C/C++从入门到精通-高级程序员之路

【 数据结构 】

哈希表及其企业级应用

第 1 节 哈希表的故事导入

故事情节

为了提高开发团队精神，缓解工作压力，某 IT 公司组织开发团队的 12 位男同事和测试团队的 12 位女同事开展真人 CS 4vs4 野战联谊！面对性感的女同事，男同事们个个摩拳擦掌，跃跃欲试！

野战活动那天，根据男女搭配，干活不累的原则，带队的专业教练让男同事站成一排，女同事站成一排，然后要求从女生这排开始从 1 开始报数，每个报数的队员都要记住自己的编号：
1， 2， 3，4。。。。。。林子里响起了百灵鸟般的报数声！

报数时，教练发给每人一个白色的臂章贴在肩膀上，每个臂章上写着报数人自己报过的编号！

当所有人都报完数后，教练发出命令将 24 人均分成 6 个组!
编号除 6 能整除的为第一组： 6 12 18 24
编号除 6 余数为 1 的为第二组： 1 7 13 19
编号除 6 余数为 2 的为第三组： 2 8 14 20
编号除 6 余数为 3 的为第四组： 3 9 15 21
编号除 6 余数为 4 的为第五组： 4 10 16 22
编号除 6 余数为 5 的为第六组： 5 11 17 23

通过这种编号方式划分队列，无论队员归队，还是裁判确认队员身份，都非常方便，此后林子里传来隆隆的笑声和枪炮声！

这种编号的方式就是高效的散列，我们俗称“哈希”！

以上过程是通过把关键码值 key（编号）映射到表中一个位置(数组的下标)来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

