CS 3345

Programming Assignment #4

University of Texas at Dallas — *October 31, 2024*

Lecturer: Dr. Andrew Nemec — *Due: November 15, 2024*

Introduction.

When converting a written English-language text to binary for digital transmission or storage, a naive approach might be to encode each character in its respective ASCII character, where each character has a unique 7-bit binary string associated with it. However, in practice different characters do not appear with the same frequency. For example, if you picked a letter at random in a book, you would likely find that the letter `e` appears with a frequency several orders of magnitude larger than that of the letter `z`. A more efficient encoding scheme would be to use a shorter binary string to encode `e` and a longer one to encode `z`. One such approach to data compression is Huffman coding.

This assignment is intended to give you experience with implementing the Priority Queue ADT with an underlying binary heap.

Huffman Coding.

As discussed in class, Huffman coding is a data compression technique that assigns each character a bitstring whose length depends on the frequency of the character; more frequent characters have shorter bitstrings and less frequent characters have longer bitstrings. (For more information on Huffman coding see the lecture notes and the worked out example from class.)

To create a Huffman code, we first construct a prefix-free tree using the following steps:

1. Create a priority queue P out of the initial characters, with the frequency of each character as that node's key. You should use the constructor that performs a BuildHeap operation.
2. While P has more than one element:
   (a) Remove the first two elements from the priority queue, and create a dummy node, with the two elements as its children.
   (b) Make the first element removed from P the left child and the second element removed the right child.
   (c) The key of the dummy node should be the sum of the two children nodes.
   (d) Insert the dummy node back into P (the size of P will have decreased by one).
3. Remove the minimum element from P, which will be our prefix-free tree T.

Once we have our prefix-free tree T, we can construct a codebook. This is done by associating the path from the root to the leaf node containing the character with a bitstring, with a *left* edge corresponding to a 0 and a *right* corresponding to a 1. For example, a sequence of edges *left*, *left*, *right*, *right*, *left*, *right* would correspond to the bitstring 001101.

Priority Queue ADT.

The Priority Queue ADT usually has the following operations defined for an instance P:

- SIZE( ): Return the number of elements in P.
- EMPTY( ): Return true if P is empty and false otherwise.
- INSERT(k, v): If P does not have an element e with key equal to k, then add element e to P, otherwise replace the value of element e with v; return a pointer or reference to the inserted or modified element.
- DELETEMIN( ): Remove from P the element e with the minimum valued key k; an error condition occurs if P has no such element.
- MIN( ): Return a pointer or reference to the element e with the minimum valued key k; an error condition occurs if P has no such element.

You will implement the Priority Queue ADT using the binary heap data structure discussed in class. In addition to implementing all of the operations listed above, you must implement a constructor for your priority queue:

- A default constructor (optional) that creates an empty heap.
- A constructor that takes in an array A as input and creates a heap from it using the BUILDHEAP operation discussed in class.

When implementing your binary heap, you can choose to index your array from either 0 (similar to the heap sort) or 1 (as done in class for priority queues). **Make sure you don't forget which one you've chosen.**

Instructions.

Implement your priority queue data structure as described in the section **Priority Queue ADT** located above.

You will turn in a single file, whose main program should do the following:

- Use the file `merchant.txt` (from the previous assignment) as the input; it is Shakespeare's *The Merchant of Venice* with only lower case letters and spaces.
- Scan through the input text and count the number of times each character appears. Ignore any newline characters (e.g., `\n` and `\r`).
- Use your priority queue implementation to create a codebook for your Huffman code as described in the section **Huffman Coding**.
- Prompt the user for a positive integer N, the number of characters from the input text that should be encoded for output text.
- Write to a text file an output text containing:

  - Your codebook, with the bitstrings of each characters, in the order `' '`, `'a'`, `'b'`, etc., with each bitstring on a separate line.
  - N lines corresponding to the first N characters in the input file. Each line should contain (separated by tabs):
    * The corresponding bitstring for that character.
    * The running sum of total number of bits used so far using your Huffman code (do not count the codebook).
    * The running sum of total number of bits you would have used so far using the 7-bit `ASCII` representation.
  - In all there should be $N + 27$ lines of output.

- Use good style and make sure to comment your code well. If you discuss the problem with someone, write their name in your comments.
- Turn in your program as one source code file titled `FILIPA4.java` or `FILIPA4.cpp`, where `FI` is the first two letters of your first (given) name and `LI` is the first two letters of your last (family) name, e.g., Andrew Nemec $\rightarrow$ `ANNEPA4.cpp`.

You may base your code on that in the textbook, but otherwise all code should be your own.

If desired, you can use the premade data structures `std::vector` **in C++ or** `ArrayList` **in Java to implement your binary heap, as well as any string data structures you wish. However, you should not use any other premade data structures.**