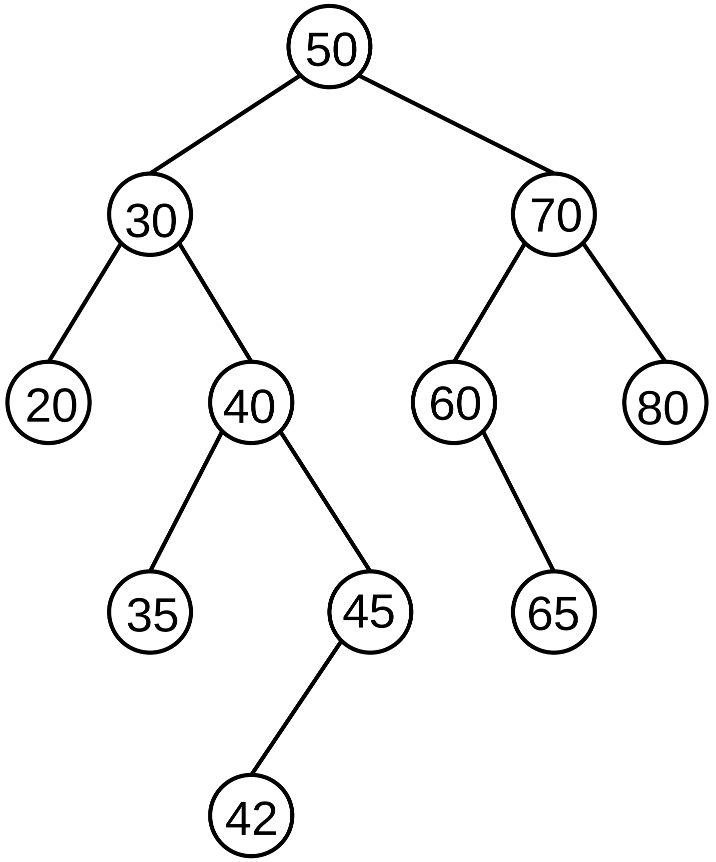
# COT 6405
# ANALYSIS OF ALGORITHMS

## Binary Search Trees - Review

Computer & Electrical Engineering and Computer Science Department
Florida Atlantic University

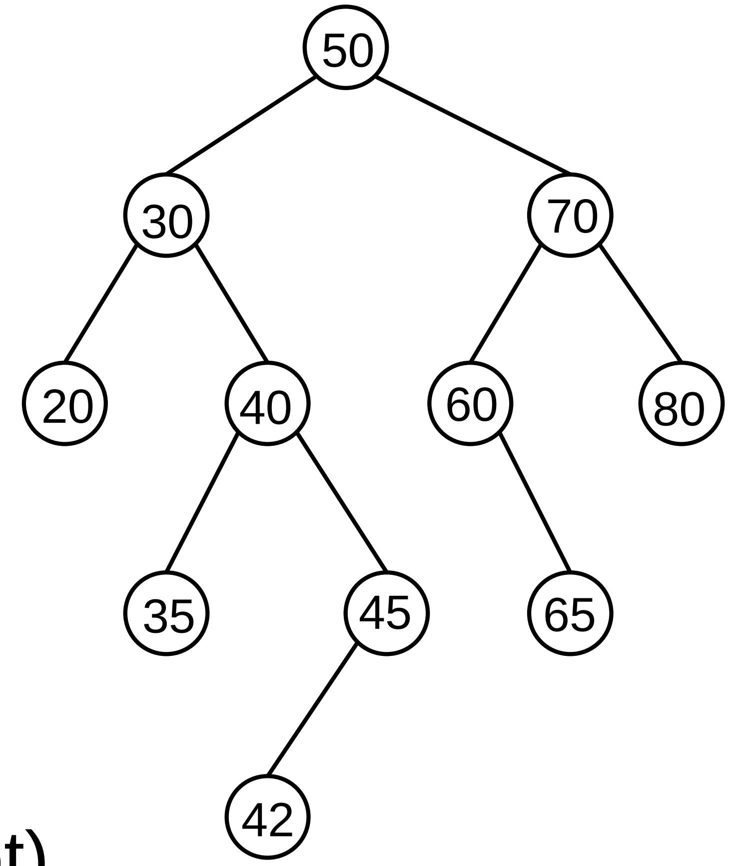
# Binary Search Trees (BST)