第 2 节 哈希表的原理精讲

哈希表 - 散列表，它是基于快速存取的角度设计的，也是一种典型的“空间换时间”的做法

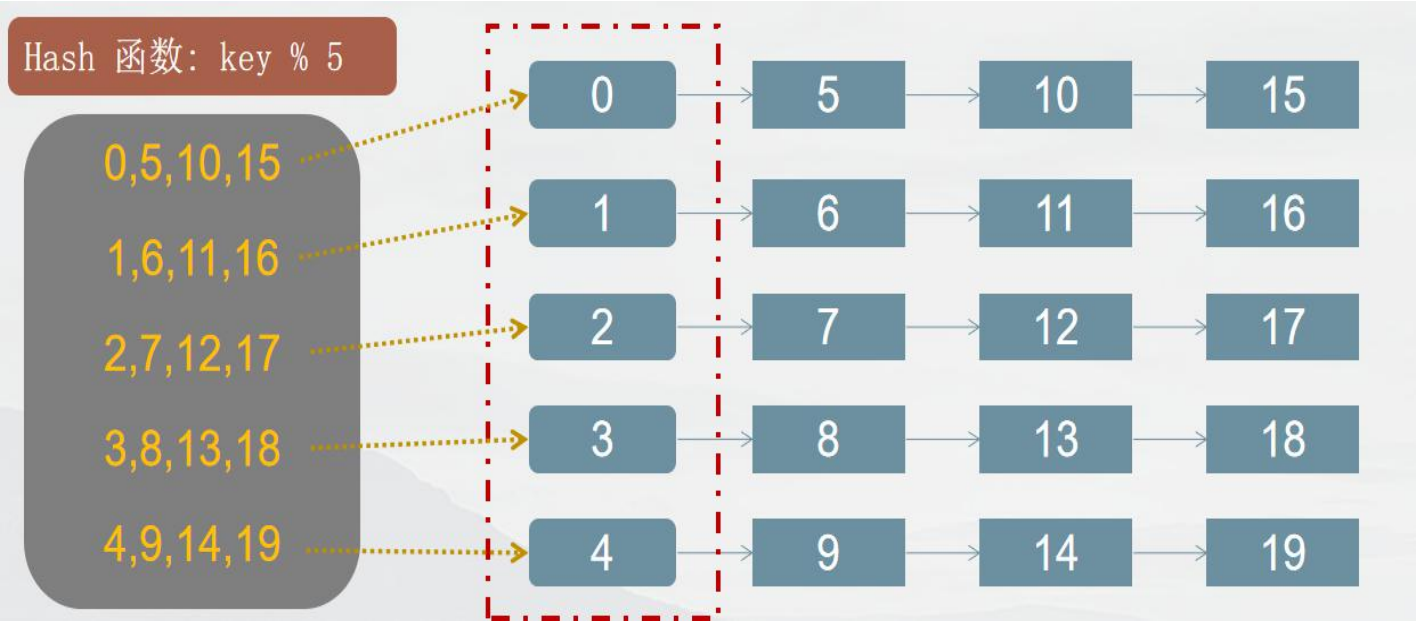
键(key)： 组员的编号 如， 1 、 5 、 19 。 。 。

值(value)： 组员的其它信息（包含 性别、年龄和战斗力等）

索引: 数组的下标(0,1,2,3,4) ，用以快速定位和检索数据

哈希桶: 保存索引的数组(链表或数组)，数组成员为每一个索引值相同的多个元素

哈希函数: 将组员编号映射到索引上，采用求余法 ，如： 组员编号 19



第 3 节　哈希链表的算法实现

哈希链表数据结构的定义

```
#define  DEFAULT_SIZE  16

typedef struct _ListNode
{
    struct _ListNode *next;
    int key;
    void *data;
}ListNode;

typedef ListNode *List;
typedef ListNode *Element;

typedef struct _HashTable
{
    int TableSize;
    List *Thelists;
}HashTable;
```

哈希函数

```
/*根据 key 计算索引，定位 Hash 桶的位置*/
int Hash(int key, int TableSize)
{
    return (key%TableSize);
}
```

哈希链表初始化

```
/*初始化哈希表*/
HashTable *InitHash(int TableSize)
{
    int i = 0;
    HashTable *hTable = NULL;

    if (TableSize <= 0) {
        TableSize = DEFAULT_SIZE;
    }

    hTable = (HashTable *)malloc(sizeof(HashTable));
    if (NULL == hTable)
    {
        printf("HashTable malloc error.\n");
        return NULL;
    }
    hTable->TableSize = TableSize;

    //为 Hash 桶分配内存空间，其为一个指针数组
    hTable->Thelists = (List *)malloc(sizeof(List)*TableSize);
    if (NULL == hTable->Thelists)
    {
        printf("HashTable malloc error\n");
        free(hTable);
        return NULL;
    }
    //为 Hash 桶对应的指针数组初始化链表节点
    for (i = 0; i < TableSize; i++)
    {
        hTable->Thelists[i] = (ListNode *)malloc(sizeof(ListNode));
        if (NULL == hTable->Thelists[i])
        {
            printf("HashTable malloc error\n");
            free(hTable->Thelists);
            free(hTable);
            return NULL;
        }
        else
        {
            memset(hTable->Thelists[i], 0, sizeof(ListNode));
        }
    }

    return hTable;
}
```

哈希链表插入元素

```
/*哈希表插入元素，元素为键值对*/
void Insert(HashTable *HashTable, int key, void *value )
{
    Element e=NULL, tmp=NULL;
    List L=NULL;
    e = Find(HashTable, key);

    if (NULL == e)
    {
        tmp = (Element)malloc(sizeof(ListNode));
        if (NULL == tmp)
        {
            printf("malloc error\n");
            return;
        }
        L = HashTable->Thelists[Hash(key, HashTable->TableSize)];
        tmp->data = value;
        tmp->key = key;
        tmp->next = L->next;
        L->next = tmp;
    }
    else
        printf("the key already exist\n");
}
```

哈希链表查找元素

```
/*从哈希表中根据键值查找元素*/
Element Find(HashTable *HashTable, int key)
{
    int i = 0;
    List L = NULL;
    Element e = NULL;
    i = Hash(key, HashTable->TableSize);
    L = HashTable->Thelists[i];
    e = L->next;
    while (e != NULL && e->key != key)
        e = e->next;

    return e;
}
```

