第二章 线性表(linear list)

本章的基本内容是：

2.1 线性表的定义

2.2 线性表的顺序存储及实现

2.3 线性表的链接存储及实现

2.4 线性表的其他存储及实现

2.5 应用实例

2.1 线性表的定义

例1：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 线性表的定义

例2：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

例1、例2中数据元素之间的关系是什么？

2.1 线性表的定义

- 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 。

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：

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。

2.2 线性表的顺序存储结构及实现

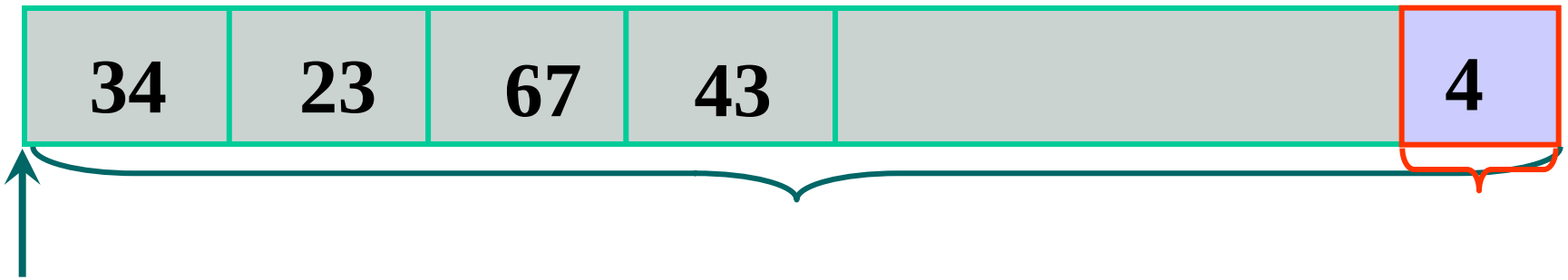
顺序表(sequential list)——线性表的顺序存储结构(sequential storage structure)

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

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

顺序表——线性表的顺序存储结构

例：（34, 23, 67, 43）

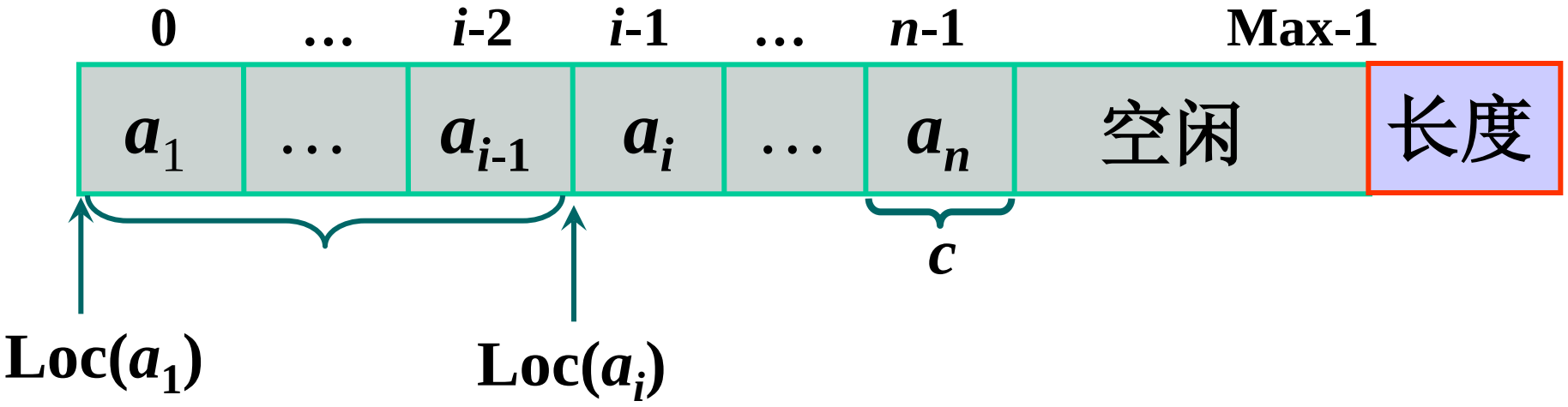
如何实现顺序表的内存分配？

顺序表 → 一维数组

```
int data[4];
```

顺序表——线性表的顺序存储结构

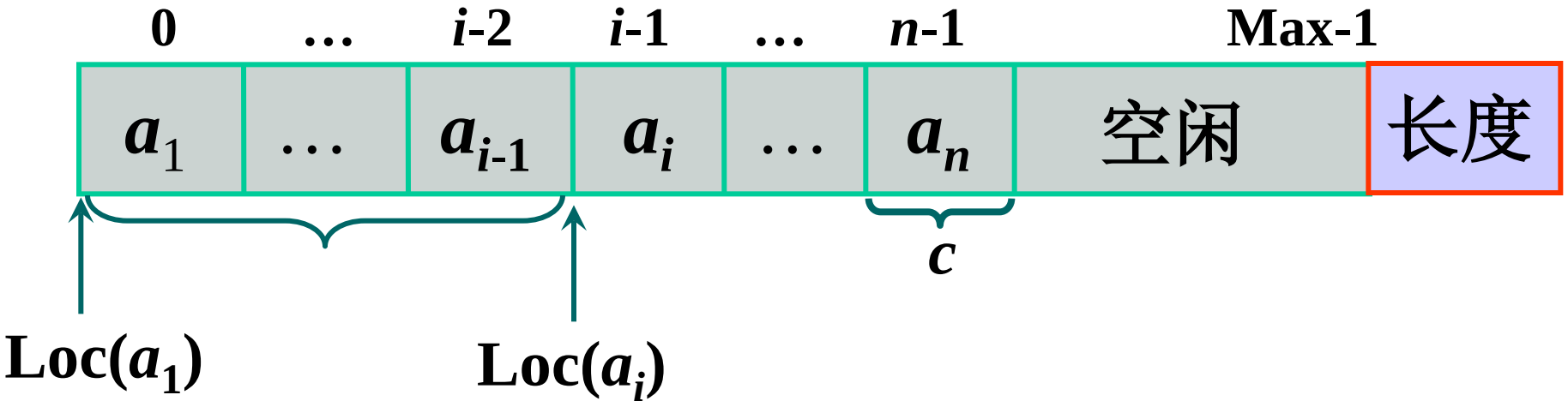
一般情况下，$(a_1, a_2, \ldots, a_{i-1}, a_i, \ldots, a_n)$的顺序存储：



如何求得任意元素的存储地址？

顺序表——线性表的顺序存储结构

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(random access)： 在$O(1)$时间内存取数据元素

顺序表——线性表的顺序存储结构

存储结构是数据及其逻辑结构在计算机中的表示；

“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$O(1)$。

顺序表的实现

顺序表类的声明

```
const int maxSize=100;    //表的最大尺寸
typedef int dataType;     // 表元素为整型
class seqList
{
 public:
    seqList( ){len=0 ;}                    //构造函数
    seqList(dataType a[ ], int n);
    ~seqList( ) {}                         //析构函数
    int length( ){return len;}
    dataType get(int i);
    int locate(dataType x );
    void insert(int i, dataType x);
    dateType remove(int i);
 private:
    dataType data[maxSize];
    int len;
};
```

还可以增加，诸如：

```
bool full();
bool empty();
void clear();
```

等函数。

顺序表的实现（采用C++的模板机制）

顺序表类的声明

```
const int maxSize=100;    //表的最大尺寸
template <class datatype>  //定义模版类
class seqList
{
 public:
    seqList( ){len=0;}                    //构造函数
    seqList(dataType a[ ], int n);
    ~seqList( ) ;                         //析构函数
    int length( ){return len;}
    dataType get(int i);
    int locate(dataType x );
    void insert(int i, dataType x);
    dataType remove(int i);
 private:
    dataType data[maxSize];
    int len;
};
```

还可以增加，诸如：

```
bool full();
bool empty();
void clear();
```

等函数。

顺序表的实现——无参构造函数

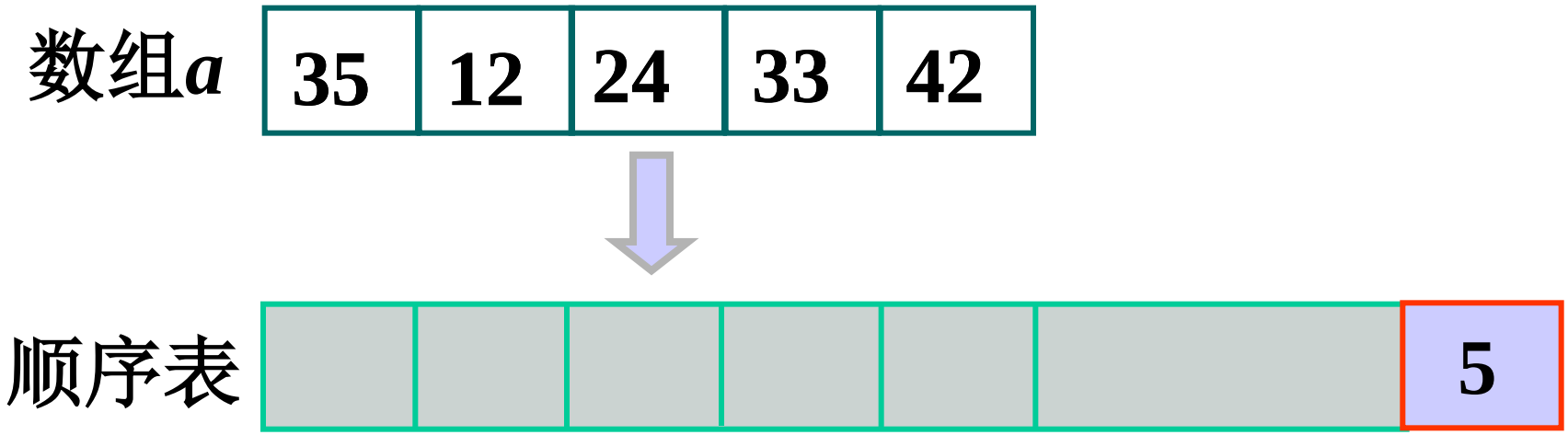
操作接口：**seqList（ ）**

算法描述：

```
seqList::seqList( )
{
   len=0;
}
```

顺序表的实现——有参构造函数

操作接口：**seqList（dataType a[ ], int n）**



顺序表的实现——有参构造函数

算法描述：

```
seqList::seqList(dataType  a[ ], int n)
{
    if (n>maxSize) throw "参数非法";
    for (int  i=0; i<n; i++)
        data[i]=a[i];
    len=n;
}
```

