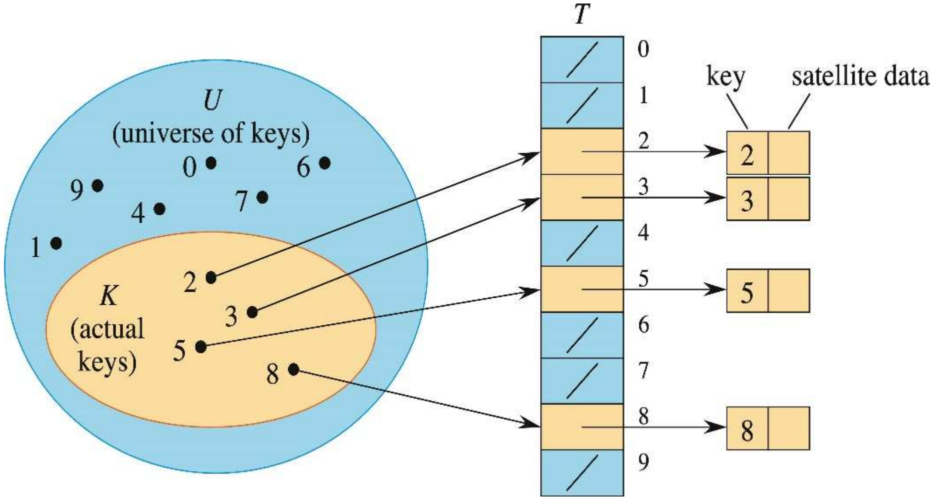
Part II

Data Structures

§ Three Types of Data Structures Considered:

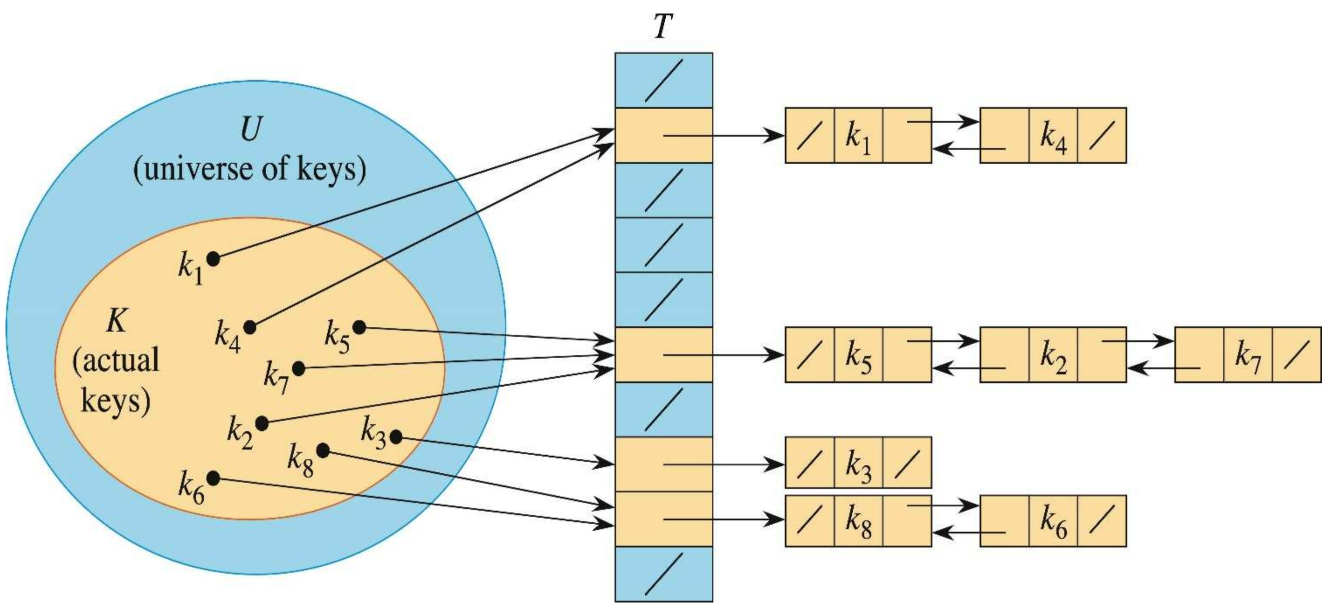
- Hash Tables
- Trees
- Heaps

- **Hash Table**
  - – suitable for large key space with small numbers of actual keys stored
  - – average search time over the hash table equal to $O(1)$
  - – collision-handling strategy required
  - – keys must be stored to ensure right items retrieved
  - – hash function maps a key to one table index



Hash Tables (continued)

- **Hash Table with Collision Resolution by Chaining**
  - multiple elements with different keys mapped to the same table entry (collisions)
  - they can be chained, with a lookup going through the chain
  - for load factor of $\alpha = n/m$, where n (or m) is the number of elements (or table entries)
  - under uniform distribution, one search (be successful or not) takes $\Theta(1+\alpha)$, according to Theorems 11.1 and 11.2
  - if hash table size (m) grows in proportion to n, then the mean search time is $\Theta(1)$
  - there are other more efficient ways for handling collisions



Hash Tables (continued)

- **Hash Functions**
  - division method
  - multiplication method

1. Division method: (more restrictive on the suitable m)

$h(k) = k \bmod m$, where the choice of $m = 2^p - 1$ is poor but a *prime* not close to an exact power of 2 is often good

For example, given $n = 2000$ character strings to be stored in a hash table with $m = 701$ entries, then an unsuccessful search takes some 3 accesses

<u>Hash Tables</u> (continued)

2. Multiplication method:

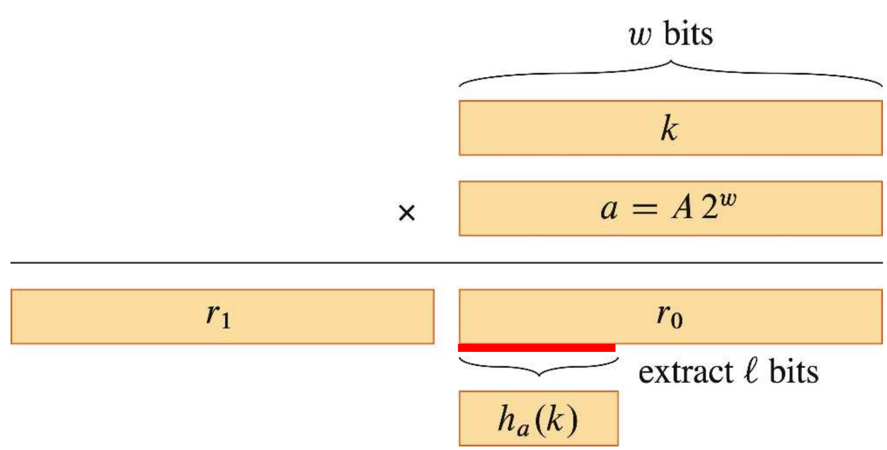
○ Multiply k by a constant A, with $\underline{0 < A < 1}$ and extract the <u>fraction part</u> of $k \cdot A$, e.g., $\underline{h(k) = \lfloor m\ (kA \mod 1) \rfloor}$, where "$kA \mod 1$" equals $kA - \lfloor kA \rfloor$

– table size (m) can be arbitrary, e.g., $m = 2^p$
– let A be a fraction of the form $s/2^w$ as shown in the figure below
– perform multiplication of w-bit k and w-bit s, to get a $2w$-bit product
– product denoted by $r_1 \cdot 2^w + r_0$, then p-bit hash value is obtained from r_0

– while arbitrary A works, $A \approx (\sqrt{5} - 1)/2 = 0.6180339887 \ldots$ recommended

Golden ratio: a root of x^2 + x -1 = 0.

\+ Let $k = 123456$, $p = 14$, $m = 2^{14}$, $w = 32$ and $A = 2654435769/2^{32}$, we have $k \cdot s = (76300 \cdot 2^{32}) + 17612864$ to get $h(k) = 67$.



Hash Tables (continued)

- **Properties of Random Hashing**
  - family of hash functions, $\mathcal{H}$, with domain U and range $R = \{0, 1, \ldots, m-1\}$
  - $\mathcal{H}$ is *uniform* if probability of hashing a key k, $h(k)$, to q in R equals $1/m$
  - $\mathcal{H}$ is *universal* if probability of distinct keys, k_1 and k_2, with $h(k_1) = h(k_2)$ is $\leq 1/m$
  - $\mathcal{H}$ is ε-*universal* if probability of distinct keys, k_1 and k_2, with $h(k_1) = h(k_2)$ is $\leq 1/\varepsilon$
  - $\mathcal{H}$ is *d-independent* if distinct keys, $k_1, k_2, \ldots, k_d$, and slots of $q_1, q_2, \ldots, q_d$ in R, with $h(k_i) = h(q_i)$ for all $1 \leq i \leq d$, has the probability of $1/m^d$

Hash Tables (continued)

- **Universal Hashing**

  For the hash function of $\boldsymbol{h_{ab}(k) = ((a \cdot k + b) \bmod p) \bmod m}$, where p is a large prime number, with $p > m$, $a \in \{1, 2, \ldots p\text{-}1\}$ and $b \in \{0, 1, 2, p\text{-}1\}$, the collection of such hash functions is *universal*.

In this case, m can be any number and does not have to be a prime.
For example, for $p = 17$ and $m = 8$, we have $h_{34}(15) = 7$.

((3*15+4) mod 17) mod 8 = 7

Hash Tables (continued)

- **Open Addressing** (to deal with collision-chaining)
  - elements stored inside the table
  - calculating probe sequence of given key, instead of using pointers:
    $\langle h(k, 0), h(k, 1), h(k, 2), \ldots, h(k, m-1) \rangle$
  - if probe sequence is a permutation of $\langle 0, 1, 2, \ldots, m-1 \rangle$, every table entry is a candidate location for the element
  - the probe sequence is fixed for a given key
  - key has to be stored in the table entry

```
HASH-INSERT(T, k)
  i = 0
  repeat
      j = h(k, i)
      if T[j] == NIL
          T[j] = k
          return j
      else i = i + 1
  until i == m
  error "hash table overflow"
```

```
HASH-SEARCH(T, k)
  i = 0
  repeat
      j = h(k, i)
      if T[j] == k
          return j
      i = i + 1
  until T[j] == NIL or i = m
  return NIL
```

<u>Hash Tables</u> (continued)

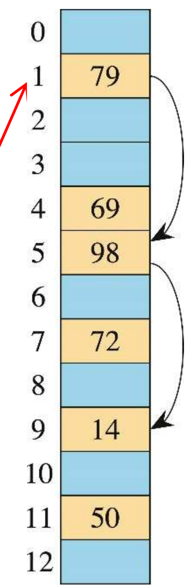
- **Uniform Hashing Analysis on Open Addressing**
  - key probe sequence equal likely in one of $m!$ permutations of $\langle 0, 1, 2, \ldots, m-1\rangle$
  - probe sequences defined for open addressing
    - + <u>linear probing</u>: m different sequences
      - ○ dictated by $h(k, i) = (h'(k) + i) \bmod m$
      - ○ subject to *primary clustering*
    - + <u>quadratic probing</u>: m different sequences
      - ○ dictated by $h(k, i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$
      - ○ subject to *secondary clustering*, i.e., $h(k_1, 0) = h(k_2, 0) \Longrightarrow$ both keys follow the <u>same</u> probe sequence
    - + <u>double hashing</u>: m^2 different sequences
      - ○ dictated by $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$
      - ○ many good choices: $h_2(k)$ <u>relatively prime</u> to m; m itself a prime; m a power of 2 and $h_2(k)$ always an odd number; or below:

        m a prime and m' slightly less than m (e.g., m-1 or m-2) with

        $h_1(k) = k \bmod m$

        $h_2(k) = 1 + (k \bmod m')$

e.g., for k =14 under m = 13, m' = 11; upon inserting k = 14.



