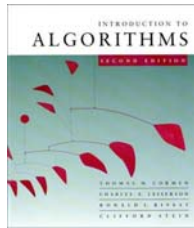




CS 3343 -- Spring 2009



Red-black trees

Carola Wenk

Slides courtesy of Charles Leiserson with small changes by Carola Wenk

3/5/09

CS 3343 Analysis of Algorithms

1

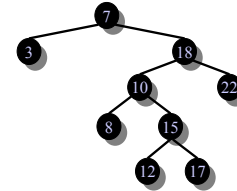


Search Trees

- A binary search tree is a binary tree. Each node stores a key. The tree fulfills the **binary search tree property**:

For every node x holds:

- $y \leq x$, for all y in the subtree left of x
- $x < y$, for all y in the subtree right of x



3/5/09

CS 3343 Analysis of Algorithms

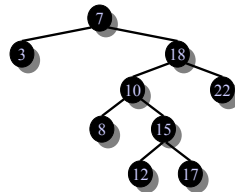
2



Search Trees

Different variants of search trees:

- Balanced search trees (guarantee height of $\log n$ for n elements)
- k -ary search trees (such as B-trees, 2-3-4-trees)
- Search trees that store the keys only in the leaves, and store additional split-values in the internal nodes



3/5/09

CS 3343 Analysis of Algorithms

3

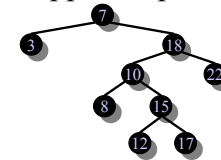


ADT Dictionary / Dynamic Set

Abstract data type (ADT) Dictionary
(also called **Dynamic Set**):

A data structure which supports operations

- Insert
- Delete
- Find



Using **balanced binary search trees** we can implement a dictionary data structure such that each operation takes $O(\log n)$ time.

3/5/09

CS 3343 Analysis of Algorithms

4



Balanced search trees

Balanced search tree: A search-tree data structure for which a height of $O(\log n)$ is guaranteed when implementing a dynamic set of n items.

Examples:

- AVL trees
- 2-3 trees
- 2-3-4 trees
- B-trees
- Red-black trees



Red-black trees

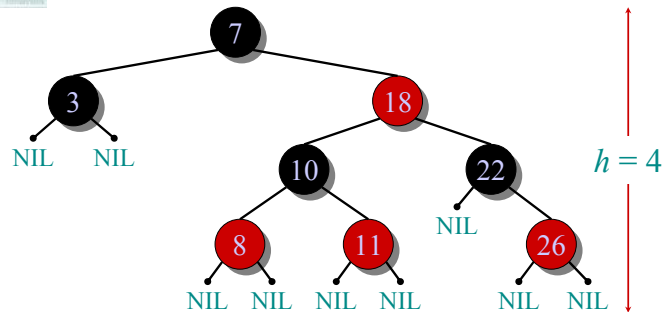
This data structure requires an extra one-bit **color** field in each node.

Red-black properties:

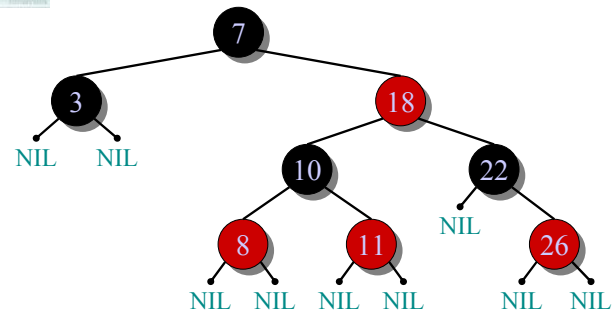
1. Every node is either red or black.
2. The root is black.
3. The leaves (NIL's) are black.
4. If a node is red, then both its children are black.
5. All simple paths from any node x , excluding x , to a descendant leaf have the same number of black nodes = **black-height**(x).



