Algorithm Archive

(Work in progress)

Scott Gasch, scott@acm.vt.edu
Virginia Polytechnical Institute and State University
(Virginia Tech)
Computer Science Undergraduate

$Id : alg.tex, v1.61998/03/3004 : 41 : 38 scott Exp scott$

Contents

0.1 Prelude

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 mistakes. I have been working on this collection for a year now in my spare time so, from time to time, you will notice that it expands. 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 in LaTeX format and in Adobe Postscript format.

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! (scott@acm.vt.edu)

0.1.1 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.

0.2 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. This method 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 this reason a 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:

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

```
#define YES             1
#define NO              0
#define NUM_ELEMENTS    100

int array[NUM_ELEMENTS];
int found=NO, i=0;
int target = 5;

int main(void) {

  srand(1);

  for (i = 0; i<NUM_ELEMENTS; i++) {
    array[i] = rand();
  }

  /* now find it */
  while ((i < NUM_ELEMENTS) && (!found)) {
    if (array[i] == target) {
      printf("Found at element %d!\n", i);
      found = YES;
    }
    i++;
  }

  if (found == NO) 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.

Both take advantage 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 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 $O(log_2n)$ 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 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. It also proves very expensive to add or delete items from such a sorted array as such operations require a **ripple-shift**of array elements to open or close a element "hole" in the array. Secondly, the binary search assumes an array is the data structure to be searched. It is difficult 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 is also sometimes implemented as a recursive algorithm.

```
typedef keytype   int;

#define NOTFOUND -1
#define YES       1
#define NO        0

int bsearch(keytype target, keytype *searchspace) {

  int found = NO;
  int first = 1;

  /* assumed defined elsewhere */
  int last = num_elements(searchspace);

  int midpoint;

  while ((first <= last) && (!found)) {

    /* calculate the midpoint of the current [sub]range */
    midpoint = (first + last) / 2;

    /* compare the target with the value of the midpoint */
    if (searchspace[midpoint] == target) found = YES;
      else {

        /* if the midpoint is not the target
           adjust the range accordingly */
        if (target < searchspace[midpoint])
          last = midpoint - 1;
        else first = midpoint + 1;
      }

  }
```

```
  if (found) return(midpoint);
  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 a **binary search algorithm**.Binary search trees, often called "BSTs," retain the excellent search speed of a binary search on 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.

A BST is a special type of **binary tree**.This means that every **node**in the tree has 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." This can be accomplished by traversing to the right child of the node to be deleted and then traversing to the left as far as possible. Promote this newly selected node to the place in the tree occupied by the node to be deleted.

Source Code

Below is double pointer implementation of a BST in C:

```
#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.

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. 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

Below is an ANSI C implementation of a red-black tree:

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

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

/* red-black tree description */
typedef enum { Black, Red } NodeColor;

typedef struct Node_ {
    struct Node_ *Left;         /* left child */
    struct Node_ *Right;        /* right child */
    struct Node_ *Parent;       /* parent */
    NodeColor Color;            /* node color (black, red) */
    T Data;                     /* data stored in node */
} Node;

#define NIL \&Sentinel           /* all leafs are sentinels */
Node Sentinel = { NIL, NIL, 0, Black, 0};

Node *Root = NIL;               /* root of red-black tree */


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

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

```
    ***********************************************/

    /* setup new node */
    if ((X = malloc (sizeof(*X))) == 0) {

        printf ("insufficient memory (InsertNode)\n");
        exit(1);
    }
    X->Data = Data;
    X->Left = NIL;
    X->Right = NIL;
    X->Parent = 0;
    X->Color = Red;

    /* find where node belongs */
    Current = Root;
    Parent = 0;
    while (Current != NIL) {
        if (CompEQ(X->Data, Current->Data)) return (Current);
        Parent = Current;
        Current = CompLT(X->Data, Current->Data) ?
            Current->Left : Current->Right;
    }

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

    InsertFixup(X);
    return(X);
}
```

```
void InsertFixup(Node *X) {

   /*************************************
    *  maintain red-black tree balance  *
    *  after inserting node X           *
    *************************************/

    /* check red-black properties */
    while (X != Root && X->Parent->Color == Red) {
        /* we have a violation */
        if (X->Parent == X->Parent->Parent->Left) {
            Node *Y = X->Parent->Parent->Right;
            if (Y->Color == Red) {

                /* uncle is red */
                X->Parent->Color = Black;
                Y->Color = Black;
                X->Parent->Parent->Color = Red;
                X = X->Parent->Parent;
            } else {

                /* uncle is black */
                if (X == X->Parent->Right) {
                    /* make X a left child */
                    X = X->Parent;
                    RotateLeft(X);
                }

                /* recolor and rotate */

                X->Parent->Color = Black;
                X->Parent->Parent->Color = Red;
                RotateRight(X->Parent->Parent);
            }
        } else {

            /* mirror image of above code */
            Node *Y = X->Parent->Parent->Left;
```

```
            if (Y->Color == Red) {

                /* uncle is red */
                X->Parent->Color = Black;
                Y->Color = Black;
                X->Parent->Parent->Color = Red;
                X = X->Parent->Parent;
            } else {

                /* uncle is black */
                if (X == X->Parent->Left) {
                    X = X->Parent;
                    RotateRight(X);
                }
                X->Parent->Color = Black;
                X->Parent->Parent->Color = Red;
                RotateLeft(X->Parent->Parent);
            }
        }
    }
    Root->Color = Black;
}

void RotateLeft(Node *X) {

   /***************************
    *  rotate Node X to left *
    ***************************/

    Node *Y = X->Right;

    /* establish X->Right link */
    X->Right = Y->Left;
    Y->Left->Parent = X;

    /* establish Y->Parent link */
    Y->Parent = X->Parent;
    if (X->Parent) {
```

```
        if (X == X->Parent->Left)
            X->Parent->Left = Y;
        else
            X->Parent->Right = Y;
    } else {
        Root = Y;
    }

    /* link X and Y */
    Y->Left = X;
    X->Parent = Y;
}

void RotateRight(Node *X) {

   /****************************
    *  rotate Node X to right  *
    ****************************/

    Node *Y = X->Left;

    /* establish X->Left link */
    X->Left = Y->Right;
    Y->Right->Parent = X;

    /* establish Y->Parent link */
    Y->Parent = X->Parent;
    if (X->Parent) {
        if (X == X->Parent->Right)
            X->Parent->Right = Y;
        else
            X->Parent->Left = Y;
    } else {
        Root = Y;
    }


    /* link X and Y */
```

```
    Y->Right = X;
    X->Parent = Y;
}

void DeleteNode(Node *Z) {
    Node *X, *Y;
    NodeColor YColor;

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

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


    /* find tree successor */
    if (Z->Left == NIL || Z->Right == NIL)

        Y = Z;
    else {
        Y = Z->Right;
        while (Y->Left != NIL) Y = Y->Left;
    }

    /* X is Y's only child */
    if (Y->Left != NIL)
        X = Y->Left;
    else
        X = Y->Right;

    /* remove Y from the parent chain */
    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. */
    YColor = Y->Color;
    if (Y != Z) {
        Y->Left = Z->Left;
        Y->Left->Parent = Y;
        Y->Right = Z->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;

        Y->Color = Z->Color;
        free (Z);
    } else {
        free (Y);

    }

    if (YColor == Black)
        DeleteFixup (X);
}

void DeleteFixup(Node *X) {

   /*************************************
    *  maintain red-black tree balance  *
    *  after deleting node X            *
    *************************************/
```

