# CSharp\_单链表

1. 结构定义
2. 单个结点的定义：

public class Node<T> {

public T value;

public Node<T> next;

public Node(T value, Node<T> next) {

this.value = value;

this.next = next;

}

public override string ToString() {

return value.ToString() + " ";

}

}

1. 单链表的定义：

class MyLinkList<T>{ //单链表

private Node<T> headNode;//头结点

private int count;//结点数量

public int Count {

get {

return count;

}

}

//比较器用于合并两个链表

private Comparer<T> comparer;

//初始化

public MyLinkList() {

headNode = null;

count = 0;

comparer = Comparer<T>.Default;

}

}

1. 方法
2. 增、删、查
   1. 查找：从前往后循环遍历查找是否包含此值
3. //检查链表中是否包含这些值（查看）
4. public bool Contains(T value) {
5. if (value == null) throw new *ArgumentNullException*();
6. else {
7. for(Node<T> current = headNode; current!=null;current = current.next) {
8. if (value.Equals(current.value)) {
9. return true;
10. }
11. }
12. return false;
13. }
14. }
    1. 增加：从尾部查找，先判断头结点是否存在，再遍历到
15. //添加链表由值,返回插入的尾部结点(增加)
16. public Node<T> Add(T value) {
17. if (value == null) throw new *ArgumentNullException*();
18. else if(headNode == null) {
19. //如果头结点为空，构建头结点并返回
20. headNode = new Node<T>(value, null);
21. count++;
22. return headNode;
23. }
24. else {
25. if (!Contains(value)) {//如果值不存在则尾部插入
26. Node<T> node = headNode;
27. while (node.next != null)
28. node = node.next;
29. node.next = new Node<T>(value, null);
30. count++;
31. return node.next;
32. }
33. }
34. return null;
35. }
36. b) 删除：从尾部查找，先判断头结点是否存在，再遍历到

//移除结点由值（删除）

1. public bool Remove(T value) {
2. if (value == null) throw new *ArgumentNullException*();
3. else if (headNode == null) return false;
4. //先看表头是不是
5. if (value.Equals(headNode.value)) {
6. headNode = headNode.next;
7. count--;
8. return true;
9. }
10. //不是表头
11. Node<T> node = headNode;
12. while(node.next != null) {
13. if (value.Equals(node.next.value)) {
14. node.next = node.next.next;
15. count--;
16. return true;
17. }
18. node = node.next;
19. }
20. return false;
21. }
22. 判断链表是否有环：
    1. 原理：此方法也可以用一个更生动的例子来形容：在一个环形跑道上，两个运动员在同一地点起跑，一个运动员速度快，一个运动员速度慢。当两人跑了一段时间，速度快的运动员必然会从速度慢的运动员身后再次追上并超过，原因很简单，因为跑道是环形的。
    2. 算法流程图：

c)代码示例：

//判断链表是否有环

public bool HasCircle() {

Node<T> slowNode = headNode;

Node<T> fastNode = headNode;

//使用快慢指针相遇则有环

while(fastNode !=null&& fastNode.next != null) {

slowNode = slowNode.next;

fastNode = fastNode.next.next;

if (slowNode.value.Equals(fastNode.value)) {

return true;

}

}

return false;

}

1. 查找入环点：
   1. 实现思路：首先判断是否有环，然后找到入环点。如何找到入环点，这里需要一点推导：
      1. 假设链表环的长度为r，慢指针走了s步，快指针就走了2s步。当快慢指针相遇时，2s = s + r，得到s = r。
      2. 假设链表的总长度为L，环入口与相遇点距离为x，表头到环入口点的距离为a，得出a + x = s，根据上面的结论，得到a + x = r，交换一下得到a = r - x。
      3. 同时，L - a = r，也就是链表总长度减去表头到环入口点的距离等于环的长度，这个直接可以得到的结论。
      4. 因此，a = L - a - x。（L - a - x）就是相遇点到环入口的距离。

最终得到结论：表头到环入口点的距离等于相遇点到环入口的距离。

根据上面的结论，我们可以从链表头结点、相遇点分别设置一个指针，每次走一步，两个指针必然相遇，而且相遇第一点就是环入口点。

b) 代码示例：//找到环链表的环入口点

public Node<T> FindEnterCircleNode() {

bool hasCircle = false;

Node<T> slowNode = headNode;

Node<T> fastNode = headNode;

//使用快慢指针相遇则有环

while (fastNode != null && fastNode.next != null) {

slowNode = slowNode.next;

fastNode = fastNode.next.next;

if (slowNode.value.Equals(fastNode.value)) {

hasCircle = true;

break;

}

}

if (hasCircle) {

fastNode = headNode;

while (slowNode.value.Equals(fastNode.value)) {

return fastNode;

}

slowNode = slowNode.next;

fastNode = fastNode.next;

}

return null;

}一些文字和图案

描述已自动生成