random access to the data set on which it is operating and because, due to the nature of a linked list, the need to allocate extra space for aggregate lists is eliminated (as discussed above). The Mergesort can also be modified slightly to produce an effective external (i.e. on disk) sort.

Here is a more formal analysis of the algorithm. In order to Mergesort two items we need one comparison. Thus:

$$T(2) = 1$$

In order to Mergesort n items we will need to split the n items in to two sets of $\frac{n}{2}$ and then combine them. In order to merge two data sets of $\frac{n}{2}$ items we will need to make, at most, $(n-1)$ comparisons. Thus:

$$T(n) = 2T(\frac{n}{2}) + n - 1$$

If n is restricted to be 2^x then $T(n)$ can be expressed as:

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + n - 1 \\ &= 2(2T(\tfrac{n}{4}) + \tfrac{n}{2} - 1) + n - 1 \\ &= 4T(\tfrac{n}{4}) + 2n - 2 - 1 \\ &= 4(2T(\tfrac{n}{8}) + \tfrac{n}{4} - 1) + 2n - 2 - 1 \\ &= 8T(\tfrac{n}{8} + 3n - 4 - 2 - 1 \\ &\vdots \end{aligned}$$

Therefore,

$$\begin{aligned} T(n) &= 2^i T(\tfrac{n}{2^i}) + in - \textstyle\sum_{j=0}^{i-1} 2^j \\ &= 2^i T(\tfrac{n}{2^i}) + in - 2^i + 1 \end{aligned}$$

Now, since $i = log_2 n$:

$$\begin{aligned} T(n) &= 2^{log_2 n} T(\tfrac{n}{2^{log_2 n}} + nlog_2 n - 2^{log_2 n} + 1 \\ &= nlog_2 n - n + 1 \end{aligned}$$

Improvements

One way to improve of the Mergesort algorithm is to reverse the order of each "right" list during its creation. This allows the merge procedure to work inward from the opposite ends of its two input lists which, in turn, allows the end of each input list to act as a guard for the other inside the merge routine and eliminates the need for special tests to see if a particular input list has been exhausted. This is discussed more fully in Sedgewick's *Algorithms in C*.

Source Code

The Mergesort, implemented in C, follows:

```
void mergesort (int *array, int left, int right) {
  if (left < right) {
    merge_sort (array, left, (left+right)/2);
    merge_sort (array, (left+right)/2 + 1, right);
    merge (array, left, (left+right)/2, (left+right)/2+1, right);
  }
}


void merge (int *A, int first1, int last1, int first2, int last2) {
  int temp[MAX_ARRAY];
  int index, index1, index2;
  int num;

  index = 0;
  index1 = first1;
  index2 = first2;

  num = last1 - first1 + last2 - first2 + 2;

  /*
   *  Merge the two lists back together until one list runs out of
   *  items...
   *
   */

  while ((index1 <= last1) && (index2 <= last2)) {
    if (A[index1] < A[index2]) temp[index++] = A[index1++];
    else temp[index++] = A[index2++];
  }

  /*
   *  At which point fill in the merged list with the "rest" of the
   *  non exhausted list.
   *
   */

  if (index1 > last1)
    move (A, index2, last2, temp, index);
  else
    move (A, index1, last1, temp, index);
```

```
  /* finally move our merged array in temp space to the real array */
  move(temp, 0, num-1, A, first1);
}


void move (int *list1, int first1, int last1, int *list2, int first2) {
  while (first1 <= last1)
    list2[first2++] = list1[first1++];
}
```

References

1. Manber, Udi. 1989, *Introduction to Algorithms: A Creative Approach.* (Reading, MA: Addison-Wesley), pp. 130-131
2. Parberry, Ian. 1997, *Analysis of Mergesort* (htt p://mycroft.csci.unt.edu/ian/)
3. Sedgewick, Robert 1998, *Algorithms in C, 3rd ed* (Reading, MA: Addison-Wesley), pp. 335-359

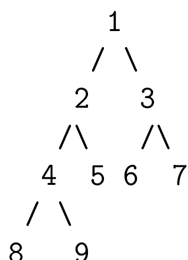
0.3.3 Heapsort

A Heapsort is based on a **heap data structure**. A heap is a **complete binary tree**. This means that every successive **level** of the tree must fill up from left to right. Further, an entire level must be full before any **nodes** at that level can have **children nodes**. In a heap, the **parent** nodes always have a

greater (or lower) key value than their children nodes. A heap in which the parents are always greater than their children is called a **max-heap** whereas the opposite is called a **min-heap**.

Building a Max-Heap

The first step in a Heapsort algorithm is to build a max-heap! Any array can be "heap shaped" if we look at it the right way; place the root of the heap at element one (or zero) in the array. Further, let any node's two children in the heap reside at indexes $(2n)$ and $(2n+1)$ in the array. Our heap-shaped array, then, would look something like this:



Note that each level of the heap is **full**. If we were to add an element to this heap we would add at the end of the array, at position ten. Position ten, by the above formula, is the left child of node five.

However, in a max-heap every element has to have a greater key value than either of its children. Clearly, then, the root node, or position one in our array, must be the data item with the greatest key value in the set to be sorted.

We can build this heap by putting our data items into the array in any order and then "heapifying" the array. Examine the first **non-leaf** node in the heap-array and compare its key value against that of its greatest child. If it is greater than its greater children, proceed to the next non-leaf node and repeat this process. However, if it is not greater than its greater children then swap these elements. Before continuing to the next non-leaf node and repeating the process the algorithm must be certain that the newly demoted value is in the correct spot. Thus the process must recurse on this demoted value before it can continue with the next leaf on its way to the root. By moving up the whole heap-array in this manner all nodes will end up being greater than their children.

Analysis

Imagine the distance from a given node down to the most distant leaf node below it is represented by d. In the process of "heapification," the number of comparisons needed to process a given node is, at most, $2d$. This is because, in the worst case, when the value at said node is compared to it's two children it will have to be demoted. To determine that the node must be demoted two comparisons are needed: one to compute the greatest child and one to compare the parent node value with the greatest child. Once the values have been swapped it becomes necessary to continue this two-comparison cost process until we reach a leaf node, in the worst case.

Thus the total complexity of the entire "heapify" process is, at most, two times the sum of the heights of all nodes in the heap. The goal now is to evaluate this sum.

Consider complete trees (where all nodes have either two children or none). Let $H(i)$ denote the sum of the heights of all nodes a complete tree of height i. A tree of height one has one node with height one. A tree of height i consists of two trees of height $(i-1)$ plus one root node. Hence,

$$H(i) = 2H(i-1) + i$$

Of course $H(1) = 1$.

The closed form solution of this recurrence relation is:

$$H(i) = 2^{i+1} - (i+2)$$

Since a complete binary tree has more nodes than an incomplete one, the total of all node heights in any tree of height i will be less-than or equal to the total node height of the complete binary tree with height i.

Thus the total number of comparisons needed to "heapify" a tree of height i will be, at most, $O(2(2^{i+1} - (i+2)))$. This linear complexity can be simplified to $O(i)$ where i is the tree height. Thus, heapification is a linear process.

Operations on a Max-Heap

Once a heap has been built, a Heapsort can simply remove the maximum value (root node) and create the output, sorted array one item at a time. This process is somewhat akin to the Selection Sort but much more efficient.

When the root node in a heap is removed to become part of the final, ordered data set, the last item on the heap is promoted to fill the vacancy at the root position. Clearly, in many cases, this last item will now be out of place (that is, it may be smaller than one of its new children). To ensure that the modified heap retains the max-heap property it becomes necessary to "push down" the newly promoted root item until it is back in the right place. This pushing down process entails examining the node's key value and comparing it with the key value of the node's greatest child. If the node's greater child is larger in value than the node itself, a swap is performed. The process repeats, following the node from the root down through demotion, until no swap is needed. At this point the heap is back in order, the new root may be popped off, and the sorting process can continue.

The process of removing the root, promoting the last node, and re-heapifying continues until the heap is exhausted.

Analysis

While the Heapsort is a constant factor slower than the Quicksort for average data sets, it still has a complexity of $(nlog_2n)$ for best, worst and average data. It also has some interesting properties that make it particularly useful for sorting data sets that are too large to fit into primary computer memory.

As we have seen, the initial heapification operation is a linear time complexity process. The push-down operation described is only a variation of the heapify process. In the worst case $2i$ comparisons will be needed to fully demote a given node from the root to its proper place.

Source Code

Below is an implementation of a Heapsort in C:

```
/* ------------------------------------------------------------------------ */
/*   h_sort - perform a heap sort on a range of data items
 */

void h_sort(int low, int hi) {
  int i;

  /* heapify the data set */
  heapify(low, hi);

  /* repeatedly swap top and last then pushdown the promoted last */
  for (i = TOPITEM - 1; i > 1; i--) {
```

```
    swap(&data[1], &data[i]);
    pushdown(1, i - 1);
  }
}
```

As you can see, the real trick is taking the input data set and making it into a max-heap. The support routine to push down the root node in a heap to its proper place is used both to put the data into a heap initially and at each step in the sort process.

```
/* ------------------------------------------------------------------------- */
/*   pushdown - push a data item down the heap until in the proper place
 */

void pushdown(int which, int limit) {

  /* we will determine the node's max child */
  int max_child = which * 2;

  /* if this is a leaf node (i.e. it has no children) then we're done */
  if (max_child > limit) return;

  /* if it has a second child, make max_child the index of the greater kid */
  if (((which * 2) + 1) <= limit)
    if (data[max_child] < data[(which * 2) + 1]) max_child = (which * 2) + 1;

  /* now see if the node in question if greater than its max child... */
  if (data[which] < data[max_child]) {

    /* if it's not, swap them and keep going with the push down */
    swap (&data[which], &data[max_child]);
    pushdown(max_child, limit);
  }
}

/* ------------------------------------------------------------------------- */
/*   heapify - given a data range, make it into a heap
 */

void heapify(int low, int hi) {

  /* we only have to start at the first node with children */
  int mid = (low + hi) / 2;
  int i;

  /* work backwards to the top of the heap calling pushdown */
```

```
  for (i = mid; i > 0; i--) pushdown(i, TOPITEM-1);
}
```

References

1. Manber, Udi. 1989, *Introduction to Algorithms: A Creative Approach.* (Reading, MA: Addison-Wesley), pp. 139-141.

0.3.4 Benchmarking the Quicksort and the Heapsort

The Quicksort and the Heapsort are two $nlog_2n$ algorithms for sorting data. One way to more empirically measure the performance of these two algorithms is to write a benchmarking program and count the number of comparisons each uses in order to sort a random data set. Another total that may be of interest is the number of swaps a given algorithm uses in the sorting process. Below is a benchmarking program that does just that.

```
/**********************************************************************
**                                                                   **
**  Sorting Benchmark                                                **
**                                                                   **
**  Data and Algorithm Analysis                                      **
**                                                                   **
**  Assignment #6                                                    **
**                                                                   **
**  Due Dec 1, 1997                                                  **
**                                                                   **
**  Scott Gasch                                                      **
**                                                                   **
**  available online at http://wannabe.guru.org/alg/alg.html         **
**                                                                   **
**********************************************************************/

#include <stdio.h>
#include <stdlib.h>

/*
 *  We need math.h for the log stuff in the table
 *
 */

#include <math.h>

/*
 *  We need time.h in order to seed the random number generator based on the
```

```
 *  value of the system clock...
 *
 */

#include <time.h>

/*
 *  This is the index number of the top item in the data set to be sorted...
 *
 */

int TOPITEM = 100;

/*
 *  The random value generator will fill the data set with values between
 *  zero and RANGE, inclusive.
 *
 */

int RANGE = 10000;

/*
 *  The array itself is global to memory and me the headache of passing it
 *  all around the place.
 *
 */

int data[1000];

/*
 *  To keep track of number of swaps and comparisons each alg uses
 *
 */

int num_swaps, num_comps;


/*
 *  Function protos
 *
 */

void swap (int *a, int *b);
void q_sort (int low, int hi);
void show_array(void);
void fill_array(void);
int get_pivot (int low, int hi);
```

```
void pushdown(int which, int limit);
void heapify(int low, int hi);
void h_sort(int low, int hi);


/* ------------------------------------------------------------------------ */
/*   show_array - dump the contents of the data set to stdout
 */
void show_array(void) {
  int i;

  for (i = 1; i < TOPITEM; i++) {
    printf("%d: %d\n", i, data[i]);
  }
}

/* ------------------------------------------------------------------------ */
/*   fill_array - place random numbers between
 *   0..RANGE (inclusive) in data[]
 */
void fill_array(void) {
  int i;
  float r;

  /* clean slate */
  num_comps = num_swaps = 0;

  /* [re]randomize */
  srand(time(NULL));

  for (i = 1; i < TOPITEM; i++) {
    r = (float) rand() / (float) RAND_MAX;
    data[i] = r * RANGE + 1;
  }
}

/* ------------------------------------------------------------------------ */
/*   get_pivot - return the index of the selected pivot value
 */

int get_pivot (int low, int hi) {

  /* safety net, this should not happen */
  if (low == hi) return(data[low]);

     /* return the greater of the first two items in the range
      *
```

```
     * return( (data[low] > data[low+1]) ? low : (low+1) );
     *
     */

  /* return the midpoint as the pivot element */
  return( (low + hi) / 2 );
}

/* ---------------------------------------------------------------------------- */
/*   swap - given two pointers to integers, swap their contents
 */

void swap (int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
  num_swaps++;
}

/* ---------------------------------------------------------------------------- */
/*   q_sort - Quicksort a data range
 */

void q_sort (int low, int hi) {
  int pivot_index;                /* index in the data set of the pivot */
  int pivot_value;                /* the value of the pivot element     */
  int left, right;

  /* select the pivot element and remember its value */
  pivot_index = get_pivot(low, hi);
  pivot_value = data[pivot_index];

  /* do the partitioning */
  left = low; right = hi;
  do {

    /* move left to the right bypassing elements already on the correct side */
    while ((left <= hi) && (data[left] < pivot_value)) {
      num_comps++;
      left++;
    }
    num_comps++;

    /* move right to the left bypassing elements already on the correct side */
    while ((right >= low) && (pivot_value < data[right])) {
      num_comps++;
      right--;
```

```
    }
    num_comps++;

    /*
     *  if the pointers are in the correct order then they are pointing to two
     *  items that are on the wrong side of the pivot value, swap them...
     */
    if (left <= right) {
      swap(&data[left], &data[right]);
      left++;
      right--;
    }

  } while (left <= right);

  /* now recurse on both partitions as long as they are large enough */
  if (low < right) q_sort(low, right);
  if (left < hi) q_sort(left, hi);
}
```

```
/* ------------------------------------------------------------------------ */
/*   pushdown - push a data item down the heap until in the proper place
 */

void pushdown(int which, int limit) {

  /* we will determine the node's max child */
  int max_child = which * 2;

  /* if this is a leaf node (i.e. it has no children) then we're done */
  if (max_child > limit) return;

  /* if it has a second child, make max_child the index of the greater kid */
  if (((which * 2) + 1) <= limit) {
    num_comps++;
    if (data[max_child] < data[(which * 2) + 1]) max_child = (which * 2) + 1;
  }

  /* now see if the node in question if greater than its max child... */
```

```
  num_comps++;
  if (data[which] < data[max_child]) {

    /* if it's not, swap them and keep going with the push down */
    swap (&data[which], &data[max_child]);
    pushdown(max_child, limit);
  }
}

/* ------------------------------------------------------------------------- */
/*   heapify - given a data range, make it into a heap
 */

void heapify(int low, int hi) {

  /* we only have to start at the first node with children */
  int mid = (low + hi) / 2;
  int i;

  /* work backwards to the top of the heap calling pushdown */
  for (i = mid; i > 0; i--) pushdown(i, TOPITEM-1);
}

/* ------------------------------------------------------------------------- */
/*   h_sort - perform a heap sort on a range of data items
 */

void h_sort(int low, int hi) {
  int i;

  /* heapify the data set */
  heapify(low, hi);

  /* repeatedly swap top and last then pushdown the promoted last */
  for (i = TOPITEM - 1; i > 1; i--) {
    swap(&data[1], &data[i]);
    pushdown(1, i - 1);
  }
}


/* ------------------------------------------------------------------------- */
/*   main - the driver
 */

int main(void) {
  int save_array[1000];       /* used to store/reset the global array */
```

```
 int tab[6][4];              /* tabular values for #comps, #swaps    */
 int i, j;                   /* loop control                         */
 int n;

 /*
  * requirements:
  *
  *  1) see above
  *
  *  2) Check your code works by comparing to answers for hw 4, #2
  *
  */

 init_data();
 q_sort(1, 10);
 printf("homework data, quicksort:"
"%d comparisons, %d swaps\n", num_comps, num_swaps);

 hwdata();
 h_sort(1, 10);
 printf("homework data, heapsort:"
"%d comparisons, %d swaps\n", num_comps, num_swaps);


 /*
  *
  * (footnote: my version of quicksort works better)
  *
  *
  * 3) Use a random number generator to generate ints from 1 to 10000
  *    (see fill_array())
  *
  * 4) Take 100 random numbers and demonstrate your code works...
  *
  */

 TOPITEM=101; RANGE=10000;
 printf("creating an array of 100 random numbers... here it is:\n");
 fill_array();
 show_array();

 /* save this untarnished list so heapsort can sort later */
 for (i = 0; i<1000; i++) save_array[i] = data[i];

 /* run a qsort and show the nice people */
 printf("quick sorting...\n");
 q_sort(1, TOPITEM-1);
```

```
show_array();

/* restore the list to its unsorted splendor */
for (i = 0; i<1000; i++) data[i] = save_array[i];

/* heap sort */
printf("heap sorting...\n");
h_sort(1, TOPITEM);
show_array();


/*
 *
 * 5) Take N random numbers (500<N<1000 by 100 steps) and make a
 *    nice table.
 *
 */

for (i = 500; i<=1000; i+=100) {

  TOPITEM = i+1;
  num_swaps = num_comps = 0;

  fill_array();

  /* store data set */
  for (j = 0; j<1000; j++) save_array[j] = data[j];

  /* quicksort */
  q_sort(1, TOPITEM);

  /* save the results */
  tab[(i - 500) / 100][0] = num_comps;
  tab[(i - 500) / 100][1] = num_swaps;

  /* get ready to heapsort */
  num_swaps = num_comps = 0;

  /* restore the data set and doit */
  for (j = 0; j<1000; j++) data[j] = save_array[j];
  h_sort(1, TOPITEM);

  /* save results */
  tab[(i - 500) / 100][2] = num_comps;
  tab[(i - 500) / 100][3] = num_swaps;
}
```

```
  /* now show them the table */

  printf("                     Quicksort           Heapsort\n"
         "  N     NlogN     #comps   #swaps     #comps   #swaps\n"
 "-------------------------------------------------------\n");
  for (i = 0; i<6; i++) {
    n = (i * 100) + 500;
    printf(" %4d   %4d      %5d   %5d       %5d   %5d\n",
   n,
   n * (int)(log(n) / log(2)),
   tab[i][0],
   tab[i][1],
   tab[i][2],
   tab[i][3]);
  }
}
```

0.3.5 Insertion Sort

The Insertion Sort is slow when compared to the $nlog_2n$ algorithms like Quicksort, Mergesort and Heapsort. It has one redeeming characteristic, however: it is very fast for "almost sorted" data. For this reason several other algorithms have been designed to nearly sort a data set and then call the Insertion Sort as their last step. The most notable of these is the algorithm designed by D.L. Shell called the Shellsort.

The insertion sort operates by finding the proper location of an item in an already sorted subset of data and then shifting the range of data via a ripple-shift operation in such a way as to leave a hole for the item to be inserted. The subset of already sorted data is initially empty and grows by one element with each ripple-shift, insert operation.

Analysis

In the worst case each item inserted will be compared to the previous $n-1$ sorted items. This means the overall time complexity of the algorithm is:

$$1 + 2 + 3 + 4 + 5 + 6 ... + (n-2) + (n-1)$$

Rewritten as a sum this is:

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

Which is a variant of the arithmetic series and reduces to:

$$\frac{1}{2}(n-1)(n-2)$$

By multiplying the above out it becomes clear that this is a $O(n^2)$ algorithm.

For best case data, sets that are already sorted, the Insertion sort runs in linear time. However for average data sets the efficiency is n^2.

Improvements

In order to improve the speed at which the insertion sort runs for non-best-case data sets it is possible to use a binary search in order to place item number n into the $n - 1$ sorted items. This leads to a drastic reduction in number of comparisons; each binary search takes only $log_2 n$ comparisons and each of the n items must be inserted leading to a total of $nlog_2 n$ comparisons. However, the number of data move operations will remain unchanged. Thus even the improved insertion sort is deemed a quadratic algorithm.

Source Code

Below is an Insertion Sort implemented in C:

```
void insert (int *array, int num, int x) {
  int i;

  for (i = n-1; i >= 0; i--) {
    if (x < array[i]) array[i+1] = a[i] else break;
  }
  array[i+1] = x;
}

void insertion_sort (int *array, int n) {
  int i;

  for (i = 0; i < n; i++) {
    insert(array, i, array[i]);
  }
}
```

0.3.6 Shellsort

The Shellsort, also known as the **diminishing increment sort**, exploits the excellent best-case performance of the Insertion Sort by attempting to “almost sort” a data set and then call an Insertion Sort to finish off the sorting process. This is very similar to the optimization discussed at the end of the Quicksort section.

Analysis

Shellsort, like the Mergesort and Quicksort, among others, divides a data set into subsets, sorts the subsets, and then recombines these sorted subsets. A Shellsort's average performance is thought to be about $n^{3/2}$. The exact complexity of this algorithm is still being debated and is far too advanced for this presentation. Suffice to say that for mid-sized data sets it performs nearly as well if not better than the faster ($nlogn$) sorts.

Source Code

Below is a C implementation of the Shellsort:

```
void insertionsort (int *A, int n, int incr) {
  int i, j;

  for (i = incr; i < n; i+= incr)
    for (j = i; (j >= incr) && (A[j] < A[j-incr]); j-=incr)
      swapi (A[j], A[j-incr]);
}


void shellsort (int *array, int n) {
  int i, j;

  for (i = (n/2); i > 1; i /= 2)
    for (j = 0; j < i; j++)
      insertionsort(\&array[j], n-j, i);

  insertionsort(array, n, 1);
}
```

0.3.7 Selection Sort

Logically, the Selection Sort is very easy to follow. It looks through the data set for the data item with the largest (or smallest) value and then, once found, puts this item at the end (or beginning) of the data set. The Selection Sort continues to look for maxima or minima on the remaining set of data until it has exhausted the data set.

Analysis

One advantage of the Selection Sort is that there are very few record swaps. Unlike some other sorting algorithms, like insertion sort, for example, the Selection sort swaps elements only when it knows it is

swapping one of them into the correct position in the final data set. In total a maximum of $n-1$ data movement operations must be performed by this algorithm.

For this reason if you are dealing with a situation where it is "expensive" to exchange data items, this sort might be the best solution. Another, possibly better, way to sort "expensive to move" records is to keep a pointer to each record's key and simply sort the pointers. This is akin to labeling all the boxes in your warehouse with cards that give their location and then sorting the cards rather than getting out a forklift and sorting the boxes.

This is a relatively expensive algorithm in terms of data comparison operations. It always will cost $n-1$ comparisons to find the maximum item in a set. Since the maximum must be found n times, the overall number of comparisons needed by the Selection Sort is quadratic, order n^2 to be exact.

This sort is essentially a modified **Bubble Sort** which does not continually swap adjacent elements but, rather, remembers where to put elements once they are selected. Although Selection Sort is more efficient than the Bubble Sort by a constant factor in most situations, there is usually a better way to sort data than to choose a selection sort.

Source Code

A C implementation of the Selection Sort which finds the lowest value in the remaining data set is given above.

```
void selectionsort(int *array, int size) {
  int i, j, lowindex;

  for (i = 0; i < size - 1; i++) {
    lowindex = i;
    for (j = size - 1; j > i; j--)
      if (array[j] < array[lowindex]) lowindex = j;
    swapi (\&array[i], \&array[lowindex]);
  }
}
```

0.3.8 Bubble Sort

The Bubble Sort has no redeeming characteristics. It is always very slow, no matter what data it is sorting. This algorithm is included here for the sake of completeness, not because of any merit.

The C code for a Bubble sort follows:

```
void bubblesort (int *array, int size) {
  int i, j;

  for (i = 0; i < size; i++)
```

```
    for (j = size - 1; j > i; j--)
      if (array[j] < array[j-1]) swapi(&array[j], &array[j-1]);
}
```

A Bubble Sort always runs in n^2 time and is a very poor choice for a sorting algorithm in any circumstance.

0.3.9 Bucket and Radix Sorting

The bucket sort is a non-comparison-based sorting algorithm in which we allocate one storage location for each item to be sorted and proceed to process the data set to be sorted assigning each item into its corresponding bucket. In order to bucket sort n unique items in the range of 1 through m, allocate m buckets and then iterate over the n items assigning each one to the proper bucket. Finally loop through the buckets and collect the items putting them into final order.

The bucket sort is a good choice when items to be sorted are from a small data range that is known in advance.

One problem with the bucket sort is that, if the range of items to be sorted is very large, an unreasonable amount of memory is required to allocate enough buckets. A method very similar to bucket sorting called radix sorting elegantly handles this problem. In a radix sort the data to be sorted is broken down into several buckets, like in the bucket sort. The difference is that many items are assigned to each bucket in the radix sort because to assign an item to a bucket the radix sort only considers a subset of the item key. Often the bucket to which an item is assigned with the radix sort is based on a certain digit or subset of the item's key value. The radix sort recursively processes the contents of each bucket by allocating sub-buckets and assigning items into sub-buckets by considering a different subrange of the items' keys. This process continues until there is only one item per bucket at which point items are recollected in order.

For example, sorting social security numbers with a radix method one might divide the entire data space into ten sets. The members of set number one would be the social security numbers beginning with the digit 1, etc. The numbers in set one would be subdivided based on the second digit in each social security number, and so on until the entire set was sorted.

Analysis

In order to bucket sort n items in the range of 1 to m, m buckets must be allocated, n items must be assigned to a bucket, and the contents of m buckets must be collected. The overall complexity is $O(m + n)$ operations requiring $O(m)$ storage locations. In contrast, in order to radix sort n items considering each key from left to right requires $O(kn)$ time. In the preceding formula, n is the number of items to be sorted while k is the number of times the radix sort has to recurse, subdividing the contents of each bucket. As you can see, both of these algorithms have linear running time complexity.

0.3.10 The nth Largest

The problem of finding the nth-largest (or smallest) item in a set is closely related to the process of sorting. Indeed one way to accomplish this goal is to sort the list and then count off until reaching the item desired. However, as we will see, this is a waste of time as there are many algorithms specifically designed to find nth-largest items faster than the time required to sort and count.

Modified Radix Sort

One way to tackle this problem is with a modified radix sort. To find the nth largest number in a set, classify the set into b buckets by considering the most significant piece of each item's key value. More than one item will be in each bucket, usually. Maintain a count of how many items are placed in each bucket.

Now, since we are only interested in the nth item, consider the first bucket. If there are less than n items in this bucket, ignore its contents and continue to the second bucket. Continue to consider and ignore buckets until reaching the one which must house the nth largest data item. At this point either explicitly sort this bucket and count off items or, more likely, continue the radix sort procedure.

Modified Quicksort

It's also possible to efficiently find the nth largest number in a set using a modified version of the Quicksort. As you recall, the Quicksort selects a pivot value and proceeds to partition the data set into two halves: those which are less than or equal to the pivot value and those which are greater than it. After partitioning a normal Quicksort would reiterate on both halves. However, if there are less than n items in the left partition it makes no sense to reiterate on it. The item we seek must be in the right partition. The coverse is true also; if there are more than n items in the left partition it makes no sense to reiterate on the right partition.

Source Code

```
/* ------------------------------------------------------------------------ */
/*   get_pivot - return the index of the selected pivot value
 */

int get_pivot (int low, int hi) {

  /* safety net, this should not happen */
  if (low == hi) return(data[low]);

  /* return the greater of the first two items in the range  */
  return( (data[low] > data[low+1]) ? low : (low+1) );
}
```

```
/* ---------------------------------------------------------------------- */
/*   swap - given two pointers to integers, swap their contents
 */

void swap (int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
  num_swaps++;
}

/* ---------------------------------------------------------------------- */
/*   q_sort - Quicksort a data range
 */

int nth_largets_q_sort (int nth, int low, int hi) {
  int pivot_index;                /* index in the data set of the pivot */
  int pivot_value;                /* the value of the pivot element     */
  int left, right;

  if (low == hi) return(data[low]);

  /* select the pivot element and remember its value */
  pivot_index = get_pivot(low, hi);
  pivot_value = data[pivot_index];

  /* do the partitioning */
  left = low; right = hi;
  do {

    /* move left to the right bypassing elements already on the correct side */
    while ((left <= hi) && (data[left] < pivot_value)) {
      num_comps++;
      left++;
    }
    num_comps++;

    /* move right to the left bypassing elements already on the correct side */
    while ((right >= low) && (pivot_value < data[right])) {
      num_comps++;
      right--;
    }
    num_comps++;

    /*
     *  if the pointers are in the correct order then they are pointing to two
     *  items that are on the wrong side of the pivot value, swap them...
```

```
     */
    if (left <= right) {
      swap(&data[left], &data[right]);
      left++;
      right--;
    }

  } while (left <= right);

  if ((right - low) > nth) nth_largest_q_sort(nth, low, right);
    else nth_largets_q_sort((nth - left), left, hi);

}
```

0.4 Graph Algorithms

A graph is a finite set of **vertices** and **edges**. Each edge in a graph must begin and end at a different vertex (and, thus, no edges can "loop" joining one vertex to itself). Moreover between any pair of vertices there can be only one edge. Vertices are sometimes also called **nodes**. Nodes joined by an edge are said to be **adjacent**.

We call moving between vertices along edges **traversing** a graph.

In **directed graphs**, or **di-graphs**, every edge has orientation. It is only possible to traverse a given edge in one direction. All edges are like one-way streets; traffic can only flow in a single direction.

In **weighted graphs**, each edge has a **cost** associated with it. To move from a pair of vertices with an edge joining them you have to pay a toll – the price of that edge. It is often possible, however, to circumvent expensive edges by finding a less expensive, more indirect route, between an edge's endpoints. In the following section we will explore several algorithms designed to find the shortest path between two vertices.

0.4.1 Graph Representation

It is possible to represent graphs in computer memory with a variety of different data structures. One strategy is to use an **adjacency matrix**. An adjacency matrix is a two dimensional array in which the row and column headers represent different vertexes in the graph. A one-way edge between, for example, vertex one and three, is denoted by a positive value in array position (1, 3). In a weighted graph the value stored in each array location corresponds to the weight or cost of each particular edge. If an edge is bi-directional it has two entries in the matrix. One entry represents the (source, destination) route while the other handles the (destination, source) return route.

Another method for representing graphs is as a more complicated **linked list** structure. Each vertex in the graph is a node in a master linked list. Another linked list emanates from each vertex node and denotes the vertexes directly adjacent to a given source vertex. This method, often called an **adjacency**

list, is more space efficient than the adjacency matrix for graphs which do not have very many edges (so-called **sparce graphs**).

