12 Binary Search Tree

Binary Search Trees (BSTs) are data structures that support **many dynamic set operations**.
The typical operations include:

- SEARCH
- INSERT
- DELETE
- MINIMUM
- MAXIMUM
- PREDECESSOR
- SUCCESSOR

12.1 Binary Search Trees

The keys in a binary search tree are always stored in such a way that satisfy the binary search tree property:

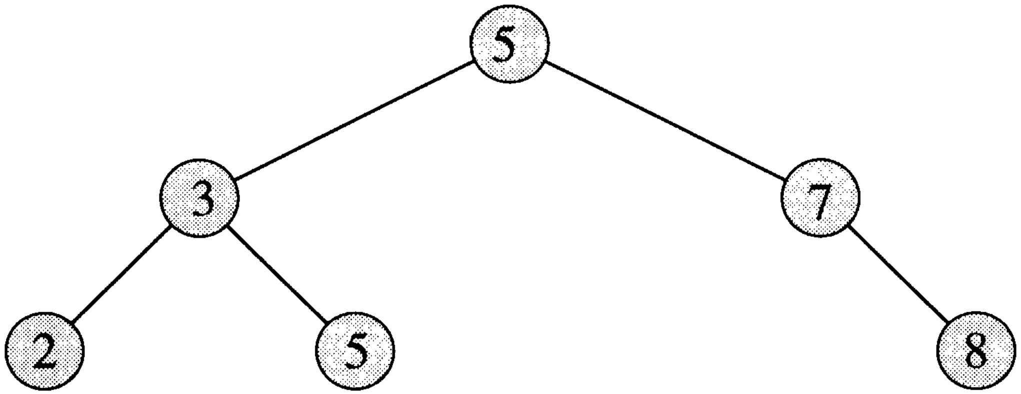
Let x be an internal node, not a leaf node!

- If y is a node in the left subtree of x, then $key[y] \leq key[x]$.
- If y is a node in the right subtree of x, then $key[y] > key[x]$.

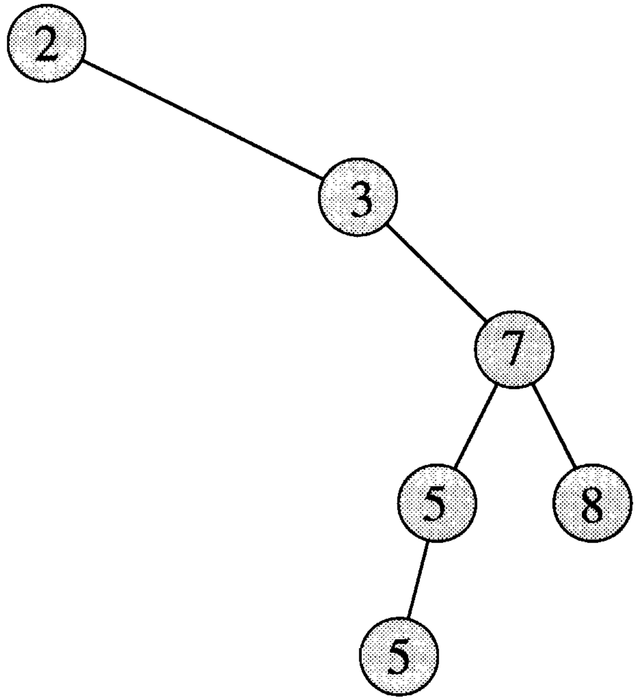
Each node x in a binary tree has three pointers:

- The parent pointer $p[x]$
- The left child pointer $left[x]$
- The right child pointer $right[x]$

12.1 Binary Search Trees (cont.)



(a)



(b)

12.1 Properties of binary search trees

Three ways to traverse a binary search tree:

- **In-order tree walk (LNR):** visit the Left subtree, the root Node, and the Right subtree
- **Pre-order tree walk (NLR):** visit the root Node, the Left subtree and the Right subtree
- **Post-order tree walk (LRN):** visit the Left subtree, the Right subtree, and the root Node.

