

计算机问题求解 – 论题2-10

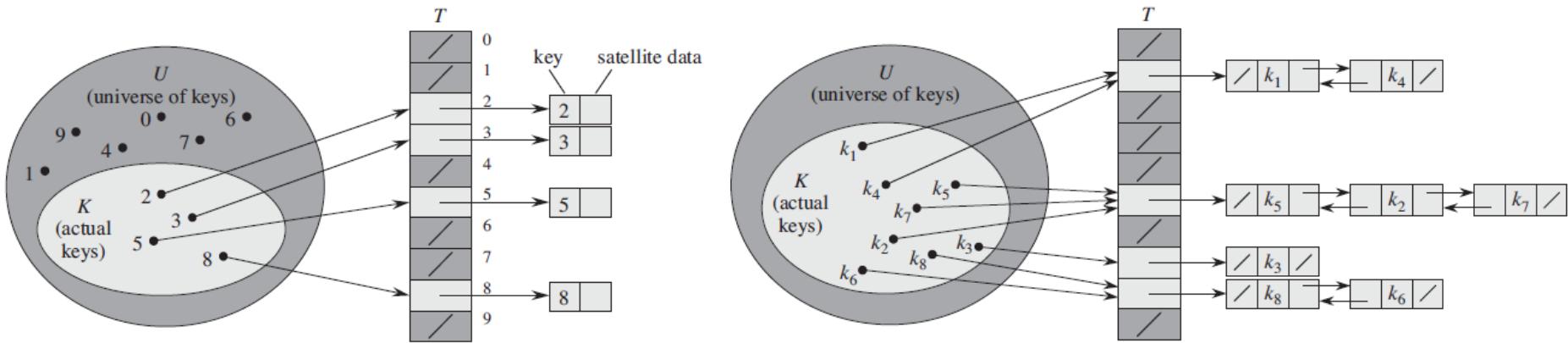
● Hashing方法

课程研讨

- TC第11章
- CS第5章第5节

问题1： dictionary

- 什么是dictionary？
- 你如何理解它的两种实现？
 - direct-address table
 - hash table
- 你能分析它们的存储空间和插入/删除/查找时间吗？
- 因此，你能对比它们的优缺点吗？



问题1： dictionary (续)

- 你理解这段话了吗？

In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

What does this analysis mean? If the number of hash-table slots is at least proportional to the number of elements in the table, we have $n = O(m)$ and, consequently, $\alpha = n/m = O(m)/m = O(1)$. Thus, searching takes constant time on average. Since insertion takes $O(1)$ worst-case time and deletion takes $O(1)$ worst-case time when the lists are doubly linked, we can support all dictionary operations in $O(1)$ time on average.

- 对于dynamic set，如何做到那个“if”？

问题1： dictionary (续)

```
void addEntry(int hash, K key, V value, int bucketIndex) {  
    if ((size >= threshold) && (null != table[bucketIndex])) {  
        resize(2 * table.length);  
        hash = (null != key) ? hash(key) : 0;  
        bucketIndex = indexFor(hash, table.length);  
    }  
  
    createEntry(hash, key, value, bucketIndex);  
}
```

Worst-case Analysis of the Insertion

- For n execution of insertion operations
 - A bad analysis: the worst case for one insertion is the case when expansion is required up to n
Of course NOT!
 - So, the worst case cost is in $O(n^2)$.
- Note the expansion is required during the i th operation only if $i=2^k$, and the cost of the i th operation

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is exactly power of 2} \\ 1 & \text{otherwise} \end{cases}$$

So, the total cost is : $\sum_{i=1}^n c_i \leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n$

问题2：hash function

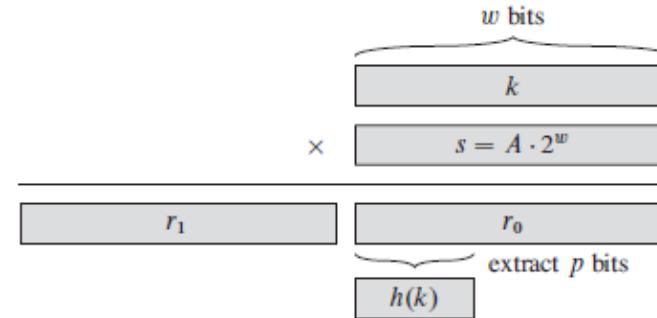
- 你如何理解一个好的hash function应有的这些要素？
 - Satisfies (approximately) the assumption of simple uniform hashing.
 - Derives the hash value in a way that we expect to be independent of any patterns that might exist in the data.
- 你如何理解simple uniform hashing？
它对hash table为什么至关重要？

问题2：hash function (续)

