A Little Bottle ... with the words 'DRINK ME',

(or Adaptive data compression from binary tree rotations)

Due: Tuesday, October 1, 10:00 p.m.

1 Binary Trees and Adaptive Data Compression

1.1 Introduction

In this assignment you are to implement an algorithm for *data compression.* The purpose of data compression is to take a file A and transform it into another file B in such a way that (1) B is smaller than A, and (2) it is possible to reconstruct A given B. It is desirable that the transformation be efficient, and that file B be a lot smaller than A. A program that converts A into B is called a *compressor*, and one that undoes this operation is called an *uncompressor.* A method that can compress all files is not possible. (It is easy to see this: what would happen if you just kept iterating the method, compressing the output of the compressor?) Since we cannot compress all files, the goal of data compression must be to compress files typically found on computers. A number of powerful methods have been discovered to do this.

1.2 Prefix codes

For simplicity, suppose we have a set of five letters: A, B, C, D, E. For each of these letters, a codeword will be assigned. These codewords will be binary, and have the property that no one is a prefix of any other. Such a set of codewords is called a *prefix code.* (Prefix codes also have the property that any sequence of bits can be partitioned into codewords in a unique way.) There is a one-to-one correspondence between binary prefix codes for N letters, and binary trees with N leaves where every internal node has exactly two children. Figure 1 is an example of a prefix code and corresponding tree for the five letters above.

Here is the correspondence: the codeword for a leaf is just a binary string representing the path from the root of the tree to the leaf, where "left" corresponds to "0" and "right" corresponds to "1". Given a tree, any sequence of bits can be turned back into letters in this way:

> Start at the root, and walk down the tree as you read the bits, going left if you read a 0 and right if you read a 1. When you come to a leaf you've completed one codeword (so, print out its letter). Go back to the root, and continue.

```
A: 00                               o    <--Root
B: 010                             / \
C: 011                            /   \
D: 10                            /     \
E: 11                           o       o   <--Internal node
                               / \     / \
                              A   o   D   E   <--Leaf
                                 / \
                                B   C
```

Figure 1: A prefix code binary tree with five letters

Note that the codewords vary in length. By constructing a code in which the number of bits needed for the common letters is very small, we might obtain a way to represent a sequence of letters in fewer bits. (A *Huffman code* is the binary code that is the most efficient for a given distribution of letters. It is perhaps the single best known scheme; years ago I used to teach it in data structures.)

1.3 Adaptive Data Compression

In this assignment you'll be implementing an *adaptive* data compression algorithm, based on prefix codes. This algorithm is continually changing the code tree as it reads its input. Among the advantages of such an adaptive method are: (1) If one region of the input file to be compressed has a very different distribution than another region, then the adaptive method can be more efficient than a static optimal Huffman code. (2) It is not necessary to make a preliminary pass over the data to compute the optimal Huffman tree, (3) The optimal static tree does not have to be transmitted along with the codewords.

By the way, most of the popular data compression programs (e.g., Unix "compress" and "gzip") use adaptive algorithms, although they are not based on prefix codes as described here.

Our adaptive compression algorithm starts out with a tree that is well balanced. The first letter is sent using the code for it in this initial tree. Then the tree is restructured in a way that depends on the letter just sent. The uncompressor sees the code for the first letter, puts out the first letter, and restructures the tree in exactly the same fashion. The compressor and the uncompressor are in lock-step, and each character of the file is sent using a different prefix code.

The method for restructuring the code tree is based on the premise that if a letter occurs in the input, then it is likely that that letter will be occur again in the near future. Thus, each time a letter occurs, it is moved closer to the root of the tree (thereby shortening the codeword needed to represent it).

```
        x                     v
       / \                   / \
      v   y    ======>      u   x
     / \                       / \
    u   w                     w   y
```

Figure 2: A right rotation

Restructuring the tree

Let L be the leaf whose code has just been sent. The restructuring operation is divided into two parts. First, a subset of the nodes along the path from L to the root have their left and right children swapped, so that the path from the root to L becomes the leftmost path in the tree.

