第3章 链表

**一、复习要点**

本章重点讨论最简单的链表结构——单链表。详细地介绍了单链表的抽象数据类型，单链表的类定义，相应操作的实现，引入了带表头结点的单链表结构。进一步定义了用模板描述的单链表类。作为一种应用，讨论了一元多项式的类定义及其加法操作的实现。此外，讨论了循环链表和双向链表。在复习这一章时需要对C++ 语言中的指针和引用类型的使用有清楚的理解。对带表头结点的链表和不带表头结点的链表在插入、删除、搜索时的差别有清楚的认识。而且需要明确：链表是一种实现级的结构。

本章复习的要点：

**1、**基本知识点

单链表是一种线性结构，链表各结点的物理存储可以是不连续的，因此各结点的逻辑次序与物理存放次序可以不一致。必须理解单链表的定义和特点，单链表的抽象数据类型和类定义，单链表成员函数，如构造函数、搜索、插入、删除等操作的实现，对比带表头结点单链表的搜索、插入、删除操作，比较其优缺点。其次是循环链表的定义和特点，它与单链表的差别，它的搜索、插入、删除操作的实现。最后是双向链表的定义，它的插入与删除操作的实现。

**2、**算法设计

⮚ 单链表的迭代求解算法，包括统计链表结点个数，在链表中寻找与给定值value匹配的结点，在链表中寻找第i个结点，在链表中第i个位置插入新结点，删去第i个结点，单链表各结点顺序逆转算法，在单链表中按从左到右和从右到左的顺序遍历的逆转链算法。

⮚ 带表头结点的单链表的迭代算法，包括统计链表结点个数，在链表中寻找与给定值value匹配的结点，在链表中寻找第i个结点，在链表中第i个位置插入新结点，删去第i个结点，连续删除链表中含有value值的结点，两个有序链表的合并。

⮚ 单链表的递归算法，包括统计链表结点个数，在链表中寻找与给定值value匹配的结点，在链表中寻找第i个结点，求链表各结点值的和，求链表各结点的值的平均值。

⮚ 循环链表的迭代算法：包括统计链表结点个数，在链表中寻找与给定值value匹配的结点，在链表中寻找第i个结点，在链表中第i个位置插入新结点，删去第i个结点，将循环链表链入单链表的表头。

⮚ 多项式的建立，两个多项式的相加，两个多项式的相减。

⮚ 用单链表实现字符串操作，每个结点仅存一个字符。

**二、难点和重点**

1、单链表：单链表定义、相应操作的实现。

⮚ 单链表的两种定义方式(复合方式与嵌套方式)

⮚ 单链表的搜索算法与插入、删除算法

⮚ 单链表的递归与迭代算法

2、循环链表：单链表与循环链表的异同

3、双向链表：带表头结点的双向循环链表

⮚ 双向循环链表的定义，带表头结点的优点

⮚ 双向链表的搜索、插入与删除算法

4、多项式：多项式的定义、多项式的表示及加法

⮚ 多项式.的三种表示

⮚ 多项式链接表示的优点

⮚ 多项式加法的实现(有序链表的合并算法)

**三、教材中习题的解析**

3-1线性表可用顺序表或链表存储。试问：

(1) 两种存储表示各有哪些主要优缺点?

(2) 如果有n个表同时并存，并且在处理过程中各表的长度会动态发生变化，表的总数也可能自动改变、在此情况下，应选用哪种存储表示？为什么？

(3) 若表的总数基本稳定，且很少进行插入和删除，但要求以最快的速度存取表中的元素，这时，应采用哪种存储表示？为什么？

【解答】

(1) 顺序存储表示是将数据元素存放于一个连续的存储空间中，实现顺序存取或(按下标)直接存取。它的存储效率高，存取速度快。但它的空间大小一经定义，在程序整个运行期间不会发生改变，因此，不易扩充。同时，由于在插入或删除时，为保持原有次序，平均需要移动一半(或近一半)元素，修改效率不高。

链接存储表示的存储空间一般在程序的运行过程中动态分配和释放，且只要存储器中还有空间，就不会产生存储溢出的问题。同时在插入和删除时不需要保持数据元素原来的物理顺序，只需要保持原来的逻辑顺序，因此不必移动数据，只需修改它们的链接指针，修改效率较高。但存取表中的数据元素时，只能循链顺序访问，因此存取效率不高。

(2) 如果有n个表同时并存，并且在处理过程中各表的长度会动态发生变化，表的总数也可能自动改变、在此情况下，应选用链接存储表示。

如果采用顺序存储表示，必须在一个连续的可用空间中为这n个表分配空间。初始时因不知道哪个表增长得快，必须平均分配空间。在程序运行过程中，有的表占用的空间增长得快，有的表占用的空间增长得慢；有的表很快就用完了分配给它的空间，有的表才用了少量的空间，在进行元素的插入时就必须成片地移动其他的表的空间，以空出位置进行插入；在元素删除时，为填补空白，也可能移动许多元素。这个处理过程极其繁琐和低效。

如果采用链接存储表示，一个表的存储空间可以连续，可以不连续。表的增长通过动态存储分配解决，只要存储器未满，就不会有表溢出的问题；表的收缩可以通过动态存储释放实现，释放的空间还可以在以后动态分配给其他的存储申请要求，非常灵活方便。对于n个表(包括表的总数可能变化)共存的情形，处理十分简便和快捷。所以选用链接存储表示较好。

(3) 应采用顺序存储表示。因为顺序存储表示的存取速度快，但修改效率低。若表的总数基本稳定，且很少进行插入和删除，但要求以最快的速度存取表中的元素，这时采用顺序存储表示较好。

3-2 针对带表头结点的单链表，试编写下列函数。

(1) 定位函数Locate：在单链表中寻找第i个结点。若找到，则函数返回第i个结点的地址；若找不到，则函数返回NULL。

(2) 求最大值函数max：通过一趟遍历在单链表中确定值最大的结点。

(3) 统计函数number：统计单链表中具有给定值x的所有元素。

(4) 建立函数create：根据一维数组a[n]建立一个单链表，使单链表中各元素的次序与a[n]中各元素的次序相同，要求该程序的时间复杂性为O(n)。

(5) 整理函数tidyup：在非递减有序的单链表中删除值相同的多余结点。

【解答】

单链表的结点类(ListNode **class**)和链表类(List **class**)的类定义。

**#ifndef** LIST\_H //将单链表定义在List.h

**#define** LIST\_H

**template <class Type> class** List**;** //前视的类定义

**template <class Type> class** ListNode **{** //链表结点类的定义

**friend class** List**<Type>;** //List类作为友元类定义

**private:**

**Type** data**;** //数据域

ListNode**<Type>** \*link**;** //链指针域

**public:**

ListNode ( ) **:** link (NULL) **{ }** //仅初始化指针成员的构造函数

ListNode ( **Type** item, ListNode<**Type**> \* next = NULL ) **:** data (item)**,** link (next) **{ }**

//初始化数据与指针成员的构造函数

ListNode**<Type>** \* getLink ( ) **{ return** link**;** **}** //取得结点的下一结点地址

**Type** getData ( ) **{ return** data**; }** //取得结点中的数据

**void** setLink ( ListNode**<Type>** \* next ) **{** link = next**; }** //修改结点的link指针

**void** setData ( **Type** value ) **{** data = value**; }** //修改结点的data值

**};**

**template <class Type> class** List **{** //单链表类定义

**private:**

ListNode**<Type>** \*first**,** \*current**;** //链表的表头指针和当前元素指针

**public:**

List ( **Type** value ) **{** first = current = **new** ListNode**<Type>** ( value )**; }** //构造函数

~List ( ) **{** MakeEmpty ( )**; delete** first**; }** //析构函数

**void** MakeEmpty ( )**;** //将链表置为空表

**int** Length ( ) **const;** //计算链表的长度

ListNode**<Type>** \* Find ( **Type** value )**;** //搜索含value的元素并成为当前元素

ListNode<**Type**> \* Locate( **int** i ); //搜索第i个元素并置为当前元素

**Type** GetData ( ) **{ return** current->data**; }**  //取出表中当前元素的值

**int** Insert ( **Type** value )**;** //将value插在当前位置后并成为当前元素

**Type** \*Remove ( )**;** //将表中当前元素删去, 填补者为当前元素

ListNode**<Type>** \* Firster ( ) **{** current = first**; return** first**; }** //当前指针定位于表头

