USTC

Chapter 3
Data Structures

王子磊 (Zilei Wang)
Email: zlwang@ustc.edu.cn
http://vim.ustc.edu.cn/

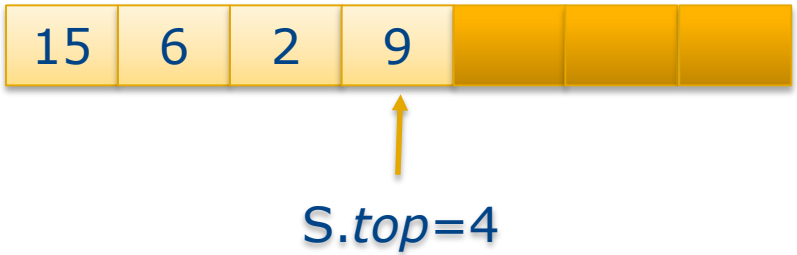
学习要点

- 基本数据结构
  - 栈、队列、链表、有根树、图
- 堆与堆排序
- 散列表 Hash Table
- 二叉搜索树
  - 红黑树

基本数据结构

栈

- 栈实现的是一种后进先出（LIFO）策略的动态集合
  - 用一个数组S[1:*n*]实现一个可容纳 *n* 个元素的栈



O(1)

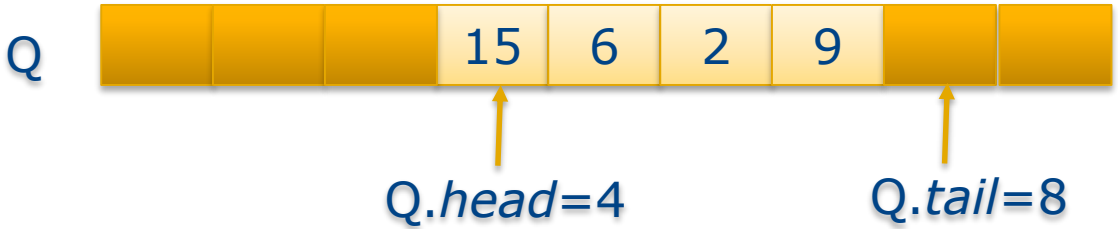
STACK-EMPTY (S)
1. if S.*top*==0
2. return True;
3. else return False;

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];

队列

- **队列**实现的是一种先进先出（FIFO）策略的动态集合
  - 用一个数组Q[1:*n*]实现一个可容纳 *n*-1个元素的队列
  - Q.*head*指向队头元素，Q.*tail*指向下一个元素要插入的位置



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;

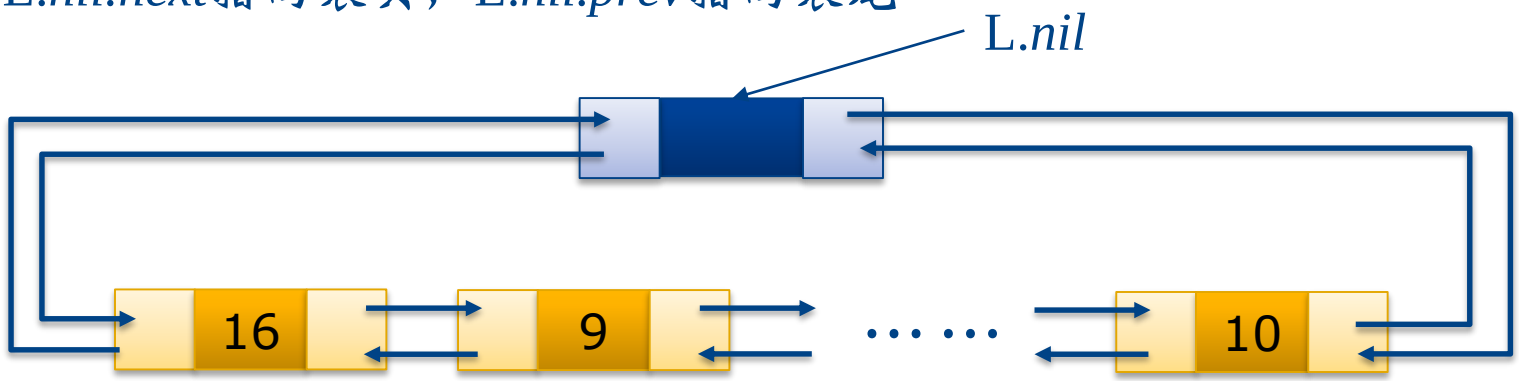
DEQUEUE(Q)

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*;

⇨ O(1)

链表

- **链表(linked list)**实现的是一种线性顺序排列的对象集合
  - 双向链表：每个元素一个*key*和两个指针：*next, prev*
  - 循环链表：表头元素和表尾元素相连Q.*head*指向队头元素，Q.*tail*指向下一个元素要插入的位置
- 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.next*指向表头，L.*nil.prev*指向表尾



链表

❖ 有哨兵的双向循环链表

- 使用哨兵的好处是使代码整洁，并提高效率
- 假定数据没有排序

LIST-SEARCH (L, *k*)

1. *x*=L.*nil.next*;

2. while $x \neq$ L.*nil* and *x.key* $\neq k$

3. $\quad x = x.next$;

4. return *x*;

O(n)

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*;

O(1)

LIST-DELETE(L, *x*)

1. *x.next.prev*=*x.prev*;

2. *x.prev.next*=*x.next*;

O(1)

有根树

- 用链式结构表示
  - 二叉树：节点x包含属性 p 和指针$left$, $right$
    - $x.p=nil$表示根节点
  - 分支无限制的有根树
    - $Child_k$ —— 难以预知孩子的节点数，分配内存麻烦
    - 左孩子右兄弟表示法

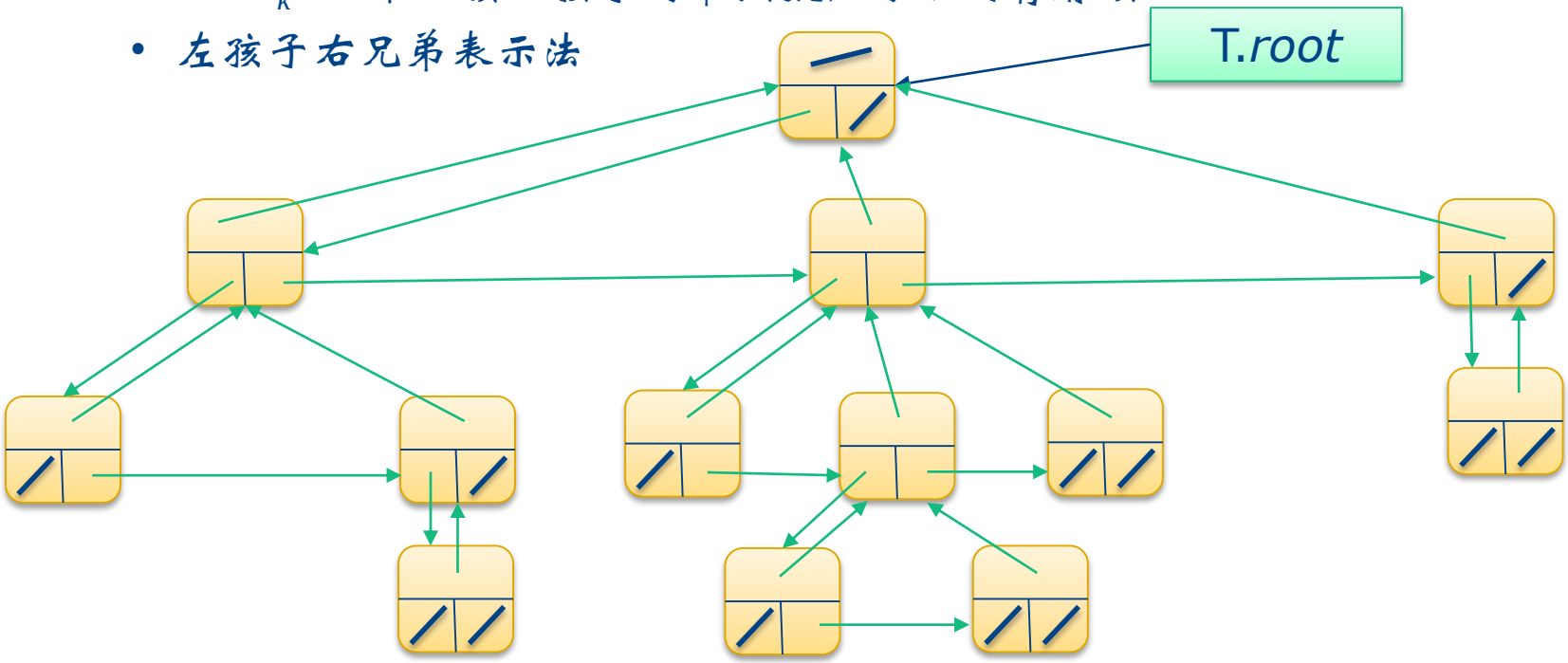


图 (Graph)

- 定义
  - 表示为 $G=(V, E)$，其中V为顶点集合，$E \subseteq V \times V$为边集合
  - 有向图：边是一个有向顶点对
  - 无向图：边是一个无向顶点对
- 特点
  - 所有的情况下 $|E| = O(V^2)$
  - 如果图示连通的(connected)，$|E| \geq |V| - 1$
  - → $\log|E| = \Theta(\log|V|)$

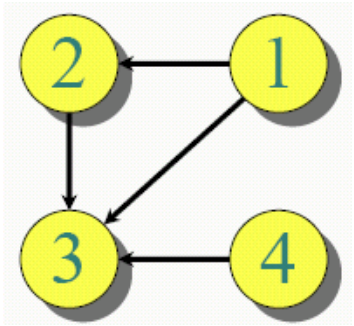
图表示

❖ 邻接矩阵法

- 给定 $G=(V, E)$，$|V| = n$，用矩阵$A_{n\times n}$表示图

$$A[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

- 适用于有向图和无向图



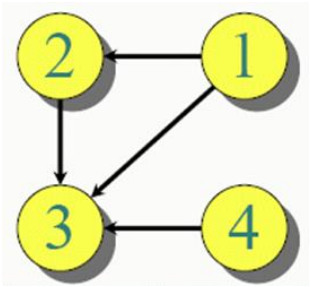
A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

$\Theta(V^2)$ storage
$\Rightarrow$ ***dense*** representation.

图表示

❖ 邻接链表法

- 给定 $G=(V, E)$，$|V| = n$，用链表$Adj[v]_n$表示



Adj [1] = {2, 3}
Adj [2] = {3}
Adj [3] = {}
Adj [4] = {3}

- 适用于有向图和无向图
  - 无向图： $|Adj[v]|$=degree(v)
  - 有向图： $|Adj[v]|$=out-degree(v)
- 握手定理
  - 对无向图： $\sum_{v \in V} = 2\,|E|$

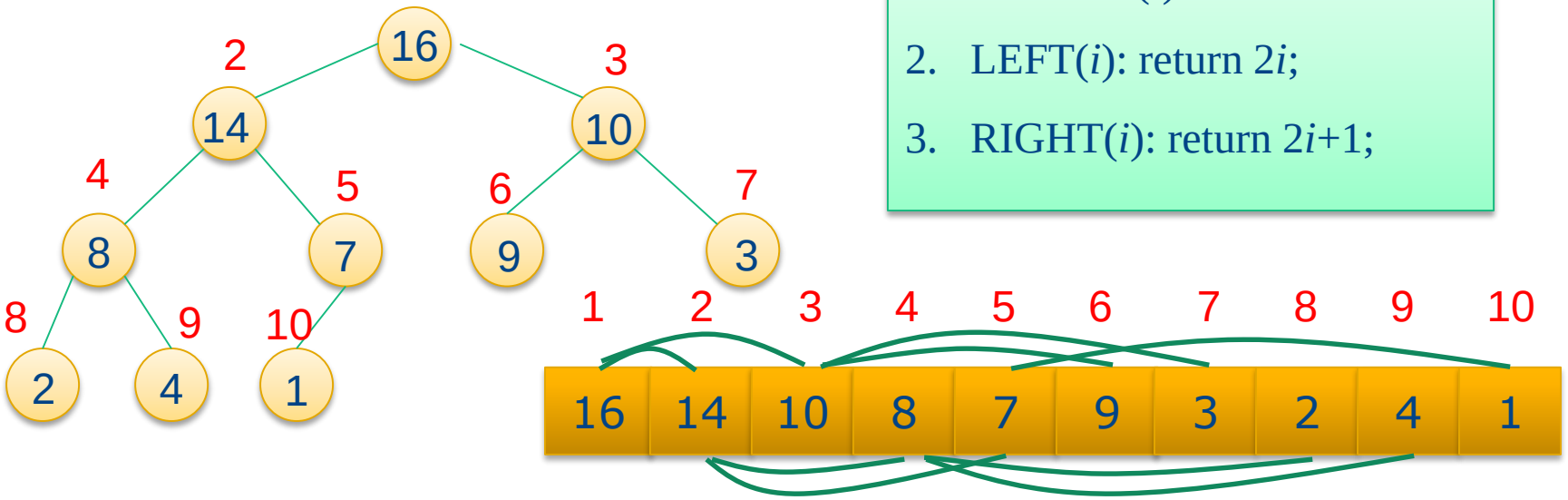
使用 $\Theta(V + E)$ 的存储 — a sparse representation.

