**题目：动态散列索引的实现**

**中文摘要：**

有很多涉及散列表的数据结构可用作索引，作为主存数据结构的散列表都有一个散列函数，它以查找键作为参数，计算出一个桶的数目，动态散列技术允许散列函数动态改变，通过桶的合并和分解实现数据库的增大或缩小的需求，这样既继承了散列高效查找效率又保持了良好的空间压缩率。

**关键词：**位运算，二叉排序树，桶 ，散列表 ，搜索码 ，二进制文件存储，

块存储与解析

**一、前言：**

通过动态散列索引的实现，了解索引结构及其原理。通过学习，设计出动态散列索引的实现的设计思想和实现方案，利用语言工具实现程序并进行小数量的模拟和大数量的验证。

**二、平台：**Windows操作系统，vs2012平台

**三、设计思想：**

动态散列技术允许散列函数动态改变，通过桶的合并和分解实现数据库的增大或缩小的需求，这样既继承了散列高效查找效率又保持了良好的空间压缩率。

动态散列是逐步扩充散列值的位数来构造索引，它通过位比较来实现散列值的定位，这种比较方式计算机通过几个CPU机器指令即可实现，故它的效率很高。

首先，选择一个具有均匀性和随机性特性的散列函数h，此散列函数产生值的范围相对较大，是b位二进制整数N。接着初始化一个空桶，且使用散列位数为零，将记录逐条插入桶中；当桶已经满时，

，新建一个新的桶，与原来的桶构成一组分别代表0和1的桶，作为分裂结果的存储桶，利用N的一位作为散列位数，将原有的桶里的记录，按照第一位为0或1，重新分配到新的桶中，然后继续持续的插入待建索引的记录；直到其中某一个桶满时，重复上述操作，新建一个新桶，与原来的桶构成一组分别代表0和1的桶，作为下一位分裂结果的存储桶，利用N的下一位作为散列位数，将原有的桶里的记录，按照所选下一位为0或1，重新分配到新的桶中，然后继续持续的插入待建索引的记录；当所利用的散列位数等于N的位数，即有索引值相同总数超过桶容量时，通过建立链表的方式，新建一个新桶，将数据存于新桶中即可，如果持续超出，则扩充链表长度即可。相反，当数据量减小的时候，如果具有相同位数且只在最后一位不同的两个桶可以合并为一个桶时，可直接合并两个桶，并将其中一个多余的桶删除，实现动态散列的功能。查找时，确定含有搜索码值ＫL的桶的位置，系统取得h(ＫL)的第i个高字节，然后为这个位串查找对应的表项，再根据表项中的指针来得到桶的位置，桶内使用顺序扫描即可。桶的结构必须包括必须的用于存储一个桶所能容纳的记录的空间，此外，还需有当前已放入的记录数与可容纳记录数的存储空间，索引所利用的N位数，及其具体值。

**四、实现方案：**

选择一个具有均匀性和随机性特性的散列函数h，此散列函数产生值的范围相对较大，是b位二进制整数N。，并根据该函数结果的特点建立一棵二叉排序树，作为桶位置的指向，保存的是对应桶在二进制文件中的块数目。

利用一个二叉排序树，0代表左子树，1代表右子树，树的节点有四个指针，分别为左右子树对应的子节点指针，或者为叶子节点时指向的桶地址指针，以及保存父节点的指针。起初的时候树只有一个根节点，并同时建立一个桶，根节点指向该桶，且左右子树为空，不断往桶中插入记录。桶的结构必须包括用于存储一个桶所能容纳的记录的空间，可新建一个记录结构体，然后在桶中建立数组代表一个桶，此外，还需有当前已放入的记录数与可容纳记录数的存储空间，索引所利用的N位数（这三者可用整形数表示），及其具体值，该桶所对应的树节点地址，每个桶的大小为内存块的大小。

如果根节点为空的时候，通过扫描Index，如果扫描到的节点不为空时，若所指向的块为-1，则继续扫描，则到找到对应的二进制块，往其中插入数据；若扫描到的节点为空，则直接新建一个块，且保存位置于二叉树中，并往新建块中插入新数据。

