# 7SENG010W Data Structres & Algorithms

## Week 7 Lecture

## Balanced Trees

# Overview of Week 7 Lecture: Balanced Trees

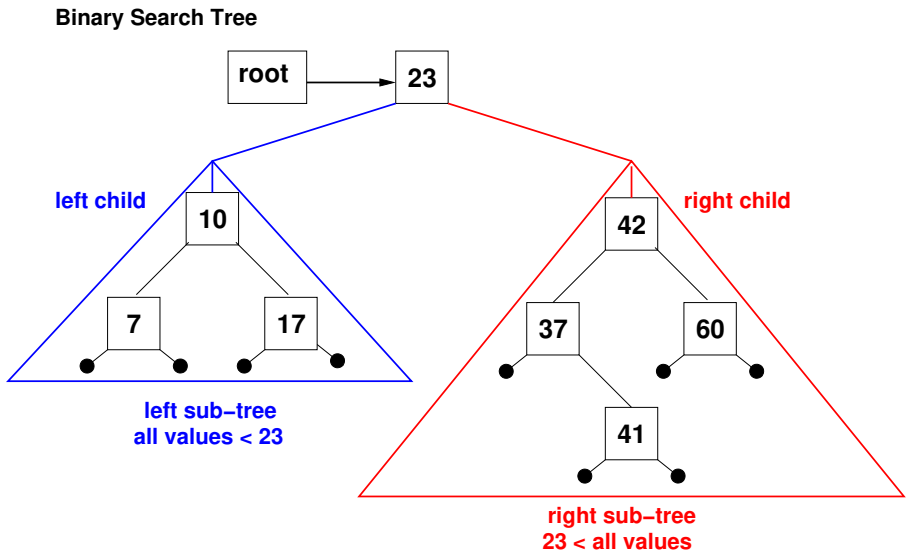
- *Recap of BSTs & Tree Traversal*
- *Tree Properties*
  - Node Level
  - Tree Height
  - Balanced & Unbalanced Trees
- *AVL Trees – Balanced BST*
  - *Definition of* an AVL Tree
  - AVL Tree *Operations*
  - AVL Tree – Rebalancing "*Rotation*" operations
- *B-Trees – N-ary Balanced Trees*

# PART I

## *Recap of BSTs & Tree Traversal*

# Recap: Binary Search Trees (BST) (1/2)

Remember the BST from the previous lecture:




It represents a set of *keys* & their associated *values*.

The characteristic *BST property* is that for all **node keys** in the tree:

1. all keys in each node's *left sub-tree* appear to the *left* of the node, & are **"less than or equal to"** the node's key value;
2. all keys in each node's *right sub-tree* appear to the *right* of the node, & are **"greater than or equal to"** the node's key value;
3. then we **always get the keys in sorted order**.

# Recap: Binary Search Trees (BST) (2/2)

- In addition, the BST *insert* & *delete* a value operations (& any other operations that modify the BST) **must** maintain the *BST property*.

- Complexity of BST operations:
  *insert* an item into the tree, *delete* an item from the tree, *search* for an item in the tree,
  all with *worst-case* complexity of $O(N)$, but with the *average-case* complexity of $\Theta(\log_2(N))$ & *best-case* complexity of $\Omega(1)$.

- We also looked at 3 tree *Traversal orders*, using the previous tree:

  | | |
  |---|---|
  | *In-order (L,N,R):* | 7, 10, 17, 23, 37, 41, 42, 60 |
  | *Pre-order (N,L,R):* | 23, 10, 7, 17, 42, 37, 41, 60 |
  | *Post-order (L,R,N):* | 7, 17, 10, 41, 37, 60, 42, 23 |

This week, we will look at the *properties of trees* & "*balanced trees*", such as *AVL trees* & *B-trees*.

# PART II

## *Properties of Nodes & Trees:*

## *level, height, balanced*

# Node & Tree Properties

Before we consider what a **balanced** tree is we need to define some properties of nodes & trees[9]:

- the **level** of a node,
- the **height** (or **depth**) of a tree.

### Definition: Level of a Node

The **level** of a node is the *"distance"* from the **node** to the **root** of the tree, i.e. the *number of branches*.

The level of the root is 0, as no branches are traversed.
The level of a node is *one plus the level of its parent*, i.e. the additional branch between the child node & parent node.

E.g. So the depths of the root's child nodes is 1, its grandchild nodes is 2, its great-grandchildren is 3, ...

[9] These definitions also work for a sub-tree of a tree, i.e. a root node other than the tree's root node.

# Tree Properties: Level & Height

**Definition: Level of a Tree**

The **level** of a tree is *all the nodes of a tree with the same level*.

So *level k* of a tree is all the nodes in the tree with *level k*.

Level 0 of a tree are the level 0 nodes, which is just the root node.
Level 1 of a tree are all the child nodes of the root.
Level 2 of a tree are all the grandchildren nodes of the root. Etc.

**Definition: Height of a Tree:**

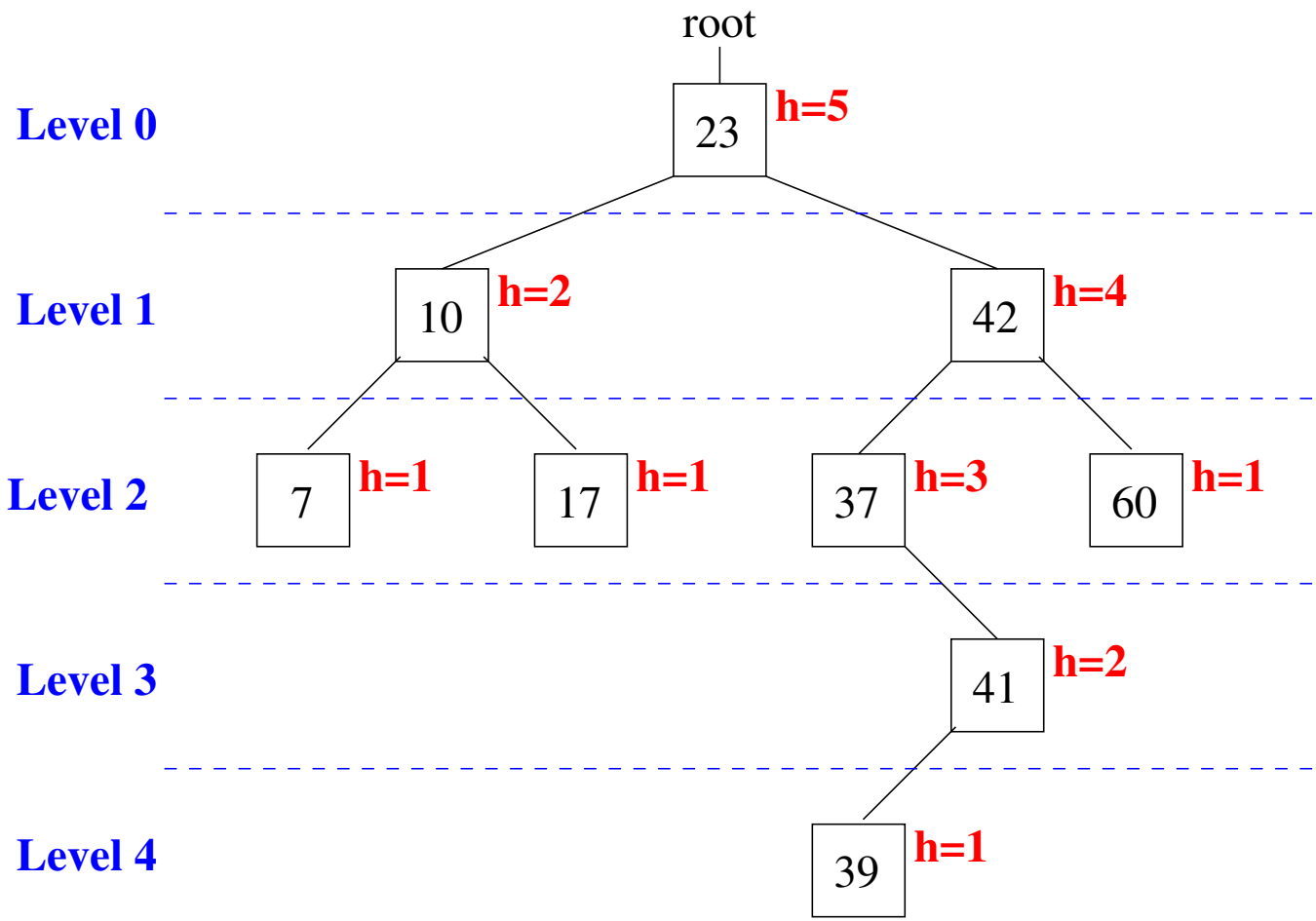
The **height** (or **depth**) of a tree is the *number of nodes on the longest path from the root to a leaf*.

$$height(tree) \ = \ 1 + max(\ height(leftSubTree),\ height(rightSubTree\ )\ )$$

The height of an **empty tree** (root equals `null`), is 0.

