

## 

**《离散数学》课程实验报告**

# 题目 最优二元树

## 姓 名 尹诚成

学 号 2351279

学 院 计算机科学与技术学院

专 业 软件工程

教 师 唐剑锋

二〇二四 年 十二 月 二十 日

**目录**

1 实验目的1

2 实验内容1

3 实验环境1

4 实验原理1

4.1 最优二元树1

4.2 最优二元树编码2

5 实验过程2

5.1 实验思路2

5.2 实验设计2

5.2.1 数据结构设计2

5.2.2 结构体TreeNode设计3

5.2.3 程序主体架构设计4

5.3 程序功能实现5

5.3.1 输入最优二元树节点个数与节点值功能的实现 5

5.3.2 退出程序功能的实现6

5.3.3 动态内存申请失败后异常处理的实现6

5.4 核心算法实现6

5.4.1构建最优二元树算法的实现6

5.4.2 遍历最优二元树算法的实现7

6 实验数据分析7

7 实验心得9

8 程序源文件9

1. **实验目的**

本次实验主要目的是理解和实现最优二元树的算法，通过实际编程来理解算法的工作原理，并将理论应用于实际问题解决中。本实验旨在加强学生对二叉树的理解，并提高使用计算机语言解决问题的能力。

1. **实验内容**

问题描述：输入一组通信符号的使用频率，并求出各通信符号对应的前缀码。并使其平均编码长度最小。

1. **实验环境**

程序开发语言：C++

集成开发环境：Microsoft Visual Studio 2022 (Release模式)

编译运行环境：本项目适用于 x86 架构和 x64 架构

1. **实验原理**

4.1.最优二元树

最优二元树，通常被称为哈夫曼树（Huffman Tree），是一种特殊的二叉树， 用于有效编码。在哈夫曼树中，每个叶子节点代表一个字符或通信符号，而每个节点的权值则代表该字符的使用频率或概率。哈夫曼树的主要特点是提供一种高效的编码方法，其中最常用的字符有最短的编码，而不常用的字符则有较长的编码。

哈夫曼树的构建基于这样一个原则：给定一组权值，构建一棵具有最小带权路径长度的二叉树。带权路径长度指的是树中所有叶子节点的权值与其到根节点路径长度乘积的总和。 最优二元树（哈夫曼树）构建过程：

(1)初始化：根据给定的每个数据的权值，创建一个森林，其中每个数据为一个节点，构成一个单节点树。

(2)构建树：在森林中选取两个权值最小的树合并，作为左右子树构造一棵新的二叉树，新树的根节点权值为这两棵子树根节点权值之和；

(3)重复合并：重复上述过程，每次合并后更新森林，直到森林中只剩下一棵树为止。这棵树就是最终的哈夫曼树。

4.2.最优二元树编码

最优二元树编码（哈夫曼编码）是一种变长编码方式，用于无损数据压缩。 在哈夫曼编码中，每个字符根据其出现频率被赋予一个唯一的二进制码。字符的编码长度与其频率成反比，即出现频率高的字符有更短的编码。

最优二元树编码编码原则：

(1)不等长编码：字符的编码长度不同，频率高的字符编码短，频率低的字符编码长；

(2)前缀无歧义：任一字符编码都不是其他字符编码的前缀，保证了编码的前缀性。

最优二元树编码编码过程：

(1)构建哈夫曼树：根据字符的使用频率构建哈夫曼树；

(2)生成编码：从根节点到每个叶子节点的路径确定了该节点字符的编码。 路径中左分支表示 “0”，右分支表示“1”。

哈夫曼编码广泛应用于数据压缩领域，如文件压缩、多媒体数据编码等。它通过有效减少常用字符的编码长度，达到压缩数据的目的。

1. **实验过程**

5.1.实验思路

本实验的核心思路是利用一维向量std::vector<HuffmanTreeNode> nodes存储通信符号的使用频率，并通过构建最优二元树（哈夫曼树）来为每个通信符号生成前缀码。在这个过程中，使用链表来保存这个最优二元树，并通过树的遍历方法输出前缀码。

链表的概念主要体现在构建哈夫曼树的过程中。链表不是以传统的 std::list形式出现，而是通过std::vector< HuffmanTreeNode > 来实现，其中 HuffmanTreeNode是自定义的结构，用于表示树中的每个节点。每个 TreeNode 包含指向其左右子节点的指针（left和right），这些指针实际上形 成了一个链表结构。 树的遍历在本程序中体现在printHuffmanTree函数中，该函数用于输出哈夫曼编码。遍历是通过递归调用实现的，分别对树的左子树和右子树进行遍历。