当桶装满的时候，通过二叉树，找到该树节点，分裂该树叶子节点，分别为该节点的左右子树新建叶子节点，并在叶子节点中插入新桶位置，左右子树均为空，然后分裂桶，将原桶中的数据根据具体比较位的不同分配到新桶中，持续这个过程，直到将所有记录分配到对应的动态散列桶中，如果桶中的数据都集中于一边，则将另一个块删除，同时删除该树节点，并不断递归，直到分配结果不会出现都集中到一个桶的情况。但当所利用的散列位数等于N的位数，即有索引值相同总数超过桶容量时，通过建立链表的方式，新建一个新桶，将数据存于新桶中即可，如果持续超出，则扩充链表长度即可。

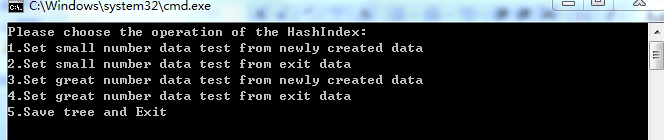
查询的时候通过扫描利用散列函数h计算出来的数值N，通过扫描N的位，对比二叉排序树，但扫描到某一个位为叶子节点时，即可通过该节点找到桶的位置，桶内顺序查找即可。

删除时，找到所在桶，删除记录，并通过桶中所记录二叉树中节点位置，找到父节点，比较父节点左右子树能否合并为一个桶，有则合并两桶，并删除左右子树，通过父节点指向新合成的桶。