顺序表的实现——插入

操作接口： **void insert(int i, dataType 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

↓

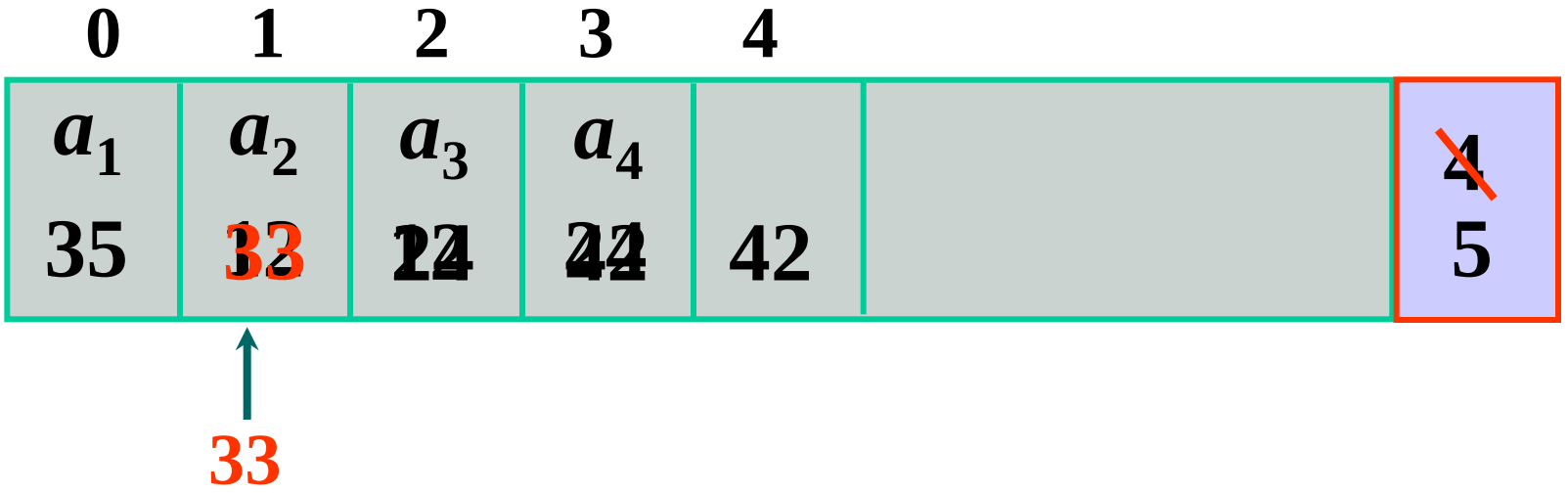
顺序存储要求存储位置反映逻辑关系

↓

存储位置要反映这个变化

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：len>=MaxSize

合理的插入位置：1≤i≤len+1（i指的是元素的序号）

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；

顺序表的实现——插入

算法描述——C++描述

基本语句？

```
void seqList::insert（int i, dataType  x）
{
    if（len>=maxSize）throw "上溢";
    if（i<1 || i>len+1）throw “位置异常";
    for （j=len; j>=i; j−−）
        data[j]=data[j−1];
    data[i−1]=x;
    len++;
}
```

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$n+1$次，时间复杂度为$O(n)$。

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$O(n)$。

顺序表的实现——删　除

操作接口： **dataType remove(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

↓

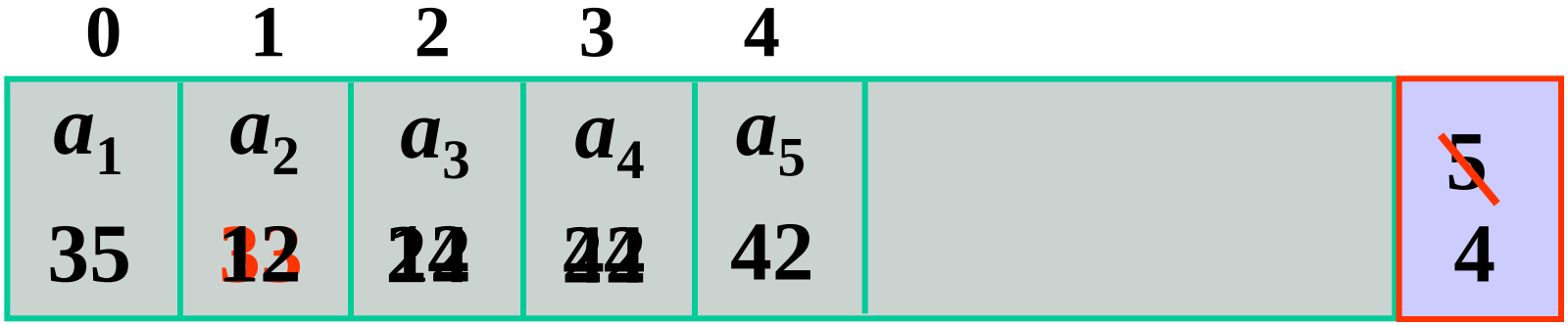
顺序存储要求存储位置反映逻辑关系

↓

存储位置要反映这个变化

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（自行完成）和C++描述的算法；
3. 分析时间复杂度。

顺序表的实现——删除

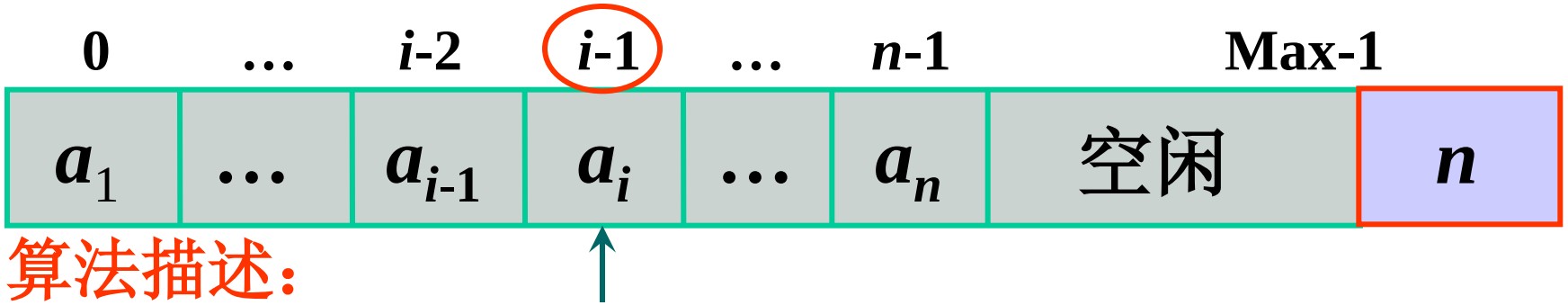
算法描述——C++描述

基本语句？

```
dataType  seqList::remove(int i)
{
    if（len==0）throw “下溢”；
    if（i<1 || i>len）throw “位置异常”；
    dataType x=data[i-1];
    for （j=i;j<len; j++）
        data[j-1]=data[j];
    len--;
    return x;
}
```

顺序表的实现——按位查找

操作接口： dataType get(int i)



算法描述：

```
dataType seqList::get( int i )
{
    if (i<1 || i>len) throw “查找位置非法”;
    else return data[i-1];
 }
```

