Dictionary: Binary Search Tree

Fatemeh Panahi
Department of Computer Science
University of Toronto
CSC263-Fall 2017
Lecture 3

Announcements

- Midterm:
  - LEC0101, LEC2003: Fri, Oct 27, 10:00-11:00
  - LEC0201, LEC2000, LEC2201: Fri, Oct 27, 13:00-14:00
  - Location will be announced.

- Tutorial on Friday:

Textbook exercise: 12.2-6, 12.2-7, 12.2-8, 12-2
and problem 6-3 from Chapter 6.

Dictionary

Today

- Dictionary
- Binary Search Tree (BST)
  - Tree Traversals (Inorder, Preorder and Postorder)
  - Successor and Predecessor
  - Operations
    - Search
    - Insert
    - Delete
  - Height of a BST

Reading Assignments

Part III (introduction), Sections 12.1, 12.2, 12.3

Abstract Data Type: Dictionary

Objects:

Set of elements where each element x has field x.key.

Keys are some totally ordered value, the keys are distinct.

Operations:

- **SEARCH(S,k):** return x in S s.t. x.key = k, or NIL if no such x
- **INSERT(S,x):** insert x in S; if some y in S has y.key = x.key, replace y by x.
- **DELETE(S,x):** remove node x from S

Implement a Dictionary using simple data structures

Unsorted (doubly) linked list

Search(S, k)

- $O(n)$ worst case
- go through the list to find the key

Insert(S, x)

- $O(n)$ worst case
- need to check if $x.key$ is already in the list

Delete(S, x)

- $O(1)$ worst case
- Just delete, $O(1)$ in a doubly linked list

Sorted Array

Search(S, k)

- $O(\log n)$ worst case
- Binary search!

Insert(S, x)

- $O(n)$ worst case
- Insert at front, everything has to shift to back

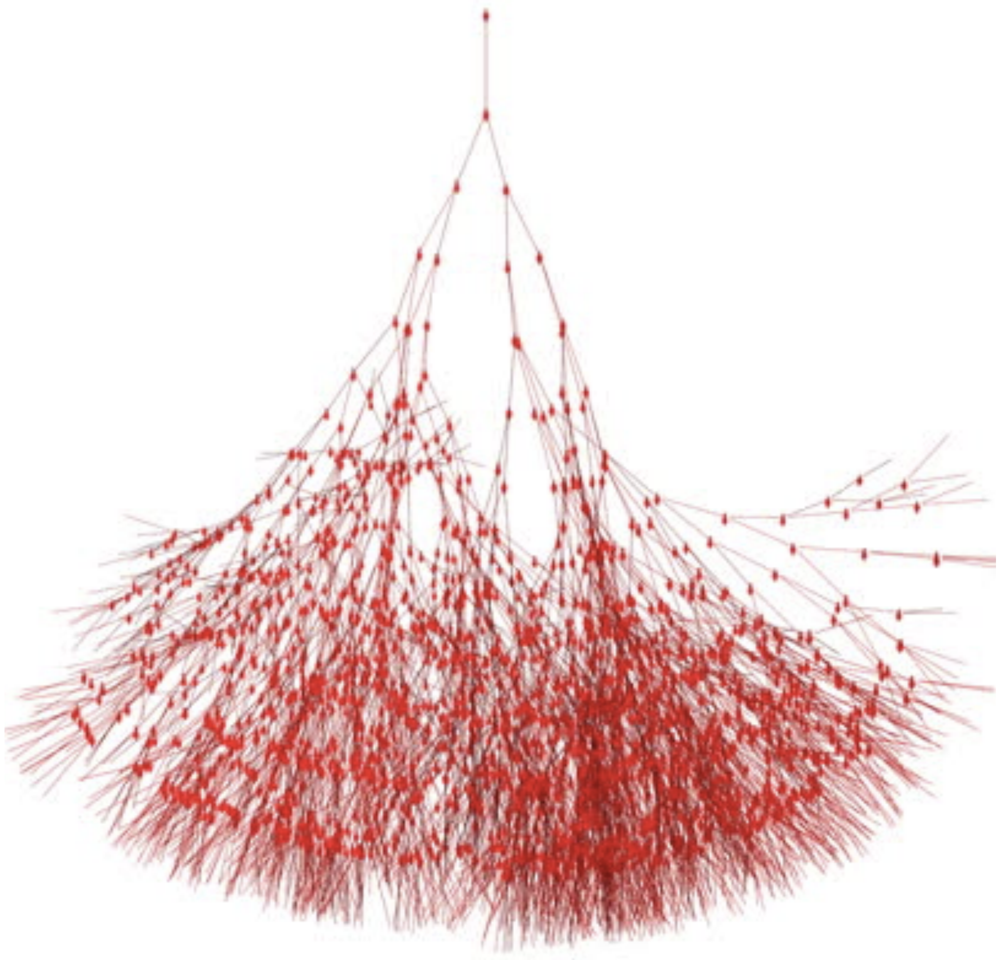
Delete(S, x)

- $O(1)$ worst case
- Delete at front, everything has to shift to front

Better data structures?

	Unsorted list	Sorted Array	BST	Balanced BST
Search	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$
Insert	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
Delete	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$

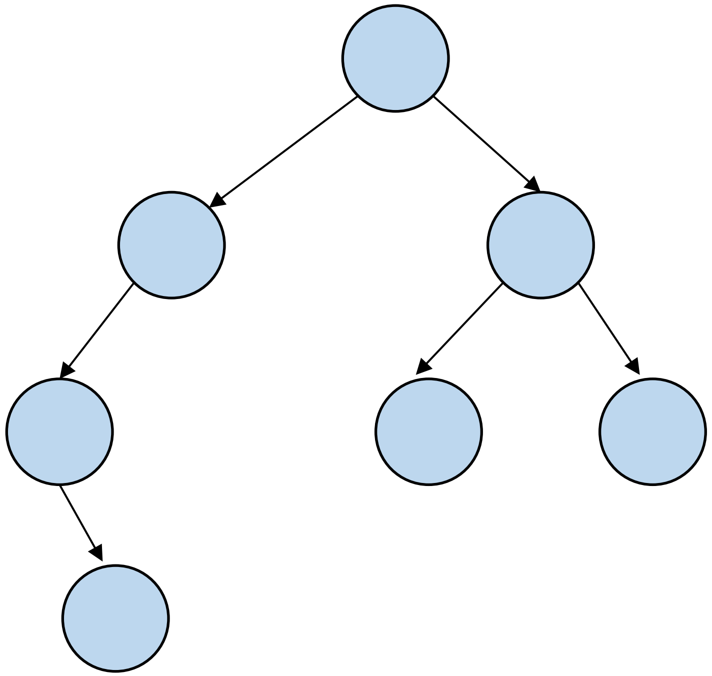
Binary Search Tree



Binary Tree

Binary tree is a linked structure with

- root
- left subtree *(maybe empty)*
- right subtree *(maybe empty)*
- Parent *(maybe empty)*



Representation:

- $x.key$: the key
- $x.left$: the left child (node)
- $x.right$: the right child (node)
- $x.p$: the parent (node)

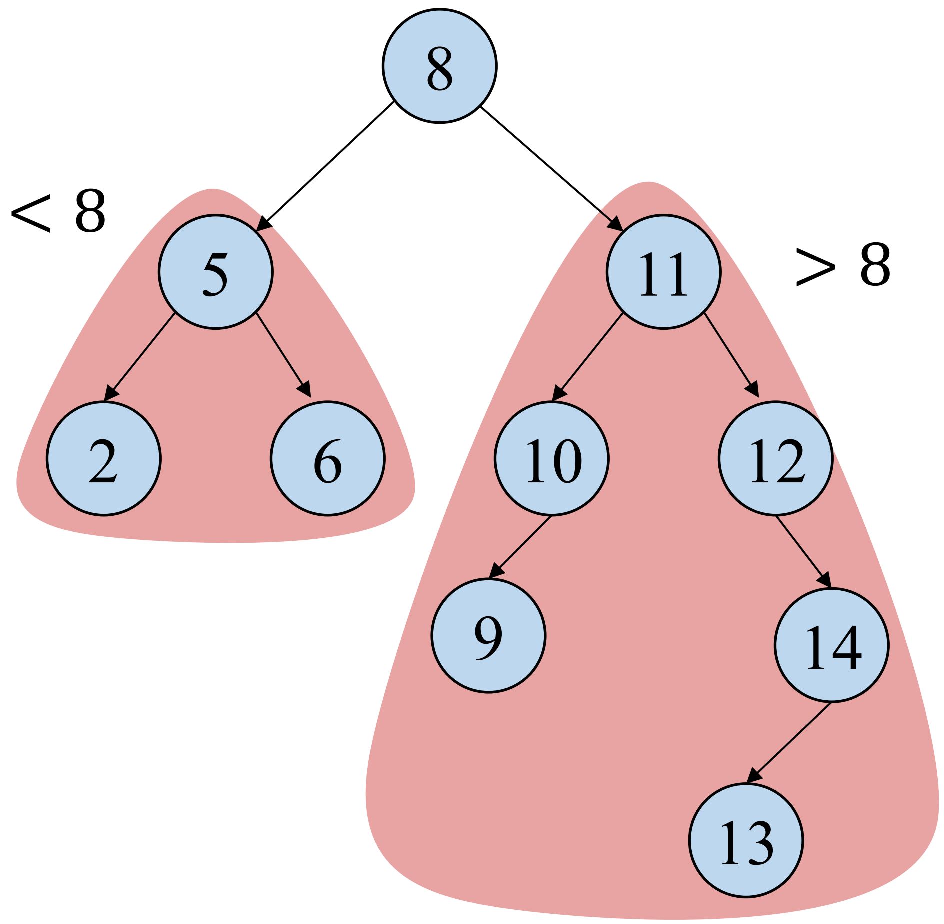
Binary Search tree property

All keys in left subtree smaller than root's key

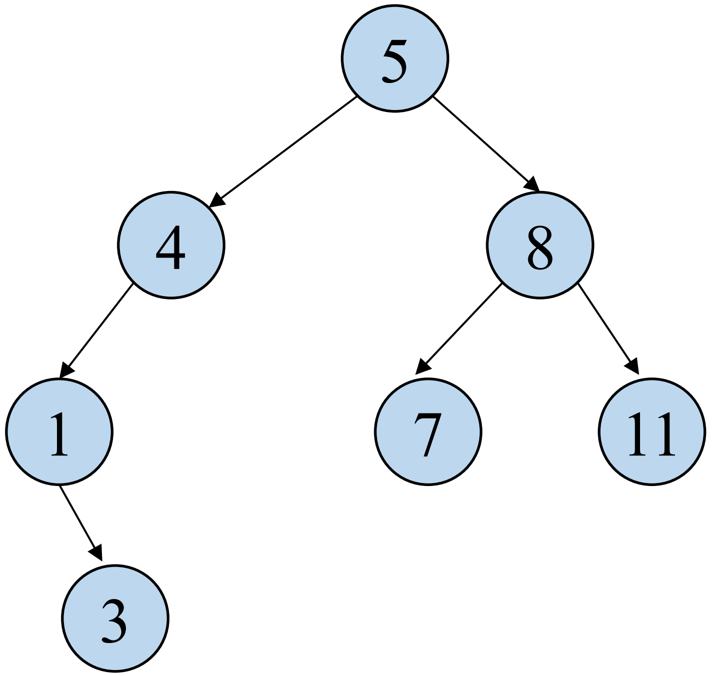
All keys in right subtree larger than root's key

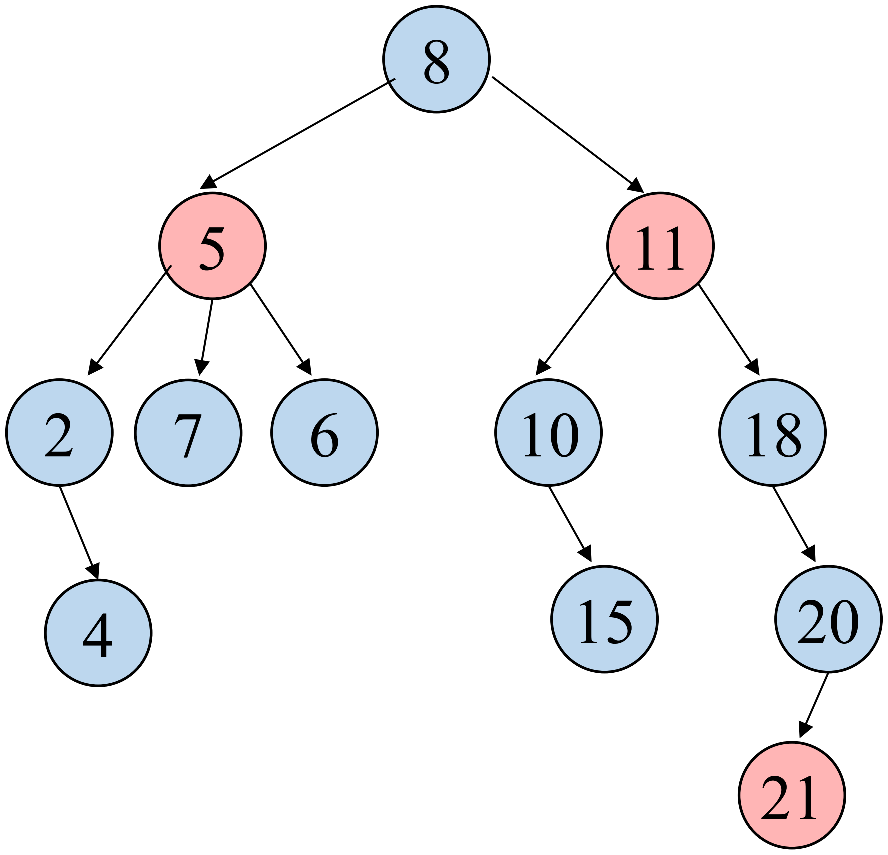
Therefore:

- easy to find any given key
- Insert/delete by changing links



Binary Search Tree - Example



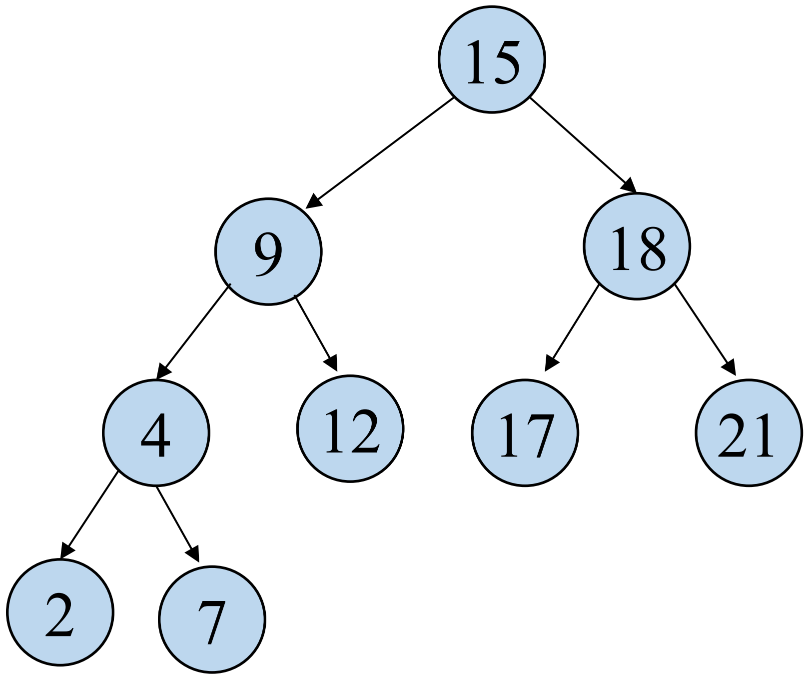


BINARY SEARCH TREE

NOT A
BINARY SEARCH TREE

Complete Binary Search Tree

Links are completely filled, except possibly bottom level, which is filled left-to-right.