**（在此具体实现方案中，考虑进一步减少查询时间，避免了使用数组结构指向桶的方式找到桶，使用二叉排序树可更快的实现查找，而且利用树本身左右子树的特点表示0和1，可减少增加新的扫描位时更新数组大小及其位数和指向的问题，进一步提高效率。）**

**五、程序测试：**

主界面：选择数据的量及其生成类型

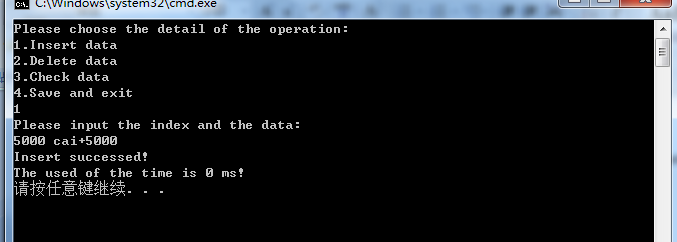


测试第一种生成数据方式

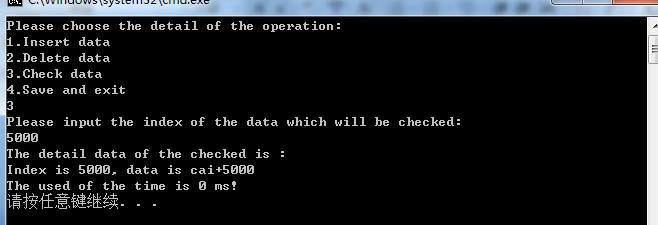
主界面选择1：用随机数创建一个500的数据量

次界面：选择操作类型

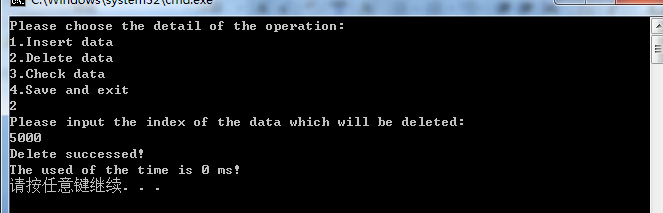
选择1：插入一个数据项5000，用时为0ms



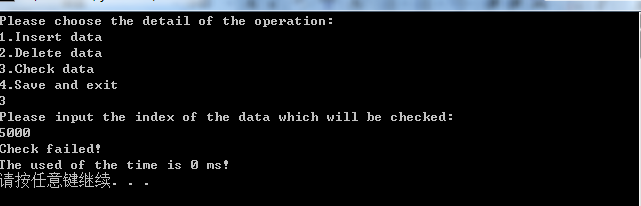
选择3：查询插入的项5000，用时为0ms



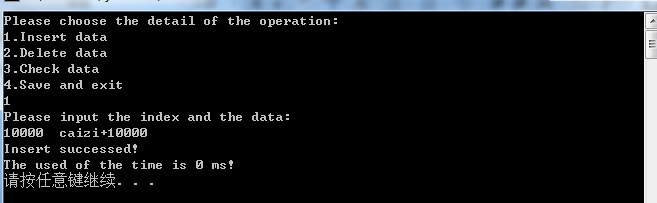
选择2：删除该插入项5000，用时为0ms



再查询，查询失败，则数据项已删除



选择1：再插入一个数据项10000，用于检测主界面中第二种方式，看能否查到



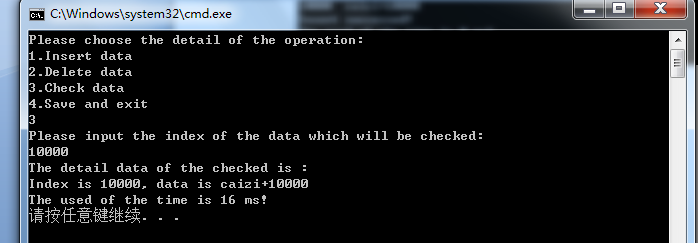
选择4，保存退出

主界面选择5，保存退出

测试第二种创建数据方式：

重新运行程序，主界面选择2：读入已有小数据量数据的二叉排序树

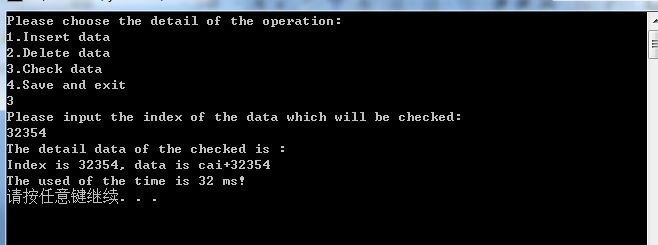
次界面选择3:查询上次插入数据10000,用时16ms



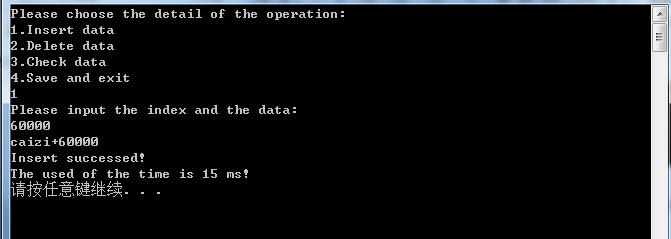
测试第三种创建数据方式：

重新运行程序，主界面选择3：用连续50000个数创建大量数据（此时需要较长创建时间）

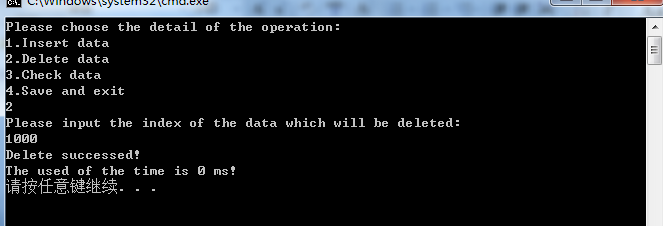
次界面选择3：查询数据32354，用时为32ms



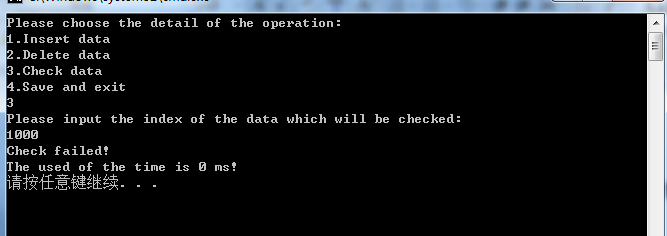
次界面选择1：插入数据60000，用时为15ms



选择2：删除1000，用时为0ms



再查询，已不存在：



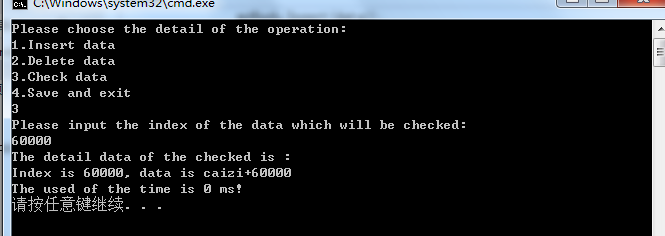
选择4，保存退出

主界面选择5，保存退出

测试第三种创建数据方式：

重新运行程序，主界面选择4，读入已有的大数据量测试数据

选择3：查询上次插入数据60000，查询成功，用时为0ms



**经测试，小量数据及大量数据的测试结果均正确，且可以正确的保存为外存中，以便下次使用。**

**六、总结：**

**优点：**

少量数据和大量数据的测试对于插入、删除、查看都可以做到正确，且时间多数在几毫秒之间就可以实现，相对快速快速，满足动态hash索引的基本要求。

**不足：**

用于索引的二叉排序树没有实现保存于外存、动态读入的方式，所以当测试50000量数据量的时候，采用随机数产生的方法，由于数据过于分散，可能出现二叉排序树会过大，导致溢出，所以本例只能采用递增数据实现此测试。

**七、参考文献：**

1. 《数据库系统实现》 第2版/(美) 加西亚-莫利纳 等著 杨冬青 等译 机械工业出版社 2010.5

**附录：**

**程序代码：**

#include<string>

#include<cstdio>

#include<fstream>

#include<stack>

#include<cstring>

#include<time.h>

#include<iostream>

using namespace std;

#define num 2147483648;

int blockNum = 1;

int BSzie=1;

struct myData

{

unsigned Index;

char Data[124];

};

struct myHashBlock

{

int BlockIndex;

unsigned IndexNum;

int DataNUM;

int CheckNUM;

myData \*DataArray;

int nextBlockNum;

};

struct myHashNode

{

int BlockNum;

myHashNode \*Parent,\*rChild,\*lChild;

};

class myHashIndex

{

private:

myHashNode \*root;

int bufferSize;

public:

char TreeFile[10],DataFile[10];

//char \*buffer;

myHashIndex(int bSize)

{

root=NULL;

bufferSize = bSize;

strcpy(TreeFile ,"sTree.txt");

strcpy(DataFile , "sData.dat");

}

~myHashIndex()

{

//saveTree();

//delete []buffer;

if(root!=NULL)

deleteTree(root);

}

void deleteTree(myHashNode \*current)

{

if(current->lChild!=NULL)

deleteTree(current->lChild);

if(current->rChild!=NULL)

deleteTree(current->rChild);

delete current;

}

//保存二叉排序树

bool saveTree()

{

stack<myHashNode\*>s; //开一个栈用于保存右节点地址

myHashNode \*p=root;

s.push(NULL);

ofstream out(TreeFile,ios::out);

if(!out)

{

cout<<"Output file open failed";

return false;

}

while(p!=NULL||!s.empty())

{

while(p!=NULL)

{

out<<p->BlockNum<<" ";

s.push(p);

p=p->lChild;

}

out<<"#";

if(!s.empty())

{

p=s.top();s.pop();

p=p->rChild;

while(p==NULL&&!s.empty())

{

out<<"#";

p=s.top();s.pop();

if(p!=NULL)

p=p->rChild;

}

}

}

out<<"#\n";

out.close();

return true;

}

//读取保存于外存的二叉排序树

bool readTree()

{

ifstream inData(TreeFile, ios::in);//打开文件.txt

if (!inData)//判断打开文件是否成功

return false;

readTree(root,NULL,inData);

inData.close();

return true;

}

bool readTree(myHashNode \*&current,myHashNode \*parent,ifstream &inData )

{

char w;

int bNum;

if(!inData.eof())

{

inData>>w;

if(w!='#')

{

inData.seekg(-1,ios::cur);

inData>>bNum;

current = new myHashNode;

current->Parent = parent;

current->BlockNum = bNum;

readTree(current->lChild,current,inData);

readTree(current->rChild,current,inData);

}

else

{

current = NULL;

}

}

return true;

}

//读入块

myHashBlock\*& readBlock(int bNUM)

{

char buffer[1024\*2];

char \*p = buffer;

//新建桶

myHashBlock \*pB = NULL;

pB = new myHashBlock;

pB->BlockIndex = 0;

pB->IndexNum = 0;

pB->CheckNUM = 0;

pB->DataNUM = bufferSize;

pB->DataArray = new myData[bufferSize];

pB->nextBlockNum = -1;

//读入对应块

ifstream in(DataFile,ios::in||ios::binary);

if(!in)

{

cerr<<"Input file open failed";

return pB;

}

in.seekg((bNUM-1)\*1024\*BSzie,ios::beg);

in.read(buffer,1024\*BSzie);

in.close();

