CMPN302: Algorithms Design and Analysis

Lecture 04: Binary Search Trees

Ahmed Hamdy

Computer Engineering Department

Cairo University

Fall 2017

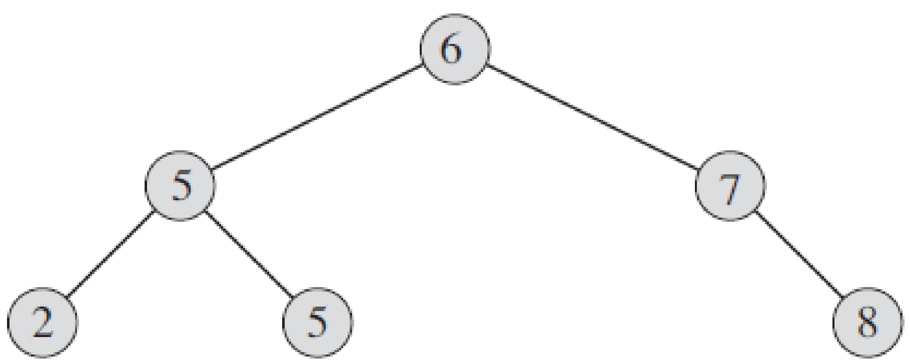
Binary search trees (BST)

- For all nodes y in left subtree of x,

$$y.key \leq x.key$$

- For all nodes y in right subtree of x,

$$y.key \geq x.key$$



BST operations

- Operations:
  - Search
  - Minimum
  - Maximum
  - Predecessor
  - Successor
  - Insert
  - Delete
- Complexity: $O(h)$
  - Complete/balanced tree: $O(\log n)$
  - Linear chain: $O(n)$

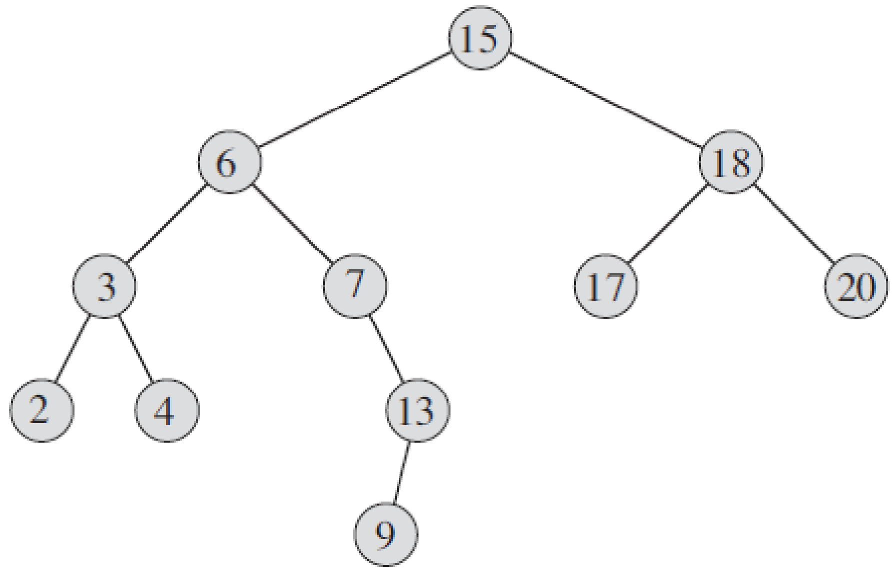
BST traversal

- Inorder tree walk:

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

- Preorder tree walk: visit root first
- Postorder tree walk: visit root last
- Complexity: $\Theta(n)$