- 你理解这两种hash function了吗？

$$h(k) = k \bmod m$$

$$h(k) = \lfloor m (kA \bmod 1) \rfloor$$



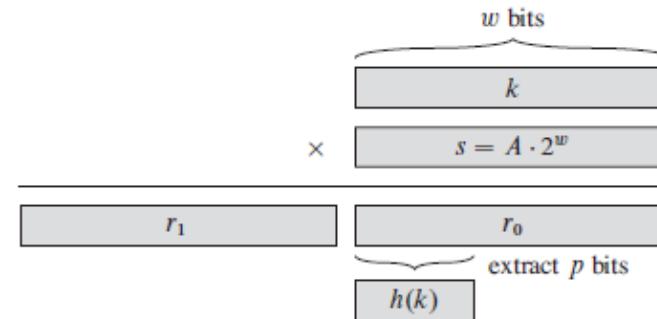
- 这些hash function在实际中能确保是simple uniform hashing吗？
如果不能，可能的原因是什么？如何解决？

问题2：hash function (续)

- 你理解这两种hash function了吗？

$$h(k) = k \bmod m$$

$$h(k) = \lfloor m (kA \bmod 1) \rfloor$$



- 这些hash function在实际中能确保是simple uniform hashing吗？
如果不能，可能的原因是什么？如何解决？
 - universal hashing: to choose the hash function randomly in a way that is independent of the keys that are actually going to be stored

问题3： probability calculations in hashing

- 你会计算这些期望值吗？
 - expected number of items per location n/k
 - expected number of empty locations $k(1 - \frac{1}{k})^n$
 - expected number of collisions $n - k + k(1 - \frac{1}{k})^n$
 - expected time until all locations have at least one item

$$\sum_{j=1}^k \frac{k}{k-j+1}$$

问题4：闭地址散列

- 采用Hashing by Chaining 计算成功搜索与不成功搜索的代价有什么不同?
- 什么是简单一致hash?
- 不成功检索的代价?
 - $\Theta(1 + \alpha)$

问题4：开地址散列

- 你理解open addressing了吗？
它与chaining的本质区别是什么？
因此，它有哪些相对的优缺点？

HASH-INSERT(T, k)

```
1   $i = 0$ 
2  repeat
3       $j = h(k, i)$ 
4      if  $T[j] == \text{NIL}$ 
5           $T[j] = k$ 
6          return  $j$ 
7      else  $i = i + 1$ 
8  until  $i == m$ 
9  error "hash table overflow"
```

HASH-SEARCH(T, k)

```
1   $i = 0$ 
2  repeat
3       $j = h(k, i)$ 
4      if  $T[j] == k$ 
5          return  $j$ 
6       $i = i + 1$ 
7  until  $T[j] == \text{NIL}$  or  $i == m$ 
8  return NIL
```

开地址散列

- 将关键字序列(7、8、30、11、18、9、14)散列存储到散列表中，散列表的存储空间是一个下标从0开始的一个一维数组散列，函数为： $H(key)=(key * 3) \text{MOD } T$ ，处理冲突采用线性探测再散列法，要求装载因子为0.7。问题：
 - 请画出所构造的散列表。
 - 分别计算等概率情况下，查找成功和查找不到成功的平均查找长度。

问题4： collision resolution (续)

- 一个好的h函数应该具有哪些特点？
 -
 -
- 你理解这些h函数了吗？它们为什么不是最好的h函数？
 - linear probing $h(k, i) = (h'(k) + i) \bmod m$
 - quadratic probing $h(k, i) = (h'(k) + c_1i + c_2i^2) \bmod m$
 - double hashing $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$
- 你理解这些具体原因了吗？
 - linear probing: primary clustering
 - quadratic probing: secondary clustering

问题4： collision resolution (续)

- 一个好的h函数应该具有哪些特点？
 - The probe sequence is a permutation of $\langle 0, 1, \dots, m-1 \rangle$.
 - uniform hashing: The probe sequence of each key is equally likely to be any of the $m!$ permutations of $\langle 0, 1, \dots, m-1 \rangle$.
- 你理解这些h函数了吗？它们为什么不是最好的h函数？
 - linear probing $h(k, i) = (h'(k) + i) \bmod m$
 - quadratic probing $h(k, i) = (h'(k) + c_1i + c_2i^2) \bmod m$
 - double hashing $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$
- 你理解这些具体原因了吗？
 - linear probing: primary clustering
 - quadratic probing: secondary clustering

计算机问题求解 – 论题2-11

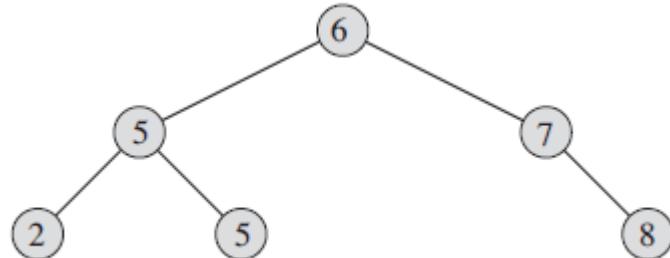
- 搜索树

课程研讨

- TC第12、13章

问题1：binary search trees

- 什么样的binary tree称作binary search tree？
- 和hash table相比，两者作为dictionary的优缺点各是什么？
作为dynamic set呢？



