Lecture 12

AVL Trees

Dr. Yusuf H. Sahin
Istanbul Technical University

sahinyu@itu.edu.tr

Maintenance Operations

- Developed in 1962 by G. M. Adelson-Velskii and E. M. Landis.
- A type of balanced binary search tree which ensures subtree height difference is at most 1.

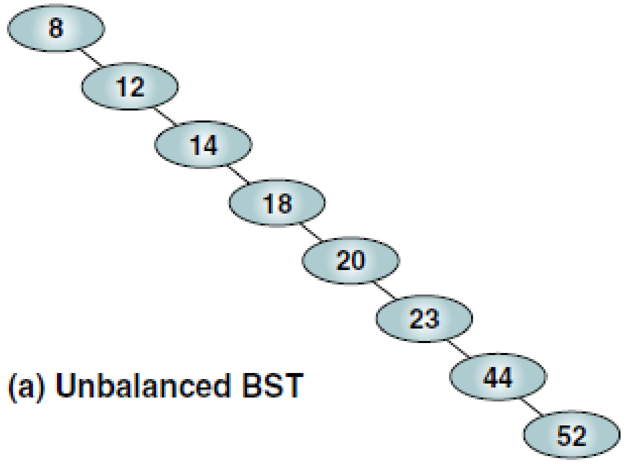
G. M. Adelson-Velskii (1922-2014)

E. M. Landis (1921-1997)

An AVL tree is a binary tree that either is empty or consists of two AVL subtrees, T_L, and T_R, whose heights differ by no more than 1.

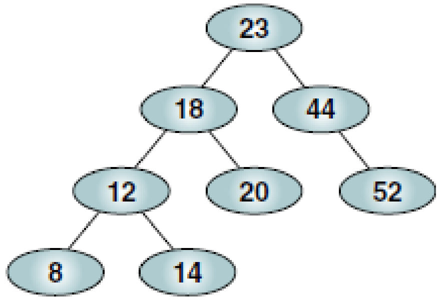
$$|H_L - H_R| \leq 1$$

Search: $O(n)$



(a) Unbalanced BST

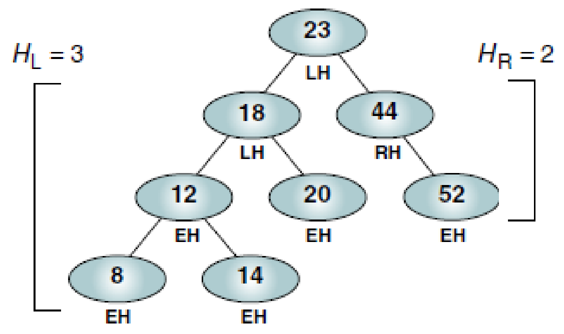
Search: $O(logn)$



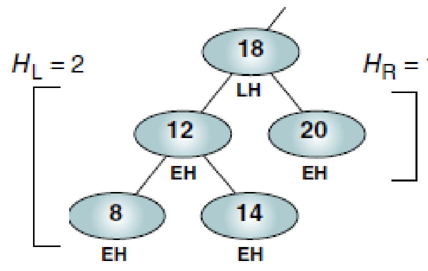
(b) AVL tree

AVL Tree Balance Factor

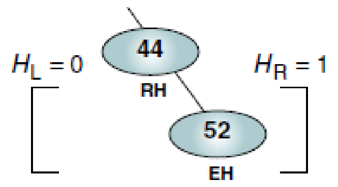
- Balance Factor: Calculated as the height of the left subtree minus the height of the right subtree.
- Allowed Values:
  - **+1 (LH)**: Left subtree is taller.
  - **0 (EH)**: Subtrees are of equal height.
  - **–1 (RH)**: Right subtree is taller.



(a) Tree 23 appears balanced: $H_L - H_R = 1$



(b) Subtree 18 appears balanced: $H_L - H_R = 1$



(c) Subtree 44 is balanced: $| H_L - H_R | = 1$

Balancing Trees

- Tree Imbalance:
  - Occurs after inserting or deleting a node.

- Rebalancing:
  - Detect imbalance and restore balance. Achieved through left or right rotations in AVL trees.

