# 定义

线性表是N个数据元素的有限序列。

说明：这里的数据元素可以是简单的一个变量，比如姓名，也可以是复杂的一个结构体，比如包含id，name，age的Person结构。

# 分类

（1）顺序表：数组

（2）链表：静态链表，单链表，循环链表，双向链表

说明：顺序表带有下标，在遍历和寻址的时候速度快，即查找相关效率高；当插入或者删除元素的时候，需要移动数组中的元素，效率较低。

## 数组

数组元素访问时间为什么是常数时间？

访问数组元素的时候，需要计算出该元素距离数组基址的偏移量，而为了计算偏移量，必须先做乘法，然后把乘法所得的结果，加到基址上面，已确定出该元素的内存地址。

整个过程涉及一次乘法与一次加法。由于两项操作都只需要花费常数级别的时间，因此访问数组元素，只需要常数级别时间。

### 动态数组

动态数组能够改变长度的数据结构。

有一种简单的实现方式，是先分配固定的数组空间，等到数组填满时，再多分配一倍的空间。同理，如果数组现存的元素还不到总量的1/2，那就把数组大小减半。

### 特点

数组优点：

1. 简单易用；
2. 可以迅速访问其元素（只需要消耗常数级别的时间）

缺点：

1. 大小固定；
2. 必须一次分配一整块内存；
3. 按照位置来插入元素是一种较为复杂的操作。

## 链表

为了克服数组的缺点，引入了链表。

链表的优缺点如下：

优点：能够在常数时间内增长，而数组不能。

缺点：

1. 访问单个元素花费的时间比较长；
2. 在空间局部性上不如数组（分配给数组的内存是连续的）；
3. 存储于获取数据时开销很大；
4. 在某些情况下很难维护。

### 单链表

单链表是单向的链表。

#### 定义

#### 遍历

#### 插入元素

#### 删除元素

### 静态链表

静态链表不使用链表，而是使用数组完成相关的操作。

### 循环链表

循环链表与单链表不同的是，最后尾节点的指针域指向了第一个元素的数据域。

### 双向链表

双向链表的每一个节点由两个指针域和一个数据域组成，一个指针域是寻找头结点，一个是寻找尾节点。

# 排序

## 单链表节点的交叉重排

题目要求：给定一个单链表如下所示：L0🡪L1🡪……🡪Ln-1🡪Ln，重新排序使其变成L0🡪Ln🡪L1🡪Ln-1🡪L2🡪Ln-2🡪……

代码：

#include <iostream>

#include <vector>

using namespace std;

/\*

按照要求重新为单链表排序

\*/

typedef struct list\_node List;

struct list\_node

{

struct list\_node\* next;

int value;

};

void print\_list(List\* list)

{

List\* tmp=list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

/\*

初始化List 将从1~n的数字插入到链表中

\*/

void Init\_List(List\*& head,int\* array,int n)

{

head = NULL;

List\* tmp;

List\* record;

for(int i=1;i<=n;i++)

{

tmp = new List;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

//求链表长度

int Len\_list(List\* list)

{

if(list == NULL)

return 0;

else

return Len\_list(list->next)+1;

}

/\*

链表的翻转

\*/

void Reverse(List\*& list)

{

List\* tmp = NULL;

List\* cur = list;

List\* next = list->next;

while(next != NULL)

{

cur->next = tmp;

tmp = cur;

cur = next;

next = next->next;

}

cur->next = tmp;

list = cur;

}

/\*

重新排序链表，将一个链表拆分，然后重新组合

关键点在于链表个数是偶数还是奇数

\*/

void Reorder\_list(List\*& list)

{

List\* first = list;

List\* second;

List\* tmp\_first,\*tmp\_second;

//需要根据链表中节点的个数来分割链表

int len = Len\_list(first);

int i;

if(len%2 == 0)

{

for(i=1;i<len/2;i++)

first = first->next;

}

else

{

for(i=1;i<len/2+1;i++)

first = first->next;

}

second = first->next;

first->next = NULL;

//将后面的链表进行翻转

Reverse(second);

