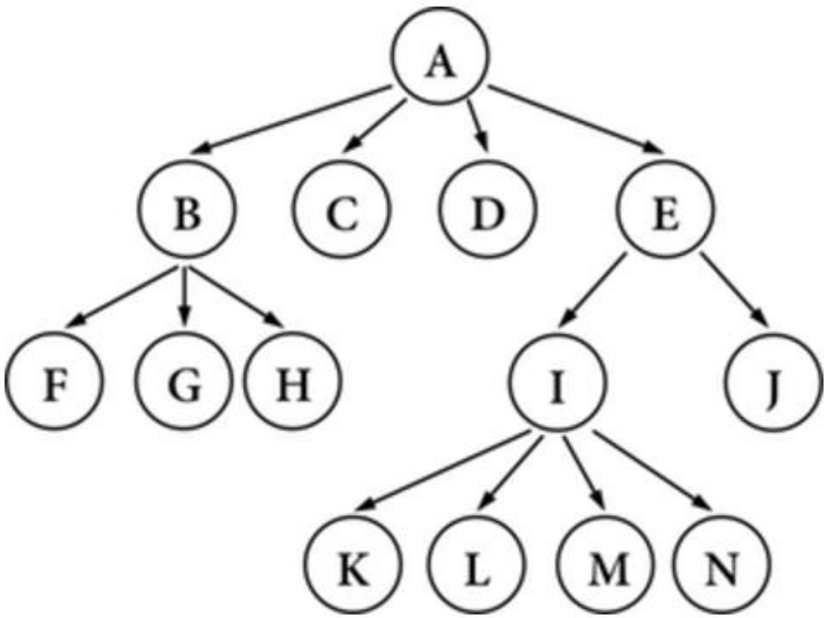
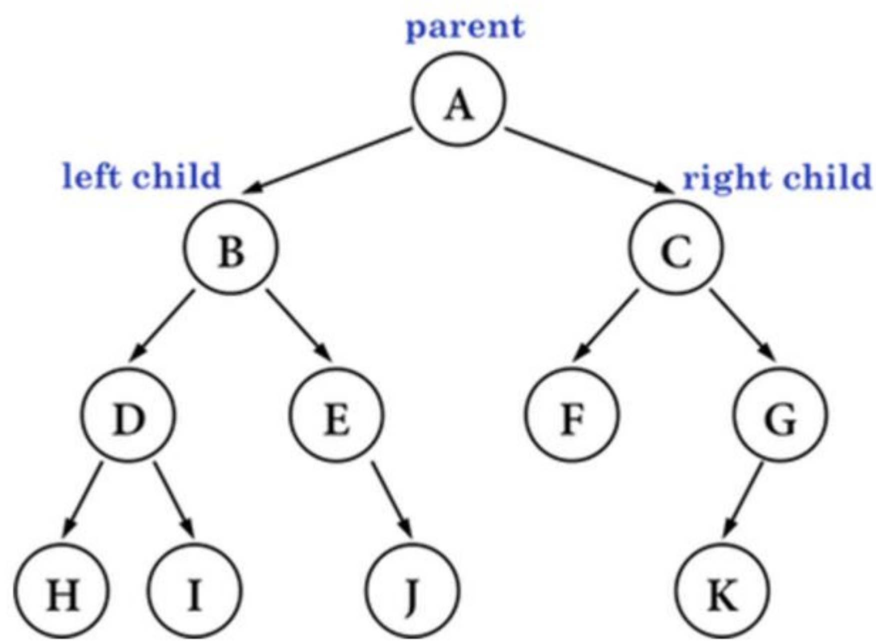
12.Binary Search Trees

Yu-Shuen Wang, CS, NCTU

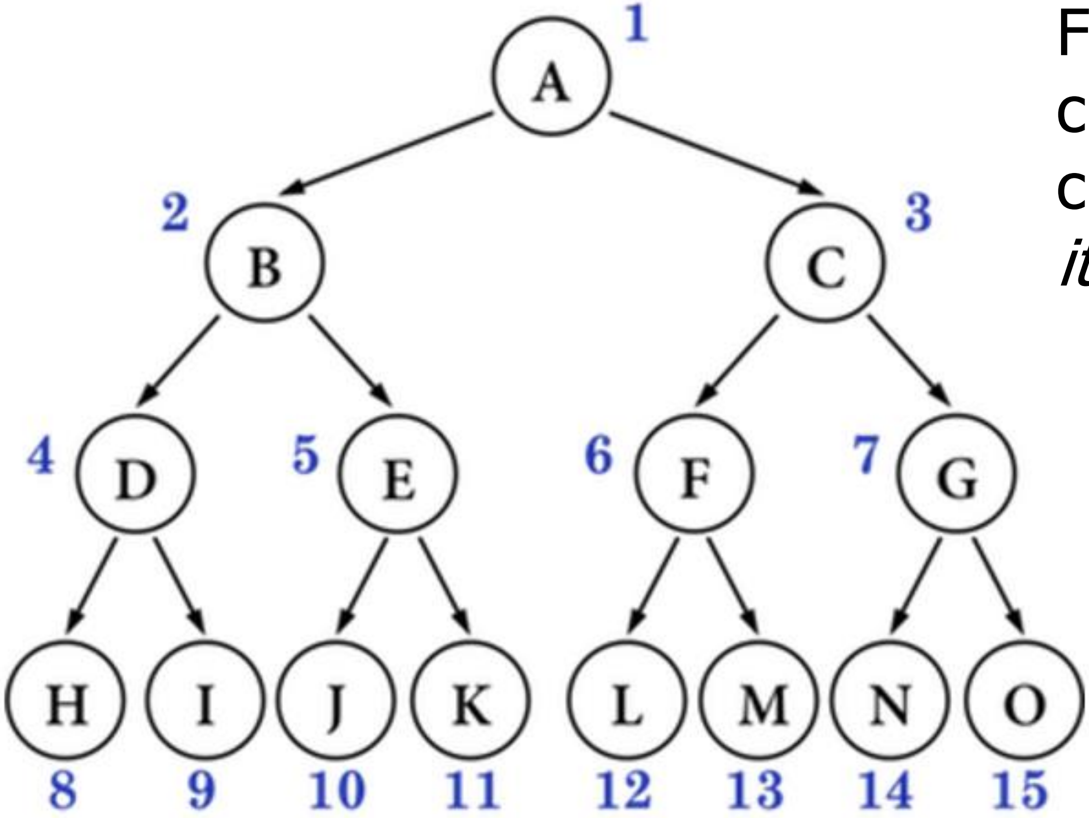
Thanks the images from http://alrightchiu.github.io/SecondRound/mu-lu-yan-suan-fa-yu-zi-liao-jie-gou.html

