# 哈希表

# 【基本概念】

## 一：什么是哈希表

实质上是数组，每个元素看成bucket(桶)。

其数组的下标是key经过hash算法得出的hashvalue 再对bucketMaxNum取模得到的。

而取模可以写成keyToBucketIndex = hashFunc(key) % bucketMaxNum,由于bucketMaxNum 常常是2的幂次方，因此

取模等价于keyToBucketIndex = hashFunc(key) & (bucketMaxNum - 1).

## 二：哈希函数

有很多优秀的hash算法，需要效率高，并且冲突小。

分为整形散列函数，子符串散列函数等等。

## 三：解决冲突的方法

### 开放定地址法：

### 拉链法：

比较常用，对于不同key但算出的bucketIndex相同的2个对象需要放到同一个桶的链表中。数据结构其实就是bucket数组，每个bucket其实都是带有链表特性的节点。

## 四：扩容

当同一个桶中的元素过多，会导致哈希表的查找性能降低，因此需要进行对桶进行扩容，并且对原先的hash表中的所有元素重新进行散列存储。

## 五：排序

常规情况下，遍历桶数组时，对桶的链表进行遍历是可以的，但是如何对元素排序，其实就需要另外的链表属性对所有添加到hash表中的数据进行串起来。

# 【知识补充】

## 一：归并排序

### （1）数组的归并排序

对一个乱序的数组进行排序。例如

#include<stdio.h>

#include<stdlib.h>

void MergeSortVec(int\* vec, int vec1Start, int vec1End, int vec2Start, int vec2End)

{

int dstIndex = 0;

int vec1StartIndex = 0;

int vec2StartIndex = 0;

int vec1Size = vec1End - vec1Start + 1;

int vec2Size = vec2End - vec2Start + 1;

int\* vec1 = (int\*)calloc(vec1Size, sizeof(int));

int\* vec2 = (int\*)calloc(vec2Size, sizeof(int));

int\* dst = (int\*)calloc(vec1Size + vec2Size, sizeof(int));

// 先拷贝到2个数组中

memcpy(vec1, &vec[vec1Start], vec1Size \* sizeof(int));

memcpy(vec2, &vec[vec2Start], vec2Size \* sizeof(int));

while (vec1StartIndex < vec1Size && vec2StartIndex < vec2Size) {

if (vec1[vec1StartIndex] < vec2[vec2StartIndex]) {

dst[dstIndex++] = vec1[vec1StartIndex++];

} else if (vec1[vec1StartIndex] > vec2[vec2StartIndex]) {

dst[dstIndex++] = vec2[vec2StartIndex++];

} else {

dst[dstIndex++] = vec1[vec1StartIndex++];

}

}

while (vec1StartIndex < vec1Size) {

dst[dstIndex++] = vec1[vec1StartIndex++];

}

while (vec2StartIndex < vec2Size) {

dst[dstIndex++] = vec2[vec2StartIndex++];

}

memcpy(&vec[vec1Start], dst, (vec1Size + vec2Size) \* sizeof(int));

free(dst);

free(vec1);

free(vec2);

}

void VecMergeSort(int\* vec, int startIndex, int endIndex)

{

int mid;

// 表明最多只有一个元素了

if (endIndex <= startIndex) {

return;

}

// 1.分割

mid = startIndex + (endIndex - startIndex) / 2;

VecMergeSort(vec, startIndex, mid);

VecMergeSort(vec, mid + 1, endIndex);

// 2.进行合并

MergeSortVec(vec, startIndex, mid, mid +1, endIndex);

}

int main()

{

int order[] = {2,1,4,3,0,8,6,4,2,1};

int orderSize = sizeof(order) / sizeof(order[0]);

int i;

VecMergeSort(order, 0, orderSize - 1);

for (i = 0; i < orderSize; i++) {

printf("%d ",order[i]);

}

}

### （2）链表的归并排序

typedef int (\*CmpFuncInt)(int, int);