```
while (X != Root && X->Color == Black) {
    if (X == X->Parent->Left) {
        Node *W = X->Parent->Right;
        if (W->Color == Red) {
            W->Color = Black;
            X->Parent->Color = Red;
            RotateLeft (X->Parent);
            W = X->Parent->Right;
        }
        if (W->Left->Color ==
            Black && W->Right->Color == Black) {
            W->Color = Red;
            X = X->Parent;
        } else {
            if (W->Right->Color == Black) {
                W->Left->Color = Black;
                W->Color = Red;
                RotateRight (W);
                W = X->Parent->Right;
            }
            W->Color = X->Parent->Color;
            X->Parent->Color = Black;
            W->Right->Color = Black;
            RotateLeft (X->Parent);
            X = Root;
        }
    } else {
        Node *W = X->Parent->Left;
        if (W->Color == Red) {
            W->Color = Black;
            X->Parent->Color = Red;
            RotateRight (X->Parent);
            W = X->Parent->Left;
        }
        if (W->Right->Color ==
            Black && W->Left->Color == Black) {
            W->Color = Red;
```

```
                X = X->Parent;
            } else {
                if (W->Left->Color == Black) {
                    W->Right->Color = Black;
                    W->Color = Red;
                    RotateLeft (W);
                    W = X->Parent->Left;
                }
                W->Color = X->Parent->Color;
                X->Parent->Color = Black;
                W->Left->Color = Black;
                RotateRight (X->Parent);
                X = Root;
            }
        }
    }
    X->Color = Black;
}

Node *FindNode(T Data) {

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

    Node *Current = Root;
    while(Current != NIL)
        if(CompEQ(Data, Current->Data))
            return (Current);
        else
            Current = CompLT (Data, Current->Data) ?
                Current->Left : Current->Right;
    return(0);
}
```

References

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

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. The internal nodes of the tree are stored in computer memory. In a B-Tree the leaf nodes are the only ones that store data. All other nodes are called **index nodes** and simply store keys which allow us to traverse the structure and arrive at the desired leaf (data) node. If you have B-tree source code you are willing to donate to the archive, please mail me!

0.2.6 B+-Trees

```
/*-- bio.c --*/
#include <stdio.h>
#include <stdlib.h>
#include "b+tree.h"
#include "global.h"
#include "bio.h"
#include "buffers.h"

leaf_node *ln;
internal_node *in;
leaf_node *test;
FILE *fp;

int purge_file (char *filename) {
  fclose(fp);
  return(remove(filename));
}
```

```
int open_file (char *filename) {
  initialize_buffers(MANY_BUFFERS);
  return((fp = fopen(filename, "w+")) != NULL);
}

int write_node (void *node, int type) {
  if (!fp) return(NO);

  switch (type) {
    case LEAF_NODE:
      ln = (leaf_node *) node;
      ln->type = LEAF_NODE;
      fseek(fp, (long)
           (ln->node_num * MAX_NODE_SIZE),
           SEEK_SET);
      fwrite(ln, sizeof(leaf_node), 1, fp);
      checkin_node (ln->node_num, node, LEAF_NODE);
      break;

    case INTERNAL_NODE:
      in = (internal_node *) node;
      in->type = INTERNAL_NODE;
      fseek(fp, (long)
           (in->node_num * MAX_NODE_SIZE),
           SEEK_SET);
      fwrite(in, sizeof(internal_node), 1, fp);
      checkin_node (in->node_num, node, INTERNAL_NODE);
      break;

    default:
      exit(1);
      break;
  }

/*  print_buffers(); */
  return(1);
}
```

```
void *read_node (int which, int *type) {
  long bytepos;
  int rval;
  void *vp;
  int kind;

  if (checkout_node (&vp, &kind, which)) {
    *type = kind;
    return(vp);

  } else {

    bytepos = (which * MAX_NODE_SIZE);
    ln = (leaf_node *) malloc(sizeof(leaf_node));
    in = (internal_node *) malloc(sizeof(internal_node));
    fseek (fp, (long) bytepos, SEEK_SET);
    rval = fread(in, sizeof(internal_node), 1, fp);

    if (in->type != INTERNAL_NODE) {

      memcpy((void *)ln, (void *)in, sizeof(leaf_node));
      *type = LEAF_NODE;
      free(in);
      return(ln);

    } else {

      *type = INTERNAL_NODE;
      free(ln);
      return(in);
    }
  }
}
```

```
/*-- b+tree.c --*/
#include <stdlib.h>
```

```
#include <stdio.h>
#include "b+tree.h"
#include "global.h"
#include "errors.h"
#include "bio.h"

/*
 *
 * module: b+tree.c
 *
 * purpose: ADT containing b+tree manipulation functions
 *
 **/


/* number of nodes currently on the tree */
int num_nodes;

/* number of leaves currently on the tree */
int num_leaf;

/* number of internal nodes on the tree */
int num_internal;

/* a pointer to the current root */
nodetype root;


/* function prototypes */
static int node_full(int node);
static int node_not_full(int node);
static int split_leaf(int node);
static int split_internal(int node);
static int simple_add_key_to_leaf(int key,
                    char data[SIZE_OF_DATA], int node);
static void arrange(keytype keys[KVPAIRS_AT_LEAF],
                    datatype data[KVPAIRS_AT_LEAF][SIZE_OF_DATA],
                    int number);
```

```
static int dupe_key(keytype key);
static int find_pos (keytype key_to_insert, nodetype node);
static void show_leaf (leaf_node *it);
static int is_leaf (int pos);
static int is_internal (int pos);
static int add_key_to_internal (int key, int node, int llink, int gelink);
static void show_node (int node);
static void show_internal (internal_node *it);
static void update_parent (int node, int parent);
static char *lookup_key(keytype key);
static int delete_would_make_less_than_half_full(int node);
static void perform_simple_delete(int node, int key);
static int search_leaf(leaf_node *ln, int key);
static int search_internal(internal_node *in, int key);
static void delete_from_leaf(int node, keytype key);
void bplus_delete(int key);
static int can_merge(int node);
static void do_merge(int node, keytype key);
static int shuffle_and_delete(int node, int key);

/* initialize the system function */
void init_bplus (void) {

  leaf_node *ln;                             /* the first node is a leaf node */
  int i;                                                     /* loop control */

  /* get the disk file ready */
  purge_file(BPLUS_FILENAME);
  open_file(BPLUS_FILENAME);

  /* initialize b+-tree variables */
  root = 1;
  num_nodes = 1;                                   /* one node in the system */
  num_leaf = 1;                                          /* it's a leaf node */
  num_internal = 0;                               /* no internal nodes (yet) */

  /* create the first (leaf) node in memory */
  ln = (leaf_node *) malloc(sizeof(leaf_node));
```

```
  ln->type = LEAF_NODE;
  ln->next_leaf = 0;
  ln->prev_leaf = 0;
  ln->node_num = root;
  ln->kvpairs_here = 0;
  for (i = 0; i < KVPAIRS_AT_LEAF; i++) {
    ln->keys[i] = 0;
    strcpy(ln->data[i], "empty!");
  }

  /* write this node to disk and dispose of the memory record */
  write_node(ln, LEAF_NODE);
  free(ln);
}
```

```
/* close the system */
void bplus_close(void) {
}
```

```
/* delete a key value from the tree */
void bplus_delete(int key) {
  int node = find_pos(key, root);

  delete_from_leaf(node, key);
}
```

```
/* delete a key value from a leaf */
static void delete_from_leaf(int node, keytype key) {
```

```
  if (delete_would_make_less_than_half_full(node)) {
    /* if (can_merge(node)) do_merge(node, key);
      else shuffle_and_delete(node, key);*/
    perform_simple_delete(node, key);
  } else {
    perform_simple_delete(node, key);
  }
}



/* boolean functoin, returns true or false on whether a merge is possible */
static int can_merge(int node) {
  int kind;                                        /* what kind of node is it? */
  int rval;                                                   /* return value */
  int x;                     /* x and y are used to make calculations shorter */
  int y;
  leaf_node *lneigh;                              /* my left and right neighbors */
  leaf_node *rneigh;
  leaf_node *me = read_node(node, &kind);                          /* and me! */


