BIRZEIT UNIVERSITY

COMP242

Data Structure

Lectures Note: AVL Trees

Prepared by: **Dr. Mamoun Nawahdah**

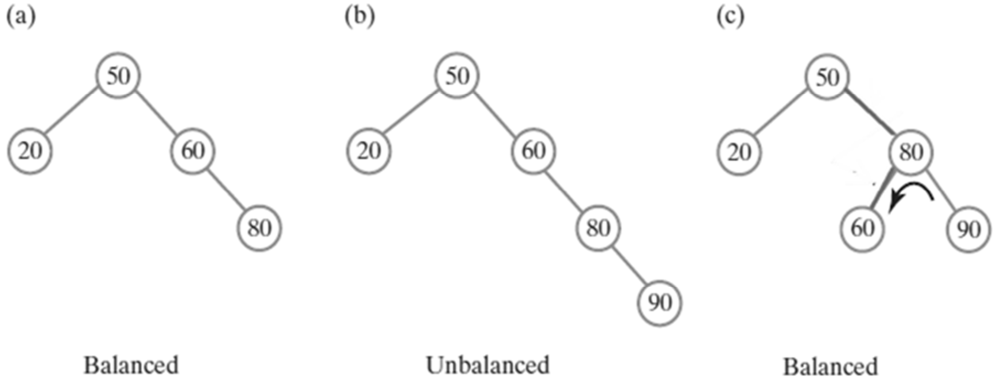
2016/2017

AVL Trees

- An **AVL tree** (Georgy **A**delson-**V**elsky and Evgenii **L**andis' tree**)** is a **BST** with the additional **balance** property that, for any node in the tree, the height of the **left** and **right** subtrees can differ by at most **1**.
- **Complete** binary trees are **balanced.**

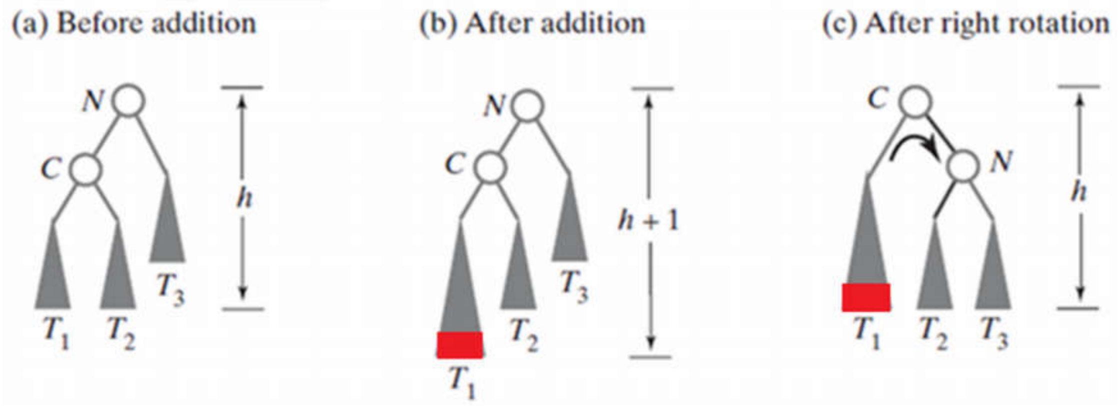
Single Rotation

Example: After inserting (a) 60; (b) 50; and (c) 20 into an initially empty **BST**, the tree is **not balanced**; (d) a corresponding **AVL** tree rotates its nodes to restore balance



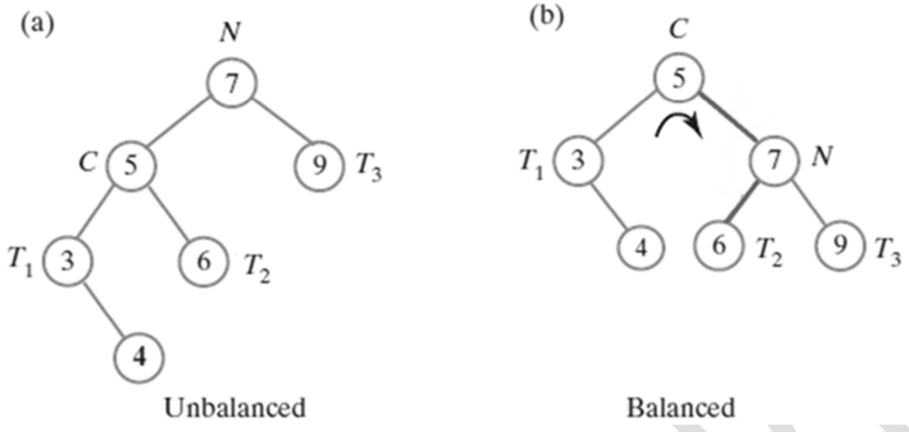
Example: (a) Adding 80 to the tree does not change the balance of the tree; (b) a subsequent addition of 90 makes the tree **unbalanced** ; (c) a left rotation restores its balance

Case 1: Single Right Rotation (left-left addition)



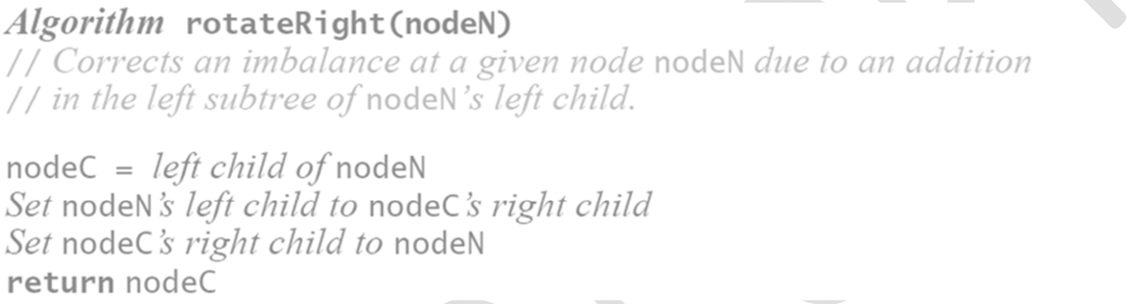
Before and after an addition to an **AVL** subtree that requires a **right rotation** to maintain its balance.

Example: a) before and b) after a **right rotation** restores balance to an **AVL** tree



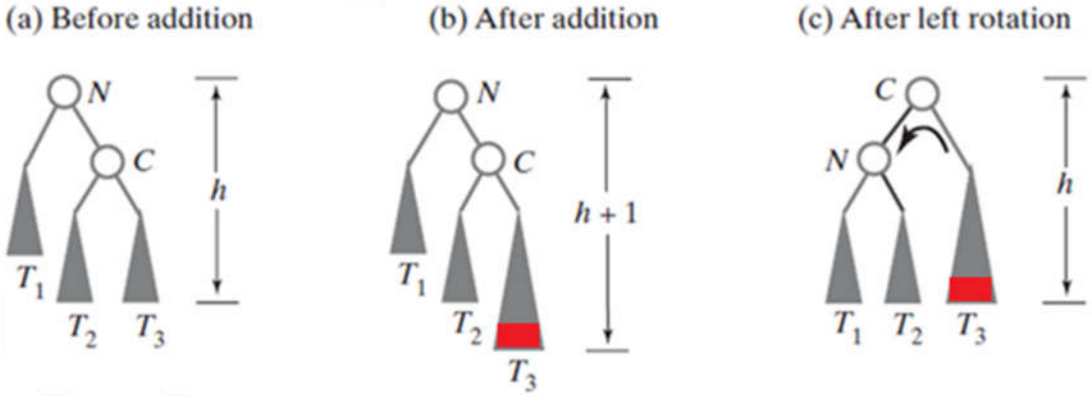
Unbalanced

Balanced



```
Algorithm rotateRight(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the left subtree of nodeN's left child.

nodeC = left child of nodeN
Set nodeN's left child to nodeC's right child
Set nodeC's right child to nodeN
return nodeC
```

