算法设计与分析

主讲人：吴庭芳

Email: *tfwu@suda.edu.cn*

苏州大学 计算机学院







第七讲 基本数据结构

内容提要：

- 栈和队列
- 链表
- 指针和对象的实现
- 有根树的表示

动态集合

- 集合是计算机科学的基础，如同在数学中所起的作用。
- 数学中的集合是不变的，而计算机科学中由算法操作的集合却在整个过程中能增大、缩小或者发生其他变化，称为动态集合。
- 字典：支持在一个集合中插入和删除元素以及测试元素是否属于集合的操作的动态集合被称为字典。。
- 动态集合的元素：每个元素都由一个对象来表示，如果有一个指向对象的指针，能够实现对各个属性的检查和操作。
- 一些类型的动态集合假定对象中的一个属性为标识关键字（key）。如果关键字全不相同，可以将动态集合视为一个关键字值的集合。对象可能包含卫星数据，与其他对象属性一起移动。

动态集合

- 动态集合上的操作：
  - 查询操作：简单返回有关集合信息的操作，包括SEARCH操作，MINIMUM操作等；
  - 修改操作：改变集合的操作，包括INSERT操作，DELETE操作。







第七讲 基本数据结构

内容提要：

- 栈和队列
- 链表
- 指针和对象的实现
- 有根树的表示

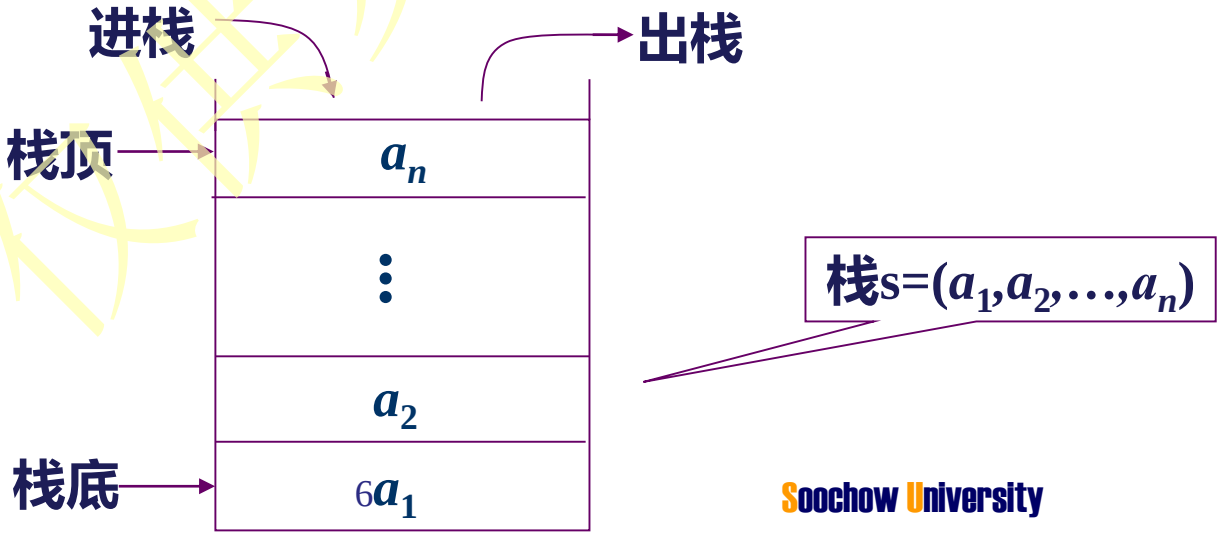
栈和队列







- 栈和队列都是动态集合。
- 栈的定义和特点：
  - 定义：限定仅在表尾进行插入或删除操作的线性表，表尾-**栈顶**，表头-**栈底**，不含元素的空表称**空栈**
  - 特点：**先进后出**（FILO）或**后进先出**（LIFO）



栈和队列

- **栈的操作：**
  - **INSERT操作：压入（PUSH），e.g., PUSH(*S*, *x*)**
  - **DELETE操作：弹出（POP），无元素参数的操作，所移除的元素是预先设定的，被删除的是最近插入的元素。**
- **通过使用数组来实现栈：用一个数组 *S*[1..*n*] 来实现一个最多容纳 *n*个元素的栈。**
  - **属性 *S.top*：指向最新插入的元素；**
  - **栈中包含的元素为 *S*[1..*S.top*]，其中 *S*[1] 是栈底元素，*S*[*S.top*] 是栈顶元素。**