- tree T implementation:
  - T.root
  - each node is an object with fields:
    - key (and satellite data)
    - pointers: left, right, p
- the keys of a BST must satisfy the BST property: for any node x
  - if y is a node in x's left subtree then y.key $\leq$ x.key
  - if y is a node in x's right subtree then x.key $\leq$ y.key
- what is the maximum height h ?
  - maximum height h = n-1, therefore h = O(n)



# BST-walk: prints all the keys in the tree

- Inorder tree walk:
  - print x's left subtree
  - print node x's key
  - print x's right subtree



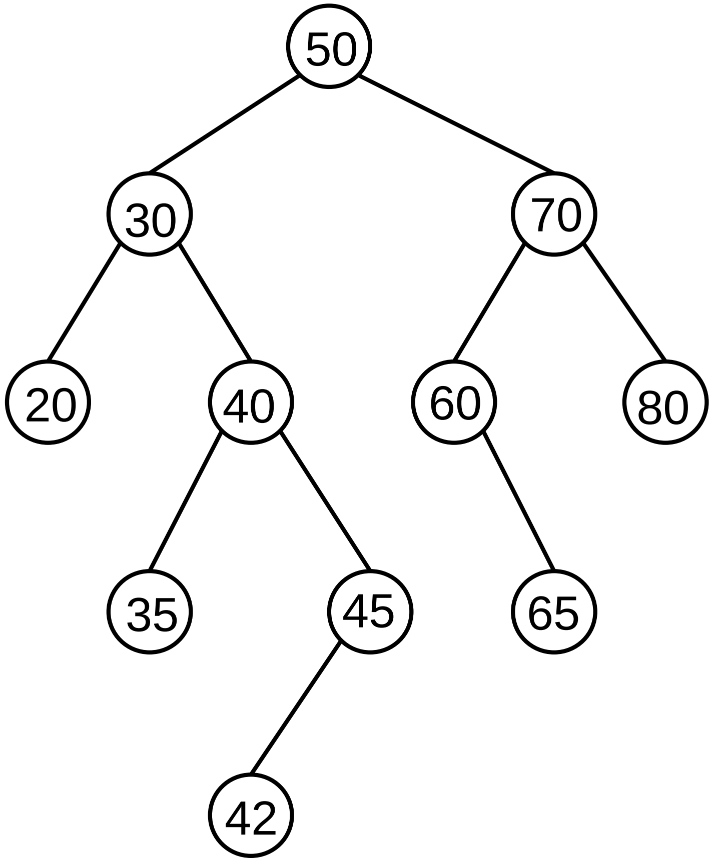

```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```

- Initial call: INORDER-TREE-WALK (T.root)
- RT = $\Theta(n)$
- example: 20, 30, 35, 40, 42, 45, 50, 60, 65, 70, 80.
- **Property:** *prints the keys in sorted order*

# BST-walk

- Preorder tree walk:
  - print node x's key
  - print x's left subtree
  - print x's right subtree

- Postorder tree walk:
  - print x's left subtree
  - print x's right subtree
  - print node x's key



# Querying a BST

All operations have the worst case RT = $\Theta(h)$

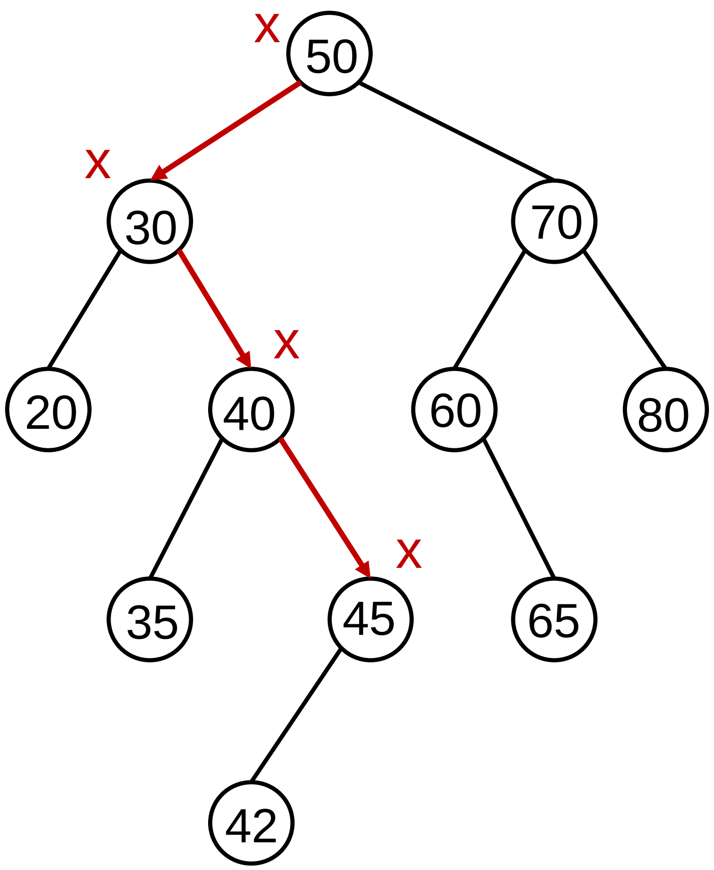
- search
- minimum
- maximum
- successor
- predecessor

# 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)
```

