**数据结构实习作业报告**

**211294029 李梦麟、211294043 江炫烨**

**0 摘要**

本次实习作业希望利用C++将数据结构课程中所学知识应用于小型数据库的构建中，通过实践加深对数据逻辑结构与其所映射的物理结构的理解，根据需求情境与数据结构特点，采用不同类型的数据结构解决问题。实习作业中主要应用B+树、字符串、数组等数据结构来实现关系型数据库中不同功能的类的定义与具体操作，包括数据存储、顺序访问、插入、删除、查找、范围查询、SQL语句解析等操作。由于尚未实现导入外部数据的功能，本次实习将具体的业务定位为小型企业的员工管理。

**1 实践背景**

在信息资源管理发展的初级阶段，数据存储和数据管理通过传统的文件系统完成，随着信息技术发展，数据生产与日俱增，计算机开始应用于数据管理，数据库技术作为现代化的信息管理技术，能够很好地应对当下数据量大、信息繁杂等问题，能够提高数据管理的效率和准确性、降低数据冗余，节约存储空间，同时提供多种查询方式，方便用户进行数据筛选和分析，增强了数据的安全性和可靠性。

数据库是结构化信息或数据的有序集合，一般以电子形式存储在计算机系统中。通常由数据库管理系统(DBMS)来控制。为提高数据处理与查询效率，当下最常见的数据库通常以行和列的形式将数据存储在一系列的表中，以支持用户便捷访问、管理、修改、更新、控制和组织数据。目前大多数数据库使用结构化查询语言(SQL)来查询、操作和定义数据，用户只需在高层数据结构上进行数据处理，无须用户指定数据的存取方法,也无需用户了解具体的数据存储方式,就可以使用SQL对不同数据库进行数据访问控制。数据库按其结构可分为多种类型，包含层次数据库、网状数据库、关系数据库、图数据库等，其中在关系型数据库里，项被组织为一组具有列和行的表，实现结构化数据的存储，使其访问更加便捷灵活。

关系型数据库广泛应用于企业的人员、交易等管理系统上。于企业而言，建立高效的信息管理系统是其适应社会发展形势的必经之路，其中面向小型企业的员工管理系统，要求整体性强、操作方便、流程简洁，利用关系型数据库进行数据存储以及通过SQL进行信息的录入、更新、删除、查找即可较好地满足这一要求，因此本次实习旨在实现关系型数据库基本架构搭建的基础上，进一步实现员工信息基本的增删改查等功能。

**2 实践内容**

本次实习作业采用C++为主要编程语言，C++17及以上的编译环境，利用C++面向对象的特性构建了BPlusTree（B+树类）、Table（实现数据存储和操作的核心类）、Colunm（存储数据库关系模式的类）、Row（存储数据行）、Page（对应B+树的节点）、DB（进行输入交互与语句解析的虚拟机）等类来实现数据库的功能。接下来将详细介绍各个类的构建思路：

1. **BPlusTree类**

BPlusTree类主要实现了B+树的定义及相关操作，B+树是一种常用的自平衡搜索树，广泛应用于数据库和文件系统等领域。它是基于B树的一种变体，相比于B树，B+树在内部节点不存储数据，只存储键值（或范围），而所有的数据都存储在叶子节点中。这种设计使得B+树具有更高的查询效率和更好的顺序访问特性。

下面介绍B+树的主要方法的实现逻辑：

1. 插入（Insert）：

- 从根节点开始，根据插入键值找到叶子节点。

- 如果叶子节点未满，直接插入键值并保持有序。

- 如果叶子节点已满，进行分裂操作：

- 将叶子节点一分为二，分成左右两个节点。

- 将中间键值提升到父节点，并递归地进行插入操作。

- 如果父节点已满，继续向上递归进行分裂操作，直到根节点。

2. 删除（Delete）：

- 从根节点开始，根据删除键值找到叶子节点。

- 如果叶子节点中存在要删除的键值，直接删除。

- 如果叶子节点删除后过小，进行合并或者借用操作：

- 如果兄弟节点有富余的键值，可以借用一个键值过来。

- 如果兄弟节点的键值不够，可以进行合并操作，将当前节点和兄弟节点合并成一个节点。

- 如果父节点删除后过小，继续向上递归进行合并或者借用操作，直到根节点。

3. 查找（Search）：

- 从根节点开始，根据查找键值逐层向下搜索。

- 如果当前节点是内部节点，根据键值选择下一个子节点。

- 如果当前节点是叶子节点，根据键值在叶子节点中进行查找。

4. 范围查询：

- B+树天然支持范围查询，可以通过最左叶子节点和最右叶子节点进行范围限制，减少不必要的遍历。

总结起来，B+树的主要方法包括插入、删除、查找和范围查询。通过合适的平衡操作，B+树能够在动态数据集中高效地支持插入、删除和查找操作，使其成为一种广泛应用于存储和检索大量数据的数据结构。

如下图1展示了BPlusTree的类图，其中MTreeNode结构体的定义如下：

template <class T>

struct MtreeNode {

        bool isLeaf;

        int n;

        MtreeNode\* parent;

        std::vector<T> keys;

        std::vector<MtreeNode\*> pointers;

        std::vector<int\*> recptrs;

    };