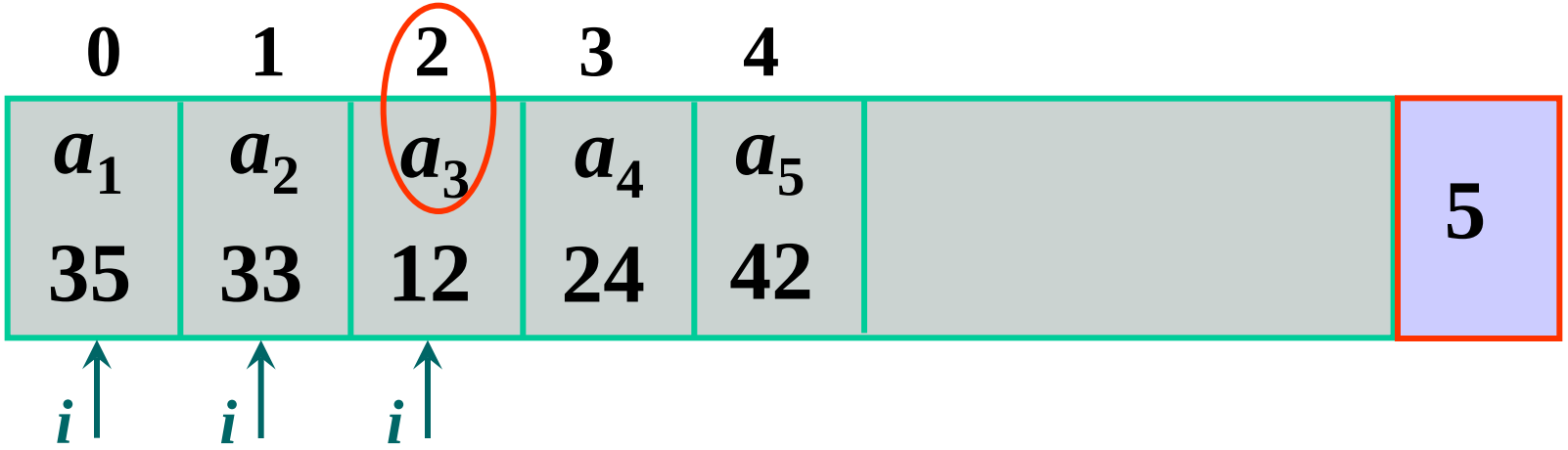
时间复杂度？

顺序表的实现——按值查找

操作接口： **int locate(dataType 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



顺序表的实现——按值查找

算法描述：

```
int seqList::locate（dataType x）
{
    for（i=0; i<len; i++）
        if（data[i]==x） return i+1;
    return 0;
}
```

时间复杂度？

顺序表的优缺点

顺序表的优点：

- 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；
- 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：

- 插入和删除操作需要移动大量元素；
- 表的容量难以确定，表的容量难以扩充；
- 造成存储空间的碎片。

顺序表的应用举例

查找特定位置上的乘客编号

下列程序要求用户输入一连串数据，存放在表中，当用户询问其输入的第n个数据项时，程序将输出位于这个位置上的数据项的值。

顺序表的应用举例（续1）

```
int main(){
//输入：用户提供数值n，代表n个乘客
//输出：将用户指定位置上的乘客编号（或其它数据项）打印出来
  int n;
  int item;
  seqList passengers; //定义一个表，表明是passengers
  cout<<“输入乘客人数n”<<endl;
  cin>>n;
  cout<<“按任意顺序输入乘客编号”<<endl;
  for (int i=1; i<=n; i++){
    cin>>item;
    passengers.insert(i,item);
  }
```

顺序表的应用举例（续2）

```
  cout<<endl<<endl;
  cout<<“请输入一个位置号：”
  cin>>n;
  while(n!=-1){
    if (n>=1&&n<=passengers.length())
      cout<<“位于该位置上的乘客编号是：”<<passengers.get(n)<<endl;
    else
      cout<<“该位置已经超出表的范围”<<endl;
    cout<<“请输入一个位置号（-1结束）：”;
    cin>>n;
  }
  cout<<endl;
  return 0;
}
```

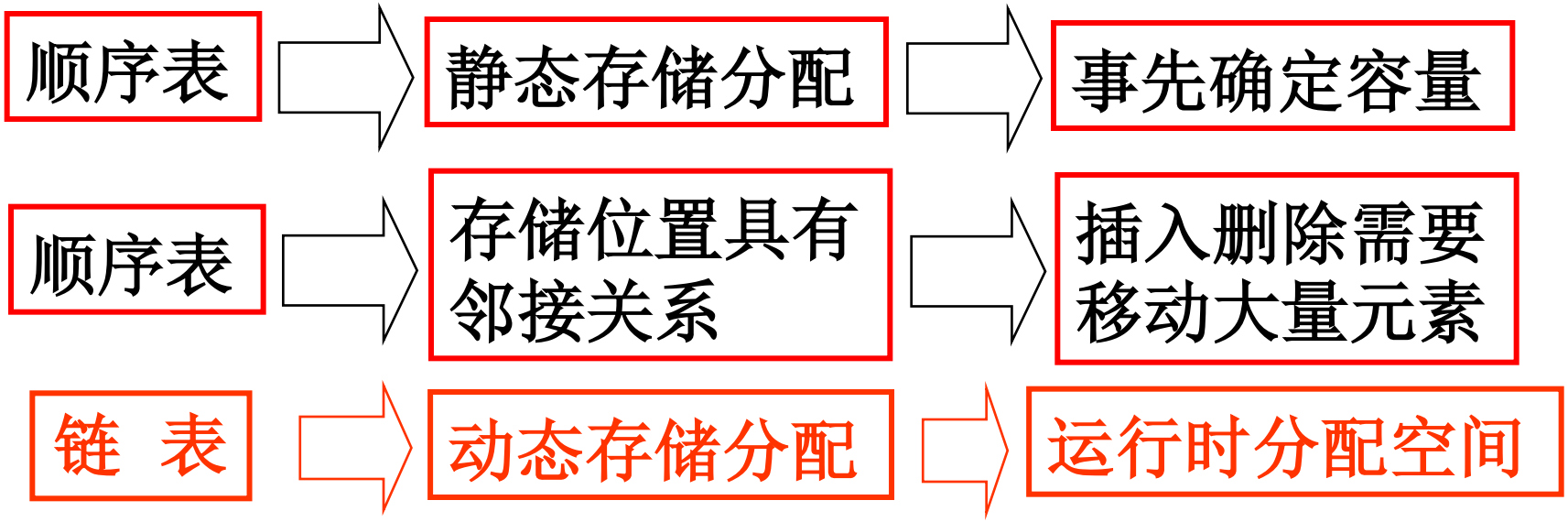
```
输入乘客人数n
3
按任意顺序输入乘客编号
1001 1002 1003



请输入一个位置号：2
位于该位置上的乘客编号是：1002
请输入一个位置号（-1结束）：3
位于该位置上的乘客编号是：1003
请输入一个位置号（-1结束）：-1

Press any key to continue
```

