CS 457, Fall 2016

Drexel University, Department of Computer Science

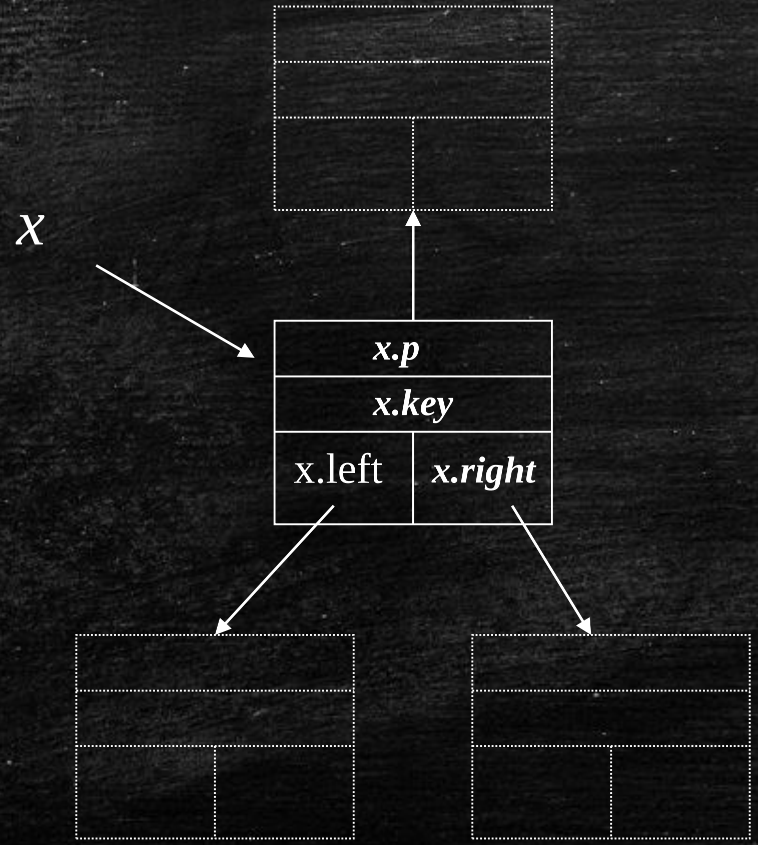
Lecture 10

Today's Lecture

- Red-Black Trees

Binary Search Trees

- Each node ***x*** in a binary search tree (BST) contains:



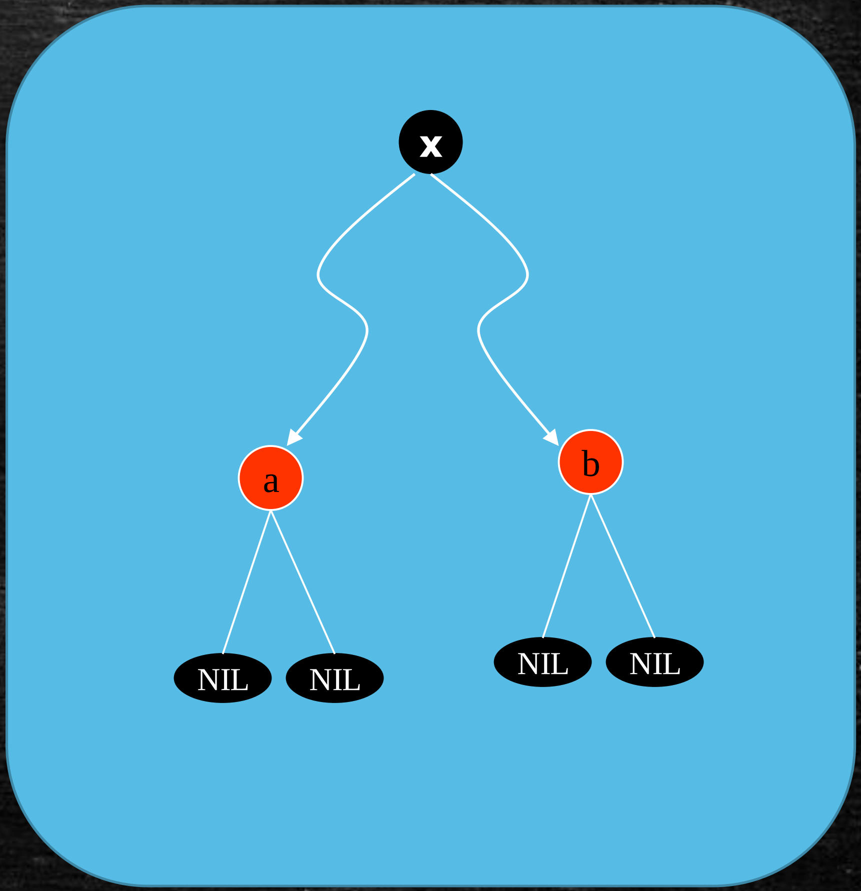
- ***x.key*** - The value stored at *x*.
- ***x.left*** - Pointer to left child of ***x.***
- ***x.right*** - Pointer to right child of ***x.***
- ***x.p*** - Pointer to parent of ***x.***

Red-Black Trees

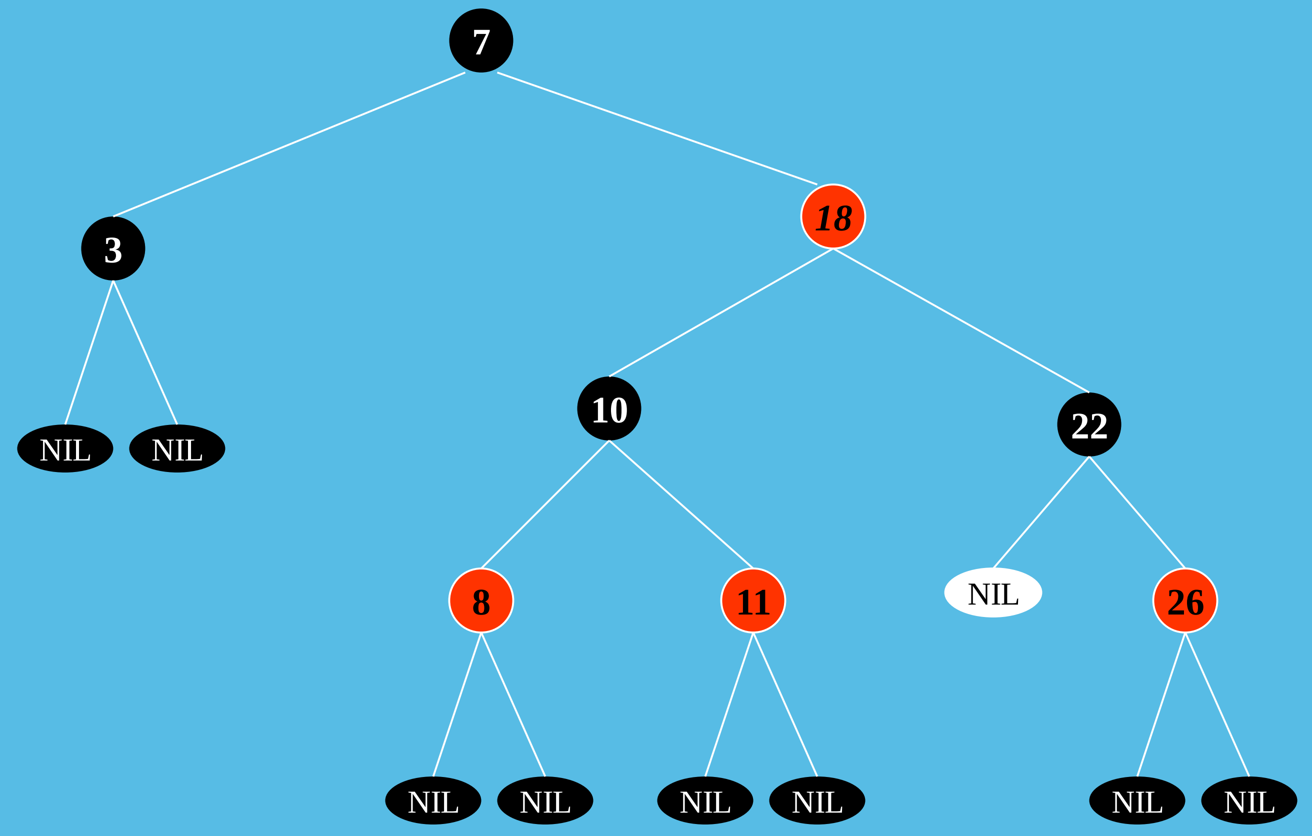
- They are balanced search trees (their height is $O(\log n)$)
- Most of the search and update operations on these trees take $O(\log n)$ time
- The structure is well balanced, i.e., each subtree is a balanced search tree.