//重新规划链表

first = list;

//开始进行合并，同时second链表的个数肯定不会比first链表的节点数多

while(second != NULL)

{

tmp\_first = first->next;

tmp\_second = second->next;

first->next= second;

second->next = tmp\_first;

second = tmp\_second;

first = tmp\_first;

} //能否使用伪指针将两个链表串联

}

int main()

{

int array[]={1,2,3,4,5,6,7,8,9,10,11};

List\* head;

Init\_List(head,array,sizeof(array)/sizeof(int));

Reorder\_list(head);

print\_list(head);

return 0;

}

## 单链表的分割

题目要求：给定一个单链表L和一个值x，经过处理，使得小于x的节点值在前面，不小于x的节点值在后面。

例如：给定一个链表1🡪4🡪3🡪2🡪5🡪 2和一个值x=3，链表经过处理之后变成1🡪2🡪2🡪4🡪3🡪5

代码：

#include <iostream>

#include <vector>

#include <string>

using namespace std;

typedef struct list\_node List;

struct list\_node

{

struct list\_node\* next;

int value;

};

void print\_list(List\* list)

{

List\* tmp=list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

/\*

初始化List 将从1~n的数字插入到链表中

\*/

void Init\_List(List\*& head,int\* array,int n)

{

head = NULL;

List\* tmp;

List\* record;

for(int i=1;i<=n;i++)

{

tmp = new List;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

//求链表长度

int Len\_list(List\* list)

{

if(list == NULL)

return 0;

else

return Len\_list(list->next)+1;

}

/\*

思路：类似于快排，由于要求不改变原来的相对顺序，所以必须有节点的交换

要不然之间交换节点内部的值即可。

\*/

/\*

类似于快速排序的分割

\*/

void PartitionList(List\*& list,int key)

{

if(list == NULL)

return ;

List\* record,\*cur,\*pre,\*tmp;

record = NULL;

cur = list;

pre = NULL;

//开始处理链表

while(cur != NULL)

{

if(cur->value< key) //插入到pre之后，需要特殊处理头节点

{

tmp = cur->next;

if(pre == NULL)

pre = cur;

if(record ==NULL)

{

record = list;

list = cur;

cur->next = record;

record = cur;

pre->next = tmp;

}

else

{

if(pre != record)

{

cur->next = record->next;

record->next = cur;

pre->next = tmp;

record = cur;

}

else

{

record = pre= cur;

}

}

cur = tmp;

}

else

{

pre= cur;

cur = cur->next;

}

}

}

//第二种方法 使用伪指针

List\* PartitionList2(List\* list,int key)

{

List\* head = new List;

head->next = list;

List\* temp = head;

List\* pre = head,\*cur = list,\*next;

//开始处理

while(cur != NULL)

{

next = cur->next;

if(cur->value < key)

{

pre->next = next;

cur->next = temp->next;

temp->next = cur;

temp = cur;

cur = next;

}

else

{

pre = cur;

cur = next;

}

}

return head->next;

}

int main()

{

int array[]={5,1,2,7,8,4,3,6,10,9};

List\* list;

Init\_List(list,array,sizeof(array)/sizeof(int));

list = PartitionList2(list,5);

print\_list(list);

return 0;

}

## 单链表的合并

题目要求：有多个已经排序好的单链表，将这些单链表合并成一个有序的单链表。

代码：

#include <iostream>

#include <vector>

using namespace std;

/\*

合并K个已经排序好的链表

\*/

typedef struct list\_node List;

struct list\_node

{

struct list\_node\* next;

int value;

};

void print\_list(List\* list)

{

List\* tmp=list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

/\*

初始化List 将从1~n的数字插入到链表中

\*/

void Init\_List(List\*& head,int\* array,int n)

{

head = NULL;

List\* tmp;

List\* record;

for(int i=1;i<=n;i++)

{

tmp = new List;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

//求链表长度

int Len\_list(List\* list)

{

if(list == NULL)

return 0;

else

return Len\_list(list->next)+1;