Non-binary and binary trees



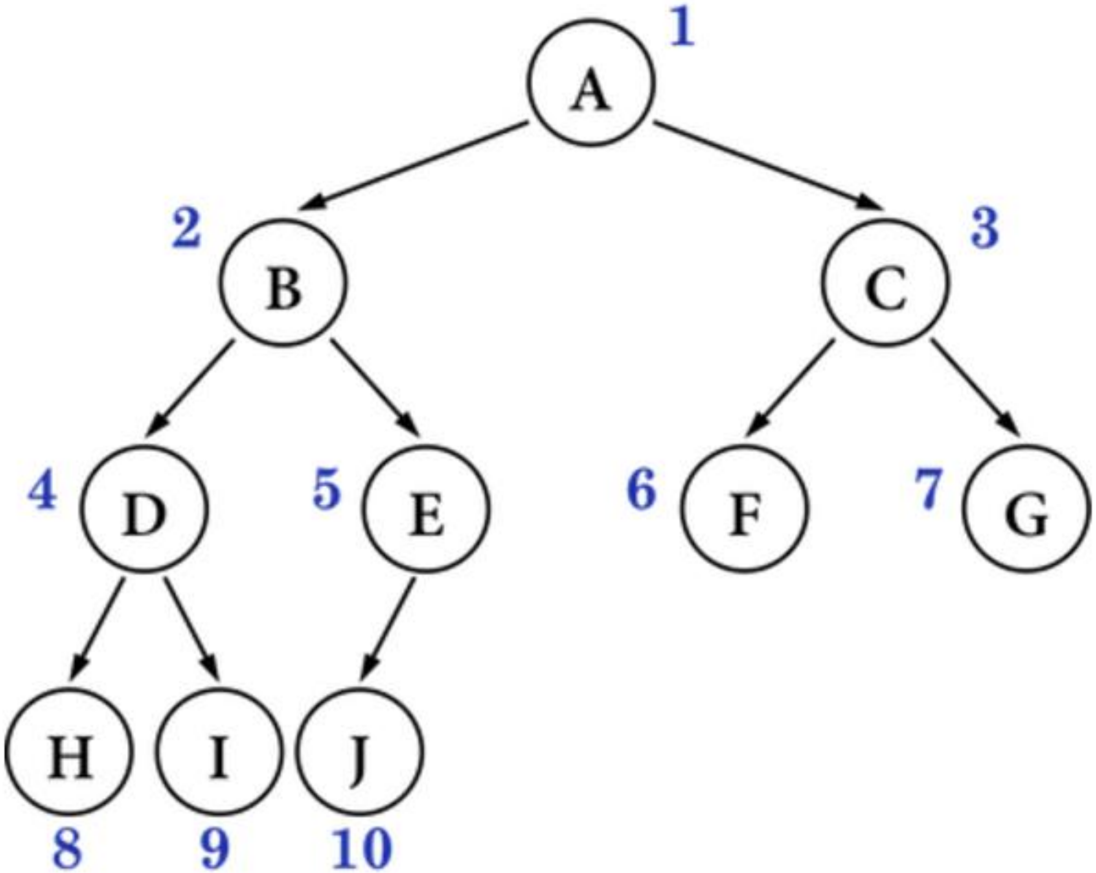


Full binary tree



For a node i, it's left child index is $2i$, it's right child index is $2i+1$, *and it's parent index is* $\lfloor i/2 \rfloor$

