项目说明文档

《离散数学》课程实验报告

——最优2元树的应用

作 者 姓 名： 杨滕超

学 号： 2151298

指 导 教 师： 唐剑锋

学院、 专业： 软件学院 软件工程

同济大学

Tongji Universit

目 录

[1 题目简介](#_Toc495668153) 4

[1.1 题目原理背景](#_Toc495668154) 4

[1.1.1 最优2元树](#_Toc495668163) 4

[1.2 题目概述](#_Toc495668155) 4

[1.3 程序功能](#_Toc495668154) 5

[2 解题思路](#_Toc495668156) 5

[3 数据结构设计](#_Toc495668161) 5

[3.1 堆](#_Toc495668162) 5

[3.1.1 说明](#_Toc495668167) 5

[3.2 优先队列 6](#_Toc495668166)

[3.2.1 说明 6](#_Toc495668167)

[3.2.2 UML图](#_Toc495668167) 6

[3.3 向量](#_Toc495668170) 6

[3.3.1 说明](#_Toc495668167) 6

[3.3.2 UML图](#_Toc495668167) 7

[3.4 Vector游标类](#_Toc495668170) 7

[3.4.1 说明](#_Toc495668167) 7

[3.4.2 UML图](#_Toc495668167) 7

[3.5 链表结点类](#_Toc495668170) 8

[3.5.1 说明](#_Toc495668167) 8

[3.5.2 UML图](#_Toc495668167) 8

[3.6 链表类](#_Toc495668170) 8

[3.6.1 说明](#_Toc495668167) 8

[3.6.2 UML图](#_Toc495668167) 9

[3.7 链表结点类](#_Toc495668170) 9

[3.7.1 说明](#_Toc495668167) 9

[3.7.2 UML图 1](#_Toc495668167)0

[3.8 哈夫曼树结点 1](#_Toc495668170)0

[3.8.1 说明 1](#_Toc495668167)0

[3.8.2 UML图 1](#_Toc495668167)1

[3.9 优先队列结点 1](#_Toc495668170)1

[3.9.1 说明 1](#_Toc495668167)1

[3.9.2 UML图 1](#_Toc495668167)1

[3.10 求哈夫曼编码类 1](#_Toc495668170)1

[3.10.1 说明 1](#_Toc495668167)1

[3.10.2 UML图 1](#_Toc495668167)2

[4 核心算法](#_Toc495668161) 12

[4.1 堆向上调整](#_Toc495668174) 12

[4.1.1 流程图](#_Toc495668176) 12

[4.1.2 说明](#_Toc495668176) 13

[4.1.3 主要代码](#_Toc495668176) 13

[4.2 堆向下调整](#_Toc495668174) 14

[4.2.1 流程图](#_Toc495668176) 14

[4.2.2 说明](#_Toc495668176) 14

[4.2.3 主要代码](#_Toc495668176) 14

[4.3 结点个数输入](#_Toc495668174) 16

[4.3.1 流程图](#_Toc495668176) 16

[4.3.2 说明](#_Toc495668176) 17

[4.3.3 主要代码](#_Toc495668176) 17

[4.4 叶子节点值输入](#_Toc495668174) 18

[4.4.1 流程图](#_Toc495668176) 18

[4.4.2 说明](#_Toc495668176) 18

[4.4.3 主要代码](#_Toc495668176) 19

[4.5 构建哈夫曼树](#_Toc495668174) 20

[4.5.1 流程图](#_Toc495668176) 20

[4.5.2 说明](#_Toc495668176) 20

[4.5.3 主要代码](#_Toc495668176) 21

[4.6 得到哈夫曼编码](#_Toc495668174) 22

[4.6.1 流程图](#_Toc495668176) 22

[4.6.2 说明](#_Toc495668176) 23

[4.6.3 主要代码](#_Toc495668176) 23

[4.7 叶子节点值输入](#_Toc495668174) 24

[4.7.1 说明](#_Toc495668176) 24

[4.7.2 主要代码](#_Toc495668176) 24

[5 实验结果](#_Toc495668161) 24

[5.1 样例测试](#_Toc495668174) 24

[5.2 叶子节点值为小数](#_Toc495668174) 25

[5.3 叶子节点值相同](#_Toc495668174) 25

[5.4 输入结点错误](#_Toc495668174) 26

[5.5 输入节点值错误](#_Toc495668174) 26

[6 心得体会](#_Toc495668161) 26

1 题目简介

* 1. 题目原理背景
     1. 最优2元树

首先介绍最优二元树的相关名词：

1. 路径：在一棵树中，一个结点到另一个结点之间的通路，称为路径。
2. 路径长度：在一条路径中，每经过一个结点，路径长度都要加 1 。例如在一棵树中，规定根结点所在层数为1层，那么从根结点到第 i 层结点的路径长度为 i - 1 。
3. 结点的权：给每一个结点赋予一个新的数值，被称为这个结点的权。

所谓最优二元树，又称哈夫曼树，是一种带权路径长度最短的[二叉树](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%A0%91?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E5%93%88%E5%A4%AB%E6%9B%BC%E6%A0%91/_blank)。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）。简单来说，构造哈夫曼树的过程就是希望权值大的结点与根的距离近，相反，希望权值小的结点与根的距离远，由此构成最小的带权路径长度最短。

利用哈夫曼树的构造，可以得到一些问题的最优解，在相关问题中具有重要作用。例如哈夫曼编码，作为哈夫曼树的典型应用，它实在数字通信中，将传送的文字转换0和1组成的二进制编码，人们希望总体编码尽可能地简短，哈夫曼树的构造正好满足这一要求。

* 1. 题目概述

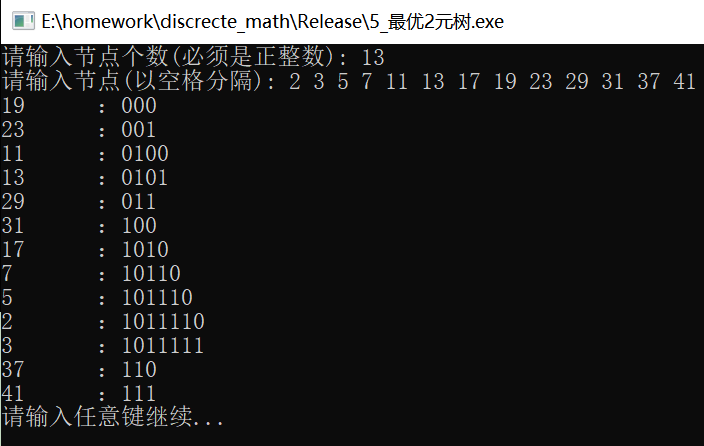
在计算机数据处理中，哈夫曼编码使用变长编码表对源符号进行编码，其中变长编码表是通过一种评估来源符号出现几率的方法得到的，出现几率高的字母使用较短的编码，反之出现几率低的则使用较长的编码，这便使得编码之后的字符串的平均长度、期望值降低。从而达到五寸压缩数据的目的。

