# 第二次作业

**2.11 设计算法以删除链表中值为 x 的元素结点。**

bool listDeleteX(node\* L, elementType x)

{

node\* u;

node\* p=L; //指向头结点

int succ=0; //是否删除成功标记，成功删除 succ=1，失败 succ=0。

while(p->next) {

if(p->next->data==x) {

//找到目标结点，删除此结点

//p 指向目标结点的前驱，p->next 指向待删除目标结点，并以 u 保存。

u=p->next;

p->next=u->next;//ai-1 的 next 指向 ai+1 节点，或为空（ai-1 为最后节点）

delete u; //释放目标结点占据的空间

succ=1; //标记结点成功删除

break; //退出循环

}

p=p->next;

}

if(succ==1)

return true;

//成功删除，返回 true else return false;

//删除失败，返回 false，x 不在链表中。

}

**【算法分析】时间复杂度 O(n)**

**2.18 假设递增有序的带头结点的单循环链表 A、B 分别表示两个集合，设计算法以求解 A= A ∪B，并分析算法的时间复杂度。**

void unionSet(node \*A, node \*B)

{

node\*pa,\*pb,\*u,\*R;

pa=A; //pa 初始化指向 A 的头结点，为了 A 表的插入处理

pb=B->next; //pb 初始化指向第一个元素结点

while(pa->next!=A && pb!=B)

{

if(pa->next->data<pb->data) //pa 为并集结点

pa=pa->next;

else if(pa->next->data>pb->data) //pa 有可能在 pb 的后面，pb 是交集结点，申请新结点插入 pa 之后

{

//pa 不动

u=new node

u->data=pb->data;

u->next=pa->next;

pa->next=u;

pb=pb->next;

}

else //pa->data==pb->data，pa、pb 同时后移

{

pa=pa->next;

pb=pb->next;

}

}

//如果 B 没有结束，pb 及其后结点皆为并集结点，插入 A 表

while(pb!=B)

{

u=new node;

u->data=pb->data;

u->next=pa->next;

pa->next=u;

pa=u; //pa 指尾结点

pb=pb->next;

}

}

**【算法分析】时间复杂度：O(|A|+|B|)。**

**2.22 设计算法将两个递增有序的带头结点的单链表 A、B 合并为一个递减有序的带头结点的单链表，并要求算法的时间复杂度为两个表长之和的数量级。**

node\* ListJoinAndReverse\_L(node \*A, node \*B)

{

node \*pa,\*pb,\*p,\*L;

//pa、pb 分别指向 A 和 B 待逆置的结点

L=A; //以 L 为合并后的头结点指针，即 A 的头结点指针

pa=A->next;

pb=B->next;

L->next=NULL;

while(pa!=NULL && pb!=NULL)

{

if(pa->data<pb->data)

{

p=pa->next; //p 指向 A 下一个未逆置结点

pa->next=L->next;

L->next=pa; //pa 作为已经合并部分的第一个结点，至此逆置完成

pa=p; //pa 指向 A 第一个未逆置结点

}

else

{

p=pb->next; //p 指向 B 下一个未逆置结点

pb->next=L->next; //pb->next 指向已经合并部分的第一个数据结点

L->next=pb; //pb 作为已经合并部分的第一个结点，至此逆置完成

pb=p; //pb 指向 A 第一个未逆置结点

}

}

//以下处理一个表结束，另一个表未结束情况

while(pa!=NULL && pb==NULL) //A 未结束，B 已经结束

{

p=pa->next;

pa->next=L->next;

L->next=pa;

pa=p;

while(pa==NULL && pb!=NULL) //A 已结束，B 未结束

{

p=pb->next;

pb->next=L->next;

L->next=pb;

pb=p;

}

return(L);

}

**2.24 设计算法将带头结点的双循环链表 L就地逆置，即利用原表各结点的空间实现逆置。**

void DLListReverse(node \*L)

{

node \*P, \*u; //P 指向当前待逆置的结点，u 指向未逆置的下一个结点

P=L->next; //P 指向原链表的首元素结点，此结点逆置后为尾结点，需要单独处理

if(P!=L) //非空链表，处理第一个结点

{

u=P;

p=p->next; //p 指向链表下一个结点

u->next=L; //建立 next 向循环

L->prior=u; //建立 prior 向循环

}

//循环逆置剩余结点

while(P!=L)

{

u=P;

p=p->next; //p 指未逆置部分下一个结点

u->prior=L; //u 头插成为首元素结点，前向指针指向头结点

u->next=L->next; //u->next 指向已逆置部分原来的首元素结点

L->next->prior=u; //已逆置部分原来首元素结点的 prior 指针指向 u

L->next=u; //L->next 指向 u，u 插入成为首元素结点

}

}