Hash tables, Binary search trees, Red-black trees

Congduan Li

Chinese University of Hong Kong, Shenzhen

congduan.li@gmail.com

Sep 26 & 28, 2017

Direct-address tables

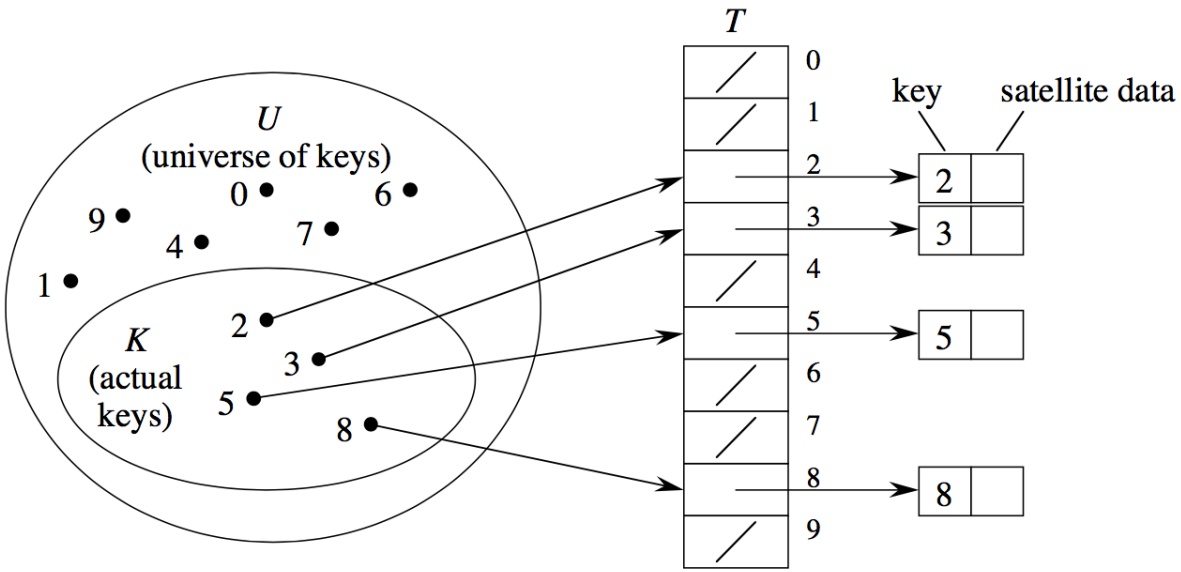
Scenario:

- Maintain a dynamic set.
- Each element has a key drawn from a universe $U = \{0, 1, \ldots, m-1\}$ where m isn't too large.
- No two elements have the same key.

Represent by a ***direct-address table***, or array, $T[0 \ldots m-1]$:

- Each ***slot***, or position, corresponds to a key in U.
- If there's an element x with key k, then $T[k]$ contains a pointer to x.
- Otherwise, $T[k]$ is empty, represented by NIL.

Direct-address tables



Dictionary operations are trivial and take $O(1)$ time each:

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

DIRECT-ADDRESS-INSERT(T, x)
$T[key[x]] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)
$T[key[x]] \leftarrow$ NIL

Hash tables

The problem with direct addressing is if the universe U is large, storing a table of size $|U|$ may be impractical or impossible.

Often, the set K of keys actually stored is small, compared to U, so that most of the space allocated for T is wasted.

- When K is much smaller than U, a hash table requires much less space than a direct-address table.
- Can reduce storage requirements to $\Theta(|K|)$.
- Can still get $O(1)$ search time, but in the *average case*, not the *worst case*.

Idea: Instead of storing an element with key k in slot k, use a function h and store the element in slot $h(k)$.

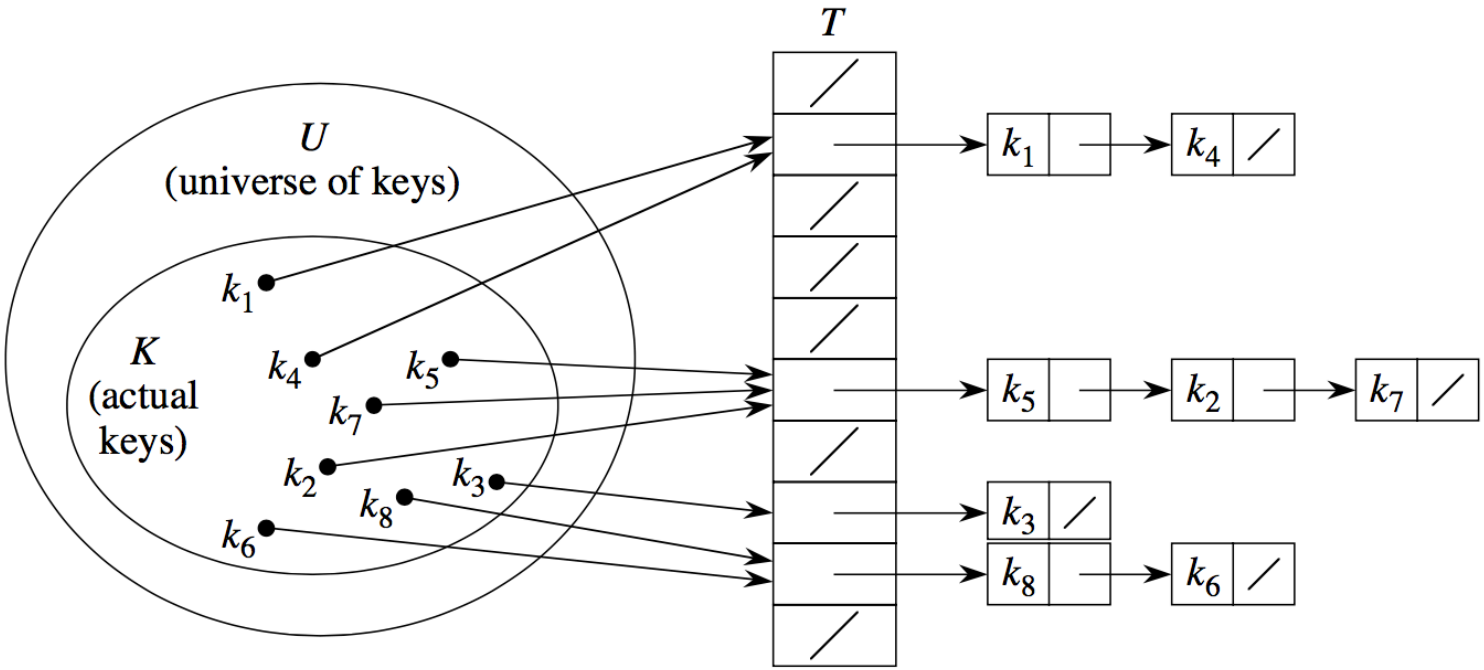
- We call h a ***hash function***.
- $h : U \to \{0, 1, \ldots, m-1\}$, so that $h(k)$ is a legal slot number in T.
- We say that k ***hashes*** to slot $h(k)$.

Collisions

Collisions: When two or more keys hash to the same slot.

- Can happen when there are more possible keys than slots ($|U| > m$).
- For a given set K of keys with $|K| \leq m$, may or may not happen. Definitely happens if $|K| > m$.
- Therefore, must be prepared to handle collisions in all cases.
- Use two methods: chaining and open addressing.
- Chaining is usually better than open addressing.

Collision resolution by chaining



[This figure shows singly linked lists. If we want to delete elements, it's better to use doubly linked lists.]

- Slot j contains a pointer to the head of the list of all stored elements that hash to j *[or to the sentinel if using a circular, doubly linked list with a sentinel]* ,
- If there are no such elements, slot j contains NIL.

