COMP 2150 – Winter 2021 – Assignment 4

Due April 15 at 11:59 PM

April 3: Fixed an incorrect weight on step 8 of the Huffman coding walkthrough, added information on charCodeAt().

In this assignment, you will implement javascript classes to support Huffman data compression. Data compression (as used in zip and rar, for instance) is the process of taking large files and making them smaller by analyzing their contents. In this assignment the algorithm we will use is called Huffman encoding, and it works by building a binary tree that encodes each character as a sequence of 0s and 1s. The encoding is variable-length: some characters will be encoded as a short sequence of 0s and 1s, while other (more rare) characters will be encoded with longer sequences.

Part 1: Hash Table Dictionary

Create a javascript class called Dictionary that is implemented as a hash table. In the Dictionary, each entry is a Key-Value pair, and there can't be two Key-Value pairs in a dictionary with the same Key (that is, keys are unique in a dictionary). The Key will be an object that has a hashVal method (see below for the Hashable class and hashVal information). A Value can be anything (except undefined).

When initialized, the Dictionary should be given a size as a parameter[1]. The Dictionary will create an array of that size as a field. When adding or searching for a Key-Value pair in the Dictionary, the position of the pair in the hash table will be the key's hashVal modulo the length of the array.

The hash table should use separate chaining. If there is a collision between two elements based on their hashVal, the hash table should place both elements in a linked list of elements at the array position. The order of elements in the linked list at a position is not important.

The Dictionary should implement the following methods:

1. put(k,v): takes a Hashable key k and a value v and adds it to the dictionary, if it does not exist. If a key-value pair with k as the key already exists, the current value is replaced with v. The method does not have a return value.
2. get(k): takes a Hashable key k and returns the value v associated with it, if it appears in the dictionary. The method should return undefined if the key does not appear in the dictionary.
3. contains(k): takes a Hashable key k and determines if it is a key in the dictionary. Returns a boolean value.
4. isEmpty(): returns a boolean depending on whether the dictionary is empty or not.

For each of the first three methods, the method should ensure that the key parameter has type Hashable (**or** has a hashVal() method, and an equals() method, as appropriate), and it should throw an error if the condition does not hold.

[1] When using Dictionaries in this assignment, you should set the size to a large value of your choice.

Hashable

You should also implement a hierarchy of Hashable classes that can be stored as Keys in the hash table. The Hashable class should be abstract (using the techniques shown in class) with an abstract hashVal() and an abstract equals(x) method. You should implement at least the following two Hashable subclasses:

- IntHash, whose hashVal function is the value of the integer. Two IntHash objects are equal if they contain the same integer value.
- StringHash, whose hashVal function is the following expression:

  $$s[0]*p^{n-1} + s[1]*p^{n-2} + ...+ s[n-3]*p^{2} + s[n-2]*p + s[n-1]$$

  where s[i] is the ASCII value of the i-th character of the string, n is the length of the string, and p is a prime.[2] You can assume that all characters in the strings are ASCII characters. Two StringHash objects are equal if they contain the same strings.

  Note: to get the ASCII value of a character from a string, you can use .charCodeAt(): x.charCodeAt(i) gives the ASCII value of the character at position i of the string x.

The subclasses can have additional methods if you need them, including getters. You may implement further Hashable subclasses, but you are not required to do so.

Unit Testing

Construct a set of at least **five** different unit tests for your Dictionary. To do unit testing in javascript, you should simply write a separate file of tests that can be run (i.e., you should not need to use a testing framework like jest or mocha -- do not assume the markers will have either installed on their system.)

1. Create a test file with "let assert = require('assert');" near the top of the file.
2. Write a file with a series of tests in whatever format you would like. Each test should be its own function.
3. Write a main function that calls all of the test methods.
4. Have a single executable line that calls the main function.
5. In your tests, say assert([something]); to give assertions. You need at least one assertion per test method.

At least two of your tests should test boundary conditions: an empty Dictionary, and a Dictionary with one element. You should develop additional tests that will verify that your code is working as appropriate. You will be graded on your tests, so write useful tests.

