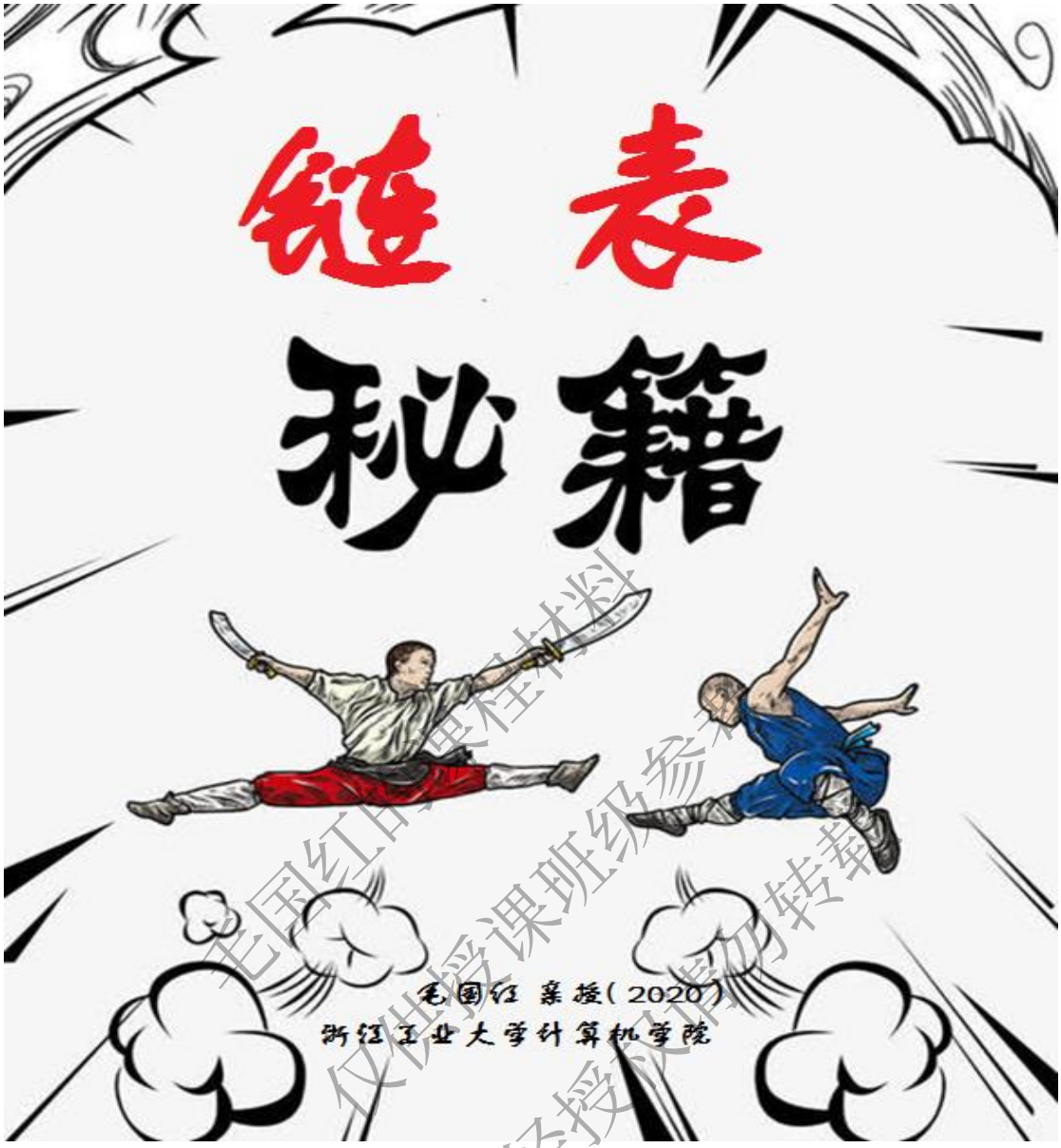
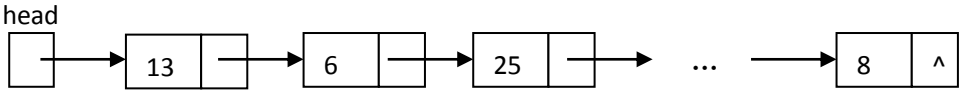