Search
Insert
Delete
Minimum
Maximum
Successor
Predecessor

问题1：binary search trees (续)

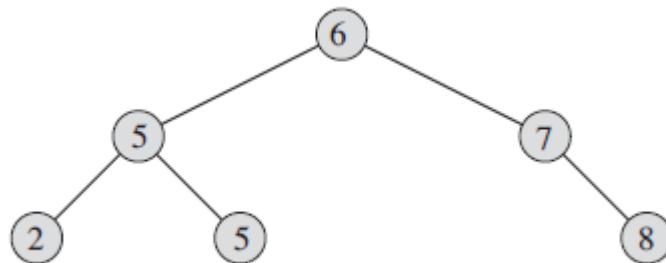
TREE-SEARCH(x, k)

```
1 if  $x == \text{NIL}$  or  $k == x.key$ 
2   return  $x$ 
3 if  $k < x.key$ 
4   return TREE-SEARCH( $x.left, k$ )
5 else return TREE-SEARCH( $x.right, k$ )
```

ITERATIVE-TREE-SEARCH(x, k)

```
1 while  $x \neq \text{NIL}$  and  $k \neq x.key$ 
2   if  $k < x.key$ 
3      $x = x.left$ 
4   else  $x = x.right$ 
5 return  $x$ 
```

- 这两个算法的作用是什么？
- 你能简述它们的主要过程吗？
- 你能证明它们的正确性吗？
- 你能给出它们的运行时间吗？



问题1：binary search trees (续)

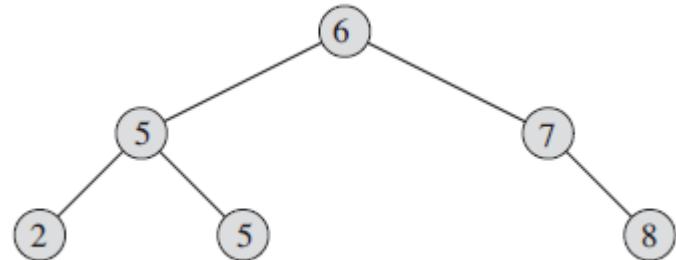
TREE-MINIMUM(x)

```
1 while  $x.left \neq \text{NIL}$ 
2      $x = x.left$ 
3 return  $x$ 
```

TREE-MAXIMUM(x)

```
1 while  $x.right \neq \text{NIL}$ 
2      $x = x.right$ 
3 return  $x$ 
```

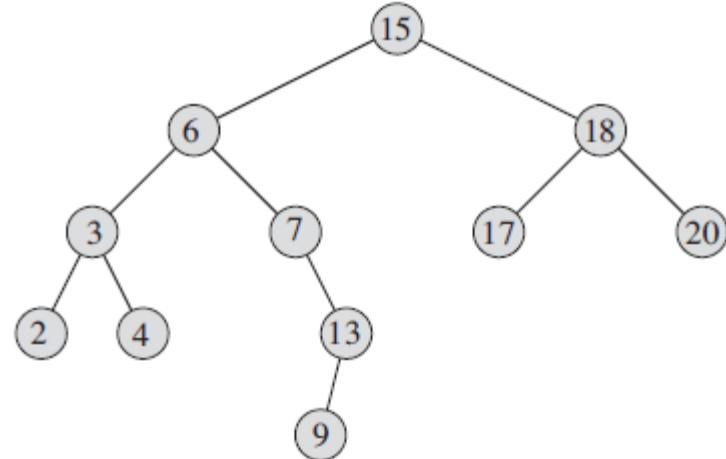
- 这两个算法的作用是什么？
- 你能简述它们的主要过程吗？
- 你能证明它们的正确性吗？
- 你能给出它们的运行时间吗？
- 你能将它们改写成递归形式吗？



问题1：binary search trees (续)

TREE-SUCCESSOR(x)

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



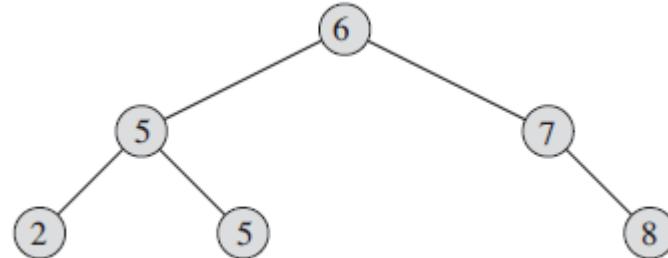
- 这个算法的作用是什么？
- 你能简述它的主要过程吗？
(successor是哪个元素？为什么？)
- 你能给出它的运行时间吗？

问题1：binary search trees (续)

- 你能简述这个算法的主要过程吗？
- 什么样的输入会导致一棵糟糕的binary search tree？
- 如果有人恶意这么做，如何应对？

TREE-INSERT(T, z)

