Huffman Coding and Decoding

Tianyu Zhang

(UFID: [redacted])

I. Introduction

The object of this project is to implement Huffman coding to compress large input data to a relatively smaller encoded binary file. Huffman coding is one of the most commonly used strategies for reducing data size by representing each data based on its frequency. The implementation of Huffman coding in this project is mainly divided into two parts: encoding and decoding. In the encoding part, the frequency of each data in the large input file is counted first. Then it is used for constructing Huffman Tree and generating the compressed output and a code table. In the decoding part, Huffman Tree data structure is reconstructed with the code table. The compressed binary file is decoded by traversing the corresponding path of the Huffman Tree. This process results in a decoded file, which is identical to the original input.

In addition, the performance of binary heap, four-way heap, and pairing heap are tested for generating Huffman Tree. This report will present program structure, function prototypes, min-heap performance, encoding and decoding algorithm, and complexity analysis.

II. Program Structure

In total, 7 java files were composed in this project. The ***HuffmanNode.java*** defies the data structure of Huffman tree nodes. ***HuffmanTree.java*** generates Huffman trees with binary heap, four-way heap, and pairing heap, respectively. It also measures the performance of each heap in microseconds. ***encoder.java*** and ***decoder.java*** contain the main method, which can be used to compress and decompress the input data. ***BinaryHeap.java***, ***FourWayHeap.java***, and ***PairingHeap.java*** are heap implementations, which can be called when necessary.

Working Environment

- **Language**: Java openjdk version "11.0.9.1" 2020-11-04
- **Runtime Environment**: OpenJDK Runtime Environment (build 11.0.9.1+1-Ubuntu-0ubuntu1.20.04)
  OpenJDK 64-Bit Server VM (build 11.0.9.1+1-Ubuntu-0ubuntu1.20.04, mixed mode, sharing)
- **IDE**: IntelliJ IDEA 2020.2.3 Build #IU-202.7660.26, built on October 6, 2020

Compiling Procedure

All java files and a makefile are included in the zip file. Under Linux environment, all java files can be compiled by executing the makefile.

$ make
The make command compiles the ***encoder.java, decoder.java*** and their corresponding class files with javac compiler. The following commands can be used to run encoder class and decoder class:

$ java encoder <input_file_name>
A text file, ***code_table.txt***, and a binary file, ***encoded.bin***, will be generated after executing the above command. The ***code_table.ext*** contains unique Huffman code for each input entry. The encoded file contains converted input information, which has a relatively smaller file size.

$ java decoder <encoded_file > code_table_file >
This command retrieves the original input file from the encoded file and code table. A text file named "decoded" will be generated. The file size and contents of the newly generated file is identical to the original input file.

III. Function Prototypes

HuffmanNode.java

This class contains the Huffman node data structure. The Huffman node is involved in all processes, such as generating Huffman trees, constructing different types of heaps, and encoding / decoding the input file.

int data – stores the input data
int freq – stores the frequency of each data
HuffmanNode left – point to the left child of this node
HuffmanNode right – point to the right of this node

HuffmanNode(int data, int freq, HuffmanNode left, HuffmanNode right)

This is the constructors of Huffman nodes. It receives data, frequency of the data, pointers to left and right children as variables.

int compareTo (HuffmanNode that)

compare two Huffman nodes based on their frequency. Return negative number if less than; positive number if larger than; return 0 if equal.

boolean isLeaf()

return true if the node is located at leaf, in which condition, the node should point it's both children to null.

BinaryHeap.java

HuffmanNode[] heap – Define an array that consists of Huffman nodes. The array is used to store the minimum binary heap.
int heapSize – Define an integer that stores how many Huffman that inserted into the heap.

