CS106B
Winter 2020

Handout 28
March 6, 2020

Assignment 8: Huffman Coding

This assignment is a modified version of one developed by Owen Astrachan of Duke University, with edits by a number of talented folks including Julie Zelenski, Cynthia Lee, and Marty Stepp.

Your final assignment of the quarter is to write a program that uses Huffman coding to compress and decompress files. It pulls together ideas from all over the quarter – recursive exploration, linked structures, trees, and streaming algorithms. Once you've finished coding this one up, you'll have a fairly impressive piece of software that's powered by a bunch of clever algorithms. We hope this is a fitting capstone to your experience in CS106B!

Due Friday, March 13th at the start of lecture.
You are welcome to work on this assignment in pairs.

☞ ***No late days may be used*** ☜
and
☞ ***no late submissions will be accepted.*** ☜

Because no late days may be used on this assignment, it's important to start early and to make slow and steady progress. Here's our recommended timetable.

- Aim to complete Milestone One within one day of this assignment going out.
- Aim to complete Milestone Two within three days of this assignment going out.
- Aim to complete Milestone Three within five days of this assignment going out.
- Aim to complete Milestone Four within six days of this assignment going out, leaving a buffer day to go back and fix any issues that surfaced in this step.
- Bask in the glory of having completed the last assignment of the quarter, and explore Milestone Five to appreciate the program you've just written.

We recommend that you periodically submit your work to Paperless just in case something comes up and you can't submit by the deadline. We'll only grade the last version you submit.

Milestone One: Build the Huffman Tree

In this first milestone, you'll implement this function:

```
HuffmanNode* huffmanTreeFor(const string& str);
```

This function takes as input a piece of text, then builds a Huffman coding tree for that text using the algorithm from class. (It's the one that makes a priority queue of a bunch of singleton nodes, then repeatedly combines nodes together.)

Before you code this up, take a minute to make sure you understand how Huffman coding works.

Edit the file `res/ShortAnswers.txt` with your answer to the following question:

Q1. Draw the Huffman coding tree that would be produced for the input string `"aabbbbccc"` by following the algorithm from class. One specific note: when merging trees together, structure your tree so that the first node removed from the priority queue becomes the 0 child of the tree and the second node removed from the priority queue becomes the 1 child of the tree.

Now, back to C++. The result of `huffmanTreeFor` is a `HuffmanNode*` object, where `HuffmanNode` is defined as follows:

```
struct HuffmanNode {
    char ch;           // Which character is stored here.
    HuffmanNode* zero; // Child tree labeled 0.
    HuffmanNode* one;  // Child tree labeled 1.
};
```

The `ch` variable here is only meaningful if the `HuffmanNode` is a leaf node in the tree (do you see why?), so you should not read or write this variable in a `HuffmanNode` unless that node is a leaf.

In the course of coding up this part of the assignment, you'll need to use a priority queue. You built one of these in Assignment 5, but that `HeapPQueue` type was specifically designed to work with `DataPoint` objects. We've provided the `"priorityqueue.h"` header file, which defines a `PriorityQueue<T>` template that can store objects of any type. Consult the "Stanford C++ Library Documentation" on the course website for more information about how to use this type. In particular, make sure you know how to enqueue and dequeue elements and how to determine the priorities of relevant items.

One little nuance that you'll need to be aware of: the test cases in the starter files we've provided all assume that, when you're running the Huffman tree construction algorithm, the first node you pull out of the queue will end up as the 0 child of the newly-constructed node, and the second node you pull out of the queue will end up as the 1 child. Make sure to follow this convention in this assignment.

To summarize, here's what you need to do:

1. Implement the `huffmanTreeFor` function in `Huffman.cpp`.
2. Add at least one custom test case for this function, and test your code thoroughly.

Our test coverage here is not as robust as in the previous assignments. You will need to test your code thoroughly. A recommendation: write a test case that creates a Huffman tree from a string of your choosing, then set a breakpoint and run the tests with the debugger on. Using the same techniques from Assignment 7 that you used to map out and escape the labyrinth, poke around in the debugger and draw out the trees that you're producing. Do they match what you expected to find? If so, great! If not, step through your construction algorithm in the debugger and see what happens.