- Initial call: TREE-SEARCH (T.root, k)
- RT = O(h)

TREE-SEARCH(T.root, 45)



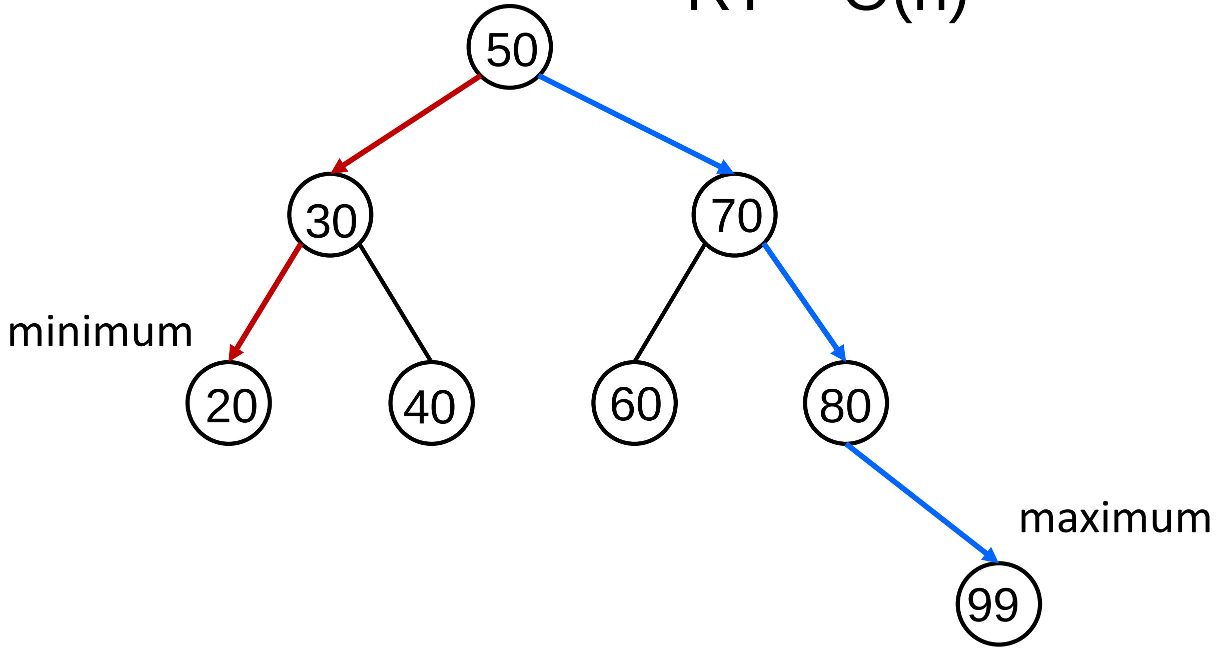
# Minimum & Maximum

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

- Initial call: TREE-MINIMUM (T.root)
- RT = O(h)

```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```

- Initial call: TREE-MAXIMUM (T.root)