0.4.2 Graph Traversal

Given a starting point it is possible to systematically traverse an entire graph (directed or not) visiting every node reachable from the starting point. This can be accomplished in many ways but by far the most common two are **breadth-first traversal** and **depth-first traversal**. Because **trees** are really just restricted graphs, these two traversals can and are used on trees (binary or otherwise) as well.

Depth-first traversal

Depth-first searching, also sometimes known as backtracking, builds a path from emanating at the starting point and continuing as far as possible into the graph. Each new edge traversed must be one that has not already been covered by the algorithm. The path-building operation continues until it reaches a point at which there is no edge leading to a vertex that has not already been visited. At such a point we backtrack to the previous node, choose a new edge from that point, and build as long a path as possible from there. In backtracking, if all the edges from the previous node have been tried we backtrack again to the next-previous node and so on. Eventually this method will visit every node in a connected graph.

The actual traversal is often accomplished using a stack data structure. First, the starting node is pushed onto the stack. Then the following process repeats:

- Pop a node off the stack
- Traverse to this current node
- Push all nodes adjacent to the current node onto the stack

The process terminates when the stack is empty. The same process can be implemented as a recursive function which uses the system stack instead of a data segment structure.

Depth-first traversal of a graph is an $O(V + E)$ operation where V is the number of vertices in the graph and E is the number of edges.

```
int visited[MAX_VERTS];

/*
 * Assume visited[x] = 0 for all 0 <= x <= MAX_VERTS on first call.
 *
 */
void df_traverse(int vertex) {
  link t;
```

```
  /*
   * record the fact that we've visited this vertex,
   * possibly call a special function here to do something more than
   * simply mark it ``visited''
   *
   */
  visited[vertex] = 1;

  /*
   * traverse recursively at every vertex adjacent to vertex.  This
   * assumes an adjacency list representation of the graph.
   *
   */
  for (t = adj[k]; t != NULL; t = t->next) {
    if (!visited[t->v]) df_traverse(t->v);
  }
}
```

References

1. Sedgewick 1998, *Algorithms in C* (Reading, MA: Addison-Wesley). pg. 247.

Breadth-first traversal

Like a depth-first traversal, breadth-first traversal algorithms visit each node in a connected graph. However, when a breadth-first traversal arrives at a certain node, v, it visits all neighbors of node v before continuing to process other, more distant, parts of the graph. Whereas a depth first traversal can be implemented recursively, a breadth-first search is not naturally recursive. Instead of using a stack data structure, breadth-first traversal usually makes use of a queue. A queue is much like a line of people; it operates on a FIFO (first in first out) or "first come, first served" principle.

The actual traversal procedure begins by enqueueing the starting node. Then the following process repeats:

- Dequeue a current node
- Enqueue all non-visited nodes adjacent to the current node
- Mark non-visited nodes adjacent to the current node visited

The above cycle repeats until the queue becomes empty.

```
void bf_traverse (int vertex) {
  link t;
```

Euler Cycles

A cycle in a graph is a traversal which visits no node more than once. An Euler cycle is a special cycle that traverses all the edges in a graph and visits every node at least once. It can be proven that an Euler cycle can exist in a graph if and only if all vertices are of even degree. That is to say, the must be an even number of edges emanating from every vertex in the graph.

Hamilton Circuits

0.4.3 Floyd's Algorithm - Shortest Paths

This algorithm is designed to find the least-expensive paths between all the vertices in a graph. It does this by operating on a matrix representing the costs of edges between vertices.

Before we invoke Floyd's algorithm we must build a matrix, usually in a two-dimensional array. If there are n vertices in our graph, our matrix will be nxn. Each row in the matrix represents a "starting" vertex in the graph while each column in the matrix represents an "ending" point in the graph. If there is an edge between a starting point i and ending point j in the graph, the cost of this edge is placed in position (i,j) of the matrix. If we are dealing with an undirected graph in which all edges are bi-directional, an entry is also made in position (j,i) of the matrix. If there is no edge directly linking two vertices, an infinite (or, in practice, very large) value is placed in the (i,j) position of the matrix to specify that it is impossible to directly move from i to j.

For example, if we have a graph in which points 1 and 5 are connected by a bi-directional edge with a cost of 22 units, we would place the number 22 into positions (1,5) and (5,1) of our matrix.

Once we have set this matrix up, we use Floyd's algorithm to compute the shortest distance between all points in the graph. Floyd's algorithm is given below in C:

```
int floyds(int *matrix) {
  int k, i, j;

  for (k = 1; k <= n; k++)
    for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++)
        if (matrix[i][j] < (matrix[i][k] + matrix[k][j]))
          matrix[i,j] = matrix[i][k] + matrix[k][j];
}
```

When this routine finishes the entries in all positions of the matrix represent the lowest-cost traversal between the row-vertex and column-vertex.

Floyd's algorithm works by looking for all non-direct paths between two vertices that have a less-expensive total cost than the best way yet found to move between said vertices. If such a path is found, it becomes the value against which future indirect paths between these vertices are tested. In the end,

each element of the matrix represents the lowest-cost traversal between the vertices it's row and column represent. Remember that if the graph is directed, so is the answer in (i,j) of the matrix; moreover, (i,j) may not be equal to (j,i) in a di-graph.

It is clear that Floyd's algorithm takes n^3 time. In the next section we will discuss an alternative to Floyd's algorithm called Dijkstra's algorithm. It is important to note, however, that for dense graphs (i.e. graphs with many edges) Floyd's algorithm is as good as or better than Dijkstra's algorithm.

0.4.4 Dijkstra's Algorithm - Shortest Path

While Floyd's algorithm determines the lowest-cost path between all vertices in a graph, Dijkstra's was designed to find the lowest cost path between a single starting vertex and all of the other vertices in a graph. Dijkstra's can, clearly, be used to obtain the same information as Floyd's algorithm if it is called repeatedly for every vertex in a graph.

Dikjstra's algorithm is a **greedy algorithm** which means, if given a choice, it operates by choosing the biggest or most valuable alternative.

The first step of this algorithm is to "label" the starting vertex with an ordered pair, (-,0), and initialize a distance counter to one. Next, look at all edges between labeled vertices and unlabeled vertices. If the cost of a particular edge added to the second item in the ordered pair of the initial vertex is equal to the distance counter, label the terminal vertex of this edge (`name_of_starting_vertex`, `distance_counter`) and continue this process, incrementing the distance counter by one at each iteration and continuing until all vertices in the graph are labeled. The second member of the ordered pair at each vertex is the lowest cost walk from it to the starting vertex. The first member of the ordered pair of the ordered pair is the node immediately preceding the current node on the shortest path from source to destination.

The algorithm actually coded is implemented in a slightly different manner. Because it is inefficient to store the distance counter and look at every edge at every increment, I choose to begin at the starting vertex and traverse to all nodes reachable from it. Each node is labelled with a distance from the start and a previous vertex. Each node is also enqueued for later processing.

Once all nodes adjacent to the starting vertex are processed, labelled and enqueued, the algorithm dequeues the first node. All nodes adjacent to this vertex that have not been visited are labelled and enqueued. Additionally, any node that has been visited but can be reached more cheaply is re-labelled and enqueued.

This process continues until the queue is empty. The shortest path to a given vertex n is labelled on that vertex. The path can be determined by examining the prior steps recursively back to the starting node.

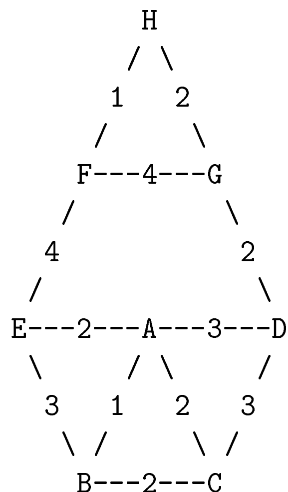
Analysis

Dijkstra's algorithm has a complexity of $O(N^2)$. Does anyone have a proof for this?

Source Code

An implementation of Dijkstra's algorithm is given below.

```
#include <stdlib.h>
#include <stdio.h>

#include "debug.h"

#define NUM_NODES                          8
#define NONE                               9999
#define X(l)                               ((l) - 'A')
#define Y(n)                               ((n) + 'A')

struct _NODE
{
  int iDist;
  int iPrev;
};
typedef struct _NODE NODE;

struct _QITEM
{
  int iNode;
  int iDist;
  int iPrev;
  struct _QITEM *qNext;
};
typedef struct _QITEM QITEM;

QITEM *qHead = NULL;

/*
                                       H
                                      / \
                                     1   2
                                    /     \
                                   F---4---G
                                  /         \
                                 4           2
                                /             \
                               E---2---A---3---D
                                \     / \     /
                                 3   1   2   3
                                  \ /     \ /
                                   B---2---C
```



```
   */



int AdjMatrix[NUM_NODES][NUM_NODES] =
{
        /*   A     B     C     D     E     F     G     H     */
  /* A */ { NONE,    1,    2,    3,    2, NONE, NONE, NONE },
  /* B */ {    1, NONE,    2, NONE,    3, NONE, NONE, NONE },
  /* C */ {    2,    2, NONE,    2, NONE, NONE, NONE, NONE },
  /* D */ {    3, NONE,    2, NONE, NONE, NONE,    2, NONE },
  /* E */ {    2,    3, NONE, NONE, NONE,    4, NONE, NONE },
  /* F */ { NONE, NONE, NONE, NONE,    4, NONE,    4,    1 },
  /* G */ { NONE, NONE, NONE,    2, NONE,    4, NONE,    2 },
  /* H */ { NONE, NONE, NONE, NONE, NONE,    1,    2, NONE }
};

int g_qCount = 0;


void print_path (NODE *rgnNodes, int chNode)
{
  if (rgnNodes[chNode].iPrev != NONE)
  {
    print_path(rgnNodes, rgnNodes[chNode].iPrev);
  }
  printf (" %c", Y(chNode));
  fflush(stdout);
}


void enqueue (int iNode, int iDist, int iPrev)
{
  QITEM *qNew = (QITEM *) malloc(sizeof(QITEM));
  QITEM *qLast = qHead;

  if (!qNew)
  {
    fprintf(stderr, "Out of memory.\n");
    exit(1);
  }
  qNew->iNode = iNode;
  qNew->iDist = iDist;
  qNew->iPrev = iPrev;
```

```
  qNew->qNext = NULL;

  if (!qLast)
  {
    qHead = qNew;
  }
  else
  {
    while (qLast->qNext) qLast = qLast->qNext;
    qLast->qNext = qNew;
  }
  g_qCount++;
  ASSERT(g_qCount);
}


void dequeue (int *piNode, int *piDist, int *piPrev)
{
  QITEM *qKill = qHead;

  if (qHead)
  {
    ASSERT(g_qCount);
    *piNode = qHead->iNode;
    *piDist = qHead->iDist;
    *piPrev = qHead->iPrev;
    qHead = qHead->qNext;
    free(qKill);
    g_qCount--;
  }
}


int qcount (void)
{
  return(g_qCount);
}


int main(void)
{

  NODE rgnNodes[NUM_NODES];
  char rgchLine[255];
  char chStart, chEnd, ch;
  int iPrev, iNode;
  int i, iCost, iDist;
```

```
  for (ch = 'A'; ch <= 'A' + NUM_NODES; ch++)
  {
    rgnNodes[X(ch)].iDist = NONE;
    rgnNodes[X(ch)].iPrev = NONE;
  }

  printf("What is the starting node? ");
  gets(rgchLine);
  chStart = toupper(rgchLine[0]);

  printf("What is the ending node? ");
  gets(rgchLine);
  chEnd = toupper(rgchLine[0]);

  if (chStart == chEnd)
  {
    printf("Shortest path is 0 in cost.\nJust stay where you are.\n");
    exit(0);
  }
  else
  {
    chStart = X(chStart);
    chEnd = X(chEnd);
    rgnNodes[chStart].iDist = 0;
    rgnNodes[chStart].iPrev = NONE;

    enqueue (chStart, 0, NONE);

    while (qcount() > 0)
    {
      dequeue (&iNode, &iDist, &iPrev);
      for (i = 0; i < NUM_NODES; i++)
      {
        if ((iCost = AdjMatrix[iNode][i]) != NONE)
{
  if ((NONE == rgnNodes[i].iDist) ||
      (rgnNodes[i].iDist > (iCost + iDist)))
  {
    rgnNodes[i].iDist = iDist + iCost;
            rgnNodes[i].iPrev = iNode;
    enqueue (i, iDist + iCost, iNode);
          }
}
      }
    }
```

```
    printf("Shortest path is %d in cost.\n", rgnNodes[chEnd].iDist);
    printf("Path is: ");
    print_path(rgnNodes, chEnd);
  }


  exit(0);
}
```

Some of the above functions are not implemented and the code has not ever been tested but it should suffice to communicate the general idea behind Dijkstra's algorithm.

0.4.5 Spanning Trees

A **spanning tree** in a graph is a **tree** that visits every node in the graph. Recall that a tree is a set of nodes and edges that has a single **root node** and no **circuits**.

In many graphs each edge has an associated weight or cost. Such graphs are called **weighted graphs**. Often in weighted graphs we want to find a spanning tree of minimal cost.

In this section we explore several algorithms for finding minimal spanning trees in graphs.

Prim's Algorithm

This algorithm for finding a minimal cost spanning tree in a connected graph is very easy to follow. It begins by adding the lowest cost edge and its two endpoints to the solution set. It then loops adding the lowest cost edge that connects a vertex in the solution set to one outside it. It also adds the endpoint of this edge that is not already in the solution set. The algorithm terminates when all vertices are in the solution set. The edges and vertices in the solution set at this point constitute a minimal cost spanning tree of the input graph.

Source Code Below is an implementation of Prim's algorithm in C++. It relies on several classes and methods that are not included but should be sufficient to give you an understanding of Prim's algorithm and serve as a guide for coding your own implementation.

```
void prim(graph \&g, vert s) {

  int dist[g.num_nodes];
  int vert[g.num_nodes];

  for (int i = 0; i < g.num_nodes; i++) {
```

```
    dist[i] = INFINITY;

  dist[s.number()] = 0;

  for (i = 0; i < g.num_nodes; i++) {
    vert v = minvertex(g, dist);

    g.mark(v, VISITED);
    if (v != s) add_edge_to_MST(vert[v], v);
    if (dist[v] == INFINITY) return;

    for (edge w = g.first_edge; g.is_edge(w), w = g.next_edge(w)) {
      if (dist[g.first_vert(w)] = g.weight(w)) {
         dist[g.second_vert(w)] = g.weight(w);
 vert[g.second_vert(w)] = v;
      }
    }
  }
}


int minvertex(graph \&g, int *d) {
  int v;

  for (i = 0; i < g.num_nodes; i++)
    if (g.is_marked(i, UNVISITED)) {
      v = i; break;
    }

  for (i = 0; i < g.num_nodes; i++)
    if ((g.is_marked(i, UNVISITED)) && (dist[i] < dist[v])) v = i;

  return (v);
}
```

Kruskal's Algorithm

Kruskal's algorithm for finding a minimal spanning tree in a connected graph is a greedy algorithm; that is, given a choice, it always processes the edge with the least weight.

This algorithm operates by considering edges in the graph in order of weight from the least weighted edge up to the most while keeping track of which nodes in the graph have been added to the spanning tree. If an edge being considered joins either two nodes not in the spanning tree, or joins a node in the spanning tree to one not in the spanning tree, the edge and its endpoints are added to the spanning tree. After considering one edge the algorithm continues to consider the next higher weighted edge. In

the event that a graph contains equally weighted edges the order in which these edges are considered does not matter. The algorithm stops when all nodes have been added to the spanning tree.

Note that, while the spanning tree produced will be connected at the end of the algorithm, in intermediate steps Kruskal can be working on many independent, non-connected sections of the tree. These sections will be joined before the algorithm completes.

Often this algorithm is implemented using parent pointers and **equivalence classes**. At the start of the processing, each vertex in the graph is an independent equivalence class. Looping through the edges in order of weight, the algorithm groups the vertices together into one or more equivalence classes to denote that these nodes have been added to the solution minimal spanning tree.

It is a good idea to process the edges by putting them into a min-heap. This is usually much faster than sorting the edges by weight since, in most cases, not all the edges will be added to the minimal spanning tree. See the section on the **heapsort** and the **heap data structure** for more information about min-heaps.

Source Code Here is some untested code that implements Kruskal's algorithm in C++. It relies on a heap class, a graph class, and a general tree class with equivalence class operation methods. Although none of these classes are included here, this should be sufficient to give you an idea of how to implement Kruskal's algorithm:

```
void kruskal(graph \&g) {

  // our general equivalence class tree will have as many nodes as are
  // in the input graph G.  We will be linking these nodes up in the
  // body of the algorithm.
  gentree a(g.num_nodes());

  // we will also need an edge heap array in order to process the
  // edges in G one at a time in the order small->large
  edge e(G.num_edges());

  // the edge we are considering... also a loop storage variable
  edge w;

  // local counter
  int edge_count = 0;

  // loop control
  int i;

  // this is the number of spanning trees we are considering.  This
  // will start at the number of vertices in G but, as vertices are
  // joined up by lowest-weight edges, this will reduce.  Eventually
  // it will hit one at which point we know we should stop the algorithm.
```

```
  int numMST;

  // these are the endpoints of the edge we are considering
  vertex v, u;

  // save all the edges in G for later processing.
  for (i = 0; i < g.num_nodes; i++) {
    for (w = g.first_edge(i); g.is_edge(w); w = g.next_edge(w))
      e[edge_count++] = w;


  // min-heapify the edges
  minheap h(e, edge_count);

  // initially there are the same numer of equivalence classes as
  // the number of vertices in G; each vertex is its own class.
  numMST = G.num_nodes();

  // while we have not yet arrived at one equivalence class (spanning
  // tree)...
  for (i = 0; numMST > 1; i++) {

     // consider the next smallest edge
     w = h.remove_min();

     // look at w's endpoints
     v = g.vertex_one(w); u = g.vertex_two(w);

     // if these two vertices are not in the same equivalence class
     // (i.e. they are in different tree sections) join their
     // respective equivalence classes into one.
     if (a.differ(v.identifier(), u.identifier())) {
       a.union(v, u);
       add_edge_to_MST(w);

       // one less
       numMST--;
     }
  }
}
```

0.4.6 Transitive Closure

0.5 Miscellaneous Algorithms

0.5.1 Maximum Consecutive Subsequence

Given a sequence of numbers, how can the subsequence of greatest (or least) value be determined? In this section two efficient algorithms for finding maximum consecutive subsequences are explored and analyzed. Before these methods can be presented, though, a precise understanding of the problem is essential.

Imagine a series of n numbers $(x_1, x_2, x_3, x_4, ...x_n)$ called set S. The following algorithms will find the subset S' of set S such that the sum of the members of set S' is greater than the sum of numbers in any other subset of the original set S. If all the members of S are positive then set S' will be the entire S set. However, when negative values are included in S this problem becomes more interesting and difficult to solve.

Divide and Conquer Solution

The premise behind the first solution to the maximum subsequence problem is that any set can be divided into two sets of one-half the aggregate size. The maximum subsequence of the original set must: be entirely on the left half of the division, be entirely on the right half of the division, or span the middle of the division.

Analysis In order to calculate the maximum subsequence of a set this algorithm first calculates three values, max_right, max_left and max_spanning. Once these values are computed, two comparisons will yield the maximum subsequence in the set. The cost of computing these three values is what determines the complexity of this algorithm.

$$T(n) = 2T(n/2) + C(n) + 3$$

Because this is a recursive solution, the time to find the maximum subsequence of one set depends on the amount of time spent finding the maximum subsequence in each half. It is for this reason that in the above recurrence relation $T(n)$ is related to $2T(n/2)$. However the addition of the middle term, $C(n)$ greatly complicates this analysis. $C(n)$, above, represents the cost of calculating the spanning subsequence of maximum value.

This term $C(n)$ involves processing each of the n items. One pointer works right from the center until it reaches the right set limit. At each step we add a value to a running sum and check whether the new sum has a greater value than the max so far. This process involves a total of $n/2$ comparisons. The same process takes place as another pointer moves left from the center until it reaches the leftmost limit. So, the $C(n)$ procedure is a linear process. This means the recurrence relation of the entire algorithm can be written as:

$$T(n) = 2T(n/2) + n + 2$$

By expanding this relation we get:

$$\begin{aligned}
&T(n) = 2T(n/2) + n + 2\\
&T(n/2) = 2T(n/4) + n/2 + 2\\
&T(n/4) = 2T(n/8) + n/4 + 2\\
&\vdots\\
&T(1) = 0
\end{aligned}$$

So we know that $T(n)$ can be written as:

$$T(n) = n + 2 + 2(n/2 + 2 + 2(n/4 + 2 + 2(...1)...))$$

That is:

$$T(n) = \sum_{i=1}^{q} n + 2^i$$

The term q in the above expression is a bit non-deterministic. If we restrict n to be 2^x in order to simplify this exercise, then $q = log_2 n = x$.

$$T(n) = \sum_{i=1}^{log_2 n} n + 2^i$$

Writing that out in long format:

$$T(n) = n + 2 + n + 4 + n + 8 + ...n + 2^{x-1} + n + 2^x$$

Now we'll invoke cancellation by subtracting $T(n)$ from $2T(n)$

$$\begin{array}{llllllll}
2T(n) = & & 2n+4 & +2n+8 & +2n+16 & +... & 2n+2^x & +2n+2^{x+1}\\
-T(n) = & -n-2 & -n-4 & -n-8 & -n-16 & -... & n-2^x & \\
\hline
T(n) = & -n-2 & +n & +n & +n & +... & n & +2n+2^{x+1}
\end{array}$$

(x above is $log_2 n$). This gives us:

$$T(n) = (n-1)log_2 n + 2^{log_2 n+1}$$

Or...

$$T(n) = (n-1)log_2 n + 2n$$

The dominant term is $(n-1)log_2 n$ making this an $O(nlog_2 n)$ algorithm.

Source Code Below is the source code to the divide and conquer maximum subsequence problem in C:

```
#include <stdio.h>

#define MAXSEQ 400

int numbers[MAXSEQ];
int endat;
int doit(int, int);

int main(void) {
  int i = 0;
  int num;
  int count;

  printf("enter the numbers, -99999 to end... \n");

  do {
    scanf("%d", &num);
    numbers[i] = num;
    i++;
  } while (num != -99999);
  count = i - 1;

  printf("max subseq value was %d end at %d\n", doit(0, count), endat);
}


int doit(int l, int r) {
  int mid;
  int ma, mb, mc;
  int i, sum, max;

  if (l == r) {
    return(numbers[l]);

  } else {
    mid = (l + r) / 2;

    if (mid == l) {
      ma = doit(l, l);
      mb = doit(r, r);
    } else {
      ma = doit(l, mid);
      mb = doit(mid, r);
    }
```

```
    /* now build mc */
    i = 1;
    sum = numbers[mid];
    max = numbers[mid];

    /* work our way right adding to the sum at each step */
    while ((mid + i) <= r) {
      sum += numbers[mid + i];
      if (sum > max) max = sum;
      i++;
    }
    mc = max - numbers[mid];

    i = 1;
    sum = numbers[mid];
    max = numbers[mid];

    /* now work left */
    while ((mid - i) >= l) {
      sum += numbers[mid - i];
      if (sum > max) max = sum;
      i++;
    }
    mc += max;

    if (mc > mb)
      if (mc > ma)
return(mc);
      else
return(ma);
    else
      if (mb > ma)
return(mb);
      else
return(ma);
  }
}
```

Linear Solution

Source Code

```
#include <stdio.h>

#define MAXSEQ 400
```

```
int numbers[MAXSEQ];
int count;
int endat;
void doit(void);

int main(void) {
  int i = 0;
  int num;

  printf("enter the numbers, -99999 to end... \n");

  do {
    scanf("%d", &num);
    numbers[i] = num;
    i++;
  } while (num != 0);
  count = i - 1;

  doit();
}


void doit(void) {
  int maxsofar = numbers[0];
  int maxendhere = numbers[0];
  int i, a, b;

  for (i = 1; i <= count; i++) {

    a = maxendhere + numbers[i];
    b = numbers[i];
    if (a > b) maxendhere = a; else maxendhere = b;

    if (maxendhere > maxsofar) {
      maxsofar = maxendhere;
      endat = i;
    }
  }

  printf("Max value subseq was %d ending at pos %d\n", maxsofar, endat);
}
```

0.5.2 Permutations

Imagine we have a collection of n distinct objects. There are $n!$ ways to order these objects; that is, we can form $n!$ different arrangements of these n objects. This is true because any such arrangement will consist of n items, no matter which happens to be first. To choose the first object in a particular arrangement we have n options. However, to choose the second object after already having placed the first, we are left with one less choice. The first object is fixed at this point. Thus, we have $(n-1)$ alternatives. As we place more and more objects we have less and less choices of objects to place. The summation below follows from this discussion:

$$\Pi_{i=1}^{n}(n)$$

This is the same as $n!$.

This section presents an algorithm for calculating all possible permutations (that is, not just the number of permutations but the actual permutated data) given the number of distinct data items to be arranged.

Source Code

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_NUM 100

//
//  A single call to permut(k, n) will produce (n - k + 1)!
//  permutations consisting of the integers:
//
//               r[1] ... r[k-1] ... r[k] ... r[n]
//
//  In the output, the first r[1]...r[k-1] numbers will not
//  change.  The r[k]...r[n] numbers will be permiated.  An
//  initial call of permut(1, n) will produce the full n!
//  permutations of these n numbers.
//
//

void permut(int k, int n, int *nums)
{
  int i, j, tmp;

  /* when k > n we are done and should print */
  if (k <= n)
  {

    for (i = k; i <= n; i++)
    {
```

```
      /**
        *  each element i is promoted to the kth place while the rest
        *  of the items from k to i-1 are shifted to make room with
        *  a ripple-shift operation.
        *
        **/
      tmp = nums[i];
      for (j = i; j > k; j--)
      {
nums[j] = nums[j-1];
      }
      nums[k] = tmp;

      /* recurse on k+1 to n */
      permut(k + 1, n, &(nums[0]));

      for (j = k; j < i; j++)
      {
        nums[j] = nums[j+1];
      }
      nums[i] = tmp;
    }
  }
  else
  {
    for (i = 1; i <= n; i++)
    {
      printf("%d ", nums[i]);
    }
    printf("\n");
  }
}

int main(void)
{
  int iCount;
  int rgNums[MAX_NUM];
  int i;

  printf("Enter n: ");
  scanf("%d", &iCount);


  /* create a workspace of numbers in their respective places */
  for (i = 1; i <= iCount; i++)
  {
```

```
    rgNum[i] = i;
  }

  printf("Permutations:\n");
  permut(1, iCount, rgNum);
}
```

References

1. Ammeraal, Leendert 1996, *Algorithms and Data Structures in C++* (New York, NY: John Wiley & Sons Ltd.), pp. 290-296

0.5.3 Combinations

Given a set of n distinct items, a k combination of these items is any subset of the original set with exactly k members (where $0 < k <= n$). There is a formula for determining exactly how many k combinations (subsets of size k) exist for a given set of size n:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

The notation $\binom{n}{k}$ is often read "the number of n items taken k at a time."

However sometimes it is not enough to know how many combinations exist for a given set but rather we are interested in determining what those combinations are. That is to say, what, exactly, are the subsets of size k.

Source Code

```
int *r, n, k;

int main (void) {
  printf(``Please enter n and k (0 < k <= n): ``);
  scanf(``%d %d'', &n, &k);
  r = (int *) malloc(k+1 * sizeof(int));
  r[0] = 0;
  combin(1);
}

void combin (int m) {
  int i;
```

```
  if (m <= k) {
    for (r[m] = r[m-1] + 1; r[m] <= n-k-m; r[m]++)
      combin(m + 1);
  } else {
    for (i = 1; i <= k; i++)
      printf(``%d `, r[i]);
    printf(``\n'');
  }
}
```

Here is a non-recursive implementation written by Jock Cooper that, given the number of items and the number of items in a set, computes the sets.

```
/* --- combin.c --- */

#include <stdio.h>
#include <stdlib.h>

int increment(int numcnt, int setcnt, int counter[]);
int addup(int setcnt, int counter[]);
int initcounter(int setcnt, int counter[]);
int combinations(int numcnt, int setcnt, int items[], int counter[], int results[]);

main(int c,char *v[])
{
  int numitems, numcomb;
  int *array, *results;
  int *counter;
  int idx;

  if (c != 3)
  {
    fprintf(stderr,"usage: %s numitems numcomb\n", v[0]);
    exit(1);
  }
  numitems= atoi(v[1]);
  numcomb = atoi(v[2]);

  array = calloc(numitems, sizeof(int));
  counter = calloc(numcomb, sizeof(int));
  results = calloc(numcomb, sizeof(int));

  for (idx=0; idx< numitems; ++idx)
  {
    array[idx] = rand() % 5000;
    printf("%d ", array[idx]);
  }
  printf("\n");
  initcounter(numcomb, counter);
```

```
  while (combinations(numitems, numcomb, array, counter, results))
  {
    for (idx=0; idx<numcomb; ++idx)
    {
      printf("%d ", results[idx]);
    }
    printf("\n");
  }
}

int initcounter(int setcnt, int counter[])
{
  int idx;

  for (idx=0; idx<setcnt; ++idx)
  {
    counter[idx]=1;
  }
  counter[--idx]=0;
}


int combinations(int numcnt, int setcnt, int items[], int counter[], int results[])
{
  int idx, iidx;

  if (increment(numcnt,setcnt,counter) == 0)
  {
    return 0;
  }

  for (iidx=idx=0; idx<setcnt; ++idx)
  {
    iidx+=counter[idx];
    results[idx] = items[iidx-1];
  }
  return 1;
}

int increment(int numcnt, int setcnt, int counter[])
{
  int digit=setcnt-1;
  int digit2;
  int overflow;

  do
  {
    counter[digit]++;
    if (addup(setcnt,counter) > numcnt)
    {
```

```
      overflow=1;

      digit2=digit;
      --digit;
      if (digit <0)
      {
        return 0;
      }

      while (digit2 < setcnt) counter[digit2++]=1;
    }
    else
    {
      overflow=0;
    }
  }
  while (overflow==1);
  return 1;
}


int addup(int setcnt, int counter[])
{
  int result=0;
  int idx;

  for (idx=0; idx<setcnt; ++idx)
  {
    result+=counter[idx];
  }
  return result;
}
```

References

1. Ammeraal, Leendert 1996, *Algorithms and Data Structures in C++* (New York, NY: John Wiley & Sons Ltd.), pp. 296-300

0.5.4 Exponentiation

```
/* efficiantly compute x^n */

double pow(double x; int n) {
  double y = 1;
```

```
  int neg = (n < 0);

  if (neg) n = -n;

  while (n) {
    if (n & 1) y *= x;
    x *= x;
    n >>= 1;
  }
  return neg ? 1.0/y : y;
}
```

0.5.5 Julian Calendar Algorithms

Included here are several Julian day routines. The `date_to_julian` is a calculation that assigns (almost) any date a single (large) integer. This integer cooresponds to the number of days between a set starting date and the given date. The difference between the Julian numbers of any two dates is exactly the number of days between the respective dates. By performing modulo division on the Julian number of a date you can easily determine on what day-of-the-week it fell. A division of a date's Julian number by seven with no remainder means that the date is a Sunday. A remainder of six means that the day is a Saturday. Further, a date's validity can be checked by converting it into a Julian number, converting the resultant Julian number back into a calendar date, and comparing the two.

A Brief History of the Julian Calendar

The original Julian Calendar was introduced by Julius Caesar (thus the name) in 44 B.C. The length of a single year was to be equal to the time time it takes for the earth to orbit the sun once. This amount was estimated to be 365 days and 6 hours. Every fourth year the extra six hours were collected and added as an extra day to the year, creating a leap year of 366 days.

Because of a slight inaccuracy in estimation of a year's length under the original Julian Calendar (The true amount of time it takes for the Earth to complete one solar orbit is closer to 365 days, 5 hours, 49 minutes and 12 seconds), over a long period of time the extra minutes and seconds began to accumulate and things began to go wrong. Most notably the Spring Equinox, which occurs at a measurable fixed point in the Earth's orbit, began not to fall on the same date.

The "new style" Julian Calendar, upon which the algorithm presented here is based, dates from the year 1582. That is when Pope Gergory XIII decided to fix the messed up Julian calendar. To do so he skipped the date forward from October 5th to October 14th to trim time added due to the inaccuracy of the Julian calendar. He also redefined which years would be counter as leap years. Before, every year evenly divisible by four was leap; Pope Gregory changed the rules so that: All years evenly divisible by four were leap years *except* those evenly divisible by 100 *and* not also evenly divisible by 400. For instance, 1700, 1800, and 1900 are not leap years. They are evenly divisible by four but are also divisible by 100. The year 2000 is a leap year because it is evenly divisible by four and, even though it is also divisible by 100, it is also divisible by 400. The Gregorian calendar also set New Year's day to January

first. The “new style” Julian Calendar is popularly called the Gregorian Calendar in honor of Pope Gregory.

Interestingly, only Catholic countries used this better calendar right away. Many Protestent countries, however, adopted it over the next two centuries. Most did so around 1700. England held out until 1752. Others waited until the early 1900s. Some Greek Orthadox countries opted to create their own calendar (a modified version of the Gregorian system) and use it instead today.

The “new style” (Gregorian) calendar should accurately model the Earth's movement around the sun for some 40,000 more years.

A mistake commonly made is to confuse “Julian day numbers” with the “Julian calendar.” The Julian calendar is the system implemented in 44 A.D. by Caesar. Julian day numbers (which are based on the calendar and are calculated by the code in the following section) were proposed by a monk and named “Julian” after his father (not Caesar). Day numbers are simply the number of days elapsed since a fixed starting date. That date is arbitrary but often set to January 1st, 4713 B.C. (which the monk thought was pretty darn close to when the biblical creation story in Genesis must have taken place). Other day numbering routines use different days as “day zero.”

Source Code

Below are routines to convert from mon, day, year dates into julian dates and the reverse.

