第十一章　第十一次作业

1. (计算题) 根据以下给出三个基本表 Student、Course、SC，其中 Student（学生表）的字段为学号、姓名、性别、年龄、所属院系；

   Course（课程表）的字段按顺序为课程编号、课程名、先行课程、课程学分；

   SC（选课表）的字段按顺序为学号、课程号、成绩。

   各表的记录如表 1.1、表 1.2、表 1.3 所示：

关系 Student

Sno	Sname	Ssex	Sage	Sdept
95001	李勇	男	20	CS
95002	刘晨	女	19	IS
95003	王明	女	18	MA
95004	张立	男	19	IS

关系 Course

Cno	Cname	Cpno	Ccredit
1	数据库	5	4
2	数学		2
3	信息系统	1	4
4	操作系统	6	3
5	数据结构	7	4
6	数据处理		2
7	PASCAL	6	4

关系 SC

Sno	Cno	Grade
95001	1	92
95001	2	85
95001	3	88
95002	2	90
95003	3	80

首先，使用 C 语言或 Python 语言模拟数据结构和函数：

(1) 假设我们为 SC 表中的 Sno（学号）字段创建了顺序索引，要求使用顺序查找方法从表中查找某个学生（如学号为 95001）的所有选课记录。

编写一个函数，该函数接受一个学号（Sno），并输出该学号对应的所有选课记录。

要求：没有使用哈希表或其他高级索引，直接通过顺序扫描来实现查询。

```python
def question_1_1(sno):
    return [(sno, cno, grade) for (sno, cno, grade) in sc]
```

(2) 假设我们为 Student 表中的 Sno（学号）字段创建了一个 B+ 树索引。请编写一个简单的 B+ 树实现，并使用它来查找某个学号对应的学生信息。

定义一个 B+ 树类，支持插入（Insert）和查找（Search）操作。每个节点最多有 3 个键，最少有 2 个键（阶数为 3）。

实现查找学号为 95002 的学生信息。

由于 B+ 树非常复杂，所以直接使用大模型生成代码了。（题目中说“使用 C 语言或 Python 语言”，实现 B+ 树这种底层的数据结构这里就用 C 了，而题目中又说“定义一个 B+ 树类”，C 语言中没有类的概念，这里就只能模拟一个类了。）

文件 bplus_tree.h

```c
#ifndef BPLUS_TREE_H
#define BPLUS_TREE_H

#include <stdio.h>
#include <stdlib.h>

#define ORDER 3

typedef struct Node {
    int num_keys;
    int keys[ORDER - 1];
    struct Node *children[ORDER];
    struct Node *parent;
    void *values[ORDER - 1]; // Assuming values are pointers to some data
    struct Node *next;       // Pointer to next leaf node
} Node;

typedef struct {
    Node *root;
    Node *first_leaf;
} BPlusTree;

// Function declarations
```

```
BPlusTree* create_bplus_tree();
Node* create_node(int is_leaf);
void insert(BPlusTree *tree, int key, void *value);
Node* search(BPlusTree *tree, int key);
void split_node(Node *node);
void insert_into_parent(Node *left, int key, Node *right);
void print_tree(Node *root);

#endif
```