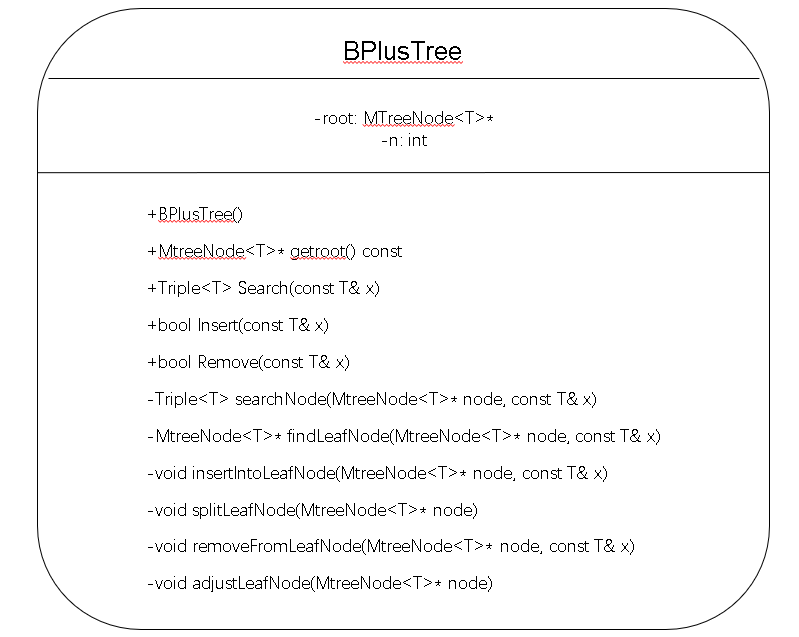


图1

1. **Row、Column和Page类**

Row类和Column类是为了方便实现存储数据库关系模式而设计的类。Row类主要用于存储数据行，其使用的核心数据结构是一个map(即键值对)，用来存储列名和相对应的值。为了保证Row可以存储字符串、整形、浮点型等多种类型的数据使用了一个特殊的数据结构即std::variant类型，std::variant 是 C++17 引入的标准库类型，它提供了一种能够容纳不同类型值的数据结构，类似于联合体（union）或变体（variant）的概念。不同于联合体，std::variant 保证了类型安全和异常安全，并提供了更多的功能和灵活性。

Column类主要储存数据表的定义，其主要的数据成员是用std::string类型存储的列的列名及其对应的数据类型。通过这两个类实现了数据表结构的控制。

而Page类则实现了一个分页，其中存储的是一个Row类的数组。设置该类是为了实现数据的存储结构与B+树类的连接，即Page类实际上就是B+树叶子节点下指针指向的数据。同时为了能够正常使用B+树相关的插入删除操作，还对该类进行了>、<、==等操作符的重载。下面的图2、图3、图4展示了Row和Column对应的类图。

表格

描述已自动生成

图2

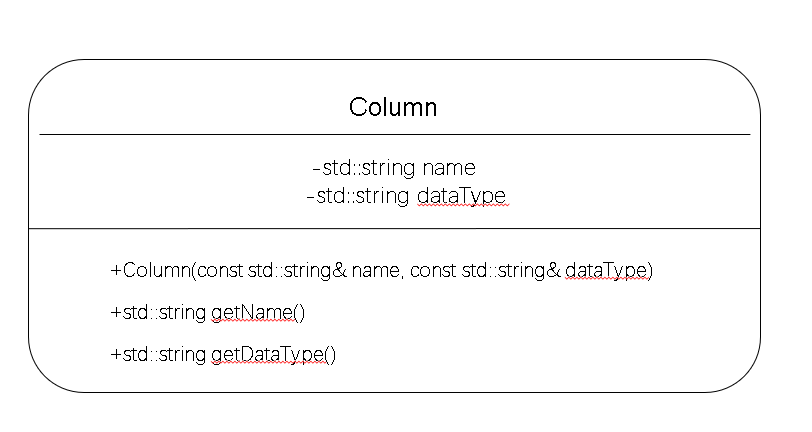


图3

表格

描述已自动生成

图4

1. **Table类**

Table类即对应数据库中的Table，其拥有如下数据成员：

    std::string name;               // 表名

    std::vector<Column> columns;    // 列列表

    BPlusTree<Page> pages;          // 数据存储结构

    uint32\_t nrow;                  // 表存储数据的行数

Table通过columns来控制输入数据格式的正确、通过调用B+树相应的插入、删除、搜索等方法实现数据的插入、删除等操作。、

下面是对几个成员函数的简单解释：

printRows() const：打印所有行的方法。首先，通过调用pages的traverseData()方法，获取当前页的向量。然后，使用循环遍历每个页，并获取页中的行向量。最后，使用嵌套循环遍历每一行，并调用行的print()方法打印行的内容

addRow(const Row& row)：添加行的方法。接受一个Row对象作为参数，用于表示要添加的行。该方法首先调用getEnd()方法获取当前表格的最后一页，然后判断该页是否存在。如果不存在，则创建一个新的Page对象，并将行插入其中，然后将该页插入到成员变量pages（数据存储结构）中。如果存在最后一页，则判断最后一页是否已满。如果未满，则将行插入最后一页。如果最后一页已满，则创建一个新的Page对象，并将行插入其中，然后将该页插入到成员变量pages中。如果表格已满，则输出错误信息。

deleteRow(const Row& row)：删除指定行的方法。首先，通过调用pages的traverseData()方法，获取当前页的向量。然后，使用循环遍历每个页，并调用页的DeleteRow()方法删除指定的行。

updateRow(const std::string& column, const std::any& oldValue, const std::any& newValue)：更新指定行的方法。首先，通过调用pages的traverseData()方法，获取当前页的向量。然后，使用嵌套循环遍历每一行，并检查是否存在指定的列名。如果存在，则获取该列的值，并与oldValue进行比较。如果相等，则将行中的值修改为newValue。

