计算机问题求解 – 论题2-11

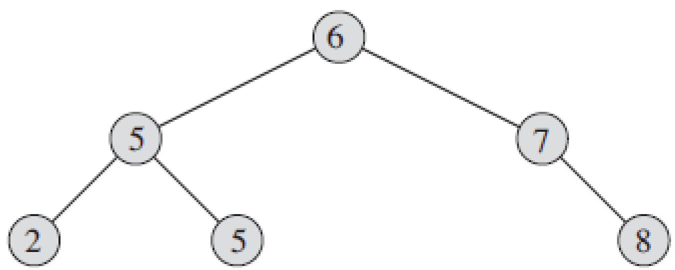
- 搜索树

课程研讨

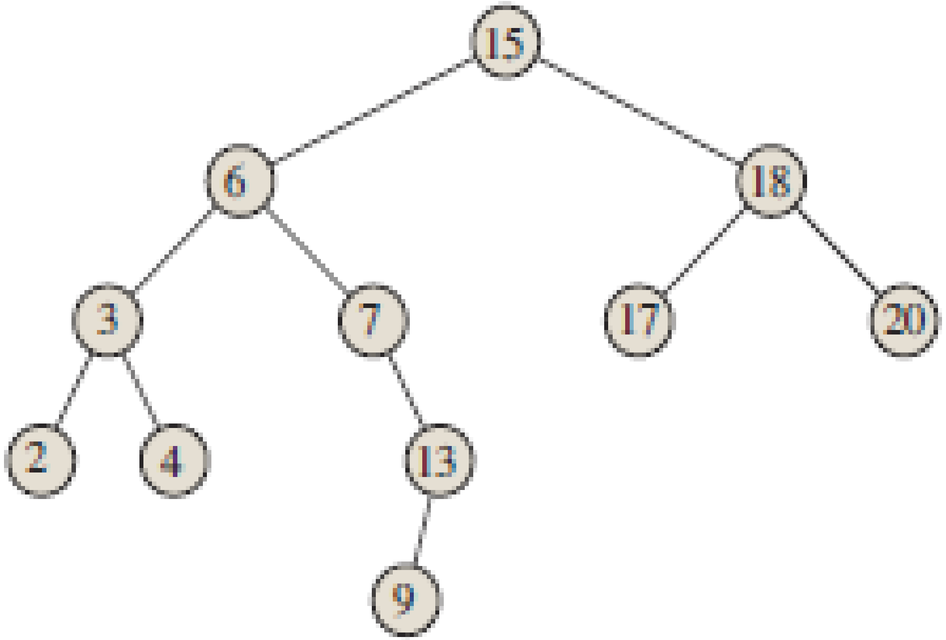
- TC第12、13章

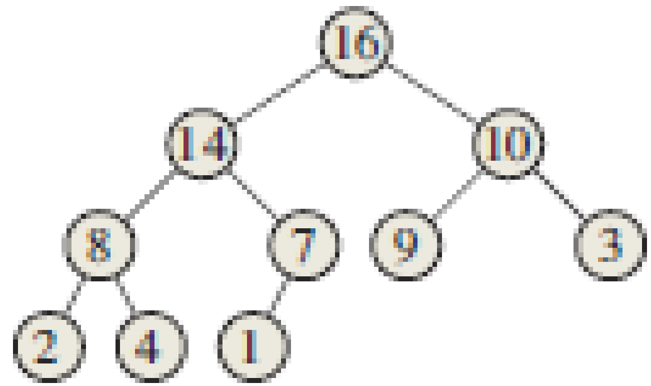
问题1： binary search trees

- 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？



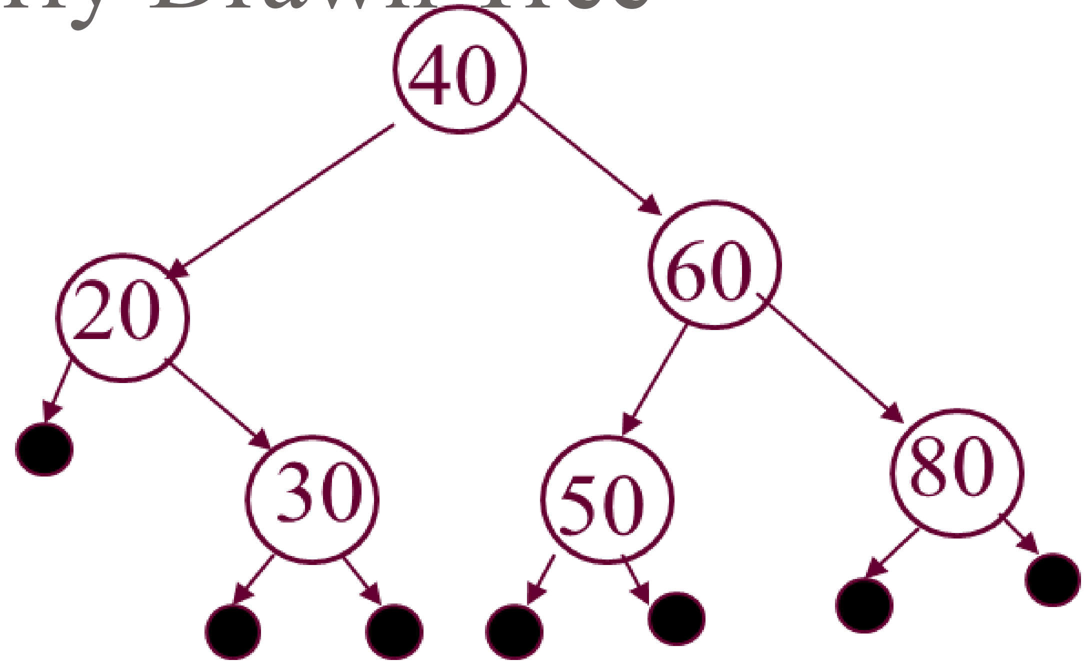
Search
Insert
Delete
Minimum
Maximum
Successor
Predecessor

