**Huffman Coding Programming Project**

Advanced Data Structures (COP 5536) Spring 2017

Chang Xu

Email: theoneneo@ufl.com UFID: 94101882

**Table of Contents**

[1. Introduction 3](#_Toc479191587)

[2. Priority Queue Implementation 4](#_Toc479191588)

[2.1. Program Structure 4](#_Toc479191589)

[2.2. Binary Heap 5](#_Toc479191590)

[2.2.1. Data Structure 5](#_Toc479191591)

[2.2.2. Function Prototype 5](#_Toc479191592)

[2.3. Four-Way Heap 6](#_Toc479191593)

[2.3.1. Data Structure 6](#_Toc479191594)

[2.3.2. Function Prototype 6](#_Toc479191595)

[2.4. Pairing Heap 7](#_Toc479191596)

[2.4.1. Data Structure 7](#_Toc479191597)

[2.4.2. Function Prototype 7](#_Toc479191598)

[2.5. Performance Analysis 8](#_Toc479191599)

[3. Huffman Encoding 9](#_Toc479191600)

[3.1. Encoding Algorithm 9](#_Toc479191601)

[3.2. Program Structure 10](#_Toc479191602)

[3.3. Data Structure 11](#_Toc479191603)

[3.4. Function Prototype 11](#_Toc479191604)

[3.5. Encoding Result 12](#_Toc479191605)

[4. Huffman Decoding 14](#_Toc479191606)

[4.1. Decoding Algorithm 14](#_Toc479191607)

[4.2. Program Structure 16](#_Toc479191608)

[4.3. Data Structure 17](#_Toc479191609)

[4.4. Function Prototype 17](#_Toc479191610)

[4.5. Decoding Result 18](#_Toc479191611)

[5. Performance in Other Environments 20](#_Toc479191612)

[5.1. Windows 20](#_Toc479191613)

[5.2. Ubuntu 20](#_Toc479191614)

# **Introduction**

This report gives detailed description how the Huffman Coding Program is implemented. The Program is developed in C++ and it comprises two major executable parts: Encoder and Decoder. Besides the two, it also contains a Benchmark part to examine the performance of 3 types of Priority Queue.

Section 2 Priority Queue Implementation provides details about the 3 types of Priority Queue – Binary Heap, Four-Way Heap, and Pairing Heap – and their performance analysis.

Section 3 Huffman Encoding provides details about the algorithm, data structure, program structure, function prototypes and encoding result of the Encoder.

Section 4 Huffman Decoding provides details about the algorithm, data structure, program structure, function prototypes and encoding result of the Decoder.

Section 5 Performance in Other Environments gives more program performance results in environments other than CISE Server.

# **Priority Queue Implementation**

Priority Queue is needed to build the Encoder Huffman Tree using Greedy Algorithm. There are 3 types of Priority Queue implemented in this project: Binary Heap, Four-Way Heap, and Pairing Heap.

## **Program Structure**

|  |  |
| --- | --- |
| **Heap Program Structure** | The program structure of all 3 Heaps is the same as shown on the left. It takes frequency table along with its size as input and gives generated Huffman Tree as output. |

## **Binary Heap**

Binary Heap is implemented according to the program structure in Section 2.1.

The heap is stored in an array. It is easy to access node parent and children using array index. The index of root is set to 1 instead of 0. And the index of parent is (current\_index / 2); the index of left child is (current\_index \* 2); the index of right child is (current\_index \* 2 + 1).

The complexity of Heap Initialization is O(n log n); the complexity of building Huffman Tree is O(n log n). Hence, the complexity of Binary Heap is O(n log n).

