COP 5536 Spring 2017

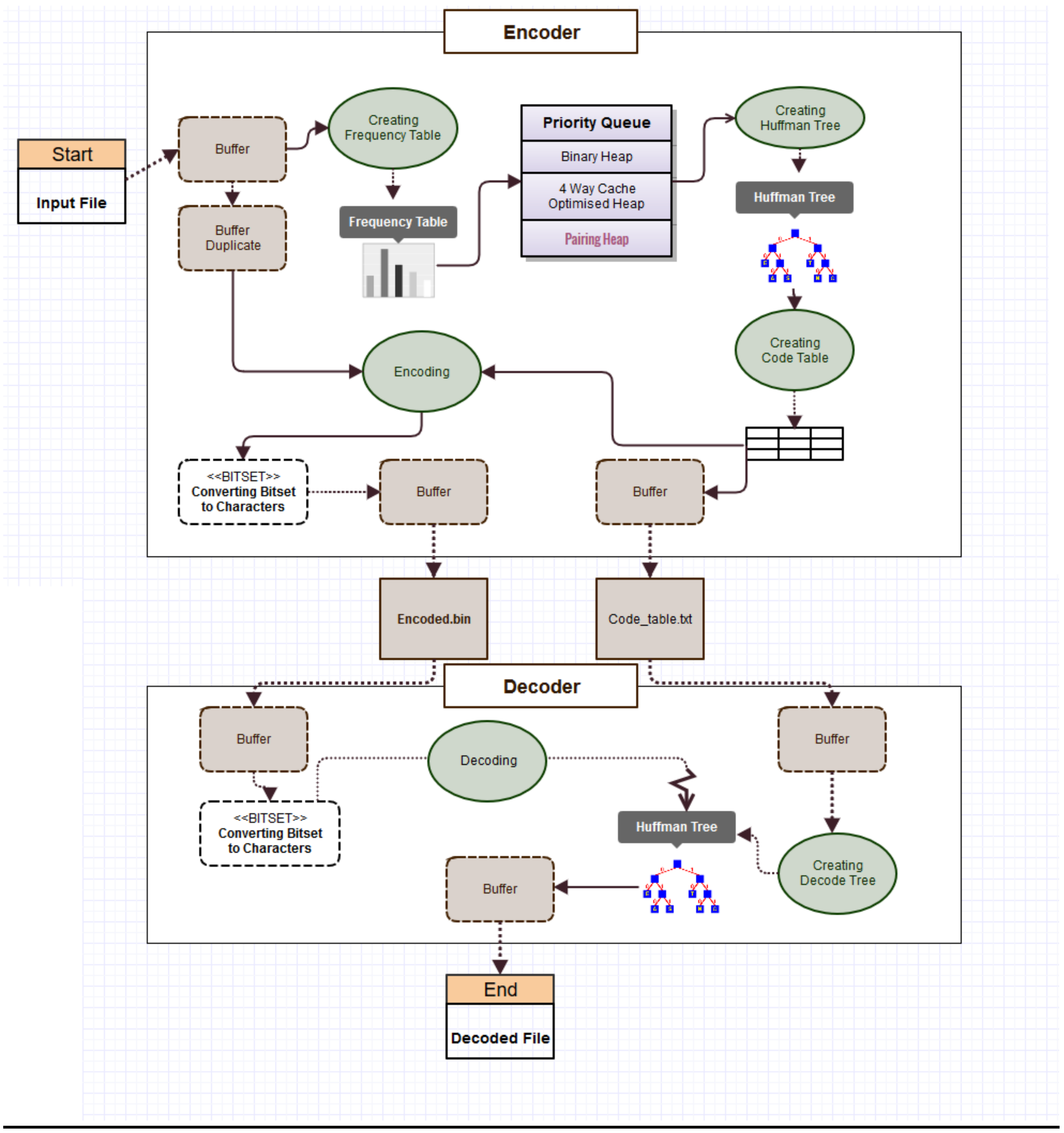
Advanced Data Structures

Programming Project

Name:Mohit Israni

UFID:4338 4979

Activity Diagram



Introduction

This document serves as a report to the Data Structure Coding Project. The Program is implemented in C++ and the required files along with the make file are included in the compressed directory with the report.

