Balanced Search Trees

Data structures

Fall 2018

Balanced Binary Search Trees

- height is O(log n), where n is the number of elements in the tree
- AVL (Adelson-Velsky and Landis) trees
- red-black trees
- get, put, and remove take O(log n) time

Balanced Binary Search Trees

- Indexed AVL trees
- Indexed red-black trees
- Indexed operations also take O(log n) time

Balanced Search Trees

- weight balanced binary search trees
- 2-3 & 2-3-4 trees
- AA trees
- B-trees
- BBST
- etc.

AVL Tree

- binary tree
- for every node x, define its balance factor

  balance factor of x = height of left subtree of x

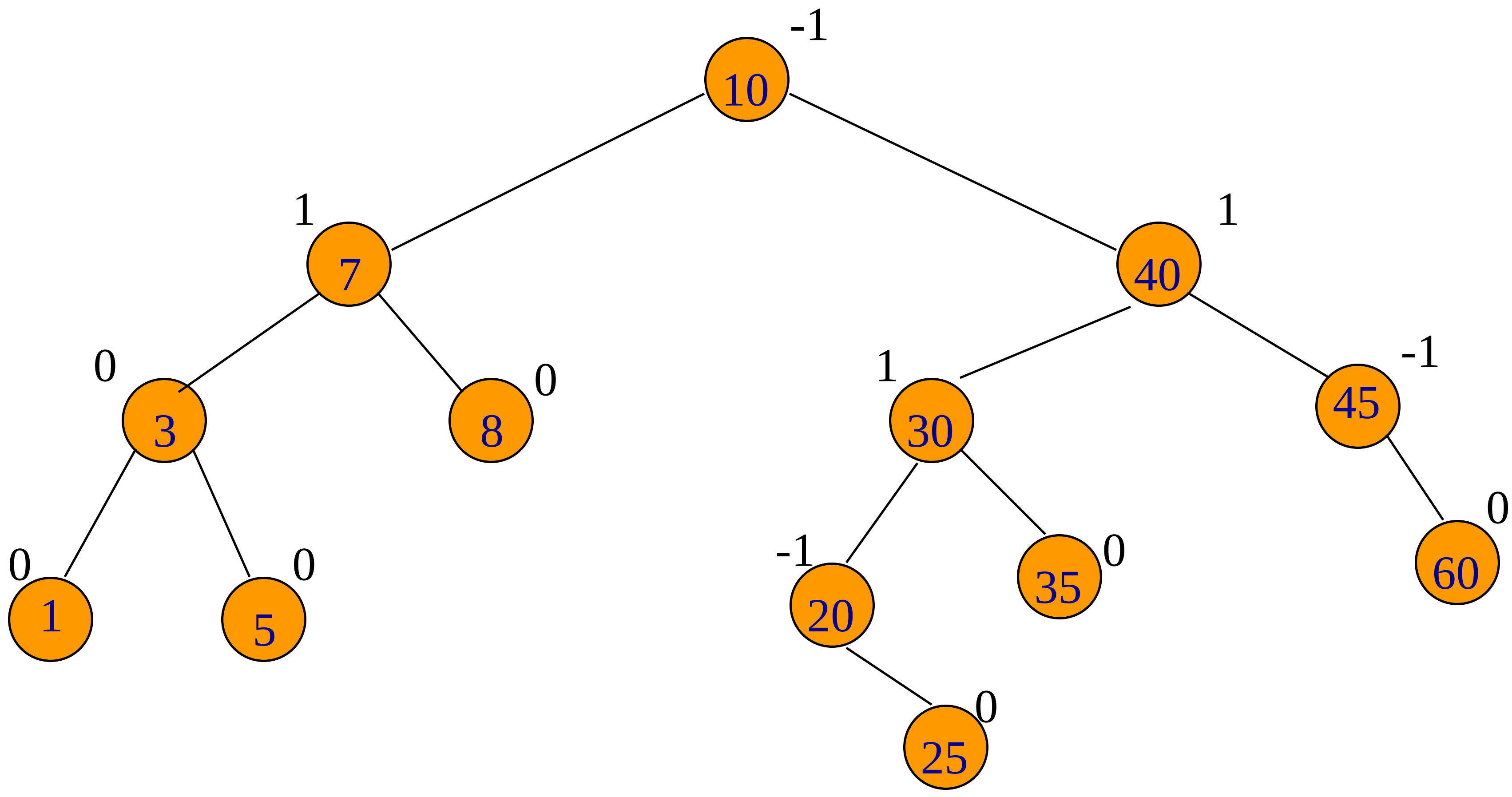
  - height of right subtree of x
- balance factor (BF) of every node x is -1, 0, or 1

