目录

[1运行环境需求和开发语言 1](#_Toc2775559)

[2配置文件内容 2](#_Toc2775560)

[2.1默认配置文件 2](#_Toc2775561)

[2.2自定义配置文件 2](#_Toc2775562)

[3读取config文件 3](#_Toc2775563)

[3.1 用户选择默认配置文件 3](#_Toc2775564)

[3.2 用户选择自定义配置文件 3](#_Toc2775565)

[4 block的位置映射 4](#_Toc2775566)

[4.1 block的个数 4](#_Toc2775567)

[4.2 映射block的起始位置 4](#_Toc2775568)

[4.3 映射block的block\_num 4](#_Toc2775569)

[4.4 映射block的fat表项 4](#_Toc2775570)

[5 block结构设计 5](#_Toc2775571)

[5.1 文件分配表 5](#_Toc2775572)

[5.2 存储管理型block 6](#_Toc2775573)

[5.3 数据索引型block 6](#_Toc2775574)

[5.4 数据存储型block 7](#_Toc2775575)

[6 空闲block管理 8](#_Toc2775576)

[6.1 fat\_initialize()函数 8](#_Toc2775577)

[6.2 fat\_block\_apply()函数 8](#_Toc2775578)

[6.3 btree\_block\_apply()函数 8](#_Toc2775579)

[7 程序初始化 9](#_Toc2775580)

[7.1 读取配置文件config.ini 9](#_Toc2775581)

[7.2 创建新配置文件define\_config.ini 10](#_Toc2775582)

[8 数据操作 12](#_Toc2775583)

[8.1 建立数据表 13](#_Toc2775584)

[8.2 插入一条数据 14](#_Toc2775585)

[8.3 删除数据 16](#_Toc2775586)

[8.4 修改数据 18](#_Toc2775587)

[8.5 查询数据 20](#_Toc2775588)

[8.6 查阅数据表 21](#_Toc2775589)

[8.7 退出程序 22](#_Toc2775590)

[总结 24](#_Toc2775591)

[附录 25](#_Toc2775592)

# 

# 1运行环境需求和开发语言

本系统在任何地点都可以使用，只需要有一台安装Visual Studio计算机即可，计算机配置如下所示：

处理器：酷睿i5以上

内存：4G以上

操作系统：windows 7/8/10

运行平台：Visual Studio 2017

开发语言：C语言

# 2配置文件内容

## 2.1默认配置文件

程序读取congfig.txt文件，默认创建数据文件及相关设置。

第1行：flag=0(表示config.txt文件是否是已被读取过的标志变量，0为否，1为是)

第2行：block\_size=2KB（一个block数据块的大小）

采用循环读取每行的内容，一次读取两行，即一个文件的创建。

第3行：file\_name1=File1.txt（第一个数据文件的名字）

第4行：file\_size1=2MB（第一个数据文件的大小）

第5行：file\_name2=File2.txt（第二个数据文件的名字）

第6行：file\_size2=2MB（第二个数据文件的大小）

……

## 2.2自定义配置文件

程序调用congfig.c文件，创建自定义配置文件define\_config.txt完成与用户进行交互式创建数据文件及相关设置。

（1）循环调用File ()函数完成每个file的创建。

file\_name=用户输入.txt（数据文件的名字，判断文件名是否已经存在，若存在，则更换名字或进行覆盖创建，若不存在，则创建新文件）

file\_size=用户输入MB（数据文件的大小）

（3）block\_size（）函数，完成blocksize的定义

block\_size=用户输入KB（一个block数据块的大小，鉴于block结构的设计，不允许用户设定block大小）

# 3读取config文件

系统启动，运行包含主函数的fat\_block.c文件，在该文件中用户进行默认和自定义配置文件的选择，若选择默认配置文件，则程序调用read\_config.c文件读取congfig.ini文件中的相关参数完成配置工作。若用户选择自定义配置文件，则程序调用write\_config.c文件与用户交互完成配置工作并写入define\_config.ini文件。

## 3.1 用户选择默认配置文件

当用户选择默认配置文件时，程序读取config.txt文件，若不能打开则返回错误信息，并选择是否进行自定义创建。打开之后，程序读取flag标志变量，判断是否为首次读取，若是首次创建，则将flag标志变量由0置为1，按行读取config.ini内容并调用get\_value()函数获取数据，进行文件创建和block的大小设定。若不是首次读取，即flag已被置1，向用户反馈当前信息，判断是否继续创建文件。

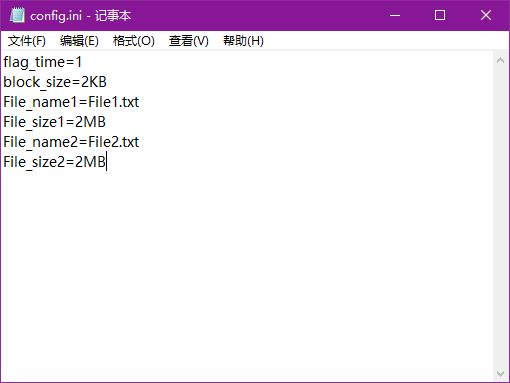


图3.1 config.ini文件

## 3.2 用户选择自定义配置文件

当用户选择自定义配置文件时，系统调用creat\_define\_config()函数，该函数调用其中的write\_file（）函数完成磁盘文件名与文件大小的设定与创建，调用write\_block\_size（）函数完成block\_size的设定，从而完成配置文件的设定。

# 4 block的位置映射

假设一个block的大小为block\_size字节，共m个文件，每一个文件的大小为File\_size[i]字节，block号为block\_num(由0开始编号，即第一块block为0号block)，则：

## 4.1 block的个数

文件系统中block的个数n：

## 4.2 映射block的起始位置

一个block的起始位置s、终止位置e：

block由0开始编号

s = block\_num \* block\_size

e =（block\_num + 1） \* block\_size或s+ block\_size-1

起始位置s表示从文件第s个字节开始，终止位置e表示到文件第e个字节结束。

## 4.3 映射block的block\_num

第j个文件的第k个字节所在的block\_num：

## 4.4 映射block的fat表项

fat\_value\_amount表示一个block中存放fat表项值的总数。

由block\_num计算其在fat中的位置：

# 5 block结构设计

根据block的功能，将其分为文件分配表、数据存储两种类型。由于用户自定义配置文件的文件大小和块大小不确定，所以在此以默认配置文件为例设计block的结构。根据默认配置文件block的设计，可以按比例设定用户自定义配置文件中的文件、block大小和block的结构。

## 5.1 文件分配表

文件分配表型block。用于指示块的存储状态，即记录数据存储区每一个块的使用情况。也就是记录某个数据表存储在数据区的哪个位置。

特性：

* FAT中每个块的状态，使用32bit(4字节)记录在FAT表中。FAT表中的所有字节位置以4个字节为单位进行划分；
* 并以所有划分后的位置由0进行地址编号，“0号和1号”地址被系统保留并存储特殊标识内容，从fat\_block\_amount号地址开始，每个地址对应于数据存储区的块号，FAT表中的地址编号与数据存储区中的块号相同；
* FAT表由一些列大小相等的表项组成。称FAT中的这些单字节划分的项的地址为FAT表项，FAT表项中记录的值为FAT表项值。每个表项值记录下面五种信息中的一种。
  + 链中下一个块的地址
  + 一个特殊的文件结束值（1）指示链的结束
  + 数值（0）来表示空闲块，非（0）表示块被占用

设m为文件分配表占用的block个数，n为文件系统中block的个数，则m上取整：

根据公式知，与文件系统全部block作对应需要m个block作为文件分配表,即0～m-1号block存储文件分配表。从m号block开始存储根目录。

表项结构如下：

（1）Block\_num：当前block的块号。定义为int类型，占用4个字节，存储位置为当前block的0字节～3字节。

（2）FAT表项：存储与数据存储区block\_num相对应的地址的表项值。每个表项值定义为int类型，占用4个字节。共占用block\_size-4个字节，可记录个表项值。第k(1≤k≤)个表项值的起始位置为当前block的4+（k-1）\*4字节～7+（k-1）\*4字节。

表5.1 i(0≤i≤m-1)号block的存储结构（文件分配表）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Block\_num | FAT表项 | | | |
|  |  | … |  |
| 表项值 | 表项值 | … | 表项值 |

## 5.2 存储管理型block

m号block属于存储管理型block，用于记录存储文件信息、空闲块信息。

（1）Block\_num：当前block的块号。定义为int类型，占用4个字节，存储位置为当前block的0字节～3字节。

（2）Contents:文件存储目录区，共占用block\_size-4个字节,每个文件的存储信息占用8个字节，可记录(下取整)个文件，第i（1≤i≤）个文件目录的起始位置为4+（i-1）\*8字节～11+（i-1）\*8字节。

表5.2 存储管理型block的存储结构

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Block\_num | Contents | | | | | |
| 0th\_file | | 1st\_file | | … | 剩余 |
| index\_block | data\_block | index\_block | data\_block | … |

## 5.3 数据索引型block

索引型block用于存储文件原始数据的key,以便用于Btree查询方法

（1）Block\_num：当前block的块号。定义为int类型，占用4个字节，存储位置为当前block的0字节～3字节。

（2）Index：索引记录存储区。共占用block\_size-4个字节，每条索引占用12+12a（a为树的阶数）字节，可存储（下取整）条索引。第i（1≤i≤）条记录的起始位置为当前block的4+(i-1)\*(12+12a)字节～15+12a+(i-1)\*(12+12a)字节。

1）key\_num：记录数据的关键字个数，定义为int类型，占用4个字节，存储位置为当前block的4+(i-1)\*(12+12a)字节～7+12a+(i-1)\*(12+12a)字节。

2）key：记录数据的关键字，定义为int类型，占用4a个字节，存储位置为当前block的8+(i-1)\*(12+12a)字节～7+4a+(i-1)\*(12+12a)字节。

3）key\_block：记录关键字对应的数据所在的block的block号，定义为int类型，占用4a个字节，存储位置为当前block的8+4a+(i-1)\*(12+12a)字节～7+8a+(i-1)\*(12+12a)字节。

4）child\_block：记录孩子节点指针所在的block的block号，定义为int类型，占用5a个字节，存储位置为当前block的7+8a+(i-1)\*(12+12a)字节～6+13a+(i-1)\*(12+12a)字节。

5）pareat\_block：记录孩子节点指针所在的block的block号，定义为int类型，占用4个字节，存储位置为当前block的6+13a+(i-1)\*(12+12a)字节～9+13a+(i-1)\*(12+12a)字节。

表5.3 索引型block的存储结构

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Block\_num | Index | | | | | | |
| Index 0 | | | | | … | 剩余 |
| key\_num | key | key\_block | child\_block | parent\_block | … |

## 5.4 数据存储型block

数据存储型block用于存储文件的内容信息。

（1）Block\_num：当前block的块号。定义为int类型，占用4个字节，存储位置为当前block的0字节～3字节。

（2）Record：数据存储区。共占用block\_size-4个字节，每条记录占用20字节，可存储（下取整）条记录。第i（1≤i≤）条记录的起始位置为当前block的4+（i-1）\*20字节～23+（i-1）\*20字节。

表5.4 数据型block的存储结构

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Block\_num | Record | | | | | |
| Record0 | Record1 | Record2 | … | Record | 剩余 |

# 6 空闲block管理

根据FAT表项的值是否为0可以判断该地址对应的block是否为空。统计0～m-1号block中的表项值为0的个数即可得知空闲block的个数。

## 6.1 fat\_initialize()函数

程序调用fat\_initialize()函数，完成文件分配表的初始化，写入块号和空闲值“0”。

## 6.2 fat\_block\_apply()函数

当程序存储数据时，调用fat\_block\_apply()函数申请空闲block，调用时传递三个参数apply\_flag(结束申请)、first\_apply(首次申请)、prior\_num(当前block\_num)，fat\_block\_apply()函数遍历fat表项寻找空闲块。

若无空闲，则反馈“空间不足”。

若充足，则返回空闲块的block\_num。若结束申请，则在新表项中写入“1”，否则不做处理；若是首次申请，则prior\_num传递“0”，否则prior\_num传递当前块号，在当前块号的表项中写入申请的新空闲block\_num。

## 6.3 btree\_block\_apply()函数

当程序存储索引时，调用btree\_block\_apply()函数申请空闲block，调用时无需传递参数，btree\_block\_apply()函数遍历fat表项寻找空闲块。

