## 一、单链表

目录

1.单链表反转

2.找出单链表的倒数第4个元素

3.找出单链表的中间元素

4.删除无头单链表的一个节点

5.两个不交叉的有序链表的合并

6.有个二级单链表，其中每个元素都含有一个指向一个单链表的指针。写程序把这个二级链表称一级单链表。

7.单链表交换任意两个元素（不包括表头）

8.判断单链表是否有环？如何找到环的“起始”点？如何知道环的长度？

9.判断两个单链表是否相交

10.两个单链表相交，计算相交点

11.用链表模拟大整数加法运算

12.单链表排序

13.删除单链表中重复的元素

首先写一个单链表的C#实现，这是我们的基石：

public class Link

{

public Link Next;

public string Data;

public Link(Link next, string data)

{

this.Next = next;

this.Data = data;

}

}

其中，我们需要人为地在单链表前面加一个空节点，称其为head。例如，一个单链表是1->2->5，如图所示：

对一个单链表的遍历如下所示：

static void Main(string[] args)

{

Link head = GenerateLink();

Link curr = head;

while (curr != null)

{

Console.WriteLine(curr.Data);

curr = curr.Next;

}

}

### 1.单链表反转

这道题目有两种算法，既然是要反转，那么肯定是要破坏原有的数据结构的：

算法1：我们需要额外的两个变量来存储当前节点curr的下一个节点next、再下一个节点nextnext：

public static Link ReverseLink1(Link head)

{

Link curr = head.Next;

Link next = null;

Link nextnext = null;

//if no elements or only one element exists

if (curr == null || curr.Next == null)

{

return head;

}

//if more than one element

while (curr.Next != null)

{

next = curr.Next; //1

nextnext = next.Next; //2

next.Next = head.Next; //3

head.Next = next; //4

curr.Next = nextnext; //5

}

return head;

}

算法的核心是while循环中的5句话

我们发现，curr始终指向第1个元素。

此外，出于编程的严谨性，还要考虑2种极特殊的情况：没有元素的单链表，以及只有一个元素的单链表，都是不需要反转的。

算法2：自然是递归

如果题目简化为逆序输出这个单链表，那么递归是很简单的，在递归函数之后输出当前元素，这样能确保输出第N个元素语句永远在第N+1个递归函数之后执行，也就是说第N个元素永远在第N+1个元素之后输出，最终我们先输出最后一个元素，然后是倒数第2个、倒数第3个，直到输出第1个：

public static void ReverseLink2(Link head)

{

if (head.Next != null)

{

ReverseLink2(head.Next);

Console.WriteLine(head.Next.Data);

}

}

但是，现实应用中往往不是要求我们逆序输出（不损坏原有的单链表），而是把这个单链表逆序（破坏型）。这就要求我们在递归的时候，还要处理递归后的逻辑。

首先，要把判断单链表有0或1个元素这部分逻辑独立出来，而不需要在递归中每次都比较一次：

public static Link ReverseLink3(Link head)

{

//if no elements or only one element exists

if (head.Next == null || head.Next.Next == null)

return head;

head.Next = ReverseLink(head.Next);

return head;

}

我们观测到：

head.Next = ReverseLink(head.Next);

这句话的意思是为ReverseLink方法生成的逆序链表添加一个空表头。

接下来就是递归的核心算法ReverseLink了：

static Link ReverseLink(Link head)

{

if (head.Next == null)

return head;

Link rHead = ReverseLink(head.Next);

head.Next.Next = head;

head.Next = null;

return rHead;

}

算法的关键就在于递归后的两条语句：

head.Next.Next = head; //1

head.Next = null; //2

啥意思呢？画个图表示就是：

这样，就得到了一个逆序的单链表，我们只用到了1个额外的变量rHead。

### 2.找出单链表的倒数第4个元素

这道题目有两种算法，但无论哪种算法，都要考虑单链表少于4个元素的情况：

第1种算法，建立两个指针，第一个先走4步，然后第2个指针也开始走，两个指针步伐（前进速度）一致。

static Link GetLast4thOne(Link head)

{

Link first = head;

Link second = head;

for (int i = 0; i < 4; i++)

{

if (first.Next == null)

throw new Exception("Less than 4 elements");

first = first.Next;

}

while (first != null)

{

first = first.Next;

second = second.Next;

}

return second;

}

第2种算法，做一个数组arr[4]，让我们遍历单链表，把第0个、第4个、第8个……第4N个扔到arr[0]，把第1个、第5个、第9个……第4N+1个扔到arr[1]，把第2个、第6个、第10个……第4N+2个扔到arr[2]，把第3个、第7个、第11个……第4N+3个扔到arr[3]，这样随着单链表的遍历结束，arr中存储的就是单链表的最后4个元素，找到最后一个元素对应的arr[i]，让k=(i+1)%4，则arr[k]就是倒数第4个元素。

static Link GetLast4thOneByArray(Link head)

{

Link curr = head;

int i = 0;

Link[] arr = new Link[4];

while (curr.Next != null)

{

arr[i] = curr.Next;

curr = curr.Next;

i = (i + 1) % 4;

}

if (arr[i] == null)

throw new Exception("Less than 4 elements");

return arr[i];

}

本题目源代码下载：

推而广之，对倒数第K个元素，都能用以上2种算法找出来。

### 3.找出单链表的中间元素

算法思想：类似于上题，还是使用两个指针first和second，只是first每次走一步，second每次走两步：

static Link GetMiddleOne(Link head)

{

Link first = head;

Link second = head;

while (first != null && first.Next != null)

{

first = first.Next.Next;

second = second.Next;

}

return second;

}

但是，这道题目有个地方需要注意，就是对于链表元素个数为奇数，以上算法成立。如果链表元素个数为偶数，那么在返回second的同时，还要返回second.Next也就是下一个元素，它俩都算是单链表的中间元素。

下面是加强版的算法，无论奇数偶数，一概通杀：

static void Main(string[] args)

{

Link head = GenerateLink();

bool isOdd = true;

Link middle = GetMiddleOne(head, ref isOdd);

if (isOdd)

{

Console.WriteLine(middle.Data);

}

else

{

Console.WriteLine(middle.Data);

Console.WriteLine(middle.Next.Data);

}

Console.Read();

}

static Link GetMiddleOne(Link head, ref bool isOdd)

{

Link first = head;

Link second = head;

while (first != null && first.Next != null)

{

first = first.Next.Next;

second = second.Next;

}

if (first != null)

isOdd = false;

return second;

}

### 4.一个单链表，很长，遍历一遍很慢，我们仅知道一个指向某节点的指针curr，而我们又想删除这个节点。

这道题目是典型的“狸猫换太子”，如下图所示：

如果不考虑任何特殊情况，代码就2行：

curr.Data = curr.Next.Data;

curr.Next = curr.Next.Next;

上述代码由一个地方需要注意，就是如果要删除的是最后一个元素呢？那就只能从头遍历一次找到倒数第二个节点了。

此外，这道题目的一个变身就是将一个环状单链表拆开（即删除其中一个元素），此时，只要使用上面那两行代码就可以了，不需要考虑表尾。

相关问题：只给定单链表中某个结点p(非空结点)，在p前面插入一个结点q。

话说，交换单链表任意两个节点，也可以用交换值的方法。但这样就没意思了，所以，才会有第7题霸王硬上工的做法。

### 5.两个不交叉的有序链表的合并

有两个有序链表，各自内部是有序的，但是两个链表之间是无序的。

算法思路：当然是循环逐项比较两个链表了，如果一个到了头，就不比较了，直接加上去。

注意，对于2个元素的Data相等（仅仅是Data相等哦，而不是相同的引用），我们可以把它视作前面的Data大于后面的Data，从而节省了算法逻辑。

static Link MergeTwoLink(Link head1, Link head2)

