

2 线性表

董洪伟

<http://hwdong.com>

主要内容

- 线性表的类型定义
 - 即抽象数据类型
- 顺序实现
 - 即用一连续的存储空间来表示
- 链式实现
 - 即用链表实现
- 一元稀疏多项式
 - 链表实现

线性表的类型定义

- 线性表

- n 个元素的有限序列

数据项



元素
(记录)



姓名	学号	性别	年龄	班级	健康状况
王小林	790631	男	18	计91	健康
陈红	790632	女	20	计91	一般
刘建平	790633	男	21	计91	健康
...

线性表的类型定义

- 线性表的长度

- 元素的个数 n
- 如果 $n = 0$ ，则为空表

- 位序

- 非空线性表中的每个元素都有一个确定的位置， a_i 是第 i 个元素， i 称为数据元素 a_i 在线性表中的位序

线性表的抽象数据类型

ADT List{

数据关系: 数据元素具有线性关系 (a_1, a_2, \dots, a_n)

基本操作:

//构造一个空的线性表L

InitList(List &L)

//销毁线性表L

DestroyList(List &L)

//将L重置为空表

ClearList(List &L)

线性表的抽象数据类型

//L是空表吗？

`IsEmpty(List L)`

//返回L中数据元素的个数

`ListLength(List L)`

//把L中第i个元素的值赋给e

`GetElem(List L,int i, ElemType &e)`

//把e值赋给L中第i个元素

`SetElem(List L,int i, ElemType e)`

线性表的抽象数据类型

//查询L中第1个与e满足关系compare()

//的数据元素的位序

FindElem(List L, ElemType e, compare())

//在L中第i个位置之前插入新的数据元素e, L的长度加1

ListInsert(List &L,int i, ElemType e)

//删除L的第i个数据元素, 并把其值赋给e, L的长度减1

ListDelete(List &L,int i, ElemType &e)

//依次对L的每个数据元素调用函数visit()

ListTraverse(List L, visit())

}

线性表的顺序表示和实现

- 即用一连续的存储空间来表示

- 比如数组: **ElemType array[n];**

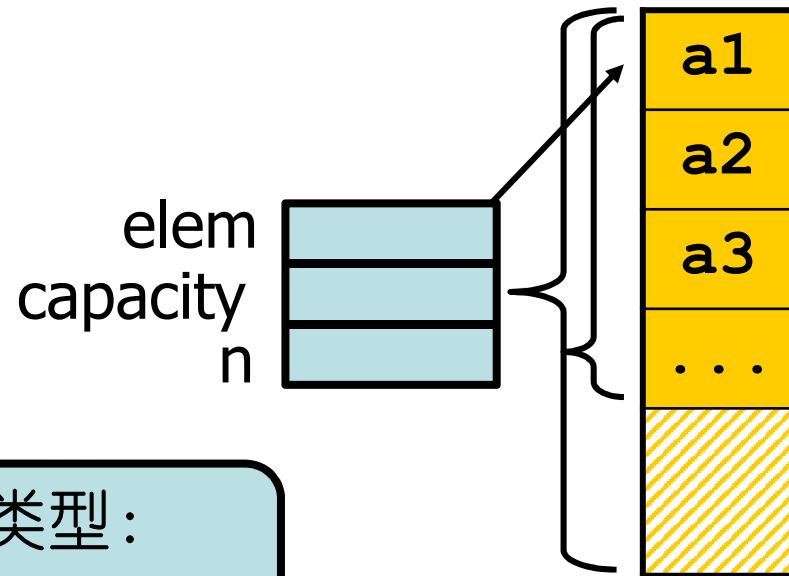
- 或动态分配的一块空间:

```
ElemType *array= (ElemType*)  
    malloc (n*sizeof (ElemType) );
```

```
ElemType *array= (ElemType*)  
    realloc (p,n*sizeof (ElemType) );
```


动态分配顺序存储结构

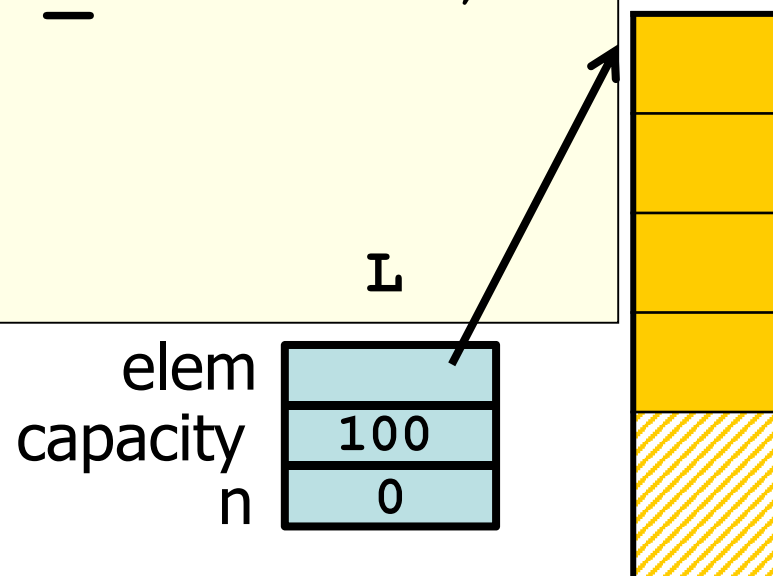
```
#define LIST_INIT_CAPACITY 100
#define LISTINCREMENT 10
typedef struct{
    ElemType *elem;
    int capacity;
    int n;
} SqList;
```



可以把ElemType定义为任何类型：
`typedef int ElemType;`

初始化

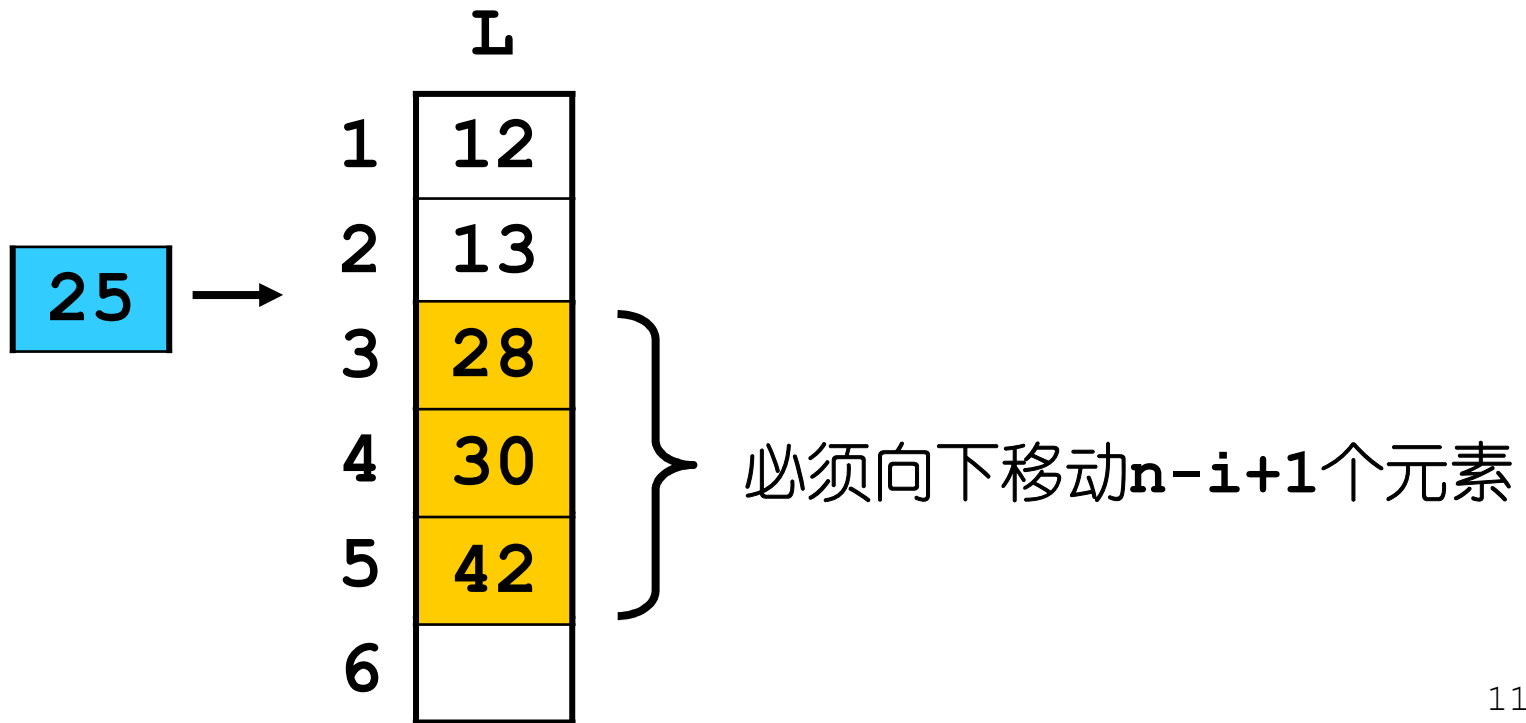
```
bool InitList_Sq(Sqlist &L) {  
    L.elem = (ElemType*)malloc  
            (LIST_INIT_CAPACITY *  
             sizeof(ElemType));  
    if (!L.elem)    return false;  
    L.capacity = LIST_INIT_CAPACITY;  
    L.n = 0;  
    return true;  
}
```



