电 子 科 技 大 学

作 业 报 告

学生姓名：陈睿　　学 号：202122080911　　指导教师：刘杰彦

学生 E-mail：760684682@qq.com

Linux 下表的实现与应用

第一章 需求分析

1. 总体要求：

在 Linux 环境下，使用 C 或 C++语言，应用现代程序设计思想来存储一张表，并且能对该表进行查询数据、增加数据等操作。上述功能以 API 的形式提供给应用使用。

2. 存储要求：

利用已学的文件操作 API，在文件系统中存储一张表；该表中共有 100 个属性，即能存储 100 列数据，每个属性的大小都是 8 字节。该表能支持大数据存储，能支持的最大行数为一百万行，也可看作支持行数没有上限限制。

3. 添加要求：

提供一个 API 函数，能够实现向表中添加一行的功能（将插入的数据添加到表格的末尾）。

4. 搜索要求：

提供一个 API 函数，实现对表格的某一个属性进行范围查找或精确查找的功能。例如：查找在属性 A 上，大于等于 a，小于等于 b 的所有行，当上下限相等时，即为精确查找。用户可以根据自己的需求，指定在哪一个属性上进行搜索，当搜索结果所返回的行数过多时，可以只返回一小部分，在本实验中，只返回前 10 行数据。

5. 索引要求：

提供 API 函数，为表格的某一个属性建立索引结构，以实现快速搜索。本实验中选用 B+树来建立索引结构，对每一个属性所建立的索引结构，需要保存到

一个文件中（索引文件）；下次重启应用程序，并在搜索数据时，应先检查是否已为相应属性建立了索引结构。每次搜索时需要先检查是否有索引文件存在，如果存在，则使用已经建立的索引进行搜索，实现加速。

6. 并发要求：

应用程序可以以多线程的方式，使用上述 API。并且需要考虑互斥的要求，在保证多线程环境下，表、索引结构、索引文件的一致性。

7. 测试要求：

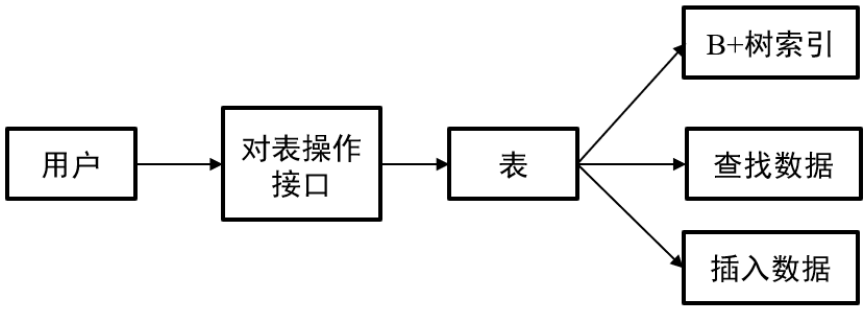
随机生成表中的大量数据，测试用例需要覆盖上述要求，以检测这些 API 功能是否完善。

第二章 总体设计

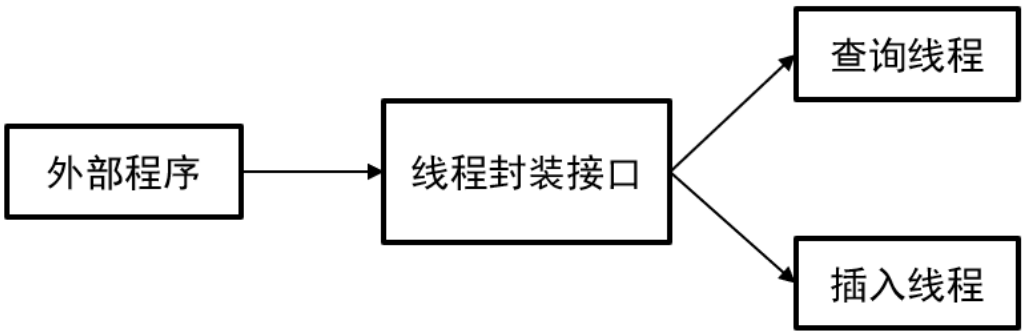
本实验是基于表完成多线程下索引表的创建、插入、查找操作。所以，首先需要实现表的数据结构的设计以及实例化一个表对象后对其进行初始化等具体操作。

本实验对表中的属性建立了索引，该索引是由 B+树实现的，能够完成对表中某个属性的快速查找，B+树数据结构将所有关键字保存在叶子节点中，在构建 B+树时需要考虑节点的分裂。需要将构建好的 B+树索引保存，供用户多次使用，所以也需要实现索引文件的读取。

接下来再实现本实验提供给用户使用的两个基本功能：数据插入和数据查找，这两个是基于索引实现的，在实验中将其进行封装供外部程序调用。



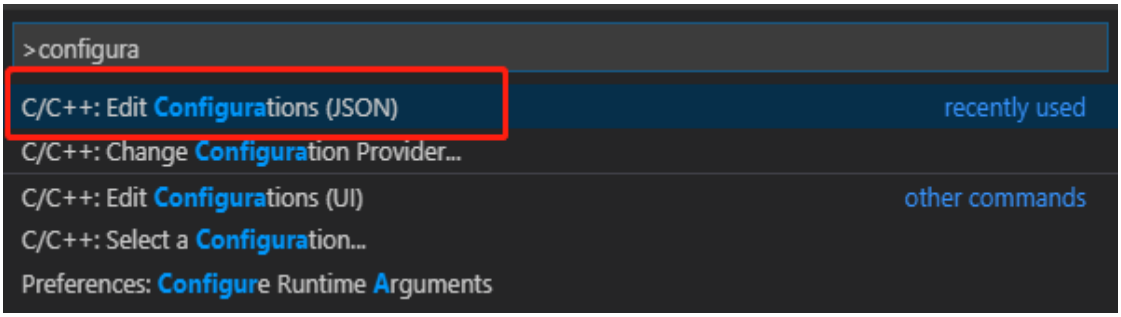
最后为了提高使用效率，引入了多线程同时进行数据查找和插入操作，利用线程封装的设计思想，用户可以通过调用接口创建两个不同的线程，实现并发，提高了使用效率。