Program Structure

The project is divided into two parts. First part consists of developing a program that generates Huffman codes and creates an encoded file, given an input file. As a requirement of the project, performance of three priority queues, Binary Heap, 4-way cache optimized heap and Pairing Heap were evaluated for optimization. Code to generate Huffman trees using these three data structures was written and was then used to measure the run time of input data of *sample_input_large.txt*. Pairing Heap was obtained to give best performance. The first part returns two files *encoded.bin* and *code_table.txt*, which are then used as input files for second part of the project, consisting of decoding, to generate back the original input file. The structure of encoder and decoder is separately given below.

Encoder

Reading file into Buffer

Performing Bigger reads improves some performance by reducing the overhead of having too many read calls. Hence this is achieved by reading the file in parts, if large or complete if small into a buffer stream which is done by allocating a buffer space equal to the length of file. Since the file is going to be used twice, once for making a frequency table and then for encoding the data, copy of the buffer is made to avoid reading it twice.

```
class infileBuffer{
ifstream infile;
char * buffer;
char * buffer_cp;
buffer = new char [length];
buffer_cp = new char [length];
infile.read (buffer, length);
buffer_cp = copybuffer(buffer); }
```

Creation of Frequency Table using Array

For generation of Huffman Tree we need to arrange the list of different numbers(*n)* in order of their reoccurrence in a given file. The frequency of the numbers can be collected by incrementing its value for every occurrence in Hash maps or if the range of numbers is known, it can be done using an array or a vector. Since for this project we know the range of numbers, here I am using a vector of integers initialized to the size of 1000000.

$$\boldsymbol{class\ freqT\_arr}\{$$

$$void\ createFreqT();$$

$$void\ printFreqT();$$

$$\}$$

<u>Priority Queues</u>

To create a Huffman code, we need priority queue structures. To optimize the performance, we evaluate the performance of the three priority queue structures: Binary Heap, 4-way cache optimized heap and Pairing Heap. All the priority queue structures have Extract min(), Delete min(), Insert Element() and Size() functions as required for creating the Huffman Tree.

1. Binary Heap Structure

The ordering of the binary heap structure which is a complete binary tree is done such that it follows the *min-heap property*, which states that the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root. The Binary heap here is represented by storing its level order traversal in an vector. With:

- its left child located at 2*k+1 index
- its right child located at 2*k+2 index
- its parent located at (k-1)/2 index

Binary heap is implemented as follows:

$$\boldsymbol{class\ BinaryHeap}\{$$

$$void\ Insert(Node *\ element)\{\}$$

The function inserts a Node and calls heapifyup() function that brings the binary heap back to its Min heap property.

$$void\ DeleteMin()\{\}$$

It first deletes the top Node of the min heap, removes the last element from the heap and places it in the place of removed node. For the structure to follow the MinHeap Property it calls the Heapifydown() function at the top Node.

$$Node*\ ExtractMin()\{\}$$

The function eturns the the first element of the vector on which the Binary Heap is initialized.

$$int\ Size()\{\}$$

It returns the size of the vector, which is equal to the size of the binary heap.

$$int\ left(int\ parent)\{\}$$

$$int\ right(int\ parent)\{\}$$

$$int\ parent(int\ child)\{\}$$

The functions return left, right and parent child of the current node respectively.

$$void\ heapifyup(int\ in)\{\}$$

Heapifyup function takes the index of a node. If the value at that index is smaller than its parent, it swaps the nodes and iteratively calls itself with newly assigned index.

$$void\ heapifydown(int\ in)\{\}$$

Heapifydown function works like heapifyup, except that it compares a nodes value with its child instead of parent. It first compares with the left child, if its value is greater than child it swaps the nodes or else compares with the right child. Again, the function is called iteratively till the min heap property is satisfied.

2. ***4-Way Cache Optimized Heap Structure.***

The 4-way heap is a generalization of the binary heap in which the nodes have 4 children instead of 2. It can perform decrease priority operations to be performed more quickly than binary heaps, at the expense of slower delete minimum operations. Additionally, they have a better memory cache optimization, allowing them to run more quickly in practice despite having a theoretically larger worst-case running time. They can be cache optimized by starting the indexing at 3 instead of zero. With:

- its 1st child located at 4 * (parent-2) index
- its 2nd child located at 4 * (parent-2)+1 index
- its 3rd child located at 4 * (parent-2)+2 index
- its 4th child located at 4 * (parent-2)+3 index
- its parent located at (child/4)+2 index

4-way cache optimized heap is implemented as follows:

$$\boldsymbol{class\ FwayHeap}\{$$

$$void\ Insert(Node*\ element)\{\}$$

$$void\ DeleteMin$$

$$Node*\ ExtractMin()$$

```
int Size(){}
int child1(int parent){}
int child2(int parent){}
int child3(int parent){}
int child4(int parent){}
int parent(int child){}
void heapifyup(int in){}
void heapifydown(int in){}
}
```

The functions for 4-way cache optimized heap are almost same as the binary heap, except for the following:

- ExtractMin and DeleteMin operate at index 3 instead of 0.
- Heapifydown now compares with 4 childs instead of two.