栈和队列

- 当 *S.top* = 0 时，栈中不包含任何元素，即栈是空(empty)的。
- 测试一个栈是否为空可以用查询操作 STACK-EMPTY。

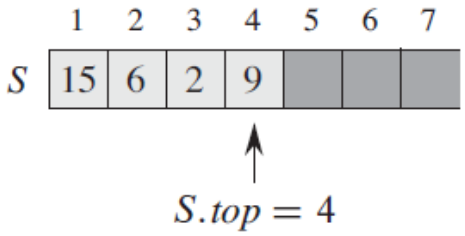
```
STACK-EMPTY(S)
1 if  S.top == 0
2       return TRUE
3 else return FALSE
```

- 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）。

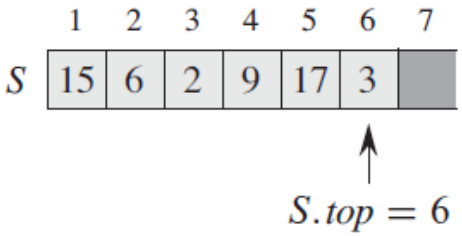
```
PUSH(S, x)
1 S.top = S.top + 1
2       S[S.top]=x
```

```
POP(S)
1 if STACK-EMPTY(S)
2         error “underflow”
3 else S.top = S.top - 1
4         Return S[S.top+1]
```

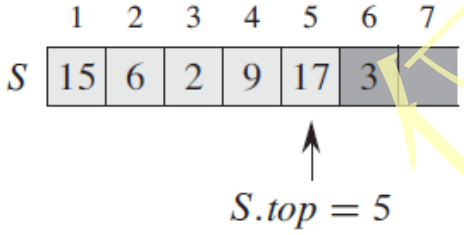
栈和队列



- 栈 *S* 有4个元素，栈顶元素为9。



- 调用PUSH(*S*, 17) 和PUSH(*S*, 3)后的栈。



- 调用POP(*S*) 并返回最后压入的元素3的栈 *S*。虽然元素3仍在数组里，但它已不在栈内了，此时在栈顶的元素是17。

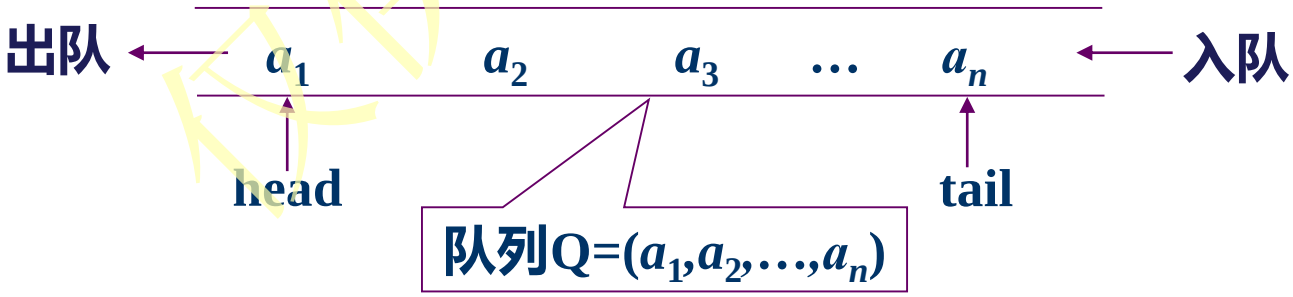
栈和队列







- 队列的定义和特点：
  - 定义：队列是限定只能在表的一端进行插入，在表的另一端进行删除的线性表
  - 队尾(tail)——允许插入的一端
  - 队头(head)——允许删除的一端
  - 特点：先进先出（FIFO）



栈和队列

- 栈的操作：
  - INSERT操作：入队（ENQUEUE），e.g., ENQUEUE(S, x)
  - DELETE操作：出队（DEQUEUE），无元素参数的操作，所移除的元素是预先设定的，被删除的是**集合中存在时间最长的那个元素**。
- 通过使用数组来实现栈：用一个数组 $Q[1..n]$ 来实现一个最多容纳 n-1个元素的栈。
  - 属性 $Q.head$：指向队头元素；
  - 属性 $Q.tail$：指向下一个新元素将要插入的位置；
  - 队列中的元素存放在位置 $Q.head$，$Q.head$+1，…，$Q.tail$-1。

