Escuela Profesional de
Ciencia de la Computación
Algoritmos y Estructuras de Datos
2020-B

Red-Black Tree

M.Sc. Franci Suni Lopez

Universidad Nacional de San Agustín de Arequipa

Red-Black Trees

- "Balanced" binary search trees guarantee an ***O(lgn)*** running time
- Red-black-tree
  - Binary search tree with an additional attribute for its nodes: **color** which can be red or black
  - Constrains the way nodes can be colored on any path from the root to a leaf:

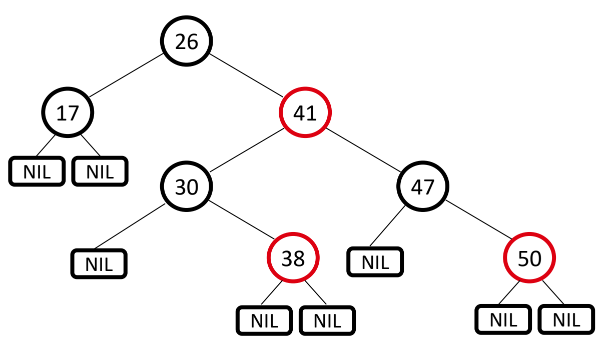
> Ensures that no path is more than twice as long as any other path $\Rightarrow$ the tree is balanced

Red-Black Trees

- Rudolf Bayer – 1972 – Symmetric Binary Tree.
- Leonidas Guibas, Robert Sedgewick – 1978 – A Dichromatic Framework for Balanced Trees

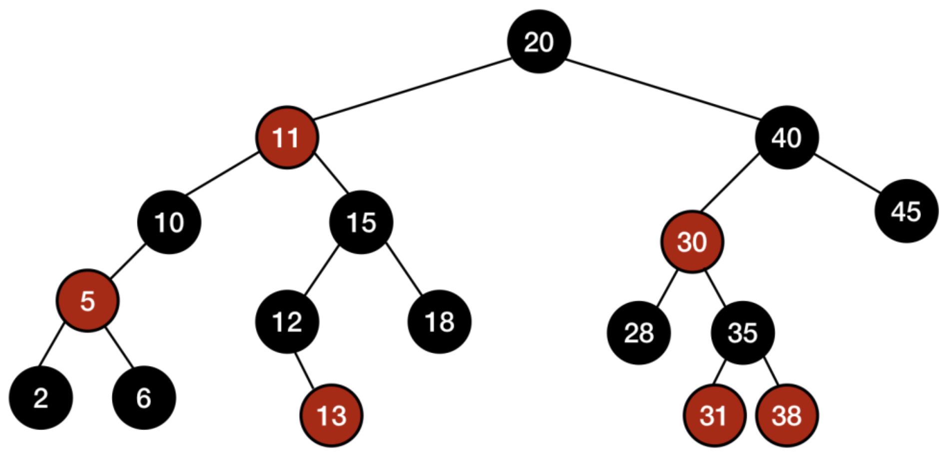

A DICHROMATIC FRAMEWORK FOR BALANCED TREES

Example: RED-BLACK-TREE



- For convenience we use a sentinel NIL[T] to represent all the NIL nodes at the leafs
  - NIL[T] has the same fields as an ordinary node
  - *Color[NIL[T]] = BLACK*
  - The other fields may be set to arbitrary values

Any difference with AVL?

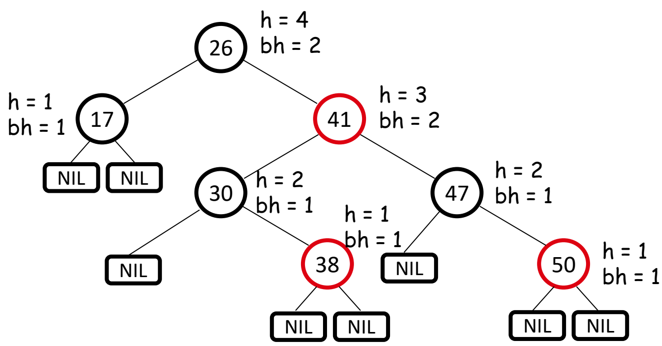


Red-Black-Trees Properties

(**Satisfy the binary search tree property**)

1. Every **node** is either red or black
2. The **root** is black
3. Every **leaf** (NIL) is black
4. If a node is red, then both its children are black
   - No two consecutive red nodes on a simple path from the root to a leaf
5. For each node, all paths from that node to descendant leaves contain the same number of black nodes

Black-Height of a Node



- **Height of a node:** the number of edges in the **longest** path to a leaf
- **Black-height** of a node **x: bh(x)** is the number of black nodes (including NIL) on the path from **x** to a leaf, not counting **x**