若无空闲，则反馈“空间不足”。

若充足，则返回空闲块的block\_num，并在新表项中写入“1”，

# 7 程序初始化

系统初始化分为两种方式：一是读取配置文件，以配置文件中的内容对系统进行初始化；二是自定义配置文件内容，通过交互方式获取相关配置信息。

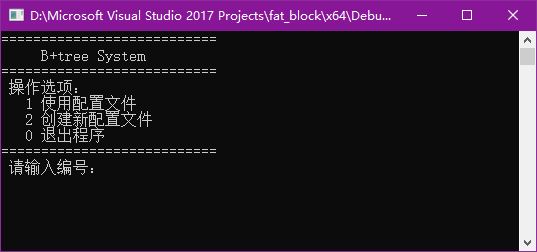


图7.1 初始化选项

## 7.1 读取配置文件config.ini

选择“1 使用配置文件”选项，读取配置文件进行初始化操作。

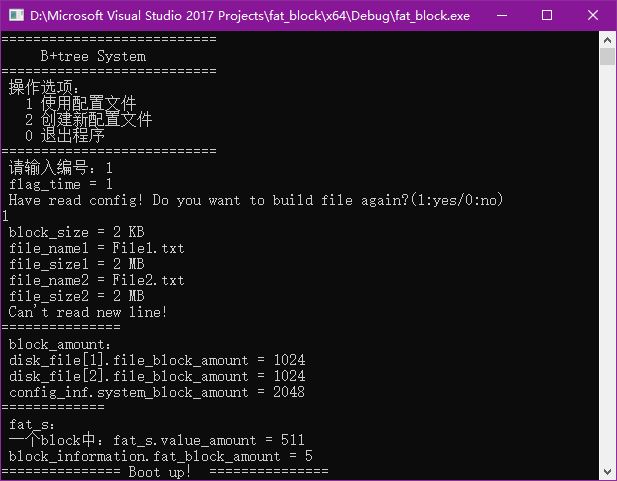
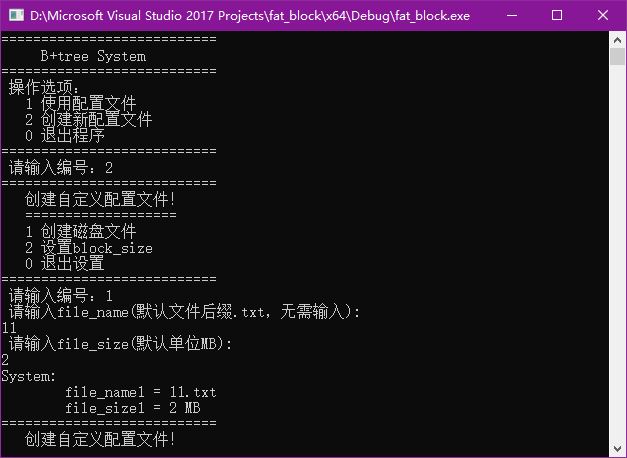


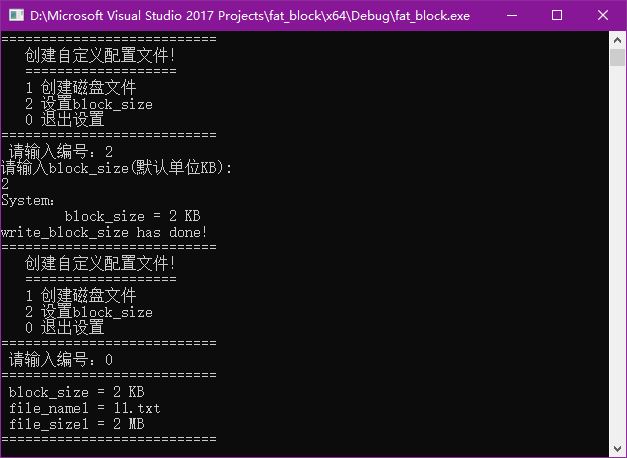
图7.2 读取配置文件

## 7.2 创建新配置文件define\_config.ini

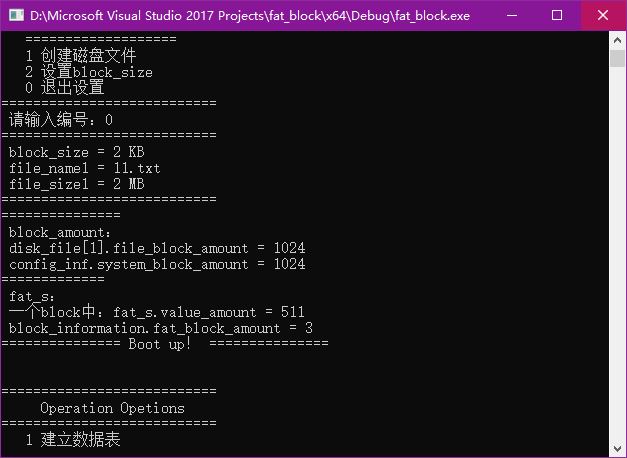
选择“2 创建新配置文件”选项，交互设定配置文件进行初始化操作。



(a) 配置文件内容



(b) 设置block\_size



(c) block相关计算

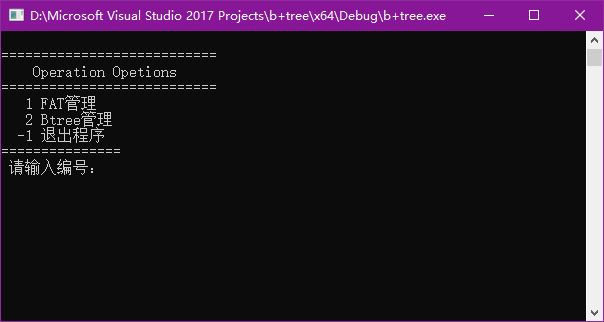
图 7.3交互配置



图 7.4 写入配置文件

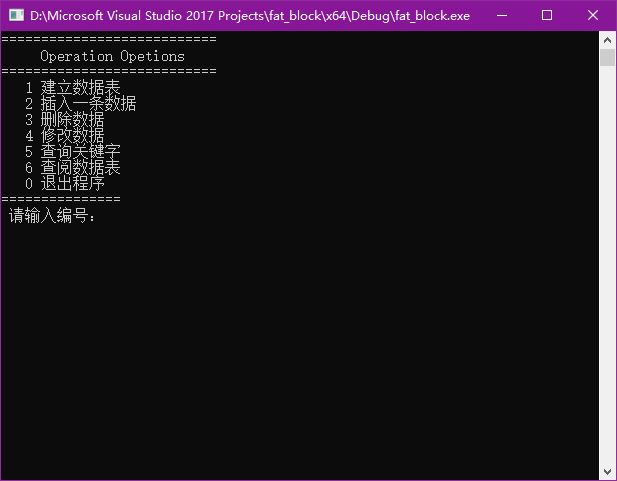
# 8 数据操作

控制台弹出“Operation Opetions”，用户可以通过输入选项编号选择管理数据方式。



(a)管理选项

控制台弹出所选管理方式的“Operation Opetions”，用户可以通过输入选项编号进行建立数据表、插入一条数据查询、删除数据、修改数据、查询关键字、查阅数据表操作。



(b)数据操作选项

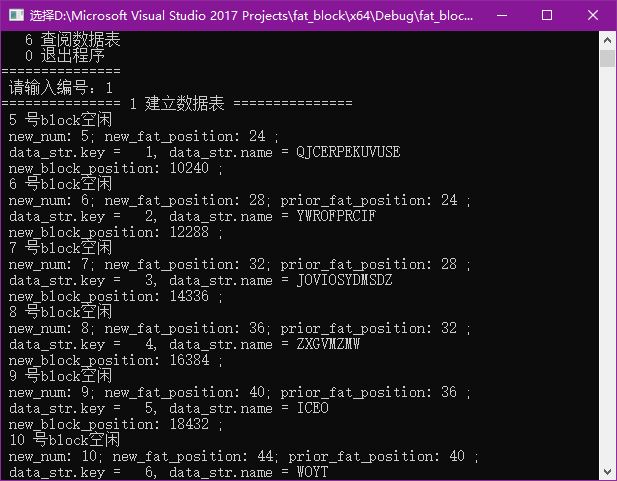
图8.1 Operation Opetions

## 8.1 建立数据表

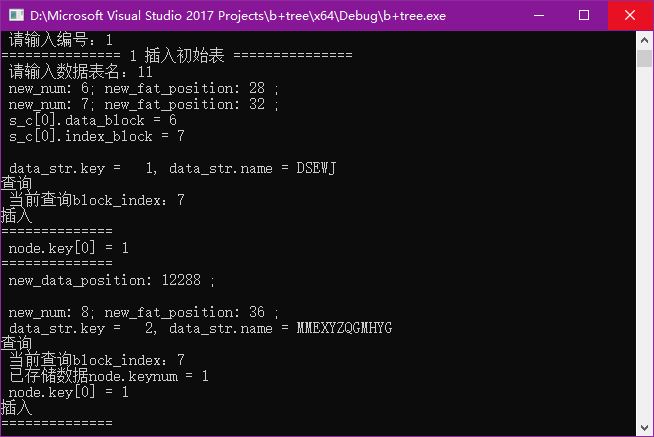
选择“1 建立数据表”，程序调用insert\_table()函数自动建立数据表。

在FAT管理数据中，insert\_table()函数调用fat\_block\_apply()函数申请空闲block，调用make\_name()函数生成姓名属性，将姓名和关键字写入空闲block。

在Btree管理数据中，insert\_table()函数调用btree\_block\_apply()函数申请空闲data\_block，调用make\_name()函数生成姓名属性，将姓名和关键字写入空闲block，申请空闲index\_block存储btree节点。



（a）fat建立数据表



(b)btree建立数据表

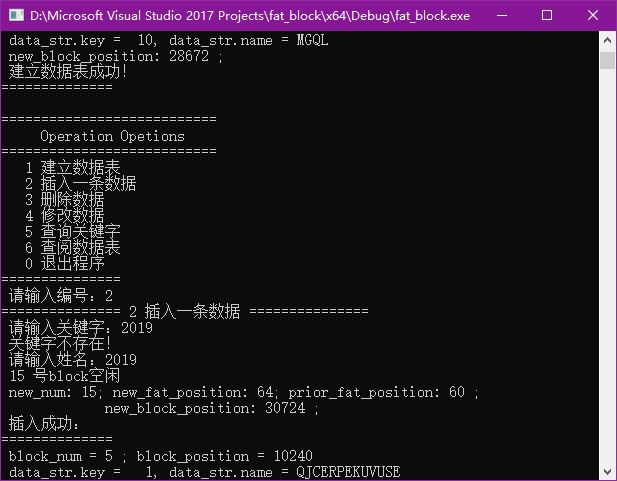
图 8.2 fat建立数据表

## 8.2 插入一条数据

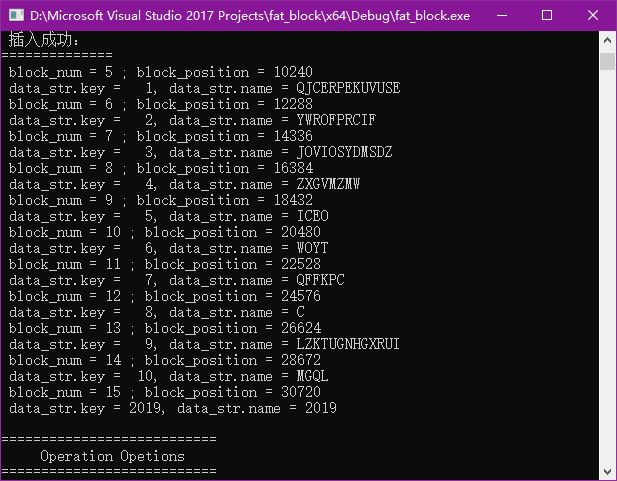
选择“2 插入一条数据”，程序调用相关函数获取关键字及其姓名属性。

在FAT管理数据中，程序调用fat\_insert\_data()函数获取关键字及其姓名属性。用户输入关键字，fat\_insert\_data()函数调用fat\_search\_data()函数查询关键字，若关键字存在则退出操作，若关键字不存在则获取姓名属性，调用fat\_block\_apply()函数申请空闲block并写入数据，将其对应fat表项值写入数据表中小于该关键字的关键字对应的fat表项中，使得查阅数据表时，关键字升序排序。

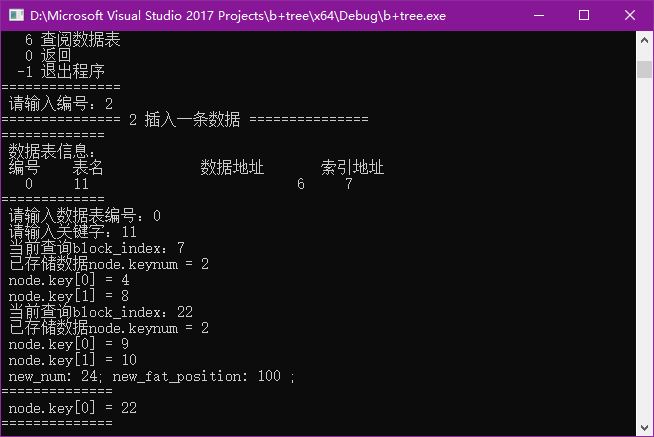
在Btree管理数据中，程序调用btree\_insert\_data()函数获取关键字及其姓名属性。用户输入关键字，btree\_insert\_data()函数调用btree\_search\_data()函数查询关键字，若关键字存在则退出操作，若关键字不存在则获取姓名属性，调用btree\_block\_apply()函数申请空闲block并写入数据，当btree节点分裂时，调用btree\_block\_apply()函数申请新节点，并进行相关节点数据调整。



(a) 插入操作



(b)查阅数据表



(c)btree插入数据

图 8.3 插入一条数据

## 8.3 删除数据

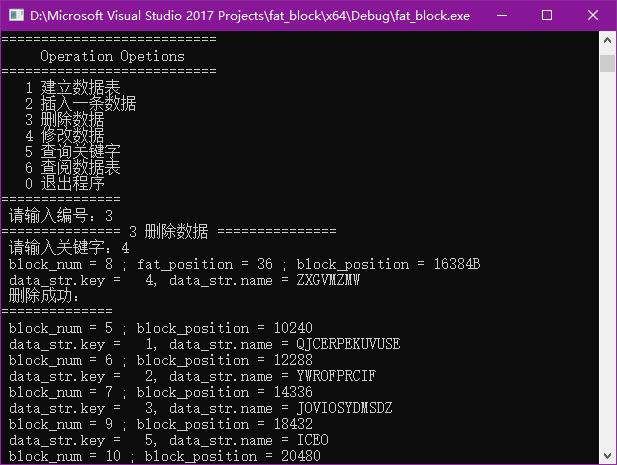
选择“3 删除数据”，程序调用相关函数删除关键字对应的block。

在FAT管理数据中，程序调用fat\_delete\_data()函数获取用户输入的要删除的关键字，fat\_delete\_data()函数调用fat\_searc\_data()函数查询关键字，若关键字不存在则退出操作，若关键字存在则将其fat表项值写入存当前block\_num的表项中，当前表项值置0。

