Lab
CS124

Specification

Huffman encoding is an algorithm devised by David A. Huffman of MIT in 1952 for compressing text data to make a file occupy a smaller number of bytes. Normally text data is stored in a standard format of 8 bits per character using ASCII encoding. It maps every character to a binary integervalue from 0-255. The idea of Huffman encoding is to abandon the 8-bits-per-character requirement and use different-length binary encodings for different characters. The advantage of doing this is that if a character occurs frequently in the file, such as the common letter 'e', it could be given a shorter encoding,making the file smaller. The trade off is that some characters may need touse encodings that are longer than 8 bits, but this is reserved for characters that occur infrequently, so usually the extra cost is worth it.

The purpose of this lab is to create a program that can encode and decode a text file using Huffman encoding.

Design

One structure and five functions are created to implement Huffman encoding:

HNode structure:

Structure attributes:

- c: the character that appears in the file
- weight: the frequency of this character in the text
- l: left child of the node
- r: right child of the node

Overloaded operators:

- <: compare the weight of 2 HNodes; return true if the first HNode's weight is larger than the second's.
- >: compare the weight of 2 HNodes; return true if the first HNode's weight is smaller than the second's.
- ==: compare the weight of 2 HNodes or 1 Hnode and 1 integer; return true if they are equal.
- !=: compare the weight of 2 HNodes; return true if they are not equal.
- +: add the weight of 2 HNodes.
- *: multiply the weight of a HNode to an integer.

Constructors:

- The default constructor initializes w and c to 0 and sets l and r to null pointer.
- The constructor that takes an integer and sets w to that integer. c is set to 0, and l and r are null pointer.
- The constructor that takes one integer and one character sets to the integer and c to the character. l and r are null pointer.

Functions:

- buildHuffmanTree creates and returns a Huffman tree.

  parameter: a string that stores the text that we want to encode

  return: a HNode that is the root of the Huffman tree

- freeMemory frees the memory used to allocate HNodes for the Huffman tree.

  parameter: HNode that is the root of the tree

  return: void

- traverse traverses through the Huffman tree and stores the characters and their code into a map.

  parameter: A pointer to HNode that points to the root of the subtree, a string that represents the base code, and a map that is used to store pairs.

  return: void

- encode generates a encoded file of a string.

  parameter: a HNode that is the root of the tree and a string that we want to encode.

  return: void

- decode is used to decode an encoded file using a given Huffman tree.

  parameter: a HNode that is the root of the Huffman tree.

  return: void

Implementation

Functions

```
#pragma once
#include <sstream>
#include <bitset>
#include <fstream>
#include <map>
#include <string>
#include <iostream>
#include "binaryHeap.h"
#include "arrayQueue.h"
#include "hnode.h"
```

```
HNode buildHuffmanTree(std::string s)
{
```

create the Huffman tree
will be called after some file has been
read into a string

bh: binary heap used to create Huffman tree
m: map used to count the frequency of each character in the string

```
BinaryHeap<HNode> bh;
std::map<char, int> m;
```

count the weight of each character

```
for(auto e : s)
    m[e]++;
```

create HNode for each character and add to the binary heap

```
for (auto e : m)
{
    HNode n(e.first, e.second);
    bh.add(n);
}
```

while there are more than 1 node in the binary heap

```
while(bh.size()>1)
{
```

pop out and store the 2 smallest weighted HNode in the binary heap and create an internal node that has the sum of u1's and u1's weight

```
HNode u1 = bh.remove();
HNode u2 = bh.remove();
HNode internal(u1 + u2);
```

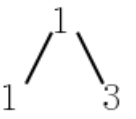
allocate memory to store u1 and u2

```
HNode* n1 = new HNode(u1);
HNode* n2 = new HNode(u2);
```

connect the 3 nodes by making n1 the left child of internal and n2 the right child of internal.

```
internal.l = n1;
internal.r = n2;
```

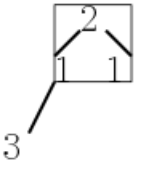
Example: (The number represents weight.)



remove the 2 nodes with the smallest weight

connect the two node using an internal node, which has a weight of their sum.

put the internal node back into the heap. repeat the steps above until there is only one node left in the heap.



add internal back to the binary heap

```
            bh.add(internal);
        }
```

Now, the Huffman tree is created, and we can remove the HNode in the binary heap, and that will be the root of the Huffman tree.

```
        return bh.remove();
    }
```

```
void freeMemory(HNode root)
{
```

free memory allocated for the HNodes using bf-traverse

create an array queue to store nodes and add the left and right child of the root to the queue

```
    ArrayQueue<HNode*> nodes;
    nodes.add(root.l);
    nodes.add(root.r);
```

while there are still nodes in the queue, remove it from the tree and add its left and right child to the queue if they are not null pointer. Then, delete the current node.

```
    while(nodes.size() != 0)
    {
        HNode* u = nodes.remove();
        if (u->l != nullptr)
            nodes.add(u->l);
        if (u->r != nullptr)
            nodes.add(u->r);
        delete u;
    }
}
```

```
void traverse(HNode* u, std::string code, std::map<char,
    std::string> &table)
{
```

traverse through the Huffman tree to generate a map for encoding the text file

If the left child of the current node is a leaf node, then add it to the map. Else, traverse to the next node(left child) and add 0 to the huffman code.

```
    if(u->l->c != 0)
        table.insert(std::pair<char, std::string>(u->l->c,
    code+"0"));
    else
        traverse(u->l, code+"0", table);
```

