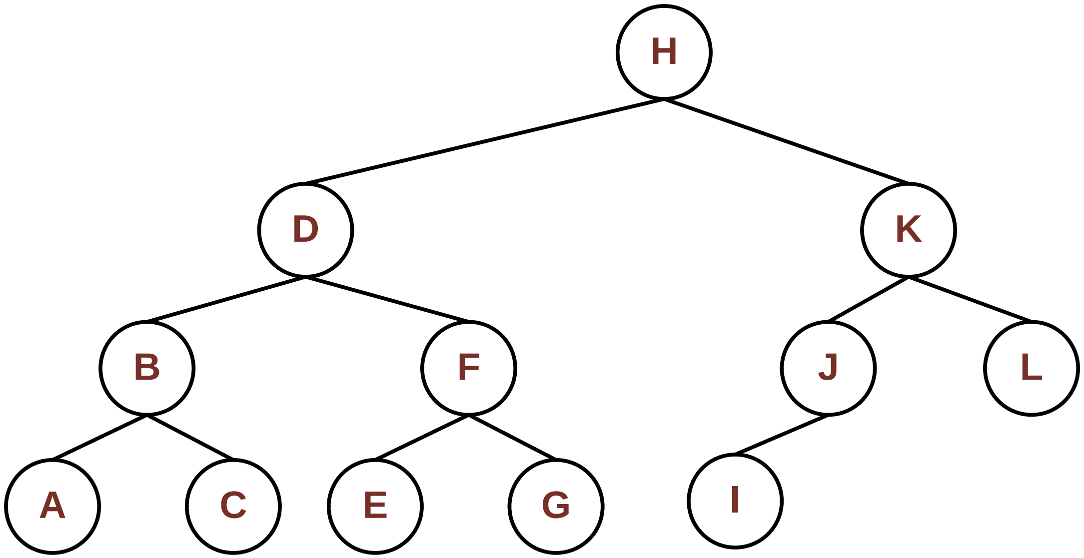
Trees and Tree-Based Algorithms

Dr. Robert Amelard
(adapted from Dr. Igor Ivkovic)

ramelard@uwaterloo.ca

Binary Tree Traversals

- **Traversing binary trees**
  - **Pre-order traversal:** Visit root, left subtree, then right subtree
  - **In-order traversal:** Visit left subtree, root, then right subtree
  - **Post-order traversal:** Visit left subtree, right subtree, then root



Pre-order:

H D B A C F E G K J I L

In-order:

A B C D E F G H I J K L

Post-order:

A C B E G F D I J L K H

PreOrder Traversal /1

- **Algorithm: PreOrder(T) (recursive)**
  - Input: BinaryTreeNode T
  - Output: preorder traversal of the tree node data
  - Steps:

```
void PreOrder(T) {
    if (T == null) return;
    Visit(T.data); // print root data or other processing
    PreOrder(T.leftChild);
    PreOrder(T.rightChild);
}
```

PreOrder Traversal /2

- **Algorithm Trace:**

  PreOrder(T_H);

  Visit(H);

  PreOrder(T_D);

  Visit(D);

  PreOrder(T_B);

  Visit(B);

  PreOrder(T_F);

  Visit(F);

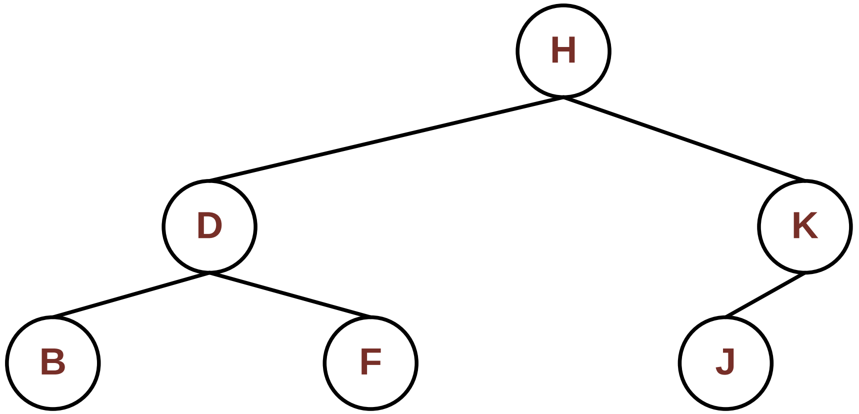
  PreOrder(T_K);

  Visit(K);

  PreOrder(T_J);

  Visit(J);

- **Visitation Order:**
  - H D B F K J



InOrder Traversal /1

- **Algorithm: InOrder(T) (recursive)**
  - Input: BinaryTreeNode T
  - Output: inorder traversal of the tree node data
  - Steps:

```
void InOrder(T) {
    if (T == null) return;
    InOrder(T.leftChild);
    Visit(T.data); // print root data or other processing
    InOrder(T.rightChild);
}
```

InOrder Traversal /2

- **Algorithm Trace:**

InOrder(T_H);

InOrder(T_D);

InOrder(T_B);

Visit(B);

Visit(D);

InOrder(T_F);

Visit(F);

Visit(H);

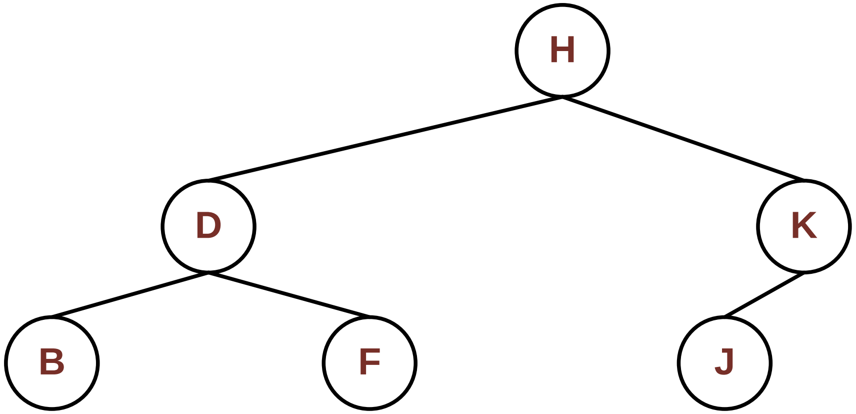
InOrder(T_K);

InOrder(T_J);

Visit(J);