BinaryHeap(int capacity)

Constructor of minimum binary heap. It initializes the heap size to zero and claims an

array length of capacity + 1 for storing the Huffman node.

boolean isEmpty()

This method returns the heap size if zero.

boolean isFull()k

This method returns if the heap size has reached the maximum length of the heap.

int parent(int i)

Calculating and returning the index of ith node's parent node.

int kthChild(int i, int k)

Calculating and returning the index of ith node's kth child. The variable k could be 1 or 2.

int size()

returns the heap size.

void insert(HuffmanNode x)

When invoking this method, the Huffman node x will be placed at the end of all pre-existing nodes. Then we invoke the heapfyUp method to replace the Huffman node x to its correct position in the array heap.

HuffmanNode removeMin(int x)

We use this method for x equals to zero only. This method pops the first element in the heap, which is the minimum element. After this operation, the method fills the empty first position with the last element. Since the refilled element is not less than the second element, we apply the heapifyDown method to the element on heap top. As a result, the root of the adjusted binary will be returned. The value of heap size is also deducted with 1.

void heapifyUp(int i)

In this method, we try to find the parent node of i, and compare those two nodes. If the ith element smaller than its parent, then we swap them. In this way, we can make sure all parent nodes are larger than their children. The min-heap property is maintained.

void heapifyDown(int i)

In this method, we first invoke the minChild method to get the minimum child of the ith element. If the element is larger than its minimum child, we swap those two nodes. Otherwise, we check the minimum child's children. Repeatedly get the minimum child and compare it with the ith element, swap if the ith element is larger, until to the end of the heap array.

int minChild(int i)

This method invokes the kthChild method to get ith element's first and second child. Then it returns the index of the smaller one of those two nodes.

FourWayHeap.java

HuffmanNode[] heap – Define an array that consists of Huffman nodes. The array is used to store the minimum four-way heap. The four-way heap is catch optimized by adding three dummy

int heapSize – Define an integer that stores how many Huffman that inserted into the heap.

FourWayHeap(int capacity)

The constructor of a four-way minimum heap. It initializes the heap size to three and claims an

elements in the first three positions of the array. Thus, we can obtain the minimum element from the fourth position.

array length of the inputted capacity for storing the Huffman node.

boolean isEmpty()

This method returns the heap size of three. The three are the size of a heap only containing dummy nodes.

boolean isFull()

This method returns if the heap size has reached the maximum length of the heap.

int parent(int i)

Return the index of ith node's parent node.

int kthChild(int i, int k)

Return the index of ith node's kth child.

int size()

returns the heap size, which is the number of inputted elements.

void insert (HuffmanNode x)

When invoking this method, the Huffman node x will be placed at the end of all pre-existing nodes. Then we invoke the heapfyUp method to replace the Huffman node x to its correct position in the array heap.

HuffmanNode removeMin(int i)

We use this method for i equals to three only. This method pops the third element in the heap, which is the minimum element. After this operation, the method fills the empty first position with the last element. Since the refilled element is not less than the second element, we apply the heapifyDown method to the element on heap top. As a result, the root of the adjusted binary will be returned. The value of heap size is also deducted with 1.

void heapifyUp(int i)

In this method, we try to find the parent node of i, and compare those two nodes. If the ith element smaller than its parent, then we swap them. In this way, we can make sure all parent nodes are larger than their children. The min-heap property is maintained.

void heapifyDown(int i)

In this method, we first invoke the minChild method to get the minimum child of the ith element. If the element is larger than its minimum child, we swap those two nodes. Otherwise, we check the minimum child's children. Repeatedly get the minimum child and compare it with the ith element, swap if the ith element is larger, until to the end of the heap array.

int minChild(int i)

This method invokes the kthChild method to get all four ith element's child. Then it returns the index of the smallest one of those nodes.

PairingHeap.java

PairNode root – representing the root of Pairing heap.
PairNode[] treeArray – Define an array that used to store pair nodes. The initial length of the array is set to 2.
int heapSize – Store the number of elements that are in the heap so far.

Class PairingNode()

This is a class that defines the data structure of a pair node. A pair node contains a Huffman node, pointers to its left child, next sibling and to its parent. The pair node is initialized by receiving a Huffman node from another method. Its three pointers are set to null in the first.