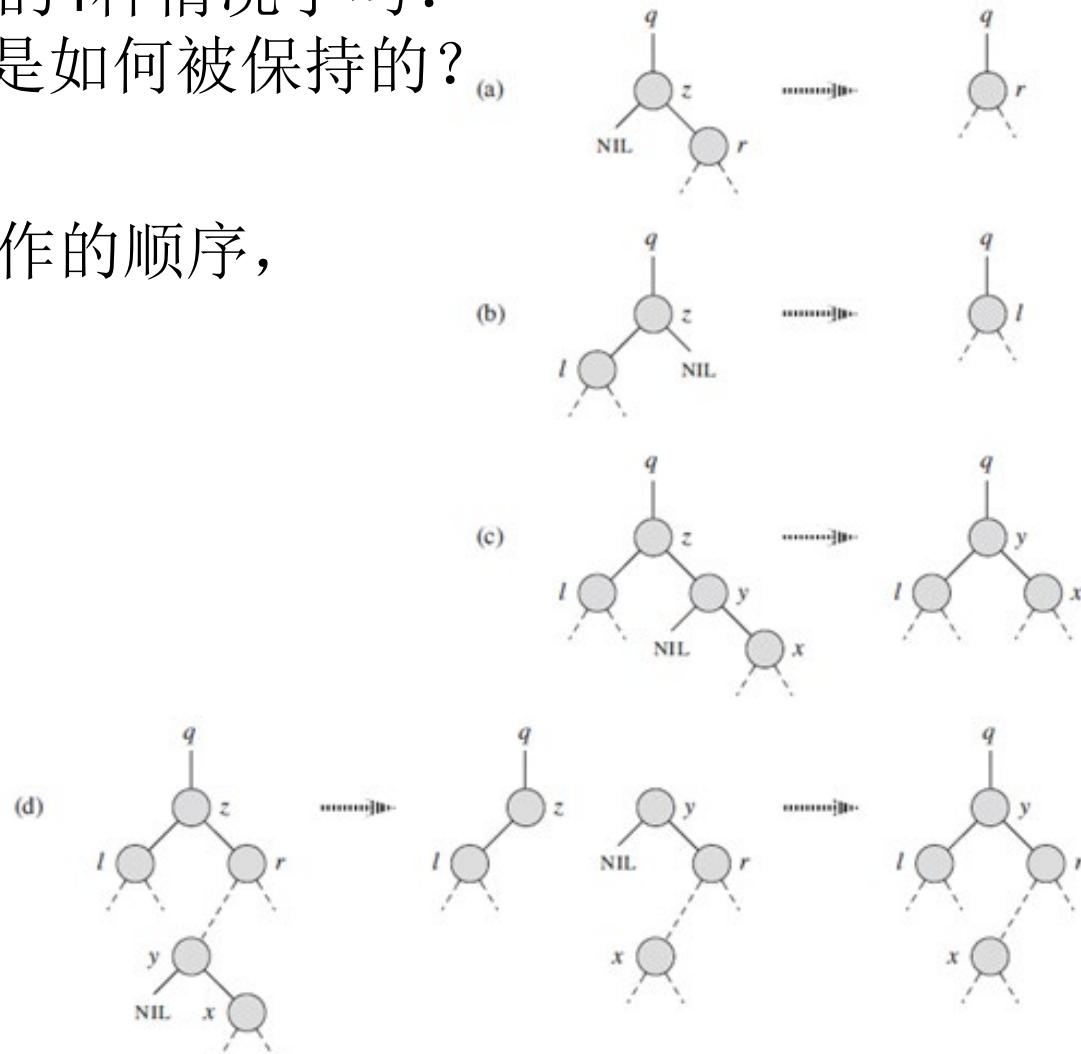
```
1   $y = \text{NIL}$ 
2   $x = T.\text{root}$ 
3  while  $x \neq \text{NIL}$ 
4       $y = x$ 
5      if  $z.\text{key} < x.\text{key}$ 
6           $x = x.\text{left}$ 
7      else  $x = x.\text{right}$ 
8   $z.p = y$ 
9  if  $y == \text{NIL}$ 
10      $T.\text{root} = z$       // tree  $T$  was empty
11  elseif  $z.\text{key} < y.\text{key}$ 
12       $y.\text{left} = z$ 
13  else  $y.\text{right} = z$ 
```



问题1：binary search trees (续)