The height of a tree with: just the root is 1; a root & a child is 2; a root, a child & a grandchild is 3, etc.

# Diagram of Tree & Node Properties




Tree labelled with the properties: nodes with the **height** of the tree with it as the root, and the **nodes levels**.

# Tree Node Levels Algorithm

Based on one way to traverse a BST, the pseudo code of one way to print out the levels of all the nodes in a BST is:

```
printNodeLevels( root, level ):

  IF   ( root equals null )
  THEN
       nothing to print out
  ELSE
       PRINT root, level

       // process sub-trees
       printNodeLevels( left sub-tree,  level + 1 )

       printNodeLevels( right sub-tree, level + 1 )
  ENDIF
```

# Tree Height Algorithm

Based on post-order traversal of a BST, one way to calculate the height of a BST is given by following pseudo code:

```
TreeHeight( root ):

  IF   ( root equals null )
  THEN
       RETURN 0 ( what if return -1)
  ELSE
       // get heights of sub-trees

       leftHeight =  TreeHeight( left sub-tree )

       rightHeight = TreeHeight( right sub-tree )

       RETURN 1 + max( leftHeight, rightHeight )
  ENDIF
```

**Question:** why does it make sense to base it on post-order traversal, rather than one of the others?

# A Balanced Tree

Now we have covered the basic properties of nodes & trees we can look at what it means for a tree to be *balanced*.

**Definition: Balanced Tree**

A binary tree is **balanced** (or **height-balanced**)

if, and only if,

for each **node** in the tree,

the **heights** of its *left* and *right sub-trees* **differ by at most one**.

**Notes:** The "**differ by at most one**" is calculated using the **absolute value** of the difference, e.g. $|4 - 9| = |-5| = 5$, $|42 - 12| = |30| = 30$.

So a **tree is balanced** if & only if:

$$| \, height(leftSubTree) - height(rightSubTree) \, | \; \leq \; 1$$

# Points about Balanced Trees

An **empty tree** is **balanced**.

A tree with *just 1 node* (the root) or *2 nodes* is **balanced**.

But *any tree with 3 or more nodes* can be **unbalanced**.

If for a node one of its sub-trees is empty, then the other sub-tree must either be empty or consist of just a leaf.

So if there is *at least one node in a tree* such that:

$$height(leftSubTree) = hlst$$
$$height(rightSubTree) = hrst$$

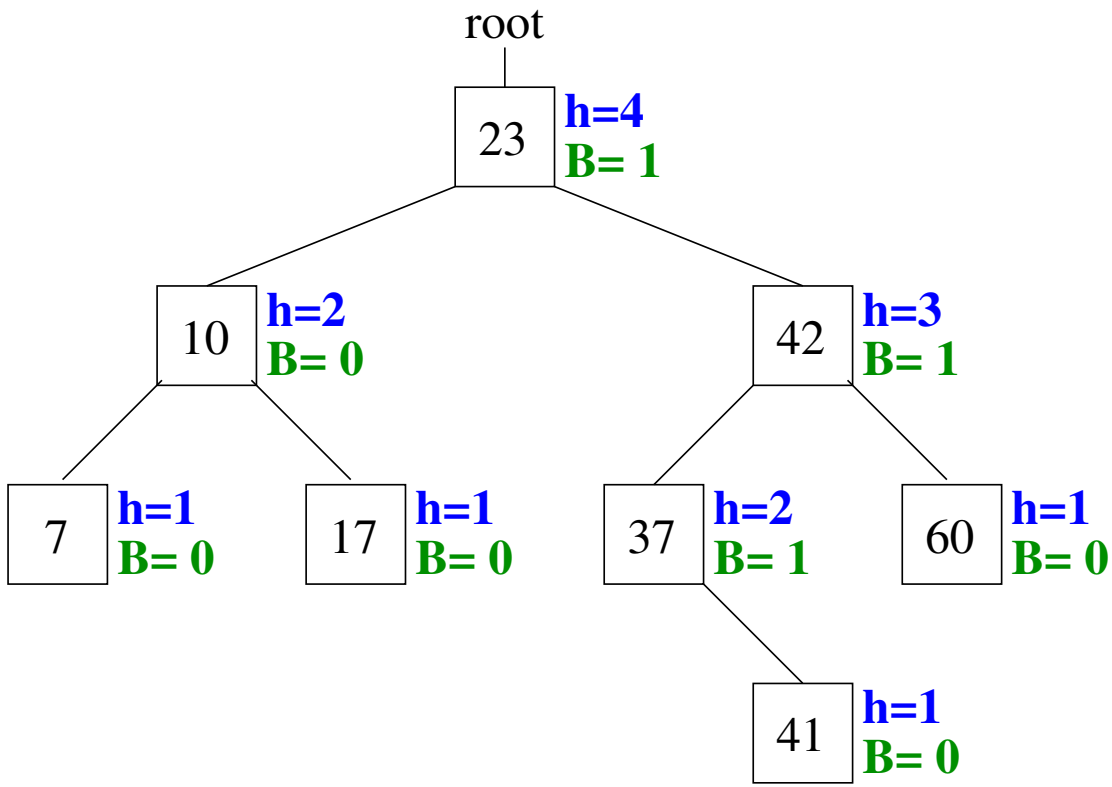
& the difference between $hlst$ & $hrst$ is more than 1,

then the **tree is not balanced**.

Assume that a BST contains $K$ nodes, and that it is **balanced**, then the number of **levels** will be approximately $log_2(K)$.

So if $K = 100$ then an **insert** operation will take approximately at most $log_2(100) = 6.6$, i.e. 7 comparisons.

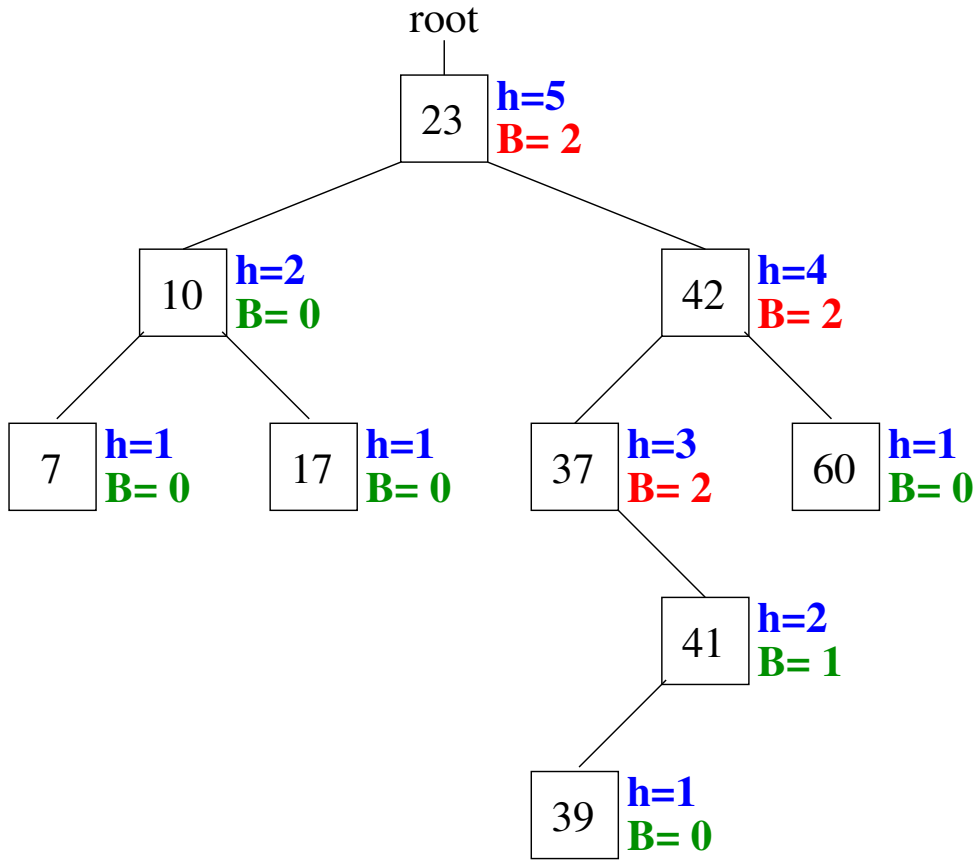
# Balanced BST Trees




Each node is labelled with the **height**, e.g. **h=3** of the tree with that node as the root, and the **balance** value: **B=0, B=1** indicates the tree is **balanced**, but a value of **B=2** or more indicates the tree or sub-tree is **not balanced**.

