

项目说明文档

数据结构课程设计

——二叉排序树

作者姓名 _____ 林继申

学 号 _____ 2250758

指导教师 _____ 张 颖

学院专业 _____ 软件学院 软件工程



同濟大學
TONGJI UNIVERSITY

二〇二三年十二月十三日

目录

| | |
|--|---|
| 1 项目分析..... | 1 |
| 1.1 项目背景分析..... | 1 |
| 1.2 项目需求分析..... | 1 |
| 1.3 项目功能分析..... | 1 |
| 1.3.1 插入元素功能..... | 1 |
| 1.3.2 查询元素功能..... | 2 |
| 1.3.3 异常处理功能..... | 2 |
| 2 项目设计..... | 2 |
| 2.1 数据结构设计..... | 2 |
| 2.2 结构体与类设计..... | 3 |
| 2.2.1 MyLinkNode 结构体的设计..... | 3 |
| 2.2.1.1 概述..... | 3 |
| 2.2.1.2 结构体定义..... | 3 |
| 2.2.1.3 数据成员..... | 3 |
| 2.2.1.4 构造函数..... | 3 |
| 2.2.2 MyQueue 类的设计..... | 3 |
| 2.2.2.1 概述..... | 3 |
| 2.2.2.2 类定义..... | 4 |
| 2.2.2.3 私有数据成员..... | 4 |
| 2.2.2.4 构造函数..... | 4 |
| 2.2.2.5 析构函数..... | 4 |
| 2.2.2.6 公有成员函数..... | 4 |
| 2.2.3 MyBinTreeNode 结构体的设计..... | 5 |
| 2.2.3.1 概述..... | 5 |
| 2.2.3.2 结构体定义..... | 5 |
| 2.2.3.3 数据成员..... | 5 |
| 2.2.3.4 构造函数..... | 5 |
| 2.2.4 MyBinaryTree 类的设计..... | 6 |
| 2.2.4.1 概述..... | 6 |
| 2.2.4.2 类定义..... | 6 |
| 2.2.4.3 受保护的数据成员..... | 8 |
| 2.2.4.4 VisitFunction 类型定义..... | 8 |
| 2.2.4.5 构造函数..... | 8 |

| | |
|---|----|
| 2.2.4.6 析构函数 | 8 |
| 2.2.4.7 私有成员函数 | 8 |
| 2.2.4.8 公有成员函数 | 9 |
| 2.2.4.9 运算符重载 | 10 |
| 2.2.5 BinarySortTree 类的设计 | 11 |
| 2.2.5.1 概述 | 11 |
| 2.2.5.2 类定义 | 11 |
| 2.2.5.3 私有成员函数 | 11 |
| 2.2.5.4 公有成员函数 | 11 |
| 2.3 项目主体架构设计 | 12 |
| 3 项目功能实现..... | 13 |
| 3.1 项目主体架构的实现 | 13 |
| 3.1.1 项目主体架构实现思路 | 13 |
| 3.1.2 项目主体架构核心代码 | 14 |
| 3.1.3 项目主体架构示例 | 15 |
| 3.2 插入元素功能的实现 | 16 |
| 3.2.1 插入元素功能实现思路 | 16 |
| 3.2.2 插入元素功能核心代码 | 17 |
| 3.2.3 插入元素功能示例 | 17 |
| 3.3 查询元素功能的实现 | 18 |
| 3.3.1 查询元素功能实现思路 | 18 |
| 3.3.2 查询元素功能核心代码 | 18 |
| 3.3.3 查询元素功能示例 | 19 |
| 3.4 异常处理功能的实现 | 19 |
| 3.4.1 动态内存申请失败的异常处理 | 19 |
| 3.4.2 MyQueue 类队列空的异常处理 | 20 |
| 3.4.3 MyBinaryTree 类自赋值的异常处理 | 20 |
| 3.4.4 输入非法的异常处理 | 20 |
| 3.4.4.1 二叉排序树元素个数和元素的值输入非法的异常处理 | 20 |
| 3.4.4.2 操作类型输入非法的异常处理 | 21 |
| 4 项目测试..... | 22 |
| 4.1 输入二叉排序树元素个数功能测试 | 22 |
| 4.2 建立二叉排序树功能测试 | 23 |
| 4.3 插入元素功能测试 | 24 |
| 4.4 查询元素功能测试 | 25 |

| | |
|--------------------|----|
| 5 集成开发与编译运行环境..... | 25 |
|--------------------|----|

1 项目分析

1.1 项目背景分析

在计算机科学中，数据结构的选择和实现对于算法的效率至关重要。二叉排序树（Binary Sort Tree，简称 BST）是一种常见的数据结构，用于存储数据以便快速检索和插入操作。二叉排序树具有以下特性：每个节点的左子树仅包含小于该节点的值，而每个节点的右子树仅包含大于该节点的值。二叉排序树的查找过程依赖于其独特的结构，其中每个节点都按照特定的顺序排列。当开始查找时，首先将要查找的关键字与树的根节点进行比较。如果二者相等，则查找立即成功。如果待查找的关键字小于根节点的值，则继续在左子树中搜索；反之，如果大于根节点的值，则在右子树中继续查找。这种方法能有效缩减查找范围并减少比较次数，从而提升查找效率。

1.2 项目需求分析

基于以上背景分析，本项目需要实现需求如下：

- (1) 实现一个二叉排序树，允许用户插入新的数据项到树中，并支持快速查找树中是否存在特定的数据项；
- (2) 支持多种树遍历方式，如前序遍历、中序遍历、后序遍历和层序遍历；
- (3) 实现异常处理机制，确保系统稳定性和安全性，避免因用户输入错误导致系统崩溃或信息丢失。

1.3 项目功能分析

本项目旨在通过实现一个二叉排序树，允许用户插入新的数据项到树中，并支持快速查找树中是否存在特定的数据项。下面对项目的功能进行详细分析。

1.3.1 插入元素功能

允许用户将新的数据项加入到二叉排序树中。插入操作从树的根节点开始，依据二叉排序树的特性，逐层比较并确定新节点的正确位置。

为维持树的独特性，如果插入的值在树中已存在，则不会重复插入。

在平衡的二叉排序树中，插入操作的平均时间复杂度为 $O(\log n)$ 。

1.3.2 查询元素功能

用户可以查询特定的数据项是否存在于树中。从根节点开始，根据二叉排序树的性质，逐步向下查找，直到找到相应的节点或确定该值不存在于树中。

由于二叉排序树的结构，查找操作能够迅速缩小搜索范围，提升查找效率。在最佳情况下，查找操作的时间复杂度同样为 $O(\log n)$ 。

1.3.3 异常处理功能

实现异常处理机制，处理用户可能输入的非法信息，确保系统的稳定性和安全性。

2 项目设计

2.1 数据结构设计

二叉排序树(Binary Sort Tree, 简称BST)是一种特别的二叉树数据结构，二叉排序树的特点如下：

(1) 节点的有序性

每个节点包含一个键（通常是一种数值）和两个指向其子节点的指针。对于树中的每个节点，其左子树中的所有节点的键都小于该节点的键，而右子树中的所有节点的键都大于该节点的键；

