Advanced Data Structures (COP 5536)

Spring 2017

Programming Project Report

Name: Purnendu Mukherjee
UFID: 68016371
purnendu@ufl.edu

Introduction to the problem

MyTube needs to send massive amounts of data to Toggle server. To compress the data before sending to Toggle, MyTube decided to use Huffman Coding. I implemented the Huffman coding for this process as follows:

Function Prototypes and Structure of the program

Generate Frequency Table:

public HashMap<Integer,Integer> buildFreTable(String path)

At first, we need to count the frequency of each of the occurrences of the numbers/strings in each line. For this purpose I used a HashMap to store the actual string in the key and its frequency in the input file as the value for that key. This is done in the file readfile.java.

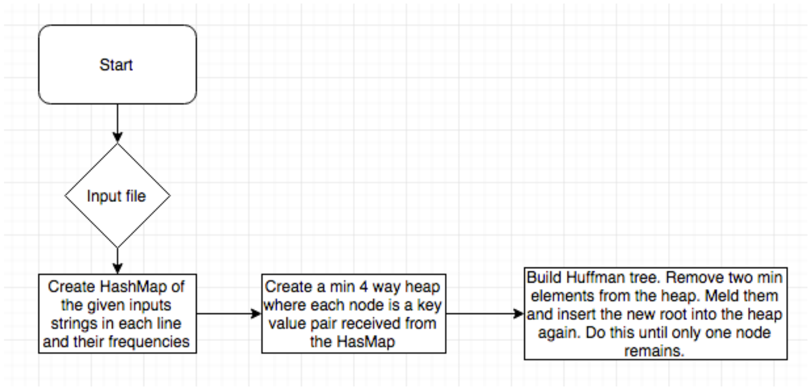
Create the 4-way Heap

public void buildtreef(HashMap<Integer,Integer> freqTable)
public fourWayHeap(int capacity)

/** Function to insert element */
public void insert(Node x)
private void heapifyUp(int childInd)
HeapifyUp is carried out after each insert.

/** Function to delete min element **/
public Node deleteM()
private void heapifyDown(int ind)
HeapifyDown is carried out after each removeMin

From the frequency table (HashMap) created in the above step we need to build the the Heap. After doing performance analysis on Binary Heap, 4-way Heap and Pairing Heap, I found Cache optimized 4-way heap to be the fastest for building the Huffman tree. Thus we are going to create the 4-way Heap now. We first create each <key,value> pair in the HashMap as a Node and then insert this node in the heap. The Node class (Node.java) is used for this. We insert all the values in the HashMap in the heap. The heap is a min heap and thus the root is the minimum element in the heap. We need to use min heap as we are going to use the removeMin operation to build the Huffman tree. Once all the nodes are inserted in the Heap. We remove 2 nodes and meld them. This generates a new root node for the two melded trees/nodes. We then insert this rootnode into the heap. We repeat this process till there is only 1 node left in the heap. This the rootnode of the Huffman tree.


Start
Input file
Create HashMap of the given inputs strings in each line and their frequencies
Create a min 4 way heap where each node is a key value pair received from the HasMap
Build Huffman tree. Remove two min elements from the heap. Meld them and insert the new root into the heap again. Do this until only one node remains.

Build the code table

public void buildCodeTable(String path)
public void trav(Node node, String str, String num)

From the above step we have generated the Huffman tree. Next we need to generate the code table where each string (from the input file) will be associated with a binary code. To generate this we need to traverse the whole Huffman tree. Each of the unique values from the input file is now present in the leaf node of the Huffman tree. The ones with high frequency will be close to the root whereas the strings appears few times will be farthest from the root. We obtain the code for each of the unique strings by traversing from the root to that node. For each string we write the string and its corresponding code into the file code_table.txt

Encoding the file -

public void encoder(String path)

Now that we have the code table, we need to encode the given input file. For every new number/word in each line we have its associated code from the code table. For each word/number in the input file we write its code into a newfile. We do this for all the words/numbers in the input file while appending the codes one after the other. Since the codes are in 1s and 0s for every 8bit we convert it into byte and then write into the file encoded.bin. This encoded.bin file along with the code_table.txt is sent to the receiver.