Some notes on this problem:

- The tree construction algorithm we provided in class assumes that the input string has at least two different characters in it, and if this isn't the case, your code should call the `error()` function to report an error. (Do you see why the algorithm doesn't work in these cases?)
- If multiple characters are tied for having the same weight, or if two or more intermediate trees are tied for having the same weight, you can break ties arbitrarily. Keep in mind that there can be many equally good Huffman trees for a given string, differing only in how the algorithm broke ties. As a result, if you're writing custom test cases, make sure that your tests don't assume anything about the specific way that the algorithm breaks ties when they arise.
- You should completely ignore the `ch` field of `HuffmanNode` for all nodes except leaves, the same way that you completely ignored the `value` field of the `LinearProbingHashTable::Slot` type when the `isEmpty` field was set.
- You should be able to handle strings made of any characters, not just letters, numbers, etc. In particular, you can't reserve certain `char` values to mean "this is an internal node." Many files on disk – especially files containing non-text data – make use of all possible characters.

Milestone Two: Implement Text Encoding and Decoding

Your next task is to write a pair of functions that are inverses of one another:

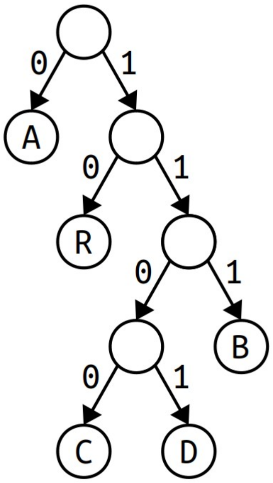
```
     string decodeText(Queue<Bit>& bits, HuffmanNode* tree);
Queue<Bit> encodeText(const string& str, HuffmanNode* tree);
```

This `decodeText` function takes as input a `Queue<Bit>` representing some data that's been compressed with Huffman coding, along with a pointer to the encoding tree that was used, then returns the decompressed string. The `encodeText` function takes as input a string and a Huffman coding tree, then returns a `Queue<Bit>` loaded with the bits formed by running the Huffman coding algorithm on that string.

Before you code this up, though, take a minute to make sure you know how to run the algorithms with a pencil and paper.

Edit the file `res/ShortAnswers.txt` with your answer to the following questions.

Q2. Decode the string of bits `1100010011000100` using the Huffman tree shown below:



Q3. Encode the string `ABRACADABRA` using the Huffman tree from Q2.

Returning to C++ Land, you'll notice that these functions make reference to a `Bit` type, which represents a single bit. The `Bit` type works like a regular integer, though note that if you try assigning a value to a bit that isn't a 0 or a 1 it'll report an error. For example:

```
Bit b = 0;
queue.enqueue(0);
if (queue.dequeue() == 1) { … }
```

You can't perform arithmetic on `Bit`s. But that's a good thing, because there's no reason you should need to do that in this assignment. 😃

To summarize, here's what you need to do:

1. Add a custom test case for the `decodeText` function in `Huffman.cpp`.
2. Implement the `decodeText` function and test thoroughly.
3. Add a custom test case for the `encodeText` function.
4. Implement `encodeText` and test thoroughly.

Some notes on this problem:

- You're given the input `Queue` by reference in `decodeText`, and your code can modify that `Queue` however it feels like. The queue can end up empty, or could contain the original bits, or could contain whatever sequence of 0s and 1s that you'd like.
- In our initial examples of encoding and decoding text in lecture, we had access to an explicit table that represented the bit patterns associated with each character. It is *significantly* easier to implement the `encodeText` function if you build a table like this before writing out the characters. However, having a table like that isn't necessary for the `decodeText` function – do you see why?
- You can assume that the input tree is not null and doesn't consist of just a single node. (These are the two degenerate cases that we said you didn't need to handle in `huffmanTreeFor`.)
- The tree given to `encodeText` and `decodeText` will always contain leaves holding all the characters that appear in the input string or input queue of bits (respectively), but may also contain other characters. The input tree does not necessarily have to be the optimal Huffman coding tree.
- You can assume that the bits provided to `decodeText` form the correct encoding of a piece of text that was encoded using the same encoding tree that was provided to you. However, you should be careful about how you use this assumption. Specifically, if there's a bug in your `encodeText`, then you may end up passing invalid bits into `decodeText` – oops! – and so we recommend making your `decodeText` function as "bulletproof" as possible by detecting and handling errors as you find them.
- In C++ there is a difference between the numbers `0` and `1` and the characters `'0'` and `'1'`. Be careful not to write something like

  ```
  Bit zero = '0'; // Error!
  Bit one  = '1'; // Error!
  ```

  since this attempts to take the ASCII codes for `'0'` and `'1'` (namely, 48 and 49) as values for bits. Instead, use the numbers themselves:

  ```
  Bit zero = 0; // Beautiful!
  Bit one  = 1; // Pulchritudinous!
  ```

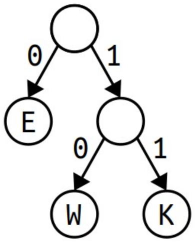
Milestone Three: Implement Tree Encoding and Decoding

One of the practical concerns of Huffman coding that we discussed in class is that if you do choose to use Huffman coding, you need to somehow store information about which encoding tree you used. That way, whoever is decoding the information knows which codes corresponding to which characters. This means that we'll need to devise a pair of functions that allow you to take a tree and encode it in a way that can be stored on disk, plus a way to decode the encoded representation back into the tree.

