**数据结构**

# 1、抽象数据类型（ADT: abstract data type）

可以用数组实现动态列表，但对于内存的利用率和消耗来说它不是最有效的（根据时间复杂度得出的结论）。因为在申明数组是就固定了数组的大小，如果后续需要更大的数组，就需要重新创建数组，并且把钱一个数组的内容全部复制过去，然后再把之前的那一块内存释放。因此需要一种更加有效的方式---链表

# 2、linked-list

内存块的地址是第一个字节的地址，类似的函数指令的地址是第一句指令的地址。对于数组来说是一个连续的内存块存储在内存中。

#include <stdio.h>

int main(int arge,const char\* argv)

{

    int a = 5;

    printf("%p\n",&a);

    printf("%p\n",&a+1);

    char\* p = (char\*)&a;

    printf("%p\n",p+1);

    printf("hello world!\n");

    printf("%d\n",sizeof(a));

    return 0;

}

对于这段代码就可以看出来，一个整型的地址是第一个字节的地址。并且对于&a+1代表&a+1\*sizeof(type(a))。

链表：

struct node

{

    int data;

    struct node\* next;

};

以上整体称为一个节点。next为一个指针，指向下一节点。链表有一个头指针head指向第一个节点，最后的一个指针指向NULL。

以下代码用于验证，一个结构体所占字节，以及结构体内部是怎样实现内存分配，字节填充。

#include <stdio.h>

typedef struct \_\_attribute\_\_((packed)) node

{

    int data;

    struct node\* next;

}node1;

node1 a;

int main(int arge,const char\* argv)

{

    // int a = 5;

    printf("%d\n",sizeof(node1));

    printf("%d\n",sizeof(a.data));

    printf("%d\n",sizeof(a.next));

    printf("%d\n",sizeof(a));

    printf("%d\n",&a);

    printf("%d\n",&a+1);

    char\* p = (char\*)&a;

    printf("%p\n",p+1);

    printf("hello world!\n");

    printf("%d\n",sizeof(a));

    return 0;

}

\_\_attribute\_\_((packed))，这个函数用于设置禁止字节填充，在某些情况下，编译器会为了对齐数据而在结构体成员之间插入一些填充字节，以保证结构体的每个成员都按照某种对齐规则放置。

# 3、linked-list compared with array

对于数组和链表各有各的优势：

1）对于直接访问某个元素，数组显然要更加快捷，时间复杂度为O(1)，而对于linked-list来说时间复杂度为O(n)。

2）对于插入/删除某个特定的元素来说，时间复杂度数组和链表在不同地方时间复杂度不一样。

3）对于内存利用率来说，内存的使用情况链表更高，但也对应有额外数值用于存储下个数据的地址。但链表可变长短，操作性更高。

4）一个数据加一个指向下一个数据的指针称为节点，而只向下一个数据的指针称为链接点。

1.4、linked-list implementation in C

#include <stdio.h>

#include <stdlib.h>

typedef struct \_\_attribute\_\_((packed)) node

{

    int data;

    struct node\* next;

}node1;

int main(int arge,const char\*\* argv)

{

    node1\* a = (node1\*)calloc(sizeof(node1),1);

    a->data = 2;

    a->next = (node1\*)calloc(sizeof(node1),1);

    a->next->data = 4;

    a->next->next = (node1\*)calloc(sizeof(node1),1);

    a->next->next->data = 6;

    a->next->next->next = NULL;

    printf("%d\n",a->data);

    printf("%d\n",(\*a).data);

    printf("%d\n",a->next);

    printf("%d\n",a->next->data);

    printf("%d\n",a->next->next);

    printf("%d\n",a->next->next->data);

    printf("%d\n",a->next->next->next);

    printf("%d\n",sizeof(node1));

    free(a->next->next);

    free(a->next);

    free(a);

}

通过不断的指向下一个结构体，实现数据的存储。对于printf(“%d\n”,a->data)这里a->data是指一个数值，表示通过指针a访问结构体成员data的值，这个值的类型是int，而不是指针，等价于(\*a).data。记得释放内存，和多维数组一样，从里到外依次释放。

用while进行遍历

#include <stdio.h>

#include <stdlib.h>

typedef struct \_\_attribute\_\_((packed)) node

{

    int data;

    struct node\* next;

}node1;

int main(int arge,const char\*\* argv)

{

    node1\* a = (node1\*)calloc(sizeof(node1),1);

    a->data = 2;

    a->next = (node1\*)calloc(sizeof(node1),1);

    a->next->data = 4;

    a->next->next = (node1\*)calloc(sizeof(node1),1);

    a->next->next->data = 6;

    a->next->next->next = (node1\*)calloc(sizeof(node1),1);

    a->next->next->next->data = 0;

    a->next->next->next->next = NULL;

    node1\* temp = a;

    do

    {

        printf("%d\n",temp->data);

        // printf("%d\n",(\*temp).data);

        printf("%d\n",temp->next);

        temp = temp->next;

    }while(temp->next);

    printf("%d\n",sizeof(node1));

    free(a->next->next);

    free(a->next);

    free(a);

}

注意不要移动a指针，他是一个头指针，不能丢，否则就找不到他移动后之前的信息了。头指针永远不能变。

# 4、在链表的开头插入数据：

#include <stdio.h>

#include <stdlib.h>

#include "main.h"

typedef struct node

{

    int data;

    struct node\* next;

}node1;

node1\* head = NULL;

void insert(int x);

void print();

int main(int arge,const char\*\* argv)

{

    printf("please input number about your wanner create linked-list\n");

    int n,x;

    scanf("%d",&n);

    for(int i = 0;i<n;i++)

    {

        printf("enter the number\n");

        scanf("%d",&x);

        insert(x);

        print();

    }

    return 0;

}

void insert(int x)

{

    node1\* temp = (node1\*)malloc(sizeof(node1));

    temp->data = x;

    temp->next = head;

    head = temp;

}

void print()

{

    node1\* temp = head;

    printf("the list is: ");

    while(temp)

    {

        printf("%d ",temp->data);

        // printf("%d\n",temp->next);

        temp = temp->next;

    }

    printf("\n");

}

注意其中对于链表开始插入数据的逻辑关系。防止数据丢失。此时的head为全局变量。

修改，此时head为局部变量，但每个里面都有一个局部变量head。并且注意此时的insert函数的返回值为指针。

#include <stdio.h>

#include <stdlib.h>

#include "main.h"

typedef struct node

{

    int data;

    struct node\* next;

}node1;

node1\* insert(node1\* head,int x);

void print(node1\* head);

int main(int arge,const char\*\* argv)

{

    node1\* head = NULL;

    printf("please input number about your wanner create linked-list\n");

    int n,x;

    scanf("%d",&n);

    for(int i = 0;i<n;i++)

