

项目说明文档

数据结构课程设计

——二叉排序树

作者姓名：_____杨鑫_____

学号：_____1950787_____

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 -
1.1	背景分析.....	- 1 -
1.2	功能分析.....	- 1 -
2	设计.....	- 1 -
2.1	数据结构设计.....	- 1 -
2.2	类结构设计.....	- 1 -
2.3	成员与操作设计.....	- 2 -
2.4	系统设计.....	- 3 -
3	实现.....	- 4 -
3.1	插入元素功能的实现.....	- 4 -
3.1.1	插入元素功能的流程图.....	- 4 -
3.1.2	插入元素功能的核心代码.....	- 4 -
3.1.3	插入元素功能的截屏示例.....	- 5 -
3.2	删除元素功能的实现.....	- 5 -
3.2.1	删除元素功能的流程图.....	- 5 -
3.2.2	删除元素功能的核心代码.....	- 5 -
3.2.3	删除元素功能的截屏示例.....	- 6 -
3.3	查找元素功能的实现.....	- 6 -
3.3.1	查找元素功能的流程图.....	- 6 -
3.3.2	查找元素功能的核心代码.....	- 7 -
3.3.3	查找元素功能的截屏示例.....	- 7 -
4	测试.....	- 7 -
4.1	功能测试.....	- 7 -
4.1.1	完整功能综合测试.....	- 7 -
4.2	边界测试.....	- 8 -
4.2.1	AVL 树中无元素时输出测试.....	- 8 -
4.3	出错测试.....	- 9 -
4.3.1	插入重复元素测试.....	- 9 -
4.3.2	删除不存在元素测试.....	- 9 -

1 分析

1.1 背景分析

在现在这个信息技术高速发展的时代，信息内容爆炸般地增长，如何快速处理信息成为一个重要的问题。在处理数据时，一个非常重要的操作就是给数据排序。我们有时候在存储数据时，希望将来在找到其中某个数据时能够尽量快速的找到它，这时就不能依靠传统的数组或者链表存储数据了，因为它们的查找效率比较低。

二叉排序树就是指将原来已有的数据根据大小构成一棵二叉树，二叉树中的所有结点数据满足一定的大小关系，所有的左子树中的结点均比根结点小，所有的右子树的结点均比根结点大。

在二叉排序树中查找就是指按照二叉排序树中结点的关系进行查找，查找关键自首先同根结点进行比较，如果相等则查找成功；如果比根节点小，则在左子树中查找；如果比根结点大，则在右子树中进行查找。这种查找方法可以快速缩小查找范围，大大减少查找关键的比较次数，从而提高查找的效率。

1.2 功能分析

依次输入关键字并建立二叉排序树，实现二叉排序数的插入和查找功能。此外，还要支持用户增加和删除元素，打印二叉树中信息的功能。

2 设计

2.1 数据结构设计

由于要实现快速的查找，添加和删除操作，所以本程序会使用二叉排序树的数据结构。但考虑到普通的二叉排序树的效率在最坏的情况下并不高（当在极端情况下，建立的二叉树可能是一个只有左/右子节点的树，在形式上也就类似于一个单链表），增加，删除，查找操作的时间复杂度都会变成 $O(n)$ 。当然在平均情况下时间复杂度还是 $O(\log n)$ ，比一般的数组和链表要好很多。基于此，本程序做了一定的改进，采用平衡二叉排序树（AVL）来作为核心数据结构，这样本程序的增加，删除和查找的时间复杂度在最坏的情况下仍然是 $O(\log n)$ ，大大提高了性能。

2.2 类结构设计

本程序核心类是平衡二叉树类（AVLTree）和它的节点结构体（AVLNode）。为了实现它的插入删除后仍然是一颗平衡二叉树，必须实现它的旋转功能并相应

的在添加和删除节点时做一些额外的操作。为了实现这些，还要实现一个栈类（LinkStack）以及它的节点结构体（LinkNode）。此外，还设计了一个系统类（System），用于存放和维护这个平衡二叉树，并实现相应的一些功能。

