离散数学课程实验说明文档

——最优二元树的应用

作 者 姓 名： 马威

学 号： 2151294

指 导 教 师： 唐剑锋

学院、 专业： 软件学院 软件工程

****

目 录

[1 项目分析 1](#_Toc495668153)

[1.1 项目背景 1](#_Toc495668154)

[1.2 项目要求 1](#_Toc495668155)

[2 项目设计 2](#_Toc495668156)

[2.1 数据结构设计 2](#_Toc495668157)

2.2 类设计 2

2.2.1 结点类（HuffmanNode） 2

2.2.2 哈夫曼树类（HuffmanTree） 2

2.2.3 字符串类（String） 3

2.3 算法设计 6

2.2.1 算法思路 6

2.2.2 性能评估 6

2.2.3 寻找Huffman编码算法 7

·流程图表示 7

·构造Huffman树代码实现 7

·寻找Huffman编码代码实现 8

·项目主体部分代码实现 9

[3 项目测试 1](#_Toc495668161)1

[3.1 项目示例 1](#_Toc495668157)1

[3.2 仅有一个结点的情况 1](#_Toc495668157)1

[3.3 输入错误处理 1](#_Toc495668157)2

4 心得体会 12

**1.项目分析**

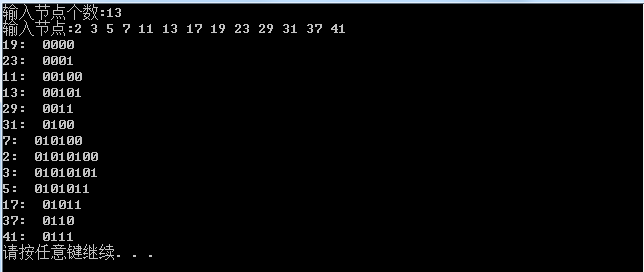
1.1 项目背景

在通讯上，选择编码是重要的一环，如何合理地将需要传达的信息按照某一规律进行转化，是送信者需要考虑的问题。一个很好的解决方法，是根据字符的出现频率将字符转为01串，同时每一个01串不会是其他所有01串的前缀（避免产生歧义），再调整每一种01串的长度，使得在更小的长度期望中将信息表达完整。这正是有名的Huffman编码。

1.2 项目要求

输入一组通信符号的使用频率，求各通信符号对应的前缀码。

项目示例：



**2.项目设计**

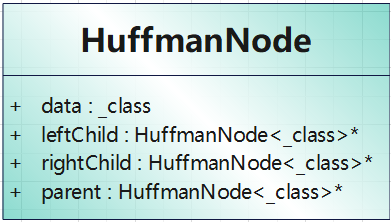
2.1 数据结构设计

由项目分析可以得出，该项目需要完成一棵Huffman树的生成，根结点的权值是每个结点的出现频率。因此需要设计一个Huffman树的类。此外，哈夫曼编码需要随着在Huffman树中遍历的过程进行改变，可以用字符串类进行存储，再通过重载运算符的方式进行其它操作。

2.2 类设计

**2.2.1 结点类（HuffmanNode）**

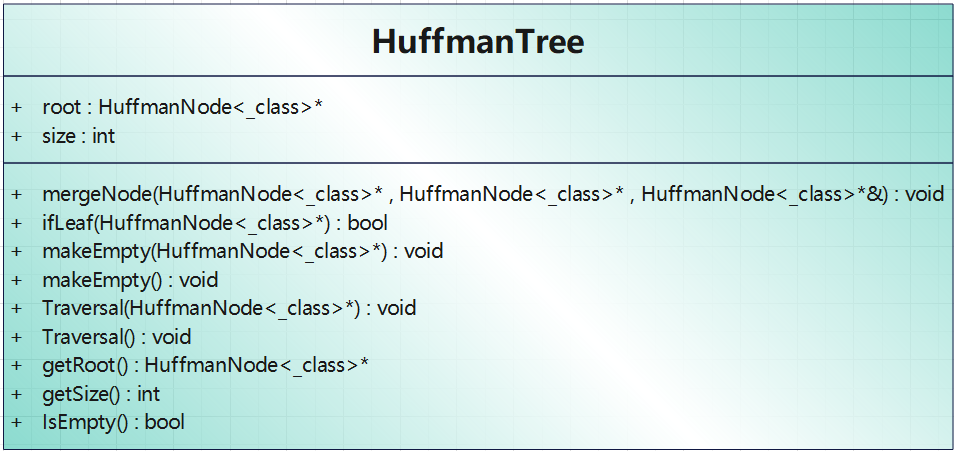
结点类（HuffmanNode）除了存储了一个结点的权值外，还存储了其左右子女和父结点的指针，其UML图如下：