    {

        printf("enter the number\n");

        scanf("%d",&x);

        head = insert(head,x);

        print(head);

    }

    return 0;

}

node1\* insert(node1\* head,int x)

{

    node1\* temp = (node1\*)malloc(sizeof(node1));

    temp->data = x;

    temp->next = head;

    head = temp;

    return head;

}

void print(node1\* head)

{

    printf("the list is: ");

    while(head)

    {

        printf("%d ",head->data);

        // printf("%d\n",temp->next);

        head = head->next;

    }

    printf("\n");

}

对上面算法进一步修改，使得head为main函数中的局部变量，其他都是在这个地址上进行操作。

#include <stdio.h>

#include <stdlib.h>

#include "main.h"

typedef struct node

{

    int data;

    struct node\* next;

}node1;

void insert(node1\*\* head,int x);

void print(node1\* head);

void freememory(node1\* head);

int main(int arge,const char\*\* argv)

{

    node1\* head = NULL;

    printf("please input number about your wanner create linked-list\n");

    int n,x;

    scanf("%d",&n);

    for(int i = 0;i<n;i++)

    {

        printf("enter the number\n");

        scanf("%d",&x);

        insert(&head,x);

        print(head);

    }

    freememory(head);

    return 0;

}

void insert(node1\*\* head,int x)

{

    node1\* temp = (node1\*)malloc(sizeof(node1));

    temp->data = x;

    temp->next = \*head;

    \*head = temp;

}

void print(node1\* head)

{

    printf("the list is: ");

    while(head)

    {

        printf("%d ",head->data);

        // printf("%d\n",temp->next);

        head = head->next;

    }

    printf("\n");

}

void freememory(node1\* head)

{

    node1\* current = head;

    node1\* next;

    while(current)

    {

        next = current->next;

        free(current->next);

        current = next;

    }

}

此时head为main函数中的局部变量，在insert函数中时，传递了指向main函数局部变量head的地址，来进行处理。而对于print函数来说，不需要用head的地址来进一步处理了。当然也可以那么做，但直接传递head给print函数更加省事。最后一段为内存释放语句，采用逐级释放。

# 5、数组转链表、链表转数组输出

#include <stdio.h>

#include <stdlib.h>

typedef struct listed

{

    int val;

    struct listed \*next;

} node;

node\* mergeTwoLists(node\* list1, node\* list2)

{

    if (list1 == NULL)

        return list2;

    else if (list2 == NULL)

        return list1;

    else if (list1->val < list2->val) {

        list1->next = mergeTwoLists(list1->next, list2);

        return list1;

    } else {

        list2->next = mergeTwoLists(list2->next, list1);

        return list2;

    }

}

node\* arraytolisted(int\* a, int size)

{

    node\* list = NULL;

    node\* temp = NULL;

    for (int i = 0; i < size; i++) {

        node\* newnode = (node\*)malloc(sizeof(node));

        newnode->next = NULL;

        newnode->val = a[i];

        if (list == NULL) {

            list = newnode;

            temp = newnode;

        } else {

            temp->next = newnode;

            temp = temp->next;

        }

    }

    return list;

}

char\* listToArray(node\* list, int\* size)

{

    int count = 0;

    node\* temp = list;

    while (temp != NULL) {

        count++;

        temp = temp->next;

    }

    char\* array = (char\*)malloc(count \* sizeof(char));

    temp = list;

    for (int i = 0; i < count; i++) {

        array[i] = temp->val + '0'; //要将一个整数转换为对应的数字字符，您只需将整数值加上字符 '0' 的ASCII码。

        temp = temp->next;

    }

    \*size = count;

    return array;

}

int main(int argc, char\*\* argv) {

    int a[] = {3, 4, 5, 6};

    int b[] = {4, 5, 6, 7};

    node\* mergedList = mergeTwoLists(arraytolisted(a, sizeof(a) / sizeof(a[0])), arraytolisted(b, sizeof(b) / sizeof(b[0])));

    int size;

    char\* array = listToArray(mergedList, &size);

    printf("%.\*s\n", size, array); // Print the array as string

    // Free dynamically allocated memory

    free(array);

    return 0;

}

# 6、最差、最佳、平均时间复杂度

#include <stdio.h>

#include <stdlib.h>

#include "main.h"

int main(int argc, char\*\* argv)

{

    printf("please input an integer number: \n");

    int n, number;

    scanf("%d", &n);

    printf("please input the number you want to find: \n");

    scanf("%d", &number);

    int\* array = randomnum(n); // 保存返回的指针

    int index = find(number, array, n); // 存储查找结果

    if (index != -1) {

        printf("The number %d is found at index %d.\n", number, index);

    } else {

        printf("The number %d is not found in the array.\n", number);

    }

    free(array); // 在不再需要数组时释放内存

    return 0;

}

int\* randomnum(int n)

{

    int\* array = (int\*)malloc(sizeof(int) \* n);

    if (array == NULL) {

        printf("Memory allocation failed!\n");

        exit(1);

    }

    for (int i = 0; i < n; i++)

    {

        array[i] = i;

    }

    for (int i = n - 1; i > 0; i--)

    {

        int j = rand() % (i + 1);

        int temp = array[i];

        array[i] = array[j];

        array[j] = temp;

    }

    return array;

}

int find(int number, int\* array, int n)

{

    int i = 0;

    while (i < n && array[i] != number)

    {

        i++;

    }

    if (i < n)

    {

        return i; // 找到数字，返回索引

    }

    else

    {

        return -1; // 未找到数字，返回特殊值

    }

}

# 7、列表

main.c

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include "main.h"

#include "happy.h"

int main(int arge, char \*\*argv)

{

    mylist\* list = newMyList();

    arraytolist(list);

    printHELLO\_WORLD();

}

//数组数据存入

void arraytolist(mylist\* nums)

{

    printf("please input your wanner store data!\n");

    int data;

    while(scanf("%d",&data)==1)

    {

        AddMyListhead(nums,data);

    }

    printf("Data stored successfully!\n");

}

//构建列表

mylist\* newMyList()

{

    mylist\* nums = malloc(sizeof(mylist));

    nums->capacity = 10;

    nums->size = 0;

    nums->array = (int\*)malloc(sizeof(int)\*nums->capacity);

    nums->extendRatio = 2;

    return nums;

}

//清空列表

void delMyList(mylist\* nums)

{

    free(nums->array);

    free(nums);

}

//获取列表长度

int MyListSzie(mylist\* nums)

{

    return nums->size;

}

//获取列表容量

int MyListCapacity(mylist\* nums)

{

    return nums->capacity;

}

//访问元素

int GetMyList(mylist\* nums,int index)