(2) 高效的查找、插入和删除操作

由于其有序性，二叉排序树能够提供对数时间复杂度 ($O(\log n)$) 的平均性能，用于查找、插入和删除操作。这使得它特别适合于实现动态查找表；

(3) 中序遍历的有序性

对二叉排序树进行中序遍历将按照键的升序访问树中的每个节点，这是由于其“左-根-右”的访问顺序恰好符合其左小右大的性质；

(4) 动态数据集处理

二叉排序树非常适合于频繁变化的数据集，因为它允许在不重新构建整个数据结构的情况下插入和删除节点；

(5) 内存使用

由于其结构，二叉排序树不需要为表示树结构额外消耗大量内存，每个节点仅存储其键和两个指针。

这种结构使得二叉排序树在处理有序数据和提供快速查询方面非常高效，尤其适用于那些需要频繁插入和删除操作的场景。

2.2 结构体与类设计

2.2.1 MyLinkNode 结构体的设计

2.2.1.1 概述

MyLinkNode 结构体是一个用于构建链表节点的模板结构体。该结构体用于表示链表中的每个节点，其中包括节点存储的数据以及指向下一个节点的指针。经典的链表一般包括两个抽象数据类型（ADT）——链表结点类（**LNode**）与链表类（**LinkList**）。本项目希望链表结点类可以直接访问链表结点，所以使用 **struct** 而不是 **class** 描述链表结点类。

2.2.1.2 结构体定义

```
template <typename Type>
struct MyLinkNode {
    Type data;
    MyLinkNode<Type>* link;
    MyLinkNode(MyLinkNode<Type>* ptr = NULL) { link = ptr; }
    MyLinkNode(const Type& item, MyLinkNode<Type>* ptr = NULL)
{ data = item; link = ptr; }
};
```

2.2.1.3 数据成员

Type data: 数据域，存储节点的数据

MyLinkNode<Type>* link: 指针域，指向下一个节点的指针

2.2.1.4 构造函数

MyLinkNode(MyLinkNode<Type>* ptr = NULL);

构造函数，初始化指针域。

MyLinkNode(const Type& item, MyLinkNode<Type>* ptr = NULL);

构造函数，初始化数据域和指针域。

2.2.2 MyQueue 类的设计

2.2.2.1 概述

该通用模板类 **MyQueue** 用于存储和管理数据元素，遵循先进先出（**First-in-First-Out, FIFO**）的原则。该队列是基于链表数据结构实现的，链表节点由 **MyLinkNode** 结构体表示，其中包含数据和指针域。此模板类 **MyQueue** 允许在队列的前端（**front**）添加元素，以及从队列的前端（**front**）移除元素。

2.2.2.2 类定义

```
template <typename Type>
class MyQueue {
private:
    MyLinkNode<Type>* front;
    MyLinkNode<Type>* rear;
public:
    MyQueue() : front(NULL), rear(NULL) {}
    ~MyQueue() { makeEmpty(); }
    bool isEmpty(void) const { return front == NULL; }
    void makeEmpty(void);
    void enqueue(const Type& item);
    bool dequeue(Type& item);
    bool getFront(Type& item) const;
    int getSize(void) const;
};
```

2.2.2.3 私有数据成员

MyLinkNode<Type>* front: 指针指向队列的前端，即队列中的第一个元素

MyLinkNode<Type>* rear: 指针指向队列的后端，即队列中的最后一个元素

2.2.2.4 构造函数

```
MyQueue() : front(NULL), rear(NULL) {}
```

默认构造函数，创建一个空的队列。

2.2.2.5 析构函数

```
~MyQueue() { makeEmpty(); }
```

析构函数，清空队列并释放内存。

2.2.2.6 公有成员函数

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

检查队列是否为空。

```
void makeEmpty(void);
```

清空队列，释放所有元素占用的内存。


```
void enqueue(const Type& item);
```

将元素添加到队列的末尾。

```
bool dequeue(Type& item);
```

移除队列的前端元素并将其值通过引用返回。

```
bool getFront(Type& item) const;
```

获取队列的前端元素的值。

```
int getSize(void) const;
```

获取队列中元素的数量。

2.2.3 MyBinTreeNode 结构体的设计

2.2.3.1 概述

MyBinTreeNode 结构体是一个模板结构体，用于构建和表示一个二叉树的节点。二叉树是一种重要的数据结构，广泛应用于各种计算机科学领域。在这个结构体中，每个节点包含一个数据元素和两个指向其子节点的指针：左子节点和右子节点。

2.2.3.2 结构体定义

```
template <typename Type>
struct MyBinTreeNode {
    Type data;
    MyBinTreeNode<Type>* leftChild;
    MyBinTreeNode<Type>* rightChild;
    MyBinTreeNode() : leftChild(NULL), rightChild(NULL) {}
    MyBinTreeNode(Type item, MyBinTreeNode<Type>* l = NULL,
MyBinTreeNode<Type>* r = NULL) : data(item), leftChild(l),
rightChild(r) {}
};
```

2.2.3.3 数据成员

Type data: 数据域，存储节点的数据

MyBinTreeNode<Type>* leftChild: 指针域，指向左子节点的指针

MyBinTreeNode<Type>* rightChild: 指针域，指向右子节点的指针

2.2.3.4 构造函数

```
MyBinTreeNode() : leftChild(NULL), rightChild(NULL) {}
```

构造函数，初始化指针域。

```

    MyBinTreeNode(Type item, MyBinTreeNode<Type>* l = NULL,
MyBinTreeNode<Type>* r = NULL) : data(item), leftChild(l),
rightChild(r) {}

```

构造函数，初始化数据域和指针域。

2.2.4 MyBinaryTree 类的设计

2.2.4.1 概述

MyBinaryTree 类是一个模板类，用于创建和管理二叉树结构。二叉树是一种基础且重要的数据结构，广泛应用于多种场景，如管理、排序和搜索算法等。本模板类提供了二叉树的基本操作，包括插入、查找、遍历、修改和删除节点等功能。

2.2.4.2 类定义

```

template <typename Type>
class MyBinaryTree {
protected:
    MyBinTreeNode<Type>* root;
private:
    typedef void (MyBinaryTree::*
VisitFunction)(MyBinTreeNode<Type>*);
    MyBinTreeNode<Type>* copy(const MyBinTreeNode<Type>*
subTree);
    void outputNode(MyBinTreeNode<Type>* node) { if (node !=
NULL) std::cout << node->data << " -> "; }
public:
    MyBinaryTree() : root(NULL) {}
    MyBinaryTree(Type& item);
    MyBinaryTree(MyBinaryTree<Type>& other) { root =
copy(other.root); }
    ~MyBinaryTree() { destroy(root); }
    void destroy(MyBinTreeNode<Type>*& subTree);
    bool isEmpty(void) { return root == NULL; }
    int getHeight(MyBinTreeNode<Type>* subTree) { return
(subTree == NULL) ? 0 : (std::max(getHeight(subTree->leftChild),
getHeight(subTree->rightChild)) + 1); }

