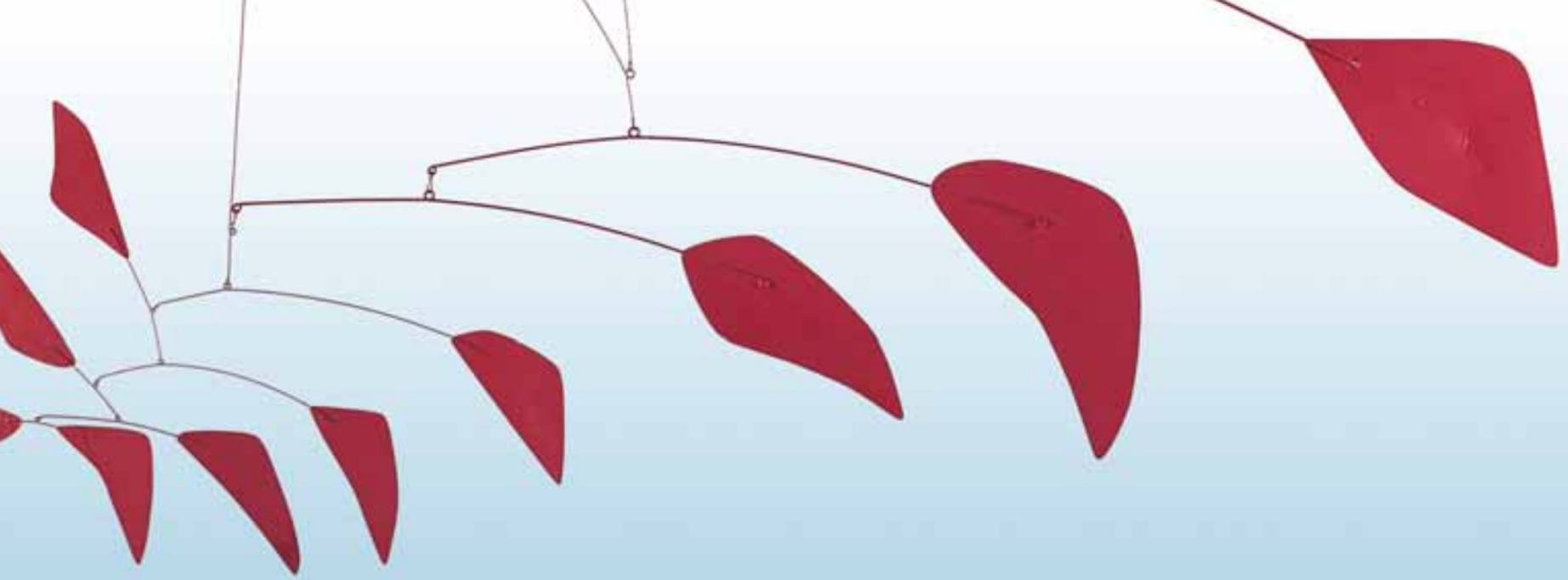


# 第三部分：数据结构

INTRODUCTION TO

# ALGORITHMS

THIRD EDITION

# 数据结构

- 数据结构：在计算机上表示和操作有限动态集合的技术。
- 有限动态集合：数学中的集合是不变的，而由算法操作的集合却在整个过程中能增大、缩小或发生其他变化。
- 动态集合的元素：对象
  - 一些动态集合假定对象中的一个属性为标识关键字（key）
  - 一些动态集合以其关键字来自于某个全序集为前提条件
  - 对象可能包含卫星数据，它们与对象的其他属性一起移动，除此之外，动态集合的实现不使用它们。
  - 对象也可以有由集合操作使用的属性，这些属性可能包含有关集合中其他对象的数据或指针。

# 动态集合上的操作

- 查询操作
  - $SEARCH(S,k)$：返回 $x \in S$ 使得 $x.key = k$，若 $x \notin S$ 则返回 $NIL$
  - $MINIMUM(S)$：返回全序集 $S$ 中具有最小关键字的元素
  - $MAXIMUM(S)$：返回全序集 $S$ 中具有最大关键字的元素
  - $SUCCESSOR(S,x)$：返回全序集 $S$ 中关键字比 $x$ 大的下一个元素，$x$ 就是最大则返回 $NIL$
  - $PREDECESSOR(S,x)$：返回全序集 $S$ 中关键字比 $x$ 小的前一个元素，$x$ 就是最小则返回 $NIL$
- 修改操作
  - $INSERT(S,x)$：将元素 $x$ 加入集合 $S$ （$S$ 所需的 $x$ 的属性已初始化）
  - $DELETE(S,x)$：从集合 $S$ 中删除元素 $x$

# 本部分概览

- 第 9 讲：基本数据结构
  - 栈、队列、链表、有根树
- 第 10 讲：散列表
  - 支持期望上常数项时间的字典操作（$INSERT, DELETE, SEARCH$）
- 第 11 讲：二叉搜索树
  - 支持所有动态集合操作，随机情况下对数时间
- 第 12 讲：红黑树
  - 支持所有动态集合操作，最坏情况下对数时间
- 第 13 讲：数据结构的扩张
  - 扩展经典数据结构以适应特定需求

另一种重要的数据结构“堆”已经在第 5 讲介绍过了。

# 基本数据结构

《算法导论》—— 第9讲

jiacaicui@163.com

# 内容提要

- 栈和队列
  - 比较、操作、表示、实现
- 链表
  - 表示、搜索、插入、删除、哨兵
- 指针和对象的实现
  - 多数组表示、单数组表示、分配与释放
- 有根树的表示
  - 二叉树
  - 分支无限制的有根树：左子女右兄弟表示法
  - 树的其它表示方法

# 栈和队列

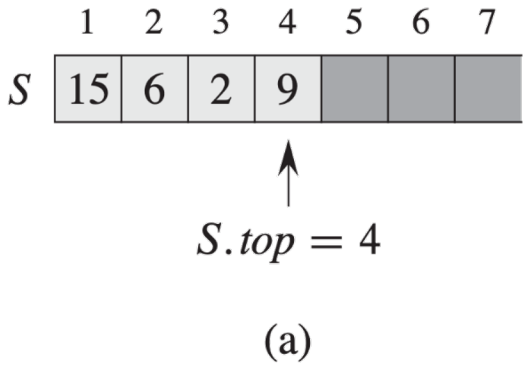
生活中的数据结构

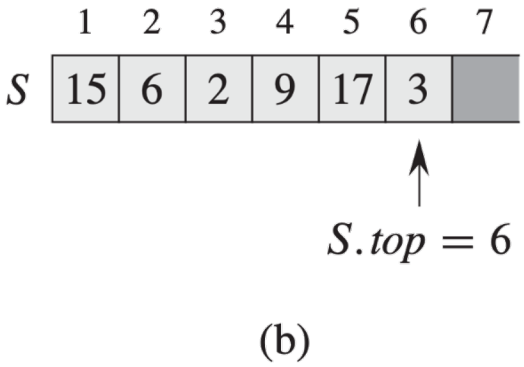
# 栈和队列

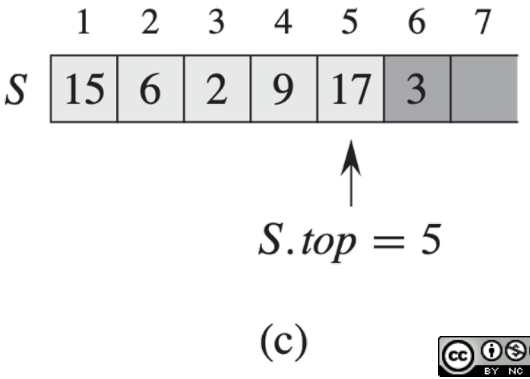
- 共同特点：
  - 都是动态集合，在其上进行 $DELETE$ 操作所移除的元素是预先设定的。
- 不同的删除策略：
  - 栈（Stack）：后进先出（Last-in, First-out, LIFO），被删去的总是在集合中存在时间最短的那个元素。
  - 队列（Queue）：先进先出（First-in, First-out, FIFO），被删去的总是在集合中存在时间最长的那个元素。
- 栈和队列都是抽象数据类型
  - 回忆：“优先队列”（第 5 讲）
  - 本讲将介绍“栈”和“队列”基于简单数组的实现

# 栈

- 栈的操作
  - 栈上的 $INSERT$ 操作称为压入（$PUSH$）
  - 栈上无元素参数的 $DELETE$ 操作称为弹出（$POP$）
- 栈的表示
  - 可以用数组 $S[1..n]$ 来实现最多容纳 $n$ 个元素的栈
  - 属性 $S.top$ 指向最新插入的元素
  - 栈中元素为 $S[1..S.top]$，其中 $S[1]$ 是栈底，$S[S.top]$ 是栈顶



