**项目说明文档**

**数据结构课程设计**

**——二叉排序树**

作 者 姓 名 刘淑仪

学 号 2251730

指 导 教 师 张 颖

学 院 专 业 软件学院 软件工程



二〇二三 年 十二 月 十 日

目录

1 项目分析 1

1.1 项目背景分析 1

1.2 项目功能分析 1

2 项目设计 2

2.1 数据结构设计 2

2.2 类结构设计 2

2.3 成员与操作设计 2

2.4 系统设计 6

3 功能设计与项目实现 7

3.1 建立二叉树的实现 7

3.1.1 建立二叉树实现思路 7

3.1.2 建立二叉树核心代码 7

3.1.3 建立二叉树实现示例 9

3.2 插入元素功能的实现 9

3.2.1 插入元素功能实现思路 9

3.2.2 插入元素功能核心代码 9

3.2.3 插入元素功能实现示例 10

3.3 查询元素功能的实现 10

3.3.1 查询元素功能实现思路 10

3.3.2 查询元素功能核心代码 11

3.3.3 查询元素功能实现示例 11

3.4 本项目涉及BinaryTree.h内容的主要实现 12

3.4.1 树结点和二叉树类的定义 12

3.4.2 BinaryTree.h核心代码 12

4 项目测试 12

4.1 功能测试 12

4.1.1 基本功能测试 12

4.1.2构建二叉树测试 13

4.1.3 插入元素测试 14

4.1.4 查询元素测试 15

4.2 错误测试 15

4.2.1 异常情况处理 15

4.3 Linux环境测试 16

5 集成开发环境与编译运行环境 17

# 项目分析

## 项目背景分析

二叉排序树（Binary Sort Tree），也称为二叉搜索树或二叉查找树，是一种特殊的数据结构。在这种树中，每个节点都有最多两个子节点。它按照特定的顺序安排节点：每个节点的左子树只包含小于该节点的值，而右子树只包含大于该节点的值。这种结构使得查找、插入和删除操作可以高效进行。二叉排序树提供了一种灵活、高效的方式来处理动态数据集，特别是在数据经常更改的情况下。

## 项目功能分析

此项目的主要目标是实现一个二叉排序树，并通过这个数据结构执行一系列的操作。

其主要功能的实现主要依赖于

1、二叉排序树的构建：源代码中定义了 TreeNode 类，用于表示树中的每个节点。每个节点包含数据值、指向第一个孩子的指针和指向兄弟的指针。MyBinaryTree 类负责构建和管理二叉树。它使用 TreeNode 类的实例来创建树的结构。

2、插入操作：代码提供了 leftInsertNode 和 rightInsertNode 方法，允许在树的左侧或右侧插入新节点。这是构建二叉排序树的基础操作。

3、查找操作：通过 findNode 方法，可以在树中查找包含特定值的节点。这是基于二叉排序树性质的高效查找方法。

4、遍历操作：代码中实现了 preOrder、inOrder 和 postOrder 方法，分别对应于前序、中序和后序遍历。这些方法用于按特定顺序访问树中的每个节点。通过 levelOrder 方法，可以层次遍历树，这是按照从上到下、从左到右的顺序访问树的每一层。

5.删除和修改操作：deleteChildren 方法用于删除指定节点的所有子节点。modifyNode 方法允许修改树中特定节点的值。

6.二叉树的复制和销毁：通过 copy 方法可以复制整个树的结构和内容。destroy 方法用于安全地销毁树，释放分配的资源。

# 项目设计

## 数据结构设计

从项目功能分析来看，本项目的核心数据结构是基于模板的二叉排序树（Binary Sort Tree），并辅以链式队列来支持树的层次遍历。以下是对这些数据结构设计的详细说明。

项目中的 TreeNode 类是构成二叉树的基本单元。每个节点包含一个数据字段和两个指针，分别指向其第一个孩子和下一个兄弟节点。这种设计使得每个节点可以灵活地表示二叉树的结构，同时简化了节点之间的关系。MyBinaryTree 类作为二叉排序树的主体，包含一个指向根节点的指针。它提供了插入、查找、遍历（前序、中序、后序和层次）、修改、删除等功能。这个类使用泛型设计，可以适用于不同数据类型的树结构。

