Lecture 9: Balanced Binary Search Tree

WHAT IS BINARY SEARCH TREE ?

- A *rooted binary tree*.

WHAT IS BINARY SEARCH TREE ?

- A *rooted binary tree*.
- Internal nodes each store a *key*

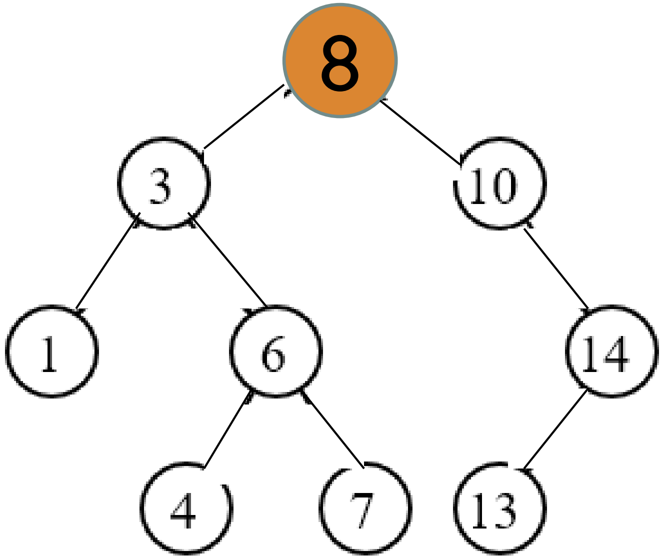
WHAT IS BINARY SEARCH TREE ?

- A *rooted binary tree*.
- Internal nodes each store a *key*
- Each have two distinguished sub-trees,
commonly denoted *left* and *right*

WHAT IS BINARY SEARCH TREE ?

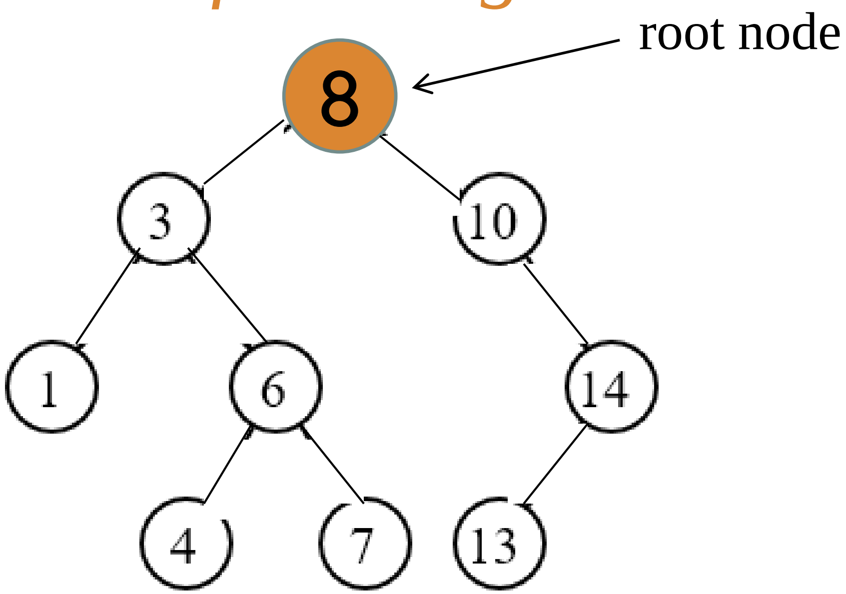
- A *rooted binary tree*.
- Internal nodes each store a *key*
- Each have two distinguished sub-trees,

commonly denoted *left* and *right*



WHAT IS BINARY SEARCH TREE ?

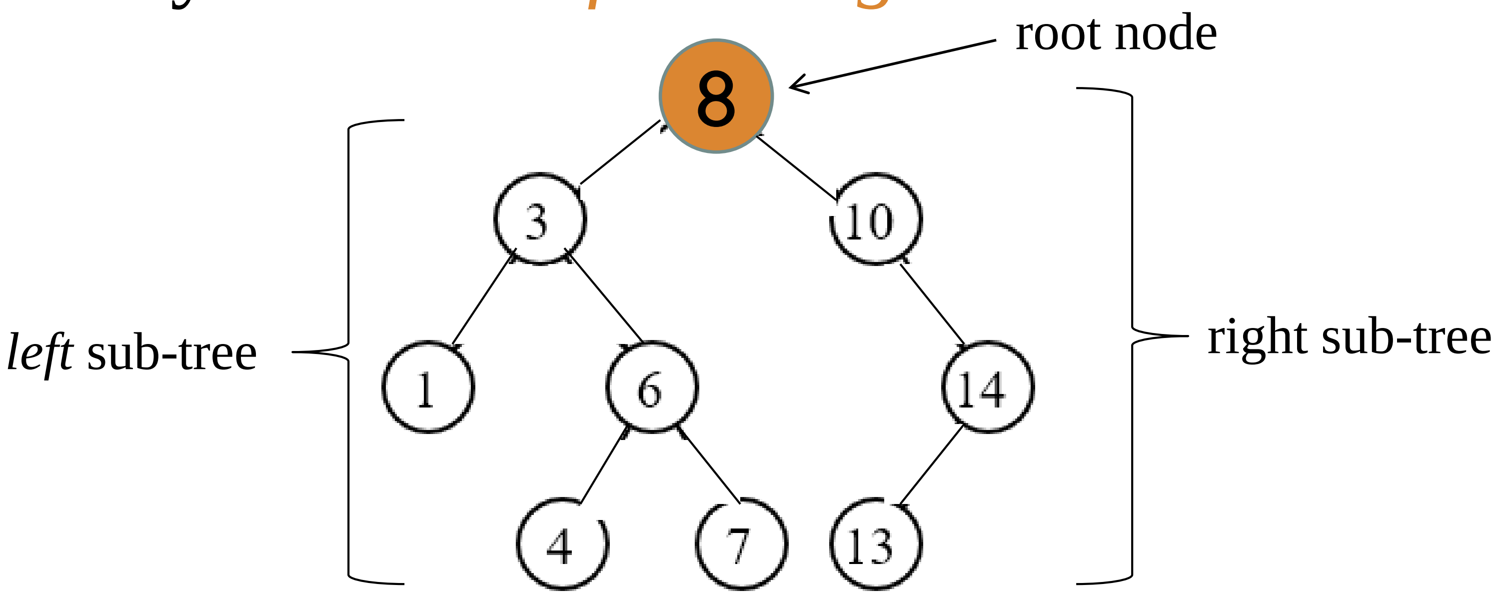
- A *rooted binary tree*.
- Internal nodes each store a *key*
- Each have two distinguished sub-trees,
commonly denoted *left* and *right*



WHAT IS BINARY SEARCH TREE ?

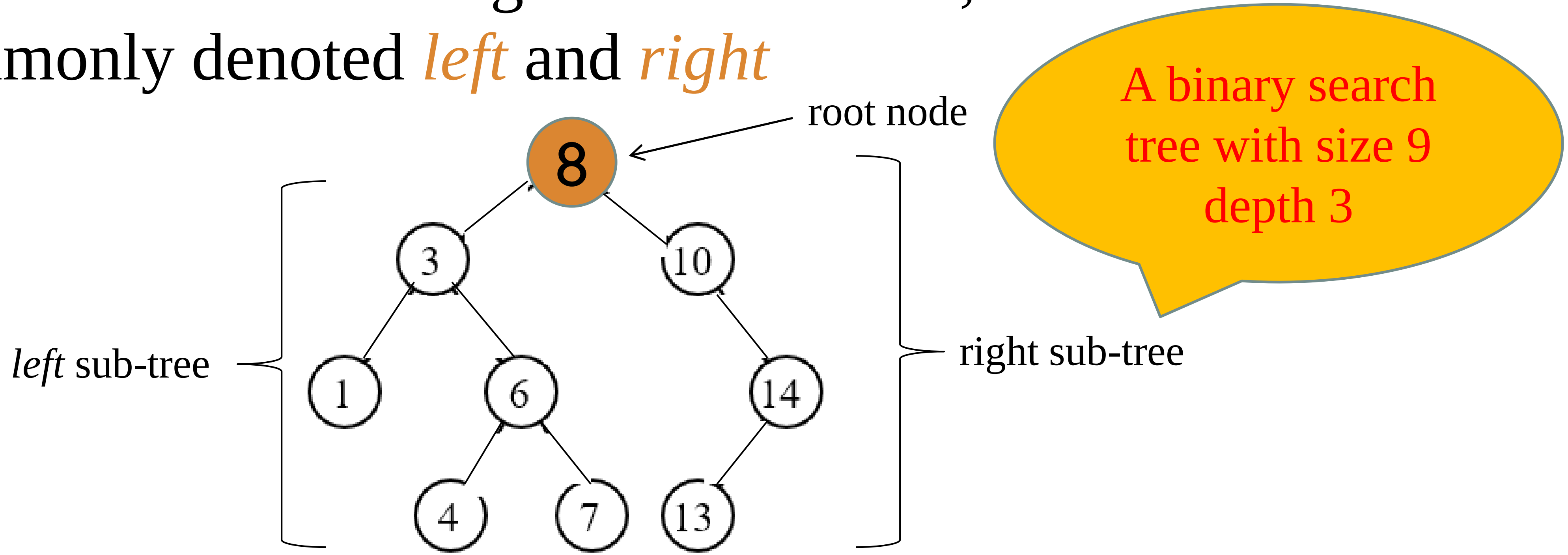
- A *rooted binary tree*.
- Internal nodes each store a *key*
- Each have two distinguished sub-trees,

commonly denoted *left* and *right*



WHAT IS BINARY SEARCH TREE ?

- A *rooted binary tree*.
- Internal nodes each store a *key*
- Each have two distinguished sub-trees, commonly denoted *left* and *right*



BINARY SEARCH PROPERTY

If x is *node* in *Binary search tree*

- If y is in *left sub-tree* of x then $key[y] \leq key[x]$
- If y is in *right sub-tree* of x then $key[y] \geq key[x]$

BINARY SEARCH PROPERTY

If x is *node* in *Binary search tree*

- If y is in *left sub-tree* of x then $key[y] \leq key[x]$
- If y is in *right sub-tree* of x then $key[y] \geq key[x]$

Operation of BST:

- Insertion of elements
- Deletion of elements
- Lookup (checking whether a key is present).

BINARY-SEARCH-TREE SORT

$T \leftarrow \varnothing$ ⊳ Create an empty BST

for $i = 1$ to n

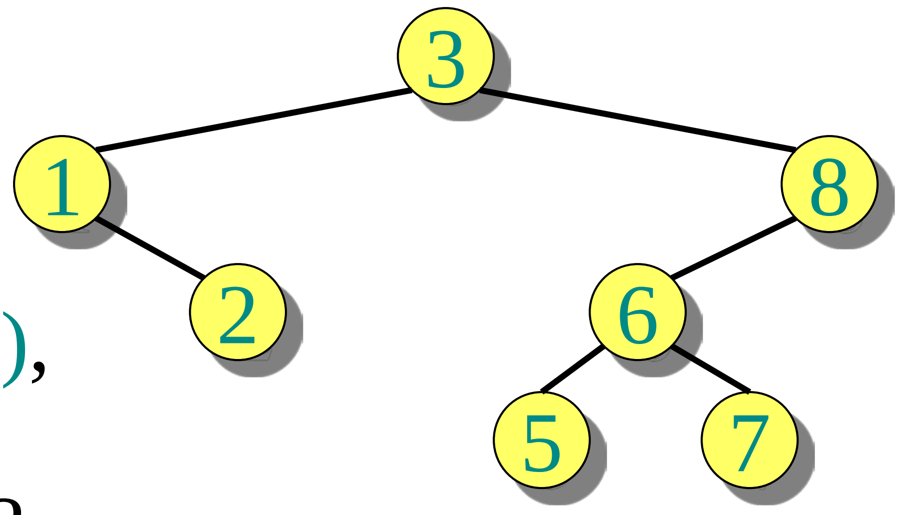
do TREE-INSERT$(T, A[i])$

Perform an inorder tree walk of T.