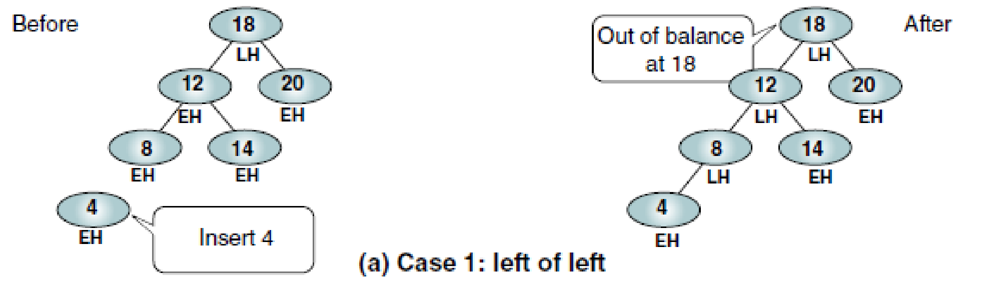
There are four cases:

1. Left of left—A subtree of a tree that is left high has also become left high.

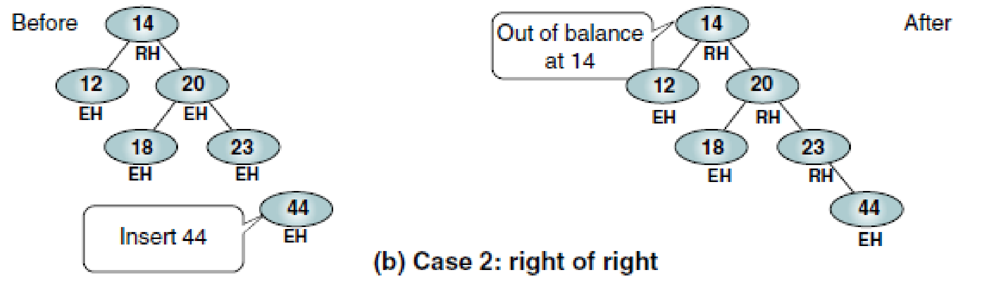
2. Right of right—A subtree of a tree that is right high has also become right high.

3. Right of left—A subtree of a tree that is left high has become right high.

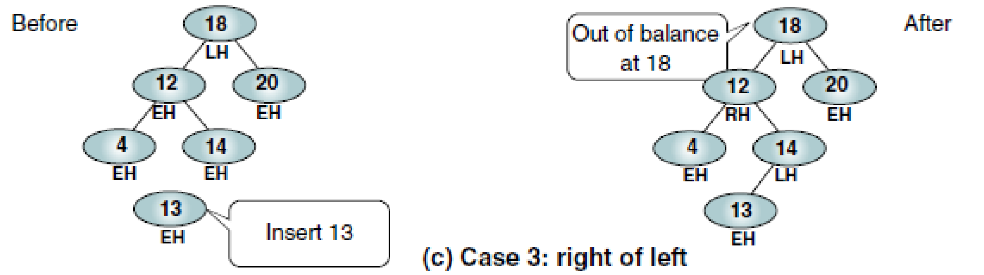
4. Left of right—A subtree of a tree that is right high has become left high.



