第二章 线性表

本章的基本内容是:

- 线性表的逻辑结构
- 线性表的顺序存储及实现
- 线性表的链接存储及实现
- 顺序表和单链表的比较
- 线性表的其他存储及实现

2.1 线性表的逻辑结构

学生成绩登记表

学号	姓　名	数据结构	英语	高数
0101	丁一	78	96	87
0102	李二	90	87	78
0103	张三	86	67	86
0104	孙红	81	69	96
0105	王冬	74	87	66

2.1 线性表的逻辑结构

职工工资登记表

职工号	姓　名	基本工资	岗位津贴	奖金
0101	丁一	278	600	200
0102	李二	190	300	100
0103	张三	186	300	100
0104	孙红	218	500	200
0105	王冬	190	300	100

数据元素之间的关系是什么？

2.1 线性表的逻辑结构

线性表（ Linear List ）的定义

线性表：简称表，是n（$n\geq 0$）个具有**相同类型**的数据元素的**有限序列**。

线性表的长度：线性表中数据元素的个数。

空表：长度等于零的线性表，记为：$L=(\ )$。

非空表记为：$L=(a_1, a_2, \ldots, a_{i-1}, a_i, \ldots, a_n)$

其中，a_i（$1\leq i\leq n$）称为数据元素；
下角标 i 表示该元素在线性表中的位置或序号 。

2.1 线性表的逻辑结构

线性表的图形表示

线性表（$a_1, a_2, \ldots, a_{i-1}, a_i, \ldots, a_n$)的图形表示如下：

2.1 线性表的逻辑结构

线性表的特性

1.有限性：线性表中数据元素的个数是有穷的。

2.相同性：线性表中数据元素的类型是同一的。

3.顺序性：线性表中相邻的数据元素a_{i-1}和a_i之间存在序偶关系(a_{i-1}, a_i)，即a_{i-1}是a_i的前驱，a_i是a_{i-1}的后继；a_1无前驱，a_n无后继，其它每个元素有且仅有一个前驱和一个后继。

2.1 线性表的逻辑结构

线性表的抽象数据类型定义

ADT List

Data

线性表中的数据元素具有相同类型，
相邻元素具有前驱和后继关系

Operation

InitList

前置条件：表不存在

输入：无

功能：表的初始化

输出： 无

后置条件：建一个空表

2.1 线性表的逻辑结构

线性表的抽象数据类型定义

DestroyList

前置条件：表已存在
输入：无
功能：销毁表
输出：无
后置条件：释放表所占用的存储空间

Length

前置条件：表已存在
输入：无
功能：求表的长度
输出：表中数据元素的个数
后置条件：表不变

2.1 线性表的逻辑结构

线性表的抽象数据类型定义

Get

前置条件：表已存在
输入：元素的序号i
功能：在表中取序号为i的数据元素
输出：若i合法，返回序号为i的元素值，否则抛出异常
后置条件：表不变

Locate

前置条件：表已存在
输入：数据元素x
功能：在线性表中查找值等于x的元素
输出：若查找成功，返回x在表中的序号，否则返回0
后置条件：表不变

2.1 线性表的逻辑结构

线性表的抽象数据类型定义

Insert

前置条件：表已存在

输入：插入i；待插x

功能：在表的第i个位置处插入一个新元素x

输出：若插入不成功，抛出异常

后置条件：若插入成功，表中增加一个新元素

Delete

前置条件：表已存在

输入：删除位置i

功能：删除表中的第i个元素

输出：若删除成功，返回被删元素，否则抛出异常

后置条件：若删除成功，表中减少一个元素

2.1 线性表的逻辑结构

线性表的抽象数据类型定义

Empty

前置条件：表已存在

输入：无

功能：判断表是否为空

输出：若是空表，返回1，否则返回0

后置条件：表不变

PrintList

前置条件：表已存在

输入：无

功能：遍历操作，按序号依次输出表中元素

输出：表的各数据元素

后置条件：表不变

endADT

2.1 线性表的逻辑结构

进一步说明：

（1）线性表的基本操作根据实际应用而定；

（2）复杂的操作可以通过基本操作的组合来实现；

- 假设利用两个线性表LA和LB分别表示两个集合A和B(即：线性表中的数据元素即为集合中的成员)，现要求一个新的集合A=AUB：

Get(one from B)-Locate(in A)-Insert(to A)?

（3）对不同的应用，操作的接口可能不同。

- 删除表中的值为x的元素
- 删除表中的第i个元素

2.2 线性表的顺序存储结构及实现

顺序表——线性表的顺序存储结构

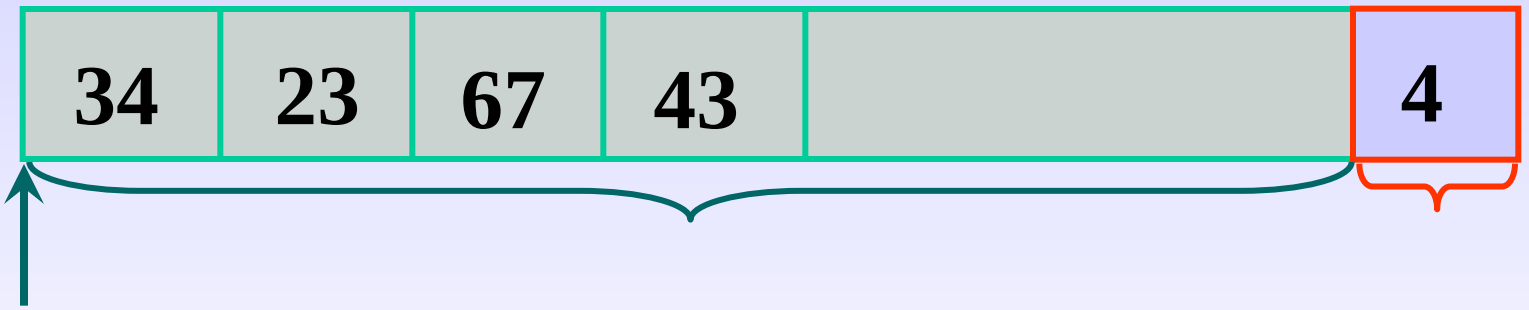
例：（34, 23, 67, 43）

存储要点
- 用一段地址连续的存储单元
- 依次存储线性表中的数据元素

2.2 线性表的顺序存储结构及实现

顺序表——线性表的顺序存储结构

例：（34, 23, 67, 43）



用什么属性来描述顺序表？

- 存储空间的起始位置
- 顺序表的容量（最大长度）
- 顺序表的当前长度

2.2 线性表的顺序存储结构及实现

线性表的顺序存储结构——顺序表

例：（34, 23, 67, 43）

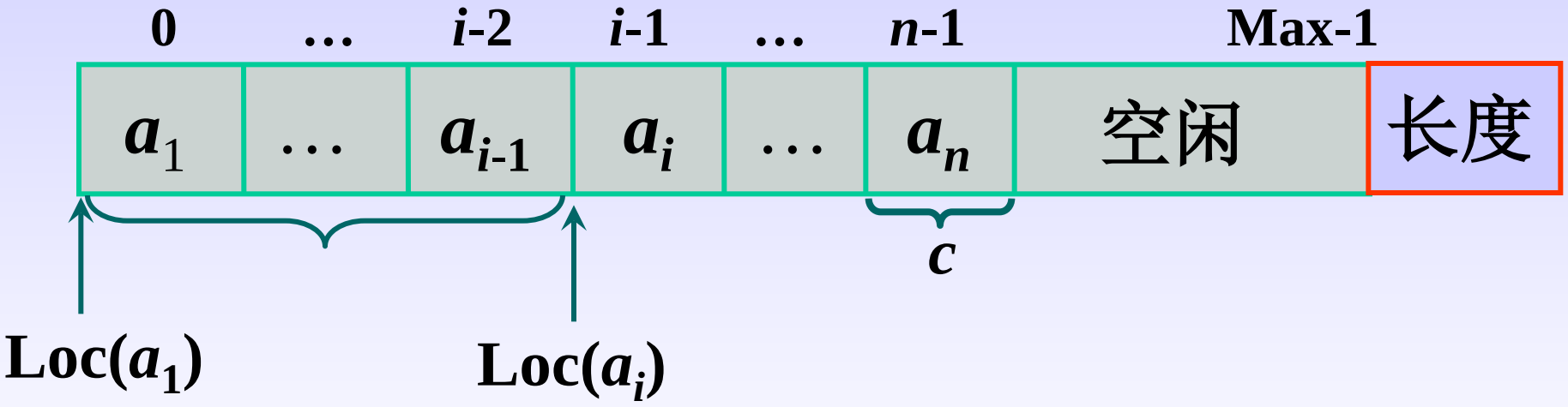
如何实现顺序表的内存分配？

顺序表 → 一维数组

2.2 线性表的顺序存储结构及实现

顺序表

一般情况下，$(a_1, a_2, \ldots, a_{i-1}, a_i, \ldots, a_n)$的顺序存储：

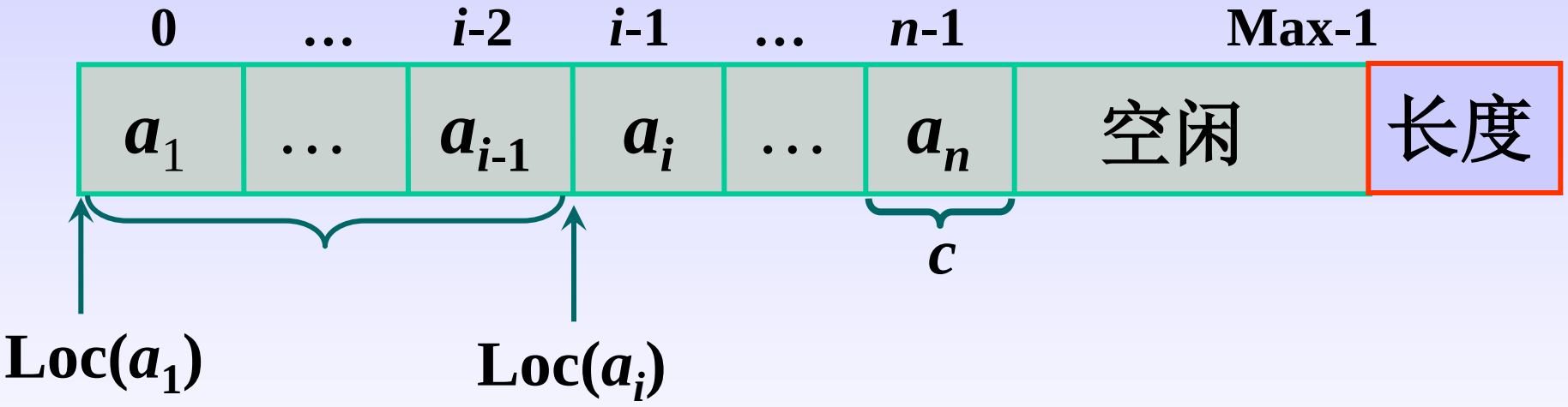


如何求得任意元素的存储地址？

2.2 线性表的顺序存储结构及实现

顺序表

一般情况下，$(a_1, a_2, \ldots, a_{i-1}, a_i, \ldots, a_n)$的顺序存储：



$$Loc(a_i)=Loc(a_1) + (i-1)\times c$$

随机存取： 在$O(1)$时间内存取数据元素

2.2 线性表的顺序存储结构及实现

存储结构和存取结构

存储结构是数据及其逻辑结构在计算机中的表示；
存取结构是在一个数据结构上对查找操作的时间性能的一种描述。

“顺序表是一种随机存取的存储结构”的含义为：在顺序表这种存储结构上进行的查找操作，其时间性能为$O(1)$。

2.2 线性表的顺序存储结构及实现

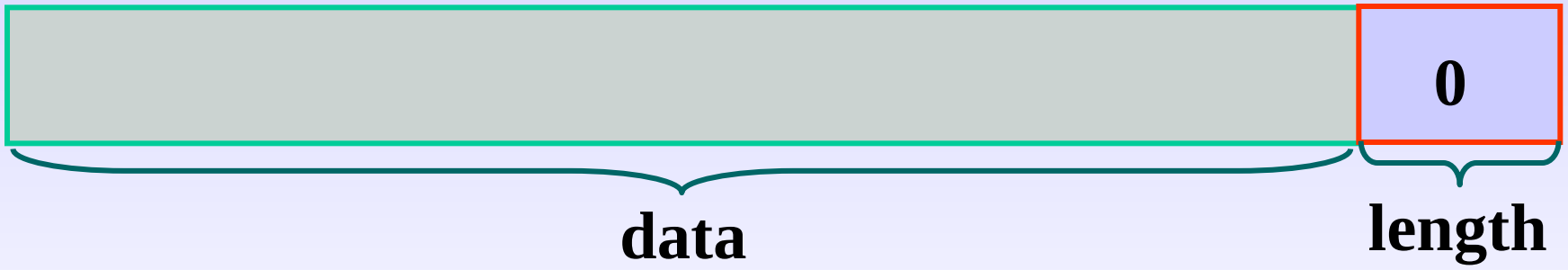
顺序表类的声明

```
const int MaxSize=100;
template <class T>          //模板类
class SeqList
{
 public:
    SeqList( ) ;                //构造函数
    SeqList(T a[ ], int n);
    ~SeqList( ) ;               //析构函数
    int Length( ){return length;}
    T Get(int i);
    int Locate(T x );
    void Insert(int i, T x);
    T Delete(int i);
    void PrintList();
 private:
    T data[MaxSize];
    int length;
};
```

2.2 线性表的顺序存储结构及实现

顺序表的实现——无参构造函数

操作接口：**SeqList（ ）**



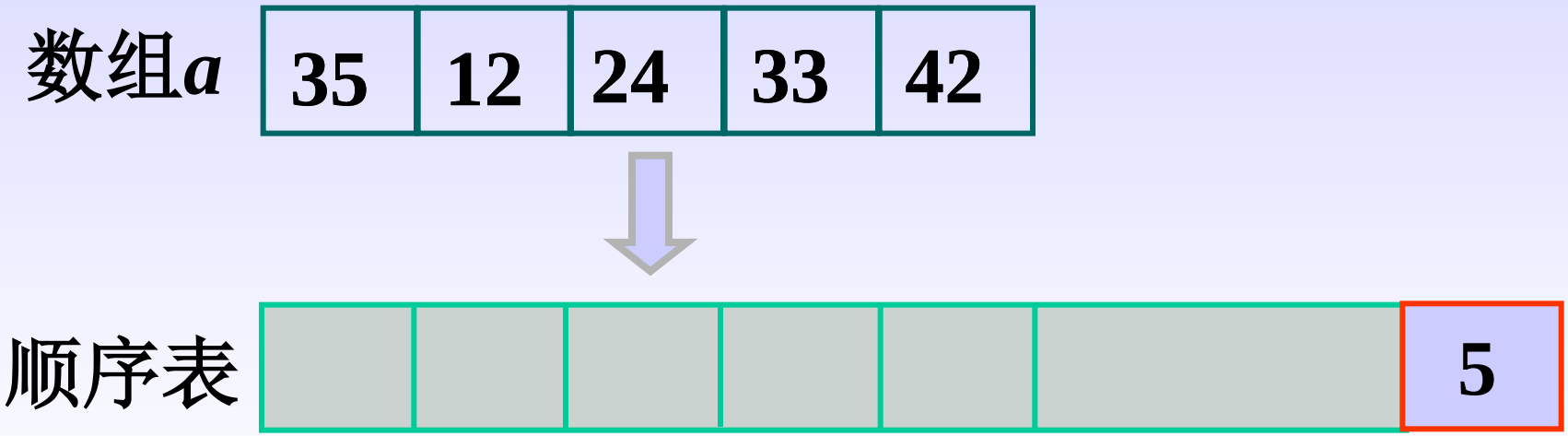
算法描述：

```
template <class T>
SeqList<T>:: SeqList( )
{  length=0;}
```

2.2 线性表的顺序存储结构及实现

顺序表的实现——有参构造函数

操作接口：**SeqList(T a[ ], int n)**



2.2 线性表的顺序存储结构及实现

顺序表的实现——有参构造函数

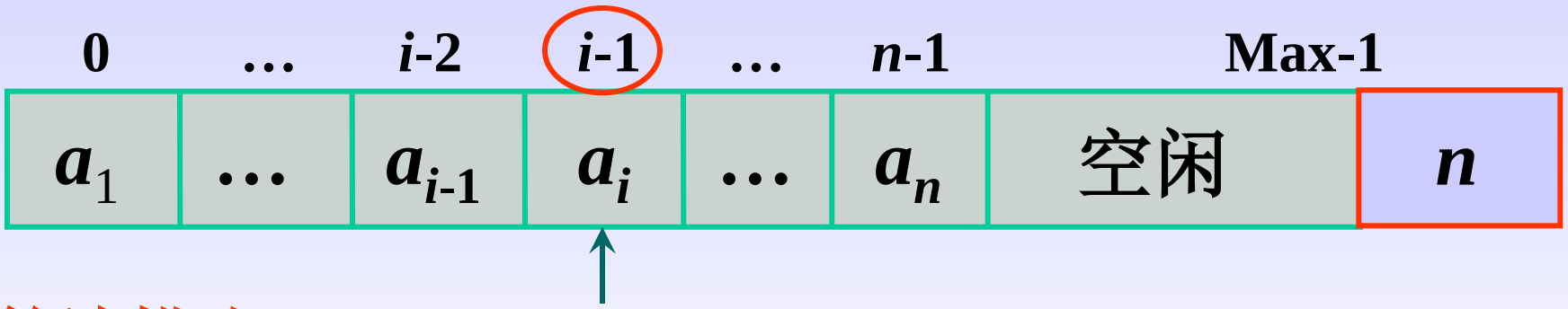
算法描述：

```
template <class T>
SeqList<T>::SeqList（T a[ ], int n）
{
      if（n>MaxSize）throw "参数非法";
      for（i=0; i<n; i+ +）
            data[i]=a[i];
      length=n;
 }
```

2.2 线性表的顺序存储结构及实现

顺序表的实现——按位查找

操作接口： T Get(int i)



算法描述：

时间复杂度？

```
template <class T>
T SeqList<T>::Get( int i )
{
    if (i>=1 && i<=length) return data[i-1];
 }
```

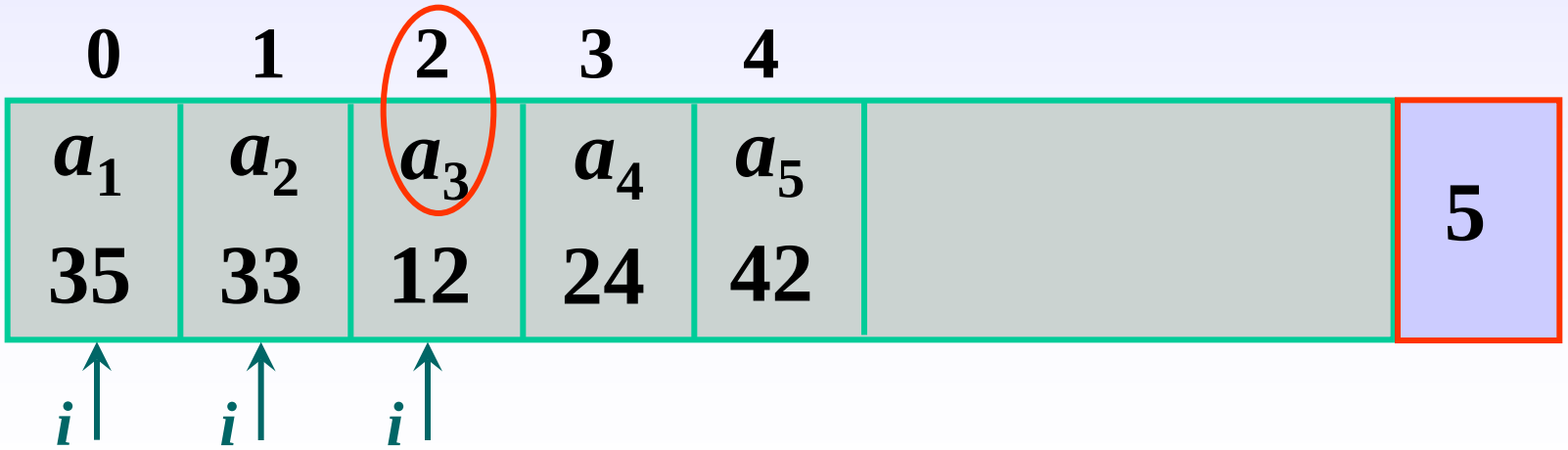
2.2 线性表的顺序存储结构及实现

顺序表的实现——按值查找

操作接口： **int Locate(T x)**

例：在（35, 33, 12, 24, 42） 中查找值为12的元素，返回在表中的序号。

注意序号和下标之间的关系



2.2 线性表的顺序存储结构及实现

顺序表的实现——按值查找

算法描述：

```
template <class T>
int SeqList<T>::Locate(T x)
{
    for(i=0; i<length; i++)
        if(data[i]==x) return i+1;
    return 0;
}
```

时间复杂度？

2.2 线性表的顺序存储结构及实现

顺序表的实现——遍历

算法描述：

```
template <class T>
void SeqList<T>::PrintList()
{
    for（i=0; i<length; i++）
        cout<<data[i];
}
```