为了使数据结构更具有泛用性，本系统将 LinkStack 类，AVLTree 类等都设计为了模板类。

2.3 成员与操作设计

节点结构体（LinkNode）：

```
1. T data;
2. LinkNode <T>* link;
3. LinkNode(LinkNode<T>* ptr = NULL) : link(ptr) {}; // 构造函数
4. LinkNode(const T& tem, LinkNode<T>* ptr = NULL) : data(tem), link(ptr) {}; // 构造函数
```

栈类（LinkStack）：

私有成员：

```
1. LinkNode<T>* top; // 栈顶元素
```

公有操作：

```
1. LinkStack() : top(NULL) {} // 构造函数
2. ~LinkStack() { makeEmpty(); } // 析构函数
3. void Push(const T& x); // 入栈
4. bool Pop(T& x); // 出栈
5. bool getTop(T& x) const; // 得到栈顶元素
6. bool IsEmpty() const { return top == NULL; } // 判断是否栈空
7. int getSize() const; // 返回栈中元素个数
8. void makeEmpty(); // 栈置空
```

AVL 树节点结构体（AVLNode）：

```
1. K key;
2. int bf;
3. AVLNode<K>* left, * right;
4. AVLNode() : bf(0), left(NULL), right(NULL) {} // 构造函数
5. AVLNode(K newKey, AVLNode<K>* newLeft = NULL, AVLNode<K>* newRight = NULL) : key(newKey), bf(0), left(newLeft), right(newRight) {} // 构造函数
6. ~AVLNode() {} // 析构函数
7.
8. // 三个重载比较运算符的函数
9. bool operator < (const AVLNode<K>& AN) { return this->key < AN.key; }
10. bool operator > (const AVLNode<K>& AN) { return this->key > AN.key; }
```

```
11. bool operator == (const AVLNode<K>& AN) { return this->key == AN.
    key; }
```

AVL 树类 (AVLTree) :

私有成员:

```
1. AVLNode<K>* root;
2. bool Insert(AVLNode<K>*& ptr, K& d); // 插入
3. bool Remove(AVLNode<K>*& ptr, K& d); // 删除
4. void RotateL(AVLNode<K>*& ptr); // 左单旋转
5. void RotateR(AVLNode<K>*& ptr); // 右单旋转
6. void RotateLR(AVLNode<K>*& ptr); // 先左后右旋转
7. void RotateRL(AVLNode<K>*& ptr); // 先右后左旋转
8. AVLNode<K>* Search(K& d, AVLNode<K>*& ptr); // 查找函数
9. void makeEmpty(AVLNode<K>*& ptr); // 置空
10. void Show(AVLNode<K>*& ptr); // 输出
```

公有操作:

```
1. AVLTree() : root(NULL) {} // 构造函数
2. ~AVLTree() { makeEmpty(root); } // 析构函数
3. AVLNode<K>* Search(K& d) { return Search(d, root); } // 查找
4. bool Insert(K& d) { return Insert(root, d); } // 插入
5. bool Remove(K& d) { return Remove(root, d); } // 删除
6. void SetRoot(K& d); // 设置根节点
7. void Show() { Show(root); } // 输出
```

系统类: (System)

私有成员:

```
1. AVLTree<int>* Tree;
```

公有操作:

```
1. System(); // 构造函数
2. ~System(); // 析构函数
3. bool BuildTree(); // 建立树
4. void Show(); // 展示
5. bool Insert(); // 插入
6. bool Remove(); // 删除
7. bool Search(); // 寻找
8. void Loop(); // 主循环
```