哈希链表删除元素

```
/*哈希表删除元素，元素为键值对*/
void Delete(HashTable *HashTable, int key )
{
    Element e=NULL, last=NULL;
    List L=NULL;
    int i = Hash(key, HashTable->TableSize);
    L = HashTable->Thelists[i];

    last = L;
    e = L->next;
    while (e != NULL && e->key != key){
        last = e;
        e = e->next;
    }

    if(e){//如果键值对存在
        last->next = e->next;
        delete(e);
    }
}
```

源码实现：

hash_table.h

```
#pragma once

#define  DEFAULT_SIZE  16

/*哈希表元素定义*/
typedef struct _ListNode
{
    struct _ListNode *next;
    int key;
    void *data;
}ListNode;

typedef ListNode *List;
typedef ListNode *Element;

/*哈希表结构定义*/
typedef struct _HashTable
{
    int TableSize;
```

```
    List *Thelists;
}HashTable;

/*哈希函数*/
int Hash( void *key, int TableSize );

/*初始化哈希表*/
HashTable *InitHash( int TableSize );

/*哈希表插入*/
void Insert(HashTable *HashTable, int key, void *value);

/*哈希表查找*/
Element Find( HashTable *HashTable, int key);

/*哈希表销毁*/
void Destory( HashTable *HashTable );

/*哈希表元素中提取数据*/
void *Retrieve( Element e );
```

hash_table.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hash_table.h"


/*根据 key 计算索引，定位 Hash 桶的位置*/
int Hash(int key, int TableSize)
{
    return (key%TableSize);
}

/*初始化哈希表*/
HashTable *InitHash(int TableSize)
{
    int i = 0;
    HashTable *hTable = NULL;

    if (TableSize <= 0) {
        TableSize = DEFAULT_SIZE;
    }

    hTable = (HashTable *)malloc(sizeof(HashTable));
```

```
    if (NULL == hTable)
    {
        printf("HashTable malloc error.\n");
        return NULL;
    }

    hTable->TableSize = TableSize;

    //为 Hash 桶分配内存空间，其为一个指针数组
    hTable->Thelists = (List *)malloc(sizeof(List)*TableSize);
    if (NULL == hTable->Thelists)
    {
        printf("HashTable malloc error\n");
        free(hTable);
        return NULL;
    }

    //为 Hash 桶对应的指针数组初始化链表节点
    for (i = 0; i < TableSize; i++)
    {
        hTable->Thelists[i] = (ListNode *)malloc(sizeof(ListNode));
        if (NULL == hTable->Thelists[i])
        {
            printf("HashTable malloc error\n");
            free(hTable->Thelists);
            free(hTable);
            return NULL;
        }
        else
        {
            memset(hTable->Thelists[i], 0, sizeof(ListNode));
        }
    }

    return hTable;
}

/*从哈希表中根据键值查找元素*/
Element Find(HashTable *HashTable, int key)
{
    int i = 0;
    List L = NULL;
    Element e = NULL;
    i = Hash(key, HashTable->TableSize);
    L = HashTable->Thelists[i];
    e = L->next;
    while (e != NULL && e->key != key)
```

```
        e = e->next;

    return e;
}

/*哈希表插入元素，元素为键值对*/
void Insert(HashTable *HashTable, int key, void *value )
{
    Element e=NULL, tmp=NULL;
    List L=NULL;
    e = Find(HashTable, key);

    if (NULL == e)
    {
        tmp = (Element)malloc(sizeof(ListNode));
        if (NULL == tmp)
        {
            printf("malloc error\n");
            return;
        }
        L = HashTable->Thelists[Hash(key, HashTable->TableSize)];
        tmp->data = value;
        tmp->key = key;
        tmp->next = L->next;
        L->next = tmp;
    }
    else
        printf("the key already exist\n");
}

/*哈希表删除元素，元素为键值对*/
void Delete(HashTable *HashTable, int key )
{
    Element e=NULL, last=NULL;
    List L=NULL;
    int i = Hash(key, HashTable->TableSize);
    L = HashTable->Thelists[i];

    last = L;
    e = L->next;
    while (e != NULL && e->key != key){
        last = e;
        e = e->next;
    }

    if(e){//如果键值对存在
        last->next = e->next;
```

```
        delete(e);
    }
}

/*哈希表元素中提取数据*/
void *Retrieve(Element e)
{
    return e?e->data:NULL;
}

/*销毁哈希表*/
void Destory(HashTable *HashTable)
{
    int i=0;
    List L = NULL;
    Element cur = NULL, next = NULL;
    for (i=0; i < HashTable->TableSize; i++)
    {
        L = HashTable->Thelists[i];
        cur = L->next;
        while (cur != NULL)
        {
            next = cur->next;
            free(cur);
            cur = next;
        }
        free(L);
    }
    free(HashTable->Thelists);
    free(HashTable);
}

void main(void)
{
    char *elems[] = { "翠花","小芳","苍老师" };
    int i = 0;

