CMP448: Algorithms

Lecture 07: Binary Search Trees

Mohamed Alaa El-Dien Aly
Computer Engineering Department
Cairo University
Spring 2013

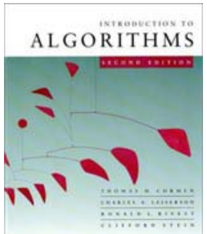
Agenda

- Binary Search Trees
  - Insert, Search, Delete
  - Expected Height
- Red Black Trees

Acknowledgment
A lot of slides adapted from the slides of Erik Demaine, Piotr Indyk, and Charles Leiserson

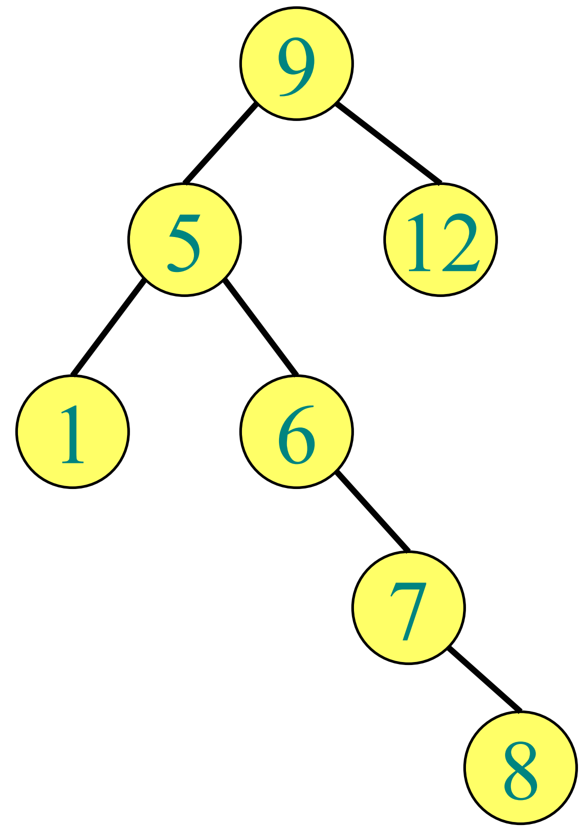
Dictionary Data Structure

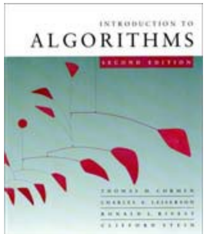
- Data structures that support
  - Insert(data, key)
  - Delete(data, key)
  - Search(data, key)
- Last lecture
  - Hash tables
  - Operations in expected constant time
- This lecture
  - Binary Trees
  - Operations in expected log time



Binary Search Tree

- Each node x has:
  - key[x]
  - Pointers:
    - left[x]
    - right[x]
    - parent[x]





Binary Search Tree (BST)

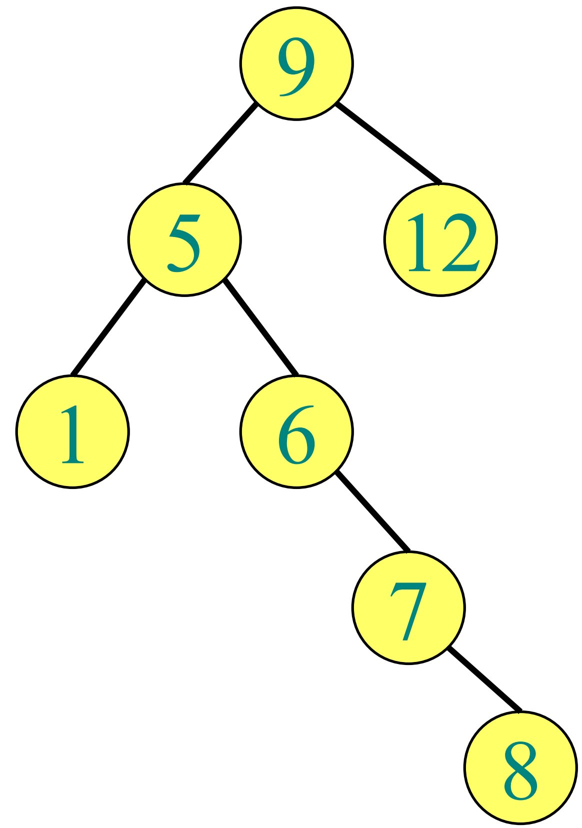
BST Property

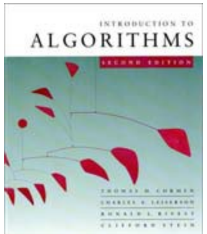
- Property: for any node x:
  - For all nodes y in the left subtree of x:

    $$\text{key}[y] \leq \text{key}[x]$$

  - For all nodes y in the right subtree of x:

    $$\text{key}[y] \geq \text{key}[x]$$





Binary Search Tree (BST)

BST Property

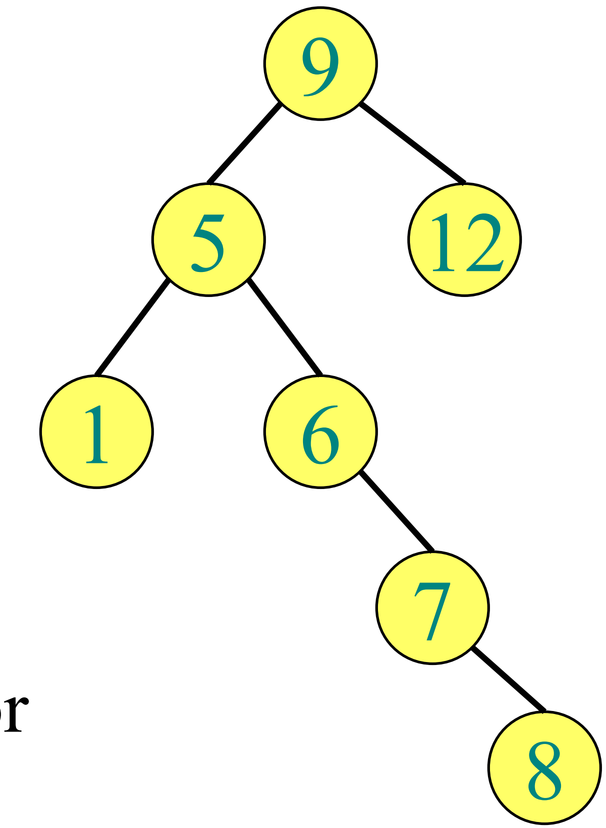
- Property: for any node x:
  - For all nodes y in the left subtree of x:

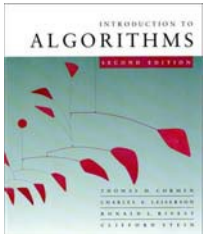
    $$key[y] \leq key[x]$$

  - For all nodes y in the right subtree of x:

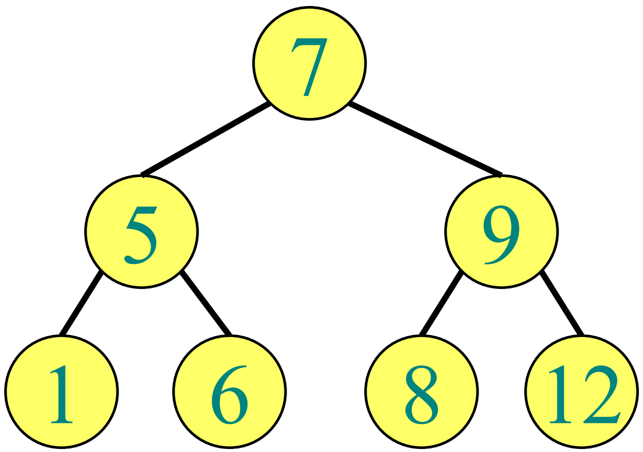
    $$key[y] \geq key[x]$$

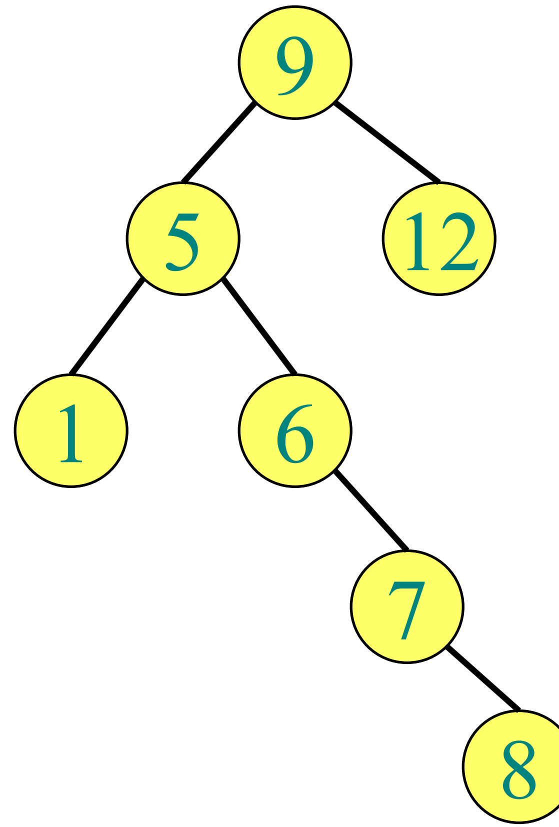
- Given a set of keys, is BST for those keys unique?

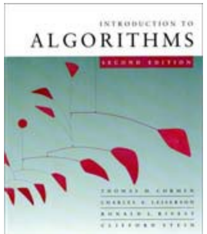




No uniqueness

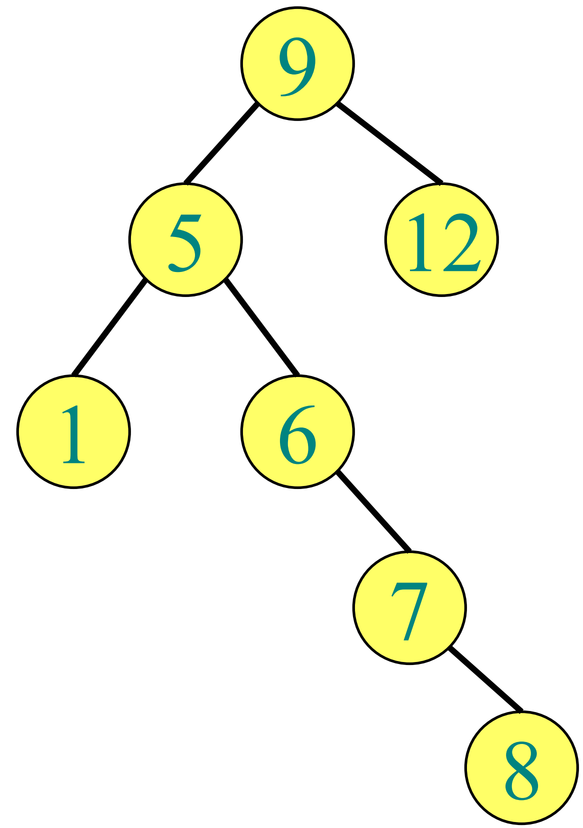


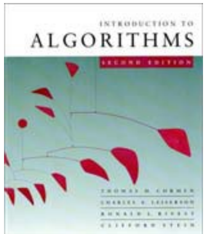




What can we do given BST ?

- Sort !
- Inorder-Walk(x):

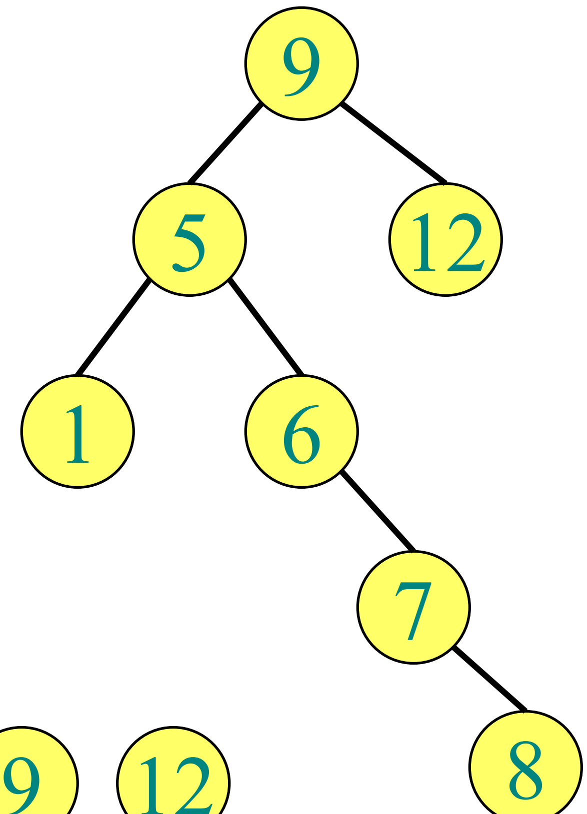




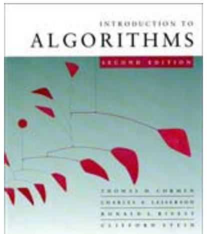
What can we do given BST ?

- Sort !
- Inorder-Walk(x):

  If x≠NIL then
  - Inorder-Walk(left[x])
  - print key[x]
  - Inorder-Walk(right[x])

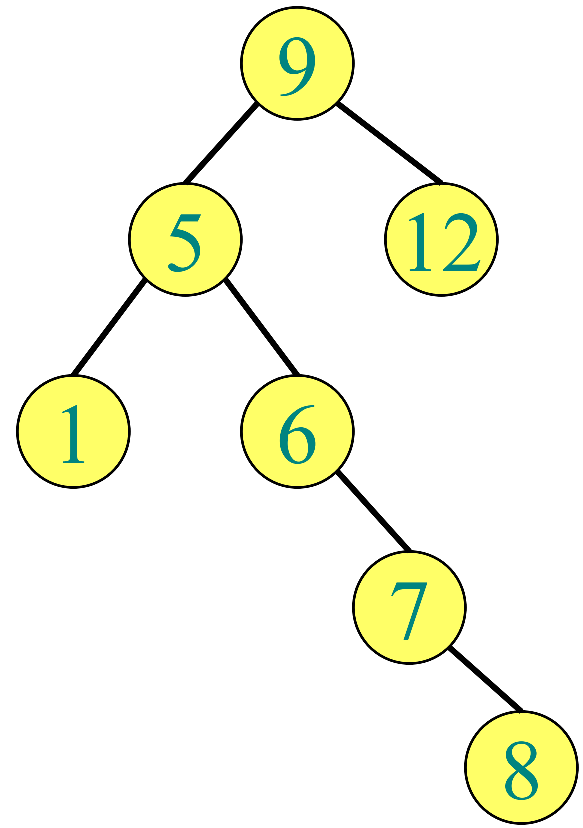


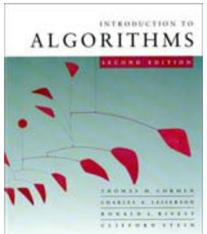
- Output: 1 5 6 7 8 9 12



Sorting, ctd.