```

```

    int  getSize(MyBinTreeNode<Type>*  subTree)  {  return
(subTree ==  NULL)  ?  0  :  (getSize(subTree->leftChild)  +
getSize(subTree->rightChild)  +  1);  }

    MyBinTreeNode<Type>*  getRoot(void)  {  return  root;  }

    MyBinTreeNode<Type>*      getParent(MyBinTreeNode<Type>*
current, MyBinTreeNode<Type>*  subTree);

    MyBinTreeNode<Type>*  getLeftChild(MyBinTreeNode<Type>*
current)  {  return  current ==  NULL  ?  NULL  :  current->leftChild;  }

    MyBinTreeNode<Type>*  getRightChild(MyBinTreeNode<Type>*
current)  {  return  current ==  NULL  ?  NULL  :  current->rightChild;  }

    MyBinTreeNode<Type>*  findNode(const  Type&  item,
MyBinTreeNode<Type>*  subTree);

    void  preOrder(VisitFunction  visit,  MyBinTreeNode<Type>*
subTree);

    void  inOrder(VisitFunction  visit,  MyBinTreeNode<Type>*
subTree);

    void  postOrder(VisitFunction  visit,  MyBinTreeNode<Type>*
subTree);

    void          levelOrder(VisitFunction          visit,
MyBinTreeNode<Type>*  subTree);

    void  preOrderOutput(MyBinTreeNode<Type>*  subTree)
{  preOrder(&MyBinaryTree<Type>::outputNode,  subTree);  }

    void  inOrderOutput(MyBinTreeNode<Type>*  subTree)
{  inOrder(&MyBinaryTree<Type>::outputNode,  subTree);  }

    void  postOrderOutput(MyBinTreeNode<Type>*  subTree)
{  postOrder(&MyBinaryTree<Type>::outputNode,  subTree);  }

    void  levelOrderOutput(MyBinTreeNode<Type>*  subTree)
{  levelOrder(&MyBinaryTree<Type>::outputNode,  subTree);  }

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

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

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

    MyBinaryTree<Type>&  operator=(const  MyBinaryTree<Type>&

```

```
other);  
};
```

2.2.4.3 受保护的数据成员

`MyBinTreeNode<Type>* root`: 指向树的根节点的指针

2.2.4.4 VisitFunction 类型定义

```
typedef void (MyBinaryTree::* VisitFunction)(MyBinTreeNode<Type>*);
```

在 `MyBinaryTree` 类中, `VisitFunction` 是一个特殊的类型定义, 属于类定义的一部分。这是一个函数指针类型, 用于指向 `MyBinaryTree` 类的成员函数, 这些成员函数接受一个指向 `MyBinTreeNode<Type>` 类型的指针作为参数并返回 `void`。

`VisitFunction` 类型主要用于树的遍历操作中, 允许将成员函数作为参数传递给遍历函数, 从而实现对树中每个节点执行特定操作的功能。通过这种方式, 可以在遍历树时灵活地应用不同的处理逻辑, 例如打印节点数据、计算节点总数、修改节点数据等。

2.2.4.5 构造函数

```
MyBinaryTree() : root(NULL) {}
```

默认构造函数, 初始化一个空的二叉树, 即根节点设置为 `NULL`。

```
MyBinaryTree(Type& item);
```

转换构造函数, 创建一个二叉树, 并设置根节点的数据为提供的 `item`, 适用于已知根节点数据的情况, 便于直接构建含有根节点的二叉树。

```
MyBinaryTree(MyBinaryTree<Type>& other) { root =  
copy(other.root); }
```

复制构造函数, 创建一个新的二叉树, 其结构和数据是另一个二叉树 (`other`) 的深拷贝。

2.2.4.6 析构函数

```
~MyBinaryTree() { destroy(root); }
```

析构函数, 用于在 `MyBinaryTree` 类的对象不再需要时, 安全地销毁该对象。它负责释放二叉树中所有节点占用的内存资源, 防止内存泄漏。

2.2.4.7 私有成员函数

```
MyBinTreeNode<Type>* copy(const MyBinTreeNode<Type>*  
subTree);
```

深拷贝一个子树。

```
void outputNode(MyBinTreeNode<Type>* node) { if (node != NULL)
std::cout << node->data << " -> "; }
```

输出一个节点的数据。

2.2.4.8 公有成员函数

```
void destroy(MyBinTreeNode<Type>*& subTree);
```

递归地销毁整个子树。

```
bool isEmpty(void) { return root == NULL; }
```

检查树是否为空。

```
int getHeight(MyBinTreeNode<Type>* subTree) { return
(subTree == NULL) ? 0 : (std::max(getHeight(subTree->leftChild),
getHeight(subTree->rightChild)) + 1); }
```

计算树或子树的高度。

```
int getSize(MyBinTreeNode<Type>* subTree) { return (subTree
== NULL) ? 0 : (getSize(subTree->leftChild) +
getSize(subTree->rightChild) + 1); }
```

计算树或子树的节点总数。

```
MyBinTreeNode<Type>* getRoot(void) { return root; }
```

获取树的根节点。

```
MyBinTreeNode<Type>* getParent(MyBinTreeNode<Type>* current,
MyBinTreeNode<Type>* subTree);
```

查找指定节点的父节点。

```
MyBinTreeNode<Type>* getLeftChild(MyBinTreeNode<Type>*
current) { return current == NULL ? NULL : current->leftChild; }
```

获取指定节点的左子节点。

```
MyBinTreeNode<Type>* getRightChild(MyBinTreeNode<Type>*
current) { return current == NULL ? NULL : current->rightChild; }
```

获取指定节点的右子节点。

```
MyBinTreeNode<Type>* findNode(const Type& item,
MyBinTreeNode<Type>* subTree);
```

在树中查找包含特定数据的节点。

```
void preOrder(VisitFunction visit, MyBinTreeNode<Type>*
subTree);
```

前序遍历（Pre-Order Traversal）树。先访问根节点，然后递归地进行左子树的前序遍历，最后递归地进行右子树的前序遍历。

```
void inOrder(VisitFunction visit, MyBinTreeNode<Type>*
```

```
subTree);
```

中序遍历(In-Order Traversal)树。首先递归地进行左子树的中序遍历，然后访问根节点，最后递归地进行右子树的中序遍历。

```
void postOrder(VisitFunction visit, MyBinTreeNode<Type>*
subTree);
```

后序遍历(Post-Order Traversal)树。首先递归地进行左子树的后序遍历，然后递归地进行右子树的后序遍历，最后访问根节点。

```
void levelOrder(VisitFunction visit, MyBinTreeNode<Type>*
subTree);
```

层序遍历(Level-Order Traversal)树。按照树的层次从上到下，从左到右逐层访问每个节点。通常使用队列来实现。将根节点入队，然后在循环中不断出队一个节点并将其子节点入队，直到队列为空。

```
void preOrderOutput(MyBinTreeNode<Type>* subTree)
{ preOrder(&MyBinaryTree<Type>::outputNode, subTree); }
```

前序遍历树输出。

```
void inOrderOutput(MyBinTreeNode<Type>* subTree)
{ inOrder(&MyBinaryTree<Type>::outputNode, subTree); }
```

中序遍历树输出。

```
void postOrderOutput(MyBinTreeNode<Type>* subTree)
{ postOrder(&MyBinaryTree<Type>::outputNode, subTree); }
```

后续遍历树输出。

```
void levelOrderOutput(MyBinTreeNode<Type>* subTree)
{ levelOrder(&MyBinaryTree<Type>::outputNode, subTree); }
```

层数遍历树输出。

```
bool leftInsertNode(MyBinTreeNode<Type>* current, const
Type& item);
```

在指定节点插入左子节点。

```
bool rightInsertNode(MyBinTreeNode<Type>* current, const
Type& item);
```