时间复杂度？

2.2 线性表的顺序存储结构及实现

顺序表的实现——插入

操作接口： **void Insert(int i, T x)**

插入前：$(a_1, \dots, a_{i-1}, a_i, \dots, a_n)$

插入后：$(a_1, \dots, a_{i-1}, x, a_i, \dots, a_n)$

a_{i-1}和a_i之间的逻辑关系发生了变化

↓

顺序存储要求存储位置反映逻辑关系

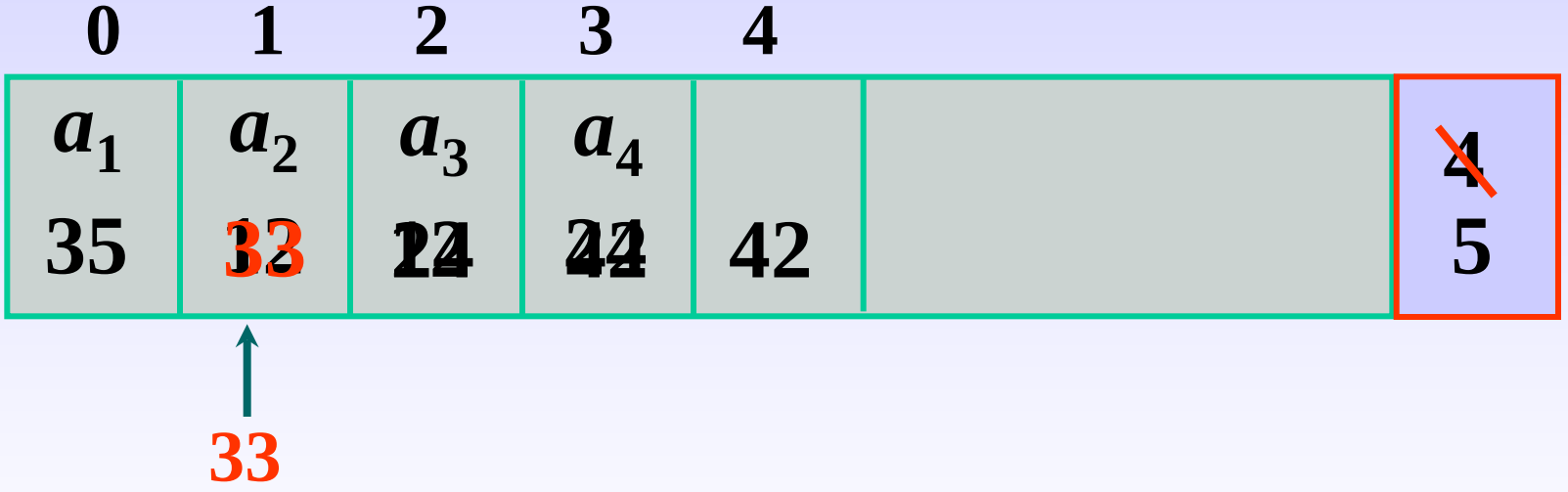
↓

存储位置要反映这个变化

2.2 线性表的顺序存储结构及实现

顺序表的实现——插入

例：（35，12，24，42），在i=2的位置上插入33。



什么时候不能插入？ ➡ 注意边界条件

表满：length>=MaxSize

合理的插入位置：1≤i≤length+1（i指的是元素的序号）

2.2 线性表的顺序存储结构及实现

顺序表的实现——插入

算法描述——伪代码

1. 如果表满了，则抛出上溢异常；
2. 如果元素的插入位置不合理，则抛出位置异常；
3. 将最后一个元素至第i个元素分别向后移动一个位置；
4. 将元素x填入位置i处；
5. 表长加1；

2.2 线性表的顺序存储结构及实现

顺序表的实现——插入

算法描述——C++描述

基本语句？

```
template <class T>
void SeqList<T>::Insert(int i, T x)
{
    if (length>=MaxSize) throw "上溢";
    if (i<1 || i>length+1) throw "位置";
    for (j=length; j>=i; j--)
        data[j]=data[j-1];
    data[i-1]=x;
    length++;
}
```

2.2 线性表的顺序存储结构及实现

顺序表的实现——插入

时间性能分析

最好情况（$i=n+1$）：

基本语句执行0次，时间复杂度为$O(1)$。

最坏情况（$i=1$）：

基本语句执行n次，时间复杂度为$O(n)$。

2.2 线性表的顺序存储结构及实现

顺序表的实现——插入

时间性能分析

平均情况（$1 \le i \le n+1$）：

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

时间复杂度为$O(n)$。

插入位置+移动次数=n+1

插入位置	移动次数
n+1	0
n	1
n-1	2
…	…
i	(n+1)-i
…	…
1	n

2.2 线性表的顺序存储结构及实现

顺序表的实现——删 除

操作接口： **T Delete(int i)**

删除前：$(a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$

删除后：$(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)$

a_{i-1}和a_i之间的逻辑关系发生了变化

↓

顺序存储要求存储位置反映逻辑关系

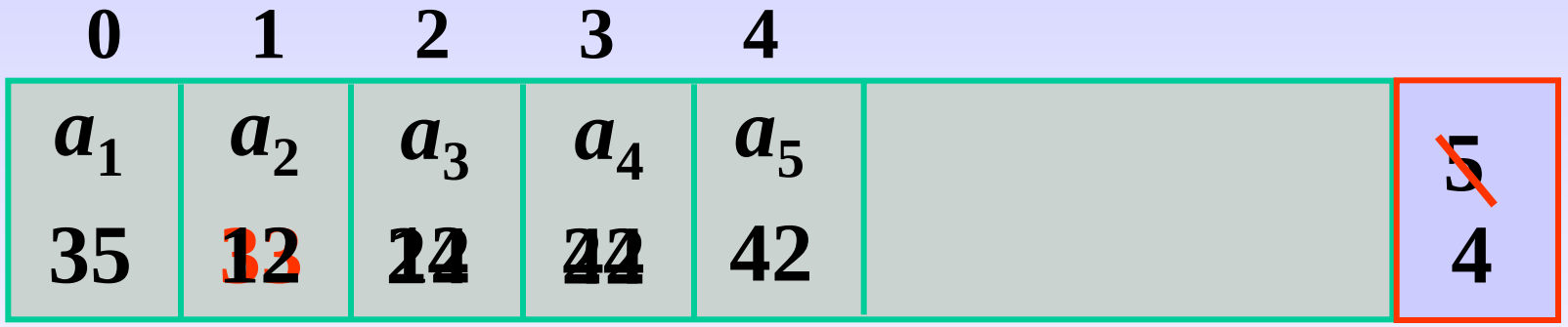
↓

存储位置要反映这个变化

2.2 线性表的顺序存储结构及实现

顺序表的实现——删 除

例：（35, 33, 12, 24, 42），删除i=2的数据元素。



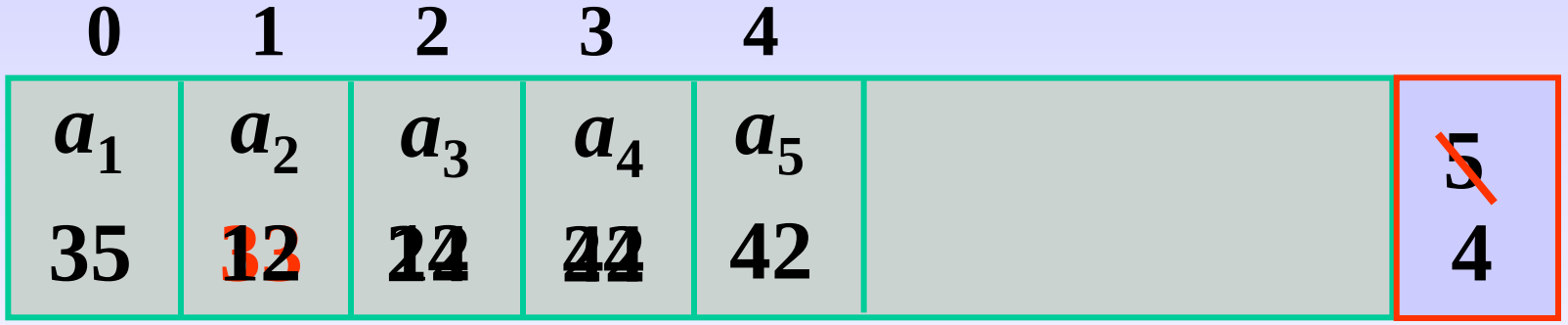
仿照顺序表的插入操作，完成：

1. 分析边界条件；
2. 分别给出伪代码和C++描述的算法；
3. 分析时间复杂度。

2.2 线性表的顺序存储结构及实现

顺序表的实现——删 除

例：（35, 33, 12, 24, 42），删除i=2的数据元素。



什么时候不能删除？ ➡ 注意边界条件

表空：length==0

合理的删除位置：1≤i≤length（i指的是元素的序号）

2.2 线性表的顺序存储结构及实现

顺序表的实现——删 除

算法描述——C++描述

基本语句？

```
template <class T>
T SeqList<T>::Delete（int i）
{
    if（length==0）throw “下溢";
    if（i<1 || i>length）throw “位置”；
    x=data[i-1];
    for （j=i; j<length; j++）
        data[j-1]=data[j];
     length--;
     return x;
}
```

2.2 线性表的顺序存储结构及实现

顺序表的实现——删 除

时间性能分析

最好情况（ $i=n$）：

基本语句执行0次，时间复杂度为$O(1)$。

最坏情况（ $i=1$）：

基本语句执行$n-1$次，时间复杂度为$O(n)$。

2.2 线性表的顺序存储结构及实现

顺序表的实现——删 除

时间性能分析

平均情况（$1 \le i \le n$）：

$$\sum_{i=1}^{n} p_i (n-i) = \frac{1}{n}\sum_{i=1}^{n}(n-i) = \frac{n-1}{2} = O(n)$$

时间复杂度为$O(n)$。

删除位置+移动次数=n

删除位置	移动次数
n	0
n-1	1
n-2	2
…	…
i	n-i
…	…
1	n-1

2.2 线性表的顺序存储结构及实现

顺序表的优缺点

- 顺序表的优点：

(1) 无需为表示表中元素之间的逻辑关系而增加额外的存储空间；

(2) 随机存取：可以快速地存取表中任一位置的元素。

- 顺序表的缺点：

(1) 插入和删除操作需要移动大量元素；

(2) 表的容量难以确定，表的容量难以扩充；

(3) 造成存储空间的碎片。

2.3 线性表的链接存储结构及实现

单链表

顺序表 → 静态存储分配 → 事先确定容量

链 表 → 动态存储分配 → 运行时分配空间

单链表：线性表的链接存储结构。

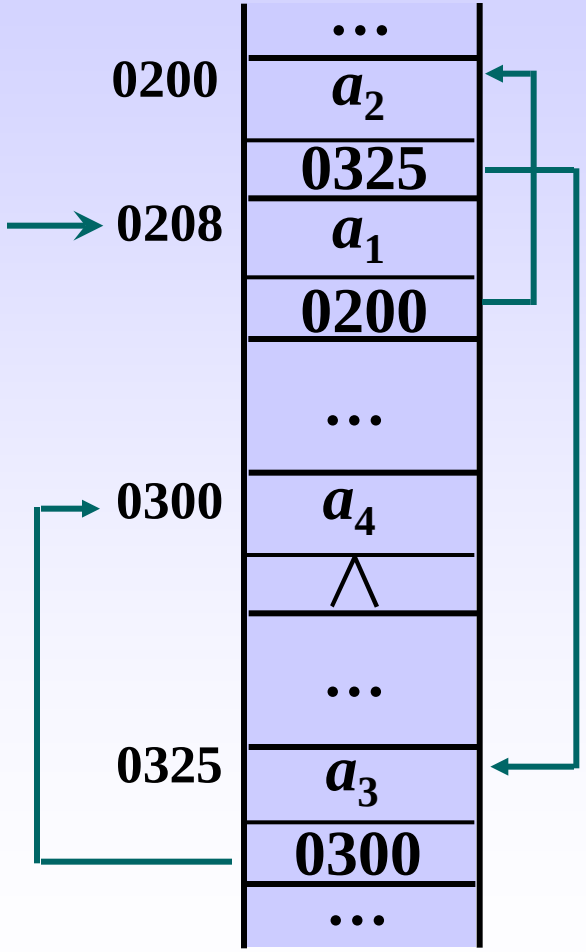
存储思想：用一组任意的存储单元存放线性表的元素。

→ 连续
→ 不连续
→ 零散分布

2.3 线性表的链接存储结构及实现

单链表

例：(a_1, a_2, a_3, a_4)的存储示意图



存储特点：

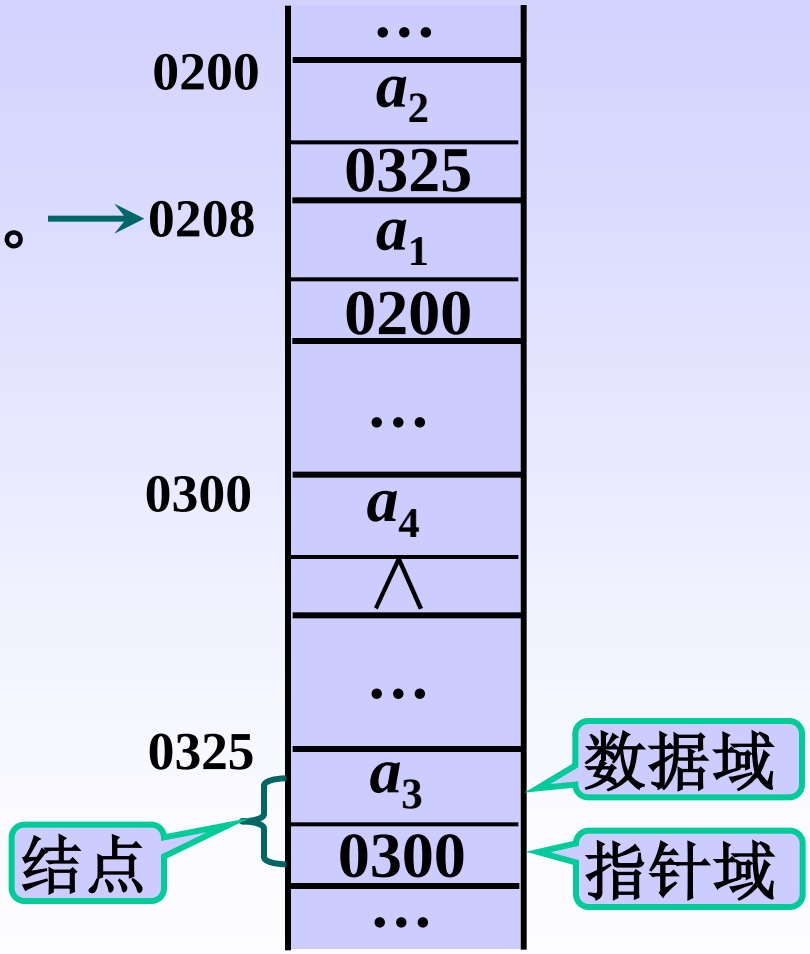
1. 逻辑次序和物理次序不一定相同。
2. 元素之间的逻辑关系用指针表示。

2.3 线性表的链接存储结构及实现

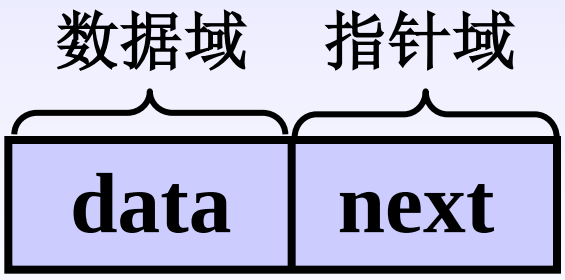
单链表

单链表是由若干结点构成的；

单链表的结点只有一个指针域。



单链表的结点结构：



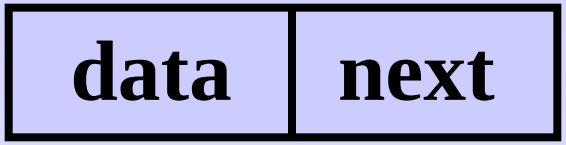
data：存储数据元素

next：存储指向后继结点的地址

2.3 线性表的链接存储结构及实现

单链表

单链表的结点结构：



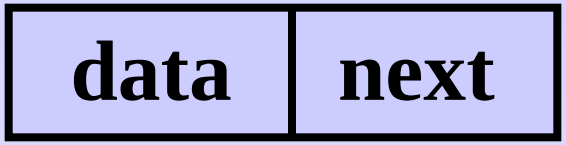

```
template <class T>
struct Node
{
    T data;
    Node<T> *next;
};
```

如何申请一个结点？

2.3 线性表的链接存储结构及实现

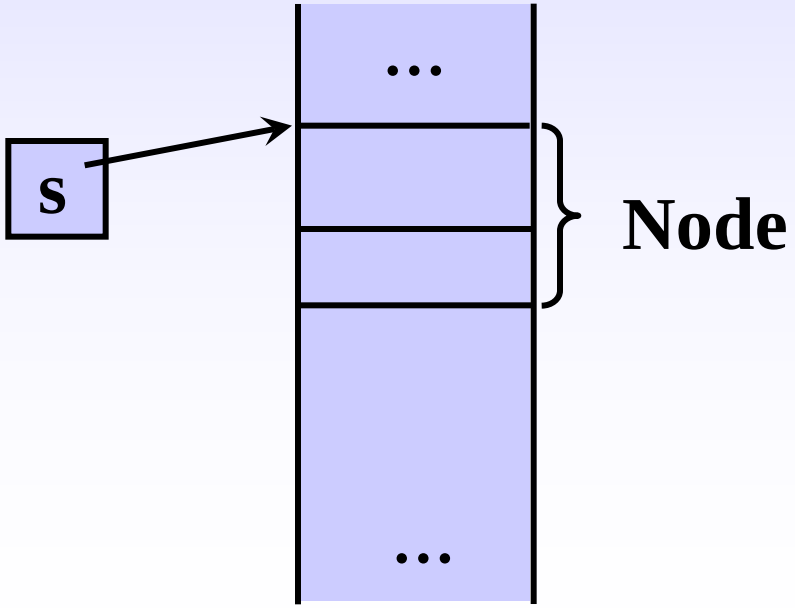
单链表

单链表的结点结构：




```
template <class T>
struct Node
{
    T data;
    Node<T> *next;
};
 Node<T> *s, *first;
```

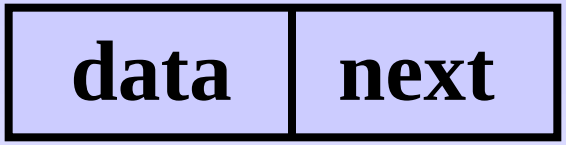
s=new Node<int> ;



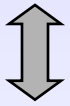
2.3 线性表的链接存储结构及实现

单链表

如何引用数据元素？



*s.data ;

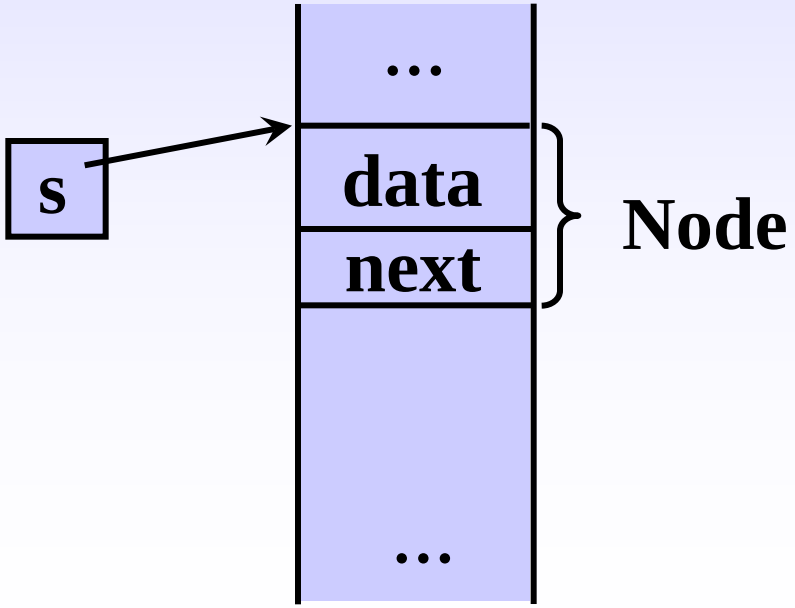


s->data ;

s=new Node<int> ;

如何引用指针域？

s->next;

