

“数据结构”

课程设计报告

**设计题目**  b+树

**姓 名**  姜舟

**学 号**  2022217361

**专 业**  计算机科学与技术

**班 级**

**完成日期**

课程设计成绩评定表

|  |  |  |  |
| --- | --- | --- | --- |
| **设计题目** |  | **成绩** |  |
| **内容简介** |  | | |
| **评语** | **教师签名：** | | |

1. **需求和规格说明**

编程实现B+树，功能要求：

① 插入关键字

② 删除关键字

③ 查找关键字

1. **二．设计：**
2. b+树的基本概念：

为了实现动态多层索引，通常采用 B-树 和 B+树。但是，用于索引的 B-树 存在缺陷，它的所有中间结点均存储的是数据指针（指向包含键值的磁盘文件块的指针），与该键值一起存储在B-树的结点中。这就会导致可以存储在 B-树中的结点目数极大地减少了，从而增加 B-树的层数，进而增加了记录的搜索时间。

B+树通过仅在树的叶子结点中存储数据指针而消除了上述缺陷。因此，B+树的叶结点的结构与 B-树的内部结点的结构完全不同。在这里应该注意，由于数据指针仅存在于叶子结点中，因此叶子结点必须将所有键值及其对应的数据指针存储到磁盘文件块以便访问。此外，叶子结点被链接磁盘的某个位置，以提供对记录的有序访问。因此，叶子结点形成第一级索引，而内部结点形成多层索引的其他层。叶子结点的某些关键字 key 也出现在内部结点中，充当控制搜索记录的媒介。

与 B-树不同，B+树中的结点存在两个阶（order）：对于阶 “a” 和 “ b”，一个用于内部结点，另一个用于外部（或叶）结点。

1. B+树中需要注意的点：

1.B+ 树非叶子节点上是不存储数据的，仅存储键值，而 B 树节点中不仅存储键值，也会存储数据。如果不存储数据，那么就会存储更多的键值，相应的树的阶数（节点的子节点树）就会更大，树就会更矮更胖。

2.另外，B+ 树的阶数是等于键值的数量的，如果我们的 B+ 树一个节点可以存储 1000 个键值，那么 3 层 B+ 树可以存储 1000×1000×1000=10 亿个数据。3.因为 B+ 树索引的所有数据均存储在叶子节点，而且数据是按照顺序排列的。那么 B+ 树使得范围查找，排序查找，分组查找以及去重查找变得异常简单。

（3）：原理解析图：

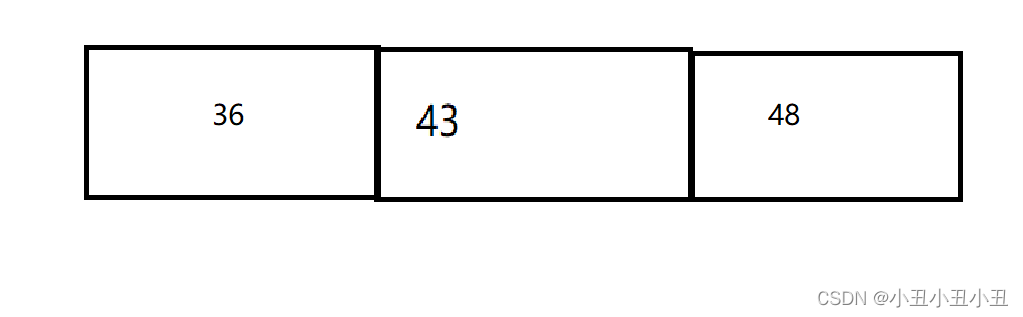
1.插入过程：

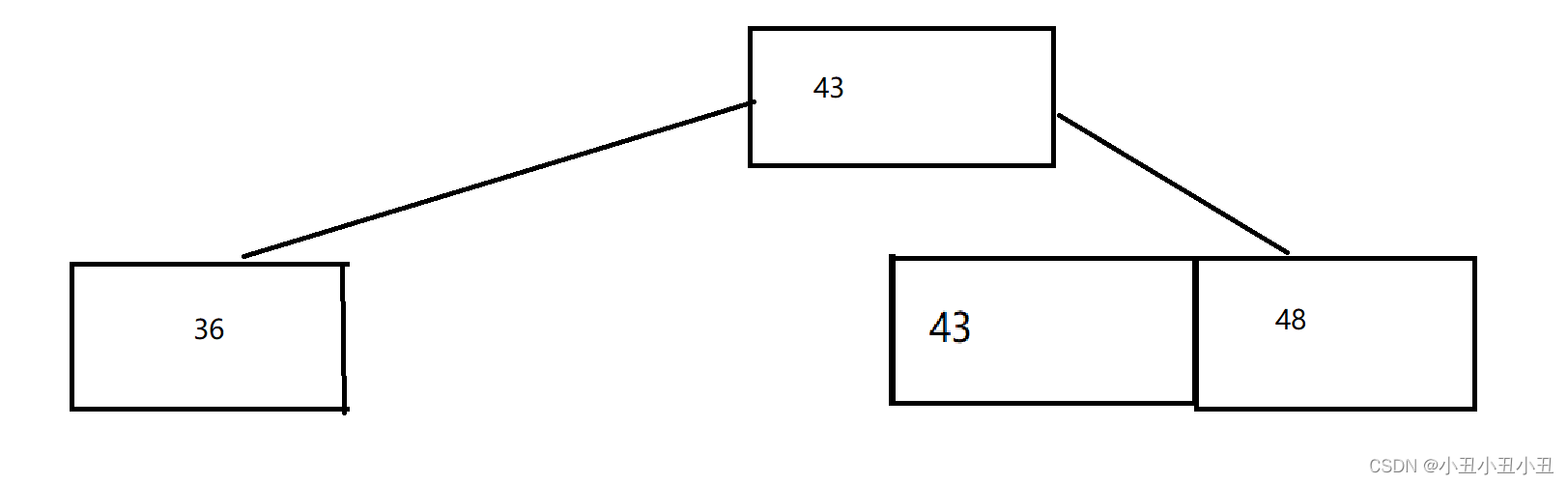
1. 非根节点至少有M/2-1个数据.
2. 在偶数情况下：假设M = 10，分裂时就会被分成4 1 5，因此非根节点有M / 2 - 1。  
   在奇数情况下：假设M = 9，分裂时就会被分成4 1 4，因此非根节点有M/2个。