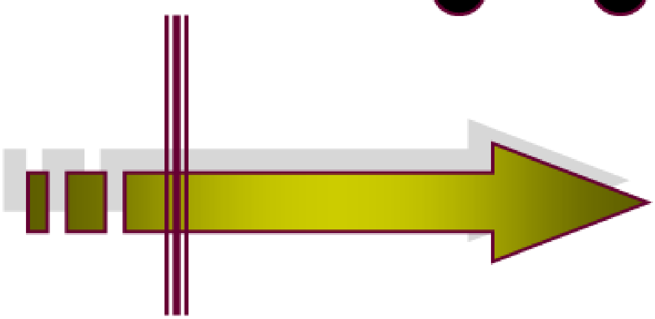




这是什么结构？他们有什么相同与不同之处？

Properly Drawn Tree





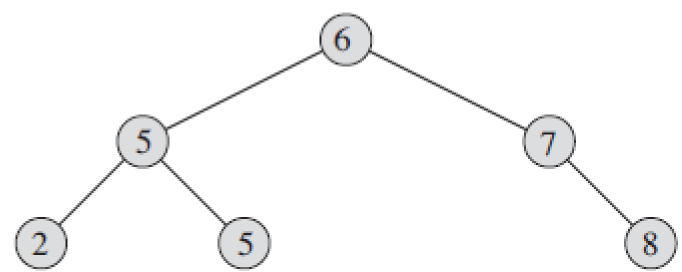
In a properly drawn tree, pushing forward to get the ordered list.

问题1： binary search trees (续)

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
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？

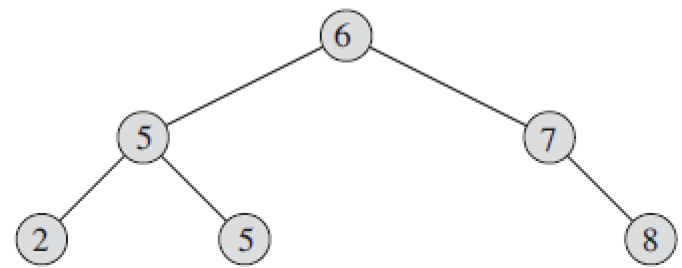


问题1： binary search trees (续)

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

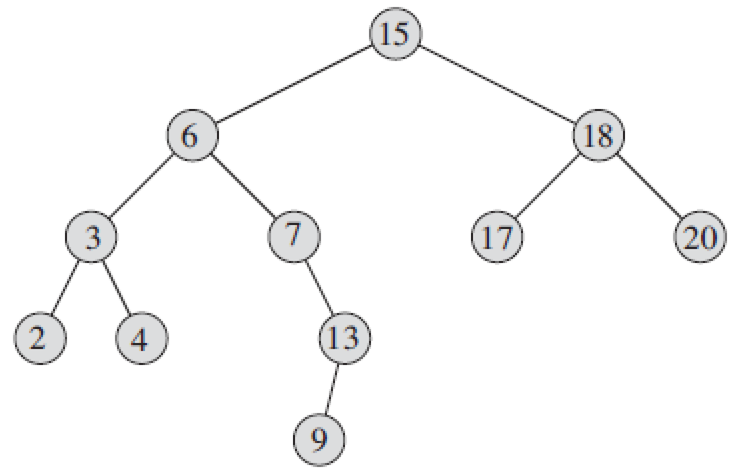
```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```

- 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)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

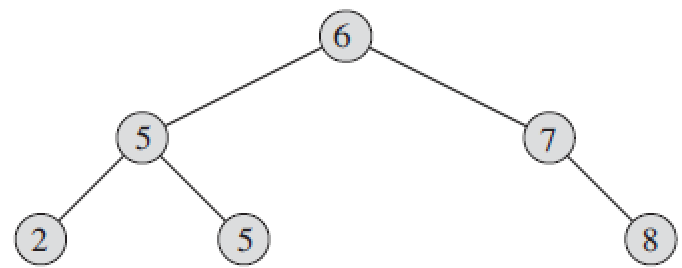


- 这个算法的作用是什么？
- 你能简述它的主要过程吗？
  （ successor是哪个元素？为什么？ ）
- 你能给出它的运行时间吗？

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？

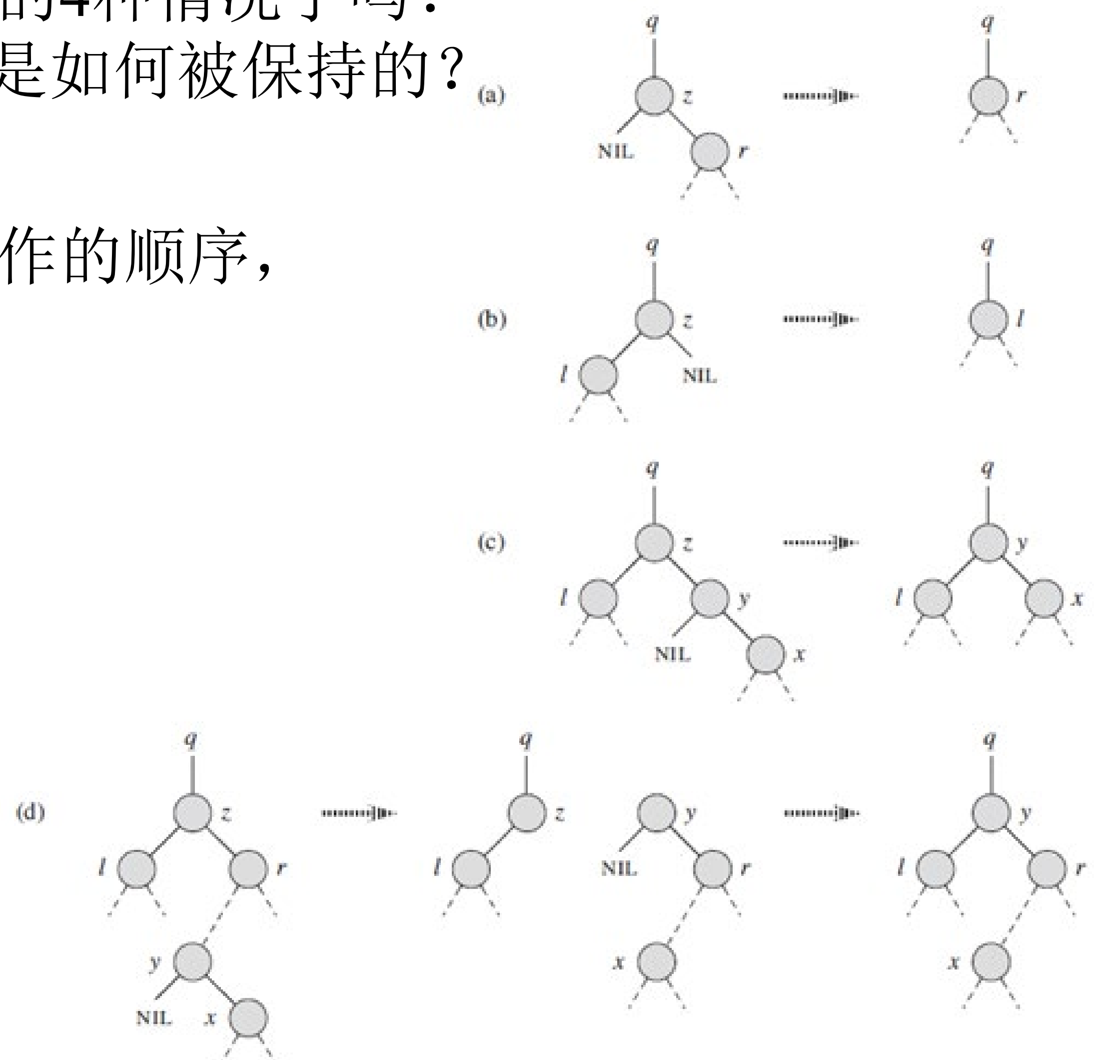
```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z          // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```