2.4 系统设计

程序开始后不断从用户处接受指令并执行, 先根据一系列输入建立一个平衡

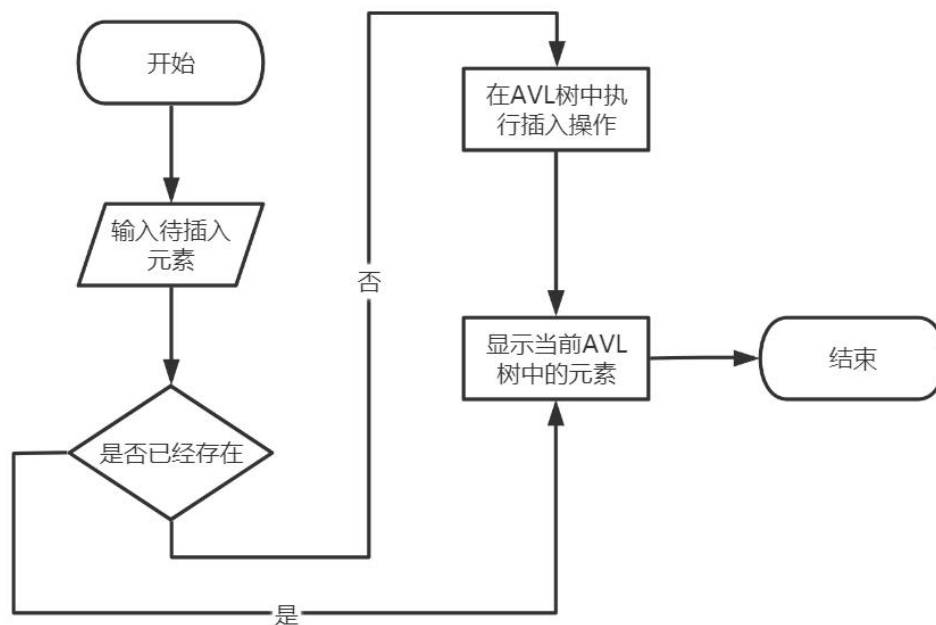
二叉树，然后可以基于平衡二叉树执行增删查等操作，并且可以保证拥有较高的性能。

程序兼容了 windows 和 LINUX 平台，在双平台下均可以正常运行。

3 实现

3.1 插入元素功能的实现

3.1.1 插入元素功能的流程图



3.1.2 插入元素功能的核心代码

```
1. // 尝试插入， 若失败， 说明已经存在该元素
2. if (Tree->Insert(num)) {
3.     cout << "Successfully inserted ! " << endl;
4.     return true;
5. }
6. else {
7.     cout << num << " is repeated ! " << endl;
8.     return false;
9. }
```

3.1.3 插入元素功能的截屏示例

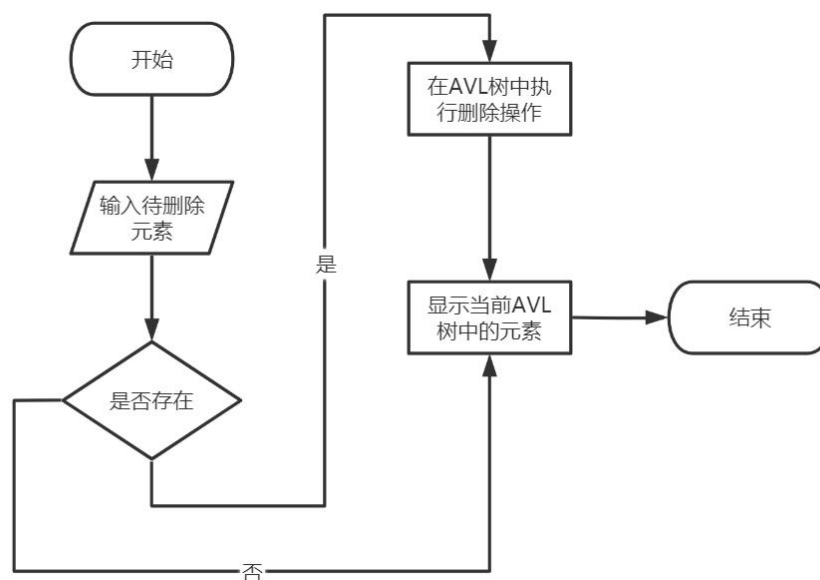
```
Please select : 1
Please input keys to create tree ( input '#' to stop ) :
32 43 12 65 #
The Tree is :
12 -> 32 -> 43 -> 65

Please select : 2
Please input key which inserted : 44
Successfully inserted !
The Tree is :
12 -> 32 -> 43 -> 44 -> 65

Please select :
```

3.2 删除元素功能的实现

3.2.1 删除元素功能的流程图



3.2.2 删除元素功能的核心代码

```
1. // 尝试删除， 若失败说明该元素不存在
2. if (Tree->Remove(num)) {
3.     cout << "Successfully removed ! " << endl;
4.     return true;
5. }
6. else {
7.     cout << num << " is not in ! " << endl;
8.     return false;
9. }
```