Operations with chaining

- ***Insertion:***

  CHAINED-HASH-INSERT(T, x)
  insert x at the head of list $T[h(key[x])]$

  - Worst-case running time is $O(1)$.
  - Assumes that the element being inserted isn't already in the list.
  - It would take an additional search to check if it was already inserted.

- ***Search:***

  CHAINED-HASH-SEARCH(T, k)
  search for an element with key k in list $T[h(k)]$

  Running time is proportional to the length of the list of elements in slot $h(k)$.

Operations with chaining

- ***Deletion:***

  CHAINED-HASH-DELETE(T, x)
  delete x from the list $T[h(key[x])]$

  - Given pointer x to the element to delete, so no search is needed to find this element.
  - Worst-case running time is $O(1)$ time if the lists are doubly linked.
  - If the lists are singly linked, then deletion takes as long as searching, because we must find x's predecessor in its list in order to correctly update *next* pointers.

Analysis of hashing with chaining

Given a key, how long does it take to find an element with that key, or to determine that there is no element with that key?

- Analysis is in terms of the ***load factor*** $\alpha = n/m$:
  - $n =$ # of elements in the table.
  - $m =$ # of slots in the table $=$ # of (possibly empty) linked lists.
  - Load factor is average number of elements per linked list.
  - Can have $\alpha < 1$, $\alpha = 1$, or $\alpha > 1$.
- Worst case is when all n keys hash to the same slot $\Rightarrow$ get a single list of length n $\Rightarrow$ worst-case time to search is $\Theta(n)$, plus time to compute hash function.
- Average case depends on how well the hash function distributes the keys among the slots.

Analysis of hashing with chaining

We focus on average-case performance of hashing with chaining.

- Assume ***simple uniform hashing***: any given element is equally likely to hash into any of the m slots.
- For $j = 0, 1, \ldots, m-1$, denote the length of list $T[j]$ by n_j. Then $n = n_0 + n_1 + \cdots + n_{m-1}$.
- Average value of n_j is $\mathrm{E}[n_j] = \alpha = n/m$.
- Assume that we can compute the hash function in $O(1)$ time, so that the time required to search for the element with key k depends on the length $n_{h(k)}$ of the list $T[h(k)]$.

We consider two cases:

- If the hash table contains no element with key k, then the search is unsuccessful.
- If the hash table does contain an element with key k, then the search is successful.

Unsuccessful search

Theorem

An unsuccessful search takes expected time $\Theta(1+\alpha)$.

Proof Simple uniform hashing $\Rightarrow$ any key not already in the table is equally likely to hash to any of the m slots.

To search unsuccessfully for any key k, need to search to the end of the list $T[h(k)]$. This list has expected length $\mathrm{E}[n_{h(k)}] = \alpha$. Therefore, the expected number of elements examined in an unsuccessful search is α.

Adding in the time to compute the hash function, the total time required is $\Theta(1+\alpha)$. ■

Successful search

Theorem

A successful search takes expected time $\Theta(1+\alpha)$.

Proof Assume that the element x being searched for is equally likely to be any of the n elements stored in the table.

The number of elements examined during a successful search for x is 1 more than the number of elements that appear before x in x's list. These are the elements inserted *after* x was inserted (because we insert at the head of the list).

So we need to find the average, over the n elements x in the table, of how many elements were inserted into x's list after x was inserted.

For $i = 1, 2, \ldots, n$, let x_i be the ith element inserted into the table, and let $k_i = key[x_i]$.

For all i and j, define indicator random variable $X_{ij} = \mathrm{I}\{h(k_i) = h(k_j)\}$.

Simple uniform hashing $\Rightarrow \Pr\{h(k_i) = h(k_j)\} = 1/m \Rightarrow \mathrm{E}[X_{ij}] = 1/m$

Successful search

Expected number of elements examined in a successful search is

$$
\begin{aligned}
& \mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}\left[X_{ij}\right]\right) && \text{(linearity of expectation)} \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&= 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) \\
&= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) && \text{(equation (A.1))} \\
&= 1+\frac{n-1}{2m} \\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}\,.
\end{aligned}
$$

Adding in the time for computing the hash function, we get that the expected total time for a successful search is $\Theta(2+\alpha/2-\alpha/2n)=\Theta(1+\alpha)$.

Successful search

Interpretation: If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1)$, which means that 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, all dictionary operations take $O(1)$ time on average.

Hash functions

What makes a good hash function?

- Ideally, the hash function satisfies the assumption of simple uniform hashing.
- In practice, it's not possible to satisfy this assumption, since we don't know in advance the probability distribution that keys are drawn from, and the keys may not be drawn independently.
- Often use heuristics, based on the domain of the keys, to create a hash function that performs well.

Keys as natural numbers

- Hash functions assume that the keys are natural numbers.
- When they're not, have to interpret them as natural numbers.
- ***Example:*** Interpret a character string as an integer expressed in some radix notation. Suppose the string is `CLRS`:
  - ASCII values: `C` = 67, `L` = 76, `R` = 82, `S` = 83.
  - There are 128 basic ASCII values.
  - So interpret `CLRS` as $(67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0) =$ 141,764,947.

Hash functions

Division method

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

Example: $m = 20$ and $k = 91 \Rightarrow h(k) = 11$.

Advantage: Fast, since requires just one division operation.

Disadvantage: Have to avoid certain values of m:

- Powers of 2 are bad. If $m = 2^p$ for integer p, then $h(k)$ is just the least significant p bits of k.
- If k is a character string interpreted in radix 2^p (as in `CLRS` example), then $m = 2^p - 1$ is bad: permuting characters in a string does not change its hash value (Exercise 11.3-3).

Good choice for m: A prime not too close to an exact power of 2.

Hash functions

Multiplication method

1. Choose constant A in the range $0 < A < 1$.
2. Multiply key k by A.
3. Extract the fractional part of kA.
4. Multiply the fractional part by m.
5. Take the floor of the result.

Put another way, $h(k) = \lfloor m\,(k\,A \bmod 1)\rfloor$, where $k\,A \bmod 1 = kA - \lfloor kA\rfloor =$ fractional part of kA.

Disadvantage: Slower than division method.

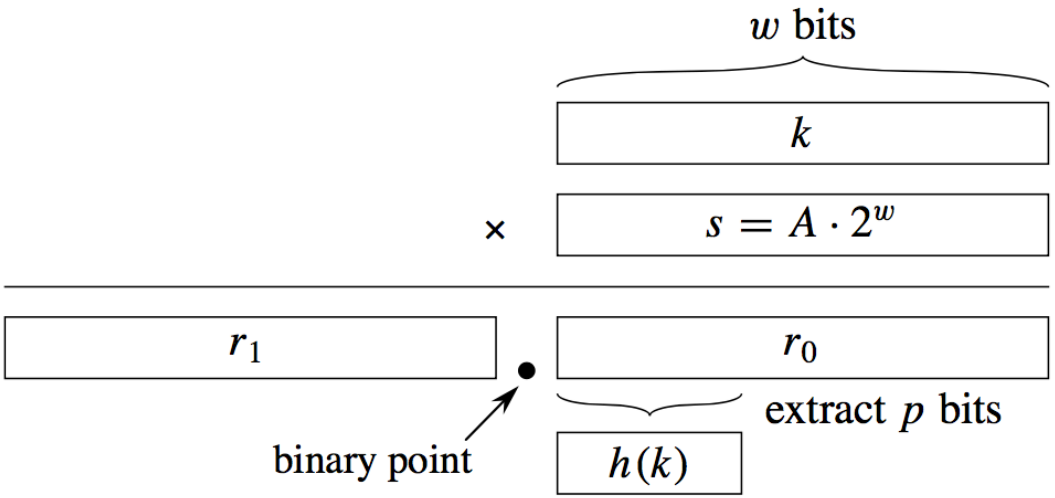
Advantage: Value of m is not critical.