{

Link head = new Link(null, Int16.MinValue);

Link pre = head;

Link curr = head.Next;

Link curr1 = head1;

Link curr2 = head2;

//compare until one link run to the end

while (curr1.Next != null && curr2.Next != null)

{

if (curr1.Next.Data < curr2.Next.Data)

{

curr = new Link(null, curr1.Next.Data);

curr1 = curr1.Next;

}

else

{

curr = new Link(null, curr2.Next.Data);

curr2 = curr2.Next;

}

pre.Next = curr;

pre = pre.Next;

}

//if head1 run to the end

while (curr1.Next != null)

{

curr = new Link(null, curr1.Next.Data);

curr1 = curr1.Next;

pre.Next = curr;

pre = pre.Next;

}

//if head2 run to the end

while (curr2.Next != null)

{

curr = new Link(null, curr2.Next.Data);

curr2 = curr2.Next;

pre.Next = curr;

pre = pre.Next;

}

return head;

}

如果这两个有序链表交叉组成了Y型呢，比如说：

这时我们需要先找出这个交叉点（图中是11），这个算法参见第9题，我们这里直接使用第10道题目中的方法GetIntersect。

然后局部修改上面的算法，只要其中一个链表到达了交叉点，就直接把另一个链表的剩余元素都加上去。如下所示：

static Link MergeTwoLink2(Link head1, Link head2)

{

Link head = new Link(null, Int16.MinValue);

Link pre = head;

Link curr = head.Next;

Link intersect = GetIntersect(head1, head2);

Link curr1 = head1;

Link curr2 = head2;

//compare until one link run to the intersect

while (curr1.Next != intersect && curr2.Next != intersect)

{

if (curr1.Next.Data < curr2.Next.Data)

{

curr = new Link(null, curr1.Next.Data);

curr1 = curr1.Next;

}

else

{

curr = new Link(null, curr2.Next.Data);

curr2 = curr2.Next;

}

pre.Next = curr;

pre = pre.Next;

}

//if head1 run to the intersect

if (curr1.Next == intersect)

{

while (curr2.Next != null)

{

curr = new Link(null, curr2.Next.Data);

curr2 = curr2.Next;

pre.Next = curr;

pre = pre.Next;

}

}

//if head2 run to the intersect

else if (curr2.Next == intersect)

{

while (curr1.Next != null)

{

curr = new Link(null, curr1.Next.Data);

curr1 = curr1.Next;

pre.Next = curr;

pre = pre.Next;

}

}

return head;

}

### 6.有个二级单链表，其中每个元素都含有一个指向一个单链表的指针。写程序把这个二级链表展开称一级单链表。

这个简单，就是说，这个二级单链表只包括一些head：

public class Link

{

public Link Next;

public int Data;

public Link(Link next, int data)

{

this.Next = next;

this.Data = data;

}

}

public class CascadeLink

{

public Link Next;

public CascadeLink NextHead;

public CascadeLink(CascadeLink nextHead, Link next)

{

this.Next = next;

this.NextHead = nextHead;

}

}

下面做一个二级单链表，GenerateLink1和GenerateLink2方法在前面都已经介绍过了：

public static CascadeLink GenerateCascadeLink()

{

Link head1 = GenerateLink1();

Link head2 = GenerateLink2();

Link head3 = GenerateLink1();

CascadeLink element3 = new CascadeLink(null, head3);

CascadeLink element2 = new CascadeLink(element3, head2);

CascadeLink element1 = new CascadeLink(element2, head1);

CascadeLink head = new CascadeLink(element1, null);

return head;

}

就是说，这些单链表的表头head1、head2、head3、head4……，它们组成了一个二级单链表head：null –> head1 –> head2 –> head3 –> head4

–>

我们的算法思想是： 进行两次遍历，在外层用curr1遍历二级单链表head，在内层用curr2遍历每个单链表：

public static Link GenerateNewLink(CascadeLink head)

{

CascadeLink curr1 = head.NextHead;

Link newHead = curr1.Next;

Link curr2 = newHead;

while (curr1 != null)

{

curr2.Next = curr1.Next.Next;

while (curr2.Next != null)

{

curr2 = curr2.Next;

}

curr1 = curr1.NextHead;

}

return newHead;

}

其中，curr2.Next = curr1.Next.Next; 这句话是关键，它负责把上一个单链表的表尾和下一个单链表的非空表头连接起来。

### 7.单链表交换任意两个元素（不包括表头）

先一次遍历找到这两个元素curr1和curr2，同时存储这两个元素的前驱元素pre1和pre2。

然后大换血

public static Link SwitchPoints(Link head, Link p, Link q)

{

if (p == head || q == head)

throw new Exception("No exchange with head");

if (p == q)

return head;

//find p and q in the link

Link curr = head;

Link curr1 = p;

Link curr2 = q;

Link pre1 = null;

Link pre2 = null;

int count = 0;

while (curr != null)

{

if (curr.Next == p)

{

pre1 = curr;

count++;

if (count == 2)

break;

}

else if (curr.Next == q)

{

pre2 = curr;

count++;

if (count == 2)

break;

}

curr = curr.Next;

}

curr = curr1.Next;

pre1.Next = curr2;

curr1.Next = curr2.Next;

pre2.Next = curr1;

curr2.Next = curr;

return head;

}

注意特例，如果相同元素，就没有必要交换；如果有一个是表头，就不交换。

### 8.判断单链表是否有环？如何找到环的“起始”点？如何知道环的长度？

算法思想：

先分析是否有环。为此我们建立两个指针，从header一起向前跑，一个步长为1，一个步长为2，用while（直到步长2的跑到结尾）检查两个指针是否相等，直到找到为止。

static bool JudgeCircleExists(Link head)

{

Link first = head; //1 step each time

Link second = head; //2 steps each time

while (second.Next != null && second.Next.Next != null)

{

second = second.Next.Next;

first = first.Next;

if (second == first)

return true;

}

return false;

}

那又如何知道环的长度呢？

根据上面的算法，在返回true的地方，也就是2个指针相遇处，这个位置的节点P肯定位于环上。我们从这个节点开始先前走，转了一圈肯定能回来：

static int GetCircleLength(Link point)

{

int length = 1;

Link curr = point;

while (curr.Next != point)

{

length++;

curr = curr.Next;

}

return length;

}

继续我们的讨论，如何找到环的“起始”点呢？

延续上面的思路，我们仍然在返回true的地方P，计算一下从有环单链表的表头head到P点的距离

static int GetLengthFromHeadToPoint(Link head, Link point)

{

int length = 1;

Link curr = head;

while (curr != point)

{

length++;

curr = curr.Next;

}

return length;

}

如果我们把环从P点“切开”（当然并不是真的切，那就破坏原来的数据结构了），那么问题就转化为计算两个相交“单链表”的交点（第10题）：

一个单链表是从P点出发，到达P（一个回圈），距离M；另一个单链表从有环单链表的表头head出发，到达P，距离N。

我们可以参考第10题的GetIntersect方法并稍作修改。

private static Link FindIntersect(Link head)

{

Link p = null;

//get the point in the circle

bool result = JudgeCircleExists(head, ref p);

if (!result) return null;

Link curr1 = head.Next;

Link curr2 = p.Next;