2.3 线性表的链接存储结构及实现



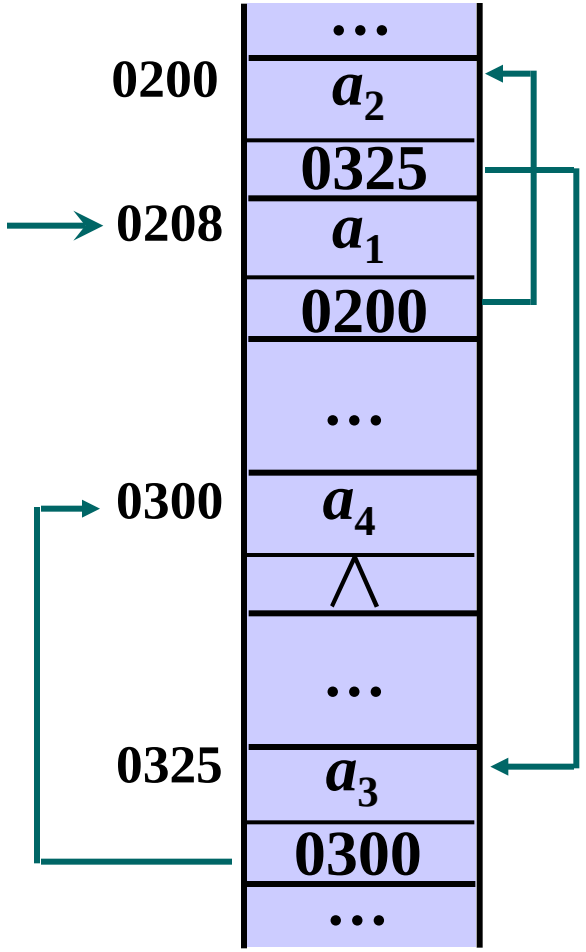
链表(linked list) 就是线性表的链接存储结构。

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

单链表结构

例：(a_1, a_2, a_3, a_4)的存储示意图



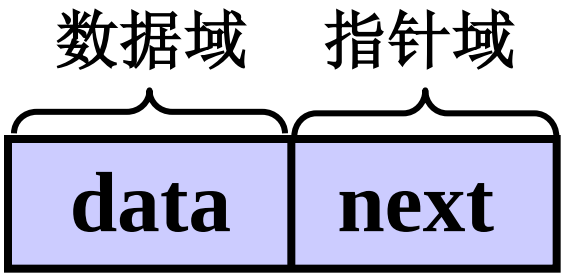
存储特点：

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。

单链表结构

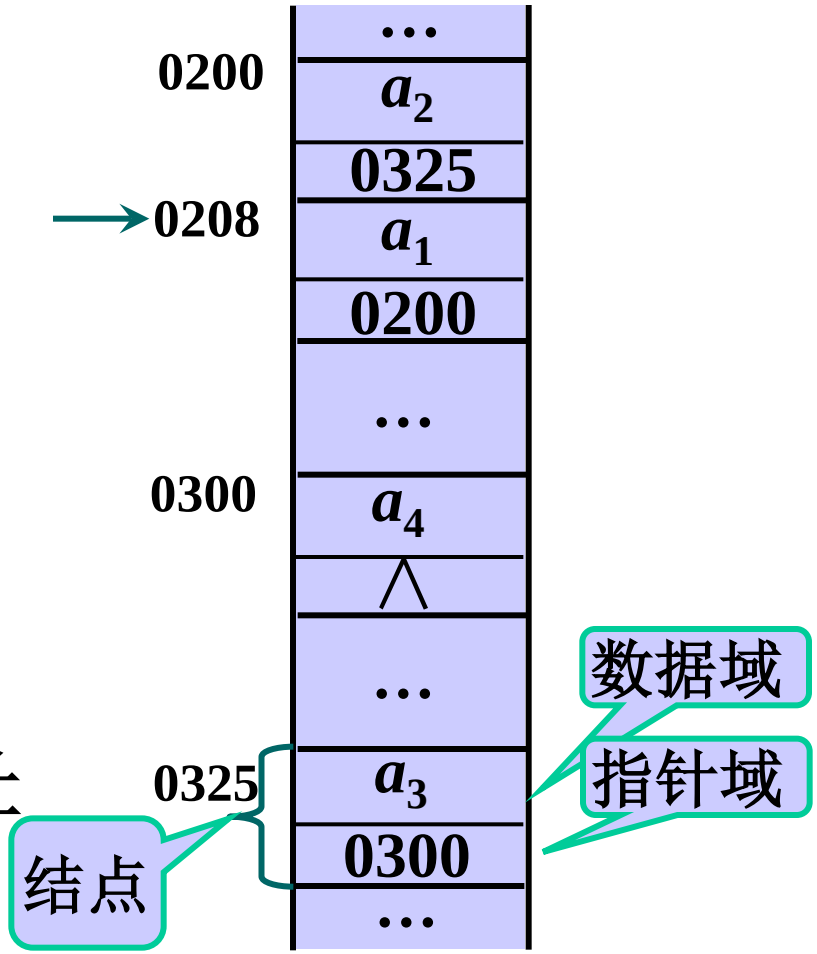
单链表是由若干结点构成的；
且每个结点只有一个指针域。

单链表的结点结构：



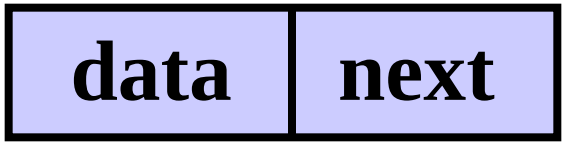
data：存储数据元素

next：存储指向后继结点的地址



单链表结构

单链表的结点结构：

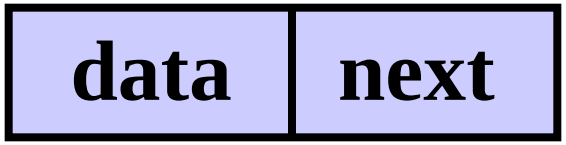



```
typedef int dataType
struct node
{
    dataType data;
    node *next;
};
```

如何申请一个结点？

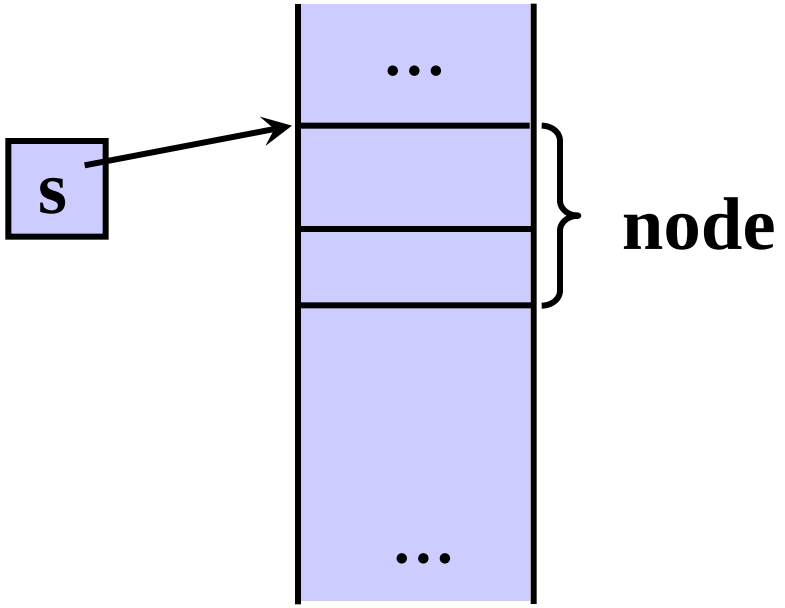
单链表结构

单链表的结点结构：



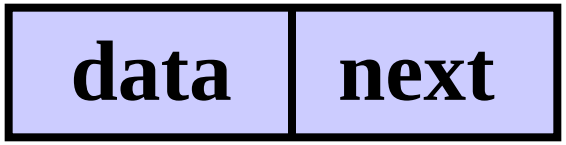

```
typedef int dataType
struct node
{
    dataType data;
    node *next;
};
```

s=new node ;//申请一个结点



单链表结构

如何引用数据元素？

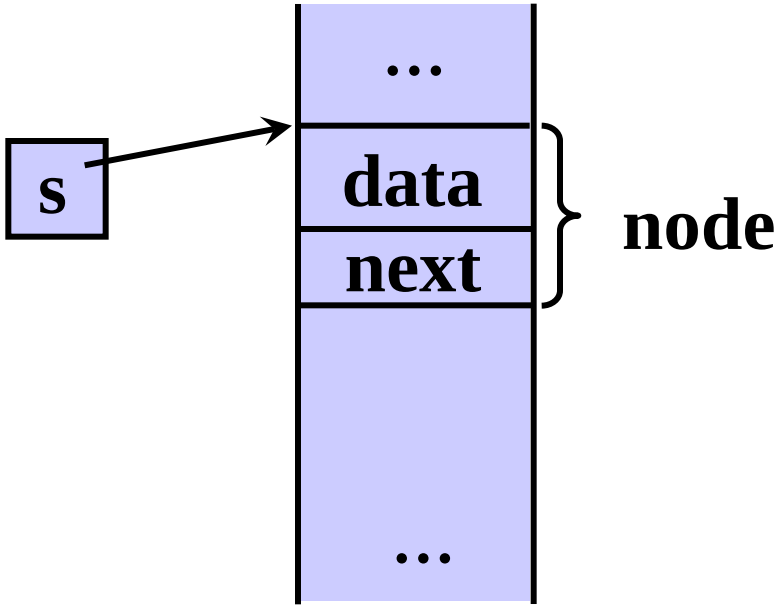


s=new node ;

s->data;

如何引用指针域？

s->next;



