数据结构课程设计

项目说明文档

二叉排序树

|  |  |
| --- | --- |
| 作者姓名： | 罗斌江 |
| 学 号： | 2053285 |
| 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项目分析 3](#_Toc91051196)

[1.1项目背景 3](#_Toc91051197)

[1.2 项目需求分析 3](#_Toc91051198)

[1.3 项目要求 4](#_Toc91051199)

[1.3.1 功能要求 4](#_Toc91051200)

[1.3.2 输入格式 4](#_Toc91051201)

[1.3.3 输出格式 4](#_Toc91051202)

[2 项目设计 4](#_Toc91051203)

[2.1 数据结构设计 4](#_Toc91051204)

[2.2 类设计 4](#_Toc91051205)

[2.2.1平衡二叉搜索树结点类（AVLNode） 5](#_Toc91051206)

[2.2.2平衡二叉搜索树类（AVLTree） 5](#_Toc91051207)

[3 项目实现 7](#_Toc91051208)

[3.1 项目整体功能的实现 7](#_Toc91051209)

[3.1.1 项目整体功能流程图 7](#_Toc91051210)

[3.1.2 项目整体功能代码 7](#_Toc91051211)

[3.2二叉排序的实现 9](#_Toc91051212)

[3.2.1 二叉排序原理 9](#_Toc91051213)

[3.2.2 AVL树Insert函数 10](#_Toc91051214)

[4 项目测试 12](#_Toc91051215)

[4.1 插入元素 12](#_Toc91051216)

[4.2 查询元素 13](#_Toc91051217)

[4.3 退出 13](#_Toc91051218)

# 1项目分析

## 1.1项目背景

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。

二叉排序树查找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 项目需求分析

针对于排课软件这一系统，本项目在实现的过程中，考虑并且满足了以下的需求：

* 功能完善

插入、删除、查找元素。

* 执行效率高

针对数据量比较大的情况，本系统也应该具有在较短时间内求解出正确答案的能力。

* 代码可读性强

本项目在实现过程中，将代码根据功能的不同划分为了不同的代码块，同时进行了合理封装。

* 健壮性

当用户输入的数据不合理时，系统应当给予相应的提示而非直接报错。

* 可视化

该系统通过输出当前正在执行的操作内容，使得用户可以直观的了解到当前操作内容。

## 1.3 项目要求

### 1.3.1 功能要求

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。

### 1.3.2 输入格式

选中的操作；必要的数据 ；

### 1.3.3 输出格式

对应的输出；

# 2 项目设计

## 2.1 数据结构设计

按照题目要求要实现一个二叉排序树，一棵二又排序树或者为空，或者具有下面的性质：其根结点保存着一个数据项（及其关键码）；如果其左子树不空，那么其左子树的所有结点保存的（关键码）值均小于（如果不要求严格小于，也可以是“不大于"）根结点保存的（关键码）值；如果其右子树不空，那么其右子树的所有结点保存的（关犍码）值均大于它的根结点保存的（关犍码）值；

非空的左子树或右子树也是二叉排序树。

## 2.2 类设计

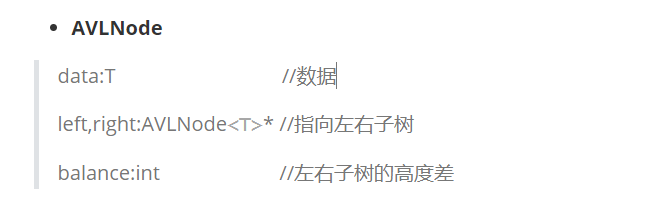
本项目中使用到**二叉排序树**这一数据结构，但是普通的二叉搜索树在某些情况下会产生退化，甚至变成线性结构，性能大大减弱，所以本项目选择平衡二叉树；

。平衡情况下，本系统实现的平衡二叉树结构各种操作的时间复杂度如下：

* 插入操作：O()
* 查询操作：O()

### 2.2.1平衡二叉搜索树结点类（AVLNode）

作为平衡二叉搜索树的结点，比普通二叉搜索树结点多了一个balance变量，记录左右子树高度差。其UNL图如下所示：



### 2.2.2平衡二叉搜索树类（AVLTree）

平衡二叉树是对二叉搜索树的优化，所以它首先满足二叉搜索树的一切性质和特征，不同的是，相较于普通的二叉搜索树，平衡二叉搜索树每个结点多了一个balance变量，记录其左右子树高度的差值，balance只能为-1，0，1，且AVL树类多了四种旋转调整函数。其UNL图如下所示：