Hash Tables (continued)

• Open-Address Hashing Analysis

Theorem 1:

For an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in unsuccessful search under uniform hashing is at most $1/(1 - \alpha)$.

Note: unsuccessful search ends at seeing an empty entry.

.

+ if α is a constant, an unsuccessful search runs in *O*(1) time
+ inserting an element into an open-address hash table with load factor α takes at most $1/(1 - \alpha)$ probes on an average under uniform hashing, because it has (1-α) for 1 probe, plus prob. α(1-α) to take 2 probes, plus prob. $\alpha^2(1-\alpha)$ to take 3 probes, etc., yielding $1+\alpha+\alpha^2+\alpha^3+ \ldots = 1/(1 - \alpha)$

e.g., for α = 50/100, we have 2 probes.

Theorem 2:

For an open-address hash table with load factor α < 1, the expected number of probes in a successful search under uniform hashing is at most $\frac{1}{\alpha} \cdot \ln\frac{1}{1-\alpha}$.

► successful search for a key equals the sequence of inserting the key

2nd key insertion takes ≤ $1/(1-(1/m))$ probes, on an average, when α is $1/m$

3rd key takes ≤ $1/(1-(2/m))$ probes, on an average, when α is $2/m$

:

i+1th key takes ≤ $1/(1-(i/m))$ probes

Mean no. of probes equals results over all *n* keys inserted

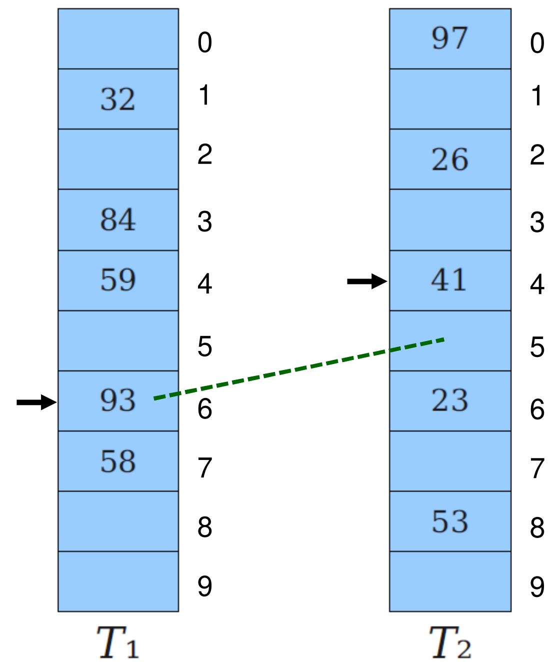
This is because **insertion and probes** follow the same hash function.

We have: $1/n\,(\sum_{i=0}^{n-1} 1/(1-i/m))$

e.g., for α = 50/100, we have ≈ 1.39 probes, as ln(2) ~ 0.693. (base e ~ 2.718)

Hash Tables (continued)

- **Cuckoo Hashing**[§]
  - move keys around upon insertion for worst probe case of $O(1)$
    - + quick search but possibly lengthy insertion
    - + multiple hash functions required for separate hash tables
  - delete keys in the worst-case of $O(1)$
  - example with two hash functions below:
    - + two hash tables, each with m elements
    - + two hash functions, h_1 and h_2, one for a table
    - + each key x at either $h_1(x)$ or $h_2(x)$
    - + new key '10' with $h_1(10) = 6$ and $h_2(10) = 4$
      - * move key '93' around, if $h_2(93)=5$, for insertion



  - Algorithm
    1. insert "new key" to T_1 located at h_1(new key), when available
    2. otherwise, "new key" displaces "existing key" in T_1; place "existing key" in T_2 located at h_2(existing key); repeat the displacement process, if needed.

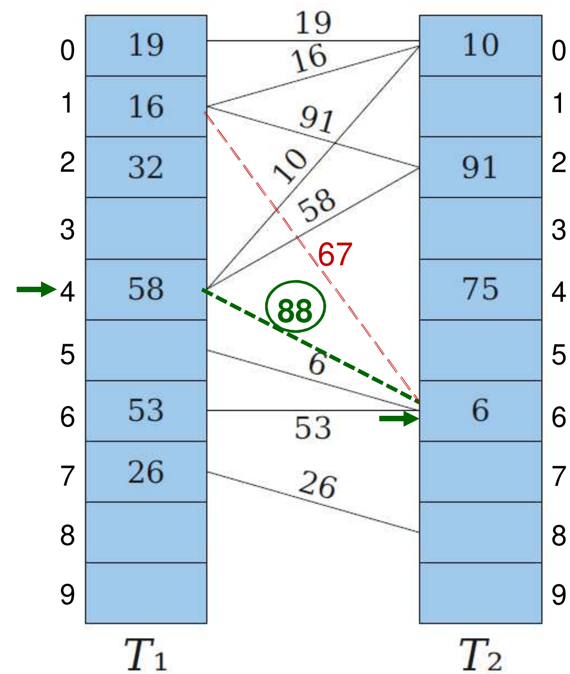
§ R. Pagh and F. Rodler, "Cuckoo Hashing," *Journal of Algorithms*, Aug. 2001, pp. 121-133.

<u>Hash Tables</u> (continued)

- **Cuckoo Graph**
  – derived from cuckoo hash tables
    + each <u>table entry</u> is a <u>node</u>
    + each <u>key</u> is an <u>edge</u>, which links the entries that can hold the key
    + an insertion adds a new edge to the graph
      * let h_1(88) = 4 and h_2(88) = 6
      * displace key '58' in T_1 to make room for key '88'
      * displace key '91' in T_2 to make room for key '58'
      * displace key '16' in T_1 to make room for key '91'
      * displace key '10' in T_2 to make room for key '16'
      * repeat ….

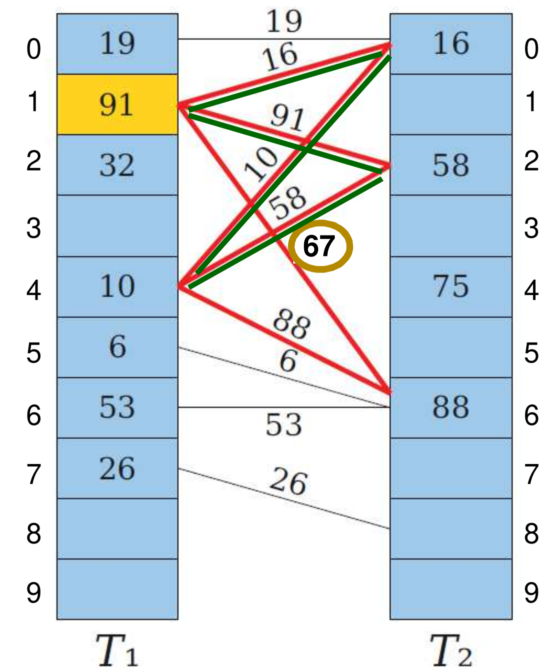
    + insertion done by tracing a path over the graph

    + what about inserting <u>next</u> key "67" with
      h_1(67) = 1 and h_2(67) = 6?



Hash Tables (continued)

- **Cuckoo Graph**
  – insertion succeeds if and only if the new edge (defined by the new key) is on at most one cycle



Proof sketch:

1. each edge denotes a key and requires a table entry (i.e., node) to hold it
2. for a cycle exists, the number of its edges equals the number of its nodes involved, so that all nodes (table entries) are taken
3. if a new edge (after added) is on two cycles, the edge must link two taken nodes AND all nodes on the two cycles have been used
4. the new key (edge) *cannot* be accommodated.

Side note: two cycles (one with 4 distinct nodes and the other with at least one distinct node) that share one edge, will include (at least) 5 distinct nodes but they contain (at least) 6 distinct edges; impossible.

Binary Search Trees

§ Binary Search Tree Property

+ Given tree node *x*, if node *y* is in the left (or right) subtree of *x*, we have *y.key* ≤ *x.key* (or *y.key* ≥ *x.key*).

Theorem 1

INORDER-TREE-WALK(x) across an n-node subtree rooted at x takes $\Theta(n)$.

(because the tree height may equal n)

Proof.

The tree walk has to visit every node, and hence, the time complexity $T(n)$ is lower bounded by $\Omega(n)$.

Next, considering that INORDER-TREE-WALK(x) is called on x whose left and right subtrees have k and $(n\text{-}k\text{-}1)$ nodes, respectively, we have

$T(n) \le T(k) + T(n\text{-}k\text{-}1) + d$ for some $d > 0$.

For simplicity, we may let *d* equal 1 here.

By the substitution method, we prove that $T(n) \le (c+d)n + c$, for a constant c, below:

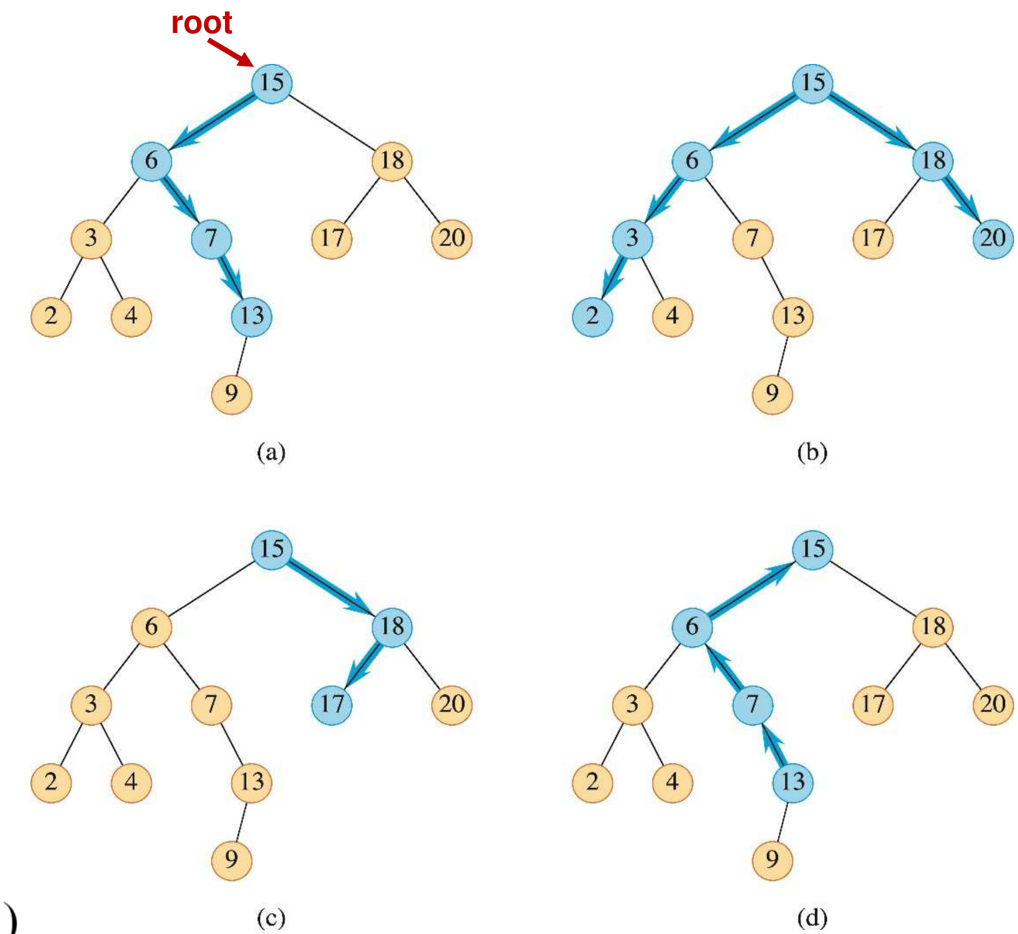
For simplicity, we may let *c* equal 1 here to have $T(n) \le 2 \cdot n + 1$.