12.1 Binary Search Trees (cont.)

In-order tree walk

INORDER_TREE_WALK(x)

if $x \neq NIL$

then INORDER_TREE_WALK($left[x]$);

print $key[x]$;

INORDER_TREE_WALK($right[x]$);

Required time: $O(n)$

12.1 Binary Search Trees (cont.)

Pre-order tree walk

PREORDER_TREE_WALK(x)

if $x \neq NIL$

    print $key[x]$;

then PREORDER_TREE_WALK($left[x]$);

    PREORDER_TREE_WALK($right[x]$);

Required time: $O(n)$

12.1 Binary Search Trees (cont.)

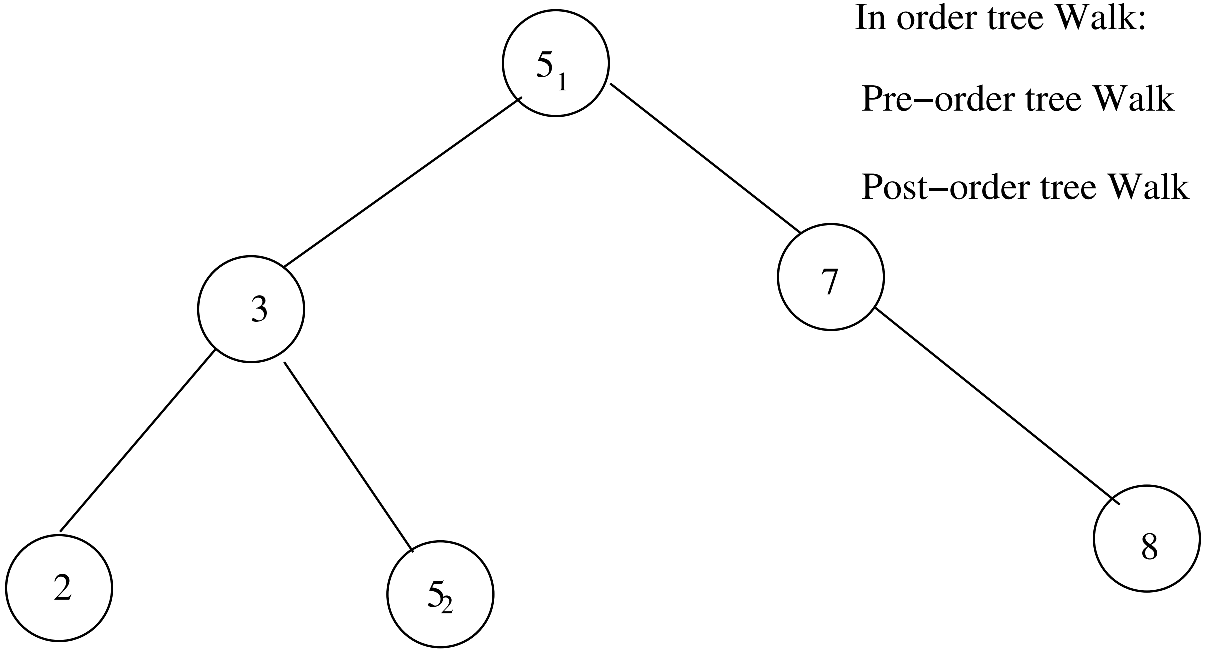
Post-order tree walk

POSTORDER_TREE_WALK(x)

1	if	$x \neq NIL$	
2		then	POSTORDER_TREE_WALK($left[x]$);
3			POSTORDER_TREE_WALK($right[x]$);
4			**print** $key[x]$

Required time: $O(n)$

12.1 Binary Search Trees (cont.)



In order tree Walk: $2, 3, 5_2, 5_1, 7, 8$

Pre–order tree Walk $5_1, 3, 2, 5_2, 7, 8$

Post–order tree Walk $2, 5_2, 3, 8, 7, 5_1$

Examples for different tree traversals

12.2 Querying a binary search tree

Searching: given a key k, **search** whether there is an element with key k in the dynamic set.

```
TREE_SEARCH(x, k)
    if    x = NIL or k = key[x]
          then return x
    if    k ≤ key[x]
          then return TREE_SEARCH(left[x], k)
          else return TREE_SEARCH(right[x], k)
```

Required time: $O(h)$, where h is the height of the tree.

