# 数据结构与算法

## 第2章 线性表

# 主要内容

- 线性结构
- 顺序表
- 链表
- 线性表实现方法的比较

# 线性结构

- 元素间满足线性关系
  - “一对一”的关系
  - 按此关系结构中的所有元素排成一个线性序列
- 二元组B = (K, R)，

  $K = \{a_0, a_1, \ldots, a_{n-1}\}$，$R = \{r\}$：
  - 结点集K中有一个唯一的开始结点，它没有前驱，但有一个唯一的后继；
  - 对于有限集K，它存在一个唯一的终止结点，该结点有一个唯一的前驱而没有后继；
  - 其它的结点皆称为内部结点，每一个内部结点都有且仅有一个唯一的前驱，也有一个唯一的后继；

    $a_0, a_1, \ldots, a_{n-1}$

    $\langle a_i, a_{i+1} \rangle$　$a_i$是$a_{i+1}$的前驱，　$a_{i+1}$是$a_i$的后继

# 线性结构

- 特点：
  - 均匀性：虽然不同线性表的数据元素可以是各种各样的，但对于同一线性表的各数据元素必定具有相同的数据类型和长度
  - 有序性：各数据元素在线性表中都有自己的位置，且数据元素之间的相对位置是线性的

# 2.1 线性表 (linear list)

- 三个方面
  - 线性表的逻辑结构
  - 线性表的存储结构
  - 线性表运算分类

# 线性表的逻辑结构

- 线性表：
  - 由称为元素（element）的数据项组成的一种有限且有序的序列，这些元素也可称为结点或表目
- 二元组(K , r)：
  - 由结点集K，以及定义在结点集K上的线性关系 *r* 所组成的线性结构
  - 线性表所包含的结点个数称为线性表的长度，它是线性表的一个重要参数；长度为0的线性表称为空表；
  - 线性表的关系 r，简称前驱/后继关系，具有反对称性和传递性

# 线性表逻辑结构

- 主要属性包括：
  - 线性表的长度
  - 表头(head)
  - 表尾 (tail)
  - 当前位置(current position)

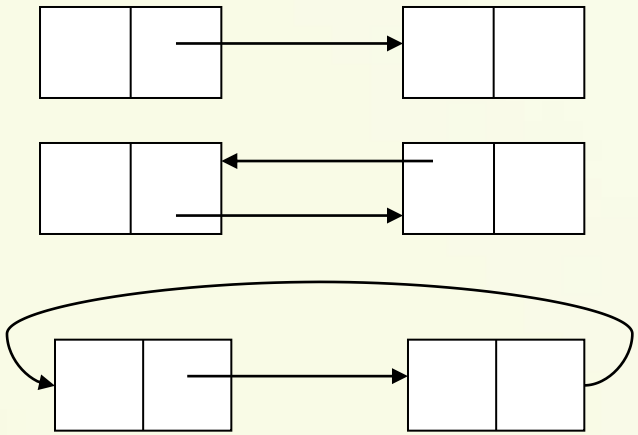
# 线性表的存储结构

- 定长的一维数组结构
  - 又称为向量型的顺序存储结构
- 变长的线性表存储结构
  - 链接式存储结构
  - 串结构、动态数组、以及顺序文件

# 线性表的存储结构

- 顺序表
  - 按索引值从小到大存放在一片相邻的连续区域
  - 紧凑结构，存储密度为1


- 链表
  - 单链表
  - 双链表
  - 循环链表



# 线性表运算分类

- 创建线性表的一个实例list(-)
- 清除线性表（即析构函数）~list()
- 获取有关当前线性表的信息
- 访问线性表并改变线性表的内容或结构
- 线性表的辅助性管理操作

# 线性表类模板

```
template <class T>
class List  {
    void clear();                          // 置空线性表
    bool isEmpty();                        // 线性表为空时，返回true
    bool append(const T value);            // 在表尾添加一个元素value，表的长度增1
    bool insert(const int p, const T value);
                                           // 在位置p上插入一个元素value，表的长度增1
    bool delete(const int p);              // 删除位置p上的元素，表的长度减 1
    bool getPos(int & p, const T value)        // 查找值为value的元素并返回其位置
    bool getValue(const int p, T& value);
                                           // 把位置p的元素值返回到变量value中
    bool setValue(const int p, const T value);// 用value修改位置p的元素值
    bool getPos (int &p, const T value);
                                           // 把值为value的元素所在位置返回到变量p中
};
```

# 2.2 顺序表（*array-based list*）

- 采用定长的一维数组存储结构
- 也称向量
  - 主要特性：
    - 元素的类型相同
    - 元素顺序地存储在连续存储空间中，每一个元素有唯一的索引值
    - 使用常数作为向量长度

