Computer Engineering 12
Project 5: Being up a Tree, You’re in a Heap of Trouble

Due: Sunday, Mar. 12nd at 11:59 pm

1 Introduction

Well, Professor Gosheim Loony has done it again! He has written part of a project, but did not have time to finish it, so he has given you that task. Professor Loony has been working on a file compression project using Huffman coding. This time around, he has written the function to perform the actual compression given the Huffman tree, but still needs to construct the tree. (Are you really surprised that Professor Loony has left **you** up a tree?) Of course, being a believer in abstract data types, his function makes use of a binary tree abstract data type. Therefore, for this assignment, you will need to write a binary tree ADT that conforms to Professor Loony’s interface.

2 Interface

The interface to your abstract data type must provide the following operations:

- struct tree *createTree(int data, struct tree *left, struct tree *right);
  return a pointer to a new tree with the specified *left* and *right* subtrees and *data* for its root

- void destroyTree(struct tree *root);
  deallocate memory for the entire subtree pointed to by *root*

- int getData(struct tree *root);
  return the data in the root of the subtree pointed to by *root*

- struct tree *getLeft(struct tree *root);
  return the left subtree of the subtree pointed to by *root*

- struct tree *getRight(struct tree *root);
  return the right subtree of the subtree pointed to by *root*

- struct tree *getParent(struct tree *root);
  return the parent tree of the subtree pointed to by *root*

- void setLeft(struct tree *root, struct tree *left);
  update the *left* subtree of *root*

- void setRight(struct tree *root, struct tree *right);
  update the *right* subtree of *root*

3 Implementation

As required by Professor Loony, you will write a binary tree abstract data type, incorporating parent pointers as well as pointers to the two children. All operations except destroyTree are required to run in *O*(1) time. The teaching assistant, Mr. Noah Tall, points out that you are building a binary tree ADT, **not** a binary search tree ADT.

3.1 TreeSort

To test your abstract data type, Professor Loony has written a simple sorting algorithm called **treesort**. (Professor Loony is a simple sort of person after all.) The sorting algorithm reads integers from the standard input, inserts them into a binary search tree, and then performs an inorder traversal of the tree to print the integers in sorted order.

Mr. Noah Tall out that inserting *n* values into a binary search tree requires *O*(*n* log*n*) operations on average and *O*(*n*2) operations in the worst case. When compared with other sorting algorithms, it is not a particularly efficient algorithm, requiring significant overhead in both time and space. Other, more efficient algorithms such as heapsort and quicksort would outperform treesort in practice. Nevertheless, it does show how easy it is to sort data using a binary search tree.

3.2 Huffman Coding

Huffman coding is a variable-length coding technique that can be used for lossless data compression (i.e., the original data can be reconstructed exactly from the compressed data). Lossless data compression is in contrast to lossy compression, in which the original data cannot be reconstructed exactly (i.e., some information is "lost" during compression). Professor Loony (who some say lost it years ago) asks you to write a utility to perform compression on a file using Huffman coding. Although Huffman coding has been largely replaced withmore advanced file compression schemes, it is still used today in parts of JPEGand MP3 encoding aswell as the bzip2 compression utility. Professor Loony has outlined the basic steps for you:

1. Count the number of occurrences of each character in the file.
2. Create a binary tree consisting of just a leaf for each character with a nonzero count. The data for the tree is the character's count. Also, create a tree with a zero count for the end of file marker (see below). Insert all trees into a priority queue.
3. While the priority queue has more than one tree in it, remove the two lowest priority trees. Create a new tree with each of these trees as subtrees. The count for the new tree is the sum of the counts of the two subtrees. Insert the new tree into the priority queue.
4. Eventually, there will be only one tree remaining in the priority queue. This is the Huffman tree. Incidentally, the data at the root of this tree should equal the number of characters in the file.
5. Once the tree is constructed, a bit encoding can be assigned to each character by starting at the root and walking down the tree toward the leaves. Traditionally, each left branch taken results in a zero bit, and each right branch taken results in a one bit.

The UNIX utilities pack and unpack perform compression using Huffman coding. These utilities can still be found on the Solaris machines in the Design Center. The pack utility was the predecessor of the compress utility, which itself was a predecessor of other compression tools such as gzip and bzip2. In fact, gunzip can still decompress files created using pack. The actual file format used by pack is quite arcane and convoluted, so Professor Loony (who is quite arcane and convoluted himself) has written the function for you to use:

```
void pack(char *infile, char *outfile, struct tree *leaves[257]);
```

where infile is the name of the file to be compressed, outfile is the name of the file for the compressed data, and leaves is an array of pointers to the leaves in the Huffman tree indexed by character. In other words, leaves[c] points to a leaf in the Huffman tree if c occurs in the input file and is NULL otherwise. The leaf for the end of file marker is stored in leaves [256]. (It is unlikely that the last byte of a Huffman-encoded file contains only valid data. More than likely, the last byte has some extra, unused bits, so we need to use an explicit end of file marker.)

Your program will require two filenames as command-line arguments. The first filename is the file to be compressed, and the second filename is the file to hold the compressed data. Your job is to construct the Huffman tree for the file and display the codes for each character to the standard output. The following example is for "the fat cat sat on the mat":

```
012: 1 000000
' ': 6 10
'a': 4 011
'c': 1 00001
'e': 2 0011
'f': 1 00010
'h': 2 0100
'm': 1 00011
'n': 1 0101
'o': 1 00100
's': 1 00101
't': 6 11
400: 0 000001
```

A printable character is written in single quotes, and a nonprintable character is written as its three-digit octal value. Mr. Noah Tall suggests using the isprint function declared in <ctype.h> to determine if a character is printable. (Admittedly, this is one of Noah's more useful suggestions, but he does tell you to "read the man page" to figure out how it works.) After displaying the characters, occurrences, and bit encodings, your program will call Professor Loony's pack function to perform the actual compression. If everything works correctly, you can uncompress your output file using gunzip and get back your original file!

Note that, in general, a Huffman code need not be unique. For example, any of the five-bit codes listed above could be exchanged with one another and the result would still be correct. However, a valid Huffman code always minimizes the length of the encoded input. The length can be computed by multiplying the frequency of each character by the length of its code and summing each product. Finally, for the priority queue itself, Professor Loony requires you to use a **binary heap**, implemented using an array as discussed in class and in the text.

4 Submission

Create a directory called project5 to hold your solution. Call the source file for the binary tree implementation tree.c and your source file for the huffman coding application huffman.c. Submit a tar file containing the project5 directory using the online submission system.

5 Grading

Your implementation will be graded in terms of correctness, clarity of implementation, and commenting and style. Your implementation **must** compile and run on the workstations in the lab. The algorithmic complexity of each function in your tree abstract data type **must** be documented.