class LinkList //单链表类

{

#region 链表结构的结点通过类来实现

class Node //结点类

{

#region 结点类的属性

public object data; //结点的数据域（公有、对象型）

public Node next; //结点的指针域（公有、结点型）

#endregion

#region 结点类的系列构造函数

public Node() //创建空结点

{

this.data = null;

this.next = null;

}

public Node(object e) //重载：创建带有数据的结点

{

this.data = e;

this.next = null;

}

#endregion

}

#endregion

#region 链表的字段及属性

private Node head; //链表的头结点

private int length; //链表的长度，即除去头结点的结点个数

public int Length //链表的长度属性只读

{

get { return length; }

}

#endregion

#region 单链表基本算法

#region 1.初始化

public LinkList() //创建空链表

{

this.head = new Node();

this.length = 0;

}

#endregion

#region 2.查询信息

public object getElem(int location) //单链表的按位置 查找 操作,返回所在位置的结点数据域，位置索引从“1”开始

{

object result = null;

if (location <= 0 || location > this.length) //如果查找位置超出链表范围，则无数据域返回，即返回null

return result;

Node p = this.head.next;

for (int i = 1; i < location; i++) //使指针指向制定位置的结点

{

p = p.next;

}

result = p.data;

return result;

}

public string getLocation(object Elem) //单链表的按数值 查找 操作，返回存在个数及其位置信息，位置索引从“1”开始

{

int count = 0; //用以记录数值个数

int location = 0;

string Loc = ""; //用以记录数值位置信息

Node p = this.head.next;

while (p != null)

{

location++;

if (p.data.ToString() == Elem.ToString())

{

count++;

if (count == 1)

Loc += location.ToString();

else

Loc += "、" + location.ToString();

}

p = p.next;

}

if (count == 0)

return "当前链表无该数据";

if(count==1)

return "在当前链表中，" + Elem.ToString() + "有" + count.ToString() + "个，位于第" + Loc + "个长度位置";

return "在当前链表中，" + Elem.ToString() + "有" + count.ToString() + "个，分别位于第" + Loc + "个长度位置";

}

#endregion

#region 3.插入

public bool Insert(object Elem, int Location) //单链表的 插入 操作,位置索引从“1”开始

{

if (Location < 1 || Location > this.length + 1) //判断插入位置是否合法

return false;

int loc = 1;

Node p = this.head;

while (loc < Location) //寻找需要插入位置是前一个位置

{

p = p.next;

loc++;

}

Node q = new Node(Elem); //需要插入的结点

q.next = p.next;

p.next = q;

this.length++;

return true;

}

#endregion

#region 4.删除

public bool DeleteByLocation(int Location) //单链表的 删除 操作，删除特定具体位置的数据，位置索引从“1”开始

{

if (Location < 1 || Location > this.length) //判断删除位置是否合法

return false;

int loc = 1;

Node p = this.head;

while (loc < Location) //寻找所需删除位置的前一个位置

{

p = p.next;

loc++;

}

Node q = p.next;

p.next = q.next;

q = null;

length--;

return true;

}

public bool DeleteByLocation(int minLocation, int maxLocation) //单链表的 删除 操作，删除特点范围位置的数据s，位置索引从“1”开始

{

if (minLocation < 1 || maxLocation > this.length || minLocation > maxLocation) //判断删除范围是否合理

return false;

int minloc = 1, maxloc;

Node p = this.head;

while (minloc < minLocation)

{

p = p.next;

minloc++;

}

maxloc = minloc;

Node q = p.next;

while (maxloc < maxLocation)

{

q = q.next;

maxloc++;

}

p.next = q.next;

q = null;

length -= (maxLocation - minLocation + 1);

return true;

}

public int DeleteElems(object Elem) //单链表的 删除 操作，删除特定具体的数据,返回删除个数，位置索引从“1”开始

{

int delelteCount = 0;

Node p = this.head;

while (p.next != null)

{

if (p.next.data.ToString() == Elem.ToString())

{

Node q = p.next;

p.next = q.next;

q = null;

length--;

delelteCount++;

continue;

}

p = p.next;

}

return delelteCount;

