Vincent Siu
12/4/22
Vsiu2
asgn7

Assignment 7 Final Design

Goal for this assignment: To implement a Huffman encoding/ decoding algorithm in a way such that each symbol takes up less number of bits. We will encode using a huffman tree and keep track of the codes of each number in a Code, in which each code can be obtained by traversing through the tree to find the desired value.

1. **We will first define the implementation for the node, which everything else relies on for the Huffman encoder/ decoder**

- Define the struct for the node. This should include its
    - Left child (another Node)
    - Right child (another Node)
    - Its symbol
    - The frequency of the symbol.
- We now create a node
    - Memory allocate the node using malloc.
    - Set the symbol.
- We will define a way to delete the node
    - Free the node
    - Set it to null. We have to do that, so its children do not refer to it.
- Define a function to join the left child node and the right child node
    - Call Node create to create the parent. We must define the symbol to be “$” and the frequency of this newly created node to be left->frequency + right-.frequency.
    - Return the Node that was just created.
- Compare two different Nodes
    - Will check if one node frequency is bigger than the other. Return true if the first node is bigger than the second, return false otherwise.
- Define a function to print node (for debugging purposes)
    - Print the symbol, names, values, and frequency of the left and right child.

2. **Secondly, we will define the implementation of a stack, which will be used to reconstruct the tree. This will consist of a stack of nodes (what I have in the first section).**

- Define a struct. This includes
    - Top, how many items are enqueued
    - Capacity, is the maximum number of items that can be in this stack at one time
    - Nodes, these are the nodes consisting of each unique symbol, frequency, etc that may be used to be stacked.
- We have to create a stack
    - Memory allocate the struct of the stack.
    - Set Stack->capacity equal to the capacity that is passed in.
    - Set Stack->top equal to zero.

- We have to create a list of nodes. We will do this using calloc passing in the capacity and the size of the Node struct
- We will implement stack delete.
    - If the stack is not equal null
        - Free the array of Nodes of the list is defined.
        - Free stack itself
        - Set it equal null
- Check if a stack is empty
    - If stack->top is equal 0, return true
    - Return false otherwise
- Check if the stack is full
    - If stack->top is equal stack->capacity, return true
    - Return false otherwise.
- Implement a function to push a node onto a stack, returning a bool on whether or not it was successful.
    - If the stack is full
        - Return false
    - Else
        - While the current index of the array of nodes is inbounds
        - Set the current index of the node array (top) to the Node being passed in.
        - Increment stack->top by 1.
        - Return true.
- Implement a function to pop a node off a stack
    - If the stack is empty,
        - Return false
    - Else
        - Set *Node equal to the current index
        - Call node delete of the node at the current index of the current aray.
        - Return true.

3. **We will now define the implementation for the Code ADT. This will represent a stack of bits, almost like a bit vector ADT. For each symbol, we will keep track each of the left moves (0) and the right moves (1) starting from node all the way down to the desired symbol.**

- We will have a struct defined, which can also be used to pass by struct. The struct includes
    - Top
    - An array of bits of the size MAX_CODE_SIZE, each being uint8_t
- Define a constructor. No need to memory allocate nor destruct.
    - Set code->top = 0.
    - Run through the array of bits. We do not want any unwanted values.
        - AND the bits with 0<<7
    - Return code struct
- Define a function to set a bit in the code