在指定节点插入右子节点。

```
bool modifyNode(const Type& oldItem, const Type& newItem,
MyBinTreeNode<Type>* subTree);
```

修改树中特定数据的节点。

2.2.4.9 运算符重载

```
MyBinaryTree<Type>& operator=(const MyBinaryTree<Type>&
```

other);

重载赋值运算符，用于实现树的深拷贝。

2.2.5 BinarySortTree 类的设计

2.2.5.1 概述

BinarySortTree 类是一个基于二叉排序树原理设计的类，继承自 **MyBinaryTree<Type>** 类。这个类的核心目标是实现一个高效的数据存储和访问机制，特别适用于需要频繁插入、删除和搜索操作的场景。通过保持每个节点的左子树只包含小于节点值的元素，而右子树只包含大于节点值的元素，它确保了数据的有序性和快速访问。这种数据结构的主要应用场景包括数据排序、搜索优化以及在动态数据集中管理数据。**BinarySortTree** 类通过提供简单而强大的接口，使得数据处理变得更加高效和直观。

2.2.5.2 类定义

```
template <typename Type>
class BinarySortTree : public MyBinaryTree<Type> {
private:
    void insertPrivate(const Type& item,
MyBinTreeNode<Type>*& subTree);
public:
    void insert(const Type& item) { insertPrivate(item,
this->root); }
    bool search(const Type& item);
    void outputBST(void);
};
```

2.2.5.3 私有成员函数

```
void insertPrivate(const Type& item, MyBinTreeNode<Type>*&
subTree);
```

这是一个递归函数，用于在二叉排序树中插入新元素。它递归地在树中找到正确的位置来插入新节点，保持树的排序属性不变。

2.2.5.4 公有成员函数

```
void insert(const Type& item) { insertPrivate(item,
this->root); }
```

此公有成员函数为用户提供了插入新元素到树中的接口。它调用

`insertPrivate` 函数，实现在正确位置插入新节点的功能。

```
bool search(const Type& item);
```

此函数用于在二叉排序树中查找特定元素。如果找到该元素，返回 `true`；否则返回 `false`。这一操作利用了二叉排序树的特性，确保查找过程高效。

```
void outputBST(void);
```

此函数用于输出二叉排序树的内容，通过中序遍历实现，以展示树中的数据项及其顺序。

2.3 项目主体架构设计

项目主体架构设计为：

- (1) 系统启动提示；
- (2) 二叉排序树的初始化：
 - ① 创建二叉排序树；
 - ② 数据输入；
 - ③ 查重操作；
 - ④ 插入操作；
- (3) 打印树结构；
- (4) 用户选择操作；
- (5) 程序退出。

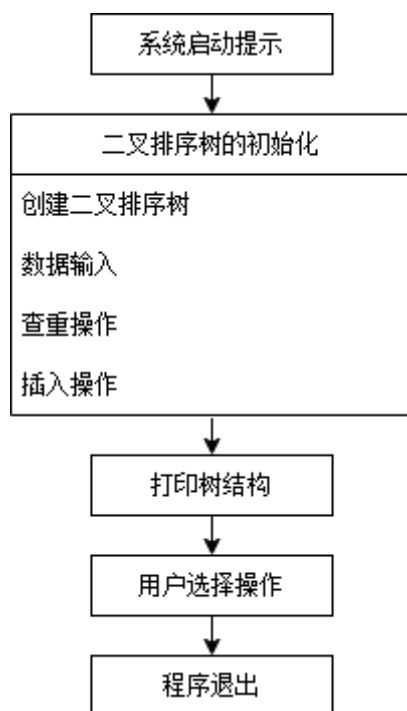


图 2.3.1 项目主体架构设计流程图

3 项目功能实现

3.1 项目主体架构的实现

3.1.1 项目主体架构实现思路

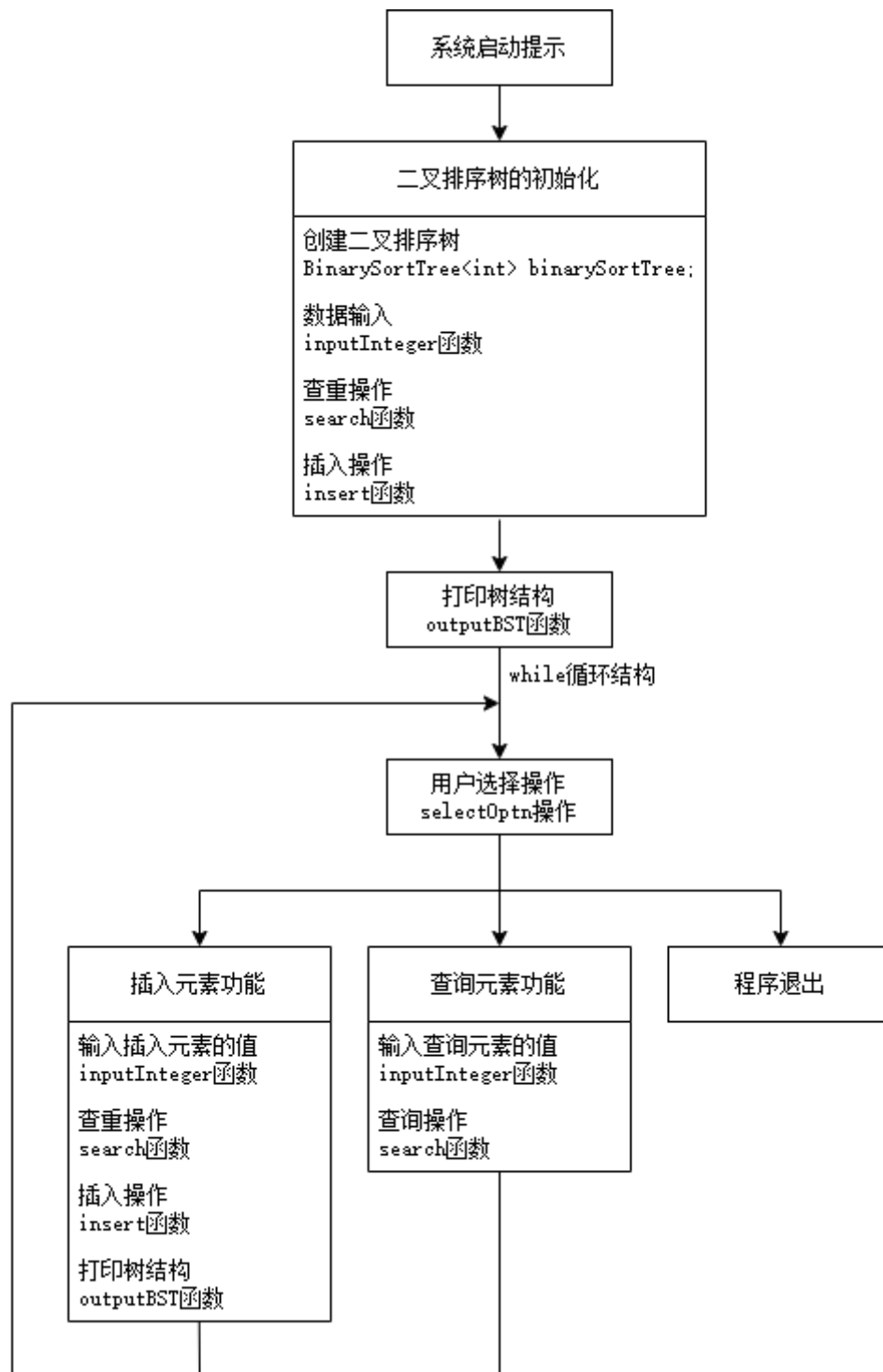


图 3.1.1.1 项目主体架构实现流程图

项目主体架构实现思路为：

(1) 系统启动提示：程序开始时，会在控制台打印提示信息，这提供了一个用户友好的界面来引导用户建立二叉排序树；

(2) 二叉排序树的初始化

① 创建二叉排序树：通过 `BinarySortTree<int>` 类型，创建一个二叉排序树实例 `binarySortTree`，专门用于处理整数类型的数据；

