第2章 线性表

学习目标

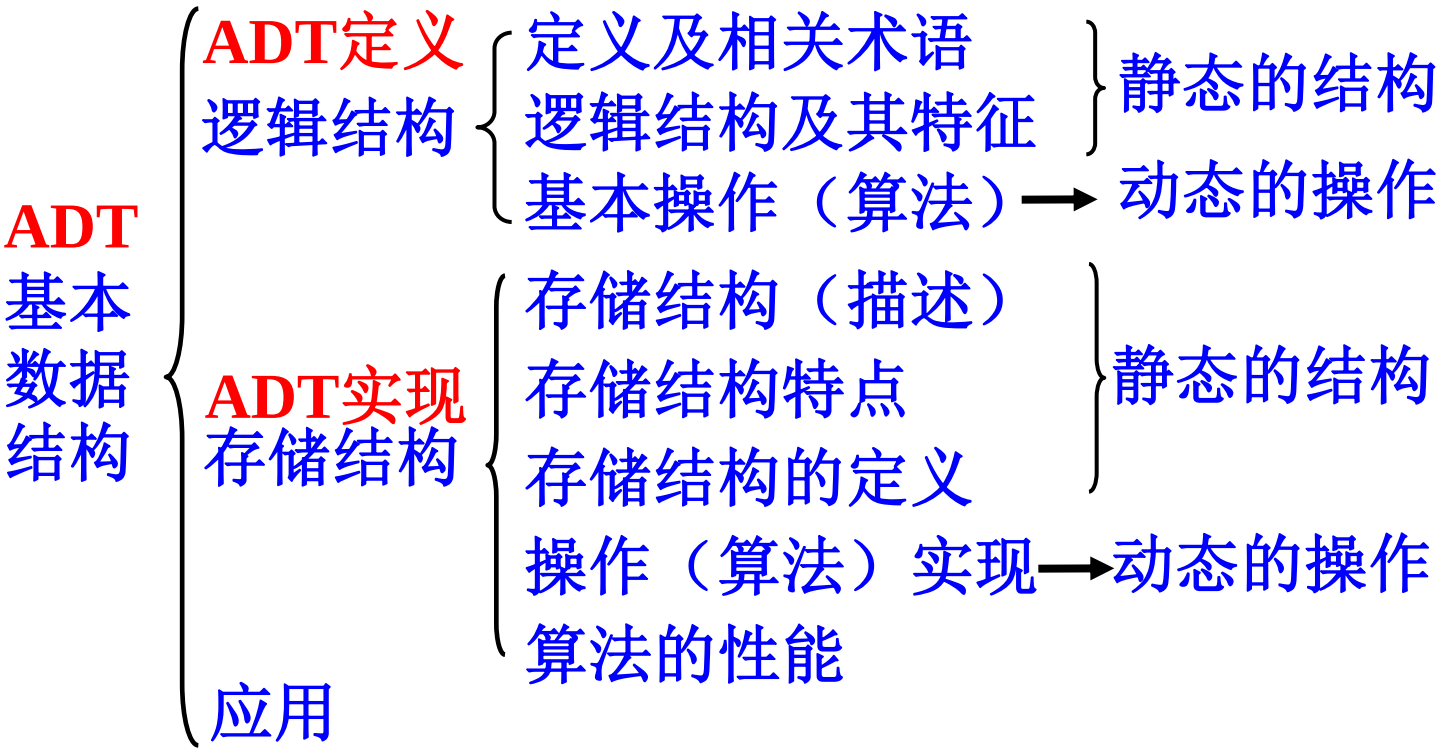
- 掌握线性表的逻辑结构，线性表的顺序存储结构和链式存储结构的描述方法；熟练掌握线性表在顺序存储结构和链式存储结构的结构特点以及相关的查找、插入、删除等基本操作的实现；并能够从时间和空间复杂性的角度综合比较两种存储结构的不同特点
- 掌握栈和队列的结构特性和描述方法，熟练掌握栈和队列的基本操作的实现，并且能够利用栈和队列解决实际应用问题
- 掌握串的结构特性以及串的基本操作，掌握针对字符串进行操作的常用算法和模式匹配算法
- 掌握多维数组的存储和表示方法，掌握对特殊矩阵进行压缩存储时的下标变换公式，了解稀疏矩阵的压缩存储表示方法及适用范围
- 了解广义表的概念和特征

本章主要内容

- 2.1 线性表的逻辑结构
- 2.2 线性表的存储结构
- 2.3 栈（Stack）
- 2.4 队列（Queue）
- 2.5 串（String）
- 2.6 数组（Array）
- 2.7 广义表（Generalized List）
- 本章小结

本章的知识点结构

- 基本的数据结构（ADT）
  - 线性表、栈、队列、串、（多维）数组、广义表
- 知识点结构



2.1 线性表的逻辑结构

- 线性表的定义：
  - 是由n（n≥0）个性质（类型）相同的序列。
  - 记为：
    - L=（a_1，a_2，a_3，……a_{i-1}，a_i，a_{i+1}，……，a_n）
  - a_i（1≤i≤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 。
  - n为线性表中元素个数，称为线性表的长度；
    - 当n=0时，为空表，记为L=（ ）。
  - 图示表示：
    - 线性表L=（a_1，a_2，…a_i，……，a_n）的图形表示如下：

2.1 线性表的逻辑结构(Cont.)

- 逻辑特征：L=（a_1，a_2，a_3，……a_{i-1}，a_i，a_{i+1}，…，a_n）
  - 有限性:
    - 线性表中数据元素的个数是有穷的。
  - 相同性:
    - a_i为线性表中的元素元素类型相同
  - 相继性:
    - a_1为表中第一个元素，无前驱元素；a_n为表中最后一个元素，无后继元素；
    - 对于…a_{i-1},a_i,a_{i+1}…(1<i<n)，称a_{i-1}为a_i的直接前驱， a_{i+1}为a_i的直接后继。
    - 中间不能有缺项。

2.1 线性表的逻辑结构(Cont.)