// 用户的结构体，ListNode listNode这个是封装了的，拿来组织用户数据成链表

typedef struct {

    int score;

    char name[10];

    CmpFuncStr myCmpStr;

    CmpFuncInt myCmpInt;

    ListNode listNode;

}Student;

int MyIntCmp(int a, int b)

{

    if (a > b) {

        return 1;

    } else if (a < b) {

        return -1;

    } else {

        return 0;

    }

}

/\*

help是辅助头节点,记得释放，相当于数组归并中的的第三个数组;

help\_cur相当于下标;

合并完返回辅助头结点的下一个结点即可

\*/

ListNode\* MergeSortList(ListNode\* list1, ListNode\* list2)

{

    ListNode\* help = NULL;

    ListNode\* help\_cur = NULL;

    Student \*stu1, \*stu2;

    /\*\* help是list1的辅助头节点,记得释放\*/

    ListCreate(help);

    help\_cur = help;

    while (list1 != NULL && list2 != NULL) {

        stu1 = ListGetEntry(list1, Student, listNode);

        stu2 = ListGetEntry(list2, Student, listNode);

        // list1 < list2

        if (stu1->myCmpInt(stu1->score, stu2->score) <= 0) {

            help\_cur->next = list1;

            help\_cur = help\_cur->next;

            list1 = list1->next;

        } else {

            help\_cur->next = list2;

            help\_cur = help\_cur->next;

            list2 = list2->next;

        }

    }

    /\*\*这里用if就可以了\*/

    if (list1 != NULL) {

        help\_cur->next = list1;

    }

    if (list2 != NULL) {

        help\_cur->next = list2;

    }

    help\_cur = help->next;

    free(help);

    help = NULL;

    return help\_cur;

}

ListNode\* ListMergeSort(ListNode\* head)

{

    ListNode\* fast = NULL;

    ListNode\* slow = NULL;

    ListNode\* leftNode = NULL;

    ListNode\* rightNode = NULL;

    /\*\* 空链表或者只有一个节点的链表\*/

    if (head == NULL || head->next == NULL) {

        return head;

}

/\*1.快慢指针取中间结点\*/

    fast = head->next;/\*\*多走一个，否则取不到中间，和数组归并的差异\*/

    slow = head;

    while (fast != NULL && fast->next != NULL) {

        fast = fast->next->next;

        slow = slow->next;

    }

    leftNode = head;

    rightNode = slow->next;

slow->next = NULL; // 将左边的的链表需要置空

/\*2.分别左右合并\*/

    leftNode = ListMergeSort(leftNode);

    rightNode = ListMergeSort(rightNode);

    return MergeSortList(leftNode, rightNode);

}

## 二：通用链表宏定义形式封装 common\_list.h

1.宏定义形式，主要为了方便将用户数据组织成链表。

2.头文件中只有对链表结点的操作，不允许对其他任何类型操作。

3.重点看创建、遍历、与获取用户结构体的首地址

### 【common\_list.h摘要】：

#### （0）通用链表结构体定义

typedef struct TagListNode {

    struct TagListNode\* pre;

    struct TagListNode\* next;

}ListNode;

#### （1）创建头结点的宏，

一般用做辅助头结点，head是用户定义的。

ListNode head = NULL;

ListCreate(head)

#### （2）头插宏：将结点插到head后

ListAddTohead(head, add)

#### （3）查找

上层判断res是否是NULL,来判断节点是否在链表中

ListNode\* res = NULL;

ListFind(head, targetNode, res);

#### （4）删除

ListDelOneNode(head, \_del)

#### （5）用户用于从链表结点获取用户结构体的入口

memberAddr是listNode的地址如&stu.listNode

type 指的是用户结构体

member指的是listNode

ListGetEntry(memberAddr, type, member)

### 【common\_list.h具体代码】

#ifndef \_COMMON\_LIST\_H\_

#define \_COMMON\_LIST\_H\_

/\*\*用这一条宏定义可以很好的组织链表，可以不管用户定义的结构是啥样的，都可以用这些接口，这里暂时没有使用pre\*/

typedef struct TagListNode {

    struct TagListNode\* pre;

    struct TagListNode\* next;

}ListNode;

/\*\*1.