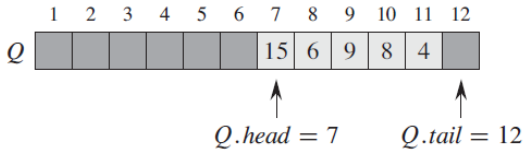
栈和队列

- 当 *Q.head* = *Q.tail* 时，队列为空。
- 初始时，*Q.head* = *Q.tail* = 1。
- 如果试图从一个空队列中删除一个元素，则队列发生下溢（underflow）。当 *Q.head* = *Q.tail*+1，队列是满的，此时若试图插入一个元素，则队列发生上溢。

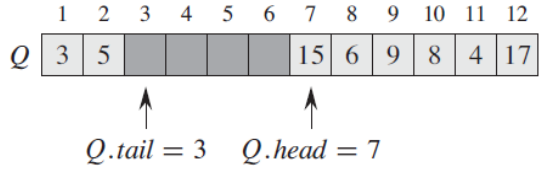
```
ENQUEUE(Q, x)
1 Q[Q.tail] = x
2 if Q.tail == Q.length
3     Q.tail = 1
4 else Q.tail = Q.tail + 1
```

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

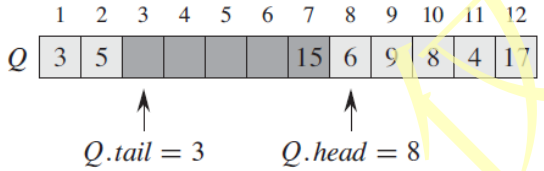
栈和队列



- 队列 Q 有5个元素，位于 $Q[7..11]$。



- 依次调用ENQUEUE(Q, 17)、ENQUEUE(Q, 3)和 ENQUEUE(Q, 5)后构成的队列。



- 调用ENQUEUE(Q) 并返回原队头的关键字15后，构成的队列。此时新的队头元素的关键字为6。







第七讲 基本数据结构

内容提要：

- 栈和队列
- 链表
- 指针和对象的实现
- 有根树的表示

链表







- 链表是一种**动态地进行存储分配**的数据结构。
- 链表存储结构的特点：
  - 用一组**任意的存储单元**存储线性表的**数据元素（关键字）**，同时还存储**下一个结点地址（指针）**，这两部分信息组成为**结点**。
  - 结点的组成：

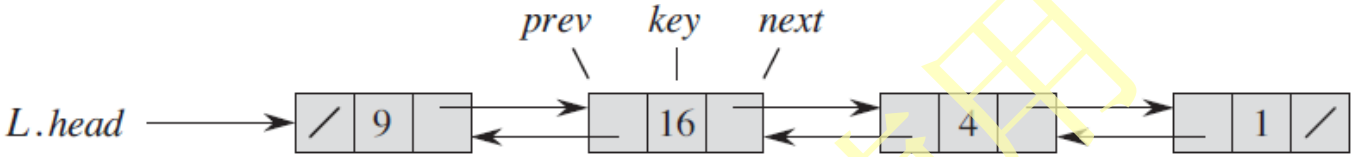
data	link

  其中：data是数据域，用来存放关键字；

  link是指针域，用来存放下一个结点的地址。

  - **结点物理存储顺序与其逻辑顺序不一定相同。**

链表



- **双向链表的每个元素包含关键字和两个指针：*next* 和 *prev***
  - ***x.next*** **指向它在链表中的后继元素，** ***x.prev*** **指向它的前驱元素；**
  - **如果** ***x.prev*** **= NIL，则结点** ***x*** **没有前驱，因此是链表的第一个元素，即链表的头 (head)；**
  - **如果** ***x.next*** **= NIL，则结点** ***x*** **没有后继，因此是链表的最后一个元素，即链表的尾 (tail)；**
  - **属性** ***L.head*** **指向链表的第一个元素。如果** ***L.head*** **= NIL，则链表为空。**

链表

- 链表的分类：
  - **单链表**(singly linked)：省略双链表中每个元素的 *prev* 指针；
  - **已排序链表**：链表的线性顺序与链表元素中关键字的线性顺序一致；因此，最小的元素就是表头元素，最大的元素则是表尾元素；
  - **循环链表**：表头元素的 *prev* 指针指向表尾元素，而表尾元素的 *next* 指针指向表头元素，即 $L.head.prev = L.tail$，$L.tail.next = L.head$。

