Be sure to read the complete spec, prior to starting to code. Do not miss the Tips and Testing section at the end, for useful directions and advice.

A reminder that students who share or copy code or solicit or view solutions online are subject to academic and disciplinary penalties.

As always, all written work is expected to be individual. If you discussed your code with anyone, please list them here (excluding teaching staff):

_____

_____

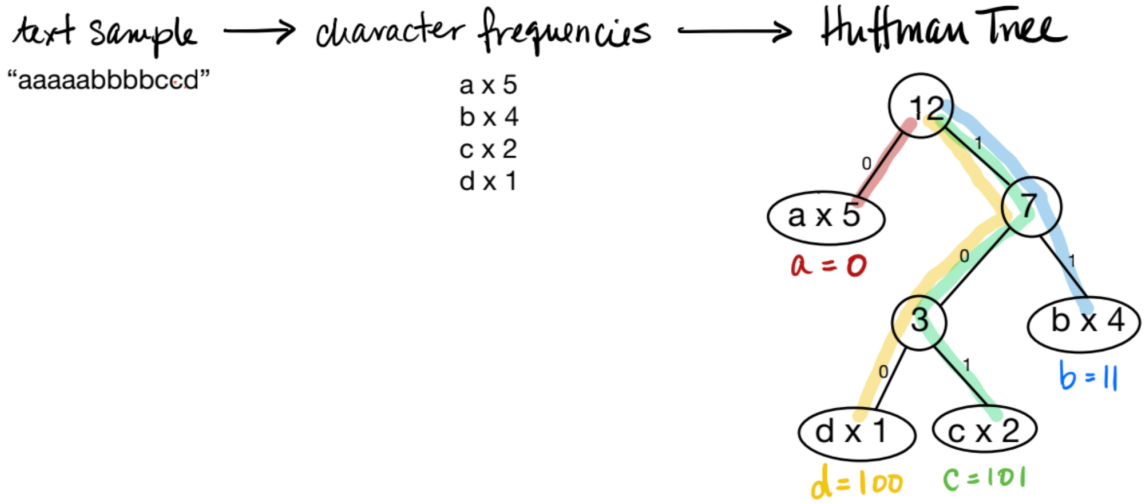
_____

_____

Overview

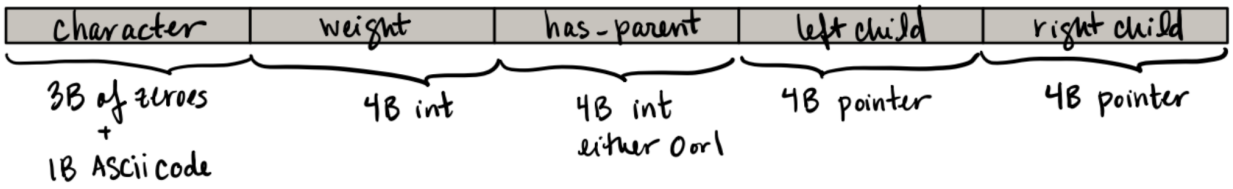
In this assignment, you will build a Huffman decoder. Rather than one ASCII code byte per character, for a total size of 8 bits per character, a Huffman code is a variable length encoding, meaning some characters will be encoded using more bits, others fewer. Huffman codes are designed to compress text by using fewer bits to encode the more common characters and more bits for the uncommon characters. This 6m30s video gives a nice illustrated tutorial on the technique.

Huffman codes are represented as binary trees, where each leaf node contains a character and the individual weight (aka frequency) of the character. Each internal node in the tree is weighted as the sum of its two children. From the root of the tree, descending down the left child corresponds to a 0 bit in the code and descending down the right child corresponds to a 1 in the code. The path from root to leaf determines the Huffman code for each character, as illustrated below:



Given a null terminated string of ASCII characters, your code will analyze the relative frequency of each character and build the corresponding Huffman tree. You will then use that tree to decompress and read a message:

The tree will reside in memory as an array of individual nodes. Each node of the tree has five fields, each field is one word, for a total of 20B:



Programming Details

The assignment is broken down into seven functions that work together to build a tree. We recommend completing them in the order they are listed here which is (roughly) easiest to hardest. Moreover, the later functions invoke the earlier ones, so you will want to have working implementations of the first ones before moving on. Be sure to adhere to MIPS calling conventions to isolate the implementation of each function.

count_char

Our first procedure, called count_char, takes two arguments: $a0 and $a1, which hold a character and a pointer to a null-terminated string in memory, respectively The function should return, in $v0, the number of instances of the character in the string.

minmax_chars

This function, called minmax_chars, takes a pointer to a null terminated string, and returns two values: in $v0, the smallest character in the string by ASCII code value, and in $v1, the largest character in the string by ASCII code value. You can assume this function is always called on a non-zero-length string.

make_leaf

This function initializes a leaf node for the Huffman tree. It has no return value and takes three arguments:

- $a0: the character associated with the leaf node
- $a1: the character weight (i.e., frequency)
- $a2: a pointer to memory where the new leaf should go

As a leaf node, both child pointers should be null. And, at creation this node is not connected to a tree, so the has_parent field should be initialized to zero.

merge_roots

This function initializes a non-leaf node with two children. It takes three arguments:

- $a0: a pointer to the left child node
- $a1: a pointer to the right child node
- $a2: a pointer to memory where the new node should go

The new parent node should point to the two children, and its weight should be the sum of the two children's weights. The new parent node has no associated character, so the character field should be zeroed out, as should the has_parent field. However, the two children now have a parent, so their has_parent fields should be updated to be 1.

count_roots

This function should walk an array of nodes and return the number of roots (i.e., nodes without parents). The function takes two arguments:

- $a0: a pointer to the start of the array
- $a1: a pointer to the end of the array (where the last node ends)

lightest_roots

This function is given an array of nodes, via the same arguments as count_roots:

- $a0: a pointer to the start of the array
- $a1: a pointer to the end of the array (where the last node ends)

It should return two pointers:

- $v0: a pointer to the lightest root node in the array
- $v1: a pointer to the second lightest root node in the array

This function will be called only when there are two or more roots in the array. Furthermore, it will be called only on arrays where there is no tie (e.g, the two lightest nodes having the same weight). You thus do not need to worry about these scenarios.

build_tree

This last function constructs the Huffman tree. It takes two arguments:

- $a0: a pointer to a string with the text sample on which to calculate character frequency
- $a1: a pointer to a location in memory with sufficient space for the tree

It should return a pointer to the root of the tree.

To build the tree

1. find the min and max characters in the text sample
2. for each character between min and max, inclusive, add leaf nodes to the array
3. while there is more than one root in the tree
    a. find the two smallest roots in the tree
    b. merge them with the lightest root as the left child and the second lightest as the right
4. when there is only one root in the tree, return a pointer to it

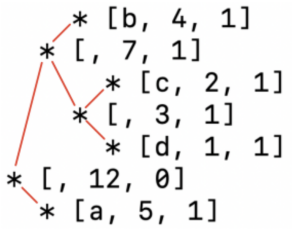
Testing, Tips, and Additional Instructions

We have supplied scaffolding.s, where you should put all of your code.

You will see separator lines in this file. Do not delete these lines and put all of your code for each function after the function name label and before the next separator. Our testing infrastructure relies on these to splice out individual functions from your submission.

In the scaffolding, we have provided a number of test functions, all invoked from main. You can comment/uncomment these invocations according to the tests you wish to run. After these tests, main will build two trees and use them to decompress two strings.

The scaffolding also includes a number of printing functions if you wish to use them (print_char, print_int, etc.) Given a pointer to a node, print_tree prints the tree below that node. Here is how this printer prints the example tree from the start of this spec. The red annotations explain how to interpret the output.



The lightest_roots and build_tree functions are the most challenging, so keep this in mind when budgeting your time.

Be sure to adhere to calling conventions to isolate the implementation of each function. For example, your implementation of build_tree should work just fine if we substitute our implementations count_char, minmax_chars, and so on. Conversely, our implementation of build_tree should work fine if we called your implementations of those helpers.

Comment your code, particularly when the behavior is not self-evident. You will have an easier time debugging if your code has clear notes about what is supposed to be happening.

Please upload your code as a single file, named with your uni (e.g., mak2191.s) on gradescope.