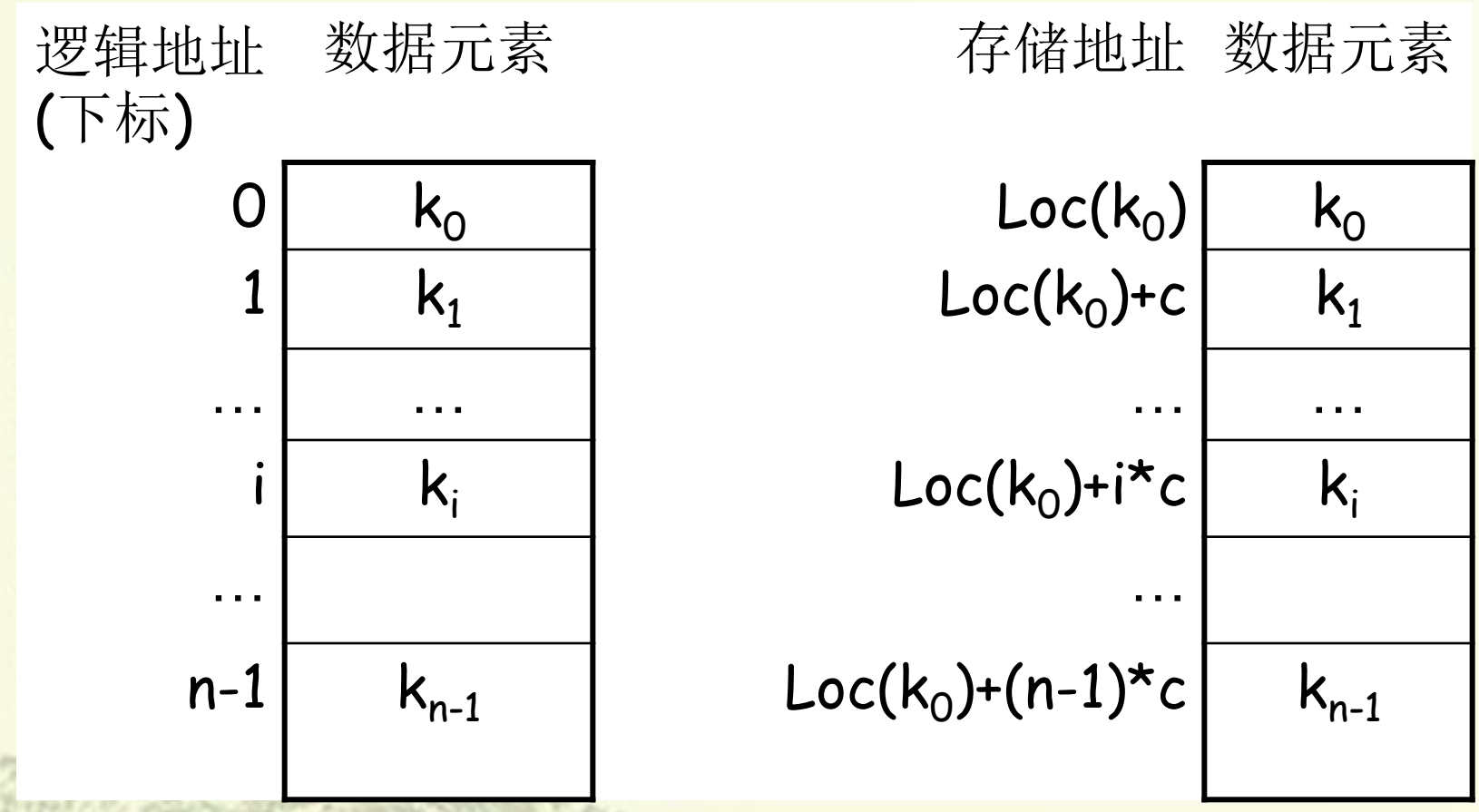
# 顺序表

- 数组存储
- 读写其元素很方便，通过下标即可指定位置
  - 只要确定了首地址，线性表中任意数据元素都可以随机存取。地址计算如下所示：

$$loc(k_i) = loc(k_0) + c * i$$

$$c = sizeof(ELEM)$$

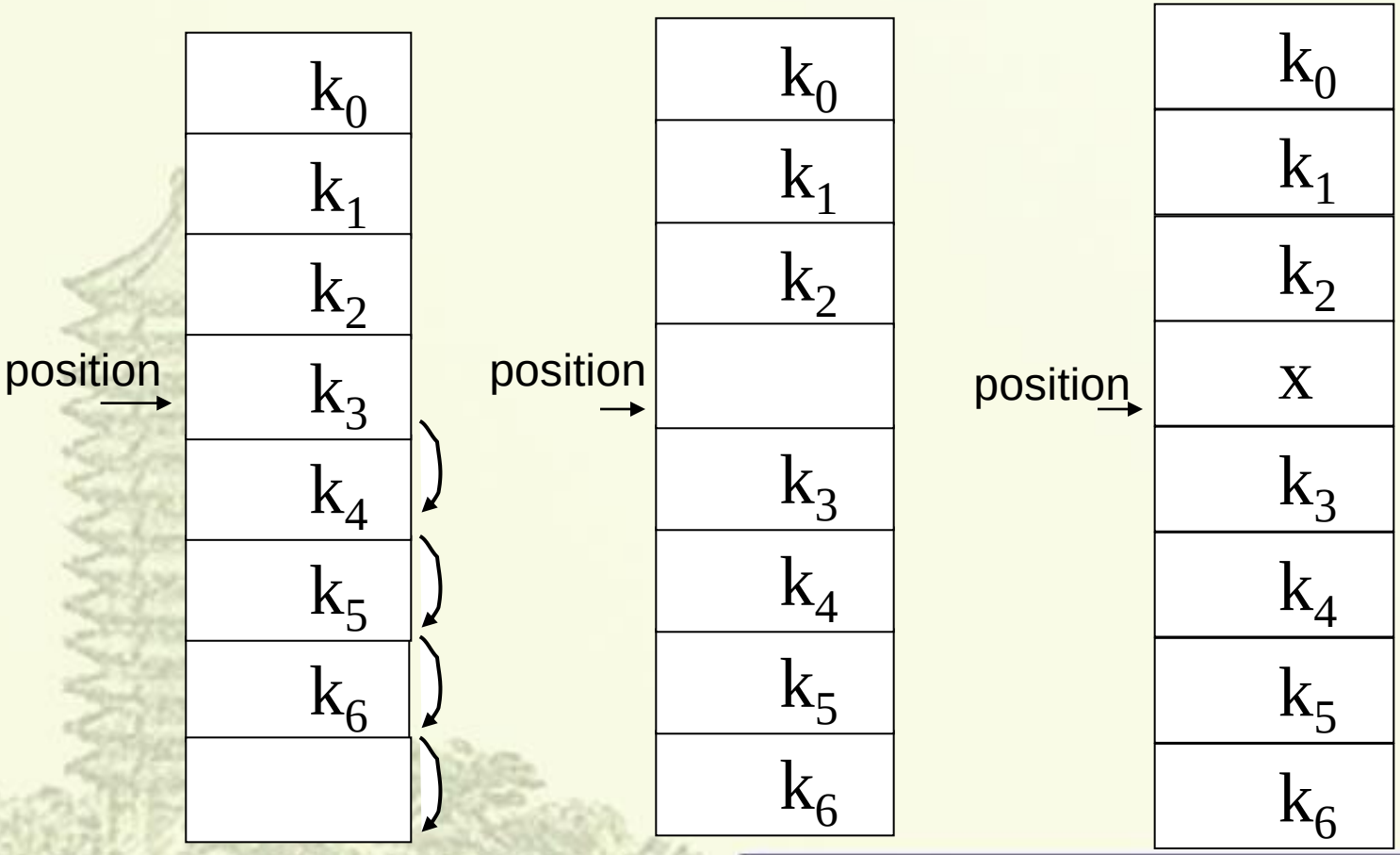
# 顺序表


逻辑地址
(下标)
数据元素
0
$k_0$
1
$k_1$
…
…
i
$k_i$
…
n-1
$k_{n-1}$
存储地址
数据元素
$Loc(k_0)$
$k_0$
$Loc(k_0)+c$
$k_1$
…
…
$Loc(k_0)+i*c$
$k_i$
…
$Loc(k_0)+(n-1)*c$
$k_{n-1}$

