

# UNIVERSITY OF FLORIDA

## Data Compression Via Huffman Tree Encoding & Decoding

### Project Report



**NAME:** Ashwin A Nair

**COURSE:** Advanced Data structures (COP5536)

**DATE:** 5<sup>th</sup> April, 2017

**UFID:**0919-7990

## FUNCTION PROTOTYPES

The Huffman tree generation of our sample input data is achieved using code frequency table through different types of data structures including pairing heap, binary heap and 4-way cache optimized heap and their implementations of operations including primarily two consecutive remove\_min operations and an insert operation.

Major parts of the project are segregated into various sections:-

### Header Files:

- binaryheap.h
- pairingheap.h
- fourarycoheap.h
- decoder.h
- nodehuffman.h

### CPP Files:

- binaryheap.cpp
- pairingheap.cpp
- fourarycoheap.cpp
- decoder.cpp
- decoder\_huffman.cpp
- encoder.cpp

The classes of the data structures used in the project is written using object-oriented programming. Instead of writing all the code in one file, and compiling that one file, the files are separated as .cpp and .h files. A .cpp file consists of implementation methods while .h (header) file contains class declaration and function prototypes.

- binaryheap: This is binary heap code which is used for inserting node, deleting the min node and for every insert and remove\_min, the heap is heapified using functions heapup and heapdown.
- pairingheap: The pairing heap is different than the previous two. The insert function inserts the element in the tree but on deleting the minimum node from the tree, the subtrees are combined using sib\_combine. The meld function compares two subtrees and the subtree with greater root node is attached to the other subtree.

- **fourarycoheap**: This is four-way heap, the index of the nodes start from 3 to achieve cache optimization. Thus the parent and children relation is modified using formulae,  $i/4+2$  to find the parent of  $i$ th node and  $(4 * (i-2) + (k-1))$  as  $k$ th child of node  $i$ . Rest of the code has functions to insert element and delete the minimum node, and heapification is implemented using **heapup** and **heapdown**.

**node\_huffman**: It is defined as a binary tree node's class which is attributed with data, frequency value, and some designated pointers as left and right to efficiently parse through the nodes of our generated Huffman Trees.

Other integral methods associated with these data structures include: **minextract** (used in **remove\_min**), **heapup**, **heapdown**, **size**, **parent**, **lchild** (left child), **rchild** (right child), **k\_child** ( $k$ th smallest child of the tree), **min\_ch** (smallest child), **meld** (used in pairing heaps).

The significant functions of **decoder\_huffman.cpp**, **decoder.cpp** are:

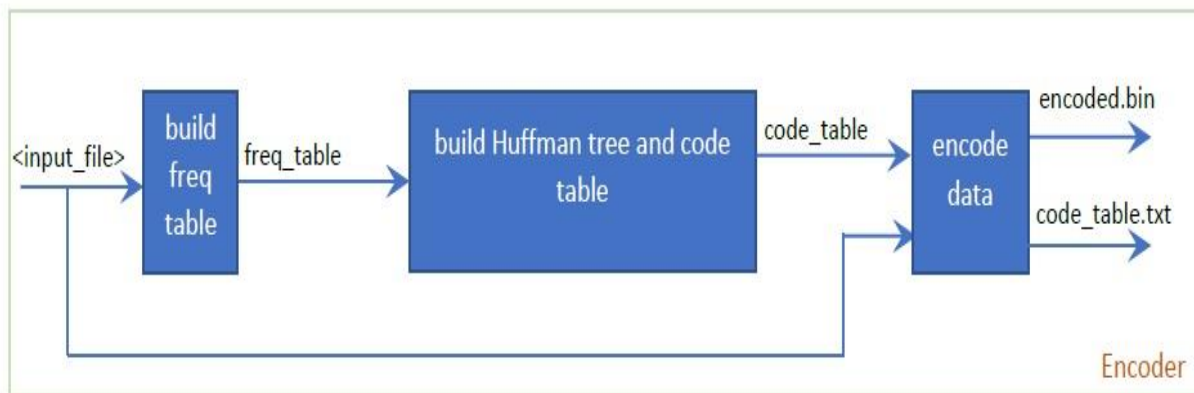
1. **struct Node {struct Node \*right = NULL; struct Node \*left = NULL; int value; bool isLeaf = false;};**- This structure is used to define the node of the Huffman tree which is intended to be decoded.
2. **Node\* getRoot()**-This method is used to get the root of the Huffman tree.
3. **void addleaf\_at(int value, string binaryString)**-After parsing the **code\_table.txt** it constructs the decode-huffman tree.
4. **int extract(string binaryString)**-It traverses the decode tree to parse the binary string and returns the valid value stored in the leaf node.
5. The primary function of decoding the entire **encoded.bin** file is executed by the **main** of **decoder\_huffman.cpp**.