问题1： binary search trees (续)

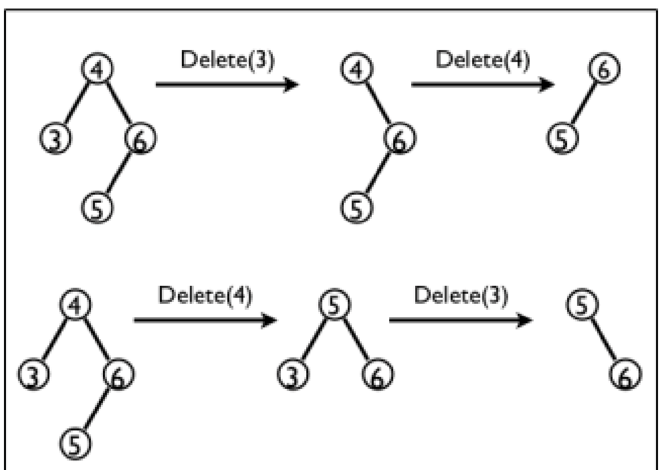
- 你理解删除顶点的4种情况了吗？
  BST的性质分别是如何被保持的？

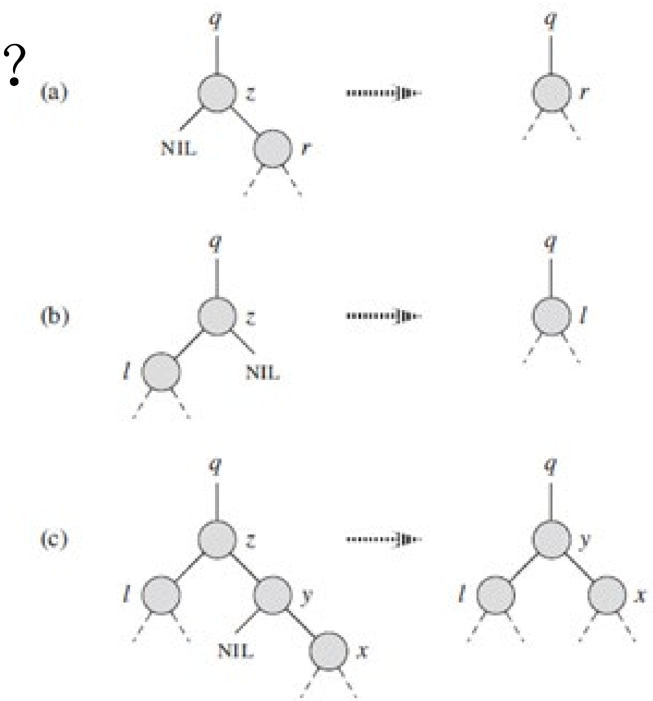
- 交换两个删除操作的顺序，
  结果一样吗？

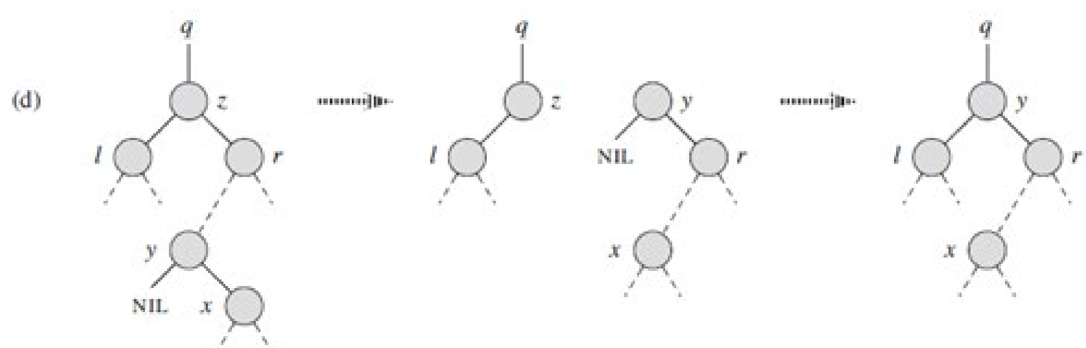


问题1： binary search trees (续)