  /* what's our situation -- is there something to both sides of us? */
  if ((me->prev_leaf != 0) && (me->next_leaf != 0)) {

    /* there are two neighbors, left and right! */


    /* read them */
    lneigh = read_node(me->prev_leaf, &kind);
    rneigh = read_node(me->next_leaf, &kind);


    /* now see if we can merge with either one of them */
    x = (lneigh->kvpairs_here + me->kvpairs_here - 1);
    y = (rneigh->kvpairs_here + me->kvpairs_here - 1);
    rval = (((x > (KVPAIRS_AT_LEAF / 2)) &&
              (x <= KVPAIRS_AT_LEAF)) ||
```

```
          ((y > (KVPAIRS_AT_LEAF / 2)) &&
                (y <= KVPAIRS_AT_LEAF)));

  /* clean up... */
  free (lneigh);
  free (rneigh);
  free (me);
  return(rval);



/* okay, maybe there is something to the left but not right? */

} else if (me->prev_leaf != 0) {

  /* we've got nothin to the right, somethin to the left */


  /* read it */
  lneigh = read_node(me->prev_leaf, &kind);


  /* determine if we can merge */
  x = (lneigh->kvpairs_here + me->kvpairs_here - 1);
  rval = ((y > (KVPAIRS_AT_LEAF / 2)) &&
               (y <= KVPAIRS_AT_LEAF));

  /* and clean up */
  free(lneigh);
  free(me);
  return(rval);


/* vice versa... maybe something right but not left? */

} else if (me->next_leaf != 0) {

  /* nothin to the left, somethin to the right */
```

```
    /* read it */
    rneigh = read_node(me->next_leaf, &kind);

    /* see if we can merge */
    x = (rneigh->kvpairs_here + me->kvpairs_here - 1);
    rval = ((y > (KVPAIRS_AT_LEAF / 2)) &&
            (y <= KVPAIRS_AT_LEAF));

    /* clean up */
    free(rneigh);
    free(me);
    return(rval);


  /* we must be the only node */
  } else {

    /* we have NO neighbors (we must smell bad) */

    free(me);

    /* nothing to merge with! */
    return(NO);
  }
}




/* actually merge two nodes */
static void do_merge(int node, keytype key) {
  int kind, rval, x, y;
  leaf_node *lneigh, *rneigh, *me = read_node(node, &kind);

  if ((me->prev_leaf != 0) && (me->next_leaf != 0)) {
```

```
    /* there are two neighbors, left and right! */

    lneigh = read_node(me->prev_leaf, &kind);
    rneigh = read_node(me->next_leaf, &kind);

    x = (lneigh->kvpairs_here + me->kvpairs_here - 1);
    y = (rneigh->kvpairs_here + me->kvpairs_here - 1);
    if (((x > (KVPAIRS_AT_LEAF / 2)) &&
              (x <= KVPAIRS_AT_LEAF))) {
/*        merge_em(me->node_num, lneigh->node_num, key);  */
    } else if (((y > (KVPAIRS_AT_LEAF / 2)) &&
              (y <= KVPAIRS_AT_LEAF))) {
/*        merge_em(me->node_num, rneigh->node_num, key);  */
    }
    free (rneigh);
    free (lneigh);
    free (me);
    return;


  } else if (me->prev_leaf != 0) {

    /* we've got nothin to the right, somethin to the left */

    lneigh = read_node(me->prev_leaf, &kind);

/*       merge_me(me->node_num, lneigh->node_num, key); */
    free(lneigh);
    free(rneigh);
    free(me);
    return;

  } else if (me->next_leaf != 0) {

    /* nothin to the left, somethin to the right */

    rneigh = read_node(me->next_leaf, &kind);
```

```
/*    merge_em(me->node_num, rneigh->node_num); */

    free(me);
    free(rneigh);
    free(lneigh);
    return;

  } else {

    /* we have NO neighbors?!?!  Who the hell are we supposed to merge with? */
    return;
  }
}



static int shuffle_and_delete(int node, int key) {
  int kind, rval;
  leaf_node *lneigh, *rneigh, *me = read_node(node, &kind);

  if ((me->prev_leaf != 0) && (me->next_leaf != 0)) {

    /* there are two neighbors, left and right! */

    lneigh = read_node(me->prev_leaf, &kind);
    rneigh = read_node(me->next_leaf, &kind);

    if (lneigh->kvpairs_here > (KVPAIRS_AT_LEAF / 2)) {


    } else if (rneigh->kvpairs_here >
              (KVPAIRS_AT_LEAF / 2)) {


    } else {


    }
```

```
    free (lneigh);
    free (rneigh);
    return(rval);



  } else if (me->prev_leaf != 0) {

    /* we've got nothin to the right, somethin to the left */

    lneigh = read_node(me->prev_leaf, &kind);

    rval = lneigh->kvpairs_here > (KVPAIRS_AT_LEAF / 2);
    free(lneigh);
    return(rval);

  } else if (me->next_leaf != 0) {

    /* nothin to the left, somethin to the right */

    rneigh = read_node(me->next_leaf, &kind);

    rval = rneigh->kvpairs_here > (KVPAIRS_AT_LEAF / 2);
    return(rval);

  } else {

    /* we have NO neighbors (we must smell bad) */

    return(NO);
  }
}
```

```
/* search for a particular value on a leaf */
static int search_leaf(leaf_node *ln, int key) {
  int found = NO;  /* did we find it yet? */
  int first = 0;   /* where to start the search */
  int last = ln->kvpairs_here - 1;  /* where to end the search */
  int mid;  /* calculated midpoint */


  /* binary search it... */
  while ((first <= last) && (!found)) {

    /* look midway */
    mid = (first + last) / 2;

    /* is it there? */
    if (key == ln->keys[mid])
      found = YES;

    /* if not, adjust the upper or lower bound accordingly */
    else {
      if (key < ln->keys[mid])
        last = mid - 1;
      else
        first = mid + 1;
    }
  }

  if (found)
    return(mid);
  else
    return(0);
}


/* same as above, bsearch for a value in an internal */
static int search_internal(internal_node *in, int key) {
  int found = NO;
```

```
  int first = 0;
  int last = in->klpairs_here - 1;
  int mid;

  while ((first <= last) && (!found)) {
    mid = (first + last) / 2;
    if (key == in->keys[mid])
      found = YES;
    else {
      if (key < in->keys[mid])
        last = mid - 1;
      else
        first = mid + 1;
    }
  }

  if (found)
    return(in->links[mid+1]);
  else
    return(in->links[mid]);
}


/* search the index for a value */
void bplus_query(keytype key) {

  /* use internal helper to look it up */
  char *data = lookup_key(key);

  /* if it found anything, show the data */
  if (strcmp(data, "")) printf("QUERY RESULT: %s \n", data);

  /* cleanup */
  free(data);
}


/* simple delete case... where we only have to delete one thing and not
```

```
   worry about the damn merges or shuffles */
static void perform_simple_delete(int node, int key) {
  int kind, i;
  leaf_node *ln = (leaf_node *) read_node(node, &kind);
  internal_node *in;
  int pos;


  /* handle both leaf nodes and internal */
  if (kind == LEAF_NODE) {

    /* where is it? */
    if ((pos = search_leaf(ln, key)) != 0) {

      /* nuke it */
      ln->keys[ln->kvpairs_here] = 0;
      strcpy(ln->data[ln->kvpairs_here], "empty!");

      /* move everything else over */
      for (i = pos; i < ln->kvpairs_here; i++) {
        ln->keys[i] = ln->keys[i+1];
        strcpy(ln->data[i], ln->data[i+1]);
      }

