Australian National University

COMP3600/6466 Algorithms

Lecture 18

S2 2016

Dr. Hassan Hijazi

Prof. Weifa Liang

Binary Search Trees

WHAT ARE THEY?

Data Structures

a structured way of storing data

Independently discovered by a number of people in the late 1950s

Data

(key,information)

TREE; BINARY; SEARCH

Binary Search Trees

Dynamic Ordered Binary Trees

Dynamic

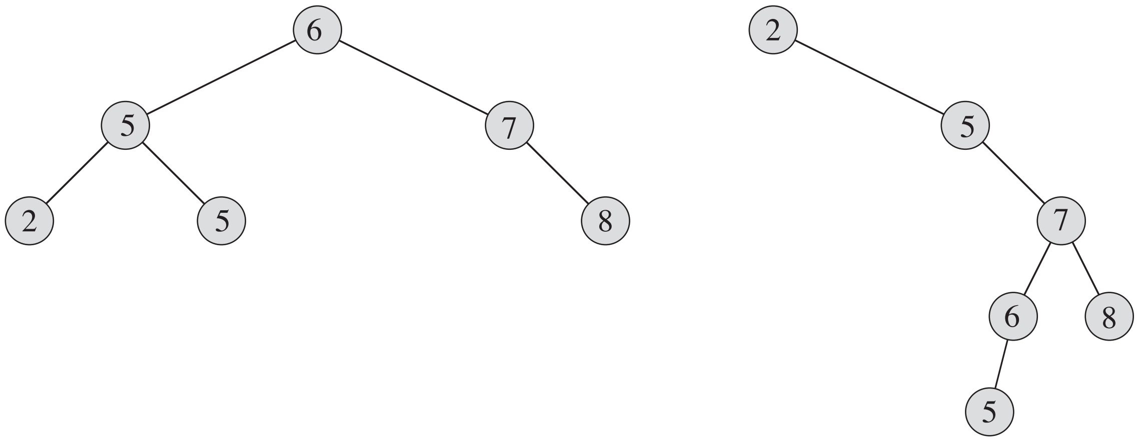
The tree changes after inserting/deleting an element

Ordered

binary-search-tree property

Left subtree = nodes with ≤ key values

Right subtree = nodes with ≥ key values



(a) (b)

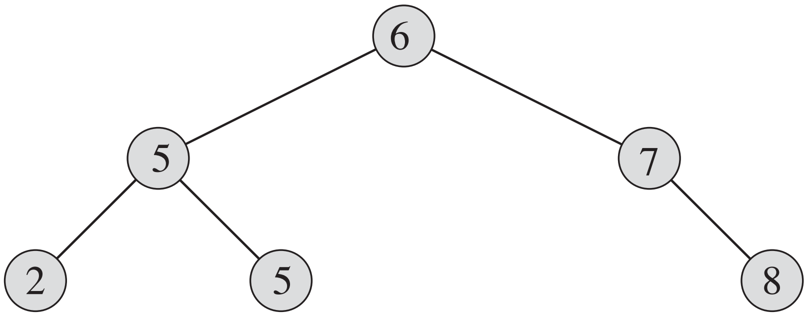
Binary Search Trees

Dynamic Ordered Binary Trees

Tree attributes:
root

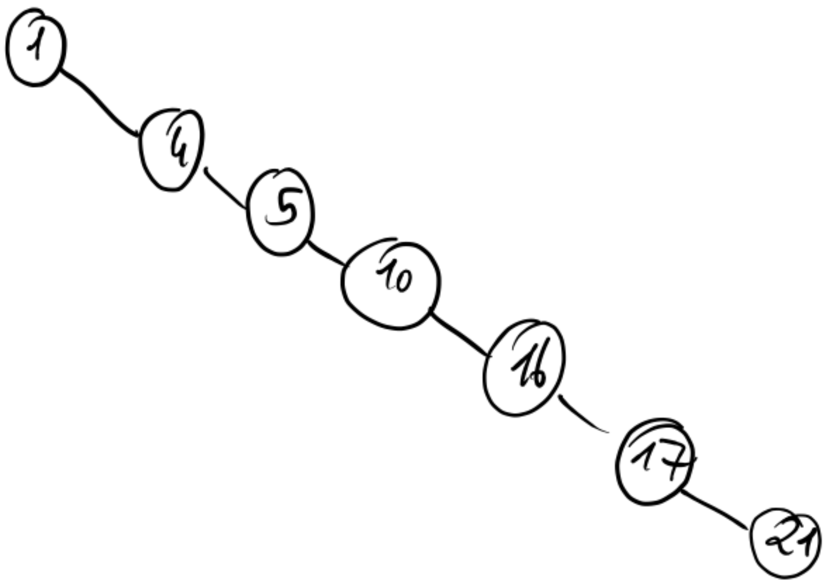
Node attributes:
key, left, right and parent