插入

- ListInsert(SqList& L, int i, ElemType e)
- 在线性表L的第i个元素前面，插入元素e

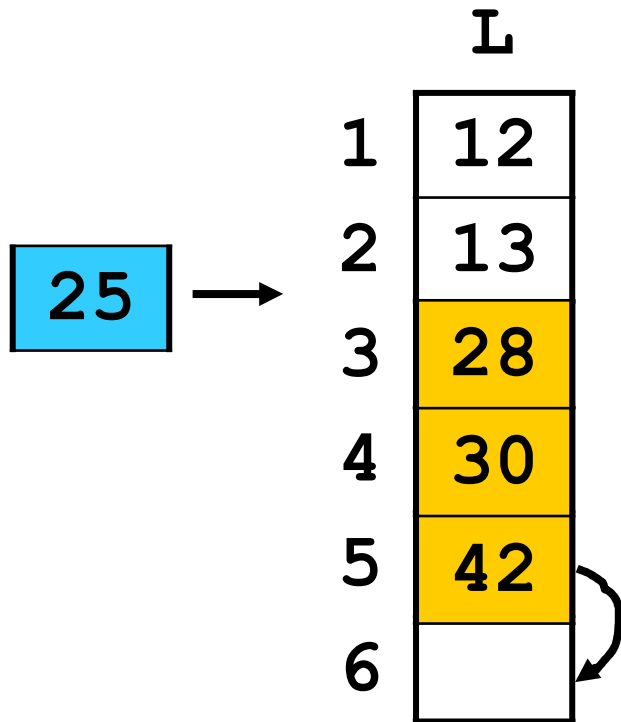
$$i = 3$$



插入

- ListInsert(SqList& L, int i, ElemType e)
- 在线性表L的第i个元素前面，插入元素e

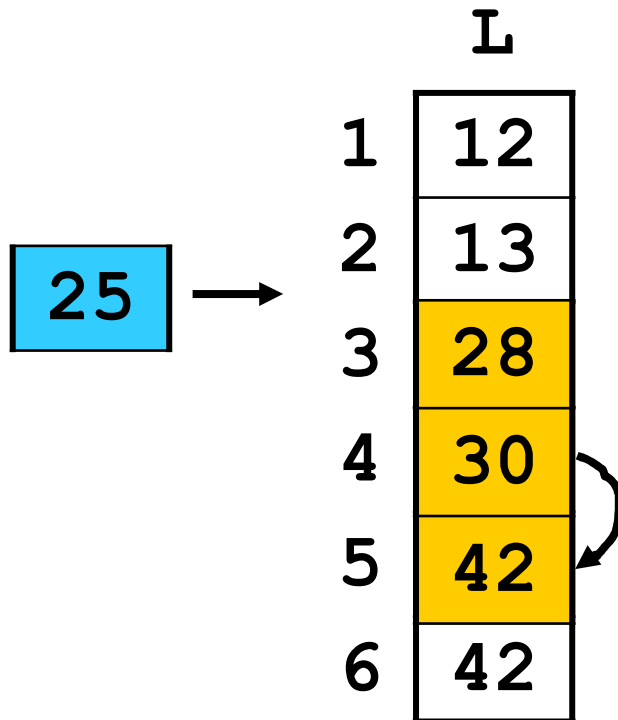
i = 3



插入

- ListInsert(SqList& L, int i, ElemType e)
- 在线性表L的第i个元素前面，插入元素e

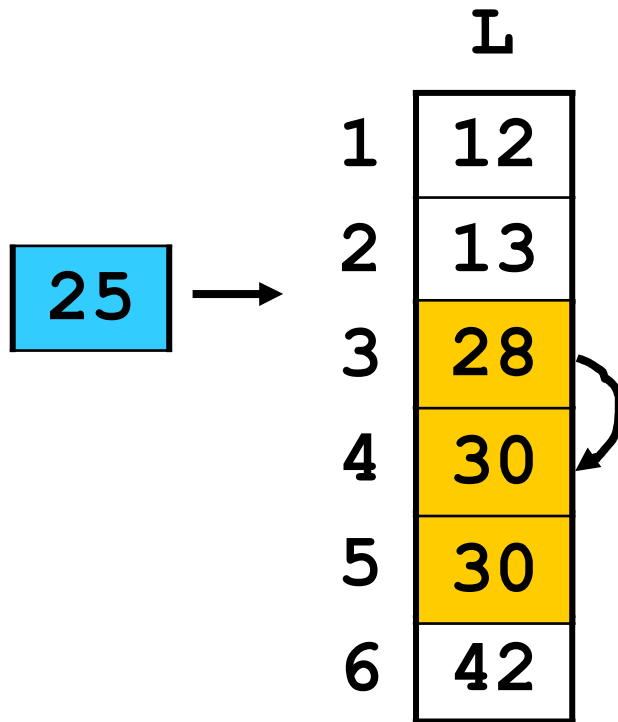
i = 3



插入

- ListInsert(SqList& L, int i, ElemType e)
- 在线性表L的第i个元素前面，插入元素e

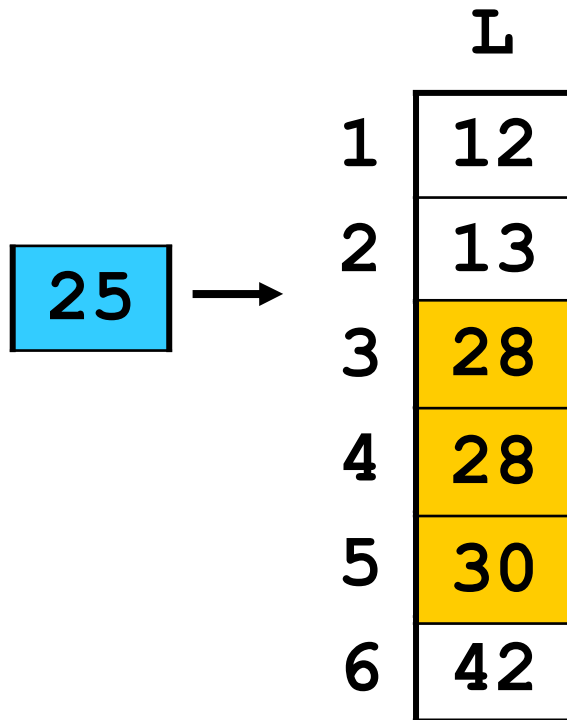
i = 3



插入

- ListInsert(SqList& L, int i, ElemType e)
- 在线性表L的第i个元素前面，插入元素e

i = 3



```
bool ListInsert_Sq(SqList &L, int i, ElemType e) {  
    // 判断i是否合法  
    if (i < 1 || i > L.n + 1)  
        return false;  
  
    // 若线性表空间不足, 再分配一些空间  
    if (L.n >= L.capacity)  
    {  
        ElemType *temp = (ElemType *)  
            realloc(L.elem,  
                (L.capacity+LISTINCREMENT) *  
                    sizeof(ElemType) );  
        if (temp==0) return false;  
        free(L.elem);  
        L.elem = temp;  
        L.capacity += LISTINCREMENT;  
    }  
}
```



```

// q指向插入的位置
ElemType * q = &(L.elem[i-1]);

// p指向最后一个元素,
// 从p到q的所有元素后移一个单元
ElemType *p;
for (p = &(L.elem[L.n - 1]);
      p >= q; --p)
    *(p+1) = *p;

*q = e;          // 写进待插入的元素e
++ L.n;         // 表长加1
return true;
}

```

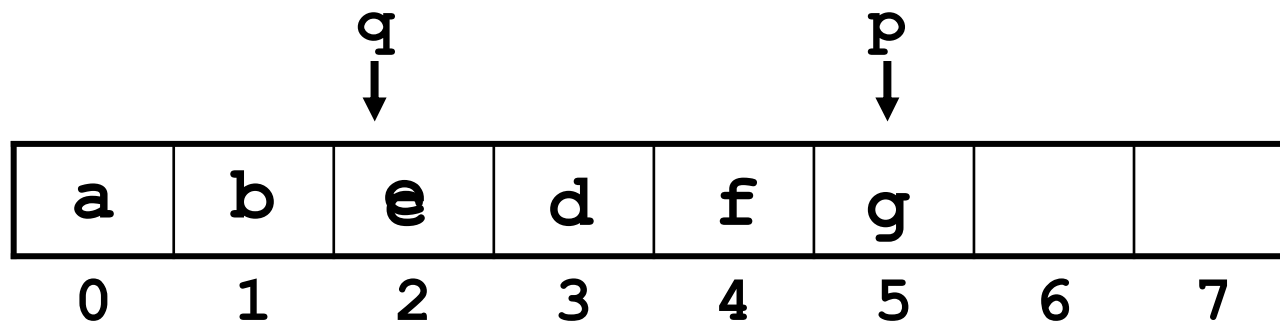
```

q = &(L.elem[i-1]);    // q指向插入的位置
// p指向最后一个元素,
// 从p到q的所有元素后移一个单元
for (p=&(L.elem[L.n-1]); p>=q; --p)
    *(p+1) = *p;

*q = e;                // 写进待插入的元素e
++ L.n;               // 表长加1
return true;
}

```

$i=3$, 即插入在第三个元素之前



插入操作的时间复杂度

- 很显然，插入操作的复杂度由需要移动的元素个数决定
- 而需要移动元素的个数由插入位置决定
 - $i = n+1$ 时，需要移动0个
 - $i = n$ 时：1个
 - ...
 - $i = 1$ 时：n个
 - 即：需要移动的元素个数 = $n+1-i$

插入操作的时间复杂度

- 最差情况

- $T(n) = O(n)$

- 平均情况呢？

- 一共有 $1, 2, \dots, n+1$, $n+1$ 个可能的插入位置, 在第 i 个位置上插入的概率是 $1/(n+1)$

- 所以平均需要移动元素的个数

$$= \frac{1}{n+1} \sum_{i=1}^{n+1} (n+1-i) = \frac{1}{n+1} \times \frac{n(n+1)}{2} = \frac{n}{2}$$

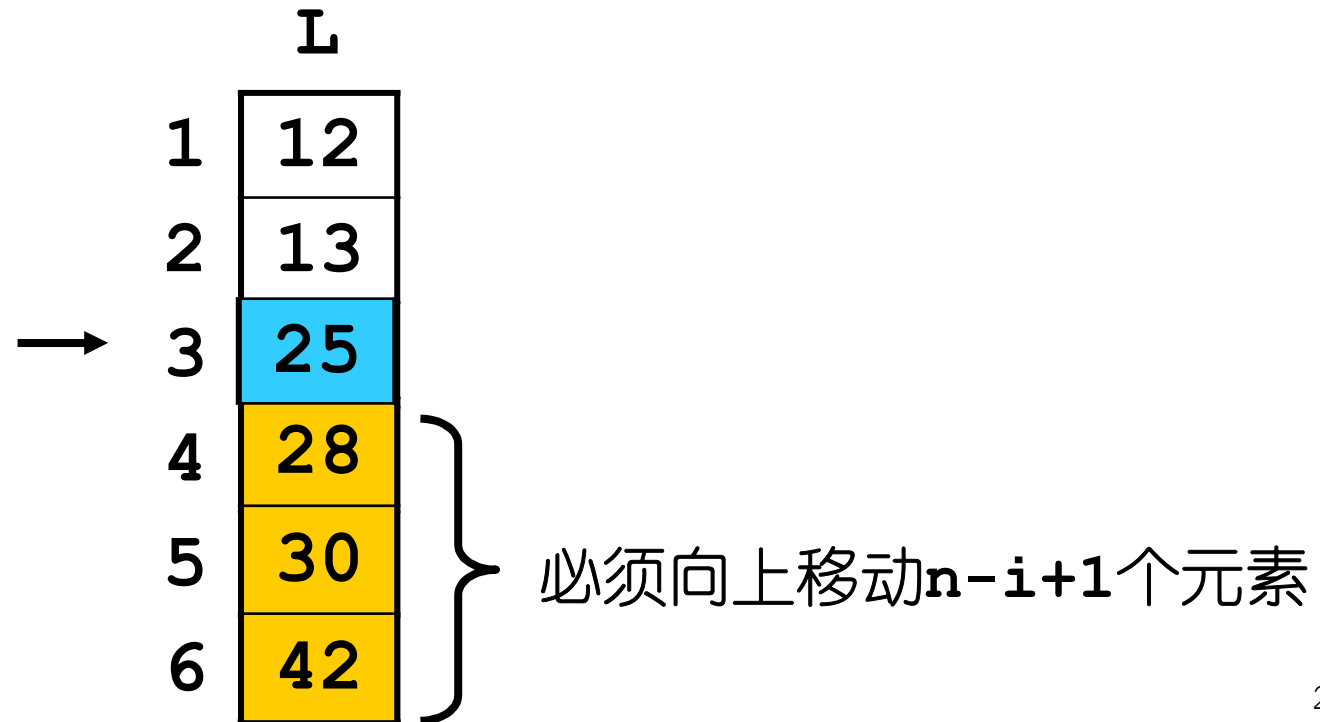
- 所以平均复杂度 = $O(n)$

删除

- `ListDelete(&L, i, &e)`

- 删除第 i 个元素，其值赋给 e

$i = 3$



```

bool ListDelete_Sq(SqList &L, int i, ElemType &e)
{
    if((i < 1) || (i > L.length))
        return false;
    ElemType *p,*q;
    p = &(L.elem[i-1]); //p指向被删除的节点
    e = *p;             //e得到被删除的节点的值

    // q指向最后一个节点
    q = L.elem + L.n - 1;

    // 从p+1到q的所有节点前移一个单元
    for(++ p; p <= q; ++ p)
        *(p-1) = *p;

    -- L.n;           // 表长减1
    return true;
}

