Project 3b — Huffman Decoding

Please read the entire document carefully! I would suggest reading the entire document *before* writing any code.

1 Introduction

For this project, you will implement a program to decode text that has been encoded using Huffman coding. This project assume that the encoded file is in the format that was output from the previous part of the assignment.

For decoding, you will need to recreate the Huffman Tree that was used to determine the Huffman codes used for encoding. The header in the input file contains the character frequency information that will be needed. After reading in the frequency information, you will be able to use the same `build_huffman_tree` function that you wrote for the encoding portion of the assignment.

Once the Huffman tree is recreated, you rill be able to traverse the tree using the encoded `0`s and `1`s from the input file. The decoding process starts by beginning at the root node of the Huffman tree. A `0` will direct the navigation to the left, while a `1` will direct the navigation to the right. When a leaf node is reached, the character stored in that node is the decoded character that is added to the decoded output. Navigation of the Huffman tree is then restarted at the root node, until all of the characters from the original file have been written.

2 Specification

In the same `huffman.py` file from the first part of the assignment, you will add the following:

2.1 Parse Header

You will implement a function call `parse_header` that takes as an argument a string (the first line of the file) and returns a list of frequencies. The list of frequencies should be in the same format that `count_frequencies` returned in the first part of the assignment, i.e., a list with 256 entries, indexed by the ASCII value of the characters.

2.2 Huffman Decoding

You will implement a function `huffman_decode` that takes as arguments two file names (for the input and output), decodes the input file, and stores the resulting decoded data in the output file.

This will be done by first building the Huffman tree (using functions you've already written), and now recreating the original file one character at a time using the tree traversing method described above.

2.3 More Notes

You may find it useful to read a single line of text from a file. In Python, you can read just one line with `file.readline()` which will return one line in the file.

For our purposes, recall that the first line will be the header and the second line will be all the `0`s and `1`s.

3 Testing

When testing, always consider *edge cases* like the following cases:

- If the input file consists only of some number of a single character, say `"aaaaa"`, what do you think the result should look like? What is the "code" for the letter `'a'`?
- If the input file is empty (as in 0 bytes large), what should the result look like?

Now when we decode those files, how do we handle these cases? These will possibly require special cases in your decoding code.

In your `huffman_tests.py`, you should leave all the tests you had for the previous part (they're needed for 100% coverage) and additionally add tests for the new functions that you're writing.

There are sample tests provided on Canvas (`huffman_decode_tests.py`) that you can/should copy over into your `huffman_tests.py` file.

Like the previous assignments, you are still *required* to achieve 100% test coverage. This means that every line of code in your functions *must* be executed at some point in at least one of your tests.

As discussed in the syllabus, when I grade your code, you will ordinarily receive feedback regarding any tests that fail, *unless* you do not have 100% test coverage. In the event you do not have 100% test coverage, the only feedback you will receive is that you need to do more testing. I don't want you using my grading script to do your testing in the last day.

4 GitHub Submission

Push your finished code back to GitHub. Refer to Lab 0, as needed, to remember how to push your local code.

The files you need to have submitted are:

- `ordered_list.py`
  - Your correct implementation of the `OrderedList` from Lab 4.
- `huffman.py`
  - Contains the functions specified above, and any helper functions that you find necessary.
- `huffman_tests.py`
  - Contains tests for all your functions specified by the assignment. These should run on *any* correct implementation of this project. As such, you should not include any tests for helper functions.
- `huffman_helper_tests.py`
  - Optional file containing tests for all your helper functions. These tests will be used in conjunction with your `huffman_tests.py` to check for 100% coverage.
- Any text files you use for testing.
  - If you use a text file for testing, you'll need to submit it so that your tests run on my end.