//length from head to p

int M = 1;

while (curr1 != p)

{

M++;

curr1 = curr1.Next;

}

//circle length

int N = 1;

while (curr2 != p)

{

N++;

curr2 = curr2.Next;

}

//recover curr1 & curr2

curr1 = head.Next;

curr2 = p.Next;

//make 2 links have the same distance to the intersect

if (M > N)

{

for (int i = 0; i < M - N; i++)

curr1 = curr1.Next;

}

else if (M < N)

{

for (int i = 0; i < N - M; i++)

curr2 = curr2.Next;

}

//goto the intersect

while (curr1 != p)

{

if (curr1 == curr2)

{

return curr1;

}

curr1 = curr1.Next;

curr2 = curr2.Next;

}

return null;

}

### 9.判断两个单链表是否相交

这道题有多种算法。

算法1：把第一个链表逐项存在hashtable中，遍历第2个链表的每一项，如果能在第一个链表中找到，则必然相交。

static bool JudgeIntersectLink1(Link head1, Link head2)

{

Hashtable ht = new Hashtable();

Link curr1 = head1;

Link curr2 = head2;

//store all the elements of link1

while (curr1.Next != null)

{

ht[curr1.Next] = string.Empty;

curr1 = curr1.Next;

}

//check all the elements in link2 if exists in Hashtable or not

while (curr2.Next != null)

{

//if exists

if (ht[curr2.Next] != null)

{

return true;

}

curr2 = curr2.Next;

}

return false;

}

算法2：把一个链表A接在另一个链表B的末尾，如果有环，则必然相交。如何判断有环呢？从A开始遍历，如果能回到A的表头，则肯定有环。

注意，在返回结果之前，要把刚才连接上的两个链表断开，恢复原状。

static bool JudgeIntersectLink2(Link head1, Link head2)

{

bool exists = false;

Link curr1 = head1;

Link curr2 = head2;

//goto the end of the link1

while (curr1.Next != null)

{

curr1 = curr1.Next;

}

//join these two links

curr1.Next = head2;

//iterate link2

while (curr2.Next != null)

{

if (curr2.Next == head2)

{

exists = true;

break;

}

curr2 = curr2.Next;

}

//recover original status, whether exists or not

curr1.Next = null;

return exists;

}

算法3：如果两个链表的末尾元素相同，则必相交。

static bool JudgeIntersectLink3(Link head1, Link head2)

{

Link curr1 = head1;

Link curr2 = head2;

//goto the end of the link1

while (curr1.Next != null)

{

curr1 = curr1.Next;

}

//goto the end of the link2

while (curr2.Next != null)

{

curr2 = curr2.Next;

}

if (curr1 != curr2)

return false;

else

return true;

}

### 10.两个单链表相交，计算相交点

分别遍历两个单链表，计算出它们的长度M和N，假设M比N大，则长度M的链表先前进M-N，然后两个链表同时以步长1前进，前进的同时比较当前的元素，如果相同，则必是交点。

public static Link GetIntersect(Link head1, Link head2)

{

Link curr1 = head1;

Link curr2 = head2;

int M = 0, N = 0;

//goto the end of the link1

while (curr1.Next != null)

{

curr1 = curr1.Next;

M++;

}

//goto the end of the link2

while (curr2.Next != null)

{

curr2 = curr2.Next;

N++;

}

//return to the begining of the link

curr1 = head1;

curr2 = head2;

if (M > N)

{

for (int i = 0; i < M - N; i++)

curr1 = curr1.Next;

}

else if (M < N)

{

for (int i = 0; i < N - M; i++)

curr2 = curr2.Next;

}

while (curr1.Next != null)

{

if (curr1 == curr2)

{

return curr1;

}

curr1 = curr1.Next;

curr2 = curr2.Next;

}

return null;

}

### 11.用链表模拟大整数加法运算

例如：9>9>9>NULL + 1>NULL =>

1>0>0>0>NULL

肯定是使用递归啦，不然没办法解决进位+1问题，因为这时候要让前面的节点加1，而我们的单链表是永远指向前的。

此外对于999+1=1000，新得到的值的位数（4位）比原来的两个值（1个1位，1个3位）都多，所以我们将表头的值设置为0，如果多出一位来，就暂时存放到表头。递归结束后，如果表头为1，就在新的链表外再加一个新的表头。

//head1 length > head2, so M > N

public static int Add(Link head1, Link head2, ref Link newHead, int M, int N)

{

// goto the end

if (head1 == null)

return 0;

int temp = 0;

int result = 0;

newHead = new Link(null, 0);

if (M > N)

{

result = Add(head1.Next, head2, ref newHead.Next, M - 1, N);

temp = head1.Data + result;

newHead.Data = temp % 10;

return temp >= 10

1 : 0;

}

else // M == N

{

result = Add(head1.Next, head2.Next, ref newHead.Next, M - 1, N - 1);

temp = head1.Data + head2.Data + +result;

newHead.Data = temp % 10;

return temp >= 10

1 : 0;

}

}

这里假设head1比head2长，而且M、N分别是head1和head2的长度。

### 12.单链表排序

无外乎是冒泡、选择、插入等排序方法。关键是交换算法，需要额外考虑。第7题我编写了一个交换算法，在本题的排序过程中，我们可以在外层和内层循环里面，捕捉到pre1和pre2，然后进行交换，而无需每次交换又要遍历一次单链表。

在实践中，我发现冒泡排序和选择排序都要求内层循环从链表的末尾向前走，这明显是不合时宜的。

所以我最终选择了插入排序算法，如下所示：

先给出基于数组的算法：

代码

static int[]

InsertSort(int[] arr)

{

for(int i=1; i<arr.Length;i++)

{

for(int j =i; (j>0)&&arr[j]<arr[j-1];j--)

{

arr[j]=arr[j]^arr[j-1];

arr[j-1]=arr[j]^arr[j-1];

arr[j]=arr[j]^arr[j-1];

}

}

return arr;

}

仿照上面的思想，我们来编写基于Link的算法：

public static Link SortLink(Link head)

{

Link pre1 = head;

Link pre2 = head.Next;

Link min = null;

for (Link curr1 = head.Next; curr1 != null; curr1 = min.Next)

{

if (curr1.Next == null)

break;

min = curr1;

for (Link curr2 = curr1.Next; curr2 != null; curr2 = curr2.Next)

{

//swap curr1 and curr2

if (curr2.Data < curr1.Data)

{

min = curr2;

curr2 = curr1;

curr1 = min;

pre1.Next = curr1;

curr2.Next = curr1.Next;

curr1.Next = pre2;

//if exchange element n-1 and n, no need to add reference from pre2 to curr2, because they are the same one

if (pre2 != curr2)

pre2.Next = curr2;

}

pre2 = curr2;

}

pre1 = min;

pre2 = min.Next;

}

return head;

}

值得注意的是，很多人的算法不能交换相邻两个元素，这是因为pre2和curr2是相等的，如果此时还执行pre2.Next = curr2; 会造成一个自己引用自己的环。

交换指针很是麻烦，而且效率也不高，需要经常排序的东西最好不要用链表来实现，还是数组好一些。

### 13.删除单链表中重复的元素

用Hashtable辅助，遍历一遍单链表就能搞定。

实践中发现，curr从表头开始，每次判断下一个元素curr.Netx是否重复，如果重复直接使用curr.Next = curr.Next.Next; 就可以删除重复元素——这是最好的算法。唯一的例外就是表尾，所以到达表尾，就break跳出while循环。

public static Link DeleteDuplexElements(Link head)

