数据结构实验七

姓名：刘俊傲　　学号：**U201617047**　　班级：软工**1603**

1. 使用分离链路法处理冲突

1. 问题描述

    - Hash表的大小为2k-1，初始可以为15
    - 表中的装填因子达到3/4时，增加表的大小至2k+1 –1，完成再哈希
    - 实现插入，删除和查找操作

2. 问题分析与算法设计

    1. hash表 ：首先定义一个数组链表，元素存储在链表中，并且每个链表是首元素为仅为表头，不存储任何元素。
    2. 插入 ：首先找到链表数组位置，然后将元素查到元素前端
    3. 删除 ：同样先查找元素，如果存储，则删除，不存在，就提示没有该元素
    4. 查找 ：首先查找链表数组，然后依次遍历链表

3. 算法实现：

```
//使用分离链路法处理hash冲突
//并同时实现添加，删除，查找操作
#include <stdio.h>
#include <stdlib.h>

#define SUCCESS 1
#define FAILURE 0

typedef struct ListNode *position;
typedef position list;
typedef struct HashTable *hashTable;
typedef int ElementType;

hashTable initializeTable(int tablesize);
void destroyTable(hashTable H);
position find(ElementType key, hashTable hash);
void insert(ElementType key, hashTable hash);
int Hash(ElementType key, int tableSize);
int Delete(ElementType key, hashTable hash);
void extend(hashTable hash);

/*****************************
    *数据存储链表
    *element ：  存储的数据
    *next ：  指向下一个元素地址的指针
*****************************/
struct ListNode
{
    ElementType element;
    position next;
};

/*****************************
    *hash表
    *tableSize ：  hash表是尺寸
    *curSize ：  当前已经填入的元素的个数
    *thelists ：  指向ListNode的指针的指针
*****************************/
struct HashTable
{
    int tableSize;
    int curSize;
    list *thelists;
};

/*****************************
    *初始化hash表
    *tableSize ：  hash表输入尺寸，实际尺寸为 2^k - 1
*****************************/
hashTable initializeTable(int tableSize)
{
    //初始化为实际尺寸
    tableSize = (tableSize >> 1) - 1;
```

```
    hashTable hash;
    hash = (hashTable)malloc(sizeof(struct HashTable));
    if (hash == NULL)
    {
        printf("out of space\n");
        return NULL;
    }
    hash->tableSize = tableSize;
    hash->curSize = 0;

    //创建指针数组,数组里不放数据，数据放在链表里
    hash->thelists = (list*)malloc(sizeof(list) * hash->tableSize);

    if (hash->thelists == NULL)
    {
        printf("out of space!\n");
        return NULL;
    }
    //创建指针数组元素指向的链表表头
    for (int i = 0; i < tableSize; i++)
    {
        hash->thelists[i] = (struct ListNode*)malloc(sizeof(struct ListNode));
        if (hash->thelists[i] == NULL)
        {
            printf("out of space!\n");
            return NULL;
        }
        else
        {
            hash->thelists[i]->element = 0;
            hash->thelists[i]->next = NULL;
        }
    }
    return hash;
}

/*****************************
    *删除原有的hash表
    *hash :  hash表
*****************************/
void destroyTable(hashTable hash)
{
    position list, tmp, ptr;
    int i;
    for (i = 0; i < hash->tableSize; i++)
    {
        list = hash->thelists[i];
        ptr = list->next;
        while (ptr)
        {
            tmp = ptr->next;
            if (!tmp)

            {
```

```
                free(tmp);
                tmp = NULL;
            }
            else
            {
                free(ptr);
                ptr = tmp;
            }
        }
    }
    free(hash);
}

/*****************************
    *根据元素查找hash表
    *key ：  要查找的元素
    *hash ：  被查找的hash表
*****************************/
position find(ElementType key, hashTable hash)
{
    position pos;
    list list;
    //hash，找到 key 本该被存入的位置
    list = hash->thelists[Hash(key, hash->tableSize)];
    if (list == NULL)
        return NULL;
    pos = list->next;
    while(pos)
    {
        if (pos->element == key)
        {
            return pos;
        }
        pos = pos->next;
    }
    return NULL;
}

/*******************************
    *自定义hash方法，求模运算
    *key ：  元素值
    *tableSize ：  hash表尺寸
*******************************/
int Hash(ElementType key, int tableSize)
{
    return key % tableSize;
}

/*******************************
    *进行hash表元素的插入，如果元素个数超过hash表尺寸的3/4，则进行再hash
    *key ：  要插入的元素
    *hash ：  被插入的hash表

*******************************/
```

```
void insert(ElementType key, hashTable hash)
{
    position pos, cell;
    position lsit;
    pos = find(key, hash);
    if (pos == NULL) //错把pos=null当作判断条件，此条件永远为真。
    {
        cell = (struct ListNode*)malloc(sizeof(struct ListNode));
        if (cell == NULL)
        {
            printf("out of space\n");
            return;
        }
        else
        {
            lsit = hash->thelists[Hash(key, hash->tableSize)];
            if (lsit->next == NULL)
                hash->curSize++;
            //将元素插入到链表前端，list为链表表头，不存储元素
            cell->next = lsit->next;
            cell->element = key;
            lsit->next = cell;

            //将int型转化为float型
            //c编译系统会自动向高精度类型进行转化。
            if (hash->curSize * 1.0 / hash->tableSize >= 0.75)
            {
                extend(hash);
            }
        }
    }
}

/*******************************
    *实现 hash 表元素的删除
    *key : 要删除的元素
    *hash : 要被删除的 hash 表
*******************************/
int Delete(ElementType key, hashTable hash)
{
    position pos, list;
    //对应 hash 表位置的表头
    list = hash->thelists[Hash(key, hash->tableSize)];

    if (list->next == NULL)
    {
        printf("can't find that key!\n");
        return FAILURE;
    }
    else
    {
        while (list != NULL && list->next != NULL && list->next->element != key)

        {
```

```
            list = list->next;
        }
        //要删除的元素在链表的表尾
        if (list->next == NULL && list->element == key)
        {
            free(list);
            return SUCCESS;
        }
        if (list->next != NULL && list->next->element == key)
        {
            //pos 为要删除的元素的指针
            pos = list->next;
            list->next = pos->next;
            free(list);
            return SUCCESS;
        }
    }
    return FAILURE;
}

/*******************************
    *进行再 hash
    *hash : 被再 hash 的hash表
*******************************/
void extend(hashTable hash)
{
    int preSize = hash->tableSize;
    hash->tableSize = (1 << preSize) + 1;
    int nowSize = hash->tableSize;
    list* pre = hash->thelists;
    hash->thelists = (list*)malloc(sizeof(list) * nowSize);
    for (int i = 0; i < hash->tableSize; i++)
    {
        hash->thelists[i] = (struct ListNode*)malloc(sizeof(struct ListNode));
    }
    for (int i = 0; i < nowSize; i++)
    {
        list temp = pre[i]->next;
        while (temp != NULL)
        {
            insert(temp->element, hash);
            list temp2 = temp;
            temp = temp->next;
            free(temp2);
        }
    }
}
```