}

/\*

可以类似于合并几个已经排序好的数组这种思想，比如借助最小堆，每次将堆顶元素插入新的链表中，但是也可以使用数组来进行标记，每次从数组中挑选最小的节点直到所有链表的节点都被选中

\*/

List\* Merge\_k(vector<List\*>& vec)

{

int i;

//使用数组进行标记

vector<int> flag(vec.size(),1);

List\* head = NULL;

List\* cur = NULL;

List\* tmp = NULL;

int pos;

//开始处理K个链表

while(1)

{

tmp = NULL;

//开始遍历标记数组，找到合适的节点

for(i =0;i<vec.size();i++)

{

if(vec[i] == NULL)

flag[i] = 0;

if(flag[i])

{

if(tmp ==NULL)

{

tmp = vec[i];

pos =i;

}

if(tmp && tmp->value > vec[i]->value)

{

tmp = vec[i];

pos =i;

}

}

}

//开始特殊处理头节点

if(head == NULL)

{

head = tmp;

cur = head;

}

else

{

cur->next = tmp;

cur= cur->next;

}

//处理被选中的链表

vec[pos] = vec[pos]->next;

if(vec[pos] == NULL)

flag[pos] =0;

//判断是否所有的节点都已经被选中

for(i=0;i<flag.size();i++)

{

if(flag[i])

break;

}

if(i>=flag.size())

break;

}

return head;

}

int main()

{

int array1[]={1,4,7,8,13,19};

int array2[]={5,8,9,10,12,15,17,22,23};

int array3[]={3,6,11,16,17,18,21,24};

int array4[]={2,14,20,25};

vector<List\*> vec(4);

int i;

Init\_List(vec[0],array1,sizeof(array1)/sizeof(int));

Init\_List(vec[1],array2,sizeof(array2)/sizeof(int));

Init\_List(vec[2],array3,sizeof(array3)/sizeof(int));

Init\_List(vec[3],array4,sizeof(array4)/sizeof(int));

List\* head = Merge\_k(vec);

print\_list(head);

return 0;

}

## 单链表的排序

题目要求：

代码：

#include <iostream>

#include <vector>

using namespace std;

/\*

对于链表的排序 使用归并排序最好

\*/

typedef struct list\_node List;

struct list\_node

{

struct list\_node\* next;

int value;

};

void print\_list(List\* list)

{

List\* tmp=list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

/\*

初始化List 将从1~n的数字插入到链表中

\*/

void Init\_List(List\*& head,int\* array,int n)

{

head = NULL;

List\* tmp;

List\* record;

for(int i=1;i<=n;i++)

{

tmp = new List;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

//求链表长度

int Len\_list(List\* list)

{

if(list == NULL)

return 0;

else

return Len\_list(list->next)+1;

}

void FindMid(List\*& list,List\*& pre,List\*& last)

{

pre = list;

last = list->next;

while(last != NULL && last->next !=NULL)

{

pre = pre->next;

last = last->next;

if(last->next != NULL)

last = last->next;

}

last = pre->next;

pre->next = NULL;

pre = list;

}

//合并两个有序的链表

void Merge(List\*& list,List\*& pre,List\*& last)

{

if(pre == NULL)

{

list = last;

return;

}

if(last == NULL)

{

list =pre;

return;

}

List\* cur;

List\* tmp;

if(pre->value > last->value)

swap(pre,last);

//始终将Last链表中的节点插入到pre链表中

list = pre;

cur = pre;

while(cur->next != NULL && last != NULL)

{

if(cur->next->value > last->value)//插入元素

{

tmp = last->next;

last->next = cur->next;

cur->next = last;

cur = last;

last = tmp;

}

else

cur =cur->next;

}

if(last != NULL)

cur->next = last;

}

//更好一点的合并

void Merge\_sec(List\*& list,List\*& pre,List\*& last)

{

List\* tmp = new List;

list = tmp;

while(pre != NULL && last != NULL)

{

if(pre->value < last->value)

{

tmp->next = pre;

pre = pre->next;

}

else

{

tmp->next = last;

last = last->next;

}

tmp = tmp->next;

}

if(last != NULL)

tmp->next = last;

else

tmp->next = pre;

list = list->next;