Most important property of Red-Black-Trees

A red-black tree with **n** internal nodes has height at most **2lg(n + 1)**

- Need to prove two claims first …

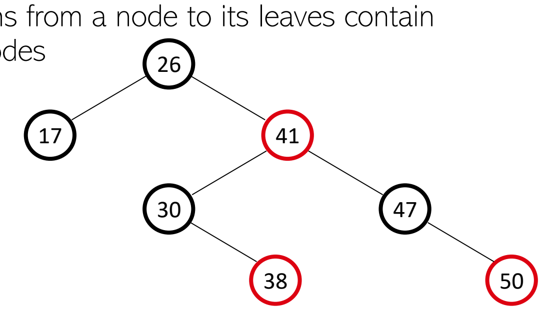
Operations on Red-Black-Trees

- The non-modifying binary-search-tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in $O(h)$ time
  - They take $O(\lg n)$ time on red-black trees
- What about TREE-INSERT and TREE-DELETE?
  - They will still run on $O(\lg n)$
  - We have to guarantee that the modified tree will still be a red-black tree

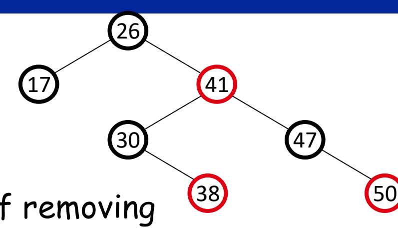
INSERT

INSERT: what color to make the new node?

- Red? Let's insert 35!
  - Property 4 is violated: if a node is red, then both its children are black
- Black? Let's insert 14!
  - Property 5 is violated: all paths from a node to its leaves contain the same number of black nodes



DELETE



DELETE: what color was the node that was removed? Black?

1. Every **node** is either red or black *OK!*
2. The **root** is black

   Not OK! If removing the root and the child that replaces it is **red**
3. Every **leaf** (NIL) is black *OK!*
4. If a node is red, then both its children are black

   Not OK! Could create two red nodes in a row

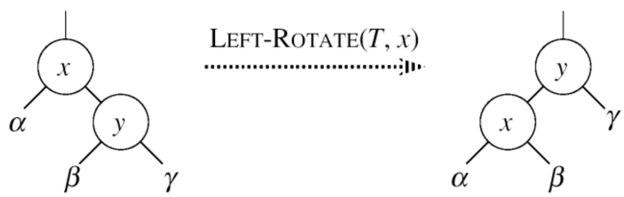
   Not OK! Could change the black heights of some nodes
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

Rotations

- Operations for re-structuring the tree after insert and delete operations on red-black trees
- Rotations take a red-black-tree and a node within the tree and:
  - Together with some node re-coloring they help restore the red-black-tree property
  - Change some of the pointer structure
  - Do not change the binary-search tree property
- Two types of rotations:
  - Left & right rotations

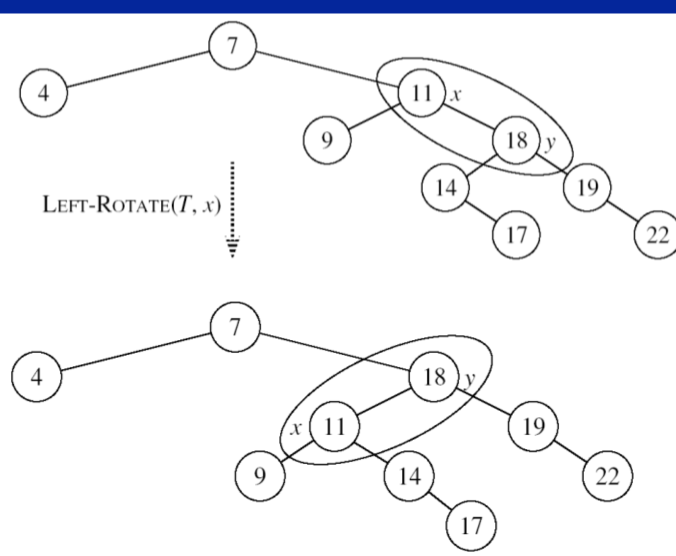
Left Rotations

- Assumptions for a left rotation on a node **x**:
  - The right child of **x** (**y**) is not NIL



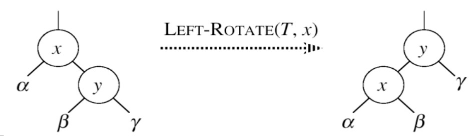
- Idea:
  - Pivots around the link from **x** to **y**
  - Makes **y** the new root of the subtree
  - **x** becomes **y**'s left child
  - **y**'s left child becomes **x**'s right child