② 数据输入：用户被提示输入要在二叉排序树中存储的元素数量，以及具体的元素值。这些值将被用来构建树；

③ 查重操作与插入操作：用户输入的每个值都会通过 `binarySortTree.insert(val)` 方法插入到树中。在插入之前，程序会先通过 `binarySortTree.search(val)` 检查这个值是否已经存在于树中。如果存在，将提示用户重新输入；

(3) 打印树结构：一旦树被构建，使用 `binarySortTree.outputBST()` 方法将树的结构输出到控制台。这是通过中序遍历实现的；

(4) 用户选择操作：用户可以选择进行更多操作，如插入元素功能或查询元素功能。这允许用户插入新的数据项到树中，并支持快速查找树中是否存在特定的数据项。程序会持续地提供选项供用户进行操作，直到用户选择退出；

(5) 程序退出：当用户选择退出时，程序结束运行，返回值为 0。

3.1.2 项目主体架构核心代码

```
int main()
{
    /* System entry prompt */
    std::cout << "+-----+" << std::endl;
    std::cout << "|      二叉排序树      |" << std::endl;
    std::cout << "| Binary Sort Tree |" << std::endl;
    std::cout << "+-----+" << std::endl;

    /* Establish binary sort tree */
    BinarySortTree<int> binarySortTree;
    std::cout << std::endl << ">>> 请建立二叉排序树" <<
std::endl << std::endl;
    int num = inputInteger(SHRT_MIN, SHRT_MAX, "二叉排序树元
素个数");
    std::cout << std::endl << ">>> 请依次输入元素的值（使用空格
分隔）" << std::endl << std::endl;
    for (int i = 0; i < num; i++) {
        char tmp[64] = { 0 };
        sprintf(tmp, "第 %d 个元素的值", i + 1);
```

```

        int val = inputInteger(SHRT_MIN, SHRT_MAX, tmp);
        if (binarySortTree.search(val)) {
            std::cout << std::endl << ">>> 输入元素的值 " <<
val << " 在二叉排序树中已存在，请重新输入！" << std::endl <<
std::endl;
            i--;
        }
        else
            binarySortTree.insert(val);
    }
    std::cout << std::endl << ">>> 二叉排序树建立完成" <<
std::endl;
    binarySortTree.outputBST();

    /* Perform operations */
    while (true) {
        int optn = selectOptn();
        if (optn == 1)
            insertElement(binarySortTree);
        else if (optn == 2)
            searchElement(binarySortTree);
        else
            return 0;
    }
}

```

3.1.3 项目主体架构示例

```

+-----+
|  二叉排序树  |
| Binary Sort Tree |
+-----+

>>> 请建立二叉排序树
请输入二叉排序树元素个数 [整数范围：-32768~32767]: 5
>>> 请依次输入元素的值（使用空格分隔）
请输入第 1 个元素的值 [整数范围：-32768~32767]: 5
请输入第 2 个元素的值 [整数范围：-32768~32767]: -10
请输入第 3 个元素的值 [整数范围：-32768~32767]: 0
请输入第 4 个元素的值 [整数范围：-32768~32767]: -5
请输入第 5 个元素的值 [整数范围：-32768~32767]: 10
>>> 二叉排序树建立完成
>>> 二叉排序树（元素个数：5）：-10 -> -5 -> 0 -> 5 -> 10 -> [END]

```

图 3.1.3.1 项目主体架构示例

3.2 插入元素功能的实现

3.2.1 插入元素功能实现思路

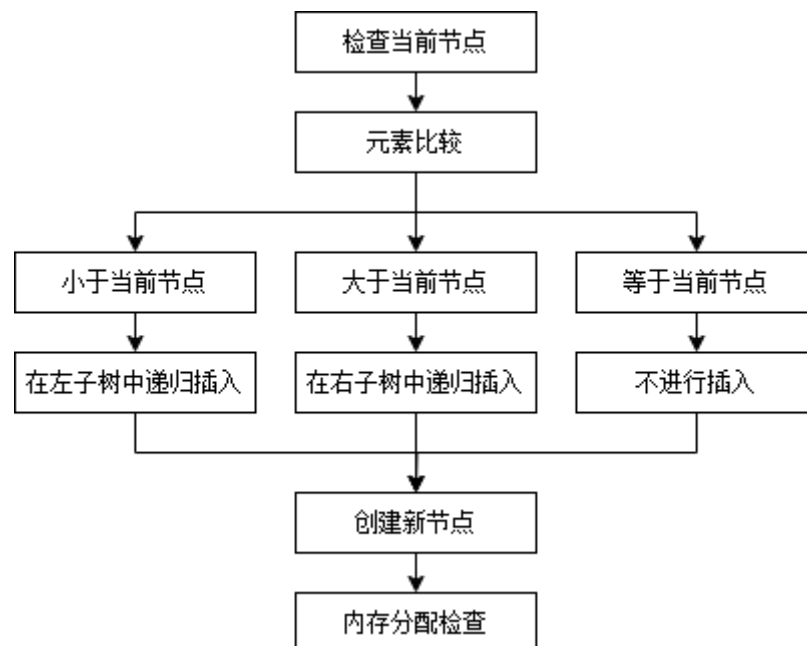


图 3.2.1.1 插入元素功能实现流程图

插入元素功能的函数为 **BinarySortTree** 类的成员函数 **insert**，插入元素功能实现的思路为：

(1) 检查当前节点：首先检查当前节点（初始时为树的根节点）。如果当前节点为空，这意味着已找到插入新元素的位置；

(2) 元素比较：如果当前节点不为空，则将待插入的元素与当前节点的数据进行比较；

① 小于当前节点：如果待插入元素小于当前节点的数据，则应在当前节点的左子树中继续寻找插入位置；

② 大于当前节点：如果待插入元素大于当前节点的数据，则应在当前节点的右子树中继续寻找插入位置；

③ 等于当前节点：如果待插入元素等于当前节点的数据，则不进行插入，因为二叉排序树不允许有重复的元素；

(3) 递归插入：通过递归调用 **insertPrivate** 函数，继续在相应的子树中寻找合适的插入位置；

(4) 创建新节点：当找到合适的插入位置（即遇到一个空的子节点），在该位置创建一个新的节点，并存储待插入的元素；

(5) 内存分配检查：在创建新节点时，需要检查内存分配是否成功。如果内存分配失败，则输出错误信息并终止程序。

3.2.2 插入元素功能核心代码

```
void insert(const Type& item) { insertPrivate(item, this->root); }

template <typename Type>
void BinarySortTree<Type>::insertPrivate(const Type& item,
MyBinTreeNode<Type>*& subTree)
{
    if (subTree == NULL) {
        subTree = new(std::nothrow) MyBinTreeNode<Type>(item);
        if (subTree == NULL) {
            std::cerr << "Error: Memory allocation failed." << std::endl;
            exit(MEMORY_ALLOCATION_ERROR);
        }
    }
    else if (item < subTree->data)
        insertPrivate(item, subTree->leftChild);
    else if (item > subTree->data)
        insertPrivate(item, subTree->rightChild);
    // If item is equal to subTree->data, then it is not inserted,
    // because binary sort tree (BST) does not allow duplicate values.
}

void insertElement(BinarySortTree<int>& binarySortTree)
{
    std::cout << std::endl;
    int val = inputInteger(SHRT_MIN, SHRT_MAX, "插入元素的值");
    if (binarySortTree.search(val))
        std::cout << std::endl << ">>> 输入元素的值 " << val << " 在二
叉排序树中已存在!" << std::endl;
    else {
        binarySortTree.insert(val);
        binarySortTree.outputBST();
    }
}
```