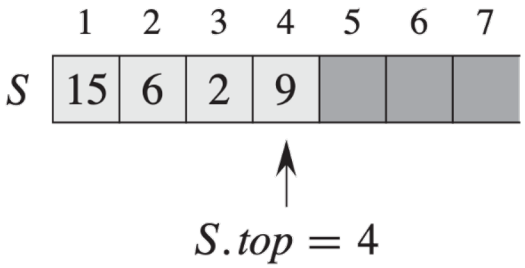
# 栈

- 栈的实现
  - 如果试图对一个空栈执行弹出操作，则称栈下溢（Underflow）
  - 如果 $S.top$ 超过了 $n$，则称栈上溢（Overflow）（暂不考虑上溢）
  - 三种栈操作的执行时间都为 $O(1)$

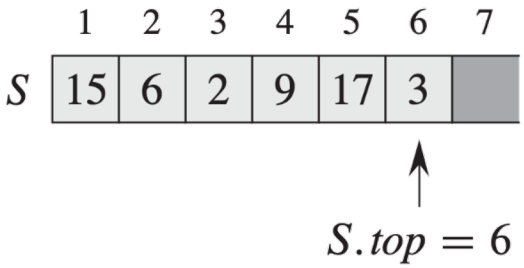
```
STACK-EMPTY(S)
1  if S.top == 0
2      return TRUE
3  else return FALSE
```

```
PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x
```

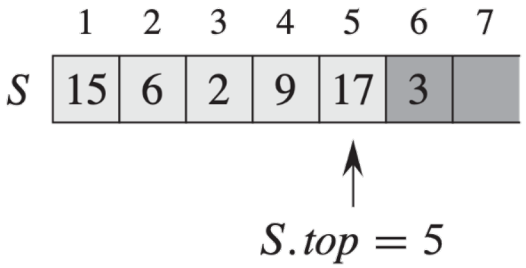
```
POP(S)
1  if STACK-EMPTY(S)
2      error "underflow"
3  else S.top = S.top − 1
4      return S[S.top + 1]
```




(a)




(b)



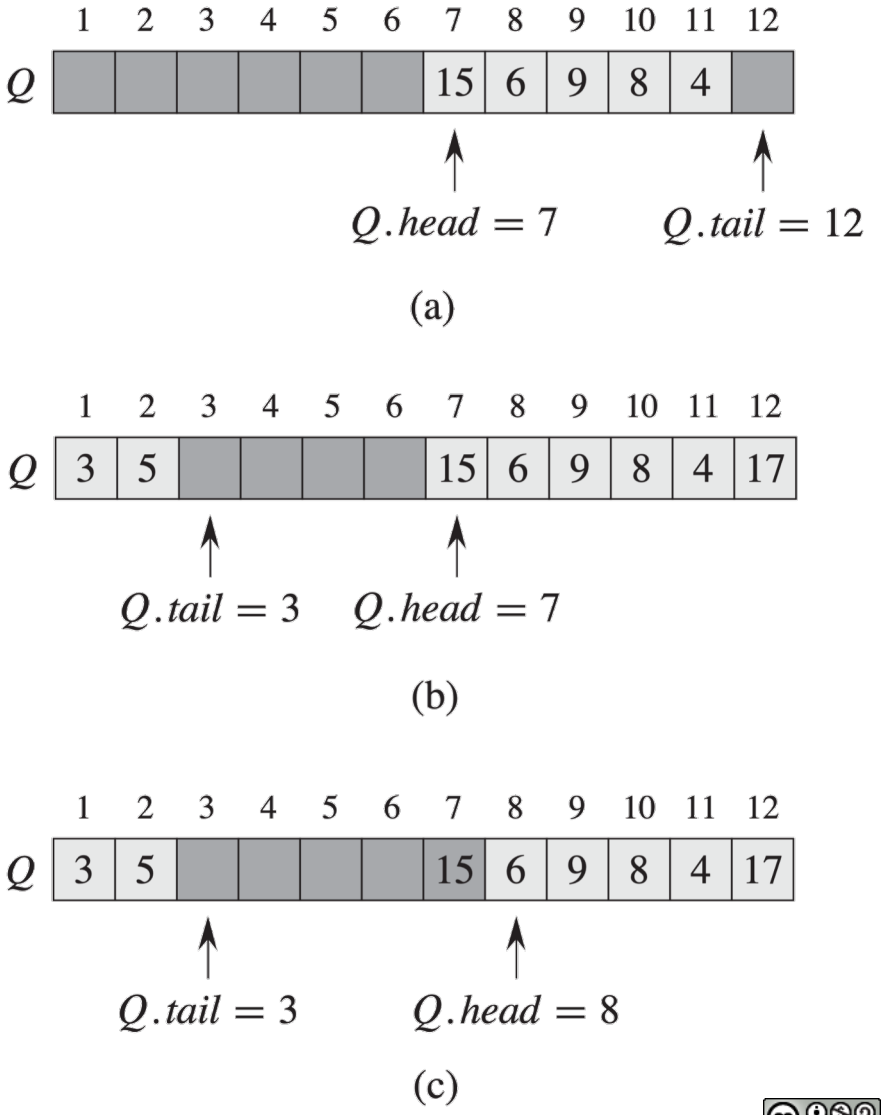

(c)

# 队列

- 队列的操作
  - 队列上的 $INSERT$ 操作称为入队（$ENQUEUE$）
  - 队列上无元素参数的 $DELETE$ 操作称为出队（$DEQUEUE$）

# 队列

- 队列的表示
  - 可以用数组 $Q[1..n]$ 实现最多容纳 $n-1$ 个元素的队列
  - 属性 $Q.head$ 指向队头（Head）元素
  - 属性 $Q.tail$ 指向下一个新元素将要插入的位置，即队尾（Tail）
  - 队列中的元素存放在位置 $Q.head, Q.head+1, \cdots, Q.tail-1$，并在最后的位置“环绕”，逻辑上位置 1 紧邻在位置 $n$ 后面形成一个环序



# 队列

- 队列的实现

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3        Q.tail = 1
4  else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

- 当 $Q.head = Q.tail$ 时，队列为空，初始时有 $Q.head = Q.tail = 1$
- 当 $Q.head = Q.tail + 1$ 时（在环序意义下），队列是满的
- 试图对空队列进行出队操作称队列发生下溢（Underflow）
- 试图对满队列进行入队操作称队列发生上溢（Overflow）
- 两种操作的执行时间都为 $O(1)$ （溢出检查见*练习9-1问题1*）

# 链表

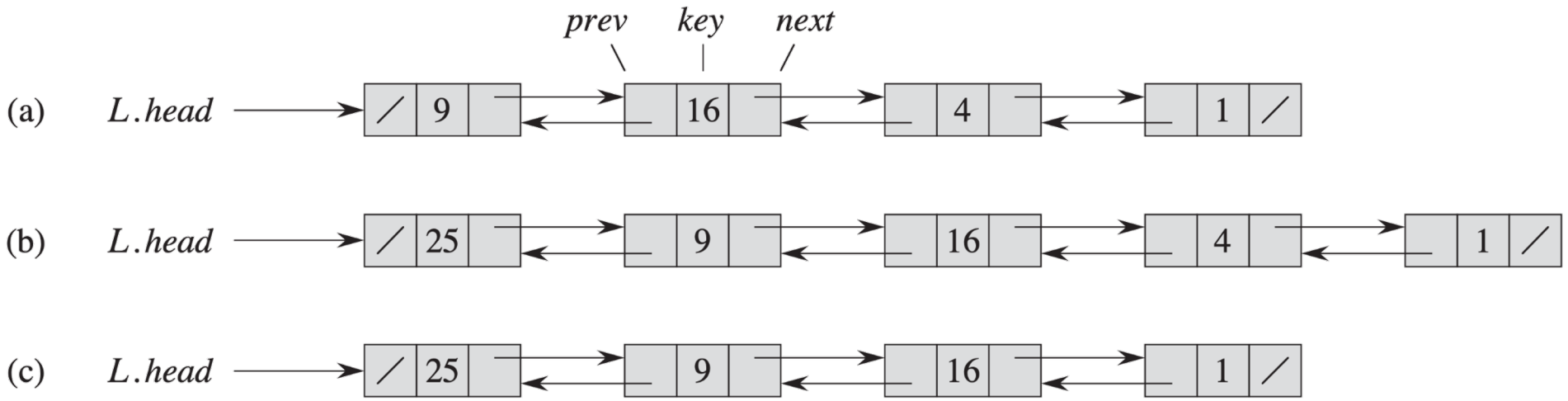
万能的数据结构

# 链表

- 链表（Linked List）是一种线性数据结构
  - 其线性顺序是由各个对象里的指针决定的
  - 数组的线性顺序是由数组下标决定的
  - 链表可以支持所有动态集合操作（虽然未必都很高效）
- 链表的多种形式
  - 双向链表：每个元素有 $prev$ 和 $next$ 两个指针指向前驱和后继元素
  - 单向链表（Singly Linked List）：省略双向链表里元素的 $prev$ 指针
  - 循环链表（Circular List）：头元素的 $prev$ 指针指向尾元素，尾元素的 $next$ 指针指向头元素。
- 本讲我们只讨论未排序的双向链表