**2.2.2 哈夫曼树类（HuffmanTree）**

哈夫曼树（HuffmanTree）是一种特殊的树，其每个非叶结点的权值是其左右子女权值之和，叶子结点的权值一般人为给出。同时，由于其构造时使用的特殊算法，其带权路径长度为最短。

其UML图如下：



其中，主要函数如下：

//将两个结点作为子女链接到同一个父节点上

void mergeNode(HuffmanNode<\_class>\* node1, HuffmanNode<\_class>\* node2, HuffmanNode<\_class>\*& parent);

//判断一个结点是否为叶结点

bool ifLeaf(HuffmanNode<\_class>\* node)const;

//清空整个哈夫曼树递归过程

void makeEmpty(HuffmanNode<\_class>\* ptr);

//遍历整个哈夫曼树递归过程

void Traversal(HuffmanNode<\_class>\* ptr)const;

//清空整个哈夫曼树

void makeEmpty();

//取哈夫曼树的根结点

HuffmanNode<\_class>\* getRoot()const;

//取哈夫曼树的结点数

int getSize()const;

//判断哈夫曼树是否为空

bool IsEmpty()const;

//遍历整个哈夫曼树

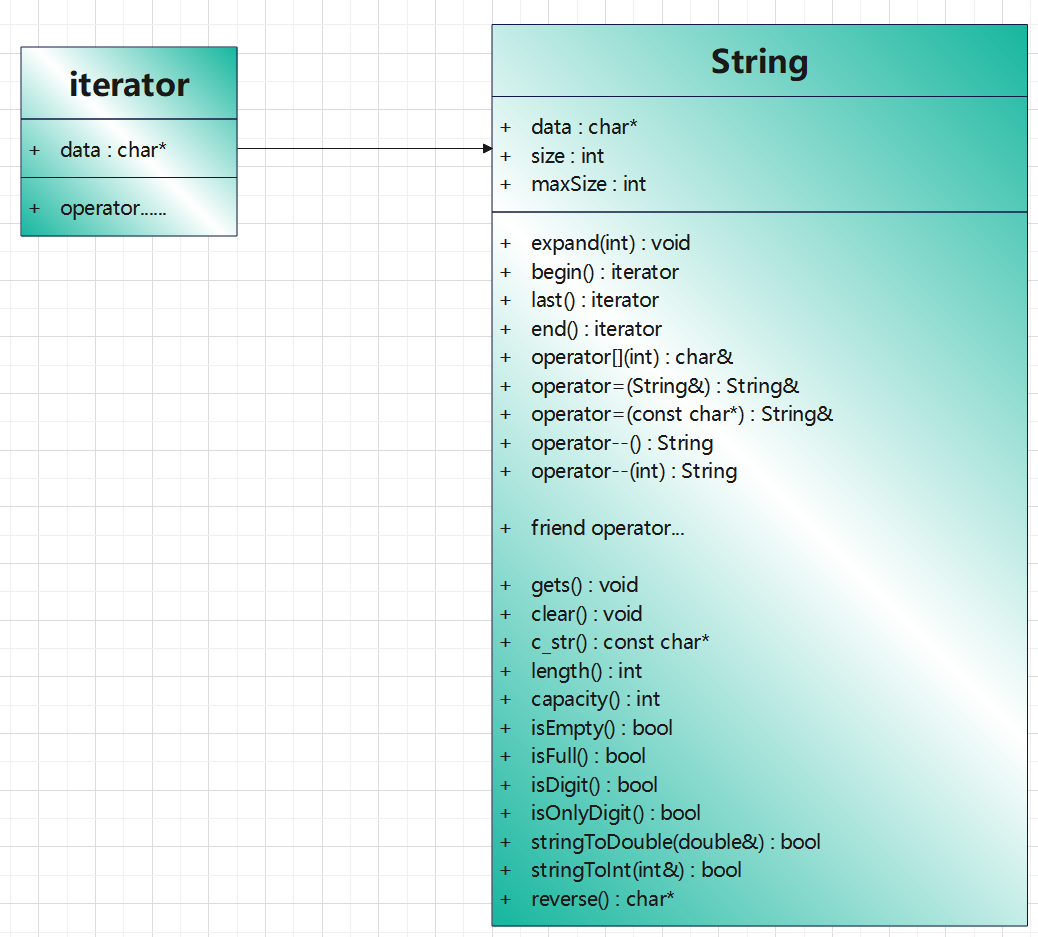
void Traversal()const;