单链表结构

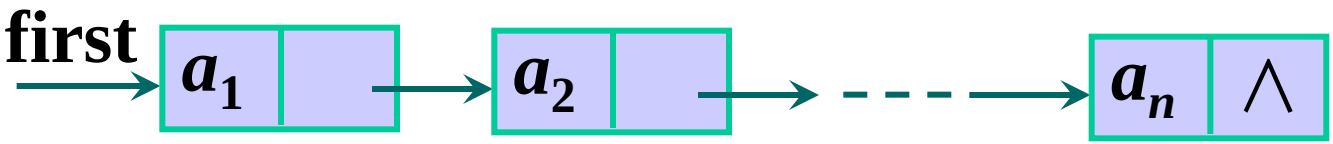
如何实现链接存储结构？

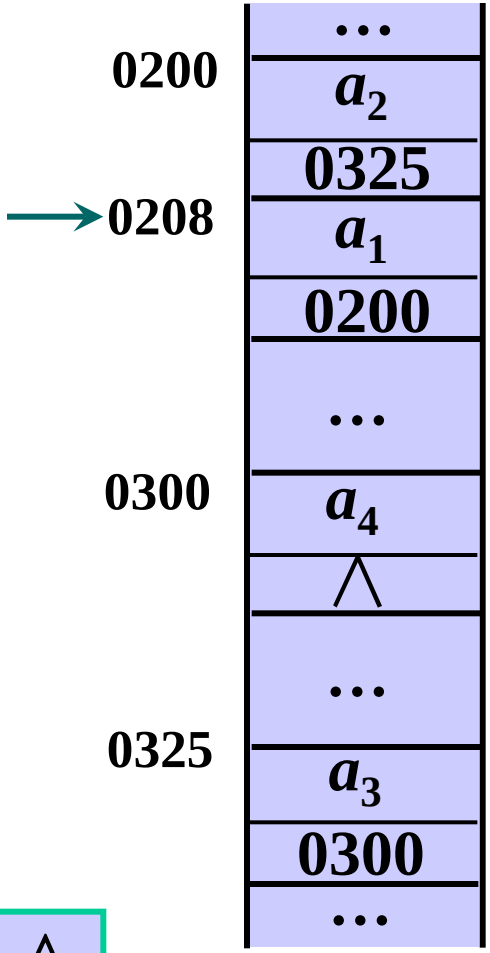
数据元素之间的逻辑关系是利用指针来实现的。

空表

first=NULL

非空表

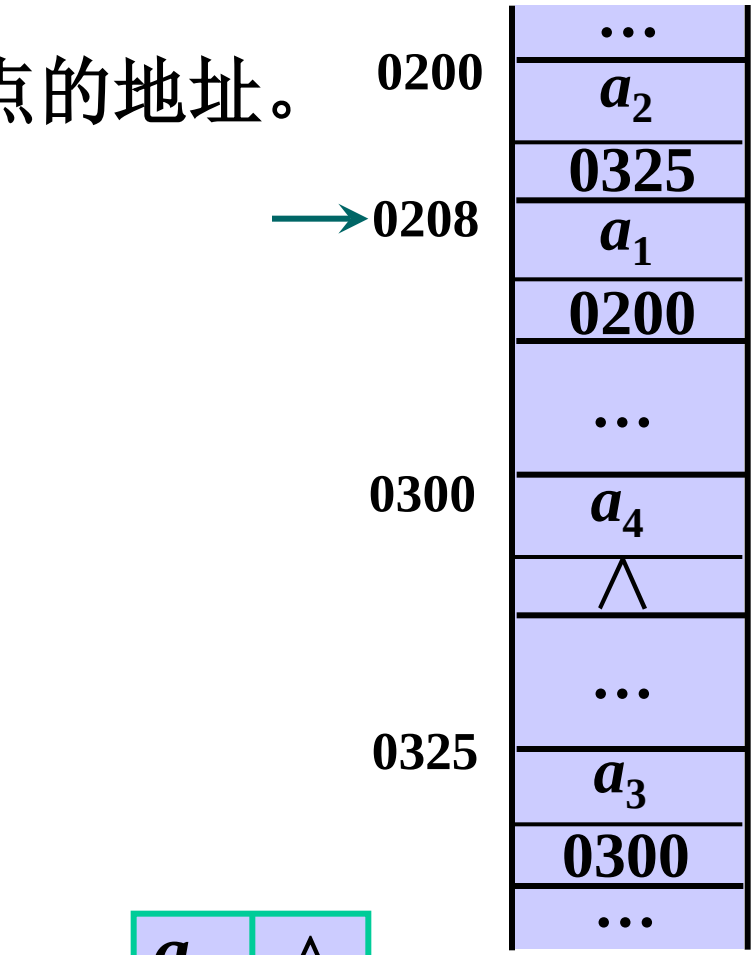




单链表结构

头指针(first)： 指向第一个结点的地址。

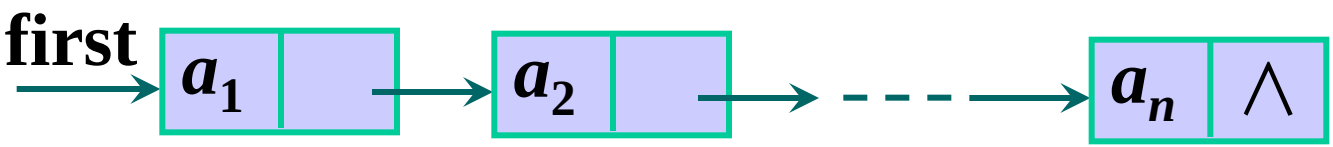
尾标志： 结点的指针域为空。



空表

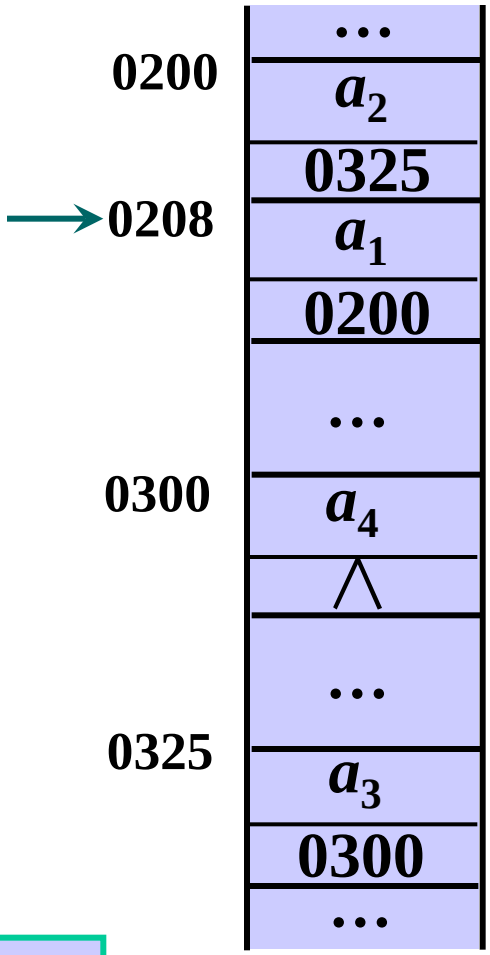
first=NULL

非空表



单链表结构

由于空表的头指针为空，非空表的头指针指向开始结点，所以设计算法时需要特殊处理空表的情况，从而增加程序的复杂性。

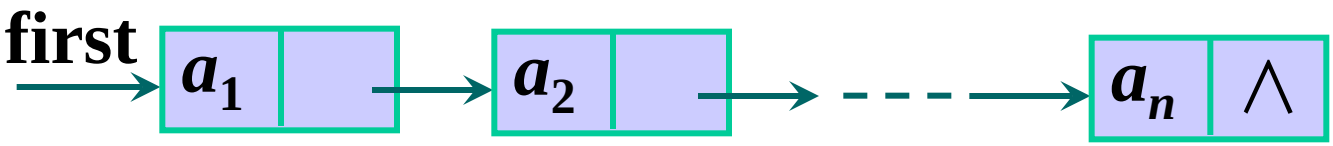


如何将空表与非空表统一？

空表

first=NULL

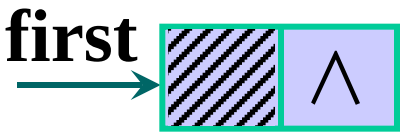
非空表



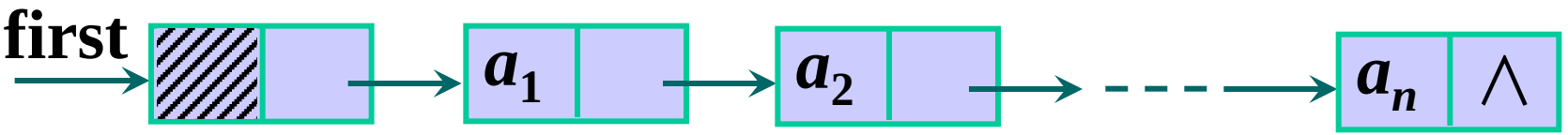
单链表结构

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，以便空表和非空表的处理算法能够统一。

空表



非空表



单链表的实现

单链表类的声明

链表的结点定义

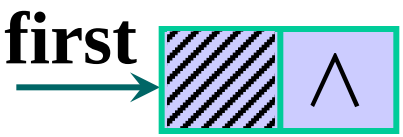
```
typedef int dataType;
struct node
{
    dataType data;
    node *next;
};
```

```
class linkList
{
public:
    linkList( );                  //建立空链表
    ~linkList( );                 //析构函数
    int length( );                //求表长
    dataType get(int i);          //取第i个元素
    int locate(dataType x);       //定位x
    void insert(int i, dataType x);//插入x
    void remove(int i);           //删除第i个元素
    void printList( );            //遍历单链表
private:
    node *first;                  //单链表的头指针
};
```

单链表的实现——构造函数

操作接口：**linkList();**

空表



算法描述：

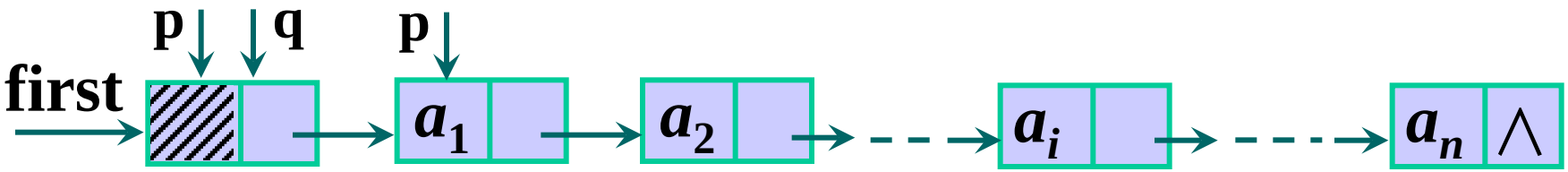
first=new node;
first->next=NULL;

```
linkList:: linkList()
{
   first=new node ;        //生成头结点
   first->next=NULL;   //尾指针初始化
}
```

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

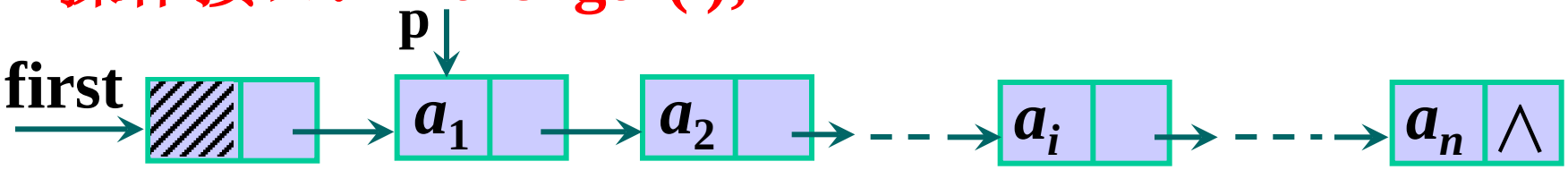
```
q=p;
p=p->next;
delete q;
```

单链表的实现——析构函数

```
linkList:: ~linkList( )
{
    node *p,*q;
    p=first;              //工作指针p初始化
    while (p)
      //释放单链表的每一个节点的存储空间
   {
        q=p;              //暂存被释放节点
        p=p->next;   //使单链表不断开
        delete q;
    }
}
```