    用户界面声明创建了一个名为oneList的空链表操作如下

    ListNode\* userList = NULL;

    ListCreate(userList);

\*/

#define ListCreate(head)                                    \

{                                                           \

    head = (ListNode\*)calloc(1, sizeof(ListNode));          \

    head->next = NULL;                                      \

    head->pre = NULL;                                       \

}

/\*\*2.添加

    用户界面在头节点后增加节点

    ListAddTohead(userList, &Type.listNode)

\*/

#define ListAddTohead(head, add)                            \

{                                                           \

    ListNode\* \_temp = NULL;                                  \

    \_temp = head->next;                                      \

    head->next = (add);                                     \

    (add)->next = \_temp;                                     \

}

/\*\*3.查找

    ListNode\* res = NULL;

    ListFind(head, targetNode, res);

    上层判断res是否是NULL,来判断节点是否在链表中

\*/

#define ListFind(head, \_target, \_res)         \

{                                           \

     ListNode\* \_temp = head;                \

     ListNode\* \_last = \_temp;               \

     while (\_temp != NULL) {                \

        if (\_temp == \_target) {              \

            \_res = \_target;                   \

            break;                          \

        }                                   \

        \_last = \_temp;                      \

        \_temp = \_temp->next;                \

     }                                      \

}

/\*\*4.删除

    用户界面删除节点

\*/

#define ListDelOneNode(head, \_del)                           \

{                                                           \

    ListNode\* \_temp = head;                            \

    ListNode\* \_last = \_temp;                                \

    while (\_temp != NULL) {                  \

        if (\_temp == (\_del)) {                                  \

            break;                                          \

        }                                                   \

        \_last = \_temp;                                        \

        \_temp = \_temp->next;                                 \

    }                                                       \

    \_last->next = (\_del)->next;                                \

}

/\*\*5.遍历

    用户界面想要遍历可以通过这个方式

    ListNode\* curPos;

    ListForEach(head, curPos) {

        //对curPos即可操作

    }

\*/

#define ListForEach(head, curPos)  for (curPos = head->next; curPos != NULL; curPos = curPos->next)

/\*\*6.

    用户界面获取curPos所在的结构体的入口

    成员地址 - 成员在结构体中的偏移

    type\* temp;

    temp = ListGetEntry(memberAddr, type, member);

    ANSI C标准允许任何值为0的常量被强制转换成任何一种类型的指针，并且转换结果是一个NULL指针，因此((s\*)0)的结果就是一个类型为s\*的NULL指针。

    如果利用这个NULL指针来访问s的成员当然是非法的，但&(((s\*)0)->m)的意图并非想存取s字段内容，而仅仅是计算当结构体实例的首址为((s\*)0)时m字段的地址。

    聪明的编译器根本就不生成访问m的代码，而仅仅是根据s的内存布局和结构体实例首址在编译期计算这个(常量)地址，这样就完全避免了通过NULL指针访问内存的问题。

\*/

#define ListGetEntry(memberAddr, type, member)  (type\*)((char\*)memberAddr - (char\*)(&(((type\*)0)->member)))

#endif // \_LIST\_H\_

### 【测试代码】

#include "common\_list.h"

typedef int (\*CmpFuncStr)(char\*, char\*);

typedef int (\*CmpFuncInt)(int, int);

/\*\*用户结构体\*/

typedef struct {

    int score;

    char name[10];

    CmpFuncStr myCmpStr;

    CmpFuncInt myCmpInt;

    ListNode listNode; /\*\*通用链表结构以方便组织数据为链表\*/

}Student;

int MyStrCmp(char\* name1, char\* name2)

{

    return strcmp(name1, name2);