The significant functions of encoder.cpp are:

1. **void FillFreqMapFromFile**- It fills the frequency map from the sample input file.
2. **void CreateHnodeVecFromFreqMap**(std::unordered\_map<int,unsigned int> const &freq\_map, std::vector<node\_huffman const\*> &vec)- It creates a Huffman-Tree vector with the help of the frequency map.
3. **node\_huffman const\* BuildTreeFheap**(std::unordered\_map<int,unsigned int> const &freq\_map)- It builds the Huffman tree by utilizing a 4-way cache optimized heap as a data structure.
4. **node\_huffman const\* BuildTreeBheap**(std::unordered\_map<int,unsigned int> const &freq\_map)- It builds the Huffman tree by utilizing a binary heap as a data structure.
5. **node\_huffman const\* BuildTreePheap**(std::unordered\_map<int,unsigned int> const &freq\_map)- It builds the Huffman tree by utilizing a pairing heap as a data structure.
6. **void printCodes**(node\_huffman const \*root, std::string str, std::unordered\_map<int,std::string> &code\_table\_map)- It traverses the Huffman tree to generate key value pairs for the code table.
7. **void compress**(std::string &value,std::vector<char> &result)- It carries out the 8-bit compression process.
8. **void binfilecreate**(std::unordered\_map<int,std::string> &code\_table\_map, char \*filename, char const \*out)- It creates the final encoded binary by using the hash map and the code table generated intermediately.

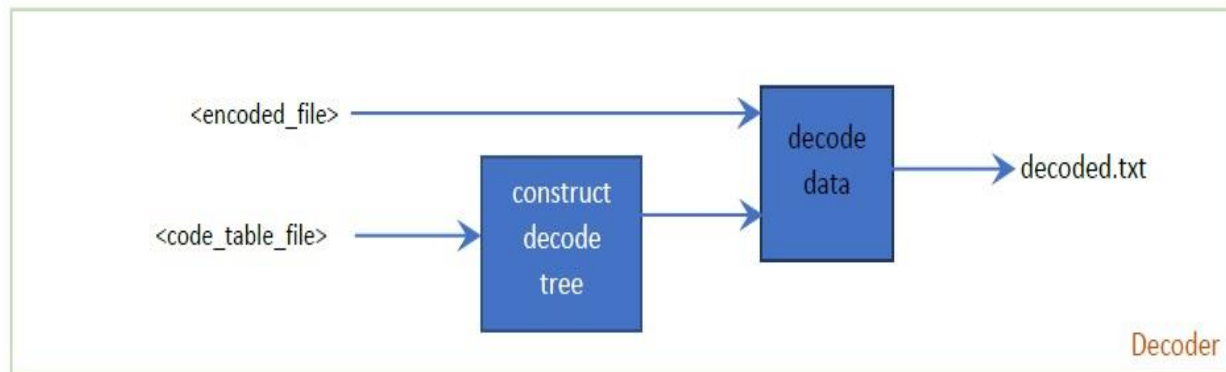
## Encoding Design

- Input both sample\_input\_small & sample\_input\_large files separately.
- Build the Frequency map.
- Insert into binary, pairing & four-way cache optimized heap with frequency as the key value.
- Huffman trees for the respective data structures are built.
- Huffman codes are also specified for each input value in the file and stored in a code table
- Creating a binary file with Huffman codes which also undergoes 8-bit compression.



