离散数学

项目说明文档

最优二元树及前缀码

作者姓名: 高逸轩

学　　号: 2053385

指导教师: 唐剑锋

学院专业: 软件学院 软件工程

同济大学

Tongji University

本文称最优二元树为哈夫曼(Huffman)树，最优二元前缀码为哈夫曼编码：

1 功能简介

1.1 题目要求

根据用户给出的通信编码使用频率，建立哈夫曼树，并求每个节点的对应二元曼编码。

1.2 项目需求分析

本项目在实现的过程中，考虑并且满足了以下的需求：

- ✓ 健壮性
  当用户输入的数据不合理时，系统应当给予相应的提示而非直接报错。
- ✓ 代码可读性高
  本项目在实现过程中，将代码根据功能的不同划分为了不同的代码块，便于阅读。
- ✓ 安全性
  本项目在实现时，对内存管控严格，不造成浪费或泄露的情况，保证了程序的安全性。

1.3 项目要求

首先由用户输入给出每个节点的对应使用频率，利用指针指向的方式，建立哈夫曼树，然后借助生成的哈夫曼树生成对应的哈夫曼编码，最终将其和初始节点以一一对应的方式输出。

1.3.1 功能要求

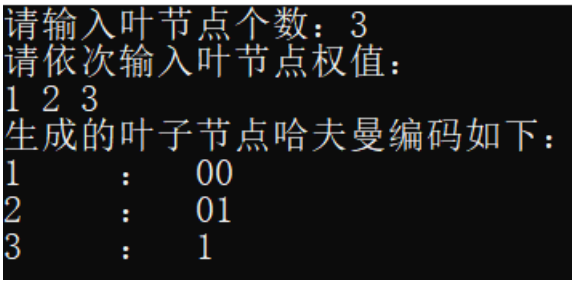
根据用户输入的编码使用频率及对应节点数目，开辟合适的空间大小，建立哈夫曼树，并遍历哈夫曼树至叶子节点生成哈夫曼编码，最终输出每个叶子节点的哈夫曼编码，并销毁已开辟的空间，避免内存泄露。

1.3.2 输入格式

以下内容利用 input_tools 函数集实现健壮性

```
叶子节点数目 nodeNumber

FOR i:=1 to nodeNumber
{
    第 i 个点的使用频率
}
```

1.3.3 项目简单示例



2 项目实现

本项目核心部分：

- ✓ 健壮性实现
- ✓ 快速排序实现
- ✓ 实现哈夫曼树的合并
- ✓ 实现哈夫曼树的建立
- ✓ 实现哈夫曼树的销毁
- ✓ 实现哈夫曼编码的生成

下面将对本项目核心的部分类进行介绍，部分代码如下：

```cpp
// 哈夫曼树节点
struct HuffmanTreeNode
{
    string coding = "";    // 哈夫曼编码
    int weight = 0;        // 点权值
    HuffmanTreeNode* leftChild = NULL, * rightChild = NULL, * parent = NULL;    // 左右子树、父节点初始设置为空
    bool operator<(const HuffmanTreeNode& x) { return weight < x.weight; }      // 重载小于号，用于排序
    bool operator>(const HuffmanTreeNode& x) { return weight > x.weight; }      // 重载大于号，用于排序

    // 重载加号
    HuffmanTreeNode operator+(const HuffmanTreeNode& x)
    {
        HuffmanTreeNode ret;
        ret.weight = weight + x.weight;
        return ret;
    }
};
```