{

    assert(index>=0);

    assert(index<nums->size);

    return nums->array[index];

}

//更新元素

void UpdateMyList(mylist\* nums,int index,int data)

{

    assert(index>=0);

    assert(index<nums->size);

    nums->array[index] = data;

}

//列表扩容

void extenCapacity(mylist\* nums)

{

    int newcapacity = nums->capacity \* nums->extendRatio;

    nums->array = (int\*)realloc(nums->array,sizeof(int)\*newcapacity);

}

//在尾部添加元素

void AddMyListtail(mylist\* nums,int data)

{

    if(nums->size == nums->capacity)

    {

        extenCapacity(nums);

    }

    nums->array[nums->size] = data;

    nums->size++;

}

//在中间插入元素

void insert(mylist\* nums,int index,int data)

{

    assert(index>=0);

    assert(index<nums->size);

    if(nums->size == nums->capacity)

    {

        extenCapacity(nums);

    }

    for(int i=nums->size;i>index;--i)

    {

        nums->array[i] = nums->array[i-1];

    }

    nums->array[index] = data;

    nums->size++;

}

//在头部添加元素

void AddMyListhead(mylist\* nums,int data)

{

    if(nums->size == nums->capacity)

    {

        extenCapacity(nums);

    }

    for(int i=nums->size-1;i>0;--i)

    {

        nums->array[i] = nums->array[i-1];

    }

    nums->array[0] = data;

    nums->size++;

}

//删除元素

int removeMyList(mylist\* nums, int index)

{

    assert(index>=0);

    assert(index < nums->size);

    int num = nums->array[index];

    for(int i = index;i<nums->size;i++)

    {

        nums->array[i] = nums->array[i+1];

    }

    nums->size--;

    return num;

}

//列表转数组打印

int \*listtoarray(mylist\* nums)

{

    return nums->array;

}

printHELLO\_WORLD.c

#include <stdio.h>

#include "happy.h"

void printHELLO\_WORLD()

{

    printf("hello world!\n");

}

注意对于结构体可以应用nums.array[i]，也可以用nums->array[i]，对于结构体中的数组成员的内部进行访问。

## 7.1、for循环

for循环的表达式一般如下：

for(表达式1;表达式2;表达式3)

{

表达式4;

}

执行的顺序为：

第一次循环：

首先执行表达式1（一般为初始化语句，只执行一次），再执行表达式2（条件判断语句），判断表达式2的条件，如果符合，则执行表达式4，最后执行3。 如果判断2不符合，则停止执行，且不会执行表达式3.

之后的循环：

首先执行表达式2，如果符合，继续执行表达式4，然后是3。否则停止执行，且不执行表达式3

如此往复，直到不再满足表达式2的条件。执行顺序为：1243，243，243

# 8、VScode+Cmake+gbd

对于Cmakelist.txt

cmake\_minimum\_required(VERSION 3.29.1)

project(xixi VERSION 0.1.0 LANGUAGES C CXX)

include(CTest)

enable\_testing()

#file(GLOB\_RECURSE SOURCES "src/\*.cpp")  # 自动从 src 目录读取所有 .cpp 文件

file(GLOB\_RECURSE SOURCES "src/\*.c")  # 自动从 src 目录读取所有 .c 文件

include\_directories("lib")              # 设置头文件搜索路径为 lib 目录

set(EXECUTABLE\_OUTPUT\_PATH ${CMAKE\_SOURCE\_DIR}/bin)

add\_executable(xixi ${SOURCES})         # 将所有 .cpp 文件或者 .c 文件添加到可执行文件中

set(CPACK\_PROJECT\_NAME ${PROJECT\_NAME})

set(CPACK\_PROJECT\_VERSION ${PROJECT\_VERSION})

include(CPack)

采用这个配置，C++/C通用。

注意此时的.exe文件生成位置在当前目录下的bin文件夹下。

配置文件位src存储.c文件，lib存储头文件。对于C++的使用时仅对于编译，调试文件夹进行更改。

# 9、链表实现栈

main.c

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <stdbool.h>

#include "main.h"

#include "happy.h"

int main(int arge, char \*\*argv)

{

   LinkedListStack\* node = newLinkedListStack();

   int data;

   while(scanf("%d",&data)==1)//用于检测输入数据是否为整数

   {

    push(node,data);

   }

   printf("Data stored successfully!\n");

   printf("the length of stack is %d\n",node->size);

}

//构造初始栈空间

LinkedListStack\* newLinkedListStack()

{

    LinkedListStack\* nodes = (LinkedListStack\*)malloc(sizeof(LinkedListStack));

    nodes->top = NULL;

    nodes->size = 0;

    return nodes;

}

//出栈-消除栈空间

void DelLinkedListStack(LinkedListStack\* nodes)

{

    while(nodes->top)

    {

        ListNode\* n = nodes->top->next;

        free(nodes->top);

        nodes->top = n;

    }

    free(nodes);

}

//获取栈的长度

int size(LinkedListStack\* nodes)

{

    return nodes->size;

}

//判断栈是否为空

bool isEmpty(LinkedListStack\* nodes)

{

    return nodes->size == 0;

}

//入栈

void push(LinkedListStack\* nodes, int num)

{

    ListNode\* node = (ListNode\*)malloc(sizeof(ListNode));

    node->val = num;

    node->next = nodes->top;

    nodes->top = node;

    nodes->size++;

}

//访问栈顶元素

int peek(LinkedListStack\* nodes)

{

    if(nodes->size == 0)

    printf("the stack is empty!\n");

    return nodes->top->val;

}

//出栈

int pop(LinkedListStack\* nodes)

{

    int val = nodes->top->val;

    ListNode\* node = nodes->top->next;

    nodes->top = nodes->top->next;

    free(node);

    nodes->size--;

    return val;

}

main.h

#ifndef MIAN\_H

#define MAIN\_H

typedef struct main

{

    struct main\* next;

    int val;

}ListNode;

typedef struct node1

{

    ListNode\* top;

    int size;

}LinkedListStack;

LinkedListStack\* newLinkedListStack();

void DelLinkedListStack(LinkedListStack\* nodes);

int size(LinkedListStack\* nodes);

bool isEmpty(LinkedListStack\* nodes);

void push(LinkedListStack\* nodes, int num);

int peek(LinkedListStack\* nodes);

int pop(LinkedListStack\* nodes);

#endif

栈的思想是先进后出，单链表即可实现。

10、数组实现栈

main.c

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <stdbool.h>

#include "main.h"

#include "happy.h"

int main(int arge, char \*\*argv)

{

   ArrayStack\* stack = newArrayStack();

   int data;

   while(scanf("%d",&data)==1)//用于检测输入数据是否为整数

   {

    push(stack,data);