Example: LEFT-ROTATE



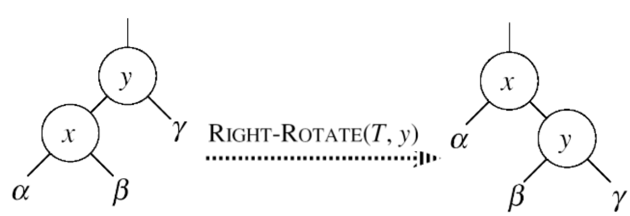
LEFT-ROTATE(T, x)

```
y ← right[x]                        ▶ Set y
right[x] ← left[y]                  ▶ y's left subtree becomes x's right subtree
if left[y] ≠ NIL
   then p[left[y]] ← x              ▶ Set the parent relation from left[y] to x
p[y] ← p[x]                         ▶ The parent of x becomes the parent of y
if p[x] = NIL
   then root[T] ← y
   else if x = left[p[x]]
           then left[p[x]] ←
           else right[p[x]] ← y
left[y] ← x                         ▶ Put x on y's left
p[x] ← y                            ▶ y becomes x's parent
```



Right Rotations

- Assumptions for a right rotation on a node x:
  - The left child of y (x) is not NIL



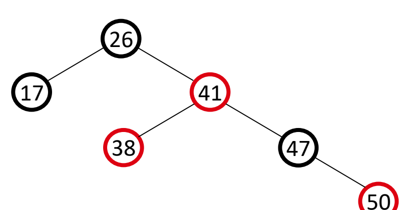
- Idea:
  - Pivots around the link from y to x
  - Makes x the new root of the subtree
  - y becomes x's right child
  - x's right child becomes y's left child

Insertion

- Goal:
  - Insert a new node z into a red-black-tree
- Idea:
  - Insert node z into the tree as for an ordinary binary search tree
  - Color the node red
  - Restore the red-black-tree properties
    - Use an auxiliary procedure RB-INSERT-FIXUP

RB Properties Affected by Insert

1. Every **node** is either red or black — OK!
2. The **root** is black — If z is the root ⇒ not OK
3. Every **leaf** (NIL) is black — OK!
4. If a node is red, then both its children are black — If p(z) is red ⇒ not OK: z and p(z) are both red
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes — OK!



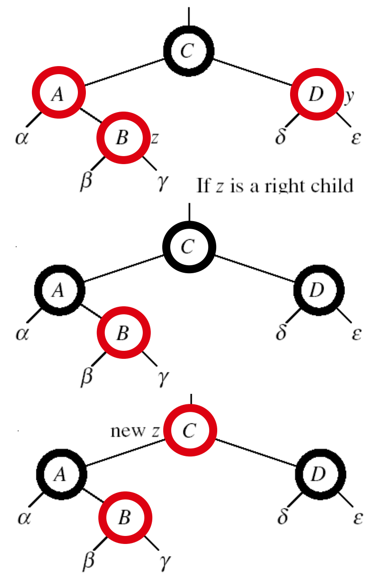
RB-INSERT-FIXUP – Case 1

z's "uncle" (y) is red

Idea: (z is a right child)

- p[p[z]] (z's grandparent) must be black: z and p[z] are both red

- Color p[z] black
- Color y black
- Color p[p[z]] red
- z = p[p[z]]
  - Push the "red" violation up the tree



25

RB-INSERT-FIXUP – Case 1

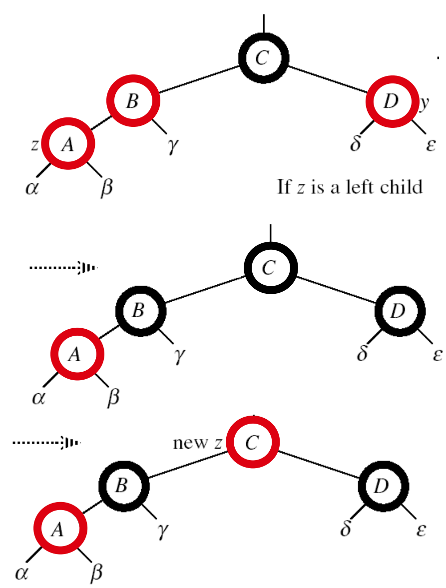
z's "uncle" (y) is red

Idea: (z is a left child)

