# 作业五线性表的链式表示2

## 请简述单链表是如何实现删除第i个结点的。

1. 创建一个临时节点作为头节点的引用。
2. 遍历到第i-1个节点。
3. 更改第i-1个节点的next引用，使其指向第i+1个节点，从而跳过第i个节点。

## 2、请解释以下代码的含义：

public void delete (StuNode p){

 if(head==null) return;

 StuNode sl = head;

 while(sl.getNext()!=nul1){

 if(sl.getNext()==p){

 s1.setNext(p.getNext());

 break;

 }

 sl = sl.getNext();

 }

 }

这段代码是一个名为 delete 的方法，它的功能是在链表中删除一个特定的节点 p。以下是这段代码的详细解释：

1. public void delete (StuNode p): 这是一个公共方法，名为 delete，它接受一个类型为 StuNode 的参数 p。这个参数 p 是你想从链表中删除的节点。
2. if(head==null) return;: 这一行代码检查链表是否为空（即头节点 head 是否为 null）。如果链表为空，则方法立即返回，不执行任何操作。
3. StuNode sl = head;: 这一行代码创建了一个新的 StuNode 类型的变量 sl，并将其初始化为链表的头节点 head。这个 sl 变量用于遍历链表。
4. while(sl.getNext()!=nul1){: 这是一个 while 循环，只要 sl 节点的下一个节点不是 null，循环就会继续。这个循环用于遍历链表。
5. if(sl.getNext()==p){: 这是一个 if 语句，检查 sl 节点的下一个节点是否是要删除的节点 p。
6. s1.setNext(p.getNext());: 如果 sl 节点的下一个节点就是要删除的节点 p，那么这行代码就会将 sl 节点的 next 指针设置为 p 节点的下一个节点。这样，p 节点就从链表中被移除了。
7. break;: 这行代码会跳出 while 循环。因为已经找到并删除了节点 p，所以没有必要继续遍历链表。
8. sl = sl.getNext();: 这行代码将 sl 节点移动到链表中的下一个节点。这是 while 循环的最后一部分，用于遍历链表。

## 3、请简述双向链表是如何实现插入和删除一个结点的。

双向链表是一种更复杂的线性表，它的每个节点都有两个链接，一个指向前一个节点，另一个指向后一个节点。

### 插入操作：

1. 创建一个新的节点。

2. 如果链表为空，将新节点设置为头节点。

3. 如果链表不为空，将新节点的前驱链接设置为给定节点的前驱链接，将新节点的后继链接设置为给定节点。

4. 如果给定节点的前驱链接不为空，将给定节点的前驱链接的后继链接设置为新节点。

5. 将给定节点的前驱链接设置为新节点。

### 删除操作：

1. 如果链表为空或给定节点为空，不进行任何操作。

2. 如果给定节点的前驱链接不为空，将给定节点的前驱链接的后继链接设置为给定节点的后继链接。

3. 如果给定节点的后继链接不为空，将给定节点的后继链接的前驱链接设置为给定节点的前驱链接。

4. 删除给定节点。

以上就是双向链表的插入和删除操作的简单描述。在实际编程中，还需要考虑一些边界条件和异常处理。

## 4、请解释以下代码的含义：

public void insert(StuDouNode p,StuDouNode s){

 StuDouNode pp =p.getPrior();

 s.setPrior(pp);

 pp.setNext(s);

 s.setNext(p);

 p.setPrior(s);

}

这段代码是在一个双向链表中插入一个新的节点。具体来说，它在节点 `p` 之前插入节点 `s`。以下是每行代码的详细解释：

1. `StuDouNode pp = p.getPrior();`：获取节点 `p` 的前一个节点 `pp`。

2. `s.setPrior(pp);`：将新节点 `s` 的前一个节点设置为 `pp`。

3. `pp.setNext(s);`：将 `pp` 的下一个节点设置为新节点 `s`。

4. `s.setNext(p);`：将新节点 `s` 的下一个节点设置为 `p`。

5. `p.setPrior(s);`：将节点 `p` 的前一个节点设置为新节点 `s`。

总的来说，这段代码的目的是将新节点 `s` 插入到节点 `p` 之前，同时保持链表的完整性。这是双向链表插入操作的典型实现。

## 5、请解释以下代码的含义：

public void delete(StuDouNode p){

 if(head==nul1) return;