//解析二进制块

memcpy(&(pB->BlockIndex),p,4);

memcpy(&(pB->IndexNum),p+4,4);

memcpy(&(pB->CheckNUM),p+8,4);

memcpy(&(pB->DataNUM),p+12,4);

memcpy(&(pB->nextBlockNum),p+16,4);

p = p+20;

int i,max = pB ->CheckNUM;

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

{

memcpy(&(pB->DataArray[i].Index),p,4);

memcpy((pB->DataArray[i].Data),p+4,124);

p = p + 128;

}

//返回解析完数据

return pB;

}

//写出块

bool writeBlock(int bNUM,myHashBlock \*pB)

{

char buffer[1024\*2];

char \*P;

//将数据以二进制形式写入缓冲区

P = buffer;

memcpy(P,&(pB->BlockIndex),4);

memcpy(P+4,&(pB->IndexNum),4);

memcpy(P+8,&(pB->CheckNUM),4);

memcpy(P+12,&(pB->DataNUM),4);

memcpy(P+16,&(pB->nextBlockNum),4);

P = P+20;

int i,max = pB ->CheckNUM;

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

{

memcpy(P,&(pB->DataArray[i].Index),4);

memcpy(P+4,(pB->DataArray[i].Data),124);

P = P + 128;

}

//写出块

ofstream out(DataFile,ios::out||ios::binary||ios::ate);

if(!out)

{

cout<<"Output file open failed";

return false;

}

out.seekp((bNUM-1)\*1024\*BSzie,ios::beg);

out.write(buffer,1024\*BSzie);

out.close();

return true;

}

bool Insert(myData data)

{

myHashBlock \*pB;

myHashNode \*current;

int i;

//判断跟节点是否为空，新建第一个Hash块

if(root==0)

{

pB = new myHashBlock;

pB->CheckNUM = 0;

pB->DataNUM = bufferSize;

pB->DataArray = new myData[bufferSize];

pB->nextBlockNum = -1;

root = new myHashNode;

pB->IndexNum = 0;

pB->BlockIndex = 0;

root->BlockNum = blockNum;

blockNum++;

root->lChild=root->rChild=root->Parent=NULL;

pB->DataArray[pB->CheckNUM]=data;

pB->CheckNUM++;

writeBlock(root->BlockNum,pB);

delete []pB->DataArray;

delete pB;

return true;

}

//当根节点指向Hash块时，在根节点所指向的块中填入数据知道满，则新建新左右节点

unsigned index = num;

if(root->BlockNum >=0 )

{

pB =readBlock(root->BlockNum);

//未满，持续插入

if(pB->CheckNUM<(pB->DataNUM))

{

pB->DataArray[pB->CheckNUM]=data;

pB->CheckNUM++;

writeBlock(root->BlockNum,pB);

}

else

{

//满则调用分拆函数，新建其新的左右结点

bool f1 = setNewNode(root,data,index);

delete []pB->DataArray;

delete pB;

return f1;

}

delete []pB->DataArray;

delete pB;

return true;

}

//当根节点不指向Hash时，扫描Data的index，搜索root所指向的树，找到Data的归属Hash块

current = root;

for(i=0;i<32;i++,index=index>>1)

{

if((data.Index & index)==0)

if(current->lChild!=NULL)

current = current->lChild;

else

{

//为空的时候新建块

pB = new myHashBlock;

pB->CheckNUM = 0;

pB->DataNUM = bufferSize;

pB->DataArray = new myData[bufferSize];

pB->nextBlockNum = -1;

current->lChild = new myHashNode;

pB->IndexNum = 0;

pB->BlockIndex = 0;

current->lChild->BlockNum = blockNum;

blockNum++;

current->lChild->lChild=current->lChild->rChild=NULL;

current->lChild->Parent = current;

current = current->lChild;

break;

}

else

if(current->rChild!=NULL)

current = current->rChild;

else

{

//为空的时候新建块

pB = new myHashBlock;

pB->CheckNUM = 0;

pB->DataNUM = bufferSize;

pB->DataArray = new myData[bufferSize];

pB->nextBlockNum = -1;

current->rChild = new myHashNode;

pB->IndexNum = 0;

pB->BlockIndex = 0;

current->rChild->BlockNum = blockNum;

blockNum++;

current->rChild->lChild=current->rChild->rChild=NULL;

current->rChild->Parent = current;

current = current->rChild;

break;