下图为Table类对应的类图

表格

中度可信度描述已自动生成

图5

1. **DB类**

DB类即database类，DB类只拥有一个数据成员，一个Table类的数组来存储创建的数据表。DB类实现了这样一些功能：

-打印一个database>的提示符

-解析例如”.exit”（退出数据库）等的元命令

-解析SQL语句

-执行SQL操作

其中实现解析SQL语句的过程中采用了许多字符串相关的数据结构的知识，包括子串定位、字符串的分割等。实习作业中简单实现了KMP算法，但因为SQL语句的语法结构较为固定，实际实现的过程中采用的是C++中自带的find()函数。执行SQL操作的过程被分割为多个子函数，其实现的逻辑主要是通过读入SQL语句解析的结果并调用Table类的相应方法来实现SQL操作。其中SQL语句解析结果的读入主要是通过一个Statement类，其具体定义如下：

class Statement {

    //定义SQL语句类型

    public:

        StatementofSQL type;

        std::vector<std::string> create\_request;

        std::vector<std::string> insert\_request;

        std::vector<std::string> select\_request;

        std::vector<std::string> update\_request;

        std::vector<std::string> delete\_request;

};

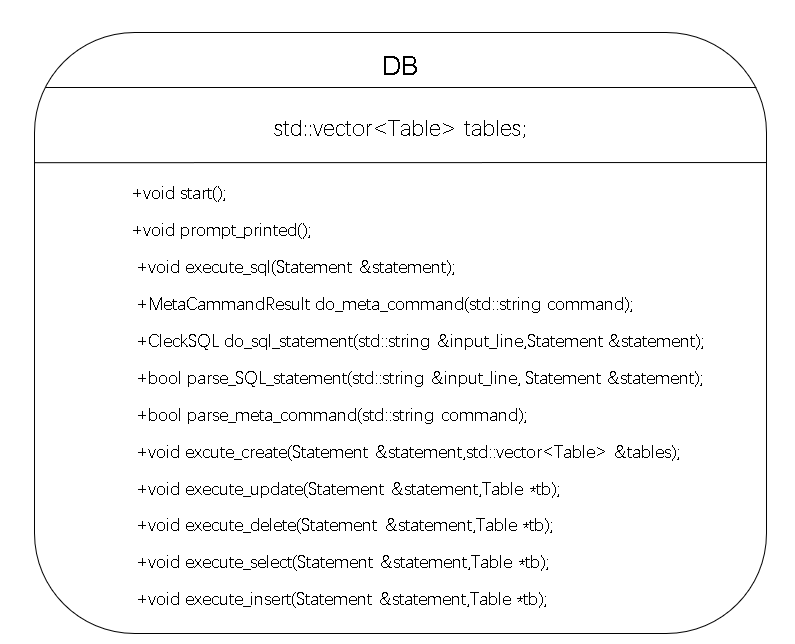


图6

**2.2 操作示例**

文本

描述已自动生成

**3 总结与局限**

通过本次实习任务，我们加深了对B+树、字符串、数组等数据结构的理解，并在C++中实现了一些基本的数据库功能。你学会了根据任务需求选择合适的解决方案，并结合数据结构的特点选择最适合的数据结构和算法。你还通过调试工具（如Visual Studio）进行了代码调试。

然而，本次实习作业只实现了数据库的基本功能，还有一些关系数据库的范式约束没有很好地实现。此外，你提到在实际操作中未能实现从外部文件导入数据的功能。这表明本实习作业仍有进步的空间。

总的来说，本次实习任务为我们提供了学习和实践的机会，增强了编程能力和对数据库相关概念的理解。在今后的学习和实践中，我们可以继续深入研究数据库的各种功能和约束，探索更高级的数据库操作和优化技术，以提升编程、开发能力。

**4 附录代码**

#include <iostream>

#include <string>

#include<string.h>

#include <vector>

#include<map>

#include<any>

#include <algorithm>

//以下定义一系列需要用到的常量或简单函数

#define size\_of\_attribute(class,attribute) sizeof(((class\*)0)->attribute)

//定义一个获取类中属性偏移值的函数，主要用来辅助表中属性的定义

//以下定义一系列枚举类型以增强代码的可读性

enum MetaCammandResult{

    //标识基础命令是否解析成功

    CAMMAND\_PARSE\_SUCCESS,

    CAMMAND\_UNRECOGNIZED

};

enum CleckSQL{

    //标识sql语句是否解析成功

    PARSE\_SQL\_SUCCESS,

    UNRECOGNIZED\_SQL,

    PARSE\_SYNTAX\_ERROR

};

enum StatementofSQL{

    //标识sql语句被解析为哪一种类型

    STATMENT\_CREATE\_TABLE,

    STATEMENT\_INSERT,

    STATEMENT\_SELECT,

    STATEMENT\_DELETE,

    STATEMENT\_UPDATE

};

//以下定义一系列需要用到的类

const int MaxValue = 10000000;

#define TREE\_NUM 4

template <class T>

struct MtreeNode {

        bool isLeaf;

        int n;

        MtreeNode\* parent;

        std::vector<T> keys;

        std::vector<MtreeNode\*> pointers;

        std::vector<int\*> recptrs;

    };

template <class T>

struct Triple{          //返回搜索结果的结构体

    MtreeNode<T> \*r;

    int i; int tag;     //i为结果中关键码序号

                        //tag = 0 搜索成功；tag = 1 搜索不成功；

};

template <class T>

class BPlusTree {

public:

    BPlusTree() {

        root = new MtreeNode<T>();

        m = TREE\_NUM;

    }

    MtreeNode<T>\* getroot() const {

        return root;

    }

    Triple<T> Search(const T& x) {

        return searchNode(root, x);

    }

    bool Insert(const T& x) {

        if (root == nullptr) {

            root = createLeafNode();

            root->keys.push\_back(x);

            root->pointers.push\_back(nullptr);

            root->n++;

            return true;