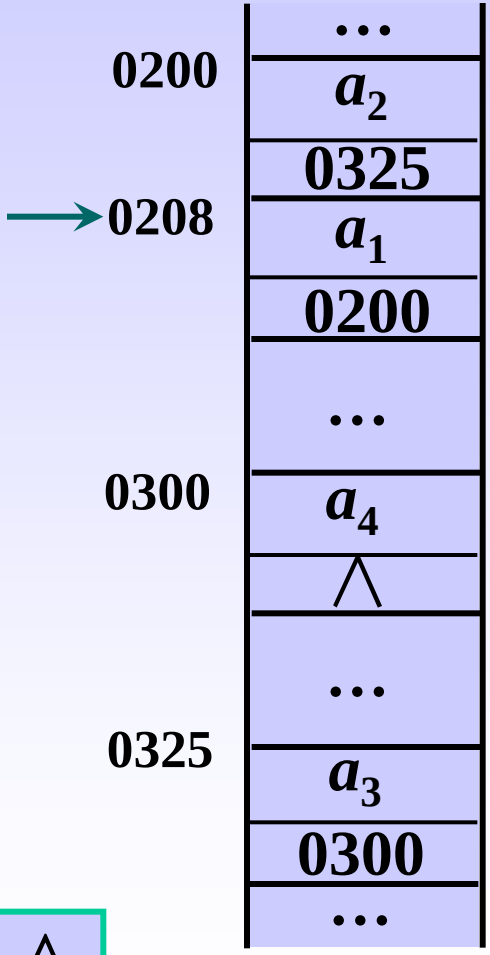


2.3 线性表的链接存储结构及实现

单链表

什么是存储结构？

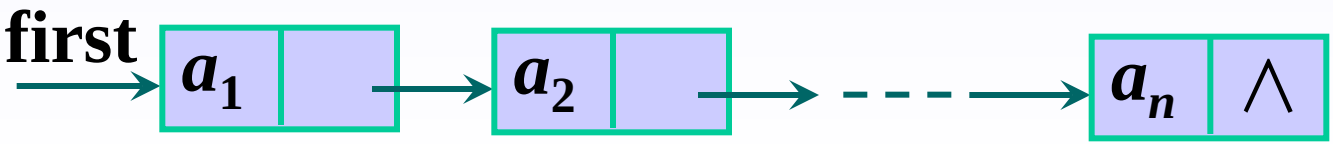
重点在数据元素之间的逻辑关系的表示，所以，将实际存储地址抽象。



空表

first=NULL

非空表



2.3 线性表的链接存储结构及实现

单链表

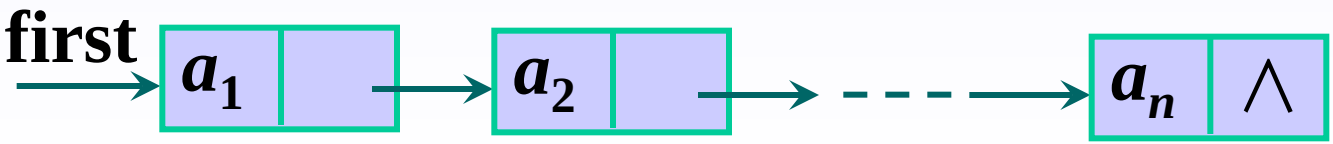
头指针：存储第一个结点的地址，即，指向第一个结点。

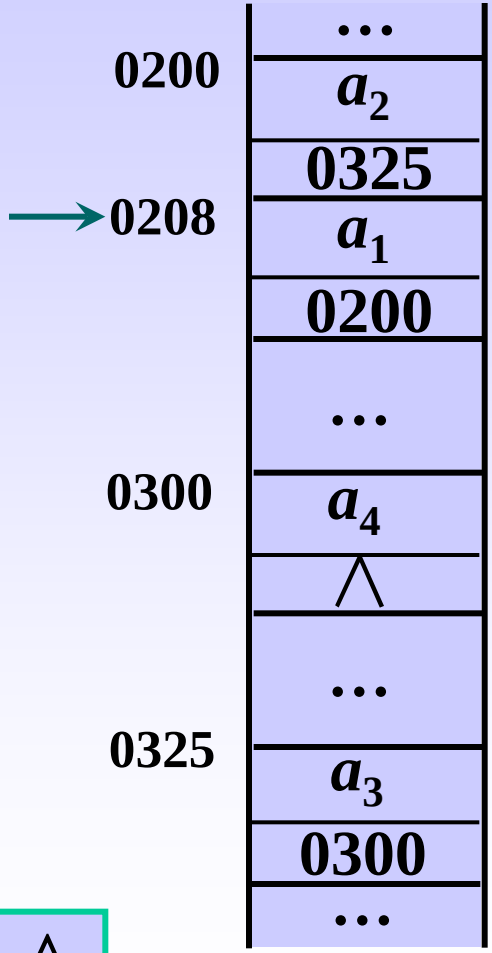
尾标志：终端结点的指针域为空。

空表

first=NULL

非空表

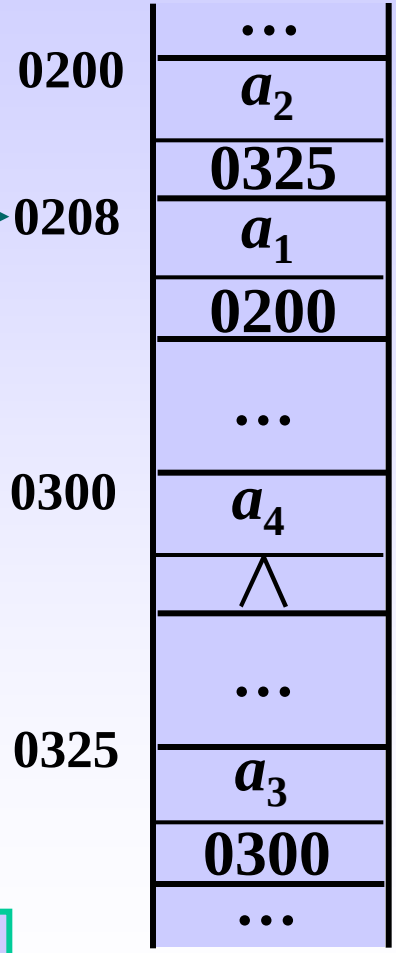




2.3 线性表的链接存储结构及实现

单链表

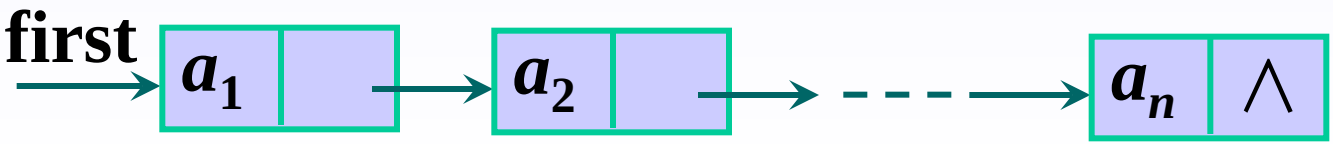
空表和非空表不统一，缺点？
如何将空表与非空表统一？



空表

first=NULL

非空表

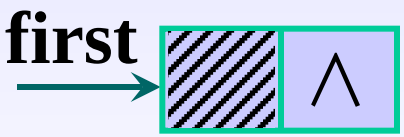


2.3 线性表的链接存储结构及实现

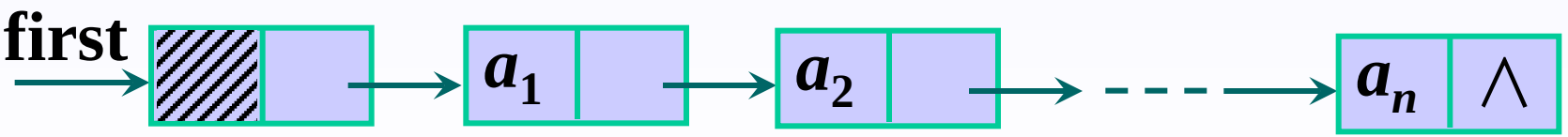
单链表

头结点：在单链表的第一个元素结点之前附设一个类型相同的结点，以便空表和非空表处理统一。

空表



非空表



2.3 线性表的链接存储结构及实现

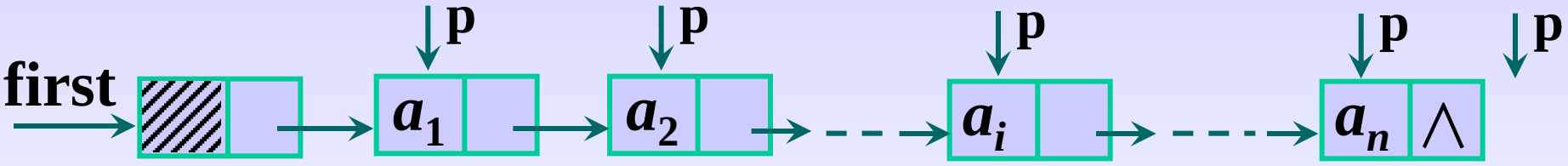
单链表类的声明

```
template <class T>
class LinkList
{
public:
    LinkList( );
    LinkList(T a[ ], int n);
    ~LinkList( );
    int Length( );
    T Get(int i);
    int Locate(T x);
    void Insert(int i, T x);
    T Delete(int i);
    void PrintList( );
private:
    Node<T> *first;
};
```

2.3 线性表的链接存储结构及实现

单链表的实现——遍历

操作接口： void PrintList();

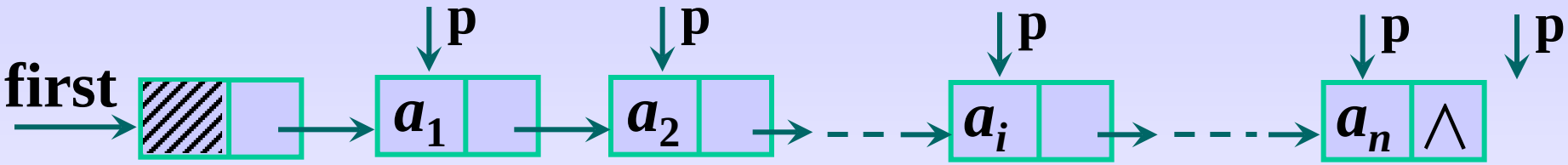


❖核心操作（关键操作）：工作指针后移。从头结点（或开始结点）出发，通过工作指针的反复后移而将整个单链表“审视”一遍的方法称为扫描（或遍历）。

2.3 线性表的链接存储结构及实现

单链表的实现——遍历

操作接口： void PrintList();

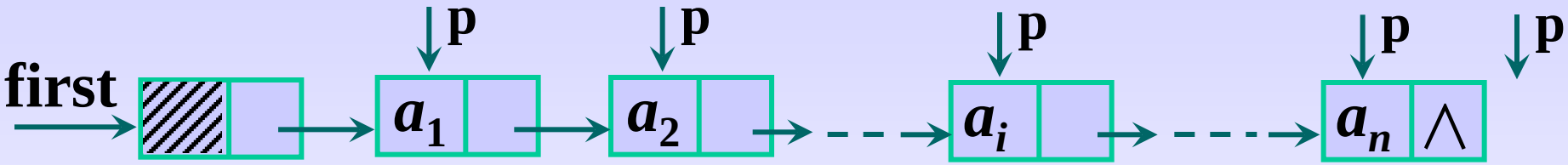



```
template <class T>
int LinkList<T>:: PrintList()
{
    p=first->next;
    while (p!=NULL)
    {
        cout<<p->data;
        p=p->next;
    }
}
```

2.3 线性表的链接存储结构及实现

单链表的实现——求线性表的长度

操作接口： int Length();

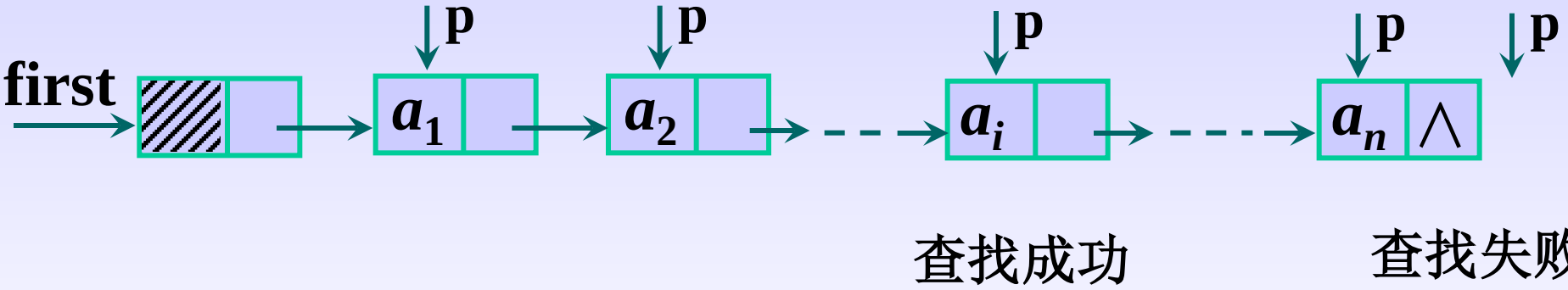



```
template <class T>
int LinkList<T>:: Length()
{
    p=first->next;          count=0;
    while (p!=NULL)
    {
        count++;
        p=p->next;
    }
return count;
}
```

2.3 线性表的链接存储结构及实现

单链表的实现——按位查找

操作接口： T Get(int i);



❖核心操作（关键操作）：工作指针后移——遍历

2.3 线性表的链接存储结构及实现

单链表的实现——按位查找

算法描述——伪代码

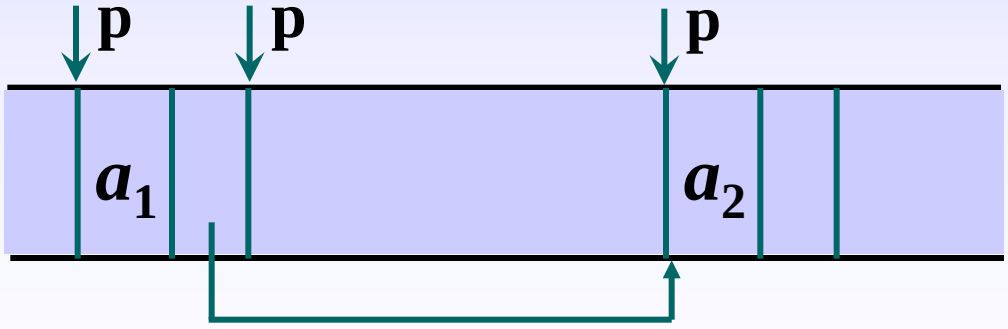
1. 工作指针p初始化; 累加器j初始化;
2. 循环直到p为空或p指向第i个结点
   2.1 工作指针p后移;
   2.2 累加器j加1;
3. 若p为空，则第i个元素不存在，抛出查找位置异常；否则查找成功，返回结点p的数据元素；

2.3 线性表的链接存储结构及实现

单链表的实现——按位查找

算法描述——C++描述

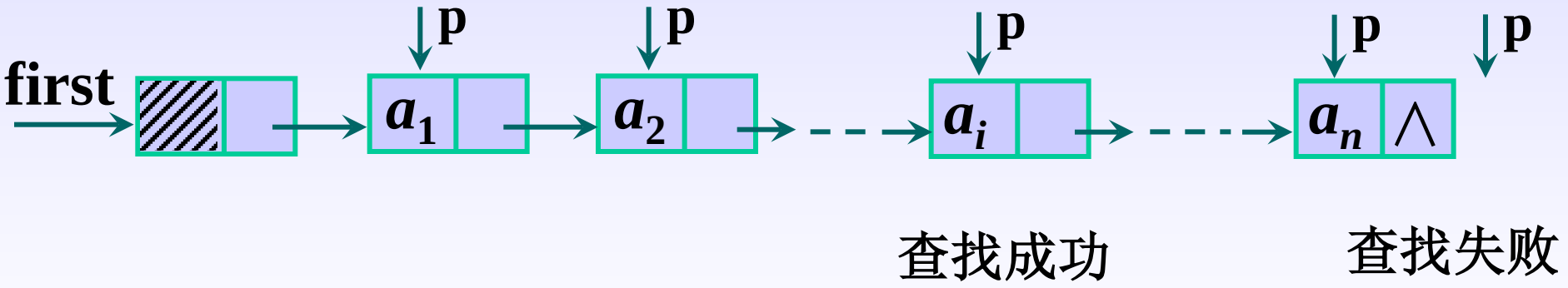
```
template <class T>
T LinkList<T>::Get(int i)
{
    p=first->next;  j=1;
    while (p!=NULL && j<i)
    {
        p=p->next;
        j++;
    }
    if (p==NULL)   throw "位置";
    else return p->data;
}
```



2.3 线性表的链接存储结构及实现

单链表的实现——按值查找

操作接口： **int Locate(T x)**

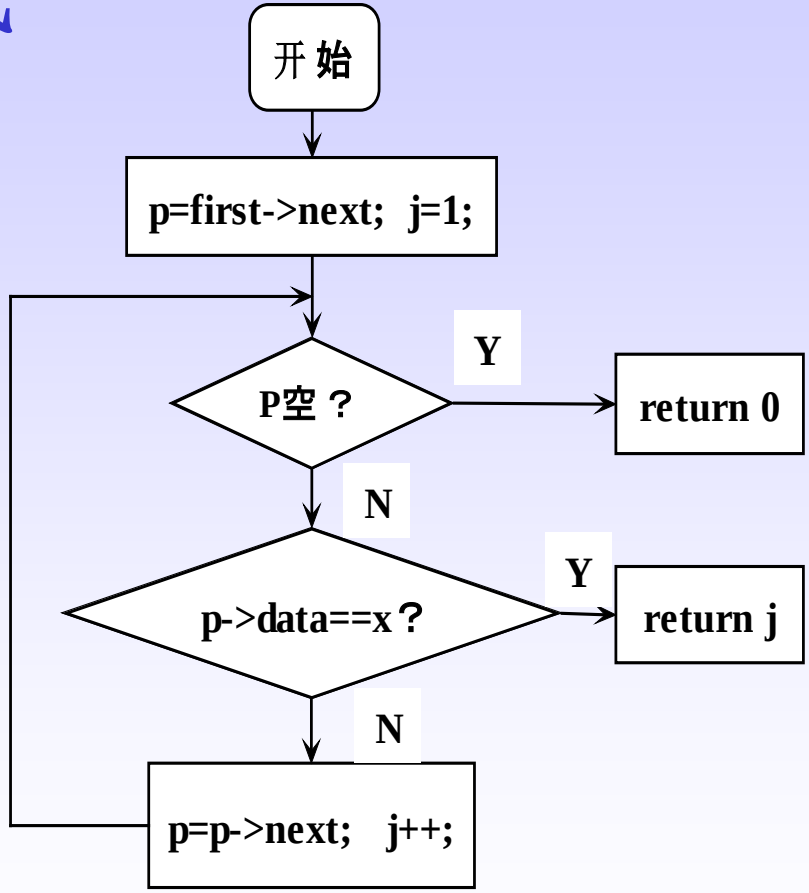


2.3 线性表的链接存储结构及实现

单链表的实现——按值查找

算法描述：

```
template <class T>
int LinkList<T>::Locate(T x)
{
        p=first->next; j=1;
        while (p){
          if(p->data==x)    return j;
          else
          { p=p->next;    j++; }
        }
        return 0;
}
```



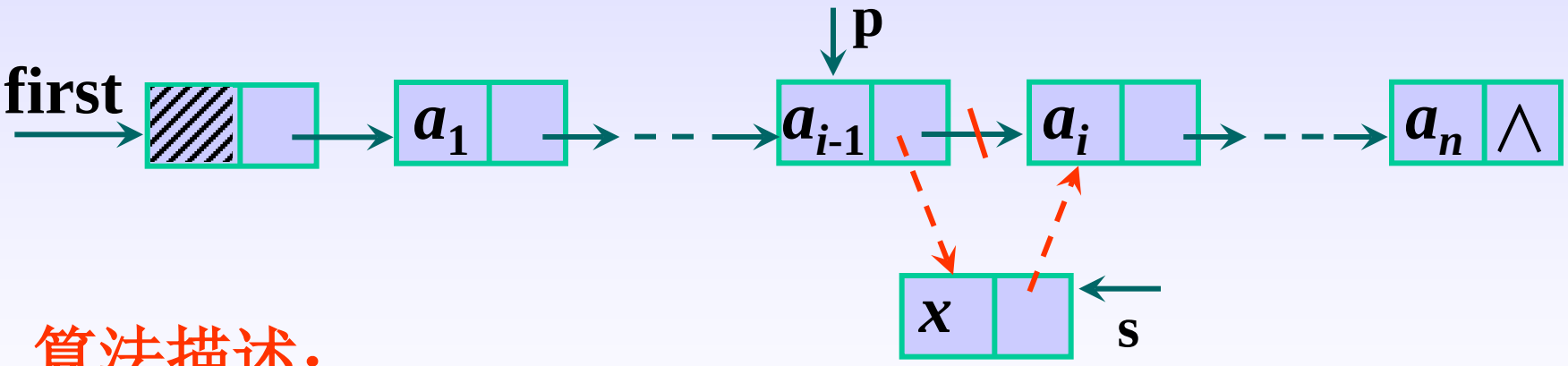
时间复杂度？

2.3 线性表的链接存储结构及实现

单链表的实现———插入

操作接口： void Insert(int i, T x);

如何实现结点a_{i-1}、x和a_i之间逻辑关系的变化？



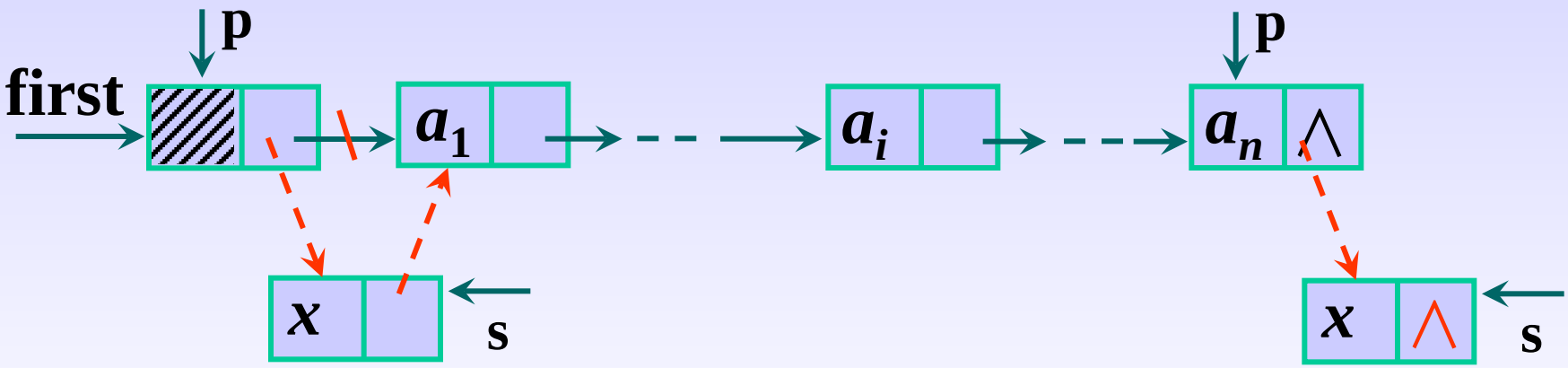
算法描述：

```
s=new Node<T>;  s->data=x;
s->next=p->next;  p->next=s;
```

2.3 线性表的链接存储结构及实现

单链表的实现———插入

注意分析边界情况——表头、表尾。



算法描述：

```
s=new Node<T>;  s->data=x;
s->next=p->next;  p->next=s;
```

由于单链表带头结点，表头、表中、表尾三种情况的操作语句一致。

2.3 线性表的链接存储结构及实现

单链表的实现———插入

算法描述——伪代码

1. 工作指针p初始化；累加器j清零；

2. 查找第i−1个结点并使工作指针p指向该结点；

3. 若查找不成功，说明插入位置不合理，抛出插入位置异常；否则，

3.1 生成一个元素值为x的新结点s；

3.2 将新结点s插入到结点p之后；

2.3 线性表的链接存储结构及实现

单链表的实现———插入

算法描述——C++描述

```
template <class T>
void LinkList<T>::Insert(int i, T x)
{
    p=first ;   j=0;
    while（p!=NULL && j<i−1）
    {
        p=p−>next;
        j++;
    }
if (p==NULL) throw "位置";
else {
        s=new Node<T>;
        s->data=x;
        s->next=p->next;
        p->next=s;
    }
}
```

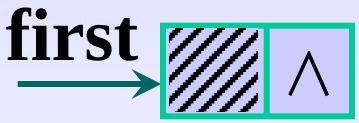
基本语句？时间复杂度？

2.3 线性表的链接存储结构及实现

单链表的实现———构造函数

操作接口：LinkList()　——无参构造函数

初始化



算法描述：

```
first=new Node<T>;
first->next=NULL;
```

```
template <class T>
LinkList<T>:: LinkList()
{
        first=new Node<T>;
        first->next=NULL;
}
```

