

《数据库系统》实验报告

实验名称：数据库操作的实现算法

成 绩：

专业班级：

姓 名：

学 号：

实 验 日 期：2018 年 05 月 21 日

实验报告日期：2018 年 05 月 23 日

一、实验目的

掌握 B 树索引查找算法，多路归并排序算法，并用高级语言实现

二、实验内容

选择熟悉的高级语言设计实现归并排序和 B 树索引。

具体要求如下：

- 1) 随机生成具有 1,000,000 条记录的文本文件，每条记录的长度为 16 字节。

属性 A(4 字节整数)	属性 B (12 字节字符串)
--------------	-----------------

- 2) 其中包含两个属性 A 和 B。A 为 4 字节整型， B 为 12 字节字符串，属性值 A 随机生成，属性值 B 自己定义并填充。
- 3) 针对属性 A，用高级语言实现多路归并排序算法。
- 4) 用于外部归并排序的内存空间不大于 1MB。
- 5) 以属性 A 为键值，实现 B 树索引。完成索引的插入，删除和查找操作。

三、实验结果

1. 外部归并排序

所谓外排序，顾名思义，即是在内存外面的排序，因为当要处理的数据量很大，而不能一次装入内存时，此时只能放在读写较慢的外存储器（通常是硬盘）上。

外排序通常采用的是一种“排序-归并”的策略。

- 在排序阶段，先读入能放在内存中的数据量，将其排序输出到一个临时文件，依此进行，将待排序数据组织为多个有序的临时文件；
- 尔后在归并阶段将这些临时文件组合为一个大的有序文件，也即排序结果。

假定现在有 20 个数据的文件 A：{5 11 0 18 4 14 9 7 6 8 12 17 16 13 19 10 2 1 3 15}，但一次只能使用仅装 4 个数据的内容，所以，我们可以每趟对 4 个数据进行排序，即 5 路归并，具体方法如下述步骤：

- 我们先把“大”文件 A，分割为 a1，a2，a3，a4，a5 等 5 个小文件，每个小文件 4 个数据
 - a1 文件为：5 11 0 18
 - a2 文件为：4 14 9 7
 - a3 文件为：6 8 12 17

- a4 文件为: 16 13 19 10
 - a5 文件为: 2 1 3 15
- 然后依次对 5 个小文件分别进行排序
 - a1 文件完成排序后: 0 5 11 18
 - a2 文件完成排序后: 4 7 9 14
 - a3 文件完成排序后: 6 8 12 17
 - a4 文件完成排序后: 10 13 16 19
 - a5 文件完成排序后: 1 2 3 15
- 最终多路归并, 完成整个排序
 - 整个大文件 A 文件完成排序后: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

k 路归并算法其实整体的思路并不复杂。我们常用的两种方法一种是建立一个最小 k 堆, 然后每次取最小的元素, 也就是堆顶的元素。然后再调整。还有一种就是建立一棵胜者树。其实思路也类似, 每次取最顶端的元素, 这个元素就是胜者, 也就是最小的那个元素。然后从胜者树所在的叶节点对应的序列里取下一个元素。然后再进行比较向上调整。这两种方法都有一个需要注意的地方就是需要根据当前的操作节点来确定该节点所处的序列。从某种角度来说, k 路归并算法对于处理大规模的数据有非常重要的意义。

1.1 效率分析

由于共有 1000000 个元组, 每个元组 16B, 因此总大小为 16M, 又内存限制为 1M, 则至少要分成 16 组。为了尽可能的减少 I/O 次数, 因此采用 16 路归并的方法; 为了减少归并的时间, 采用败者树, 在 $O(\lg n)$ 的时间复杂度内选出最小的 A 属性元组。