第三章 详细设计与实现

1. 环境配置

本实验要求在 Linux 环境下实现 C/C++程序设计。本实验使用 Visual Studio Code 与远程 WSL 扩展一起使用，直接从 VS Code 使用 WSL 作为开发环境。由于 VS Code 只是一款文本编辑器，而不是编译器，所以需要先配置好环境才能进行编译。首先在 WSL 程序目录下打开 VS Code，在创建 cpp 文件后需要配置 C++环境。



注意在配置 c_cpp_properties.json 文件时，编译器路径“compilerPath”需要选择 WSL 中的“/usr/bin/gcc”，而不是 Windows 系统下的 mingw，否则会无法进行编译。

配置编译任务，将 tasks.json 文件修改为以下配置，修改完成后，可以根据自己的需求修改 command、args 或其他字段。

```
{
    "tasks": [
        {
            "type": "cppbuild",
            "label": "C/C++: g++ 生成活动文件",
            "command": "/usr/bin/g++",
            "args": [
                "-fdiagnostics-color=always",
                "-g",
                "${file}",
                "-o",
                "${fileDirname}/${fileBasenameNoExtension}"
            ],
            "options": {
                "cwd": "${fileDirname}"
            },
            "problemMatcher": [
                "$gcc"
            ],
            "group": {
                "kind": "build",
                "isDefault": true
            },
            "detail": "调试器生成的任务。"
        }
    ],
    "version": "2.0.0"
}
```

配置调试环境，配置 lanch.json 文件如下图所示。利用 VS Code 中的可视化界面，可以直接在代码中添加断点进行调试。

```
{
    // 使用 IntelliSense 了解相关属性。
    // 悬停以查看现有属性的描述。
    // 欲了解更多信息，请访问: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "name": "(gdb) 启动",
            "type": "cppdbg",
            "request": "launch",
            "program": "/mnt/d/linux_project/test",
            "args": [],
            "stopAtEntry": false,
            "cwd": "${fileDirname}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "miDebuggerPath": "/usr/bin/gdb",
            "setupCommands": [
                {
                    "description": "为 gdb 启用整齐打印",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                }
            ]
        }
    ]
}
```

2. 表结构设计与实现：

2.1 元组数据结构

本实验的核心数据结构为表，需要对表进行初始化，并在表上进行插入和搜索操作。在本实验中设计了一个表对象，其中包含了指向该表的文件描述符、表中现存有的元组数量以及存储在表中的元组，每个元组即为表中的一行数据，可以使用如下数据结构来表示一条元组。

```
typedef struct Tuple {
    int64_t id; //该元组的序号
    int64_t attribute[RECORD_LENGTH];//属性值
} Tuple;
```

2.2 实例化表对象

在用户调用 API 对表进行操作时，在实例化表对象时就把表打开，不需要每次插入数据时都进行文件打开操作。首先需要判断查看表中现有元组数量，如果表中现有元组数量为 0，需要对表进行初始化，随机生成大量的随机数，并产生 10000 个元组。

```
table::table() {
    m_Fd = open(RECORD_FILE, O_RDWR | O_CREAT | O_APPEND, S_IRUSR
| S_IWUSR);
    if (m_Fd == -1)
        throw "In table::table(),open error";

    //读取文件中已有的记录数量
    records = new Tuple[MAX_RECORD_NUM];
    //判断表格是否为空
    if (lseek(m_Fd, 0, SEEK_END) == 0) {
        record_num = 0;
        InitializeTable();
    } else {
        record_num = lseek(m_Fd, 0, SEEK_END) / RECORD_SIZE_BYTE;
        lseek(m_Fd, 0, SEEK_SET);
        read(m_Fd, records, record_num * RECORD_SIZE_BYTE);
    }
    tree = new BPlusTree;
    srand((unsigned) time(NULL));//随机数种子

    }
}
```

如果每次对表进行操作都创建一个表实例，会造成极大的资源消耗，所以考虑设置一个 table 类型的公有静态变量。在获取一个表实例时，只需要访问这个静态变量，如果为空则实例化一个表对象，非空则调用该实例。

2.3 向表中插入数据

对表进行插入数据时，随机生成一组数据。在将该组数据加入表后，需要更新索引，本实验中采用的是 B+树作为索引，如之前对某一个属性建立了 B+树，那么需要将插入的数据中对应属性的数据都加入到该 B+树中。关于 B+树的相关内容将在下一部分介绍。

```
//插入记录 void table::InsertRecord() {
    Tuple tuple = CreateRecord();
    if (!AppendRecord(tuple))
        throw "In table::InsertRecord(),insert error";
    std::cout << "已成功添加一条记录： " << std::endl;
    DisplayRecord(record);
    //更新索引
    for (int col = 1; col < RECORD_LENGTH + 1; col++) {
        if (Is_Index_File_Exists(col))
            UpdateIndexFile(col);
    }
}
```