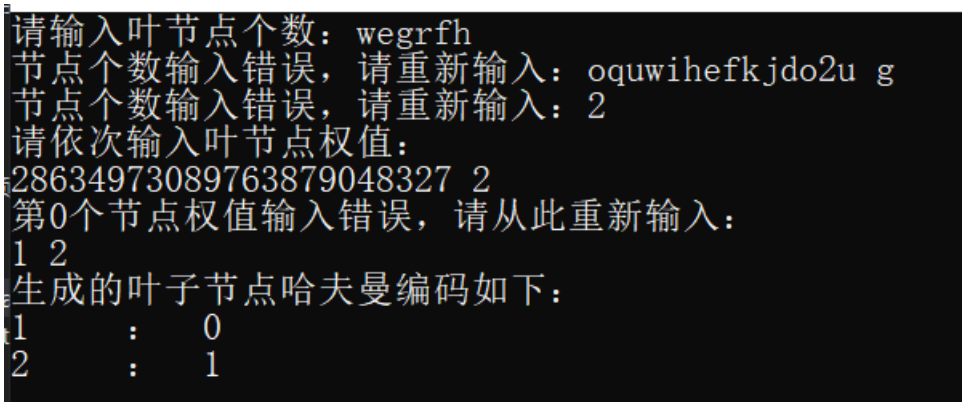
```cpp
class HuffmanTree
{
public:
    // 构造函数
    HuffmanTree(const int n = 0) :nodeNumber(n) { root = new HuffmanTreeNode; };

    // 析构函数，调用销毁函数，释放空间
    ~HuffmanTree() { DeleteTree(root); }

    // 建立哈夫曼树
    void BuildHuffmanTree(const int w[]);

    // 合并hft1 hft2两树，存储于新节点parent并以地址形式返回
    void MergeTree(HuffmanTreeNode& hft1, HuffmanTreeNode& hft2, HuffmanTreeNode*& parent);

    // 销毁以current为地址节点根的树
    void DeleteTree(HuffmanTreeNode* current);

    // 返回根节点地址
    HuffmanTreeNode* Root() { return root; }

    // 生成哈夫曼编码表
    void GenerateHuffmanTable(HuffmanTreeNode* current);

    // 输出哈夫曼编码表
    void PrintHuffmanTable(HuffmanTreeNode* current);

    // 快速排序函数
    friend void QuickSort(HuffmanTreeNode a[], int l, int r);

private:
    HuffmanTreeNode* root = NULL; // 根节点
    int nodeNumber = 0;           // 叶节点数目
};
```

哈夫曼树相关概念如下：

在一棵树中，从一个结点到另一个结点所经过的所有结点，被我们称为两个结点之间的路径。在一棵树中，从一个结点到另一个结点所经过的“边”的数量，被我们称为两个结点之间的路径长度。树的每一个结点，都可以拥有自己的“权重”（Weight），权重在不同的算法当中可以起到不同的作用。结点的带权路径长度，是指树的根结点到该结点的路径长度，和该结点权重的乘积。在一棵树中，所有叶子结点的带权路径长度之和，被称为树的带权路径长度，也被简称为 WPL。而哈夫曼树（Huffman Tree）是在叶子结点和权重确定的情况下，带权路径长度最小的二叉树，也被称为最优二叉树。

2.1 健壮性

通过 input_tools 函数集实现每条数据输入时的错误输入处理。



2.2 快速排序

2.2.1 运算原理

根据网络资料，学习快速排序概念与优秀原因如下：

1、快速排序的基本思想：

快速排序使用**分治**的思想，通过一趟排序将待排序列分割成两部分，其中一部分记录的关键字均比另一部分记录的关键字小。之后分别对这两部分记录继续进行排序，递归地以达到整个序列有序的目的。

2、快速排序的三个步骤：

(1)选择基准：在待排序列中，按照某种方式挑出一个元素，作为 "基准"（pivot）

(2)分割操作：以该基准在序列中的实际位置，把序列分成两个子序列。此时，在基准左边的元素都比该基准小，在基准右边的元素都比基准大

(3)递归地对两个序列进行快速排序，直到序列为空或者只有一个元素。

3、选择基准的方式

对于分治算法，当每次划分时，算法若都能分成两个等长的子序列时，那么分治算法效率会达到最大。也就是说，基准的选择是很重要的。选择基准的方式决定了两个分割后两个子序列的长度，进而对整个算法的效率产生决定性影响。

2.2.2 时间复杂度分析与快速排序的优化

在最坏的情况下，仍可能是相邻的两个数进行了交换。因此快速排序的最差时间复杂度和冒泡排序是一样的都是 $O(n^2)$，平均的时间复杂度为 $O(nlogn)$. 根据课本讲解，选择序列第一个元素为比较基准，代码实现如下：

```
void QuickSort(int a[], int begin, int end)
{
    if (begin > end)
        return;
    int i = begin, j = end;
    while (i != j)
    {
        while (a[j] >= a[begin] && j > i)
            j--;
        while (a[i] <= a[begin] && j > i)
            i++;
        if (j > i)
            swap(a[i], a[j]);
    }
    swap(a[i], a[begin]);
    QuickSort(a, begin, i - 1);
    QuickSort(a, i + 1, end);
}
```