高级篇

第五部分：链表之炉火纯青

链结点定义同前

对链表的假设：链表由 head 指针管理，可空。



一、链表逆置

```
//保留原始链表，做新链表 一个遍历，一个对应拷贝
node* ReverseNewList( node* head)
{
        node* NewHead=nullptr;
        p=head;
        while(p){
            node* Nnode =new node(p->data);
            Nnode->next=NewHead; //插入新结点在首
            NewHead=Nnode;

            p=p->next;
        }
      return NewHead;
  }
```

讨论函数封装。

```
//利用原有链表结点建逆置链表
node* ReverseList( node* head)
{
        node* p=head;
        head=nullptr;
        while(p){
            node* t=p;
            p=p->next;
            t->next=head;
            head=t;
        }
      return head;
  }
```

讨论函数封装。

注意：函数封装的问题，对 ReverseList 函数若设计返回类型为 void,有问题,拷贝传递参数,head 值在函数内被修改,无法影响实参值,解决途径 1) 如示范,给出返回类型 node*,返回更新后的表首；2)设计引用参数，即 node*& head 指针的引用，用于实参同步函数内的修改。只要动作可能涉及表首的更新，就都需要考虑这样的问题。

有没有其他方法？考虑：（顺序访问链表，将链表内数据存储数组， 按数组逆序访问写数据到链表）

二、 删除链表内所有 value 点（0-多个 value 值）

从删除一个点开始考虑。

回顾删除一个点：

```
node* pre=nullptr, *p=head;
while(p&&p->data!=value){ pre=p; p=p->next; }
if(p){
    if(pre) pre->next=p->next;   //找到的 p 有前驱
    else head=p->next; //找到的 p 是表头 或者 head=head->next
    delete p;
}
```

带回溯的删除策略（直接在删除一个点的基础上修改）

```
node* DeleteAllValuesWithBack(node *head,DataType value){
  while(1){
    node* pre=nullptr, *p=head;
    while(p&&p!=value){ pre=p; p=p->next; }
    if(p){
      if(pre) pre->next=p->next;   //找到的 p 有前驱
      else head=p->next; //找到的 p 是表头 或者 head=head->next
      delete p;
     }
     else break;   //再也找不到匹配点
  }
  return head;
  }
```

无回溯的策略

删除可能的多个点（p 一直向前查找待删除点）

```
    node* DeteleAllValues(node* head, dataType value)
    {
            node* pre=nullptr, *p=head;
            while(p){
                if(p->data!=value) {   pre=p;   p=p->next; }
                else//p->data==value
               {
                        if(pre) { //找到的 p 有前驱
                              pre->next=p->next;
                              delete p;
                              p=pre->next; //为下次查找作准备
                        }
                        else { //找到的 p 是表头
```

```
                    head=p->next ;
                    delete p;
                    p=head; //为下一次的查找作准备
                }
                //可以写公共语句 delete p;
            }
            return head;
        }
```

三、链表排序

1) 利用单结点有序添加到链表

```
node* SortList(node* head)
{
    node* p=head;
    head=nullptr; //排序结果
    while(p){
        node* t=p;   p=p->next;
        head=InsertList(head,t);   //利用有序插入
    }
    return   head;
}
```

回顾前期：链表的有序添加 Nnode：寻找第一个比添加值大的元素所在的位置，添加在其前

```
node*p,*pre; //pre 为 p 前一个结点的指针，两指针同步
p=head;
pre=nullptr;
while(p&&p->data<Nnode->data) { pre=p;   p=p->next; }
if(pre){   //p 不是头结点   pre!=nullptr   或者  p 不存在，插入在表尾
    Nnode->next=p;
    pre->next=Nnode;
}
else //p 为头结点或者表为空
{      Nnode->next=head;
       head=Nnode;
}
```

封装为函数：

```
node* InsertList(node* head,node* Nnode)
{
    node*p,*pre; //pre 为 p 前一个结点的指针，两指针同步
    p=head;
    pre=nullptr;
    while(p&&p->data<Nnode->data) { pre=p;   p=p->next; }
```

```
    if(pre){   //p 不是头结点   pre!=nullptr    或者  p 不存在，插入在表尾
        Nnode->next=p;
        pre->next=Nnode;
    }
    else //p 为头结点或者表为空
    {      Nnode->next=head;
           head=Nnode;
    }
    return head;
}
```