}

public bool DeleteElems(object minElem, object maxElem) //单链表的 删除 操作，删除特定范围的数据，位置索引从“1”开始

{ //仅针对数值型

if (Convert.ToDecimal(minElem) > Convert.ToDecimal(maxElem))

return false;

Node p = this.head;

while (p.next != null)

{

if (Convert.ToDecimal(p.next.data) >= Convert.ToDecimal(minElem) &&

Convert.ToDecimal(p.next.data) <= Convert.ToDecimal(maxElem))

{

Node q = p.next;

p.next = q.next;

q = null;

length--;

continue;

}

p = p.next;

}

return true;

}

#endregion

#region 5.判断是否为空

public bool IsEmpty()

{

if (this.head.next != null)

return false;

else

return true;

}

#endregion

#region 6.打印输出

public string Print()

{

string result = "";

Node p = this.head.next;

while (p != null)

{

if (p == this.head.next)

result += p.data.ToString();

else

result += "," + p.data.ToString();

p = p.next;

}

return result;

}

#endregion

#region 7.找出最大结点

public string findmax()

{

Node p = head.next;

Node q = head.next;

int j = 1;

while (p != null)

{

if (Convert.ToDecimal(q.data) < Convert.ToDecimal(p.data))

{

q = p;

j++;

}

p = p.next;

}

return q.data.ToString();

}

public string getLocationa(object Elem)

{

int location = 0;

string Loc = "";

Node p = this.head.next;

while (p != null)

{

location++;

if (p.data.ToString() == Elem.ToString())

{

Loc = location.ToString();

}

p = p.next;

}

return Loc;

}

#endregion

#region 8.求两个链表的交集

public void mergejiao(LinkList l2)

{

Node p = this.head.next;

Node q = l2.head.next;

bool aaa = true;

while (p != null && q != null)

{

if (Convert.ToDecimal(p.data) == Convert.ToDecimal(q.data))

{

if (aaa)

{

head = new Node();

length = 0;

}

aaa = false;

this.insert(length + 1, p.data);

p = p.next;

q = q.next;

}

else if (Convert.ToDecimal(p.data) < Convert.ToDecimal(q.data))

{

p = p.next;

}

else

{

q = q.next;

}

}

}

public void quchong()

{

Node p = this.head.next;

int i = 1;

while (p != null)

{

Node q = p.next;

while (q != null)

{

int j = i + 1;

if (Convert.ToDecimal(q.data) == Convert.ToDecimal(p.data))

{

this.delete(j);

}

q = q.next;

j++;

}

p = p.next;

i++;

}

}

public bool delete(int i)

{

Node p = head;

int j = 0;

if (i < 1 || i > length)

return false;

while (j < i - 1)

{

p = p.next;

j++;

}

Node q = new Node();

q = p.next;

p.next = q.next;

return true;

}

#endregion

#endregion

#region 链表的排序算法 从小到大

#region 直接插入排序

public void StraightInsertionSort()

{

Node p = this.head.next;

while (p.next != null)

{

//decimal stop = 0;

if (Convert.ToDecimal(p.data) > Convert.ToDecimal(p.next.data))

{

Node q = p.next;

p.next = q.next;

length--;

q.next = null;

Node t = this.head;

while (Convert.ToDecimal(t.next.data) < Convert.ToDecimal(q.data))

{

t = t.next;

}

q.next = t.next;

t.next = q;

length++;

q = null;

continue;

}

p = p.next;

}

}

#endregion

#region 冒泡排序

public void BubbleSort()

{

Node p = this.head;

int flag = 1;

while (p !=this.head.next && flag == 1)

{

flag = 0;

Node q = this.head;

while (q.next.next != null)

{

if (Convert.ToDecimal(q.next.data) > Convert.ToDecimal(q.next.next.data))

{

Node stop = new Node(q.next.next.data);

q.next.next = q.next.next.next;

stop.next = q.next;

q.next = stop;

stop = null;

flag = 1;

}

q = q.next;

}

p = q.next;

}

}

#endregion

#region 简单选择排序

public void SimpleSelectionSort()

{

Node p = this.head;

int minLoc = 1;

int firstLoc = 1, nowLoc = 1;

while (p.next != null)

