CH08-320201

Algorithms and Data Structures

ADS

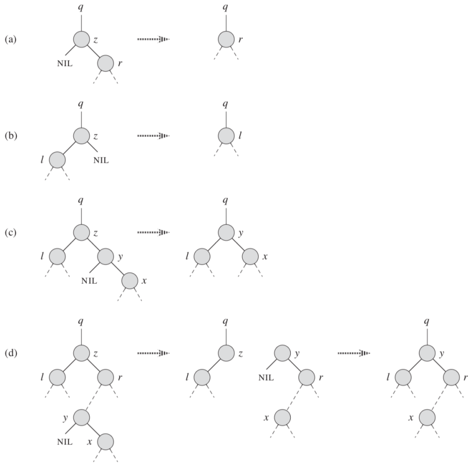
Lecture 16

Dr. Kinga Lipskoch

Spring 2019

Deletion (Remember BST)

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```



Deletion (RB) (1)

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

```
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)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

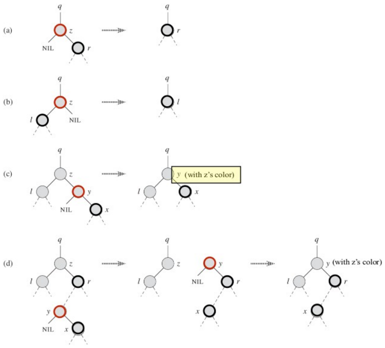
Deletion (RB) (2)

- node y
  - either removed (a/b)
  - or moved in the tree (c/d)
  - y-original-color
- node x
  - the node that moves into y's original position
  - x.p points to y's original parent (since it moves into y's position, note special case in 12/13)

```
RB-DELETE(T, z)
1   y = z
2   y-original-color = y.color
3   if z.left == T.nil
4       x = z.right
5       RB-TRANSPLANT(T, z, z.right)
6   elseif z.right == T.nil
7       x = z.left
8       RB-TRANSPLANT(T, z, z.left)
9   else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      else RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17      RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if y-original-color == BLACK
22      RB-DELETE-FIXUP(T, x)
```

Deletion (RB) (3)

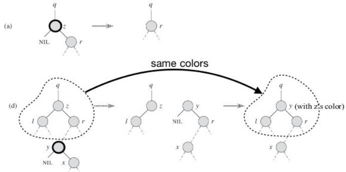
- y-original-color == red




```
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)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

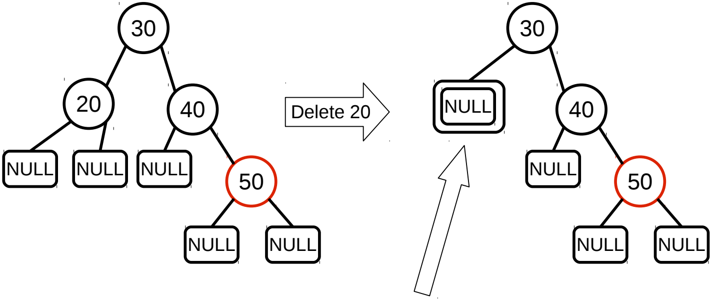
Deletion (RB) (4)

- y-original-color == red
  - no problem
- y-original-color == black
  - violations might occur (2,4,5)
  - main idea to fix
    - x gets an “**extra black**” & needs to get rid of it
  - 4 cases




```
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)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

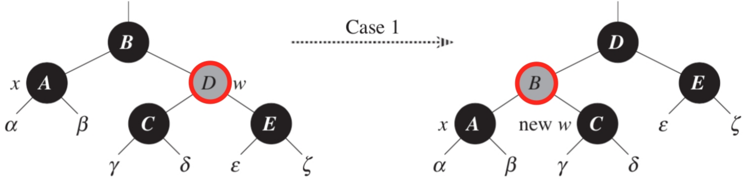
Extra Black or Double Black Node



Child caries „extra black“ information
also called „double black“ node

Fixing Red-Black Tree Properties (1)

Case 1: x's sibling w is red.
Transform to Case 2, 3, or 4 by left rotation and changing colors of nodes B and D.

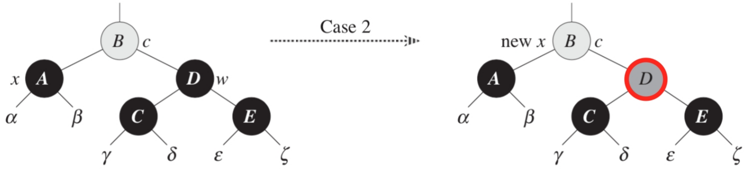


x = node with extra black
w = x's sibling

if $w.color$ == RED
  $w.color$ = BLACK
  $x.p.color$ = RED
  LEFT-ROTATE$(T, x.p)$
  $w = x.p.right$

Fixing Red-Black Tree Properties (2)

Case 2: x's sibling w is black and the children of w are black.
Set color of w to red and propagate upwards.



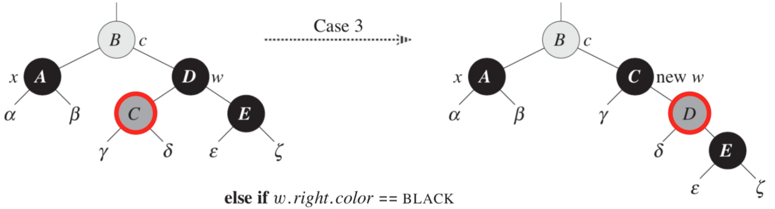
x = node with extra black
w = x's sibling
c = color of the node

if $w.left.color$ == BLACK and $w.right.color$ == BLACK
$\quad w.color = \text{RED}$
$\quad x = x.p$

Fixing Red-Black Tree Properties (3)

Case 3: x's sibling w is black and the left child of w is red, while the right child of w is black.

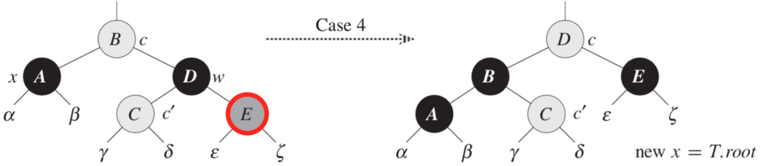
Transform to Case 4 by right rotation and changing colors of nodes C and D.