   }

   printf("Data stored successfully!\n");

   printf("the length of stack is %d\n",stack->size);

}

//构造初始栈空间

ArrayStack\* newArrayStack()

{

    ArrayStack\* stack = (ArrayStack\*)malloc(sizeof(ArrayStack));

    stack->Array = (int\*)malloc(sizeof(int)\*MAX\_SIZE);

    stack->size = 0;

    return stack;

}

//出栈-消除栈空间

void delArrayStack(ArrayStack\* stack)

{

    free(stack->Array);//是一整块内存单元

    free(stack);

}

//获取栈的长度

int size(ArrayStack\* stack)

{

    return stack->size;

}

//判断栈是否为空

bool isEmpty(ArrayStack\* stack)

{

    return stack->size == 0;

}

//入栈

void push(ArrayStack\* stack, int num)

{

    if(stack->size == MAX\_SIZE)

    {

        printf("the stack is ovweflow\n");

        return stack->size;

    }

    stack->Array[stack->size] = num;

    stack->size++;

}

//访问栈顶元素

int peek(ArrayStack\* stack)

{

    int val;

    if(stack->size == 0)

    printf("the stack is empty!\n");

    return stack->Array[stack->size - 1];

}

//出栈

int pop(ArrayStack\* stack)

{

    int val = peek(stack);

    stack->size--;

    return val;

}

main.h

#ifndef MIAN\_H

#define MAIN\_H

#define MAX\_SIZE 10

typedef struct k

{

    int \*Array;

    int size;

}ArrayStack;

ArrayStack\* newArrayStack();

void delArrayStack(ArrayStack\* stack);

int size(ArrayStack\* stack);

bool isEmpty(ArrayStack\* stack);

void push(ArrayStack\* stack, int num);

int peek(ArrayStack\* stack);

int pop(ArrayStack\* stack);

#endif

数组实现栈在数组底部进行数据的增删。

# 10、链表实现队列

main.c

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <stdbool.h>

#include "main.h"

#include "happy.h"

int main(int arge, char \*\*argv)

{

    queue\* queuelist = newQueueList();

    int data;

    while(scanf("%d",&data)==1)

    {

        push(queuelist,data);

    }

    printf("inputing is successful!\n");

    printf("the length of queue is %d\n",size(queuelist));

    printQueueList(queuelist);

}

//创建队列

queue\* newQueueList()

{

    queue\* queuelist = (queue\*)malloc(sizeof(queue));//开辟队列空间

    queuelist->front = NULL;

    queuelist->rear = NULL;

    queuelist->size = 0;

    printf("queue is succeed in creating!\n");

    return queuelist;

}

//析构队列

void delQueueList(queue\* queuelist)

{

    while(queuelist->front==NULL)

    {

        Listnode \*tmp = queuelist->front;

        queuelist->front = queuelist->front->next;

        free(tmp);

    }

    free(queuelist);

}

//入队

void push(queue\* queuelist,int num)

{

    Listnode\* node = (Listnode\*)malloc(sizeof(Listnode));

    node->val = num;

    node->next = NULL;

    if(queuelist->front == NULL)

    {

        queuelist->front = node;

        queuelist->rear = node;

    }

    else

    {

        queuelist->rear->next = node;

        queuelist->rear = node;

    }

    queuelist->size++;

}

//队列长度

int size(queue\* queuelist)

{

    return queuelist->size;

}

//判断队列是否为空

bool isEmpty(queue\* queuelist)

{

    return queuelist->size == 0;

}

//获取队列首元素

int peek(queue\* queuelist)

{

    assert(size(queuelist)&&queuelist->front);

    return queuelist->front->val;

}

//出队

int pop(queue\* queuelist)

{

    int num = peek(queuelist);

    Listnode\* node = queuelist->front;

    queuelist->front = queuelist->front->next;

    free(node);

    queuelist->size--;

    return num;

}

//打印队列

void printQueueList(queue\* queuelist)

{

    Listnode\* node = queuelist->front;

    for(int i=0;i<queuelist->size;i++)

    {

        printf("%d\n",node->val);

        node = node->next;

    }

}

main.h

#ifndef MIAN\_H

#define MAIN\_H

typedef struct list

{

    struct list\* next;

    int val;

}Listnode;

typedef struct queue

{

    Listnode \*front,\*rear;

    int size;

}queue;

queue\* newQueueList();

void delQueueList(queue\* queuelist);

void push(queue\* queuelist,int num);

int size(queue\* queuelist);

bool isEmpty(queue\* queuelist);

int peek(queue\* queuelist);

int pop(queue\* queuelist);

void printQueueList(queue\* queuelist);

#endif

对于队列是一个先进先出的结构，因此对于结构上来说具有两个指针。规定队列从头部删除尾部进入。

# 11、数组实现队列

main.c

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <stdbool.h>

#include "main.h"

#include "happy.h"

int main(int argc, char \*\*argv)

{

    int data;

    queue\* queuearray = newqQueueArray();

    printf("Please input your numbers, press Ctrl+D to finish:\n");

    while(scanf("%d", &data) == 1)

    {

        push(queuearray, data);

    }

    printf("Data is inputted successfully\n");

    printQueueArray(queuearray);

    DelQueueArray(queuearray);

    return 0;

}

// 创建队列

queue\* newqQueueArray()

{

    queue\* queuearray = (queue\*)malloc(sizeof(queue));

    queuearray->expand = 2;

    queuearray->quecapacity = 10;

    queuearray->array = (int\*)malloc(sizeof(int) \* queuearray->quecapacity);

    queuearray->front = 0;

    queuearray->queSzie = 0;

    queuearray->size = 0;

    printf("Queue is successfully created!\n");

    return queuearray;

}

// 析构队列

void DelQueueArray(queue\* queuearray)

{

    free(queuearray->array);

    free(queuearray);

}

// 队列容量变化

void ExpandQueueArray(queue\* queuearray)

{

    queuearray->quecapacity = queuearray->quecapacity \* queuearray->expand;

    queuearray->array = (int\*)realloc(queuearray->array, sizeof(int) \* queuearray->quecapacity);

    printf("Expansion successful, current capacity: %d\n", queuearray->quecapacity);

}

// 入队

void push(queue\* queuearray, int num)

{

    if(queuearray->size == queuearray->quecapacity)

    {

        printf("Array room overflow! Creating new room...\n");

        ExpandQueueArray(queuearray);

    }

    int rear = (queuearray->front + queuearray->size) % queuearray->quecapacity;

    queuearray->array[rear] = num;

    queuearray->size++;

}

// 队列长度

int Queuesize(queue\* queuearray)

{

    return queuearray->size;

}

// 判断队列是否为空

bool isEmpty(queue\* queuearray)

{

    return queuearray->size == 0;