# 顺序表类定义

```
class arrList : public List<T> {          // 顺序表，向量
private:                                                  // 线性表的取值类型和取值空间
      T *aList ;                                          // 私有变量，存储顺序表的实例
      int  maxSize;                                       // 私有变量，顺序表实例的最大长度
      int  curLen;                                        // 私有变量，顺序表实例的当前长度
      int  position;                                      // 私有变量，当前处理位置
public:                                   // 顺序表的运算集
      arrList(const int size) {                           // 创建一个新的顺序表，参数为表实例的最大长度
            maxSize = size;  aList = new T[maxSize]; curLen = position = 0;
      }
      ~arrList() {                                        // 析构函数，用于消除该表实例
            delete [] aList;
      }
      void clear() {                                      // 将顺序表存储的内容清除，成为空表
            delete [] aList; curLen = position = 0;
            aList = new T[maxSize];
      }
      int length();                                       // 返回此顺序表的当前实际长度
      bool append(const T value);                         // 在表尾添加一个元素value，表的长度增1
      bool insert(const int p, const T value);            // 在位置p上插入一个元素value，表的长度增1
      bool delete(const int p);                           // 删除位置p上的元素，表的长度减 1
      bool setValue(const int p, const T value);          // 用value修改位置p的元素值
      bool getValue(const int p, T& value);               // 把位置p的元素值返回到变量value中
      bool getPos(int & p. const T value);                // 查找值为value的元素，并返回第1次出现的位置
};
```

# 顺序表的插入图示


position
$k_0$
$k_1$
$k_2$
$k_3$
$k_4$
$k_5$
$k_6$
position
$k_0$
$k_1$
$k_2$
$k_3$
$k_4$
$k_5$
$k_6$
position
$k_0$
$k_1$
$k_2$
x
$k_3$
$k_4$
$k_5$
$k_6$

# 插入算法

```
// 设元素的类型为T， aList是存储顺序表的数组， maxSize是其最大长度；
// p为新元素value的插入位置，插入成功则返回true， 否则返回false
template <class T> bool arrList<T> :: insert (const int p, const T value) {
    int i;
    if (curLen >= maxSize) {                    // 检查顺序表是否溢出
        cout << "The list is overflow"<<endl; return false;
    }
    if (p < 0 || p > curLen) {                  // 检查插入位置是否合法
        cout << "Insertion point is illegal"<<endl; return false;
    }
    for (i = curLen; i > p; i--)
        aList[i] = aList[i-1];                  // 从表尾curLen -1起往右移动直到p
    aList[p] = value;                           // 位置p处插入新元素
    curLen++;                                   // 表的实际长度增1
    return true;
}
```

# 插入算法的执行时间

- 插入操作的主要代价体现在表中元素的**移动**
  - 在位置i插入元素，需移动n-i 个元素

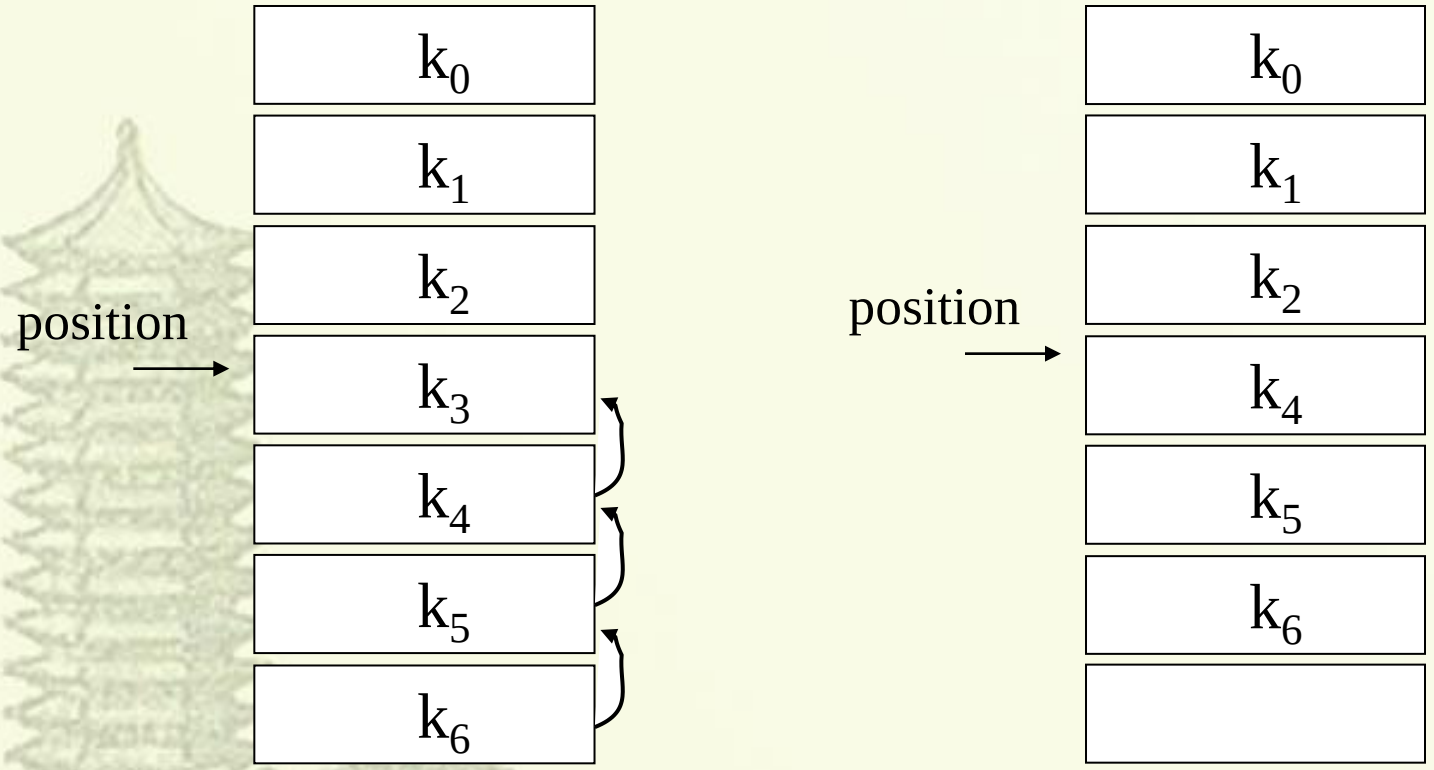
•元素总个数为k，假设各个位置插入的概率相等，为p＝1/k

•平均移动元素次数为

$$\sum_{i=0}^{k-1} 1/k \bullet (k-i) \approx \frac{k}{2}$$

总时间开销估计为O(k)