# 链表的表示

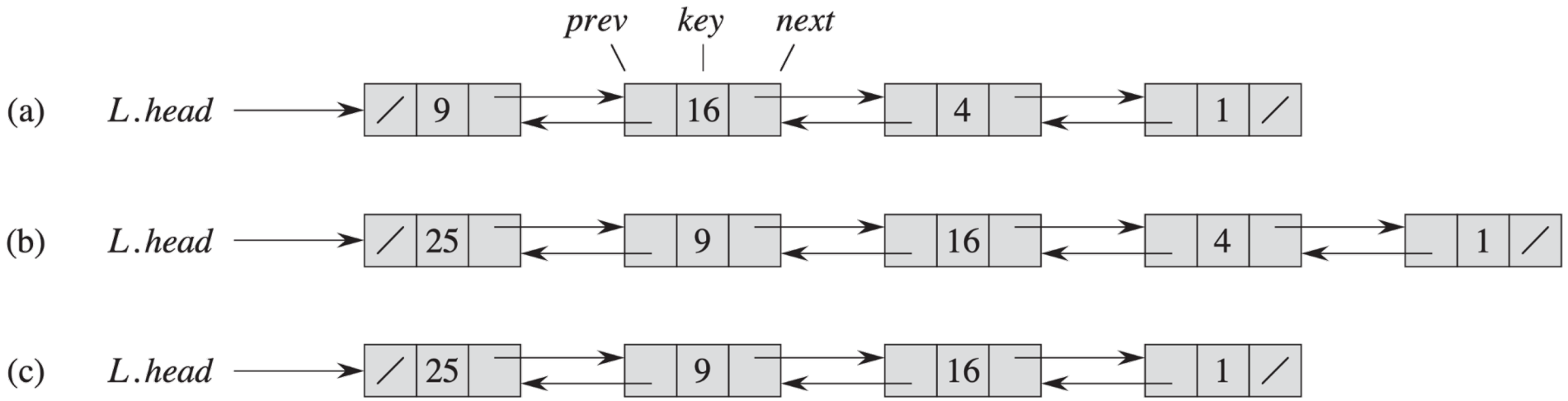
- 双向链表（Doubly Linked List） $L$ 的每个元素都是一个对象 $x$
  - $x.key$ 是元素的关键字，$x$ 可能还有其他的卫星数据
  - $x.prev$ 指向它在链表中的前驱元素，$x.next$ 指向它在链表中的后继元素
  - 若 $x.prev = NIL$，则 $x$ 是链表的第一个元素，即链表的头（Head）
  - 若 $x.next = NIL$，则 $x$ 是链表的最后一个元素，即链表的尾（Tail）
  - $L.head$ 指向链表的第一个元素，若 $L.head = NIL$ 则链表为空



# 链表的搜索

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

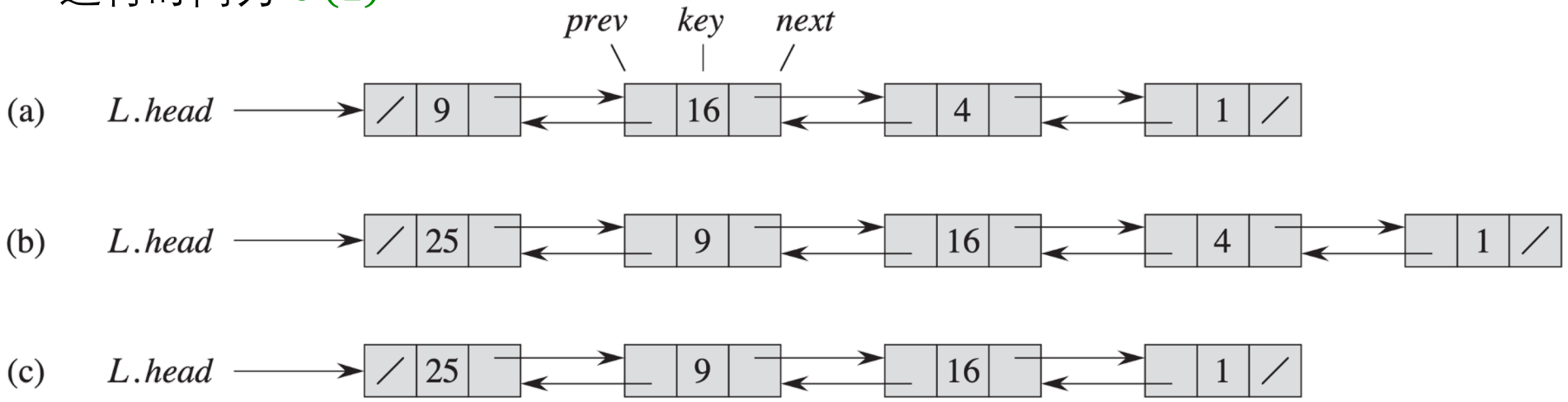
- 查找链表 $L$ 中第一个关键字为 $k$ 的元素，没有则返回 $NIL$
- 采用简单的线性搜索方法，最坏情况运行时间为 $\Theta(n)$



# 链表的插入

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

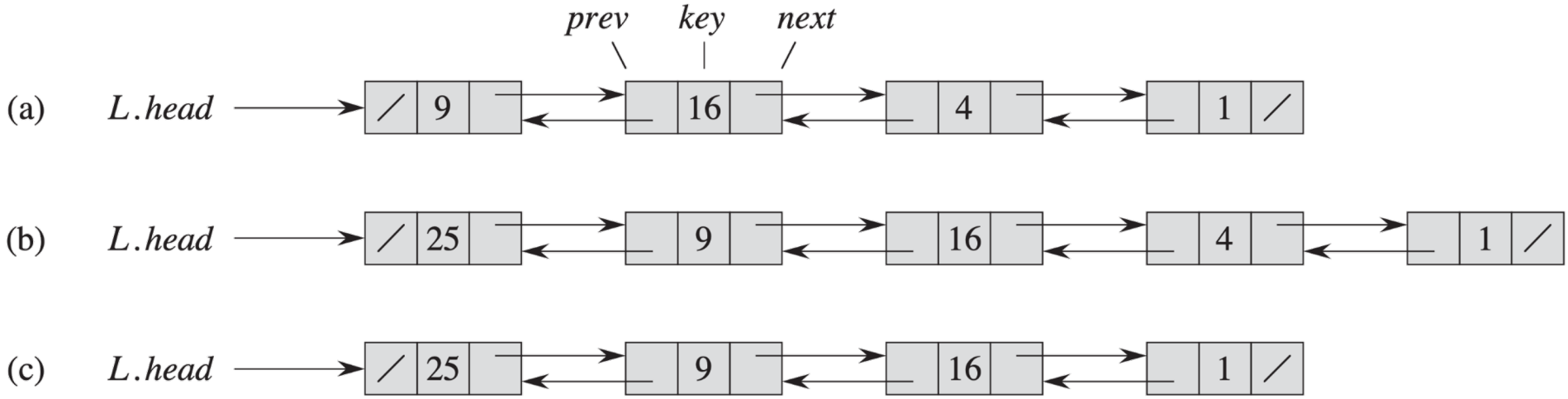
- 给定一个已经设置好关键字的元素 $x$，将 $x$ “链接”到链表的前端
- 运行时间为 $O(1)$



# 链表的删除

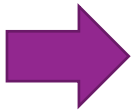
```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5      x.next.prev = x.prev
```

- 通过修改一些指针，将 $x$ “删除出”链表，运行时间为 $O(1)$
- 若要删除给定关键字的元素，则需先花 $O(n)$ 调用 $LIST\text{-}SEARCH$ 来找到 $x$



