Advanced Data Structure Project
Spring 2017
COP5536

Name : Ankita Bose
UF id : 38933777

Introduction

In this project, we were assigned to implement Huffman Encoding and Decoding. Our knowledge of different data structures was tested as we were asked to test 3 different data structures for implementing the priority queue, and select the one that gives the best performance.
We implement the following priority queue structure during the course of a month, namely Binary Heap, 4-way cache optimized heap, pairing heap.
Upon testing the data structures for both small input files and large input files, the priority queue implementation that worked the fastest for me was Binary Heap.

Structure of Program

There are 2 main classes Encoder and Decoder. They in-turn call other classes to perform the Huffman Encoding and Decoding respectively.

❖ encoder.java

- **Input -**

  We need to provide the file to be encoded in command line itself in the following way :

  java encoder ./sample1/sample_input.small.txt
  or,
  java encoder ./sample2/sample_input_large.txt

- **Output -**

  Following were the output of the program

  1. encoded.bin – The encoded data (which is a binary file since the data after Huffman Encoding is in the form of 1's and 0's)
  2. code_table.txt – This file shows the generated Huffman code for each number in the input file.

- **Exceptions** –

  The encoded.java handles 2 kinds of exception.

  1. IOException – This results when the input file is not present.
  2. NumberFormatException – Since we are generating the Huffman code for only numbers in the input file, this exception results if the given file contains anything other than numbers.

- **Member Variables –**

  1. int freqTable[100000000] ;
     - Stores the frequency of each number in the range of (0-999999)

2. Huffman huffman ;
    - Object of the class Huffman

- **Functions –**

1. public static void main(String[]) - Program execution begins here. The input file is read. A frequency table is created for counting the occurrence of each number, which is in turn used to create the Huffman tree. Furthermore, we generate the code for each element and create a file that displays the element and its code.

2. public void createFreqTable(ArrayList<Integer>) – This function does the aforementioned task of creating a frequency table based on the occurrence of every element. Now in the frequency table (freqTable[]), each index stores the occurrence of index value in the input file.

3. public void createCodeTable(String[]) – This function simply writes in the file code_table.txt. The string array that is the argument to this function, stores the code for an item in the index equal to the item.

4. public void encoding(String[], ArrayList<Integer>) – This function generates the actual encoded file. For every element in the input file, it gets the Huffman code for that value, converts it to binary and writes the result in the file encoded.bin.

- **<u>Huffman.java</u>**

- **Functions –**

1. public Node createHuffmanTree(int[]) – This function creates nodes with their frequencies from the frequency table generated before, and adds the nodes to the priority queue. In my code, I have added the nodes to a Binary Heap. After initialization of the priority queue, two nodes with the minimum frequency is deleted from the queue, and a new node with their combined frequency is added to the end of the queue. This process is continued until the priority queue contains only a single node.

2. public void generateCode(String[], Node, String) – This is the function where the Huffman Code is generated from the Huffman Tree. This is a recursive function, where we start at the root and append '0' to the code string, each time we enter the left subtree and append '1' each time we enter a right subtree (Huffman Code logic).

- **<u>BinaryHeap.java</u>**

- **Member Variables –**

1. T[] key ;

- key represents an element in the priority queue. <T> is a template that allows the priority queue to store any type of elements

2. int size ;
    - the total number of elements in the priority queue
3. Comparator<T> compareObj;
    - Object of the type comparator that allows us to compare elements of type <T>

- **Functions –**

1. public void insert(T) – A node of type T is inserted into the priority queue. Each time the queue gets full, we increase the size by two times.

2. public T delmin() – The minimum element, which is at the root is returned and deleted from the tree. It gets replaced by the last element, and the size of the heap reduces by 1. After which we min heap is generated again.

3. public int getSize() – Returns the 'size' member variable.

4. private void upTraversal(int i) – From the index 'i', it traverses up to the root to check where the min heap property is not satisfied.

5. private void downTraversal(int i) – From the index 'i', the program traverses down to the last element to check where the min heap property is not satisfied.

6. private void swap(int x,int y) – Swaps the keys in index x and y.

7. private boolean checkMinHeap(int n) – Checks whether the binary heap is a min heap or not.

8. private boolean minHeapCheck – Calls checkMinHeap() from the root.

9. private boolean isBigger (int i, int j) – Returns true if node in index 'i' is greater than node in index 'j'.

10. private boolean isEmpty() – Checks whether the heap is empty.

- **<u>Node.java</u>**

- **Member Variables –**

1. int value;
    - The data or the number.
2. int frequency ;
    - The occurrence of the 'value' in the input file.