- p[p[z]] (z's grandparent) must be black: z and p[z] are both red

- color p[z] ← black
- color y ← black
- color p[p[z]] ← red

- z = p[p[z]]
  - Push the "red" violation up the tree

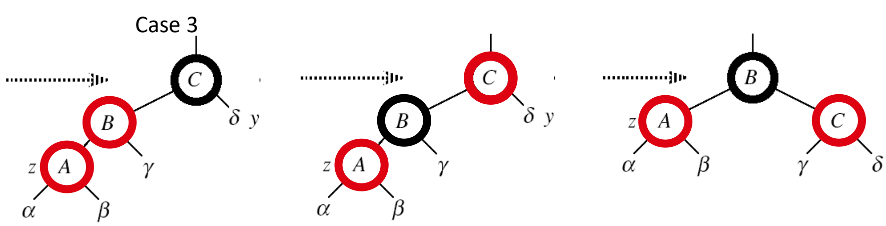


26

RB-INSERT-FIXUP – Case 3

Case 3:
- z's "uncle" (y) is **black**
- z is a left child

Idea:
- color p[z] ← **black**
- color p[p[z]] ← **red**
- RIGHT-ROTATE(T, p[p[z]])
- No longer have 2 reds in a row
- p[z] is now black



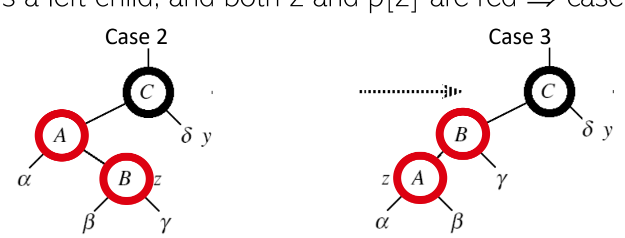
27

RB-INSERT-FIXUP – Case 2

Case 2:
- z's "uncle" (y) is **black**
- z is a right child

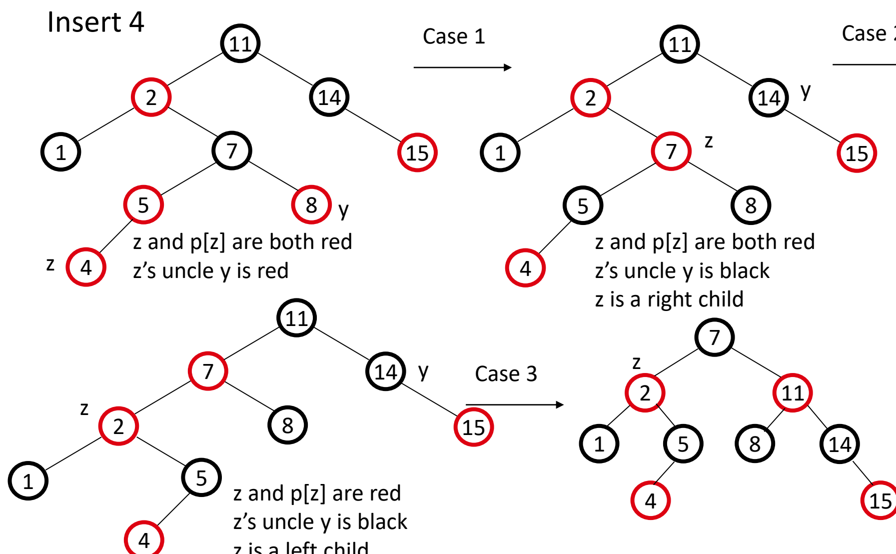
Idea:
- z ← p[z]
- LEFT-ROTATE(T, z)

⇒ now z is a left child, and both z and p[z] are red ⇒ case 3



28

Example



Analysis of RB-INSERT

- Inserting the new element into the tree ***O(lgn)***
- RB-INSERT-FIXUP
  - The while loop repeats only if CASE 1 is executed
  - The number of times the while loop can be executed is ***O(lgn)***
- Total running time of RB-INSERT: ***O(lgn)***

Red-Black Trees - Summary

- Operations on red-black-trees:
  - SEARCH $O(h)$
  - PREDECESSOR $O(h)$
  - SUCCESOR $O(h)$
  - MINIMUM $O(h)$
  - MAXIMUM $O(h)$
  - INSERT $O(h)$
  - DELETE $O(h)$
- Red-black-trees guarantee that the height of the tree will be $O(lgn)$

AVL vs Red-Black

	AVL	Red-Black
- data		
Search	Faster, lower height	
Insert		Faster, in average less rotations
Remove		Faster, in average less rotations
+ data		
Search	Faster, lower height	
Insert	Faster, problems with the search	
Remove	Faster, in average less rotations	Faster than AVL in the worst case

Homework

- Segment Tree.
- Interval Tree.
- 3 points in class participation.
- Problems or limitations with balanced trees -> AVL and Red-Black

School of Computer Science

Thanks!

Questions?

M.Sc. Franci Suni Lopez
fsunilo@unsa.edu.pe