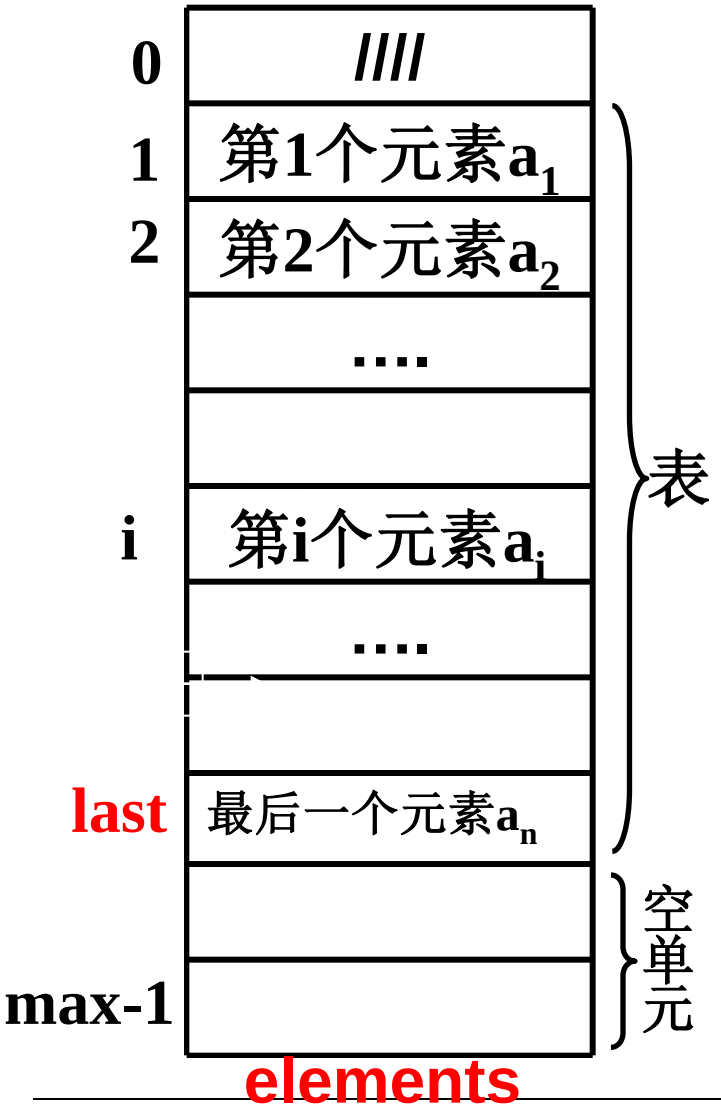
- 定义在线性表的操作（算法）：
  - 设L是类型为LIST线性表实例，x 的型为ElemType的元素实例，p 为位置变量。所有操作描述为：
    - ① Insert(x, p, L)
    - ② Delete(p, L)
    - ③ Locate(x, L)
    - ④ Retrieve(p, L)
    - ⑤ Previous(p, L)
    - ⑥ Next(p, L)
    - ⑦ MakeNull(L)
    - ⑧ First(L)
    - ⑨ END(L)

2.1 线性表的逻辑结构(Cont.)

- ADT应用举例：
  - 设计函数 DeleteVal（LIST &L, ElemType d)， 其功能为删除 L 中所有值为 d 的元素。
  - void DeleteVal(LIST &L, ElemType d)

```
{    position p ;
     p = First( L ) ;
     while ( p != End( L ) )
     {    if ( Same( Retrieve( p, L ), d ) )
               Delete(p, L ) ;
          else
               p = Next(p, L ) ;
     }
}
```

2.2线性表的存储结构----顺序表



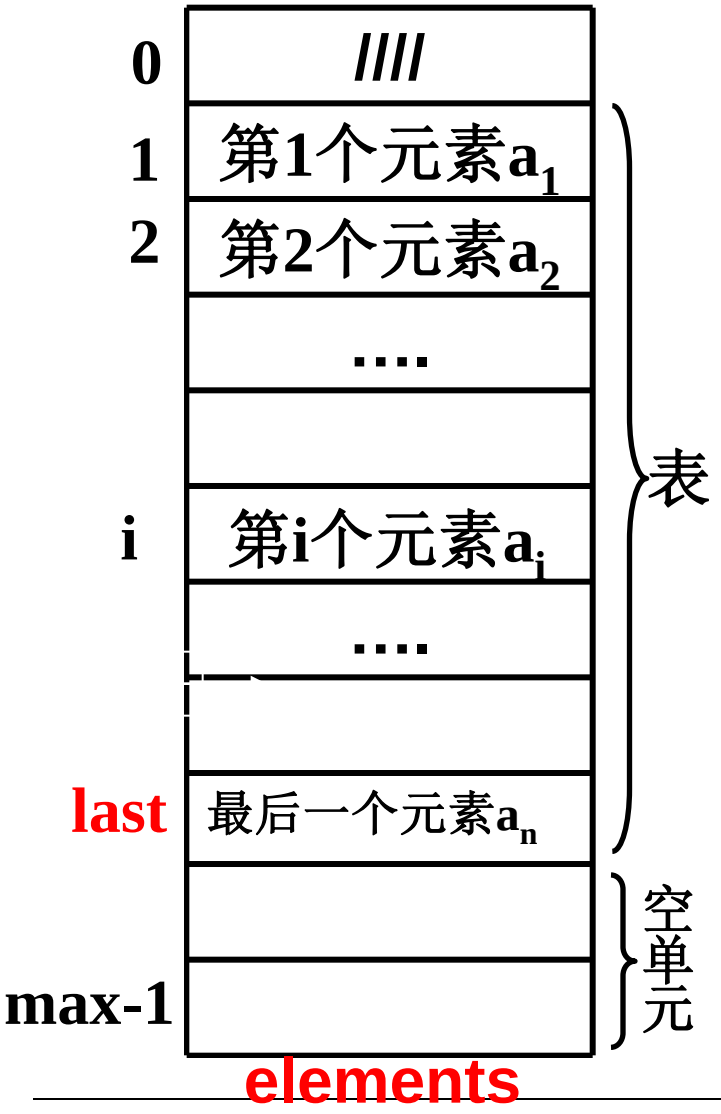
➢2.2.1 顺序表：

- 把线性表的元素按照逻辑顺序依次存放在数组的连续单元内；
- 再用一个整型量表示最后一个元素所在单元的下标，即表长。

➢存储结构特点：

- 元素之间逻辑上的相继关系，用物理上的相邻关系来表示（用物理上的连续性刻画逻辑上的相继性）
- 是一种随机访问存取结构，也就是可以随机存取表中的任意元素，其存储位置可由一个简单直观的公式来表示。

2.2线性表的存储结构--顺序表(Cont.)



➢存储结构定义：

■类型定义：

```
#define  max  100
struct LIST{
    ElemType elements[max];
    int last;
};
```

■位置类型：

```
typedef  int  position;
```

■线性表的实例L：LIST L;

■元素和长度：

L.elements[p] // L的第p个元素

L.last L的长度，最后元素的位置

2.2线性表的存储结构--顺序表(Cont.)

