《数据结构》上机报告

2020 年 11 月 10 日

姓名： 林日中 学号： 1951112 班级： 10072602 得分：

实验题目	二叉树的应用实验报告 - 哈夫曼编码和译码	
问题描述	哈夫曼树又称最优二叉树，是一种带权路径长度最短的二叉树。所谓树的带权路径长度，就是树中所有的叶结点的权值乘上其到根结点的路径长度（若根结点为0层，叶结点到根结点的路径长度为叶结点的层数）。树的路径长度是从树根到每一结点的路径长度之和，记为$WPL=(W_1*L_1+W_2*L_2+W_3*L_3+\cdots+W_n*L_n)$，$N$个权值$W_i\ (i=1,2,...,n)$构成一棵有$N$个叶结点的二叉树，相应的叶结点的路径长度为$L_i\ (i=1,2,...,n)$。可以证明霍夫曼树的$WPL$是最小的。 请你理解最优二叉树，即哈夫曼树(Huffman tree)的概念，熟悉它的构造过程。	
基本要求	(1) 实现对 ASCII 字符文本进行 Huffman 压缩，并且能够进行解压。	
	已完成基本内容（序号）：	(1)
选做要求		
	已完成选做内容（序号）：	无
数据结构设计	在本次上机实验中，我设计了两个数据结构：struct Node 和 class HuffmanTree，分别表示哈夫曼树的结点和哈夫曼树。 struct Node 中有4个成员变量，它们的名称和功能分别为：char ch 存储了当前结点对应的字符；int freq 存储了该字符在整个字符串中出现的次数；Node *left 和 Node *right 分别存储了该结点的左、右子结点的地址，若无则为 nullptr。 class HuffmanTree 中有5个成员变量，它们的名称和功能分别为：std::unordered_map<char, std::string> huffmanCode 存储了每一个字符与对应的哈夫曼编码字符串一一对应的哈希表；std::string original_string, code_str, out_str 分别存储了原字符串、哈夫曼编码字符串和解码字符串；Node *root 存储了本哈夫曼树的根节点。本程序设计的哈夫曼树没有头结点。	

功能（函数）说明

```cpp
// A Tree node
struct Node
{
    char ch;
    int freq;
    Node *left, *right;
    Node(const char c = '\0', const int f = 0, Node *l = nullptr, Node *r = nullptr)
        : ch(c), freq(f), left(l), right(r) {}
    bool is_leaf() const { return !left && !right; }
}; // struct Node
```

struct Node 的构造函数传入四个参数，分别赋给成员变量 ch，freq，left 和 right。is_leaf()函数返回一个表示结点是否为叶结点的 bool 值。

```cpp
class HuffmanTree
{
private:
    std::unordered_map<char, std::string> huffmanCode;
    std::string original_str;
    std::string code_str;
    std::string out_str;
    Node *root = nullptr;
    // Comparison object to be used to order the heap; rule: highest priority item has lowest frequency
    struct comp···
    // traverse the Huffman Tree and store Huffman Codes in a map.
    void encode(Node *root, const std::string &str)···
    // traverse the Huffman Tree and decode the encoded string
    void decode(Node *root, int &index)···
    // Builds Huffman Tree and decode code_str
    void buildHuffmanTree()···

public:
    HuffmanTree(const std::string &t) : original_str(t) { buildHuffmanTree(); }
    ~HuffmanTree()···
    void delete_node(Node *&p)···
    std::string get_code_str() const { return code_str; }
    void display()···
}; // class HuffmanTree
```