}

if(current->BlockNum>=0)

{

//不为空的时候读入块

pB = readBlock(current->BlockNum);

break;

}

}

if(pB->CheckNUM<(pB->DataNUM))

{

pB->DataArray[pB->CheckNUM]=data;

pB->CheckNUM++;

writeBlock(current->BlockNum,pB);

}

else

{

//满则调用分拆函数，新建其新的左右结点

index = index>>1;

bool f2 = setNewNode(current,data,index);

delete []pB->DataArray;

delete pB;

return f2;

}

delete []pB->DataArray;

delete pB;

return true;

}

bool DeleteAll(unsigned Index)

{

myHashNode \*current;

myHashBlock \*pB,\*p ;

unsigned index = num;

int i;

//判断根节点是否为空

if(root == NULL)

return false;

bool Flag1 =false;

bool Flag2 = false;

//当根节点指向Hash块时，扫描该块，得到是否有删除数据

if(root->BlockNum >=0 )

{

pB = readBlock( root->BlockNum);

for(i=0;i<=pB->CheckNUM;i++)

if(pB->DataArray[i].Index == Index)

{

deleteData(i,pB);

writeBlock(root->BlockNum,pB);

Flag1 = true;

//return true;

}

if(Flag1)

{

delete []pB->DataArray;

delete pB;

return true;

}

else

{

delete []pB->DataArray;

delete pB;

return false;

}

}

//当根节点不指向Hash时，扫描Data的index，搜索root所指向的树，找到Data的归属Hash块

current = root;

for(i=0;i<32;i++,index=index>>1)

{

if((Index & index)==0)

if(current->lChild!=NULL)

current = current->lChild;

else

return false;

else

if(current->rChild!=NULL)

current = current->rChild;

else

return false;

if(current->BlockNum>=0)

break;

}

//持续扫描所得块，及其后续块，删除所有

pB = readBlock(current->BlockNum) ;

if(pB->nextBlockNum != -1)

Flag2 = true;

while (pB != NULL)

{

for(i=0;i<=pB->CheckNUM;i++)

if(pB->DataArray[i].Index == Index)

{

deleteData(i,pB);

writeBlock(current->BlockNum,pB);

if(pB->CheckNUM<0)

current->BlockNum = pB->nextBlockNum;

if(!Flag2)

checkCombine(current->Parent);

delete []pB->DataArray;

delete pB;

return true;

}

if(pB->nextBlockNum!=-1)

{

p = pB;

pB=readBlock(pB->nextBlockNum);

delete []p->DataArray;

delete p;

}

else

{

delete []pB->DataArray;

delete pB;

pB = NULL;

}

}

}

myData\* Check(unsigned Index)

{

myHashNode \*current;

myData \*pD = new myData;

myHashBlock \*pB ,\*p;

unsigned index = num;

int i;

//判断根节点是否为空

if(root == NULL)

return NULL;

//当根节点指向Hash块时，扫描该块，得到是否有所检查数据

if(root->BlockNum >=0 )

{

pB =readBlock( root->BlockNum );

for(i=0;i<=pB->CheckNUM;i++)

if(pB->DataArray[i].Index == Index)

{

\*pD = pB->DataArray[i];

delete []pB->DataArray;

delete pB;

return pD;

}

delete []pB->DataArray;

delete pB;

return NULL;

}

//当根节点不指向Hash时，扫描Data的index，搜索root所指向的树，找到Data的归属Hash块

current = root;

for(i=0;i<32;i++,index=index>>1)

{

if((Index & index)==0)

if(current->lChild!=NULL)

current = current->lChild;

else

return NULL;

else

if(current->rChild!=NULL)

current = current->rChild;

else

return NULL;

if(current->BlockNum>=0)

break;

}

//扫描所得块，判断index，如果相同，则返回第一个找到值

pB = readBlock(current->BlockNum) ;

while (pB != NULL)

{

for(i=0;i<=pB->CheckNUM;i++)

if(pB->DataArray[i].Index == Index)

{

\*pD = pB->DataArray[i];

delete []pB->DataArray;

delete pB;

return pD;

}

if(pB->nextBlockNum!=-1)