- RT = O(h)



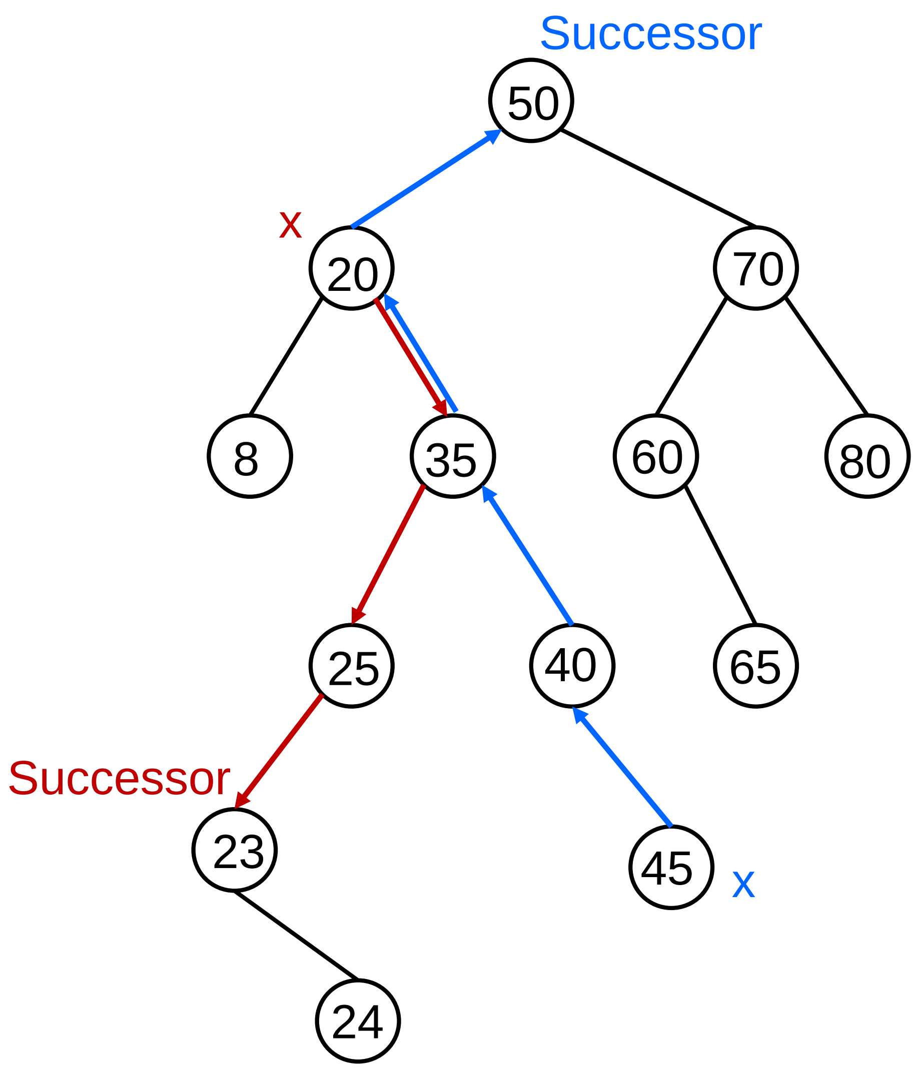
# Successor

- Assuming the keys are distinct,

the successor of x is the node y

with the smallest key y.key ≥ x.key

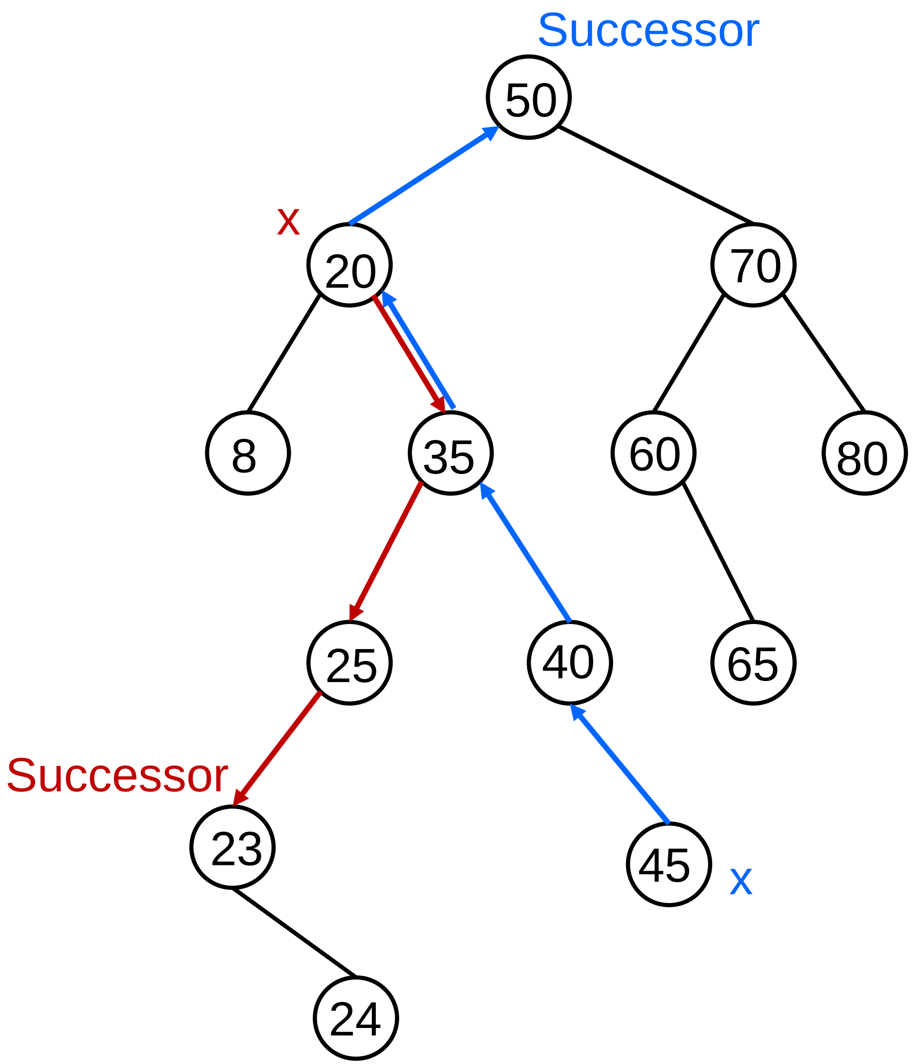
- Successor of x
  - if x.right ≠ NIL, then the successor is the TREE-MINIMUM(x.right)