3.当data个数等于M 的时候要分裂

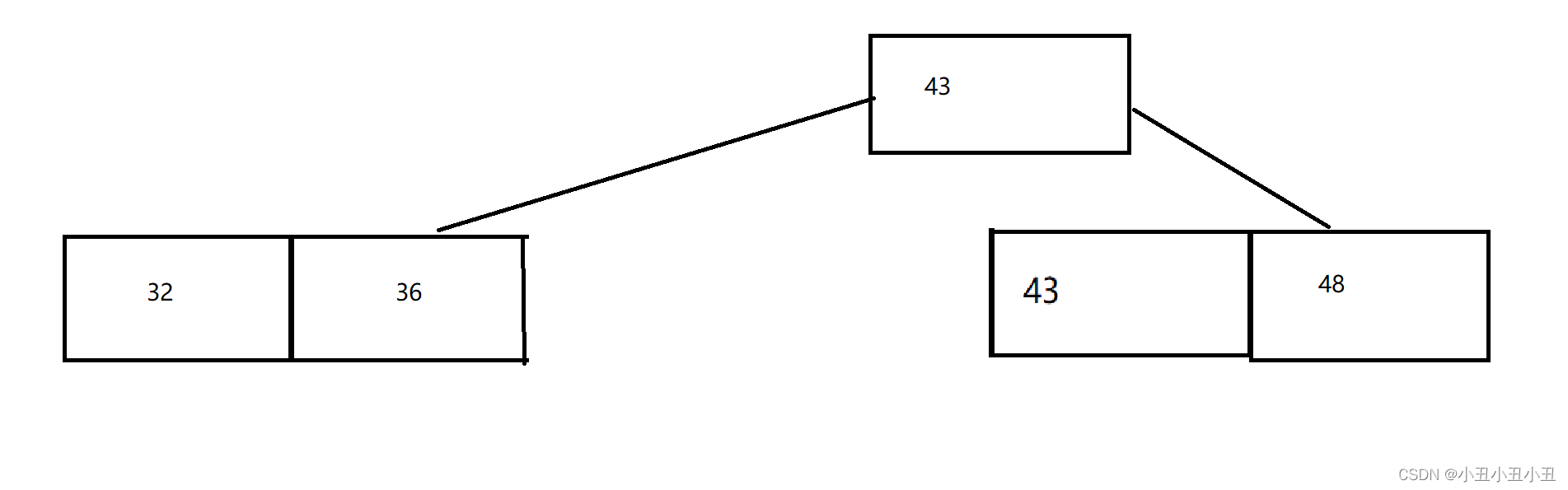
4.分裂的时候M / 2（下取整）个的右半部分要给兄弟节点，第M / 2 个数据要给根节点。给的时候按照插入排序的思想

举个例子：  
43，48，36，32，M = 3

插入43，48，36后  


由于大于3个关键字了，开始分裂。  
记住左边下取整，右边上取整，然后把第M/2（上取整）个key复制给父节点  


插入32



两相比较：

* B+树的叶子结点都是相链的，因此对整棵树的便利只需要一次线性遍历叶子结点即可。而且由于数据顺序排列并且相连，所以便于区间查找和搜索。而B树则需要进行每一层的递归遍历。相邻的元素可能在内存中不相邻，所以缓存命中性没有B+树好。

（4）算法设计：

算法共有两个大块：

1. ：如同树一般，将b+树的整体结构分为树的整体结构与结点两部分。并且在对b+树的叶子节点设计时，使用链表进行存储，在开始时，使用类模板完成对所有数据的兼容，下面，介绍每个部分中每个函数的作用：

1.B+树部分：

* BPlusTree(int order = 3) 是一个构造函数，用于创建一个新的BPlusTree对象。它接受一个可选的整数参数order，用于指定树的阶数。如果未指定，则默认为3。
* bool insert(T data) 是一个插入函数，用于在树中插入新数据。它接受一个T类型的参数data，表示要插入的数据。如果插入成功，则返回true；否则返回false。
* bool myDelete(T data) 是一个删除函数，用于从树中删除数据。它接受一个T类型的参数data，表示要删除的数据。如果删除成功，则返回true；否则返回false。
* bool find(T data) 是一个查找函数，用于在树中查找指定数据。它接受一个T类型的参数data，表示要查找的数据。如果找到，则返回true；否则返回false。
* bool setOrder(int n) 是一个设定阶数函数，用于重新设定树的阶数。它接受一个整数参数n，表示新的阶数。只有在树为空时才能进行此操作。如果操作成功，则返回true；否则返回false。
* bool isEmpty() 是一个判断树是否为空的函数。如果树为空，则返回true；否则返回false。
* void printData() 是一个打印函数，用于打印树中所有数据。
* void printTree() 是一个打印函数，用于按树的结构打印整个树。
* void findh(T data) 是一个查找函数，用于在树中查找指定数据并输出查找结果。

下面是私有成员和方法：

* private: 指定以下成员为私有成员，只能在类内部访问。
* BPTNode<T>\* Root 是一个指针，指向树的根节点。
* int order 是一个整数，表示树的阶数。
* int count 是一个整数，表示树中元素的数量。
* BPTNode<T>\* getHead() 是一个私有方法，用于返回叶节点的第一个节点。
* BPTNode<T>\* findLoc(T data) 是一个私有方法，用于查找指定数据应该插入或删除的节点位置。