# 哨兵

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5      x.next.prev = x.prev
```

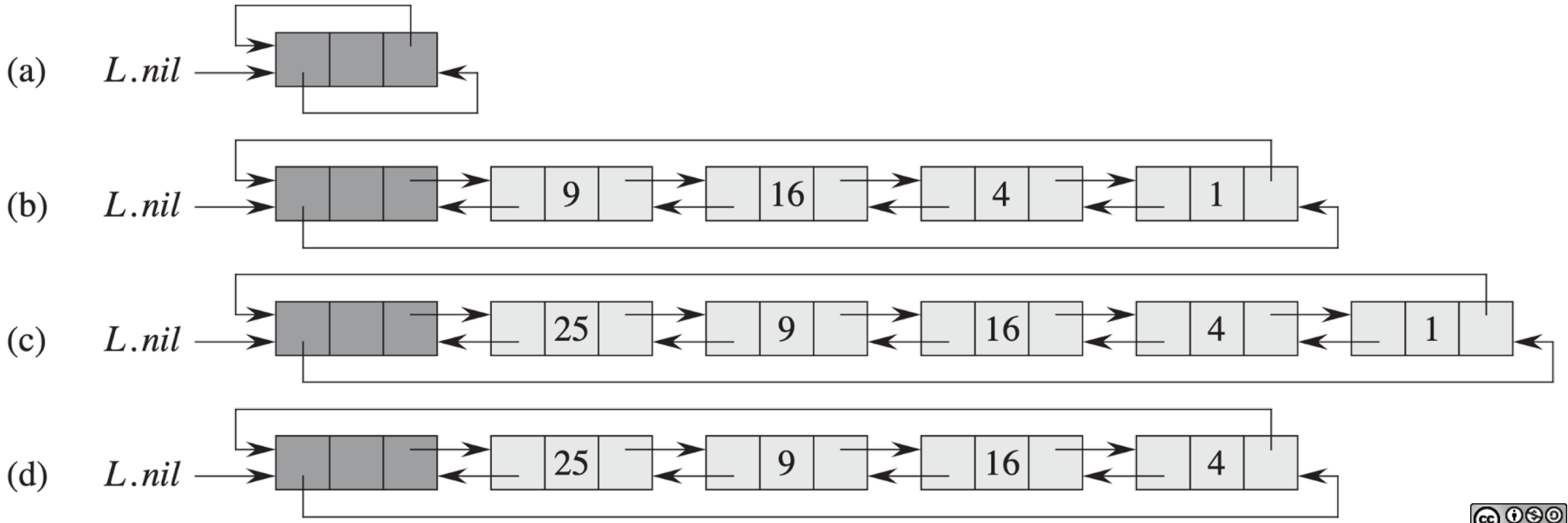


```
LIST-DELETE'(L, x)
1  x.prev.next = x.next
2  x.next.prev = x.prev
```

- 若能忽视表头和表尾处的边界条件，则 *LIST-DELETE* 的代码可以更简单些
- 哨兵（Sentinel）是一个哑对象（Dummy Object），其作用是简化边界条件的处理。

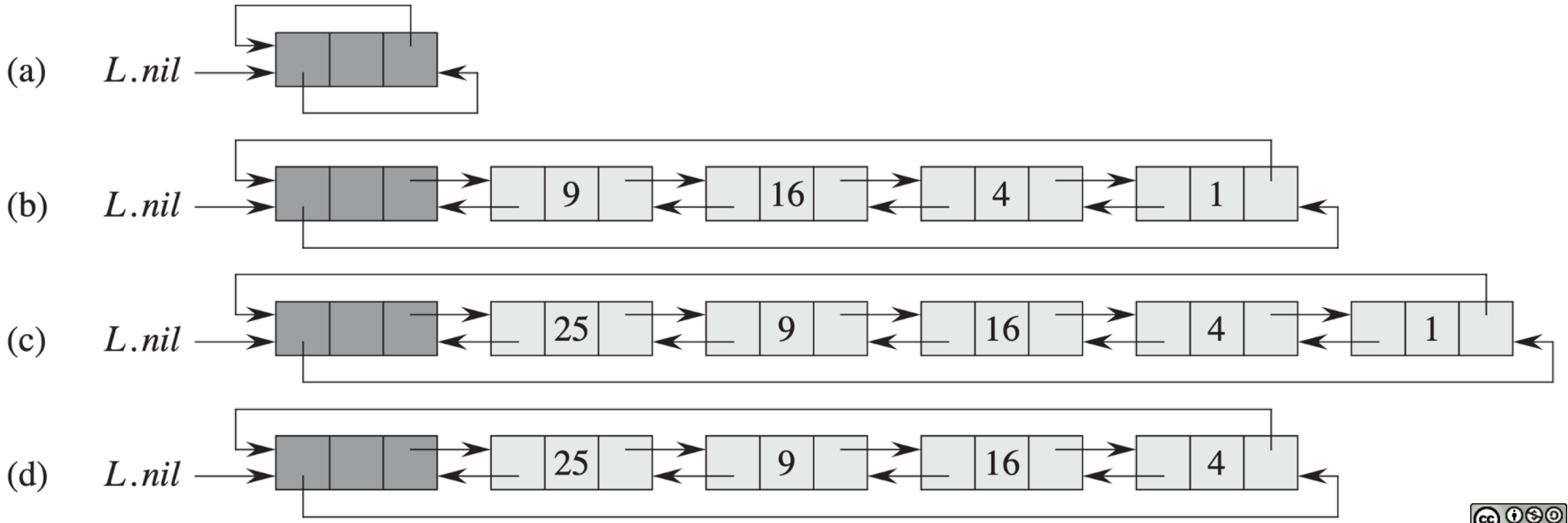
# 有哨兵的双向循环链表

- 哨兵（Sentinel）是一个哑对象（Dummy Object），其作用是简化边界条件的处理。
  - 在链表 $L$ 中设置一个对象 $L.nil$，该对象代表 $NIL$，但也有和其他对象相同的各个属性
  - 对于链表 代码中出现的每一处 $NIL$，都代之以对哨兵 $L.nil$ 的引用



# 有哨兵的双向循环链表

- 哨兵 $L.nil$ 位于表头和表尾之间
  - 属性 $L.nil.next$ 指向表头，可以用来代替 $L.head$
  - 属性 $L.nil.pre$ 指向表尾
  - 表头的 $prev$ 属性和表尾的 $next$ 属性同时指向 $L.nil$
  - 空链表只有一个哨兵，$L.nil.prev = L.nil.next = L.nil$



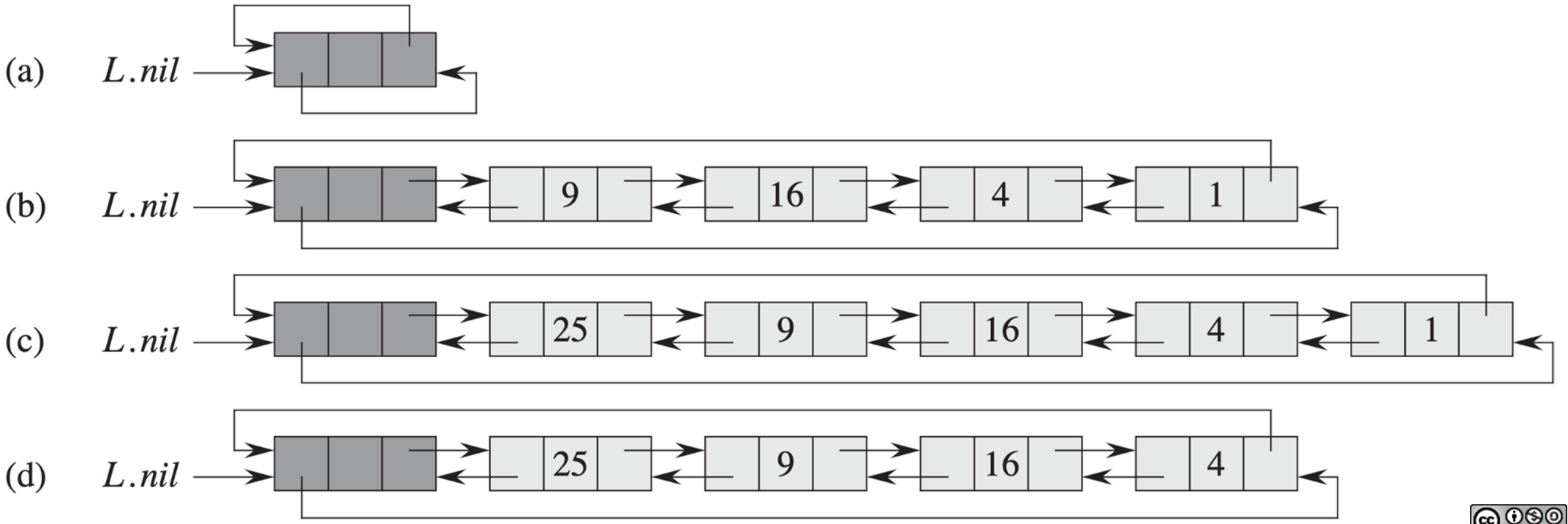
# 哨兵的双向循环链表-删除

LIST-DELETE$(L, x)$