- 你理解删除顶点的4种情况了吗？
BST的性质分别是如何被保持的？

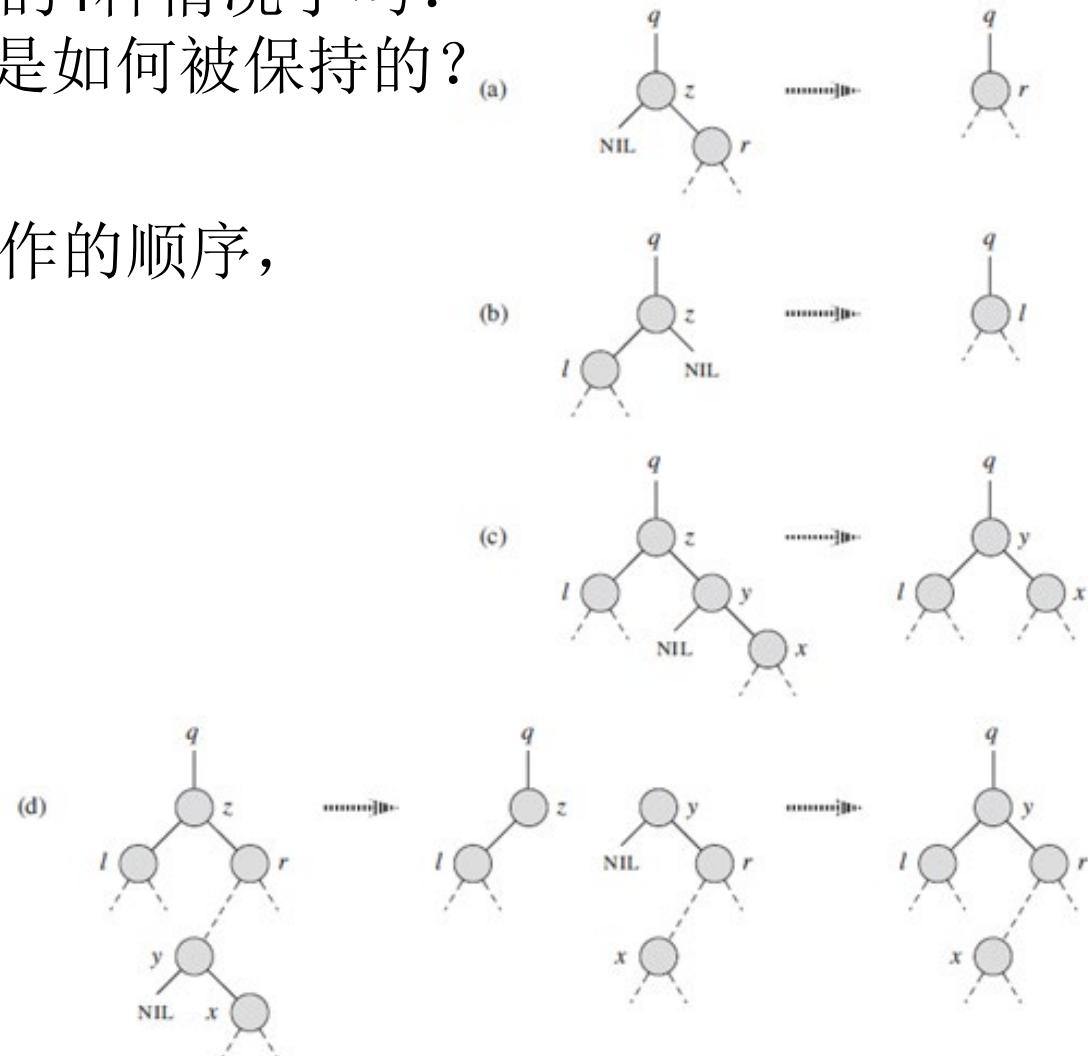
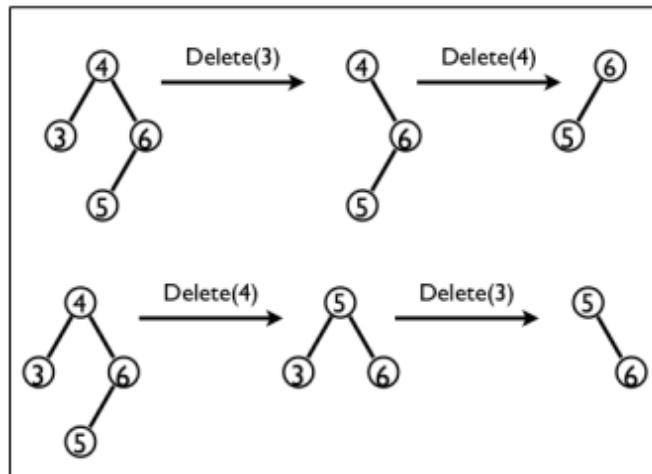
- 交换两个删除操作的顺序，
结果一样吗？