BST 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)
```

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

BST operations

- Minimum

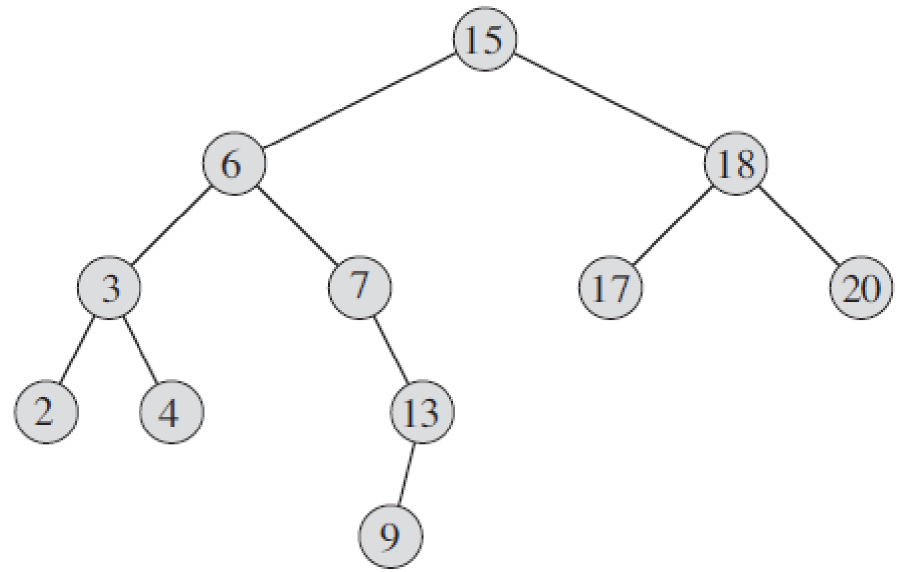
```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

- Maximum

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

BST operations

- Successor:
  - Successor(node15)
  - Successor(node13)




```
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
```

BST operations

- Insert

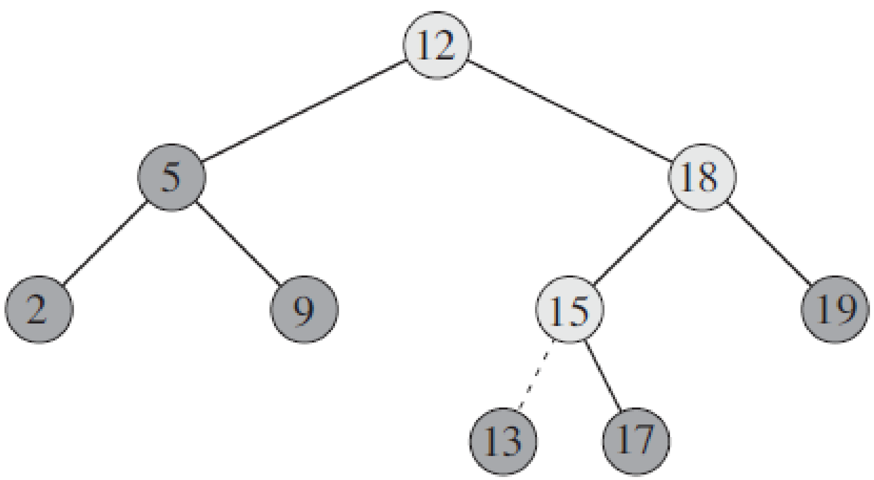


Figure 12.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

BST operations

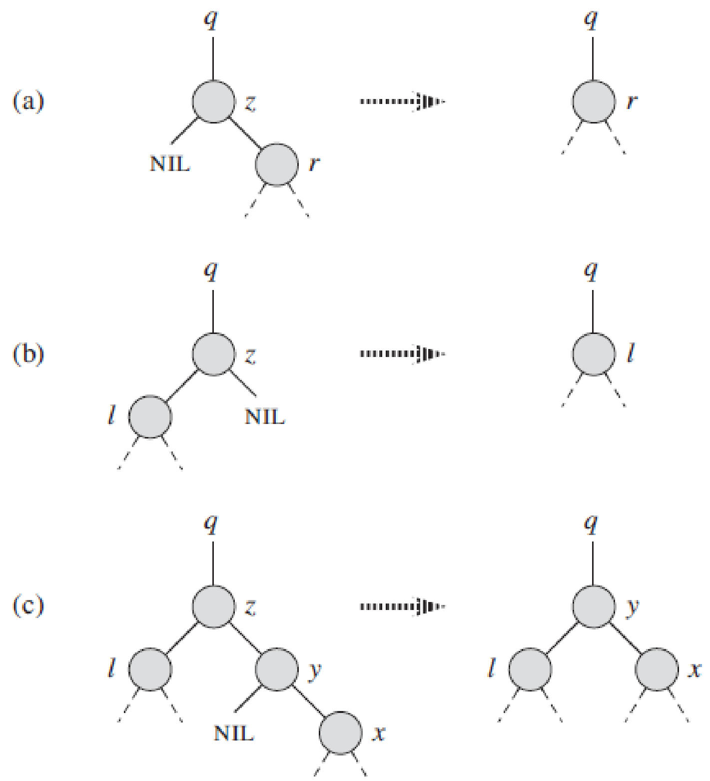
- Insert

```
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
```