1. B+树节点部分：

这是构建b+树左移动与移动的关键。先介绍下列应存在的函数：

* class keyNode 定义了一个名为keyNode的嵌套类，表示B+树节点中的关键字节点。
* keyNode() 是一个默认构造函数，用于创建一个新的keyNode对象。
* keyNode(const T data) 是一个构造函数，用于根据给定数据创建一个新的keyNode对象。它接受一个T类型的参数data，表示关键字节点中存储的数据。
* keyNode(const T data, BPTNode<T>\* left, BPTNode<T>\* right) 是一个构造函数，用于根据给定数据和左右子节点创建一个新的keyNode对象。它接受三个参数：一个T类型的参数data表示关键字节点中存储的数据；两个BPTNode<T>类型的指针参数left和right分别表示关键字节点的左右子节点。
* T data 是一个T类型的成员变量，表示关键字节点中存储的数据。
* BPTNode<T>\* left = nullptr 是一个指针，指向关键字节点的左子节点。对于叶节点，默认为nullptr。
* BPTNode<T>\* right = nullptr 是一个指针，指向关键字节点的右子节点。对于叶节点，默认为nullptr。
* vector<keyNode<T>> key = {} 是一个向量容器，存储B+树节点中的所有关键字节点。
* BPTNode<T>\* father = nullptr 是一个指针，指向B+树节点的父亲节点。
* BPTNode<T>\* next = nullptr 是一个指针，指向下一个叶子结点。
* BPTNode() 是一个默认构造函数，用于创建一个新的BPTNode对象。默认初始化为值为0的叶结点。
* BPTNode(const T data, int order = 3) 是一个构造函数，用于根据给定数据和阶数创建一个新的BPTNode对象。它接受两个参数：一个T类型的参数data表示要插入到新建B+树结点中的数据；一个可选的整数参数order表示新建结点的阶数。如果未指定，则默认为3。
* BPTNode(vector<T>& key, int order = 3) 是一个构造函数，用于根据给定数据数组和阶数创建一个新的内部结点。它接受两个参数：一个T类型向量引用参数key表示要插入到新建内部结点中的数据数组；一个可选整数参数order表示新建内部结点的阶数。如果未指定，则默认为3。

（3）B+树结点与树的结构体设计：

每个结点设计：

class keyNode

{

public:

keyNode() {};

keyNode(const T data) :data(data) {};//构造函数

keyNode(const T data, BPTNode<T>\* left, BPTNode<T>\* right) :data(data), left(left), right(right) {};//进行构造函数初始化，并储存每个结点的关键信息。

T data;//关键字数据

BPTNode<T>\* left = nullptr;//左孩子，叶节点左孩子默认为空

BPTNode<T>\* right = nullptr;//右孩子，叶节点右孩子默认为空

};

vector<keyNode<T>> key = {};//标识符更新为空

BPTNode<T>\* father = nullptr;//父节点更行为空

BPTNode<T>\* next = nullptr;//下一个接待你表示为空等待改写

在进行b+树结点的定义时，已经完成初始化，并且将初始化的值定义为零或空。

（5）实现注释：各项要求的实现程度、在完成基本要求的基础上还实现了什么功能？

* 1. 实现构造b+树算法，并额外增加打印b+树，便于观察。
  2. 使用vector动态数组完成对未知类别属性数据样例的分类，实现了功能，同时也满足兼容不同类型的数据集。
  3. 检验部分并不够完善，还存在部分情况并未包含在程序中。

1. **用户手册**

请严格按照以下要求进行操作：

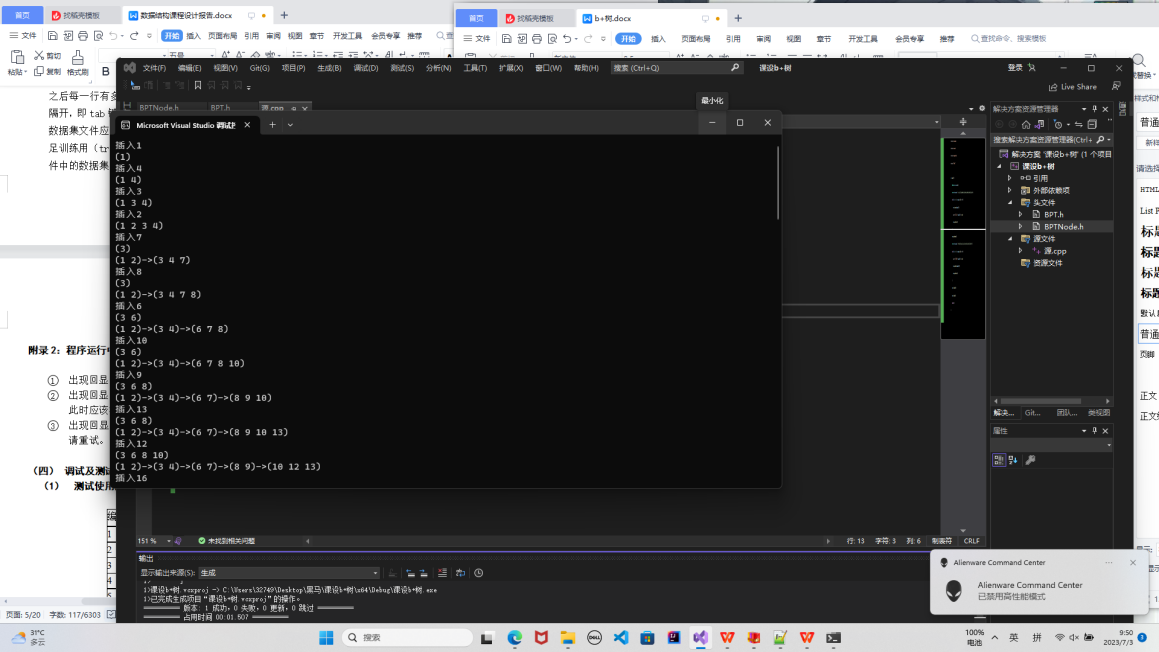
1. 将训练用数据集按照规定的方式进行整理。（参见用户手册附录1），插入数据请将数据集存到nums1中；
2. 将测试用数据集按照规定的方式进行整理。（参见用户手册附录1），

