数据结构

2 线性表

http://hwdong.com 董洪伟

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

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，$\mathbf{a}_i$是第**i**个元素，**i**称为数据元素 $\mathbf{a}_i$在线性表中的位序

线性表的抽象数据类型

ADT List{

数据对象:$D=\{a_i | a_i \in Elemset,\ i=1,2,...,n,\ n\geq 0\}$

数据关系:$R1=\{<a_{i-1},a_i>,\ a_i \in D,\ i=2,...,n\}$

基本操作:

InitList(&L)

操作结果：构造一个空的线性表L。

DestroyList(&L)

初始条件：线性表L已存在。

操作结果：销毁线性表L。

ClearList(&L)

初始条件：线性表L已存在。

操作结果：将L重置为空表。

线性表的抽象数据类型

ListEmpty(L)

初始条件：线性表L已存在。

操作结果：若L为空表，则返回TRUE；否则返回 FALSE。

ListLength(L)

初始条件：线性表L已存在。

操作结果：返回L中数据元素的个数。

GetElem(L, i, &e)

初始条件：线性表L已存在，1<=i<=Listlength(L)。

操作结果：把L中第i个元素的值赋给e。

线性表的抽象数据类型

LocateElem(L, e, compare())

初始条件：线性表**L**已存在，

compare()是数据元素判定函数。

操作结果：返回**L**中第**1**个与**e**满足关系

compare()的数据元素的位序。

若这样的元素不存在，则返回**0**。

PriorElem(L, cur_e, &pre_e)

初始条件：线性表**L**已存在。

操作结果：若**cur_e**是**L**的数据元素，且不是第一个，则把它前一个元素的值赋给**pre_e**，否则操作失败，**pre_e**无意义。

线性表的抽象数据类型

NextElem(L, cur_e, &next_e)

初始条件：线性表L已存在。

操作结果：若cur_e是L的数据元素，且不是最后一个，则把它后一个元素的值赋给next_e，否则操作失败，next_e无意义。

ListInsert(&L, i, e)

初始条件：线性表L已存在，1<=i<=ListLength(L)+1。

操作结果：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。

线性表的抽象数据类型

ListDelete(&L, i, &e)

初始条件：线性表**L**已存在，

1<=i<=ListLength(L)。

操作结果：删除**L**的第**i**个数据元素，并把其值赋给**e**，**L**的长度减**1**。

ListTraverse(L, visit())

初始条件：线性表**L**已存在。

操作结果：依次对**L**的每个数据元素调用函数**visit()**。一旦**visit()**失败，则操作失败。

}ADT List

一些定义

```
typedef int Status;
#define OK 0
#define ERROR 1
#define OVERVIEW 2
#define NoMemory 3
```

线性表的顺序表示和实现

- 顺序表示
  - 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*) malloc (n*sizeof(ElemType));

      (ElemType*) realloc (n*sizeof(ElemType));

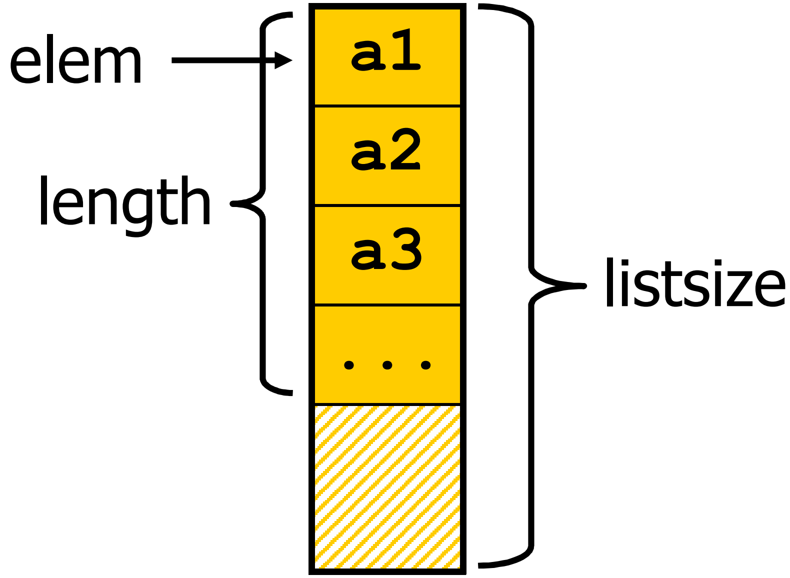
线性表的顺序表示和实现

- 动态分配顺序存储结构

```
#define LIS
#define LIS
typedef struc
    ElemType *elem;
    int length;
    int listsize;
}SqList;
```

可以把ElemType定义为任何类型：

```
typedef int ElemType;
```



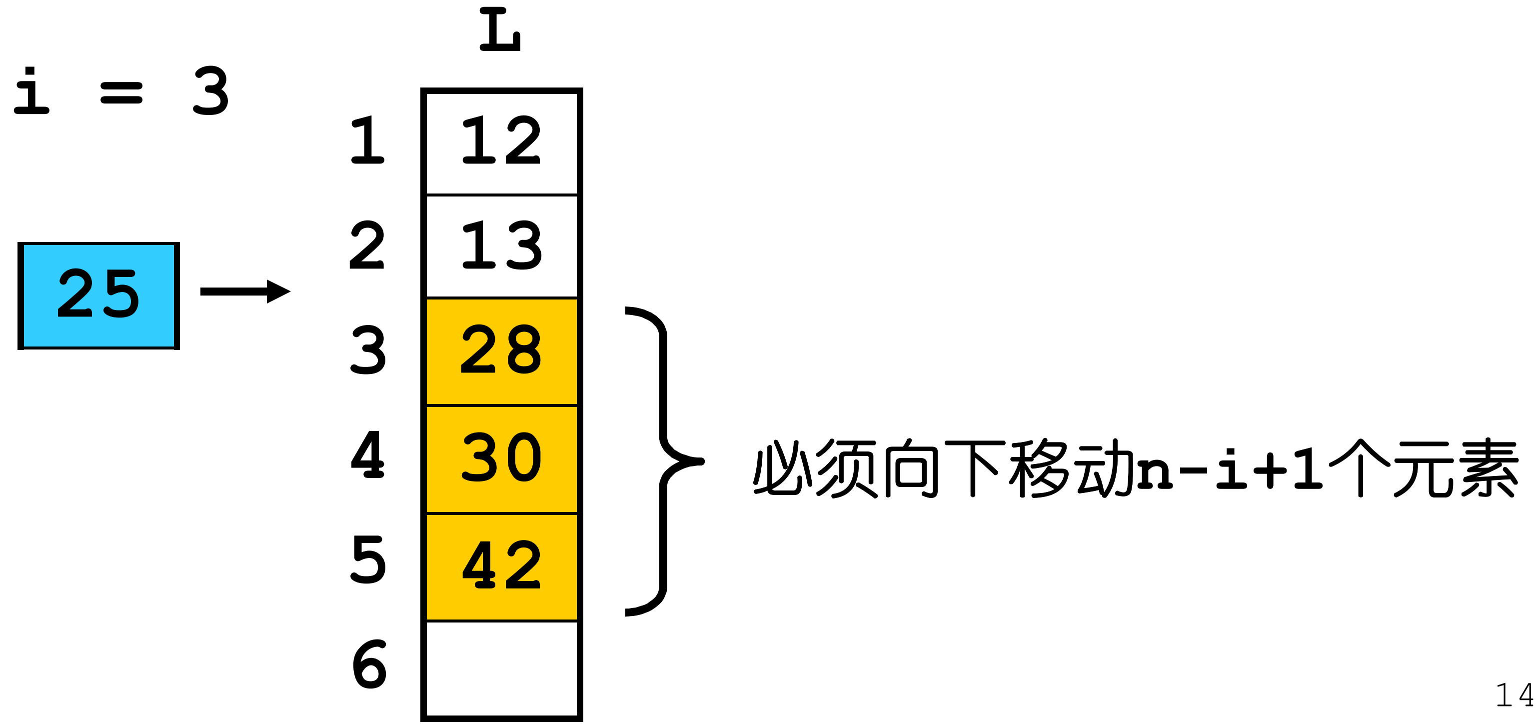
线性表的顺序表示和实现

- 初始化

```
Status InitList_Sq(SqList &L){
     L.elem = (ElemType*)malloc
               (LIST_INIT_SIZE*
                sizeof(ElemType));
     if(!L.elem)  return ERROR;
     L.length = 0;
     L.listsize = LIST_INIT_SIZE;
     return OK;
}
```