```
#include <math.h>
#include <stdio.h>

/* Oct. 15, 1582 */
#declare IGREG (15+31L*(10+12L*1582))

long date_to_julian (int mon, int day, int year) {

  long jul;
  int ja;
  int jy = year;
  int jm;

  if (jy == 0) return(0);
  if (jy < 0) jy++;

  if (mon > 2)
    jm = mon + 1;
  else {
    jy--;
    jm = mon + 13;
  }

  jul = (long) (floor(365.25 * jy) + floor(30.6001 * jm) + day + 1720995);
```

```
  if (day + 31L * (mon + 12L * year) >= IGREG) {
    ja = (int)(0.01 * jy);
    jul += 2 - ja + (int) (0.25 * ja);
  }

  return(jul);
}


void julian_to_date(long julian, int *mon, int *day, int *year) {

  long ja, jalpha, jb, jc, jd, je;

  if (julian >= IGREG) {
    jalpha = (long) (((float) (julian - 1867216) - 0.25) / 36524.25);
    ja = julian + 1 + jalpha - (long) (0.25 * jalpha);
  } else
    ja = julian;

  jb = ja + 1524;
  jc = (long) (6680.0 + ((float) (jb - 2439870) - 122.1) / 365.25);
  jd = (long) (365 * jc + (0.25 * jc));
  je = (long) ((jb - jd) / 30.6001);
  *day = jb - jd - (long) (30.6001 * je);
  *mon = je - 1;
  if (*mon > 12) *mon -= 12;
  *year = jc - 4715;
  if (*mon > 2) (*year)--;
  if (*year <= 0) (*year)--;
}
```

References

1. Binstock, Andrew and Rex, John. 1995, *Practical Algorithms for Programmers* (Reading, MA: Addison-Wesley), pp. 363-369.

2. Craig, John C. 1988, *Microsoft QuickBASIC Programmer's Toolbox* (Redmond, WA: Microsoft Press), pp. 60-70

3. Press, William H. et al. 1992, *Numerical Recipies in C* (New York, NY: Cambridge University Press), pp. 12-14

0.5.6 Greatest Common Divisor, Least Common Multiple

The greatest common divisor of two integers, i and j, is the largest integer that divides both i and j exactly. Likewise the least common multiple of the same two integers is returns the smallest integer is an even multiple of both i and j. The given implementation of the `greatest_common_div` operation runs $O(logm)$. Because the `least_common_mul` operation calls `greatest_common_div`, it, too, runs in logarithmic time. While these are not, in the strictest sense, "computer science" algorithms they may be useful to you nonetheless.

Source Code

The algorithm to compute the greatest common divisor presented below was proposed by Euclid in 300 B.C. Historians believe that Euclid did not come up with this on his own but rather recorded an algorithm that was as much as 200 years older. This is the oldest non-trivial algorithm known to exist. Two procedures are included here, a recursive and iterative version.

```
void euclid_i (int m, int n) {
    int r;

    while (1) {
      r = m % n;
      if (r == 0) {
        printf("%d\n", n);
        break;
      }
      m = n;
      n = r;
    }
}

void euclid_r (int m, int n) {
    int r;

    r = m % n;
    if (!r) {
      printf("%d\n", n);
      return;
    }
    euclid_r(n, r);
}

int least_common_mul(int i, int j) {
  return (abs( (i * j) / euclid_i(i, j)));
}
```

0.5.7 Addition Chaining

Addition chaining is a method of computing $a^x \bmod n$ efficiently that is sometimes used in cryptographic systems. Instead of performing a series of multiplications followed by a large division, there are ways to minimize the size and number of multiplications. Because the operations of multiplication and modulo division are distributive, it is faster to take the modulus between successive multiplications thus limiting the size of the operands being multiplied. For instance,

$$a^8 modn = ((a^2 modn)^2 modn)^2 modn$$

If x in $a^x \bmod n$ is not a power of two then the binary representation of x can be used.

$$a^{25} modn = (a * a^{24}) modn$$

But we also know that $a^{24} modn = (a^8 * a^{16}) modn$. a^8 is $((a^2)^2)^2$ $(a^{(2^3)})$. Likewise $a^{16} = a^{(2^4)}$.

Source Code

The algorithm presented is also called the binary square and multiply method. It reduces the number of operations (multiplies and modulo divisions) to about $1.5 * (log_2 x)$.

Source Code

```
//
// Compute x^y mod n efficiently
//

unsigned long AddChain(unsigned long x, unsigned long y, unsigned long n)
{
  unsigned long s, t, u;

  s = 1;
  t = x;
  u = y;

  while (u)
  {

    //
    // if u is not evenly divisible by two then multiply and mod once
    //
    if (u & 1)
    {
```

```
      s = (s * t) % n;
    }

    //
    // now, u must be divisible by two (every other number is...)
    // So, divide u by two, square t, and divide by n.
    //
    u >>= 1;
    t = (t * t) % n;
  }

  return(s);
}
```

References

1. Schneier, Bruce 1996, *Applied Cryptography, 2nd Ed.* (New York, NY: John Wiley & Sons, Inc.). pg. 243-244.

0.5.8 Fibonacci Calculation

The Fibonacci sequence is easy to understand but is a frequent target for inefficient calculation. The first two terms in the sequence are ones. Beginning with the third number in the series, the value is defined to be the sum of the previous two. So, the beginning of the Fibonacci sequence is as follows:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...$$

Unfortunately the typical way to implement a Fibonacci calculation function is by recursion. This is inefficient because of the enormous call stack generated to calculate non-trivial terms in the series and the great amount of duplicate work the computer must do to give you a result. In the next section I give an efficient iterative Fibonacci number calculator.

There is also an explicit formula for finding the nth Fibonacci number. It is:

$$f(n) = \frac{\phi^n - (-1/\phi)^n}{\sqrt{5}}$$

Where ϕ is the golden ratio, an irrational number approximately equal to 0.6180339...

Source Code

```
/* return the nth term of the fib seq */
```

```
int fib3(int n) {
  int i = 1;
  int j = 0;
  int k = 0;
  int h = 1;
  int t = 999;

  while (n > 0) {

    if (n % 2) {
      t = j * h;
      j = i * h + j * k + t;
      i = i * k + t;
    }

    t = h * h;
    h = 2 * k * h + t;
    k = k * k + t;
    n = n / 2;
  }

  return(j);
}
```

Source Code

Sedgewick gives a dynamic programming example of efficiently calculating the nth Fibonacci number as follows:

```
/*
 * we will use this array to store computations and avoid repeating
 * work that has already been completed.
 *
 */
int known_fib[MAX_FIB];


/*
 * assume that known_fib is initialized to all UNKNOWN or with valid
 * Fibonnacci numbers
 *
 */
int fib(int i) {

  int t;
```

```
  if (known_fib[i] != UNKNOWN) return(know_fib[i]);
  if (i == 0) t = 0;
  if (i == 1) t = 1;
  if (i > 1) t = fib(i - 1) + fib(i - 1);
  known_fib[i] = t;
  return(t);
}
```

References

1. Sedgewick 1998, *Algorithms in C* (Reading, MA: Addison-Wesley). pg. 212.

0.5.9 Cartesian and Polar Coordinates

These functions can be used to convert coordinates between Cartesian (rectangular) and polar systems. They are not really "computer science" algorithms in the strictest sense of the word but you may find them useful nonetheless.

Source Code

```
#define PI              3.141593;
#define HALF_PI         (PI / 2);

float angle(float X, float Y) {
  if (X == 0) {
    if (Y == 0)
      return(HALF_PI);
    else if (Y < 0)
      return(-HALF_PI);
    else
      return(0);
  } else if (Y == 0)
    if (X == 0)
      return(PI);
    else
      return(0);
  } else {
    if (X < 0) {
      if (Y > 0)
        return(atan(Y / X) + PI);
      else
        return(atan(Y / X) - PI);
    } else {
```

```
      return (atan(Y / X));
    }
  }
}


float magnitude(float X, float Y) {
  return(sqrt(X * X + Y * Y));
}


void polar_to_rectangular(float r, float theta, float *X, float *Y) {
  *X = r * cos(theta);
  *Y = r * sin(theta);
}


void rectangular_to_polar(float X, float Y, float *r, float *theta) {
  *r = magnitude(X, Y);
  *theta = angle(X, Y);
}
```

0.5.10 Soundex English word-sounding Algorithm

M. K. Odell and R. C. Russell patented the Soundex phonetic comparison system in 1918 and 1922. Soundex coding takes an English word and produces a four digit representation of the word designed to match the phonetic pronunciation of the word. It is normally used for "fuzzy" searches where a close match may be desired. For example, to come up with alternative possibilities for a misspelled word some spelling checker programs generate a Soundex code for the misspelled word and then suggest other words with the same Soundex value. Additionally Soundex codes are often used on surnames which are difficult to spell.

The creation of a Soundex code is a pretty simple operation. The first step is to remove all non-English letters or symbols. In the case of accented vowels, simply remove the accents. Any hyphens, spaces, etc... also. In addition, remove all H's and W's unless they are the initial letter in the word. Next, take the first letter in the word and make it the first letter of the Soundex code. For each remaining letter in the word, translate it to a number with the table below and concatenate the numbers, preserving order, on to the Soundex value.

```
      A, E, I, O, U, Y = 0
            B, F, P, V = 1
C, G, J, K, Q, S, X, Z = 2
                  D, T = 3
                     L = 4
                  M, N = 5
                     R = 6
```

Now, combine any double numbers into a single instance of that number. Further, if the first number in the Soundex value is the same as the code number for the initial letter, delete the first number. Now, remove all zeros from the Soundex string. Finally, return the first four characters of the end product as the Soundex encoding. If there are less than four characters to be returned, concatenate enough zeros to make the length four.

Source Code

```
#include <ctype.h>
#include <string.h>

int soundex (char *input, char *output) {

  char values[] =

    /* ABCDEFGHIJKLMNOPQRSTUVWXYZ */
     ``01230120022455012623010202'';


  int done;
  char this_char, prev_char, replace_char;
  char code[5];


  strcpy(code, ``     '');
  if (isalpha(this_char = toupper(*input))) {
    code[0] = this_char;
    done = 1;
    input++;
  } else return(0);
  prev_char = ' ';

  while (done < 4 && isalpha(*input)) {
    this_char = toupper(*input);
    replace_char = values[this_char = 'A'];

    if ((replace_char != prev_char) && (replace_char != '0')) {
      prev_char = code[done++] = replace_char;
    }
    input++;
  }


  if ((*input != '\0') && (!isalpha(*input))) {
```

```
    return(0);
  } else {
    strcpy (output, code);
    return(1);
  }
}
```

References

1. Binstock, Andrew and Rex, John. 1995, *Practical Algorithms for Programmers* (Reading, MA: Addison-Wesley), pp. 158-160.

0.5.11 Metaphone Algorithm

Like the previously discussed Soundex algorithm, Metaphone is a system for transforming words into codes based on phonetic properties. However, unlike Soundex which operates on a letter-by-letter scheme, Metaphone analizes both single consonants and groups of letters called dipthongs.

Metaphone was invented by Lawrence Philips and first described in *Computer Language* magazine in December 1990.

The Metaphone algorithm operates by first removing non-English letters and characters from the word being processed. Next, all vowels are also discarded unless the word begins with an initial vowel in which case all vowels except the initial one are discarded. Finally all consonents and *groups of consonents* are mapped to their Metaphone code. The rules for grouping consonants and groups thereof then mapping to metaphone codes are fairly complicated; for a full list of these conversions check out the comments in the source code section.

Source Code

```
//
// Metaphone.cc
//
// Implementation of Metaphone Algorithm
//


//*****************************************************************************
//
// Metaphone::Metaphone()
//
Metaphone::Metaphone()
{
name = "metaphone";
```

```
}


//***************************************************************************
// Metaphone::~Metaphone()
//
Metaphone::~Metaphone()
{
}


//***************************************************************************
//
// Metaphone invented by Lawrence Philips
// Metaphone implementation copied from C Gazette, June/July 1991, pp 56-57,
// author Gary A. Parker, with changes by Bernard Tiffany of the
// University of Michigan, and more changes by Tim Howes of the
// University of Michigan.
//

/* Character coding array */
static char     vsvfn[26] = {
1, 16, 4, 16, 9, 2, 4, 16, 9, 2, 0, 2, 2,
/* A   B  C   D  E  F  G   H  I  J  K  L  M  */
2, 1, 4, 0, 2, 4, 4, 1, 0, 0, 0, 8, 0 };
/* N  O  P  Q  R  S  T  U  V  W  X  Y  Z  */

/* Macros to access character coding array */
#define vowel(x)  ((x) != 0 && vsvfn[(x) - 'A'] & 1)   /* AEIOU */
#define same(x)   ((x) != 0 && vsvfn[(x) - 'A'] & 2)   /* FJLMNR */
#define varson(x) ((x) != 0 && vsvfn[(x) - 'A'] & 4)   /* CGPST */
#define frontv(x) ((x) != 0 && vsvfn[(x) - 'A'] & 8)   /* EIY */
#define noghf(x)  ((x) != 0 && vsvfn[(x) - 'A'] & 16)  /* BDH */

#define MAXPHONEMELEN   6

void
Metaphone::generateKey(char *word, String &key)
{
char *n;
String ntrans;

/*
 * Copy Word to internal buffer, dropping non-alphabetic characters
 * and converting to upper case
 */
```

```
ntrans << "0000";

for (; *word; word++)
{
if (isalpha(*word)) ntrans << *word;
}
ntrans.uppercase();

/* ntrans[0] will always be == 0 */
n = ntrans.get();

/* Pad with nulls */
*n++ = 0;
*n++ = 0;
*n++ = 0;
*n = 0;

/* Assign pointer to start */
n = ntrans.get() + 4;
ntrans << '\0';
ntrans << '\0';
ntrans << '\0';

/* Check for PN, KN, GN, AE, WR, WH, and X at start */
switch (*n)
{

case 'P':
case 'K':
case 'G':

/* 'PN', 'KN', 'GN' becomes 'N' */
if (*(n + 1) == 'N')
*n++ = 0;
break;

case 'A':
/* 'AE' becomes 'E' */
if (*(n + 1) == 'E')
*n++ = 0;
break;

case 'W':
/* 'WR' becomes 'R', and 'WH' to 'H' */
if (*(n + 1) == 'R')
*n++ = 0;
else if (*(n + 1) == 'H')
```

```
{
*(n + 1) = *n;
*n++ = 0;
}
break;

case 'X':
/* 'X' becomes 'S' */
*n = 'S';
break;
}

//
// Now, loop step through string, stopping at end of string or when
// the computed 'metaph' is MAXPHONEMELEN characters long
//

for (; *n && key.length() < MAXPHONEMELEN; n++)
{

/* Drop duplicates except for CC */
if (*(n - 1) == *n && *n != 'C') continue;

/* Check for F J L M N R or first letter vowel */
if (same(*n) || (*(n - 1) == 0 && vowel(*n)))
key << *n;

else
{
switch (*n)
{
case 'B':
// B unless in -MB
if (*(n + 1) || *(n - 1) != 'M')
key << *n;
break;
case 'C':
/*
 * X if in -CIA-, -CH- else S if in
 * -CI-, -CE-, -CY- else dropped if
 * in -SCI-, -SCE-, -SCY- else K
 */
if (*(n - 1) != 'S' || !frontv(*(n + 1)))
{
if (*(n + 1) == 'I' && *(n + 2) == 'A')
key << 'X';
else if (frontv(*(n + 1)))
```

```
key << 'S';
else if (*(n + 1) == 'H')
key << (((*(n - 1) == '\0' && (!vowel(*(n + 2)))
  || *(n - 1) == 'S')) ?
      (char) 'K' : (char) 'X');
else
key << 'K';
}
break;
case 'D':

/*
 * J if in DGE or DGI or DGY else T
 */
key << ((*(n + 1) == 'G' && frontv(*(n + 2)))
? (char) 'J' : (char) 'T');
break;
case 'G':

/*
 * F if in -GH and not B--GH, D--GH,
 * -H--GH, -H---GH else dropped if
 * -GNED, -GN, -DGE-, -DGI-, -DGY-
 * else J if in -GE-, -GI-, -GY- and
 * not GG else K
 */
if ((*(n + 1) != 'J' || vowel(*(n + 2))) &&
(*(n + 1) != 'N' || (*(n + 1) &&
 (*(n + 2) != 'E' ||
                                          *(n + 3) != 'D'))) &&
(*(n - 1) != 'D' || !frontv(*(n + 1))))
key << (frontv(*(n + 1)) &&
*(n + 2) != 'G') ? (char) 'G' : (char) 'K';
else if (*(n + 1) == 'H' && !noghf(*(n - 3)) &&
 *(n - 4) != 'H')
key << 'F';
break;
case 'H':

/*
 * H if before a vowel and not after
 * C, G, P, S, T else dropped
 */
if (!varson(*(n - 1)) && (!vowel(*(n - 1
)) ||
  vowel(*(n + 1))))
key << 'H';
```

```
break;
case 'K':

/*
 * dropped if after C else K
 */
if (*(n - 1) != 'C')
key << 'K';
break;
case 'P':

/*
 * F if before H, else P
 */
key << (*(n + 1) == 'H' ?
(char) 'F' : (char) 'P');
break;
case 'Q':

/*
 * K
 */
key << 'K';
break;
case 'S':

/*
 * X in -SH-, -SIO- or -SIA- else S
 */
key << ((*(n + 1) == 'H' ||
 (*(n + 1) == 'I' && (*(n + 2) == 'O' ||
  *(n + 2) == 'A')))
? (char) 'X' : (char) 'S');
break;
case 'T':

/*
 * X in -TIA- or -TIO- else 0 (zero)
 * before H else dropped if in -TCH-
 * else T
 */
if (*(n + 1) == 'I' && (*(n + 2) == 'O' ||
*(n + 2) == 'A'))
key << 'X';
else if (*(n + 1) == 'H')
key << '0';
else if (*(n + 1) != 'C' || *(n + 2) != 'H')
```

```
key << 'T';
break;
case 'V':

/*
 * F
 */
key << 'F';
break;
case 'W':

/*
 * W after a vowel, else dropped
 */
case 'Y':

/*
 * Y unless followed by a vowel
 */
if (vowel(*(n + 1)))
key << *n;
break;
case 'X':

/*
 * KS
 */
if (*(n - 1) == '\0')
key << 'S';
else
key << "KS";        /* Insert K, then S */
break;
case 'Z':

/*
 * S
 */
key << 'S';
break;
}
}
}
}


//****************************************************************************
// void Metaphone::addWord(char *word)
```

```
//
void
Metaphone::addWord(char *word)
{
if (!dict)
{
dict = new Dictionary;
}

String  key;
generateKey(word, key);

if (key.length() == 0)
return;
String  *s = (String *) dict->Find(key);
if (s)
{
if (mystrcasestr(s->get(), word) != 0)
(*s) << ' ' << word;
}
else
{
dict->Add(key, new String(word));
}
}
```

References

1. Binstock, Andrew and Rex, John. 1995, *Practical Algorithms for Programmers* (Reading, MA: Addison-Wesley), pp. 160-163.

2. The DLR Freeware/Shareware Archive, Jan 1998.

0.5.12 A Pseudo-Random Number Generator

The below given `rand` function returns a **pseudo-random** integer in the range 0 to 999999999 inclusive. It works by creating an array of one-hundred integers from an initial key (or "seed") and then then partitioning the range into two subranges. The first fifty-five entries are integers to be used in pairs to generate a random result. The next forty-two entries are random results already generated but "on hold" until selected in order to disturb any pattern in output caused by a poor key and to shuffle the order of random integers on output. The final two elements in the structure simply contain internal pointers which always reference elements in the 1 to 55 range. The values referenced by these internal pointers are subtracted and then adjusted to be greater than zero if needed to form a result.

This table-subtraction method produces a good degree of randomness alone, but another technique is also used to augment its operation. The result stored at position 100 in the array is used to select one

of the 42 entries in locations 56 to 97 in the table for replacement. The number which is being replaced at this location is both promoted up to location 100 and tagged to be returned from the function as it's return value. The number generated by subtraction in the 1 to 55 range replaces this newly promoted number at its old position where it will wait as a possible random return value until a subtraction selects it. This additional "shuffling" serves to negate any subtle patterns that the sequence of values generated with rotating table subtractions might produce.

The `init_rand` function takes a seed key in the form of an arbitrary length ASCII string. This key is used to seed the random table used, later, by `rand` to produce random numbers. You will notice that the algorithm adds the string "aEbFcGdHeI " the the beginning of the key before initializing the table. This is to ensure there are both characters with even and odd ASCII values in the seed string. Once the (adjusted) ASCII values of the key are put into the table, the `init_rand` algorithm cycles through the table shuffling values to form an even, random initial numeric distribution in the table. It then initializes the pointers in positions 98, 99 and 100. At this point everything is ready for the first call to `rand`.

Analysis

At the heart of this algorithm are two techniques described by Knuth (See references). The algorithm used in `rand` if fairly fast, operating in constant time while the call to `init_rand` will operate in linear time.

Source Code

```
int r[100]; // "global" pseudo-random table --
            // must be visible to rand and init_rand

//
// return a random number in the range 0 to 999999999
//
int rand (void)
{
  int i = r[98];
  int j = r[99];
  int k;
  int t;

  if ((t = r[i] - r[j]) < 0) t += 1000000000L;

  r[i] = t;

  r[98]--; r[99]--;
  if (r[98] == 0) r[98] = 55;
  if (r[99] == 0] r[99] = 55;
```

```
  k = r[100] % 42 + 56;
  r[100] = r[k];

  r[k] = t;

  return(r[100]);
}


//
// seed the random number table
//
int init_rand (char *seed)
{
  char buf[101];
  int i, j, k;

  if (strlen(seed) > 85) return(0);
  sprintf(buf, "aEbFcGdHeI%s", seed);
  while (strlen(buf) < 98) strcat(buf, "Q");

  for (i = 1; i < 98; i++)
    r[i] = buf[i] * 8171717 + i * 997;

  i = 97; j = 12;
  for (k = 1; k < 998; k++) {
    r[i] -= r[j];
    if (r[i] < 0) r[i] += 1000000000;

    i--; j--;
    if (i == 0) i=97;
    if (j == 0) j=97;
  }

  r[98] = 55;
  r[99] = 24;
  r[100] = 77;
}


//
// return a random int between a and b
// assumes init_rand already called.
//
int rand_int(int a, int b)
{
  return (a + rand() % (b - a + 1));
```

```
}
```

References

1. Craig, John C. 1988, *Microsoft QuickBASIC Programmer's Toolbox* (Redmond, WA: Microsoft Press).

0.5.13 Horner's Rule

0.5.14 Chinese Remainder Theorem

0.5.15 Large Prime Number Generation

0.5.16 Fast Fourier Transform

0.6 Formal Language Parsing Algorithms

0.6.1 Recursive Descent Parsing

0.6.2 Cheatham Sattley Method

Cheatham Sattley is an easy to implement, tabular method of language parsing. It is a top-down system, like recursive descent, but it is a predictive system where, based on alternatives in syntax, the parser attempts to guess at what language element will come next.

Cheatham Sattley is good at parsing languages whose syntax can be stated in the following format:

$$< lhs >::=< e_{11} >< e_{12} > | < e_{21} >< e_{22} >$$

Of course some elements can be terminals or empty elements.

When told to match a $< lhs >$, Cheatham Sattley will initially predict that the first token presented should be a e_{11}. If this prediction fails, it will next try to parse the same initial token as a e_{21}. This process can continue n times until either the token is matched or no matches occur. In essence, Cheatham Sattley is a tree walking algorithm.

$$\begin{aligned} cS &\Longrightarrow BA|C \\ B &\Longrightarrow ab \\ A &\Longrightarrow a \\ C &\Longrightarrow c \end{aligned}$$

0.6.3 Samuelson-Bauer xpression analysis

Samuelson-Bauer is a technique for parsing expressions often used to convert infix style arithmetic expressions to a more computer friendly postfix notation. Samuelson-Bauer parsers are often used in the semantic analysis modules of modern compilers.

SB parsers operate by considering lexical tokens one at a time. If a token is an operand it is placed on the operand stack. If the token is an operator it causes the correct number of operands to be taken from their stack and placed on the output stack immediately followed by the operator.

Source Code

Many thanks to Rob McClinton of Virginia Tech and (now) Microsoft for allowing me to use this code. Rob, I, and three other students at Virginia Tech worked on a compiler for a web programming language called golo for credit in Fall of 1997. This code is in java.