删除数据请将数据集存到nums2中；

1. 查找数据请自己手输入一个数值，并且通过控制台观察结果。
2. 运行程序，选择运行并等待显示效果。
3. 结果将会在控制台上显现。

用户手册附录：

**附录：程序运行中的问题及解决**

出现如下图所示的控制台程序时，说明已经可以运行。

1. **调试及测试**
2. **测试使用的数据集**

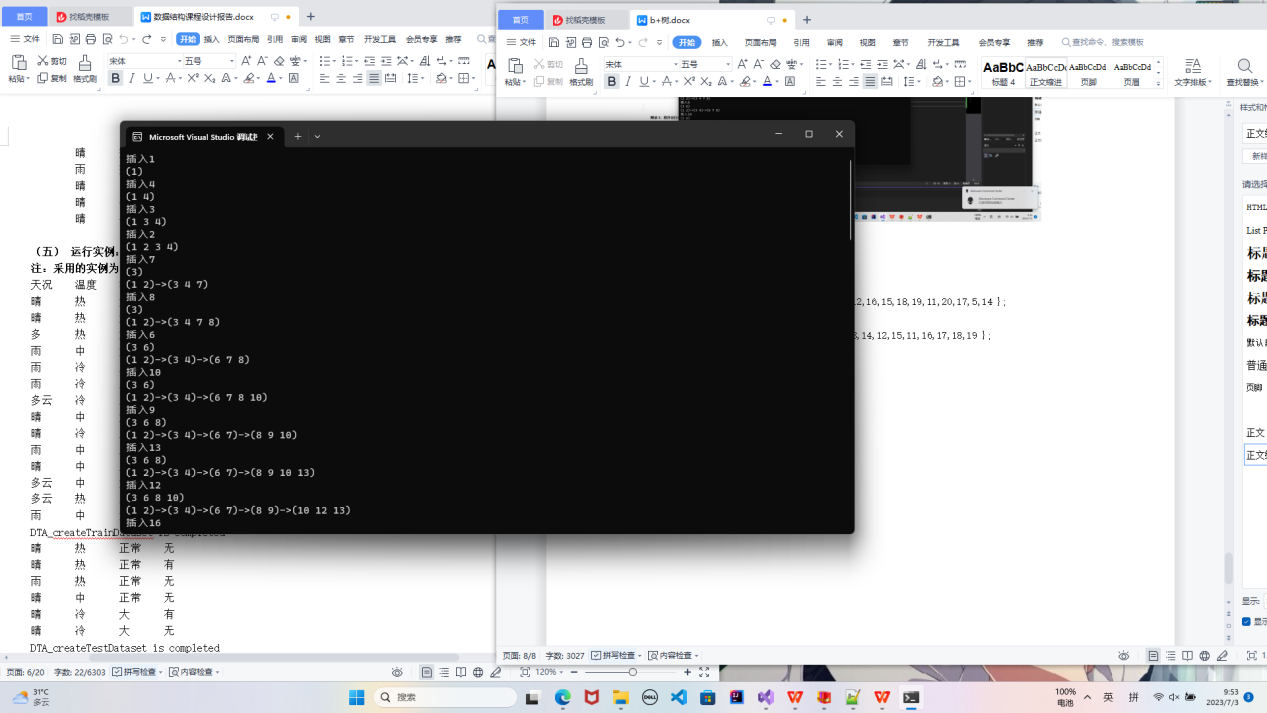
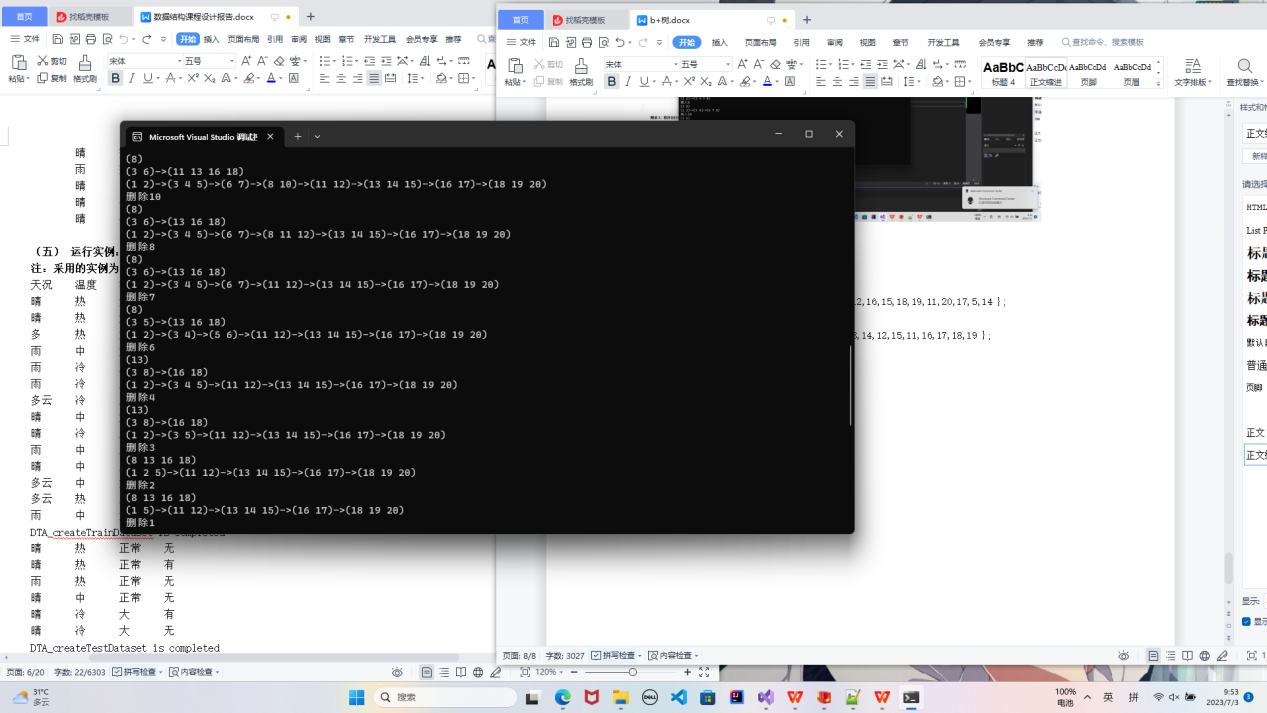
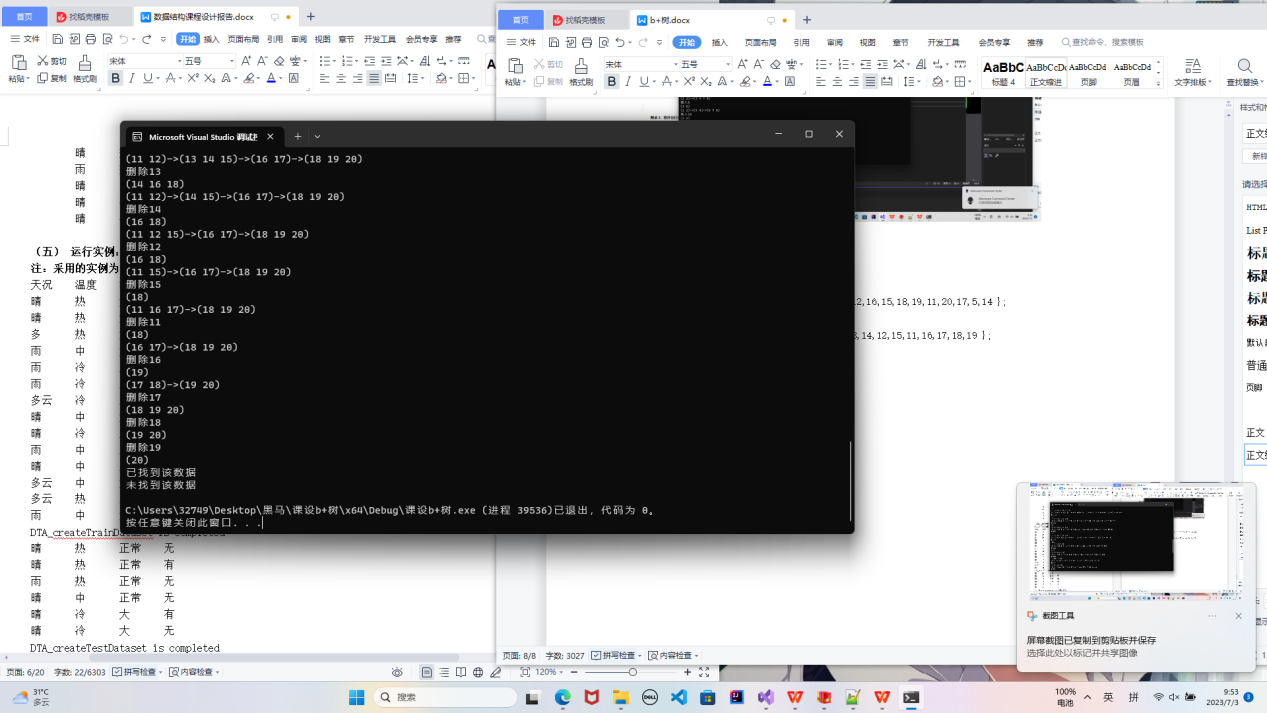
插入测试：nums1 = { 1,4,3,2,7,8,6,10,9,13,12,16,15,18,19,11,20,17,5,14 };