2.3 线性表的链接存储结构及实现

单链表的实现———构造函数

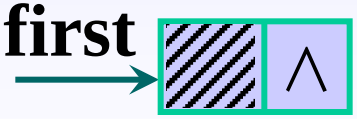
操作接口：**LinkList(T a[ ], int n)**

头插法：将待插入结点插在头结点的后面 。

数组*a*

35	12	24	33	42

初始化



算法描述：

```
first=new Node<T>;
first->next=NULL;
```

2.3 线性表的链接存储结构及实现

单链表的实现———构造函数

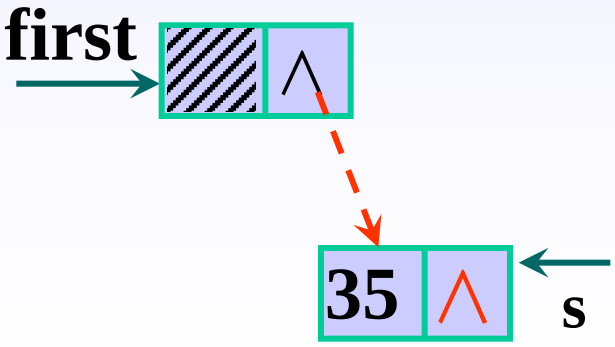
操作接口：**LinkList(T a[ ], int n)**

头插法：将待插入结点插在头结点的后面 。

数组*a*

35	12	24	33	42

插入第一个元素结点



算法描述：

```
s=new Node<T>;
s->data=a[0];
s->next=first->next;
first->next=s;
```

2.3 线性表的链接存储结构及实现

单链表的实现———构造函数

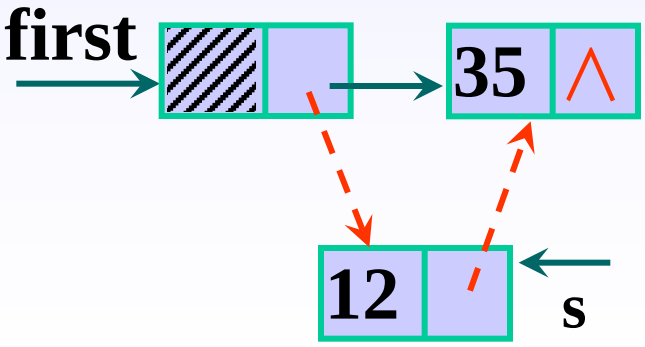
操作接口：**LinkList(T a[ ], int n)**

头插法：将待插入结点插在头结点的后面 。

数组*a*

35	12	24	33	42

依次插入每一个结点



算法描述：

```
s=new Node<T>;
s->data=a[1];
s->next=first->next;
first->next=s;
```

2.3 线性表的链接存储结构及实现

单链表的实现———构造函数

算法描述：

```
template <class T>
LinkList<T>:: LinkList(T a[ ], int n)
{
    first=new Node<T>;
    first->next=NULL;
    for (i=0; i<n; i++)
    {
        s=new Node<T>;
        s->data=a[i];
        s->next=first->next;
        first->next=s;
    }
}
```

2.3 线性表的链接存储结构及实现

单链表的实现———构造函数

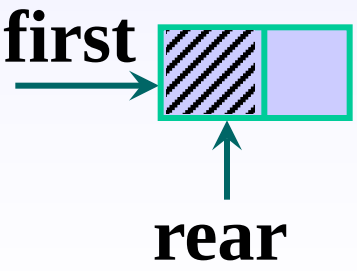
操作接口：**LinkList(T a[ ], int n)**

尾插法：将待插入结点插在终端结点的后面。

数组*a*

35	12	24	33	42

初始化



算法描述：

```
first=new Node<T>;
rear=first;
```

2.3 线性表的链接存储结构及实现

单链表的实现———构造函数

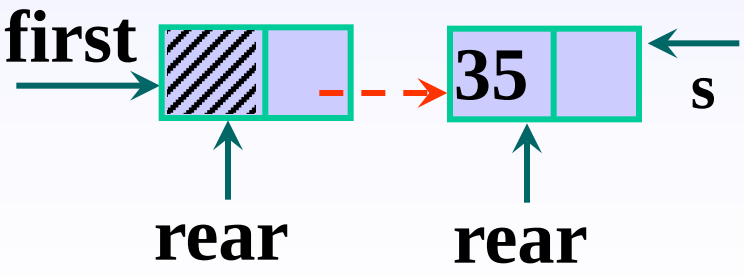
操作接口：LinkList(T a[ ], int n)

尾插法：将待插入结点插在终端结点的后面。

数组*a*

35	12	24	33	42

插入第一个元素结点



算法描述：

```
s=new Node<T>;
s->data=a[0];
rear->next=s;
rear=s;
```

2.3 线性表的链接存储结构及实现

单链表的实现———构造函数

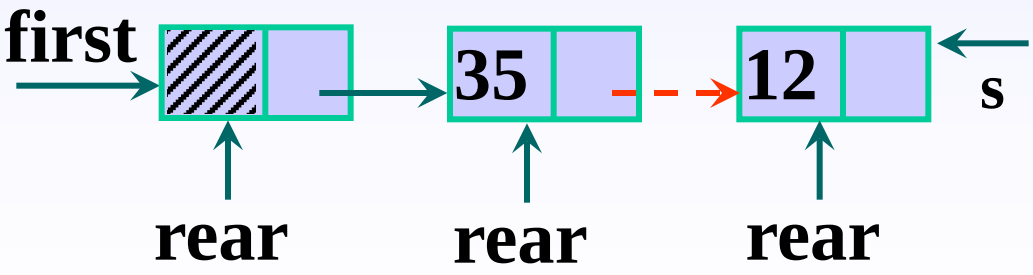
操作接口：LinkList(T a[ ], int n)

尾插法：将待插入结点插在终端结点的后面。

数组*a*

35	12	24	33	42

依次插入每一个结点



算法描述：

```
s=new Node<T>;
s->data=a[1];
rear->next=s;
rear=s;
```

2.3 线性表的链接存储结构及实现

单链表的实现———构造函数

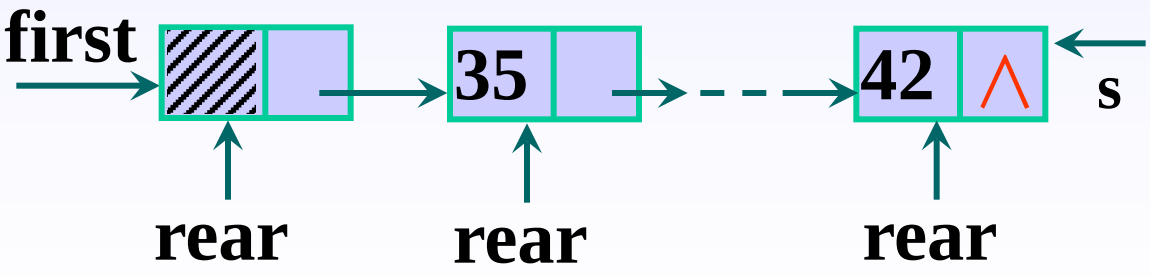
操作接口：**LinkList(T a[ ], int n)**

尾插法：将待插入结点插在终端结点的后面。

数组*a*

35	12	24	33	42

最后一个结点插入后



算法描述：

rear->next=NULL;

2.3 线性表的链接存储结构及实现

单链表的实现———构造函数

算法描述：

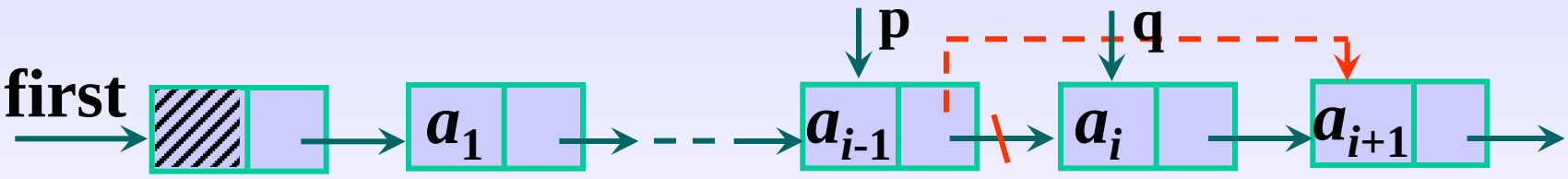
```
template <class T>
LinkList<T>:: LinkList（T a[ ], int n）
{
    first=new Node<T> ;   rear=first;
    for（i=0; i<n; i++）
    {
        s=new Node<T> ;
        s->data=a[i];
        rear->next=s;
        rear=s;
    }
    rear->next=NULL;
}
```

2.3 线性表的链接存储结构及实现

单链表的实现———删除

操作接口： T Delete(int i);

如何实现结点a_{i-1}和a_i之间逻辑关系的变化？



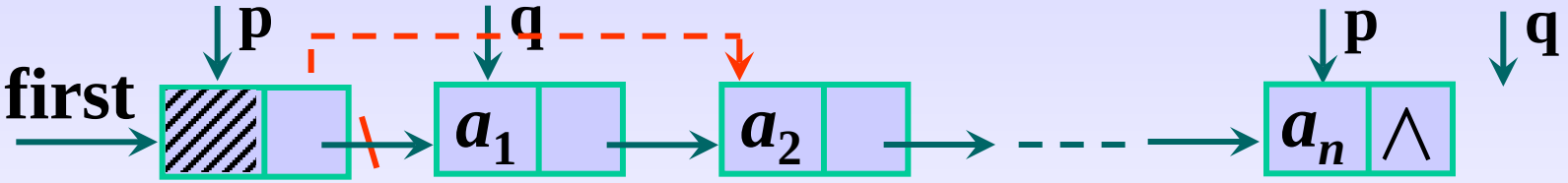
算法描述：

```
q=p->next;  x=q->data;
p->next=q->next; delete q;
```

2.3 线性表的链接存储结构及实现

单链表的实现———删除

注意分析边界情况——表头、表尾。



p->next=NULL

算法描述：

q=p->next; x=q->data;
p->next=q->next; delete q;

表尾的特殊情况：

虽然被删结点不存在，
但其前驱结点却存在。

2.3 线性表的链接存储结构及实现

单链表的实现———删除

算法描述——伪代码

1.工作指针p初始化； 累加器j清零；

2. 查找第i-1个结点并使工作指针p指向该结点；

3. 若p不存在或p不存在后继结点，则抛出位置异常；
否则，

3.1 暂存被删结点和被删元素值；

3.2 摘链，将结点p的后继结点从链表上摘下；

3.3 释放被删结点；

3.4 返回被删元素值；

2.3 线性表的链接存储结构及实现

单链表的实现———删除

算法描述——C++描述

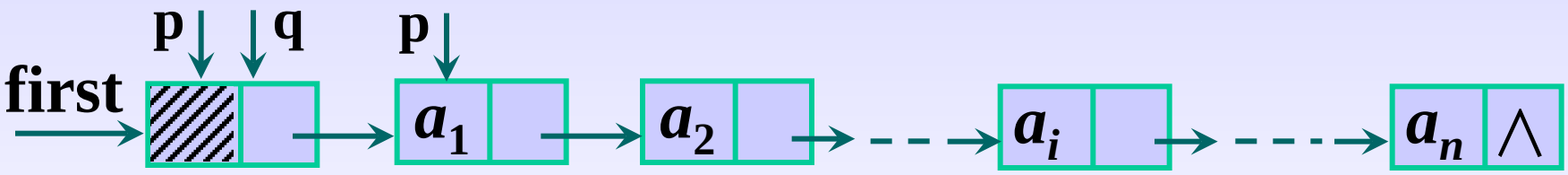
```
template <class T>
T LinkList<T>::Delete(int i)
{
    p=first ; j=0;
    while (p && j<i-1)
    {
        p=p->next;
        j++;
     }
if (p==NULL || p->next ==NULL )
        throw “位置”;
else {
        q=p->next;
        x=q->data;
        p->next=q->next;
        delete q;
        return x;
    }
 }
```

2.3 线性表的链接存储结构及实现

单链表的实现——析构函数

操作接口：~LinkList( );

析构函数将单链表中所有结点的存储空间释放。



算法描述：

注意：保证链表未处理的部分不断开

```
q=p;
p=p->next;
Delete q;
```

2.3 线性表的链接存储结构及实现

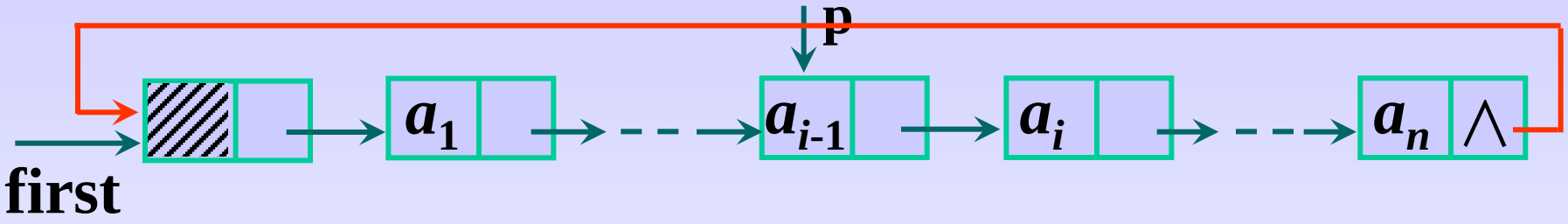
单链表的实现——析构函数

算法描述：

```
template <class T>
LinkList<T>:: ~LinkList（ ）
{
    p=first;
    while（p!=NULL）
    {
        q=p;
        p=p->next;
        delete q;
    }
}
```

2.3 线性表的链接存储结构及实现

循环链表



从单链表中某结点p出发如何找到其前驱？

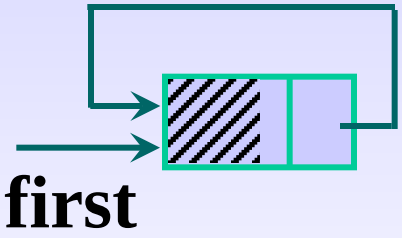
将单链表的首尾相接，将终端结点的指针域由空指针改为指向头结点，构成单循环链表，简称循环链表。

2.3 线性表的链接存储结构及实现

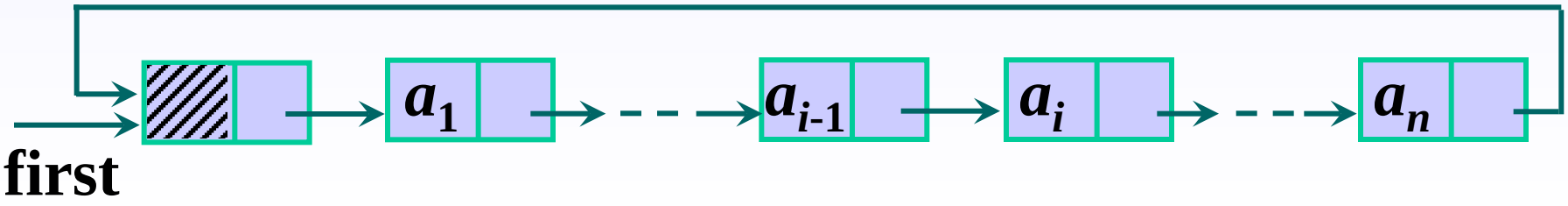
循环链表

空表和非空表的处理一致 ➡ 附设头结点

空循环链表

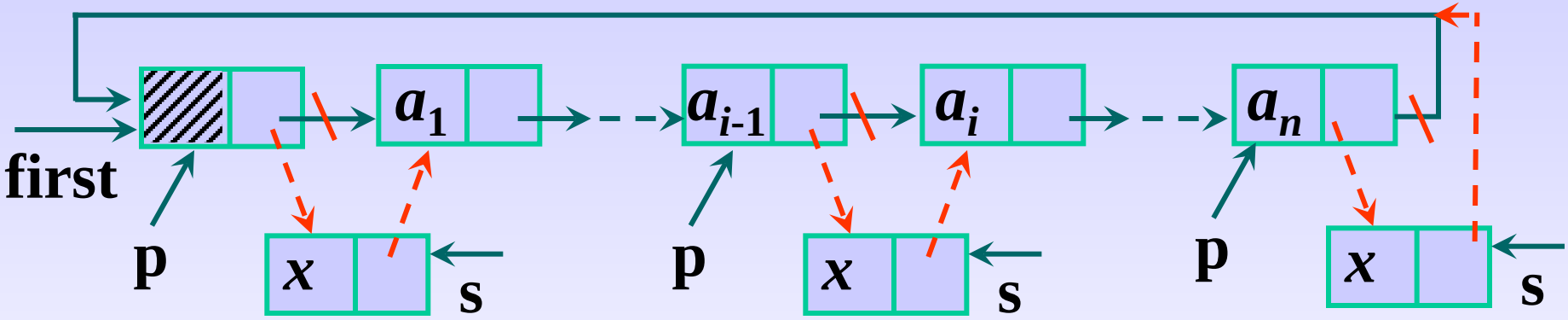


非空循环链表



2.3 线性表的链接存储结构及实现

循环链表——插入



算法描述：

```
s=new Node<T>;
s->data=x;
s->next=p->next;
p->next=s;
```

2.3 线性表的链接存储结构及实现

循环链表——插入

```
template <class T>
void LinkList<T>::Insert(int i, T x)
{
  p=first ; j=0;    //工作指针p初始化
  while (p->next!=first && j<i-1)
  {
    p=p->next;   //工作指针p后移
    j++;
  }
    if (j<i-1) throw "位置";
    else {
          s=new Node<T>;
          s->data=x;
          s->next=p->next;
          p->next=s;
          }
}
```

与单链表的插入操作相比，差别是什么？

2.3 线性表的链接存储结构及实现

循环链表

循环链表中没有明显的尾端 ➡ 如何避免死循环

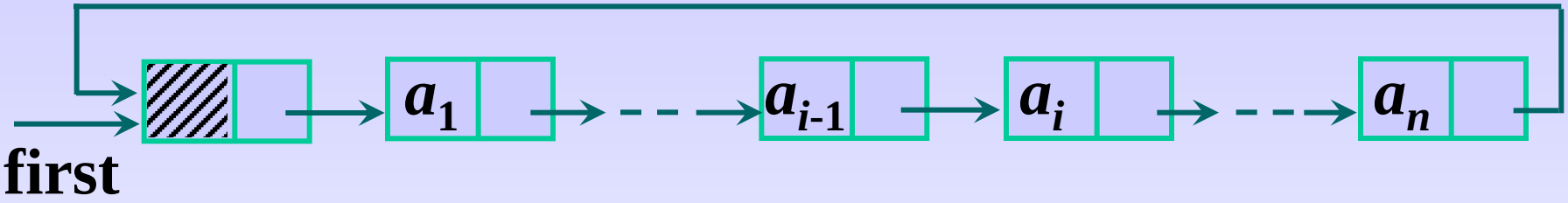
循环条件：

p!=NULL➔p!=first

p->next!=NULL➔p->next!=first

2.3 线性表的链接存储结构及实现

循环链表



如何查找开始结点和终端结点？

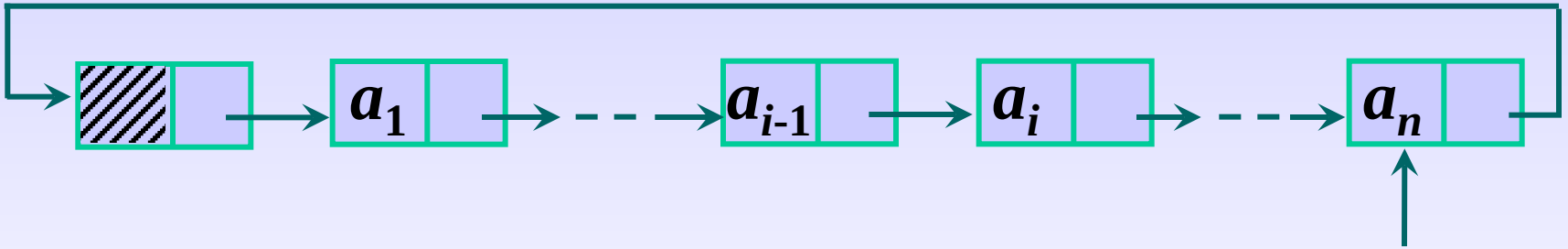
开始结点：**first->next**

终端结点：将单链表扫描一遍，时间为$O(n)$

2.3 线性表的链接存储结构及实现

循环链表

带尾指针的循环链表



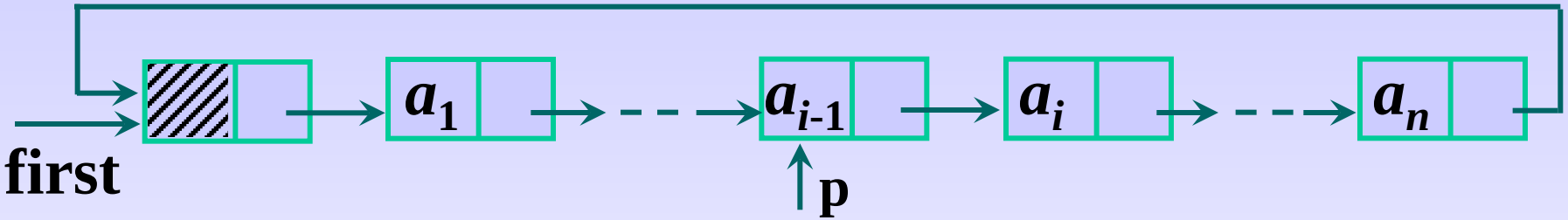
开始结点：rear->next->next

终端结点：rear

一个存储结构设计得是否合理，取决于基于该存储结构的运算是否方便，时间性能是否提高。

2.3 线性表的链接存储结构及实现

双链表



如何求结点p的直接前驱，时间性能？

为什么可以快速求得结点p的后继？

如何快速求得结点p的前驱？

2.3 线性表的链接存储结构及实现

双链表

双链表：在单链表的每个结点中再设置一个指向其前驱结点的指针域。

结点结构：

data：数据域，存储数据元素；
prior：指针域，存储该结点的前趋结点地址；
next：指针域，存储该结点的后继结点地址。

2.3 线性表的链接存储结构及实现

双链表

定义结点结构：


```
template <class T>
struct DulNode
{
    T data;
    DulNode<T> *prior, *next;
};
```

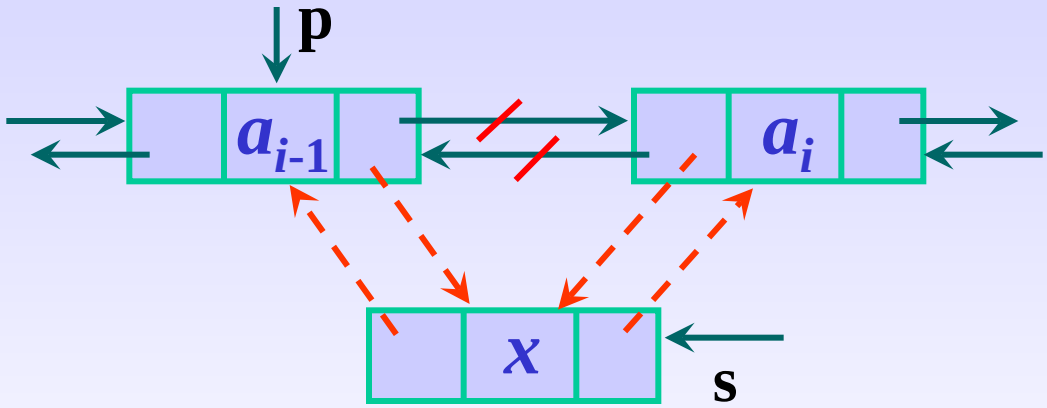
启示？

时空权衡——空间换取时间

2.3 线性表的链接存储结构及实现

双链表的操作——插入

操作接口： void Insert(DulNode<T> *p, T x);