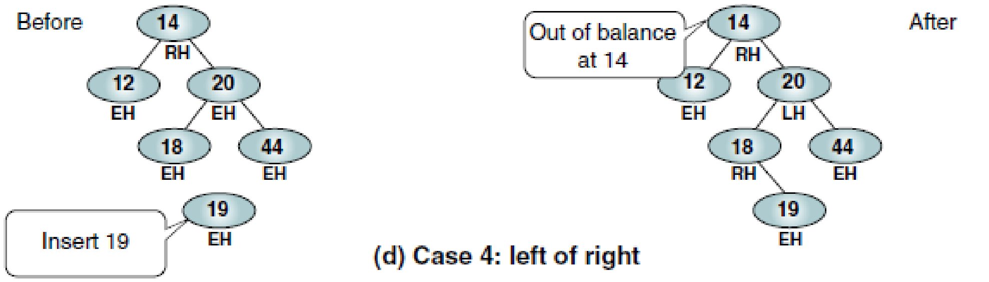
(a) Case 1: left of left



(b) Case 2: right of right



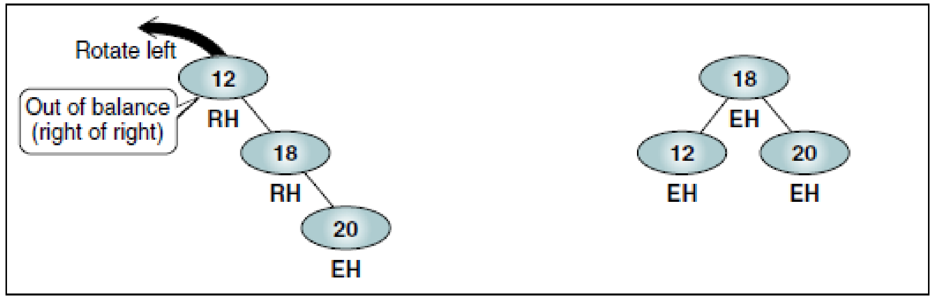
(c) Case 3: right of left



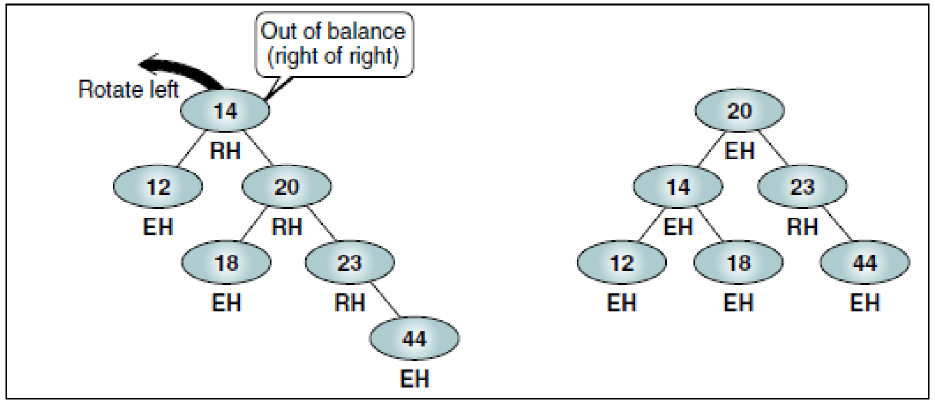
(d) Case 4: left of right

Balancing Trees

2. Right of right— We must balance the tree by rotating the out-of-balance node to the left.



(a) Simple left rotation

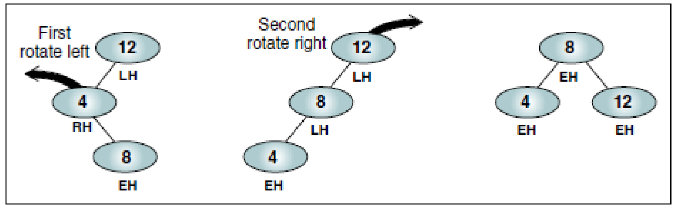


(b) Complex left rotation

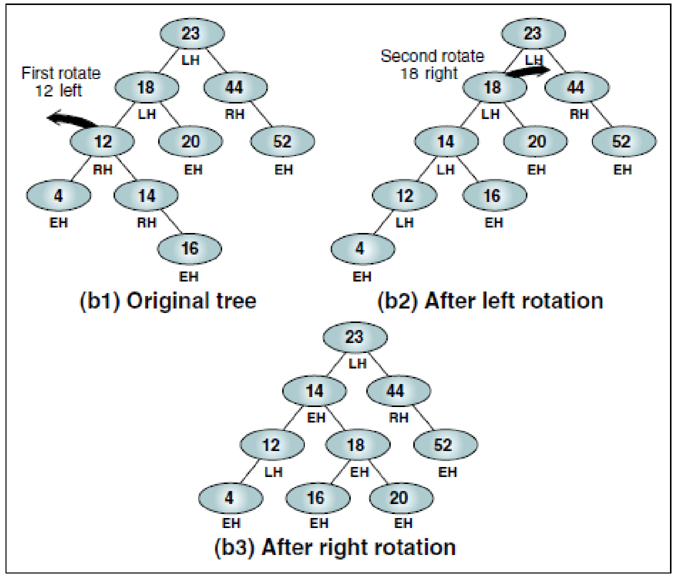
Balancing Trees

Both left and rigth rotations are required for certain imbalance conditions.