删除测试：nums2 = { 9,10,8,7,6,4,3,2,1,5,13,14,12,15,11,16,17,18,19 };

查找测试：20，0.

1. **运行实例：**

**注：采用的实例为上述中的数据。**

1. 插入：
2. 删除：
3. 查找：

**（六）仍需进一步改进**

关于程序会出现的问题，进行的反馈和判断并不足够。因为诸多原因，关于数据的判别并不够严谨。数据的引用并不足够，可以引用更多数据完成实验数据测试。

**（七）心得体会**

（可描述课程设计过程中出现的主要问题、原因及解决方法，课程设计的主要收获等）

**出现问题：**

**出现问题1：如何实现B+树算法？**

答：使用控制台输入数据完成对B+树的构建，并且将B+树的结点与数据分开经行分别的算法构建后运行。其中特别注意BPTNode与keyNode的区别：

keyNode 类的 data 成员表示关键字数据，而不是实际数据。在 B+ 树的内keyNode 和 BPTNode 是两个不同的类。keyNode 是一个嵌套类，它定义在 BPTNode 类中。keyNode 类表示 B+ 树节点中的关键字节点，它包含一个数据成员 data，用于存储关键字节点中的数据，以及两个指针成员 left 和 right，分别指向关键字节点的左右子节点。

而 BPTNode 类表示 B+ 树中的一个节点。它包含一个向量容器成员 key，用于存储 B+ 树节点中的所有关键字节点。它还包含一些其他成员和方法，用于实现 B+ 树的各种操作。

简而言之，keyNode 类表示 B+ 树节点中的一个关键字节点，而 BPTNode 类表示 B+ 树中的一个节点，它可以包含多个关键字节点。

部节点中，每个 keyNode 对象都包含一个关键字数据和两个指向子树的指针。这些关键字数据用于指示搜索路径，而不是存储实际数据。

只有在 B+ 树的叶子节点中，keyNode 对象的 data 成员才表示实际数据。在叶子节点中，每个 keyNode 对象都包含一个实际数据，而不包含指向子树的指针。

因此，在这段代码中，B+树的非叶子节点不存储实际数据，它们只存储关键字和指向子树的指针。

**出现问题2：如何实现对任何类型的数据都可以构建？**

答：使用templte模板对所有类型数据进行填写，大大增强了函数的泛用性，并且在使用后，代码变得更加简易美观。这样既没有修改决策树算法的核心，也提高了代码的泛用性。

**出现问题3：如何进行测试？**

答：这里采用了简单直接的输入数据进行测试的方法。在程序语法阶段，是通过大量的debug和警告提示进行修改，而在程序语法通过后，程序逐渐改为了选择输入数据进行逻辑的测试，当然也有简单的增加回显以提示项目的进程。

**项目收获：**

1. 清楚了解和基本掌握了b+树算法，对数据的处理和录入有了更加深入的了解
2. 对树、顺序表、链表等数据结构有了更极深刻的了解，同时也学习了vector动态数组的用法，对不同的结构应该采用何种算法有了一些认识。
3. 对程序的debug和修改优化，还有如何书写注释，进行了学习，在之后的程序设计过程中会更加有经验。
4. 对C++函数式的设计更加熟悉，对C++语言的一些语法常识有了了解。
5. 完善了从问题出发，再到如何解决问题的具体流程，丰富了设计的经验。

**（八）对课程设计的建议**

希望可以完善一些时间安排，题目方面的描述很棒，但是设计的具体流程需要自己安排。这将耗费大量时间。

**（九）附录⎯⎯源程序**

也可以参见实验附录程序。

1.B+树结点头文件

#pragma once

#include <vector>

#include <algorithm>

using namespace std;

template <class T>

class BPTNode

{

public:

template<class T>//关键字结点类

class keyNode

{

public:

keyNode() {};

keyNode(const T data) :data(data) {};//构造函数

keyNode(const T data, BPTNode<T>\* left, BPTNode<T>\* right) :data(data), left(left), right(right) {};

T data;//关键字数据

BPTNode<T>\* left = nullptr;//左孩子，叶节点左孩子默认为空

BPTNode<T>\* right = nullptr;//右孩子，叶节点右孩子默认为空

};

vector<keyNode<T>> key = {};

BPTNode<T>\* father = nullptr;