**Type** First ( ) { **;** //当前指针定位于表第一个元素并返回值

**Type** \*Next ( )**;** //将当前指针进到表中下一个元素并返回值

**int** NotNull ( ) **{** **return** current != NULL**; } //**表中当前元素空否？空返回1, 不空返回0

**int** NextNotNull ( ) **{** **return** current != NULL **&&** current->link != NULL**; }**

**}; //**当前元素的下一元素空否？空返回1, 不空返回0

(1) 实现定位函数的算法如下：

**template <class Type>** ListNode **<Type>** \* List **<Type> ::** Locate ( **int** i ) **{**

//取得单链表中第i个结点地址, i从1开始计数, i <= 0时返回指针NULL

**if** ( i <= 0 ) **return** NULL**;** //位置i在表中不存在

ListNode <**Type**> \* p = first**; int** k = 0**;** //从表头结点开始检测

**while** ( p != NULL **&&** k < i ) **{** p = p->link**;** k++**; }** //循环, p == NULL表示链短, 无第i个结点

**return** p**; //**否则k == i, 返回第i个结点地址

**}**

(2) 实现求最大值的函数如下：

**template <class Type>** ListNode **<Type>** \* List **<Type> ::** Max ( ) **{**

//在单链表中进行一趟检测，找出具有最大值的结点地址, 如果表空, 返回指针NULL

**if** ( first->link == NULL ) **return** NULL**;** //空表, 返回指针NULL

ListNode <**Type**> \* pmax = first->link, p = first->link->link**;**

//假定第一个结点中数据具有最大值

**while** ( p != NULL ) **{** //循环, 下一个结点存在

**if** ( p->data > pmax->data ) pmax = p**;** //指针pmax记忆当前找到的具最大值结点

p = p->link**;** //检测下一个结点

**}**

**return** pmax**;**

**}**

(3) 实现统计单链表中具有给定值x的所有元素的函数如下：

**template <class Type> int** List **<Type> ::** Count ( **Type&** x ) **{**

//在单链表中进行一趟检测，找出具有最大值的结点地址, 如果表空, 返回指针NULL

int n = 0**;**

ListNode <**Type**> \* p = first->link**;** //从第一个结点开始检测

**while** ( p != NULL ) **{** //循环, 下一个结点存在

**if** ( p->data == x ) n++**;** //找到一个, 计数器加1

p = p->link**;** //检测下一个结点

**}**

**return** n**;**

**}**

(4) 实现从一维数组A[n]建立单链表的函数如下：

**template <class Type> void** List **<Type> ::** Create ( **Type** A[ ], **int** n ) **{**

//根据一维数组A[n]建立一个单链表，使单链表中各元素的次序与A[n]中各元素的次序相同

ListNode**<Type**> \* p**;**

first = p = **new** ListNode<**Type**>**;** //创建表头结点

**for** ( **int** i = 0**;** i < n**;** i++ ) **{**

p->link = **new** ListNode<**Type**> ( A[i] )**;** //链入一个新结点, 值为A[i]

p = p->link**;** //指针p总指向链中最后一个结点

**}**

p->link = NULL**;**

**}**

采用递归方法实现时，需要通过引用参数将已建立的单链表各个结点链接起来。为此，在递归地扫描数组A[n]的过程中，先建立单链表的各个结点，在退出递归时将结点地址p（被调用层的形参）带回上一层（调用层）的实参p->link。

**template<Type> void** List<**Type> ::** create ( **Type** A[ ]**, int** n**, int** i, ListNode<**Type> \*&** p ) **{**

//私有函数：递归调用建立单链表

**if** ( i == n ) p = NULL**;**

**else {** p = **new** ListNode<**Type**>( A[i] )**;** //建立链表的新结点

create ( A, n, i+1, p->link )**;** //递归返回时p->link中放入下层p的内容

**}**

**}**

**template<Type> void** List<**Type> ::** create ( **Type** A[ ], **int** n ) **{**

//外部调用递归过程的共用函数

first = current = **new** ListNode<**Type>;** //建立表头结点

create ( A, n, 0, first->link )**;** //递归建立单链表

**}**

(5) 实现在非递减有序的单链表中删除值相同的多余结点的函数如下：

**template <class Type> void** List **<Type> ::** tidyup ( ) **{**

ListNode**<Type**> \* p = first->link, temp**;** //检测指针, 初始时指向链表第一个结点

**while** ( p != NULL **&&** p->link != NULL ) //循环检测链表

**if** ( p->data == p->link->data ) **{** //若相邻结点所包含数据的值相等

temp = p->first**;** p->link = temp->link**;** //为删除后一个值相同的结点重新拉链

**delete** temp**;** //删除后一个值相同的结点

**}**

**else** p = p->link**;** //指针p进到链表下一个结点

**}**

3-3 设ha和hb分别是两个带表头结点的非递减有序单链表的表头指针, 试设计一个算法, 将这两个有序链表合并成一个非递增有序的单链表。要求结果链表仍使用原来两个链表的存储空间, 不另外占用其它的存储空间。表中允许有重复的数据。

【解答】

**#include** <iostream.h>

**template <class Type> class** List**;**

**template <class Type> class** ListNode **{**

**friend class** List<**Type**>**;**

**public:**

ListNode ( ) **:** link ( NULL ) **{ }** //构造函数, 仅初始化指针成员

ListNode ( **Type** item, ListNde<**Type**> \* next = NULL ) **:** data ( item )**,** link ( next ) **{ }**

**private:** //构造函数, 初始化数据与指针成员

**Type** data**;**

ListNode<**Type**> \*link**;**

**};**

**template <class Type> class** List **{**

**private:**

ListNode<**Type**> \*first**,** \*last**;**

**public:**

List ( **Type** finishied ) **{** first = last = **new** ListNode<**Type**>( finished )**; }**

//建立链表, 在表头结点的data域中存放数据输入结束标志, 它是表中不可能出现的数据

**void** Merge ( List<**Type**> **&**hb )**;** //连接链表

**friend istream& operator** >> ( **istream&** in**,** List<**Type**> inList )**;** //输入链表

**friend ostream& operator <<** ( **ostream&** out**,** List**<Type>** outList )**;** //输出链表

**}**

**istream& operator** >> ( **istream&** in**,** List<**Type**> inList ) **{**

**Type** value**;** ListNode<**Type**> \*p, \*q, \*s**;**

in >> value**;**

**while** ( value != inList.first->data ) **{** //循环建立各个结点

s = **new** ListNode<**Type**>( value )**;**

q = first**;** p = inList.first->link**;** //寻找新结点插入位置

**while** ( p != NULL **&&** p->data <= value ) **{** q = p**;** p = p->link**; }**

q->link = s**;** s->link = p**;** //在q, p间插入新结点

**if** ( p == NULL ) inList.last = s**;**

in >> value**;**

**}**

**}**

**ostream& operator <<** ( **ostream&** out**,** List**<Type>** outList ) **{**

**cout**<<"\nThe List is : \n"**;**

ListNode<**Type>** \*p = outList.first->link**;**

**while** ( p != NULL ) **{**

out << p->data**;**

**if** ( p != last ) out << "->"**;**

**else** out **<< endl;**

p = p->link**;**

**}**

**}**

**template <class Type> void** List <**Type**> **::** Merge ( List<**Type**>**&** hb ) **{**

//将当前链表**this**与链表hb按逆序合并，结果放在当前链表**this**中。 ListNode<**Type**> \*pa, \*pb, \*q, \*p**;** pa = first->link**;** pb = hb.first->link**;** //检测指针跳过表头结点

first->link = NULL**; /**/结果链表初始化

**while** ( pa != NULL **&&** pb != NULL ) **{** //当两链表都未结束时 **if** ( pa->data <= pb->data )

**{** q = pa**;** pa = pa->link**; }** //从pa链中摘下

**else**

**{** q = pb**;** pb = pb->link**; }** //从pb链中摘下

q→link = first->link**;** first->link = q**;** //链入结果链的链头

**}**p **=** ( pa != NULL ) **?** pa **:** pb**;** //处理未完链的剩余部分

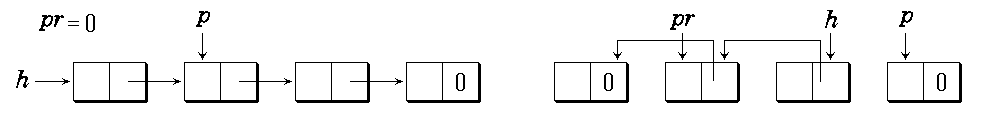
**while** ( p != NULL ) **{** q = p**;** p = p->link**;**

q->link = first->link**;** first->link = q**;**

**}**

**}**

3-4 设有一个表头指针为h的单链表。试设计一个算法，通过遍历一趟链表，将链表中所有结点的链接方向逆转，如下图所示。要求逆转结果链表的表头指针h指向原链表的最后一个结点。