5.2.实验设计

5.2.1.数据结构设计

在本实验中，数据结构设计的核心是构建和管理哈夫曼树。关键的数据结构包括：

(1)频率数组：使用std::vector类型来存储通信符号的使用频率。这个一维向量是构建哈夫曼树的基础，它记录了每个符号的使用频率；

(2)节点向量：利用std::vector< HuffmanTreeNode >来管理哈夫曼树的节点。

(3)哈夫曼树：通过连接 HuffmanTreeNode 结构构成的哈夫曼树。每个节点包含一个数字（表示频率或权值），以及指向其左右子节点的指针。

5.2.2.结构体HuffmanTreeNode设计

HuffmanTreeNode结构体是实现最优二元树（哈夫曼树）的基础。它表示树中的每一个节点，包括节点存储的数据（如频率）和指向子节点的指针。其数据成员、构造函数、运算符重载定义及含义如下：

int num：表示节点存储的数据，如频率或权值

std::string：表示节点的编码

HuffmanTreeNode\* leftChild, \* rightChild：指针，分别指向该节点的左子节点和右子节点

HuffmanTreeNode (int n): num(n) , leftChild(NULL) , rightChild(NULL) {}:构造函数。初始化HuffmanTreeNode实例，设置其num成员为传入的参数值n，将左右子节点指针left和right初始化为NULL，表示在创建节点时，它不具有任何子节点。

5.2.3.程序主体架构设计

程序主体架构设计为：

(1)初始化和创建树节点：当程序开始执行时，首先进入main函数。在这里，程序首先执行初始化操作，包括清除屏幕并显示用户界面。接下来，程序请求用户输入最优二元树节点的数量，这是通过调用inputInteger函数实现的。用户输入后，使用这个数值来创建std::vector<HuffmanTreeNode> nodes实例，此实例代表最优二元树的所有节点；

(2)输入节点的值：一旦最优二元树初始化完成，程序接着引导用户依次输入每个节点的值（频率），这些值通过inputInteger函数设置到nodes的num中。

(3)构建最优二元树：输入完最优二元树的所有节点值后，main函数调用createHuffmanTree 函数来构建最优二元树。

(4)输出结果：构建完最优二元树后，程序通过调用printHuffmanTree 函数遍历树并输出哈夫曼编码。最后，main 函数中包含用户退出程序的选项。一旦用户选择退出，程序将清理所有资源并结束执行。

图 5.2.3 程序主体架构流程设计图

5.3 程序功能实现

5.3.1.输入最优二元树节点个数与节点值功能的实现

程序通过调用 inputInteger 函数最优二元树节点个数与节点值。

inputInteger 函数用于获取用户输入的整数，同时限制输入必须在指定的范围内。inputInteger 函数对输入非法的情况进行了处理，代码具体执行逻辑如下：

(1)进入一个无限循环，它会一直运行直到用户提供有效的输入；

(2)用户的输入被读取到num变量中，这里采用double类型来接收输入以便后续检查；

(3)进行输入验证：std::cin.fail()检查输入流的状态是否异常，确保没有发生数据类型输入错误，num==static\_cast<int>(num)检查用户输入是否为整数，通过将其转换为整数再比较，num < minNum || num > maxNum确保输入在指定的范围内；

(4)合法输入处理：如果用户提供了合法的输入，函数会返回转换后的整数值；

(5)非法输入处理：如果用户提供的输入不合法，函数会输出错误消息，清除输入流的错误状态，丢弃输入缓冲区中的内容，并继续循环以等待用户提供合法的输入。

5.3.2.退出程序功能的实现

退出程序的功能是通过在main函数内部的一个循环结构实现的。此功能允许用户在完成操作后选择是否退出程序。

程序功能实现思路：

(1)程序使用 do-while 循环来反复执行程序和询问用户是否退出程序。在每次循环的开始，使用system("cls")清除屏幕，以提供清晰的界面；

(2)在完成一轮操作后，程序会输出询问用户是否退出程序的提示：“是否退出程序 [y/n]: ”。此时，程序再次调用inputLogicalValue函数，这次用于接收用户的退出决定。参数被设置为'n'（不退出）和'y'（退出）；

(3)如果用户输入'y'（对应真值），则inputLogicalValue函数返回true，导致 do-while 循环结束，程序随之退出。如果用户输入'n'（对应假值），inputLogicalValue 函数返回 false，do-while 循环继续，用户可以进行新一轮操作。