BPTNode<T>\* next = nullptr;

BPTNode();//默认构造函数，默认初始化为值为0的叶结点

BPTNode(const T data, int order = 3);//根据单一数据和阶数初始化B+树结点

BPTNode(vector<T>& key, int order = 3);//根据数据数组和阶数初始化B+树结点

bool find(const T keyValue);//在当前结点查找对应数据是否存在

bool insert(const T keyValue, BPTNode<T>\*& Root);//叶结点专用，插入键值

bool myDelete(const T keyValue, BPTNode<T>\*& Root);//叶结点专用，删除键值

bool isEmpty();//结点是否为空

bool setorder(int order);//修改阶数，只有修改后键值数仍小于阶数才可以修改

private:

int order = 3;

bool insert(keyNode<T>\* newKey, BPTNode<T>\*& Root);//内部结点专用，插入键值

bool myDelete(BPTNode<T>\* now, BPTNode<T>\*& Root);//内部结点专用，删除键值

BPTNode<T>\* getLeftBro(BPTNode<T>\* now,BPTNode<T>\* Root);//找到当前结点的左兄弟结点

BPTNode<T>\* getHead(BPTNode<T>\* Root);//找到当前结点所在树的最小结点

};

template<class T>

inline BPTNode<T>::BPTNode() {

}

template<class T>

inline BPTNode<T>::BPTNode(const T data, int order)

{

key = {};

this->order = order;

keyNode<T> tmp(data);

key.push\_back(tmp);

}

template<class T>

inline BPTNode<T>::BPTNode(vector<T>& key,int order)//新建内部结点

{

this->order = order;

sort(key.begin(), key.end());

for (auto p : key) {

keyNode<T> tmp(p);

this->key.push\_back(tmp);

}

}

template<class T>

inline bool BPTNode<T>::find(const T keyValue)

{

for (auto& p : key)

if (p.data == keyValue)

return true;

return false;

}

template<class T>

inline bool BPTNode<T>::insert(const T keyValue, BPTNode<T>\*& Root)//叶节点插入数据

{

if (find(keyValue)) return false;//如果结点中已经存在该数据，插入失败

if (key.size() == 0)//结点为空直接插入

key.push\_back(keyNode<T>(keyValue));

else if (keyValue > (\*key.rbegin()).data)//插入数据比已有数据最大值还大，则直接插入最后位置

key.push\_back(keyNode<T>(keyValue));

else {//找到应该插入位置并插入数据

int len = key.size();

int i = 0;

for (i = 0; i < len; i++)

if (keyValue < key[i].data) break;

key.emplace(key.begin() + i, keyNode<T>(keyValue));

}

if (key.size() < order) {//空间充足,插入后仍符合定义

return true;

}

else {//叶节点空间不够，插入后分裂成两个叶节点，并把中间的键值进位到父结点中

int m = order / 2;

keyNode<T>\* newKey = new keyNode<T>(key[m]);//复制中间位置的键值

BPTNode<T>\* newBPTNode = new BPTNode<T>;//创建新的叶结点

newBPTNode->father = father;

newBPTNode->setorder(order);

newKey->left = this;

newKey->right = newBPTNode;

newBPTNode->next = this->next;

this->next = newBPTNode;

for (int i = m; i < key.size(); i++) {//后一半的键值复制到新的叶节点中

keyNode<T>\* tmp = new keyNode<T>;

tmp->data = key[i].data;

tmp->left = key[i].left;

tmp->right = key[i].right;

newBPTNode->key.push\_back(\*tmp);

}

key.resize(m);//删除当前叶节点后半部分键值

if (father) {//若父节点存在则把中间位置的键值插入父结点

father->insert(newKey, Root);

return true;

}

else {//若父节点不存在则创造一个新的内部节点作为父节点

BPTNode<T>\* tmp = new BPTNode<T>;//创建新的叶结点

tmp->setorder(order);

this->father = tmp;

newBPTNode->father = tmp;

tmp->key.push\_back(\*newKey);

Root = tmp;

return true;

}

}

return true;

}

template<class T>

inline bool BPTNode<T>::myDelete(const T keyValue, BPTNode<T>\*& Root)//结点中删除一个键值

{

if (!find(keyValue))//找不到要删除的键值

return false;

for (int i = 0; i < key.size(); i++) {

if (key[i].data == keyValue) {

key.erase(key.begin() + i);

break;

}

}

int m = (order + 1) / 2 - 1;

if (father == nullptr) return true;//如果是根结点，直接删除即可

if (key.size() >= m) {//删除后结点满足定义，直接删除并调整父节点中对应关键字即可

for (auto& p : father->key) {

if (p.right == this) {

p.data = key[0].data;

break;

}

}

return true;

}

else {//如果兄弟结点有富裕，则从兄弟结点借一个，同时把父节点中对应关键字更新为借的关键字

BPTNode<T>\* leftBro = nullptr;

BPTNode<T>\* rightBro = nullptr;

int leftLoc = 0, rightLoc = 0;

for (int i = 0; i < father->key.size(); i++) {

if (father->key[i].left == this) {

rightBro = father->key[i].right;

rightLoc = i;

}

if (father->key[i].right == this) {

leftBro = father->key[i].left;

leftLoc = i;

}

}

if (leftBro != nullptr && leftBro->key.size() > m) {//左兄弟结点够借

keyNode<T> newKey = leftBro->key[leftBro->key.size() - 1];

key.emplace(key.begin(), newKey);

leftBro->key.erase(leftBro->key.end() - 1);

father->key[leftLoc].data = key[0].data;//更新父节点中对应关键字

return true;

}

if (rightBro != nullptr && rightBro->key.size() > m) {//右兄弟结点够借

keyNode<T> newKey = rightBro->key[0];

key.push\_back(newKey);

rightBro->key.erase(rightBro->key.begin());

father->key[rightLoc].data = rightBro->key[0].data;//更新父节点中对应关键字

return true;

}

if (leftBro != nullptr) {//如果左右兄弟都不够借且左兄弟存在，则和左兄弟合并，并删除父节点中对应关键字

for (int i = 0; i < leftBro->key.size(); i++) {//左兄弟数据复制到当前结点前面

key.emplace(key.begin() + i, leftBro->key[i]);