        }

        if (searchNode(root, x).tag == 0) {

            // Key already exists

            return false;

        }

        MtreeNode<T>\* node = findLeafNode(root, x);

        insertIntoLeafNode(node, x);

        if (node->n > m) {

            splitLeafNode(node);

        }

        return true;

    }

    bool Remove(const T& x) {

        if (root == nullptr) {

            return false;

        }

        Triple<T> searchResult = searchNode(root, x);

        if (searchResult.tag == 1) {

            // Key not found

            return false;

        }

        MtreeNode<T>\* node = findLeafNode(root, x);

        removeFromLeafNode(node, x);

        if (node->n < (m + 1) / 2 && node != root) {

            adjustLeafNode(node);

        }

        return true;

    }

    MtreeNode<T>\* findMaxKeyNode(MtreeNode<T>\* node) const{

        if (node == nullptr) {

         return nullptr;

        }

        if (node->isLeaf) {

            return node;

        }

        return findMaxKeyNode(node->pointers[node->n]);

    }

    std::vector<T\*> traverseData() const {

        std::vector<T\*> result;

        MtreeNode<T>\* leafNode = findMinKeyLeafNode(root);

        while (leafNode != nullptr) {

            for (int i = 0; i < leafNode->n; i++) {

                result.push\_back(leafNode->recptrs[i]);

            }

            leafNode = leafNode->pointers[leafNode->n];

        }

        return result;

    }

    MtreeNode<T>\* findMinKeyLeafNode(MtreeNode<T>\* node) const {

        if (node == nullptr) {

            return nullptr;

        }

        if (node->isLeaf) {

            return node;

        }

        return findMinKeyLeafNode(node->pointers[0]);

    }

    ~BPlusTree() {

    // 递归地删除根节点及其子节点

        deleteNode(root);

    }

    void deleteNode(MtreeNode<T>\* node) {

        if (node == nullptr) {

            return;

        }

        if (!node->isLeaf) {

            for (int i = 0; i <= node->n; i++) {

                deleteNode(node->pointers[i]);

            }

        }

        delete node;

    }

private:

    MtreeNode<T>\* root;

    int m = 4;

    Triple<T> searchNode(MtreeNode<T>\* node, const T& x) {

        int i = 0;

        while (i < node->n && x > node->keys[i]) {

            i++;

        }

        if (i < node->n && x == node->keys[i]) {

            return { node, i, 0 };

        } else if (node->isLeaf) {

            return { node, i, 1 };

        } else {

            return searchNode(node->pointers[i], x);

        }

    }

    MtreeNode<T>\* findLeafNode(MtreeNode<T>\* node, const T& x) {

        int i = 0;

        while (i < node->n && x > node->keys[i]) {

            i++;

        }

        if (node->isLeaf) {

            return node;

        } else {

            return findLeafNode(node->pointers[i], x);

        }

    }

    void insertIntoLeafNode(MtreeNode<T>\* node, const T& x) {

        int i = 0;

        while (i < node->n && x > node->keys[i]) {

            i++;

        }

        node->keys.insert(node->keys.begin() + i, x);

        node->pointers.insert(node->pointers.begin() + i, nullptr);

        node->recptrs.insert(node->recptrs.begin() + i, nullptr);

        node->n++;

    }

    void splitLeafNode(MtreeNode<T>\* node) {

        MtreeNode<T>\* newNode = createLeafNode();

        int splitIndex = (node->n + 1) / 2;

        newNode->n = node->n - splitIndex;

        for (int i = splitIndex; i < node->n; i++) {

            newNode->keys.push\_back(node->keys[i]);

            newNode->pointers.push\_back(node->pointers[i]);

            newNode->recptrs.push\_back(node->recptrs[i]);

        }

        node->n = splitIndex;

        node->pointers[splitIndex - 1] = newNode;

        if (node->parent == nullptr) {

            MtreeNode<T>\* newRoot = createInternalNode();

            newRoot->keys.push\_back(newNode->keys[0]);

            newRoot->pointers.push\_back(node);

            newRoot->pointers.push\_back(newNode);

            node->parent = newRoot;

            newNode->parent = newRoot;

            root = newRoot;

        } else {

            MtreeNode<T>\* parent = node->parent;

            int insertIndex = 0;

            while (insertIndex < parent->n && node->keys[0] > parent->keys[insertIndex]) {

                insertIndex++;

            }

            parent->keys.insert(parent->keys.begin() + insertIndex, newNode->keys[0]);

            parent->pointers.insert(parent->pointers.begin() + insertIndex + 1, newNode);

            newNode->parent = parent;

            if (parent->n > m) {

                splitInternalNode(parent);

            }

        }

    }

    void splitInternalNode(MtreeNode<T>\* node) {

        MtreeNode<T>\* newNode = createInternalNode();

        int splitIndex = (node->n + 1) / 2;

        newNode->n = node->n - splitIndex - 1;

        for (int i = splitIndex + 1; i < node->n; i++) {

            newNode->keys.push\_back(node->keys[i]);

            newNode->pointers.push\_back(node->pointers[i]);

        }

        newNode->pointers.push\_back(node->pointers[node->n]);

        node->n = splitIndex;

        if (node->parent == nullptr) {

            MtreeNode<T>\* newRoot = createInternalNode();

            newRoot->keys.push\_back(node->keys[splitIndex]);

            newRoot->pointers.push\_back(node);

            newRoot->pointers.push\_back(newNode);

            node->parent = newRoot;

            newNode->parent = newRoot;

            root = newRoot;

        } else {

            MtreeNode<T>\* parent = node->parent;

            int insertIndex = 0;

            while (insertIndex < parent->n && node->keys[splitIndex] > parent->keys[insertIndex]) {

                insertIndex++;

