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

Due: Sunday, June 2nd 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?)

The tree data type itself is easy to construct as it consists simply of a parent pointer and a value. Therefore, Professor Loony does not require you write a tree ADT type. However, since one of the key components of the Huffman coding algorithm is a priority queue, he does require you to write a priority queue ADT for this assignment.

2 Interface

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

- `PQ *createQueue(int (*compare)());`
  return a pointer to a new priority queue using *compare* as its comparison function
- `void destroyQueue(PQ *pq);`
  deallocate memory associated with the priority queue pointed to by *pq*
- `int numEntries(PQ *pq);`
  return the number of entries in the priority queue pointed to by *pq*
- `void addEntry(PQ *pq, void *entry);`
  add *entry* to the priority queue pointed to by *pq*
- `void *removeEntry(PQ *pq);`
  remove and return the smallest entry from the priority queue pointed to by *pq*

3 Implementation

As required by Professor Loony, you will write a priority queue abstract data type using a binary heap implemented using an array. Operations to add and remove entries are required to run in $O(\log n)$ time. Unlike previous data types that used arrays, the `createQueue` function does not require the maximum number of values as a parameter. Instead, Professor Loony wants you to increase the length of your array when it reaches capacity so it dynamically increases to accommodate the number of values. Thus, we eliminate the need for the client to inform us how large of an array it requires.

Fortunately, this technique is easy in C using the function `realloc`, which reallocates an array to a given size and returns a pointer to the new array. However, reallocation can be expensive, as it may require allocating a completely new array and then copying the data from the old array to the new array. Therefore, we don't want to do it too often. A common technique is to start the array at a small size (we'll use ten elements). When the array reaches capacity (i.e., the number of stored values equals the length of the array), then double the array's length using `realloc`. Doubling the array's length may seem excessive, but it guarantees that for the next n insertions, we won't need to do a reallocation. So, although we may have had to copy n values in a reallocation, the next n values inserted won't require a reallocation, so the ***amortized*** cost over time is $O(n)/O(n) = O(1)$. (Note that this technique is not specific to a binary heap, but works with any array, so we could go back and reimplement our unsorted and sorted array implementations in the same manner.)

3.1 Heapsort

To test your abstract data type, Professor Loony has written a simple sorting algorithm based on ***heapsort***. (Professor Loony is a simple sort of person after all.) The sorting algorithm reads integers from the standard input, inserts them into a priority queue, and then repeatedly removes and prints the smallest value from the queue in order to print all the integers in sorted order.

Mr. Noah Tall notes that inserting and then removing n values into a binary heap requires $O(n \log n)$ operations in the worst case, which is theoretically the best you can do (for a comparison-based sorting algorithm). In reality, heapsort is implemented in a clever way that uses a single array to store the data to be sorted, the binary heap, and the sorted output, so it requires no extra space. However, the principles behind heapsort are exactly those used in Professor Loony's simple implementation, which shows how easy it is to sort data using a binary heap.

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 with more advanced file compression schemes, it is still used today in parts of JPEG and MP3 encoding as well 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. 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, `gzip` 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 node *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 total number of occurrences and length of the Huffman code for each character to the standard output. The following example is for "`the fat cat sat on the mat`":

```
012: 1 x 5 bits = 5 bits
' ': 6 x 2 bits = 12 bits
'a': 4 x 3 bits = 12 bits
'c': 1 x 6 bits = 6 bits
'e': 2 x 4 bits = 8 bits
'f': 1 x 4 bits = 4 bits
'h': 2 x 4 bits = 8 bits
'm': 1 x 5 bits = 5 bits
'n': 1 x 5 bits = 5 bits
'o': 1 x 5 bits = 5 bits
's': 1 x 5 bits = 5 bits
't': 6 x 2 bits = 12 bits
400: 0 x 6 bits = 0 bits
```

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 lengths, 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 `gzip` and get back your original file!

Note that, in general, a Huffman code need not be unique. 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.

4 Submission

Download the `project5.tar` file from the course website to get started. Call your source files `pqueue.c` and `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 abstract data type ***must*** be documented.