Tree Traversal

Tree Traversal refers to the process of visiting each node in a tree, exactly once.

- Inorder: visits the root of a subtree between visiting the nodes in its left subtree and visiting those in its right subtree.

  The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by inorder tree walk.

- Preorder: visits the root before the elements in either subtree,

- Postorder: visits the root after the elements in its subtrees.

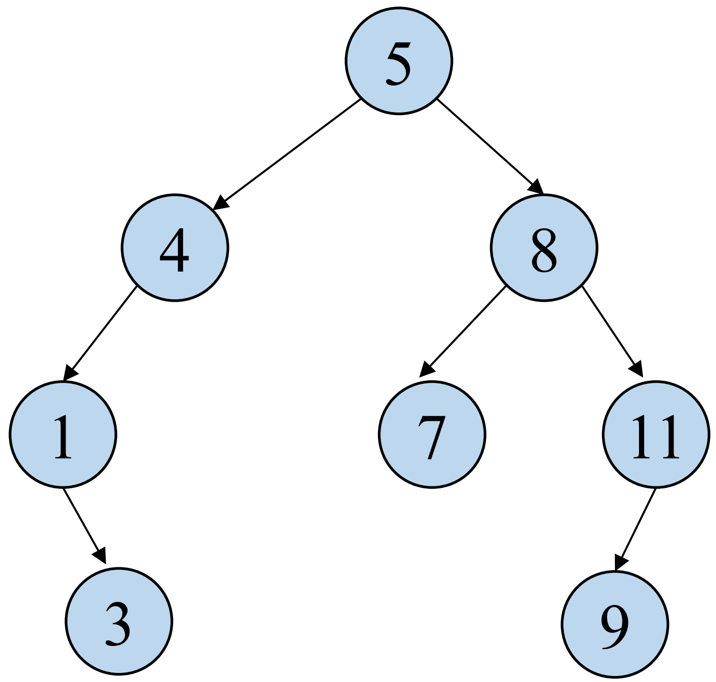
Inorder Walk

visit left subtree

visit node

visit right subtree

INORDER-TREE-WALK(x)
    if $x \neq NIL$
        INORDER-TREE-WALK ($x.left$)
        print $x.key$
        INORDER-TREE-WALK($x.right$)



In order listing:

$1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 11$

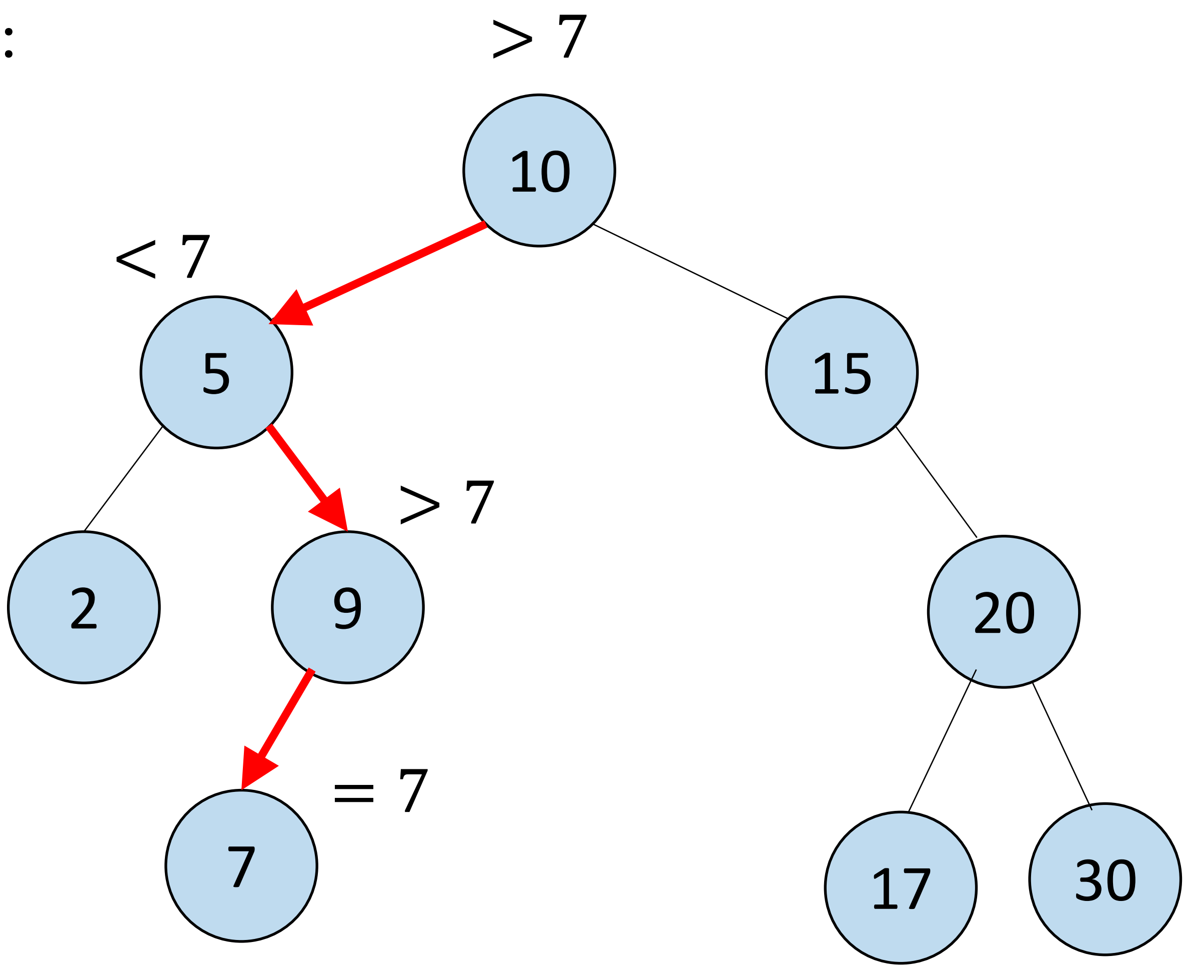
What does this guarantee with a BST?

Operations on a BST

- read-only operations
  - TreeSearch(root, k)
  - TreeMinimum(x) / TreeMaximum(x)
  - Successor(x) / Predecessor(x)
- modifying operations
  - TreeInsert(root, x)
  - TreeDelete(root, x)

Search in a BST - Example

TreeSearch(root, 7):



Search in a BST - Recursive

```
TreeSearch(root, k):
    if root = NIL or k = root.key:
        return root
    if k < root.key:
        return TreeSearch(root.left, k)
    else:
        return TreeSearch(root.right, k)
```

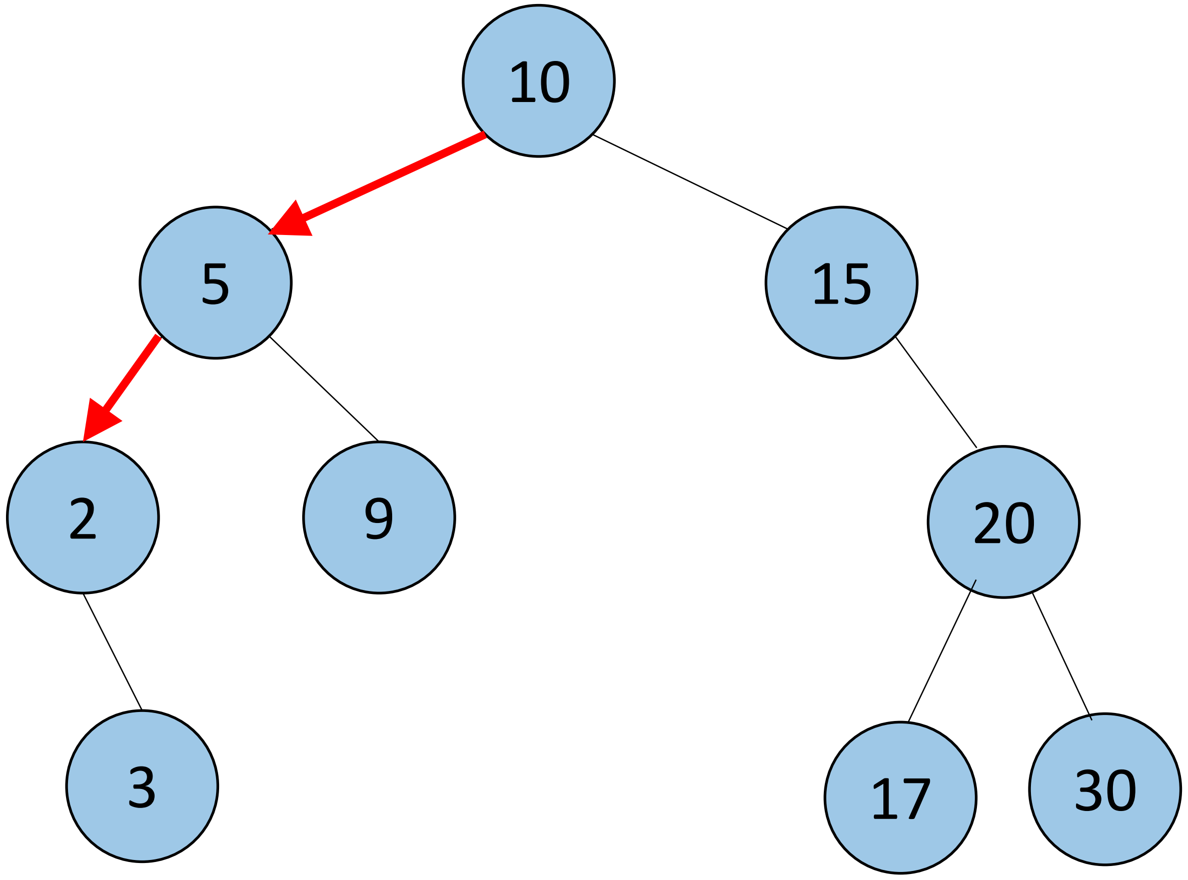
Runtime:

- Worts case

  $\Theta(h)$

TreeMinimum(x): Example

TreeMinimum(root):



TreeMinimum(x): pseudo-code

TreeMinimum(x):

  while $x.left \neq NIL$:

    $x \leftarrow x.left$

  return x

Runtime:

- Worts case

  $\Theta(h)$

TreeMaximum(x) is exactly the same, except that it goes to the right instead of to the left.

Successor(x) and Predecessor

- Successor(x):

  Find the node with the smallest key larger than x. (The node which is the successor of x in the sorted list obtained by inorder traversal.)

- Predecessor(x)

  Find the node with the largest key smaller than x. (The node which is the predecessor of x in the sorted list obtained by inorder traversal.)

Successor(x) - Example

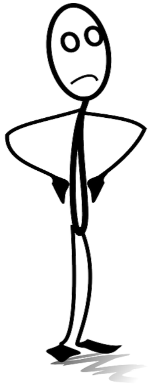
Successor(2)?

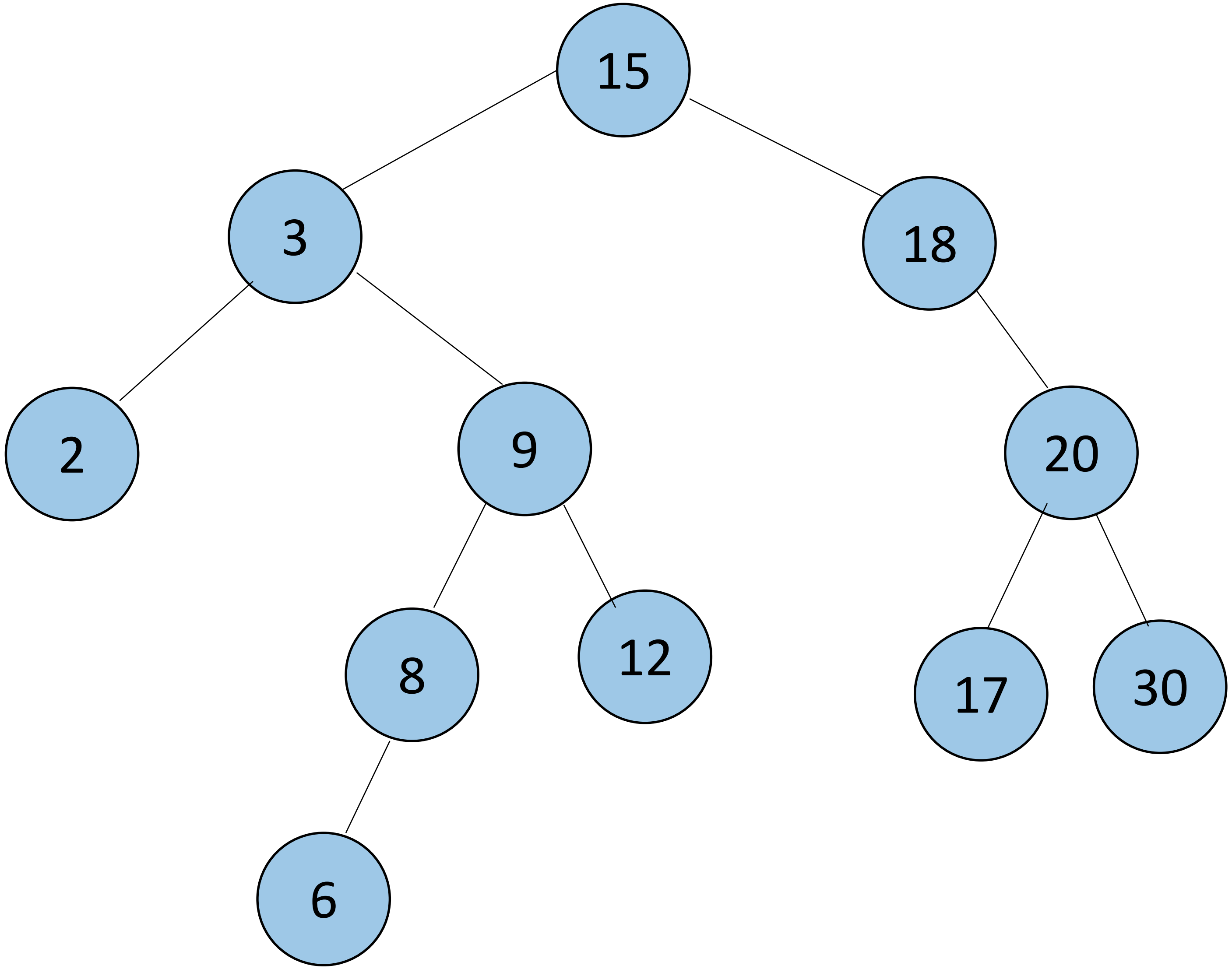
Successor(3)?