线性表的顺序表示和实现

- 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**

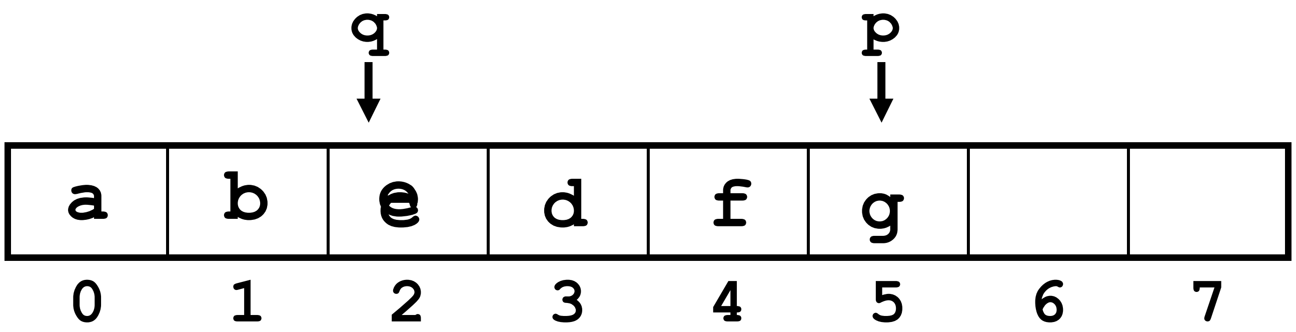


```
Status ListInsert_Sq(
         SqList &L, int i, ElemType e)
{
     // 判断i是否合法
     if(i < 1 || i > L.length + 1)
          return ERROR;
     // 若线性表空间不足，再分配一些空间
     if(L.length >= L.listsize)
     {
          newbase = (ElemType *) realloc
          (    L.elem,
               (L.listsize+LISTINCREMENT)*
                    sizeof(ElemType)
          );
          if(!newbase)   exit(OVERFLOW);
          free(L.elem); L.elem = newbase;
          L.listsize += LISTINCREMENT;
     }
```

```
    // q指向插入的位置
    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
    for(p = &(L.elem[L.length - 1]);
                      p >= q; --p)
        *(p+1) = *p;

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

```
    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.length-1]); p>=q; --p)
        *(p+1) = *p;

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

i=3，即插入在第三个元素之前



线性表的顺序表示和实现

- 插入操作的算法复杂度
  - 很显然，插入操作的复杂度由需要移动的元素的个数决定
  - 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**

线性表的顺序表示和实现

- 最差情况
  - T(n) = O(n)
- 平均情况呢？
  - 一共有1,2,...,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

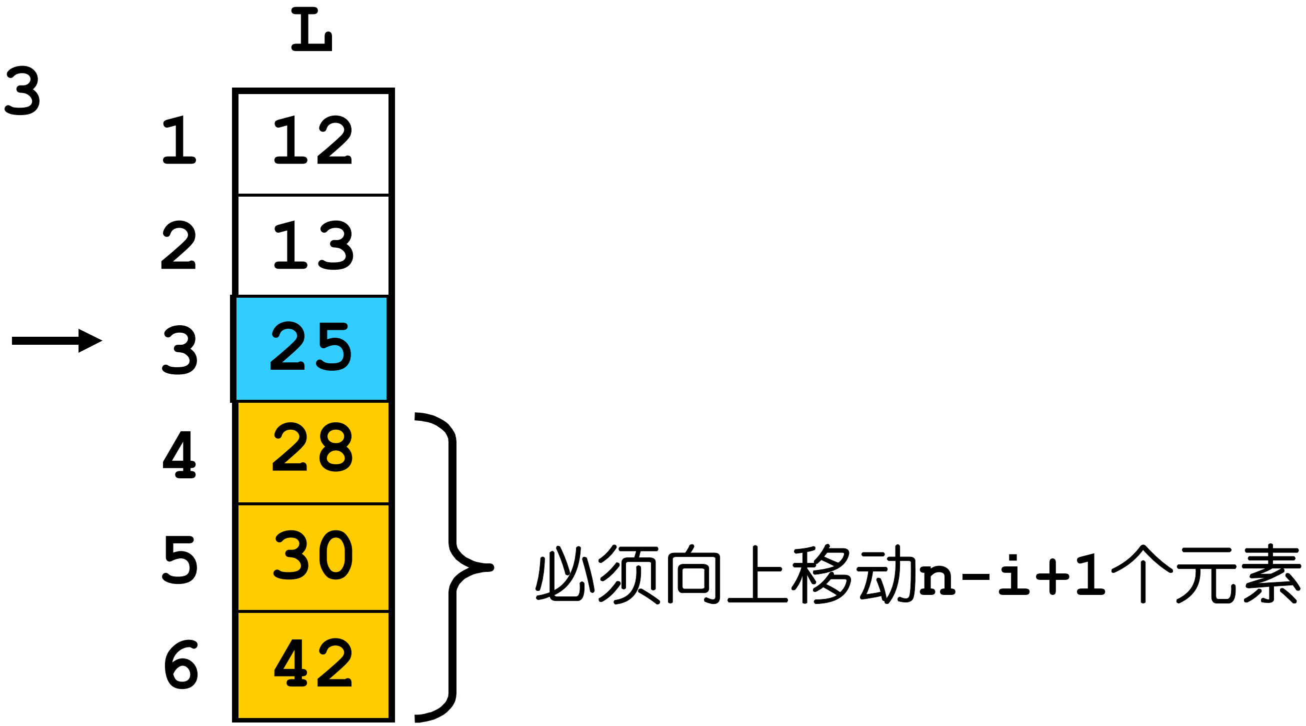
$$= \frac{1}{n+1}\sum_{i=1}^{n+1}(n+1-i) = \frac{1}{n+1} \times \frac{n(n+1)}{2} = \frac{n}{2}$$

  - 所以平均复杂度 = O(n)

线性表的顺序表示和实现

- 删除
  - ListDelete(&L, i, &e)
  - 删除第i个元素，其值赋给e

i = 3



```
Status ListDelete_Sq
          (SqList &L, int i, ElemType &e)
{
     if((i < 1) || (i > L.length))
          return ERROR;
     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.length - 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.length;        // 表长减1
     return OK;
}
```