```

删除操作的时间复杂度是多少？

特点及优缺点

- 特点

- 各单元的内存地址连续

- 优点

- 可随机访问任一元素
- 即访问任何一个元素所用时间都相同

- 缺点

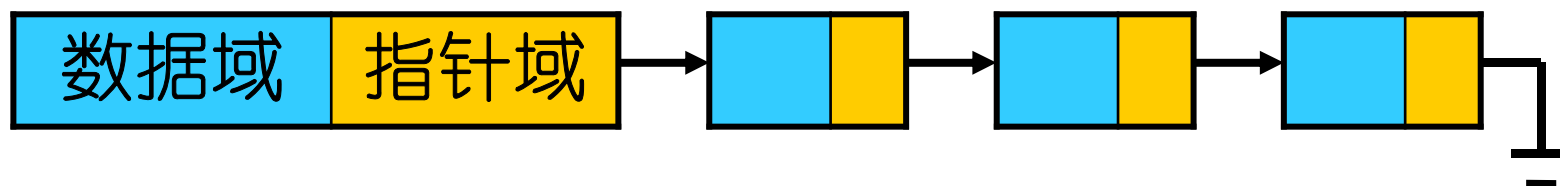
- 插入、删除操作需移动大量元素
- 算法复杂度 = $O(n)$

线性表的链式表示和实现

- 特点

- 每个元素的存储地址任意
- 使用指针相链接

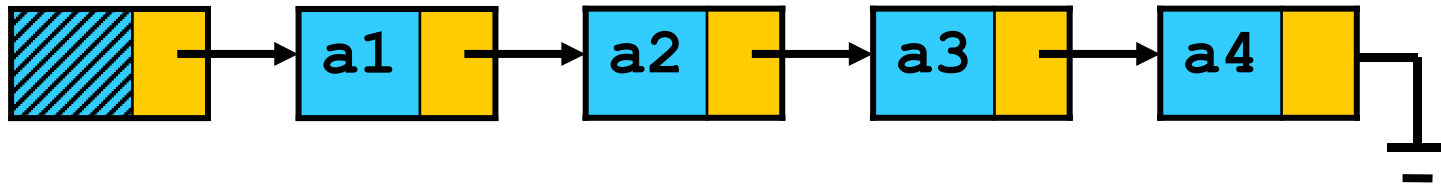
节点



存储结构

```
typedef struct node{  
    ElemType data;  
    struct node *next;  
} LNode ;
```

- 头节点数据域为空



为什么要有一个头结点？

- 因为原来的第一个结点有点儿特殊：
 - 第一个节点的前面没有结点(前驱结点)
 - 而其它节点的前面都有前驱结点
- 这个特殊性导致了链表操作的很多算法都必须分第一个结点和非第一个结点两种情况来讨论
- 所以增加一个“无用”的空结点，这样消除了这种不一致，从而简化了算法

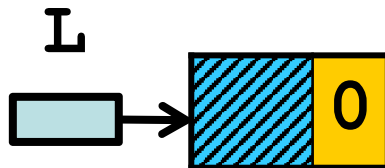
初始化

```
typedef LNode* LkList;
```

```
bool InitLkList(LkList &L);
```

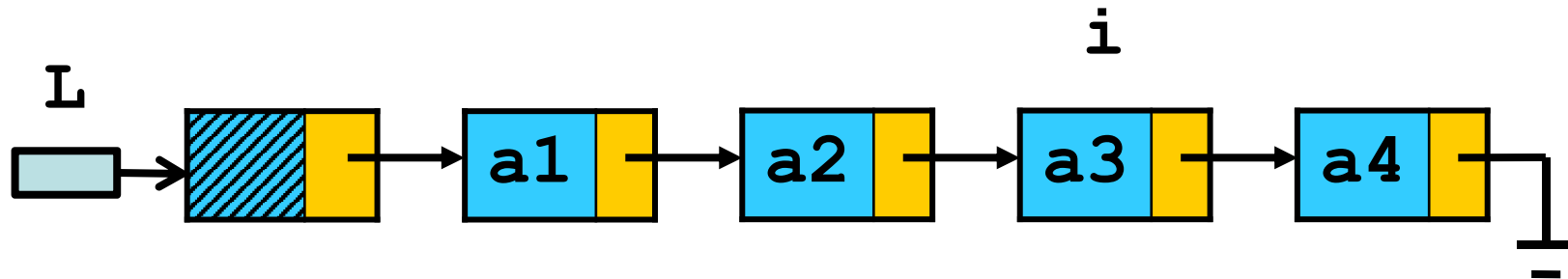
```
bool InitLkList(LNode* &L);
```

```
bool InitLkList(LNode* &L) {  
    L = (LNode *) malloc (sizeof(LNode));  
    if(!L) return false;  
    L->next = 0; return true;  
}
```



存取操作

- 要访问线性表的第*i*个元素，要从表头起沿着指针一个一个元素的查找
- 显然，访问第*i*个节点所需时间由*i*决定
- 所以存取操作复杂度 = $O(n)$

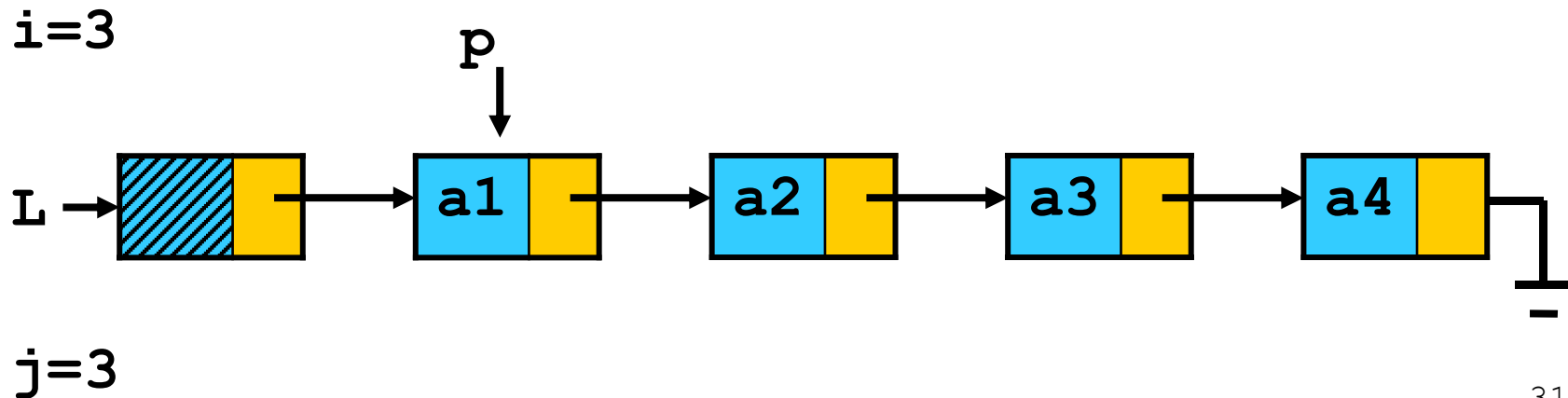


```
bool GetElem_L
(LNode* L, int i, ElemType &e)
{
    LNode* p = L->next;    int j = 1;

    // 循环直到p为空或到了第i个节点
    while (p && j < i) {
        p = p->next;
        ++ j;
    }
    if (!p || j > i)      // 第i个节点不存在
        return false;
    e = p->data;          // copy数据到e中
    return true;
}
```

```
p = L->next;    j = 1;

// 循环直到p为空或到了第i个节点
while (p && j < i) {
    p = p->next;
    ++ j;
}
if (!p || j > i) // 第i个节点不存在
    return false;
```



```
p = L->next;    j = 1;
```

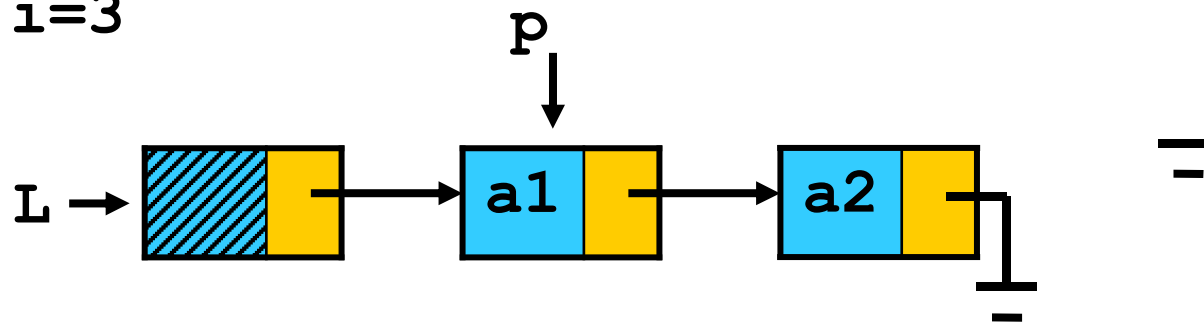
```
// 循环直到p为空或到了第i个节点
```

```
while (p && j < i) {  
    p = p->next;  
    ++ j;  
}
```

```
if (!p || j > i) // 第i个节点不存在  
    return false;
```

p已经走到了尽头，
却还没找到第i个节点，
说明第i个节点不存在

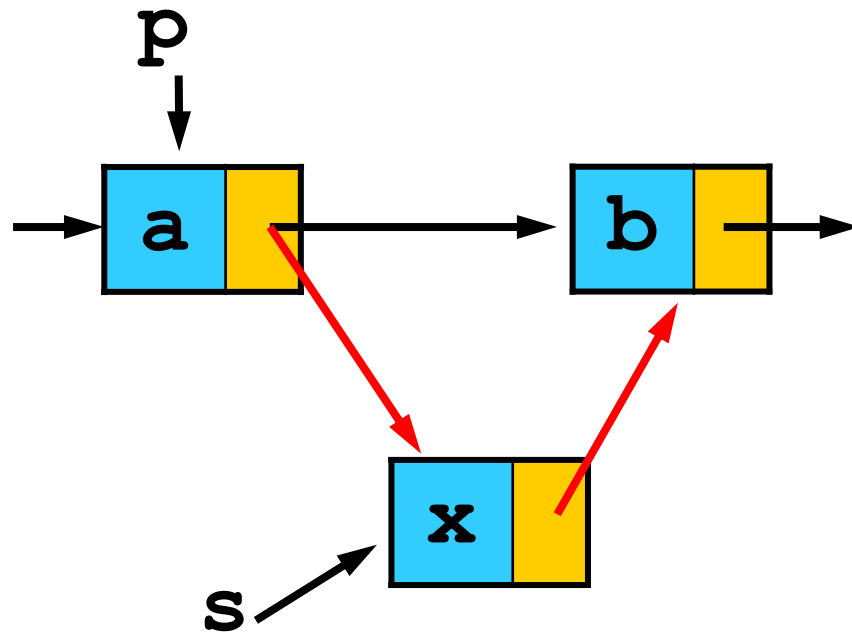
i=3



j=3

插入操作

- 因为各个元素的存储地址任意，所以不需要移动节点，只需修改**next**指针



```
bool ListInsert_Lk
    (LNode* &L, int i, ElemType e)
{
    LNode* p = L;    j = 0;

    // 寻找第i-1个节点
    while (p && j < i-1) {
        p = p->next;
        ++ j;
    }

    // 若第i-1个节点不存在
    if (!p || j > i-1)
        return false;
}
```