Example of a red-black tree



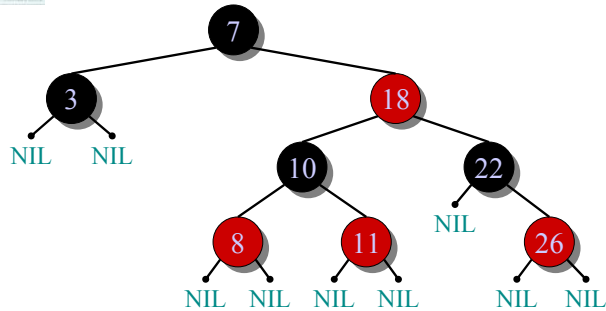
Example of a red-black tree



1. Every node is either red or black.



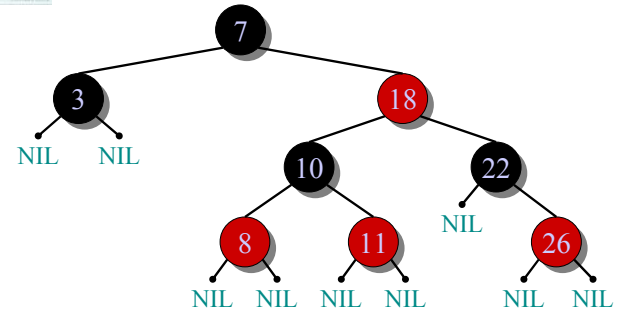
Example of a red-black tree



2., 3. The root and leaves (NIL's) are black.



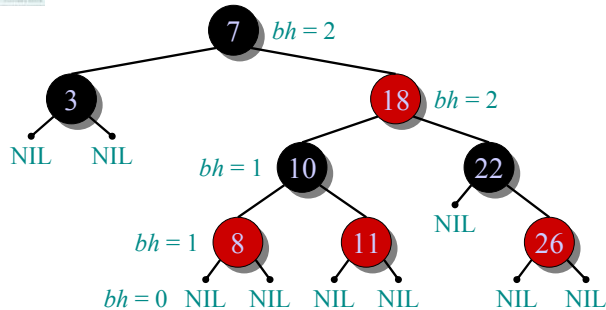
Example of a red-black tree



4. If a node is red, then both its children are black.



Example of a red-black tree



5. All simple paths from any node x , excluding x , to a descendant leaf have the same number of black nodes = *black-height*(x).



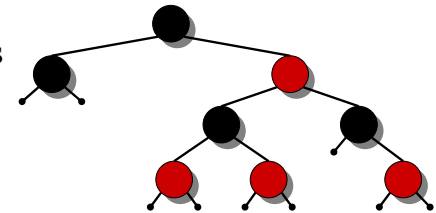
Height of a red-black tree

Theorem. A red-black tree with n keys has height $h \leq 2 \log(n + 1)$.

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.





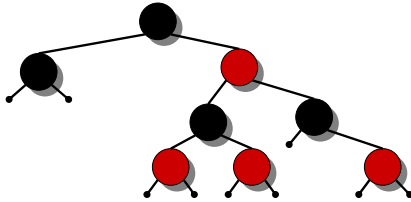
Height of a red-black tree

Theorem. A red-black tree with n keys has height $h \leq 2 \log(n + 1)$.

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



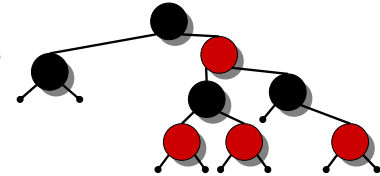
Height of a red-black tree

Theorem. A red-black tree with n keys has height $h \leq 2 \log(n + 1)$.

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



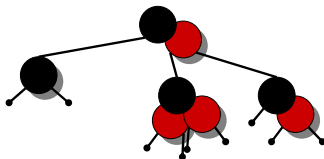
Height of a red-black tree

Theorem. A red-black tree with n keys has height $h \leq 2 \log(n + 1)$.

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



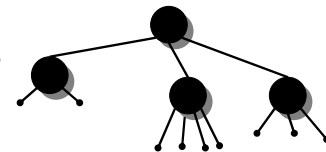
Height of a red-black tree

Theorem. A red-black tree with n keys has height $h \leq 2 \log(n + 1)$.

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.





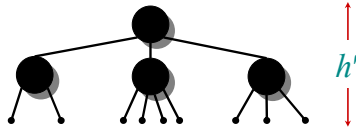
Height of a red-black tree

Theorem. A red-black tree with n keys has height $h \leq 2 \log(n + 1)$.

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.
- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth h' of leaves.



