Report

> *Luis Carbonell*
>
> *UFID: 9401-6887*

Function/Class Prototypes

Below is a list of Classes and Functions implemented in the source code.
The three data structures were initially implemented in using just integers as input, however once the highest speed data structure was determined, it was then modified to use the HuffmanNodes as inputs instead.

Encoder Script Methods

```
// Parses an incoming file and stores the result into an array
vector<int> parseFile(string filename);
// Converts a given string of 0's and 1's to a byte. requires input length of 8 bits
char convertStringToBin(string binaryString);
```

Decoder Script Methods

```
// Reads in an encoded file, finds the key from the Decod
```

```
e Tree and writes it to an output file
void writeByteToFile(string encoded_filename, DecodeTree
DT);
```

Class Prototypes

```
class EncodeTree {
private:
    HuffmanNode* root;
    FourWayHeap heap;
    string outFileName;
    map<int,string> code_table;


    HuffmanNode* buildTree(vector<HuffmanNode*> inputArra
y);
    // Build a complete EncodeTree given an input of and
array Huffman Nodes
    // Called by the constructor function
    void printSubTree(HuffmanNode* root, string output);
    // A recursive function to print out the given nodes
left and right nodes
    void updateCodeTable();
    // Updates the internal code table based on the curre
nt state of the EncodeTree
    void updateSubTree(HuffmanNode* root, string output);
    // A recursive function to update the code table valu
```

```
es of a given node and its left and right nodes
public:
    HuffmanNode* combineElements(HuffmanNode* first, Huff
manNode* second);
    // A public function to combine elements and returns
the resulting parent node
    void writeCodeTableToOutfile();
    // Writes the internal code table to an output file n
amed code_table.txt
    string getBinPathTo(int key);
    // Gets the binary path for the root node to a given
key, returns a string of values
    EncodeTree(vector<HuffmanNode*> inputArray, string ou
tFileName);
};
```

```
class DecodeTree {
private:
    HuffmanNode* root;
public:
    HuffmanNode* getRoot();
    // Returns the nodes at the root of a decode tree
    void addLeafAt(int value, string binaryLocation);
    // Adds a leaf to the decode tree at any given locati
on
    int getAt(string binaryLocation);
    // Returns the value at a given string location
```

```
    DecodeTree() {
        root = new (struct HuffmanNode);
    }
};
```

```
class FourWayHeap {
    private:
        vector<HuffmanNode*> heap;
        void buildHeap();
        int smallestChild(int hole);
        int parent(int i);
        int kthChild(int i, int k);
        void heapifyDown(int hole);
        void heapifyUp(int hole);
    public:
        FourWayHeap(){}
        bool isEmpty();
        void insert(HuffmanNode* x);
        HuffmanNode* popMin();
        HuffmanNode* Delete(int hole);
        int heapSize();
};
```

```
class BinaryHeap
{
    private:
        vector <int> heap;
```

```
        int left(int parent);
        int right(int parent);
        int parent(int child);
        void heapifyup(int index);
        void heapifydown(int index);
    public:
        BinaryHeap(){}
        void Insert(int element);
        void deleteMin();
        int ExtractMin();
        int Size();
};
```

```
class PairingHeap {
    private:
        PairNode *root;
        void compareAndLink(PairNode * &first, PairNode *
second);
        PairNode *combineSiblings(PairNode *firstSibling)
;
    public:
        PairingHeap();
        bool isEmpty();
        int &findMin();
        PairNode *Insert(int &x);
        void deleteMin();
};
```

Performance Results

Below are the performance results from a test of the three difference algorithms. Each run constructs 10 Huffman trees for each data structure, and reports the results.

```
$ ./test
Average time using binary heap (microseconds): 1117740
Average time using 4-way heap (microseconds): 807588
Average time using pairing heap (microseconds): 945825
```

The results show that the most efficient data structure for the given implementation is a 4 way heap. This can be explained by the fact that the number of nodes that are traversed are much less when a given node has four children as opposed to two.

Decoding Algorithm

The algorithm used to build the decoded tree is as follows:

- An entry from the code table is read from the `code_table.txt` file
- The binary path from the given entry value is followed from the root node of the DecodeTree
    - If the node for which the path follows does not exists, a

new node is created and added to the tree at that location
    - If the node already exists, move to the next node and continue along the path
- Once the binary path is completed, a leaf node is created with the key value from the entry in the code table