Case 2: Single Left Rotation (right-right addition)

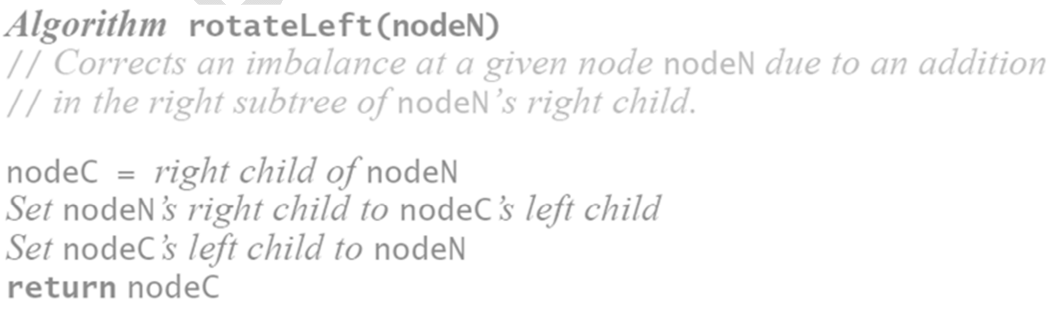


(a) Before addition

(b) After addition

(c) After left rotation

Before and after an addition to an **AVL** subtree that requires a **left rotation** to maintain its balance



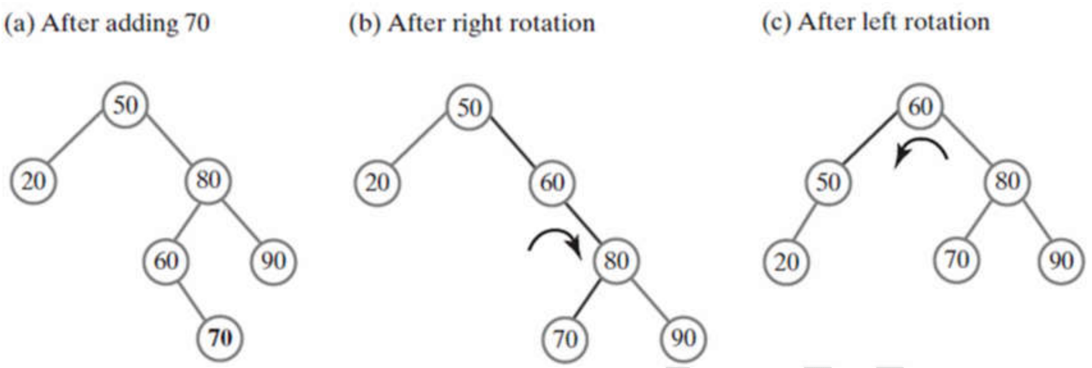
```
Algorithm rotateLeft(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the right subtree of nodeN's right child.

nodeC = right child of nodeN
Set nodeN's right child to nodeC's left child
Set nodeC's left child to nodeN
return nodeC
```

Double Rotations

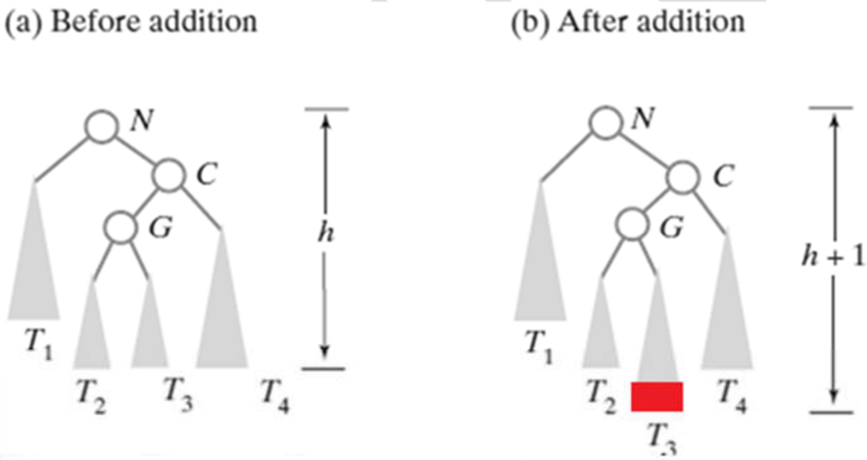
A **double rotation** is accomplished by performing two single rotations:

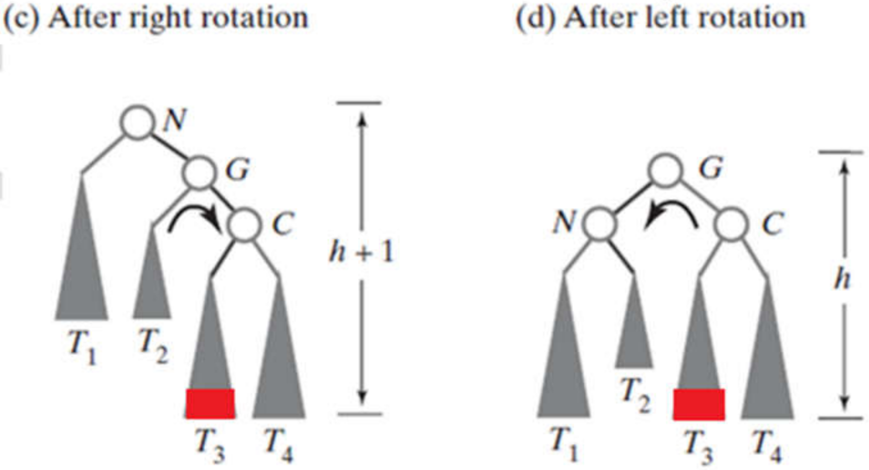
1. A rotation about node ***N*****'s grandchild *G*** (its child's child)
2. A rotation about node ***N*****'s new child**

Case 3: Right-Left Double Rotations (right-left addition)



Example: (a) Adding 70 destroys tree's balance; to restore the balance, perform both (b) a **right rotation** and (c) a **left rotation**



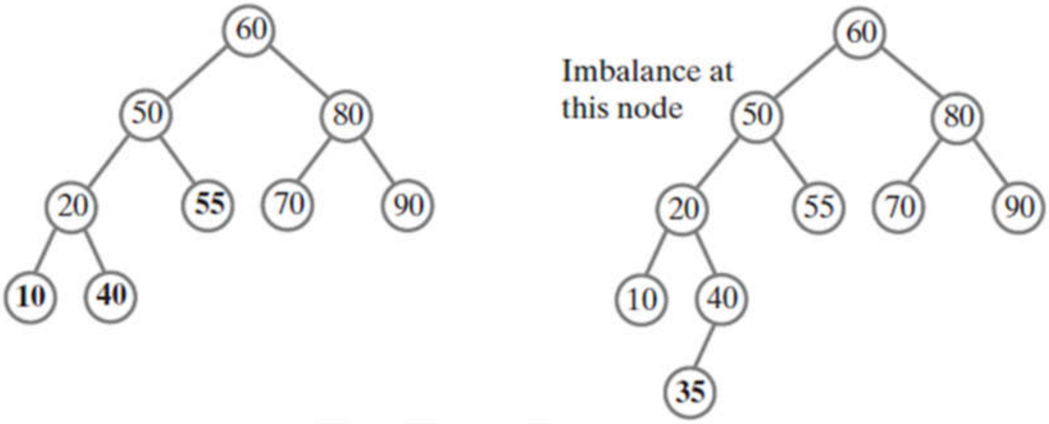


Before and after an addition to an **AVL** subtree that requires both a **right rotation** and a **left rotation** to maintain its balance

```
Algorithm rotateRightLeft(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the left subtree of nodeN's right child.

nodeC = right child of nodeN
Set nodeN's right child to the node returned by rotateRight(nodeC)
return rotateLeft(nodeN)
```