3/5/09

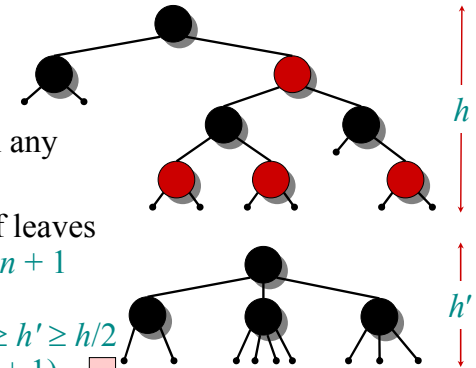
CS 3343 Analysis of Algorithms

17



Proof (continued)

- We have $h' \geq h/2$, since at most half the vertices on any path are red.
- The number of leaves in each tree is $n + 1$
 $\Rightarrow n + 1 \geq 2^{h'}$
 $\Rightarrow \log(n + 1) \geq h' \geq h/2$
 $\Rightarrow h \leq 2 \log(n + 1)$. \square



3/5/09

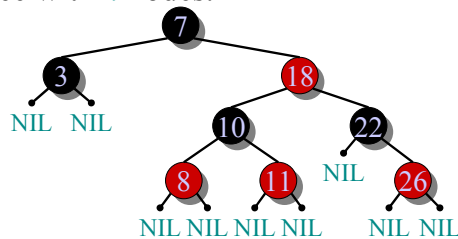
CS 3343 Analysis of Algorithms

18



Query operations

Corollary. The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR all run in $O(\log n)$ time on a red-black tree with n nodes.



3/5/09

CS 3343 Analysis of Algorithms

19



Modifying operations

The operations INSERT and DELETE cause modifications to the red-black tree:

1. the operation itself,
2. color changes,
3. restructuring the links of the tree via **“rotations”**.

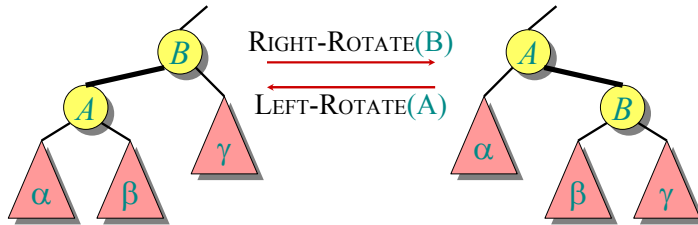
3/5/09

CS 3343 Analysis of Algorithms

20



Rotations



- Rotations maintain the inorder ordering of keys:
 $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c.$
- Rotations maintain the binary search tree property
- A rotation can be performed in $O(1)$ time.

3/5/09

CS 3343 Analysis of Algorithms

21



Red-black trees

This data structure requires an extra one-bit **color** field in each node.

Red-black properties:

1. Every node is either red or black.
2. The root is black.
3. The leaves (NIL's) are black.
4. If a node is red, then both its children are black.
5. All simple paths from any node x , excluding x , to a descendant leaf have the same number of black nodes = **black-height**(x).

3/5/09

CS 3343 Analysis of Algorithms

22

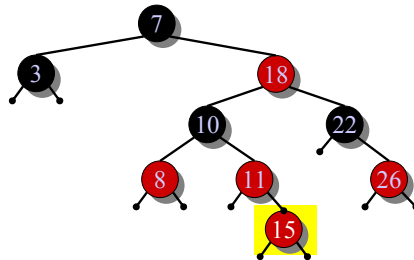


Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 4 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.



3/5/09

CS 3343 Analysis of Algorithms

23

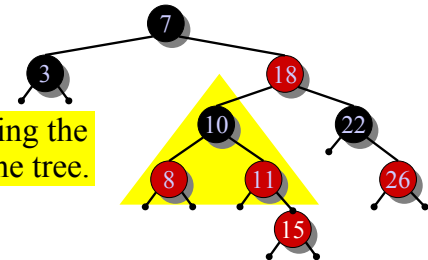


Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 4 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.



3/5/09

CS 3343 Analysis of Algorithms

24

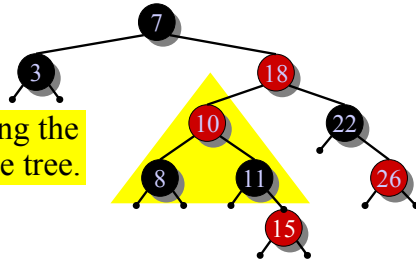


Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 4 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.