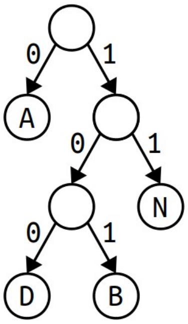
There are many ways to do this, but one of the most space-efficient ways is the following. We'll encode each tree as a pair of a `Queue<Bit>` representing the shape of the tree and a `Queue<char>` representing which characters are in the leaves of the tree. The shape of the tree is encoded as follows:

- If the tree is a single leaf node, it's represented by the bit 0.
- If the tree is not a leaf node, it's represented by a 1 bit, followed by the encoding of its zero (left) subtree, followed by the encoding of its one (right) subtree.

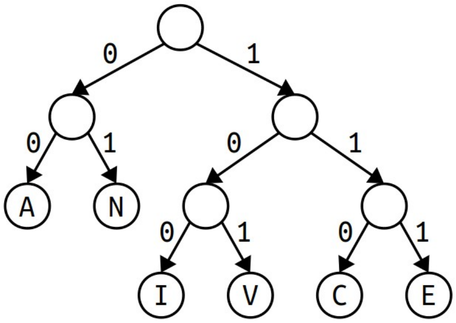
The `Queue<char>` representing which characters are stored in the leaves of the tree is found by reading the leaves in the pattern you'd get by doing an inorder traversal of the tree. For example, here are several Huffman trees and how they'd be represented as a sequence of bits and a sequence of characters:



```
10100
EWK
```



```
1011000
ADBN
```

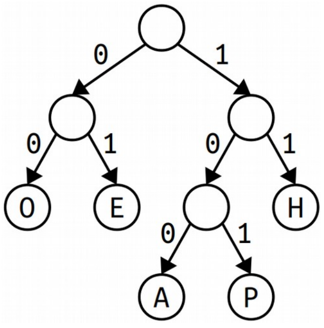


```
11001100100
ANIVCE
```

This is an extremely compact way of writing out a tree: the tree itself is written out with one bit per node, and the contents of the leaves are written out as-is. Most importantly, it's possible to reverse the encoding back into a tree. Before writing code to compute or process these sequences, grab a pencil and paper and make sure you're comfortable with how these work.

Edit the file `res/ShortAnswers.txt` with your answers to the following questions.

Q4. Write out the `Queue<Bit>` and `Queue<char>` associated with this Huffman tree:



Q5. Decode this `Queue<Bit>` and `Queue<char>` into a Huffman tree and draw the result:

```
111000100
MDCLV
```

Your task in this part of the assignment is to write a pair of functions

```
void encodeTree(HuffmanNode* tree, Queue<Bit>& bits, Queue<char>& leaves);
     HuffmanNode* decodeTree(Queue<Bit>& bits, Queue<char>& leaves);
```

This first function takes a tree and produces the two `Queue`s shown above. The second function takes in the two `Queue`s and reconstructs the tree they represent.

Specifically, you should do the following:

1. Implement the `encodeTree` and `decodeTree` functions in `Huffman.cpp`.
2. Add at least two custom test cases, one for each function, and test your code thoroughly.

Some notes on this problem:

- There are no requirements about what the `Queue<Bit>` or `Queue<char>` should contain after `decodeTree` finishes running. They could be unchanged, or completely empty, or filled with whatever contents you'd like.
- You can assume the inputs to `decodeTree` are correct, in that the bits and characters are a correct representation of some Huffman tree, that there aren't stray bits in the queue, that you have the exact correct number of characters that you need, etc. However, as with `encodeText` and `decodeText`, a bug in your implementation of `encodeTree` may manifest as errors in `decodeTree`, so we recommend making your `decodeTree` function "robust" and able to identify and flag errors if it finds them.
- You can assume the tree provided as input to `encodeTree` is a valid coding tree – each internal node will have exactly two children, only the characters in the leaves matter, each character appears in at most one leaf, etc.
- You can assume the input `Queue<Bit>` and `Queue<char>` to `encodeTree` are empty when that function is first called.
- You don't need to worry about the edge cases that came up in the first milestone. You can assume the tree exists and contains at least two leaves.