3. Node left;
    - Left pointer.
4. Node right;
    - Right pointer.

- **Functions –**
1. public int compareTo(Node) – Returns the difference between the frequencies of the current value and the frequency of the argument node.
2. public boolean isLeaf() – Returns 'true' if both the left and right pointer of the node is null.

❖ **<u>BinaryConversion.java</u>**

- **Member Variables –**
1. BufferedOutputStream bout;
    - Output stream used to print the output in the file.
2. int buffer;
    - 8-bit buffer of bits.
3. int n;
    - number of available bits in the buffer

- **Functions –**
1. private void writeBit(boolean) – Writes 1 bit to the buffer. When the buffer is full it calls clearBuffer().
2. private void clearBuffer() – This function writes to the file when the buffer is full and empties the buffer.

❖ **<u>decoder.java</u>**

- **Input -**

We need to provide the encoded file and the code table as command line arguments. We can do that in the following way :

java decoder encoded.bin code_table.txt

- **Output –**

The output of the program would be the decoded text file in the form of 'decoded.txt'

- **Member Variables –**
1. public class Node{} - It has a Node which stores the values, and has a left and right pointer

- **Functions –**

1. public static void main(String[]) – The program creates a Huffman Tree from the code table, after which it reads the encoded file and decodes it using the Huffman tree. Finally, it writes the decoded text in the file.
2. public Node createHuffmanTree(String) - In this function the Huffman tree is created from the code table.
3. private String readCodeFile(String) – Reads the encoded file.
4. private void writeFile(Node, String) - Uses the encoded string and the Huffman tree to decode it and write it in a file

Performance Analysis

We implemented three different priority queues and measured the performance both in our system and on the UF server.

Here is a screenshot of the performance of the three priority queue on thunder.cise.ufl.edu server. The time is in milliseconds.

```
thunderx:20% java Encoder ./sample2/sample_input_large.txt
Time :683
thunderx:21% java EncoderMulti ./sample2/sample_input_large.txt
Time: 1070
thunderx:22% java EncoderPair ./sample2/sample_input_large.txt
Time: 1107
thunderx:23% java Encoder ./sample2/sample_input_large.txt
Time :679
thunderx:24% java EncoderMulti ./sample2/sample_input_large.txt
Time: 1145
thunderx:25% java EncoderPair ./sample2/sample_input_large.txt
Time: 1105
thunderx:26% java Encoder ./sample2/sample_input_large.txt
Time :702
thunderx:27% java EncoderMulti ./sample2/sample_input_large.txt
Time: 1088
thunderx:28% java EncoderPair ./sample2/sample_input_large.txt
Time: 1171
thunderx:29% java Encoder ./sample2/sample_input_large.txt
Time :760
thunderx:30% java EncoderMulti ./sample2/sample_input_large.txt
Time: 1138
thunderx:31% java EncoderPair ./sample2/sample_input_large.txt
Time: 1188
thunderx:32% java Encoder ./sample2/sample_input_large.txt
Time :806
thunderx:33% java EncoderMulti ./sample2/sample_input_large.txt
Time: 1098
thunderx:34% java EncoderPair ./sample2/sample_input_large.txt
Time: 1213
```

Figure 1: Snapshot of the execution of different priority queue

Average runtime for encoding

Method	Time in seconds
Binary Heap	0.732
4-way Heap	1.112
Pairing Heap	1.256

Conclusion –

From the above result, it was clear that Binary Heap gave the best performance among all three. Between 4-way heap and Pairing heap, 4 way heap mostly performed better although there were some instances when Pairing Heap defeated 4-way marginally by giving better performance.

However, we must acknowledge that there are several other factors that play a significant role in determining the execution time of each data structure. The performance of every system varies, and so does the traffic on a server, and therefore, we must realize that this could be the reason of possible anomalies. From the results I got, I would like to conclude that Binary Heap performs the best, followed by 4-way heap, and then Pairing Heap.

Decoding Algorithm

The decoding algorithm used goes as follows -

- Step 1:
  - Build a Huffman Tree for all elements from the code table that was generated during encoding.
- Step 2:
  - Read the encoded file.
- Step 3:
  - For every 8 bits, generate a string and decode it using the Huffman tree.
- Step 4:
  - Write the decoded value in a file.
- Step 5 :

  Repeat steps 2- 4 until the end of the encoded file is reached.

Complexity of the Decoding Algorithm

There are n bits in the encoded text and each bit traverses one branch in the Huffman Tree. Consequently, the runtime of the decoder would be n bits, or O(n).