## Decoding Design

- Parse the code table.
- Build the Decoder Tree.
- Read the encoded.bin file.
- Scan every byte and convert it to its respective binary string.
- Create its respective Huffman-coded string.
- Traverse the Decoder Tree and replace Huffman codes with values.



## PERFORMANCE RESULTS, ANALYSIS & EXPLANATION

It was evident that for small input files all three data structures clocked same timings for Huffman tree building. Yet for the larger sample input file four-way cache optimized heap clocked the best performance among the three data structures. The “encoded.bin” file generated is of size 24.051MB.

The average time for each of the three data structures used for Huffman tree generations are: -

1. Binary Heap- 2.14536s
2. Pairing Heap- 2.164925s
3. Four-Way Cache Optimized Heap- 1.691915s

Also, the time taken by the encoding and decoding process is clocked at 15.86s and 48.90s respectively.

- The four way is fastest because it has smaller height compared to the other two. Hence, when the input is very large, the height of the tree matters because every heapifying process of the data structure takes  $O(h)$  time.
- Cache optimization is more in Four-way because the index of the parents starts from 3, the children will thus be at indices 4, 5, 6, 7. Thus, the siblings will be at consecutive indices which enables them to be at the same cache line reducing the number of memory accesses.
- Additionally, Pairing heaps are relatively slower since every insert takes  $O(1)$  whereas its `remove_min` operation takes  $O(n)$  time and this are the operations utilized for Huffman code generation as mentioned earlier.

## DECODING ALGORITHM & ANALYSIS

### ALGORITHM: -

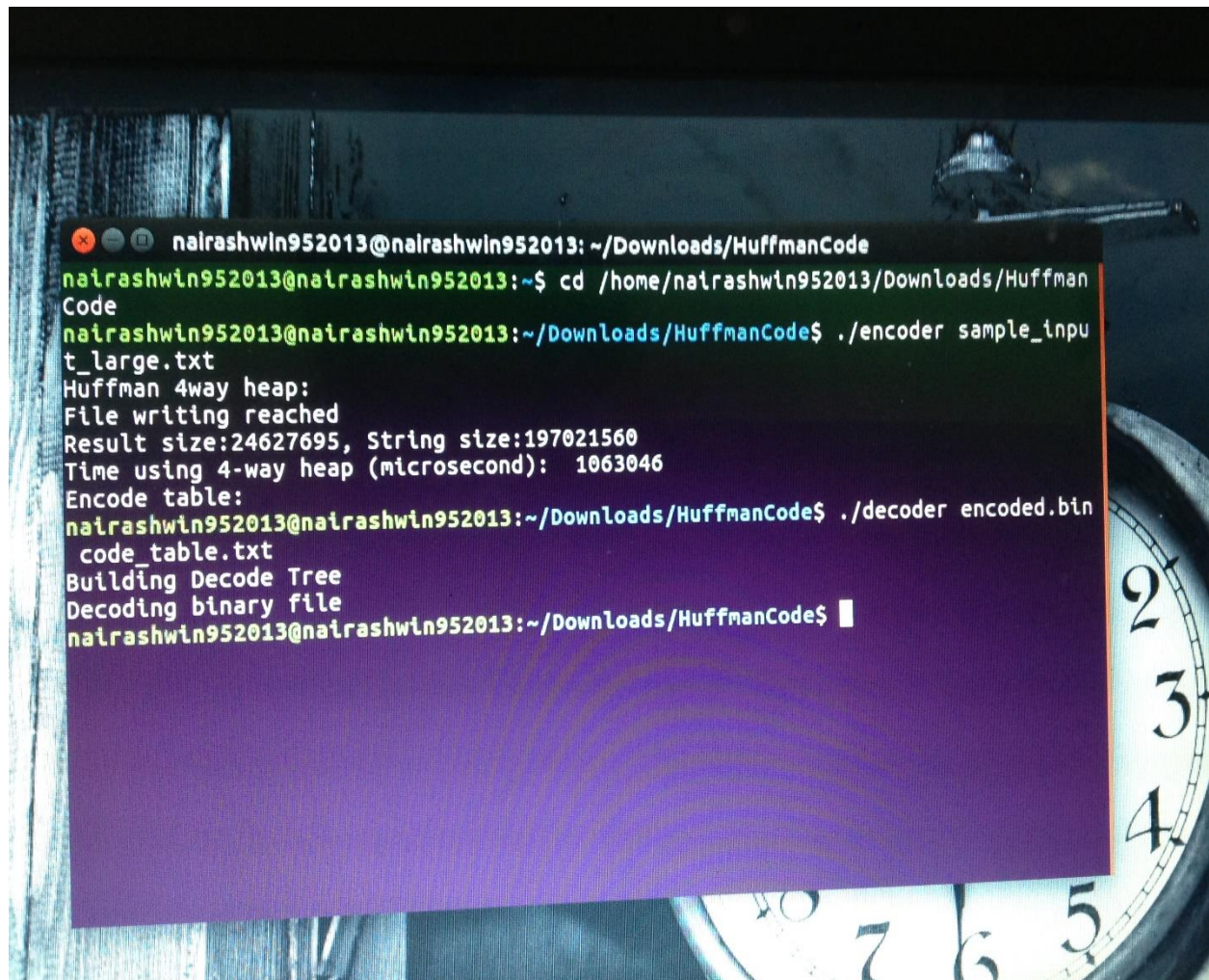
1. Input encoded.bin & code\_table.txt
2. The code table stores key value pairs with values (leaf nodes) and their associated Huffman codes.
3. Generate a Huffman decode tree.
  - a. Create nodes in the tree, if '1' is encountered then create a new node as right child  
else if '0' is encountered then create a new node as left child.
  - b. When the complete code length is traversed, store the value in node and terminate and make the node a leaf node of the Huffman decode tree
  - c. Repeat until all key value pairs of the code table are finished.
  - d. Complete Decode-Tree is successfully generated.
4. Move around Decode Tree using binary string till you reach a leaf and return node value.  
The node value is returned only if the node is a leaf node
  - a. If binary string is at '1', parse to the right child else if at '0' , then parse to the left child of the current node.
5. Store all returned values (int) and write them into a file called decoded.txt until the binary string is completely read.

