**《离散数学》课程实验报告**

# 《离散数学》课程实验报告5

最优2元树在通信编码中的应用

作 者 姓 名： 毛凌骏

学 号： 2053058

指 导 教 师： 唐剑锋

学院、 专业： 软件学院 软件工程

同济大学

1. **题目简介**

**1.1实验内容：**

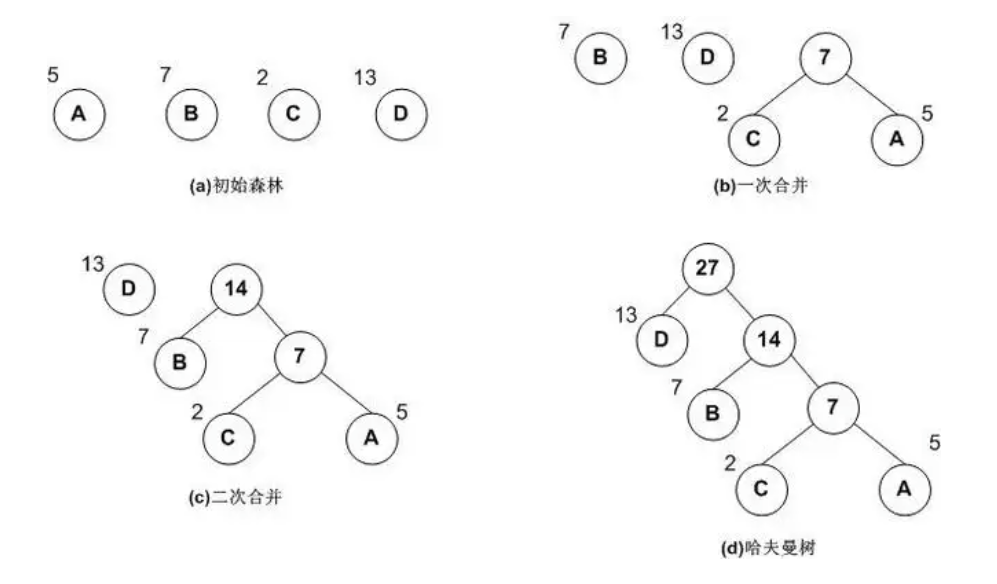
输入一组通信符号的使用频率，求各通信符号对应的前缀码。

**1.2实验环境：**

采用C＋＋编程语言， Visual Studio Code 2019实验环境实现。

1. **解题思路**

在本问题中，通信所需的成本跟通信符号前缀码的长度成正比。当某一通信符号需要反复使用时，其前缀码造成的成本被反复计算，很显然我们希望使用频率高的通信符号前缀码尽可能短。因此本题的一种实现方法为优先队列，将序列的所有数值放入队列，每次选取队列中数值最小的两个元素出队列，再将二者之和放入队列中，如此反复，当队列中只存在一个元素时即为最终解。另一种思路为将前缀码具象为一颗二叉树，通信的成本在二叉树中表现为带权路径长度WPL，因此可建立一颗哈夫曼树，哈夫曼树所有的非叶子节点权重之和即为本题所求的最小成本，叶子节点所在的位置即为本题所求的哈夫曼编码。



1. **数据结构**

**3.1 数据结构设计**

充分利用顺序存储和链式存储结构的特点，可实现对树存储结构的表示。常用的三种表示法为: 双亲表示法、孩子表示法、孩子兄弟表示法。

双亲表示法定义结构数组存放树的结点，每个结点含数据域和双亲域，其中数据域用于存放结点本身信息，双亲域用于指示本结点的双亲结点在数组中的位置。此存储方式可高效查找孩子的双亲，但查找双亲的孩子则需花费较长的时间。 孩子表示法用二叉链表作树的[存储结构](https://so.csdn.net/so/search?q=%E5%AD%98%E5%82%A8%E7%BB%93%E6%9E%84&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/daocaoren_/article/details/_blank)，链表中每个结点的两个指针域分别指向其第一个孩子结点和下一个兄弟结点，可以快速查找孩子节点但难以找到双亲。孩子兄弟表示法同样使用二叉链表[存储](https://so.csdn.net/so/search?q=%E5%AD%98%E5%82%A8%E7%BB%93%E6%9E%84&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/daocaoren_/article/details/_blank)，链表中每个结点的两个指针域分别指向其第一个孩子结点和下一个兄弟结点。由于本题每个树节点仅有两个孩子，且同时需要搜索树节点的孩子和双亲，因此可以用修改过后的孩子表示法存储哈夫曼树，即给每个树节点增添一个双亲指针指向该节点的双亲。