## **Data Structure**

|  |  |
| --- | --- |
| **Structure** | binary\_entry |
| **Attribute** | int key  void\* value\_pointer |
| **Description** | Binary Heap entry. |

## **Function Prototype**

|  |  |
| --- | --- |
| **Function** | insert |
| **Argument** | int key  void\* value\_pointer |
| **Return Type** | int |
| **Return Value** | 0 - success |
| **Description** | Insert entry into binary heap. |
| **Complexity** | O(log n) |

|  |  |
| --- | --- |
| **Function** | remove\_min |
| **Argument** | binary\_entry\* output |
| **Return Type** | int |
| **Return Value** | 0 - success  -1 - heap is empty |
| **Description** | Remove min from binary heap. And calculate new root after removal. |
| **Complexity** | O(log n) |

|  |  |
| --- | --- |
| **Function** | build\_huffman\_tree\_using\_binary\_heap |
| **Argument** | int\* frequency\_table  int size\_of\_frequency\_table |
| **Return Type** | huffman\_node\* |
| **Return Value** | Pointer to generated Huffman Tree. |
| **Description** | Build Huffman Tree using binary heap. |
| **Complexity** | O(n log n) |

## **Four-Way Heap**

Four-Way Heap is implemented according to the program structure in Section 2.1.

The heap is stored in an array. It is easy to access node parent and children using array index. The index of root is set to 3 instead of 0. And the index of parent is (current\_index / 4 + 2); the index of the first child is (current\_index \* 4); the index of other children can be calculated based on the first child.

The complexity of Heap Initialization is O(n log n); the complexity of building Huffman Tree is O(n log n). Hence, the complexity of Four-Way Heap is O(n log n).

## **Data Structure**

|  |  |
| --- | --- |
| **Structure** | four\_way\_entry |
| **Attribute** | int key  void\* value\_pointer |
| **Description** | Four-Way Heap entry. |

## **Function Prototype**

|  |  |
| --- | --- |
| **Function** | insert |
| **Argument** | int key  void\* value\_pointer |
| **Return Type** | int |
| **Return Value** | 0 - success |
| **Description** | Insert entry into four-way heap. |
| **Complexity** | O(log n) |

|  |  |
| --- | --- |
| **Function** | remove\_min |
| **Argument** | four\_way­\_entry\* output |
| **Return Type** | int |
| **Return Value** | 0 - success  -1 - heap is empty |
| **Description** | Remove min from four-way heap. And calculate new root after removal. |
| **Complexity** | O(log n) |

|  |  |
| --- | --- |
| **Function** | build\_huffman\_tree\_using\_four\_way\_heap |
| **Argument** | int\* frequency\_table  int size\_of\_frequency\_table |
| **Return Type** | huffman\_node\* |
| **Return Value** | Pointer to generated Huffman Tree. |
| **Description** | Build Huffman Tree using four-way heap. |
| **Complexity** | O(n log n) |

## **Pairing Heap**

Pairing Heap is implemented according to the program structure in Section 2.1.

The heap is stored in an array but node is accessed via pointer instead of array index. Insert function uses the meld function directly; remove\_min uses meld function to implement two-pass melding.

The complexity of Heap Initialization is O(n); the complexity of building Huffman Tree is O(n log n). Hence, the complexity of Four-Way Heap is O(n log n).

## **Data Structure**

|  |  |
| --- | --- |
| **Structure** | pairing\_node |
| **Attribute** | int key  void\* value\_pointer  pairing\_node\* left\_sibling  pairing\_node\* right\_sibling  pairing\_node\* child |
| **Description** | Pairing Heap node. |

## **Function Prototype**

|  |  |
| --- | --- |
| **Function** | meld |
| **Argument** | pairing\_node\* root1  pairing\_node\* root2 |
| **Return Type** | pairing\_node\* |
| **Return Value** | Pointer to result tree. |
| **Description** | Meld two trees into one. |
| **Complexity** | O(1) |

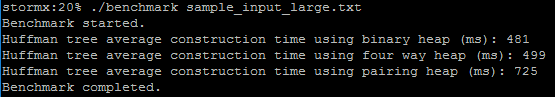
|  |  |
| --- | --- |
| **Function** | insert |
| **Argument** | pairing\_node\* node |
| **Return Type** | void |
| **Return Value** | NA |
| **Description** | Insert node into pairing heap using meld function. |
| **Complexity** | O(1) |