{

Hashtable ht = new Hashtable();

Link curr = head;

while (curr != null)

{

if (curr.Next == null)

{

break;

}

if (ht[curr.Next.Data] != null)

{

curr.Next = curr.Next.Next;

}

else

{

ht[curr.Next.Data] = "";

}

curr = curr.Next;

}

return head;

}

结语：

单链表只有一个向前指针Next，所以要使用1-2个额外变量来存储当前元素的前一个或后一个指针。

尽量用while循环而不要用for循环，来进行遍历。

哇塞，我就是不用指针，照样能“修改地址”，达到和C++同样的效果，虽然很烦~

遍历的时候，不要在while循环中head=head.Next;这样会改变原先的数据结构。我们要这么写：Link curr=head;然后curr=curr.Next;

有时我们需要临时把环切开，有时我们需要临时把单链表首尾相连成一个环。

究竟是玩curr还是curr.Next，根据不同题目而各有用武之地，没有定论，不必强求。

## 二、栈和队列

目录：

1.设计含min函数的栈，要求min、push和pop的时间复杂度都是o(1)。

2.设计含min函数的栈的另解

3.用两个栈实现队列

4.用两个队列实现栈

5.栈的push、pop序列是否一致

6.递归反转一个栈，要求不得重新申请一个同样的栈，空间复杂度o(1)

7.给栈排个序

8..如何用一个数组实现两个栈

9..如何用一个数组实现三个栈

### 1.设计含min函数的栈，要求min、push和pop的时间复杂度都是o(1)。

算法思想：需要设计一个辅助栈，用来存储当前栈中元素的最小值。网上有人说存储当前栈中元素的最小值的所在位置，虽然能节省空间，这其实是不对的，因为我在调用Min函数的时候，只能得到位置，还要对存储元素的栈不断的pop，才能得到最小值——时间复杂度o(1)。

所以，还是在辅助栈中存储元素吧。

此外，还要额外注意Push操作，第一个元素不用比较，自动成为最小值入栈。其它元素每次都要和栈顶元素比较，小的那个放到栈顶。

public class NewStack

{

private Stack dataStack;

private Stack mindataStack;

public NewStack()

{

dataStack = new Stack();

mindataStack = new Stack();

}

public void Push(int element)

{

dataStack.Push(element);

if (mindataStack.Count == 0)

mindataStack.Push(element);

else if (element <= (int)mindataStack.Peek())

mindataStack.Push(element);

else //(element > mindataStack.Peek)

mindataStack.Push(mindataStack.Peek());

}

public int Pop()

{

if (dataStack.Count == 0)

throw new Exception("The stack is empty");

mindataStack.Pop();

return (int)dataStack.Pop();

}

public int Min()

{

if (dataStack.Count == 0)

throw new Exception("The stack is empty");

return (int)mindataStack.Peek();

}

}

### 2.设计含min函数的栈的另解

话说，和青菜脸呆久了，就沾染了上海小市民意识，再加上原本我就很抠门儿，于是对于上一题目，我把一个栈当成两个用，就是说，每次push，先入站当前元素，然后入栈当前栈中最小元素；pop则每次弹出2个元素。

算法代码如下所示（这里最小元素位于当前元素之上，为了下次比较方便）：

public class NewStack

{

private Stack stack;

public NewStack()

{

stack = new Stack();

}

public void Push(int element)

{

if (stack.Count == 0)

{

stack.Push(element);

stack.Push(element);

}

else if (element <= (int)stack.Peek())

{

stack.Push(element);

stack.Push(element);

}

else //(element > stack.Peek)

{

object min = stack.Peek();

stack.Push(element);

stack.Push(min);

}

}

public int Pop()

{

if (stack.Count == 0)

throw new Exception("The stack is empty");

stack.Pop();

return (int)stack.Pop();

}

public int Min()

{

if (stack.Count == 0)

throw new Exception("The stack is empty");

return (int)stack.Peek();

}

}

之所以说我这个算法比较叩门，是因为我只使用了一个栈，空间复杂度o(N)，节省了一半的空间（算法1的空间复杂度o(2N)）。

### 3.用两个栈实现队列

实现队列，就要实现它的3个方法：Enqueue（入队）、Dequeue（出队）和Peek（队头）。

1）stack1存的是每次进来的元素，所以Enqueue就是把进来的元素push到stack1中。

2）而对于Dequeue，一开始stack2是空的，所以我们把stack1中的元素全都pop到stack2中，这样stack2的栈顶就是队头。只要stack2不为空，那么每次出队，就相当于stack2的pop。

3）接下来，每入队一个元素，仍然push到stack1中。每出队一个元素，如果stack2不为空，就从stack2中pop一个元素；如果stack2为空，就重复上面的操作——把stack1中的元素全都pop到stack2中。

4）Peek操作，类似于Dequeue，只是不需要出队，所以我们调用stack2的Peek操作。当然，如果stack2为空，就把stack1中的元素全都pop到stack2中。

5）注意边界的处理，如果stack2和stack1都为空，才等于队列为空，此时不能进行Peek和Dequeue操作。

按照上述分析，算法实现如下：

public class NewQueue

{

private Stack stack1;

private Stack stack2;

public NewQueue()

{

stack1 = new Stack();

stack2 = new Stack();

}

public void Enqueue(int element)

{

stack1.Push(element);

}

public int Dequeue()

{

if (stack2.Count == 0)

{

if (stack1.Count == 0)

throw new Exception("The queue is empty");

else

while (stack1.Count > 0)

stack2.Push(stack1.Pop());

}

return (int)stack2.Pop();

}

public int Peek()

{

if (stack2.Count == 0)

{

if (stack1.Count == 0)

throw new Exception("The queue is empty");

else

while (stack1.Count > 0)

stack2.Push(stack1.Pop());

}

return (int)stack2.Peek();

}

}

### 4.用两个队列实现栈

这个嘛，就要queue1和queue2轮流存储数据了。这个“轮流”发生在Pop和Peek的时候，假设此时我们把所有数据存在queue1中（此时queue2为空），我们把queue1的n-1个元素放到queue2中，queue中最后一个元素就是我们想要pop的元素，此时queue2存有n-1个元素（queue1为空）。

至于Peek，则是每次转移n个数据，再转移最后一个元素的时候，将其计下并返回。

那么Push的操作，则需要判断当前queue1和queue2哪个为空，将新元素放到不为空的队列中。

public class NewStack

{

private Queue queue1;

private Queue queue2;

public NewStack()

{

queue1 = new Queue();

queue2 = new Queue();

}

public void Push(int element)

{

if (queue1.Count == 0)

queue2.Enqueue(element);

else

queue1.Enqueue(element);

}

public int Pop()

{

if (queue1.Count == 0 && queue2.Count == 0)

throw new Exception("The stack is empty");

if (queue1.Count > 0)

{

while (queue1.Count > 1)

{

queue2.Enqueue(queue1.Dequeue());

}

//还剩一个

return (int)queue1.Dequeue();

}

else //queue2.Count > 0

{

while (queue2.Count > 1)

{

queue1.Enqueue(queue2.Dequeue());

}

//还剩一个

return (int)queue2.Dequeue();

}

}

public int Peek()

{

if (queue1.Count == 0 && queue2.Count == 0)

throw new Exception("The stack is empty");

int result = 0;

if (queue1.Count > 0)

{

while (queue1.Count > 1)

{

queue2.Enqueue(queue1.Dequeue());

}

//还剩一个

result = (int)queue1.Dequeue();

queue2.Enqueue(result);