**2.2.3 字符串类（String）**

字符串本质上是通过字符数组实现的，通过动态开辟一个数组存放各个字符。由于字符串的添加、比较、赋值等操作使用得十分频繁，故该类重载了许多运算符的操作，使得这些操作使用起来更易上手。

另一种常用的字符串操作是对其的遍历，在此添加了迭代器类（iterator）和一些运算符重载，以便对字符串的遍历等操作。由于这个类名在标准库中已有实现，且此处的迭代器类为String类特有，故使用了嵌套类的形式，使得iterator只能在String类内访问。

该类和其内部的iterator类的UML图如下所示：



其中，主要函数如下：

//增加数组空间大小

void expand(const int \_timesOfExpandingDefaultSize);

//返回字符串的起始位置

inline String::iterator begin();

inline const String::iterator begin()const;

//返回字符串末尾的后一个位置

inline String::iterator end();

inline const String::iterator end()const;

//返回字符串的末尾位置

inline String::iterator last();

inline const String::iterator last()const;

//重载函数：下标访问

char& operator[](const int pos)const;

//重载函数：复制（从String对象复制）

String& operator=(const String& str);

//重载函数：复制（从const char\*变量复制）

String& operator=(const char\* str);

//输入字符串（支持空格输入）

void gets();

//将字符串置为空

void clear();

//返回字符串首地址（以const char\*形式返回）

const char\* c\_str()const;

//返回字符串长度

int length()const;

//返回字符串最大容量

int capacity()const;

//判断字符串是否为空

bool isEmpty()const;

//判断字符串是否已满

bool isFull()const;

//判断字符串是否表示一个数值

bool isDigit()const;

//判断字符串是否仅有数字

bool isOnlyDigit()const;

//将字符串转成双精度数

bool stringToDouble(double& d)const;

//将字符串转成整型数

bool stringToInt(int& i)const;

//将字符串反转

char\* reverse();

2.3 算法设计

**2.2.1 算法思路**

构建Huffman树算法如下：

（1）根据给定的n个权值，构造具有n棵树的森林，每棵二叉树仅一个结点，左右子树均为空。

（2）在森林中选择两颗根结点权值最小的二叉树，作为左、右子树构造一棵新的二叉树。新二叉树根结点权值为左、右子树根结点的权值之和。

（3）在森林中删除这两棵二叉树

（4）把新的二叉树加入森林

（5）重复（2）-（4）直到森林中仅剩一棵树，则Huffman树构建完成。

寻找Huffman编码算法如下：

对Huffman树进行遍历，若左子树不为空，则向左子树递归遍历，字符串末尾添字符0，递归完成回到上一层时去掉这个0；若右子树不为空，则向右子树递归遍历，字符串末尾添字符1，递归完成回到上一层时去掉这个1；若遇到叶结点，则直接打印字符串内容，这就是该叶结点对应的Huffman编码。

