CSCI 620 Spring 2016

Program 5: Huffman Compression, Step 2

1 The Goals and the Purpose.

1. To use an array and a heap.
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 You have a complete and working Tally class from Program 2. Include your Tally class as part of this project.

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 Node constructor that takes two parameters: a char (the input character), and an int (its frequency). Use the parameters and two `nullptrs` 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 arbitrary 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 a print function that displays all four fields. The pointers will be printed in hex if you use <<.
- Write accessor functions *if and only if you need them.*

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: a `vector<Node*>`.
- Write a Heap constructor with one parameter, an array of type Tally. See details below.
- Implement the downHeap, buildHeap, and printHeap functions first, and debug that much.
- 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 pushing a dummy Node* into the vector to occupy subscript 0. We want the real data to start at subscript 1.
- Then process the data in the tally vector:
  - Take each non-zero Tally in the array.
  - Use the character and the counter to create a new Node*.
  - Push the Node* into the vector.
- 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 vector

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

- Call `pop()` 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 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 to Node*s.
- Push the new node onto the end of the heap and perform an `upHeap()` operation to re-establish heap order.

The main Function

1. Create a Tally object and use it, as you did in P2, to tally the characters in an input file. Main should own the Tally array.
2. Construct a Heap object using the filled Tally array. The constructor will call buildHeap().

When the heap constructor finishes, print the heap array and make sure the heap is correct. 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. To do this, you will need to do a recursive tree walk. In this recursion, print the contents of a node then indent four columns and recursively print the let son, then the right son. Return from the recursion when the tree you are trying to print is a nullptr; print "———-" instead. (Remember: every leaf node has two nullptr sons.)

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.