}

/\*

归并排序的主体思想

\*/

void MergeSort(List\*& list)

{

if(list == NULL || list->next == NULL)

return ;

//找到中间点

List\* pre = NULL;

List\* last = NULL;

FindMid(list,pre,last);//将一个链表List从中间分成pre和last两部分

MergeSort(pre);//归并排序使得前半部分有序

MergeSort(last);//归并排序使得后半部分有序

Merge(list,pre,last);//将量部分的链表合并

}

/\*

归并排序第二种实现

\*/

List\* Merge(List\* first,List\* second)

{

List\* head = NULL;

List\* current = NULL;

if(first == NULL)

return second;

if(second == NULL)

return first;

if(first->value > second->value)

{

current = first;

first = second;

second = current;

}

head = first;

current = first;

first = first->next;

//始终将second的节点插入到first链表中

while(first != NULL && second != NULL)

{

List\* temp = NULL;

if(first->value > second->value)

{

temp = second->next;

current->next = second;

second->next = first;

current = second;

second = temp;

}

else

{

current = first;

first = first->next;

}

}

if(first == NULL)

current->next = second;

return head;

}

List\* MergeSort(List\* list,int size)

{

if(size == 0 | size == 1)

return list;

//找中间点

List\* middle = list;

int i;

for(i =1;i<size/2;i++)

middle = middle->next;

List\* temp = middle->next;

middle->next =NULL;

middle = temp;

List\* left = MergeSort(list,i);

List\* right = MergeSort(middle,size-i);

return Merge(right,left);

}

int main()

{

int array[]={7,4,9,15,2,1,6,10,12,11};

List\* head;

Init\_List(head,array,sizeof(array)/sizeof(int));

head = MergeSort(head,10);

print\_list(head);

return 0;

}

# 应用

## 通讯录

## 一元多项式

## 链表构造和翻转

代码：

#include <iostream>

using namespace std;

typedef struct list\_node ListNode;

struct list\_node

{

struct list\_node\* next;

int value;

};

/\*

初始化List 将从1~n的数字插入到链表中

\*/

void Init\_List(ListNode\*& head,int\* array,int n)

{

head = NULL;

ListNode\* tmp;

ListNode\* record;

for(int i=1;i<=n;i++)

{

tmp = new ListNode;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

void print\_list(ListNode\* list)

{

ListNode\* tmp = list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

//求单链表中节点的个数

int GetListLength(ListNode\* phead)

{

if(phead == NULL)

return 0;

int length = 0;

ListNode\* current = phead;

while(current != NULL)

{

length++;

current = current->next;

}

return length;

}

//递归的方法求解链表的长度

int Len\_list(ListNode\* list)

{

if(list == NULL)

return 0;

else

return Len\_list(list->next)+1;

}

//翻转单链表

ListNode\* ReverseList(ListNode\* phead)

{

//如果单链表为空或者只有一个节点，无需翻转，直接返回头节点

if(phead == NULL | phead->next == NULL)

return phead;

ListNode\* preverse = NULL;//翻转后的新链表头指针，初始化为NULL

ListNode\* current = phead;

while(current != NULL)

{

ListNode\* temp = current;

current = current->next;

temp->next = preverse;//将当前节点记录，插入新链表的最前端

preverse = temp;

}

return preverse;

}

int main()

{

int array[]={1,2,3,4,5,6,7,8,9,10};

ListNode\* list;

Init\_List(list,array,sizeof(array)/sizeof(int));

print\_list(list);

list = ReverseList(list);

print\_list(list);

return 0;

}

## 链表节点查找和删除

题目要求：

删除单链表第K个节点

递归打印单链表的节点

判断单链表中是否有环

O(1)时间复杂度删除单链表中的节点

代码：

#include <iostream>

using namespace std;

typedef struct list\_node ListNode;

struct list\_node

{

struct list\_node\* next;

int value;

};

/\*

初始化List 将从1~n的数字插入到链表中

\*/

void Init\_List(ListNode\*& head,int\* array,int n)

{

head = NULL;