Example:

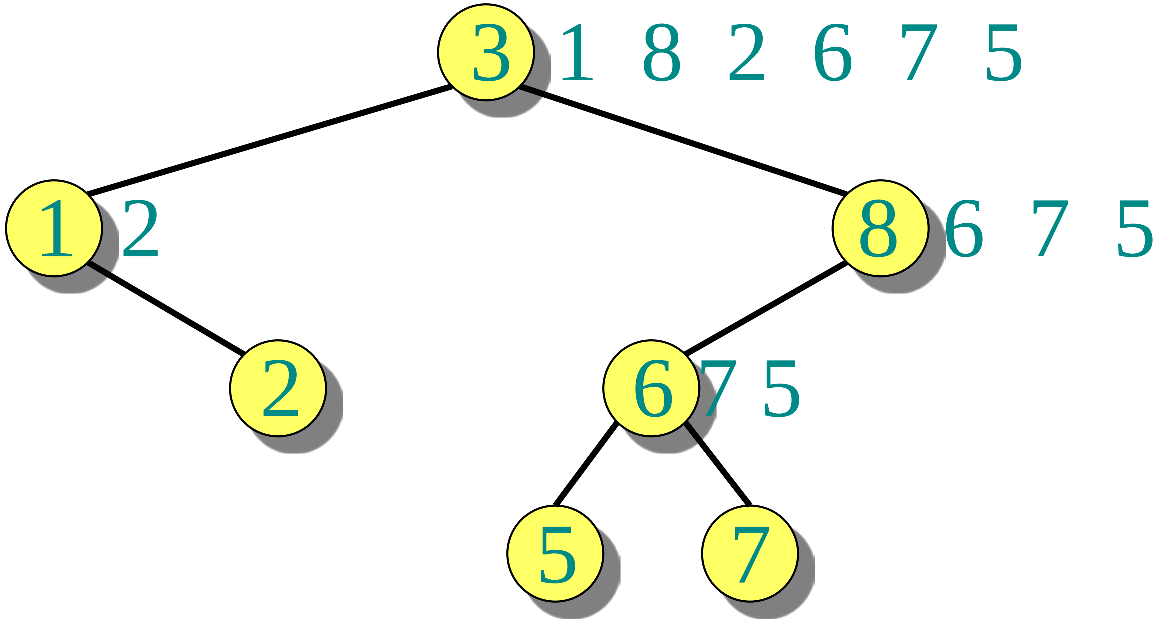
$A = [3\ 1\ 8\ 2\ 6\ 7\ 5]$

Tree-walk time $= O(n)$, but how long does it take to build the BST?



ANALYSIS OF BST SORT

BST sort performs the same comparisons as quicksort, but in a different order!



The expected time to build the tree is asymptotically the same as the running time of quicksort.

NODE DEPTH

The depth of a node = the number of comparisons made during TREE-INSERT. Assuming all input permutations are equally likely, we have

Average node depth

$$= \frac{1}{n} E\left[\sum_{i=1}^{n} (\#\text{comparisons to insert node } i)\right]$$

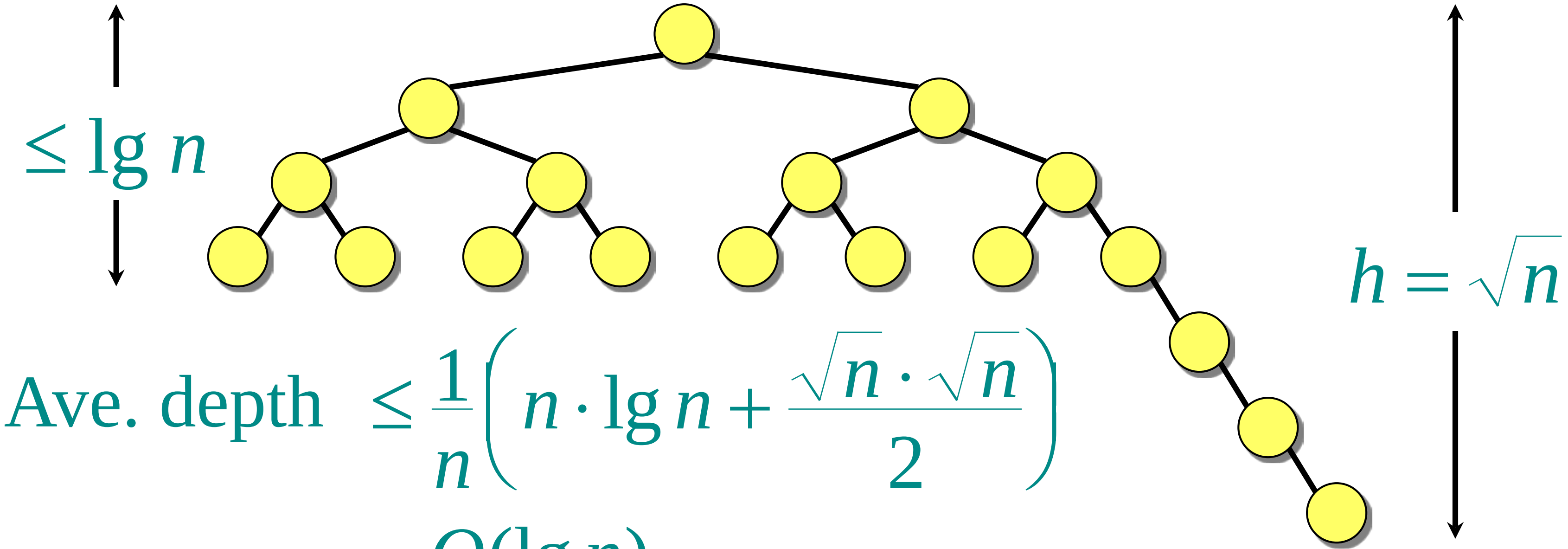
$$= \frac{1}{n} O(n \lg n) \qquad \text{(quicksort analysis)}$$

$$= O(\lg n) \ .$$

EXPECTED TREE HEIGHT

But, average node depth of a randomly built BST = $O(\lg n)$ does not necessarily mean that its expected height is also $O(\lg n)$ (although it is).

Example.



$$\text{Ave. depth} \le \frac{1}{n}\left(n \cdot \lg n + \frac{\sqrt{n} \cdot \sqrt{n}}{2}\right)$$
$$= O(\lg n)$$

HEIGHT OF A RANDOMLY BUILT BINARY SEARCH TREE

Outline of the analysis:

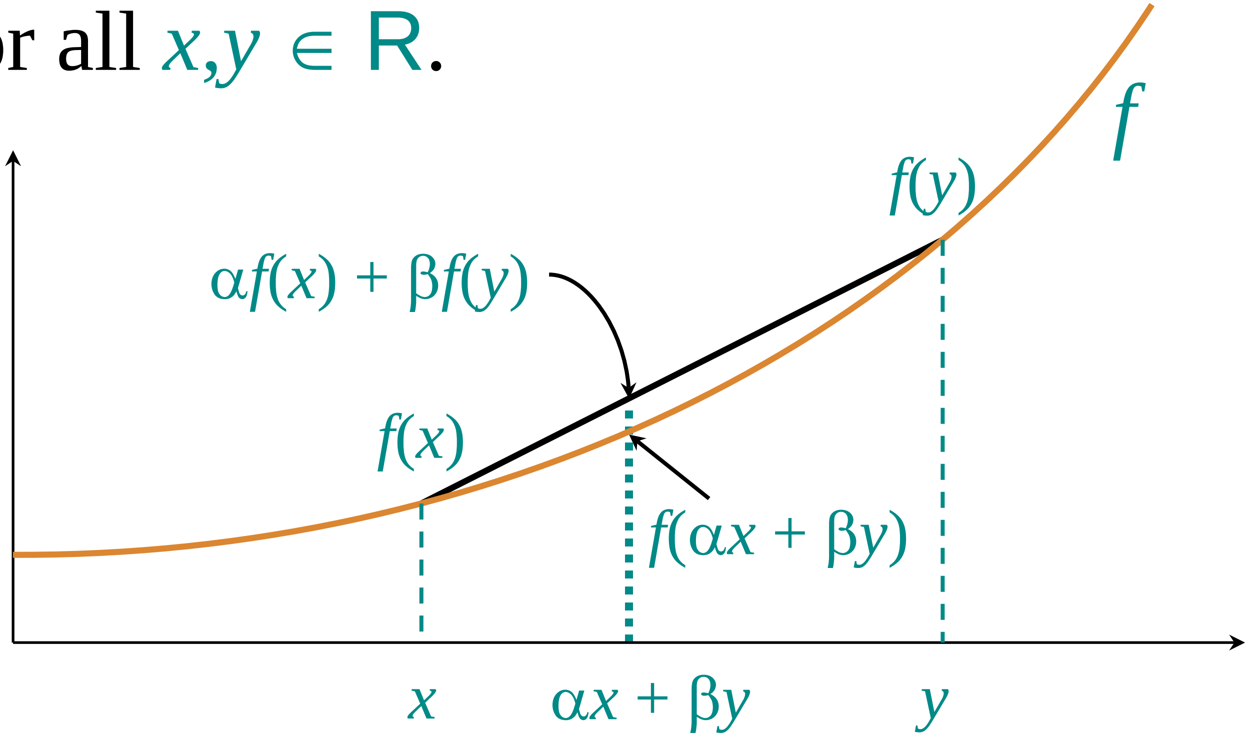
- Prove ***Jensen's inequality***, which says that $f(E[X]) \le E[f(X)]$ for any convex function f and random variable X.
- Analyze the ***exponential height*** of a randomly built BST on n nodes, which is the random variable $Y_n = 2^{X_n}$, where X_n is the random variable denoting the height of the BST.
- Prove that $2^{E[X_n]} \le E[2^{X_n}] = E[Y_n] = O(n^3)$, and hence that $E[X_n] = O(\lg n)$.

CONVEX FUNCTIONS

A function $f : \mathsf{R} \rightarrow \mathsf{R}$ is ***convex*** if for all $\alpha, \beta \geq 0$ such that $\alpha + \beta = 1$, we have

$$f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y)$$

for all $x, y \in \mathsf{R}$.



CONVEXITY LEMMA

Lemma. Let $f : \mathrm{R} \to \mathrm{R}$ be a convex function, and let $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ be a set of nonnegative constants such that $\sum_k \alpha_k = 1$. Then, for any set $\{x_1, x_2, \ldots, x_n\}$ of real numbers, we have

$$f\left(\sum_{k=1}^{n} \alpha_k x_k\right) \le \sum_{k=1}^{n} \alpha_k f(x_k) .$$

Proof. By induction on n. For $n = 1$, we have $\alpha_1 = 1$, and hence $f(\alpha_1 x_1) \le \alpha_1 f(x_1)$ trivially.