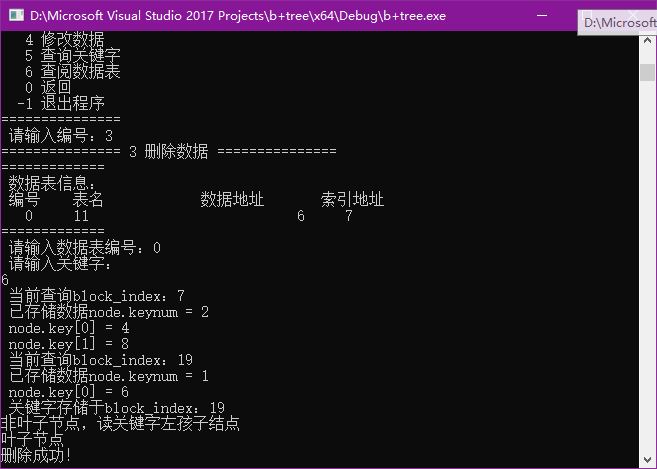
在Btree管理数据中，程序调用btree\_delete\_data()函数获取用户输入的要删除的关键字，btree\_delete\_data()函数调用btree\_searc\_data()函数查询关键字，若关键字不存在则退出操作，若关键字存在则将其data\_block对应的fat表项值置0，同时删除b树节点中的关键字及其指针，并且判断节点是否特征要求，如不符合则进行节点的合并。



(a) 删除关键字不存在



(b) 删除关键字存在



(c)btree删除数据

图8.4 删除数据

## 8.4 修改数据

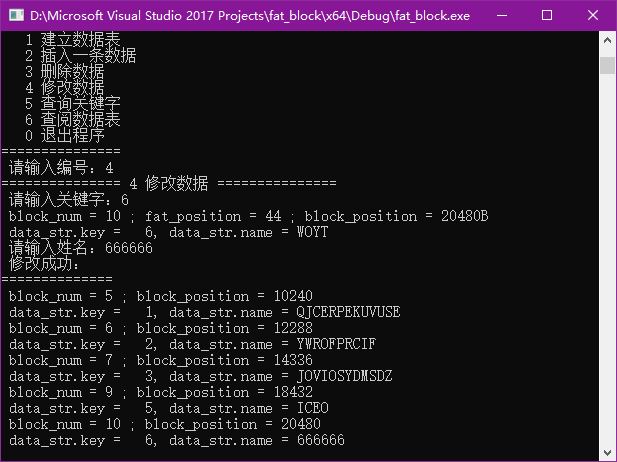
选择“4 修改数据”，程序调用相关函数修改关键字对应的数据选项。

在FAT管理数据中，程序调用fat\_alter\_data()函数获取用户输入要修改的关键字，fat\_alter\_data()函数调用fat\_search\_data()函数查询关键字，若关键字不存在则退出操作，若关键字存在则获取姓名属性，并将数据重新写入对应block中。

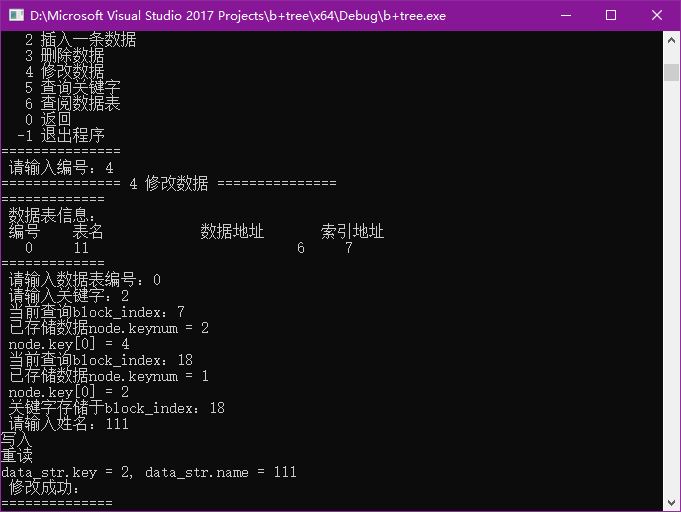
在Btree管理数据中，程序调用btree\_alter\_data()函数获取用户输入要修改的关键字，btree\_alter\_data()函数调用btree\_search\_data()函数查询关键字，若关键字不存在则退出操作，若关键字存在则获取姓名属性，并将数据重新写入对应block中。



(a) 修改关键字不存在



(b) 修改关键字存在



(c)btree修改数据

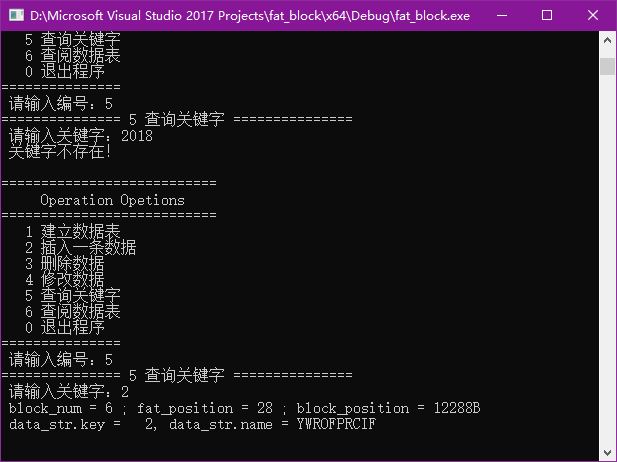
图 8.5 修改数据

## 8.5 查询数据

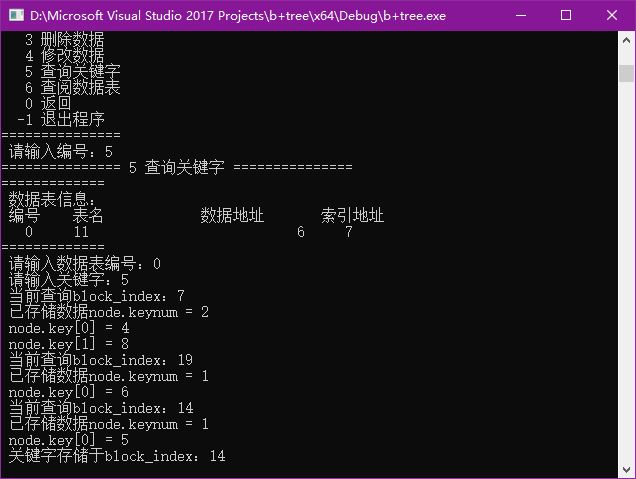
选择“5 查询数据”，程序调用相关函数查询关键字对应的数据。

在FAT管理数据中，程序调用fat\_search\_data()函数查询关键字对应的数据。用户输入关键字，fat\_search\_data()函数根据fat表项值遍历数据表，若关键字存在则读取block数据并输出，若关键字不存在则退出。

在Btree管理数据中，程序调用btree\_search\_data()函数查询关键字对应的数据。用户输入关键字，btree\_search\_data()函数从根节点开始查询比对关键字，并进行自身的递归调用，以查询数据表，若关键字存在则读取block数据并输出，若关键字不存在则退出。



(a)fat查询数据



(b)btree查询数据

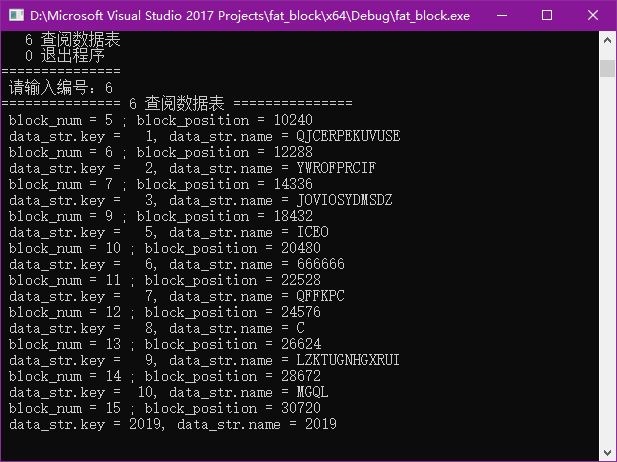
图 8.6 查询关键字

## 8.6 查阅数据表

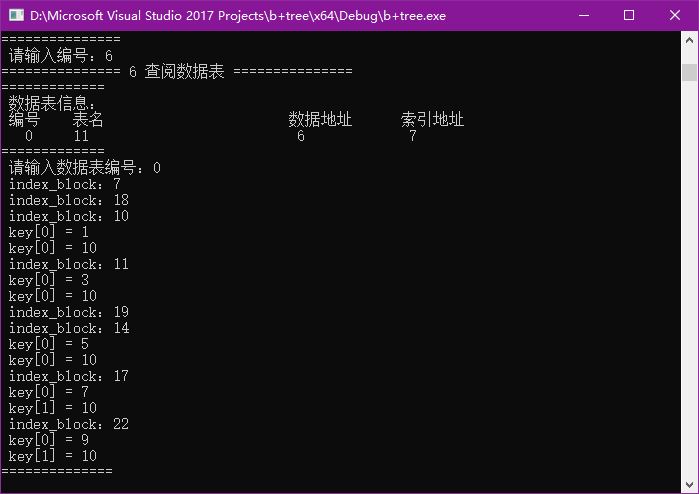
选择“6 查阅数据表”，程序调用相关函数读取数据表。

在FAT管理数据中，调用fat\_read\_table()函数根据fat表项值遍历数据表并输出到控制台，若未存储数据则弹出“无数据存储”。

在Btree管理数据中，调用btree\_read\_table()函数根据索引节点中的关键字遍历数据表并输出到控制台，若未存储数据则弹出“无数据存储”。



(a)fat查阅



(b)btree查阅

图 8.7 查阅数据表

## 8.7 退出程序

选择“0 退出程序”，关闭文件流，程序结束

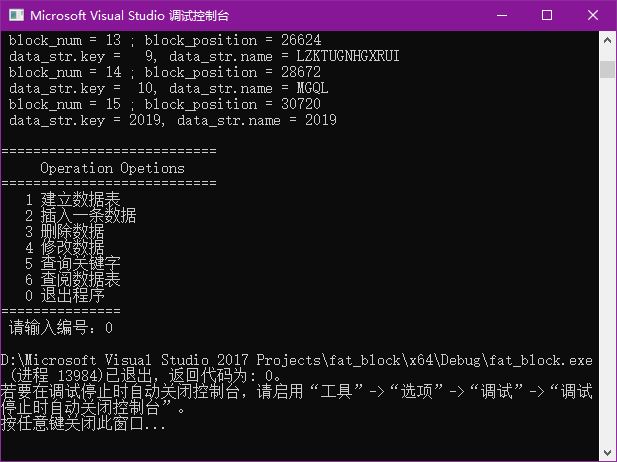


图 8.8 退出程序

# 总结

写程序锻炼大脑的灵活度。在此次数据库大作业中，我借鉴了FAT的思想，在磁盘文件中取前几个block用作文件分配表存储，在block读写的过程中，计算文件指针位置，不断切换于block在磁盘文件的位置和其在文件分配表中对应表项的位置，使得读写一致。

在编写过程中，我对FAT、block和Btree的结构和存储有了进一步的了解。在别写过程中不断地查阅相关资料，比如FAT的表项设置、block的映射等等。在编写初期，因为对二进制文件的读写不够熟练，造成了很大的困扰，也耽误了很多时间，最后发现主要原因是在文件读写时首先要用fseek()函数定位，多个位置时可用ftell()函数记录位置以达到准确定位的目的。

利用FAT完成增、删、改、查操作中，增、删操作较复杂。增加操作需要申请空闲block，同时需要在数据链表中特定位置插入该block\_num以形成升序排序的数据表，将该block\_num写入小于插入关键字中的最大的关键字对应的表项中，并将其原表项值写入该block\_num对应的表项。删除操作需要将该block\_num对应表项中的值写入值为该block\_num的表项中，并将该block\_num的表项值置0。

编写Btree的整个过程中思路还是比较清晰的，只是由于知识点的欠缺造成错误中断的次数比较多，最终都通过不断地在大脑中逻辑上的模拟运行和调试解决了。确实让大脑得到了很好的思维锻炼。

感谢李老师不厌其烦的指导，每次都我们讲的很仔细，每次听完都会有新的想法，编程思路和整体框架更加清晰，都会感觉自己还可以继续往下做。谢谢老师！

# 附录

1、pch.h文件：

#ifndef PCH\_H

#define PCH\_H

//config

#define FILE\_AMOUNT 10 //文件名个数

#define FILE\_NAME\_LENGTH 20 //文件名长度

#define Max\_Line\_LENGTH 1024 //每行最多读取字节数

//table数据表

#define DATA\_AMOUNT 10 //table数据表中数据个数

#define MAN\_NAME\_LENGTH 16 //人名长度

//#define Max\_FREE\_AMOUNT 20 //最大申请空闲个数

//head

#include<stdio.h>

#include<stdlib.h>

#include <string.h>

#include <math.h>

#include<time.h>

//config的信息

struct config\_information

{

int block\_size; //系统block大小

int disk\_file\_amount; //系统文件个数

int system\_block\_amount; //block个数

};