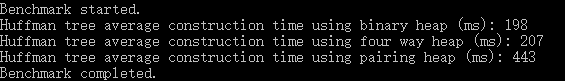
|  |  |
| --- | --- |
| **Function** | remove\_min |
| **Argument** | void |
| **Return Type** | pairing\_node\* |
| **Return Value** | Pointer to min node in pairing heap. |
| **Description** | Remove min from pairing heap. And meld subtrees using two-pass scheme. |
| **Complexity** | O(log n) |
| **Function** | build\_huffman\_tree\_using\_pairing\_heap |
| **Argument** | int\* frequency\_table  int size\_of\_frequency\_table |
| **Return Type** | huffman\_node\* |
| **Return Value** | Pointer to generated Huffman Tree. |
| **Description** | Build Huffman Tree using pairing heap. |
| **Complexity** | O(n log n) |

## **Performance Analysis**

The benchmark reuslt of 3 types of Heap is shown below. The input file is a 66.4MB sample file containing 10 million records; and for each heap the iteration time is 10.

Though the time consumed to build Huffman Tree varies on different plantform, the relative speed is consistent: Binary Heap has the best performance, then is Four-Way Heap, and Pairing Heap is the slowest. However, considering the size of input frequency table is nearly 1 million, all the 3 Heaps are giving quite excellent performance. The building of Huffman Tree takes less than 1 second to complete.

  
**Benchmark Result on CISE Server**

  
**Benchmark Result on Local Windows**

Theoretically, Four-Way Heap is supposed to have the best performance, but the benchmark reuslt shows different. The discrepancy is probrably due to the optimized implementation of Binary Heap: storage array index starts at 1 rather than 0 makes the children of every heap node lie at the same cache line. Hence, the Binary Heap is cache optimized.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **root** | **c1** | **c2** | **c11** | **c12** | **c21** | **c22** | **c111** | **c112** | **c121** | **c122** | **c211** | **c212** | **c221** | **c222** |

**Binary Heap Array**

# **Huffman Encoding**

## **Encoding Algorithm**

The algorithm used to encode file is:

1. Go through input file to get frequency of each value.
2. Build encoding Huffman Tree based on frequency data:
   1. Use Binary Heap as Priority Queue (because it has the best performance).
   2. Use Greedy Algorithm to build up Huffman Tree.
3. Traverse generated Huffman Tree to output code table file and build internal coding table.
4. Iterate every line of input file again to translate into coded bit.
5. Write bit code to output file via bit buffer function.

|  |
| --- |
| //build huffman tree  new\_huffman\_memory(num\_of\_values);  huffman\_tree = build\_huffman\_tree(frequency\_table, num\_of\_values);  delete[] frequency\_table;  frequency\_table = NULL;  //build and output code table  code\_table = new code\_entry[num\_of\_values];  output\_file.open(output\_code\_table\_file\_name, ios::binary);  traverse\_tree(huffman\_tree); //traverse huffman tree to initialize and output code table  output\_file.close();  delete\_huffman\_memory();  //encode and output source file  input\_file.clear();  input\_file.seekg(0, ios::beg);  output\_file.open(output\_encoded\_file\_name, ios::binary);  while (input\_file >> input\_string) {  code\_index = string\_to\_int(input\_string);  write\_bit(code\_table[code\_index].bit\_code, code\_table[code\_index].bit\_size);  } |

**Code Snippet of Encoding File**

The complexity to get frequency data is O(n); the complexity to encoding input file is O(n). Hence, the overall complexity of encoder is O(n).

## **Program Structure**

|  |  |
| --- | --- |
| **Encoder Program Structure** | Encoder takes input file name as input and gives “encoded.bin” and “code\_table.txt” as output. |

## **Data Structure**

|  |  |
| --- | --- |
| **Structure** | huffman\_node |
| **Attribute** | int numeric  char string[SIZE\_OF\_STRING\_VALUE]  huffman\_node\* left\_child  huffman\_node\* right\_child |
| **Description** | Huffman Tree node. |

