FULL SAIL UNIVERSITY

# Data Structures and Algorithms

# Lab 8: Huffman Compression

## Overview

Huffman is a lossless compression algorithm and is partially what the zip algorithm uses. It uses a frequency table to build a binary tree that is used to generate the encoded bit sequences. Depending on how many unique symbols are present in a file, the compressed data could take up as little as one bit, and never more than eight (the original size).

## Things to Review

- Binary file I/O
- Binary trees
- Arrays of class objects
- std::vector

## New Topics

- std::priority_queue
- BitFStream classes

## Data Members

### HuffNode

| | |
|---|---|
| **value** | ASCII value of the node, or -1 if not a leaf node |
| **freq** | Number of times this symbol occurs in the file |
| **left** | Pointer to the left child |
| **right** | Pointer to the right child |
| **parent** | Pointer to the parent |

### Huffman

| | |
|---|---|
| **mFilename** | Stores the name of the file (could be input or output, depending on the method) |
| **mFrequencyTable** | Array of 256 unsigned values to store how often any ASCII symbol appears in an uncompressed file |
| **mLeafList** | std::vector that stores the address of all leaf nodes |
| **mRoot** | Pointer to the top of the tree |
| **mEncodingTable** | Array of 256 vectors containing bools.<br>Each index of this will store the "bitcode" for an individual ASCII value |

## Methods

*The HuffNode struct is already implemented. You will use the constructor when creating Nodes in several places. The HuffCompare struct is also fully implemented, and will be used in the* GenerateTree *method.*

## Huffman Constructor

- Assign the appropriate data member the value from the parameter
- Zero out the entire frequency table
- Set the root to a value that indicates the tree is currently empty

## GenerateFrequencyTable

- Open the file in binary mode, using a **std::ifstream**
- Read the file one byte at a time, and increment the corresponding index
    - *The indices of the frequency table line up with the ASCII values*
- Close the file when complete

## GenerateLeafList

- Iterate through the frequency table and dynamically create a leaf node for each non-0 frequency
- Add each node to the mLeafList vector

## GenerateTree

- Create the priority_queue
    - This will be storing **HuffNode*** in a vector, and uses **HuffCompare** for the comparator
- Populate the priority_queue with the data in the leaf list
- Generate the tree with the following algorithm
    - While the queue has more than 1 node
        - Store the top two nodes into some temporary pointers and pop them
        - Create a **new** parent node with first node as the left child, and second node as the right child
        - Set the parent's value to -1, and the frequency to the sum of its two children's frequencies
        - Set the first and second nodes' parent to the newly created node
        - Insert the new node into the queue
- Set the root data member
    - *There is only one node in the queue*

## ClearTree

- Perform a post-order traversal to delete all of the nodes
    - *Same as Clear from the BST*

## Destructor

- Clean up all of the dynamically allocated memory *(There's a method to help with this)*

## GenerateEncodingTable

- Go through all of the leaf nodes and generate the bit codes
- This is done by traversing up the tree from each leaf node with a temporary pointer, and storing the direction in the corresponding vector
    - *Each index of the encoding table aligns to an ASCII value*
- As you move up, push a 0 to the vector if you passed through a left child connection, and a 1 if you passed through a right connection
- Once you hit the root node, **reverse** the values in the vector

## Compress

In this method, **mFileName** is the file to compress, and the parameter is the name of the file to write to.

- Create the frequency table, leaf list, tree, and encoding table by calling the existing methods (in this order)
- Create a **BitOfstream** and supply it the Huffman header
- Open the input file in binary mode with a **std::ifstream**
- Compress the file
    - For each byte in the original file, write out the corresponding bit-code from the encoding table
- Close both streams

## Decompress

In this method, **mFileName** is the file to decompress, and the parameter is the name of the file to write to.

- Create a **BitIfstream** and read the frequency table
- Create the leaf list and tree by calling the existing methods (in this order)
- Create a **std::ofstream** for output in binary mode
- Create a bool to use for traversing down the tree, and an unsigned char for writing to the file
- Create a HuffNode pointer for use in traversing the tree (start at the top)
- Go through the compressed file one bit at a time, moving the temporary pointer down the tree

    - When a leaf node is reached, write the value to the uncompressed file, and go back to the root
        - *This will need to be done a number of times equal to the total frequency of the original file*
- Close both streams