线性表的顺序表示和实现

- 思考：
  - 删除操作的时间复杂度是多少？

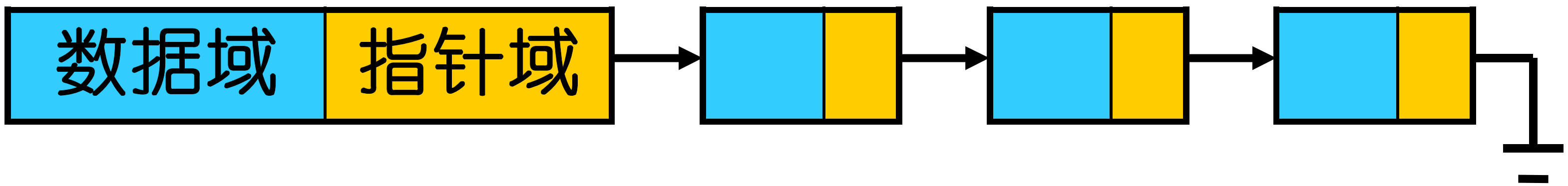
线性表的顺序表示和实现

- 线性表的顺序表示和实现
  - 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

- 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

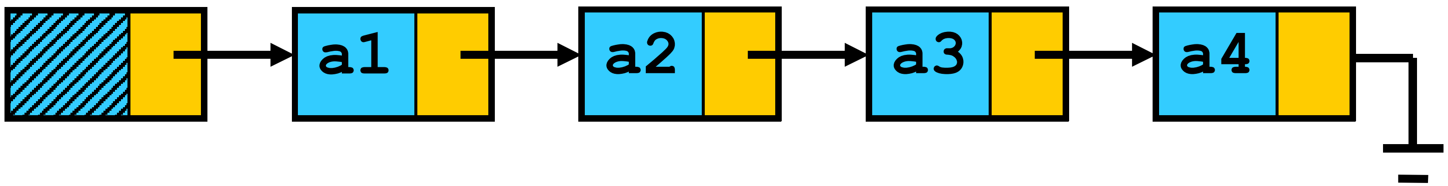


线性表的链式表示和实现

- 存储结构

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

- 头节点数据域为空

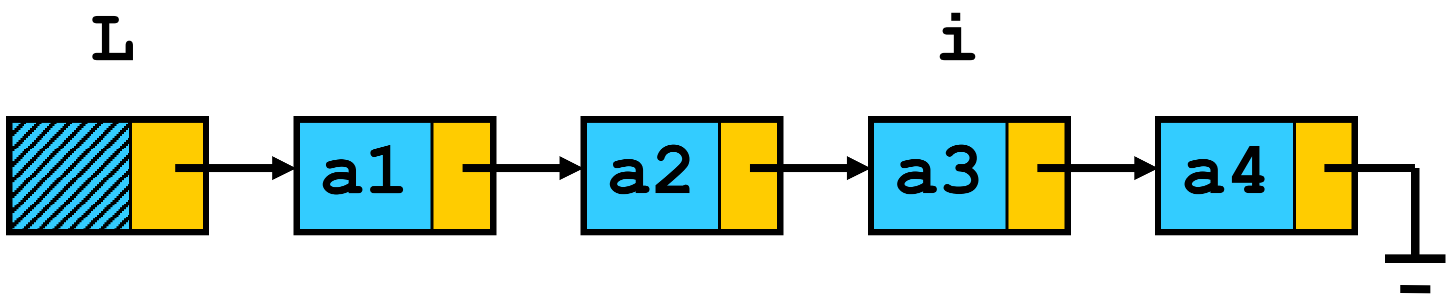


线性表的链式表示和实现

- 为什么要有一个没有数据的头结点？
  - 因为原来的第一个节点有点儿特殊：
    - 第一个节点的前面没有节点(前驱节点)
    - 而其它节点的前面都有前驱节点
  - 这个特殊性导致了链表操作的很多算法都必须分第一个节点和非第一个节点两种情况来讨论
  - 所以增加一个"无用"的空节点，这样消除了这种不一致，从而简化了算法

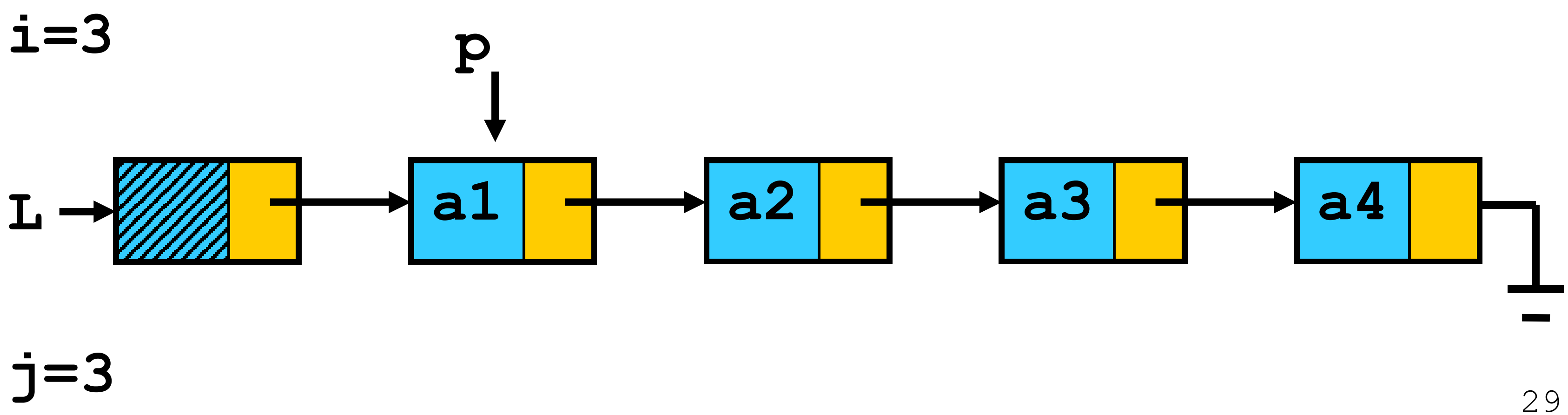
线性表的链式表示和实现

- 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)



```
Status GetElem_L
      (LinkList L, int i, ElemType &e)
{
  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 ERROR;
  e = p->data;        // copy数据到e中
  return OK;
}
```

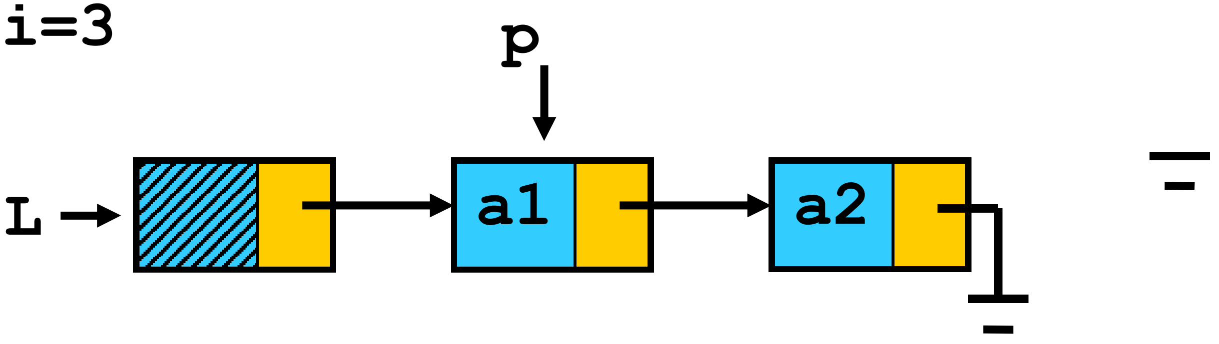
```
p = L->next;    j = 1;
// 循环直到p为空或到了第i个节点
while(p && j < i) {
    p = p->next;
    ++ j;
}
if(!p || j > i)    // 第i个节点不存在
    return ERROR;
```



```
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 ERROR;
```

