《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：1403106

姓　　名：张茗帅

学　　号：1140310606

实 验 日 期 ： 2017 年 6 月 4 日

实验报告日期： 2017 年 6 月 4 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

实验环境：

- 自行选择高级语言

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。

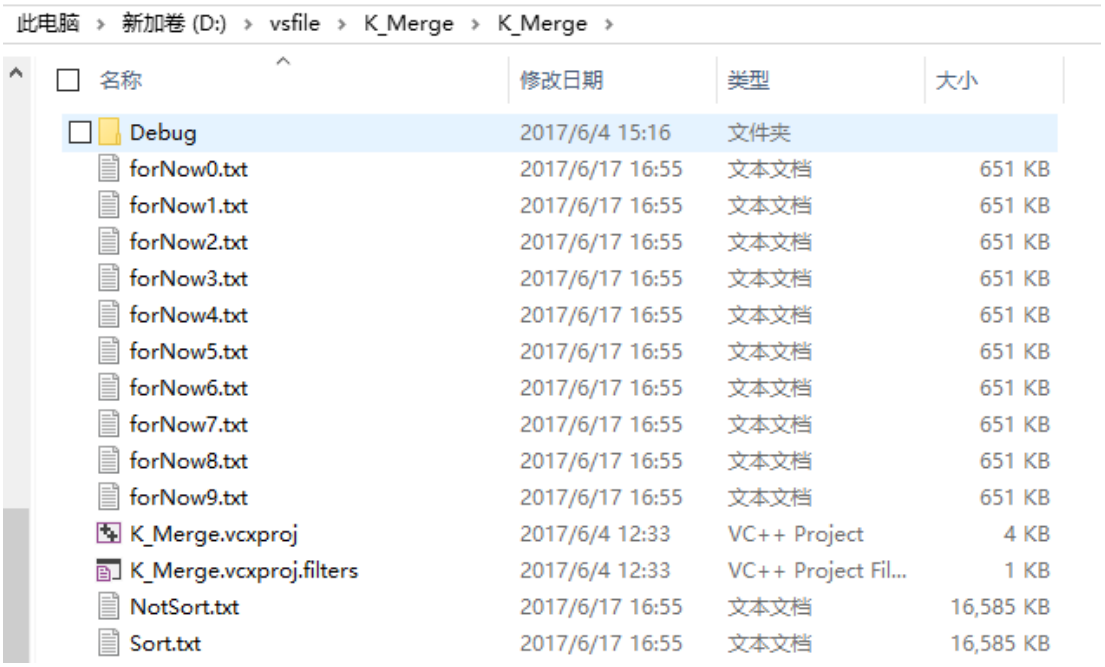
5） 以属性 A 为键值，实现 B 树索引。完成索引的插入，删除和查找操作。

三、实验结果