```
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

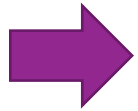
LIST-DELETE′$(L, x)$

```
1  x.prev.next = x.next
2  x.next.prev = x.prev
```

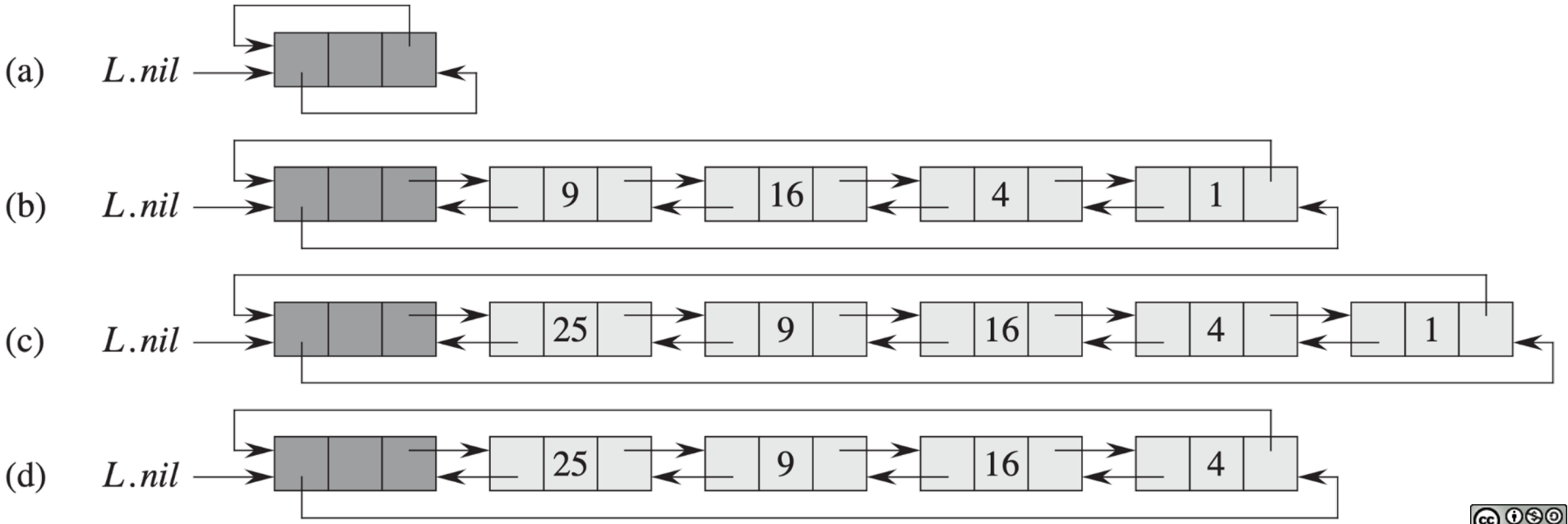


# 哨兵的双向循环链表-查找

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

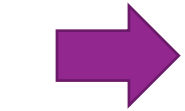


```
LIST-SEARCH′(L, k)
1  x = L.nil.next
2  while x ≠ L.nil and x.key ≠ k
3      x = x.next
4  return x
```