      /* handle accounting and update disk */
      ln->kvpairs_here--;
      write_node(ln, LEAF_NODE);

    } else {
      printf("Not found!\n");
      free(ln);
      exit(9);
    }

  /* not done */
  } else {
    in = (internal_node *) ln;
```

```
  }
  free(ln);
}


/* return true if a delete would make a node less than half full, false
   otherwise */
static int delete_would_make_less_than_half_full(int node) {
  int kind, rval;
  leaf_node *ln = (leaf_node *) read_node(node, &kind);
  internal_node *in;

  if (kind == LEAF_NODE) {

    /* here's the beef -- just this formula */
    rval = ((ln->kvpairs_here - 1) <= (KVPAIRS_AT_LEAF / 2));
  } else {
    in = (internal_node *) ln;

    /* same */
    rval = ((in->klpairs_here - 1) <= (KLPAIRS_AT_INTERNAL / 2));
  }
  free(ln);
  return(rval);
}




/* is the node full? */
static int node_full(int node) {
  int kind, rval;
  leaf_node *ln = (leaf_node *) read_node(node, &kind);
  internal_node *in;

  /* depends on what kind of node it is! */
  if (kind == LEAF_NODE) {
    rval = (ln->kvpairs_here == KVPAIRS_AT_LEAF);
```

```
  } else {
    in = (internal_node *) ln;
    rval = (in->klpairs_here >= KLPAIRS_AT_INTERNAL);
  }
  free(ln);
  return(rval);
}


/* the exact opposite of the above one */
static int node_not_full(int node) {
  return (!node_full(node));
}




/* function to split a leaf node in half */
static int split_leaf(int node) {
  int kind;
  int move_from;
  int move_to;
  int rval;
  internal_node *in;
  leaf_node *new_node = (leaf_node *) malloc(sizeof(leaf_node));
  leaf_node *old_node = read_node(node, &kind);


  /* it can only be slpit when it is full */
  if (node_not_full(node)) {
    free(new_node);
    free(old_node);
    printf("Attempt to split a non-full node, %d!\n", node);
    exit(1);
  }

  /* well we are gunna spliut, so there will be one more node on the
     tree and one more leaf on the tree */
```

```
num_nodes++;
num_leaf++;

/* let's build the new node in memory */
new_node->node_num = num_nodes;
new_node->type = LEAF_NODE;
new_node->next_leaf = old_node->next_leaf;
new_node->prev_leaf = old_node->node_num;
old_node->next_leaf = new_node->node_num;
new_node->parent = old_node->parent;
new_node->kvpairs_here = 0;

/* move the latter half of old_node's key/value pairs over to new */
for (move_from = (KVPAIRS_AT_LEAF / 2) + 1;
     move_from < (KVPAIRS_AT_LEAF);
     move_from++)                                                {

  /* calculate where in the new node to move this kvpair */
  move_to = (move_from - (KVPAIRS_AT_LEAF / 2 + 1));

  /* move the key and data */
  new_node->keys[move_to] = old_node->keys[move_from];
  old_node->keys[move_from] = 0;
  strcpy(new_node->data[move_to], old_node->data[move_from]);
  strcpy(old_node->data[move_from], "empty!");
  old_node->kvpairs_here--;
  new_node->kvpairs_here++;
}


if (old_node->parent == 0) {
  /* we just split the root we need a new level in the tree */
  #ifdef DEBUG
    printf("DB: B+-tree just grew by one level.\n");
  #endif

  in = (internal_node *) malloc(sizeof(internal_node));
```

```
    /* update system */
    num_nodes++;
    num_internal++;
    root = num_nodes;

    /* fill our new root node */
    in->node_num = root;
    in->type = INTERNAL_NODE;
    in->parent = 0;
    in->klpairs_here = 1;
    in->keys[0] = new_node->keys[0];
    in->links[0] = old_node->node_num;
    in->links[1] = new_node->node_num;

    /* update these two nodes */
    new_node->parent = root;
    old_node->parent = root;

    /* update disk and clean up */
    write_node(in, INTERNAL_NODE);
    free(in);

  } else {

    /* did not split the root or add a level but we do need to update our
       parent with a pointer to the newly made node */
    add_key_to_internal(new_node->keys[0], new_node->parent, old_node->node_num, new_node->node_num);
  }

  rval = new_node->node_num;


  /* update disk */
  write_node(new_node, LEAF_NODE);
  write_node(old_node, LEAF_NODE);

  /* clean up */
  free(new_node);
```

```
  free(old_node);

  return(rval);
}


/* routine to split an internal node */
static int split_internal(int node) {
  int kind;
  int move_from;
  int move_to;
  int rval;
  keytype promote_me;
  internal_node *in;
  internal_node *new_node = (internal_node *) malloc(sizeof(internal_node));
  internal_node *old_node = read_node(node, &kind);

  if (node_not_full(node)) {
    free(new_node);
    free(old_node);
    printf("Trying to split a non-full internal node %d\n", node);
    exit(1);
  }

  /* update system */
  num_nodes++;
  num_internal++;

  /* fill in the new node in memory */
  new_node->node_num = num_nodes;
  new_node->type = INTERNAL_NODE;
  new_node->parent = old_node->parent;
  new_node->klpairs_here = 0;

  for (move_from = (KLPAIRS_AT_INTERNAL / 2) + 1;
       move_from < KLPAIRS_AT_INTERNAL;
       move_from++)                                                    {
```

```
    /* what index position to move to in the new node? */
    move_to = (move_from - ((KLPAIRS_AT_INTERNAL / 2) + 1));

    /* do the move */
    new_node->keys[move_to] = old_node->keys[move_from];
    old_node->keys[move_from] = 0;
    new_node->links[move_to] = old_node->links[move_from];
    update_parent(new_node->links[move_to], new_node->node_num);
    old_node->links[move_from] = 0;

    old_node->klpairs_here--;
    new_node->klpairs_here++;
  }

  /* move that last link */
  new_node->links[(KLPAIRS_AT_INTERNAL / 2)] = old_node->links[KLPAIRS_AT_INTERNAL];
  old_node->links[KLPAIRS_AT_INTERNAL] = 0;

  promote_me = old_node->keys[KLPAIRS_AT_INTERNAL / 2];
  old_node->keys[KLPAIRS_AT_INTERNAL / 2] = 0;
  old_node->klpairs_here--;

  if (old_node->parent == 0) {

    /* we just split the root and need a new one */
    in = (internal_node *) malloc(sizeof(internal_node));

    #ifdef DEBUG
       printf("DB: B+-tree just grew by one level.\n");
    #endif

    /* update system */
    num_nodes++;
    num_internal++;
    root = num_nodes;

    /* fill our new root node */
    in->node_num = num_nodes;
```

```
    in->type = INTERNAL_NODE;
    in->parent = 0;
    in->klpairs_here = 1;
    in->keys[0] = promote_me;
    in->links[0] = old_node->node_num;
    in->links[1] = new_node->node_num;

    /* update these two nodes */
    new_node->parent = root;
    old_node->parent = root;

    /* update disk and clean up */
    write_node(in, INTERNAL_NODE);
    free(in);

  } else {

    add_key_to_internal(promote_me, new_node->parent, old_node->node_num, new_node->node_num);
  }

  write_node(new_node, INTERNAL_NODE);
  write_node(old_node, INTERNAL_NODE);
  rval = new_node->node_num;

  free(new_node);
  free(old_node);

  return(rval);
}

static void update_parent (int node, int parent) {
  int kind;
  leaf_node *ln = read_node(node, &kind);

  ln->parent = parent;

  write_node(ln, kind);
  free(ln);
```

```
}


static int simple_add_key_to_leaf(int key, char data[SIZE_OF_DATA], int node) {
  int kind;
  leaf_node *ln = (leaf_node *) read_node(node, &kind);

  if (kind != LEAF_NODE) {
    printf("Called simple_add_key_to_leaf with non-leaf node %d\n", node);
    exit(1);
  } else if (node_full(node)) {
    return(NO);
  }

  ln->kvpairs_here++;
  ln->keys[ln->kvpairs_here - 1] = key;
  strcpy(ln->data[ln->kvpairs_here - 1], data);