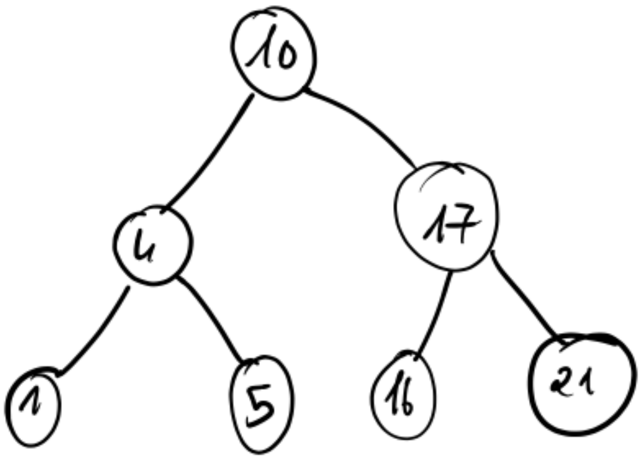
NIL
replaces missing child or parent



Exercise 18.1

For the set of $\{1, 4, 5, 10, 16, 17, 21\}$ of keys, draw binary search trees of heights 2, 3, 4, 5, and 6.

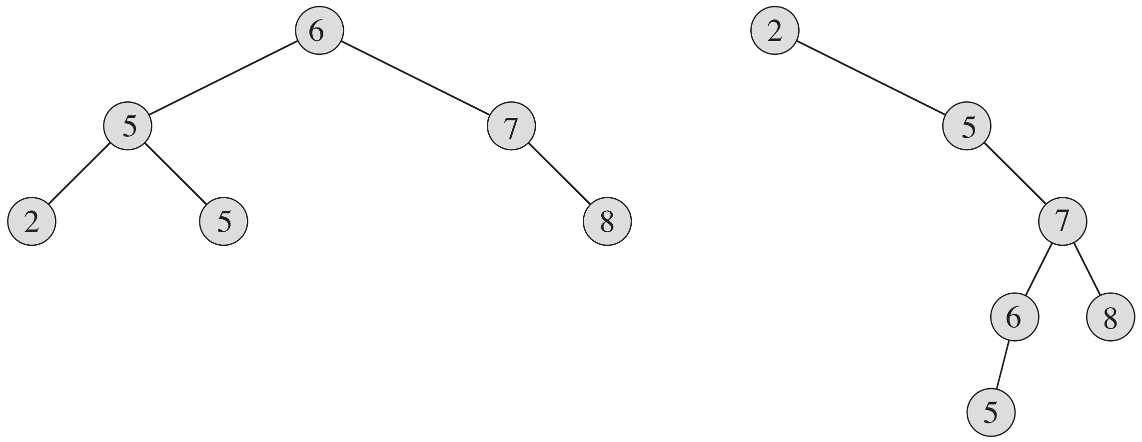




Binary Search Trees

Dynamic-Set Operations

TRAVERSE, SEARCH, INSERT, DELETE,
MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR



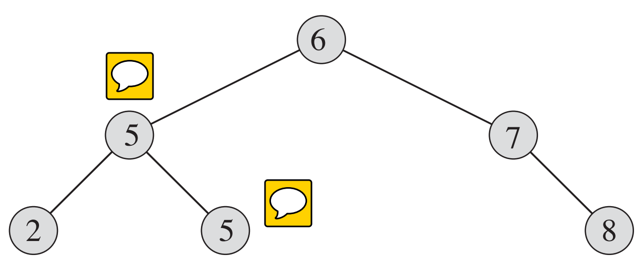
(a) (b)