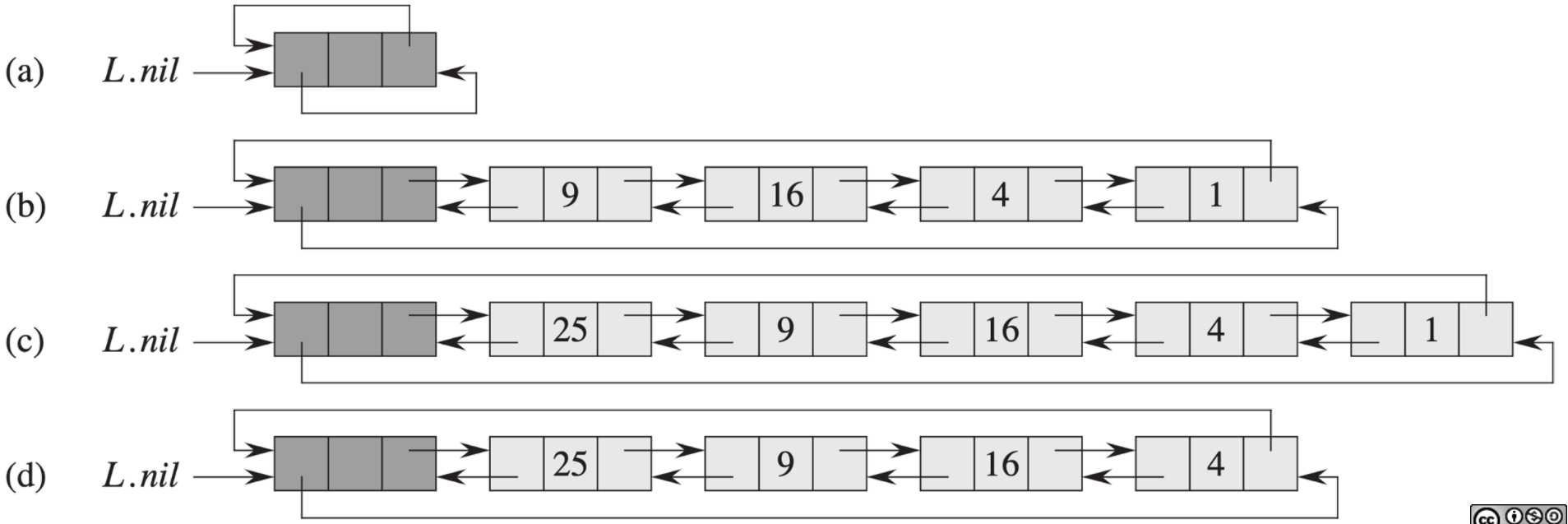


# 哨兵的双向循环链表-插入

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
5  x.prev = NIL
```



```
LIST-INSERT′(L, x)
1  x.next = L.nil.next
2  L.nil.next.prev = x
3  L.nil.next = x
4  x.prev = L.nil
```



# 哨兵

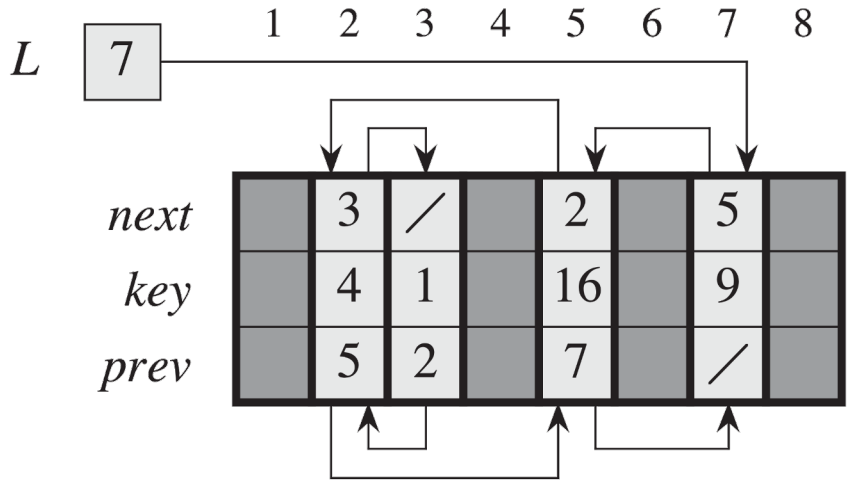
- 哨兵基本不能降低数据结构相关操作的渐进复杂度，但可以降低常数因子。
- 使用哨兵的好处往往在于可以使代码简洁，减少特判，而非提高速度。
- 我们应当审慎使用哨兵。
  - 假如有许多个很短的链表，它们的哨兵所占用的额外存储空间会造成严重的存储浪费；
  - 比如在实现链接法散列表时，每个桶内的链表就不应当使用哨兵。

# 指针和对象的实现

当语言不支持指针和对象数据类型时

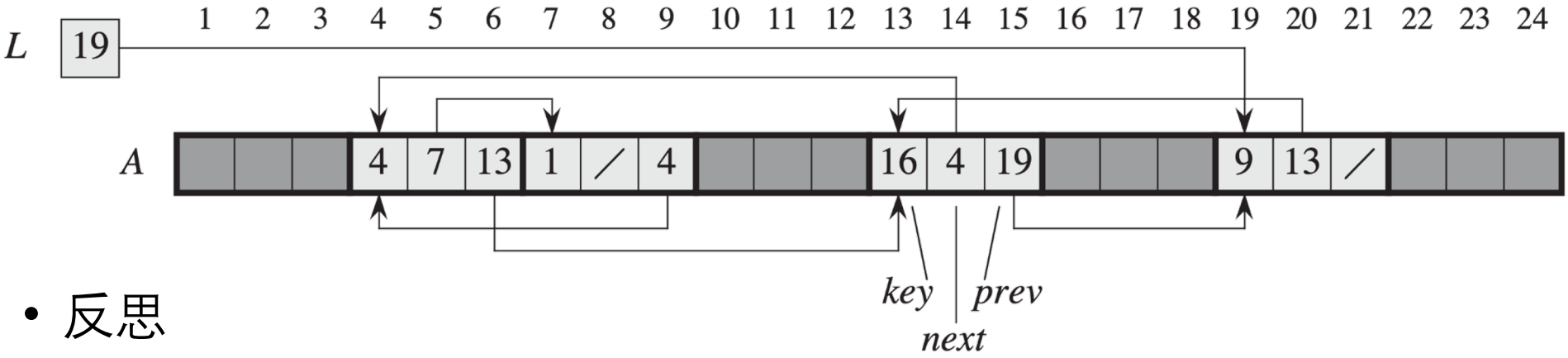
# 对象的多数组表示

- 对每个属性使用一个数组表示，可以表示一组有相同属性的对象
- 以链表为例：
  - 数组 $key$ 存放该动态集合中现有的关键字
  - 前驱后继指针则分别存储在数组 $next$ 和 $prev$ 中
  - 对于一个给定的数组下标 $x$，$key[x]$、$next[x]$ 和 $prev[x]$ 一起表示链表中的一个对象
  - 变量 $L$ 存放表头元素的下标
- 反思
  - 多数组表示只能管理同构对象
  - 动态集合大多由同构元素组成



# 对象的单数组表示

- 一个对象在计算机内存中占据一组连续的存储单元
  - 指针：该对象所在的第一个存储单元的地址
  - 属性访问：指针 + 偏移量
  - 我们可以采用同样的策略来实现对象（伪代码实现见*练习9-2问题1*）



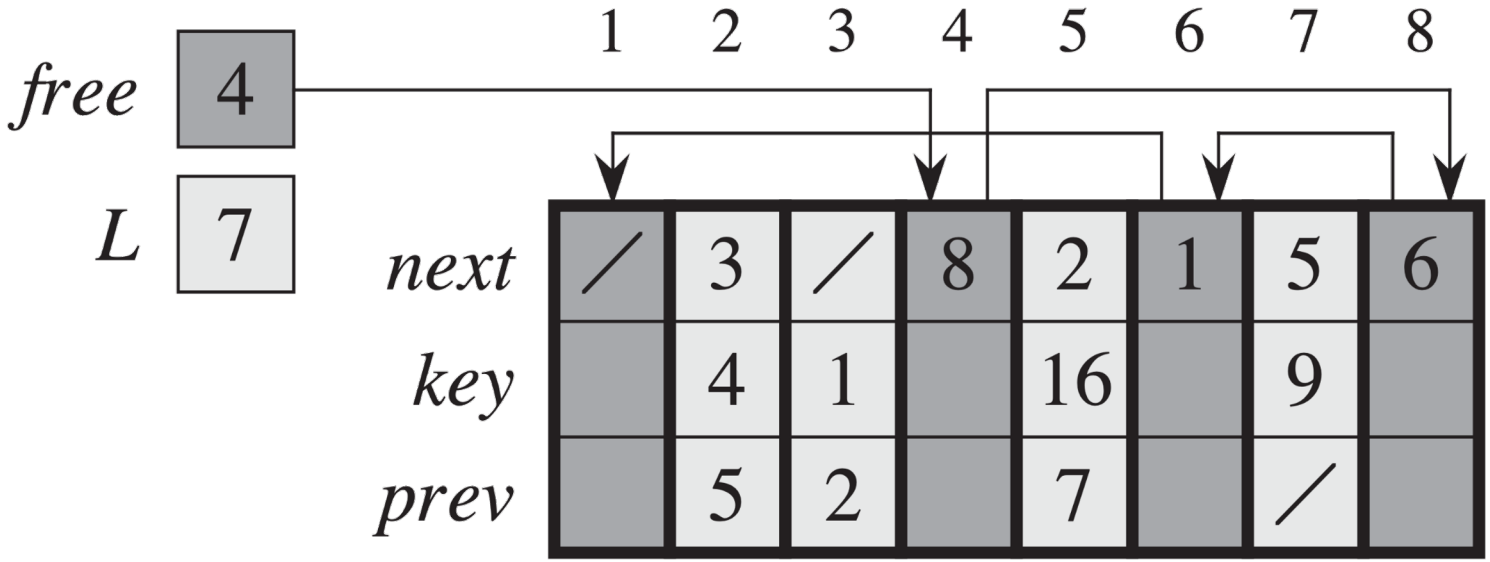

- 反思
  - 单数组表示法更加灵活，允许不同长度的对象存储于同一数组中
  - 管理一组异构的对象比管理一组同构的对象更困难

# 对象的分配与释放

- 空间管理
  - 使尚未利用的对象空间能够被分配（Allocate）
  - 使已经分配的对象空间能够被释放（Free）
  - 某些系统中，由垃圾收集器（Garbage Collector）负责管理内存空间
  - 许多简单的应用可以自己负责这个过程
- 同构对象的分配与释放（以多数组表示的双向链表为例）
  - 假设多数组表示法中各数组长度为 $m$，在某一时刻该动态集合含有 $n \leq m$ 个元素，则 $n$ 个对象代表现存于该动态集合的元素；
  - 余下 $m-n$ 个对象是自由的（Free），这些自由对象可用来表示将要插入该动态集合的元素。

# 对象的分配与释放

- 核心想法：把自由对象保存在一个单链表中—自由表（Free List）
  - 自由表只使用 $next$ 数组，表示每个自由对象在自由表中的后继
  - 自由表的头保存在全局变量 $free$ 中
  - 当由链表 $L$ 表示的动态集合非空时，自由表可能会和链表 $L$ 相互交错
  - 自由表类似于一个栈：下一个被分配的对象就是最后被释放的那个

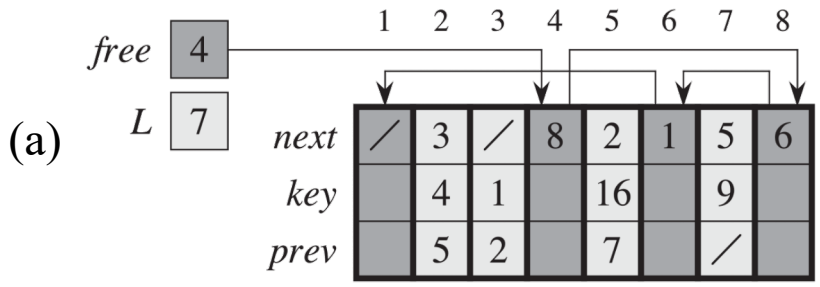


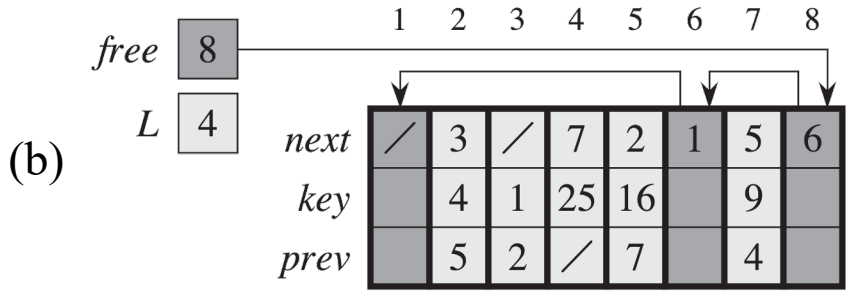
# 对象的分配与释放

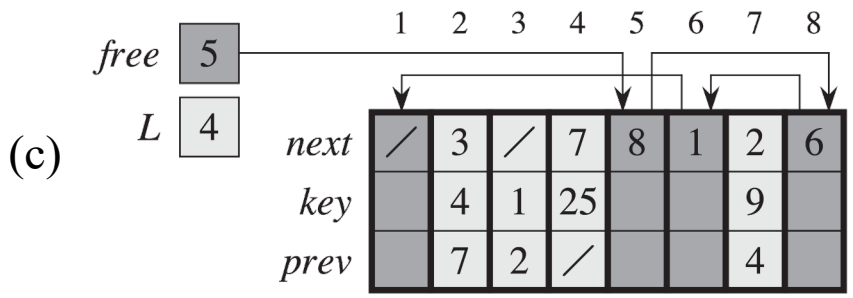
- 核心想法：把自由对象保存在一个单链表中—自由表（Free List）
  - 自由表类似于一个栈：下一个被分配的对象就是最后被释放的那个