PROOF (CONTINUED)

Inductive step:

$$f\left(\sum_{k=1}^{n}\alpha_k x_k\right) = f\left(\alpha_n x_n + (1-\alpha_n)\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n}x_k\right)$$

Algebra.

PROOF (CONTINUED)

Inductive step:

$$f\left(\sum_{k=1}^{n}\alpha_k x_k\right) = f\left(\alpha_n x_n + (1-\alpha_n)\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n}x_k\right)$$

$$\leq \alpha_n f(x_n) + (1-\alpha_n) f\left(\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n}x_k\right)$$

Convexity.

PROOF (CONTINUED)

Inductive step:

$$f\left(\sum_{k=1}^{n}\alpha_k x_k\right)=f\left(\alpha_n x_n+(1-\alpha_n)\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n}x_k\right)$$

$$\leq \alpha_n f(x_n)+(1-\alpha_n)f\left(\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n}x_k\right)$$

$$\leq \alpha_n f(x_n)+(1-\alpha_n)\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n}f(x_k)$$

Induction.

PROOF (CONTINUED)

Inductive step:

$$
\begin{aligned}
f\left(\sum_{k=1}^{n}\alpha_k x_k\right) &= f\left(\alpha_n x_n + (1-\alpha_n)\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n}x_k\right) \\
&\le \alpha_n f(x_n) + (1-\alpha_n) f\left(\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n}x_k\right) \\
&\le \alpha_n f(x_n) + (1-\alpha_n)\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n} f(x_k) \\
&= \sum_{k=1}^{n}\alpha_k f(x_k). \quad \square
\end{aligned}
$$

Algebra.

JENSEN'S INEQUALITY

Lemma. Let f be a convex function, and let X be a random variable. Then, $f(E[X]) \leq E[f(X)]$.

Proof.

$$f(E[X]) = f\left(\sum_{k=-\infty}^{\infty} k \cdot \Pr\{X = k\}\right)$$

Definition of expectation.

JENSEN'S INEQUALITY

Lemma. Let f be a convex function, and let X be a random variable. Then, $f(E[X]) \le E[f(X)]$.

Proof.

$$f(E[X]) = f\left(\sum_{k=-\infty}^{\infty} k \cdot \Pr\{X = k\}\right)$$

$$\le \sum_{k=-\infty}^{\infty} f(k) \cdot \Pr\{X = k\}$$

Convexity lemma (generalized).

JENSEN'S INEQUALITY

Lemma. Let f be a convex function, and let X be a random variable. Then, $f(E[X]) \leq E[f(X)]$.

Proof.

$$f(E[X]) = f\left(\sum_{k=-\infty}^{\infty} k \cdot \Pr\{X = k\}\right)$$

$$\leq \sum_{k=-\infty}^{\infty} f(k) \cdot \Pr\{X = k\}$$

$$= E[f(X)]. \quad \square$$

Tricky step, but true—think about it.

ANALYSIS OF BST HEIGHT

Let X_n be the random variable denoting the height of a randomly built binary search tree on n nodes, and let $Y_n = 2^{X_n}$ be its exponential height.

If the root of the tree has rank k, then

$$X_n = 1 + \max\{X_{k-1}, X_{n-k}\} ,$$

since each of the left and right subtrees of the root are randomly built. Hence, we have

$$Y_n = 2 \cdot \max\{Y_{k-1}, Y_{n-k}\} .$$

ANALYSIS (CONTINUED)

Define the indicator random variable Z_{nk} as

$$Z_{nk} = \begin{cases} 1 & \text{if the root has rank } k, \\ 0 & \text{otherwise.} \end{cases}$$

Thus, $\Pr\{Z_{nk} = 1\} = \mathrm{E}[Z_{nk}] = 1/n$, and

$$Y_n = \sum_{k=1}^{n} Z_{nk} \left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right) .$$

EXPONENTIAL HEIGHT RECURRENCE

$$E[Y_n] = E\left[\sum_{k=1}^{n} Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

Take expectation of both sides.

EXPONENTIAL HEIGHT RECURRENCE

$$E[Y_n] = E\left[\sum_{k=1}^{n} Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

$$= \sum_{k=1}^{n} E[Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)]$$

Linearity of expectation.

EXPONENTIAL HEIGHT RECURRENCE

$$E[Y_n] = E\left[\sum_{k=1}^{n} Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

$$= \sum_{k=1}^{n} E[Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)]$$

$$= 2\sum_{k=1}^{n} E[Z_{nk}] \cdot E[\max\{Y_{k-1}, Y_{n-k}\}]$$

Independence of the rank of the root from the ranks of subtree roots.

EXPONENTIAL HEIGHT RECURRENCE

$$E[Y_n] = E\left[\sum_{k=1}^{n} Z_{nk}\left(2\cdot\max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

$$= \sum_{k=1}^{n} E[Z_{nk}\left(2\cdot\max\{Y_{k-1}, Y_{n-k}\}\right)]$$

$$= 2\sum_{k=1}^{n} E[Z_{nk}]\cdot E[\max\{Y_{k-1}, Y_{n-k}\}]$$

$$\le \frac{2}{n}\sum_{k=1}^{n} E[Y_{k-1} + Y_{n-k}]$$

The max of two nonnegative numbers is at most their sum, and $\mathrm{E}[Z_{nk}] = 1/n$.

EXPONENTIAL HEIGHT RECURRENCE

$$E[Y_n] = E\left[\sum_{k=1}^{n} Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

$$= \sum_{k=1}^{n} E[Z_{nk}\left(2 \cdot \max\{Y_{k-1}, Y_{n-k}\}\right)]$$

$$= 2\sum_{k=1}^{n} E[Z_{nk}] \cdot E[\max\{Y_{k-1}, Y_{n-k}\}]$$

$$\leq \frac{2}{n}\sum_{k=1}^{n} E[Y_{k-1} + Y_{n-k}]$$

$$= \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k]$$

Each term appears twice, and reindex.

SOLVING THE RECURRENCE

Use substitution to show that $E[Y_n] \le cn^3$ for some positive constant c, which we can pick sufficiently large to handle the initial conditions.

$$E[Y_n] = \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k]$$

SOLVING THE RECURRENCE

Use substitution to show that $E[Y_n] \le cn^3$ for some positive constant c, which we can pick sufficiently large to handle the initial conditions.

$$E[Y_n] = \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k]$$

$$\le \frac{4}{n}\sum_{k=0}^{n-1} ck^3$$

Substitution.

SOLVING THE RECURRENCE

Use substitution to show that $E[Y_n] \le cn^3$ for some positive constant c, which we can pick sufficiently large to handle the initial conditions.

$$E[Y_n] = \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k]$$

$$\le \frac{4}{n}\sum_{k=0}^{n-1} ck^3$$

$$\le \frac{4c}{n}\int_0^n x^3 dx$$

Integral method.

SOLVING THE RECURRENCE

Use substitution to show that $E[Y_n] \leq cn^3$ for some positive constant c, which we can pick sufficiently large to handle the initial conditions.

$$
\begin{aligned}
E[Y_n] &= \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k] \\
&\leq \frac{4}{n}\sum_{k=0}^{n-1} ck^3 \\
&\leq \frac{4c}{n}\int_0^n x^3 dx \\
&= \frac{4c}{n}\left(\frac{n^4}{4}\right)
\end{aligned}
$$

Solve the integral.

SOLVING THE RECURRENCE

Use substitution to show that $E[Y_n] \le cn^3$ for some positive constant c, which we can pick sufficiently large to handle the initial conditions.

$$
\begin{aligned}
E[Y_n] &= \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k] \\
&\le \frac{4}{n} \sum_{k=0}^{n-1} ck^3 \\
&\le \frac{4c}{n} \int_0^n x^3 dx \\
&= \frac{4c}{n} \left(\frac{n^4}{4} \right) \\
&= cn^3. \quad \text{Algebra.}
\end{aligned}
$$

THE GRAND FINALE

Putting it all together, we have

$$2^{E[X_n]} \leq E[2^{X_n}]$$

Jensen's inequality, since $f(x) = 2^x$ is convex.

THE GRAND FINALE

Putting it all together, we have

$$2^{E[X_n]} \le E[2^{X_n}]$$
$$= E[Y_n]$$

Definition.

THE GRAND FINALE

Putting it all together, we have

$$\begin{aligned} 2^{E[X_n]} &\leq E[2^{X_n}] \\ &= E[Y_n] \\ &\leq cn^3 . \end{aligned}$$

What we just showed.

THE GRAND FINALE

Putting it all together, we have

$$2^{E[X_n]} \le E[2^{X_n}]$$
$$= E[Y_n]$$
$$\le cn^3 .$$

Taking the lg of both sides yields

$$E[X_n] \le 3 \lg n + O(1) .$$

POST MORTEM

Q. Does the analysis have to be this hard?

Q. Why bother with analyzing exponential height?

Q. Why not just develop the recurrence on

$$X_n = 1 + \max\{X_{k-1}, X_{n-k}\}$$

directly?

POST MORTEM (CONTINUED)

A. The inequality

$$\max\{a, b\} \le a + b\ .$$

provides a poor upper bound, since the RHS approaches the LHS slowly as $|a - b|$ increases. The bound

$$\max\{2^a, 2^b\} \le 2^a + 2^b$$

allows the RHS to approach the LHS far more quickly as $|a - b|$ increases. By using the convexity of $f(x) = 2^x$ via Jensen's inequality, we can manipulate the sum of exponentials, resulting in a tight analysis.

THOUGHT EXERCISES

- See what happens when you try to do the analysis on X_n directly.
- Try to understand better why the proof uses an exponential. Will a quadratic do?
- See if you can find a simpler argument. (This argument is a little simpler than the one in the book—I hope it's correct!)

