项目文档

谈瑞

项目九：二叉排序树

专业班级：软件工程 4 班 学号：1452775 课目：数据结构课程设计
电话：18936361545 电子邮件：tanrui106@163.com 个人网站：http://guitoubing.top

目录

项目简介

项目概要

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。

项目功能及要求

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。

项目结构

本项目的基本架构是通过输入一串数字建立二叉搜索树，而后完成二叉搜索树的插入、删除、查找、替换、销毁等操作。二叉搜索树有一个很重要的特点，即对于任意一个结点（假设其左右子树都不为空），其左子树上所有结点的值都比右子树上的值要小，意即它是有一定顺序的存储方式。而且在中序遍历中，可得到其顺序排列。

本项目在基本要求外还实现了二叉树的前中后序的递归、非递归遍历方法，并在选择菜单提供了输出窗口，可随时输出二叉树。

项目类的实现

BSortTree 类

类成员	作用
BSortTree();	构造函数
~BSortTree();	析构函数
inline bool isEmpty();	判断二叉树是否为空
void createBSortTree();	创建二叉树的入口函数
bool setOperator();	选择对二叉树进行的操作
void createTreeFormIn();	从输入端输入二叉树数值，这里的输入的是个文件，而文件内是1000个随机数
bool insertTree();	往二叉树中添加结点
bool searchTree();	在二叉树中寻找结点
bool destroyTree();	销毁二叉树
bool eraseTree();	在二叉树中删除结点
void printTree();	输出二叉树
void printTree(int op);	递归输出二叉树，op 为选择输出方法（包括前序、中序、后序）
void printTreeNoRecursion(int op);	非递归输出二叉树，op 为选择输出方法（包括前序、中序、后序）
inline void printTreeInOrder (BSortTreeNode *_current)	中序遍历递归输出二叉树
inline void printTreePreOrder (BSortTreeNode *_current)	前序遍历递归输出二叉树
inline void printTreePostOrder (BSortTreeNode *_current)	后序遍历递归输出二叉树
inline void printTreeInOrderNoRecursion (BSortTreeNode *_current)	中序遍历非递归输出二叉树
inline void printTreePreOrderNoRecursion (BSortTreeNode *_current)	前序遍历非递归输出二叉树
inline void printTreePostOrderNoRecursion (BSortTreeNode *_current)	后序遍历非递归输出二叉树
BSortTreeNode *getParent (BSortTreeNode *_current)	获取某结点的父亲结点

public 成员

项目类的实现

protected 成员

类成员	作用
bool insertTree (BSortTreeNode *&_current, int _data)	往以指定节点为根节点的二叉树插入数据
BSortTreeNode *searchTree (BSortTreeNode *_current, int _data)	在以指定节点为根节点的二叉树中搜索数据
bool destroyTree (BSortTreeNode *_current)	销毁以指定节点为根节点的二叉树
bool eraseTree (BSortTreeNode *&_current)	删除指定节点
inline void InOrder (BSortTreeNode *_current, void(*visit)(BSortTreeNode *p))	在以指定节点为根节点的二叉树中中序递归遍历，遍历操作为 visit 函数指针所指函数
inline void PreOrder (BSortTreeNode *_current, void(*visit)(BSortTreeNode *p))	在以指定节点为根节点的二叉树中前序递归遍历，遍历操作为visit函数指针所指函数
inline void PostOrder (BSortTreeNode *_current, void(*visit)(BSortTreeNode *p))	在以指定节点为根节点的二叉树中后序递归遍历，遍历操作为 visit 函数指针所指函数
inline void PreOrderNoRecursion (BSortTreeNode *_current, void(*visit)(BSortTreeNode *p))	在以指定节点为根节点的二叉树中前序非递归遍历，遍历操作为 visit 函数指针所指函数
inline void InOrderNoRecursion (BSortTreeNode *_current, void(*visit)(BSortTreeNode *p))	在以指定节点为根节点的二叉树中中序非递归遍历，遍历操作为 visit 函数指针所指函数
inline void PostOrderNoRecursion (BSortTreeNode *_current, void(*visit)(BSortTreeNode *p))	在以指定节点为根节点的二叉树中后序非递归遍历，遍历操作为 visit 函数指针所指函数
BSortTreeNode* getFirstNodeInOrder (BSortTreeNode *_current)	获取某节点右子树中序遍历下的第一个节点
BSortTreeNode *getParent (BSortTreeNode *_current)	获取某结点的父亲结点
BSortTreeNode *root	二叉树根节点