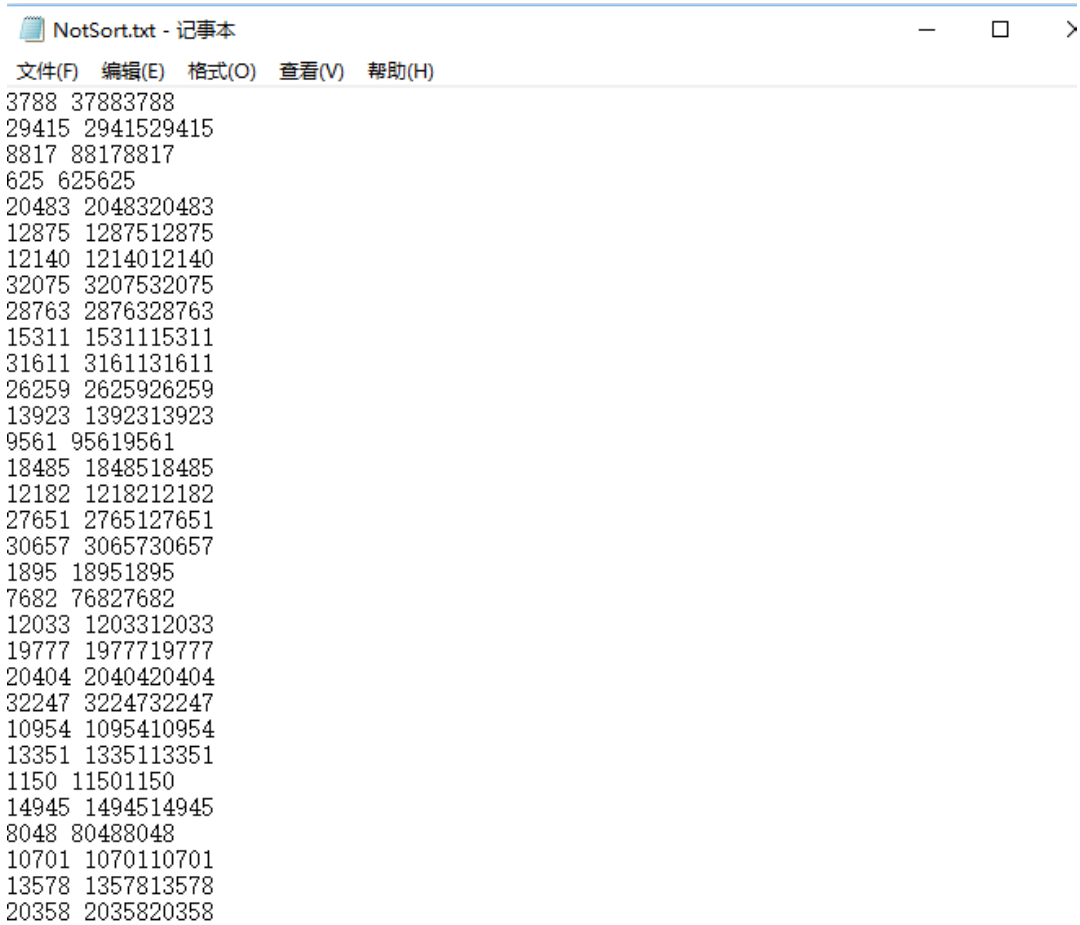
- **截屏保留实验结果**

采用 10 路归并排序

得到的文件结果如下图，首先为未排序的 NotSort 文件，forNow 为每次提取 10w 数据进行排序的结果，一共有一个 forNow 文件，然后对这 10 个 forNow 进行 10-路归并排序最终得到 Sort.txt

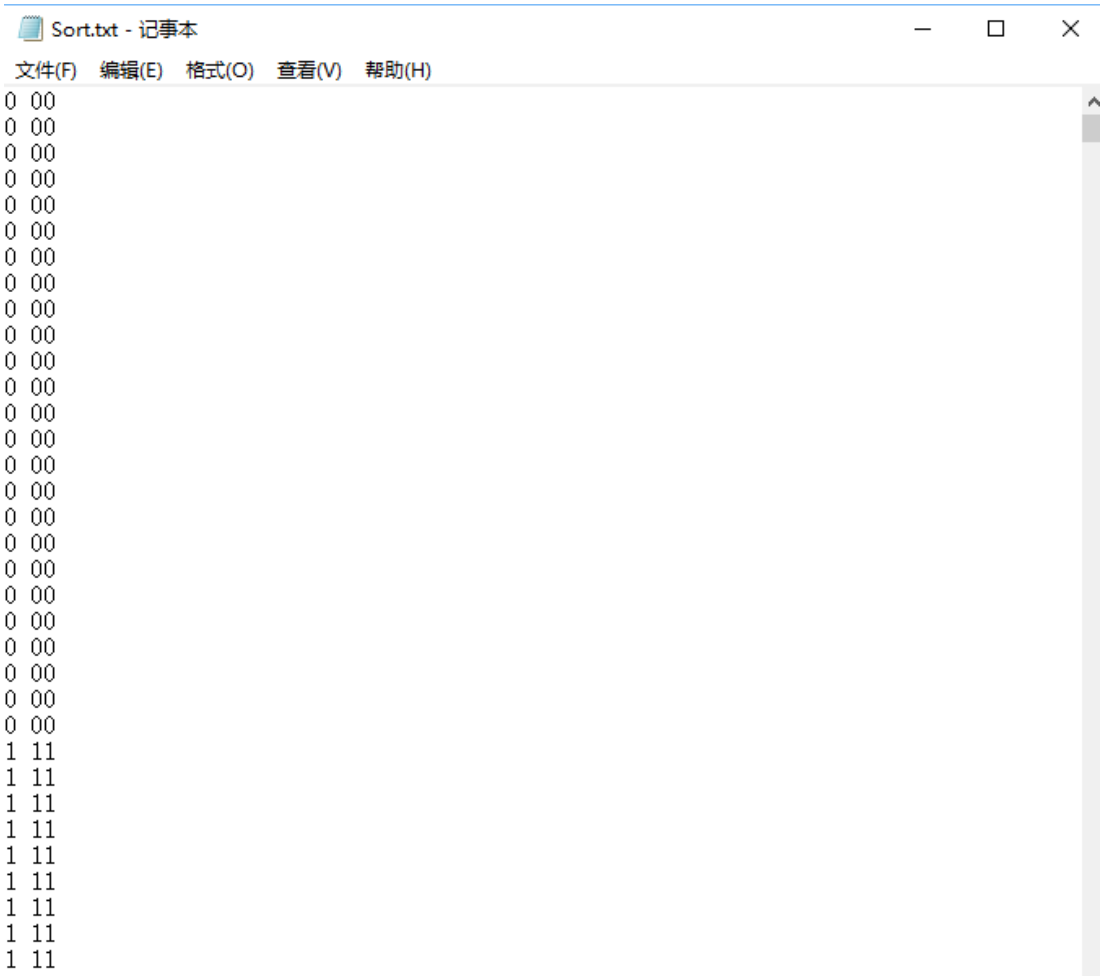


未经过排序的 NotSort.txt 如下图



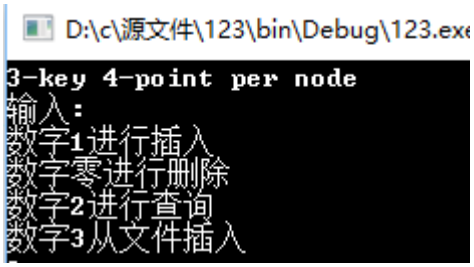
每一路经过排序后得到的有序文件(这里以 forNow0 作为示范，且只显示键 A，等到最后排序完毕后再一同显示键 A 和值 B，其他类似)