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

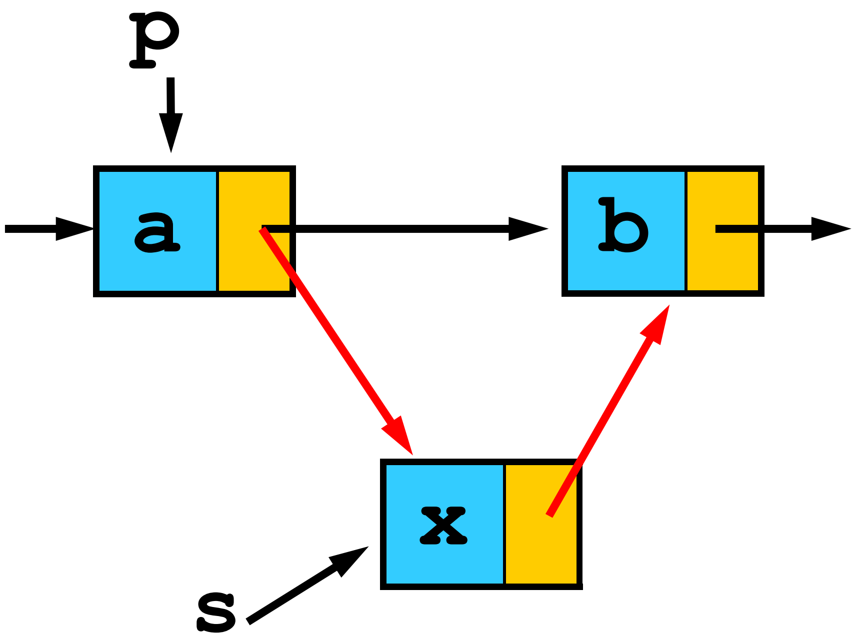
i=3



j=3

线性表的链式表示和实现

- 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



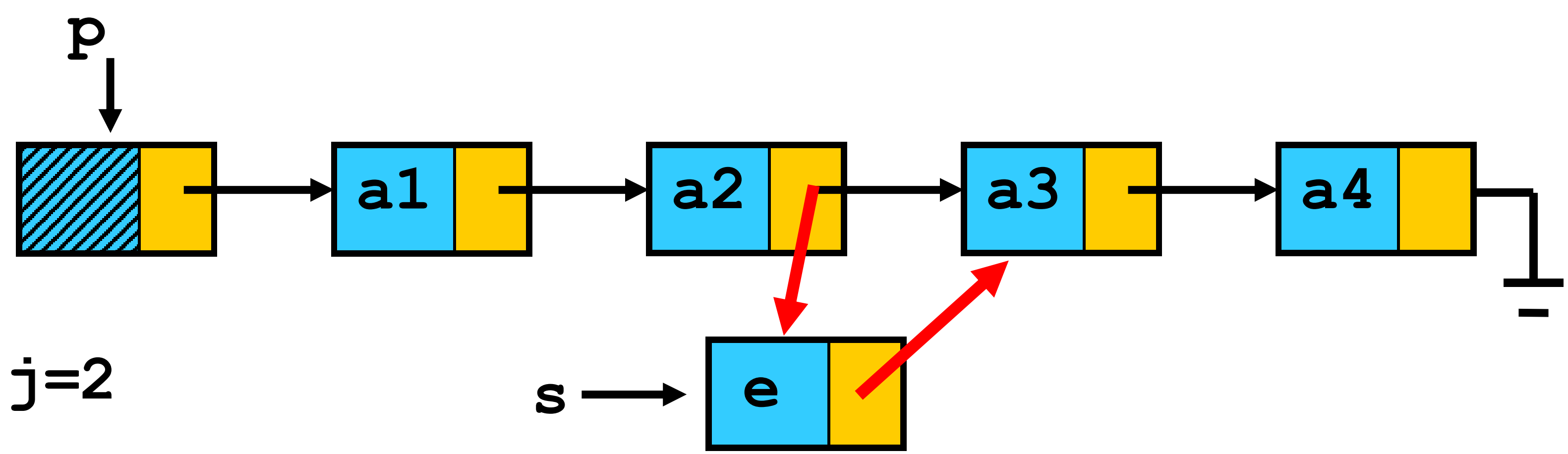
```
Status ListInsert_L
        (LinkList &L, int i, ElemType e)
{
    p = L;   j = 0;

    // 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 ERROR;
```

不就是前面的GetElem_L算法么？

```c
// 生成一个新节点，并链接到L中
s = (LinkList) malloc (sizeof(LNode));
s->data = e;
s->next = p->next;
p->next = s;
return OK;
}
```

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

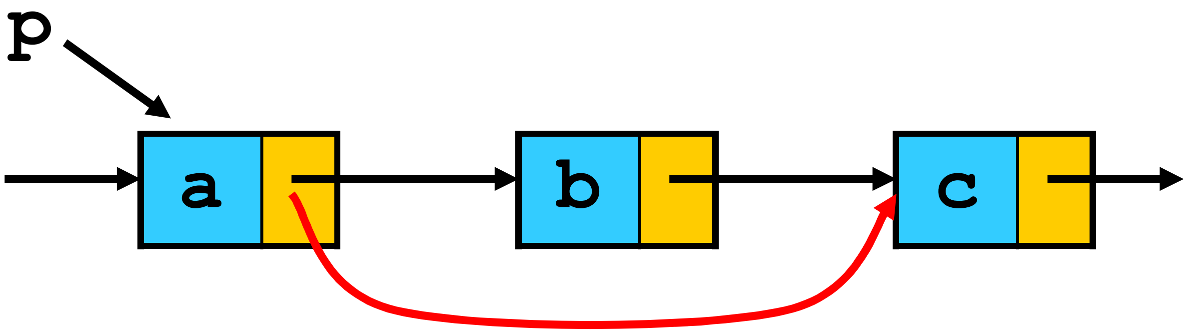


线性表的链式表示和实现

- 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 = $O(n)$

线性表的链式表示和实现

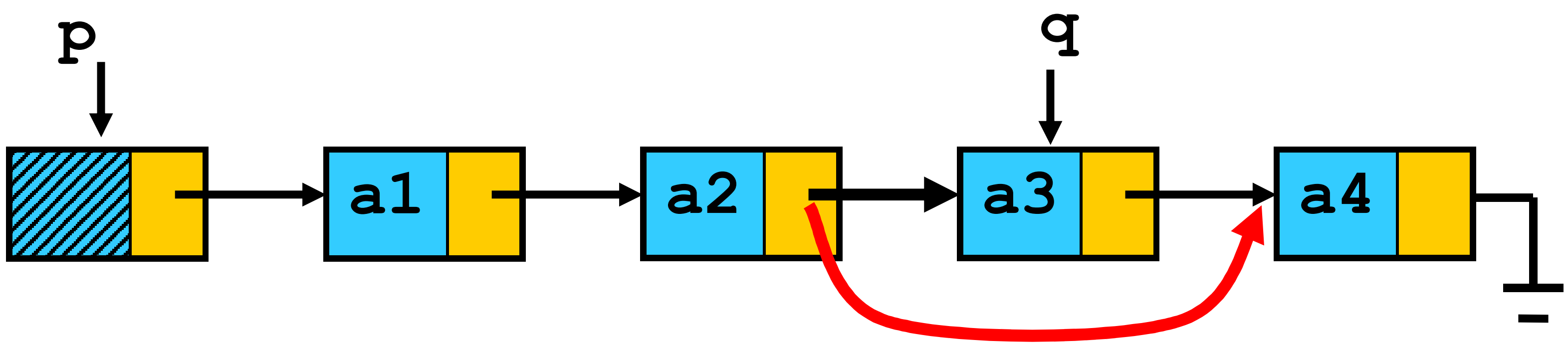
- 删除
  - 和插入类似，只需移动几个指针
  - 但是也必须先找到待删除的节点



```
Status ListDelete_L
     (LinkList &L, int i, ElemType& e)
{
     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 ERROR;
```

```
    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 OK;
}
```