PairingHeap()

The constructor that set root to null and initialize heap size to zero.

boolean isEmpty()

This method returns the heap size if zero.

PairNode insert (HuffmanNode x)

Created a new pair node with the Huffman node x. if there is no node in the heap now, we point root to this node. Otherwise, we invoke the

compareAndLink method to link this new node to the current root. Since we add a new node to the heap, the heap size increases by one.

PairNode compareAndLink(PairNode first, PairNode second)

This method compares the two inputted pair nodes and links the larger one as the leftmost child of the smaller one. The method returns the new root of the combined heap.

PairNode[] doubleIfFull(PairNode[] array, int index)

This method doubles the array size when the array gets full of pair nodes. It first creates a new array with double size. Then copy all elements from the old array to the new array.

public HuffmanNode deleteMin()

This method deletes and returns the minimum Huffman node (which is the node) from the heap. If the root is the only element in the heap, set the root to null. If the root has children, perform the combineSiblings to combine all the children to form a new heap. After deleting the minimum element, the heap size decreases by one.

PairNode combineSiblings(PairNode firstSibling)

This method combines all sibling nodes to form on pairing heap. It first check if the first sibling pair node has any siblings. If not, the pair node should be root of the heap, return this node. Otherwise, we store the subtrees in an array. Then repeatedly combine the subtrees in two-path scheme from left to right. Finally, return the first node in the array, which is root of the heap.

HuffmanTree.java

String inputFile – Record the file path of the input.

void compress(int type)

This method reads the input file into the memory and counts each data's frequency while reading. Each data and its corresponding frequency are stored in a hashmap named input. The Method then uses a selected min-heap to build the Huffman tree. The min-heap is designated by the incoming variable. The time lapse was counted by taking an average of 10 repetitions of building Huffman tree step.

HuffmanNode buildTrieBinaryHeap (Map<Integer,Integer> input)

This method builds Huffman trees with binary heap. It applies a greedy algorithm by combining two smallest subtrees into one. Then it inserts the combined subtree into heap for another loop. It returns the root of Huffman tree by invoke removeMin method in BinaryHeap class.

HuffmanNode buildTrieFourWayHeap (Map<Integer,Integer> input)

This method builds Huffman trees with a 4-way heap. It applies a greedy algorithm by combining two smallest subtrees into one. Then it inserts the combined subtree into heap for another loop.

HuffmanNode buildTriePairingHeap (Map<Integer,Integer> input)

This method builds Huffman trees with pairing heap. It applies a greedy algorithm by combining two smallest subtrees into one. Then it inserts the combined subtree into heap for another loop.

encoder.java

String inputFile – stores the input file path.

void compress()

This method reads the input file into the memory and counts each data's frequency while reading. Each data and its corresponding frequency are stored in a hashmap named input. The Method then uses a 4-way heap to build the Huffman tree. Then the method uses buildCodeTable method to construct a hashmap with data and its Huffman code, and uses outputCodeTable method to output the table to a text file. After this, it calls outputEncodedFile method to convert the input data and output it as a binary file with Huffman tree.

HuffmanNode buildTrie_dAryHeap (Map<Integer,Integer> input)

This method builds Huffman trees with a 4-way heap. It applies a greedy algorithm by combining two smallest subtrees into one. Then it inserts the combined subtree into heap for another loop.

void buildCodeTable (Map<String, String> codeTable, HuffmanNode x, String path)

The code table is constructed to traverse the Huffman from root. This process is implemented recursively. If a node is not a leaf node, we traverse its left and right child respectively. In the meantime, we build up Huffman code by adding 0 for going to the left and 1 for right. If a leaf node has been reached, we record the input data in the leaf and the Huffman code into a code table.

void outputCodeTable(Map<String, String> codeTable)

This method traverses the key set of code tables, outputs the key value and its corresponding Huffman code to a text file.

void outputEncodedFile(Map<String, String> codeTable)