测试数据分析：如果输入序列是随机的，处理时间可以接受的。如果数组已经有序时，此时的分割就是一个非常不好的分割。因为每次划分只能使待排序序列减一，此时为最坏情况，快速排序沦为冒泡排序，时间复杂度为 $O(n^2)$。而且，考虑到输入的数据是有序或部分有序的情况是相当常见的。因此，使用第一个元素作为枢纽元是非常糟糕的，为了避免这个情况，我们更换了选择比较基准的元素，取中间元素为比较基准，实现优化：

```
// 优化版快速排序
void QuickSort(HuffmanTreeNode a[], int l, int r)
{
    if (l > r)
        return;
    HuffmanTreeNode mid = a[(l + r) / 2];    // 取中间元素
    int i = l, j = r;
    while (i < j)
    {
        while (a[i] < mid) i++;              // 查找左半部分比中间数大的数
        while (a[j] > mid) j--;              // 查找右半部分比中间数小的数
        if (i <= j)                          // 如果有一组不满足排序条件（左小右大）的数
        {
            swap(a[i], a[j]);                // 交换
            i++;
            j--;
        }
    }
    if (l < j) QuickSort(a, l, j);           // 递归排序左半部分
    if (i < r) QuickSort(a, i, r);           // 递归排序右半部分
}
```

2.3 哈夫曼树的合并

当进行哈夫曼树的构建时，我们常常需要将两个子节点合并，构成新的父节点并加入到树中，以下为哈夫曼树的合并操作，其中函数的参数以引用形式调用，减少了重新创建临时变量所花费的时间：

```
// 合并hf1 hft2两节点至parent节点，并以地址形式返回
void HuffmanTree::MergeTree(HuffmanTreeNode& hft1, HuffmanTreeNode& hft2, HuffmanTreeNode*& parent)
{
    // 申请空间
    parent = new HuffmanTreeNode;

    // 左右子树位置设置
    parent->leftChild = &hft1, parent->rightChild = &hft2;

    // 权重设置
    parent->weight = hft1.weight + hft2.weight;

    // 左右子树父节点位置设置
    hft1.parent = parent, hft2.parent = parent;

}
```

2.4 哈夫曼树的建立

2.4.1 运算原理

哈夫曼树的构建步骤如下：

1、将给定的 n 个权值看做 n 棵只有根节点（无左右孩子）的二叉树，组成一个集合，每棵树的权值为该节点的权值。

2、从集合中选出 2 棵权值最小的二叉树，组成一棵新的二叉树，其权值为这 2 棵二叉树的权值之和。

3、将步骤 2 中选出的 2 棵二叉树从集合中删去，同时将步骤 2 中新得到的二叉树加入到集合中。

4、重复步骤 2 和步骤 3，直到集合中只含一棵树，这棵树便是哈夫曼树。

2.4.2 维护集合有序性

其中，在每次从集合中取出两棵树的时候，要求这两棵树的权值最小，这要求我们维护这个集合的有序性。我开始的思路是运用 STL 的优先队列（小根堆）的数据结构来非常方便的存储集合中的数据，但由于本次作业中不允许使用 STL 工具，我便用数组来模拟队列（对于拥有 n 个点的哈夫曼树，共有 2*n-1 个节点，故数组空间开辟时不要造成浪费），以快速排序的方式来维护队列有序，从而达到优先队列的效果。具体

2.4.3 动态空间管控

在哈夫曼树的建立过程中，每一个新的节点产生时，都需要申请新的空间，并不断更改哈夫曼树的根节点地址。同时，在模拟优先队列的数组中，也需要开辟合适的空间大小。另外，临时变量的创建、调用函数时参数传值还是传址的选择、变量的生存期及销毁等也需要层层考虑。在最终，还需要将无用的空间进行销毁，而保证哈夫曼树的空间不被破坏，对于空间的控制要求较高。

在这个过程中，每次取出两个队头元素时，我会开辟新的空间来记录这两个元素，而不是直接对队列中的元素进行操作，以避免更改队列中元素的数值。同时，在通过调用合并函数每次得到新的节点时，我采取函数参数传父节点地址的方式，避免函数中的变量生存期结束导致数据丢失。另外，每次加入新的节点后，

我会更新根节点的值为加入元素的地址，这样可以保证最后合并成为一个点的时候，哈夫曼树根节点即为最后一个新创建的节点。最后，在创建结束后，要释放掉用以模拟队列的数组空间，避免内存泄露。

