**Anthony Kirkland & Michael Coffey**

**4/5/2019**

**HW4 SLC Report**

**Phase 1: Specification**

Using a priority queue and a binary tree of your design to implement a file compression/decompression algorithm using a modified Huffman Coding approach.

**Phase 2: Design**

**2.1 Modules and Basic Structure:**

My program has 7 modules

1. Module 1: Class Compressor
   1. Contains constructor that takes filenames and what to label left and right child nodes
   2. Fields:
      1. 2 strings for filenames
      2. 2 hashmaps one for frequency and the other for key
      3. A priority queue
      4. An arraylist of characters in order
      5. A Huffman tree
      6. And a long array for times
   3. countLetters(): reads file character by character and keeps track of and updates frequency
   4. setPriority(): puts all items in hashmap in priority queue to be sorted by frequency
   5. createCompressedFile(): using key to encode text file and write them to another
   6. printKey(): prints encodings nicely
   7. printFreq(): prints character and its frequency neatly
   8. getHuffmanTree(): getter for key
   9. getTime(): getter for times
2. Module 2: Class Decompressor
   1. Contains constructor that takes Huffman tree and file names
   2. Fields:
      1. Hashmap for key
      2. Long for timing
      3. 2 strings used for filenames
   3. Decomp(): reads compressed file “bit for bit” and decodes it using hashmap and writes it to said file
   4. getTime(): getter for time
3. Module 3: class Huffman
   1. Contains constructor that takes priority queue and what to label left and right child nodes
   2. Fields:
      1. Priority queue
      2. 2 hashmaps one for encodeing key and the other for decoding key
      3. Huffman node for tree root
   3. createTree(): uses priority queue to create Huffman tree
   4. makeKeys(): using the huffman tree, it creates the keys for encoding and decoding
   5. getEncodeKey(): getter for encoding key
   6. getDecodeKey(): getter for decoding key
4. module 4: Class HuffmanNode
   1. contains two constructors one is empty the other takes a character and its frequency
   2. fields:
      1. character
      2. int frequency
      3. 2 huffmanNodes for left and right tree
   3. getChar(): getter for char
   4. getFreq():getter for frequency
   5. getLeft():getter for left child
   6. getRight():getter for right child
   7. setChar():setter for char
   8. setFreq():setter for frequency
   9. setLeft():setter for left child
   10. setRight():setter for right child
5. module 5: class Main
   1. fields:
      1. strings for file names
      2. compressor
      3. decompressor
      4. long and long array for times
      5. scanner to read user input
      6. fileOutputStream to write to file
   2. main(): driver for application
   3. getInput(): gets user input and writes randomly to txt file
6. module 6: class Node
   1. contains 2 constructors one that takes a HuffmanNode,and another that takes that and its next node
   2. fields:
      1. Huffman node
      2. Node
   3. getData(): returns data in node
   4. getValue(): returns frequency
   5. getNext(): gets next node
   6. setNext(): sets next node
7. module 7: Class PriorityQueue
   1. fields:
      1. node head
      2. node tail
      3. int for size of queue
   2. isEmpty(): returns whether or not the queue is empty
   3. enqueue(): add the node to the queue and places it in the correct priority
   4. dequeue(): dequeues item with the least frequency
   5. size(): returns the size of the queue

**Phase 3: Risk**

There is no risk.

**Phase 4: Verification**

The implementation has been tested thoroughly.

**Phase 6: Testing**

Test cases of size n=30, 60, 100 have all been tested by me and they all seem to function properly.

**Phase 7: Refining The Program**

Using a hashmap instead of traversing the tree everytime saves a lot of time during decompression.

**Phase 8: Production**

A zip file with the required output format and a javadocs have been provided

electronically using elearning’s dropbox.

**Phase 9: Maintenance**

I use a lot of hashmaps for keys and frequencys. This saves time but uses more space. This could be changed if space was more of a problem than time.

**Phase 10: Graphs and Comparison**

**theoretical vs. empirical**

finding the Huffman coding information takes O(n) time and space.

Encoding and decoding takes O(n) time.

Below are time averages and graphs:

 

 

 