Save all of your unit tests as part of a single file and submit them with your code. The markers will be running your tests, so ensure that they pass. **Give instructions on how to run them in your readme file.** You are strongly advised to complete the unit tests before starting further parts of the assignment.

Part 2: Huffman Trees

[2] In your code you should use a prime number of your choice that is less than 1000.

To allow us to complete Part 3, you will create a class to represent a Huffman Tree, which is a type of binary tree used in Huffman Encoding (in Part 3). A Huffman Tree has the following properties:

1. Each tree is a binary tree. The children are not ordered in the same way as a binary search tree – we only talk about "left" and "right" subtrees. The tree is also not structured in a balanced way: left and right subtrees of a node, for instance, may have different heights.
2. Every internal node has two children.
3. Leaf nodes in a Huffman tree are labelled with a single character. Internal nodes in the tree are not labelled with any data.
4. Each Huffman Tree has a weight, which is a floating point value between zero and one.

Create a class that represents Huffman Trees. You will need the following operations:

1. Creating a new tree: you should be able to create a tree that has one node, based on two parameters: a single character and a weight. This will create a tree with one node (a leaf node) and the weight given.
2. Combine two trees to create a new tree: you should be able to create a tree from two other trees (the left and right subtrees). The new tree should have a new root node, whose children are the two subtrees given, and the weight of the new tree is the sum of the weights of the two subtrees given.
3. You should implement a compareTo method for Huffman Trees. This method should take a parameter (another Huffman Tree) and return +1,0 or -1 to represent whether the parameter tree comes before (+1) or after (-1) the tree whose method is being called.

   For any two trees, the tree with the lowest weight should come first. If the two trees have the same weight, the tree that contains (in any leaf) the smallest character (in order) should come first.[3] For example, if two trees had the same weight and if one tree contained the characters "C","O","M","P" and the other contained "E","N","G","L", then the first tree would come first (it contains "C"). Because of how Huffman Trees are built, you do not have to worry about two trees containing the same smallest character (but if they did, you could return 0).
4. You will additionally need a search or traversal method in your Huffman Trees. This should be able to determine, for each character in the tree, the path (left/right moves) from the root to the leaf labelled with that character. This is described more in step 4 of the algorithm in Part 3.

You are strongly encouraged to write unit tests for these trees, however it is not required.

Part 3: Huffman Encoding

After creating your Dictionary and Huffman Tree classes, use them to implement Huffman encoding. Huffman encoding is used to compress text files by replacing characters in the text with sequences of bits (0-1).[4]

[3] To compare characters, use the less than operator < in javascript.

[4] You will notice after finishing this assignment that this technique does not compress files – this is because we are encoding 0s and 1s as characters, not as bits. This saves us the issue of having to deal with individual bits and writing them to files.

The algorithm works as follows:

1. The input to the algorithm is a text file.
2. Before encoding anything, the entire file should be read, and character frequencies are calculated for each character that appears in the file. You then calculate, for each character X in the file "what is the percentage of the file that is the character X?" (as a number between 0 and 1). Store this information in a Dictionary from Part 1. This calculation should be done for all characters, including spaces, newlines, etc.
3. Next, the Huffman Encoding is constructed. The Huffman Encoding is built by constructing a set of Huffman trees, and then joining these trees until one final Huffman tree remains.
    a. Initially, we construct a tree for each character in the file. The weight used for these initial trees are the percentages (0-1) calculated in step 2.
    b. At each step, we choose the two Huffman Trees in the set that are smallest, according to the order given by the compareTo algorithm in Part 2. We join these two trees, using the joining algorithm in Part 2. (The left tree should be the smaller tree using compareTo). Remove the two trees that were joined from the set of Huffman Trees and add the new tree (thus, the set will have one fewer tree at each step).
    c. Repeat this until all of the Huffman Trees are combined into one tree, which we call T.