- 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？







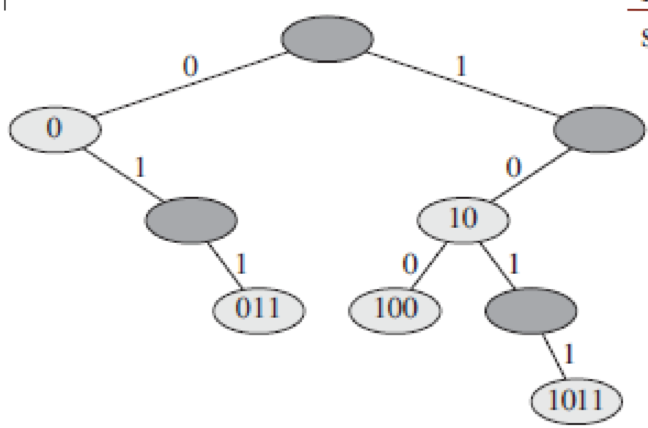
Given two strings $a = a_0 a_1 \ldots a_p$ and $b = b_0 b_1 \ldots b_q$, where each a_i and each b_j is in some ordered set of characters, we say that string a is ***lexicographically less than*** string b if either

1. there exists an integer j, where $0 \le j \le \min(p, q)$, such that $a_i = b_i$ for all $i = 0, 1, \ldots, j-1$ and $a_j < b_j$, or
2. $p < q$ and $a_i = b_i$ for all $i = 0, 1, \ldots, p$.

For example, if a and b are bit strings, then $10100 < 10110$ by rule 1 (letting $j = 3$) and $10100 < 101000$ by rule 2. This ordering is similar to that used in English-language dictionaries.

The ***radix tree*** data structure shown in Figure 12.5 stores the bit strings 1011, 10, 011, 100, and 0. When searching for a key $a = a_0 a_1 \ldots a_p$, we go left at a node of depth i if $a_i = 0$ and right if $a_i = 1$. Let S be a set of distinct bit strings whose lengths sum to n. Show how to use a radix tree to sort S lexicographically in $\Theta(n)$ time. For the example in Figure 12.5, the output of the sort should be the sequence 0, 011, 10, 100, 1011.