单链表的实现——求表长

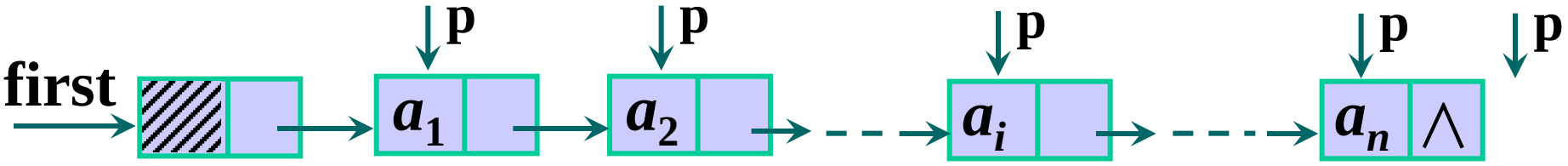
操作接口：int length( );




```
int linkList::length()
{
   node *p=first->next; int count=0;
      //p指向第一个结点，且计数器初始化为0
   while (p)
   {
        count++;
        p=p->next;    //指针下移
   }
   return count;
}
```

单链表的实现——按位查找

操作接口： dataType get(int i);



查找成功　　查找失败

基本语句：工作指针后移。

最好情况：查找第一个元素，移动0次

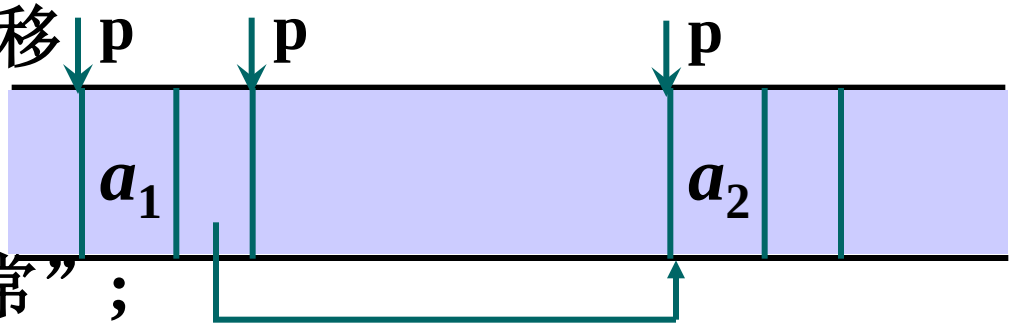
最坏情况：查找最后一个元素，移动 n-1 次

平均情况：查找第i个(1≤i ≤ n)，移动i-1次，等概率情况下，时间性能为$O(n)$。

单链表的实现——按位查找

```
dataType linkList::get(int i)
{
    node *p=first->next;      //p指向头结点
    int j=1;
    while (p && j<i)
    {
        p=p->next;  //指针后移
        j++;
    }
    if (!p)   throw “位置异常”;
    else return p->data;
}
```

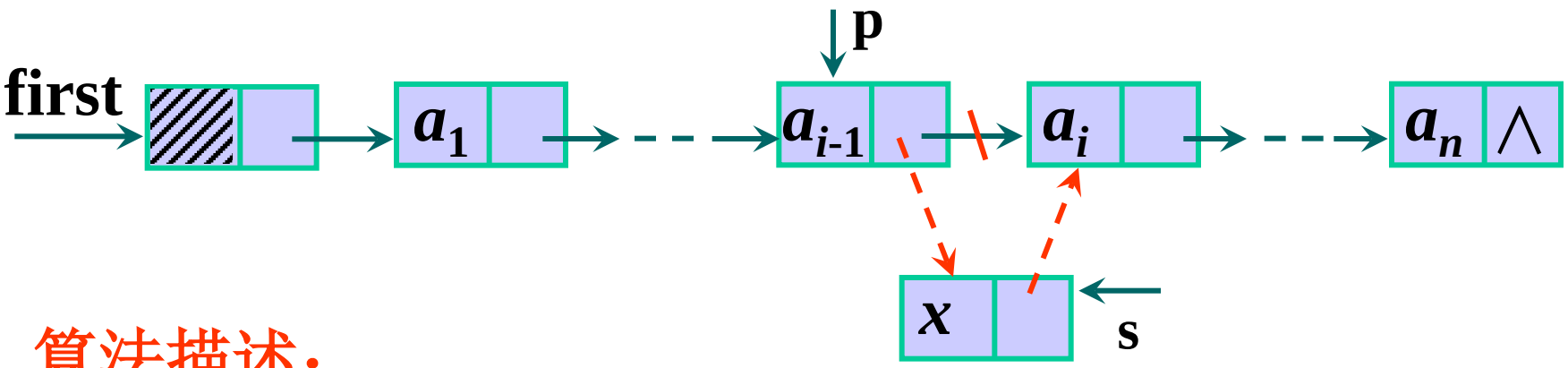
p+ +能否完成指针后移？



单链表的实现——插入

操作接口： void insert(int i, dataType x);

如何实现结点a_{i-1}、x和a_i之间逻辑关系的变化？

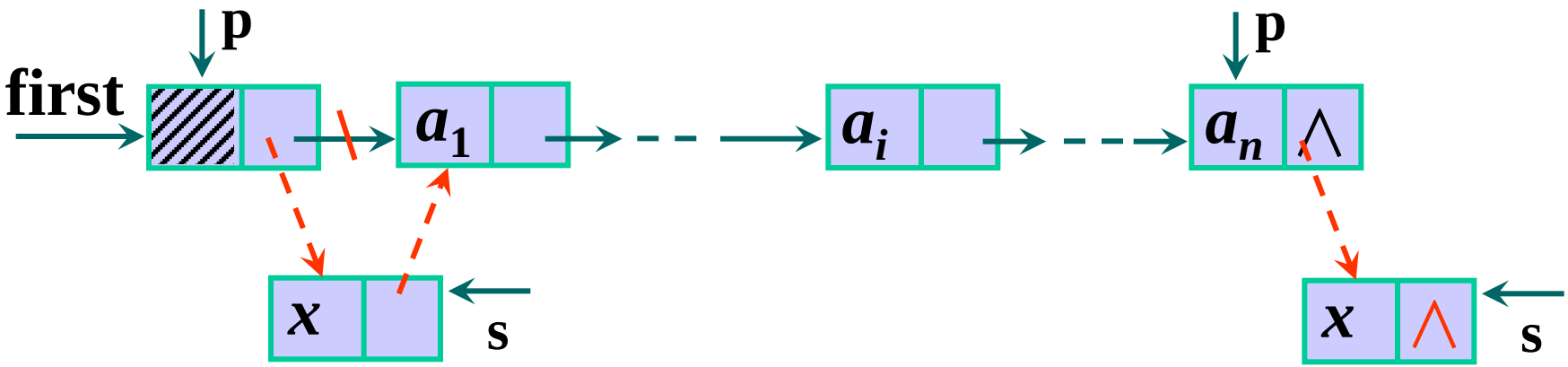


算法描述：

```
s=new node;  s->data=x;
s->next=p->next;  p->next=s;
```

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;
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。

单链表的实现——插入

```
void linkList::insert(int i, dataType x)
{
   if(i<1)throw "位置异常";
   node *p,*s;
   p=first ;  //工作指针p初始化
   int j=0;
   while(p && j<i-1)

      p=p->next;     //p后移
      j++;
   }
   if (!p) throw "位置异常";
   else {
      s=new node;
         //申请一个结点
      s->data=x;
      s->next=p->next;
      p->next=s;
   }
}
```

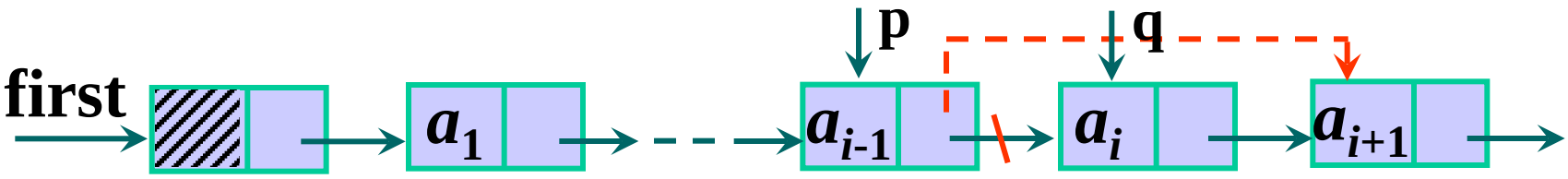
时间复杂度？

插入算法的时间主要耗费在查找正确的位置上，所以时间复杂度也是$O(n)$。

单链表的实现——删除

操作接口： **void remove(int i);**

如何实现结点a_{i-1}和a_i之间逻辑关系的变化？

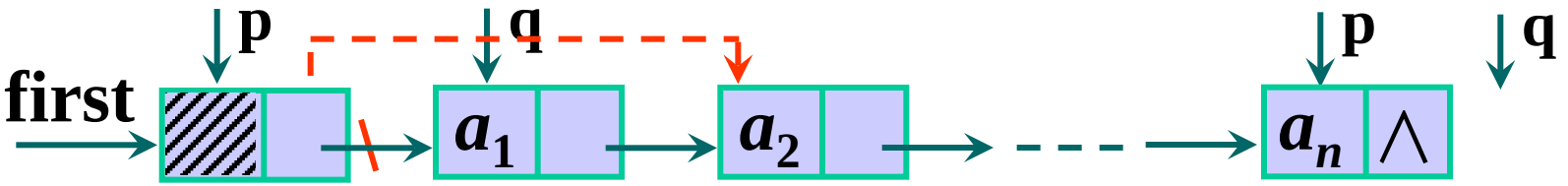


算法描述：

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

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;
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。