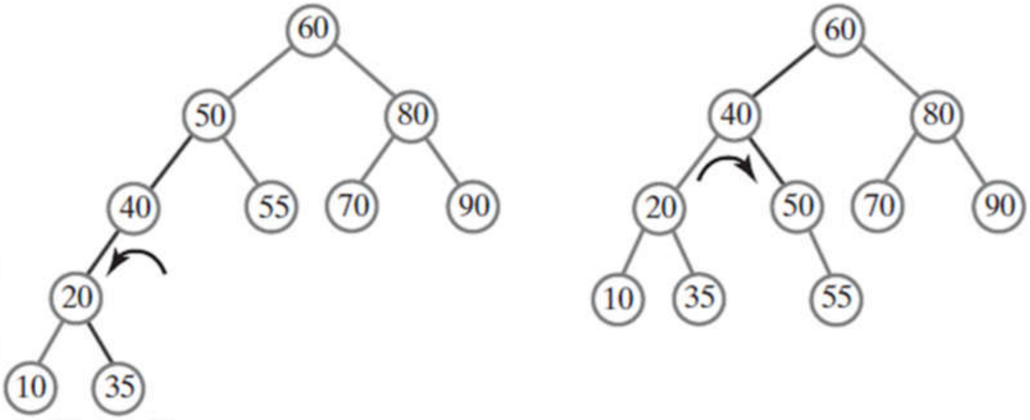
Case 4: Left-Right Double Rotations (left-right addition)

Example:



(a) After adding 55, 10, and 40

(b) After adding 35

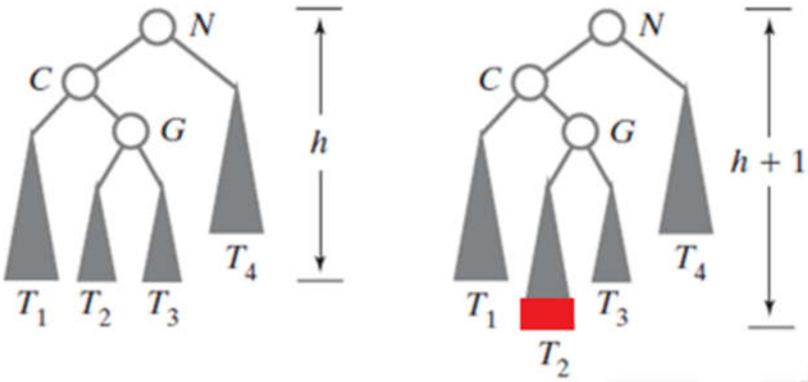


(c) After left rotation about 40

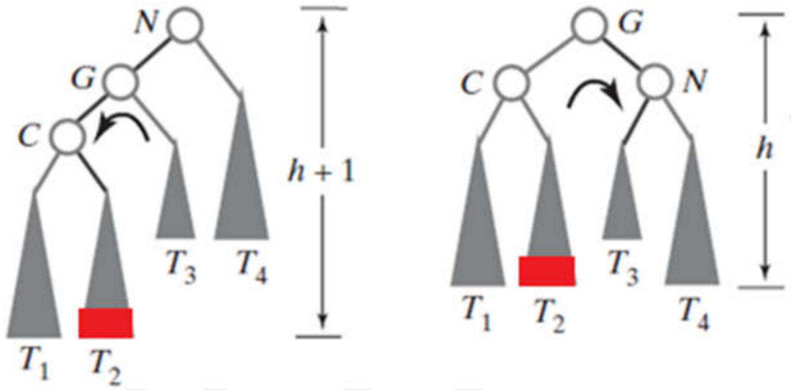
(d) After right rotation about 40

(a) The **AVL** tree after additions that maintain its balance;
(b) after an addition that destroys the balance;
(c) after a **left rotation**;
(d) after a **right rotation**

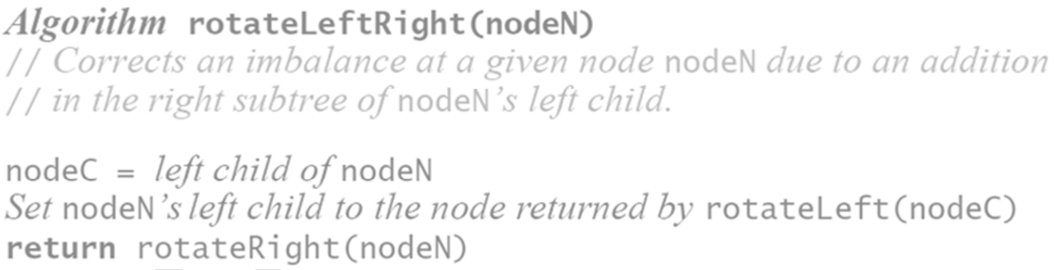
(a) Before addition (b) After addition



(c) After left rotation (d) After right rotation



Before and after an **addition** to an **AVL** subtree that requires both a **left rotation** and a **right rotation** to maintain its balance

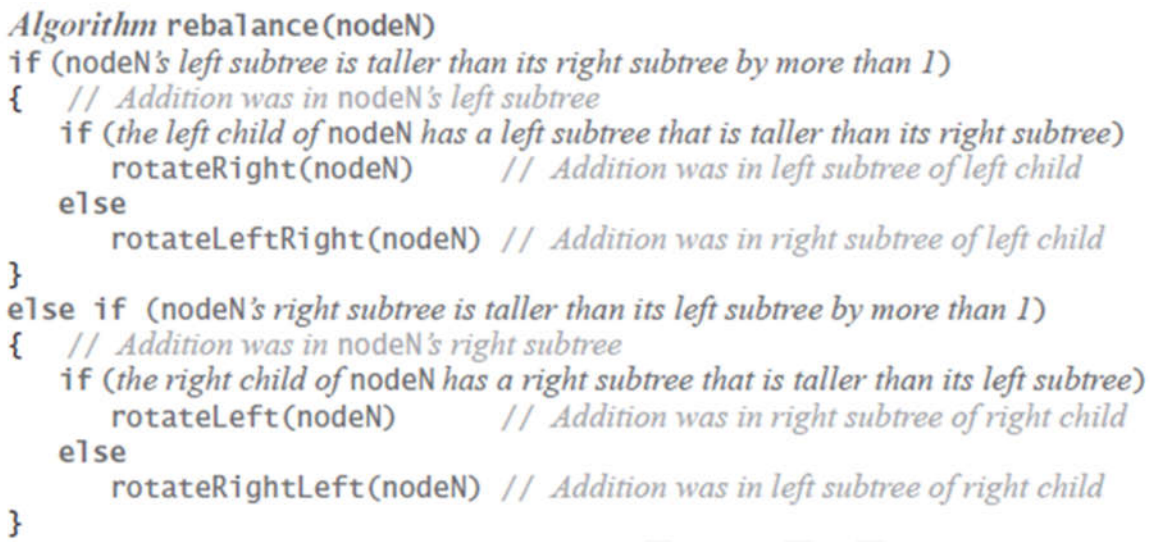


```
Algorithm rotateLeftRight(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the right subtree of nodeN's left child.

nodeC = left child of nodeN
Set nodeN's left child to the node returned by rotateLeft(nodeC)
return rotateRight(nodeN)
```

