Assignment 7 Final Design

James Contini

2022 November 26

Description

This is the final design document 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. In the node creator function, fields left and right will remain uninitialized.

2 pq.c

For the priority queue, there are three functions to swap two pointer values, one for unsigned bytes, one for unsigned longs, and one for Node pointers. 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 four arrays as fields, for the frequencies, symbols, and children 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 swaps and insertions but not comparisons are repeated for all four arrays despite the fact only the frequency array is ever talked about here. 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 setting its child fields too. 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. A bit can be popped off of this stack too with the same logic.

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.

write_code()

This function has one case for exiting early; if there is no more code in *Code*. Essentially, the function gets bits from the first index of the code and appends them to the buffer. If the *get_code* function fails *AND* the bit offset is lesser than 32768 (4096 bytes) the function just ends and won't pass the second test. If at one point as the function is removing bits from the code the bit offset surpasses 32768, the while loop ends and checks to see if the offset truly did hit the number. If it did, it writes the buffer out to the outfile and then continues to append bits to the freshly written buffer. When there are no more codes in *code* the function terminates.

flush_codes()

flush_codes is very simple, it just checks to see if the bit offset modulus eight equals zero, if that case is true the code was already byte aligned. If this case is not true the rest of the bits in the function have to be zeroed and then in both cases, the buffer is written to the outfile. It took me a while to understand this but the reason bits are zeroed in the same direction they are appended is that I didn't realize the decode function was going to read each byte from right to left from the start of that byte. This means that as bits get zeroed in the left direction, that's actually the direction of the MSB so they will become insignificant from the perspective of the decoder.

5 stack.c

This is like the code stack except it's a stack for nodes. Internally, it's a pointer to type Node pointers. This means it's just an array of pointers to nodes where the top of the stack is just the furthest index. It's similar to the last assignment's hashtable data structure in that it's a data structure in an array. It functions the same way code.c does.

6 huffman.c

build_tree()

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.

build_codes

Using a global static boolean I determine whether *build_codes* has been called before, if not a global static code is initialized. The reason I do this is that this function is recursive and I do not want the Code to be reinitialized every time I call it. The function traverses the tree in post-order pushing recursing down the correct side and then popping bits off of a code. When it reaches a leaf the code is added to the table at the symbols index.

dump_tree()

This function also uses post-order traversal to print out leaves and interior node characters in the order of post order.

delete_tree()

This function uses post-order traversal to delete all the children of a tree and then finally the root. This works because no dependencies are deleted only dependents first.

rebuild_tree()

This function rebuilds the tree one node at a time by inverting the order of the queue by pushing nodes to a stack which are then popped off and joined the same way that the queue implementation makes a tree.

7 encode.c

Encode parses command arguments, if *-i* or *-o* are not specified they default to zero and one. If *-o* is specified but doesn't exist a file is created. Next, a temporary text file is created which holds a copy of the infile. During this copying process, a histogram of symbols is also created. Next, a tree is created with the histogram and then a code table is built from the tree. Then some metadata about the infile is returned into a stat struct. The size and permissions are received and inserted into a union of type uint8 and Header. Lastly, the seek of the temporary file is moved to the beginning, and the codes from the code table are written into the outfile in the same order as seen in the infile. All files if not standard input or output, are closed and the tree is destroyed. If *-v* is specified the stats are printed.

8 decode.c

The same options are parsed in the same fashion that encode parses its options. If an outfile is specified and doesn't exist it is also created. A union of uint8 and Header is created, then the header from the infile is read into the union. Then using the fields of the union decode checks the magic number and sets the outfile's permissions. Next, an array is made and of size *tree_size* from the union where then the tree bytes are inserted into. *rebuild_tree()* then rebuilds the tree and returns the root. Lastly, a while loop reads bits from the infile until the same amount of bytes as the uncompressed file size has been written to the outfile.