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。

* 1. 程序功能

首先请用户输入结点的个数，然后根据节点个数请用户输入对应个数的通信符号的使用频率。然后输出每个通信符号的频率对应的前缀码。

示例如下：



2 解题思路

通过构建一棵哈夫曼树以求得哈夫曼编码，其中的哈夫曼编码就是对应频率的前缀码。

3 数据结构设计

3.1 堆

3.1.1 说明

堆是一种树形结构，是完全树，因此常常利用数组储存。分为大根堆和小根堆，以小根堆为例，任意一个根的孩子结点的值总是不小于其父节点，同时两个孩子也是一个小根堆。

我们利用向量作为堆的基础，通过迭代方式，实现堆的操作，主要成员函数如下：

·上浮调整void siftFix(Iter beg, Iter end, Compare cmp);

·下沉调整void sinkFix(Iter beg, Iter end, Compare cmp);

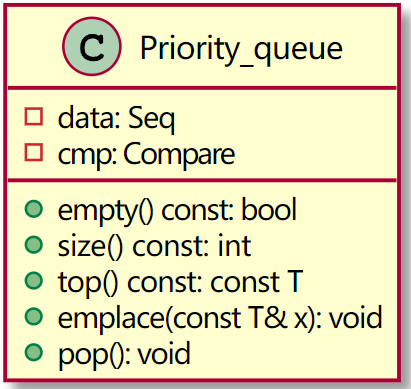
3.2 优先队列

3.2.1 说明

基于Vector类作为存储数据的数据结构，Heap类中的方法，形成优先队列，其中参数模板包括数据类型、数据底层结构（默认为Vector）以及元素之间的比较方法。比较方法作为仿函数传入类中作为类的数据成员。其中入队出队的时间复杂度均为O(logn)。

其中定义模板类型Seq与仿函数Compare，以适应不同的数据类型，进一步增强代码的可复用性。Seq类型代表优先队列底层的存储结构，Compare代表数据之间比较的仿函数。

3.2.2 UML图

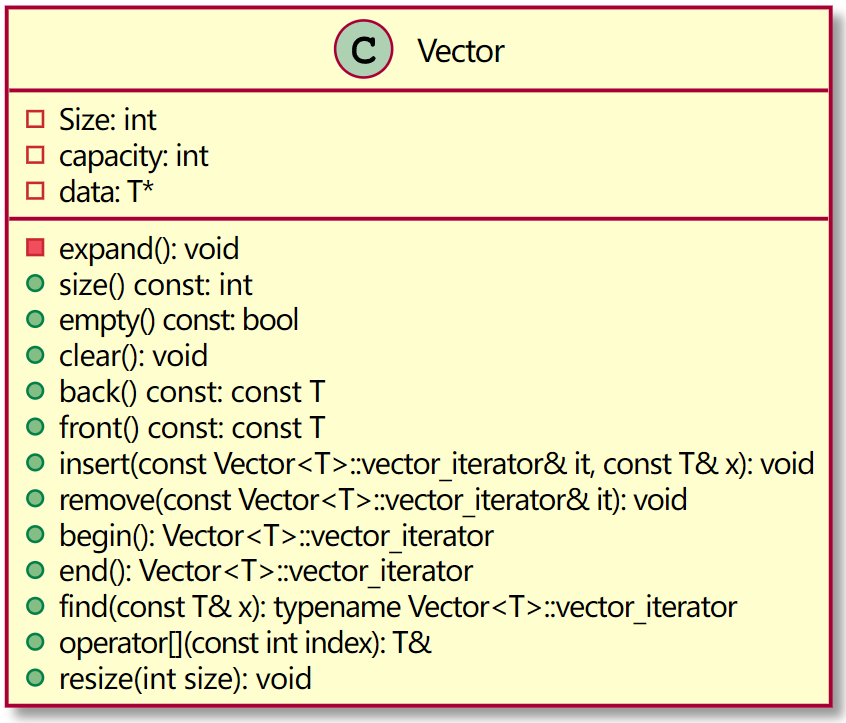


3.3 向量

3.3.1 说明

向量作为存储同一种类型数据的一维数组，相当于是数组的扩展，在其基础上增加方便程序员工作的操作。其中的优点在于：对于查找某个位置的元素的时间复杂度为O(1)，但是它的缺点也十分明显：向量的删除与插入的时间代价是巨大的，时间复杂度基本是O(n)级别；但特殊的是，在最后插入元素时，时间复杂度为O(1)