3. Right of left

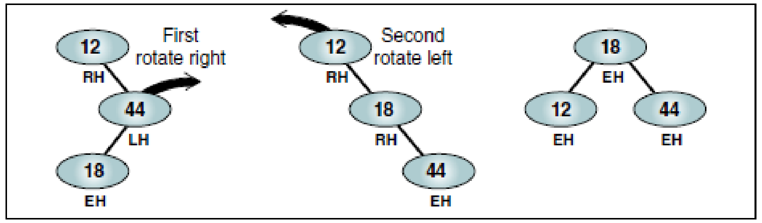


(a) Simple double rotation right

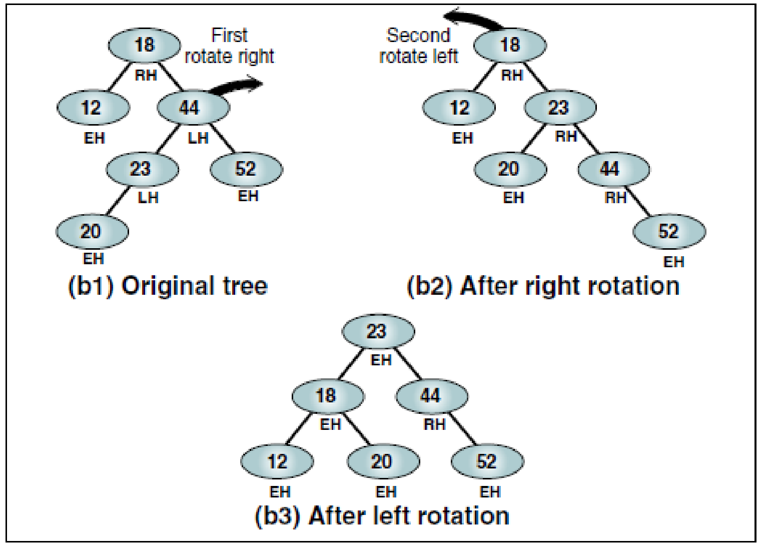


(b) Complex double rotation right

4. Left of right



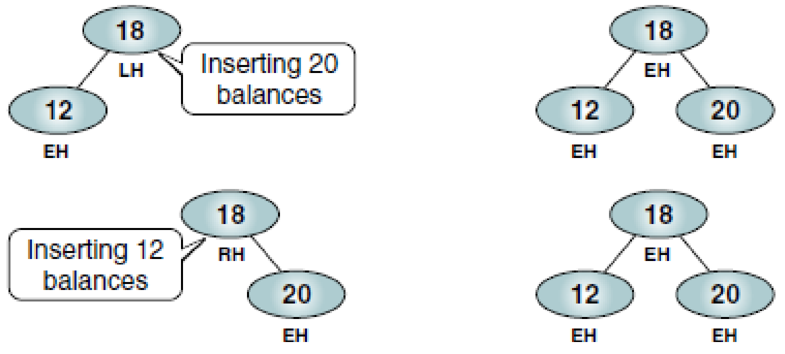
(a) Simple double rotation right



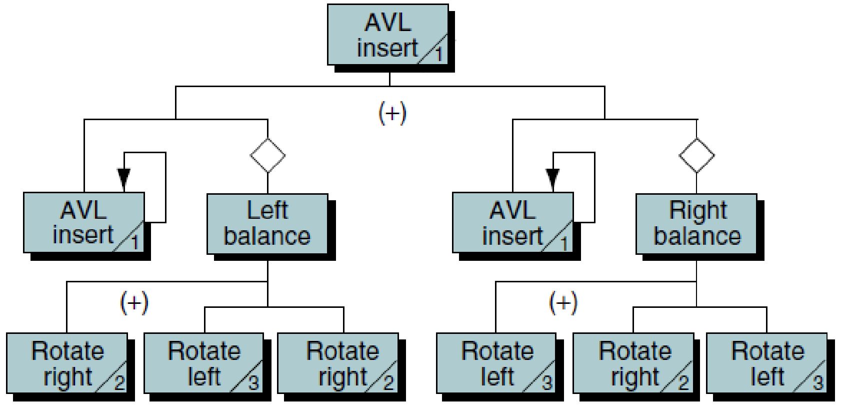
(b) Complex double rotation right

AVL Tree Implementations

- **Search and Retrieval**:
  - Same as standard binary tree operations.
  - Use inorder traversal due to AVL's search tree structure.
- **Insert Algorithm**:
  - Inserts occur at leaf nodes.
  - Path determined by comparing keys:
    - Go left if the new key is smaller.
    - Go right if the new key is larger.
  - Leaf is connected to its parent, and backtracking begins.
- **Balancing During Insert**:
  - Balance is checked at each node while backtracking.
  - If a node is unbalanced, perform rotations to restore balance.
- **Automatic Balancing**:
  - Adding to the right branch of a left-high node makes it even high.
  - Adding to the left branch of a right-high node balances it automatically.



Insert Algorithm



- Left branch insertion is mirrored by right branch insertion.