Height

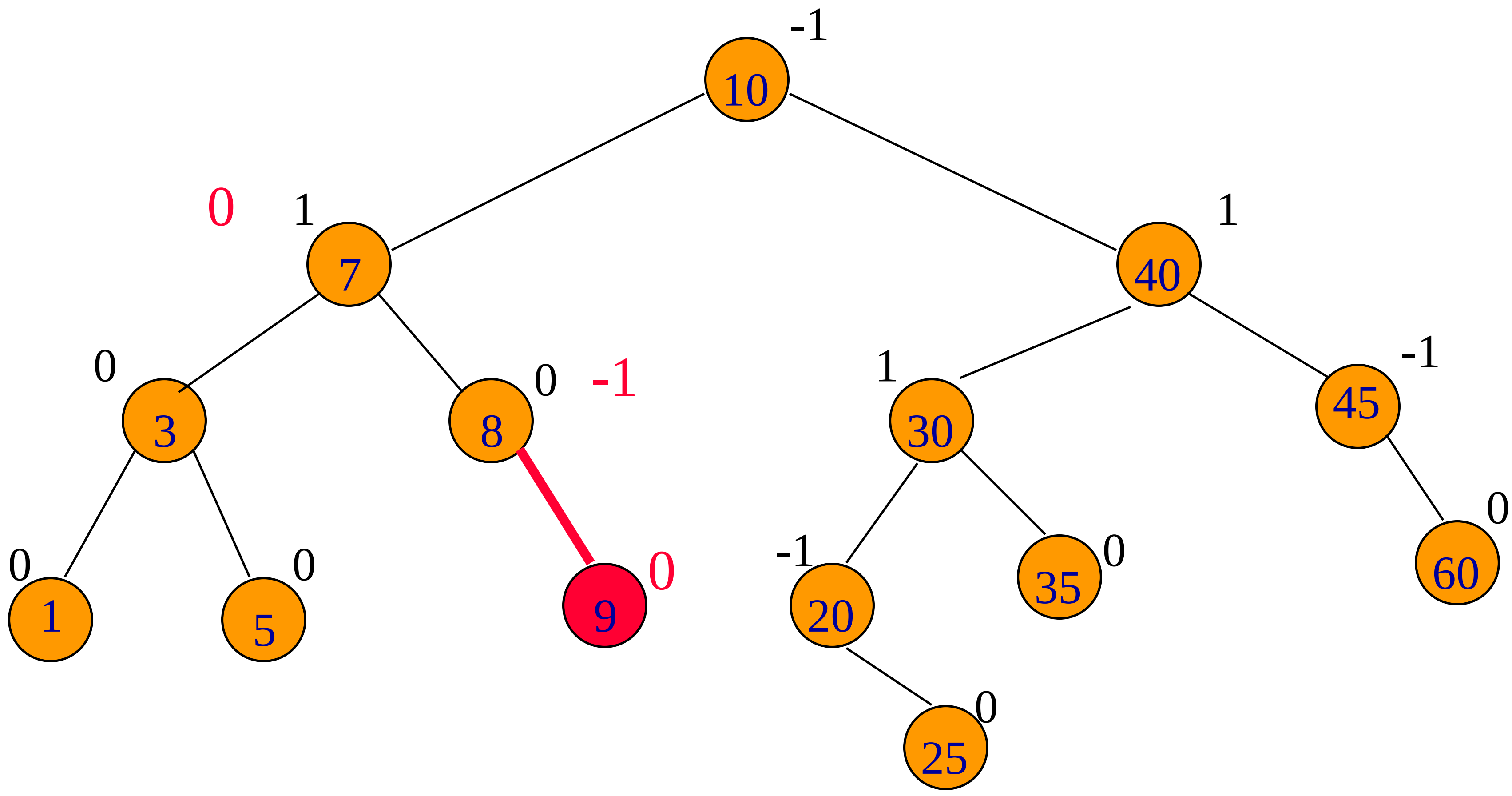
The height of an AVL tree that has n nodes is at most $1.44 \log_2 (n+2)$.

The height of every n node binary tree is at least $\log_2 (n+1)$.