BALANCED SEARCH TREES

Balanced search tree: A search-tree data structure for which a height of $O(\lg n)$ 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

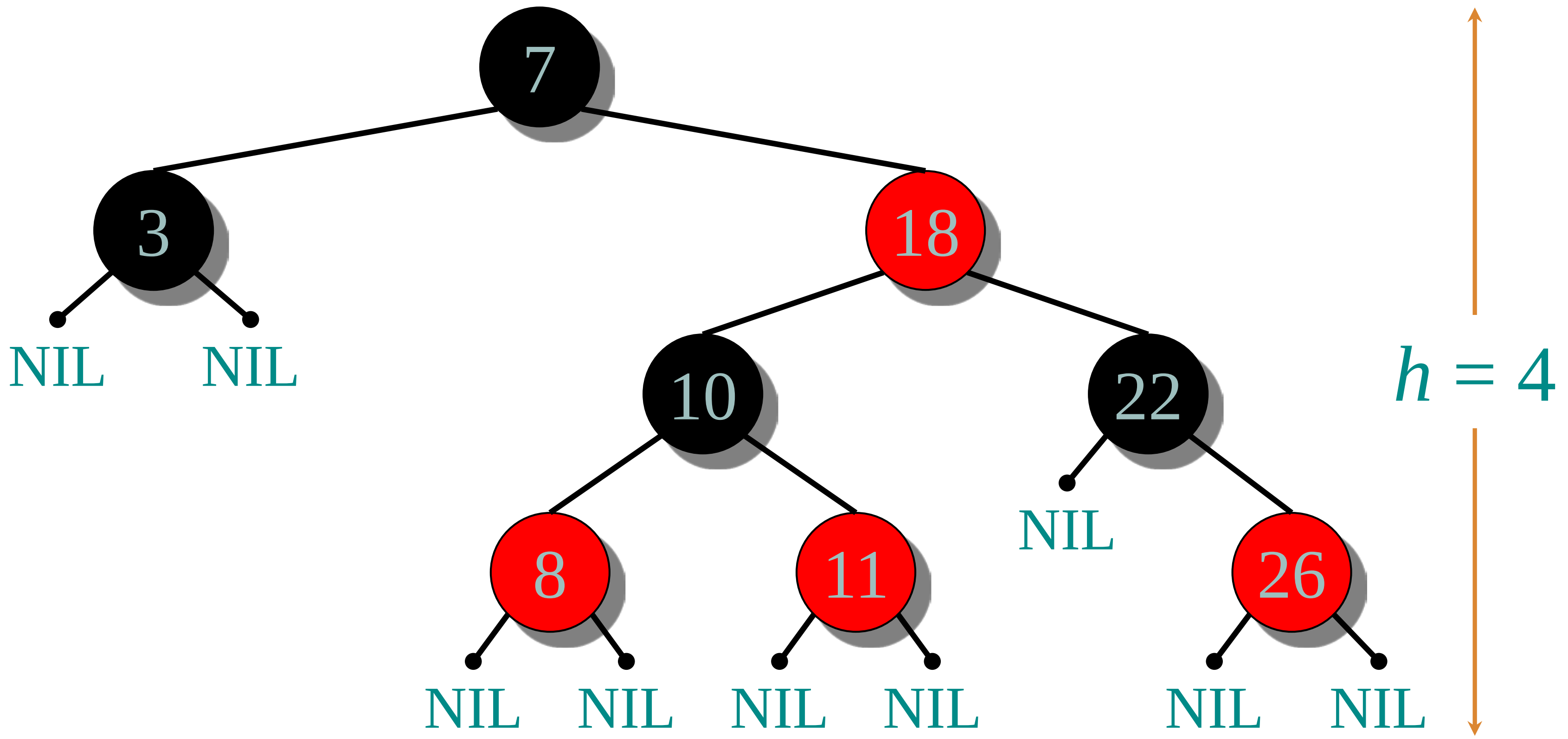
RED-BLACK TREES

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

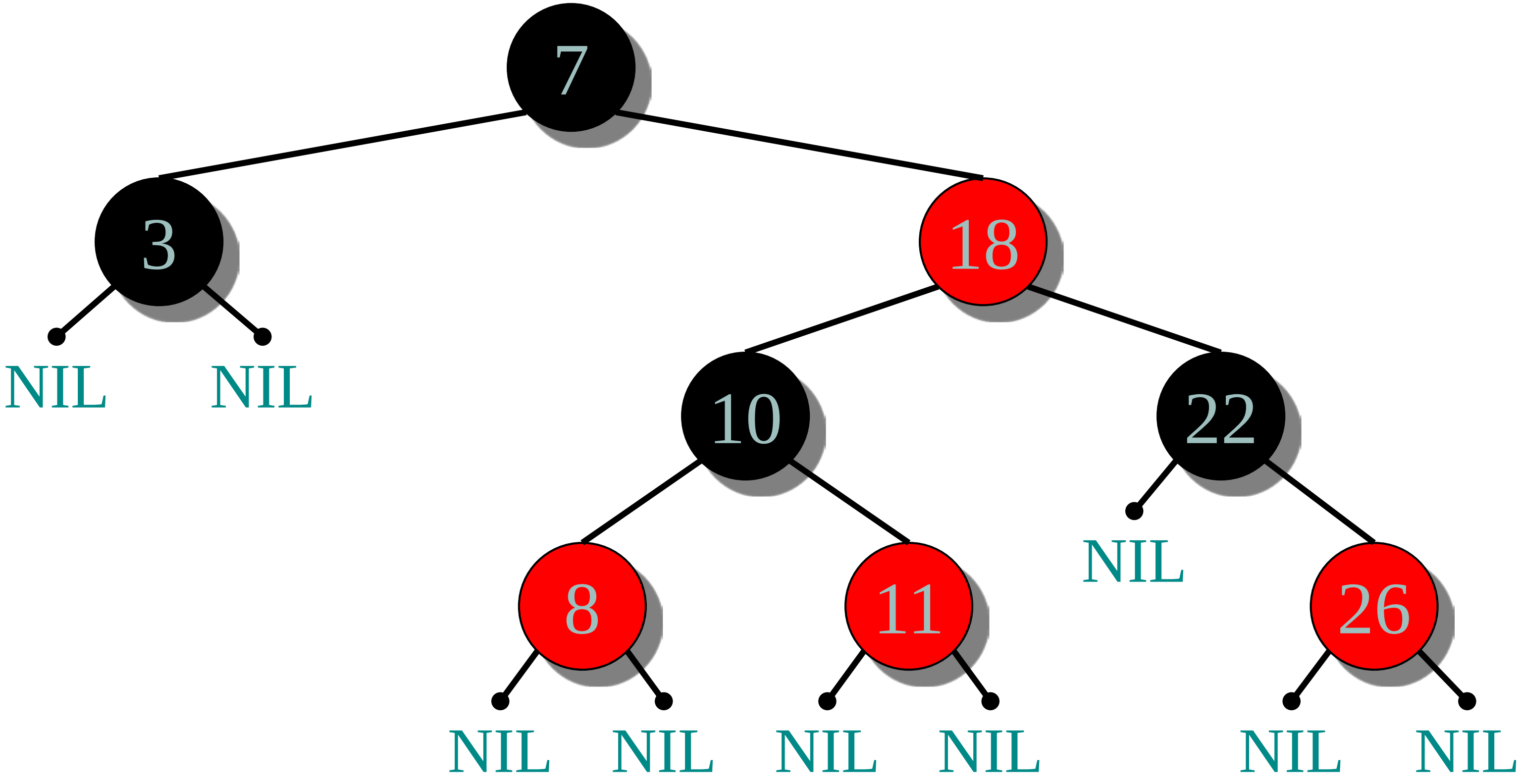
Red-black properties:

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

EXAMPLE OF A RED-BLACK TREE

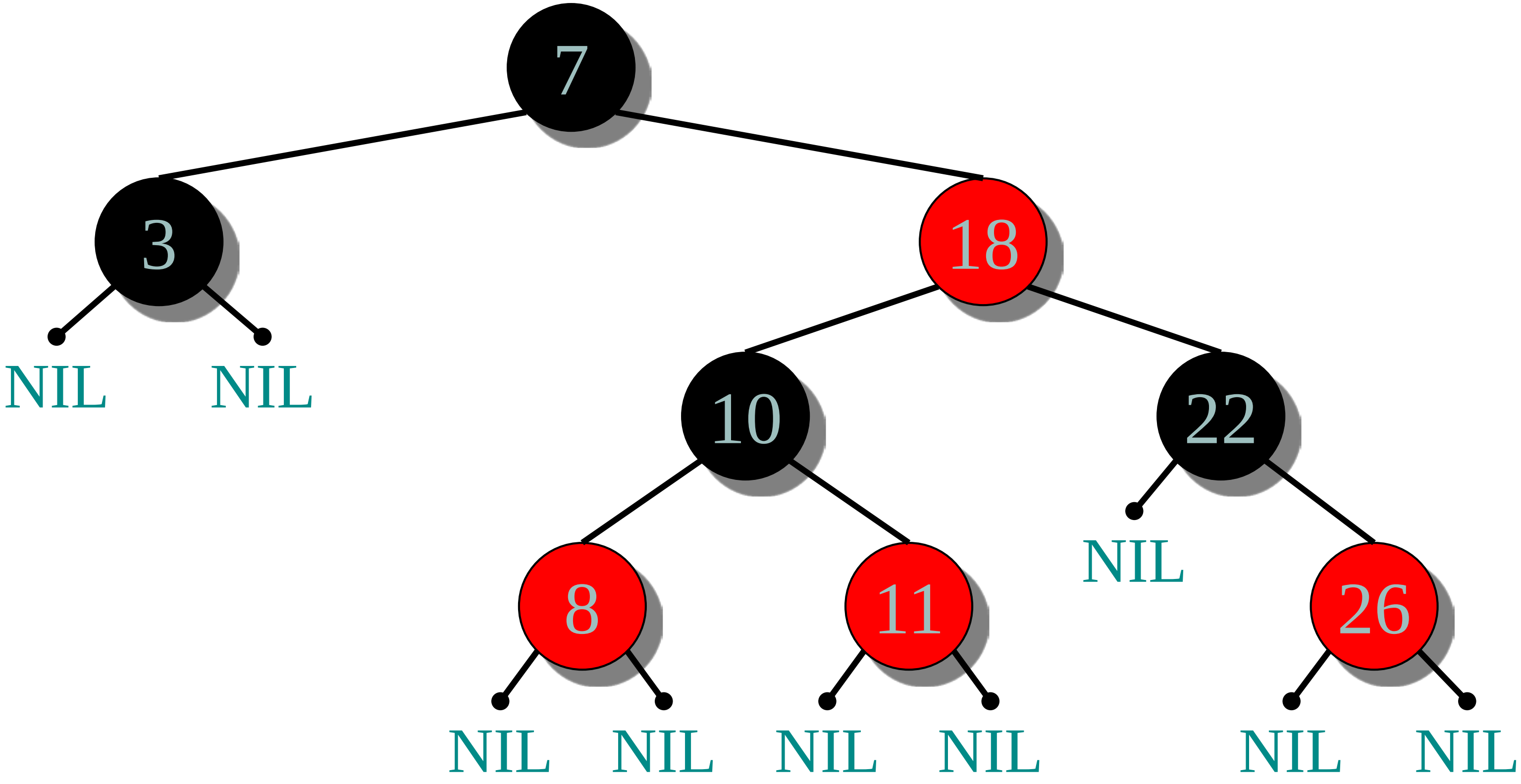


EXAMPLE OF A RED-BLACK TREE



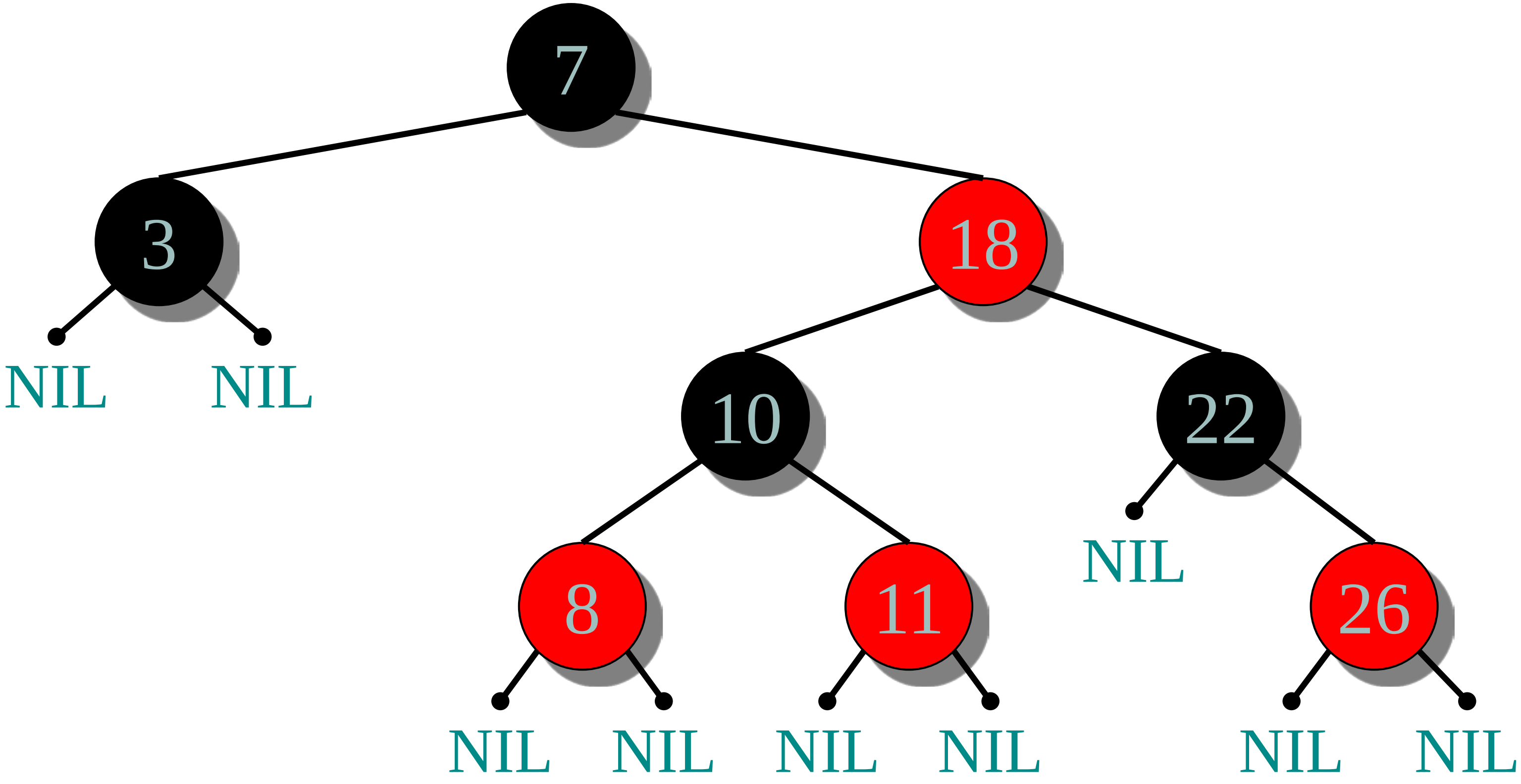
1. Every node is either red or black.

EXAMPLE OF A RED-BLACK TREE



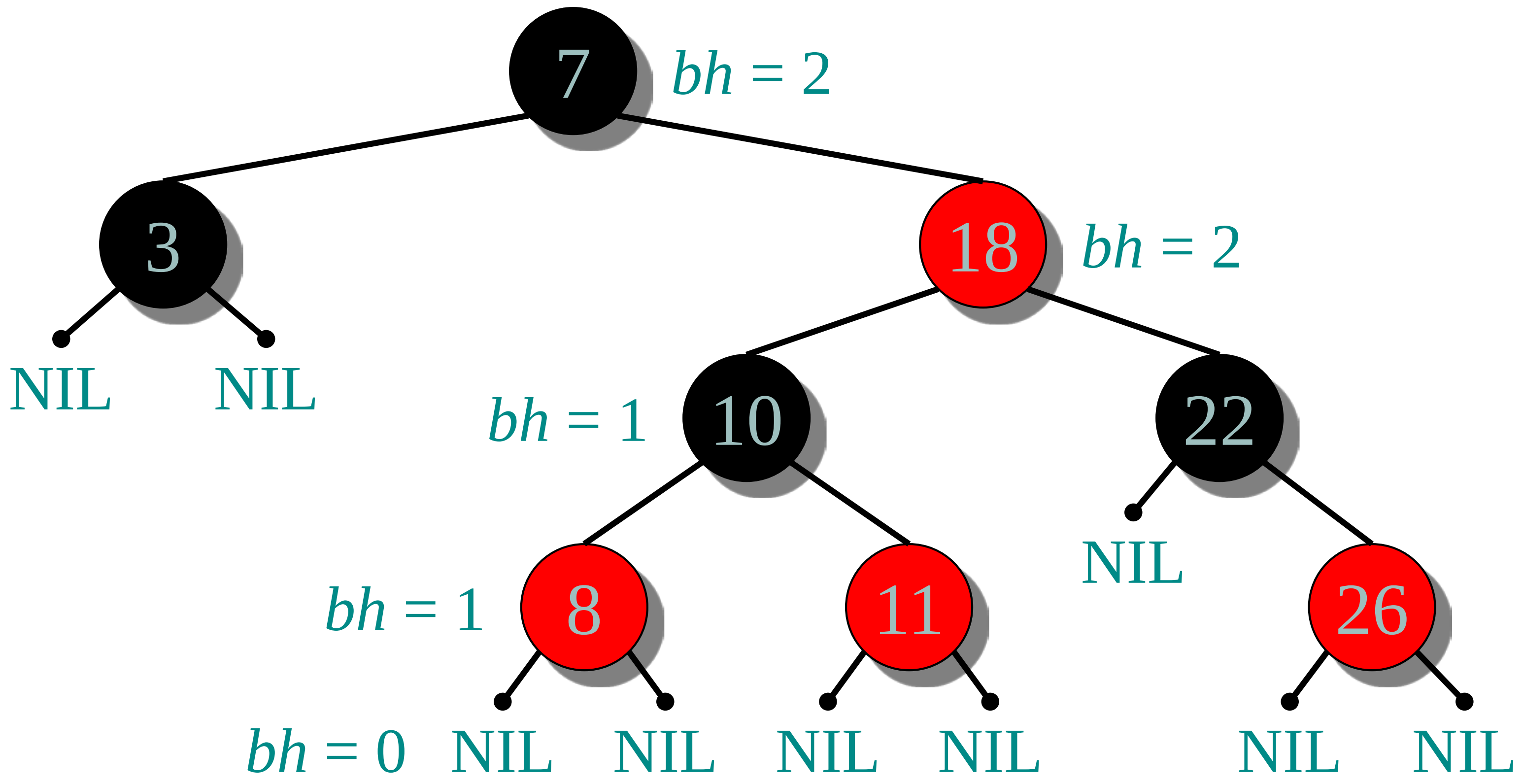
2. The root and leaves (NIL’s) are black.

EXAMPLE OF A RED-BLACK TREE



3. If a node is red, then its parent is black.

EXAMPLE OF A RED-BLACK TREE



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

HEIGHT OF A RED-BLACK TREE

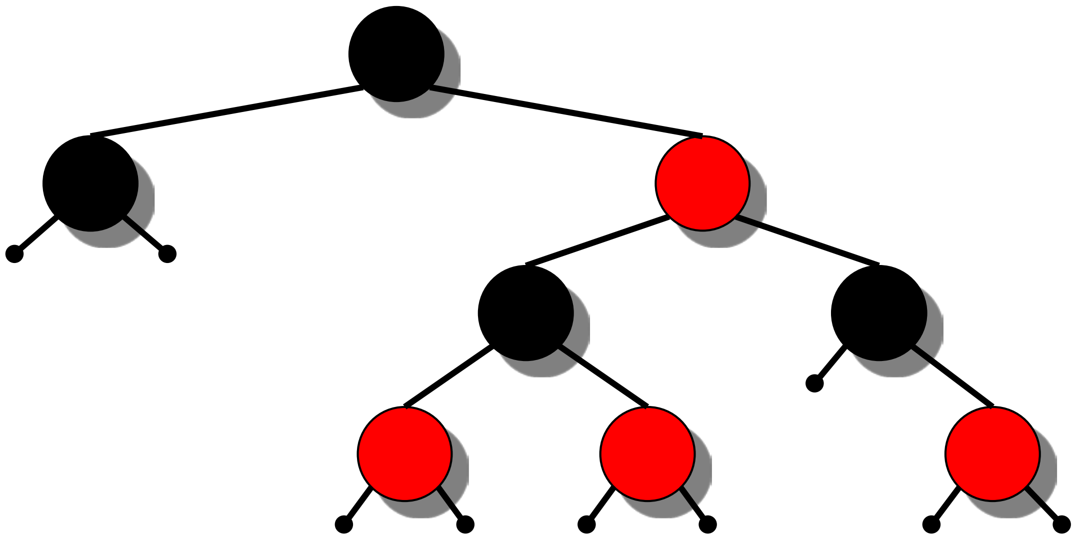
Theorem. A red-black tree with n keys has height

$$h \leq 2 \lg(n + 1).$$

Proof.

INTUITION:

- Merge red nodes into their black parents.



HEIGHT OF A RED-BLACK TREE

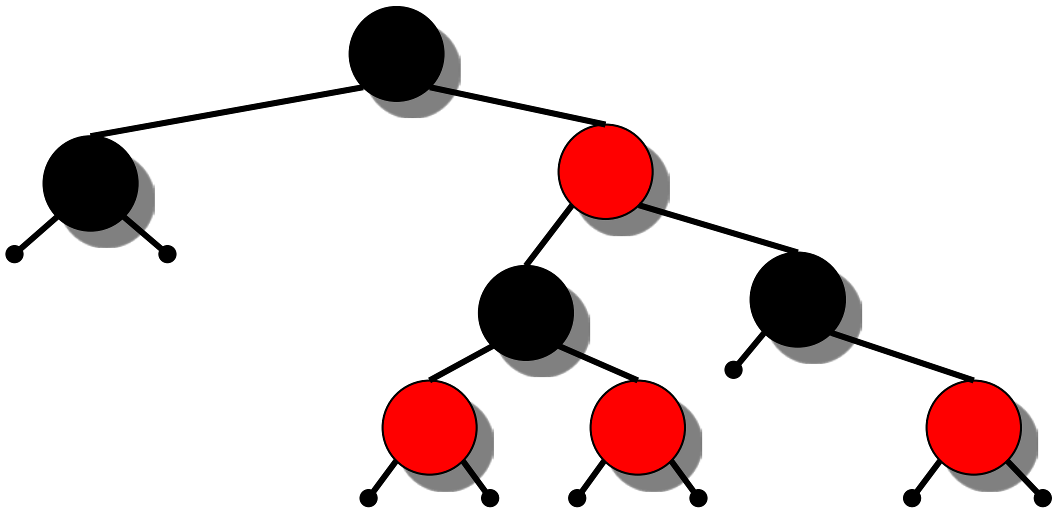
Theorem. A red-black tree with n keys has height

$$h \leq 2 \lg(n + 1).$$

Proof.

INTUITION:

- Merge red nodes into their black parents.



