实验二 数据库索引及查询算法实现

1、　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

2、　实验环境：

操作系统：windows

编程语言：c++

3、实验内容及要求：

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。

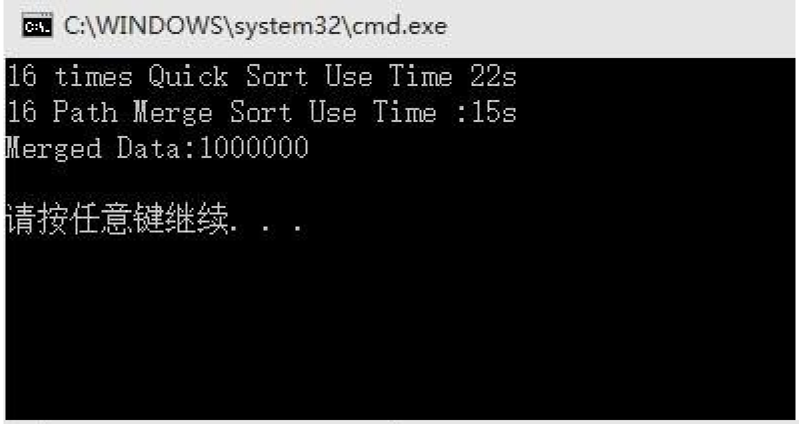
3）以属性A为键值，实现B树索引。完成索引的插入，删除和查找操作。

4）针对属性A，用高级语言实现多路归并排序算法。

5）用于外部归并排序的内存空间不大于1MB。

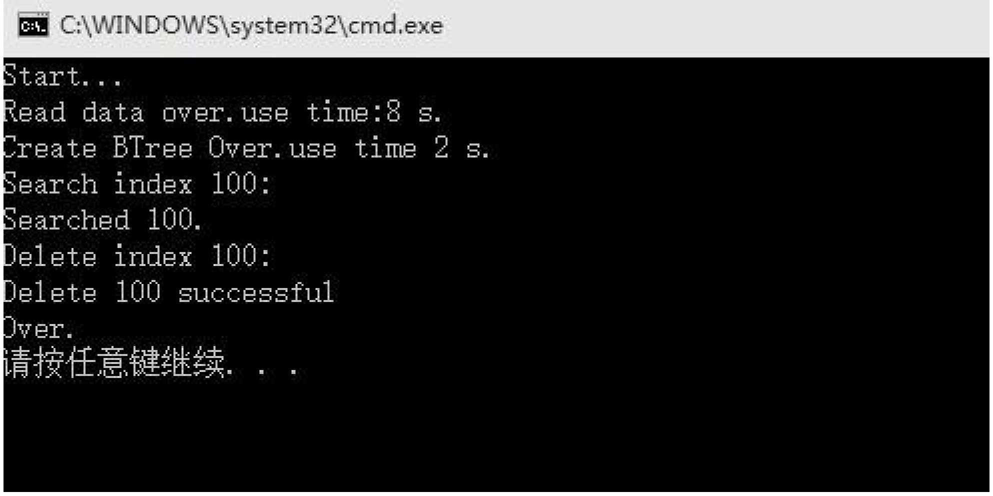
4、　实验内容

截屏给出实验结果



```
16 times Quick Sort Use Time 22s
16 Path Merge Sort Use Time :15s
Merged Data:1000000

请按任意键继续. . .
```



```
Start...
Read data over.use time:8 s.
Create BTree Over.use time 2 s.
Search index 100:
Searched 100.
Delete index 100:
Delete 100 successful
Over.
请按任意键继续. . .
```

算法流程：

由于共有1000000个元组，每个元组128B，因此总大小为128M，又内存限制为8M，则至少要分成128M/8M=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(lgn)的时间复杂度内选出最小的A属性元组，这种方法从划分到归并，读入两次，写出两次，最少的I/O次数是4*1000000.