12.2 Querying a binary search tree (cont.)

Minimum and Maximum: Find the elements with the minimum and maximum keys in a dynamic set

TREE_MINIMUM(x)

while $left[x] \neq NIL$
    do $x \leftarrow left[x]$
return x

TREE_MAXIMUM(x)

while $right[x] \neq NIL$
    do $x \leftarrow right[x]$
return x

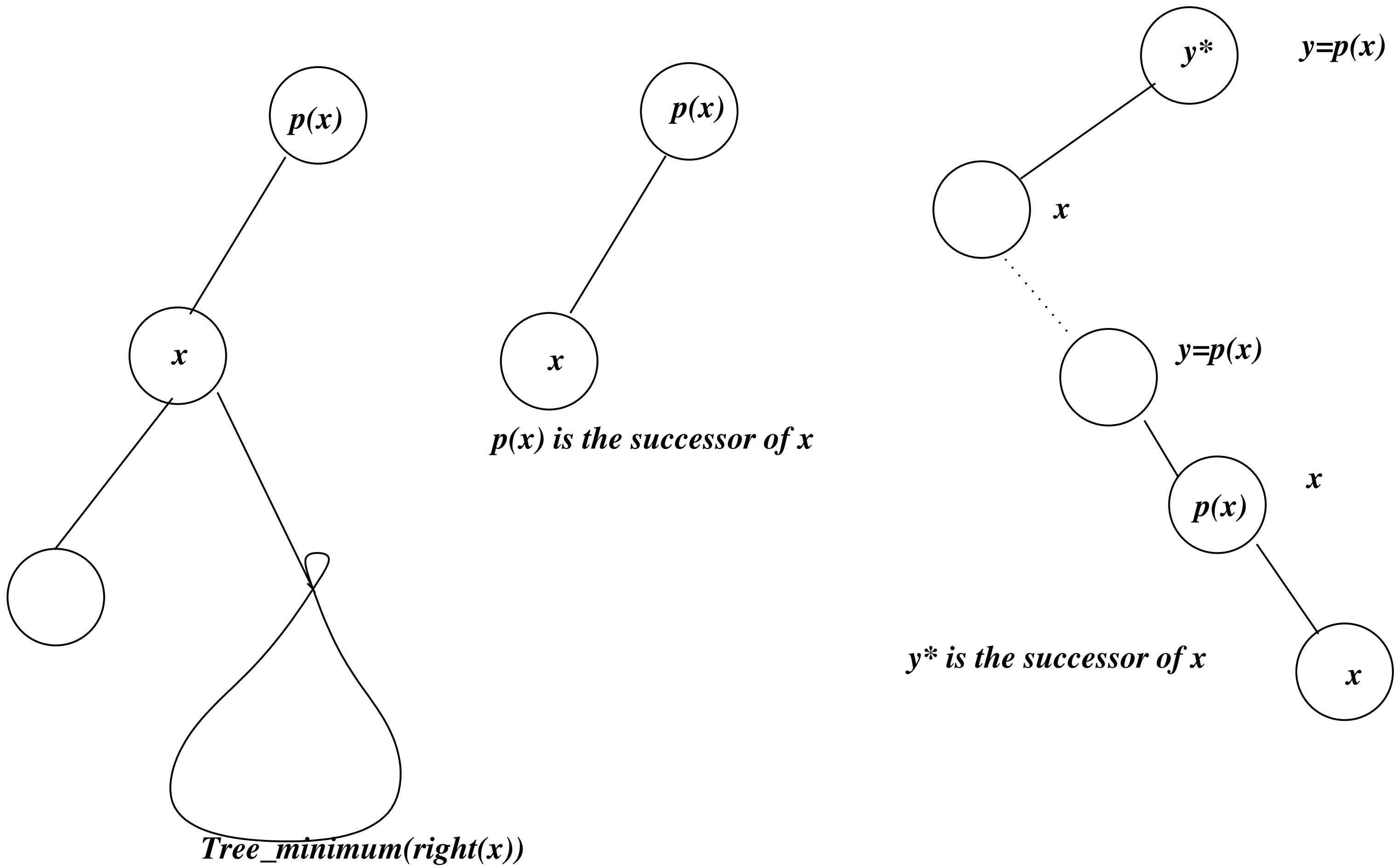
12.2 Querying a binary search tree (cont.)

Successor and Predecessor: If all keys are distinct, the successor of a node x is the node with the smallest key greater than $key[x]$.