排序后得到的 Sort.txt 如下（我的程序中，A 属性和 B 属性的值程序自动生成，A 属性为随机数， B 属性是 2 个 A 属性连接起来，）

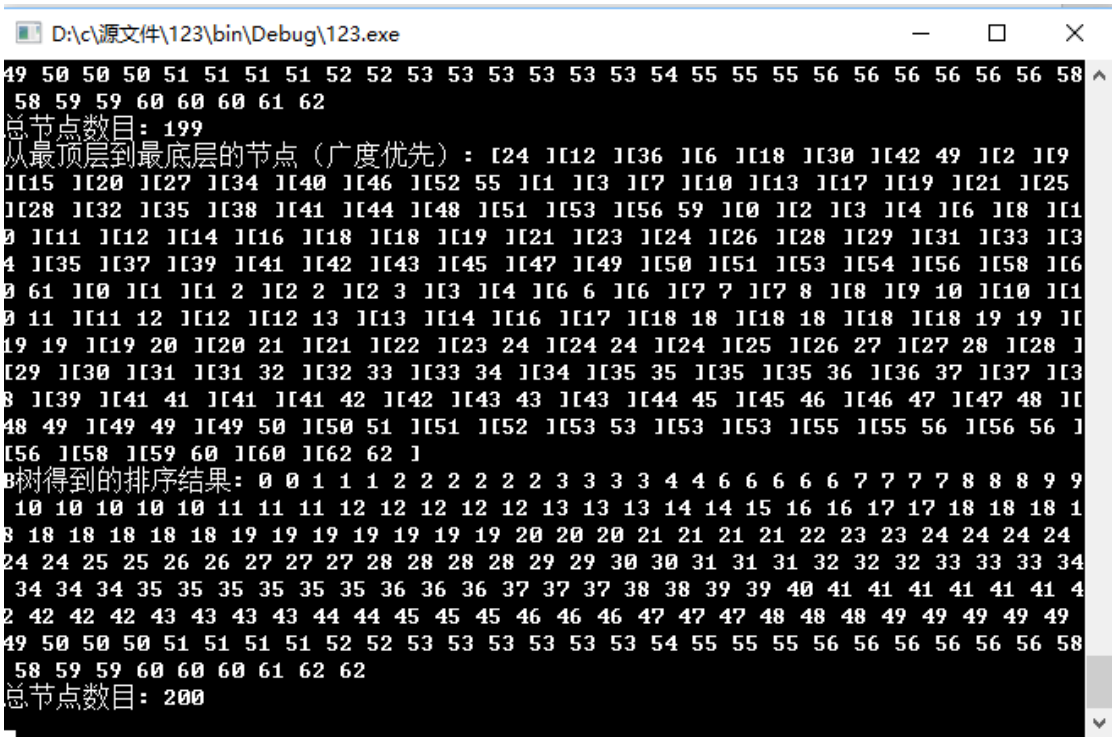


```
17208 1720817208
17208 1720817208
17208 1720817208
17208 1720817208
17208 1720817208
17208 1720817208
17208 1720817208
17208 1720817208
17208 1720817208
17208 1720817208
17208 1720817208
17208 1720817208
17209 1720917209
17209 1720917209
17209 1720917209
17209 1720917209
17209 1720917209
17209 1720917209
17209 1720917209
17209 1720917209
17209 1720917209
17209 1720917209
```

B 树测试：

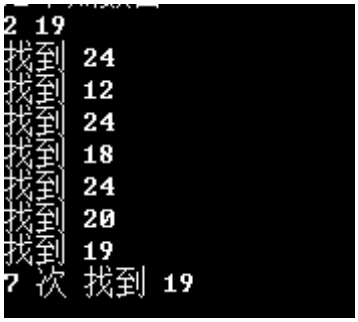


首先从刚才得到的文件 Sort.txt 进行排序（这里为了节约时间，只排序前 200 条）



然后再输入一些简单的命令进行测试

查询路径测试，查询键 19 的位置

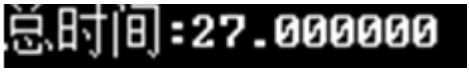


删除测试，删除键 7，可以发现删除后，总结点数由 200 变成了 199，说明成功删除

插入测试 插入键值 23

- **给出对程序效率的分析**

多路归并：

用时为 

时间复杂度为 O(nlogn) 这是该算法中最好、最坏和平均的时间性能。

空间复杂度为 O(n)

比较操作的次数介于(nlogn) / 2 和 nlogn - n + 1。

赋值操作的次数是(2nlogn)。归并算法的空间复杂度为：0 (n)

归并排序比较占用内存，但却是一种效率高且稳定的算法。

B 树：

B- 树查找算法分析

从查找算法中可以看出， 在 B- 树中进行查找包含两种基本操作:

(1) 在 B- 树中查找结点；

(2) 在结点中查找关键字。

由于 B- 树通常存储在磁盘上， 则前一查找操作是在磁盘上进行的， 而后一查找操作是在内存中进行的，即在磁盘上找到指针 p 所指结点后，先将结点中的信息读入内存，然后再利用顺序查找或折半查找查询等于 K 的关键字。显然， 在磁盘上进行一次查找比在内存中进行一次查找的时间消耗多得多.