This tree is **balanced** because all of its nodes are balanced trees, i.e. **B=0** or **B=1**.

# Unbalanced BST Trees




This tree is an **unbalanced** tree because the trees for the 3 nodes: 37, 42 and 23 have **B=2**, because the difference in height of their sub-trees is 2, and therefore they are **unbalanced**.

$$\begin{aligned}
\mathbf{B}(37) = &\quad |height(\texttt{null}) - height(41)| &= |0-2| = \mathbf{2} \\
\mathbf{B}(42) = &\quad |height(37) - height(60)| &= |3-1| = \mathbf{2} \\
\mathbf{B}(23) = &\quad |height(10) - height(42)| &= |2-4| = \mathbf{2}
\end{aligned}$$

# Why we want Balanced Trees

However, there is **no guarantee** that a BST will be *balanced*.

Because the *structure* of the tree, e.g. its *height*, is determined by the *order of arrival of new keys*, irrespective of the algorithm which built the tree.

The running times of algorithms on BSTs depend on the *shapes of the trees*, which, in turn, depend on the *order in which keys are inserted*.

*Worst scenario* for a tree is when values are inserted in *ascending (or descending) order*, the tree will be a *linear list* of *N* nodes, **not a tree**.

Inserting a new value would be a linear function $O(N)$ of the number of keys already in the tree rather than a logarithmic one $O(\log_2(N))$.

The **best-case** for a tree with *N* nodes would be *perfectly balanced*, with approximately $\log_2(N)$ nodes between the root and each `null` link.

In a typical **average-case** tree, it is usually much closer to being a *balanced* tree than the **worst case** of a *list*.

**Example:** in the case with a totally unbalanced tree with 100 nodes, inserting the $101^{st}$ value into it, could take up to a 100 comparisons, i.e. getting to the end of the list, versus 7 for a balanced tree. (Similarly for delete & search.)

# Advantages of Balanced Trees

The *order of complexity* for operations on types of BST are:

| **BST Tree Type** | *Average-case* $\Theta(-)$ | *Worst-case* $O(-)$ |
|---|---|---|
| Unbalanced Trees | $\Theta(\ \log_2(N)\ )$ | $O(N)$ |
| Balanced Trees | $\Theta(\ \log_2(N)\ )$ | $O(\ \log_2(N)\ )$ |

In summary, the overall *best performance* for the insert, delete & search operations is *logarithmic*, when the *tree is balanced*.

Generally, the efficiency of operations on trees is *improved when a tree is balanced*.

For a given number of nodes the *better balanced* a tree is, the *lower its height*, & the *more efficient* are the operations on the tree.

A number of algorithms exist for balancing BSTs.

So, the advantage of using a *balanced* tree over an *unbalanced* tree is that the **worst-case** complexity is the same as the **average-case** complexity, i.e. it goes from $O(N)$ to $O(\ \log_2(N)\ )$.

We shall now look at one type of *balanced BSTs* known as *AVL trees*.

# PART III

## *AVL Trees*

## *Balanced Binary Search Trees (BST)*

# AVL Trees

An **AVL tree** is an example of a *balanced binary search tree*.

Named after its inventors *Georgii M. **A**delson-**V**elskii* and *Evgenii M. **L**andis* (1962).

AVL trees were the first dynamically balanced trees to be proposed.

**NOTE: Every AVL tree** is a BST, but **not every BST** is an AVL tree.

They are *not perfectly balanced*, because pairs of sub-trees are allowed to differ in height by at most 1, if they differ by 2 or more then the tree is **not balanced**.

**Definition: AVL Tree**
An AVL tree is a *binary search tree* (BST) which has the following properties:

1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.

So the **balance property** for an AVL tree is that:
*the left and right sub-trees differ by at most 1 in height*.

# AVL Tree: Insertion

AVL tree insertion requires the consideration of several issues.

*Inserting a value* into an AVL tree is performed in three stages:

1. **Insert the new node into the AVL tree as a leaf node**, in exactly the same way as is done for a BST.

2. **Check if the AVL tree has become unbalanced** as a result of inserting the node.

   Check the *path* from the *inserted node* back up to the *root*, to make sure all of the nodes on the path are still *balanced*.

3. If the **AVL tree is unbalanced**, i.e. a node on the path is **not balanced**, then **rebalance the AVL tree** with that node as its root.

   Then continue up to the root.

   Otherwise, node insertion is finished.

The AVL tree *deletion* operation uses similar actions, but uses BST deletion, so we will only consider *insertion*.

Both these operations require additional information – a node's *balance factor (BF)* & "*rotation*" operations used to *rebalance* the tree.

# AVL Tree: Balance Factor (BF) (1/2)

Implementations of **AVL trees** add an extra attribute to each node, that relates to its role as the root of a tree, usually either:

- the *balance factor (BF)* of the node, &/or
- the *height* of the node.

This information is used to determine if the tree with the node as its root is **balanced** or not, by allowing its *balance factor (BF)* to be calculated.

Where the *balance factor for a node* in an AVL tree is given by:

$$BF(node) \;=\; height(leftSubTree) - height(rightSubTree)$$

Its value can be either *negative* $(\ldots, -2, -1)$, *zero* $(0)$ or *positive* $(1, 2, \ldots)$.

Then an AVL tree is **balanced** if *all its nodes satisfy*:

$$-1 \;\leq\; BF(node) \;\leq\; 1, \qquad\qquad (BF(node) \text{ equals } -1,\ 0,\ 1)$$

Or the AVL tree is **unbalanced** if *one or more nodes satisfy*:

$$BF(node) \;\leq\; -2 \;\; or \;\; 2 \;\leq\; BF(node) \qquad (BF(node) \text{ equals } \ldots, -3, -2, 2, 3, \ldots)$$

# AVL Tree: Balance Factor (BF) (2/2)

The value of the *balance factor* of a *node* in an AVL tree indicates the "*balance status*" of the *tree with that node as its root*.

The "*balance status*" is indicated by the value of $BF(node)$ as follows:

$-k$ "*right-heavy*" — height of the node's right sub-tree is $k$ greater than the left sub-tree,

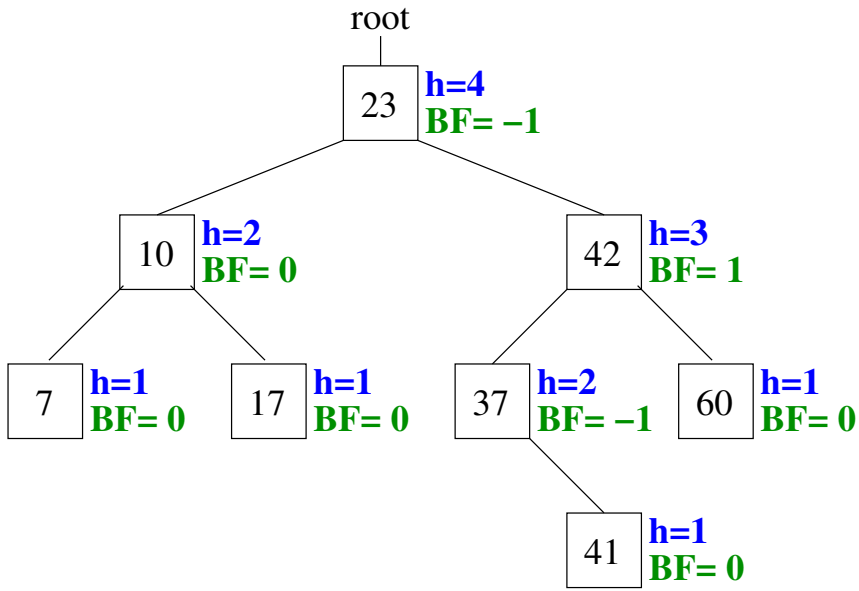
$0$ "*balanced*" — both of the node's sub-trees are the same height,

$k$ "*left-heavy*" — height of the node's left sub-tree is $k$ greater than the right sub-tree.