- Four rotations cover the only four possibilities for the cause of the imbalance at node ***N***
- The addition occurred at:
    - The left subtree of **N**'s left child (case 1: right rotation)
    - The right subtree of **N**'s left child (case 4: left-right rotation)
    - The left subtree of **N**'s right child (case 3: right-left rotation)
    - The right subtree of **N**'s right child (case 2: left rotation)

Rebalance Code Implementation

- Pseudo-code to rebalance the tree:



```
Algorithm rebalance(nodeN)
if (nodeN's left subtree is taller than its right subtree by more than 1)
{   // Addition was in nodeN's left subtree
    if (the left child of nodeN has a left subtree that is taller than its right subtree)
        rotateRight(nodeN)        // Addition was in left subtree of left child
    else
        rotateLeftRight(nodeN)    // Addition was in right subtree of left child
}
else if (nodeN's right subtree is taller than its left subtree by more than 1)
{   // Addition was in nodeN's right subtree
    if (the right child of nodeN has a right subtree that is taller than its left subtree)
        rotateLeft(nodeN)         // Addition was in right subtree of right child
    else
        rotateRightLeft(nodeN)    // Addition was in left subtree of right child
}
```

```
private TNode rebalance(TNode nodeN){
   int diff = getHeightDifference(nodeN);
   if ( diff > 1) { // addition was in node's left subtree
      if(getHeightDifference(nodeN.left)>0)
         nodeN = rotateRight(nodeN);
      else
         nodeN = rotateLeftRight(nodeN);
   }
   else if ( diff < -1){  // addition was in node's right subtree
      if(getHeightDifference(nodeN.right)<0)
         nodeN = rotateLeft(nodeN);
      else
         nodeN = rotateRightLeft(nodeN);
   }
   return nodeN;
}
```

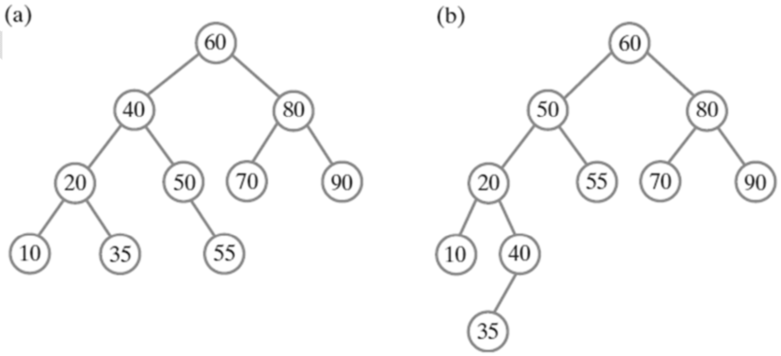
Insert Code Implementation:

```
public void insert(T data) {
   if(isEmpty())       root = new TNode<>(data);
   else {
      TNode rootNode = root;
      addEntry(data, rootNode);
      root = rebalance(rootNode);
   }
}

public void addEntry(T data, TNode rootNode){
   assert rootNode != null;
   if(data.compareTo((T)rootNode.data) < 0){ // right into left subtree
      if(rootNode.hasLeft()){
         TNode leftChild = rootNode.left;
         addEntry(data, leftChild);
         rootNode.left=rebalance(leftChild);
      }
      else         rootNode.left = new TNode(data);
   }
   else {  // right into right subtree
      if(rootNode.hasRight()){
         TNode rightChild = rootNode.right;
         addEntry(data, rightChild);
         rootNode.right=rebalance(rightChild);
      }
      else         rootNode.right = new TNode(data);
   }
}
```

Delete Code Implementation:

```
public TNode delete(T data) {
   TNode temp = super.delete(data);
   if(temp!= null){
      TNode rootNode = root;
      root = rebalance(rootNode);
   }
   return temp;
}
```

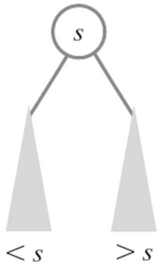
An **AVL** Tree versus a **BST:**

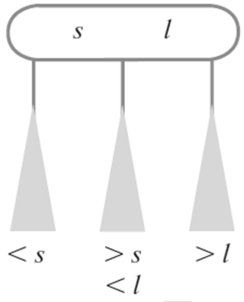


Example: The result of adding 60, 50, 20, 80, 90, 70, 55, 10, 40, and 35 to an initially empty (a) **AVL** tree; (b) **BST**

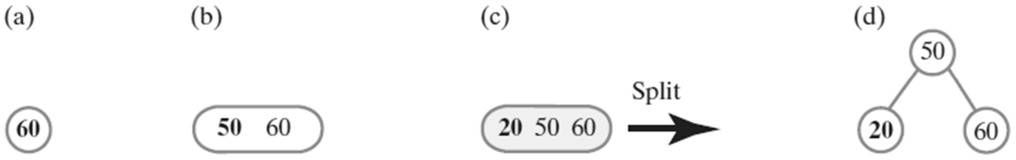
2-3 Trees

- **Definition**: general search tree whose interior nodes must have either **2** or **3** children.
    - A **2-node** contains one data item ***s*** and has two children.
    - A **3-node** contains two data items, ***s*** and ***l***, and has three children.

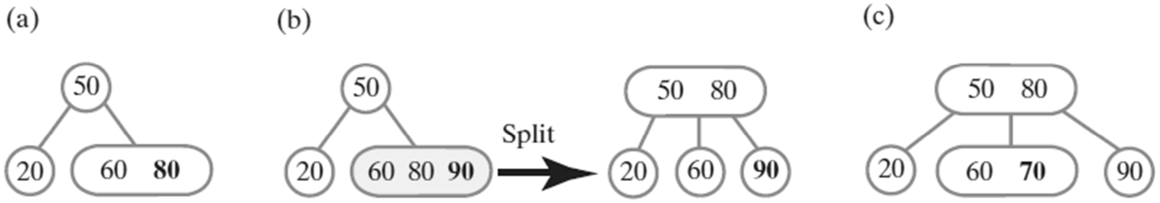




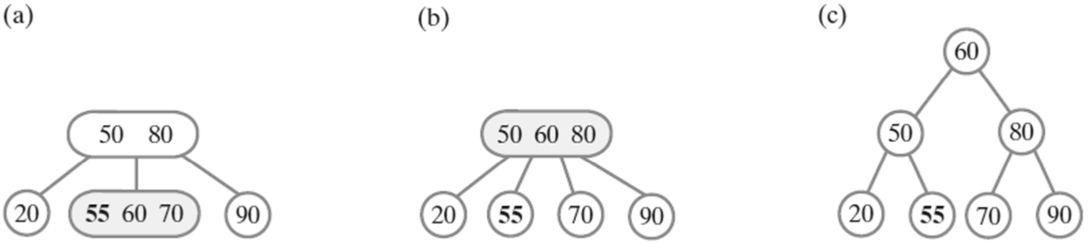
Adding Entries to a 2-3 Tree:



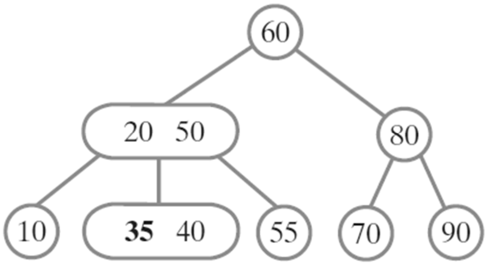
Adding (a) **60** and (b) **50**; (c), (d) adding **20** causes the 3-node to split