Red-Black Trees

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.
5. All paths from a node ***x*** to a leaf have same number of black nodes (Black-Height(***x***))

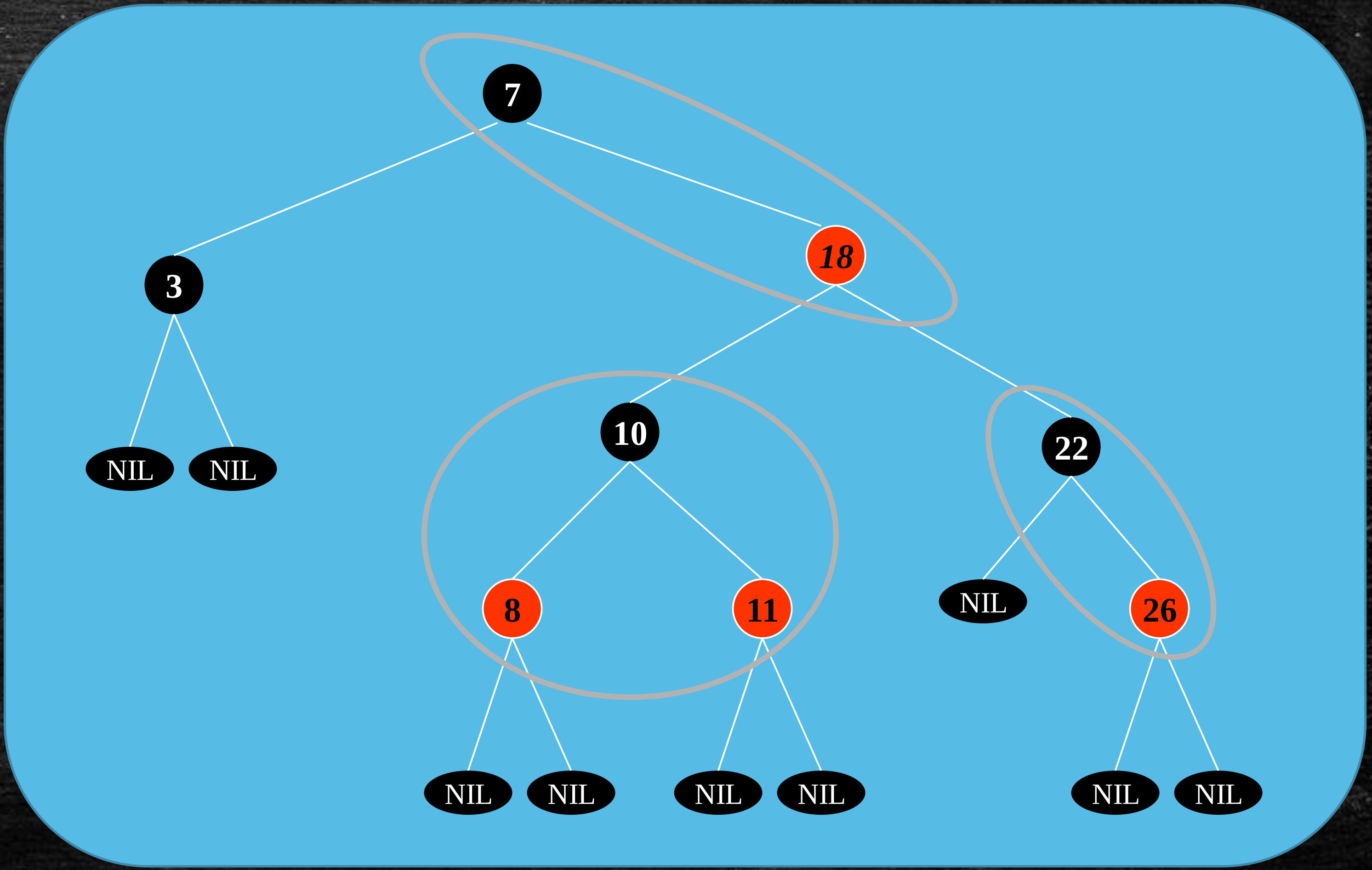


Example



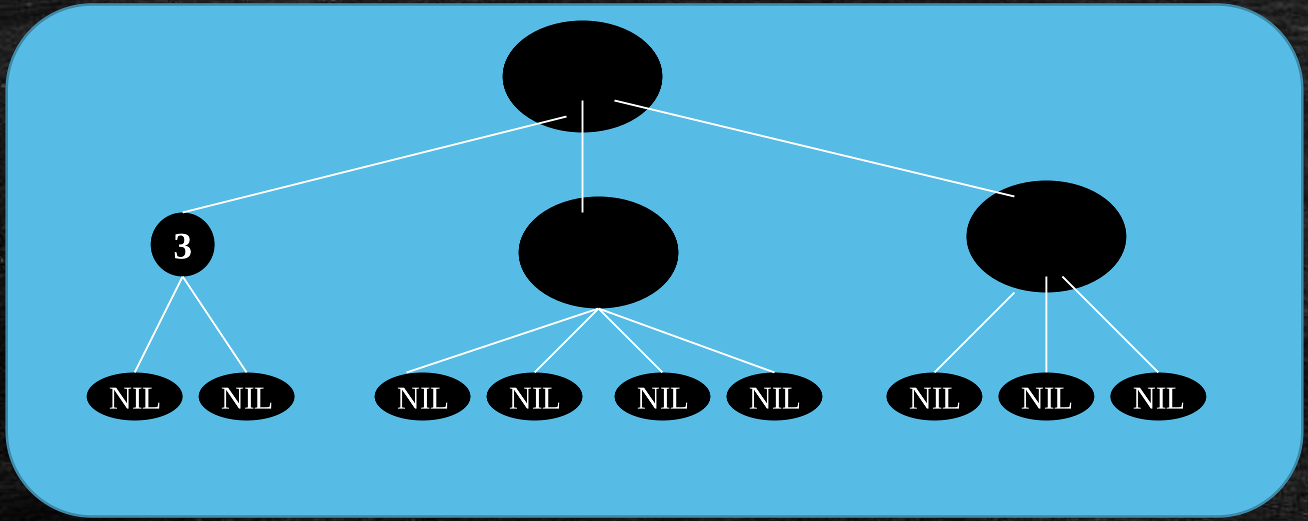
Height

- A red-black tree with ***n*** keys has height at most 2lg(***n***+1).
- Proof (Intuition): Merge the red nodes into their parents