因此， 在磁盘上进行查找的次数、即待查找关键字所在结点在 B- 树上的层次树， 是决定 B 树查找效率的首要因素

那么，对含有 n 个关键码的 m 阶 B-树，最坏情况下达到多深呢？可按二叉平衡树进行类似分析。首先，讨论 m 阶 B-数各层上的最少结点数。

由 B 树定义：B 树包含 n 个关键字。因此有 n+1 个树叶都在第 J+1 层。

1）第一层为根，至少一个结点，根至少有两个孩子，因此在第二层至少有两个结点。

2）除根和树叶外，其它结点至少有[m/2]个孩子,因此第三层至少有 2*[m/2]个结点,在第四层至少有 2*[m/2]2 个结点…

3）那么在第 J+1 层至少有 2*[m/2]J-1 个结点,而 J+1 层的结点为叶子结点,于是叶子结点的个数 n+1。有：

$$n+1 \geqslant 2*(\lceil m/2 \rceil)^{k-1}$$

即

$$k \leq \log_{\lceil m/2 \rceil}(\frac{n+1}{2})+1$$

也就是说在 n 个关键字的 B 树查找，从根节点到关键字所在的节点所涉及的节点数不超过：

$$\log_{\lceil m/2 \rceil}(\frac{n+1}{2})+1$$

B 树的插入和删除都是都是先查找，再插入或删除，再对树进行合理性调整，其时间复杂度主要为 B 树查找的时间复杂度，为 O(log n)