Note that an AVL tree can be *left-heavy* ($BF(node) = 1$) or *right-heavy* ($BF(node) = -1$) & still be **balanced**.

An AVL tree can become **unbalanced** ($BF(node) \leq -2$ or $2 \leq BF(node)$) because of an insertion or deletion, then at least one "*rotation*" operation must be performed to **rebalance** the tree.

# AVL Tree: Insertion – Balanced Tree

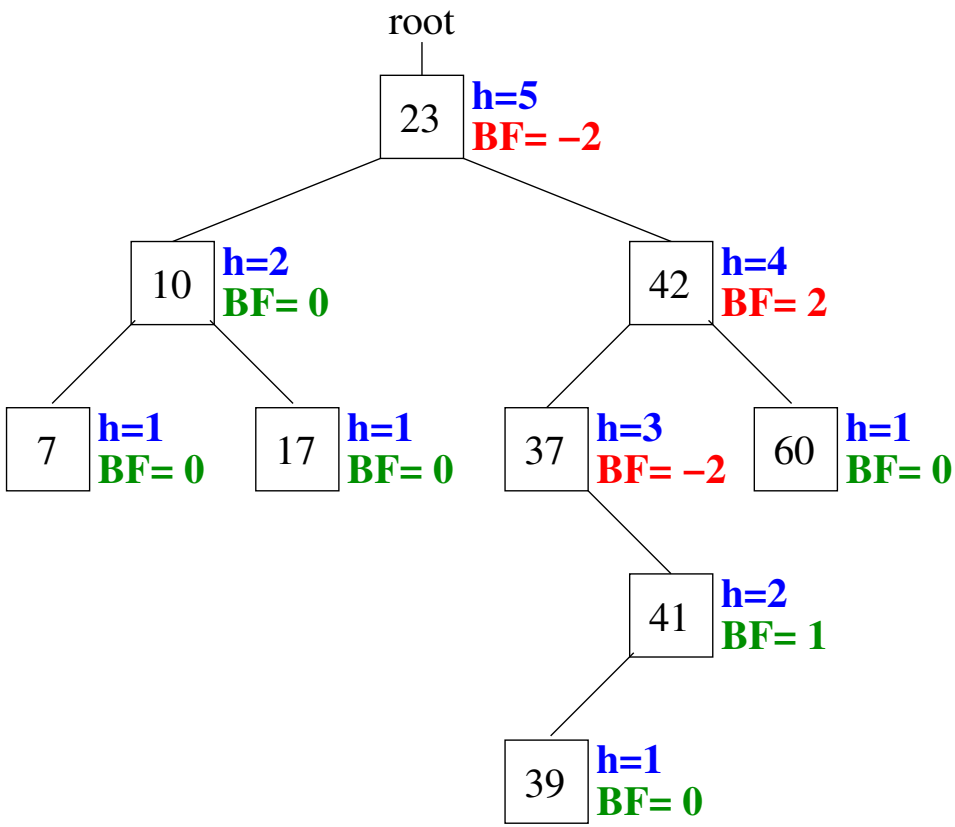



In this AVL tree assume $41$ has just been inserted into the tree, which is added as the right sub-tree of $37$, resulting in the tree remaining **balanced**.

Because none of the nodes on the *path* from $41$ to $23$, are **unbalanced**:
$BF(41) = height(\texttt{null}) - height(\texttt{null}) = 0 - 0 = 0$
$BF(37) = height(\texttt{null}) - height(41) = 0 - 1 = -1$.
$BF(42) = height(37) - height(60) = 2 - 1 = 1$.
$BF(23) = height(10) - height(42) = 2 - 3 = -1$.

The tree is still **balanced**, so the insertion operation is completed, but it is *right-heavy*.

# AVL Tree: Insertion – Unbalanced Tree




In this AVL tree assume $39$ has just been inserted into the tree, which is added as the left sub-tree of $41$, resulting in the tree becoming **unbalanced**.

This is indicated by finding node $37$ on the path up to the root with:

$$BF(37) = height(\texttt{null}) - height(41) = 0 - 2 = -2.$$

So now the **tree with root 37 must be rebalanced**.

# AVL Tree Operation: Rebalancing – Rotations

Insertions and deletions on an AVL tree can destroy the *balanced* property.

So we need a way to restore the balanced property, this is done using **rotations**.

**Definition: Rotation**

A **rotation** is a local operation in a search tree that switches children and parents among two or three adjacent nodes to restore balance to a tree.

A **rotation**:

1. May change the *height of some nodes*, usually it reduces the height by 1.
2. Restores the *balance property*, i.e. $BF(root)$ is $-1, 0, 1$.
3. Preserves *In-order traversal of key ordering*, i.e. the *BST property*.

There are 4 AVL tree **rotation** operations:

*Left, Right, Left-Right, Right-Left* .

We first look at the single rotations: *Left* and *Right*, then the combination rotations: *Left-Right* and *Right-Left*.

# Notes on the example AVL Trees

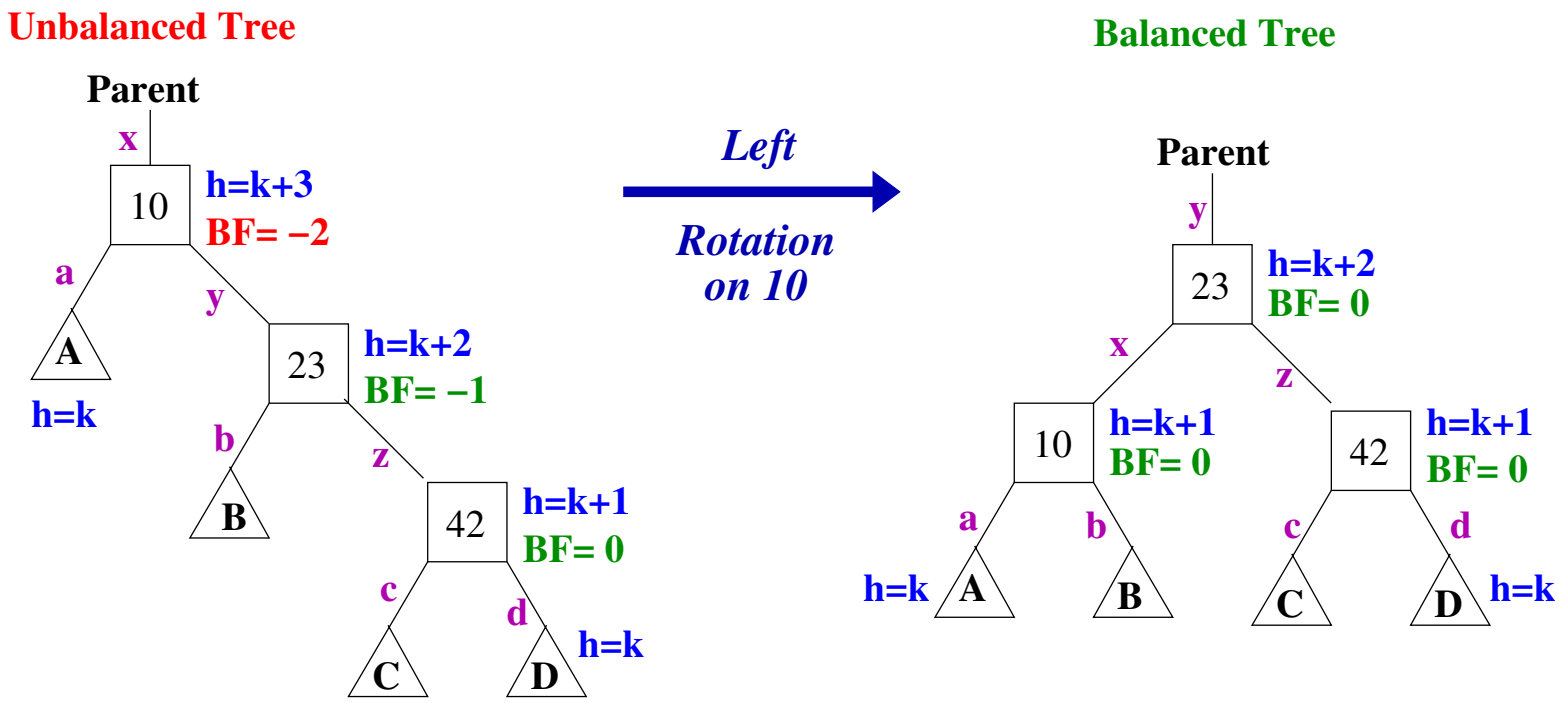
In the following example AVL trees we will use:

- The **Parent** represents the parent node of the root node of the example AVL tree, this could be null if the example's root was also the root of the whole tree.
- Four sub-trees A, B, C & D, each will be assumed to be **balanced (BF = 0)** AVL trees & all of them have the same height **k**.
- The following *BST property* ordering relationship holds between the numbers 10, 23, 42 & all the values in each of the four sub-trees A, B, C & D:

$$A < 10 < B < 23 < C < 42 < D$$

- Links **a**, **b**, **c**, **d** will always point to the sub-trees A, B, C & D, respectively.
- Links **x**, **y**, **z** will always point to the nodes 10, 23 & 42, respectively.

# AVL Tree Balance Operation: Left Rotation (1/2)




This tree is **unbalanced** because node 10 has **BF(10) = -2**, so it must be rebalanced, in this case using the *Left rotation* operation.

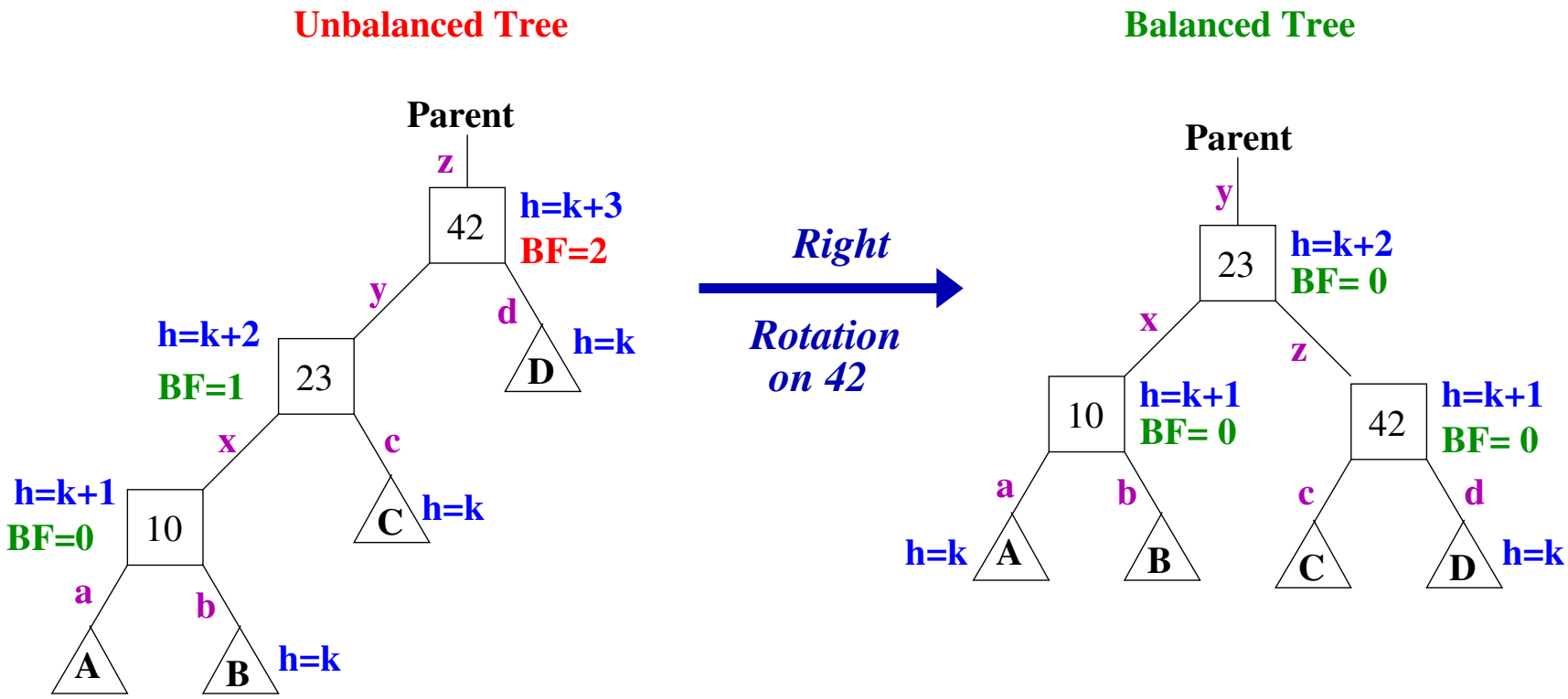
*Left rotation* rebalances a tree by moving every node in the *unbalanced tree* one position to the left from its current position, i.e. "*rotates them to the left*".

# AVL Tree Balance Operation: Left Rotation (2/2)

For this tree the effect is that:

- 23 becomes the new root.
- The **parent** node now points to 23, via **y**.
- 10 becomes the left sub-tree of 23, via **x**.
- 23's left sub-tree B becomes 10's right sub-tree, via **b**.
- The tree is balanced, i.e. **BF(10) = 0, BF(23) = 0, BF(42) = 0**.
- Also note that correct **in-order** traversal maintains **sorted order** in the balanced tree.
- The tree's height has been reduced by 1, from **k+3** to **k+2**, as now the node with the greatest height is 23, $height(23) =$ **k+2**, whereas before it was 10 with $height(10) =$ **k+3**.

# AVL Tree Balance Operation: Right Rotation (1/2)




This tree is **unbalanced** because node 42 has **BF(42) = 2**, so it must be rebalanced, in this case using the *Right rotation* operation.

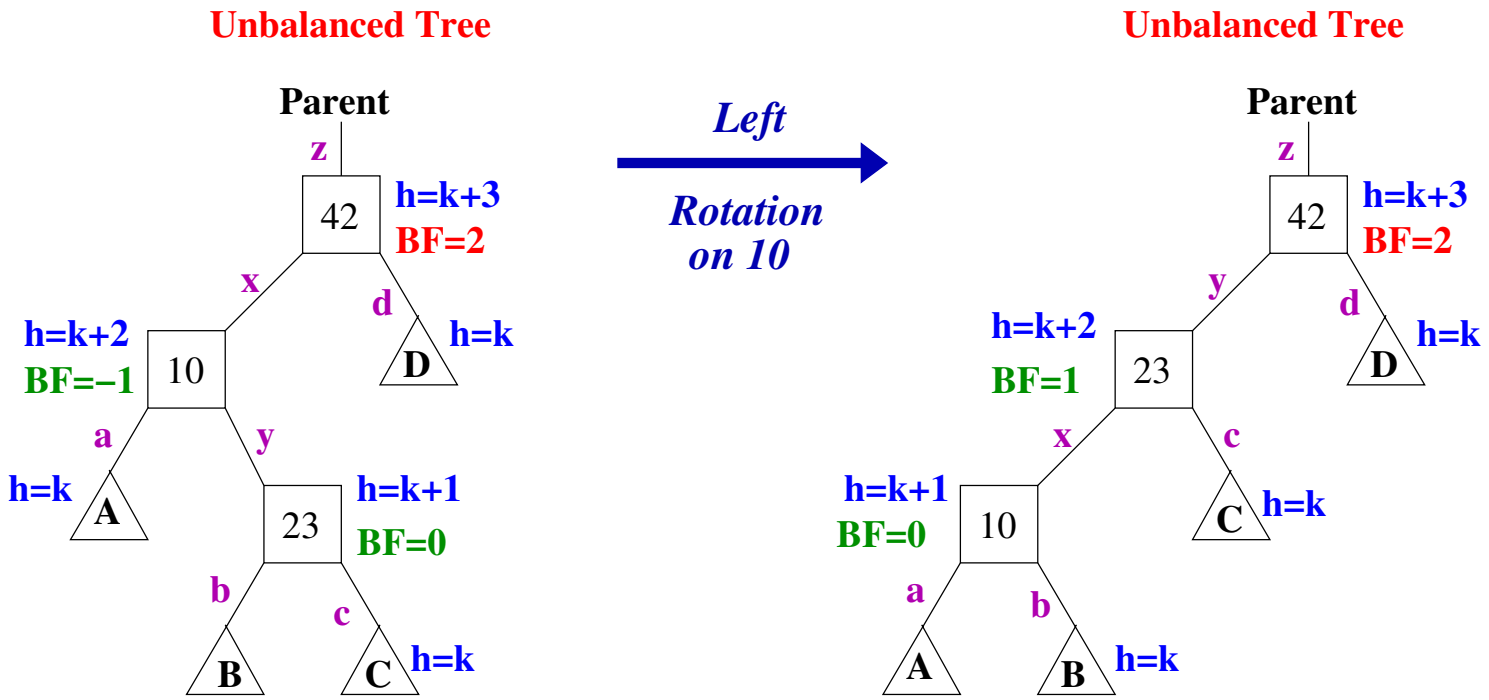
*Right rotation* rebalances a tree by moving every node in the unbalanced tree one position to the right from its current position, i.e. "*rotates them to the right*".