}

// 获取队列首元素

int peek(queue\* queuearray)

{

    assert(queuearray->size);

    return queuearray->array[queuearray->front];

}

// 出队

int pop(queue\* queuearray)

{

    int val = peek(queuearray);

    queuearray->front = (queuearray->front + 1) % queuearray->quecapacity;

    queuearray->size--;

    return val;

}

// 打印队列

void printQueueArray(queue\* queuearray)

{

    printf("Printing the queue:\n");

    for(int i = 0; i < queuearray->size; i++)

    {

        printf("%d\n", queuearray->array[(queuearray->front + i) % queuearray->quecapacity]);

    }

}

main.h

#ifndef MIAN\_H

#define MAIN\_H

typedef struct queue

{

    int\* array;

    int front;

    int queSzie;

    int quecapacity;

    int expand;

    int size;

}queue;

queue\* newqQueueArray();

void DelQueueArray(queue\* queuearray);

void ExpandQueueArray(queue\* queuearray);

void push(queue\* queuearray,int num);

int Queuesize(queue\* queuearray);

bool isEmpty(queue\* queuearray);

int peek(queue\* queuearray);

int pop(queue\* queuearray);

void printQueueArray(queue\* queuearray);

#endif

对于数组实现队列，从数组顶部出从底部入，因此会存在内从的消耗，所以采用循环数组。此时需要注意循环执行所采用的取余数的方式，不能丢失数据，遂于数据的输出也需要对索引采用取余实现，防止出现数据乱序。

# 12、双向链表实现双向队列

mian.c

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <stdbool.h>

#include "main.h"

#include "happy.h"

int main(int argc, char \*\*argv)

{

    int data;

    LinkListQueue\* deque =  newLinkListQueue();

    printf("please input you wanner number! ctrl+d stop inputting!\n");

    while(scanf("%d",&data)==1)

    {

        push\_first(deque,data);

    }

    printf("the length od queue is %d\n",deque->size);

    printQueue(deque);

    delLinkListQueue(deque);

}

/\* 节点构造函数 \*/

DoubldeListNode\* newDoubldeListNode(int num)

{

    DoubldeListNode\* newnode = (DoubldeListNode\*)malloc(sizeof(DoubldeListNode));

    newnode->next = NULL;

    newnode->prev = NULL;

    newnode->val = num;

    return newnode;

}

/\* 节点析构函数 \*/

void delDoubldeListNode(DoubldeListNode\* newnode)

{

    free(newnode);

}

/\* 构造函数 \*/

LinkListQueue\* newLinkListQueue()

{

    LinkListQueue\* deque = (LinkListQueue\*)malloc(sizeof(LinkListQueue));

    deque->front = NULL;

    deque->rear = NULL;

    deque->size = 0;

    printf("Queue is successful to creat!\n");

    return deque;

}

/\* 析构函数 \*/

void delLinkListQueue(LinkListQueue\* newqueue)

{

    int i = 0;

    while(newqueue->front!=NULL)

    {

        DoubldeListNode\* tmp = newqueue->front;

        newqueue->front = newqueue->front->next;

        free(tmp);

    }

    free(newqueue);

}

/\* 获取队列的长度 \*/

int sizeQueue(LinkListQueue\* newqueue)

{

    return newqueue->size;

}

/\* 判断队列是否为空 \*/

bool isEmpty(LinkListQueue\* newqueue)

{

    return newqueue->size == 0;

}

/\* 入队 \*/

void push(LinkListQueue\* newqueue,int num,bool isfront)

{

    DoubldeListNode\* node = newDoubldeListNode(num);

    if(isEmpty(newqueue))

    {

        newqueue->front = newqueue->rear = node;

    }

    else if(isfront)

    {

        newqueue->front->prev = node;

        node->next = newqueue->front;

        newqueue->front = node;

    }

    else

    {

        newqueue->rear->next = node;

        node->prev = newqueue->rear;

        newqueue->rear = node;

    }

    newqueue->size++;

}

/\* 队首入队 \*/

void push\_first(LinkListQueue\* newqueue, int num)

{

    push(newqueue,num,true);

}

/\* 队尾入队 \*/

void push\_last(LinkListQueue\* newqueue, int num,bool isfront)

{

    push(newqueue,num,false);

}

/\* 访问队首元素 \*/

int peek\_first(LinkListQueue\* newqueue)

{

    return newqueue->front->val;

}

/\* 访问队尾元素 \*/

int peek\_last(LinkListQueue\* newqueue)

{

    return newqueue->rear->val;

}

/\* 出队 \*/

int pop(LinkListQueue\* newqueue,bool isfirst)

{

    int val;

    DoubldeListNode\* node;

    if(isfirst)

    {

        val = peek\_first(newqueue);

        node = newqueue->front;

        newqueue->front = newqueue->front->next;

        newqueue->front->prev = NULL;

    }

    else

    {

        val = peek\_last(newqueue);

        node = newqueue->rear;

        newqueue->rear = newqueue->rear->prev;

        newqueue->rear->next = NULL;

    }

    delDoubldeListNode(node);

    newqueue->size--;

    return val;

}

/\* 队首出队 \*/

int pop\_first(LinkListQueue\* newqueue)

{

    return pop(newqueue,true);

}

/\* 队尾出队 \*/

int pop\_last(LinkListQueue\* newqueue)

{

    return pop(newqueue,false);

}

/\* 打印队列 \*/

void printQueue(LinkListQueue\* newqueue)

{

    DoubldeListNode\* node = newqueue->rear;

    for(int i = 0;i<newqueue->size;i++)

    {

        printf("%d\n",node->val);

        node = node->prev;

    }

}

main.h

#ifndef MIAN\_H

#define MAIN\_H

/\* 双向链表节点 \*/

typedef struct DoubldeListNode

{

    struct DoubldeListNode \*next;

    struct DoubldeListNode \*prev;

    int val;

}DoubldeListNode;

/\* 基于双向链表实现的双向队列 \*/

typedef struct LinkListQueue

{

    DoubldeListNode \*front,\*rear;

    int size;

}LinkListQueue;

DoubldeListNode\* newDoubldeListNode(int num);

void delDoubldeListNode(DoubldeListNode\* newnode);

LinkListQueue\* newLinkListQueue();

void delLinkListQueue(LinkListQueue\* newqueue);

int sizeQueue(LinkListQueue\* newqueue);

bool isEmpty(LinkListQueue\* newqueue);

void push(LinkListQueue\* newqueue,int num,bool isfront);

void push\_first(LinkListQueue\* newqueue, int num);

void push\_last(LinkListQueue\* newqueue, int num,bool isfront);

int peek\_first(LinkListQueue\* newqueue);

int peek\_last(LinkListQueue\* newqueue);

int pop(LinkListQueue\* newqueue,bool isfirst);

int pop\_first(LinkListQueue\* newqueue);

int pop\_last(LinkListQueue\* newqueue);

void printQueue(LinkListQueue\* newqueue);