2 B 树

2.1 B 树产生的原因

B 树是一种查找树, 我们知道, 这一类树 (比如二叉查找树, 红黑树等等) 最初生成的目的都是为了解决某种系统中, 查找效率低的问题。B 树也是如此, 它最初启发于二叉查找树, 二叉查找树的特点是每个非叶节点都只有两个孩子节点。然而这种做法会导致当数据量非常大时, 二叉查找树的深度过深, 搜索算法自根节点向下搜索时, 需要访问的节点也就变的相当多。如果这些节点存储在外存储器中, 每访问一个节点, 相当于就是进行了一次 I/O 操作, 随着树高度的增加, 频繁的 I/O 操作一定会降低查询的效率。

这里有一个基本的概念, 就是说我们从外存储器中读取信息的步骤, 简单来分, 大致有两步:

1. 找到存储这个数据所对应的磁盘页面, 这个过程是机械化的过程, 需要依靠磁臂的转动, 找到对应磁道, 所以耗时长。

2. 读取数据进内存, 并实施运算, 这是电子化的过程, 相当快。

综上, 对于外存储器的信息读取最大的时间消耗在于寻找磁盘页面。那么一个基本的想法就

是能不能减少这种读取的次数，在一个磁盘页面上，多存储一些索引信息。B 树的基本逻辑就是这个思路，它要改二叉为多叉，每个节点存储更多的指针信息，以降低 I/O 操作数。

2.2 B 树的定义

有关于 B 树概念的定义，不同的资料在表述上有所差别。我在这里采用《算导》中的定义，用最小度 t 来定义 B 树。一棵最小度为 t 的 B 树是满足如下四个条件的平衡多叉树：

1. 每个节点最多包含 $2t-1$ 个关键字；除根节点外的每个节点至少有 $t-1$ 个关键字（ $t \leq 2$ ），根节点至少有一个关键字；

2. 一个节点 u 中的关键字按非降序排列： $u.key_1 \leq u.key_2 \leq \dots \leq u.key_n$ ；

3. 每个节点的关键字对其子树的范围分割。设节点 u 有 $n+1$ 个指针，指向其 $n+1$ 棵子树，指针为 $u.p_1, \dots, u.p_n$ ，关键字 k_i 为 $u.p_i$ 所指的子树中的关键字，有 $k_1 \leq u.key_1 \leq k_2 \leq u.key_2 \dots$ 成立；

4. 所有叶子节点具有相同的深度，即树的高度 h 。这表明 B 树是平衡的。平衡性其实正是 B 树名字的来源，B 表示的正是单词 Balanced；

2.3 插入操作

插入一个元素时，首先在 B 树中是否存在，如果不存在，即在叶子结点处结束，然后在叶子结点中插入该新的元素，注意：如果叶子结点空间足够，这里需要向右移动该叶子结点中大于新插入关键字的元素，如果空间满了以致没有足够的空间去添加新的元素，则将该结点进行“分裂”，将一半数量的关键字元素分裂到新的其相邻右结点中，中间关键字元素上移到父结点中（当然，如果父结点空间满了，也同样需要“分裂”操作），而且当结点中关键元素向右移动了，相关的指针也需要向右移。如果在根结点插入新元素，空间满了，则进行分裂操作，这样原来的根结点中的中间关键字元素向上移动到新的根结点中，因此导致树的高度增加一层。