【解答1】

**template<class Type> void** List<**Type**> **::** Inverse ( ) {

**if** ( first == NULL ) **return;**

ListNode<**Type**> \*p = first->link, **\***pr =NULL**;**

**while** ( p != NULL ) **{**

first->link = pr**;** //逆转first指针

pr = first**;** first = p**;** p = p->link**;** //指针前移

**}**

first->link = pr**;**

**}**

【解答2】

**template<class Type> void** List<**Type**> **::** Inverse ( ) **{**

ListNode<**Type**> \*p, \*head = **new** ListNode<**Type**> ( )**;** //创建表头结点, 其link域默认为NULL

**while** ( first != NULL ) **{**

p = first**;** first = first->link**;** //摘下first链头结点

p->link = head->link**;** head->link = p**; /**/插入head链前端

**}**

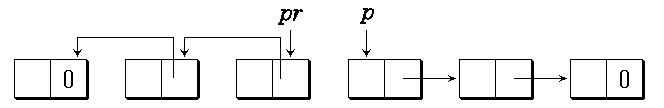
first = head->link**; delete** head**;** //重置first, 删去表头结点

**}**

3-5 从左到右及从右到左遍历一个单链表是可能的，其方法是在从左向右遍历的过程中将连接方向逆转，如右图所示。在图中的指针p指向当前正在访问的结点，指针pr指向指针p所指结点的左侧的结点。此时，指针p所指结点左侧的所有结点的链接方向都已逆转。

(1) 编写一个算法，从任一给定的位置(pr, p)开始，将指针p右移k个结点。如果p移出链表，则将p置为0，并让pr停留在链表最右边的结点上。

(2) 编写一个算法，从任一给定的位置(pr, p)开始，将指针p左移k个结点。如果p移出链表，则将p置为0，并让pr停留在链表最左边的结点上。



【解答】

(1) 指针p右移k个结点

**template<class Type> void** List<**Type**> **::**

siftToRight ( ListNode<**Type**> \***&** p**,** ListNode<**Type**> \***&** pr**, int** k ) **{**

**if** ( p == NULL **&&** pr != first ) **{** //已经在链的最右端

**cout <<** "已经在链的最右端，不能再右移。" << **endl;**

**return;**

**}**

**int** i**;** ListNode<**Type**> \*q**;**

**if** ( p == NULL) //从链头开始

**{** i **=** 1**;** pr = NULL**;** p = first**; }** //重置p到链头也算一次右移

**else** i = 0**;**

**while** ( p != NULL **&&** i < k ) **{** //右移k个结点

q = p->link**;** p->link = pr**;** //链指针p→link逆转指向pr

pr = p**;** p = q**;** i**++;** //指针pr, p右移

**}**

**cout <<** "右移了" << i << "个结点。" << **endl;**

**}**

(2) 指针p左移k个结点

**template<class Type> void** List<**Type**> **::**

siftToLeft ( ListNode<**Type**> \***&** p**,** ListNode<**Type**> \***&** pr**, int** k ) **{**

**if** ( p == NULL **&&** pr == first ) **{** //已经在链的最左端

**cout <<** "已经在链的最左端，不能再左移。" << **endl;**

**return;**

**}**

**int** i = 0**;** ListNode<**Type**> \*q**;**

**while** ( pr != NULL **&&** i < k ) **{** //左移k个结点

q = pr->link**;** pr->link = p**;** //链指针pr->link逆转指向p

p = pr**;** pr = q**;** i**++;** //指针pr, p左移

**}**

**cout <<** "左移了" << i << "个结点。" << **endl;**

**if** ( i < k ) **{** pr = p**;** p = NULL**; }** //指针p移出表外，重置p, pr

**}**

3-6 试写出用单链表表示的字符串类及字符串结点类的定义，并依次实现它的构造函数、以及计算串长度、串赋值、判断两串相等、求子串、两串连接、求子串在串中位置等7个成员函数。要求每个字符串结点中只存放一个字符。