(二叉)堆

二叉堆

- 二叉堆是一个数组，表达的是一种数据结构
  - 不是内存分配，也不是垃圾回收器
  - 可以近似的看作为一个完全二叉树
  - A.*length*表示数组元素的个数，A.*heap-size*表示堆元素的个数

1. PARENT(*i*): return $\lfloor i/2 \rfloor$
2. LEFT(*i*): return $2i$;
3. RIGHT(*i*): return $2i+1$;



二叉堆

- 二叉堆性质
  - 最大堆：$A[PARENT(i)] \geq A[i]$ —— 堆排序中使用的
  - 最小堆： $A[PARENT(i)] \leq A[i]$
- 堆上的算法
  - 堆的高度为 $\Theta(\log n)$
  - 维护堆的性质 MAX-HEAPIFY: $O(\log n)$
  - 建堆 BUILD-MAX-HEAP: $O(n)$
  - 原址排序 HEAPSORT：$O(n\log n)$
  - 实现一个优先级队列：$O(\log n)$
    - MAX-HEAP-INSERT、HEAP-EXTRACT-MAX、HEAP-INCREASE-KEY和HEAP-MAXIMUM

二叉堆：MAX-HEAPIFY

- 维护堆的性质 MAX-HEAPIFY
  - 输入A和i，假定当前根节点的LEFT(i)和RIGHT(i)已经都是二叉堆
- IDEA
  - 通过让A[i]的值在最大堆中逐级下降

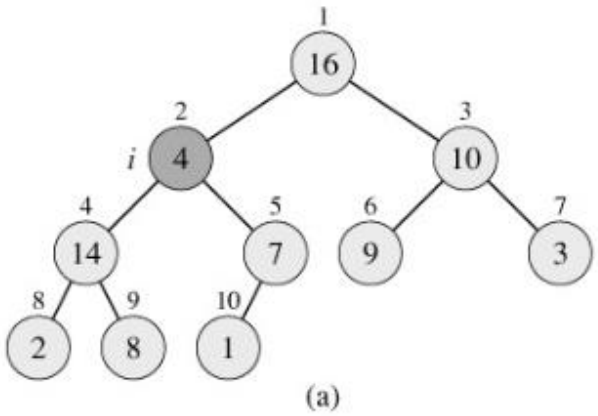
```
MAX-HEAPIFY(A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l≤heap-size[A] and A[l]>A[i]
4     then largest ← l
5     else largest ← i
6  if r≤heap-size[A] and A[r]>A[largest]
7     then largest ← r
8  if largest≠i
9     then exchange A[i]↔A[largest]
10         MAX-HEAPIFY(A, largest)
```

$$T(n) \le T\left(\frac{2n}{3}\right) + \Theta(1)$$

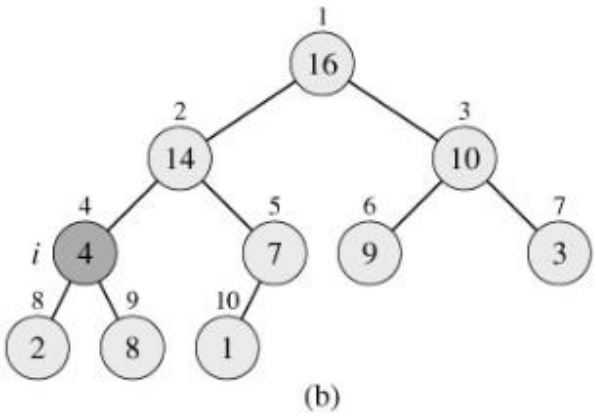
$$T(n) = O(h) = O(\log n)$$

二叉堆：MAX-HEAPIFY

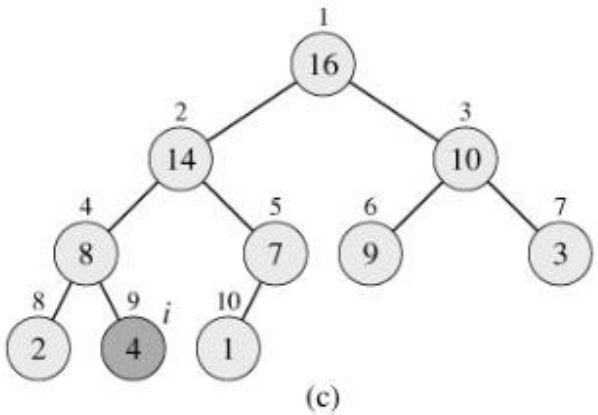
- 维护堆的性质 MAX-HEAPIFY
  - 通过让A[*i*]的值在最大堆中逐级下降



(a)



(b)



(c)

二叉堆：BUILD-MAX-HEAP

- 建堆 BUILD-MAX-HEAP
  - 将一个数组A[1:n]变成一个最大堆
  - 子数组A[($\lfloor n/2 \rfloor$+1: n)]中的元素都是树的叶结点
- 算法
  - 结点 i+1, i+2, …, n 都是一个最大堆的根
  - 在任意高度h上，之多有$\lceil n/2^{h+1} \rceil$个结点

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3        do MAX-HEAPIFY(A, i)
```

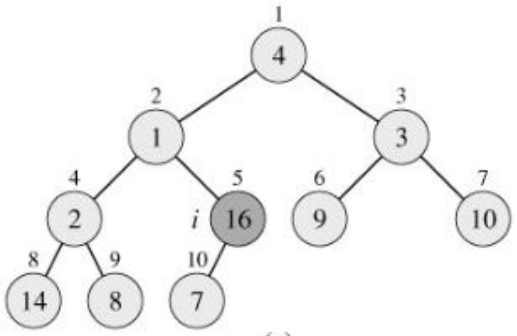
$$T(n) = \sum_{h=0}^{\lceil \log n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

$$= O\left(n \sum_{h=0}^{\lceil \log n \rceil} \frac{h}{2^h} \right)$$

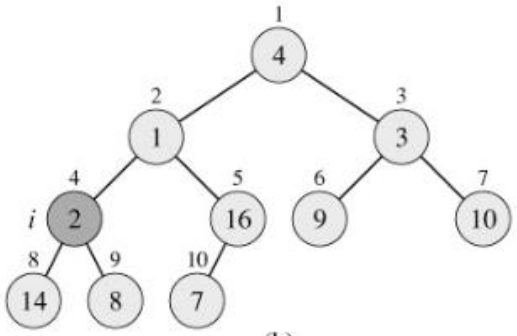
$$\leq O\left(n \sum_{h=0}^{} \frac{h}{2^h} \right) = O(n)$$

二叉堆：BUILD-MAX-HEAP

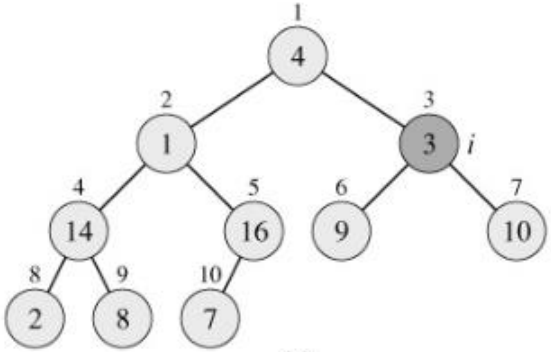
A	4	1	3	2	16	9	10	14	8	7



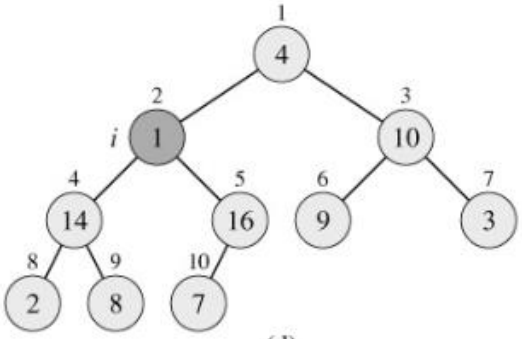
(a)



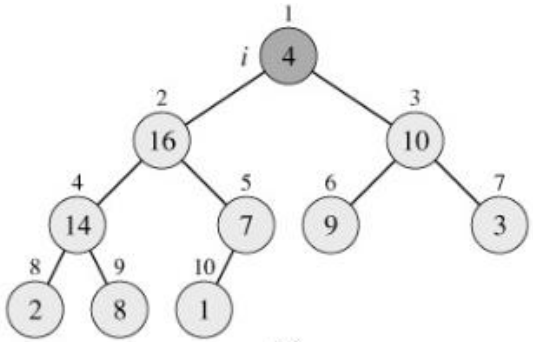
(b)



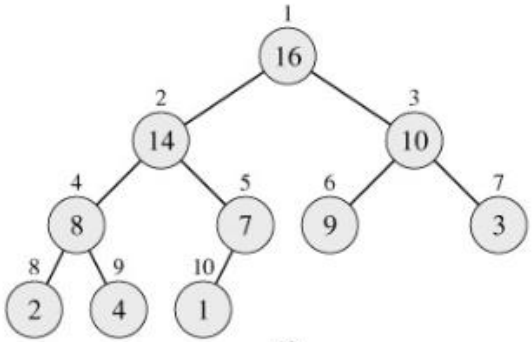
(c)



(d)



(e)



(f)

二叉堆：HEAPSORT

- 原址排序 HEAPSORT
  - 初始时将数组A[1:*n*]建成一个最大堆
  - 互换A[1]和A[*n*]，然后迭代在A[1:n-1]上维护最大堆性质

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2
3       do exchange A[1]↔A[i]
4          heap-size[A] ← heap-size[A]−1
5          MAX-HEAPIFY(A, 1)
```

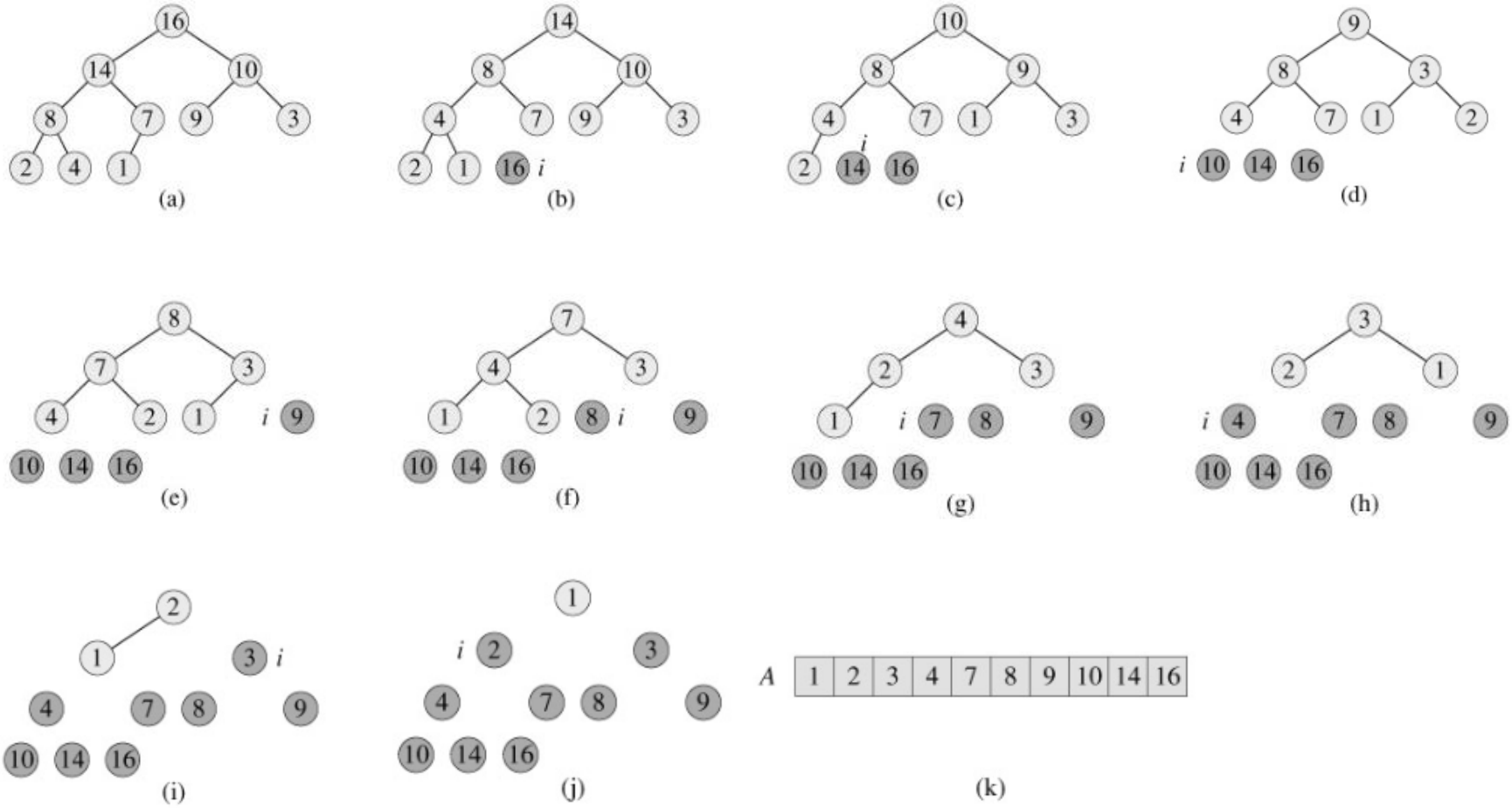
复杂度分析：

每次调用BUILD-MAX-HEAP的复杂度为O(*n*)，MAX-HEAPIFY的复杂度为O(log*n*)

因而总复杂度为：

O(*n*)+(*n*-1)O(log*n*)=O(*n*log*n*)

