```c
/* ClosedHash.c */
#include <stdlib.h>
#include <malloc.h>
#include <assert.h>
#include <string.h>
#include "ClosedHash.h"

static int IntCmp(const void *keyAddr, const void *dataAddr)
{
    int *p1 = (int *)keyAddr;
    int *p2 = (int *)dataAddr;
    return (*p1 - *p2);
}

//散列表初始化
void HashNew(HASH *h, int capacity, int keySize, int valSize, HashFunc *hashFn,
HashCollide* collideFn, HashCmp *cmpFn, HashFree *freeFn)
{
    assert(0 < capacity);
    assert(0 < keySize);
    assert(0 <= valSize);
    assert(NULL != hashFn);
    assert(NULL != collideFn);
    h->capacity = capacity;
    h->keySize = keySize;
    h->valSize = valSize;
    h->size = 0;
    h->hashFn = hashFn;
    h->collideFn = collideFn;
    h->cmpFn = cmpFn;
    h->freeFn = freeFn;
    VectorNew(&(h->vEntry), keySize + valSize, capacity, 0, cmpFn, freeFn);
    int i;
    //将所有节点的key置为-1
    void *initEntry = malloc(keySize + valSize);
    assert(NULL != initEntry);
    memset(initEntry, -1, keySize + valSize);
    for (i = 0; i < capacity; i ++)
    {
        VectorInsertByPos(&(h->vEntry), initEntry, i);
    }
    free(initEntry);
    VectorNew(&(h->vLazy), sizeof(int), capacity, 0, IntCmp, NULL);
    //将所有节点的lazyFlag置0
    int initLazy = 0;
    for (i = 0; i < capacity; i ++)
    {
        VectorInsertByPos(&(h->vLazy), &initLazy, i);
    }
}

//获取散列表数据数量
int HashSize(HASH *h)
{
    return h->size;
}

//获取散列表容量
int HashCapacity(HASH *h)
{
    return h->capacity;
}

//散列表销毁
void HashDispose(HASH *h)
{
    VectorDispose(&(h->vEntry));
    VectorDispose(&(h->vLazy));
    h->size = 0;
}

//沿关键码对应的查找链，找到与之匹配的桶
static int probe4Hit(HASH *h, int hashKey, const void *e)
```

```
73  {
74      int r = hashKey;
75      void *fEmpty = malloc(h->keySize);
76      assert(NULL != fEmpty);
77      memset(fEmpty, -1, h->keySize);
78      int fLazy = 1;
79      int count = 0;
80      //跳过所有冲突的桶，以及带懒惰删除标记的桶
81      while ((!VectorFind(&(h->vEntry), r, fEmpty) && !VectorFind(&(h->vEntry), r, e))
82          || (VectorFind(&(h->vEntry), r, fEmpty) && (VectorFind(&(h->vLazy), r,
            &fLazy))))
83      {
84          count ++;
85          r = h->collideFn(r, count);
86      }
87      free(fEmpty);
88      return r;
89  }
90
91  //散列表读取
92  void *HashGet(HASH *h, const void *e)
93  {
94      int hashKey = h->hashFn(e);
95      hashKey = probe4Hit(h, hashKey, e);
96      if (!VectorFind(&(h->vEntry), hashKey, e))
97      {
98          return NULL;
99      }
100     return VectorGetByPos(&(h->vEntry), hashKey);
101 }
102
103 static int probe4Free(HASH *h, int hashKey)
104 {
105     int r = hashKey;
106     void *fEmpty = malloc(h->keySize);
107     assert(NULL != fEmpty);
108     memset(fEmpty, -1, h->keySize);
109     int fLazy = 1;
110     int count = 0;
111     //跳过所有冲突的桶，以及带懒惰删除标记的桶
112     while (!VectorFind(&(h->vEntry), r, fEmpty))
113     {
114         count ++;
115         r = h->collideFn(r, count);
116     }
117     free(fEmpty);
118     return r;
119 }
120
121 //散列表插入
122 int HashPut(HASH *h, const void *e, const void *val)
123 {
124     int hashKey = h->hashFn(e);
125     hashKey = probe4Hit(h, hashKey, e);
126     if (VectorFind(&(h->vEntry), hashKey, e))
127     {
128         return -1;
129     }
130     hashKey = probe4Free(h, hashKey);
131     void *data = malloc(h->keySize + h->valSize);
132     assert(NULL != data);
133     memcpy(data, e, h->keySize);
134     memcpy((char *)data + h->keySize, val, h->valSize);
135     VectorUpdate(&(h->vEntry), hashKey, data);
136     free(data);
137     h->size ++;
138     return 0;
139 }
140
141 //散列表删除
142 int HashRemove(HASH *h, void *e)
143 {
144     int hashKey = h->hashFn(e);
```

```
145         hashKey = probe4Hit(h, hashKey, e);
146         if (!VectorFind(&(h->vEntry), hashKey, e))
147         {
148             return -1;
149         }
150         void *initEntry = malloc(h->keySize + h->valSize);
151         assert(NULL != initEntry);
152         memset(initEntry, -1, h->keySize + h->valSize);
153         VectorRemoveByPos(&(h->vEntry), hashKey);
154         VectorInsertByPos(&(h->vEntry), initEntry, hashKey);
155         free(initEntry);
156         int fRemove = 1;
157         VectorUpdate(&(h->vLazy), hashKey, &fRemove);
158         h->size --;
159         return 0;
160     }
161
162     //重散列
163     static void HashRehash(HASH *hN, HASH *hO)
164     {
165         int i = 0;
166         void *fEmpty = malloc(hO->keySize);
167         assert(fEmpty);
168         memset(fEmpty, -1, hO->keySize);
169         for (; i < hO->capacity; i ++)
170         {
171             if (!VectorFind(&(hO->vEntry), i, fEmpty))
172             {
173                 void *dataGet = VectorGetByPos(&(hO->vEntry), i);
174                 HashPut(hN, dataGet, dataGet + hO->keySize);
175             }
176         }
177     }
```