ListNode\* tmp;

ListNode\* record;

for(int i=1;i<=n;i++)

{

tmp = new ListNode;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

void print\_list(ListNode\* list)

{

ListNode\* tmp = list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

//查找单链表中倒数第K个节点

ListNode\* GetKthNode(ListNode\* phead,int k)

{

//这里K的计数是从1开始

if(k == 0 || phead == NULL)

return NULL;

ListNode\* pAhead = phead;

ListNode\* pBehind = phead;

//前面的指针先走到正向第K个节点

while(k>1 && pAhead != NULL)

{

pAhead = pAhead->next;

k--;

}

//节点个数小于K，直接返回NULL

if(k>1 || pAhead == NULL)

return NULL;

//前后两个指针一起向前走，直到前面的指针指向最后一个节点

while(pAhead->next != NULL)

{

pBehind = pBehind->next;

pAhead = pAhead->next;

}

return pBehind;//后面的指针所指向的节点就是倒数第K个节点

}

//从尾到头打印链表，使用递归的方法

void RPrintList(ListNode\* phead)

{

if(phead == NULL)

return ;

else

{

RPrintList(phead->next);

cout<<phead->value<<endl;

}

}

//判断单链表中是否有环

bool HasCircle(ListNode\* phead)

{

ListNode\* pfast = phead;//快指针每次前进两步

ListNode\* pslow = phead;//慢指针每次前进一步

while(pfast!=NULL&& pfast->next != NULL)

{

pfast = pfast->next->next;

pslow = pslow->next;

if(pslow == pfast)

return true;

}

return false;

}

/\*

给出一单链表头指针phead和一个待删除的节点指针，

在O(1)时间复杂度内删除此节点

\*/

void Delete(ListNode\* phead,ListNode\* tobedelete)

{

if(tobedelete == NULL || phead == NULL)

return;

ListNode\* temp = phead;

//将下一个节点的数据复制到本节点，然后删除下一个节点

if(tobedelete->next != NULL)

{

tobedelete->value = tobedelete->next->value;

ListNode\* temp = tobedelete->next;

tobedelete->next = tobedelete->next->next;

delete temp;

}

else //要删除的是最后一个节点

{

if(phead == tobedelete)//链表中只有一个节点的情况

{

phead = NULL;

delete tobedelete;

}

else

{

ListNode\* pnode = phead;

while(pnode->next != tobedelete)//找到倒数第二个节点

pnode =pnode->next;

pnode->next = NULL;

delete tobedelete;

}

}

}

int main()

{

int array[]={1,2,3,4,5,6,7,8,9,10};

ListNode\* list;

Init\_List(list,array,sizeof(array)/sizeof(int));

// print\_list(list);

Delete(list,list->next->next->next);

print\_list(list);

return 0;

}

## 带环的单链表中入环的第一个节点

题目要求：在一个存在环的单链表中，查找进入环的第一个节点。

代码：

//已知一个单链表中存在环，求进入环中的第一个节点

ListNode\* GetFirstNodeInCircle(ListNode\* phead)

{

if(phead == NULL || phead->next == NULL)

{

return NULL;

}

ListNode\* pfast = phead;

ListNode\* pslow = phead;

//先判断是否存在环

while(pfast != NULL && pfast->next != NULL)

{

pslow = pslow->next;

pfast = pfast->next->next;

if(pslow == pfast)

break;

}

if(pfast == NULL || pfast->next == NULL)

return NULL;

//如果存在环，快慢指针都以同样的速度前进，相遇即为第一个节点

pfast = phead;

while(pslow != pfast)

{

pslow= pslow->next;

pfast = pfast->next;

}

return pslow;

}

//按照一定的要求删除链表中的节点

typedef boolk (\*remove\_fn)(ListNode\* node);

ListNode\* remove\_if(ListNode\* head,remove\_fn rm)

{

for(ListNode\* prev= NULL,\*curr = head;curr != NULL)

{

ListNode\* next = curr->next;

if(rm(curr))

{

if(prev)

prev->next = next;

else

head= next;

delete curr;