# AVL Tree Balance Operation: Right Rotation (2/2)

For this tree the effect is that:

- 23 becomes the new root.
- The **parent** node now points to 23, via **y**.
- 42 becomes the right sub-tree of 23, via **z**.
- 23's right sub-tree C becomes 42's left sub-tree, via **c**.
- The tree is balanced, i.e. **BF(10) = 0, BF(23) = 0, BF(42) = 0**.
- Also note that correct **in-order** traversal maintains **sorted order** in the balanced tree.
- The tree's height has been reduced by 1, from **k+3** to **k+2**, as now the node with the greatest height is 23, $height(23) =$ **k+2**, whereas before it was 42 with $height(42) =$ **k+3**.

# AVL Tree Balance Operation: Left-Right Rotation (1/3)




This tree is unbalanced since **BF(42) = 2**, so it must be rebalanced, but we need to perform two rotations to achieve this.

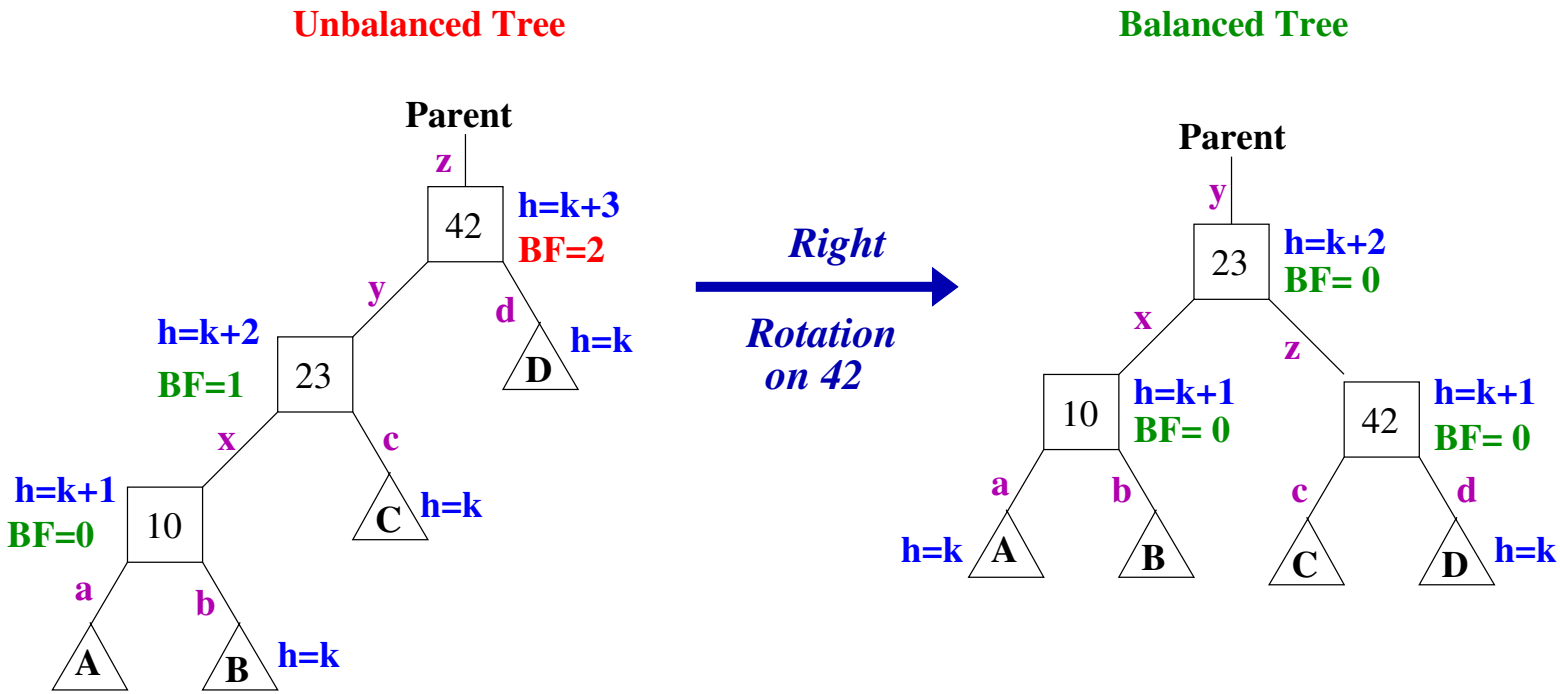
1. Perform a *Left rotation* on the sub-tree with root 10.
2. But this still results in an **unbalanced** tree, but now we can rebalance it by performing a *Right rotation*.

# AVL Tree Balance Operation: Left-Right Rotation (2/3)

The effect of performing the *Left rotation* on node 10 is that:

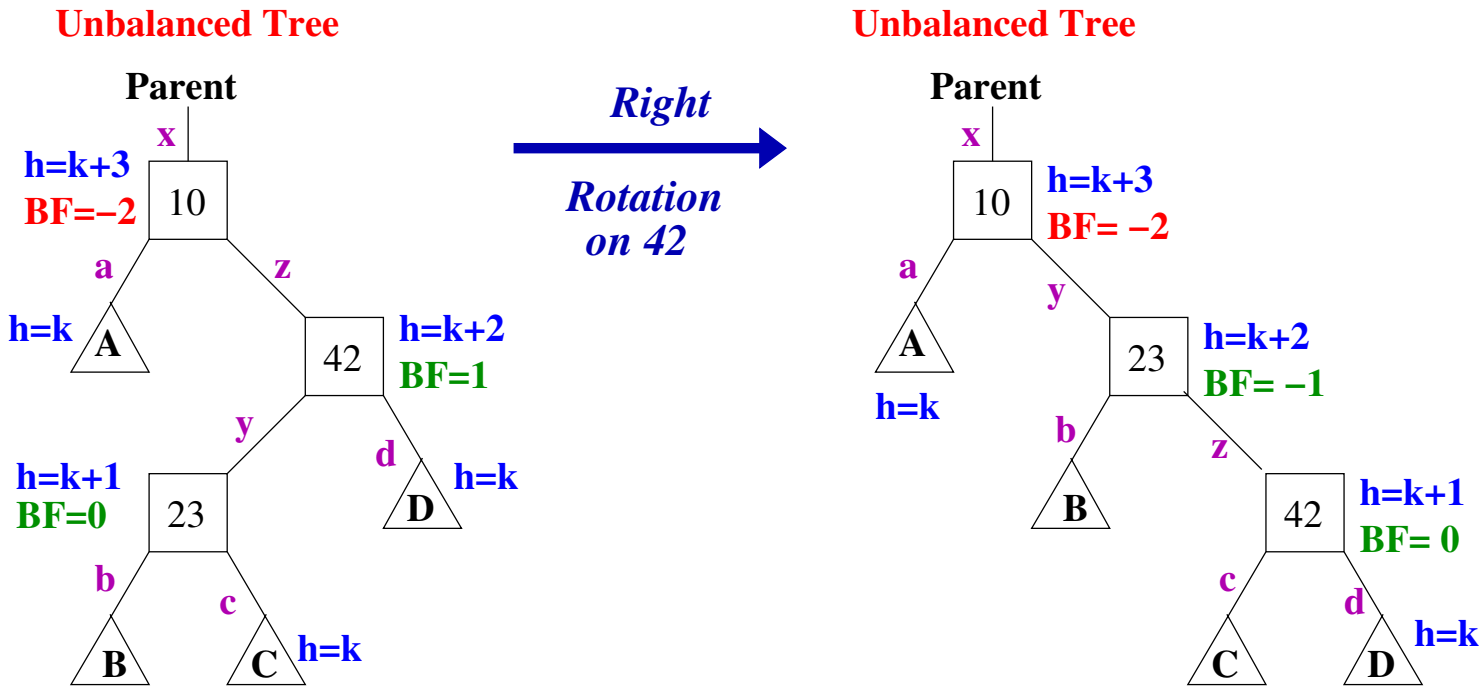
- 42 remains the root & the **parent** node still points to it, via **z**.
- 23 becomes 42's new left sub-tree, via **y**.
- 10 becomes the left sub-tree of 23, via **x**.
- 23's left sub-tree B becomes 10's right sub-tree, via **b**.
- The tree is still **unbalanced** because **BF(42) = 2**.
- Correct **in-order** traversal maintains **sorted order** in the **unbalanced** tree.
- The tree's height is still **k+3**, since $height(42) =$ **k+3**.

