《数据结构》上机报告

2019 年 11 月 2 日

姓名：李佳庚 学号：1852409 班级：计算机1班 得分：

实验题目	二叉树	
问题描述	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。 2. 程序要添加适当的注释，程序的书写要采用缩进格式 。 3. 程序要具在一定的健壮性，即当输入数据非法时，程序也能适当地做出反应，如插入删除时指定的位置不对等等。 4. 程序要做到界面友好，在程序运行时用户可以根据相应的提示信息进行操作。 5. 根据实验报告模板详细书写实验报告，在实验报告中给出主要算法的复杂度分析。	
	已完成基本内容（序号）：	1，2，3，4，5
选做要求		
	已完成选做内容（序号）	

数据结构设计	对于 huffman 树，设计有关它的数据结构的时候，要领会构建 huffman 树的步骤：
	第一步：
	根据输入的字符串，计算整个字符串中，各个字符出现的次数。
	如：“sakljdaljwoi”，其中：s,a,k,l,j,d,w,o,i 分别出现的次数要准确的计算出来。
	第二步：
	根据各个字符出现的次数，构建最优二叉树，即构建 huffman table。
	第三步：
	根据最优二叉树的结构，确定每个字母对应的编码，即 huffman code。
	第四步：
	根据编码表，将字符串中的字符一一对换为编码表中的“01 串”。
	第五步：
	利用位运算，将每 8 个二进制数压缩为 8bits/1byte 的长度。
	
	而对应的解码则是反过来求解：
	第一步：
	利用位运算，将被压缩为 bit 单位的“01 串”转化为方便处理的“01 字符串”。
	第二步：
	根据 huffman code 表，将“01 字符串”直接转化为原本的字符串。
	
	其实 huffman 解码比编码步骤少的最核心原因便是之前的 huffman code 表已经求出了。不需要再进行一次重复操作。
	所以可以看出，huffman 编码的关键便是准确求出 huffman code，而 huffman code 又需要通过 huffman tree 的叶子节点进行向根节点的回溯得到。
	
	数据结构的设计就是要关照这三个步骤。
	在一个 huffman class 中设置：weight_char_tb, hm_tb, hmcode
	`struct weight_char_tbNode { char ch; // 对应字符 int weight; // 权值 };`
	`class hm_tbNode { public: char content; int weight; int parent; int lchild; int rchild; };`
	`struct huffmanCode { char ch; // 对应字符 string code; // 编码 };`

```
class huffman {

public:
    struct weight_char_tbNode {
        char ch;    // 对应字符
        int weight; // 权值
    };

    class hm_tbNode {
    public:
        char content;
        int weight;
        int parent;
        int lchild;
        int rchild;
    };

    struct huffmanCode {
        char ch;    // 对应字符
        string code; // 编码
    };

};
```

我没有给 huffman 这个类写任何的操作。
因为这个类构造出来只是为了对 huffman 编码的三个过程进行一种抽象的表示。
这个类很好的体现了 huffman 编码的过程和次序。

功能（函数）说明

1. 权重计算函数：

```
pair<huffman::weight_char_tbNode*,int> weightCalcu(string strInput) { ... }
```

传入 string 类型的 input，通过对 input 的操作，计算出字符串中各个字符出现的次数。

```
pair<huffman::weight_char_tbNode*,int> weightCalcu(string strInput) {
    // 有多少不同的数字
    int tbNodeNum = 0;
    huffman::weight_char_tbNode *wctb = new huffman::weight_char_tbNode[strInput.length()];
    for (size_t index = 0; index < strInput.length(); index++) {
        if (strInput[index] != '\0') {
            wctb[tbNodeNum].ch = strInput[index];
            wctb[tbNodeNum].weight = 1;
            // count num
            for (size_t jndex = index + 1; jndex < strInput.length(); jndex++)
                if (strInput[jndex] == strInput[index]) {
                    wctb[tbNodeNum].weight++;
                    strInput[jndex] = '\0';
                }
            tbNodeNum++;
        }
    }
    return pair<huffman::weight_char_tbNode*, int>(wctb, tbNodeNum);
}
```

返回的是 wctb 这个“权重-字符对应关系表”以及表的大小。

2. Huffman table 生成函数：

```
pair<huffman::hm_tbNode *, int> WCtoHMTB(huffman::weight_char_tbNode *wctb, int num) { ... }
```

传入的是 wctb 和表的大小。

在其中，通过建立 huffman tree，将树静态表示为一张对应关系表。表中包含了对所有叶子节点和根节点的描述。

```
pair<huffman::hm_tbNode *, int> WCtoHMTB(huffman::weight_char_tbNode *wctb, int num)
    huffman::hm_tbNode *hmTb = new huffman::hm_tbNode[2 * num];
    int index = 1;
    hmTb[0] = { '\0', INT_MAX, 0, 0, 0 };
    for (huffman::hm_tbNode *work = hmTb + 1; index <= num; index++)
        hmTb[index] = { wctb[index - 1].ch, wctb[index - 1].weight, 0, 0, 0 };

    for (; index < 2 * num; index++)
        hmTb[index] = { '\0', 0, 0, 0, 0 };

    // create huffman table
    for (index = num + 1; index < 2 * num; index++) {
        int minIndex1 = 0, minIndex2 = 0;
        __selectMin2(hmTb, index - 1, minIndex1, minIndex2);
        hmTb[minIndex1].parent = hmTb[minIndex2].parent = index;
        hmTb[index].lchild = minIndex1;
        hmTb[index].rchild = minIndex2;
        hmTb[index].weight = hmTb[minIndex1].weight + hmTb[minIndex2].weight;
    }

    return pair<huffman::hm_tbNode *, int>(hmTb, index);
}
```