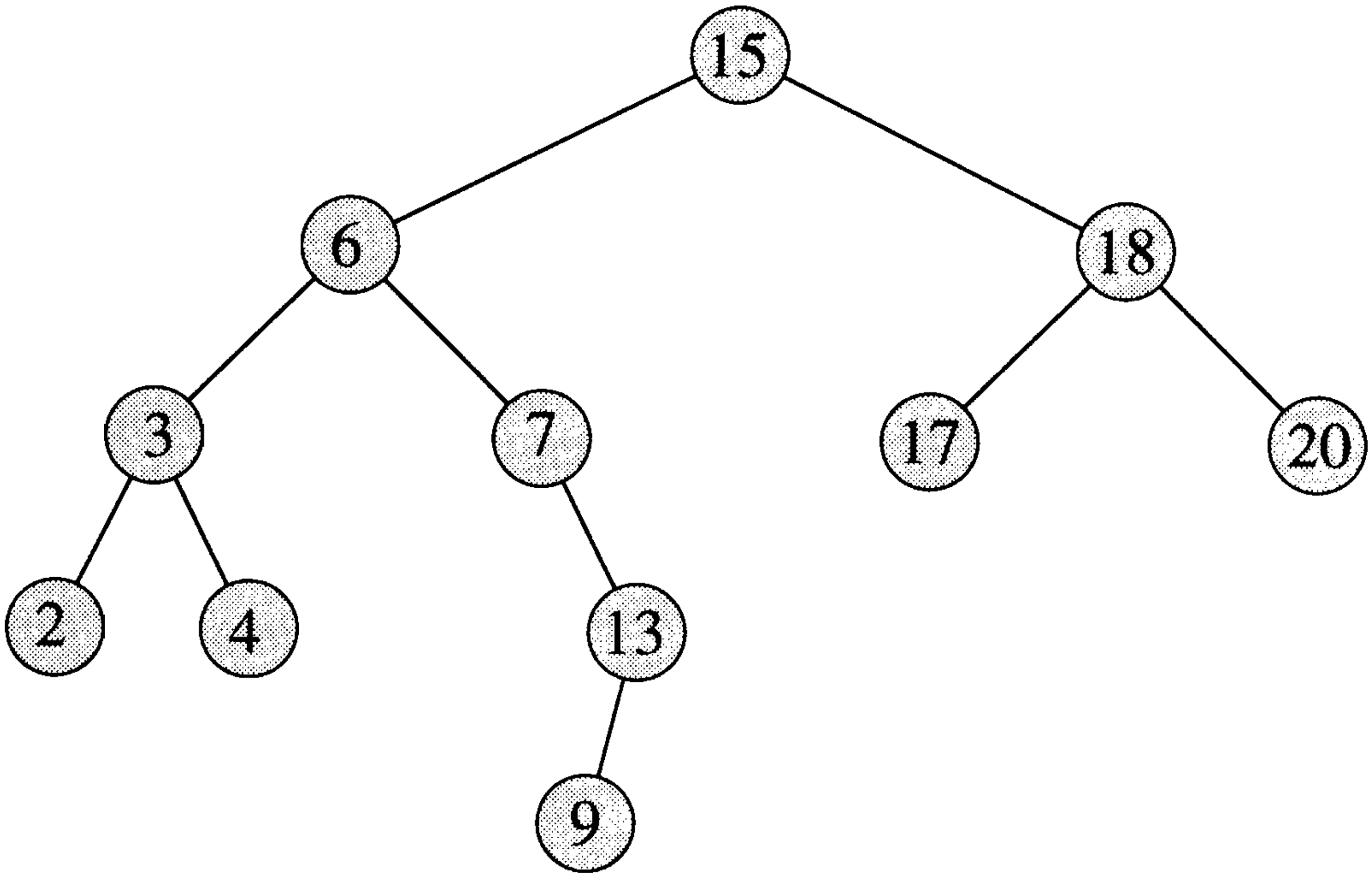
```
TREE_SUCCESSOR(x)
    if     right[x] ≠ NIL
           then    return TREE_MINIMUM(right[x])
    y ← p[x];
    while  y ≠ NIL and x = right[y]
           do      x ← y;
                   y ← p[y];
    return y.
```

Required time: $O(h)$, where h is the height of the tree.

12.2 Querying a binary search tree (cont.)



Exercise: Can you devise a procedure for finding the predecessor of an element in a BST?



Queries in the tree: (1) search for the key 13 in the tree; (2) the minimum key and the maximum key in the tree; (3) the successor of key 13.

12.3 Insertion and Deletion

The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change.

The data structure must be modified to reflect this change, but in such a way that the binary search tree property continues to hold.

That is, let x be an internal node in a binary search tree.