  - if x.right = NIL, then the successor is the first ancestor larger than x



# Successor

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

- RT = O(h)



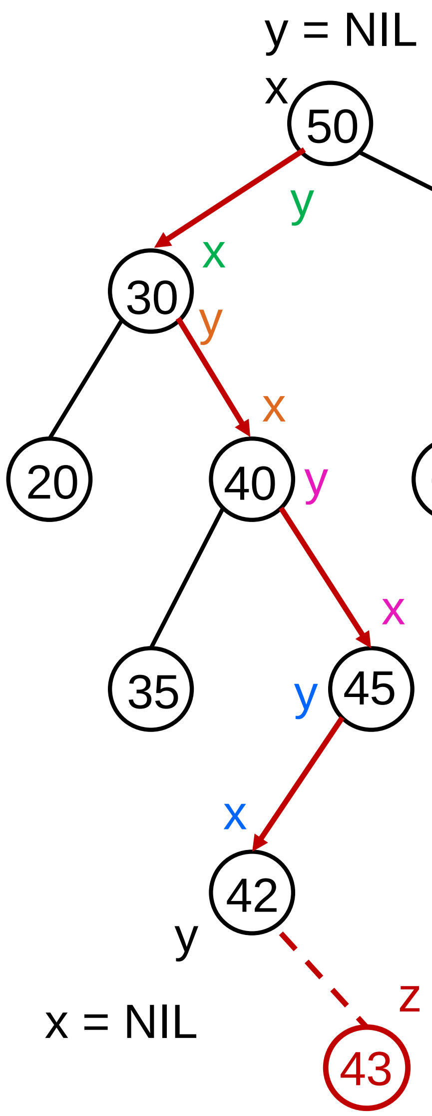
# Insert operation

- To insert a new key v, the procedure takes as argument a new node z with z.key = v, z.left = NIL, and z.right = NIL