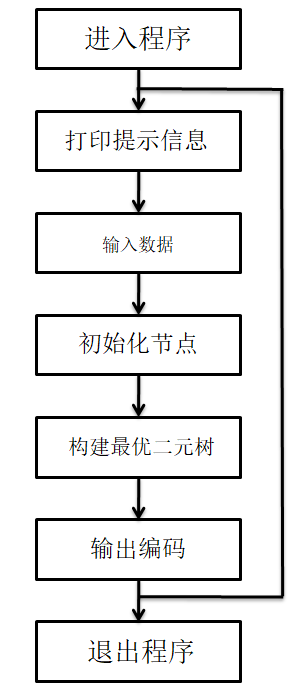


图 5.3.2 程序主体架构设计流程图

5.4 核心算法实现

5.4.1.构建最优二元树算法的实现

最优二元树（哈夫曼树）是根据输入数据（符号频率）构建的特殊树结构， 旨在实现高效编码。此算法通过组合频率最小的两个节点来逐步构建树。createHuffmanTree 函数实现思路如下：

(1)循环直到节点向量只剩一个元素（树的根节点）；

(2)在每次循环中，首先调用std::sort对节点进行排序；

(3)选取两个最小的节点作为左右子节点，创建一个新的父节点，其值为两 子节点之和；

(4)将新创建的父节点添加回节点向量中。

算法特点：保证树的最终形态在所有可能的树结构中具有最小的加权路径长度，确保了编码的高效性。

5.4.2.遍历最优二元树算法的实现

递归遍历算法用于生成哈夫曼编码。它遍历树的每个节点，根据节点的位置确定编码。printHuffmanTree 函数实现思路如下：

(1)从根节点开始，递归遍历每个节点；

(2)每向左遍历一步，编码添加"0"，每向右一步，编码添加"1"；

(3)当到达叶子节点时，打印节点的值（频率）和累计的前缀码；

(4)使用std::setw 确保输出格式整齐。

算法特点：递归方法简化了遍历过程，使代码更加简洁易读。同时，确保了每个符号的唯一且高效的编码生成。

1. **实验数据分析**

实验数据分析1：分别输入超过上下限的整数、浮点数、字符、字符串，可以验证程序对输入非法的情况进行了处理，并要求用户重新输入最优二元树节点个数。

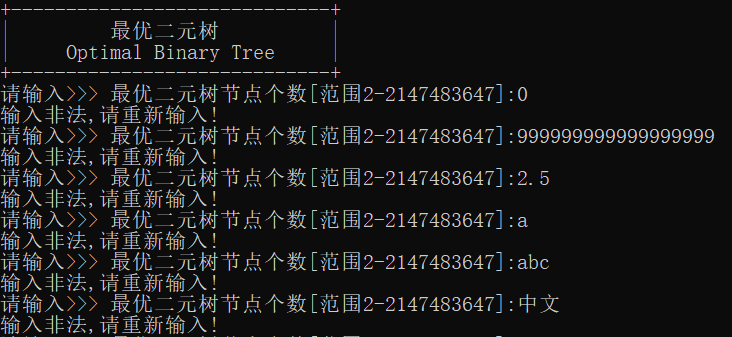


图 6.1 实验数据 1

实验数据分析2：分别输入超过上下限的整数、浮点数、字符、字符串，可以验证程序对输入非法的情况进行了处理，并要求用户重新输入最优二元树节点值。

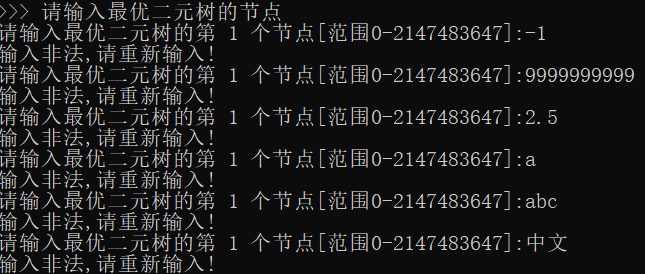


图 6.2 实验数据 2

实验数据分析3：输入最优二元树的13个节点（2、3、5、7、11、13、17、19、23、29、31、37、41）构建最优二元树，程序输出最优二元树编码 （19:000、23:001、11:0100、13:0101、29:011、31:100、17:1010、7:10110、5:101110、2:1011110、3:1011111、37:110、41:111）。