**3.2 类设计**

本项目共定义了三个类，分别是最小堆的模板类、哈夫曼树节点的模板类、哈夫曼树的模板类。哈夫曼树节点类和哈夫曼树类主要是用于建立哈夫曼树，最小堆则是用于优化哈夫曼树建立的效率。在传统的哈夫曼树建立的过程中，每次选取森林中权重最小的两个根节点合并为一颗新的树，这个过程需要遍历森林中所有的元素，复杂度较高。而如若采用最小堆优化，则每次取出最小堆的根节点即可，大大提高了哈夫曼树建立的效率。

**3.2.1 哈夫曼树节点的模板类**

哈夫曼树节点类主要用于存放哈夫曼树的节点信息。其protected属性含有指针域与数据域，数据域存放的是节点权重value，指针域存放的有指向左孩子的指针lChild、指向右孩子的指针rChild、指向双亲的指针parent。哈夫曼树节点类的public属性主要用于存放各项操作，主要为构造函数和析构函数。

//哈夫曼树节点的模板类

**template** <**class** Type>

**class** haffmanTreeNode {

**protected**:

    Type value;                                     //数据域

    haffmanTreeNode\* lChild, \* rChild, \* parent;    //指针域

**public**:

    haffmanTreeNode() :value(0), lChild(NULL), rChild(NULL), parent(NULL) {}    //无参构造函数

    haffmanTreeNode(Type v):value(v),lChild(NULL),rChild(NULL),parent(NULL){}   //带一个参数的构造函数

**friend** **class** haffmanTree<Type>;

**friend** **class** minHeap<haffmanTreeNode<Type>\*>;

};

**3.2.2 哈夫曼树的模板类**

哈夫曼树类在哈夫曼树节点类的基础上构成了一颗完整的树，其protected属性包含哈夫曼树的节点个数size和用于存放哈夫曼树节点元素的数组elems。哈夫曼树的public属性存放各项操作，包括构造函数、析构函数、计算哈夫曼树的最短带权路径长度WPL等。其中项目主体包括输入处理放于哈夫曼树的构造函数中，以提高程序的完整性。

//哈夫曼树的模板类

**template** <**class** Type>

**class** haffmanTree {

**protected**:

**int** size;                       //哈夫曼树的节点个数

    haffmanTreeNode<Type>\*\* elems;  //存放哈夫曼树节点元素

**public**:

    haffmanTree();                                  //构造函数

    ~haffmanTree();                                 //析构函数

**int** calculateWPL();                             //计算哈夫曼树的WPL

**friend** **class** haffmanTreeNode<Type>;

**friend** **class** minHeap<haffmanTreeNode<Type>\*>;

};

**3.2.3 最小堆的模板类**

最小堆，是一种经过排序的[完全二叉树](https://baike.baidu.com/item/%E5%AE%8C%E5%85%A8%E4%BA%8C%E5%8F%89%E6%A0%91/7773232?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E6%9C%80%E5%B0%8F%E5%A0%86/_blank)，其中任一非终端节点的数据值均不大于其左子节点和右子节点的值，可以看作是一种优先级队列的实现，有些应用场景需要从队列中获取最小的或者最大的元素，而且不要求数据全部有序，使用最小堆或者最大堆能很好的解决这类问题。最小堆的元素是按[完全二叉树](https://so.csdn.net/so/search?q=%E5%AE%8C%E5%85%A8%E4%BA%8C%E5%8F%89%E6%A0%91&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/Huoon/article/details/_blank)的顺序存储方式存放在一维数组中，其根节点对应数组的第一个元素，后面接着依次从上到下每一层树节点从左往右排列，按照这个顺序将一段[线性](https://so.csdn.net/so/search?q=%E7%BA%BF%E6%80%A7&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/Huoon/article/details/_blank)内存结构映射为一个树结构。

//最小堆的模板类

**template** <**class** Type>

**class** minHeap {

**protected**:

**int** size;                   //当前容量

**int** maxSize = 100000;       //最大容量

    Type \*heap;                 //存放堆元素的数组

**public**:

    minHeap(**int** n, Type\* elems);        //构造函数

    ~minHeap();                         //析构函数

    Type removeMin();                   //最小元素出堆

**void** insert(**const** Type& x);         //插入元素

**void** siftDown(**int** start);           //自下而上调整最小堆

**void** siftUp(**int** start);             //自上而下调整最小堆

**friend** **class** haffmanTree<Type>;

**friend** **class** haffmanTreeNode<Type>;

};