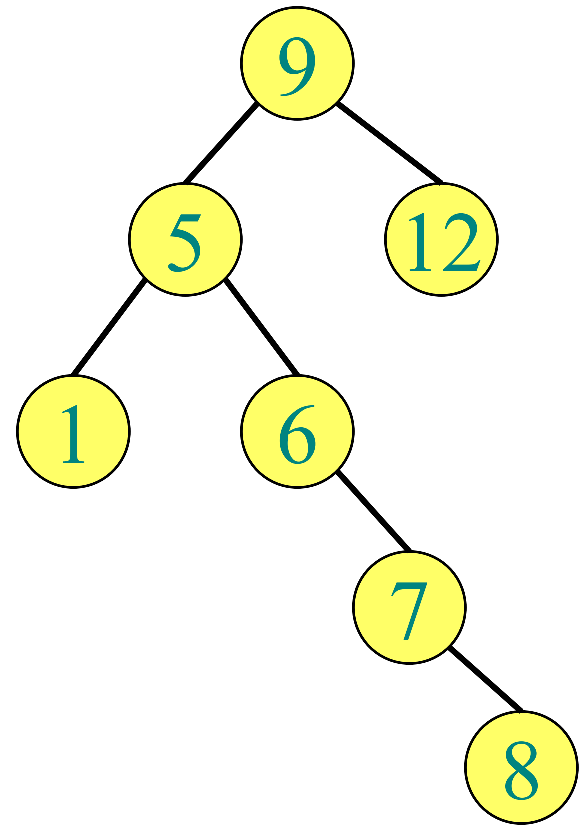
- What is the running time of Inorder-Walk?

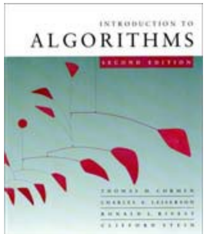




Sorting, ctd.

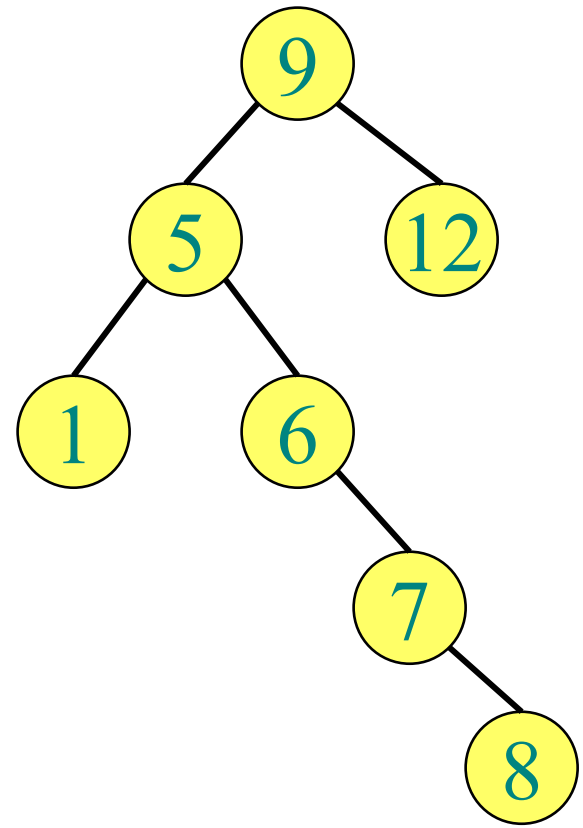
- What is the running time of Inorder-Walk?
- It is O(n)
- Because:
  - Each link is traversed twice
  - There are O(n) links

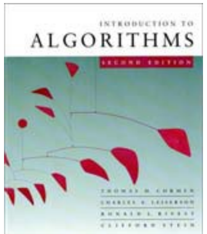




Sorting, ctd.

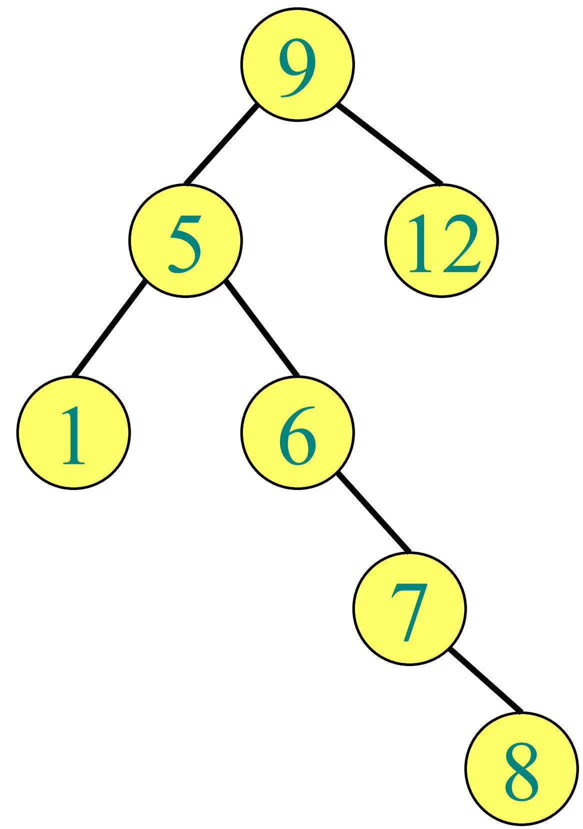
- Does it mean that we can sort n keys in O(n) time ?

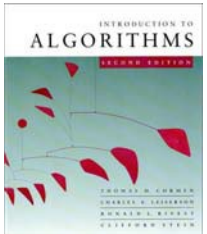




Sorting, ctd.

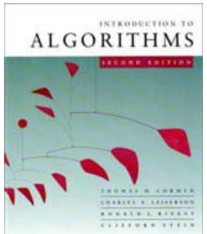
- Does it mean that we can sort n keys in $O(n)$ time ?
- No
- It just means that building a BST takes $\Omega(n \log n)$ time
(in the comparison model)





BST as a data structure

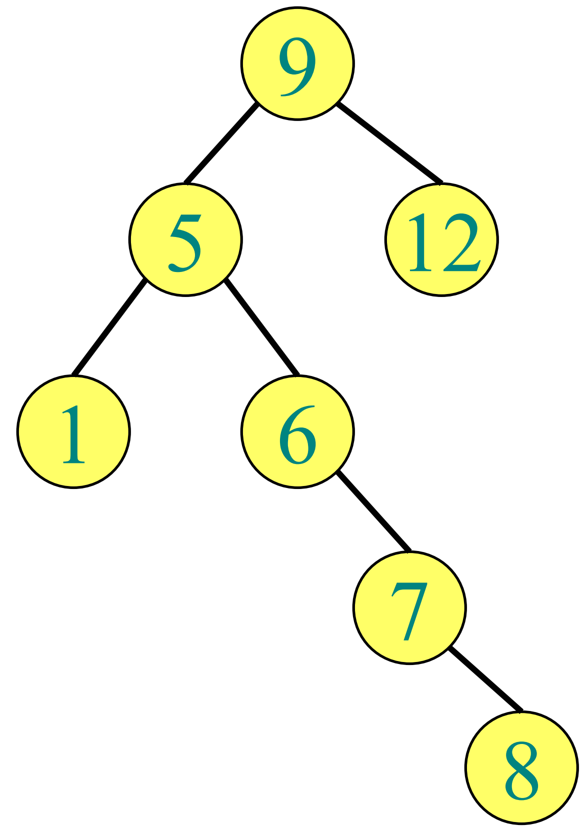
- Operations:
  - Insert(x)
  - Delete(x)
  - Search(k)

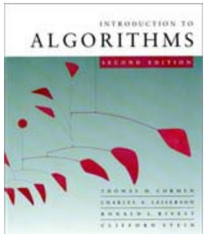


Search

Search(x):

- If x≠NIL then
  - If key[x] = k then return x
  - If k < key[x] then return Search(left[x])
  - If k > key[x] then return Search(right[x])
- Else return NIL

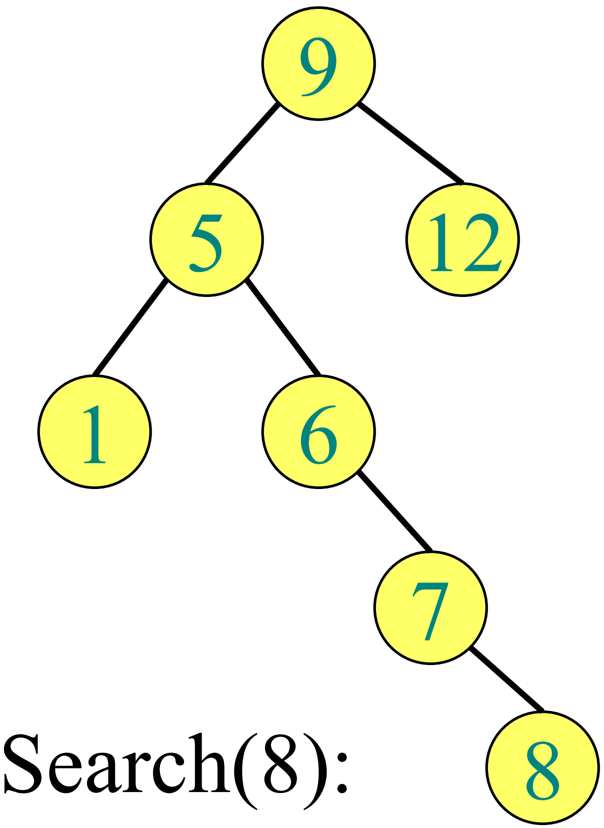




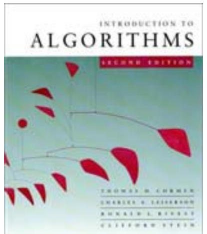
Search

Search(x):

- If x≠NIL then
  - If key[x] = k then return x
  - If k < key[x] then return Search(left[x])
  - If k > key[x] then return Search(right[x])
- Else return NIL



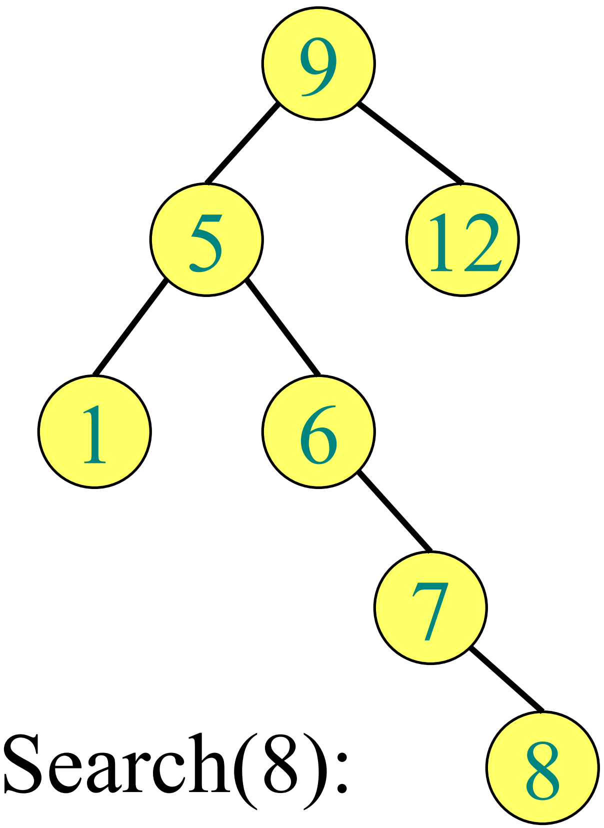
Search(8):



Search

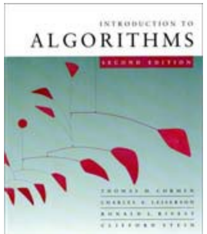
Search(x):

- If x≠NIL then
  - If key[x] = k then return x
  - If k < key[x] then return Search(left[x])
  - If k > key[x] then return Search(right[x])
- Else return NIL



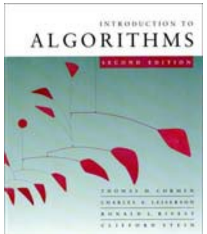
Search(8):

Search(8.5):



Predecessor/Successor

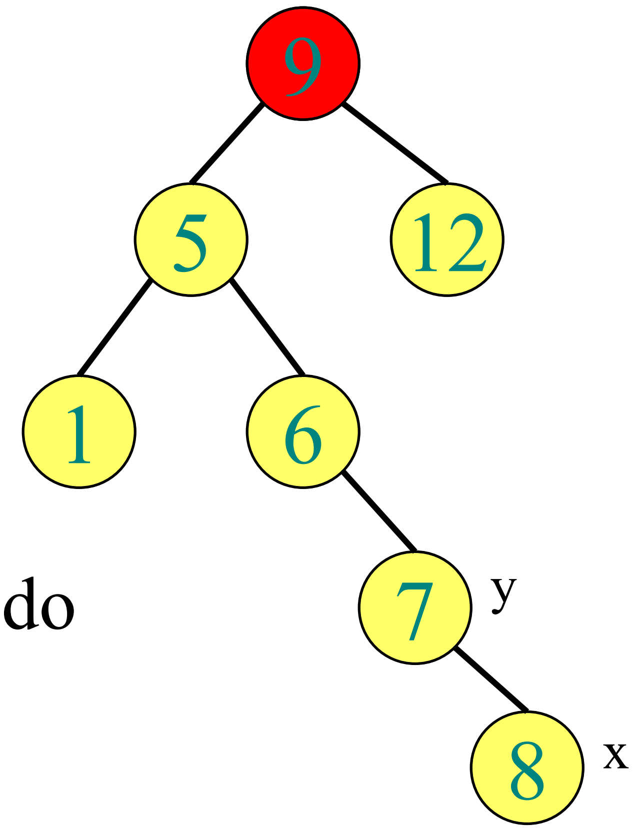
- Can modify Search (into Search') such that, if k is not stored in BST, we get x such that:
  - Either it has the largest key[x]<k, or
  - It has the smallest key[x]>k
- Useful when k prone to errors
- What if we always want a successor of k ?
  - x=Search'(k)
  - If key[x]<k, then return Successor(x)
  - Else return x



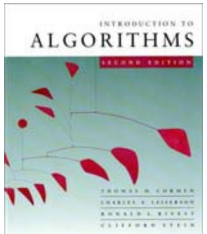
Successor

Successor(x):

- If right[x] ≠ NIL then return Minimum(right[x])
- Otherwise
  - – y ← p[x]
  - – While y≠NIL and x=right[y] do
    - x ← y
    - y ← p[y]
  - – Return y



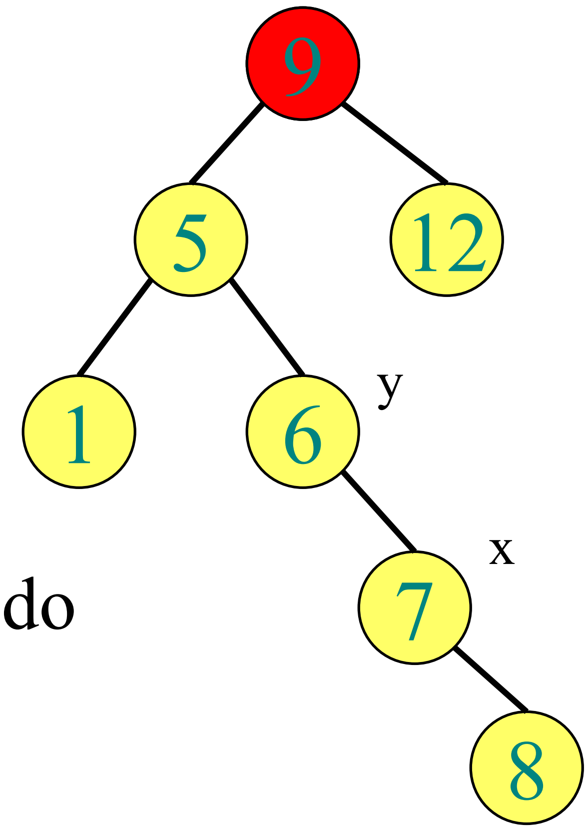
The lowest ancestor that has x in the left subtree