#endif

巧妙地运用布尔值，对于结构体的传递，注意引用。要分节点结构体和队列结构体。

13、数组实现双向队列

mian.c

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include "main.h"

int main(int arge,char\*\* argv)

{

    int data;

    arrayQueue\* queue = newArrayQueue();

    while (scanf("%d",&data) == 1)

    {

        //push\_first(queue,data);

        push\_last(queue,data);

    }

    printf("successfully!\n");

    printf("the curret memory is %d and the size of queue is %d!\n",queue->quecapacity,queue->quesize);

    printQueue(queue);

    delArrayQueue(queue);

    return 0;

}

//构造初始化双向队列

arrayQueue\* newArrayQueue()

{

    printf("the queue is successful to crate!\n");

    arrayQueue\* queue = (arrayQueue\*)malloc(sizeof(arrayQueue));

    queue->quecapacity = 10;

    queue->queexpand = 2;

    queue->array = (int\*)malloc(sizeof(int)\*queue->quecapacity);

    queue->front = queue->quesize = 0;

    printf("the curret memory is %d and the size of queue is %d!\n",queue->quecapacity,queue->quesize);

    return queue;

}

//析构双向队列

void delArrayQueue(arrayQueue\* queue)

{

    free(queue->array);

    free(queue);

}

//从头部入队

void push\_first(arrayQueue\* queue,int num)

{

    if(queue->quesize == queue->quecapacity)

    {

        printf("Queue memory overflow! The expanatition has been carried!\n");

        printf("the current memory is %d\n",queue->quecapacity);

        expandQueue(queue);

    }

    queue->array[calculation(queue , queue->front - 1)] = num;

    queue->front = calculation(queue , queue->front - 1);

    queue->quesize++;

}

//从尾部入队

void push\_last(arrayQueue\* queue, int num)

{

    if(queue->quesize == queue->quecapacity)

    {

        printf("Queue memory overflow! The expanatition has been carried!\n");

        printf("the current memory is %d\n",queue->quecapacity);

        expandQueue(queue);

    }

    queue->array[calculation(queue , queue->front + queue->quesize)] = num;

    queue->quesize++;

}

//扩充容量

void expandQueue(arrayQueue\* queue)

{

    queue->quecapacity \*= queue->queexpand;

    queue->array = (int\*)realloc(queue->array,sizeof(int)\*queue->quecapacity);

    printf("enlager the memory successfully!\n");

    printf("the curret memory is %d\n",queue->quecapacity);

}

//双向列表索引计算

int calculation(arrayQueue\* queue,int current)

{

    int rear = (current + queue->quecapacity) % queue->quecapacity;

    return rear;

}

//获取队列长度

int Qsize(arrayQueue\* queue)

{

    return queue->quesize;

}

//判断队列是否为空

bool isEmpty(arrayQueue\* queue)

{

    return Qsize(queue) == 0;

}

//访问对首元素

int peek\_first(arrayQueue\* queue)

{

    return queue->array[queue->front];

}

//访问队尾元素

int peek\_last(arrayQueue\* queue)

{

    return queue->array[calculation(queue,queue->front + queue->quesize) - 1];

}

//队首元素出队

int pop\_first(arrayQueue\* queue)

{

    int val = queue->array[queue->front];

    queue->front = calculation(queue,queue->front + 1);

    queue->quesize--;

    return val;

}

//队尾元素出队

int pop\_last(arrayQueue\* queue)

{

    int val = peek\_last(queue);

    queue->quesize--;

    return val;

}

//队列元素打印

void printQueue(arrayQueue\* queue)

{

    for(int i=0;i<queue->quesize;i++)

    {

        printf("%d\n",queue->array[calculation(queue,queue->front+i)]);

    }

}

main.h

#ifndef MIAN\_H

#define MAIN\_H

typedef struct arrayQueue

{

    int \*array;

    int front;

    int quesize;

    int quecapacity;

    int queexpand;

}arrayQueue;

arrayQueue\* newArrayQueue();

void delArrayQueue(arrayQueue\* queue);

void push\_first(arrayQueue\* queue,int num);

void push\_last(arrayQueue\* queue, int num);

void expandQueue(arrayQueue\* queue);

int calculation(arrayQueue\* queue,int current);

int Qsize(arrayQueue\* queue);

bool isEmpty(arrayQueue\* queue);

int peek\_first(arrayQueue\* queue);

int peek\_last(arrayQueue\* queue);

int pop\_first(arrayQueue\* queue);

int pop\_last(arrayQueue\* queue);

void printQueue(arrayQueue\* queue);

#endif

对于数组实现双向队列整体难度比双向链表实现要简单的多，但要注意索引计算。索引更新。

# 13、数组实现哈希表

main.c

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

#include "main.h"

int main(int arge,char\*\* argv)

{

    int key;

    char\* val;

    ArrayHashMap\* hamp = newArrayHashMap();

    printf("The hash map is successfully created!\n");

    printf("Please input your key-value pairs ('key -> value'):\n");

    while (scanf("%d -> %s", &key, val) == 2)//scanf 函数返回成功读取的输入项的数量

    {

        put(hamp,key,val);

    }

    printf("Completely\n");

    print(hamp);

    return 0;

}

//hashFunction

int hashFunc(ArrayHashMap\* hamp,int key)

{

    return key%hamp->HASHTABLE\_CABACITY;

}

//创建哈希函数基于数组

ArrayHashMap\* newArrayHashMap()

{

    ArrayHashMap\* hamp = (ArrayHashMap\*)malloc(sizeof(ArrayHashMap));

    hamp->HASHTABLE\_CABACITY = 100;

    hamp->buckets = (paris\*\*)malloc(sizeof(paris\*)\*hamp->HASHTABLE\_CABACITY);//hamp->bucket[i]中存放的是指向paris结构体的指针

    for(int i=0;i<hamp->HASHTABLE\_CABACITY;i++)

    {

        hamp->buckets[i] = NULL;

    }

    return hamp;

}

//析构函数

void delArrayHashMap(ArrayHashMap\* hamp)

{

    int i = 0;

    for(i =0;i<hamp->HASHTABLE\_CABACITY;i++)

    {

        if(hamp->buckets[i] != NULL)

        {

            free(hamp->buckets[i]->vals);

            free(hamp->buckets[i]);

        }

    }

    free(hamp);

}

//添加操作

void put(ArrayHashMap\* hamp,const int key,const char \*val)