## **Function Prototype**

|  |  |
| --- | --- |
| **Function** | write\_bit |
| **Argument** | unsigned long long bit\_code  int bit\_size |
| **Return Type** | void |
| **Return Value** | NA |
| **Description** | Write bit code to output file |
| **Complexity** | O(1) |

|  |  |
| --- | --- |
| **Function** | traverse\_node |
| **Argument** | huffman\_node\* node  string string\_code |
| **Return Type** | void |
| **Return Value** | NA |
| **Description** | Traverse Huffman Tree node to set and output code. |
| **Complexity** | O(n) |

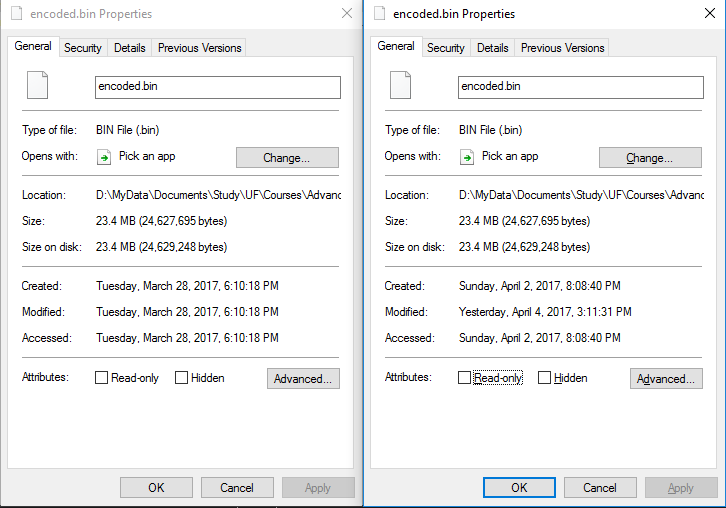
|  |  |
| --- | --- |
| **Function** | traverse\_tree |
| **Argument** | uffman\_node\* root |
| **Return Type** | void |
| **Return Value** | NA |
| **Description** | Calling function of traverse\_node. |
| **Complexity** | O(n) |

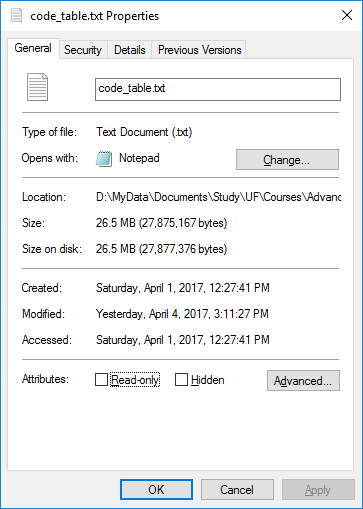
|  |  |
| --- | --- |
| **Function** | encoder |
| **Argument** | char\* input\_source\_file\_name  char\* output\_encoded\_file\_name  char\* output\_code\_table\_file\_name |
| **Return Type** | int |
| **Return Value** | 0 - success  -1 - error |
| **Description** | Encode input file. |
| **Complexity** | O(n) |

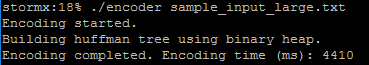
|  |  |
| --- | --- |
| **Function** | main |
| **Argument** | int argc  char\* argv[] |
| **Return Type** | int |
| **Return Value** | 0 - success |
| **Description** | Calling function encoder. |
| **Complexity** | O(n) |

## **Encoding Result**

The encoding processing gives a quite excellent performance. It took only 4410 ms on CISE Server to encoder the 66.4MB size 10 million record file. The encoded file has the same size as the sample encoded file. And encoder also outputs a code table file as required. The Huffman Encoding is successful.