二叉堆：HEAPSORT



二叉堆：实现优先级队列

- 优先级队列 (priority queue)
  - 用来维护有一组元素构成的集合，每个元素有一个key
- 支持操作 (以最大优先级队列为例)
  - INSERT(S, x)
    - 把元素x插入集合S中，及S←S ∪ {x}
  - MAXIMUM(S)
    - 返回S中具有最大关键字的元素
  - EXTRACT-MAX(S)
    - 去掉并返回S中具有最大关键字的元素
  - INCREASE-KEY(S, x, k)
    - 将元素x的关键字的值增加到k，这里k大于等于x的原关键字

二叉堆：实现优先级队列

❖ 操作实现

```
HEAP-MAXIMUM(A)
1  return A[1]
```

$O(1)$

```
HEAP-EXTRACT-MAX(A)
1  if heap-size[A]<1
2     then error "heap underflow"
3  max ← A[1]
4  A[1] ← A[heap-size[A]]
5  heap-size[A] ← heap-size[A]−1
6  MAX-HEAPIFY(A, 1)
7  return max
```

$O(\log n)$

```
HEAP-INCREASE-KEY(A, i, key)
1  if key<A[i]
2     then error "new key is smaller than current key"
3  A[i] ← key
4  while i>1 and A[PARENT(i)]<A[i]
5        do exchange A[i]↔A[PARENT(i)]
6           i ← PARENT(i)
```

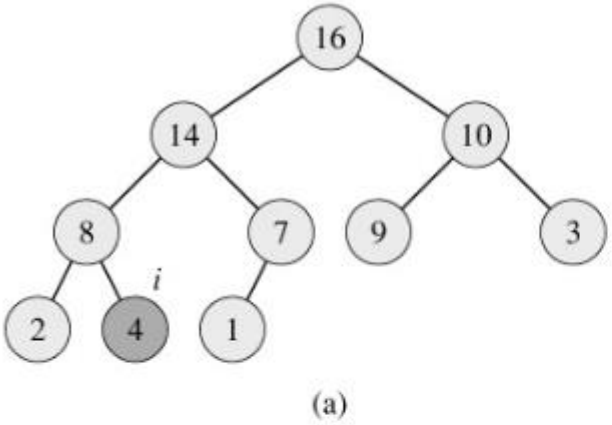
$O(\log n)$

```
MAX-HEAP-INSERT(A, key)
1  heap-size[A] ← heap-size[A]+1
2  A[heap-size[A]] ← −∞
3  HEAP-INCREASE-KEY(A, heap-size[A], key)
```

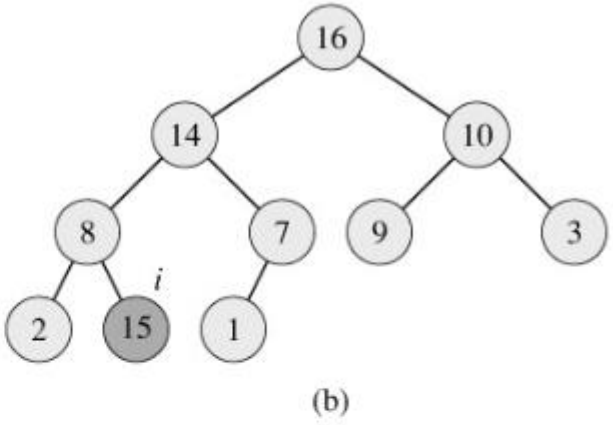
$O(\log n)$

二叉堆：实现优先级队列

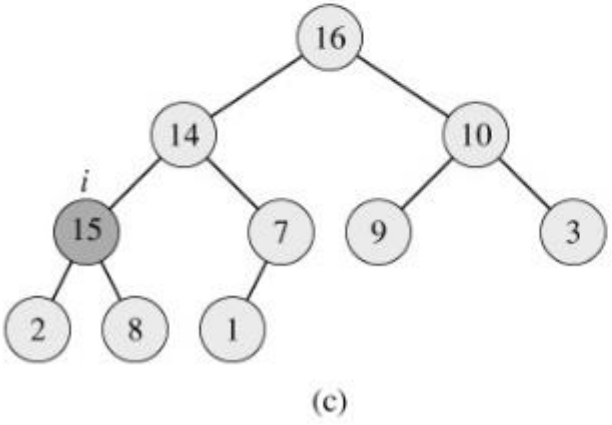
- HEAP-INCREASE-KEY



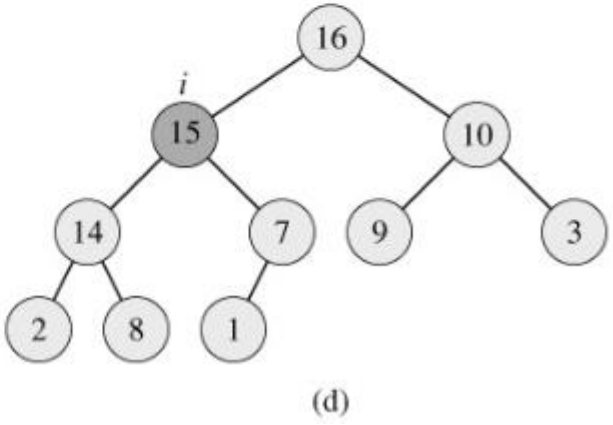
(a)



(b)



(c)

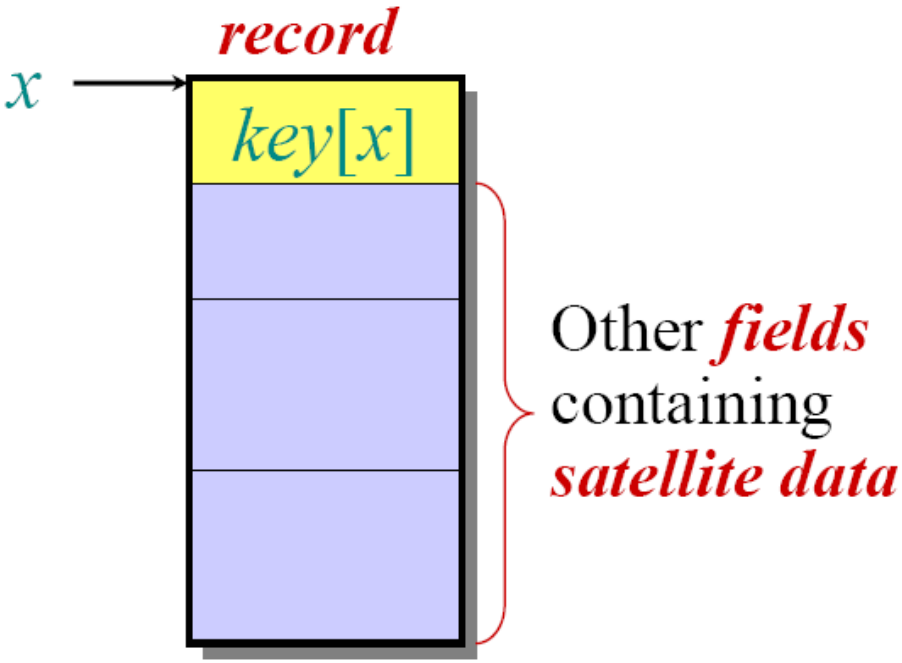


(d)

散列表 hash table

符号表问题 (Symbol-table)

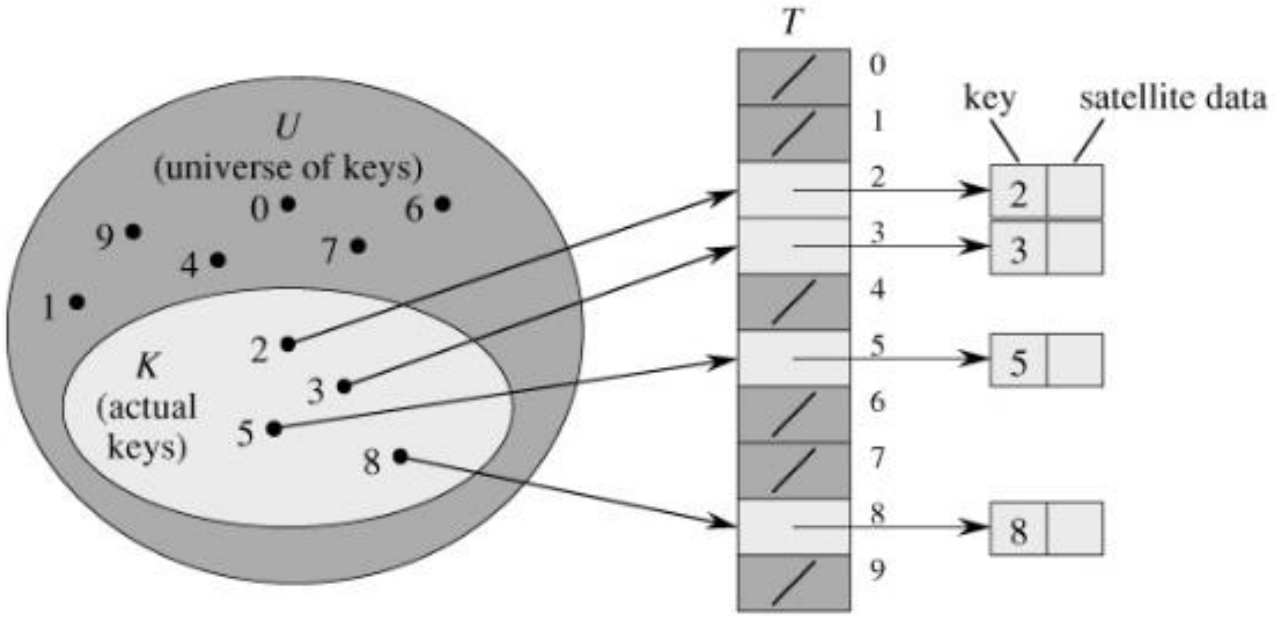
- 符号表 T 总共有 n 个记录
- T 上的操作
  - INSERT(T, x)
  - DELETE(T, x)
  - SEARCH(T, k)



- 数据结构 T 如何进行组织才能比较高效 (O(1))?

直接寻址表

- 关键字的全域U比较小时使用
  - U={0,1,…,*m*-1}
  - 用一个数组*T*[0:*m*-1]进行表示
  - 若关键字*k*没有元素，则*T*[*k*]=NIL



直接寻址表

- 基本操作
  - → Θ(1)
- 问题
  - 当关键字的范围变的很大时如何处理？
  - 如果关键字是字符串而不是整数时如何处理？

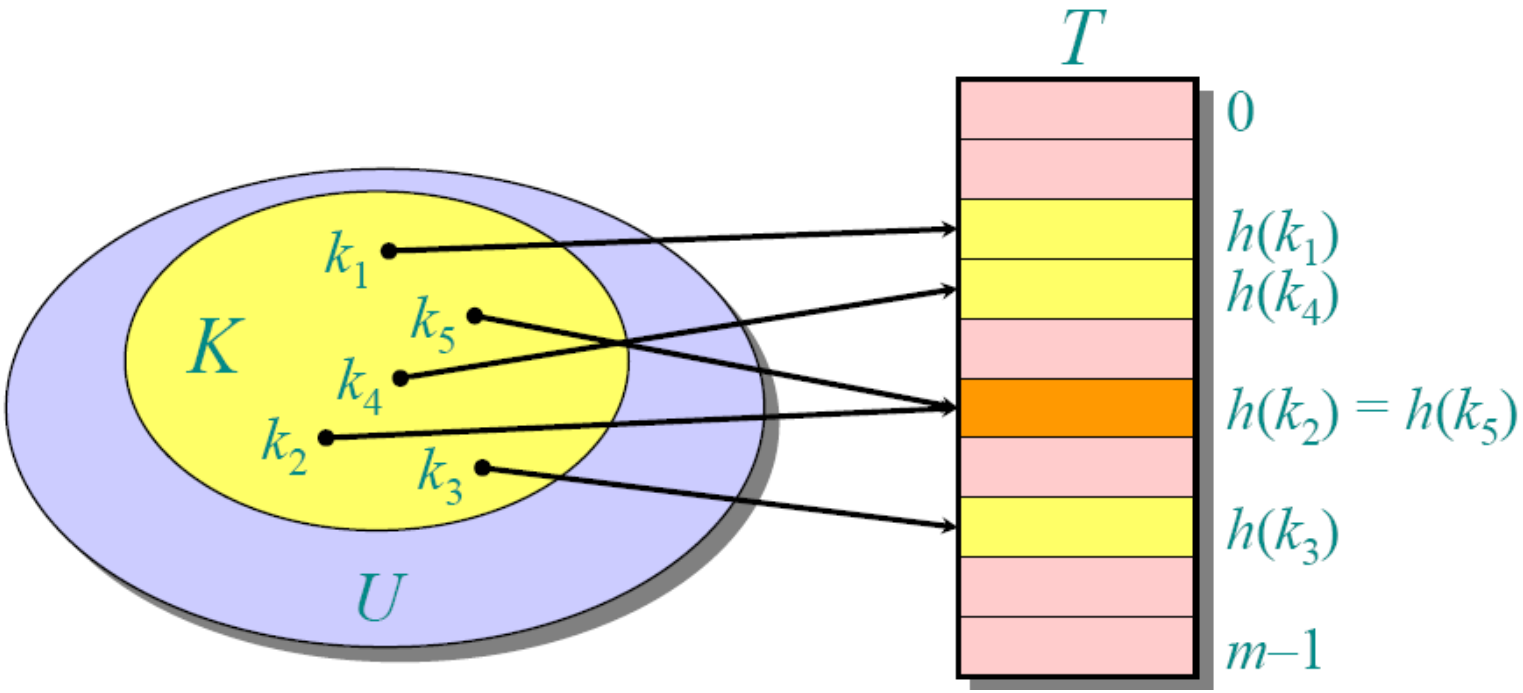
```
DIRECT-ADDRESS-SEARCH(T, k)
    return T[k]

DIRECT-ADDRESS-INSERT(T, x)
    T[key[x]] ← x

DIRECT-ADDRESS-DELETE(T, x)
    T[key[x]] ← NIL
```