文件 bplus_tree.c

```
#include "bplus_tree.h"

BPlusTree* create_bplus_tree() {
    BPlusTree *tree = (BPlusTree*)malloc(sizeof(BPlusTree));
    tree->root = NULL;
    tree->first_leaf = NULL;
    return tree;
}

Node* create_node(int is_leaf) {
    Node *node = (Node*)malloc(sizeof(Node));
    node->num_keys = 0;
    for (int i = 0; i < ORDER; i++) {
        node->children[i] = NULL;
    }
    node->parent = NULL;
    node->next = NULL;
    if (is_leaf) {
        node->next = NULL;
    }
    return node;
}

void insert(BPlusTree *tree, int key, void *value) {
    if (tree->root == NULL) {
        tree->root = create_node(1);
        tree->root->keys[0] = key;
        tree->root->values[0] = value;
        tree->root->num_keys = 1;
```

```
        tree->first_leaf = tree->root;
    } else {
        Node *leaf = find_leaf(tree->root, key);
        if (leaf->num_keys < ORDER - 1) {
            int i = leaf->num_keys - 1;
            while (i >= 0 && leaf->keys[i] > key) {
                leaf->keys[i + 1] = leaf->keys[i];
                leaf->values[i + 1] = leaf->values[i];
                i--;
            }
            leaf->keys[i + 1] = key;
            leaf->values[i + 1] = value;
            leaf->num_keys++;
        } else {
            Node *new_leaf = create_node(1);
            new_leaf->keys[0] = key;
            new_leaf->values[0] = value;
            new_leaf->num_keys = 1;

            int virtual_keys[ORDER];
            void *virtual_values[ORDER];
            Node *virtual_children[ORDER];

            int i = 0;
            while (i < ORDER - 1) {
                virtual_keys[i] = leaf->keys[i];
                virtual_values[i] = leaf->values[i];
                i++;
            }
            virtual_keys[i] = key;
            virtual_values[i] = value;

            int split_point = ORDER / 2;

            leaf->num_keys = 0;
            i = 0;
            while (i < split_point) {
                leaf->keys[i] = virtual_keys[i];
                leaf->values[i] = virtual_values[i];
                i++;
                leaf->num_keys++;
            }
```

```
            i = split_point;
            int j = 0;
            while (i < ORDER) {
                new_leaf->keys[j] = virtual_keys[i];
                new_leaf->values[j] = virtual_values[i];
                i++;
                j++;
                new_leaf->num_keys++;
            }

            new_leaf->next = leaf->next;
            leaf->next = new_leaf;

            for (i = leaf->num_keys; i < ORDER - 1; i++) {
                leaf->keys[i] = 0;
                leaf->values[i] = NULL;
            }
            for (i = new_leaf->num_keys; i < ORDER - 1; i++) {
                new_leaf->keys[i] = 0;
                new_leaf->values[i] = NULL;
            }

            if (leaf->parent == NULL) {
                Node *new_root = create_node(0);
                new_root->keys[0] = new_leaf->keys[0];
                new_root->children[0] = leaf;
                new_root->children[1] = new_leaf;
                new_root->num_keys++;
                leaf->parent = new_root;
                new_leaf->parent = new_root;
                tree->root = new_root;
            } else {
                insert_into_parent(leaf, new_leaf->keys[0], new_leaf);
            }
        }
    }
}

Node* find_leaf(Node *root, int key) {
    if (root == NULL) {
        return root;
```

```
    }
    int i = 0;
    while (i < root->num_keys) {
        if (key < root->keys[i]) {
            break;
        }
        i++;
    }
    if (root->children[0] != NULL) {
        return find_leaf(root->children[i], key);
    } else {
        return root;
    }
}

Node* search(BPlusTree *tree, int key) {
    Node *leaf = find_leaf(tree->root, key);
    if (leaf == NULL) {
        return leaf;
    }
    int i = 0;
    while (i < leaf->num_keys) {
        if (leaf->keys[i] == key) {
            return leaf;
        }
        i++;
    }
    return NULL;
}

void split_node(Node *old_node) {
    Node *new_node = create_node(old_node->children[0] != NULL ? 0 : 1);
    int mid = ORDER / 2;
    old_node->num_keys = mid;

    int i = mid;
    int j = 0;
    while (i < ORDER - 1) {
        new_node->keys[j] = old_node->keys[i];
        new_node->values[j] = old_node->values[i];
        if (old_node->children[0] != NULL) {
            new_node->children[j] = old_node->children[i];
```

```
            new_node->children[j]->parent = new_node;
        }
        i++;
        j++;
        new_node->num_keys++;
    }
    if (old_node->children[0] != NULL) {
        new_node->children[j] = old_node->children[i];
        new_node->children[j]->parent = new_node;
    }

    if (old_node->next != NULL) {
        new_node->next = old_node->next;
    }
    old_node->next = new_node;

    if (old_node->parent == NULL) {
        Node *new_root = create_node(0);
        new_root->keys[0] = new_node->keys[0];
        new_root->children[0] = old_node;
        new_root->children[1] = new_node;
        new_root->num_keys++;
        old_node->parent = new_root;
        new_node->parent = new_root;
        tree->root = new_root;
    } else {
        insert_into_parent(old_node, new_node->keys[0], new_node);
    }
}

void insert_into_parent(Node *left, int key, Node *right) {
    Node *parent = left->parent;
    int i = parent->num_keys - 1;
    while (i >= 0 && parent->keys[i] > key) {
        parent->keys[i + 1] = parent->keys[i];
        parent->children[i + 2] = parent->children[i + 1];
        i--;
    }
    parent->keys[i + 1] = key;
    parent->children[i + 2] = right;
    parent->num_keys++;
```

```
    if (parent->num_keys == ORDER) {
        split_node(parent);
    }
}

void print_tree(Node *root) {
    if (root == NULL) {
        return;
    }
    if (root->children[0] == NULL) {
        printf("Leaf Node: ");
        for (int i = 0; i < root->num_keys; i++) {
            printf("%d ", root->keys[i]);
        }
        printf("\n");
    } else {
        for (int i = 0; i <= root->num_keys; i++) {
            print_tree(root->children[i]);
        }
        printf("Internal Node: ");
        for (int i = 0; i < root->num_keys; i++) {
            printf("%d ", root->keys[i]);
        }
        printf("\n");
    }
}

int main() {
    BPlusTree *tree = create_bplus_tree();

    insert(tree, 5, NULL);
    insert(tree, 15, NULL);
    insert(tree, 25, NULL);
    insert(tree, 35, NULL);
    insert(tree, 45, NULL);
    insert(tree, 55, NULL);
    insert(tree, 65, NULL);

    print_tree(tree->root);

    int search_key = 35;
    Node *result = search(tree, search_key);
```

```
    if (result != NULL) {
        printf("Key %d found in the tree.\n", search_key);
    } else {
        printf("Key %d not found in the tree.\n", search_key);
    }

    return 0;
}
```