    HashTable *HashTable;
    HashTable = InitHash(31);
    Insert(HashTable, 1, elems[0]);
    Insert(HashTable, 2, elems[1]);
    Insert(HashTable, 3, elems[2]);
    Delete(HashTable, 1);

    for (i = 0; i < 4; i++) {
        Element e = Find(HashTable, i);
        if (e) {
```

```
            printf("%s\n", (const char *)Retrieve(e));
        }
        else {
            printf("Not found [key:%d]\n",i);
        }
    }
    system("pause");
}
```

第 4 节　哈希表的顺序存储实现

(作为课后作业自行实现，完成后交与 Martin 老师检查)

第 5 节 哈希表的企业级应用案例

5.1 淘宝分布式文件系统

项目背景介绍

根据淘宝 2016 年的数据分析，淘宝卖家已经达到 900 多万，有上十亿的商品。每一个商品有包括大量的图片和文字(平均：15k)，粗略估计下，数据所占的存储空间在 1PB 以上，如果使用单块容量为 1T 容量的磁盘来保存数据，那么也需要 1024 x 1024 块磁盘来保存.

```
1 PB = 1024 TB = 1024 * 1024 GB
```

思考？ 这么大的数据量，应该怎么保存呢？就保存在普通的单个文件中或单台服务器中吗？显然是不可行的。

淘宝针对海量非结构化数据存储设计出了的一款分布式系统,叫 TFS,它构筑在普通的 Linux 机器集群上,可为外部提供高可靠和高并发的存储访问。

设计思路

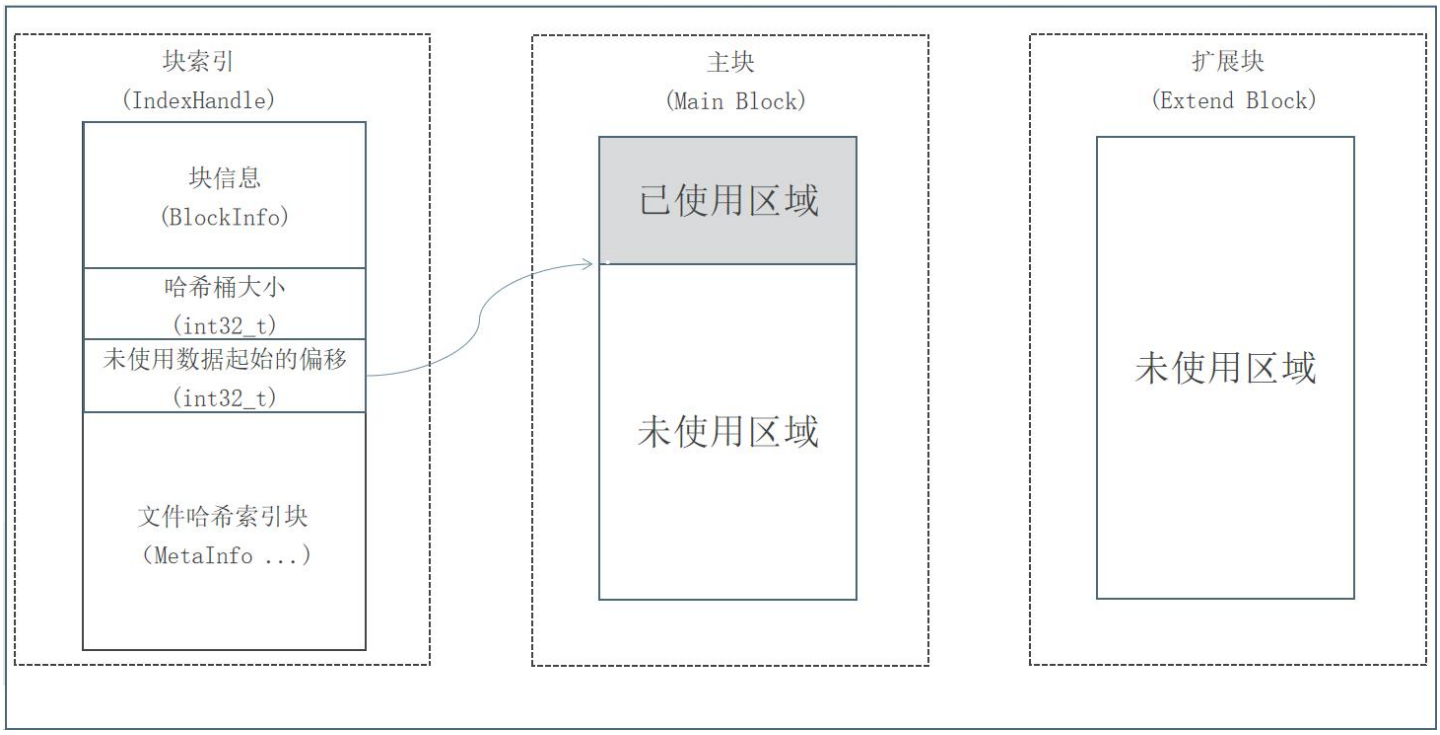
以 block 文件的形式存放数据文件(一般 64M 一个 block),以下简称为“块”，每个块都有唯一的一个整数编号，块在使用之前所用到的存储空间都会预先分配和初始化。

每一个块由一个索引文件、一个主块文件和若干个扩展块组成，“小文件”主要存放在主块中，扩展块主要用来存放溢出的数据。