i=3，即删除第3个节点



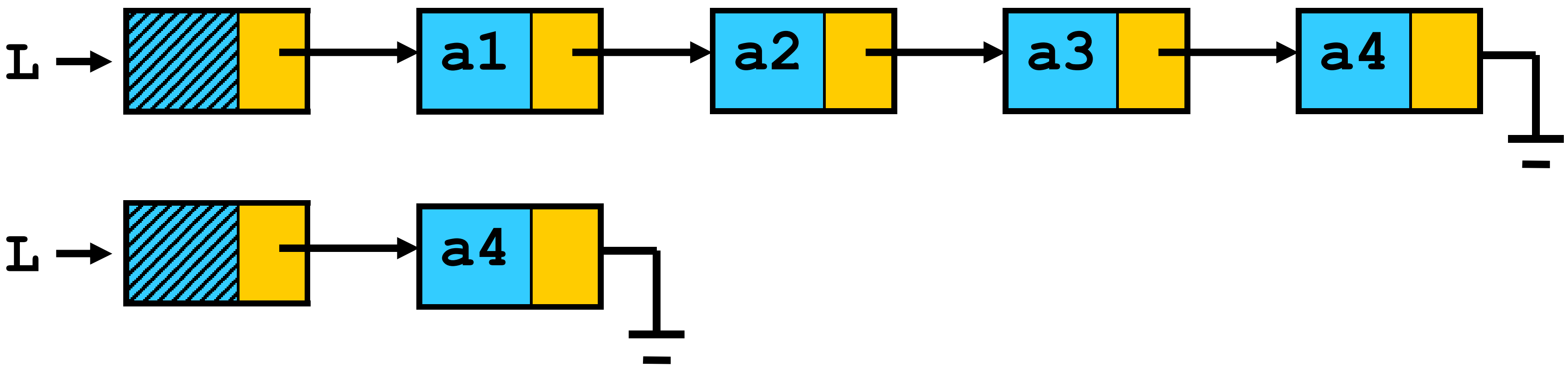
j=2

线性表的链式表示和实现

- 思考
  - 删除操作的时间复杂度是多少？

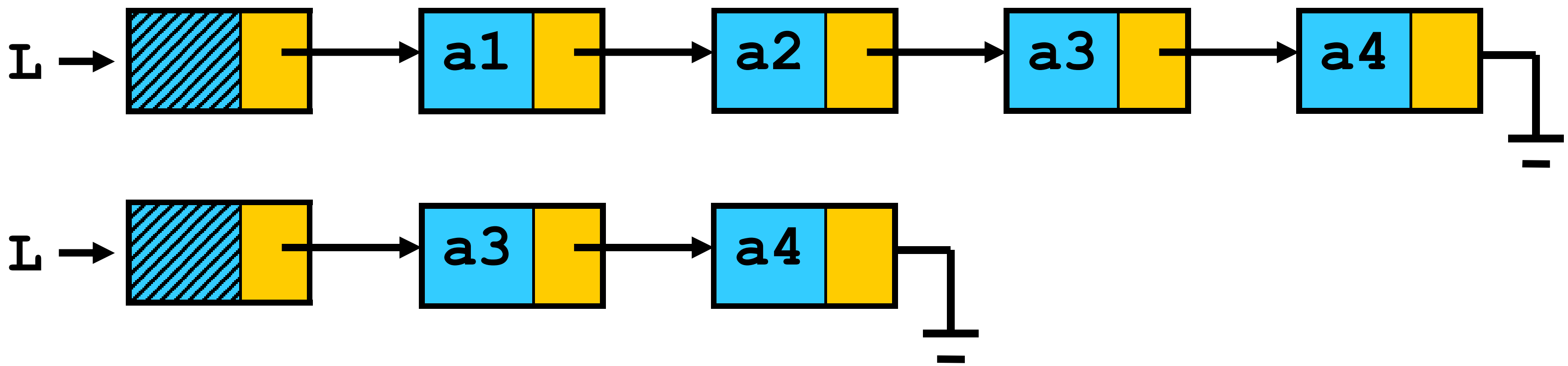
线性表的链式表示和实现

- 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



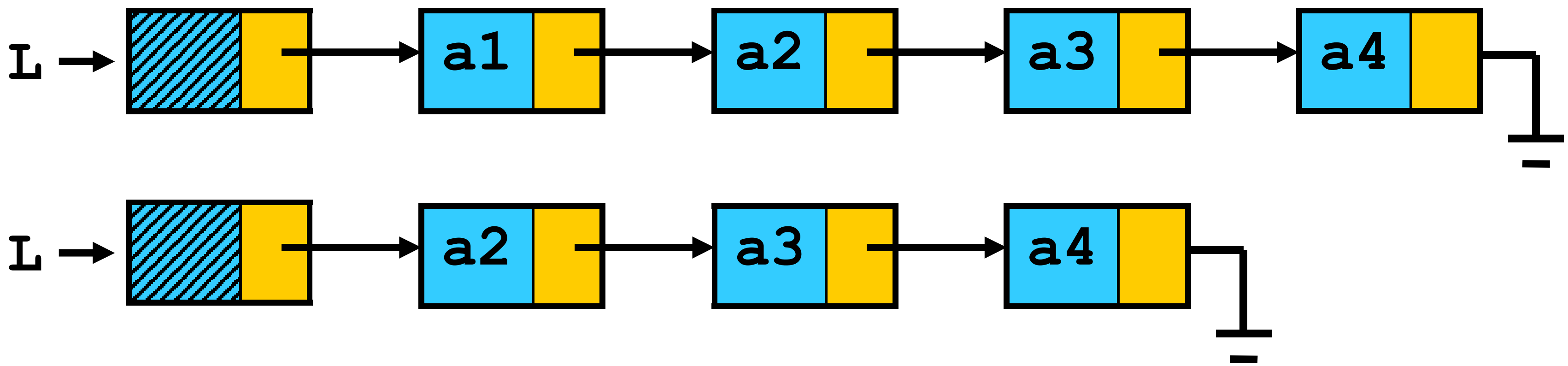
线性表的链式表示和实现

- 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



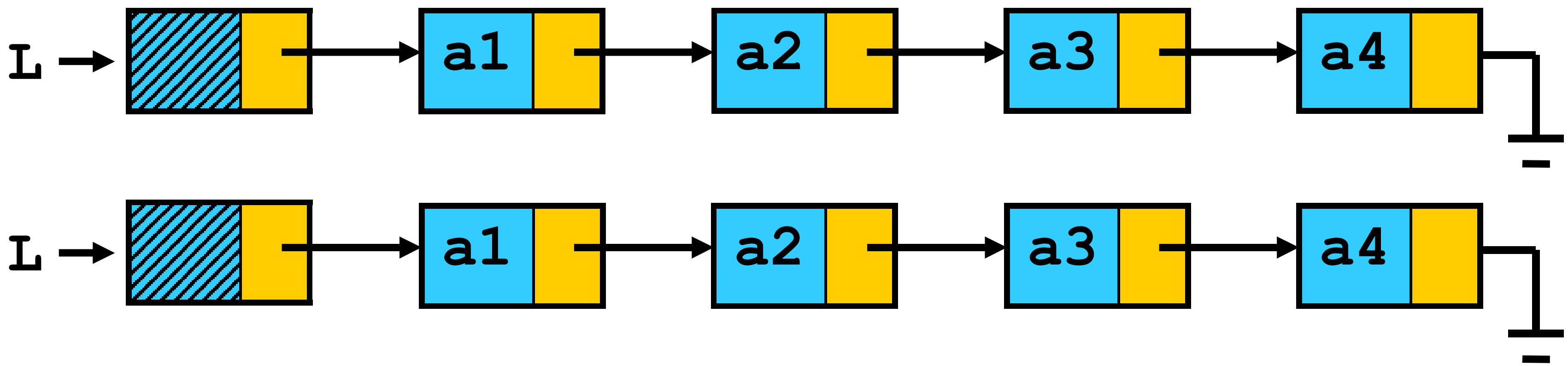
线性表的链式表示和实现

- 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



线性表的链式表示和实现

- 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



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”