This method first reads the input data and converts it to Huffman code with a code table. Since the Huffman code is stored as integer for now, it needs to be transformed to binary format. To achieve this, we first convert the Huffman code from integer to a char array. Then we traverse the char array in 8- character increments. For each 8-character interval we use a buffer to store its corresponding binary information. From the left to right, if the character is '1', we add one to the buffer and shift its bit to left by 1. Otherwise, we do nothing and shift its bit to the left by 1. After finishing one 8-character interval, we reset the buffer to 0 and start to process the next interval. The information in the buffer is written in the output file.

decoder.java

String binPath – stores the encoded binary file's path.
String codeTablePath – stores the code table's path.

void expand()

This method first reads the code table and converts it into a hash map with Huffman code as the key and the data as the value. Then it invokes the buildHTree method to construct a Huffman tree. After this, it calls writeDecodedFile method to output the decoded file.

HuffmanNode buildHTree(Map<String, String> codeTable)

This method builds the Huffman Tree with a top-down depth first approach. It first creates a null root node and uses a pointer to point to the root node. After converting each Huffman code to a char array, it builds a new node as the left child

or right child of the parent node based on the value of Huffman code.

void writeDecodedFile (HuffmanNode root)

This method reads the encoded binary file and converts it to a byte array. Then it transforms the byte array to strings that are composed with "0" and "1". By using this string, we can repeatedly traverse the Huffman tree. Once we reach a leaf node, we write its value to the output file and start at root until the end of the string.

Performance Measurement Results

The performance of binary heap, 4-way cached heap and pairing heap has been tested for building Huffman tree on CISE thunder server. For each test run, the Huffman tree is constructed for 10 times with selected heap and the average time is computed in microsecond.

Results: The 4-way cached heap demonstrated the best performance in terms of execution time over five test runs. The binary heap has the second fast performance in this study. The pairing heap shows the worst results in all runs which have execution time almost as twice as the 4-way cached heap.

"sample_input_large.txt" was used as the input file. The performance comparison of each heap is shown in Figure 1. The exact time of each result can be found in Table 1.

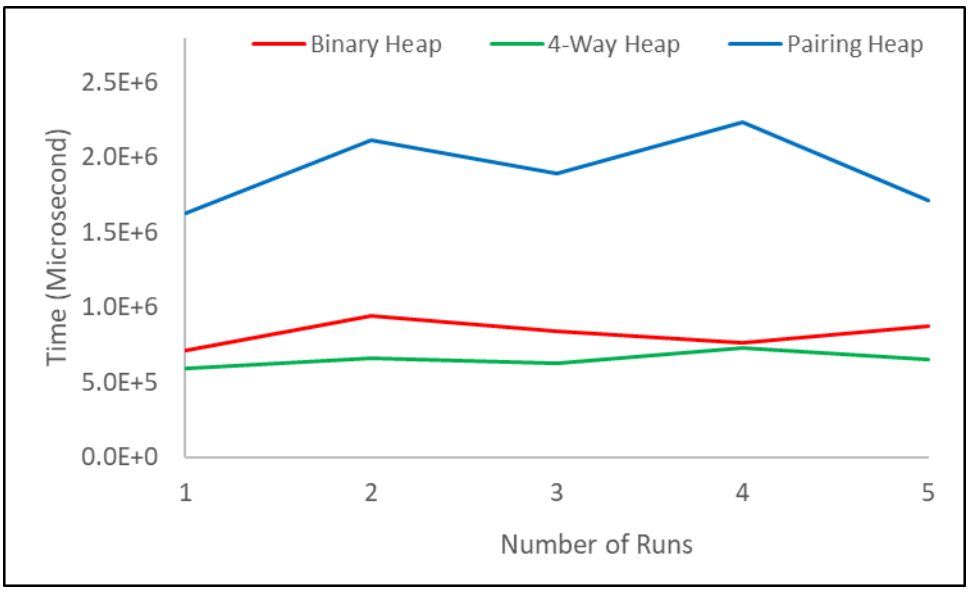

Figure 1 Performance comparison of binary heap, 4-way cached heap and pairing heap.