四、程序代码

多路归并

```
#include <iostream>
#include <ctime>
#include <fstream>
#include <cassert>
#include <string>
//#include "ExternSort.h"
using namespace std;

//使用多路归并进行外排序的类
//ExternSort.h

/*
* 大数据量的排序
* 多路归并排序
* 以千万级整数从小到大排序为例
* 一个比较简单的例子，没有建立内存缓冲区
*/

#ifndef EXTERN_SORT_H
#define EXTERN_SORT_H



#define MIN -1//这里开始的时候出现了一个 BUG，如果定义的 MIN 大于等于待排序的数，则会是算法出现错误
#define MAX 10000000//最大值，附加在归并文件结尾
typedef int* LoserTree;
typedef int* External;

class ExternSort
{
public:
    void sort()
    {
        time_t start = time(NULL);

        //将文件内容分块在内存中排序，并分别写入临时文件
        k = memory_sort();  //

        //归并临时文件内容到输出文件
        //merge_sort(file_count);
        ls = new int[k];
        b = new int[k + 1];
        K_Merge();
        delete[]ls;
```

```
        delete[]b;

        time_t end = time(NULL);
        printf("总时间:%f\n", (end - start) * 1000.0 / CLOCKS_PER_SEC);
    }

    //input_file:输入文件名
    //out_file:输出文件名
    //count: 每次在内存中排序的整数个数
    ExternSort(const char *in_file, const char * out_file, int count)
    {
        m_in_file = new char[strlen(in_file) + 1];
        m_out_file = new char[strlen(out_file) + 1];
        m_count = count;
        strcpy_s(m_in_file, strlen(in_file) + 1, in_file);
        strcpy_s(m_out_file, strlen(out_file) + 1, out_file);
    }
    virtual ~ExternSort()
    {
        delete[] m_in_file;//清理空间
        delete[] m_out_file;
    }

private:
    int m_count; //数组长度
    char *m_in_file;   //输入文件的路径
    char *m_out_file; //输出文件的路径
    int k;//归并数，此数必须要内排序之后才能得到，所以下面的 ls 和 b 都只能定义为
指针
    LoserTree ls;//定义成为指针，之后动态生成数组
    External b;//定义成为指针，在成员函数中可以把它当成数组使用
    //int External[k];
protected:
    int read_data(FILE* f, int a[], int n)
    {
        int i = 0;
        long long sran;//B 区
        while (i < n && (fscanf_s(f, "%d %d\n", &a[i], &sran)) != EOF) { i++; }
        printf_s("读入:%d 整数\n", i);//读入整数个数
        return i;
    }
    void write_data(FILE* f, int a[], int n)
    {
        for (int i = 0; i < n; ++i)
            fprintf(f, "%d\n", a[i]);
        fprintf(f, "%d", MAX);//在最后写上一个最大值
    }
    char* temp_filename(int index)
    {
        char *tempfile = new char[100];//10^100
        sprintf_s(tempfile, 100, "forNow%d.txt", index);//临时文件起名
```

```
    return tempfile;
}
static int cmp_int(const void *a, const void *b)
{
    return *(int*)a - *(int*)b;
}

int memory_sort()
{
    FILE*fin;
    fopen_s(&fin, m_in_file, "rt");
    int n = 0, file_count = 0;
    int *array = new int[m_count];

    //每读入 m_count 个整数就在内存中做一次排序，并写入临时文件
    while ((n = read_data(fin, array, m_count)) > 0)
    {
        qsort(array, n, sizeof(int), cmp_int);//对结构体排序
        char *fileName = temp_filename(file_count++);
        FILE*tempFile;
        fopen_s(&tempFile, fileName, "w");
        free(fileName);
        write_data(tempFile, array, n);//MAX 结尾
        fclose(tempFile);
    }

    delete[] array;
    fclose(fin);

    return file_count;
}

void Adjust(int s)
{//沿从叶子节点 b[s]到根节点 ls[0]的路径调整败者树
    int t = (s + k) / 2;//ls[t]是 b[s]的双亲节点
    while (t>0)
    {
        if (b[s]>b[ls[t]])//如果失败，则失败者位置 s 留下，s 指向新的胜利者
        {
            int tmp = s;
            s = ls[t];
            ls[t] = tmp;
        }
        t = t / 2;
    }
    ls[0] = s;//ls[0]存放调整后的最小值的位置
}

void CreateLoserTree()
{
    b[k] = MIN;//额外的存储一个最小值
```

```
        for (int i = 0; i<k; i++)ls[i] = k;//先初始化为指向最小值，这样后面的调整才是正确的
        //这样能保证非叶子节点都是子树中的“二把手”
        for (int i = k - 1; i >= 0; i--)
            Adjust(i);//依次从b[k-1],b[k-2]...b[0]出发调整败者树
    }

    void K_Merge()
    {//利用败者数把k个输入归并段归并到输出段中
        //b中前k个变量存放k个输入段中当前记录的元素
        //归并临时文件
        FILE*fout;
        fopen_s(&fout, m_out_file, "wt");
        FILE**farray = new FILE*[k];
        int i;
        char *cran = new char[12];
        for (i = 0; i < k; ++i)  //打开所有k路输入文件
        {
            char* fileName = temp_filename(i);
            fopen_s(&farray[i], fileName, "rt");
            free(fileName);
        }

        for (i = 0; i < k; ++i)  //初始读取
        {
            if (fscanf_s(farray[i], "%d", &b[i]) == EOF)//读每个文件的第一个数到data数组
            {
                cout << "there is no " << k << " file to merge!" << endl;
                return;
            }
        }
        //    for(int i=0;i<k;i++)input(b[i]);

        CreateLoserTree();
        int q;
        while (b[ls[0]] != MAX)//每个temp文件末尾皆为MAX
        {
            q = ls[0];//q用来存储b中最小值的位置，同时也对应一路文件
            //output(q);
            strcpy_s(cran, 12, (to_string(b[q]) + to_string(b[q])).c_str());
            fprintf(fout, "%d %s\n", b[q], cran);
            //fprintf(fout, "%d ", b[q]);
            //input(b[q],q);
            fscanf_s(farray[q], "%d", &b[q]);
            Adjust(q);
        }
        //output(ls[0]);
        //fprintf(fout, "%d ", b[ls[0]]);//MAX

        for (i = 0; i < k; ++i)  //清理工作
```

```
        {
            fclose(farray[i]);
        }
        delete[] farray;
        fclose(fout);
    }

};

#endif


//测试主函数文件
/*
* 大文件排序(AB 格式)
* 数据不能一次性全部装入内存
* 排序文件里有多个整数，整数之间用空格隔开
*/

//const unsigned int count = 10000000; // 文件里数据的行数
const unsigned int number_to_sort = 100000; //在内存中一次排序的数量
const char *unsort_file = "NotSort.txt"; //原始未排序的文件名
const char *sort_file = "Sort.txt"; //已排序的文件名
void init_data(unsigned int num); //随机生成数据文件
void isSorted(const char* filename);//检测已排序文件
int main(int argc, char* *argv)
{
    unsigned int count = 1000000;
    srand(time(NULL));
    init_data(count);
    ExternSort extSort(unsort_file, sort_file, number_to_sort);
    extSort.sort();
    isSorted(sort_file);
    system("pause");
    return 0;
}

void init_data(unsigned int num)
{
    FILE *f;
    fopen_s(&f, unsort_file, "wt");
    int ran;//A 区
    char *cran = new char[12];//B 区
    for (unsigned int i = 0; i < num; ++i)
    {
        ran = rand();
        strcpy_s(cran, 12, (to_string(ran) + to_string(ran)).c_str());
        fprintf(f, "%d %s\n", ran, cran);
    }
    delete[] cran;
    fclose(f);
```

```
}

void isSorted(const char* filename) {
    ifstream CheckFile(filename, ios::binary);
    if (CheckFile.is_open() == false) {
        cout << "Fail to open " << filename << "\n";
        return;
    }
    int a, b;
    long long sran;//B区
    long i = 1;
    CheckFile >> a;
    CheckFile >> sran;
    while (CheckFile >> b) {
        if (a > b) {
            cout << i << "th number is wrong\n";
            return;
        }
        else {
            a = b;
            ++i;
        }
        CheckFile >> sran;
    }
    CheckFile.close();
    cout << "Check Finished\n";
}
```