主要代码分析

BSortTree

三个递归遍历函数

1. 前中后序递归遍历：

```
//下面三个递归遍历函数，传入的第二个参数为一个函数指针，即调用时需要制定行为函数，在输出函数中，使用的 lambda 函数表达式就是一种方法
    inline void InOrder(BSortTreeNode *_current,
void(*visit)(BSortTreeNode *p)) {
        if (_current != NULL) {
            InOrder(_current->left_child, visit);
            visit(_current);
            InOrder(_current->right_child, visit);
        }
    };
    inline void PreOrder(BSortTreeNode *_current,
void(*visit)(BSortTreeNode *p)) {
        if (_current != NULL) {
            visit(_current);
            PreOrder(_current->left_child, visit);
            PreOrder(_current->right_child, visit);
        }
    };
    inline void PostOrder(BSortTreeNode *_current,
void(*visit)(BSortTreeNode *p)) {
        if (_current != NULL) {
            PostOrder(_current->left_child, visit);
            PostOrder(_current->right_child, visit);
            visit(_current);
        }
    };
```

2. 前中后序非递归遍历

```
//非递归遍历二叉树一般都是需要在牺牲空间的条件下换取更快的运行时间，这里的前中后序非递归遍历都会用到辅助栈（实际上是 vector 容器）
    inline void PreOrderNoRecursion(BSortTreeNode *_current,
void(*visit)(BSortTreeNode *p)) {
        //对于前序遍历来说，访问任意一个结点时都是“一左到底”，同时将访问的结点的右子树根节点放入到辅助栈中（若空则跳过），一左到底结束后便将栈顶设为当前结点，循环采用刚才的操作，直到栈空
        vector<BSortTreeNode *> node_stack;//辅助栈
        BSortTreeNode *temp = _current;
        //temp 为当前结点
        while (temp != nullptr) {
            //终止条件：temp 为栈顶元素，当栈空 temp 亦为空，终止循环
            visit(temp);
```

```
            //遍历当前结点
            if (temp->right_child != nullptr) {
                node_stack.push_back(temp->right_child);
            }//右结点不为空时将其放到栈中
            if (temp->left_child != nullptr) {
                temp = temp->left_child;
            }
            else if (!node_stack.empty()) {
                temp = *--node_stack.end();
                node_stack.pop_back();
            }
            else {
                temp = nullptr;
            }//一左到底直到当前结点的左子树为空，接下来是否循环则看栈是否为空
        }
    }
    inline void InOrderNoRecursion(BSortTreeNode *_current, void(*visit)(BSortTreeNode *p)) {
        //对于中序遍历来说，由于是先访问左子树，用辅助栈将需要遍历的节点先存放至栈中，存放的顺序是先“一左到底”将左子树所有节点压入栈中，而后对栈顶元素先遍历弹出，而后对其右子树执行上述操作，如此循环即可
        vector<BSortTreeNode *> node_stack;
        BSortTreeNode *temp = _current;
        node_stack.push_back(temp);//先将根节点入栈
        do {
            if (temp == nullptr && !node_stack.empty()) {
                //若栈不为空且 temp 不为空，将 temp 置为栈顶元素，而后遍历 temp
                temp = *--node_stack.end();
                if (temp->right_child != nullptr) {
                    //如果 temp 的右节点不为空，则先遍历 temp，而后先将 temp 弹出栈，再将 temp 的右节点入栈
                    BSortTreeNode *p = temp->right_child;
                    visit(temp);
                    node_stack.pop_back();
                    temp = p;
                    node_stack.push_back(temp);
                    p = nullptr;
                }
                else {
                    //如果 temp 的右节点为空，则直接遍历 temp，而后将其弹出
                    visit(temp);
                    node_stack.pop_back();
                    temp = nullptr;
                }
            }
            else{
                while (temp->left_child != nullptr) {
```

```
                    temp = temp->left_child;
                    node_stack.push_back(temp);
                }
                temp = nullptr;
            }

        } while (temp != nullptr || !node_stack.empty());
    }
    inline void PostOrderNoRecursion(BSortTreeNode *_current,
void(*visit)(BSortTreeNode *p)) {
        struct BSortTreeNodeStruct {
            BSortTreeNode *current;
            int tag;
            BSortTreeNodeStruct(BSortTreeNode *p = nullptr) :
current(p), tag(LEFT) {};
        };
        BSortTreeNodeStruct node_struct;
        vector<BSortTreeNodeStruct> node_stack;
        BSortTreeNode *temp = _current;
        do {
            while (temp != nullptr) {
                node_struct.current = temp;
                node_struct.tag = LEFT;
                node_stack.push_back(node_struct);
                temp = temp->left_child;
            }
            int ifContinue = 1;
            while (ifContinue && !node_stack.empty()) {
                BSortTreeNodeStruct temp_struct = *--node_stack.end();
                node_stack.pop_back();
                temp = temp_struct.current;
                switch (temp_struct.tag) {
                    case LEFT: {
                        temp_struct.tag = RIGHT;
                        node_stack.push_back(temp_struct);
                        ifContinue = 0;
                        temp = temp->right_child;
                        break;
                    }
                    case RIGHT: {
                        visit(temp);
                        break;
                    }
                }
            }
        } while (!node_stack.empty());
    }
```