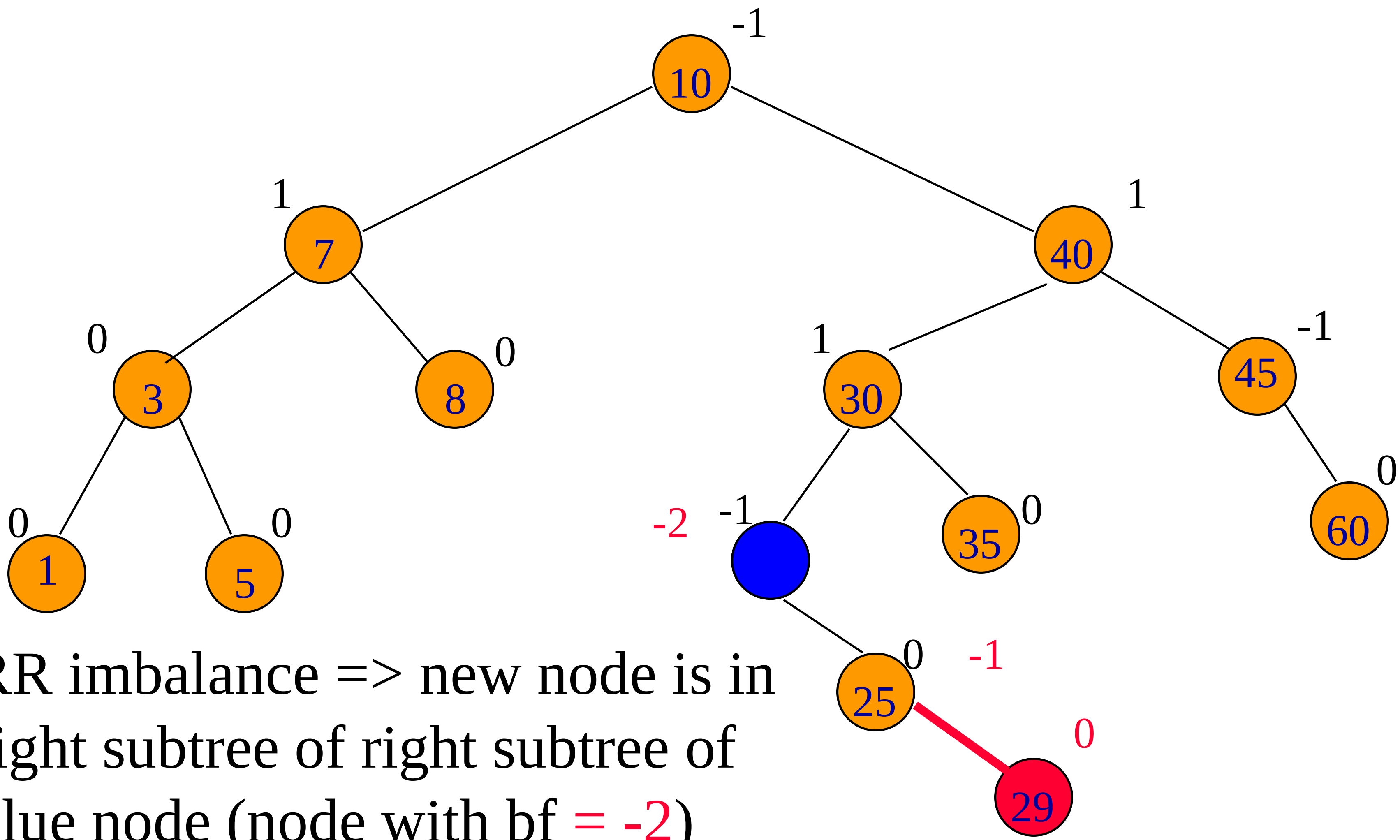
AVL Search Tree



put(9)

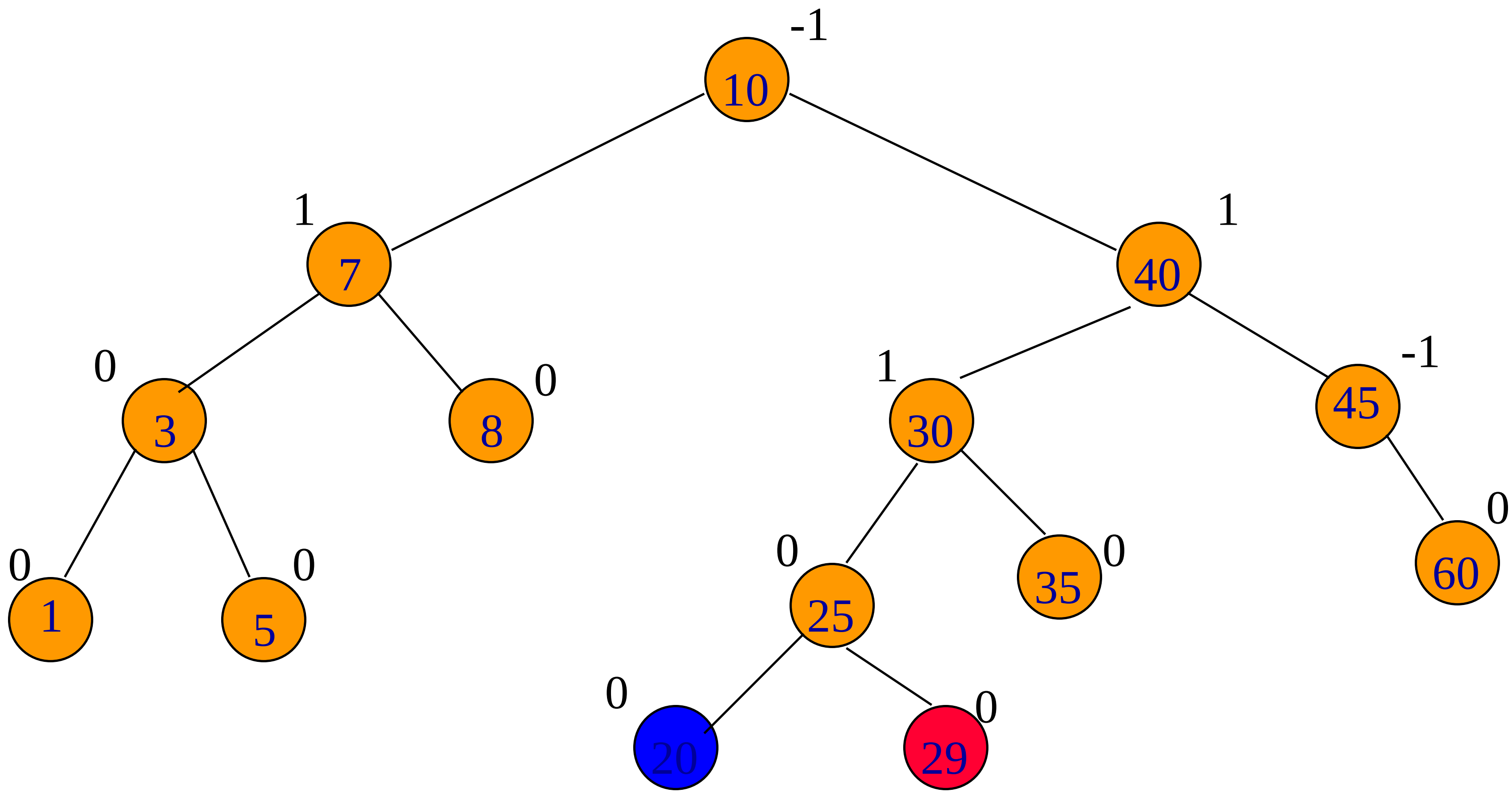


put(29)



RR imbalance => new node is in right subtree of right subtree of blue node (node with bf = -2)

put(29)



RR rotation.

Observations

1. In the unbalanced tree the BFs are limited to –2, -1, 0, 1, 2
2. A node with BF “2” had a BF “1” before the insertion
3. The BF of only those nodes on the path from the root to the newly inserted node can change as a result of the insertion
4. Let **A** denote the nearest ancestor of the newly inserted node whose BF is either –2 or 2. The BF of all nodes on the path from A to the newly inserted node was 0 prior to the insertion
5. Imbalance can happen in the last node encountered that has a balance factor 1 or –1 prior to the insertion

AVL Rotations

- The imbalance at A is one of the types
  - LL (when new node is in the left-subtree of the left-subtree of A)
  - LR (when new node is in the right-subtree of the left-subtree of A)
  - RR (when new node is in the right-subtree of the right-subtree of A)
  - RL (when new node is in the left-subtree of the right-subtree of A)
- LL and RR imbalances require single rotation
- LR and RL imbalances require double rotation
- At most one rotation suffices to restore balance.