B树 BTree.cpp

```
#include "BTree.h"
#include "struct.h"
#include <stdio.h>
#include <stdlib.h>
#include <io.h>



btree_node* BTree::btree_node_new()
{
    btree_node* node = (btree_node *)malloc(sizeof(btree_node));
    if(NULL == node) {
        return NULL;
    }

    for(int i = 0; i < 2 * M -1; i++) {
        node->k[i] = 0;
    }
```

```
    for(int i = 0; i < 2 * M; i++) {
        node->p[i] = NULL;
    }

    node->num = 0;
    node->is_leaf = true;      //默认为叶子

    return node;
}

btree_node* BTree::btree_create()
{
    btree_node *node = btree_node_new();
    if(NULL == node) {
        return NULL;
    }
    return node;
}

int BTree::btree_split_child(btree_node *parent, int pos, btree_node *child)
{
    btree_node *new_child = btree_node_new();
    if(NULL == new_child) {
        return -1;
    }
    // 新节点的 is_leaf 与 child 相同，key 的个数为 M-1
    new_child->is_leaf = child->is_leaf;
    new_child->num = M - 1;

    // 将 child 后半部分的 key 拷贝给新节点
    for(int i = 0; i < M - 1; i++) {
        new_child->k[i] = child->k[i+M];
    }

    // 如果 child 不是叶子，还需要把指针拷过去，指针比节点多 1
    if(false == new_child->is_leaf) {
        for(int i = 0; i < M; i++) {
            new_child->p[i] = child->p[i+M];
        }
    }

    child->num = M - 1;

    // child 的中间节点需要插入 parent 的 pos 处，更新 parent 的 key 和 pointer
    for(int i = parent->num; i > pos; i--) {
        parent->p[i+1] = parent->p[i];
    }
    parent->p[pos+1] = new_child;

    for(int i = parent->num - 1; i >= pos; i--) {
        parent->k[i+1] = parent->k[i];
```

```
    }
    parent->k[pos] = child->k[M-1];

    parent->num += 1;
    return 0;
}

// 执行该操作时，node->num < 2M-1
void BTree::btree_insert_nonfull(btree_node *node, int target)
{
    if(1 == node->is_leaf) {
        // 如果在叶子中找到，直接删除
        int pos = node->num;
        while(pos >= 1 && target < node->k[pos-1]) {
            node->k[pos] = node->k[pos-1];
            pos--;
        }

        node->k[pos] = target;
        node->num += 1;
        btree_node_num+=1;

    } else {
        // 沿着查找路径下降
        int pos = node->num;
        while(pos > 0 && target < node->k[pos-1]) {
            pos--;
        }

        if(2 * M -1 == node->p[pos]->num) {
            // 如果路径上有满节点则分裂
            btree_split_child(node, pos, node->p[pos]);
            if(target > node->k[pos]) {
                pos++;
            }
        }

        btree_insert_nonfull(node->p[pos], target);
    }
}

//插入入口
btree_node* BTree::btree_insert(btree_node *root, int target)
{
    if(NULL == root) {
        return NULL;
    }

    // 对根节点的特殊处理，如果根是满的，唯一使得树增高的情形
    // 先申请一个新的
    if(2 * M - 1 == root->num) {
```

```
        btree_node* node = btree_node_new();
        if(NULL == node) {
            return root;
        }

        node->is_leaf = 0;
        node->p[0] = root;
        btree_split_child(node, 0, root);
        btree_insert_nonfull(node, target);
        return node;
    } else {
        btree_insert_nonfull(root, target);
        return root;
    }
}

// 将 y，root->k[pos]，z 合并到 y 节点，并释放 z 节点，y,z 各有 M-1 个节点
void BTree::btree_merge_child(btree_node *root, int pos, btree_node *y, btree_node
*z)
{
    // 将 z 中节点拷贝到 y 的后半部分
    y->num = 2 * M - 1;
    for(int i = M; i < 2 * M - 1; i++) {
        y->k[i] = z->k[i-M];
    }
    y->k[M-1] = root->k[pos];// k[pos]下降为 y 的中间节点

    // 如果 z 非叶子，需要拷贝 pointer
    if(false == z->is_leaf) {
        for(int i = M; i < 2 * M; i++) {
            y->p[i] = z->p[i-M];
        }
    }

    // k[pos]下降到 y 中，更新 key 和 pointer
    for(int j = pos + 1; j < root->num; j++) {
        root->k[j-1] = root->k[j];
        root->p[j] = root->p[j+1];
    }

    root->num -= 1;
    free(z);
}

/************  删除分析    **************
*
在删除 B 树节点时，为了避免回溯，当遇到需要合并的节点时就立即执行合并，B 树的删除
算法如下：从 root 向叶子节点按照 search 规律遍历：
（1）  如果 target 在叶节点 x 中，则直接从 x 中删除 target，情况（2）和（3）会保证
当再叶子节点找到 target 时，肯定能借节点或合并成功而不会引起父节点的关键字个数少
于 t-1。
```