Binary Search Trees

TRAVERSE

INORDER_TREE_WALK(x)

if $x \neq$ NIL
  INORDER_TREE_WALK(x.left)
  print $x.key$
  INORDER_TREE_WALK(x.right)



$\mathrm{O}(n)$

2,5,5,6,7,8

Binary Search Trees

TRAVERSE

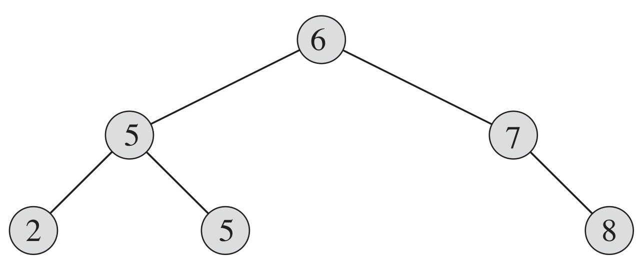
$\text{PREORDER\_TREE\_WALK}(x)$

$\quad$ **if** $x \neq \text{NIL}$

$\qquad$ **print** $x.key$

$\qquad \text{PREORDER\_TREE\_WALK}(x.\text{left})$

$\qquad \text{PREORDER\_TREE\_WALK}(x.\text{right})$



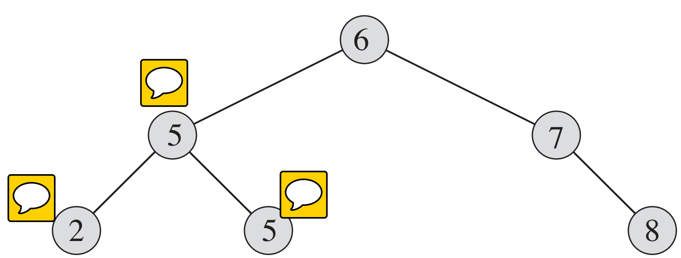
$$O(n)$$

6,5,2,5,7,8

Binary Search Trees

TRAVERSE

POSTORDER_TREE_WALK(x)
if $x \neq$ NIL
  POSTORDER_TREE_WALK(x.left)
  POSTORDER_TREE_WALK(x.right)
  print $x.key$



2,5,5,8,7,6

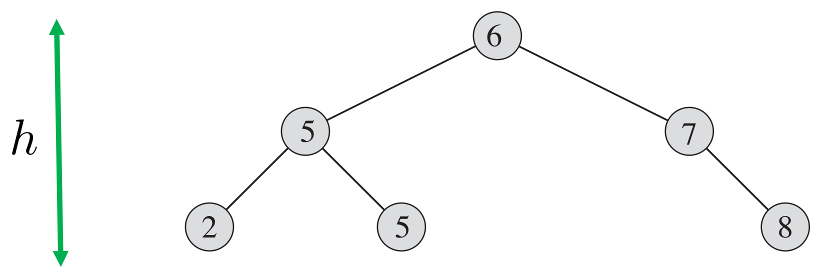
$$O(n)$$

Binary Search Trees

SEARCH

TREE-SEARCH(x, k)

if x = NIL or $k = x$.key
  return x
if $k < x$.key
  return TREE-SEARCH$(x$.left,$k)$
  else return TREE-SEARCH$(x$.right,$k)$



$\mathrm{O}(h)$

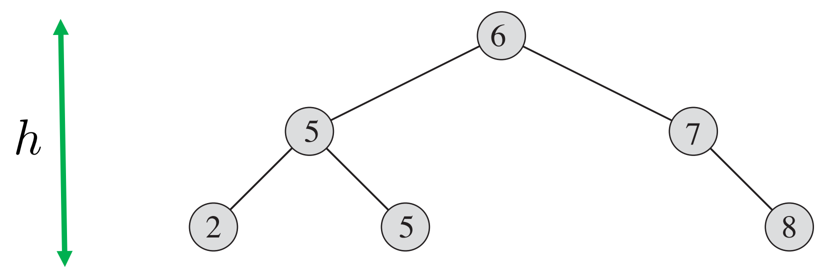
TREE-SEARCH(T.root,2)