3.3.2 UML图

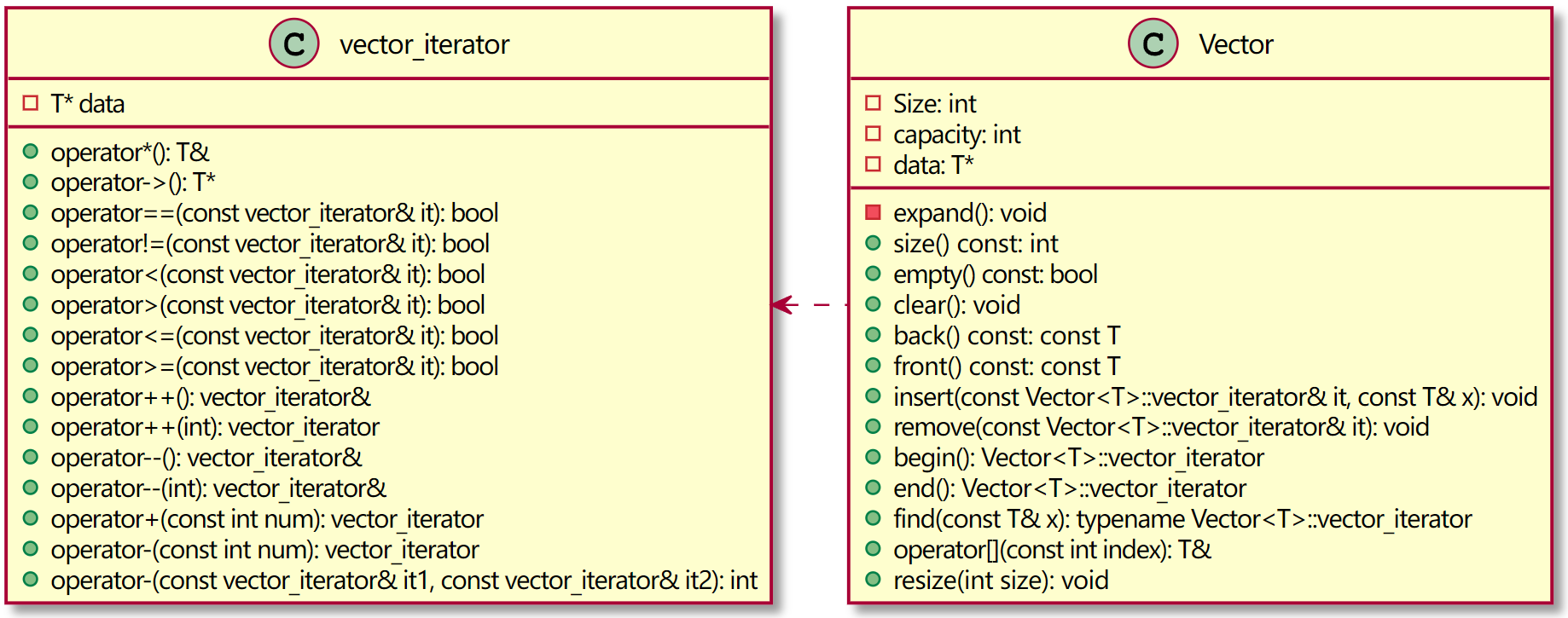


3.4 Vector游标类

3.4.1 说明

其中数据成员包含了指向Vector中数组元素的指针，并实现的重载\*、->、++、--、<、>等操作符。

3.4.2 UML图

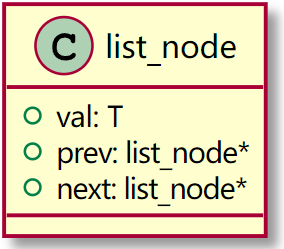


3.5 链表结点类

3.5.1 说明

链表结点作为链表的基本单元，是链表的重要组成部分。这里将链表设计为双链表，其中包括需要储存的数据，指向前驱节点和后继节点的指针。

3.5.2 UML图

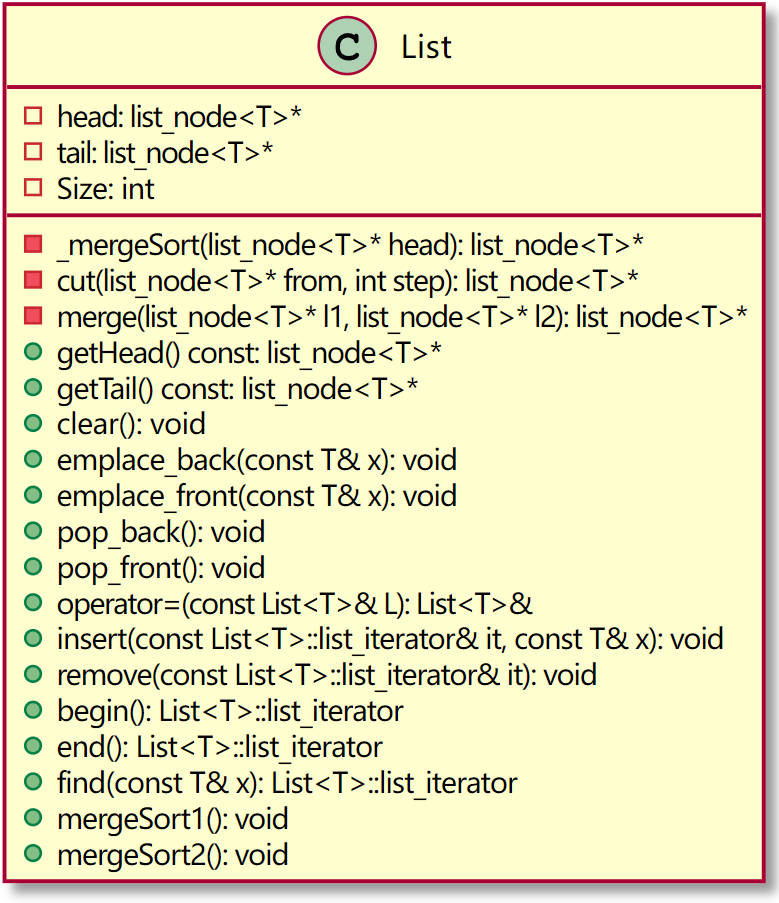


3.6 链表类

3.6.1 说明

链表是一种物理[存储单元](https://baike.baidu.com/item/%E5%AD%98%E5%82%A8%E5%8D%95%E5%85%83/8727749?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E9%93%BE%E8%A1%A8/_blank)上非连续、非顺序的[存储结构](https://baike.baidu.com/item/%E5%AD%98%E5%82%A8%E7%BB%93%E6%9E%84/350782?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E9%93%BE%E8%A1%A8/_blank)，[数据元素](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%85%83%E7%B4%A0/715313?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E9%93%BE%E8%A1%A8/_blank)的逻辑顺序是通过链表中的[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88/2878304?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E9%93%BE%E8%A1%A8/_blank)链接次序实现的。链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储[数据元素](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%85%83%E7%B4%A0?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E9%93%BE%E8%A1%A8/_blank)的数据域，另一个是存储下一个结点地址的[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88/2878304?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E9%93%BE%E8%A1%A8/_blank)域。 相比于[线性表](https://baike.baidu.com/item/%E7%BA%BF%E6%80%A7%E8%A1%A8/3228081?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E9%93%BE%E8%A1%A8/_blank)[顺序结构](https://baike.baidu.com/item/%E9%A1%BA%E5%BA%8F%E7%BB%93%E6%9E%84/9845234?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E9%93%BE%E8%A1%A8/_blank)，操作复杂。由于不必须按顺序存储，链表在插入的时候可以达到O(1)的复杂度，比另一种线性表顺序表快得多，但是查找一个节点或者访问特定编号的节点则需要O(n)的时间，而线性表和顺序表相应的时间复杂度分别是O(log2n)和O(1)。

3.6.2 UML图



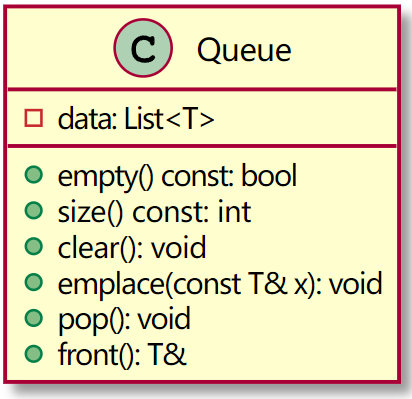
3.7 队列类

3.7.1 说明

队列和栈一样，同样是一种运算受限的线性表，它允许的操作是在表的前端(front)进行删除，表的后端(rear)进行插入，遵循先进先出的原则。其中的前端称为队头，后端称为队尾。队列的实现同样分为两种方式，顺序队列和链式队列，对于顺序队列，为了实现空间的循环利用，采用循环队列的方式，其中的要点在于队头指针和队尾指针往后移动增加1后对最大空间大小取模。而对于链式队列，其优点在于没有空间的限制，可以灵活地申请空间。