3.2.3 插入元素功能示例

```
>>> 菜单: [1]插入元素 [2]查询元素 [0]退出程序
请选择操作类型: [1]
请输入插入元素的值 [整数范围: -32768~32767]: 8
>>> 二叉排序树 (元素个数: 6) : -10 -> -5 -> 0 -> 5 -> 8 -> 10 -> [END]
```

图 3.2.3.1 插入元素功能示例

3.3 查询元素功能的实现

3.3.1 查询元素功能实现思路

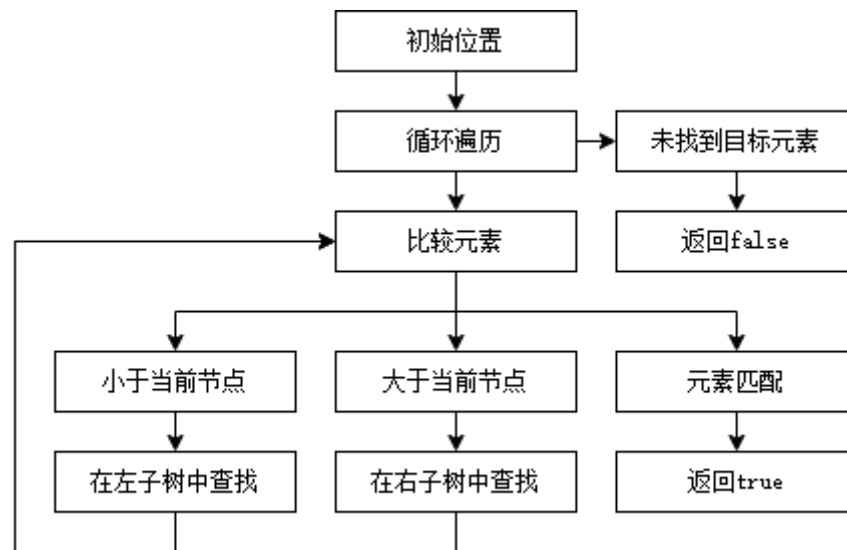


图 3.3.1.1 查询元素功能实现流程图

查询元素功能的函数为 `BinarySortTree` 类的成员函数 `search`，查询元素功能实现的思路为：

- (1) 初始位置：从树的根节点开始查询；
- (2) 循环遍历：使用一个循环来遍历树，该循环将持续进行，直到找到目标元素或遍历完所有可能的节点（即遇到一个空节点）；
- (3) 比较元素：在每个节点，将待查询的元素与当前节点的数据进行比较；
 - ① 元素匹配：如果待查询元素与当前节点的数据相等，则表明元素在树中存在，返回 `true`；
 - ② 小于当前节点：如果待查询元素小于当前节点的数据，则继续在当前节点的左子树中查找；
 - ③ 大于当前节点：如果待查询元素大于当前节点的数据，则继续在当前节点的右子树中查找；
- (4) 结束条件：如果找到了目标元素，则函数返回 `true`。如果遍历到一个空节点（即未找到目标元素），则函数返回 `false`。

3.3.2 查询元素功能核心代码

```
template <typename Type>
bool BinarySortTree<Type>::search(const Type& item)
{
    MyBinTreeNode<Type>* current = this->root;
    while (current != NULL) {
        if (item == current->data)
```

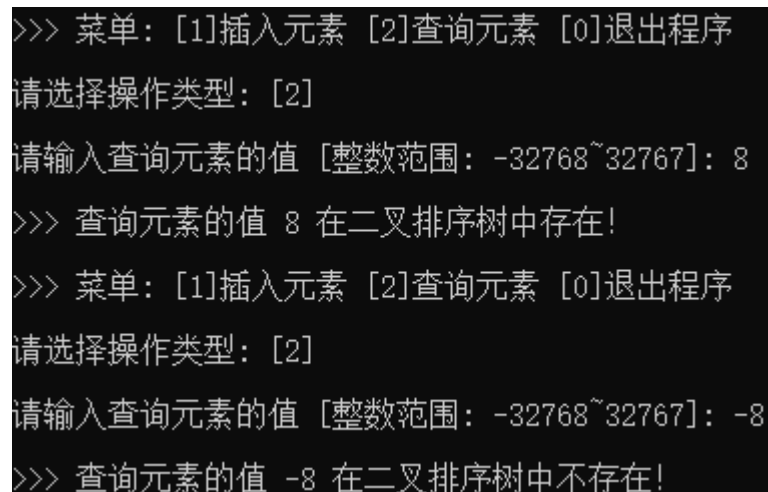
```

        return true;
    else if (item < current->data)
        current = current->leftChild;
    else
        current = current->rightChild;
    }
    return false;
}

void searchElement(BinarySortTree<int>& binarySortTree)
{
    std::cout << std::endl;
    int val = inputInteger(SHRT_MIN, SHRT_MAX, "查询元素的值");
    std::cout << std::endl << ">>> 查询元素的值 " << val << " 在二叉排序树中" << (binarySortTree.findNode(val, binarySortTree.getRoot()) ? " " : "不") << "存在!" << std::endl;
}

```

3.3.3 查询元素功能示例



```

>>> 菜单: [1]插入元素 [2]查询元素 [0]退出程序
请选择操作类型: [2]
请输入查询元素的值 [整数范围: -32768~32767]: 8
>>> 查询元素的值 8 在二叉排序树中存在!
>>> 菜单: [1]插入元素 [2]查询元素 [0]退出程序
请选择操作类型: [2]
请输入查询元素的值 [整数范围: -32768~32767]: -8
>>> 查询元素的值 -8 在二叉排序树中不存在!