Deletion from AVL tree

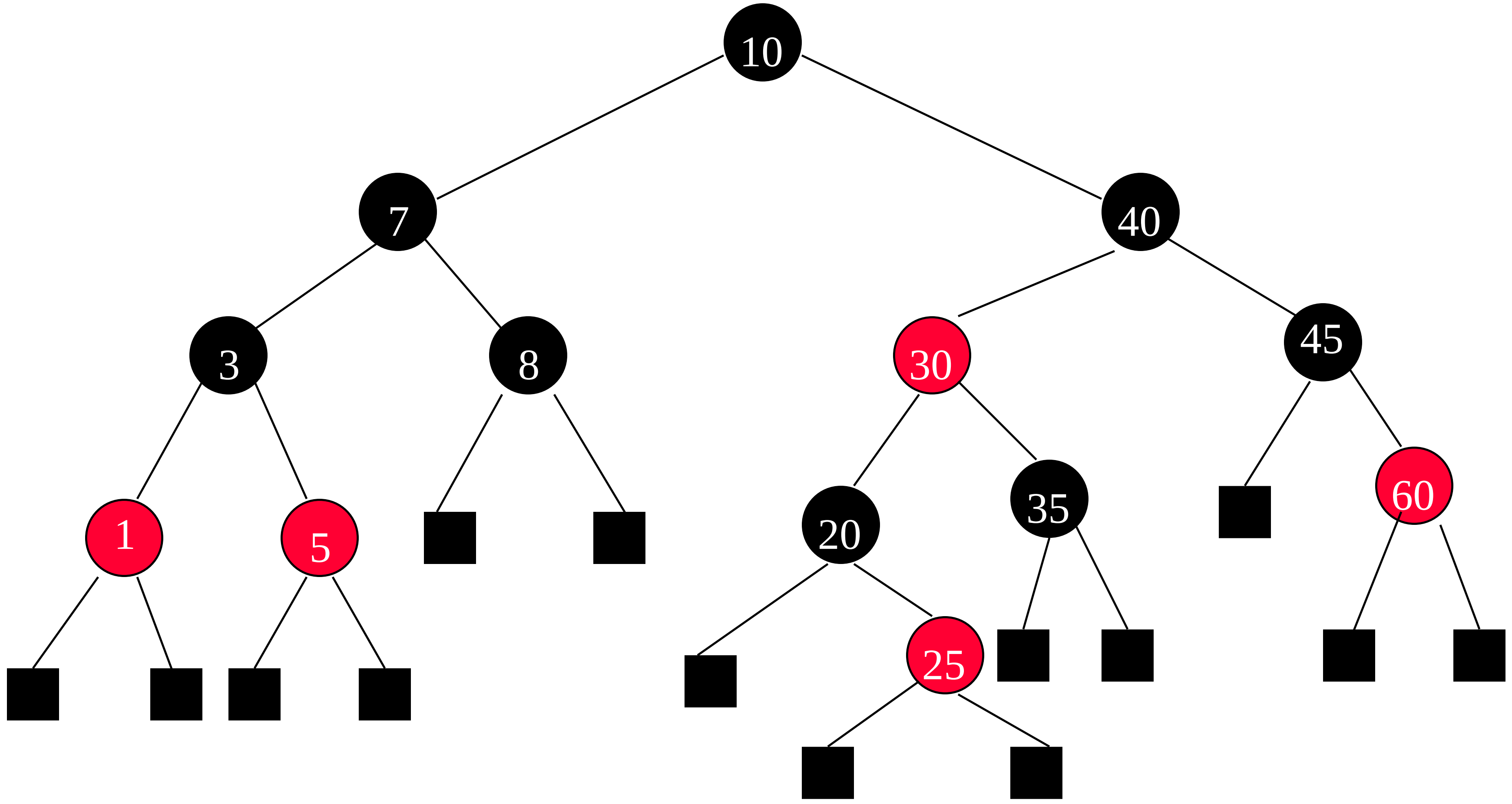
- Delete the node as in a normal binary search tree
- If the BF of the parent of the node that was physically deleted changes to 2 or -2, then we need to rebalance the tree by rotations
- Unlike in the case of insertions, we might need to perform several rotations all the way up to the root.

Red Black Trees

Colored Nodes Definition

- Binary search tree.
- Each node is colored red or black.
- Root and all external nodes are black.
- No root-to-external-node path has two consecutive red nodes.
- All root-to-external-node paths have the same number of black nodes