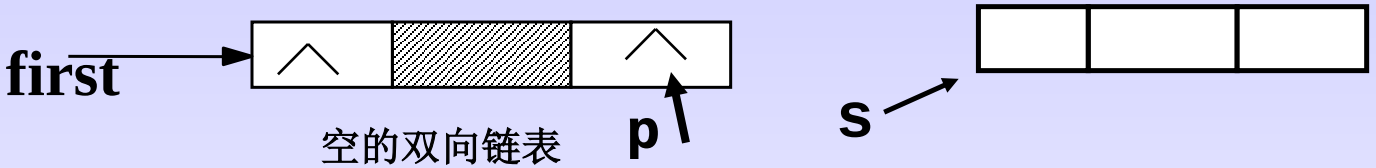

```
s->prior=p;
s->next=p->next;
p->next->prior=s;
p->next=s;
```

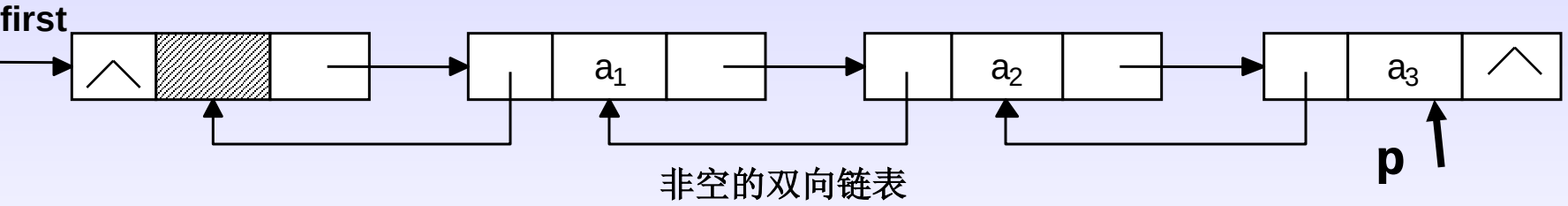
注意指针修改的相对顺序

2.3 线性表的链接存储结构及实现

双链表的操作——插入

特殊情况:在空表中插入一个结点（在表尾插入一个结点）





s->prior=p;

s->next=p->next;

p->next->prior=s;

p->next=s;

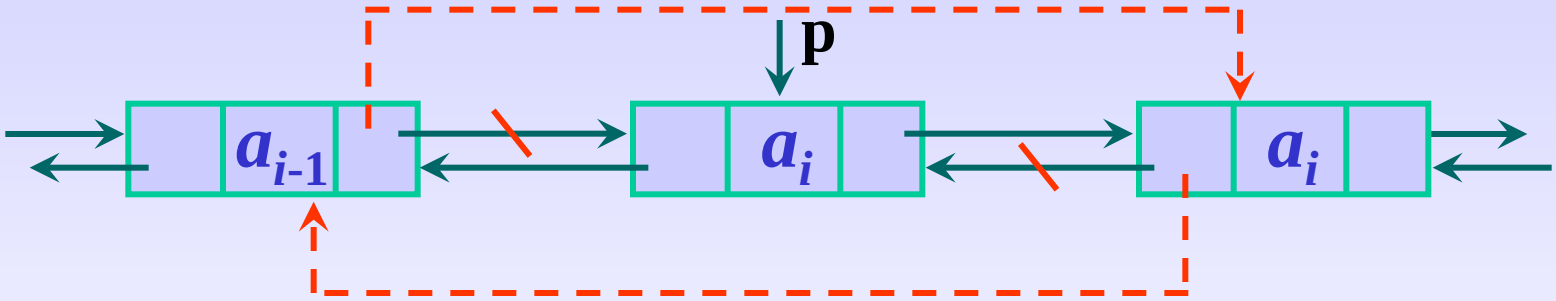
统一插入操作：

```
s->prior=p;
s->next=p->next;
if(p->next!=NULL)
    p->next->prior=s;
p->next=s;
```

2.3 线性表的链接存储结构及实现

双链表的操作——删除

操作接口： T Delete(DulNode<T> *p);



（p->prior）->next=p->next;

（p->next）->prior=p->prior;

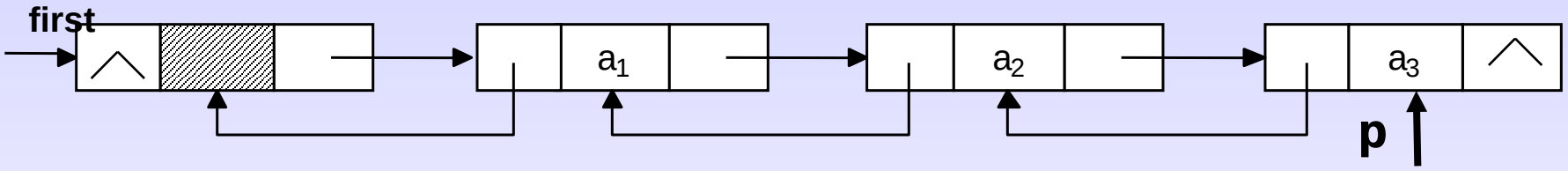
结点p的指针是否需要修改？

delete p;

2.3 线性表的链接存储结构及实现

双链表的操作——删除

特殊情况:删除表尾结点




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

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

delete p;
```

统一删除操作:

```
(p->prior)->next=p->next;
if(p->next!=NULL)
    (p->next)->prior=p->prior;
delete p;
```

2.3 线性表的链接存储结构及实现

双链表类的定义

```
template <class T>
struct DulNode
{
  T data;
  DulNode<T> *prior;
  DulNode<T> *next;
};
```

```
template <class T>
class DoubleLink
{
private:
    DulNode<T> *first;
public:
    DoubleLink() ;
    ~DoubleLink();
    void Append(T data);
    void Display();
    void Insert(int locate , T data);
    T Get(int locate);
    T Delete(int locate);
};
```

2.3 线性表的链接存储结构及实现

双链表的构造-空表的构造 DoubleLink()；

```
template <class T>
DoubleLink <T>::DoubleLink(){
        first =new DulNode<T>;
        first ->prior=NULL;
        first ->next=NULL;
   }
```

2.3 线性表的链接存储结构及实现

追加（头插）：void Append(T data);

```
template <class T>
void DoubleLink<T>::Append(T data){
    DulNode<T> *s;
    s=new DulNode<T>;
    if (s！=NULL)     {
          s->data=data;
          s->next= first ->next;
           first ->next=s;
          s->prior= first;
          if (s->next！=NULL)          s->next->prior=s;
    }
    else throw "内存空间不足";
    return;
}
```

2.3 线性表的链接存储结构及实现

遍历 void Display();

```
template <class T>
void DoubleLink<T>::Display(){
    DulNode <T> *p;
    p=first->next;
    while(p ！ =NULL)    {
        cout<<p->data<<"   ";
        p=p->next;
    }
    cout<<endl;
}
```

2.3 线性表的链接存储结构及实现

析构 ~DoubleLink();

```
template <class T>
DoubleLink<T>::~DoubleLink()
{
    DulNode<T>  *p,*q;
    p=first;
    while(p！ =NULL)
    {
        q=p->next;
        delete p;
        p=q;
    }
}
```

2.4 顺序表和链表的比较

存储分配方式比较

- 顺序表采用顺序存储结构，即用一段地址连续的存储单元依次存储线性表的数据元素，数据元素之间的逻辑关系通过存储位置来实现。
- 单链表采用链接存储结构，即用一组任意的存储单元存放线性表的元素。用指针来反映数据元素之间的逻辑关系。

2.4 顺序表和单链表的比较

时间性能比较

时间性能是指实现基于某种存储结构的基本操作（即算法）的时间复杂度。

按位查找：

- 顺序表的时间为 $O(1)$，是随机存取；
- 单链表的时间为 $O(n)$，是顺序存取。

插入和删除：

- 顺序表需移动表长一半的元素，时间为 $O(n)$；
- 单链表不需要移动元素，在给出某个合适位置的指针后，插入和删除操作所需的时间仅为 $O(1)$。

2.4 顺序表和单链表的比较

空间性能比较

空间性能是指某种存储结构所占用的存储空间的大小。

定义结点的存储密度：

$$\text{存储密度} = \frac{\text{数据域占用的存储量}}{\text{整个结点占用的存储量}}$$

单链表中，如果数据域是整型变量int

则单链表结点的存储密度是：

$$\frac{4}{4+4} = 50\%$$

2.4 顺序表和单链表的比较

空间性能比较

结点的存储密度：

- 顺序表中每个结点的存储密度为1（只存储数据元素），没有浪费空间；
- 单链表的每个结点的存储密度<1（包括数据域和指针域），有指针的结构性开销。

整体结构：

- 顺序表需要预分配存储空间，如果预分配得过大，造成浪费，若估计得过小，又将发生上溢；
- 单链表不需要预分配空间，只要有内存空间可以分配，单链表中的元素个数就没有限制。

2.4 顺序表和单链表的比较

结论

⑴若线性表需：

•查找多，插入和删除少，或

•操作和元素在表中的位置密切相关

宜采用顺序表作为存储结构；

若线性表需频繁插入和删除时，则宜采用单链表做存储结构。

⑵若线性表中元素个数变化较大或未知时，最好使用单链表实现；

如果事先知道线性表的大致长度，使用顺序表的空间效率会更高。

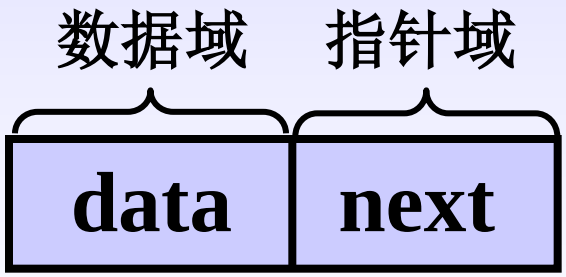
总之，线性表的顺序实现和链表实现各有其优缺点，不能笼统地说哪种实现更好，只能根据实际问题的具体需要，并对各方面的优缺点加以综合平衡，才能最终选定比较适宜的实现方法。

2.5 线性表的其它存储方法

静态链表

静态链表：用数组来表示单链表，用数组元素的下标来模拟单链表的指针。

数组元素（结点）的构成：



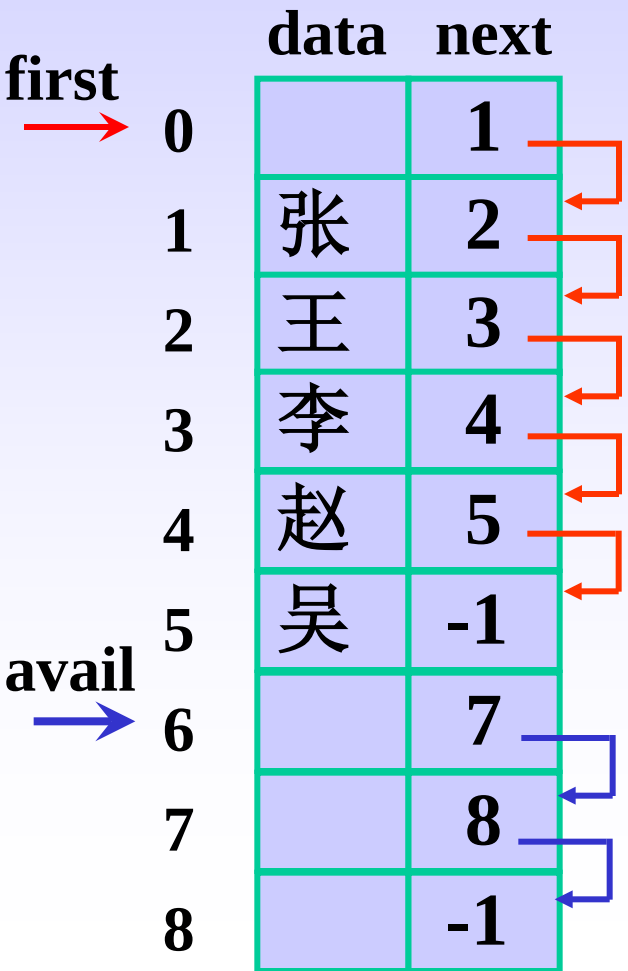
data：存储放数据元素；

next：也称游标，存储该元素的后继在数组的下标。

2.5 线性表的其它存储方法

静态链表

例：线性表(张，王，李，赵，吴)的静态链表存储



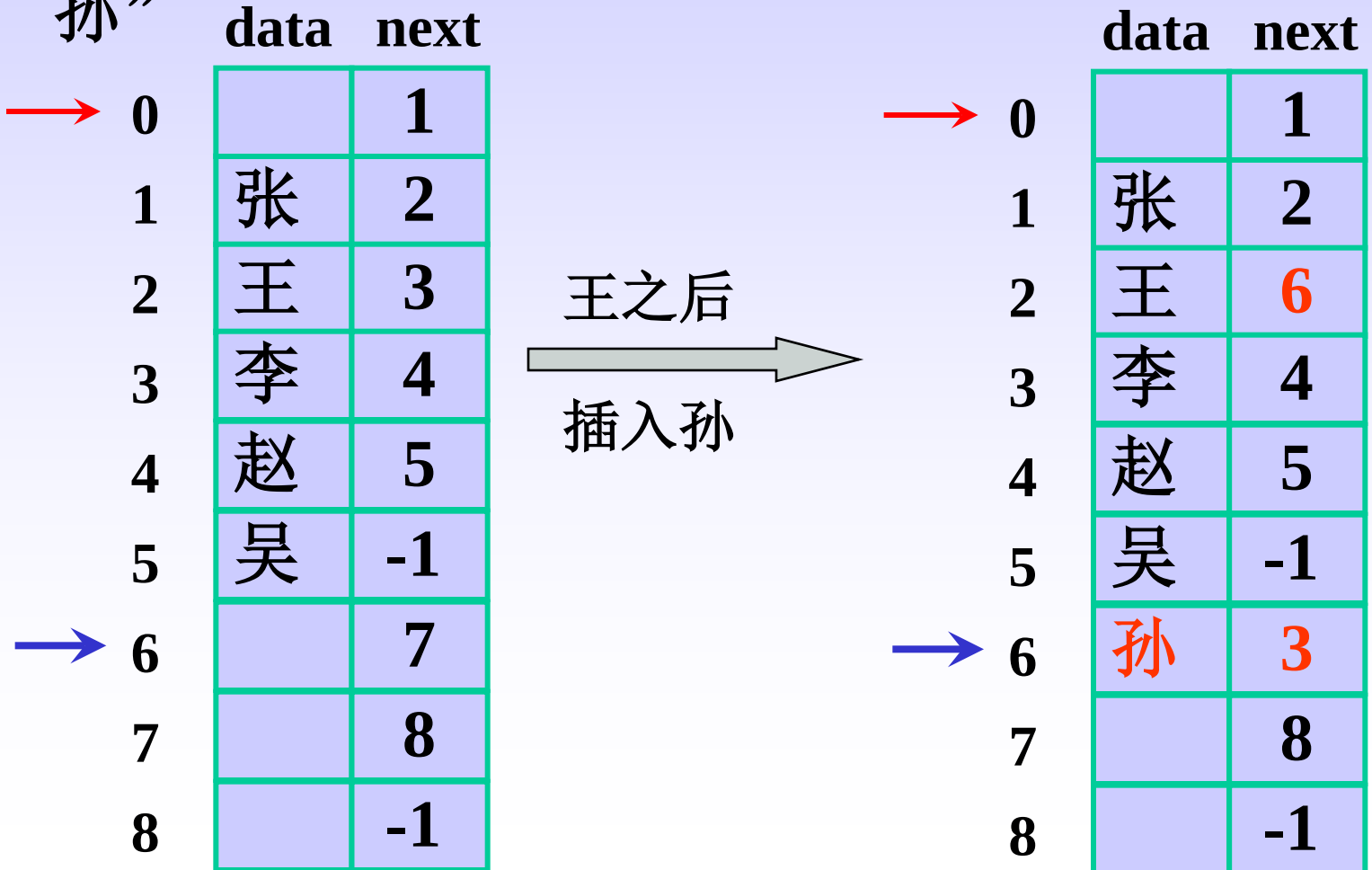
first：静态链表头指针，为了方便插入和删除操作，通常静态链表带头结点；

avail：空闲链表头指针，空闲链表由于只在表头操作，所以不带头结点；

2.5 线性表的其它存储方法

静态链表

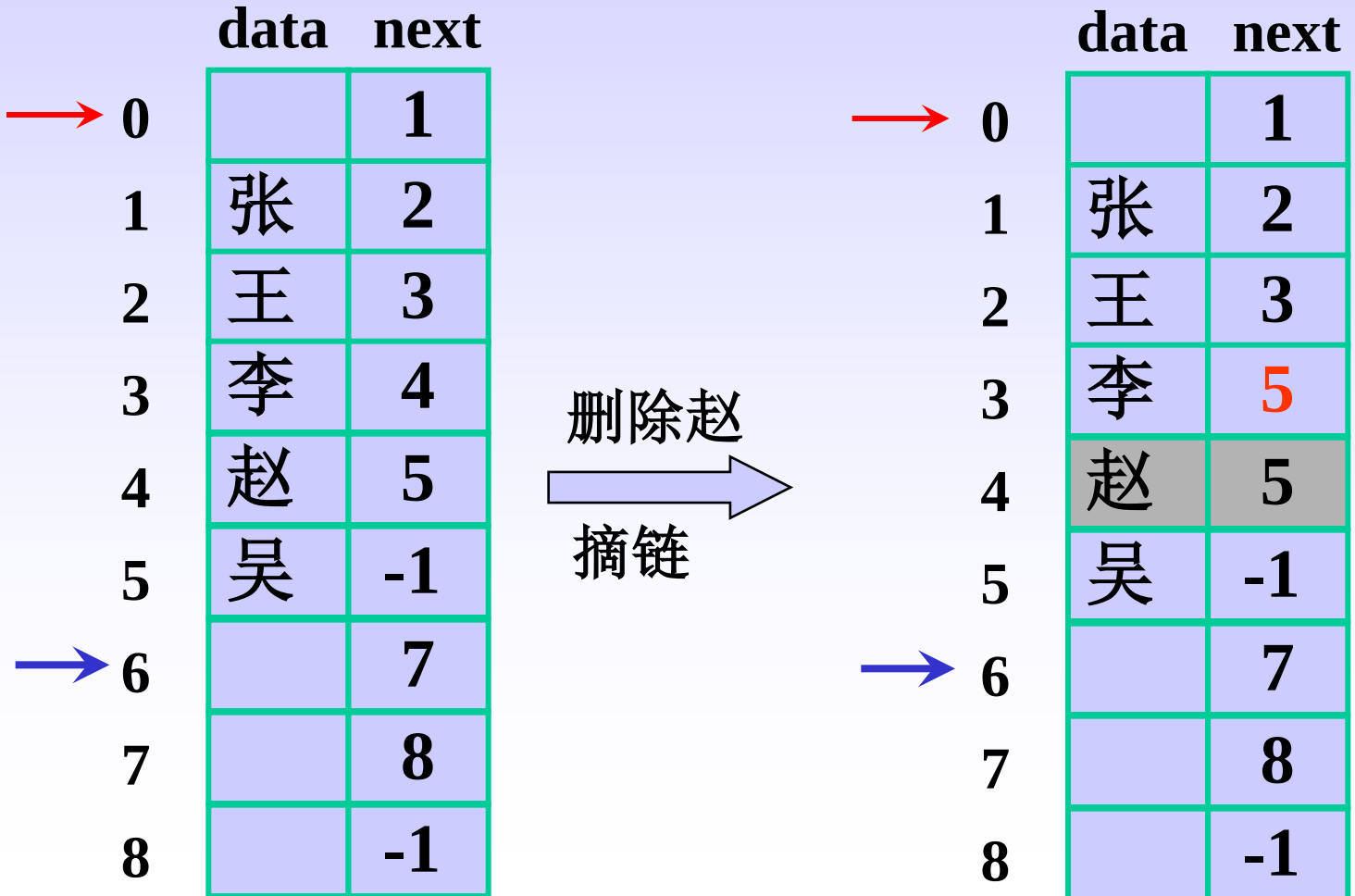
在线性表(张，王，李，赵，吴)中“王”之后插入“孙”



2.5 线性表的其它存储方法

静态链表

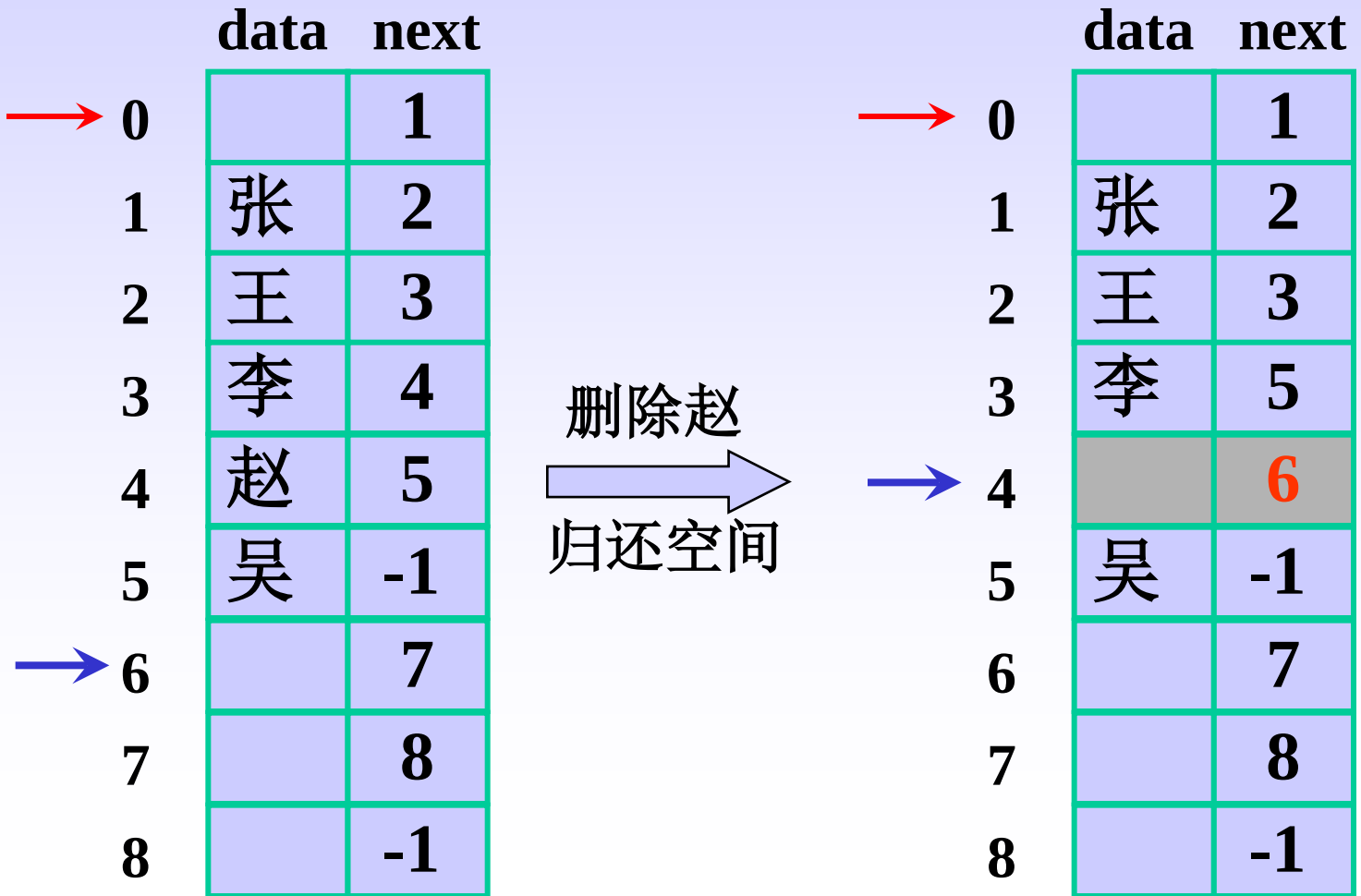
在线性表(张，王，李，赵，吴)中删除“赵”



2.5 线性表的其它存储方法

静态链表

在线性表(张，王，李，赵，吴)中删除“赵”



2.5 线性表的其它存储方法

静态链表

相对于顺序表而言，静态链表有什么优点？

优点：在执行插入和删除操作时，只需修改游标，不需要移动表中的元素，从而改进了在顺序表中插入和删除操作需要移动大量元素的缺点。

缺点：

没有解决连续存储分配带来的表长难以确定的问题；

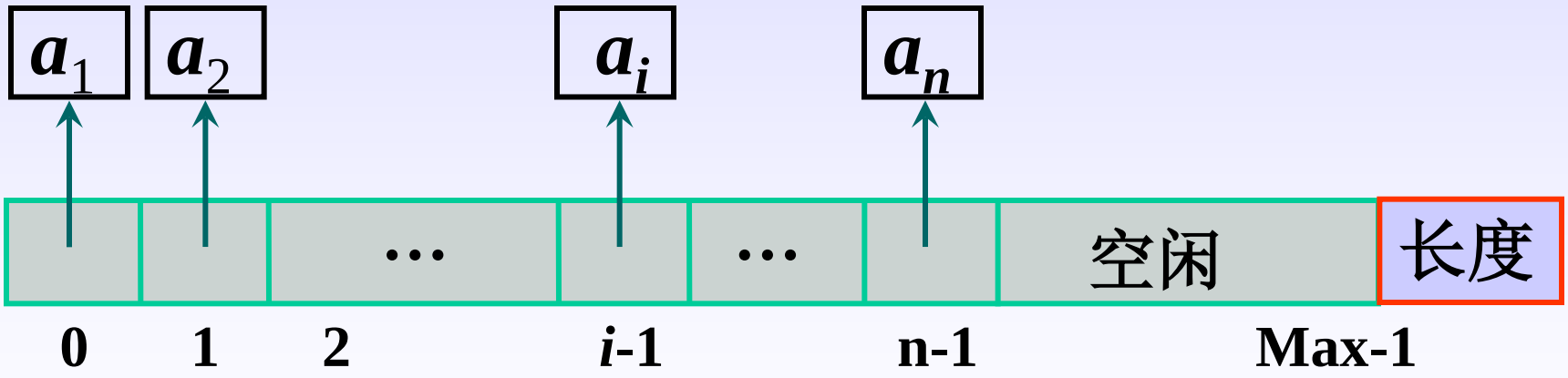
静态链表还需要维护一个空闲链；

静态链表不能随机存取。

2.5 线性表的其它存储方法

间接寻址

间接寻址：是将数组和指针结合起来的一种方法，它将数组中存储数据元素的单元改为存储指向该元素的指针 。



插入操作移动的不是元素而是指向元素的指针。当元素占用的空间较多时，比顺序表的插入操作快得多。

2.6 应用举例

2.6.1 顺序表应用举例——大整数求和

问题描述：

在用某种程序设计语言编程时，可能需要处理非常大或对运算精度要求非常高的整数，

这种大整数用该语言的基本数据类型无法表示。

例如，C++ 的长整型数的范围为

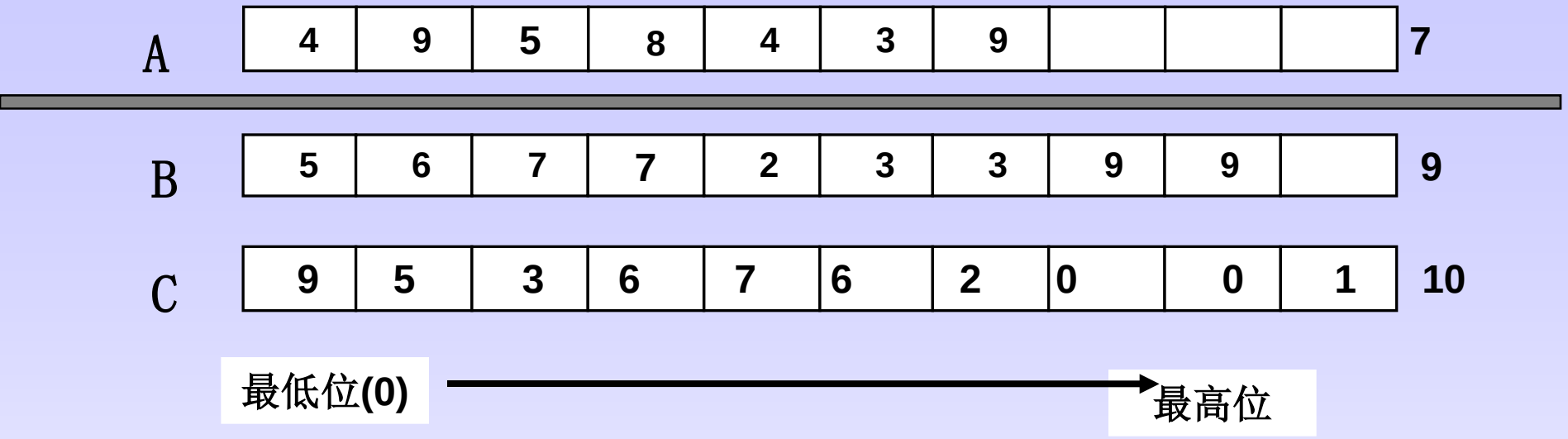
-2147483648~2147483647

超过该范围的整数无法表示

怎么办？

借助顺序表解决。

Eg.9348594+993327765



1. 设计标志位flag（int,初值为0）,用来记录计算过程中是否有进位
2. 从数组低端开始，将对应位置上的两个数及flag的值相加，取结果的个位数，存入的c的对应位置上，将10位数上的值赋值给flag
3. 重复进行计算，直到A或B计算完毕
4. 处理A或B中剩余的部分
5. 计算结果的位数

1. 设计标志位flag（int,初值为0）,用来记录计算过程中是否有进位
2. 从数组低端开始，将对应位置上的两个数及flag的值相加，取结果的个位数，存入的c的对应位置上，将10位数上的值赋值给flag
3. 重复进行计算，直到A或B计算完毕
4. 处理A或B中剩余的部分
5. 计算结果的位数