}

else //queue2.Count > 0

{

while (queue2.Count > 1)

{

queue1.Enqueue(queue2.Dequeue());

}

//还剩一个

result = (int)queue2.Dequeue();

queue1.Enqueue(result);

}

return result;

}

}

### 5.栈的push、pop序列是否一致

输入两个整数序列。其中一个序列表示栈的push顺序，判断另一个序列有没有可能是对应的pop顺序。为了简单起见，我们假设push序列的任意两个整数都是不相等的。

比如输入的push序列是1、2、3、4、5，那么4、5、3、2、1就有可能是一个pop系列。因为可以有如下的push和pop序列：push 1，push 2，push 3，push 4，pop，push 5，pop，pop，pop，pop，这样得到的pop序列就是4、5、3、2、1。但序列4、3、5、1、2就不可能是push序列1、2、3、4、5的pop序列。

网上的若干算法都太复杂了，现提出包氏算法如下：

先for循环把arr1中的元素入栈，并在每次遍历时，检索arr2中可以pop的元素。如果循环结束，而stack中还有元素，就说明arr2序列不是pop序列。

static bool

JudgeSequenceIsPossible(int[] arr1,int[] arr2)

{

Stack stack = new Stack();

for (int i = 0, j = 0; i < arr1.Length; i++)

{

stack.Push(arr1[i]);

while(stack.Count > 0 && (int)stack.Peek() == arr2[j])

{

stack.Pop();

j++;

}

}

return stack.Count == 0;

}

### 6.递归反转一个栈，要求不得重新申请一个同样的栈，空间复杂度o(1)

算法思想：汉诺塔的思想，非常复杂，玩过九连环的人都想得通的

static void ReverseStack(ref Stack stack)

{

if (stack.Count == 0)

return;

object top = stack.Pop();

ReverseStack(ref stack);

if (stack.Count == 0)

{

stack.Push(top);

return;

}

object top2 = stack.Pop();

ReverseStack(ref stack);

stack.Push(top);

ReverseStack(ref stack);

stack.Push(top2);

}

### 7.给栈排个序

本题目是上一题目的延伸

static void Sort(ref Stack stack)

{

if (stack.Count == 0)

return;

object top = stack.Pop();

Sort(ref stack);

if (stack.Count == 0)

{

stack.Push(top);

return;

}

object top2 = stack.Pop();

if ((int)top > (int)top2)

{

stack.Push(top);

Sort(ref stack);

stack.Push(top2);

}

else

{

stack.Push(top2);

Sort(ref stack);

stack.Push(top);

}

}

### 8..如何用一个数组实现两个栈

继续我所提倡的抠门儿思想，也不枉我和青菜脸相交一场。

网上流传着两种方法：

方法1

采用交叉索引的方法

一号栈所占数组索引为0, 2, 4, 6, 8......(K\*2)

二号栈所占数组索引为1，3，5，7，9 ......(K\*2 + 1)

算法实现如下：

public class NewStack

{

object[] arr;

int top1;

int top2;

public NewStack(int capticy)

{

arr = new object[capticy];

top1 = -1;

top2 = -2;

}

public void Push(int type, object element)

{

if (type == 1)

{

if (top1 + 2 >= arr.Length)

throw new Exception("The stack is full");

else

{

top1 += 2;

arr[top1] = element;

}

}

else //type==2

{

if (top2 + 2 >= arr.Length)

throw new Exception("The stack is full");

else

{

top2 += 2;

arr[top2] = element;

}

}

}

public object Pop(int type)

{

object obj = null;

if (type == 1)

{

if (top1 == -1)

throw new Exception("The stack is empty");

else

{

obj = arr[top1];

arr[top1] = null;

top1 -= 2;

}

}

else //type == 2

{

if (top2 == -2)

throw new Exception("The stack is empty");

else

{

obj = arr[top2];

arr[top2] = null;

top2 -= 2;

}

}

return obj;

}

public object Peek(int type)

{

if (type == 1)

{

if (top1 == -1)

throw new Exception("The stack is empty");

return arr[top1];

}

else //type == 2

{

if (top2 == -2)

throw new Exception("The stack is empty");

return arr[top2];

}

}

}

方法2：

第一个栈A：从最左向右增长

第二个栈B：从最右向左增长

代码实现如下：

public class NewStack

{

object[] arr;

int top1;

int top2;

public NewStack(int capticy)

{

arr = new object[capticy];

top1 = 0;

top2 = capticy;

}

public void Push(int type, object element)

{

if (top1 == top2)

throw new Exception("The stack is full");

if (type == 1)

{

arr[top1] = element;

top1++;

}

else //type==2

{

top2--;

arr[top2] = element;

}

}

public object Pop(int type)

{

object obj = null;

if (type == 1)

{

if (top1 == 0)

throw new Exception("The stack is empty");

else

{

top1--;

obj = arr[top1];

arr[top1] = null;

}

}

else //type == 2

{

if (top2 == arr.Length)

throw new Exception("The stack is empty");

else

{

obj = arr[top2];

arr[top2] = null;

top2++;

}

}

return obj;

}

public object Peek(int type)

{

if (type == 1)

{

if (top1 == 0)

throw new Exception("The stack is empty");

return arr[top1 - 1];

}

else //type == 2

{

if (top2 == arr.Length)

throw new Exception("The stack is empty");

return arr[top2];

}

}

}

综合比较上述两种算法，我们发现，算法1实现的两个栈，每个都只有n/2个空间大小；而算法2实现的两个栈，如果其中一个很小，另一个则可以很大，它们的和为常数n。

### 9..如何用一个数组实现三个栈

最后，让我们把抠门儿进行到底，相信看完本文，你已经从物质和精神上都升级为一个抠门儿主义者。

如果还使用交叉索引的办法，每个栈都只有N/3个空间。

让我们只好使用上个题目的第2个方法，不过这只能容纳2个栈，我们还需要一个位置存放第3个栈，不如考虑数组中间的位置——第3个栈的增长规律可以如下：

第1个入栈C的元素进mid处

第2个入栈C的元素进mid+1处

第3个入栈C的元素进mid-1处

第4个入栈C的元素进mid+2处

这个方法的好处是， 每个栈都有接近N/3个空间。

public class NewStack

{

object[] arr;

int top1;

int top2;

int top3\_left;

int top3\_right;

bool isLeft;

public NewStack(int capticy)

{

arr = new object[capticy];

top1 = 0;

top2 = capticy;

isLeft = true;

top3\_left = capticy / 2;

top3\_right = top3\_left + 1;

}

public void Push(int type, object element)

{

if (type == 1)

{

if (top1 == top3\_left + 1)

throw new Exception("The stack is full");

arr[top1] = element;

top1++;

}

else if (type == 2)

{

if (top2 == top3\_right)

throw new Exception("The stack is full");

top2--;

arr[top2] = element;

}

else //type==3

{

if (isLeft)

{

if (top1 == top3\_left + 1)

throw new Exception("The stack is full");

arr[top3\_left] = element;

top3\_left--;

}

else

{

if (top2 == top3\_right)

throw new Exception("The stack is full");

arr[top3\_right] = element;

top3\_right++;

}

isLeft = !isLeft;

}

}

public object Pop(int type)

{

object obj = null;

if (type == 1)

{

if (top1 == 0)

throw new Exception("The stack is empty");

else

{

top1--;

obj = arr[top1];

arr[top1] = null;

}

}

else if (type == 2)