Visit(K);

- **Visitation Order:**
  - B D F H J K



PostOrder Traversal /1

- **Algorithm: PostOrder(T) (recursive)**
  - Input: BinaryTree T
  - Output: postorder traversal of the tree node data
  - Steps:

```
void PostOrder(T) {
    if (T == null) return;
    PostOrder(T.leftChild);
    PostOrder(T.rightChild);
    Visit(T.data); // print root data or other processing
}
```

PostOrder Traversal /2

- **Algorithm Trace:**

  PostOrder(T_H);

  PostOrder(T_D);

  PostOrder(T_B);

  Visit(B);

  PostOrder(T_F);

  Visit(F);

  Visit(D);

  PostOrder(T_K);

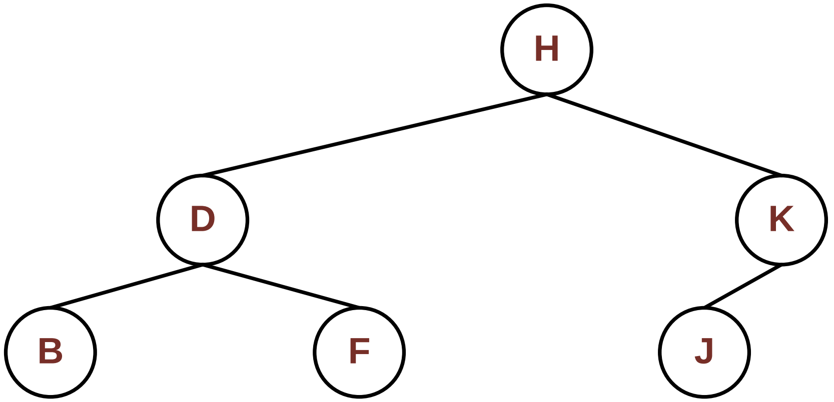
  PostOrder(T_J);

  Visit(J);

  Visit(K);

  Visit(H);

- **Visitation Order:**
  - B F D J K H



Depth-First Traversals (DFT)

- **Tree traversals covered so far are considered Depth-First Traversals (DFT)**
  - The key idea is that each branch is traversed as far as possible before backtracking.
  - Before a node is considered traversed, all of its subnodes have to be traversed too

- **Example: solving a maze**

- **Pre-order, In-order, and Post-order are specific implementations of DFT**

PreOrder Traversal Using Stack /1

- **Algorithm: PreOrderUsingStack(T)**
  - Input: BinaryTreeNode T
  - Output: preorder traversal of the tree node data
  - Steps:

```
void PreOrderUsingStack (T) {
    Stack S = new Stack();
    S.push(T);
    while (!S.isEmpty()) {
            BinaryTreeNode P = S.pop();
            Visit(P.data); // print root data or other processing
            if (P.rightChild != NULL) S.push(P.rightChild);
            if (P.leftChild != NULL) S.push(P.leftChild);
    }
}
```

PreOrder Traversal Using Stack /2

- **Algorithm Trace:**

PreOrderUsingStack(T_a)

$S = \{T_H\}$

$P = T_H$, Visit(H), $S = \{T_K, T_D\}$

$P = T_D$, Visit(D), $S = \{T_K, T_F, T_B\}$

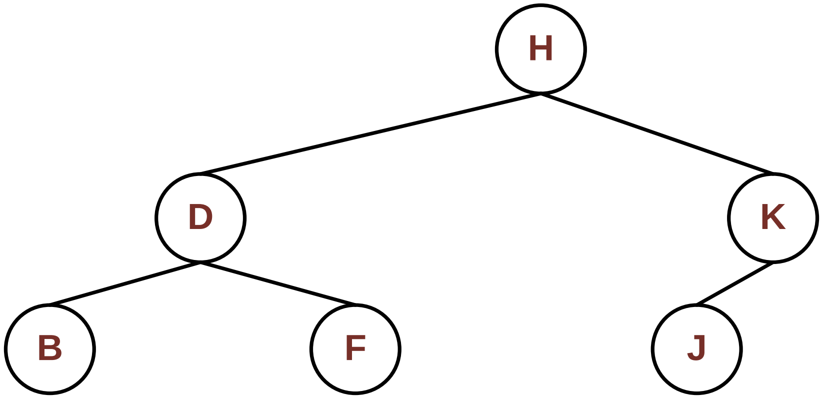
$P = T_B$, Visit(B), $S = \{T_K, T_F\}$

$P = T_F$, Visit(F), $S = \{T_K\}$

$P = T_K$, Visit(K), $S = \{T_J\}$

$P = T_J$, Visit(J), $S = \{\}$,

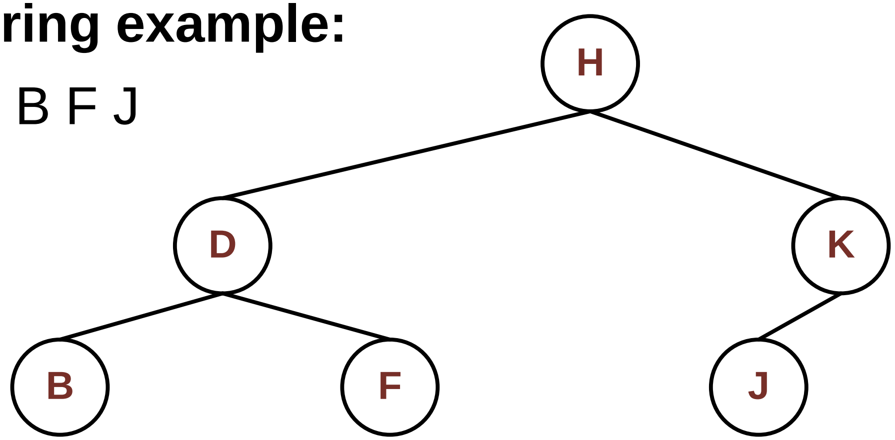
- **Visitation Order:**
  - H D B F K J



Breadth-First Traversals (BFT)

- **Breadth-First Traversals (BFT)**
  - The key idea is that neighbouring nodes are explored before moving down the tree.
  - Also known as **level-based traversal**
  - All nodes at level 0 are visited first
    - Then the nodes at level 1 are visited
    - Then the nodes at level 2 are visited
    - …
    - Finally, the nodes at the maximum level (tree height) are visited