静态链表

- 存储结构

```
#define MAXSIZE 1000
typedef struct{
     ElemType data;
     int cur;
}component,
SLinkList[MAXSIZE];
```

	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		

静态链表

- 优点：
  - 可以应用于没有指针的语言（比如Java）
- 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

静态链表

- 从备用链表中分配一个节点

0		1
1		2
2		3
3		4
4		5
5		0

0		2
1		0
2		3
3		4
4		5
5		0

返回该节点的下标

静态链表

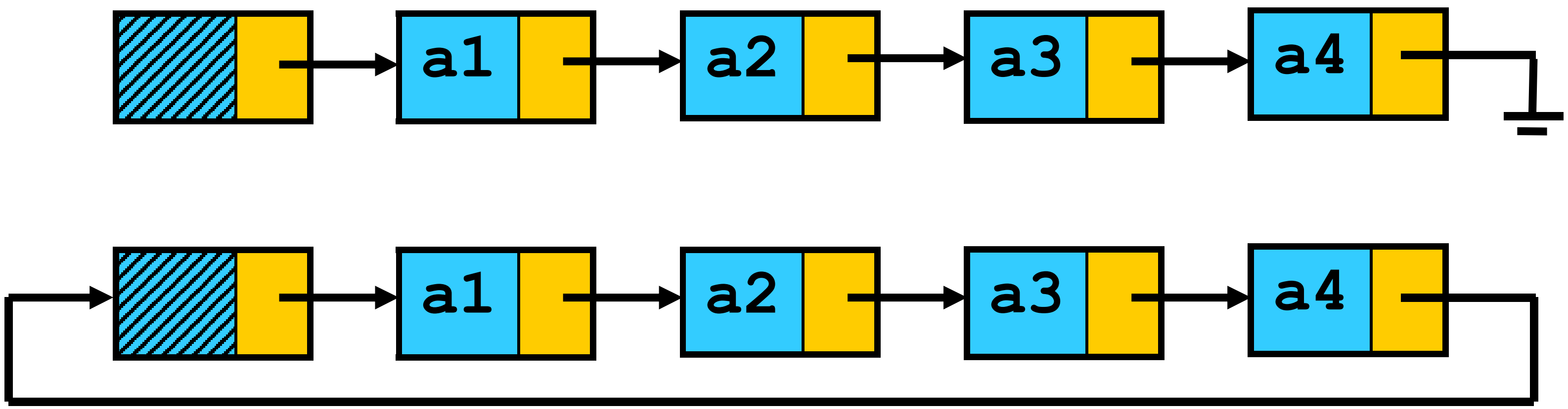
- 释放一个节点到备用链表

0		4
1		2
2		0
3		4
4		5
5		0

0		2
1		0
2		3
3		4
4		5
5		0

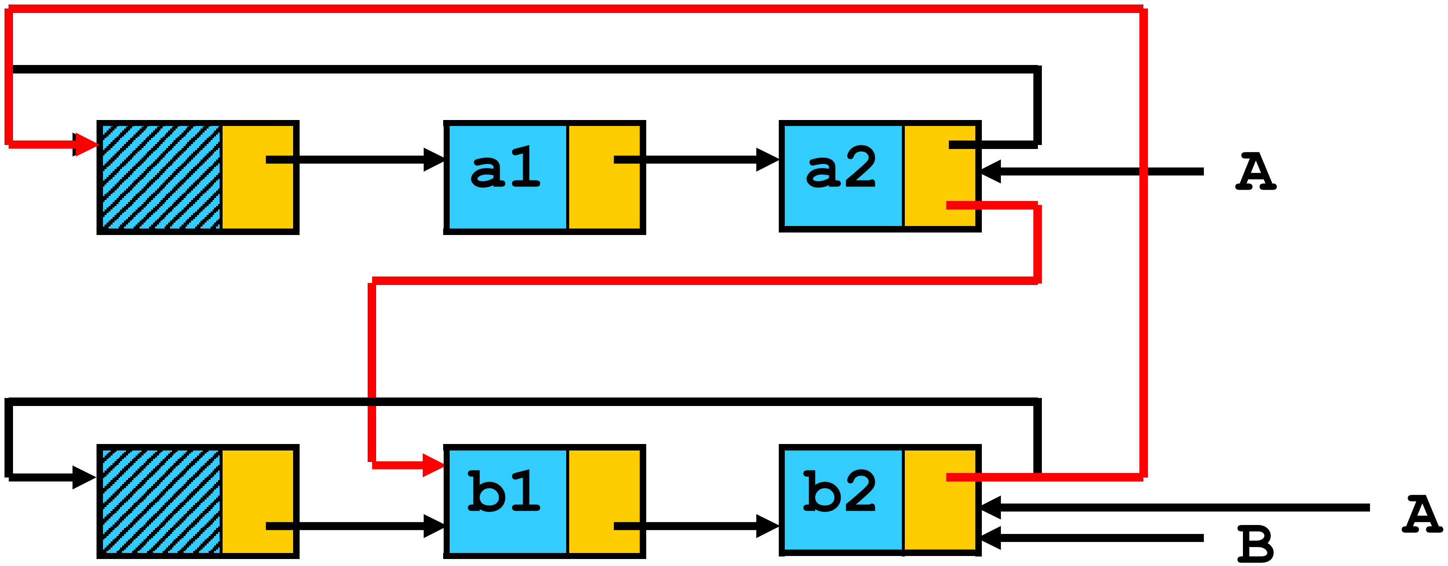
循环链表

- 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



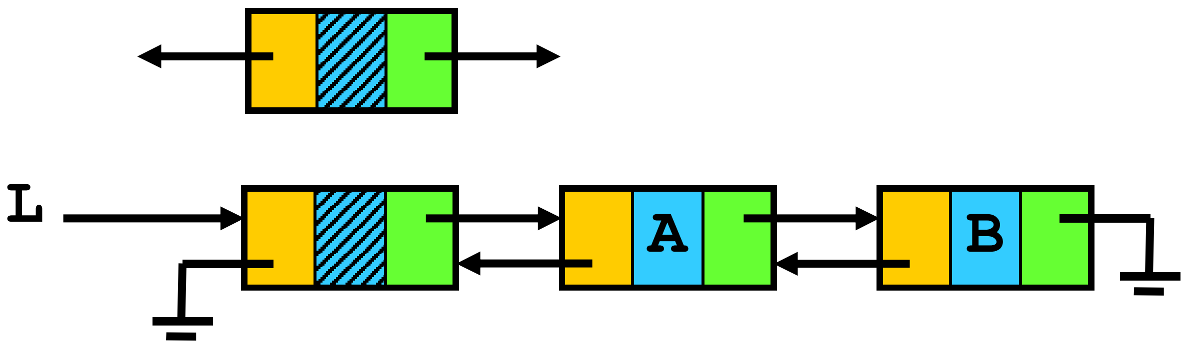
双向链表

- 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

双向链表

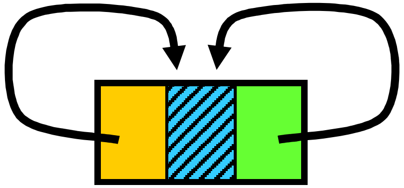
- 存储结构

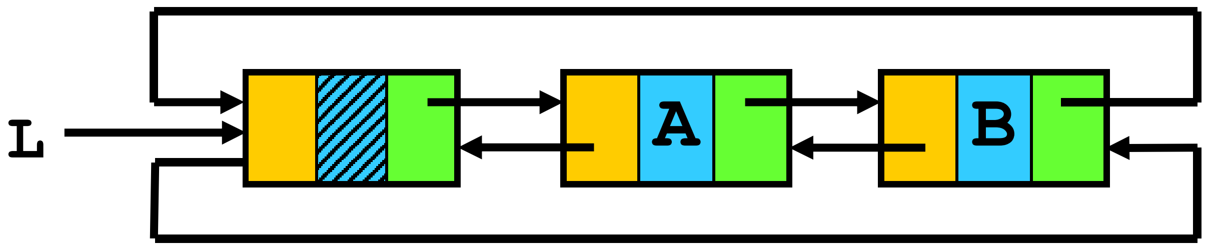
```
typedef struct DulNode{
    ElemType        data;
    struct DulNode *prior;
    struct DulNode *next;
}DulNode, *DuLinkList;
```