HEIGHT OF A RED-BLACK TREE

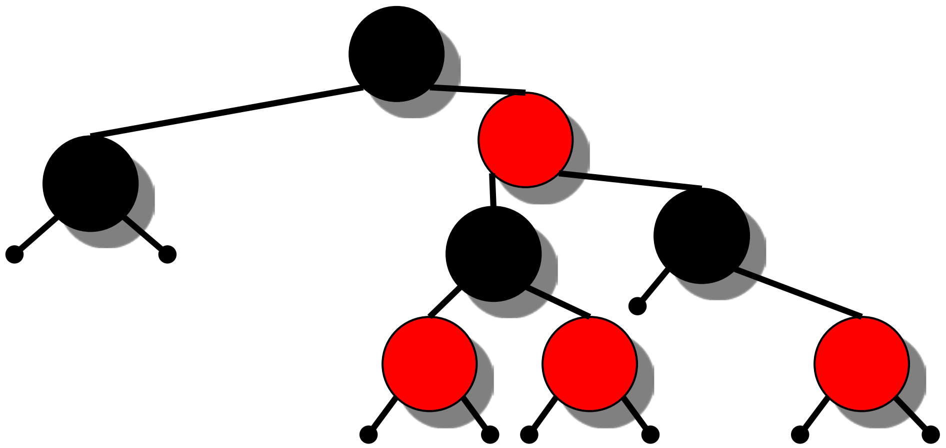
Theorem. A red-black tree with n keys has height

$$h \leq 2 \lg(n + 1).$$

Proof.

INTUITION:

- Merge red nodes into their black parents.



HEIGHT OF A RED-BLACK TREE

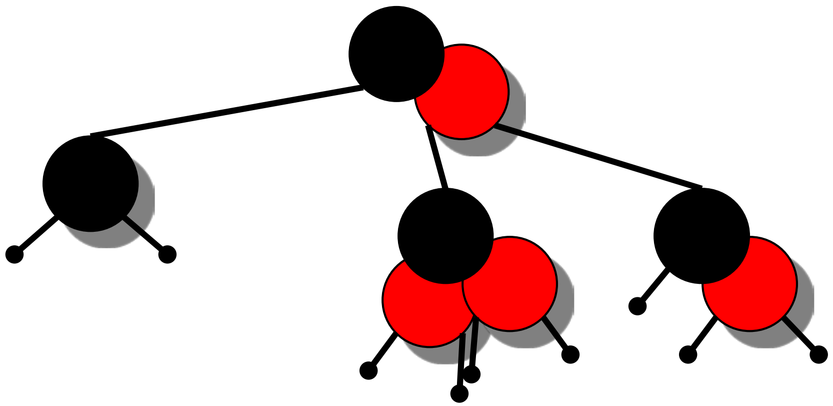
Theorem. A red-black tree with n keys has height

$$h \leq 2\lg(n+1).$$

Proof.

INTUITION:

- Merge red nodes into their black parents.



HEIGHT OF A RED-BLACK TREE

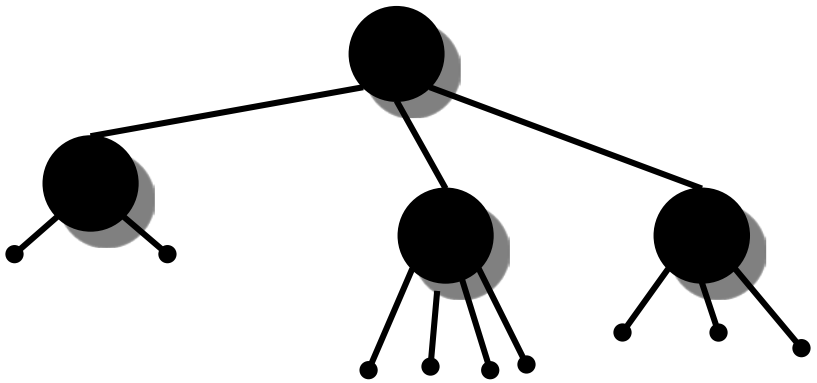
Theorem. A red-black tree with n keys has height

$$h \leq 2 \lg(n+1).$$

Proof.

INTUITION:

- Merge red nodes into their black parents.



HEIGHT OF A RED-BLACK TREE

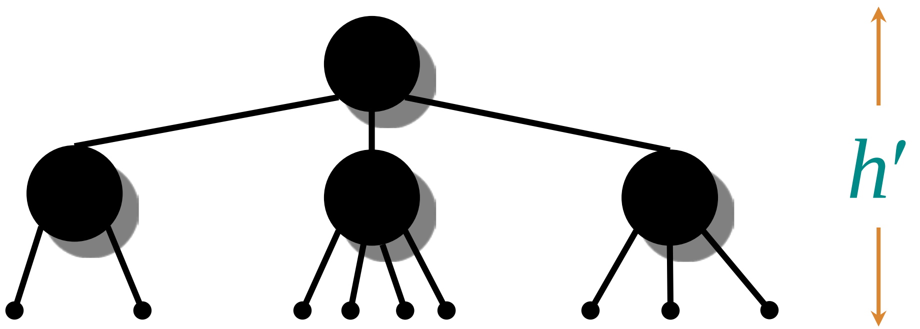
Theorem. A red-black tree with n keys has height

$$h \leq 2 \lg(n + 1).$$

Proof.

INTUITION:

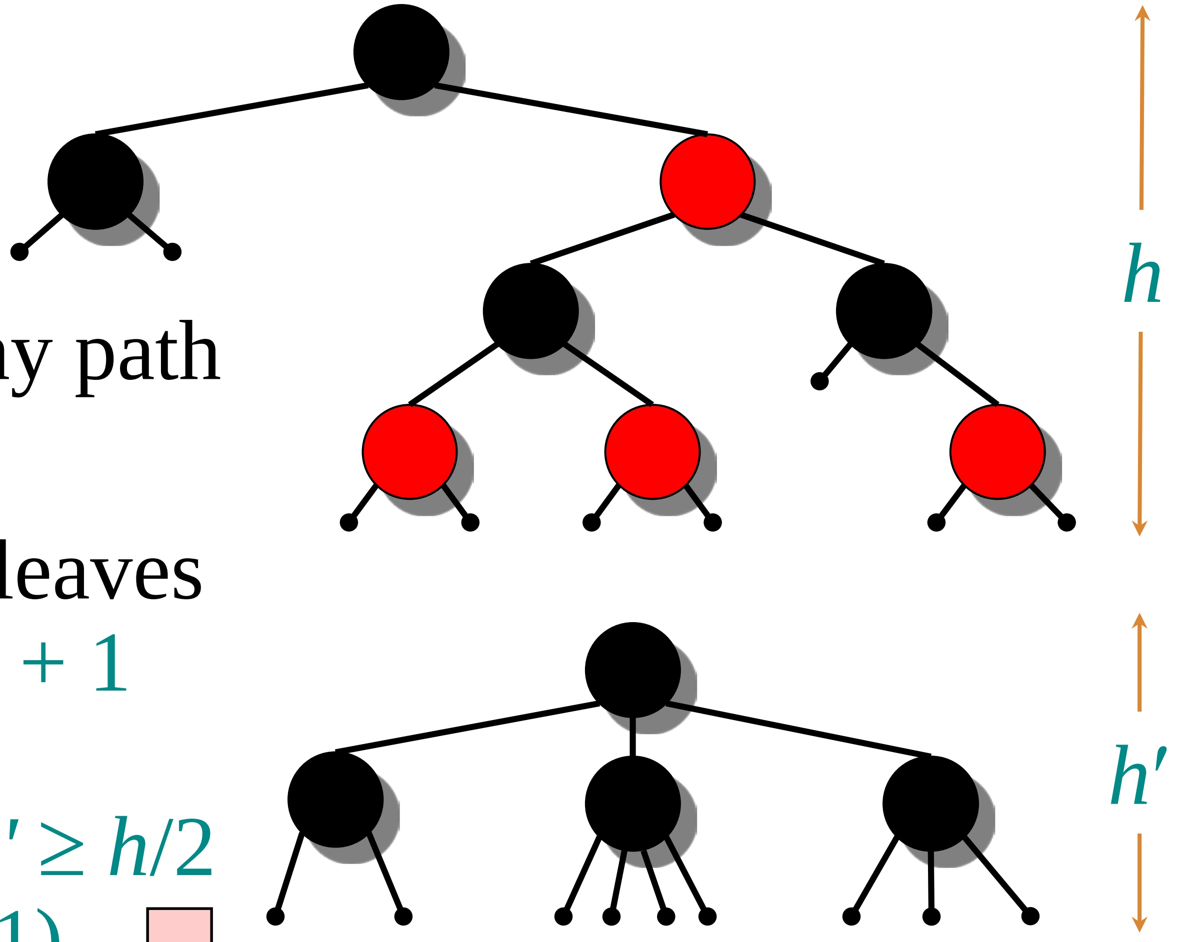
- Merge red nodes into their black parents.



- 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.

PROOF (CONTINUED)

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



QUERY OPERATIONS

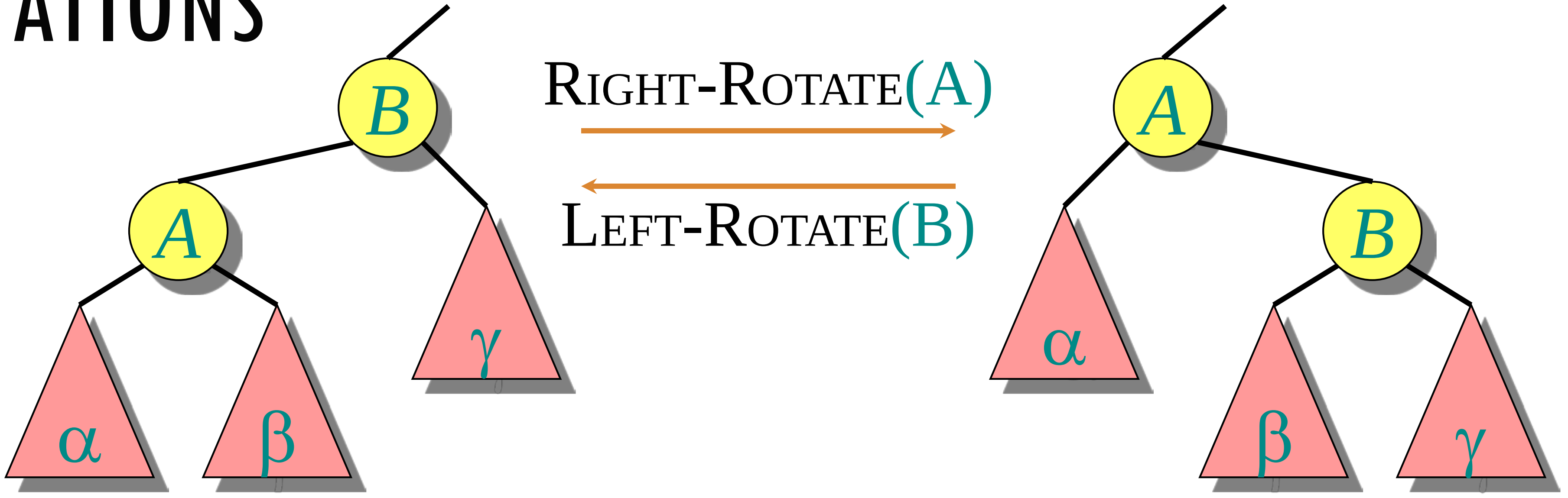
Corollary. The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR all run in $O(\lg n)$ time on a red-black tree with n nodes.

MODIFYING OPERATIONS

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

- the operation itself,
- color changes,
- restructuring the links of the tree: ***"rotations"***.