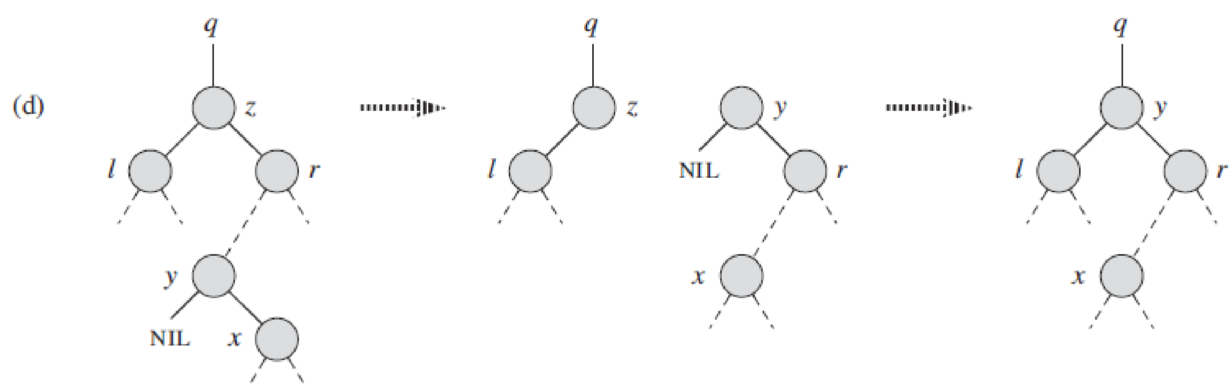
BST operations

- Delete

```
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
```




```
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
```



BST building

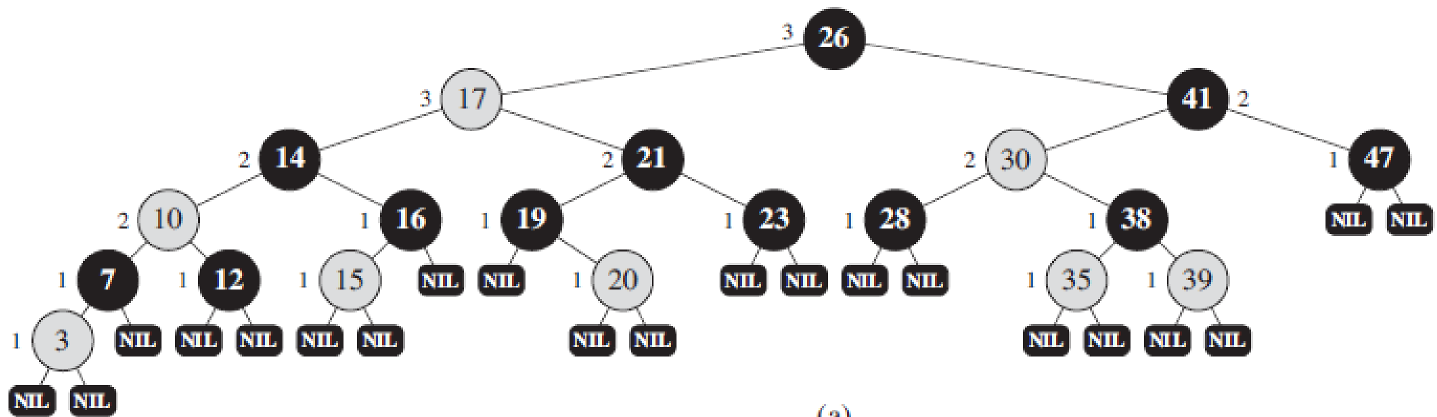
- Linked-list trees cost $O(n)$ for operations
  - Caused by insertion of sorted elements
- To minimize worst case: randomized insertion
  - Average case $O(\log n)$
- To have $O(\log n)$ height in the worst case
  - Use self-balancing trees, i.e. AVL and red-black trees

AVL trees

- Heights of the two child subtrees of any node differ by at most one

- If after insertion or deletion, the difference is more than one, then rebalancing takes place