不就是前面的
GetElem_L算法么？

```
// 生成一个新节点，并链接到L中
```

```
s = (LNode *) malloc (sizeof(LNode));
```

```
s->data = e;
```

```
s->next = p->next;
```

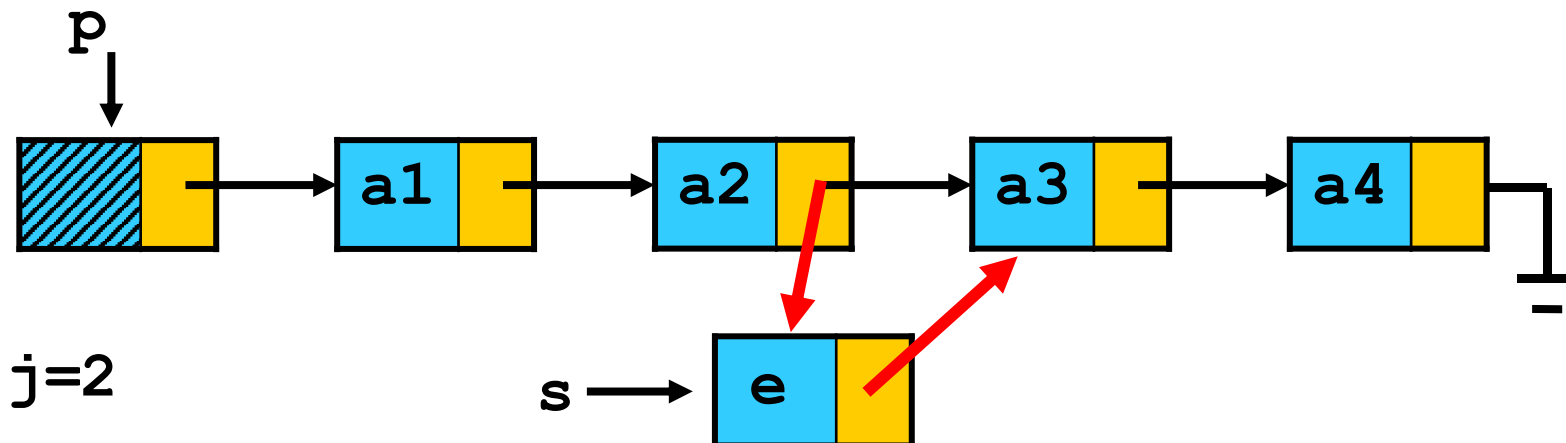
```
p->next = s;
```

```
return true;
```

```
}
```

} 注意：这两条语句的顺序不能颠倒

$i=3$ ，即在第3个节点前面插入一个新的节点

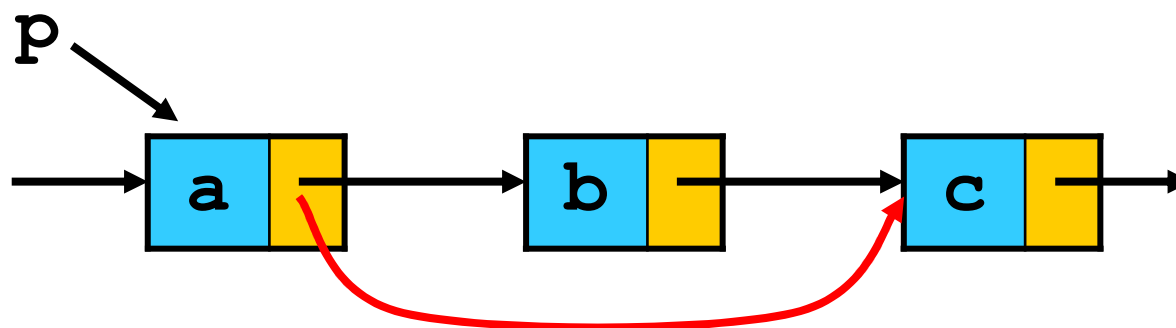


插入操作的时间复杂度

- 插入这个过程所需时间为常数
- 但是找到插入位置的复杂度 = $O(n)$
- 所以插入操作的复杂度 = $O(n)$

删除

- 和插入类似，只需移动几个指针
- 但是也必须先找到待删除的节点



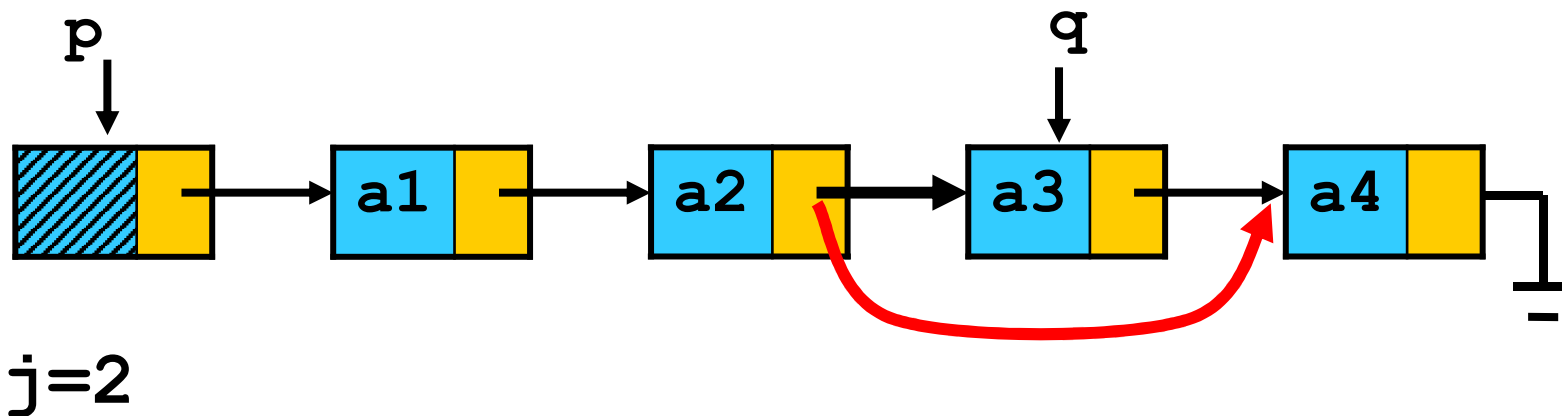
```
bool ListDelete_Lk
(LNode* &L, int i, ElemType& e)
{
    LNode* p = L;    j = 0;

    // 让p指向第i-1个节点
    while (p && j < i-1) {
        p = p->next;
        ++ j;
    }

    // 若第i个节点不存在
    if (! (p->next) || j > i-1)
        return false;
}
```

```
LNode* q = p->next; // q指向待删除节点
p->next = q->next; // 使q脱离链表
e = q->data; // e得到q的数据
free(q); // 释放q的空间
return true;
}
```