返回的是一张 huffman table 和表的大小。

3. Huffman code 生成函数：

```
huffman::huffmanCode *HMTBtoHC(huffman::hm_tbNode *hmtb, int num)
```

通过 huffman table，利用表中描述的最优二叉树。

从叶子节点开始，向根节点回溯。如果是其左孩子，就在前加 0，如果是有孩子就在前加 1。

```
huffman::huffmanCode *HMTBtoHC(huffman::hm_tbNode *hmtb, int num) {
    huffman::huffmanCode *hc = new huffman::huffmanCode[num + 1];
    for (int index = 1; index <= num; index++) {
        string work;
        for (int jndex = index, prev = hmtb[index].parent; prev != 0;
            jndex = prev, prev = hmtb[prev].parent) {
            if (hmtb[prev].lchild == jndex) work = '0' + work;
            else work = '1' + work;
        }
        hc[index].code = work;
        hc[index].ch = hmtb[index].content;
    }
    return hc;
}
```

返回的是生成出来的 huffman code 表。

4. Encode 函数：

```
string encode(huffman::huffmanCode *hc, int numOfHC, string str) { ... }
```

通过输入 huffman code 表和字符串，进行压缩操作。这里为了展示方便，用字符串进行输出：

```
string encode(huffman::huffmanCode *hc, int numOfHC, string str) {
    // to be continued

    string toReturn;
    for (size_t index = 0; index < str.length(); index++) {
        int jndex = 1;
        for (huffman::huffmanCode *work = hc; jndex <= numOfHC; jndex++)
            if (work[jndex].ch == str[index]) {
                toReturn.append(work[jndex].code);
                break;
            }
    }
    return toReturn;
}
```

返回字符串。

5. Decode 函数：

对应的解码和压缩是一个思路。代码实现基本相同：

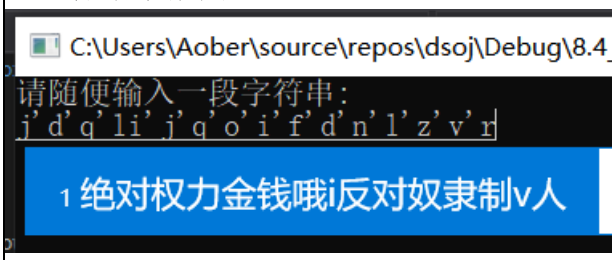
```
string decode(huffman::huffmanCode *hc, int numOfHC, string str) { ... }
```

```
string decode(huffman::huffmanCode *hc, int numOfHC, string str) {

    string strOutput;
    int strOutput_index = 0;

    for (size_t index = 0, jndex = 0; index <= str.length() && jndex <= str.length(); jndex++) {
        string strPool;
        for (size_t kndex = index; kndex < jndex; kndex++)
            strPool += str[kndex];
        // memcpy(strPool, str, jndex - index);

        int indexInHC;
        if (indexInHC = __isin(hc, numOfHC, strPool)) {
            strOutput += hc[indexInHC].ch;
            index = jndex;
        }
    }

    return strOutput;
}
```

开发环境	Win10 Microsoft Visual Studio Community 2017 15.9.3 Debug x86

调试分析

（运行结果截图）




```
下面进行权重计算，权重如下所示：
2 j
2 d
2 q
2 l
2 i
1 o
1 f
1 n
1 z
1 v
1 r
```

```
下面进行huffman table生成，如下所示：
1 2 j 15 0 0
2 2 d 15 0 0
3 2 q 16 0 0
4 2 l 16 0 0
5 2 i 17 0 0
6 1 o 12 0 0
7 1 f 12 0 0
8 1 n 13 0 0
9 1 z 13 0 0
10 1 v 14 0 0
11 1 r 14 0 0
12 2   17 6 7
13 2   18 8 9
14 2   18 10 11
15 4   19 1 2
16 4   19 3 4
17 4   20 5 12
18 4   20 13 14
19 8   21 15 16
20 8   21 17 18
21 16   0 19 20
```

```
遍码如下：
00000101001110000001010101001011001110001111011111011111
解码如下：
jdqlijqoifdnlzvr
```

心得体会	（对整个实验过程做出总结，对重要的算法做出性能分析。） 我总结出了 Huffman 树的核心思想 不是什么简简单单的一句“生成最优二叉树” 不是用“01 串”表示字符 不是通过回溯 huffman tree 生成 huffman table 这些都不是 最核心最核心的，应该是 huffman 编码能够做到压缩文件大小的原因： 那就是，用更少的 bit 表示出现次数多的字符，用较多的 bit 表示出现次数少的字符， 以此在看似“平衡”的情况下“压缩”，从而得到稳定压缩。