在 class HuffmanTree 内，有一个结构体 struct comp 被定义，它是自定义的比较函数对象。各成员函数的名称和功能分别为：构造函数传入一个常 std::string 对象引用，将其值赋给成员 original_str，并开始构建哈夫曼树；encode(Node *, const std::string &)函数实现了在构建好哈夫曼树的结构后，生成各叶结点对应的哈夫曼编码字符串；decode(Node *, int &)函数实现了根据各字符的哈夫曼编码，重新创建并遍历哈夫曼树，得到解码字符串 out_str；buildHuffmanTree()函数包装了编码、解码的全过程，如构建优先队列，从优先队列中依次弹出元素构建哈夫曼树，等等；delete_node(Node *)函数配合析构函数完成对象的析构；get_code_str()函数返回编码好的哈夫曼编码字符串；display()函数能够输出三个字符串，并输出压缩过程的信息（原字符串占用的空间、编码字符串占用的空间、压缩比率）。

```cpp
int test_my_huffman_tree()
{
    std::string str;
    std::cout << "Please enter a string : \n";
    std::getline(std::cin, str);
    HuffmanTree ht(str);
    ht.display();
    return 0;
}
```

test_my_huffman_tree()函数封装了测试哈夫曼树压缩、解压缩字符串功能的操作：从缓冲区中整行读入一个字符串，经过哈夫曼树的压缩、解压缩后，输出相关信息。