基于上述考虑，我们采用以链表为基础的队列。考虑到链表无论是在头或尾的插入删除结点的时间复杂度均为O(1)，因此我们将链表头作为队列头，链表为作为队列尾。

3.7.2 UML图



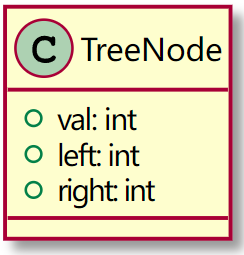
3.8 哈夫曼树结点

3.8.1 说明

队这里为了使用优先队列进行比较，经过实践发现，当优先队列中的元素是动态申请的空间的地址的时候，不能较方便地对地址进行比较。经过思考，我想到的方法是：构建一个二元结构体，其中包括的信息是地址和地址所指向的空间存储的信息。对于本题目，还有另一种方法，我们使用静态二叉链表的方式作为哈夫曼树的结点，由此可以通过重载大于小于操作符号进行结构体的比较，从而使得优先队列发挥作用。当然仅仅有这个结构体仍然不能很好地使用优先队列，原因在于我们不知道进入优先队列的哈夫曼树的结点所在静态链表中的下标，因此需要再构建以下结构体。

结点中的数据成员包括：值和左孩子和右孩子在数组中的下标。

3.8.2 UML图

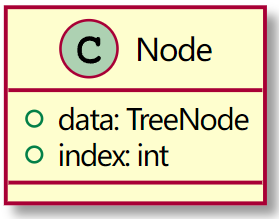


3.9 优先队列结点

3.9.1 说明

为了结点进入优先队列的哈夫曼树的结点不具有在静态链表中的下标的信息，因此构建一个二元结构体，数据成员包括：哈夫曼树的结点以及在静态链表中的下标。

3.9.2 UML图



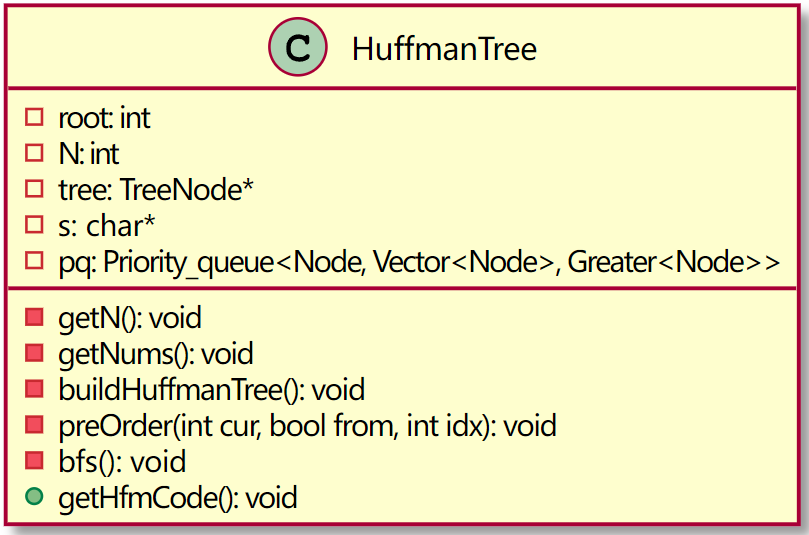
3.10 求哈夫曼编码类

3.10.1 说明

为了求解哈夫曼编码，我们将该方法封装成一个类，包括由用户输入数据，然后求解哈夫曼编码的一系列函数。其中数据成员包括：int整型类型的root表示树根的结点信息；int整型类型的N表示树结点多个数；TreeNode\*哈夫曼树结点指针类型tree作为申请静态链表空间的首地址；优先队列pq以及存放哈夫曼编码的字符类型数组首地址s。

成员函数包括：得到结点个数的函数、得到每个结点的值的函数、构建哈夫曼树的函数以及求解哈夫曼编码的函数等等。

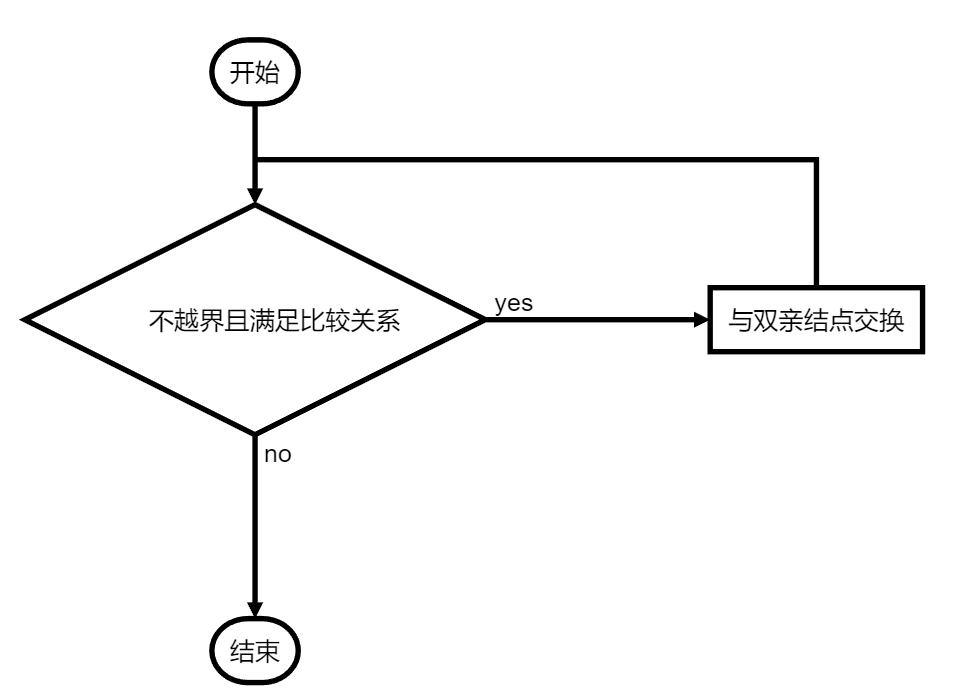
3.10.2 UML图



4 核心算法

4.1 堆向上调整

4.1.1 流程图



4.1.2 说明

对于向上调整具体过程，这里将其存入临时变量val中，设置变量hole表示val最终目标位置。在循环过程中，val与其双亲结点的值比较，满足比较条件则将双亲结点的值下移，hole位置上移，将双亲结点的位置赋给hole，计算hole新一轮的双亲，直到不满足比较条件，退出循环，这时候将最终目标位置hole处的值用val填充。

在上述过程中，需要注意的是要时刻保证下标不会越界。

4.1.3 主要代码

template<class Iter, class Compare>

void siftFix(Iter beg, Iter end, Compare cmp){

typedef typename Iter::value\_type value\_type;

//新加入的元素的值

value\_type val = \*(end - 1);

//要处理的hole的位置

int hole = (end - 1) - beg;

//hole的父节点

int parent = (hole - 1) / 2;

//上溯 结束 条件 hole越界 or 不满足cmp关系

while (hole >= 0 && hole != parent && cmp( \*(beg + parent), val)){

//父节点的值下来

\*(beg + hole) = \*(beg + parent);

//hole上去

hole = parent;

parent = (hole - 1) / 2;