Red-Black trees

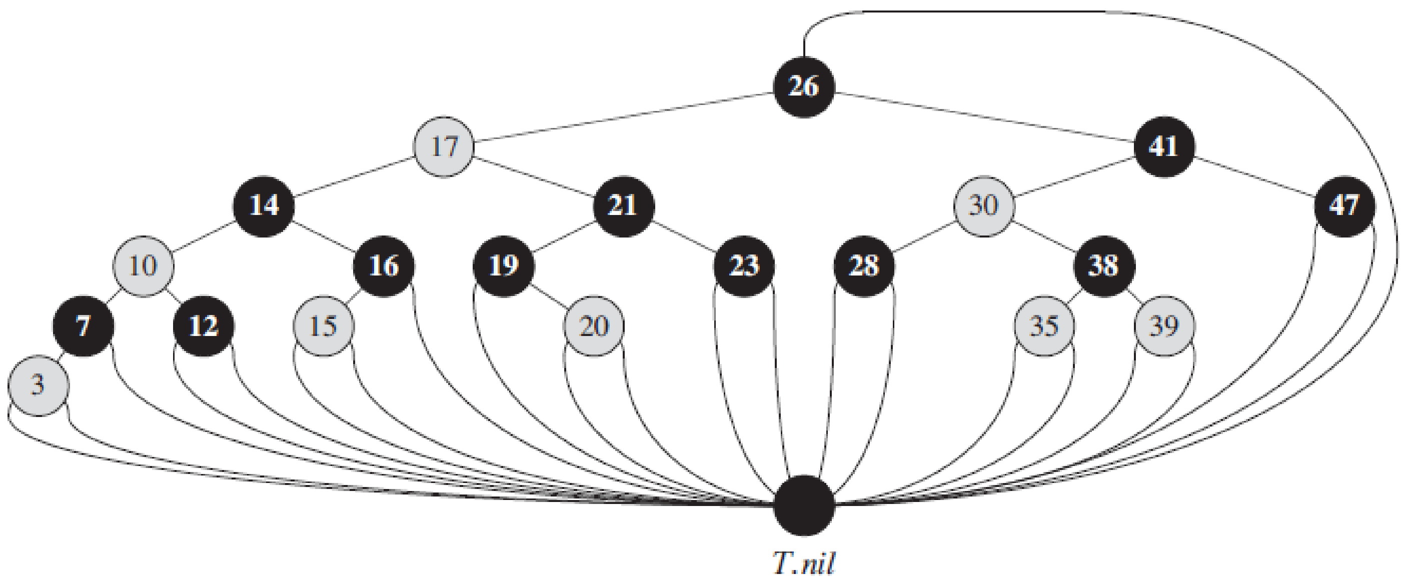


(a)

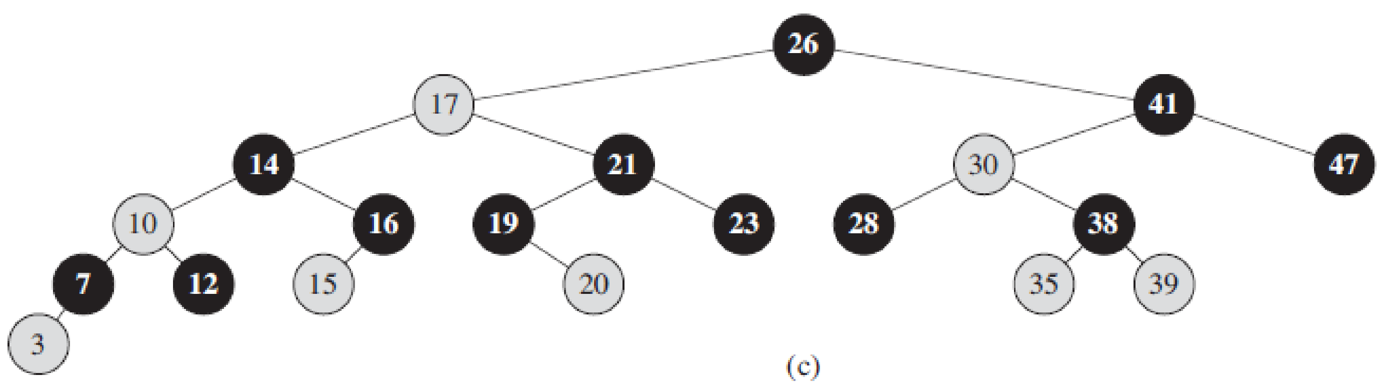
1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Lemma 13.1
A red-black tree with n internal nodes has height at most $2\lg(n+1)$.