链表

- 链表的操作
  - 链表的搜索 LIST-SEARCH(L, k)：用于查找链表 L 中第一个关键字为 k 的元素，并返回该元素的指针。如果链表中没有关键字 k 的元素，则返回 NIL。

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

  - 要搜索一个有 n 个对象的链表，过程LIST-SEARCH 在最坏情况下的运行时间是$\Theta(n)$，因为可能需要搜索整个链表。

链表

- 链表的插入 LIST-INSERT(L, x)：给定一个已设置好关键字 key 的元素 x，过程LIST-INSERT 将 x “连接”到链表的前端。

```
LIST-INSERT(L, x)
1 x.next = L.head
2 if L.head ≠ NIL
3       L.head.prev = x  // 表示L.head所指向的对象的prev属性
4  L.head = x
5   x.prev = NIL
```

LIST-INSERT(L, x)
$x.key$=25

- 在一个含 n 个元素的链表上执行LIST-INSERT 的运行时间是 O(1)。

链表

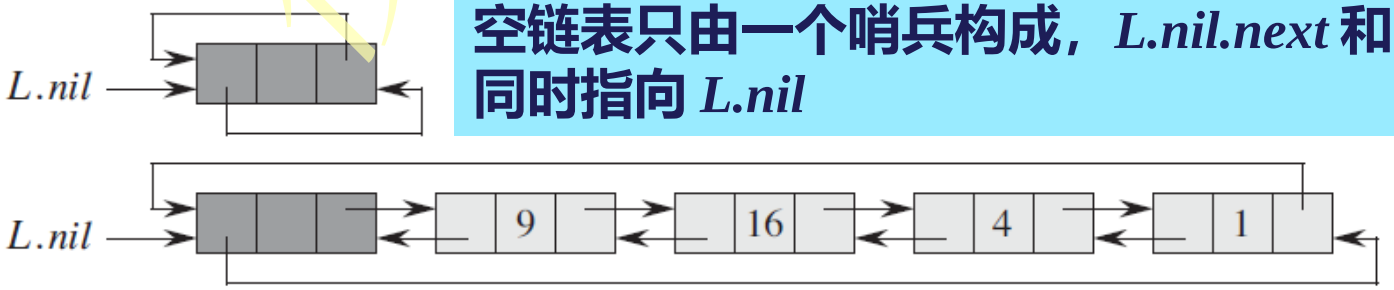
- 链表的删除 LIST-DELETE(L, x)：将一个元素 x 从链表 L 中移除。该过程要求给定一个指向 x 的指针，然后通过修改一些指针，将 x “删除出”该链表。如果要删除具有给定关键字值的元素，则必须先调用 LIST-SEARCH 找到该元素。

```
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
```

- 如果要删除具有给定关键字的元素，则最坏情况运行时间是$\Theta(n)$。

链表

- 哨兵(sentinel)：是一个哑对象，其作用是简化表头和表尾处边界条件的处理。
- 假设在链表 L 中设置一个对象 $L.nil$，该对象代表 NIL，但也具有和其他对象相同的各个属性。
- 将一个常规的双向链表转变为一个有哨兵的双向循环链表，哨兵 $L.nil$ 位于表头和表尾之间。
- $L.nil$ 属性 $L.nil.next$ 指向表头，$L.nil.prev$ 指向表尾。表尾的 $next$ 属性和表头的 $prev$ 属性同时指向 $L.nil$。因为 $L.nil.next$ 指向表头，可以去掉属性 $L.head$，并把对其的引用代替为对 $L.nil.next$ 的引用。



空链表只由一个哨兵构成，$L.nil.next$ 和 $L.nil.prev$ 同时指向 $L.nil$

链表

- 链表的搜索：从 $n = L.nil.next$ 开始遍历整个链表，知道 key 等于给定参数或 $n = L.nil$。

```
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
```

- 链表的插入：插入到 $L.nil$ 和 $L.nil.next$ 中间

```
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
```