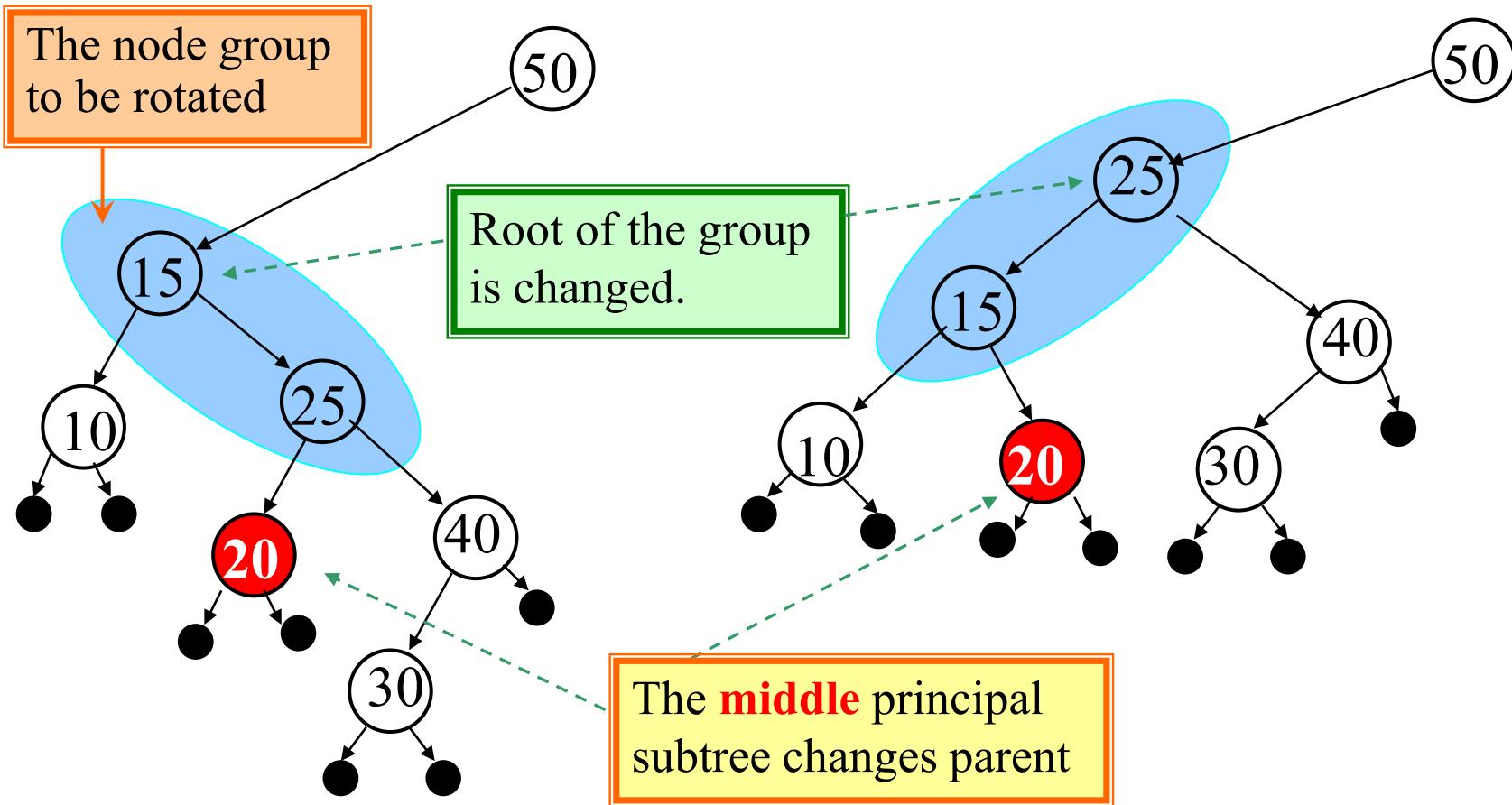
问题1：binary search trees (续)

- 你理解删除顶点的4种情况了吗？
BST的性质分别是如何被保持的？

- 交换两个删除操作的顺序，
结果一样吗？



Improving the Balancing by Rotation

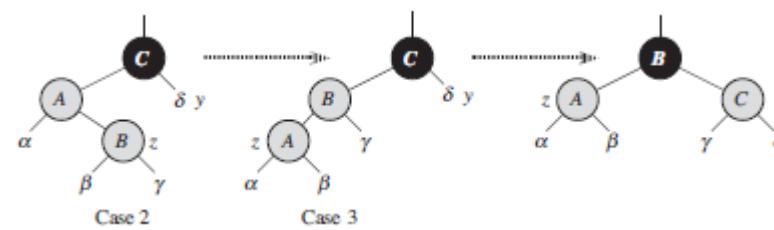
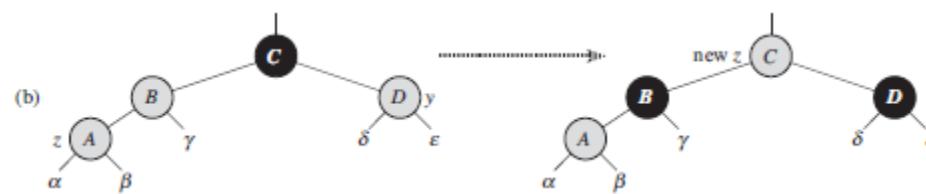
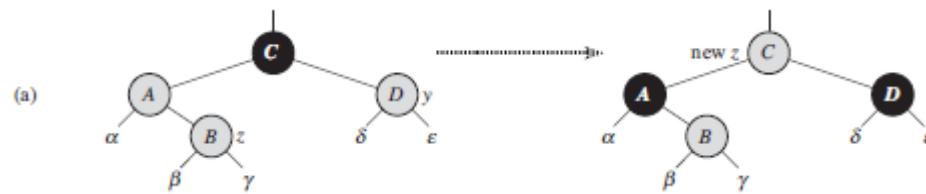