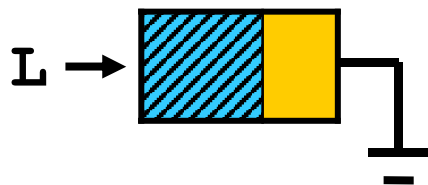
$i=3$, 即删除第3个节点



删除操作的时间复杂度是多少？

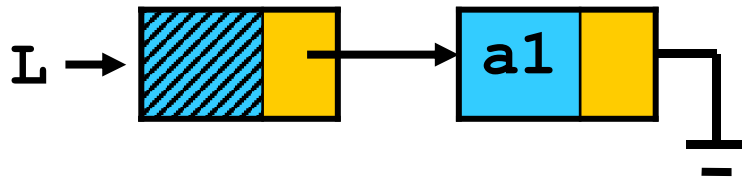
创建

- 即从空表开始不停的插入节点
 - 不过如果每次都是从表尾插入的话，需要先搜索到表尾的位置
 - 所以从表头插入更快



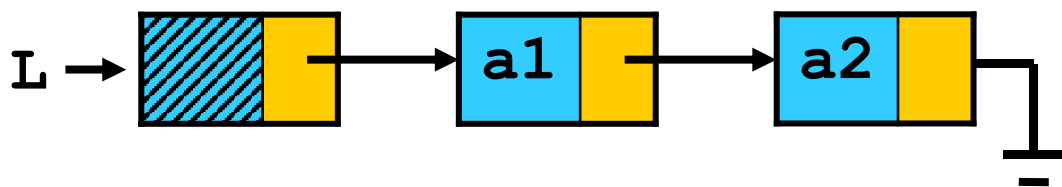
创建

- 即从空表开始不停的插入节点
 - 不过如果每次都是从表尾插入的话，需要先搜索到表尾的位置
 - 所以从表头插入更快



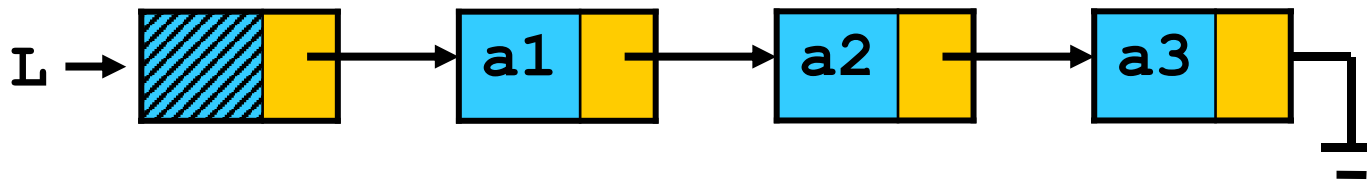
创建

- 即从空表开始不停的插入节点
 - 不过如果每次都是从表尾插入的话，需要先搜索到表尾的位置
 - 所以从表头插入更快



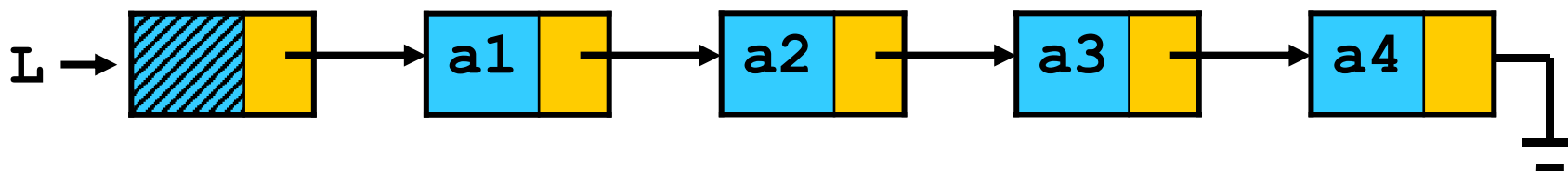
创建

- 即从空表开始不停的插入节点
 - 不过如果每次都是从表尾插入的话，需要先搜索到表尾的位置
 - 所以从表头插入更快



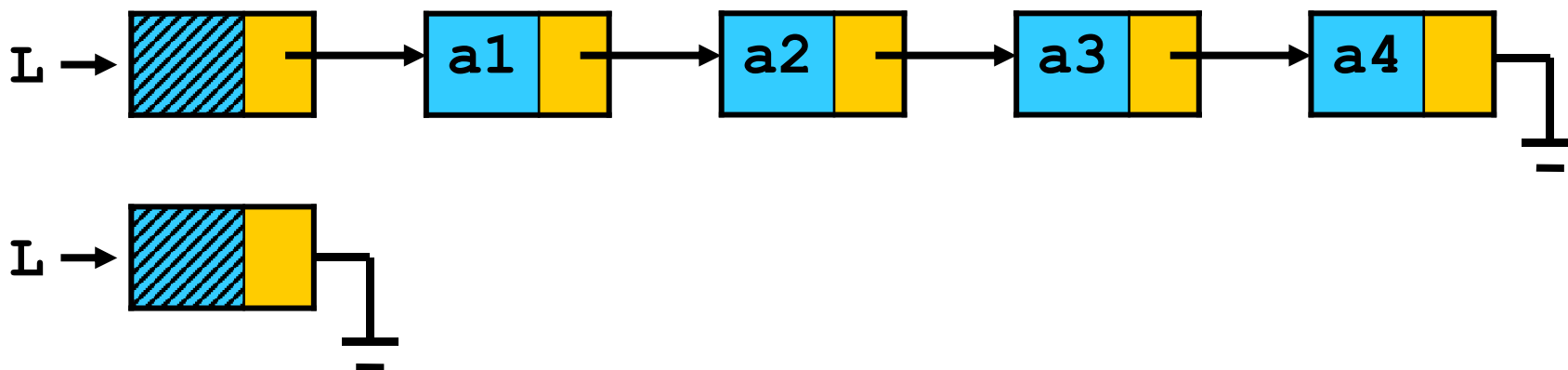
创建

- 即从空表开始不停的插入节点
 - 不过如果每次都是从表尾插入的话，需要先搜索到表尾的位置
 - 所以从表头插入更快



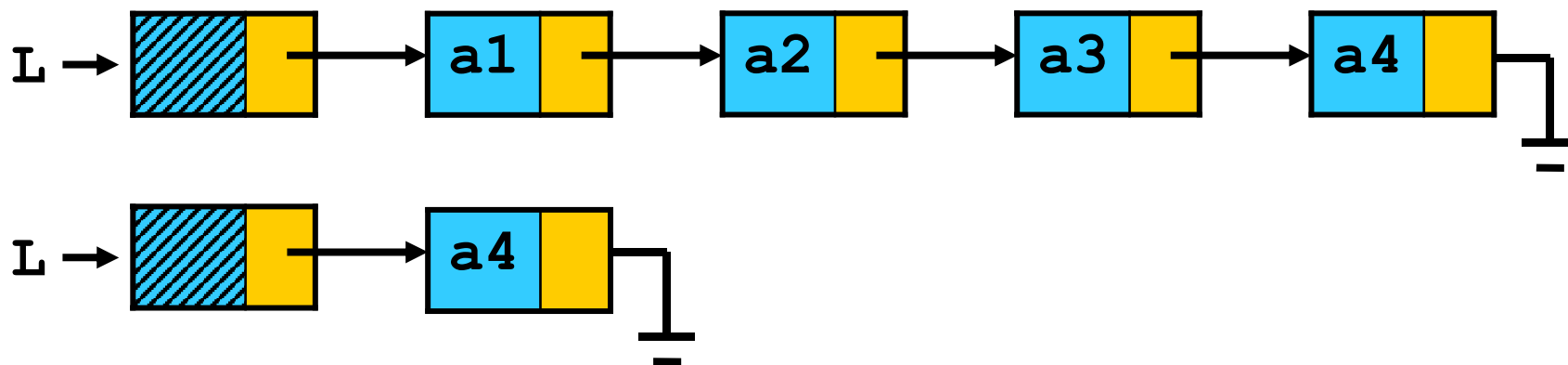
创建

- 即从空表开始不停的插入节点
 - 不过如果每次都是从表尾插入的话，需要先搜索到表尾的位置
 - 所以从表头插入更快



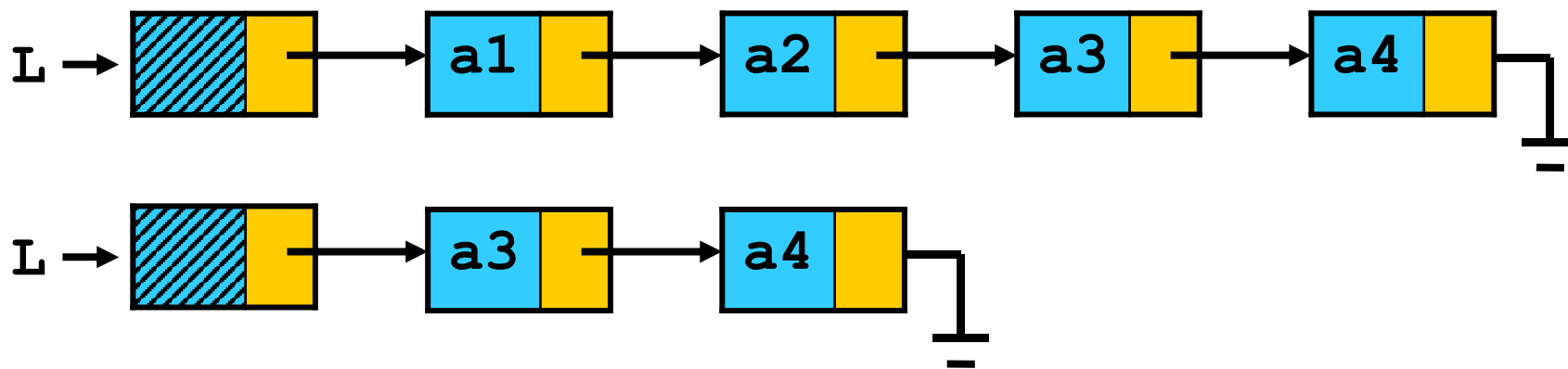
创建

- 即从空表开始不停的插入节点
 - 不过如果每次都是从表尾插入的话，需要先搜索到表尾的位置
 - 所以从表头插入更快



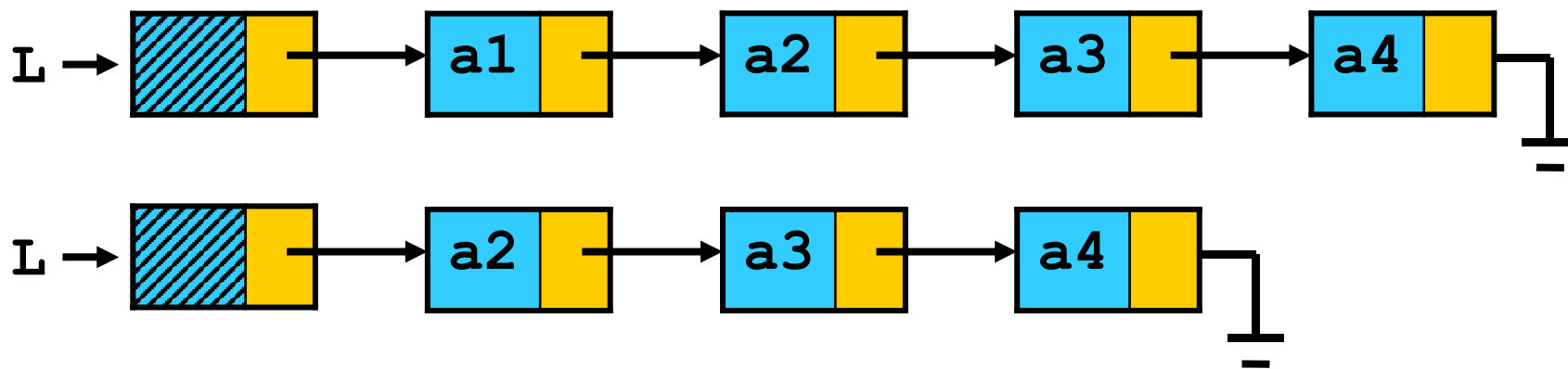
创建

- 即从空表开始不停的插入节点
 - 不过如果每次都是从表尾插入的话，需要先搜索到表尾的位置
 - 所以从表头插入更快



创建

- 即从空表开始不停的插入节点
 - 不过如果每次都是从表尾插入的话，需要先搜索到表尾的位置
 - 所以从表头插入更快

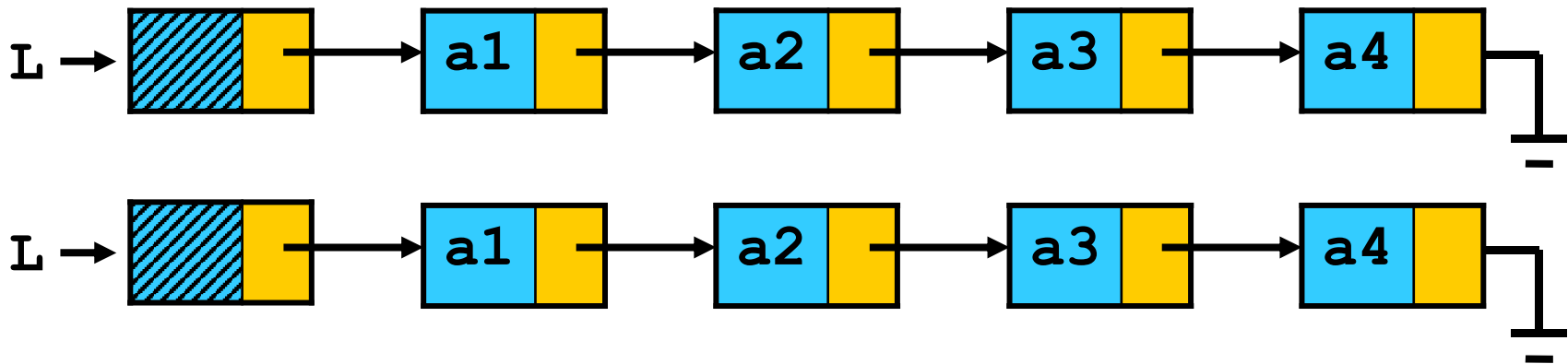


创建

- 即从空表开始不停的插入节点

- 不过如果每次都是从表尾插入的话，需要先搜索到表尾的位置
- 所以从表头插入更快

```
bool push_back(Lnode *L, elemType e);  
bool insert_front(Lnode *L, elemType e);
```



静态链表

- 动态链表

- 链表中的每个节点都是动态分配
 - 使用`malloc`函数
- 节点之间使用指针链接

- 静态链表

- 静态分配空间
 - 使用数组来存储
- 但是像动态链表一样，节点地址不一定连续
- 使用下标作为“指针”

存储结构

```
#define SIZE 500
typedef struct{
    ElemType data;
    int next;
} sNode;
typedef sNode sLkList[SIZE];
```

```
sLkList L;
```