(Relatively) easy implementation:

- Choose $m = 2^p$ for some integer p.
- Let the word size of the machine be w bits.
- Assume that k fits into a single word. (k takes w bits.)
- Let s be an integer in the range $0 < s < 2^w$. (s takes w bits.)

Hash functions

- Restrict A to be of the form $s/2^w$.



- Multiply k by s.
- Since we're multiplying two w-bit words, the result is $2w$ bits, $r_1 2^w + r_0$, where r_1 is the high-order word of the product and r_0 is the low-order word.
- r_1 holds the integer part of kA ($\lfloor kA \rfloor$) and r_0 holds the fractional part of kA ($k\,A \bmod 1 = kA - \lfloor kA \rfloor$). Think of the "binary point" (analog of decimal point, but for binary representation) as being between r_1 and r_0. Since we don't care about the integer part of kA, we can forget about r_1 and just use r_0.
- Since we want $\lfloor m\,(k\,A \bmod 1) \rfloor$, we could get that value by shifting r_0 to the left by $p = \lg m$ bits and then taking the p bits that were shifted to the left of the binary point.

Hash functions

- We don't need to shift. The p bits that would have been shifted to the left of the binary point are the p most significant bits of r_0. So we can just take these bits after having formed r_0 by multiplying k by s.
- ***Example:*** $m = 8$ (implies $p = 3$), $w = 5$, $k = 21$. Must have $0 < s < 2^5$; choose $s = 13 \Rightarrow A = 13/32$.
  - Using just the formula to compute $h(k)$: $kA = 21 \cdot 13/32 = 273/32 = 8\frac{17}{32}$ $\Rightarrow k\ A \bmod 1 = 17/32 \Rightarrow m\ (k\ A \bmod 1) = 8 \cdot 17/32 = 17/4 = 4\frac{1}{4} \Rightarrow \lfloor m\ (k\ A \bmod 1) \rfloor = 4$, so that $h(k) = 4$.
  - Using the implementation: $ks = 21 \cdot 13 = 273 = 8 \cdot 2^5 + 17 \Rightarrow r_1 = 8$, $r_0 = 17$. Written in $w = 5$ bits, $r_0 = 10001$. Take the $p = 3$ most significant bits of r_0, get 100 in binary, or 4 in decimal, so that $h(k) = 4$.

How to choose A:

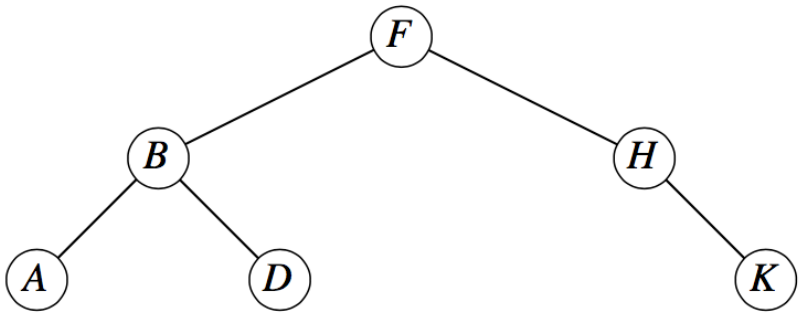
- The multiplication method works with any legal value of A.
- But it works better with some values than with others, depending on the keys being hashed.
- Knuth suggests using $A \approx (\sqrt{5} - 1)/2$.

Binary search trees (Chap 12)

Binary search trees are an important data structure for dynamic sets.

- Accomplish many dynamic-set operations in $O(h)$ time, where h = height of tree.
- As in Section 10.4, we represent a binary tree by a linked data structure in which each node is an object.
- $root[T]$ points to the root of tree T.
- Each node contains the fields
  - *key* (and possibly other satellite data).
  - *left*: points to left child.
  - *right*: points to right child.
  - p: points to parent. $p[root[T]] = \text{NIL}$.
- Stored keys must satisfy the ***binary-search-tree property***.
  - If y is in left subtree of x, then $key[y] \leq key[x]$.
  - If y is in right subtree of x, then $key[y] \geq key[x]$.

Binary search trees



The binary-search-tree property allows us to print keys in a binary search tree in order, recursively, using an algorithm called an ***inorder tree walk***. Elements are printed in monotonically increasing order.

How INORDER-TREE-WALK works:

- Check to make sure that x is not NIL.
- Recursively, print the keys of the nodes in x's left subtree.
- Print x's key.
- Recursively, print the keys of the nodes in x's right subtree.

Binary search trees

INORDER-TREE-WALK(x)
if $x \neq$ NIL
  then INORDER-TREE-WALK(*left*[x])
    print *key*[x]
    INORDER-TREE-WALK(*right*[x])

Example: Do the inorder tree walk on the example above, getting the output $A\,B\,D\,F\,H\,K$.

Correctness: Follows by induction directly from the binary-search-tree property.

Time: Intuitively, the walk takes $\Theta(n)$ time for a tree with n nodes, because we visit and print each node once. *[Book has formal proof.]*

Search on binary search trees

TREE-SEARCH(x, k)
if $x =$ NIL or $k = key[x]$
  then return x
if $k < key[x]$
  then return TREE-SEARCH$(left[x], k)$
  else return TREE-SEARCH$(right[x], k)$

Initial call is TREE-SEARCH$(root[T], k)$.

Example: Search for values D and C in the example tree from above.

Time: The algorithm recurses, visiting nodes on a downward path from the root. Thus, running time is $O(h)$, where h is the height of the tree.

Max and Min on binary search trees

The binary-search-tree property guarantees that

- the minimum key of a binary search tree is located at the leftmost node, and
- the maximum key of a binary search tree is located at the rightmost node.

Traverse the appropriate pointers (*left* or *right*) until NIL is reached.

TREE-MINIMUM(x)
while $left[x] \neq$ NIL
    do $x \leftarrow left[x]$
return x

TREE-MAXIMUM(x)
while $right[x] \neq$ NIL
    do $x \leftarrow right[x]$
return x

Time: Both procedures visit nodes that form a downward path from the root to a leaf. Both procedures run in $O(h)$ time, where h is the height of the tree.

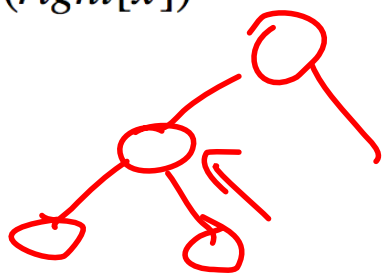
Successor and predecessor

There are two cases:

1. If node x has a non-empty right subtree, then x's successor is the minimum in x's right subtree.
2. If node x has an empty right subtree, notice that:
   - As long as we move to the left up the tree (move up through right children), we're visiting smaller keys.
   - x's successor y is the node that x is the predecessor of (x is the maximum in y's left subtree).

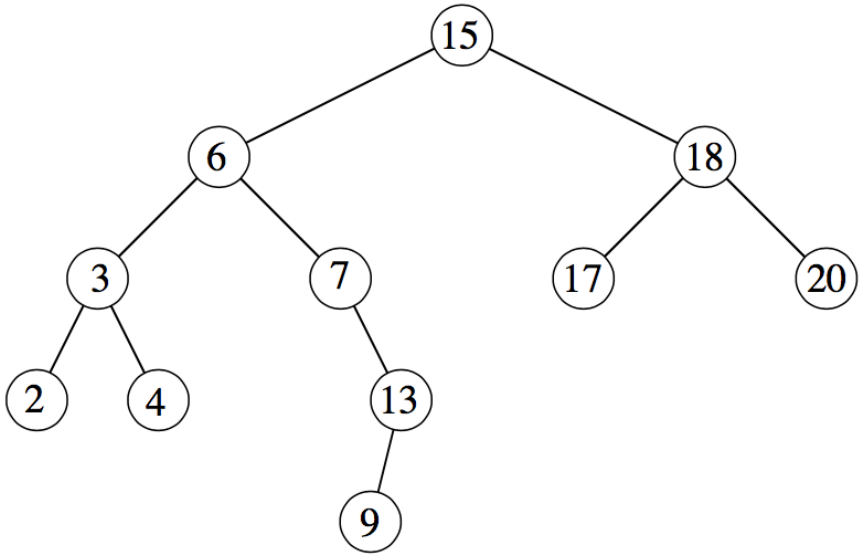
TREE-SUCCESSOR(x)

if $right[x] \neq$ NIL
  then return TREE-MINIMUM($right[x]$)
$y \leftarrow p[x]$
while $y \neq$ NIL and $x = right[y]$
  do $x \leftarrow y$
    $y \leftarrow p[y]$
return y (y=nil，没有)



TREE-PREDECESSOR is symmetric to TREE-SUCCESSOR.