单链表的实现——删除

```
void linkList::remove(int i)
{
   if(i<1)throw “位置异常”；
   node *p,*q;
   p=first ;  //工作指针p初始化
   int j=0;
   while (p && j<i-1)
   {
       p=p->next;    //p后移
       j++;
    }
   if ( !p->next)
       throw “位置异常”；
          //p的后继不存在
   else {
       q=p->next;
       p->next=q->next;
          //摘链
       delete q;
    }
}
```

删除算法的时间主要耗费在查找正确的位置上，所以时间复杂度也是$O(n)$。

链表的应用举例（和顺序表相同）

```
int main(){
      //输入：用户提供数值n，代表n个乘客
      //输出：用户指定位置上的乘客编号（或其它数据项）
      //passengers的数据类型由seqlist变为linkList，其他不变
  int n;
  int item;
  linkList passengers; //定义一个表，表明是passengers
  cout<<“输入乘客人数n”<<endl;
  cin>>n;
  cout<<“按任意顺序输入乘客编号”<<endl;
  for (int i=0; i<n; i++){
     cin>>item;
     passengers.insert(i,item);
  }
```

链表的应用举例（续1）

```
  cout<<endl<<endl;
  cout<<“请输入一个位置号：”
  cin>>n;
  while(n!=-1){
     if (n>=1&&n<=passengers.length()){
        cout<<“位于该位置上的乘客编号是：”;
        cout<<passengers.get(n)<<endl;}
     else
        cout<<“该位置已经超出表的范围”<<endl;
     cout<<“请输入一个位置号（-1结束）：”;
     cin>>n;
  }
  cout<<endl;
  return 0;
}
```

顺序表和单链表的比较

- 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反映。
- 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，数据元素之间的逻辑关系用指针来反映。

顺序表和单链表的比较

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：

- 顺序表是随机存取，查找的时间性能为 $O(1)$。
- 单链表是顺序存取，查找的时间性能为 $O(n)$。

插入和删除：

- 顺序表需平均移动一半的元素，时间性能为 $O(n)$。
- 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 $O(1)$。

顺序表和单链表的比较

空间性能是指某种存储结构所占用的存储空间的大小。

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。

结论

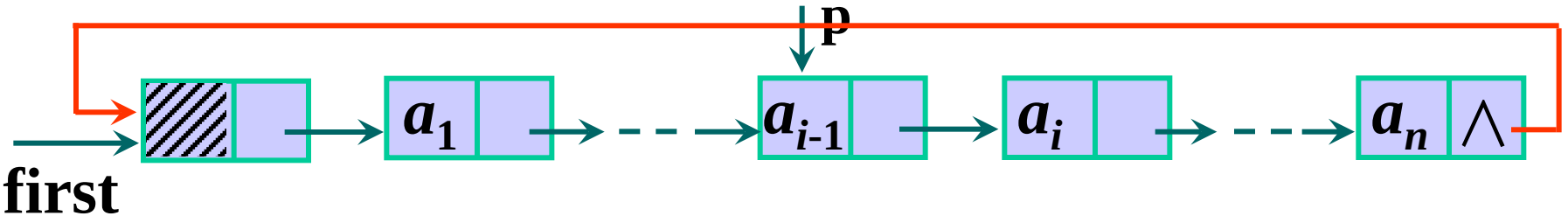
- 若线性表需频繁查找却很少插入和删除操作，或其操作和元素在表中的位置密切相关时，宜采用顺序表作为存储结构；若线性表需频繁插入和删除时，则宜采用单链表做存储结构。

- 当线性表中元素个数变化较大或者未知时，最好使用单链表实现；而如果用户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，各有其优缺点，要根据实际问题的具体需要，加以综合平衡，才能选定比较适宜的存储结构。

2.4 线性表的其它存储方法

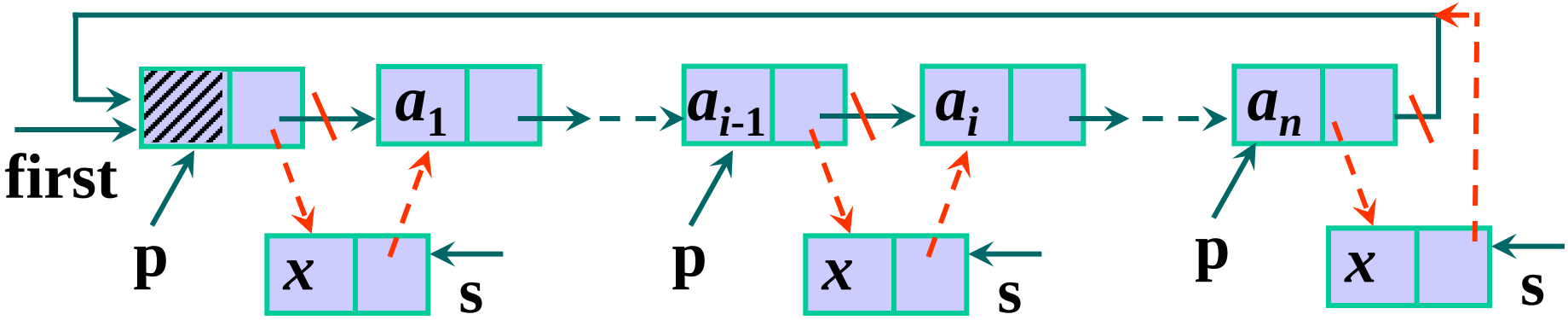
循环链表(circular linked list)



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。

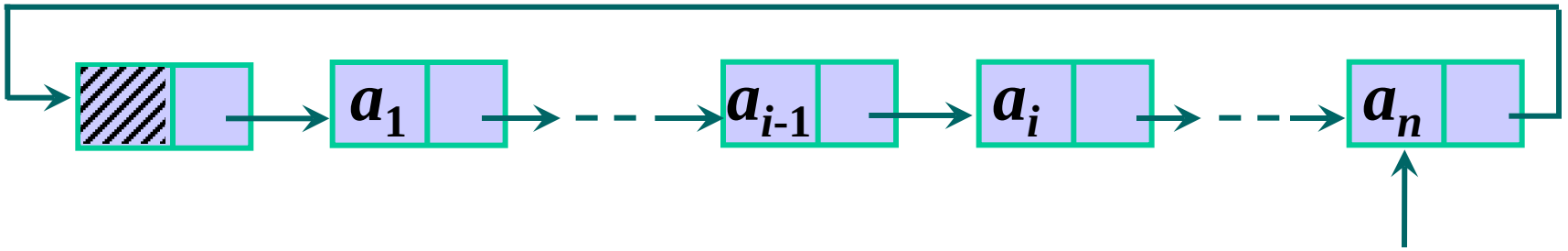
循环链表——插入



算法描述：

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

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。

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。

双链表

定义结点结构：


```
struct DulNode
{
    dataType data;
    DulNode<dataType> *prior, *next;
};
```

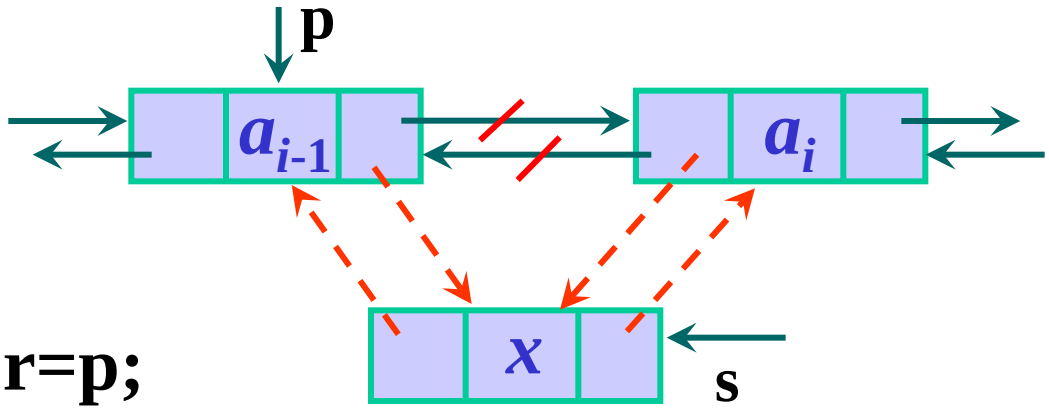
启示？

时空权衡——空间换取时间

双链表的操作——插入

操作接口：

void Insert(DulNode<dataType> *p, dataType x);



s->prior=p;

s->next=p->next;

p->next->prior=s;

