CS 351 – Advanced Data Structures Practicum

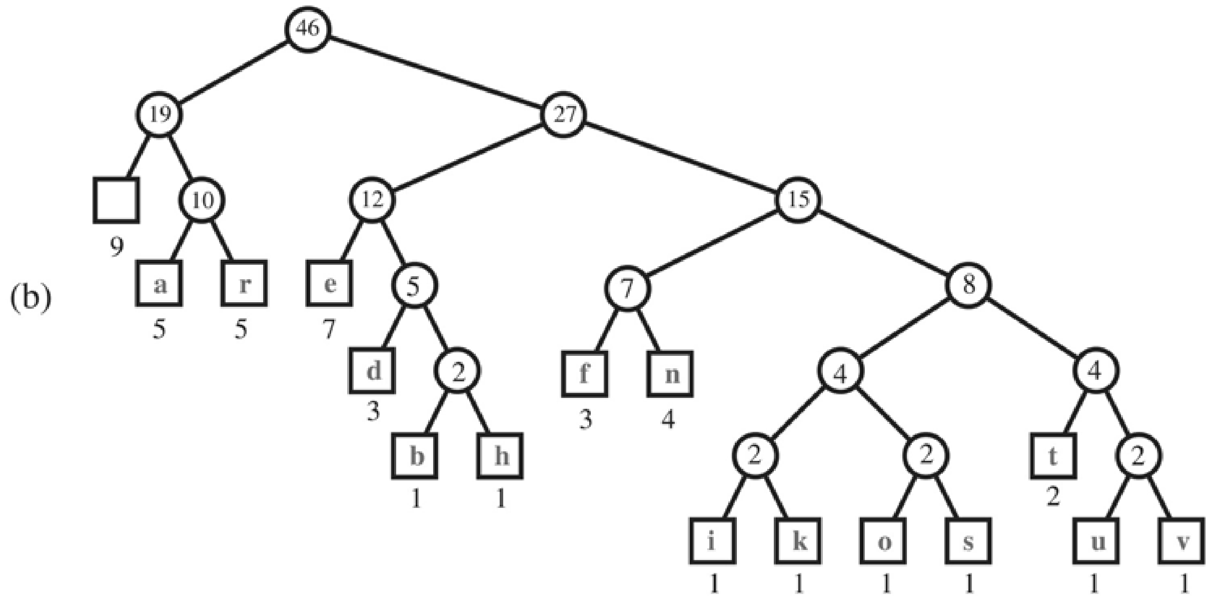
File Compression – Version 1.0

Due: Monday, April 24, 2023 at 11:59 pm

- `24 hours late: Tuesday, April 25th, 2023 @ 11:59pm for 10% deduction`
- `48 hours late: Wednesday, April 26th, 2-23 @ 11:59pm for 30% deduction`

Optional: You may work on this project with one other student currently taking CS 351. If you work on this project with a partner, you are to submit only one submission for the both of you. Gradescope will allow you to select your partner when submitting. It is your responsibility to properly do this.

You are to write a C++ program named "filecompress.cpp" that will perform file compression and decompression using the Hoffman Coding technique. The following image (from figure 10.4.1b in the zyBook for our course) is an encoding tree from the input string of "**a fast runner need never be afraid of the dark**". One should note that the resulting tree may not be unique for a given input string (i.e. there could be multiple trees that could represent the same information).



The tree is built first by finding the frequency that each letter/character occurred:

Space: 9	a: 5	b: 1	d: 3	e: 7	f: 3	h: 1	i: 1
k: 1	n: 4	o: 1	r: 5	s: 1	t: 2	u: 1	v: 1

Note that all the letters are leaves in the resulting tree. The new code for each character is determined by using the position in the tree. The length of the code is the same as the depth of the node containing the letter (the number of edges from the root to the node). If the path follows an edge to the left child, a "0" bit is used in the resulting code. If the path follows an edge to the right child, a "1" bit is used in the resulting code. The resulting binary codes for each letter would become:

Space: 00	a: 010	b: 10110	d: 1010	e: 100	f: 1100	h: 10111	i: 111000
k: 111001	n: 1101	o: 111010	r: 011	s: 111011	t: 11110	u: 111110	v: 111111

The beginning of the compressed version of the message in binary 0's and 1's would be (spaces have been added to make it easier to see the letters but, no spaces would be in actual compressed version):

```
a   sp f    a   s      t     sp r   u       n    n    e   r   sp …
011 00 1100 001 111011 11110 00 011 111110 1101 1101 100 011 00 …
```