{

if (top2 == arr.Length)

throw new Exception("The stack is empty");

else

{

obj = arr[top2];

arr[top2] = null;

top2++;

}

}

else //type==3

{

if (top3\_right == top3\_left + 1)

throw new Exception("The stack is empty");

if (isLeft)

{

top3\_left++;

obj = arr[top3\_left];

arr[top3\_left] = null;

}

else

{

top3\_right--;

obj = arr[top3\_right];

arr[top3\_right] = null;

}

isLeft = !isLeft;

}

return obj;

}

public object Peek(int type)

{

if (type == 1)

{

if (top1 == 0)

throw new Exception("The stack is empty");

return arr[top1 - 1];

}

else if (type == 2)

{

if (top2 == arr.Length)

throw new Exception("The stack is empty");

return arr[top2];

}

else //type==3

{

if (top3\_right == top3\_left + 1)

throw new Exception("The stack is empty");

if (isLeft)

return arr[top3\_left + 1];

else

return arr[top3\_right - 1];

}

}

}

## 三、二叉树

目录：

1.二叉树三种周游（traversal）方式：

2.怎样从顶部开始逐层打印二叉树结点数据

3.如何判断一棵二叉树是否是平衡二叉树

4.设计一个算法，找出二叉树上任意两个节点的最近共同父结点，复杂度如果是O(n2)则不得分。

5.如何不用递归实现二叉树的前序/后序/中序遍历？

6.在二叉树中找出和为某一值的所有路径

7.怎样编写一个程序，把一个有序整数数组放到二叉树中？

8.判断整数序列是不是二叉搜索树的后序遍历结果

9.求二叉树的镜像

10.一棵排序二叉树（即二叉搜索树BST），令 f=(最大值+最小值)/2，设计一个算法，找出距离f值最近、大于f值的结点。复杂度如果是O(n2)则不得分。

11.把二叉搜索树转变成排序的双向链表

首先写一个二叉树的C#实现，这是我们的基石：

public class BinNode

{

public int Element;

public BinNode Left;

public BinNode Right;

public BinNode(int element, BinNode left, BinNode right)

{

this.Element = element;

this.Left = left;

this.Right = right;

}

public bool IsLeaf()

{

return this.Left == null && this.Right == null;

}

}

### 1.二叉树三种周游（traversal）方式：

1）前序周游（preorder）：节点 –> 子节点Left（包括其子树） –> 子节点Right（包括其子树）

static void PreOrder(BinNode root)

{

if (root == null)

return;

//visit current node

Console.WriteLine(root.Element);

PreOrder(root.Left);

PreOrder(root.Right);

}

2）后序周游（postorder）：子节点Left（包括其子树） –> 子节点Right（包括其子树） –> 节点

static void PostOrder(BinNode root)

{

if (root == null)

return;

PostOrder(root.Left);

PostOrder(root.Right);

//visit current node

Console.WriteLine(root.Element);

}

3）中序周游（inorder）：子节点Left（包括其子树） –> 节点 –> 子节点Right（包括其子树）

static void InOrder(BinNode root)

{

if (root == null)

return;

InOrder(root.Left);

//visit current node

Console.WriteLine(root.Element);

InOrder(root.Right);

}

我们发现，三种周游的code实现，仅仅是访问当前节点的这条语句所在位置不同而已。

### 2.怎样从顶部开始逐层打印二叉树结点数据

有2种算法：

算法1：基于Queue来实现，也就是广度优先搜索（BFS）的思想

static void PrintTree1(BinNode root)

{

if (root == null) return;

BinNode tmp = null;

Queue queue = new Queue();

queue.Enqueue(root);

while (queue.Count > 0)

{

tmp = (BinNode)queue.Dequeue();

Console.WriteLine(tmp.Element);

if (tmp.Left != null)

queue.Enqueue(tmp.Left);

if (tmp.Right != null)

queue.Enqueue(tmp.Right);

}

}

话说，BFS和DFS思想本来是用于图的，但我们不能被传统的思维方式所束缚。

算法2：基于单链表实现

如果没有Queue给我们用，我们只好使用单链表，把每个节点存在单链表的Data中，实现如下：

public class Link

{

public Link Next;

public BinNode Data;

public Link(Link next, BinNode data)

{

this.Next = next;

this.Data = data;

}

}

看过了Queue的实现，我们发现永远是先出队1个（队头），然后入队2个（把出队的Left和Right放到队尾）。

对于单链表而言，我们可以先模拟入队——把first的Data所对应的Left和Right，先后插到second的后面，即second.Next和second.Next.Next位置，同时second向前走0、1或2次，再次到达链表末尾，这取决于Left和Right是否为空；然后我们模拟出队——first前进1步。

当first指针走不下去了，那么任务也就结束了。

static void PrintTree2(BinNode root)

{

if (root == null) return;

Link head = new Link(null, root);

Link first = head;

Link second = head;

while (first != null)

{

if (first.Data.Left != null)

{

second.Next = new Link(null, first.Data.Left);

second = second.Next;

}

if (first.Data.Right != null)

{

second.Next = new Link(null, first.Data.Right);

second = second.Next;

}

Console.WriteLine(first.Data.Element);

first = first.Next;

}

}

### 3.如何判断一棵二叉树是否是平衡二叉树

平衡二叉树的定义，如果任意节点的左右子树的深度相差不超过1，那这棵树就是平衡二叉树。

算法思路：先编写一个计算二叉树深度的函数GetDepth，利用递归实现；然后再递归判断每个节点的左右子树的深度是否相差1

static int GetDepth(BinNode root)

{

if (root == null)

return 0;

int leftLength = GetDepth(root.Left);

int rightLength = GetDepth(root.Right);

return (leftLength > rightLength

leftLength : rightLength) + 1;

}

注意这里的+1，对应于root不为空（算作当前1个深度）

static bool IsBalanceTree(BinNode root)

{

if (root == null)

return true;

int leftLength = GetDepth(root.Left);

int rightLength = GetDepth(root.Right);

int distance = leftLength > rightLength

leftLength - rightLength : rightLength - leftLength;

if (distance > 1)

return false;

else

return IsBalanceTree(root.Left) && IsBalanceTree(root.Right);

}

上述程序的逻辑是，只要当前节点root的Left和Right深度差不超过1，就递归判断Left和Right是否也符合条件，直到为Left或Right为null，这意味着它们的深度为0，能走到这一步，前面必然都符合条件，所以整个二叉树都符合条件。

### 4.设计一个算法，找出二叉树上任意两个节点的最近共同父结点，复杂度如果是O(n2)则不得分。

本题网上有很多算法，都不怎么样。这里提出包氏的两个算法：

算法1：做一个容器，我们在遍历二叉树寻找节点的同时，把从根到节点的路径扔进去（两个节点就是两个容器）。由于根节点最后一个被扔进去，但我们接下来又需要第一个就能访问到它——后进先出，所以这个容器是一个栈。时间复杂度O(N)，空间复杂度O(N)。

static bool GetPositionByNode(BinNode root, BinNode node, ref Stack stack)

{

if (root == null)

return false;

if (root == node)

{

stack.Push(root);

return true;

}

if (GetPositionByNode(root.Left, node, ref stack) || GetPositionByNode(root.Right, node, ref stack))

{

stack.Push(root);

return true;

}

return false;

}

然后我们要同时弹出这两个容器的元素，直到它们不相等，那么之前那个相等的元素就是我们要求的父亲节点。

static BinNode FindParentNode(BinNode root, BinNode node1, BinNode node2)

{

Stack stack1 = new Stack();