```
else if w.right.color == BLACK
        w.left.color = BLACK
        w.color = RED
        RIGHT-ROTATE(T, w)
        w = x.p.right
```

Fixing Red-Black Tree Properties (4)

Case 4: x's sibling w is black and the right child of w is red. Perform a left-rotate and change colors of B, D, and E. Then, the loop terminates.



$w.color = x.p.color$
$x.p.color = \text{BLACK}$
$w.right.color = \text{BLACK}$
$\text{LEFT-ROTATE}(T, x.p)$

Fixing Red-Black Tree Properties (5)

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

Time complexity: $O(h) = O(\lg n)$

Conclusion

Modifying operations on red-black trees can be executed in $O(\lg n)$ time.

Direct Access Table

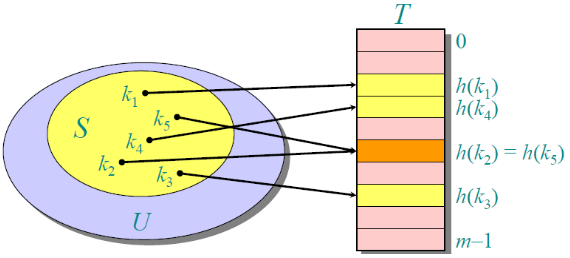
- The idea of a direct access table is that objects are directly accessed via their key.
- Assuming that keys are out of $U = \{0, 1, ..., m-1\}$.
- Moreover, assume that keys are distinct.
- Then, we can set up an array $T[0..m-1]$ with

$$T[k] = \begin{cases} x & \text{if } x \in K \text{ and } key[x] = k \\ \text{NIL} & \text{otherwise.} \end{cases}$$

- Time complexity: With this set-up, we can have the dynamic-set operations (Search, Insert, Delete, ...) in $\Theta(1)$.
- Problem: m is often large. For example, for 64-bit numbers we have $18,446,744,073,709,551,616$ different keys.

Hash Function

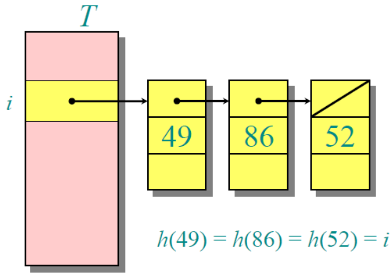
- Use a function h that maps U to a smaller set $\{0, 1, ..., m-1\}$.



- Such a function is called a hash function.
- The table T is called a hash table.
- If two keys are mapped to the same location, we have a collision.

Resolving Collisions

- Collisions can be resolved by storing the colliding mappings in a (singly-)linked list.



- Worst case: All keys are mapped to the same location. Then, access time is $\Theta(n)$.

Average Case Analysis (1)

- Assumption (simple uniform hashing): Each key is equally likely to be hashed to any slot of the table, independent of where other keys are hashed.
- Let n be the number of keys.
- Let m be the number of slots.
- The load factor $\alpha = n/m$ represents the average number of keys per slot.

Average Case Analysis (2)

Theorem:
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.
Proof:

- Any key k not already stored in the table is equally likely to hash to any of the m slots.
- The expected time to search unsuccessfully for a key k is the expected time to search to the end of list $T[h(k)]$.
- Expected length of the list is $E[n_{h(k)}] = \alpha$.
- Time for computing $h(k) = O(1) \Rightarrow$ overall time $\Theta(1+\alpha)$.

Average Case Analysis (3)

- Runtime for unsuccessful search:
  The expected time for an unsuccessful search is $\Theta(1+\alpha)$ including applying the hash function and accessing the slot and searching the list.
- What does this mean?
  - $m \sim n$, i.e., if $n = O(m) \Rightarrow \alpha = n/m = O(m)/m = O(1)$
  - Thus, search time is $O(1)$
- A successful search has the same asymptotic bound.