开发环境

Windows 10, C++ language, Visual Studio Code with g++

调试分析

经过测试，本程序功能完好。原字符串与解码字符串相等。

本程序输出的所有文本如下：

```
Please enter a string :
Huffman coding is a popular method for data compression. It serves as the basis for several popular
programs run on various platforms. Some programs use just the Huffman method, while others use it as
one step in a multistep compression process. The Huffman method [Huffman 52] is somewhat similar to
the Shannon-Fano method. It generally produces better codes, and like the Shannon-Fano method, it
produces the best code when the probabilities of the symbols are negative powers of 2. The main
difference between the two methods is that Shannon-Fano constructs its codes top to bottom (from the
leftmost to the rightmost bits), while Huffman constructs a code tree from the bottom up (builds the
codes from right to left). Since its development, in 1952, by D. Huffman, this method has been the
subject of intensive research into data compression. ↙

Original string was :
Huffman coding is a popular method for data compression. It serves as the basis for several popular
programs run on various platforms. Some programs use just the Huffman method, while others use it as
one step in a multistep compression process. The Huffman method [Huffman 52] is somewhat similar to
the Shannon-Fano method. It generally produces better codes, and like the Shannon-Fano method, it
produces the best code when the probabilities of the symbols are negative powers of 2. The main
difference between the two methods is that Shannon-Fano constructs its codes top to bottom (from the
leftmost to the rightmost bits), while Huffman constructs a code tree from the bottom up (builds the
codes from right to left). Since its development, in 1952, by D. Huffman, this method has been the
subject of intensive research into data compression.

Huffman Codes are :
i : 11111
h : 11110
t : 1110
  : 110
o : 1011
H : 1000000
m : 10100
f : 01111
n : 0110
I : 100011011
c : 01010
p : 00010
e : 001
v : 0001111
, : 1000111
T : 00011100
y : 10000011
a : 0100
r : 0000
) : 00011000
s : 1001
u : 01110
d : 01011
S : 0001101
. : 1010101
F : 00011101
k : 1000110101
5 : 00011001
2 : 10000010
( : 101010001
w : 1000010
- : 10001100
g : 1000011
b : 100010
9 : 1000110100
j : 101010000
[ : 1010100100
] : 1010100101
D : 1010100110
1 : 1010100111
l : 101011

Encoded string is :
1000000011100111101111110100010001101100101010110101111111011010000111101111111001110010011000010101 1000
1001110101011010000001101010000111101111010110101111001111101100001100101101001110010011001010101110 10
0000100000000110011001111111011011010101011101000110111110110100100100000001111001100111001001001110 111
0111100011101000100100100111111100111001111101100001101001001000111100100000100101011110000101011000 10
0111010101101000000110000100000101110000110000010010100100111000000111001101101011011011000011110100 00
0011111101101110100111000010101011010011100111110110000101001001101010111000011011011101000011100001 00
0001011100001100000010010100100111001110100100111010101000001110100111101101110111100011101000000001 1100
1111011111010001000110110101000011110111101011010111000111101000010111101111111010110011101011111101 111
0001000001001110011101001001110111111111011001001001110101101100011101001111000100010110111111011011 00100
1101010001110101011111011111110011110001000101100101010111010000010000000110011001111111011011011000 010
0000101101010001100110011010101110000111001111000111010000000011100111101111110100010001101101010000 1111
0111101011010111101010100100100000000111001111011111101000100011011000011001100000101010100101110111 1110
0111010011011101000011000010111100100111011010011111110100111111010110100000011011101011110111011110 00
```

```
1110000110111111001000110011010110110100011000001110101000110101111010100001111011110101101011101010111
0100011011111011010000110010110001000000100101011101011100000111100001000000101101011011100101000110011 1
0100010001111011100010000110010101011010110011001100011111001000110010111101010111111111000110101001110
1110111100011100001101111110010001100110101101101000110000011101010001101011110101000011110111101011010
1110001111101111111101100001000000101101011011100101000110011101110111100011101000100011001111011001010
1011010110011101000010111100010110110111011110001110000100000101110001001001000101111110101111111111101
1111001100111010110111111011101111000111010011000001110100100010101110101110011100100000000011100110001
10000110100111011111100011110011100001010111000010001000010011101011011111110100000101010101110000111001
1110001110101000100111110110110010111111101111011110010000000101100101000111010001000111101000010001001
011011011101111000111011101000010101111010100001111011110101101011100111011111100111011101111001001110
11000011011111100100011001101011011010001100000111010100011010111100101010110110100111100000001110010101
1101001110111111111010011100101010110101100110011101110101100010110111010111101000101011111011101011101
0011010101000101111000010111010011011101111000111010101100101111111010100101110011110110111010111101111
0111100011100000111111000011111101110101001011100111101101000101111111101001000110001000111101000010 1
1110111111010110011101000000001110011110111111010001000110110010101011011010011110000000111001010111010 01
1100100110010101011010110011101110000000010011100111100001011101001101110111100011101000101011111011101
0111010011001110000101101010100011000100111011111101011010111001110111011110001110010101011010110 01100
1110011110000101110100110000011111100001111110111011011101011110101011001011111110000110001010101110 00
0110111111011001010001110111111111010011100101100100011110011010111011000101010000101101110100011111011
1110110110101010011110001101000000110011000000101000111110100010100000111101010100110101010111010000 0001
11001111011111010001000110100011111011101111011111100111010100001111011110101101011110111100100100111 0
1000100010010110110111011110001110100101110100010101010000000101010111011010110111111011111011011100010
1101001111110001111001110000000011001001010000000010101111011011111011011101011110010110100111001001100 1
010101110100000010000000011001100111111101101101010101110

Decoded string is :
Huffman coding is a popular method for data compression. It serves as the basis for several popular
programs run on various platforms. Some programs use just the Huffman method, while others use it as
one step in a multistep compression process. The Huffman method [Huffman 52] is somewhat similar to
the Shannon-Fano method. It generally produces better codes, and like the Shannon-Fano method, it
produces the best code when the probabilities of the symbols are negative powers of 2. The main
difference between the two methods is that Shannon-Fano constructs its codes top to bottom (from the
leftmost to the rightmost bits), while Huffman constructs a code tree from the bottom up (builds the
codes from right to left). Since its development, in 1952, by D. Huffman, this method has been the
subject of intensive research into data compression.

The original size is    : 850 byte(s)
The compressed size is  : 467 byte(s)
The compression rate is : 54.94%
```

综上，本实验设计的数据结构很好地完成了题目对哈夫曼树的功能的要求，功能完好，使用方便，并且能够较好地处理非法数据的输入并反馈相关错误提示；程序界面友好，具有比较恰当的人性化提醒，具有一定的鲁棒性。