- **BFT ordering example:**
  - H D K B F J



BFT Traversal /1

- **Algorithm: BFT(T)**
  - Input: BinaryTreeNode T
  - Output: breadth-first traversal of the tree node data
  - Steps:

```
void BFT(T) {
    Queue Q = new Queue();
    Q.enqueue(T);
    while (!Q.isEmpty()) {
            BinaryTreeNode P = Q.dequeue();
            Visit(P.data); // print root data or other processing
            if (P.leftChild != NULL) Q.enqueue(P.leftChild);
            if (P.rightChild != NULL) Q.enqueue(P.rightChild);
    }
}
```

BFT Traversal /2

- **Algorithm Trace:**

BFT(T_H)

$Q = \{T_H\}$

$P = T_H$, Visit(H), $Q = \{T_D, T_K\}$

$P = T_D$, Visit(D), $Q = \{T_K, T_B, T_F\}$

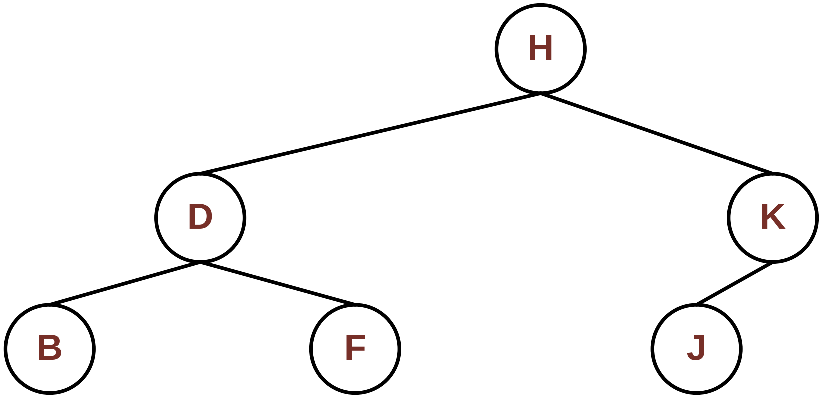
$P = T_K$, Visit(K), $Q = \{T_B, T_F, T_J\}$

$P = T_B$, Visit(B), $Q = \{T_F, T_J\}$

$P = T_F$, Visit(F), $Q = \{T_J\}$

$P = T_J$, Visit(J), $Q = \{\}$

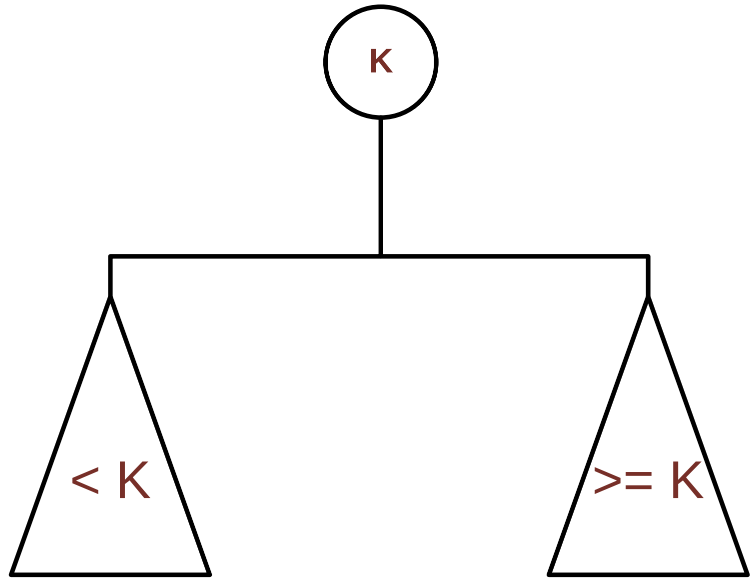
- **Visitation Order:**
  - H D K B F J



Binary Search Tree (BST) /1

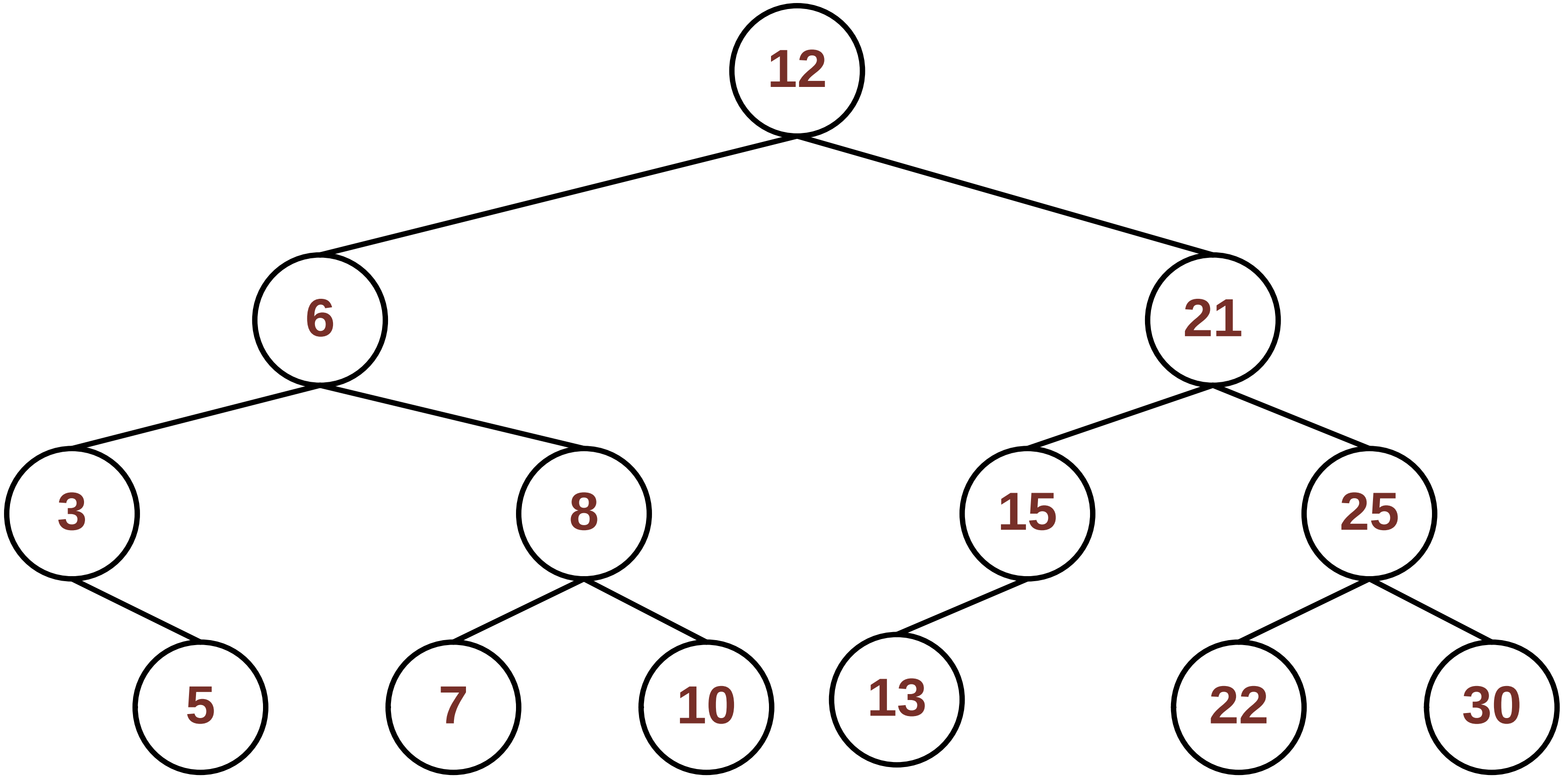
- **Binary Search Tree (BST) Property:**
  - Each node has a key value K
  - All keys in the left subtree are less than K
  - All keys in the right subtree are greater than or equal to K