//每个磁盘文件的信息

struct disk\_file\_information

{

char file\_name[FILE\_NAME\_LENGTH];

int file\_size;

//int block\_size;

//int file\_amount;

int file\_block\_amount; //block个数

FILE \*file\_point; //指向本文件的文件指针

};

//data\_block structure

struct data\_block\_structure

{

int block\_num;

int file\_amount;

};

//fat\_block structure

struct fat\_block\_structure

{

//表项值：下一块地址；FF文件结束；BB坏；KB保留;0空闲。

int block\_num;//编号

int fat\_block\_amount;//fat占用block个数

int fat\_value\_amount;//fat的一个block放多少表项

};

//table数据表

struct data\_structure

{

int key;

char name[MAN\_NAME\_LENGTH];

};

#define MAXM 10 //定义B树的最大的阶数

#define m 3 //设定B树的阶数

#define Max m - 1 //结点的最大关键字数量

#define Min (m - 1) / 2 //结点的最小关键字数量

typedef int KeyType; //KeyType为关键字类型

//管理block

#define content\_num 2\*1024/(4+4+20)-4 //-4:块号

struct save\_content {

char table\_name[FILE\_NAME\_LENGTH];

int data\_block;

int index\_block;

};

typedef struct node { //B树和B树结点类型

int keynum; //结点关键字个数

KeyType key[m]; //关键字数组

int key\_block[m]; //存储关键字的block

int parent\_block\_num; //双亲结点指针

int child\_block\_num[m + 1]; //孩子结点指针数组

//int leaf;

//1是叶子节点，0不是叶子节点.若是叶子节点，孩子指针则为关键字数据存储的block

}bnode;

typedef enum function {

fat, //fat管理

btree //btree管理

}Function;

typedef enum status { //枚举类型（依次递增）

true,

false,

ok,

error,

overflow,

empty

}Status;

#endif //PCH\_H

2、btree.c文件：

#include "pch.h"

struct config\_information config\_inf; //config信息列表

struct disk\_file\_information disk\_file[FILE\_AMOUNT]; //每个磁盘文件的信息

struct fat\_block\_structure fat\_s;

struct save\_content s\_c[content\_num]; //目录块

struct data\_structure data\_str; //table数据表中数据格式

//系统初始选项列表

void system\_list()

{

printf("===========================\n");

printf(" B+tree System\n");

printf("===========================\n");

printf(" 操作选项：\n");

printf(" 1 使用配置文件\n");

printf(" 2 创建新配置文件\n");

printf(" 0 退出程序\n");

printf("===========================\n");

printf(" 请输入编号：");

}

//fat主操作选项

void manage\_ope\_list()

{

printf("\n===========================\n");

printf(" Operation Opetions\n");

printf("===========================\n");

printf(" 1 FAT管理\n");

printf(" 2 Btree管理\n");

printf(" -1 退出程序\n");

printf("===============\n");

//printf("===========================\n");

printf(" 请输入编号：");

}

//fat主操作选项

void fat\_operation\_list()

{

printf("\n===========================\n");

printf(" Fat Management\n");

printf("===========================\n");

}

//fat主操作选项

void btree\_operation\_list()

{

printf("\n===========================\n");

printf(" Btree Management\n");

printf("===========================\n");

}

//fat-btree操作选项

void fat\_btree\_operation\_list() {

printf(" 1 插入初始表\n");

printf(" 2 插入一条数据\n");

printf(" 3 删除数据\n");

printf(" 4 修改数据\n");

printf(" 5 查询关键字\n");

printf(" 6 查阅数据表\n");

printf(" 0 返回\n");

printf(" -1 退出程序\n");

printf("===============\n");

//printf("===========================\n");

printf(" 请输入编号：");

}

void content() {

int table\_num = 0;

long position;

printf("=============\n");

printf(" 数据表信息：\n");

printf(" 编号 表名 数据地址 索引地址\n");

position = fat\_s.fat\_block\_amount \* config\_inf.block\_size \* 1024;

position = position + sizeof(int) + table\_num \* sizeof(s\_c[0]); //越过块号，定位block中数据表位置

fseek(disk\_file[0].file\_point, position, 0); //定位文件起始

fread(&s\_c[table\_num], sizeof(s\_c[0]), 1, disk\_file[0].file\_point);

while (s\_c[table\_num].data\_block != 0)

{

printf(" %3d %-20s %-6d%9d\n", table\_num, s\_c[table\_num].table\_name, s\_c[table\_num].data\_block, s\_c[table\_num].index\_block);

table\_num++;

fread(&s\_c[table\_num], sizeof(s\_c[0]), 1, disk\_file[0].file\_point);

}

printf("=============\n");

}

void find() {

int unit;

fseek(disk\_file[0].file\_point, 24 + 1 \* sizeof(int), 0);

fread(&unit, sizeof(int), 1, disk\_file[0].file\_point);

printf("\n24 %d\n", unit);

fread(&unit, sizeof(int), 1, disk\_file[0].file\_point);

printf("25 %d\n", unit);

fread(&unit, sizeof(int), 1, disk\_file[0].file\_point);

printf("26 %d\n", unit);

fread(&unit, sizeof(int), 1, disk\_file[0].file\_point);

printf("27 %d\n\n", unit);

}

int main()

{

config\_inf.disk\_file\_amount = 0;

Function fun;

int reoperate1 = 1, reoperate2 = 1, fat\_reoperate = 1, btree\_reoperate = 1; //重新选择编号：1重选：0不选

static int choose; //选择默认/自定义config

int key, unit = 0; //查询关键字,获取前一block号

int table\_num = 0;; //表个数

int index\_num, data\_num; //btree\_search

Status flag = false, \*find\_flag = &flag;

//config set

while (reoperate1)

{

//系统初始选项列表

system\_list();

scanf("%d", &choose);

switch (choose)

{

case 1:

//读config

read\_config();

//block设置

block\_count();

fat\_inf();

fat\_initialize();

reoperate1 = 0;

break;

case 2:

//自定义config

creat\_define\_config();

//block设置

block\_count();

fat\_inf();

fat\_initialize();

reoperate1 = 0;

break;

case 0:

reoperate1 = 0;

reoperate2 = 0;

printf(" 0 退出程序\n");

break;

default:

printf(" 输入编号无效，请重新输入！\n");

break;

}

}

printf("=============== Boot up！ ===============\n\n");

//主操作

while (reoperate2)

{

//主操作选项

manage\_ope\_list();

scanf("%d", &choose);

switch (choose)

{

case 1: ////////// fat管理

fun = fat;

fat\_reoperate = 1;

while (fat\_reoperate)

{

//操作选项

fat\_operation\_list();

fat\_btree\_operation\_list();

scanf("%d", &choose);

switch (choose)

{

case 1:

//插入初始表

printf("=============== 1 插入初始表 ===============\n");

//find();

insert\_table(table\_num, fun);

//printf("%d\n", table\_num);

//find();

table\_num++;

break;

case 2:

//插入一条数据

printf("=============== 2 插入一条数据 ===============\n");

fat\_insert\_data();

break;

case 3:

//删除

printf("=============== 3 删除数据 ===============\n");

fat\_delete\_data();

break;

case 4:

//修改

printf("=============== 4 修改数据 ===============\n");

fat\_alter\_data();

break;

case 5:

//查询

printf("=============== 5 查询关键字 ===============\n");

//输入数据表编号

content();

printf(" 请输入数据表编号：");

scanf("%d", &table\_num);

//输入关键字

printf(" 请输入关键字：");

scanf("%d", &key);

unit = s\_c[table\_num].data\_block;

fat\_search\_data(table\_num, key, &unit);

break;

case 6:

//浏览browse

printf("=============== 6 查阅数据表 ===============\n");

content();

printf(" 请输入数据表编号：");

scanf("%d", &table\_num);

fat\_read\_table(table\_num);

break;

case 0:

//退出程序

fat\_reoperate = 0;

break;

case -1:

//退出程序

fat\_reoperate = 0;

reoperate2 = 0;

break;

default:

printf(" 输入编号无效，请重新输入！\n");

break;

}

}

break;

case 2: ////////// btree管理

fun = btree;

btree\_reoperate = 1;

while (btree\_reoperate)

{

//操作选项

btree\_operation\_list();

fat\_btree\_operation\_list();

scanf("%d", &choose);

switch (choose)

{

case 1:

//插入初始表

printf("=============== 1 插入初始表 ===============\n");

insert\_table(table\_num, fun);

//find();

table\_num++;

break;

case 2:

//插入一条数据

printf("=============== 2 插入一条数据 ===============\n");

content();

printf(" 请输入数据表编号：");

scanf("%d", &table\_num);

//输入关键字

printf(" 请输入关键字：");

scanf("%d", &key);

index\_num = btree\_search\_data(key, s\_c[table\_num].index\_block, find\_flag);

if (find\_flag == true)

{

printf(" 关键字已存在！\n");

}

else

{

data\_num = btree\_block\_apply();

btree\_insert\_data(key, index\_num, data\_num);

}

break;

case 3:

//删除

printf("=============== 3 删除数据 ===============\n");

//查找

content();

printf(" 请输入数据表编号：");

scanf("%d", &table\_num);

//输入关键字

printf(" 请输入关键字：");

scanf("%d", &key);

index\_num = btree\_search\_data(key, s\_c[table\_num].index\_block, find\_flag);

btree\_delete\_data(key, index\_num, flag);

break;

case 4:

//修改

printf("=============== 4 修改数据 ===============\n");

//查找

content();

printf(" 请输入数据表编号：");

scanf("%d", &table\_num);

//输入关键字

printf(" 请输入关键字：");

scanf("%d", &key);

btree\_alter\_data(table\_num, key);

break;

case 5:

//查询

printf("=============== 5 查询关键字 ===============\n");

content();

printf(" 请输入数据表编号：");

scanf("%d", &table\_num);

//输入关键字

printf(" 请输入关键字：");

scanf("%d", &key);

btree\_search\_data(key, s\_c[table\_num].index\_block, find\_flag);

break;

case 6:

//浏览browse

printf("=============== 6 查阅数据表 ===============\n");

content();

printf(" 请输入数据表编号：");

scanf("%d", &table\_num);

btree\_read\_table(s\_c[table\_num].index\_block);

break;

case 0:

//退出程序

btree\_reoperate = 0;

break;

case -1:

//退出程序

btree\_reoperate = 0;

reoperate2 = 0;

break;

default:

printf(" 输入编号无效，请重新输入！\n");

break;

}

printf("==============\n");

}

break;

case -1:

reoperate2 = 0;

break;

default:

printf(" 输入编号无效，请重新输入！\n");

break;

}

}

//关闭文件指针

for (int i = 0; i < config\_inf.disk\_file\_amount; i++)

{

fclose(disk\_file[i].file\_point);

}

}

3、read\_config.c文件

#include "pch.h"

extern struct config\_information config\_inf; //config信息列表//b+tree.c

extern struct disk\_file\_information disk\_file[FILE\_AMOUNT]; //b+tree.c

extern void creat\_define\_config(); //writer\_config.c

static FILE \*config\_point;

static char lineBuf[Max\_Line\_LENGTH] = { 0 }; //行缓冲

static char \*name\_judge = NULL; //name开始位置

static int kind\_flag; //0.flag\_time; 1.block; 2.File\_name; 3.File\_size

static int temp; //临时变量

static int rebuild\_flag = 1; //是否重建表标志，1：默认重建；0：不建

void get\_value()

{

//char \*mark\_temp = NULL; //mark position

char \*value\_start = NULL; //value开始位置

char \*value\_endif = NULL; //value结束位置

//char \*temp = NULL; //temp position

char \*tempValue = NULL; //temp position 读size用

int len1;

//char value\_buf[FILE\_NAME\_LENGTH] = { 0 }; //文件名缓冲

//value start position

value\_start = strchr(lineBuf, '=') + 1;//lineBuf中字符和=匹配，指针指向=后一个字符

//printf("start=%s\n",start);

value\_endif = value\_start;

//0.flag\_time; 1.block; 2.File\_name; 3.File\_size

switch (kind\_flag)

{

case 0:

//0.flag\_time

//int flag\_time; //flag\_time 初次读取config？

printf(" flag\_time = %d\n", atoi(value\_start)); //atoi翻译字符串到整型//1:have read ,0 have not read

if (atoi(value\_start))

{

printf(" Have read config! Do you want to build file again?(1:yes/0:no)\n");

scanf("%d", &temp);

if (temp == 0) // "= return 1"

{

rebuild\_flag = 0;//不重建

//printf("")

}

}

break;

case 1:

//1.block\_size

while (1) {

if (\*value\_endif == 'K') {

break;

}

else {

value\_endif++;

}

}

len1 = value\_endif - value\_start;//为了计算正确长度，不减一：0 1 2 k，3-0=3个

tempValue = (char\*)malloc(len1 \* sizeof(char));

memcpy(tempValue, value\_start, len1);//将以start指针开始的en1个字节的内容复制到tempValue中,tempValue是指针或数组都可以

config\_inf.block\_size = atoi(tempValue);//把字符串temp转换为整形

printf(" block\_size = %d KB\n", config\_inf.block\_size);

break;

case 2:

//2.File\_name

while (1) {

if (\*value\_endif == '\n') {

break;

}

else {

value\_endif++;

}

}

for (int i = 0; value\_start + i < value\_endif; i++)

{

disk\_file[config\_inf.disk\_file\_amount].file\_name[i] = \*(value\_start + i);

}

printf(" file\_name%d = %s\n", config\_inf.disk\_file\_amount + 1, disk\_file[config\_inf.disk\_file\_amount].file\_name);

break;

case 3:

//3.File\_size

while (1) {

if (\*value\_endif == 'M') {

break;

}

else {

value\_endif++;

}

}

len1 = value\_endif - value\_start;//为了计算正确长度，不减一：0 1 2 k，3-0=3个

tempValue = (char\*)malloc(len1 \* sizeof(char));

memcpy(tempValue, value\_start, len1);//将以start指针开始的en1个字节的内容复制到tempValue中,tempValue是指针或数组都可以

disk\_file[config\_inf.disk\_file\_amount].file\_size = atol(tempValue);//把字符串temp转换为整形

printf(" file\_size%d = %d MB\n", config\_inf.disk\_file\_amount + 1, disk\_file[config\_inf.disk\_file\_amount].file\_size);

break;

default:

break;

}

}

//read "config.ini"

int read\_config()

{

int flag\_flag = 0;

int block\_flag = 0;

long now;//记录config\_point位置

//open file 写flag

if ((config\_point = fopen("config.ini", "r+")) == NULL) //rb代表按照二进制的方式进行读

{

printf(" Can't open config.ini!\nPlease define config.ini！(1:yes/0:no)\n");

scanf("%d", &temp);

if (temp == 1) // "= return 0"

{

creat\_define\_config();

}

else

{

printf(" So byebye！\n");

exit(0);

}

}

//circulation read a line

while (1)

{

//mate name 根据行前边的名字判断本行是什么信息

name\_judge = fgets(lineBuf, Max\_Line\_LENGTH, config\_point);//一次读取一行,每次最多读取1024个字节,读取第一行

//printf("%s\n", name\_judge);//第一行的所有内容

if (name\_judge == NULL)

{

printf(" Can't read new line!\n");

break;

}

if (\*(name\_judge) == 'f' && flag\_flag == 0)

{

//flag

kind\_flag = 0;

get\_value();

flag\_flag = 1;

//write flag\_time 临时point，使config\_point正常读

//fp = config\_point;

now = ftell(config\_point);

fseek(config\_point, 10, 0);// 将文件的指针 移至 指定大小的位置

//写入1，表示已读

fputc('1', config\_point);

//fclose(fp);

fseek(config\_point, now, 0);// 定位开头，重新到行结束

//name\_judge = fgets(lineBuf, Max\_Line\_LENGTH, config\_point);//一次读取一行,每次最多读取1024个字节,读取第一行

}

else if (\*(name\_judge) == 'b' && block\_flag == 0)

{

//block

kind\_flag = 1;

get\_value();

block\_flag = 1;

}

else if (\*(name\_judge + 5) == 'n')

{

//File\_name

kind\_flag = 2;

get\_value();

//disk\_file.file\_amount++;//文件大小取定，这个文件才结束

}

else if (\*(name\_judge + 5) == 's')

{

//File\_size

kind\_flag = 3;

get\_value();

config\_inf.disk\_file\_amount++;

}/\*

else

{

printf("Have not mate name!\n");

}\*/

}

//创建磁盘文件

if (rebuild\_flag)

{

//覆盖

int control\_size = 0;

for (int i = 0; i < config\_inf.disk\_file\_amount; i++)

{

//为了读写，建立一个新的二进制文件

if ((disk\_file[i].file\_point = fopen(disk\_file[i].file\_name, "wb+")) == NULL)

{

printf(" 不能打开:%s\n", disk\_file[i].file\_name);

//exit(0);

}

//printf("........\n");

fseek(disk\_file[i].file\_point, disk\_file[i].file\_size \* 1024 \* 1024 - 1, 0);// 将文件的指针 移至 指定大小的位置

//在要指定大小文件的末尾随便放一个数据,如果不写此句，则创建出的文件的大小都为0KB

fputc(0, disk\_file[i].file\_point);//在要指定大小文件的末尾随便放一个数据,如果不写此句，则创建出的文件的大小都为0KB

}

}

else

{

//不覆盖

for (int i = 0; i < config\_inf.disk\_file\_amount; i++)

{

//为了读写，打开一个二进制文件

if ((disk\_file[i].file\_point = fopen(disk\_file[i].file\_name, "rb+")) == NULL)

{

printf(" 不能打开:%s\n", disk\_file[i].file\_name);

//exit(0);

}

}

}

//关流

fclose(config\_point);

return rebuild\_flag;

}

4、write\_config.c文件

// b+tree.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结束。

#include "pch.h"

extern struct config\_information config\_inf; //config信息列表//b+tree.c

extern struct disk\_file\_information disk\_file[FILE\_AMOUNT]; //b+tree.c

static FILE \*d\_config\_point;

//creat file and write in define\_config.ini

void write\_file()

{

//char file\_name[FILE\_NAME\_LENGTH];

long file\_size;

//获取输入

printf(" 请输入file\_name(默认文件后缀.txt，无需输入):\n");

scanf("%s", disk\_file[config\_inf.disk\_file\_amount].file\_name);

printf(" 请输入file\_size(默认单位MB):\n");

scanf("%d", &disk\_file[config\_inf.disk\_file\_amount].file\_size);

//write "txt" in file\_name

strcat(disk\_file[config\_inf.disk\_file\_amount].file\_name, ".txt");

//创建磁盘文件

//为了读写，建立一个新的二进制文件

if ((disk\_file[config\_inf.disk\_file\_amount].file\_point = fopen(disk\_file[config\_inf.disk\_file\_amount].file\_name, "wb+")) == NULL)

{

printf(" 不能打开:%s\n", disk\_file[config\_inf.disk\_file\_amount].file\_name);

//exit(0);

}

fseek(disk\_file[config\_inf.disk\_file\_amount].file\_point, disk\_file[config\_inf.disk\_file\_amount].file\_size \* 1024 \* 1024 - 1, 0);// 将文件的指针 移至 指定大小的位置

fputc(0, disk\_file[config\_inf.disk\_file\_amount].file\_point);//在要指定大小文件的末尾随便放一个数据,如果不写此句，则创建出的文件的大小都为0KB

//写入

fprintf(d\_config\_point, "file\_name%d = %s\n", config\_inf.disk\_file\_amount+1, disk\_file[config\_inf.disk\_file\_amount].file\_name);

fprintf(d\_config\_point, "file\_size%d = %d MB\n", config\_inf.disk\_file\_amount+1, disk\_file[config\_inf.disk\_file\_amount].file\_size);

//输出

printf("System:\n\tfile\_name%d = %s\n", config\_inf.disk\_file\_amount + 1, disk\_file[config\_inf.disk\_file\_amount].file\_name);

printf("\tfile\_size%d = %d MB\n", config\_inf.disk\_file\_amount + 1, disk\_file[config\_inf.disk\_file\_amount].file\_size);

//文件总数

config\_inf.disk\_file\_amount++;

}

//get block\_size

void write\_block\_size()

{

//获取输入

printf("请输入block\_size(默认单位KB):\n");

scanf("%d", &config\_inf.block\_size);

printf("System：\n\tblock\_size = %d KB\n", config\_inf.block\_size);

printf("write\_block\_size has done!\n");

//连接、写入config

//strcat(block\_size, size);

//strcat(block\_size, "MB");

//写入

fprintf(d\_config\_point, "block\_size = %d KB\n", config\_inf.block\_size);

//fwrite(file\_name, sizeof(char), strlen(file\_name), file\_point);//二进制读写

}

//creat\_define\_config list

void setting\_list()

{

printf("===========================\n");

printf(" 创建自定义配置文件！\n");

printf(" ===================\n");

printf(" 1 创建磁盘文件\n");

printf(" 2 设置block\_size\n");

printf(" 0 退出设置\n");

printf("===========================\n");

printf(" 请输入编号：");

}

//输出配置

void display()

{

printf("===========================\n");

printf(" block\_size = %d KB\n", config\_inf.block\_size);

for (int i = 0; i < config\_inf.disk\_file\_amount; i++)

{

printf(" file\_name%d = %s\n", i + 1, disk\_file[i].file\_name);

printf(" file\_size%d = %d MB\n", i + 1, disk\_file[i].file\_size);

}

printf("===========================\n");

}

//创建文件并写入config

void creat\_define\_config()//define\_config.ini指针

{

char define\_config\_name[] = "define\_config.ini";

//char filename[] = "./define\_config.ini";

static int choose = 1;//目录选项

static int flag = 1;//block\_size只能设置一次

//创建配置文件

//file\_point = fopen(filename, "a");//追加读写，不存在则创建

fopen\_s(&d\_config\_point, define\_config\_name, "w");//写，不存在则创建

while (choose)

{

setting\_list();//creat\_define\_config list

scanf("%d", &choose);

switch (choose)

{

case 1:

write\_file();

break;

case 2:

if (flag)

{

write\_block\_size();

flag = 0;

}

else

{

printf(" write\_block\_size was already done!\n");

}

break;

case 0:

default:

break;

}

}

//输出

display();

fclose(d\_config\_point);

}

5、block\_count.c文件

#include "pch.h"

extern struct config\_information config\_inf; //config信息列表 b+tree.c

extern struct disk\_file\_information disk\_file[FILE\_AMOUNT]; //每个磁盘文件的信息 b+tree.c

extern struct fat\_block\_structure fat\_s; //b+tree.c

//block信息计算

void block\_count()

{

//初始磁盘文件数为0

config\_inf.system\_block\_amount = 0;

//count block amount

printf("===============\n");

printf(" block\_amount：\n");

for (unsigned int i = 0; i < config\_inf.disk\_file\_amount; i++)

{

disk\_file[i].file\_block\_amount = disk\_file[i].file\_size \* 1024 / config\_inf.block\_size;

printf(" disk\_file[%d].file\_block\_amount = %d\n", i + 1, disk\_file[i].file\_block\_amount);

config\_inf.system\_block\_amount += disk\_file[i].file\_block\_amount;

}

printf(" config\_inf.system\_block\_amount = %d\n", config\_inf.system\_block\_amount);

}

6、fat\_manage.c文件

#include "pch.h"

extern struct disk\_file\_information disk\_file[FILE\_AMOUNT]; //每个磁盘文件的信息//b+tree.c

extern struct config\_information config\_inf; //config信息列表//b+tree.c

extern struct fat\_block\_structure fat\_s; //b+tree.c

/\*

单块block申请：

寻找新块号；

寻找新块号，将新块号写入上一块号；

写入结束符。

\*/

int fat\_block\_apply(int apply\_flag, int first\_apply, int prior\_num) {

/\*

形参：

apply\_flag:0结束申请（文件结尾），1申请；

first\_apply:1是第一次申请，0否；

prior\_num：当前存储的block的号，即将为前一block号。

return值：

0：无空闲；

num：块号。

\*/

int no\_free = 1; //1:无空闲：0：有空闲。

int unit; //表项值 =1 表示结束

int new\_num; //新的block号

long prior\_fat\_position, new\_fat\_position, write\_position;

int i, j;

fseek(disk\_file[0].file\_point, 0, 0); //定位起始

//fat块循环找

for (i = 0; i < fat\_s.fat\_block\_amount; i++)

{

fseek(disk\_file[0].file\_point, sizeof(int), 1); //跳过每块的块号

//定位每个block中的起始数据表项

//0块里存的还有与fat对应的单元格

if (!i)

{

j = fat\_s.fat\_block\_amount + 1;

fseek(disk\_file[0].file\_point, j \* sizeof(int), 1);

//fat占用block个数的数值 = 第一个存数据的表项值标签对应的块号的数值；

}

else

{

j = 0;

}

//表项值循环找

for (; j < fat\_s.fat\_value\_amount; j++)

{

fread(&unit, sizeof(int), 1, disk\_file[0].file\_point);

if (!unit) //是0，空闲block

{

new\_num = j + i \* fat\_s.fat\_value\_amount;

if (new\_num != prior\_num) //新空闲块不是上一还未写入表项值的空闲块

{

//记录新block位置

write\_position = sizeof(int);

fseek(disk\_file[0].file\_point, -write\_position, 1);

new\_fat\_position = ftell(disk\_file[0].file\_point);

//结束申请

if (apply\_flag == 0)

{

//写入结束值“1”

unit = 1;

fwrite(&unit, sizeof(int), 1, disk\_file[0].file\_point);

}

if (first\_apply)

{

printf(" new\_num: %d; new\_fat\_position: %d ;\n", new\_num, new\_fat\_position);

}

else

{

prior\_fat\_position = ((prior\_num / fat\_s.fat\_value\_amount + 1) + prior\_num) \* sizeof(int);

fseek(disk\_file[0].file\_point, prior\_fat\_position, 0);

fwrite(&new\_num, sizeof(int), 1, disk\_file[0].file\_point);

printf(" new\_num: %d; new\_fat\_position: %d ; prior\_fat\_position: %d ;\n", new\_num, new\_fat\_position, prior\_fat\_position);

write\_position = sizeof(int);

fseek(disk\_file[0].file\_point, -write\_position, 1);

fread(&new\_num, sizeof(int), 1, disk\_file[0].file\_point);

}

//no\_free = 0; //有空闲

return new\_num;

}

}

}

}

//未找到空闲

if (no\_free)

{

return 0;

}

}

int btree\_block\_apply() {

/\*

形参：

apply\_flag:0结束申请（文件结尾），1申请；

first\_apply:1是第一次申请，0否；

prior\_num：当前存储的block的号，即将为前一block号。

return值：

0：无空闲；

num：块号。

\*/

int no\_free = 1; //1:无空闲：0：有空闲。

int unit; //表项值 =1 表示结束

int new\_num; //新的block号

long new\_fat\_position, write\_position;

int first\_unit = 0; //起始数据表项

fseek(disk\_file[0].file\_point, 0, 0); //定位起始

//fat块循环找

for (int i = 0; i < fat\_s.fat\_block\_amount; i++)