心得体会

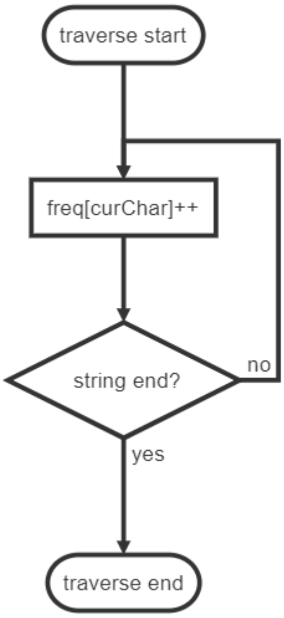
一、 实验总结

在本次上机实验中，我复习了理论课上学到的二叉树的定义和作用，将课本上哈夫曼树的链式存储结构和功能封装成了 `class HuffmanTree`，实现了针对字符串的哈夫曼编码和解码。

二、 性能分析

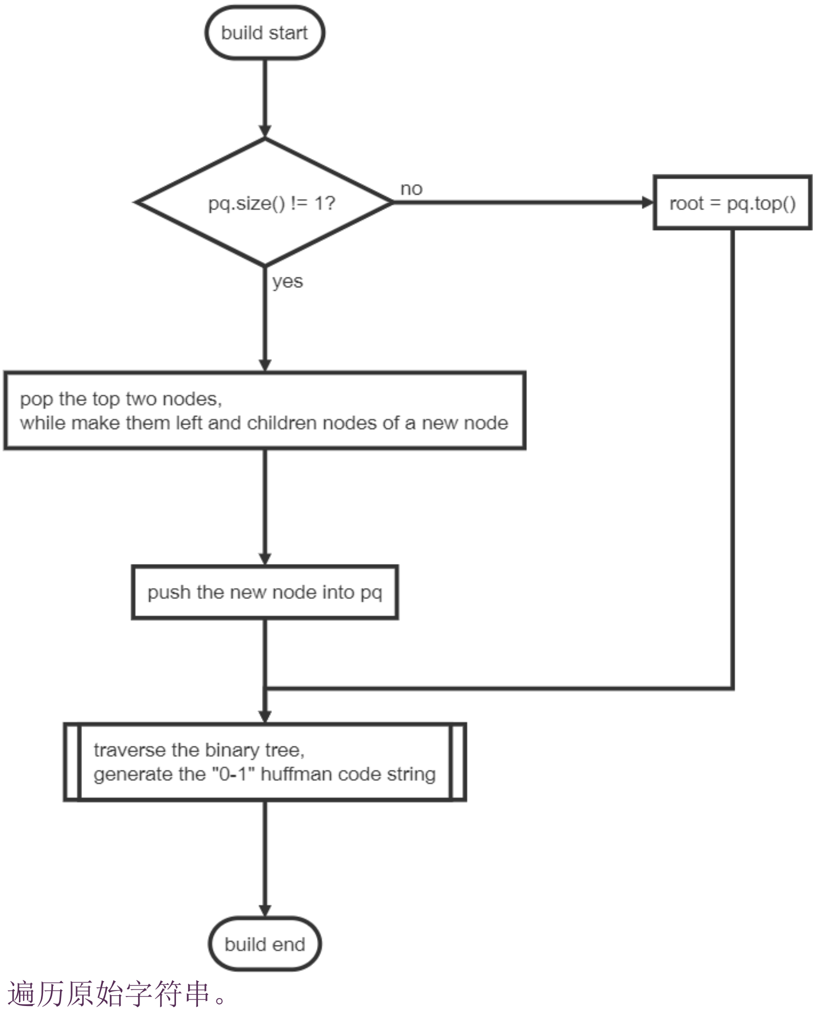
（1） 遍历原始字符串。