}

int MyIntCmp(int a, int b)

{

    if (a > b) {

        return 1;

    } else if (a < b) {

        return -1;

    } else {

        return 0;

    }

}

ListNode\* MergeSortList(ListNode\* list1, ListNode\* list2)

{

    ListNode\* help = NULL;

    ListNode\* help\_cur = NULL;

    Student \*stu1, \*stu2;

    /\*\* help是list1的辅助头节点,记得释放\*/

    ListCreate(help);

    help\_cur = help;

    while (list1 != NULL && list2 != NULL) {

        stu1 = ListGetEntry(list1, Student, listNode);

        stu2 = ListGetEntry(list2, Student, listNode);

        // list1 < list2

        if (stu1->myCmpInt(stu1->score, stu2->score) <= 0) {

            help\_cur->next = list1;

            help\_cur = help\_cur->next;

            list1 = list1->next;

        } else {

            help\_cur->next = list2;

            help\_cur = help\_cur->next;

            list2 = list2->next;

        }

    }

    /\*\*这里用if就可以了\*/

    if (list1 != NULL) {

        help\_cur->next = list1;

    }

    if (list2 != NULL) {

        help\_cur->next = list2;

    }

    help\_cur = help->next;

    free(help);

    help = NULL;

    return help\_cur;

}

ListNode\* ListMergeSort(ListNode\* head)

{

    ListNode\* fast = NULL;

    ListNode\* slow = NULL;

    ListNode\* leftNode = NULL;

    ListNode\* rightNode = NULL;

    /\*\* 空链表或者只有一个节点的链表\*/

    if (head == NULL || head->next == NULL) {

        return head;

    }

    fast = head->next;/\*\*多走一个\*/

    slow = head;

    while (fast != NULL && fast->next != NULL) {

        fast = fast->next->next;

        slow = slow->next;

    }

    leftNode = head;

    rightNode = slow->next;

    slow->next = NULL; // 前面的链表需要置空

    leftNode = ListMergeSort(leftNode);

    rightNode = ListMergeSort(rightNode);

    return MergeSortList(leftNode, rightNode);

}

int main()

{

    ListNode\* studentList = NULL;

    ListNode\* curPos = NULL;

    int score[] = {1,2,3,4,5};

    char\* name[] = {"1","2","3","4","5"};

    int stuSize = sizeof(score) / sizeof(score[0]);

    Student student[stuSize];

    Student\* stuTmp = NULL;

    int i;

    int ret;

    ListCreate(studentList);

    for (i = 0; i < stuSize; i++) {

        student[i].score = score[i];

        student[i].myCmpStr = MyStrCmp;

        student[i].myCmpInt = MyIntCmp;

        strcpy(student[i].name, name[i]);

        ListAddTohead(studentList, (ListNode\*)(&student[i].listNode));

    }

    printf("/\*\*1.遍历链表\*/\n");

    ListForEach(studentList, curPos) {

      stuTmp = ListGetEntry(curPos, Student, listNode);

      printf("score %d name %s\n", stuTmp->score, stuTmp->name);

    }

    printf("/\*\*2.链表归并并升序排序\*/\n");

    studentList->next = ListMergeSort(studentList->next);

    ListForEach(studentList, curPos) {

      stuTmp = ListGetEntry(curPos, Student, listNode);

      printf("score %d name %s\n", stuTmp->score, stuTmp->name);

    }

    printf("/\*\*3.查找节点\*/\n");

    curPos = NULL;

    ListFind(studentList, &student[2].listNode, curPos);

    printf("curPos is %s studentList\n", curPos == NULL ? "not in" : "in");

    printf("/\*\*4.删除后查找节点\*/\n");

    ListDelOneNode(studentList, (ListNode\*)(&student[2].listNode));

    curPos = NULL;

    ListFind(studentList, &student[2].listNode, curPos);

    printf("after delete, curPos is %s studentList\n", curPos == NULL ? "not in" : "in");

    printf("hello\n");

}