Successor(12)?

Successor(18)?

It is all over the place!!

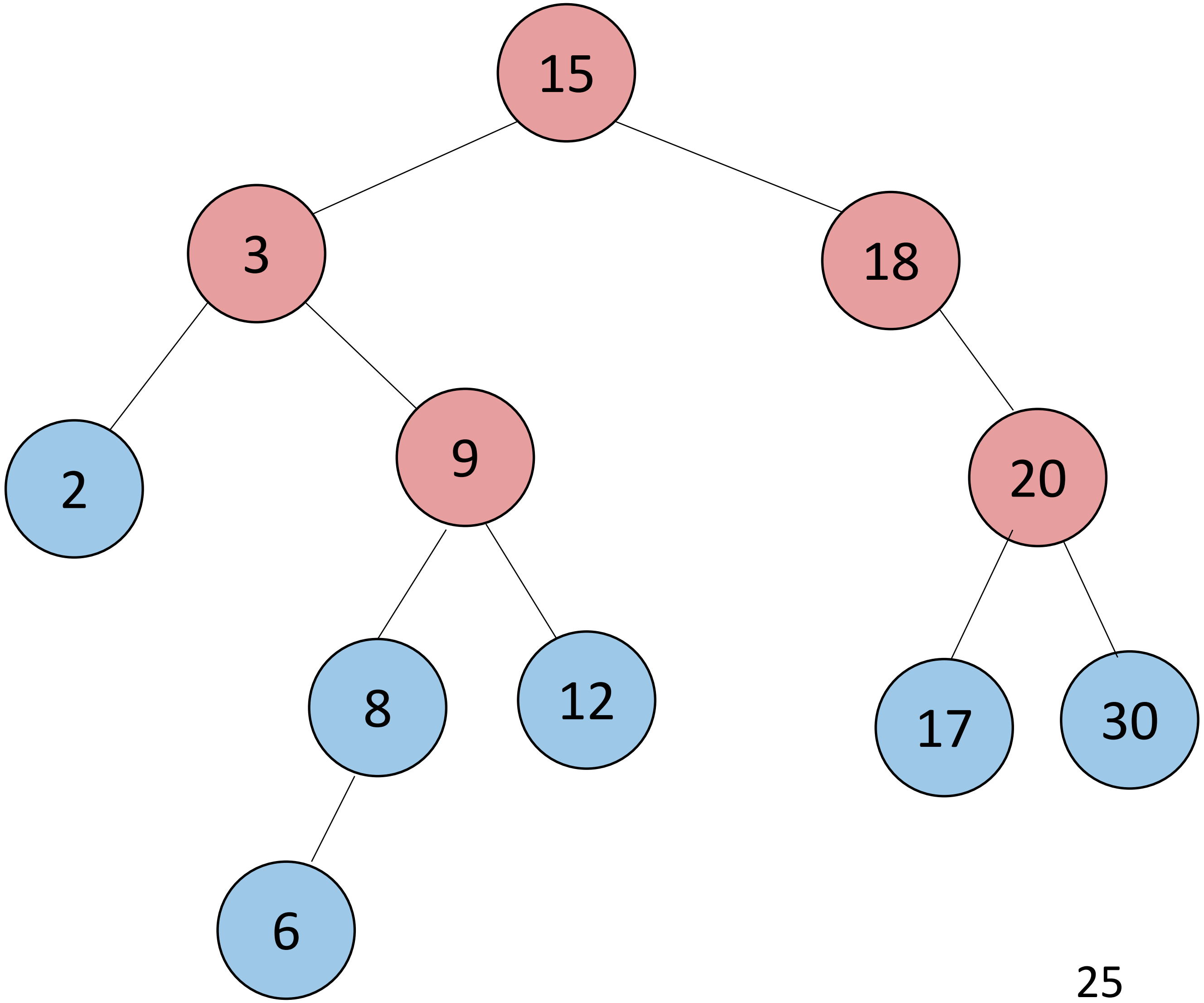




Successor(x) - Cases

Successor(x) Organize into two cases

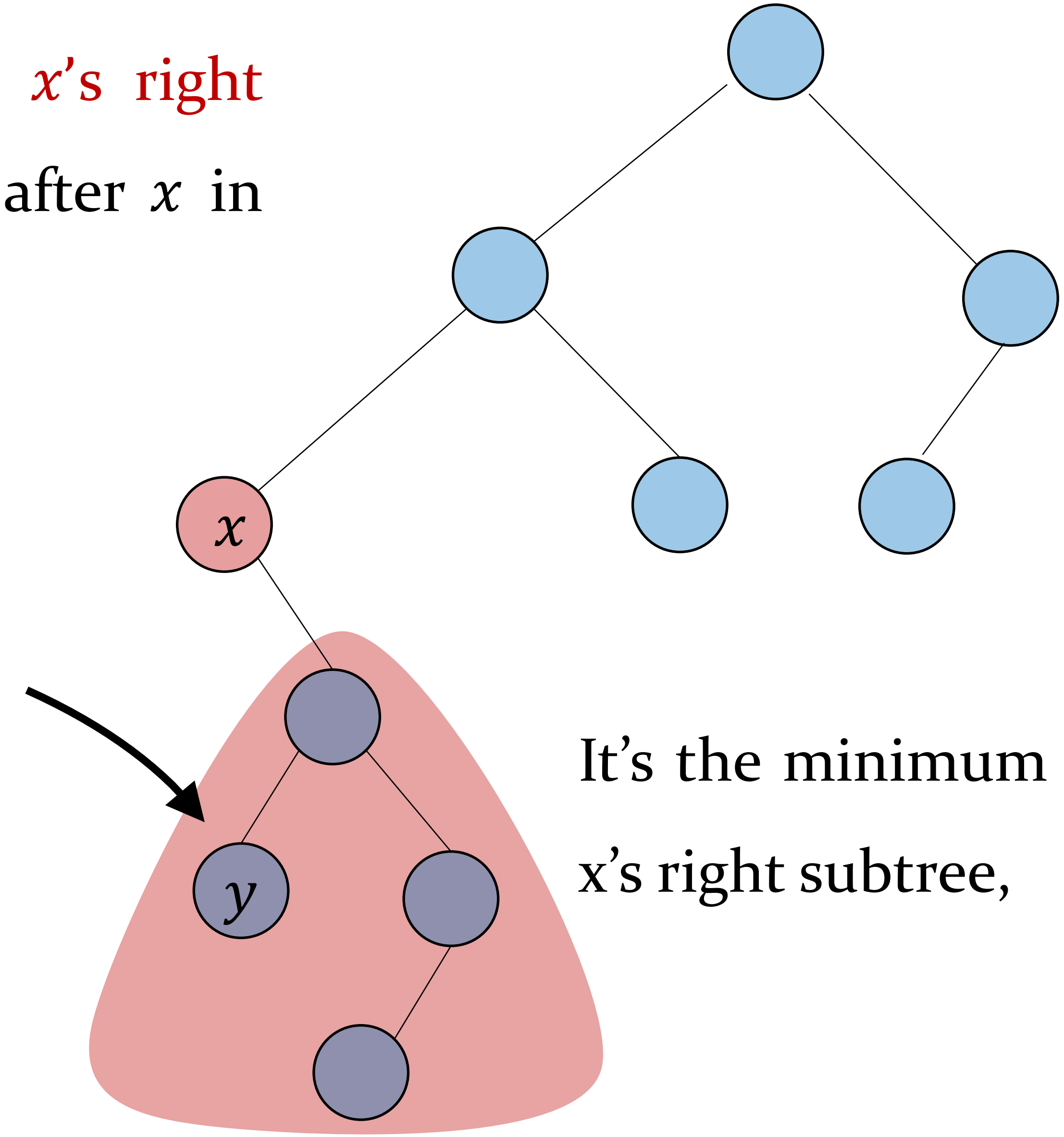
- x has a right child
- x does not have a right child



Case 1: x has a right child

Successor(x) must be in x's right subtree (the nodes right after x in the inorder traversal)

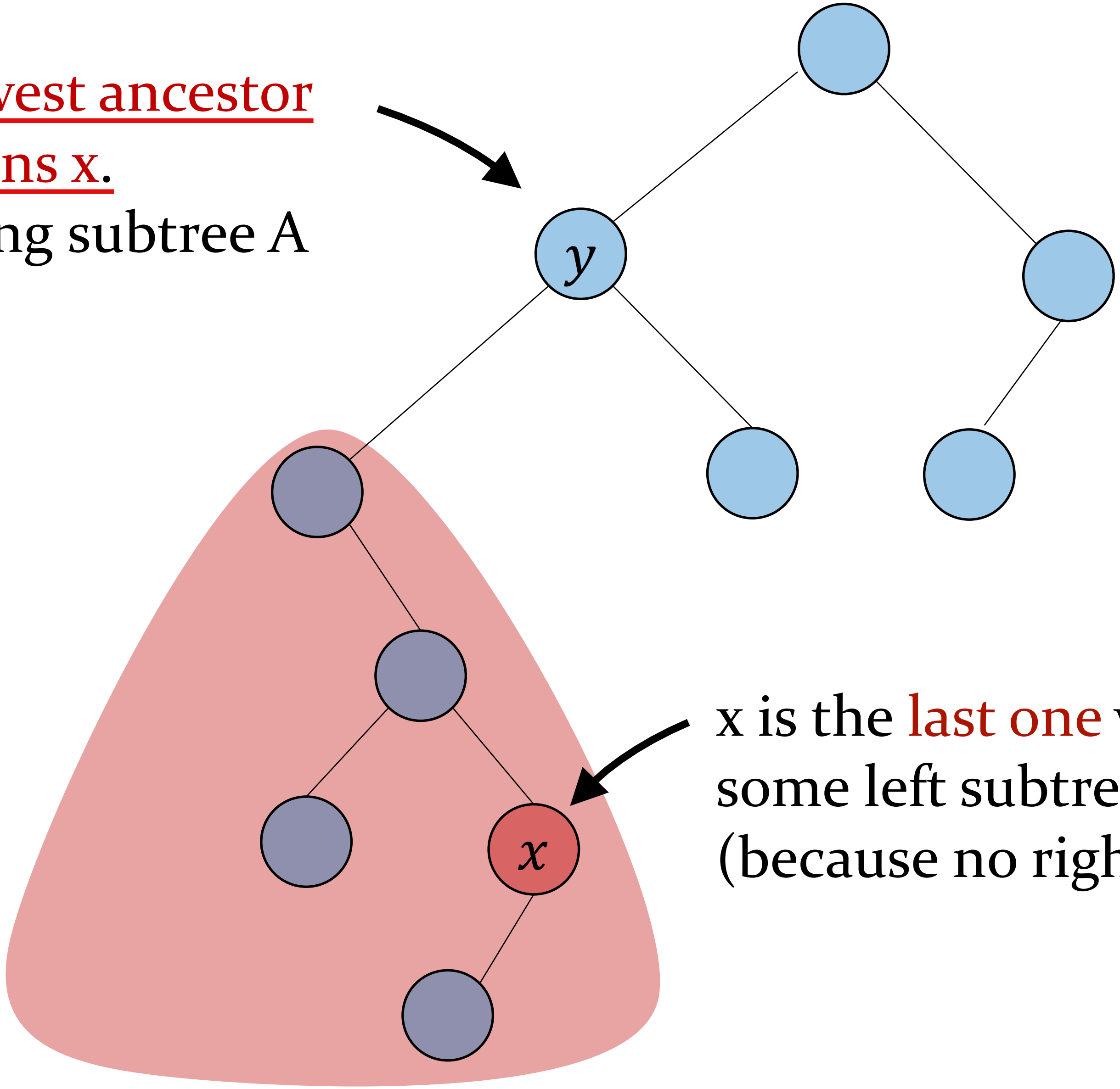
TreeMinimum($x.right$)



It's the minimum of x's right subtree,

Case 2: x does not have a right child

The successor y of x is the lowest ancestor of x whose left subtree contains x.
(y is visited right after finishing subtree A in inorder traversal)