- **Binary Search Tree (BST):**
  - A binary tree where BST property holds for each node



Binary Search Tree (BST) /2

- **Binary Search Tree (BST) Example:**
  - BST property holds for each node

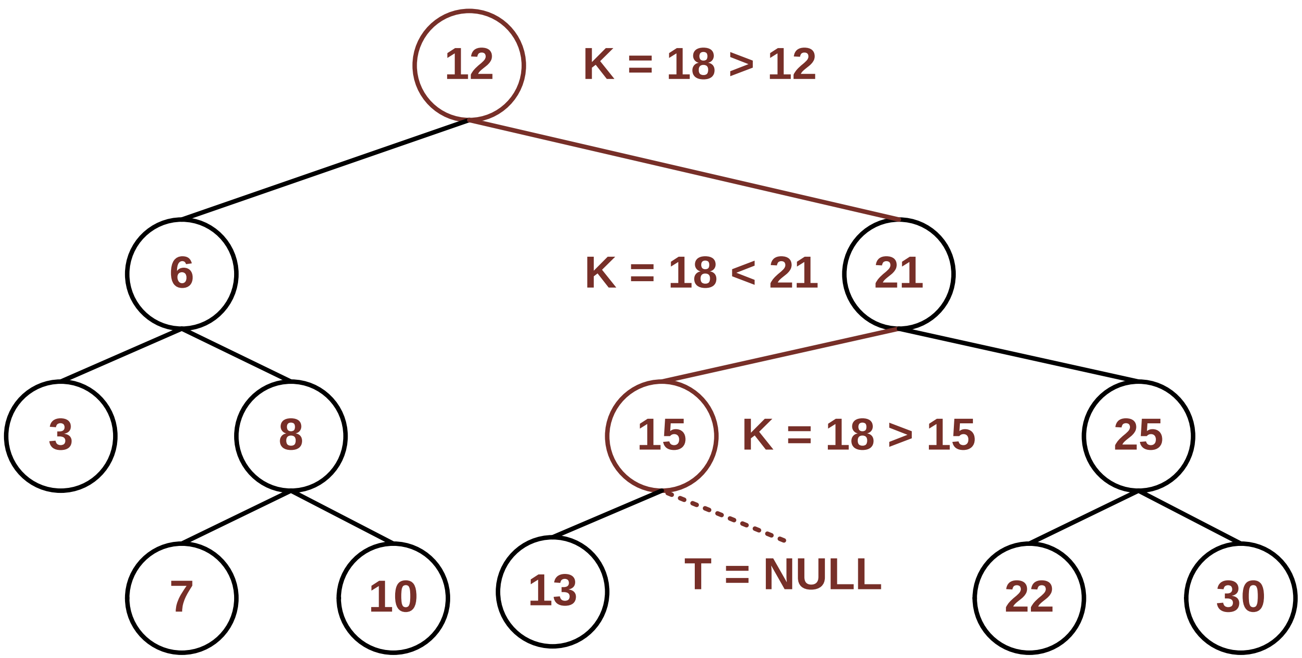


Searching using BST /1

- **BSTNode Search(BSTNode T, int K):**
  - If T is NULL, the search has failed, so return NULL and terminate
  - Compare the desired key value K with the key value of the current node K_T
  - If K == K_T, then return T and terminate
  - If K < K_T, then continue search in the left subtree of T
  - If K > K_T, then continue search in the right subtree of T

Searching using BST /2

- **Example: Search(T_{12}, 18)**
  - Step 1. K = 18 > 12, so traverse right
  - Step 2. K = 18 < 21, so traverse left
  - Step 3. K = 18 > 15, so traverse right
  - Step 4. T = NULL, so return NULL and terminate



Searching using BST /3

- **Algorithm: Search(T, K)**
  - Input: BST node T, key value K
  - Output: if found, return a node with the key value K; otherwise, return NULL
  - Steps:

```
BSTNode Search(T, K) {
    if (T == NULL) return NULL;
    if (T.key == K) return T;
    else if (K < T.key) Search(T.leftChild, K);
    else if (K > T.key) Search(T.rightChild, K);
}
```

Searching using BST /4

- **Algorithm Trace:**

  Search(T_{12},18);