## 类结构设计

在 TreeNode.h 文件中，定义了 TreeNode 类，用于表示二叉树中的每个节点。每个 TreeNode 对象包含三个私有成员变量：value 存储节点的数据，firstChild 和 nextSibling 分别指向该节点的第一个孩子节点和下一个兄弟节点。构造函数用于初始化节点，重载的输出运算符 << 用于打印节点值，便于调试和展示树结构。

在 MyBinaryTree.h 文件中，定义了 MyBinaryTree 类，用于创建和管理二叉排序树。包含一个指向根节点的指针 root，以及若干私有辅助函数，如 copy 用于复制树，destroy 用于销毁树。提供了构造函数、复制构造函数、析构函数、树的插入、删除、查找、遍历（包括前序、中序、后序和层次遍历）等操作的公有方法。这些方法实现了二叉排序树的核心功能，并提供了用户接口。

## 成员与操作设计

未注释的函数请见下文实现对应功能时代码的函数标签。

BinaryTree.h内存储基本二叉树的实现。

#include <iostream>

using namespace std;

const int defaultSize = 65536;

/\* 操作算法中用到的预定义常量和类型 \*/

// 函数结果状态代码

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASIBLE -1

typedef int DataType;

template<typename Type>

class TreeNode;

class BinarySortTree;

// 定义队列便于进行层次序遍历

// 链表节点定义

template<typename Type>

class QueueNode {

public:

TreeNode<Type>\* data;

QueueNode<Type>\* next;

QueueNode(TreeNode<Type>\* node) : data(node), next(NULL) {}

};

// 链队定义

template<typename Type>

class LinkedQueue {

private:

QueueNode<Type>\* front; // 队列头指针

QueueNode<Type>\* rear; // 队列尾指针

public:

LinkedQueue() : front(NULL), rear(NULL) {}

~LinkedQueue() {

while (!isEmpty()) { dequeue(); }

}

bool isEmpty() const { return front == NULL; }

void enqueue(TreeNode<Type>\* node) {

QueueNode<Type>\* newNode = new QueueNode<Type>(node);

if (rear == NULL) {

front = rear = newNode;

}

else {

rear->next = newNode;

rear = newNode;

}

}

TreeNode<Type>\* dequeue() {

if (isEmpty()) {

return NULL;

}

QueueNode<Type>\* temp = front;

TreeNode<Type>\* node = front->data;

front = front->next;

if (front == NULL) {

rear = NULL;

}

delete temp;

return node;

}

TreeNode<Type>\* getFront() const {

return isEmpty() ? NULL : front->data;

}

};

/\* 构建二叉树 \*/

// 孩子兄弟法定义结点

template<typename Type>

class TreeNode {

public:

Type value;

TreeNode\* firstChild, \* nextSibling;

TreeNode() : firstChild(NULL), nextSibling(NULL) {}

TreeNode(Type item, TreeNode<Type>\* l = NULL, TreeNode<Type>\* r = NULL) : value(item), firstChild(l), nextSibling(r) {}

template<typename T>

friend std::ostream& operator<<(std::ostream& os, const TreeNode<T>& node);

};

// 重载 << 运算符

template<typename Type>

std::ostream& operator<<(std::ostream& os, const TreeNode<Type>& node) {

os << node.value;

return os;

}

// 定义二叉树

template<typename Type>

class MyBinaryTree {

protected:

TreeNode<Type>\* root; // 根节点

TreeNode<Type>\* copy(const TreeNode<Type>\* subTree);

void outputNode(TreeNode<Type>\* node) { if (node != NULL) std::cout << node->value << "->"; }

bool containsRecursive(TreeNode<Type>\* node, const Type& data) {

if (node == nullptr) { return false; }

if (node->value == data) { return true; }

return containsRecursive(node->firstChild, data) || containsRecursive(node->nextSibling, data);

}

public:

MyBinaryTree() : root(NULL) {} // 构造函数

MyBinaryTree(Type& item); // 构造函数

MyBinaryTree(MyBinaryTree<Type>& other) { root = copy(other.root); } // 复制构造函数

~MyBinaryTree() { destroy(root); } // 析构函数

void setRoot(TreeNode<Type>\* newRoot) { root = newRoot; }

void destroy(TreeNode<Type>\*& subTree);

bool isEmpty(void) { return root == NULL; } // 判断二叉树是否为空

// 得到二叉树的高度

int getHeight(TreeNode<Type>\* subTree) { return (subTree == NULL) ? 0 : (std::max(getHeight(subTree->firstChild), getHeight(subTree->nextSibling)) + 1); }

// 得到二叉树的结点数

int getSize(TreeNode<Type>\* subTree) { return (subTree == NULL) ? 0 : (getSize(subTree->firstChild) + getSize(subTree->nextSibling) + 1); }

TreeNode<Type>\* getRoot(void) const { return root; }

TreeNode<Type>\* getParent(TreeNode<Type>\* current, TreeNode<Type>\* subTree);

TreeNode<Type>\* getLeftChild(TreeNode<Type>\* current) { return current == NULL ? NULL : current->firstChild; }

TreeNode<Type>\* getRightChild(TreeNode<Type>\* current) { return current == NULL ? NULL : current->nextSibling; }

TreeNode<Type>\* findNode(const Type& item, TreeNode<Type>\* subTree); // 获取指定值的节点

// 遍历二叉树

void preOrder(TreeNode<Type>\* subTree); // 前序遍历

void inOrder(TreeNode<Type>\* subTree); // 中序遍历

void postOrder(TreeNode<Type>\* subTree); // 后序遍历

void levelOrder(TreeNode<Type>\* subTree); // 层次序遍历

// 插入二叉树

bool leftInsertNode(TreeNode<Type>\* current, const Type& item);

bool rightInsertNode(TreeNode<Type>\* current, const Type& item);

bool modifyNode(const Type& oldItem, const Type& newItem, TreeNode<Type>\* subTree);

bool insertAsFirstChild(TreeNode<Type>\* current, const Type& item);

void deleteChildren(TreeNode<Type>\* node);

MyBinaryTree<Type>& operator=(const MyBinaryTree<Type>& other);

};

binary sort tree.cpp内存储该问题的对应解决方案类：

#include <iostream>

#include "BinaryTree.h"

#define MAX\_SIZE 100

typedef class BinarySortTree:public MyBinaryTree<int>{

private:

TreeNode<int>\* insertRecursive(TreeNode<int>\* node, int value);

public:

void menu();

void creatTree();

void addNode();

void searchNode();

void insert(int value){ root = insertRecursive(root, value); }

bool operation();

int inputNum();

void Exit() {

std::cout << "按任意键继续...\n";

// 清除输入缓冲区

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

// 等待用户按键

std::cin.get();

}

}BST;

## 系统设计

本项目的代码结构被细分为多个模块，每个模块负责处理特定的功能，确保了代码的清晰性和模块化。具体来说，如 TreeNode 和 MyBinaryTree 负责二叉树的基础实现，而 LinkedQueue 负责辅助实现层次遍历。通过这种方式，二叉排序树的结构和操作被有效地模拟和管理。

在这个系统中，二叉排序树用于有效地存储和管理数据，这对于需要快速查找、插入和删除操作的应用场景特别合适。这种数据结构尤其适合处理排序和搜索任务，因为它保持了数据的有序性，同时也提供了高效的查找和遍历方法。

系统中的数据和方法被封装在各个类中，减少了全局变量的使用，这不仅提高了代码的可读性和可维护性，也增强了数据安全性。例如，TreeNode 类封装了树节点的数据和链接，MyBinaryTree 类则封装了树的基本操作，如插入、删除和遍历等。此外，LinkedQueue 类提供了一种高效的方式来支持树的层次遍历。

# 功能设计与项目实现

## 建立二叉树的实现

### 建立二叉树实现思路

insertRecursive 是一个递归函数，它接收两个参数：一个 TreeNode<int>\* node（表示当前考虑的树节点）和一个要插入的整数值 int value。如果当前节点 node 是 NULL，这意味着已经到达了应该插入新节点的位置。因此，函数创建一个新的 TreeNode<int> 节点，存储传入的 value，并返回这个新节点。