- If y is a node in the left subtree of x, then $key[y] \leq key[x]$.
- If y is a node in the right subtree of x, then $key[y] > key[x]$.

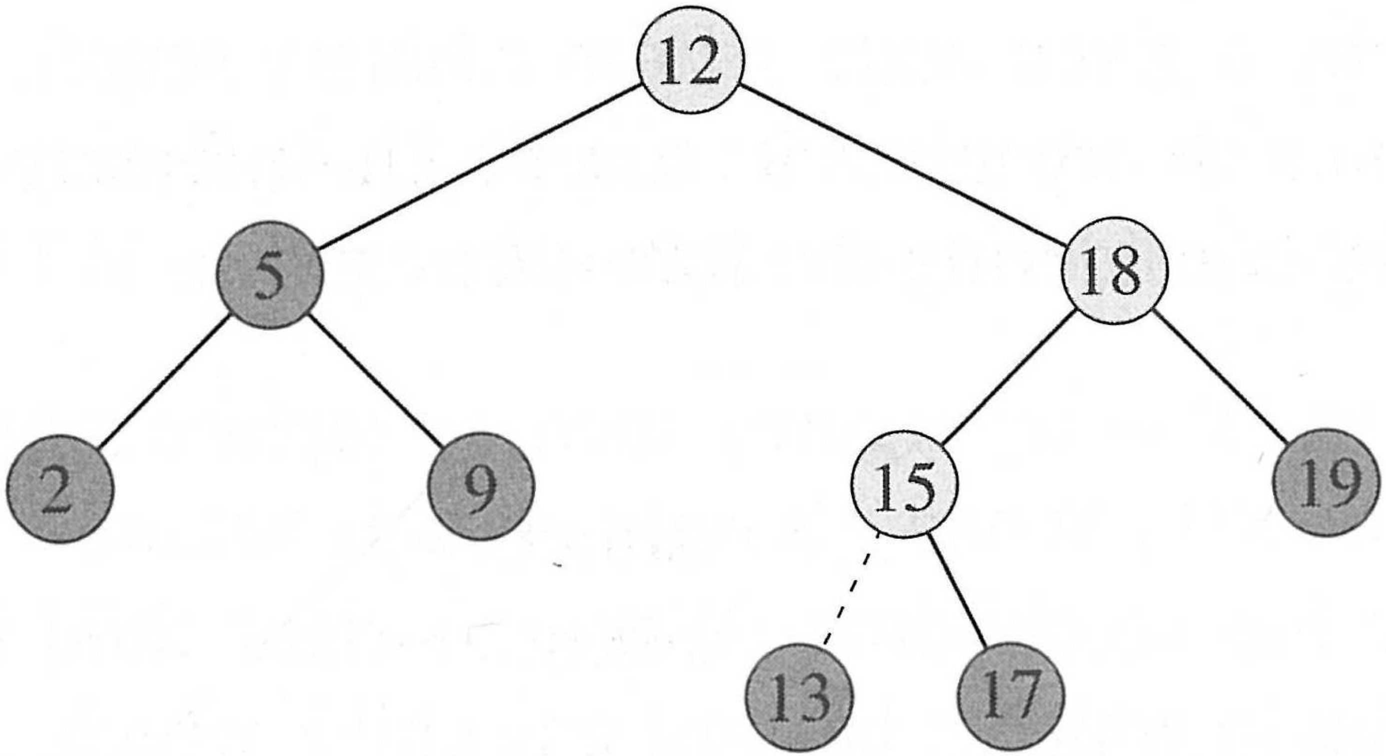
12.3 Insertion

TREE_INSERT(T, z): insert a new element z into T

```
y ← NIL
x ← root[T]
while x ≠ NIL
      do y ← x
         if key[z] ≤ key[x]
            then x ← left[x]
            else x ← right[x]
p[z] ← y
if y = NIL
   then root[T] ← z
   else if key[z] ≤ key[y]
           then left[y] ← z
   else right[y] ← z
```

Required time: $O(h)$, where h is the height of the tree.

12.3 Insertion (cont)



Insertion of entry 13 (Cormen et al., p295), What about the insertion of entry 20?