链表

- 链表的删除：不再需要边界界的判断

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

- 哨兵节点不能降低数据结构相关操作的渐近时间界，但可以降低常数因子，比如LIST-DELETE'和LIST-INSERT'都节约了O(1)。
- 在循环语句中，使用哨兵的好处往往在于可以**使代码简洁**，而非提高速度。
- 但应当慎用哨兵。假设存在很多的短小链表，那么再给每个链表配上一个哨兵就不划算了，由于哨兵要占用额外的存储空间，而短小的链表很多时，就造成了严重的浪费。







第七讲 基本数据结构

内容提要：

- 栈和队列
- 链表
- 指针和对象的实现
- 有根树的表示

指针和对象的实现





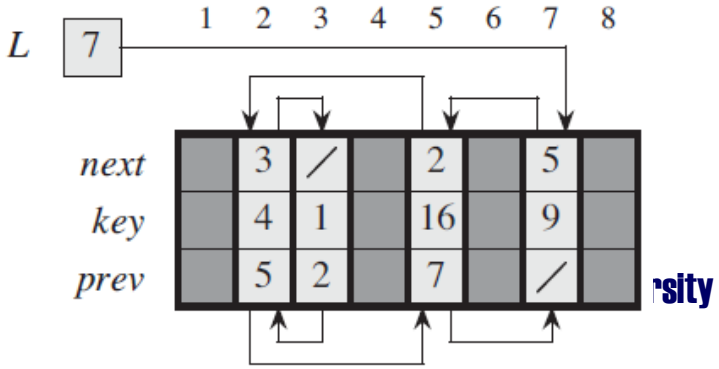


- 常用结构体、指针、对象这类数据类型储存链表。
- 但当程序语言不支持这些数据类型时，可以用**单数组或多数组**的形式实现链式数据结构，即通过数组和数组下标来构造对象、指针。
- 对象的多数组表示：对一组具有相同属性的对象，每一个属性都可以用一个数组来表示。

指针和对象的实现

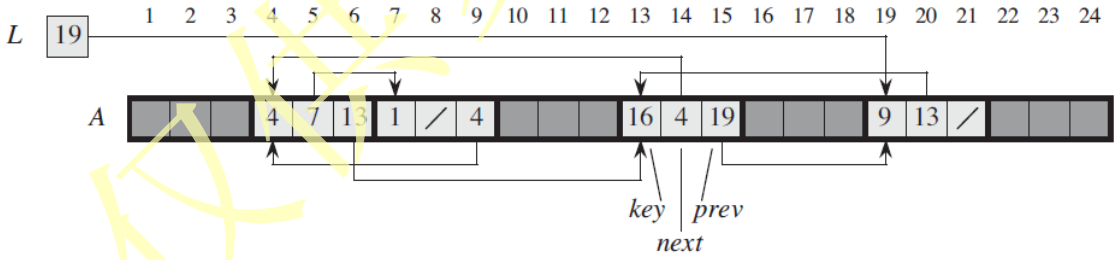
- 用三个数组实现双向链表：
  - 数组 *key* 存放动态集合中现有的关键字；
  - 指针则分别存放在数组 *next* 和 *prev* 中；
  - 对于一个给定的数组下标 *x*，三个数组项 *key*[*x*]、*next*[*x*] 和 *prev*[*x*] 一起表示链表中一个对象。即数组下标 *x* 为数组 *key*、*next* 和 *prev* 的一个共同下标。
  - 常数 NIL 出现在表尾的 *next* 属性和表头的 *prev* 属性，通常用一个不能代表数组中任何实际位置的整数（0或者-1）表示。变量 *L* 存放表头元素的下标。

key[2]=4，*key*[5]=16
prev[2]=5，*next*[5]=2
L=7



指针和对象的实现

- 对象的单数组表示：一个对象占据存储中的**一组连续位置**，指针即指向某对所占存储区的**第一个位置**，后续位置可以**通过加上相应的偏移量进行寻址**。
- 用单个数组实现双向链表：
  - 一个对象占用一段连续的子数组 $A[j..k]$，对象中的每个属性对应于从0到 $k-j$ 之间的偏移量，指向该对象的指针就是下标 j；



对应于属性 *key*、*next* 和 *prev* 的偏移量分别为0、1和2。给定一个指针 i，读取 $i.prev$ 的值，只需要在指针的值 i 加上偏移量2，读取的是 $A[i+2]$。