Decoding Process

public void buildTree(String codeTable1)

public void decodeData(String encoded1)

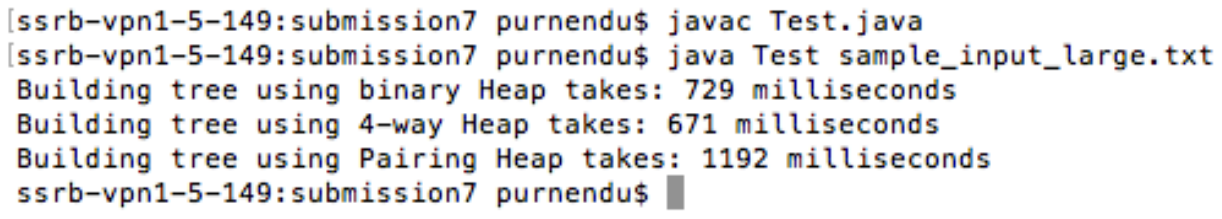
The receiver receives the two files encoded.bin and code_table.txt from the sender. The receiver now needs to decode the encoded file using the code table and reconstruct the original input file.

To do this the code table is used to create a binary Huffman tree. During the encoding process traversing to the left child from a node was denoted as a 0 and 1 for traversing to the right child. From the code table we recreate the binary tree using this policy. The leaf nodes are assigned the values for their corresponding codes.

The encoded.bin file is parsed using this binary tree. The byte values are converted to binary code. We keep travelling down the tree as per the binary input and as we reach a leafnode we take its value and store it in the decoded.txt file. The parsed binary values are truncated from the buffer and the parsing starts again from the root of the tree. We do this process till all of the code in encoded.bin is parsed and we have reconstructed the original input file as decoded.txt.

Performance Analysis, Results and Explanation:

Before generating the code table for the encoding process, one of our goal was to test the data structure which lets us build the Huffman tree in least amount of time.



```
[ssrb-vpn1-5-149:submission7 purnendu$ javac Test.java
[ssrb-vpn1-5-149:submission7 purnendu$ java Test sample_input_large.txt
 Building tree using binary Heap takes: 729 milliseconds
 Building tree using 4-way Heap takes: 671 milliseconds
 Building tree using Pairing Heap takes: 1192 milliseconds
 ssrb-vpn1-5-149:submission7 purnendu$ 
```

After implementing the 3 heaps and taking their average time to build the Huffman tree we found the following:

Heap	Complexity		Time Taken (Average)
	Insert	RemoveMin	
Binary Heap	O($\log_2 n$)	O($\log_2 n$)	729 ms
4-way Heap – Cache optimized	O($\log_4 n$)	O($4\log_4 n$)	677 ms
Pairing Heap	O(1)	O($\log_2 n$)(Amortized)	988 ms

While theoretically Pairing Heap should take the least time as it has the least Amortized complexity for insert and remove, it also depends on the implementation of the whole thing. The amortized case for Pairing Heap is similar to the amortized case for the other two algorithms. There is almost an equal number of insert and removeMin operation for building the Huffman tree hence the actual cost are close for all the 3 heaps and 4-way Heap runs in best time.

Cache optimized 4way heap: To cache optimize the 4way Heap, we left the first 3 places in the Node array as it is and started putting in nodes from the 4th position in the array of Nodes.

Decoding Algorithm used and its complexity

Decoding Algorithm

During the encoding process traversing to the left child from a node was denoted as a 0 and 1 for traversing to the right child. From the code table we recreate the binary tree using this policy. The leaf nodes are assigned the values for their corresponding codes. The encoded.bin file is parsed using this binary tree. The byte values are converted to binary code. We keep travelling down the tree as per the binary input and as we reach a leafnode we take its value and store it in the decoded.txt file. The parsed binary values are truncated from the buffer and the parsing starts again from the root of the tree.

Decoding Complexity
The time taken to build the Huffman tree from the code table is Linear in the sum of length of all the bit codes in the Code Table

Thus, the time taken to construct the Huffman tree is:

$$O(N*m)$$

where N is the size of the Code Table and m is the maximum length of a code in the Code table.

To decode the data we need to traverse the reconstructed Huffman tree. For each bit in the encoded.bin file we need to traverse from one node to another. Thus the complexity is linear in the number of bits in the encoded.bin file.