项目说明文档

数据结构课程设计

——二叉排序树

作 者 姓 名： 刘畅

学 号： 2054164

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目 录

[1 分析 1](#_Toc88865957)

[1.1 背景分析 1](#_Toc88865958)

[1.2 功能分析 1](#_Toc88865959)

[2 设计 1](#_Toc88865960)

[2.1 数据结构设计 1](#_Toc88865961)

[2.2 类结构设计 1](#_Toc88865962)

[2.3 成员与操作设计 1](#_Toc88865963)

[2.4 系统设计 2](#_Toc88865964)

[3 实现 2](#_Toc88865965)

[3.1 插入功能的实现 2](#_Toc88865966)

[3.1.1 实现方法 2](#_Toc88865967)

[3.1.2核心代码 2](#_Toc88865968)

[3.2 删除功能的实现 3](#_Toc88865969)

[3.2.1 实现方法 3](#_Toc88865970)

[3.2.2 核心代码 3](#_Toc88865971)

[3.3 搜索功能的实现 4](#_Toc88865972)

[3.3.1 实现方法 4](#_Toc88865973)

[3.3.2 核心代码 4](#_Toc88865974)

# 1 分析

## 1.1 背景分析

二叉查找树（Binary Search Tree），（又：二叉搜索树，二叉排序树）它或者是一棵空树，或者是具有下列性质的二叉树： 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值； 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值； 它的左、右子树也分别为二叉排序树。二叉搜索树作为一种经典的数据结构，它既有链表的快速插入与删除操作的特点，又有数组快速查找的优势；所以应用十分广泛，例如在文件系统和数据库系统一般会采用这种数据结构进行高效率的排序与检索操作。

## 1.2 功能分析

一般而言，对于一棵二叉查找树，具有一下三种功能：1插入一个数据，2查询是否包含某个数值，2删除某个数值；此外，还应当有清空整个二叉树、遍历等操作。

# 2 设计

## 2.1 数据结构设计

如上功能分析所述，该系统需要大量的插入、查询、删除功能，因此考虑使用链表来构造二叉树，在本项目中采用二叉链表实现二叉查找树的结构。

## 2.2 类结构设计

首先有二叉树结点类，包含其左孩子和右孩子指针，以及该结点的数据；其次是二叉搜索树类，其中包含根结点指针，以及各种成员函数的实现。

## 2.3 成员与操作设计

**二叉树结点类（BSTNode）**

T \_data;

BSTNode<T>\* \_left, \* \_right;

BSTNode() { \_left = NULL; \_right = NULL; }

BSTNode(const T data, BSTNode<T>\* left = NULL, BSTNode<T>\* right = NULL) : \_data(data), \_left(left), \_right(right) {}

**二叉搜索树类（BanarySearchTree）**

public:

BSTNode<T>\* getRoot() { return \_root; }

int getNum() { return \_number; }

BinarySearchTree() : \_root(NULL), \_number(0) {}//构造一棵空树

BinarySearchTree(const T x) { \_root = new BSTNode<T>(x); } //构造根节点

~BinarySearchTree() { Clear(\_root); }

bool Remove(const T& x) {return Remove(x, \_root); } //二叉搜索树的删除算法

bool Insert(const T& x) { return Insert(x, \_root); }//二叉搜索树的插入算法

bool Search(const T x) {return Search(x, \_root); } //二叉搜索树的搜索算法

void Clear() { Clear(\_root); \_root = NULL; }//二叉搜索树的清空操作

void inOrder(void(\*visit)(BSTNode<T>\* p)) {

if (\_root == NULL) { cout << "the tree is empty!\n"; }

inOrder(\_root, visit);

}//中序遍历

## 2.4 系统设计

系统首先生成一棵空的二叉搜索树，其根结点指向空；而后调用PrintInterface()打印提示词，再根据用户输入的操作码进行对应功能的执行。

# 3 实现

## 3.1 插入功能的实现

### 3.1.1 实现方法

采用递归算法，在以ptr为根的二叉搜索树中插入元素x，若x小于ptr结点的值，则在ptr的左子树中进行插入，若x大于ptr结点的值，则在ptr的右子树中进行插入，不然则x的值等于ptr结点的值，此时返回false，其余返回true。最后，如若ptr为空，则表明就该在该位置插入x。

### 3.1.2核心代码

bool Insert(const T& x, BSTNode<T>\*& ptr) {

if (ptr == NULL) {

ptr = new BSTNode<T>(x);

\_number++;

return true;

}

else if (x < ptr->\_data) { return Insert(x, ptr->\_left); }

else if (x > ptr->\_data) { return Insert(x, ptr->\_right); }

else { return false; }

}

## 3.2 删除功能的实现

### 3.2.1 实现方法

首先根据要删除的值找到要删除的结点，由于二叉搜索树中每个元素的值只有一个，因此x对应的结点是唯一的。（若不存在则返回false）

观察该结点

### 3.2.2 核心代码

bool Remove(const T x, BSTNode<T>\*& ptr) {

BSTNode<T>\* temp;

if (ptr != NULL) {

if (x < ptr->\_data) { return Remove(x, ptr->\_left); }

else if (x > ptr->\_data) { return Remove(x, ptr->\_right); }

else if (ptr->\_left != NULL && ptr->\_right != NULL) {

temp = ptr->\_right;

while (temp->\_left != NULL) { temp = temp->\_left; }

ptr->\_data = temp->\_data;

Remove(ptr->\_data, ptr->\_right);

}

else {

//ptr即为需要删除的结点

temp = ptr;

if (ptr->\_left == NULL) { ptr = ptr->\_right; }

else { ptr = ptr->\_left; }

delete temp;

\_number--;

return true;

}

}

return false;

}

## 3.3 搜索功能的实现

### 3.3.1 实现方法

采用递归算法，先判断当前结点的值是否等于要查找的值，若相等则返回true；若不相等则判断它与结点值的相对大小，小于结点值则查找其左子树，否则查找其右子树。

### 3.3.2 核心代码

bool Search(const T x, BSTNode<T>\* ptr) {

if (ptr == NULL) { return false; }

else if (x < ptr->\_data) { return Search(x, ptr->\_left); }

else if (x > ptr->\_data) { return Search(x, ptr->\_right); }

else { return true; }

}