2.4 从表中搜索数据

对表中数据进行搜索时，如果逐行进行扫描，则效率十分低下。本实验基于B+树构建了索引。在进行搜索时首先需要检查对应的属性是否已经构建过索引，如果有现存的索引文件，则读取该索引。否则为该属性创建索引，并将该索引文件保存。在本实验中实现了两种 API 来实现两种不同的搜索。SearchValueEqual(root, left, right, num)函数可以实现精确搜索某一属性中值为left的元组，该函数形参中 left 和 right 值相等，代表搜索某一数值。函数SearchValueRange(root, left, right, num)实现了在某一属性中搜索范围在[left, right]的所有元组，这两个 API 具体的实现数据的搜素都是基于 B+树上实现的，将在后面详细介绍。

```
//搜索记录
void table::SearchRecord(int left, int right, int col) {
    std::cout << "正在搜索第" << col << "列，范围：[" << left << ","
<< right << "]" << std::endl;
    //P(mutex)
    pthread_mutex_lock(m_pMutexForOperatingTable);

    //存放搜索结果
    int64_t *result = new int64_t[MAX_RESULT_NUM];
    int num = 0;
    BPlusTreeNode *root;//存放 B+树
    //首先判断是否有索引文件
    if (Is_Index_File_Exists(col)) {
        std::cout << "已查找到索引文件，正在读取..." << std::endl;
        root = tree->ReadBPlusTree(col);
        if (!root) {
            std::cout << "读取索引文件失败" << std::endl;
            return ;
        }

    } else {
        std::cout << "未查找到索引文件，正在创建..." << std::endl;
        root = CreateBPlusTree(col);
        if (!root) {
            std::cout << "创建索引文件失败";
            return;
        }
        //创建索引文件
        CreateIndexFile(root, col);
        std::cout << "已为第" << col << "列创建索引文件" <<
std::endl;
```

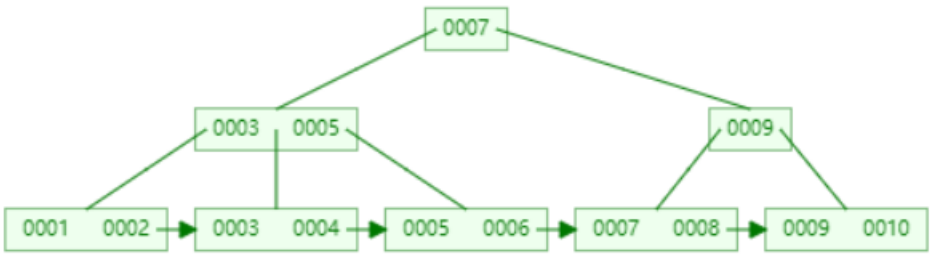
```
    }

    if (left == right) {
        tree->SearchValueEqual(root, left, result, num);
    } else {
        tree->SearchValueRange(root, left, right, result, num);
    }
    std::cout << "搜索结果为: " << std::endl;
    for (int i = 0; i < num; i++)
        DisplayRecord(records[result[i] - 1]);
    //V(mutex)
    pthread_mutex_unlock(m_pMutexForOperatingTable);
}
```

3. 索引 B+树的设计与实现：

3.1 B+树的基本概念

B+ 树是一种树数据结构，是一个 m 叉树，每个节点通常有多个孩子，一棵 B+树包含根节点、内部节点和叶子节点。每个节点中只能存储 k 个关键字 (ceil(m/2) ≤ k < m)，内部节点有 k+1 个指针指向子节点。每个节点中的元素从小到大排列，节点当中 k-1 个元素正好是 k 个孩子包含的元素的值域分划。B+树中的所有叶子节点都位于同一层，所有的关键字都存储在叶子节点上，进行搜索时，只需在叶子节点上查找。而且各个叶子节点直接按照从小到大的顺序相连，每个叶子节点都有指针指向下一个叶子节点，利用这个性质，可以完成数据的索引，实现快速搜索。

一棵 B+树的示意图如下所示：



3.2 B+树节点的数据结构

下面考虑 B+树中的每个节点的数据结构，在本实验中设计了如下数据结构。假设该 B+树的度为 2M，理论上每个节点只能存储 2M-1 个关键字和 2M 个指向它们子节点的指针。考虑到在实际生成 B+树时节点需要进行分裂操作，需要在

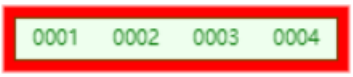
节点中增加一个空间来保存分裂前的数据，所以在每个节点中有一个大小为 2M 的关键字数组和一个大小为 2M+1 的子节点数组来存储数据；在保存数据的同时，需要记录节点中关键字数量并标记其是否为叶子节点。同时，需要一个指针来分别指向父节点，如果是叶子节点则需要一个指针指向其兄弟节点。

```
//B+树节点结构
typedef struct BPlusTreeNode {
    IndexNode index_nodes[2 * M ];//关键字，即为索引节点  key
    struct BPlusTreeNode* childs[2 * M + 1]; //子节点数
组       value
    int num;//关键字数量
    bool is_leaf; //判断是否为叶子节点
    struct BPlusTreeNode* parent;
    struct BPlusTreeNode* next;
} BPlusTreeNode;
```

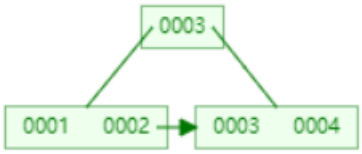
3.3 B+树节点的分裂

节点的分裂。在向 B+树插入关键字时，如果一个节点中的关键字超过了 2M-1 个，那么这个节点需要进行分裂，就是将该节点分裂成两个节点，并且在分裂节点的父节点上添加一条新的链接到新产生的节点；如果根节点需要分裂，由于它并没有父亲节点，所以需要创建一个节点来作为它的父亲节点，也就是这颗 B+树分裂后新的根节点。

