`

COMSATS UNIVERSITY, ISLAMABAD

Data Structure and algorithm

PROJECT

|  |  |
| --- | --- |
| Group Members | Muhammad Momin,  Zeeshan Haider Khan  Hamza Tariq  Muhammad Faizan Ahmed |
| Class | BS(Software Engineering) |
| Roll Number | FA18-BSE-060  FA18-BSE-103  FA17-BSE-049  FA16-BSE-138 |
| Subject | Data Structure and Algorithm |
| Teacher | Sir Tanveer Ahmed |

Contents

[1. Introduction 3](#_Toc27775840)

[1.1 Problem Statement 5](#_Toc27775841)

[1.2 Scope of Project 5](#_Toc27775842)

[2. Project Description 5](#_Toc27775843)

[2.1 For Compression 6](#_Toc27775844)

[2.2 For Decompression 6](#_Toc27775845)

[3. Functionalities 7](#_Toc27775846)

[F1: calcHeapLen() 7](#_Toc27775847)

[F2: compression() 7](#_Toc27775848)

[F3: create\_huffNode() 8](#_Toc27775849)

[F4: create\_pixNode() 8](#_Toc27775850)

[F5: discard\_zeros() 8](#_Toc27775851)

[F6: fillHeap\_blank() 9](#_Toc27775852)

[F7: getMin() 9](#_Toc27775853)

[F8: heapify\_up() 9](#_Toc27775854)

[F9: insert() 9](#_Toc27775855)

[F10: isLeaf() 10](#_Toc27775856)

[F11: minHeapify() 10](#_Toc27775857)

[F12: parent() 10](#_Toc27775858)

[F13: readImageBMP() 10](#_Toc27775859)

[F14: readTextFile() 11](#_Toc27775860)

[F15: toHuffmanTree() 11](#_Toc27775861)

[4. Algorithms 12](#_Toc27775862)

[Algo1: toHuffmanTree() 12](#_Toc27775863)

[Algo2: discard\_zeros() 12](#_Toc27775864)

[Algo3: parent() 13](#_Toc27775865)

[Algo4: printHuffmanCodes 13](#_Toc27775866)

[Algo5: heapify\_up() 13](#_Toc27775867)

[Algo6: minHeapify() 14](#_Toc27775868)

[Algo7: getMin() 14](#_Toc27775869)

[5. Test Plan 14](#_Toc27775870)

# Introduction

Huffman coding is one of the basic compression methods that have proven useful in image, text and video compression standards. When applying Huffman encoding technique on an Image, the source symbols can be either pixel intensities of the Image, ASCII of a character, or the output of an intensity mapping function. It is evident that, not all pixel intensity values may be present in the image and hence will not have non-zero probability of occurrence. For this, encoding of only required pixels is done using Huffman Algorithm for Compression.

The concepts used in order to implement the Huffman Compression methodology are:

* Heap( Min-Heap) using Array
  + Insertion
  + Heapify( Min-Heapify)
  + Deletion
  + Getting the Minimum Item and Rearranging the Heap
  + Discard of Zero-Frequency Values using Heap
* Binary Tree
  + Development of Huffman Tree
  + Tree Traversing for Generating Prefix Codes
* Image Reading in C++
  + Reading BMP Files with BGR Pixel Values
  + Conversion of BGR to Greyscale Values
  + Reading Width and Height of the Image
  + Filling the Heap Array using Pixel Values
* Text Reading in C++
  + Reading the Text File with ASCII values of Character
  + Filling the Heap Array using Character ASCII values
* Text Writing in C++
  + Writing a Text File using ASCII Codes

**Concept-1: Heap:** Heap is used as the building block of this Project. The Worst-Case and Average-Case complexity of Heap for Insertion and Deletion (O(log n)) makes the Compilation of Results fast. We used Min-Heap for the arrangement of Frequencies of Pixels and also we used it for getting the minimum two numbers in O(1) Complexity. Heap in this project is used to develop the Huffman tree. There were other options like Linked List implemented Priority Queue which increases the Complexity and also sorting again and again using sorting techniques makes the program burden for CPU.

**Concept-2: Binary Tree:** The most fundamental building block of this Project is based on Binary Tree without Binary Tree the project wouldn’t have been completed. We used Binary Tree for Development of Huffman Tree by taking the two minimum frequency items from heap and appending them into the tree and making new inner node with the frequency of two minimum items derived until items in heap are equal to one. We used binary tree to derive the prefix codes for leaf node by traversing from root to leaf nodes using recursion and making the path in binary .i.e. ‘0’ for left, ‘1’ for right.

**There are other Tree methods also but for our knowledge level this was easy for us to implement.**

## 1.1 Problem Statement

Increase of Image Quality has increased the Size of Images, for this Huffman Compression Method is implemented to minimize the size of Image to save the memory.