  
**File Size Comparison**

  
**Output Code Table File**

  
**Time Used on CISE Server**

# **Huffman Decoding**

## **Decoding Algorithm**

The algorithm used to decode file is:

1. Build decoding Huffman Tree from input code table file.
2. While iterating the input code table file, find out the minimal length of bit code.
3. Traverse the generated Huffman Tree to every node whose degree is the minimal length of bit code. Store the node pointers as the starting node table.
4. Go through the encoded file to decode.
   1. Read bit code of minimal length from the file via a bit buffer function.
   2. User starting node table to get access to a node which could be very close to the leaf.
   3. Read 1 bit at a time to find the corresponding leaf.
   4. Write decoded string to output file.
   5. Loop until end of input file.

|  |
| --- |
| //decode and output file  output\_file.open(output\_decoded\_file\_name, ios::binary);  while (read\_bit(&bit\_code, min\_bit\_size) == 0) { //performance is 30% better than starting at root and reading 1 bit each time  current\_node = starting\_node\_table[bit\_code]; //direct access to node very close to leaf  if (current\_node->left\_child != NULL) {  while (read\_bit(&bit\_code, 1) == 0) {  current\_node = bit\_code == 0 ? current\_node->left\_child : current\_node->right\_child;  if (current\_node->left\_child == NULL) {  break;  }  }  }  output\_file << current\_node->string << "\x0A";  } |

**Code Snippet of Decoding File**

To build the decoding Huffman Tree, following processing is made:

1. Read one line of string from the input code table file.
2. Start traverse from the root of Huffman Tree.
3. Iterate every byte of the string.
4. Create needed node if not exist.
5. Copy decoding string to leaf.
6. Recording minimal bit size of code.
7. Loop the processing until end of the code table file.

|  |
| --- |
| //build huffman tree from code table file  huffman\_tree = new\_huffman\_node();  while (code\_table\_file >> string\_value >> string\_code) {  current\_node = huffman\_tree;  for (i = 0; string\_code[i] != '\0'; i++) {  if (string\_code[i] == '0') {  if (current\_node->left\_child == NULL) {  current\_node->left\_child = new\_huffman\_node();  }  current\_node = current\_node->left\_child;  } else {  if (current\_node->right\_child == NULL) {  current\_node->right\_child = new\_huffman\_node();  }  current\_node = current\_node->right\_child;  }  }  copy\_string(current\_node->string, string\_value);  if (strlen(string\_code) < min\_bit\_size) {  min\_bit\_size = strlen(string\_code);  }  } |

**Code Snippet of Building Decoding Huffman Tree**

The complexity of building Huffman Tree is O(n) in terms of the total length of all code bit (not including value string part) in the code table file.

Since every bit in the encoded file is read only once and trigger at most one changing of node in the Huffman Tree, the overall complexity of decoder is O(n).

## **Program Structure**

|  |  |
| --- | --- |
| **Decoder Program Structure** | Decoder takes encoded file name and code table file name as input and gives “decoded.txt” as output. |

## **Data Structure**

|  |  |
| --- | --- |
| **Structure** | huffman\_node |
| **Attribute** | int numeric  char string[SIZE\_OF\_STRING\_VALUE]  huffman\_node\* left\_child  huffman\_node\* right\_child |
| **Description** | Huffman Tree node. |

## **Function Prototype**

|  |  |
| --- | --- |
| **Function** | read\_bit |
| **Argument** | unsigned long long\* bit\_code  int bit\_size |
| **Return Type** | int |
| **Return Value** | Bit code value in integer format. |
| **Description** | Read bit code from encoded file |
| **Complexity** | O(1) |

|  |  |
| --- | --- |
| **Function** | traverse\_node |
| **Argument** | huffman\_node\* node  string string\_code |
| **Return Type** | void |
| **Return Value** | NA |
| **Description** | Traverse Huffman Tree node to set starting node table. Recursion function. |
| **Complexity** | O(n) |