问题2： red-black trees

- red-black tree能平衡到什么程度？
 - No simple path from the root to a leaf is more than twice as long as any other.
- 为什么会具有这种平衡性？
 1. Every node is either red or black.
 2. The root is black.
 3. Every leaf (NIL) is black.
 4. If a node is red, then both its children are black.
 5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

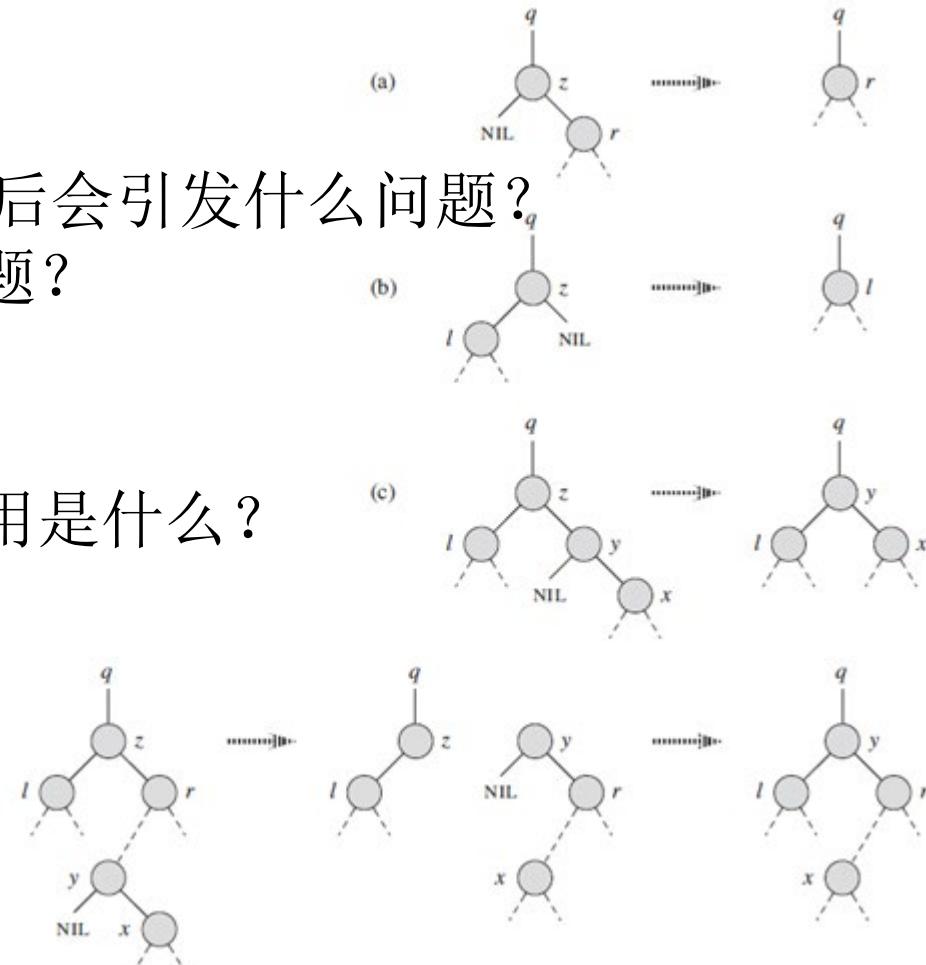
问题2： red-black trees (续)