```
import java.util.*;
import java.io.*;
import golo.codegen.*;
import golo.lex.LexLevel1;
import golo.common.*;
import golo.symtable.*;
import golo.syntax.SyntaxErrorException;

public class samBauer
{
   private symbolTable symtable;
   private CodeGenLevel1 codegen;
   private LexLevel1 lex;

   private boolean debug_sam = false;

   public samBauer(LexLevel1 _lex,symbolTable _symtable,CodeGenLevel1 _codegen)
   {
      symtable = _symtable;
      codegen = _codegen;
      lex = _lex;
   }

   /**
    * Reads tokens from Lex and parses them as a GOLO expression.
    * CodeGen is called to generate the code to evaluate the expression.
    * Semantic processing is done to coerce types as needed/possible and
    * the final type is returned as an integer.
    *
    * @return the final type of the expression: expObject.[REAL|INTEGER|BOOLEAN]
```

```
 */
public int readExp() throws SyntaxErrorException
{
    Stack prefix = new Stack();
    binNode head;
    int finalType;

    // parse into a reverse pre-order stack
    debugPrint("Input Expression:");
    RailRoad(prefix);
    debugPrintln("");

    if (debug_sam)
    {
       System.out.print("Result of railroad: ");
       printStack(prefix);
    }

    // build parse tree
    head = makeTree(prefix,(binNode)null);
    if (head==null)
    {
      System.err.println("makeTree returned null, what the hell??");
      Runtime.getRuntime().exit(1);
    }

    // add typing info to the parse tree and insert type coercions
    finalType = addTypes(head);
    // traverse the parse tree and tell codegen how to generate the exp
    generateCode(head);

    return finalType;
}

/**
 * Implements the samelson-bauer algorithm to parse the tokens read from
 * lex as a golo expression.  The result of the parse is pushed onto the
 * stack 'result', such that poping elements off result will provide a
 * reverse prefix notiation expression.
 *
 * @param result stack to put the parsed expression on
 */
private void RailRoad(Stack result) throws SyntaxErrorException
{
   Stack ops = new Stack();
   expObject temp;
   boolean expectBin = false, mismatch = false;
```

```
      int parin = 0;
      TokenLevel1 tok = lex.nextToken();

      while (!mismatch)
      {
         debugPrint(" "+tok.getString());
         if (expectBin)
         {
//       System.out.println("expecting bin");
            if (tok.getCode()==TokenLevel1.RPAR)
            {
//                System.out.println("got right paren");
               if (parin==0)
                  mismatch = true;
               else
               {
                  parin--;
                  expectBin = true;
               }
            }
            else
            if (!isBinOp(tok))
               mismatch = true;
            else
            {
               expectBin = false;
               temp = new expObject(tok,parin,true);
               while ( !ops.empty() &&
                          ( ((expObject)ops.peek()).rank() >= temp.rank()) )
                  result.push( ops.pop() );
               ops.push( temp );
            }
         }
         else
         {
//       System.out.println("expecting unary");
            if (isUnaryOp(tok))
            {
               temp = new expObject(tok,parin,false);
               while ( !ops.empty() &&
                          ( ((expObject)ops.peek()).rank() >= temp.rank()) )
                  result.push( ops.pop() );
               ops.push( temp );
               expectBin = false;
            }
            else
            if (isBinOp(tok))    // notice there is overlap between unary and
```

```
            mismatch = true;       //                           binary
         else
         if (isBuiltinFunc(tok))
         {
            temp = new expObject(tok,expObject.FUNCTION);
            while ( !ops.empty() &&
                       ( ((expObject)ops.peek()).rank() >= temp.rank()) )
               result.push( ops.pop() );
            ops.push( temp );
            // *** get arguments
            // for( int i=0; i<numArgs; i++)
            //    RailRoad(lex,result);
            expectBin = true;
         }
         else
         if (tok.getCode()==TokenLevel1.LPAR)
         {
//             System.out.println("got left paren");
            parin++;
         }
         else
         if (isConst(tok))
         {
             temp = new expObject(tok,expObject.CONST);
             result.push( temp );
             expectBin = true;
         }
         else
         if ( isVar( tok ) )
         {
             temp = new expObject(tok,expObject.VAR);
             result.push( temp );
             expectBin = true;
         }
         else
         if (isUserFunction(tok))
         {
            temp = new expObject(tok,expObject.FUNCTION);

     if (numParam(temp)>0)
     {
         // read LPAR from lex
         lex.advance();
         if (lex.nextToken().getCode()!=TokenLevel1.LPAR)
                   throw new SyntaxErrorException("Syntax Error line "+
                                lex.nextToken().getLineNo()+
                                ": Expecting '(' but got '"+
```

```
                              lex.nextToken().getString()+"'.");
        debugPrint(" (");
        lex.advance();
        // *** get arguments
        RailRoad(result);
        for( int i=1; i<numParam(temp); i++)
        {
                    // read comma separating params
                    if (lex.nextToken().getCode()!=TokenLevel1.COMMA)
                throw new SyntaxErrorException(
                                          "Syntax Error line "+
                                          lex.nextToken().getLineNo()+
                                          ": Expecting ',' but got '"+
                                           lex.nextToken().getString()+
                                           "'.");
                    debugPrint(" ,");
                    lex.advance();
                    // read next parameter
                    RailRoad(result);
        }
        // read RPAR from lex
        if (lex.nextToken().getCode()!=TokenLevel1.RPAR)
                  throw new SyntaxErrorException("Syntax Error line "+
                              lex.nextToken().getLineNo()+
                              ": Expecting ')' but got '"+
                               lex.nextToken().getString()+"'.");
        debugPrint(" )");
            }
            result.push(temp);
            expectBin = true;
         }
         else
         {
            mismatch = true;
            if (isUndefined(tok))
               throw new SyntaxErrorException(
                                        "undefined identifier: '"+
                                         tok.getString()+"' on line "+
                                         tok.getLineNo());
        }
      }
      if (!mismatch)
      {
         lex.advance();
         tok = lex.nextToken();
      }
   }
```

```
      debugPrint("X");
      // move the operators still on the op stack onto the result
      while ( !ops.empty() )
         result.push( ops.pop() );

      if (!expectBin || result.empty() )
          throw new SyntaxErrorException("incomplete expression on line "
                                 + tok.getLineNo());

            //throw syntax error since the expression is incomplete
      if (parin!=0)
         throw new SyntaxErrorException("unbalenced parentheses on line "
                                 + tok.getLineNo());
   }

    /**
     * Takes a reverse prefix expression from a stack and builds an equivalent
     * binary parse tree.
     * @param exp The stack which contains a the expression.
     * @param parent A reference to a node to use as the root of the parse tree
     */
    private binNode makeTree(Stack exp,binNode parent)
    {
       expObject temp;
       binNode ret=null,params;

       if (exp.empty()) return null;

       temp = (expObject)exp.pop();
       if ( temp.isOperand() )
       {
          // we have an operand, so this node of the tree can have no children
          ret = new binNode(temp,parent,null,null);
       }
       else
       {
          // we have some type of operator
          if ( temp.getType()==expObject.UNOP )
          {
             ret = new binNode(temp,parent,null,null);
             ret.setLeftChild(makeTree(exp,ret));
          }
          else if ( temp.getType()==expObject.BINOP )
          {
             ret = new binNode(temp,parent,null,null);
             ret.setLeftChild(makeTree(exp,ret));
             ret.setRightChild(makeTree(exp,ret));
```

```
        }
        else if ( temp.getType()==expObject.FUNCTION )
        {
           int numP = numParam(temp);
           Vector vect = new Vector(numP);

           vect.ensureCapacity(numP);
           for( int i=0; i<numP; i++)
              vect.addElement(null);

           ret = new binNode(temp,parent,null,null);
           // functions may need more than 2 children,
           // so store them in a Vector.
           ret.setLeftChild( new binNode( vect ) );
           for ( numP-- ; numP>=0 ; numP--)
              vect.setElementAt(makeTree(exp,ret),numP);
        }
     }
     return ret;
  }


/*
 * Adds semantic type information to our parse tree, so we know if
 * add means Real Add or Int Add.  This also inserts type conversion
 * operations when needed.
 *
 * @param head reference to the root node of a (sub)parse tree
 *
 * @return the final type of the expression: expObject.[REAL|INTEGER|BOOLEAN]
 */
private int addTypes(binNode head) throws SyntaxErrorException
{
   Vector params,realParams;
   int typeL,typeR,typeF;

   expObject obj = (expObject)head.getData();
   switch (obj.getType())
   {
      case expObject.CONST:
          switch(obj.getToken().getCode())
         {
            case TokenLevel1.INTTHING:
               obj.setStorageType(expObject.INTEGER);
               return expObject.INTEGER;
            case TokenLevel1.TRUE:
            case TokenLevel1.FALSE:
```

```
            obj.setStorageType(expObject.BOOLEAN);
            return expObject.BOOLEAN;
        }
        break; // not needed
    case expObject.VAR:
        try {
          switch( symtable.getKind(obj.getToken().getId()) )
          {
             case kinds.INT:
                obj.setStorageType(expObject.INTEGER);
                return expObject.INTEGER;
             case kinds.BOOL:
                obj.setStorageType(expObject.BOOLEAN);
                return expObject.BOOLEAN;
             case kinds.REAL:
                obj.setStorageType(expObject.REAL);
                return expObject.REAL;
             default:
                System.err.println("Fatal Internal Error in samBauer.\n");
                System.err.print("An expObject claimed to be a variable, ");
                System.err.println("but 'kind' doesn't match.\n");
                Runtime.getRuntime().exit(1);
          }
        }
        catch (stException e) {
          System.err.println("systable error: " + e.getMessage());
          e.printStackTrace();
          System.err.println("Fatal Internal Error in samBauer.\n");
          System.err.print("An expObject claimed to be a variable, ");
          System.err.println("but symTable didn't agree.\n");
          Runtime.getRuntime().exit(1);
        }
    case expObject.UNOP:
        if ( expObject.BOOLEAN == (typeL=addTypes( head.getLeftChild() )) )
        {
           if (!isBoolOp(obj))
              semanticError("Invalid type for operand to operator "
                            + obj.getToken().getString() );
           obj.setStorageType(expObject.BOOLEAN);
           return expObject.BOOLEAN;
        }
        else  // the operand is a real or integer
        {
           if (isBoolOp(obj))
              semanticError("Invalid type for operand to operator "
                            + obj.getToken().getString() );
           obj.setStorageType(typeL);
```

```
            return typeL;
         }
      case expObject.BINOP:
         typeL = addTypes( head.getLeftChild() );
         typeR = addTypes( head.getRightChild() );
         typeF = balenceTypes(typeL,typeR);
         if ( typeL == expObject.BOOLEAN )
         {
            if (!isBoolOp(obj))
               semanticError("Invalid type for operand to operator "
                                    + obj.getToken().getString() );
            obj.setStorageType(expObject.BOOLEAN);
            return expObject.BOOLEAN;
         }
         else  // the operand is a real or integer
         {
            if (isBoolOp(obj))
            {
               semanticError("Invalid type for operand to operator "
                                    + obj.getToken().getString() );
               System.err.println(typeL+" "+typeR);
            }
            else
            {
               if ( typeF != typeL )
               {
                  // insert type conversion operator on left
                  semanticError("SamB Warning: forgot type conversion\n");
               }
               if ( typeF != typeR )
               {
                  // insert type conversion operator on right
                  semanticError("SamB Warning: forgot type conversion\n");
               }
            }
            obj.setStorageType(typeF);
            if (isRelOp(obj))
               return expObject.BOOLEAN;
            else
               return typeF;
         }
      case expObject.FUNCTION:
         try {
         // run addtypes for each parameter and check with def
         params = (Vector)head.getLeftChild().getData();
   realParams = (Vector)symtable.getParamterList(
                              obj.getToken().getId() );
```

```
  for ( int i=0; i<params.size(); i++ )
          {
             typeL = addTypes( (binNode)params.elementAt(i) );
     switch (((param)realParams.elementAt(i+1)).type())
     {
        case kinds.BOOLEAN:
   if (typeL!=expObject.BOOLEAN)
      semanticError("Type mismatch for argument "+(i+1)
  +" to function "+obj.getToken().getString()
  +" on line "+obj.getToken().getLineNo());
   break;
case kinds.INTEGER:
   if (typeL==expObject.BOOLEAN)
      semanticError("Type mismatch for argument "+(i+1)
  +" to function "+obj.getToken().getString()
  +" on line "+obj.getToken().getLineNo());
           else if (typeL==expObject.REAL)
                   {
                      // insert a conversion to integer
                      semanticError("SamB Warning: forgot type conversion\n");
                   }
                   break;
                case kinds.REAL:
                   if (typeL==expObject.BOOLEAN)
      semanticError("Type mismatch for argument "+(i+1)
  +" to function "+obj.getToken().getString()
  +" on line "+obj.getToken().getLineNo());
                   else if (typeL==expObject.INTEGER)
                   {
                      // insert a conversion to real
                       semanticError("SamB Warning: forgot type conversion\n");
                   }
       }
          }
          // lookup return type and stow it
          if (realParams.elementAt(0)==null)
             throw new SyntaxErrorException(
                               "Error - Non-Function in an expression\n");
          if (!(realParams.elementAt(0) instanceof param))
     timeToDie("First Element in Function param list is not a param or null");
  switch ( ((param)realParams.elementAt(0)).type() )
          {
             case kinds.INTEGER:
                obj.setStorageType( expObject.INTEGER );
                return expObject.INTEGER;
             case kinds.REAL:
                obj.setStorageType( expObject.REAL );
```

```
                return expObject.REAL;
             case kinds.BOOLEAN:
                obj.setStorageType( expObject.BOOLEAN );
                return expObject.BOOLEAN;
          }

          } catch ( stException e ) {
              System.err.println("Internal error: " + e.getMessage());
              e.printStackTrace();
              Runtime.getRuntime().exit(1);
          }
    }
    System.err.println("Internal error: samBauer.addTypes() (hit bottom)");
    Runtime.getRuntime().exit(1);
    return 0;  // see java suck. suck, java, suck!
 }


 /**
  * Generates instructions to codeGen, which if performed will result
  * in the result of the GOLO expression being placed on the top of the
  * stack.  The instructions are given via a forward post-order traversal
  * of the parse tree.
  * @param head The root node of the parse tree for the expression.
  */
 private void generateCode( binNode head )
 {
    Vector params;
    expObject obj = (expObject)head.getData();

    // if this node is a function, the children are arrainged differently
    if ( obj.getType() == expObject.FUNCTION )
    {
       // put the function's children on the stack from right to left
       params = (Vector) head.getLeftChild().getData();
       for( int i = params.size()-1; i>=0; i-- )
          generateCode( (binNode)params.elementAt(i) );
       // now call the function
       codegen.callFunction( obj.getToken().getId() );
return;
    }

    // generate code for any possible operands
    if (head.getLeftChild()!=null)
       generateCode( head.getLeftChild() );
    if (head.getRightChild()!=null)
       generateCode( head.getRightChild() );
```

```
      // generate code for this node
      switch ( obj.getType() )
      {
         case expObject.VAR:
            codegen.pushVariable( obj.getToken().getID() );
            break;
         case expObject.CONST:
            switch ( obj.getToken().getCode() )
            {
               case TokenLevel1.TRUE:
                  codegen.pushBool(true);
                  break;
               case TokenLevel1.FALSE:
                  codegen.pushBool(false);
                  break;
               case TokenLevel1.INTTHING:
                  codegen.pushInt( obj.getToken().intValue() );
                  break;
//                case TokenLevel1.REAL:
//                   codegen.pushReal( Float.valueOf(
//                                              obj.getToken().getString() ) );
//                   break;
            }
            break;
         case expObject.BINOP:
            switch ( obj.getStorageType() )
            {
               case expObject.REAL:
                   switch ( obj.getToken().getCode() )
                   {
                      case TokenLevel1.PLUSOP:
                         codegen.floatAdd();
                         break;
                      case TokenLevel1.MINUSOP:
                         codegen.floatSub();
                         break;
                      case TokenLevel1.TIMESOP:
                         codegen.floatMult();
                         break;
                      case TokenLevel1.DIVOP:
                         codegen.floatDiv();
                          break;
                      case TokenLevel1.EQUALTO    :
                         codegen.floatEqual();
                          break;
                      case TokenLevel1.LESSTHAN   :
```

```
            codegen.floatLess();
             break;
         case TokenLevel1.GREATERTHAN:
            codegen.floatGreater();
             break;
         case TokenLevel1.LEQ        :
            codegen.floatNotGreater();
             break;
         case TokenLevel1.GEQ        :
            codegen.floatNotLess();
             break;
     }
     break;
  case expObject.INTEGER:
      switch ( obj.getToken().getCode() )
      {
         case TokenLevel1.PLUSOP:
            codegen.intAdd();
            break;
         case TokenLevel1.MINUSOP:
            codegen.intSub();
            break;
         case TokenLevel1.TIMESOP:
            codegen.intMult();
            break;
         case TokenLevel1.DIVOP:
            codegen.intDiv();
             break;
         case TokenLevel1.EQUALTO    :
            codegen.intEqual();
             break;
         case TokenLevel1.LESSTHAN   :
            codegen.intLess();
             break;
         case TokenLevel1.GREATERTHAN:
            codegen.intGreater();
             break;
         case TokenLevel1.LEQ        :
            codegen.intNotGreater();
             break;
         case TokenLevel1.GEQ        :
            codegen.intNotLess();
             break;
      }
     break;
  case expObject.BOOLEAN:
      switch ( obj.getToken().getCode() )
```

```
                {
                   case TokenLevel1.AND:
                      codegen.logicalAnd();
                      break;
                   case TokenLevel1.OR:
                      codegen.logicalOr();
                      break;
                }
                break;
          }
          break;
       case expObject.UNOP:
          switch ( obj.getStorageType() )
          {
             case expObject.REAL:
                codegen.floatNegate();
                break;
             case expObject.INTEGER:
                codegen.intNegate();
                break;
             case expObject.BOOLEAN:
                codegen.logicalNot();
                break;
          }
          break;
    }
 }


 /**
  * Used for testing purposes.  Use the java interpreter to run
  * this.  Pass the program 1 argument, which is the name of a file
  * which contains the expression(s) to be tested.
  */
 public static void main(String args[])
 {
    try {
    symbolTable sym = new symbolTable();
    LexLevel1 lex = new LexLevel1(args.length>0?args[0]:"testexp.in",sym);
    CodeGenLevel1 cg = new CodeGenLevel1(sym);
    samBauer sb = new samBauer(lex,sym,cg);
//      semRecord sr;

    // define a variable to play with
    ID id = sym.createEntry("intVar");
    sym.setKind(id,kinds.INTEGER);
    cg.newvar(id);
```

```
      id = sym.createEntry("boolVar");
      sym.setKind(id,kinds.BOOLEAN);
      cg.newvar(id);

      sb.debug_sam = false;

      sb.readExp();

      if (lex.nextToken().getCode()!=TokenLevel1.SCANEOF)
      {
         System.out.println("Extra data after expression: '"+
             lex.nextToken().getString()+"'");
      }

      }
      catch (FileNotFoundException e)
      {
         System.err.println("Couldn't open file exptest.in");
      }
      catch (stException e)
      {
         e.printStackTrace();
      }
      catch (SyntaxErrorException e)
      {
         System.err.println("Got syntax error");
         System.err.println(e.getMessage());
 e.printStackTrace();
      }
   }


   // ************************************************************
   // ************************************************************
   // *******          Utility  Functions   *********************
   // ************************************************************

   /**
    * prints an error message.  I don't know why I wanted to encapsulate this
    * functionality in a method. just felt like it.
    */
   private void semanticError(String msg)
   {
      Exception e = new Exception(msg);
//       System.err.println(msg);
      e.printStackTrace();
   }
```

```
private void debugPrint(String str)
{
   if (debug_sam)
      System.out.print(str);
}
private void debugPrintln(String str)
{
   if (debug_sam)
      System.out.println(str);
}

private static int balenceTypes(int left, int right)
                                        throws SyntaxErrorException
{
   if (left==right)
      return left; //easy out

   if ( (left==expObject.BOOLEAN) || (right==expObject.BOOLEAN) )
      throw new SyntaxErrorException();

   return expObject.REAL; // it is the only remaining choise,  honest.
}

private static boolean isBoolOp( expObject obj )
{
   switch (obj.getToken().getCode())
   {
      case TokenLevel1.OR:
      case TokenLevel1.AND:
      case TokenLevel1.NOT:
         return true;
      default:
         return false;
   }
}

private static boolean isRelOp( expObject obj )
{
   switch (obj.getToken().getCode())
   {
      case TokenLevel1.GREATERTHAN:
      case TokenLevel1.LESSTHAN:
      case TokenLevel1.EQUALTO:
      case TokenLevel1.LEQ:
      case TokenLevel1.GEQ:
         return true;
```

```
      default:
         return false;
   }
}

private boolean isBinOp(TokenLevel1 tok)
{
   switch (tok.getCode())
   {
      case TokenLevel1.PLUSOP:
      case TokenLevel1.MINUSOP:
      case TokenLevel1.TIMESOP:
      case TokenLevel1.DIVOP:
      case TokenLevel1.OR:
      case TokenLevel1.AND:
      case TokenLevel1.EQUALTO:
      case TokenLevel1.LESSTHAN:
      case TokenLevel1.GREATERTHAN:
      case TokenLevel1.LEQ:
      case TokenLevel1.GEQ:
          return true;
      default:
          return false;
   }
}

private boolean isUnaryOp(TokenLevel1 tok)
{
   switch (tok.getCode())
   {
      case TokenLevel1.MINUSOP:
      case TokenLevel1.NOT:
          return true;
      default:
          return false;
   }
}

private boolean isBuiltinFunc(TokenLevel1 tok)
{
   switch (tok.getCode())
   {
      case TokenLevel1.ROUND:
      case TokenLevel1.FLOOR:
      case TokenLevel1.CEILING:
      case TokenLevel1.XCORQM:
      case TokenLevel1.YCORQM:
```

```
        case TokenLevel1.HEADINGQM:
        case TokenLevel1.PENCOLORQM:
        case TokenLevel1.PENSIZEQM:
        case TokenLevel1.PENDOWNQM:
        case TokenLevel1.SHOWNQM:
            System.err.println("Lex didn't leave a builtin function as an identifier\n");
            System.exit(1);
        default:
            return false;
    }
}

private boolean isIdent(TokenLevel1 tok)
{
    if (tok.getCode()==TokenLevel1.IDENTIFIER)
        return true;
    else
        return false;
}

private boolean isConst(TokenLevel1 tok)
{
    switch (tok.getCode())
    {
        case TokenLevel1.INTTHING:
        case TokenLevel1.TRUE:
        case TokenLevel1.FALSE:
            return true;
        default:
            return false;
    }
}

private boolean isVar(TokenLevel1 tok)
{
    try {
    if ( isIdent(tok) &&
            kinds.isVar(symtable.getKind(tok.getID())) )
         return true;
    else
         return false;
    }
    catch (stException e)
    {
         System.err.println("Internal error: " + e.getMessage());
         e.printStackTrace();
         Runtime.getRuntime().exit(1);
```

```
    }
    return false; // never reached
  }

   private int numParam(expObject obj)
   {
       if (obj.getType()==expObject.UNOP)
          return 1;
       if (obj.getType()!=expObject.FUNCTION)
          return 0;
       else
       {
           try {
             if (obj.getToken().getId()==null)
             {
                Exception e = new Exception("Bad token.  getId()==null.");
                e.printStackTrace();
                System.exit(1);
             }
             return
        symtable.getParamterList(obj.getToken().getId()).size()-1;
           }
           catch (stException e) {
             System.err.println("Fatal Internal Error in samBauer.\n");
             System.err.print("An expObject claimed to be a function, ");
             System.err.println("but symTable didn't agree.\n");
             Runtime.getRuntime().exit(1);
             return 0;  // java is retarded
           }
       }
   }

  private boolean isUndefined(TokenLevel1 tok)
  {
    try {
    if ( (tok.getCode()==TokenLevel1.IDENTIFIER) &&
          (symtable.getKind(tok.getID())==kinds.UNKNOWN) )
        return true;
    else
        return false;
    }
    catch (stException e)
    {
        System.err.println("Internal error: " + e.getMessage());
        e.printStackTrace();
        Runtime.getRuntime().exit(1);
    }
```

```
      return false; // never reached
   }

   private boolean isUserFunction(TokenLevel1 tok)
   {
      try {
      if ( (tok.getCode()==TokenLevel1.IDENTIFIER) &&
              (symtable.getKind(tok.getID()) == kinds.FUNCTION) )
          return true;
      else
          return false;
      }
      catch (stException e)
      {
          System.err.println("Internal error: " + e.getMessage());
          e.printStackTrace();
          Runtime.getRuntime().exit(1);
      }
      return false; // never reached
   }

   /**
    * Prints a stack of expObjects.  It is printed from top to bottom
    * outputing the related string of the token for each expObject.
    * The input stack is returned in the same condition.
    */
   private void printStack(Stack exp)
   {
      expObject temp;
      Stack save = new Stack();

      while( !exp.empty() )
      {
         temp = (expObject)exp.pop();
         System.out.print(temp.getToken().getString()+' ');
         save.push( temp );
      }
      System.out.println();
      while( !save.empty() )
         exp.push( save.pop() );
   }

   private void timeToDie(String str)
   {
      System.err.println(str);
      System.exit(1);
   }
```

```
}
```

0.6.4 Shift-reduce bottom-up parsing

0.7 Operating Systems Algorithms

0.7.1 Dijkstra's Banker's Algorithm for Deadlock Prevention

The Banker's Algorithm is a strategy for **deadlock** prevention. In an operating system, deadlock is a state in which two or more processes are "stuck" in a **circular wait** state. All deadlocked processes are waiting for resources held by other processes. Because most systems are **non-preemptive** (that is, will not take resources held by a process away from it), and employ a **hold and wait** method for dealing with system resources (that is, once a process gets a certain resource it will not give it up voluntarily), deadlock is a dangerous state that can cause poor system performance.

One reason this algorithm is not widely used in the real world is because to use it the operating system must know the maximum amount of resources that every process is going to need at all times. Therefore, for example, a just-executed program must declare up-front that it will be needing no more than, say, 400K of memory. The operating system would then store the limit of 400K and use it in the deadlock avoidance calculations.

The Banker's Algorithm seeks to prevent deadlock by becoming involved in the granting or denying of system resources. Each time that a process needs a particular **non-sharable** resource, the request must be approved by the banker.

The banker is a conservative loaner. Every time that a process makes a request of for a resource to be "loaned" the banker takes a careful look at the bank books and attempts to determine whether or not a deadlock state could possibly arise in the future if the loan request is approved.

This determination is made by "pretending" to grant the request and then looking at the resulting post-granted request system state. After granting a resource request there will be an amount of that resource left free in the system, f. Further, there may be other processes in system. We demanded that each of these other processes state the maximum amount of all system resources they needed to terminate up-front so, therefore, we know how much of each resource every process **is holding** and **has claim to**.

If the banker has enough free resource to guarantee that even one process can terminate, it can then take the resource held by that process and add it to the free pool. At this point the banker can look at the (hopefully) now larger free pool and attempt to guarantee that another process will terminate by checking whether its **claim** can be met. If the banker can guarantee that all jobs in system will terminate, it approves the loan in question.

If, on the other hand, at any point in the reduction the banker cannot guarantee any processes will terminate because there is not enough free resource to meet the smallest claim, a state of deadlock can ensue. This is called an **unsafe state**. In this case the loan request in question is denied and the requesting process is usually **blocked**.

The efficiency of the Banker's algorithm depends greatly on how it is implemented. For example, if the bank books are kept sorted by process claim size, adding new process information to the table is $O(n)$ but reducing the table is simplified. However if the table is kept in no order, adding a new entry is $O(1)$ however reducing the table is less efficient.

Source Code

The following code is taken from one of my OS projects and is incomplete in that it makes calls to a system module that is not included. The queue module which is called heavily is, however, included with this implementation:

```
/* --- banker.h --- */

/*-----------------------------------------------------------------------------*\

$Id: banker.h,v 1.4 1997/02/20 17:56:20 scott Exp scott $

$Log: banker.h,v $
Revision 1.4  1997/02/20 17:56:20  scott
Final version.

Revision 1.3  1997/02/20 00:21:04  scott
Total rewrite...

Revision 1.2  1997/02/19 18:58:44  scott
This may still be buggy.

Revision 1.1  1997/02/18 19:18:59  scott
Initial revision

\*-----------------------------------------------------------------------------*/

#ifndef __BANKER__
#define __BANKER__

bool bank_approves(int pid, int current, int deltam, int max, int freenow);

#endif



/* --- banker.c --- */

/****************************************************************************\
*                                                                            *
*     module: banker                                                         *
```

```
*    program: simulate (OS assignment #1, Spring 1997)                     *
* programmer: Scott Gasch                                                  *
*      descr: This is my implementation of Dijkstra's banker's algorithm.  *
*             It is called by the system module whenever a process makes   *
*             a memory request.  When called, this module builds a table   *
*             with information about the pid, claim and hold of all jobs   *
*             in the the run and ready states.  It then reduces the table  *
*             to see whether granting the memory request would leave the   *
*             system in an unsafe state.                                   *
*                                                                          *
* $Id: banker.c,v 1.5 1997/02/20 17:53:46 scott Exp scott $                *
*                                                                          *
* $Log: banker.c,v $                                                       *
* Revision 1.5  1997/02/20 17:53:46  scott                                 *
* Final version -- added comments and changed the handling of the "banker" *
* queue... I leave it open clearing the contents between calls.            *
*                                                                          *
* Revision 1.4  1997/02/20 00:21:04  scott                                 *
* Total rewrite... this version actually seems to work.                    *
*                                                                          *
* Revision 1.3  1997/02/19 18:58:44  scott                                 *
* This may still be buggy...                                               *
*                                                                          *
* Revision 1.2  1997/02/18 21:26:53  scott                                 *
* This still isn't working... damn thing                                   *
*                                                                          *
* Revision 1.1  1997/02/18 19:18:59  scott                                 *
* Initial revision                                                         *
*                                                                          *
\**************************************************************************/

#include "global.h"
#include "banker.h"
#include "system.h"
#include "queue.h"
#include "errors.h"

/*------------------------------------------------------------------------*/

                           /* Data Structures */

/*
 *  a "line" in the internal table ("logbook") we build and reduce.
 *
 */

struct banker_entry {
```

```
  int pid;                                          /* process ID number */
  int hold;                               /* amount of memory currently held */
  int claim;                          /* max - current (i.e. potential claim) */
  int max;                            /* the max memory this job can allocate */
};


/*
 *  This is a duplicate of the internal pcb structure used by the system
 *  module.  This is kinda cheating... I should really provide interface
 *  functions in system.c to allow access to this information...
 *
 *  Think of banker as a "friend" of system... friends can play with e/o's
 *  private parts.
 *
 */

struct pcb {
  int pid;                                          /* process ID number */
  char name[MAX_JOB_NAME_LEN];                     /* the name of the job */
  int arrival_time;                /* when did it first enter the system? */
  int finish_time;              /* when did it finish execution (or zero) */
  int cpu_needed;                /* CPU time this process needs to finish */
  int cpu_so_far;                /* amount of CPU time it has seen so far */
  int current_memory;                      /* how much memory is it using */
  int initial_memory;                /* how much memory did it start with */
  int max_memory;      /* what's the maximum amount of memory it can have */
  struct mevent *memory_events;                     /* any memory events */
  int state;                                             /* where is it? */
  int priority;    /* how important (see scale)  (VIP  1 2 3 ... X  SCUM) */
};

/*-----------------------------------------------------------------------*/

                   /* internal "helper" routine prototypes*/

/*
 *  These are what those stupid C++ programmers call "private methods"
 *  See comments about each one's purpose at the actual implementation.
 *
 */

static bool reduce_logbook(char *name, int free);
static void clean_logbook(char *name);
static void build_logbook(char *logbook);

/*-----------------------------------------------------------------------*/
```

```
/*
 *  build_logbook - (internal) create a banker's table
 *
 *  parameters:
 *    char *logbook - the name of the queue in which to create table
 *
 *  purpose:
 *    this function sweeps through the ready and run system states and
 *    adds an entry for each process it encounters, thus creating the
 *    internal banker's table.
 *
 *  returns:
 *    nothing
 *
 */

static void build_logbook(char *logbook) {
  struct pcb *job;                        /* space for pcbs of jobs examined */
  struct banker_entry *new_line;          /* one entry in the banker table */
  int count, place;            /* how many jobs in rdy queue, loop control */

  /* run down the ready queue and process each item... */
  count = qcount("ready");
  for (place = 1; place <= count; place++) {
    if (job = (struct pcb *) peekq("ready", place)) {

      /* create a new logbook entry */
      if (new_line = (struct banker_entry *) malloc (sizeof(struct banker_entry))) {
        new_line->pid = job->pid;
        new_line->hold = job->current_memory;
        new_line->claim = job->max_memory - job->current_memory;
        new_line->max = job->max_memory;
        enqIO((void *) new_line, new_line->claim, logbook);
      } else {
        error_handler(OUT_OF_MEMORY, 1);
        /* die */
      }
    }
  }

  /* check the run state also */
  if (job = (struct pcb *) peekq("run", 1)) {

    /* create a new logbook entry */
    if (new_line = (struct banker_entry *) malloc (sizeof(struct banker_entry))) {
      new_line->pid = job->pid;
```

```
      new_line->hold = job->current_memory;
      new_line->claim = job->max_memory - job->current_memory;
      new_line->max = job->max_memory;
      enqIO((void *) new_line, new_line->claim, logbook);
    } else {
      error_handler(OUT_OF_MEMORY, 1);
      /* die */
    }
  }
}


/*----------------------------------------------------------------------------*/

/*
 *  reduce_logbook - (internal) reduce a banker table to determine safety
 *
 *  parameters:
 *    char *name - the name of the queue in which the logbook resides
 *    int freemem - the amount of memory we have to work with
 *
 *  purpose:
 *    this routine reduces the banker table in order to determine whether
 *    or not the system it represents is in a safe state or not.  It does
 *    this by recursively calling itself until the banker table is either
 *    empty or it cannot satisfy the lowest claim.
 *
 *  returns:
 *    YES - the logbook was reduced and the system it represents is safe.
 *    NO - otherwise
 *
 */

static bool reduce_logbook(char *name, int freemem) {
  struct banker_entry *min_claim;     /* table entry with the smallest claim */

  /* if we've reduced the bank book to empty, it's safe */
  if (qcount(name) == 0) return(YES);

  else {

    /* otherwise let's see if we can satisfy the process with smallest claim */
    min_claim = (struct banker_entry *) deqmin(name);

    /* if ever we can't satisfy at least the smallest claim, we're in trouble */
    if (min_claim->claim > freemem) {
      free(min_claim);
```

```
      return (NO);
    }

    /* however if we can satisfy it, we can take all of its memory and continue */
    freemem += min_claim->hold;
    free(min_claim);
    return(reduce_logbook(name, freemem));
  }
}

/*----------------------------------------------------------------------------*/

/*
 *  clean_logbook - (internal) deallocate all entries in the banker table
 *
 *  parameters:
 *    char *name - the name of the queue in which the table resides.
 *
 *  purpose:
 *    When the banker's algorithm has completed this internal routine is
 *    called in order to clean up the memory we used and the mess we made.
 *    It is important that this gets called between calls to bank_approves
 *    or else it will cause banker table corruption.
 *
 *  returns:
 *    nothing
 *
 */


static void clean_logbook(char *name) {

  int count = qcount(name);               /* how many things in the table? */
  int place;                                               /* loop control */
  struct banker_entry *nukeme;        /* pointer to an entry to deallocate */

  /* run down the queue and free up the space we used */
  while (nukeme = (struct banker_entry *) deqmin(name)) {
    free(nukeme);
  }

}

/*----------------------------------------------------------------------------*/

/*
 *  bank_approves - (interface) does the bank approve a memory transaction?
```

```
 *
 *  parameters:
 *    int pid - the PID number of the process requesting memory
 *    int deltam - the amount of memory in the transaction
 *    int current - the amount of memory this process currently holds
 *    int max - the max amount of memory this process can allocate legally
 *    int freenow - the amount of system memory free at time of call
 *
 *  purpose:
 *    This is the interface the banker module; it is called from system
 *    whenever a process make a memory request.  It's job is to consider
 *    the system state after granting that memory request and determine
 *    if said state is safe [i.e. cannot lead to deadlock] or not [i.e.
 *    can lead to dealock].
 *
 *  returns:
 *    YES - if it is safe to grant the request
 *    NO - if granting said request could lead to deadlock
 *
 */

bool bank_approves (int pid, int current, int deltam, int max, int freenow) {
  struct banker_entry *new_line;

  /* if there's just not enough memory, say no (this is a redundant check) */
  if (freenow - deltam < 0) return(NO);

  /* however, always take memory back! */
  if (deltam < 1) return(YES);


  /* ok here we are... build the logbook (banker table) and then reduce it */
  build_logbook("banker");

  /*
   *  make sure we remember to include the requesting job... note, this
   *  assumes said job is *NOT* in ready or run.  The system module adheres
   *  to this little practice by removing the running process from the run
   *  queue before calling banker.  Bear this restriction in mind if you
   *  add calls to this function.
   *
   */

  if (new_line = (struct banker_entry *) malloc (sizeof(struct banker_entry))) {
    new_line->pid = pid;
    new_line->hold = current + deltam;
    new_line->claim = (max - current) - deltam;
```

```
    new_line->max = max;
    enqIO((void *) new_line, new_line->claim, "banker");
  }


  /* now, can we reduce it? */
  if (reduce_logbook("banker", freenow - deltam)) {
    clean_logbook("banker");
    return(YES);
  }
  clean_logbook("banker");
  return(NO);
}



/* --- queue.h (queue routines required by banker) */
/*----------------------------------------------------------------------------*\

$Id: queue.h,v 1.3 1997/02/19 18:58:44 scott Exp $

$Log: queue.h,v $
Revision 1.3  1997/02/19 18:58:44  scott
Made two functions which had returned int return void.