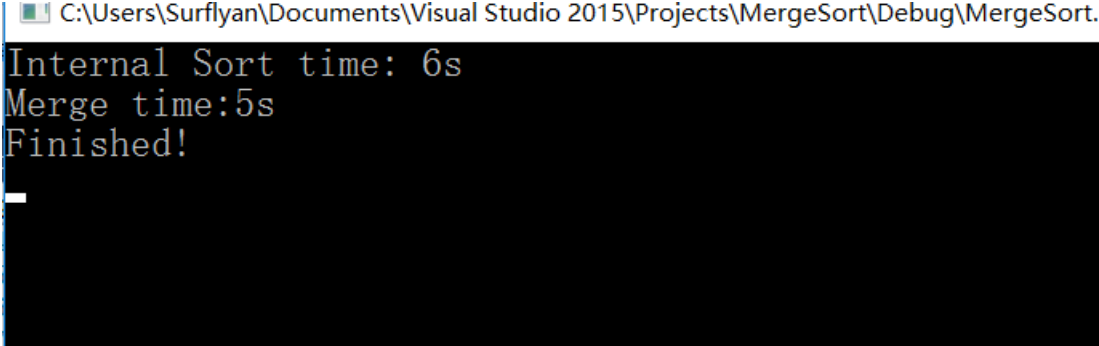
2.4 删除操作

首先查找 B 树中需删除的元素，如果该元素在 B 树中存在，则将该元素在其结点中进行删除，如果删除该元素后，首先判断该元素是否有左右孩子结点，如果有，则上移孩子结点中的某相近元素（“左孩子最右边的节点”或“右孩子最左边的节点”）到父结点中，然后是移动之后的情况；如果没有，直接删除后，移动之后的情况。

删除元素，移动相应元素之后，如果某结点中元素数目（即关键字数）小于 $\text{ceil}(m/2)-1$ ，则需要看其某相邻兄弟结点是否丰满（结点中元素个数大于 $\text{ceil}(m/2)-1$ ）。如果丰满，则向父节点借一个元素来满足条件；如果其相邻兄弟都刚脱贫，即借了之后其结点数目小于 $\text{ceil}(m/2)-1$ ，则该结点与其相邻的某一兄弟结点进行“合并”成一个结点。

3.实验结果

外部归并排序



```
C:\Users\Surflyan\Documents\Visual Studio 2015\Projects\MergeSort\Debug\MergeSort.  
Internal Sort time: 6s  
Merge time: 5s  
Finished!
```

排序前文件

排序后文件

```
data.txt (~\Documents\Visual Studio 2015) data-sorted.txt (~\Documents\Visual Studio 20
1: data.txt 1: data-sorted.txt
1 1裙肋评逊辱潭 1 1裙肋评逊辱潭
2 2012辙讷丸忆这神 2 2一秋畏绥晾仁
3 3仟深宿习味企 3 3仟深宿习味企
4 18005螺芯严獭茸蚤 4 4瞥漾畏妻述隙
5 5惕履锰豫稀沔 5 5惕履锰豫稀沔
6 6虞它颇柳裳蒸 6 6虞它颇柳裳蒸
7 8013泰稳谭胃小谣 7 7图莎乙隋澡胀
8 3018拳萆莎峡刃欧 8 8孙嘶雾裘磐汝
9 57039批销须印敲市 9 9且涂设略悄蘑
10 10廖屈倚熏青箠 10 10廖屈倚熏青箠
11 22001品臀脓深淖蜗 11 11倘辕柳鹿凄绍
12 12弥永掠迁荧远 12 12弥永掠迁荧远
13 13吻衷枢橡圣杂 13 13吻衷枢橡圣杂
14 14吕樱怒漂写州 14 14吕樱怒漂写州
15 61037氧曙祁迂指血 15 15扰异杨园映末
16 16懈扭律塙露躺 16 16懈扭律塙露躺
17 26迈吕受修磷命 17 17扞脑夏娶瓢芯
18 18淫郑搪筌葡双 18 18淫郑搪筌葡双
19 19赁茺桐七侵立 19 19赁茺桐七侵立
20 20溪诈弯轮外祥 20 20溪诈弯轮外祥
21 72100惺妥蝎涂宋盐 21 21虚哦疡声宜伦
22 22条攘迂皖卿啼 22 22条攘迂皖卿啼
23 14001谱镣兆菩省批 23 23拎淤漂森谅尹
24 22022量乌秀笑氛歧 24 24吮谈朋裴糯数
25 25瞬泄沃墅蚀陋 25 25瞬泄沃墅蚀陋
26 29005踢帘辙芽提胖 26 26迈吕受修磷命
27 27萧穆摘萆咱唆 27 27萧穆摘萆咱唆
28 33000演瞞崖郑郑全 28 28每徐戡娜酖校
29 29僻外强沫箱露 29 29僻外强沫箱露
30 30瑞碎谄偏虜笑 30 30瑞碎谄偏虜笑
31 31坪肆言塌遥幼 31 31坪肆言塌遥幼
32 32僂迂所莖展肆 32 32僂迂所莖展肆
NORMAL data.txt NORMAL data-sorted.txt
"~\Documents\Visual Studio 2015" "~\Documents\Visual Studio 2015\Pr
```

B 树插入结点

C:\Users\Surflyan\Documents\Visual Studio 2015\Projects\B-Tree\Debug\B-Tree.exe

请输入B树阶数 ($3 < m < 20$):

4


创建B树成功! 共插入 999999 条记录!