【解答】

//用单链表表示的字符串类string1的头文件string1.h

#include <iostream.h>

**const int** maxLen = 300**;** //字符串最大长度为300（理论上可以无限长）

**class** string1 **{**

**public:**

string1 ( )**;** //构造空字符串

string1 ( **char** \* obstr )**;** //从字符数组建立字符串

~string1 ( )**;** //析构函数

**int** Length ( ) **const { return** curLen**; }** //求字符串长度

string1**& operator** = ( string1**&** ob )**;** //串赋值

**int operator** == ( string1**&** ob )**;** //判两串相等

**char operator** [ ] ( **int** i )**;** //取串中字符

string1 **operator** ( ) ( **int** pos**, int** len )**;**  //取子串

string1**& operator** += ( string1**&** ob )**;** //串连接

**int** Find ( string1**&** ob )**;** //求子串在串中位置(模式匹配)

**friend ostream& operator** << ( **ostream&** os**,** string1**&** ob )**;**

**friend istream& operator** >> ( **istream&** is**,** string1**&** ob )**;**

**private:**

ListNode<**char**>\*chList**;** //用单链表存储的字符串

**int** curLen**;** //当前字符串长度

**}**

//单链表表示的字符串类string1成员函数的实现，在文件string1.cpp中

**#include** <iostream.h>

**#include** "string1.h"

string1 **::** string1( ) **{** //构造函数

chList = **new** ListNode<**char**> ('\0' )**;**

curLen = 0**;**

**}**

string1 **::** string1( **char** \*obstr ) **{** //复制构造函数

curLen = 0**;**

ListNode<**char**> \*p = chList = **new** ListNode<**char**> ( \*obstr )**;**

**while** ( \*obstr != '\0' ) **{**

obstr++**;**

p = p->link = **new** ListNode<**char**> ( \*obstr )**;**

curLen++**;**

**}**

**}**

string1**&** string1 **:: operator** = ( string1**&** ob ) **{** //串赋值

ListNode<**char**> \*p = ob.chList**;**

ListNode<**char**> \*q = chList = **new** ListNode<**char**> ( p->data )**;**

curLen = ob.curLen**;**

**while** ( p->data != '\0' ) **{**

p = p->link**;**

q = q->link = **new** ListNode<**char**> ( p->data )**;**

**}**

**return this;**

**}**

**int** string1 **:: operator** == ( string1**&** ob ) **{** //判两串相等

**if** ( curLen != ob.curLen ) **return** 0**;**

ListNode <**char**> \*p = chList**,** \*q = ob.chList**;**

**for** ( **int** i = 0**;** i < curLen**;** i++ )

**if** ( p->data != q->data ) **return** 0**;**

**else {** p = p->link**;** q = q->link**; }**

**return** 1**;**

**}**

**char** string1 **:: operator** [ ] ( **int** i ) **{** //取串中字符

**if (** i >= 0 **&&** i < curLen **) {**

ListNode <**char**> \*p = chList**; int** k = 0**;**

**while** ( p != NULL **&&** k < i ) **{** p = p->link**;** k++**; }**

**if** ( p != NULL ) **return** p->data**;**

**}**

**return** '\0'**;**

**}**

string1 string1 **:: operator** ( ) ( **int** pos**, int** len ) **{** //取子串

string1 temp**;**

**if** ( pos >= 0 **&&** len >= 0 **&&** pos < curLen **&&** pos + len - 1 < curLen ) **{**

ListNode<**char**> \*q, \*p = chList**;**

**for** ( **int** k = 0**;** k < pos**;** k++**;** ) p = p->link**;** //定位于第pos结点

q = temp.chList = **new** ListNode<**char**> ( p->data )**;**

**for** ( **int** i = 1**;** i < len**;** i++ ) **{** //取长度为len的子串

p = p->link**;**

q = q->link = **new** ListNode<**char**> ( p->data )**;**

**}**

q->link = **new** ListNode<**char**> ( '\0' )**;** //建立串结束符

temp.curLen = len**;**

**}**

**else {** temp.curLen = 0**;** temp.chList = **new** ListNode<**char**> ( '\0' )**; }**

**return** temp**;**

**}**

string1**&** string1 **:: operator** += ( string1**&** ob ) **{** //串连接

**if** ( curLen + ob.curLen > maxLen ) len = maxLen - curLen**;**

**else** len = ob.curLen; //传送字符数

ListNode<**char**> \*q = ob.chList**,** \*p = chList**;**

**for** ( **int** k = 0**;** k < curLen - 1**;** k++**;** ) p = p->link**;** //**this**串的串尾

k = 0**;**

**for** ( k = 0**;** k < len**;** k++ ) **{** //连接

p = p->link = **new** ListNode<**char**> ( q->data )**;**

q = q->link**;**

**}**

p→link = **new** ListNode<**char**> ( '\0' )**;**

**return this;**

**}**

**int** string1 **::** Find ( string1**&** ob ) **{** //求子串在串中位置(模式匹配)

**int** slen = curLen**,** oblen = ob.curLen**,** i = slen - oblen**;**

string1 temp = **this**;

**while** (i > -1 )

**if** ( temp( i**,** oblen ) == ob ) **break;**

**else** i*--* **;**

**return** i**;**

**}**

3-7 如果用循环链表表示一元多项式，试编写一个函数Polynomial **::** Calc(x)，计算多项式在x处的值。

【解答】

下面给出表示多项式的循环链表的类定义。作为私有数据成员，在链表的类定义中封装了3个链接指针：first、last和current，分别指示链表的表头结点、链尾结点和最后处理到的结点。

**class** Polynomal**;** //多项式前视类定义

**class** Term **{** //项类定义

**friend class** Polynomal**;**

**private:**

**double** coef**;** //系数

**int** expn**;** //指数

Term \*link**;** //项链接指针

**public:**

Term ( **double** c = 0, **int** e = 0, Term \* next = NULL ) **:** coef (c), expn(e), link (next) **{ }**

**}**

**class** Polynomal **{** //多项式类定义

**private:**

Term \*first, \*current**;** //头指针, 当前指针

**int** n**;** //多项式阶数

**public:**

Polynomal ( )**;** //构造函数

~Polynomal ( )**;** //析构函数

**int** Length ( ) **const;** //计算多项式项数

**int** IsEmpty ( ) **{ return** first->link == first**;** **}** //判是否零多项式

**int** Find ( **int** value )**;** //在多项式中寻找其指数值等于value的项

**int** getExpn( ) **const;** //返回当前项中存放的指数值

**double** getCoef( ) **const;** //返回当前项中存放的系数值

**void** Firster ( ) **{** current = first**; }** //将当前指针置于头结点

**int** First ( )**;** //将当前指针指向链表的第一个结点

**int** Next ( )**;** //将当前指针指到当前结点的后继结点

**int** Prior ( )**;** //将当前指针指到当前结点的前驱结点

**void** Insert ( **const double** coef**, int** expn )**;** //插入新结点

**void** Remove ( )**;** //删除当前结点

**double** Calc ( **double** x )**;** //求多项式的值

**friend** Polynomial **operator** + ( Polynomial **&,** Polynomial **&** )**;**

**friend** Polynomial **operator** \* ( Polynomial **&,** Polynomial **&** )**;**

**};**

对于多项式Pn(x) = a0 + a1x + a2x2 + a3x3 + … + an-1xn-1 + anxn，可用Horner规则将它改写求值： Pn(x) = a0 + (a1x + ( a2+ ( a3+ … + ( an-1+ an\*x )\*x … )\*x )\*x )\*x

因为不是顺序表，必须采用递归算法实现：

返回求值

Value(n) = a0 + Value(n-1)\*x

Value(n-1) = a1 + Value(n-2)\*x

……

Value(1) = an-1 + Value(0)

Value(0) = an

递归求解

**double** Polynomal **::** Value ( Term \*p, **double** x ) **{**

//私有函数：递归求子多项式的值

**if** ( p->link == first ) **return** p->coef**;**

**else return** p->coef + x \* Value ( p->link, x )**;**

**}**

**double** Polynomal **::** Calc ( **double** x ) **{**

//共有函数：递归求多项式的值

Term \* pc = first->link**;**

**return** ( pc == first ) ? 0.0 **:** Value ( pc, x )**;**

**}**

但当多项式中许多项的系数为0时变成稀疏多项式，如P50(x) = a0 + a13x13 + a35x35 + a50x50，为节省存储起见，链表中不可能保存有零系数的结点。此时，求值函数要稍加改变：

#**include** <math.h>

**double** Polynomal **::** Value ( Term \*p, **double** e, **double** x ) **{**

//私有函数：递归求子多项式的值。pow(x, y)是求x的y次幂的函数, 它的原型在“math.h”中

**if** ( p->link == first ) **return** p->coef**;**

**else return** p->coef + pow( x, p->expn – e ) \* Value ( p->link, p->expn, x )**;**

**}**

**double** Polynomal **::** Calc ( **double** x ) **{**

//共有函数：递归求多项式的值

Term \* pc = first->link**;**

**return** ( pc == first ) ? 0.0 **:** Value ( pc, 0, x )**;**

**}**

3-8 设a和b是两个用带有表头结点的循环链表表示的多项式。试编写一个算法，计算这两个多项式的乘积c = a\*b，要求计算后多项式a与b保持原状。如果这两个多项式的项数分别为n与m, 试说明该算法的执行时间为O(nm2)或O(n2m)。但若a和b是稠密的, 即其很少有系数为零的项, 那么试说明该乘积算法的时间代价为O(nm)。