查找学号为 95002 的学生信息就应该使用

```
find_leaf(tree, 95002)
```

(3) 假设我们为 SC 表中的 Cno（课程编号）字段创建了散列索引，要求使用散列索引来查找某个课程（如课程编号为 2）的所有选课记录。

编写一个函数，该函数接受课程编号（Cno），并输出该课程编号对应的所有选课记录。

要求：使用散列表的方式将课程编号（Cno）映射到槽位，并在槽位中存储选课记录。

```
def question_1_3(cno):
    # 这里的 sc 应该是一个字典（使用的是散列索引）
    return sc[cno]
```

2. (计算题)

1. **Employee（员工信息表）：**

EmpID	Name	Dept	Position	Salary
1	张三	IT	程序员	8000
2	李四	HR	人事专员	5000
3	王五	IT	高级程序员	10000
4	赵六	财务	会计	6000
5	周七	IT	实习生	4000
6	钱八	HR	人事经理	9000
7	孙九	财务	财务主管	11000
8	孙十	市场	市场经理	8500

2. Department（部门信息表）：

DeptID	DeptName
1	IT
2	HR
3	财务
4	市场

3.SalaryHistory（薪资历史表）：

EmpID	SalaryDate	Salary
1	2024-01-01	8000
2	2024-01-01	5000
3	2024-01-01	10000
4	2024-01-01	6000
5	2024-01-01	4000
6	2024-01-01	9000
7	2024-01-01	11000
8	2024-01-01	8500

使用 SQL 语言实现：

(1) 在 Employee 表的 EmpID 列上创建聚集索引，并查询员工编号为 3 的员工信息。

```
create index index_id on Employee(EmpID);
select * from Employee where EmpID = 3;
```

(2) 创建一个索引，加速员工薪资在 5000 到 9000 之间的员工信息的查询。

```
create index index_salary on Employee(Salary);
select * from Employee where Salary between 5000 and 9000;
```

(3) 创建一个索引，加速部门为 IT 的所有员工信息的查询。

```
create index index_dept on Employee(Dept);
select * from Employee where Dept = 'IT';
```

(4) 在 Employee 表上创建多列索引，查询部门为 HR 且职位为人事经理的员工信息。

```
create index index_dept_position on Employee(Dept, Position);
select * from Employee where Dept = 'HR' and Position = '人事经理';
```

3. (判断题) 一个表只能建一个索引和一个聚簇。()

A. 对　　B. [错]

4. (判断题) 聚集索引与普通索引相比，数据的插入、更新和删除效率更高。()

A. 对　　B. [错]

5. (判断题) 索引有助于提高数据检索的速度，因此建立索引的数量越多越好。()

A. 对　　B. [错]

6. (判断题) 对一个基本表需要建立多个聚集索引。()

A. 对　　B. [错]

7. (判断题) 对一个基本表建立索引不会改变基本表中的数据。()

A. [对]　　B. 错

8. (简答题) B+ 索引与散列索引的区别。

B+ 索引会在插入、删除记录时自动调整索引结构，使得查询任何一个值的时间复杂度相同。而散列索引只是在插入记录时计算散列值，选择一个合适的桶，在桶中还是需要顺序搜索，因此桶的数量会影响性能。

9. (简答题) 请简述稠密索引与稀疏索引的区别，并说明在什么情况下使用稠密索引和稀疏索引比较合适?

稠密索引是文件中的每个搜索码值都有一个索引记录，稀疏索引是只为搜索码的某些值建立索引记录。稠密索引定位一条记录的速度比较快，稀疏索引插入和删除时所需的空间及维护开销较小。

在查找操作较多时，适合使用稠密索引。在插入和删除的操作较多时，适合使用稀疏索引。