B＋树的叶子节点和内部节点分裂时的操作是不同的，所以本实验编写了两个函数来区别这两种不同的分裂方式。首先考虑叶子节点分裂，在向一棵 B＋树插入关键字时，要先找到叶子节点，在叶子节点上插入，如果该 B＋树需要分裂，则一定会先在叶子节点上分裂。当 B+树只有一个节点时，此时的根节点就是叶子节点，下面来演示当 M＝ 2 时，叶子节点是如何分裂的。

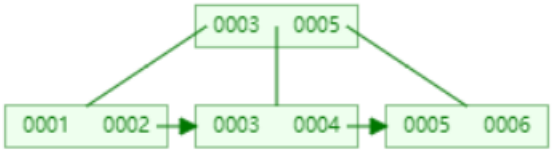


此时节点中的关键字个数超过了 2 M- 1 ＝ 3 个，需要进行分裂。待分裂节点需要分裂成两个大小为 M 的节点，其中待分裂节点并不会实际删除其中的数据，而是使用标记删除的方法，将其中的关键字数量标记为 M 即可。同时，需要将待分裂节点中的第 M＋ 1 个关键字上提到父节点中，第 M＋ 1 个关键字及以后的关键字分裂到新产生的节点中。由于此时是根节点需要分裂，则需要创建一个新节点，将第 M＋ 1 个关键字加入到新产生的父节点中。分裂后的 B+树如下图所示：



下面来考虑一般情况下叶子节点是如何分裂的。向上图中的 B＋树中再加入两个关键字，此时根节点的第二个叶子节点需要分裂。

首先需要将该叶子节点分裂成两个叶子节点，原叶子节点中包含关键字 3 和 4，新产生的叶子节点包含关键字 5 和 6 。同时，需要将关键字 5 上提到父节点中，找到需要插入的位置，并产生一条新的链接指向新产生的叶子节点。分裂后的 B＋树如图所示：



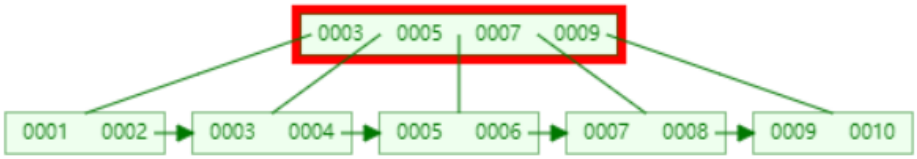
在对叶子节点进行分裂后，需要从该节点向上检查其父节点是否需要分裂，整个 B＋树的内部节点，是否也需要分裂，直到这棵树的根节点为止。以上就完成了 B＋树叶子节点的分裂操作。本实验中实现叶子节点分裂的函数 BPlusTree_Split_leaf()如下，该函数输入待分裂的子节点，返回分裂后 B＋树的根节点。

```
//叶子节点需要分裂
BPlusTreeNode *BPlusTree::BPlusTree_Split_leaf(BPlusTreeNode *node){
    BPlusTreeNode *new_node = (BPlusTreeNode
*)malloc(sizeof(BPlusTreeNode));
    //分裂节点，创建新节点，对其赋值
    new_node->next = node->next;
    node->next = new_node;
    new_node->is_leaf = node->is_leaf;
    node->num = M;
    new_node->num = M;
    for (int i = 0; i < M; i++){
```

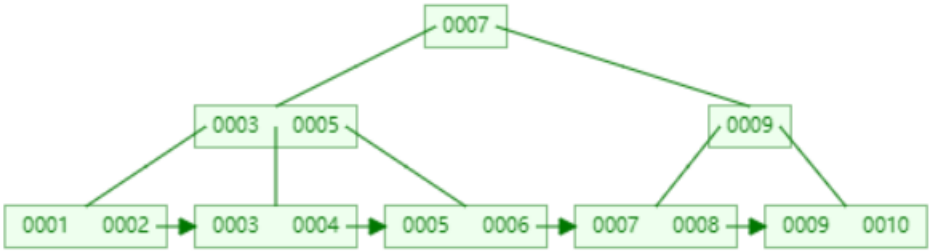
```
        new_node->index_nodes[i] = node->index_nodes[M + i];
        new_node->childs[i] = node->childs[M + i];
    }
    int insert_num = node->index_nodes[M].value;
    BPlusTreeNode *parent = node->parent;
    //如果是根节点，分裂时需要产生一个新的父节点
    if (parent == NULL){
        parent = BPlusTreeNode_new();
        parent->is_leaf = false;
        parent->num = 1;
        parent->index_nodes[0] = node->index_nodes[M];
        parent->childs[0] = node;
        parent->childs[1] = new_node;
        new_node->parent = parent;
        node->parent = parent;
    }
    else{
        new_node->parent = parent;
        int loc = 0;
        //找到在父节点中的插入位置
        while (loc < parent->num && insert_num >
parent->index_nodes[loc].value)
            loc++;
        //插入 key
        for (int i = parent->num - 1; i >= loc; i--)
            parent->index_nodes[i + 1] = parent->index_nodes[i];
        parent->index_nodes[loc] = node->index_nodes[M];
        loc++;
        //插入 value
        for (int i = parent->num; i >= loc; i--)
            parent->childs[i + 1] = parent->childs[i];
        parent->childs[loc] = new_node;
        parent->num++; //在父节点中也需要添加一个节点
    }
    BPlusTreeNode *root = node; //返回的 B+树根节点
    //从分裂节点向上遍历到根节点
    while (root->parent){
        root = root->parent;
        //如果内部节点需要分裂
        if (root->num == 2 * M)
            root = BPlusTree_Split_inner(root);
    }
    return root;
}
```