{

Node q = p.next;

decimal minElem = Convert.ToDecimal(q.data);

while (q.next != null)

{

if (minElem > Convert.ToDecimal(q.next.data))

{

minElem = Convert.ToDecimal(q.next.data);

nowLoc++;

minLoc = nowLoc;

}

else

nowLoc++;

q = q.next;

}

if (minLoc != firstLoc)

this.Exchange(firstLoc, minLoc);

firstLoc++;

nowLoc = firstLoc;

minLoc++;

p = p.next;

}

}

#endregion

#region 归并排序

public void MergingSort()

{

}

#endregion

#endregion

#region 单链表进阶算法

#region 链表翻转

public void Filp()

{

Node p = this.head.next;

if (p == null)

return;

while (p.next != null)

{

Node q = p.next;

p.next = q.next;

q.next = this.head.next;

this.head.next = q;

}

}

#endregion

#region 结点位置交换

public void Exchange(int Location\_1, int Location\_2)

{

if (Location\_1 < 1 || Location\_1 > this.length || Location\_2 < 1 || Location\_2 > this.length||Location\_1==Location\_2)

return;

Node BfLoc\_1 = this.head;

Node BfLoc\_2 = this.head;

for (int i = 0; i < Location\_1 - 1; i++)

BfLoc\_1 = BfLoc\_1.next;

for (int i = 0; i < Location\_2 - 1; i++)

BfLoc\_2 = BfLoc\_2.next;

Node loc\_1 = BfLoc\_1.next;

Node loc\_2 = BfLoc\_2.next;

if (loc\_1 == BfLoc\_2 || loc\_2 == BfLoc\_1)

{

loc\_1.next = loc\_2.next;

loc\_2.next = loc\_1;

BfLoc\_1.next = loc\_2;

return;

}

BfLoc\_1.next = loc\_1.next;

loc\_1.next = null;

BfLoc\_2.next = loc\_2.next;

loc\_2.next = null;

loc\_1.next = BfLoc\_2.next;

BfLoc\_2.next = loc\_1;

loc\_2.next = BfLoc\_1.next;

BfLoc\_1.next = loc\_2;

}

#endregion

#region 链表的合并

public bool Merge(LinkList L1,LinkList L2)

{

if (!IsEmpty())

return false;

Node p = L1.head.next;

Node q = L2.head.next;

Node k = this.head;

while (p != null && q != null)

{

if (Convert.ToDecimal(p.data) < Convert.ToDecimal(q.data))

{

k.next = p;

length++;

k = k.next;

p = p.next;

}

else

{

k.next = q;

length++;

k = k.next;

q = q.next;

}

}

while (p != null)

{

k.next = p;

length++;

k = k.next;

p = p.next;

}

while (q != null)

{

k.next = q;

length++;

k = k.next;

q = q.next;

}

return true;

}

#endregion

#region 递增有序链表的合并（表中不许有重复数据）

public void Merge(LinkList lst)

{

Node p = this.head;

Node q = lst.head;

while (p.next != null && q.next != null)

{

if (Convert.ToDecimal(p.next.data) > Convert.ToDecimal(q.next.data))

{

Node t = new Node(q.next.data);

t.next = p.next;

p.next = t;

p = p.next;

q = q.next;

}

else if (Convert.ToDecimal(p.next.data) == Convert.ToDecimal(q.next.data))

q = q.next;

else

p = p.next;

}

if (q.next != null)

p.next = q;

this.length += lst.length;

}

#endregion

#region 分解

public bool decompose(LinkList lst1, LinkList lst2, decimal n)

{

if (!lst1.IsEmpty() || !lst2.IsEmpty()) //存放分解结点的链表应为空链表

return false;

Node t = this.head.next;

Node p = lst1.head;

Node q = lst2.head;

while (t != null)

{

Node r = new Node(t.data);

if (Convert.ToDecimal(r.data) < n)

{

r.next = p.next;

p.next = r;

p = p.next;

lst1.length++;

}

else

{

r.next = q.next;

q.next = r;

q = q.next;

lst2.length++;

}

t = t.next;

}

return true;

}

#endregion

#endregion

}