$$
\begin{aligned}
T(n) &\le T(k) + T(n\text{-}k\text{-}1) + d \\
&\le ((c+d)k + c) + ((c+d)(n\text{-}k\text{-}1) + c) + d \\
&= (c+d)n + c - (c+d) + c + d \\
&= \underline{(c+d)n + c} \quad \text{(upper bounded)}
\end{aligned}
$$

Binary Search Trees (continued)

- **Querying binary search trees**
  - + searching
  - + successor and predecessor
  - + insertion and deletion

  + Searching via TREE-SEARCH
    in time complexity $O(h)$



TREE-SEARCH(x, k)

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

search path from root to *key* = 13 is
$15 \rightarrow 6 \rightarrow 7 \rightarrow 13$

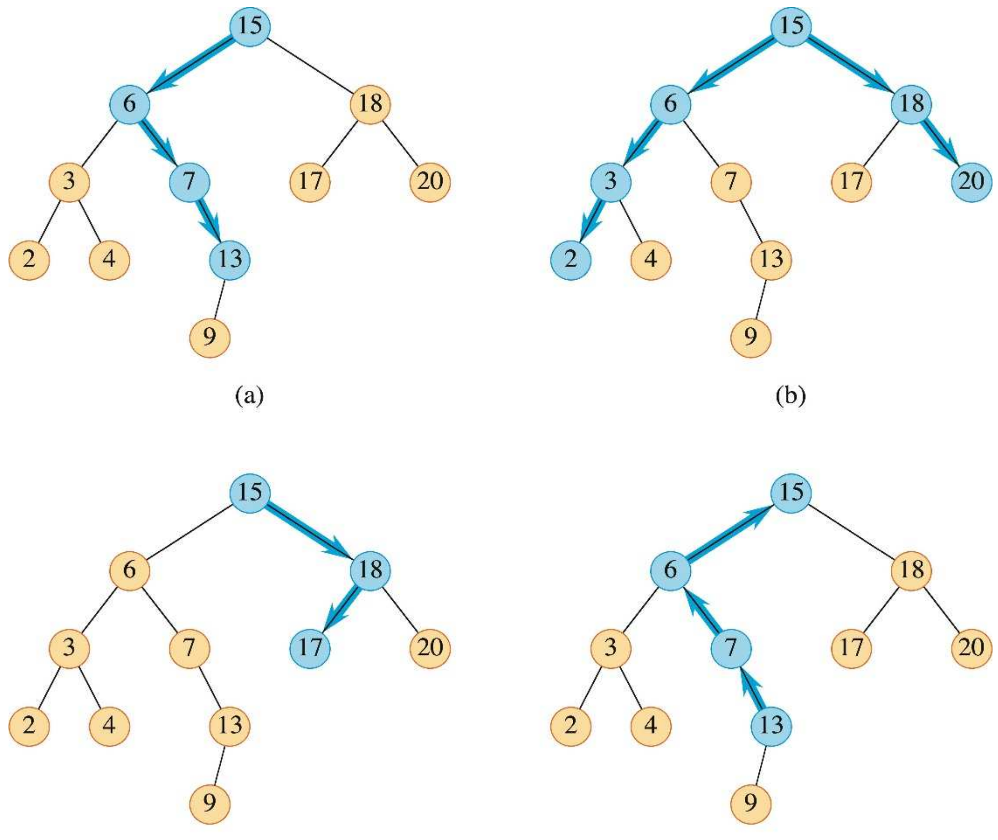
Binary Search Trees (continued)

– Successor and predecessor
  + by inorder tree walk
  + separate results for non-empty right subtree and for empty right subtree
  + time complexity for a tree of height h is $O(h)$

TREE-SUCCESSOR(x)

```
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p                      y is the parent node of x.
while y ≠ NIL and x == y.right
    x = y                    climb up the tree.
    y = y.p
return y
```

If node x has no right subtree, its successor is the lowest ancestor of x whose left subtree contains x.



(a) (b) (c) (d)

successor of *key* = 15 is 17
successor of *key* = 13 is 15

Binary Search Trees (continued)

– Insertion
  + tree node (say, *z*) to be added, with *z.key* = *v*, *z.left* = nil, *z.right* = nil
  + *z* added to one appropriate node with an ***absent*** left or right child (never a full node)
  + tree in-order walk, with the <u>trailing pointer *y*</u> maintained
  + time complexity for a tree of height *h* is <u>*O*(*h*)</u>

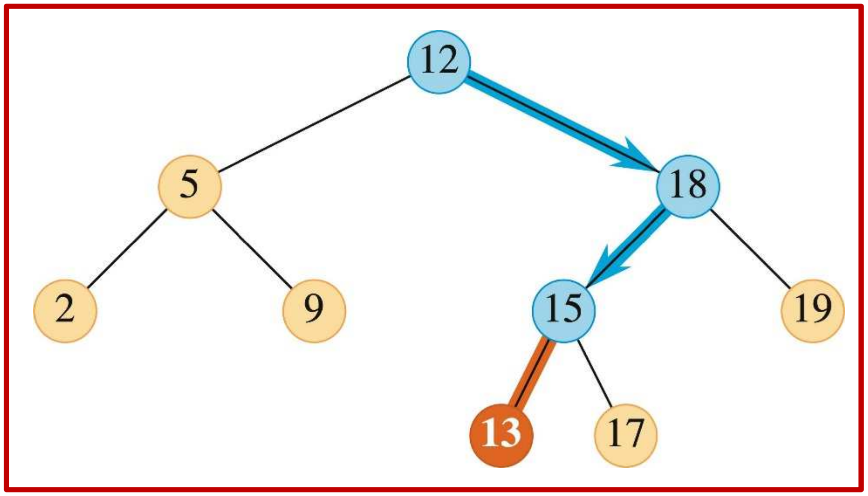
TREE-INSERT(T, z)

$y = \text{NIL}$
$x = T.root$
while $x \neq \text{NIL}$
  $y = x$
  if $z.key < x.key$
    $x = x.left$
  else $x = x.right$

y is the parent node of *x* after *x* is reassigned to move down

tree traversal downward until the insertion point, with the added node as a **leaf node**

$z.p = y$ // *y* is the parent node of *z*.
if $y == \text{NIL}$
  $T.root = z$ // tree T was empty
elseif $z.key < y.key$
  $y.left = z$
else $y.right = z$



Inserting *key* = 13 into binary search tree, with affected path marked.

Binary Search Trees (continued)

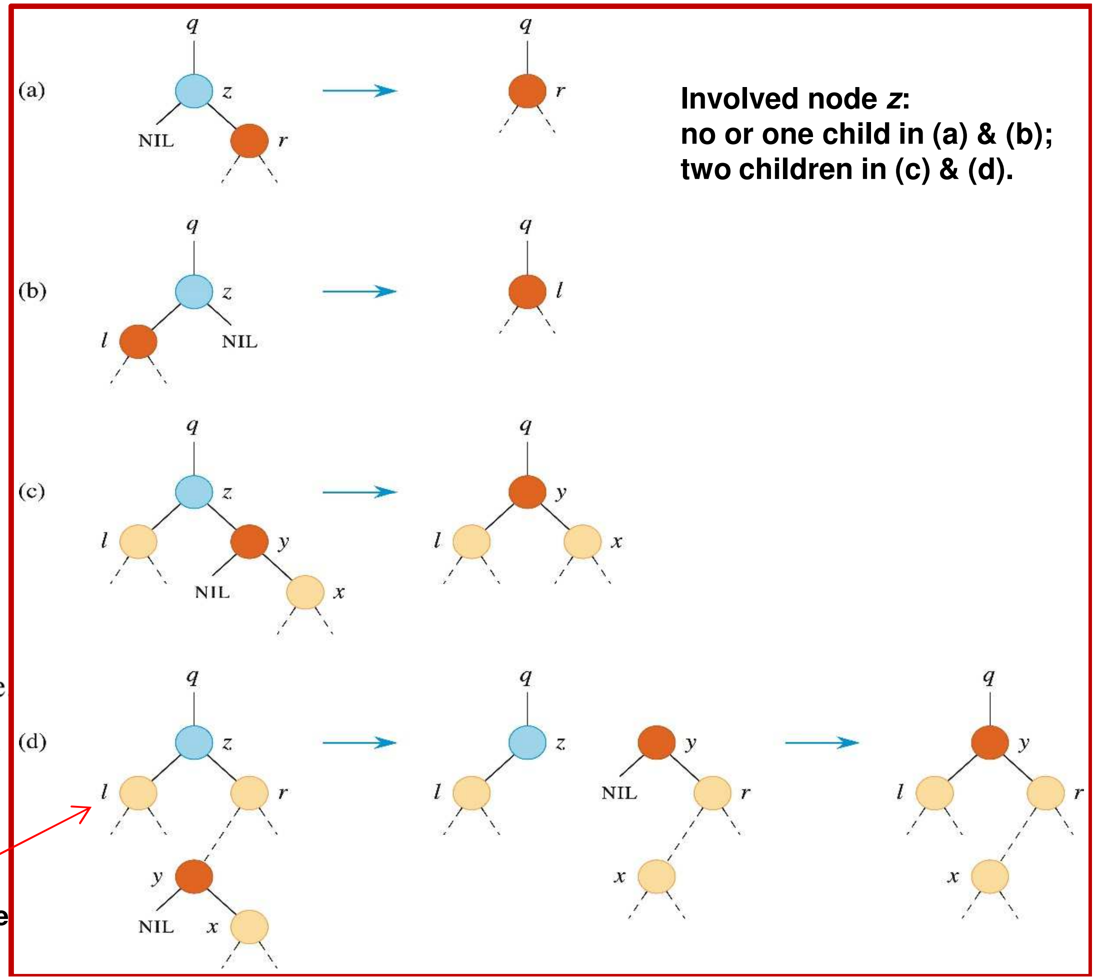
– Deletion (on *z*)

+ three basic cases involved: *z* having no child, exactly one child, two children
+ based on TRANSPLANT that replaces sibling subtree with another subtree
+ time complexity for a tree of height *h* is *O*(*h*)

```
TRANSPLANT(T, u, v)
  if u.p == NIL
      T.root = v
  elseif u == u.p.left
      u.p.left = v
  else u.p.right = v
  if v ≠ NIL
      v.p = u.p
```

```
TREE-DELETE(T, z)
 if z.left == NIL
     TRANSPLANT(T, z, z.right)        // z has no left child
 elseif z.right == NIL
     TRANSPLANT(T, z, z.left)         // z has just a left child
 else // z has two children.
     y = TREE-MINIMUM(z.right)        // y is z's successor
     if y.p ≠ z   <- This is case (d) shown in the figure.
         // y lies within z's right subtree but not the root of this subtree
         TRANSPLANT(T, y, y.right)  ]  Prepare a subtree rooted at y
         y.right = z.right          ]  whose left subtree is empty.
         y.right.p = y              ]
     // Replace z by y.
     TRANSPLANT(T, z, y)
     y.left = z.left
     y.left.p = y
```

Alternatively, one may cut the left-hand subtree of *z* and move it to become the left-hand subtree of *y*.