## 1.2 Scope of Project

Compression of Video Files, Other Image files which includes Multiple Dimension Colors are difficult at our level of knowledge. For this, this project is Limited to Text and Image Compression and within Image Compression the compression of BMP files. Perfection to achieve the most correct results for compression needs deep knowledge and grip on implementation of data structures. As for our level, expectation from project of unimaginable minimization of time complexity is nothing but a mere expectation. Verification of Compressed Image is undeniably a hurdle for this project as writing to bytes using bits into file and that for an Image is work of pure high level programming skills which we lack and also not the fundamental part of this project. Expectation of correct logic implementation is acceptable. Verification of the logic implemented can be viewed by compressing a text file.

# Project Description

This project is core implemented on Huffman Compression. The algorithm implemented is work of logic learned in Data Structure Class. Verification of Image Compression is Difficult at this level hence it is tested on text and also the “.pgm” file which is image file but in text format. Both the tests have shown the correct results. This project also shows the total number of bits and compressed number of bits. Decompression is done within one run of the code from the tree built for compression. No other space in files are taken for the table saving. The fundamental thing about this project was to implement the logic not the file handling but these are extra skills learned during the project that is why this project may contain some very small amount of code taken from stackOverFlow for file handling.

The building block of this project is as follows:

## For Compression

1. Read the Image pixels and Increment the frequencies of frequency items in array of length 256
2. Use heap and heapify the array according to frequencies
3. Get the two minimum values from the heap using getMin() function
4. Make a node which contains a reference to two previously taken minimum values nodes and create new node with frequency value as sum of frequencies of the two nodes taken
5. Insert the new node into the heap and heapify it.
6. Do above two procedures until there is only one item remaining in heap which will contain the address to root of the tree
7. Generate prefix codes by traversing from root to each node by taking empty string and appending “1” for right and “0” for left one by one and adding the generated string to heap array with index equal to leaf node pixel value. Do until you have all prefix codes.
8. Read the pixels in image again one by one but on every increment get the prefix code from heap array of index pixel value and write it to file. Do this until you have reached the end of the file.
9. By this the file will contain all the compressed data with ones and zeros

## 2.2 For Decompression

1. Read the file character one by one
2. Check if the character is “1” the move right in the tree else move left in the tree
3. Check if the node you are on is leaf node or not
4. If it is a leaf node print the leaf node pixel value
5. Else repeat 1,2 & 3
6. Do all above steps until the end of the file

# Functionalities

## F1: calcHeapLen()

|  |  |
| --- | --- |
| Function Prototype | calcHeapLen(); |
| Function Description | Calculates the length of heap It calculates the length of the table for all distinct objects in the heap |
| Pre-Condition | Last index li and starting index si of the heap list is initialized |
| Post-Condition | Returns the li-si+1 which is length of list or heap |

## F2: compression()

|  |  |
| --- | --- |
| Function Prototype | compression() |
| Function Description | Contains set of methods applied to compress a file with this method it also contains the UI for the project. It contains all the steps in order to compress a text or BMP image file. This method return huffNode which is address to root of the Huffman tree. |
| Pre-Condition | noOfBits, totalNoOfBits, readTextFile(), readImageBMP,heap[] methods and items are initialized and defined |
| Post-Condition | Does all the procedure required for compression and returns the tree root of Huffman tree |

## F3: create\_huffNode()

|  |  |
| --- | --- |
| Function Prototype | create\_huffNode(int pixel,huffNode\* left, huffNode\* right) |
| Function Description | makes a huffNode with pixel , left node and right node as argument. |
| Pre-Condition | Struct huffNode{} has been defined |
| Post-Condition | returns a huffNode with pixel , left node and right node as argument. |

## F4: create\_pixNode()

|  |  |
| --- | --- |
| Function Prototype | create\_pixNode(int pixel,int freq,huffNode\* node) |
| Function Description | creates pixNode with pixel, frequency and address to huffNode which is address to tree elements. |
| Pre-Condition | Struct pixNode{} has been defined |
| Post-Condition | returns pixNode with pixel, frequency and address to huffNode |

## F5: discard\_zeros()

|  |  |
| --- | --- |
| Function Prototype | discard\_zeros() |
| Function Description | Discards all the zero frequency elements from a heap, which makes the tree ambiguous by eliminating the items which as frequency 0. |
| Pre-Condition | Heap[], heapify\_up() are initialized and defined |
| Post-Condition | Makes heap 0 frequency items free |

## F6: fillHeap\_blank()

|  |  |
| --- | --- |
| Function Prototype | fillHeap\_blank() |
| Function Description | Fills the heap by assigning null items to the heap elements to keep the program from errors. |
| Pre-Condition | heap[], si and li are initialized |
| Post-Condition | Makes heap pure empty by assigning null and -1 items |