Revision 1.2  1997/02/15 17:45:36  scott
Added grep_deq.

Revision 1.1  1997/02/15 00:17:51  scott
Initial revision


\*----------------------------------------------------------------------------*/

#ifndef __QUEUE__
#define __QUEUE__

/* this is the max size in characters that a "tag" (name) of a queue may be */

#define MAX_QUEUE_TAG_LENGTH 22


/* interface definition */

void init_queue_module(void);
void close_queue_module(void);
```

```
/* initialize or delete a queue */
void initQ(char *name);
void delQ (char *name);

/* enq/deq FIFO */
void enqFIFO(void *data, char *name);
void *deqFIFO(char *name);

/* enq/deq inorder */
void enqIO(void *data, int key, char *name);
void *deqmin(char *name);
void *deqmax(char *name);
void *deqkey(char *name, int key);
void *peekkey(char *name, int key);

/* informational functions about queues */
int qcount(char *name);
int qempty(char *name);

/* cheats */
void *peekq (char *name, int element);
void showq(char *name);
void *grep_deq (void *target, char *name);

#endif
```

```
/* --- queue.c (queue routines required by banker) */

/****************************************************************************\
*                                                                            *
*     module: queue                                                          *
*    program: simulate (OS Assignment #1, Spring 1997)                       *
* programmer: Scott Gasch                                                    *
*      descr: This is a generic queue module.  It maintains its queues by    *
*             keeping a linked list of index records.  Each record has a     *
*             qname field and pointers to the head and tail of the attached *
*             queue.  Queues hold void pointers and, optionally, integer     *
*             keys.  So you can store anything you want on these queues,     *
*             in any order you compute, but you have to keep track of what   *
*             you put on here to cast it to the right struct when dequeue-   *
*             ing.                                                           *
*                                                                            *
* $Id: queue.c,v 1.6 1997/02/20 00:21:04 scott Exp scott $                   *
*                                                                            *
* $Log: queue.c,v $                                                          *
* Revision 1.6  1997/02/20 00:21:04  scott                                   *
```

```
* OK... hopefully these are the last bugs... In initQ the check for        *
* colliding names was screwing the master index all up so that when       *
* the closeQmodule funct got called it entered an endless loop.  This     *
* is fixed.  Also changed enqIO slightly and added more error checking.   *
*                                                                         *
* Revision 1.5  1997/02/19 18:58:44  scott                                *
* Fixed some problems with the in-order (sorted) queues...                *
*                                                                         *
* Revision 1.4  1997/02/18 21:26:53  scott                                *
* Fixed some problems and implemented priority queueing with IO queues.   *
*                                                                         *
* Revision 1.3  1997/02/18 19:18:59  scott                                *
* Modified the IO queue code... it may still be buggy.                    *
*                                                                         *
* Revision 1.2  1997/02/15 17:45:36  scott                                *
* Added a "grep_deq" function to support kills of processes anywhere in a *
* queue.                                                                  *
*                                                                         *
* Revision 1.1  1997/02/15 00:17:51  scott                                *
* Initial revision                                                        *
*                                                                         *
\*************************************************************************/

#include <stdlib.h>
#include <string.h>
#include "global.h"
#include "queue.h"
#include "errors.h"

/* for some reason ultrix does not have a prototype for this in string.h */
#ifdef ultrix
int strcasecmp(char *s1, char *s2);
#endif

/*\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

                          /* Data Structures */


struct queue_element {
  void *data;                                 /* pointer to item on queue */
  int key;                                   /* it's [optional] key value */
  struct queue_element *next;        /* pointer to the next thing on queue */
};


/* this is an index entry... total administrative stuff. */
```

```
struct index_element {
  char qname[MAX_QUEUE_TAG_LENGTH];                   /* name of this queue */
  struct queue_element *head;                        /* pointer to the head */
  struct queue_element *tail;                        /* pointer to the tail */
  int count;                                /* number of items on the queue */
  struct index_element *next;                       /* the next index entry */
};

/*\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

                    /* Internal helper function prototypes */

static int num_queues;
static struct index_element *the_index;
static void zapqueue(struct queue_element *head);
static void seek(char *name, struct index_element **found,
                 struct index_element **prefound);

/*\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

void init_queue_module(void) {
  the_index = NULL;                          /* there is nothing on the index */
  num_queues = 0;                        /* there are no queues in the system */
}

/*\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

/*
 *  close_queue_module - (interface) close down index and queues.
 *
 *  parameters:
 *    none
 *
 *  purpose:
 *    This is called to cleanup the queue module before program termination.
 *    It basically deallocates one queue at a time and one queue index entry
 *    at a time.
 *
 *  returns:
 *    nothing
 *
 */

void close_queue_module(void) {
  struct index_element *current;                 /* the current queue index */
  struct index_element *zapme;      /* a queue index item to be deallocated */
```

```
  /* if there are queues... clean them up */
  if (num_queues) {

    current = the_index;

    while (current != NULL) {

      zapme = current;
      current = current->next;
      #ifdef DEBUG
        printf("CLOSE %s\n", zapme->qname);
      #endif
      zapqueue(zapme->head);
      free(zapme);

    }
    num_queues = 0;
    the_index = NULL;

  }

}

/*\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

/*
 *  initQ - (interface) Initialize a queue (create an index entry) for use.
 *
 *  parameters:
 *    char *name - the "tag name" which will be used to identify this queue
 *
 *  purpose:
 *    this function initializes an empty queue by creating an index entry
 *    for this queue.  After calls to initQ user functions can use enq/deq...
 *
 *  returns:
 *    nothing
 *
 *  restrictions:
 *    does not warn if the tag specified in name already exists, simply
 *    deletes the old queue and builds a new one.
 *
 */

void initQ(char *name) {
  struct index_element *new_index_entry;          /* new index for new queue */
  struct index_element *a, *b;                /* junko index entries used to */
```

```
                                                      /* see if the tag is in use */
  /* check too long */
  if (strlen(name) > MAX_QUEUE_TAG_LENGTH) {

    error_handler(TAG_TOO_LONG, 0);
    return;

  }

  if (num_queues) {

    /* check colliding names */
    seek(name, &a, &b);
    if (a) delQ(a->qname);

  }

  /* well, allocate space and do it -- here we allocate an index for new queue */
  if ((new_index_entry = (struct index_element *)
      malloc(sizeof(struct index_element)))) {

      strcpy(new_index_entry->qname, name);
      new_index_entry->count = 0;
      new_index_entry->head = NULL;
      new_index_entry->tail = NULL;
      new_index_entry->next = the_index;
      the_index = new_index_entry;
      num_queues++;
      return;

  /* problem allocating more memory?  ack! */
  } else {

      error_handler(OUT_OF_MEMORY, 1);
      /* die */

  }
}

/*\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

/*
 *  delQ - (interface) delete a queue by removing it's index entry and contents
 *
 *  parameters:
 *    char *name - the "tag" identity of the queue to delete.
 *
```

```
 *  purpose:
 *    This is the opposite of initQ; it takes an existing queue and, by
 *    deallocating its index entry and contents, makes it unavailable.
 *
 *  returns:
 *    nothing
 *
 */

void delQ (char *name) {
  struct index_element *current_entry;      /* pointer to name's index entry */
  struct index_element *prev_entry;  /* pointer to the index entry before it */

  seek (name, &current_entry, &prev_entry);

  /* is it valid? */
  if (current_entry != NULL) {

      /* first free all the memory on the queue */
      zapqueue(current_entry->head);

      /* now delete its entry from the index */
      if (current_entry == the_index) {
        the_index = current_entry->next;
      }
      if (prev_entry) prev_entry->next = current_entry->next;
      free(current_entry);
      num_queues--;

  }

}

/*\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

/*
 *  seek - (internal) given a "tag" name, return its index entry
 *
 *  parameters:
 *    char *name - the tag name
 *    struct index_element **found - place to return name's index entry
 *    struct index_element **prev - place to return index entry before name
 *
 *  purpose:
 *    in order to do much of anything with actual queues functions in here
 *    need to run down the main queue index and find the entry matching
 *    the tag that the user function proovided.  This function does that.
```

```
 *
 *  returns:
 *    found - pointer to the index entry of the requested queue.
 *    prefound - pointer to the index entry before found in the main index.
 *
 *    both of the above are NULL if name was not found.
 *
 */

static void seek(char *name,
                 struct index_element **found,
                 struct index_element **prefound) {

  struct index_element *current_entry = the_index;          /* index item */
  struct index_element *prev_entry = NULL;                  /* index item */

  /* run down the internal index and look for name */
  while (current_entry != NULL) {

    if (!strcasecmp(current_entry->qname, name)) {

/* gotcha! */
*prefound = prev_entry;
*found = current_entry;
return;

    }

    /* move on down, move on down the road... */
    prev_entry = current_entry;
    current_entry = current_entry->next;
  }

  /* if we get out here there was no such tag in the index */
  *prefound = NULL;
  *found = NULL;
  return;
}

/*\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

/*
 *  zapqueue - (internal) routine to free the contents of a queue
 *
 *  parameters:
 *    struct queue_element *head - the head of the queue to zap.
 *
```

```
 *  purpose:
 *    Given the head of a queue this routine runs down the queue and frees
 *    the memory of the items on the queue.
 *
 *  returns:
 *    nothing
 *
 */

static void zapqueue(struct queue_element *head) {
  struct queue_element *current;                          /* queue elements */
  struct queue_element *zapme;

  /* start at the beginning and free the whole beast */
  current = head;
  while (current != NULL) {

    zapme = current;
    current = current->next;
    free(zapme->data);
    free(zapme);

  }
}

/*\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

/*
 *  enqFIFO - (interface) enqueue in First-In-First-Out order
 *
 *  parameters:
 *    void *data - what to enqueue
 *    char *name - "tag" name of the queue to enqueue onto
 *
 *  purpose:
 *    Given a data item this function enqueues it in FIFO order (at the
 *    tail of the queue specified).
 *
 *  returns:
 *    nothing
 *
 */

void enqFIFO(void *data, char *name) {
  struct index_element *queue_entry;
  struct index_element *trash;
  struct queue_element *tail;
```

```
  struct queue_element *new;

  seek (name, &queue_entry, &trash);

  if (queue_entry != NULL) {

    if ((new = (struct queue_element *)
         malloc (sizeof(struct queue_element)))) {

       new->data = data;
       new->key = 0;                   /* this is a non-ordered queue */
       queue_entry->count += 1;
       tail = queue_entry->tail;

       /* if there's at least 1 item on queue, add new after current tail */
       if (tail) tail->next = new;

         /* otherwise this is the first thing on the queue - head and tail */
         else queue_entry->head = new;

       /* queue's new tail is this item */
       queue_entry->tail = new;

    } else error_handler(OUT_OF_MEMORY, 1);

  } else error_handler(NO_SUCH_TAG, 0);

}

/*\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

/*
 *  enqIO - (interface) enqueue In-order
 *
 *  parameters:
 *    void *data - what to enqueue
 *    int key - the key value with which to order said data item
 *    char *name - queue on which to enqueue the data item
 *
 *  purpose:
 *    Given the name of a queue, a data item, and it's computed "rank"
 *    this routine will enqueue it in order...
 *
 *  returns:
 *    nothing
 *
 */
```

```
void enqIO(void *data, int key, char *name) {
  struct index_element *queue_entry;          /* the index of the req queue */
  struct index_element *trash;          /* (unused) index of the prev queue */
  struct queue_element *current;                           /* queue element */
  struct queue_element *new;   /* newly allocated element for new data item */

  /* find it */
  seek (name, &queue_entry, &trash);

  if (queue_entry != NULL) {

    /* can we allocate? */
    if ((new = (struct queue_element *)
       malloc (sizeof(struct queue_element)))) {

       new->data = data;
       new->key = key;
       queue_entry->count += 1;


       /* at this point all we have to do is figure out where to put new */


       /* perhaps the queue is empty, then new is the head and the tail */
       if (queue_entry->count == 1) {

         queue_entry->head = new;
         queue_entry->tail = new;
         return;

       /* maybe new is the "lowest" item in the set, then put it at new head */
       } else if ((new->key) < (queue_entry->head->key)) {

         new->next = queue_entry->head;
         queue_entry->head = new;

       } else if ((new->key) >= (queue_entry->tail->key)) {

         queue_entry->tail->next = new;
         queue_entry->tail = new;
         return;

       /* otherwise step down the queue and look for where to put new */
       } else {

         current = queue_entry->head;
```

```
        while ((current->next) && (current->next->key <= new->key))
          current = current->next;

          new->next = current->next;
         current->next = new;

      }

    } else error_handler(OUT_OF_MEMORY, 1);

  } else error_handler(NO_SUCH_TAG, 0);

}

/*\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

/*
 *  deqFIFO - (interface) dequeue in a First-In-First-Out order.
 *
 *  parameters:
 *    char *name - the name of the queue to dequeue from.
 *
 *  purpose:
 *    Given the name of the queue, dequeue the front element on the
 *    queue and return it.
 *
 *  returns:
 *    The data element enqueued at the front of the queue specified OR
 *    NULL if the queue is empty / not found.
 *
 */


void *deqFIFO(char *name) {
  struct index_element *queue_entry;          /* the index of the req queue */
  struct index_element *trash;          /* (unused) index of the prev queue */
  struct queue_element *temp;            /* temporary queue element pointer */
  void *ret;                                 /* data item (found) to return */

  /* find it */
  seek (name, &queue_entry, &trash);

  if (queue_entry != NULL) {

     /* if there is only one thing on the queue, deq it */
     if (queue_entry->count == 1) {
```

```
      ret = queue_entry->head->data;
      free(queue_entry->head);
      queue_entry->head = NULL;
      queue_entry->tail = NULL;
      queue_entry->count = 0;
      return(ret);

    /* is there more than one thing on the queue? */
    } else if (queue_entry->count > 1) {

      temp = queue_entry->head;      /* we know tail is not the same thing */
      queue_entry->head = queue_entry->head->next;
      queue_entry->count--;
      ret = temp->data;
      free(temp);
      return(ret);

    /* else the queue is empty */
    } else return(NULL);

  } else {

     error_handler(NO_SUCH_TAG, 0);
     return(NULL);
  }
}

/*\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

/*
 *  deqmin - (interface) Given the name of a queue, deq the data item with
 *           the smallest key value on the queue.
 *
 *  parameters:
 *    char *name - the name of the queue to dequeue from.
 *
 *  purpose:
 *    This routine dequeues the smallest item with the smallest key value
 *    on a queue and returns the data stored in that item.  Because of the
 *    way that the queue module is implemented, this is the exact same
 *    thing as deqFIFO... so...
 *
 *  returns:
 *    pointer to data stored in the queue item with the smallest key.
 *
 */
```

```
void *deqmin(char *name) {
  return(deqFIFO(name));
}

/*\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

/*
 *  deqmax - (interface) Given the name of a queue, deq the data item with
 *           the largest key value on the queue.
 *
 *  parameters:
 *    char *name - the name of the queue to dequeue from.
 *
 *  purpose:
 *    See above
 *
 *  returns:
 *    pointer to the data item with the largest key value on the queue.
 *
 */

void *deqmax(char *name) {
  struct index_element *queue_entry;          /* the index of the req queue */
  struct index_element *trash;          /* (unused) index of the prev queue */
  struct queue_element *temp;                      /* queue element we find */
  void *ret;                                        /* data value to return */

  /* find it */
  seek (name, &queue_entry, &trash);

  if (queue_entry != NULL) {

     /* two or more things on the queue? */
     if (queue_entry->count > 1) {

       temp = queue_entry->head;
       while (temp->next != queue_entry->tail)
         temp = temp->next;

       /* now we've got the "next to last" thing */
       ret = queue_entry->tail->data;
       free(queue_entry->tail);
       queue_entry->tail = temp;
       temp->next = NULL;
       queue_entry->count--;
       return(ret);
```

```
     /* ah, the easy case... one thing on the queue */
     } else if (queue_entry->count == 1) {

       ret = queue_entry->head->data;
       queue_entry->count = 0;
       queue_entry->tail = NULL;
       free(queue_entry->head);
       queue_entry->head = NULL;
       return(ret);

     } return(NULL); /* my dear watson, the queue is empty... */

  /* my dear holmes, there is no such queue! */
  } else {

      error_handler(NO_SUCH_TAG, 0);
      return(NULL);
  }
}

/*\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

/*
 *  qcount - (interface) return the number of items on a certain queue.
 *
 *  parameters:
 *    char *name - the name of the queue to count.
 *
 *  purpose:
 *    returns the number of items currently on a certain queue.
 *
 *  returns:
 *    the integer numeral representing the numner of items on the queue...
 *
 */

int qcount(char *name) {
  struct index_element *queue_entry;                     /* out queue's index */
  struct index_element *trash;                                    /* [unused] */

  /* find it */
  seek(name, &queue_entry, &trash);

  /* if we've heard of it, tell the nice people how tall it is */
  if (queue_entry != NULL) return(queue_entry->count);

  error_handler(NO_SUCH_TAG, 0);
```

```
  return(0);

}

/*\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

/*
 *  qempty - (interface) is a certain queue empty?
 *
 *  parameters:
 *    char *name - which queue do you want to know anout?
 *
 *  returns:
 *    YES - if the queue is empty,
 *    NO - otherwise
 *
 */

bool qempty(char *name) {

  /* i love simple routines */
  return (qcount(name) == 0);

}

/*\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

/*
 *  peekq - (interface) peek at any item on a particular queue.
 *
 *  parameters:
 *    char *name - which queue to peek at.
 *    int element - what number element to peek at.
 *
 *  purpose:
 *    Given the name of a queue and a position, this routine will return the
 *    data item enqueued at position position.  Note, this routine does not
 *    dequeue said item.
 *
 *  returns:
 *    a pointer to the requested item or NULL if queue/item not found.
 *
 */

void *peekq (char *name, int element) {
  struct index_element *queue_entry;             /* out queue's index entry */
  struct index_element *trash;                                 /* [unused] */
```

```
  struct queue_element *current;                        /* item to peek at */
  int index;                                               /* loop control */


  /* range check */
  if ((element <= 0) || (element > qcount(name))) return(NULL);

  /* find our queue */
  seek(name, &queue_entry, &trash);
  if (queue_entry != NULL) {

    /* figure out what item they want */
    index = 1;
    current = queue_entry->head;

    while (current != NULL) {

      if (index == element) return(current->data);

      index++;
      current = current->next;
    }

  } else error_handler(NO_SUCH_TAG, 0);

}

/*\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

/*
 *  grep_deq - (interface) scan a queue's contents and dequeue the first one
 *             whose data item matches the target.
 *
 *  parameters:
 *    void *target - what data item should be dequeued.
 *    char *name - which queue to scan.
 *
 *  purpose:
 *    given the name of a queue and a particular data item that may be
 *    on it, this routine scans the contents of the queue and, if it
 *    find a match, dequeues and returns the contents of that match.
 *
 *  returns:
 *    (needlessly) returns a pointer to the match.  Should be equal to
 *    the input value of target.
 *
 */
```

```
void *grep_deq (void *target, char *name) {
  struct index_element *queue_entry;                 /* our queue's index */
  struct index_element *trash;                                /* [unused] */
  struct queue_element *current, *me;          /* looping qelement, found */
  void *ret;                                           /* value to return */

  /* find it [the queue, not the target] */
  seek(name, &queue_entry, &trash);

  if (queue_entry != NULL) {

    /* it we luck out and it's on the head/tail... */
    if (queue_entry->head->data == target)
      return(deqFIFO(name));

    if (queue_entry->tail->data == target)
      return(deqmax(name));


    /* if we are here it is not at the head or tail */

    /* run down the list until we find it or run out of list */
    current = queue_entry->head;
    while ((current->next->data != target) &&
           (current->next != queue_entry->tail)) current = current->next;

    /* gotcha */
    if (current->next->data == target) {

      me = current->next;
      current->next = me->next;
      queue_entry->count--;
      ret = me->data;
      free(me);
      return(ret);


    /* not found */
    } else return(NULL);

  } else error_handler(NO_SUCH_TAG, 0);

}
```

```
/* --- system.h and global.h you will have to define yourself --- */
```

0.7.2 Dekker's Algorithm for Mutual Exclusion

0.7.3 Peterson's Algorithm for Mutual Exclusion

0.8 Geometric Algorithms

0.8.1 Convex Hull Problem

Given a set of points in a two dimensional plane, a convex hull algorithm finds the set of perimeter points which, when connected, enclose all other points. This set of points is called the "convex hull."

There is more than one algorithm for convex hull finding but one thing that most have in common is the first step.

Gift Wrapping

Graham's Scan

0.8.2 Closest Pair Problem

0.8.3 Determining Whether a Point is Inside a Polygon

The following algorithm was sent to me by Romano Giannetti who got it from the comp.graphics.algorithms FAQ where it was originally written by Wm. Randolph Franklin. It works on convex or concave algorithms, simple and complex.

It works by the **ray tracing** method. Starting at the coordinates of the point in question draw a straight line in any direction. If the number of times it intersects the polygon border is odd the starting point is inside the polygon. If it is even the starting point is outside the polygon.

Another good polygon algorithm page is Darel Finley's. Check it out at http://freeweb.pdq.net/smokin/polygon.

Source Code

```
#include <iostream.h>
#include <math.h>
#include <stdlib.h>

//
```

```
//  Input parameters:
//     npol               : number of vertices
//     xp[npol], yp[npol] : x,y coord of vertices
//     x,y                : coord of point to test
//
//  Return Value:
//     0 : test point is outside polygon
//     1 : test point is inside polygon
//
//  Notes:
//     if test point is on the border, 0 or 1 is returned.
//     If there exists an adiacent polygon, the point is
//     _in_ only in one of the two.
//

int pnpoly(int npol, float *xp, float *yp, float x, float y)
{
    int i, j, c = 0;
    for (i = 0, j = npol-1; i < npol; j = i++)
    {

        if (
             (
               ((yp[i]<=y) && (y<yp[j])) ||
               ((yp[j]<=y) && (y<yp[i]))
             ) &&
             (x < (xp[j] - xp[i]) * (y - yp[i]) / (yp[j] - yp[i]) + xp[i]))

          c = !c;
    }
    return c;
}
```

0.8.4 Knapsack Problem

The knapsack problem deals with packing known size objects into a space such that the value of these items is maximized. The objects and the space can be x-dimensional. The number of objects is unlimited but there are n classes of objects. All members of one class are the same size. All classes are different sizes.

In most versions of the knapsack problem the value of the classes is variable and we must maximize the value of the objects in the space.

Packing n types of objects into a fixed space s can be solved by a recursive algorithm.

```
int knap(int cap)
```

```
{
  int i, space, max, t;

  /* N is the number of item classes */
  for (i = 0, max = 0; i < N; i++)
  {

    /*
     * ... figure out how much space would be left if we packed this
     * class of item (current free space minus this item requirement)
     */
    space = cap - items[i].size;

    /* if the space is still positive (i.e. we can pack it legally)... */
    if (space >= 0)
    {

      /* ... then do it and recurse on the new (less) amount of free space */
      t = knap(space) + items[i].val;

      if (t > max)
      {
        max = t;
      }
    }
  }

  /*
   * if we did not pack anything (i.e. nothing will fit) max = 0 and
   * return zero.  Else max is the ``most valuable'' way we can pack
   * at this point and we return its value to the previous layer.  If
   * the USING_ITEM_VALUES macro is not defined, all items are worth
   * one and we are trying to maximize the number of items we can fit,
   * not the value of items we can fit.
   */
  return(max);
}
```

The problem is that this solution is inefficient. Tracing the execution of the recursive solution you will notice that much of the algorithms time is spent recomputing the answer to a subproblem that it has seen in the past. For instance, if we have already discovered the best way to pack a space of size 14 once, why should we go though the process of computing it all over again? Yet the above solution does just that. Imagine we have a starting knapsack size of twenty-five. We pack an item value v_1 of size ten into the sack. Then we pack another item of value v_2 and size one into the sack. We now have fourteen free space in the knapsack and want to maximize the value we can fit into this space. Assume we do so.

Now we recurse out to the first instance of knap in the call stack and proceed to change the first item. We now pack an item of value v_3 and size six as the first thing in the sack. In the following call to knap's loop, at some point, we pack a second item into the sack with a value of v_4 and a size of five. We now are facing the same exact problem we solved before: how to pack a free space of size fourteen and maximize the value. Because we have solved this problem, it would make sense to simply reuse the solution and avoid the recomputation. The recursive solution to the knapsack problem does not do this, though; rather it dumbly recomputes. In fact, because of the massive amounts of recumputation involved, an analysis of this recursive solution will discover that it takes exponential time. This is unacceptable for anything but the smallest problems.

Enter **dynamic programming**. This is an often-misunderstood concept; all it means is that if you have one large problem made up of many subproblems, the first time you tackle a subproblem save its result so that if you must handle the same subproblem again at some point you can skip the computation and just use the old result. Here's how we can use this technique to make the recursive example previously shown linear in complexity:

```
int knap(int cap) {
  int i, space, max, maxi, t;

  /* the first thing we do is see if we have an answer to how to pack
   * items into cap space maximizing the value of the items.  If so we
   * do not need to do anything but return!
   */

  if (max_known[cap] != UNKNOWN)
  {
    return(max_known[M]);
  }

  /* if we got here we have not yet solved this problem, so lets do
   * it!
   */
  for (i = 0, max = 0; i < N; i++)
  {

    /* see comments on recursive solution -- understand it before you
     * read this
     */

    space = cap - items[i].size;
    if (space >= 0)
    {
      t = knap(space) + items[i].val;
      if (t > max)
      {
        max = t;
        maxi = i;
```

```
      }
    }
  }

  /* now that we have packed the required space as well as we can,
   * remember how well we could do it.  Thus, if we are ever required to
   * do it again we can just use this saved value...
   */

  max_known[cap] = max;
  item_known[cap] = items[maxi];

  return(max);
}
```

References

1. Sedgewick, Robert 1998, *Algorithms in C, 3rd ed* (Reading, MA: Addison-Wesley), pp. 211-220

0.9 Data Encryption Algorithms

0.10 Data Compression Algorithms

The goal of data compression is to represent a some set of information as space efficiently as possible. A data compression **code** is a mapping between some set of **source messages** and a set of **codewords**. A source message does not have to be and is not usually an entire string being compressed. Rather, it is the set of symbols or strings into which the data being compressed is partitioned for processing. These basic units may be single symbols from the source string's alphabet, or they may be strings of such symbols. The process of converting from a source stream into a coded (hopefully compressed) message is called **encoding** while the inverse operation is called **decoding**. A **lossless** encoding method is one in which the process of decompression results in no loss of original data whereas **lossy** encoding method is one in which the original data cannot be fully recovered.

Codes can be of the types **block-block** or **variable-variable**. Codes of the block-block variety operate on static, fixed-length codewords and source messages while their counterparts operate on dynamic length codewords and source messages. One example of a block-block type code is the **ASCII code**. It is of the block-block variety because it maps fixed-length source messages (characters) into fixed-length codewords (their ASCII values).

Because variable-variable type codes produce codewords that do not have a fixed length, when processing the output of a variable-variable code it is vital to be able to differentiate between codewords in the stream. Fixed length codewords are easy to distinguish due to their regular spacing pattern. However, we do not have this luxury when dealing with variable-variable codes.

The sequence of codewords or source messages in a stream is called an **ensemble**.

A coding function is called **distinct** if its mapping from source messages to codewords is one-to-one. Such a code is called **uniquely decodable** if every codeword is recognizable even when immersed in a stream of other codewords. A uniquely decodable code is known as a **prefix code** if it has the property that no codeword in the code is a prefix of any other codeword.

Data compression schemes are said to be either **static** or **dynamic** (or **adaptive**). A static function is one in which the mapping from the input source messages to the set of codewords is fixed before the data compression begins. In such a system a given message is always represented by the same codeword regardless of where it appears in the ensemble. In contrast, a dynamic (or adaptive) algorithm may change the mapping for a particular source message during the compression process.

0.10.1 Run-Length Encoding

Sometimes called **recurrence coding**, This is one of the simplest data compression algorithms but is effective for data sets which are comprised of long sequences of a single repeated character. For instance, text files with large runs of spaces or tabs may compress well with this algorithm. Old versions of the `arc` compression program used run-length encoding.

The way RLE works is by finding runs of repeated characters in the input stream and replacing them with a three-byte code. The code consists of a flag character, a count byte, and the repeated character. For instance, the string "AAAAAABBBBCCCCC" could be more efficiently represented as "*A6*B4*C5". That saves us six bytes. Of course, since it does not make sense to represent runs less than three characters in length with a code, none is used. Thus "AAAAAABBCCCDDDD" might be represented as "*A6BBCCC*D4". The flag byte is called a **sentinel byte**.

One problem with this approach lies in the selection of the sentinel value. This flag signals to the decompression code that an encoded sequence follows. Ideally we would like to select a byte value that does not occur in the input stream. However, running through the input stream looking for an absent value slows down this algorithm. Since the main advantage of this algorithm is its speed, slowing it down for only slightly better compression results is not usually considered. Often an arbitrary, non-letter ASCII value is chosen as the sentinel. When compressing a stream with natural occurrences of the sentinel flag, the flag byte must be represented. Sometimes a three-byte code is used with a count of less than three. Another aproach is to use a doubled flag byte represent one naturally occurring flag byte in the input. Thus, "AAAAAA*BBBBCC*D" would encode to "*A6***B4CC**D". The two stars after "A6" and the final two stars denote a natural star in the input stream.

It is wasteful to only encode run counts with ordinal numbers from zero to nine. Instead we can use the value of the byte to denote the number of characters in the run. In this way we can get a range of zero to 255. However, since we are never encoding any run with less than four characters, the range becomes four to 259. Runs comprised of more than 259 characters in them will be broken up into two or more three-byte flag-character-count sequences.

RLE is used as one of the steps in the JPEG image compression process. Basically the JPEG algorithm breaks an image up into a series of 8x8 matrixes and proceeds to run a "DCT" transformation on each matrix. This transformation isolates the important image components in the upper left portion of the matrix. The lossy step of the JPEG process happens when values far from the upper-left corner are

rounded. Unless these values are of very high magnitude they tend to become zeros. However, the further away from the upper left corner of the matrix, the less likely bytes are to have a high magniture so there ends up being a lot of zeros in the transformed matrix. To maximize the length of zero runs, JPEG scans the matrix in a diagonal pattern and then uses RLE to encode multiple zeros in a row efficiently. This is a terribly high-level explaination of how JPEG works and I would recommend reading Nelson and Gailly's chapter on lossy graphic compression for more information.

Source Code