}

else

prev= curr;

curr = next;

}

return head;

}

// 升级版(使用二级指针删除单链表中的节点)

void remove\_if(ListNode\*\* head,remove\_if rm)

{

for(ListNode\*\* curr = head;\*curr;)

{

ListNode\* entry = \*curr;

if(rm(entry))

{

\*curr = entry->next;

delete entry;

}

else

curr = &entry->next;

}

} //仔细体味！

## 约瑟夫环

题目要求：使用单链表来解决约瑟夫环问题。

代码：

#include <iostream>

#include <vector>

#include <list>

using namespace std;

// 约瑟夫环问题，使用STL中的List，

int Joseph(list<int>& ring,int k)

{

list<int>::iterator itr = ring.begin(),temp;

int m;

//链表节点个数大于1就会一直删除下去

while(ring.size()>1)

{

m =1;

//判断当前迭代器是否需要重新置位

if(itr == ring.end())

itr = ring.begin();

//查找合适的节点进行删除

while(m <k)

{

//始终需要判断是否需要重新置位

if(itr == ring.end())

itr = ring.begin();

itr++;

m++;

if(itr == ring.end())

itr = ring.begin();

}

temp = itr;

itr++;

//删除被选中的节点

ring.erase(temp);

}

//只剩下一个节点

itr = ring.begin();

return \*itr;

}

int main()

{

int i;

list<int> ring;

for(i=1;i<=9;i++)

ring.insert(ring.begin(),i);

cout<<Joseph(ring,5)<<endl;

return 0;

}

## 删除单链表中重复的元素

题目要求：在一个已经排序好的链表中，删除链表中的重复元素，使得重复元素只保留一个。

代码：

#include <iostream>

#include <vector>

using namespace std;

typedef struct list\_node List;

struct list\_node

{

int value;

struct list\_node\* next;

};

void Init\_List(List\*& head,int\* array,int n)

{

head = NULL;

List\* tmp;

List\* record;

for(int i=1;i<=n;i++)

{

tmp = new List;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

void print\_list(List\* list)

{

List\* tmp=list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

/\*

删除重复的元素

\*/

void RemoveDuplicate(List\*& head)

{

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

return;

List\* slow = head;

List\* fast = head->next;

List\* tmp;

while(fast != NULL)

{

if(slow->value == fast->value)

{

tmp = fast;

fast = fast->next;

slow->next = fast;

delete tmp;

}

else

{

slow = slow->next;

fast = fast->next;

}

}

}

//另一种方法

List\* removeDuplicate(List\* head)

{

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

return head;

List\* temp,\*pre,\*cur;

List\* newhead = new List;

newhead->next = head;

temp = newhead;

pre = head;

cur = head->next;

while(cur != NULL)

{

while(cur != NULL && cur->value == pre->value)

cur = cur->next;

if(pre->next == cur)

{

temp->next = pre;

temp = pre;

pre = cur;

}

else

{

pre = cur;

}

if(cur != NULL)

cur = cur->next;

}

temp->next = cur;

return newhead->next;

}

int main()

{

int array[]={1,1,1,2,3,3,4,5,6,6,7,7};

List\* head;

Init\_List(head,array,sizeof(array)/sizeof(int));

head = removeDuplicate(head);

print\_list(head);

return 0;

}

拓展：

在一个已经排序号的链表中，删除链表中的重复元素，使得重复元素不保留。

## 单链表的K翻转

题目要求：在一个单链表中，给定一个值K，使得每K个节点都进行一次翻转。

例如给定一个单链表：1🡪2🡪3🡪4🡪5

如果给定K=2，操作之后返回的链表为2🡪1🡪4🡪3🡪5

如果给定K=3，操作之后返回的链表为3🡪2🡪1🡪4🡪5

代码：

#include <iostream>

#include <vector>

using namespace std;

/\*

按照要求做K翻转

\*/

typedef struct list\_node List;

struct list\_node

{

int value;

struct list\_node\* next;

};

void Init\_List(List\*& head,int\* array,int n)

{

head = NULL;

List\* tmp;

List\* record;

for(int i=1;i<=n;i++)