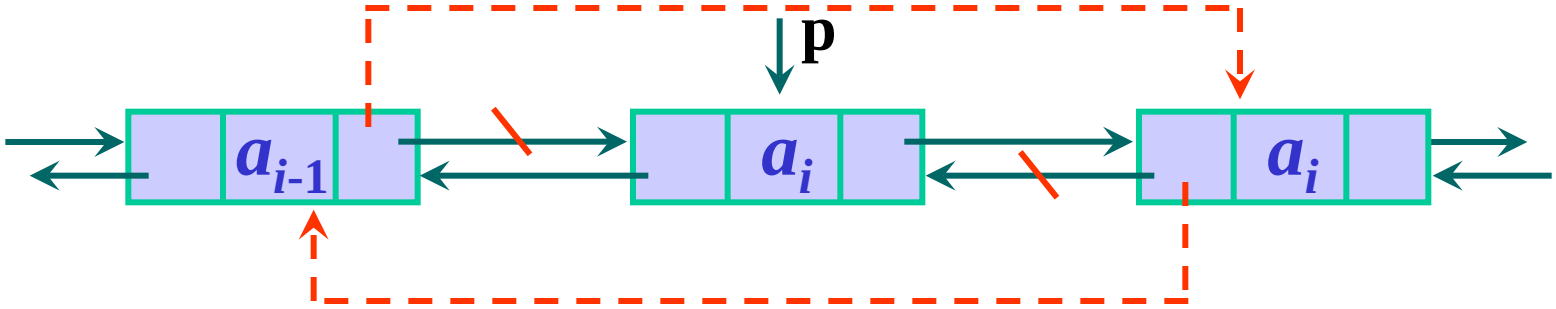
p->next=s;

注意指针修改的相对顺序

双链表的操作——删除

操作接口：

dataType Delete(DulNode<dataType> *p);



（p->prior）->next=p->next;

（p->next）->prior=p->prior;

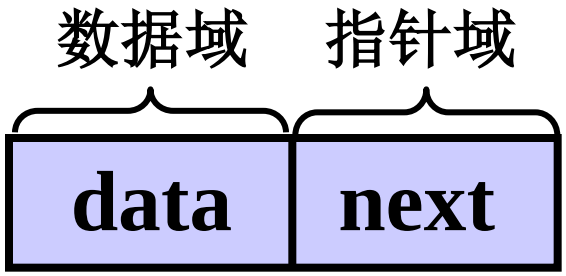
结点p的指针是否需要修改？

delete p;

静态链表(static linked list)

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：

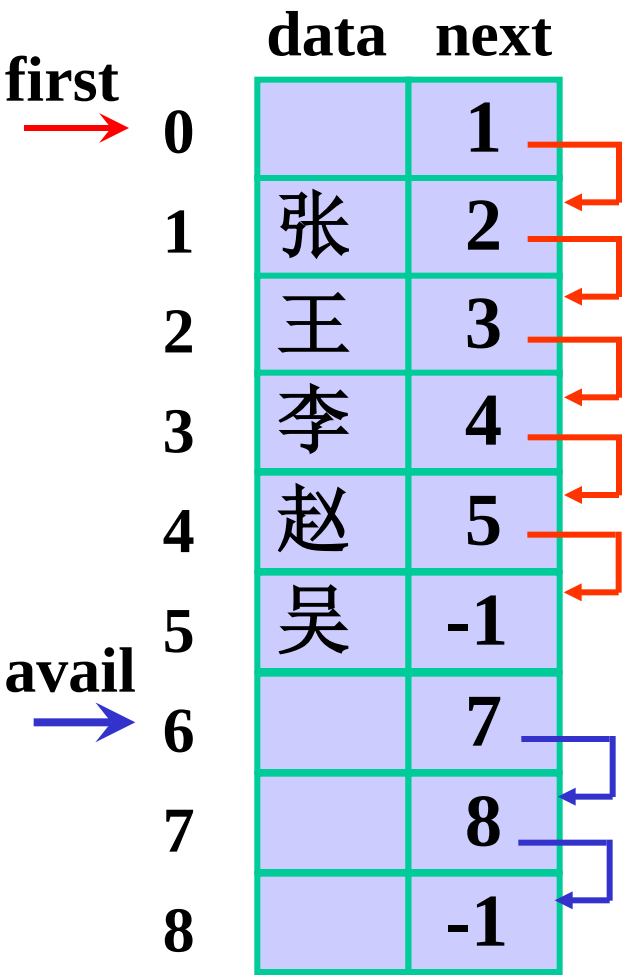


data：存储放数据元素；

next：也称游标，存储该元素的后继在数组的下标。

静态链表

例：线性表(张，王，李，赵，吴)的静态链表存储



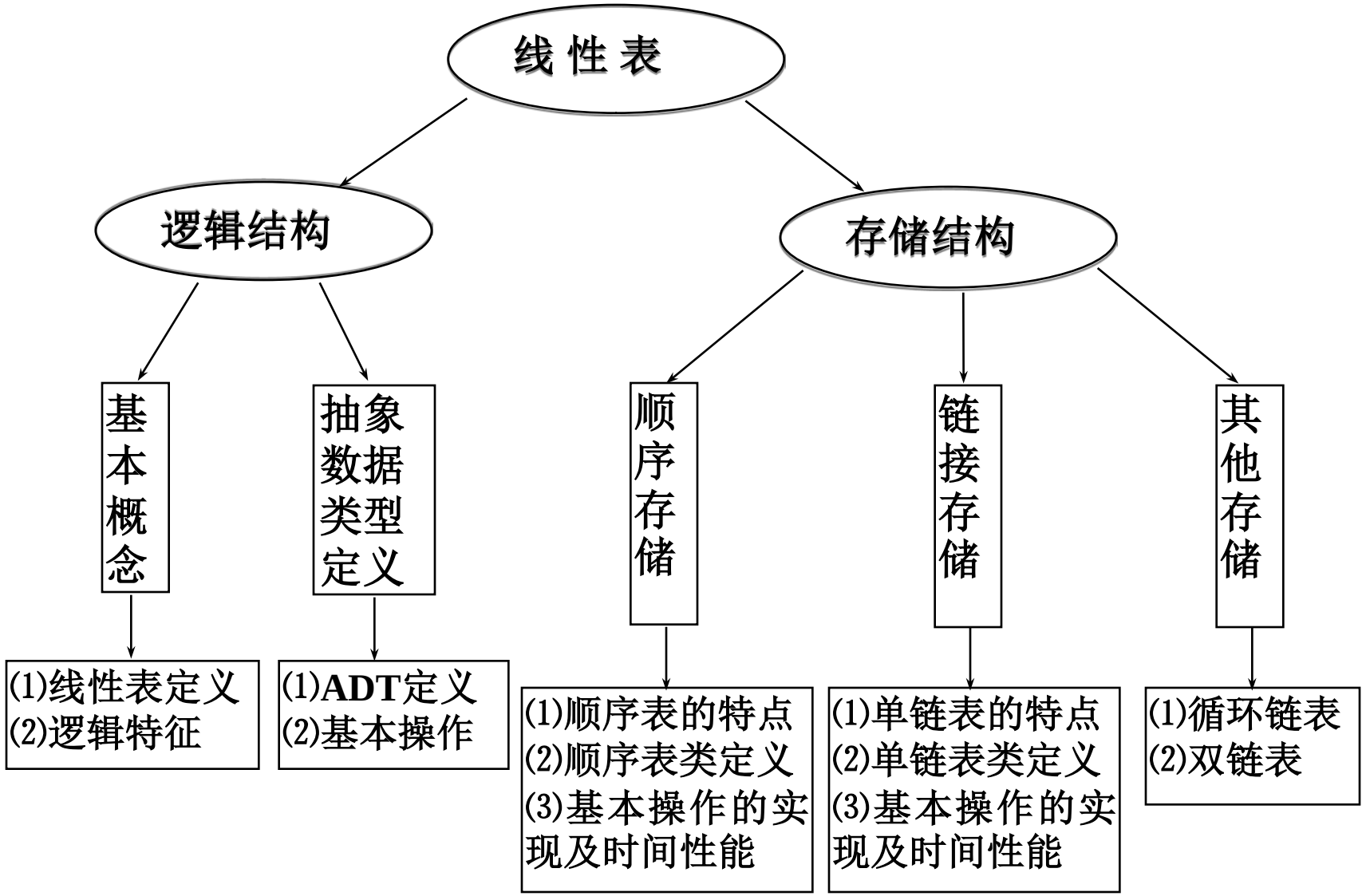
first：静态链表头指针，为了方便插入和删除操作，通常静态链表带头结点；

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 应用举例

- 电话簿（查找某一个人的电话号）
- 电子字典（由于字典的单词数量可多可少，并不能事先就完全规范下来）
- Josephus问题（设有n个人围坐在一个圆桌周围，现从第s个人开始报数，数到第m的人出列，然后从出列的下一个人重新开始报数，数到第m的人又出列，…，如此反复直到所有的人全部出列为止。Josephus问题是：对于任意给定的n, s和m，求出按出列次序得到的n个人员的序列。）
- 多项式加法运算（课堂讨论）

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
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

本章作业

习题 2 （P53）：4(1)(2)、5(1)(2)(3)(4)(6)(7)