Binary Search Trees (continued)

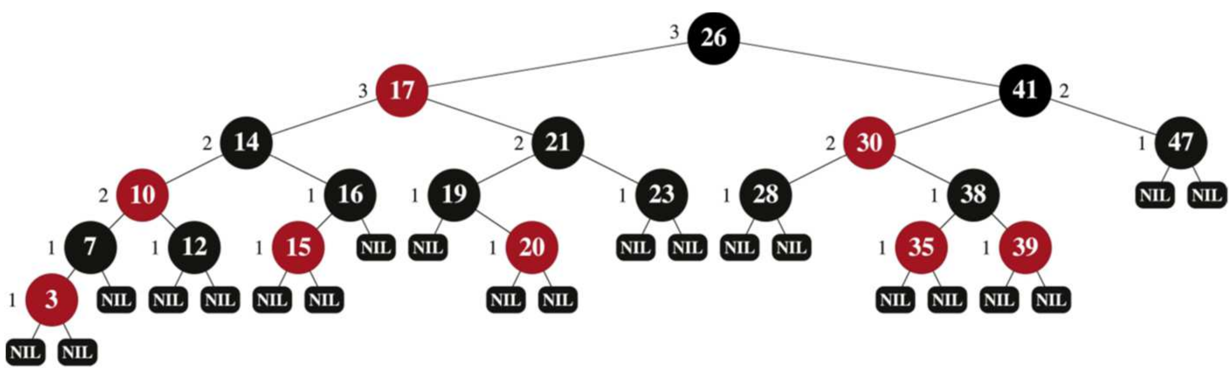
– Randomly build binary search trees
  + worst-case tree height (*h*) being n-1
  + can be shown that $h \geq \lfloor \lg n \rfloor$, which is the best case
  + like quicksort, average case proven to be much closer to best case than worst case
  + special case of creating binary search trees (via insertion alone) with random keys, we have following theorem:

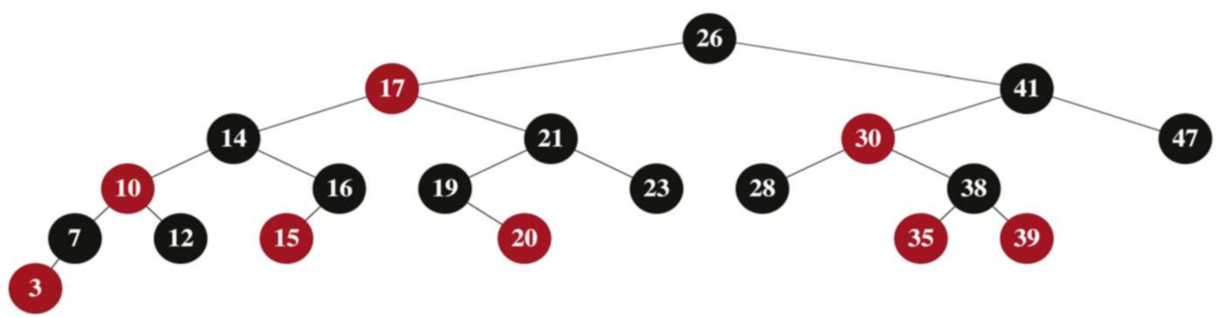
Theorem 2. ← This theorem refers to the special case that the tree is created by insertion alone (and no deletion).

The expected height of a randomly built binary search tree on n distinct keys is $\boldsymbol{O(\lg n)}$.

Red-Black Trees

– Red-black tree is a binary search tree that satisfies:
  (1) every node is either red or black
  (2) root is black
  (3) every leaf (NIL) is black
  (4) red node has its both children being black → **CANNOT have two connected red nodes**
  (5) all simple paths from any node to its descendant leaves contains the same number of black nodes (called the node's black-height, *bh*)





Red-Black Trees (continued)

+ Red-black tree is a good search tree because of its bounded small height.

Lemma 1:

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

- proof by showing that any subtree rooted at Node *x* has at least $2^{bh(x)} - 1$ internal nodes by induction, as the bh of each *x*'s child is $\geq bh(x) - 1$ to contain $\geq 2^{bh(x)-1} - 1$ internal nodes
- from RB tree property (4), if the height of *x* is *h*, then $bh(x) \geq h/2$, to yield $n \geq 2^{h/2} - 1$

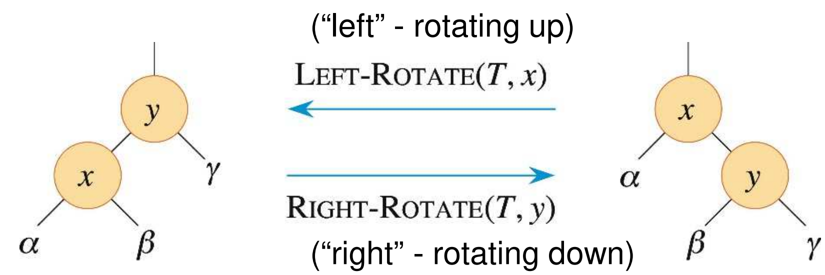
– Lemma 1 signifies that all operations (SEARCH, MIN, MAX, SUCCESSOR, PREDECESSOR) on an RB tree with *n* internal nodes take $O(\lg n)$ each.

<u>Red-Black Trees</u> (continued)

Rotations

+ insertion to, and deletion from, an RB tree involves <u>subtree rotations</u>

+ left rotations and right rotations



<u>LEFT-ROTATE(T, x)</u>

```
y = x.right
x.right = y.left          // turn y's left subtree into x's right subtree
if y.left ≠ T.nil         // if y's left subtree is not empty ...
    y.left.p = x          // ... then x becomes the parent of the subtree's root
y.p = x.p                 // x's parent becomes y's parent
if x.p == T.nil           // if x was the root ...
    T.root = y            // ... then y becomes the root
elseif x == x.p.left      // otherwise, if x was a left child ...
    x.p.left = y          // ... then y becomes a left child
else x.p.right = y        // otherwise, x was a right child, and now y is
y.left = x                // make x become y's left child
x.p = y
```

<u>Red-Black Trees</u> (continued)

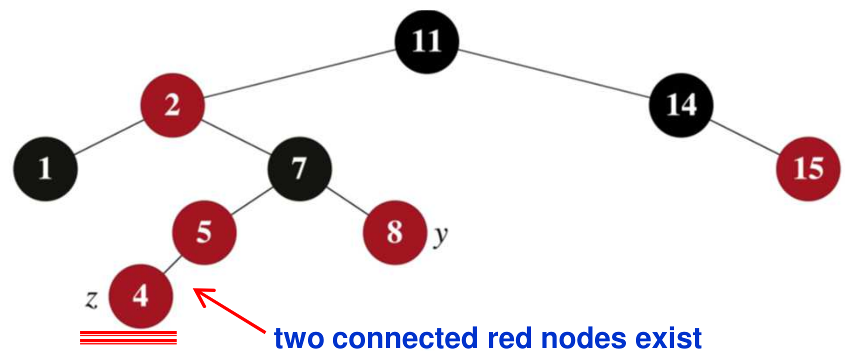
Insertion

+ inserts a node to an RB tree, RB-INSERT(T, z), with the inserted node z in <u>red</u> and descending to a tree <u>leaf node</u>

+ requires RB-INSERT-FIXUP(T, z) to ensure RB properties after insertion

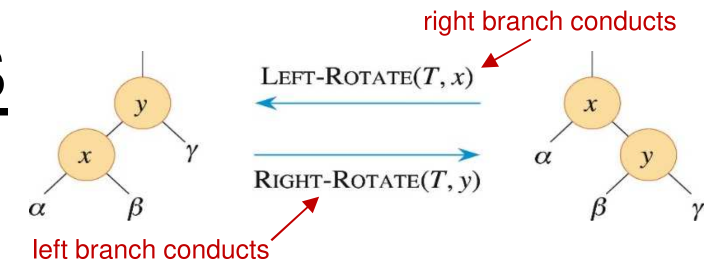
<u>RB-INSERT(T, z)</u>

```
x = T.root                       // node being compared with z
y = T.nil                        // y will be parent of z
while x ≠ T.nil                  // descend until reaching the sentine
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y                          // found the location—insert z with parent y
if y == T.nil
    T.root = z                   // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil                   // both of z's children are the sentinel
z.right = T.nil
z.color = RED                    // the new node starts out red
RB-INSERT-FIXUP(T, z)            // correct any violations of red-black properties
```



