CSCI 620 Spring 2016

Program 5: Huffman Compression, Step 2

1 The Goals and the Purpose.

1. To use an array and a heap to build a Huffman tree.
2. To implement the first two phases of a Huffman Encoding application.

A Huffman code can be used to compress a file. It replaces fixed-size, one-byte characters by variable- length codes (strings of bits). The codes for the most frequently used characters will be shorter than 8 bits, those for unusual characters may be longer. A new code is defined for each file. This helps to keep the codes short, since no codes are generated for characters that are not used in the file. The input file is read twice, once to generate the code, then again to encode the file. This assignment and the next two form one large project to generate and use a Huffman code. It is essential to debug this part before going on to the rest of the project.

2 Modules to Create

Class Tally Create a class named Tally. The Tally constructor should take a filename parameter, and open and verify the file. The Tally destructor should close the file.

Define `tallyArray` to be an array of 256 integer counters, all initialized to 0.

Functions in the Tally class include:

- `void doTally()`: This is similar to but easier than the tally function from your Cryptogram program.

  Read the input file one character at a time. Use the character to subscript the tallyArray and increment the counter. Return when end-of-file is found. Remember that, in C and C++, you must read from the file before you test for eof. (Note: DO NOT try to read the file as a series of strings or lines. You will be sorry.)

- `int& getTally()`: Return the tallyArray to the caller. If the return value is defined as an `int&`, this return statement is appropriate: `return tallyArray[0]`. The caller will store this address in a variable of type `int*`.

A Heap Class Adapt the function definitions from the class Heap example to implement a MIN-heap that stores Trees instead of numbers.

- Your Heap should have one data member: an array of 257 `Node*`.
- Write a Heap constructor with one parameter, a tallyArray. See details below.
- Implement the downHeap(), buildHeap(), and print() functions first, and debug that much.
- Write `Heap::print()`
- After your heap works, implement `push()` and `pop()` to put things into the heap and take them out.
- Also write a function `reduceHeap()` that calls `push()` and `pop()` to reduce the heap to one element. See details below.

The Heap Constructor

- Start by initializing slot 0 of the heap to something distinctive, such as `MAXINT`. This will never be used because the real data will start at subscript 1. However, during debugging, it can be very useful to have something there.
- Then process the data in the tally array:

  - Take each non-zero tally in the array.
  - Use the character and the counter to create a new Node. See details below.
  - Store the Node* in the next unused slot of the heap array.
- Call the buildHeap function to arrange the data in heap order, according to the frequency of the letters, with the least frequent letter in slot 1 of the array.

Return when all nonzero tallies have been transferred to your heap.

`reduceHeap()` The goal of this function is to combine two Nodes into one. Eventually, all the Nodes in the heap will be combined into one Huffman tree. Repeat this until there is only one node left:

- Call `pop()` to remove the node at the root (slot 1), replace it by the last node in the heap, and perform a `downHeap()` operation to re-establish heap order. The popped node will be the left son of a new node.
- Call `pop()` again to remove the node at the root, replace it by the last node in the heap, and perform a `downHeap()` operation to re-establish heap order. This node will be the right son of a new node.
- Create a new Node with these two Node*s.
- Push the new node onto the end of the heap and perform an `upHeap()` operation to re-establish heap order.

Return when there is only one element left in your heap, in slot 1.

Class Node and Tree

- Define a tree Node with four data members: a char, an int, and two Node*s. Also typedef Tree to be a Node*.
- Write a Node constructor that takes two parameters: a char (the input character), and an int (its frequency). Use the parameters and two `nullptr`s to initialize the Node object.
- Write a second Node constructor that takes two different parameters: two Node*s (the left and right sons of the new node). Use the parameters to initialize the pointer fields of the Node object. Set the char field to any distinctive and highly visible char value: you will never use it but you need to be able to see it on a printout. Set the frequency field to the sum of the frequencies of the left and right sons.
- You may also need a default constructor for Node. If so, use `=default`.
- For this assignment, define a null default destructor. A real destructor will be needed in the last phase of this program, but not at this stage. The output of this phase will be a binary tree of Nodes that contains every Node that you have allocated.
- Write get functions *if and only if you need them.*
- Write a print function that displays all four fields of a Node with no newlines. The pointers will be printed in hex if you use <<.
- Write a `printTree()` function with two parameters, a Node* and string. The first time you call this function, the parameters will be the root of the Huffman tree and the empty string.

  The function will do a recursive infix treewalk: recursively print the left side, print the node, then recursively print the right side. Each time you make a recursive call, append four spaces to the string parameter. This will print the tree in an indented format with virtually no effort.

  Every leaf node has two nullptr sons, so you know it is time to return if either one of them is nullptr. In that case, print "————–" instead and return from the recursion.

The main Function

1. Either read an input file name or grab one from the command line.
2. Create a Tally object for this filename and call `doTally` to count the characters in your input file. Save the int* returned by `doTally()`.
3. Construct a Heap object using the filled Tally array. The Heap constructor will call buildHeap().

When the Heap constructor finishes, print the heap array and make sure the contents of the heap are in legal heap order. Get this debugged before you go further. Then go on to the next phase:

1. Call `reduceHeap()` to unify the nodes in the Heap.
2. Print the resulting tree that is in subscript 1 of the heap, using your recursive treewalk. recursive tree walk.

3 Future Work

The last two phases of the Huffman project are:

- P6: Do a recursive treewalk to generate a code. Write the code to a binary file.
- P7: Reopen the original text file. Encode each letter in it. Write the encodings to a binary file.

In a real Huffman project, the code and the encoded message are written to the same file. I am asking you to keep them separate for reasons related to debugging and grading.