## F7: getMin()

|  |  |
| --- | --- |
| Function Prototype | getMin() |
| Function Description | used to get the least frequency item |
| Pre-Condition | hea[], si, li and minHeapify() are initialized and defined |
| Post-Condition | returns the miniumum frequency item from the heap |

## F8: heapify\_up()

|  |  |
| --- | --- |
| Function Prototype | heapify\_up() |
| Function Description | heapifies the item into heap by moving to the real place of the item according to heap concept |
| Pre-Condition | Heap[], si, li and parent() are initialized and defined |
| Post-Condition | Heapifies the item by swapping until it is at the right place |

## F9: insert()

|  |  |
| --- | --- |
| Function Prototype | insert(pixNode newNode) |
| Function Description | inserts the node into heap and heapifies it to it’s exact place into heap |
| Pre-Condition | Heap[],si,li and struct pixNode{} has been initialized and defined |
| Post-Condition | Inserts at the right place in the heap according to frequency |

## F10: isLeaf()

|  |  |
| --- | --- |
| Function Prototype | isLeaf(huffNode\* node) |
| Function Description | Checks if a node is leaf node or not |
| Pre-Condition | Heap[],si,li and struct huffNode{} has been initialized and defined |
| Post-Condition | Returns true for leaf node else will return false |

## F11: minHeapify()

|  |  |
| --- | --- |
| Function Prototype | minHeapify() |
| Function Description | Heapifies the whole heap list |
| Pre-Condition | Heap[],si,li and struct pixNode{}, heapify\_up() has been initialized and defined |
| Post-Condition | Places the items of heap list to their desired place according to heap conditions |

## F12: parent()

|  |  |
| --- | --- |
| Function Prototype | parent(int i); |
| Function Description | Finds the parent location of index as argument |
| Pre-Condition | No precondition |
| Post-Condition | Returns the parent location according to formula implemented |

## F13: readImageBMP()

|  |  |
| --- | --- |
| Function Prototype | readImageBMP(string filename) |
| Function Description | reads the BMP image by getting a pixel and incrementing the pixel frequency in the array of pixNodes which stores the frequency of the pixel |
| Pre-Condition | Heap[],si,li and struct pixNode{} has been initialized and defined |
| Post-Condition | Fills the heap list with frequencies of the pixels |

## F14: readTextFile()

|  |  |
| --- | --- |
| Function Prototype | readTextFile(string filename) |
| Function Description | reads the text file by getting the value of each character and incrementing in the pixNodes frequency of the ascii |
| Pre-Condition | Heap[],si,li and struct pixNode{} has been initialized and defined |
| Post-Condition | Fills the heap list with frequencies of the ASCII codes |

## F15: toHuffmanTree()

|  |  |
| --- | --- |
| Function Prototype | toHuffmanTree() |
| Function Description | Converts the heap into Huffman tree. By taking the two minimum items from the heap and making new huffNode and adding that huffNode again into heap until there is only one item and in the heap and returns the root of the tree. It is the bridge from heap to Huffman tree |
| Pre-Condition | Heap[],si,li and struct pixNode{},struct huffNode, create\_huffNode(), create\_pixNode() has been initialized and defined |
| Post-Condition | Returns root of the Huffman tree created from heap |

# Algorithms

## Algo1: toHuffmanTree()

//Descriptions of Functions are in above tables

**ALGORITHM** huffNode\* toHuffmanTree()

## Algo2: discard\_zeros()

**ALGORITHM** void discard\_zeros()

## Algo3: parent()

**ALGORITHM** int parent(int i)

Algo4: printHuffmanCodes**ALGORITHM** void printHuffmanCodes(huffNode\* root, int arr[], int top)

## Algo5: heapify\_up()

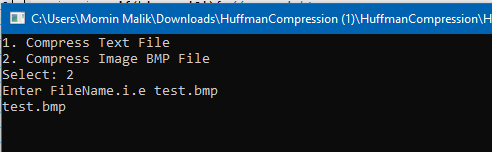
**ALGORITHM** void heapify\_up(int i)

## Algo6: minHeapify()

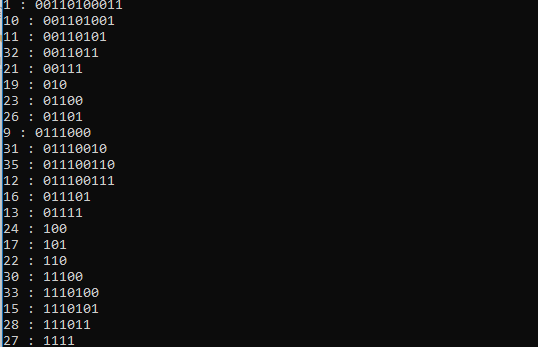
**ALGORITHM** minHeapify()

## Algo7: getMin()

**ALGORITHM** getMin()

1. Test Plan  
   Input of file

Output



|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

After file compression