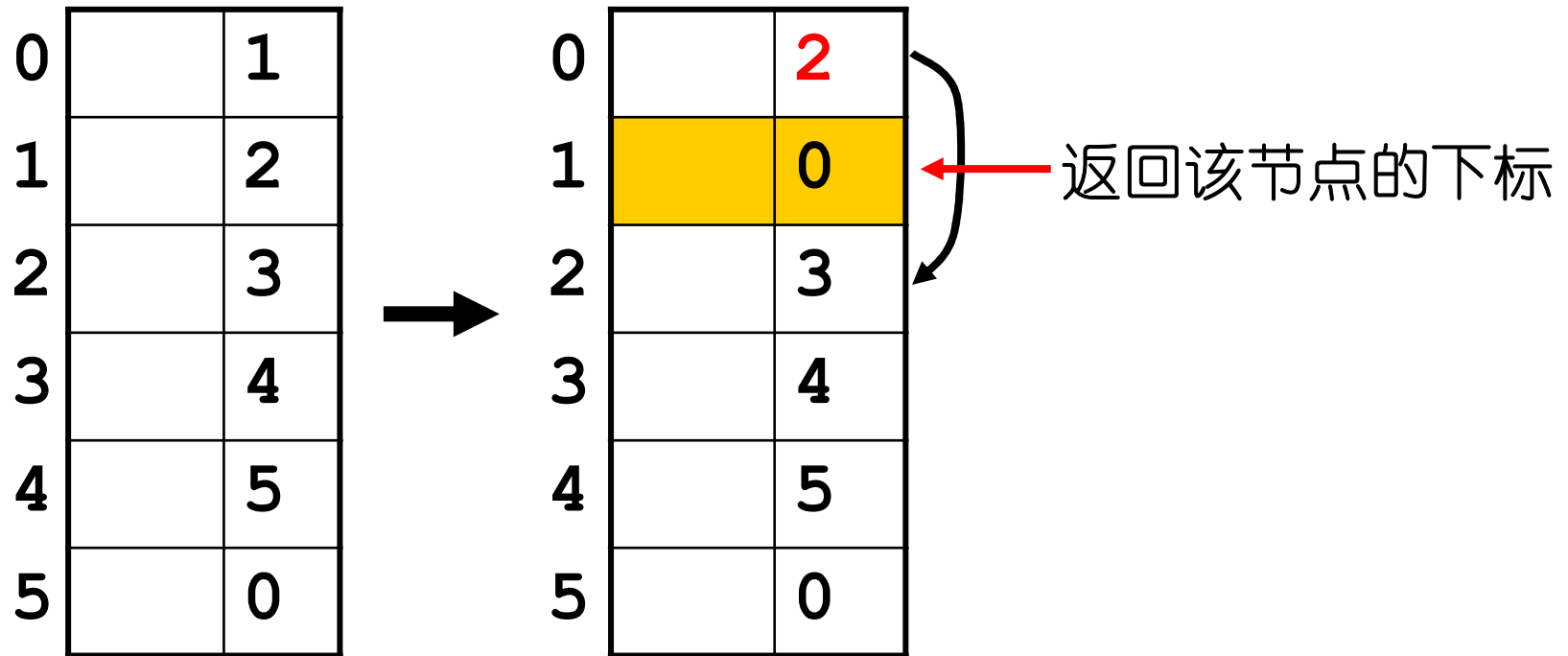
L
数据 “指针”

0		1
1	Zhao	4
2	Wu	0
3	Sun	6
4	Qian	3
5	Zhou	2
6	Li	5
7		

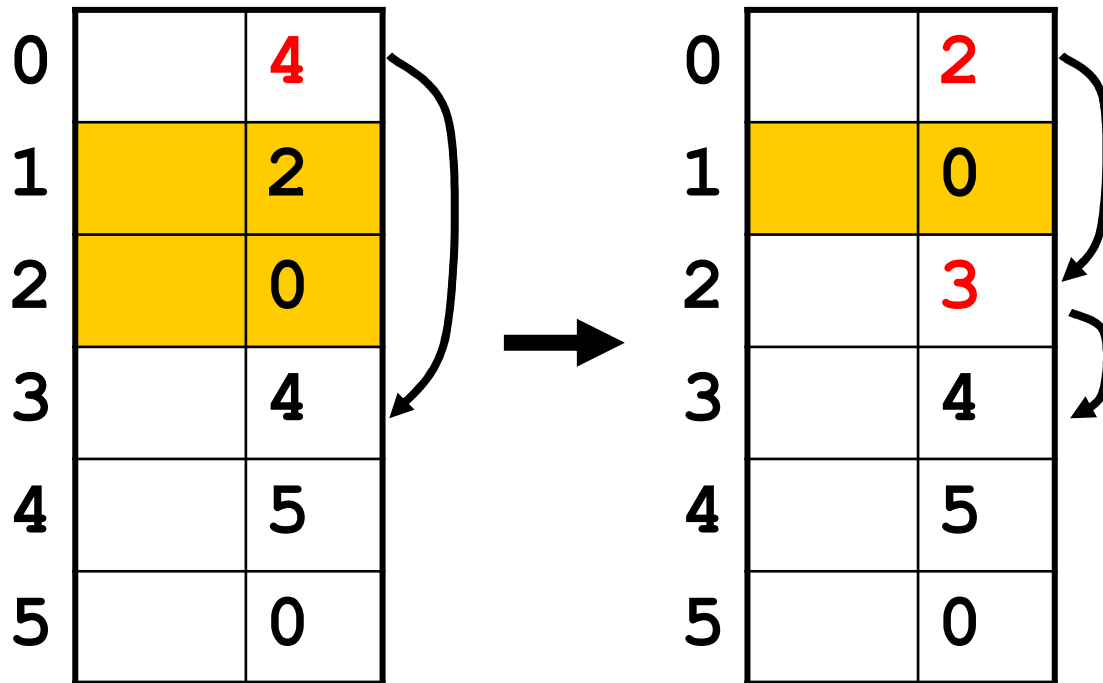
优缺点

- 优点：
 - 可以应用于没有指针的语言（比如**Matlab**）
- 缺点：
 - 空间是静态分配的，缺乏灵活性
- 因此引入“备用链表”模拟动态分配
 - 当需要一个节点的空间时，从备用链表中分配
 - 当删除一个节点时，把该节点链入备用链表

从备用链表中分配一个节点

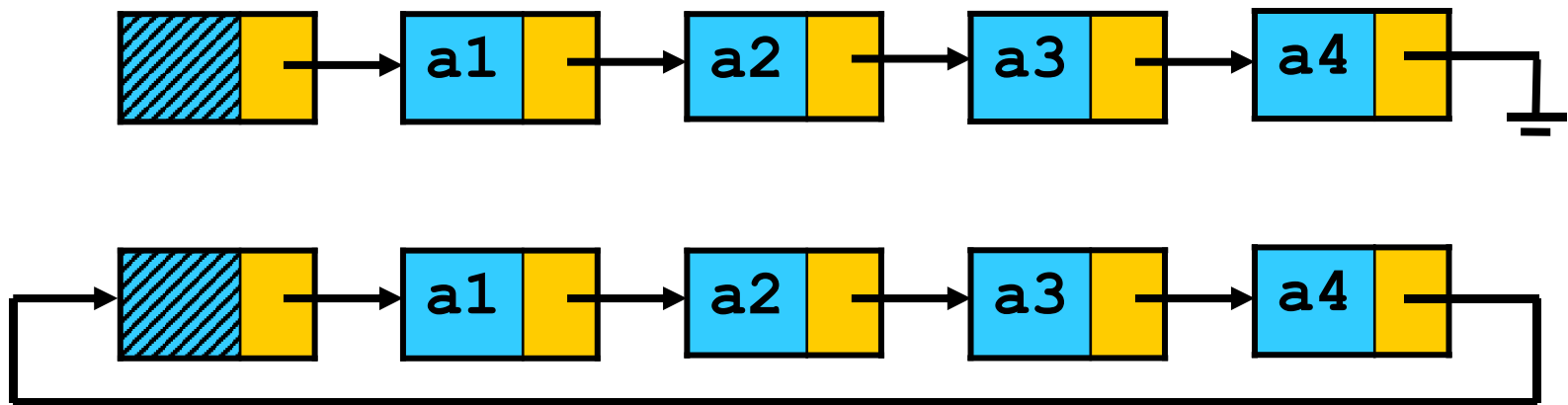


释放一个节点到备用链表



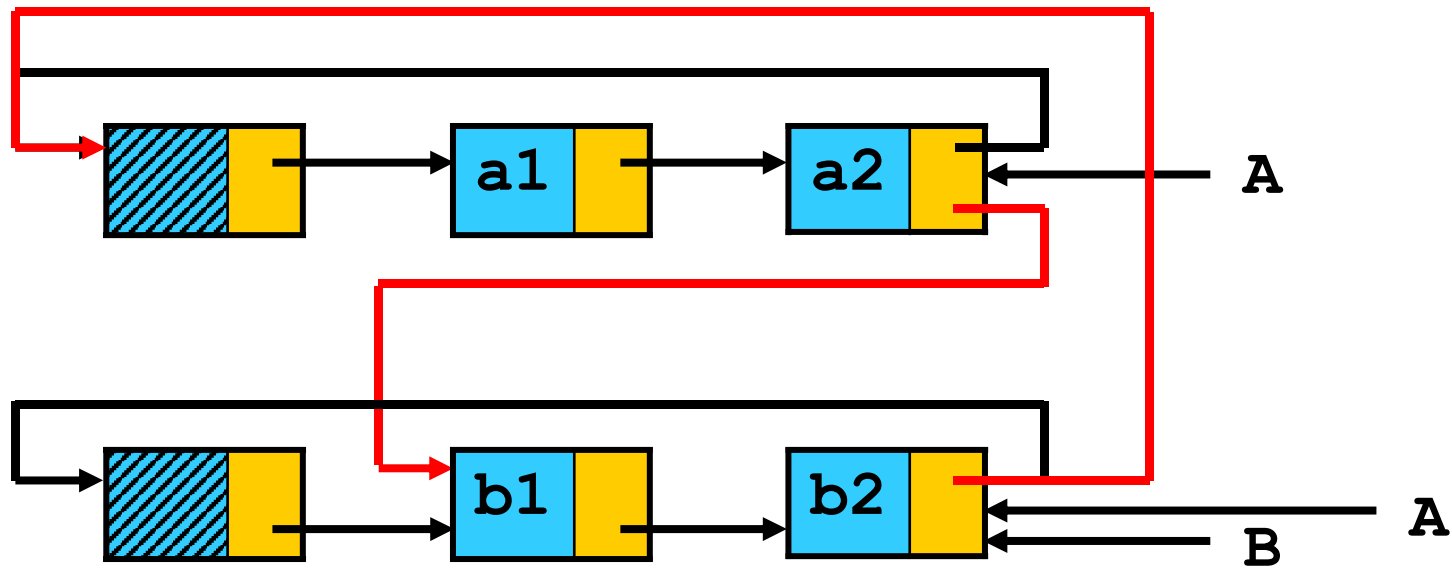
循环链表

- 非循环链表
 - 尾指针为空，浪费
- 循环链表
 - 尾指针指向表头



循环链表

- 循环链表通常设尾指针
 - 要找头指针？尾指针指向的就是
 - 便于链表合并



双向链表

- 单向链表

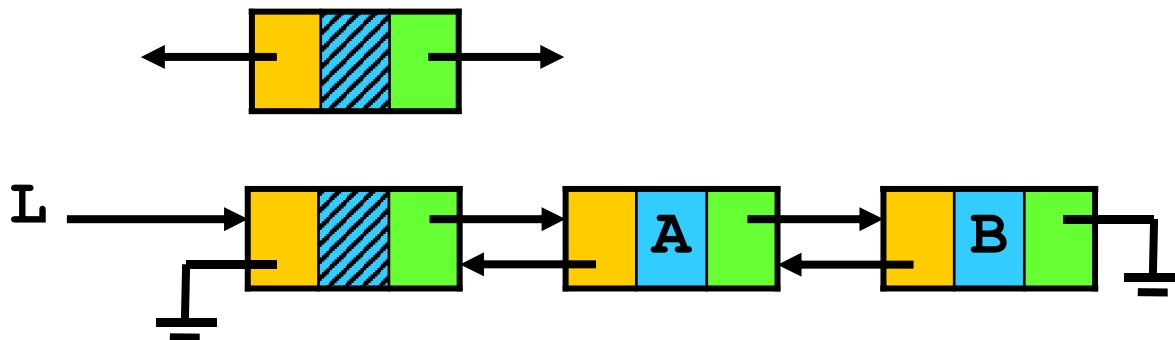
- 只知道后继节点，不知前趋节点
- **NextElem**操作复杂度为 $O(1)$
- **PriorElem**操作复杂度为 $O(n)$
 - 必须从头开始查找