3. *Pairing Heap Structure*

A pairing heap is a type of heap data structure with relatively simple implementation and excellent amortized performance. It is either an empty heap, or a pair consisting of a root element and a possibly empty list of pairing heaps. The heap ordering property requires that all the root elements of the subheaps in the list are not greater than the root element of the heap. For the Encoding program, the value used for comparison is the frequency of the heap. The nodes have the following structure:

```
struct Node{
struct Node * left = NULL;
struct Node * right = NULL;
struct Node * leftX = NULL;
struct Node * rightX = NULL;
struct Node * child = NULL;
int d1;
int d2;}
```

The first left and right nodes are used for Huffman tree ordering creation, while leftX and rightX nodes are used for creating the pairing heap.

```
class pairingHeap{
```

$$Node * \; Meld(Node * \; l, Node * \; r)\{\}$$

Meld function returns a root node of a new heap with its root as minimum of the two root elements as and just adds the heap with the larger root to the list of subheaps. It also breaks all the earlier left and right links of the nodes.

$$void \; Insert(Node * \; element)\{\}$$

If the heap is empty, insert makes the element root of the pairing heap, otherwise it melds the original pairing heap and the new element. The function also increase the stored heap size by 1;

$$void \; DeleteMin()\{\}$$

Delete min here is performed using the 2 pass scheme. The root node is deleted followed by pass 1 and pass 2.

Pass 1:

Children of the root are pairwise melded, reducing the number of children by half. If the number of children are odd, meld last child with the last newly generated subtree.

Pass 2:

Meld the last generated tree from the pass 1 with its previous tree. Then meld this tree with tree generated from pass 1 before the previous tree. Continues till only one tree remains.

Whenever trees are melded, care has is taken to update its left and right sibling fields, that is leftX and rightX here.

At last, the stored size of the heap is decremented by 1.

$$Node * \; ExtractMin()\{\}$$

It simply returns the top element/root of the heap.

$$int \; Size()\{\}$$

Returns the size of Pairheap.

Creation of Huffman Tree and Code Table and Encoding.

$$\boldsymbol{class \; huffmanT}\{$$

$$Node * \; CreateTree()\{\}$$

CreateTree() first creates new nodes for all the values in the frequency table while simultaneously inserting them in the Priority queue. Once all the trees are created, it extracts two minimum nodes at a time, inserting them as the left and right child of a new node, which has the d2(frequency) value as the sum of d2 values of two nodes. Then reinsert the newly created node back into the queue. It does till only one Node remains in the queue, which is then returned as the root of the Huffman tree created.

$$void \; codeTable(Node * \; element, vector < char > \; st)\{\}$$

CodeTable() starts with initializing a null vector of characters. It does inorder traversal of the Huffman tree. Each time the functions enters a left child char ‘0’ is pushed back into the vector and ‘1’ when it enters the right child. Whenever it comes out of the recursive function last element is popped out. At leaf nodes, the d1 value of the node, which represents the number in the file is pushed into the string stream along with the current value of bitstring. It values of bitstring are also stored in a vector of strings corresponding to each d1, which will be used will be used while encoding. When recursion ends, complete code table is stored in the string stream

$$void\ printCodeTable()\{\}$$

It writes the complete code table string stream into the $code\_table.txt$ file.

$$void\ Ncode()\{\}$$

For every number in the input file buffer, bitstring generated from the Huffman code is pushed into a output string stream. The output string stream of ones and zeros is written as a stream of ASCII characters made by using bitset of size 8 onto output file $encoded.bin$.