}

BPTNode<T>\* cousin = getLeftBro(leftBro,Root);

if (cousin != nullptr) cousin->next = this;

if (leftLoc > 0)

father->key[leftLoc - 1].right = this;

father->key.erase(father->key.begin() + leftLoc);//删除父节点中对应关键字

delete leftBro;

return this->myDelete(father, Root);

}

else if (rightBro != nullptr) {//和右兄弟合并，并删除父节点中对应关键字

for (int i = 0; i < rightBro->key.size(); i++) {//右兄弟数据复制到当前结点末尾

key.push\_back(rightBro->key[i]);

}

next = rightBro->next;

if (rightLoc + 1 < father->key.size())

father->key[rightLoc + 1].left = this;

father->key.erase(father->key.begin() + rightLoc);//删除父节点中对应关键字

delete rightBro;

return this->myDelete(father, Root);

}

}

return false;

}

template<class T>

inline bool BPTNode<T>::isEmpty()

{

if (key.size() == 0) return true;

return false;

}

template<class T>

inline bool BPTNode<T>::setorder(int order)

{

if (key.size() >= order)

return false;

this->order = order;

return true;

}

template<class T>

inline bool BPTNode<T>::insert(keyNode<T>\* newKey, BPTNode<T>\*& Root)//内部结点插入数据

{

if (newKey->data > (\*key.rbegin()).data)//插入数据比已有数据最大值还大，则直接插入最后位置

key.push\_back(\*newKey);

else {//找到应该插入位置并插入数据

int len = key.size();

int i = 0;

for (i = 0; i < len; i++)

if (newKey->data < key[i].data) break;

if (i != 0) key[i - 1].right = newKey->left;

key[i].left = newKey->right;

key.emplace(key.begin() + i, \*newKey);

}

if (key.size() < order) {//空间充足,插入后仍符合定义

return true;

}

else {//内部节点空间不够，插入后分裂成两个内部节点，并把中间的键值进位到父结点中

int m = (order - 1) / 2;

keyNode<T>\* newKey = new keyNode<T>(key[m]);//复制中间位置的键值

BPTNode<T>\* newBPTNode = new BPTNode<T>;//创建新的内部结点

newBPTNode->father = father;

newBPTNode->setorder(order);

newKey->left = this;

newKey->right = newBPTNode;

key[m].right->father = newBPTNode;

for (int i = m + 1; i < key.size(); i++) {//后一半的键值复制到新的叶节点中

if (key[i].right) key[i].right->father = newBPTNode;

keyNode<T>\* tmp = new keyNode<T>;

tmp->data = key[i].data;

tmp->left = key[i].left;

tmp->right = key[i].right;

newBPTNode->key.push\_back(\*tmp);

}

key.resize(m);//删除当前叶节点后半部分键值

if (father) {//若父节点存在则把中间位置的键值插入父结点

father->insert(newKey, Root);

return true;

}

else {//若父节点不存在则创造一个新的内部节点作为父节点

BPTNode<T>\* tmp = new BPTNode<T>;//创建新的内部结点

tmp->setorder(order);

this->father = tmp;

newBPTNode->father = tmp;

tmp->key.push\_back(\*newKey);

Root = tmp;

return true;

}

}

return false;

}

template<class T>

inline bool BPTNode<T>::myDelete(BPTNode<T>\* now, BPTNode<T>\*& Root)

{

int m = (order + 1) / 2 - 1;

if (Root == now) {//当前结点是根结点，若当前结点不为空直接返回，为空修改根节点后返回

if (now->key.size() == 0) {

this->father = nullptr;

Root = this;

delete now;

return true;

}

return true;

}

if (now->key.size() >= m) return true;//当前结点不是根结点，删除后符合定义则直接返回

BPTNode<T>\* dad = now->father;

BPTNode<T>\* leftBro = nullptr;

BPTNode<T>\* rightBro = nullptr;

int leftLoc = 0, rightLoc = 0;

for (int i = 0; i < dad->key.size(); i++) {

if (dad->key[i].right == now) {

leftLoc = i;

leftBro = dad->key[i].left;

}

if (dad->key[i].left == now) {

rightLoc = i;

rightBro = dad->key[i].right;

}

}

if (leftBro != nullptr && leftBro->key.size() > m) {//如果左兄弟结点够借

keyNode<T> newKey(dad->key[leftLoc].data);

newKey.left = leftBro->key[leftBro->key.size() - 1].right;

if (now->key.size() != 0)

newKey.right = now->key[0].left;

now->key.push\_back(newKey);

dad->key[leftLoc].data = leftBro->key[leftBro->key.size() - 1].data;

leftBro->key.erase(leftBro->key.end() - 1);

return true;

}

if (rightBro != nullptr && rightBro->key.size() > m) {//如果右兄弟结点够借

keyNode<T> newKey(dad->key[rightLoc].data);

newKey.right = rightBro->key[0].left;

if (now->key.size() != 0)

newKey.left = now->key[now->key.size() - 1].right;

now->key.push\_back(newKey);

dad->key[rightLoc].data = rightBro->key[0].data;

rightBro->key.erase(rightBro->key.begin());

return true;

}

if (leftBro != nullptr) {//如果左右兄弟都不够借且左兄弟存在

BPTNode<T>\* dad = now->father;

keyNode<T> newKey;

newKey.data = dad->key[leftLoc].data;

newKey.left = leftBro->key[leftBro->key.size() - 1].right;

if (now->key.size() != 0)

newKey.right = now->key[0].left;

dad->key.erase(dad->key.begin() + leftLoc);//删除父节点中对应关键字

now->key.emplace(now->key.begin(), newKey);//父结点中对应关键字下移到当前结点

for (int i = 0; i < leftBro->key.size(); i++) {//左兄弟数据复制到当前结点前面

now->key.emplace(now->key.begin() + i, leftBro->key[i]);

leftBro->key[i].left->father = now;

leftBro->key[i].right->father = now;

}

delete leftBro;

return now->myDelete(dad, Root);

}

if(rightBro!=nullptr){//如果左右兄弟都不够借且右兄弟存在

BPTNode<T>\* dad = now->father;

keyNode<T> newKey;

newKey.data = dad->key[rightLoc].data;

newKey.right = rightBro->key[0].left;

if (now->key.size() != 0)

newKey.left = now->key[now->key.size() - 1].right;

dad->key.erase(dad->key.begin() + rightLoc);//删除父节点中对应关键字

now->key.push\_back(newKey);//父结点中对应关键字下移到当前结点

for (int i = 0; i < rightBro->key.size(); i++) {//右兄弟数据复制到当前结点后面

now->key.push\_back(rightBro->key[i]);

rightBro->key[i].left->father = now;

rightBro->key[i].right->father = now;