# 顺序表的删除图示



# 删除算法

```
// 设元素的类型为T；aList是存储顺序表的数组； p为即将删除元素的位置
// 删除成功则返回true，否则返回false

template <class T>                                  // 顺序表的元素类型为T
bool arrList<T> :: delete(const int p) {
    int i;
    if (curLen <= 0 ) {                             // 检查顺序表是否为空
        cout << " No element to delete \n"<<endl;
    return false ;
    }
    if (p < 0 || p > curLen-1) {                    // 检查删除位置是否合法
        cout << "deletion is illegal\n"<<endl;
        return false ;
    }
    for (i = p; i < curLen-1; i++)
        aList[i] = aList[i+1];              // 从位置p开始每个元素左移直到curLen
    curLen--;                                       // 表的实际长度减1
    return true;
}
```

# 删除算法时间代价

- 与插入操作相似，主要的代价在于元素的移动
- 等概率情况下平均时间代价为O(k)

# 顺序表各运算的算法分析

- 插入和删除操作的主要代价体现在表中元素的**移动**
  - 插入：**移动** n-i
  - 删除：**移动** n-i-1个
- 若在下标为 i 的位置上插入和删除元素的概率分别是$p_i$和$p_i'$，
  - 则插入时的平均**移动次数**是：

$$M_i = \sum_{i=0}^{n} (n-i)p_i;$$

  - 删除的平均**移动次数**是：

$$M_d = \sum_{i=0}^{n-1} (n-i-1)p_i'$$

# 算法分析

- 如果在顺序表中每个位置上插入和删除元素的概率相同，即：　　$p_i=1/(n+1)$, $p_i'=1/n$

$$M_i = \frac{1}{n+1}\sum_{i=0}^{n}(n-i) = \frac{1}{n+1}(\sum_{i=0}^{n}n - \sum_{i=0}^{n}i)$$

$$= \frac{n(n+1)}{n+1} - \frac{n(n+1)}{2(n+1)} = \frac{n}{2}$$

$$M_d = \frac{1}{n}\sum_{i=0}^{n}(n-i-1) = \frac{1}{n}(\sum_{i=0}^{n}n - \sum_{i=0}^{n}i - n)$$

$$= \frac{n^2}{n} - \frac{(n-1)}{2} - 1 = \frac{n-1}{2}$$

采用大O表示法，则时间代价为**O(n)**

# 顺序表的优缺点

- 优点
  - 不需要附加空间
  - 随机存取任一个元素（根据下标）
- 缺点
  - 很难估计所需空间的大小
  - 开始就要分配足够大的一片连续的内存空间
  - 更新操作代价大

# 2.3 链表(Linked List)

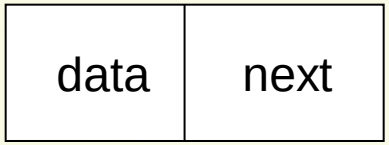
- 通过指针来链接结点的存储方式。
  - 利用指针来表示数据元素之间的逻辑关系
  - 逻辑上相邻的元素在物理位置上不要求也相邻
  - 按照需要为表中新的元素动态地分配存储空间，动态改变长度
- 根据链接方式和指针多寡
  - 单链表
  - 双链表
  - 循环链表

# 链表的运算

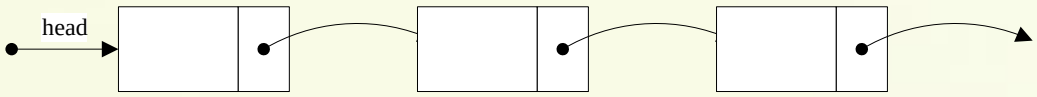
- 检索：
  - 在链表中查找满足某种条件的元素
- 插入：
  - 在链表的适当位置插入一个元素
- 删除：
  - 从链表中删除一个指定元素

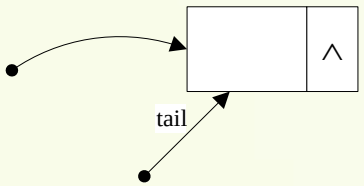
# 单链表（*singly linked list*）

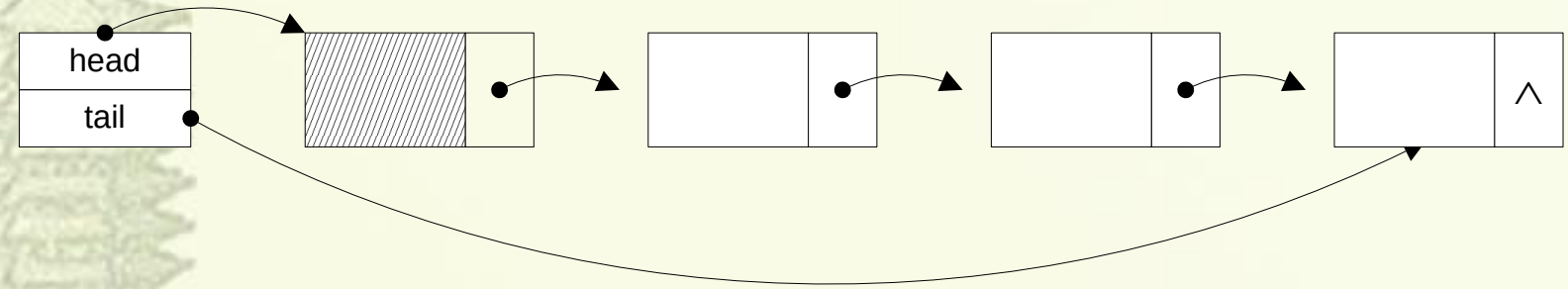
- 通过指针把它的一串存储结点链接成一个链
- 存储结点由两部分组成：
  - 数据字段 + 指针字段（后继地址）

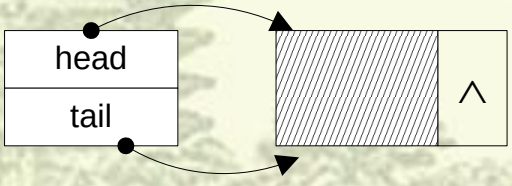


# 单链表的存储结构



# 单链表的结点类型

```
template <class T> class Link {
public:
    T    data;              // 用于保存结点元素的内容
         Link<T> * next; // 指向后继结点的指针

         Link(const T info, const Link<T>* nextValue =  NULL) {
                                   data = info;
                 next = nextValue;
         }
         Link(const Link<T>* nextValue) {
                 next = nextValue;
         }
};
```