3.2.3 删除元素功能的截屏示例

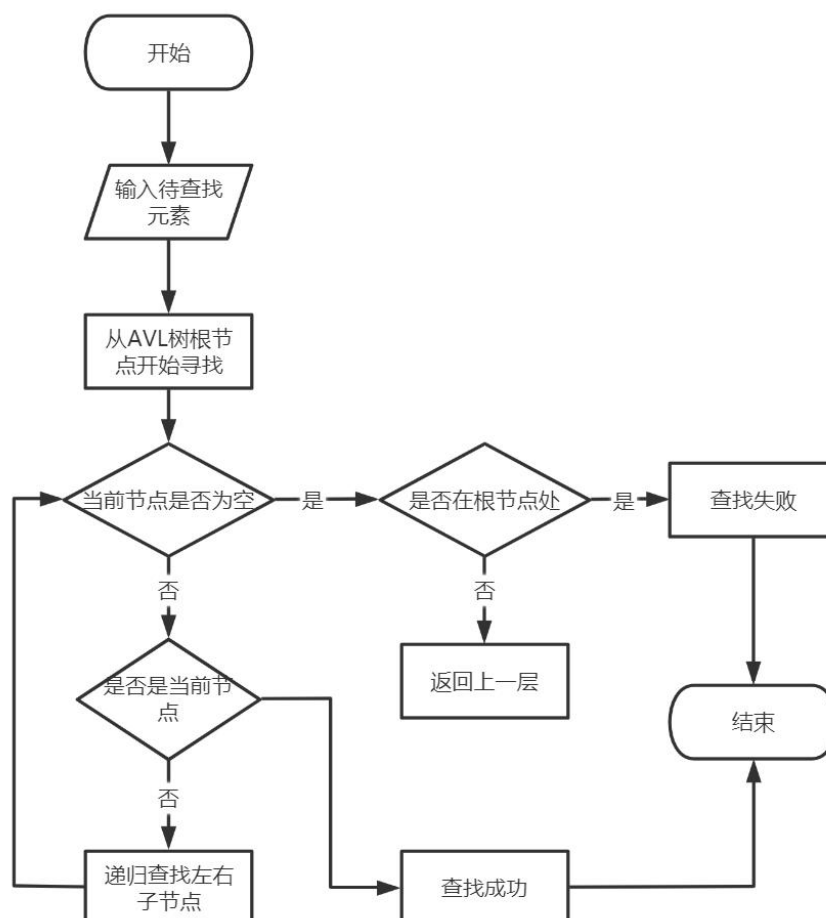
```
Please select : 1
Please input keys to create tree ( input '#' to stop ) :
32 43 12 65 #
The Tree is :
12 -> 32 -> 43 -> 65

Please select : 3
Please input key which removed : 32
Successfully removed !
The Tree is :
12 -> 43 -> 65

Please select :
```