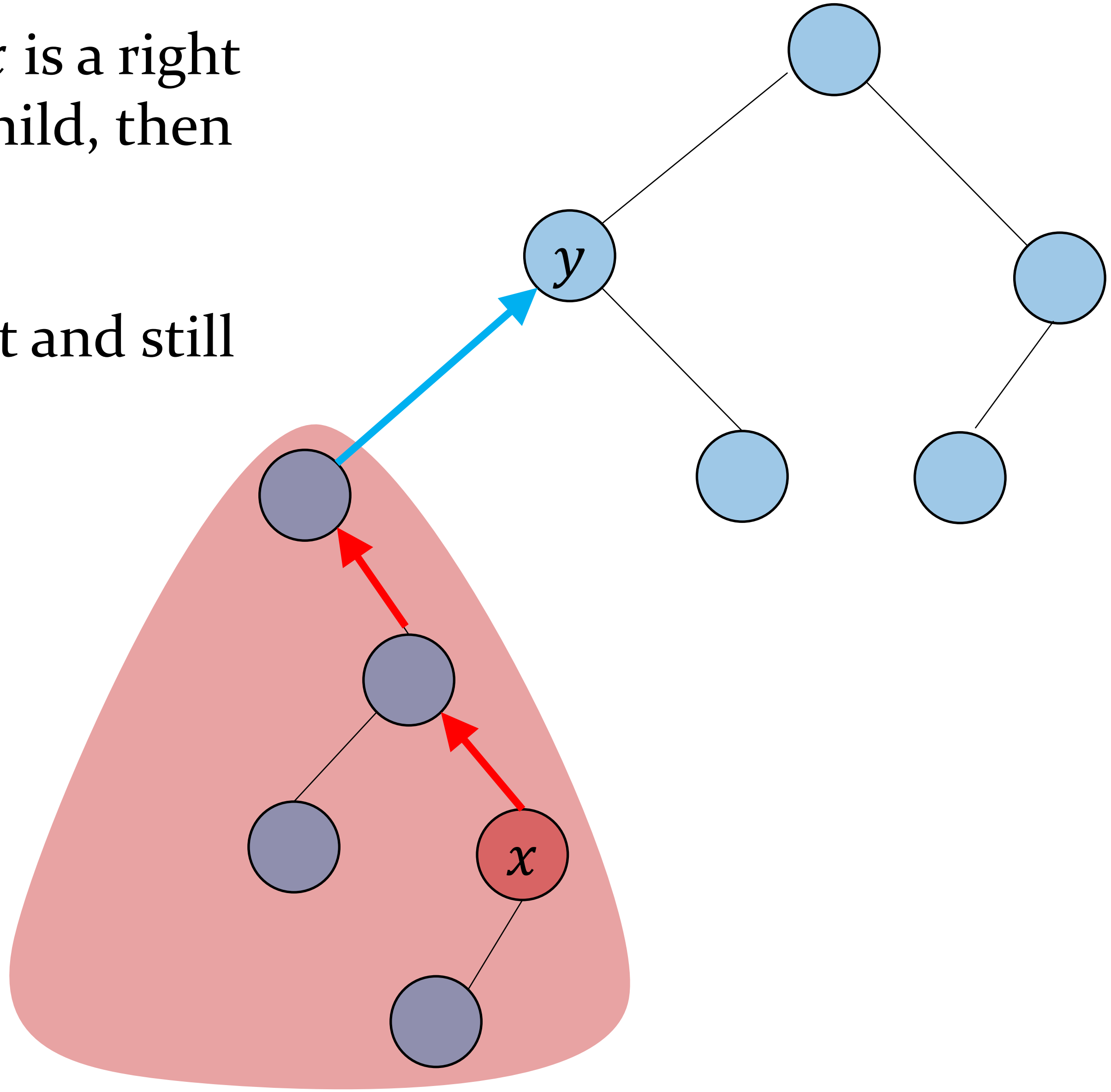


x is the last one visited in some left subtree A (because no right child)

Case 2: x does not have a right child

- keep going up to $x.p$ while x is a right child, stop when x is a left child, then return $x.p$.

- if already gone up to the root and still not finding it, return NIL.



Successor(x) - Sudocode

```
Successor(x):
    if x.right ≠ NIL:
        return TreeMinimum(x.right)
    y ← x.p
    while y ≠ NIL and x = y.right      #x is right child
        x = y
        y = y.p                    # keep going up
    return y
```

Worstcase Runtime:

$O(h)$

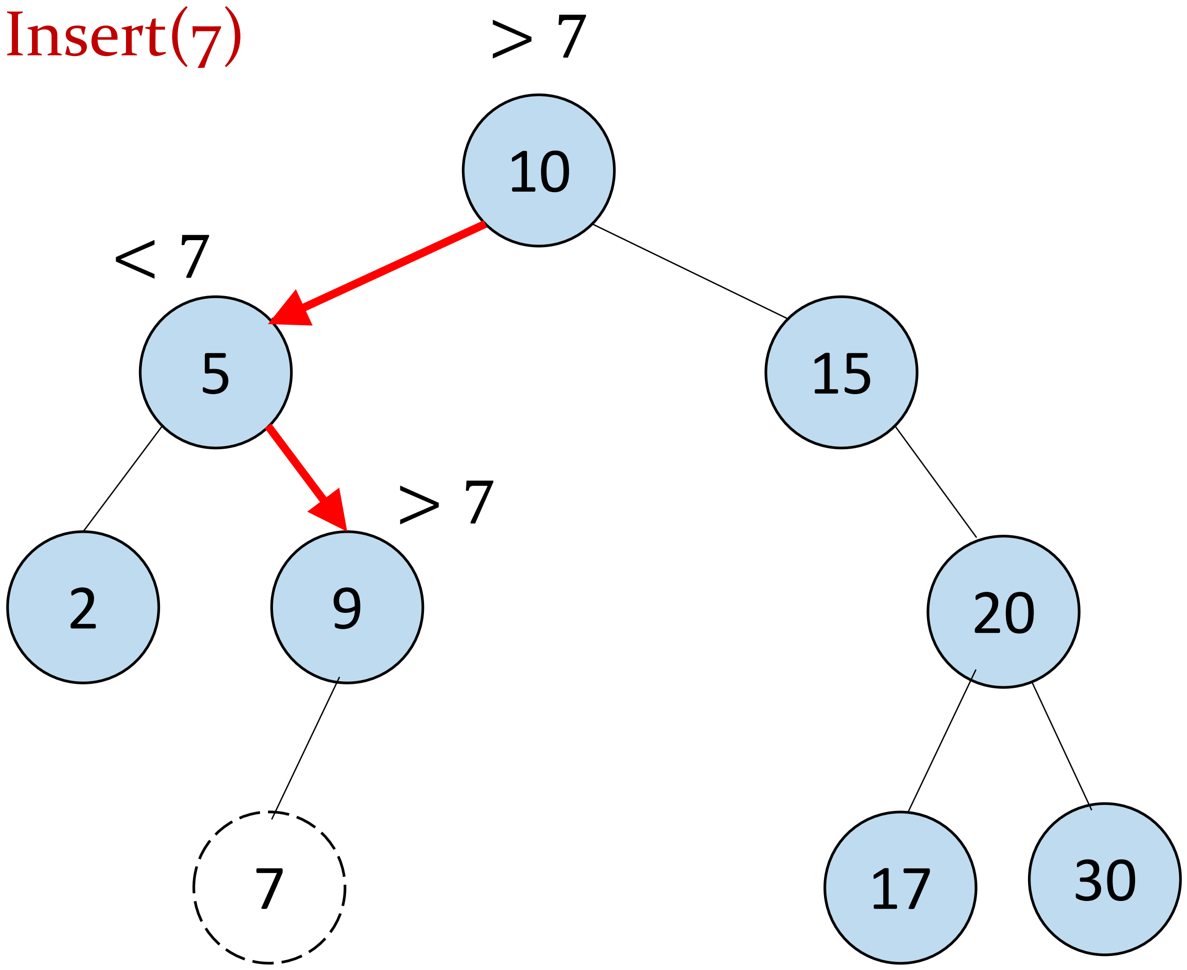
TreeInsert(root, x)

Insert node x into the BST

if $\exists y, \quad y.key = x.key$, replace y with x

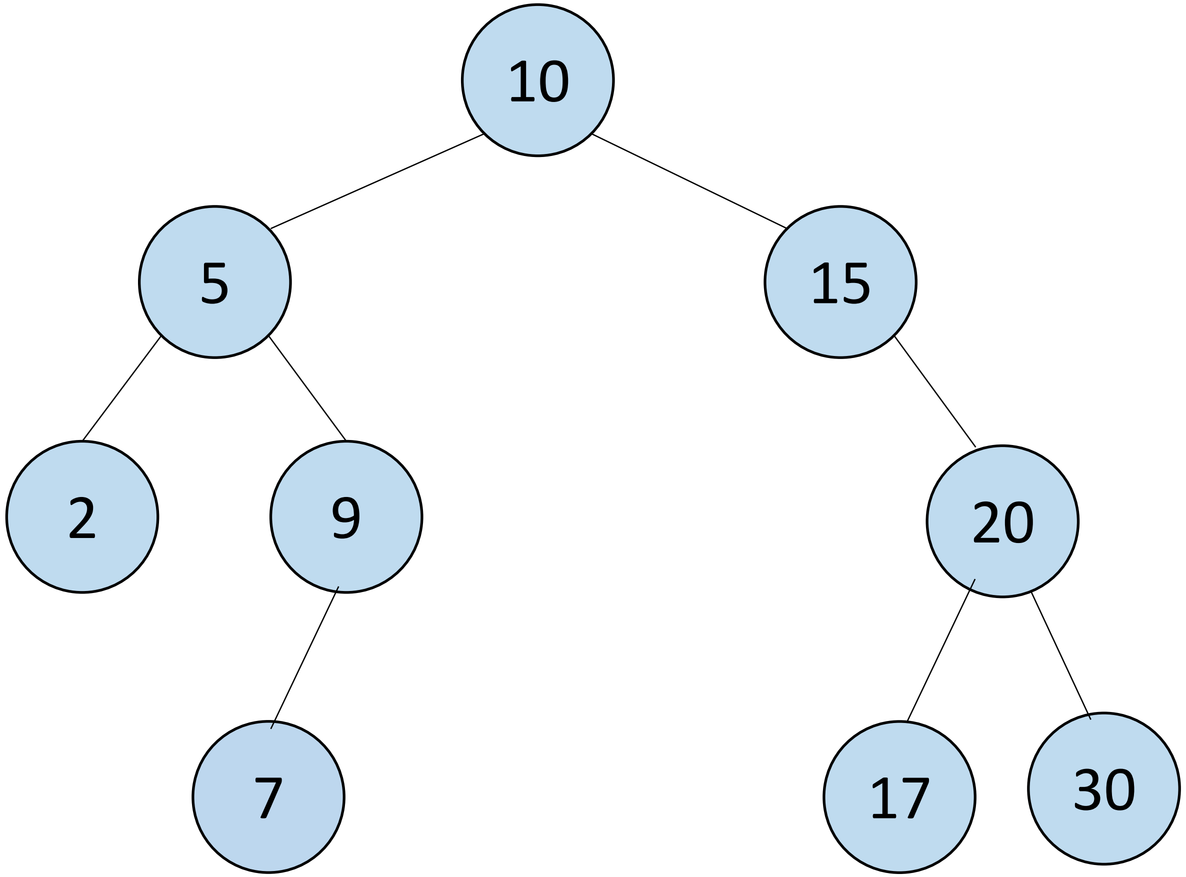
TreeInsert ($root, x$) - example

- Proceed down tree as in Find
- If the key is found replace with the new element
- If new key not found, then insert a new node at last spot traversed



TreeInsert $(root, x)$ – Practice example 1

Insert(18)



TreeInsert $(root, x)$ – Psudocode

```
TreeInsert(root, x):                 # insert and return the new root
    if root = NIL:
        root ← x
    elif x.key < root.key
        root.left ← TreeInsert(root.left, x)
    elif x.key > root.key:
        root.right ← TreeInsert(root.right, x)
    else                             # x.key = root.key:
        replace root with x          # update x.left, x.right
    return root
```

Worst case running time:
$O(h)$

Insert sequence - Practice

Question 1: Is it possible that a different BST results when we try to insert the same sequence in an empty binary tree in a different order?

Question 2: Insert the following sequence number in an empty binary search tree. 14, 8, 16, 15, 9, 3, 17

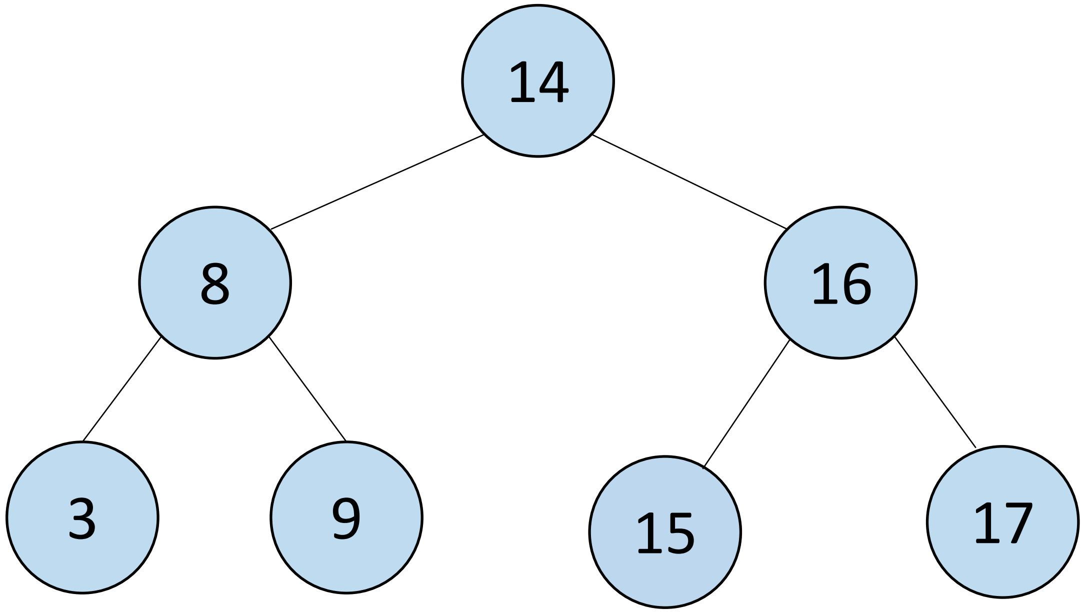
What is the hight of the tree?