Decoder

Creation of Decoder (Huffman) Tree and Decoding of encoded file

The files $encoded.bin$ and $code\_table.txt$ generated from the Encoder are used here to generate the decoded file. The decoder first uses code table to generate the Huffman tree, which is in turn used to decode the encoded file.

$$\boldsymbol{Class\ huffmanT}\{$$

$$void\ CreateTree()\{\}$$

CreateTree() function first reads the $code\_table.txt$ file onto a buffer. Every line in the code table buffer is broken down into two strings, one represents Huffman code consisting of zeros and ones and other the actual value to be written. Huffman code in every line leads to a distinct leaf node created line by line. Huffman tree is initialized by creating a root node and making it the current node. Then Huffman code in the line is used to travel or create new node if required. For every ‘0’, if left child exists, it’s made the current node otherwise a new node is created and set as left child of current node and is then made the current node. Similarly, for every’1’, if right child exists, it’s made into the current node otherwise a new node is created and set as the right child of current node and then made the current node. When end of string is reached, the d1 of the current node is set to actual value to be written and then current node is set back to root. This continues till end of the code table buffer.

$$void\ Dcode()$$

This function first reads the $encoded.bin$ onto a buffer. Every ASCII character in the buffer is converted into a bitset of size 8 and then pushed onto a new string stream of ones and zeros. Now ones and zeros on the string are used to traverse the Decoded tree. The decoding starts by initializing the current node to root of the decode tree. One character at a time are read from the string stream. For every ‘0’, left child is made the current node, and for every ‘1’ right child is made the current node. If left or right child does not exist, d1 value of the current node is written to the output buffer and current node is set to the left or right child of node respectively. The traversal and writing continues till the end of string stream. At last the output buffer is written onto the $decoded.txt$ file.

Complexity analysis of Decoding Algorithm.

For every number to be decoded it must traverse to the leaf node of the decoded Huffman tree. Hence traversal can be of maximum height of the Huffman tree, which is of $O(n)$, where n is the number of distinct numbers in the input file or equal to the number of lines in $code\_table.txt$. There are N total numbers to be decoded, where N is equal to number of lines in the input file. Therefore the decoding has the complexity of $O(nN)$. Whereas creating the decoding tree requires n traversals of $O(n)$, hence the complexity of $O(n^2)$.

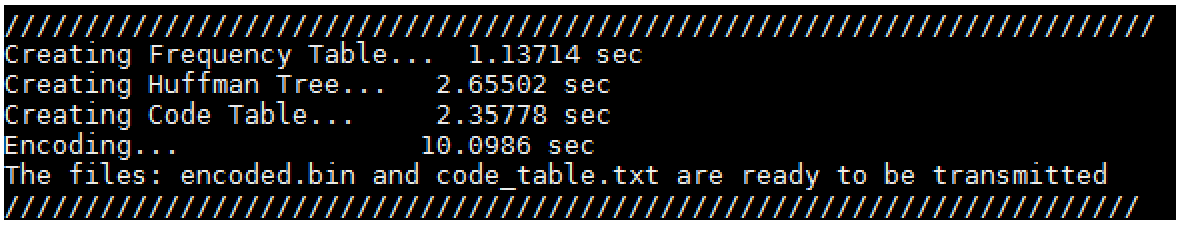
The total complexity of Decoding program will then be $O(n^2) + O(nN)$, which is equal to $O(nN)$.

<u>Performance Analysis</u>

The Table below shows the average time different Priority queues take to create the Huffman Tree.

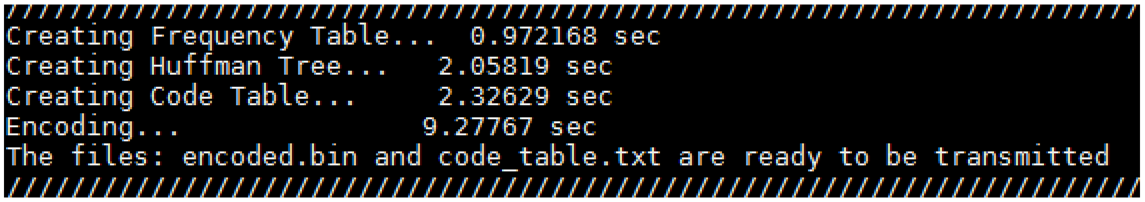
	Binary Heap	**4-way Heap (Cache Optimized)**	**Pairing Heap**
1	2.65502	2.95559	1.44169
2	2.92778	2.44292	1.34743
3	2.53718	2.04671	1.38167
4	2.92776	2.05819	1.45434
5	2.88118	2.76932	1.34767
Average	**2.785784**	**2.454546**	**1.39456**