# AVL Tree Balance Operation: Left-Right Rotation (3/3)




3. Now perform a *Right rotation* on the tree with root 42.
4. This results in a **balanced** tree, with root 23, left child 10 & right child 42.
5. The balance factor for all three nodes is now **BF(10) = 0, BF(23) = 0, BF(42) = 0**, & its height has been reduced by 1 to **k+2**.
6. **In-order** traversal maintains **sorted order** in the balanced tree.

# AVL Tree Balance Operation: Right-Left Rotation (1/3)




This tree is unbalanced since **BF(10) = -2**, so it must be rebalanced, but we need to perform two rotations to achieve this.

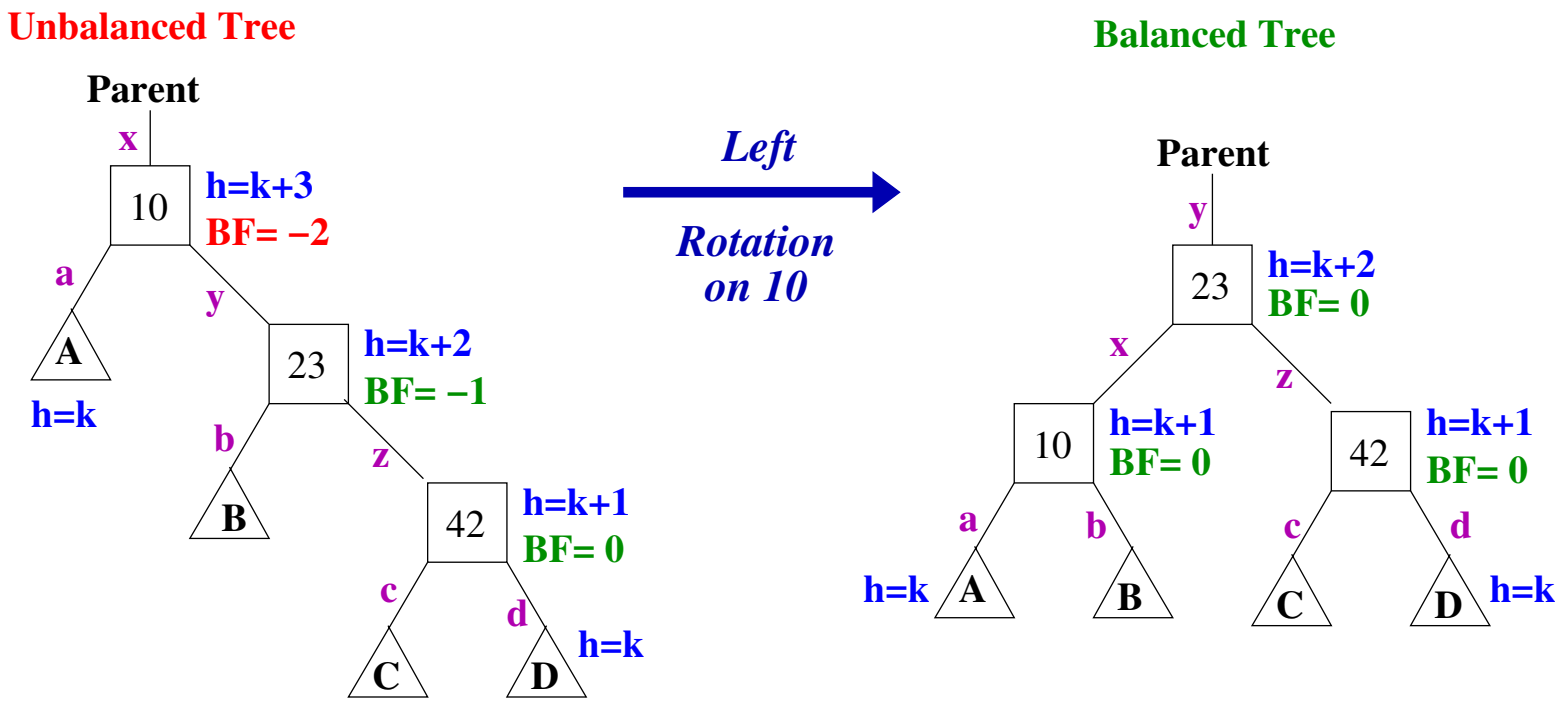
1. Perform a *Right rotation* on the sub-tree with root 42.
2. But this still results in an unbalanced tree, but now we can rebalance it by performing a *Left rotation*.

# AVL Tree Balance Operation: Right-Left Rotation (2/3)

The effect of performing the *Right Rotation* on node 42 is that:

- 10 remains the root & the **parent** node still points to it, via **x**.
- $23$ becomes 10's new right sub-tree, via **y**.
- $42$ becomes the right sub-tree of $23$, via **z**.
- 23's right sub-tree C becomes 42's left sub-tree, via **c**.
- The tree is still **unbalanced** because **BF(10) = -2**.
- Correct **in-order** traversal maintains **sorted order** in the **unbalanced** tree.
- The tree's height is still **k+3**, since $height(10) =$ **k+3**.

# AVL Tree Balance Operation: Right-Left Rotation (3/3)




3. So now perform a *Left rotation* on the tree with root 10.
4. This results in a **balanced** tree, with root 23, left child 10 & right child 42.
5. The balance factor for all three nodes is now **BF(10) = 0, BF(23) = 0, BF(42) = 0**, & its height has been reduced by 1 to **k+2**.
6. **In-order** traversal maintains **sorted order** in the balanced tree.

# How to decide which Rotations to apply

Question: What determines which Rotations to apply to an **unbalanced** node?

Answer: The balance factors (**BFs**) of the unbalanced node & either its left child node or its right child node depending on their signs $(-/+)$ and values $-2, -1, 0, 1, 2$.