Complete binary tree

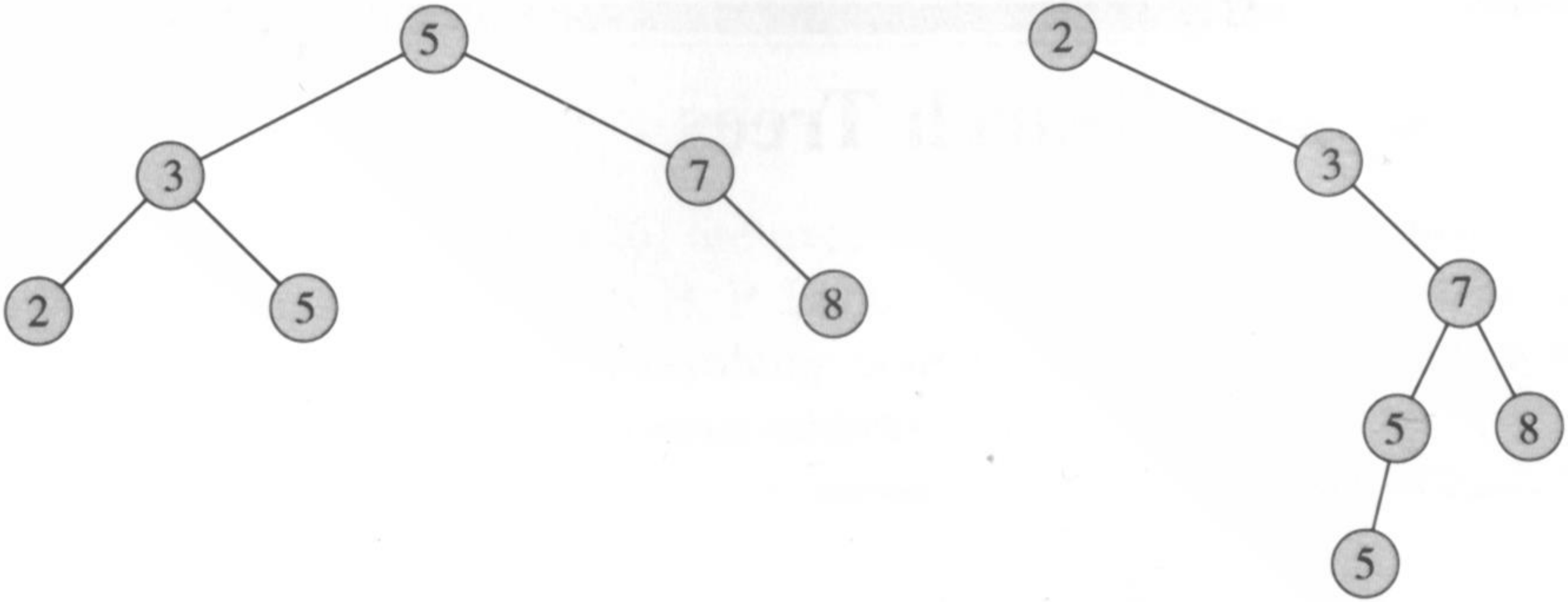


12.1 What is a binary search tree?

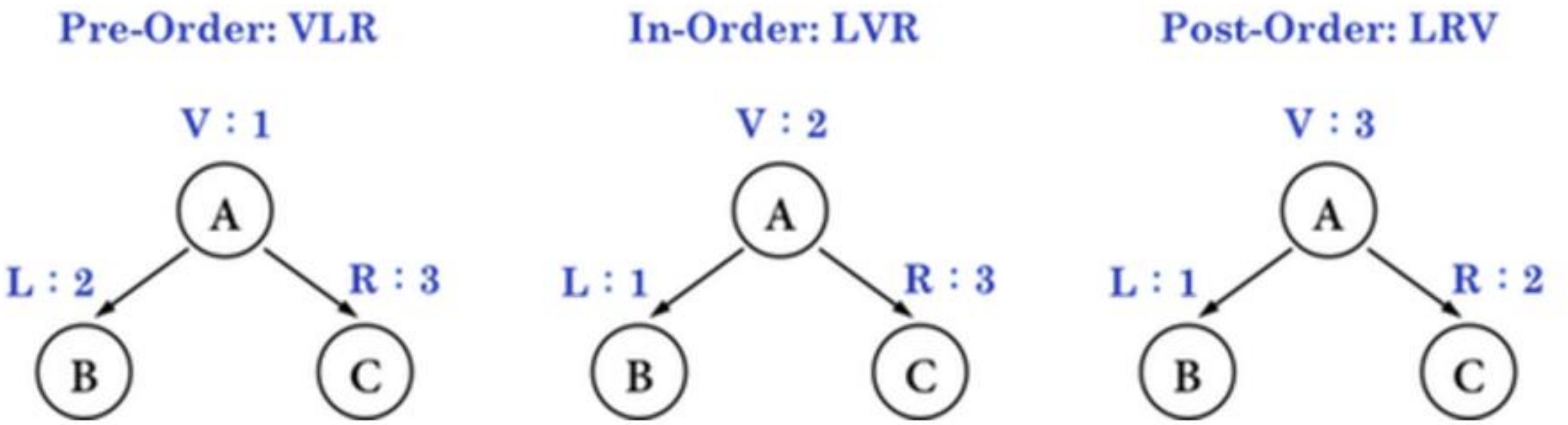
- ***Binary-search property***:

  Let x be a 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[x] $\leq$ key[y].

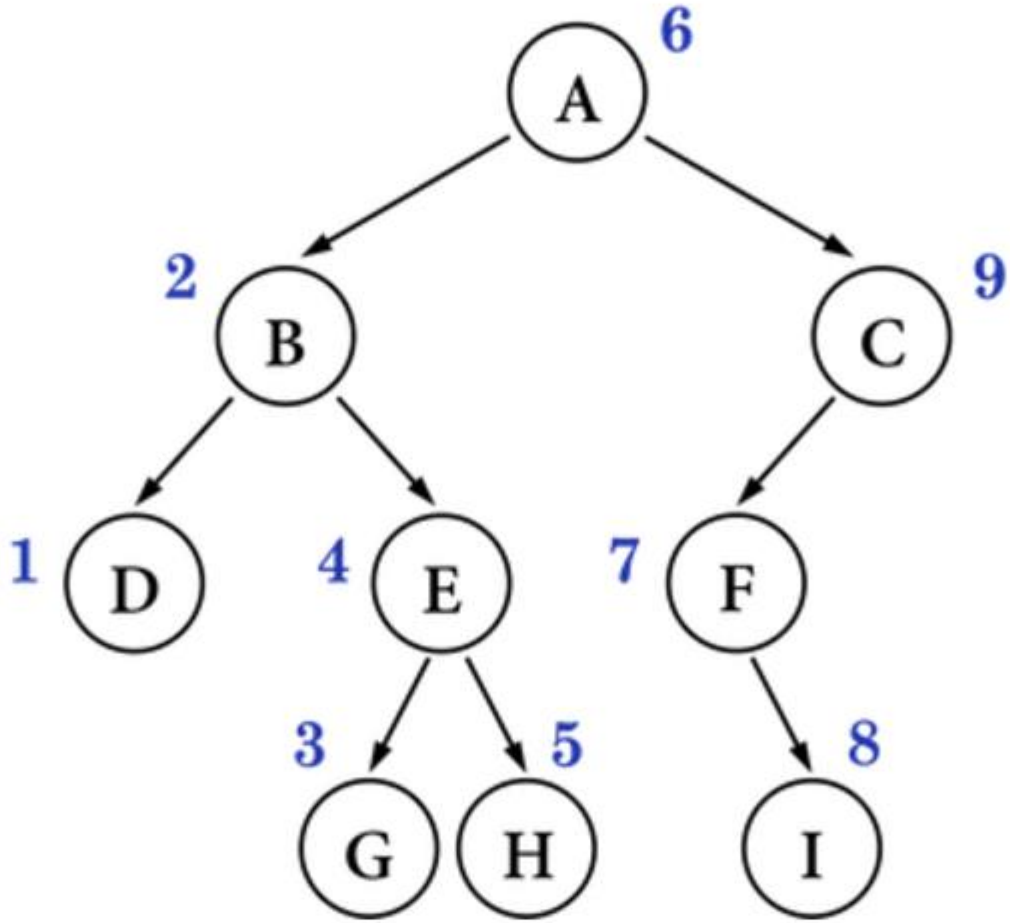
Binary search Tree



Tree traversal



Inorder tree walk (LVR)



Inorder tree walk

INORDER_TREE_WALK(x)

1 **if** $x \neq nil$

2 **then** INORDER_TREE_WALK(*left*[x])