<u>Encoding With Binary Heap</u>



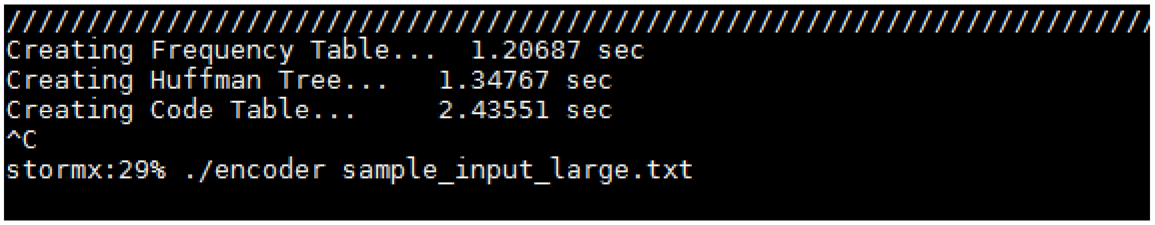
```
////////////////////////////////////////////////////////////////////////////
Creating Frequency Table...  1.13714 sec
Creating Huffman Tree...    2.65502 sec
Creating Code Table...      2.35778 sec
Encoding...                10.0986 sec
The files: encoded.bin and code_table.txt are ready to be transmitted
////////////////////////////////////////////////////////////////////////////
```

Encoding With 4Way Cache Optimized Heap



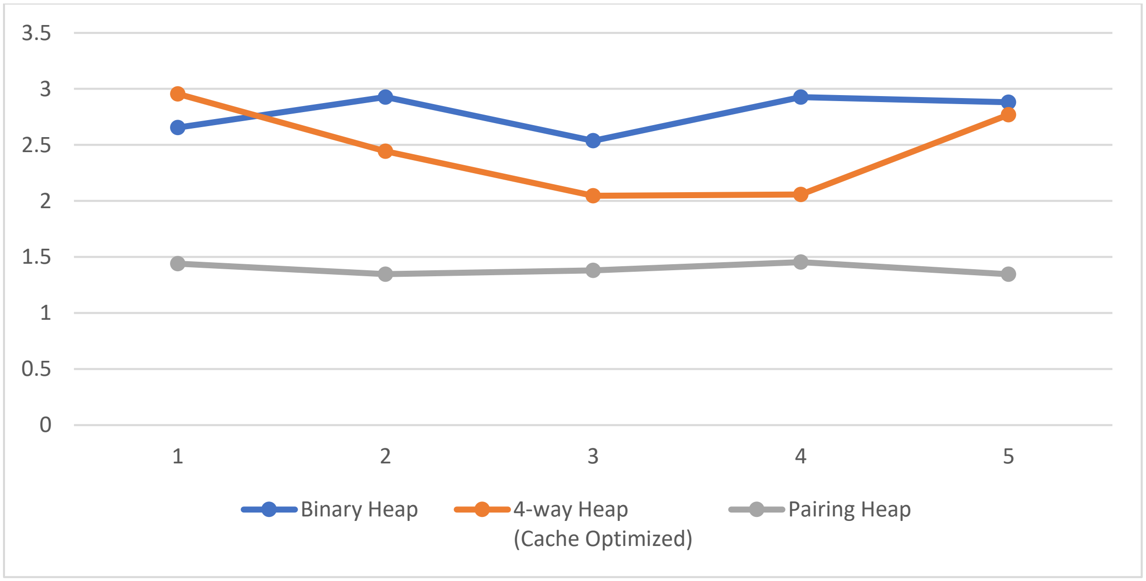
```
//////////////////////////////////////////////////////////////////////
Creating Frequency Table...  0.972168 sec
Creating Huffman Tree...    2.05819 sec
Creating Code Table...      2.32629 sec
Encoding...                9.27767 sec
The files: encoded.bin and code_table.txt are ready to be transmitted
//////////////////////////////////////////////////////////////////////
```

Encoding With Pairing Heap



```
//////////////////////////////////////////////////////////////////////
Creating Frequency Table...  1.20687 sec
Creating Huffman Tree...    1.34767 sec
Creating Code Table...      2.43551 sec
^C
stormx:29% ./encoder sample_input_large.txt
```

The Chart below shows the trend of time taken to form Huffman tree in seconds using different priority queue



As can be seen from above Table and charts, Pairing Heap as a priority queue data structure gives the best performance. Average time for creation of Huffman code using Pairing heap is 1.39 seconds.

The overall average time for Encoding a file is 14 seconds while decoding a file takes 15 seconds.

Conclusion

- Pairing Heap gives the best Performance as a priority queue data structure.
- The total complexity of Decoding program will then be $O(n^2) + O(nN)$, which is equal to $O(nN)$.