- 操作的实现----插入操作
  - 操作接口void Insert (ElemType x, position p, LIST &L)
  - 插入前：$(a_1, \ldots, a_{p-1}, a_p, \ldots, a_n)$
  - 插入后：$(a_1, \ldots, a_{p-1}, x, a_p, \ldots, a_n)$

a_{p-1}和a_p之间的逻辑关系发生了变化

↓

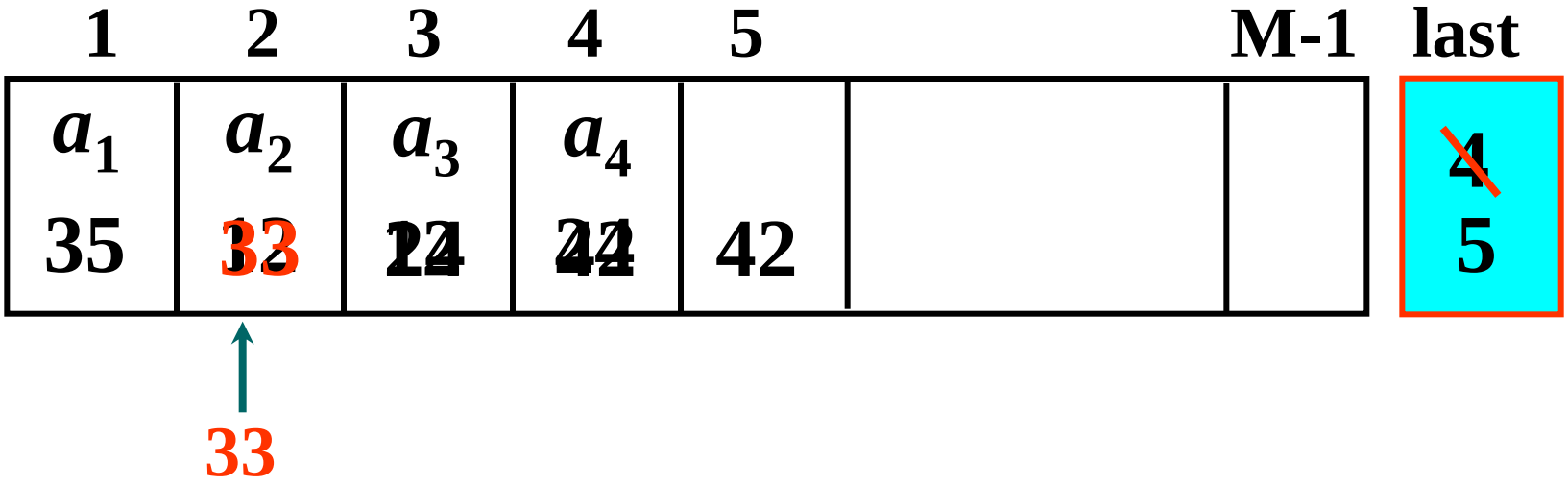
顺序存储要求存储位置反映逻辑关系

↓

存储位置要反映这个变化

2.2线性表的存储结构--顺序表(Cont.)

- 例:（35，12，24，42），在p=2的位置上插入33。

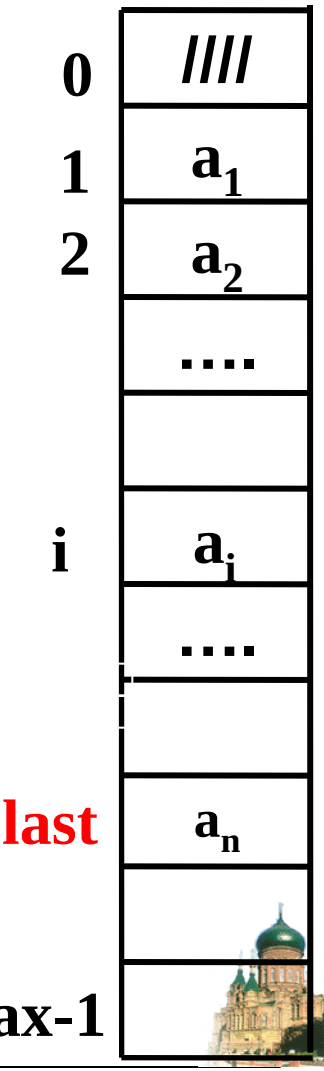


- 什么时候不能插入？
  - 表满时：L.last >=Max
  - 合理的插入位置：1 ≤ p ≤ L.last+1

➡ 注意边界条件

2.2线性表的存储结构--顺序表(Cont.)

```
void Insert ( ElemType x, position p, LIST &L)// ①
{ position q ;
  if (L.last >= Max – 1)
        cout<< “ 表满 ” ;
  else if (( p > L.last +1 ) || ( p < 1) )
        cout<< “ 指定位置不存在 ” ;
  else {
     for ( q = L.last; q >= p; q -- )
          L.elements[ q+1] = L.elements[ q ] ;
      L.elements[ p ] = x ;
      L.last = L.last + 1 ;
   }
}
```

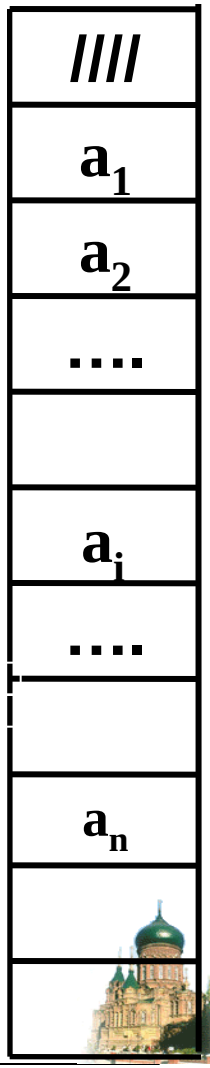


2.2线性表的存储结构--顺序表(Cont.)

- 时间性能分析
  - 基本语句？
  - 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$O(n)$。
  - 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$O(n)$。



2.2线性表的存储结构--顺序表(Cont.)

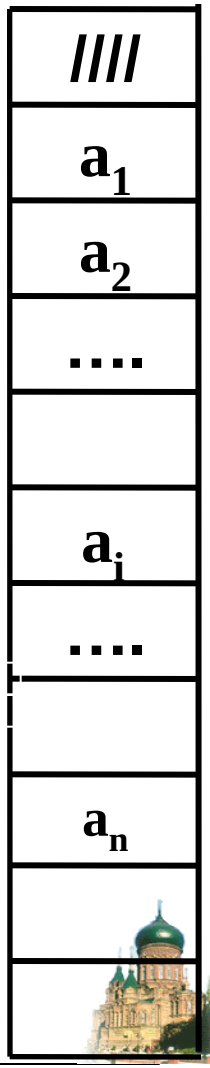
- 操作的实现----删除操作
  - 操作接口：void Delete(position p, LIST &L)
  - 删除前：$(a_1, \dots, a_{p-1}, a_p, a_{p+1}, \dots, a_n)$
  - 删除后：$(a_1, \dots, a_{p-1}, a_{p+1}, \dots, a_n)$
- 例：（35, 33, 12, 24, 42），删除p=2的数据元素。