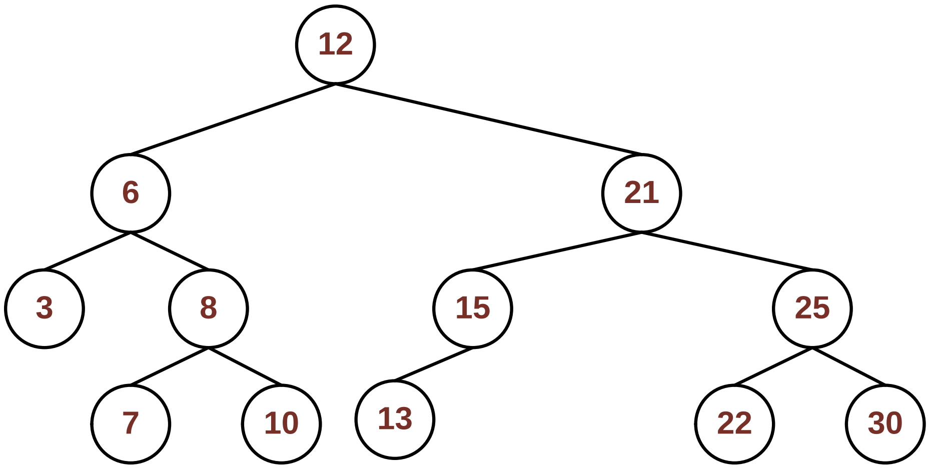
    Search(T_{21},18);

      Search(T_{15},18);

        Search(NULL,18);

          return NULL;

- **Visitation Order:**
  - 12, 21, 15, NULL

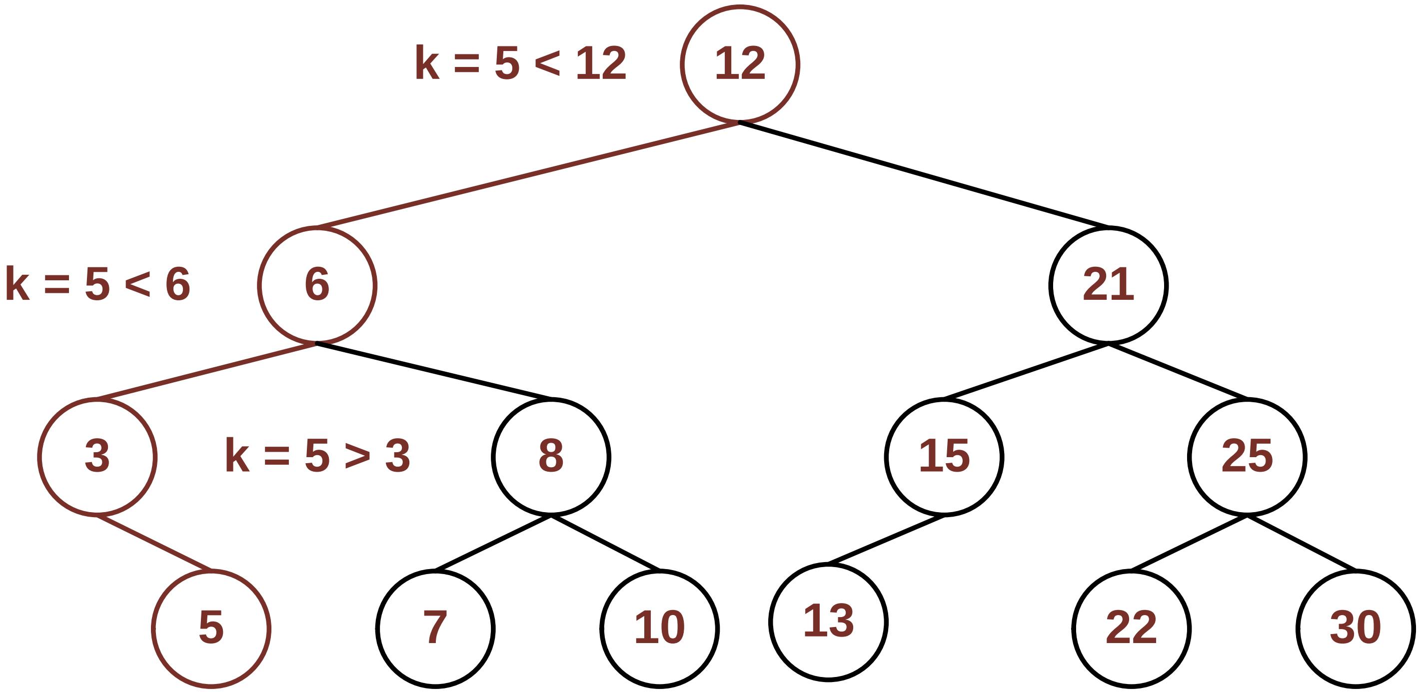


Inserting Nodes into BST /1

- **bool Insert(BSTNode T, BSTNode N):**
  - If T is NULL, insert the node as the root of T, then return true and terminate
  - Compare the desired key value K_N with the key value of the current node K_T
  - If $K_N == K_T$, then return false and terminate
  - If $K_N < K_T$, then continue insertion in the left subtree of T
  - If $K_N > K_T$, then continue insertion in the right subtree of T

Inserting Nodes into BST /2

- **Example: Insert(T_{12}, BSTNode(5))**
  - Step 1. $K_N = 5 < 12$
  - Step 2. $K_N = 5 < 6$
  - Step 3. $K_N = 5 > 3$
  - Step 4. T.rightChild = N, so return true and terminate



Inserting Nodes into BST /3

- **Algorithm Trace:**

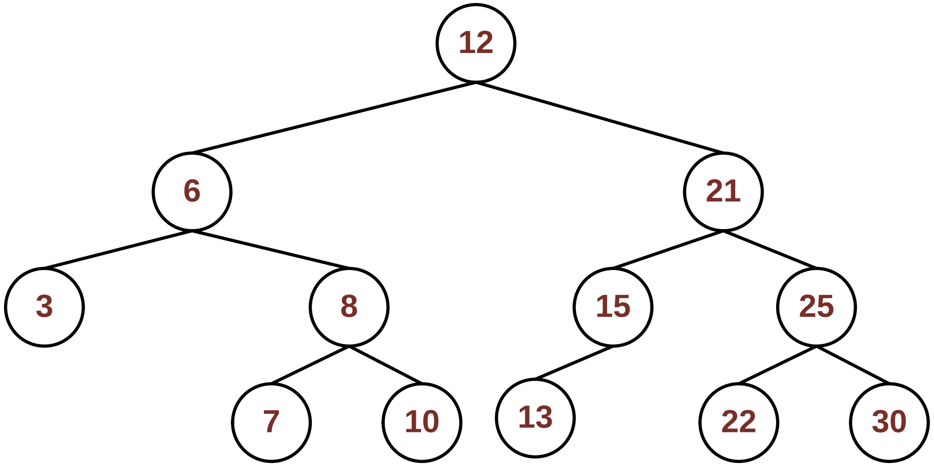
  Insert(T_{12},N_5);