  /* this is inefficient but I am leaving it for now */
  arrange(ln->keys, ln->data, ln->kvpairs_here);

  write_node(ln, LEAF_NODE);
  free(ln);

  return(YES);
}


int bplus_insert (keytype key, datatype *data) {
  int node = find_pos(key, root);
  int new_node;

/*  #ifdef DEBUG
    printf("I am trying to add key %u to the tree at node %d...\n", key, node);
  #endif */
```

```
  if (dupe_key(key)) {
    printf("  ...%u is already in the index, insert failed\n", key);
    return(NO);
  }

  /* if the leaf node where it belongs is not full, just add it */
  if (!node_full(node)) {
    simple_add_key_to_leaf(key, data, node);
    return(YES);

  } else {
    new_node = split_leaf(node);
    node = find_pos(key, root);
    simple_add_key_to_leaf(key, data, node);
  }

  return(YES);
}


static void arrange(keytype keys[KVPAIRS_AT_LEAF],
        datatype data[KVPAIRS_AT_LEAF][SIZE_OF_DATA],
        int number) {

  int i, j;
  int lowindex;
  keytype tempkey;
  datatype tempdata[SIZE_OF_DATA];

  for (i = 0; i < number; i++) {
    lowindex = i;

    for (j = number - 1; j > i; j--)
      if (keys[j] < keys[lowindex]) lowindex = j;

    tempkey = keys[i];
    keys[i] = keys[lowindex];
```

```
    keys[lowindex] = tempkey;

    strcpy(tempdata, data[i]);
    strcpy(data[i], data[lowindex]);
    strcpy(data[lowindex], tempdata);
  }
}

static int dupe_key(keytype key) {
  int kind;
  int node = find_pos(key, root);
  leaf_node *ln = read_node(node, &kind);
  int i;
  int found = 0;

  i = 0;
  while (i < ln->kvpairs_here) {
    found = (ln->keys[i] == key);
    if (found) {
      free(ln);
      return(YES);
    }
    i++;
  }
  free(ln);
  return(NO);
}



static char *lookup_key(keytype key) {
  int kind;
  char *data = NULL;
  int node = find_pos(key, root);
  leaf_node *ln = read_node(node, &kind);
  int i;
  int found = 0;
```

```
  data = (char *) malloc(SIZE_OF_DATA+1);
  strcpy(data, "");
  i = 0;
  while ((i < KVPAIRS_AT_LEAF) && (!found)) {
    found = ((ln->keys[i]) == key);
    if (found) strcpy(data, ln->data[i]);
    i++;
  }
  free(ln);
  return(data);
}


static int find_pos (keytype key_to_insert, nodetype node) {
  nodetype i = node;
  internal_node *in;
  int kind;
  int j;

  /* if it's a leaf node, insert here! */
  if (is_leaf(i))
    return(i);

  /* otherwise it's an internal node and we need to traverse... */
  else {
    in = (internal_node *) read_node(i, &kind);

    j = 0;
    while (j < in->klpairs_here) {
      if (key_to_insert < in->keys[j]) {
        i = in->links[j];
        free(in);
        return(find_pos(key_to_insert, i));
      }
      j++;
    }
    i = in->links[in->klpairs_here];
    free(in);
```

```
    return(find_pos(key_to_insert, i));
  }

  return(999);
}



static void show_leaf (leaf_node *it) {
  int i;

  printf("[LEAF] NODE #%d\n", it->node_num);
  printf("\n[keys]\t\t[data]\n--------------------------------------------------------------------------\n");
  i = 0;
  while (i < it->kvpairs_here) {
    printf("%u\t\t%s\n", it->keys[i], it->data[i]);
    i++;
  }
  printf("\n[Prev: %d]\t[Next: %d]\t[Keys: %d]\t[Parent: %d]\n", it->prev_leaf, it->next_leaf, it->kvpairs_here,
         it->parent);
}

static void show_node (int node) {
  int kind;
  leaf_node *ln = read_node(node, &kind);
  internal_node *in;

  if (kind == LEAF_NODE) {
    show_leaf(ln);
  } else {
    in = (internal_node *) ln;
    show_internal(in);
  }
  free(ln);
}


static void show_internal (internal_node *it) {
```

```
  int i;

  printf("[INTERNAL] NODE #%d\n\n", it->node_num);
  i = 0;
  while (i < it->klpairs_here) {
    printf("(%u) %u ", it->links[i], it->keys[i]);
    i++;
  }
  printf("(%u)\n\n", it->links[it->klpairs_here]);

  printf("\nKeys: %d\tParent: %d\n", it->klpairs_here, it->parent);
}


static int is_leaf (int pos) {
  void *vp;
  int kind;

  vp = read_node(pos, &kind);
  free(vp);
  return(kind == LEAF_NODE);
}

static int is_internal (int pos) {
  return (!is_leaf(pos));
}



static int add_key_to_internal (int key, int node, int llink, int gelink) {
  int kind, i, j;
  int new_node;
  internal_node *nn;
  internal_node *in = (internal_node *) read_node(node, &kind);

  if (kind != INTERNAL_NODE) {
    printf("Called add_key to internal with non-internal node %d\n", node);
    exit(1);
```

```
}

if (node_not_full(node)) {
  if (key < in->keys[0]) {
    for (j = in->klpairs_here; j > 0; j--)
      in->keys[j] = in->keys[j-1];
    for (j = in->klpairs_here + 1; j > 0; j--)
      in->links[j] = in->links[j-1];

    in->keys[0] = key;
    in->links[0] = llink;
    in->links[1] = gelink;
    in->klpairs_here++;

    write_node(in, INTERNAL_NODE);
    free(in);
    return(YES);

  } else if (key > in->keys[in->klpairs_here - 1]) {

    in->keys[in->klpairs_here] = key;
    in->links[in->klpairs_here] = llink;
    in->klpairs_here++;
    in->links[in->klpairs_here] = gelink;

    write_node(in, INTERNAL_NODE);
    free(in);
    return(YES);

  } else {

    i = 0;
    while (i <= in->klpairs_here - 2) {
      if ((in->keys[i] < key) && (in->keys[i+1] > key)) {

        /* insert between i and i+1 */
        for (j = in->klpairs_here; j > (i + 1); j--) {
          in->keys[j] = in->keys[j-1];
```

```
          in->links[j+1] = in->links[j];
        }

        in->keys[i+1] = key;
        in->links[i+1] = llink;
        in->links[i+2] = gelink;
        in->klpairs_here++;
        write_node(in, INTERNAL_NODE);
        free(in);
        return(YES);

      }

      i++;
    }

  }

} else {
  free(in);
  new_node = split_internal(node);
  nn = read_node(new_node, &kind);

  if (key > nn->keys[0]) {
    add_key_to_internal(key, new_node, llink, gelink);
  } else {
    add_key_to_internal(key, node, llink, gelink);
  }

  free(nn);
}

return(YES);
}



void query_range(unsigned a, unsigned b) {
```

```
  int kind;
  unsigned x = a;
  char *data = NULL;
  int node = find_pos(a, root);
  leaf_node *ln = read_node(node, &kind);
  leaf_node *next = ln;
  void *junk;
  int i, n;

  data = (char *) malloc(SIZE_OF_DATA+1);
  strcpy(data, "");

  while (x <= b) {
    if (next == ln) next = read_node(ln->next_leaf, &kind);
    while (next->keys[0] < x) {
      junk = ln;
      ln = next;
      free(junk);
      if (ln->next_leaf == 0) return;
      next = read_node(ln->next_leaf, &kind);
    }
    i = search_leaf(ln, x);
    if (i != 0) {
      printf("RESULT: %u == %s\n", x, ln->data[x]);
    }
    x++;
    if (ln->next_leaf == 0) return;
  }
}
```

0.2.7 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

0.2.8 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**.

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);
}
```

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

Below is an implementation of a linked-list style open hashing, division method hash table in ANSI C:

```
#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;
}
```

References

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

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

0.2.9 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.