时间复杂度：$O(n)$。



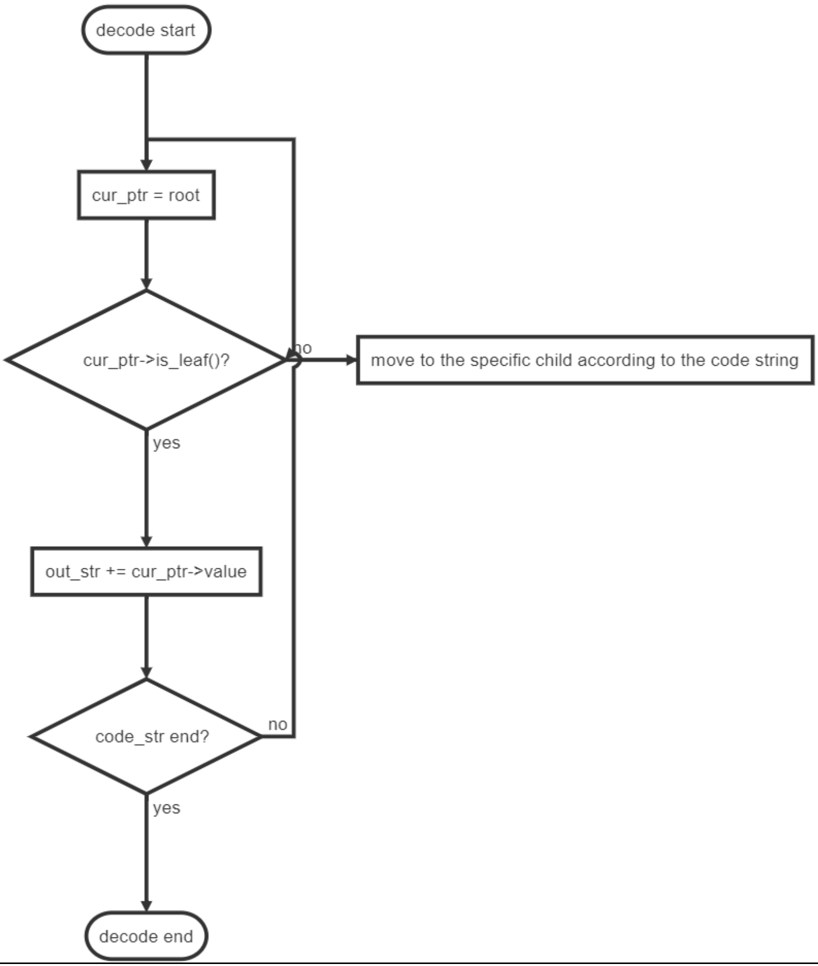
（2） 生成哈夫曼树和哈夫曼编码字典。

时间复杂度：$O(n\log_2 n)$。



（3） 遍历原始字符串。

时间复杂度：$O(n)$。对于每部分编码寻找其对应叶节点的平均时间复杂度为$O(\log_2 n)$，对于整段编码来说总的时间复杂度为$O(n\log_2 n)$。