  Insert(T_6, N_5);

  Insert(T_3, N_5);

  T_3.rightChild = N_5;

- **Visitation Order:**
  - 12, 6, 3, NULL (Insert)



Deleting Nodes from BST /1

- **bool Delete(BSTNode T, BSTNode D):**
  - Find D in T
  - If D cannot be found, return false
  - If D is found, then do the following:

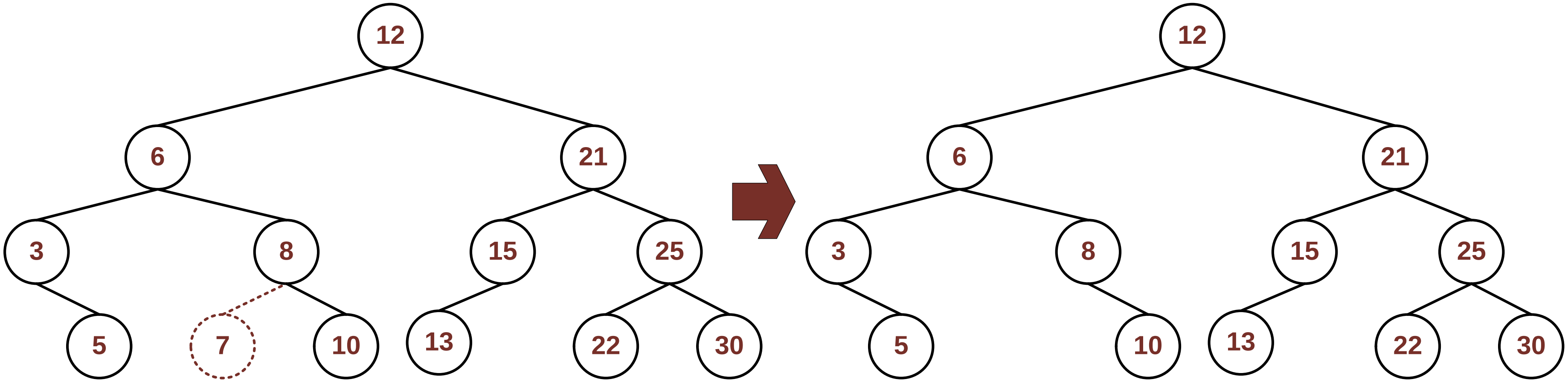
1. (Case1) If D is a leaf node in T, remove it, then return true and terminate

2. (Case2) If D has one child node, swap with the child node, delete the child node, then return true and terminate

3. (Case3) If D has two child nodes, swap the values with the successor or predecessor, delete the successor or predecessor respectively, and then return true and terminate
   - *Predecessor is the maximum value in the left subtree of BST*
   - *Successor is the smallest value in the right subtree of BST*

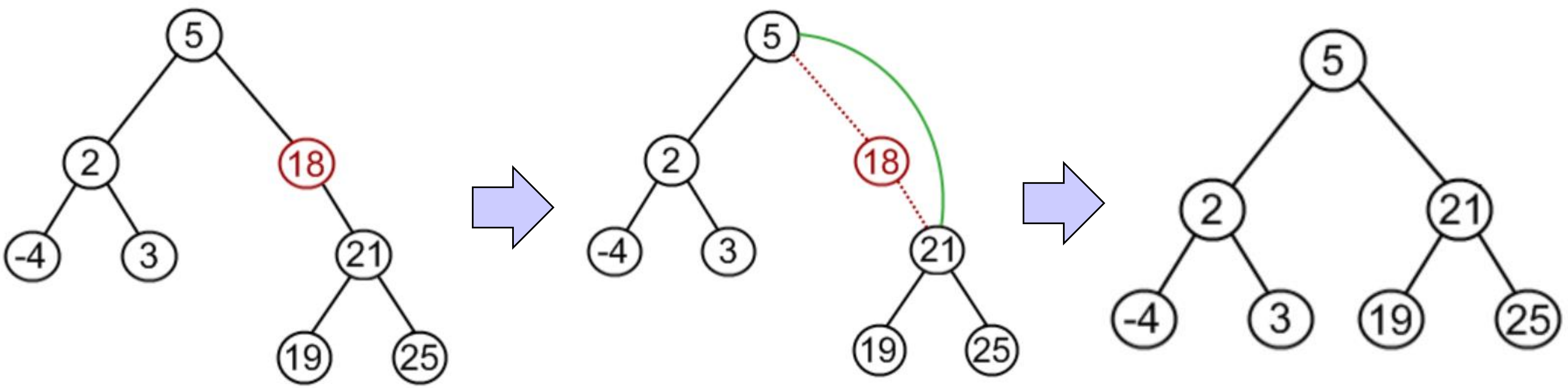
Deleting Nodes from BST /2

- **Deleting Nodes from BST:**
  - (Case1) If D is a leaf node in T, remove it and terminate
  - Example: Delete(T_{12}, D_7)