{

fseek(disk\_file[0].file\_point, sizeof(int), 1); //跳过每块的块号

//定位每个block中的起始数据表项

//0块里存的还有与fat对应的单元格

if (!i)

{

//fat占用block个数的数值 = 第一个存数据和索引的表项值标签对应的块号的数值（只用来固定存第一块索引）

//第二块开始存数据或其他索引

first\_unit = fat\_s.fat\_block\_amount + 1;

fseek(disk\_file[0].file\_point, first\_unit \* sizeof(int), 1);

}

//表项值循环找

for (int j = first\_unit; j < fat\_s.fat\_value\_amount; j++)

{

fread(&unit, sizeof(int), 1, disk\_file[0].file\_point);

if (!unit) //是0，空闲block

{

//记录新block位置

write\_position = sizeof(int);

fseek(disk\_file[0].file\_point, -write\_position, 1);

//写入结束值“1”

unit = 1;

fwrite(&unit, sizeof(int), 1, disk\_file[0].file\_point);

//disk\_file[0].file\_block\_amount起始block\_index

//new\_num = j + i \* fat\_s.fat\_value\_amount + disk\_file[0].file\_block\_amount;用第二个文件存索引时

new\_num = j + i \* fat\_s.fat\_value\_amount;

new\_fat\_position = ((new\_num / fat\_s.fat\_value\_amount + 1) + new\_num) \* sizeof(int);

printf(" new\_num: %d; new\_fat\_position: %d ;\n", new\_num, new\_fat\_position);

return new\_num;

}

}

}

//未找到空闲

if (no\_free)

{

return 0;

}

}

7、insert\_table.c文件

//1 插入初始表

#include "pch.h"

extern struct disk\_file\_information disk\_file[FILE\_AMOUNT]; //每个磁盘文件的信息//b+tree.c

extern struct fat\_block\_structure fat\_s; //b+tree.c

extern struct config\_information config\_inf; //config信息列表

extern struct save\_content s\_c[content\_num]; //目录块

struct data\_structure data\_str; //table数据表中数据格式

extern int fat\_block\_apply(int apply\_flag, int first\_apply, int prior\_num);

extern int btree\_block\_apply();

extern int root\_index\_initialize();

extern void btree\_insert\_data(int table\_num, int key, int index\_num, int data\_num);

extern int btree\_search\_data(int key, int block\_index, Status \*find\_flag);

//产生随机名

void make\_name(char name[]) {

int name\_length;

name\_length = rand() % (MAN\_NAME\_LENGTH - 1) + 1; //1-15

for (int i = 0; i < name\_length; i++)

{

name[i] = rand() % 26 + 65; //小写英文字母

}

name[name\_length] = '\0';

}

//产生初始信息表(名单)，写序号(关键字)、姓名

void insert\_table(int table\_num, Function fun) {

int data\_num, index\_num; //新的block号

int new\_position; //fat中哪一个block的哪一表项

char name[MAN\_NAME\_LENGTH];

srand(time(NULL));

long write\_position; //申请块号的位置，用于测试读数据

Status flag = false, \*find\_flag = &flag;

//printf("=============== 1 插入初始表 ===============\n");

printf(" 请输入数据表名：");

scanf("%s", s\_c[table\_num].table\_name);

//写入数据表

for (int i = 1; i <= DATA\_AMOUNT; i++)

{

//记录生成数据

data\_str.key = i;

make\_name(name);

strcpy(data\_str.name, name);

//根据管理方式以申请空闲block

if (fun == fat)

{

//申请空闲block

if (i == 1)

{

//申请首个空闲数据块

data\_num = fat\_block\_apply(1, 1, 0);

//存到s\_c的block中

s\_c[table\_num].data\_block = data\_num;

s\_c[table\_num].index\_block = 0;

new\_position = fat\_s.fat\_block\_amount \* config\_inf.block\_size \* 1024;

new\_position = new\_position + sizeof(int) + table\_num \* sizeof(s\_c[0]); //+越过块号+定位block中数据表位置

fseek(disk\_file[0].file\_point, new\_position, 0); //定位文件起始

fwrite(&s\_c[table\_num], sizeof(s\_c[0]), 1, disk\_file[0].file\_point);

//printf(" %d 号block空闲\n", data\_num);

}

else if (i == DATA\_AMOUNT)

{

//结束申请,fat中最有一个表项存结束值1

data\_num = fat\_block\_apply(0, 0, data\_num);

}

else

{

//其中申请

data\_num = fat\_block\_apply(1, 0, data\_num);

}

}

else //if (fun == btree)

{

if (i == 1)

{

//申请首个空闲块

data\_num = btree\_block\_apply();

//存到s\_c的block中

s\_c[table\_num].data\_block = data\_num;

s\_c[table\_num].index\_block = root\_index\_initialize();

printf(" s\_c[%d].data\_block = %d\n", table\_num, s\_c[table\_num].data\_block);

printf(" s\_c[%d].index\_block = %d\n\n", table\_num, s\_c[table\_num].index\_block);

new\_position = fat\_s.fat\_block\_amount \* config\_inf.block\_size \* 1024;

new\_position = new\_position + sizeof(int) + table\_num \* sizeof(s\_c[0]); //+越过块号+定位block中数据表位置

fseek(disk\_file[0].file\_point, new\_position, 0); //定位文件起始

fwrite(&s\_c[table\_num], sizeof(s\_c[0]), 1, disk\_file[0].file\_point);

}

else if (i == DATA\_AMOUNT)

{

//结束申请,fat中最有一个表项存结束值1

data\_num = btree\_block\_apply();

}

else

{

//其中申请

data\_num = btree\_block\_apply();

}

}

//无空闲

if (data\_num == 0)

{

printf(" 空闲不足！\n");

return 0;

}

printf(" data\_str.key = %3d, data\_str.name = %s\n", data\_str.key, data\_str.name);

//写index

if (fun == btree)

{

printf("查询\n");

index\_num = btree\_search\_data(data\_str.key, s\_c[table\_num].index\_block, find\_flag);

//////自动建表，所以关键字不会重复

printf("插入\n");

btree\_insert\_data(table\_num, data\_str.key, index\_num, data\_num);

}

//写入data

new\_position = data\_num \* config\_inf.block\_size \* 1024;

printf(" new\_data\_position: %d ;\n", new\_position);

fseek(disk\_file[0].file\_point, new\_position + sizeof(int), 0); //越过块号

fwrite(&data\_str, sizeof(data\_str), 1, disk\_file[0].file\_point);

printf("\n");

}

printf(" 插入初始表成功！\n");

}

8、fat\_insert\_data.c文件

//2 插入一条数据

#include "pch.h"

extern struct disk\_file\_information disk\_file[FILE\_AMOUNT]; //每个磁盘文件的信息//b+tree.c

extern struct fat\_block\_structure fat\_s; //b+tree.c

extern struct config\_information config\_inf; //config信息列表

struct data\_structure data\_str; //table数据表中数据格式

extern int query\_data(int key, int \*target\_unit); //query\_data.c

extern void read\_table(); //read\_table.c

void insert\_data() {

int unit, next\_unit; //表项值=1 表示该表项值有数据，下一表项值，

int new\_num; //新的block号

long fat\_position, block\_position;

int query\_result = 2; //查询关键字返回值：0无数据；1未找到关键字；2存在关键字

int small\_unit = 0 ;

int key; //数据关键字

char name[MAN\_NAME\_LENGTH]; //数据人名

//printf("=============== 2 插入一条数据 ===============\n");

//查找

//输入关键字

printf(" 请输入关键字：");

scanf("%d", &key);

query\_result = query\_data(key, &small\_unit);

if (query\_result == 2)

{

printf(" 关键字已存在!\n");

}

else //插入

{

printf(" 请输入姓名：");

scanf("%s", &name);

data\_str.key = key;

strcpy(data\_str.name, name);

//无存储数据

if (query\_result == 0)

{

//申请首块空闲

new\_num = fat\_block\_apply(1, 1, 0);

block\_position = new\_num \* config\_inf.block\_size \* 1024 + sizeof(int); //越过块号

//写入数据位置

printf(" new\_block\_position: %d ;\n", block\_position);

fseek(disk\_file[0].file\_point, block\_position, 0);

fwrite(&data\_str, sizeof(data\_str), 1, disk\_file[0].file\_point);

printf(" 插入成功！\n");

//读表

read\_table();

}

else //有存储数据

{

//不存在关键字，query\_data()返回记录小于关键字中的最大关键字位于的块号

fat\_position = ((small\_unit / fat\_s.fat\_value\_amount + 1) + small\_unit) \* sizeof(int);

fseek(disk\_file[0].file\_point, fat\_position, 0);

fread(&next\_unit, sizeof(int), 1, disk\_file[0].file\_point);

//printf("%d\n", small\_unit);

//printf("%d\n", next\_unit);

//插入数据的block号

new\_num = fat\_block\_apply(1, 0, small\_unit);

//printf("%d\n", new\_num);

//插入数据申请的block对应的fat表项值写入下一fat表项值

fat\_position = ((new\_num / fat\_s.fat\_value\_amount + 1) + new\_num) \* sizeof(int);

fseek(disk\_file[0].file\_point, fat\_position, 0);

fwrite(&next\_unit, sizeof(int), 1, disk\_file[0].file\_point);

//插入数据位置，并写入

block\_position = new\_num \* config\_inf.block\_size \* 1024 + sizeof(int); //越过块号

printf(" new\_block\_position: %d ;\n", block\_position);

fseek(disk\_file[0].file\_point, block\_position, 0);

fwrite(&data\_str, sizeof(data\_str), 1, disk\_file[0].file\_point);

printf(" 插入成功！\n==============\n");

//读表

read\_table();

}

}

}

9、fat\_delete\_data.c文件

//3 删除

#include "pch.h"

extern struct disk\_file\_information disk\_file[FILE\_AMOUNT]; //每个磁盘文件的信息//b+tree.c

extern struct fat\_block\_structure fat\_s; //b+tree.c

struct data\_structure data\_str; //table数据表中数据格式

extern int query\_data(int key, int \*target\_unit); //query\_data.c

extern void read\_table(); //read\_table.c

void delete\_data() {

int unit, next\_unit; //表项值=1 表示该表项值有数据，下一表项值，

int new\_num; //新的block号

long fat1\_position, fat2\_position;

int query\_result; //查询关键字返回值：0无数据；1未找到关键字；2存在关键字

int target\_unit = 0;

int key; //数据关键字

char name[MAN\_NAME\_LENGTH]; //数据人名

//printf("=============== 3 删除数据 ===============\n");

//查找

//输入关键字

printf(" 请输入关键字：");

scanf("%d", &key);

query\_result = query\_data(key, &target\_unit);

//\*target\_unit记录小于关键字中的最大关键字位于的块号，即目标关键字block的前一block号

//printf(" small\_unit = %d\n", target\_unit);

if (query\_result == 2) //删除

{

//目标关键字的block号

fat1\_position = ((target\_unit / fat\_s.fat\_value\_amount + 1) + target\_unit) \* sizeof(int);

fseek(disk\_file[0].file\_point, fat1\_position, 0);

fread(&unit, sizeof(int), 1, disk\_file[0].file\_point); //unit：目标关键字的block号

//目标关键字的下一block号

fat2\_position = ((unit / fat\_s.fat\_value\_amount + 1) + unit) \* sizeof(int);

fseek(disk\_file[0].file\_point, fat2\_position, 0);

fread(&next\_unit, sizeof(int), 1, disk\_file[0].file\_point);

unit = 0;

fseek(disk\_file[0].file\_point, fat2\_position, 0);

fwrite(&unit, sizeof(int), 1, disk\_file[0].file\_point); //写入空

//下一block号写入上一block号的表项值

fseek(disk\_file[0].file\_point, fat1\_position, 0);

fwrite(&next\_unit, sizeof(int), 1, disk\_file[0].file\_point); //next\_unit：目标关键字的block号

printf(" 删除成功：\n==============\n");

//读表

read\_table();

}

}

10、fat\_alter\_data.c文件

//4 修改

#include "pch.h"

extern struct disk\_file\_information disk\_file[FILE\_AMOUNT]; //每个磁盘文件的信息//b+tree.c

extern struct fat\_block\_structure fat\_s; //b+tree.c

extern struct config\_information config\_inf; //config信息列表

struct data\_structure data\_str; //table数据表中数据格式

extern int query\_data(int key, int \*target\_unit); //query\_data.c

extern void read\_table(); //read\_table.c

void alter\_data() {

int unit, next\_unit; //表项值=1 表示该表项值有数据，下一表项值，

int new\_num; //新的block号

long fat\_position, block\_position;

int query\_result = 2; //查询关键字返回值：0无数据；1未找到关键字；2存在关键字

int target\_unit = 0 ;

int key; //数据关键字

char name[MAN\_NAME\_LENGTH]; //数据人名

//printf("=============== 4 修改数据 ===============\n");

//查找

//输入关键字

printf(" 请输入关键字：");

scanf("%d", &key);

query\_result = query\_data(key, &target\_unit);

if (query\_result == 2) //修改

{

printf(" 请输入姓名：");

scanf("%s", &name);

data\_str.key = key;

strcpy(data\_str.name, name);

//目标关键字的block号

fat\_position = ((target\_unit / fat\_s.fat\_value\_amount + 1) + target\_unit) \* sizeof(int);

fseek(disk\_file[0].file\_point, fat\_position, 0);

fread(&unit, sizeof(int), 1, disk\_file[0].file\_point); //next\_unit：目标关键字的block号

