

PONDER 08 : HUFFMAN CODES

Due Saturday at 11:59 PM MST

The next programming assignment will be to implement the binary tree data structure and use it to generate Huffman Codes from a given alphabet.

Binary Tree

Create a class to capture the notion of a single node in a binary tree. There will be four member variables: a pointer to the node to the right child (pRight), a pointer to the node to the left child (pLeft), a pointer to the parent (pParent), and the data associated with the node. Of course, any data-type will need to be supported, so your class will be a template class. It will need to be defined in its own header file (bnode.h). The class name must be BinaryNode and will need to support the following operations:

* **Constructors**: Default constructor (setting pLeft, pRight, and pParent to NULL), and a non-default constructor (taking a template variable as a parameter and assigning all the pointers to NULL).
* **addLeft()**: Adds a node to the left of the current node. There are two versions of this method: one that takes a template variable as a parameter, and one that takes aBinaryNode \* as a parameter. In the former case, a new node will be allocated. In the later case, the passed node will be used. In both cases, pLeft of the current node will point to the new node, and pParent of the new node will point to the current node. The return value is this. In the case of an allocation error, the following c-string exception will be thrown:  
  ERROR: Unable to allocate a node
* **addRight()**: Exactly the same as addLeft() except on the right side.

Note that all the member variables and methods must be public. At this point, a BinaryNode has no way to validate whether a given node value is valid or not. When we encapsulate this class into another class (Lesson 08), we will be in a better position to validate and make the member variables private. In addition to the above methods, two non-member functions will be needed:

* **deleteBinaryTree()**: Takes a BinaryNode as a parameter and deletes all the children and itself. This is a recursive function.
* **Insertion operator**: Takes a constant BinaryNode \* as a parameter and displays the tree under the passed parameter. This is a recursive function traversing the tree in infix order. A single space will be displayed after every element.

Driver Program

A driver program is provided. This file (/home/cs235/week08/week08.cpp) will pound-include your header file (bnode.h) and expect a template class BinaryNode to be defined therein. This program will exercise your class, filling the container with user input and displaying the results. As with previous assignments, a makefile will be provided (/home/cs235/week08/makefile). You will need to create a header file (huffman.h), and an implementation file (huffman.cpp) as well as the binary node header file (bnode.h).

Huffman Codes

In addition to passing the four test functions for the BinaryNode class, you will also need to use the BinaryNode class to implement a Huffman Code generator. A discussion of this exists in section 15.1 in the textbook. You will implement the function called huffman() that does the following:

1. Prompts the user for a filename containing value frequencies.
2. Reads the values into some data structure. You may use any data structure developed this semester thus far; you are not allowed to use anything from the STL.
3. Generate the Huffman Codes for the data in the data structure.
4. Display the resulting Huffman Codes in the same order they were read from the file.

For example, consider the following data in a file called huffman1.txt:

A 0.2  
B 0.1  
C 0.1  
D 0.15  
E 0.45

The resulting execution of the program would be:

Enter the filename containing the value frequencies: huffman1.txt  
A = 111  
B = 100  
C = 101  
D = 110  
E = 0

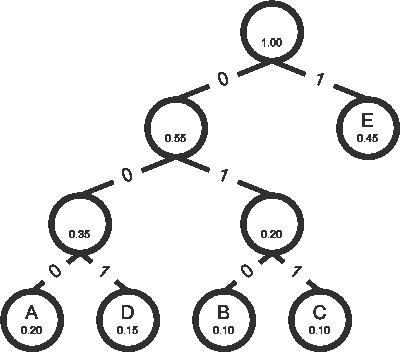
A few hints to help you with your program:

* You might want to create a class that contains a single Huffman tree. To follow the algorithm in the textbook, you will need to maintain a collection of these trees.
* Every node in a Huffman tree will have more than one data value (the token and the frequency). You might want to consider using the Pair container for this. The Pair class is located at:

/home/cs235/week08/pair.h

* As the textbook describes, there are many possible equivalent Huffman codes for a given set of token-frequency pairs. To pass test-bed, the following variation to the algorithm must be followed:
  + When two trees are combined, a new tree will be created. This new tree goes into the left-most of the two slots. The newly vacated right-most slot is then filled with the last element in the list.
  + When two trees are combined into a new tree, a new root will have to be created for this new tree. Under this root, the smaller of the two trees goes to the left and the larger of the two goes to the right. If they are the same, then the order they are in the list of trees is preserved.
  + When generating the codes from the finished tree, the left child inherits the 0, the right child inherits the 1.

Therefore, the final Huffman tree for the above example is:



Common Mistakes

The most common mistakes students make with this assignment include the following:

* **Failure to make a Huffman class**. While it is possible to solve this problem with procedural tools, it is much easier to use a class to encapsulate the notion of a Huffman Code.
* **Wrong node data-type**. Some thought is needed in determining what data structures are needed and what data types are to be stored in each data structure.
* **Using the wrong data-type for a node value**. A Huffman Code may be a single character or more than one character. Make sure your program can handle both.

Test Bed

The testBed for this assignment is:

testBed cs235/week08 week08.tar

You can also run testBed on the executable:

testBed cs235/week08 a.out

Of course, you will need to pass testBed to get full credit on the assignment.

Submitting

You will submit this assignment individually using the Linux submit command. Please:

1. Create a TAR file built from the makefile, which will contain five files:
   * makefile: Directly from /home/cs235/week08/makefile except with your edits on the comment block.
   * bnode.h: Your class definition for BinaryNode.
   * huffman.h: Containing the prototype for huffman() and any other function you may need.
   * huffman.cpp: Implementation for all the functions and classes necessary for the Huffman Code generation.
   * week08.cpp: Unmodified from /home/cs235/week08/week08.cpp.
2. Run the program by hand a few times through all four test cases as well as the Huffman code generator.
3. Verify your solution with testBed.
4. Submit your file using the submit command. The submit command will prompt you for your instructor, the class (cs235), and the assignment (week08). You submit your file with:

submit week08.tar

Your program will be graded according to the following rubric:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Exceptional 100% | Good 90% | Acceptable 70% | Developing 50% | Missing 0% |
| BinaryNode Interface  20% | The interfaces are perfectly specified with respect to const, pass-by-reference, etc. | week08.cppcompiles without modification | All of the methods in BinaryNodematch the problem definition | BinaryNodehas many of the same interfaces as the problem definition | The public methods and variables in the BinaryNodeclass do not resemble the problem definition |
| BinaryNode Implementation  10% | Passes all fourBinaryNodetestBed tests | Passes three testBed tests | Passes two testBed tests | Passes one testBed test | Program fails to compile or does not pass any testBed tests |
| Huffman Code  40% | The code is elegant and efficient | Passes the Huffman Code testBed test | The code essentially works but with minor defects | Elements of the solution are present | The Huffman Code problem was not attempted |
| Code Quality  20% | There is no obvious room for improvement | All the principles of encapsulation and modularization are honored | One function is written in a "backwards" way or could be improved | Two or more functions appears "thrown together" | The code appears to be written without any obvious forethought |
| Style  10% | Great variable names, no errors, great comments | No obvious style errors | A few minor style errors: non-standard spacing, poor variable names, missing comments, etc. | Overly generic variable names, misleading comments, or other gross style errors | No knowledge of the BYU-I code style guidelines were demonstrated |

Please make sure to fill out the program header in the makefile with the following information: the amount of coding time required to complete the assignment and what was the most difficult part. Failure to do this will result in a loss of 10% on the assignment.