Example Red Black Tree

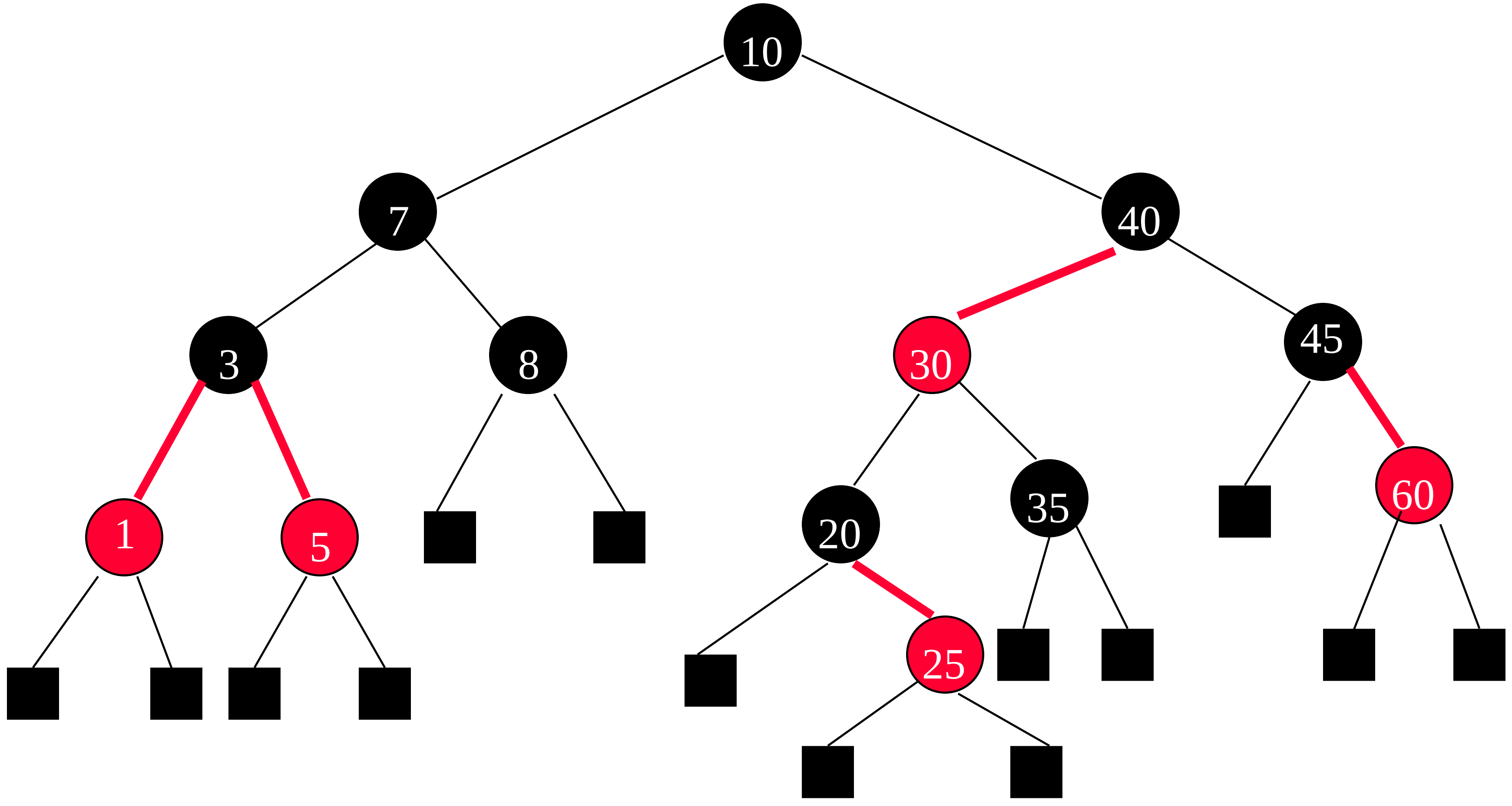


Red Black Trees

Colored Edges Definition

- Binary search tree.
- Child pointers are colored red or black.
- Pointer to an external node is black.
- No root to external node path has two consecutive red pointers.
- Every root to external node path has the same number of black pointers.

Example Red Black Tree



Red Black Tree

- If P and Q are two root-to-external-node paths in a red-black tree, then $\text{length}(P) \leq 2 * \text{length}(Q)$

- The height of a red black tree that has n (internal) nodes is between $\log_2(n+1)$ and $2\log_2(n+1)$.

Insertions

- Color the newly inserted node red (otherwise it will violate the last property)
- If the parent of the inserted node is red, then
  - If the sibling of the parent is also colored red, then we color the parent and its sibling black, and color the grand-parent red, and continue
  - Otherwise, performing one appropriate rotation will restore the red-black properties.

Deletions

- Let y be the node that takes the place of the node that is physically removed
- A violation occurs only when the deleted node was black, and y is not the root of the resulting tree.
- The red-black properties can be restored by performing zero or more color changes followed by one rotation.

Red Black Tree

- The height of a red black tree that has n (internal) nodes is between $\log_2(n+1)$ and $2\log_2(n+1)$.
- java.util.TreeMap => red black tree

BST Sequence of Operations

- Worst case for a single BST operation can be *O*(N)
- Not so bad if this happens only occasionally
- BUT...it's not uncommon for an entire sequence of "bad" operations to occur. In this case, a sequence of M operations take *O*(M*N) time and the time for the sequence of operations becomes noticeable.

Splay Tree Sequence of Operations

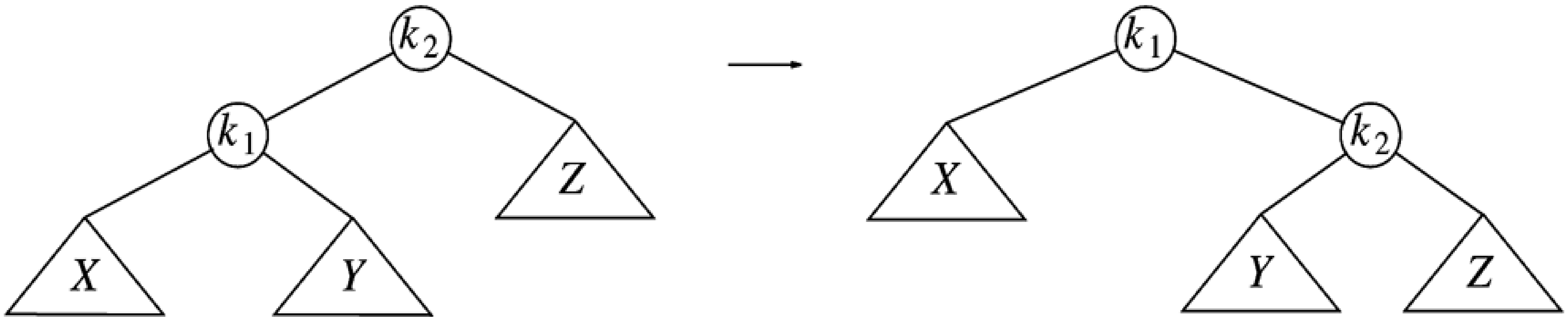
- Splay trees guarantee that a sequence of M operations takes at most *O*(M * lg N) time.
- We say that the splay tree has ***amortized*** running time of *O*(lg N) cost per operation. Over a sequence of operations, some may take more than lg N time, some will take less.
- Does not preclude the possibility that any particular operation is still *O*(N) in the worst case.
  - Therefore, amortized O(lg N) not as good as worst case O(lg N)
  - But, the effect is the same – there is no “bad” sequence of operations or bad input sequences

Splay Trees

- The basic idea of the splay tree is that every time a node is accessed, it is pushed to the root by a series of ***tree rotations.*** This series of tree rotations is known as “splaying”.