GetPositionByNode(root, node1, ref stack1);

Stack stack2 = new Stack();

GetPositionByNode(root, node2, ref stack2);

BinNode tempNode = null;

while (stack1.Peek() == stack2.Peek())

{

tempNode = (BinNode)stack1.Pop();

stack2.Pop();

}

return tempNode;

}

算法2：如果要求o(1)的空间复杂度，就是说，只能用一个变量来辅助我们。

我们选择一个64位的整数，然后从1开始，从左到右逐层为二叉树的每个元素赋值，root对应1，root.Left对应2，root.Right对应3，依次类推，而不管实际这个位置上是否有节点，我们发现两个规律：

//// 1

//// 2 3

//// 4 5 6 7

//// 8 9 10

如果要找的是5和9位置上的节点。

我们发现，它们的二进制分别是101和1001，右移1001使之与101位数相同，于是1001变成了100（也就是9的父亲4）。

这时101和100（也就是4和5位于同样的深度），我们从左往右找，101和100具有2位相同，即10，这就是我们要找的4和5的父亲，也就是9和5的最近父亲。

由上面观察，得到算法：

1)将找到的两个节点对应的数字

static bool GetPositionByNode(BinNode root, BinNode node, ref int pos)

{

if (root == null)

return false;

if (root == node)

return true;

int temp = pos;

//这么写很别扭，但是能保证只要找到就不再进行下去

pos = temp \* 2;

if (GetPositionByNode(root.Left, node, ref pos))

{

return true;

}

else

{

//找不到左边找右边

pos = temp \* 2 + 1;

return GetPositionByNode(root.Right, node, ref pos);

}

}

2)它们的二进制表示，从左向右逐一比较，直到一个结束或不再相同，则最大的相同子串，就是我们需要得到的最近父亲所对应的位置K。

static int FindParentPosition(int larger, int smaller)

{

if (larger == smaller) return larger;

int left = GetLen(larger) - GetLen(smaller);

while (left > 0)

{

larger = larger >> 1;

left--;

}

while (larger != smaller)

{

larger = larger >> 1;

smaller = smaller >> 1;

}

return smaller;

}

static int GetLen(int num)

{

int length = 0;

while (num != 0)

{

num = num >> 1;

length++;

}

return length;

}

3)第3次递归遍历，寻找K所对应的节点。

函数GetNodeByPosition的思想是，先算出k在第几层power，观察k的二进制表示，比如说12，即1100，从左向右数第一个位1不算，还剩下100，1表示向右走，0表示向左走，于是从root出发，1->3->6->12。

static BinNode GetNodeByPosition(BinNode root, int num)

{

if (num == 1) return root;

int pow = (int)Math.Floor(Math.Log(num, 2)); //1 return 0, 2-3 return 1, 4-7 return 2

//第一个位不算

num -= 1 << pow;

while (pow > 0)

{

if ((num & 1 << (pow - 1)) == 0)

root = root.Left;

else

root = root.Right;

pow--;

}

return root;

}

总结上面的3个步骤：

static BinNode FindParentNode(BinNode root, BinNode node1, BinNode node2)

{

int pos1 = 1;

GetPositionByNode(root, node1, ref pos1);

int pos2 = 1;

GetPositionByNode(root, node2, ref pos2);

int parentposition = 0;

if (pos1 >= pos2)

{

parentposition = FindParentPosition(pos1, pos2);

}

else //pos1<pos2

{

parentposition = FindParentPosition(pos2, pos1);

}

return GetNodeByPosition(root, parentposition);

}

### 5.如何不用递归实现二叉树的前序/后序/中序遍历？

算法思想：三种算法的思想都是让root的Left的Left的Left全都入栈。所以第一个while循环的逻辑，都是相同的。

下面详细分析第2个while循环，这是一个出栈动作，只要栈不为空，就始终要弹出栈顶元素，由于我们之前入栈的都是Left节点，所以每次在出栈的时候，我们都要考虑Right节点是否存在。因为前序/后序/中序遍历顺序的不同，所以在具体的实现上有略为区别。

1)前序遍历

这个是最简单的。

前序遍历是root->root.Left->root.Right的顺序。

因为在第一个while循环中，每次进栈的都可以认为是一个root，所以我们直接打印，然后root.Right和root.Left先后进栈，那么出栈的时候，就能确保先左后右的顺序。

static void PreOrder(BinNode root)

{

Stack stack = new Stack();

BinNode temp = root;

//入栈

while (temp != null)

{

Console.WriteLine(temp.Element);

if (temp.Right != null)

stack.Push(temp.Right);

temp = temp.Left;

}

//出栈，当然也有入栈

while (stack.Count > 0)

{

temp = (BinNode)stack.Pop();

Console.WriteLine(temp.Element);

while (temp != null)

{

if (temp.Right != null)

stack.Push(temp.Right);

temp = temp.Left;

}

}

}

//后序遍历比较麻烦，需要记录上一个访问的节点，然后在本次循环中判断当前节点的Right或Left是否为上个节点，当前节点的Right为null表示没有右节点。

static void PostOrder(BinNode root)

{

Stack stack = new Stack();

BinNode temp = root;

//入栈

while (temp != null)

{

if (temp != null)

stack.Push(temp);

temp = temp.Left;

}

//出栈，当然也有入栈

while (stack.Count > 0)

{

BinNode lastvisit = temp;

temp = (BinNode)stack.Pop();

if (temp.Right == null || temp.Right == lastvisit)

{

Console.WriteLine(temp.Element);

}

else if (temp.Left == lastvisit)

{

stack.Push(temp);

temp = temp.Right;

stack.Push(temp);

while (temp != null)

{

if (temp.Left != null)

stack.Push(temp.Left);

temp = temp.Left;

}

}

}

}

//中序遍历，类似于前序遍历

static void InOrder(BinNode root)

{

Stack stack = new Stack();

BinNode temp = root;

//入栈

while (temp != null)

{

if (temp != null)

stack.Push(temp);

temp = temp.Left;

}

//出栈，当然也有入栈

while (stack.Count > 0)

{

temp = (BinNode)stack.Pop();

Console.WriteLine(temp.Element);

if (temp.Right != null)

{

temp = temp.Right;

stack.Push(temp);

while (temp != null)

{

if (temp.Left != null)

stack.Push(temp.Left);

temp = temp.Left;

}

}

}

}

### 6.在二叉树中找出和为某一值的所有路径

算法思想：这道题目的苦恼在于，如果用递归，只能打出一条路径来，其它符合条件的路径打不出来。

为此，我们需要一个Stack，来保存访问过的节点，即在对该节点的递归前让其进栈，对该节点的递归结束后，再让其出栈——深度优先原则（DFS）。

此外，在递归中，如果发现某节点（及其路径）符合条件，如何从头到尾打印是比较头疼的，因为DFS使用的是stack而不是queue，为此我们需要一个临时栈，来辅助打印。

static void FindBinNode(BinNode root, int sum, Stack stack)

{

if (root == null)

return;

stack.Push(root.Element);

//Leaf

if (root.IsLeaf())

{

if (root.Element == sum)

{

Stack tempStack = new Stack();

while (stack.Count > 0)

{

tempStack.Push(stack.Pop());

}

while (tempStack.Count > 0)

{

Console.WriteLine(tempStack.Peek());

stack.Push(tempStack.Pop());

}

Console.WriteLine();

}

}

if (root.Left != null)

FindBinNode(root.Left, sum - root.Element, stack);

if (root.Right != null)

FindBinNode(root.Right, sum - root.Element, stack);

stack.Pop();