The 2-3 tree after adding (a) **80**; (b) **90**; (c) **70**



Adding **55** to the 2-3 tree, causes a leaf and then the root to split

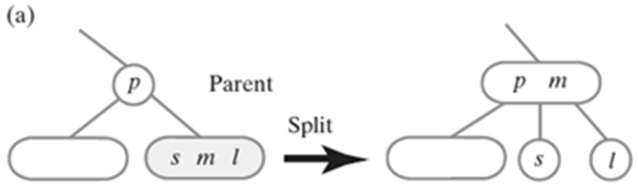


The 2-3 tree, after adding **10, 40, 35**

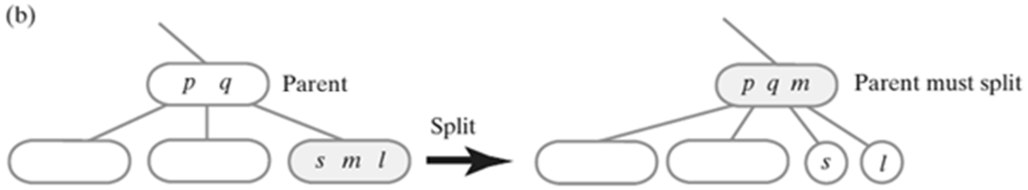
Splitting Nodes during Addition:

- Splitting a **leaf** to accommodate a new entry when the leaf's **parent** contains:

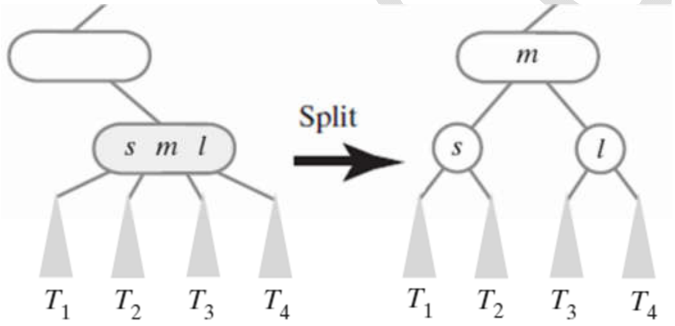
(a) one entry:



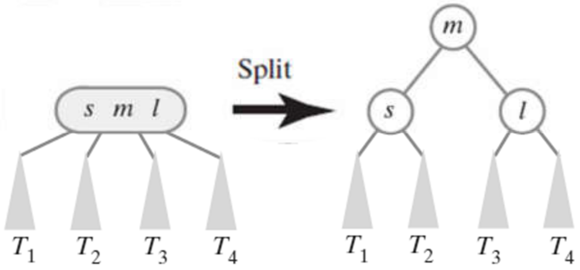
(b) two entries:



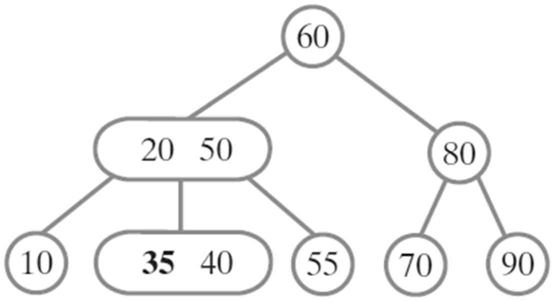
- Splitting an internal node to accommodate a new entry:



- Splitting the root to accommodate a new entry:



Searching a 2-3 Tree:



2-3 tree: performance:

2-3 tree is a perfect balanced tree: Every path from **root** to a **leaf** has same length.

Tree height:

- Worst case: **log N**. [all 2-nodes]
- Best case: **$\log_3 N \approx .631 \log N$**. [all 3-nodes]
    - Between 12 and 20 for a million nodes.
    - Between 18 and 30 for a billion nodes.

2-3 tree: implementation?

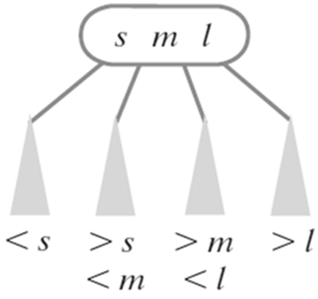
Direct implementation is complicated, because:

- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

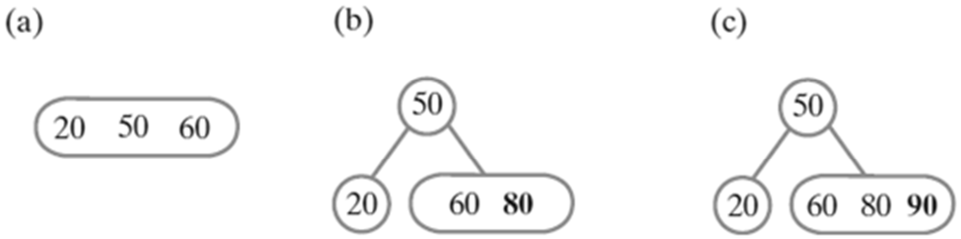
exercise: 50 60 70 40 30 20 10 80 90 100

2-4 Trees

- Sometimes called a 2-3-4 tree.
    - General search tree
    - Interior nodes must have either two, three, or four children
    - Leaves occur on the same level
    - A 4-node contains three data items ***s, m***, and ***l*** and has four children.

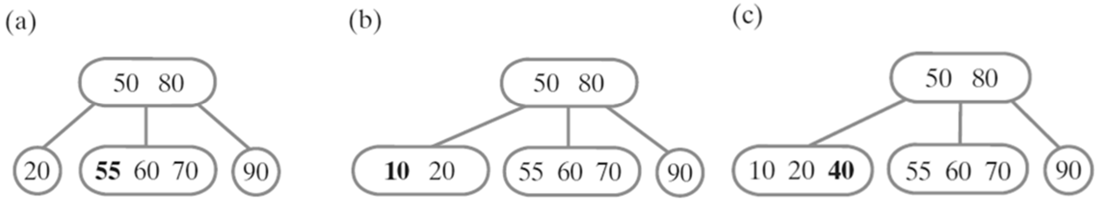


Adding Entries to a 2-4 Tree



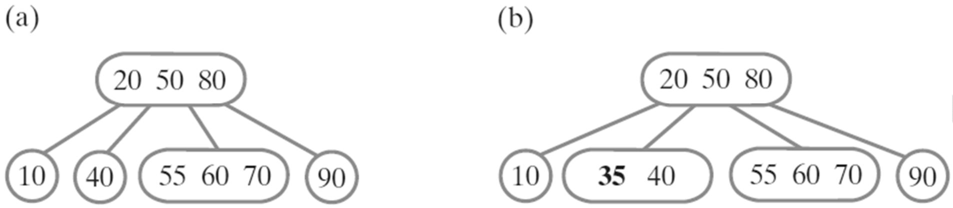
The 2-4 tree, after (a) adding **20**, **50**, and **60** (b) adding **80** and splitting the root; (c) adding **90**