The table shows which rotations to apply based on the combinations of balance factors (**BFs**) of the **unbalanced** root & its left & right children:

| **Root** | **Left Child** | **Right Child** | **Rotations** |
|---|---|---|---|
| **-2** | n/a | **-1, 0** | Left-Rotation( root ) |
| **-2** | n/a | **1** | Right-Rotation( rightChild ),<br>Left-Rotation( root ) |
| **2** | **1, 0** | n/a | Right-Rotation( root ) |
| **2** | **-1** | n/a | Left-Rotation( leftChild ),<br>Right-Rotation( root ) |

# Pseudo code: Applying AVL Rotations

Sample pseudo code for determining which *AVL rotation* should be applied to an **unbalanced** AVL tree's root node, i.e. when $BF(\text{root}) = -2 \; or \; 2$.

```
ApplyRotation( NODE root )
BEGIN
   IF ( root is right heavy )                              // BF(root) = -2

       IF ( root.rightChild balanced or right heavy )      // BF(rightChild)
            RETURN DoLeftRotation( root )                  // = 0 or -1
       ELSE
            IF ( root.rightChild is left heavy )           // BF(rightChild) = 1
                 RETURN DoRightLeftRotation( root )
            ENDIF
       ENDIF
   ELSE
       IF ( root is left heavy )                            // BF(root) = 2

         IF ( root.leftChild balanced or left heavy )       // BF(leftChild)
              RETURN DoRightRotation( root )                // = 0 or 1
         ELSE
             IF ( root.leftChild is right heavy )           // BF(rightChild) = -1
                  RETURN DoLeftRightRotation( root )
             ENDIF
         ENDIF
       ENDIF

   ENDIF
END
```

# Pseudo code: Left & Right Rotations

Sample pseudo code for performing the *Left* & *Right* rotations to an **unbalanced** root node. (See rotation diagrams for **x, y, z, b, c**.)

```
DoLeftRotation( NODE oldRoot )
BEGIN
      NODE newRoot = oldRoot.rightChild          // set new root to y

      oldRoot.rightChild = newRoot.leftChild     // move new root's leftChild b
                                                 // to old root's rightChild

      newRoot.leftChild = oldRoot                // move old root x to be new
                                                 // root's leftChild
      RETURN newRoot
END


DoRightRotation( NODE oldRoot )
BEGIN
      NODE newRoot = oldRoot.leftChild           // set new root to y

      oldRoot.leftChild = newRoot.rightChild     // move new root's rightChild c
                                                 // to old root's leftChild

      newRoot.rightChild = oldRoot               // move old root z to be
                                                 // new root leftChild
      RETURN newRoot
END
```

# AVL Tree Operation Complexity

- Since AVL trees **remain balanced** this means that the complexity of AVL tree operations can be guaranteed.
- So all of the AVL tree operations: *insertion*, *deletion*, *searching* take $O(\log_2(N))$ time.
- It is also the case that any **unbalanced** AVL tree can be **rebalanced** within $O(\log_2(N))$ time.
- However, they are not perfectly balanced, but since pairs of sub-trees differ in height by at most 1, this is sufficient to maintain $O(\log_2(N))$ for the operations.

# PART IV

## *Overview of other types of Balanced Search Trees — B-trees*

# Searching Very Large Amounts of Data: Files

For a very large data file containing records which would be stored on disk, it is usually stored along with an **index into the file of the records**.

The **index** comprises a collection of nodes of the format:
**(key, pointer, nextnode)**.

Where:

- **key** represents the *unique identifier of a record* in the file (or tuple in a relational database),
- **pointer** holds the location of the record on disk,
- **nextnode** holds routing information where to find the next index node.

Clearly, the best way to search the index would to be to use some form of search tree, but the file's index and its tree structure maybe far too big or inefficient to deal with in one go, e.g. $O(\log_2(N))$ reads & writes to & from disc.

So an alternative approach has to be used to deal with these situations.

The usual approach is to use **B-trees** which are a type of search tree.

# B-Trees

**B-trees**[10] are usually attributed to *R. Bayer* & *E. McKnight* (1972).

By 1979, B-trees had replaced virtually all large file access methods other than hashing.

**B-trees**, (or their variants: $B^+$-trees, $B^*$-tree), are the standard file organisation for applications requiring *insertion, deletion*, and *key range searches*.

**B-trees** address all of the major problems encountered when implementing disk-based search trees:

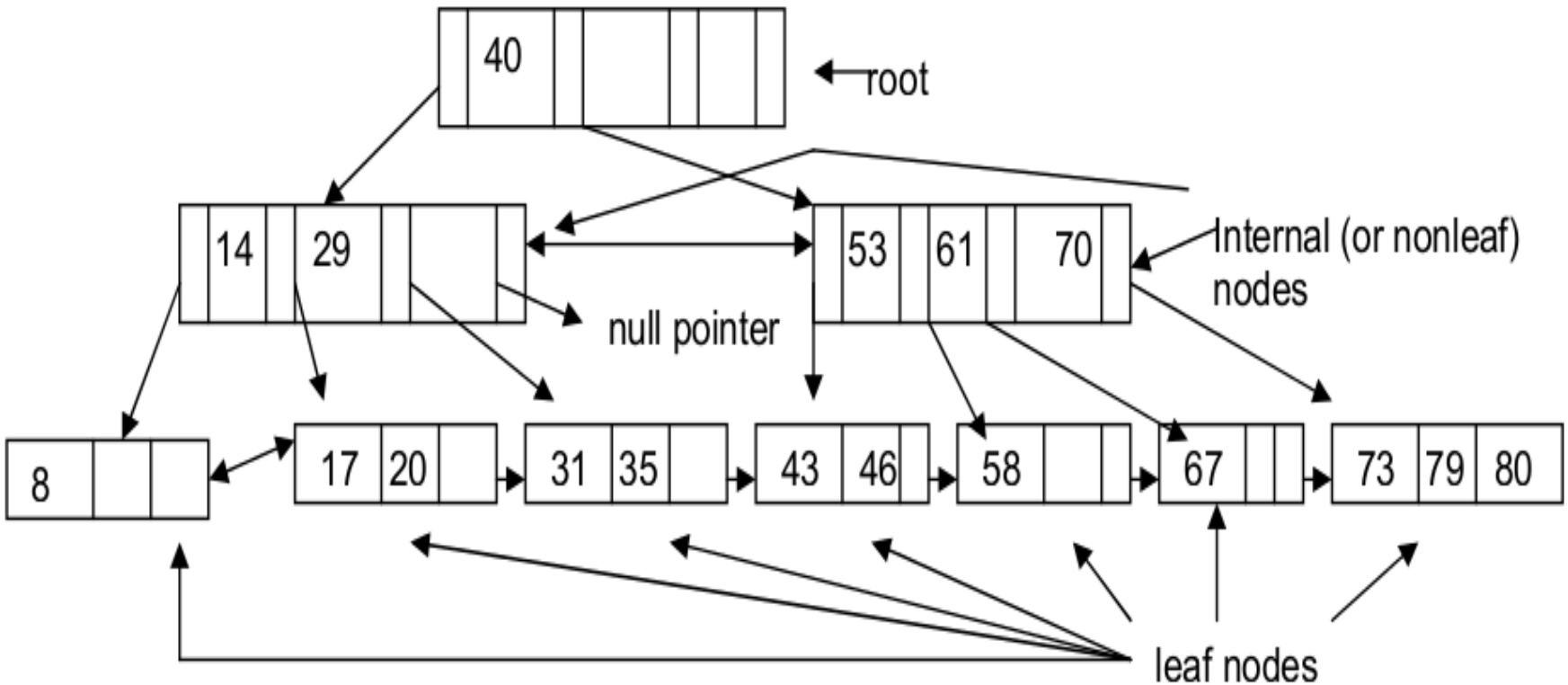
1. **always completely balanced**, all **leaf nodes** at the same level.
2. **updates** and **searches** affect only a few disk blocks, so good performance.
3. **related key values kept in the same disk block**, locality of reference.
4. **guarantee that every node in the tree will be full** at least to a certain minimum percentage.
   *Space efficient* & reduces typical number of disk fetches during a search or update operation.

[10] **These B-tree notes are based on Ping Brennan's B-Tree notes.**

# Example B-Tree

This example of a B-tree has **order four**.

This means that each: **node** contains *up to three keys*, and **internal/non-leaf** nodes can have *up to four children*.



# Searching a B-Tree

**Searching** in a B-tree for a **key** is an alternating two-step process, starting at the root node:

1. Do a binary search on the **keys** in the current node.
   If the key is **found** then return its *associated record*.
   If the current node is a **leaf node** and the key is **not found** then search failed.
2. Otherwise, follow the proper branch and repeat the process.

For example, a search for the record with **key 67** in the example B-tree:

1. The root node (level 0) is examined and the *second (right) branch* is taken, because $40 \leq 67$.
2. After examining the node at level 1, the *third branch* is taken to the next level 2, because $61 \leq 67 < 70$.
3. After examining the leaf node at level 2, **key 67** is found in the first position.