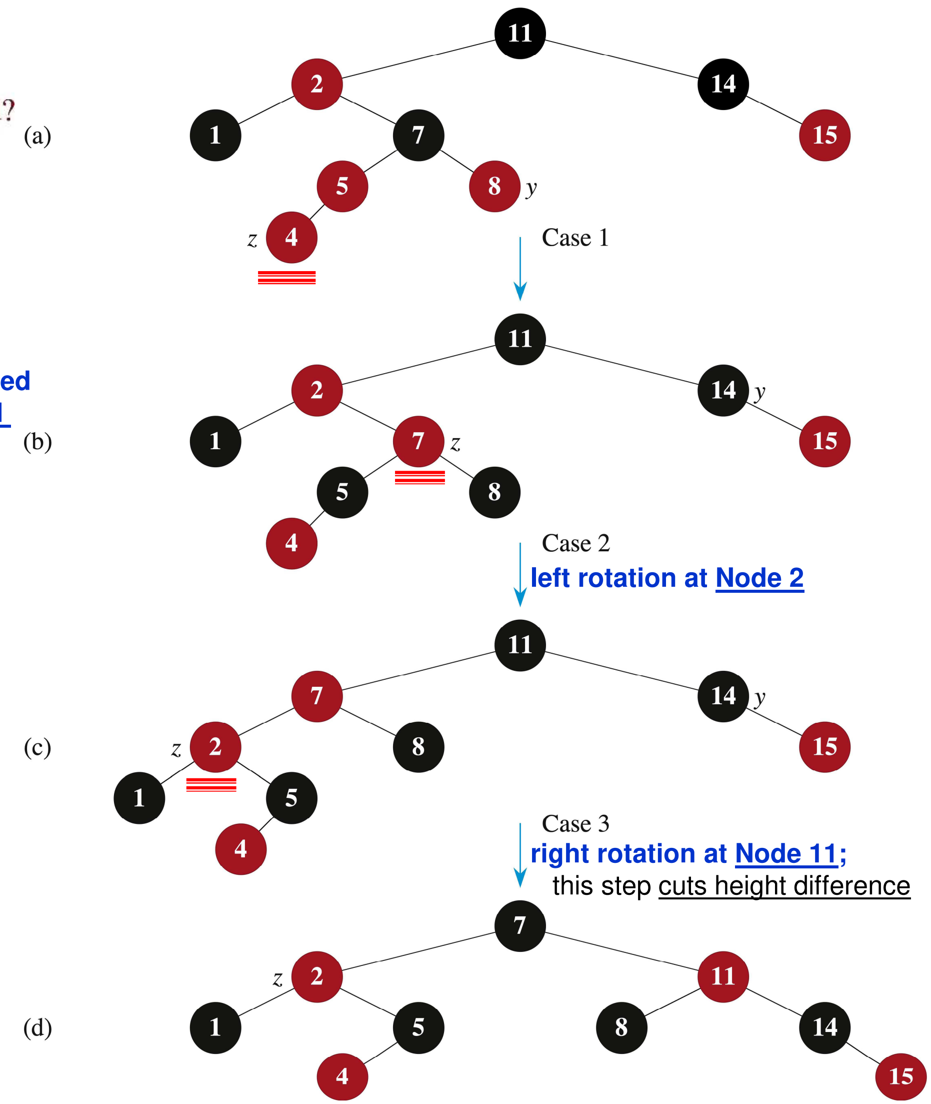
Red-Black Trees

+ RB-INSERT-FIXUP(T, z) to ensure RB properties




```
RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left              // is z's parent a left child?
3          y = z.p.p.right               // y is z's uncle
4          if y.color == RED             // are z's parent and uncle both red?
5              z.p.color = BLACK         ┐
6              y.color = BLACK           │ case 1
7              z.p.p.color = RED         │
8              z = z.p.p                 ┘
9          else
10             if z == z.p.right
11                 z = z.p               ┐ case 2
12                 LEFT-ROTATE(T, z)     ┘
13             z.p.color = BLACK         ┐
14             z.p.p.color = RED         │ case 3
15             RIGHT-ROTATE(T, z.p.p)    ┘
16     else // same as lines 3–15, but with "right" and "left" exchanged
17         y = z.p.p.left
18         if y.color == RED
19             z.p.color = BLACK
20             y.color = BLACK
21             z.p.p.color = RED
22             z = z.p.p
23         else
24             if z == z.p.left
25                 z = z.p
26                 RIGHT-ROTATE(T, z)
27             z.p.color = BLACK
28             z.p.p.color = RED
29             LEFT-ROTATE(T, z.p.p)
30 T.root.color = BLACK
```

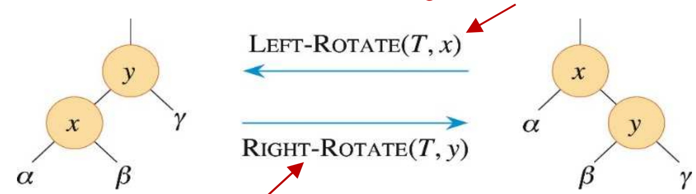
rotation performed if two connected red nodes exist

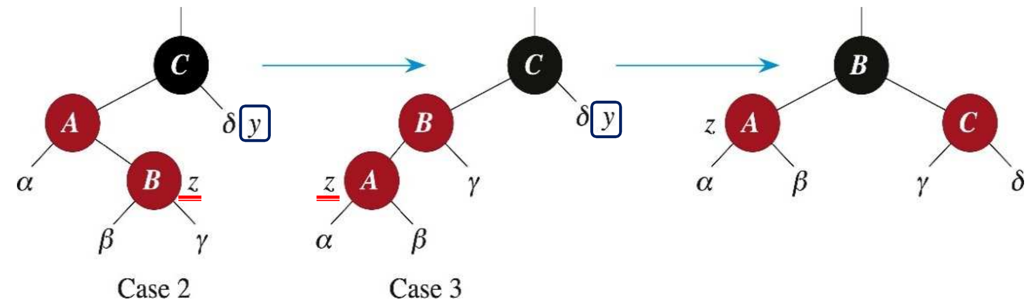


Red-Black Trees (continued)

+ RB-INSERT-FIXUP(T, z) to ensure RB properties

```
RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left               // is z's parent a left child?
3          y = z.p.p.right                // y is z's uncle
4          if y.color == RED              // are z's parent and uncle both red?
5              z.p.color = BLACK         ⎫
6              y.color = BLACK           ⎬ case 1
7              z.p.p.color = RED         ⎪
8              z = z.p.p                 ⎭
9          else
10             if z == z.p.right
11                 z = z.p               ⎫ case 2
12                 LEFT-ROTATE(T, z)     ⎭
13             z.p.color = BLACK         ⎫
14             z.p.p.color = RED         ⎬ case 3
15             RIGHT-ROTATE(T, z.p.p)    ⎭
16     else // same as lines 3–15, but with "right" and "left" exchanged
17         y = z.p.p.left
18         if y.color == RED
19             z.p.color = BLACK
20             y.color = BLACK
21             z.p.p.color = RED
22             z = z.p.p
23         else
24             if z == z.p.left
25                 z = z.p
26                 RIGHT-ROTATE(T, z)
27             z.p.color = BLACK
28             z.p.p.color = RED
29             LEFT-ROTATE(T, z.p.p)
30 T.root.color = BLACK
```





left rotation at Node *A*

right rotation at Node *C*, after making it red,
to cuts tree height difference

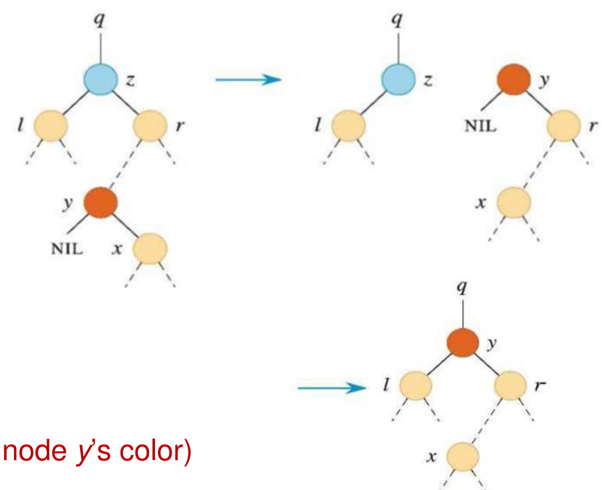
Red-Black Trees (continued)