   To support this step, you may need to construct additional javascript classes.
4. Then, the Huffman code is calculated. For each different character X that appears in the file, calculate the path from the root of T to the leaf that is labelled with the character X. Build a string that represents this path to X: Each time the path moves from a node to the left child, add a zero to the string and each time it moves to the right child, add a 1 to the string.
5. Read the characters from the input file again. For each character in the input file, write the binary encoding sequence from step 4 to the output file. Add spaces between each symbol but no newline characters, except for one at the end.

See an example run of Huffman encoding at the end of the assignment.

Your Huffman Encoder class should have a constructor and one method called encode(). The constructor should take the name of the file that should be encoded. The encode() method should open a new output file (by adding – not replacing - the ".huff" extension to the original file name), open the input file, and write the codes for the Huffman algorithm to the output file. (You should write the values as strings, with a single space separating the codes. Do not output any newline symbols, except for one at the end. Yes, the output file will be larger than the input file.)

You do **not** need to implement a decoder.

Hand-in

Submit all source code for all classes, including the unit tests. Submit all files on umlearn.

You will be provided a test file for your encoder before the deadline. Submit your output on that file with your submission.

You MUST submit all of your files in a zip file. Additionally, you MUST follow these rules:

- All of the files required for your project should be in a single directory.
- Include a README.TXT file that describes exactly how to run your code from the command line. The markers will be using these instructions exactly and if your code

does not run directly, you will lose marks. The readme file should also describe how to run your tests.

The easier it is for your assignment to mark, the more marks you are likely to get. Do yourself a favour.

Example Huffman encoding

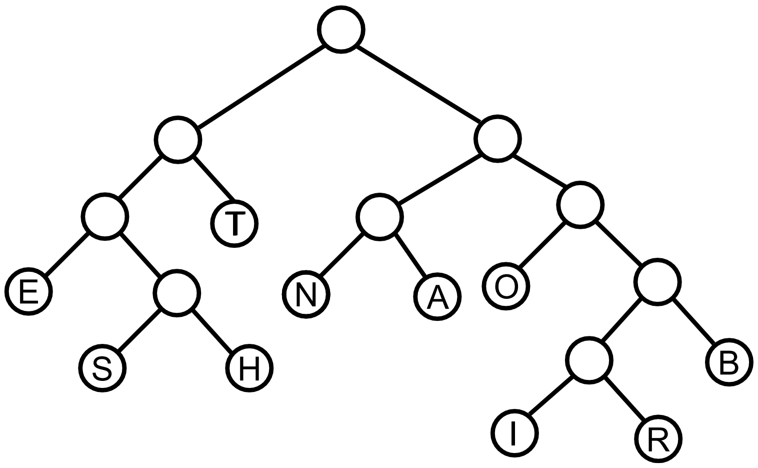
Consider the string “TOBEORNOTTOBETHATISTHEBANANA”. The Huffman algorithm first calculates the following weights:

Character	**Weight**
T	0.21428571428571427
O	0.14285714285714285
B	0.10714285714285714
E	0.10714285714285714
R	0.03571428571428571
N	0.10714285714285714
H	0.07142857142857142
A	0.14285714285714285
I	0.03571428571428571
S	0.03571428571428571

The tree joinings that occur are:

1. Join I and R into a tree (weight 0.07142857142857142)
2. Join S and H into a tree (weight 0.10714285714285714)
3. Join I,R and B into a tree (weight 0.17857142857142855)
4. Join E and S,H into a tree (weight 0.21428571428571427)
5. Join N and A into a tree (weight 0.25)
6. Join O and I,R,B into a tree (weight 0.3214285714285714)
7. Join E,S,H and T into a tree (weight 0.42857142857142855)
8. Join N,A and O,I,R,B (weight 0.5714285714285714)
9. Join N,A,O,I,R,B and E,S,H,T into a final tree.

The final tree is illustrated below.



Thus, the Huffman encoding for the file is:

T	01
E	000
S	0010
H	0011
N	100
A	101
O	110
I	11100
R	11101
B	1111