【解答】

假设



则它们的乘积为



例如，a = 1 + 2x + 3x2 + 4x3 + 5x4, b = 6 + 7x + 8x2 + 9x3, 它们的乘积

c = (1+2x+3x2+4x3+5x4)\*(6+7x+8x2+9x3) =

= 1\*6 + (1\*7+2\*6)x +(1\*8+2\*7+3\*6)x2+(1\*9+2\*8+3\*7+4\*6)x3+

+ (2\*9+3\*8+4\*7+5\*6)x4+ (3\*9+4\*8+5\*7)x5+(4\*9+5\*8)x6+5\*9x7

在求解过程中，固定一个ai，用它乘所有bj，得到xi+j的系数的一部分。这是一个二重循环。

i = 0 **:**

1\*6

1\*7

1\*8

1\*9

i = 1 **:**

1\*6

1\*7+2\*6

1\*8+2\*7

1\*9+2\*8

2\*9

i = 2 **:**

1\*6

1\*7+2\*6

1\*8+2\*7+3\*6

1\*9+2\*8+3\*7

2\*9+3\*8

3\*9

i = 3 **:**

1\*6

1\*7+2\*6

1\*8+2\*7+3\*6

1\*9+2\*8+3\*7+4\*6

2\*9+3\*8+4\*7

3\*9+4\*8

4\*9

i = 4 **:**

3\*9+4\*8+5\*7

4\*9+5\*8

5\*9

1\*6

1\*8+2\*7+3\*6

1\*9+2\*8+3\*7+4\*6

2\*9+3\*8+4\*7+5\*6

1\*7+2\*6

根据以上思想得到的算法如下：

Polynomal **operator** \* ( Polynomal**&** a, Polynomal**&** b ) **{**

Polynomal temp**;**

Term \* pa = a.first->link, pb, pc, fc**;** //pa与pb是两个多项式链表的检测指针

**double** value**;** temp.first = fc = pc = **new** Term**;**

//fc是每固定一个ai时ai结点指针, pc是存放指针

**while** ( pa != NULL ) **{** //每一个ai与b中所有项分别相乘

pb = b.first->link**;**

**while** ( pb != NULL ) **{** //扫描多项式b所有项

value = pa->data \* pb->data**;** //计算ai \* bj

**if** ( pc->link != NULL ) pc->link->data = pc->link->data + value**;** //累加

**else** pc->link = **new** Term (value)**;** //增加项, 每次pa变化, 链结点要随之增加

pc = pc->link**;** pb = pb->link**;**

**}**

pc = fc = fc->link**;** pa = pa->link**;** //处理多项式a 的下一ai

**}**

pc->link = NULL**;**

**return** temp**;**

**}**

这个算法有一个二重循环，内层循环中语句的重复执行次数是O(n\*m)。其中，n是第一个多项式的阶数，m是第二个多项式的阶数。这是稠密多项式的情形。

对于稀疏多项式的情形请自行考虑。

3-9 计算多项式 Pn (x) = a0 xn + a1 xn-1 + a2 xn-2 + …… + an-1 x + an的值，通常使用的方法是一种嵌套的方法。它可以描述为如下的迭代形式：b0 = a0 , bi+1 = x \* bi + ai+1, i = 0, 1, …, n-1。若设bn = pn （x）. 则问题可以写为如下形式：Pn (x) = x \* Pn-1 (x) + an ，此处， Pn-1 (x) = a0 xn-1 + a1 xn-2 + …… + an-2 x + an-1，这是问题的递归形式。试编写一个递归函数，计算这样的多项式的值。

【解答】

如果用循环链表方式存储多项式，求解方法与3-7题相同。如果用数组方式存储多项式，当零系数不多时，可用顺序存放各项系数的一维数组存储多项式的信息，指数用数组元素的下标表示

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 |  | i |  | n-2 | n-1 |
| coef | a0 | a1 | a2 | a3 | … | ai | … | an-2 | an-1 |

多项式的类定义如下：

**struct** Polynomal **{**

**double** \* coef**;**

**int** n**;**

**}**

这样可得多项式的解法：

**double** Polynomal **::** Value ( **int** i, **double** x ) **{**

//私有函数：递归求子多项式的值

**if** ( i == n-1 ) **return** coef[n-1]**;**

**else return** coef[i] + x \* Value ( i+1, x )**;**

**}**

**double** Polynomal **::** Calc ( **double** x ) **{**

//共有函数：递归求多项式的值

**if** ( n == 0 ) **return** 0.0**;**

**else return** Value ( 0, x )**;**

**}**

3-10 试设计一个实现下述要求的Locate运算的函数。设有一个带表头结点的双向链表L，每个结点有4个数据成员：指向前驱结点的指针lLink、指向后继结点的指针rLink、存放数据的成员data和访问频度freq。所有结点的freq初始时都为0。每当在链表上进行一次Locate (L, x)操作时，令元素值为x的结点的访问频度freq加1，并将该结点前移，链接到与它的访问频度相等的结点后面，使得链表中所有结点保持按访问频度递减的顺序排列，以使频繁访问的结点总是靠近表头。

【解答】

**#include** <iostream.h>

**template <class Type> class** DblList**;**

**template <class Type> class** DblNode **{**

**friend class** DblList**<Type>;**

**private:**

**Type** data**;** //链表结点数据

DblNode**<Type>** \*lLink, \*rLink**;** //链表前驱(左链)、后继(右链)指针

**int** freq**;**  //访问频度

**public:**

DblNode ( **Type** value**,** DblNode**<Type>** \*left**,** DblNode**<Type>** \*right ) **:**

data (value)**,** lLink (left)**,** rLink (right), freq(0) **{ }** //构造函数

DblNode ( **Type** value ) **:** data (value)**,** lLink ( NULL )**,** rLink ( NULL ), freq(0) **{ }** //构造函数

**Type** getData ( ) **const { return** data**; }** //读取结点数据

**void** setData ( **Type** value ) **{** data = value**; }** //修改结点数据

DblNode**<Type>** \* getNext ( ) **{ return** rLink**; }**  //读取rLink中的地址

DblNode**<Type>** \* getPrior ( ) **{ return** lLink**; }** //读取lLink中的地址

**void** setNext ( DblNode**<Type>** \*p ) **{** rLink = p**; }** //修改rLink中的地址

**void** setPrior ( DblNode**<Type>** \*p ) **{** lLink = p**; }**  //修改lLink中的地址

**};**

**template <class Type> class** DblList **{**

**private:**

DblNode**<Type>** \*first**,** \*current**;** //双向循环链表表头指针和当前指针

**public:**

DblLIst ( **Type** uniqueVal )**;** //构造函数**:** 建立双向循环链表的表头结点

**int** Length ( ) **const;** //计算双向循环链表的长度

**int** IsEmpty ( ) **{** **return** first->rLink == first**; } //**判双向循环链表空否

**int** Locate ( **Type&** target ) **const;** //在链表中寻找等于给定值的结点

DblNode**<Type>** \*Firster ( ) **{** current = first**; return** first**; }**

//初始化**:** 将当前指针指到表头结点

**Type** First ( ) **{** current = first->rLink**; return** current->data**; }**

//当前指针指向链表第一结点

**Type** RLink ( ) **{** //当前指针指到当前结点的后继结点

current = ( current->rLink == first ) **?** first->rLink **:** current->rLink**;**

**return** current->data**;**

**}**

**Type** LLink ( ) **{** //当前指针指到当前结点的前驱结点

current = ( current->lLink == first ) **?** first->lLink **:** current->lLink**;**