指针和对象的实现

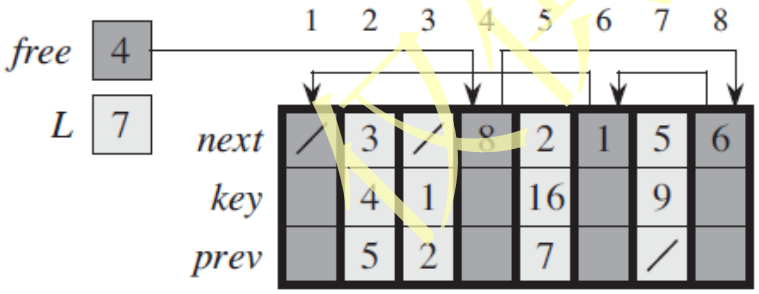
- 单数组的表示法比较灵活，因为它允许不同长度的对象存储于同一数组中。
- 而管理一组**异构的对象**比管理一组**同构的对象**（即所以对象有相同的属性）更困难。
- 实际应用中，考虑的数据结构大多都是同构的元素构成，因此采用对象的多数组表示法即可满足要求。

指针和对象的实现

- 对象的分配与释放：
  - 向一个双向链表表示的动态集合中插入一个关键字，必须分配一个指向该链表表示中尚未利用的对象的指针。
  - 某些系统中，**垃圾收集器**（garbage collector）负责确定哪些对象是未使用的。
  - 假设多数组表示法中的各数组长度为 m，且在某一时刻该动态集合含有 $n \le m$ 个元素。则 n 个对象代表现存于该动态集合中的元素，而余下的 $m-n$ 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。

指针和对象的实现

- 对象的分配与释放：
  - 将自由对象保存在一个单链表中，称为自由表（free list）。自由表只使用 *next* 数组，该数组只存储链表中的 *next* 指针。自由表的表头保存在全局变量 *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。但该表示中的每个对象不是在链表 L 中，就在自由表中，但不会同时属于两个表。

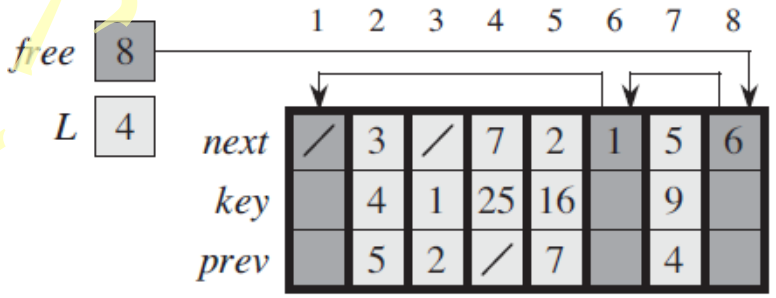


浅阴影部分表示链表 L，
深阴影部分表示自由表。

指针和对象的实现

- 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。可以分别利用栈操作 PUSH 和 POP 的链表实现形式来实现分配和释放对象的过程。假设下述过程中的全局变量 *free* 指向自由表的第一个元素。

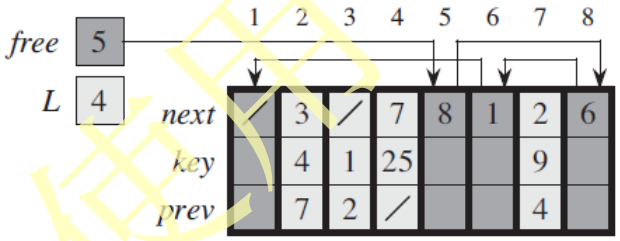
```
ALLOCATE-OBJECT( )
1  if free == NIL
2        error "out of space"
3  else x = free
4        free = x.next
5  return x
```



调用 ALLOCATE-OBJECT()，返回下标 4。将 *key*[4]=25，调用 LIST-INSERT(*L*, 4) 处理。新自由表的头为原自由表中 *next*[4] 所指的对象 8。