最小堆的操作主要包括构造函数、析构函数、最小元素出堆、插入元素、自下而上调整最小堆、自上而下调整最小堆等。通过反复调整可以使一颗原本无序的二叉树变成最小堆，每次取根节点即为序列中的最小值。最小堆各成员函数的具体实现如下：

/\*最小堆成员函数的体外实现\*/

//最小堆构造函数

**template** <**class** Type>

minHeap<Type>::minHeap(**int** n, Type\* elems) {

    size = n;

    heap = **new** Type[maxSize];

**for** (**int** i = 0; i < n; i++) {

        heap[i] = elems[i];

    }

**int** start = (n - 2) / 2;

**while** (start >= 0) {

        siftDown(start);

        start--;

    }

}

//最小堆析构函数

**template** <**class** Type>

minHeap<Type>::~minHeap() {

**delete** heap;

}

//自上而下调整最小堆

**template**<**class** Type>

**void** minHeap<Type>::siftDown(**int** start) {

**int** i = start, j = 2 \* i + 1;

    Type temp = heap[i];

**while** (j < size) {

**if** (j < size - 1 && heap[j]->value > heap[j + 1]->value)

            j++;

**if** (heap[j]->value < temp->value) {

            heap[i] = heap[j];

            i = j;

            j = 2 \* i + 1;

        }

**else**

**break**;

    }

    heap[i] = temp;

}

//自下而上调整最小堆

**template** <**class** Type>

**void** minHeap<Type>::siftUp(**int** start) {

**int** i = start, j = (i - 1) / 2;

    Type temp = heap[i];

**while** (i > 0) {

**if** (heap[j]->value > temp->value) {

            heap[i] = heap[j];

            i = j;

            j = (j - 1) / 2;

        }

**else**

**break**;

    }

    heap[i] = temp;

}

//插入元素

**template** <**class** Type>

**void** minHeap<Type>::insert(**const** Type &x) {

    heap[size] = x;

    siftUp(size);

    size++;

}

//最小元素出堆

**template** <**class** Type>

Type minHeap<Type>::removeMin() {

    Type result;

**if** (!size)

**return** 0;

**else** {

        result = heap[0];

        heap[0] = heap[size - 1];

        size--;

        siftDown(0);

    }

**return** result;

}

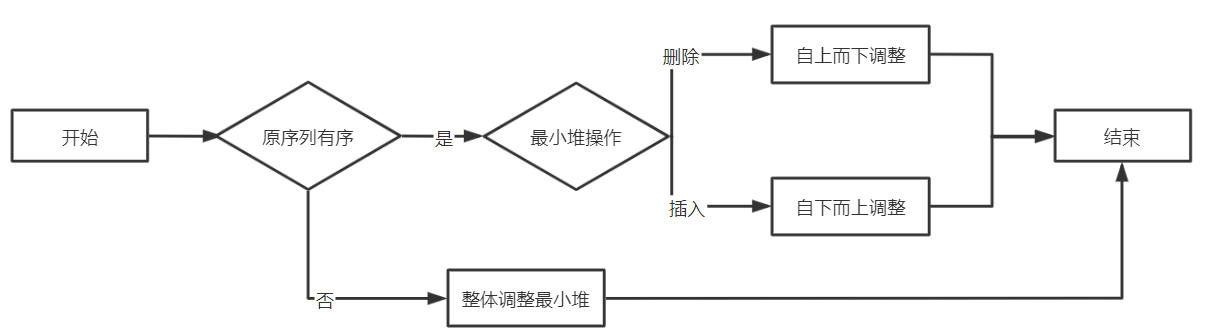
1. **核心算法**

## **4.1 最小堆整体调整的实现**

### **4.1.1 最小堆调整实现思路**

最小堆调整可分为自上而下的调整方法和自下而上的调整方法。假若原序列满足最小堆，当插入一个元素时可将该元素放于序列的最末端，然后采取自下而上的调整法；当取出堆内最小元素后可将序列最末端元素放于序列首地址，最小堆大小减一后采用自下而上的调整法。当原序列整体无序，可以从后往前访问最小堆的每个非叶子节点即分支节点，对每个非叶子节点使用自下而上调整法，反复执行后原序列满足最小堆的定义。

### **4.1.2 最小堆调整实现流程图**



### **4.1.3 最小堆调整实现代码**

//自上而下调整最小堆

**template**<**class** Type>

**void** minHeap<Type>::siftDown(**int** start) {

**int** i = start, j = 2 \* i + 1;

    Type temp = heap[i];

