Advanced Data Structures

COP 5536 Spring 2017

Programming Project Report

Huffman Encoding and Decoding

Name: **Omkar Vaidya**
UFID: **9356-5481**

1. Project Description

Scenario: Software giant Toggle recently bought video streaming site MyTube. Now MyTube needs to send its data to Toggle server. As enormous amount of data will be transferred, they decided to use Huffman coding to reduce data size. Being a software engineer in MyTube, we are asked by our manager to write code for Huffman encoder and decoder.

2. Structure of the Program

The program consists of three application programs –

a. **Evaluation –** This program evaluates the performance of Binary Heap, Four Way Heap, Cache Optimized Four Way Heap and Pairing Heap. The program creates a frequency table, builds all 4 types of heap using the input file passed as argument and finally creates a Huffman Tree. This is repeated ten times using a for loop to achieve a better idea of which heap provides the best performance.

What did we observe? – For the sample_input_large.txt, the ***cache optimized four-way heap*** offered slightly better performance compared to the other heap structures on running the program multiple times. Following are the screenshots depicting the evaluation of all types of heaps.

Run 1:

```
C:\Users\Omkar Vaidya\Desktop\new>java evaluation sample_input_large.txt
Evaluating....

Frequency Table being built...
Frequency Table Built: 9110 milliseconds

Evaluating Binary Heap:
Time using binary heap: 9160 milliseconds

Evaluating Four-ary Heap...
Time using 4-ary heap: 7600 milliseconds

Evaluating 4-way Cache Optimized Heap...
Time using 4-Way cache optimized heap: 6677 milliseconds

Evaluating Pairing Heap...
Time using pairing heap: 11643 milliseconds
-----------------------------------------------------

C:\Users\Omkar Vaidya\Desktop\new>
```

Run 2:

```
C:\Users\Omkar Vaidya\Desktop\new>java evaluation sample_input_large.txt
Evaluating....

Frequency Table being built...
Frequency Table Built: 9093 milliseconds

Evaluating Binary Heap:
Time using binary heap: 9295 milliseconds

Evaluating Four-ary Heap...
Time using 4-ary heap: 7281 milliseconds

Evaluating 4-way Cache Optimized Heap...
Time using 4-Way cache optimized heap: 6946 milliseconds

Evaluating Pairing Heap...
Time using pairing heap: 10737 milliseconds
-----------------------------------------------------

C:\Users\Omkar Vaidya\Desktop\new>
```

b. encoder – The encoder program takes an input filename as argument (here it was sample_input_large.txt) and inserts the values along with their frequencies into a ***cache optimized four-way heap***. It then builds a Huffman Tree from the heap and creates a code table file name code_table.txt and an encoded file named encoded.bin

```
C:\Users\Omkar Vaidya\Desktop\new>java encoder sample_input_large.txt
  Frequency Table Build Time : 9076 milliseconds
  Heap Build Time            : 34 milliseconds
  Huffman Tree Building Time    : 1208 milliseconds
  Writing Code Table File Time : 705 milliseconds
  Writing Encoded Bin File     : 8993 milliseconds
  Total Time                 : 20016 milliseconds
-----------------------------------------------------

C:\Users\Omkar Vaidya\Desktop\new>
```

c. **decoder –** The decoder program takes as input the name of the code table file (code_table.txt) and the encoded file (encoded.bin). It builds a Huffman tree using the codes in the code table file. Once the Huffman tree is built, it then traverses the encoded.bin file to decode the binary string with the original string.

```
C:\Users\Omkar Vaidya\Desktop\new>java decoder encoded.bin code_table.txt
  Huffman Tree Build Time    : 3490 milliseconds
  Binary File Read Time      : 1409 milliseconds
  Build Original String Time : 6828 milliseconds
  Write Decoded File Time    : 940 milliseconds
  Total Time                 : 12667 milliseconds
-------------------------------------------------------

C:\Users\Omkar Vaidya\Desktop\new>
```

Moreover, there are other classes to support their working. These are as follows:

a. **DaryHeap** – This heap structure allows the user to specify the degree (number of children) and shift (shift the root position for cache optimization). With the help of degree and shift, this structure is used to instantiate a binary heap (which would be degree = 2, shift = 0), four-way heap (which would be degree = 4, shift = 0) and a cache optimized four-way heap (which would be degree = 4, shift = 3). DaryHeap uses an ArrayList of Node as ArrayLists are not synchronized and hence offer better performance than Vectors. The **Node** used by DaryHeap consist of two int values (2 x 4 bytes) and an object reference (8 bytes) to Huffman Node. Thus, the total size of a single Node is 16 bytes.