            }

            parent->keys.insert(parent->keys.begin() + insertIndex, node->keys[splitIndex]);

            parent->pointers.insert(parent->pointers.begin() + insertIndex + 1, newNode);

            newNode->parent = parent;

            if (parent->n > m) {

                splitInternalNode(parent);

            }

        }

    }

    void removeFromLeafNode(MtreeNode<T>\* node, const T& x) {

        int i = 0;

        while (i < node->n && x > node->keys[i]) {

            i++;

        }

        if (x == node->keys[i]) {

            node->keys.erase(node->keys.begin() + i);

            node->pointers.erase(node->pointers.begin() + i);

            node->recptrs.erase(node->recptrs.begin() + i);

            node->n--;

        }

    }

    void adjustLeafNode(MtreeNode<T>\* node) {

        int index = 0;

        while (node->parent->pointers[index] != node) {

            index++;

        }

        if (index > 0 && node->parent->pointers[index - 1]->n > (m + 1) / 2) {

            LeftAdjust(node->parent, node->parent->pointers[index - 1], index - 1, index);

        } else if (index < node->parent->n && node->parent->pointers[index + 1]->n > (m + 1) / 2) {

            RightAdjust(node->parent, node->parent->pointers[index + 1], index, index + 1);

        } else if (index > 0) {

            merge(node->parent, node->parent->pointers[index - 1], node, index - 1);

        } else {

            merge(node->parent, node, node->parent->pointers[index + 1], index);

        }

    }

    MtreeNode<T>\* createLeafNode() {

        MtreeNode<T>\* node = new MtreeNode<T>;

        node->isLeaf = true;

        node->n = 0;

        node->parent = nullptr;

        return node;

    }

    MtreeNode<T>\* createInternalNode() {

        MtreeNode<T>\* node = new MtreeNode<T>;

        node->isLeaf = false;

        node->n = 0;

        node->parent = nullptr;

        return node;

    }

    void LeftAdjust(MtreeNode<T>\* p, MtreeNode<T>\* q, int d, int j) {

        q->keys.push\_back(p->keys[d]);

        q->pointers.push\_back(p->pointers[d]);

        q->recptrs.push\_back(p->recptrs[d]);

        q->n++;

        p->keys[d] = q->keys[0];

        p->pointers[d] = q;

        p->recptrs[d] = q->recptrs[0];

        q->keys.erase(q->keys.begin());

        q->pointers.erase(q->pointers.begin());

        q->recptrs.erase(q->recptrs.begin());

        q->n--;

        p->keys[j] = q->keys[0];

    }

    void RightAdjust(MtreeNode<T>\* p, MtreeNode<T>\* q, int d, int j) {

        q->keys.insert(q->keys.begin(), p->keys[d]);

        q->pointers.insert(q->pointers.begin(), p->pointers[d]);

        q->recptrs.insert(q->recptrs.begin(), p->recptrs[d]);

        q->n++;

        p->keys[d] = q->keys[q->n - 1];

        p->pointers[d + 1] = q;

        p->recptrs[d + 1] = q->recptrs[q->n - 1];

        q->keys.pop\_back();

        q->pointers.pop\_back();

        q->recptrs.pop\_back();

        q->n--;

        p->keys[j] = q->keys[q->n - 1];

    }

    void merge(MtreeNode<T>\* p, MtreeNode<T>\* q, MtreeNode<T>\* pl, int j) {

        pl->keys.push\_back(p->keys[j]);

        pl->pointers.push\_back(q->pointers[0]);

        pl->recptrs.push\_back(q->recptrs[0]);

        pl->n++;

        for (int i = 0; i < q->n; i++) {

            pl->keys.push\_back(q->keys[i]);

            pl->pointers.push\_back(q->pointers[i + 1]);

            pl->recptrs.push\_back(q->recptrs[i + 1]);

            pl->n++;

        }

        p->keys.erase(p->keys.begin() + j);

        p->pointers.erase(p->pointers.begin() + j + 1);

        p->recptrs.erase(p->recptrs.begin() + j + 1);

        p->n--;

        delete q;

        if (p == root && p->n == 0) {

            delete p;

            root = nullptr;

        }

    }

};

#define COLUMN\_USERNAME\_SIZE 32

#define COLUMN\_EMAIL\_SIZE 255

class Column {

public:

    // 构造函数

    Column(const std::string& name, const std::string& dataType)

        : name(name), dataType(dataType) {}

    // 获取列名

    std::string getName() const {

        return name;

    }

    // 获取数据类型

    std::string getDataType() const {

        return dataType;

    }

private:

    std::string name;       // 列名

    std::string dataType;   // 数据类型

};

class Row {

public:

    // 添加列的值

    template <typename T>

    void addValue(const std::string& columnName, const T& value) {

        values.push\_back({{columnName, value}});

    }

    // 获取列的值

    template <typename T>

    T getValue(const std::string& columnName) const {

        for (const auto& entry : values) {

            auto it = entry.find(columnName);

            if (it != entry.end()) {

                try {

                    return std::any\_cast<T>(it->second);

                } catch (const std::bad\_any\_cast&) {

                    std::cout << "Error: Incorrect column type." << std::endl;

                }

            }

        }

        return T();

    }

    // 检查列是否存在值

    bool hasValue(const std::string& columnName) const {

        for (const auto& entry : values) {

            if (entry.find(columnName) != entry.end()) {

                return true;

            }

        }

        return false;

    }

    // 设置列的值

    void setValue(const std::string& columnName, const std::any& value){

        for (auto& map : values) {

            if (map.find(columnName) != map.end()) {

                auto& anyValue = map.at(columnName);

                try {

                    anyValue = std::any\_cast<decltype(anyValue)>(value);