特殊地，如果Huffman树只有一个结点，直接用0表示其Huffman编码。

**2.3.2 性能评估**

设构造前有n个权值。

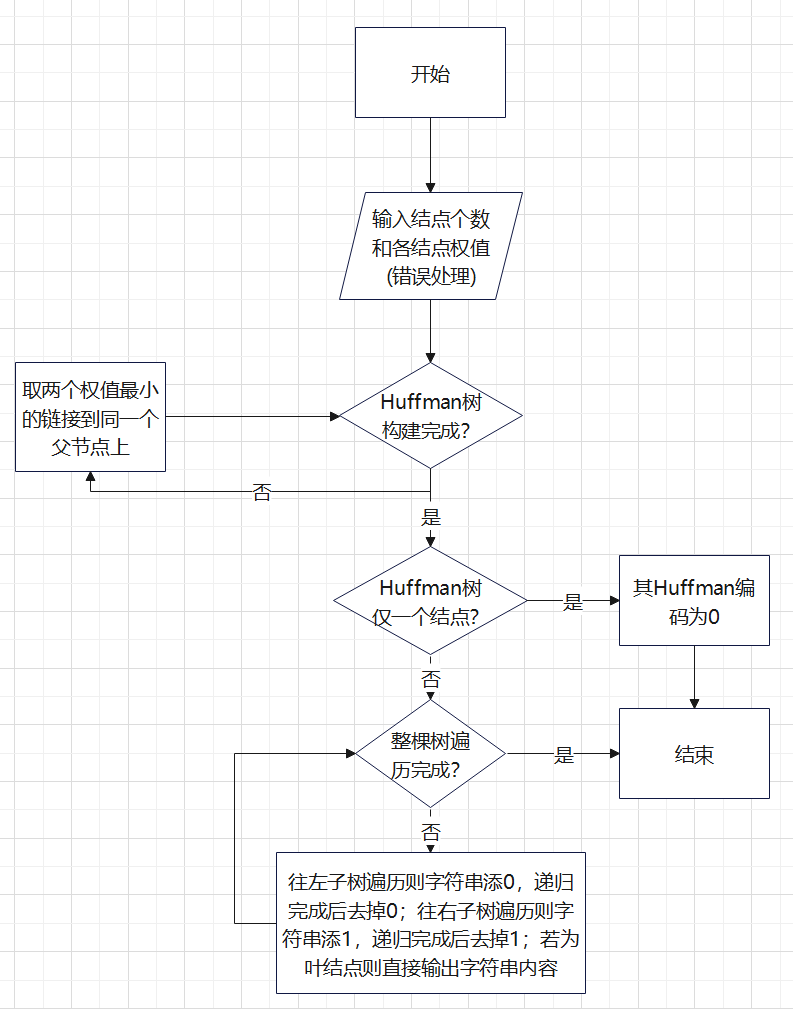
构造Huffman树时，构造最小堆需要O(nlog**2** n)，每趟取出2个权值最小元素需要O(log**2** n)，链接在一起需要O(1)，将新结点放回最小堆并调整需要O(log**2** n)，经过n趟后总共需要O(nlog**2** n)。故总时间复杂度仍为O(nlog**2** n)。

寻找Huffman编码时，需要对整棵树遍历，有n个叶结点和n-1个非叶结点，故需要O(n)。

综上，整个算法需要O(n+nlog**2** n)，总时间复杂度仍为O(nlog**2** n)。

**2.3.3 寻找Huffman编码算法**

·流程图表示



·构造Huffman树代码实现

template<class \_class>

HuffmanTree<\_class>::HuffmanTree(\_class weight[], const int num)

{

    size = num;

    if (num == 1) {

        root = new HuffmanNode<\_class>(weight[0]);

        if (root == NULL) {

            cerr << "存储分配错误！" << endl;

            exit(1);

        }

        return;

    }

    MinHeap<InfoNode<\_class> > heap;

    HuffmanNode<\_class>\* parent = NULL;

    for (int i = 0; i < num; i++) {

        HuffmanNode<\_class>\* ptr = new HuffmanNode<\_class>(weight[i]);

        if (ptr == NULL) {

            cerr << "存储分配失败！" << endl;

            exit(1);

        }

        heap.Insert(InfoNode<\_class>(ptr->data, ptr));

    }

    for (int i = 0; i < num - 1; i++) {

        InfoNode<\_class> info1, info2;

        heap.RemoveMin(info1);

        heap.RemoveMin(info2);

        mergeNode(info1.pos, info2.pos, parent);

        heap.Insert(InfoNode<\_class>(parent->data, parent));

    }

    root = parent;

}

·寻找Huffman编码代码实现

template<class \_class>

void findCode(HuffmanNode<\_class>\* ptr, String& code)

