ECE190 MP5 - Text Compression with Huffman Coding, Spring 2010

ASCII coding is inefficient. Compare for instance the traps PUTS and PUTSP in the LC-3 ISA. The PUTS trap requires a single 8-bit ASCII character to be placed in a single 16-bit memory location. However, the PUTSP trap requires two 8-bit ASCII characters to be placed in one 16-bit memory location on the LC-3. For a finite amount of memory, for instance 100 locations, PUTSP can store twice as many characters as PUTS. This is the idea behind the following programming assignment, but we will take it farther and utilize a more efficient coding method than ASCII to store text. The coding method is called Huffman coding and is in fact as efficient as a coding scheme can be--it is provably optimal!

Consider the length 8 input string "aaaabbcd". How many different letters does this string contain? Four: 'a' occurs 4 times, 'b' occurs 2 times, and both 'c' and 'd' occursonce. The total string length is 8. Thus, 'a' occurs half the time (the most frequently), 'b' occurs 2/8 of the time, and both 'c' and 'd' occur 1/8 of the time (the least frequently). The compression idea we are after is to assign a unique binary string for each letter, where the most frequently occurring letter gets the shortest string so that it takes the least amount of memory to store, and the least frequently occurring letter gets the longest string. For this example, such an encoding would be: 'a' is "1", 'b' is "01", 'c' is "001", and 'd' is "000".

The unique binary string for each character actually has a stronger property than uniqueness--it is prefix-free, which gives the ability to avoid confusing the binary strings of letters. This has a very elegant relationship with the finite-state machines studied earlier in the course.

Compare the binary strings for each letter to ASCII encoding for each letter. In ASCII (and neglecting any architecture-specific details, such as the 16-bit addressability considered above in the LC-3 example) any string of length 8 takes (8 characters)*(8 bits per character)=64 bits. In Huffman code, we see that this takes only (4 'a's)*(1 bit per 'a')+(2 'b's)*(2 bits per 'b')+(1 'c')*(3 bits per 'c')+(1 'd')*(3 bits per 'd')=14 bits! Even in the worst case where the length 8 string is composed of 8 different letters, such as in "abcdefgh", the encoding would be 20 bits, still a large improvement over the 64 bits required for ASCII.

Milestone 1 - Huffman Tree Generation and Text Compression

The task for the first milestone is given a text file of ASCII characters, take the file's contents (assumed to be less than some fixed length to avoid annoying edge cases, see the given code), generate a binary tree to determine an optimal encoding and encode the given string. Note that we will be skipping an important step--the output of your program will still be in ASCII, just written as the codes. This saves a lot of tedious work. So for the example "aaaabbcd", the input and output of the program are:

```
./mp5.1 -c tests/test0.txt
```

test.txt contents are: `aaaabbcd`

(with no newline or that would also be encoded, see the given test files)

where -c stands for "compress" and executing mp5 on the test0.txt file creates an output file tests/test0.txt.hc with contents: `11110101001000`
(these are ASCII ones and zeros, so '1' and '0', with no newline at the end, see the given test files)

The major steps of the Huffman coding algorithm are:

1. Read ASCII text from a specified input file into a buffer in memory. THIS MUST BE DONE USING `fopen()` and `getc()` (or `fscanf`, just some function that works with file streams).

2. Generate an initial forest of Huffman trees based on the contents of this buffer. A Huffman tree is a binary tree. A forest is a disjoint union of trees--for our purposes, a forest is just a singly linked-list. The data type for the Huffman tree and the forest is a singly-linked list of binary trees:

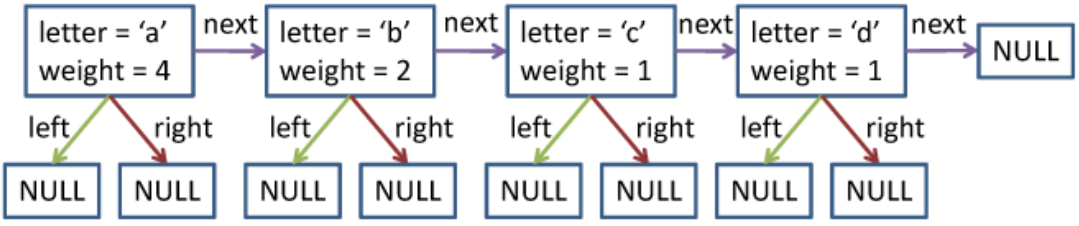
```
typedef struct htree_t {
        char letter;  /* letter represented by this tree */
        int weight;   /* number of times letter appears */
        struct htree_t* left;  /* pointer to left subtree */
        struct htree_t* right; /* pointer to right subtree */
        struct htree_t* next;  /* pointer to next tree in the forest */
} HTREE;
```

   where letter is the letter specified by the tree, weight is the number of times the letter appears in the ASCII text, left is the pointer to the left child, right is the pointer to the right child, and next is the pointer to the next Huffman tree in the linked-list.

   Implement the function `appendForest` to add new trees to the forest, and implement the function `searchForestLetter` to determine if a letter already has a corresponding tree in the list of trees and you should increment its weight.

   You must use `malloc()` for the generation of new entries of this data structure as you will be appending, updating, and removing entries from the forest as a part of the algorithm.

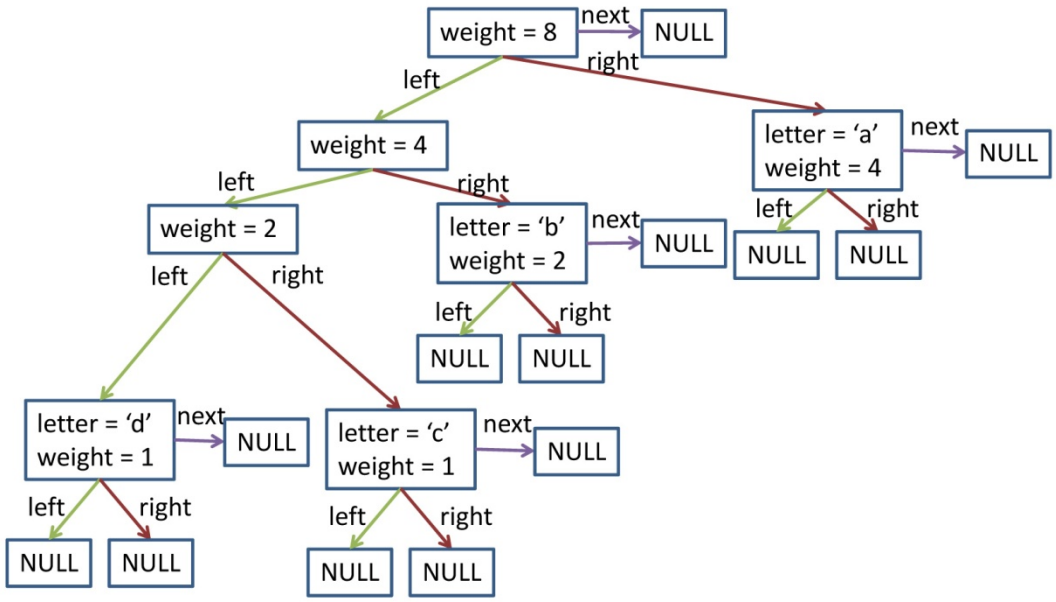
   The initial forest of Huffman trees for the example of "aaaabbcd" is:



3. Run the Huffman coding algorithm on the initial forest. This is done with a call to the `huffmanize` function (**hint**: recall pass-by-value versus pass-by-reference):

    a. Find the tree (called `smallestTree`) in the forest with the lowest weight: ties are broken by taking the last occurrence of the lowest weight tree using the functions `searchForestMin` and `searchForestMinHelp`. In the example above, this would be the tree corresponding to the letter 'd'. COMPARE TO THE GOLD FILE OUTPUT AS THERE ARE MANY POSSIBLE UNIQUE ENCODINGS, BUT YOU MUST BREAK TIES IN THIS WAY OR YOU WILL RECEIVE NO CREDIT BECAUSE YOUR OUTPUT WILL BE DIFFERENT THAN THE GOLD FILE.
    b. Remove this tree from the forest using the function `cutForest`. **Hint**: do not `free()` the memory, just change the pointers in the list.
    c. Repeat 3a and 3b to find and remove the next smallest weight tree, called `smallTree`. In the example above, this would be the tree corresponding to the letter 'c'.
    d. Create a new tree called `newTree` with no letter, weight equal to the sum of the weights of `smallestTree` and `smallTree`, left equal to `smallestTree`, and right equal to `smallTree`.
    e. Add `newTree` to the forest using `appendForest`.
    f. Recursively repeat 3a to 3e until there is a single tree in the forest (that means until next = NULL for every tree in the list). The single tree in the forest is called the **root** of the Huffman tree.

    For the example above, the tree is:



4. From this single Huffman tree, there is now a unique encoding for each letter in the tree, where following a left pointer in the tree yields a value of '0' and following a right pointer in tree yields a value of '1'.

   So for instance, ‘a’ is encoded simply as the binary string “1” since it follows one right pointer, and ‘b’ is encoded as the binary string “01” since this letter is at a the path one left pointer, one right pointer, ‘d’ is encoded as “000” since it is along three left pointers, and ‘c’ is encoded as “001” since it is two left pointers and then a right.

5. Finally, using `fopen`() and `fprintf`(), create a file called <inputFileName>.hc where <inputFileName> is whatever file was specified when calling mp5.

   THIS FILE MUST BE NAMED THE SAME AS THE INPUT FILE NAME WITH .hc APPENDED OR YOU WILL RECEIVE NO CREDIT.

We will use the following command to compile your code, so make sure the file is called mp5.1.c.

```
gcc -Wall -g -ansi mp5.1.c -o mp5.1
```

Milestone 2 - Decompression

To perform decompression, there is a small addition that must be made in the compression step: the Huffman tree must be encoded in some way and placed at the beginning of the compressed file. There are varieties of ways to do this, but we will choose the following. This technique is an example of *serialization*.

For the example of “aaaabbcd”, the serialized Huffman tree is a string: “a;4;b;2;c;1;d;1;\n”, which just means that the letter ‘a’ has weight 4, the letter ‘b’ has weight 2, the letter ‘c’ has weight 1, and the letter ‘d’ has weight 1. This is a semicolon-delimited string.

This string must be stored at the beginning of the compressed file to be used by the decompression algorithm, so modify the compression code appropriately, for the example the output file test0.txt.hc contains:

a;4;b;2;c;1;d;1;
11110101001000

(where there is a new line at the end of the first line and there is no new line at the end of the second line).

The initial forest of trees recreated from this header is the same as before.

YOU MAY ASSUME THAT THE INPUT TEST WILL NOT CONTAIN ANY SEMICOLONS (';' characters). With this assumption, you can easily recreate the initial forest of trees from this header description by using the `strtok` function. This should be implemented in the

Upon creating this initial forest of trees, simply run the `huffmanize` function you implemented in Milestone 1, and now you have the Huffman tree used to encode the file in memory (the same tree above).

Now given the Huffman tree used to encode the compressed file, the decompression algorithm is simple: read a symbol---call this `sym` for the following discussion---from the Huffman encoded file while keeping a pointer in the tree, starting at the root. If the current pointer in the Huffman tree corresponds to a tree with a letter that has been initialized to a value other than '\0', you have reached a leaf of the tree, which contains an ASCII character. Write this character to a buffer. If this is not the case, then the tree is not a leaf. If `sym` is '0', look at the left subtree and repeat. If `sym` is '1', look at the right subtree and repeat.