**return** current->data**;**

**}**

**void** Insert ( **const Type &** value )**;** //在当前结点的rLink方向插入一个含值value的新结点

**void** Remove ( )**;** //删除当前结点

**friend istream& operator** >> ( **istream&** in, DblList**&** DL )**;** //重载操作：输入链表

**friend ostream& operator** << ( **ostream&** out, DblList**&** DL )**;** //重载操作：输出链表

**};**

**template <class Type>** DblList**<Type> ::** DblList ( **Type** uniqueVal ) **{**

first = **new** DblNode<**Type**> ( uniqueVal)**;** //创建表头结点, uniqueVal是输入结束标志

first->lLink = first->rLink = first**;** current = first**;** //当前指针指向表头结点

**}**

**template <class Type> void** DblList**<Type> ::** Insert ( **const Type &** value ) **{**

//建立一个包含有值value的新结点, 并将其插入到当前结点之后。

**if** ( first->rLink == first ) //原为空表

current = first->rLink = **new** DblNode ( value, first, first )**;**

**else** current = current->rLink = **new** DblNode ( value, current, current->rLink )**;** //原为非空表

current->rLink->lLink = current**;** //完成重新链接

**}**

**template <class Type> int** DblList**<Type> ::** Length ( ) **const {**

//计算带表头结点的双向循环链表的长度, 通过函数返回。

DblNode**<Type>** \* p = first->rLink**;** **int** count = 0**;**

**while** ( p != first ) **{** p = p->rLink**;**  count++**;** **}**

**return** count**;**

**}**

**template <class Type> istream& operator** >> ( **istream&** in, DblList<**Type**>**&** DL ) **{**

**Type** value**;** in >> value**;** //每次新结点插入在表头结点后面

**while** ( value != DL.first->data ) **{** DL.Insert ( value )**;** in >> value**; }**

**}**

**template <class Type> ostream& operator** << ( **ostream&** out, DblList<**Type**>**&** DL ) **{**

**int** i**,** n = DL.Length( )**; Type** value = DL.First ( )**;**

**for** ( i = 0**;** i < n**;** i++ ) **{** out **<<** value << **endl;** value = DL.RLink( )**; }**

**}**

**template <class Type> int** DblList<**Type**> **::** Locate ( **Type&** x ) **{**

//在双向循环链表中查找值为x的结点，找到后该结点成为当前结点，再将它搬到适当位置

DblNode<**Type**> \*p = first->rLink**;**

**while** ( p != first **&&** p->data != x ) p = p->rLink**;**

**if** ( p != first ) **{** //链表中存在x

p→freq++**;** current = p**;** //该结点的访问频度加1

current->lLink->rLink = current->rLink**;** //从链表中摘下这个结点

current->rLink->lLink = current->lLink**;**

p = current->lLink**;** //寻找从新插入的位置

**while** ( p != first **&&** current->freq > p->freq ) p = p->lLink**;**

current->rLink = p->rLink**;** //插入在p之后

current->lLink = p**;**

p->rLink->lLink = current**;**

p->rLink = current**;**

**}**

**else cout**<<"Sorry. Not find!\n"**;** //没找到

**}**

3-11 利用双向循环链表的操作改写2-2题，解决约瑟夫(Josephus)问题。

【解答】

**#include** <iostream.h>

**#include** “DblList.h”

**Template <class Type> void** DblList **<Type> ::** Josephus ( **int** n, **int** m ) **{**

DblNode<**Type>** p = first, temp**;**

**for** ( **int** i = 0**;** i < n-1**;** i++ ) **{** //循环n-1趟，让n-1个人出列

**for** ( **int** j = 0**;** j < m-1**;** j++ ) p = p->rLink**;** //让p向后移动m-1次

**cout** << “Delete person ” << p->data << **endl;**

p->lLink->rLink = p->rLink**;** //从链中摘下p

p->rLink->lLink = p->lLink**;**

temp = p->rlink**; delete** p**;** p = temp**;** //删除p所指结点后, p改指下一个出发点

**}**

**cout** << “The winner is ” << p->data << **endl;**

**}**

**void** main ( ) **{**

DblList<**int**> dlist**;** //定义循环链表dlist并初始化

**int** n, m**;** //n是总人数，m是报数值

**cout** << “Enter the Number of Contestants?”**;**

**cin** >> n >> m**;**

**for** ( **int** i = 1**;** i <= n**;** i++ ) dlist**.**insert (i)**;** //建立数据域为1, 2, … 的循环链表

dlist**.**Josephus (n, m)**;** //解决约瑟夫问题，打印胜利者编号

**}**

3-12 试设计一个算法，改造一个带表头结点的双向链表，所有结点的原有次序保持在各个结点的rLink域中，并利用lLink域把所有结点按照其值从小到大的顺序连接起来。

【解答】

**template<Type> void** DblList<**Type> ::** sort ( ) **{**

DblNode<**Type**> \* s = first->link**;** //指针s指向待插入结点, 初始时指向第一个结点

**while** ( s != NULL ) **{** //处理所有结点

pre = first**;**  p = first->lLink**;** //指针p指向待比较的结点, pre是p的前驱指针

**while** ( p != NULL **&&** s->data < p->data ) //循lLink链寻找结点 \*s的插入位置

**{** pre = p; p = p->lLink**; }**

pre->lLink = s**;** s->lLink = p**;** //结点 \*s在lLink方向插入到 \*pre与 \*p之间

**}**

**}**

**四、其他练习题**

3-13 设单链表中结点的结构为（data, link）。已知指针q所指结点是指针p所指结点的直接前驱，若在\*q与\*p之间插入结点\*s，则应执行下列哪一个操作？

(1) s->link = p->link**;** p->link = s**;** (2) q->link = s**;** s->link = p**;**

(3) p->link = s->link**;** s->link = p**;** (4) p->link = s**;** s->link = q**;**

【解答】 (2)

3-14 设单链表中结点的结构为（data, link）。已知指针p所指结点不是尾结点，若在\*p之后插入结点\*s，则应执行下列哪一个操作？

(1) s->link = p**;** p->link = s**;** (2) s->link = p->link**;** p->link = s**;**

(3) s->link = p->link**;** p = s**;** (4) p->link = s**;** s->link = p**;**

【解答】 (2)

3-15 设单链表中结点的结构为（data, link）。若想摘除结点\*p的直接后继，则应执行下列哪一个操作？

(1) p->link = p->link->link**;** (2) p = p->link**;** p->link = p->link->link**;**

(3) p->link = p->link**;** (4) p = p->link->link**;**

【解答】 (1)

3-16 设单循环链表中结点的结构为（data, link），且rear是指向非空的带表头结点的单循环链表的尾结点的指针。若想删除链表第一个结点，则应执行下列哪一个操作？

1. s = rear**;** rear = rear->link**;** **delete** s**;**
2. rear = rear->link**;** **delete** rear**;**
3. rear = rear->link->link**;** **delete** rear**;**

(4)s = rear->link->link**;** rear->link->link = s->link**; delete** s**;**

【解答】 (4)

3-17 设双向循环链表中结点的结构为（data, lLink, rLink），且不带表头结点。若想在指针p所指结点之后插入指针s所指结点，则应执行下列哪一个操作？

1. p->rLink = s**;** s->lLink = p**;** p->rLink->lLink = s**;** s->rLink = p->rLink**;**
2. p->rLink = s**;** p->rLink->lLink = s**;** s->lLink = p**;** s->rLink = p->rLink**;**
3. s->lLink = p**;** s->rLink = p->rLink**;** p->rLink = s**;** p->rLink->lLink = s**;**

(4) s->lLink = p**;** s->rLink = p->rLink**;** p->rLink->lLink = s**;** p->rLink = s**;**

【解答】 (4)

3-18 判断一个带表头结点的双向循环链表L是否对称相等的算法如下所示，请在算法中的 处填入正确的语句。

**int** symmetry ( DblList DL ) **{**

**int** sym = 1**;**

DblNode \* p = DL->rLink**,** \*q = DL->lLink**;**

**while** ( ( p != q || p->lLink *==* q ) **&&** ① )

**if** ( p->data *==* q->data ) **{**

② **;**

③ **;**

**}**

**else** sym = 0**;**

**return** sym**;**

**}**

【解答】

① sym *==* 1 ② p = p->rLink ③ q = q->lLink

3-19 根据两个有序单链表生成一个新的有序单链表，原有单链表保持不变。如假定两个有序单链表中的元素为（2,8,10,20）和（3,8,9,15,16），则生成的新单链表中的元素为（2,3,8,8,9,10,15,16,20）。（单链表定义参看题 3.3）

【解答】

List**<Type>** Merge ( List**<Type>&** L1**,** List<**Type>&** L2 ) **{**

//根据两个有序单链表L1和L2, 生成一个新的有序单链表

List<**Type**> temp**;** //用temp作为新的有序单链表

ListNode<**Type**> \*p1 = L1.First( )**,** \*p2 = L2.First( )**,** \*p = temp.Firster( )**,** \*newptr**;**

//指针p1与p2指向两个有序单链表的第一个结点，p指向结果有序单链表的表尾结点

**while** ( p1 != NULL **&&** p2 != NULL ) **{** //当两个链表未检测完时

**if** ( p1->getData( ) <= p2->getData( ) ) //用p1->data建立新结点, p1指针后移

**{** newptr = **new** ListNode<**Type>** (p1->getData ( ) )**;** p1 = p1->getLink( )**; }**

**else** //用p2->data建立新结点, p2指针后移

**{** newptr = **new** ListNode<**Type>** (p2->getData( ) )**;** p2 = p2->getLink( )**; }**

p->setLink ( newptr )**;**  p = newptr**;** //将newptr结点插入到结果链表的表尾

**}**

**while** ( p1 != NULL ) **{** //继续处理p1链表中剩余的结点。

newptr = **new** ListNode<**Type>** (p1->getData ( ) )**;** p1 = p1->getLink( )**;**

p->setLink ( newptr )**;**  p = newptr**;**

**}**

**while** ( p2 != NULL ) **{** //继续处理p2链表中剩余的结点。

newptr = **new** ListNode<**Type>** (p2->getData ( ) )**;** p2 = p2->getLink( )**;**

p->setLink ( newptr )**;**  p = newptr**;**

**}**

p->setLink (NULL)**;**

**return** temp**;**

**}**

3-20 根据一个结点数据类型为整型的单链表生成两个单链表，使得第一个单链表中包含原单链表中所有数据值为奇数的结点，使得第二个单链表中包含原单链表中所有数据值为偶数的结点，原有单链表保持不变。（单链表定义参看题3.2）