{

    if (ptr == NULL)

        return;

    if (ptr->leftChild == NULL && ptr->rightChild == NULL)  /\*找到叶结点，打印对应编码\*/

        cout << setiosflags(ios::left) << setw(4) << ptr->data << "——" << code << endl;

    if (ptr->leftChild) {

        code += '0';  /\*往左子树遍历，编码加'0'\*/

        findCode(ptr->leftChild, code);

        code--;  /\*上一层递归结束，编码去掉最后一位\*/

    }

    if (ptr->rightChild) {

        code += '1';  /\*往右子树遍历，编码加'1'\*/

        findCode(ptr->rightChild, code);

        code--;  /\*上一层递归结束，编码去掉最后一位\*/

    }

}

template<class \_class>

void findCode(const HuffmanTree<\_class>& tree)

{

    if (tree.getRoot()->leftChild == NULL && tree.getRoot()->rightChild == NULL)  /\*若只有一个节点，该结点编码用一个0表示\*/

        cout << tree.getRoot()->data << ":0" << endl;

    String code;

    findCode(tree.getRoot(), code);

}

·项目主体部分代码实现

int main()

{

    int num, weight;

    while (1) {

        cout << "输入节点个数：";

        cin >> num;

        if (cin.good() && num > 0 && num <= INT\_MAX)

            break;

        cin.clear();

        cin.ignore(INT\_MAX, '\n');

    }

    int\* data = new int[num];  /\*开辟存储节点的数组\*/

    if (data == NULL) {

        cerr << "存储分配错误！" << endl;

        exit(1);

    }

    cout << "输入节点：";

    for (int i = 0; i < num; i++) {

        while (1) {

            cin >> weight;

            if (cin.good() && weight > 0 && weight <= INT\_MAX)

                break;

            cin.clear();

            cin.ignore(INT\_MAX, '\n');

            cout << "第" << i + 1 << "个节点输入错误，请从该结点开始继续输入：" << endl;

        }

        data[i] = weight;

    }

    HuffmanTree<int> tree(data, num);  /\*根据所输入的节点内容创建哈夫曼树\*/

    findCode(tree);  /\*根据哈夫曼树生成哈夫曼编码\*/

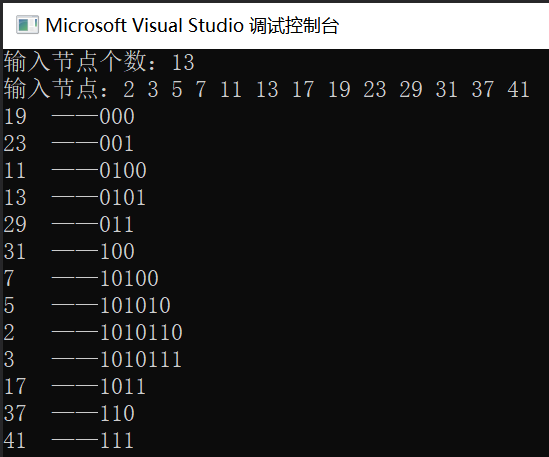
    return 0;

}

**3.项目测试**

3.1 示例情况

测试结果：



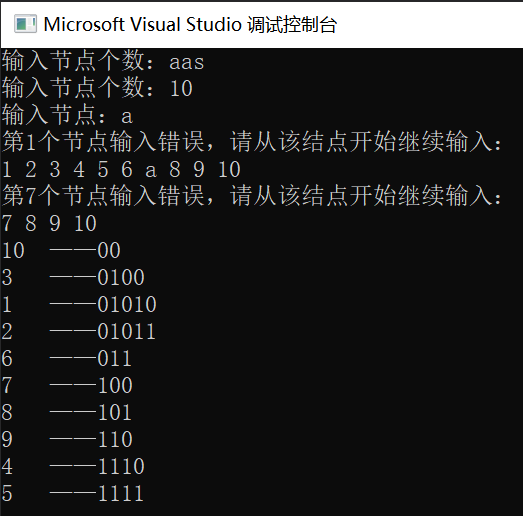
3.2 仅有一个结点的情况

测试结果：



3.3 输入错误处理

测试结果：



**4.心得体会**

在构造Huffman树时，由于涉及到指针，所以加入最小堆时需要将指向结点的指针加入。然而，由于无法重载指针的比较大小，需要再使用一种结构体InfoNode，成员有指向一个结点的指针和该结点的权值，重载InfoNode之间的比较函数即可。

在寻找Huffman编码时，记得在每一次递归完成后的地方都去除字符串最后一个字符的操作，否则会使得编码长度不对，造成平均长度变大或出现某个字符的编码是别的字符编码的前缀这样的错误。