树中主要函数如下所示：

* int Insert(T&x)

调用私有insert()函数，将x插入树中。

* void Traverse(AVLNOde<T>\*Tree,ostream&out)

以Tree为根结点中序输出该树中的数据。

* void **RotateLeft(AVLNode<T>\*&Tree)**

在指定结点对树进行左单旋操作。

* void **RotateRight(AVLNode<T>\*&Tree)**

在指定结点对树进行右单旋操作。

* void **RotateLR(AVLNode<T>\*&Tree)**

在指定结点对树进行左右双旋操作。

* void **RotateRL(AVLNode<T>\*&Tree)**

在指定结点对树进行右左双旋操作。

* istream& operator>>(istream&in,AVLTree<Type>&Tree)

重载运算符，定义AVL树的输入函数。

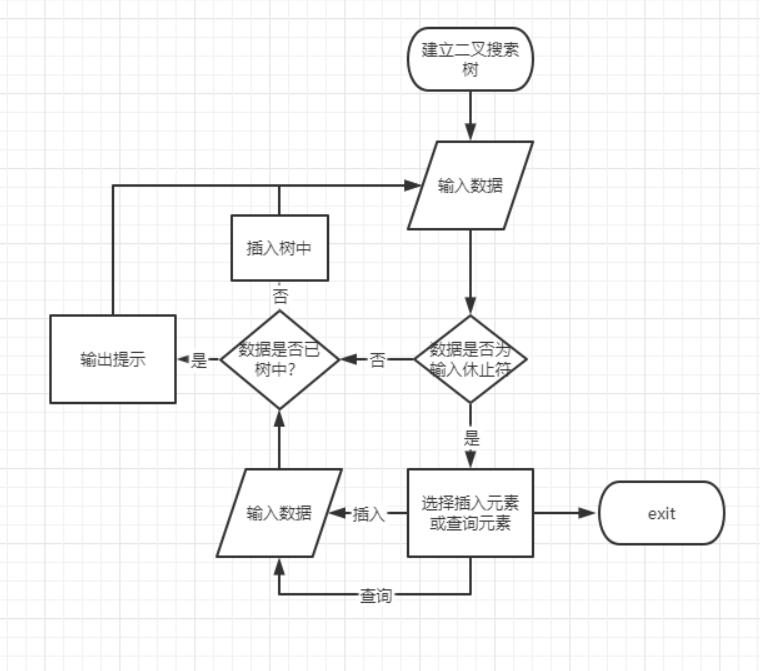
* ostream& operator<<(ostream&out,AVLTree<Type>&Tree)

重载运算符，定义AVL树的输出函数。

# 3 项目实现

# 3.1 项目整体功能的实现

### 3.1.1 项目整体功能流程图



### 3.1.2 项目整体功能代码

while(option!=4){

        std::cout<<"Please select:"<<std::endl;

        std::cin>>option;

        switch(option){

            case 1:{

                std::cin >> test;

                std::cout<<"Bsort\_Tree is:"<<std::endl;

                std::cout<<test;

                break;

            }

            case 2:{

                std::cout<<"Please input the key which inserted:"<<std::endl;

*int* key;

                std::cin >> key;

                test.Insert(key);

                std::cout<<"Bsort\_Tree is:"<<std::endl;

                std::cout<<test;

                break;

            }

            case 3:{

                std::cout<<"Please input the key which searched:"<<std::endl;

*int* key;

                std::cin >> key;

                if(test.Find(key)==NULL){

                    std::cout<<"could not find key "<<key<<std::endl;

                }

                else{

                    std::cout<<"Search Success"<<std::endl;

                }

                break;

            }

            case 4:{

                break;

            }

            default:{

                std::cout<<"Error: Unknown input"<<std::endl;

                break;