【解答】

**void** Separate ( List**<int>&** HL, List**<int>&** L1, List**<int>&** L2 ) **{**

//将一个单链表HL按各个结点中数据的奇偶性拆分成两个单链表L1和L2

ListNode<**int**> \*t1 = L1.Firster( )**,** \*t2 = L2.Firster( )**,** \*p**;**

//t1和t2分别指向L1和L2的表尾

p = HL.First ( )**;** //链表HL的扫描指针, 开始指向链表第一个结点

**while** ( p != NULL ) **{** //循环, 产生新结点, 把它加入到L1或L2链表的末尾

**int** value = p->getData( )**;**

ListNode<**int**> \*newptr = **new** ListNode<**int**> ( value )**;**

**if** ( value % 2 *==* 1 ) **{** t1->setLink (newptr)**;** t1 = newptr**; }** //奇数, 链入L1链的链尾

**else** **{** t2->setLink (newptr)**;** t2 = newptr**; }** //偶数, 链入L2链的链尾

p = p->getLink( )**;**

**}**

t1->setLink (NULL)**;** t2->setLink (NULL)**;** //链收尾

**}**

3-21 已知一个带表头结点的单链表中包含有三类字符（数字字符、字母字符和其他字符），试编写一个函数，构造三个新的单链表，使每个单链表中只包含同一类字符。要求使用原表的空间，表头结点可以另辟空间。（单链表定义参看题 3.2）

【解答】

**#include** <ctype.h>

**void** Separate ( List<**char>&** LA**,** List**<char>&** LB**,** List**<char>&** LC ) **{**

//原来的单链表是LA, 新的三个单链表是LA, LB, LC

ListNode<**char**> \*p = LA.First ( )**;**  //指针p指向原表第一个结点

ListNode<**char**> \*pa = LA.Firster( )**,** \*pb = LB.Firster( )**,** \*pc = LC.Firster( )**;**

//指针pa, pb, pc分别是三个结果链表的链尾指针, 初始时指向各表头结点

**while** ( p != NULL ) **{**

**if** ( isdigit ( p->getData ( ) ) ) //是数字字符

**{** pa->setLink ( p )**;** pa = pa->getLink( )**; }**

**else if** ( isalpha ( p->getData( ) ) ) //是字母字符

**{** pb->setLink ( p )**;** pb = pb->getLink( )**; }**

**else {** pc->setLink( p )**;** pc = pc->getLink( ); } //其他字符

p = p->getLink( )**;**

**}**

pa->setLink(NULL)**;** pb->setLink(NULL)**;** pc->setLink(NULL)**;**

**}**

3-22 设在一个带表头结点的单链表中所有元素结点的数据值按递增顺序排列，试编写一个函数，删除表中所有大于min，小于max的元素（若存在）。（单链表定义参看题 3.3）

【解答】

**void** rangeDelete ( List<**Type**> L**,** **Type** min**,** **Type** max ) **{**

ListNode<**Type**> \*pr = L.Firster( )**,** \*p = L.First( )**;** //指针p指向第一个结点, pr是其前驱

**while** ( p != NULL **&&** p->getData( ) <= min ) //寻找开始删除位置

**{** pr = p**;** p = p->getLink( )**; }**

**while** ( p != NULL **&&** p->getData( ) < max ) //删除

**{** pr->setLink( p->getLink( ) )**; delete** p**;** p = pr->getLink( )**; }**

**}**

3-23 设在一个带表头结点的单链表中所有元素结点的数据值无序排列，试编写一个函数，删除表中所有大于min，小于max的元素（若存在）。（单链表定义参看题 3.2）

【解答】

**void** rangeDelete ( List<**Type**> L**,** **Type** min**,** **Type** max ) **{**

ListNode<**Type**> \*pr = L.Firster( )**,** \*p = L.First( )**;** //指针p指向第一个结点, pr是其前驱

**while** ( p != NULL )

**if** ( p->getData( ) > min **&&** p->getData( ) < max ) //寻找到删除结点

**{** pr->setLink( p->getLink( ) )**; delete** p**;** p = pr->getLink( )**; }**

**else {** pr = p**;** p = p->getLink( )**; }** //否则继续寻找被删结点

**}**

3-24 试设计一个算法，改造一个带表头结点的双向链表，所有结点的原有次序保持在各个结点的右链域rLink中，并利用左链域lLink把所有结点按照其值从小到大的顺序连接起来。

【解答】

**template<class Type>** **void** OrderedList ( DblList<**Type**> DL ) **{**

//关于DblList和DblNode类的定义请参照双向循环链表的定义，DL是链表对象

DblNode<**Type**> \* endp = DL.Firster ( )**,** \* pr**,** \* p**,** \*s**,** \*q = endp->getPrior ( )**;**

s = q->getPrior ( )**;** q->setPrior ( endp )**;** //表头结点与其前驱结点形成循环链表

**while** ( s != endp ) **{** //循环, 把 \*s结点按序插入到以lLink域链接的有序表中

pr = endp**;** p = pr->getPrior ( )**;** //为\*s结点寻找合适的插入位置

**while** ( p != endp )

**if** ( p->getData ( ) > s->getData ( ) ) **break;**

**else {** pr = p**;** p = p->getPrior ( )**; }** //插入位置在pr和p之间，把\*s结点插入其中

q = s->getPrior( )**;** pr->setPrior ( s )**;** s->setPrior ( p )**;**

s = q**;**

**}**

**}**

3-25 利用数组可以实现静态链表结构。与动态链表结构相似，需要建立链表结点类（staticListNode）和链表类（staticLinkList）。例如，下面给出一个单链表：



对应的静态链表如下：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 75 | 62 | 83 | 44 | 57 | 94 | 50 | 68 |  |  |
| 4 | 3 | 8 | 6 | 7 | 2 | 0 | 5 | 1 |  |  |

0 1 2 3 4 5 6 7 8 ... MaxSize-1

data

link

0号数组元素是链表的表头结点，其link域中保存指向链表第一个元素结点的指针，1号到MaxSize-1号结点可存放链表的元素，它们的逻辑顺序可通过各结点的link指针链接起来。初始时，从1号结点到MaxSize-1号结点构成链接的可利用空间表，利用它们实行结点的分配和释放。link为0表示空指针。试给出静态链表的类定义及其成员函数的实现。