Next, this leftmost path is made shorter by a method called *left-splaying.* Left-splaying cuts the length of the path from the root to L by about a factor of two.

Left-splaying starting from a node x is defined as follows: If x is a leaf or if x's left child is a leaf, do nothing. Otherwise, do a local restructuring operation, called a *right rotation.* (See Figure 2.) Then do a left-splaying operation starting at the new left child (u in Figure 2).

Note that u, w, and y are arbitrary subtrees (they could be leaf nodes or have a large number of nodes below them).

Thus the left splay step continues from u on down the left path, until it reaches a leaf.

In our algorithm, we are always beginning this left splaying operation at the root. Figures 3 and 4 are an example of what happens when the input letter `7` is processed by this algorithm. We start with the tree in Figure 3. After making the path to 7 the leftmost path, and then left-splaying 7 we have Figure 4. Note that two right rotations are done in this particular left-splaying step.

2 The Assignment

You are to write two programs, `shrink` and `grow`, that implement this algorithm on an input alphabet of size 256. This is a very natural alphabet size, since in ASCII there are 8 bits per byte, and thus 256 possible values for a byte. The program will read the bytes of its input one by one, and construct a sequence of bits (by concatenating the codewords generated by the algorithm), pack these bits into bytes, and output them.

There are three useful functions for dealing with bits in the file `bit_utils.cc`:

- `int get_bit(istream& myin)` This reads one bit of the input stream `myin` (normally you'd call it with `cin`), and returns it. If there are no more bits, then it returns -1.

```
              o
             / \
            /   \
           /     \
          /       \
         /         \
        /           \
       /             \
      o               o
     / \             / \
    /   \           /   \
   /     \         /     \
  o       o       o       o
 / \     / \     / \     / \
0   1   2   3   4   5   6   o
                           / \
                          7   8
```

Figure 3: Initial tree (the code for 7 is 1110)

```
                o
               / \
              /   \
             /     \
            /       \
           /         \
          /           \
         /             \
        o               o
       / \             / \
      /   \           /   \
     /     \         /     \
    o       o       o       o
   / \     / \     / \     / \
  o   6   4   5   0   1   2   3
 / \
7   8
```

```
        o
       / \
      /   \
     /     \
    /       \
   /         \
  o           o
 / \         / \
7   o       /   \
   / \     /     \
  8   6   o       o
         / \     / \
        4   5   /   \
               /     \
              o       o
             / \     / \
            0   1   2   3
```

Figure 4: Make 7 the leftmost path, then left-splay

- `void put_bit(ostream& myout, int b)` This puts one bit into the output stream `myout` (normally you'd call this with `cout`).
- `void put_finish(ostream& myout)` If the number of bits sent so far is not a multiple of eight, this puts out enough zeros to make it so.

Note that the compressed version of a file may not have a length in bits that is a multiple of eight. In this case we will have a problem packing it into bytes, since the end of any file in Unix must end on a byte boundary. If we fill in the remaining bits with zeros (by calling `put_finish()`) this may confuse the uncompressor. It may view those remaining zeros as codewords. In order to deal with this problem, the compressor will send a special terminating code to indicate the end of the file. So instead of having 256 leaves, the code tree will have 257. Leaves numbered $0 \cdots 255$ are for sending the bytes of the input file, and a special leaf numbered 256 will correspond to a special code indicating the end of the input. The compressor writes this special code just before it calls `put_finish()` and quits. When the uncompressor sees this, it realizes the input is finished, and quits.

Note that your programs take their input from the "standard input", and write their output to the "standard output". Data can be supplied to the standard input in several ways. You can use use `<` to get the input from a file, and `>` to put the output into a file. For example:

```
shrink < junk.orig > junk.small
```

Which runs the file `junk.orig` through the compression program and puts the result into another file called `junk.small`.

2.1 Data-types and functions you should use

To help you organize your work, and to make it easier for us to grade (and give partial credit), your solution should follow the following outline.