1. 创建B树
 2. 查找结点
 3. 删除结点
 4. 销毁B树
- 请输入: _

B 树查找结点

```
请输入关键字:  
56  
56 躯姿庭曼妈妹  
1. 创建B树  
2. 查找结点  
3. 删除结点  
4. 销毁B树  
请输入:
```

B 树删除结点

 C:\Users\Surflyan\Documents\Visual Studio 2015\Projects\B-Tree\Debug\B-Tree.e:

```
请输入关键字:  
56  
删除成功!  
1. 创建B树  
2. 查找结点  
3. 删除结点  
4. 销毁B树  
请输入:
```

```
请输入关键字:  
56  
该关键字不存在!  
1. 创建B树  
2. 查找结点  
3. 删除结点  
4. 销毁B树  
请输入:
```

四、程序代码

MergeSort

```
#include <iostream>
```

```

#include <fstream>
#include <ctime>
#include <cstdlib>
#include <string.h>
#include <assert.h>
#include <string>
using namespace std;

typedef struct Data
{
    int data;
    char str[13];
} Data;

const int N = 1000000;    //数据总量
const int FILE_NUM = 16; //文件个数
const int MAX_PART = 62500; //每一个文件大小

FILE *fpreads[FILE_NUM];
const int MIN = -1;      //最小值
const int MAX = 2147483647; //最大值

int cmp(const void* a, const void *b)
{
    if ((*Data*)a).data > ((*Data *)b).data)
        return 1;
    else if ((*Data*)a).data < ((*Data *)b).data)
        return -1;
    else
        return 0;
}

/*
读取数据
每项数据格式
int (4byte) char (12) +"\0"
*/

int readData(FILE *fp, Data *array, int N)
{
    int length = 0;
    for (int i = 0; i < MAX_PART && (EOF != fscanf(fp, "%d %s\n", &array[i].data,

```



```

&array[i].str)); i++)
{
    length++;
}
return length;
}

```

```

FILE* openFile(int count, char *mode)
{
    FILE *fpwrite = NULL;
    char filename[20];
    memset(filename, 0, 20);
    sprintf_s(filename, 20, "data%d.txt", count);
    fopen_s(&fpwrite, filename, mode);
    assert(fpwrite != NULL);
    return fpwrite;
}

```

/*

向临时文件写入已排序数据

*/

```

void writeData(Data *array, int N, int count)
{

```

```

    FILE *fpwrite = openFile(count, "w");
    int i = 0;
    for (i = 0; i < N; i++)
    {
        fprintf(fpwrite, "%d%s\n", array[i].data, array[i].str);
    }

```

fprintf(fpwrite, "%d%s\n", MAX, "这是结束标志"); //在每个文件最后写入一个最大值，表示文件结束

```

    fclose(fpwrite);
}

```

/*

利用快排对每个文件进行排序

T:O(n * lgn)

S:O(lgn)

*/

```

void internalSort(void)
{

```

```

    clock_t begin = clock();
    FILE *fpread = NULL;