3.3 查找元素功能的实现

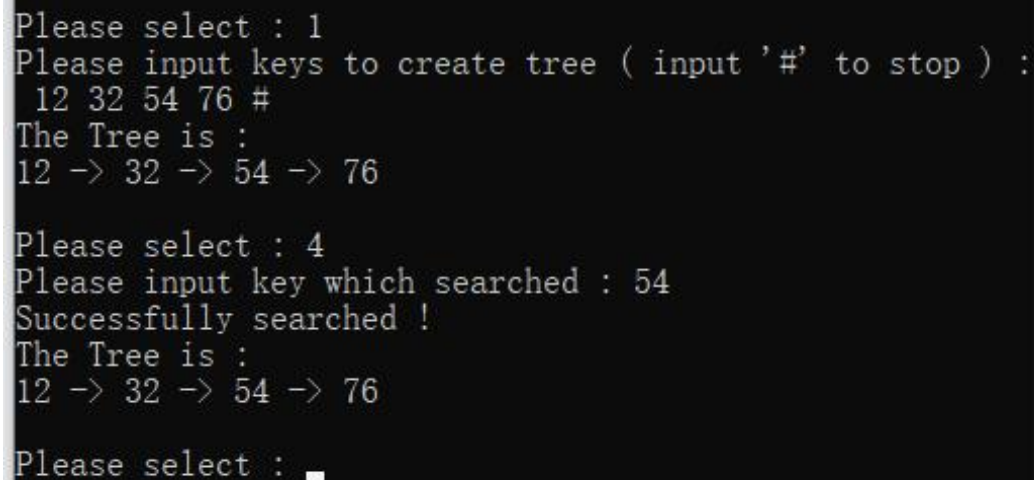
3.3.1 查找元素功能的流程图



3.3.2 查找元素功能的核心代码

```
1.  cout << "Please input key which searched : ";
2.  int num;
3.  cin >> num;
4.
5.  // 根据 AVL 树 的查询函数进行搜索
6.  if (Tree->Search(num)) {
7.      cout << "Successfully searched ! " << endl;
8.      return true;
9.  }
10. else {
11.     cout << "No this key ! " << endl;
12.     return false;
13. }
```

3.3.3 查找元素功能的截屏示例



```
Please select : 1
Please input keys to create tree ( input '#' to stop ) :
12 32 54 76 #
The Tree is :
12 -> 32 -> 54 -> 76

Please select : 4
Please input key which searched : 54
Successfully searched !
The Tree is :
12 -> 32 -> 54 -> 76

Please select : _
```

4 测试

4.1 功能测试

4.1.1 完整功能综合测试

运行截图：

```
Please select : 1
Please input keys to create tree ( input '#' to stop ) : 12 43 -32
0 -21 -4 43 653 -1 1 3 2 #
The key ( 43 ) is repeated !
The Tree is :
-32 -> -21 -> -4 -> -1 -> 0 -> 1 -> 2 -> 3 -> 12 -> 43 -> 653

Please select : 2
Please input key which inserted : 32
Successfully inserted !
The Tree is :
-32 -> -21 -> -4 -> -1 -> 0 -> 1 -> 2 -> 3 -> 12 -> 32 -> 43 -> 653

Please select : 3
Please input key which removed : 1
Successfully removed !
The Tree is :
-32 -> -21 -> -4 -> -1 -> 0 -> 2 -> 3 -> 12 -> 32 -> 43 -> 653

Please select : 4
Please input key which searched : -4
Successfully searched !
The Tree is :
-32 -> -21 -> -4 -> -1 -> 0 -> 2 -> 3 -> 12 -> 32 -> 43 -> 653

Please select : 5
The Tree is :
-32 -> -21 -> -4 -> -1 -> 0 -> 2 -> 3 -> 12 -> 32 -> 43 -> 653

Please select : 6
Welcome back soon !
```

4.2 边界测试

4.2.1 AVL 树中无元素时输出测试

测试用例：

5

预期结果：程序正常运行不崩溃，输出为空，表示 AVL 树中暂无元素。

实验结果：

```
***          二叉排序树          ***
=====
***          请选择要执行的操作：          ***
***          1 --- 建立二叉排序树          ***
***          2 --- 插入元素                  ***
***          3 --- 删除元素                  ***
***          4 --- 查找元素                  ***
***          5 --- 查看二叉树                ***
***          6 --- 退出程序                  ***
=====

Please select : 5
The Tree is :

Please select :
```

4.3 出错测试

4.3.1 插入重复元素测试

测试用例：

1
12 23 34 5 2 #
2
5

预期结果：程序正常运行不崩溃，提示用户已有该元素，插入失败。

实验结果：

```
Please select : 1
Please input keys to create tree ( input '#' to stop ) :
12 23 34 5 2 #
The Tree is :
2 -> 5 -> 12 -> 23 -> 34

Please select : 2
Please input key which inserted : 5
5 is repeated !
The Tree is :
2 -> 5 -> 12 -> 23 -> 34

Please select : _
```

4.3.2 删除不存在元素测试

测试用例：

1
12 23 34 5 2 #
3
6

预期结果：程序正常运行不崩溃，提示用户删除元素不存在，删除失败。

实验结果：

```
Please select : 1
Please input keys to create tree ( input '#' to stop ) :
12 23 34 5 2 #
The Tree is :
2 -> 5 -> 12 -> 23 -> 34

Please select : 3
Please input key which removed : 6
6 is not in !
The Tree is :
2 -> 5 -> 12 -> 23 -> 34

Please select : _
```