- If i is bigger than max_code_size then return false
  - Get the position using the bit in the array using i%8 and get the array using i/8 to get the array indices.
  - OR this index with 1<<i%8.
  - Return true.
- Define a function to clear a bit in the code
  - If i is bigger than max_code_size then return false
  - Get the position using the bit in the array using i%8 and get the array using i/8 to get the array indices.
  - ~(AND) this index with 1<<i%8.
  - Return true.
- Define a function to get a bit in the code
  - If i is bigger than max_code_size then return false
  - Get the position using the bit in the array using i%8 and get the array using i/8 to get the array indices.
  - Create another set of bits with 1 and bit shift that to the desired position
  - And these two bits together
  - With the new bit array, bit shift it to the right by i
  - If that value is equal 1, return true
    - Return false otherwise.
- Define a function to push a bit onto the code
  - If the top is equal to the max, return false.
  - If the bit to push is equal 0
    - Increment top by 1
    - Return clear_bit at top index
  - Else if the bit to push is equal to 1
    - Increment top by 1
    - Return set_bit at top index
- Define a function to pop a bit. The bit to pop off is passed in
  - If the top is equal to 0, then return false
  - Get bit at the top index and set bit (parameter) equal to that.
  - Clear bit at the top index?

4. **Implementation for I/O. This will be used to read and write bytes from files or stdin/stdout.**

- Define an implementation to read_bytes from a file.
  - While not all nbytes have been read yet or we have reached EOF
    - The amount read should be added by the amount returned by read().
      - NOTE: read() takes in 3 arguments: a numeric file descriptor, a buffer, and how much to read.
  - Return the number of bytes read.
- While not all nbytes have been wrote yet or the buffer is NULL (no bytes to write)
  - The amount written should be set equal to the amount returned by write().

        - NOTE: write() takes in 3 arguments: a numeric file descriptor, a buffer, and how much to read.
- Return the number of bytes written.
- Implementation for write code
    - This is to write all the codes to outfile for encoding.
    - While the current code size is not equal to zero
        - Loop through each bit, and put its value into buffer.
    - When full, then flush codes
- Flush codes.
    - Zero out the last byte.
    - Write to outfile

5. **Implementation of a priority queue. This will consist of a queue of nodes with the goal of having the nodes with the lowest frequencies at the bottom, so those can be dequeued.**

- Struct for a queue.
    - We have a max capacity to keep track of the max queue size.
    - We will have the number of items currently occupying the queue
    - Node double pointer, since we will be holding an array of nodes.
- Constructor for the queue.
    - Capacity passed in should set the capacity property of the priority queue.
    - We allocate memory for *capacity* number of nodes. Use calloc.
    - Set the current number of nodes occupying the queue to 0.
- We will now define several heap functions in order to build a heap and down heap. Min heap.
    - The left child
    - The right child
    - The parent
    - Building an up heap
        - While node_index is bigger than 0 and the node compare of array_n and array_parent of n return false,
            - Swap the array positions
            - Set node_index = node_index parent
    - A down heap
        - Set a node index = 0. We have to start from root
        - While left child indice is smaller than the heap size
            - If the right child indice is equal to the heap size
                - Set the smaller value equal to the left child. This is the case where the right child doesn't exist.
            - Else
                - Set the smaller equal to the left child if node comparisons of array left and array right child return false. This means the first is smaller than the second.

            - If node compare of array indexed at n and array index at smaller returns false
                - Break
            - Swap the array values of n and smaller
            - Set n = smaller
    - Building a heap
        - Allocate an array that will hold a heap
        - For each n until the capacity of the heap
            - Set heap index at n to the current index of array passed in
            - Call up heap
- Note: If heap fails to work, I will have insertion sort as an alternative. What this will do is that each time you insert something, go through the array and see in which index will the node fit in. The array will be from greatest to least for more efficiency.
- Enqueuing the heap
    - First check if the current size is equal to max size. If it returns false.
    - Set the current index of the array to be heaped to node passed in, that is the first empty slot within the array.
    - Call build heap
    - Increase current size
    - Alternative: while the node at current index is not null,
        - Check if there are no elements in the array, if there are not, just insert it into the first slot.
        - Compare the node in the array and node to be put in and if it returns false (meaning if the current in the node in the array is **smaller** than the one to be put in) put this node before the one where the comparison statement returned false.
        - Shift everything to the right by 1 spot
        - Update the number of items in array.
- Dequeuing the heap
    - For insertion sort alternative, set the node n to array index at the current number of items in the array.
    - Set 0th index of array to null
    - Call down heap
    - Decrement the number of items in array

6. **Huffman coding module. This includes the encoding and decoding implementation. Encode and decode files will only call these functions.**

- Build a Huffman tree.
    - Loop through the histogram and create nodes for each of them and track the number of non zeros for the histogram.
        - Enqueue these nodes. We have to do that so they are in the proper min heap order in order for us to create our tree.
    - If there are less than 2 non zero symbols, set the histogram indices 0 and 1 to 1.
    - Create nodes for those?