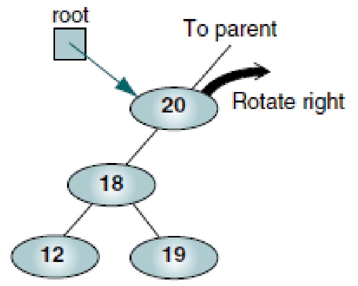
- Both follow these steps:
  1. Use a recursive call to locate the appropriate leaf.
  2. Once the leaf is found, backtrack through recursion.
  3. If the tree height has grown, call either left balance or right balance to rebalance.

```
Algorithm AVLInsert (root, newData)
Using recursion, insert a node into an AVL tree.
  Pre     root is pointer to first node in AVL tree/subtree
          newData is pointer to new node to be inserted
  Post    new node has been inserted
  Return root returned recursively up the tree
1 if (subtree empty)
  Insert at root
  1  insert newData at root
  2  return root
2 end if
3 if (newData < root)
  1  AVLInsert (left subtree, newData)
  2  if (left subtree taller)
     1  leftBalance (root)
  3  end if
4 else
  New data >= root data
  1  AVLInsert (right subtree, newPtr)
  2  if(right subtree taller)
     1  rightBalance (root)
  3  end if
5 end if
6 return root
end AVLInsert
```

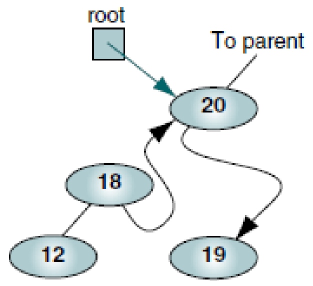
Left & Right Balance

```
Algorithm leftBalance (root)
This algorithm is entered when the root is left high (the
left subtree is higher than the right subtree).
   Pre    root is a pointer to the root of the [sub]tree
   Post   root has been updated (if necessary)
1 if (left subtree high)
   1 rotateRight (root)
2 else
   1 rotateLeft (left subtree)
   2 rotateRight (root)
3 end if
end leftBalance
```

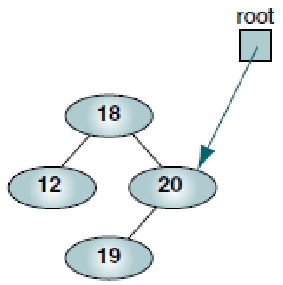
```
Algorithm rotateRight (root)
This algorithm exchanges pointers to rotate the tree right.
   Pre    root points to tree to be rotated
   Post   node rotated and root updated
1 exchange left subtree with right subtree of left subtree
2 make left subtree new root
end rotateRight

Algorithm rotateLeft (root)
This algorithm exchanges pointers to rotate the tree left.
   Pre    root points to tree to be rotated
   Post   node rotated and root updated
1 exchange right subtree with left subtree of right subtree
2 make right subtree new root
end rotateLeft
```



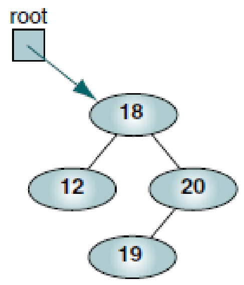
(a) At beginning



(b) After exchange



(c) After attaching root

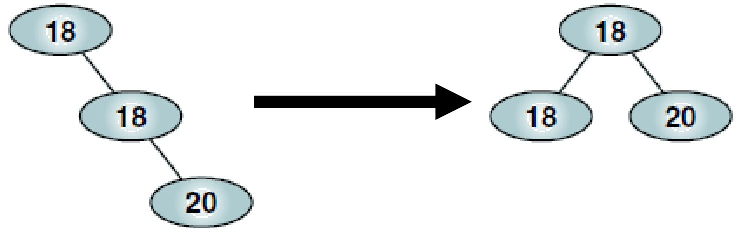


(d) At end

Duplicate Keys

- Duplicate keys can lead to an equal key being rotated to the left subtree.
- Deletion Challenge
  - Determining which node to delete when duplicate keys exist.
  - Recursive inorder search always locates the first node on a right branch.
  - It cannot locate a node in the left subtree with the same key.

- **Solution:** Implement an iterative search. This search locates the node just before the desired key for deletion.



Adjusting the Balance Factors

- There is a general pattern:
  - If the root was even balanced before an insert, it is now high on the side in which the insert was made.
  - If an insert was in the shorter subtree of a tree that was not even balanced, the root is now even balanced.
  - If an insert was in the higher subtree of a tree that was not even balanced, the root must be rotated.