```

图 3.3.3.1 查询元素功能示例

3.4 异常处理功能的实现

3.4.1 动态内存申请失败的异常处理

在进行 MyLinkNode 类和 MyBinTreeNode 类等的动态内存申请时，程序使用 new(std::nothrow) 来尝试分配内存。new(std::nothrow) 在分配内存失败时不会引发异常，而是返回一个空指针 (NULL 或 nullptr)，代码检查指针是否为空指针，如果为空指针，意味着内存分配失败，这时程序将执行以下操作：

- (1) 向标准错误流 std::cerr 输出一条错误消息 "Error: Memory

allocation failed."，指出内存分配失败；

(2)调用 `exit` 函数，返回错误码 `MEMORY_ALLOCATION_ERROR` (通过宏定义方式定义为-1)，用于指示内存分配错误，并导致程序退出。

下面是动态内存申请的异常处理的一个代码示例：

```
template <typename Type>
MyBinaryTree<Type>::MyBinaryTree(Type& item)
{
    root = new(std::nothrow) MyBinTreeNode<Type>(item);
    if (root == NULL) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(MEMORY_ALLOCATION_ERROR);
    }
}
```

3.4.2 MyQueue 类队列空的异常处理

在调用 `MyQueue` 类的 `deQueue` 和 `getFront` 等成员函数时，如果队列为空 (`isEmpty()`返回 `true`)，则函数返回 `false`，表示队列不包含任何元素。

3.4.3 MyBinaryTree 类自赋值的异常处理

`MyBinaryTree` 类重载了赋值运算符，用于将一个二叉树赋值给另一个二叉树。自赋值是一种常见的错误操作，可能导致资源泄漏和其他问题，该运算符重载函数对自赋值（将二叉树赋值给自身）的情况进行了处理。

程序会检查 `this` 指针和传入的 `other` 二叉树对象是否相同，如果 `this` 指针与 `other` 相同，表示自赋值操作，为了防止自赋值，运算符重载函数不会执行赋值操作，而是直接返回当前对象的引用 `*this`。这个错误处理机制确保了当尝试将二叉树赋值给自身时，不会导致资源泄漏或其他问题。

3.4.4 输入非法的异常处理

3.4.4.1 二叉排序树元素个数和元素的值输入非法的异常处理

程序通过调用 `inputInteger` 函数输入二叉排序树元素个数和元素的值。`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，函数的代码如下：

```
int inputInteger(int lowerLimit, int upperLimit, const char* prompt)
{
    while (true) {
        std::cout << "请输入" << prompt << " [整数范围: " << lowerLimit
        << "~" << upperLimit << "]: ";
        double tempInput;
        std::cin >> tempInput;
```



```

        if (std::cin.good() && tempInput == static_cast<int>(tempInput)
&& tempInput >= lowerLimit && tempInput <= upperLimit) {
            std::cin.clear();

std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            return static_cast<int>(tempInput);
        }
        else {
            std::cerr << std::endl << ">>> " << prompt << "输入不合法，
请重新输入" << prompt << "! " << std::endl << std::endl;
            std::cin.clear();

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

```

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 **tempInput** 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.good()**检查输入流的状态是否正常，确保没有发生数据类型输入错误，**tempInput==static_cast<int>(tempInput)**检查用户输入是否为整数，通过将其转换为整数再比较，**tempInput>=lowerLimit**和 **tempInput<=upperLimit** 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，函数会清除输入流的错误状态，丢弃输入缓冲区中的任何剩余内容，然后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。

3.4.4.2 操作类型输入非法的异常处理

操作类型输入非法的异常处理通过如下代码实现：

```

int selectOptn(void)
{
    std::cout << std::endl << ">>> 菜单： [1]插入元素 [2]查询元素 [0]退出程序" << std::endl;
    std::cout << std::endl << "请选择操作类型： ";
    char optn;
    while (true) {
        optn = _getch();
        if (optn == 0 || optn == -32)

```

```

        optn = _getch();
    else if (optn >= '0' && optn <= '2') {
        std::cout << "[" << optn << "]" << std::endl;
        return optn - '0';
    }
}
}

```

这段代码会一直等待用户输入，只有当用户输入有效的数字字符（'0'到'2'）时，才会返回该数字的整数值，表示用户选择的操作选项。如果用户输入无效字符，循环会继续等待用户提供有效的输入。这段代码的具体执行逻辑如下：

- (1) 显示提示信息，其中包含选项[1]和[2]，以及[0]退出系统的选项；
- (2) 进入一个无限循环，以等待用户输入；
- (3) 用户输入的字符会被_getch()函数获取。这个函数通常用于在控制台应用程序中获取单个字符而不显示在屏幕上。用户的输入存储在变量 optn 中；
- (4) 代码检查用户输入是否是数字字符（'0'到'2'）或特殊的控制字符。如果用户按下非数字字符或特殊控制字符，它将不执行下面的操作；
- (5) 如果用户输入是数字字符（'0'到'2'），它会在屏幕上显示用户输入的数字字符，并返回对应的整数值；
- (6) 如果用户输入的不是数字字符（'0'到'2'），代码将保持在循环中，继续等待有效的输入，这确保了只有在用户输入正确的选项时才会退出循环。

4 项目测试

4.1 输入二叉排序树元素个数功能测试

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。

```

请输入二叉排序树元素个数 [整数范围: -32768~32767]: -50000
>>> 二叉排序树元素个数输入不合法, 请重新输入二叉排序树元素个数!
请输入二叉排序树元素个数 [整数范围: -32768~32767]: 50000
>>> 二叉排序树元素个数输入不合法, 请重新输入二叉排序树元素个数!
请输入二叉排序树元素个数 [整数范围: -32768~32767]: 1.5
>>> 二叉排序树元素个数输入不合法, 请重新输入二叉排序树元素个数!

```

图 4.1.1 输入二叉排序树元素个数功能测试（输入超过上下限的整数和浮点数）

```

请输入二叉排序树元素个数 [整数范围: -32768~32767]: a
>>> 二叉排序树元素个数输入不合法, 请重新输入二叉排序树元素个数!
请输入二叉排序树元素个数 [整数范围: -32768~32767]: abc
>>> 二叉排序树元素个数输入不合法, 请重新输入二叉排序树元素个数!
请输入二叉排序树元素个数 [整数范围: -32768~32767]: 5

```

图 4.1.2 输入二叉排序树元素个数功能测试（输入字符和字符串及合法数据）

当输入合法时，程序继续运行。

4.2 建立二叉排序树功能测试

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。

```

>>> 请依次输入元素的值（使用空格分隔）
请输入第 1 个元素的值 [整数范围: -32768~32767]: -50000
>>> 第 1 个元素的值输入不合法, 请重新输入第 1 个元素的值!
请输入第 1 个元素的值 [整数范围: -32768~32767]: 10
请输入第 2 个元素的值 [整数范围: -32768~32767]: 50000
>>> 第 2 个元素的值输入不合法, 请重新输入第 2 个元素的值!
请输入第 2 个元素的值 [整数范围: -32768~32767]: -10
请输入第 3 个元素的值 [整数范围: -32768~32767]: 2.5
>>> 第 3 个元素的值输入不合法, 请重新输入第 3 个元素的值!
请输入第 3 个元素的值 [整数范围: -32768~32767]: 0
请输入第 4 个元素的值 [整数范围: -32768~32767]: a
>>> 第 4 个元素的值输入不合法, 请重新输入第 4 个元素的值!
请输入第 4 个元素的值 [整数范围: -32768~32767]: abc
>>> 第 4 个元素的值输入不合法, 请重新输入第 4 个元素的值!
请输入第 4 个元素的值 [整数范围: -32768~32767]: -5

```

图 4.2.1 输入二叉排序树元素的值功能测试（输入非法数据）

输入在二叉排序树中已存在的元素的值，程序提示输入元素的值在二叉排序树中已存在，并要求用户重新输入。可以验证程序对输入在二叉排序树中已存在的元素的值的情况进行了处理。