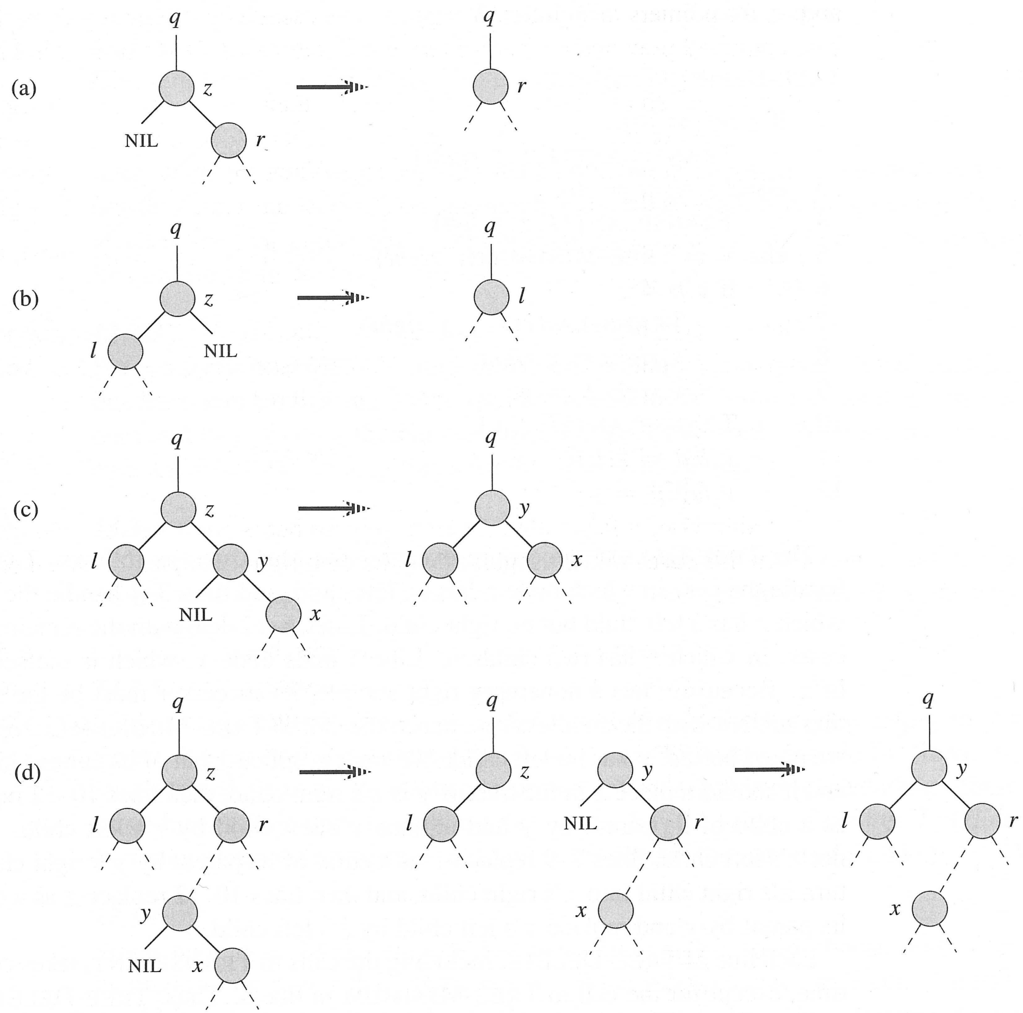
12.3 Deletion

Three cases for deletion of an element with key z in a binary search tree:

1. z has only 1 child: we replace z by its child (see case (a) and case (b) in the next slide).
2. z has 2 children: we replace z by its successor (see case (c) and case (d) in the next slide).
3. z has 0 children: we simply delete z.

In case (c) and case (d), node z is replaced by a node y, where y is the root of a subtree rooted at itself and node y does NOT have a left child, i.e., $left(y) = NIL$, Why?

12.3 Deletion



Deletion in binary search trees (Cormen et al., p297)

12.3 Deletion

For the deletion operation we use the procedure TRANSPLANT. It replaces the subtree rooted at u with the subtree rooted at v.