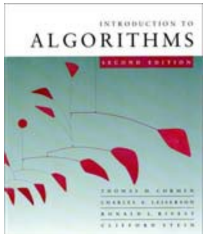


Successor

Successor(x):

- If right[x] ≠ NIL then return Minimum(right[x])
- Otherwise
  - – y ← p[x]
  - – While y≠NIL and x=right[y] do
    - x ← y
    - y ← p[y]
  - – Return y

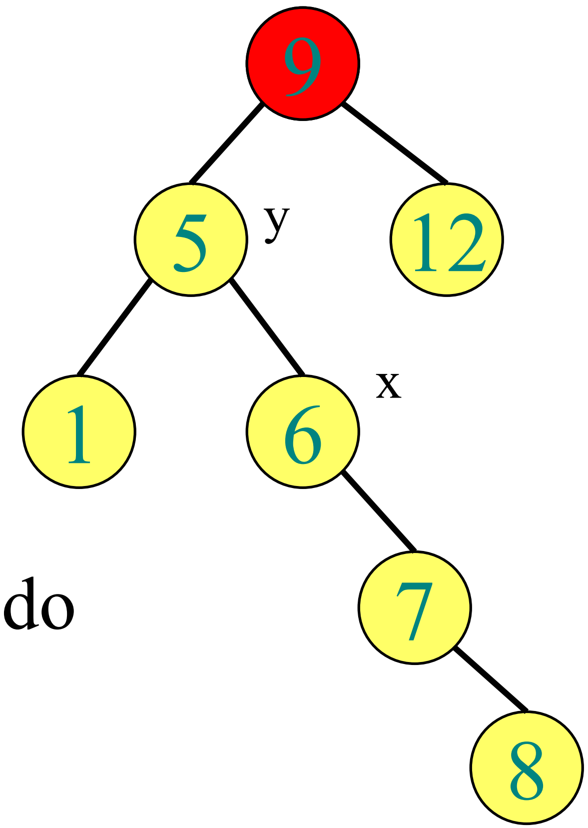


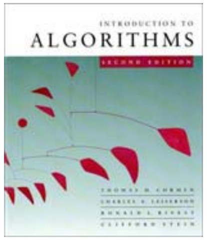


Successor

Successor(x):

- If right[x] ≠ NIL then return Minimum(right[x])
- Otherwise
  - y ← p[x]
  - While y≠NIL and x=right[y] do
    - x ← y
    - y ← p[y]
  - Return y

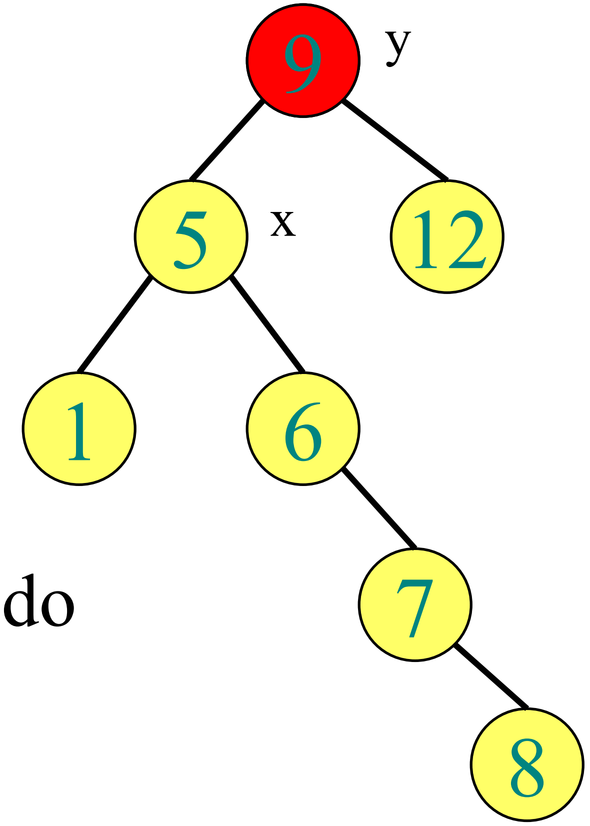


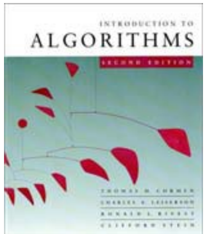


Successor

Successor(x):

- If right[x] ≠ NIL then return Minimum(right[x])
- Otherwise
  - $y \leftarrow p[x]$
  - While $y \neq$ NIL and $x =$ right[y] do
    - $x \leftarrow y$
    - $y \leftarrow p[y]$
  - Return y

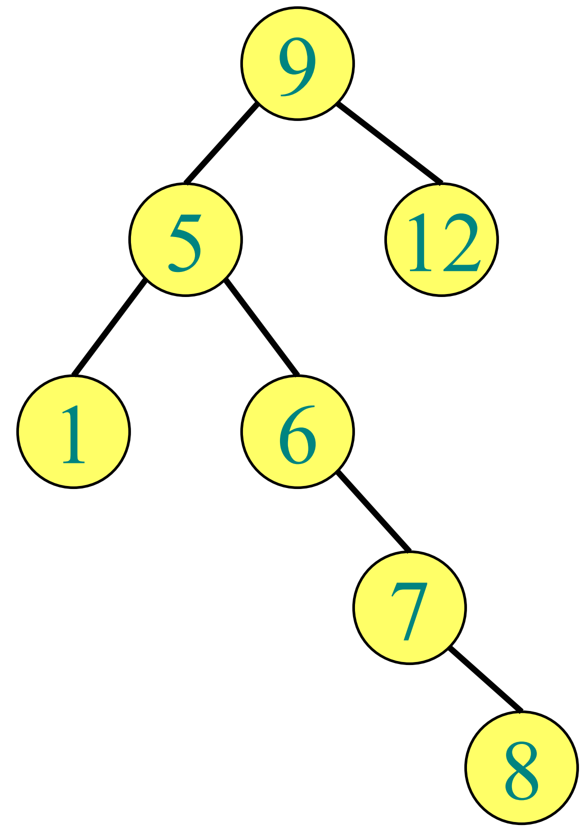


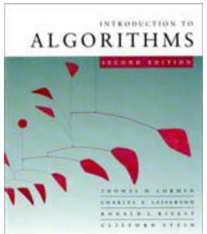


Minimum

Minimum(x)

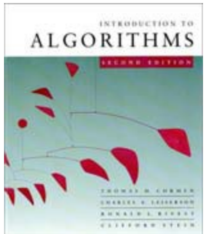
- While left[x]≠NIL do
  - x ← left[x]
- Return x





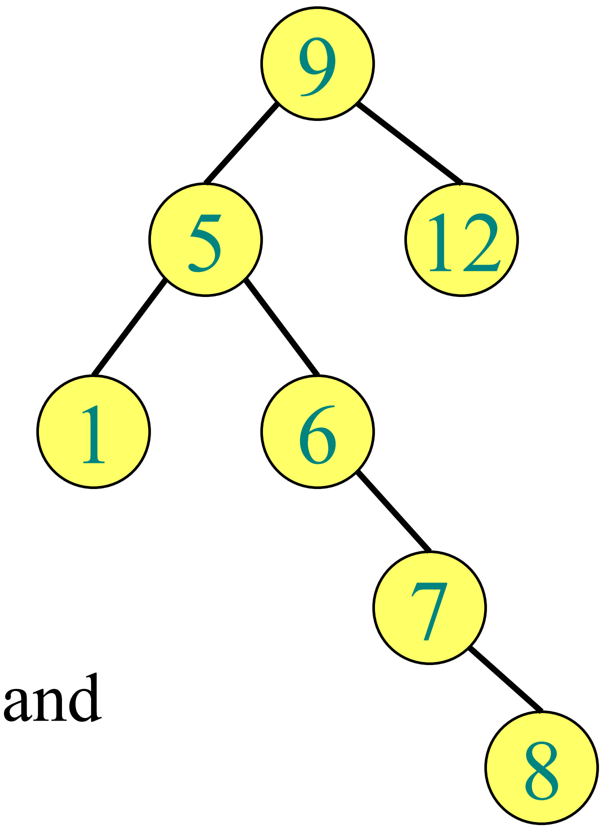
Nearest Neighbor

- Assuming keys are numbers
- For a key k, can we find x such that |k-key[x]| is minimal ?
- Yes:
  - key[x] must be either a predecessor or successor of k
  - y=Search'(k)   //y is either succ or pred of k
  - y' =Successor(y)
  - y''=Predecessor(y)
  - Report the closest of key[y], key[y'], key[y'']

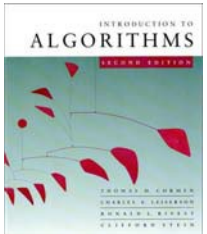


Analysis

- How much time does all of this take ?
- Worst case: O(height)
- Height really important
- Tree better be balanced



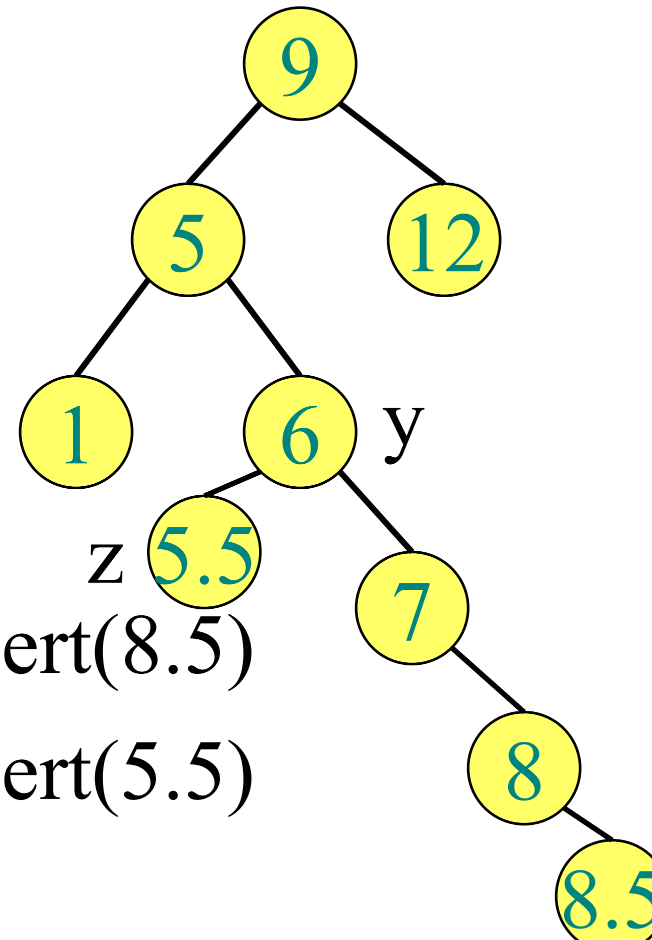
Why? Height is lg n if balanced tree and n if unbalanced.



Constructing BST

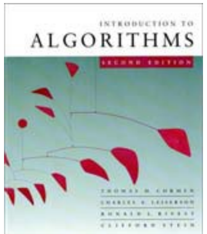
Insert(z):

- y ← NIL
- x ← root
- While x ≠ NIL do
  - y ← x
  - If key[z] < key[x]
    then x ← left[x]
    else x ← right[x]
- p[z] ← y
- If key[z] < key[y]
  then left[y] ← z
  else right[y] ← z

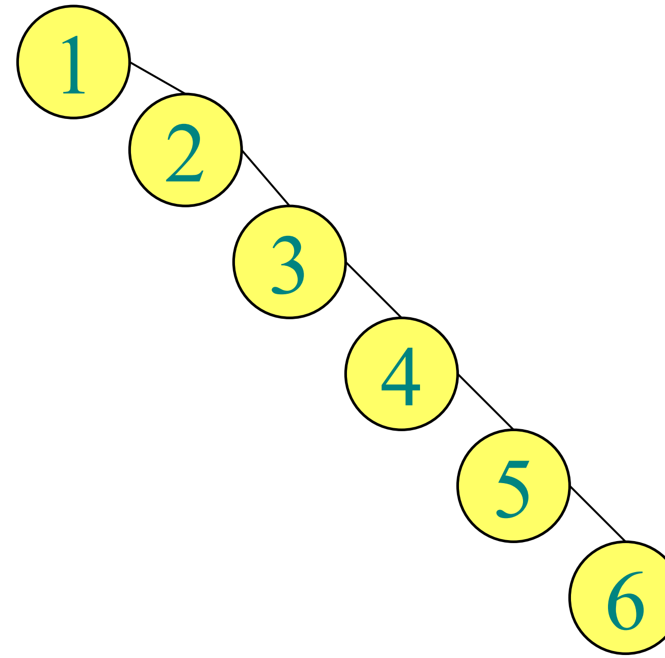


Insert(8.5)

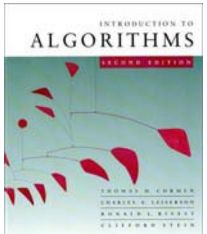
Insert(5.5)



Analysis

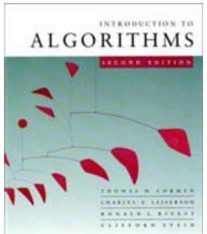


- After we insert n elements, what is the worst possible BST height ?
- Pretty bad: n-1



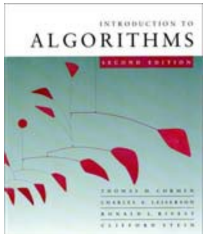
Average case analysis

- Consider keys 1,2,…,n, in a random order
- Each permutation equally likely
- For each key perform Insert
- What is the likely height of the tree ?