内部的节点分裂与叶子节点的分裂有些许区别，但大体上一支。如果一个内部节点中的关键字数量超过 2M-1 时，需要将该内部节点分裂成两个数量分别为 M 和 M-1 的内部节点，并将第 M+1 个关键字上提到该内部节点的父节点中，在父节点中增加一条链接指向新产生的内部节点。

下面来演示当 M＝ 2 时，内部节点是如何分裂的。此时内部节点中的关键字数量超过了 2M-1，需要进行分裂。



首先，将内部节点分裂成两个内部节点。原内部节点的关键字数量减少为 M，保留其中的 3 和 5 这两个关键字，保留其中前三个指向孩子节点的指针；新产生的节点中只包含 M-1 个关键字，在这个例子中只包含了 9 这个关键字。注意，这里与叶子节点的分裂不同，新产生的节点只包含了 M-1 个关键字，而非 M 个关键字！与叶子节点相同的是，将待分裂节点中的第 M+1 个关键字上提到父节点中，由于该内部节点是根节点，并不存在父节点，所以需要产生一个节点作为根节点。分裂后的 B+树如下图所示：



本实验中实现内部节点分裂的函数 BPlusTree_Split_inner()如下：与分裂叶子节点不同的是，函数的输入和返回值都为需要分裂的内部节点。

```
//内部节点需要分裂
BPlusTreeNode *BPlusTree::BPlusTree_Split_inner(BPlusTreeNode *node)
{
    BPlusTreeNode *new_node = (BPlusTreeNode
*)malloc(sizeof(BPlusTreeNode));
    new_node->is_leaf = node->is_leaf;
    int i;
    for (i = 0; i < M - 1; i++)
```

```
    {
        new_node->index_nodes[i] = node->index_nodes[M + 1 + i];
        new_node->childs[i] = node->childs[M + 1 + i];
       node->childs[M + 1 + i]->parent = new_node;
    }
    new_node->childs[i] = node->childs[2 * M];
    ode->childs[2 * M]->parent = new_node;
    new_node->num = M - 1; //新产生的节点只有 M-1 个关键字
    node->num = M;
    BPlusTreeNode *parent = node->parent;
    //父节点为空，则需要新创建一个父节点
    if (!parent)
    {
        parent = BPlusTreeNode_new();
        parent->num = 1;
        parent->is_leaf = false;
        parent->index_nodes[0] = node->index_nodes[M];
        parent->childs[0] = node;
        parent->childs[1] = new_node;
        new_node->parent = parent;
        node->parent = parent;
    }
    else
    {
        new_node->parent = parent;
        int insert_num = node->index_nodes[M].value;
        int loc = 0;
        //找到在父节点中的插入位置
        while (loc < parent->num && insert_num >
parent->index_nodes[loc].value)
            loc++;
        //插入 key
        for (int i = parent->num - 1; i >= loc; i--)
            parent->index_nodes[i + 1] = parent->index_nodes[i];
        parent->index_nodes[loc] = node->index_nodes[M];
        loc++;
        //插入 value
        for (int i = parent->num; i >= loc; i--)
            parent->childs[i + 1] = parent->childs[i];
        parent->childs[loc] = new_node;
        parent->num++;
    }
    return node;
}
```

3.4 B+树搜索数据

本实验提供了两种 API 实现数据的搜索，分别是精确搜索和范围搜索这两种搜索方法，在 B+树上实现这两种搜索是大同小异的。一般情况下，首先我们需要判断当前节点是否为叶子节点，当该节点为内部节点时，我们知道节点中的关键字是由小到大排序的，将待搜索关键字与内部节点中的关键字依次比较大小，直到待搜索关键字不大于节点中的关键字，以此来定位该关键字在该内部节点的哪个子树中。当定位到叶子节点后，首先对该叶子节点的所有关键字进行遍历，查找是否包含与待搜索关键字相等的值，然后检查该叶子节点的兄弟节点是否含有相等的关键字，直到检索到大于待搜索值的关键字。

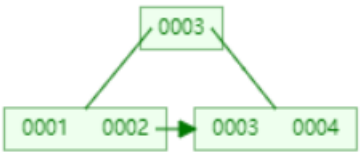
范围搜索与精确搜索是类似的，给用户的提供对 API 中含有搜索范围的上界和下界，在 B+树中则分别定位这两个关键字所在的叶子节点，从包含下界关键字的叶子节点开始遍历其兄弟节点，直到包含上界的叶子节点。在本实验中，由于表中含有大量数据，可以设置返回的查询结果的数量。

```
/** 搜索特定属性值
 * root B+树根节点
 * value 属性值
 * result 结果数组 存放记录的主键
 * num 结果数
 */
void BPlusTree::SearchValueEqual(BPlusTreeNode *root, int value, int64_t
*result, int &num)
{
    num = 0;
    if (!root)
        return;
    BPlusTreeNode *node = root;
    while (!node->is_leaf)
    { //不是叶节点，向下搜索
        int loc = 0;
        while (loc < node->num && value > node->index_nodes[loc].value)
            loc++;
        node = node->childs[loc];
    }
    //到达叶节点，顺着 next 指针往前搜索
    while (node)
    {
        for (int i = 0; i < node->num && num < MAX_RESULT_NUM; i++)
        {
            if (node->index_nodes[i].value > value)
                return;
            if (node->index_nodes[i].value == value)
```

```
                result[num++] = node->index_nodes[i].primary_key;
        }
        if (num == MAX_RESULT_NUM)
            return;
        node = node->next;
    }
}
```