}

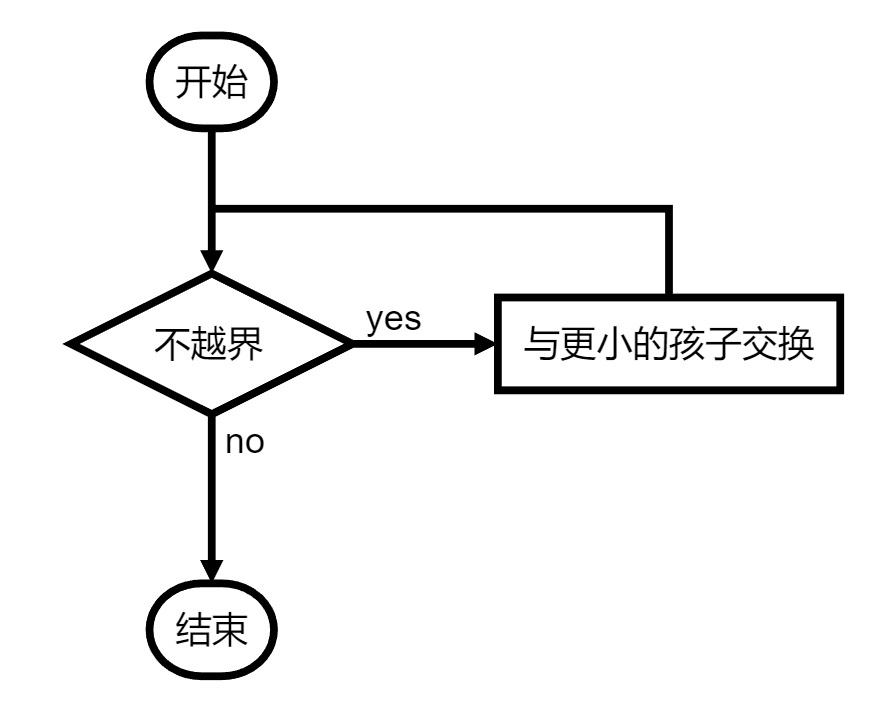
//hole上去到能上去的最高位置

\*(beg + hole) = val;

}

4.2 堆向下调整

4.2.1 流程图



4.2.2 说明

对于向下调整，首先需要选择两个孩子中更满足比较条件的孩子，沿着该孩子的路径下沉到最后。这里同样利用hole标记最终的目标位置，以减少值交换的次数。最后同样要注意的的是，对于堆是一棵完全二叉树，因此可能会出现只有一个左孩子的情况，这时候需要单独判断。

4.2.3 主要代码

template<class Iter, class Compare>

void sinkFix(Iter beg, Iter end, Compare cmp){

typedef typename Iter::value\_type value\_type;

//要处理的元素的值

value\_type val = \*beg;

int hole = 0;

int leftNode = hole \* 2 + 1;

int rightNode = leftNode + 1;

int len = end - beg;

//寻找最后一个元素要去的hole，因为第一个元素要占用

while (rightNode < len){

//选中符合条件的那个孩子交换，hole下沉

if (cmp(\*(beg + rightNode), \*(beg + leftNode))){

\*(beg + hole) = \*(beg + leftNode);

hole = leftNode;

}

else{

\*(beg + hole) = \*(beg + rightNode);

hole = rightNode;

}

leftNode = hole \* 2 + 1;

rightNode = leftNode + 1;

}

//还有一个单独的左孩子，继续hole下沉

if (leftNode < len){

\*(beg + hole) = \*(beg + leftNode);

hole = leftNode;

}

//最后一个位置要被val占用，

\*(beg + hole) = \*(end - 1);

//下来会打乱顺序，再进行上溯

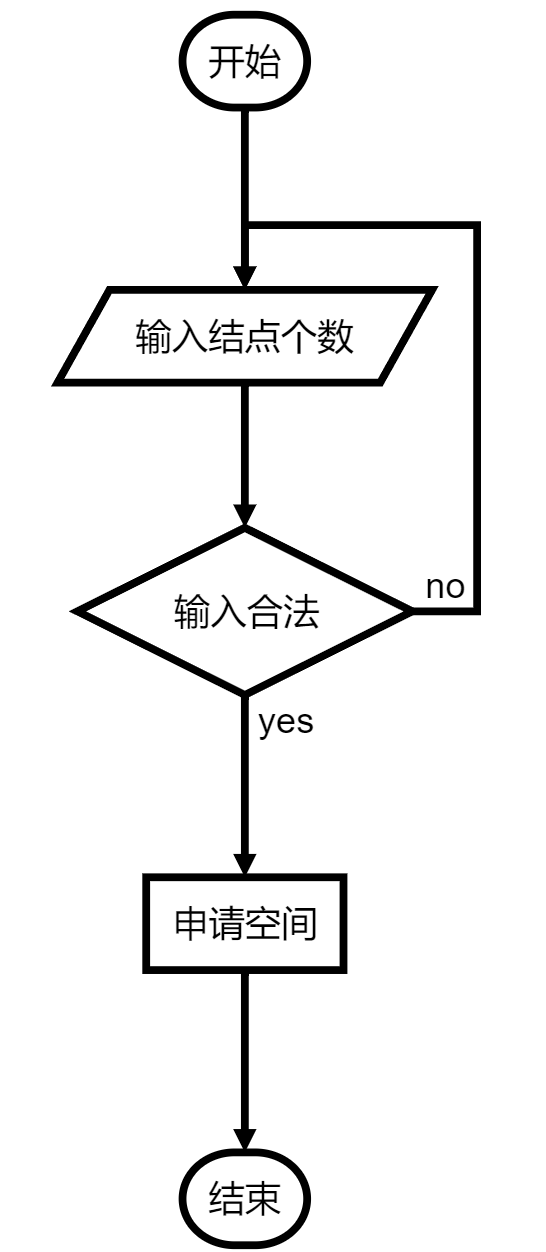
siftFix(beg, beg + hole + 1, cmp);

\*(end - 1) = val;

}

4.3 结点个数输入

4.3.1 流程图



4.3.2 说明

这里需要对用户输入的节点个数进行判断，判断是否合法以及是否为正整数。利用while永真循环，若检查到非法，需要输出输入错误的提示，重新输入该数字，否则跳出循环，结束输入。

利用输入得到的数据，对哈夫曼树数组动态申请空间以及对记录哈夫曼编码的字符数组申请空间。这里也需要对申请空间是否成功进行判断，若不成功则输出相关提示，并结束程序，防止程序继续进行导致更多错误。

需要注意的是，这里输入节点数量为N，但这并不意味着哈夫曼树的节点只有N个，因此由上课所学只是所知，哈夫曼树的节点的只有零度和二度，根据二叉树的性质：具有零度的节点总数 = 具有一度的节点总数 - 1，故哈夫曼树总共有2N - 1个节点，因此数组需要申请2N - 1的大小。同时，哈夫曼编码的长度可以近似看作树的高度，但并不方便计算，因此这里取2N作为存储空间的数组长度。

4.3.3 主要代码

void getN()

{

cout << "请输入节点个数(必须是正整数): ";

while (1)