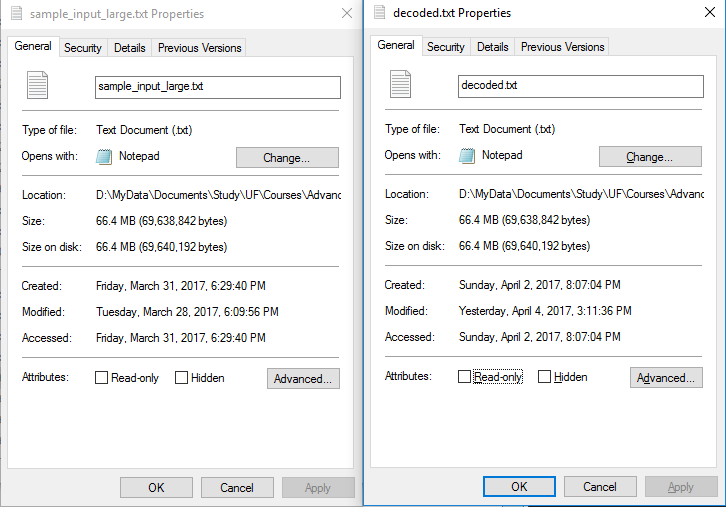
|  |  |
| --- | --- |
| **Function** | traverse\_tree |
| **Argument** | uffman\_node\* root |
| **Return Type** | void |
| **Return Value** | NA |
| **Description** | Calling function of traverse\_node. |
| **Complexity** | O(n) |

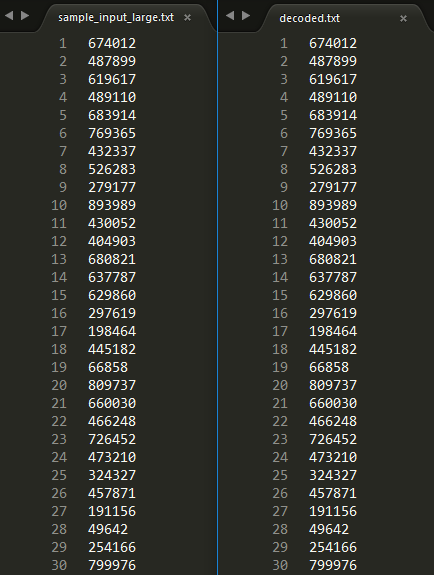
|  |  |
| --- | --- |
| **Function** | decoder |
| **Argument** | char\* input\_encoded\_file\_name  char\* input\_code\_table\_file\_name  char\* output\_decoded\_file\_name |
| **Return Type** | int |
| **Return Value** | 0 - success  -1 - error |
| **Description** | Decode input file. |
| **Complexity** | O(n) |

|  |  |
| --- | --- |
| **Function** | main |
| **Argument** | int argc  char\* argv[] |
| **Return Type** | int |
| **Return Value** | 0 - success |
| **Description** | Calling function decoder. |
| **Complexity** | O(n) |

## **Decoding Result**

Thanks to the starting node table, the decoding processing took even less time than encoding. The time used is only 4096 milliseconds on CISE Server. The decoded file has both the same size and the same content as the source sample file. The Huffman Decoding is successful.

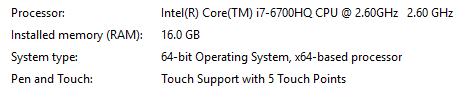
  
**File Size Comparison**

  
**File Content Comparison**

  
**Time Used on CISE Server**

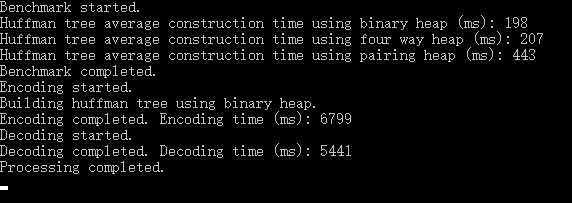
# **Performance in Other Environments**

The performance result of Encoder, Decoder and Benchmark on Windows and Ubuntu are shown below.

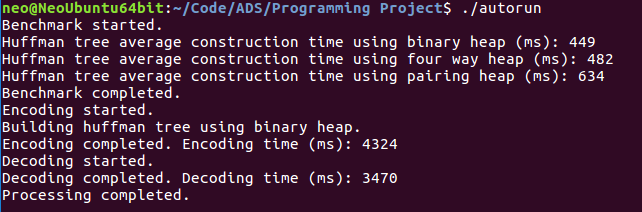


**System Information**

## **Windows**

  
**Performance Result on Windows 10 64-bit**

## **Ubuntu**

  
**Performance Result on Ubuntu 16.04 64-bit (Virtual Machine)**