{

p = pB;

pB=readBlock(pB->nextBlockNum);

delete []p->DataArray;

delete p;

}

else

{

delete []pB->DataArray;

delete pB;

pB = NULL;

}

}

return NULL;

}

bool setNewNode(myHashNode \*&node,myData data,unsigned index)

{

myHashBlock \*pL,\*pR,\*p;

if(index!=0)

{

//创建新子树Hash块

pL = new myHashBlock;

pL->CheckNUM = 0;

pL->DataNUM = bufferSize;

pL->DataArray = new myData[bufferSize];

pL->nextBlockNum = -1;

pR = new myHashBlock;

pR->CheckNUM = 0;

pR->DataNUM = bufferSize;

pR->DataArray = new myData[bufferSize]();

pR->nextBlockNum = -1;

//创建新Node指向性Hash块，并将父节点的Hash块指针置为NULL

node->lChild = new myHashNode;

node->rChild = new myHashNode;

node->lChild->BlockNum = node->BlockNum;

node->rChild->BlockNum = blockNum;

blockNum++;

p = readBlock( node->BlockNum);

node->BlockNum = -1;

node->lChild->Parent=node->rChild->Parent=node;

node->lChild->lChild=node->rChild->lChild=node->lChild->rChild=node->rChild->rChild=NULL;

pL->IndexNum = p->IndexNum+1;

pR->IndexNum = p->IndexNum+1;

pL->BlockIndex = pR->BlockIndex = 0;

//重新分配原Hash中数据，即pR指向的块

myData \*pD;

for(int i=0;i<p->DataNUM;i++)

{

pD=&p->DataArray[i];

//cout<<pD->Index<<" ";

if((pD->Index & index)==0)

{

pL->DataArray[pL->CheckNUM] = \*pD;

pL->CheckNUM++;

if(pL->BlockIndex==0)

pL->BlockIndex = pD->Index;

}

else

{

pR->DataArray[pR->CheckNUM] = \*pD;

pR->CheckNUM++;

if(pR->BlockIndex==0)

pR->BlockIndex = pD->Index;

}

}

delete []p->DataArray;

delete p;

//为新数据在其所属的Hash块中填入，如果偏向某一边，则递归实现，直到最后

if((data.Index & index)==0)

{

if(pL->CheckNUM<pL->DataNUM)

{

pL->DataArray[pL->CheckNUM++] = data;

}

else

{

index = index>>1;

delete []pR->DataArray;

delete pR;

blockNum--;

delete node->rChild;

node->rChild = NULL;

return setNewNode(node->lChild,data,index);

}

}

else

{

if(pR->CheckNUM<pR->DataNUM)

{

pR->DataArray[pR->CheckNUM++] = data;

}

else

{

index = index>>1;

delete []pL->DataArray;

delete pL;

node->rChild->BlockNum = node->lChild->BlockNum;

blockNum--;

delete node->lChild;

node->lChild = NULL;

return setNewNode(node->rChild,data,index);

}

}

writeBlock(node->lChild->BlockNum,pL);

writeBlock(node->rChild->BlockNum,pR);

delete []pL->DataArray;

delete pL;

delete []pR->DataArray;

delete pR;

}

else

{

//当所利用的散列位数等于N的位数，建立链表块

p = new myHashBlock;

p->CheckNUM = 0;

p->DataNUM = bufferSize;

p->DataArray = new myData[bufferSize];

p->nextBlockNum = node->BlockNum;

node->BlockNum = blockNum;

blockNum++;

p->BlockIndex = data.Index;

p->IndexNum = 32;

p->DataArray[p->CheckNUM++] = data;

writeBlock(node->BlockNum,p);

delete []p->DataArray;

delete p;

}

return true;

}

//删除某一项数据

void deleteData(int current,myHashBlock \*&pB)

{

int i;

for(i=current;i<pB->CheckNUM;i++)

if(i<pB->DataNUM-1)

pB->DataArray[i]= pB->DataArray[i+1];

pB->CheckNUM--;

return ;

}

//判断删除某一项以后，其父节点下两个节点是否可以合并

void checkCombine(myHashNode \*parent)