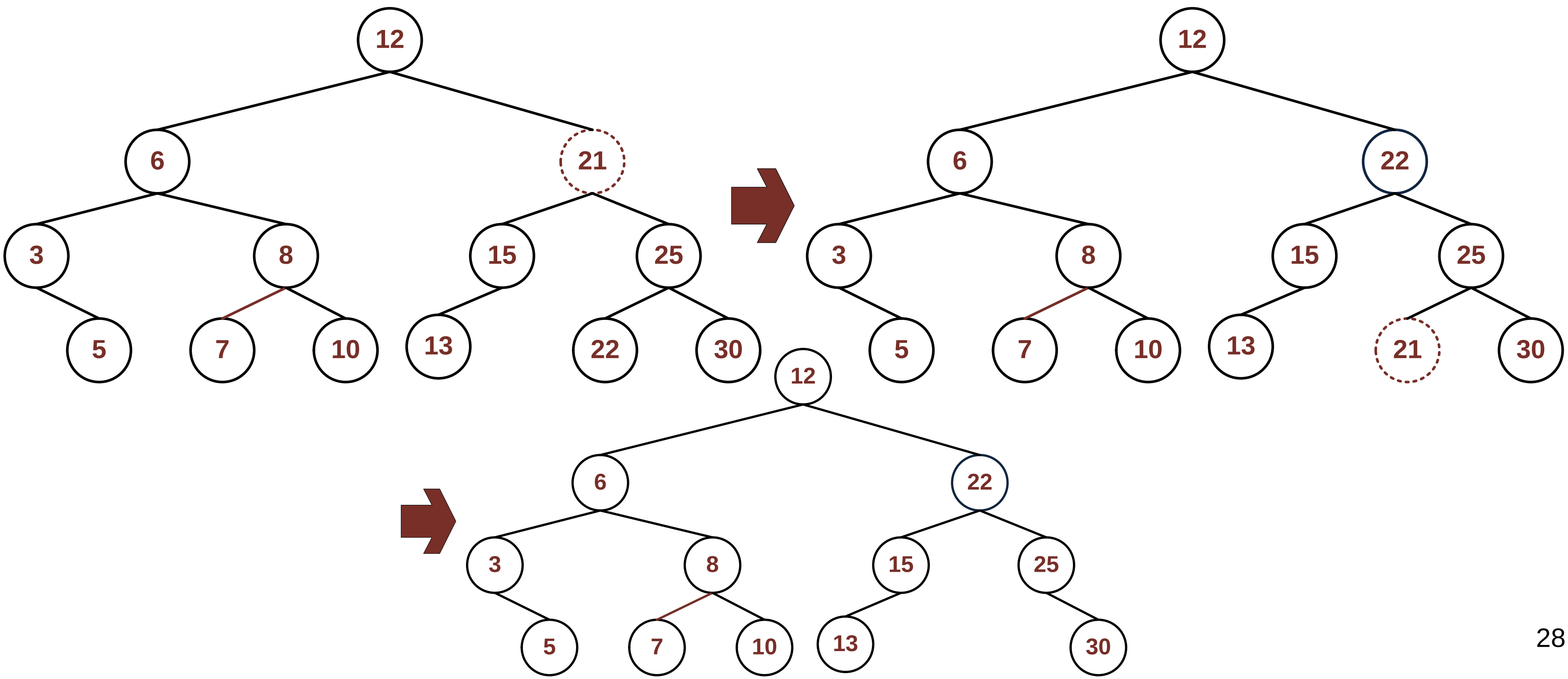
Deleting Nodes from BST /3

- **Deleting Nodes from BST:**
  - (Case2) If D has one child, link parent node to child node, delete the current node, then return true and terminate
  - Example: Delete(T_{12}, D_3)



Deleting Nodes from BST /4

- **Deleting Nodes from BST:**
  - (Case3) If D has two child nodes, swap the values with the successor or predecessor, delete the chosen node, and then return true and terminate
  - Example: Delete(T_{12}, D_{21}) // swap with the successor



BST Efficiency

- **Binary Search Tree Efficiency:**
  - Search, Insert, and Delete each take O(h) primitive operations to run, where h is the height of the tree
  - In the worst case scenario, a tree of size n can grow in a straight line, with the height h = n – 1
  - Therefore, if the BST is not kept balanced, the runtime efficiency of its key operations is O(n)

- **How to keep binary search trees balanced?**
  - Answer: AVL Trees

Introducing AVL Trees /1

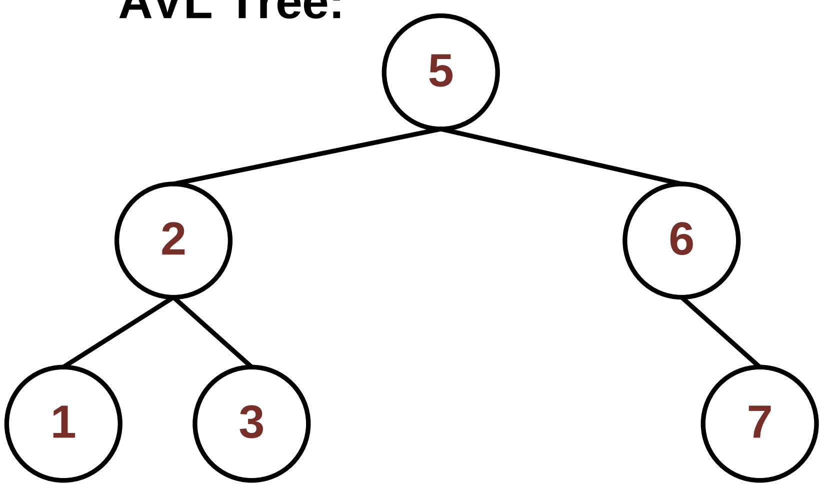
- **AVL Tree:**
  - A specialized BST that is kept balanced
    - Named after the inventors, Adelson-Velsky and Landis

- **AVL Tree Property:**
  - The absolute difference in heights between the left and right subtrees is less than or equal to 1
  - $\Delta_{AVL} = |\ \mathrm{Height}(T_L) - \mathrm{Height}(T_R)\ | \leq 1$
  - In an AVL tree, AVL Tree property holds for every node

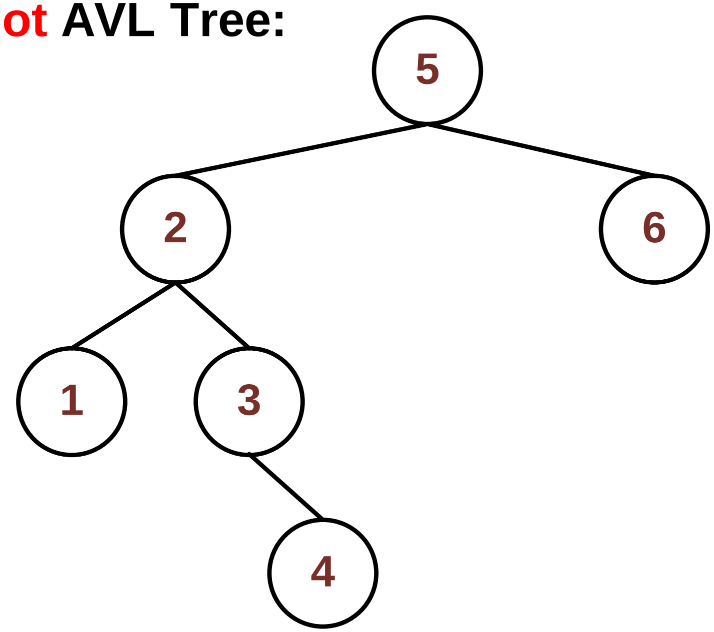
Introducing AVL Trees /2

- **AVL Tree Examples:**
  - The tree on the left is an AVL tree since
    | Height(T_L) – Height(T_R) | ≤ 1 for every node
  - The tree on the right is not an AVL tree since
    | Height(T_L) – Height(T_R) | = | 3 – 1 | = 2 at the root