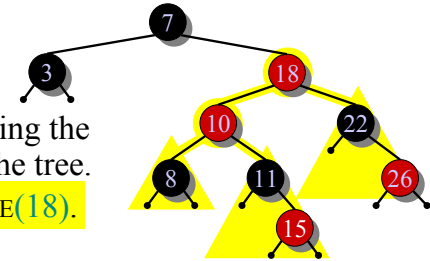


Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 4 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).

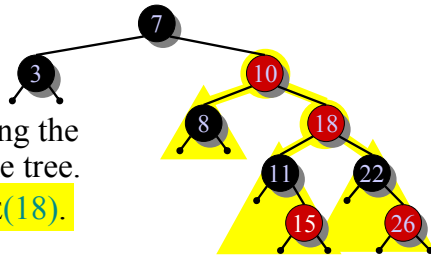


Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 4 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).

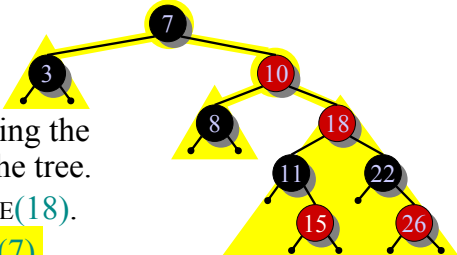


Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 4 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- LEFT-ROTATE(7).



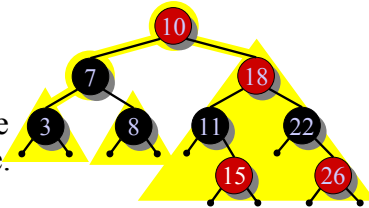


Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 4 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- LEFT-ROTATE(7)

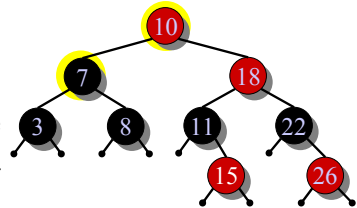


Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 4 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- LEFT-ROTATE(7) and recolor.

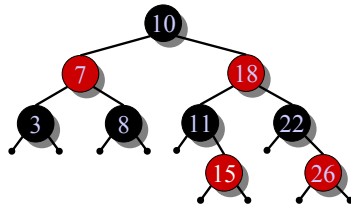


Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 4 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- LEFT-ROTATE(7) and recolor.



Pseudocode

```

RB-INSERT( $T, x$ )
  TREE-INSERT( $T, x$ )
   $color[x] \leftarrow RED$   ▷ only RB property 4 can be violated
  while  $x \neq root[T]$  and  $color[p[x]] = RED$ 
    do if  $p[x] = left[p[p[x]]]$ 
       then  $y \leftarrow right[p[p[x]]]$   ▷  $y =$  aunt/uncle of  $x$ 
          if  $color[y] = RED$ 
            then ⟨Case 1⟩
            else if  $x = right[p[x]]$ 
                 then ⟨Case 2⟩  ▷ Case 2 falls into Case 3
                 ⟨Case 3⟩
            else ⟨“then” clause with “left” and “right” swapped⟩
           $color[root[T]] \leftarrow BLACK$ 

```



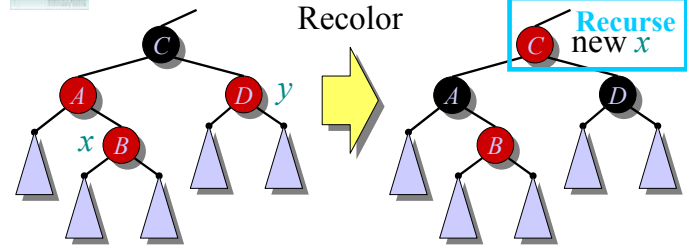

Graphical notation

Let \blacktriangle denote a subtree with a black root.

All \blacktriangle 's have the same black-height.