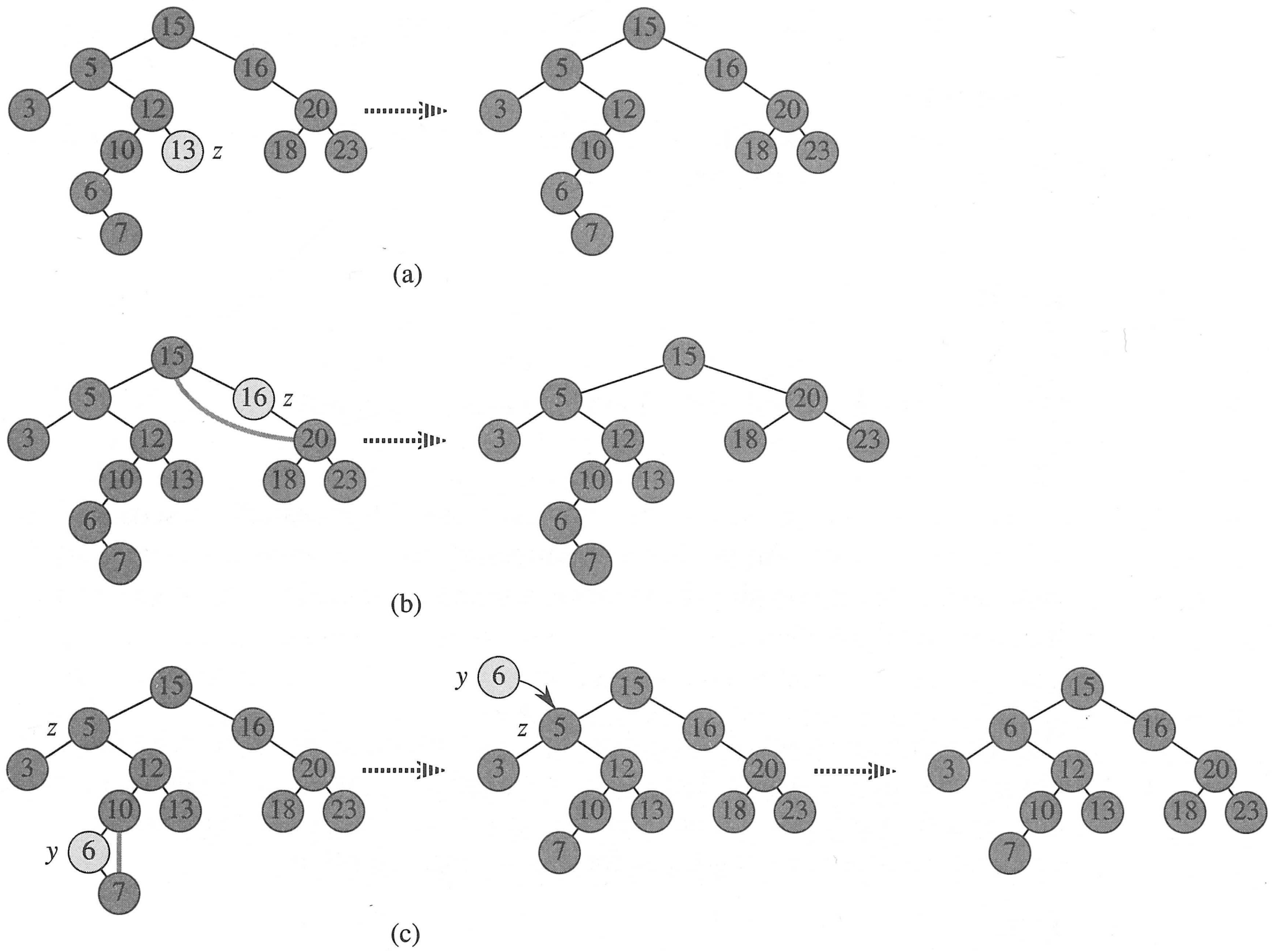
TRANSPLANT(T, u, v)

if $p[u] = NIL$
    then $root[T] \leftarrow v$
else if $u = left[p[u]]$
    then $left[p[u]] \leftarrow v$
    else $right[p[u]] \leftarrow v$
if $v \neq NIL$
    then $p[v] \leftarrow p[u]$

12.3 Deletion

```
TREE_DELETE(T, z)
    if        left[z] = NIL
              then    TRANSPLANT(T, z, right[z])
    else if   right[z] = NIL
              then    TRANSPLANT(T, z, left[z])
    else y ← TREE_MINIMUM(right[z])
              if      p[y] ≠ z
                      then    TRANSPLANT(T, y, right[y])
                              right[y] ← right[z]
                              p[right[z]] ← y
              TRANSPLANT(T, z, y)
              left[y] ← left[z]
              p[left[y]] ← y
```

12.3 Deletion



Deletion in binary search trees (Cormen et al., 2nd edition)