Milestone Four: Put It All Together

You now have all the pieces you need to build the final Huffman encoder and decoder. Implement the following pair of functions:

```
HuffmanResult compress(const string& text);
  string decompress(HuffmanResult& file);
```

This first function takes as input a string of text and produces a `HuffmanResult` containing all the information necessary to store the compressed contents of that string to disk. The second function takes as input a compressed `HuffmanResult` and returns the string that it encodes. Here, the `HuffmanResult` type represents a combination of the encoded tree and the encoded text:

```
struct HuffmanResult {
    /* Encoded version of the Huffman tree. */
    Queue<Bit> treeBits;
    Queue<char> treeLeaves;

    /* Encoded version of the message. */
    Queue<Bit> messageBits;
};
```

This section should mostly consist of putting together the different pieces you wrote earlier in the right order. In doing so, you may discover that there were some bugs lurking in your implementation, which you'll need to then correct to get everything working. (It's common in software engineering to find that each individual piece of a program passes its tests independently but fails when things come together; it's usually either due to tests not covering every possible case or to some unexpected interactions between the components). When that happens, use the debugger to isolate where the issue is. Did you build the encoding tree incorrectly? Did you encode the message bits wrong, or is the issue in the decoder?

To summarize, here's what you need to do:

1. Implement the `compress` and `decompress` functions in `Huffman.cpp`.
2. Add at least one custom test case to ensure that your code works. Take this step seriously – writing good tests here will help you identify and smoke out errors that might be lurking in your code.

Testing is key here! We have not provided enough tests with the starter files to poke and prod these functions in all the ways they can be poked and prodded, and you should not consider your implementation to be ready until you've written some tests of your own and used the interactive "Compress a File" and "Decompress a File" options to confirm that your code works end-to-end.

Some notes on this part of the assignment:

- Your implementation should call `error()` if the input string to compress doesn't have at least two distinct characters in it.
- You can assume that the input to `decompress()` is a valid compressed file and don't need to worry about what happens if this isn't the case. Then again, making this function as robust as possible is a *great* idea during testing.
- Our provided starter files contain logic to take a `HuffmanResult` and save it to disk and to take a `HuffmanResult` and load it from disk. The internal logic of how this works is more of a CS107-level topic that involves manipulating individual bits of variables, which is beyond the scope of what we're going to cover in CS106B. However, if you're curious to see how this works, you're welcome to check out `Demos/HuffmanResult.cpp` to learn more!
- Make sure not to leak any memory in your implementations.
- There are no requirements about what the final contents of the `HuffmanResult` may be after calling `decompress`. You can leave it unchanged, empty it out, etc.

Milestone Five: Enjoy Your Creation!

Congratulations! You've just built a compression and decompression algorithm! All that's left to do now is to try it out on a few examples to see how well it does.

We've bundled a number of sample files with the assignment, some already compressed (they have the suffix `.huff`) and some not yet compressed. Use your program to decompress the compressed files to see what they contain, and then try compressing some of the sample files so that you can see how well those files compress!

Something specifically that we'd recommend: try decompressing `Congratulations.bmp.huff`. That file will expand out to one named `Congratulations.bmp`, which you should definitely take a look at once you've finished the assignment. 😃

> There are no deliverables for this section. Just play around and see what you find!

(Optional) Milestone Six: Extensions!

If you enjoyed this assignment and want to run further with it, we'd love to see what you come up with! Here are a few suggestions to help you get started.

- ***Implement a more advanced compression algorithm.*** Huffman coding is a good compression algorithm, but there are better alternatives in many cases. Many modern compression algorithms are based on an algorithm called *LZW*, named after the initials of its inventors. Other topics to look up include *move-to-front coding*, *run-length encoding*, and the *Burrows-Wheeler transform*.
- ***Explore some information theory.*** The field of *information theory* explores, among other things, the limits of data compression, and gives ways of measuring how good particular compression algorithms are. Look up *Shannon entropy*, *KL divergence*, and other similar concepts and see if you can use them to tell us anything interesting about data compression.

Submission Instructions

Once you've autoindented your code so that it looks beautiful and worked through the Assignment Submission Checklist, submit the following files on Paperless, plus any other files you modified when writing up extensions:

- `res/ShortAnswers.txt`. ***(Don't forget this one, even though there's no code in it!)***
- `Huffman.cpp`.

And that's it! You're done! You know how to work with trees, linked structures, container types, and recursive problem-solving. Congratulations!

Good luck!