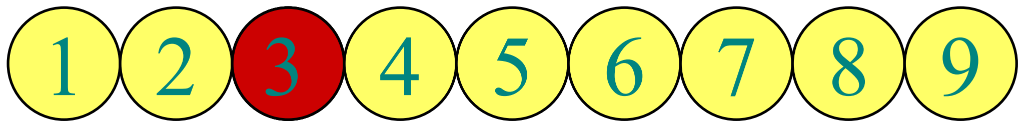
Average case analysis

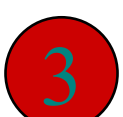
- Consider keys 1,2,…,n, in a random order
- Each permutation equally likely
- For each key perform Insert
- What is the likely height of the tree ?
- It is O(log n)

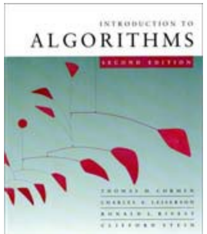


Creating a random BST

- n=9

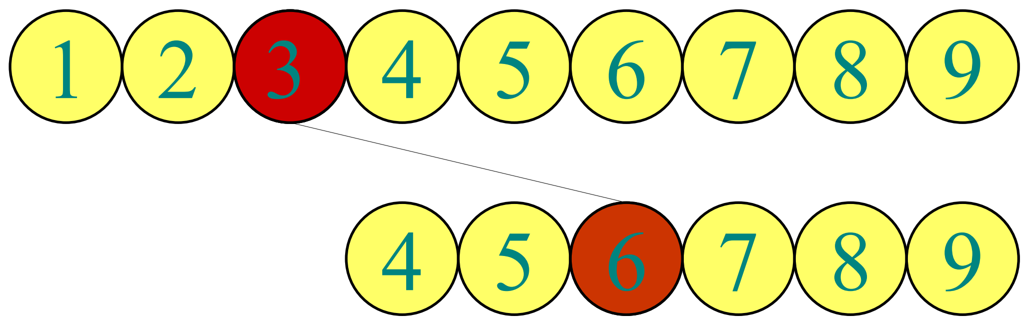


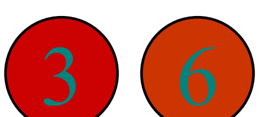


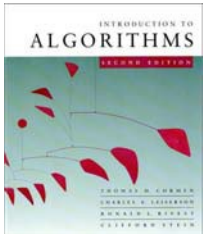


Creating a random BST

- n=9

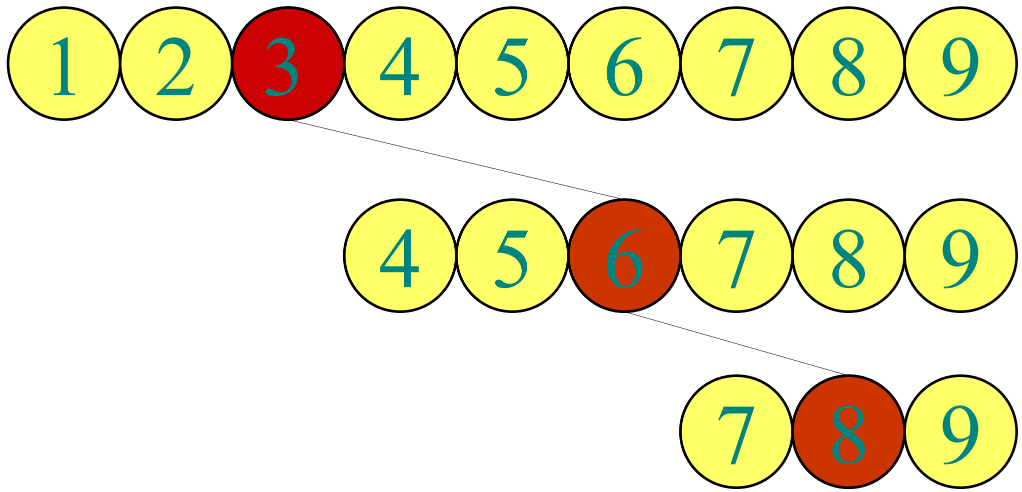


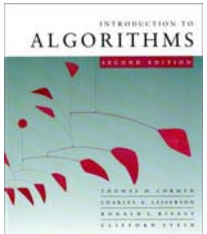




Creating a random BST

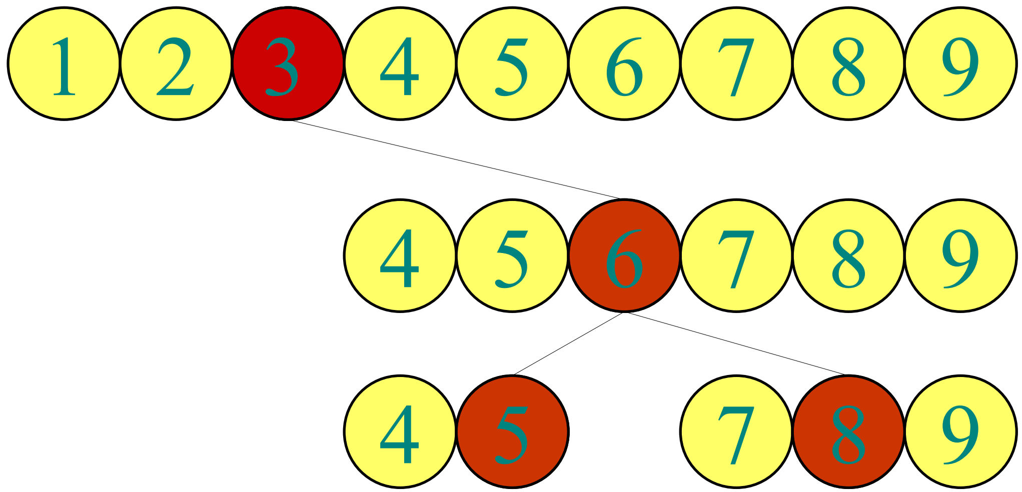
- n=9

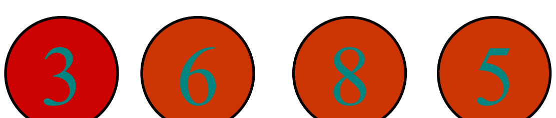


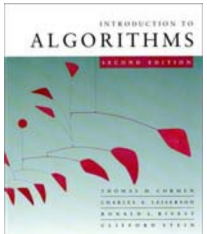


Creating a random BST

- n=9

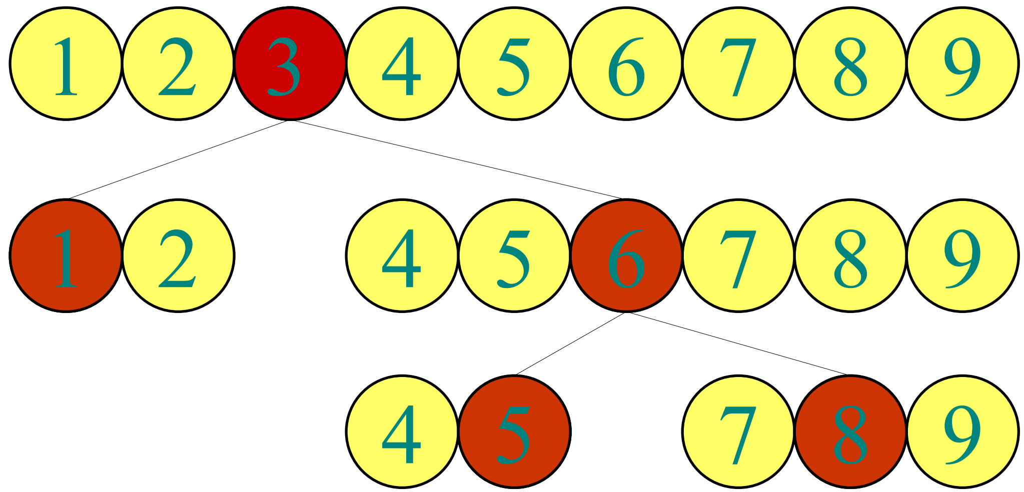


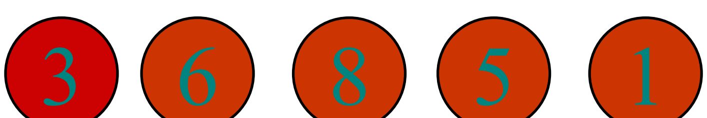


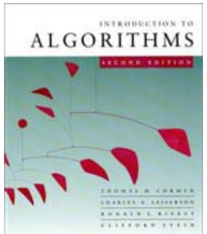


Creating a random BST

- n=9

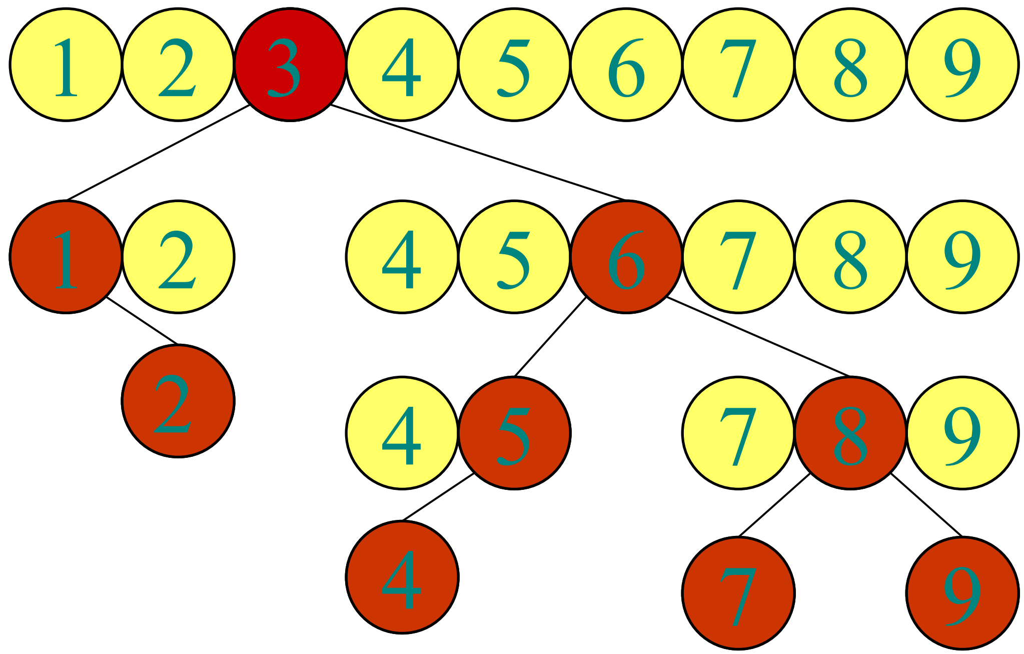


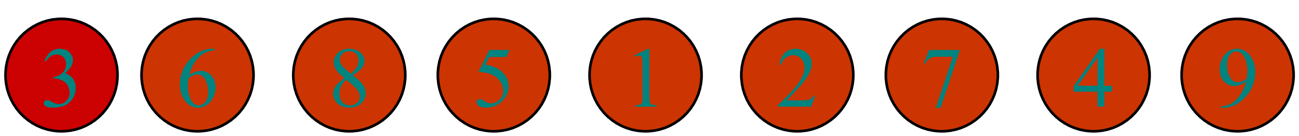




Creating a random BST

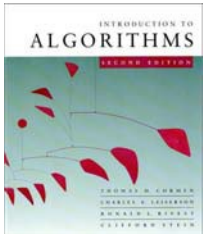
- n=9





Random BST

- Expected height?
  - $O(\lg n)$
- Proof in the book!
- What if we want height to be $O(\lg n)$ in the worst case?
  - Red-Black Trees

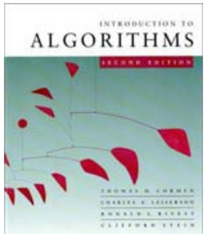


Balanced search trees

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

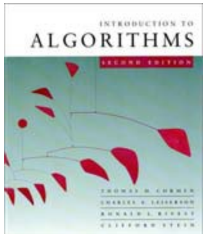


Red-black trees

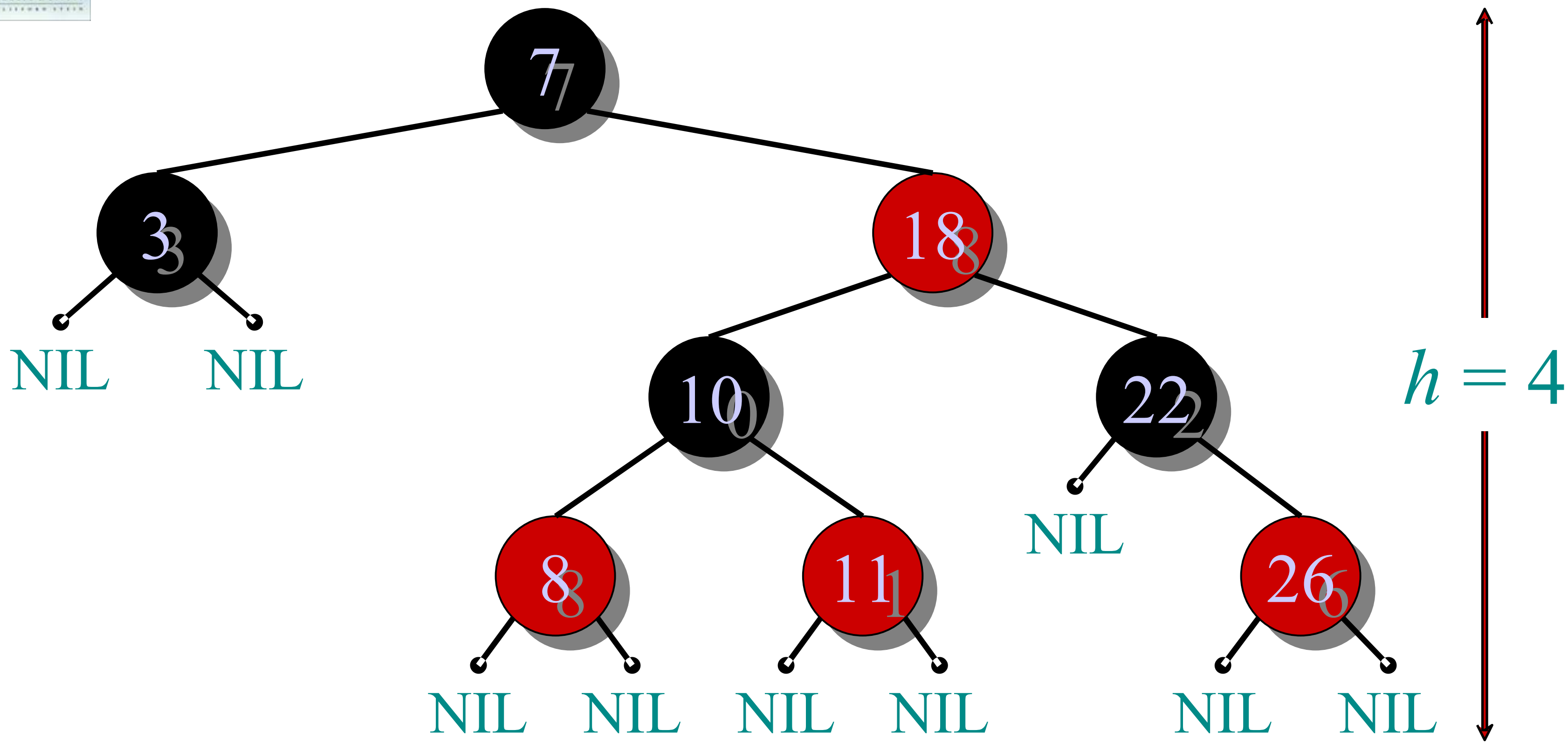
This data structure requires an extra one-bit color field in each node.

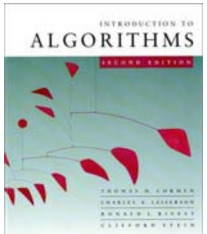
Red-black properties:

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).