程序代码：

1、生成1000000个128字节的元组代码：

```
#include <iostream> #include <fstream> #include <ctime> #include <cstdlib> using namespace std;

const int N = 1000000;

string RandomChineseCharacters()
{
//srand( (unsigned)time(NULL));
int high = 0xd7 - 0xc1;//16-55区汉字int low= 0xfe - 0xa1;
int high_zn ; int low_zn; char name[3]; name[2] = '\0';
```

```
string s;
for(int i = 0; i < 60; i++)
{
high_zn = rand()%high + 0xc1; low_zn = rand()%low + 0xa1; name[0]=high_zn; name[1]=low_zn;
s.append(name);
}
return s;
}
int main()
{
ofstream ofp ("example2.txt"); if (!ofp.is_open())
{
cout<<"can't open file!"<<endl; return 0;
}
srand( (unsigned)time(NULL)); int t1 = 0,t2 = 0;//4+
string name;
for(int i = 1;i <= N;i++)
{
t1 = rand(); t2 = i;
name = RandomChineseCharacters();
//cout<<name <<sizeof(t1)<<sizeof(t2)<<name.length()<<endl;
ofp<<t1<<" "<<t2<<" "<<name<<endl;
}

return 0;

}
```

2、16组快速排序和16路归并排序的代码

```
// Extenal_Merge_Sort.cpp : 定义控制台应用程序的入口点。
//

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


typedef struct Data
{
int data; int num;
char name[121];
```

```
} Data;
//利用败者树
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,必须比要排序数字的最小值要小，否则出错
const int MAX = 9999999; //最大值,必须比要排序数字的最大值要大，否则出错

int result = 0;
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;
}

//从unsort_data.txt中读取数据

int read_data(FILE *fp, Data *array, int N)
{
int length = 0;
for (int i = 0; i < MAX_PART && (EOF !=
fscanf_s(fp,    "%d    %d     %s\n", &array[i].data,&array[i].num,
array[i].name,_countof(array[i].name))); i++)
{
length++;
}
return length;
}



//打开data0.txt - data9.txt这10个文件FILE* open_file(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data0.txt - data9.txt这10个文件写入排好序的数据void write_data(Data *array, int N, int
count)
{
FILE *fpwrite = open_file(count, "w"); int i = 0;
for (i = 0; i < N; i++)
{
fprintf(fpwrite,          "%d     %d      %s\n", array[i].data,   array[i].num, array[i].name);
```

```
}
fprintf(fpwrite, "%d %d %s\n", MAX, array[i - 1].num, array[i-1].name);
//在每个文件最后写入一个最大值，表示文件结束fclose(fpwrite);
}

//内部排序，调用16次快速排序，产生data0.txt - data16.txt这10个有序文件
void interior_sort(void)
{
clock_t begin = clock(); FILE *fpread = NULL;
fopen_s(&fpread,"unsort_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 = read_data(fpread, array, MAX_PART); if (length == 0)
{
break;
}
qsort(array, length, sizeof(Data), cmp); write_data(array, length, count); count++;
```

```
}
delete[] array; fclose(fpread); clock_t end = clock();
cout << "16 times Quick Sort Use Time " << (end - begin) / CLK_TCK
<< "s" << endl;
}

//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;
}

void create_loser_tree(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 merge_sort_by_losertree()
{
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,"sort_data_by_losertree.txt", "w"); assert(fpwrite != NULL);

for (int i = 0; i < FILE_NUM; i++)
{
fpreads[i] = open_file(i, "r");
}
for (int i = 0; i < FILE_NUM; i++)
{
fscanf_s(fpreads[i],    "%d    %d     %s\n", &data[i].data,  &data[i].num, data[i].name,_countof(data[i].name));
}
```

```
create_loser_tree(ls, data); //创建败者树while (data[ls[0]].data != MAX)
{
index = ls[0];
fprintf(fpwrite, "%d %d %s\n", data[index].data, data[index].num,
data[index].name);
result++;//测试数据是否全部读完。
fscanf_s(fpreads[index], "%d %d %s\n", &data[index].data, &data[index].num,
data[index].name, _countof(data[index].name));
adjust(ls, data, index);
}
for (int i = 0; i < FILE_NUM; i++)
{
fclose(fpreads[i]);
}
fclose(fpwrite); clock_t end = clock();
cout << "16 Path Merge Sort Use Time :" << (end - begin) / CLK_TCK
<< "s" << endl;
}


int _tmain(int argc, _TCHAR* argv[])
{
interior_sort(); merge_sort_by_losertree();
cout << "Merged Data:" << result << endl; getchar();
```

```
return 0;
}
```

