Nayeel Imtiaz
Professor Long
CSE 13s
28 October 2021

Assignment 5 - Design Document

Description:

The main purpose of this lab is to make a Huffman encoder and decoder. There is some file compression involved in this lab. It will take data from user input to compress it into huffman code. A node abstract data type is needed to code the huffman program. The node ADT should be able to create nodes, delete nodes, and join two nodes together to form a parent node. Also, a stack adt will be required just like Lab 4. The stack should be able to do the normal things such as creating the stack, deleting the stack, pushing an element, popping an element, checking if empty, and checking if full. The priority queue ADT is a structure that holds nodes in increasing order by frequency. The array that holds the nodes automatically sorts it after enqueuing each node. The dequeue function removes the node at the very top and shifts all the other nodes up. The code structure can push and pop bits. Normally bits are not easily accessible, but the Code ADT makes it possible. The code Io.c file makes it easier to use low level system calls for reading and writing input/output. The standard library is not allowed for reading and writing. The Huffman encoder libraries should be able to build trees and implement many different tree functions, and the Huffman decoder should be able to undo the tree back into the original input file.

Pseudocode:

Stack.c

Stack_create(capacity):
    Dynamically allocate space for a stack
Stack_delete(address of stack):
    Delete memory allocated for stack
Stack_empty(stack):
    Return true if stack is empty, else return false.
Stack_full(stack):
    Return true if stack if full, else return false.
Stack_push(stack, i):
    Push i onto stack if the stack is not full.
Stack_pop(stack, address of i):
    Pop from stack and move it to i if stack is not empty.

Node.c

Node_create(symbol, frequency):
    Create and allocate space for a node with the inputted symbol and frequency.
    Set the left and right node to NULL.
Node_delete(address of node):
    Delete memory allocated for the node
Node_join(left node, right node):

Take two nodes together and combine it in one parent node.
Set left node to left node and right node to right node for the parent node.
Return the parent node

Pq.c

pq_create(capacity):
Dynamically allocate space for a priority queue filled with nodes.
pq_delete(address of priority queue):
Delete memory allocated for priority queue.
pq_empty(priority queue):
Return true if pq is empty, else return false.
pq_full(priority queue):
Return true if stack if full, else return false.
enqueue(pq, i):
Maintain a sorted heap.
Whenever an element is inserted into pq, sort the entire heap.
Organization of the nodes is done in this function.
Return false if priority queue is full.
dequeue(pq, address of i):
Pop the element at index 0.
Move every other element up by one unit.
Return false if priority queue is empty.

Code.c

code_init():
Make a code structure without allocating dynamic memory.
code_size(code):
Return the number of bits.
code_empty(code):
Return true if the code structure is empty.
code_full(code):
Return true if the code structure is full.
code_set_bit(code, index):
This function sets the ith bit to 1. There are
256 bits to account for as there are 32
bytes in total in the code structure. You
have to use the bitwise operations left shift
and 'bitwise or' to set the bit to 1.
bool code_clr_bit(code, index):
This function does the exact same thing as
code_set_byte, except it sets the ith bit to
0. You have to use the bitwise operations
left shift, bitwise not, and 'bitwise and'
to clear the ith bit.

code_get_bit(code, index):
    This function gets the ith bit. It does not change
    any bits in the code structure at all.
code_push_bit(code, element)
    Use set_bit if element == 1
    Use clr_bit if element == 0
    Return false if the code structure is full.
    Add top by 1
code_pop_bit(code)
    Subtract top by 1
    Use get_bit to see if the top element is 1 or 0.
    Assign pointer value to 1 or 0.
    Return false if the code structure is empty.

Encode.c

Process arguments using get opt function.
Take file name from -i program argument.
Make a histogram.
Increment first and last symbol by 1 in the histogram.
Build a tree and make a code table.
Make a header and use fchmod.
Write header to outfile.
Write constructed huffman tree.
Write code for each symbol.
If the user enters the wrong input or -h command argument, print the following.

```
printf("SYNOPSIS\n")
printf("A Huffman encoder.\n")
printf("Compresses a file using the Huffman coding algorithm.\n"
printf("\n")
printf("USAGE\n")
printf("  ./encode [-h] [-i infile] [-o outfile]\n");
printf("\n");
printf("OPTIONS\n");
printf("  -h           Program usage and help.\n");
printf("  -v           Print compression statistics.\n");
printf("  -i infile    Input file to compress.\n");
printf("  -o outfile   Output of compressed data.\n");
```

Encoded output should be shown in the output file.
Show stats if the -v flag is raised.
Close all files.

Decode.c

Decode Huffman code back into original user input.
Terminate the program if the magic number is not read in the program.
Rebuild huffman tree from encoded input.

Walk through the tree: if 0 is read go left. If 1 is read go right.
Write bytes whenever a block amount is in buffer or it is time to flush.
Show the help options to explain to the user how the program works.

```
printf("SYNOPSIS\n");
printf("  A Huffman encoder.\n");
printf("  Compresses a file using the Huffman coding algorithm.\n");
printf("\n");
printf("USAGE\n");
printf("  ./encode [-h] [-i infile] [-o outfile]\n");
printf("\n");
printf("OPTIONS\n");
printf("  -h           Program usage and help.\n");
printf("  -v           Print compression statistics.\n");
printf("  -i infile    Input file to decompress.\n");
printf("  -o outfile   Output of decompressed data.\n");
```

Print stats if -v flag is raised.
Close infile and outfile.