3.5 B+树中插入数据

我们在初始化 B＋树时，只初始化了一个根节点，其中不包含任何关键字。向 B＋树插入关键字时，需要从根节点出发，向下找到待插入的叶子节点的位置，将关键字插入到该节点中，并检查是否需要分裂。向一棵空的 B＋树插入关键时，此时的根节点就是叶子节点，首先往根节点中插入关键字直到根节点进行一次分裂，此时根节点指向两个叶子节点。第一个叶子节点中的所有关键字都小于根节点中的关键字，第二个叶子节点中的关键字大于等于根节点中的关键字。



一般情况下，往 B+树中插入关键字的思路和在 B+树中进行搜索是一样的。但需注意的是，在定位叶子节点时，有一个小细节时不同的。在搜索数据时，关键字不大于内部节点中的关键字就定位成功，而在插入数据时，关键字小于内部节点中的关键字才定位成功。先定位到需要插入的叶子节点的位置，将关键字插入该叶子节点后，检查是否需要分裂，如果需要则调用叶子节点分裂函数，返回分裂后的 B+树。本实验中向 B+树插入节点的函数如下：

```
//插入节点
BPlusTreeNode *BPlusTree::BPlusTree_Insert(BPlusTreeNode *root, const
Record &record, int col)
{
    IndexNode indexNode = IndexNode_new(record, col);
    if (!root)
    {
        return NULL;
    }
    BPlusTreeNode *probe = root;
    while (!probe->is_leaf)
    {
        int loc = 0;
        //找到在内部节点中需要探索的子节点
```

```
        while (loc < probe->num && indexNode.value >=
probe->index_nodes[loc].value)
            loc++;
        probe = probe->childs[loc];
    }
    int loc = 0;
    while (loc < probe->num && indexNode.value >
probe->index_nodes[loc].value)
        loc++;
    if (probe->num == 0)//如果为空，直接插入
    {
        probe->index_nodes[0] = indexNode;
    }
    else//如果非空，需要移动
    {
        //插入 key
        for (int i = probe->num - 1; i >= loc; i--)
            probe->index_nodes[i + 1] = probe->index_nodes[i];
        probe->index_nodes[loc] = indexNode;
    }
    probe->num++;
    if (probe->num == 2 * M)
    {
        root = BPlusTree_Split_leaf(probe);
    }
    return root;
}
```

4. 多线程的设计与实现：

4.1 互斥量

在多线程环境下，表是临界资源，需要使用互斥量来实现某一时刻只有单线程能对表进行操作。首先，使用互斥量 m_pMutexForCreatingTable 来保证只有一个线程来创建表实例。在本实验中使用了两个线程来分别对表进行插入和查询操作，所以对表进行操作时需要互斥。使用互斥量 m_pMutexForOperatingTable 来保证插入和查询操作不能同时进行。所以，在上述搜索和插入代码中加入 PV 操作，使用函数 pthread_mutex_lock()、pthread_mutex_unlock()对互斥量实现加锁和取锁操作。

4.2 封装线程执行体

在本实验中设计了两种线程，分别为查询线程和插入线程，本实验参考课程中第三讲线程封装方法中的基于接口的封装方法，利用该程序设计思想，通过创建不同线程的方式执行业务逻辑。

为了实现不同的业务逻辑，创建一个接口类：CLExecutiveFunctionProvider，该类是作为基类派生出了实现业务逻辑的子类，在提供给用户的 API 中，将该类在函数中作为构造函数使用，即可实现封装，降低耦合度。

```
//执行体功能的接口类，线程业务逻辑的提供者
class CLExecutiveFunctionProvider{
public:
    CLExecutiveFunctionProvider(){}
    virtual ~CLExecutiveFunctionProvider(){}
public:
    virtual void RunExecutiveFunction()=0;
};

#endif // CLEXECUTIVEFUNCTIONPROVIDER_H
```

创建一个接口类：CLExecutive，作为函数创建的接口类，实现线程创建的封装。

```
#include "CLExeFuncProvider.h"

//执行体的接口类
class CLExecutive{
public:
    explicit CLExecutive(CLExecutiveFunctionProvider*
pExecutiveFunctionProvider){
        m_pExecutiveFunctionProvider=pExecutiveFunctionProvider;
    }
    virtual ~CLExecutive(){}
    virtual void Run()=0;
    virtual void WaitForDeath()=0;

protected:
    CLExecutiveFunctionProvider* m_pExecutiveFunctionProvider;

};


#endif // CLEXECUTIVE_H
```

同时，通过 CLExecutive 派生出 CLThread 完成线程创建的具体实现，CLExecutiveFunctionProvider 派生出 CLExecutiveInsert 和 CLExecutiveSearch 具体实现表中的插入和搜索工作。

```
#include "CLThread.h"
```

```
CLThread::CLThread(CLExecutiveFunctionProvider*
pExecutiveFunctionProvider):CLExecutive(pExecutiveFunctionProvider) {
}

CLThread::~CLThread() {
}

void* CLThread::StartFunctionOfThread(void *pThis){

    CLThread* pThreadThis=(CLThread *)pThis;
    std::cout<<std::endl<<"线程 ID:"<<pThreadThis->m_ThreadID<<std::endl;
    pThreadThis->m_pExecutiveFunctionProvider->RunExecutiveFunction();
    return 0;
}

void CLThread::Run(){
    int r=pthread_create(&m_ThreadID,0,StartFunctionOfThread,this);

    if(r!=0)
    {
        std::cout<<"pthread_create error"<<std::endl;
        return ;
    }
}

void CLThread::WaitForDeath(){
    int r=pthread_join(m_ThreadID,0);
    if(r!=0)
    {
        std::cout<<"In CLThread::WaitForDeath(),pthread_join
error"<<std::endl;
        return ;
    }
}
```