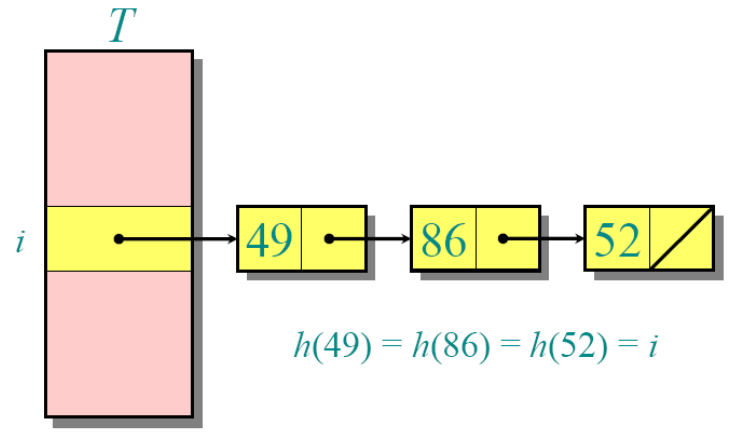
Hash函数

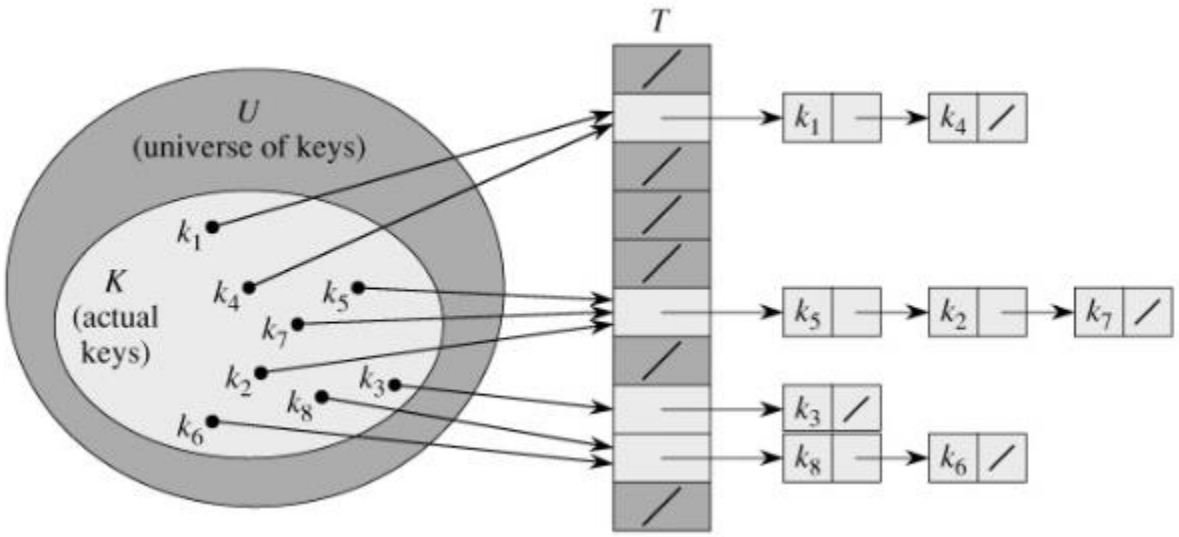
- 解决思路
  - 用一个hash函数 h 将关键字全域U映射到{0,1,…,m-1}
  - 当两个关键字映射到同一个槽中时，会发生冲突



链接法解决冲突

- IDEA
  - 在同一槽上插入一个链表
  - 为操作方便，可设计为双向链表





链接法分析

- 散列方法的平均性能依赖于所选取的hash函数
  - 此处假定为简单均匀散列 (simple uniform hashing)
    - 每个key值等可能地被散列到任一槽中，且它们是相互独立的
  - 让n表示键值的个数，m为槽数，定义T的装载因子

    $\alpha = n/m$ 为每个槽位的平均键值数
- 搜索性能
  - 在简单均匀散列条件下，链接法中搜索一个给定键值元素的期望时间为期望时间为 $\Theta(1+\alpha)$
    - 包括成功和不成功的情况 (具体证明参考 CLRS)
    - 如果$\alpha = O(1)$ or $n = O(m)$，则期望时间为 $\Theta(1)$

选择Hash函数

- 简单均匀散列条件很难满足
  - 在实际应用中，常常采用启发式方法构造散列函数
- 好的hash函数的特点
  - 应该尽量均匀地将键值分配到槽中
  - 应该独立于键值分布的可能存在的模式
  - 在某些应用中，可能会要求比较简单均匀散列更强的性质，如：
    - 相似的关键字具有截然不同的散列值
- 散列方法？

除法散列法

- 假设所有的键值都是整数
  - $h(k) = k \bmod m$
  - 非常高效
- 要避免m选择某些值
  - m不应该为2的幂次

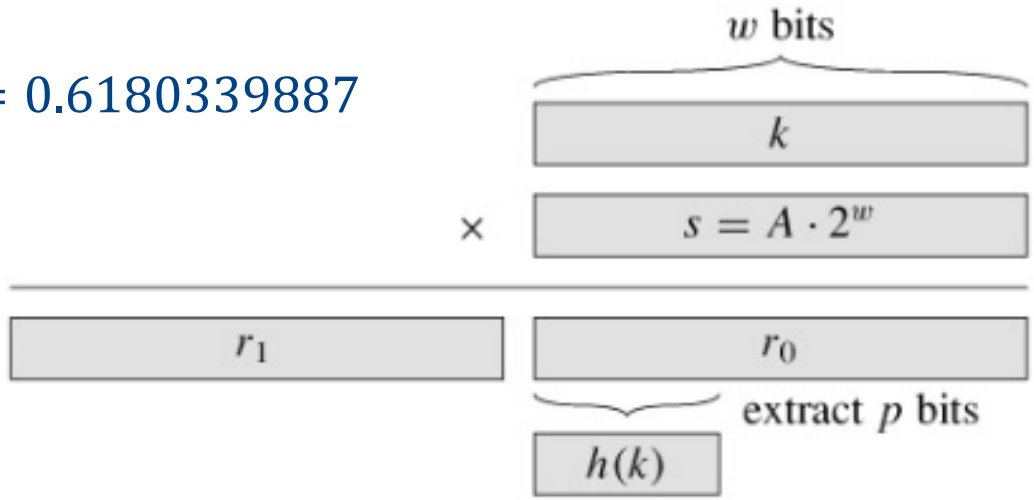
    If $k = 1011000111011010_2$ and $r = 6$, then $h(k) = 011010_2$.

    $h(k)$
  - 用一个不太接近2的幂次的素数通常是一个好的选择
    - 2000个字符串，期望平均查找三次 → m=701
    - 即：$h(k) = k \bmod 701$

乘法散列法

- 假设所有的键值都是整数
  - 常数因子A，$0<A<1$
  - $h(k)=\lfloor m(kA \bmod 1)\rfloor$
  - m一般选取为2的幂次（移位运算处理）
  - A对任何值都有效，但对某些值效果更好，如

$$A \approx \frac{\sqrt{5}-1}{2} = 0.6180339887$$



点乘散列法

❖ 随机化策略

- 用m表示一个素数
- 将键值k分解为r+1个数字，每个都属于{0,1,,…,m-1}
  即：$k = \langle k_0, k_1, \cdots, k_{m-1} \rangle$ with $0 \leq k_i < m$
- $a = \langle a_0, a_1, \cdots, a_{m-1} \rangle$，其中$a_i$是从{0,1,…$m$-1}中随机选取的
- 定义：

$$h_a(k) = \sum_{i=0}^{r} a_i k_i \bmod m$$

  - 它在实际应用中很有效，但计算代价较高

用开放寻址法解决冲突

- 所有的元素都放在散列表中
  - 每个表项或包含一个元素，或为NIL
  - 好处是不存储指针节省空间，从而提供更多的槽，同时提高效率
- 操作
  - 插入时要系统地探查所有表项直到找到一个空表项为止
  - 这里的hash函数依赖于键值和探查号，即

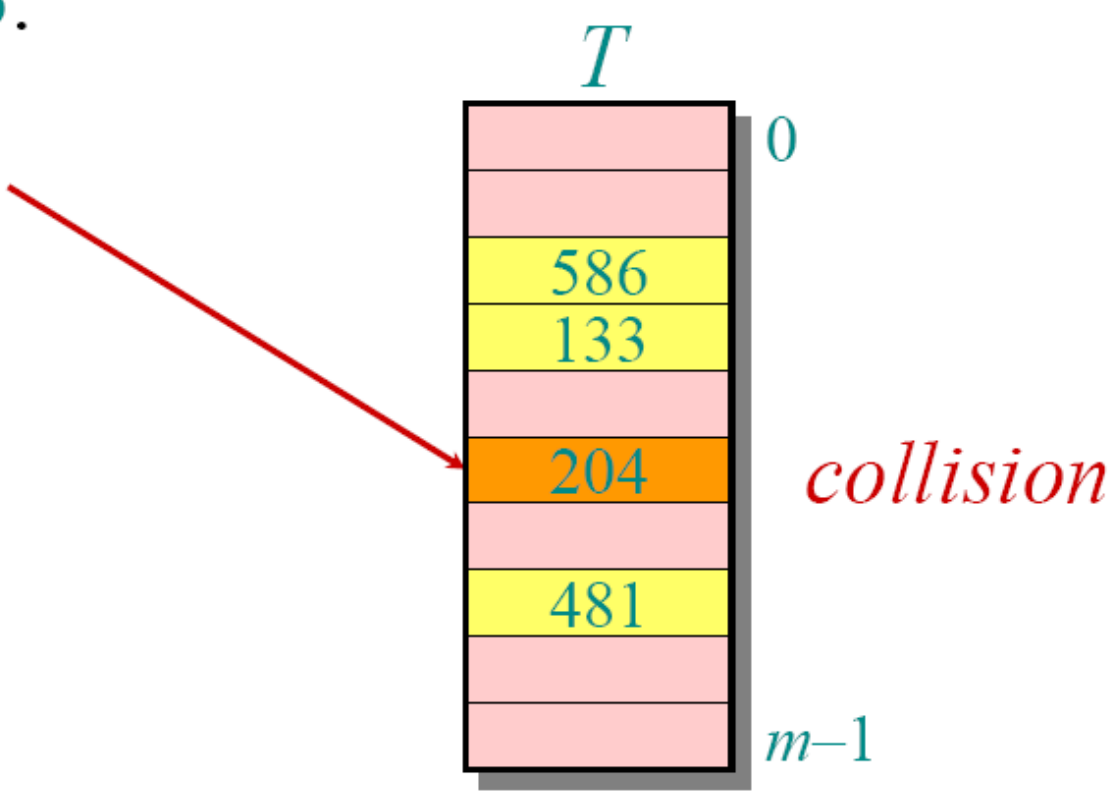
$$h: U \times \{0,1,\dots,m-1\} \to \{0,1,\dots,m-1\}$$

  - 探查序列$\langle h(k,0), h(k,1), \dots, h(k,m-1)\rangle$应当是$\{0,1,\dots,m-1\}$的一个排列
  - 当表项被填满时，删除变得比较困难

用开放寻址法示例

Insert key $k = 496$:

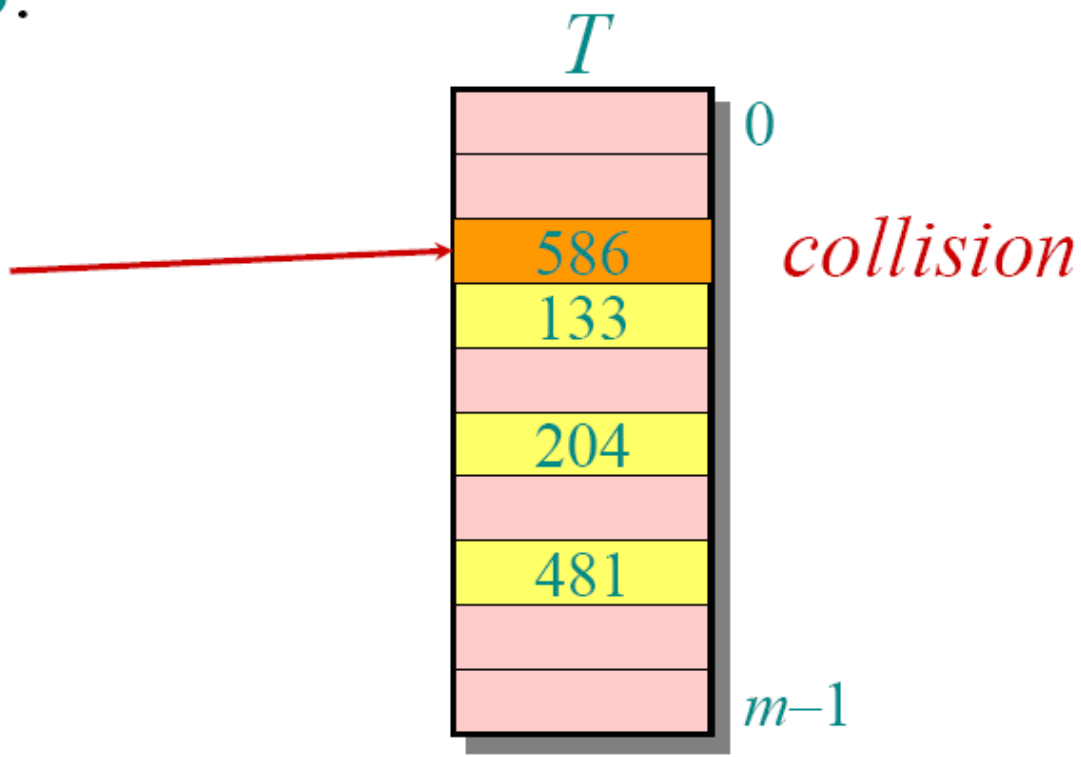
0. Probe $h(496,0)$



用开放寻址法示例

Insert key $k = 496$:

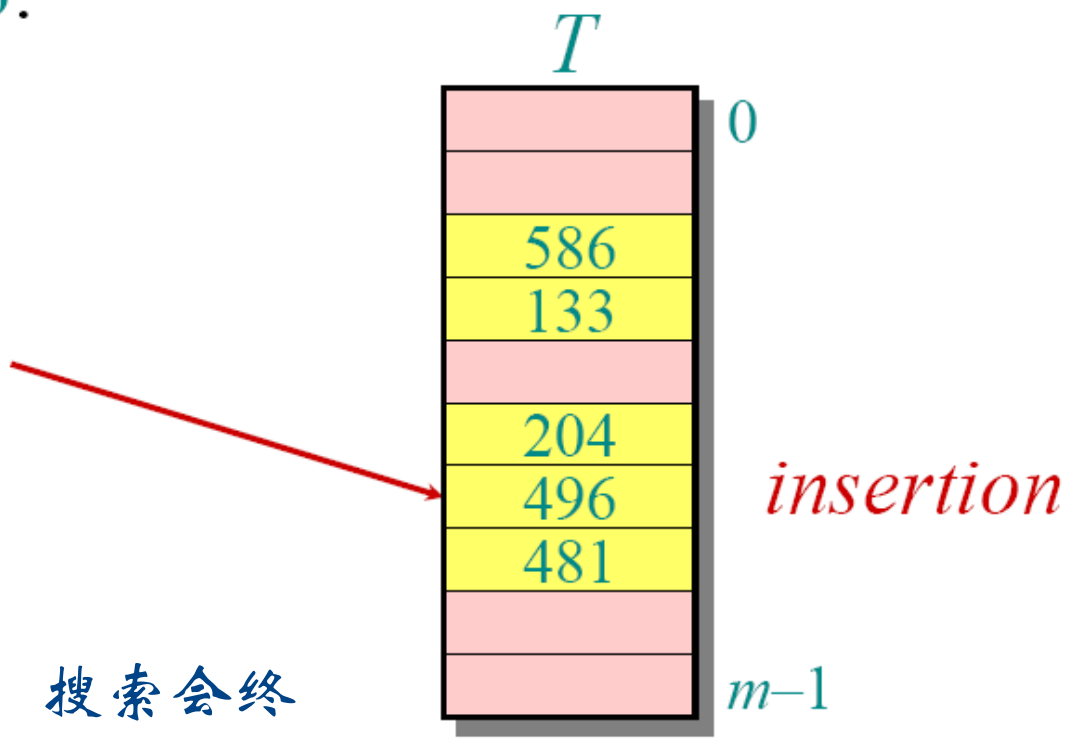
0. Probe $h(496,0)$
1. Probe $h(496,1)$



用开放寻址法示例

Insert key $k = 496$:

0. Probe $h(496,0)$
1. Probe $h(496,1)$
2. Probe $h(496,2)$



- 当发现一个空槽后，搜索会终止，并将当期值转入到该槽中

探查策略

- 线性探查
  - 给定一个普通的hash函数$h'(k)$，线性探查采用如下hash函数

$$h(k,i) = (h'(k) + i) \bmod m$$

  - 该方法容易实现，但存在一次群集(primary clustering)现象
    - 随着占用槽的增加，平均查找时间会不断增加
    - 连续被占用的槽就会变得越来越长

探查策略

❖ 双重散列

- 给定两个普通的hash函数$h_1(k)$和$h_2(k)$，双重散列采用如下hash函数

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m$$

- 该方法通常能够产生较好的结果
- 这里的$h_2(k)$必须对m是互素的
  - 取m为2的幂次，$h_2(k)$总是产生奇数
  - 取m为素数，$h_2(k)$总是产生小于m的正整数

开放寻址法的分析

- 这里假定是均匀散列的
  - 每个键值等可能地将$m!$种排列中的一个作为探查序列
- 分析定理
  - 给定装载因子$\alpha = \frac{n}{m} < 1$的开放寻址hash表，对一次不成功的查找，其期望探查次数至多为$1/(1-\alpha)$
  - 类似地，一次成功查找中的期望探查次数至多为$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$

开放寻址法的分析

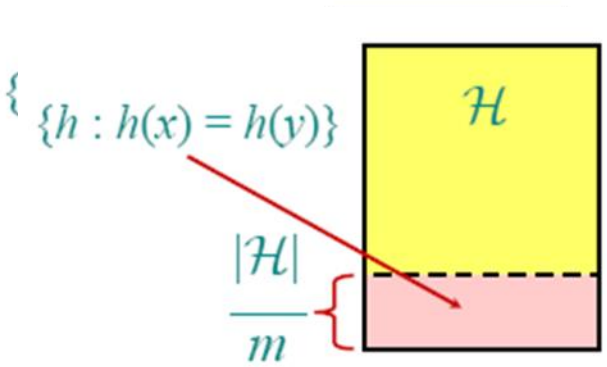
- 证明不成功情况下
  - 参考CLRS
- 定理的解释和应用
  - 如果α是一个常数，则访问一个开放寻址散列表需要的是常数时间
    - 如果hash表是半满的，则期望探查次数为 2
    - 如果hash表示是 90% 慢的，则期望探查次数为 10

全域散列法*

- 普通散列的一个问题
  - 对任一hash函数，存在一组键值它们的平均访问时间很长，如hash到同一个槽上
- IDEA
  - 独立于键值，随机地选择hash函数
    - 即使攻击者能够看到代码，也难以进行攻击，因为没有选择固定的hash函数
  - →全域散列

全域散列法*

- 设$\mathcal{H}$是一组有限散列函数
  - 每一个把U映射到$\{0,1,\ldots,m\text{-}1\}$
  - 如果对任意的$x,y \in U, x \neq y$，满足$h(x)=h(y)$的hash函数个数至多为$\frac{|\mathcal{H}|}{m}$，则称$\mathcal{H}$是全域的
    - 即x和y的冲突几率为$\frac{1}{m}$，如果随机地从$\mathcal{H}$中选取hash函数



全域散列法*

- 全域散列是好的
  - h选自一组全域散列函数，假定h用于hash任意n个键值到一个m个槽的表项中，则针对给定的任一键值x，有

$$E[\#\text{collisions with } x] < n/m.$$

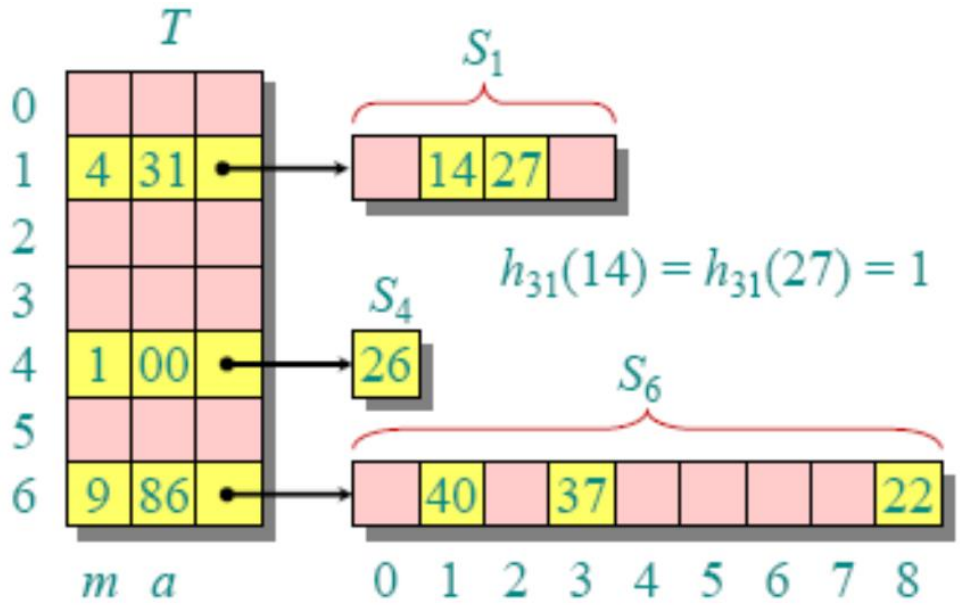
    - 证明：参考CLRS
- 构造全域散列函数
  - CLRS - pp.150 (Chinese version)
  - 随机内积hash函数簇：

$$|\mathcal{H}| = m^{r+1}$$

$$h_a(k) = \sum_{i=0}^{r} a_i k_i \bmod m .$$

完全散列*

- 针对给定的n个键值，构造一个静态散列表
  - 最坏情况下的搜索时间为$\Theta(1)$
- IDEA
  - 两级都用全域散列
  - 第二级不会发生冲突



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

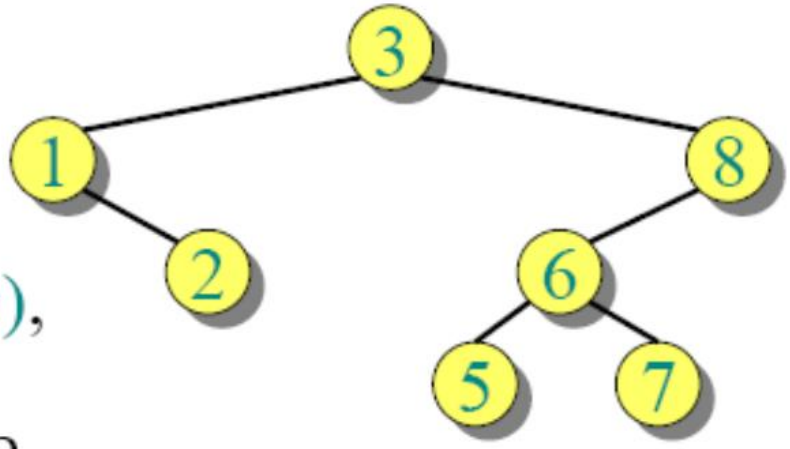
二叉搜索树

二叉搜索树

- Binary search tree – BST
  - 设x为BST任一节点，y是左子树任一节点，则$y.key \leq x.key$；是右子树任一节点，则$y.key \geq x.key$
  - 中序遍历能够顺序输出二叉搜索树中的所有元素

Example:

$A = [3\ 1\ 8\ 2\ 6\ 7\ 5]$



Tree-walk time = $O(n)$, but how long does it take to build the BST?

二叉搜索树

- 查询操作
  - 包括 SEARCH、MINIMUM、MAXIMUM、SUCCESSOR、PREDECESSOR
  - 基于树的递归定义，通常采用递归算法
  - 假定树的高度为 h → $O(h)$
- SEARCH:

```
TREE-SEARCH(x, k)
1. if x==NIL or k==x.key
2.     return x;
3. if k<x.key
4.     return TREE-SEARCH(x.left, k);
5. else
6.     return TREE-SEARCH(x.right, k);
```

```
ITERATIVE-TREE-SEARCH(x, k)
1. while x ≠ NIL and k ≠ x.key
2.     if k<x.key
3.         x=x.left;
4.     else
5.         x=x.right;
6. return x;
```

二叉搜索树

❖ 查询操作

TREE-MINIMUM(x)
1. while $x.left \neq$ NIL
2. $\quad x = x.left$;
3. return x;

TREE-MAXIMUM(x)
1. while $x.right \neq$ NIL
2. $\quad x = x.right$;
3. return x;

TREE-SUCCESSOR(x)
1. if $x.right \neq$ NIL
2. $\quad$ return TREE-MINIMUM($x.right$);
3. $y = x.p$;
4. while $y \neq$ NIL and $x==y.right$
5. $\quad x=y$;
6. $\quad y=y.p$;
7. return y;

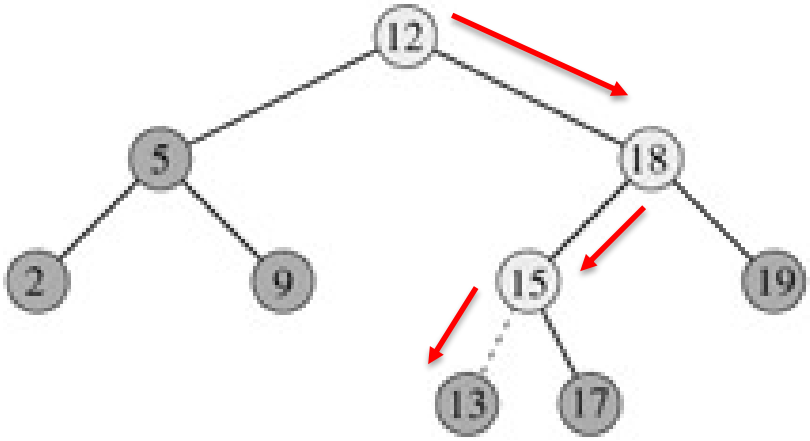
TREE-PREDECESSOR(x)
1. if $x.left \neq$ NIL
2. $\quad$ return TREE-MAXIMUM($x.left$);
3. $y = x.p$;
4. while $y \neq$ NIL and $x==y.left$
5. $\quad x=y$;
6. $\quad y=y.p$;
7. return y;