如果当前节点不是 NULL，则比较 value 和当前节点的值 node->value：如果 value 小于 node->value，则应该在左子树中继续寻找插入位置。因此，函数递归地调用 insertRecursive，传入左子树 node->firstChild 和 value 作为参数，并将返回的节点作为当前节点的新左子树。如果 value 大于或等于 node->value，则应该在右子树中继续寻找插入位置。相应地，函数递归地调用 insertRecursive，传入右子树 node->nextSibling 和 value，并将返回的节点作为当前节点的新右子树。

在每次递归调用之后，函数返回当前考虑的节点 node。这确保了树的结构在插入新节点后保持不变，并允许递归调用链正确地更新父节点的子节点链接。

### 建立二叉树核心代码

// 构建二叉排序树

void BinarySortTree::creatTree()

{

int size[MAX\_SIZE] = { 0 };

int nodeNum = 0;

std::cout << ">>>请输入树的总结点个数：";

while (1) {

std::cin >> nodeNum;

if (std::cin.fail() || nodeNum <= 0 || nodeNum > MAX\_SIZE) {

std::cin.clear();

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

std::cout << "输入数据有误，请重新输入\n";

}

else {

if (std::cin.get() == '\n')

break;

else {

std::cin.clear();

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

std::cout << "输入数据有误，请重新输入\n";

}

}

}

// 输入键值

std::cout << ">>>请输入键值以创建二叉排序树[超过总结点数不予存储]：\n";

for (int i = 0; i < nodeNum; ++i) { cin >> size[i]; }

// 查重

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

for (int j = i + 1; j < nodeNum; ++j) {

if (size[i] == size[j]) {

size[i] = 0;

std::cout << "键值" << size[j] << "重复！\n";

break;

}

}

}

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

if (size[i] != 0)

insert(size[i]);

}

inOrder(root);

std::cout << std::endl;

}

// 二叉树排序建立

TreeNode<int>\* BinarySortTree::insertRecursive(TreeNode<int>\* node, int value)

{

if (node == NULL) { return new TreeNode<int>(value);}

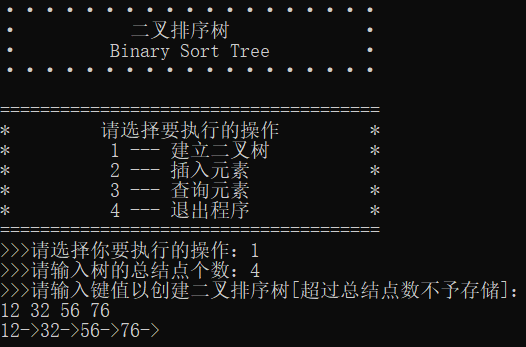
if (value < node->value) { node->firstChild = insertRecursive(node->firstChild, value); }

else { node->nextSibling = insertRecursive(node->nextSibling, value);}

return node;

}

### 建立二叉树实现示例



## 插入元素功能的实现

### 插入元素功能实现思路

首先，用户输入需要插入的键值。使用一个自定义函数 inputNum() 来读取用户的输入。这个函数用于确保输入是有效整数的工具函数。获得用户输入的新元素后，函数调用 insert(newElem)，这个方法是 BinarySortTree 类中的一个方法，用于将新元素插入到二叉排序树中。这个过程会根据二叉排序树的规则（小于节点值的元素在左子树，大于等于节点值的元素在右子树）找到合适的位置并插入新元素。

插入新元素后，函数通过调用 inOrder(root) 方法来进行树的中序遍历。这里的 root 是二叉排序树的根节点，表明遍历从树的根节点开始。

### 插入元素功能核心代码

// 增加元素

void BinarySortTree::addNode()

{

int newElem;

bool isDuplicate = false;

do {

// 输入新增的键值

std::cout << "\n>>>请输入需要插入的键值：\n";

newElem = inputNum();

// 检查新键值是否与树中的键值重复

if (findNode(newElem,root) != nullptr) {

std::cout << "键值 " << newElem << " 已存在，请重新输入。\n";

isDuplicate = true;

}

else {

insert(newElem);

inOrder(root);

std::cout << std::endl;