# 单链表的定义

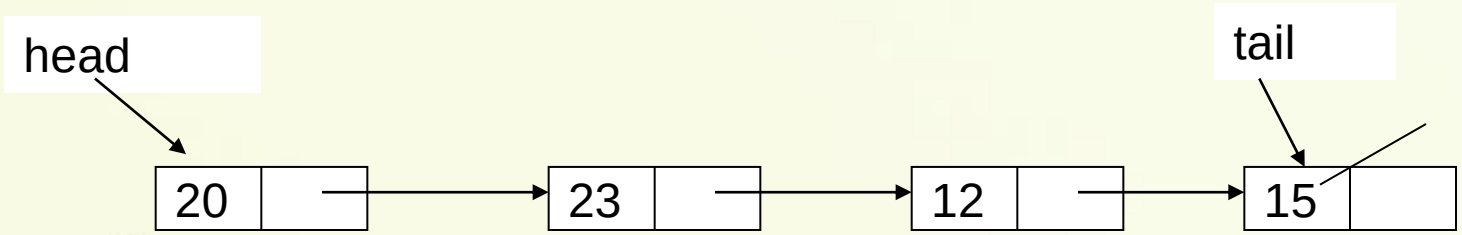
```
template <class T> class lnkList : public List<T> {
    private:
    Link<T> * head, *tail;                  // 单链表的头、尾指针
    Link<T> *setPos(const int p);           // 返回线性表指向第p个元素的指针值
public:
    lnkList(int s);                         // 构造函数
    ~lnkList();                             // 析构函数
    bool isEmpty();                         // 判断链表是否为空
    void clear();                           // 将链表存储的内容清除，成为空表
    int length();                           // 返回此顺序表的当前实际长度
    bool append(cosnt T value);             // 在表尾添加一个元素value，表的长度增1
    bool insert(cosnt int p, cosnt T value);// 在位置p上插入一个元素value，表的长度增1
    bool delete(cosnt int p);               // 删除位置p上的元素，表的长度减 1
    bool getValue(cosnt int p, T& value);// 返回位置p的元素值
    bool getPos(int &p, const T value); // 查找值为value的元素，返回第1次出现的位置
}
```

# 查找单链表中第i个结点

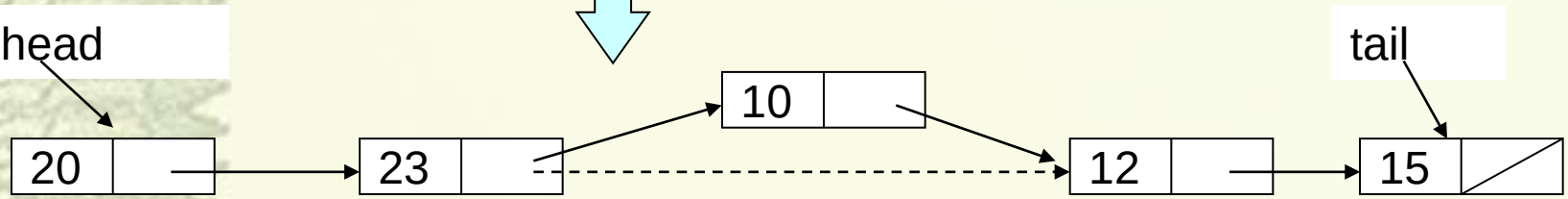
```
// 函数返回值是找到的结点指针
template <class T>                          // 线性表的元素类型为T
Link<T> * lnkList <T>:: setPos(int i) {

    int count = 0;
    if (i == -1)                            //  i为-1则定位到头结点
        return head;
    // 循链定位，若i为0则定位到第一个结点
    Link<T> *p = new Link<T>(head->next);
    while (p != NULL && count < i) {
        p = p-> next;
        count++;
    };
    return p; // 指向第 i 结点，i=0,1,...，当链表中结点数小于i时返回NULL
}
```

# 单链表的插入




在23 和12 之间插入 10



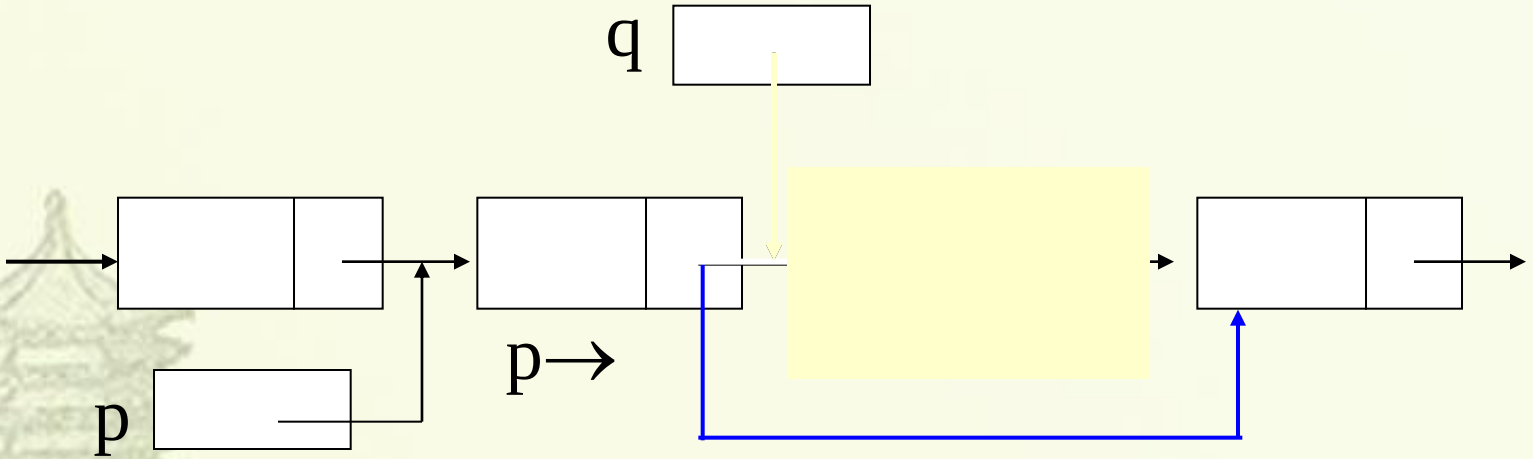

1. 创建新结点
2. 新结点指向右边的结点
3. 左边结点指向新结点

# 单链表的删除

- 从链表中删除结点x
  1. 用p指向元素x的结点的前驱结点
  2. 删除元素为x的结点
- 用p指向元素x的结点的前驱结点

```
p=head;
while (p->next!=NULL && p-> next ->info!=x)
        p=p-> next;
```

# 删除值为 x 的结点




```
q = p→next;
```

```
p→ next = q→ next;
```

```
free(q);
```

# 单链表删除算法

```
template <class T>                                  // 线性表的元素类型为T
bool lnkList<T>:: delete((const int i) {
        Link<T> *p, *q;

        if  ((p = setPos(i-1)) == NULL  ||  p == tail) {  // 待删结点不存在，即给定的i大于当前链中元素个数
            cout << " 非法删除点 " <<endl;
            return false;
        }
        q = p->next;                                // q是真正待删结点
        if (q == tail) {                            // 待删结点为尾结点，则修改尾指针
            tail = p;
            p->next = NULL:
            delete q;
        }
        else if (q != NULL)  {                      // 删除结点q 并修改链指针
            p->next = q->next;
            delete q;
        }
        return true;
}
```