指针和对象的实现

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



执行 LIST-DELETE(*L*, 5)，然后调用 FREE-OBJECT(5)。对象 5称为新自由表的表头，对象 8 紧随其后。

- 初始时自由表含有全部的 n 个未分配的对象。一旦自由表用完，再运行ALLOCATE-OBJECT 过程将提示出错。
- 上述两个过程运行时间都为 O(1)。







第七讲 基本数据结构

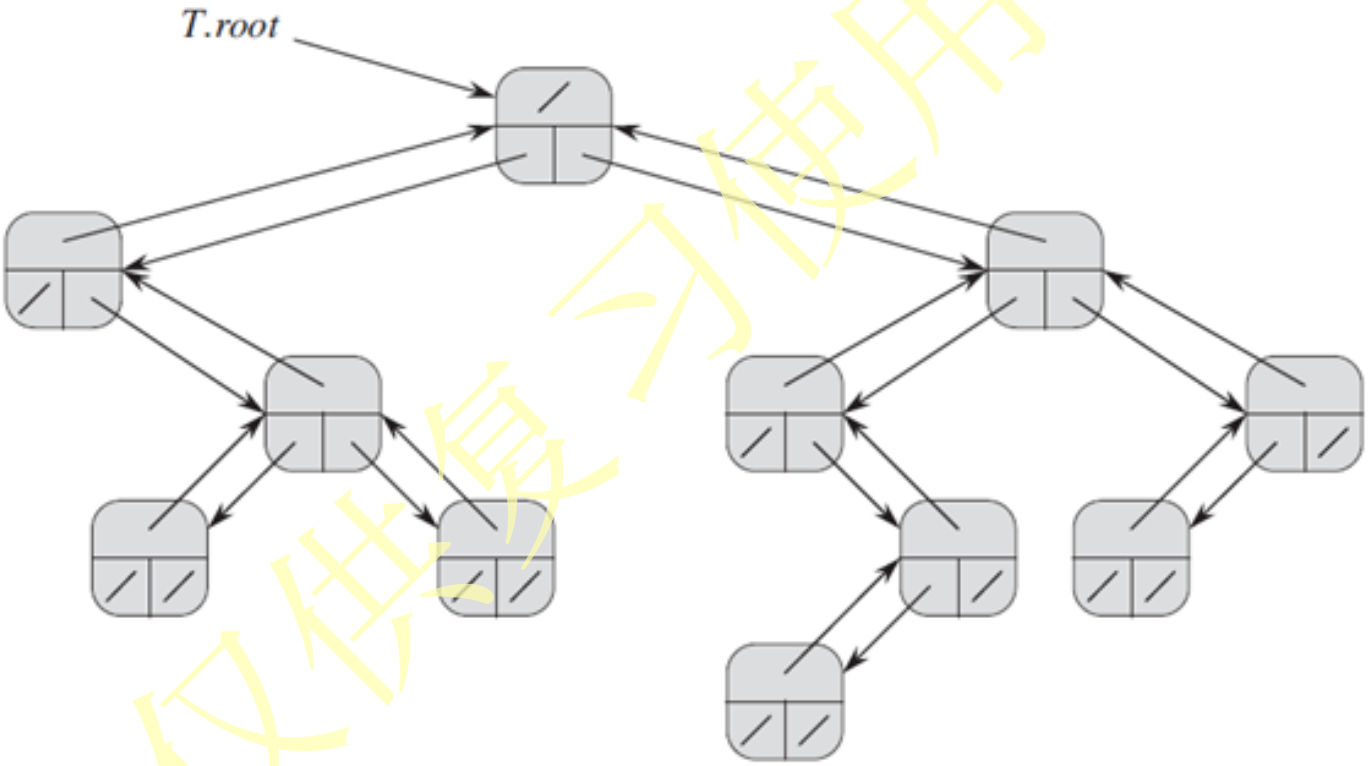
内容提要：

- 栈和队列
- 链表
- 指针和对象的实现
- 有根树的表示

有根树的表示

- 有根树：树的结点用对象表示，每个结点都含有一个关键字 *key*，以及其他感兴趣的属性，包括指向其他结点的指针。
- 二叉树：
  - 用属性 *p*、*left*、*right* 来存放指向二叉树 T 中的父结点、左孩子、右孩子的指针。
  - 如果 $x.p = \text{NIL}$，则 x 是根节点。 如果结点 x 没有左孩子，则 $x.left = \text{NIL}$，右孩子类似。
  - 属性 $T.root$ 指向整棵树 T 的根结点。如果 $T.root = \text{NIL}$，则该树为空。