> 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 CACM. A Postscript format version is available for anonymous ftp from mimsy.cs.umd.edu. The author, William Pugh, can be contacted at pugh@cs.umd.edu.

Source Code

```
/*

Example of Skip List source code for C:

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);

  };
```

References

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

0.2.10 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 it's 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, 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];
    }
}
```

0.2.11 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 working 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 -- tree.h
// Lynn Jones, Virginia Tech, lwjones@vt.edu
```

```
#ifndef _TREE
#define _TREE

#include "octreeNode.h"
#include <string.h>

class Tree_8 {
public:
  OctreeNode *root;

  Tree_8(OctreeNode *builtTree) {
    root = builtTree;
  }

  ~Tree_8() {
    delete root;
  }

  void DepthTraversal(OctreeNode *recurRoot = NULL, char *OctNodeFunction = "Print");
};

void Tree_8::DepthTraversal(OctreeNode *recurRoot, char *OctNodeFunction) {
  //entry case
  if (recurRoot == NULL)
    return;
  //else
  if (!(recurRoot->children[0])) {              //if null; at leaf
    if (!(strcmp(OctNodeFunction, "Print")))
      recurRoot->Print();
    return;
  }
  //else
  if (!(recurRoot->children[0]->children[0]))   //if null; at parent of leaf
    if (!(strcmp(OctNodeFunction, "PruneTree"))) {
      recurRoot->PruneTree();
      return;
    }
```

```
  //else
  for (int i=0; i<8; i++)
    DepthTraversal(recurRoot->children[i], OctNodeFunction);
}
#endif


// 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
#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 very 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 $O(nlog_2 n)$ solution.

Improvement Strategies

One way to speed up the Quicksort's slow worst-case performance is to "randomize" the input data before you sort. Clearly, this, too, is more overhead but makes faster-than-worst-case performance more likely.

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 $i = log_2 n$:

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

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, last);
    merge (array, left, (left+right)/2, (left+right)/2+1, last);
  }
}


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* (http://hercule.csci.unt.edu/ian)

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:

```
          1
         / \
        2   3
       /\   /\
      4  5 6  7
     / \
    8   9
```

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.

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);
  }
}

/* ------------------------------------------------------------------------ */
/*   hwdata - set the data set to that given on problem 2 of homework 4
 */

void hwdata(void) {
  num_comps = num_swaps = 0;
```

```
  TOPITEM=11;
  data[1] = 13;
  data[2] = 5;
  data[3] = 1;
  data[4] = 11;
  data[5] = 8;
  data[6] = 4;
  data[7] = 14;
  data[8] = 6;
  data[9] = 2;
  data[10]= 9;
}

/* ------------------------------------------------------------------------ */
/*   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
   *
   */

  hwdata();
  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.

Suppose we know how to sort $n-1$ items and are given n items to sort. Clearly we can easily sort the first $n-1$ and then insert the last item into the correct place in the data set. This is exactly how the insertion sort works.

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:

```
typedef int T;
typedef int TblIndex;

#define CompGT(a,b) (a > b)

void InsertSort(T *Lb, T *Ub) {
    T V, *I, *J, *Jmin;

    Jmin = Lb - 1;
    for (I = Lb + 1; I <= Ub; I++) {
        V = *I;
```

```
        /* Shift elements down until       */
        /* insertion point found.          */
        for (J = I-1; J != Jmin && CompGT(*J, V); J--)
            *(J+1) = *J;
        *(J+1) = V;
    }
}
```

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.

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.

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

0.4.1 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.

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.

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

0.4.2 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 dimension) {
  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.

0.4.3 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 operated 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.

Analysis

Dijkstra's algorithm has a complexity of $O(N^2)$.

Source Code

An implementation of Dijkstra's algorithm is given below in pseudo-code.

```
#define N 20

struct vert {
  int label;
  int number;
};

struct vert verts[N];
int matrix[N][N];

int edge_cost(int a, int b) {
  return (matrix[a][b]);
}

void check_vert (vert *begin, int distance) {
  int label = begin->label;

  if (lable == 0) return;
```

```
  else {
    for (i = 0; i < n; i++)
      if (((label + edge_cost(begin->number, i)) == distance) &&
         (reverse_map(i)->label == 0)
             labelvert(i, label + distance);
  }
}

void dijkstra (int s) {
  int distance = 1;

  labelvert(s, 0);

  check_verts(s, distance);
  distance++;
  while (!all_labeled()) {
    for (i = 0; i < N; i++)
      check_vert(i, distance);
    distance++;
  }
}
```

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.

0.4.4 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--;
    }
  }
}
```

0.4.5 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 $2T(n)$

$2T(n) =$		$2n+4$	$+2n+8$	$+2n+16$	$+...$	$2n+2^x$	$+2n+2^{x+1}$
$-T(n) =$	$-n-2$	$-n-4$	$-n-8$	$-n-16$	$-...$	$n-2^x$	
$T(n) =$	$-n-2$	$+n$	$+n$	$+n$	$+...$	n	$+2n+2^{x+1}$

(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) >= 1) {
      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 Exponentiation

0.5.3 Julian Calendar Algorithms

Included here are several Julian calendar routines. The `date_to_julian` is an astronomical calculation that assigns (almost) any date a single (large) integer. 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 date, converting the resultant Julian date 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, 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. Thus, dates before 1582 will produce an error. This calendar corrected the inaccuracies of the original and "old style" calendars. The "new style" Julian Calendar is popularly called the Gregorian Calendar. The "new style" calendar should accurately model the Earth's movement around the sun for some 40,000 more years.

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. Craig, John C. 1988, *Microsoft QuickBASIC Programmer's Toolbox* (Redmond, WA: Microsoft Press), pp. 60-70

2. Press, William H. et al. 1992, *Numerical Recipies in C* (New York, NY: Cambridge University Press), pp. 12-14

0.5.4 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. 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)));
}
```

0.5.5 Fibonacci Calculation

Source Code

```
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);
}
```

0.5.6 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);
}
```

0.5.7 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.

0.5.8 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
//
```

```
//***************************************************************************
// Metaphone::Metaphone()
//
Metaphone::Metaphone()
{
  name = "metaphone";
}


//***************************************************************************
// Metaphone::~Metaphone()
//
Metaphone::~Metaphone() {}


//***************************************************************************
// void Metaphone::generateKey(char *word, String &key)
//
/*
 * 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();
        *n++ = 0;
        *n++ = 0;
        *n++ = 0;
        *n = 0;                 /* Pad with nulls */
        n = ntrans.get() + 4;         /* Assign pointer to start */
        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.