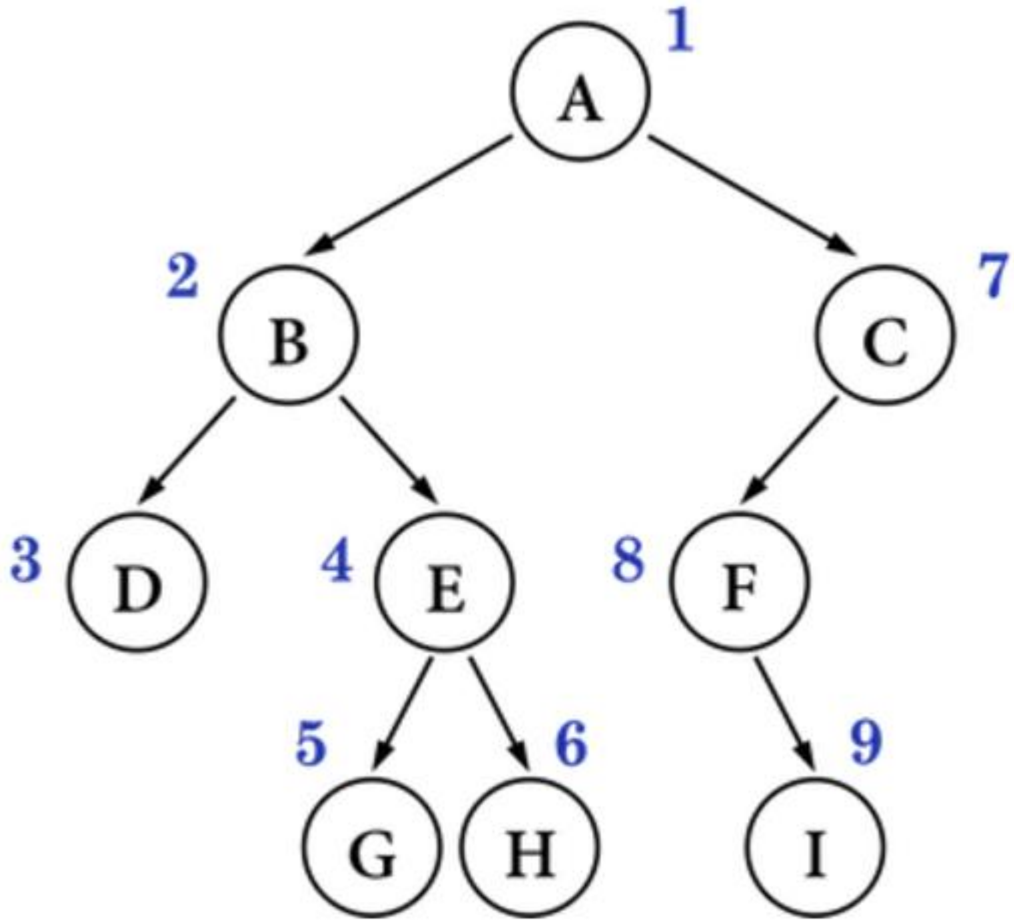
3 print *key*[x]

4 INORDER_TREE_WALK(*right*[x])

Theorem 12.1

If x is the root of an n-node subtree, then the call INORDER-TREE-WALK(x) takes $\Theta(n)$ time.

Preorder tree walk (VLR)



Preorder tree walk (VLR)

PREORDER_TREE_WALK(x)

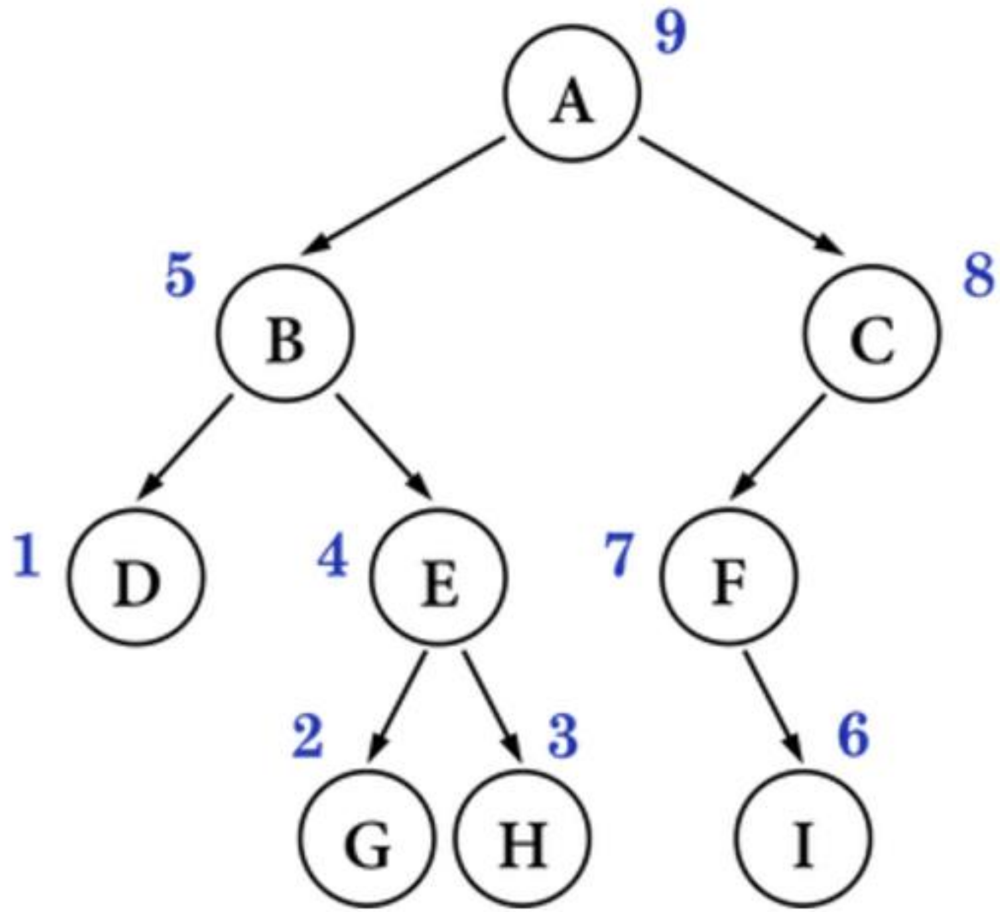
1 print $key[x]$

2 **if** $x \neq nil$

3 **then** PREORDER_TREE_WALK($left[x]$)