Proof

- Produces a tree with nodes having 2,3, or 4 nodes



- Height h' of new tree is black height of original tree

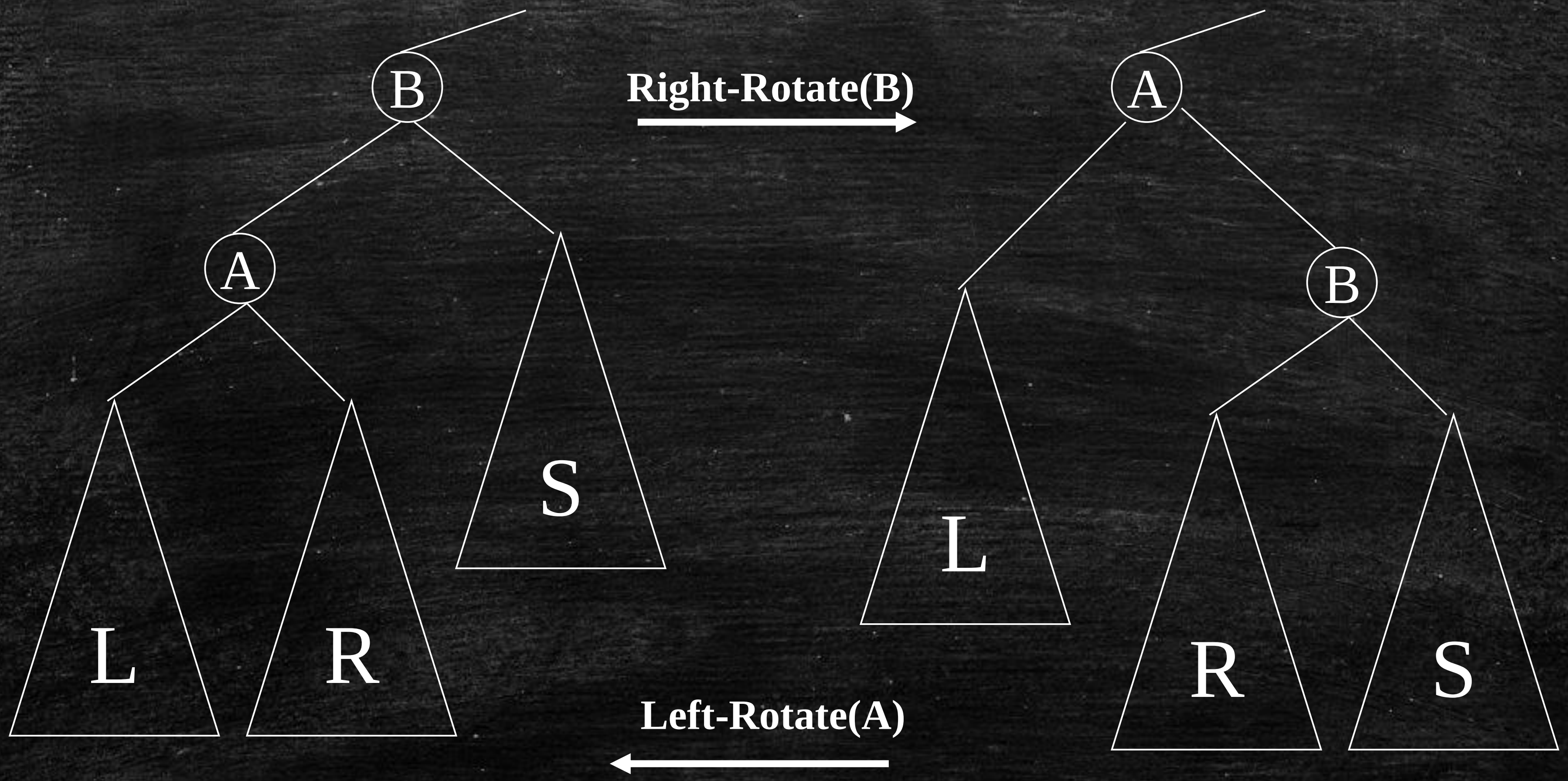
Proof

- Lemma: The subtree rooted at any node x of a red-black tree contains **at least $2^{bh(x)} - 1$ internal nodes**

Prove this using induction…

- The black height of the root must be at least h/2
- Therefore, $\boldsymbol{n \geq 2^{h/2} - 1}$
- Which implies that $\boldsymbol{h \leq \log(n+1)}$

Rotation

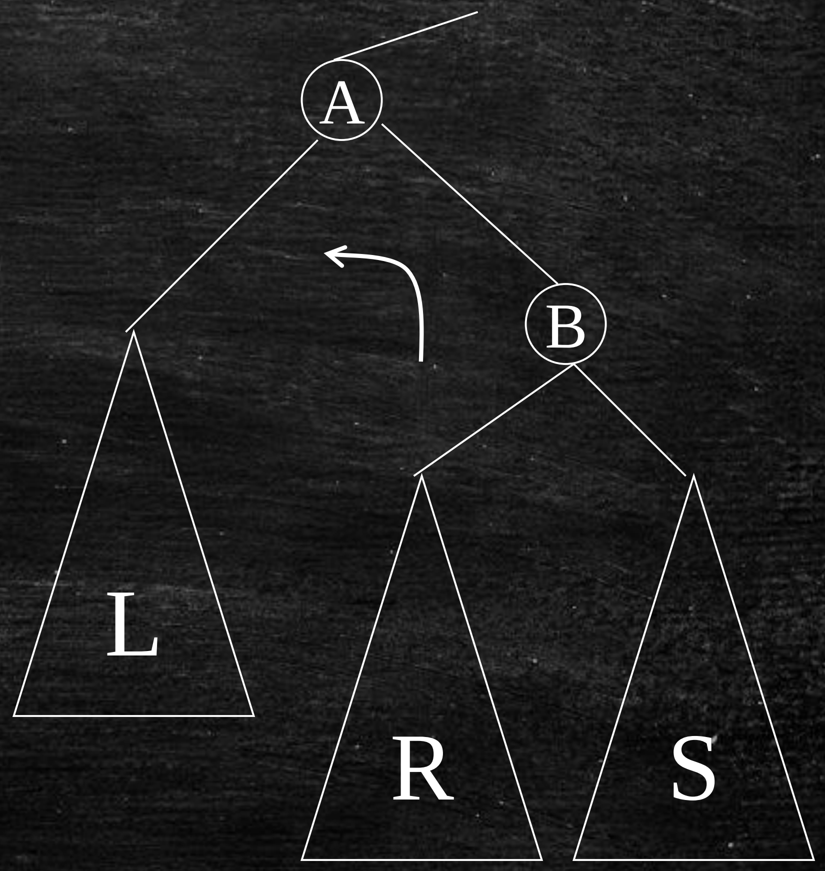


Rotation

- Rotation is the basic operation for maintaining balanced trees
- Maintains inorder key ordering:
  - For all $a \in L$, $b \in R$, $c \in S$, we have $a \leq b \leq c$