- 双向链表

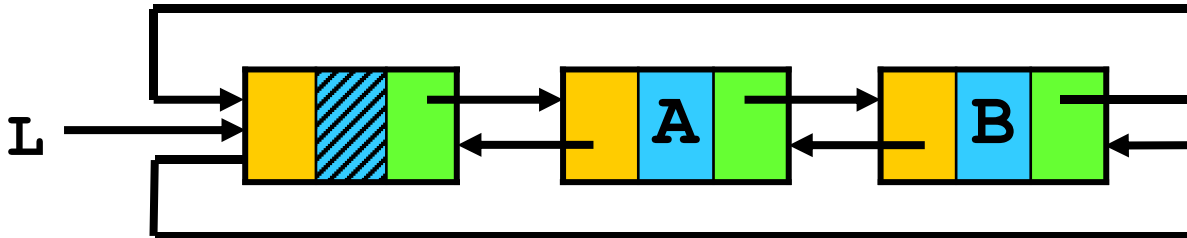
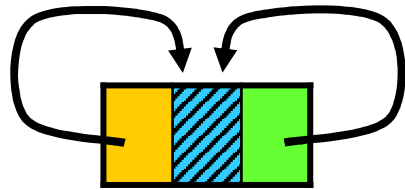
- 增加一个前趋指针

存储结构

```
typedef struct Du1Node{  
    ElemType      data;  
    struct Du1Node *prior;  
    struct Du1Node *next;  
}Du1Node;
```



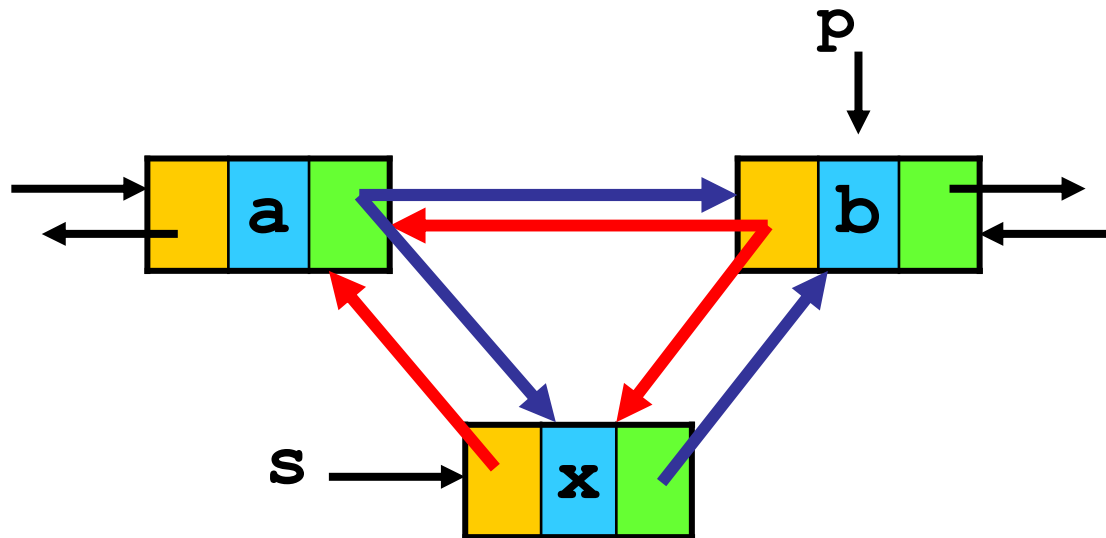
循环双向链表



双向链表：插入操作

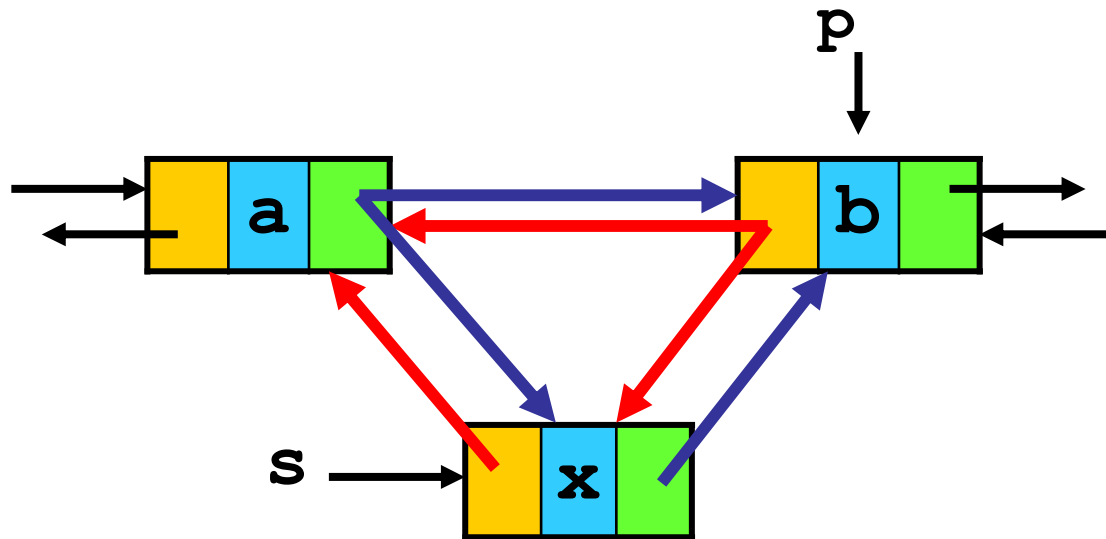
- 在第*i*个节点

之前，插入节点s



双向链表：插入操作

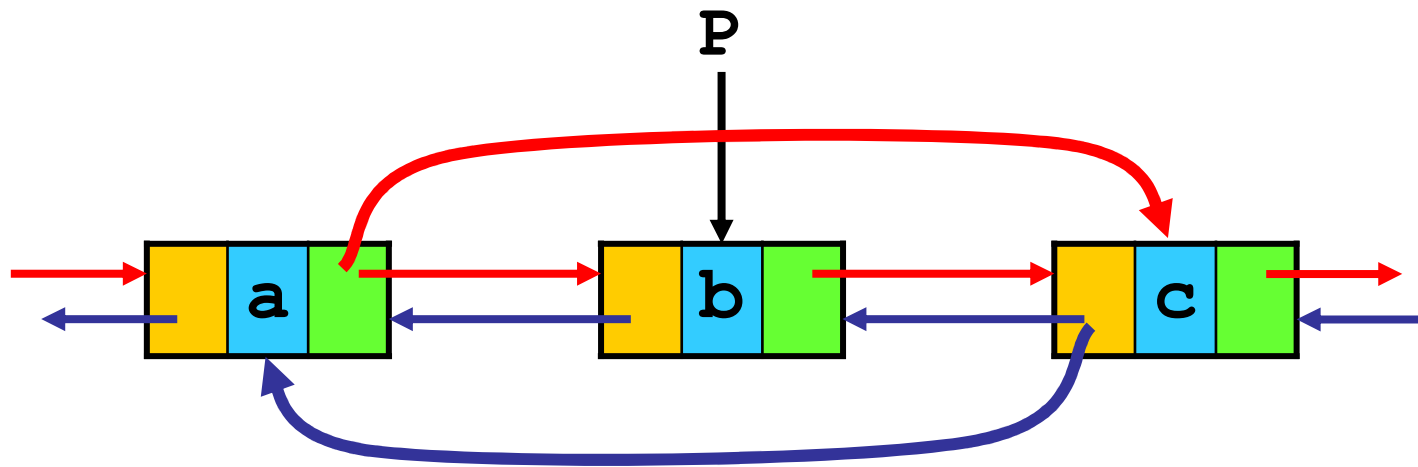
```
s->prior = p->prior;  
p->prior->next = s;  
s->next = p;  
p->prior = s;
```



双向链表：删除操作

```
p->prior->next = p->next;  
p->next->prior = p->prior;  
free (p);
```

} 绕过p



一元稀疏多项式

- 一元多项式的表示:

$P_n(x) = p_0 + p_1x + p_2x^2 + \dots + p_nx^n$ 可表示为系数的线性表:
 $P = (p_0, p_1, \dots, p_n)$

一元稀疏多项式如 $1 + 3x^{10000} - 2x^{20000}$ 可表示为(系数项, 指数项)的线性表:

$((1,0), (3,10000), (-2,20000))$

$((p_1, e_1), (p_2, e_2), \dots, (p_m, e_m))$

一元稀疏多项式：类型定义

ADT Polynomial {

数据元素是(系数、指数)的线性表

基本操作：

InitPolyn (&P); //初始化一个空的一元多项式P。

DestroyPolyn (&P); //销毁一元多项式P。

PolynLength(P); //多项式项数

Value (P, x); //变量为x时多项式的值

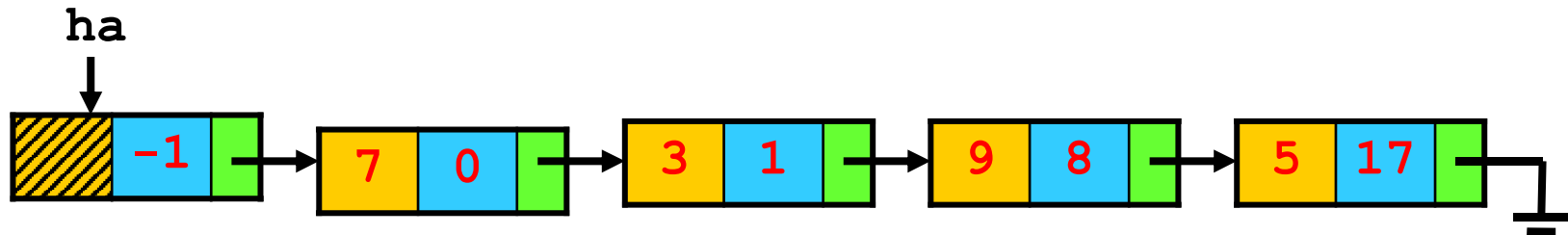
Ceof(P , int k); //指数为k的项系数

```
int MaxExp(P); //最大指数
InsertTerm( coef, expn); //插入一项
DeleteItem(&P,int k);

AddPolyn ( &Pa, &Pb ); //多项式加法
SubtractPolyn ( &Pa, &Pb ); //多项式减法
MultiplyPolyn(&Pa, &Pb); //多项式乘法
ScalPolyn (&P, coef, expn); //数乘一项
CopyPolyn(&Pa,Pb); //复制多项式
} // ADT Polynomial
```