（2） 如果 target 在分支节点 x 中：
（a） 如果 x 的左分支节点 y 至少包含 t 个关键字，则找出 y 的最右的关键字 prev，并替换 target，并在 y 中递归删除 prev。
（b） 如果 x 的右分支节点 z 至少包含 t 个关键字，则找出 z 的最左的关键字 next，并替换 target，并在 z 中递归删除 next。
（c） 否则，如果 y 和 z 都只有 t-1 个关键字，则将 targe 与 z 合并到 y 中，使得 y 有 2t-1 个关键字，再从 y 中递归删除 target。
（3） 如果关键字不在分支节点 x 中，则必然在 x 的某个分支节点 p[i]中，如果 p[i]节点只有 t-1 个关键字。
（a） 如果 p[i-1]拥有至少 t 个关键字，则将 x 的某个关键字降至 p[i]中，将 p[i-1]的最大节点上升至 x 中。
（b） 如果 p[i+1]拥有至少 t 个关键字，则将 x 个某个关键字降至 p[i]中，将 p[i+1]的最小关键字上升至 x 个。
（c） 如果 p[i-1]与 p[i+1]都拥有 t-1 个关键字，则将 p[i]与其中一个兄弟合并，将 x 的一个关键字降至合并的节点中，成为中间关键字。
*
*/

```
// 删除入口
btree_node* BTree::btree_delete(btree_node* root, int target)
{
    // 特殊处理，当根只有两个子女，切两个子女的关键字个数都为 M-1 时，合并根与两个子女
    // 这是唯一能降低树高的情形
    if(1 == root->num) {
        btree_node *y = root->p[0];
        btree_node *z = root->p[1];
        if(NULL != y && NULL != z &&
            M - 1 == y->num && M - 1 == z->num) {
                btree_merge_child(root, 0, y, z);
                free(root);
                btree_delete_nonone(y, target);
                return y;
        } else {
            btree_delete_nonone(root, target);
            return root;
        }
    } else {
        btree_delete_nonone(root, target);
        return root;
    }
}

// root 至少有个 t 个关键字，保证不会回溯
void BTree::btree_delete_nonone(btree_node *root, int target)
{
    if(true == root->is_leaf) {
        // 如果在叶子节点，直接删除
        int i = 0;
        while(i < root->num && target > root->k[i]) i++;
        if(target == root->k[i]) {
```

```
            for(int j = i + 1; j < 2 * M - 1; j++) {
                root->k[j-1] = root->k[j];
            }
            root->num -= 1;

            btree_node_num-=1;

        } else {
            printf("target not found\n");
        }
    } else {
        int i = 0;
        btree_node *y = NULL, *z = NULL;
        while(i < root->num && target > root->k[i]) i++;
        if(i < root->num && target == root->k[i]) {
            // 如果在分支节点找到 target
            y = root->p[i];
            z = root->p[i+1];
            if(y->num > M - 1) {
                // 如果左分支关键字多于 M-1，则找到左分支的最右节点 prev，替换
target
                // 并在左分支中递归删除 prev,情况 2（a)
                int pre = btree_search_predecessor(y);
                root->k[i] = pre;
                btree_delete_nonone(y, pre);
            } else if(z->num > M - 1) {
                // 如果右分支关键字多于 M-1，则找到右分支的最左节点 next，替换
target
                // 并在右分支中递归删除 next,情况 2(b)
                int next = btree_search_successor(z);
                root->k[i] = next;
                btree_delete_nonone(z, next);
            } else {
                // 两个分支节点数都为 M-1，则合并至 y，并在 y 中递归删除 target,情
况 2(c)
                btree_merge_child(root, i, y, z);
                btree_delete(y, target);
            }
        } else {
            // 在分支没有找到，肯定在分支的子节点中
            y = root->p[i];
            if(i < root->num) {
                z = root->p[i+1];
            }
            btree_node *p = NULL;
            if(i > 0) {
                p = root->p[i-1];
            }

            if(y->num == M - 1) {
                if(i > 0 && p->num > M - 1) {
```

```
                    // 左邻接节点关键字个数大于 M-1
                    //情况 3(a)
                    btree_shift_to_right_child(root, i-1, p, y);
                } else if(i < root->num && z->num > M - 1) {
                    // 右邻接节点关键字个数大于 M-1
                    // 情况 3(b)
                    btree_shift_to_left_child(root, i, y, z);
                } else if(i > 0) {
                    // 情况 3（c)
                    btree_merge_child(root, i-1, p, y); // note
                    y = p;
                } else {
                    // 情况 3(c)
                    btree_merge_child(root, i, y, z);
                }
                btree_delete_nonone(y, target);
            } else {
                btree_delete_nonone(y, target);
            }
        }

    }
}

//寻找 rightmost，以 root 为根的最大关键字
int BTree::btree_search_predecessor(btree_node *root)
{
    btree_node *y = root;
    while(false == y->is_leaf) {
        y = y->p[y->num];
    }
    return y->k[y->num-1];
}

// 寻找 leftmost，以 root 为根的最小关键字
int BTree::btree_search_successor(btree_node *root)
{
    btree_node *z = root;
    while(false == z->is_leaf) {
        z = z->p[0];
    }
    return z->k[0];
}

// z 向 y 借节点，将 root->k[pos]下降至 z，将 y 的最大关键字上升至 root 的 pos 处
void BTree::btree_shift_to_right_child(btree_node *root, int pos,
    btree_node *y, btree_node *z)
{
    z->num += 1;
    for(int i = z->num -1; i > 0; i--) {
        z->k[i] = z->k[i-1];
```

```
    }
    z->k[0]= root->k[pos];
    root->k[pos] = y->k[y->num-1];

    if(false == z->is_leaf) {
        for(int i = z->num; i > 0; i--) {
            z->p[i] = z->p[i-1];
        }
        z->p[0] = y->p[y->num];
    }

    y->num -= 1;
}

// y 向借节点，将 root->k[pos]下降至 y，将 z 的最小关键字上升至 root 的 pos 处
void BTree::btree_shift_to_left_child(btree_node *root, int pos,
    btree_node *y, btree_node *z)
{
    y->num += 1;
    y->k[y->num-1] = root->k[pos];
    root->k[pos] = z->k[0];

    for(int j = 1; j < z->num; j++) {
        z->k[j-1] = z->k[j];
    }

    if(false == z->is_leaf) {
        y->p[y->num] = z->p[0];
        for(int j = 1; j <= z->num; j++) {
            z->p[j-1] = z->p[j];
        }
    }

    z->num -= 1;
}

void BTree::btree_inorder_print(btree_node *root)
{
    if(NULL != root) {
        btree_inorder_print(root->p[0]);
        for(int i = 0; i < root->num; i++) {
            printf("%d ", root->k[i]);
            btree_inorder_print(root->p[i+1]);
        }
    }
}

bool BTree::btree_query(btree_node *root, int target, int search_counts)
{
    if(NULL != root){
        for(int i = 0; i < 2 * M -1; i++){
```

```
            printf("找到 %d\n",root->k[i]);
            if(root->k[i]==target){
                search_counts++;
                printf("%d 次 找到 %d\n",search_counts,target);
                return true;
            }
            else if(root->k[i]==0){
                search_counts++;
                return btree_query(root->p[i],target,search_counts);
            }
            else if(root->k[i]>target){
                search_counts++;
                return btree_query(root->p[i],target,search_counts);
            }
            else{
                search_counts++;
                continue;
            }
        }
    }
    else
    {
        printf("Not found\n");
        return false;
    }
}

void BTree::btree_level_display(btree_node *root)
{
    // just for simplicty, can't exceed 200 nodes in the tree
    btree_node *queue[200] = {NULL};
    int front = 0;
    int rear = 0;

    queue[rear++] = root;

    while(front < rear) {
        btree_node *node = queue[front++];

        printf("[");
        for(int i = 0; i < node->num; i++) {
            printf("%d ", node->k[i]);
        }
        printf("]");

        for(int i = 0; i <= node->num; i++) {
            if(NULL != node->p[i]) {
                queue[rear++] = node->p[i];
            }
        }
    }
```

```
    printf("\n");
}

void BTree::Save(btree_node *root)
{
    /*
    storage_struct ss;

    // malloc len space
    ss.len = btree_node_num;
    ss.snode = (storage_node *)malloc(sizeof(storage_node)*ss.len);

    ss.snode[0].bnode = *root;
    for(int i=1;i<ss.len;i++)
    {
        btree_node *node = btree_node_new();
        if(NULL == node) {
            return;
        }
    }

//  fwrite(&ss,sizeof(ss),1,pfile);
*/
}

BTree::BTree(void)
{
    // 先判断文件是否存在
    // windows 下, 是 io.h 文件, linux 下是 unistd.h 文件
    // int access(const char *pathname, int mode);
    if(-1==access("define.Bdb",F_OK))
    {
        // 不存在 ,创建
    //      pfile = fopen("bstree.b","w");
        roots = btree_create();
    }
    else
    {
//      pfile = fopen("bstree.b","r+");
        roots = btree_create();
//      fread(roots,sizeof(roots),1,pfile);
    }

}


BTree::~BTree(void)
{
//  fclose(pfile);
}
```