Red-Black trees



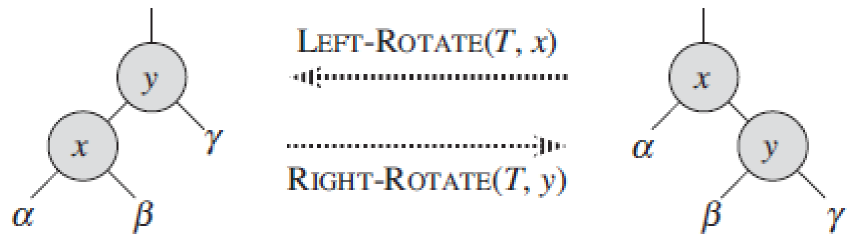
(b)



(c)

Red-Black trees

- Rotations

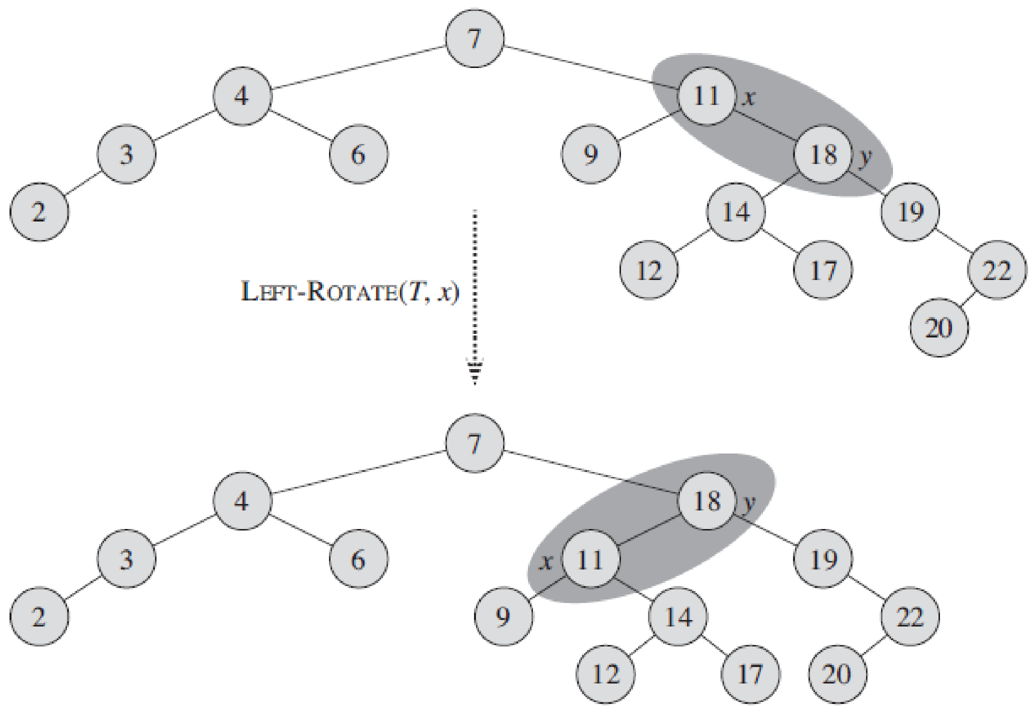


LEFT-ROTATE(T, x)

```
y = x.right                 // set y
x.right = y.left            // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                   // link x's parent to y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x                  // put x on y's left
x.p = y
```

Red-Black trees

- Rotations



Red-Black trees

- Insertion

```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

Red-Black trees

- Insertion

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK                  // case 1
            y.color = BLACK                    // case 1
            z.p.p.color = RED                  // case 1
            z = z.p.p                          // case 1
        else if z == z.p.right
                z = z.p                        // case 2
                LEFT-ROTATE(T, z)              // case 2
            z.p.color = BLACK                  // case 3
            z.p.p.color = RED                  // case 3
            RIGHT-ROTATE(T, z.p.p)             // case 3
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

Red-Black trees

- Insertion