每个索引文件存放对应的块信息和“小文件”索引信息，索引文件会在服务启动是映射（mmap）到内存，以便极大的提高文件检索速度。“小文件”索引信息采用在索引文件中的数据结构哈希链表来实现。

每个文件有对应的文件编号，文件编号从 1 开始编号，依次递增，同时作为哈希查找算法的 Key 来定位“小文件”在主块和扩展块中的偏移量。文件编号+块编号按某种算法可得到“小文件”对应的文件名。

大文件存储结构图



哈希链表实现

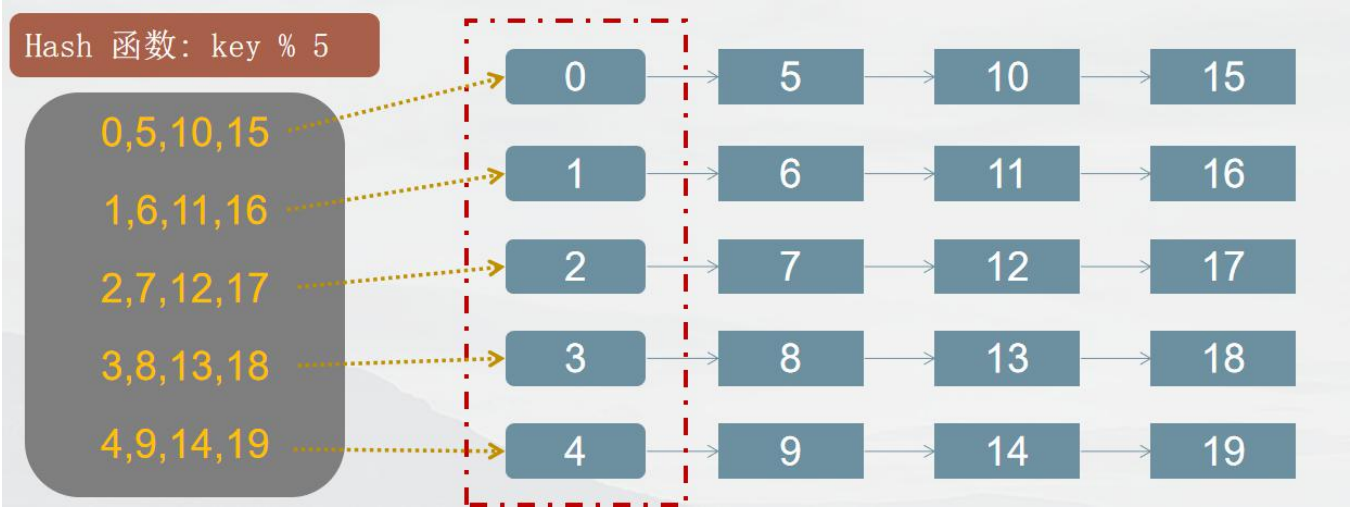
键(key)： 文件的编号 如， 1 、 5 、 19 。 。 。

值(value)： 文件的索引信息（包含 文件大小、位置）

索引： 数组的下标(0,1,2,3,4) ，用以快速定位和检索数据

哈希桶： 保存索引的数组，数组成员为索引值相同的多个元素（以链表的形式链接）

哈希函数: 将文件编号映射到索引上，采用求余法 ，如： 文件编号 19



5.2 DNA 检测字串匹配

随着生物基因测试的技术成熟，科学家们可以通过基因相似度检测，现在要对 N 个人进行测试基因测试，通过基因检测是否为色盲。

测试色盲的基因组包含 8 位基因，编号 1 至 8。每一位基因都可以用一个字符来表示，这个字符是'A'、'B'、'C'、'D'四个字符之一。

如：ABDBCBAD

通过认真观察研究，生物学家发现，有时候可能通过特定的连续几位基因，就能区分开是正常者还是色盲者。对于色盲基因，不需要 8 位基因，只需要看其中连续的 4 位基因就可以判定是正常者还是色盲者，这 4 位基因编号分别是：（第 2、3、4、5）。也就是说，只需要把第 2,3,4,5 这四位连续的基因与色盲基因库的记录对比，就能判定该人是正常者还是色盲者。

假设给定的色盲基因库如下：
ADBB
BDDC
CDBC
BDBB
.....

请测试下列的基因是否为色盲
AADBBBAD
ABDDCBAA
CCDBCBAA
ABDBBBAC
ABDBCBAD
ABDDBBAD

解答思路：

1. 可以直接把待测试基因的 2,3,4,5 位直接与基因库里的记录逐一对比，但如果色盲基因库很庞大，程序执行效率很低

2. 可以使用哈希表来存储色盲基因库数据，通过哈希函数把 4 位色盲基因映射到哈希表中，大大提高检索的效率.

源码实现：