Successor and predecessor



- Find the successor of the node with key value 15. (Answer: Key value 17)
- Find the successor of the node with key value 6. (Answer: Key value 7)
- Find the successor of the node with key value 4. (Answer: Key value 6)
- Find the predecessor of the node with key value 6. (Answer: Key value 4)

Time: For both the TREE-SUCCESSOR and TREE-PREDECESSOR procedures, in both cases, we visit nodes on a path down the tree or up the tree. Thus, running time is $O(h)$, where h is the height of the tree.

Insertion

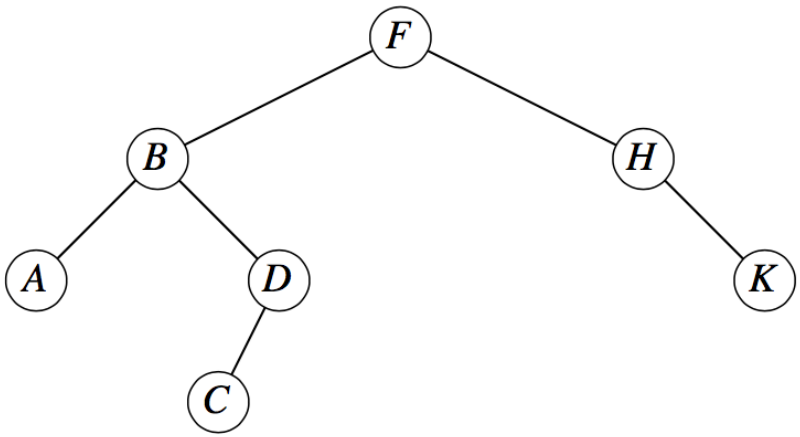
```
TREE-INSERT(T, z)
y ← NIL
x ← root[T]
while x ≠ NIL
      do y ← x
         if key[z] < key[x]
            then x ← left[x]
            else  x ← right[x]
p[z] ← y
if y = NIL
   then root[T] ← z                ▷ Tree T was empty
   else  if key[z] < key[y]
            then left[y] ← z
            else  right[y] ← z
```

- To insert value v into the binary search tree, the procedure is given node z, with $key[z] = v$, $left[z] = \text{NIL}$, and $right[z] = \text{NIL}$.
- Beginning at root of the tree, trace a downward path, maintaining two pointers.
  - Pointer x: traces the downward path.
  - Pointer y: "trailing pointer" to keep track of parent of x.

Insertion

- Traverse the tree downward by comparing the value of node at x with v, and move to the left or right child accordingly.
- When x is NIL, it is at the correct position for node z.
- Compare z's value with y's value, and insert z at either y's *left* or *right*, appropriately.

Example: Run TREE-INSERT(C) on the first sample binary search tree. Result:



Time: Same as TREE-SEARCH. On a tree of height h, procedure takes $O(h)$ time.

Deletion

TREE-DELETE is broken into three cases.

Case 1: z has no children.

- Delete z by making the parent of z point to NIL, instead of to z.

Case 2: z has one child.

- Delete z by making the parent of z point to z's child, instead of to z.

Case 3: z has two children.

- z's successor y has either no children or one child. (y is the minimum node—with no left child—in z's right subtree.)
- Delete y from the tree (via Case 1 or 2).
- Replace z's key and satellite data with y's.

Deletion

TREE-DELETE(T, z)

▷ Determine which node y to splice out: either z or z's successor.
if $left[z] = \text{NIL}$ or $right[z] = \text{NIL}$
  then $y \leftarrow z$
  else $y \leftarrow$ TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL if y has no children.
if $left[y] \neq \text{NIL}$
  then $x \leftarrow left[y]$
  else $x \leftarrow right[y]$
▷ y is removed from the tree by manipulating pointers of $p[y]$ and x.
if $x \neq \text{NIL}$
  then $p[x] \leftarrow p[y]$
if $p[y] = \text{NIL}$
  then $root[T] \leftarrow x$
  else if $y = left[p[y]]$
    then $left[p[y]] \leftarrow x$
    else $right[p[y]] \leftarrow x$
▷ If it was z's successor that was spliced out, copy its data into z.
if $y \neq z$
  then $key[z] \leftarrow key[y]$
    copy y's satellite data into z
return y

Deletion

Example: We can demonstrate on the above sample tree.

- For Case 1, delete K.
- For Case 2, delete H.
- For Case 3, delete B, swapping it with C.

Time: $O(h)$, on a tree of height h.

We've been analyzing running time in terms of h (the height of the binary search tree), instead of n (the number of nodes in the tree).

- Problem: Worst case for binary search tree is $\Theta(n)$—no better than linked list.
- Solution: Guarantee small height (balanced tree)—$h = O(\lg n)$.

In later chapters, by varying the properties of binary search trees, we will be able to analyze running time in terms of n.

- Method: Restructure the tree if necessary. Nothing special is required for querying, but there may be extra work when changing the structure of the tree (inserting or deleting).

Red-black trees are a special class of binary trees that avoids the worst-case behavior of $O(n)$ like "plain" binary search trees. Red-black trees are covered in detail in Chapter 13.

Red-black trees (Chap 13)

A ***red-black tree*** is a binary search tree + 1 bit per node: an attribute *color*, which is either red or black.

All leaves are empty (nil) and colored black.

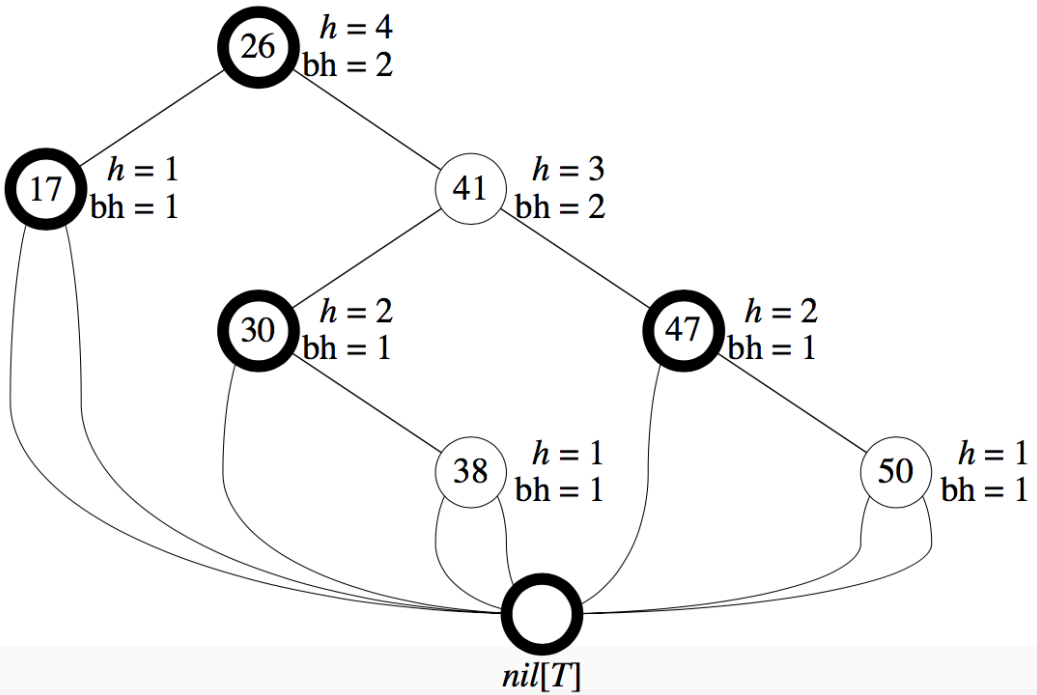
- We use a single sentinel, $nil[T]$, for all the leaves of red-black tree T.
- $color[nil[T]]$ is black.
- The root's parent is also $nil[T]$.