{

    paris\* pari = (paris\*)malloc(sizeof(paris));

    pari->key = key;

    pari->vals = (char\*)malloc(strlen(val) + 1);

    strcpy(pari->vals,val);

    int index = hashFunc(hamp,key);

    hamp->buckets[index] = pari;

    printf("Successfully added key-value pair!\n");

}

//删除操作

void removeItem(ArrayHashMap\* hamp,const int key)

{

    int index = hashFunc(hamp,key);

    free(hamp->buckets[index]->vals);

    free(hamp->buckets[index]);

    hamp->buckets[index] = NULL;

}

//获取所有键值对

void pairSet(ArrayHashMap\* hamp,MapSet\* set)

{

    int i = 0;

    int total = 0,index = 0;

    paris\* entries;

    for(i = 0;i<hamp->HASHTABLE\_CABACITY;i++)

    {

        if(hamp->buckets[i] != NULL)

        {

            total++;

        }

    }

    entries = (paris\*)malloc(sizeof(paris)\*total);

    for(i = 0;i<hamp->HASHTABLE\_CABACITY;i++)

    {

        if(hamp->buckets[i] != NULL)

        {

            entries[index].vals = (char\*)malloc(strlen(hamp->buckets[i]->vals)+1);

            entries[index].key = hamp->buckets[i]->key;

            strcpy(entries[index].vals,hamp->buckets[i]->vals);

            index++;

        }

    }

    set->set = (char\*)entries;

    set->length = total;

}

//获取所有键

void keySet(ArrayHashMap\* hamp , MapSet\* set)

{

    int i;

    int\* keys;

    int total =0;

    int index = 0;

    for(i=0;i<hamp->HASHTABLE\_CABACITY;i++)

    {

        if(hamp->buckets[i] != NULL)

        {

            total++;

        }

    }

    keys = (int\*)malloc(sizeof(int)\*total);

    for(i = 0;i<hamp->HASHTABLE\_CABACITY;i++)

    {

        if(hamp->buckets[i] != NULL)

        {

            keys[index] = hamp->buckets[i]->key;

            index++;

        }

    }

    set->length = total;

    set->set = (char\*)keys;//set的类型为char\*，这其中涉及到类型的隐式转换

}

//获取所有值

void valueSet(ArrayHashMap\* hamp,MapSet\* set)

{

    int i,index=0;

    int total = 0;

    char\*\* vals;

    for(i=0;i<hamp->HASHTABLE\_CABACITY;i++)

    {

        if(hamp->buckets[i] != NULL)

        {

            total++;

        }

    }

    vals = (char\*\*)malloc(sizeof(char\*)\*total);

    for(i=0;i<hamp->HASHTABLE\_CABACITY;i++)

    {

        if(hamp->buckets[i] != NULL)

        {

            vals[index] = hamp->buckets[i]->vals;

            index++;

        }

    }

    set->length = total;

    set->set = (char\*)vals;

}

//打印哈希表

// 打印哈希表

void print(ArrayHashMap\* hamp)

{

    MapSet\* set = (MapSet\*)malloc(sizeof(MapSet)); // 为 set 结构体分配内存

    pairSet(hamp, set); // 调用 pairSet 函数

    paris \*entries = (paris\*)set->set;

    for (int i = 0; i < set->length; i++)

    {

        printf("%d -> %s\n", entries[i].key, entries[i].vals);

    }

    free(set->set); // 释放内存

    free(set); // 释放内存

}

main.h

#ifndef MIAN\_H

#define MAIN\_H

typedef struct paris

{

    char\* vals;

    int key;

}paris;

typedef struct set

{

    char\* set;

    int length;

}MapSet;

typedef struct

{

    int HASHTABLE\_CABACITY;

    paris\*\* buckets;

} ArrayHashMap;

int hashFunc(ArrayHashMap\* hamp,int key);

ArrayHashMap\* newArrayHashMap();

void delArrayHashMap(ArrayHashMap\* hamp);

void put(ArrayHashMap\* hamp,const int key,const char \*val);

void removeItem(ArrayHashMap\* hamp,const int key);

void pairSet(ArrayHashMap\* hamp,MapSet\* set);

void keySet(ArrayHashMap\* hamp , MapSet\* set);

void valueSet(ArrayHashMap\* hamp,MapSet\* set);

void print(ArrayHashMap\* hamp);

#endif

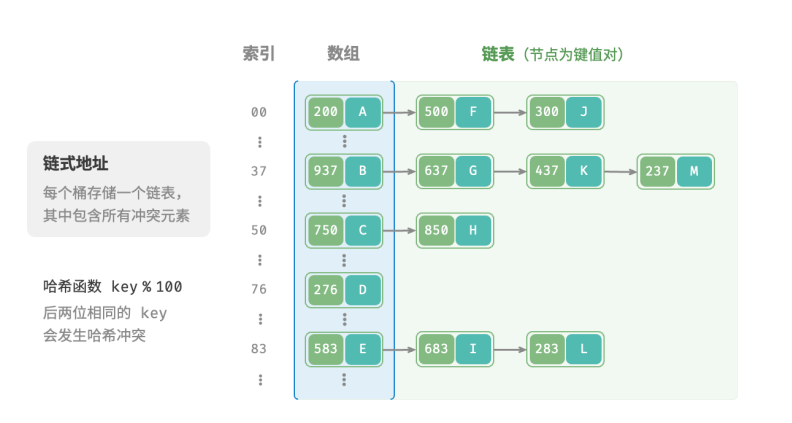
动态数组实现哈希表，首先要设置每个bucket内部存储数据的键值对paris，然后将键值对放进bucket动态数组中，设置动态数组可以实现当出现哈希冲突时，对哈希表进行扩容，类似于数组扩容，哈希表扩容需将所有键值对从原哈希表迁移至新哈希表，非常耗时；并且由于哈希表容量 capacity 改变，我们需要通过哈希函数来重新计算所有键值对的存储位置，这进一步增加了扩容过程的计算 开销。为此，编程语言通常会预留足够大的哈希表容量，防止频繁扩容。

注意：对于scanf函数类属于stdio.h库函数，他有返回值即一次所读取的数据量。

# 14、哈希表的结构改良方法

主要包括“链式地址”和“开放寻址”。

## 14.1、链式地址



main.c

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include "main.h"

#include <stdbool.h>

bool flag = false;

int main()

{

    int key;

    char\* vals;

    HashMapChaining\* hashMap = newHashMapChaining();

    while(scanf("%d->%s",&key,vals)==2)

    {

        if (addHashChainingMap(hashMap, key, vals))

        {

            printf("Adding successfully!\n");

        }

    }

    printf("completely!\n");

    pringHashChaining(hashMap);

    return 0;

}

/\* 构造函数 \*/

HashMapChaining\* newHashMapChaining()