}

delete rightBro;

return now->myDelete(dad, Root);

}

return false;

}

template<class T>

inline BPTNode<T>\* BPTNode<T>::getLeftBro(BPTNode<T>\* now,BPTNode<T>\* Root) {

BPTNode<T>\* tmp = getHead(Root);

while (tmp->next) {

if (tmp->next == now)

return tmp;

tmp = tmp->next;

}

return nullptr;

}

template<class T>

inline BPTNode<T>\* BPTNode<T>::getHead(BPTNode<T>\* Root) {

BPTNode<T>\* tmp = Root;

while (tmp->key.size() > 0 && tmp->key[0].left)

tmp = tmp->key[0].left;

return tmp;

}

2.B+树头文件

#include <queue>

#include "BPTNode.h"

template<class T>

class BPlusTree {

public:

BPlusTree(int order = 3);//构造函数

bool insert(T data);//插入函数

bool myDelete(T data);//删除函数

bool find(T data);//查找数据，返回是否找到

bool setOrder(int n);//设定阶数

bool isEmpty();//返回B+树是否为空

void printData();//打印树中所有数据

void printTree();//按树的结构打印整个树

private:

BPTNode<T>\* Root;

int order;//阶数，即子树最大数量,关键字最大数量加1，默认为3

int count;//已有元素数量，初始为0

BPTNode<T>\* getHead();//返回叶节点的第一个节点

BPTNode<T>\* findLoc(T data);//查找函数，返回数据所在位置或应该插入位置

};

template<class T>

inline BPlusTree<T>::BPlusTree(int order)//根据阶数初始化B+树

{

this->order = order;

count = 0;

Root = new BPTNode<T>;

Root->father = nullptr;

Root->setorder(order);

}

template<class T>

bool BPlusTree<T>::insert(T data)//找到应该插入数据的节点并插入数据，若数据已经存在则插入失败

{

BPTNode<T>\* tmp = findLoc(data);

bool flag = tmp->insert(data,Root);

if (flag) count++;

if (!flag)std::cout << "插入失败，数据" << data << "己存在！" << endl;

return flag;

}

template<class T>

bool BPlusTree<T>::myDelete(T data)//找到应该删除数据的节点并删除数据，若数据不存在则删除失败

{

BPTNode<T>\* tmp = findLoc(data);

bool flag = tmp->myDelete(data,Root);

if (flag) count--;

if (!flag)std::cout << "删除失败，找不到数据" << data << "!" << endl;

return flag;

}

template<class T>

inline bool BPlusTree<T>::find(T data)//在树中查找指定数据，返回查找结果

{

BPTNode<T>\* tmp = findLoc(data);

return tmp->find(data);

}

template<class T>

bool BPlusTree<T>::setOrder(int n)//重新设定阶数，只有在树为空树时可以进行此操作

{

if (!isEmpty())

return false;

order = n;

return true;

}

template<class T>

bool BPlusTree<T>::isEmpty()//返回树是否为空

{

if (count)

return false;

return true;

}

template<class T>

inline void BPlusTree<T>::printData()//输出树中所有数据

{

if (Root->isEmpty()) {

cout << "Tree is empty!\n";

return;

}

BPTNode<T>\* tmp = getHead();

while (tmp && !tmp->isEmpty()) {

int len = tmp->key.size();

std::cout << "(";

for (int i = 0; i < len; i++) {

std::cout << tmp->key[i].data << " ";

if (i == len - 1)

tmp = tmp->next;

}

std::cout << "\b)->";

}

std::cout << "\b\b";

std::cout << " " << endl;

}

template<class T>

inline void BPlusTree<T>::printTree()//输出整个树的结构

{

if (Root->isEmpty()) {

cout << "Tree is empty!\n";

return;

}

queue<BPTNode<T>\*> qu;

qu.push(Root);

while (!qu.empty()) {

int len = qu.size();

for (int i = 0; i < len; i++) {

BPTNode<T>\* tmp = qu.front();

qu.pop();

for (int j = 0; j < tmp->key.size(); j++) {

if (tmp->key[j].left)

qu.push(tmp->key[j].left);

if (j == tmp->key.size() - 1 && tmp->key[j].right)

qu.push(tmp->key[j].right);

}

std::cout << "(";

for (int j = 0; j < tmp->key.size(); j++) {

std::cout << tmp->key[j].data << " ";

}

std::cout << "\b)->";

}

std::cout << "\b\b " << endl;

}

}

template<class T>

inline BPTNode<T>\* BPlusTree<T>::getHead()//找到树中最小的数据所在结点

{

BPTNode<T>\* tmp = Root;

while (tmp->key.size() > 0 && tmp->key[0].left)

tmp = tmp->key[0].left;

return tmp;

}

template<class T>

BPTNode<T>\* BPlusTree<T>::findLoc(T data)//找到指定数据应该插入的结点位置

{

if (count == 0) return Root;

BPTNode<T>\* tmp = Root;

if (Root->key[0].left == nullptr) return tmp;

while (tmp->key[0].left != nullptr) {

for (int i = 0; i < tmp->key.size(); i++) {

if (data < tmp->key[i].data) {

tmp = tmp->key[i].left;

break;

}

if (i == tmp->key.size() - 1) {

tmp = tmp->key[i].right;

break;

}

}

}

return tmp;

}

1. 源文件

#include <iostream>

#include <vector>

#include <algorithm>

#include "BPT.h"

int main() {

BPlusTree<int> tree(5);

vector<int> nums1 = { 1,4,3,2,7,8,6,10,9,13,12,16,15,18,19,11,20,17,5,14 };

for (int i = 0; i < nums1.size(); i++) {

tree.insert(nums1[i]);

cout << "插入" << nums1[i] << endl;

tree.printTree();

}

tree.printData();

vector<int> nums2 = { 9,10,8,7,6,4,3,2,1,5,13,14,12,15,11,16,17,18,19 };

for (int i = 0; i < nums2.size(); i++) {

cout << "删除" << nums2[i] << endl;

tree.myDelete(nums2[i]);

tree.printTree();

}

tree.findh(20);

tree.findh(0);

return 0;

}