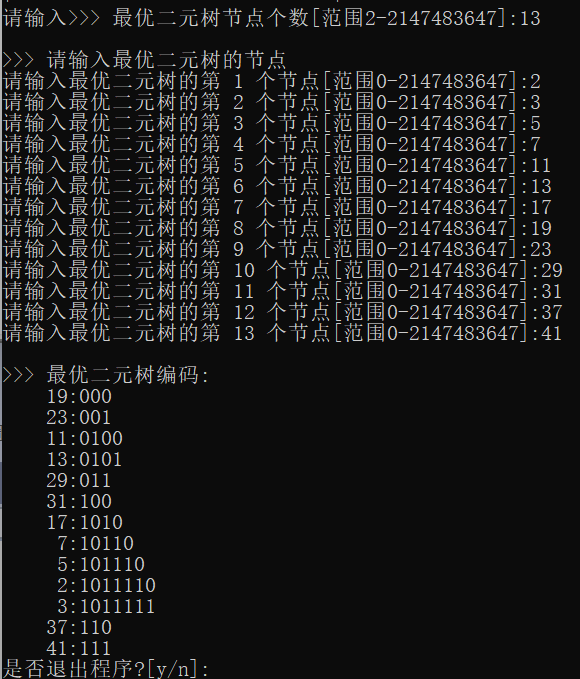


图 6.3 实验数据 3

实验数据分析4：当输入n时，程序清空屏幕并执行下一次最优二元树的构建。

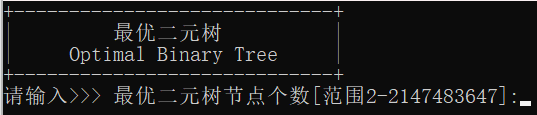


图 6.4 实验数据 4

实验数据分析5：当输入y时，程序退出。

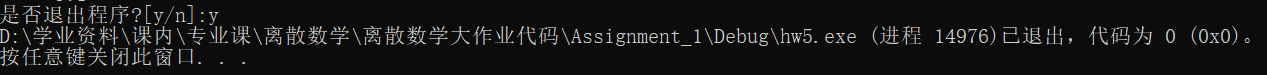


图 6.5 实验数据 5

1. **实验心得**

在本次最优二元树实验中，我获得了许多宝贵的经验和深刻的理解。以下是我的实验心得：

(1)数据结构应用：这次实验使我深入理解了二叉树，特别是哈夫曼树在数 据处理和存储中的实际应用。通过亲手构建哈夫曼树，我更加清晰地理解了其数 据结构和工作原理。

(2)编程技能提升：在实验中，我使用C++语言编写代码，这对于增强我的编程技能。

(3)算法理解实现：通过本实验，我学习了构建哈夫曼树的算法，并亲自实现了它。这不仅提高了我的算法理解能力，也锻炼了我解决问题和实现复杂算法的能力。在实现哈夫曼树的过程中，我使用递归来遍历树和生成编码，这加深了我对递归概念的理解。

(4)用户交互设计：实验的一个重要方面是设计用户友好的交互界面。我学会了如何有效地收集用户输入并提供清晰的输出，这是软件开发中一个重要的技能。

(5)问题解决调试：在实验过程中，我遇到了各种各样的错误和问题。通过调试和修改代码，我提高了自己的问题解决能力。

(6)理论实践结合：这次实验是理论知识和实践技能结合的极好示例。它使我理解了理论知识在实际编程中的应用，以及如何将复杂的理论转化为实际的代码。

1. **程序源文件**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include<iostream>

#include<vector>

#include<iomanip>

#include<conio.h>

#include<algorithm>

#define MEMORY\_ALLOCATION\_ERROR -1

/\*define the structure of HuffmanTreeNode\*/

struct HuffmanTreeNode {

int num;

std::string code;

HuffmanTreeNode\* leftChild, \* rightChild;

HuffmanTreeNode(int n) :num(n), leftChild(NULL), rightChild(NULL) {}

/\*operator < is used to sort the nodes in the priority queue\*/

bool operator<(const HuffmanTreeNode& otherNode)const {

return this->num < otherNode.num;

}

/\*operator << is used to print the HuffmanTreeNode\*/

friend std::ostream& operator<<(std::ostream& out, const HuffmanTreeNode& huffmanTreeNode) {

out << std::setiosflags(std::ios::right) << std::setw(6) << huffmanTreeNode.num << ":" << huffmanTreeNode.code;

return out;

}

};