All other attributes of binary search trees are inherited by red-black trees (*key*, *left*, *right*, and p). We don't care about the key in $nil[T]$.

Red-black trees properties

1. Every node is either red or black.
2. The root is black.
3. Every leaf ($nil[T]$) is black.
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Example:



Height of red-black trees

- ***Height of a node*** is the number of edges in a longest path to a leaf.
- ***Black-height*** of a node x: $\mathrm{bh}(x)$ is the number of black nodes (including $nil[T]$) on the path from x to leaf, not counting x. By property 5, black-height is well defined.

[Now label the example tree with height h and bh *values.]*

Claim

Any node with height h has black-height $\geq h/2$.

Proof By property 4, $\leq h/2$ nodes on the path from the node to a leaf are red. Hence $\geq h/2$ are black. ■ (claim)

Height of red-black trees

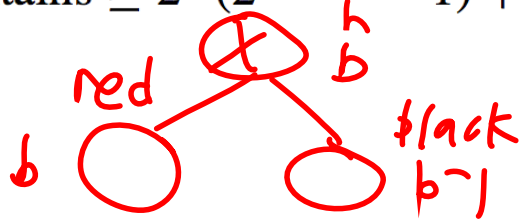
Claim

The subtree rooted at any node x contains $\geq 2^{\text{bh}(x)} - 1$ internal nodes.

Proof By induction on height of x.

Basis: Height of $x = 0 \Rightarrow x$ is a leaf $\Rightarrow \text{bh}(x) = 0$. The subtree rooted at x has 0 internal nodes. $2^0 - 1 = 0$.

Inductive step: Let the height of x be h and $\text{bh}(x) = b$. Any child of x has height $h - 1$ and black-height either b (if the child is red) or $b - 1$ (if the child is black). By the inductive hypothesis, each child has $\geq 2^{\text{bh}(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains $\geq 2 \cdot (2^{\text{bh}(x)-1} - 1) + 1 = 2^{\text{bh}(x)} - 1$ internal nodes. (The $+1$ is for x itself.) ■ (claim)



Lemma

A red-black tree with n internal nodes has height $\leq 2\lg(n+1)$.

$2^{b-1} - 1$

Proof Let h and b be the height and black-height of the root, respectively. By the above two claims,

$n \geq 2^b - 1 \geq 2^{h/2} - 1$. $(b \geq h/2)$

Adding 1 to both sides and then taking logs gives $\lg(n+1) \geq h/2$, which implies that $h \leq 2\lg(n+1)$. ■ (theorem)

Operations on red-black trees

The non-modifying binary-search-tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in $O(\text{height})$ time. Thus, they take $O(\lg n)$ time on red-black trees.

Insertion and deletion are not so easy.

If we insert, what color to make the new node?

Color the new node in red and fix 4

- Red? Might violate property 4. / 2 if empty
- Black? Might violate property 5.

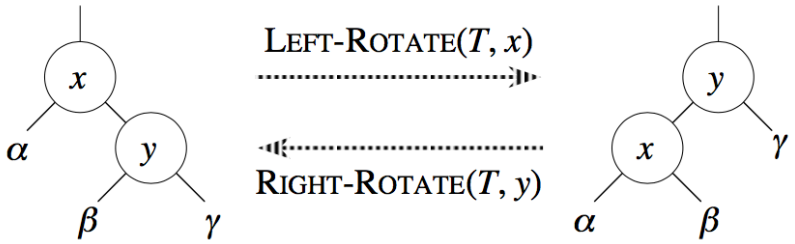
If we delete, thus removing a node, what color was the node that was removed?

- Red? OK, since we won't have changed any black-heights, nor will we have created two red nodes in a row. Also, cannot cause a violation of property 2, since if the removed node was red, it could not have been the root.
- Black? Could cause there to be two reds in a row (violating property 4), and can also cause a violation of property 5. Could also cause a violation of property 2, if the removed node was the root and its child—which becomes the new root—was red.

Rotation

- The basic tree-restructuring operation.
- Needed to maintain red-black trees as balanced binary search trees.
- Changes the local pointer structure. (Only pointers are changed.)
- Won't upset the binary-search-tree property.
- Have both left rotation and right rotation. They are inverses of each other.
- A rotation takes a red-black-tree and a node within the tree.

Left rotation



LEFT-ROTATE(T, x)

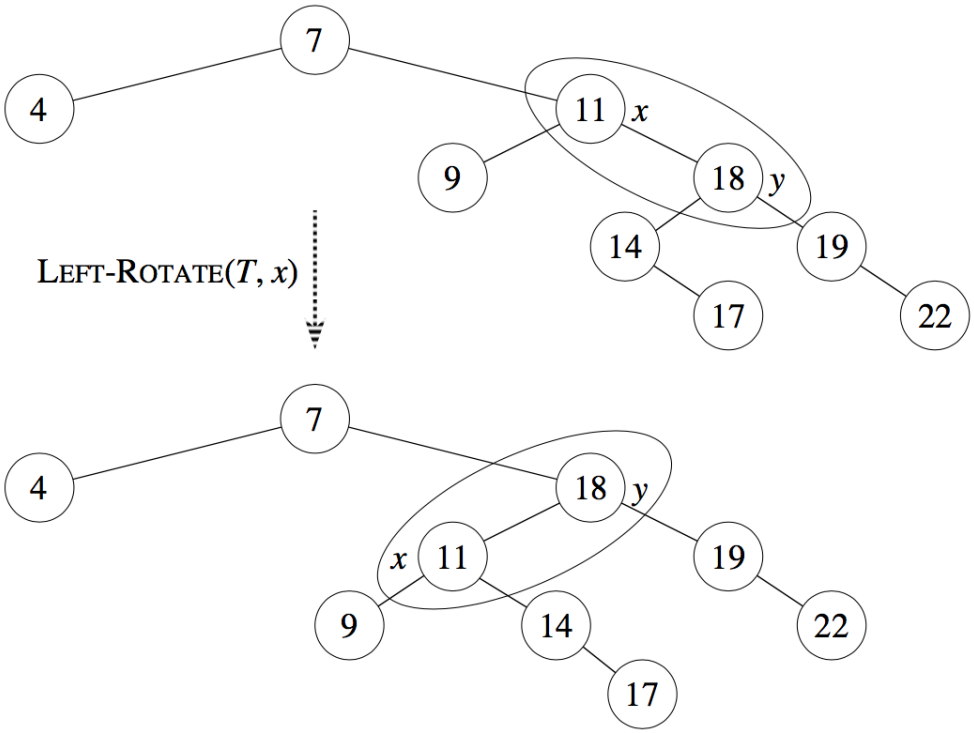
$y \leftarrow right[x]$ ▷ Set y.
$right[x] \leftarrow left[y]$ ▷ Turn y's left subtree into x's right subtree.
if $left[y] \neq nil[T]$
  then $p[left[y]] \leftarrow x$
$p[y] \leftarrow p[x]$ ▷ Link x's parent to y.
if $p[x] = nil[T]$
  then $root[T] \leftarrow y$
  else if $x = left[p[x]]$
    then $left[p[x]] \leftarrow y$
    else $right[p[x]] \leftarrow y$
$left[y] \leftarrow x$ ▷ Put x on y's left.
$p[x] \leftarrow y$

The pseudocode for LEFT-ROTATE assumes that

- $right[x] \neq nil[T]$, and
- root's parent is $nil[T]$.