- If the node being “splayed” is deep, many nodes on the path to that node are also deep and by restructuring the tree, we make access to all of those nodes cheaper in the future.

Basic “Single” Rotation in a BST



Rotating k_1 around k_2

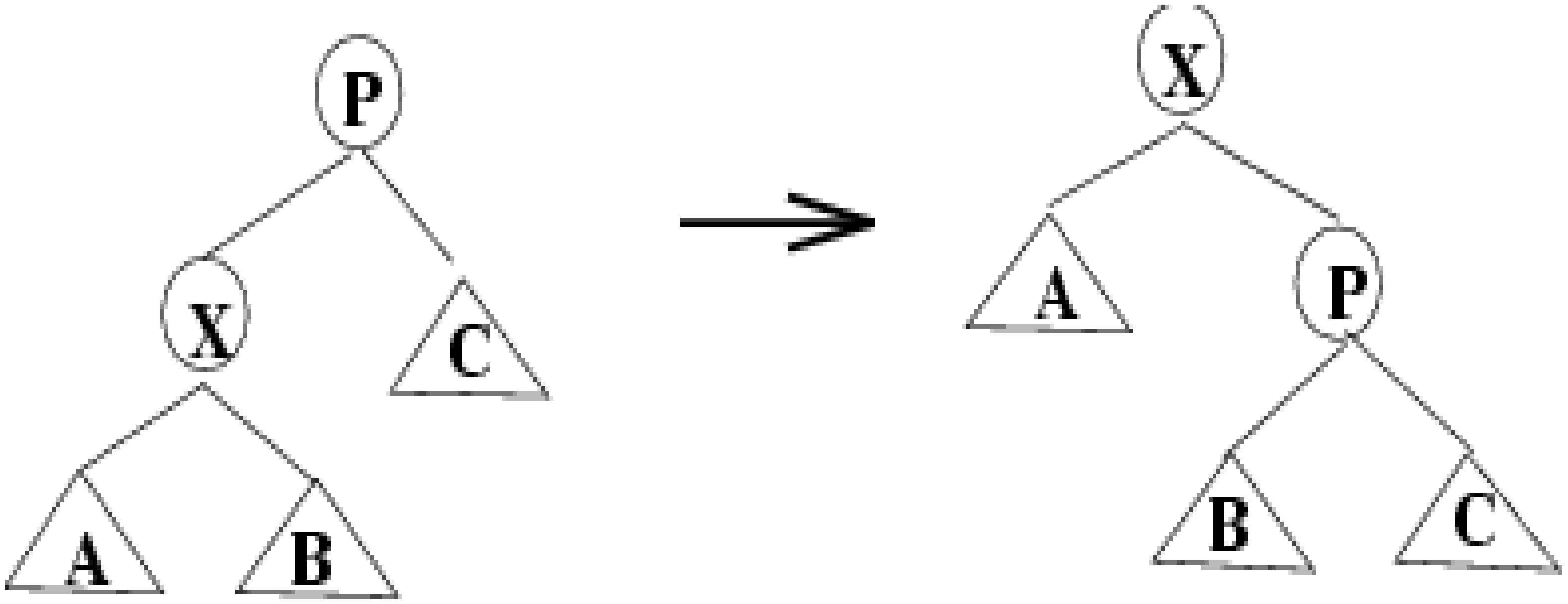
Assuming that the tree on the left is a BST, how can we verify that the tree on the right is still a valid BST?

Note that the rotation can be performed in either direction.