```
/* --- rle_encode.c --- */

#include <stdlib.h>
#include <stdio.h>

#include "global.h"
#include "debug.h"

//
// Modified version of Rex & Binstock's rle1.c, see Practical Algorithms
// for Programmers pg. 470-472.
//


//
// This is the byte value that will denote an encoded run sequence
//
#define FLAG            (char) 0xF0

//
// Input buffer size
//
#define BUFFER_SIZE     30000

//
// Macro for writing a three-byte run sequence code to output
//
#define WRITE_CODE(a, b, c)                               \
    ASSERT((b) > 3);                                      \
    ASSERT((b) < 260);                                    \
    fprintf(pfOutFile, "%c%c%c", (a), (b - 4), (c));      \


//
// Global file handles
//
FILE *pfInFile, *pfOutFile;

int main (int argc, char *argv[])
{
```

```
char *buf;
char chPrevChar;
DWORD dwBytesRead;
DWORD dwCount;
DWORD dwI;
BOOL fEof = FALSE;
BOOL fFirstTime = TRUE;

if (argc != 3)
{
  fprintf(stderr, "Usage: %s infile outfile\n", argv[0]);
  exit(1);
}

if ((pfInFile = fopen(argv[1], "rb")) == NULL)
{
  perror(argv[1]);
  exit(1);
}

if ((pfOutFile = fopen(argv[2], "wb")) == NULL)
{
  perror(argv[2]);
  exit(1);
}

//
// This code writes a header to the output file -- it records what
// value was used as the FLAG and that this is an RLE encoded file.
// This header can be omitted if the flag value will always be the
// same and you don't care about magic numbers idenfitying binary file
// types.
//

fprintf(pfOutFile, "%c%c%c%c", 'R', 'L', 'E', FLAG);

//
// Get memory for the buffer
//

buf = (char *) malloc(BUFFER_SIZE);
if (!buf)
{
  fclose(pfInFile);
  fclose(pfOutFile);
  fprintf(stderr, "Out of memory, cannot allocate buffer.\n");
  exit(1);
}

//
```

```
  // Process input
  //

  while (!fEof)
  {
    dwBytesRead = fread(buf, 1, BUFFER_SIZE, pfInFile);
    if (!dwBytesRead)
    {
      fEof = TRUE;
      break;
    }


    for (dwI = 0; dwI < dwBytesRead; dwI++)
    {

      //
      // First time is a special case
      //

      if (fFirstTime)
      {
chPrevChar = buf[dwI];
dwCount = 1;
fFirstTime = FALSE;
dwI++;
      }



      //
      // See if we have a run in the making
      //
      if (buf[dwI] == chPrevChar)
      {
dwCount += 1;
if (dwCount == 259)
{
  WRITE_CODE(FLAG, dwCount, chPrevChar);
  dwCount = 0;
}
continue;
      }

      //
      // Else there's a new character... possibly write data,
      // definately reset the count and prevchar...
      //
      else
      {
```

```
//
// Write code
//
if (dwCount < 3)
{

  //
  // Non-adjusted count for flag character.
  //
  if (chPrevChar == FLAG)
  {
    fprintf(pfOutFile, "%c%c%c", FLAG, dwCount, FLAG);
  }
  else
  {
    do
    {
      fputc(chPrevChar, pfOutFile);
    }
    while (--dwCount);
  }
}
else
{
  WRITE_CODE(FLAG, dwCount, chPrevChar);
}

chPrevChar = buf[dwI];
dwCount = 1;
      }
    }

    //
    // End of bytes read... is it eof?
    //
    if (dwBytesRead < BUFFER_SIZE)
    {
      fEof = TRUE;
    }

  }

  //
  // At EOF, flush out buffers
  //
  if (dwCount < 3)
  {
    if (chPrevChar == FLAG)
    {
```

```
      //
      // Encode a flag char with FLAG, count, FLAG -- note that in
      // this case the count is not adjusted.
      //
      fprintf(pfOutFile, "%c%c%c", FLAG, dwCount, FLAG);
    }
    else
    {
      do
      {
fputc(chPrevChar, pfOutFile);
      }
      while(--dwCount);
    }
  }
  else
  {
    WRITE_CODE(FLAG, dwCount, chPrevChar);
  }

  fclose(pfInFile);
  fclose(pfOutFile);
  exit(0);
}
```

```
/* --- rle_decode.c --- */

#include <stdio.h>
#include <stdlib.h>

#include "global.h"
#include "debug.h"

//
// Modified version of Rex & Binstock's rle1.c, see Practical Algorithms
// for Programmers pg. 474-476.
//


//
// Input buffer size
//
#define BUFFER_SIZE     30000

//
// Global file handles
//
```

```
FILE *pfInFile, *pfOutFile;

int main (int argc, char *argv[])
{
  char FLAG;
  char *buf;
  DWORD dwBytesRead;
  DWORD dwCount;
  DWORD dwI;

  if (argc != 3)
  {
    fprintf(stderr, "Usage: %s infile outfile\n", argv[0]);
    exit(1);
  }

  if ((pfInFile = fopen(argv[1], "rb")) == NULL)
  {
    perror(argv[1]);
    exit(1);
  }

  if ((pfOutFile = fopen(argv[2], "wb")) == NULL)
  {
    perror(argv[2]);
    exit(1);
  }

  //
  // Get memory for the buffer
  //

  buf = (char *) malloc(BUFFER_SIZE);
  if (!buf)
  {
    fclose(pfInFile);
    fclose(pfOutFile);
    fprintf(stderr, "Out of memory, cannot allocate buffer.\n");
    exit(1);
  }

  //
  // Decode the header...
  //
  dwBytesRead = fread(buf, 1, 4, pfInFile);

  if (dwBytesRead != 4)
  {
    fprintf(stderr, "Unable to read %s\n", argv[1]);
    fclose(pfInFile);
```

```
    fclose(pfOutFile);
    exit(1);
  }

  if ((buf[0] == 'R') && (buf[1] == 'L') && (buf[2] == 'E'))
  {
    FLAG = buf[3];
  }
  else
  {
    fprintf(stderr, "Warning: %s may not be in valid format.\n", argv[1]);
    FLAG = 0xF0;
  }

  //
  // Process input
  //

  while (1)
  {
    dwBytesRead = fread(buf, 1, BUFFER_SIZE, pfInFile);
    if (!dwBytesRead)
    {

      //
      // Unable to read = EOF
      //
      break;
    }


    //
    // Process the buffer
    //
    for (dwI = 0; dwI < dwBytesRead; dwI++) {

      //
      // Not a sentinel -- put it straight through
      //
      if (buf[dwI] != FLAG)
      {
fputc(buf[dwI], pfOutFile);
      }

      //
      // Is a sentinel -- process it
      //
      else
      {
```

```
//
// Near buffer end -- how near?
//
if (dwI > dwBytesRead - 3) {

  //
  // Missing two bytes in buffer, read from the file
  //
  if (dwI > dwBytesRead - 2)
  {
    dwBytesRead = fread(buf, 1, BUFFER_SIZE, pfInFile);
    if (dwBytesRead < 2)
    {
      fprintf(stderr, "Error in %s.\n", argv[1]);
      fclose(pfInFile);
      fclose(pfOutFile);
      exit(1);
    }
    else
    {

      //
      // buffer now starts with [count][byte] -- process it
      //
      dwI = -1;
      goto process_count;
    }
  }

  //
  // Missing one byte in buffer, get it
  //
  else
  {
    dwCount = buf[dwI + 1];
    dwBytesRead = fread(buf, 1, BUFFER_SIZE, pfInFile);

    if (dwBytesRead < 1)
    {
      fprintf(stderr, "Error in %s.\n", argv[1]);
      fclose(pfInFile);
      fclose(pfOutFile);
      exit(1);
    }

    //
    // buf now starts with [byte]
    //
    dwI = 0;
    if (buf[dwI] != FLAG)
```

```
    {
      dwCount += 4;
    }

    do
    {
      fputc(buf[dwI], pfOutFile);
    }
    while (--dwCount);
  }
}

//
// Not end of buffer
//
else
{

process_count:

  dwCount = buf[++dwI];

  if (buf[dwI + 1] != FLAG)
  {
    dwCount += 4;
  }

  dwI++;
  do
  {
    fputc(buf[dwI], pfOutFile);
  }
  while (--dwCount);
}

      } // if sentinel

    } // for (process buffer)

    if (dwBytesRead < BUFFER_SIZE)
    {
      break;
    }

  } // while (1)

  fclose(pfInFile);
  fclose(pfOutFile);
  exit(0);
```

```
} // main
```

References

1. Binstock, Andrew and Rex, John. 1995, *Practical Algorithms for Programmers* (Reading, MA: Addison-Wesley), pp. 468-478.

2. Nelson, Mark and Gailly, Jean-Loup. 1996, *The Data Compression Book* (New York, NY: M&T Books), pp. 31-74.

0.10.2 Integer Coding

Section note: this description and source code are provided courtesy of Hugh Williams.

Integer coding is a method by which a set of integer values can be represented more efficiently. Two of the most often used methods of integer coding are called **Elias codes** and **Golomb codes**. Both Elias and Golomb codes are frequently used in compressing inverted indexes of English text.

In the Elias gamma code, a positive integer x is represented by: $1 + \lfloor \log_2 x \rfloor$ in unary. (that is, $\lfloor \log_2 x \rfloor$ 0-bits followed by a 1-bit), followed by the binary representation of x without its most significant bit.

Thus the number nine is represented by 0001001, since: $1 + \lfloor \log_2 9 \rfloor = 4$ or 0001 in unary, and 9 is 001 in binary with the most significant bit removed. In this way, 1 is represented by 1, that is, is represented in one bit. Gamma coding is efficient for small integers but is not suited to large integers for which parameterized Golomb codes or a second Elias code, the delta code, are more suitable.

Elias delta codes are somewhat longer than gamma codes for small integers, but for larger integers, such as ordinal sequence numbers, the situation is reversed. A delta code stores the gamma code representation of an integer x, followed by the binary representation of x less the most significant bit. However, while Elias codes yield acceptable compression and fast decoding, better performance in both respects is possible with Golomb codes.

Golomb codes are a form of parameterized coding in which integers to be coded are stored as values relative to a constant k. Using Golomb coding, a positive integer x is represented in two parts: the first is a unary representation of the quotient: $q = \lfloor (x-1)/k \rfloor + 1$ the second is a binary representation of the remainder: $x - qk - 1$. In this way, the binary representation requires $\lfloor \log k \rfloor or \lceil \log k \rceil$ bits.

Witten et al. report that for cases where the probability of any particular value occurring is small, an approximate calculation of k can be used. Where there is a wide range of values to be coded and each occurs with reasonable frequency, a practical global approximation of the Golomb parameter k is

$$k \approx 0.69 \times \frac{N \times p}{f}$$

where N is the number of documents, p is the number of distinct terms in the collection, and f is the count of document identifiers stored in inverted lists, that is, f is the sum of the lengths of all inverted lists. This model for selection of k is often referred to as a global Bernoulli model, since each term is assumed to have an independent probability of occurrence and the occurrences of terms have a geometric distribution.

Another approach to selecting k is to use a local model. Local models use the information stored within a list to calculate an appropriate k value for that list; local models result in better compression than global models, but require a parameter for each locality. For example, by using a simple local Bernoulli model for storing sequence identifiers, a possible choice of a k value for a given list is an approximation of the mean difference between the document identifiers in that list, or, using the scheme above,

$$k \approx 0.69 \times \frac{N}{l}$$

where l is the length of a given list, that is, the count of entries in the list.

Skewed Bernoulli models, where a simple mean difference is not used, typically result in better compression than simple local models.

With integers of varying magnitudes, as is the case in document occurrence counts and inverted-file offsets that vary from 1 to the database size, efficient storage is possible by using a variable-byte integer scheme. We use a representation in which seven bits in each byte is used to code an integer, with the least significant bit set to 0 if this is the last byte, or to 1 if further bytes follow. In this way, we represent small integers efficiently; for example, we represent 135 in two bytes, since it lies in the range

$$[2^7 \cdots 2^{14})$$

, as 00000011 00001110; this is read as 00000010000111 by removing the least significant bit from each byte and concatenating the remaining 14 bits.

Source Code

```
/* --- Makefile --- */

CC = gcc
CFLAGS = -O4
LIBS = -lm
RM = /bin/rm -f

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

BINARIES = testgolomb

.c.o:
$(CC) $(CFLAGS) -c $<

all: $(BINARIES)
```

```
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

testgolomb: testgolomb.o vec.o vbyte.o $(HFILES)
$(CC) $(CFLAGS) -o testgolomb testgolomb.o vec.o vbyte.o $(LIBS)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

clean:
/bin/rm -f a.out core *.o *~ $(BINARIES)
```

```
/* --- def.h --- */

#include <stdio.h>
#include <ctype.h>
#include <math.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>

/* Miscellaneous Constants */
#define true 1
#define false 0
#define FALSE 0
#define TRUE 1

#define BIGVECLEN 5000000
#define GOLHASHTABLESIZE 102397

/* Structure to hold bitvector */
typedef struct bitvecrec
{
    char *vector; /* Sequence of bytes to hold bitvector */
    int size; /* Of vector, in bytes */
    int pos; /* Current byte number */
    int bit; /* Current bit in byte */
    int cur; /* Temporary space for putting, getting bits */
    int len; /* Number of bits used */
    int last; /* Total of runlengths seen so far */
/* The last field is an oddity; it allows us to have
   several bitvectors open at once */
} BITVEC;
```

```
typedef struct golombhashstruct
{
int b; /* b value used for lookup */
int q;  /* q value that is floor(log_2(b)) */
} GOLHASH;

#define L2(f) ( log10((f)) / 0.301029995 /*=log10(2.0)*/ )
#define L4(f) ( log10((f)) / 0.602059999 /*=log10(4.0)*/ )
#define L6(f) ( log10((f)) / 0.778151250 /*=log10(6.0)*/ )
#define L8(f) ( log10((f)) / 0.903089990 /*=log10(8.0)*/ )
#define LN(f) ( log10((f)) / 0.434294480 /*=log10(e)*/ )

char *malloc(int);
void padvec(BITVEC *), expandvec(BITVEC *), initcmp(), initvec(BITVEC *, int),
freevec(BITVEC *), resetvec(BITVEC *), inittree(), cleanuptree(),
dumpvec(BITVEC *,FILE *), dumptree(char *);

int vbyteread(FILE *);
int vbytewrite(int, FILE *);
```

```
/* --- vec.c --- */

/* VECTOR routine suite for variable-bit integer coding
 * Original vector and Elias coding Copyright (C) 1996 to Justin Zobel
 * This code, variable byte integers, and Golomb coding Copyright (C) 1996-8
 * to Hugh Williams
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * See Copyright.txt for further details. */

/* Code for creating and accessing bitvectors.  They cannot be accessed
   until complete; once complete, they can only be extended by decoding,
   then recoding into a new vector. The code also assumes that the numbers
   to be encoded are at least one bit shorter than an integer. */
```

```
#include "def.h"

/* #define DEBUGGOLOMB              */
/* #define DEBUGELIAS */

int disaster = FALSE;
static int putnbits(BITVEC *,int,int);
static int getnbits(BITVEC *,int);
static int putbit(BITVEC *,int);
static int getbit(BITVEC *);

#ifdef DIAGS
int totalvec = 0;
#endif

GOLHASH golhashtable[GOLHASHTABLESIZE];

/* Number of bits written to any bitvector; used for consistency check */
static int bitcount;

void
initcmp()
{
    bitcount = 0;
}


/* Return length of vector */
int
veclen(BITVEC *vp)
{
    return(vp->len);
}


/* Before a vector can be read, it must be reset */
void
resetvec(BITVEC *vp)
{
    vp->last = -1;
    vp->pos = vp->bit = 0;
vp->cur = vp->vector[0];
}

void
freevec(BITVEC *vp)
{
```

```
free(vp->vector);
}

/* Initialising a vector before insertion of data */
void
initvec(BITVEC *vp, int initveclen)
{
    vp->vector = malloc(initveclen * sizeof(char));
    vp->size = initveclen;
    vp->last = -1;
    vp->bit = vp->pos = vp->len = vp->cur = 0;
}


/* Write the contents of a vector to file.  Assumes that padvec has
   been called. */
void
dumpvec(BITVEC *vp, FILE *fp)
{
#ifdef DEBUG
    int i;

    resetvec(vp);
    fprintf(stderr,"[", i);
    while( (i=getrunlen(vp)) >=0 )
fprintf(stderr," %d", i);
    fprintf(stderr,"] {", i);
    resetvec(vp);
    while( (i=getdelta(vp)) >=0 )
fprintf(stderr," %d", i);
    fprintf(stderr,"}", i);
#endif  /* DEBUG */

    fwrite(&vp->len, sizeof(int), 1, fp);
    fwrite(vp->vector, sizeof(char), vp->len/8 + 1, fp);
}


/* Read a vector back from file */
/* Returns -1 on error */
int
readvec(BITVEC *vp, FILE *fp)
{
    int len;

    fread(&vp->len, sizeof(int), 1, fp);
    vp->size = 4 * ( ( vp->len/8 + 8 ) / 4 ); /* word boundaries */
```

```
    len = vp->len/8 + 1;
    vp->vector = malloc(vp->size * sizeof(char));
    fread(vp->vector, sizeof(char), len, fp);
    resetvec(vp);
    return(0);
}


/* Put a unary number in a vector (unary means x is represented as x 0-bits, followed
   by a single 1-bit. Cute for small numbers */
int putunary(BITVEC *vp, int n)
{
int ret;

    ret = putnbits(vp, 0, n);
    ret += putbit(vp, 1);

return(ret);
}

int getunary(BITVEC *vp)
{
int ret = 0;

for(;getbit(vp) != 1;ret++);

return(ret);
}

/* Add the number n to the vector, assuming run-length coding */
int
putrunlen(BITVEC *vp, int n)
{
    int ret;

    if( n <= vp->last )
return(0); /* hack: don't add the same code twice */
    ret = putdelta(vp, n - vp->last);
    vp->last = n;
    return(ret);
}


/* Get a number from the vector, assuming run length coding */
int
getrunlen(BITVEC *vp)
{
```

```
    int ret;

    ret = getdelta(vp);
    if( ret > 0 )
    {
vp->last += ret;
return(vp->last);
    }
    else
return(-1);
}


/* Standard delta (Elias) */
int
putdelta(BITVEC *vp, int n)
{
    double floor(), log10();
    int mag;
    int ret;

    mag = (int) floor(L2((double) n));

    ret = putgamma(vp, mag+1);
    ret += putnbits(vp, n, mag); /* don't output leftmost (ie, top) bit */

    return(ret);
}


/* Returns -1 on eof */
int
getdelta(BITVEC *vp)
{
    int mag, val;

    mag = getgamma(vp) - 1;
    if( mag < 0 )
return(-1);

    val = getnbits(vp, mag);

    if( val < 0 )
return(-1);

    return( ( 1 << mag ) | val );
}
```

```
/* Standard gamma (Elias) */
int
putgamma(BITVEC *vp, int n)
{
    double floor(), log10();
    int mag;
    int ret;

    mag = (int) floor(L2((double) n));

    ret = putnbits(vp, 0, mag);
    ret += putnbits(vp, n, mag+1);

    return(ret);
}


/* Returns -1 on eof */
int
getgamma(BITVEC *vp)
{
    int b, mag, val;

    for( mag=0 ; ( b = getbit(vp) ) == 0 ; mag++ );

    if( b < 0 )
    {
return(-1);
    }
    val = getnbits(vp, mag);
    if( val < 0 )
    {
return(-1);
    }
    return( ( 1 << mag ) | val );
}

void
initgolhash()
{
int x;
for (x=0;x<GOLHASHTABLESIZE;x++)
golhashtable[x].b = -1;
}
```

```
/* Standard Golomb */
/* x = value to be put in */
/* b = coding parameter */
int
putgolomb(BITVEC *vp, int x, int b)
{
    int mag;
    int ret;
    int     rem;
int q;
int d;

/* Force +1 onto any value stored---to ensure non-zero values */
x++;

/* work out floor of (x - 1)/b */
    mag = (int) floor(((double)(x - 1))/((double)b));
#ifdef DEBUGGOLOMB
printf("-----------\nstoring: %d with b: %d, mag: %d\n", x,b,mag);
#endif

/* Store mag+1 in unary */
/* Think about it: mag=3 means storing 001, hence this DOES work */
ret = putnbits(vp, 0, mag);
ret += putbit(vp, 1);

#ifdef DEBUGGOLOMB
printf("stored mag of: %d\n", mag+1);
#endif

/* What's the remainder? ie. x - (mag * b) */
rem = x - (mag * b) - 1;
#ifdef DEBUGGOLOMB
printf("rem: %d\n", rem);
#endif

/* Now work out the point at which coding requires floor(log(b))+1 bits */
/* There are b remainders, so 2^k < b <= 2^(k+1) */
/* Therefore, 2^k <= b and then k = log_2 b, so first k remainders */
/* can be represented in floor(log_2 b) bits [phew] */

q = (int) floor(L2((double) b));

/* Now, work out the base value to use when we hit the toggle point (q)  */
/* Since there are q+1 bits (max) used to represent a remainder, there  */
/* can be 2^(q+1) such values using q+1 bits. Since there are b values  */
/* we want to represent, if the remainder is less than 2^(q+1) - b, in other  */
```

```
/* words if there are "spares", we can use q bits [phew again!].  */
/* Geez, Golomb coding isn't fun. */
/* So, d = 2^(q+1) - b (jz/sara figured this, thanks) */
d = (1 << (q+1)) - b;

#ifdef DEBUGGOLOMB
printf("toggle point q: %d with a d of: %d\n", q, d);
#endif
/* Can this remainder be represented in floor(log_2 b) bits? */
if (rem < d)
/* Store the value of remainder directly */
ret += putnbits(vp, rem, q);
else
{
#ifdef DEBUGGOLOMB
printf("actually storing: %d in toggle+1 (%d) bits\n", d+rem, q+1);
#endif
/* Store the value of (d + remainder) in floor(log_2 b) + 1 bits */
ret += putnbits(vp, d+rem, q+1);
}
#ifdef DEBUGGOLOMB
printf("stored: %d bits\n-----------\n", ret);
#endif
return(ret);
}

/* Standard Golomb */
/* b = coding parameter */
int
getgolomb(BITVEC *vp, int b)
{
    int mag;
int q, d;
int rem;
int ret;
int slot;
GOLHASH golhash;

/* Get mag+1 in unary */
/* By setting mag=0, we actually get mag [rather than mag+1, which */
/* is what was stored] [phew!] */
    for( mag=0 ; getbit(vp) == 0 ; mag++ );
#ifdef DEBUGGOLOMB
printf("-----------\ngot a mag: %d, meaning mag is: %d\n", mag+1, mag);
#endif

/* Try and look up the log_2(b)... if not there, calculate and store */
```

```
golhash = golhashtable[b % GOLHASHTABLESIZE];
if (golhash.b == b)
q = golhash.q;
else
{
/* Now work out the toggle point for remainders as before */
q = (int) floor(L2 ((double) b));
slot = b % GOLHASHTABLESIZE;
golhashtable[slot].b = b;
golhashtable[slot].q = q;
}

#ifdef DEBUGGOLOMB
printf("toggle: %d\n", q);
#endif

/* Get the first q bits (we may need (q+1) actually) */
    rem = getnbits(vp, q);

#ifdef DEBUGGOLOMB
printf("rem: %d\n", rem);
#endif

/* Now, work out the base value to use for the toggle point (q) */
/* It's d=2^(q+1) - b           (jz/sara figured this, thanks) */
/* See my notes in putgolomb() */
d = (1 << (q+1)) - b;

/* Do we need to get an extra bit? */
if (rem >= d)
{
/* If so:
   (1) shift rem left and OR with the bit we get
       (2) subtract d to get remainder */
rem = ((rem << 1) | getbit(vp)) - d;

#ifdef DEBUGGOLOMB
printf("new rem: %d\n", rem);
#endif
}

/* The value is the mag * b + remainder + 1 */
ret =  mag * b + rem + 1;

#ifdef DEBUGGOLOMB
printf("ret stored: %d", ret);
#endif
```

```
/* Force -1 onto any value stored---used it putgolomb to ensure non-zero values */
ret--;

#ifdef DEBUGGOLOMB
printf("ret returned: %d\n-----------\n", ret);
#endif
return(ret);
}

/* Put n bits in vector, starting with nth from right, going to rightmost */
/* This is not efficient.  Should be recoded to insert several bits
   simultaneously ... but not really worth the effort ... */
int
putnbits(BITVEC *vp, int n, int num)
{
    int i;
    int ret;

    for( ret=0, i=num-1 ; i>=0 ; i-- )
ret += putbit(vp, ( n >> i ) & 0x1 );
    return(ret);
}


static int masks[9] = { 0x0, 0x1, 0x3, 0x7, 0xf, 0x1f, 0x3f, 0x7f, 0xff };


/* Get n bits from vector */
/* Rather than calling getbit for each bit, shifts several bits at
   once if this would empty the current byte; only makes it one or
   two percent faster. */
int
getnbits(BITVEC *vp, int num)
{
    int mask, shift, val, b;

    b = val = 0;

#if true
    for( shift = 8-vp->bit ; num >= shift ; num -= shift, shift = 8-vp->bit )
/* copy out whole of cur */
    {
if( 8*vp->pos + vp->bit >= vp->len)
    return(-1);
mask = masks[shift];
val = ( val << shift ) | ( vp->cur & mask );
```

```
vp->bit = 0;
vp->pos++;
vp->cur = vp->vector[vp->pos];
    }
#endif /* true */

/* Get any remaining bits */
    for( ; num>0 && ( b = getbit(vp) ) >= 0 ; num-- )
val = ( val << 1 ) | ( b & 0x1 );
#ifdef VECDEBUG
    if( val < 0 )
printf("Panic in getnbits: overflow\n");
    if( b < 0 && vp->pos < vp->len/8 )
printf("Early termination in getnbits.\n");
#endif /* VECDEBUG */
    return( ( b<0 ) ? -1 : val );
}


/* Put a bit into vector; bytes are filled left to right */
/* Because of the way bytes are filled, vectors can't be accessed until
   padvec has been called. */
int
putbit(BITVEC *vp, int b)
{
    bitcount++;

    vp->cur = ( vp->cur << 1 ) | ( b & 0x1 );
    vp->bit++;
    if( vp->bit == 8 ) /* go to next byte */
    {
if( vp->pos >= vp->size )
    expandvec(vp);

if (disaster == FALSE)
{
vp->vector[vp->pos] = vp->cur & 0xff;
vp->cur = 0;
vp->pos++;
vp->bit = 0;
    }
}
    return(1);
}


/* Get a bit from a vector */
```

```
int
getbit(BITVEC *vp)
{
    int b;

#ifdef DEBUGBIT
printf("yep\n");
#endif

    if( 8*vp->pos + vp->bit >= vp->len )
return(-1);

    b = ( vp->cur >> ( 7 - vp->bit ) ) & 0x1;
    vp->bit++;

    if( vp->bit == 8 )
    {
vp->pos++;
vp->bit = 0;
vp->cur = vp->vector[vp->pos];
    }
    return(b);
}


/* Terminate a vector */
void
padvec(BITVEC *vp)
{
    if( vp->pos >= vp->size ) /* this is a bit yucky */
expandvec(vp);
if (disaster == FALSE)
{
    vp->vector[vp->pos] = ( vp->cur << ( 8-vp->bit ) ) & 0xff;
    vp->len = 8*vp->pos + vp->bit;
}
}


/* Enlarge vector */
void
expandvec(BITVEC *vp)
{
    char *newvec;
    int i;

/* should use realloc here */
```

```
    if ((newvec = malloc( vp->size*2 * sizeof(char) )) == NULL)
    {
     fprintf(stderr, "Error in expandvec malloc. Tried to allocate: %d bytes\n", vp->size*2 *
disaster = TRUE;
    }
else
{
for( i=0 ; i<vp->size ; i++ )
newvec[i] = vp->vector[i];
vp->size = vp->size*2;
free(vp->vector);
vp->vector = newvec;
}
}
```

```
/* --- vbyte.c --- */

/* Variable-byte integer coding
 * This code Copyright (C) 1996-8 to Hugh Williams
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * See Copyright.txt for further details. */

#include "def.h"

/*  Variable byte length reads and writes of integers */
/*  We use a representation in which seven bits in each byte is used to code an
integer, with the least significant bit set to 0 if this is the last byte,
or to 1 if further bytes follow.
In this way, we represent small integers efficiently; for example, we
represent 135 in two bytes, since it lies in the range $[2^7\cdots 2^{14})$,
as 00000011~00001110; this is read as 00000010000111 by removing the least
significant bit from each byte and concatenating the remaining~14~bits. */

int vbyteread(FILE *fp)
```

```
{
char tmp = 0x1;
int val = 0;

while((tmp & 0x1) == 0x1)
{
if (fread(&tmp, sizeof(char), 1, fp) == 0)
{
if (feof(fp))
return(-1);
else
return(0);
}
val = (val << 7) + ((tmp >> 1) & 127);
}
return(val);
}

int vbytewrite(int number, FILE *fp)
{
char bytearray[4];
char tmp = 0;
int x, started = FALSE;
int charswritten = 0;

for(x=0;x<4;x++)
{
tmp = (number%128) << 1;
bytearray[x] = tmp;
number /= 128;
}
for(x=3;x>0;x--)
{
if (bytearray[x] != 0 || started == TRUE)
{
started = TRUE;
bytearray[x] |= 0x1;
fwrite(&bytearray[x], sizeof(char), 1, fp);
charswritten++;
}
}
bytearray[0] |= 0x0;
fwrite(&bytearray[0], sizeof(char), 1, fp);
charswritten++;
return(charswritten);
}
```

```
/* --- testgolomb.c --- */

/* VECTOR routine suite for variable-bit integer coding
 * Original vector and Elias coding Copyright (C) 1996 to Justin Zobel
 * This code, variable byte integers, and Golomb coding Copyright (C) 1996-8
 * to Hugh Williams
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * See Copyright.txt for further details. */

/* Code to test Golomb coding */

#include "def.h"

void
main(int argc, char *argv[])
{
    BITVEC vp2;
int b;
int x;

/* Initialise the "b" lookup table for Golomb coding. */
/* This is faster than calculating on-the-fly */
initgolhash();

/* Initialise the vector code */
initcmp;

/* Create a vector */
initvec(&vp2, BIGVECLEN);

/* Now, write the numbers 0 to 998 into the vector with weird b values */
/* using Golomb coding */
for (b=1000, x=0;x<999;x++, b--)
{
```

```
printf("Golomb putting: %d with b=%d\n", x, b);
fflush(NULL);
putgolomb(&vp2, x, b);
}

/* Now, let's put some Elias stuff in for good measure */
for (x=1;x<999;x++)
{
printf("Elias gamma putting: %d\n", x);
putgamma(&vp2, x);
}

for (x=1;x<999;x++)
{
printf("Elias delta putting: %d\n", x);
putdelta(&vp2, x);
}

/* Pad out the vector prior to writing */
padvec(&vp2);

/* reset the vector and read */
resetvec(&vp2);

printf("-------------\n");

/* Now, read the vector */
for (b=1000, x=0;x<999;x++, b--)
{
printf("Golomb getting: %d\n", getgolomb(&vp2, b));
fflush(NULL);
}

for (x=1;x<999;x++)
printf("Elias gamma getting: %d\n", getgamma(&vp2));

for (x=1;x<999;x++)
printf("Elias delta putting: %d\n", getdelta(&vp2));

/* Free the vector memory */
freevec(&vp2);
}
```

References

1. Witten et al. 1994, *Managing Gigabytes: Compressing and Indexing Documents and Images.* (New York, NY: Van Nostrand Reinhold).

0.10.3 Huffman Compression

Huffman compression was first proposed in a paper by D. A. Huffman in the early 1950's. Huffman coding is used in the `pack` compression program and a varient called dynamic huffman coding is used (along with LZ) in `freeze`.

Before compressing data, Huffman encoders first carefully analize the input stream. In the analysis, Huffman routines carefully tally character frequency data. That is, for each distinct character in the input (up to 256 character byte values) a count is stored.