b. **PairingHeap** – Heap structure implementing a pairing heap. This structure used a different node structure (**PairingHeapNode**) as it consists of additional data – left sibling, right sibling and child pointers. For the melding process, the two-pass scheme is implemented.

c. **HuffmanTree** – It is similar to a binary tree. Each **HuffmanNode** in the tree consist of left and right pointers it children. Additionally, each node consists of a huffCode which is updated at specific points in the program. The huffCode of a node indicates the path traversed from the root of the tree to reach that node. 0 indicates left and 1 indicates right.

d. **TreeBuilder –** It helps to build a Huffman tree from a DaryHeap. It removes min value from the heap twice, makes those two values left and right child of a Huffman node, and pushes the sum of frequencies of those two values back into the heap. This is done until there is just one element in the heap.

3. Function Prototypes

DaryHeap Class

public boolean isEmpty ()

Description	Returns boolean indicating if the heap is empty
Parameters	None
Return Value	True - Heap is empty False - Heap is not empty
Calls	None
Called By	getHuffmanTreeAtRoot()

public int size ()

Description	Returns the size of the heap
Parameters	None
Return Value	Size of the heap
Calls	None
Called By	insert (int k, int f), insert (Node n), deleteMin (),minChild(int n), printHeap (), heapifyDown (int parent)

public HuffmanNode getHuffmanTreeAtRoot ()

Description	Returns the Huffman node that the root in the heap points to
Parameters	None
Return Value	Pointer to Huffman node
Calls	None
Called By	TreeBuilder - buildHuffmanTree() Evaluation – main (String[] args)

public int parent (int child)

Description	Returns the index of parent node for a given child node index
Parameters	Index of the child
Return Value	Index of the parent
Calls	None
Called By	heapifyUp (int child), heapifyDown (int parent)

public int kthChild (int parent, int k)

Description	Returns the index of Kth child node for a given parent node index

Parameters	Index of the parent, Index of the child (1 / 2 / … / n) where n is degree
Return Value	Index of the parent
Calls	None
Called By	minChild (int n), heapifyDown (int parent)

public void flush ()

Description	Empties the heap node
Parameters	None
Return Value	None
Calls	None
Called By	TreeBuilder - buildHuffmanTree() Evaluation – main (String[] args)

public void insert (int k, int f)

Description	Inserts a new node with key k and frequency f
Parameters	Values for key k and frequency f
Return Value	None
Calls	heapifyUp (int child)
Called By	Encoder – main(String[] args) Evaluation – main (String[] args)

public void insert (Node n)

Description	Inserts a new node n
Parameters	New node n
Return Value	None
Calls	heapifyUp (int child)
Called By	TreeBuilder - buildHuffmanTree()

public Node deleteMin ()

Description	Removes the minimum element from the heap
Parameters	None
Return Value	Deleted node
Calls	heapifyDown (int parent), size (),
Called By	TreeBuilder - buildHuffmanTree() Evaluation – main (String[] args)

public int minChild(int n)

Description	Returns index of child with minimum frequency
Parameters	Index of parent node
Return Value	Index of the minimum child
Calls	kthChild (int parent, int k),
Called By	heapifyDown (int parent)

public void heapifyUp (int child)

Description	To push values higher in the heap to satisfy heap property
Parameters	Index of child node
Return Value	None
Calls	None
Called By	insert (int k, int f), insert (Node n)

public void heapifyDown (int parent)

Description	To push values lower in the heap to satisfy heap property
Parameters	Index of the parent
Return Value	None
Calls	kthChild (int parent, int k)
Called By	deleteMin ()

PairingHeap Class

public boolean isEmpty ()

Description	Returns boolean indicating if the heap is empty
Parameters	None
Return Value	True - Heap is empty False - Heap is not empty
Calls	None
Called By	insert (PairingHeapNode hn), deleteMin()

public void flush ()

Description	Empties the heap node
Parameters	None
Return Value	None
Calls	None
Called By	Evaluation – main (String[] args)

public HuffmanNode getHuffmanTreeAtRoot ()

Description	Returns the Huffman node pointer at the root of the heap
Parameters	None
Return Value	Huffman node
Calls	None
Called By	Evaluation – main (String[] args)

public void insert (int v, int f)