# 求长度算法

```
int length()  {
    Link<T>  *p = head->next;
    int count = 0;
    while (p != NULL) {
        p = p->next;
        count++;
    }
    return count;
}
```

# 单链表上运算的分析

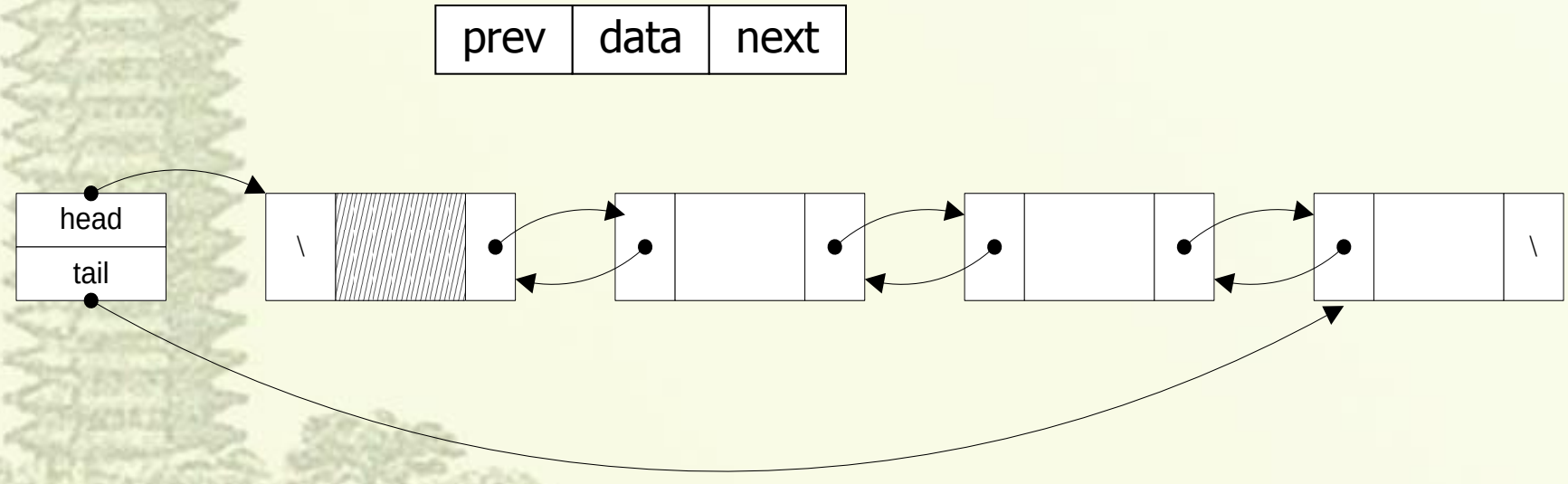
1. 对一个结点操作，必先找到它，即用一个指针指向它
2. 找单链表中任一结点，都必须从第一个点开始：

```
p = head;
while (没有到达)
    p = p->next;
```

- 单链表的时间复杂度　O(n)
  - 定位：O(n)
  - 插入：O(n) + O(1)
  - 删除：O(n) + O(1)

# 双链表(double linked list)

- 为弥补单链表的不足, 而产生双链表
  - 单链表的next字段仅仅指向后继结点，不能有效地找到前驱, 反之亦然
  - 增加一个指向前驱的指针



# 双链表及其结点类型的说明

```
template <class T> class Link {
public:
    T      data;                    // 用于保存结点元素的内容
    Link<T> * next;                 // 指向后继结点的指针
    Link<T> *prev;                  // 指向前驱结点的指针
    Link(const T info, Link<T>* preValue = NULL, Link<T>* nextValue = NULL) {
                                    // 给定值和前后指针的构造函数
        data = info;
        next = nextValue;
        prev = preValue;
    }
    Link(Link<T>* preValue = NULL, Link<T>* nextValue = NULL)  {
                                    // 给定前后指针的构造函数
        next = nextValue;
        prev = preValue;
    }
}
```

# 双链表的插入

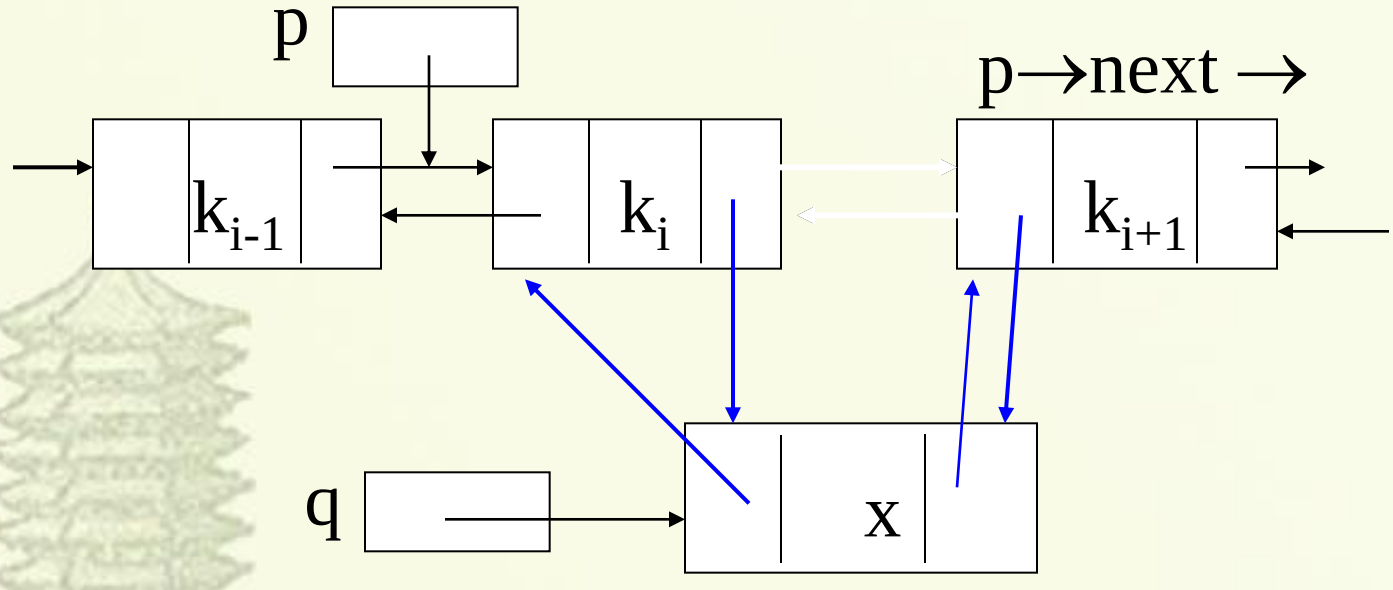
- 如果要在p所指结点后插入一个新结点
  - 执行new q开辟结点空间；
  - 让该新结点的next填入p所指的后继地址；
  - 新结点的prev填入p所指结点的后继的prev字段；

```
new q;
q->next = p-> next;
q->prev = p-> next ->prev;
```