插入、删除函数

1. 插入函数

```
bool BSortTree::insertTree() {
    int _data;
    cin >> _data;
    return insertTree(root, _data);
}
bool BSortTree::insertTree(BSortTreeNode *&_current, int _data) {
      //插入节点是个递归操作，比当前节点小则递归插入至左子树，比当前节点大则递归插入至右子树，等于当前节点则不插入且报错，递归终止条件为当前节点为空
    if (_current == nullptr) {
        _current = new BSortTreeNode(_data);
        return true;
    }
    if (_current->data > _data)
        return insertTree(_current->left_child, _data);
    else if (_current->data < _data)
        return insertTree(_current->right_child, _data);
    else {
        cout << "数据" << _data << "已存在于二叉排序树中！";
        return false;
    }
}
```

2. 删除函数

```
bool BSortTree::eraseTree() {
    int _data;
    cin >> _data;
    BSortTreeNode *p = searchTree(root, _data);
    if (!p) {
        cout << "要删除的数据不存在于二叉树中。";
        return false;
    }
    else if (p->data == root->data && p->left_child == nullptr && p->right_child == nullptr) {
        delete root;
        root = nullptr;
        return true;
    }
    eraseTree(p);
}
```

主要代码分析

```
bool BSortTree::eraseTree(BSortTreeNode *&_node) {
    //删除结点时有几点需要注意：
    //当结点为叶子结点时可直接删除，当结点为非叶子结点时需要进行讨论，见以下分支
    if (_node->left_child == nullptr && _node->right_child == nullptr) {
        //当前结点为叶子结点
        if (_node == root) {
            //当前结点为二叉树根结点，则直接删除根结点，相当于销毁二叉树
            delete root;
            root = nullptr;
        }
        BSortTreeNode *p = getParent(_node);
        //获取要删除节点的父结点
        if (p->left_child != nullptr && p->left_child->data == _node->data)
        //当其父结点左子树不为空，说明该结点为其父结点的左子结点
            p->left_child = nullptr;
        else if (p->right_child != nullptr && p->right_child->data == _node->data)
        //当其父结点右子树不为空，说明该结点为其父结点的右子结点
            p->right_child = nullptr;
        delete _node;
    }
    else if (_node->left_child == nullptr) {
        //若其左子树为空，则直接将其右子树根节点移至该结点
        getParent(_node)->right_child = _node->right_child;
        delete _node;
    }
    else if (_node->right_child == nullptr) {
        //若其右子树为空，则直接将其左子树根节点移至该结点
        getParent(_node)->left_child = _node->left_child;
        delete _node;
    }
    else {
        //若上述情况均不满足，则说明其左右子树均不为空，则将其右子树下中序遍历第一个结点移至该位置，并对此结点递归进行删除操作
        BSortTreeNode *p = getFirstNodeInOrder(_node->right_child);
        int temp = p->data;
        eraseTree(p);
        _node->data = temp;
    }
    return true;
}
```