Binary Search Trees

ITERATIVE SEARCH

ITERATIVE-TREE-SEARCH(x, k)

while $x \neq$ NIL and $k \neq x$.key
  if $k < x$.key
    $x = x$.left
  else $x = x$.right
return x



$$O(h)$$

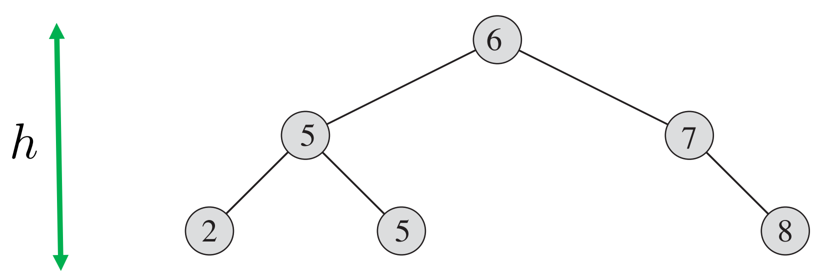
ITERATIVE-TREE-SEARCH(T.root,2)

Binary Search Trees

MIN, MAX

MIN(x)
while x.left $\neq$ NIL
    $x = x$.left
return x

MAX(x)
while x.right $\neq$ NIL
    $x = x$.right
return x



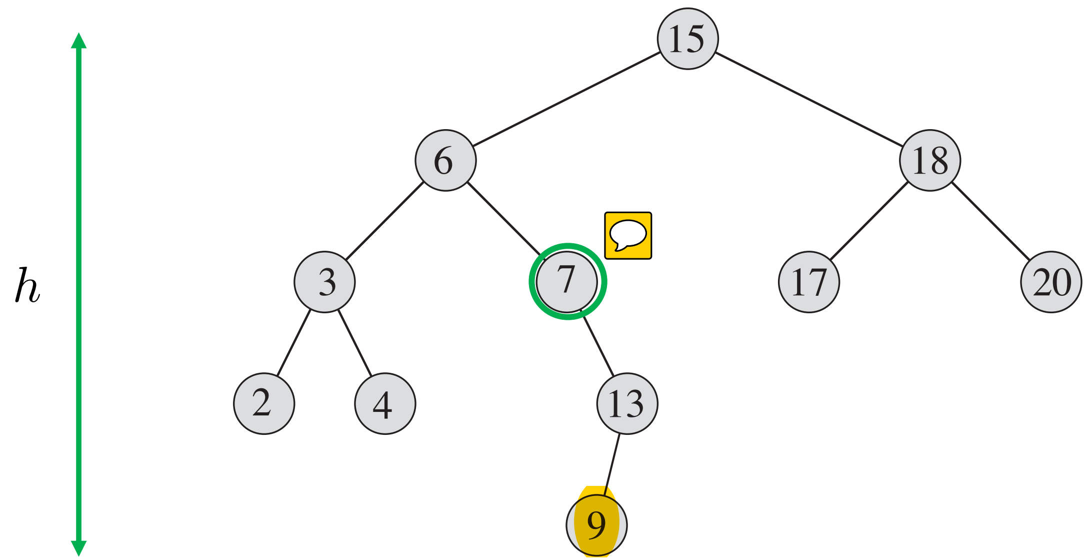
$$\mathrm{O}(h)$$

MIN(T.root) and MAX(T.root)