//修改数据位置，并写入

block\_position = unit \* config\_inf.block\_size \* 1024 + sizeof(int); //越过块号

fseek(disk\_file[0].file\_point, block\_position, 0);

fwrite(&data\_str, sizeof(data\_str), 1, disk\_file[0].file\_point);

printf(" 修改成功：\n==============\n");

//读表

read\_table();

}

}

11、fat\_search\_data.c文件

//5 查询关键字

#include "pch.h"

extern struct disk\_file\_information disk\_file[FILE\_AMOUNT]; //每个磁盘文件的信息//b+tree.c

extern struct fat\_block\_structure fat\_s; //b+tree.c

extern struct config\_information config\_inf; //config信息列表

struct data\_structure data\_str; //table数据表中数据格式

int query\_data(int key, int \*target\_unit) { //关键字，记录小于目标关键字的块号（小于取最大）

int unit, next\_unit; //表项值=1 表示该表项值有数据，下一表项值，

long fat\_position, block\_position;

int first\_unit = 0; //起始数据表项

//存储文件的表项值:0无数据；1（块号）未找到关键字（记录小于关键字中的最大关键字位于的块号）；2存在关键字

int read\_over\_flag = 0;

//printf("=============== 5 查询关键字 ===============\n");

fseek(disk\_file[0].file\_point, 0, 0); //定位文件起始

//fat块循环读

for (int i = 0; i < fat\_s.fat\_block\_amount && !read\_over\_flag; i++)

{

fseek(disk\_file[0].file\_point, sizeof(int), 1); //跳过每块的块号

//定位每个block中的起始数据表项

//0块里存的还有与fat对应的单元格

if (!i)

{

fseek(disk\_file[0].file\_point, fat\_s.fat\_block\_amount \* sizeof(int), 1);

first\_unit = fat\_s.fat\_block\_amount; //fat占用block个数的数值 = 第一个存数据的表项值标签对应的块号的数值；

}

//表项值循环找

unit = first\_unit;

for (int j = first\_unit; j < fat\_s.fat\_value\_amount && !read\_over\_flag; j++)

{

fread(&next\_unit, sizeof(int), 1, disk\_file[0].file\_point);

while (next\_unit != 0 && !read\_over\_flag) //非空闲block

{

//计算当下数据block的位置，读出数据

block\_position = unit \* config\_inf.block\_size \* 1024;

fseek(disk\_file[0].file\_point, block\_position + sizeof(int), 0);

fread(&data\_str, sizeof(data\_str), 1, disk\_file[0].file\_point);

if (data\_str.key < key)

{

//记录小于关键字中的最大关键字位于的块号

\*target\_unit = unit;

}

else if (data\_str.key == key)

{

//关键字存在

printf(" block\_num = %d ; fat\_position = %d ; block\_position = %dB \n", unit, fat\_position, block\_position);

//printf(" next\_block\_num = %d\n", next\_unit);

read\_over\_flag = 2;

}

//读取下一fat表项值

if (!read\_over\_flag)

{

if (next\_unit != 1)

{

unit = next\_unit; //保存块号，以读下一表项值

//计算next\_unit在fat中的位置

fat\_position = ((next\_unit / fat\_s.fat\_value\_amount + 1) + next\_unit) \* sizeof(int);

//定位

fseek(disk\_file[0].file\_point, fat\_position, 0);

//读取next\_unit在fat中的表项值

fread(&next\_unit, sizeof(int), 1, disk\_file[0].file\_point);

}

else

{

read\_over\_flag = 1;

}

}

}

}

}

//查询结果

if (read\_over\_flag == 0)

{

printf("无数据存储！\n");

return 0;

}

else if (read\_over\_flag == 1)

{

printf(" 关键字不存在！\n");

return 1;

}

else

{

//read\_over\_flag == 2

printf(" data\_str.key = %3d, data\_str.name = %s\n", data\_str.key, data\_str.name);

return 2;

}

}

12、fat\_read\_table.c文件

//6 查阅数据表

#include "pch.h"

extern struct disk\_file\_information disk\_file[FILE\_AMOUNT]; //每个磁盘文件的信息//b+tree.c

extern struct fat\_block\_structure fat\_s; //b+tree.c

extern struct config\_information config\_inf; //config信息列表

struct data\_structure data\_str; //table数据表中数据格式

void read\_table() {

int unit, next\_unit; //表项值=1 表示该表项值有数据，下一表项值，

int block\_num; //新的block号

long position;

int first\_unit = 0; //起始数据表项

int read\_over\_flag = 0; //存储文件的表项值是否是最后一个表项值, 结束读表

//printf("=============== 6 查阅数据表 ===============\n");

fseek(disk\_file[0].file\_point, 0, 0); //定位文件起始

//printf("1\n");

//fat块循环读

for (int i = 0; i < fat\_s.fat\_block\_amount && !read\_over\_flag; i++)

{

//printf("2\n");

fseek(disk\_file[0].file\_point, sizeof(int), 1); //跳过每块的块号

//定位每个block中的起始数据表项

//0块里存的还有与fat对应的单元格

if (!i)

{

fseek(disk\_file[0].file\_point, fat\_s.fat\_block\_amount \* sizeof(int), 1);

first\_unit = fat\_s.fat\_block\_amount; //fat占用block个数的数值 = 第一个存数据的表项值标签对应的块号的数值；

}

//表项值循环找

unit = first\_unit;

for (int j = first\_unit; j < fat\_s.fat\_value\_amount && !read\_over\_flag; j++)

{

//printf("3\n");

fread(&next\_unit, sizeof(int), 1, disk\_file[0].file\_point);

while (next\_unit != 0 && !read\_over\_flag) //非空闲block

{

//printf("4\n");

//计算当下数据block的位置

position = unit \* config\_inf.block\_size \* 1024;

fseek(disk\_file[0].file\_point, position + sizeof(int), 0); //越过块号

printf(" block\_num = %d ; block\_position = %d \n", unit, position);

fread(&data\_str, sizeof(data\_str), 1, disk\_file[0].file\_point);

printf(" data\_str.key = %3d, data\_str.name = %s\n", data\_str.key, data\_str.name);

if (next\_unit != 1)

{

unit = next\_unit; //保存块号，以读下一表项值

//计算next\_unit在fat中的位置

position = ((next\_unit / fat\_s.fat\_value\_amount + 1) + next\_unit) \* sizeof(int);

//定位

fseek(disk\_file[0].file\_point, position, 0);

//读取next\_unit在fat中的表项值

fread(&next\_unit, sizeof(int), 1, disk\_file[0].file\_point);

}

else

{

read\_over\_flag = 1;

}

}

}

}

//未找到空闲

if (!read\_over\_flag)

{

printf("无数据存储！\n");

}

}

13、btree\_insert\_data.c文件

// 2插入

#include "pch.h"

extern struct config\_information config\_inf; //config信息列表 b+tree.c

extern struct disk\_file\_information disk\_file[FILE\_AMOUNT]; //每个磁盘文件的信息 b+tree.c

extern struct save\_content s\_c[content\_num]; //目录块

extern int btree\_block\_apply();

void btree\_insert\_data(int table\_num, int key, int index\_num, int data\_num) {

//插入关键字

bnode node;

int unit;

int block\_index0, block\_index1;

long position;

int i, middle\_count; //关键字排序

printf("==============\n");

position = index\_num \* config\_inf.block\_size \* 1024;

fseek(disk\_file[0].file\_point, position + sizeof(int), 0); //定位索引文件起始

fread(&node, sizeof(bnode), 1, disk\_file[0].file\_point);

//插入关键字

//printf("关键字排序！\n");

for (i = 0; i < node.keynum; i++)

{

if (key <= node.key[i])

{

middle\_count = node.key[i];

node.key[i] = key;

key = middle\_count;

}

printf(" node.key[%d] = %d\n", i, node.key[i]);

}

node.key[node.keynum] = key; //第node.keynum个key存最大的数据 //key最小则key值变，否则不变

printf(" node.key[%d] = %d\n", node.keynum, node.key[node.keynum]);

node.keynum++;

printf("==============\n");

//写入以被插入新节点时读取正确

fseek(disk\_file[0].file\_point, position + sizeof(int), 0); //定位索引文件起始

fwrite(&node, sizeof(bnode), 1, disk\_file[0].file\_point);

if (node.keynum == m)

{

printf(" node.keynum = %d,", m);

if (index\_num != s\_c[table\_num].index\_block)

{

//disk\_file[0].file\_block\_amount首块索引块号

//非首块索引块，分裂申请"一块"空闲快

printf(" 非根节点分裂\n");

//申请空块,写入另一半关键字，从m / 2 + 1开始为中间值向上一节点插入

block\_index1 = btree\_insert\_node(0, 1, node.parent\_block\_num, node.key, node.key\_block, node.child\_block\_num);

//0左复制，1向上传关键字，孩子指针加一写入

//[m / 2]为第m / 2 + 1个数据，作为中间值向上一节点插入，（从0编号）

btree\_insert\_data(table\_num, node.key[m / 2], node.parent\_block\_num, node.key\_block[m / 2]); //索引上传

//清0

for (i = m / 2; i < m; i++)

{

node.key[i] = 0;

node.key\_block[i] = 0;

node.child\_block\_num[i] = 0;

}

node.child\_block\_num[m] = 0;

node.keynum = m / 2;

}

else

{

//首块索引块号，分裂申请"两块"空闲快，自身仍做首块，将自身数据分到两子块，使得树长一层

printf(" 根节点分裂\n");

//node.leaf = 0; //设为分支节点

//申请空块

// 左孩子

//block\_index0 = btree\_insert\_node(node.key[0], index\_num, node.key\_block[0]);

block\_index0 = btree\_insert\_node(1, 0, index\_num, node.key, node.key\_block, node.child\_block\_num);

// 右孩子

block\_index1 = btree\_insert\_node(0, 0, index\_num, node.key, node.key\_block, node.child\_block\_num);

//关键字调整

node.key[0] = node.key[m / 2];

node.key\_block[0] = node.key\_block[m / 2];

node.keynum = 1;

//清0

for (i = 1; i < m; i++)

{

node.key[i] = 0;

node.key\_block[i] = 0;

}

//指针调整

node.child\_block\_num[0] = block\_index0;

node.child\_block\_num[1] = block\_index1;

//清0

for (i = 2; i <= m; i++)

{

node.child\_block\_num[i] = 0;

}

}

}

//重新写入

fseek(disk\_file[0].file\_point, position + sizeof(int), 0); //定位索引文件起始

fwrite(&node, sizeof(bnode), 1, disk\_file[0].file\_point);

}

//int btree\_insert\_node(int key, int prior\_block\_index, int block\_data) {

int btree\_insert\_node(int left, int up\_data, int parent\_block\_num, int key[], int key\_block[], int child\_block\_num[]) {

bnode node, parent\_node, child\_node;

int child\_node\_key[m + 1] = { 0 }, middle\_count, middle\_index; //关键字排序

int block\_index, position;

int i; //循环变量

printf(" 插入新索引节点\n");

//节点初始化

if (left) //向左添加

{

for (i = 0; i < m; i++)

{

if (i < m / 2)

{

node.key[i] = key[i];

node.key\_block[i] = key\_block[i];

node.child\_block\_num[i] = child\_block\_num[i];

}

else {

node.key[i] = 0;

node.key\_block[i] = 0;

if (i == m / 2)

{

node.child\_block\_num[i] = child\_block\_num[i]; //尾指针复制

}

else

{

node.child\_block\_num[i] = 0;

}

}

}

}

else //向右添加

{

for (i = 0; i < m; i++)

{

if (i + m / 2 + 1 < m)

{

node.key[i] = key[m / 2 + 1 + i];

node.key\_block[i] = key\_block[m / 2 + 1 + i];

node.child\_block\_num[i] = child\_block\_num[m / 2 + 1 + i];

}

else {

node.key[i] = 0;

node.key\_block[i] = 0;

if (i + m / 2 + 1 == m)

{

node.child\_block\_num[i] = child\_block\_num[m / 2 + 1 + i]; //尾指针复制

}

else

{

node.child\_block\_num[i] = 0;

}

}

}

}

node.child\_block\_num[m] = 0; //数组从0开始计。0到m为m+1个数

node.keynum = m - m / 2 - 1;

//node.leaf = 1; //是叶子节点？

node.parent\_block\_num = parent\_block\_num;

//申请新节点

block\_index = btree\_block\_apply();

//插入数据

position = block\_index \* config\_inf.block\_size \* 1024;

fseek(disk\_file[0].file\_point, position + sizeof(int), 0); //定位索引文件起始

fwrite(&node, sizeof(bnode), 1, disk\_file[0].file\_point);

//输出节点

for (i = 0; i < node.keynum; i++)

{

printf(" node.key[%d] = %d ", i, node.key[i]);

printf(" node.key\_block[%d] = %d ", i, node.key\_block[i]);

printf(" node.child\_block\_num[%d] = %d \n", i, node.child\_block\_num[i]);

}

printf(" node.child\_block\_num[%d] = %d \n", node.keynum, node.child\_block\_num[node.keynum]);

printf(" keynum = %d, node.parent\_block\_num = %d\n", node.keynum, node.parent\_block\_num);

//双亲中写孩子节点，并排序

//读双亲节点

position = parent\_block\_num \* config\_inf.block\_size \* 1024;

fseek(disk\_file[0].file\_point, position + sizeof(int), 0); //定位索引文件起始

fread(&parent\_node, sizeof(bnode), 1, disk\_file[0].file\_point);