Left Rotation:

- Depth(**L**) increases by 1
- Depth(**R**) stays the same
- Depth(**S**) decreases by 1
- Takes $O(1)$



Rotation

Left-Rotate(***T, x***)

y = ***x.right***

x.right = ***y.left***

if ***y.left !***= NIL then

y.left .p = ***x***

y.p = ***x.p***

if ***x.p*** = NIL then ***T.root*** = ***y***

else if ***x*** = ***x.p.left*** then

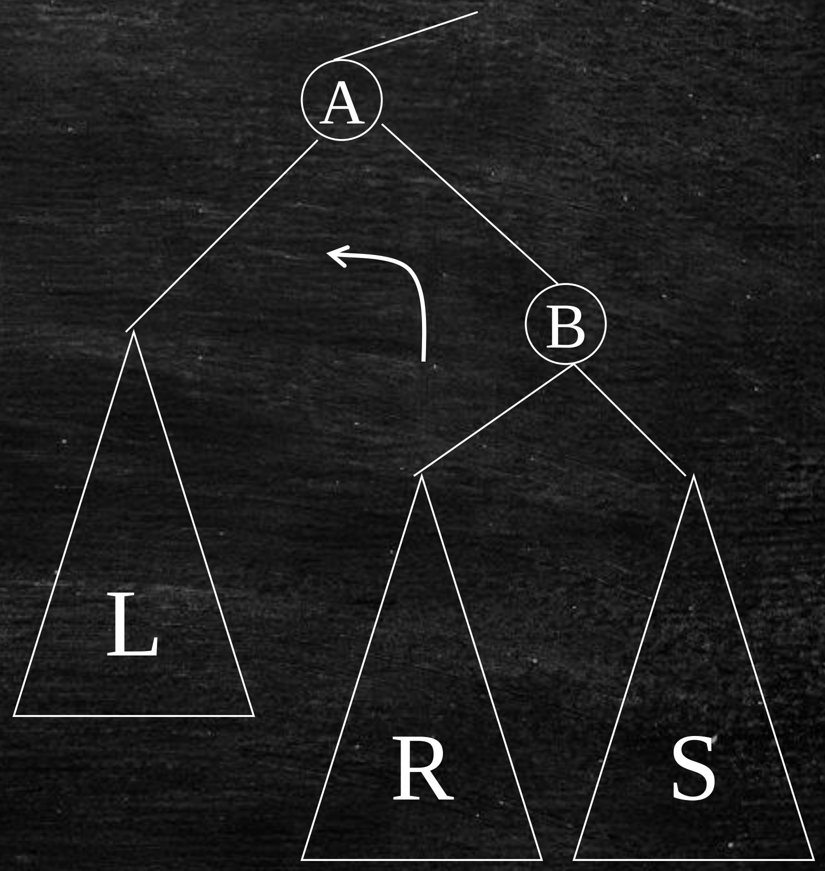
x.p.left = ***y***

else

x.p.right = ***y***

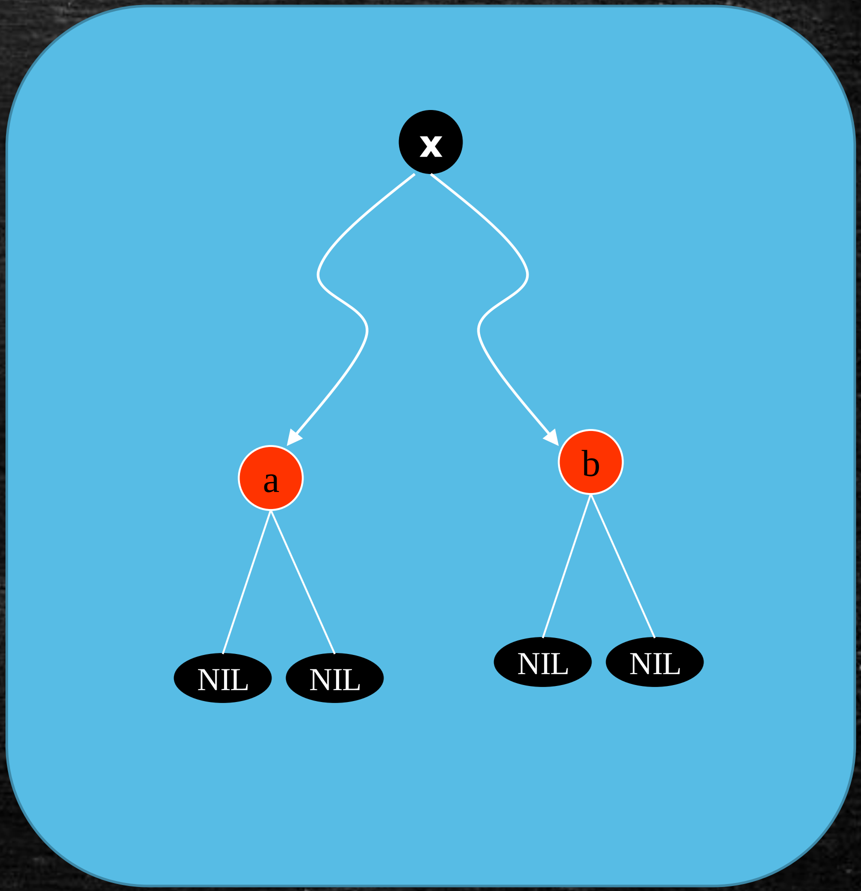
y.left = ***x***

x.p = ***y***



Red-Black Trees

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.
5. All paths from a node ***x*** to a leaf have same number of black nodes (Black-Height(***x***))