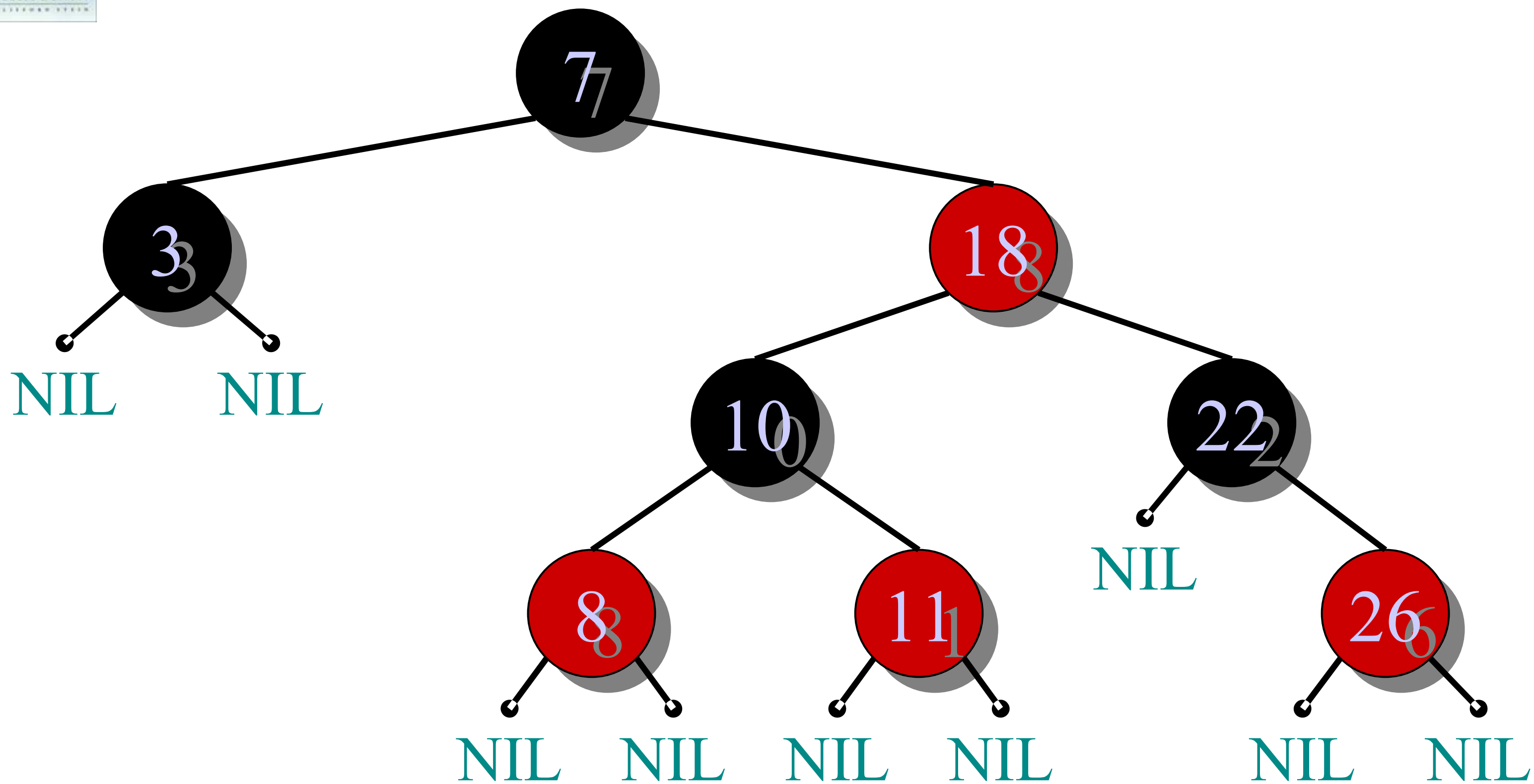


Example of a red-black tree

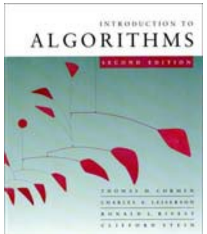




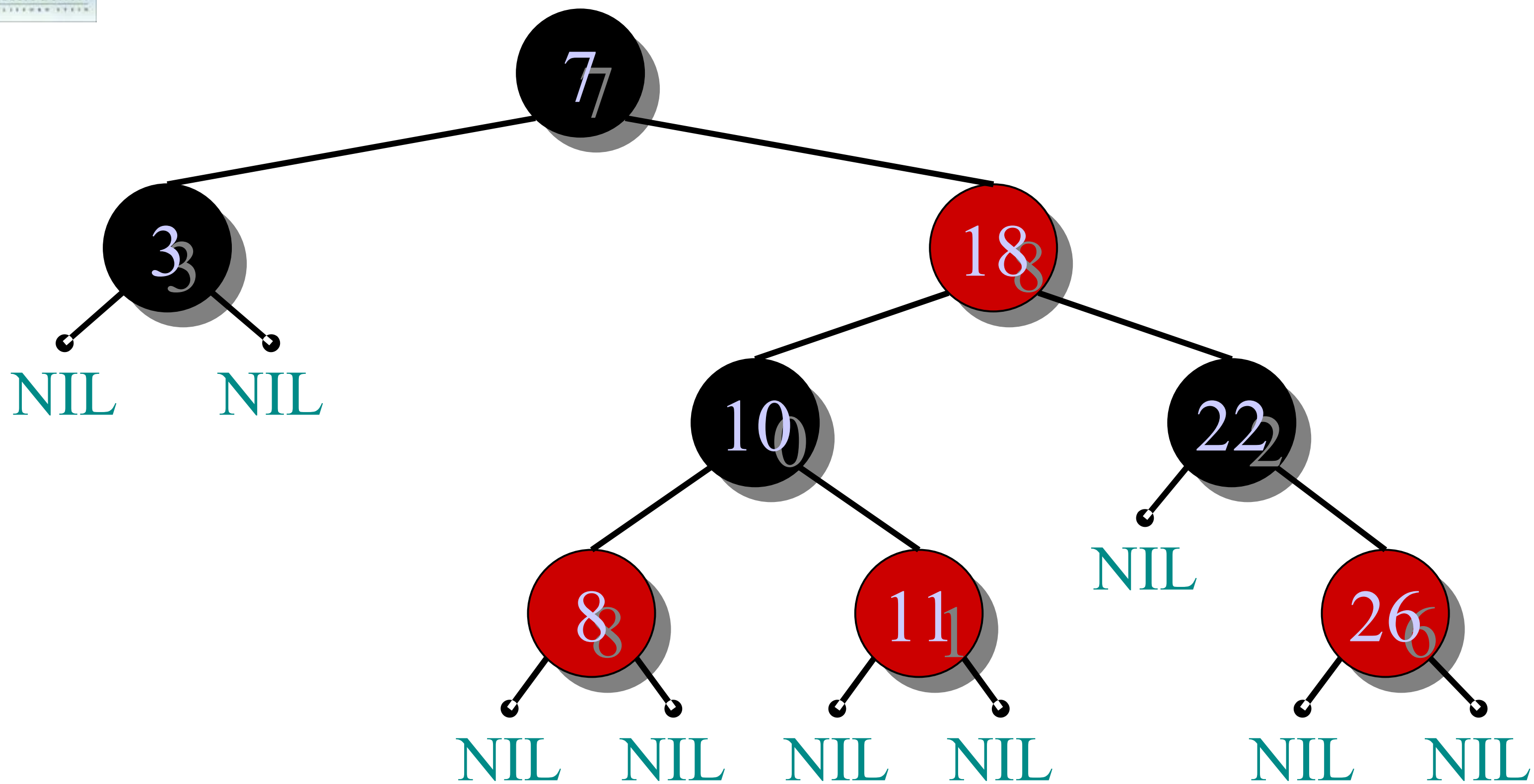
Example of a red-black tree



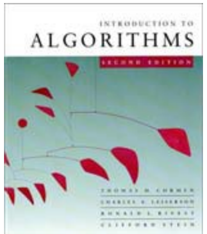
1. Every node is either red or black.



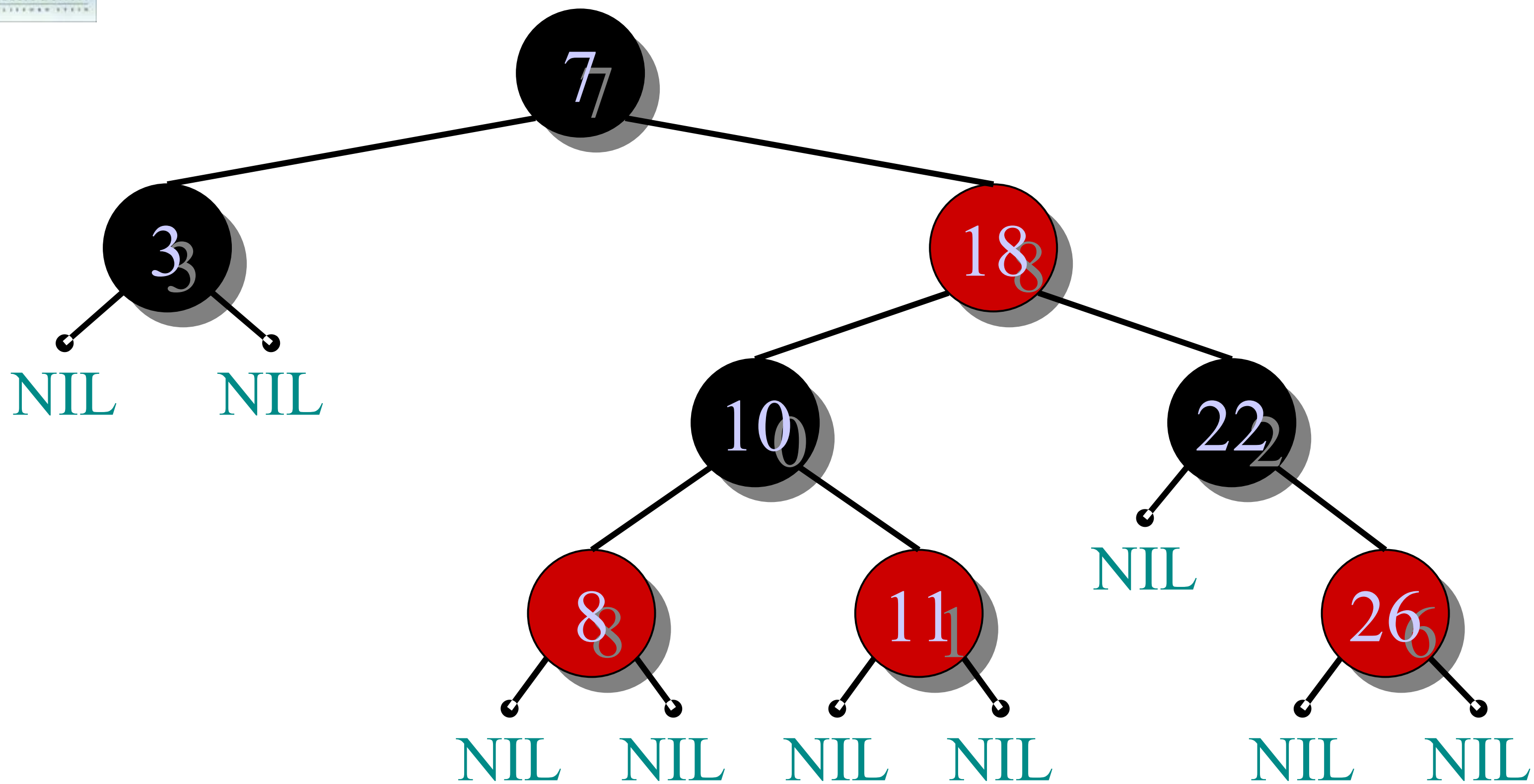
Example of a red-black tree



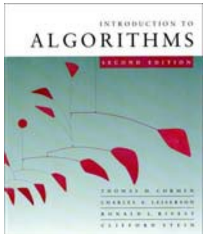
2. The root and leaves (NIL's) are black.



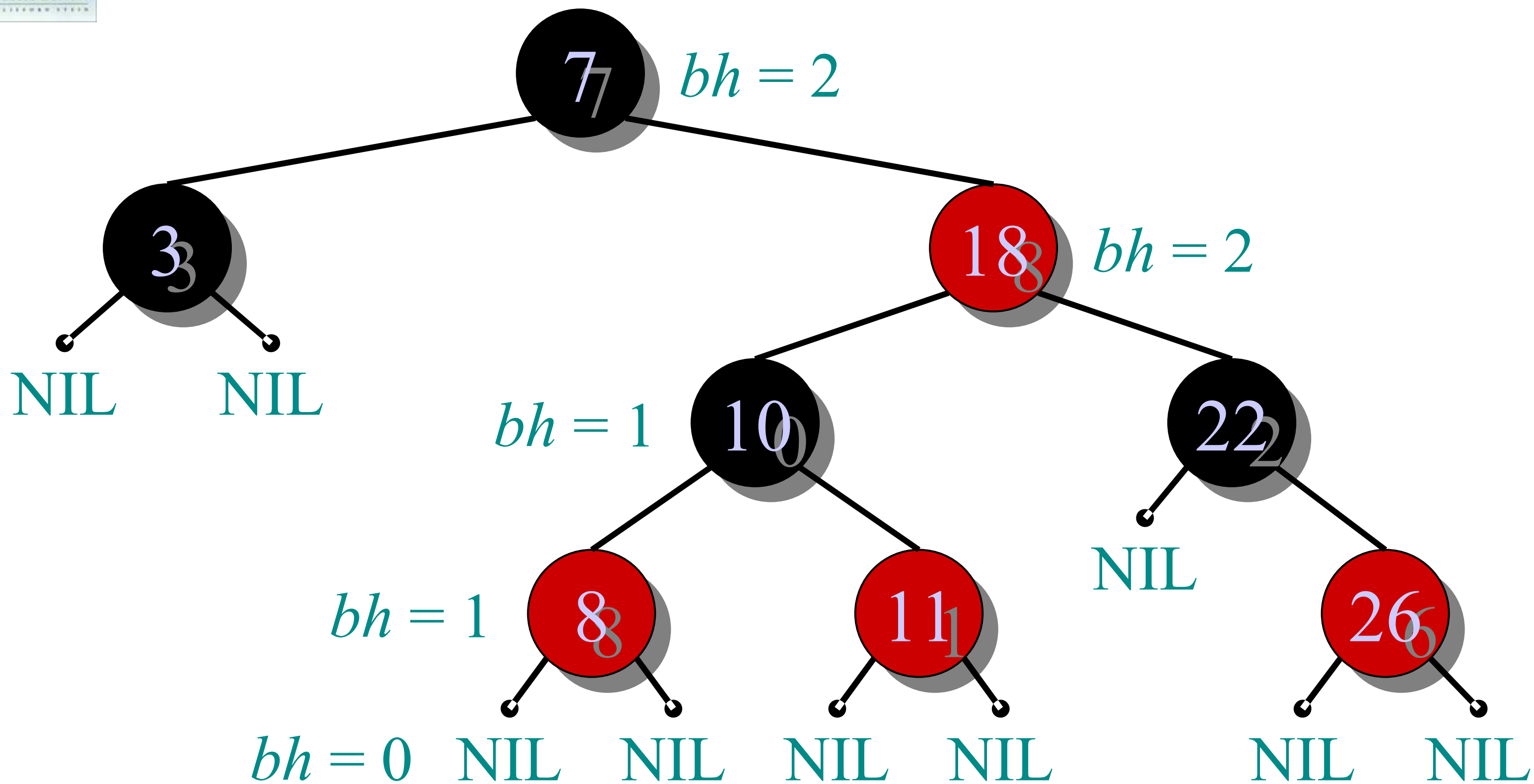
Example of a red-black tree



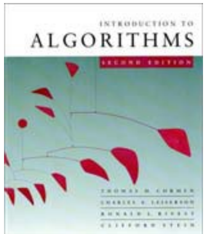
3. If a node is red, then its parent is black.