4 PREORDER_TREE_WALK($right[x]$)

Postorder tree walk (LRV)



Postorder tree walk (LRV)

POSTORDER_TREE_WALK(*x*)

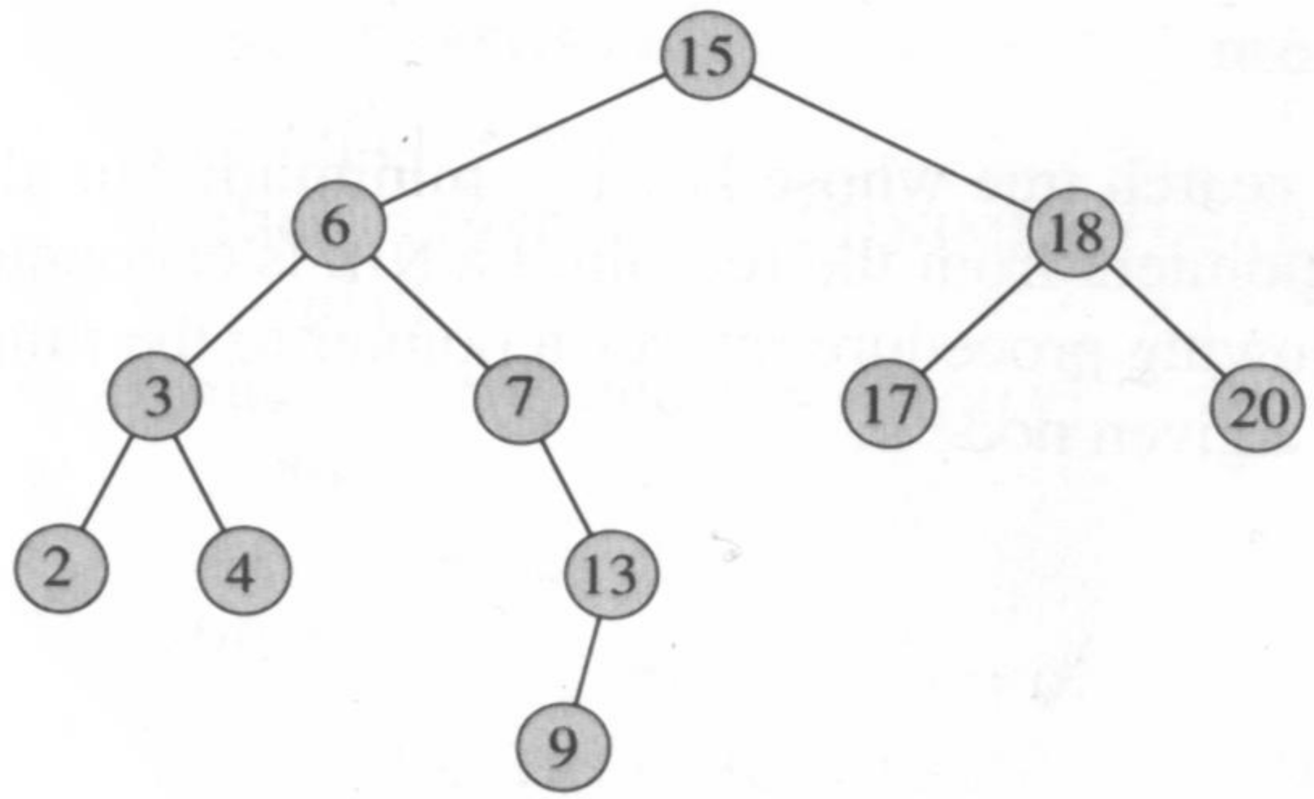
1 **if** $x \neq nil$

2 **then** POSTORDER_TREE_WALK(*left*[*x*])

3 POSTORDER_TREE_WALK(*right*[*x*])

4 print *key*[*x*]

12.2 Querying a binary search tree



TREE_SEARCH(x,k)

TREE_SEARCH(x,k)

1 **if** $x = nil$ **or** $k = key[x]$

2 **then return** x

3 **if** $k < key[x]$

4 **then return** TREE_SEARCH(left[x],k)

5 **else return** TREE_SEARCH(right[x],k)

ITERATIVE_SEARCH(x,k)

ITERATIVE_SEARCH(x,k)

1 **While** $x \neq nil$ **or** $k \neq key[x]$

2 **do if** $k < key[x]$

3 **then** $x \leftarrow left[x]$

4 **else** $x \leftarrow right[x]$

5 **return** x

MAXIMUM and MINIMUM

- TREE_MINIMUM(x)

```
while left[x] ≠ NIL
    do x ← left[x]
return x
```

- TREE_MAXIMUM(x)

```
while right[x] ≠ NIL
    do x ← right[x]
return x
```

SUCCESSOR and PREDECESSOR

TREE_SUCCESSOR

if $right[x] \neq nil$

then return $\text{TREE\_MINIMUM}(right[x])$

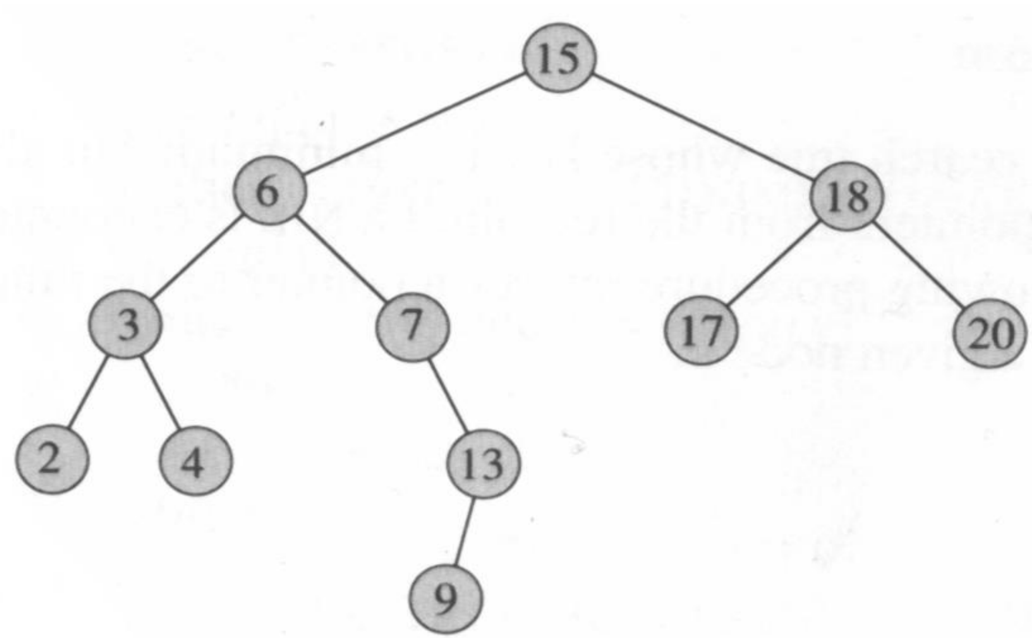
$y \leftarrow p[x]$

while $y \neq nil$ **and** $x = right[y]$

do $x \leftarrow y$

$y \leftarrow p[y]$

return y



Quiz:

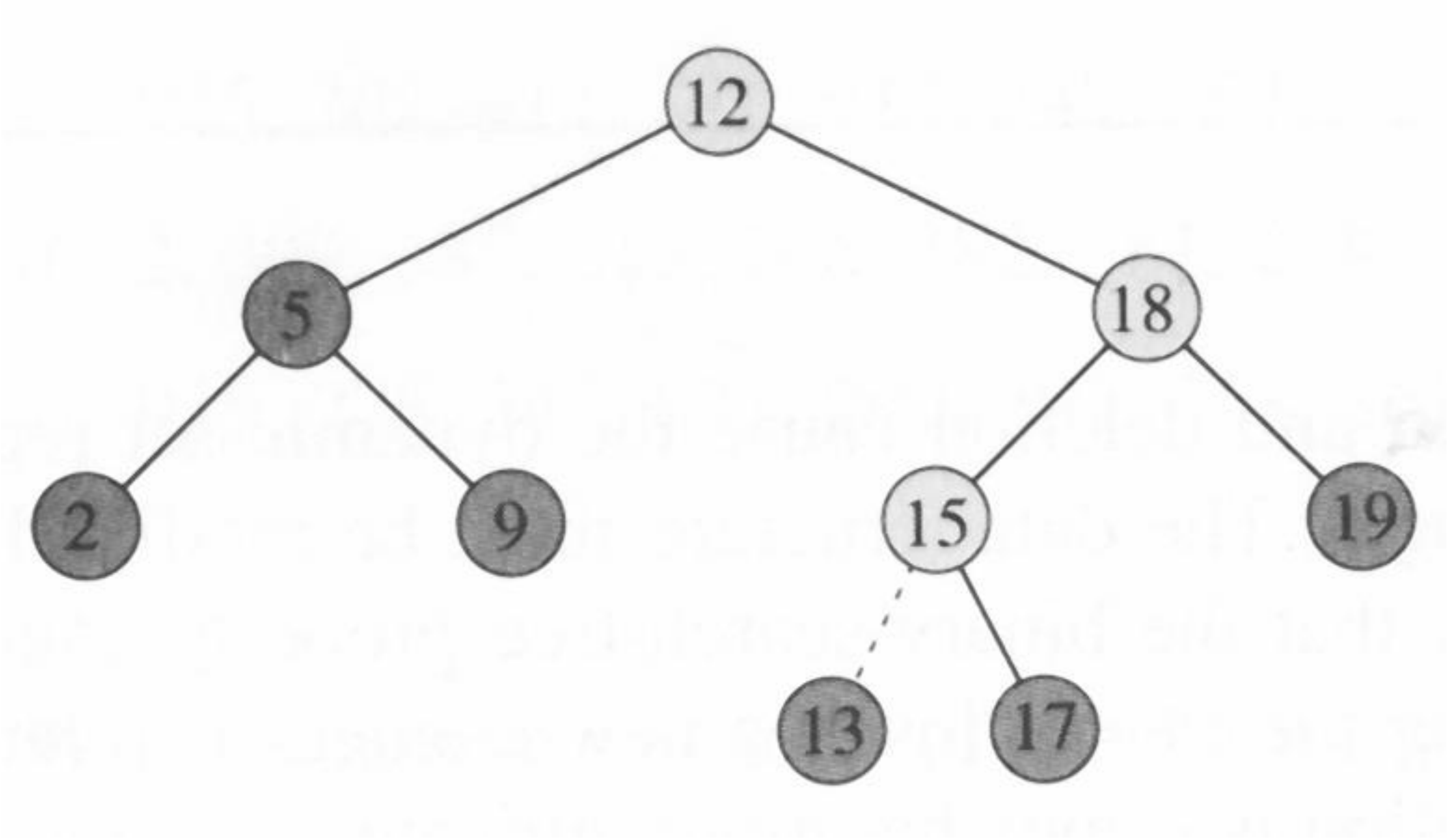
Write the sudo-codes of TREE_PREDECESSOR

***Theorem* 12.2**

- The dynamic-set operations, SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR can be made to run in O(h) time on a binary search tree of height h.