Finally, write the buffer of decoded characters to a file named <filename>.dc. Where .dc stands for decompressed. Here is an example of the executable being called as intended:

```
./mp5.2 -d tests/test0.txt.hc
```

This creates a file called test0.txt.hc.dc, which must have contents exactly equal to test0.txt. In the case of the "aaaabbcd" example, the contents of test0.txt.hc.dc would be that same string (with no newline). If you are confused, look at the test files and play with the gold code.

At this point, you have gone full circle: you compressed a file using Huffman coding and decompressed a Huffman encoded file. (Again, modulo the detail about this representation being in ASCII, but the point is valid nonetheless.) The complete compression and decompression sequence is:

```
./mp5.2 -c tests/test0.txt
./mp5.2 -c tests/test0.txt.hc
diff tests/test0.txt tests/test0.txt.hc.dc
```

The first line creates the file test0.txt.hc and the second line creates the file test0.txt.hc.dc.

We will use the following command to compile your code, so make sure the file is called mp5.2.c.

```
gcc -Wall -g -ansi mp5.2.c -o mp5.2
```

Hints

- Initialize buffers for strings using `calloc`

- Initialize the elements of the Huffman forest using `malloc`
- Review pointers, make liberal use of gdb will constructing the initial forest to look at the next, left, and right pointers. The graphical version is extremely useful in this regard. Likewise, you can use the printTree function given, but if your tree is bad, this may go into an infinite loop or be otherwise not helpful.
- There are two different forms of recursive functions being implemented. Most are pass-by-reference (such as appendForest) and utilize this double indirection (such as HTREE**) to modify the Huffman tree. In these cases, the recursive argument is accumulated as a parameter. The cutForest function on the other hand relies on returning the recursive result and the pointer to the Huffman tree must be reassigned based on the return value of the function.

Specifics

- When you turn in your code, your program files must be called mp5.1.c and mp5.2.c.
- You must implement the required functions. YOU MAY NOT MODIFY THE ARGUMENT LISTS. You are not required to use additional functions. You may if you think they are helpful.
- You may not include any additional header files (other than the ones already included).
- You may not use any additional global variables (other than the ones provided).
- You may define additional constants to improve the readability of your code. (i.e., using #define statements).
- Your code should be in good coding style, with proper indentation on each line to indicate functional blocks and explanatory comments. A brief header describing the program and headers for main function and support functions are important.
- Your code must pass compilation. You will not receive any functionality points on a code that fails to generate a binary. You will also lose points for compiler warnings.
- Turn in your code using the ECE190 handin script. Your files must be called mp5.1.c and mp5.2.c. We will NOT grade files with any other name. To turn in the code, first login to any EWS machine and type ece190. Change to the directory containing your code and type (for respectively milestone 1 and 2)

  ```
  handin --MP 5.1 mp5.1.c

  handin --MP 5.2 mp5.2.c
  ```

  You may hand in as many times as you like, but only the last submission will be time stamped and graded. Note that only the .c file is submitted.

Grading

- Your code functionality will be graded by a computer script. Test your program using our test cases provided in the tests directory of the provided files. You must ensure that your program terminates properly and does not loop forever.
- COMPARE YOUR PROGRAM'S OUTPUT TO THE PROVIDED TEST CASES USING `DIFF` OR `VIMDIFF`.

Functionality (65%)

- 5%: Program terminates correctly.
- 30%: (Checkpoint 1): Program takes an input ASCII text file and creates a Huffman encoded output text file (using ASCII 0s and 1s as the binary strings).
- 30%: (Checkpoint 2) Program takes an input Huffman encoded text file (using ASCII 0s and 1s as the binary strings) and creates the corresponding ASCII encoded text file.

Style (20%)

- 5%: Program compiles without errors or warnings.
- 5%: Follows appropriate instructions such as using `getc` (or `fscanf`) and `fprintf` for interacting with files.
- 10%: Program uses the given data structure, specified functions, and recursion where specified.

Documentation (15%)

- 5%: Introductory paragraph explaining program's purpose and approach.
- 10%: Good comments and variable names.