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

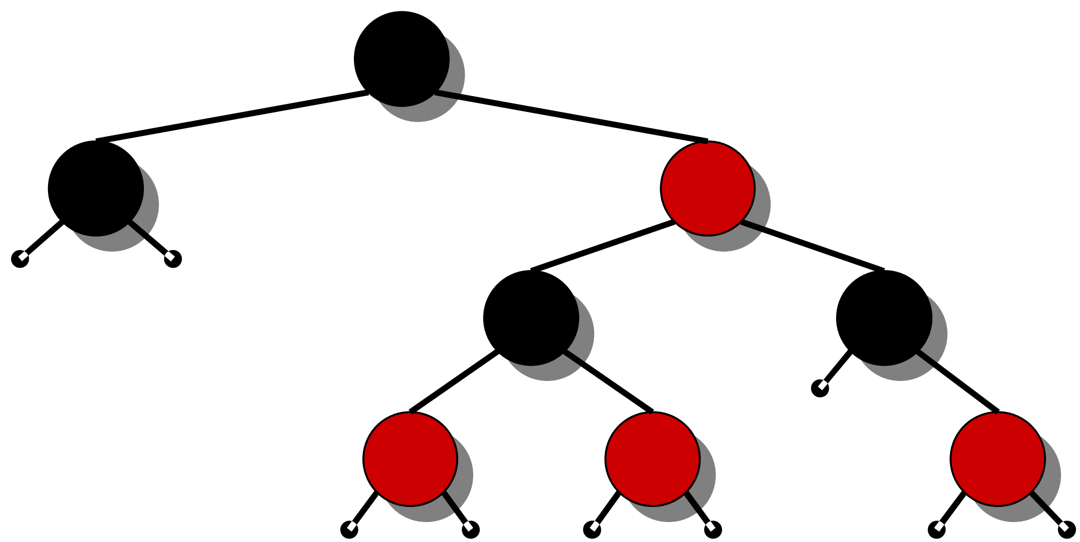
Theorem. A red-black tree with n keys has height

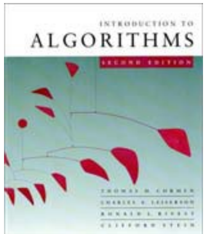
$$h \leq 2\lg(n + 1).$$

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

INTUITION:

- Merge red nodes into their black parents.





Height of a red-black tree

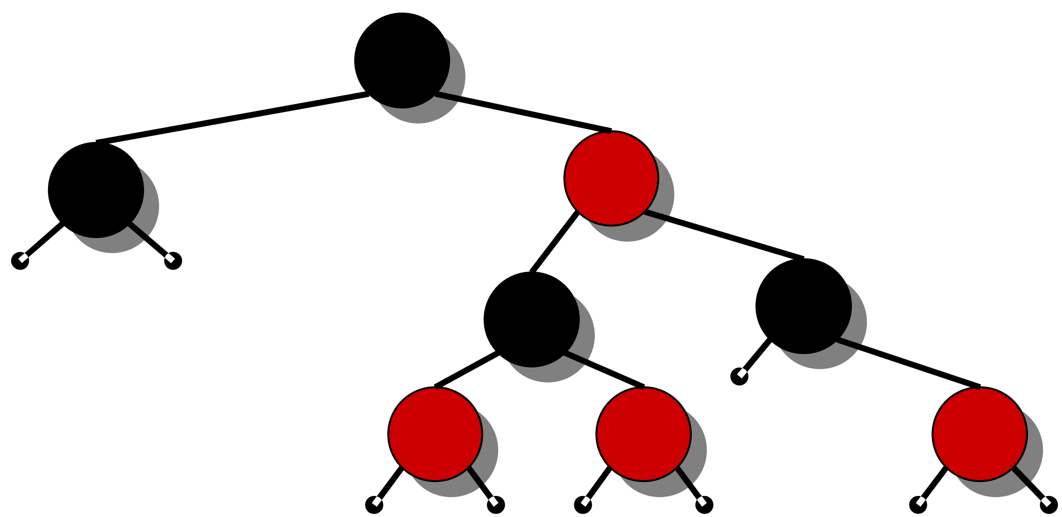
Theorem. A red-black tree with n keys has height

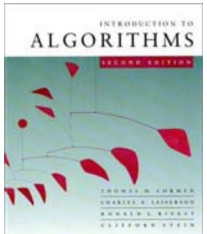
$$h \leq 2 \lg(n + 1).$$

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

INTUITION:

- Merge red nodes into their black parents.





Height of a red-black tree

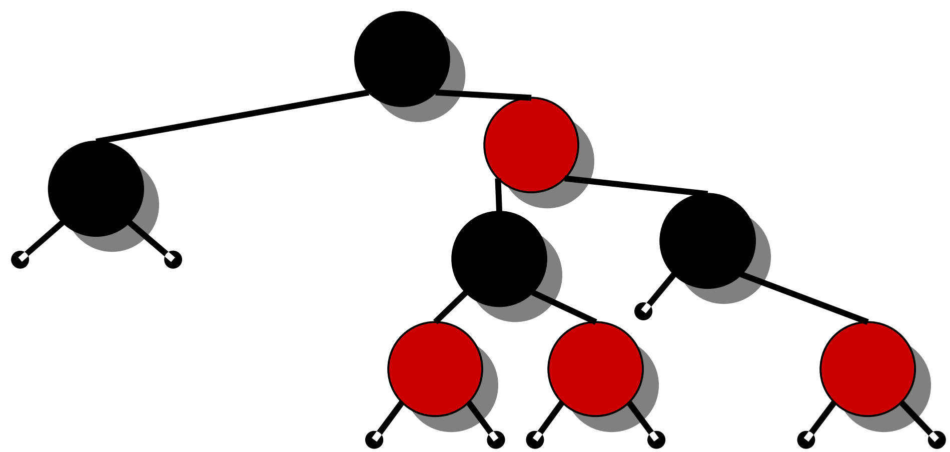
Theorem. A red-black tree with n keys has height

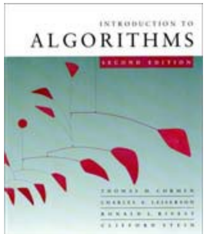
$$h \leq 2 \lg(n + 1).$$

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

INTUITION:

- Merge red nodes into their black parents.





Height of a red-black tree

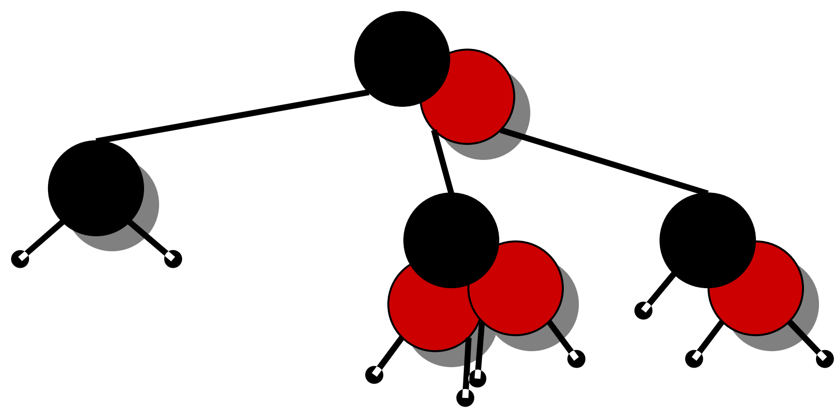
Theorem. A red-black tree with n keys has height

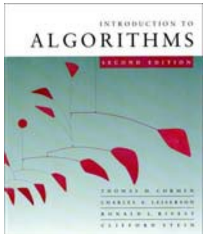
$$h \le 2\lg(n+1).$$

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

INTUITION:

- Merge red nodes into their black parents.





Height of a red-black tree

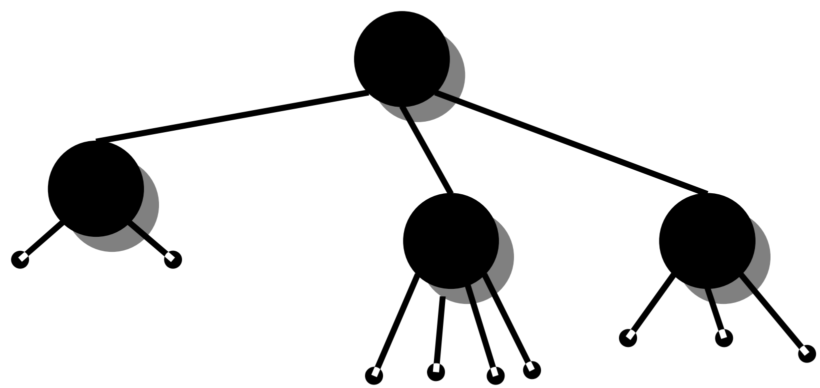
Theorem. A red-black tree with n keys has height

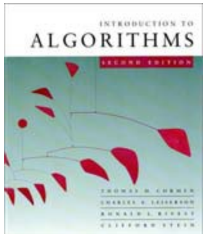
$$h \leq 2\lg(n + 1).$$

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

INTUITION:

- Merge red nodes into their black parents.





Height of a red-black tree

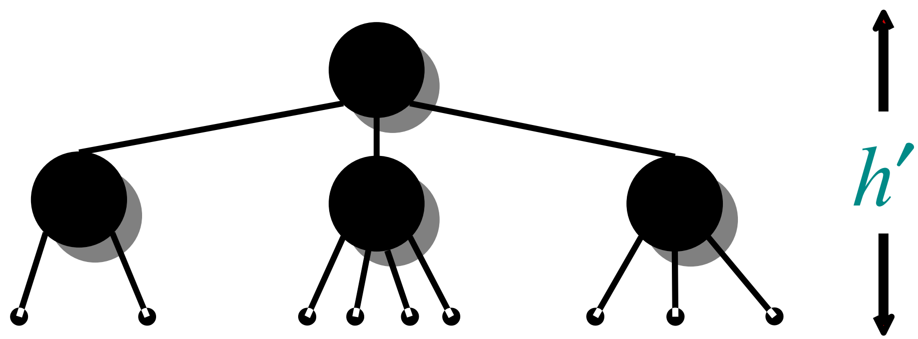
Theorem. A red-black tree with n keys has height

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

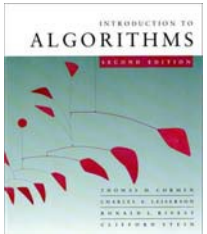
Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.

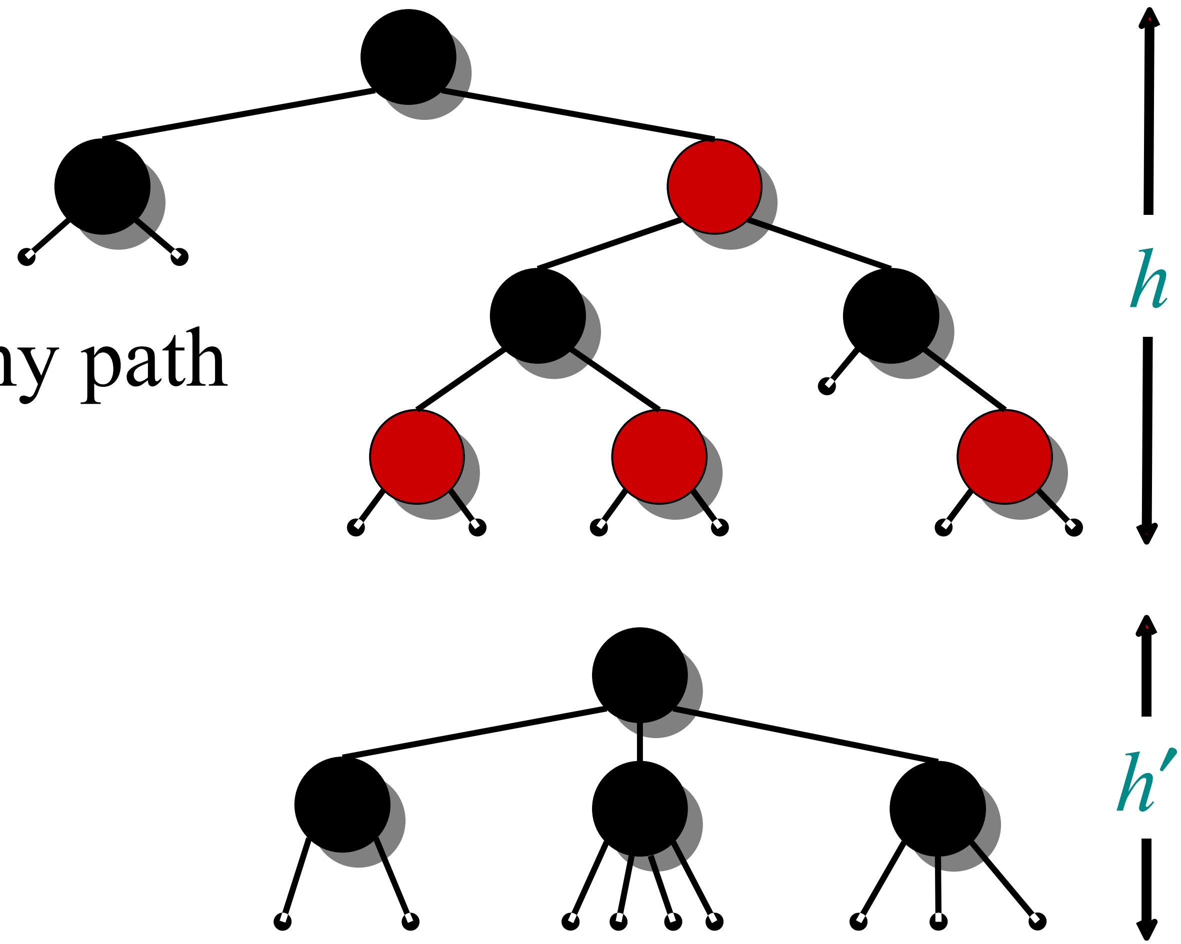


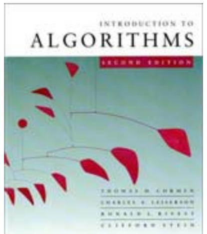
- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth h' of leaves.



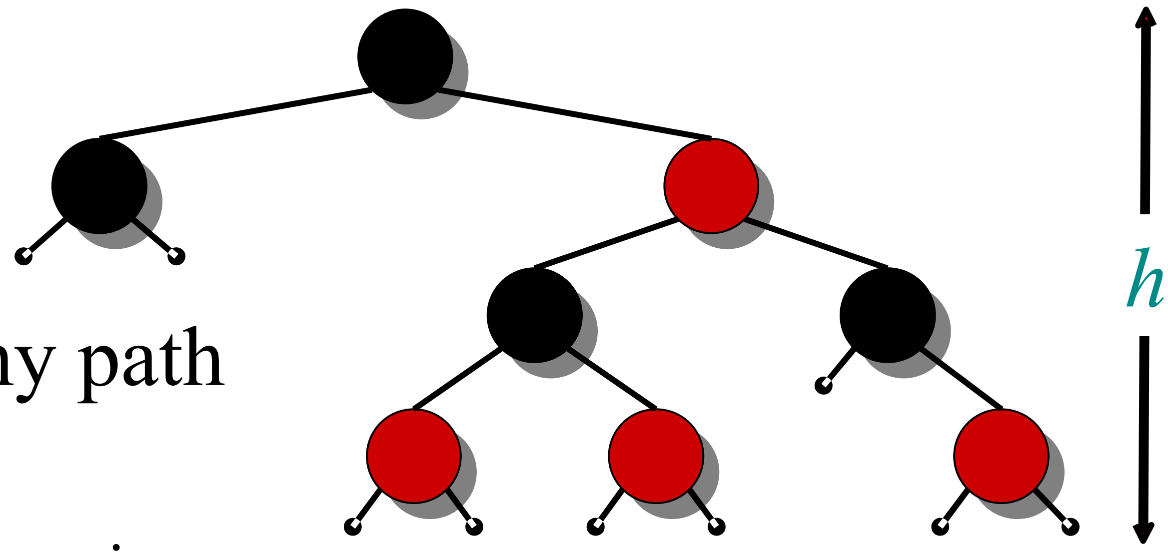
Proof (continued)

- We have $h' \geq h/2$, since at most half the leaves on any path are red.