12.3 Insertion and deletion

Inserting an item with key 13 into a binary search tree



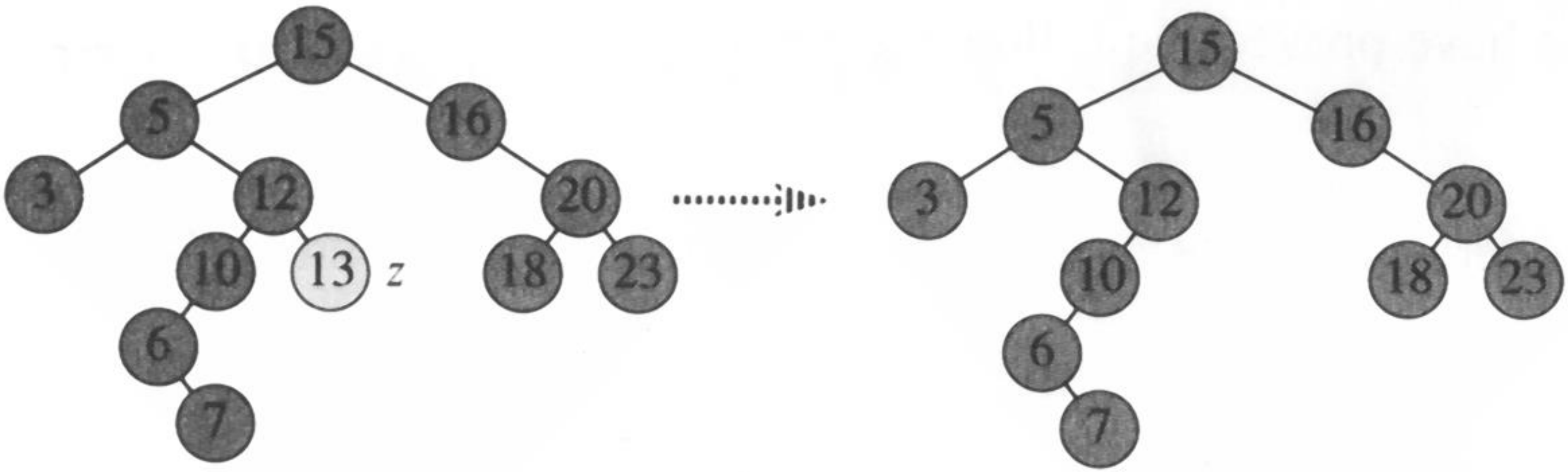
Insertion

Tree-Insert(T,z)

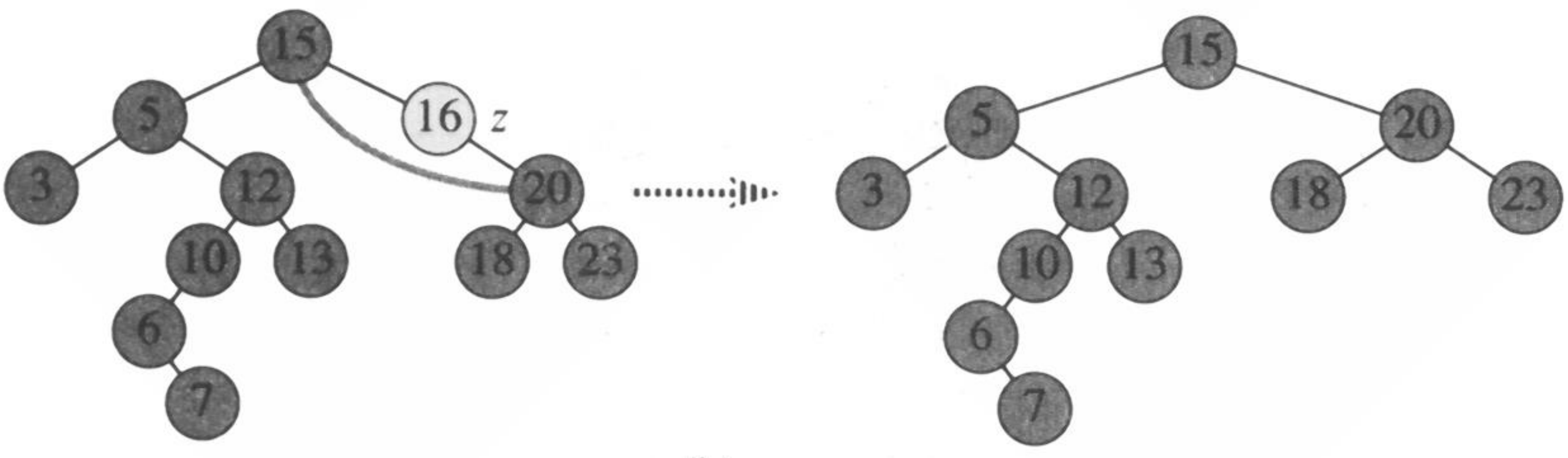
```
y ← NIL
x ← root[T]
while x ≠ NIL                        ▶ unsuccessful search
    do y ← x
        if key[z] < key[x]
            then x ← left[x]
            else x ← right[x]
p[z] ← y
```

```
if y = NIL                          ▷ tree T was empty
    then root[T] ← z
    else if key[z] < key[y]         ▷ link to child
        then left[y] ← z
        else right[y] ← z
```