项目的一些拓展知识

此项目编写时比较中规中矩，按照所学知识来，这里主要提一个引用型指针的用法。

引用型指针

为什么要使用引用型指针？

当我们把一个指针作为参数传递给一个函数时，其更改的是指针所指的值（或对象），而函数中则会创建该指针的一个副本，意即将该指针指向其他值（或对象）时，并不会对原有指针所指值造成影响，而二叉树中很多地方涉及到更改指针所指值，此时使用指针参数就很难进行相关操作了。

使用方法

只需要将函数参数声明为引用型指针即可， 如 func(Type *&p)。其原理与一般参数引用类似，即给变量起了个别名。而这样声明的作用是使得在函数中操作函数外指针变量时，既能改变指针所指的值（或对象），又能改变指针本身。

本项目中很多地方就用到了引用型指针，如：

```
bool BSortTree::insertTree(BSortTreeNode *&_current, int _data) {…
}
bool BSortTree::eraseTree(BSortTreeNode *&_node) {…
}
```

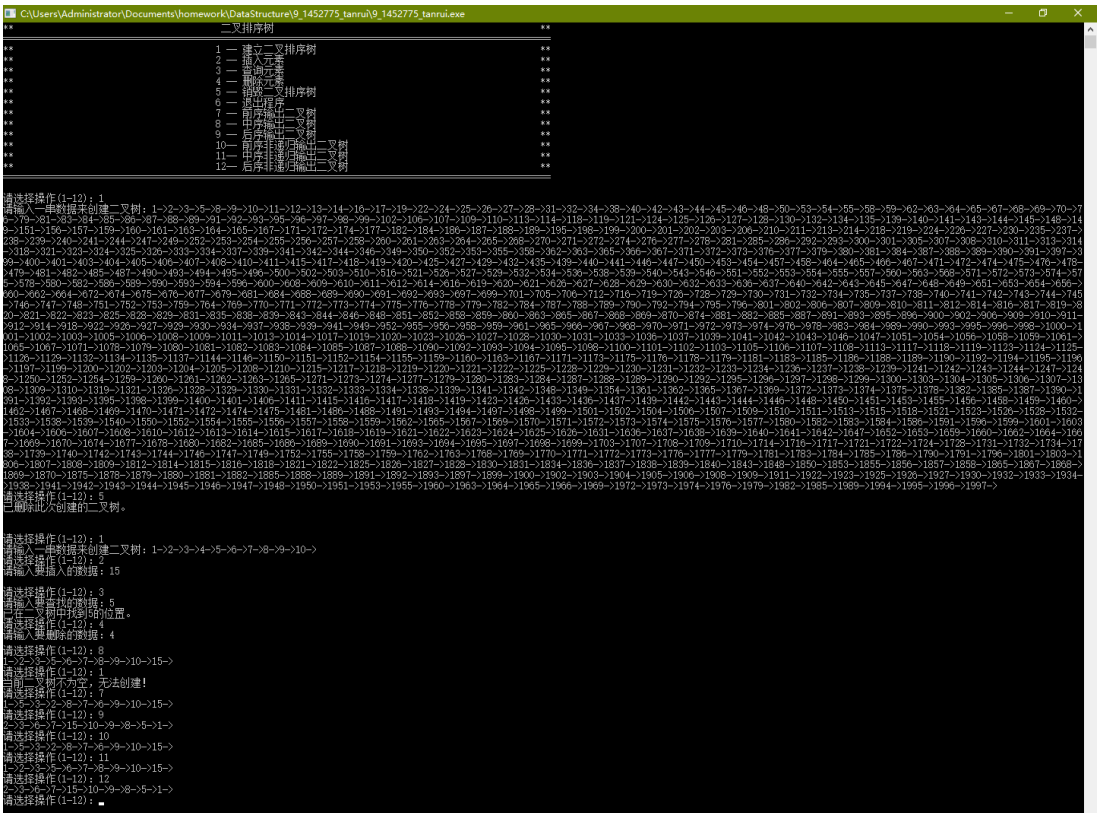
这使得操作指针变得简单的多。

指针的指针也能进行相似的操作

只需要将函数参数声明为引用型指针即可， 如 func(Type **p)。其能与指针的引用起到相同的作用。

运行测试

正常运行



错误检测