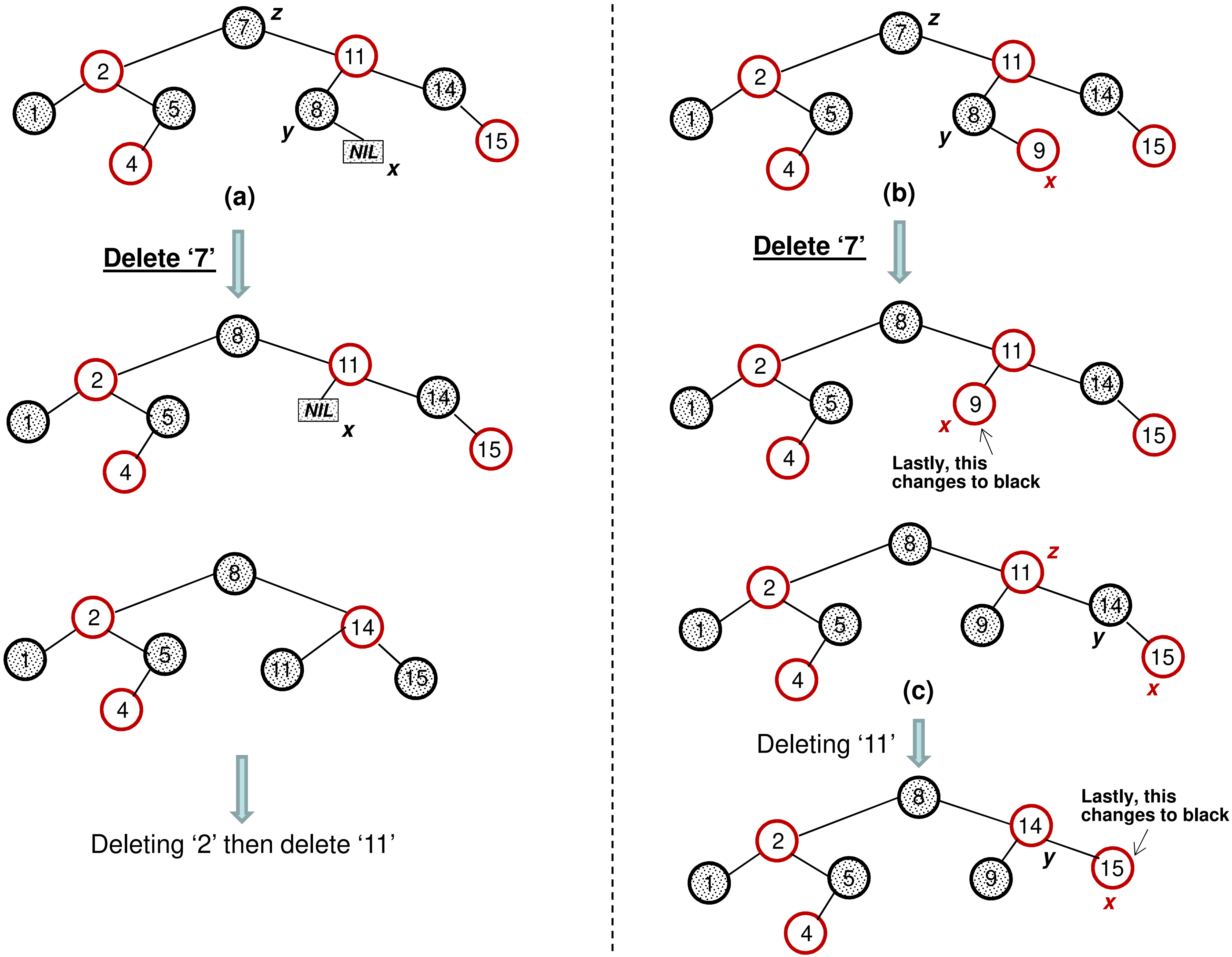
Deletion

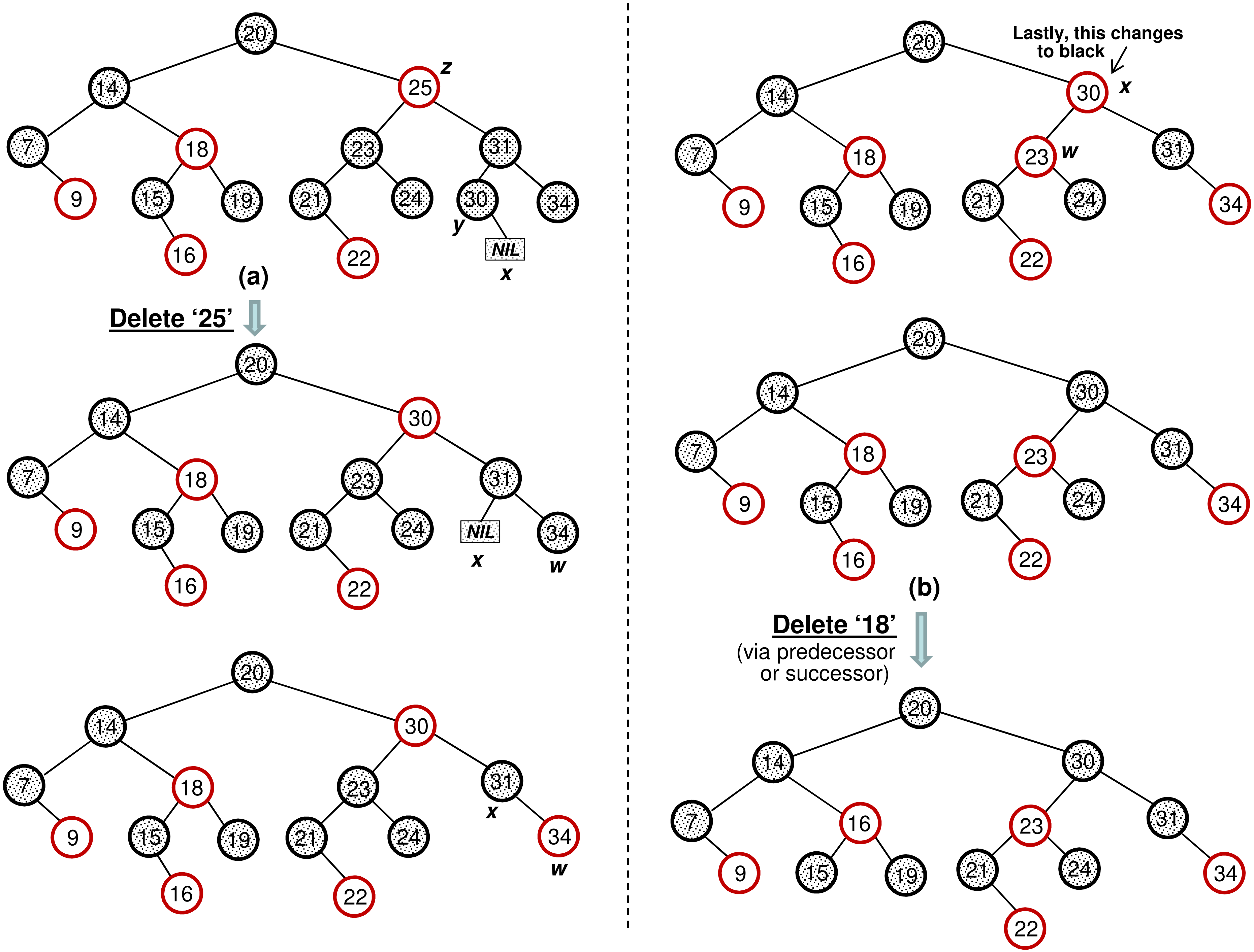
+ deletes an arbitrary node from RB tree, RB-DELETE(T, z)
+ requires RB-DELETE-FIXUP(T, z) to ensure RB properties after deletion

```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)       // replace z by its right child
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)        // replace z by its left child
else y = TREE-MINIMUM(z.right)         // y is z's successor
    y-original-color = y.color
    x = y.right
    if y ≠ z.right                     // is y farther down the tree?
        RB-TRANSPLANT(T, y, y.right)   // replace y by its right child
        y.right = z.right              // z's right child becomes
        y.right.p = y                  //     y's right child
    else x.p = y                       // in case x is T.nil
    RB-TRANSPLANT(T, z, y)             // replace z by its successor y
    y.left = z.left                    // and give z's left child to y,
    y.left.p = y                       //     which had no left child
    y.color = z.color
if y-original-color == BLACK       // if any red-black violations occurred,
    RB-DELETE-FIXUP(T, x)          //     correct them (based on replaced node y's color)
```

```
RB-TRANSPLANT(T, u, v)
if u.p == T.nil
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
v.p = u.p
```

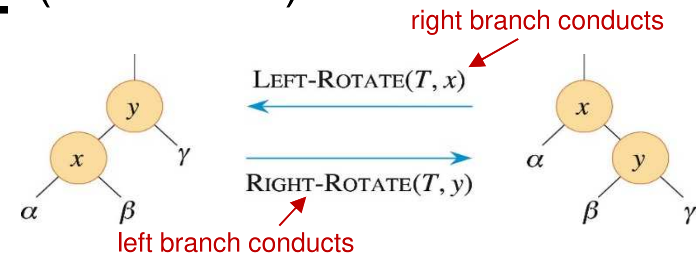



(a)
Delete ‘7’
NIL
Deleting ‘2’ then delete ‘11’
(b)
Delete ‘7’
Lastly, this changes to black
(c)
Deleting ‘11’
Lastly, this changes to black



(a)
Delete '25'
Lastly, this changes to black
(b)
Delete '18'
(via predecessor or successor)

<u>Red-Black Trees</u> (continued)

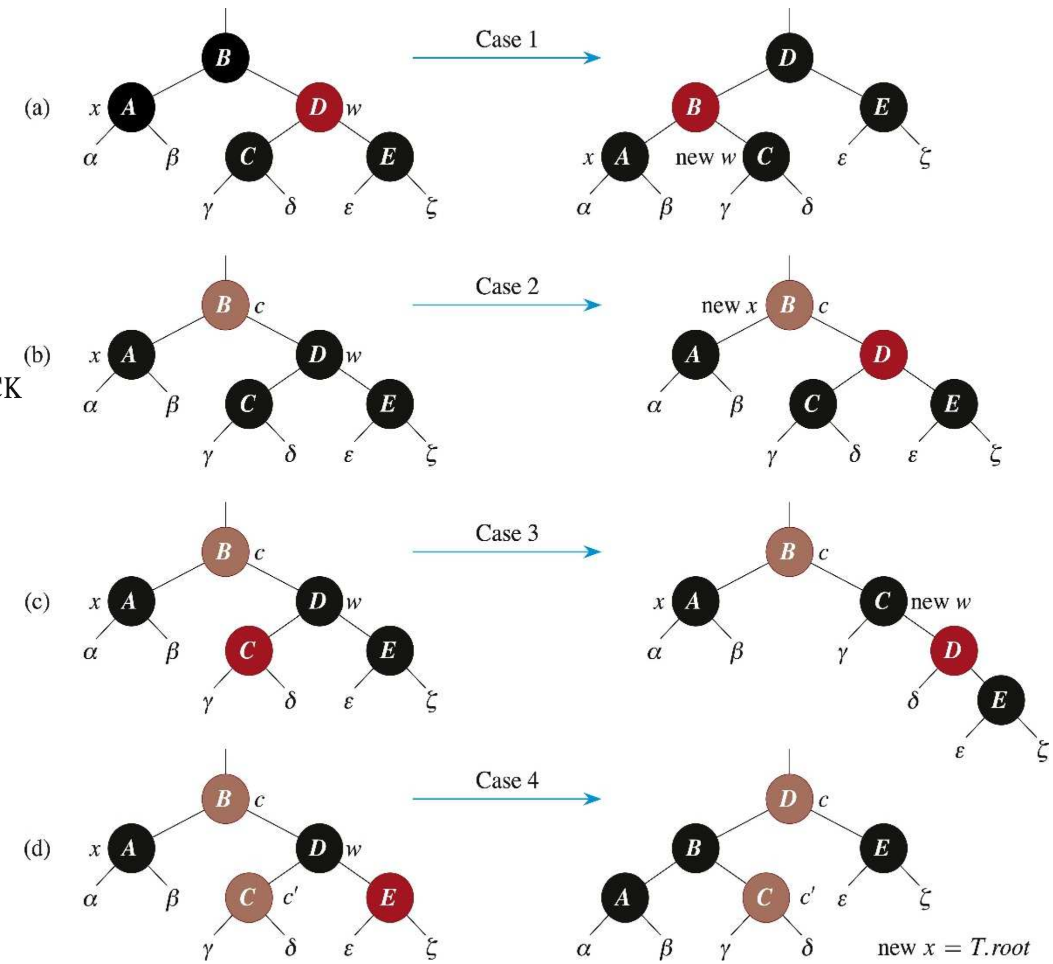


+ RB-DELETE-FIXUP(*T*, *z*) to ensure RB properties, i.e., identical <u>black-height</u>, *bh*, from any node

<u>RB-DELETE-FIXUP(T, x)</u>

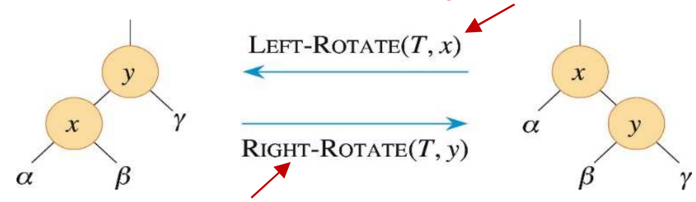
(based on the color of replacing node *x*)

```
while x ≠ T.root and x.color == BLACK
    if x == x.p.left              // is x a left child?
        w = x.p.right             // w is x's sibling
        if w.color == RED
            w.color = BLACK
            x.p.color = RED
            LEFT-ROTATE(T, x.p)              case 1
            w = x.p.right
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED
                                             case 2
            x = x.p
        else
            if w.right.color == BLACK
                w.left.color = BLACK
                w.color = RED                case 3
                RIGHT-ROTATE(T, w)
                w = x.p.right
            w.color = x.p.color
            x.p.color = BLACK
            w.right.color = BLACK            case 4
            LEFT-ROTATE(T, x.p)
            x = T.root
```



Red-Black Trees (continued)

+ RB-DELETE-FIXUP(T, z) to ensure RB properties, i.e., identical black-height, *bh*, from any node



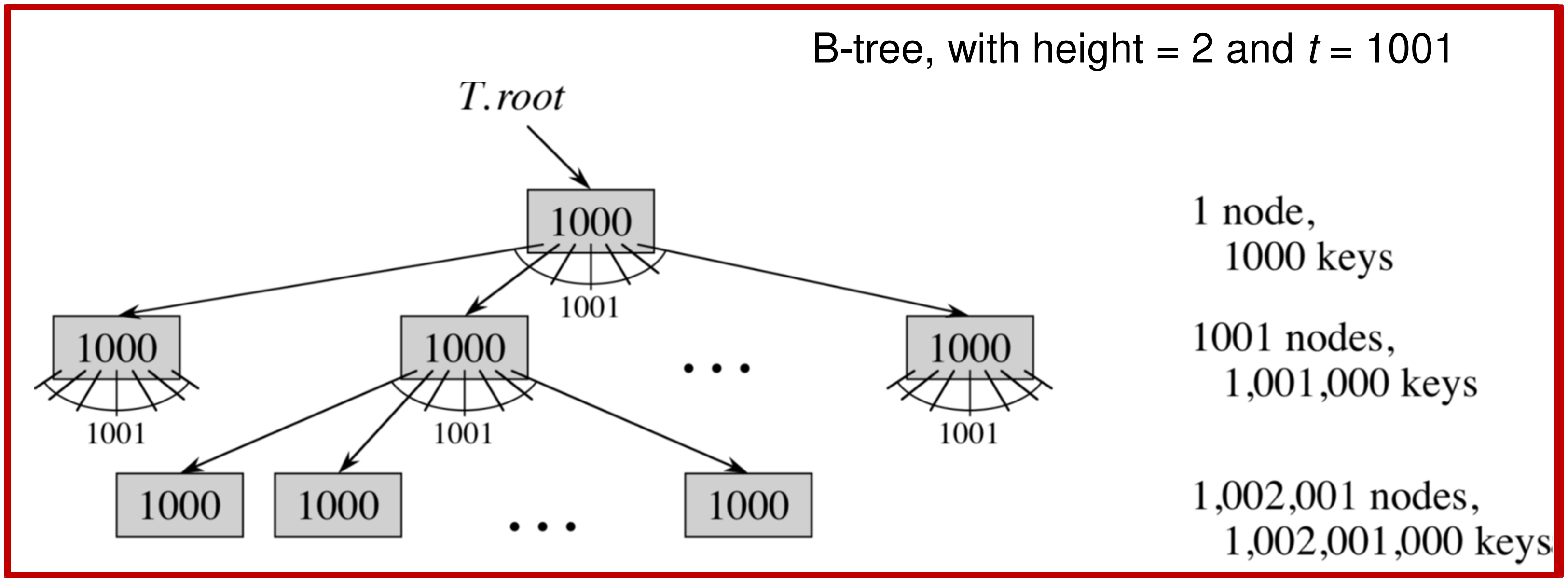
RB-DELETE-FIXUP(T, x) (based on the color of replacing node x)

```
1  while x ≠ T.root and x.color == BLACK
2      if x == x.p.left              // is x a left child?
3          w = x.p.right             // w is x's sibling
4          if w.color == RED
5              w.color = BLACK
6              x.p.color = RED
7              LEFT-ROTATE(T, x.p)                       case 1
8              w = x.p.right
9          if w.left.color == BLACK and w.right.color == BLACK
10             w.color = RED                             case 2
11             x = x.p
12         else
13             if w.right.color == BLACK
14                 w.left.color = BLACK
15                 w.color = RED                         case 3
16                 RIGHT-ROTATE(T, w)
17                 w = x.p.right
18             w.color = x.p.color
19             x.p.color = BLACK
20             w.right.color = BLACK                     case 4
21             LEFT-ROTATE(T, x.p)
22             x = T.root
23     else // same as lines 3–22, but with "right" and "left" exchanged
24         w = x.p.left
25         if w.color == RED
26             w.color = BLACK
27             x.p.color = RED
28             RIGHT-ROTATE(T, x.p)
29             w = x.p.left
30         if w.right.color == BLACK and w.left.color == BLACK
31             w.color = RED
32             x = x.p
33         else
34             if w.left.color == BLACK
35                 w.right.color = BLACK
36                 w.color = RED
37                 LEFT-ROTATE(T, w)
38                 w = x.p.left
39             w.color = x.p.color
40             x.p.color = BLACK
41             w.left.color = BLACK
42             RIGHT-ROTATE(T, x.p)
43             x = T.root
44 x.color = BLACK
```