Red-Black Insertion

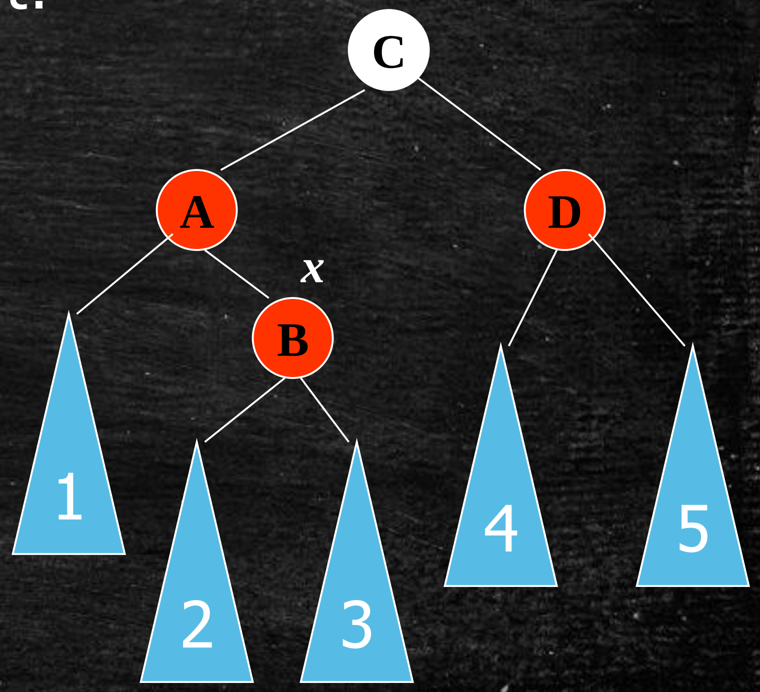
- Insert ***x*** into tree
- Color ***x*** red.
- Red-black property 1 still holds:
  - since color of x is red
- Red-Black property 3 still holds:
  - since inserted node has NILs for children.
- Red-black property 5 still holds:
  - since ***x*** replaces a black NIL and has NIL children.

Red-Black Insertion

- Two types of violations possible:
  - If ***x.p*** is red, then property 4 is violated.
  - If color of root is red, then property 2 is violated.
- To correct violation of property 4, we move violation up in tree until it can be fixed.
- No new violations will be introduced during this process.
  - root can become red at some point, which will be fixed using the same procedure.
- For each iteration, there are six possible cases.
  - 3 of these cases are symmetric of the other.

Insertion Case 1

- *x*'s parent is the left child of *x*'s grandparent.
- *x*'s uncle is Red.
- Then
  - *x.p.color*= Black
  - *(x.p.p.right).color*= Black
  - *(x.p.p).color*= Red
  - *x=x.p.p*