```
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
```

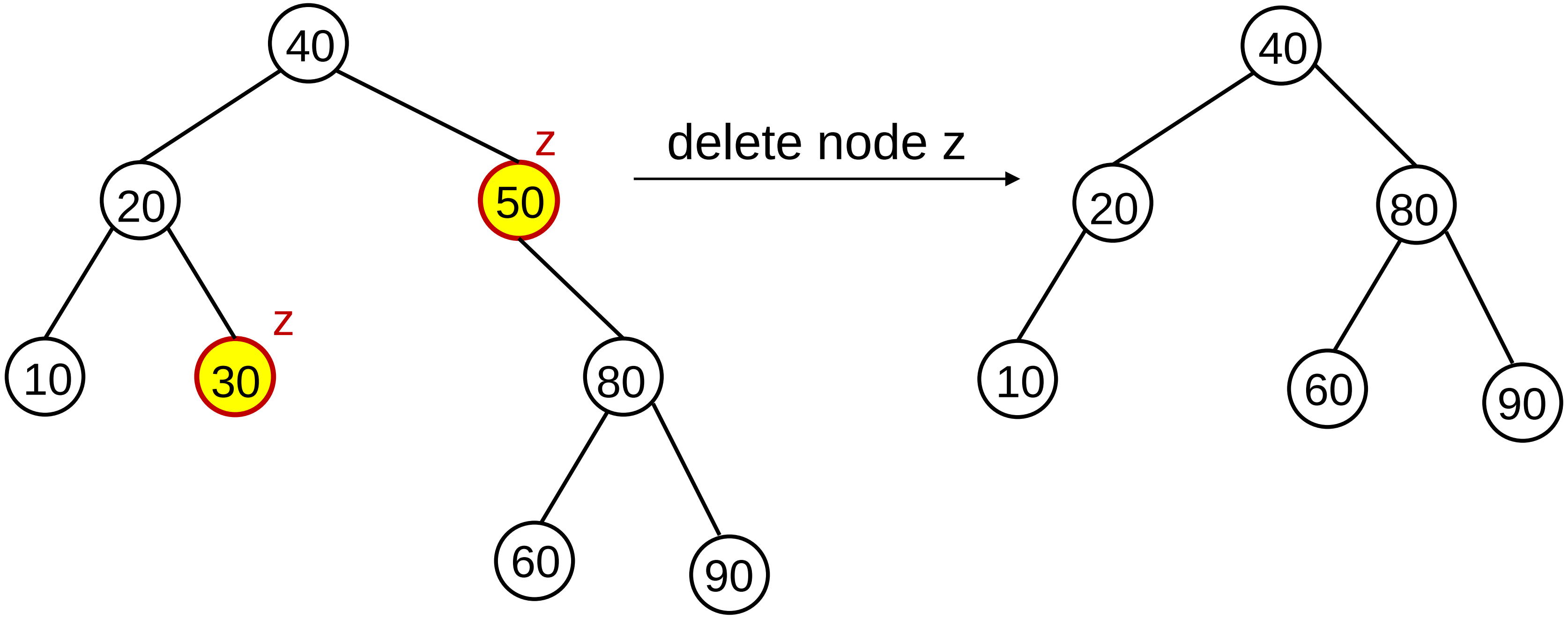
RT = O(h)

Insert key 43



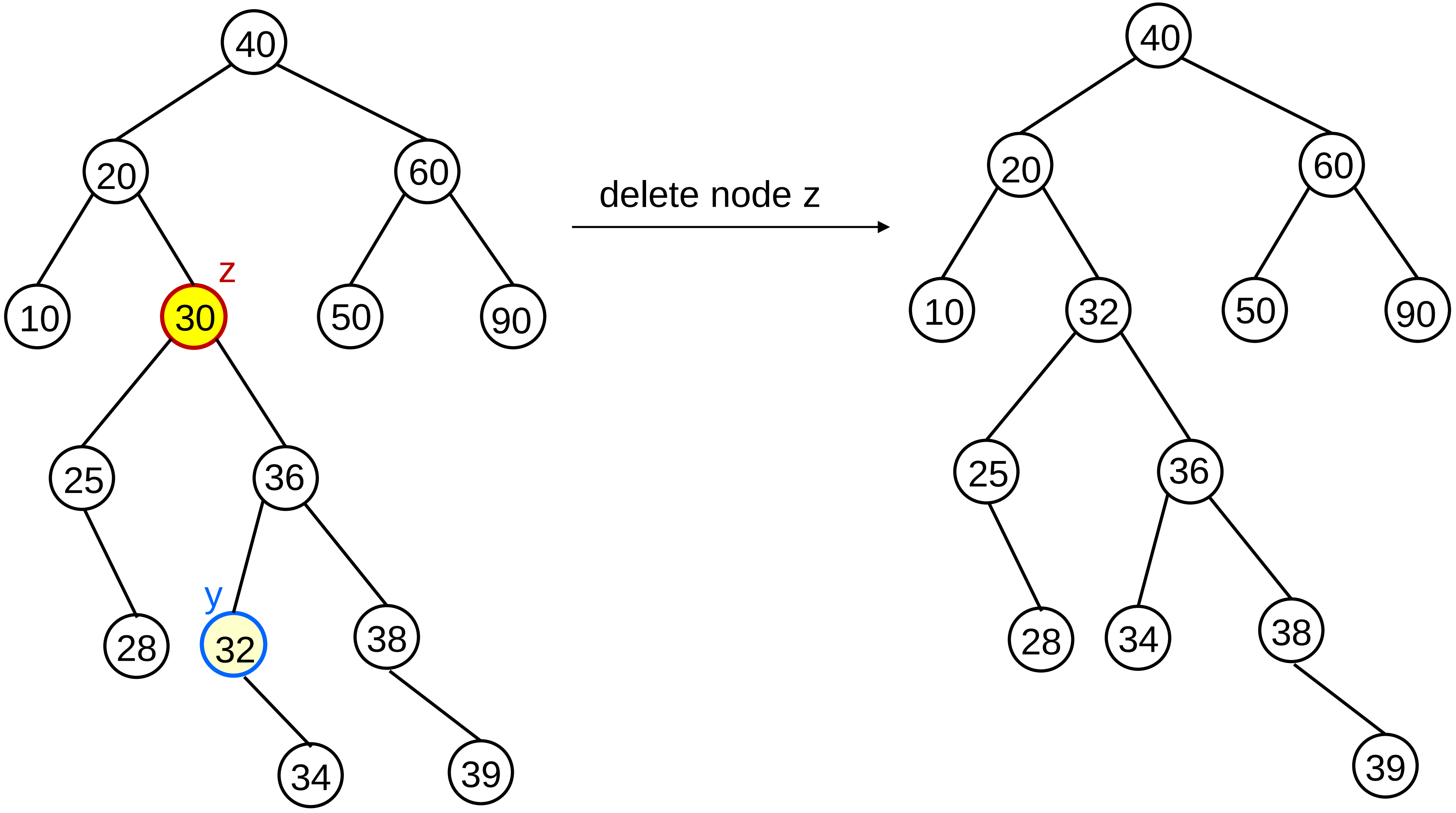
# Delete operation

- z has no children
- z has one child



# Delete operation

- z has two children



# TRANSPLANT operation

- replace the subtree rooted at node u with the subtree rooted at node v. Node u's parent becomes node v's parent.

```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

RT = $\Theta(1)$

# Delete operation

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

RT = O(h)