ROTATIONS



Rotations maintain the inorder ordering of keys:

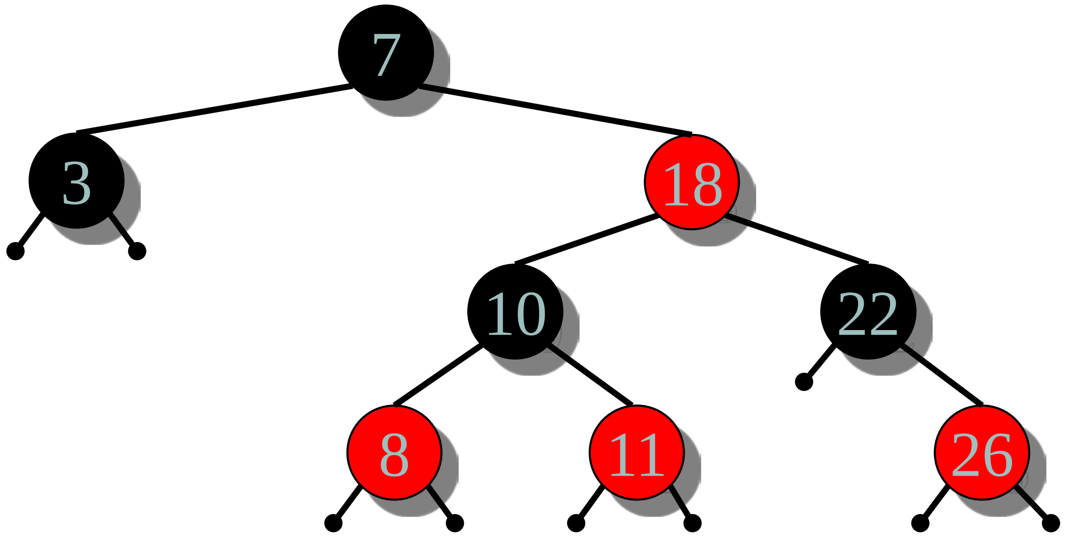
- $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c$.

A rotation can be performed in $O(1)$ time.

INSERTION INTO A RED-BLACK TREE

IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

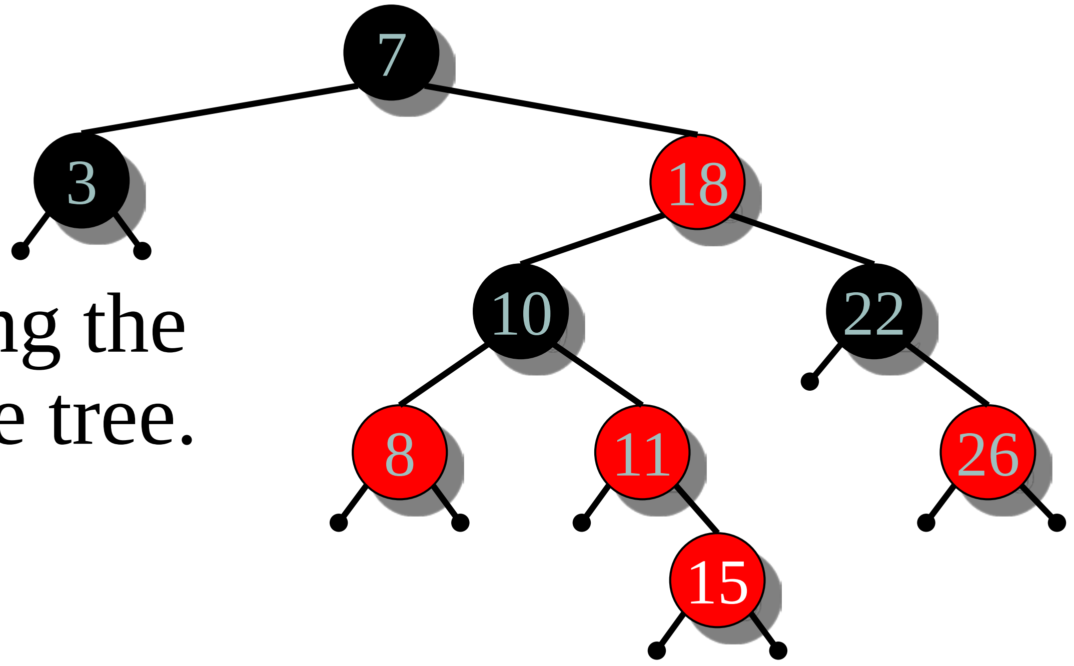


INSERTION INTO A RED-BLACK TREE

IDEA: Insert x in tree. Color x red. Only red-black property 3 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.

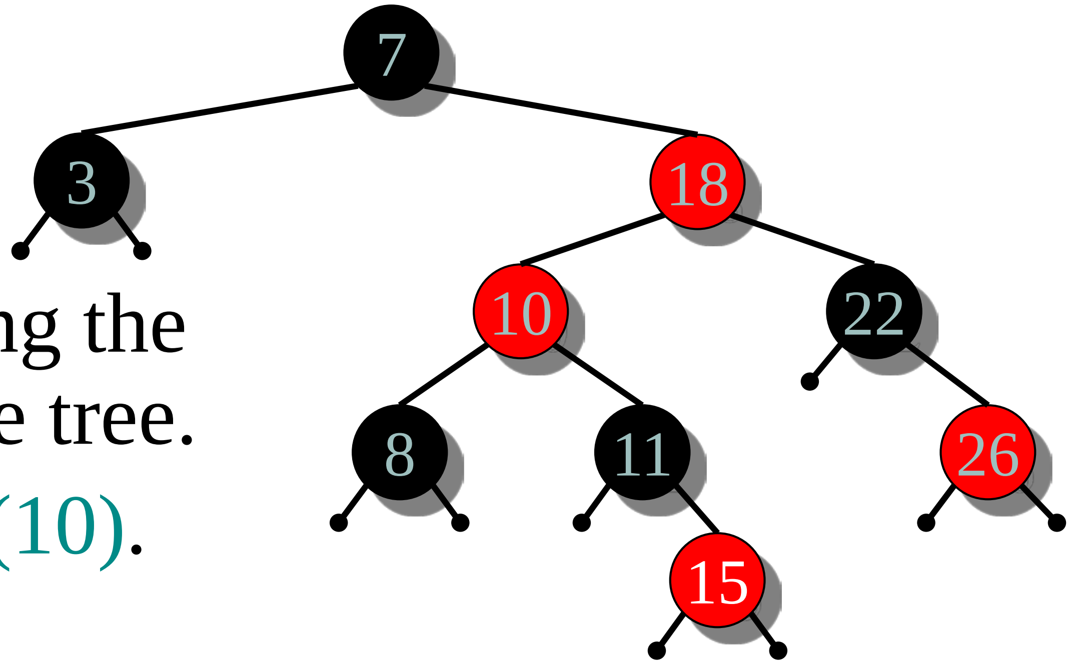


INSERTION INTO A RED-BLACK TREE

IDEA: Insert x in tree. Color x red. Only red-black property 3 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(10).

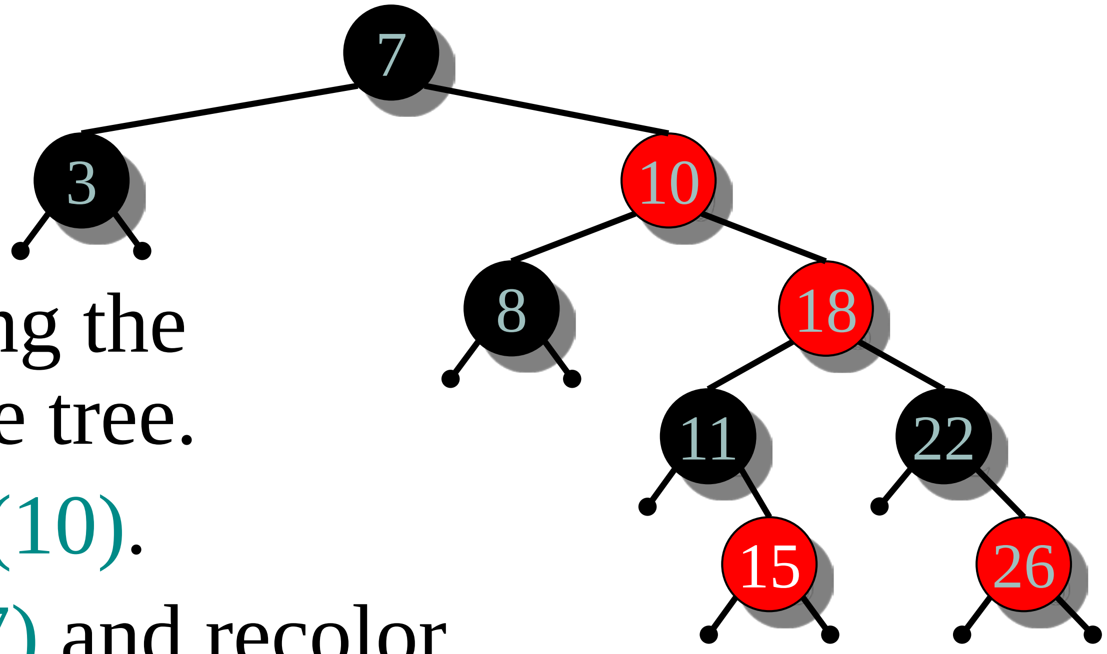


INSERTION INTO A RED-BLACK TREE

IDEA: Insert *x* in tree. Color *x* red. Only red-black property 3 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(10).
- LEFT-ROTATE(7) and recolor.

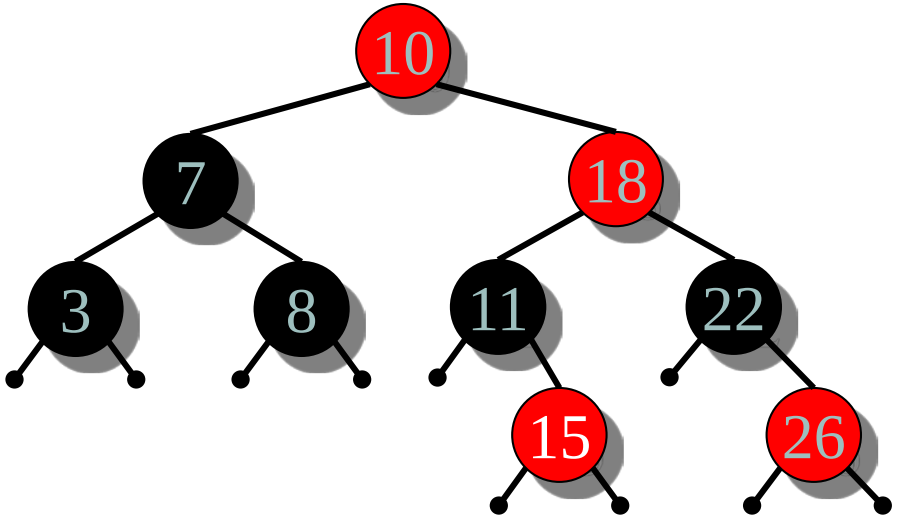


INSERTION INTO A RED-BLACK TREE

IDEA: Insert x in tree. Color x red. Only red-black property 3 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(10).
- LEFT-ROTATE(7) and recolor.