第四章 测试

1. 测试环境

本实验是在 WSL2（Windows Subsystem for Linux）完成的。WSL 提供了一个微软开发的 Linux 兼容内核接口（不包含 Linux 代码），来自 Ubuntu 的用户模式二进制文件在其上运行。查看 Linux 版本：主机名为“cr”，本人姓名首字母的缩写。

```
cr@LAPTOP-U6G05SCN:~/linux_project$ cat /proc/version
Linux version 5.10.16.3-microsoft-standard-WSL2 (oe-user@oe-host)
cr@LAPTOP-U6G05SCN:~/linux_project$
```

2. 编译执行

编写 Makefile 文件，实现了各个源文件的编译，规定了各个源文件的编译顺序，编写完成后使用命令 make，完成整个工程的自动编译。编写的 Makefile 文件如下：

```
test: main.cpp BPlusTree.o table.o CLThread.o CLExeFuncInsert.h
CLExeFuncSearch.h CLThread.h
    g++ -g main.cpp BPlusTree.o table.o CLThread.o -o test -lpthread
BPlusTree.o: BPlusTree.cpp BPlusTree.h structure.h
    g++ -g -c BPlusTree.cpp
table.o: table.cpp table.h BPlusTree.h structure.h
    g++ -g -c table.cpp
CLThread.o: CLThread.cpp CLThread.h CLExecutive.h
    g++ -g -c CLThread.cpp
clean:
    rm -rf *.o $(TARGET)
```

在命令行中输入 make 命令，执行编译：

```
cr@LAPTOP-U6G05SCN:~/linux_project$ make
g++ -g -c BPlusTree.cpp
g++ -g -c table.cpp
g++ -g -c CLThread.cpp
g++ -g main.cpp BPlusTree.o table.o CLThread.o -o test -lpthread
```

编译成功，产生一个名为 test 的可执行文件

3. 开始测试

3.1 插入测试

首先测试在实例化一个表后能否向其中插入数据。编写主函数用于测试，如果能插入成功，则显示插入数据。

```
int main()
{
    table* m_table=table::GetTable();
    m_table->InsertRecord();
```

```
}
```

在命令行输入./test 执行编译好的可执行文件

```
cr@LAPTOP-U6G05SCN:~/linux_project$ ./test
已成功添加一条记录:
-------------------------------------------------
457  203  662  922  595  698  379  384  397  126
```

可以看到成功向表中插入了一个元组，一共有 100 个属性，每个属性的范围都是 1 到 1000 之间的整数，为了便于用户查看，只显示了其中的前 10 个属性。

3.2 查找测试

在可以看到在插入的数据中，第一列包含了数值 457，接下来在表中查找第一列属性为 457 的所有元组。将测试代码修改为如下：

```
int main()
{
    //测试
    table* m_table=table::GetTable();
    m_table->SearchRecord(457,457,1);
}
```

在命令行输入 make 命令重新编译，编译完成后输入./test 执行。

```
cr@LAPTOP-U6G05SCN:~/linux_project$ make
g++ -g main.cpp BPlusTree.o table.o CLThread.o -o test -lpthread
cr@LAPTOP-U6G05SCN:~/linux_project$ ./test
正在搜索第1列，范围：[457,457]
未查找到索引文件，正在创建...
已为第1列创建索引文件
搜索结果为:
-------------------------------------------------
457  387  260  871  606  572  57   279  551  36
-------------------------------------------------
457  493  161  516  120  247  750  12   527  945
-------------------------------------------------
457  203  662  922  595  698  379  384  397  126
-------------------------------------------------
457  369  921  263  384  676  0    118  955  601
-------------------------------------------------
457  16   857  519  15   516  726  440  735  232
-------------------------------------------------
457  723  378  435  727  176  214  608  354  616
-------------------------------------------------
457  221  763  127  559  617  987  9    702  606
```

可以看到，在进行查询操作时，首先判断对第几列属性进行查找，进行的是范围查找还是精确查找。由于此时 SearchRecord(left,right,col)函数的两个形参 left=right，执行的就是精确查找。进行查找时会去查看是否为该列数据建立过索引，这是第一次查询操作，并不存在索引，所以会对第 1 列创建索引，并返回搜索结果。从图中可以看到，准确的返回了第一列为 457 的所有元组，并且包含了刚刚插入的那一条数据。

下面进行范围查找，找出第二列中所有范围内在 1 到 10 的元组。修改测试代码：

```
int main()
{
    //测试
    table* m_table=table::GetTable();
    m_table->SearchRecord(1,10,2);
}
```

在命令行输入 make 命令重新编译，编译完成后输入./test 执行。

```
问题    输出    调试控制台    终端

cr@LAPTOP-U6G05SCN:~/linux_project$ ./test
正在搜索第2列，范围：[1,10]
已查找到索引文件，正在读取...
搜索结果为：
-------------------------------------------------
398  1    983  936  329  838  524  545  277  784
-------------------------------------------------
95   1    702  435  612  185  335  297  257  319
-------------------------------------------------
253  2    679  546  305  572  493  961  154  464
-------------------------------------------------
10   2    975  576  609  426  791  146  989  799
-------------------------------------------------
408  2    872  124  515  728  534  728  156  105
-------------------------------------------------
365  2    653  230  496  287  813  71   22   791
-------------------------------------------------
80   2    450  145  418  286  693  618  386  112
-------------------------------------------------
903  2    252  433  979  158  850  631  621  886
-------------------------------------------------
390  3    907  250  323  382  773  198  484  599
-------------------------------------------------
21   3    985  243  699  434  149  292  3    527
```