{

    HashMapChaining\* hashMap = (HashMapChaining\*)malloc(sizeof(HashMapChaining));

    hashMap->HASHTABLE\_CABACITY = 4;

    hashMap->size = 0;

    hashMap->expantion = 2;

    hashMap->buckets = (Node\*\*)malloc(sizeof(Node\*)\*hashMap->HASHTABLE\_CABACITY);

    hashMap->loadThres = 2.0/3.0;

    for(int i=0;i<hashMap->HASHTABLE\_CABACITY;i++)

    {

        hashMap->buckets[i] = NULL;

    }

    printf("hashChaining is created successfully!\n");

    return hashMap;

}

/\* 析构函数 \*/

void delHashChaining(HashMapChaining\* hashMap)

{

    for(int i=0;i<hashMap->HASHTABLE\_CABACITY;i++)

    {

        Node\* cur = hashMap->buckets[i];

        while(cur)

        {

            Node\* temp = cur;

            cur = cur->next;

            free(temp->pari->vals);

            free(temp->pari);

            free(temp);

        }

    }

    free(hashMap);

}

/\* 哈希函数 \*/

int hashFunction(HashMapChaining\* hashMap,const int key)

{

    return key%hashMap->HASHTABLE\_CABACITY;

}

/\* 负载因子 \*/

double loadFactor(HashMapChaining\* hashMap)

{

    double rateFactor = (double)(hashMap->size) / (double)hashMap->HASHTABLE\_CABACITY;

    if(rateFactor > hashMap->loadThres)

    {

        flag = true;

    }

    return rateFactor > hashMap->loadThres;

}

/\* 查询操作 \*/

char\* getHashChainingMap(HashMapChaining\* hashMap,const int key)

{

    int index = hashFunction(hashMap,key);

    Node\* cur = hashMap->buckets[index];

    while(cur)

    {

        if(cur->pari->key == key)

        {

            return cur->pari->vals;

        }

        cur = cur->next;

    }

    return "Null";

}

/\* 添加操作 \*/

bool addHashChainingMap(HashMapChaining\* hashMap, const int key, const char\* val)

{

    int index = hashFunction(hashMap, key);

    if (loadFactor(hashMap))

    {

        extend(hashMap);

    }

    Node\* newNode = (Node\*)malloc(sizeof(Node));

    if (!newNode) {

        return false; // 内存分配失败

    }

    newNode->pari = (paris\*)malloc(sizeof(paris));

    if (!newNode->pari) {

        free(newNode); // 释放已分配的节点内存

        return false; // 内存分配失败

    }

    newNode->pari->key = key;

    newNode->pari->vals = (char\*)malloc(strlen(val) + 1);

    if (!newNode->pari->vals) {

        free(newNode->pari); // 释放已分配的键值对内存

        free(newNode); // 释放已分配的节点内存

        return false; // 内存分配失败

    }

    strcpy(newNode->pari->vals, val);

    newNode->next = NULL;

    if (hashMap->buckets[index] == NULL)

    {

        hashMap->buckets[index] = newNode;

    }

    else

    {

        Node\* cur = hashMap->buckets[index];

        hashMap->buckets[index] = newNode; // 新节点作为头节点

        newNode->next = cur;

    }

    hashMap->size++;

    return true; // 添加成功

}

/\* 扩容哈希表 \*/

void extend(HashMapChaining\* hashMap)

{

    int oldcapacity = hashMap->HASHTABLE\_CABACITY;

    Node\*\* oldHashchaining = hashMap->buckets;

    hashMap->HASHTABLE\_CABACITY \*= hashMap->expantion;

    hashMap->buckets = (Node\*\*)malloc(sizeof(Node\*) \* hashMap->HASHTABLE\_CABACITY);

    for(int i=0;i< hashMap->HASHTABLE\_CABACITY;i++)

    {

        hashMap->buckets[i] = NULL;

    }

    hashMap->size = 0;

    for(int j=0;j<oldcapacity;j++)

    {

        Node\* cur = oldHashchaining[j];

        while(cur)

        {

            Node\* temp = cur;

            addHashChainingMap(hashMap,cur->pari->key,cur->pari->vals);

            cur = cur->next;

            free(temp->pari->vals);

            free(temp->pari);

            free(temp);

        }

    }

    free(oldHashchaining);

    flag = false;

    printf("extending the hashchaining successfully!\n");

}

/\* 删除操作 \*/

void delHashChainingFactor(HashMapChaining\* hashMap,const int key)

{

    int index = hashFunction(hashMap,key);

    Node\* cur = hashMap->buckets[index];

    Node \*pre = NULL;

    while (cur)

    {

        if(cur->pari->key == key)

        {

            if(pre)

            {

                pre->next = cur->next;

            }

            else

            {

                hashMap->buckets[index] = cur->next;

            }

            free(cur->pari->vals);

            free(cur->pari);

            free(cur);

            return ;

        }

        else

        {

            pre = cur;

            cur = cur->next;

        }

    }

    hashMap->size--;

}

/\* 打印哈希表 \*/

void pringHashChaining(HashMapChaining\* hashMap)

{

    Node\*\* cur = hashMap->buckets;

    for(int i=0;i<hashMap->HASHTABLE\_CABACITY;i++)

    {

        Node\* temp = cur[i];

        while(temp)

        {

            printf("the buckets [%d] has these key -> values:\n",i);

            printf("\n%d -> %s\n",temp->pari->key,temp->pari->vals);

            temp = temp->next;

        }

    }

    delHashChaining(hashMap);

}

mian.h

#include <stdbool.h>

#ifndef MIAN\_H

#define MAIN\_H

//键值对

typedef struct paris

{

    char\* vals;

    int key;

}paris;

//节点

typedef struct node

{

    paris\* pari;

    struct node\* next;

}Node;

//哈希表

typedef struct

{

    int HASHTABLE\_CABACITY;

    Node\*\* buckets;

    int size;

    int expantion;

    double loadThres; // 触发扩容的负载因子阈值

}HashMapChaining;

HashMapChaining\* newHashMapChaining();

void delHashChaining(HashMapChaining\* hashMap);

int hashFunction(HashMapChaining\* hashMap,const int key);

double loadFactor(HashMapChaining\* hashMap);

char\* getHashChainingMap(HashMapChaining\* hashMap,const int key);

bool addHashChainingMap(HashMapChaining\* hashMap, const int key, const char\* val);

void extend(HashMapChaining\* hashMap);

void delHashChainingFactor(HashMapChaining\* hashMap,const int key);

void pringHashChaining(HashMapChaining\* hashMap);

#endif

在实现逻辑期间注意对于指针的应用，二级指针和一级指针以及不同类型指针之间的变换。建议实现过程中，多画图实现对于逻辑思路的理清。