二叉搜索树

❖ 插入操作

- 动态集合发生变化 → $O(h)$

```
TREE-INSERT(T, z)
1.   y = NIL;
2.   x = T.root;
3.   while x ≠ NIL
4.       y=x;
5.       if z.key < x.key
6.           x=x.left;
7.       else x=x.right;
8.   z.p = y;
9.   if y == NIL
10.      T.root =z;   //empty tree
11.  elseif z.key < y.key
12.      y.left =z;
13.  else y.right = z;
```



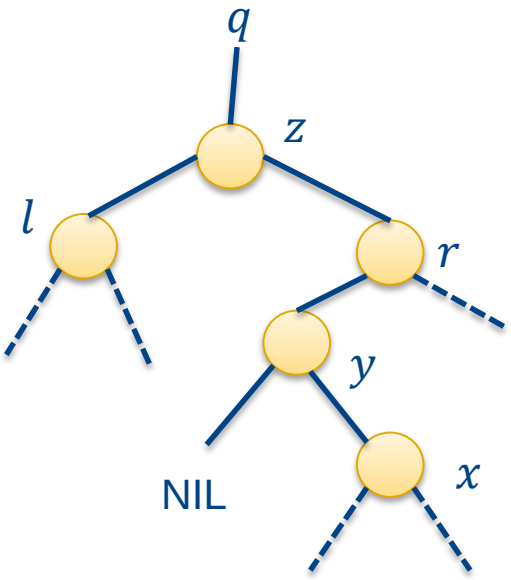
初始时：

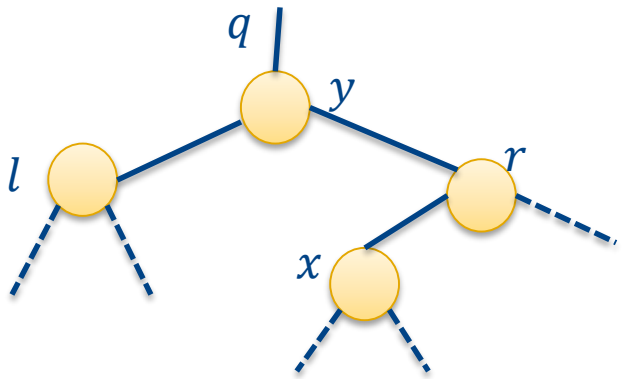
$z.key=v$, $z.left$=NIL, $z.right$ = NIL

二叉搜索树

❖ 删除操作

- 动态集合发生变化 → $O(h)$
- 需要处理情况
  - z 没有左孩子
  - z 仅有左孩子
  - z 有左右孩子，y 为其后继
    - y是z的右孩子
    - y在右子树中，但不是z的右孩子





二叉搜索树

❖ 删除操作

- 动态集合发生变化 → $O(h)$

```
TRANSPLANT(T, u, v)
1.  if u.p == NIL
2.      T.root =v;
3.  else if u=u.p.left
4.      u.p.left = v;
5.  else u.p.right =v;
6.  if v ≠ NIL
7.      v.p=u.p;
```

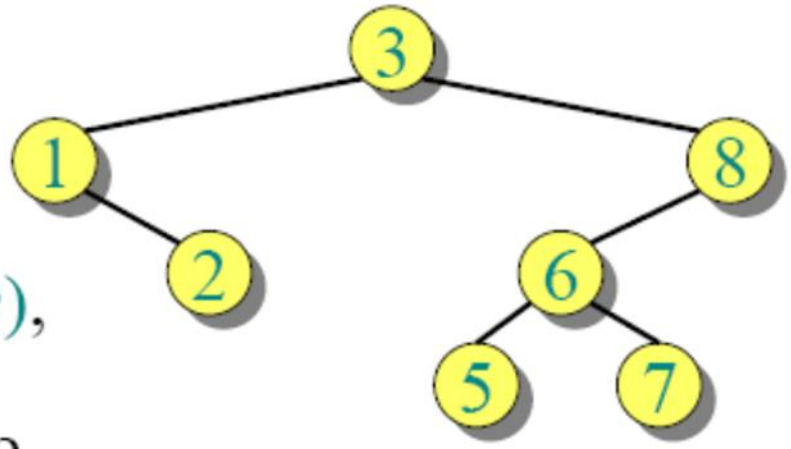
```
TREE-DELETE(T, z)
1.  if z.left == NIL
2.      TRANSPLANT(T, z, z.right);
3.  else if z.right == NIL
4.      TRANSPLANT(T, z, z.left)
5.  else y= TREE-MINIMUM(z.right)
6.      if y.p ≠ z
7.          TRANSPLANT(T, y, y.right)
8.          y.right = z.right;
9.          y.right.p=y;
10.     TRANSPLANT(T, z, y)
11.     y.left = z.left;
12.     y.left.p=y;
```

构建二叉搜索树

- 随机构建二叉搜索树
  - 创建一个空BST，按顺序“TREE-INSERT $(T, A[i])$”
  - n个关键字的$n!$个排列的每一个等可能地出现

Example:

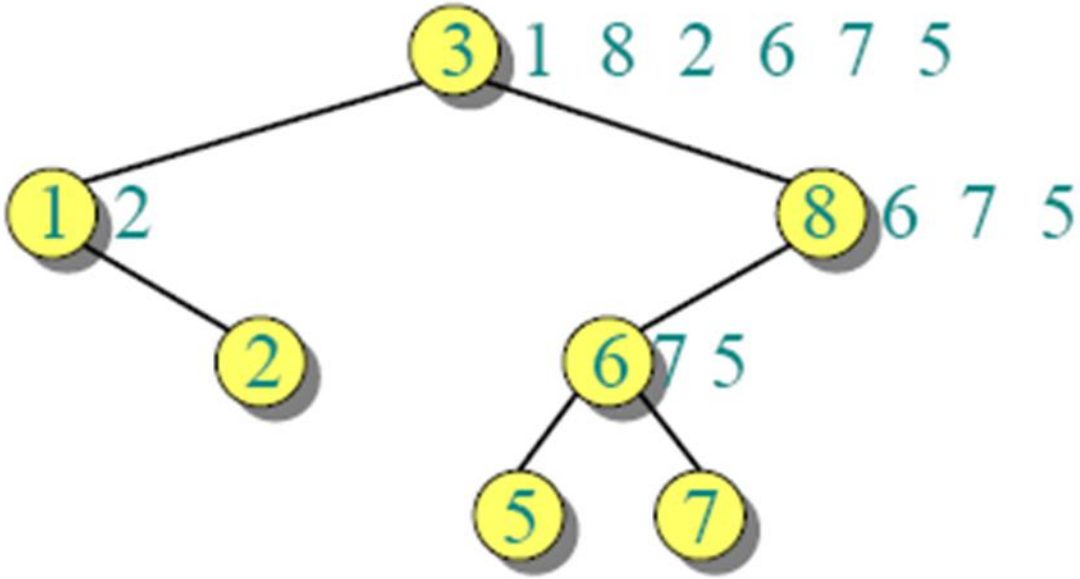
$A = [3\ 1\ 8\ 2\ 6\ 7\ 5]$



Tree-walk time $= O(n)$, but how long does it take to build the BST?

随机构建二叉搜索树

- 二叉搜索树执行与quicksort相似的比较，但顺序不同
  - 建立一个二叉树的期望时间与quicksort的运行时间是渐近相同的



节点深度

- 一棵n个不同关键字随机构建二叉搜索树的期望高度为$O(\log n)$
  - 与quicksourt对比分析的平均结点深度

$$= \frac{1}{n} E\left[\sum_{i=1}^{n} (\#\text{comparisons to insert node } i)\right]$$

$$= \frac{1}{n} O(n \lg n) \qquad \text{(quicksort analysis)}$$

$$= O(\lg n) \ .$$

  - 这不等价于树的期望高度为$O(\log n)$（实际上如此）
    - 需要额外的证明

节点深度分析

- 定义随机变量（总共n个结点）
  - X_n表示随机BST的高度，$Y_n = 2^{X_n}$为对应的指数高度
  - 如果树根的秩为 k，则

$$X_n = 1 + \max(X_{k-1}, X_{n-k})$$
$$Y_n = 2 \cdot \max(Y_{k-1}, Y_{n-k})$$

  - 定义指示随机变量 Z_{nk}如下：

$$Z_{nk} = \begin{cases} 1 & \text{if the root has rank } k \\ 0 & \text{otherwise} \end{cases}$$

    - 则有：

$$\Pr\{Z_{nk} = 1\} = \mathrm{E}[Z_{nk}] = 1/\mathrm{n}$$

节点深度分析

- $Y_n = \sum_{k=1}^{n} Z_{nk}(2 \cdot \max\{Y_{k-1}, Y_{n-k}\})$

→ $E[Y_n] = E[\sum_{k=1}^{n} Z_{nk}(2 \cdot \max\{Y_{k-1}, Y_{n-k}\})]$

$$= \sum_{k=1}^{n} E[Z_{nk}(2 \cdot \max\{Y_{k-1}, Y_{n-k}\})]$$

$$= 2\sum_{k=1}^{n} E[Z_{nk}]E[\max\{Y_{k-1}, Y_{n-k}\})]$$

$$\leq \frac{2}{n}\sum_{k=1}^{n} E[Y_{k-1} + Y_{n-k}]$$

$$= \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k]$$

Each term appears twice, and reindex.

=

节点深度分析

- $E[Y_n] \le \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k]$
  - 代入法可证明 $E[Y_n] \le cn^3$
  - $2^{E[X_n]} \le E[2^{X_n}] = E[Y_n] \le cn^3$

即：

$$E[X_n] \le 3\log n + O(1)$$

$$\begin{aligned} E[Y_n] &= \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k] \\ &\le \frac{4}{n}\sum_{k=0}^{n-1} ck^3 \\ &\le \frac{4c}{n}\int_0^n x^3 dx \\ &= \frac{4c}{n}\left(\frac{n^4}{4}\right) \\ &= cn^3. \end{aligned}$$

平衡二叉搜索树

- 什么是平衡二叉搜索树？
  - 属于二叉搜索树
  - 在动态改变中保证其树的高度为 $O(\log n)$

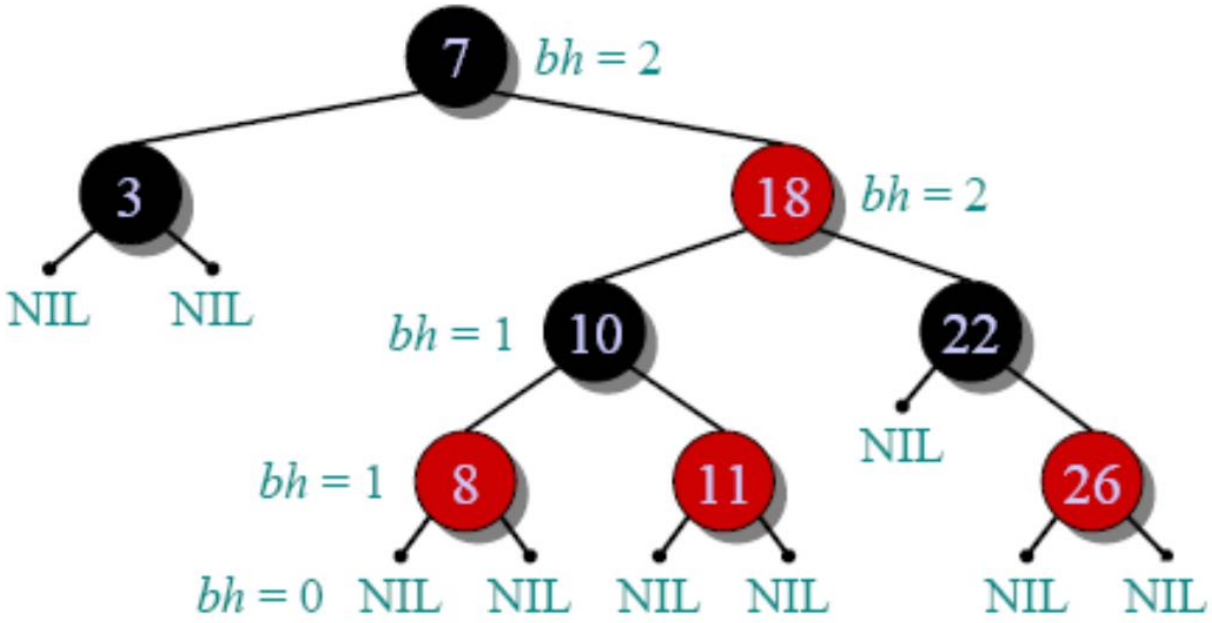
- 示例实现：
  - AVL trees
  - 2-3 trees
  - 2-3-4 trees
  - B-trees
  - Red-black trees

红黑树

- 每个结点增加额外的颜色属性：红或者黑
- 满足红黑性质：
  - 1. 每个结点或者是红色的，或者是黑色的
  - 2. 根节点是黑色的
  - 3. 每个叶节点(NIL)是黑色的
  - 4. 如果一个结点是红色的，则它的两个子结点都是黑色的
  - 5. 对每个结点 x，从该结点到其所有后代叶结点的简单路径上，均包含相同数目的黑色节点
    - = black-height(x)

红黑树

- 示例 (1,2,3,4,5)
  - 4. 如果一个结点是红色的，则它的两个子结点都是黑色的
  - 5. 对每个结点 x，从该结点到其所有后代叶结点的简单路径上，均包含相同数目的黑色节点