3、B树索引的建立、查找、删除的代码

```
#include "stdafx.h" #include <stdio.h> #include <stdlib.h> #include <assert.h> #include <ctime>
/**
@brief the degree of btree
key per node: [M-1, 2M-1]
child per node: [M, 2M]
*/
#define M 2
#define MaxSize 1000001 typedef struct btree_node {
int k[2 * M - 1];
struct btree_node *p[2 * M]; int num;
bool is_leaf;
} btree_node;

/**
@brief allocate a new btree node
default: is_leaf == true
*
@return pointer of new node
```

```
*/
btree_node *btree_node_new();


/**
@brief create a btree root
*
@return pointer of btree root
*/
btree_node *btree_create();


/**
@brief split child if num of key in child exceed 2M-1
*
@param parent: parent of child
@param pos: p[pos] points to child
@param child: the node to be splited
*
@return
*/
int btree_split_child(btree_node *parent, int pos, btree_node *child);


/**
@brief insert a value into btree
the num of key in node less than 2M-1
*
@param node: tree root
@param target: target to insert
```

```
*/
void btree_insert_nonfull(btree_node *node, int target);


/**
@brief insert a value into btree

*
@param root: tree root
@param target: target to insert
*
@return: new root of tree
*/
btree_node* btree_insert(btree_node *root, int target);


/**
@brief merge y, z and root->k[pos] to left
this appens while y and z both have M-1 keys
*
@param root: parent node
@param pos: postion of y
@param y: left node to merge
@param z: right node to merge
*/
void btree_merge_child(btree_node *root, int pos, btree_node *y, btree_node
*z);

/**
@brief delete a vlue from btree
```

```
*
@param root: btree root
@param target: target to delete
*
@return: new root of tree
*/
btree_node *btree_delete(btree_node *root, int target);

/**
@brief delete a vlue from btree
root has at least M keys
*
@param root: btree root
@param target: target to delete
*
@return
*/
void btree_delete_nonone(btree_node *root, int target);


/**
@brief find the rightmost value
*
@param root: root of tree
*
@return: the rightmost value
*/
int btree_search_predecessor(btree_node *root);
```

```
/**
@brief find the leftmost value
*
@param root: root of tree
*
@return: the leftmost value
*/
int btree_search_successor(btree_node *root);


/**
@brief shift a value from z to y
*
@param root: btree root
@param pos: position of y
@param y: left node
@param z: right node
*/
void btree_shift_to_left_child(btree_node *root, int pos, btree_node *y, btree_node *z);

/**
@brief shift a value from z to y
*
@param root: btree root
@param pos: position of y
@param y: left node
@param z: right node
*/
```

```
void btree_shift_to_right_child(btree_node *root, int pos, btree_node *y, btree_node *z);


/**
@brief inorder traverse the btree
*
@param root: root of treee
*/
void btree_inorder_print(btree_node *root);


/**
@brief level print the btree
*
@param root: root of tree
*/
void btree_level_display(btree_node *root);

btree_node *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;
}

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

return node;
}

int btree_split_child(btree_node *parent, int pos, btree_node *child)
{
btree_node *new_child = btree_node_new(); if (NULL == new_child) {
return -1;
}

new_child->is_leaf = child->is_leaf; new_child->num = M - 1;

for (int i = 0; i < M - 1; i++) { new_child->k[i] = child->k[i + M];
```

```
}

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;

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;
}

void btree_insert_nonfull(btree_node *node, int target)
{
if (1 == node->is_leaf) { 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;

}
else {
int pos = node->num;
while (pos > 0 && target < node->k[pos - 1]) { pos--;
}

if (2 * M - 1 == node->p[pos]->num) { btree_split_child(node, pos, node->p[pos]); if (target >
node->k[pos]) {
pos++;
}
}

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

btree_node* btree_insert(btree_node *root, int target)
{
if (NULL == root) { return NULL;
}

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;
}
}

void btree_merge_child(btree_node *root, int pos, btree_node *y, btree_node
*z)
{
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];

if (false == z->is_leaf) {
for (int i = M; i < 2 * M; i++) { y->p[i] = z->p[i - M];
}
}

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);
}

btree_node *btree_delete(btree_node *root, int target)
{
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;
}
}

void btree_delete_nonone(btree_node *root, int target)
{
if (true == root->is_leaf) { 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;
}
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]) {
y = root->p[i];
z = root->p[i + 1];
if (y->num > M - 1) {
```

```
int pre = btree_search_predecessor(y); root->k[i] = pre; btree_delete_nonone(y, pre);
}
else if (z->num > M - 1) {
int next = btree_search_successor(z); root->k[i] = next; btree_delete_nonone(z, next);
}
else {
btree_merge_child(root, i, y, z); btree_delete(y, target);
}
}
else {
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) { btree_shift_to_right_child(root, i - 1, p, y);
}
else if (i < root->num && z->num > M - 1) {
```

```
btree_shift_to_left_child(root, i, y, z);
}
else if (i > 0) {
btree_merge_child(root, i - 1, p, y); // note y = p;
}
else {
btree_merge_child(root, i, y, z);
}
btree_delete_nonone(y, target);
}
else {
btree_delete_nonone(y, target);
}
}

}
}

int 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];
}

int btree_search_successor(btree_node *root)
```

```
{
btree_node *z = root;
while (false == z->is_leaf) { z = z->p[0];
}
return z->k[0];
}



void 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;
}

void btree_shift_to_left_child(btree_node *root, int pos,
btree_node *y, btree_node *z)
{
y->num += 1;
>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];
}
}

>num -= 1;
}

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