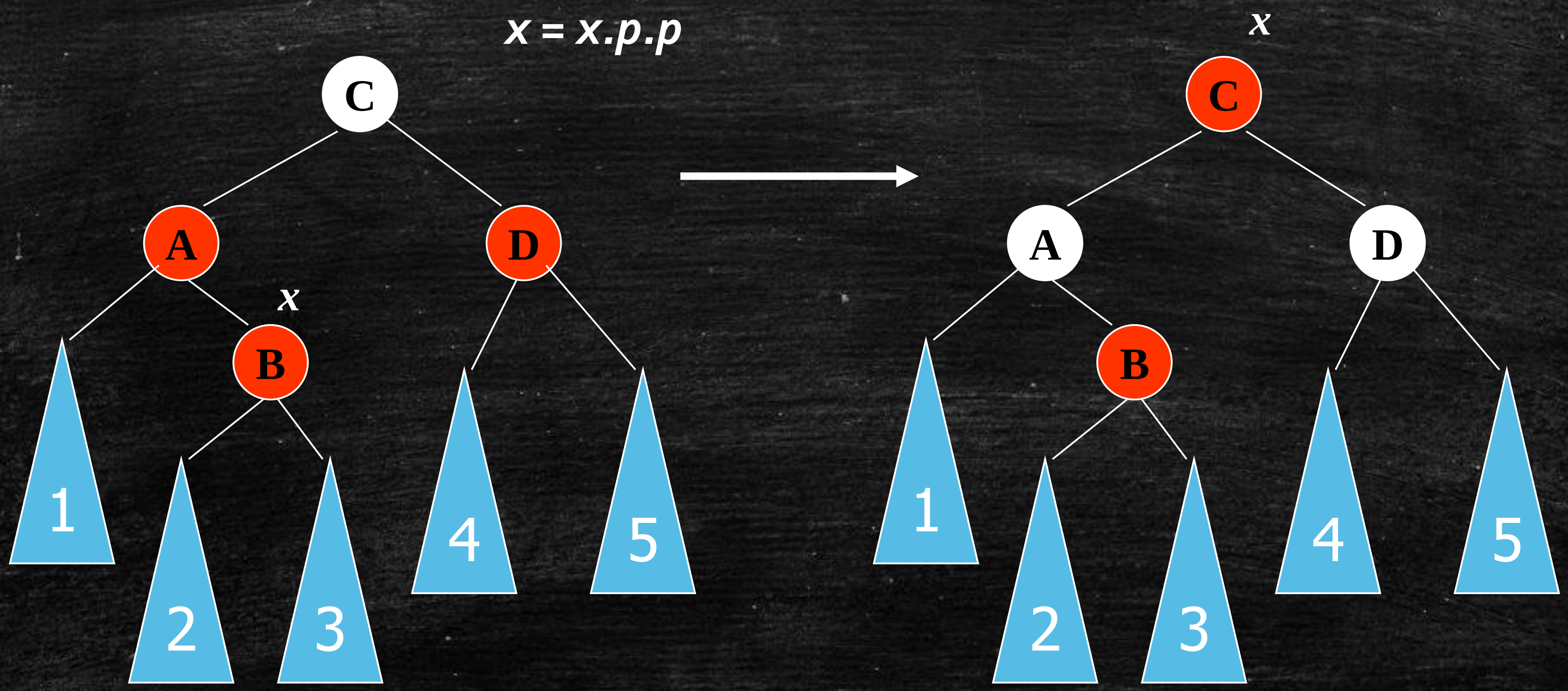
Insertion Case 1

(x.p).color = Black

(x.p.p.right).color = Black

(x.p.p).color = Red

x = *x.p.p*



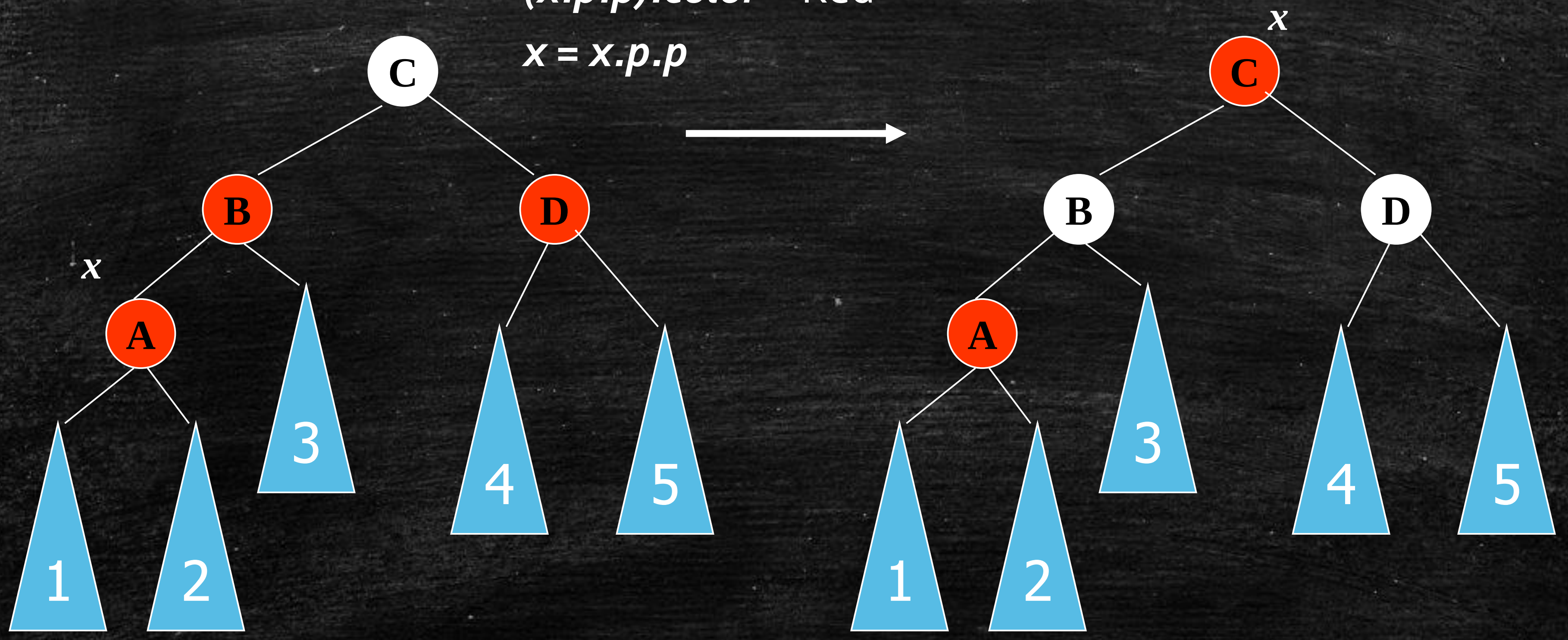
Insertion Case 1

(x.p).color = Black

(x.p.p.right).color = Black

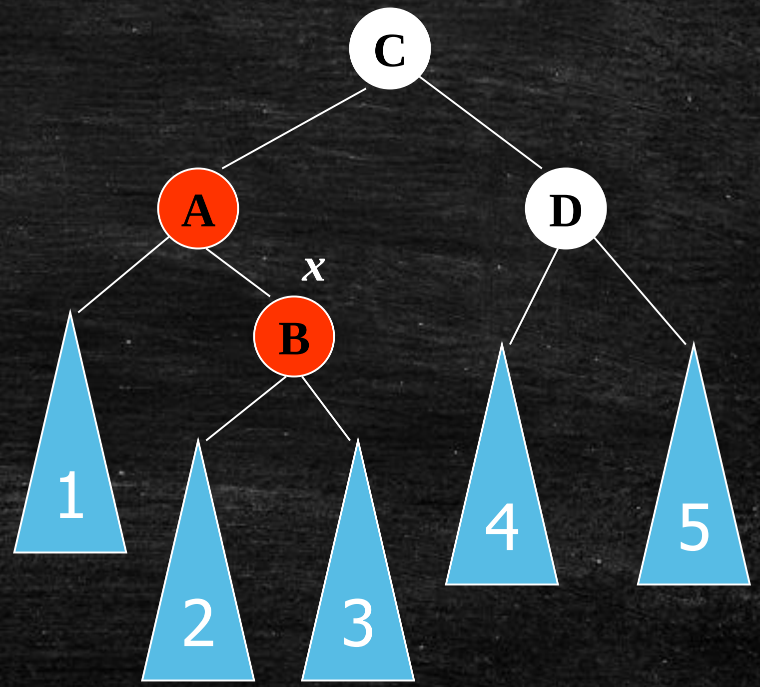
(x.p.p).color = Red

x = *x.p.p*



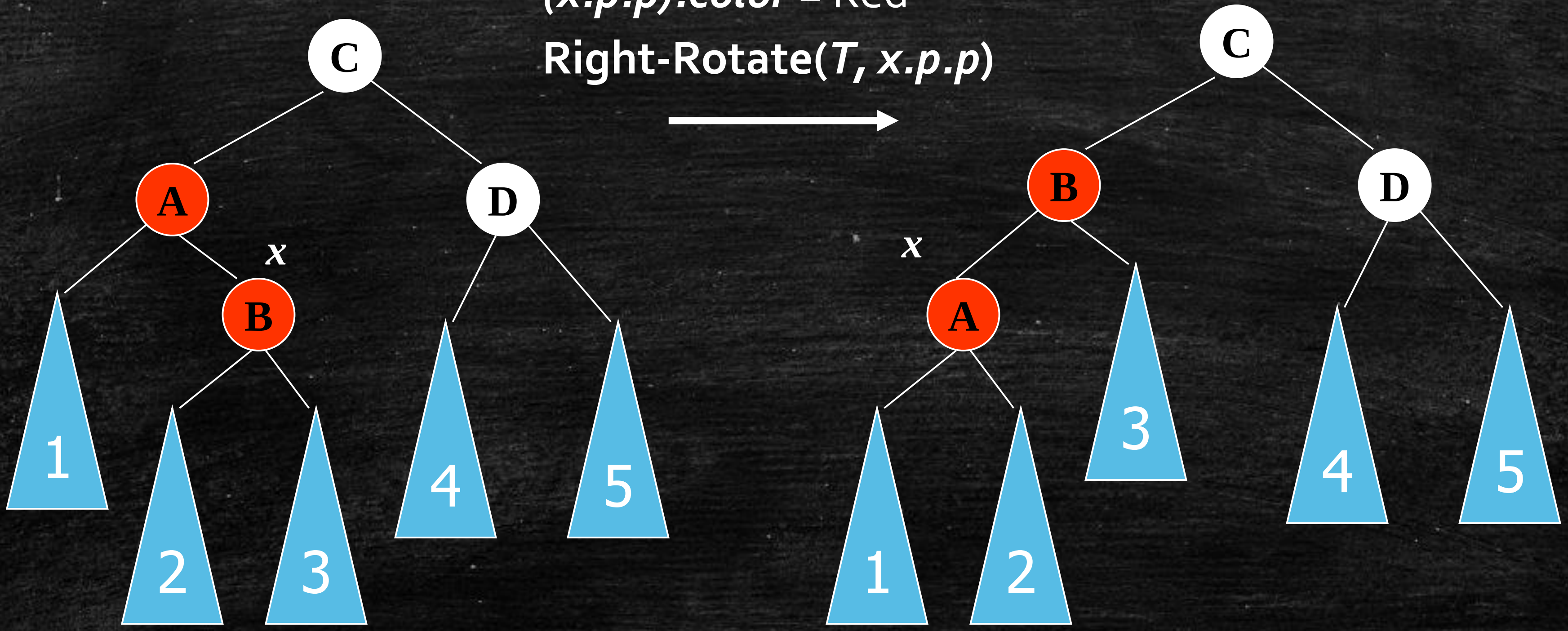
Insertion Case 2

- ***x*'s parent is the left child of *x*' s grandparent**
- ***x*'s uncle is Black**
- ***x*** is right child of ***x.p***
- Then
  - ***x*** = ***x.p***
  - **Left-Rotate(*T*, *x*)**
  - ***(x.p).color*** = Black
  - ***(x.p.p).color*** = Red
  - **Right-Rotate(*T*, *x.p.p*)**



Insertion Case 2

$\{$ $x = x.p$
Left-Rotate(*T*, *x*) $\}$
(x.p).color = Black
(x.p.p).color = Red
Right-Rotate(*T*, *x.p.p*)