**while** (j < size) {

**if** (j < size - 1 && heap[j]->value > heap[j + 1]->value)

            j++;

**if** (heap[j]->value < temp->value) {

            heap[i] = heap[j];

            i = j;

            j = 2 \* i + 1;

        }

**else**

**break**;

    }

    heap[i] = temp;

}

//自下而上调整最小堆

**template** <**class** Type>

**void** minHeap<Type>::siftUp(**int** start) {

**int** i = start, j = (i - 1) / 2;

    Type temp = heap[i];

**while** (i > 0) {

**if** (heap[j]->value > temp->value) {

            heap[i] = heap[j];

            i = j;

            j = (j - 1) / 2;

        }

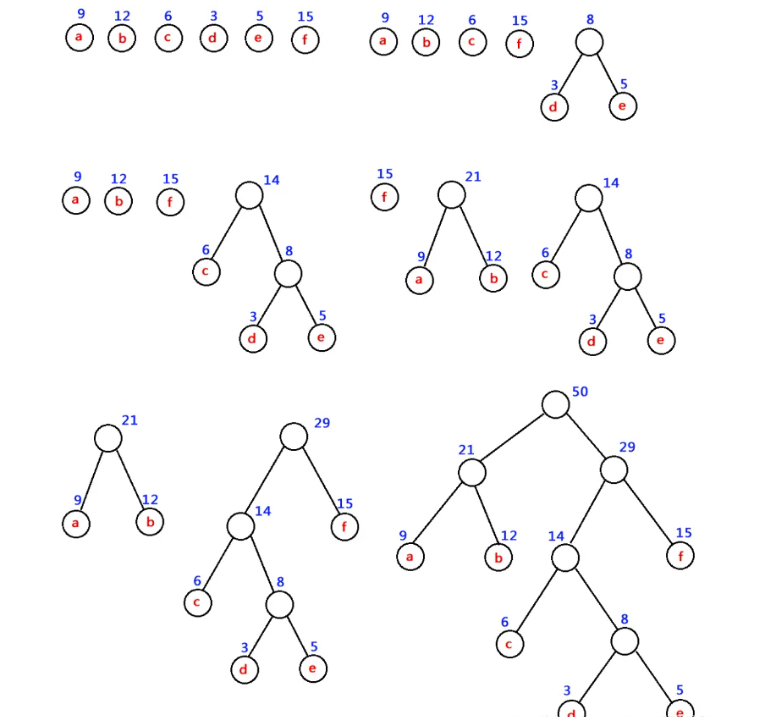
**else**

**break**;

## **4.2 建立哈夫曼树的实现**

### **4.2.1 建立哈夫曼树实现思路**

建立哈夫曼树首先将序列生成森林，初始状态下每个节点即为森林中的一棵树，每次选取森林中根节点权值最小的两棵树合并，直至森林中最后只剩下一颗树时即为所求的哈夫曼树。在引入最小堆后搜索待合并目标树的复杂度大大降低。在序列初始为森林时，序列所有值入堆，使用最小堆的整体调整法让无序树满足最小堆的定义。之后每次取根节点权值最小的两棵树时不需要遍历搜索，取最小堆的首元素即可。



### **4.2.2 建立哈夫曼树实现思路**

### 

### **4.2.3 建立哈夫曼树实现代码**

//哈夫曼树构造函数

**template** <**class** Type>

haffmanTree<Type>::haffmanTree() {

    //输入及错误处理

**int** n;

    cout << "请输入节点个数：";

**while** (1) {

        cin >> n;

**if** (cin.fail()) {

            cout << "输入错误，请重新输入：";

            cin.clear();

**char** t;

**while** ((t = cin.get()) != '\n');

        }

**else** **if** (n < 0 || n > 10000) {

            cout << "节点个数应在0到1000之间，请重新输入：";

            cin.clear();

**char** t;

**while** ((t = cin.get()) != '\n');

        }

**else**

**break**;

    }

    //创建森林

    size = 2 \* n - 1;

    elems = **new** haffmanTreeNode<Type>\*[2 \* n - 1];

**for** (**int** i = 0; i < 2 \* n - 1; i++) {

        haffmanTreeNode<Type>\* p = **new** haffmanTreeNode<Type>;

        elems[i] = p;

    }

    cout << "请输入节点：";

**for** (**int** i = 0; i < n; i++) {

**while** (1) {

            cin >> elems[i]->value;

**if** (cin.fail() || elems[i]->value < 0) {

                cout << "输入错误，请重新输入整段序列：";

                cin.clear();