//读每个孩子指针中的一个关键字

for (i = 0; i < parent\_node.keynum; i++)

{

if (parent\_node.child\_block\_num[i] != 0)

{

position = parent\_node.child\_block\_num[i] \* config\_inf.block\_size \* 1024;

fseek(disk\_file[0].file\_point, position + sizeof(int), 0); //定位索引文件起始

fread(&child\_node, sizeof(bnode), 1, disk\_file[0].file\_point);

child\_node\_key[i] = child\_node.key[0];

}

else

{

child\_node\_key[i] = 0;

}

}

//根据关键字将孩子指针排序

middle\_index = block\_index;

for (int i = 0; i < parent\_node.keynum; i++)

{

if (key <= child\_node\_key[i])

{

//交换关键字

middle\_count = child\_node\_key[i];

child\_node\_key[i] = key;

key = middle\_count;

//交换双亲中的孩子指针

middle\_count = parent\_node.child\_block\_num[i];

parent\_node.child\_block\_num[i] = middle\_index;

middle\_index = middle\_count;

}

else if (child\_node\_key[i] == 0)

{

//交换关键字

middle\_count = child\_node\_key[i];

child\_node\_key[i] = key;

key = middle\_count;

//交换双亲中的孩子指针

middle\_count = parent\_node.child\_block\_num[i];

parent\_node.child\_block\_num[i] = middle\_index;

middle\_index = middle\_count;

break;

}

}

//第node.keynum个num存最大的数据 //middle\_index最小则middle\_index值变，否则不变

if (up\_data) {

parent\_node.child\_block\_num[parent\_node.keynum + 1] = middle\_index;

}

else

{

parent\_node.child\_block\_num[parent\_node.keynum] = middle\_index;

}

//输出节点

printf(" 双亲节点！\n");

for (i = 0; i < parent\_node.keynum; i++)

{

printf(" node.key[%d] = %d ", i, parent\_node.key[i]);

printf(" node.key\_block[%d] = %d ", i, parent\_node.key\_block[i]);

printf(" node.child\_block\_num[%d] = %d \n", i, parent\_node.child\_block\_num[i]);

}

printf(" node.child\_block\_num[%d] = %d \n", parent\_node.keynum, parent\_node.child\_block\_num[parent\_node.keynum]);

printf(" keynum = %d, node.parent\_block\_num = %d\n", parent\_node.keynum, parent\_node.parent\_block\_num);

//写双亲节点

position = parent\_block\_num \* config\_inf.block\_size \* 1024;

fseek(disk\_file[0].file\_point, position + sizeof(int), 0); //定位索引文件起始

fwrite(&parent\_node, sizeof(bnode), 1, disk\_file[0].file\_point);

printf(" 插入结束！\n");

printf("==============\n");

return block\_index;

}

14、btree\_read\_table.c文件

//6 查阅数据表

#include "pch.h"

extern struct disk\_file\_information disk\_file[FILE\_AMOUNT]; //每个磁盘文件的信息//b+tree.c

extern struct config\_information config\_inf; //config信息列表//b+tree.c

struct data\_structure data\_str; //table数据表中数据格式

void btree\_read\_table(int index\_block) {

long position, data\_position;

bnode node;

position = index\_block \* config\_inf.block\_size \* 1024 + sizeof(int); //越过块号

fseek(disk\_file[0].file\_point, position, 0); //定位起始

fread(&node, sizeof(bnode), 1, disk\_file[0].file\_point);

printf(" index\_block：%d\n", index\_block);

if (node.keynum > 0)

{

if (node.child\_block\_num[0] == 0) //叶子节点

{

for (int i = 0; i < node.keynum; i++)

{

if (node.key[i] != 0)

{

printf(" key[%d] = %d\n", i, node.key[i]);

}

}

}

else //非叶子节点

{

for (int j = 0; j < node.keynum + 1; j++)

{

btree\_read\_table(node.child\_block\_num[j]);

if (j < node.keynum && node.key[j] != 0)

{

printf(" key[%d] = %d\n", j, data\_str.key);

}

}

}

}

else

{

printf(" 无数据存储！\n");

return 0;

}

}

15、btree\_alter\_data.c文件

//4 修改

#include "pch.h"

extern struct disk\_file\_information disk\_file[FILE\_AMOUNT]; //每个磁盘文件的信息//b+tree.c

extern struct config\_information config\_inf; //config信息列表

extern struct save\_content s\_c[content\_num]; //目录块

struct data\_structure data\_str; //table数据表中数据格式

extern int btree\_search\_data(int key, int block\_index, Status \*find\_flag); //btree\_search\_data.c

extern void content();

void btree\_alter\_data(int table\_num, int key) {

int index\_block, data\_block;

bnode node;

long position;

Status flag = false, \*find\_flag = &flag;

char name[MAN\_NAME\_LENGTH]; //数据人名

int i;

//printf("=============== 4 修改数据 ===============\n");

//查找

index\_block = btree\_search\_data(key, s\_c[table\_num].index\_block, find\_flag);

if (flag == true)

{

printf(" 请输入姓名：");

scanf("%s", &name);

data\_str.key = key;

strcpy(data\_str.name, name);

//获取数据位置

position = index\_block \* config\_inf.block\_size \* 1024 + sizeof(int); //越过块号

fseek(disk\_file[0].file\_point, position, 0);

fread(&node, sizeof(node), 1, disk\_file[0].file\_point);

for (i = 0; i < node.keynum; i++)

{

if (node.key[i] == key)

{

data\_block = node.key\_block[i];

break;

}

}

//数据的位置，并写入

position = data\_block \* config\_inf.block\_size \* 1024 + sizeof(int); //越过块号

fseek(disk\_file[0].file\_point, position, 0);

fwrite(&data\_str, sizeof(data\_str), 1, disk\_file[0].file\_point);

printf("写入\n");

fseek(disk\_file[0].file\_point, position, 0);

fread(&data\_str, sizeof(data\_str), 1, disk\_file[0].file\_point);

printf("重读\n");

printf("data\_str.key = %d, data\_str.name = %s\n", data\_str.key, data\_str.name);

printf(" 修改成功：\n==============\n");

}

else

{

printf(" 关键字不存在！");

}

}

16、btree\_delete\_data.c文件

// 3 删除

#include "pch.h"

extern struct disk\_file\_information disk\_file[FILE\_AMOUNT]; //每个磁盘文件的信息//b+tree.c

extern struct config\_information config\_inf; //config信息列表

extern struct fat\_block\_structure fat\_s; //b+tree.c

int btree\_delete\_data(int key, int index\_block, Status \*flag) {

bnode node, parent\_node, bro\_node, child\_node;

int unit = 0;

int data\_block;

long position, fat\_position, position2;

int i, bro\_i, j, middle\_count, key\_list[m];

//printf("=============== 5 删除数据 ===============\n");

//（1）数据块置空

position = index\_block \* config\_inf.block\_size \* 1024 + sizeof(int); //越过块号

fseek(disk\_file[0].file\_point, position, 0);

fread(&node, sizeof(node), 1, disk\_file[0].file\_point);

for (i = 0; i < node.keynum; i++)

{

if (node.key[i] == key)

{

data\_block = node.key\_block[i];

break;

}

}

//目标关键字的fat\_block号

fat\_position = ((data\_block / fat\_s.fat\_value\_amount + 1) + index\_block) \* sizeof(int);

fseek(disk\_file[0].file\_point, fat\_position, 0);

fwrite(&unit, sizeof(int), 1, disk\_file[0].file\_point); //unit：目标关键字的block号

if (node.child\_block\_num[i] != 0)

{

//非叶子节点

// 数据块置空

printf("非叶子节点，读关键字左孩子结点\n");

position = node.child\_block\_num[i] \* config\_inf.block\_size \* 1024 + sizeof(int); //越过块号

fseek(disk\_file[0].file\_point, position, 0);

fread(&child\_node, sizeof(node), 1, disk\_file[0].file\_point);

node.key[i] = child\_node.key[child\_node.keynum - 1];

node.key\_block[i] = child\_node.key\_block[child\_node.keynum - 1];

btree\_delete\_data(child\_node.key[child\_node.keynum - 1], node.child\_block\_num[i], flag);

}

else

{

//叶子节点

printf("叶子节点\n");

//node.keynum -= 1;

node.key[i] = 0;

node.key\_block[i] = 0;

}

//（2）该结点key个数大于等于Math.ceil(m/2)-1，结束删除操作，否则执行第3步。

if (node.keynum >= m / 2 - 1)

{

position = index\_block \* config\_inf.block\_size \* 1024 + sizeof(int); //越过块号

fseek(disk\_file[0].file\_point, position, 0);

fwrite(&node, sizeof(node), 1, disk\_file[0].file\_point);

printf("删除成功！\n");

return ok;

}

else //（3）

{

position = node.parent\_block\_num \* config\_inf.block\_size \* 1024 + sizeof(int); //越过块号

fseek(disk\_file[0].file\_point, position, 0);

fread(&parent\_node, sizeof(node), 1, disk\_file[0].file\_point);

//定位该关键字，寻找前一兄弟

for (i = 0; i < parent\_node.keynum; i++)

{

if (parent\_node.child\_block\_num[i] == index\_block)

{

break;

}

}

if (i - 1 >= 0) //有前兄弟

{

//i -= 1前一兄弟

position2 = parent\_node.child\_block\_num[i - 1] \* config\_inf.block\_size \* 1024 + sizeof(int); //越过块号

fseek(disk\_file[0].file\_point, position2, 0);

fread(&bro\_node, sizeof(node), 1, disk\_file[0].file\_point);

if (bro\_node.keynum > m / 2 - 1) //大于, 能借

{

//取得双亲中i-1，此值是该节点中最小值，重排序

key = parent\_node.key[i - 1];

for (j = 0; j < node.keynum; j++)

{

if (key <= node.key[j])

{

middle\_count = node.key[j];

node.key[j] = key;

key = middle\_count;

}

printf(" node.key[%d] = %d\n", j, node.key[j]);

}

node.key[node.keynum] = key; //第node.keynum个key存最大的数据 //key最小则key值变，否则不变

printf(" node.key[%d] = %d\n", node.keynum, node.key[node.keynum]);

//node.keynum++;

printf(" 双亲中i-1值下移\n");

printf("==============\n");

parent\_node.key[i - 1] = bro\_node.key[bro\_node.keynum - 1]; //兄弟最大上到双亲i - 1

btree\_delete\_data(bro\_node.key[bro\_node.keynum - 1], parent\_node.child\_block\_num[i - 1], flag);

return ok;

}

else

{

//节点结合

//暂存该节点值

for (j = 0; j < node.keynum; j++)

{

key\_list[j] = node.key[j];

}

//重新赋值，兄弟加入

for (j = 0; j < bro\_node.keynum; j++)

{

node.key[j] = bro\_node.key[j];

}

//重新赋值，取得双亲中i-1加入

node.key[j] = parent\_node.key[i - 1];

//暂存的该节点值重新写入

for (j = 0; j < node.keynum; j++)

{

node.key[j] = key\_list[j];

}

//删除兄弟指针及其节点

//目标关键字的fat\_block号

fat\_position = ((parent\_node.child\_block\_num[i - 1] / fat\_s.fat\_value\_amount + 1) + index\_block) \* sizeof(int);

fseek(disk\_file[0].file\_point, fat\_position, 0);

fwrite(&unit, sizeof(int), 1, disk\_file[0].file\_point); //写空

//双亲变化的位置后的关键字，孩子指针前移，原最后一个数据写空

for (j = i; j < node.keynum; j++)

{

//最后一位一定是有的，还是0

parent\_node.key[j - 1] = parent\_node.key[j];

parent\_node.key\_block[j - 1] = parent\_node.key\_block[j];

parent\_node.child\_block\_num[j - 1] = parent\_node.child\_block\_num[j];

}

}

}

}

}

17、btree\_search\_data.c文件

//5 查询

#include "pch.h"

extern struct disk\_file\_information disk\_file[FILE\_AMOUNT]; //每个磁盘文件的信息//b+tree.c

extern struct config\_information config\_inf; //config信息列表//b+tree.c

int btree\_search\_data(int key, int block\_index, Status \*find\_flag) {

int unit;

long position;

bnode node;

position = block\_index \* config\_inf.block\_size \* 1024 + sizeof(int); //越过块号

fseek(disk\_file[0].file\_point, position, 0); //定位起始

fread(&node, sizeof(bnode), 1, disk\_file[0].file\_point);

printf(" 当前查询block\_index：%d\n", block\_index);

//初始化的根节点，还未存储数据

if (node.keynum == 0)

{

return block\_index; //返回当前块号，查到其中

}

//已存储数据

printf(" 已存储数据node.keynum = %d\n", node.keynum);

for (int i = 0; i < node.keynum; i++)

{

printf(" node.key[%d] = %d\n", i, node.key[i]);

if (key < node.key[i])

{

if (node.child\_block\_num[i] != 0)

{

return btree\_search\_data(key, node.child\_block\_num[i], find\_flag);

}

else

{

\*find\_flag = false;

return block\_index;

}

}

else if (key == node.key[i])

{

printf(" 关键字存储于block\_index：%d\n", block\_index);

\*find\_flag = true;

return block\_index;

}

else if (key > node.key[i] && i == node.keynum - 1)

{

if (node.child\_block\_num[i] != 0)

{

return btree\_search\_data(key, node.child\_block\_num[node.keynum], find\_flag);

}

else

{

\*find\_flag = false;

return block\_index;

}

}

}

\*find\_flag = false;

printf(" 无存储！\n");

return 0;

}