B 树 main.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
//#include <bits/stdc++.h>
#include "BTree.h"
#include "BPlusTree.h"
#include "Context.h"

using namespace std;
int main()
{

    /*************************************直接调用 B 的调度使用
*****************************************/
    /************************************* Direct call B method
*****************************************/
    BTree bt;
    //BPlusTree bpt;
    int choice, digit;
    FILE *fp;
    fp = fopen("temp3.txt", "r");
    //  if(fp)printf("hello");
    int a[200];
    for (int i = 0;i<200;i++)
    {
        fscanf(fp,"%d",&a[i]);
    }
   // insertsort(a,0,39999);
    fclose(fp);
//    for(int i =1;i<100;i++)
//       printf("%d\n",a[i]);

    cout<<"3-key 4-point per node\n 输入:\n 数字 1 进行插入\n 数字零进行删除\n 数字
2 进行查询\n 数字 3 从文件插入\n";
    while(1)
    {
        cin>>choice>>digit;
        if(choice==1)
        {
            bt.insert(digit);
            cout<<"从最顶层到最底层的节点（广度优先）: ";
            bt.level_display();
            cout<<"B 树得到的排序结果: ";
            bt.inorder_print();
            cout<<endl<<"总节点数目: ";
            bt.NodeNum_print();
```

```
        }
        else if(choice==0)
        {
            bt.del(digit);
            cout<<"从最顶层到最底层的节点（广度优先）: ";
            bt.level_display();
            cout<<"B 树得到的排序结果: ";
            bt.inorder_print();
            cout<<endl<<"总节点数目: ";
            bt.NodeNum_print();
        }
        else if(choice==2)
        {
            int query_times = 0;
            bt.query(digit, query_times);
        }
        else if(choice = 3){
            int j = 0;
            for(;j<200;j++){
            bt.insert(a[j]);
            cout<<"从最顶层到最底层的节点（广度优先）: ";
            bt.level_display();
            cout<<"B 树得到的排序结果: ";
            bt.inorder_print();
            cout<<endl<<"总节点数目: ";
            bt.NodeNum_print();
            }
        }
        else break;
    }

    return 0;
}
```