**char** t;

**while** ((t = cin.get()) != '\n');

                i = 0;

            }

**else**

**break**;

        }

    }

    //利用最小堆创建哈夫曼树

    minHeap<haffmanTreeNode<Type>\*> minh(n, elems);

**for** (**int** i = n; i < 2 \* n - 1; i++) {

        haffmanTreeNode<Type>\* s1 = minh.removeMin();

        haffmanTreeNode<Type>\* s2 = minh.removeMin();

        elems[i]->lChild = s1;

        elems[i]->rChild = s2;

        elems[i]->value = s1->value + s2->value;

        s1->parent = elems[i];

        s2->parent = elems[i];

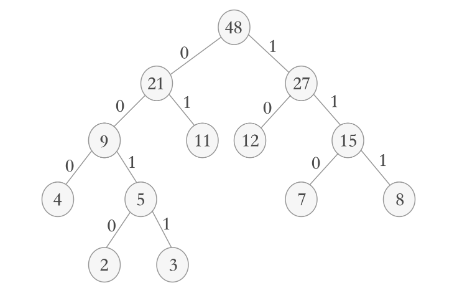
        minh.insert(elems[i]);

    }

}

## **4.3 构造哈夫曼编码的实现**

### 4.3.1 构造哈夫曼编码实现思路



### 4.3.2 构造哈夫曼编码实现代码

//打印哈夫曼编码

**template** <**class** Type>

**void** haffmanTree<Type>::printHaffmanCode(string& code) {

    printHaffmanCode(code, elems[size - 1]);

}

//递归打印哈夫曼编码

**template** <**class** Type>

**void** haffmanTree<Type>::printHaffmanCode(string &code, haffmanTreeNode<Type>\* root) {

**if** (root->lChild == NULL && root->rChild == NULL) {

        cout << root->value << "： " << code << endl;

    }

**else** {

        string codeBefore = code;

        code += "0";

        printHaffmanCode(code, root->lChild);

        code = codeBefore;

        code += "1";

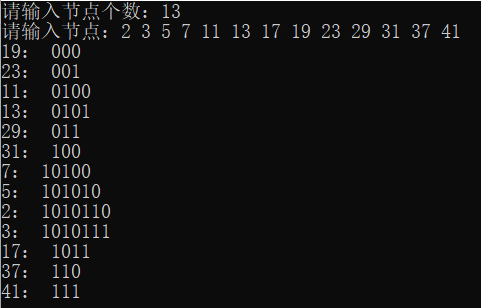
        printHaffmanCode(code, root->rChild);

    }

}

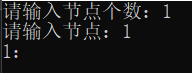
1. **实验测试**

## **5.1 常规测试**



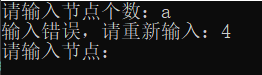
## **5.2 边界测试**

只有一个节点的情况：

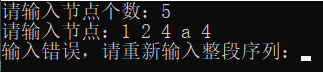


## **5.3 错误测试**

节点个数输入错误：



序列输入错误：



1. **心得体会**

[哈夫曼](https://baike.baidu.com/item/%E5%93%88%E5%A4%AB%E6%9B%BC?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E5%93%88%E5%A4%AB%E6%9B%BC%E7%BC%96%E7%A0%81/_blank)编码又称霍夫曼编码，是一种编码方式，是可变[字长](https://baike.baidu.com/item/%E5%AD%97%E9%95%BF/97660?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E5%93%88%E5%A4%AB%E6%9B%BC%E7%BC%96%E7%A0%81/_blank)编码的一种，其通过哈夫曼树的数据结构大幅压缩了信号成本。设某信源产生有五种符号u1、u2、u3、u4和u5，对应概率P1=0.4，P2=0.1，P3=P4=0.2，P5=0.1。建立哈夫曼编码的思路如下：

首先，将符号按照概率由大到小排队，如图1所示。编码时，从最小概率的两个符号开始，可选其中一个支路为0，另一支路为1。这里，我们选上支路为0，下支路为1。再将已编码的两支路的概率合并，并重新排队。多次重复使用上述方法直至合并概率归一时为止。注意哈夫曼编码的构造方法并不唯一，其原因是两支路概率合并后重新排队时，可能出现几个支路概率相等，造成排队方法不唯一。一般，若将新合并后的支路排到等概率的最上支路，将有利于缩短码长方差，且编出的码更接近于等长码。

通过本次实验，不仅让我对数据结构中哈夫曼树的应用有了更为深刻的了解，更让我对离散数学中图论部分有了新的体会，对树的结构有了新的认知。