Dept. of Computer Science
Trent University

COIS 2020H: Data Structures and Algorithms

Assignment 2

Due Sunday, November 13, 2022 at 11:59pm.
A late penalty of 10% per day is assessed until Friday, November 18, 2022.

Part A: Open Hash Table Revisited (35 marks)

Problem

Re-implement the open hash table seen in class with the following modifications.

1. Each bucket has a header node.
2. The <key,value> pairs in each bucket are sorted by TKey.
3. One additional method called Output returns all <key,value> pairs in sorted order.

```
public void Output( ) { … }
```

To test your implementation, create a small class called Point whose integer data members, x and y, represent the coordinates of a point in two-dimensional space. The Point class will serve as TKey. Therefore, ensure that Point inherits from IComparable and overrides the methods GetHashCode and Equals. For the CompareTo method, assume that points are ordered first on the x-coordinate and secondarily on the y-coordinate.

Mark Breakdown for Part A

class Point	
CompareTo	3
GetHashCode	2
Equals	2
class HashTable header	1
MakeEmpty	1
Rehash	2
Insert	4
Remove	4
Retrieve	3
Output	8
Testing	5

Part B: Huffman Codes (65 marks)

Background

In 1952, David Huffman published a paper called *"A Method for the Construction of Minimum-Redundancy Codes"*. It was a seminal paper. Given a text of characters where each character occurs with a frequency greater than 0, Huffman found a way to encode each character as a unique sequence of 0's and 1's such that the overall length (number of bits) of the encoded text was minimized.

Huffman recognized that characters with a greater frequency should have shorter codes than those with a lower frequency. Therefore, to disambiguate two codes, no shorter code of 0's and 1's can serve as the prefix (beginning) of a longer code. For example, if letter 'a' has a code of 010, no other character can have a code that begins with 010.

To find these codes, a Huffman tree is constructed. The Huffman tree is a binary tree where the left edge of each node is associated with the number 0 and the right edge is associated with the number 1. All characters are stored as leaf nodes and the code associated with each character is the sequence of 0's and 1's along the path from the root to the character.

Requirements

The principle class to implement is called Huffman which tackles each of the following tasks.

Task 1: Determine the Character Frequencies
Determine the frequency of each character in the given text and store it in an array. For simplicity, only lower-case letters from 'a' to 'z', upper-case letters from 'A' to 'Z', and the blank (space) are admissible as characters (see AnalyzeText).

Task 2: Build the Huffman Tree
Using a priority queue of binary trees, a Huffman tree is constructed for the given text. Initially, the priority queue is populated with as many as 53 individual nodes (binary trees of size 1) where each node stores a character and its frequency. The priority queue is ordered by frequency where a binary tree with a lower total frequency has a higher priority than one with a higher total frequency (see Build).

Task 3: Create the Codes
The final Huffman tree is then traversed in a prefix manner such that the code for each character is stored as a string of 0s and 1s and placed in an instance of the C# Dictionary class using its associated character as the key (see CreateCodes).

Task 4: Encode
Using the dictionary of codes, a given text is converted into a string of 0s and 1s (see Encode).

Task 5: Decode
Using the Huffman tree, a given string of 0s and 1s to converted back into the original text (see Decode).

To help implement your program, consider the following skeleton of code.

```
class Node : IComparable
{
        public char     Character       { get; set; }
        public int      Frequency       { get; set; }
        public Node     Left            { get; set; }
        public Node     Right           { get; set; }

        public Node (char character, int frequency, Node left, Node right)  { … }

        // 3 marks
        public int CompareTo ( Object obj )  { … }
}

class Huffman
{
        private Node HT;                        // Huffman tree to create codes and decode text
        private Dictionary<char,string> D;      // Dictionary to encode text

        // Constructor
        public Huffman ( string S )  { … }

        // 8 marks
        // Return the frequency of each character in the given text (invoked by Huffman)
        private int[ ] AnalyzeText ( string  S )  { … }

        // 16 marks
        // Build a Huffman tree based on the character frequencies greater than 0 (invoked by Huffman)
        private void Build ( int[ ] F )
        {
                PriorityQueue<Node>  PQ; …
        }

        // 12 marks
        // Create the code of 0s and 1s for each character by traversing the Huffman tree (invoked by Huffman)
        // Store the codes in Dictionary D using the char as the key
        private void CreateCodes ( )  { … }

        // 8 marks
        // Encode the given text and return a string of 0s and 1s
        public string Encode ( string S  )  { … }

        // 8 marks
        // Decode the given string of 0s and 1s and return the original text
        public string Decode ( string S )  { … }
}
```

Mark Breakdown for Part B

CompareTo	3
Huffman methods	52
Testing	5
Documentation	5

Submission

Zip up and hand in all source code, executable files, and tests cases (screen shots) online at Blackboard. Only one partner needs to submit the assignment but do make sure that all names are on the submission.

Happy Coding!