The tree is then built using the following algorithm (from Algorithm 10.4.1 from the zyBook for our course). Note that the set of characters and frequency is the input to this algorithm. The algorithm refers to the frequency as the weight of the character: *f(c)*. Also note: this algorithm uses a priority queue:

```
Algorithm Huffman($C$):

  Input: A set, $C$, of $d$ characters, each with a given weight, $f(c)$
  Output: A coding tree, $T$, for $C$, with minimum total path weight
   Initialize a priority queue $Q$.
   for each character $c$ in $C$ do
       Create a single-node binary tree $T$ storing $c$.
       Insert $T$ into $Q$ with key $f(c)$.
   while $Q$.size() $> 1$ do
       $f_1 \leftarrow Q$.minKey()
       $T_1 \leftarrow Q$.removeMin()
       $f_2 \leftarrow Q$.minKey()
       $T_2 \leftarrow Q$.removeMin()
       Create a new binary tree $T$ with left subtree $T_1$ and right subtree $T_2$.
       Insert $T$ into $Q$ with key $f_1 + f_2$.
   return tree $Q$.removeMin()
```

Note that the algorithm starts with as many binary trees as there are letters. Each binary tree has its frequency associated with it and it is this frequency of the tree that is used by the priority queue.

When you decompress the message, you will need access to the binary tree that was used when creating the Huffman Codes. At the start of the message, you will begin traversing down the tree from the root to a leaf, following the path to the left child when a 0 is encountered and following the path to the right child when a 1 is encountered. When a leaf is found, you have decompressed one character from the compressed message. Then you, again, begin traversing down the tree from root again to find the next character.

Consider the above compress message portion (shown without the spaces this time):

    0110011000011110111111000011111101101110110001100

The traversal will start at the root and path down the left child since the first bit is a 0, then down the right child for the second bit of a 1, and again to the right child for the third bit of a 1. At this point, the traverse is at the leaf node for the character "a", so the traverse has decoded the first character which is an "a'. Since a leaf node has been reached, the next bit is used at the root. In this case the next/fourth bit is a 0 so the traverse goes to the left child of the root. The fifth bit is a 0, so again, the traverse goes to the left child. At this point the traverse is at the leaf node for the character 'space', so the traverse has decode the second character which is a space character.

From the above descriptions, hopefully the algorithms used by the Huffman Coding for compression and decompression are clear. Now it is time to discuss the specifics needed for this assignment.

First, you notice that data structures for a binary tree and a priority queue are needed. **For this project, you are required to write the tree code and priority queue code yourself.** You may not use the standard library or pre-written trees or priority queues. Also, the priority queue must be written using an array-style implementation for the internal heap (this heap code may use the standard library vector class).

The program is to have 4 primary operations:

1. Create the needed Huffman Coding information based on the frequencies used in a given file and store this information in a file for future use.
2. Import the Huffman Code information from a file created by part 1 to use for file compression/decompression in operations 3 and 4.
3. Use the imported Huffman Code information to compress a given file.
4. Use the imported Huffman Code information to decompress a given file.

Operation 1: Creating and Storing the Huffman Coding Information

One issue with using Huffman Coding is that while the resulting compression of the file is shown to be optimal, there is additional information that is needed which adds overhead. One solution is to use the Huffman Code information based on the frequencies of one file/document on many files that you are trying to compress. This does not guarantee optimal compression but it amortizes the additional data storage information across all of the files being used and thus reduces the data storage needs for a single file.

Toward this end, the program is to store the Huffman Coding information in a file that will be given the file extension of .hi (for Huffman Information). This file will contain the Huffman Code built by the above algorithm for each of the 128 ASCII characters based on the frequencies found in a specified source file. The program could create multiple of these .hi files using different source files.

The format of the .hi files will be as follows:

- The file will contain 128 lines of information
- Each line of information will contain two pieces of information (as readable strings)
  - these two values will be separated by 4 space characters
- The first value will be the decimal value of the ASCII characters from 0 to 127
  - these ASCII values are to be listed in sorted order from 0 to 127
- The second value will be the string of 0's and 1's to represent the ASCII character by the Huffman Code

It should be noted that the first value on each line could be omitted, but this will add to readability for debugging and understanding. The use of the 4 spaces between the two values on the line is to also add readability. The program could be extended to include the Extended ASCII codes as well; however, the data files being used for this project should only contain the Standard ASCII characters.

This information is what is needed to perform the compression of the files. When a character is read in from the file that is being compressed, the string of 0's and 1's are to be written out (as binary bits) to the output file.

When this operation is run, the user will provide the name of a file that will be used to determine the frequency counts for each of the ASCII characters. The .hi file will store the resulting information in a file that will just add on the .hi file extension to the given filename. If the original file does not exist, print a appropriate error message. If should be expected that not all of the ASCII characters may exist in the original file (the Constitution file does not contain a Capitol Z), so the Huffman Coding algorithm may need to run with frequency values of zero.

Operation 2: Import the Huffman Code Information from a File

Before a file can be compressed or uncompressed, the program will need to know which Huffman Code information will be used. Instead of specifying this at the time of the compression/decompression, the program will have an operation to set up this information before doing the compression/decompression operations. This allows the program to use the same Huffman Code information on multiple files without having to reload/set-up the internal representation multiple times.