0.5.9 A 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

Note that on DOS/Windows machines you may need to declare the `int`s below as `long`s.

```
int r[100]; // "global" 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 += 1000000000;

  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, " ");

  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).

0.6 Language Parsing Algorithms

0.6.1 Samuelson-Bauer expression 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);
   }
}
```

0.6.2 Shift-reduce bottom-up parsing

0.6.3 Recursive Descent 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. It works fine but makes calls to a queue module that is not included. It should be adequate to help you understand the Banker's algorithm but if you want to use it you will have to provide some queue abstract data type methods like enqIO (enqueue in order), deqmax/min (dequeue the maximum or minimum on a queue), and grepdeq (search a queue and deq the thing with a target key). If you don't know what a queue is I suggest you read the section on queues in this book. The below code is written in ANSI C:

```
/* -- banker.h -- */

#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: alg.tex,v 1.6 1998/03/30 04:41:38 scott Exp scott $            *
*                                                                      *
\**********************************************************************/

#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*/

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);
}
```

0.7.2 UNIX System V Style Semaphores

```
/*-
 * Copyright (c) 1997 Keith Mitchell
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
```

```
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * $Id: alg.tex,v 1.6 1998/03/30 04:41:38 scott Exp scott $
 *
 */

/*
 * NOTE: The member functions of this class throw a MiscError class
 * on error conditions.  This error class must be caught somewhere for
 * the exception handling to work correctly
 */

#ifndef _SEMAPHORE_H_
#define _SEMAPHORE_H_

class Semaphore
{
private:
  int sem;                      // A handle to the semaphore set
  int nsem;                     // Number of semaphores
  int *initval;                 // The initial value for the semaphores
  int *wheresem;                // where in shared memory to store it.

  /*
   * Methods
   */
  void CreateSemaphore();       // Create a new semaphore set

public:
  Semaphore(int, int*);         // <Number of Semaphores>, <initial values>
  Semaphore(int);               // 1 Semaphore, Initial value passed in
```

```
  Semaphore(void *, int, int*); // where to store, how many, init val
  Semaphore(void *, int);       // where to store, how many
  Semaphore(void*);             // where to store
  Semaphore();                  // 1 Semaphore, Initial value 1
  ~Semaphore();

  /*
   * Methods
   */
  void Wait(int);               // Semaphore number passed in
  void Wait();                  // Semaphore number zero
  void Signal(int);             // Semaphore number passed in
  void Signal();                // Semaphore number zero
  int  GetValue(int);           // Semaphore # passed in (first arg)
  int  GetValue();              // Semaphore zero
  void SetValue(int, int);      // Semaphore # passed in (first arg)
  void SetValue(int);           // Semaphore zero
  void RemoveSemaphore();       // Remove a semaphore set
};
#endif

/*-
 * Copyright (c) 1997 Keith Mitchell
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
```

```
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * $Id: alg.tex,v 1.6 1998/03/30 04:41:38 scott Exp scott $
 *
 * $Log: alg.tex,v $
 * Revision 1.6  1998/03/30 04:41:38  scott
 * split into many included files
 *
 * Revision 1.5  1998/03/29 23:08:01  scott
 * added html stuff and an index.
 *
 * Revision 1.1  1998/03/29 23:07:24  scott
 * Initial revision
 *
 * Revision 1.4  1998/01/26 06:39:48  scott
 * hi mom.
 *
 * Revision 1.3  1998/01/22 22:14:15  scott
 * hi mom
 *
 * Revision 1.2  1997/12/02 00:46:01  scott
 * added some sort stuff.
 *
 * Revision 1.1  1997/11/23 16:37:01  scott
 * Initial revision
 *
 * Revision 1.6  1997/04/16 13:58:40  kmitch
 * Add Copyright notice
 * Add RCS id's to everything (static char rcsid[] = "$Id: alg.tex,v 1.6 1998/03/30 04:41:38 scott Exp scott $";)
 * Make sure the sitedef.h include comes first
 * Improve error checking
 *
```

```
 * Revision 1.5  1997/04/14 23:46:13  kmitch
 * #ifdef all of the syslog stuff
 * various other fixes
 *
 * Revision 1.4  1997/04/14 17:28:40  scott
 * Make it actually work
 *
 *
 * Too many individual changes to mention this lifetime
 *
 * Revision 1.3  1997/04/13 17:47:04  scott
 * Add more logging
 * Change the return type from GetEvent in class Protocol
 *
 * Revision 1.2  1997/04/13 03:15:47  kmitch
 * Add support for more than one semaphore per semaphore set
 * Add comments
 *
 *
 */

#include "sitedef.h"

#include <iostream.h>
#include <unistd.h>
#include <errno.h>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#include "MiscError.h"
#include "Semaphore.h"

/*
 * our version id
 */
```

```
static char rcsid[] = "$Id: alg.tex,v 1.6 1998/03/30 04:41:38 scott Exp scott $";

/*
 * Constructor
 *
 * PARAM <num> the number of semaphores to allocate
 * PARAM <val> the value to initialize the semaphore to
 */
Semaphore::Semaphore(int num, int* val)
{
#ifdef DEBUG
  syslog(LOG_PRIO, "Semaphore Constructing");
#endif

  if (val == NULL)
    throw MiscError(__FILE__, __LINE__, "Initial value array is NULL");

  sem = -1;
  nsem = num;
  initval = val;

  for (int i = 0; i < num; i++)
    initval[i] = val[i];

  wheresem = NULL;

  CreateSemaphore();
}

/*
 * Constructor
 *
 * PARAM <num> the number of semaphores to allocate
 *
 * NOTES: This constructor defaults to only one semaphore
 */
Semaphore::Semaphore(int val)
{
```

```
#ifdef DEBUG
  syslog(LOG_PRIO, "Semaphore Constructing");
#endif

  nsem = 1;
  sem = -1;
  initval = new int[1] = {val};
  wheresem = NULL;

  CreateSemaphore();
}

/*
 * Constructor
 *
 * PARAM <num> the number of semaphores to allocate
 *
 * NOTES: This constructor defaults to only one semaphore.
 *        This constructor initializes the semaphore to one
 */
Semaphore::Semaphore()
{
  sem = -1;
  nsem = 1;
  initval = new int[1] = {1};
  wheresem = NULL;

  CreateSemaphore();
}


/*
 * Constructor
 * PURPOSE create a semphore class in shared memory
 *
 * PARAM <vp> an area in shared memory that has the semaphore handle
 * PARAM <howmany> how many semaphores are needed
 * PARAM <val> the initial value array
```

```
 */
Semaphore::Semaphore(void *vp, int howmany, int* val) {

  if (vp == NULL)
    throw MiscError(__FILE__, __LINE__, "Shared memory is NULL");

  if (val == NULL)
    throw MiscError(__FILE__, __LINE__, "Initial value array is NULL");

  wheresem = (int *) vp;

  if (*wheresem != -1)
    {
      sem = *wheresem;
      nsem = howmany;
      initval = val;
    }
  else
    {
      nsem = howmany;
      initval = val;

      CreateSemaphore();
    }
}

/*
 * Constructor
 * PURPOSE create a semphore class in shared memory
 *
 * PARAM <vp> an area in shared memory that has the semaphore handle
 * PARAM <howmany> how many semaphores are needed
 * PARAM <val> the initial value array
 */
Semaphore::Semaphore(void *vp, int howmany)
{
  if (vp == NULL)
    throw MiscError(__FILE__, __LINE__, "Shared memory is NULL");
```

```
  wheresem = (int *) vp;

  if (*wheresem != -1)
    {
      sem = *wheresem;
      nsem = howmany;
      initval = NULL;
    }
  else
    {
      nsem = howmany;
      initval = new int[howmany];

      for (int i = 0; i < howmany; i++)
initval[i] = 1;

      CreateSemaphore();
    }
}

/*
 * Constructor
 * PURPOSE create a semaphore class in shared memory
 *
 * PARAM <vp> an area in shared memory that has the semaphore handle
 */
Semaphore::Semaphore(void* vp)
{
  if (vp == NULL)
    throw MiscError(__FILE__, __LINE__, "Shared memory is NULL");

  wheresem = (int *) vp;

  if (*wheresem != -1)
    {
      sem = *wheresem;
      nsem = 1;
```

```
    }
  else
    {
      nsem = 1;
      initval = new int[1] = {1};

      CreateSemaphore();
    }
}

/*
 * Destructor
 */
Semaphore::~Semaphore()
{
#ifdef DEBUG
  syslog(LOG_PRIO, "Semaphore is destructing");
#endif

  if ((sem >= 0) && (!wheresem))
    RemoveSemaphore();

  if (initval)
    delete initval;
}

/*
 * FUNCTION CreateSemaphore
 * PURPOSE create a new semaphore with read/write access for the
 *         current process's owner and group
 */
void Semaphore::CreateSemaphore()
{
  union semun sun;

  // Create the semaphore
  sem = semget(IPC_PRIVATE, nsem, IPC_CREAT | SEM_R | (SEM_R >> 3) |
        SEM_A | (SEM_A >> 3));
```

```
  if (sem < 0)
    throw MiscError(__FILE__, __LINE__, strerror(errno));

  if (wheresem)
    *wheresem = sem;

  for (int i = 1; i <= nsem; i++)
    {

      // Initialize the semaphore
      sun.val = initval[i-1];

      if (semctl(sem, i-1, SETVAL, sun) < 0)
throw MiscError(__FILE__, __LINE__, strerror(errno));
    }
}

/*
 * FUNCTION RemoveSemaphore
 * PURPOSE Return the semaphore set back to the kernel
 */
void Semaphore::RemoveSemaphore()
{
  union semun sun;

  sun.val = 0;
  if (semctl(sem, 0, IPC_RMID, sun) < 0)
    throw MiscError(__FILE__, __LINE__, strerror(errno));
}

/*
 * FUNCTION GetValue
 * PURPOSE retreive the current value of the semaphore
 *
 * PARAM <snum> the semaphore number to get the value of
 */
int Semaphore::GetValue(int snum)
```

```
{
  int         semval;
  union semun sun;

  sun.val = 0;
  semval = semctl(sem, snum, GETVAL, sun);

  if (semval < 0)
    throw MiscError(__FILE__, __LINE__, strerror(errno));
  else
    return semval;
}

/*
 * FUNCTION GetValue
 * PURPOSE retreive the current value of the 1st semaphore
 *
 * NOTE: this function assumes semaphore zero
 */
int Semaphore::GetValue()
{
  return GetValue(0);
}

/*
 * FUNCTION SetValue
 * PURPOSE retreive the current value of the semaphore
 *
 * PARAM <snum> the semaphore number to set the value of
 * PARAM <value> the new value for the semaphore
 */
void Semaphore::SetValue(int snum, int value)
{
  union semun sun;

  // Initialize the semaphore
  sun.val = value;
```

```
  if (semctl(sem, snum, SETVAL, sun) < 0)
    throw MiscError(__FILE__, __LINE__, strerror(errno));
}

/*
 * FUNCTION SetValue
 * PURPOSE retreive the current value of the 1st semaphore
 *
 * PARAM <value> the new value for the semaphore
 *
 * NOTE: This function assumes semaphore number 0
 */
void Semaphore::SetValue(int value)
{
  SetValue(0, value);
}

/*
 * FUNCTION Wait
 * PURPOSE Allocate one unit of resource
 *
 * PARAM <snum> the semaphore number to wait on
 */
void Semaphore::Wait(int snum)
{
  static struct sembuf sb[1];

  sb[0].sem_op = -1;
  sb[0].sem_num = snum;
  sb[0].sem_flg = 0;

  if (semop(sem, \&sb[0], 1) < 0)
    throw MiscError(__FILE__, __LINE__, strerror(errno));
}

/*
 * FUNCTION Wait
 * PURPOSE Allocate one unit of resource
```

```
 *
 * NOTE: this method operates only on semaphore number 0
 */
void Semaphore::Wait()
{
  Wait(0);
}

/*
 * FUNCTION Signal
 * PURPOSE deallocate one unit of resource
 *
 * PARAM <snum> semaphore number to act on
 */
void Semaphore::Signal(int snum)
{
  static struct sembuf sb[1];

  sb[0].sem_op  = 1;
  sb[0].sem_num = snum;
  sb[0].sem_flg = 0;

  if (semop(sem, \&sb[0], 1) < 0)
    throw MiscError(__FILE__, __LINE__, strerror(errno));
}

/*
 * FUNCTION Signal
 * PURPOSE deallocate one unit of resource
 *
 * NOTE: This function operates only on semaphore number zero
 */
void Semaphore::Signal()
{
  Signal(0);
}
```