}

### 7.怎样编写一个程序，把一个有序整数数组放到二叉树中？

算法思想：我们该如何构造这棵二叉树呢？当然是越平衡越好，如下所示：

//// arr[0]

//// arr[1] arr[2]

//// arr[3] arr[4] arr[5]

相应编码如下：

public static void InsertArrayIntoTree(int[] arr, int pos, ref BinNode root)

{

root = new BinNode(arr[pos], null, null);

root.Element = arr[pos];

//if Left value less than arr length

if (pos \* 2 + 1 > arr.Length - 1)

{

return;

}

else

{

InsertArrayIntoTree(arr, pos \* 2 + 1, ref root.Left);

}

//if Right value less than arr length

if (pos \* 2 + 2 > arr.Length - 1)

{

return;

}

else

{

root.Right = new BinNode(arr[pos \* 2 + 2], null, null);

InsertArrayIntoTree(arr, pos \* 2 + 2, ref root.Right);

}

}

### 8.判断整数序列是不是二叉搜索树的后序遍历结果

比如，给你一个数组： int a[] = [1, 6, 4, 3, 5] ，则F(a) => false

算法思想：在后续遍历得到的序列中，最后一个元素为树的根结点。从头开始扫描这个序列，比根结点小的元素都应该位于序列的左半部分；从第一个大于跟结点开始到跟结点前面的一个元素为止，所有元素都应该大于跟结点，因为这部分元素对应的是树的右子树。根据这样的划分，把序列划分为左右两部分，我们递归地确认序列的左、右两部分是不是都是二元查找树。

由于不能使用动态数组，所以我们每次递归都使用同一个数组arr，通过start和length来模拟“部分”数组。

public static bool VerifyArrayOfBST(int[] arr, int start, int length)

{

if (arr == null || arr.Length == 0 || arr.Length == 1)

{

return false;

}

int root = arr[length + start - 1];

int i = start;

for (; i < length - 1; i++)

{

if (arr[i] >= root)

break;

}

int j = i;

for (; j < length - 1; j++)

{

if (arr[j] < root)

return false;

}

bool left = true;

if (i > start)

{

left = VerifyArrayOfBST(arr, start, i - start);

}

bool right = true;

if (j > i)

{

right = VerifyArrayOfBST(arr, i, j - i + 1);

}

return left && right;

}

### 9.求二叉树的镜像

算法1：利用上述遍历二叉树的方法（比如说前序遍历），把访问操作修改为交换左右节点的逻辑：

static void PreOrder(ref BinNode root)

{

if (root == null)

return;

//visit current node

BinNode temp = root.Left;

root.Left = root.Right;

root.Right = temp;

PreOrder(ref root.Left);

PreOrder(ref root.Right);

}

算法2：使用循环也可以完成相同的功能。

static void PreOrder2(ref BinNode root)

{

if (root == null)

return;

Stack stack = new Stack();

stack.Push(root);

while (stack.Count > 0)

{

//visit current node

BinNode temp = root.Left;

root.Left = root.Right;

root.Right = temp;

if (root.Left != null)

stack.Push(root.Left);

if (root.Right != null)

stack.Push(root.Right);

}

}

### 10.一棵排序二叉树（即二叉搜索树BST），令 f=(最大值+最小值)/2，设计一个算法，找出距离f值最近、大于f值的结点。复杂度如果是O(n2)则不得分。

算法思想：最小最大节点分别在最左下与最右下节点，O(N)

static BinNode Find(BinNode root)

{

BinNode min = FindMinNode(root);

BinNode max = FindMaxNode(root);

double find = (double)(min.Element + max.Element) / 2;

return FindNode(root, find);

}

static BinNode FindMinNode(BinNode root)

{

BinNode min = root;

while (min.Left != null)

{

min = min.Left;

}

return min;

}

static BinNode FindMaxNode(BinNode root)

{

BinNode max = root;

while (max.Right != null)

{

max = max.Right;

}

return max;

}

递归寻找BST的节点，O(logN)。

static BinNode FindNode(BinNode root, double mid)

{

//如果小于相等，则从右边找一个最小值

if (root.Element <= mid)

{

if (root.Right == null)

return root;

BinNode find = FindNode(root.Right, mid);

//不一定找得到

return find.Element < mid

root : find;

}

//如果大于，则找到Left

else //temp.Element > find

{

if (root.Left == null)

return root;

BinNode find = FindNode(root.Left, mid);

//不一定找得到

return find.Element < mid

root : find;

}

}

### 11.把二叉搜索树转变成排序的双向链表，如

//// 13

//// 10 15

//// 5 11 17

//// 16 22

转变为Link：5=10=11=13=15=16=17=22

算法思想：这个就是中序遍历啦，因为BST的中序遍历就是一个从小到大的访问顺序。局部修改中序遍历算法，于是有如下代码：

static void ConvertNodeToLink(BinNode root, ref DoubleLink link)

{

if (root == null)

return;

BinNode temp = root;

if (temp.Left != null)

ConvertNodeToLink(temp.Left, ref link);

//visit current node

link.Next = new DoubleLink(link, null, root);

link = link.Next;

if (temp.Right != null)

ConvertNodeToLink(temp.Right, ref link);

}

但是我们发现，这样得到的Link是指向双链表最后一个元素22，而我们想要得到的是表头5，为此，我们不得不额外进行while循环，将指针向前移动到表头：

static DoubleLink ReverseDoubleLink(BinNode root, ref DoubleLink link)

{

ConvertNodeToLink(root, ref link);

DoubleLink temp = link;

while (temp.Prev != null)

{

temp = temp.Prev;

}

return temp;

}

这么写有点蠢，为什么不直接在递归中就把顺序反转呢？于是有算法2：

算法2：观察算法1的递归方法，访问顺序是Left -> Root –> Right，所以我们要把访问顺序修改为Right -> Root –> Left。

此外，算法的节点访问逻辑，是连接当前节点和新节点，同时指针link向前走，即5=10=11=13=15=16=17=22=link

代码如下所示：

link.Next = new DoubleLink(link, null, root);

link = link.Next;

那么，即使我们颠倒了访问顺序，新的Link也只是变为：22=17=16=15=13=11=10=5=link。

为此，我们修改上面的节点访问逻辑——将Next和Prev属性交换：

link.Prev = new DoubleLink(null, link, root);

link = link.Prev;

这样，新的Link就变成这样的顺序了：link=5=10=11=13=15=16=17=22

算法代码如下所示：

static void ConvertNodeToLink2(BinNode root, ref DoubleLink link)

{

if (root == null)

return;

BinNode temp = root;

if (temp.Right != null)

ConvertNodeToLink2(temp.Right, ref link);

//visit current node

link.Prev = new DoubleLink(null, link, root);

link = link.Prev;

if (temp.Left != null)

ConvertNodeToLink2(temp.Left, ref link);

}

以下算法属于二叉树的基本概念，未列出：

1.Huffman Tree的生成、编码和反编码

2.BST的实现

3.Heap的实现，优先队列

4.非平衡二叉树如何变成平衡二叉树？

http://www.cppblog.com/bellgrade/archive/2009/10/12/98402.html

玩二叉树，基本都要用到递归算法。

唉，对于递归函数，我一直纠结，到底要不要返回值？到底先干正事还是先递归？到底要不要破坏原来的数据结构？到底要不要额外做个stack/queue/link/array来转存，还是说完全在递归里面实现？到底终结条件要写成什么样子？ ref在递归里面貌似用的很多哦。