当输入合法时，程序继续运行。

```

请输入第 5 个元素的值 [整数范围: -32768~32767]: 10
>>> 输入元素的值 10 在二叉排序树中已存在, 请重新输入!
请输入第 5 个元素的值 [整数范围: -32768~32767]: -10
>>> 输入元素的值 -10 在二叉排序树中已存在, 请重新输入!
请输入第 5 个元素的值 [整数范围: -32768~32767]: 5
>>> 二叉排序树建立完成
>>> 二叉排序树 (元素个数: 5) : -10 -> -5 -> 0 -> 5 -> 10 -> [END]

```

图 4.2.2 输入二叉排序树元素的值功能测试 (输入在二叉排序树中已存在的元素的值)

4.3 插入元素功能测试

分别输入超过上下限的整数、浮点数、字符、字符串以及在二叉排序树中已存在的元素的值, 可以验证程序对输入非法和输入在二叉排序树中已存在的元素的值的情况进行了处理。

```

>>> 菜单: [1]插入元素 [2]查询元素 [0]退出程序
请选择操作类型: [1]
请输入插入元素的值 [整数范围: -32768~32767]: -50000
>>> 插入元素的值输入不合法, 请重新输入插入元素的值!
请输入插入元素的值 [整数范围: -32768~32767]: 50000
>>> 插入元素的值输入不合法, 请重新输入插入元素的值!
请输入插入元素的值 [整数范围: -32768~32767]: 1.5
>>> 插入元素的值输入不合法, 请重新输入插入元素的值!
请输入插入元素的值 [整数范围: -32768~32767]: a
>>> 插入元素的值输入不合法, 请重新输入插入元素的值!
请输入插入元素的值 [整数范围: -32768~32767]: abc
>>> 插入元素的值输入不合法, 请重新输入插入元素的值!
请输入插入元素的值 [整数范围: -32768~32767]: 0
>>> 输入元素的值 0 在二叉排序树中已存在!

```

图 4.3.1 插入元素功能测试

当输入合法时, 程序继续运行。

4.4 查询元素功能测试

分别输入超过上下限的整数、浮点数、字符、字符串以及在二叉排序树中存在和不存在的元素的值，可以验证程序对输入非法和输入在二叉排序树中存在和不存在的元素的值的情况进行了处理。

```
>>> 菜单：[1]插入元素 [2]查询元素 [0]退出程序
请选择操作类型：[2]
请输入查询元素的值 [整数范围：-32768~32767]：-50000
>>> 查询元素的值输入不合法，请重新输入查询元素的值！
请输入查询元素的值 [整数范围：-32768~32767]：50000
>>> 查询元素的值输入不合法，请重新输入查询元素的值！
请输入查询元素的值 [整数范围：-32768~32767]：1.5
>>> 查询元素的值输入不合法，请重新输入查询元素的值！
请输入查询元素的值 [整数范围：-32768~32767]：a
>>> 查询元素的值输入不合法，请重新输入查询元素的值！
请输入查询元素的值 [整数范围：-32768~32767]：abc
>>> 查询元素的值输入不合法，请重新输入查询元素的值！
请输入查询元素的值 [整数范围：-32768~32767]：0
>>> 查询元素的值 0 在二叉排序树中存在！
>>> 菜单：[1]插入元素 [2]查询元素 [0]退出程序
请选择操作类型：[2]
请输入查询元素的值 [整数范围：-32768~32767]：-1
>>> 查询元素的值 -1 在二叉排序树中不存在！
```

图 4.4.1 查询元素功能测试

当输入合法时，程序继续运行。

5 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

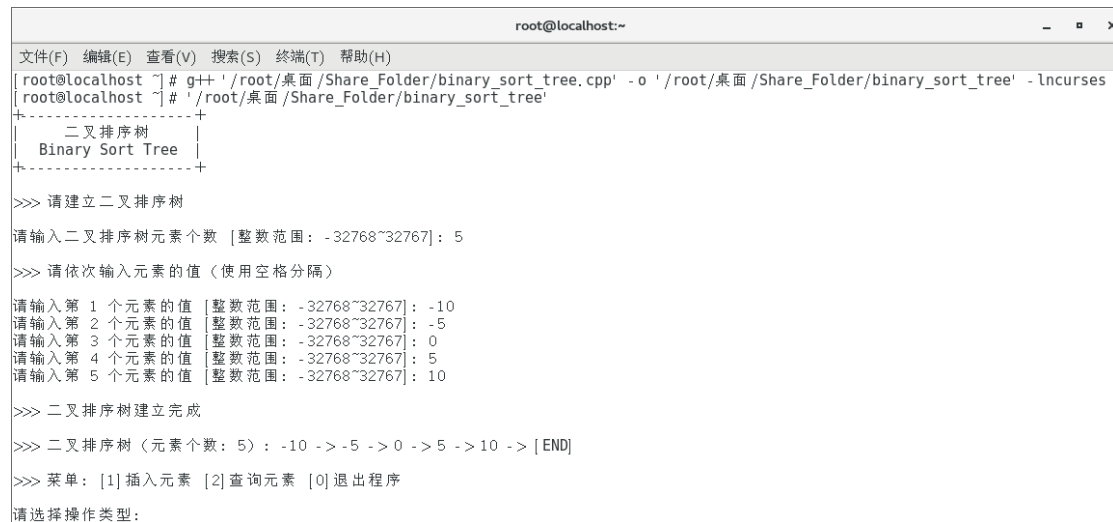
Linux 系统: CentOS 7 x64

Linux 编译命令:

```
g++ '/root/桌面/Share_Folder/binary_sort_tree.cpp' -o '/root/桌面/Share_Folder/binary_sort_tree' -lnurses
```

Linux 运行命令:

```
'/root/桌面/Share_Folder/binary_sort_tree'
```



```
root@localhost:~  
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)  
[root@localhost ~] # g++ '/root/桌面/Share_Folder/binary_sort_tree.cpp' -o '/root/桌面/Share_Folder/binary_sort_tree' -lnurses  
[root@localhost ~] # './root/桌面/Share_Folder/binary_sort_tree'  
+-----+  
|  二叉排序树  |  
| Binary Sort Tree |  
+-----+  
>>> 请建立二叉排序树  
请输入二叉排序树元素个数 [整数范围: -32768~32767]: 5  
>>> 请依次输入元素的值 (使用空格分隔)  
请输入第 1 个元素的值 [整数范围: -32768~32767]: -10  
请输入第 2 个元素的值 [整数范围: -32768~32767]: -5  
请输入第 3 个元素的值 [整数范围: -32768~32767]: 0  
请输入第 4 个元素的值 [整数范围: -32768~32767]: 5  
请输入第 5 个元素的值 [整数范围: -32768~32767]: 10  
>>> 二叉排序树建立完成  
>>> 二叉排序树 (元素个数: 5): -10 -> -5 -> 0 -> 5 -> 10 -> [END]  
>>> 菜单: [1] 插入元素 [2] 查询元素 [0] 退出程序  
请选择操作类型:
```

图 5.1 Linux 环境程序运行示例

本项目使用条件编译解决 Windows 系统和 Linux 系统编译环境的差异, 示例代码如下。

```
#ifdef _WIN32  
#include <conio.h>  
#elif __linux__  
#include <nurses.h>  
#endif
```