```

```

fopen_s(&fpread, "data.txt", "r");
assert(fpread != NULL);

int count = 0;
Data *array = new Data[MAX_PART];
assert(array != NULL);
while (1)
{
    memset(array, 0, sizeof(Data)* MAX_PART);
    int length = readData(fpread, array, MAX_PART);
    if (length == 0)
    {
        break;
    }
    qsort(array, length, sizeof(Data), cmp);
    writeData(array, length, count);
    count++;
}
delete[] array;
fclose(fpread);
clock_t end = clock();
cout << "Internal Sort time: " << (end - begin) / CLK_TCK << "s" << endl;
}

/*
败者树调整
*/
void adjust(int ls[], Data data[], int s)
{
    int t = (s + FILE_NUM) / 2;
    while (t)
    {
        if (data[s].data > data[ls[t]].data)
        {
            int temp = s;
            s = ls[t];
            ls[t] = temp;
        }
        t /= 2;
    }
    ls[0] = s;
}

/*

```

建立败者树

*/

```
void buildLoserTree(int ls[], Data data[])
{
    data[FILE_NUM].data = MIN;
    for (int i = 0; i < FILE_NUM; i++)
    {
        ls[i] = FILE_NUM;
    }
    for (int i = FILE_NUM - 1; i >= 0; i--)
    {
        adjust(ls, data, i);
    }
}

void MergeSortMain()
{
    clock_t begin = clock();
    FILE *fpreads[FILE_NUM];      //10个文件的描述符
    Data data[FILE_NUM + 1];      //10个文件的10个当前最小数据
    int ls[FILE_NUM];             //存放败者索引的节点
    int index;
    FILE *fpwrite = NULL;
    fopen_s(&fpwrite, "data-sorted.txt", "w");
    assert(fpwrite != NULL);

    for (int i = 0; i < FILE_NUM; i++)
    {
        fpreads[i] = openFile(i, "r");
    }
    for (int i = 0; i < FILE_NUM; i++)
    {
        fscanf(fpreads[i], "%d %s\n", &data[i].data, &data[i].str);
    }

    buildLoserTree(ls, data); //创建败者树

    while (data[ls[0]].data != MAX)
    {
        index = ls[0];
        fprintf(fpwrite, "%d%s\n", data[index].data, data[index].str);
        fscanf(fpreads[index], "%d %s\n", &data[index].data, &data[index].str);
        adjust(ls, data, index);
    }
}
```

```

    for (int i = 0; i < FILE_NUM; i++)
    {
        fclose(fpreads[i]);
    }
    fclose(fpwrite);
    clock_t end = clock();
    cout << "Merge time:" << (end - begin) / CLK_TCK << "s" << endl;
}

int main()
{
    internalSort();
    MergeSortMain();
    cout << "Finished!" << endl;
    getchar();
    return 0;
}

```

B-Tree

部分代码

```

#pragma once
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

#define TRUE 1
#define FALSE 0
#define OVERFLOW -1
#define OK 1
#define ERROR 0
#define M 20//定义阶数最大值

typedef int KeyType;
typedef int Status;
typedef struct { //记录的结构定义
    KeyType key;
    char data[13];
}Record;

```

```

typedef struct BTreeNode {           //B树结点类型定义
    int keynum;           //结点中关键字个数，即结点的大小
    KeyType key[M + 1]; //关键字，key[0]未用
    struct BTreeNode *parent; //双亲结点指针
    struct BTreeNode *ptr[M + 1]; //孩子结点指针数组
    char *data;           //存储数据
}BTreeNode, *BTree; //B树结点和B树类型

typedef struct {
    BTree pt; //指向找到的结点
    int i; //1<=i<=m, 在结点中的关键字位序
    int tag; //1:查找成功，0:查找失败
}result, *resultPtr; //B树的查找结果类型

//接口定义

void CreatBTree(BTree&T, int m);
/*
初始条件:初始化关键字个数n大于等于0，B树的阶数m大于3小于等于20
操作结果:构建一颗阶数为m, 含有n个关键字的B树
*/
void SearchBTree(BTree T, int k, result &r);
/*
初始条件:树T存在
操作结果:在m阶B数T上查找关键字k，返回p{pt, i, tag}
*/

void InsertBTree(BTree &T, int k, BTree q, int i, int m);
/*
初始条件:树T存在
操作结果:在B树T上结点p->pt的key[i]和key[i+1]之间插入关键字k
*/

void DeleteBTree(BTree p, int i, int m, BTree &T);
/*
初始条件:B树上p结点存在
操作结果:删除B树T上结点p->pt的关键字k