问题1

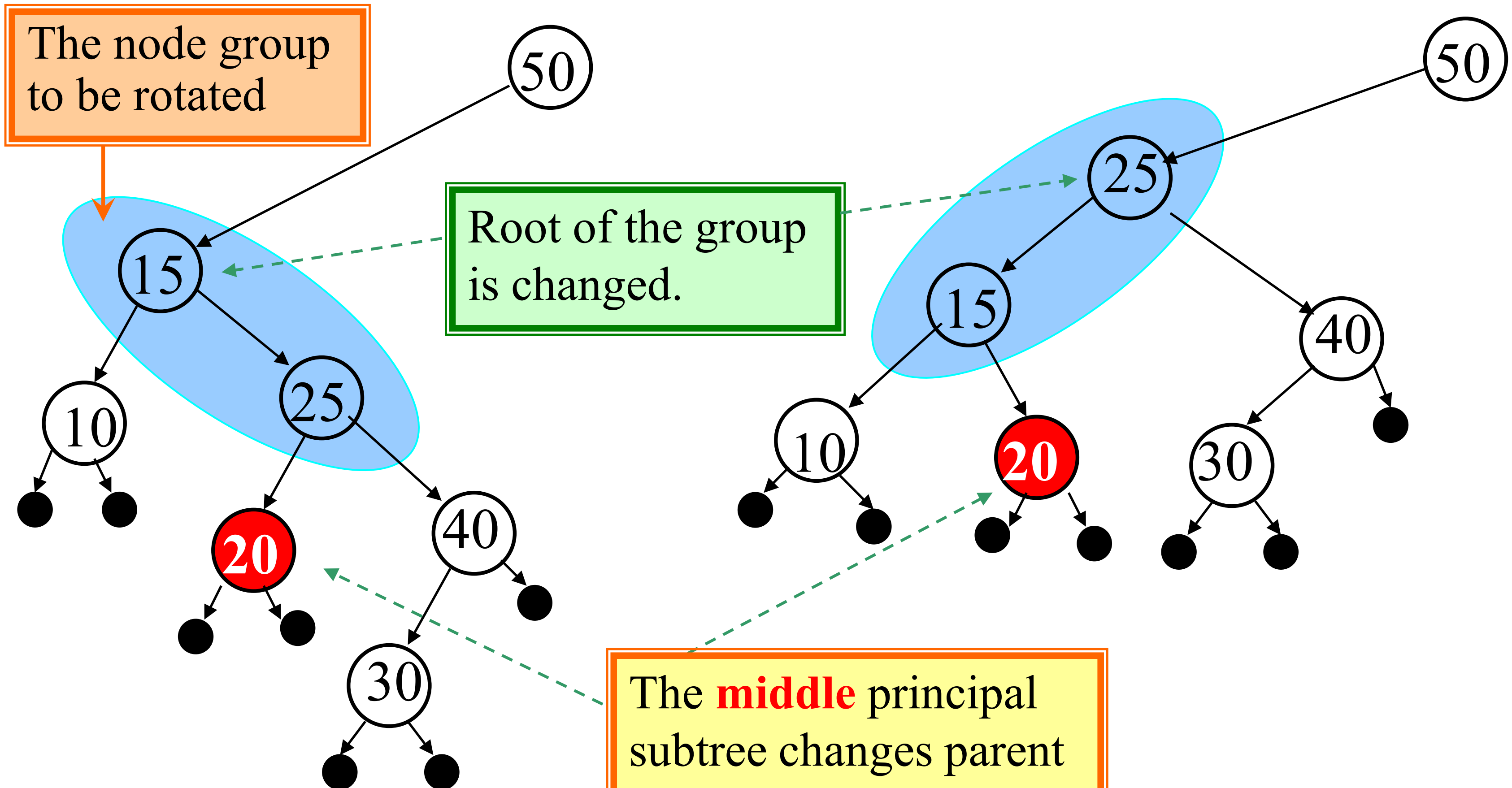


问题2：和hash table相比，两者作为dictionary，哪个更快？（仔细想一想）

Hash tables are commonly said to have expected O(1) insertion and deletion times, but this is only true when considering computation of the hash of the key to be a constant time operation. When hashing the key is taken into account, hash tables have expected O(k) insertion and deletion times, but may take longer in the worst-case depending on how collisions are handled. Radix trees have worst-case O(k) insertion and deletion.

Figure 12.5 A radix tree storing the bit strings 1011, 10, 011, 100, and 0. We can determine each node's key by traversing the simple path from the root to that node. There is no need, therefore, to store the keys in the nodes; the keys appear here for illustrative purposes only. Nodes are heavily shaded if the keys corresponding to them are not in the tree; such nodes are present only to establish a path to other nodes.

Improving the Balancing by Rotation



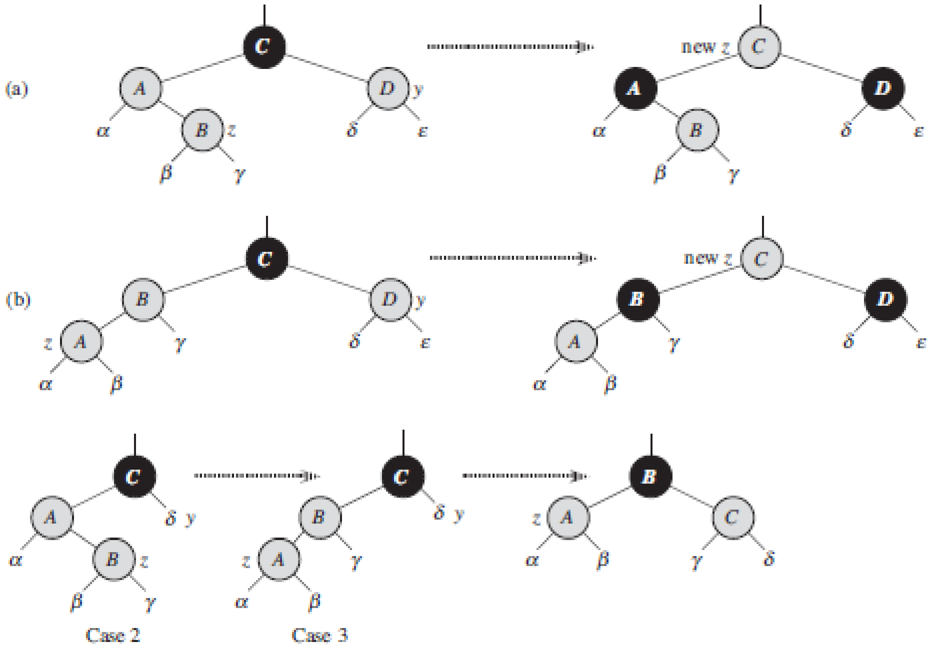
问题2： red-black trees

- 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.

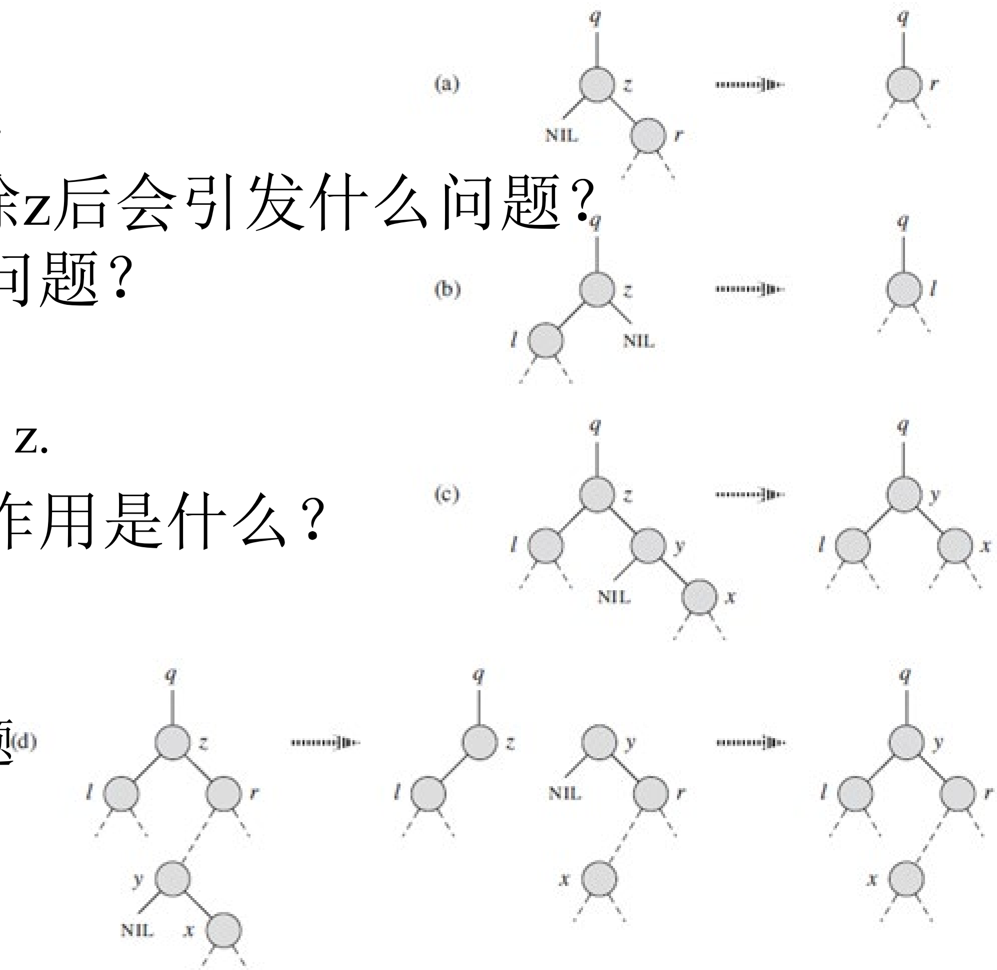
问题2：red-black trees (续)