```
void count_bytes(FILE *pfInput, unsigned long *puCounts)
{
  long lInputMarker;
  int c;

  //
  // Preconditions
  //
  ASSERT(pfInput);
  ASSERT(puCounts);

  //
  // Remember where the file pointer is now.
  //
  lInputMarker = ftell(pfInput);
  ASSERT(lInputMarker >= 0);

  //
  // Tally the characters from here to EOF
  //
  while ((c = getc(pfInput)) != EOF)
    counts[c]++;

  //
  // Put the pointer back where it was.
  //
  fseek(pfInput, lInputMarker, SEEK_SET );
  ASSERT(ftell(pfInput) == lInputMarker);
}
```

In an effort to keep counts to a reasonable size, the character frequency values are often scaled. For instance, it is often desireable to have counts fit in one byte of data. So, by searching for the highest count value, h and assigning it the value $0xFF$ then scaling the rest of the counts by $h/0xFF$ all count values should fit in one byte. In the source code I refer to scaled counts as "weights."

Next Huffman algorithms build a full binary tree which will be used to determine the codewords used to compress frequently encountered characters. To construct the tree, first the routine creates n small, one-node trees (where n is the number of distinct characters in the input stream). Each of these n trees represent a distinct input character and have a weight cooresponding to their count tallied in the analysis step.

The tree building process begins by selecting two nodes from the field of candidates. The two nodes selected are the ones with the lowest weights. These nodes are then joined into one tree. Each node becomes a leaf off of a newly created root node. The letter values and weights of the two nodes remain the same. The weight of the root node is set to the sum of the weights of its two leaves. This weight is the new weight of this three-node tree.

This process continues – the tree building process loops selecting the two trees (with anywhere from 1 to $(n-1)$ nodes) with lowest weight. The two trees selected are joined by a new root node, the root node's weight is set, and the new tree is placed back into the pool. The process repeats until one tree encompassing all the input weights has been constructed. If at any point there is more than one way to choose the two trees of smallest weight the algorithm chooses arbitrarily. This large tree is called a Huffman tree.

Once the Huffman tree has been constructed each letter can be assigned a codeword. The unique codeword for a given letter is computed by traversing the Huffman tree from the root node down to the leaf containing the letter and its weight. Because the Huffman tree is a binary tree, each node has, at most, two children. In the path from the root to a given leaf node, anytime the algorithm traverses to a left child a "0" is added to the codeword while the choice of a right child appends a "1" to the codeword. Due to the special nature of the Huffman tree, no codeword produced will every be the prefix of another. Also, letters appearing frequently in the input stream will be represented by small codewords such as 01, 11, 001 (etc...) while those that do not appear frequently will be represented by longer codewords.

Once a mapping has been created the encoding process can begin. During this operation the input stream is again scanned character by character. For each character read, the equivalent codeword is computed and appended to the output stream. In practice, a mapping between all characters and their codewords is usually computed before any encoding takes place. This can be accomplished recursively:

```
BOOL convert_tree_to_code(NODE *pnNodes, CODE *pcCodes,
  unsigned int iCodeSoFar, int iBits, int iNode)
{
  ASSERT(pnNodes);
  ASSERT(pcCodes);

  //
  // If this is a leaf node we are done recursing, assign code and pop stack
  //
```

```
  if (iNode <= END_OF_STREAM)
  {
    ASSERT(iBits);
    ASSERT(iCodeSoFar);

    //
    // Code
    //
    pcCodes[iNode].uCode = iCodeSoFar;

    //
    // Length of code
    //
    codes[iNode].iCodeBits = iBits;
    return;
  }

  //
  // Otherwise we are on an internal node and need to keep going
  //
  iCodeSoFar <<= 1;
  ASSERT((iCodeSoFar | 0) == iCodeSoFar);

  //
  // One more bit about to be added
  //
  iBits++;

  //
  // When going right, add a zero to the code so far..
  //
  convert_tree_to_code(pnNodes, pcCodes, iCodeSoFar, iBits,
       pnNodes[iNode].iLeftChild);

  //
  // When going left add a one..
  //
  convert_tree_to_code(pnNodes, pcCodes, iCodeSoFar | 1, iBits,
       pnNodes[iNode].iRightChild);
}
```

Because the process of computing codewords for each input character is slow, pre-computing helps speed up the algorithm. Since frequently used characters tend to be represented in less than 8 bits, the compressed output file tends to be smaller than the input.

In order to decode a Huffman compressed message the Huffman tree used to generate the codewords must be available to the decoding process. For this reason the encoding process often includes a

representation of the Huffman tree its output. Because the message is usually long, such overhead may be acceptable. However, an alternative is for encoder and decoder to agree on a preset encoding scheme based on average frequencies of material to be transmitted. This, however, will almost certainly lead to a less than optimal compression rate.

Because the Huffman algorithm must rescan the input stream twice it is usually slower than alternative algorithms. The process of scanning the input stream is an $O(n)$ operation (where n the input size). This occurs twice: once to create the frequency tables and once to encode the data.

The process of constructing a Huffman tree (which occurs between the two listed above) has a complexity based on the number of distinct characters in the input. To build one tree from c one-node trees, the selection process must execute $(c-1)$ times. This process must search the entire pool for a tree - which is an $\Theta(n/2)$ operation. However, pre-sorting the pool and inserting aggregate trees at the right place in the pool speeds up the selection process as it need only select the first two trees in the list – an $O(1)$ process.

Each time two trees are joined the node-creation routine must execute in order to create the root node joining the two trees. Therefore, it also executes $(c-1)$ times. The average number of nodes in a Huffman tree is approximately the number in a full binary tree with c leaves – $2^c - 1$.

In practice, Huffman encoding yields good compression results. However, it does not do as good a job as sliding window routines (based on LZ algorithms) for average input data. It also does not execute as quickly as LZ routines.

Source Code

As stated in the comments, these are modified versions of Mark Nelson and Jean-Loup Gailly's `huff.c` from their excellent publication. They rely on the bitwise-I/O package included with his book. However, you should be able to implement your own replacement fairly easily. See the References section for more information about *The Data Compression Book*.

```
/*--- Huffman.c ---*/

//
// This is huffman.c based on huff.c from Mark Nelson and Jean-Loup Gailly's
// "The Data Compression Book" and relies on the bitio library from the same.
// To use this code, make a struct called BIT_FILE and link against another
// object file with two routines:
//
//   int InputBit (BIT_FILE *)
//   void OutputBits (BIT_FILE *, int bitv, int length)
//
// Entrypoints to this module are CompressFile and ExpandFile.
//
// Or buy the book and use theirs.  One day I will get around to commenting,
// modifying and converting bitio.c but until then I do not want to put some-
// one elses' copyrighted code up on the web.  If you can't do this yourself
```

```
// then bother me and I may do it for you.  If you did it yourself, send me
// the file and I'll put it up here so others can use it.
//

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#include "bitio.h"
#include "debug.h"
#include "global.h"
#include "main.h"

//
// The NODE structure is a node in the Huffman decoding tree.  It has
// a count, which is its weight in the tree, and the node numbers of
// its two children.
//
typedef struct _NODE
{
  unsigned int uWeight;        // scaled byte count
  int iLeftChild;              // child zero
  int iRightChild;             // child one
} NODE;

//
// Since walking the Huffman tree is a slow process we only want to do it
// once for each symbol (not every time we see the symbol).  When encoding,
// first walk the tree for each symbol and save the codes in a code table.
// This is a code table entry.
//
typedef struct _CODE
{
  unsigned int uCode;
  int iCodeBits;
} CODE;

//
// The special EOS symbol is 256, the first available symbol after all
// of the possible bytes.  When decoding, reading this symbols
// indicates that all of the data has been read in.
//
#define END_OF_STREAM  256

//
// Local function prototypes, defined with ANSI rules only.
```

```
//
void compress_data(FILE *pfInput, BIT_FILE *pbfOutput, CODE *pcCodes);

BOOL output_counts(FILE *pfOutputFile, NODE *pnNodes);
BOOL input_counts(FILE *pfInputFile, NODE *pnNodes);
void count_bytes(FILE *pfInput, unsigned long *puCounts);
void scale_counts(unsigned long *puCounts, NODE *pnNodes);
int build_tree(NODE *pnNodes);
void convert_tree_to_code(NODE *pnNodes, CODE *pcCodes,
  unsigned int iCodeSoFar, int iBits, int iNode);




//
// This routine works in a fairly straightforward manner.  First, it
// has to allocate storage for three different arrays of data.  Next,
// it counts all the bytes in the input file.  The counts are all
// stored in long int, so the next step is scale them down to single
// byte counts in the NODE array.  After the counts are scaled, the
// Huffman decoding tree is built on top of the NODE array.  Another
// routine walks through the tree to build a table of codes, one per
// symbol.  Finally, when the codes are all ready, compressing the
// file is a simple matter.  After the file is compressed, the storage
// is freed up, and the routine returns.
//

BOOL CompressFile(FILE *pfInputFile, BIT_FILE *pbfOutputFile, int argc,
  char *argv[])
{
  unsigned long *puCounts = NULL;
  NODE *pnNodes = NULL;
  CODE *pcCodes = NULL;
  int iRootNode;
  BOOL fRetVal = FALSE;

  //
  // Allocate memory:
  //

  //
  // This will keep track of the count for each ASCII byte in the input
  // file.
  //
  puCounts = (unsigned long *) malloc( 256 * sizeof(unsigned long));
  if (NULL == puCounts)
  {
```

```
    fprintf(stderr, "Out of memory.\n");
    goto CompressReturn;
  }
  memset(puCounts, 0, 256 * sizeof(unsigned long));

  //
  // These will be the nodes on the Huffman tree; 514 is the largest number
  // we need because letter data will sit at leaf nodes.  Since in a tree
  // with n leaf nodes there will be (n-1) internal nodes and there are
  // 257 possible characters (256 byte values and 1 made up EOS character)
  // they will need 256 internal nodes.  That's 513 total nodes.
  //
  pnNodes = (NODE *) malloc (514 * sizeof(NODE));
  if (NULL == pnNodes)
  {
    fprintf(stderr, "Out of memory.\n");
    goto CompressReturn;
  }

  //
  // These will make up the byte -> code mapping table.  We need 257 because
  // we need one for every byte value plus one for the EOS character we made
  // up.
  //
  pcCodes = (CODE *) malloc(257 * sizeof(CODE));
  if (NULL == pcCodes)
  {
    fprintf(stderr, "Out of memory.\n");
    goto CompressReturn;
  }

  //
  // Run through the input stream and count the frequency of each byte.
  // Place the data in the counts table.
  //
  count_bytes(pfInputFile, puCounts);

  //
  // This scales the weights of characters counted in the count_bytes
  // routine down so as to limit the size of the Huffman codes (yet to
  // be generated) to at most 16 bits each.
  //
  scale_counts(puCounts, pnNodes);

  //
  // Write the counts to the output file header
  //
```

```
  if (!output_counts(pbfOutputFile->file, pnNodes ))
  {
    goto CompressReturn;
  }

  //
  // Build up the Huffman tree by combining the count trees
  //
  if (!(iRootNode = build_tree( pnNodes )))
  {
    goto CompressReturn;
  }

  //
  // Walk the tree for each character symbol and get a code value for
  // each one
  //
  convert_tree_to_code(pnNodes, pcCodes, 0, 0, iRootNode);

  //
  // Do the actual compression
  //
  compress_data( pfInputFile, pbfOutputFile, pcCodes );

  //
  // We made it!
  //
  fRetVal = TRUE;

CompressReturn:

  //
  // Cleanup
  //
  if (puCounts) free(puCounts);
  if (pnNodes)  free(pnNodes);
  if (pcCodes)  free(pcCodes);

  return(fRetVal);
}



BOOL ExpandFile(BIT_FILE *pbfInputFile, FILE *pfOutput, int argc, char *argv[])
{
  NODE *pnNodes;
  int iRootNode;
```

```
  //
  // Allocate memory for the tree -- 514 is the max sizeof the tree because
  // there are at most 257 leaf nodes and therefore max 256 internal nodes
  // and we use one node position to store a HUGE tree while building it.
  //
  if ((pnNodes = (NODE *) malloc(514 * sizeof(NODE))) == NULL)
  {
    fprintf(stderr, "Out of memory.\n");
    return(FALSE);
  }

  //
  // This routine reads the header count table from the compressed image
  //
  if (!input_counts(pbfInputFile->file, pnNodes))
  {
    free(pnNodes);
    return(FALSE);
  }

  //
  // Build the Huffman tree from the table we read in the last step.
  //
  iRootNode = build_tree(pnNodes);

  //
  // Do the actual expansion of the compressed file.
  //
  expand_data(pbfInput, pfOutput, pnNodes, iRootNode);

  //
  // Cleanup and return
  //
  free(pnNodes);
  return(TRUE);
}
*/


//
// In order for the compressor to build the same model, I have to store
// the symbol counts in the compressed file so the expander can read
// them in.  In order to save space, I don't save all 256 symbols
// unconditionally.  Instead store only characters with a positive count
// in the format:
//
```

```
// char, count, char, count, ... 0, 0 (beginning of compressed data)
//
// The code from "The Data Compression Book" stores them in a different
// (more efficient) format.  Unfortunately, I don't think the extra space
// savings are worth complicating the code.  See the source for a better
// way of doing this.
//
BOOL output_counts(FILE *pfOutputFile, NODE *pnNodes )
{
  int iFirst = 0;
  int i;

  //
  // Write each non-zero count
  //
  for (iFirst = 0; iFirst < 256; iFirst++)
  {
    if (pnNodes[iFirst].uWeight != 0)
    {
      if (putc(iFirst, pfOutputFile) != iFirst)
      {
perror("putc");
return(FALSE);
      }

      if (putc(pnNodes[iFirst].uWeight, pfOutputFile) !=
  pnNodes[iFirst].uWeight)
      {
perror("putc");
return(FALSE);
      }

    }
  }

  //
  // Write the 0 0 terminator part
  //
  for (iFirst = 0; iFirst < 2; iFirst++)
  {
    if (putc(0, pfOutputFile) != 0)
    {
      perror("putc");
      return(FALSE);
    }
  }
```

```
  return(TRUE);
}




//
// Read the counts from a compressed file header
//
BOOL input_counts(FILE *pfInputFile, NODE *pnNodes)
{
  int i;
  int iChar;
  int iCount;

  //
  // Initialize all counts to zero
  //
  for (i = 0; i < 256; i++)
  {
    pnNodes[i].uWeight = 0;
  }

  while(1)
  {

    //
    // Read the character
    //
    if ((iChar = getc(pfInputFile)) == EOF)
    {
      fprintf(stderr, "Error reading byte counts\n");
      return(FALSE);
    }

    //
    // Read the count (hi and low byte)
    //
    if ((iCount = getc(pfInputFile)) == EOF)
    {
      fprintf(stderr, "Error reading byte counts\n");
      return(FALSE);
    }

    if (iCount)
    {
      pnNodes[iChar].uWeight = (unsigned) iCount;
```

```
    }
    else
    {
      break;
    }
  }
  pnNodes[END_OF_STREAM].uWeight = 1;
  return(TRUE);
}


//
// This routine counts the frequency of occurence of every byte in
// the input file.  It marks the place in the input stream where it
// started, counts up all the bytes, then returns to the place where
// it started.  In most C implementations, the length of a file
// cannot exceed an unsigned long, so this routine should always
// work.
//
// This routine could possible be sped up by buffering the input file
// and counting characters in the buffer.  Hopefully your operating
// system is doing some buffering behind your back, though...
//
void count_bytes(FILE *pfInput, unsigned long *puCounts)
{
  long lInputMarker;
  int c;

  //
  // Preconditions
  //
  ASSERT(pfInput);
  ASSERT(puCounts);

  //
  // Remember where the file pointer is now.
  //
  lInputMarker = ftell(pfInput);
  ASSERT(lInputMarker >= 0);

  //
  // Tally the characters from here to EOF
  //
  while ((c = getc(pfInput)) != EOF)
    puCounts[c]++;

  //
```

```
  // Put the pointer back where it was.
  //
  fseek(pfInput, lInputMarker, SEEK_SET );
  ASSERT(ftell(pfInput) == lInputMarker);
}


//
// In order to limit the size of my Huffman codes to 16 bits, I scale
// my counts down so they fit in an unsigned char, and then store them
// all as initial weights in my NODE array.  The only thing to be
// careful of is to make sure that a node with a non-zero count doesn't
// get scaled down to 0.  Nodes with values of 0 don't get codes.
//
void scale_counts(unsigned long *puCounts, NODE *pnNodes)
{
  unsigned long uMaxCount;
  int i;

  //
  // Preconditions
  //
  ASSERT(puCounts);
  ASSERT(pnNodes);

  //
  // Run through the counts and look for the one with the greatest weight.
  //
  uMaxCount = 0;
  for (i = 0; i < 256; i++)
  {
    if (puCounts[i] > uMaxCount)
      uMaxCount = puCounts[i];
  }

  //
  // If there were no characters in the count table (i.e. we are trying
  // to compress an empty file) make the max count one so the scaling
  // formula works right.
  //
  if (uMaxCount == 0)
  {
    puCounts[0] = 1;
    uMaxCount = 1;
  }

  //
```

```
  // Now create the node weights for each symbol in the input stream --
  // use the counts scaled down by 1 / uMaxCount
  //
  uMaxCount /= 255;
  uMaxCount += 1;
  for (i = 0; i < 256; i++)
  {
    pnNodes[i].uWeight = (unsigned int) (puCounts[i] / uMaxCount);

    //
    // As the comment stated, make sure we never scale too much such that
    // a non-zero count byte achieves a scaled weight of zero.
    //
    if ((pnNodes[i].uWeight == 0) && (puCounts[i] != 0))
    {
      pnNodes[i].uWeight = 1;
    }
  }

  //
  // Special end of stream symbol
  //
  pnNodes[END_OF_STREAM].uWeight = 1;
}



//
// Building the Huffman tree is fairly simple.  All of the active nodes
// are scanned in order to locate the two nodes with the minimum
// weights.  These two weights are added together and assigned to a new
// node.  The new node makes the two minimum nodes into its 0 child
// and 1 child.  The two minimum nodes are then marked as inactive.
// This process repeats until their is only one node left, which is the
// root node.  The tree is done, and the root node is passed back
// to the calling routine.
//
// Node 513 is used here to arbitratily provide a node with a guaranteed
// maximum value.  It starts off being min_1 and min_2 and then we look
// for other nodes two other nodes that have smaller values.  After all
// active (i.e. non-zero weight) nodes have been scanned, we can tell if
// there is really only one active node left in the pool by checking to
// see if one of the min pointers is still set to 513 (the huge node).
//
int build_tree(NODE *pnNodes)
{
  int iNextFree;
```

```
  int i;
  int iMin1;
  int iMin2;

  ASSERT(pnNodes);

  //
  // This node is guaranteed max value.
  //
  pnNodes[513].uWeight = 0xffff;

  //
  // NextFree will be used to store positions of internal nodes.  Start
  // at 257 (0..256 may be character leaves, 257 is our little end of
  // stream symbol) and work up.
  //
  for (iNextFree = END_OF_STREAM + 1 ; ; iNextFree++)
  {
    ASSERT(iNextFree < 513);

    //
    // The minimum we have found so far (max possible weight value)
    //
    iMin1 = 513;
    iMin2 = 513;

    //
    // This could be done more efficiently by sorting
    //
    for (i = 0; i < iNextFree; i++)
    {
      if (pnNodes[i].uWeight != 0)
      {
if (pnNodes[i].uWeight < pnNodes[iMin1].uWeight)
        {
  iMin2 = iMin1;
  iMin1 = i;
}
else if (pnNodes[i].uWeight < pnNodes[iMin2].uWeight)
{
  iMin2 = i;
}
      }
    }

    //
    // Is there only one tree (the tree and the fat node at 513)?
```

```
    //
    if (iMin2 == 513)
    {
      break;
    }

    //
    // Otherwise combine trees
    //
    pnNodes[iNextFree].uWeight = pnNodes[iMin1].uWeight;
    pnNodes[iNextFree].uWeight += pnNodes[iMin2].uWeight;

    //
    // These two nodes no longer in consideration for smallest...
    //
    pnNodes[iMin1].uWeight = 0;
    pnNodes[iMin2].uWeight = 0;

    pnNodes[iNextFree].iLeftChild = iMin1;
    pnNodes[iNextFree].iRightChild = iMin2;
  }

  //
  // Subtract one to get last used -- the root node
  //
  iNextFree--;
  return(iNextFree);
}



//
// Since the Huffman tree is built as a decoding tree, there is
// no simple way to get the encoding values for each symbol out of
// it.  This routine recursively walks through the tree, adding the
// child bits to each code until it gets to a leaf.  When it gets
// to a leaf, it stores the code value in the CODE element, and
// returns.
//
// This is a really cool routine... check out how it works.
//
void convert_tree_to_code(NODE *pnNodes, CODE *pcCodes,
  unsigned int iCodeSoFar, int iBits, int iNode)
{
  //
  // Preconditions
  //
```

```
  ASSERT(pnNodes);
  ASSERT(pcCodes);

  //
  // If this is a leaf node we are done recursing, assign code and pop stack
  //
  if (iNode <= END_OF_STREAM)
  {
    ASSERT(iBits);
    ASSERT(iCodeSoFar);

    //
    // Code
    //
    pcCodes[iNode].uCode = iCodeSoFar;

    //
    // Length of code
    //
    pcCodes[iNode].iCodeBits = iBits;
    return;
  }

  //
  // Otherwise we are on an internal node and need to keep going
  //
  iCodeSoFar <<= 1;
  ASSERT((iCodeSoFar | 0) == iCodeSoFar);

  //
  // One more bit about to be added to the length
  //
  iBits++;

  //
  // When going right, add a zero to the code so far..
  //
  convert_tree_to_code(pnNodes, pcCodes, iCodeSoFar, iBits,
       pnNodes[iNode].iLeftChild);

  //
  // When going left add a one..
  //
  convert_tree_to_code(pnNodes, pcCodes, iCodeSoFar | 1, iBits,
       pnNodes[iNode].iRightChild);
}
```

```
//
// Once the tree gets built, and the CODE table is built, compressing
// the data is a breeze.  Each byte is read in, and its corresponding
// Huffman code is sent out.
//
void compress_data(FILE *pfInput, BIT_FILE *pbfOutput, CODE *pcCodes)
{
  int ch;               // used for reading data byte by byte from input

  //
  // Preconditions
  //
  ASSERT(pfInput);
  ASSERT(pbfOutput);
  ASSERT(pcCodes);

  //
  // For each character in input, output the equivalent code
  //
  while ((ch = getc(pfInput)) != EOF)
  {
    OutputBits(pbfOutput, (unsigned long) pcCodes[ch].uCode,
       pcCodes[ch].iCodeBits);
  }

  //
  // EOS mark -- needed in the expansion process
  //
  OutputBits(pbfOutput, (unsigned long) pcCodes[END_OF_STREAM].uCode,
     pcCodes[END_OF_STREAM].iCodeBits);
}



//
// Expanding compressed data is a little harder than the compression
// phase.  As each new symbol is decoded, the tree is traversed,
// starting at the root node, reading a bit in, and taking either the
// iLeftChild or iRightChild path.  Eventually, the tree winds down to a
// leaf node, and the corresponding symbol is output.  If the symbol
// is the END_OF_STREAM symbol, it doesn't get written out, and
// instead the whole process terminates.
//

void expand_data(BIT_FILE *pbfInput, FILE *pfOutput, NODE *pnNodes,
 int iRootNode)
```

```
{
  int iNode;                  // the index of the node in the huffman tree for
                              // the current input symbol.

  //
  // Preconditions
  //
  ASSERT(pbfInput);
  ASSERT(pfOutput);
  ASSERT(pnNodes);

  while (1)
  {
    iNode = iRootNode;

    //
    // We will read the input file bit by bit and move the node pointer one
    // level in the tree for each bit read.  We will stop when we get to a
    // leaf node, write the uncompressed character to output, reset the node
    // pointer to the root, and repeat the process.
    //

    do
    {

      //
      // If we read a set bit (i.e. a "1") then traverse right else
      // it's a "0" so left in the Huff tree..
      //
      if (InputBit(pbfInput))
iNode = pnNodes[iNode].iRightChild;
      else
iNode = pnNodes[iNode].iLeftChild;


    }
    while (iNode > END_OF_STREAM);
    //
    // i.e. while we're pointing to an internal node.. stop at a leaf.
    //

    //
    // We are now at a leaf node.  Determine if this is a real symbol or
    // the end of file (represented by a made up EOS character).
    //
    if (iNode == END_OF_STREAM)
      break;
```

```
    //
    // It's a real thing... put it.
    //
    if ((putc(iNode, pfOutput)) != iNode)
      fprintf(stderr, "Error writing to output file!\n");
  }
}
```

References

1. Binstock, Andrew and Rex, John. 1995, *Practical Algorithms for Programmers* (Reading, MA: Addison-Wesley), pp. 490-500.

2. Nelson, Mark and Gailly, Jean-Loup. 1996, *The Data Compression Book* (New York, NY: M&T Books), pp. 31-74.

0.10.4 Adaptive Huffman Compression

If there are x levels in a Huffman tree, read the node values on each level l_x from left to right. When you have exhausted level l_x move up a level to l_{x-1} and repeat the process. Continue until you have moved up to the root node (and, thus, have read the value of every node in the tree).

You should notice an interesting pattern – the values always remain the same or increase. This is called the **sibling property** of Huffman trees. It tell us that given a node n, its sibling $s(n)$ is the node on the same level as n to the right. If n is the rightmost node on its level, $s(n)$ is the leftmost node on previous level. If v_n is the value of node n, we know that $v_{s(n)}$, the value of the sibling, will be greater than or equal to v_n. If you understand this property understanding the rest of this algorithm is a breeze.

One of the drawbacks of static Huffman compression algorithms is the need to transmit the character frequency tally with the compressed text. While an intelligently encoded table only adds about 250 bytes (on average) to a compressed image, it would be nice to get rid of it all together. This seems to be impossible because without the table's data the decompression routine would not know the structure of the Huffman tree used to encode the compressed data and therefore would not be able to ascertain the proper codeword to character mappings.

However, adaptive Huffman compression algorithms overcome the need to store the character counts by beginning with a mostly empty tree. They then build up and fill in the tree as they go. The Huffman trees generated by adaptive algorithms are dynamic meaning they change in structure as the statistical tendancies of the text change. The compression function adds each new character encountered to the tree. The algorithm does not, however, compress the character the first time it is seen. Instead the character is passed on to the compressed text as is (in fact, a special flag character is pre-pended to it, as we will discuss in the next paragraph). When the compression system sees a character already

present in the tree, it increments that character's count by one, adjusts the tree accordingly, and uses the compression code in the output stream.

The complimentary decompression routine operates in much the same manner – when a new character is encountered it is added to the dynamic Huffman tree but otherwise left alone. When a code is found it is decoded and the weight of the cooresponding character is increased. This increase may cause the tree to be reorganized.

The way that plain (non-compressed) characters and (compressed) codewords are distinguished in the compressed stream is by use of a flag or **escape** character. This symbol, when encountered, signifies that the next byte is a literal character. All other data is assumed to be codes. The escape symbol is one of the only items present in the initial Huffman tree. When the decompression subroutine runs across the code for an escape symbol it immediately reads a byte from the input, adds it to the tree, and sends it to output. Note that I am not talking about the escape character here (ASCII 27) but rather a made-up symbol that has a node on the Huffman tree. This symbol, of course, produces no output in the uncompressed stream – its sole function is to convey a message from the compression routine to the decompression routine about the character following it in the stream.

The complicated part of this algorithm, as you might expect, is the tree manipulation. It is unreasonable to reconstruct the entire Huffman tree every time a symbol is added to it. But recall from the opening paragraph of this section that all Huffman trees must obey the sibling property.

Imagine the steps of incrementing a Huffman node's weight: first, since all nodes are stored at the leaves of the tree, we will add one to the count of a leaf. We must now make sure the tree follows the sibling property. If our incremented leaf, i, has a larger value then its sibling then the sibling rule is broken.

When the sibling property has been broken matters can be fixed by swaping the offending incremented node i with its sibling. This will not always work, though. Imagine that there is a node n with value 5. Its immediate sibling is node o with value 5 also. The immediate sibling of p is 5 also. (That is, there are three leaf nodes in a row, all value 5). Now we increment node n to 6. The sibling property is broken because now n has a larger value than its right sibling, o (which is still 5). Swapping the two would be a problem because n is also larger than o's sibling, p.

The proper way to restore the sibling property when it has been violated by an incrementation is to loop over the siblings starting with the immediate sibling of the incremented node. Continue to loop while the value of the nodes encountered stays the same. Break out of the loop when the value changes. Swap the incremented node with the last node in the run of same values:

```
...
node[i].value++;
if (node[i].value > node[i+1].value)
{
  val = node[i+1].value;
  j = i+1;
  while (node[j].value == val)
  {
    j++;
  }
```

```
  swap (node[i], node[j-1]);
}
...
```

The above code assumes, of course, that we can traverse along siblings by simply moving to adjacent positions in a node array. In order to implement an adaptive Huffman algorithm it should be easy to find the sibling of a given node many times in a row.

References

1. Nelson, Mark and Gailly, Jean-Loup. 1996, *The Data Compression Book* (New York, NY: M&T Books), pp. 31-74.

0.10.5 Sliding Window Compression

Recall the RLE (run-length encoding) is a compression algorithm that represents long runs of a single character more efficiently. Sliding window compression is an algorithm that represents *groups of characters* that occur frequently more efficiently. For instance, "can count countville count the countably infinite count" repeats the string "count" five times. It would not be compressed at all with RLE since no characters repeat. But with sliding window, the phrase "count" would be represented with a small code.

Sliding window compression is a **dictionary-based** compression algorithm based on the work of Lempel and Ziv in 1977. Dictionary-based compression algorithms maintain a group of strings from the input stream as the encoding process executes. This group of strings is called a dictionary and can be used to abbreviate recurring patterns in the text. If the algorithm spots a string in the input stream that it has seen already and has stored as part of the dictionary, the string can be represented in a more efficient manner. Usually dictionary entries are denoted with a two-byte code that points to its entry number in the dictionary and its size. During the decompression process the routine creates and maintains a dictionary with the same rules as were used in the compression process. So entry n in the dictionary is the same at compression and decompression time.

Before implementing a sliding window compression scheme the size (number of possible entries) of the dictionary must be fixed. More dictionary entries mean a greater chance that a repeated string will be found in the dictionary text and therefore compressed. However, more dictionary entries also lead to longer dictionary codes. Typical sliding window algorithms use a dictionary size of $2^{12} = 4096$ entries. The dictionary code is, therefore, always a 12 bit number.

Since some words found in the input stream will have no exact matches in the dictionary but will be prefixes of dictionary words, most sliding window algorithms choose to add a length code to the dictionary code. For instance, imagine "eyeball" is in the dictionary at position 255. The word "eye" is not in the dictionary at all. The word "eye" now appears in the input text. If we use twelve bits to represent the dictionary entry for "eyeball" (000001111111) and use four bits to say "use only the first three letters of entry 255" (0011) then we can represent "eye" as two bytes instead of three (0000011111110011). The choice of twelve bit entry codes and four bit prefix codes make the amount

of data needed to represent any dictionary entry two bytes. These choices also limit the number of dictionary entries to $2^{12} = 4096$ and the max length of a dictionary string to $2^4 = 16$ bytes. Since, however, we will never encode a one or two byte sequence, this kicks the size limit of the dictionary up to $16 + 2 = 18$ bytes.

You may be wondering how we can distinguish encoded dictionary pointers from raw (unencoded) text. The answer is to use accounting bytes. Before every sequence of eight bytes in the compressed text we use a **flag byte**. This byte contains no data from the input stream – rather it is used to tell the decompresser which bytes in the following eight are raw text and which contain dictionary compression codes. For instance imagine we have the string "invite him!" in the input stream. The "invit" is encoded using a dictionary code of 0000011111110101 – use the first five bytes of entry 255, which was "invitation". The rest of the text cannot be found in the dictionary. It is added to the dictionary for future processing but is encoded as raw "e him!". We would write the following eight bytes to the output stream: (0000011111110101)=dictionary code for "invit" (01100101)="e" (00100000)=space (01101000)="h" (01101001)="i" (01101101)="m" (00100001)="!". However, before these eight bytes we would write the flag byte 00111111. The 0's tell the decoder that the first two of the next eight bytes are dictionary codes whereas the next six are raw text. The use of flag bytes, of course, decreases the efficiency of the compression.

Given this information we can determine the best and worst case compression rates of sliding window compression. Since we are using flag bytes, if no compression is possible we will encode nine bytes in the output for every eight bytes of input. Therefore, the worst case for this algorithm increases the file size to 112.5 percent of the original. If every character sequence is represented with a dictionary entry of maximum length (18 bytes) then each 18 byte sequence will be represented by 16 bits + 1 bit (in the flag byte). That means the best possible compression will see an output file that is 11.8 percent the size of the original.

It is inefficient to search the whole previous window of text for matches. The way most implementations of sliding window algorithms handle this is by keeping the dictionary in a data structure such as a binary search tree. That way it is easy to search for newly read text in the dictionary – just traverse the binary tree. Remember that new text might only partially match text in the dictionary, though. We want the node on the root-to-leaf traversal that best matches the new text. It is also possible that the new text will exactly match a node in the tree. In this case the node must be replaced with the new data and its offset. Likewise, as text falls out of the window it is important to delete the cooresponding node. Deletion in a binary search tree is a little tricky so check the BST section for a full explaination. Studies have shown that it is not a good idea to use a more complicated data structure such as a red-black tree or an AVL tree. While these data structures will be better balanced than a BST, in most cases, the extra time spent keeping them balanced slows down the compression. Due to the dynamic nature of the tree, caused by phrases being continually added to and deleted from the dictionary, highly unbalanced trees should quickly correct themselves.

Source Code

Below is an implementation of a program to decompress Microsoft style sliding window compression files (program.ex_). I do not have the complimentary sliding window compression program – if you do, please send it.

```
/*--- slide.h ---*/

#ifndef _SLIDE_
#define _SLIDE_

//
// Compressed file header
//
typedef struct _HEADER
{
  DWORD dwMagic1;
  DWORD dwMagic2;
  BYTE bIs41;
  CHAR chFileFix;
  DWORD dwUnCompSize;
} HEADER;

//
// Microsoft's magic numbers for compressed file headers
//
#define MAGIC1 0x44445A53
#define MAGIC2 0x3327F088

//
// Misc other
//

//
// 2^12 -> 12 bit offset size
//
#define WINSIZE 4096

//
// Compression code:
//
//      bByte1:            bByte2:
// 0 1 2 3 4 5 6 7 | 0 1 2 3 4 5 6 7
// [       OFFSET DATA      ][LENGTH]
//          (12 bits)        (4 bits)
//

//
// How to extract the 4-bit length code from the 2-byte compression code
// the +3 part makes 0, 1 and 2 byte lengths impossible (since we won't ever
// use them) and allows 16, 17 and 18 byte values from a 4 bit width field.
// Pass only the second byte since it's all in there...
//
```

```
#define LENGTH(x) ((((x) & 0x0F)) + 3)

//
// How to extract the 12-bit offset code from the 2-byte compression code
// Pass both bytes.
//
#define OFFSET(x1, x2) ((((x2 & 0xF0) << 4) + (x1) + 0x0010) & 0x0FFF)

//
// Given a position in the file, xlate it into a position in the window.
//
#define WRAPFIX(x) ((x) & (WINSIZE - 1))

//
// Is a certain bit set in a byte.
//
#define BITSET(byte, bit) (((byte) & (1<<(bit))) > 0)

#endif
```

```
/*--- decode.c ---*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "global.h"
#include "debug.h"
#include "slide.h"

//
// 4K window
//
BYTE rgbWindow[WINSIZE];


//
// UncompressData - given an input file, an output file, and the output file
//                  length, decompress the input file.
//
void UncompressData (FILE *pfInFile, FILE *pfOutFile, ULONG uUnCompSize)
{

  BYTE bBitMap, bByte1, bByte2;
  DWORD dwLength, dwCounter, dwLocation;
  ULONG uCurrPos = 0L;
```

```
  //
  // Initialize the window
  //
  memset(rgbWindow, 0, WINSIZE);

  //
  // Process file
  //
  while (uCurrPos < uUnCompSize)
  {
    //
    // Get flag byte to see which is codes and which is raw data
    //
    bBitMap = fgetc(pfInFile);
    if (feof(pfInFile)) return;

    //
    // Decode next eight data items (compression codes or raw data)
    //
    for (dwCounter = 0; dwCounter < 8; dwCounter++)
    {

      //
      // If a bit in the flag byte is clear, the cooresponding byte is
      // a compression code... treat it as such.
      //
      if (!BITSET(bBitMap, dwCounter))
      {

//
// Read the two byte code
//
bByte1 = fgetc(pfInFile);
if (feof(pfInFile)) return;
bByte2 = fgetc(pfInFile);

//
// Extract length and offset info from the 16 bits of data
//
dwLength = LENGTH(bByte2);
dwLocation = OFFSET(bByte1, bByte2);
ASSERT(dwLength);

//
// Copy data from the window
//
```

```
while (dwLength > 0)
{
  //
  // Get a byte from the window
  //
  bByte1 = rgbWindow[WRAPFIX(dwLocation)];

  //
  // Put the byte in the window
  //
  rgbWindow[WRAPFIX(uCurrPos)] = bByte1;

  //
  // Put the byte in the output stream
  //
  fputc(bByte1, pfOutFile);

  //
  // Update counters
  //
  uCurrPos++;
  dwLocation++;
  dwLength--;
}

      }

      //
      // Just a raw data byte
      //
      else
      {

//
// Read the byte from the input stream
//
bByte1 = fgetc(pfInFile);

//
// Save the byte in the window
//
rgbWindow[WRAPFIX(uCurrPos)] = bByte1;

//
// Put the byte to the output stream
//
fputc(bByte1, pfOutFile);
```

```
//
// Update count
//
uCurrPos++;
      }

      if (feof(pfInFile)) return;

    } // for

  } // while

} // UnCompressData




//
// VerifyHeader - given an input file, verify magic numbers in the header
//                and extract the uncompressed file size to return.
//
ULONG VerifyHeader (FILE *pfInFile)
{
  HEADER header;
  ULONG uCompSize;

  //
  // Seek to the end of the compressed file and see how long it is.
  //
  fseek(pfInFile, 0, SEEK_END);
  uCompSize = ftell(pfInFile);

  //
  // Seek to the beginning and read the header.
  //
  fseek(pfInFile, 0, SEEK_SET);
  fread(&header, sizeof(HEADER), 1, pfInFile);

  //
  // Make sure the magic numbers in the header are correct
  //
  if ((header.dwMagic1 != MAGIC1) ||
      (header.dwMagic2 != MAGIC2))
  {
    fprintf(stderr, "Error: bad file header.");
    exit(1);
  }
```

```
  printf("Uncompressing from %lu bytes to %lu bytes.\n",
 uCompSize, header.dwUnCompSize);

  //
  // Return the uncompressed size from the header
  //
  return(header.dwUnCompSize);
}

int main(int argc, char *argv[])
{
  FILE *pfInFile, *pfOutFile;
  ULONG uUncompressedSize;

  if (argc != 3)
  {
    fprintf(stderr, "Usage: %s compressed-file uncompressed-file\n",
    argv[0]);
    exit(1);
  }

  if ((pfInFile = fopen(argv[1], "rb")) == NULL)
  {
    perror(argv[1]);
    exit(1);
  }

  if ((pfOutFile = fopen(argv[2], "wb")) == NULL)
  {
    perror(argv[2]);
    exit(1);
  }

  uUncompressedSize = VerifyHeader(pfInFile);
  UncompressData(pfInFile, pfOutFile, uUncompressedSize);

  fclose(pfInFile);
  fclose(pfOutFile);
  exit(0);
}
```

References

1. Binstock, Andrew and Rex, John. 1995, *Practical Algorithms for Programmers* (Reading, MA: Addison-Wesley), pp. 500-510.

2. Nelson, Mark and Gailly, Jean-Loup. 1996, *The Data Compression Book* (New York, NY: M&T Books), pp. 31-74.

0.10.6 Lempel-Ziv-Walsh Compression

0.10.7 Arithmetic Compression

0.11 Game-Playing Algorithms

0.11.1 Minimax Search

Minimax is a game-playing algorithm which is used to search a game tree. In games where opponents alternate taking turns which affect a board position (chess, checkers, etc...), a given position can be thought of as a node on a tree. All positions reachable from a given position are, therefore, children nodes on the game tree. Minimax recursively evaluates positions on a tree with the intent of selecting the best move for a given player in a given position. In order for a minimax routine to work a function that maps a board position into a "score" is needed. In two-player games often this evaluation function returns real values between -1 and 1. A value of -1 means that one side has won outright, 0 indicates an even position, and 1 is returned when the other side has achieved victory.

0.11.2 Alpha-Beta pruning

0.12 Data Integrity

0.12.1 Checksums

A **checksum** is a device used to establish the integrity of data. More specifically, a checksum is a value which is calculated based on some data which, later, can be used to verify that the data used to generate it remains unchanged. Usually this is accomplished by adding the checksum to the sum of the data values and verifying the result.

For example, the Intel object file specification calls for a one byte checksum at the end of every record in the file. This checksum is stored in the last byte of each record and is defined to be the two's compliment of the sum of the values of all other bytes in the record modulo 256. That is, the sum of all bytes in a record modulo 256 added to the checksum byte's value should yield a value of zero. If it does not, either the checksum byte or some other byte in the record must have been corrupted.

This is, of course, not a foolproof method of detecting corruption. If two bytes in an Intel object file are changed in such a way as to compliment each other the corruption will remain undetected by the checksum.

In order to increase the error detection ability of a checksum algorithm more than one checksum value can be generated. Further, what is known as a **weighted checksum** can be computed instead.

Source Code

The first piece of code on this page is a typical checksum generation routine that takes a pointer to the data in question and returns a checksum value which, when added to the data, will yield a multiple of ten.

```
int checksum (char *data, int data_length) {

  int sum, i = 0;

  while (i < data_length) {
    sum += *data++;
    i++;
  }
  return(10 - (sum % 10));
}
```

The following code was Bill Sasina. It takes as input an Intel hex file and computes a checksum for it. While the utility of this code is somewhat limited, it demonstrates the idea of how checksums work very well. This code is not covered by the copyright statement in the introduction of this document; it was donated by the author to the public domain.

```
/*
                Checksum Calculator Utility
Written by Bill Sasina, 1995
*/

#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <errno.h>
#include <stdlib.h>

/*the system error list is known at link time*/
extern char *sys_errlist [];

/*prototype definitions --*/
void main (int, char **);
```

```
void sum_it (long int *, char *, long int *);
char *fgetstr (char *, int, FILE *);
void errexit (unsigned);
long int prompt_for_size(void);

/* Global Variables */
int line_cnt = 0;

/*Main - open the first and second arguments and proceed as in Prog1*/
void main (int argc, char *argv[])
{
FILE *input;
char string[256];
long int cksum1 = 0;
long int cksum2 = 0;
long int byte_cnt = 0;
long int prom_size;
char *junk;

clrscr();
printf("%18c Bill's Checksum Utility  Version 2.2 \n\n", ' ');

if (argc < 2 || argc > 3)
{
errexit(1);
}

if (argc == 3)
{
prom_size = ((strtol(argv[2], &junk, 10)) * (long) 1024) - 1;
}
else
{
prom_size = prompt_for_size();
}

/*get input file*/
if ((input = fopen (argv[1], "r")) == 0)
{
errexit(2);
}

printf("\n%s\t", argv[1]);

while ((fgetstr(string, 255, input)) && (byte_cnt < prom_size))
{
line_cnt++;
```

```
sum_it(&cksum1, string, &byte_cnt);
}

cksum2 = cksum1;

if (byte_cnt < prom_size)
{
for ( ; byte_cnt <= prom_size; byte_cnt++)
{
cksum1 += 255;
}
}

printf("\n\nFill Character\t FF\t\t 00\n");
printf("Checksum =\t%4X\t\t%4X\n", (int) cksum1, (int) cksum2);

if (fclose(input))
{
errexit (6);
}

/*exit normally*/
exit (0);
}


/*sum_it - Calculate the checksum */

void sum_it (long int *sum, char *stringptr, long int *bytes)
{
int i;
long len = 0;
long line_sum = 0;
long temp = 0;
char tempstr[] = {"0XFF"};
char *junk;

if (*stringptr == 'S')
{
errexit(4);
}

if (*stringptr == ':')
{
stringptr++;
}
```

```
tempstr[2] = *stringptr++;
tempstr[3] = *stringptr++;

len = strtol(tempstr, &junk, 16);
line_sum = len;

for (i = 0; i < 3; i++)
{
tempstr[2] = *stringptr++;
tempstr[3] = *stringptr++;

temp = strtol(tempstr, &junk, 16);
line_sum += temp;
}

if ( ( temp != 2 ) && ( temp != 3 ) )
{
*bytes += len;
}
else
{
if ( ( *stringptr != '0' ) && ( *stringptr != '1' ) )
{
printf("\nWarning\t--\tFile uses extended addressing.");
printf("\nOffset\t=\t%c%c%c%c",
   stringptr[0], stringptr[1], stringptr[2], stringptr[3]);
}
}

for ( ; len; len--)
{
tempstr[2] = *stringptr++;
tempstr[3] = *stringptr++;

line_sum += strtol(tempstr, &junk, 16);
if ( ( temp != 2 ) && ( temp != 3 ) )
{
*sum += strtol(tempstr, &junk, 16);
}
}
tempstr[2] = *stringptr++;
tempstr[3] = *stringptr++;

if (((unsigned char) (~line_sum + 1)) !=
((unsigned char) strtol(tempstr, &junk, 16)))
{
printf("Line number %d", line_cnt);
```

```
printf(" %X %X\n", (~((unsigned char) line_sum) + 1),
   (unsigned char) strtol(tempstr, &junk, 16));
errexit(3);
}
}

/*fgetstr - 'gets' does not return '\n' -- 'fgets' does.
            this routine makes 'fgets' like 'gets'*/
char *fgetstr (char string[], int n, FILE *filptr)
{
char *retval, *ptr;
if ((retval = fgets (string, n, filptr)) != NULL)
{
for (ptr = string; *ptr; ptr++)
{
if (*ptr == '\n')
{
*ptr = '\0';
break;
}
}
}
return retval;
}


/*Errexit - handle errors as they arise*/
char *errlist[] =
{"invalid error",

 "wrong number of arguments."
 "\n Try: chksum <input_file> <prom_size>",

 "input file does not exist",
 "checksum error in HEX file",
 "File is a Motorola S-Record File.",
 "error on output file write",
 "error on closing input file",
 "error on closing output file",
 "debug error"};

void errexit (unsigned errnum)
{
if (errnum > 7)
{
errnum = 7;
}
```

```
fprintf (stderr, "\nchecksum utility error:  %s\n"
 "system error:          %s\n",
 errlist[errnum],
 sys_errlist [errno]);
exit (errnum);
}


long int prompt_for_size() {
long int retval;

printf("Select PROM Size:\n\n");
printf("1       8 KBytes x 8 (2764/87C528)\n");
printf("2      16 KBytes x 8 (27128)\n");
printf("3      32 KBytes x 8 (27256)\n");
printf("4      64 KBytes x 8 (27512)\n");
printf("5     128 KBytes x 8 (28F010/27C1024)\n");

while (!kbhit()) {
};

switch (getch())
{
case '1':
retval = 8191;
break;
case '2':
retval = 16383;
break;
case '3':
retval = 32767;
break;
case '4':
retval = 65535;
break;
case '5':
retval = 131071;
break;
default:
retval = 0;
break;
}

return( retval );
}
```

References

1. Binstock, Andrew and Rex, John. 1995, *Practical Algorithms for Programmers* (Reading, MA: Addison-Wesley), pp. 536-541.

0.12.2 Weighted Checksums

A weighted checksum works in nearly the same way as a normal checksum. Where a normal checksum is based on the summation of the values in a data range, however, a weighted checksum takes both data value and position into consideration. To do this, each distinct position in the data range is assigned a weight. Each position's weight is multiplied by the value at the position in question to achieve the final terms in the sum. These terms are added and the weighted checksum value is determined based on this summation.

For instance, imagine creating a weighted checksum for telephone numbers in the format 123-456-7890. If we remove the dashes from the numbers we get something that looks like 1234567890. There are ten places in a number of this format. A typical way to assign weights to places is to give the nth place the weight of 7^n. Other values, such as powers of nine or powers of four are also used.

place	**value**
0	1
1	7
2	49
3	343
4	2401
5	16807
6	117649
7	823543
8	5764801
9	40353607

Using such a scheme, the phone number 1234567890 would result in the sum:

$$S = 1 * 1 + 2 * 7 + 3 * 49 + 4 * 343 + ... + 0 * 40353607$$

The weighted checksum would then be calculated in such a way as to yield a meaningful result when added to the weighted sum calculated in the manner described above. Perhaps the summation added to the checksum would have to yield a number evenly divisible by some large prime. Or perhaps the checksum would negate the summation and their combination would result in zero.

References

1. Binstock, Andrew and Rex, John. 1995, *Practical Algorithms for Programmers* (Reading, MA: Addison-Wesley), pp. 541-543.

0.12.3 Cyclic Redundancy Checks

While checksums are good at detecting errors in data they can be fooled by transposition or complimenting changes. Weighted checksums do a better job of detecting transposition but are limited by the size of the data which they are being used to verify. For numbers longer than about fifteen digits, the powers used to generate the positional values become too large to handle with C's basic types. While writing an extended number representation is one option to rectify this shortcoming, another method called a cyclical redundancy check (CRC) is another.

A CRC computes a result based on both the value and position of individual bits in a block of data. To do so it uses individual bit values as exponents in a special polynomial:

$$bit_{n-1} * x^{n-1} + bit_{n-1} * x^{n-2} + ... + bit_0 * x^0$$

For example, for the sixteen bit number 1001101100100001 the polynomial (leaving out zero valued terms) would be:

$$x^{15} + x^{12} + x^{11} + x^9 + x^8 + x^5 + x^0$$

This polynomial is calculated and then divided by another. The value of this divisor polynomial is based on the type of CRC implemented.

CRC-CCIT

The CRC-CCIT is used by the Xmodem-CRC telecommunication protocol and takes its name from the CCIT worldwide standards organization. This flavor of CRC is used on single bytes of data. The divisor polynomial for this CRC flavor is:

$$x^{16} + x^{12} + x^5 + 1$$

The CRC-CCIT value, for an 8-bit number, is the remainder of the division of the CRC polynomial by the above divisor polynomial. For instance, for the previously examined bit pattern 10011011 the CRC-CCIT fraction would look like this:

$$\text{CRC-CCIT} = \text{remainder of } \frac{x^7+x^4+x^3+x^1+1}{x^{16}+x^{12}+x^5+1}$$

This may look like a difficult expression to simplify, especially with the speed demanded by limited buffer size and high bandwidth communications. However, by carefully choosing the value of x so as to make division possible using only bitwise operations, the value can be computed very rapidly. This value is $0x1021$.

An even faster alternative to calculating the remainder of the above division is to create a lookup table of values. Since the denominator of the division is a constant and the numerator can have only 256 possible values (one for each possible bit combination), a table of 256 remainders can provide instant access to the CRC-CCIT value of a particular byte. Such a table can be generated easily by repeatedly calling the `get_crc_ccit` function below in a loop with values ranging from zero to 255 and a crc seed value of zero.

```
unsigned short table[256];

for (i = 0; i < 256; i++) {

  /* the implementation of get_crc_ccit is given below */
  table[i] = get_crc_ccit(i, 0);
}
```

Here's the table (all CRC values are hexadecimal):

(0)	0x0000	0x1021	0x2042	0x3063	0x4084	0x50a5	0x60c6	0x70e7
(8)	0x8108	0x9129	0xa14a	0xb16b	0xc18c	0xd1ad	0xe1ce	0xf1ef
(16)	0x1231	0x0210	0x3273	0x2252	0x52b5	0x4294	0x72f7	0x62d6
(24)	0x9339	0x8318	0xb37b	0xa35a	0xd3bd	0xc39c	0xf3ff	0xe3de
(32)	0x2462	0x3443	0x0420	0x1401	0x64e6	0x74c7	0x44a4	0x5485
(40)	0xa56a	0xb54b	0x8528	0x9509	0xe5ee	0xf5cf	0xc5ac	0xd58d
(48)	0x3653	0x2672	0x1611	0x0630	0x76d7	0x66f6	0x5695	0x46b4
(56)	0xb75b	0xa77a	0x9719	0x8738	0xf7df	0xe7fe	0xd79d	0xc7bc
(64)	0x48c4	0x58e5	0x6886	0x78a7	0x0840	0x1861	0x2802	0x3823
(72)	0xc9cc	0xd9ed	0xe98e	0xf9af	0x8948	0x9969	0xa90a	0xb92b
(80)	0x5af5	0x4ad4	0x7ab7	0x6a96	0x1a71	0x0a50	0x3a33	0x2a12
(88)	0xdbfd	0xcbdc	0xfbbf	0xeb9e	0x9b79	0x8b58	0xbb3b	0xab1a
(96)	0x6ca6	0x7c87	0x4ce4	0x5cc5	0x2c22	0x3c03	0x0c60	0x1c41
(104)	0xedae	0xfd8f	0xcdec	0xddcd	0xad2a	0xbd0b	0x8d68	0x9d49
(112)	0x7e97	0x6eb6	0x5ed5	0x4ef4	0x3e13	0x2e32	0x1e51	0x0e70
(120)	0xff9f	0xefbe	0xdfdd	0xcffc	0xbf1b	0xaf3a	0x9f59	0x8f78
(128)	0x9188	0x81a9	0xb1ca	0xa1eb	0xd10c	0xc12d	0xf14e	0xe16f
(136)	0x1080	0x00a1	0x30c2	0x20e3	0x5004	0x4025	0x7046	0x6067
(144)	0x83b9	0x9398	0xa3fb	0xb3da	0xc33d	0xd31c	0xe37f	0xf35e
(152)	0x02b1	0x1290	0x22f3	0x32d2	0x4235	0x5214	0x6277	0x7256
(160)	0xb5ea	0xa5cb	0x95a8	0x8589	0xf56e	0xe54f	0xd52c	0xc50d
(168)	0x34e2	0x24c3	0x14a0	0x0481	0x7466	0x6447	0x5424	0x4405
(176)	0xa7db	0xb7fa	0x8799	0x97b8	0xe75f	0xf77e	0xc71d	0xd73c
(184)	0x26d3	0x36f2	0x0691	0x16b0	0x6657	0x7676	0x4615	0x5634
(192)	0xd94c	0xc96d	0xf90e	0xe92f	0x99c8	0x89e9	0xb98a	0xa9ab
(200)	0x5844	0x4865	0x7806	0x6827	0x18c0	0x08e1	0x3882	0x28a3
(208)	0xcb7d	0xdb5c	0xeb3f	0xfb1e	0x8bf9	0x9bd8	0xabbb	0xbb9a
(216)	0x4a75	0x5a54	0x6a37	0x7a16	0x0af1	0x1ad0	0x2ab3	0x3a92
(224)	0xfd2e	0xed0f	0xdd6c	0xcd4d	0xbdaa	0xad8b	0x9de8	0x8dc9
(232)	0x7c26	0x6c07	0x5c64	0x4c45	0x3ca2	0x2c83	0x1ce0	0x0cc1
(240)	0xef1f	0xff3e	0xcf5d	0xdf7c	0xaf9b	0xbfba	0x8fd9	0x9ff8
(248)	0x6e17	0x7e36	0x4e55	0x5e74	0x2e93	0x3eb2	0x0ed1	0x1ef0

In order to arrive at a CRC for the entire range of data to be verified, a running CRC value is used. This is to say, the CRC of the first byte of data is used to somehow affect the CRC value of the second byte, and so on...

```
for (i = 0; i < data_size; i++) {
  crc = get_crc_ccit(crc, buffer[i]);
}
```

Here is the code to calculate the CRC of a byte, based on source from Rex and Binstock. The initial value of crc should be zero on the first call to this routine.

```
#include <stdio.h>
#include <stdlib.h>

unsigned short get_crc_ccit (unsigned short crc, unsigned short data) {
  static unsigned int i;

  /* move to most significant bit */
  data <<= 8;

  /* for each bit in the character... */
  for (i = 8; i > 0; i--) {

    /* calculate */
    if ((data ^ crc) & 0x8000) crc = (crc << 1) ^ 0x1021;
    else crc <<= 1;

    /* next bit please */
    data <<= 1;
  }
  return(crc)
}
```

CRC-16

CRC-16's numerator polynomial is based on sixteen bits of data unlike the CCIT's which is based on eight. The divisor polynomial for the CRC-16 is also different than the CCIT's, it is:

$$x^{16} + x^5 + x^2 + 1$$

The CRC-16, like the CRC-CCIT, can be implemented with a table lookup strategy. However where the CCIT needed 256 distinct entries in the table (one for each possible numerator – 2^8 total) the CRC-16 can work with a much smaller table. By examining the table value of each of the four nibbles (four bits) of data in the sixteen bit numerator polynomial the CRC-16 can arrive at a value for the upper part of the fraction. The denominator is, of course, a constant. So, the CRC-16 can operate quickly with a table size of only sixteen entries.

Because it can operate quickly with a small lookup table, the CRC-16 is often used in hardware devices such as disk controllers and the like.

```
/*
 *  this source code is based on Rex and Binstock which, in turn,
 *  acknowledges William James Hunt.
 */

#include <stdlib.h>
#include <stdio.h>

unsigned int crc_16_table[16] = {
  0x0000, 0xCC01, 0xD801, 0x1400, 0xF001, 0x3C00, 0x2800, 0xE401,
  0xA001, 0x6C00, 0x7800, 0xB401, 0x5000, 0x9C01, 0x8801, 0x4400 };

unsigned short int get_crc_16 (int start, char *p, int n) {
  unsigned short int crc = start;
  int r;

  /* while there is more data to process */
  while (n-- > 0) {

    /* compute checksum of lower four bits of *p */
    r = crc_16_table[crc & 0xF];
    crc = (crc >> 4) & 0x0FFF;
    crc = crc ^ r ^ crc_16_table[*p & 0xF];

    /* now compute checksum of upper four bits of *p */
    r = crc_16_table[crc & 0xF];
    crc = (crc >> 4) & 0x0FFF;
    crc = crc ^ r ^ crc_16_table[(*p >> 4) & 0xF];

    /* next... */
    p++;
  }

  return(crc);
}
```

CRC-32

One of the drawbacks shared by the CRC-16 and the CRC-CCIT is that they will both fail to detect errors involving the omission of leading null (zero) bits. The CRC-32 attempts to solve this problem by initializing the value of the CRC to 0xFFFFFFFF. The other two methods discussed operate with an initial CRC value of zero. This use of a non-zero initial CRC value is known as **preconditioning**. The CRC-32 also makes use of something called **postconditioning** which means altering the value of the final CRC before returning. CRC-32 flips the bits on the CRC. According to Rex and Binstock, this practice is not designed to increase the accuracy of error detection but is, rather, simply part of the definition of CRC-32.

CRC-32 operates with the same scheme as the other two methods previously addressed; it creates a numerator polynomial and a denominator polynomial. The latter is what you would expect:

$$\sum_{n=0}^{31} x^n$$

The divisor in CRC-32 is:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

References

1. Binstock, Andrew and Rex, John. 1995, *Practical Algorithms for Programmers* (Reading, MA: Addison-Wesley), pp. 541-543.