Description	Insert a new node with value v and frequency f
Parameters	None
Return Value	None
Calls	insert (PairingHeapNode hn)
Called By	Evaluation – main (String[] args)

public void insert (PairingHeapNode hn)

Description	Inserts a new node t
Parameters	None
Return Value	None
Calls	isEmpty(),getMinFrequency()
Called By	Evaluation – main (String[] args)

public PairingHeapNode deleteMin()

Description	Removes the minimum node from the heap
Parameters	None
Return Value	None
Calls	meldHeaps(PairingHeapNode heap1, PairingHeapNode heap2)
Called By	Evaluation – main (String[] args)

public PairingHeapNode meldHeaps(PairingHeapNode heap1, PairingHeapNode heap2)

Description	Performs melding of heaps using the two pass scheme
Parameters	Two heaps that are to be melded
Return Value	None
Calls	None
Called By	PairingHeapNode deleteMin()

Huffman Tree

public void updateHuffCodes()

Description	Updates the huffcodes by calling updateHuffCode
Parameters	None
Return Value	None
Calls	updateHuffCode(HuffmanNode t, String s)
Called By	buildHuffmanTree()

public void updateHuffCode(HuffmanNode t, String s)

Description	Makes recursive calls to update huffcode of all nodes in the tree
Parameters	Huffman node to be updated and its parent's huffcode
Return Value	None
Calls	updateHuffCode(HuffmanNode t, String s)
Called By	updateHuffCodes(), updateHuffCode(HuffmanNode t, String s)

public void copyHuffCodes(String[] table)

Description	Calls copy(HuffmanNode t, String[] table) to copy Huffman codes into an array
Parameters	Array of string values
Return Value	None
Calls	copy(HuffmanNode t, String[] table)
Called By	TreeBuilder - buildCodeTable()

public void copy(HuffmanNode t, String[] table)

Description	Calls itself recursively to update Huffman codes in the tree
Parameters	Array and node whose Huffman code is to be copied into the array
Return Value	None
Calls	copy(HuffmanNode t, String[] table)
Called By	copyHuffCodes(String[] table), copy(HuffmanNode t, String[] table)

public void add (int value, String huffcode)

Description	Adds a new value to Huffman tree at the given given path (huffcode)
Parameters	Value to be added and its huffcode
Return Value	None
Calls	insertAt(HuffmanNode root, int value, String huffcode, String path)
Called By	Decoder – main (String[] args)

public void insertAt(HuffmanNode root, int value, String huffcode, String path)

Description	Inserts a new node with key *value* at the given *path* from *root*
Parameters	*Root* from where the *path* will be traversed, the value to be added and the path from the *root* to the new node
Return Value	None
Calls	insertAt(HuffmanNode root, int value, String huffcode, String path)
Called By	add (int value, String huffcode), insertAt(HuffmanNode root, int value, String huffcode, String path)

public StringBuilder build (String in_text)

Description	Constructs the original text from given string of 0s and 1s
Parameters	String of 0s and 1s
Return Value	None
Calls	None
Called By	Decoder – main (String[] args)

4. Performance Analysis

The cache optimized four-way heap produced slightly better performance than the other heap structures. The binary heap and pairing heap were slightly slower than the four-way heap implementations. The sample_input_large.txt was tested on a x64-based processor running 64-bit Windows 10.

Environment

Processor:	Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz 2.60 GHz
Installed memory (RAM):	8.00 GB (7.86 GB usable)
System type:	64-bit Operating System, x64-based processor

Observations

Checking which heap structure gives a better performance (on personal laptop) (Run 1):

```
C:\Users\Omkar Vaidya\Desktop\new>java evaluation sample_input_large.txt
Evaluating....

Frequency Table being built...
Frequency Table Built: 9093 milliseconds

Evaluating Binary Heap:
Time using binary heap: 9563 milliseconds

Evaluating Four-ary Heap...
Time using 4-ary heap: 7382 milliseconds

Evaluating 4-way Cache Optimized Heap...
Time using 4-Way cache optimized heap: 7197 milliseconds

Evaluating Pairing Heap...
Time using pairing heap: 12080 milliseconds
-----------------------------------------------------

C:\Users\Omkar Vaidya\Desktop\new>
```