{

tmp = new List;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

void print\_list(List\* list)

{

List\* tmp=list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

/\*

翻转链表List 使得新的头部为Head，新的尾部为tail

\*/

void Reverse\_list(List\*& list,List\*& head,List\*& tail)

{

if(list == NULL || list->next == NULL)

return;

head = list;

tail = list;

List\* cur = NULL;

List\* next;

while(head != NULL)

{

next = head->next;

head->next = cur;

cur = head;

head = next;

}

list= cur;

head = cur;

}

/\*

做K个节点的翻转

\*/

void Reverse\_k(List\*& list,int k)

{

int num =1;

int flag =1;

if(list == NULL ||list->next == NULL || k ==0)

return ;

List\* head,\*tail,\*next,\*pre;

head = list;

tail= list;

while(tail != NULL && tail->next != NULL)

{

tail = tail->next;

num++;

if(num == k)

{

if(tail != NULL)

{

next = tail->next;

tail->next = NULL;

}

else

next = NULL;

Reverse\_list(head,head,tail);

if(flag)

{

list = head;

flag =0;

pre = tail;

}

else//第二次之后的翻转

{

pre->next = head;

pre = tail;

}

head = next;

tail = next;

num =1;

}

}

pre->next = head;

}

List\* Reverse(List\* pre,List\* end)

{

if(pre == NULL || pre->next == NULL)

return pre;

List\* head = pre->next;

List\* cur = pre->next->next;

while(cur != end)

{

List\* next = cur->next;

cur->next = pre->next;

pre->next = cur;

cur = next;

}

head->next = end;

return head;

}

//另一种比较简单的方法

List\* Reverse\_K(List\* head,int k)

{

if(head == NULL)

return NULL;

List\* dummy = new List;

dummy->next = head;

int count =0;

List\* pre = dummy;

List\* cur = head;

while(cur != NULL)

{

count++;

List\* next = cur->next;

if(count == k)

{

pre = Reverse(pre,next);

count =0;

}

cur = next;

}

return dummy->next;

}

int main()

{

int array[]={1,2,3,4,5,6,7,8,9,10,11};

List\* list,\*head,\*tail;

Init\_List(list,array,sizeof(array)/sizeof(int));

list = Reverse\_K(list,3);

print\_list(list);

return 0;

}

## 单链表中区间节点的翻转

题目要求：在一个单链表中，给定两个值m和n，翻转该链表中m到n这段区间的节点。

例如给定一个单链表：1🡪2🡪3🡪4🡪5🡪NULL

如果给定值m=2andn=4，操作之后返回的链表为1🡪4🡪3🡪2🡪5🡪NULL

代码：

#include <iostream>

#include <vector>

#include <string>

using namespace std;

typedef struct list\_node List;

struct list\_node

{

int value;

struct list\_node\* next;

};

void Init\_List(List\*& head,int\* array,int n)

{

head = NULL;

List\* tmp;

List\* record;

for(int i=1;i<=n;i++)

{

tmp = new List;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

void print\_list(List\* list)

{

List\* tmp=list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

/\*

将链表中的第m个节点到第n个节点之间的元素进行翻转

\*/

void ReverseList(List\*& list,int m,int n)

{

if(list == NULL ||list->next == NULL || n-m<1)

return ;

int num =1;

List\* pre,\*next,\*cur,\*temp,\*tmp;

cur = list;

pre = NULL;

while(cur != NULL)

{

next = cur->next;

if(num < m)

{

pre = cur;

cur = next;

}

if(num == m)

{

tmp = cur;

temp = cur;

cur = next;

}

if(num >m && num <= n)

{

cur->next = temp;

temp = cur;

cur = next;

}

if(num == n)

{

if(m ==1)

list = temp;

else

pre->next = temp;

tmp->next = cur;

break;

}

num++;

}

}

int main()

{

int array[]={5,1,2,7,8,4,3,6,10,9};

List\* list ;

Init\_List(list,array,sizeof(array)/sizeof(int));

ReverseList(list,1,3);

print\_list(list);

return 0;

}