{

cin >> N;

if (cin.fail() || N <= 0)

{

cout << "输入有误，请重新输入!" << endl;

cin.clear();

cin.ignore(65536, '\n');

}

else

break;

}

//申请哈夫曼树数组存储空间，N个叶子节点共需要2\*N - 1个节点

tree = new(nothrow) TreeNode[2 \* N];

if (tree == NULL)

{

cerr << "申请空间失败!" << endl;

exit(1);

}

//申请哈夫曼编码空间

s = new(nothrow) char[2 \* N];

if (s == NULL)

{

cerr << "申请空间失败!" << endl;

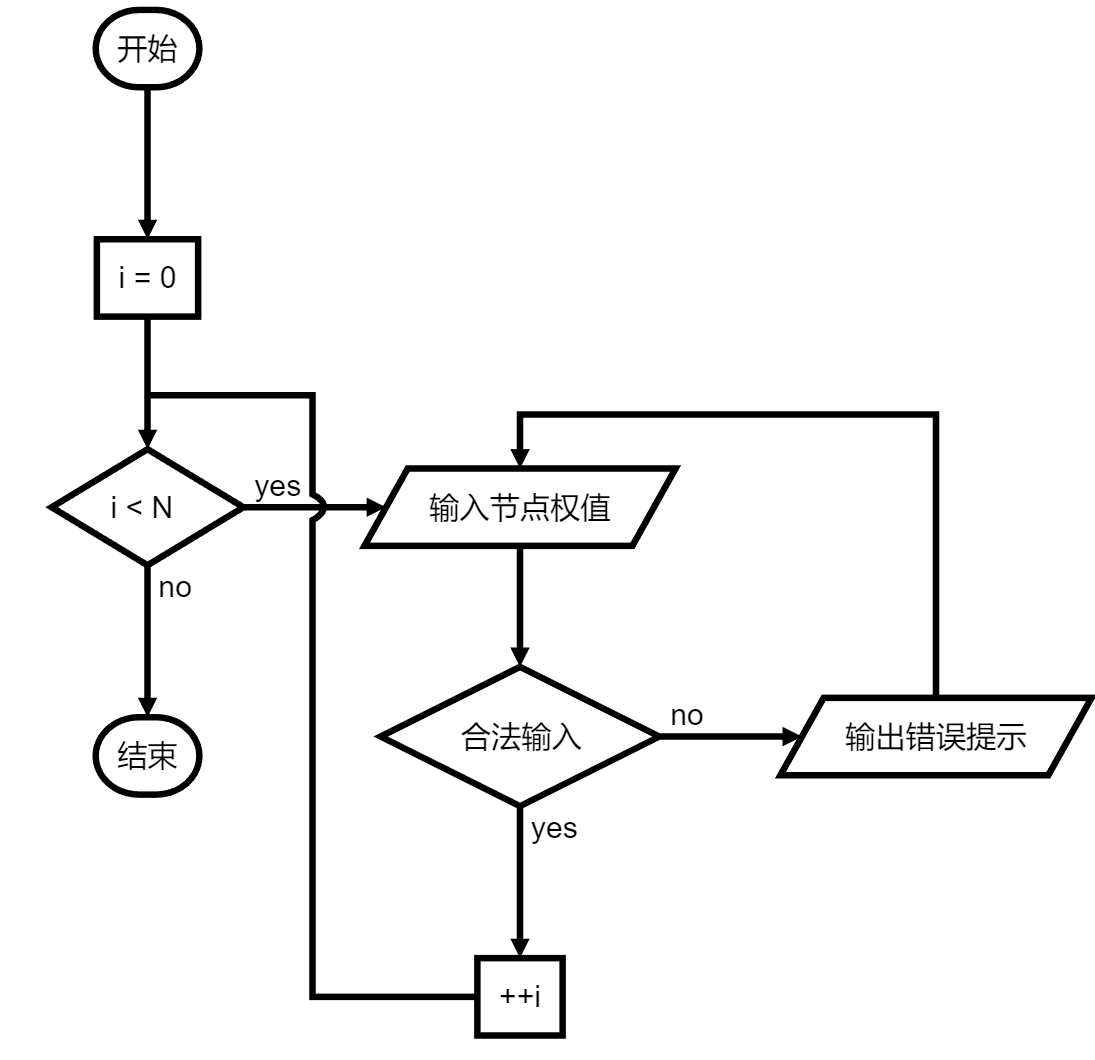
exit(1);

}

}

4.4 叶子结点值输入

4.4.1 流程图



4.4.2 说明

根据输入得到的叶子节点的个数，对每个叶子节点进行值的输入与赋值。同时这里也需要进行错误判断，若输入不合法，则输出输入错误提示，重新进行本轮的输入；反之计入哈夫曼树节点数组。

4.4.3 主要代码

void getNums()

{

cout << "请输入节点(以空格分隔): ";

double temp = 0;

for (int i = 0; i < N; )

{

cin >> temp;

if (cin.fail() || temp <= 0)//非法字符

{

cout << "输入有误，请重新输入!" << endl;

cin.clear();

cin.ignore(65536, '\n');

}

else

{

tree[i].val = temp;

pq.emplace(Node{ tree[i], i });

++i;