```
SeqList<int> Add(SeqList<int> A,SeqList<int>B){//友元函数实现
    int flag=0,i=0,m,n;
    SeqList<int> C;
    m=B.Length () ;
    n=A.Length() ;
    while(i<n && i<m)  {
        C.data[i]=(A.data[i]+B.data [i]+flag)%10;
        flag=(A.data[i]+B.data [i]+flag)/10;
        i++;
    }
```

1. 设计标志位flag（int,初值为0）,用来记录计算过程中是否有进位
2. 从数组低端开始，将对应位置上的两个数及flag的值相加，取结果的个位数，存入的c的对应位置上，将10位数上的值赋值给flag
3. 重复进行计算，直到A或B计算完毕
4. 处理A或B中剩余的部分
5. 计算结果的位数

```
for(;i<n;i++) {
        C.data[i]=(A.data[i]+flag)%10;flag=(A.data[i]+flag)/10;
}
for(;i<m;i++){
        C.data[i]=(B.data[i]+flag)%10;  flag=(B.data[i]+flag)/10;
}
int max;        max=n;
if(m>n)         max=m;
C.length =max+flag;             //如果最后一位产生进位
if(flag==1)     C.data [C.length-1 ]=1;
return C;
}
```

1. 设计标志位flag（int,初值为0）,用来记录计算过程中是否有进位
2. 从数组低端开始，将对应位置上的两个数及flag的值相加，取结果的个位数，存入的c的对应位置上，将10位数上的值赋值给flag
3. 重复进行计算，直到A或B计算完毕
4. 处理A或B中剩余的部分
5. 计算结果的位数

```
SeqList<int> Add1(SeqList<int> A,SeqList<int>B){   //非友元函数实现
    int flag=0,i=1,m,n;
    SeqList<int> C;
    m=B.Length () ;
    n=A.Length() ;
    while(i<=n && i<=m)        {
      C.Insert(i,(A.Get(i)+B.Get(i)+flag)%10);
      flag=(A.Get(i)+B.Get(i)+flag)/10;
      i++;
    }
```

1. 设计标志位flag（int,初值为0）,用来记录计算过程中是否有进位
2. 从数组低端开始，将对应位置上的两个数及flag的值相加，取结果的个位数，存入的c的对应位置上，将10位数上的值赋值给flag
3. 重复进行计算，直到A或B计算完毕
4. 处理A或B中剩余的部分
5. 计算结果的位数

```
for(;i<=n;i++) {
    C.Insert(i,(A.Get(i)+flag)%10);flag=(A.Get(i)+flag)/10;
}
for(;i<=m;i++){
    C.Insert(i,(B.Get(i)+flag)%10);  flag=(B.Get(i)+flag)/10;
}
int max;          max=n;
if(m>n)           max=m;
if(flag==1)       C.Insert(max+flag,1);
return C;
}
```

2.6 应用举例

2.6.2 单链表的应用举例——一元多项式求和

在数学上，一个一元多项式$P_n(x)$可按升幂的形式写成：

$$P_n(X) = P_0 + P_1X + P_2X^2 + P_3X^3 + \cdots + P_nX^n$$

它实际上可以由n+1个系数唯一确定。因此，在计算机内，可以用一个线性表P来表示：

$$P = (P_0, P_1, P_2, \cdots, P_n)$$

假设$Q_m(x)$是一个一元多项式，则它也可以用一个线性表Q来表示，即

$$Q = (q_0, q_1, q_2, \cdots, q_m)$$

若假设m<n，则两个多项式相加的结果$R_n(x)=P_n(x)+Q_m(x)$，也可以用线性表R来表示：

$$R = (p_0 + q_0, p_1 + q_1, p_2 + q_2, \cdots, p_m + q_m, p_{m+1}, \cdots, p_n)$$

可以采用顺序存储结构来实现顺序表的方法，使得多项式的相加的算法定义十分简单，即p［0］存系数p_0，p［1］存系数p_1, …, p［n］存系数p_n， 对应单元的内容相加即可。

但是在通常的应用中，多项式的指数有时可能会很高并且变化很大。 例如：

$$R(x) = 1 + 5x^{10000} + 7x^{20000}$$

- 若采用顺序存储，则需要20001个空间，而存储的有用数据只有三个，这无疑是一种浪费。
- 若只存储非零系数项， 则必须存储相应的指数信息。

假设一元多项式$P_n(x)=p_1x^{e1}+p_2x^{e2}+...+p_mx^{em}$，其中$p_i$是指数为$e_i$的项的系数(且$0\le e_1\le e_2\le...\le e_m=n$)， 若只存非零系数，则多项式中每一项由两项构成（指数项和系数项），用线性表来表示， 即珑

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

采用这样的方法存储，在最坏情况下，即n+1个系数都不为零， 则比只存储系数的方法多存储1倍的数据。对于非零系数多的多项式则不宜采用这种表示。

存储结构的选择

- 顺序存储
- 顺序存储的问题分析
- 链式存储

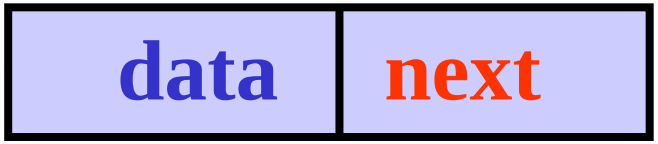
建立一元多项式的链式存储

（1） 用单链表存储多项式的结点结构如下：