0.7.3 Dekker's Algorithm for Mutual Exclusion

0.7.4 Peterson's Algorithm for Mutual Exclusion

0.7.5 A brief discussion of Scheduling Algorithms

0.8 Geometric Algorithms

0.8.1 Convex Hull Problem

Gift Wrapping

Graham's Scan

0.8.2 Closest Pair Problem

0.8.3 Determining Whether a Point is Inside a Polygon

0.9 Data Compression Algorithms

The goal of data compression is to represent a some set of information as 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**.

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.

0.9.1 Integer Coding

Section note: this description and source code are provided courtesy of Hugh Williams.

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 9 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 parameterised 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 parameterised 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 occuring 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

```
/*--- 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 * sizeof(char));
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);
}
```

References

1. Witten et al. 1994, *Managing Gigabytes: Compressing and Indexing Documents and Images.* (New York, NY: Van Nostrand Reinhold).

0.9.2 Static Huffman Coding Trees

Huffman compression was first proposed in a paper by D. A. Huffman in the early 1950's. When a Huffman style compression routine analizes an input file it carefully tallies character frequency data. For each character in the input stream a cooresponding weight or frequency is stored. Next Huffman algorithms build a full binary tree with leaves which are labeled with the weights on the input character list. Initially there are n distinct trees of one node each. Each of these n trees represents a letter and its corresponding weight. At each step in the tree building algorithm the trees corresponding to the smallest weights are combined into a tree whose weight is the sum of its two components until, at the end, 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.

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 scanned character by character. For each character read, the equivalent codeword is computed and appended to the output stream.

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 is usually acceptable. 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.

Analysis

Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include ``huffman.h''

int bytes_in;
struct huff_node {
  int count;
  int left, right, parent;
```

```
  char *code;
  int bit_count;
} tree[MAX_NODES];



int root;


static int huff_scan (FILE *infile) {
  int c;

  bytes_in = 0;
  while ((c = fgetc(infile)) != EOF) {
    tree[c].count++;
    bytes_in++;
  }

  if (ferror(infile)) {
    return(1);
  } else {
    rewind(infile);
    return(0);
  }
}


static void huff_walk (int k, int depth) {
  int dir, bitno, set, byte, bit;
  int w, trace;

  if (tree[k].left = -1;
```

0.9.3 Adaptive Huffman Codes

Adaptive Huffman coding is a **defined-word** scheme which determine the mapping from source messages to codewords based upon a running estimate of the source message frequency probabilities. The algorithm is adaptive, changing so as to remain optimal for the estimates it has computed thus far. Adaptive Huffman codes respond to locality of reference, the tendency for a codeword to appear frequently in one area of the input ensemble and then fall into disuse for a long period. The encoding process essentially learns about the characteristics of the data it is processing as it proceeds. For this reason the decoder must, also, learn by continually updating the Huffman tree during the decoding process.

One advantage of adaptive Huffman coding systems is that they require only one pass over the input. Because the statistical analysis of input token frequency occurs during the compression process, not before, as in static Huffman coding, the adaptive systems are often used in real-time or time-critical compression systems.

0.9.4 Lempel-Ziv-Walsh Codes

0.10 Game-Playing Algorithms

0.10.1 Minimax Search

0.10.2 Alpha-Beta Pruning

0.10.3 Chess Programming

0.11 Data Integrity

0.11.1 Checksums

0.11.2 Weighted Checksums

0.11.3 Cyclic Redundancy Checks

CRC-CCIT

CRC-16

CRC-32