Parth Sethi

UFID - 72136121

ADS project report

Huffman tree was implemented with help of three priority queues :

1. Binary Heap
2. 4-way cache optimized Heap
3. Pairing Heap

So, the code to calculate time elapsed for iteration of 10 in milli seconds that was given in the document for project gave the times as

Binary Heap	1179
4-way cache optimized Heap	1332
Pairing Heap	2242

So, I decided go with binary heap to implement encoder and decoder.

Function Prototypes:

- **Encoder**

  The code consists of two structure first is the struct for a Huffman tree node, other is for the priority queue i.e. binary heap(which consists of size, capacity, and an array of its own type that means the queue is implemented with array having index going up to size, and capacity is used to create a new array of Huffman nodes with size = capacity.
  I have used C++ language for the project.

  Function prototypes:

```
min_heap* createminheap(unsigned capacity);
void swap(huffnode** a, huffnode** b);
void minHeapify(min_heap* minHeap, int index);
int isSizeOne(min_heap* minHeap);
huffnode* removemin(min_heap* minHeap);
void insertMinHeap(min_heap* minHeap, huffnode* minHeapNode);
min_heap* buildminheap(min_heap* minHeap);
int isLeaf(huffnode* root);
min_heap* createAndbuildminheap(int data[], int freq[], int size);
huffnode* buildHuffmanTree(int data[], int freq[], int size);
string findPath(huffnode* root, string path);
void printit(int data[],int size);
void HuffmanCodes(int data[], int freq[], int size);
```

Structure of program:

- Firstly data from test file is read number by number and frequency is updated as frequency [number fetched] = frequency [number fetched] + 1; and by default frequency array(size = 1000000) has default value zero, then every non zero frequency and corresponding data is stored in data array and frequency array.
- From main function HuffmanCodes function is called in with the frequency table and data table as parameters;
- HuffmanCodes calls buildHuffmanTree function which calls createAndbuildminheap function to actually create a binary heap and then applies the greedy algorithm as explained in class to create a Huffman tree and return that Huffman tree.
- The in HuffmanCodes function, findPath function is called which actually stores Huffman code for every data in an array of string, this I have done with the simplest example of hashing with hash function f(x) = x.
- Then in main printit function is called which actually makes the code table with help of the data table and string array that was created on last step.
- Afterwards in main there is a while loop that goes through sample test file line by line and store the code for that number in a bitset of 8 bits and that bitset is printed in encoded.bin.
- Removemin function removes the minimum element from binary tree and returns the Huffman node with minimum frequency, isSizeOne checks if there are only two elements left in heap. Swap function just swaps two Huffman nodes with each other, minheapify function heapifies the binary heap after entry or exit of any node in or from heap.

Performance Analysis and Explanation:

As time for iteration of 10 to create Huffman tree was minimum for binary heap, i.e. = 1179 microseconds according to code supplied on project description.
And times for reading and storing data into data and frequency took 2.383 seconds for large input file.
Building Huffman tree (including making minimum heap and greedy algorithm) and building the string array (or the hash map) took time = 2.004 seconds.
Printing into the code_table.txt took 0.8 seconds
And printing into encoded.bin took 10.37 seconds,
with a total of 15.6 seconds for successful implementation for the encoder with two output files encoded.bin size 23.4 MB and code_table.txt size 27.5 MB.
As I used recursive functions for creation of my trees so creation part took O(n*log n) time but for printing in code_table.txt and encoded.bin I used a loop so for printing the time is of order O(n)
And I could not directly string print ones and zeros into encoded.bin as it would make its size so large. So I used bitset of 8 bits so that I can print 8 bits or 1 byte at once into encoded.bin as a char whose size is also 8 bits. And code in encoded.bin will always be

multiples of 8. And printing into code_table took less time because only data table was printed, compared to encoded.bin where every iteration of number was also printed so it took most of the time also string was converted to byte, that added more time to encoded.bin printing.

- **Decoder**

  In decoder I took in byte by byte from encoded.bin into memblock and then converted that to bitset(b), and then converted that into string c and appended it into itself
  (c = c + b.tostring)
  Now all code bits are inside string c, so I used a while loop and crossed every bit in c and compared if it was 1 goto index 2n+2 else 2n+1 in code array and if value is not -1 in code array print the number into decoded.txt.
  Implementation of decode tree: made an array of 100000000 and assigned a default value of -1 to each, then read from code_table.txt and read the number and its code and traversed code if encountered 1 go to index 2n+2 else 2n+1 always starting from 0 for a new number and at the end replace the -1 with number belonging to specific code, this gave a code array which has a decoded Huffman tree.
  Everything was done inside main there were no functions used.
  Making of decode tree took 13 seconds, and printing into decoded.txt took 10.48 seconds
  And so it took a total of 24 seconds for successful run of decoder.

  So it uses 2 nested loops for traversing through code_table file and other for putting the number value for that specific code, so its complexity is $O(n^2)$
  There is one more loop to append all the binary code of encoded to a string c, hence this lop has complexity has $O(b*n)$ b is number of bits in encoded
  Also it uses other loop to go through every bit in encoded.bin and decodes the number for given code so its complexity is $O(b*n)$ where b is the number of bits in encoded.bin or the size of string with all codes appended into it.

  Hence total complexity is $O(n^2)+O(b*n)+O(b*n) = O(n^2)$