                } catch (const std::bad\_any\_cast&) {

                    std::cout << "Error: Incorrect column type." << std::endl;

                }

                return;

            }

        }

    }

    void print(){

        for(const auto& map : values){

            for(const auto& pair :map){

                printValue(pair.second);

            }

        }

        std::cout<<std::endl;

    }

    // 相等比较运算符的重载

    bool operator==(const Row& other) const {

        return values == other.values;

    }

private:

    std::vector<std::map<std::string, std::any>> values;  // 列名和值的映射

    template <typename T>

    void printValue(const T& value) const {

        std::cout << value<<" ";

    }

};

const uint32\_t MaxPageSize = 1000; //某一表中最大存储的分页个数

const uint32\_t PageSize = 4096; //分页的最大大小

class Page

{

public:

    Page(int PageNum): PageNum(PageNum) {}

    int getPageNum(){

        return PageNum;

    }

    std::vector<Row> getRows(){

        return rows;

    }

    void InsertRow(const Row &row){

        rows.push\_back(row);

    }

    void DeleteRow(const Row& row) {

        auto it = std::find\_if(rows.begin(), rows.end(), [&row](const Row& r) {

            return r == row;

        });

        if (it != rows.end()) {

            rows.erase(it);

        }

    }

    bool operator<(const Page& other) const {

        return PageNum < other.PageNum;

    }

    bool operator>(const Page& other) const {

        return PageNum > other.PageNum;

    }

    bool operator==(const Page& other) const {

        return PageNum == other.PageNum;

    }

    void setRows(const std::vector<Row>& newRows) {

        rows = newRows;

    }

private:

    std::vector<Row> rows;

    int PageNum;

};

class Table {

public:

    // 构造函数

    Table(const std::string& name) : name(name) {}

    // 添加列

    void addColumn(const std::string& columnName, const std::string& dataType) {

        columns.push\_back(Column(columnName, dataType));

    }

    // 添加行

    void addRow(const Row& row) {

        MtreeNode<Page> \*page = getEnd();

        if(!page){

            Page p(0);

            p.InsertRow(row);

            pages.Insert(p);

        }

        if(sizeof(page->keys[page->n]) < PageSize){

            page->keys[page->n].InsertRow(row);

        }

        else{

            int i = page->keys[page->n].getPageNum();

            if(i < MaxPageSize){

                Page p(i + 1);

                p.InsertRow(row);

                pages.Insert(p);

            }

            else{

                std::cout<<"Error: Table is Full"<<std::endl;

            }

        }

    }

    // 获取表名

    std::string getName() const {

        return name;

    }

    // 获取列列表

    std::vector<Column> getColumns() const {

        return columns;

    }

    // 获取所有行

    MtreeNode<Page>\* getEnd() const {

        return pages.findMaxKeyNode(pages.getroot());

    }

    // 打印所有行

    void printRows() const {

        std::vector<Page\*> cur = pages.traverseData();

        for(int q = 0;q < columns.size();q++){

            std::cout<<columns[q].getName();

        }

        std::cout<<std::endl;

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

            std::vector<Row> rows = cur[i]->getRows();

            for(int j = 0;j < rows.size();j++){

                rows[j].print();

            }

        }

    }

    // 删除指定行

    void deleteRow(const Row& row) {

        std::vector<Page\*> cur = pages.traverseData();

        for(auto &page : cur){

            page->DeleteRow(row);

        }

    }

    void updateRow(const std::string& column, const std::any& oldValue, const std::any& newValue) {

        std::vector<Page\*> cur = pages.traverseData();

        for (const auto& page : cur) {

            std::vector<Row> rows = page->getRows();

            for (auto& row : rows) {

                if (row.hasValue(column)) {

                    std::any value = row.getValue<std::any>(column);

                    if (compareAny(value, oldValue)) {

                        row.setValue(column, newValue);  // 修改副本中的行对象

                    }

                }

            }

            // 将修改后的副本重新赋值给原始页对象

            page->setRows(rows);

        }

    }

    template <typename T>

    bool compareAny(const std::any& a, const T& b) {

        try {

            return std::any\_cast<T>(a) == b;

        } catch (const std::bad\_any\_cast&) {

            return false;

        }

    }

private:

    std::string name;               // 表名

    std::vector<Column> columns;    // 列列表

    BPlusTree<Page> pages;          // 数据存储结构

    uint32\_t nrow;                  // 表存储数据的行数

};

const uint32\_t TableMaxRownumble = MaxPageSize \* (PageSize/sizeof(Row));

const uint32\_t RowPrePager = PageSize/(sizeof(Row));

class Statement {

    //定义SQL语句类型

    public:

        StatementofSQL type;

        std::vector<std::string> create\_request;

        std::vector<std::string> insert\_request;

        std::vector<std::string> select\_request;

        std::vector<std::string> update\_request;

        std::vector<std::string> delete\_request;

};

std::vector<int> buildPartialMatchTable(const std::string& pattern) {

    int m = pattern.length();

    std::vector<int> table(m);

    int len = 0;

    table[0] = 0;

    int i = 1;

    while (i < m) {

        if (pattern[i] == pattern[len]) {

            len++;

            table[i] = len;

            i++;

        } else {

            if (len != 0) {

                len = table[len - 1];

            } else {

                table[i] = 0;

                i++;

            }

        }

    }

    return table;

}

// 执行KMP算法

void kmp(const std::string& text, const std::string& pattern) {

    int n = text.length();

    int m = pattern.length();

    std::vector<int> table = buildPartialMatchTable(pattern);

    int i = 0;

    int j = 0;

    while (i < n) {

        if (pattern[j] == text[i]) {

            j++;

            i++;