- 将z (red)插入之后， fixup的主要目标是什么？
 - 保持每条路径上的black数量
 - 消除相连的red
- 你理解每种情况了吗？



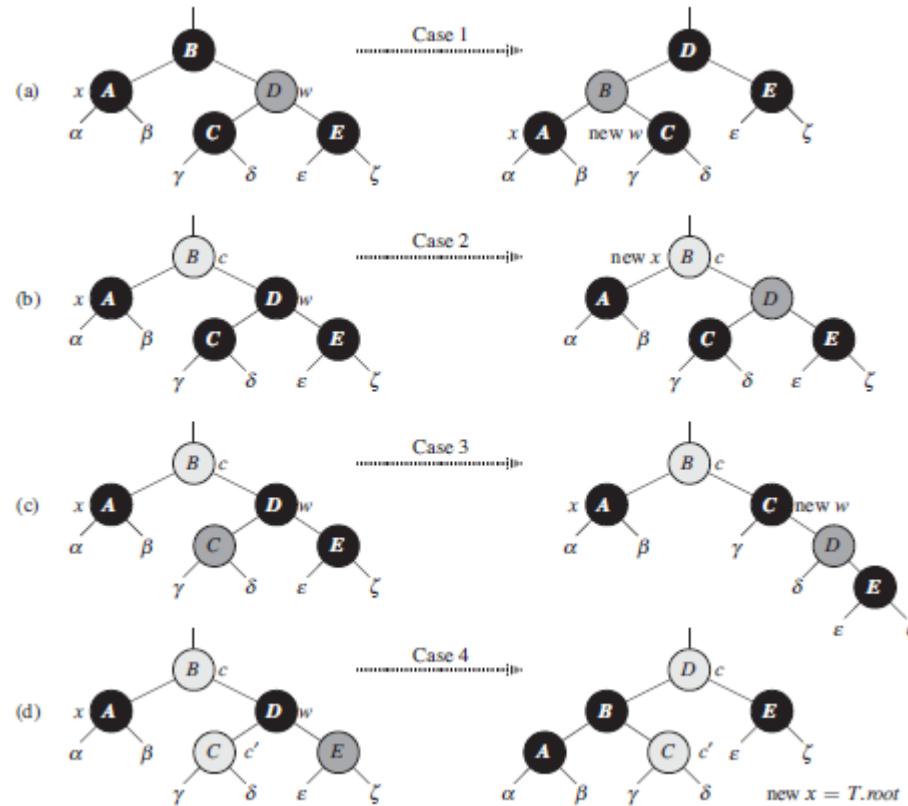
问题2：red-black trees (续)

- 还记得z, y, x的含义吗?
 - y moves into z's position.
 - x moves into y's position.
- 与BST相比，RBT删除z后会引发什么问题？如何先暂时修复这个问题？
 - y moves into z's position.
 - Gives y the same color as z.
- 这种暂时性修复的副作用是什么？什么时候会产生？
 - 当y=black时
y原来的位置可能出问题



问题2：red-black trees (续)

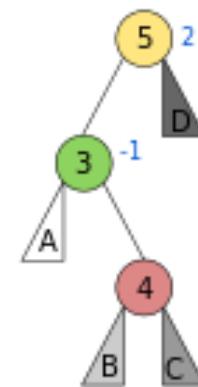
- 如何再修复移走y (black)带来的问题？
 - x moves into y's position.
 - Push y's blackness onto x.
- 这又会产生什么副作用？
 - x可能有超额blackness需要摊出去
- 你理解每种情况的解决办法了吗？



问题2： red-black trees (续)

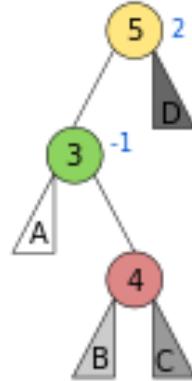
An *AVL tree* is a binary search tree that is *height balanced*: for each node x , the heights of the left and right subtrees of x differ by at most 1. To implement an AVL tree, we maintain an extra attribute in each node: $x.h$ is the height of node x . As for any other binary search tree T , we assume that $T.root$ points to the root node.

To insert into an AVL tree, we first place a node into the appropriate place in binary search tree order. Afterward, the tree might no longer be height balanced. Specifically, the heights of the left and right children of some node might differ by 2. Describe a procedure $BALANCE(x)$, which takes a subtree rooted at x whose left and right children are height balanced and have heights that differ by at most 2, i.e., $|x.right.h - x.left.h| \leq 2$, and alters the subtree rooted at x to be height balanced.

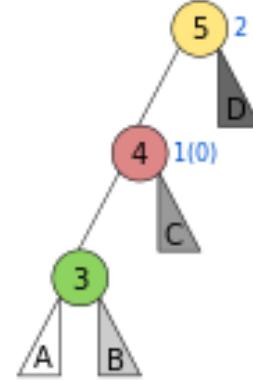


问题2：red-black trees (续)

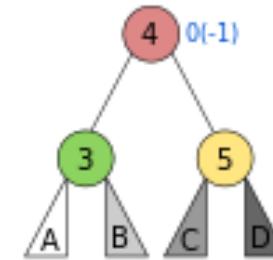
Left Right Case



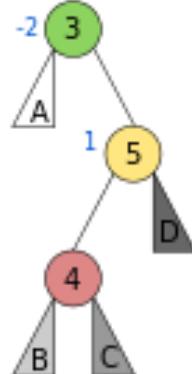
Left Left Case



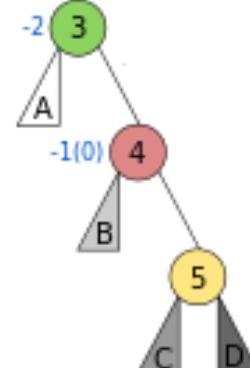
Balanced



Right Left Case



Right Right Case



Balanced