{

int i;

myHashBlock \*pBL,\*pBR ;

pBL = readBlock(parent->lChild->BlockNum);

pBR = readBlock(parent->rChild->BlockNum);

if((pBL->CheckNUM+pBR->CheckNUM+2)>pBL->DataNUM)

return ;

parent->lChild = parent->rChild = NULL;

parent->BlockNum = parent->lChild->BlockNum;

for(i=0;i<pBR->CheckNUM;i++)

{

pBL->DataArray[pBL->CheckNUM+1] = pBR->DataArray[i];

pBL->CheckNUM++;

}

pBL->IndexNum--;

writeBlock(parent->BlockNum,pBL);

delete []pBR->DataArray;

delete pBR;

return ;

}

};

int main()

{

int choose,buSize=8;

unsigned i,randNum,delIndex,cheIndex,j;

string randData,s1,s2;

bool flag,flag1 = false,flag2 = false;

myData data1,data2,data3,\*data4;

myHashIndex myHash(buSize);

BSzie = 2;

ofstream out;

char buf[10];

clock\_t start,finish;

double duration;

while(11)

{

system("cls");

cout<<"Please choose the operation of the HashIndex:\n";

cout<<"1.Set small number data test from newly created data\n";

cout<<"2.Set small number data test from exit data\n";

cout<<"3.Set great number data test from newly created data\n";

cout<<"4.Set great number data test from exit data\n";

cout<<"5.Save tree and Exit\n";

cin>>choose;

switch(choose)

{

case 1:

out.open("sData.dat",ios::out);

if(!out)

{

cout<<"Output file open failed";

}

out.close();

srand( (unsigned)time(NULL));

myHash.TreeFile[0] = 's';

myHash.DataFile[0] = 's';

for(i=1,j=-1;i<500;i++,j--)

{

randNum = rand();

data1.Index = randNum;

sprintf(buf,"%d",randNum);

strcpy(data1.Data ,"cai+");

strcat(data1.Data,buf);

myHash.Insert(data1);

}

break;

case 2:

myHash.TreeFile[0] = 's';

myHash.DataFile[0] = 's';

myHash.readTree();

break;

case 3:

out.open("gData.dat",ios::out);

if(!out)

{

cout<<"Output file open failed";

}

out.close();

srand( (unsigned)time(NULL));

myHash.TreeFile[0] = 'g';

myHash.DataFile[0] = 'g';

for(i=1;i<50000;i++)

{

randNum = i;

data1.Index = randNum;

sprintf(buf,"%d",randNum);

strcpy(data1.Data ,"cai+");

strcat(data1.Data,buf);

myHash.Insert(data1);

}

break;

case 4:

myHash.TreeFile[0] = 'g';

myHash.DataFile[0] = 'g';

myHash.readTree();

break;

case 5:

flag1 = true ;

break;

default:

cout<<"Input Error!";

continue;

}

if(flag1)

break;

flag2 = false;

while(22)

{

system("cls");

cout<<"Please choose the detail of the operation:\n";

cout<<"1.Insert data\n";

cout<<"2.Delete data\n";

cout<<"3.Check data\n";

cout<<"4.Save and exit\n";

cin>>choose;

switch (choose)

{

case 1:

cout<<"Please input the index and the data:\n";

cin>>data3.Index>>data3.Data;

start=clock();

flag = myHash.Insert(data3);

if(flag)

cout<<"Insert successed!\n";

else

cout<<"Insert failed!\n";

finish=clock();

break;

case 2:

cout<<"Please input the index of the data which will be deleted:\n";

cin>>delIndex;

start=clock();

flag = myHash.DeleteAll(delIndex);

if(flag)

cout<<"Delete successed!\n";

else

cout<<"Delete failed!\n";

finish=clock();

break;

case 3:

cout<<"Please input the index of the data which will be checked:\n";

cin>>cheIndex;

start=clock();

data4 = myHash.Check(cheIndex);

if(data4!=NULL)

{

cout<<"The detail data of the checked is :\n";

cout<<"Index is "<<data4->Index<<", data is "<<data4->Data<<endl;

}

else

{

cout<<"Check failed!\n";

}

finish=clock();

break;

case 4:

start=clock();

myHash.saveTree();

flag2 = true;

finish=clock();

break;

default:

cout<<"Input Error!\n";

break;

}

if(flag2)

break;

duration=(double)(finish-start);

cout<<"The used of the time is "<<duration <<" ms!"<<endl;

system("pause");

}

}

return 0;

}