2. 多项式乘法的改进

1. 问题描述：

   > 稀流多项式乘法的改进

2. 问题分析与算法设计：

> **基本思路**是创建一个结构体节点，储存每一项的系数（int），指数（int），和一个next指针。
>
> 输入两个链表A和B后：
>
> 1. 当指数相等时，系数相成；相加完的系数若不为0，则存进链表C；
> 2. 若指数不相等，若A的指数小于B的，将A的节点里的系数，指数都存进链表C；
> 3. 反之，则将B的节点里的系数，指数都存进链表C；
> 4. 当有一个链表的后几项多出来时，也将其存进链表C。

3. 算法实现：

```
//稀流多项式乘法的改进

#include<stdio.h>
#include<stdlib.h>

typedef struct ployNode* mul_ploy;

mul_ploy get_tail_point(mul_ploy root);
mul_ploy createList(mul_ploy root);
mul_ploy sort(mul_ploy head, mul_ploy new_poly);

/***************************************
    *存储多项式各项的结构体
    *exp :  指数
    *para :  系数
    *next :  指向下一结构体的指针
***************************************/
struct ployNode
{
    int exp;
    int para;
    mul_ploy next;
};

/***************************************
    *返回链表的表尾
    *root :  链表指针
***************************************/
mul_ploy get_tail_point(mul_ploy root)
{
    while (root->next != NULL)
        root = root->next;
    return root;
}

/***************************************
    *输入指数和系数，并将指数和系数放入链表中
    *root :  要放入的已有链表
***************************************/
mul_ploy createList(mul_ploy root)
{
    int exp;
    int para;
    scanf("请输入指数:  %d", &exp);
    scanf("请输入参数:  %d", &para);
    mul_ploy new_node = (mul_ploy)malloc(sizeof(struct ployNode));
    new_node->exp = exp;
    new_node->para = para;
    new_node->next = NULL;
    mul_ploy tail = get_tail_point(root);
    tail->next = new_node;
}
```

```
/**************************************
    *从大到小排序
    *head :  链表的首元素节点
    *new_ploy :  要插入的元素节点
**************************************/
mul_ploy sort(mul_ploy head, mul_ploy new_poly)
{
    if (head->exp < new_poly->exp)
    {
        new_poly->next = head;
        head = new_poly;
    }
    else
    {
        mul_ploy temp = head;
        mul_ploy temp_pre = head;
        while (temp != NULL)
        {
            if (temp->exp < new_poly->exp)
            {
                temp_pre->next = new_poly;
                new_poly->next = temp;
                break;
            }
            else
            {
                if (temp->exp == new_poly->exp)
                {
                    temp->para += new_poly->para;
                    free(new_poly);
                    break;
                }
                else
                {
                    temp_pre = temp;
                    temp = temp->next;
                }

            }

        }
        if (temp == NULL)
        {
            temp_pre->next = new_poly;
        }
    }
    return head;
}

/**************************************
    *多项式乘法
    *first :  第一个多项式

    *second :  另一个多项式
```

```
**************************************/
mul_ploy multiply(mul_ploy first, mul_ploy second)
{
    if (first == NULL || second == NULL)
        return NULL;

    mul_ploy head = (mul_ploy)malloc(sizeof(struct ployNode));
    int count = 0;

    mul_ploy first_head = first;
    mul_ploy second_head = second;
    while (first)
    {
        second = second_head;
        while (second)
        {
            mul_ploy new_poly = (mul_ploy)malloc(sizeof(struct ployNode));

            new_poly->exp = first->exp + second->exp;
            new_poly->para = first->para * second->para;
            if (new_poly->para == 0)
                free(new_poly);
            else
            {
                if (count)
                {
                    head = sort(head, new_poly);
                }
                else
                {
                    head->exp = new_poly->exp;
                    head->para = new_poly->para;
                    free(new_poly);
                    count++;
                }
            }
        }
    }
    return head;
}
```