 StuDouNode sl = head;

 while(sl.getNext()!=null){

 if(sl.getNext()==p){

 sl.setNext(p.getNext());

 if(p.getNext()!=null)

 p.getNext().setPrior(sl);

 break;}

 sl=sl.getNext();}

}

这段代码是一个用于删除双向链表中指定节点的方法。以下是每行代码的详细解释：

* `public void delete(StuDouNode p)`: 这是一个公共方法，名为`delete`，它接受一个类型为`StuDouNode`的参数`p`。这个参数`p`是你想从双向链表中删除的节点。
* `if(head==nul1) return;`: 如果链表的头节点`head`为空（这里应该是`null`，而不是`nul1`），那么这个方法就会立即返回，不执行任何操作。
* `StuDouNode sl = head;`: 创建一个新的`StuDouNode`类型的变量`sl`，并将其初始化为链表的头节点。
* `while(sl.getNext()!=null)`: 这是一个循环，只要`sl`节点的下一个节点不为空，就会继续执行循环。
* `if(sl.getNext()==p)`: 如果`sl`节点的下一个节点就是要删除的节点`p`，那么就执行以下的操作。
* `sl.setNext(p.getNext());`: 将`sl`节点的下一个节点设置为`p`节点的下一个节点，这样就跳过了`p`节点，相当于从链表中删除了`p`节点。
* `if(p.getNext()!=null) p.getNext().setPrior(sl);`: 如果`p`节点的下一个节点不为空，那么就将`p`节点的下一个节点的前一个节点设置为`sl`节点。这样就完成了双向链表的链接。
* `break;`: 删除节点后，跳出循环。
* `sl=sl.getNext();`: 如果`sl`节点的下一个节点不是要删除的节点，那么就将`sl`节点移动到下一个节点，继续循环。

总的来说，这个方法就是在双向链表中找到指定的节点并删除它。如果链表为空或者没有找到指定的节点，那么这个方法就不会做任何事情。

6、请解释以下代码的含义：

 public SeqList<int> Purge(SeqList<int> La){

 SeqList<int> Lb = new SeqList<int>(La.Maxsize);

 Lb.Append(La[0]);

 for (int i=1; i<=La.GetLength()-l; ++i){

 int j = 0;

 for(j=0;j<=Lb.GetLength()-1; ++j){

 if(La[i].CompareTo(Lb[j])==0)break;}

 if(j>Lb.GetLength()-1){

 Lb.Append(La[i]);}

 }

 return Lb;

}

这段代码是在实现一个功能，即从输入的序列La中移除重复的元素，并返回一个新的序列Lb。以下是每行代码的详细解释：

* public SeqList<int> Purge(SeqList<int> La){：定义一个名为Purge的公共方法，该方法接受一个整数类型的SeqList（名为La）作为参数，并返回一个整数类型的SeqList。
* SeqList<int> Lb = new SeqList<int>(La.Maxsize);：创建一个新的SeqList（名为Lb），其最大大小与La相同。
* Lb.Append(La[0]);：将La的第一个元素添加到Lb。
* for (int i=1; i<=La.GetLength()-l; ++i){：遍历La中的每个元素，从第二个元素开始。
* int j = 0;：初始化一个计数器j。
* for(j=0;j<=Lb.GetLength()-1; ++j){：遍历Lb中的每个元素。
* if(La[i].CompareTo(Lb[j])==0)break;：如果La中的元素已经存在于Lb中，则跳出内层循环。
* if(j>Lb.GetLength()-1){：如果j大于Lb的长度减一，说明La中的元素不在Lb中。
* Lb.Append(La[i]);}：将La中的元素添加到Lb。
* return Lb;：返回Lb，即移除了重复元素的La。

7、请解释以下代码的含义：

 public void ReversLinkList(LinkList<int> H){

 Node<int> p=H.Next;

 Node<int> q = new Node<int>();

 H.Next=null;

 while (p != null){

 q= p;

 p=p.Next;

 q.Next = H.Next;

 H.Next =q;

 }

 }

这段代码是一个函数，名为ReversLinkList，它的功能是反转一个链表。这里是它的工作原理：

1. Node<int> p=H.Next;：创建一个新的节点p，并将其设置为链表H的下一个节点。这是将要遍历的链表的起始点。
2. Node<int> q = new Node<int>(); H.Next=null;：创建一个新的节点q，并将链表H的下一个节点设置为null。这是因为将要创建一个新的反转链表，所以需要从空链表开始。
3. while (p != null){...}：这是一个循环，只要p不为null，就会一直执行。这意味着只要还没有到达原始链表的末尾，就会一直执行这个循环。
4. 在循环内部，首先将q设置为p，然后将p设置为p.Next。这样就可以在不丢失对当前节点的引用的情况下，向前移动p。
5. 然后，将q.Next设置为H.Next，并将H.Next设置为q。这实际上是在新链表的开始处插入了当前节点，从而实现了链表的反转。
6. 当遍历完整个链表后，p将会是null，循环将结束。此时，H.Next将指向反转后的链表的开始，就得到了一个反转后的链表。