```
#pragma once

#define  DEFAULT_SIZE  16

/*哈希表元素定义*/
typedef struct _ListNode
{
    struct _ListNode *next;
    void *key;
    void *data;
}ListNode;

typedef ListNode *List;
typedef ListNode *Element;

/*哈希表结构定义*/
typedef struct _HashTable
{
    int TableSize;
    List *Thelists;
}HashTable;

/*哈希函数*/
int Hash( void *key, int TableSize );

/*初始化哈希表*/
HashTable *InitHash( int TableSize );
```

```
/*哈希表插入*/
void Insert(HashTable *HashTable, void *key, void *value);

/*哈希表查找*/
Element Find( HashTable *HashTable, void *key);

/*哈希表销毁*/
void Destory( HashTable *HashTable );

/*哈希表元素中提取数据*/
void *Retrieve( Element e );
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hash_table.h"

#define BUCKET_SIZE  1024

#define compare(a,b)     strcmp((const char*)a, (const char*)b)
#define hash_func       SDBMHash

unsigned int SDBMHash(void *key)
{
    unsigned int hash = 0;
    char *str = (char*)key;

    while (*str)
    {
        // equivalent to: hash = 65599*hash + (*str++);
        hash = (*str++) + (hash << 6) + (hash << 16) - hash;
    }

    return (hash & 0x7FFFFFFF);
}


/*根据 key 计算索引，定位 Hash 桶的位置*/
int Hash(void *key, int TableSize)
{
    //return (key%TableSize);
```

```
    return hash_func(key)%TableSize;  //"CDBC" => 16011 %1024 =>
    //return 0;
}