红黑树的高度

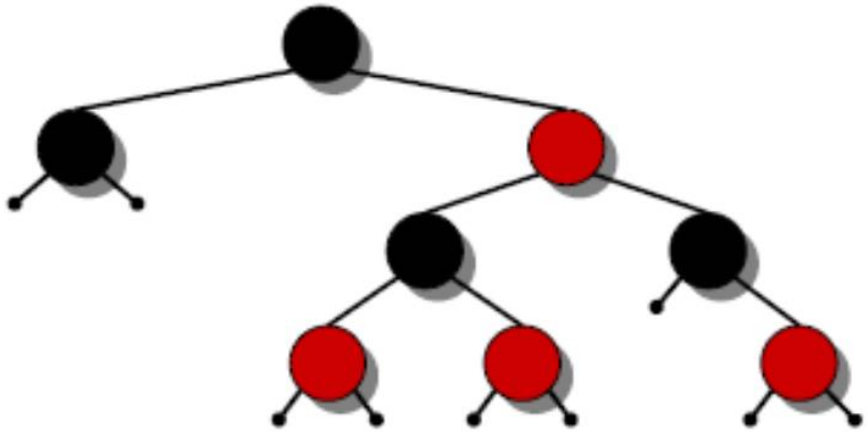
- 定理
  - 一棵具有n个键值的红黑树，其高度h满足：

$$h \leq 2\log(n+1)$$

  - 证明：参考CLRS

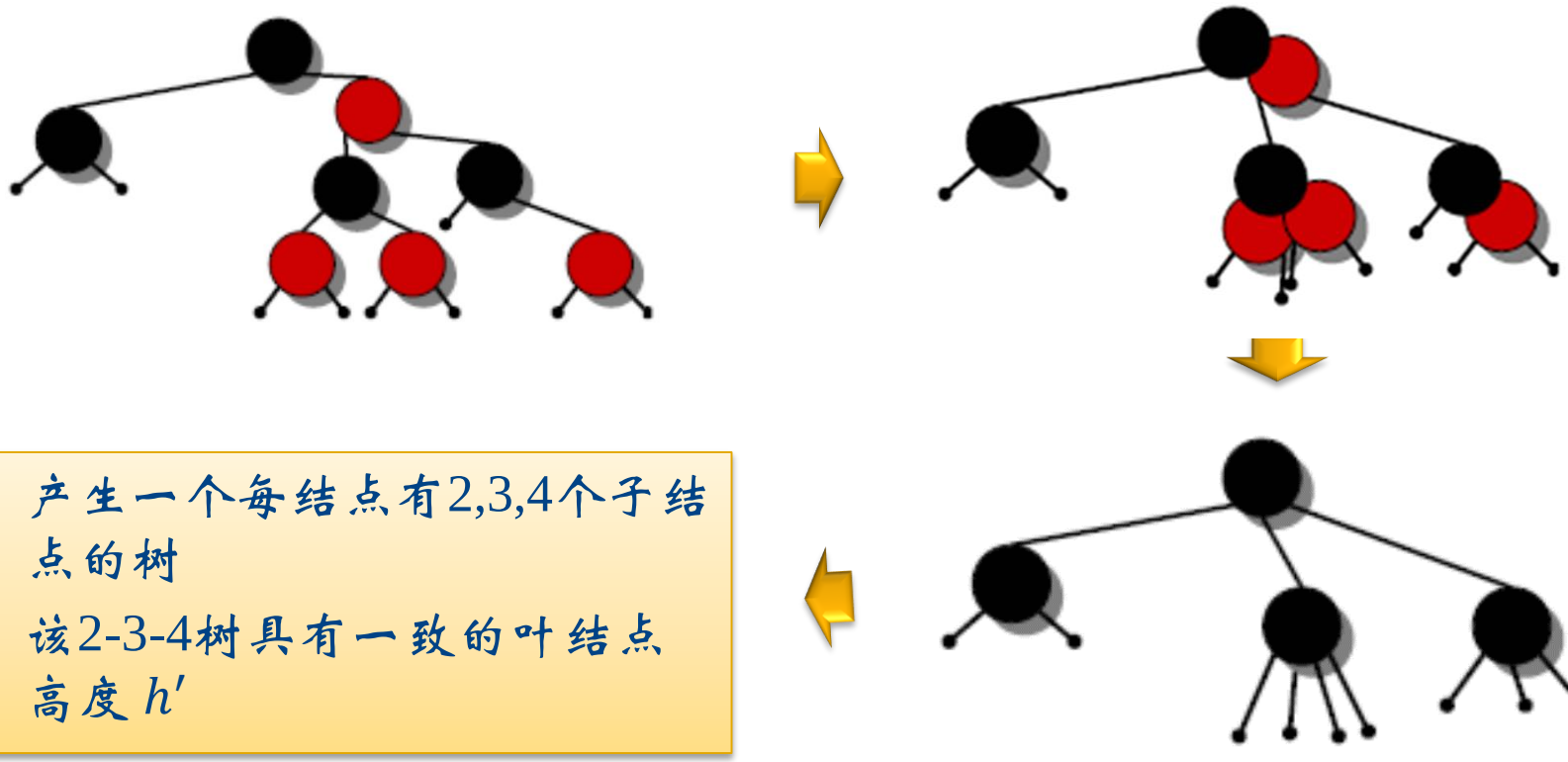
INTUITION:

- Merge red nodes into their black parents.



红黑树的高度

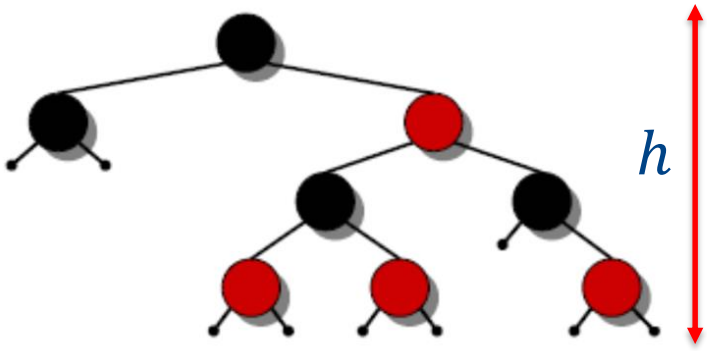
- INTUITION
  - 合并红结点到它们的黑色父结点中



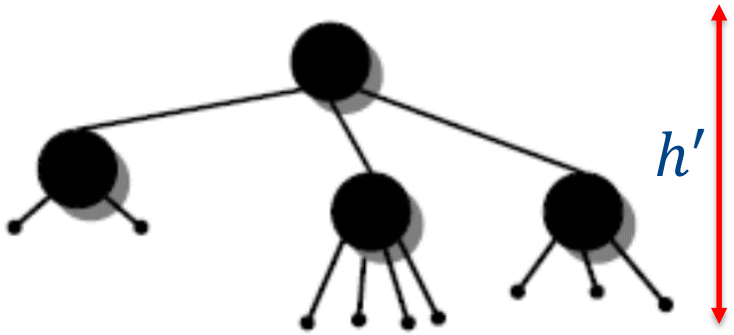
- 产生一个每结点有2,3,4个子结点的树
- 该2-3-4树具有一致的叶结点高度 h'

红黑树的高度

- INTUITION
  - 根据红黑性质，我们有
    $h' \geq h/2$
  - 每个树的叶结点个数都是 $n+1$



$$\Rightarrow n+1 \geq 2^{h'}$$
$$\Rightarrow \lg(n+1) \geq h' \geq h/2$$
$$\Rightarrow h \leq 2\lg(n+1). \quad \square$$



红黑树的操作

- 查询操作
  - 红黑树上SEARCH、MIN、MAX、SUCCESSOR、PREDECESSOR 的运行时间为 $O(\log n)$
- 修改操作
  - 操作 INSERT 和DELETE会引起红黑树的修改
    - 操作本身
    - 颜色修改
    - 需要重构树的连接也保持红黑性质（通过“rotation”操作）

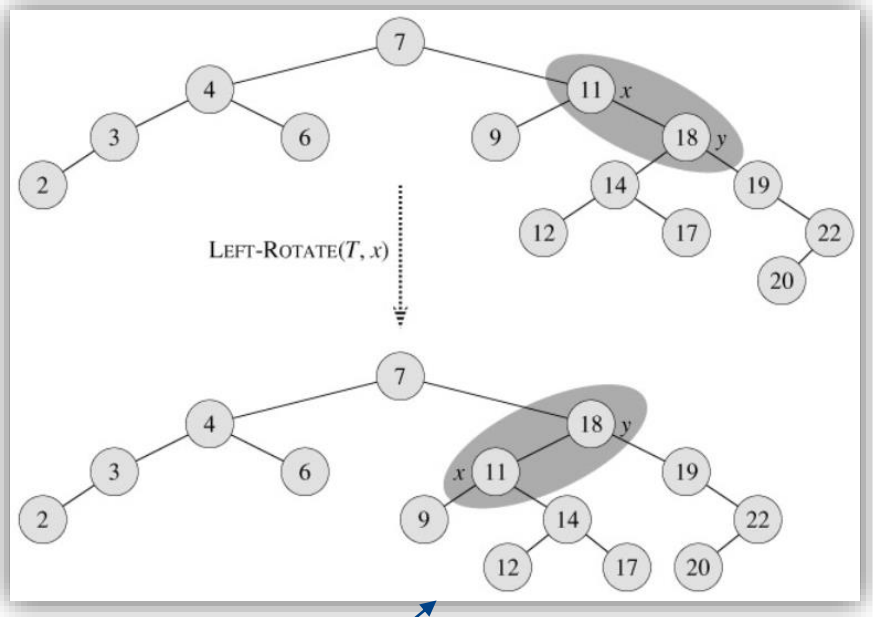
Rotation操作

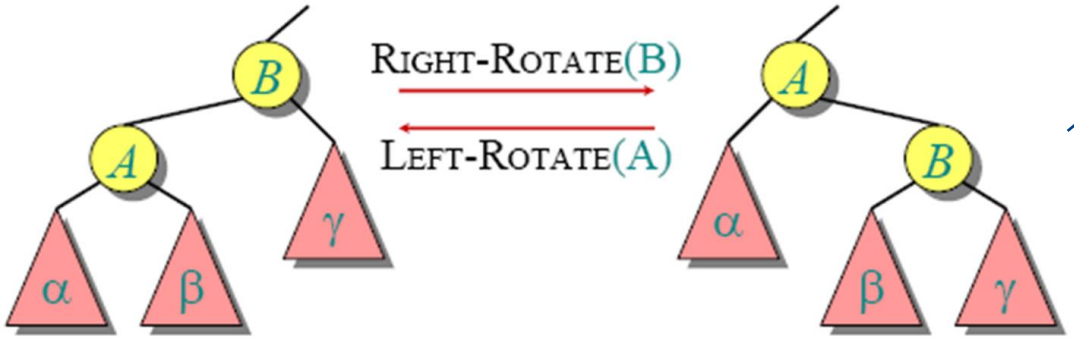
Rotations

- 保持中序的键值顺序不改变

  $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c.$

- 在$O(1)$时间内能够完成



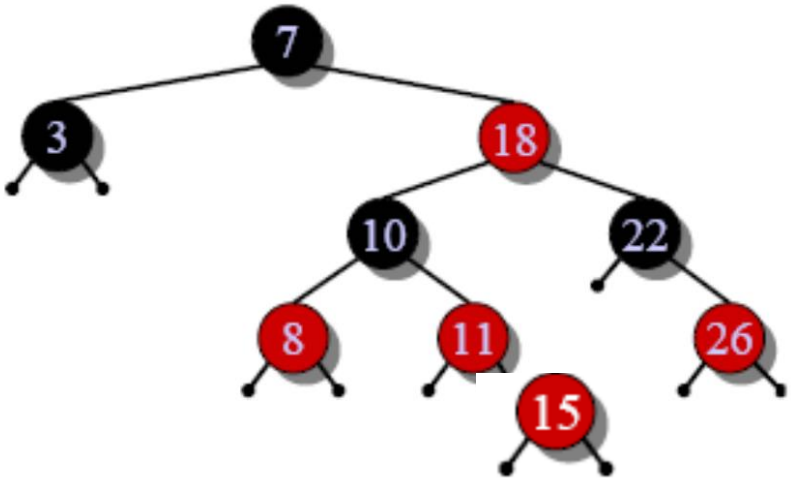


INSERT操作

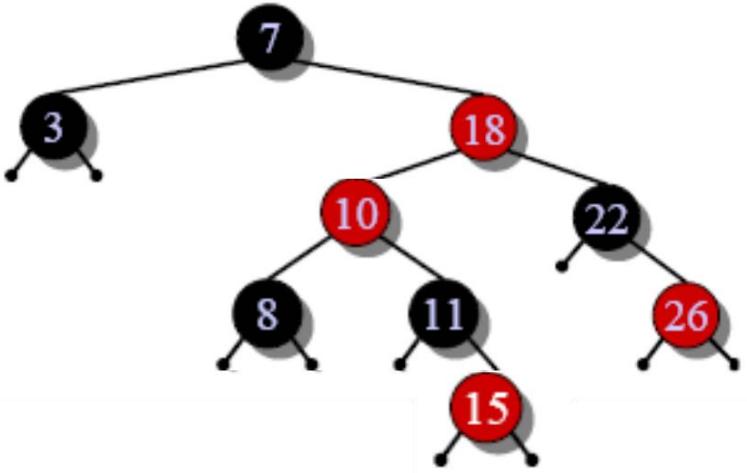
- IDEA
  - INSERT+FIXUP
  - 将待插入节点 x 着色为红色，像普通二叉搜索树那样插入新结点
  - 通过重新着色和rotation操作修复违反的红黑性质
- 可能违反的红黑性质
  - 性质2：根节点是黑色的
  - 性质4：红色节点的子结点是黑色的