When this operation starts, the user will provide the name of a .hi file that. Since this Huffman Information file (which was created in Operation 1) only contains the code needed for the compression to occur. In order to do the decompression of a file, the binary tree structure will need to be build. So as a part of this operation, two things need to occur:

1. The Huffman Code information for each character is to be stored in a vector.
2. The Huffman Tree will need to be built.

The first one should be easy since that is the information contained in the .hi file.

The second one will take a bit more work, but we know that every character will be a leaf node in the tree and the Huffman Code uniquely identifies the path from the root of the tree to that leaf node. (Note that recreation of the tree could follow a similar algorithm to inserting a word into trie; however, there are only 2 possible "letters" instead of 26. So, when a leaf for a character needs to be added to the tree, the traversal starts at the root and follows the path laid out by the Huffman Code (left on 0, right on 1). If the traversal encounters a null pointer during the traversal, a new tree node is created and inserted in the proper place as the tree is being built.

Operation 3: Compressing a File

When we compress a file, we first need to verify that Operation 2 has been done. If not, given an error message and inform the user to Import a .hi file via operation 2.

The general idea of compressing a file is very simple:

1. Read in the file character by character
2. For each character, write out the bit string for the character's Huffman Code.

The first thing that we need to do is to determine the name of the output file. The output file will add on a file extension of .hc to the filename given by the user for the input file. The extension of .hc stands for "Huffman Compressed".

When you are writing out the bit string for the character's Huffman Code, recall that this information is written out as characters of 0's and 1's in the .hi file. Here you will need to write it out as BITS of 0's and 1's.

The tricky part of this is that the C++ language only allows the smallest output for the user to write out is 1 byte or 8 bits. Since our bit strings for the Huffman Codes will be a variable length, the program will need to buffer the bits into groups of 8 and then output the byte contain those 8 bits in a single statement.

Another item for concern is that when the last character in the original file is processed, odds are that the last bit(s) of the character's Huffman Code are still in the "bit buffer". The bit buffer will need to be flushed. So you will need to write some more bits until you have a multiple of 8 bits and the "bit buffer" is written to the output. However, you do need to be careful of which bits you are writing to the "bit buffer" for these extra bits. If these extra bits match some character's Huffman Code, the decompression operation will add an additional character to the output. Thus you will need to find a character whose Huffman Code has length greater than 8 bits and use the bits from that Huffman Code to fill up the "bit buffer" at the end.

One should also note that the C++ type of "unsigned char" should be used for this. Use of the normal C++ "char" type will not work as this always writes out a 0 bit for the most significant bit of the 8 bits.

Once the file has been compressed, your program is to calculate the following information and output this to standard output:

1. The number of characters/bytes in the original file
2. The number of bytes in the compressed file
3. The compression ratio. Print this out to 5 decimal places
4. The space saving as a percentage. Again print this out to 5 decimal places

The compress ratio is the result of doing a floating point division of the number of byes in the original/uncompressed file by the number of bytes in the compressed file.

The space saving is (1.0 minus (the floating point division of the number of bytes in the compressed file by the number of bytes in the original/uncompressed file)) multiplied by 100.0.

Operation 4: Decompressing a File

When we decompress a file, we again need to verify that Operation 2 has been done. If not, given an error message and inform the user to Import a .hi file via operation 2.

The general idea of decompressing a file was given toward the beginning of the write-up.

1. Read-in the file bit-by-bit and traverse down the Huffman Tree one level for each bit.
2. If the bit is a 0, traverse to the left child. If the bit is a 1 traverse to the right child.
3. When a leaf node is encountered, output the ASCII character associated with the leaf node and begin the next traversal from the top of the tree.

The first thing that we need to do is to determine the name of the output file. The file being used is expected to have the file extension of .hc . (The extension of .hc stands for "Huffman Compressed"). The program will remove the file extension of .hc to possibly recreate the filename given by the user for the original/uncompressed file. This should result in the name of the original file. It is left up to the user to make sure they do not overwrite the original file in their disk space.

As we encountered with Compressing a file, C++ only allows the smallest input to be 1 byte or 8 bits. Thus the program will need to read-in a byte and then process those 8 bits and then read in another byte to continue processing.

It will most likely occur that you will be in the middle of the tree when the end of the file is reached. This is to be expected since the compression operation most likely needed to pad the output with bits since the total number of bits needed for the compressed character was not an exact multiple of 8.

Programming Style

Your program must be written in good programming style. This includes (but is not limited to) meaningful identifier names, a file header at the beginning of each source code file, a function header at the beginning of the function, proper use of blank lines and indentation to aide in the reading of your code, explanatory "value-added" in-line comments, etc.

The code submitted by you is to be the work created by the members of your group (or just by yourself if you are not working with another student).

There is some starter code and data files available in replit.com Teams for CS 351.

- https://replit.com/@CS351Spring2023/Sliding-Block-Puzzle