B-Trees

§ **Balanced Search Trees (B-Trees)** with t — non-root node has ≥ t & ≤ 2t children

+ balance achieved by keeping multiple (i.e., ≥ t-1) keys in each non-root node
+ node has at most 2t-1 keys, called **full node** (with degree = 2t)
+ keys stored in a node in non-decreasing order
+ node x (with $x.n$ keys) has $x.n$+1 children, pointed by $x.c_1$, $x.c_2$, …., $x.c_{x.n+1}$, then $k_1 \le x.key_1 \le k_2 \le x.key_2 \le \ldots \le x.key_{x.n} \le k_{x.n+1}$, where k_i is a key stored in subtree rooted at $x.c_i$ and $x.key_i$ is a key stored in node x
+ all leaves have the same height
+ simplest B-tree is for t = 2, called 2-3-4 tree (each *internal node* has 2, 3, or 4 children)

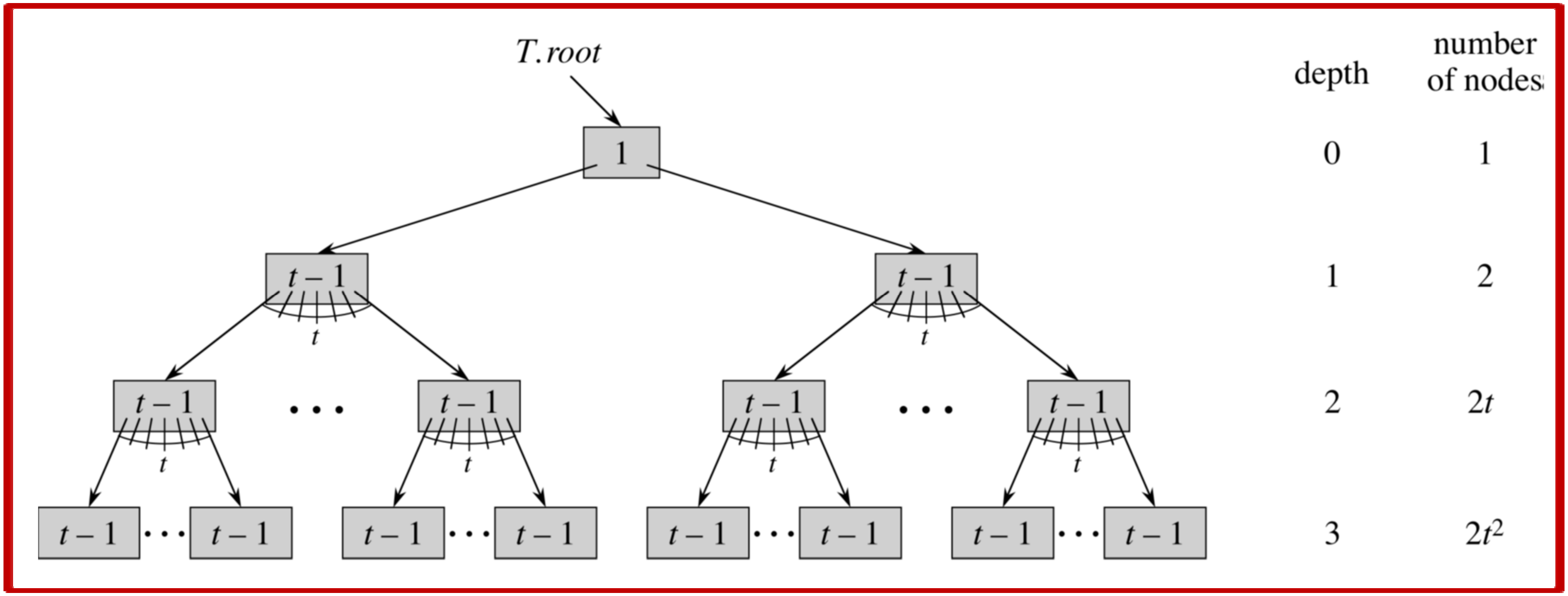


B-Trees (continued)

- **Theorem**

For any n-key B-tree T of height h and minimum node degree $t \geq 2$ (i.e., number of keys in each non-root node $\geq t$-1), we have $h \leq \log_t \frac{n+1}{2}$.

+ Proof follows the fact of $n \geq 1 + (t - 1) \cdot \sum_{i=1}^{h} 2 \cdot t^{i-1}$ illustrated in the figure below (to show the minimum possible number of keys in a B-tree with height = 3).



B-Trees (continued)

- **Basic operations**
  + Searching for record with key = k: each internal node ***x*** makes an ($x.n$ +1)-way branching decision; returning (y, i) → node ***y*** and its index i with $y.key_i = k$

```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```

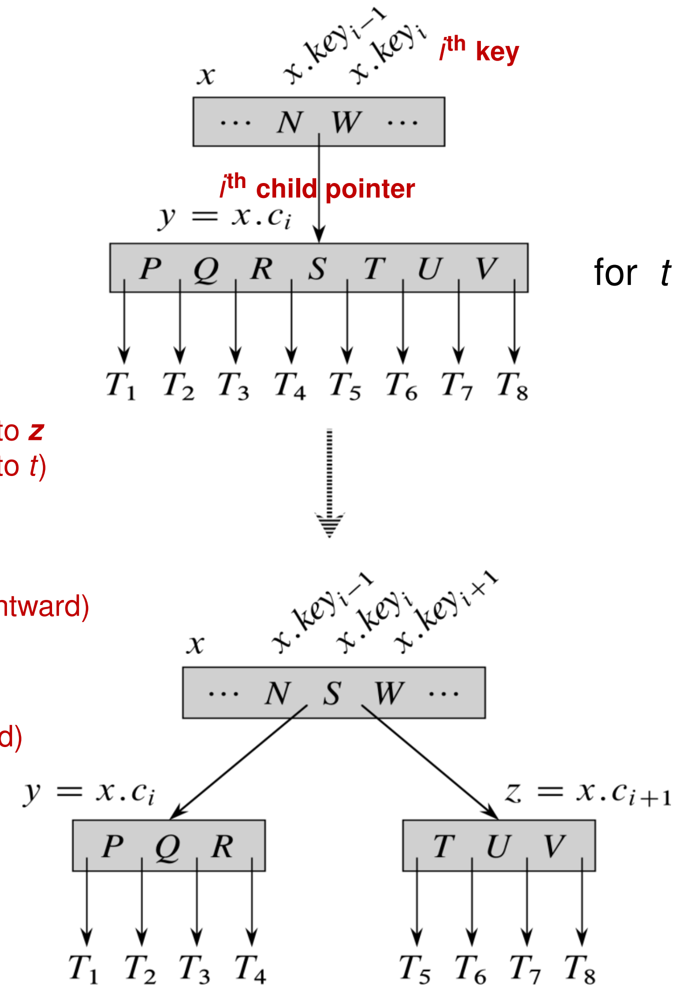
x.leaf: Boolean variable to indicate if it is a leaf node

B-Trees (continued)

- **Basic operations**
  - + Splitting a full node: for *non-full internal node* ***x*** and a *full child* of ***x*** (say, ***y*** = $x.c_i$), ***y*** is split at median key S, which moves up into ***x***, with those > S forming a new subtree

B-TREE-SPLIT-CHILD(x, i) **(split i^{th} child of x)**

```
z = ALLOCATE-NODE()
y = x.c_i                                   (Node x's i^th child is full)
z.leaf = y.leaf                             (status – leaf?)
z.n = t - 1
for j = 1 to t - 1
    z.key_j = y.key_{j+t}
if not y.leaf
    for j = 1 to t                          (move child pointers from y to z
        z.c_j = y.c_{j+t}                       with pointer # from 1 to t)
y.n = t - 1
for j = x.n + 1 downto i + 1
    x.c_{j+1} = x.c_j                       (shift child pointers in x rightward)
x.c_{i+1} = z
for j = x.n downto i
    x.key_{j+1} = x.key_j                   (shift keys in x rightward)
x.key_i = y.key_t                           (move middle key in y up)
x.n = x.n + 1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```



for $t = 4$

B-Trees (continued)