```

```
*/
```

```
FILE* openFile()
```

```
{  
    FILE *fpwrite = NULL;  
    char filename[20]="data.txt";  
    fopen_s(&fpwrite, filename, "r");  
    assert(fpwrite != NULL);  
    return fpwrite;  
}
```

```
void CreatBTree(BTree &T, int m) { //构建一颗阶数为m, 含有n个关键字的B树  
(3<=m<=M, 0<=n<=10000)
```

```
                //创建B树
```

```
    int i;  
    resultPtr p = NULL;  
    p = (result*)malloc(sizeof(result));  
  
    FILE *fp = openFile();  
    int key;  
    char data[13];  
    for (i = 0; EOF != fscanf(fp, "%d %s\n", &key, &data); i++) {  
  
        SearchBTree(T, key, *p); //查找i插入位置  
        InsertBTree(T, key, p->pt, p->i, m); //进行插入  
    }
```

```
    printf("创建B树成功! 共插入 %d 条记录! \n", i);
```

```
}
```

```
void SearchBTree(BTree T, int k, result &r) {
```

```
    //在m阶B树T上查找关键字k, 返回(pt, i, tag)  
    //若查找成功, 则特征值tag=1, 指针pt所指结点中第i个关键字等于k; 否则  
    //特征值tag=0, 等于k的关键字记录应插入在指针pt所指结点中第i-1个和第i个关键字间
```

```
    int i = 0, found = 0;  
    BTree p = T, q = NULL;  
    while (p != NULL && 0 == found) {  
        i = Search(p, k); //在p->key[1..keynum]中查找p->key[i-1]<k<=p->key[i]  
        if (i > 0 && p->key[i] == k)  
            found = 1; //找到待查关键字  
        else {
```

```

        q = p;
        p = p->ptr[i - 1];
    }
}

if (l == found) { //查找成功
    r.pt = p;
    r.i = i;
    r.tag = 1;
}
else { //查找不成功，返回key的插入位置i
    r.pt = q;
    r.i = i;
    r.tag = 0;
}
}

void split(BTree &q, int s, BTree &ap) {
    //将q结点分裂成两个结点，前半保留，后半移入新结点ap
    int i, j, n = q->keynum;
    ap = (BTreeNode*)malloc(sizeof(BTreeNode)); //生成新结点ap
    ap->ptr[0] = q->ptr[s];
    for (i = s + 1, j = 1; i <= n; i++, j++) { //后半移入ap结点
        ap->key[j] = q->key[i];
        ap->ptr[j] = q->ptr[i];
    }
    ap->keynum = n - s;
    ap->parent = q->parent;
    for (i = 0; i <= n - s; i++) {
        if (ap->ptr[i])
            ap->ptr[i]->parent = ap; //将ap所有孩子结点指向ap
    }
    q->keynum = s - 1; //q结点的前一半保留，修改keynum
}

void newroot(BTree &T, BTree p, int x, BTree ap) { //生成新的根结点
    T = (BTreeNode*)malloc(sizeof(BTreeNode));
    T->keynum = 1;
    T->ptr[0] = p;
    T->ptr[1] = ap;
    T->key[1] = x;
    if (p != NULL) p->parent = T;
    if (ap != NULL) ap->parent = T;
}