If the right child of the current node is a leaf node, then add it to the map. Else, traverse to the next node(right child) and add 1 to the huffman code.

```
    if(u->r->c != 0)
        table.insert(std::pair<char, std::string>(u->r->c,
    code+"1"));
    else
        traverse(u->r, code+"1", table);
}
```

encode the text file

```
void encode(HNode &root, std::string str)
{
```

create a map to encode the string using traverse. The left child starts with a code 0, and the right child will have 1 as the code.

```
    std::map<char, std::string> table;
    traverse(root.l, "0", table);
    traverse(root.r, "1", table);
```

translate the text file using the map that is created and save the result as a string of ones and zeros.

```
    std::ostringstream oss;
    for(auto c : str)
        oss << table[c];
    std::string converted = oss.str();
```

print out the huffman code representation of each character, the converted code, and its size. Note: the actual encoded file is larger than this because file size must be in whole bytes(8 bits), and there is also an integer at the front of the file to indicate the number of bits to read when decoding.

```
        for(auto e : table)
            std::cout << e.first << ": " << e.second << std::
        endl;
        std::cout << "length: " << converted.length() << " bits
         \ncode: " << converted << std::endl;
```

size: store the bits to read when decoding
Create an output stream to output a file, encode, and write size to the file

store into file

```
        int size = converted.length();
        std::ofstream ofs("encode", std::ofstream::binary);
        ofs.write(reinterpret_cast<char *>(&size), 4);
```

buffer: used to write a byte of code at a time
i: index used to get a certain character of the string

```
        char buffer;
        int i = 0;
```

while size is greater than 0

```
        while(size > 0)
        {
```

set buffer to 0
Then, shift the current buffer 1 bit to the left and add one bit from the string at a time.
Modular arithmetic is used to get the index and prevent out of bond exception in case that the total number of bits is not divisible by 8.

```
            buffer = 0;
            for(int j = 0; j < 8; j++)
            {
                buffer = buffer << 1;
                buffer += converted[i%converted.length()] - '0'
        ;
                size--;
                i++;
            }
```

write the code

```
            ofs.write(reinterpret_cast<char *>(&buffer), 1);
        }
        ofs.close();
    }
```

Example: adding the first 2 characters in "11001000"

00000000	shifting to the left
00000001	add the bit
00000010	shifting to the left
00000011	add the bit

decode a file

```
void decode(HNode &root)
{
```

ptr: a pointer pointing to the current node in the huffman tree
ofs: used to write the decoded file
ifs: used to read the encoded fine

```
    HNode* ptr = &root;
    std::ofstream ofs("decode", std::ofstream::binary);
    std::ifstream ifs("encode");
```

get number of bits needed to be decoded

```
    int size;
    ifs.read(reinterpret_cast<char *>(&size),4);
```

i: index used to get one bit from a bitset
buffer: used to write one character at a time

```
    int i;
    char buffer;
```

create an infinite outer loop

```
        while(1)
        {
```

read 1 byte from the input file and store into buffer. Then, create a bitset object with bits represented by buffer. i is set to 7 because the bits in bitset are arranged in a reversed order.

```
            ifs.read(reinterpret_cast<char *>(&buffer),1);
            i = 7;
            std::bitset<8> bs(buffer);
```

repeat the process for each bit(8 times)

```
            while(i >= 0)
            {
```

If the bit is not 0, move to the right child. Else, move to the left child.

```
                if(bs[i])
                    ptr = ptr->r;
                else
                    ptr = ptr->l;
```

If the current node is a leaf node, write the character to the output file and set ptr back to the root. Then, decrement i and size.

```
                if(ptr->c != 0)
                {
                    ofs.write(reinterpret_cast<char *>(&(ptr->c
      )), 1);
                    ptr=&root;
                }
                i--;
                size--;
```

If all the characters are processed, all bits are processed, and we jump to the finish label

```
                if(size <= 0)
                    goto finish;
            }
        }
```

close the files

```
    finish:
        ifs.close();
        ofs.close();
    }
```

HNode

```
#pragma once

struct HNode
{
```

```
    char c;
    int weight;
    HNode* l;
    HNode* r;
```

c: the character
weight: the frequency of char in text
l: left child
r: right child

```
    HNode() : c(0), weight(0){ l = r = nullptr; }
    HNode(int w): c(0), weight(w){ l = r = nullptr; }
    HNode(char ch, int w): c(ch), weight(w){ l  = r =
    nullptr; }
```

constructors: initialize the HNode in different ways

```
    bool operator<(HNode hn){ return weight < hn.weight; }
    bool operator>(HNode hn){ return weight > hn.weight; }
```

operator overloading: used to compare Hnodes or perform calculation

```
    bool operator==(HNode hn){ return weight == hn.weight;
    }
    bool operator==(int n){ return weight == n; }
    int operator+(HNode hn){ return weight + hn.weight; }
    int operator*(int n){ return weight * n; }
    bool operator!=(int n){ return weight != n; }
};
```