2)选择排序

```
    void   SelectSortList(node* head)
    {
          node* p=head;
          while(p){   //考虑 while(p->next)
                node* min=p;
                node* other=p->next;
                while(other){
                   if(other->data<min->data) min=other;
                   other=other->next;
                }
                if(min!=p) {
                   dataType temp=p->data;
                   p->data=min->data;
                   min->data=temp;
                }
                p=p->next;
          }
    }
```

3）考虑 ：大数沉底的冒泡策略(作业)

四、两个链表合并(集合并)

1）一般合并（不考虑表数据集合特征）

```
void Union2List(node*& L1,node*& L2)//L2 向 L1 合并,返回 L1 作为合并后的表头，参数也可以使用二级指针，在用法上需要注意。如设计为 node* *L1，则实际的表头为*L1
{
    if(L1==nullptr) { L1=L2;L2=nullptr; }
    else{
      node* tail=L1;
      node*p1,*p2;
      while(tail->next){ tail=tail->next;} //标记集合 1 尾部
```

```
        while(L2){
          p2=L2; //从 L2 中依次取结点
          L2=L2->next; //从 L2 中去掉取出点 L2=p2->next

          p1=L1; //检查 L2 中取出的结点 p2 的值是否已在合并集合中
          while(p1&&p1->data!=p2->data)   p1=p1->next;
          //按检查结果处理
          if(p1==nullptr){ p2->next=tail->next;   tail->next=p2;   tail=p2; } //不在，则该结点并入
          else delete p2; //已在，则删除该取出点
        }//while(L2) 结束
        }//else 结束

        return L1;
}
```

- 也可以将 L2 先检查一遍,处理成与 L1 无重复元素的,然后再合并。
- 拷贝合并不再赘述。

2）合并有序表，L1，L2 为有序表（升序），若无序，也可以先整理两个表，如：

```
L1=SortList(L1);   L2=SortList(L2);
node*   Merge2List(node* &L1,node* &L2) //利用原有表结点
{ node *L3,*tail3;   //合并结果 L3
  node* p;//临时指针
   tail3=L3=new node; //建立带哨兵的新链表，为了方便合并，思考哨兵的作用
   tail3->next=nullptr;

   while(L1&&L2){
        if(L1->data<L2->data) {
            tail3->next=L1;
            L1=L1->next;
            tail3=tail3->next;
            tail3->next=nullptr;

          }
        else if(L2->data<L1>data){
            tail3->next=L2;
            L2=L2->next;
            tail3=tail3->next;
            tail3->next=nullptr;
        }
        else //相等
        {     tail3->next=L1;
            L1=L1->next;
```

```
            tail3=tail3->next;
            tail3->next=nullptr;
            p=L2;
            L2=L2->next;
            delete p;
        }
         //tail3=tail3->next;
         //tail3->next=nullptr;
      }//while
      if(L1)   tail3->next=L1;
      if(L2)   tail3->next=L2;
      //处理哨兵
      p=L3;
      L3=L3->next;
      delete p;
      L1=L2=nullptr;
      return L3;
  }
```

- 也可以考虑 L2 向 L1 合并的方法，改成最后返回 L1，逐个遍历 L2 向 L1 有序插入：不同的插入，相同的删除。
- 或者,先合并集合,再排序(较慢的做法)
- 也可以考虑复制合并的方法（保留 L1，L2，生成新链 L3 为求解结果）

五、 两个链表（集合）的交（保留 L1，L2 的策略，求解 L3）：收集在 L1 中又在 L2 中的结点值

```
node*   IntersectionSet(node* L1,node* L2){
    node* L3, *tail3;
    node *p1,*p2;
    if(L1==nullptr||L2==nullptr) return nullptr;

    L3=tail3=new node;   tail3->next=nullptr;   //建立哨兵
    p1=L1;
    while(p1){
      p2=L2;
      while(p2&&p1->data!=p2->data)   p2=p2->next;
      if(p2) { //是共同元素
        node* Nnode=new node; //生成新结点，拷贝
        Nnode->data=p1->data;
        Nnode->data=nullptr;
        tail3->next=Nnode; tail3=Nnode;
      }
      p1=p1->next; //考察 L1 中的后续值
    }
```

```
//处理哨兵
p1=L3;
L3=L3->next;
delete p1;
return L3;
}
```

六、两个链表（集合）的差 L1-L2（保留 L1,L2，求解 L3）：收集 L1 中不在 L2 中的结点值

```
node*  DifferenceSet(node* L1,node* L2){
//作业
}
```