Binary Search Trees

SUCCESSOR

Case 1: x has a right child

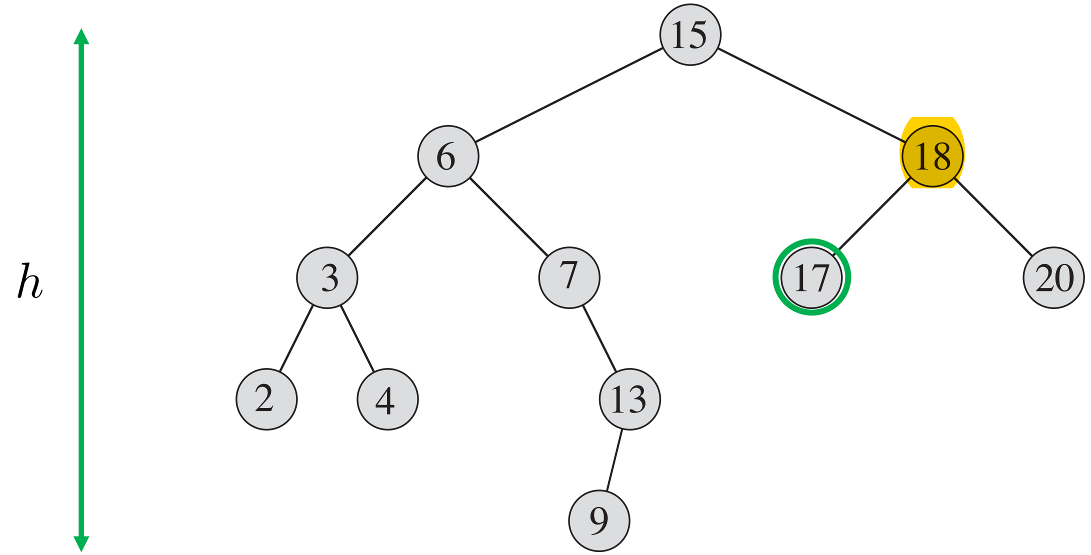


return minimum in right sub-tree

Binary Search Trees

SUCCESSOR

Case 2 (a): x has no right child and is a left child

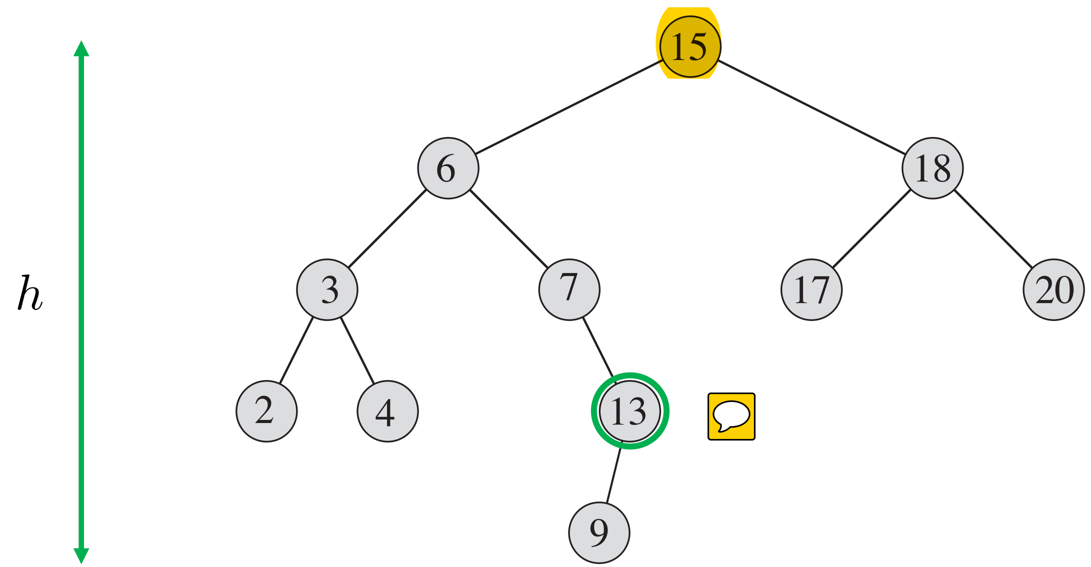


return parent

Binary Search Trees

SUCCESSOR

Case 2 (b): x has no right child and is a right child



return first ancestor on the right

Binary Search Trees

SUCCESSOR

TREE_SUCCESSOR(x)

if x.right $\neq$ NIL

  return MIN(x.right)