```
ALLOCATE-OBJECT()
1  if free == NIL
2        error "out of space"
3  else x = free
4       free = x.next
5       return x
```

```
FREE-OBJECT(x)
1  x.next = free
2  free = x
```



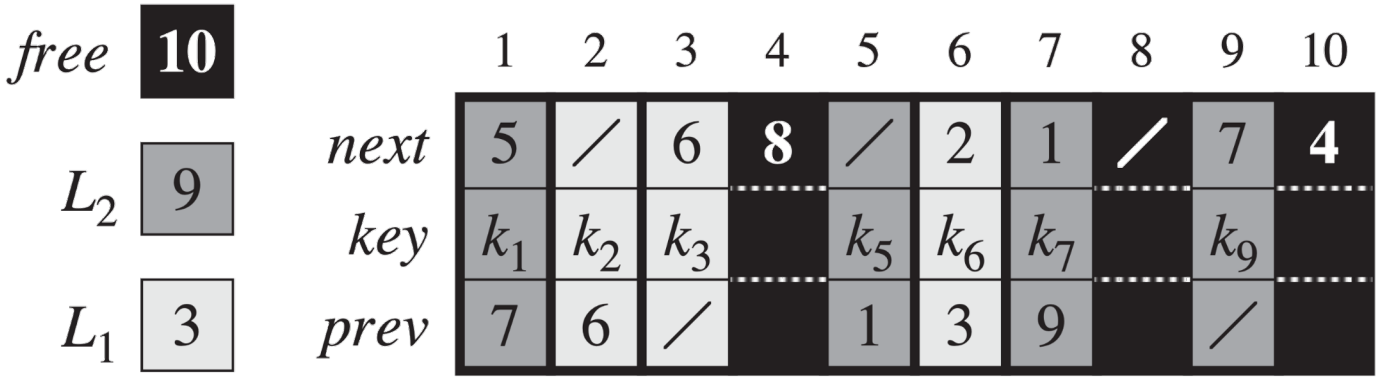



# 对象的分配与释放

```
ALLOCATE-OBJECT()
1  if free == NIL
2       error "out of space"
3  else x = free
4       free = x.next
5       return x
```

```
FREE-OBJECT(x)
1  x.next = free
2  free = x
```

- 运行时间 $O(1)$，非常实用，可以推广到任意的同构对象组
- 我们甚至可以让多个链表共用一个自由表



# 有根树的表示

链式数据结构表示有根树

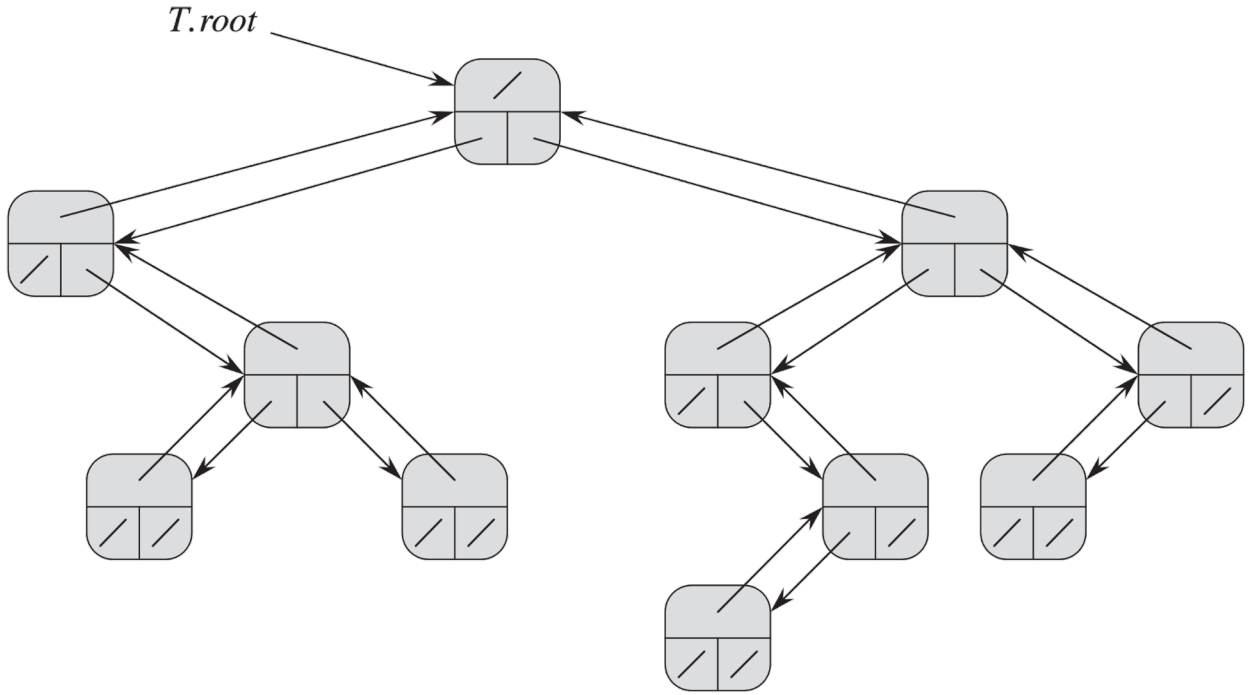
# 有根树的表示

- 树的结点用对象表示
  - 与链表类似，假设每个节点都含有一个关键字 $key$
  - 其余我们感兴趣的属性包括指向其他结点的指针的指针，它们随树的种类不同会有所变化
- 从特殊到一般
  - 二叉树
  - 结点的孩子数任意的有根树

# 二叉树

用属性 $p$、$left$ 和 $right$ 存放指向父结点、左孩子和右孩子的指针

- 如果 $x.p = NIL$，则 $x$ 是根结点
- 如果结点 $x$ 没有左孩子，则 $x.left = NIL$；右孩子的情况与此类似
- 属性 $T.root$ 指向整棵树 $T$ 的根结点；若 $T.root = NIL$ 则该树为空

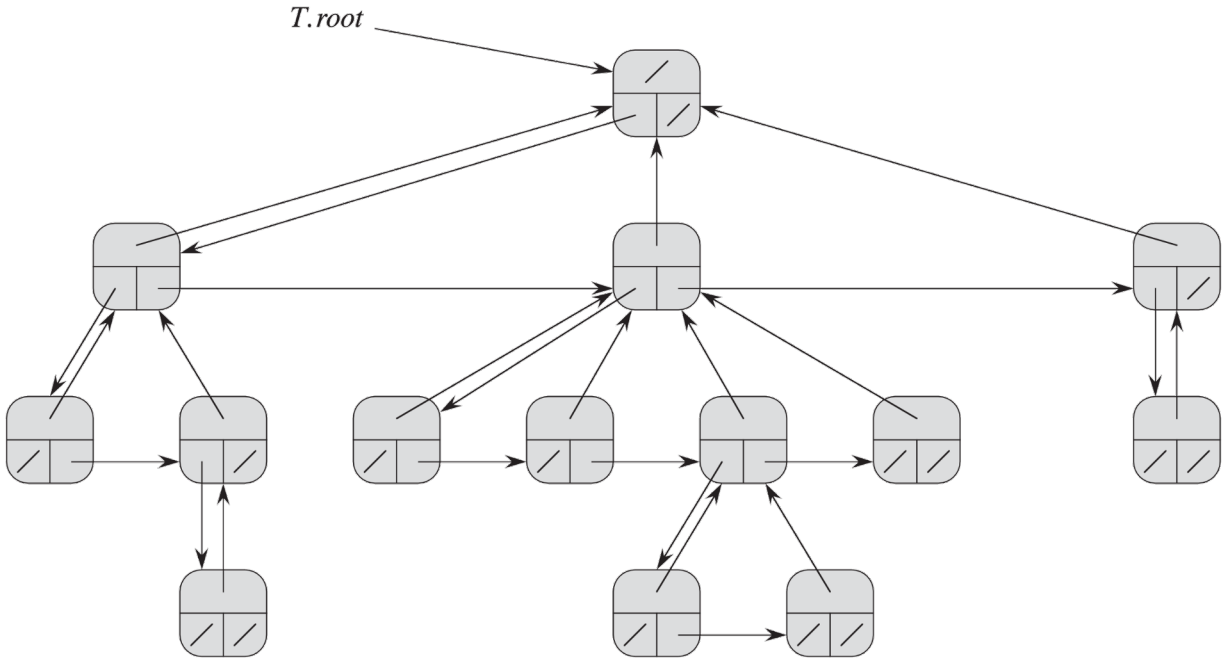


# 分支无限制的有根树

- 二叉树的表示方法可以推广到每个节点的孩子数至多为常数 $k$ 的任意类型的树（即 $k$-叉树）
  - 只需要将 $left$ 和 $right$ 属性用 $child_1, child_2, \cdots, child_k$ 代替
  - 当孩子的结点数无限制时，这种方法就失效了
    - 因为我们不知道预先分配多少个属性（在多数组表示法中就是多少个数组）
  - 即使孩子数 $k$ 限制在一个大的常数以内，但若多数结点只有少量的孩子，则会浪费大量存储空间
- 一个巧妙的方法：
  - 左孩子右兄弟表示法（Left-child, Right-sibling Representation）
  - 对于任意 $n$ 个结点的有根树，只需要 $O(n)$ 的存储空间

# 左孩子右兄弟表示法

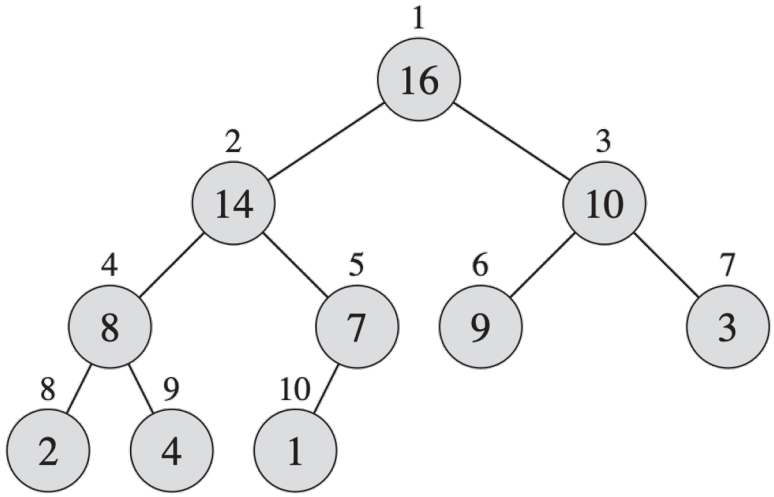
- $T.root$ 指向树的根结点，对于每个结点 $x$
  - $x.p$ 指向 $x$ 的父结点
  - $x.left\text{-}child$ 指向 $x$ 最左边的孩子结点，没有孩子则为 $NIL$
  - $x.right\text{-}sibling$ 指向 $x$ 右侧相邻的兄弟结点，没有右兄弟则为 $NIL$



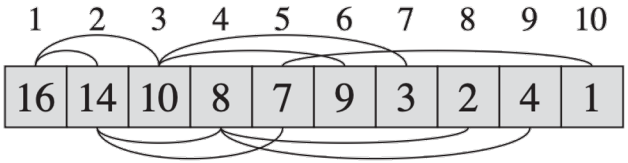

- 核心思想：用链表来表示任意数量的孩子结点

# 树的其他表示方法

- 第 5 讲：堆排序
  - 完全二叉树的单数组表示




(a)




(b)

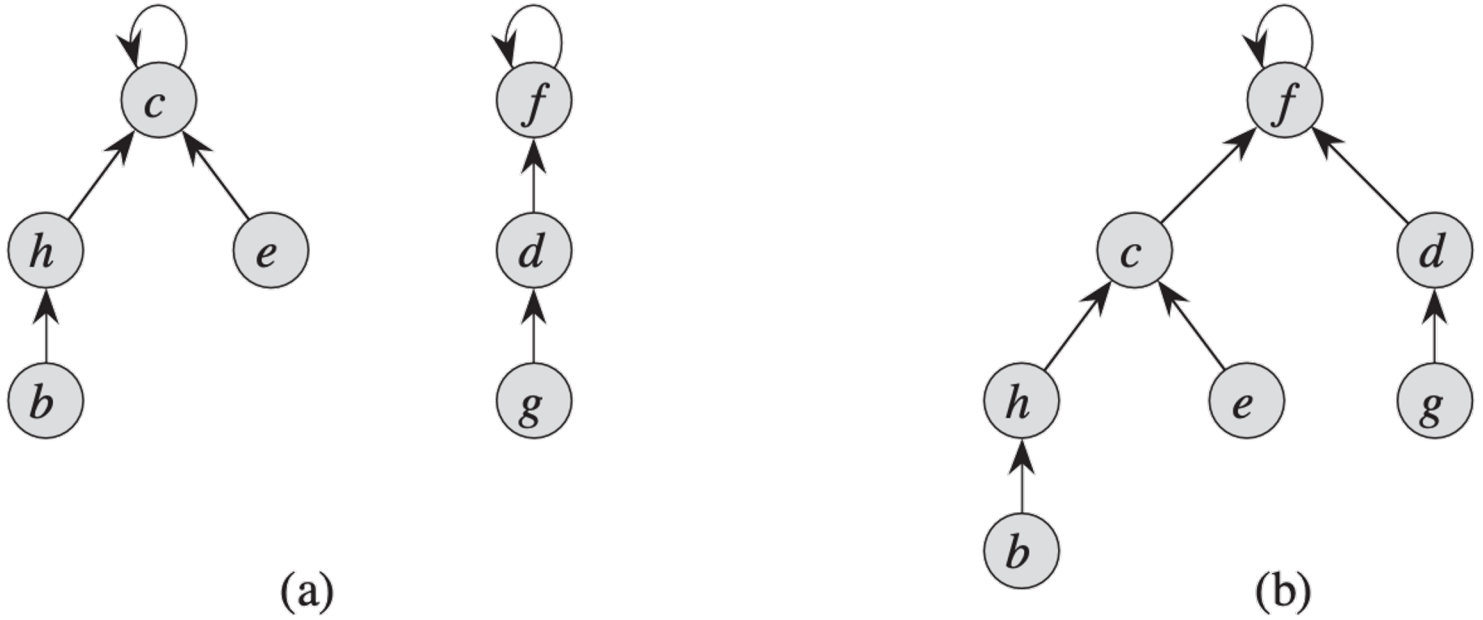
```
PARENT(i)
1  return ⌊i/2⌋
```