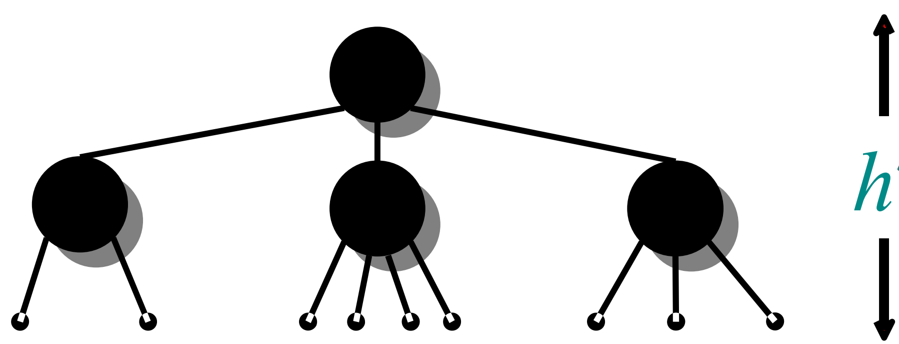


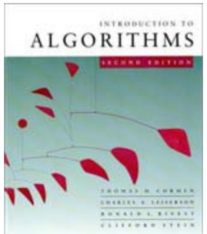
Proof (continued)



- We have $h' \geq h/2$, since at most half the leaves on any path are red.

- The number of leaves in a RBT is $n + 1$ (every node inserted adds two children and removes one except the root) which is at least $2^{h'}$ if we assume only 2 children





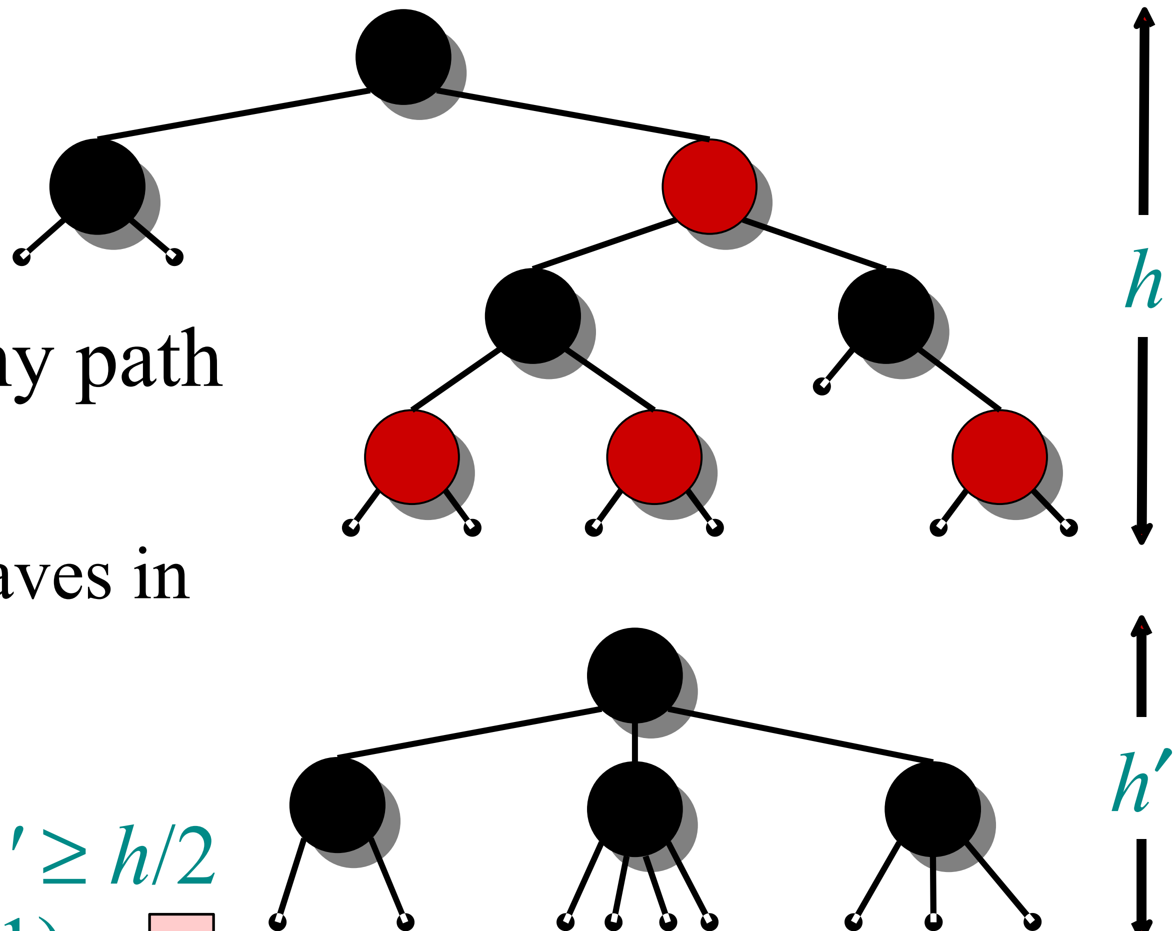
Proof (continued)

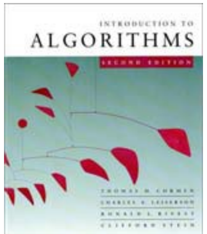
- We have $h' \geq h/2$, since at most half the leaves on any path are red.
- The number of leaves in a RBT is $n + 1$

$\Rightarrow n + 1 \geq 2^{h'}$

$\Rightarrow \lg(n + 1) \geq h' \geq h/2$

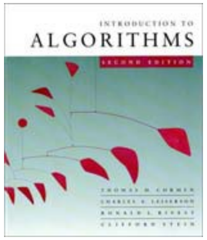
$\Rightarrow h \leq 2 \lg(n + 1)$. $\square$





Query operations

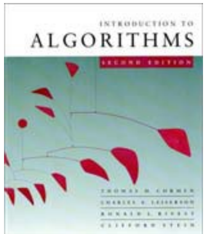
Corollary. The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR all run in $O(\lg n)$ time on a red-black tree with n nodes.



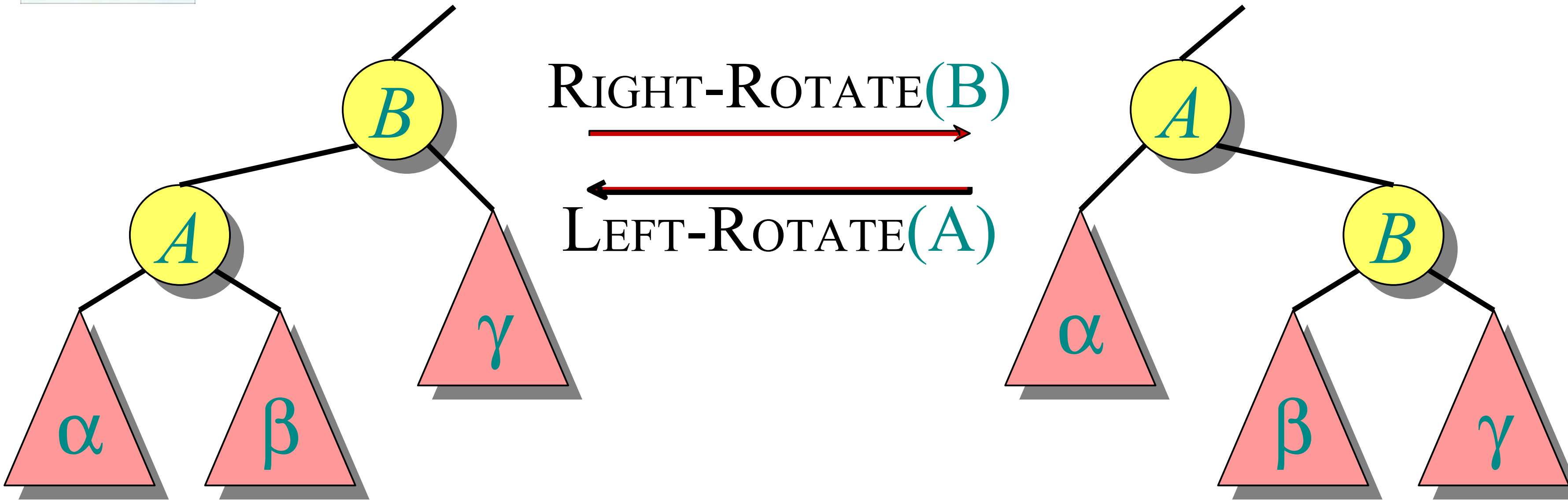
Modifying operations

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

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



Rotations



Rotations maintain the inorder ordering of keys:

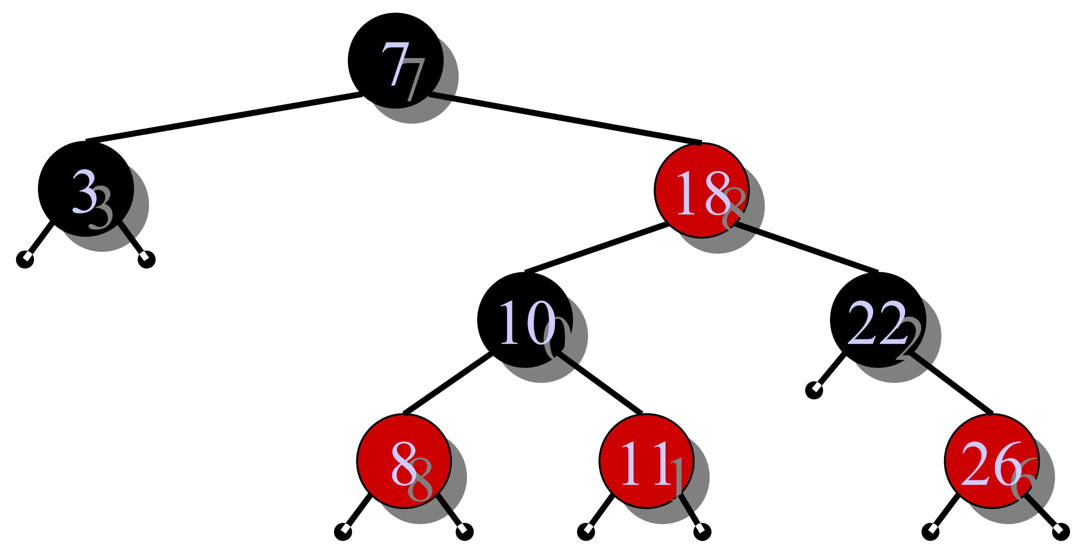
- $a \in \alpha$, $b \in \beta$, $c \in \gamma \Rightarrow a \le A \le b \le B \le c$.

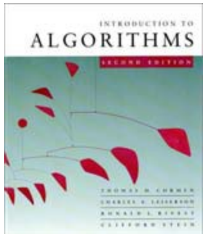
A rotation can be performed in $O(1)$ time.

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:



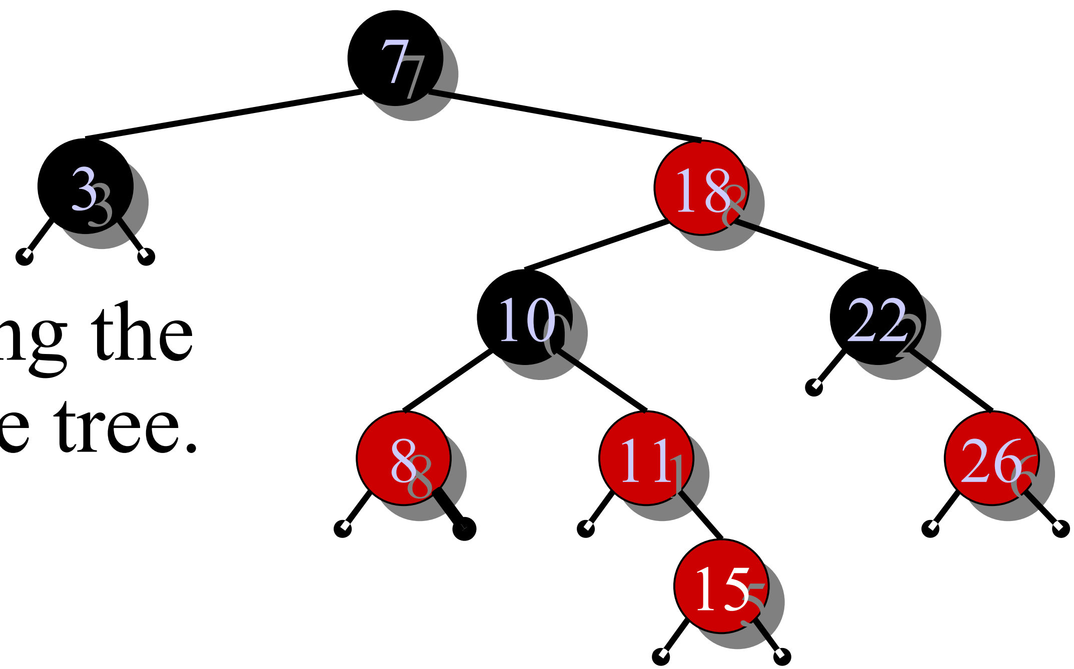


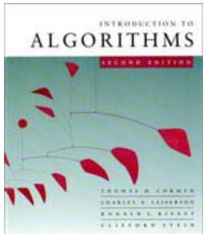
Insertion into a red-black tree

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.



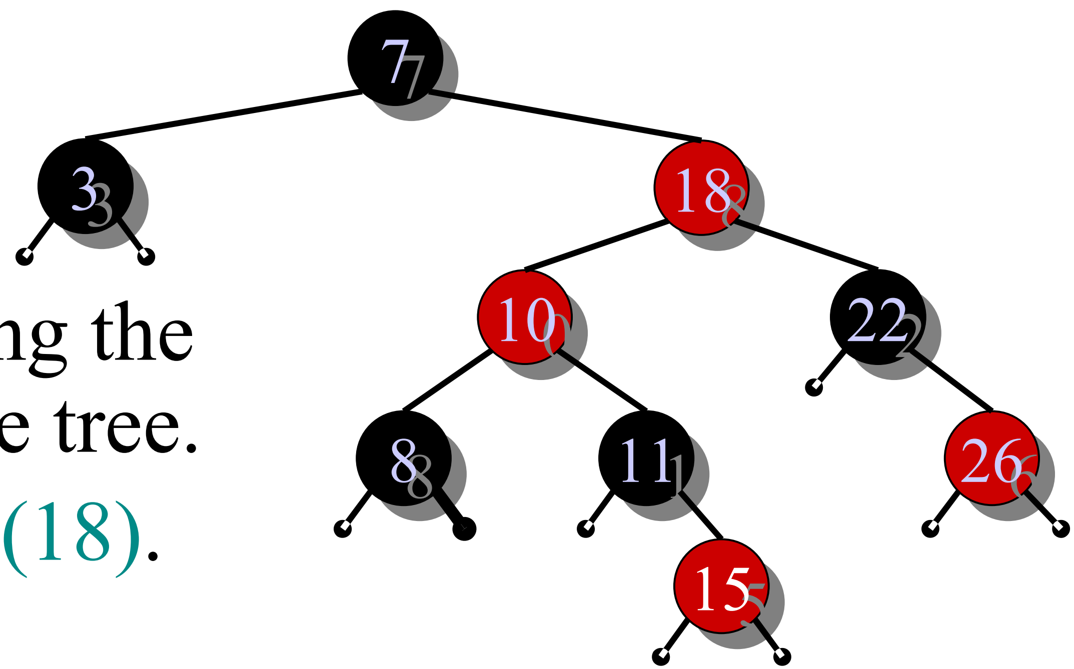


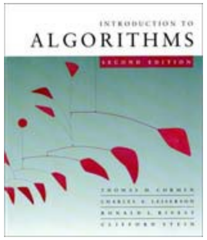
Insertion into a red-black tree

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(18).