由于该表中存储了大量的数据，我们只返回前 10 条查询结果方便查看。可以看到在查询时首先判断该查询为范围查找，由于并没有在第二列建立过索引，索引会创建索引文件，再进行查找并返回结果。此时查看我们的工作目录：

```
cr@LAPTOP-U6G05SCN:~/linux_project$ ls
BPlusTree.cpp  BPlusTree.o        CLExeFuncProvider.h  CLExecutive.h  CLThread.h  Makefile  index_1  main
BPlusTree.h    CLExeFuncInsert.h  CLExeFuncSearch.h    CLThread.cpp   CLThread.o  data.bat  index_2  main.cpp
cr@LAPTOP-U6G05SCN:~/linux_project$
```

由于刚刚的两次查询操作，生成了两个索引文件，供下次查询时读取，实现快速的搜索。下面重新执行对第二列的范围查询，查看是否返回相同结果：

```
cr@LAPTOP-U6G05SCN:~/linux_project$ ./test
正在搜索第2列，范围：[1,10]
已查找到索引文件，正在读取...
搜索结果为：
-------------------------------------------------
398  1    983  936  329  838  524  545  277  784
-------------------------------------------------
95   1    702  435  612  185  335  297  257  319
-------------------------------------------------
253  2    679  546  305  572  493  961  154  464
-------------------------------------------------
10   2    975  576  609  426  791  146  989  799
-------------------------------------------------
408  2    872  124  515  728  534  728  156  105
-------------------------------------------------
365  2    653  230  496  287  813  71   22   791
-------------------------------------------------
80   2    450  145  418  286  693  618  386  112
-------------------------------------------------
903  2    252  433  979  158  850  631  621  886
-------------------------------------------------
390  3    907  250  323  382  773  198  484  599
-------------------------------------------------
21   3    985  243  699  434  149  292  3    527
```

可以看到，执行查询操作时，检查到了以及建立好的索引文件，直接读取出索引文件，返回查询结果，同样由于结果包含元组过多，只返回前 10 条记录。比较后发现，两次查询操作返回的结果一致。

为了验证索引文件能够正确更新，将前两次测试进行结合。首先向表中插入一条元组。

```
cr@LAPTOP-U6G05SCN:~/linux_project$ ./test
已成功添加一条记录：
-------------------------------------------------
928  623  58   853  833  917  813  538  411  810
```

由于此前，我们在第一列和第二列以及建立了索引，我们需要验证插入数据后，该索引能否成功更新搜索到新插入的数据。我们精确查找第二列数值为 623 的元组。

```
int main()
{
    //测试
    table* m_table=table::GetTable();
    m_table->SearchRecord(623,623,2);
}
```

在命令行输入 make 命令重新编译，编译完成后输入./test 执行。

```
cr@LAPTOP-U6G05SCN:~/linux_project$ ./test
正在搜索第2列，范围：[623,623]
已查找到索引文件，正在读取...
搜索结果为：
-------------------------------------------------
928  623  58   853  833  917  813  538  411  810
-------------------------------------------------
802  623  256  652  762  565  180  526  222  751
```

可以发现在执行查询操作时，先读取了索引文件，第一条返回结果就刚刚插入的记录，证明索引文件更新成功。

3.3 多线程测试

```
4   int main()
5   {
6       //测试
7       table* m_table=table::GetTable();
8       CLExecutiveFunctionProvider* inserter=new
    CLExecutiveFunctionInsert();//插入进程
9       CLExecutiveFunctionProvider* Search=new
    CLExecutiveFunctionSearch(1,10,3); //搜索进程
10      CLExecutive *pThread1=new CLThread(Search);
11      pThread1->Run();
12      pThread1->WaitForDeath();
13      CLExecutive *pThread2=new CLThread(inserter);
14      pThread2->Run();
15      pThread2->WaitForDeath();
16  }
```

修改测试代码，在多线程环境下，使用一个线程向表中插入一条数据，使用另一个线程进行范围查询。可见两个线程分别执行了查找和插入操作。

```
cr@LAPTOP-U6G05SCN:~/linux_project$ ./test

线程ID：140435496621824
正在搜索第3列，范围：[1,10]
未查找到索引文件，正在创建...
已为第3列创建索引文件
搜索结果为：
-------------------------------------------------
886  290  1    344  320  980  211  931  416  740
-------------------------------------------------
354  487  1    482  806  956  679  341  676  580
-------------------------------------------------
643  886  1    761  872  761  593  54   105  520
-------------------------------------------------
437  340  1    569  874  363  336  43   552  685
-------------------------------------------------
335  891  1    748  261  896  76   104  483  170
-------------------------------------------------
15   300  1    331  759  36   673  211  382  247
-------------------------------------------------
262  367  1    812  741  187  405  812  831  961
-------------------------------------------------
545  634  2    596  532  253  703  363  199  587
-------------------------------------------------
477  225  2    129  66   572  427  599  357  528
-------------------------------------------------
973  519  3    248  77   163  652  882  862  934

线程ID：140435496621824
已成功添加一条记录：
-------------------------------------------------
213  755  100  386  564  964  778  43   855  40
```