1	2	3	4	5		last
a_1	a_2	a_3	a_4	a_5		5
35	33 12	12 24	24 42	42		4

- 分析边界条件？
- 操作算法的实现
- 时间性能分析

2.2线性表的存储结构--顺序表(Cont.)

```
void Delete( position p, LIST  &L) //②
{   position  q ;
    if  ( ( p > L.last ) || ( p < 1 ) )
        cout<< “指定位置不存在” ;
    else{
        L.last = L.last – 1;
        for ( q = p ; q <= L.last ; q ++ )
            L.elements[ q ] = L.elements[ q + 1 ];
    }
}
```



- 最好、最坏和平均移动元素个数：
- 时间复杂性：O(n)

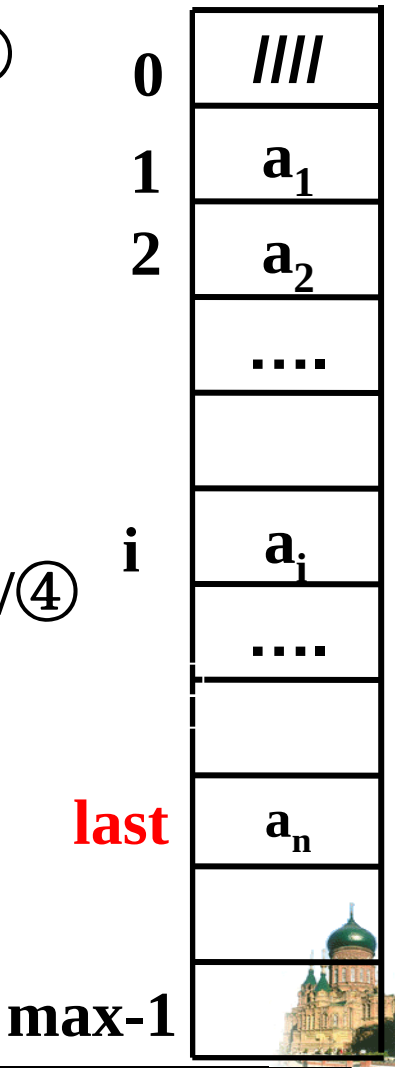
2.2线性表的存储结构--顺序表(Cont.)

- 其他操作的实现
  - position Locate (ElemType x , LIST L) //③

```
{  position  q ;
   for ( q = 1; q <= L.last ;  q++ )
      if ( L.elements[ q ] == x )
         return ( q ) ;
   return ( L.last + 1 );
} //时间复杂性：O(n)
```

  - ElemType Retrieve (position p , LIST L) //④

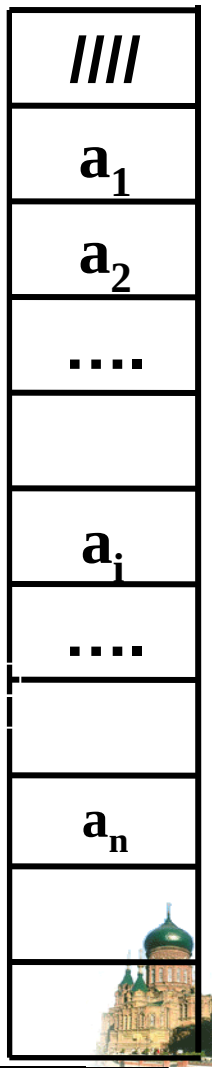
```
{  if  ( p > L.last  )
         cout<< "指定位置不存在"  ;
   else
         return ( L.elements[ p ] ) ;
} //时间复杂性：O(1)
```



2.2线性表的存储结构--顺序表(Cont.)

- 其他操作的实现

```
position Previous( position p , LIST L ) //⑤
{ if ( ( p <= 1 ) || ( p > L.last+1 ) )
      cout<< "前驱位置不存在";//return 0;
  else
     return ( p – 1 );
} //时间复杂性：O(1)
position Next( position p , LIST L ) //⑥
{   if ( ( p < 1 ) || ( p >= L.last ) )
      cout<< "前驱位置不存在" ;//return L.last+1;
    else
      return ( p + 1 );
} //时间复杂性：O(1)
```



2.2线性表的存储结构--顺序表(Cont.)

- 其他操作的实现
  - position MakeNull(LIST &L) //⑦

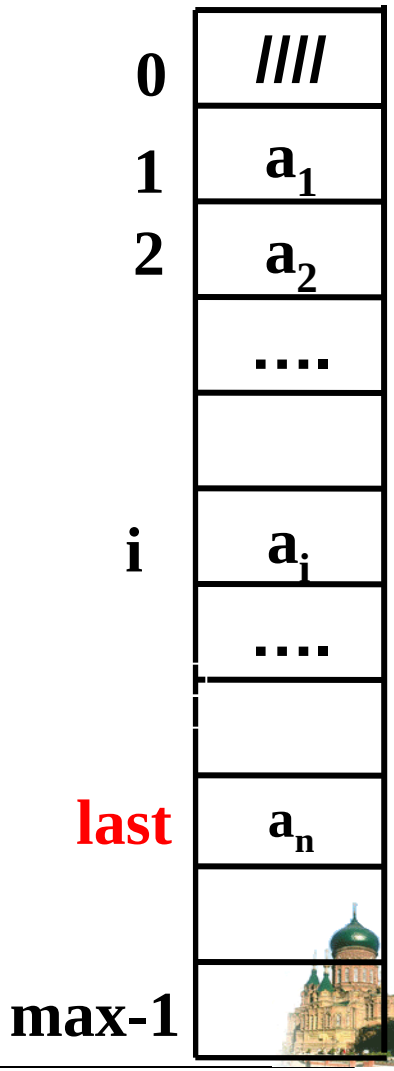
```
{   L.last = 0 ;
    return ( L.last +1 );
} //时间复杂性：O(1)
```

  - position First(LIST L) //⑧

```
{  if ( L.last>0 )   return ( 1 );
   else cout<<“表为空”;//return 0;
} //复杂性：O(1)
```