```
typedef struct Polynode
{
    Node data;
    Polynode *next;
};
```

```
struct Node
{
    int coef;
    int exp;
};
```

一元多项式单链表的结点结构：



多项式的相加

多项式相加的过程：

1. 两个多项式中所有指数相同的项的对应系数相加，

   若和不为零，

   则构成“和多项式”中的一项；

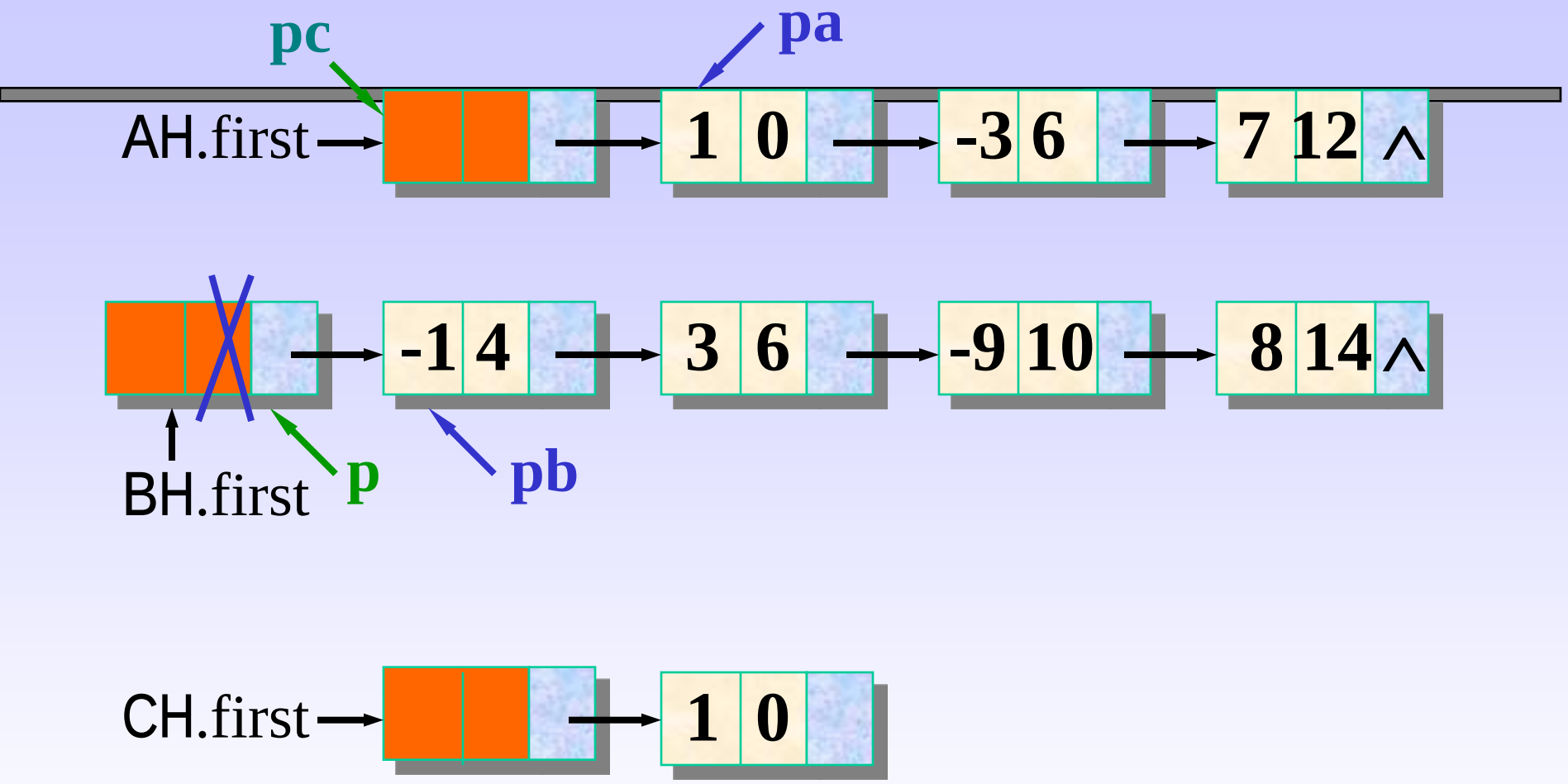
2. 所有指数不相同的项

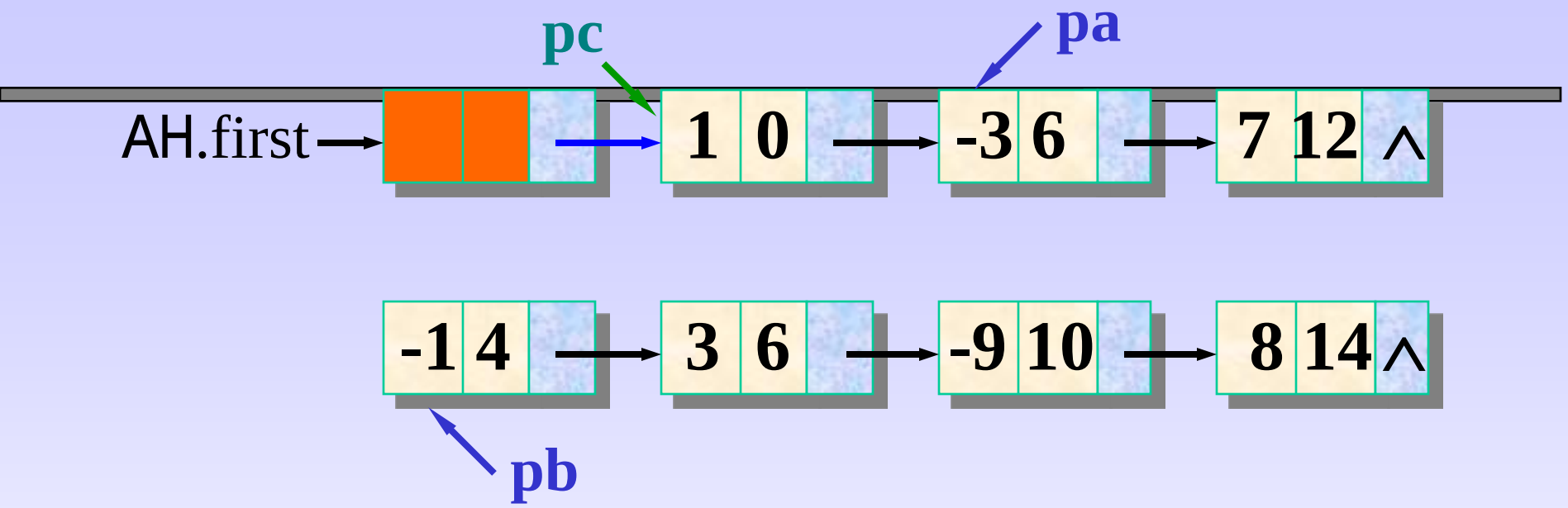
   均复抄到“和多项式”中。

多项式的相加

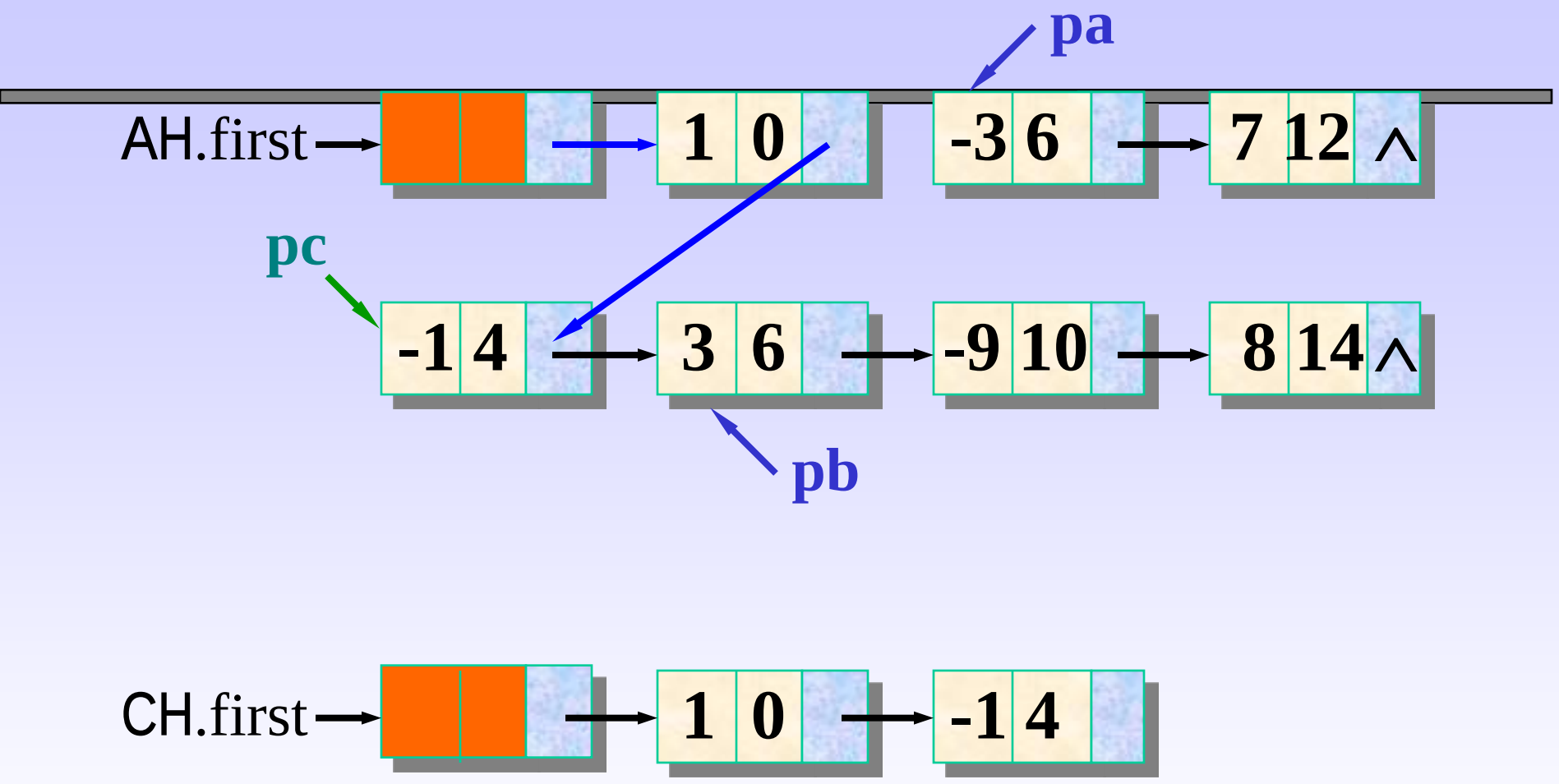
$$AH = 1 - 3x^6 + 7x^{12}$$

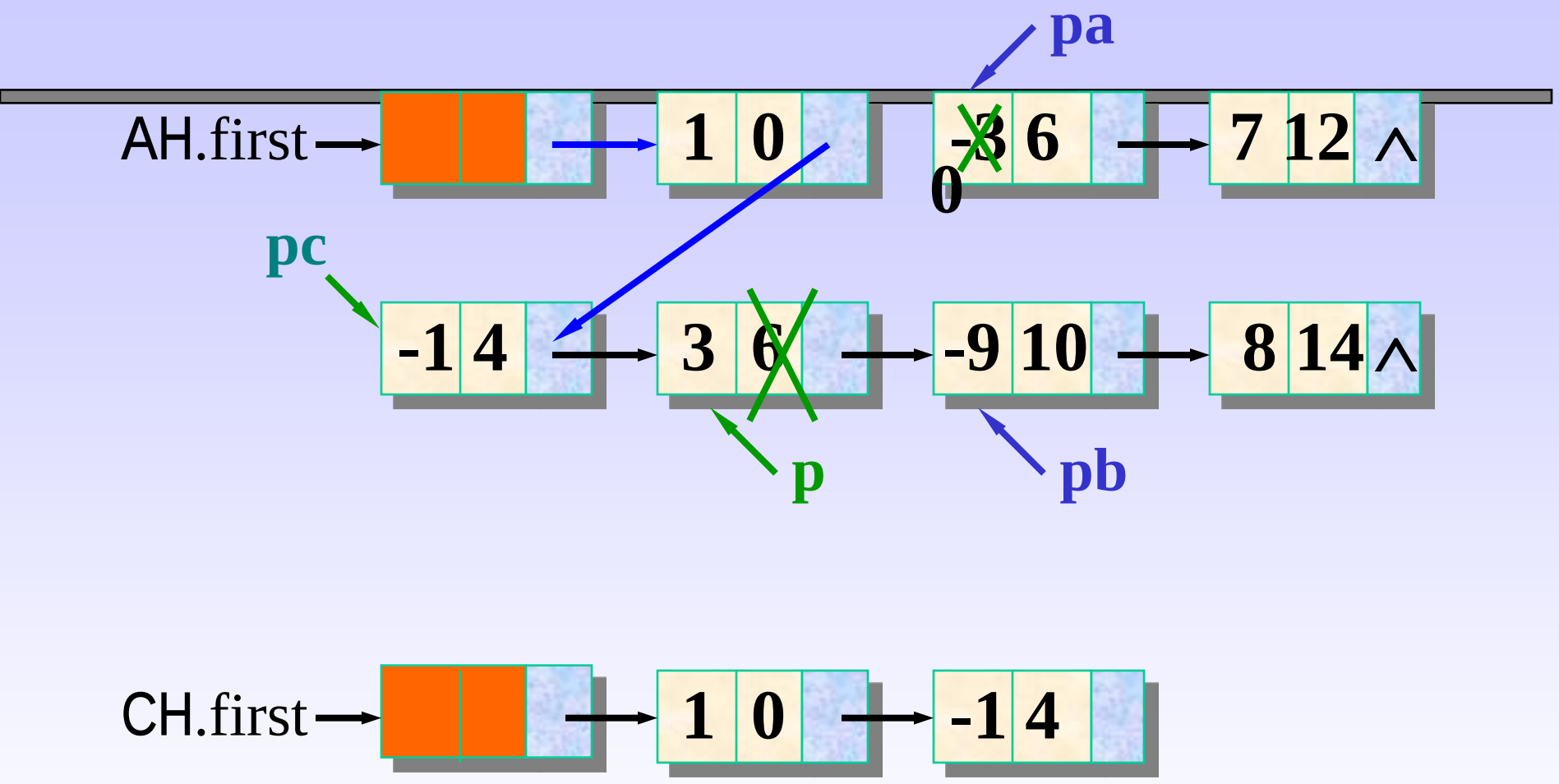
$$BH = -x^4 + 3x^6 - 9x^{10} + 8x^{14}$$

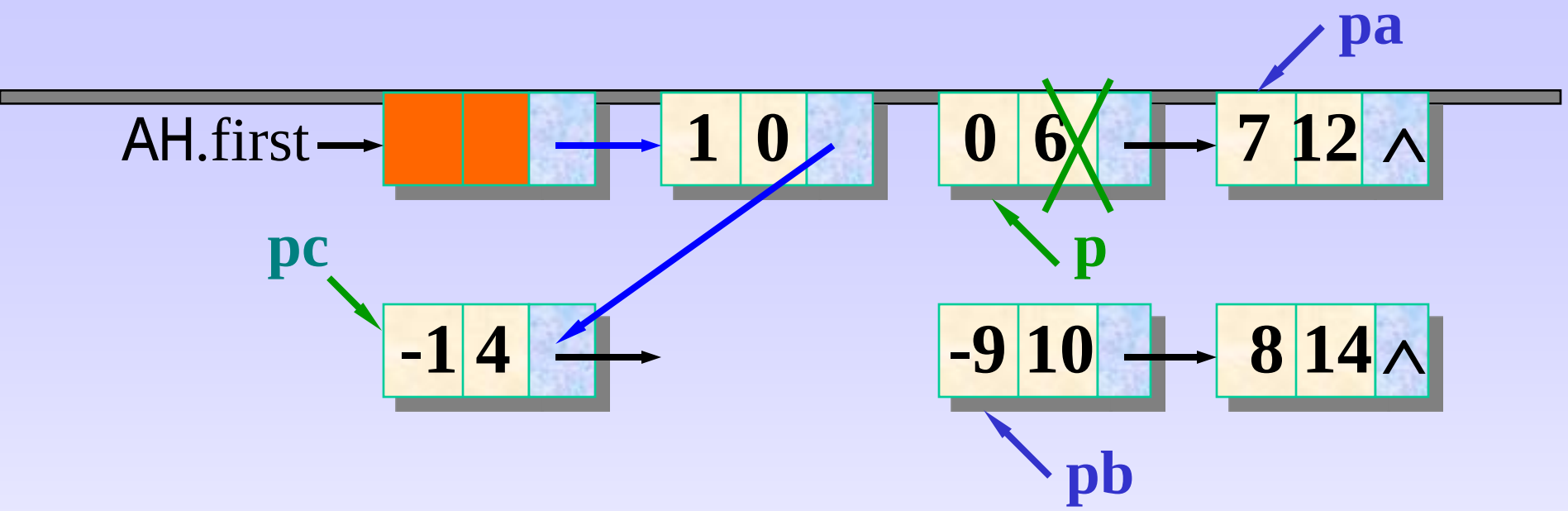

pc
pa
AH.first
1 0
-3 6
7 12 ∧
-1 4
3 6
-9 10
8 14 ∧
BH.first
p
pb
CH.first
1 0

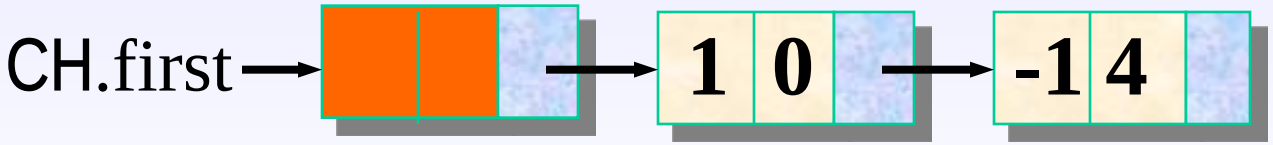

pc
pa
AH.first
1 0
-3 6
7 12 ∧
-1 4
3 6
-9 10
8 14 ∧
pb

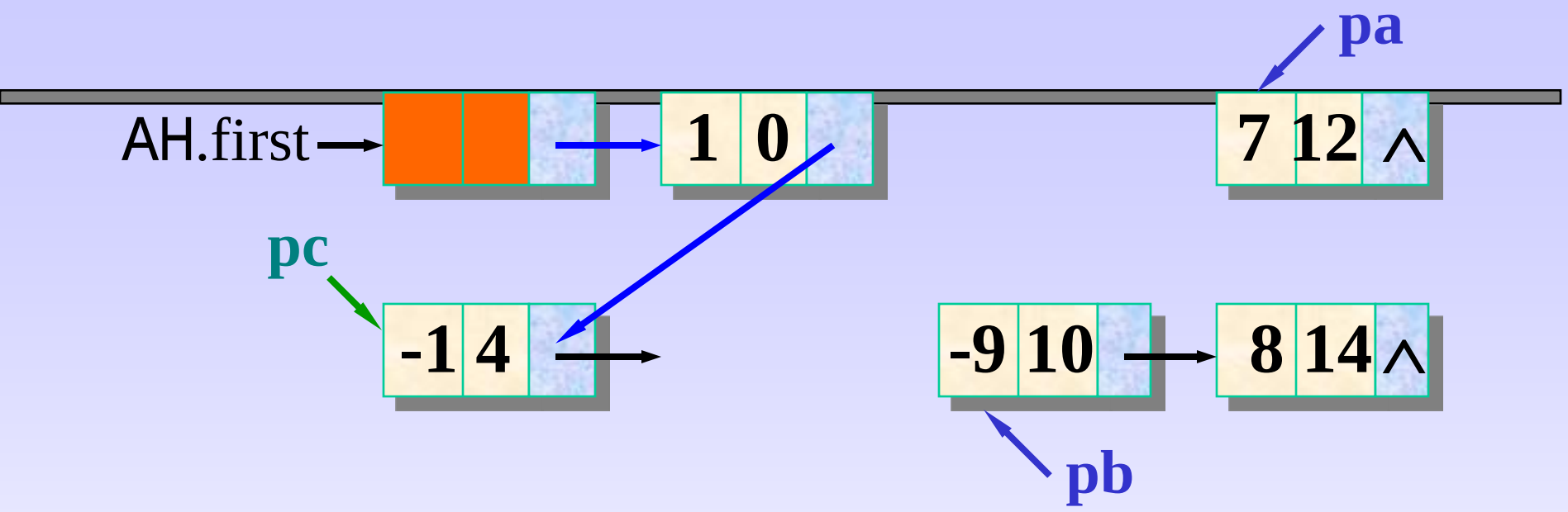

CH.first
1 0

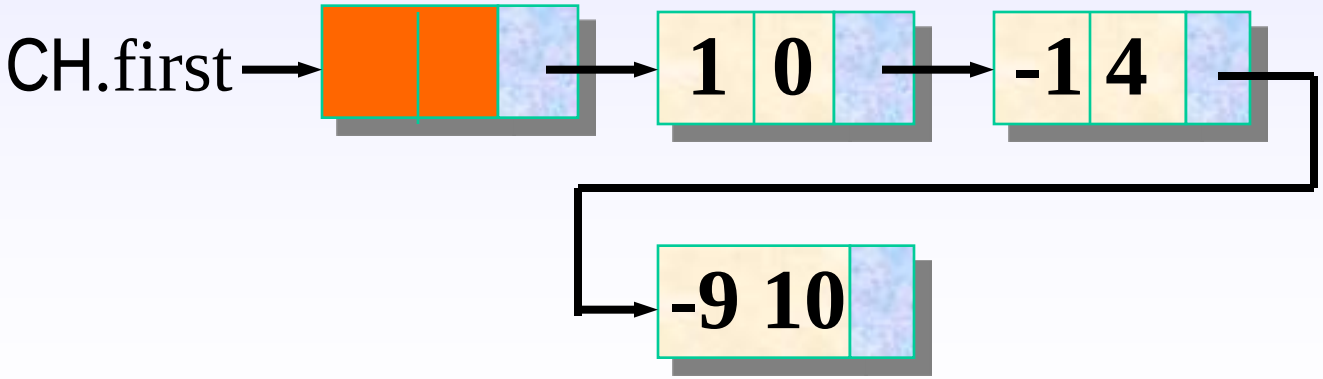

pa
AH.first
1 0
-3 6
7 12 ∧
pc
-1 4
3 6
-9 10
8 14 ∧
pb
CH.first
1 0
-1 4

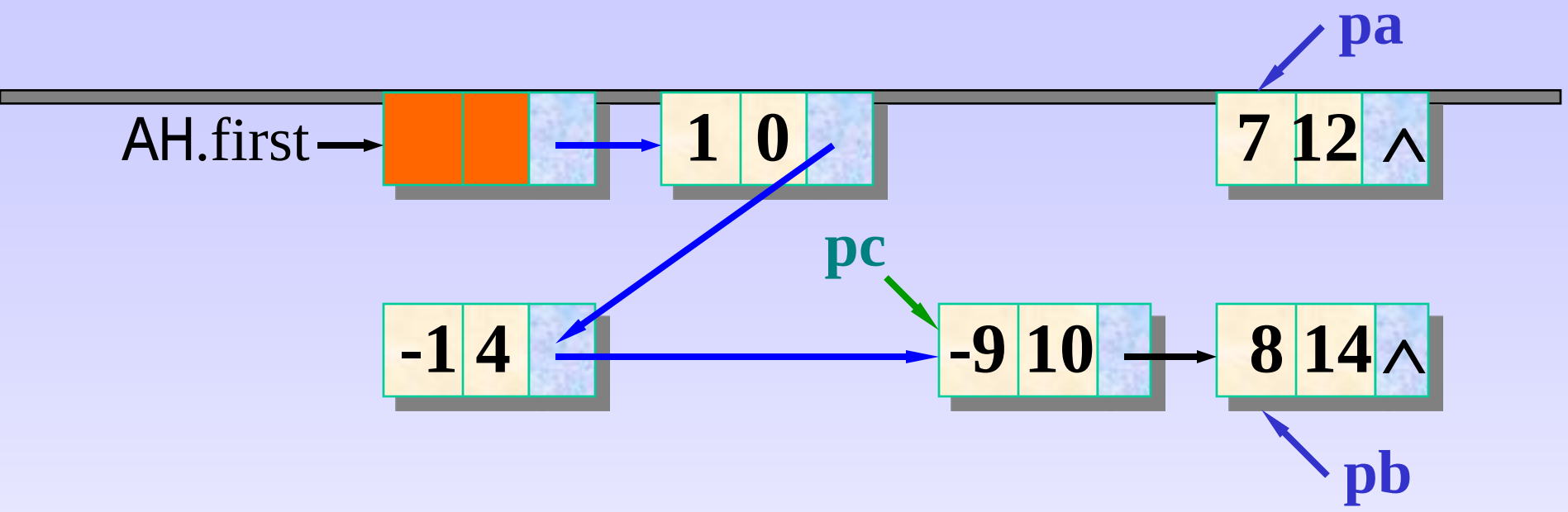

pa
AH.first
1 0
-3 6
0
7 12 ∧
pc
-1 4
3 6
-9 10
8 14 ∧
p
pb
CH.first
1 0
-1 4

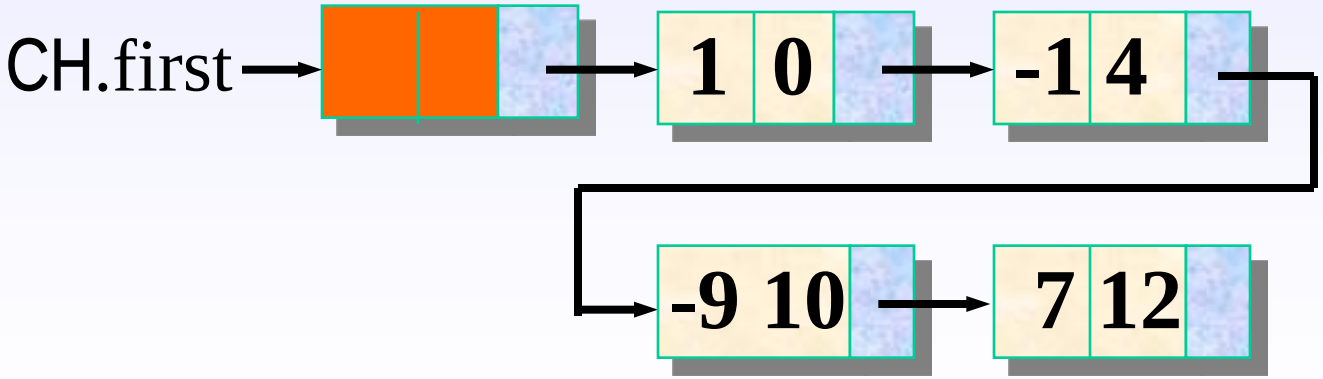

pa
AH.first
1 0
0 6
7 12 ∧
pc
p
-1 4
-9 10
8 14 ∧
pb

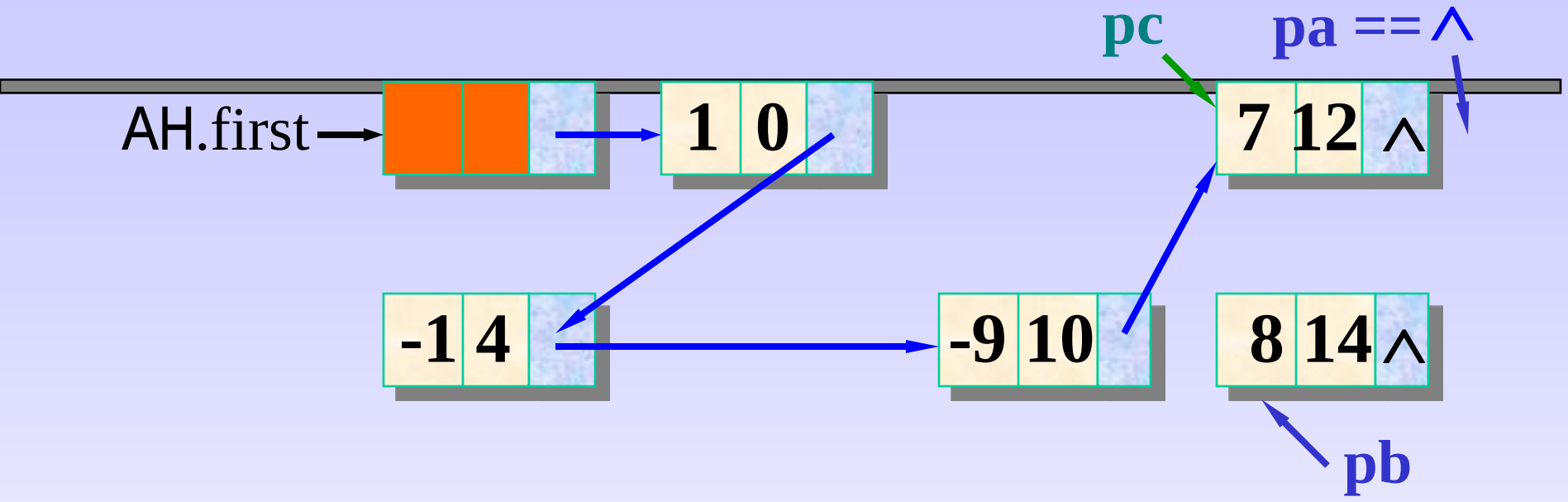

CH.first
1 0
-1 4

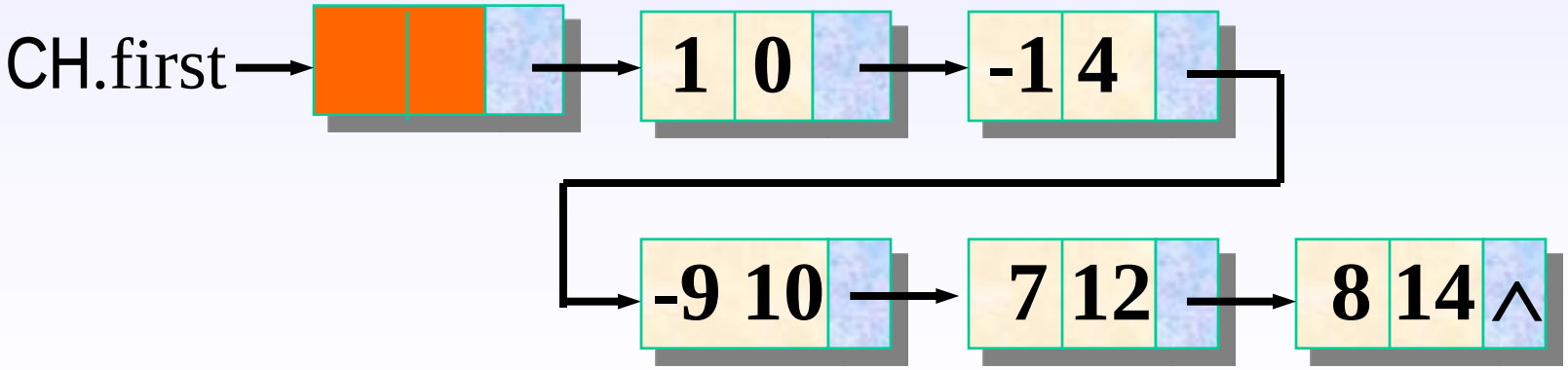

pa
AH.first
1 0
7 12 ∧
pc
-1 4
-9 10
8 14 ∧
pb

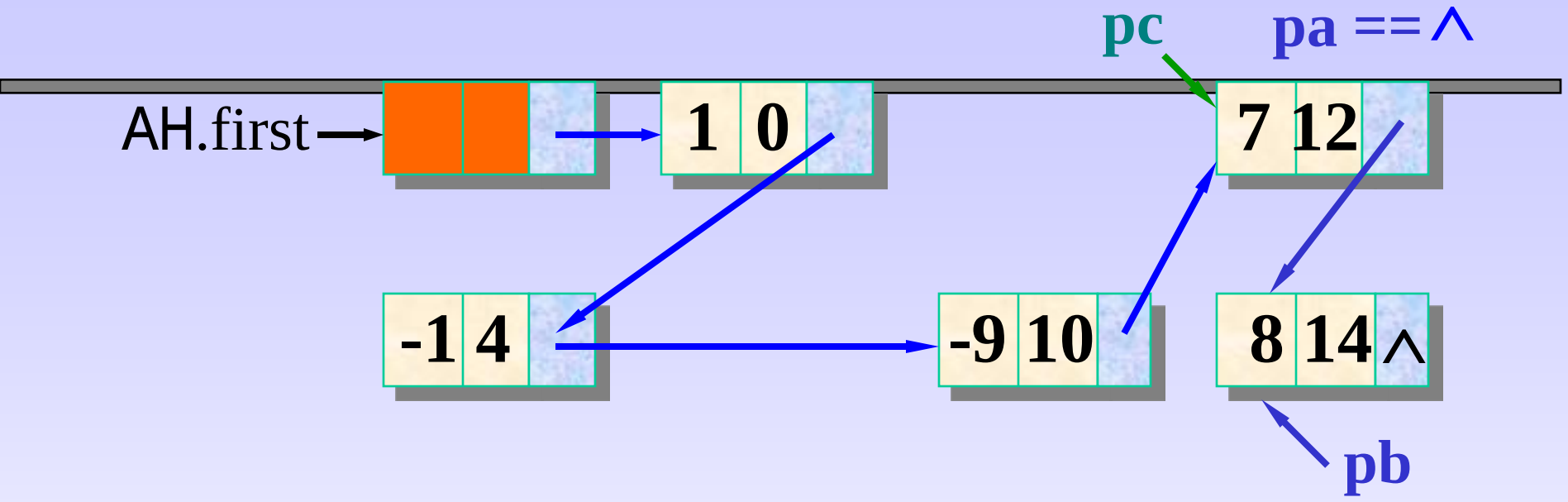

CH.first
1 0
-1 4
-9 10

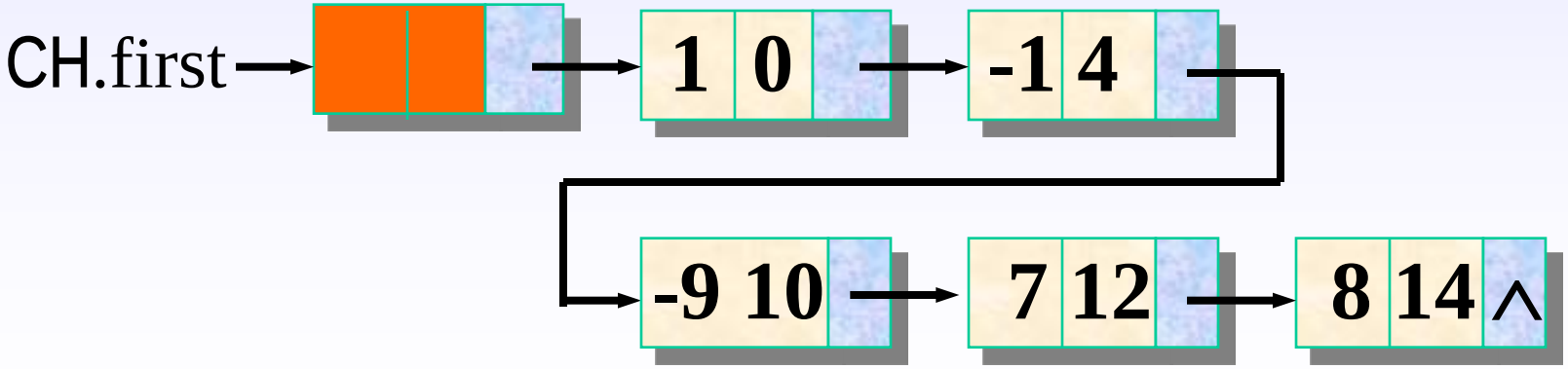

pa
AH.first
1 0
7 12 ∧
pc
-1 4
-9 10
8 14 ∧
pb


CH.first
1 0
-1 4
-9 10
7 12


pc
pa == ∧
AH.first
1 0
7 12 ∧
-1 4
-9 10
8 14 ∧
pb


CH.first
1 0
-1 4
-9 10
7 12
8 14 ∧


pc
pa == ∧
AH.first
1 0
7 12
-1 4
-9 10
8 14 ∧
pb


CH.first
1 0
-1 4
-9 10
7 12
8 14 ∧

一元多项式相加中的三种情况

1 若pa->exp<pb->exp， 则结点pa所指的结点应是“和多项式”中的一项，令指针pa、pc后移， pb 不变；

2 若pa->exp>pb->exp，则结点pb所指的结点应是“和多项式”中的一项， 将结点pb插入在结点pa之前，pb 、pc后移， pa 不变； 囍

3 若pa->exp==pb->exp， 则将两个结点中的系数相加，释放pb结点;

1 当和不为零时修改结点pa的系数域， pa、pc后移；

2 若和为零，释放pa,pa后移。

算法

pa,pb:当前比较的结点
pc：刚刚插入c的结点,
p：被删结点

1. 设置四个工作指针，pa, pb, pc, p,并对其进行初始化
2. 当 pa!=NULL 并且pb!=NULL时，重复做以下工作
   1. if(pa ->exp<pb -> exp),将pa加入到C中，后移pa、pc;
   2. if(pa -> exp>pb -> exp),将pb加入到C中，后移pb、pc;
   3. if(pa -> exp==pb -> exp) ,pa、pb的系数相加，删除pb,pb后移
      1. 如果和为0，删除pa, pa后移
      2. 否则，将和写入到pa中，将pa加入到C中，pa、pc后移;
3. 如pa!=NULL,将pa链入C中，否则将pb链入C中；

核心代码