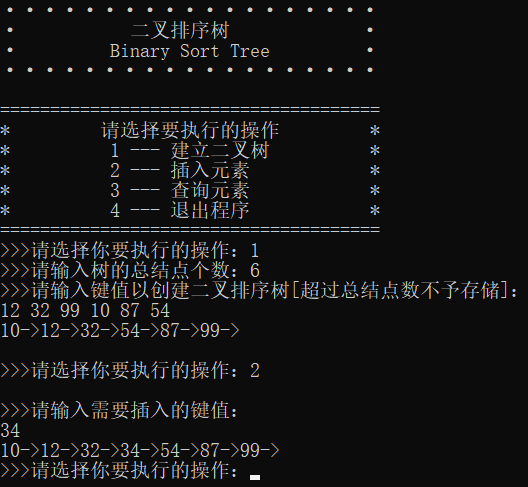
isDuplicate = false;

}

} while (isDuplicate);

}

### 插入元素功能实现示例



## 查询元素功能的实现

### 3.3.1 查询元素功能实现思路

函数提示用户输入想要查询的键值。使用 inputNum() 函数获取用户输入的键值。这个函数设计为读取并验证用户输入，确保它是一个有效的整数。函数调用 findNode(searchElem, root) 方法来在二叉排序树中查找用户输入的键值。这里 searchElem 是用户输入的键值，root 是二叉排序树的根节点。

findNode根据二叉排序树的特性在树中搜索指定的键值。查询结果通过判断 findNode 函数的返回值来确定。如果返回值不是 NULL，意味着找到了对应的元素。如果 findNode 返回 NULL，则说明树中不存在该元素，函数相应地输出错误提示信息。

### 3.3.2 查询元素功能核心代码

// 查询元素

void BinarySortTree::searchNode()

{

int searchElem;

std::cout << "\n>>>请输入需要查询的键值：\n";

searchElem = inputNum();

if (findNode(searchElem, root) != NULL)

std::cout << "查询成功，该元素在二叉排序树中！\n\n";

else

std::cout << "查询失败，该元素不存在！\n\n";

}

### 3.3.3 查询元素功能实现示例



## 本项目涉及BinaryTree.h内容的主要实现

### 3.4.1 树结点和二叉树类的定义

定义了二叉树结点TreeNode，每个节点包含值（value），指向第一个孩子（firstChild）和下一个兄弟（nextSibling）的指针。定义了LinkedQueue作为用于层次遍历二叉树的链队列，包含队列的头（front）和尾（rear）指针，以及相关的队列操作函数（如 enqueue、dequeue）。定义了二叉树的主体MyBinaryTree，表示二叉树的主体，包含根节点指针（root）以及一系列操作二叉树的函数，如 copy、destroy、isEmpty、getHeight、getSize 等。提供树的遍历方法，包括前序、中序、后序和层次序遍历。包含节点的查找、插入和删除操作。