Adding 70



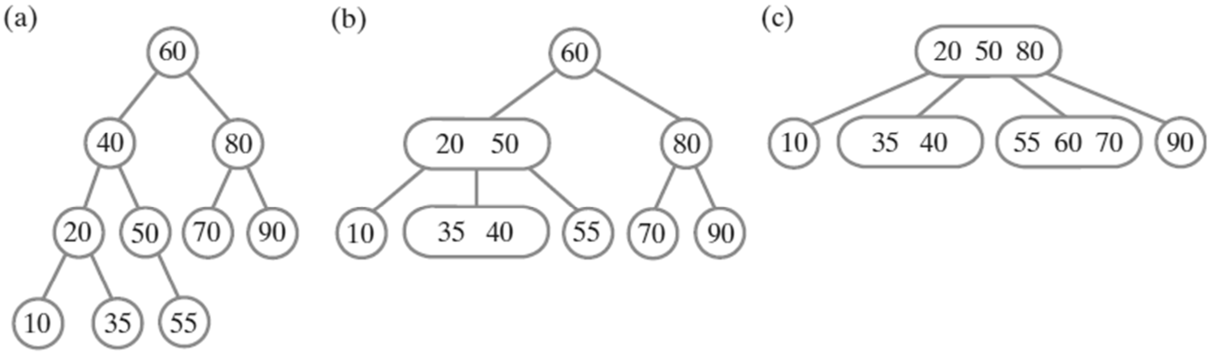
The 2-4 tree after adding (a) **55**; (b) **10**; (c) **40**

Adding 5



The 2-4 tree after (a) splitting the leftmost 4-node; (b) adding **35**

Comparing AVL, 2-3, and 2-4 Trees:



Three balanced search trees obtained by adding 60, 50, 20, 80, 90, 70, 55, 10, 40, and 35:
(a) AVL tree; (b) 2-3 tree; (c) 2-4 tree

B-Trees

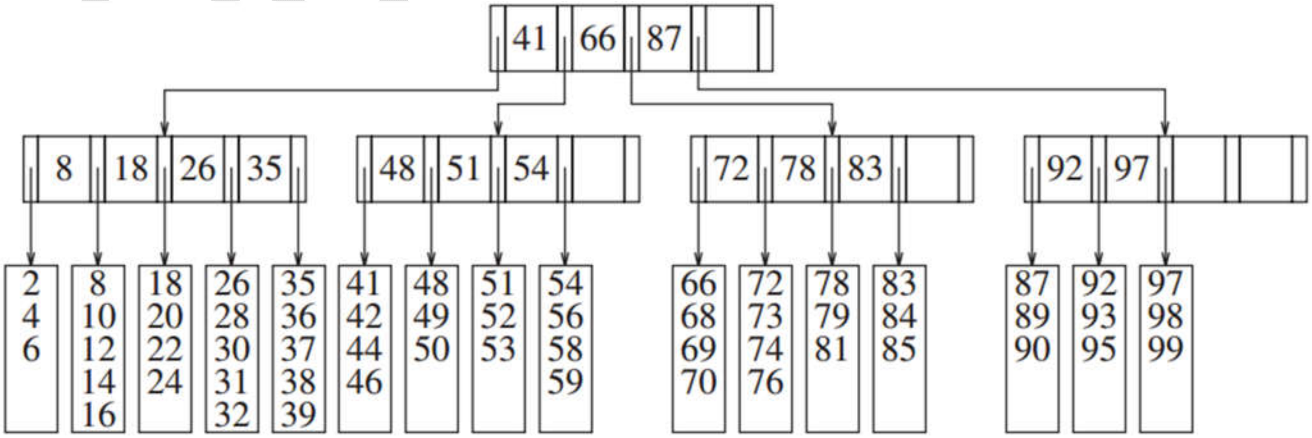
B-trees (Bayer-McCreight, 1972)

- **Definition**: multiway search tree of order ***m***
    - A general tree whose nodes have up to ***m*** children each
- A binary search tree is a multiway search tree of order 2. In a binary search tree, we need one key to decide which of two branches to take. In an M-ary search tree, we need M 1 keys to decide which branch to take.
- 2-3 trees and 2-4 trees are balanced multiway search trees of orders 3 and 4, respectively.
- As branching increases, the depth decreases. Whereas a complete binary tree has height that is roughly $\log_2 N$, a complete M-ary tree has height that is roughly $\log_M N$.
- The B-tree is the most popular data structure for disk bound searching.
- To make this scheme efficient in the worst case, we need to ensure that the M-ary search tree is balanced in some way.
- Additional properties to maintain balance:
    - The **root** has either no children or between **2** and ***m*** children.
    - Other interior nodes (non-leaves) have between $\lceil m/2 \rceil$ and ***m*** children each.
    - All leaves are on the same level.

A B-tree of order ***M*** is an ***M-ary*** tree with the following properties: (**B^+ tree**)

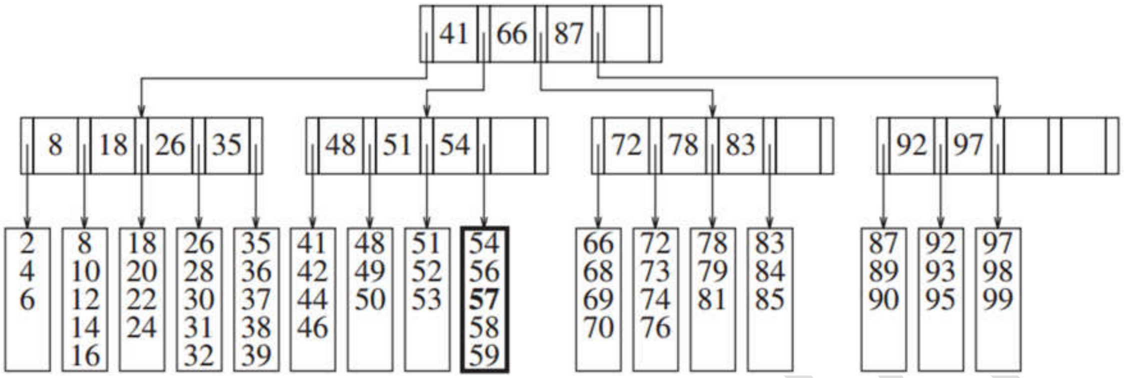
1. The data items are stored at leaves.
2. The non-leaf nodes store up to ***M 1*** keys to guide the searching; key ***i*** represents the smallest key in subtree ***i+1***.
3. The **root** is either a leaf or has between two and ***M*** children.
4. All non-leaf nodes (except the **root**) have between ***M/2*** and ***M*** children.
5. All leaves are at the same depth and have between ***L/2*** and ***L*** data items, for some ***L*** (the determination of L is described shortly).

Example: The following is an example of a B^+ tree of order ***5*** and ***L=5***

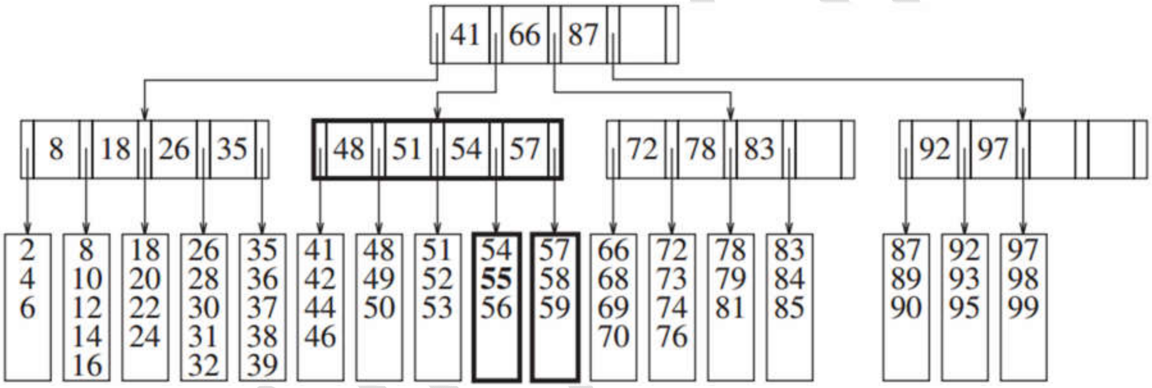


Add items from the B$^+$ tree:

- **Insert 57**: A search down the tree reveals that it is not already in the tree. We can then add it to the leaf as a fifth item:

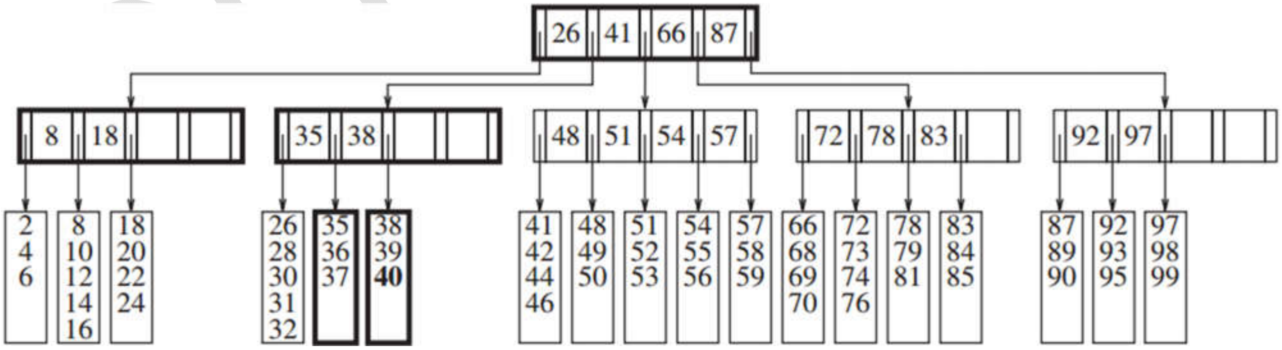


- **Insert 55**: The leaf where 55 wants to go is already full. Solution: split them into two leaves:



Note: The node splitting in the previous example worked because the parent did not have its full complement of children.

- **Insert 40**: We have to split the leaf containing the keys 35 through 39, and now 40, into two leaves.
  - The parent has six children now ➔ split the parent.

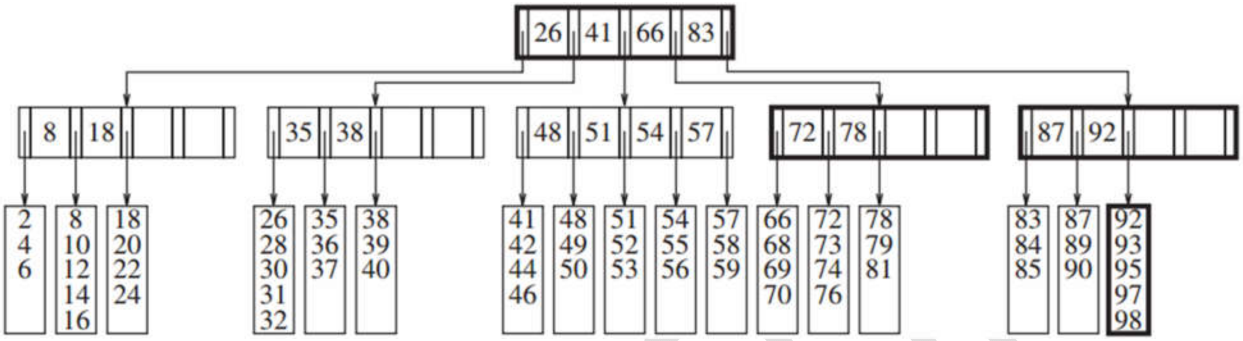


Note:

- When the parent is split, we must update the values of the keys and also the parent's parent.
- if the parent already has reached its limit of children? In that case, we continue splitting nodes up the tree until either we find a parent that does not need to be split or we reach the root. Then we split the root and this will generate a new level.

Remove items from the B^+ tree:

- We can perform deletion by finding the item that needs to be removed and then removing it.
    - The problem is that if the leaf it was in had the minimum number of data items, then it is now below the minimum.
- **Remove 99**: Since the leaf has only two items, and its neighbor is already at its minimum of three, we combine the items into a new leaf of five items.



Splay Trees

Recall: **Asymptotic analysis** examines how an algorithm will perform in worst case.

Amortized analysis examines how an algorithm will perform in practice or on average.

The **90–10 rule** states that **90%** of the accesses are to **10%** of the data items.

However, balanced search trees do not take advantage of this rule.

- The **90–10** rule has been used for many years in **disk I/O systems**.
- A **cache** stores in main memory the contents of some of the disk blocks. The hope is that when a disk access is requested, the block can be found in the main memory cache and thus save the cost of an expensive disk access.
- **Browsers** make use of the same idea: A cache stores locally the previously visited Web pages.

Splay Trees:

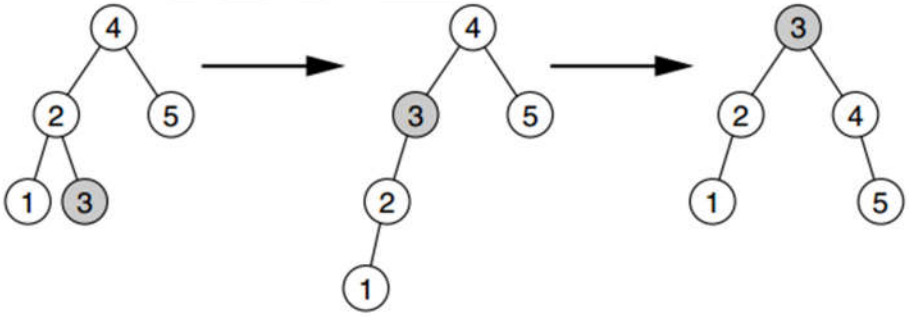
- Like **AVL** trees, use the standard binary search tree property.
- After any operation on a node, make that node the new root of the tree.

A simple self-adjusting strategy (that does not work)

The easiest way to move a frequently accessed item toward the root is to rotate it continually with its parent. Moving the item closer to the root, a process called the **rotate-to-root strategy**.

- If the item is accessed a second time, the second access is cheap.

Example: Rotate-to-root strategy applied when node **3** is accessed



- As a result of the rotation:
    - future accesses of node **3** are cheap
    - Unfortunately, in the process of moving node **3** up two levels, nodes **4** and **5** each move down a level.
- Thus, if access patterns do not follow the **90–10 rule**, a long sequence of bad accesses can occur.

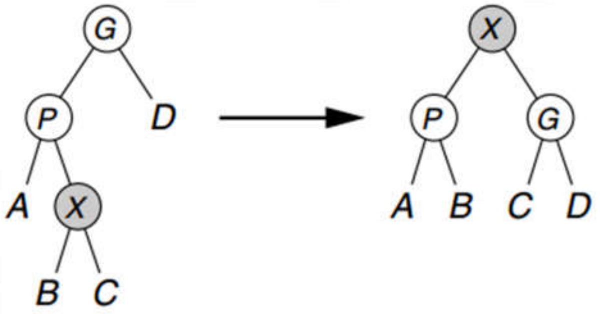
The basic bottom-up splay tree

Splaying cases:

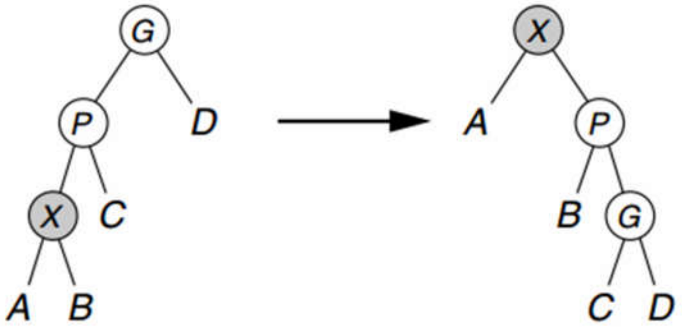
- **The zig case** (normal single rotation)
  If **X** is a non-root node on the access path on which we are rotating and the parent of **X** is the root of the tree, we merely rotate **X** and the root, as shown:

Otherwise, **X** has both a parent **P** and a grandparent **G**, and we must consider two cases and symmetries.