Insertion Case 2

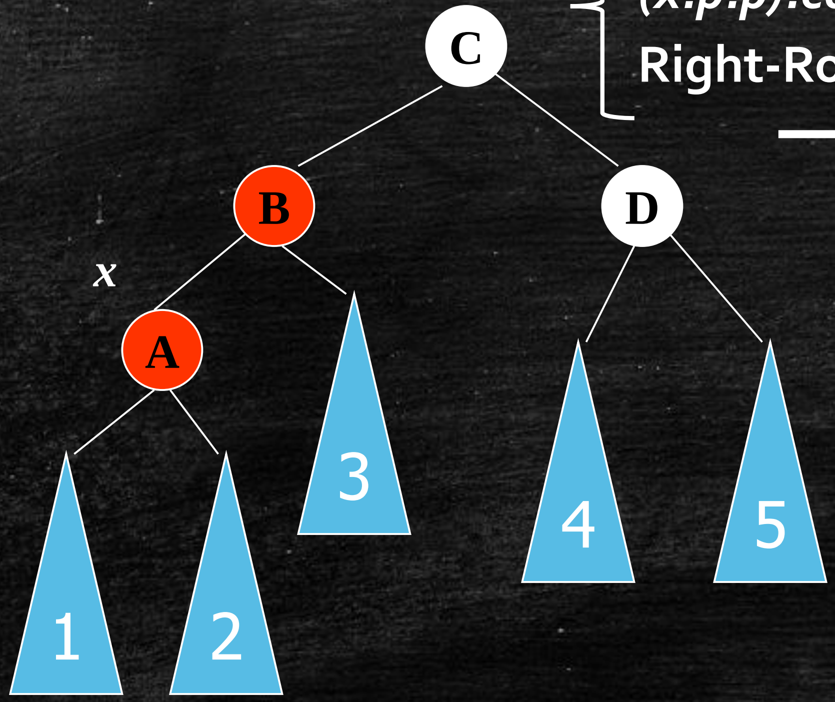
x = *x.p*

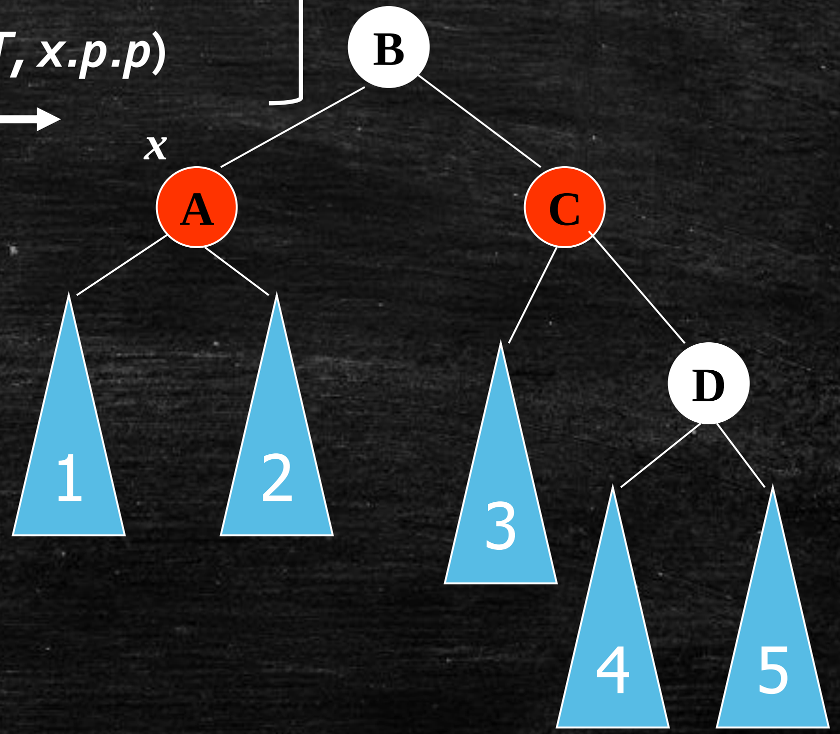
Left-Rotate(*T*, *x*)

(x.p).color = Black

(x.p.p).color = Red

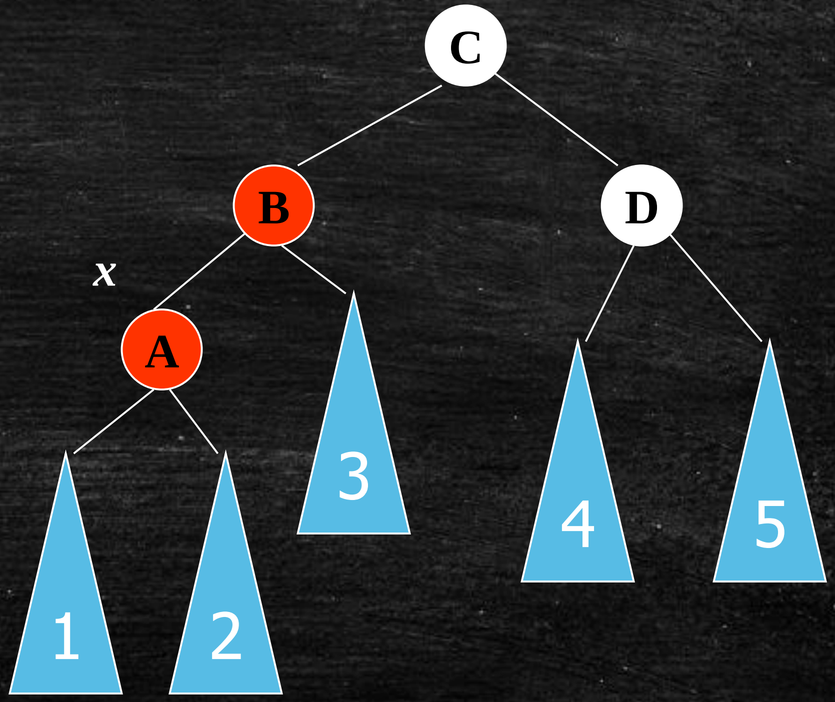
Right-Rotate(*T*, *x.p.p*)





Insertion Case 3

- ***x***'s parent is the left child of ***x***' s grandparent
- ***x***'s uncle is Black
- ***x*** is the left child of ***x.p***
- Then
  - ***(x.p).color*** = Black
  - ***(x.p.p).color*** = Red
  - **Right-Rotate(*T*, *x.p.p*)**