case 1

$y \leftarrow x$.parent

while $y \neq$ NIL and $x = y$.right

  $x \leftarrow y$

  $y \leftarrow y$.parent

return y

case 2

Binary Search Trees

INSERT

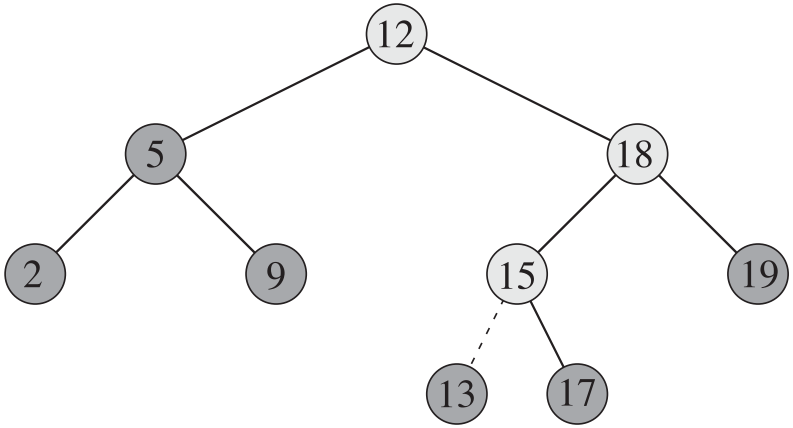


Figure 12.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

Binary Search Trees

INSERT

TREE-INSERT(T, z)

```
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z          // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

Binary Search Trees

Delete

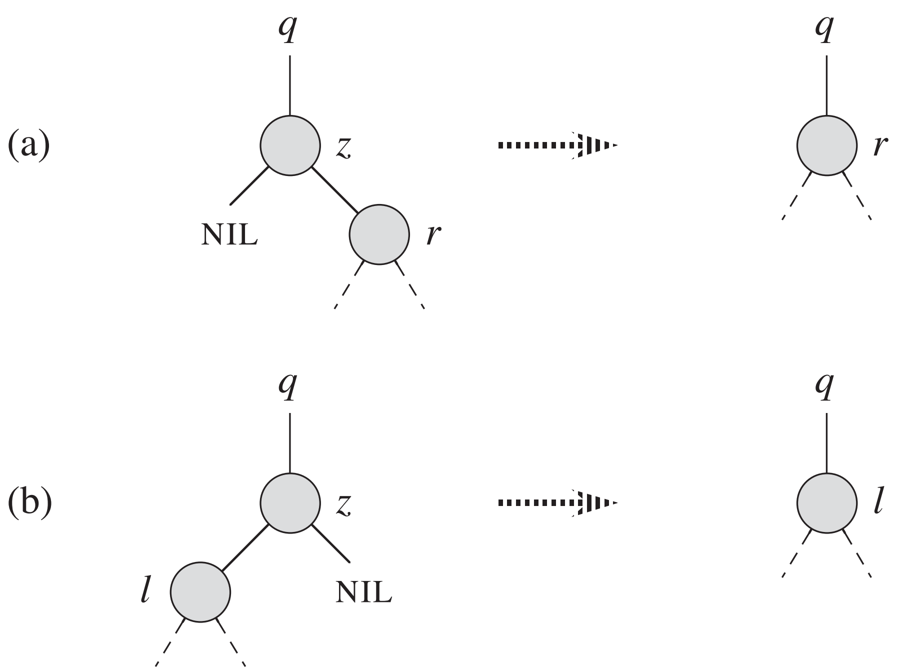
Changes the structure of the tree to maintain the binary-search-tree property

Two cases:

1. z has at most 1 child: we replace z by its child or delete z if it has none.
2. z has 2 children: we replace z by its successor.

Binary Search Trees

Delete: Case 1

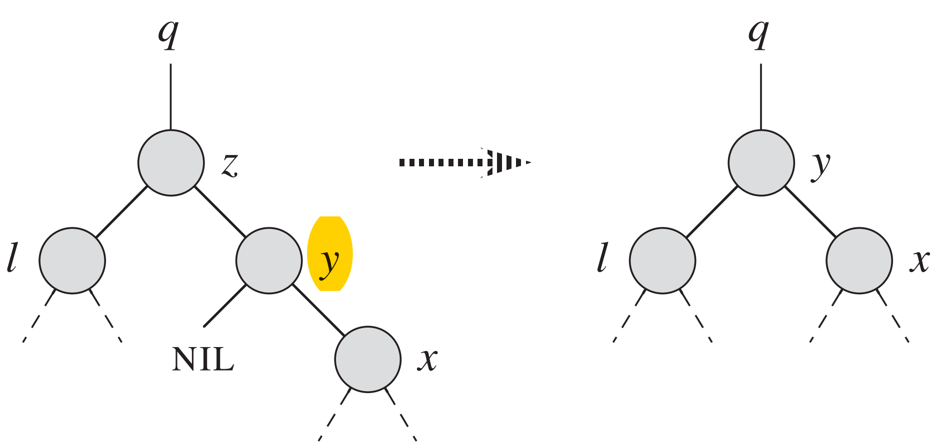