**template <class Type> class** staticLinkList**;** //静态链表类的前视声明

**template <class Type> class** staticListNode **{** //静态链表元素类的定义

**friend class** staticLinkList**<Type>;**

**private:**

**Type** data**;** //数据域

**int** link**;** //链接指针域

**public:**

**Type** getData ( ) **{ return** data**; }** //取当前结点的数据值

**void** setData ( **Type** value ) **{** data = value**; }** //将当前结点的数据值修改为value

**int** getLink ( ) **{** **return** link**; }** //取当前结点的链接指针

**void** setLink ( **int** ptr ) **{** link = ptr**;** **}** //将当前结点的链接指针置为ptr

**}**

**template <class Type> class** staticLinkList **{** //静态链表的类定义

**private:**

staticListNode <**Type**> \*Vector**;** //存储待排序元素的向量

**int** MaxSize**; //**向量中最大元素个数

**int** last**;** //链表表尾地址

**int** CurrentStart**;** //可利用空间表首地址

**int** newNode ( )**;** //从可利用空间表分配一个结点

**void** deleteNode ( **int** i )**;** //将结点i回收到可利用空间表中

**public:**

staticLinkList ( **int** Maxsz )**;** //构造函数

**void** makeEmpty ( )**;** //置空表

int Length ( ) **const;** //取得链表长度

**int** IsEmpty ( ) **const { return** Vector[0].link == 0**; }** //判断链表空否

**int** IsFull ( ) **const { return** Length( ) == MaxSize**; }** //判断链表满否

**int** Locate ( **int** i ) **const;** //定位于链表第i个位置

**int** Find ( **Type** x ) **const;** //在链表中查找包含x的结点位置

**int** Update ( **int** i, **const Type&** x )**;** //将链表中第i个元素的值更新为x

**int** InsertFront ( **const Type&** x )**; //**在链表的表头插入一个新元素x

**int** InsertRear ( **const Type&** x )**; //**在链表的表尾添加一个新元素x

**int** Insert ( **int** i**, const Type&** x )**;** //向链表第i个位置插入一个元素x

**Type** DeleteFront ( )**; //**从链表中删除表中第一个元素

**int** DeleteItem ( **const Type&** x )**;** //从链表中删除等于给定值x的第一个元素

**void** Traverse ( )**; //**遍历一个单链表

**}**

**int** staticLinkList**<Type> ::** newNode ( ) **{**

**//**私有函数：从可利用空间表分配一个结点, 若分配失败则函数返回0, 否则返回1

**if** ( CurrentStart == 0 ) **return** 0**;**

**else { int** temp = CurrentStart**;** CurrentStart = Vector[CurrentStart].link**; return** temp**; }**

**}**

**void** staticLinkList**<Type> ::** deleteNode ( **int** i ) **{**

**//**私有函数：将结点i回收到可利用空间表中

Vector[i] = CurrentStart**;** CurrentStart = i**;**

**}**

staticLinkList**<Type> ::** staticLinkList ( **int** Maxsz ) **{**

//构造函数, 初始化单链表

MaxSize = Maxsz**;** CurrentStart = 1**;**

Vector = **new** staticListNode <**Type**> [Maxsz]**;**

Vector[0].link = 0**;** last = 0**;**

//将单链表置空, 此数值0又表示下标0, 由此构成带表头结点

//的空的循环单链表，此下标为0结点就是表头结点

**for** ( **int** i = 1**;** i < MaxSize-1**;** i++ ) Vector[i].link = i+1**;** //构成可利用空间表

Vector[MaxSize-1].link = 0**;** //将可利用空间的最后一个结点的指针置0, 表示空指针

**}**

**void** staticLinkList**<Type> ::** MakeEmpty ( ) **{**

//删除单链表中的所有结点，使之成为一个空表

MaxSize = Maxsz**;** CurrentStart = 1**;**

Vector[0].link = 0**;** last = 0**;**

**for** ( **int** i = 1**;** i < MaxSize-1**;** i++ ) Vector[i].link = i+1**;**

Vector[MaxSize-1].link = 0**;**

**}**

**int** staticLinkList**<Type> ::** Length ( ) **const** **{**

**//**取得单链表的长度

**int** p = Vector[0].link**,** i = 0**;**

**while** ( p != 0 ) **{** i++**;**  p = Vector[p].link**; }**

**return** i**;**

**}**

**int** staticLinkList**<Type> ::** Locate ( **int** i ) **const** **{**

//定位于链表第i个位置, 若i太大或太小则函数返回-1, 否则返回第i 个结点位置

**if** ( i < 0 ) **{ cerr** << "pos is out range!" << **endl; return -**1**; }**

**else if** ( i == 0 ) **return** 0**;**

**int** p = Vector[0].link**,** j = 1**;**

**while** ( p !=0 **&&** j < i ) **{** j++**;** p = Vector[p].link**; }**

**if** ( p != 0 ) **return** p**;**

**else { cerr** << "i is out range!" << **endl; return -**1**; }**

**}**

**int** staticLinkList**<Type> ::** Find ( **Type** x ) **const** **{**

**//**在单链表中查找具有给定值x的第一个元素的结点位置, 查找失败函数返回0

**int** p = Vector[0].link**;**

**while** ( p != 0 )

**if** ( Vector[p].data == x ) **return** p;

**else** p = Vector[p].link**;**

**return** 0**;**

**}**

**int** staticLinkList**<Type> ::** Update ( **int** i, **const Type&** x ) **{**

//将链表中第i个元素的值更新为x, 若给定的i值不合理则函数返回0, 否则返回1

**int** p = Locate ( i )**;**

**if** ( p != 0 **&&** p != -1 ) **{** Vector[p].data == x**; return** 1**; }**

**else return** 0**;**

**}**

**int** staticLinkList**<Type> ::** InsertFront ( **const Type&** x ) **{**

//在链表的表头插入一个新元素x, 插入成功时函数返回1, 否则返回0

**int** newptr = newNode ( )**;** //从可利用空间表中取出一个结点

**if** ( !newptr ) **{ cerr** << "No empty node!" << **endl; return** 0**; }**

Vector[newptr].data = x**;**

Vector[newptr].link = Vector[0].link**;** //新结点插入到表头结点后面

Vector[0].link = newptr**;**

**if** ( last == 0 ) last = newpr**;**

**return** 1**;**

**}**

**int** staticLinkList**<Type> ::** InsertRear ( **const Type&** x ) **{**

**//**在链表的表尾添加一个新元素x, 插入成功时函数返回1, 否则返回0素

**int** newptr = newNode ( )**;** //从可利用空间表中取出一个结点

**if** ( !newptr ) **{ cerr** << "No empty node!" << **endl; return** 0**; }**

Vector[newptr].data = x**;**

Vector[last].link = newptr**;**  last = newptr**;**  //新结点链接到表尾结点后面

Vector[newptr].link = 0**;** //新表尾收尾

**return** 1**;**

**}**

**int** staticLinkList**<Type> ::** Insert ( **int** i**, const Type&** x ) **{**

//向单链表第i个位置插入一个元素

**int** p = Locate ( i-1 )**;** //寻找插入位置

**if** ( p == -1 ) **{ cerr** << "Invalid Insert Position!" << **endl;**  **return** 0**; }**

**int** newptr = newNode ( )**;** //从可利用空间表中取出一个结点

**if** ( !newptr ) **{ cerr** << "No empty node!" << **endl; return** 0**; }**

Vector[newptr].data = x**;**

Vector[newptr].link = Vector[p].link**;** //新结点链接到第i-1号结点之后

Vector[p].link = newptr**;**

**if** ( p == last ) last = newptr**;**

**}**

**Type** staticLinkList**<Type> ::** DeleteFront ( ) **{**

**//**从单链表中删除第一个元素, 通过函数返回

**if** ( Vector[0].link == 0) **{ cerr** << "Deleting from an empty list!" << **endl; exit** (1)**; }**

**int** p = Vector[0].link**;** //取表头结点

Vector[0].link = Vector[p].link**;** //使第二个结点成为新表头结点。

deleteNode ( p )**;** //把被删结点插入到可利用空间表中

**if** ( p = last ) last = 0**;**

**return** Vector[p].data**;** //返回原表头结点的元素

**}**

**int** DeleteItem ( **const Type&** x ) **{**

//从单链表中删除等于给定值x的第一个元素

**if** ( Vector[0].link == 0) **{ cerr** << "Linkedlist is empty!" << **endl; return** 0**; }**

**int** pr = 0**,** p = Vector[0].link**;**  //查找被删结点及前驱结点位置

**while** ( p != 0 && Vector[p].data != x ) **{** pr = p**;**  p = Vector[p].link**; }**

**if** ( !p ) **{ cerr** << "Deleted element is not exist!" **<< endl; return** 0**; }**

//若不存在被删除的元素则返回0

Vector[pr].link = Vector[p].link**;** //从单链表中摘下被删结点p

deleteNode ( p )**;** //把被删结点插入到可利用空间表中

**if** ( p = last ) last = pr**;**

**return** 1**;**

**}**

**void** staticLinkList**<Type> ::** Traverse ( ) **{**

**//**遍历一个单链表

**int** p = Vector[0].link**;**

**while** ( p != 0 ) **{**

**cout** << Vector[p].data << " "**;**

p = Vector[p].link**;**

**}**

**cout** << **endl;**

**}**

利用数组建立单链表的好处是：可以把建立好单链表的数组作为文件存储起来，以后再把文件读入到内存后，可以继续使用原有单链表，不必重新建立。