```
Void Add(LinkList <T>&A, LinkList <T>B){
    Node<T> *pa, *pb, *pc, *p;
    T a, b;  //存放pa、pb两个指针变量中存储的系数
    pc = A.GetHead(); //结果存放指针
    p = B.GetHead();
    pa = A.GetHead()->next;
    pb = B.GetHead()->next;
    delete p;                    //删去bh的表头结点
```

```
template <class T>
Node<T> *LinkList<T>:: GetHead()
{return first; }
```

```
struct Node
{
    int coef;
    int exp;
    };
```

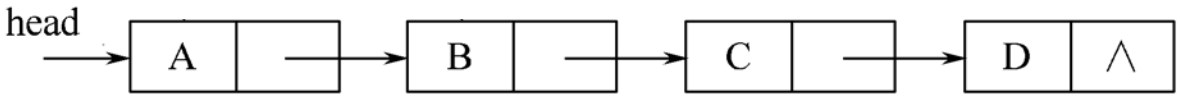
```
struct Polynode
    {
      Node data;
      Polynode *next;
    };
```

```
while ( pa != NULL && pb != NULL ) {
  a = pa->data;  b = pb->data;
  if(a.exp==b.exp){                    //pa->exp == pb->exp
      a.coef = a.coef + b.coef;        //系数相加
    p = pb;  pb = pb->next;
    delete p;                          //删去原pb所指结点
    if (a.coef  == 0) {//相加为零, 该项删除
        p = pa;  pa = pa->next;  delete p;
    }
   else {                //相加不为零, 加入c链
    pa->data = a;   pc->next = pa;
    pc = pa;  pa = pa->next;
     }
 }
```

```
else if(a.exp>b.exp) {          //pa->exp > pb->exp
            pc->next = pb;
            pc = pb;  pb = pb->next;
            }
      else{                 //pa->exp < pb->exp
            pc->next=pa;
            pc = pa;  pa = pa->next;
     }
   }
if (pa!=NULL ) pc->next=pa;    //处理剩余部分
else pc->next=pb;
}
```

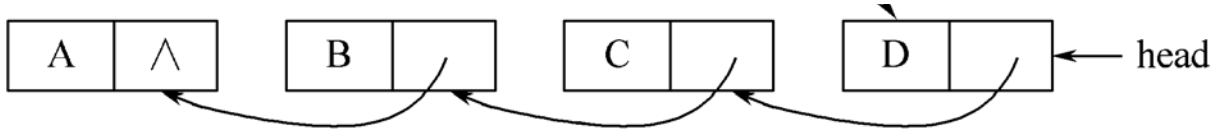
- 对结果表达式的构造：
  - 从空表开始，依次将A或B中结点加入到C中，完成C表的构造
- 教材中代码：
  - 将A看作是结果表达式的一部分，将B中的节点依次插入到A中，完成结果表达式的构造
  - 存在的问题
    - 未对指向节点前驱的指针进行及时的修改

不带头结点的单链表逆转

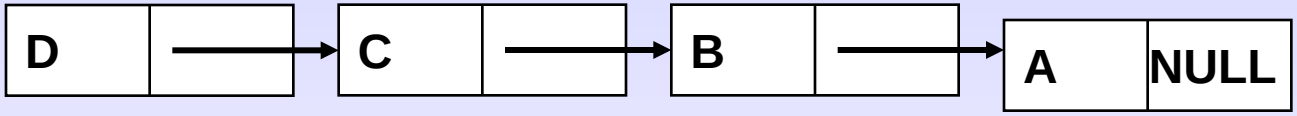


p q

j环中，使p.next指向前驱结点front



（c）循环结束后已逆转的单链表

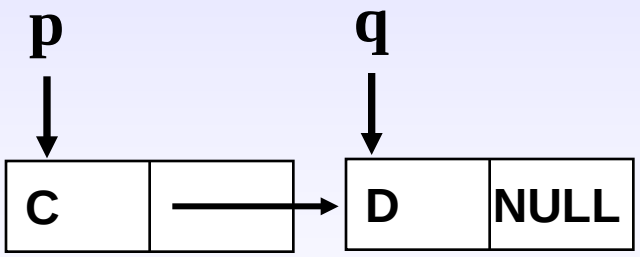


设置三个辅助变量

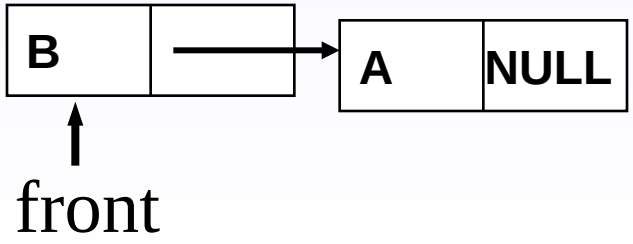
p:指向链表的头结点，头指针,初值为**list.head**

q:指向**p**的后继结点,初值为**NULL,**

front: 逆转之后的链表的头指针,初值为**NULL**。




```
q = p->next;
p->next = front;
front = p;
p = q;
```

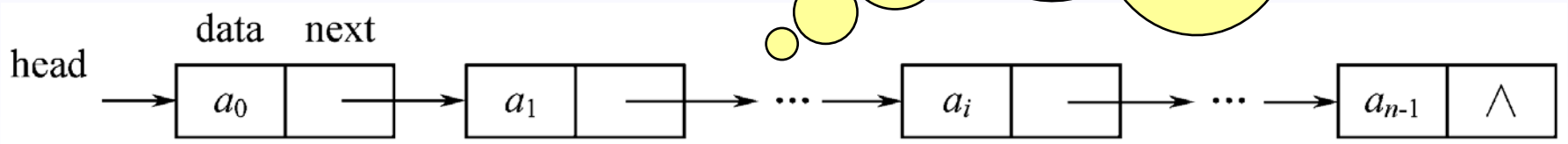


不带头结点的单链表逆转算法描述

- 如果链表为空表，抛出异常
- 设置三个辅助变量:
  - p(指向链表的头结点，头指针,初值为list.head),
  - q（指向p的后继结点,初值为null,
  - front（逆转之后的链表的头指针,初值为null）。
- 在链表不空的情况下，重复做以下工作：
  - 用q记录p的后继（原链表的新的头结点）
  - p.next=front; //将p作为头结点，链入逆转之后的表中
  - 分别修改两个链表的头指针

```
public void reverse()      //将单链表逆转
    {
        Node<E> *p= a->head, q=null, front=NULL;
        while (p!=NULL)
        {
            q = p->next;
            p->next = front;
            front = p;
            p = q;
        }
}
```

思考：如何实现带头结点的链表的逆置？



建立排序的单链表

构造链表，使得在插入过程中，依次将要插入的数据同链表中的数据进行比较，确定插入点进行插入。

插入过程(升序排列)：

寻找比插入值（element）大的第一个节点，

查找成功的情况下，进行插入。

如果查找失败，则新插入的节点为最后一个节点

b 带头结点的单链表

有序(升序)链表的插入

```
void LinkList<T>::Insert(T x)
{
    Node<T> *s,*p,*front;
    s=new Node<T>;
    s->data=x;
    front=first; p=first->next;
    while(p && x>p->data)
    {
        front=p;
        p=p->next;
    }
    s->next=p;
    front->next=s;
    return ;
}
```

b 带头结点的单链表

建立有序链表

```
void LinkList<T>::create(T a[],int n)
{

    for(int i=0,i<n;i++)
        Insert(a[i]);
}
```

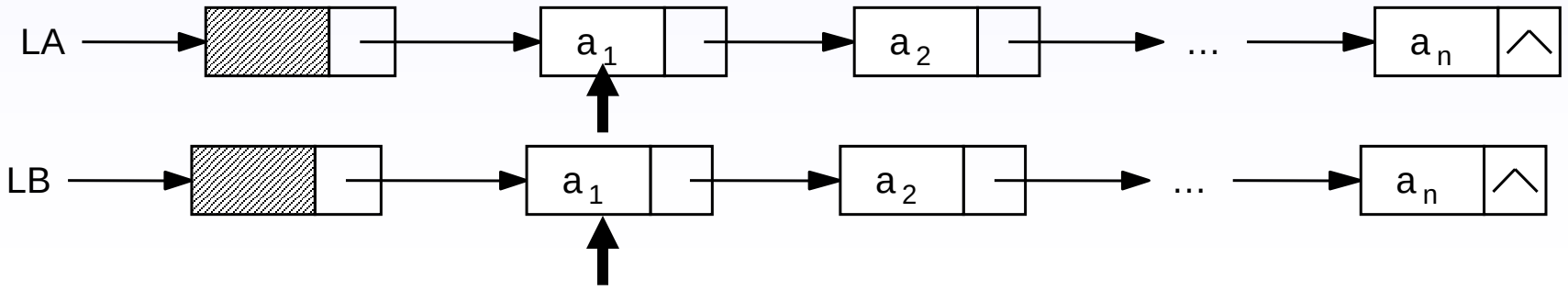
b 带头结点的单链表

有序表的合并

有两个链表LA和LB，其元素均为非递减有序排列， 编写一个算法，将它们合并成一个链表LC， 要求LC也是非递减有序排列

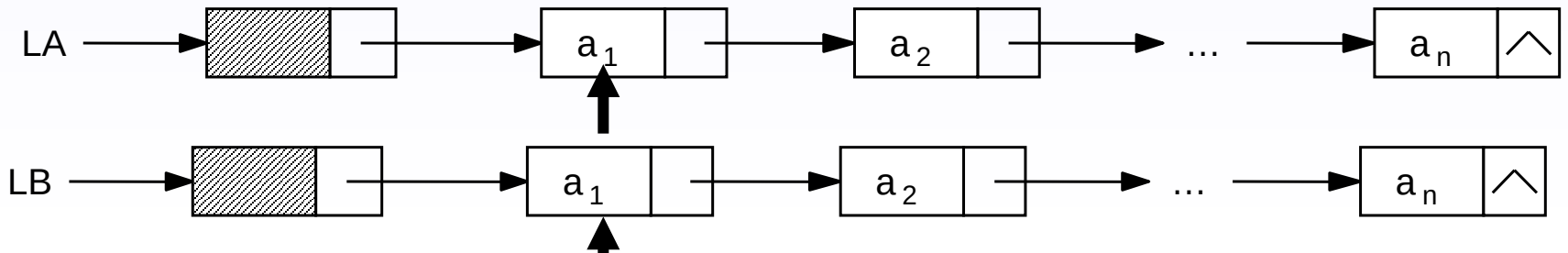
算法思想：

- 设表LC是一个空表，为使LC也是非递减有序排列，可设两个指针pa、pb、pc分别指向表LA、LB和LC中的元素，
  - 若pa->data>pb->data， 则先将pb插入到表LC中;
  - 若pa->data<=pb->data ，则先将pa插入到表LC中，
- 如此进行下去，直到其中一个表被扫描完毕，然后再将未扫描完的表中剩余的所有元素放到表LC中。



合并算法

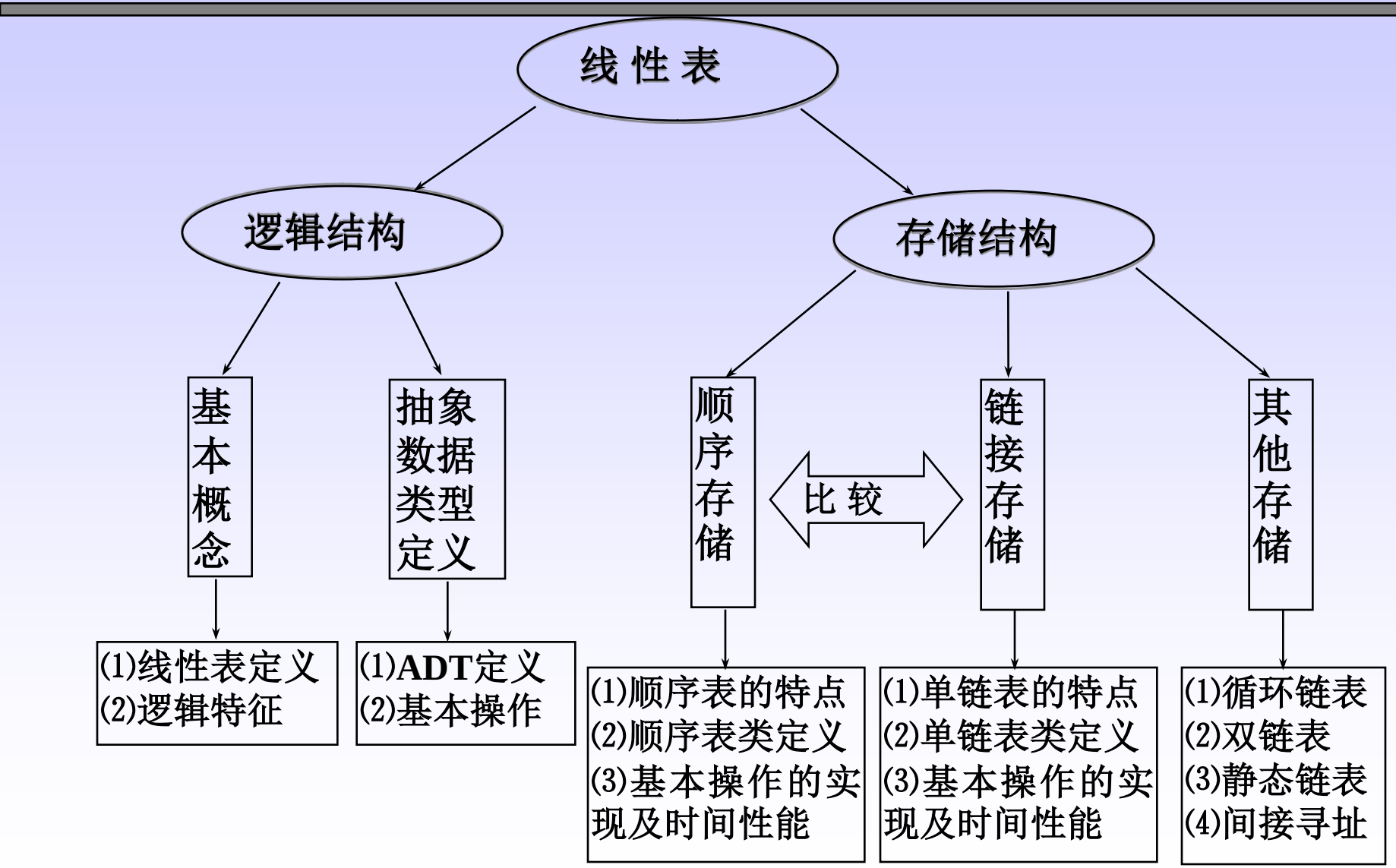
1. 设置工作指针pa,pb,pc，并对其进行初始化
2. 重复做以下工作，直到pa或pb为空
    1. 将pa的值和pb的值进行比较
        1. 如果pa->data>pb->data,
            1. pc->next=pb,pb=pb->next
        2. 如果pa->data<=pb->data,
            1. pc->next=pa,pa=pa->next
    2. Pc=pc->next
3. 如果pa不空，
    1. 则将pa中剩余的结点链入pc
    2. 否则，将pb中剩余的结点链入pc



```
template <class T>
void  merge(LinkList<T> LA, LinkList<T> LB){
   Node <T> *pa,*pb,*pc;

        pa=LA.first;  pb=LB.first;  pc=LA.first;
        pa=pa->next; pb=pb->next;
        while(pa&&pb)       {
              if (pa->data>pb->data)
                { pc->next=pb;pc=pb;pb=pb->next;}
                else
                 { pc->next=pa;        pc=pa;pa=pa->next; }
        }
        if (pa)  pc->next=pa;
        else     pc->next=pb;
}
```

本章总结


线性表
逻辑结构
存储结构
基本概念
抽象数据类型定义
顺序存储
比较
链接存储
其他存储
(1)线性表定义
(2)逻辑特征
(1)ADT定义
(2)基本操作
(1)顺序表的特点
(2)顺序表类定义
(3)基本操作的实现及时间性能
(1)单链表的特点
(2)单链表类定义
(3)基本操作的实现及时间性能
(1)循环链表
(2)双链表
(3)静态链表
(4)间接寻址

上机实验

P169 顺序表操作验证

P174 单链表操作验证（循环链表，双链表）

设计：

分别用顺序表和单链表通过基本操作的组合来实现：

假设利用两个线性表LA和LB分别表示两个集合A和B(即：线性表中的数据元素即为集合中的成员)，现要求修改集合A，使得$A=A\cup B$，即集合A存储A和B的并集。