Question 3: Insert the following sequence number in an empty binary search tree. 3, 8, 9, 14, 17, 16, 15

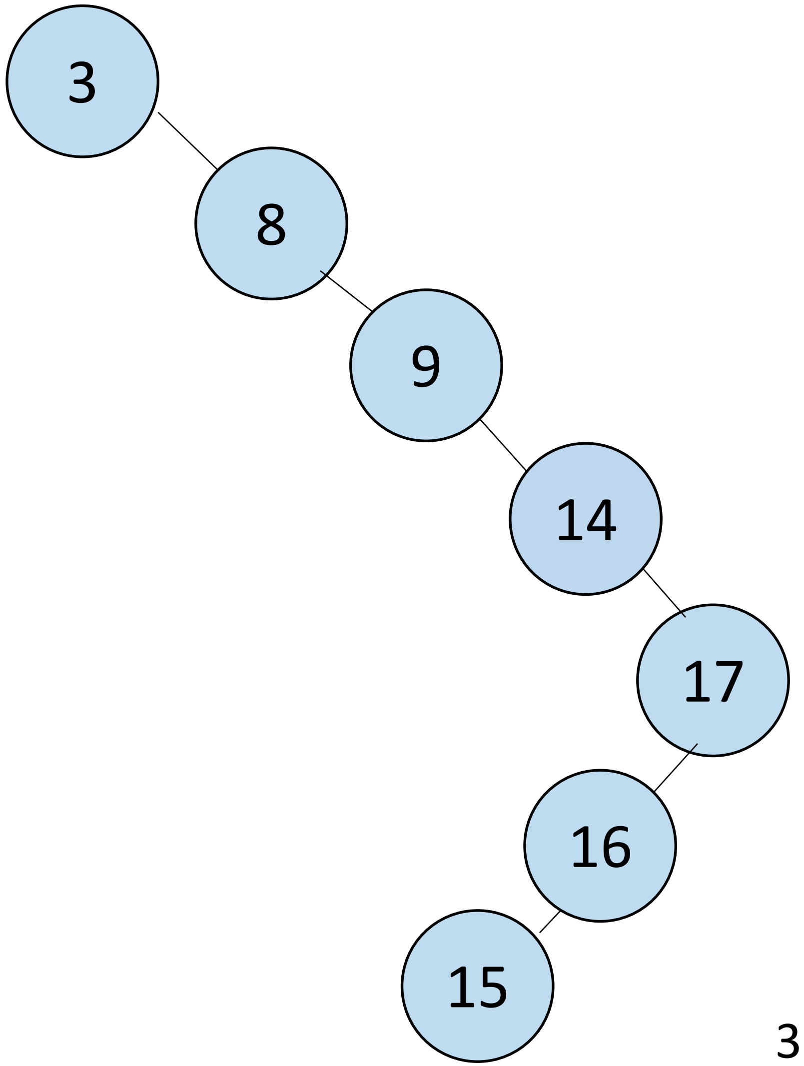
What is the hight of the tree?

Insert sequence

Insert sequence: 14, 8, 16, 15, 9, 3, 17



Insert sequence: 3, 8, 9, 14, 17, 16, 15



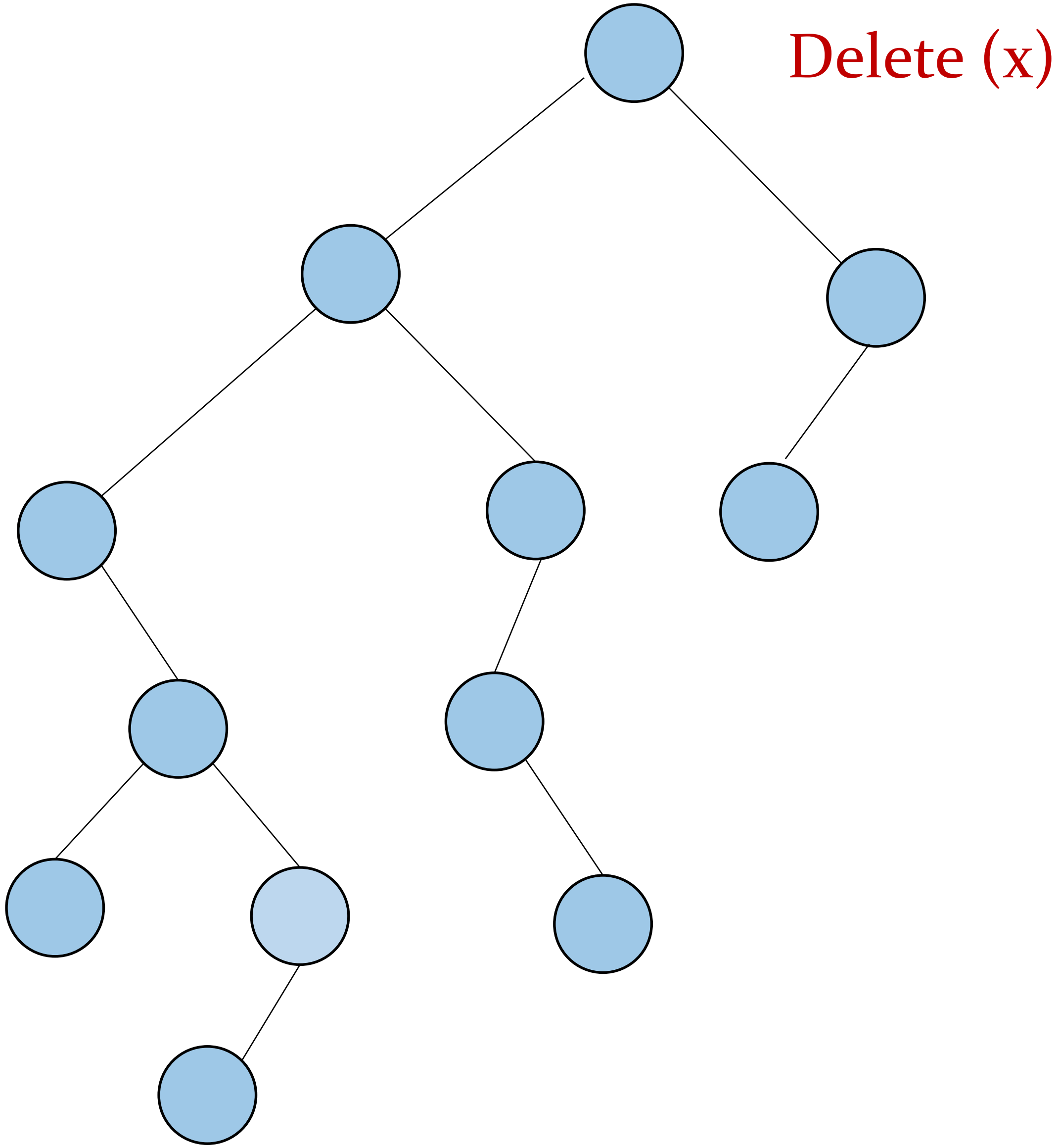
TreeDelete(root, x)

Delete node x from BST rooted at root while maintaining BST property, return the new root of the modified tree

TreeDelete ($root, x$) - cases

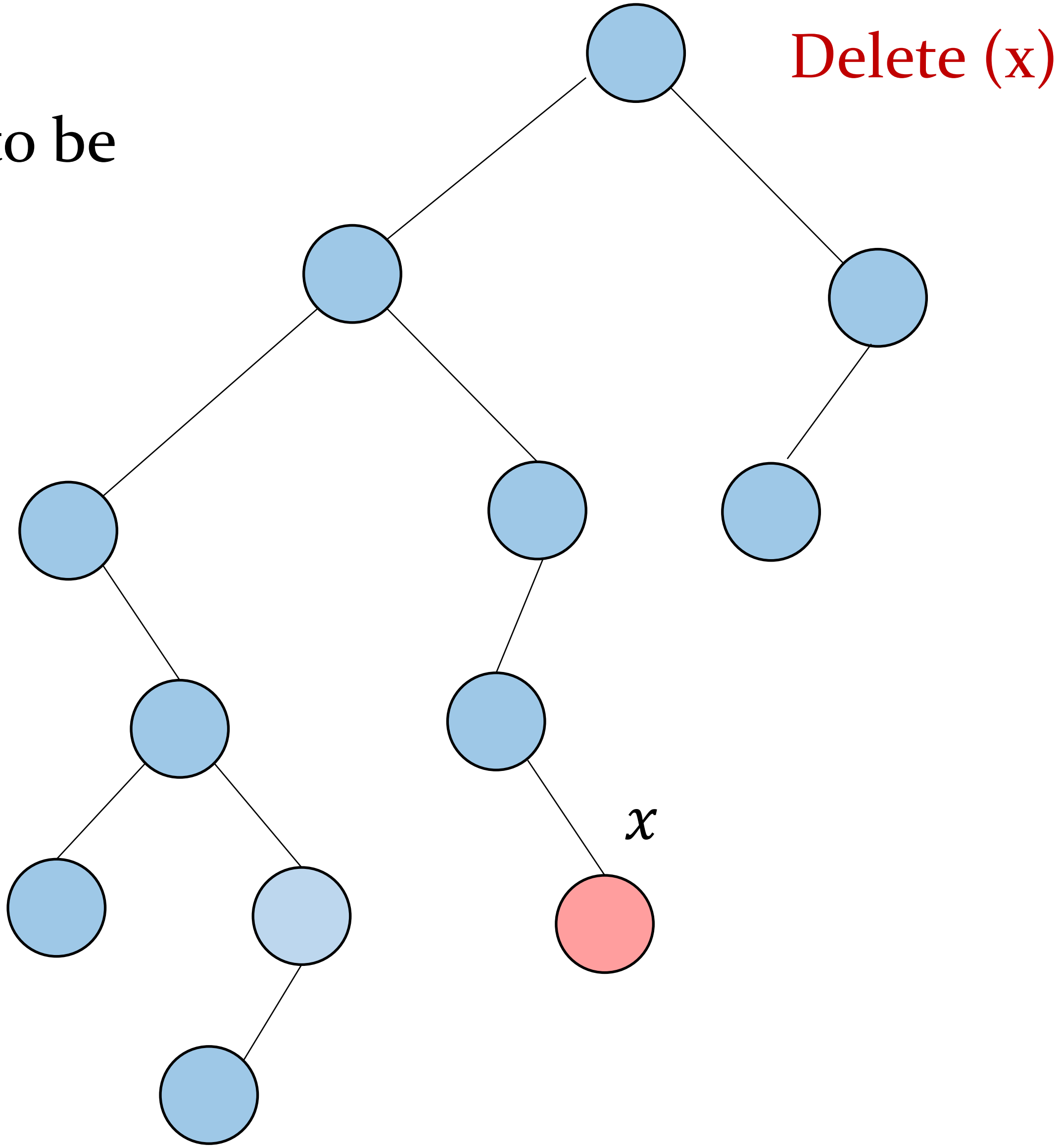
Tree Cases:

- Case 1: x has no child
- Case 2: x has one child
- Case 3: x has two children



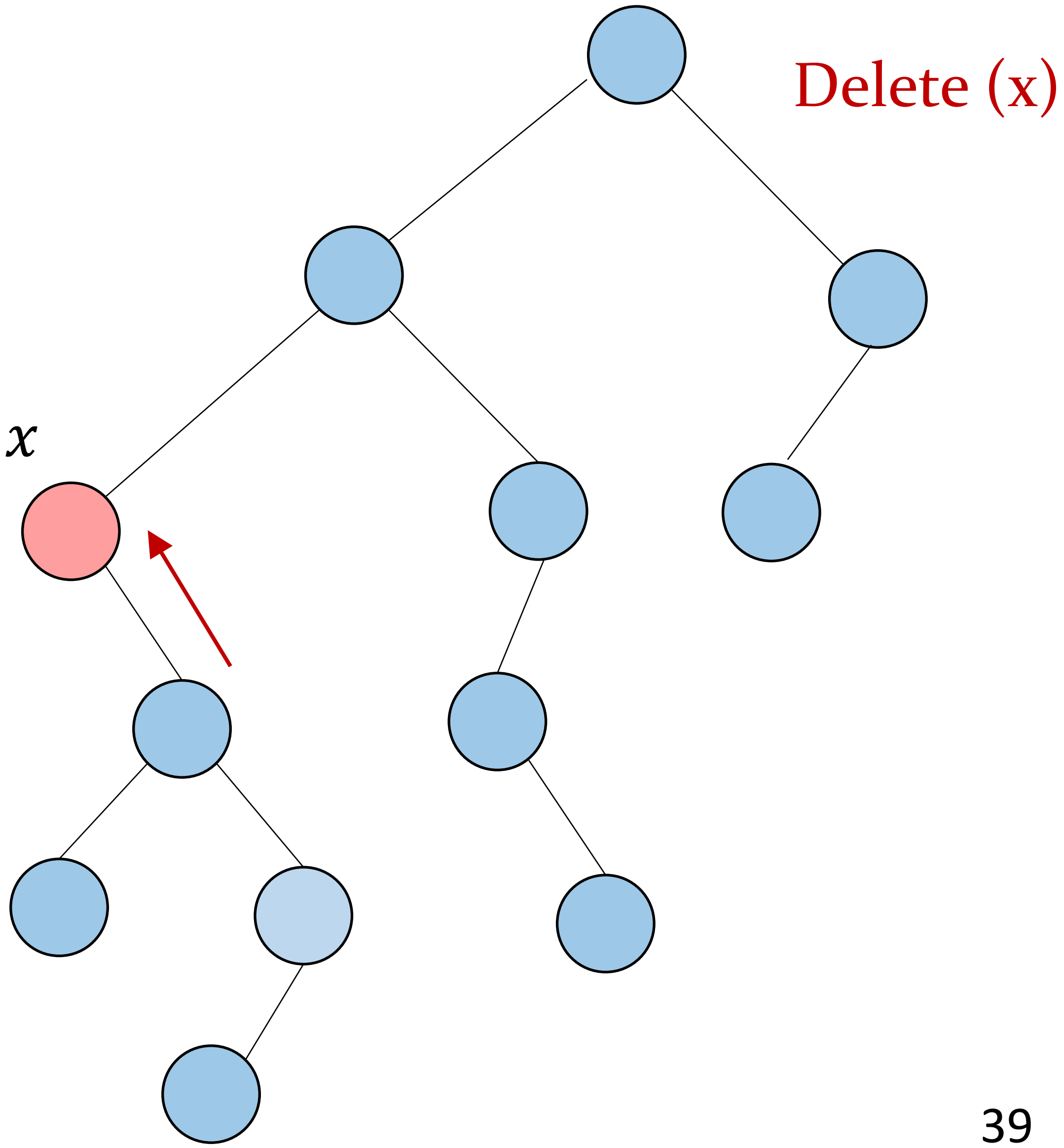
Case 1

- **x has no child**
  Just delete it, nothing else need to be changed.



Case 2

- Case 2: x has one child
  - First delete that node, which makes an open spot.
  - Then promote x’s only child to the spot, together with the only child’s subtree.
  - The procedure is called Transplant in the textbook.



