第4章：哈希表的实现

1、哈希函数H(key)为关键字(标识符)的第一个字母在字母表中的序号，处理冲突的方法为线性探测开放定址法。编写一个按第一个字母的顺序输出哈希表中所有关键字的算法。

```
/**********
【题目】已知某哈希表的装载因子小于1，哈希函数H(key)
为关键字(标识符)的第一个字母在字母表中的序号，处理
冲突的方法为线性探测开放定址法。试编写一个按第一个
字母的顺序输出哈希表中所有关键字的算法。
哈希表的类型HashTable定义如下：
#define SUCCESS    1
#define UNSUCCESS  0
#define DUPLICATE -1
typedef char StrKeyType[4];
typedef struct {
   StrKeyType key; // 关键字项
   int    tag;     // 标记 0:空；1:有效; -1:已删除
   void  *any;     // 其他信息
} RcdType;
typedef struct {
  RcdType *rcd;    // 存储空间基址
  int      size;   // 哈希表容量
  int      count;  // 表中当前记录个数
} HashTable;
**********/
void PrintKeys(HashTable ht, void(*print)(StrKeyType))
/* 依题意用print输出关键字 */
{
      char c = 'A';
      for(int num = 0;num <= ht.count&&c <= 'Z';c ++){
           //对size求余是因为不一定有26个空间
           //从0开始，因为rcd是从0开始存数据
           int hashValue = (c-'A')%ht.size;
           /*该退出条件如果遇到所有空间都不为空会死循环*/
           while(ht.rcd[hashValue].tag != 0){
                if(ht.rcd[hashValue].tag == 1){
                    if(ht.rcd[hashValue].key[0] == c){
                        print(ht.rcd[hashValue].key);
                        //测试数据似乎把删除元素也算一个记录
                        num ++;
                    }
                }
                /*根据线性探测法探测后面是否还有相同元素*/
                hashValue = (hashValue + 1)%ht.size;
           }
      }
}
```

2、用链地址法处理冲突。试编写输入一组关键字并建造哈希表的算法

```
/**********
【题目】假设哈希表长为m，哈希函数为H(x)，用链地址法
处理冲突。试编写输入一组关键字并建造哈希表的算法。
哈希表的类型ChainHashTab定义如下：
#define NUM         7
#define NULLKEY    -1
#define SUCCESS     1
#define UNSUCCESS   0
#define DUPLICATE  -1
typedef char HKeyType;
typedef struct HNode {
   HKeyType  data;
   struct HNode*  next;
}*HLink;
typedef struct {
   HLink  *rcd;   // 指针存储基址，动态分配数组
   int    count;  // 当前表中含有的记录个数
   int    size;  // 哈希表的当前容量
}ChainHashTab;    // 链地址哈希表
int Hash(ChainHashTab H, HKeyType k) { // 哈希函数
  return k % H.size;
}
Status Collision(ChainHashTab H, HLink &p) {
  // 求得下一个探查地址p
  if (p && p->next) {
    p = p->next;
    return SUCCESS;
  } else return UNSUCCESS;
}
**********/
int SearchHash(ChainHashTab &H,int p,HKeyType es){
    int flag = 0;
    HLink temp = H.rcd[p];
    /*判断是否已经存在该关键字*/
    for(;temp != NULL;temp = temp->next){
        if(temp->data == es){
            flag = 1;//该关键字已存在
            break;
        }
    }
    return flag;
}
int BuildHashTab(ChainHashTab &H, int n, HKeyType es[])
/* 直接调用下列函数                                  */
/* 哈希函数:                                         */
/*    int Hash(ChainHashTab H, HKeyType k);      */
/* 冲突处理函数:                                     */
/*    int Collision(ChainHashTab H, HLink &p);   */
{
      int flag = 0;//标志是否重复，1表示存在
      for(int i = 0;i < n;i ++){
            int p = Hash(H,es[i]);//得到哈希函数
            flag =  SearchHash(H,p,es[i]);//得到查找结果
```

```
            if(flag == 1){
                flag = 0;//记得置零
            }else{
                HLink np;
                /*开辟新节点*/
                np = (HLink)malloc(sizeof(struct HNode));
                if(np == NULL)return ERROR;
                np->data = es[i];
                /*没看懂Collision函数，自己手动解决冲突*/
                np->next = H.rcd[p];//插入到表头
                H.rcd[p] = np;
                H.count ++;
            }
        }
}
```