源代码

```
#ifdef __GNUC__
#include <bits/stdc++.h>
#endif // __GNUC__
#ifdef _MSC_VER
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <iomanip>
#include <string>
#include <queue>
#include <vector>
#include <unordered_map>
#endif // _MSC_VER

// A Tree node
struct Node
{
    char ch;
    int freq;
    Node *left, *right;
```

```
    Node(const char c = '\0', const int f = 0, Node *l = nullptr, Node *r = nullptr)
        : ch(c), freq(f), left(l), right(r) {}
    bool is_leaf() const { return !left && !right; }
}; // struct Node

class HuffmanTree
{
private:
    std::unordered_map<char, std::string> huffmanCode;
    std::string original_str;
    std::string code_str;
    std::string out_str;
    Node *root = nullptr;
    // Comparison object to be used to order the heap; rule: highest priority item has lowest frequency
    struct comp
    {
        bool operator()(Node *left, Node *right) const { return left->freq > right->freq; }
    };
    // traverse the Huffman Tree and store Huffman Codes in a map.
    void encode(Node *root, const std::string &str)
    {
        if (!root)
        {
            return;
        }
        // found a leaf node
        if (root->is_leaf())
        {
            huffmanCode[root->ch] = str;
        }
        encode(root->left, str + "0");
        encode(root->right, str + "1");
    }
    // traverse the Huffman Tree and decode the encoded string
    void decode(Node *root, int &index)
    {
        if (!root)
        {
            return;
        }
        // found a leaf node
```

```
        if (root->is_leaf())
        {
            out_str += root->ch;
            return;
        }
        index++;
        decode((code_str[index] == '0' ? root->left : root->right), index);
    }
    // Builds Huffman Tree and decode code_str
    void buildHuffmanTree()
    {
        if (original_str.empty())
        {
            std::cerr << "The string is not found. Please try again. \n";
            exit(-1);
        }
        // count frequency of appearance of each character and store it in a map
        std::unordered_map<char, int> freq;
        for (char ch : original_str)
        {
            freq[ch]++;
        }

        // Create a priority queue to store live nodes of Huffman tree
        std::priority_queue<Node *, std::vector<Node *>, comp> pq;

        // Create a leaf node for each character and add it to the priority queue.

        for (auto &pair : freq)
        {
            Node *new_node = new Node(pair.first, pair.second);
            pq.push(new_node);
        }

        // do till there is more than one node in the queue
        while (pq.size() != 1)
        {
            // Remove the two nodes of highest priority (lowest frequency) from the queue
            Node *left = pq.top();
            pq.pop();
            Node *right = pq.top();
            pq.pop();

```

```cpp
            // Create a new internal node with these two nodes as children and with frequency equal to the sum of the two nodes' frequencies. Add the new node to the priority queue.
            Node *new_node = new Node('\0', left->freq + right->freq, left, right);
            pq.push(new_node);
        }

        // root stores pointer to root of Huffman Tree
        root = pq.top();
        pq.pop();

        // traverse the Huffman Tree and store Huffman Codes in a map. Also prints them
        encode(root, "");

        for (char &ch : original_str)
        {
            code_str += huffmanCode[ch];
        }

        // traverse the Huffman Tree again and this time decode the encoded string
        int index = -1;
        do
        {
            decode(root, index);
        } while (index < (int)code_str.size() - 1);
    }

public:
    HuffmanTree(const std::string &t) : original_str(t) { buildHuffmanTree(); }

    ~HuffmanTree()
    {
        delete_node(root);
    }
    void delete_node(Node *&p)
    {
        if (p->left)
        {
            delete_node(p->left);
        }
        if (p->right)
```

```cpp
        {
            delete_node(p->right);
        }
        delete p;
        p = nullptr;
    }
    std::string get_code_str() const { return code_str; }
    void display()
    {
        // print input string
        std::cout << "\nOriginal string was :\n"
                  << original_str << '\n';

        // print huffman code dictionary
        std::cout << "\nHuffman Codes are :\n";
        for (auto &pair : huffmanCode)
        {
            std::cout << pair.first << " : " << pair.second << '\n';
        }

        // print encoded string
        std::cout << "\nEncoded string is :\n"
                  << code_str << '\n';

        // print decoded string
        std::cout << "\nDecoded string is : \n"
                  << out_str << '\n';

        std::cout << "\nThe original size is    : " << original_str.length() + 1 << " byte(s) \n"
                  << "The compressed size is  : " << (int)ceil(code_str.length() / 8.0) + 1 << " byte(s) \n"
                  << "The compression rate is : " << std::fixed << std::setprecision(2) << 100 * (ceil(code_str.length() / 8.0) + 1) / (original_str.length() + 1) << "% \n";
    }
}; // class HuffmanTree

int test_my_huffman_tree()
{
    std::string str;
    std::cout << "Please enter a string : \n";
    std::getline(std::cin, str);
    HuffmanTree ht(str);
```

```
    ht.display();
    return 0;
}

int main()
{
    return test_my_huffman_tree();
}
```