- 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？



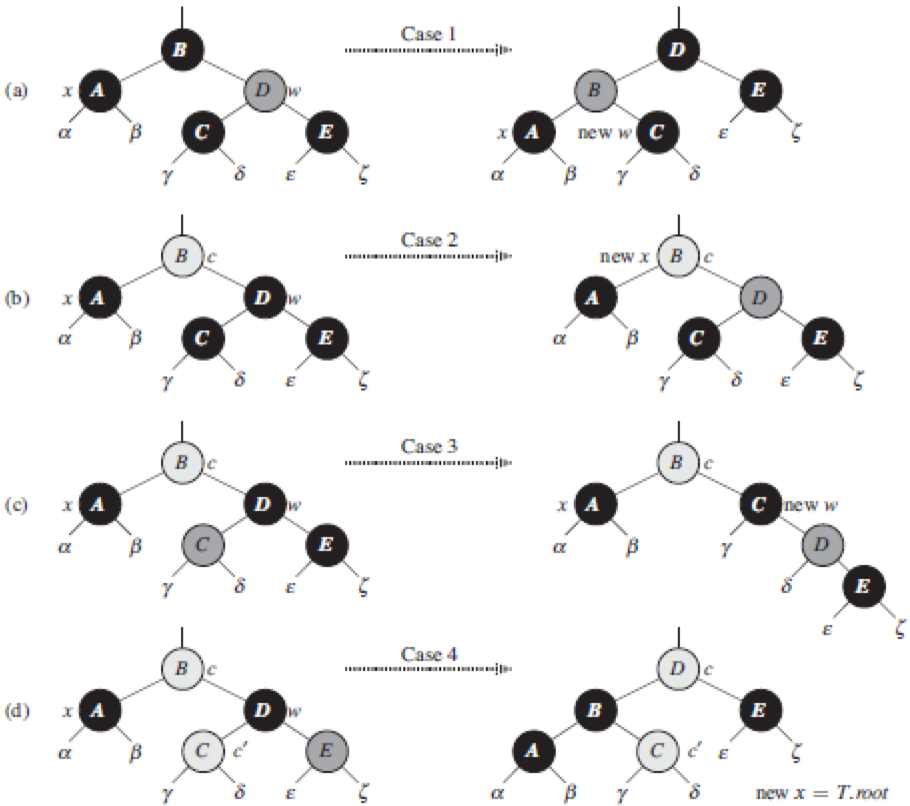
问题2： red-black trees (续)

- 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)

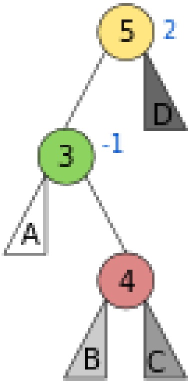
- 如何再修复移走y (black)带来的问题？
  - x moves into y’s position.
  - Push y’s blackness onto x.
- 这又会产生什么副作用？
  - x可能有超额blackness需要摊出去
- 你理解每种情况的解决办法了吗？



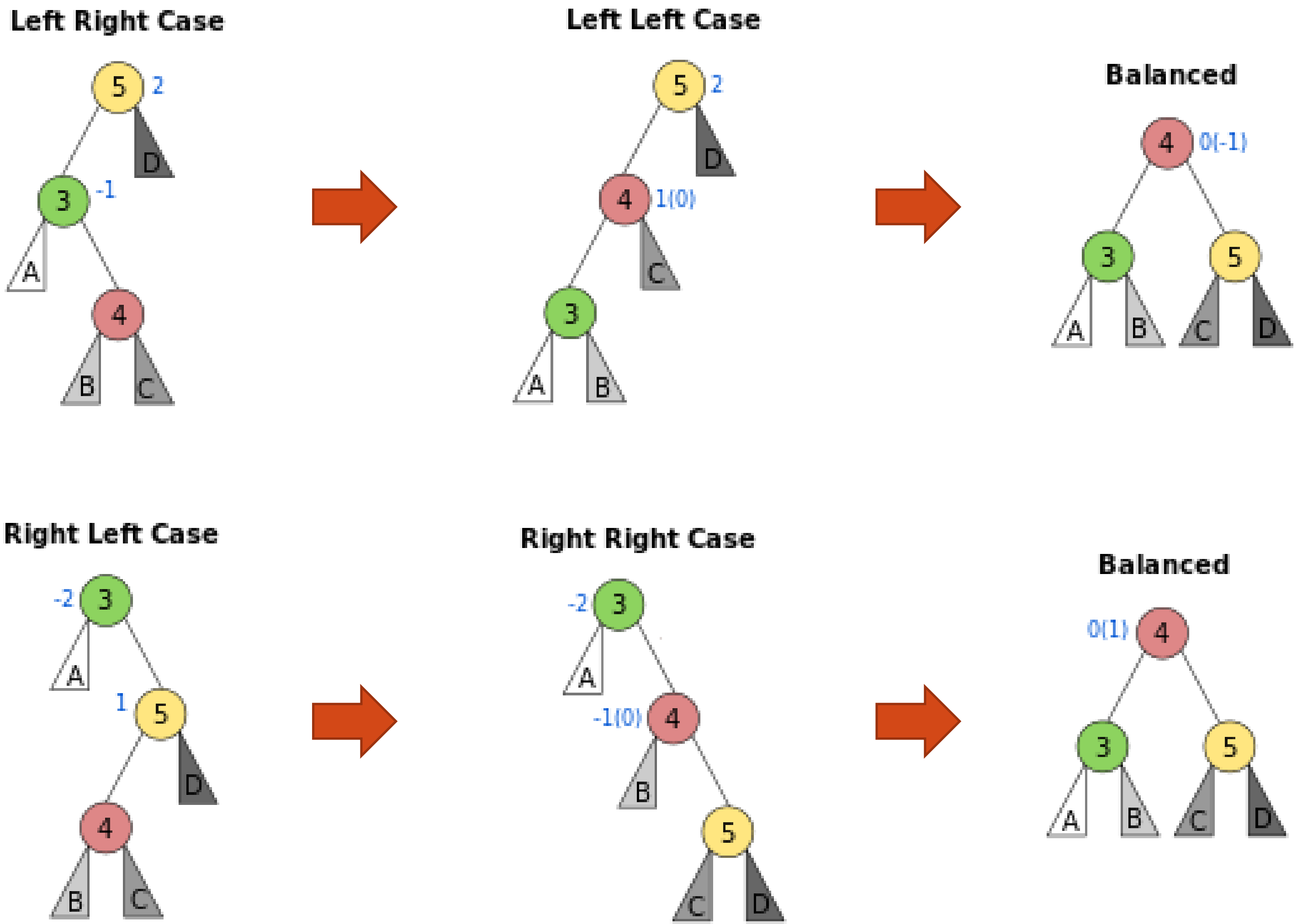
问题2： red-black trees (续)