双向链表

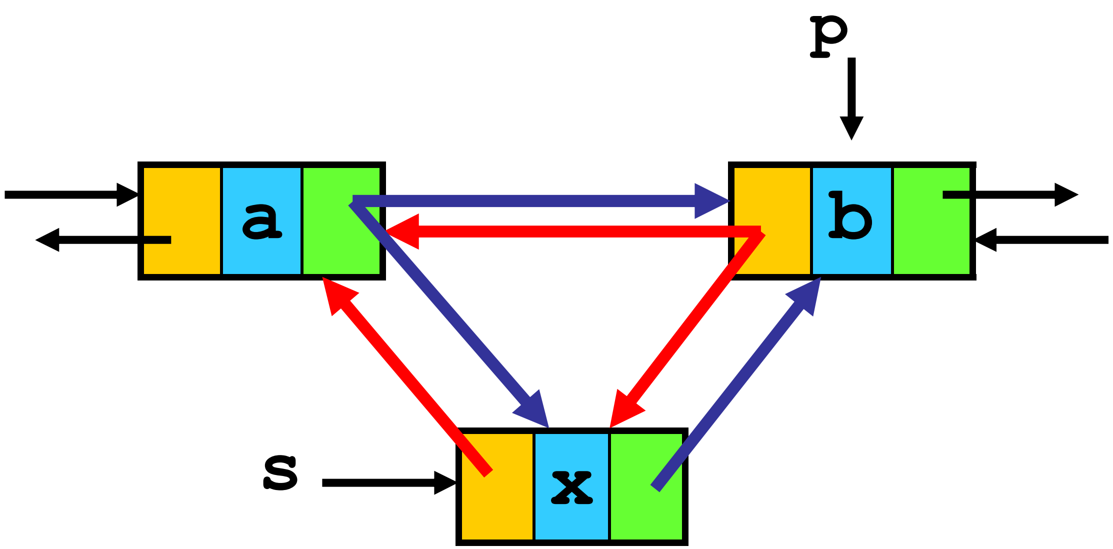
- 也可以有循环双向链表





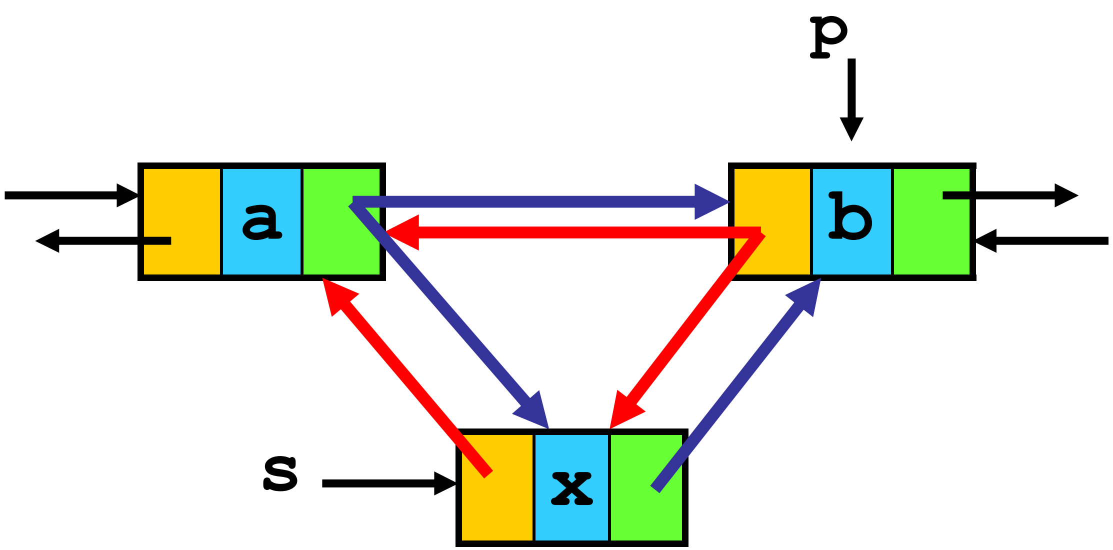
双向链表

- 插入操作
  - 在第i个节点p之前，插入节点s



双向链表

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

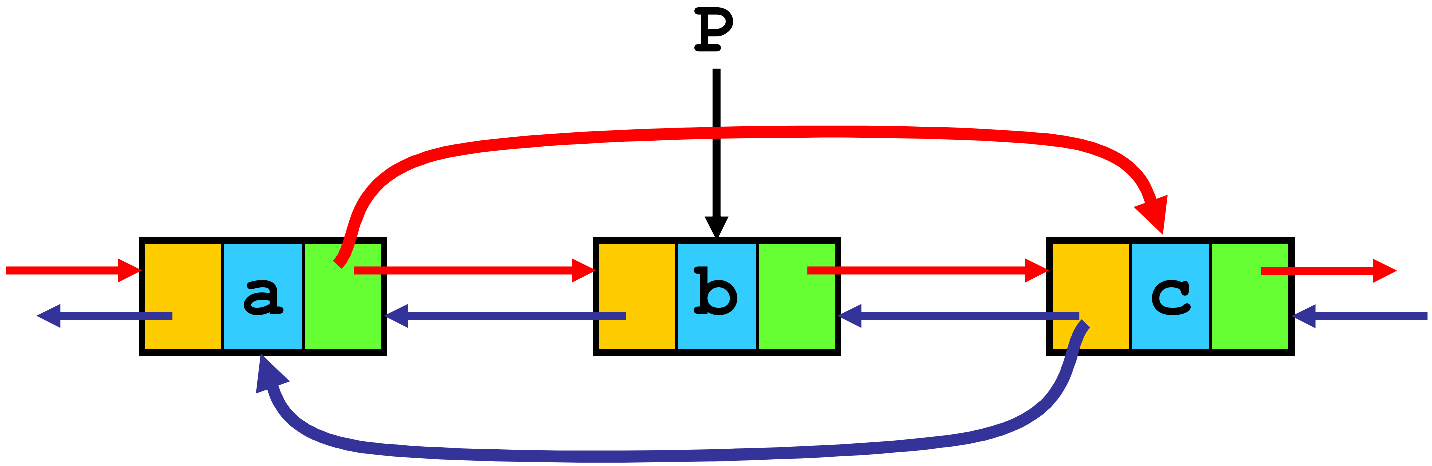


双向链表

- 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：

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

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))

