COP 5536 Spring 2017

Programming Project

Name: Deep Chakraborty
UFID: 21151818

Contents

1. Introduction

Purpose of this project is to develop Huffman encoder and decoder, which is used for lossless data compression. Huffman encoding involves use of a priority queue, which can be implemented using various data structures like Binary Heap, 4-way cache optimized heap, and Pairing Heap. Objective of this project is to compare performances of these three data structures, and use the best one to develop the Huffman encoder.

2. Function Prototypes and Program Structure

2.1 Encoder

A decompressed input text file will be given to the Encoder as input. Function of the Encoder is to build frequency table, build Huffman tree, write code table, and finally write the encoded binary file using the generated code table.

2.1.1 Build frequency table

Objective of this portion of the code is to map the distinct inputs in the decompressed input text file with its frequency of occurring in the input file. A separate Java class, named FrequencyTableElem, has been used to store this information. This same class has also been used to store the left child and right child of each of the objects. These data will be used while Huffman tree generation, explained later in this document. Structure of this class is as follows:

public class FrequencyTableElem

<u>Field Summary:</u>

Modifier and Type	Field and Description
private int	entry The distinct input in the uncompressed input text file.
private int	freq Frequency of the *entry* in the uncompressed input text file
private FrequencyTableElem	leftchild Left child of the current object in the Huffman tree
private FrequencyTableElem	rightchild Right child of the current object in the Huffman tree

<u>Constructor Summary</u>

FrequencyTableElem (int entry, int freq)
Constructs a newly created FrequencyTableElem object corresponding to the specified entry with the specified frequency freq.

<u>Method Summary</u>

1. public FrequencyTableElem getLeftchild()
   Returns the *leftchild* field of the current FrequencyTableElem object.
2. public void setLeftchild (FrequencyTableElem leftchild)
   Sets the *leftchild* field of the current object as specified in the argument.
3. public FrequencyTableElem getRightchild()
   Returns the *rightchild* field of the current FrequencyTableElem object.

4. public void setRightchild(FrequencyTableElem rightchild)
   Sets the *rightchild* field of the current object as specified in the argument.
5. public int getEntry()
   Returns the *entry* field of the current FrequencyTableElem object.
6. public void setEntry(int entry)
   Sets the *entry* field of the current object as specified in the argument.
7. public int getFreq()
   Returns the *freq* field of the current FrequencyTableElem object.
8. public void setFreq(int freq)
   Sets the *freq* field of the current object as specified in the argument.

HashMap<Integer,FrequencyTableElem>
A java.util.HashMap with the distinct inputs in the uncompressed input text as key, and its corresponding FrequencyTableElem object is used.

Frequency table is generated by reading the uncompressed input text file line by line. Every time a new input is read, a new FrequencyTableElem object is created with frequency 1 and stored in a HashMap with the input value as the key and the newly created FrequencyTableElem object as the value. And every time a repeated input is read, the object corresponding to the read input value is pulled from the HashMap, and the *freq* field of this object is incremented.

2.1.2 Build Huffman Tree

To build the Huffman tree, all the FrequencyTableElem objects are stored in a java.util.Vector. This vector is passed to the function build_tree_using_binary_heap which returns the Huffman tree after the required sequence of operations.

private static FrequencyTableElem build_tree_using_binary_heap(Vector<FrequencyTableElem>)

This function uses the FourWayHeap class (or BinaryHeap or MinPairingHeap class) as a priority queue to build the Huffman tree. Until the Vector contains only one object, we are extracting two FrequencyTableElem objects that have the minimum frequency *freq* field. Then another FrequencyTableElem object is created with frequency as the sum of the frequencies of the two extracted objects; leftchild as the object with smaller frequency of the two extracted objects; and rightchild as the remaining extracted object. This newly created object is then inserted into the heap. Ultimately when only one object remains in the Vector, and this object represents the whole Huffman tree.

Public class FourWayHeap
Field summary:
private Vector<FrequencyTableElem> list
Stores the list of FrequencyTableElem objects that are to be formed a min heap

Constructor Summary:
public FourWayHeap(Vector<FrequencyTableElem> items)
Constructs a newly created object with the specified Vector of FrequencyTableElem set as the field *list*.

Method Summary:
public void insert(FrequencyTableElem item)
Inserts the specified item into the min heap.

public void buildHeap()
`Builds the min heap such that the min heap property is restored.`

public FrequencyTableElem extractMin()
Extracts the FrequencyTableElem object with minimum frequency.

private void minHeapify(int i)
Finds the FrequencyTableElem object with minimum frequency among the object at the specified index *i*, and its children, and then swaps the minimum.

2.1.3 Write code_table.txt

After Huffman tree is achieved, the tree is traversed using Inorder traversal, and a *label* string is appended. Whenever we traverse a left child, 0 is appended, and when we traverse a right child, 1 is appended. Whenever we get a leaf node, the *entry* field of the FrequencyTableElem object is put into a HashMap along with the *label* string.

private static void build_code_table(FrequencyTableElem huffmanTree, HashMap<Integer,String> codeTableVector, StringBuilder code)

The HashTable is traversed and both the entry field, and frequency field are written to the code_table.txt.

2.1.4 Write encoded binary file encoded.bin

Finally, we are traversing the Vector containing the inputs from the uncompressed input text file, getting the corresponding code from the above HashMap, and these two information are written to encoded.bin binary file.

2.2 Decoder

An encoder binary file and code_table.txt will be given to the Decoder as input. Function of the Decoder is to build Decode tree, write decoded.txt which should be same as the uncompressed input text file.

2.2.1 Build Decode Tree

The code_table.txt is parsed, and every code in the form of binary string is read. While parsing this code, when we get a 0, we create a new node to the left of current node (by creating a new FrequencyTableElem object), and when we get 1, we create a new node to the right of current node.

2.2.2 Decode the binary file and write decoded.txt

We parse the binary file encoded.bin byte by byte, and correspondingly traverse the decode tree starting from the root. When we get a 0, we traverse to left of current node, and when we get a 1, we traverse to the right of current node. When we hit a leaf node, then we write the *entry* field of this node to decoded.txt, and update the current node to the root node.

2.2.3 Complexity of Decoder

Complexity of building the decode tree would be O(n*m),
where, *n* is the number of distinct input symbols present in the code_table.txt file, and *m* is *length of largest code corresponding to a particular input value in the code_table.txt file.

Complexity of writing the binary file would be O(n),
where n is the number of bits in the encoded bin file.

3 Performance Analysis

Performance of different data structures

Data structure	Average time in millisecond for sample_input_large.txt for 10 runs
Binary Heap	4157.2
4-Way cache optimised heap	3787.7
Min Pairing Heap	23304.8

As can be seen from above table, 4-way cache optimised heap outperformed the other two data structures. This is because since 4-way heap nodes have four children, it takes less number of traversals to reach the root after min heap generation and insertion of new element. Also 4-way cache optimised heap uses a cache optimisation technique by shifting the elements by three positions so that all the children of a node can be accessed fast in cache-optimized manner.