INSERT举例

- 示例：
  - 插入新结点 $x = 15$

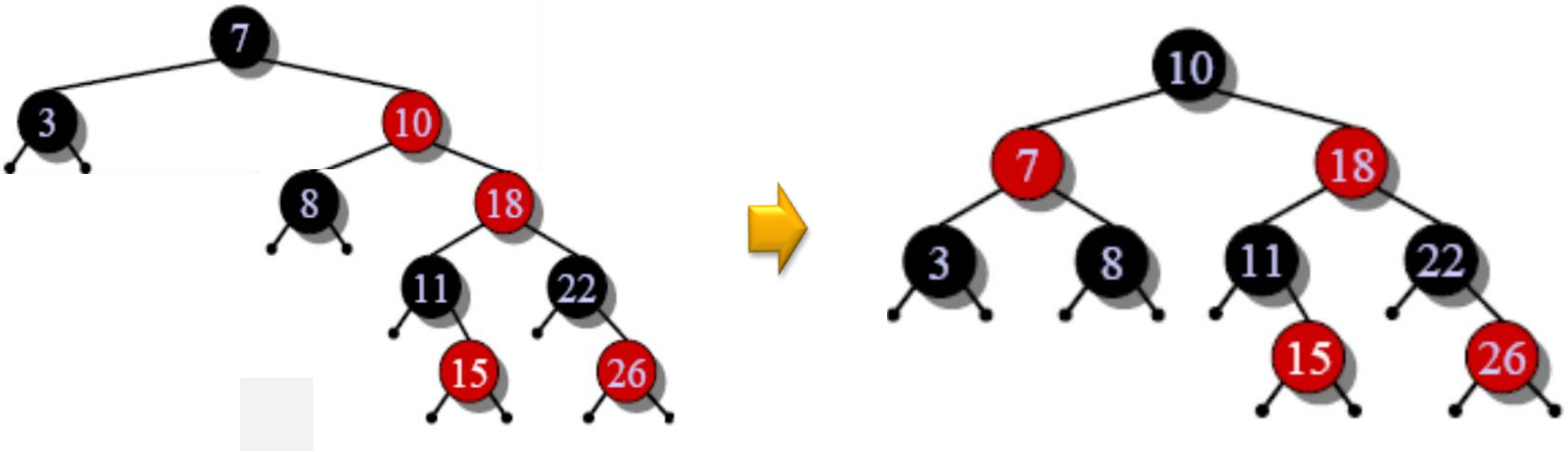


➢ 重着色



INSERT举例

- 示例：
  - 插入新结点 $x = 15$

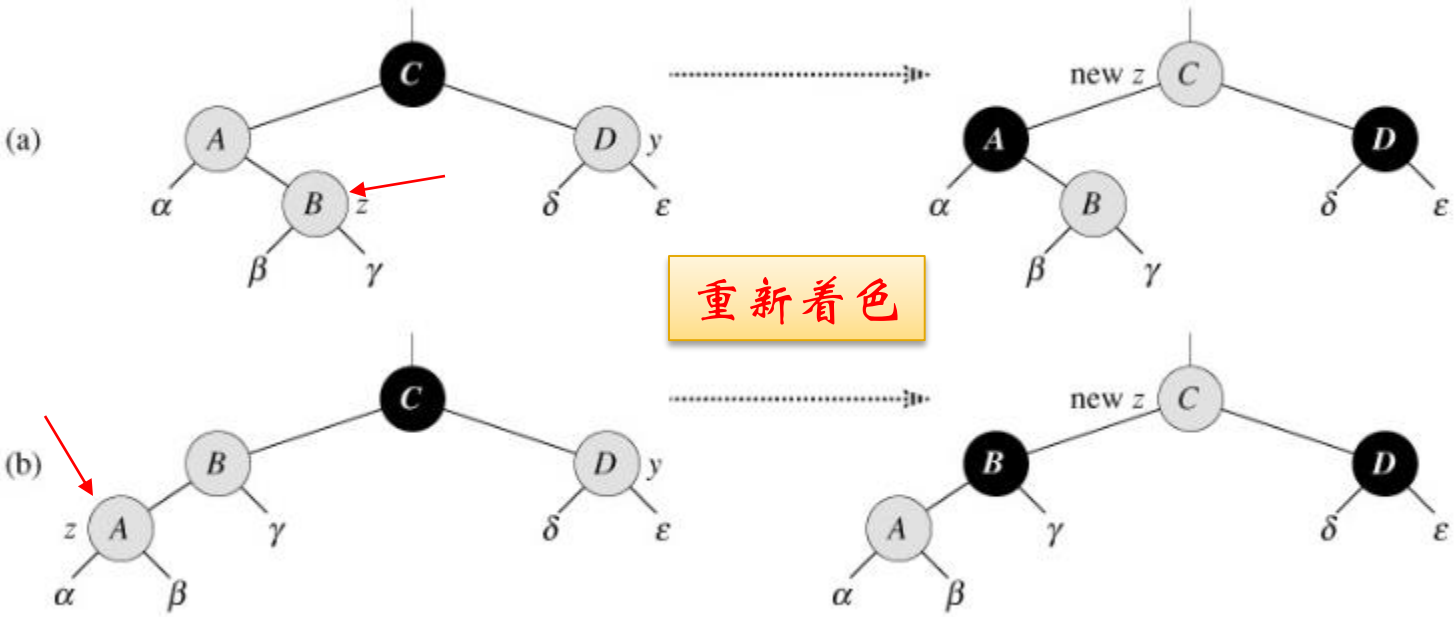


- RIGHT-ROTATE(18)
- LEFT-ROTATE(7)
- 重着色

FIXUP迭代执行

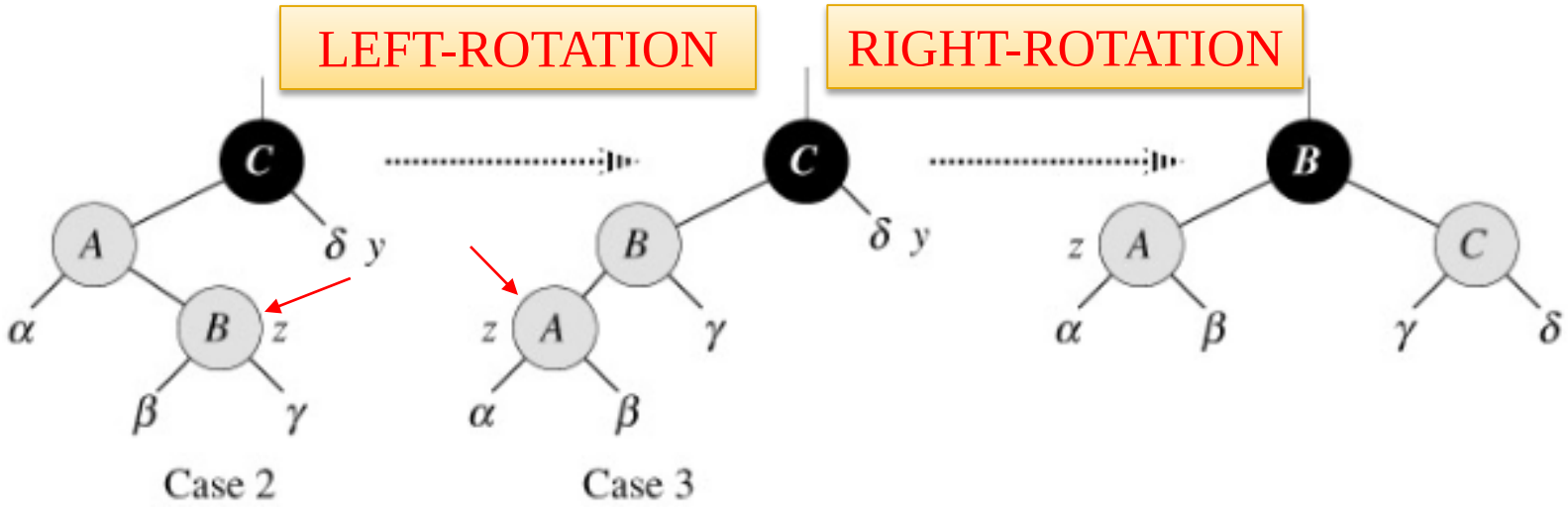
Case 1

- z 的叔结点 y 红色的，无论 z 是左孩子还是右孩子



FIXUP迭代执行

- 其他情况
  - Case 2：z 的叔结点 y 是黑色的，且 z 是右孩子
  - Case 3：z 的叔结点 y 是黑色的，且 z 是左孩子



INSERT分析

- INSERT整体性能
  - 普通插入 $O(\log n)$
  - 修复操作 $O(\log n)$
    - Case 1：重新着色，$O(1)$
    - Case 2 or case 3：执行1-2次rotations，$O(1)$
    - 执行次数 $O(\log n)$
- DELETE操作
  - 具有相似的渐进性能
  - 参考CLRS

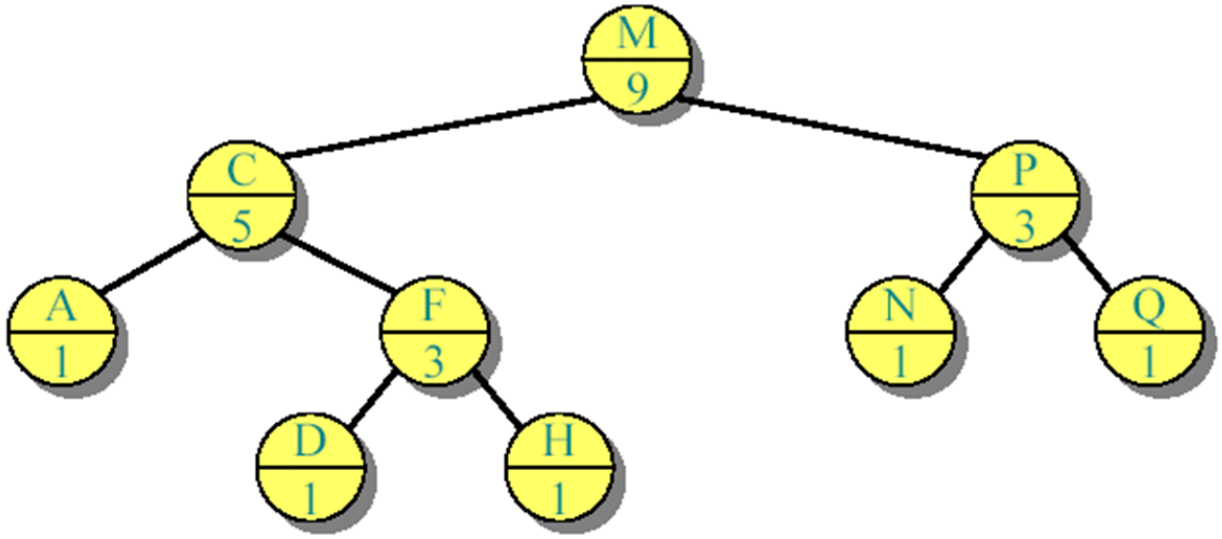
红黑树的扩展应用

- 动态顺序统计量
  - OS-SELECT(S, i)：返回第 i 小的元素
  - OS-RANK(S, x)：返回元素 x 的秩
  - → $O(\log n)$
- Solution
  - 用红黑树存储动态集合 S
  - 在每个节点上增加子树大小的属性 size

Notation for nodes 

动态顺序统计量

❖ 示例



- $x.size = x.left.size + x.right.size + 1$

为什么维护 size 属性（非顺序量）而不是rank属性（顺序量）？

动态顺序统计量

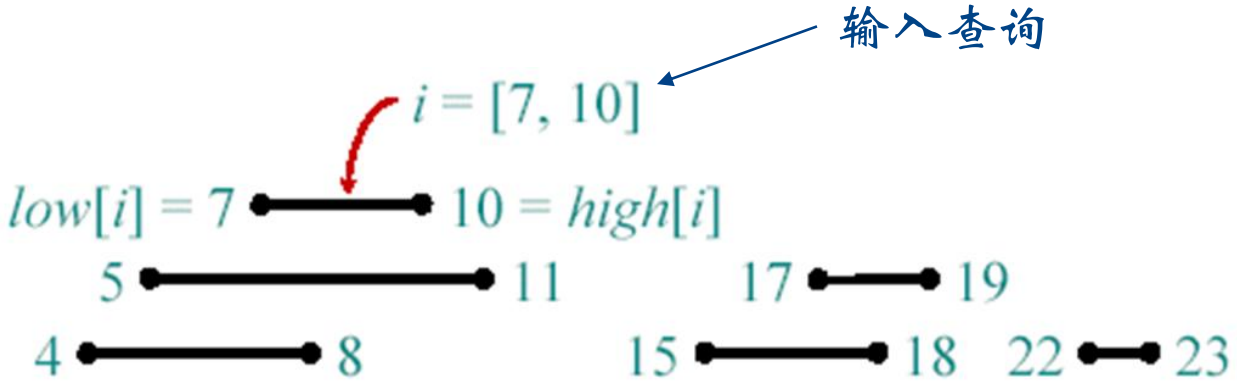
❖ 函数实现

```
OS-SELECT(x, i)
1.  r = x.left.size + 1;
2.  if i == r
3.       return x;
4.  else if i < r
5.      return OS-SELECT(x.left, i);
6.  else
7.      return OS-SELECT(x.right, i-r);
```

```
OS-RANK(T, x)
1.  r = x.left.size + 1;
2.  y = x;
3.  while y ≠ T.root
4.       if y == y.p.right  // right child
5.            r = r + y.p.left.size + 1;
6.                       // nothing for left
7.       y = y.p;
8.  return r;
```

红黑树的扩展应用

- 动态统计量的树维护
  - 在树更新操作时更新size属性
  - → $O(\log n)$
- 区间树
  - 维护一个区间构成的动态集合
  - 参考CLRS



Next

- 动态规划
  - Dynamic Programming