### 3.4.2 BinaryTree.h核心代码

详见类设计与数据结构设计。

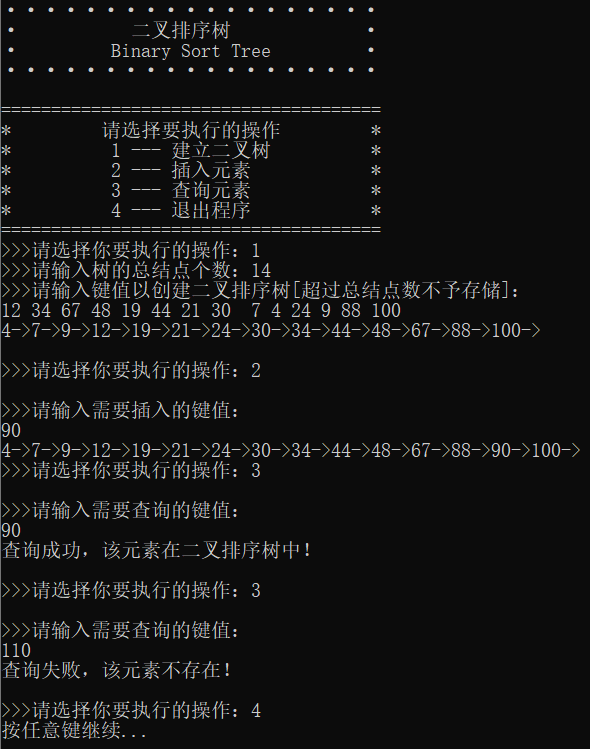
# 项目测试

## 功能测试

### 基本功能测试

**测试用例：**正常构建二叉排序树，也可以正常插入元素和查询元素，最后整理输出

**实验结果：**

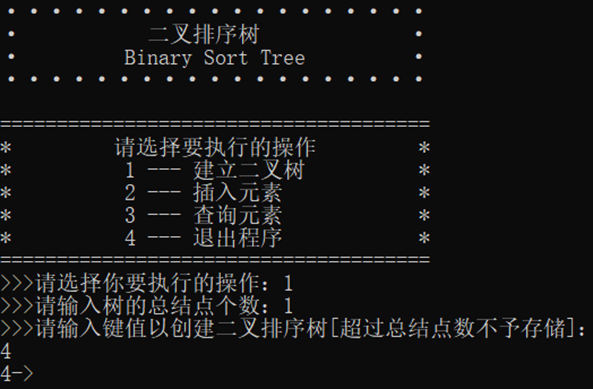


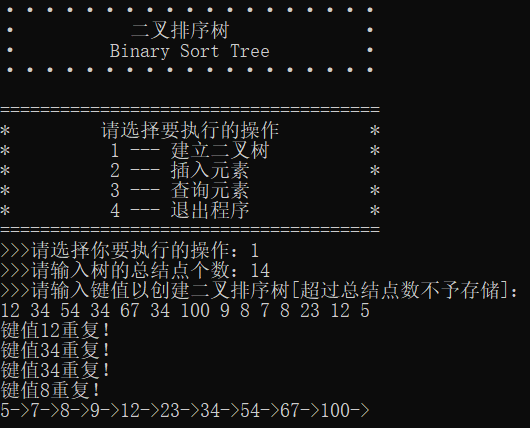
### 4.1.2构建二叉树测试

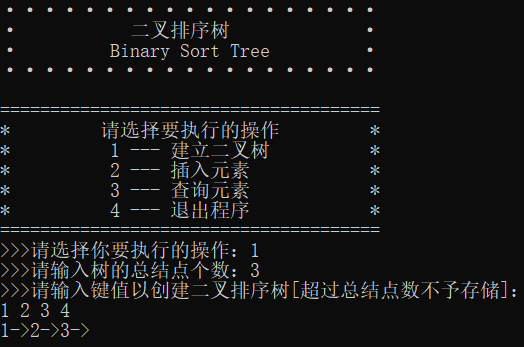
**测试用例：**构建二叉树时插入重复的元素/超过输入的结点个数/只有一个结点

**预期结果：**程序对重复的数字报错，正常构建不含重复元素的二叉树。

**实验结果：**

****



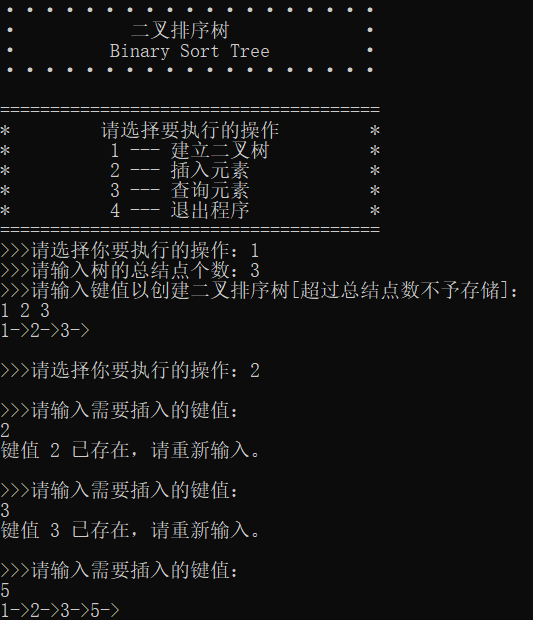


### 4.1.3 插入元素测试

**测试用例：**测试能否正确插入元素/对插入的元素查重

**预期结果：**程序对重复的数字报错，正常插入不重复的元素。

**实验结果：**

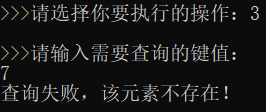
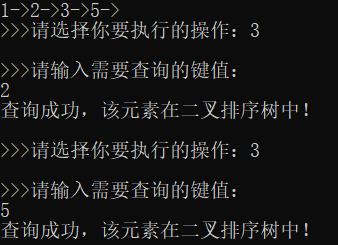