Your binary tree class should *not* be templated. Your tree nodes should have the following fields. (You can add additional fields if you like.):

```
int letter;  // -1 if internal node, else the letter this leaf represents.
TreeNode* parent, left, right;
```

Your `Tree` class should have (at least) the following four member functions:

constructor `Tree(int size)` This builds a balanced tree with `size` leaves. The left and right subtrees of every internal node should either have the same number of leaves, or the right subtree should have one more leaf than the left subtree.

You'll probably find it convenient to have this routine initialize the `letter` fields of the tree, either to zero, or their correct final values. You may add more arguments to this function if you want.

`void output_code(TreeNode start) const` Given a leaf, this should output the code for it, one bit at a time. (using functions from `bit_utils`).

`void make_path_leftmost(TreeNode start)` This swaps left and right children of nodes along the path from start to the root in such a way that start becomes a member of the leftmost path down from the root.

`void left_splay(TreeNode start)` This does the left splay operation starting from the specified node, and continuing down to the left.

2.2 What you should turn in

You will write and turn in many files, almost all of which will be pretty small (excluding comments, which will of course be voluminous!). At the least, there will be files called `shrink.c`, `grow.c`, `tree_utils.c` (and .h), code and header files for the bit utilities (described below) assuming that you do that part of the assignment, Makefile, and `README`.

`shrink.c` is the compression program, and `grow.c` is the uncompression program. `tree_utils.c` will contain all the functions that are used by both the shrink and grow programs. Each of those two has a "main;" so neither needs a .h file.

The file called `README` should list all known bugs and what you know about what might be causing them.

2.3 Hints

First write the four member functions of the `Tree` class mentioned above. Then write the `main` routines for each part (shrink and grow). The `main` routines will be pretty short. Try to write your `Tree` member functions so they can be used for both compressing and uncompressing. You may, of course, add additional methods if you wish.

Leave the bit packing and unpacking for after you have more or less everything else taken care of. For initial development, just have `shrink` output, and `grow` read plain old `int` 1 and 0. (Of course, at this point, `shrink` will in fact blow up the size of the file considerably.)

You may find it convenient to pass in a few more parameters to the `Tree` constructor. Or you may find it convenient to use some global variables here (they're not *always* bad).

Many of the functions are very easy if you use recursion.

Indeed, a correct solution for this assignment probably involves considerably *fewer* lines of code than the first assignment.

Remember: for this assignment, a node is a leaf if its *children* are `NULL`.

It is easy to get into trouble forgetting to set parent pointers correctly.

For compression, you need, given a character such as `'a'`, to be able to find its associated leaf in the tree. What would be an efficient way to implement this? (Miserably inefficient implementations will loose some points.)

3 Coping with bits as data

One problem that will confront you in this assignment is that you must work with *bits*, as opposed to, say, ints or chars. I believe that the `bit_utils` functions will shield you from almost all of this issue, but here's some facts just in case.

Bit hackery in C

Remember that a char consists of exactly 8 bits.

The integer constant 1 always has the bit pattern of a single 1 preceded by 0's; the integer constant 0 is always all 0s.

The operators on bits that you are likely to need are

```
bitwise OR:               |
bitwise AND:              &
left shift                <<
right shift               >>
```

`(Operators |= and &= are also provided.)`

For the shift operators, the first argument is the thing you want shifted, and the second argument is the number of positions that you want in shifted by.

For example, to find out whether bit position `p` of character `c` is 0 or 1 we could use the expression

```
(c >> p) & 1
```

If instead, we want to set position `p` of character `c` to be be 1 if the value of (char or int) `flag` is 1, and leave `c` unchanged if `flag` is 0, we would say

```
c |= flag << p;
```

4 Does it work?

By the way, how well does this method of data compression work? On what types of files does it work well, and on which type does it work poorly? (You don't need to hand this in; it's just something to think about.)

5 Submission

Deadline: Your code must be electronically submitted using the turnin program on the departmental machines by *10 p.m. Tuesday, October 1.*

You must provide us with a working makefile.

Please DO NOT hand in `.o`, `core`, or executable files—they just take up space on the turnin directory.