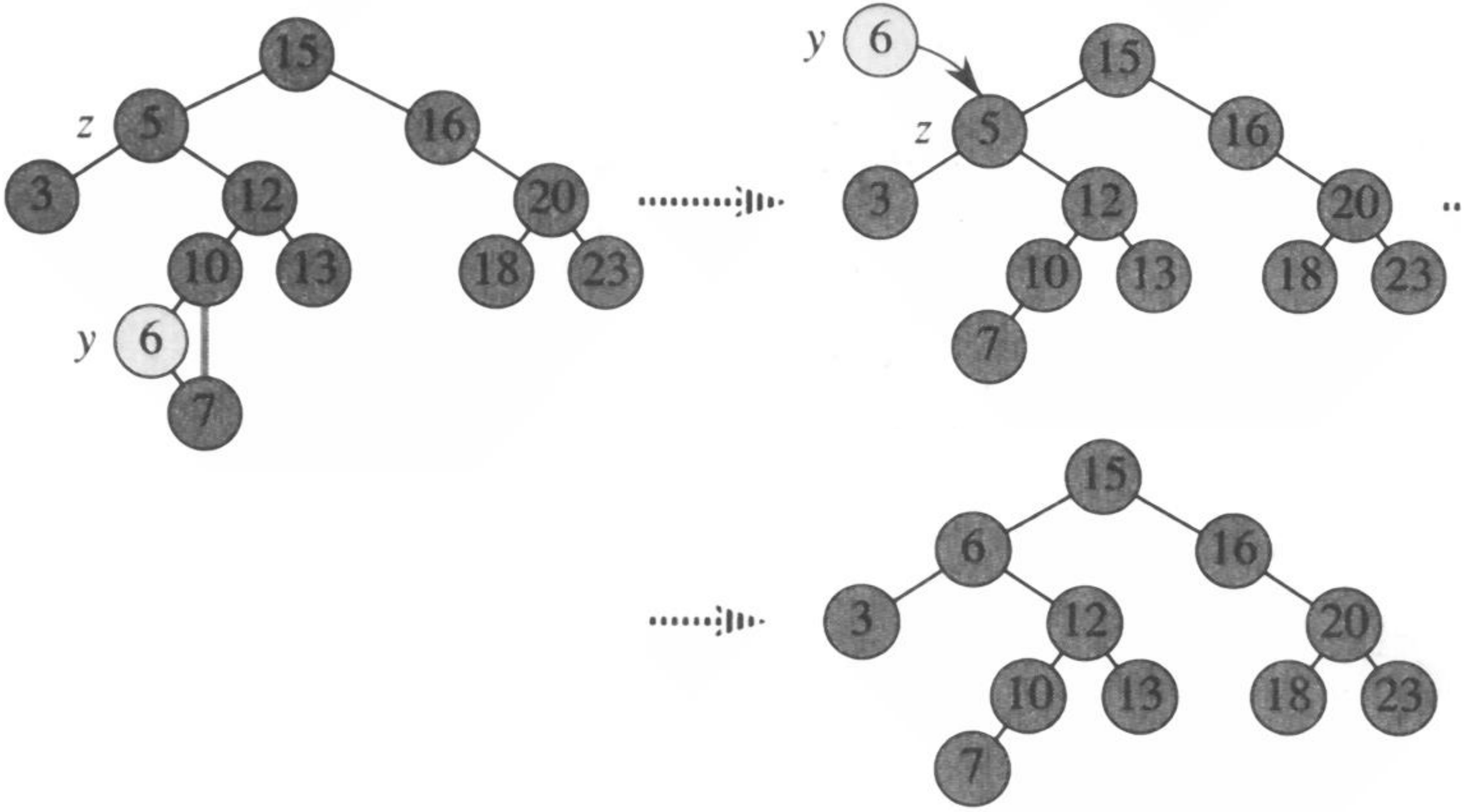
Deletion: z has no children



Deletion: z has only one child

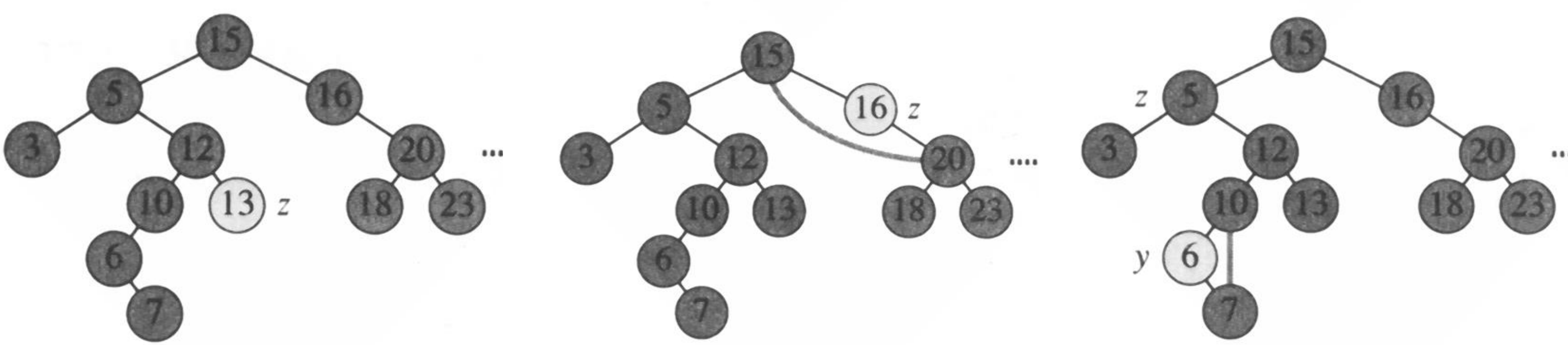


Deletion: z has two children

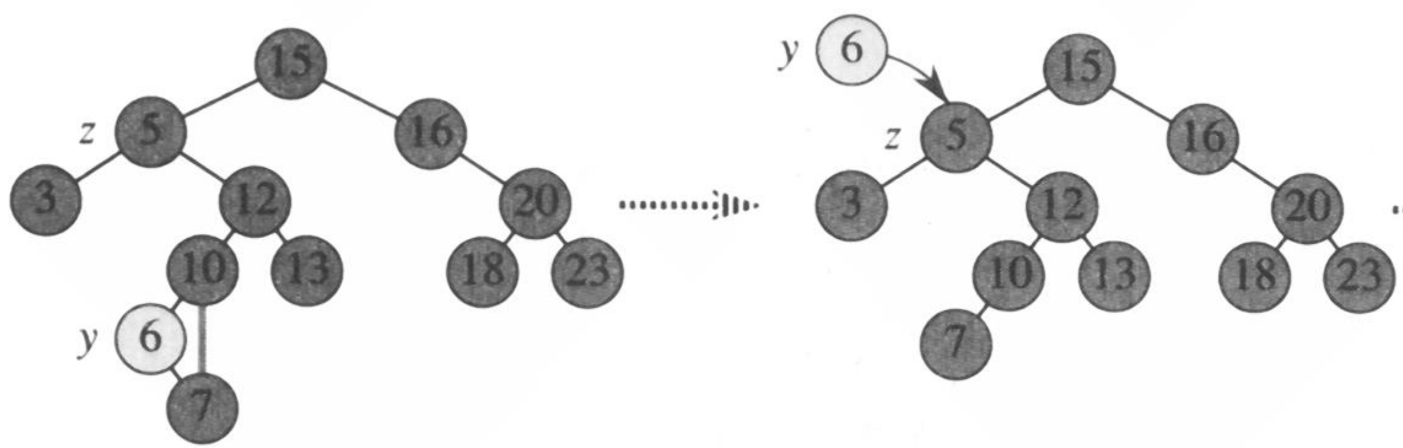


Tree-Delete(T,z)

```
if left[z] = NIL or right[z] = NIL          ▶ one or no child
    then y ← z
    else y ← Tree-Successor(z)               ▶ two children
if left[y] ≠ NIL                  ▶ set x to be y's child
    then x ← left[y]
    else x ← right[y]
if x ≠ NIL                        ▶ if at least one child
    then p[x] ← p[y]              ▶ connect the child to its parent
```



```
if p[y] = NIL                              ▶ y is root
   then root[T] ← x                        ▶ y will be deleted, x becomes root
   else if y = left[p[y]]
           then left[p[y]] ← x             ▶ connect parent to child
           else right[p[y]] ← x
if y ≠ z
   then key[z] ← key[y]
        copy y's satellite data into z
return y
```



Theorem 12.3

- The dynamic-set operations, INSERT and DELETE can be made to run in O(h) time on a binary search tree of height h.