/\*

\* Function Name: createHuffmanTree

\* Function: create a Huffman tree from a vector of HuffmanTreeNode

\* Input Parameters:std::vector<HuffmanTreeNode>& nodes

\* Return Value: void

\*/

void createHuffmanTree(std::vector<HuffmanTreeNode> &nodes) {

while (nodes.size() > 1) {

/\*sort the nodes in the priority queue\*/

std::sort(nodes.begin(), nodes.end());

/\*create a new parent node and add it to the vector\*/

HuffmanTreeNode\* tempParentNode = new HuffmanTreeNode(nodes[0].num + nodes[1].num);

if (tempParentNode == NULL) {

std::cerr << "Error:Memory allocation failed!" << std::endl;

exit(MEMORY\_ALLOCATION\_ERROR);

}

tempParentNode->leftChild = new HuffmanTreeNode(nodes[0]);

tempParentNode->rightChild = new HuffmanTreeNode(nodes[1]);

/\*remove the two nodes from the vector and add the parent node to the vector\*/

nodes.erase(nodes.begin(),nodes.begin()+2);

/\*add the parent node to the vector\*/

nodes.push\_back(\*tempParentNode);

delete tempParentNode;

}

}

/\*

\* Function Name: printHuffmanTree

\* Function: print the Huffman tree in pre-order

\* Input Parameters:HuffmanTreeNode\* node

\* Return Value: void

\*/

void printHuffmanTree(HuffmanTreeNode \*&node) {

if (node != NULL) {

if (node->leftChild != NULL) {

node->leftChild->code = node->code+ '0';

printHuffmanTree(node->leftChild);

}

if (node->rightChild != NULL) {

node->rightChild->code += node->code+'1';

printHuffmanTree(node->rightChild);

}

/\*if the node is a leaf node, print it\*/

if(node->leftChild==NULL&&node->rightChild==NULL) {

std::cout << \*node << std::endl;

}

}

}

/\*

\* Function Name: inputLogicalValue

\* Function: input a logical value from user

\* Input Parameters:char FalseValue = '0'

\* char TrueValue = '1'

\* Return Value: int: 0 for False, 1 for True

\*/

int inputLogicalValue(char FalseValue = '0', char TrueValue = '1')

{

while (true) {

char InputChar = \_getch();

if (InputChar == 0 || InputChar == -32)

InputChar = \_getch();

else if (InputChar == FalseValue || InputChar == TrueValue) {

std::cout << InputChar;

return InputChar == TrueValue;

}

}

}

/\*

\* Function Name: inputInteger

\* Function: input an integer within a range

\* Input Parameters:int minNum

\* int maxNum

\* const char\* prompt

\* Return Value: int

\*/

int inputInteger(int minNum, int maxNum, const char\* prompt = "") {

double num;

while (true) {

if (prompt != "")

std::cout << "请输入" << prompt << "[范围" << minNum << "-" << maxNum << "]:";

std::cin >> num;

if (std::cin.fail() || static\_cast<int>(num) != num || num < minNum || num > maxNum) {

std::cout << "输入非法,请重新输入!" << std::endl;

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

}

else {

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

return static\_cast<int>(num);

}

}

}

/\*

\* Function Name: main

\* Function: main function

\* Return Value: 0

\*/

int main() {

do {

system("cls");

std::cout << "+-----------------------------+" << std::endl;

std::cout << "| 最优二元树 |" << std::endl;

std::cout << "| Optimal Binary Tree |" << std::endl;

std::cout << "+-----------------------------+" << std::endl;

int nodeNum = inputInteger(2, INT\_MAX, ">>> 最优二元树节点个数");

/\*create a vector of HuffmanTreeNode\*/

std::vector<HuffmanTreeNode> nodes(nodeNum,0);

std::cout << std::endl;

std::cout << ">>> 请输入最优二元树的节点" << std::endl;

for (int i = 0; i < nodeNum; i++) {

char temp[CHAR\_MAX];

sprintf(temp, "最优二元树的第 %d 个节点", i + 1);

nodes[i].num = inputInteger(0, INT\_MAX, temp);

}

/\*create the Huffman tree\*/

createHuffmanTree(nodes);

std::cout << std::endl;

/\*print the Huffman tree\*/

std::cout << ">>> 最优二元树编码:" << std::endl;

HuffmanTreeNode\* root = &nodes[0];

printHuffmanTree(root);

std::cout << "是否退出程序?[y/n]:";

} while (!inputLogicalValue('n', 'y'));

return 0;

}