Checking which heap structure gives a better performance (on personal laptop) (Run 2):

```
C:\Users\Omkar Vaidya\Desktop\new>java evaluation sample_input_large.txt
Evaluating....

Frequency Table being built...
Frequency Table Built: 7298 milliseconds

Evaluating Binary Heap:
Time using binary heap: 8372 milliseconds

Evaluating Four-ary Heap...
Time using 4-ary heap: 6392 milliseconds

Evaluating 4-way Cache Optimized Heap...
Time using 4-Way cache optimized heap: 6107 milliseconds

Evaluating Pairing Heap...
Time using pairing heap: 9127 milliseconds
-----------------------------------------------------

C:\Users\Omkar Vaidya\Desktop\new>
```

Running encoding and decoding on storm.cise.ufl.edu gave me the following result:

```
stormx:37% java encoder sample_input_large.txt
  Frequency Table Build Time : 10670 milliseconds
  Heap Build Time            : 34 milliseconds
  Huffman Tree Building Time    : 1174 milliseconds
  Writing Code Table File Time : 570 milliseconds
  Writing Encoded Bin File     : 10133 milliseconds
  Total Time                 : 22581 milliseconds
-------------------------------------------------------
stormx:38% java decoder encoded.bin code_table.txt
  Huffman Tree Build Time    : 3725 milliseconds
  Binary File Read Time      : 1745 milliseconds
  Build Original String Time : 11140 milliseconds
  Write Decoded File Time    : 1107 milliseconds
  Total Time                 : 17717 milliseconds
-------------------------------------------------------
stormx:39%
```

Total runtime (which includes encoding and decoding) is around 39000 milliseconds. This performance varies from machine to machine (eg. Time was much less on my personal laptop with the following environment mentioned above).

Analysis:

i. Binary heap and Pairing heap and have the same amortized complexity $O(\log_2 n)$ for both insert & delete (we consider only these two as we used only these two operations).

ii. Four-way heaps have better actual complexity $O(\log_4 n)$ for these operations. Hence, they are faster than the pairing and binary heap.

iii. It should be noted that four-way heaps will make more comparisons at each level (3 comparisons) compared to the binary heap (1 comparison) but they still offer better performance than the binary heap in the above program.

iv. Comparing normal four-way heap vs the cache optimized four-way heap, the latter offers slight advantage over the former on the system environment mentioned earlier. This might be because the size of a single node is 16 bytes and the size of a cache line on the above system is 64 bytes. Hence, a heap of degree 4 (4 children = 64 bytes) and shift 3 will offer optimal performance compared to a normal 4-ary heap as it will result in lesser number of cache misses.

5. Decoding Algorithm Used

The decoding algorithm I used is as follows:

```
//building the huffman tree using the code table
for (each line in the code_table file)
```

```
        value = first token
        huffcode = second token

        if huffmantree root == null                         //create huffman tree root if not already created
                root = new huffman node
        current = root

        while ((huffcode != "0") or (huffcode != "1"))      //traverse until we have just 0 or 1
                if (huffcode) == "0"
                        if current.left == null             //create left node if it doesn't exist
                                current.left = new huffman node
                        current = current.left              //0 hence we go left
                else
                        if current.right == null            //create right node if it doesnt exist
                                current.right = new huffman node
                        current = current.right             //1 hence we go right
                huffcode = huffcode.substring(1, huffcode.size)   //remove first char from huffcode

        //here we reach the parent of the new node
        if huffcode == "0"
                current.left = create new huffman node (value)
        else
                current.right = create new huffman node (value)

str => read binary file

//decoding the encoding string of 0s and 1s
current = huffman tree root

for (each char in str)
        if (char == 0)
                current = current.left
        else
                current = current.right

        if current == leaf node
                output huffmannode value to file
                current = root
```

Complexity

Consider an input file consisting of L lines having n distinct values.

The algorithm builds the Huffman tree from the codes by traversing the codes for each token in the code table file. Hence, if the number of values (or say tokens) in the file is ***n*** and the maximum length of huffcode is ***d***, the time to build the Huffman tree will be **O(nd)**

Similarly**,** while decoding the file, the program will run in **O(m)** time where m is the number of characters in the encoded string of 0s and 1s.

Therefore, total time = **O(nd) + O(m)**

Now, for evenly distributed values, the Huffman tree will be almost balanced.

Hence, we can assume

$d = \log_2 L$ --equation 1

$m = O(n\ d) = O(L \log_2 L)$ --equation 2 (from equation 1)

Therefore, total time = **$O(n \log_2 L) + O(L \log_2 L) = O(L \log_2 L)$**
where L = number of token in original file and n = number of distinct tokens in original file