/*初始化哈希表*/
HashTable *InitHash(int TableSize)
{
    int i = 0;
    HashTable *hTable = NULL;

    if (TableSize <= 0) {
        TableSize = DEFAULT_SIZE;
    }

    hTable = (HashTable *)malloc(sizeof(HashTable));
    if (NULL == hTable)
    {
        printf("HashTable malloc error.\n");
        return NULL;
    }

    hTable->TableSize = TableSize;

    //为 Hash 桶分配内存空间，其为一个指针数组
    hTable->Thelists = (List *)malloc(sizeof(List)*TableSize);
    if (NULL == hTable->Thelists)
    {
        printf("HashTable malloc error\n");
        free(hTable);
        return NULL;
    }

    //为 Hash 桶对应的指针数组初始化链表节点
    for (i = 0; i < TableSize; i++)
    {
        hTable->Thelists[i] = (ListNode *)malloc(sizeof(ListNode));
        if (NULL == hTable->Thelists[i])
        {
            printf("HashTable malloc error\n");
            free(hTable->Thelists);
            free(hTable);
            return NULL;
        }
        else
        {
            memset(hTable->Thelists[i], 0, sizeof(ListNode));
        }
```

```
    }

    return hTable;
}

/*从哈希表中根据键值查找元素*/
Element Find(HashTable *HashTable, void *key)
{
    int i = 0;
    List L = NULL;
    Element e = NULL;
    i = Hash(key, HashTable->TableSize);
    L = HashTable->Thelists[i];
    e = L->next;
    while (e != NULL && compare(e->key, key)!=0){
        e = e->next;
    }

    return e;
}

/*哈希表插入元素，元素为键值对*/
void Insert(HashTable *HashTable, void *key, void *value )
{
    Element e=NULL, tmp=NULL;
    List L=NULL;
    e = Find(HashTable, key);

    if (NULL == e)
    {
        tmp = (Element)malloc(sizeof(ListNode));
        if (NULL == tmp)
        {
            printf("malloc error\n");
            return;
        }
        int code = Hash(key, HashTable->TableSize);
        L = HashTable->Thelists[code];//前插法
        tmp->data = value;
        tmp->key = key;
        tmp->next = L->next;
        L->next = tmp;
    }
    else
        printf("the key already exist\n");
}
```

```
/*哈希表删除元素，元素为键值对*/
void Delete(HashTable *HashTable, void *key)
{
    Element e=NULL, last=NULL;
    List L=NULL;
    int i = Hash(key, HashTable->TableSize);
    L = HashTable->Thelists[i];

    last = L;
    e = L->next;
    while (e != NULL && e->key != key){
        last = e;
        e = e->next;
    }

    if(e){//如果键值对存在
        last->next = e->next;
        free(e);
    }
}

/*哈希表元素中提取数据*/
void *Retrieve(Element e)
{
    return e?e->data:NULL;
}

/*销毁哈希表*/
void Destory(HashTable *HashTable)
{
    int i=0;
    List L = NULL;
    Element cur = NULL, next = NULL;
    for (i=0; i < HashTable->TableSize; i++)
    {
        L = HashTable->Thelists[i];
        cur = L->next;
        while (cur != NULL)
        {
            next = cur->next;
            free(cur);
            cur = next;
        }
        free(L);
    }
    free(HashTable->Thelists);
    free(HashTable);
```

```
}

void main(void)
{
    char *elems[] = { "ADBB","BDDC","CDBC","BDBB" };
    char * tester = "ABDBBBAC";
    char cur[5]={'\0'};

    int i = 0;

    HashTable *HashTable = NULL;
    HashTable = InitHash(BUCKET_SIZE);
    Insert(HashTable, elems[0], elems[0]);
    Insert(HashTable, elems[1], elems[1]);
    Insert(HashTable, elems[2], elems[2]);
    Insert(HashTable, elems[3], elems[3]);
    //Delete(HashTable, elems[0]);

    strncpy_s(cur, tester+1, 4);//ADBB'\0'


    Element e = Find(HashTable, cur);
    if (e) {
        printf("%s\n", (const char *)Retrieve(e));
    }
    else {
        printf("Not found [key:%s]\n",cur);
    }

    system("pause");
}
```