            }

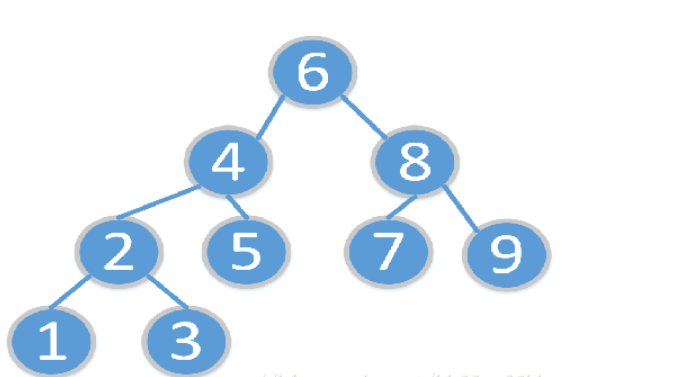
        }

    }

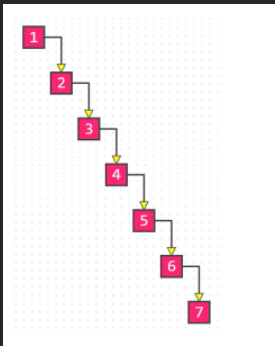
## 3.2二叉排序的实现

### 3.2.1 二叉排序原理

根据普通二叉搜索树的性质，任一结点上左子树的值均小于该结点，而右子树上的值均大于该结点，形成树状结构，结合中序遍历（先左子树后结点最后右子树的顺序）就可以在n次插入后输出有序的数据；

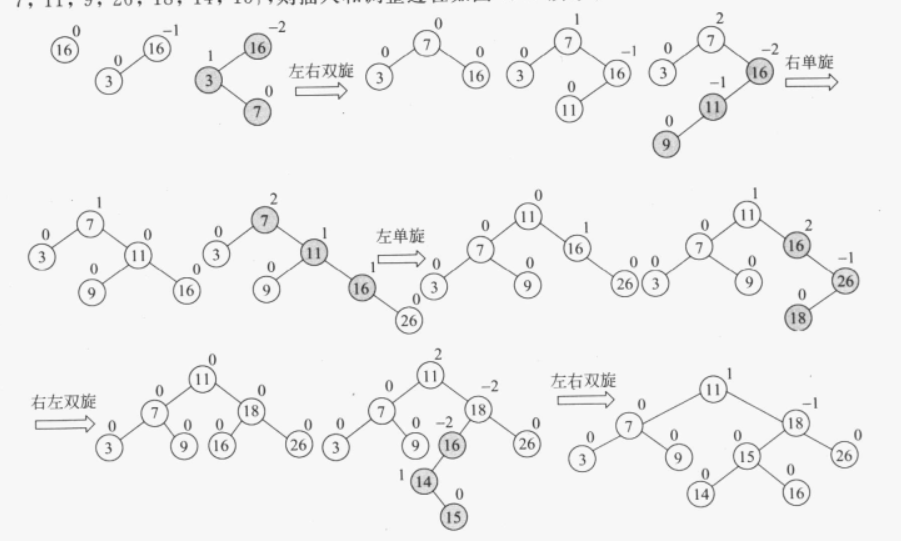


但当数据以升序或降序输入时，先插入的元素成为根结点，后续元素在同一侧，树状结构退化为线性结构；



平衡二叉树在每次插入新元素时会调整balance值，确保任一结点的balance为

-1，0，1，不至于失衡；



### 3.2.2 AVL树Insert函数

*template*<*class* T>

*bool* AVLTree<T>::insert(AVLNode<T>*\*&ptr*,T*&el*){

    AVLNode<T>\*pr=NULL,\*p=*ptr*,\*q;

*int* d;

    LinkedStack<AVLNode<T>\*>st;

    while(p!=NULL){

        if(*el*==p->data){

            std::cout<<"The Input key("<<p->data<") is already in the tree!"<<std::endl;

            return false;

        }

        pr=p;

        st.Push(pr);

        if(*el*<p->data){

            p=p->left;

        }

        else{

            p=p->right;

        }

    }

    p=new AVLNode<T>\*(*el*);

    if(p==NULL){

        std::cerr<<"存储空间不足!"<<std::endl;

        exit(1);

    }

    if(pr==NULL){*//空树*

*ptr*=p;

        return true;

    }

    if(*el*<pr->data){

        pr->left=p;

    }

    else{

        pr->right=p;

    }

    while(!st->IsEmpty()){

        st.Pop(pr);

        if(p==pr->left){

            pr->balance--;

        }

        else{

            pr->balance++;

        }

        if(pr->balance==0){

            break;

        }

        if(pr->balance==1||pr->balance==-1){

            p=pr;

        }

        else{

            d=(pr->balance<0)?-1:1;

            if(p->balance==d){

                if(d==-1){

                    RotateRight(pr);

                }

                else{

                    RotateLeft(pr);