  - position End(LIST L) //⑨

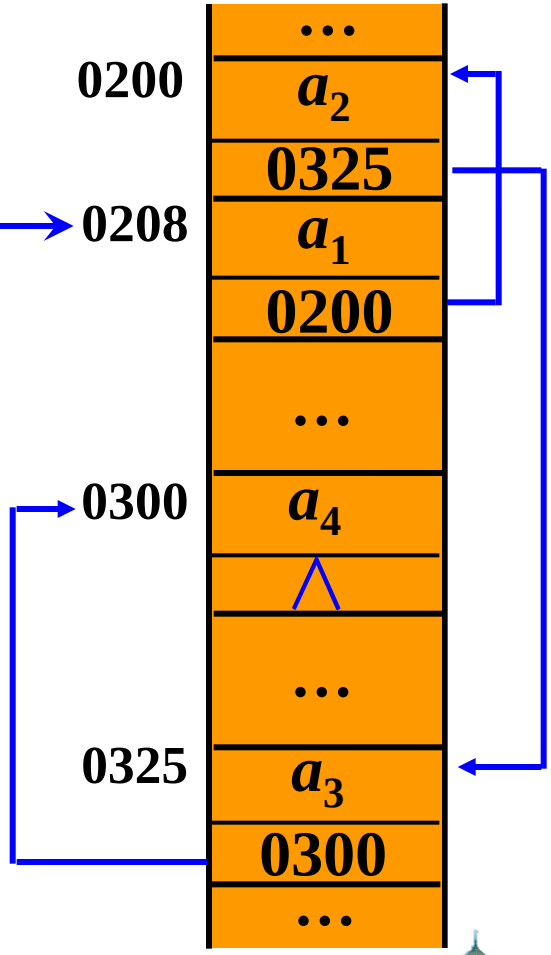
```
{ return( L.last + 1 );
} // O(1)
```



2.2线性表的存储结构--链接表

2.2.2 单链表

- 单链表:
  - 一个线性表由若干个结点组成，每个结点均含有两个域：存放元素的信息域和存放其后继结点的指针域，这样就形成一个单向链接式存储结构，简称单向链表或单链表。
  - 例： $(a1, a2, a3, a4)$的存储示意图



- 存储结构特点
  - 逻辑次序和物理次序不一定相同；
  - 元素之间的逻辑关系用指针表示；
  - 需要额外空间存储元素之间的关系
  - 非随机访问存取结构（顺序访问）

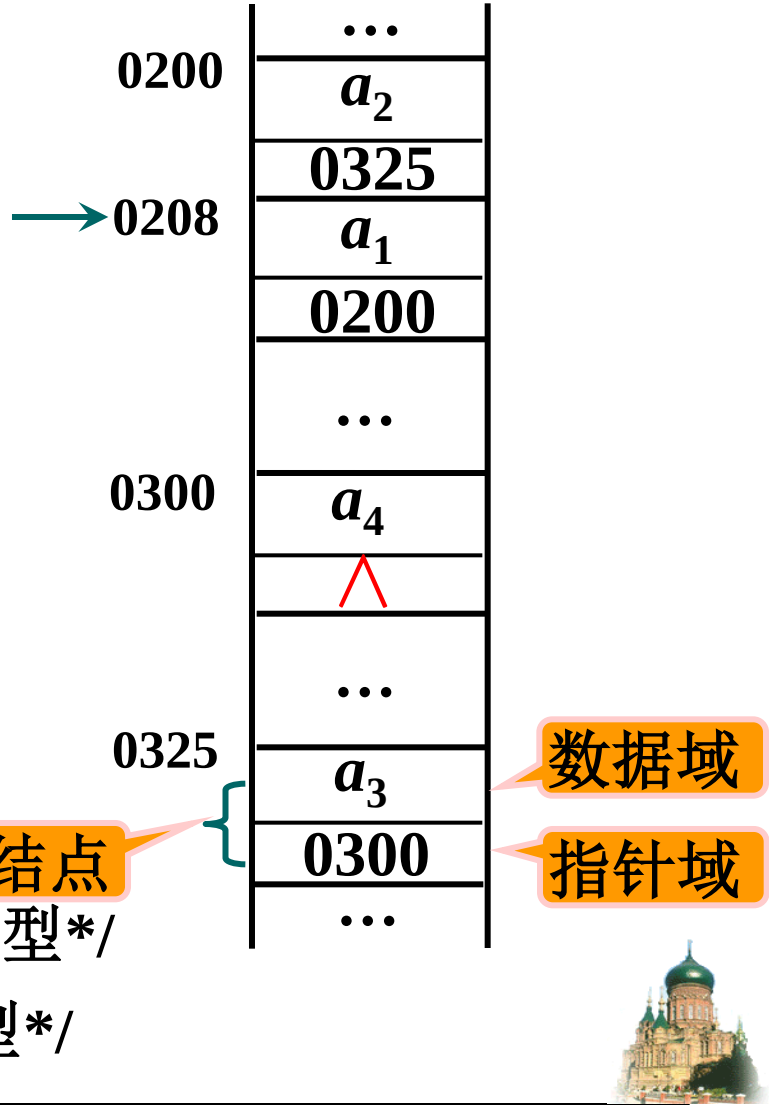
2.2线性表的存储结构--链接表(Cont.)

- 存储结构定义:
  - 结点结构:

数据域	指针域
data	next

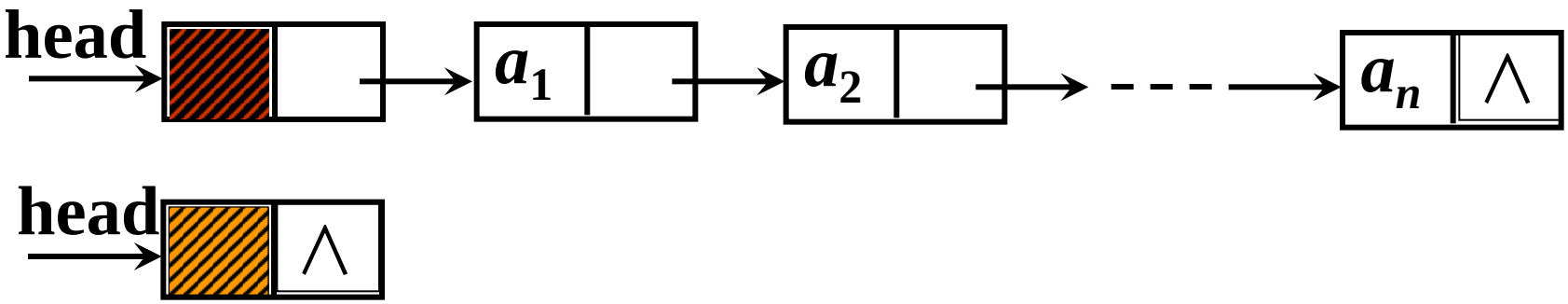
  - 存储结构类型定义

```
struct   celltype  {
        ElemType  data ;
        celltype  *next ;
} ; /*结点型*/
typedef  celltype *LIST; /*线性表的型*/
typedef  celltype  *position; /*位置型*/
```

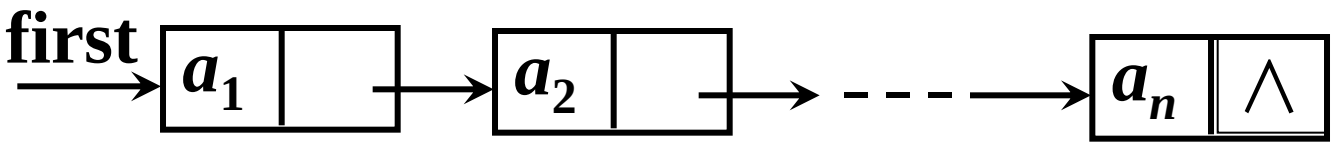


2.2线性表的存储结构--链接表(Cont.)