```

```

    T->parent = NULL; //新根的双亲是空指针
}

void Insert(BTree &q, int i, int x, BTree ap) { //x和ap分别插到q->key[i]和q->ptr[i]
    int j, n = q->keynum;
    for (j = n; j >= i; j--) {
        q->key[j + 1] = q->key[j]; //关键字指针向后移一位
        q->ptr[j + 1] = q->ptr[j]; //孩子结点指针向后移一位
    }
    q->key[i] = x; //赋值
    q->ptr[i] = ap;
    if (ap != NULL) ap->parent = q;
    q->keynum++; //关键字数+1
}

void InsertBTree(BTree &T, int k, BTree q, int i, int m) {
    //在B树T上q结点的key[i-1]和key[i]之间插入关键字k
    //若引起结点过大,则沿双亲指针进行必要的结点分裂调整,使T仍是m阶的B树
    int x, s, finished = 0, neednewroot = 0;
    BTree ap;
    if (NULL == q) //q为空, 则新建根结点
        newroot(T, NULL, k, NULL);
    else {
        x = k;
        ap = NULL;
        while (0 == neednewroot && 0 == finished) {
            Insert(q, i, x, ap); //key和ap分别插到q->key[i]和q->ptr[i]
            if (q->keynum < m) finished = 1; //插入完成
            else { //分裂q结点
                s = (m + 1) / 2;
                split(q, s, ap);
                x = q->key[s];
                if (q->parent != NULL) {
                    q = q->parent;
                    i = Search(q, x); //在双亲结点中查找x的插入位置
                }
                else neednewroot = 1;
            }
        }
    } //while
    if (1 == neednewroot) //T是空树或者根结点已分裂为q和ap结点
        newroot(T, q, x, ap); //生成含信息(q, x, ap)的新的根结点T
}

```



```
}
```

```
void Successor(BTree &p, int i) { //由后继最下层非终端结点的最小关键字代替结点中关键字key[i]。
```

```
    BTreeNode *temp;
    temp = p->ptr[i];
    for (; NULL != temp->ptr[0]; temp = temp->ptr[0]); //找出关键字的后继
    p->key[i] = temp->key[1];
    p = temp;
}
```

```
void Remove(BTree &p, int i) { //从结点p中删除key[i]
```

```
    int j;
    int n = p->keynum;
    for (j = i; j < n; j++) { //关键字左移
        p->key[j] = p->key[j + 1];
        p->ptr[j] = p->ptr[j + 1];
    }
    p->keynum--;
}
```

```
void Restore(BTree &p, int i, int m, BTree &t) { //调整B树
```

```
    int j;
    BTree ap = p->parent;
    BTree lc, rc, pr;
    int finished = 0, r = 0;
    while (0 == finished) {
        r = 0;
        while (ap->ptr[r] != p) //确定p在ap子树的位置
            r++;
        if (r == 0) {
            r++;
            lc = NULL;
            rc = ap->ptr[r];
        }
        else if (r == ap->keynum) {
            rc = NULL;
            lc = ap->ptr[r - 1];
        }
        else {
            lc = ap->ptr[r - 1];

```

```

        rc = ap->ptr[r + 1];
    }
    if (r > 0 && lc != NULL && (lc->keynum > (m - 1) / 2)) { //向左兄弟借关键字
        p->keynum++;
        for (j = p->keynum; j > 1; j--) { //结点关键字右移
            p->key[j] = p->key[j - 1];
            p->ptr[j] = p->ptr[j - 1];
        }
        p->key[1] = ap->key[r]; //父亲插入到结点
        p->ptr[1] = p->ptr[0];
        p->ptr[0] = lc->ptr[lc->keynum];
        if (NULL != p->ptr[0]) //修改p中的子女的父结点为p
            p->ptr[0]->parent = p;
        ap->key[r] = lc->key[lc->keynum]; //左兄弟上移到父亲位置
        lc->keynum--;
        finished = 1;
        break;
    }
    else if (ap->keynum > r && rc != NULL && (rc->keynum > (m - 1) / 2)) {
        p->keynum++;
        p->key[p->keynum] = ap->key[r]; //父亲插入到结点
        p->ptr[p->keynum] = rc->ptr[0];
        if (NULL != p->ptr[p->keynum]) { //修改p中的子女的父结点为p
            p->ptr[p->keynum]->parent = p;
        }
        ap->key[r] = rc->key[1]; //右兄弟上移到父亲位置
        rc->ptr[0] = rc->ptr[1];
        for (j = 1; j < rc->keynum; j++) { //右兄弟结点关键字左移
            rc->key[j] = rc->key[j + 1];
            rc->ptr[j] = rc->ptr[j + 1];
        }
        rc->keynum--;
        finished = 1;
        break;
    }
    r = 0;
    while (ap->ptr[r] != p) r++; //重新确定p在ap子树的位置
    if (r > 0 && (ap->ptr[r - 1]->keynum <= (m - 1) / 2)) { //与左兄弟合并
        lc = ap->ptr[r - 1];
        p->keynum++;
        for (j = p->keynum; j > 1; j--) { //将p结点关键字和指针右移1位
            p->key[j] = p->key[j - 1];
            p->ptr[j] = p->ptr[j - 1];
        }
    }
}