                }

            }

            else{

                if(d==-1){

                    RotateLR(pr);

                }

                else{

                    RotateRL(pr);

                }

            }

            break;

        }

    }

    if(st.IsEmpty()==true){

*ptr*=pr;

    }

    else{

        st.getTop(q);

        if(q->dara>pr->data){

            q->left=pr;

        }

        else{

            q->right=pr;

        }

    }

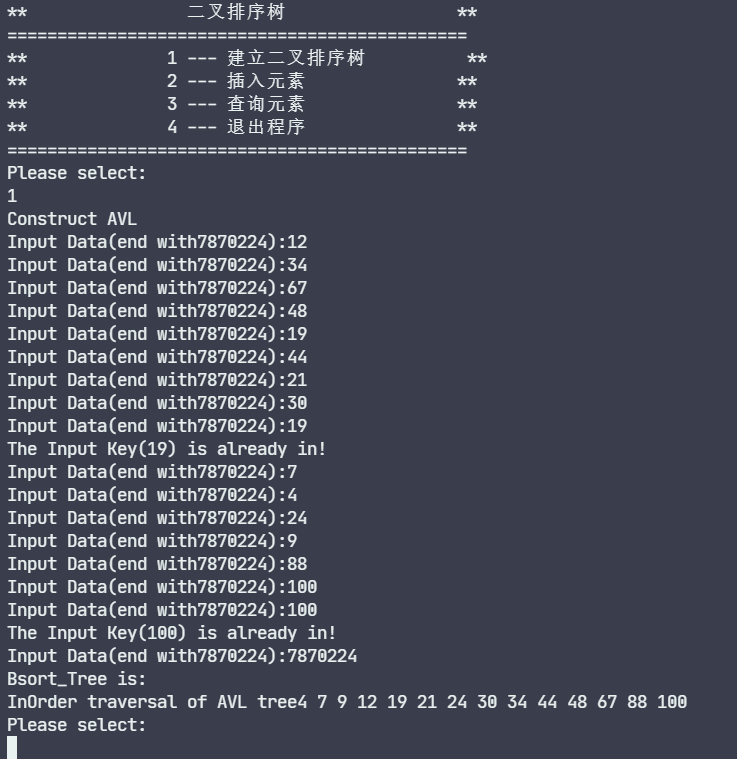
    return true;

}

# 4 项目测试

测试用例：12 34 67 48 19 44 21 30 19 7 4 24 9 88 100 100 RefValue

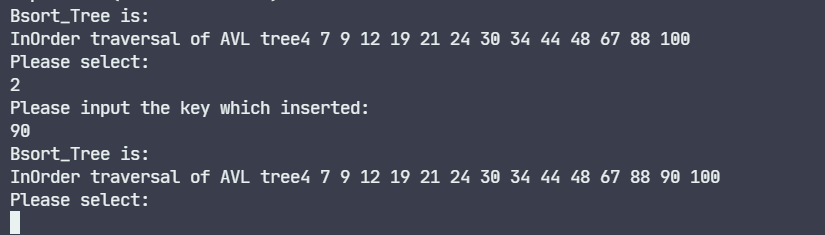
预期结果：4 7 9 12 19 21 24 30 34 44 48 67 88 100



## 4.1 插入元素

测试用例：2 90

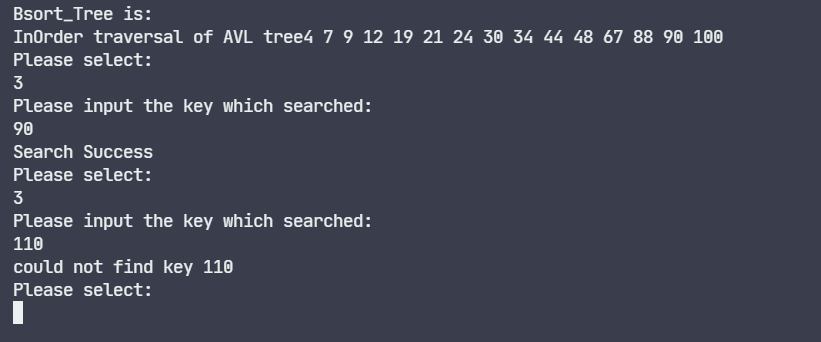
预期结果：4 7 9 12 19 21 24 30 34 44 48 67 88 90 100



## 4.2 查询元素

测试用例：3 90；3 11

预期结果：search success；110 not exit



## 4.3 退出