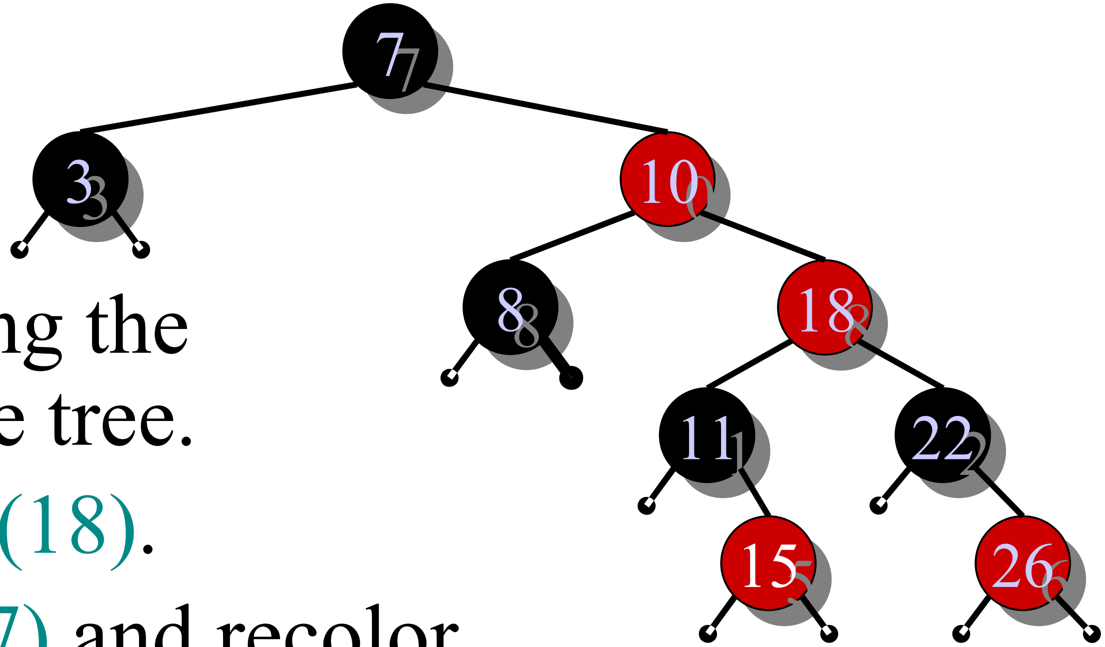




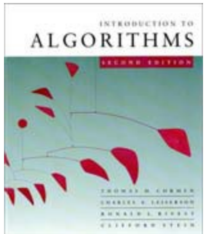
Insertion into a red-black tree

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(18).
- LEFT-ROTATE(7) and recolor.

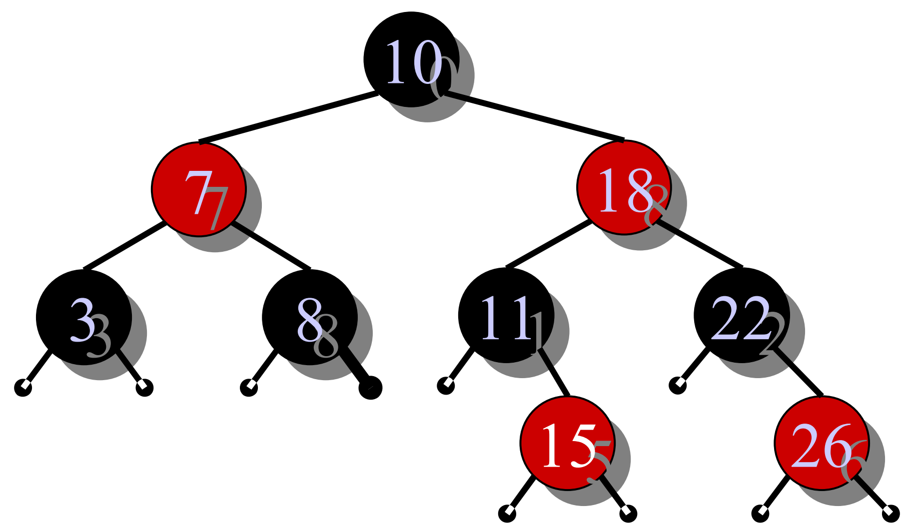


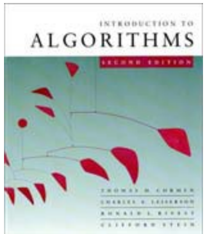
Insertion into a red-black tree

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:

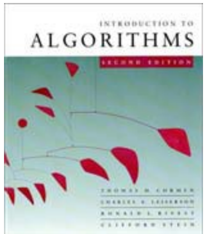
- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- LEFT-ROTATE(7) and recolor.



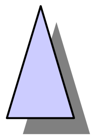


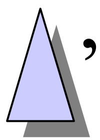
Pseudocode

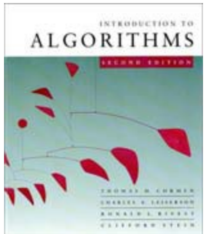
```
RB-INSERT(T, x)
    TREE-INSERT(T, x)
    color[x] ← RED      ⊳ only RB property 3 can be violated
    while x ≠ root[T] and color[p[x]] = RED
        do if p[x] = left[p[p[x]]]
            then y ← right[p[p[x]]]      ⊳ y = aunt/uncle of x
                 if color[y] = RED
                  then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



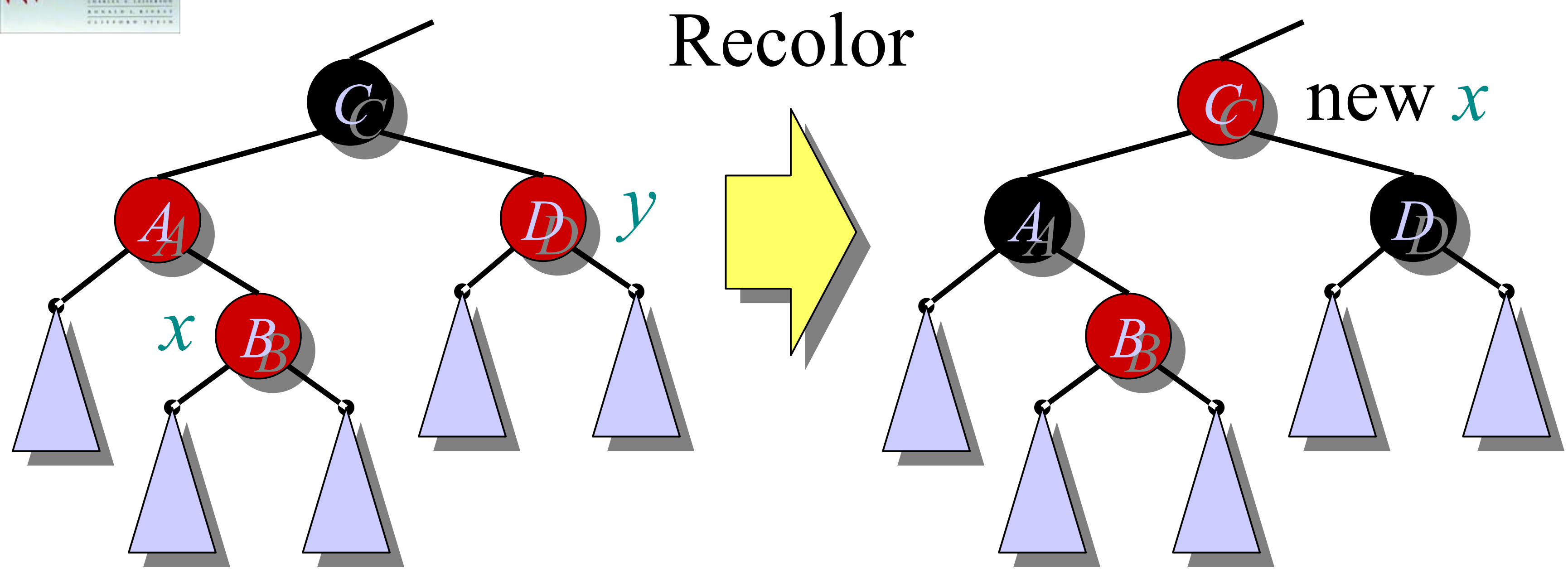
Graphical notation

Let  denote a subtree with a black root.

All ’s have the same black-height.

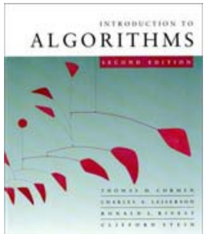


Case 1

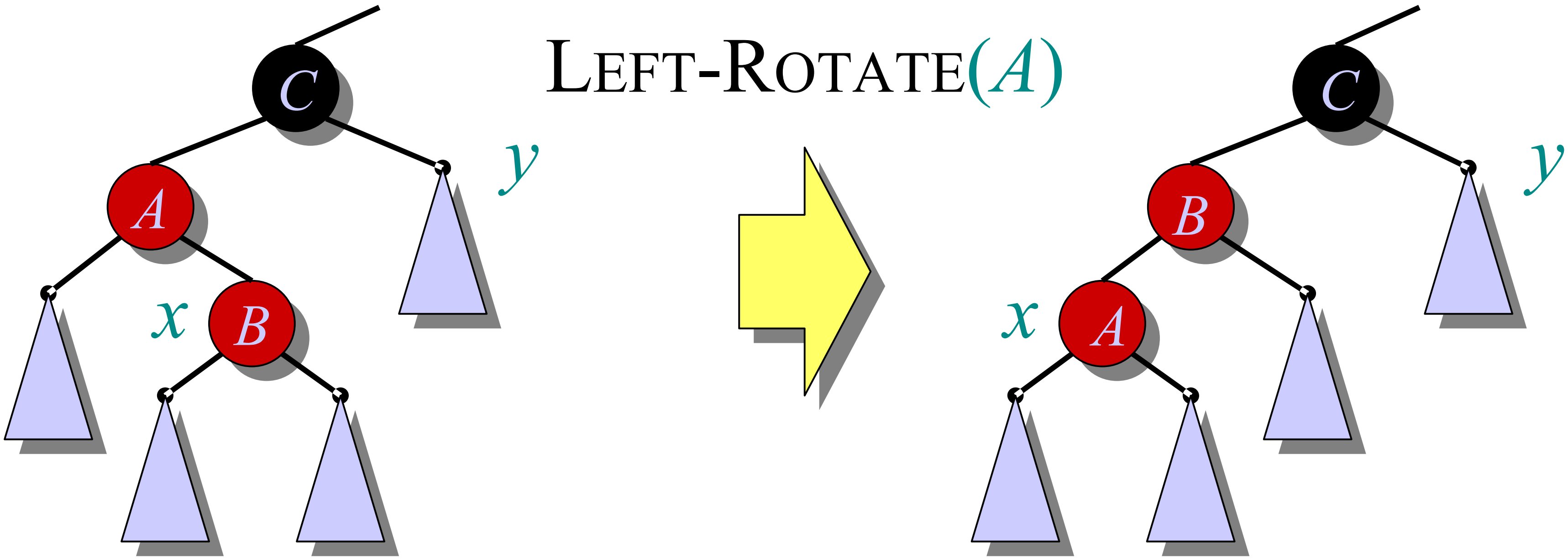


(Or, children of *A* are swapped.)

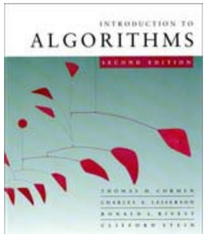
Push *C*'s black onto *A* and *D*, and recurse, since *C*'s parent may be red.



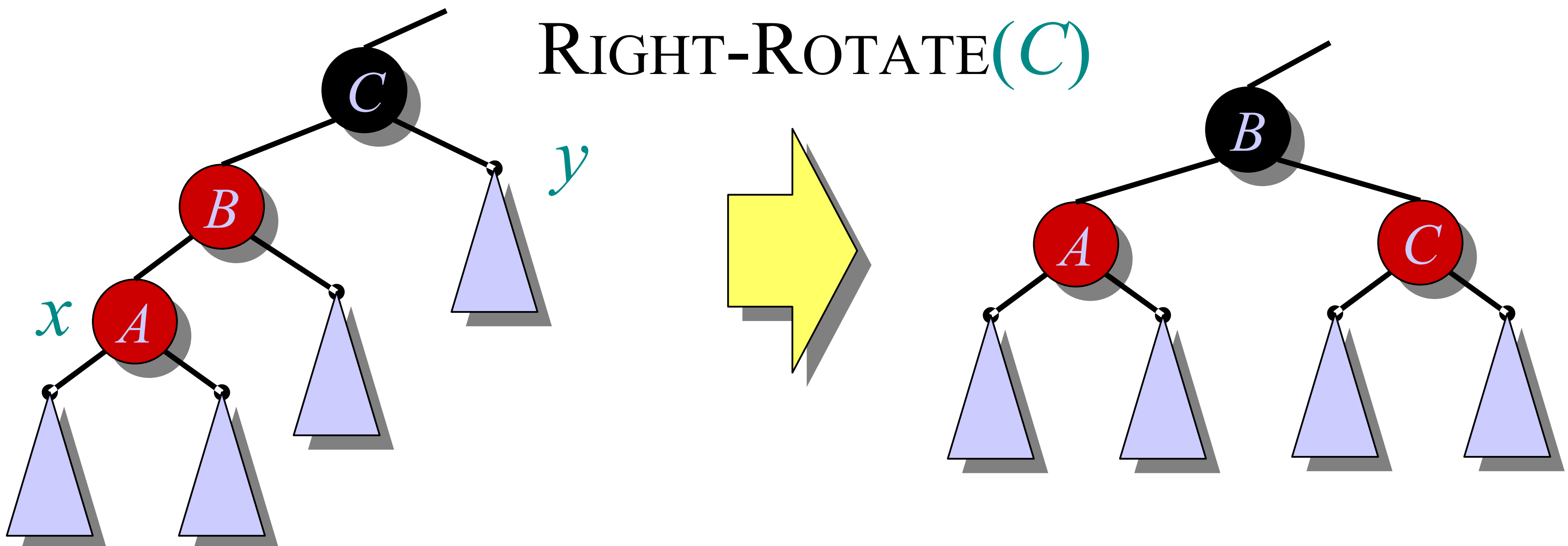
Case 2



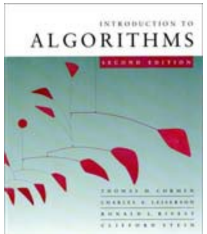
Transform to Case 3.



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 and number of rotations as RB-INSERT (see textbook).

Recap

- Binary Search Trees
  - Insert, Search, Delete
  - Expected Height
- Red Black Trees