  - 把新结点的地址填入原p所指结点的next字段；
  - 新结点后继结点的prev字段也应该回指新结点

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

# 双链表插入示意




q → prev = p;

q → next = p → next;

p →next → prev = q;

p → next = q;

# 双链表的删除

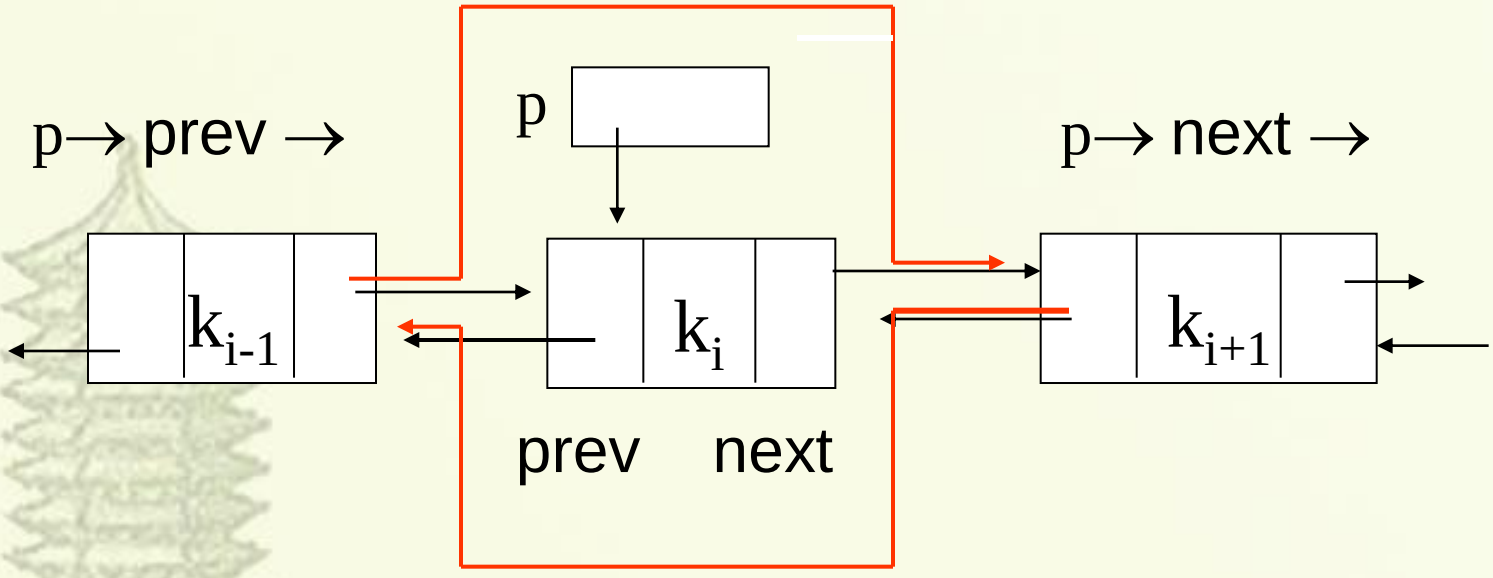
- 如果要删除指针变量 p 所指的结点，只需修改该结点前驱的 next 字段和该结点后继的prev字段，即

```
p-> prev -> next = p-> next;
p-> next -> prev = p-> prev;
```