Pseudocode for RIGHT-ROTATE is symmetric: exchange *left* and *right* everywhere.

Example



- Before rotation: keys of x's left subtree $\leq 11 \leq$ keys of y's left subtree $\leq 18 \leq$ keys of y's right subtree.
- Rotation makes y's left subtree into x's right subtree.
- After rotation: keys of x's left subtree $\leq 11 \leq$ keys of x's right subtree $\leq 18 \leq$ keys of y's right subtree.

Time: $O(1)$ for both LEFT-ROTATE and RIGHT-ROTATE, since a constant number of pointers are modified.

RB insertion

RB-INSERT(T, z)

```
y ← nil[T]
x ← root[T]
while x ≠ nil[T]
      do y ← x
         if key[z] < key[x]
           then x ← left[x]
           else  x ← right[x]
p[z] ← y
if y = nil[T]
   then root[T] ← z
   else  if key[z] < key[y]
           then left[y] ← z
           else  right[y] ← z
left[z] ← nil[T]
right[z] ← nil[T]
color[z] ← RED
```

RB-INSERT-FIXUP(T, z)

- RB-INSERT ends by coloring the new node z red.
- Then it calls RB-INSERT-FIXUP because we could have violated a red-black property.

RB insertion fixup

Which property might be violated?

1. OK.
2. If z is the root, then there's a violation. Otherwise, OK.
3. OK.
4. If $p[z]$ is red, there's a violation: both z and $p[z]$ are red.
5. OK.

Remove the violation by calling RB-INSERT-FIXUP:

RB-INSERT-FIXUP(T, z)
while $color[p[z]] = \text{RED}$
  do if $p[z] = left[p[p[z]]]$
    then $y \leftarrow right[p[p[z]]]$
      if $color[y] = \text{RED}$
        then $color[p[z]] \leftarrow \text{BLACK}$ ▷ Case 1
          $color[y] \leftarrow \text{BLACK}$ ▷ Case 1
          $color[p[p[z]]] \leftarrow \text{RED}$ ▷ Case 1
          $z \leftarrow p[p[z]]$ ▷ Case 1
        else if $z = right[p[z]]$
          then $z \leftarrow p[z]$ ▷ Case 2
            LEFT-ROTATE(T, z) ▷ Case 2
          $color[p[z]] \leftarrow \text{BLACK}$ ▷ Case 3
          $color[p[p[z]]] \leftarrow \text{RED}$ ▷ Case 3
          RIGHT-ROTATE$(T, p[p[z]])$ ▷ Case 3
    else (same as **then** clause
      with "right" and "left" exchanged)
$color[root[T]] \leftarrow \text{BLACK}$

Correctness

Loop invariant:

At the start of each iteration of the **while** loop,

a. z is red.

b. There is at most one red-black violation:

- Property 2: z is a red root, or
- Property 4: z and $p[z]$ are both red.

[The book has a third part of the loop invariant, but we omit it for lecture.]

Initialization: We've already seen why the loop invariant holds initially.

Termination: The loop terminates because $p[z]$ is black. Hence, property 4 is OK. Only property 2 might be violated, and the last line fixes it.

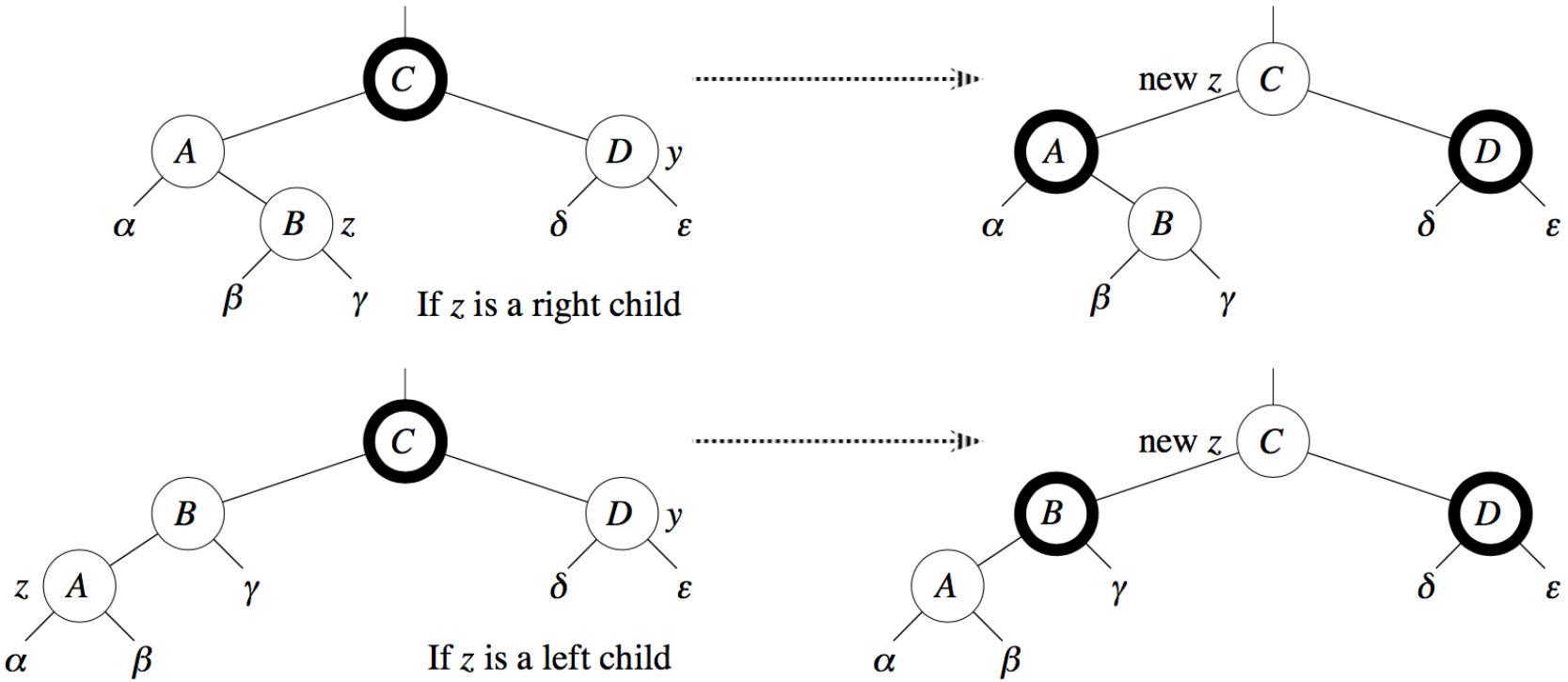
Maintenance: We drop out when z is the root (since then $p[z]$ is the sentinel $nil[T]$, which is black). When we start the loop body, the only violation is of property 4.

There are 6 cases, 3 of which are symmetric to the other 3. The cases are not mutually exclusive. We'll consider cases in which $p[z]$ is a left child.

Let y be z's uncle ($p[z]$'s sibling).