### 4.1.4 查询元素测试

**测试用例：**查询不存在的元素/查询新插入的元素

**预期结果：**程序对重复的数字报错，正常插入不重复的元素。

**实验结果：**



## 错误测试

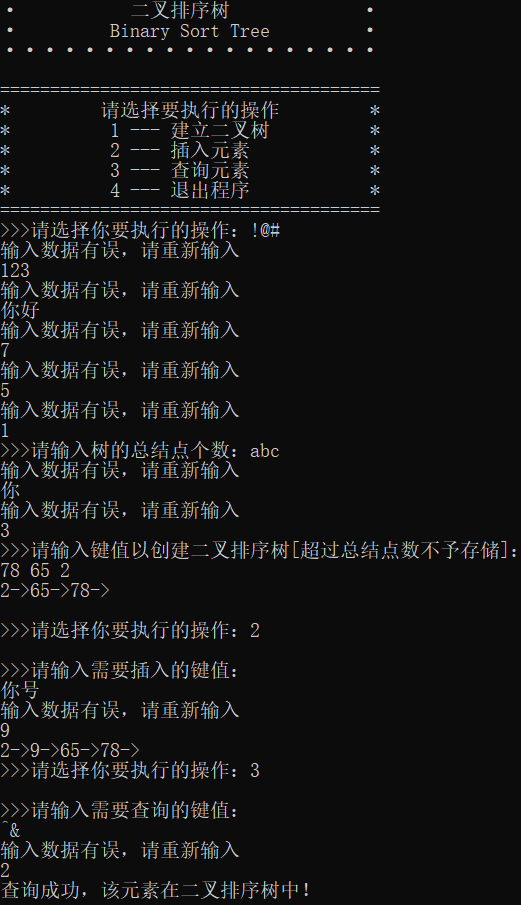
### 异常情况处理

**测试用例：**abc/ @#￥ / 五

**用例说明：**测试非法输入（如字母）

**预期结果：**程序给出错误信息，程序正常运行不崩溃。

**实验结果：**

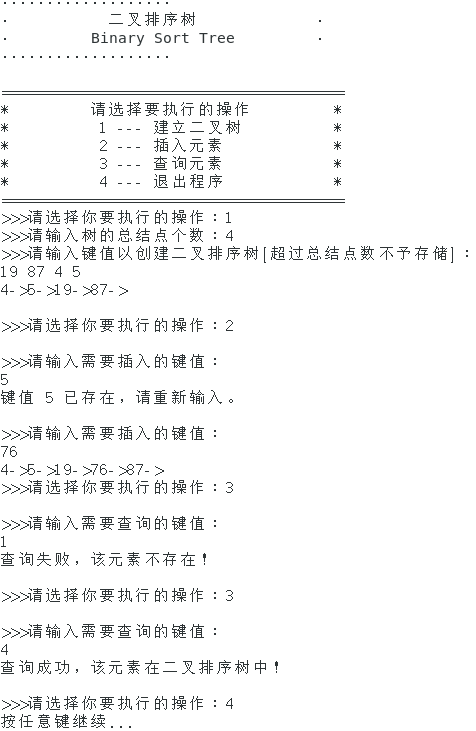


## Linux环境测试

**编译命令：** g++ -std=c++11 '/mnt/hgfs/ShareFolder/ 09\_binary\_sort\_tree' -o nineExample

**运行命令：** ./nineExample

**实验结果：**



# 集成开发环境与编译运行环境

Windows系统：Windows 11 x64

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

Windows编译运行环境：本项目适用于x86架构和x64架构

Linux系统：CentOS 7 x64