```

中

```
p->key[1] = ap->key[r]; //父结点的关键字与p合并
p->ptr[1] = p->ptr[0]; //从左兄弟右移一个指针
ap->ptr[r + 1] = lc;
for (j = 1; j <= lc->keynum + p->keynum; j++) { //将结点p中关键字移到p左兄弟

    lc->key[lc->keynum + j] = p->key[j];
    lc->ptr[lc->keynum + j] = p->ptr[j];
}
if (p->ptr[0]) { //修改p中的子女的父结点为lc
    for (j = 1; j <= p->keynum; j++) {
        p->ptr[p->keynum + j]->parent = lc;
    }
}
lc->keynum = lc->keynum + p->keynum; //合并后的关键字个数
ap->keynum--;
pr = p;
free(pr); //释放p结点空间
pr = NULL;
p = lc;
}
else { //与右兄弟合并
    rc = ap->ptr[r + 1];
    if (r == 0) r++;
    p->keynum++;
    p->key[p->keynum] = ap->key[r]; //父结点的关键字与p合并
    p->ptr[p->keynum] = rc->ptr[0]; //从右兄弟左移一个指针
    rc->keynum = p->keynum + rc->keynum; //合并后关键字的个数
    ap->ptr[r - 1] = rc;
    for (j = 1; j <= (rc->keynum - p->keynum); j++) { //将p右兄弟的关键字和指针

        rc->key[p->keynum + j] = rc->key[j];
        rc->ptr[p->keynum + j] = rc->ptr[j];
    }
    for (j = 1; j <= p->keynum; j++) { //将结点p中关键字和指针移到p右兄弟中
        rc->key[j] = p->key[j];
        rc->ptr[j] = p->ptr[j];
    }
    rc->ptr[0] = p->ptr[0];
    if (p->ptr[0]) { //修改p中的子女的父结点为rc
        for (j = 1; j <= p->keynum; j++) {
            p->ptr[p->keynum + j]->parent = rc;
        }
    }
    for (j = r; j < ap->keynum; j++) { //将父结点中关键字和指针左移
```

右移

```

        ap->key[j] = ap->key[j + 1];
        ap->ptr[j] = ap->ptr[j + 1];
    }
    ap->keynum--; //父结点的关键字个数减1
    pr = p;
    free(pr); //释放p结点空间
    pr = NULL;
    p = rc;
}
ap = ap->parent;
if (p->parent->keynum >= (m - 1) / 2 || (NULL == ap && p->parent->keynum > 0)) {
    finished = 1;
}
else if (NULL == ap) { //若调整后出现空的根结点，则删除该根结点，树高减1
    pr = T;
    T = p; //根结点下移
    free(pr);
    pr = NULL;
    finished = 1;
}
p = p->parent;
}
}

```

```

void DeleteBTree(BTree p, int i, int m, BTree &T) {
    //删除B树上p结点第i个关键字
    if (p->ptr[i - 1] != NULL) {
        Successor(p, i); //若不是最下层非终端结点
        DeleteBTree(p, 1, m, T); //由后继最下层非终端结点的最小关键字代替它
    }
    else { //若是最下层非终端结点
        Remove(p, i); //从结点p中删除key[i]
        if (p->keynum < (m - 1) / 2) //删除后关键字个数小于(m-1)/2
            Restore(p, i, m, T); //调整B树
    }
}
}

```