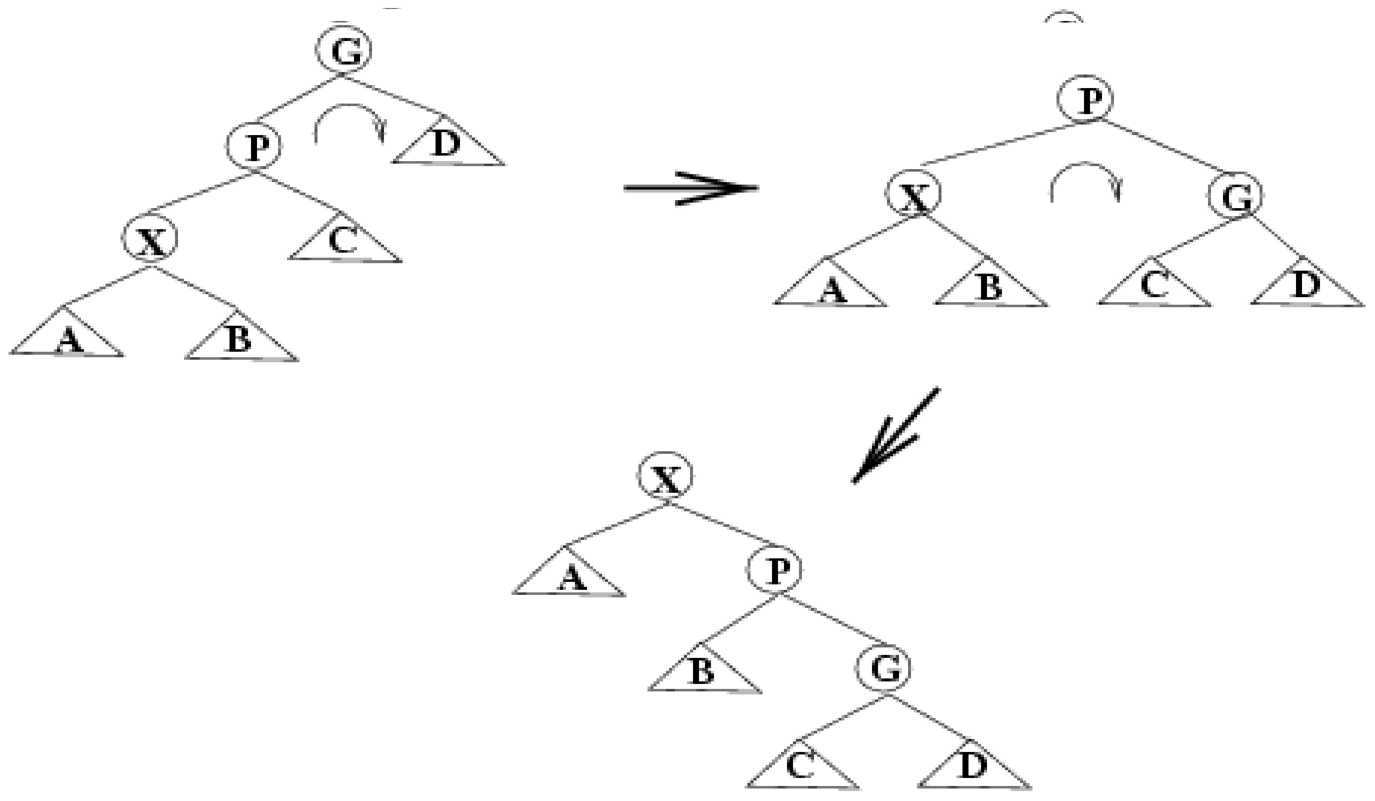
Splay Operation

- To “splay node x”, traverse up the tree from node x to root, rotating along the way until x is the root:
  - If x is root, do nothing.
  - If x has no grandparent, rotate x about its parent
  - If x has a grandparent,
    - if x and its parent are both left children or both right children, rotate the parent about the grandparent, then rotate x about its parent
    - if x and its parent are opposite type children (one left and the other right), rotate x about its parent, then rotate x about its new parent (former grandparent)