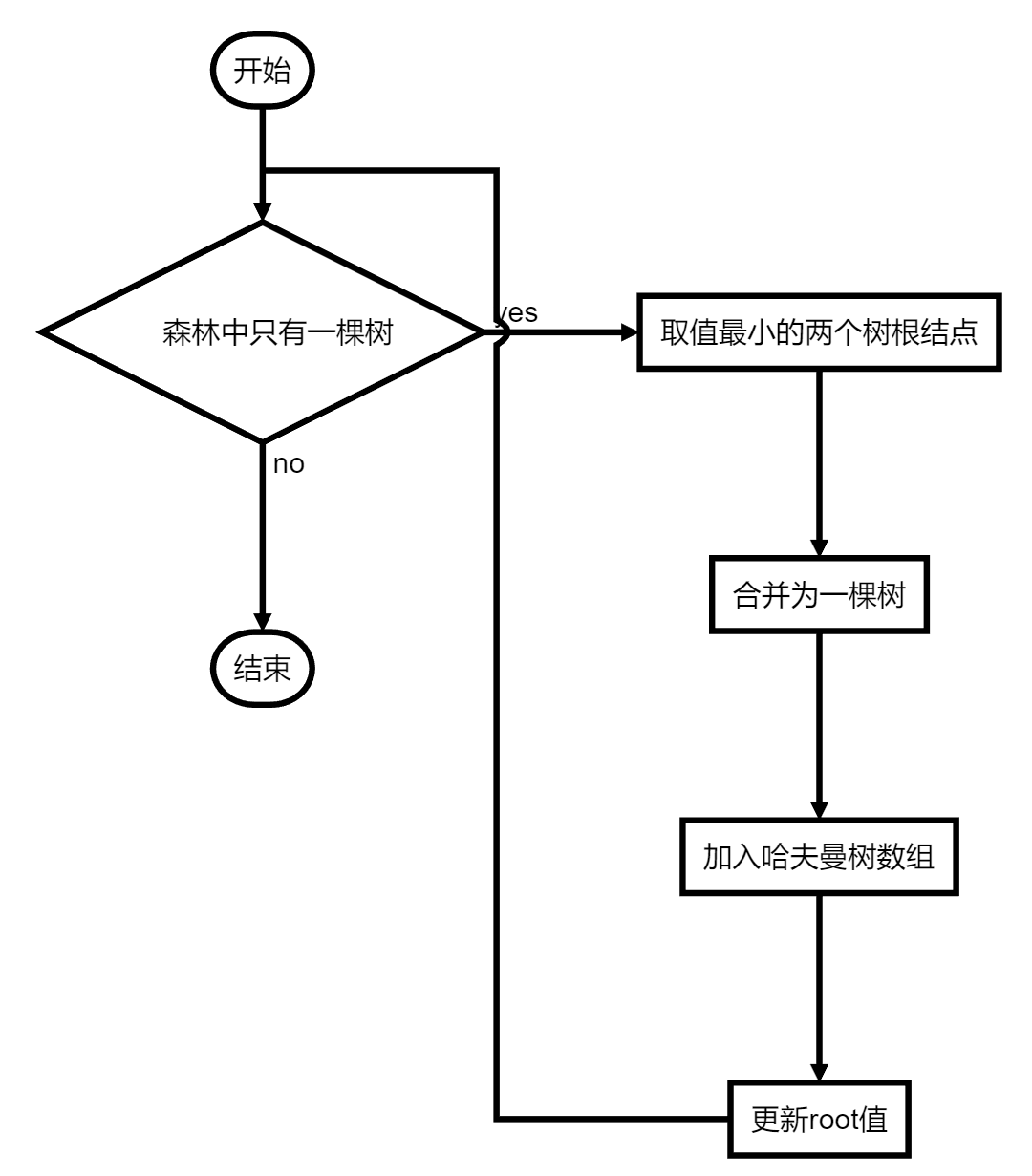
}

}

}

4.5 构建哈夫曼树

4.5.1 流程图



4.5.2 说明

在此之前，在输入叶子节点的值的同时已经将叶子节点以及对应在哈夫曼树数组的下标加入优先队列，现在只需要从优先队列中取出前两个最小的两个节点，将其合并为一棵树。接着需要进行三个个重要的操作，其一是将新的树的节点添加入哈夫曼树数组中，并且将该结点的左右孩子记录；其二是将新节点加入优先队列；其三是更新root的值，以便于获得最终哈夫曼树的根结点。重复上述步骤，直到优先队列中只剩下一个元素，说明其中的元素已经是整棵树的根结点，构建哈夫曼树完成。

每次从优先队列中取出节点是对堆的调整，其时间复杂度为O(log2n)，因此该算法总体的时间复杂度为O(nlog2n)。相比于每一轮都遍历整个数组的O(n2)的时间复杂度，对于n较大的时候具有更好的表现。

4.5.3 主要代码

void buildHuffmanTree()

{

Node temp1, temp2;

int i = N;

while (pq.size() > 1)

{

//取前两个最小的

temp1 = pq.top();

pq.pop();

temp2 = pq.top();

pq.pop();

//两个最小的合成一棵树

tree[i].val = temp1.data.val + temp2.data.val;

tree[i].left = temp1.index;

tree[i].right = temp2.index;

//两个节点合成的树重新进入队列

pq.emplace(Node(tree[i], i));

//更新root值

root = i;

//下标移动

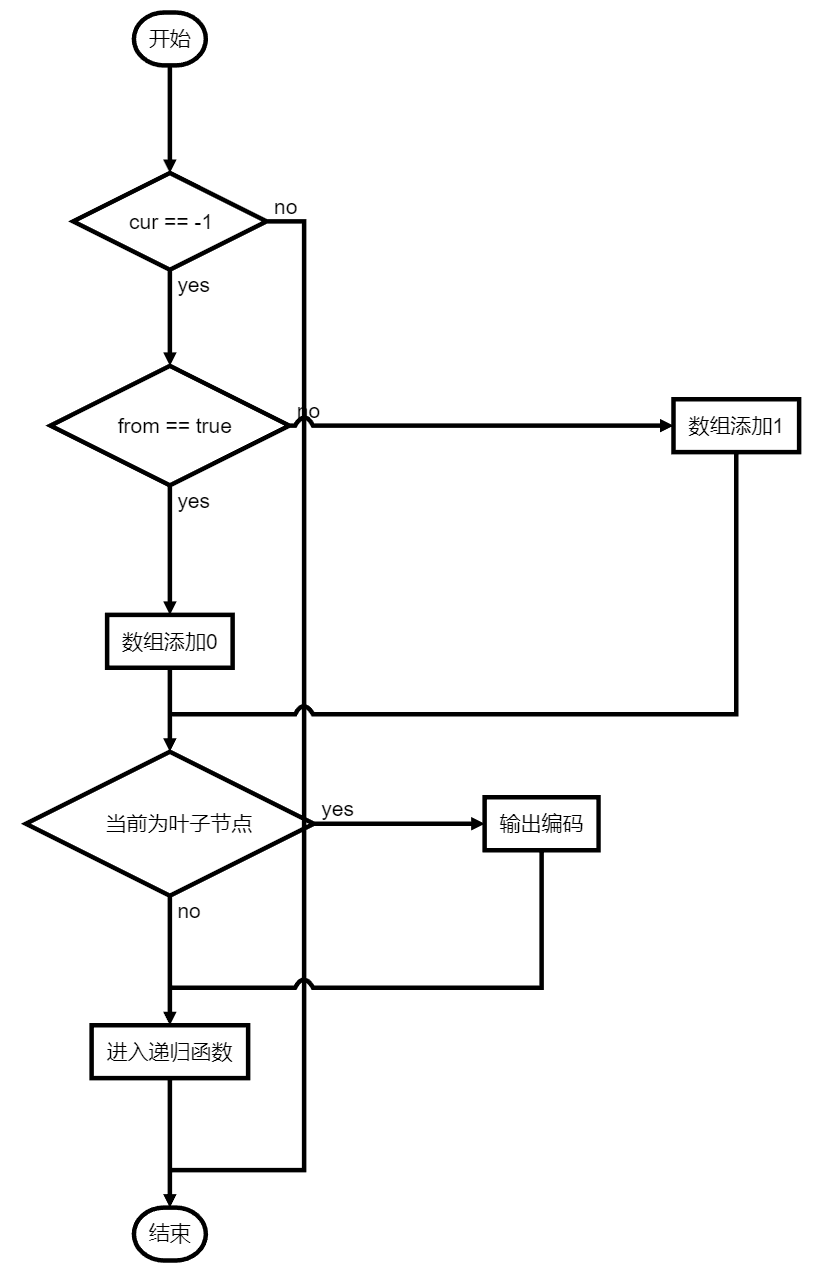
++i;