- 存储结构定义:
  - 单链表图示:
    - 带表头结点的单链表



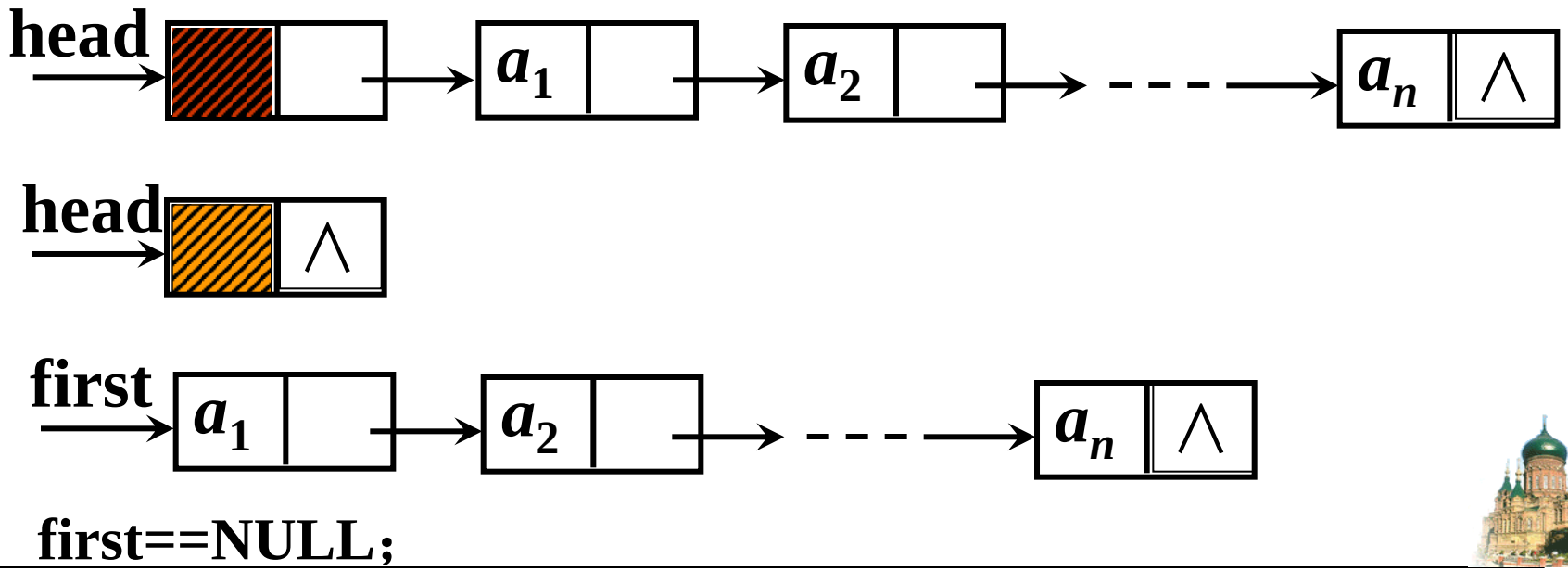
    - 不表带头结点的单链表



first==NULL;

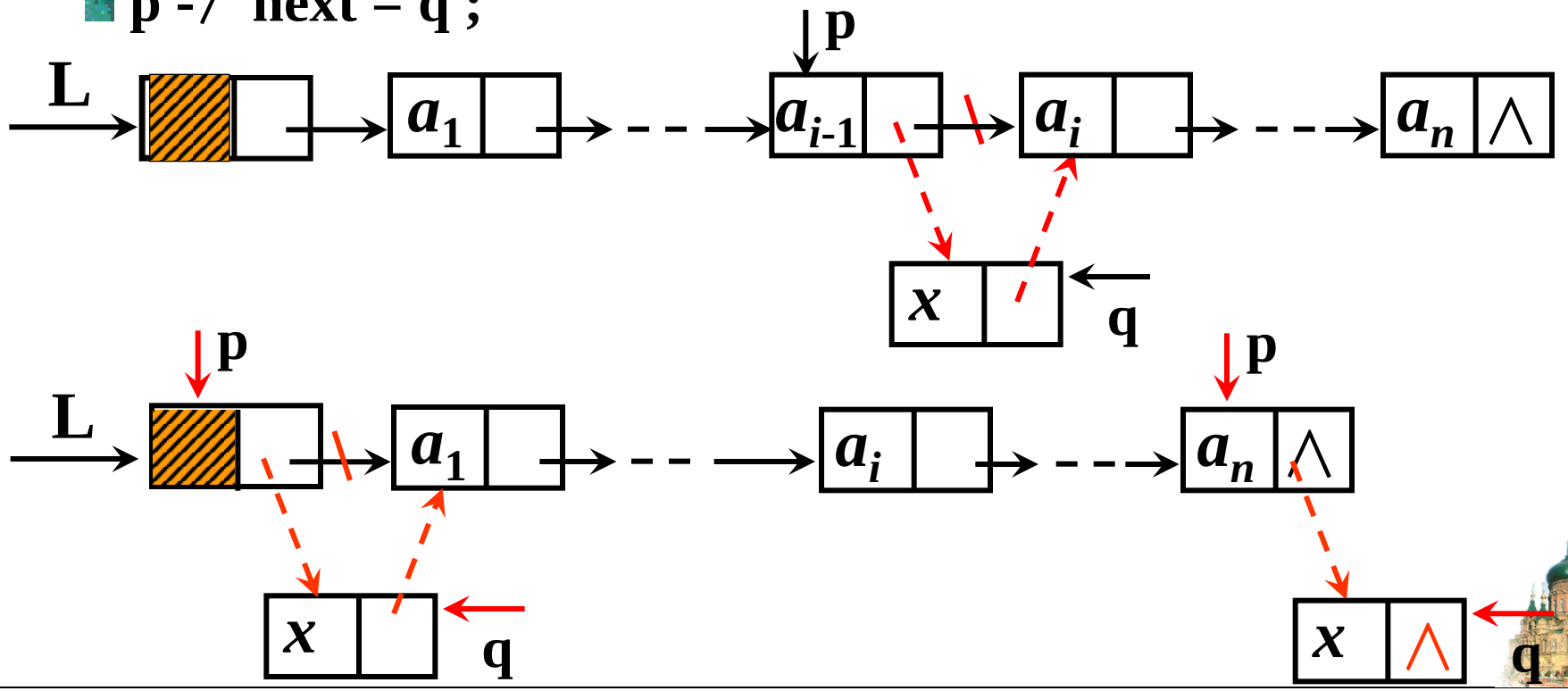
2.2线性表的存储结构--链接表(Cont.)