Node has no grandparent

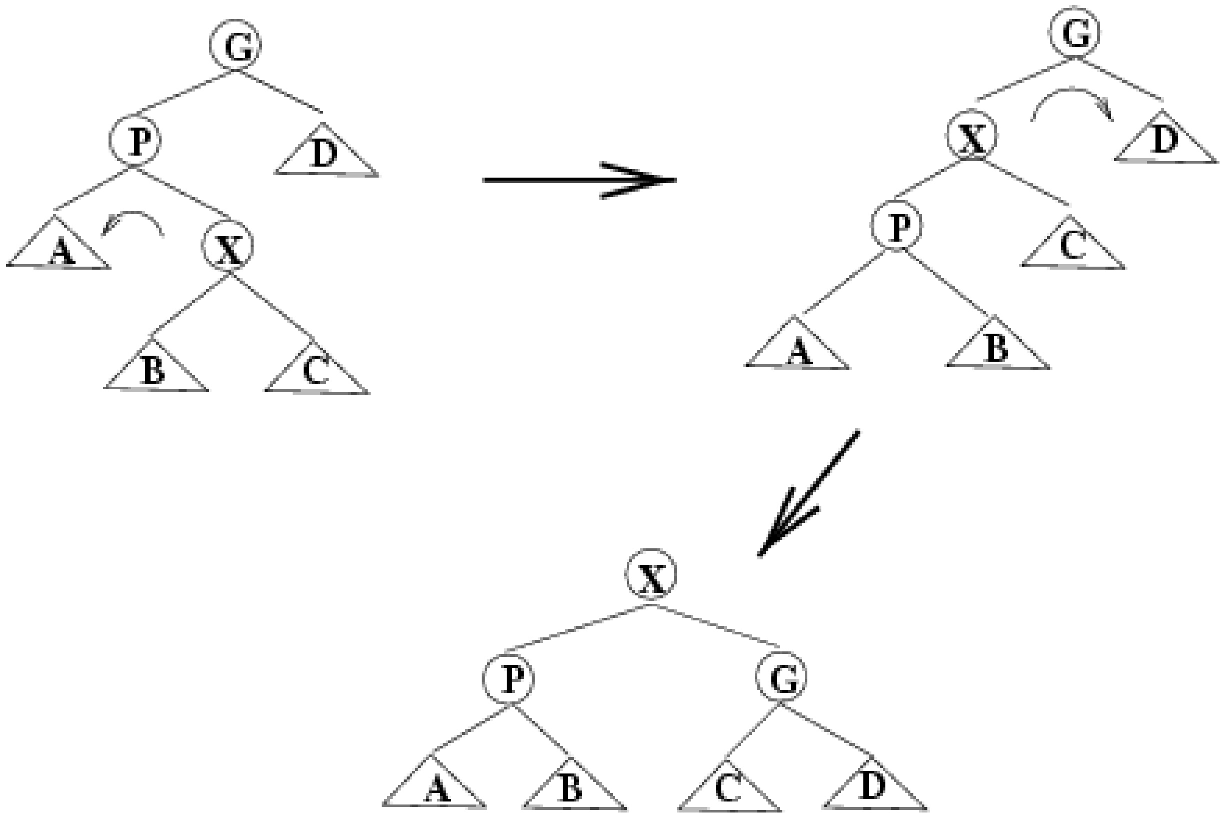


Node and Parent are Same Side Zig-Zig



Rotate P around G, then X around P

Node and Parent are Different Sides - Zig-Zag



Rotate X around P, then X around G

Operations in Splay Trees

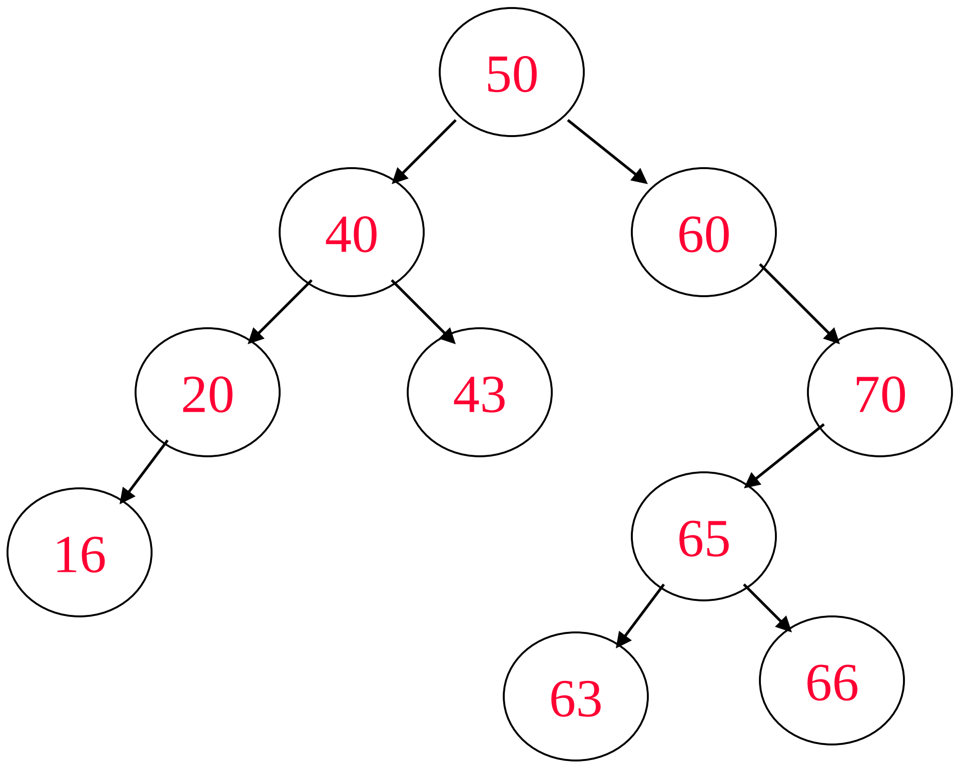
- insert
  - first insert as in normal binary search tree
  - then splay inserted node

- find/contains
  - search for node
  - if found, splay it; otherwise splay last node accessed on the search path

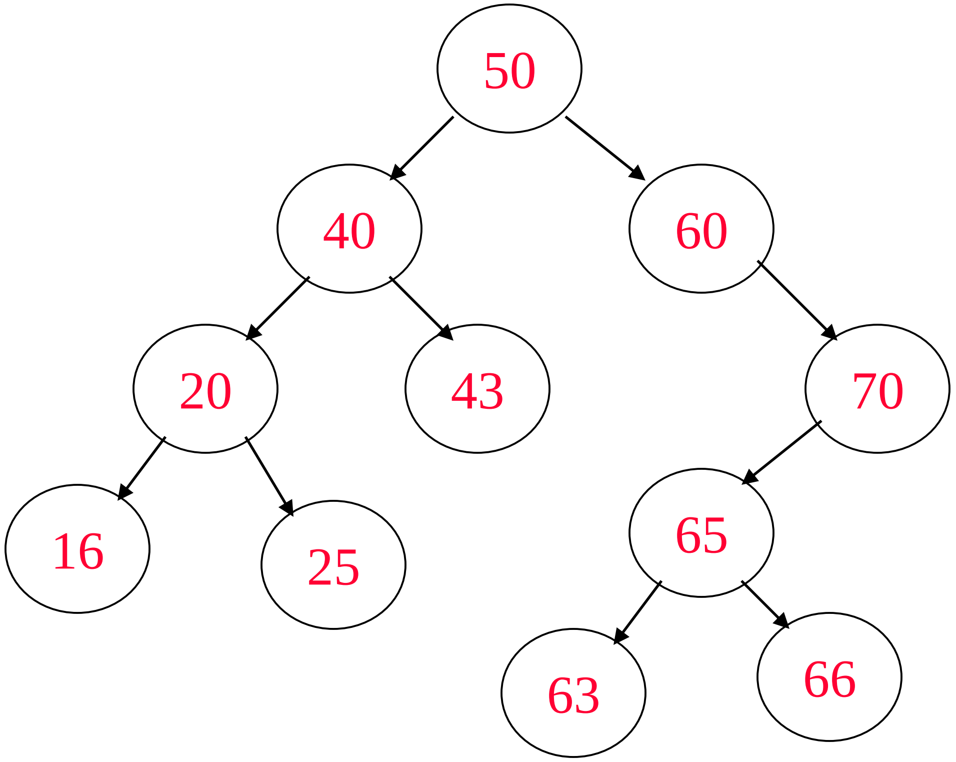
Operations on Splay Trees (cont)

- remove
  - splay element to be removed
    - if the element to be deleted is not in the tree, the node last visited on the search path is splayed
  - disconnect left and right subtrees from root
  - do one of:
    - splay max item in T_L (unless T_L has no right child)
    - splay min item in T_R (unless T_R has no left child)
  - connect other subtree to empty child of root

Exercise - find(65)



Exercise - remove(25)


50
40
60
20
43
70
16
25
65
63
66