（（p1, e1），(p2, e2), … , (pm, em)）

一元稀疏多项式：类型定义

ADT Polynomial {

数据对象：

D＝{a_i | a_i ∈TermSet, I =1,2,...,m, m≥0 }

{ TermSet 中的每个元素包含**:**

一个表示系数的实数和表示指数的整数 **}**

数据关系：

R＝{ < a_{i-1} , a_i >| a_{i-1} , a_i ∈D,

a_{i-1} 的指数值＜ **a_i**的指数值，**i=2,...,n }**

一元稀疏多项式：类型定义

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。

PrintPolyn (&P); //输出一元多项式P。

PolynLength(P); //多项式项数

Value (P, x); //变量为x时多项式的值

Ceof(P , int e); //指数为e的项系数

int MaxExp(P); //最大指数

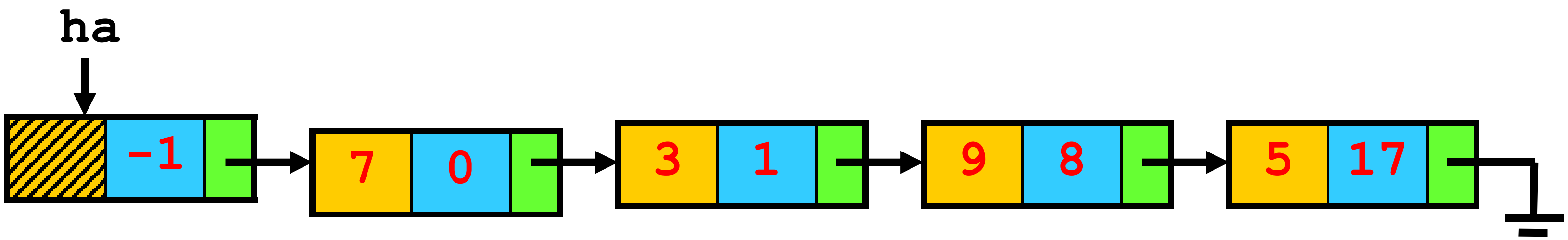
一元稀疏多项式：类型定义

```
    InsertTerm(&P, coef,expn); //插入一项
    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

- 数据表示：

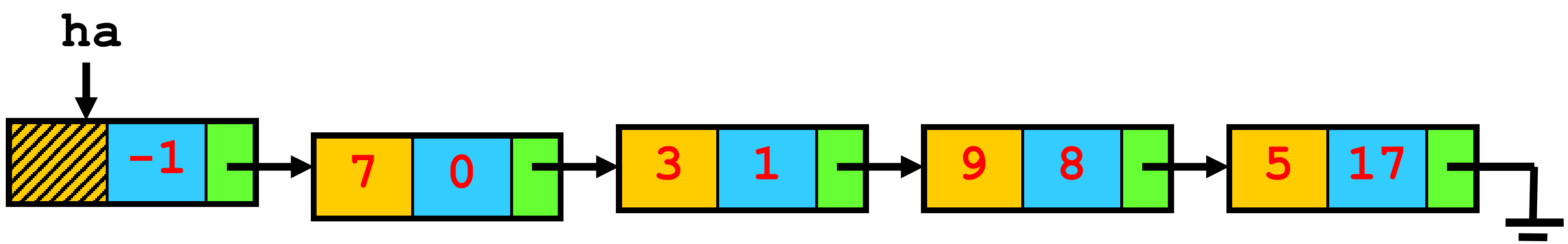
　　用带头结点的链式表表示多项式，每个结点对应多项式的一项.



一元稀疏多项式:链式实现

- ```
  typedef struct {
    float coef;
    int   expn;
  } ElemType; // 数据元素类型
  ```

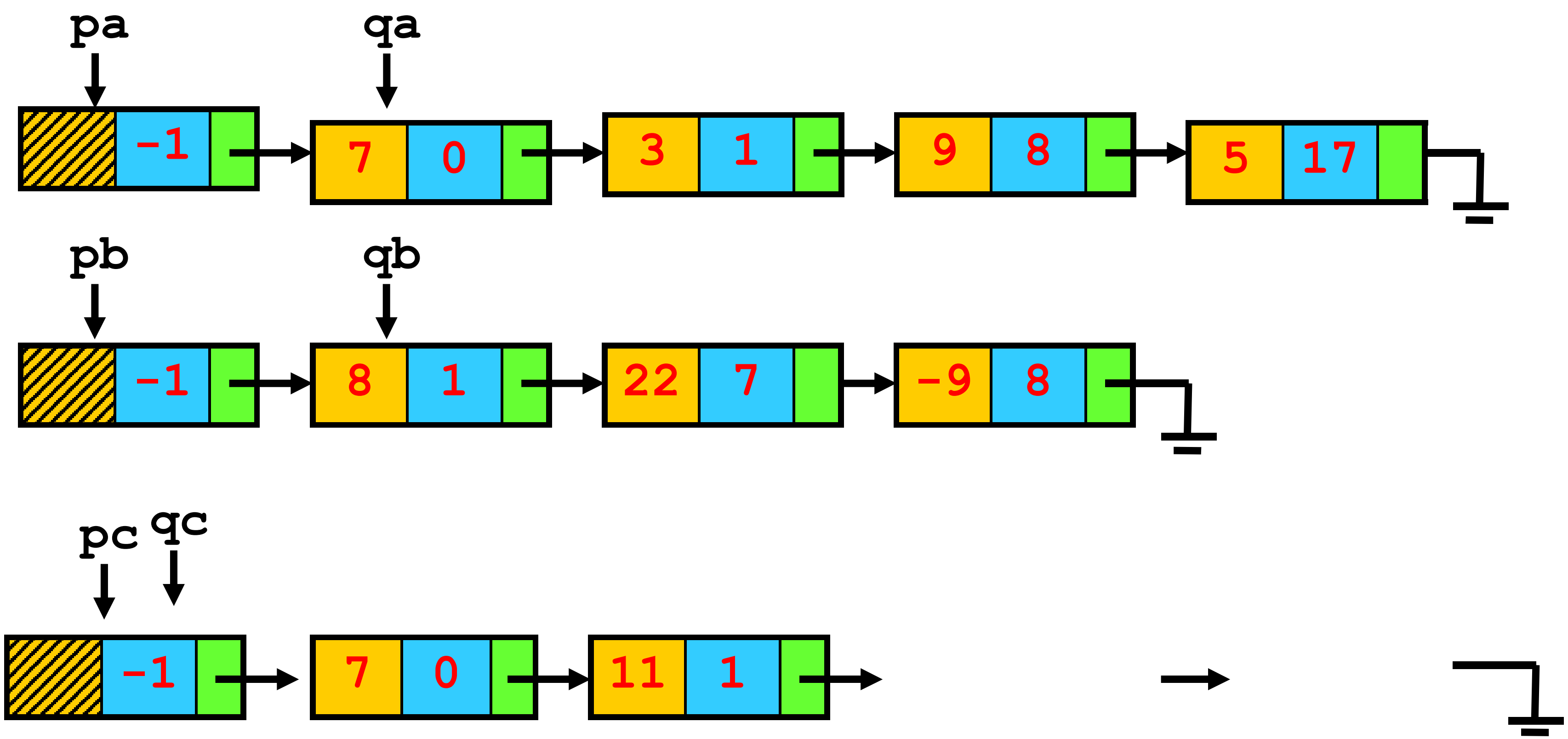
- ```
  typedef struct PolyNode{
      ElemType      data;
      struct PolyNode* next;
  }PolyNode, *Polynomial;
  ```



一元稀疏多项式：链式实现

- 多项式的基本操作实现：

  加法操作



一元稀疏多项式:链式实现

```
void AddPolyn (Polynomial Pa,
      Polynomial Pb, Polynomial &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
  }
}
```

一元稀疏多项式:链式实现

```
void AddPolyn (Polynomial &Pa,
        Polynomial Pb){


}
```

一元稀疏多项式：链式实现

- 乘法操作 ：可转化为多次加法操作

```
void MultiplyPolyn(Polynomial Pa,
    Polynomial Pb, Polynomial &Pc ){
  Polynomial 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

- 习题集 **1,2,3.**