Binary Search Trees

Delete: Case 2 (a)

right child is the successor (y)

(c)

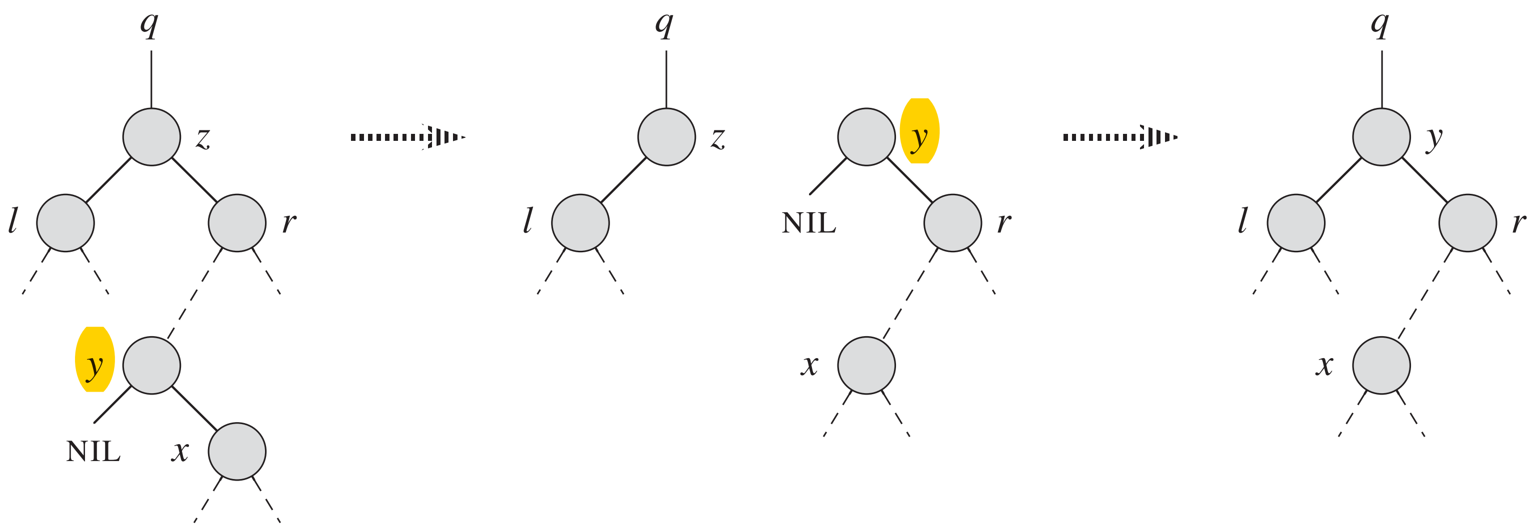


Binary Search Trees

Delete: Case 2 (b)

successor (y) is deeper in the right subtree

(d)



How can we be sure y has no left son?

Binary Search Trees

Delete

TRANSPLANT(T, u, v)

if $u.p == \text{NIL}$
    $T.root = v$

if u is the root

elseif $u == u.p.left$
    $u.p.left = v$

if u is a left child

else $u.p.right = v$

if $v \neq \text{NIL}$
    $v.p = u.p$

update v's parent

Binary Search Trees

Delete

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
```

case 1

```
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

case 2

Exercise 18.2

Is the operation of deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is or give a counterexample.

Delete A then B

Delete B then A