# 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。

# 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！