        }

        if (j == m) {

            std::cout << "Pattern found at index " << i - j << std::endl;

            j = table[j - 1];

        } else if (i < n && pattern[j] != text[i]) {

            if (j != 0) {

                j = table[j - 1];

            } else {

                i++;

            }

        }

    }

}

class DB{

    //定义数据库类型

    private:

        std::vector<Table> tables;

    public:

        void addTable(const Table& table) {

            tables.push\_back(table);

        }

    // 根据表名获取表

        Table getTable(const std::string& tableName) const {

            for (const Table& table : tables) {

                if (table.getName() == tableName) {

                    return table;

                }

            }

        // 返回一个空的表

            return Table("");

        }

        //启动虚拟机的函数

        void start();

        //打印一个提示符

        void prompt\_printed();

        void execute\_sql(Statement &statement);

        MetaCammandResult do\_meta\_command(std::string command);

        CleckSQL do\_sql\_statement(std::string &input\_line,Statement &statement);

        bool parse\_SQL\_statement(std::string &input\_line, Statement &statement);

        bool parse\_meta\_command(std::string command);

        void excute\_create(Statement &statement,std::vector<Table> &tables);

        void execute\_update(Statement &statement,Table \*tb);

        void execute\_delete(Statement &statement,Table \*tb);

        void execute\_select(Statement &statement,Table \*tb);

        void execute\_insert(Statement &statement,Table \*tb);

        ~DB(){}

};

void DB::prompt\_printed(){

    // 来打印一个提示符

    std::cout<<"database > ";

}

bool DB::parse\_meta\_command(std::string command){

    // 对输入的基础命令进行解析

    if (command[0] == '.'){

        switch (do\_meta\_command(command))

        {

        case CAMMAND\_PARSE\_SUCCESS:

            return true;

        case CAMMAND\_UNRECOGNIZED:

            std::cout<<"Unrecognized command"<<std::endl;

            return true;

        }

    }

    return false;

}

MetaCammandResult DB::do\_meta\_command(std::string command){

    //执行基本操作的函数

    if (command == ".exit"){

        //成功退出数据库

        std::cout<<"Bye!"<<std::endl;

        exit(EXIT\_SUCCESS);

    }

    else{

        return CAMMAND\_UNRECOGNIZED;

    }

    return CAMMAND\_PARSE\_SUCCESS;

}

bool DB::parse\_SQL\_statement(std::string &input\_line, Statement &statement){

    //判断SQL语句是否解析成功的函数

    switch(do\_sql\_statement(input\_line,statement)){

        case PARSE\_SQL\_SUCCESS:

            return false;

        case PARSE\_SYNTAX\_ERROR:

            std::cout<<"Syntax Error"<<std::endl;

            return true;

        case UNRECOGNIZED\_SQL:

            std::cout<<"Unrecognized statement"<<std::endl;

            return true;

    }

    return true;

}

void Stringsplit(const std::string& str, const std::string& split, std::vector<std::string>& res)

{   //插入一个分割字符串的函数来辅助解析SQL语句

    char\* strc = new char[str.size() + 1];

    strcpy(strc, str.c\_str());   // 将str拷贝到 char类型的strc中

    char\* temp = strtok(strc, split.c\_str());

    while (temp != NULL)

    {

        res.push\_back(std::string(temp));

        temp = strtok(NULL, split.c\_str()); // 下一个被分割的串

    }

    delete[] strc;

}

CleckSQL DB::do\_sql\_statement(std::string &input\_line,Statement &statement){

    //解析SQL语句的函数

    if(!input\_line.compare(0,6,"select")){

        statement.type = STATEMENT\_SELECT;

        std::sscanf(input\_line.c\_str(),"select %s from %s",

        statement.select\_request[0],

        statement.select\_request[1]);

        return PARSE\_SQL\_SUCCESS;

    }

    else if(!input\_line.compare(0,6,"insert")){

        statement.type = STATEMENT\_INSERT;

        size\_t tableStart = input\_line.find("into") + 5;

        size\_t tableEnd = input\_line.find(' ', tableStart);

        std::string tableName = input\_line.substr(tableStart, tableEnd - tableStart);

        size\_t columnsStart = input\_line.find('(', tableEnd) + 1;

        size\_t columnsEnd = input\_line.find(')', columnsStart);

        std::string columns = input\_line.substr(columnsStart, columnsEnd - columnsStart);

        size\_t valueStart = input\_line.find("values") + 8;

        size\_t valueEnd = input\_line.find(")",valueStart);

        std::string values = input\_line.substr(valueStart,valueEnd - valueStart);

        statement.insert\_request.push\_back(tableName);

        statement.insert\_request.push\_back(columns);

        statement.insert\_request.push\_back(values);

        return PARSE\_SQL\_SUCCESS;

    }

    else if(!input\_line.compare(0,6,"delete")){

        statement.type = STATEMENT\_DELETE;

        size\_t tableStart = input\_line.find("from") + 5;

        size\_t tableEnd = input\_line.find(' ', tableStart);

        std::string tableName = input\_line.substr(tableStart, tableEnd - tableStart);

        size\_t reqStart = input\_line.find('where') + 6;

        size\_t reqEnd = input\_line.size();

        std::string req = input\_line.substr(reqStart,reqEnd - reqStart);

        statement.delete\_request.push\_back(tableName);

        statement.delete\_request.push\_back(req);

        return PARSE\_SQL\_SUCCESS;