### ANALYSIS: -

1. Since there is a balanced Huffman tree for the project that we were posed, The time complexity for decode tree construction will be  $O(n.h)$  time where  $h \rightarrow$  height of the tree;  $n \rightarrow$  order of size of input.
2. The "encoded. In" file will be read and converted to a binary string taking  $O(n)$  time.
3. The total time to decode the whole binary string will take  $O(n.h)$  time since for each occurrence of a key we traverse the entire tree.
4. File writing is in  $O(n)$  since the maximum length of each value in the decoded file will be six.
5. Total Complexity=  $O(n.h)$  time or  $O(n.\log n)$  time.



## WORKING SCREENSHOTS

A terminal window screenshot with a dark background and a clock face visible on the right side. The terminal shows the execution of a Huffman encoding and decoding program. The user is in the directory ~/Downloads/HuffmanCode. They run ./encoder sample\_input\_large.txt, which outputs statistics and an encode table. Then they run ./decoder encoded.bin, which outputs the decoded binary file. The terminal text is as follows:

```
nairashwin952013@nairashwin952013: ~/Downloads/HuffmanCode
nairashwin952013@nairashwin952013:~$ cd /home/nairashwin952013/Downloads/HuffmanCode
nairashwin952013@nairashwin952013:~/Downloads/HuffmanCode$ ./encoder sample_input_large.txt
Huffman 4way heap:
File writing reached
Result size:24627695, String size:197021560
Time using 4-way heap (microsecond): 1063046
Encode table:
nairashwin952013@nairashwin952013:~/Downloads/HuffmanCode$ ./decoder encoded.bin
code_table.txt
Building Decode Tree
Decoding binary file
nairashwin952013@nairashwin952013:~/Downloads/HuffmanCode$
```