Table 1 The average time for constructing Huffman tree using binary heaps, 4-way cached heaps and pairing heaps.

Test Run #	Heap Types (Microsecond)		
	Binary Heap	4-Way Heap	Pairing Heap
1	710944	588325	1626526
2	939644	657684	2116509
3	840898	626854	1895593
4	761132	728535	2238678
5	875763	648771	1715543

Explanation:

The both d-ary heaps (binary heap and 4-way heap) are faster than pairing heap because in the process of building a Huffman tree, there is no Decrease-Key operation involved. As Larkin et al. mentioned in their paper, when the operation sequence contains Decrease-Key, the performance of d-ary heap is adversely affected due to its long sifting process and L2 cache miss[1]. Also, compared with other heaps, the cache allocation for pairing heaps could be used in a non-optimized way.

As for the performance difference of 4-way heap and binary heap, it is obvious that the 4-way heap has a smaller number of cache misses than the other one. Since the parent and child nodes are likely available in the same cache, the 4-way heap avoids spending extra time to extract data from lower level cache, resulting in faster performance than the binary heap. In addition, the 4-way heap has smaller tree height than binary heap, which results in less time spent traversing the tree.

Decoding Algorithm

1. The program reads the code table from a text file and saves it in a hash map. It splits each entry by space and saves Huffman code as key and data as value.
2. Using the hash map to build a Huffman tree
    a. Create a null Huffman node as root. Create a pointer point to the root.
    b. For each entry pair in the hash map, we convert its Huffman code to a char array.
    c. Repeatedly traverse each char array, adding nodes to the left side of the pointed node if meet a “0”; to the right side of the pointed node if meet a “1”. Move the pointer to the newly added node.
    d. When finished traverse one char array, reset the pointer to point to the root, perform (b.) until all entry pairs in the hashmap have been finished.
3. Decode the “encoded.bin” file with Huffman tree and save its result into a text file.
    a. Create a byte array that stores all information of the “encoded.bin” file.
    b. Repeatedly convert each byte in the array to a string with length of 8.
    c. Traverse the string, if the character is “0” go to the left side of the node; if the character is “1”, go to the right side of the node; if the pointer reaches to the leaf, write the data into “decoded.txt” file. And reset the pointer to point to the root.

Complexity of Decoder:

- Reading and saving the code table to a hashmap takes O(n) time, in which n is the entries in the code table.
- Construct Huffman tree with code table takes O(n*h), in which n is the entries in the code table, h is the length of the largest height of the Huffman tree.
- Decode the “encoded.bin” file with Huffman tree and output the results takes O(m) time, in which m is the number of bytes in the encoded file.
- Therefore, the total time complexity of decode algorithm is: O(m) +O(n*h)

Conclusion

This project implemented Huffman tree for encoding and decoding text files with cache optimized 4-way heap. The decoded file is exactly the same with the original input. Both input and output files have a size of 69,638,842 bytes. And their MD5 code is the same as well. The performance of binary heap, 4-way cache optimized heap and pairing heap on specific input was tested by counting the time lapse on constructing Huffman tree. As a result, 4-way cache optimized heap > binary heap > pairing heap was observed. The time complexity of the decoding algorithm is O(m) +O(n*h). Therefore, the objective of this project has been successfully met.

References:

1. Larkin, D.H., S. Sen, and R.E. Tarjan. *A back-to-basics empirical study of priority queues*. in *Proceedings of the Workshop on Algorithm Engineering and Experiments*. 2014.
2. Robert S., Kevin W., *Algorithms*, *Chapter 5.5 Data Compression.* 2011
3. Pairing Heap, https://titanwolf.org/Network/Articles/Article?AID=4b6231b8-fa25-45d2-af09-187230afff58#gsc.tab=0
4. Binary Heap, https://www.edureka.co/blog/binary-heap-in-java/#min
5. Sartarj, S., *Advanced Data Structure (lecture), Chapter7, 10, 15.* 2020