Case 1

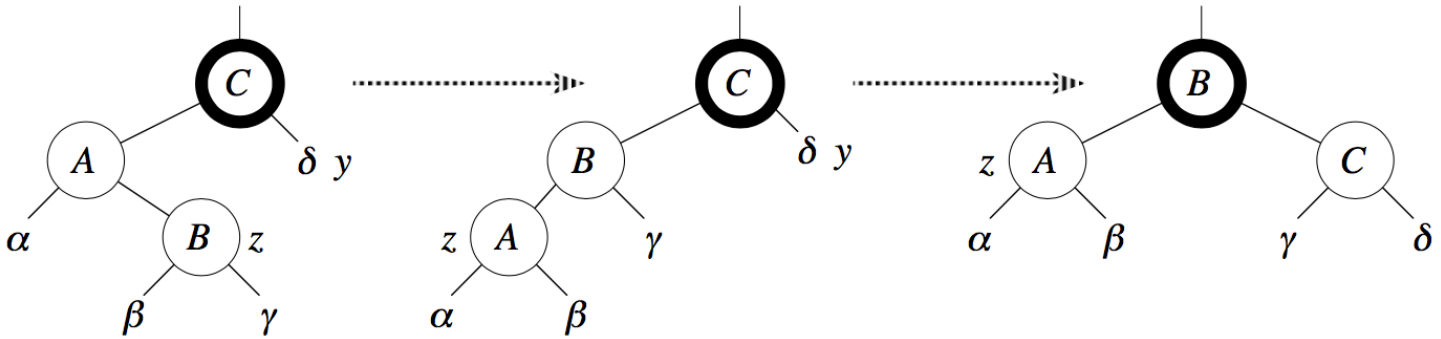
Case 1: y is red



- $p[p[z]]$ (z's grandparent) must be black, since z and $p[z]$ are both red and there are no other violations of property 4.
- Make $p[z]$ and y black $\Rightarrow$ now z and $p[z]$ are not both red. But property 5 might now be violated.
- Make $p[p[z]]$ red $\Rightarrow$ restores property 5.
- The next iteration has $p[p[z]]$ as the new z (i.e., z moves up 2 levels).

Case 2 and Case 3

Case 2: y is black, z is a right child



Case 2 Case 3

- Left rotate around $p[z] \Rightarrow$ now z is a left child, and both z and $p[z]$ are red.
- Takes us immediately to case 3.

Case 3: y is black, z is a left child

- Make $p[z]$ black and $p[p[z]]$ red.
- Then right rotate on $p[p[z]]$.
- No longer have 2 reds in a row.
- $p[z]$ is now black $\Rightarrow$ no more iterations.

Time analysis

Analysis

$O(\lg n)$ time to get through RB-INSERT up to the call of RB-INSERT-FIXUP.

Within RB-INSERT-FIXUP:

- Each iteration takes $O(1)$ time.
- Each iteration is either the last one or it moves z up 2 levels.
- $O(\lg n)$ levels $\Rightarrow O(\lg n)$ time.
- Also note that there are at most 2 rotations overall.

Thus, insertion into a red-black tree takes $O(\lg n)$ time.

RB deletion

RB-DELETE(T, z)

if $left[z] = nil[T]$ or $right[z] = nil[T]$
  then $y \leftarrow z$
  else $y \leftarrow$ TREE-SUCCESSOR(z)
if $left[y] \neq nil[T]$
  then $x \leftarrow left[y]$
  else $x \leftarrow right[y]$
$p[x] \leftarrow p[y]$
if $p[y] = nil[T]$
  then $root[T] \leftarrow x$
  else if $y = left[p[y]]$
    then $left[p[y]] \leftarrow x$
    else $right[p[y]] \leftarrow x$
if $y \neq z$
  then $key[z] \leftarrow key[y]$
    copy y's satellite data into z
if $color[y] =$ BLACK
  then RB-DELETE-FIXUP(T, x)
return y

- y is the node that was actually spliced out.
- x is either
  - y's sole non-sentinel child before y was spliced out, or
  - the sentinel, if y had no children.

RB deletion

If y is black, we could have violations of red-black properties:

1. OK.
2. If y is the root and x is red, then the root has become red.
3. OK.
4. Violation if $p[y]$ and x are both red.
5. Any path containing y now has 1 fewer black node.

- Correct by giving x an "extra black."
- Add 1 to count of black nodes on paths containing x.
- Now property 5 is OK, but property 1 is not.
- x is either ***doubly black*** (if $color[x] = \text{BLACK}$) or ***red & black*** (if $color[x] = \text{RED}$).
- The attribute $color[x]$ is still either RED or BLACK. No new values for $color$ attribute.
- In other words, the extra blackness on a node is by virtue of x pointing to the node.

RB deletion fixup

```
RB-DELETE-FIXUP(T, x)
while x ≠ root[T] and color[x] = BLACK
    do if x = left[p[x]]
        then w ← right[p[x]]
            if color[w] = RED
              then color[w] ← BLACK                          ▷ Case 1
                   color[p[x]] ← RED                         ▷ Case 1
                   LEFT-ROTATE(T, p[x])                      ▷ Case 1
                   w ← right[p[x]]                           ▷ Case 1
            if color[left[w]] = BLACK and color[right[w]] = BLACK
              then color[w] ← RED                            ▷ Case 2
                   x ← p[x]                                  ▷ Case 2
              else if color[right[w]] = BLACK
                     then color[left[w]] ← BLACK             ▷ Case 3
                          color[w] ← RED                     ▷ Case 3
                          RIGHT-ROTATE(T, w)                 ▷ Case 3
                          w ← right[p[x]]                    ▷ Case 3
                   color[w] ← color[p[x]]                    ▷ Case 4
                   color[p[x]] ← BLACK                       ▷ Case 4
                   color[right[w]] ← BLACK                   ▷ Case 4
                   LEFT-ROTATE(T, p[x])                      ▷ Case 4
                   x ← root[T]                               ▷ Case 4
        else (same as then clause with "right" and "left" exchanged)
color[x] ← BLACK
```

RB deletion fixup

Idea: Move the extra black up the tree until

- x points to a red & black node $\Rightarrow$ turn it into a black node,
- x points to the root $\Rightarrow$ just remove the extra black, or
- we can do certain rotations and recolorings and finish.

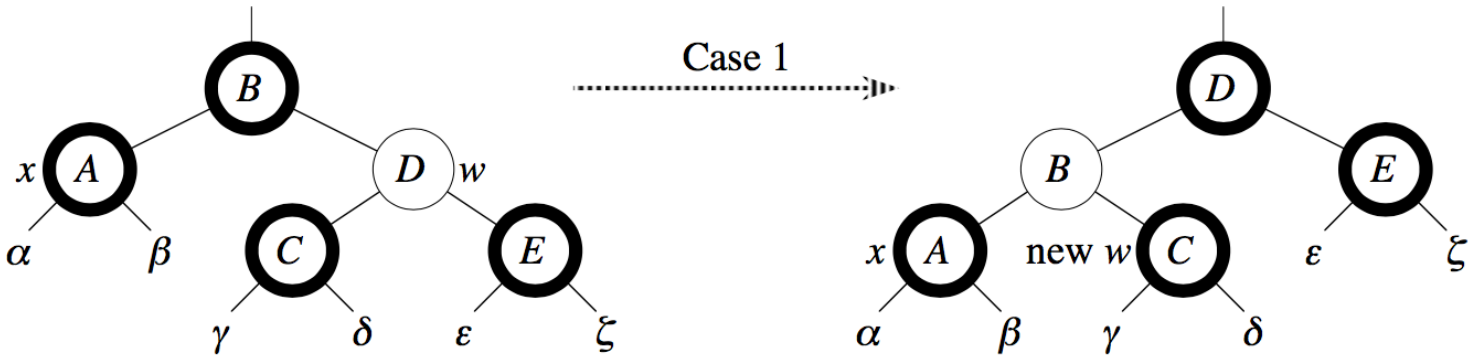
Within the **while** loop:

- x always points to a nonroot doubly black node.
- w is x's sibling.
- w cannot be $nil[T]$, since that would violate property 5 at $p[x]$.