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)

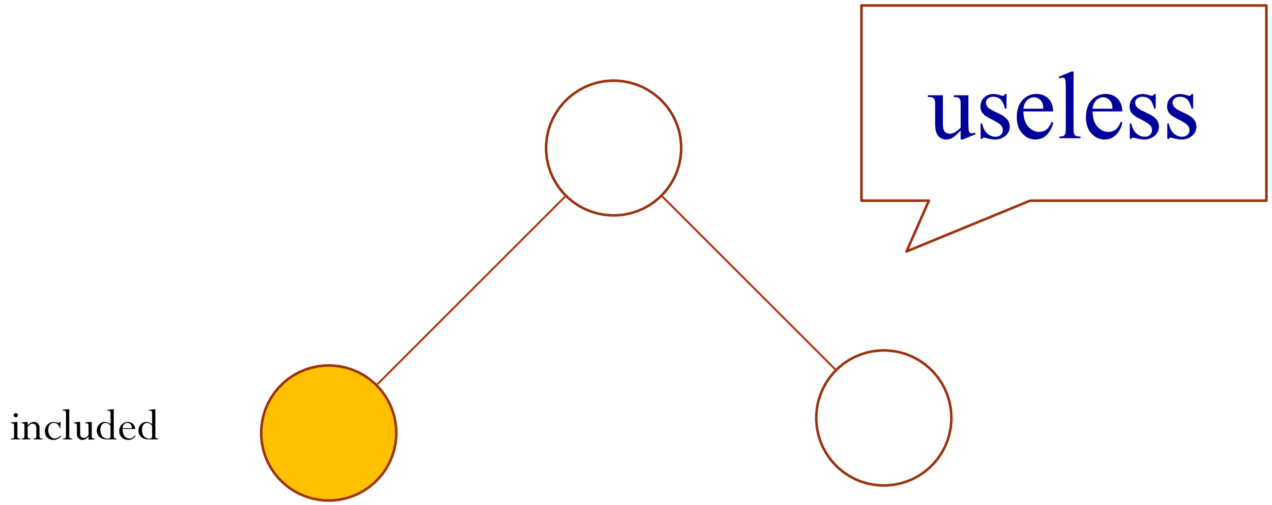


Adversary Argument

- Let $b = b_1 b_2\ b_3\ b_4\ b_5$ be a bit string of length 5, i.e. $\in \{0,1\}\ b_\mathrm{i}$ for $1 \le i \le 5$. Consider the problem of determining whether b contains three consecutive ones, i.e. whether or not b contains the substring 111. We restrict our attention to those algorithms whose only allowable operation is to peek at a bit.

First Glance…

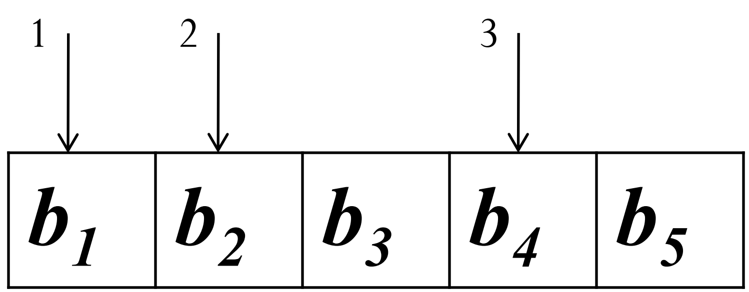
- Obviously 5 peeks are sufficient.
- A decision tree argument provides the fact that at least one peek is necessary.



Adversary Strategy

- Consider any algorithm for this problem and start it on an unspecified bit *b* string of length 5. The adversary strategy is to answer 0 to any bit peek, unless that answer would prove that *b* does not contain three consecutive ones.

0	**0**	**1**	**1**	**1**
0	**0**	**0**	**1**	**0**



0	**0**	×	**1**	×

Daemon Algorithm: Peek

- Let x =11111 and y = 00000
- Function flip(u,i)
  - which takes a bit string u and flips it's ith bit (0 to 1, or 1 to 0), then returns the new bit string.
- When the algorithm peeks at bit i, the Daemon performs the algorithm Peek(i).