- **zig-zag case**:
  - This corresponds to the inside case for **AVL** trees.
  - Here **X** is a right child and **P** is a left child (or vice versa: **X** is a left child and **P** is a right child).
  - We perform a **double rotation** exactly like an **AVL** double rotation, as shown:

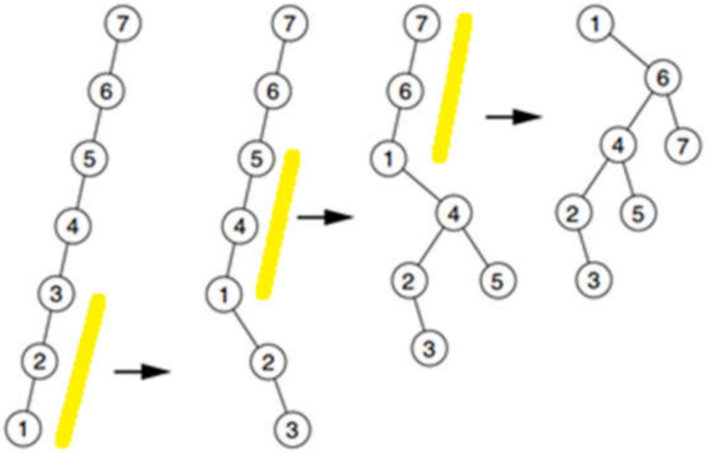


- **zig-zig case:**
  - The outside case for **AVL** trees.
  - Here, **X** and **P** are either both left children or both right children.
  - In this case, we transform the left-hand tree to the right-hand tree (or vice versa).
  - Note that this method differs from the **rotate-to-root strategy**.
    - The **zig-zig** splay rotates between **P** and **G** and then **X** and **P**, whereas the **rotate-to-root strategy** rotates between **X** and **P** and then between **X** and **G**.



Splaying has the effect of roughly **halving** the depth of most nodes on the access path and increasing by at most **two levels** the depth of a few other nodes.

Example: Result of splaying at node **1** (three zig-zigs)



Exercise: perform rotate-to-root strategy

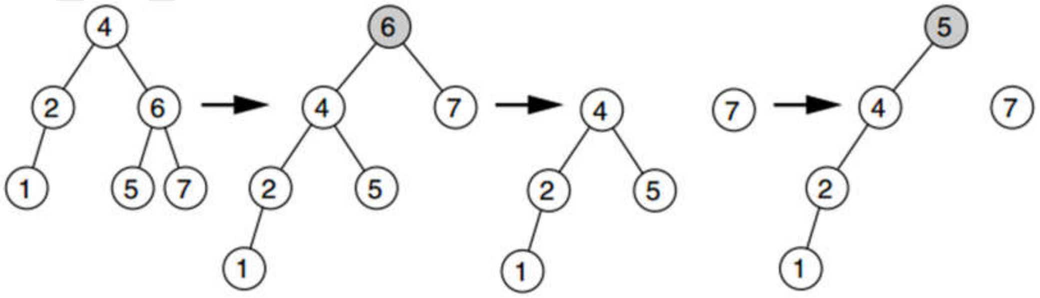
Basic splay tree operations

A splay operation is performed after each access:

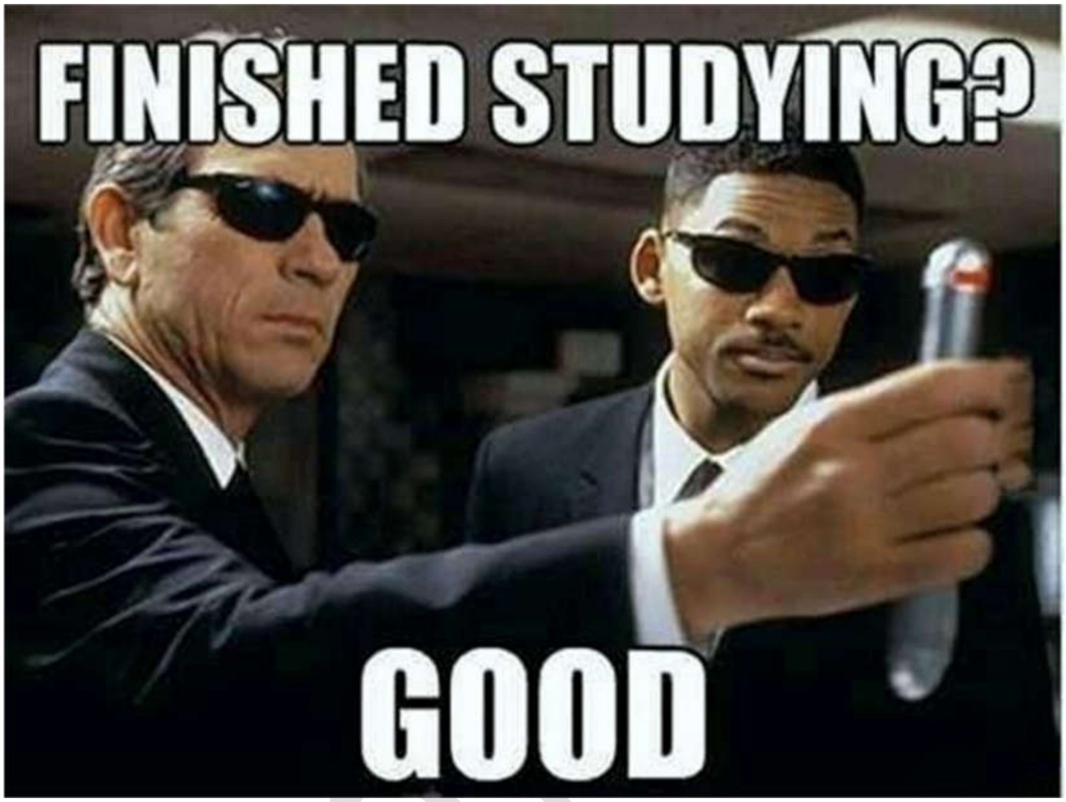
- After an item has been inserted as a leaf, it is **splayed** to the root.
- All searching operations incorporate a **splay**. (**find, findMin** and **findMax**)
- To perform deletion, we access the node to be deleted, which puts the node at the root. If it is deleted, we get two subtrees, **L** and **R** (left and right). If we find the largest element in **L**, using a **findMax** operation, its largest element is rotated to **L**'s root and **L**'s root has no right child. We finish the remove operation by making **R** the right child of **L**'s root. An example of the remove operation is shown below:

  Example: The remove operation applied to node **6**:

  - First, **6** is splayed to the root, leaving two subtrees;
  - A **findMax** is performed on the left subtree, raising **5** to the root of the left subtree;
  - Then the right subtree can be attached (not shown).



- The cost of the remove operation is **two splays**.


FINISHED STUDYING?
GOOD