2.4.4 代码实现

根据以上描述和快速排序函数，实现哈夫曼树的构建代码如下：

```
// 构造哈夫曼树
void HuffmanTree::BuildHuffmanTree(const int w[])
{
    // 哈夫曼树共 2 * n - 1 个节点，用数组模拟队列
    HuffmanTreeNode* array = new HuffmanTreeNode[2 * nodeNumber];

    // 将初始的叶子节点放入数组
    FOR(i, 0, nodeNumber)
        array[i].weight = w[i];

    HuffmanTreeNode* ancestor = new HuffmanTreeNode;
    // 将n个节点最终合并为1个根节点，需要n-1次合并
    FOR(i, 0, nodeNumber - 1)
    {
        HuffmanTreeNode* current1 = new HuffmanTreeNode, * current2 = new HuffmanTreeNode, * parent;

        // 对队列指定区间进行排序，保证每次取出的点是权值较小的点
        QuickSort(array, 2 * i, nodeNumber + i - 1);

        // 取出队列中最小元素和次小元素
        *current1 = array[2 * i], * current2 = array[2 * i + 1];

        // 将最小元素和次小元素合并至parent
        MergeTree(*current1, *current2, parent);

        // 父节点入队列
        array[nodeNumber + i] = *parent;

        // 将根节点位置更新
        root = parent;
    }

    // 释放用来模拟队列临时数组的空间
    delete[]array;
}
```

2.5 哈夫曼树的销毁

在销毁哈夫曼树时，主要需要将其占用的空间释放，这需要对哈夫曼树进行遍历。若采取递归遍历方式，则需在销毁函数的参数中设置相应变量来实现传参来实现遍历。但析构函数中不能存在参数，故重新编写了带参的释放空间的函数。在递归遍历哈夫曼树并实现空间释放的过程中，采取**后序遍历**的方式（否则提前释放掉父节点的信息，则不能访问到子节点，无法实现子节点的释放），具体代码实现如下：

```
// 析构函数，调用销毁函数，释放空间
~HuffmanTree() { DeleteTree(root); }

// 销毁以current地址为根的树的空间
void HuffmanTree::DeleteTree(HuffmanTreeNode* current)
{
    // 遇空位置返回
    if (current == NULL) return;

    // 递归向下释放空间
    DeleteTree(current->leftChild), DeleteTree(current->rightChild);

    // 释放当前节点的空间
    delete current;
}
```

2.6 哈夫曼编码的生成

在本题中，我采取将根节点的哈夫曼编码置为空，采用先序遍历的方式遍历哈夫曼树，左儿子在父节点的基础上哈夫曼编码后缀+0，右儿子在父节点的基础上哈夫曼编码后缀+1 的方式来进行编码。在遍历结束后，各个叶子节点的编码即对应的哈夫曼编码。

```
// 生成哈夫曼编码表
void HuffmanTree::GenerateHuffmanTable(HuffmanTreeNode* current)
{
    // 已经遍历至叶节点之下，则返回
    if (current == NULL) return;

    // 若当前节点不是叶节点，更新其左右子树的哈夫曼编码
    if (current->leftChild != NULL)
        (current->leftChild)->coding = current->coding + '0';  // 左子树编码在父节点基础上后缀+0
    if (current->rightChild != NULL)
        (current->rightChild)->coding = current->coding + '1'; // 右子树编码在父节点基础上后缀+1

    // 向下遍历
    GenerateHuffmanTable(current->leftChild);
    GenerateHuffmanTable(current->rightChild);
}
```

3 项目测试

本题健壮性实现优秀，帮助用户自动补全已经输入的正确部分，仅需输入错误部分即可。示例如下：

```
请输入叶节点个数：3
请依次输入叶节点权值：
123 ewfge
第1个节点权值输入错误，请从此重新输入：
123
```

```
请输入叶节点个数：7
请依次输入叶节点权值：
1 qopejrwiohf guei
第1个节点权值输入错误，请从此重新输入：
1 2 3 4 poq EIHOGT3942N0
第4个节点权值输入错误，请从此重新输入：
1 2 3 4 8 10
24 1
生成的叶子节点哈夫曼编码如下：
24      :    0
10      :    10
8       :    110
4       :    1110
1       :    111100
2       :    111101
3       :    11111
```

4 心得与总结

在本次求最优二元树及编码的作业中，我学习了哈夫曼树的建立、合并、销毁等操作，也了解了哈夫曼树的一些特点，如 n 个叶子节点的哈夫曼树共有 2*n-1 个节点；在编写程序的过程中，我也对内存、地址的分配与管理、变量的地址和引用、变量的生存期等 c++语言特色有了更好的认识，提高了 coding 能力。在最优二元编码方面，我也了解到了多种编码方式，并选择了自己最习惯的方式进行了实现，同时了解了一些拓展知识。