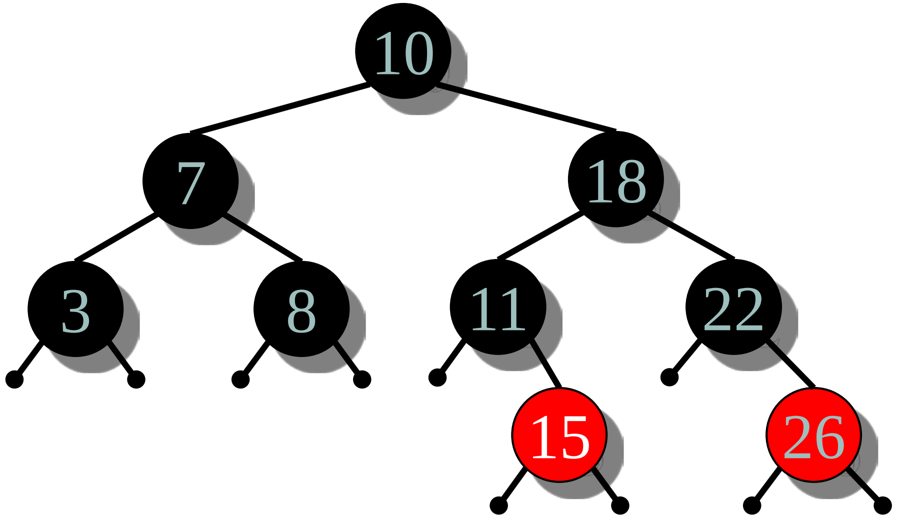


INSERTION INTO A RED-BLACK TREE

IDEA: Insert *x* in tree. Color *x* red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

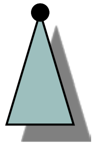
- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(10).
- LEFT-ROTATE(7) and recolor.

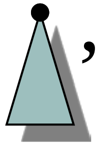


PSEUDOCODE

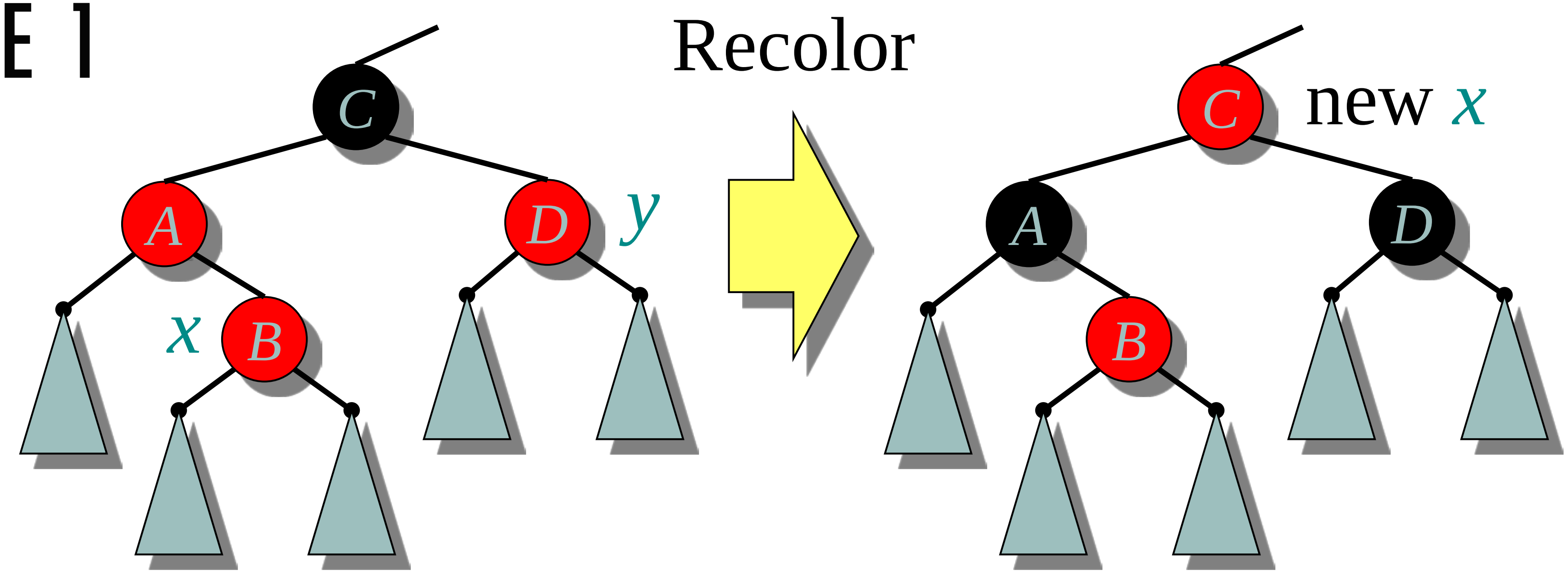
```
RB-INSERT(T, x)
    TREE-INSERT(T, x)
    color[x] ← RED        ▻ only RB property 3 can be violated
    while x ≠ root[T] and color[p[x]] = RED
        do if p[x] = left[p[p[x]]]
            then y ← right[p[p[x]]]        ▻ y = aunt/uncle of x
                if color[y] = RED
                 then ⟨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]] ← BLACK
```

GRAPHICAL NOTATION

Let  denote a subtree with a black root.

All ’s have the same black-height.

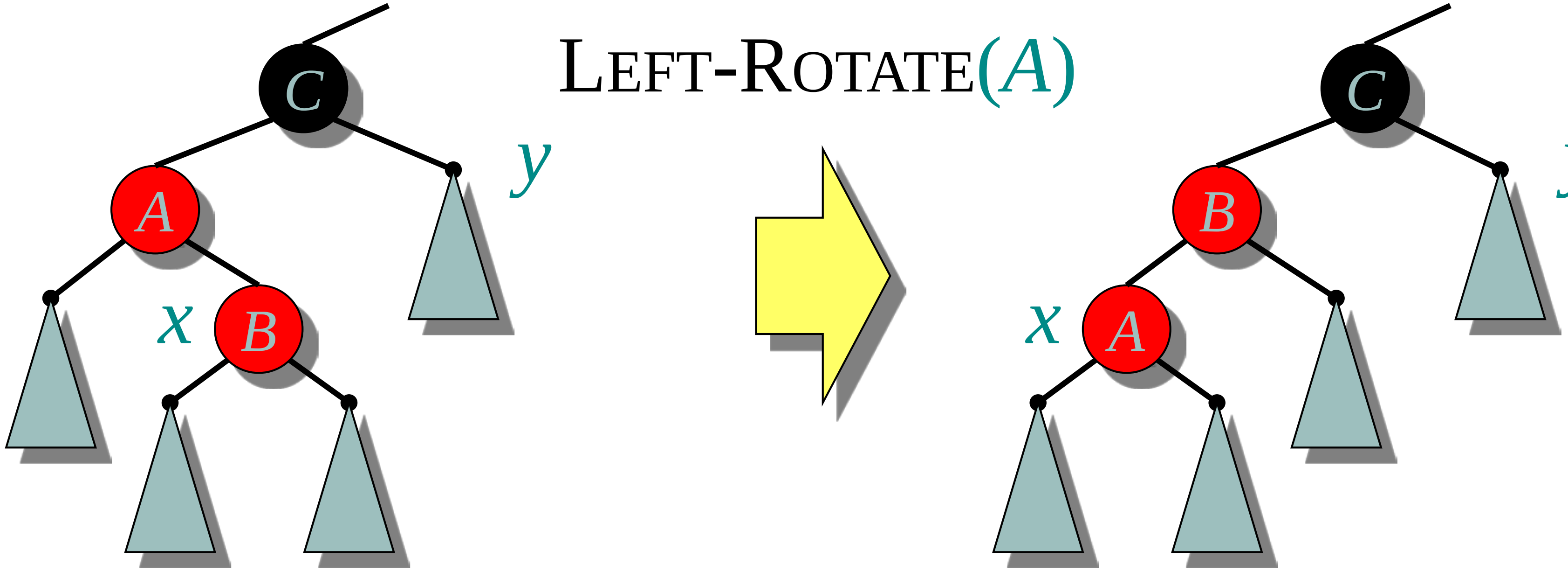
CASE 1



(Or, children of *A* are swapped.)

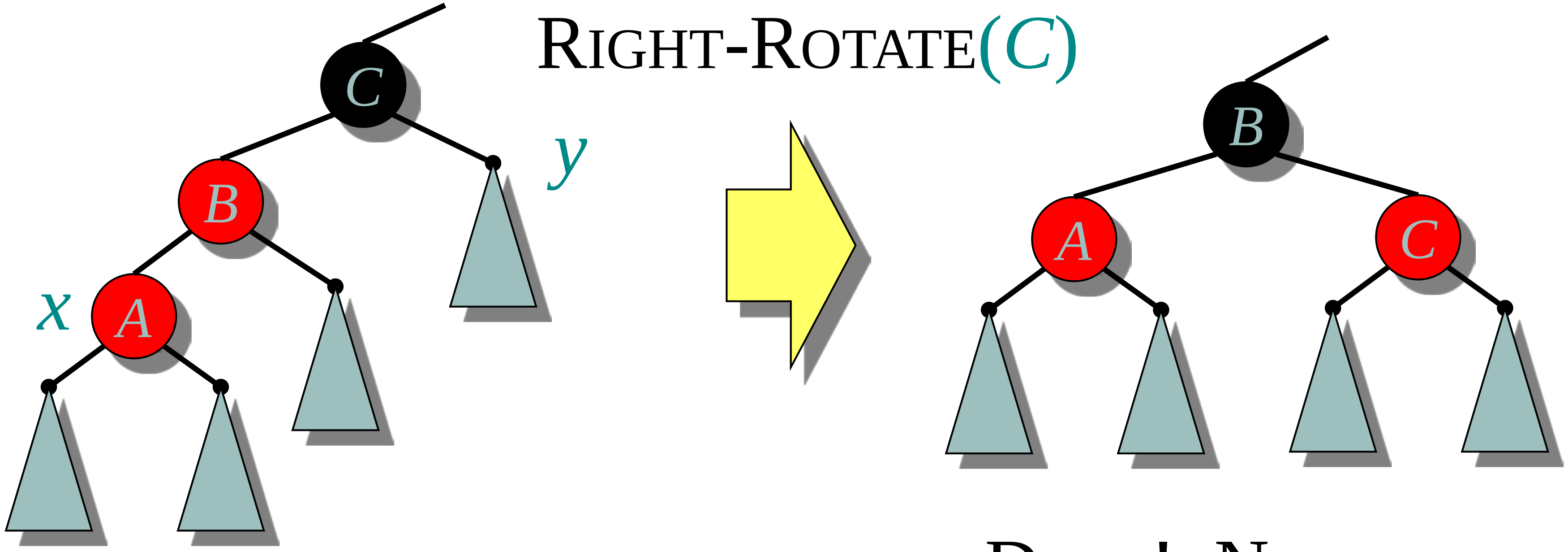
Push *C*'s black onto *A* and *D*, and recurse, since *C*'s parent may be red.

CASE 2



Transform to Case 3.

CASE 3



Done! No more violations of RB property 3 are possible.

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: $O(\lg n)$ with $O(1)$ rotations.

RB-DELETE — same asymptotic running time.