Case 1, 2 - psudocode

$TRANSPLANT(T, x, y)$

if $x.p \ == \ NIL$

$T.Root \ = y$

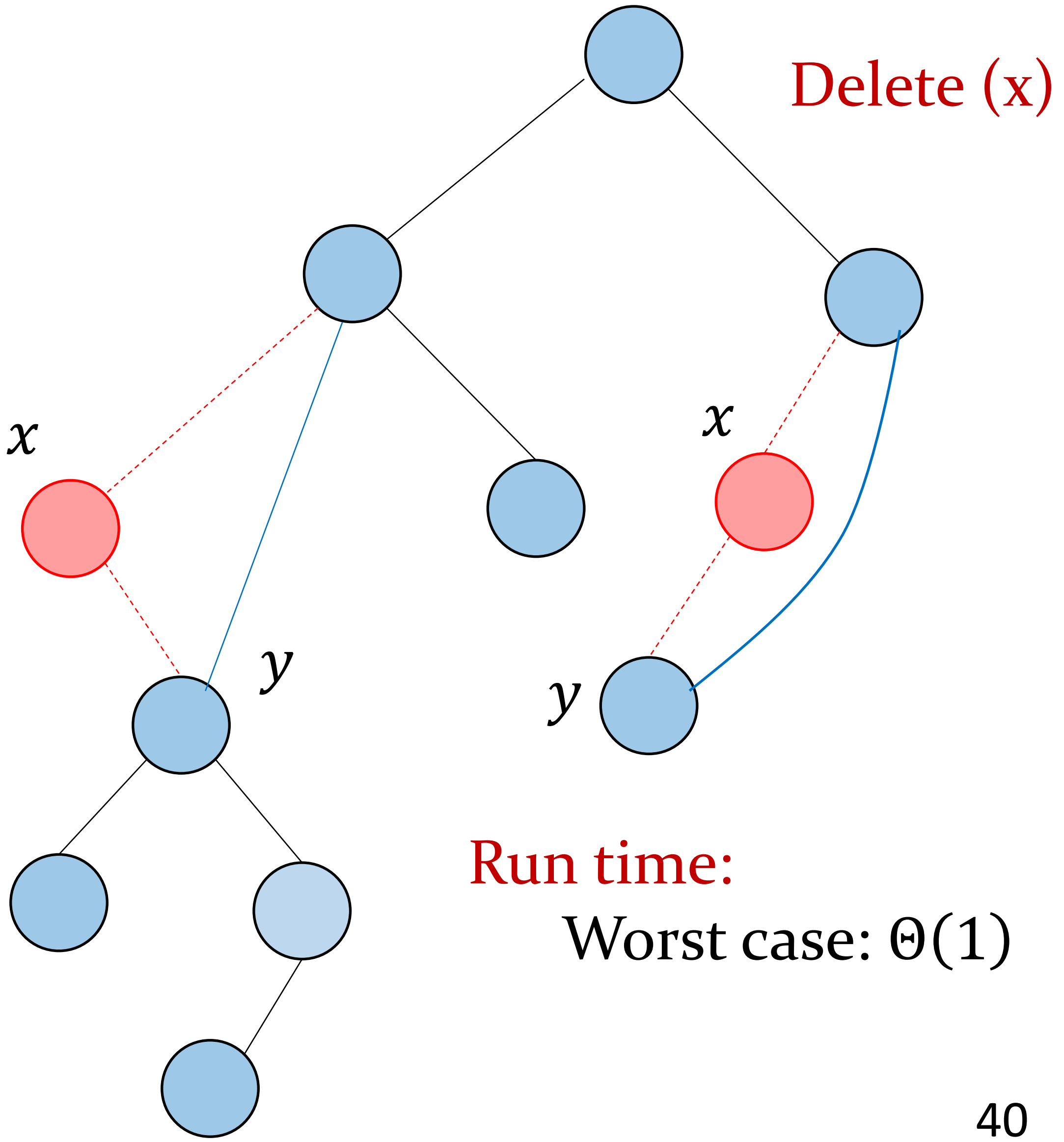
elseif $x \ == x.p.left$

$x.p.left \ = y$

else $x.p.right \ = y$

if $y \neq \ NIL$

$y.p \ = x.p$

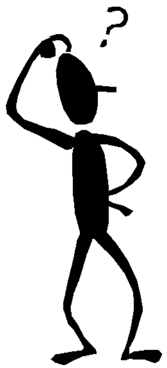


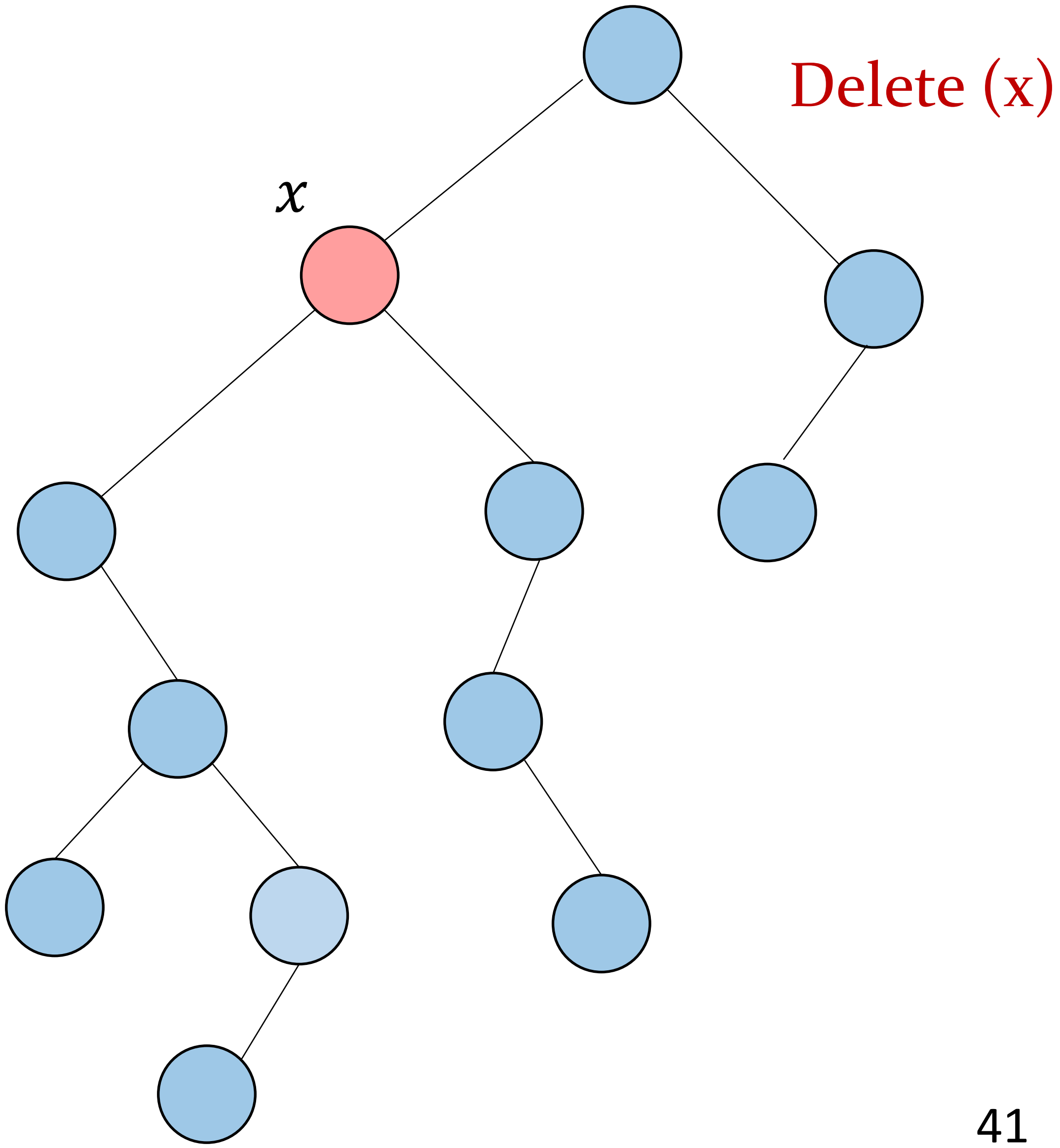
Run time:

Worst case: $\Theta(1)$

Case 3

- Case 2: x has two child
  - Delete x, which makes an open spot.
  - A node y should fill this spot, Who should be y?



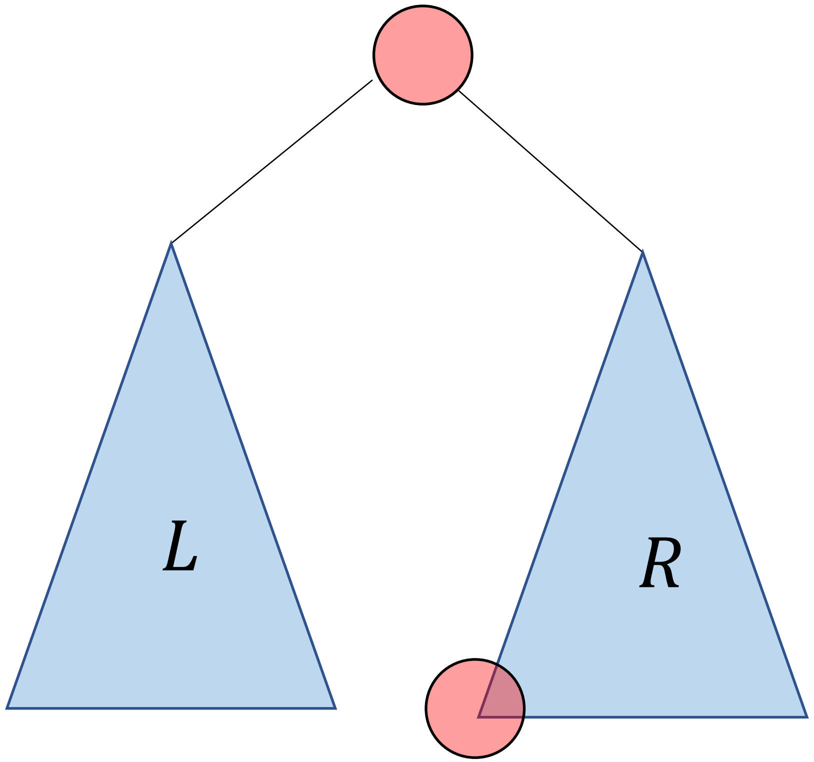


Case 3

A node y should fill this spot, such that $L < y < R$,

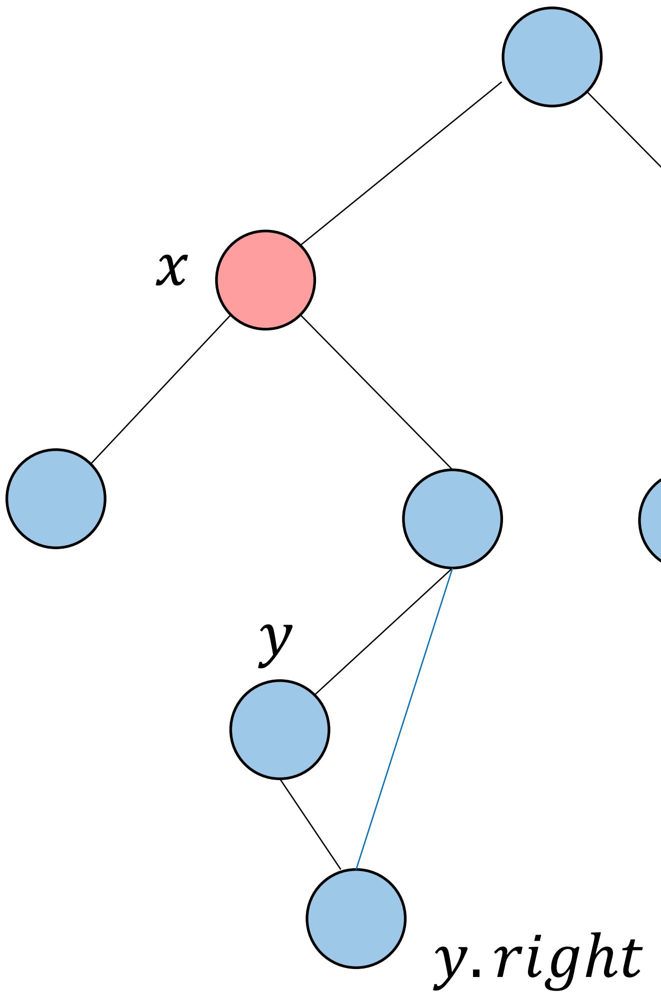
$y \leftarrow$ *the minimum of* R, i.e., Successor(x)

- $L < y$ because y is in R,
- $y < R$ because it's minimum

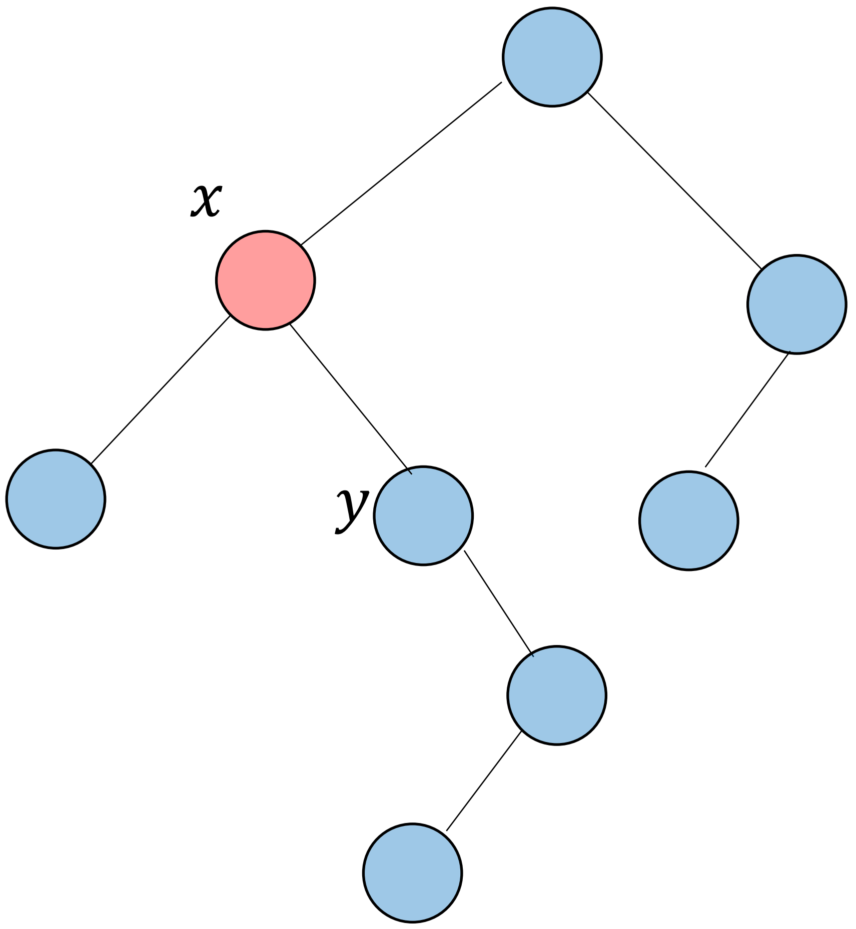


Sub cases for Case 3:

3.1: x is not y's parent

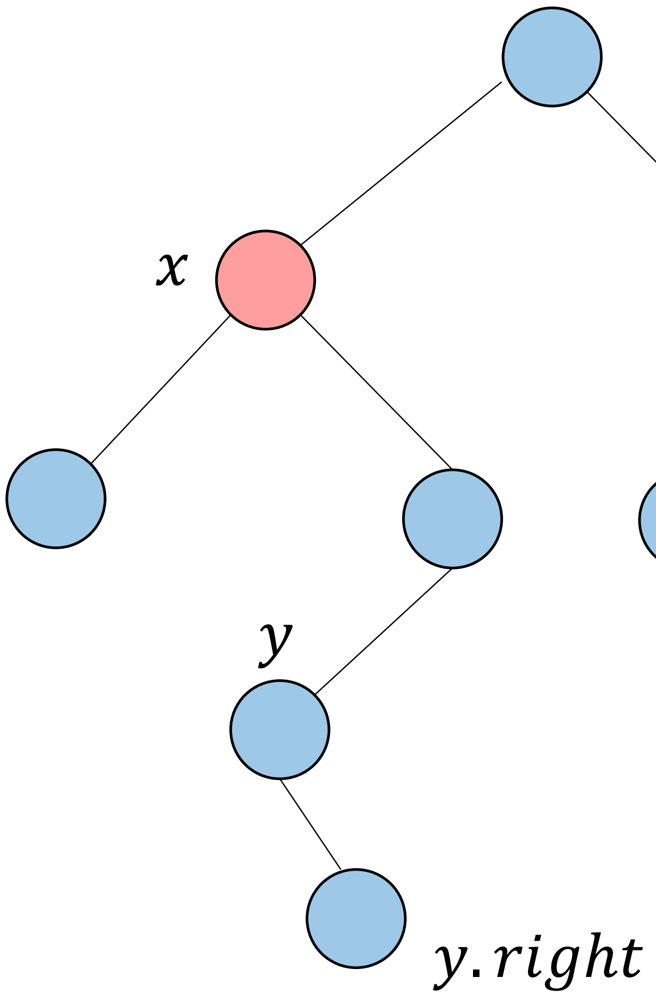


3.2: x is y's parent

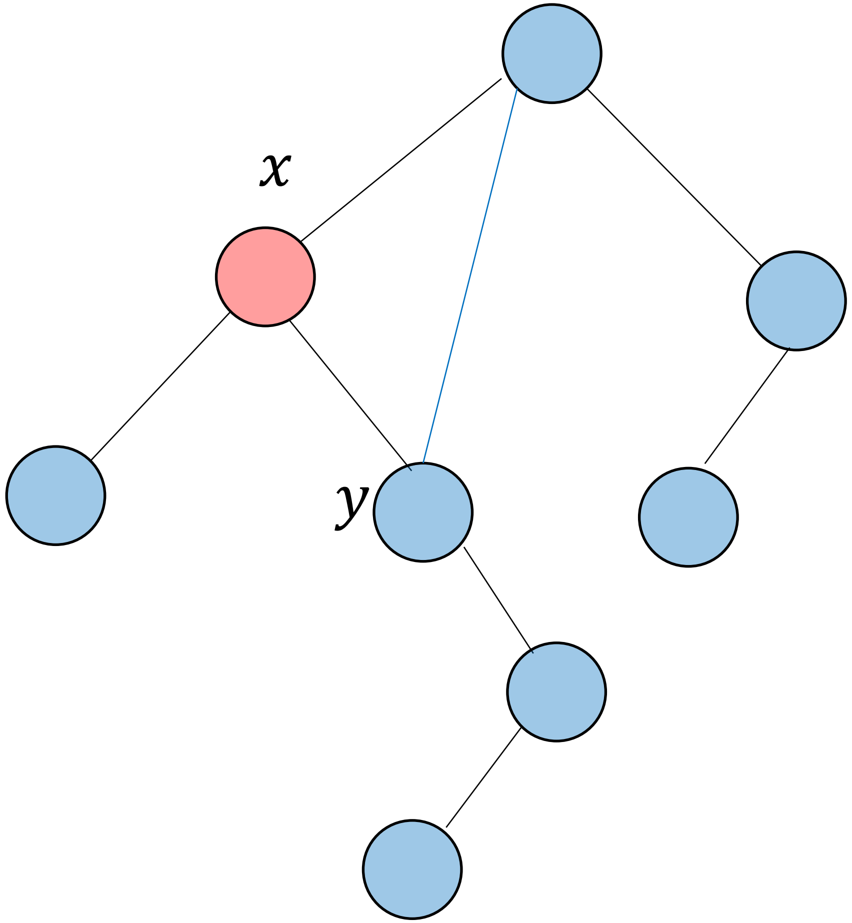


Sub cases for Case 3:

3.1: x is not y's parent

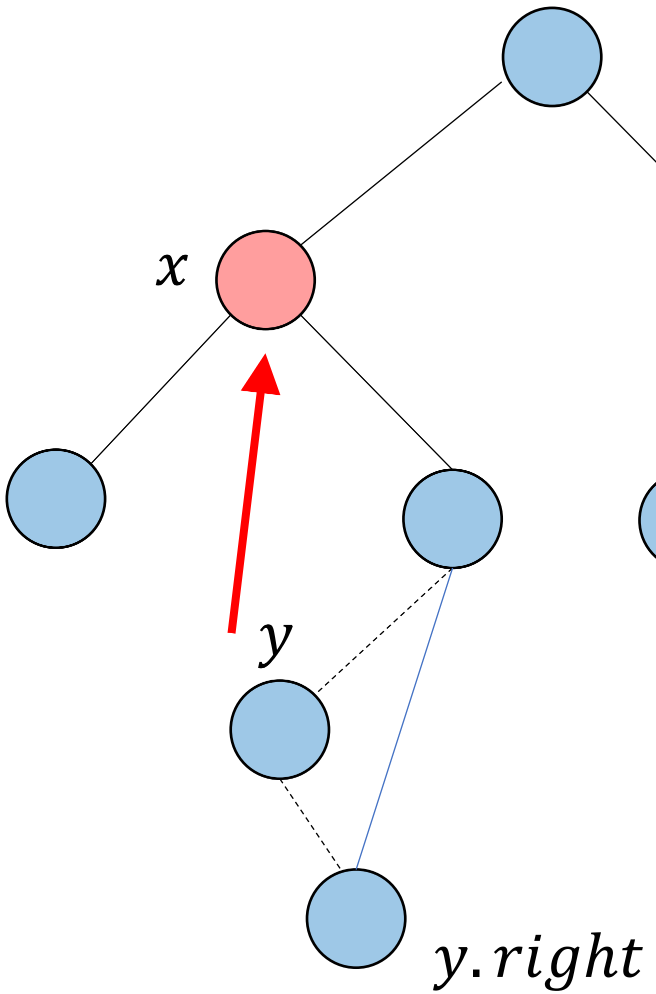


3.2: x is y's parent

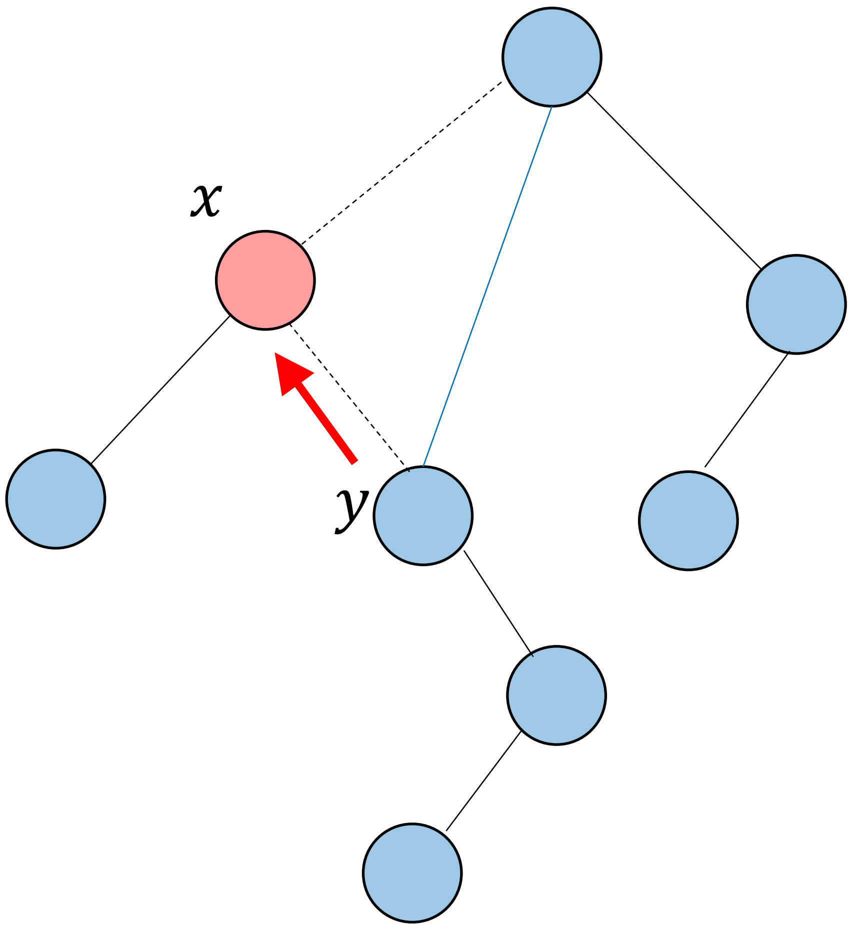


Sub cases for Case 3:

3.1: x is not y's parent



3.2: x is y's parent



TreeDelete(T, x) - psudocode

$TREE - DELETE(T, x)$

if $x.left\ == \ NIL$
    $TRANSPLANT(T, x, x.right)$
elseif $x.right\ == \ NIL$
    $TRANSPLANT(T, x, x.left)$

else $y\ =\ TREE - MINIMUM(x.right)$
    if $y.p \neq x$
        $TRANSPLANT(T, y, y.right)$
            $y.right\ = x.right$
        $y.right.p\ =\ y$
    $TRANSPLANT(T, x, y)$
    $y.left\ = x.left$
    $y.left.p\ =\ y$

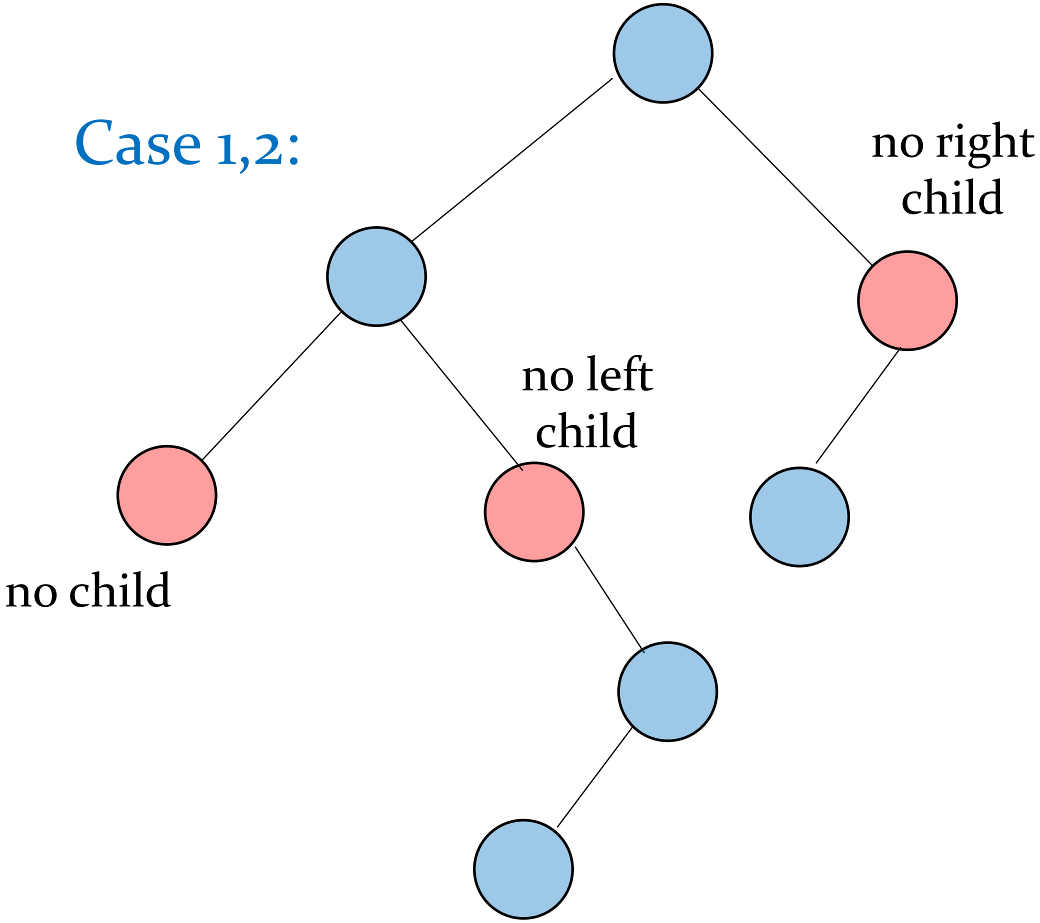
TreeDelete(T, x) - review

$TREE-DELETE(T, x)$

if $x.left\ == \ NIL$
  $TRANSPLANT(T, x, x.right)$
elseif $x.right\ == \ NIL$
  $TRANSPLANT(T, x, x.left)$

else $y\ =\ TREE-MINIMUM(x.right)$
  if $y.p \neq x$
    $TRANSPLANT(T, y, y.right)$
    $y.right\ = x.right$
    $y.right.p\ =\ y$
  $TRANSPLANT(T, x, y)$
  $y.left\ = x.left$
  $y.left.p\ =\ y$