然后把变量p变空，再把p所指空间释放即可

```
p-> next = NULL;
p->prev = NULL;
delete p;
```

# 双链表删除示意

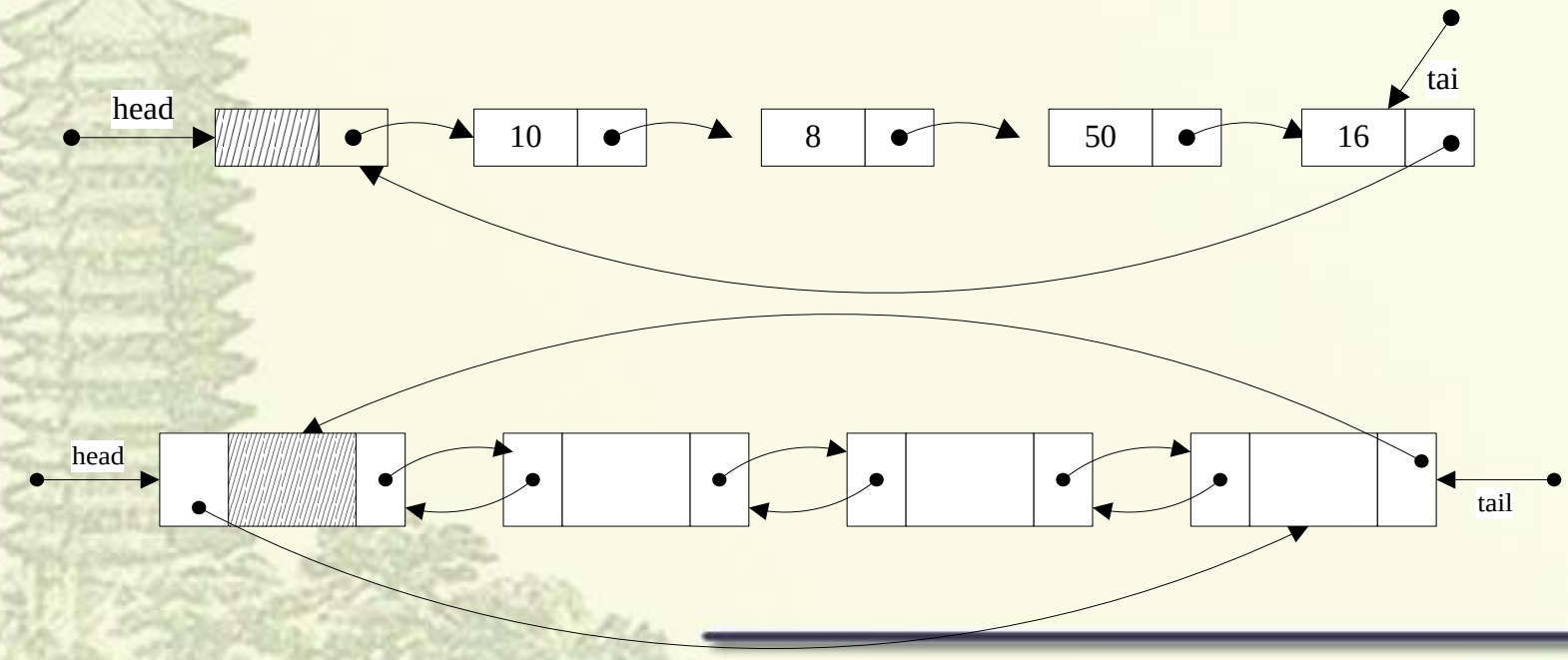



$p \rightarrow prev \rightarrow next = p \rightarrow next$

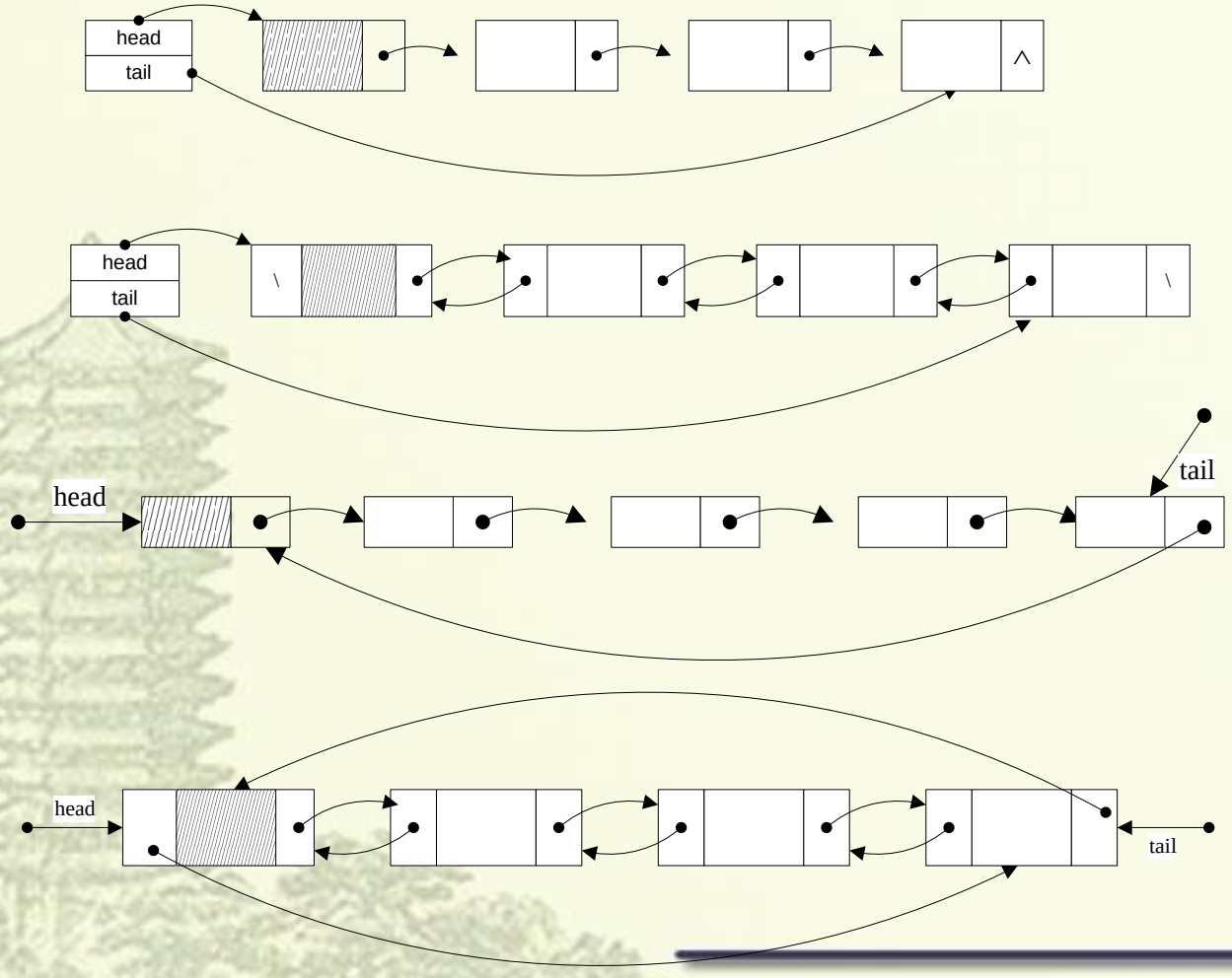
$p \rightarrow next \rightarrow prev = p \rightarrow prev$

# 循环链表（*circularly linked list*）

- 将单链表或者双链表的头尾结点链接起来，就是一个循环链表
- 不增加额外存储花销，却给不少操作带来了方便
  - 从循环表中任一结点出发，都能访问到表中其他结点



# 几种主要链表比较



# 链表的边界条件

- 几个特殊点的处理
  - 头指针处理
  - 非循环链表尾结点的指针域保持为NULL
  - 循环链表尾结点的指针回指头结点
- 链表处理
  - 空链表的特殊处理
  - 插入或删除结点时指针勾链的顺序
  - 指针移动的正确性
    - 插入
    - 查找或遍历

# 2.4 线性表实现方法的比较

- 顺序表的主要优点
  - 没有使用指针，不用花费额外开销
  - 线性表元素的读访问非常简洁便利
- 链表的主要优点
  - 无需事先了解线性表的长度
  - 允许线性表的长度动态变化
  - 能够适应经常插入删除内部元素的情况

- 顺序表是存储静态数据的不二选择
- 链表是存储动态变化数据的良方

# 顺序表和链表的比较

- 顺序表
  - 插入、删除运算时间代价**O(n)**，查找则可常数时间完成
  - 预先申请固定长度的数组
  - 如果整个数组元素很满，则没有结构性存储开销
- 链表
  - 插入、删除运算时间代价**O(1)**，但找第**i**个元素运算时间代价**O(n)**
  - 存储利用指针，动态地按照需要为表中新的元素分配存储空间
  - 每个元素都有结构性存储开销

# 应用场合的选择

- 顺序表不适用的场合
  - 经常插入删除时，不宜使用顺序表
  - 线性表的最大长度也是一个重要因素
- 链表不适用的场合
  - 当读操作比插入删除操作频率大时，不应选择链表
  - 当指针的存储开销，和整个结点内容所占空间相比其比例较大时，应该慎重选择

# 顺序表和链表的选择

- 顺序表
  - 结点总数目大概可以估计
  - 线性表中结点比较稳定（插入删除少）
- 链表
  - 结点数目无法预知
  - 线性表中结点动态变化（插入删除多）

# 小结

- 线性结构的基本概念
- 线性表的顺序实现
- 线性表的链式实现

# The End