有根树的表示



有根树的表示

- 分支无限制的有根树：
  - 二叉树的表示方法可以推广到每个结点的孩子数至多为常数 k 的任意类型的树：只需要将 *left* 和 *right* 属性用 $child_1$，$child_2$，…，$chid_k$ 代替即可。
  - 但是当孩子的结点数无限制时，上述表示方法失效，因为不知道应当预先分配多少个属性（在多数组表示法中就是多少个数组）。
  - 此外，即使孩子数 k 限制在一个大的常数以内，但是若多数结点只有少量的孩子，则会浪费大量存储空间。

有根树的表示

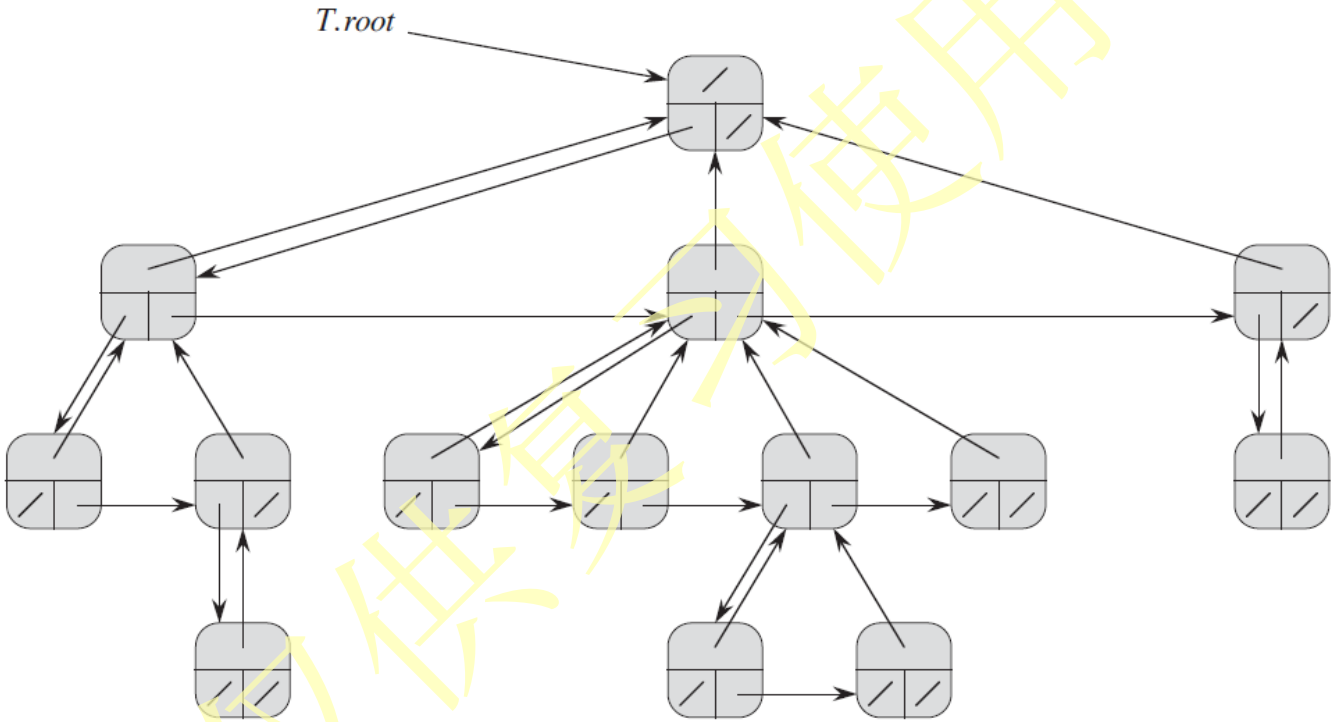
- **左孩子右兄弟表示法：**
  - 优势：对任意 n 个结点的有根树，只需要 $O(n)$ 的存储空间。
  - 每个结点都包含一个父结点指针 p，且 $T.root$ 指向树 T 的根结点。每个结点中不是包含指向每个孩子的指针，而是只有两个指针：

    1. $x.left\text{-}child$ 指向结点 x 最左边的孩子结点；

    2. $x.right\text{-}sibling$ 指向结点 x 右侧相邻的兄弟结点。

    如果结点 x 没有孩子结点，则 $x.left\text{-}child = NIL$；如果结点 x 是其父结点最右孩子，则 $x.right\text{-}sibling = NIL$。

有根树的表示









谢谢！

Q & A

作业：10.1-6，10.1-7
10.2-2，10.2-3
10.3-2
10.4-2

数据结构分类图