Daemon Algorithm: Peek

- Let x =11111 and y = 00000
- Function flip(u,i)
  - which takes a bit strin
    to 1, or 1 to 0), then re
- When the algorithm p
  Daemon performs the

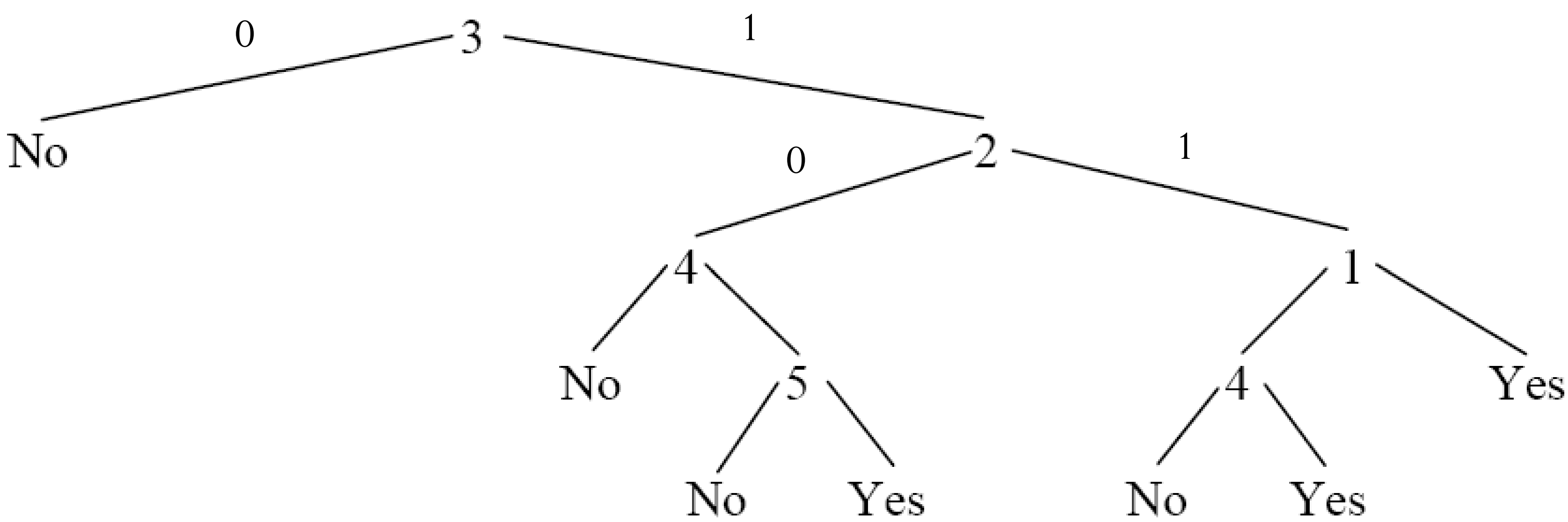
Peek(i)
1. if flip(x, i) contains
   the substring 111
2. $x \leftarrow flip(x, i)$
3. answer 0
4. else
5. $y \leftarrow flip(y, i)$
6. answer 1

Lower Bound by Adversary Strategy

- If only 3 peeks have been performed, then *y* can contain at most 2 ones.
  - To prove this, assume that after peeking at 3 bits, *y* contains 3 ones. Then it must be the case that if any of those bits were flipped in x =11111, then x would not contain the substring 111. But there are not 3 such bits in x =11111.
- If only 3 peeks are performed, *y* cannot contain the substring 111.
- Algorithm with 3 peeks could not possibly be correct
  - If the verdict is yes, we can claim that $b = y$
  - Else if the verdict is no, we can claim that $b = x$

Possible Solution

- The height of this decision tree is 4, by the above proof, this is the optimal algorithm.



Analysis of Finding the Second

- Any algorithm that finds *secondLargest* must also find *max* before. (n-1)
- The *secondLargest* can only be in those which lose directly to *max*.
- On its path along which bubbling up to the root of tournament tree, *max* beat $\lceil \lg n \rceil$ keys at most.
- Pick up *secondLargest*. ($\lceil \lg n \rceil$-1)
- $\boldsymbol{n + \lceil \lg n \rceil - 2}$

Lower Bound by Adversary

- Theorem
  - Any algorithm (that works by comparing keys) to find the second largest in a set of n keys must do at least $n+\lceil \lg n \rceil-2$ comparisons in the worst case.
  - Proof

    There is an adversary strategy that can force any algorithm that finds *secondLargest* to compare *max* to $\lceil \lg n \rceil$ distinct keys.

Weighted Key

Note: for one comparison, the weight increasing is no more than doubled.

- Assigning a weight $w(x)$ to each key. The initial values are all 1.
- Adversary rules:

Case	Adversary reply	Updating of weights
$w(x)>w(y)$	x>y	$w(x):=w(x)+w(y)$; $w(y):=0$
$w(x)=w(y)>0$	$x>y$	$w(x):=w(x)+w(y)$; $w(y):=0$
$w(y)>w(x)$	$y>x$	$w(y):=w(x)+w(y)$; $w(x):=0$
$w(x)=w(y)=0$	Consistent with previous replies	No change

Zero=Loss

Lower Bound by Adversary: Details

- Note: the sum of weights is always n.
- Let x is *max*, then x is the only nonzero weighted key, that is $w(x)=n$.
- By the adversary rules:

$$w_k(x) \leq 2w_{k-1}(x)$$

- Let K be the number of comparisons x wins against previously undefeated keys:

$$n=w_K(x)\leq 2^K w_0(x)=2^K$$

- So, $K \geq \lceil \lg n \rceil$

Tracking the Losers to *MAX*

Building a heap structure of 2*n*-1 entries, using *n*-1 extra space