There are 8 cases, 4 of which are symmetric to the other 4. As with insertion, the cases are not mutually exclusive. We'll look at cases in which x is a left child.

Case 1

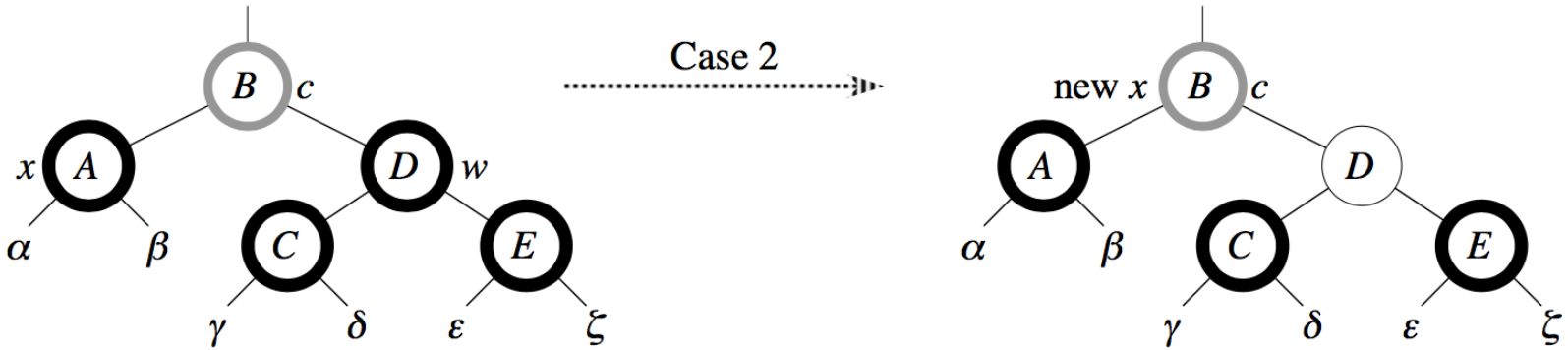
Case 1: w is red



- w must have black children.
- Make w black and $p[x]$ red.
- Then left rotate on $p[x]$.
- New sibling of x was a child of w before rotation $\Rightarrow$ must be black.
- Go immediately to case 2, 3, or 4.

Case 2

Case 2: w is black and both of w's children are black

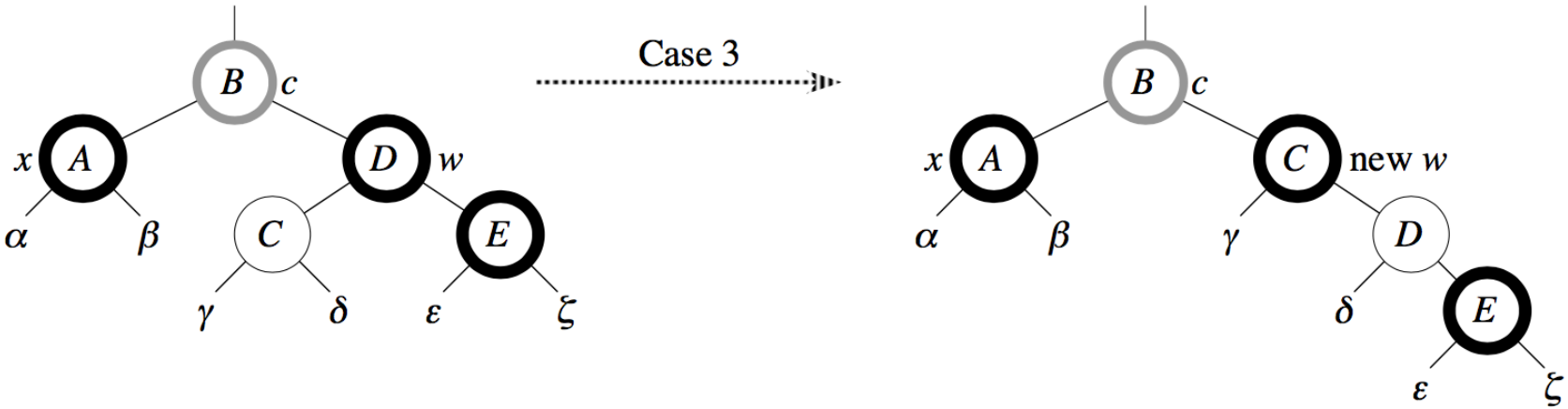


[Node with gray outline is of unknown color, denoted by c.]

- Take 1 black off x ($\Rightarrow$ singly black) and off w ($\Rightarrow$ red).
- Move that black to $p[x]$.
- Do the next iteration with $p[x]$ as the new x.
- If entered this case from case 1, then $p[x]$ was red $\Rightarrow$ new x is red & black $\Rightarrow$ color attribute of new x is RED $\Rightarrow$ loop terminates. Then new x is made black in the last line.

Case 3

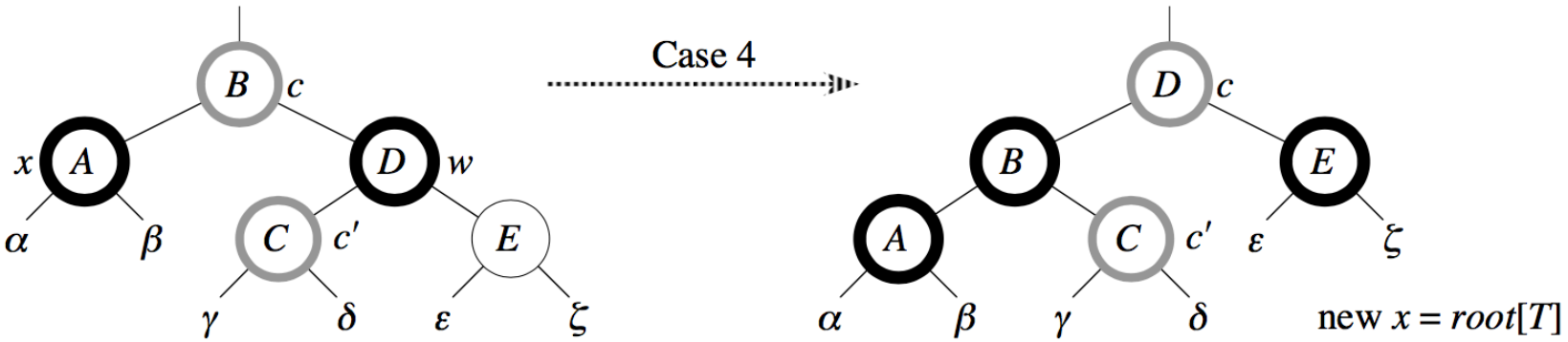
Case 3: w is black, w's left child is red, and w's right child is black



- Make w red and w's left child black.
- Then right rotate on w.
- New sibling w of x is black with a red right child $\Rightarrow$ case 4.

Case 4

Case 4: w is black, w's left child is black, and w's right child is red



[Now there are two nodes of unknown colors, denoted by c and c'.]

- Make w be $p[x]$'s color (c).
- Make $p[x]$ black and w's right child black.
- Then left rotate on $p[x]$.
- Remove the extra black on x ($\Rightarrow$ x is now singly black) without violating any red-black properties.
- All done. Setting x to root causes the loop to terminate.

Time analysis

Analysis

$O(\lg n)$ time to get through RB-DELETE up to the call of RB-DELETE-FIXUP.

Within RB-DELETE-FIXUP:

- Case 2 is the only case in which more iterations occur.
  - x moves up 1 level.
  - Hence, $O(\lg n)$ iterations.
- Each of cases 1, 3, and 4 has 1 rotation $\Rightarrow \leq 3$ rotations in all.
- Hence, $O(\lg n)$ time.