- 表头结点的作用：
  - 空表和非空表表示统一
  - 在任意位置的插入或者删除的代码统一
  - 注意：是否带表头结点在存储结构定义中无法体现，由操作决定



2.2线性表的存储结构--链接表(Cont.)

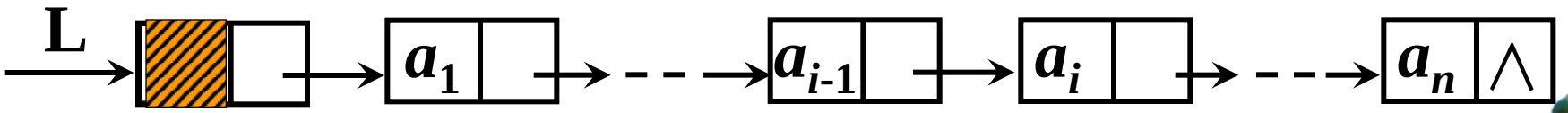
- 操作的实现---- ①插入操作
  - q = new celltype ;
  - q -〉 data = x ;
  - q -〉 next = p-〉 next ;
  - p -〉 next = q ;



2.2线性表的存储结构--链接表(Cont.)

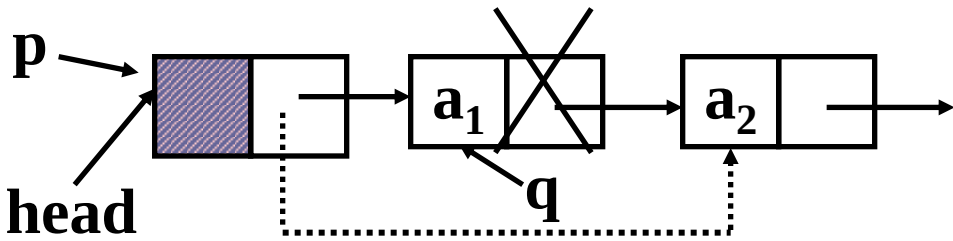
- 操作的实现---- ①插入操作

```
void  Insert ( ElemType  x, position  p, LIST  &L ) //①
{     position  q ;
      q = new  celltype ;
      q →data = x ;
      q →next = p →next ;
      p →next = q ;
} //时间复杂性：O(1)
```



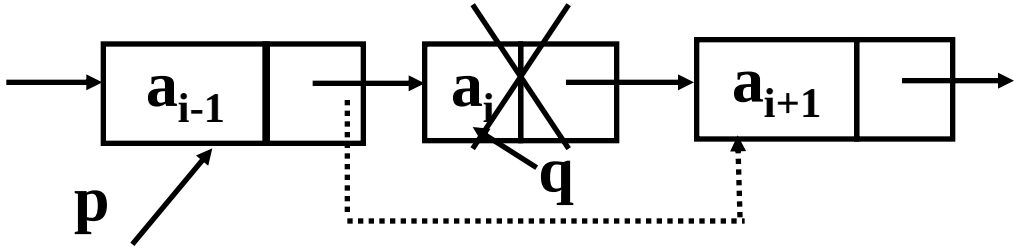
2.2线性表的存储结构--链接表(Cont.)

- 操作的实现---- ②删除操作
  - q = p→next ;
  - p→next = q→next ;
  - delete q ;

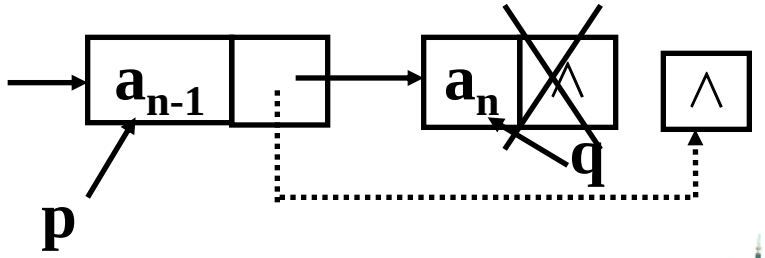


(a) 删除第一个元素

```
void Delete ( position  p, LIST  &L )
{   position  q ;
    if  ( p→next != NULL ) {
        q = p →next ;
        p →next = q →next ;
        delete  q ;
}
```



(b) 删除中间元素



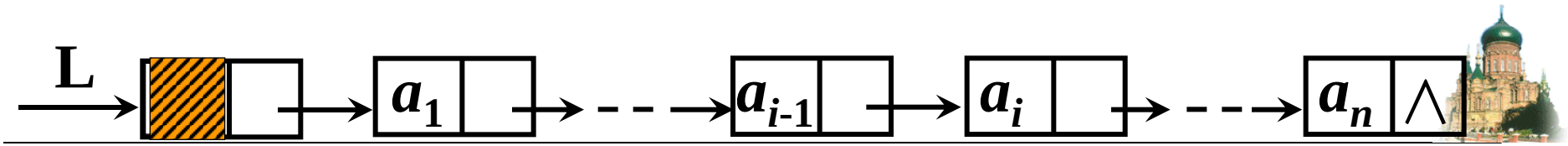
(c) 删除表尾元素

时间复杂性：O（1）

2.2线性表的存储结构--链接表(Cont.)

- 操作的实现---- ③Locate (ElemType x, LIST L)
  - position Locate (Elementtype x, LIST L)

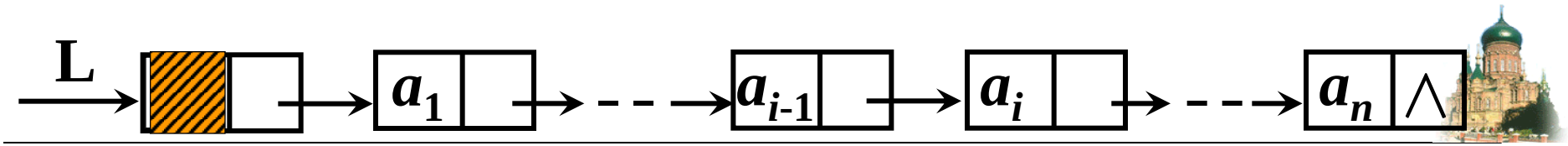
```
{  position p ;
   p = L ;
   while ( p→next != NULL )
       if ( p→next→data == x )
           return p ;
       else
           p = p→next ;
   return p ;
} //时间复杂性：O(n)
```



2.2线性表的存储结构--链接表(Cont.)