- We have to now create a sum of the children's frequencies. Make 2 temporary nodes.
- Run through the list and call dequeue two times while we have two or more nodes.
  - Node join these two nodes we have just dequeued.
- Set the last node children to be the last two Nodes created from dequeue.

- Build a new code table so we have codes for each symbol from what we get in traversing a tree
  - Init a new instance of code.
  - We will transverse the tree using post order transversal to get the code. This includes recursive calls
    - Starting from the start node, if node is not null
      - If the node is leaf, meaning if the left children and right children are null, then c will now represent the path to the symbol
      - Else, push a bit, read the left node recursively, pop a bit. Then to read the right region we will push a bit, then read the right node recursively, then pop a bit when returning the right link.
- Dumping a tree. We do that so we can write its contents to a file and, most importantly, we know how to properly reconstruct the tree.
  - If we are at root
    - Recursively call dump two times for the left part and right part of the root.
  - If root children are null
    - Write out the leaf node
    - And write out its symbol
  - Otherwise
    - Write out "I". This represents an interior node.
- **Note: The steps below are to encode text to outfile.**
- We have to construct a header in order to write encoded content to outfile.
- Set the magic, as from the header, to the macro MAGIC. Very important.
- To get the permissions of the file, use fstat with a buffer pointer passed in.
- Set permissions of the outfile passing buffer as the permission that we got earlier.
- Set tree size.
- Get file size also using fstat with buffer.size.
- Call write, and write all the things in the header to outfile
- Call dump tree and write its contents to outfile
- Write code from in file to outfile using write code.
- Close both of these files.
- **Note: Now we move on to the steps to decode a text.**
- From the infile, read the header ad if the magic number does not match, then print an error message and return error code.
- Set the permission of the outfile as based on what's in the header of infile
- Reconstruct the tree.
- Read the contents of tree dump from 0 bytes to n bytes

Vincent Siu
12/4/22
Vsiu2
asgn7

- If the current index is an L, then the next element will be a leaf, so use node create and push it onto a stack.
- If the current index is an I, or an interior node, pop off the stack to get the right child child of the interior node and then do the same thing for the left child.Then node_join these together and push its parent into the stack.
- The last node will become the root
- Now we have to read the infile.

7. **Encode.c will contain the encoding portion of Huffman**

- This program will accept options h,i,o,v.
    - h: print out a help message
    - i: specifies the input file.
    - o: specifies the output file
    - v: prints the compression statistics to stderr. This will print how much was saved depending on the compressed file size and uncompressed file size
    - Open the file
    - Set a pointer for the buffer
    - Call read byte. The buffer should have everything
    - Create an array of uint64t's. This will be used as the histogram.
    - While the buffer is not null
        - If the current index has a valid character, run through the histogram and find the valid character index to increment.
    - Call code table constructor, to keep track of the transversal codes for each of the valid symbols.
    - Encode the huffman tree. Can be done using a call to build_tree implemented in huffman.c.
    - Reset to beginning of file. While we have not read everything, loop through buffer if the letter matches with the code, write it out to the code array
    - Call write code.

8. **Decode.c will contain the decoding portion of Huffman.**

- Similar to encode, it will take h (help message), i (infile name), and o (outfile) inputs, and v (decompression statistics).
- Read emitted dumped tree from input file. This will call a function in huffman.c.
- Call to huffman.c to reconstruct the huffman tree.
- Call read bit to read down the huffman tree.