Case 1,2:



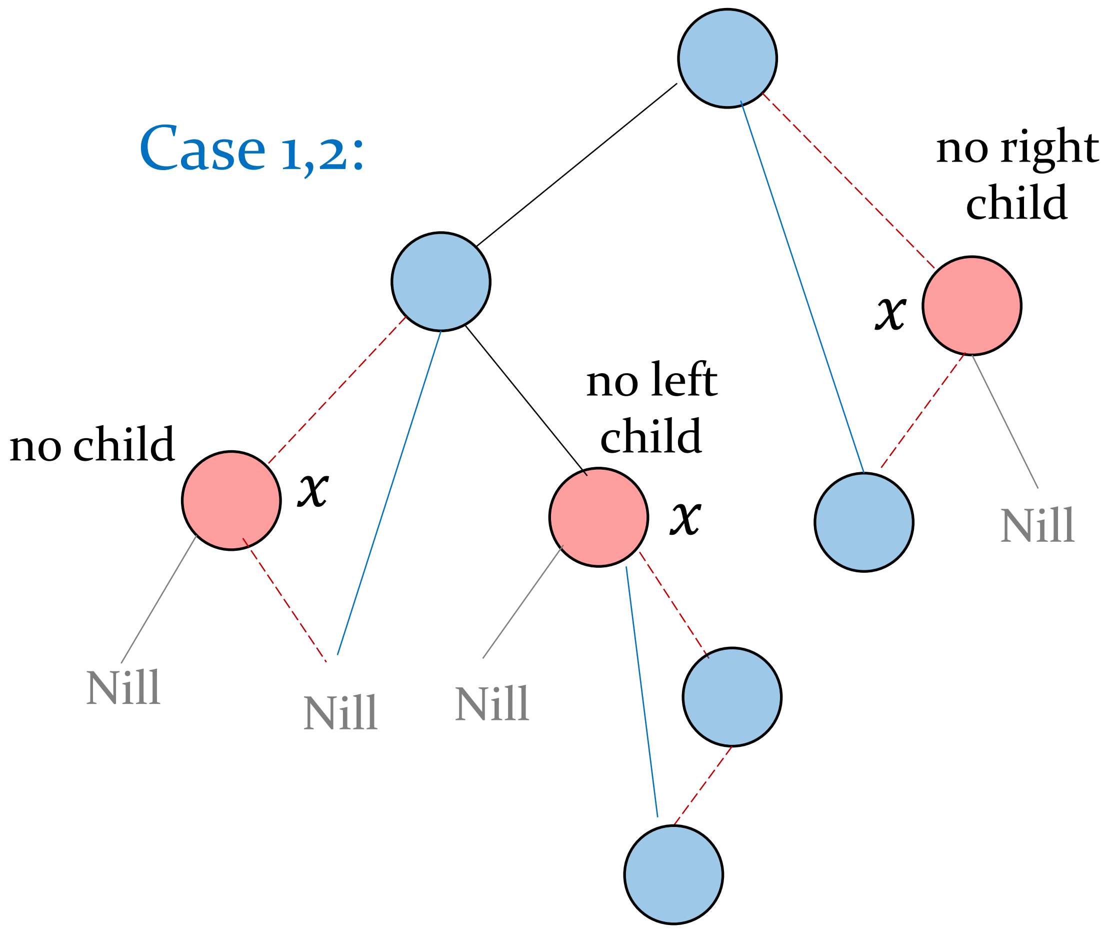
TreeDelete(T, x) - review

$TREE - DELETE(T, x)$

if $x.left\ == \ NIL$
$\quad TRANSPLANT(T, x, x.right)$
elseif $x.right\ == \ NIL$
$\quad TRANSPLANT(T, x, x.left)$

else $y\ =\ TREE - MINIMUM(x.right)$
$\quad$ **if** $y.p \neq x$
$\quad\quad TRANSPLANT(T, y, y.right)$
$\quad\quad y.right\ = x.right$
$\quad\quad y.right.p\ =\ y$
$\quad TRANSPLANT(T, x, y)$
$\quad y.left\ = x.left$
$\quad y.left.p\ =\ y$

Case 1,2:

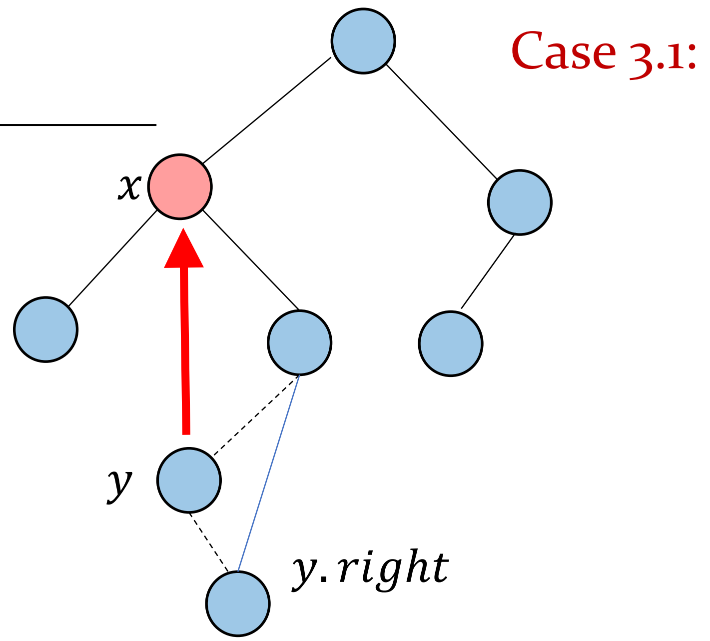


TreeDelete(T, x) - review

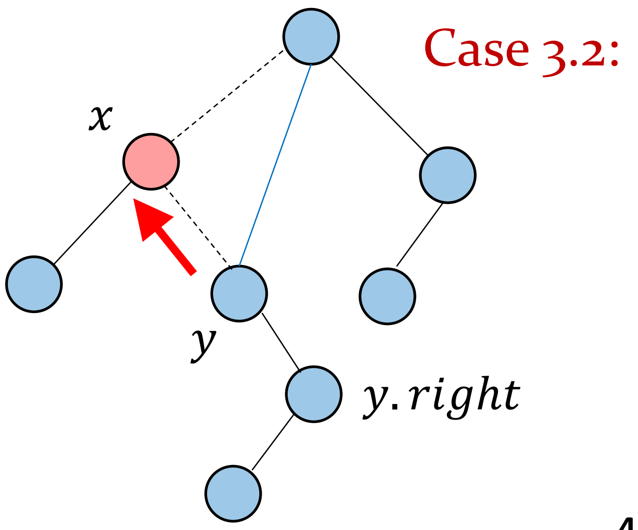
$TREE - DELETE(T, x)$

if $x.left == NIL$
$TRANSPLANT(T, x, x.right)$
elseif $x.right == NIL$
$TRANSPLANT(T, x, x.left)$

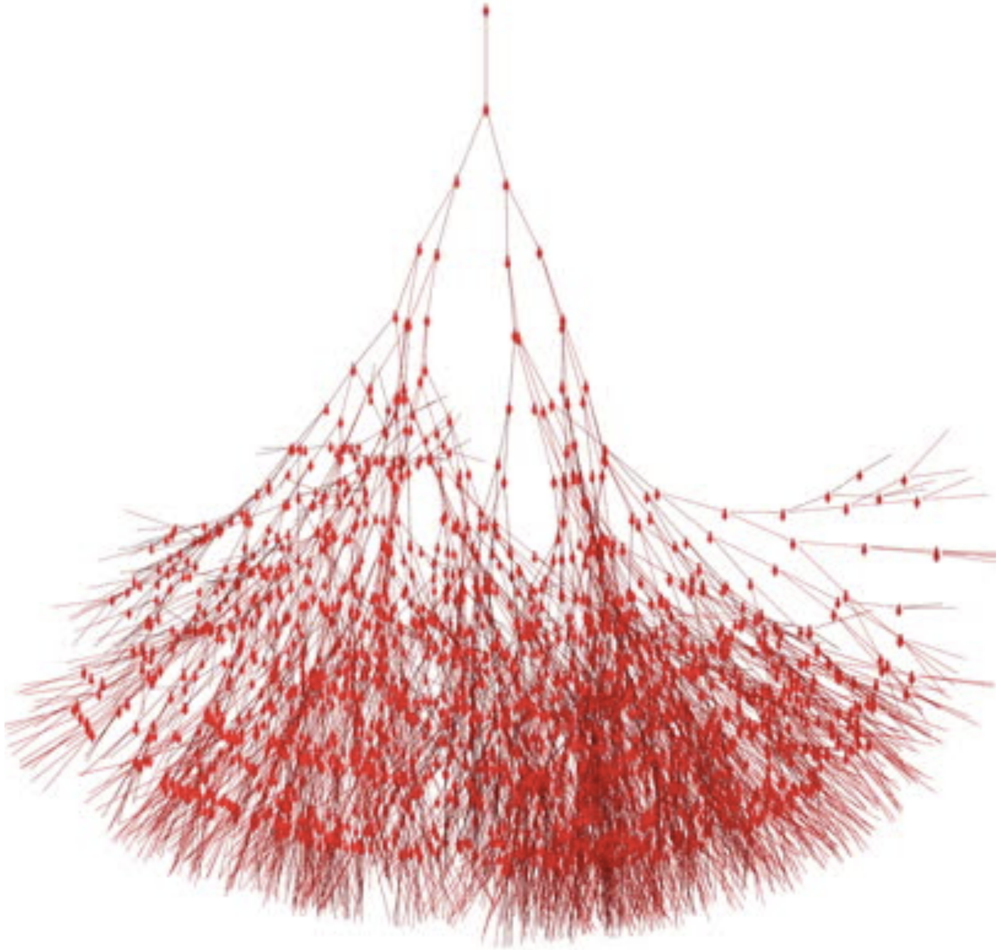
else $y = TREE - MINIMUM(x.right)$
if $y.p \neq x$ Case 3.1:
$TRANSPLANT(T, y, y.right)$
$y.right = x.right$
$y.right.p = y$
$TRANSPLANT(T, x, y)$ Case 3.2
$y.left = x.left$
$y.left.p = y$



Case 3: x has two child



About height of a tree



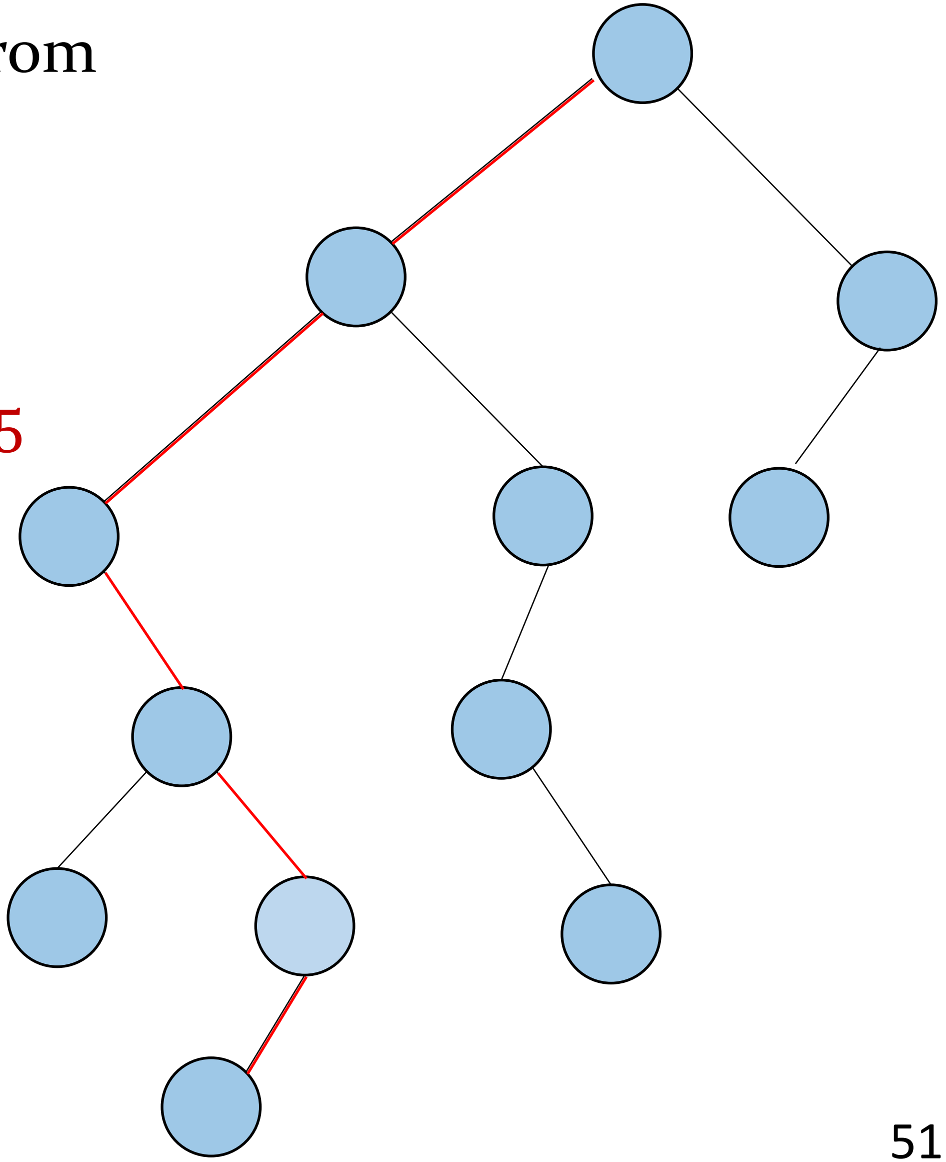
Height of a tree

Definition: Height of a tree is the longest path from the root to a leaf, in terms of number of edges.

Question: Consider a BST with n nodes, what's the highest an lowest it can be?

Answer: highest, $n-1=\Theta(n)$
Lowest, $\log \mathrm{n}\ -1=\Theta(\log n)$ when complete

$h\ =\ 5$
$\Theta(n)$



So, all algorithms with time complexity $\Theta(h)$ are $\Theta(n)$ in the worst case.

Height of a tree

Average height of a tree: $\Theta(\log n)$

For proof see the textbook, chapter 12.4.

Can we change the BST data structure such that the height of the tree is $\Theta(\log n)$ in the worst case?

A Balanced BST guarantees to have height in $\Theta(\log n)$.

We will talk about balanced Trees next week.

Questions