- 操作的实现---- ④ElemType Retrieve(position p , LIST L)
  - ElemType Retrieve (position p , LIST L)

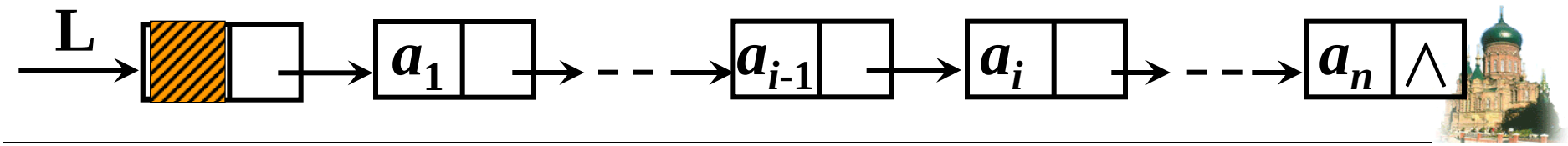
```
{
    return ( p →next →data );
} //时间复杂性：O(1)
```



2.2线性表的存储结构--链接表(Cont.)

- 操作的实现---- ⑤ position Previous(position p,LIST L)
  - position Previous (position p, LIST L)

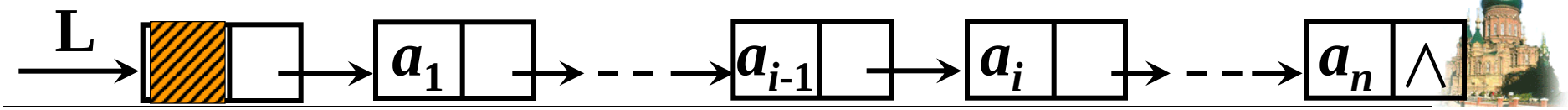
```
{   position q ;
    if ( p == L→next )
        cout << "不存在前驱位置" ;
    else {
        q = L ;
        while ( q→next != p ) q = q→next ;
        return q ;
    }
} //时间复杂性：O(n)
```



2.2线性表的存储结构--链接表(Cont.)

- 操作的实现---- ⑥ position Next (position p, LIST L)
  - position Next (position p, LIST L)

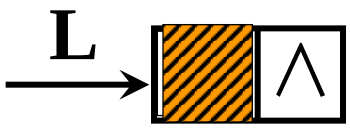
```
{   position q ;
    if ( p→next == NULL )
        cout<< "不存在后继位置" ;
    else {
        q = p→next;
        return q ;
    }
} //时间复杂性：O(1)
```



2.2线性表的存储结构--链接表(Cont.)

- 操作的实现---- ⑦ position MakeNull (LIST &L)
  - position MakeNull (LIST &L)

```
{
    L = new celltype ;
    L→next = NULL;
    return L ;
} //时间复杂性：O(1)
```



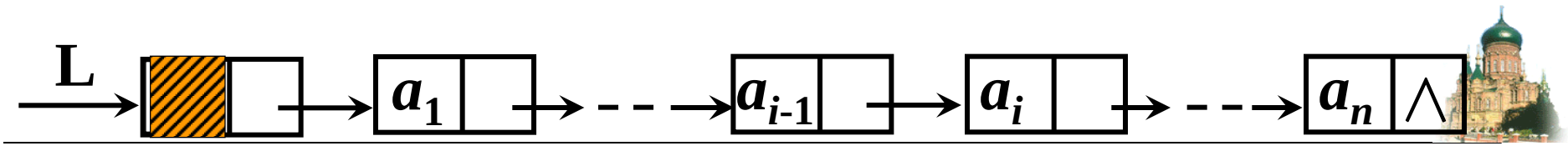
- 操作的实现---- ⑧ position First (LIST L)
  - position First (LIST L)

```
{
    return L;
} //时间复杂性：O(1)
```

2.2线性表的存储结构--链接表(Cont.)

- 操作的实现---- ⑨ position End (LIST L)
  - position End (LIST L)

```
{   position  q ;
    q = L ;
    while ( q→next != NULL )
        q = q→next ;
    return ( q ) ;
} //时间复杂性：O(n)
```

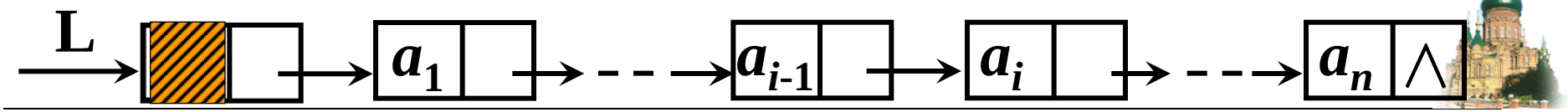


2.2线性表的存储结构--链接表(Cont.)

- 例：设计一个算法，遍历线性表，即按照线性表中元素的顺序，依次访问表中的每一个元素，每个元素只能被访问一次。

```
struct   celltype  {
        ElemType  data ;
        celltype  *next ;
} ; //结点型
typedef  celltype *LIST;
/*线性表的型*/
typedef  celltype  *position;
/*位置型*/
```

```
void   Travel( LIST  L )
{    position  p ;
     p = L→next ;
     while ( p != NULL) {
          cout << p→data ;
          p = p→next ;
      }
}
```



2.2线性表的存储结构--链接表(Cont.)

- 顺序表与 链表的比较

顺序表 单链表的比较

比较参数	顺序存储	链式存储
表的容量	固定，不易扩充	灵活，易扩充
存取操作	随机访问存取	顺序访问存取
时间	插入删除费时间	访问元素费时间
空间	估算表长度，浪费空间	实际长度，节省空间

•练习题：

1： 已知一个单链表，求倒数第k个元素。

，求中点元素。

算法1： 1) 找出长度n

2) n-k+1

算法2： 扫描一次

从左到右，设置两个变量

2：单链表逆置问题

方法很简单，采用生成单链表算法中的头插法思想就可以实现！

1，首先将第一个结点和其余结点断开；

2、然后将剩下的结点依次取下来，始终插入到第一个结点之后。就行了

- 思考题：
- 1：已知两个单链表，判断是否相交？（BAT公司面试题）

- 2：判断链表是否有环？

  若有，找出环入口结点？

• 作业：链表的维护与文件形式的保存

- 要求

- 用链表结构的有序表表示某商场家电的库存模型。当有提货或进货时需要对该链表进行维护。每个工作日结束之后，将该链表中的数据以文件形式保存，每日开始营业之前，需将以文件形式保存的数据恢复成链表结构的有序表。

- 链表结点的数据域包括家电名称、品牌、单价和数量，以单价的升序体现链表的有序性。程序功能包括：创建表、营业开始（读入文件恢复链表数据）、进货（插入）、提货（更新或删除）、查询信息、更新信息、营业结束（链表数据存入文件）等。