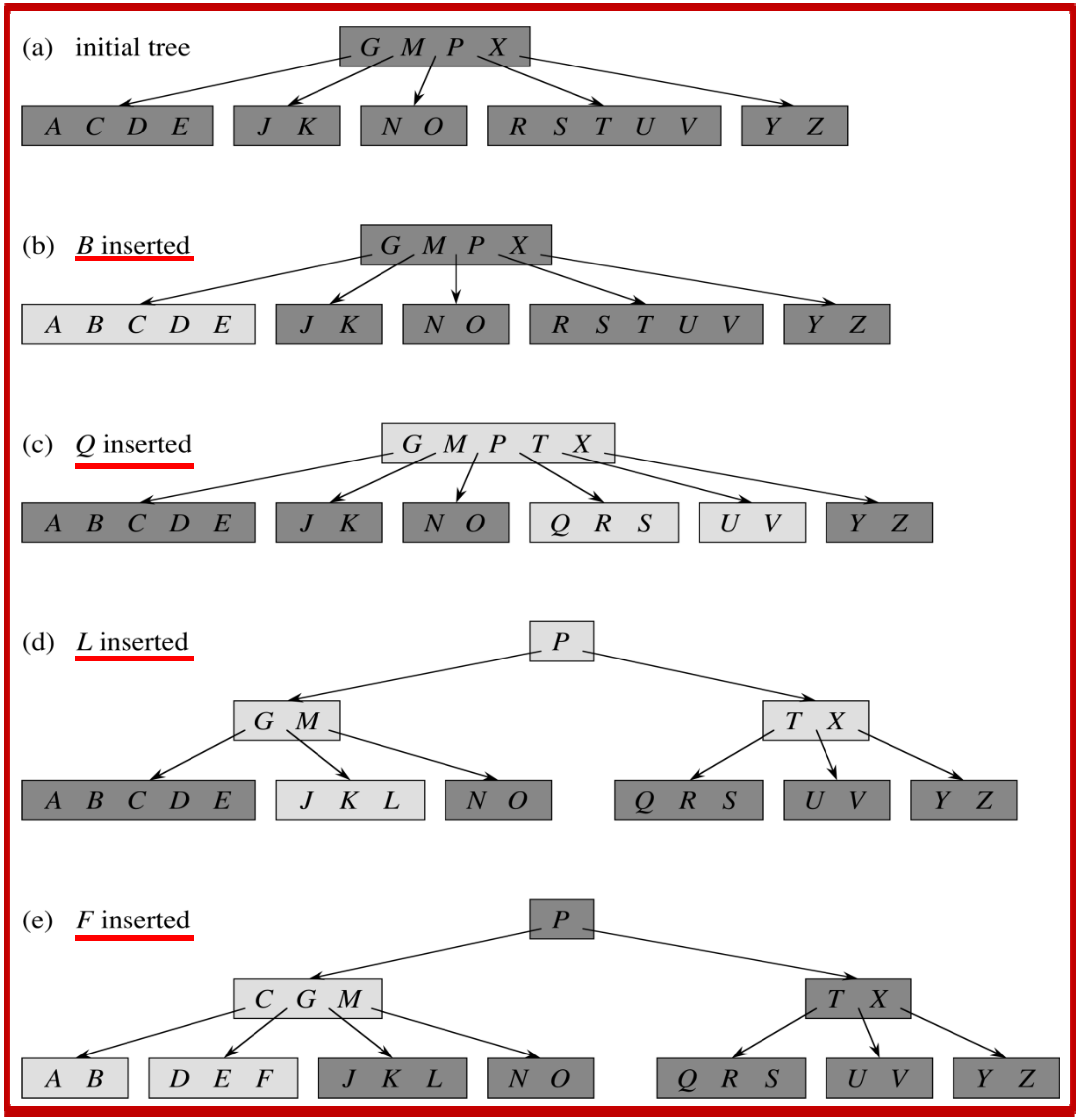
- **Basic operations**
  - + Inserting a key: B-tree ***T*** of height *h*, takes *O*(*h*) disk accesses (nodes kept as disk pages)
    - – *can't* insert the key into a newly created node *nor* into an internal node directly
      - ⟶ so, insert it only to a **leaf node**
    - – **never** descend through a **full node**, achieved by B-TREE-SPLIT-CHILD, to avoid back-tracking altogether
    - – root split increases the height by 1

```
B-TREE-INSERT(T, k)
r = T.root
if r.n == 2t − 1                                  // splitting the root?
    s = ALLOCATE-NODE()
    T.root = s
    s.leaf = FALSE
    s.n = 0
    s.c_1 = r
    B-TREE-SPLIT-CHILD(s, 1)   (split 1st child of S)
    B-TREE-INSERT-NONFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)
```

```
B-TREE-INSERT-NONFULL(x, k)
i = x.n
if x.leaf          (upon a leaf, key is inserted therein)
    while i ≥ 1 and k < x.key_i
        x.key_{i+1} = x.key_i   (shift key in x rightwar
        i = i − 1
    x.key_{i+1} = k
    x.n = x.n + 1
    DISK-WRITE(x)
else while i ≥ 1 and k < x.key_i
        i = i − 1
    i = i + 1
    DISK-READ(x.c_i)   (descend one level)
    if x.c_i.n == 2t − 1   // full child?
        B-TREE-SPLIT-CHILD(x, i)
        if k > x.key_i
            i = i + 1
    B-TREE-INSERT-NONFULL(x.c_i, k)
```

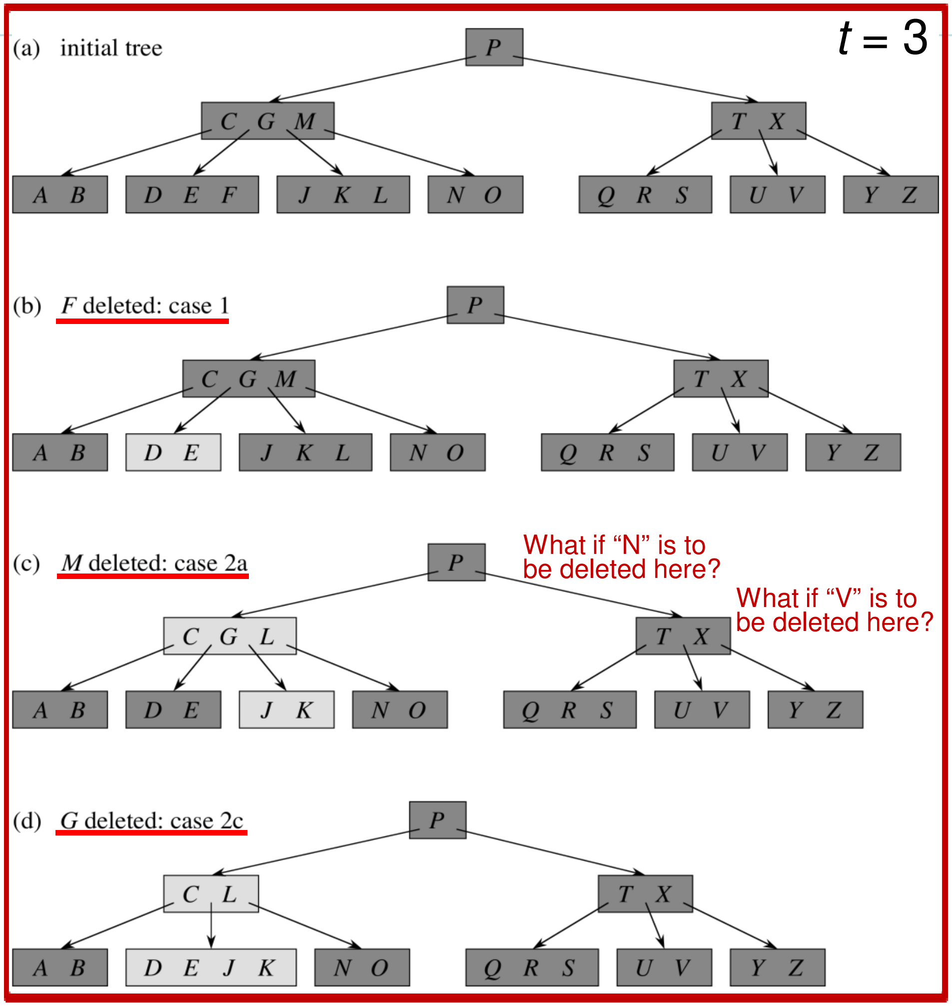
B-Trees (continued)

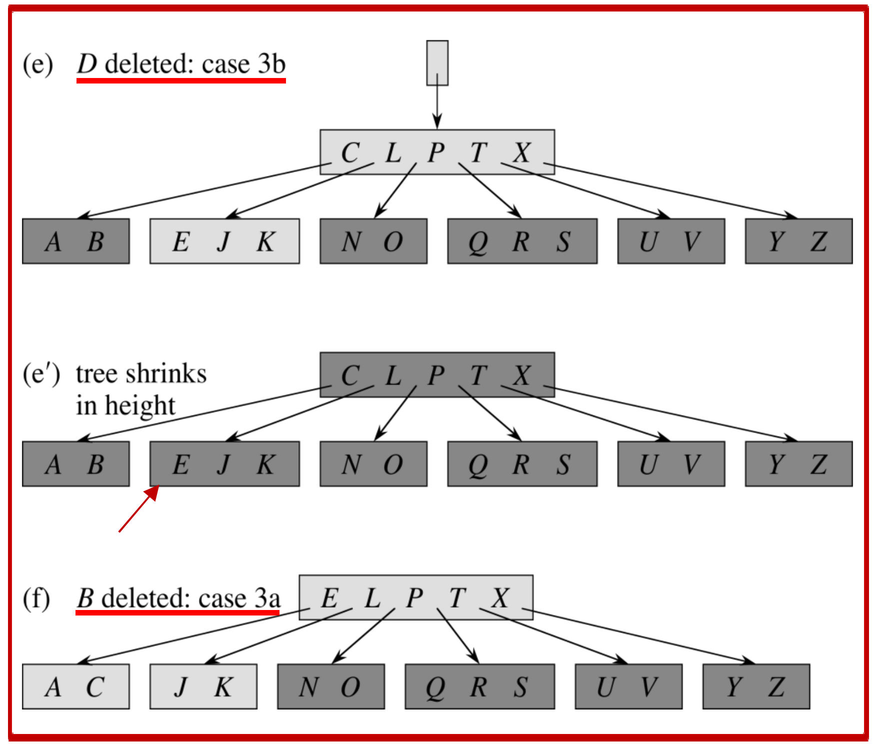
+ Inserting a key: B-tree ***T*** with *t* = 3 and modified nodes lightly shaded



B-Trees (continued)

- **Basic operations**
  + Deleting a key: delete key k from the subtree rooted at $\boldsymbol{x}$ in B-tree $\boldsymbol{T}$ with the minimum degree of t (where the key may be in any node, not just at a leaf node)
    – number of keys kept in any **internal node** upon descending through would be **at least t**





B-Trees (continued)

+ Deleting key *k* from the subtree rooted at ***x*** in B-tree ***T*** with the minimum degree of *t*
  – **Deletion Procedure** sketched below:

1. If the key k is in node x and x is a leaf, delete the key k from x.
2. If the key k is in node x and x is an internal node, do the following:
   a. If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x. (We can find k' and delete it in a single downward pass.)
   b. If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x. (We can find k' and delete it in a single downward pass.)
   c. Otherwise, if both y and z have only $t-1$ keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains $2t-1$ keys. Then free z and recursively delete k from y.
3. If the key k is not present in visited internal node x, determine the root $x.c_i$ of the appropriate subtree that must contain k, if k is in the tree at all. If $x.c_i$ has only $t-1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.
   a. If $x.c_i$ has only $t-1$ keys but has an immediate sibling with at least t keys, give $x.c_i$ an extra key by moving a key from x down into $x.c_i$, moving a key from $x.c_i$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $x.c_i$.
   b. If $x.c_i$ and both of $x.c_i$'s immediate siblings have $t-1$ keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

Targeted node with the key, but it may be non-leaf

This is Rule 1 - Key *k* present:
At internal node *x* which contains the key (say, *k*) to be deleted, it checks if predecessor or successor of *k* can be borrowed to replace *k* (and in this case, only the borrowed key is moved; and its associated pointer stays).

Prepare to walk down the tree

This is Rule 2 - Key *k* not present:
While at internal node *x*, it checks if the root of the target child has at least *t* key
If not, it prepares that root node by borrowing a key (plus associated pointer, i.e., a subtree) from its left or right sibling, if possible.