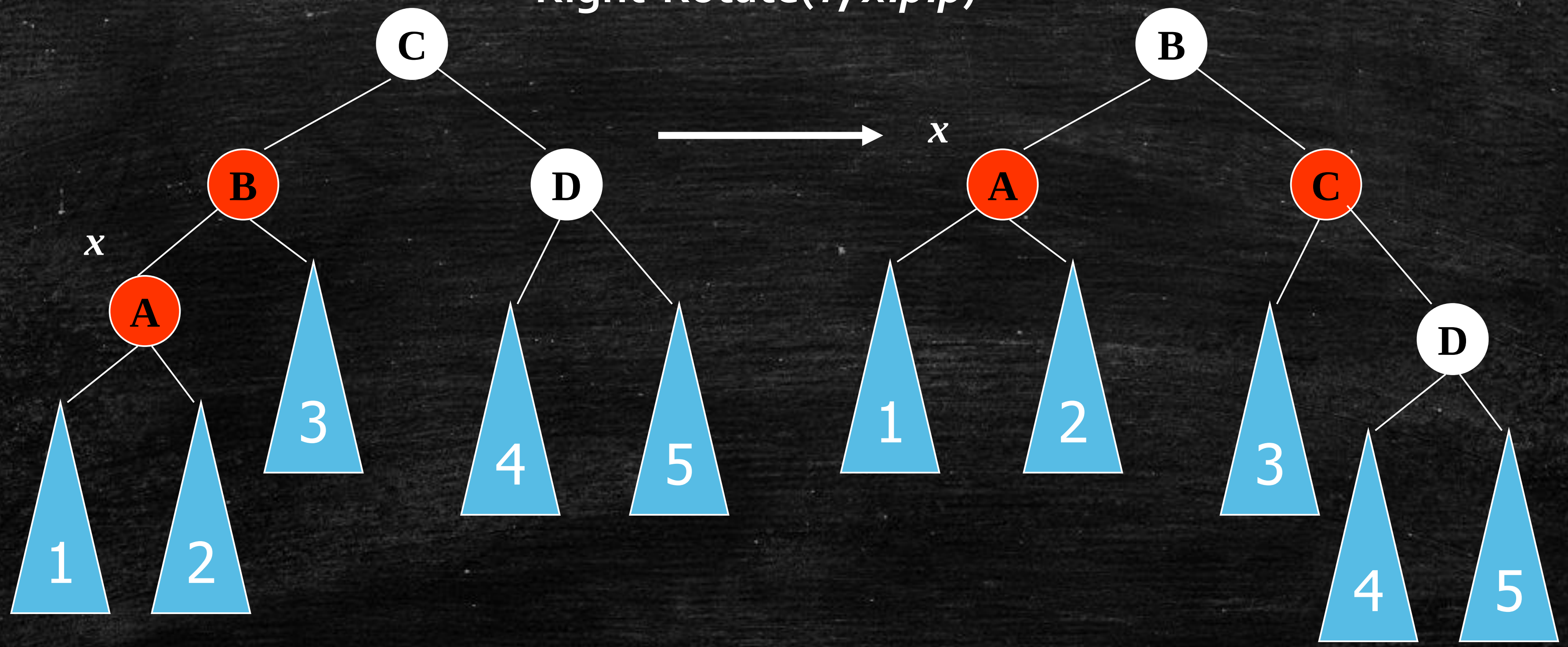


Insertion Case 3

(x.p).color = Black

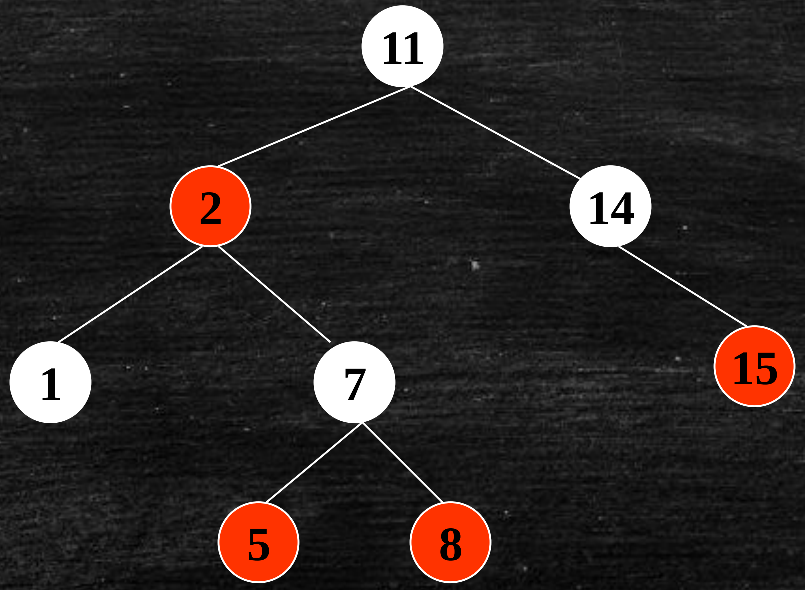
(x.p.p).color = Red

Right-Rotate(*T*, *x.p.p*)



Example

- Use R-B Insert to insert element with key 4.



Example 2

- Show the red-black tree that results after successively inserting the keys ***41, 38, 31, 12, 19, 8*** into an initially empty red-black tree

Deletion

- Similar idea with insertion
- A bit more complicated
- Read text book

Second Problem Set

1. (20 pts) Solve problem 4-3 on page 108 of your textbook.
2. (15 pts) You are given a set S of n integers, as well as one more integer v. Design an algorithm that determines whether or not there exist two distinct elements $x, y \in S$ such that $x + y = v$. Your algorithm should run in time $O(n \log n)$, and it should return (x, y) if such elements exist and (NIL, NIL) otherwise. Prove the worst case running time bound and the correctness of the algorithm.
3. (15 pts) Prove tight **worst-case** asymptotic upper bounds for the following recurrence equations that satisfy $T(n) = 1$ for $n \leq 2$, and depend on a variable $q \in [0, n/4]$:
   (a) $T(n) = T(n - 2q - 1) + T(3q/2) + T(q/2) + \Theta(1)$
   (b) $T(n) = T(n - q - 1) + T(n/2 - q) + \Theta(n)$
   (c) $T(n) = T(n - q - 1) + T(3q) + \Theta(n)$
4. (10 pts) Consider the following silly randomized variant of binary search. You are given a sorted array A of n integers and the integer v that you are searching for is chosen uniformly at random from A. Then, instead of comparing v with the value in the middle of the array, the randomized binary search variant chooses a random number r from 1 to n and it compares v with $A[r]$. Depending on whether v is larger or smaller, this process is repeated recursively on the left sub-array or the right sub-array, until the location of v is found. Prove a tight bound on the expected running time of this algorithm.
5. (10 pts) Can the master method be applied to the recurrence $T(n) = 4T(n/2) + n^2 \log n$? Why or why not? Give an asymptotic upper bound for this recurrence.

Second Problem Set

6. (10 pts) Use a recursion tree to give an asymptotically tight solution to the recurrence:

   (a) $T(n) = T(n-a) + T(a) + cn$, where $a \geq 1$ and $c > 0$ are constants.

   (b) $T(n) = T(an) + T((1-a)n) + cn$, where $a \in (0, 1)$ and $c > 0$ are constants.

7. (10 pts) For the following recurrences, find the bound implied by the master theorem. Then, try to prove the same bound using the substitution method. If your initial approach fails, show how it fails, and try subtracting off a lower-order term to make it work:

   (a) $T(n) = 4T(n/3) + n$

   (b) $T(n) = 4T(n/2) + n$

8. (5 pts) Solve the recurrence $T(n) = 3T(\sqrt{n}) + \log n$ by making a change of variable. Your solution should be asymptotically tight. Do not worry about whether values are integral.

9. (5 pts) Using proof by induction, show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.