    }

    else if(!input\_line.compare(0,6,"update")){

        statement.type = STATEMENT\_UPDATE;

        size\_t tableStart = input\_line.find("update") + 7;

        size\_t tableEnd = input\_line.find(' ', tableStart);

        std::string tableName = input\_line.substr(tableStart, tableEnd - tableStart);

        size\_t setStart = input\_line.find('set') + 4;

        size\_t setEnd = input\_line.find(' ',setStart);

        std::string set = input\_line.substr(setStart,setEnd - setStart);

        statement.update\_request.push\_back(tableName);

        statement.update\_request.push\_back(set);

        return PARSE\_SQL\_SUCCESS;

    }

    else if(!input\_line.compare(0,12,"create table")){

        statement.type = STATMENT\_CREATE\_TABLE;

        std::string tableName;

        size\_t startPos = input\_line.find("create table") + strlen("create table");

        size\_t endPos = input\_line.find("(");

        tableName = input\_line.substr(startPos, endPos - startPos);

    // 提取字段定义部分

        std::string fieldDefs;

        startPos = endPos + 1;

        endPos = input\_line.find\_last\_of(")");

        fieldDefs = input\_line.substr(startPos, endPos - startPos);

        statement.create\_request.push\_back(tableName);

        statement.create\_request.push\_back(fieldDefs);

        return PARSE\_SQL\_SUCCESS;

    }

    else

    {

        return UNRECOGNIZED\_SQL;

    }

}

void DB::execute\_sql(Statement &statement){

    //实现各种SQL操作的总函数

    switch (statement.type)

    {

    case STATEMENT\_DELETE:{

        std::string tablename = statement.delete\_request[0];

        Table tb = this->getTable(tablename);

        execute\_delete(statement,&tb);

        std::cout<<"Execute delete statement"<<std::endl;

        break;

    }

    case STATEMENT\_INSERT:{

        std::string tablename = statement.insert\_request[0];

        Table tb = this->getTable(tablename);

        execute\_insert(statement,&tb);

        std::cout<<"Execute insert statement"<<std::endl;

        break;}

    case STATEMENT\_UPDATE:{

        std::string tablename = statement.update\_request[0];

        Table tb = this->getTable(tablename);

        execute\_update(statement,&tb);

        std::cout<<"Execute delete statement"<<std::endl;

        break;}

    case STATEMENT\_SELECT:{

        std::string tablename = statement.select\_request[1];

        Table tb = this->getTable(tablename);

        execute\_select(statement,&tb);

        std::cout<<"Execute select statement"<<std::endl;

        break;}

    case STATMENT\_CREATE\_TABLE:{

        excute\_create(statement,tables);

        std::cout<<"Execute creat table statement"<<std::endl;

        break;}

    }

    return;

}

void DB::excute\_create(Statement &statement,std::vector<Table> &tables){

    std::string tablename = statement.create\_request[0];

    Table tb(tablename);

    std::string column = statement.create\_request[1];

    std::vector<std::string> res;

    std::string splite = " ,";

    Stringsplit(column,splite,res);

    if(res.size()%2 == 0){

        for(int i = 0;i < res.size();i = i + 2){

            tb.addColumn(res[i],res[i+1]);

        }

        addTable(tb);

    }

    else{

        std::cout<<"Error: ColumnName or datatype missing"<<std::endl;

    }

}

void DB::execute\_insert(Statement &statement, Table \*tb){

    //实际执行insert操作的函数

    if(tb->getName() == ""){

        std::cout<<"Error: Table not created"<<std::endl;

        delete tb;

    }

    else{

        std::string column = statement.insert\_request[1];

        std::string value = statement.insert\_request[2];

        std::vector<std::string> res1;

        std::vector<std::string> res2;

        std::string splite = ",";

        Stringsplit(column,splite,res1);

        Stringsplit(value,splite,res2);

        Row row;

        if(res1.size() == res2.size() && res1.size() == tb->getColumns().size()){

            for(int j = 0;j < res1.size();j++){

                row.addValue(res1[j],res2[j]);

            }

            tb->addRow(row);

        }

        else{

            std::cout<<"Error: Column or value missing"<<std::endl;

        }

    }

}

void DB::execute\_select(Statement &statement, Table \*tb){

    if(tb->getName() == ""){

        std::cout<<"Error: Table not created"<<std::endl;

        delete tb;

    }

    else{

        std::string req = statement.select\_request[0];

        if(req == "\*"){

            tb->printRows();

        }

        else{

            std::cout<<"Error: Unrecognized select request"<<std::endl;

        }

    }

}

void DB::execute\_delete(Statement &statement,Table \*tb){

    if(tb->getName() == ""){

        std::cout<<"Error: Table not created"<<std::endl;

        delete tb;

    }

    else{

        std::string req = statement.delete\_request[1];

        std::vector<std::string> res;

        std::string splite = ",=";

        Stringsplit(req,splite,res);

        Row row;

        row.addValue(res[0],res[1]);

        tb->deleteRow(row);

    }

}

void DB::execute\_update(Statement &statement,Table \*tb){

    if(tb->getName() == ""){

        std::cout<<"Error: Table not created"<<std::endl;

        delete tb;

    }

    else{

        std::string req = statement.update\_request[1];

        std::vector<std::string> res;

        std::string splite = ",=";

        Stringsplit(req,splite,res);

        tb->updateRow(res[0],res[1],res[2]);

    }

}

void DB::start(){

    //运行数据库程序的虚拟机

    while (true){

        prompt\_printed();

        std::string input\_line;

        std::getline(std::cin,input\_line);

        if (parse\_meta\_command(input\_line))

        {

            continue;

        }

        Statement statement;

        if (parse\_SQL\_statement(input\_line,statement)){

            continue;

        }

        execute\_sql(statement);

    }

}

//以下为主函数

int main()

{

    DB db;

    db.start();

    return 0;

}