Case 1

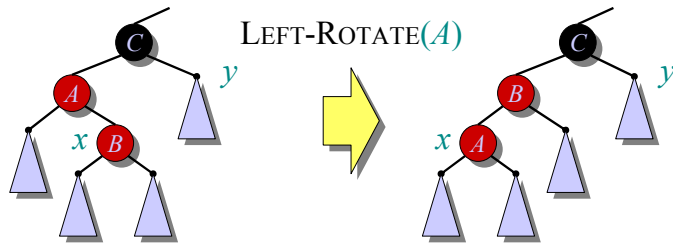


(Or, A 's children are swapped.) Push C 's black onto A and D , and recurse, since C 's parent may be red.

$p[x] = left[p[p[x]]]$
 $y = right[p[p[x]]]$
 $color[y] = RED$



Case 2

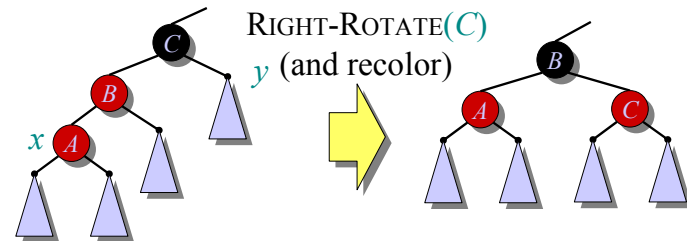


Transform to Case 3.

$p[x] = left[p[p[x]]]$
 $y = right[p[p[x]]]$
 $color[y] = BLACK$
 $x = right[p[x]]$



Case 3



Done! No more violations of RB property 4 are possible.

$p[x] = left[p[p[x]]]$
 $y = right[p[p[x]]]$
 $color[y] = BLACK$
 $x = left[p[x]]$



Analysis

- Go up the tree performing Case 1, which only recolors nodes.
- If Case 2 or Case 3 occurs, perform 1 or 2 rotations, and terminate.

Running time: $O(\log n)$ with $O(1)$ rotations.

RB-DELETE — same asymptotic running time and number of rotations as RB-INSERT (see textbook).



Pseudocode (part II)

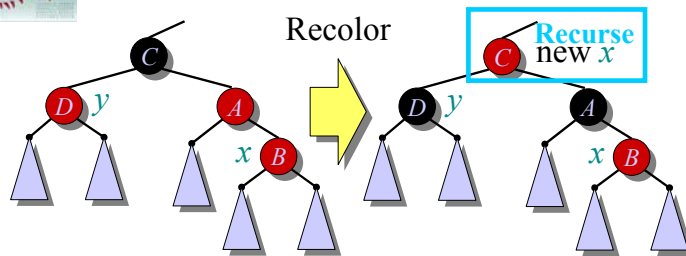
```

else (“then” clause with “left” and “right” swapped)
▷  $p[x] = \text{right}[p[p[x]]]$ 
then  $y \leftarrow \text{left}[p[p[x]]]$            ▷  $y = \text{aunt/uncle of } x$ 
  if  $\text{color}[y] = \text{RED}$ 
    then Case 1'
  else if  $x = \text{left}[p[x]]$ 
    then Case 2' ▷ Case 2' falls into Case 3'
    Case 3'
 $\text{color}[\text{root}[T]] \leftarrow \text{BLACK}$ 

```



Case 1'



(Or, A 's children are swapped.) Push C 's black onto A and D , and recurse, since C 's parent may be red.

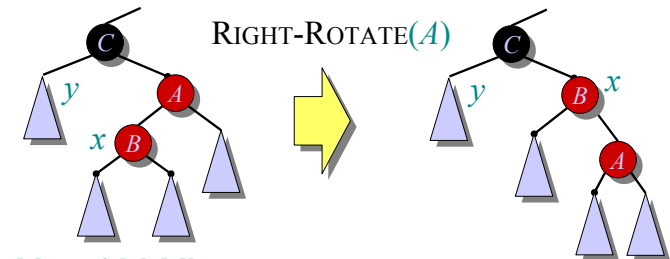
```

 $p[x] = \text{right}[p[p[x]]]$ 
 $y = \text{left}[p[p[x]]]$ 
 $\text{color}[y] = \text{RED}$ 

```



Case 2'



```

 $p[x] = \text{right}[p[p[x]]]$ 
 $y = \text{left}[p[p[x]]]$ 
 $\text{color}[y] = \text{BLACK}$ 
 $x = \text{left}[p[x]]$ 

```

Transform to Case 3'.



Case 3'