```
LEFT(i)
1  return 2i
```

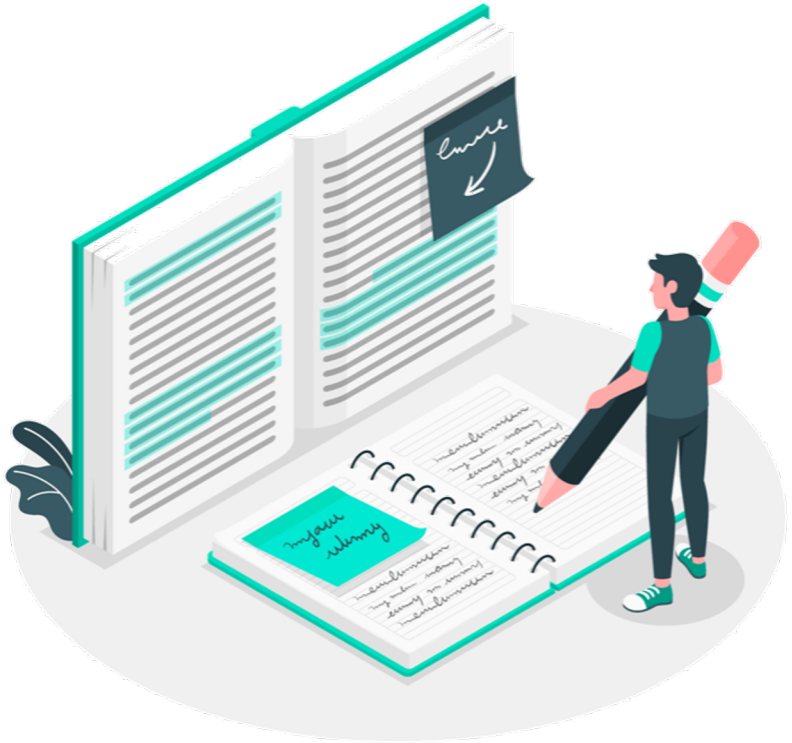
```
RIGHT(i)
1  return 2i + 1
```

# 树的其他表示方法

- 第 20 讲：并查集
  - 并查集森林的单数组表示
  - 对于结点 $1, 2, \cdots, n$， 用数组 $parent[1..n]$ 来表示其父结点
  - $x$ 的父结点为 $parent[x]$



# 本讲小结



# 内容提要

- 栈和队列
  - 比较、操作、表示、实现
- 链表
  - 表示、搜索、插入、删除、哨兵
- 指针和对象的实现
  - 多数组表示、单数组表示、分配与释放
- 有根树的表示
  - 二叉树
  - 分支无限制的有根树：左子女右兄弟表示法
  - 树的其它表示方法

# The End!