```
}

void btree_level_display(btree_node *root)
{
// just for simplicty, can't exceed 200 nodes in the tree btree_node *queue[MaxSize*2] = { 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");
}

int ReadData(int * arr)//return length
{
```

```
FILE* fp = NULL;
fopen_s(&fp,"unsort_data.txt", "r"); assert(fp != NULL);
int a = 0, length = 0; char temp[121];
for (int i = 0; (EOF != fscanf_s(fp, "%d %d %s\n", &arr[i], &a, temp,_countof(temp))); i++)
{
length++;
}
fclose(fp); return length;
}

int main()
{
printf("Start...\n");
int *arr = new int[MaxSize]; clock_t t = clock();
int length = ReadData(arr);
printf("Read data over.use time:%d s.\n", (clock() - t) / CLK_TCK); t = clock();
FILE * fp = NULL;
fopen_s(&fp, "BTree.txt", "w"); assert(fp != NULL);
btree_node *root = btree_create(); for (int i = 0; i < length; i++) {
root = btree_insert(root, arr[i]);
}
```

```
printf("Create BTree Over.use time %d s.\n", (clock() - t) / CLK_TCK);
btree_inorder_print(fp,root);
return 0;
}
```

对实验结果的分析：

实验结果表明，分成16个组后，快速排序用时22秒，而对于归并排序， 每个文件用时15秒，在128M的文件大小下表现优异。且未见遗漏或异常。