Assignment 7 First Design

James Contini

2022 November 26

Description

This is the preliminary design for assignment seven. Here is a synopsis of the algorithm.

Synopsis

First, a data stream will be parsed into unsigned one-byte numbers from standard input using our syscall functions. Then in a histogram of 256 indices, each corresponding to a different ASCII character, symbol frequencies of the stream will be tracked.

Symbols with a frequency greater than one will be converted into nodes and joined to a priority queue. In this implementation low frequencies have higher priority. Then dequeue two nodes, each time automatically fixing the queue. These nodes will be joined under an *internal* parent. The frequency of the parent is the sum of the children. This node is then also enqueued.

We keep repeating this until there is one node left in the priority queue. Essentially, dequeue two nodes, join them (which sets the left and right field), and then enqueue their fabricated parent until there is one node left in the queue, which will be a parent with two children and also be the root.

Next, we must build a table which will be a two-hundred-fifty-six-long array of one-byte-long bit vector elements. Each element corresponds to an ASCII character. Now to populate this array, post-order traverse the tree. Each recursive level will push a bit to the general stack and in the base case copy this bit vector stack to the correct index, and as the program leaves the depth each level pops a bit because that's the next line to execute. Now this table of bit vectors will be populated with traverse orders.

Next, the tree must be dumped. To dump the tree it must be traversed in post-order recursively printing 'L' and the node's symbol when the node is a leaf and 'I' otherwise. Then using this dump and the binary traverse order table push each symbol's order to a stack.

To decode, we iterate over the tree dump pushing leaf nodes to a stack until we encounter an 'I' internal signifier. At which point we pop two nodes off of the stack and join them setting all three frequencies to zero (for now) and pushing this joined node to the stack. We continue to pop two nodes off and push the parent to the stack until there ends up being one node in the stack. Thus the tree is recreated and now we will traverse the tree in the order of the binary writing out the symbol.

This will produce decoded text.

1 node.c

The node constructor initializes the symbol and frequency field of a node object. Two nodes can be joined by creating another parent node of symbol *'$'* and a frequency which is the sum of the children's frequencies. The parent will point to both children. Two nodes' frequencies can also be compared with a function.

2 pq.c

For the priority queue, there are two functions to swap two pointer values, one for unsigned bytes and one for unsigned longs. There is a function to check if it's full or empty by checking how many elements are greater than zero. The priority constructor initializes two arrays as fields, for the frequencies and the symbols of the nodes. The structure of the queue is a sorted list that is fixed by insertion.

enqueue

From this point on, assume that the insertion procedures, not comparisons, are repeated for the symbol array as well as the frequency array. If the array is empty the frequency is inserted into index zero. If the array is not empty the frequency is inserted into the first element that equals zero. Then while the new element is less than the element before it, swap them. The list is now sorted.

dequeue

To dequeue an element, take the zeroth element from both arrays and make a node with them. Then set the first element to zero. Then the entire queue is pushed back one to fill the zeroth gap.

3 code.c

Code is initialized by allocating 32 unsigned bytes off of the stack. And zeroing them out. A bit can be pushed onto this stack which is literally on *the stack* by setting the bit at the offset to whatever the bit is, then incrementing the offset by one. To get the offset field code_size is called and then a mask is made for the byte number code_size() floor divided by eight and the bit number is the code_size() modulus eight. This is how all bit fiddling is done. Division supplies the byte and modulus supplies the bit index.

4 io.c

This function uses low-level syscalls to read and write bytes. Since the syscalls don't always read or write exactly how many we want we loop the call writing a buffer until the buffer doesn't have anything in it. The same thing goes for write, looping write() until the buffer is empty. For read_bit() we will use the previously explained *read_bytes()* to read the bytes to somewhere and then using a mask that left shifts its by the *static index variable* every call and then put the discovered to bit to be returned by reference. The static index variable will be incremented every read so that the next bit can be read next call. When the buffer in *write_code()* is overwritten each time it is filled, on the last time there may be leftover bytes from the last time buffer was used and these should be zeroed out by *flush_codes()*. To do this the program will use the static index to find where the old code begins.

5 stack.c

This is like the code stack except it's a stack for nodes. It is implemented like a linked list and the nodes left and right fields are used as previous and next. Once nodes are popped off of the stack they are repurposed to be a part of the Huffman tree.

6 huffman.c

build_tree() constructs a Huffman tree out of a histogram which is turned into a priority queue. This is done by joining the two next nodes in the queue. *build_codes()* makes a code table out of each symbol in the tree each code corresponds to its binary traversal bit vector. *dump_tree()* dumps the nodes via post-order binary traversal. *rebuild_tree* rebuilds a tree from its dump.