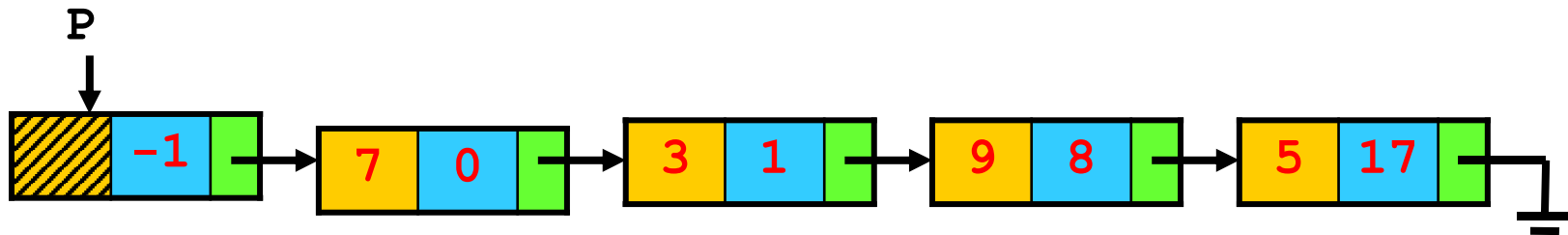
一元稀疏多项式：链式实现

- 用带头结点的链式表表示多项式，每个结点对应多项式的一项。



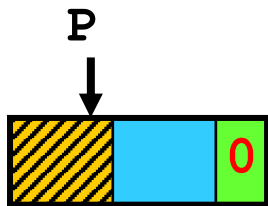
一元稀疏多项式：链式实现

- ```
typedef struct {
 float coef;
 int expn;
} ElemType; // 数据元素类型
```
- ```
typedef struct polyNode{  
    ElemType      data;  
    struct polyNode* next;  
} PolyNode, *Poly;
```



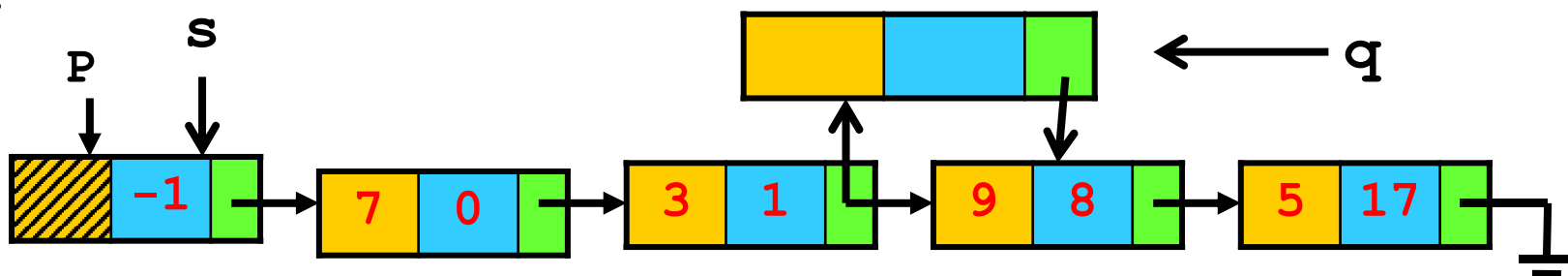
初始化

```
bool InitPoly(Poly &P) {  
    P = new PolyNode();  
    if(!P) return false;  
    P->next = 0;  
    return true;  
}
```

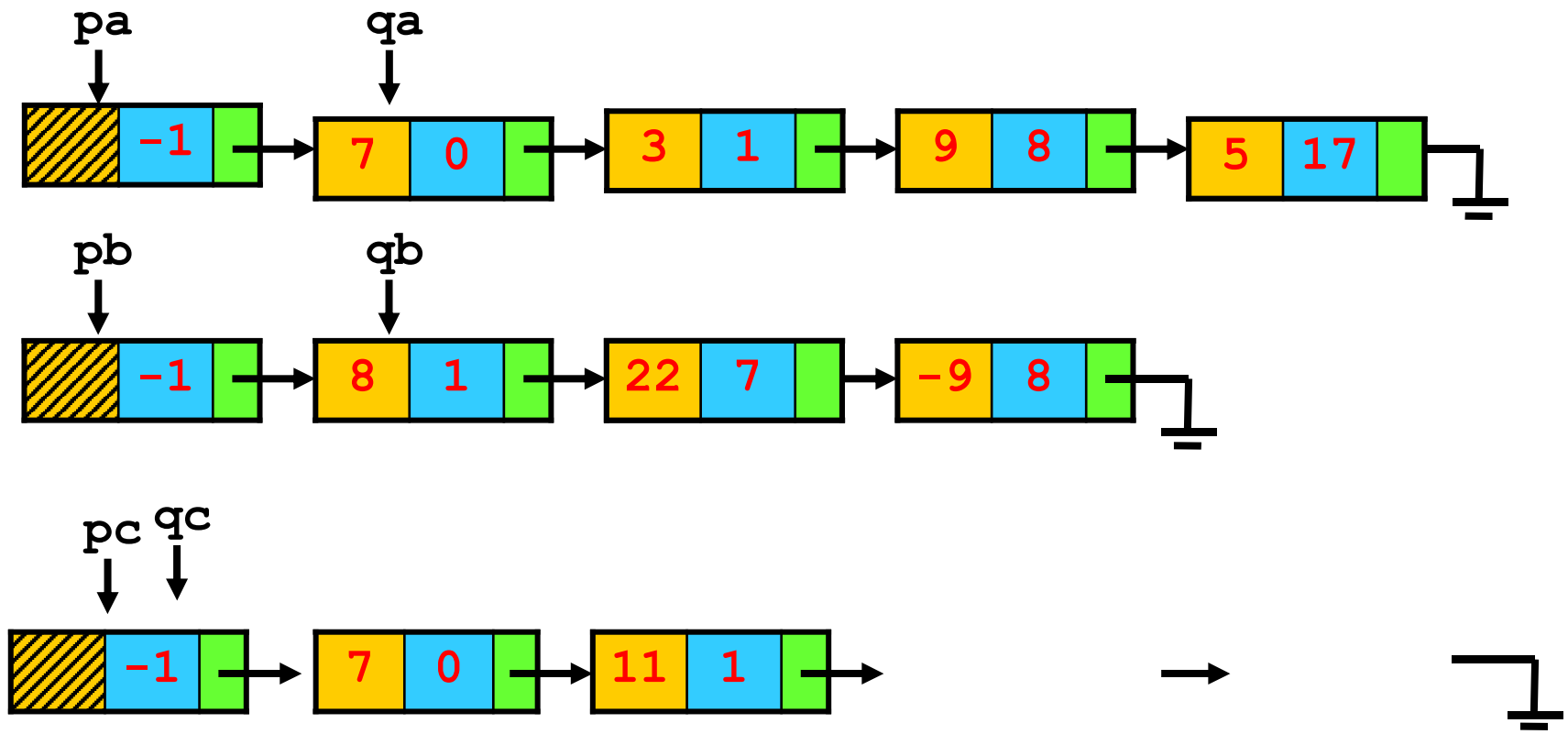


插入一项

```
bool InsertPoly(Poly &P, float2.5 c, int4 k) {  
    PolyNode *q = new PolyNode();  
    q->data.coef = c; q->data.expn = k;  
  
    PolyNode *s = P;  
    while (s->next && s->next->data.expn < k)  
        s = s->next;  
  
    q->next = s->next; s->next = q;  
}
```



加法



加法 : $Pc = Pa + Pb$

```
void AddPolyn (Poly Pa, Poly Pb,
               Poly &Pc ) {
    PolyNode *qa,*qb,*qc;
    InitPolyn(Pc);
    qa = Pa->next;qb = Pb->next;qc = Pc;
    while( qa && qb){//都不空时
    }
    while( qa) { //Pa还有剩余
    }
    while( qb) { //Pb还有剩余
    }
}
```


加法? Pa += Pb

```
void AddPolyn (Poly &Pa, Poly Pb) {  
  
}
```

乘法

```
void MultiplyPolyn (Poly Pa,  
    Poly Pb, Poly &Pc ) {  
    Poly TP;  
    PolyNode *qb;  
    InitPolyn (Pc) ;  
    if (! Pa->next || !Pb->next) return ;  
    CopyPolyn (Pc, Pa) ;
```

```
qb = qb->next;
while (qb) {
    CopyPolyn (TP, Pa) ;
    ScalPolyn (TP, qb->coef, qb->expn) ;
    AddPolyn (Pc, TP) ;
    qb = qb->next;
}
}
```

本章小结

- 线性表的类型定义
 - 理解线性表的概念
- 顺序实现
 - 用静态、连续的空间来存储数据
 - 存取操作方便
 - 但是插入、删除操作需要大量移动数据

本章小结

- 链式实现

- 即用链表实现
- 动态分配空间
- 结点的地址之间不一定连续
- 存取操作需要从头结点开始一个一个的搜索，复杂度为 $O(n)$
- 插入、删除操作因为先要找到待处理的节点，所以复杂度仍然是 $O(n)$

本章小结

- 静态链表
 - 静态存储，动态使用
- 循环链表
 - 尾指针指向指向头节点，便于合并操作
- 双向链表
 - 增加前驱指针
- 算法重点
 - 插入、删除操作，注意语句的顺序

本章小结

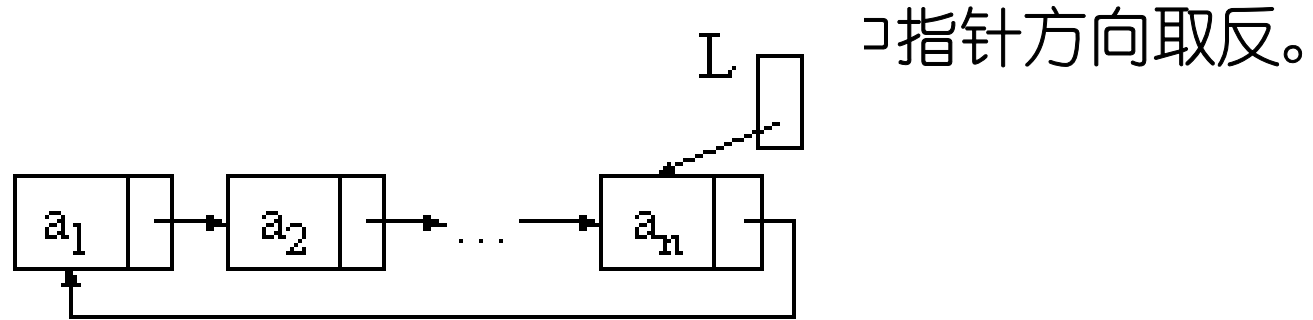
- 一元稀疏多项式的表示与实现
 - 一种特殊的线形表

作业

1. 双向链表中有两个指针域，**llink**和**rlink**，分别指回前驱及后继，设**p**指向链表中的一个结点，**q**指向一待插入结点，现要求在**p**前插入**q**，则正确的插入为（ ）。

- A. `p->llink=q; q->rlink=p;`
`p->llink->rlink=q; q->llink=p->llink;`
- B. `q->llink=p->llink; p->llink->rlink=q;`
`q->rlink=p; p->llink=q->rlink;`
- C. `q->rlink=p; p->rlink=q;`
`p->llink->rlink=q; q->rlink=p;`
- D. `p->llink->rlink=q; q->rlink=p;`
`q->llink=p->llink; p->llink=q;`

2. 设有行



3. 对2个有序的线性表进行合并

- 1) 编写对2个有序的顺序表进行合并的程序并测试
- 2) 编写对2个有序的链式表进行合并的程序并测试

4. 编写程序判断：用单链表存储的一个整数序列是否是等差数列