}

}

4.6 得到哈夫曼编码

4.6.1 流程图



4.6.2 说明

这里通过对树的前序遍历得到哈夫曼编码。我们利用递归方法，其中递归函数的参数有当前遍历到的节点的下标cur、当前节点作为左孩子还是右孩子的标记from以及记录哈夫曼编码的字符数组的当前下标。其中from为布尔型变量，from == true时，表示是左孩子；反之表示右孩子。

开始首先进行递归结束条件的判断：若cur == -1说明当前不存在哈夫曼树的节点，可以结束函数。

然后判断当前节点左孩子还是右孩子，若是左孩子，则在哈夫曼编码数组中添加’0’，否则添加’1’。

接着判断当前节点是否为叶子节点，若是叶子节点，则输出存储在数组中的哈夫曼编码。

最后进入递归函数，分为左孩子和右孩子，左孩子的标记参数设置为true，右孩子的标记参数设置为false。

时间复杂度分析为遍历所有节点，因此为O(n)级别；对于递归所用到的递归工作栈，最多需要树的深度的递归层数，因此空间复杂度为O(h)。

4.6.3 主要代码

void preOrder(int cur, bool from, int idx)

{

//递归结束，到没有的节点

if (cur == -1)

return;

//当前结点作为左孩子，添加0；右孩子，添加1

if (from)

s[idx] = '0';

else

s[idx] = '1';

//到了叶子结点

if (tree[cur].left == -1 && tree[cur].right == -1)

{

//最后添加尾零

s[idx + 1] = '\0';

//设置占位八个空格，左对齐

cout.width(8);

cout.setf(ios::left);

cout << tree[cur].val << "：" << s << endl;

return;//结束

}

//进入递归

preOrder(tree[cur].left, true, idx + 1);

preOrder(tree[cur].right, false, idx + 1);

}

4.7 对外接口

4.7.1 说明

首先调用叶子节点个数的输入函数，然后调用叶子节点值输入函数，接着调用构建哈夫曼树函数，最后前序遍历输出哈夫曼编码。

4.7.2 主要代码

void getHfmCode()

{

getN();

getNums();

buildHuffmanTree();

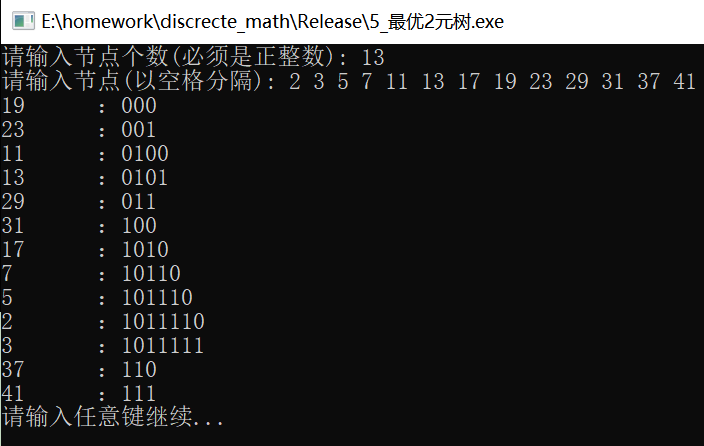
preOrder(tree[root].left, true, 0);

preOrder(tree[root].right, false, 0);

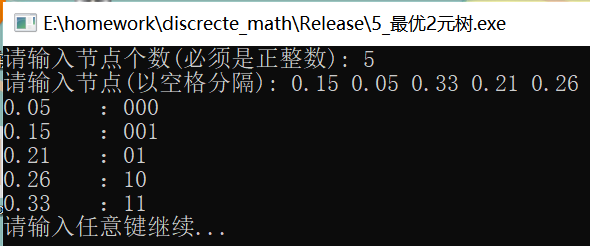
}

5 实验结果

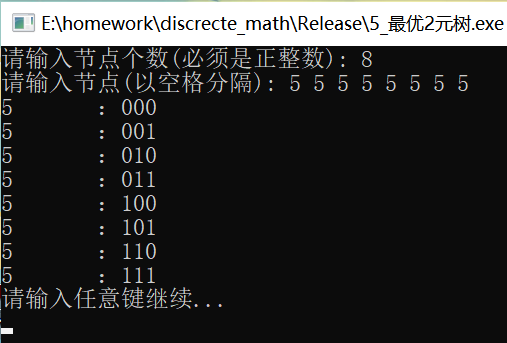
5.1 样例测试



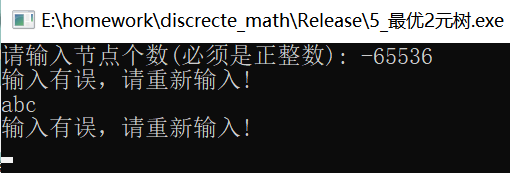
5.2 叶子节点值为小数



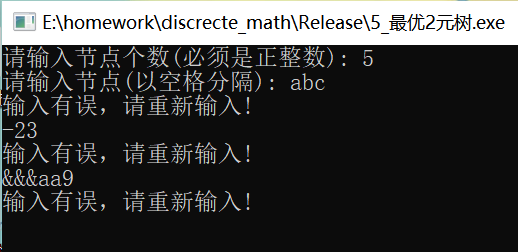
5.3 叶子节点值相同



5.4 输入节点个数错误



5.5 输入节点值错误



6 心得体会

通过对本次题目的解答，我加深了对最优2元树的理解。同时，对于树的相关数据结构的知识也得到了加强。

首先是，通过本次题目，我明白了最优2元树的构造方法。在此过程中，利用到了堆这一种数据结构。原因在于我们需要动态地找到数据序列中最小的两个数据，并且要将两个数据相加，再加入序列当中。为了较好的时间效率，我们堆堆结构做出选择。最优2元树的构造过程其实质上是一种贪心算法的过程，局部最优解可以作为整体最优解，因此每一步都做出当前情况下最好的选择，即可得到整个过程中最好的选择。

其次是对错误输入的判断，特别是如同本题目有着大量用户输入的操作，我们应该保持对输入非法的警惕，尽量地将用户输入的所有情况都考虑到，进行输入错误的提示输出，不至于让程序崩溃。我们也应该在此过程中养成这样良好的编程习惯，虽然这只是一个小程序，但对于日后工作接触大的项目来说有着重要的意义。

对于是树形结构，我们常常用二叉链表存储，受到惯性思维的影响，发现在本题目中用数组结构以及静态链表存储，算作是对惯性思维的突破，更加锻炼了自己的思维和代码能力。

同时，对于哈夫曼编码的求解，本算法具有巨大的实际意义。对于通信传输过程中的编码形式的优化，可以大大减少传输的时间，减少传输的工作量等等优点。因此在实际生活中本算法的应用可以帮助我们更好地理解通信的传输过程与优化方法。同时，最优2元树的应用当然不仅限于此，我们也应该多多思考，广泛应用。