AVL Tree:



Not AVL Tree:



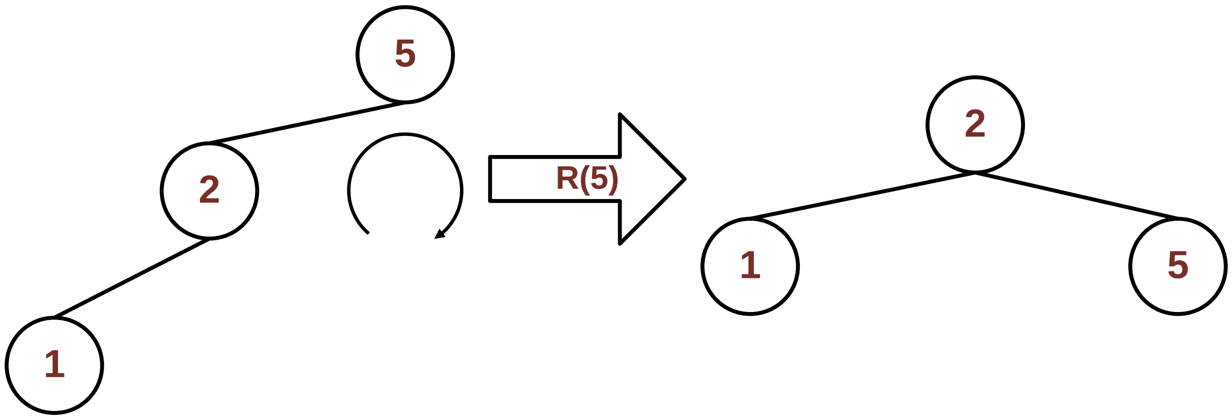
Introducing AVL Trees /3

- **AVL Insert:**
  - Use the regular BST Insert operation
  - Then balance the AVL tree using one of the four rotations, based on where the new node was inserted:
    - Insert into left subtree of left child of α: **Single Right**
    - Insert into right subtree of right child of α: **Single Left**
    - Insert into right subtree of left child of α: **Double Left-Right**
    - Insert into left subtree of right child of α: **Double Right-Left**
  - **α is the first node (traversing up) that violates the AVL property.**

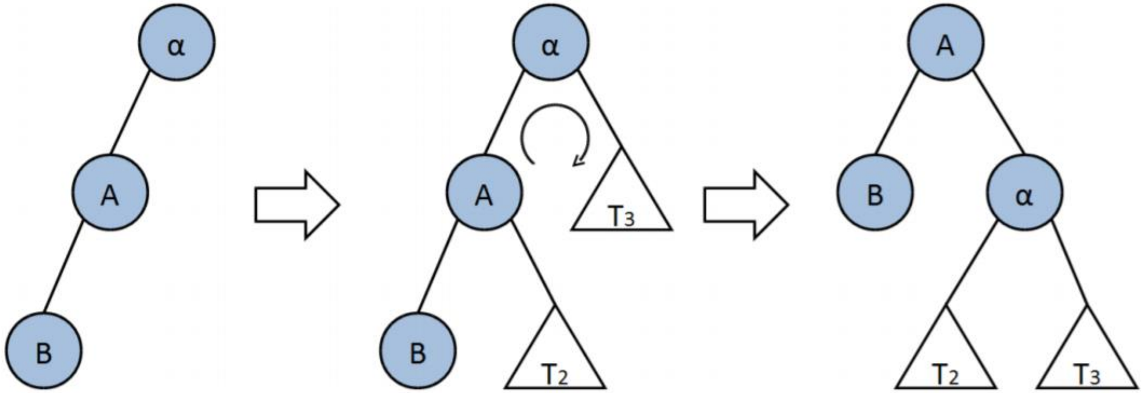
Introducing AVL Trees /4

Insert into **left subtree** of **left child**

- **Single Right Rotation – Specific Example:**



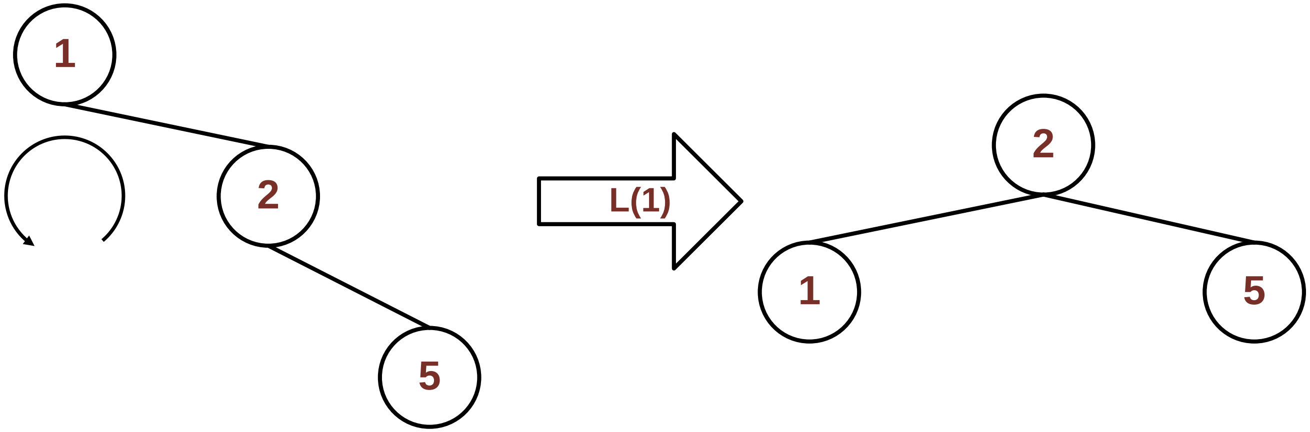
- **Single Right Rotation – Generic Example:**



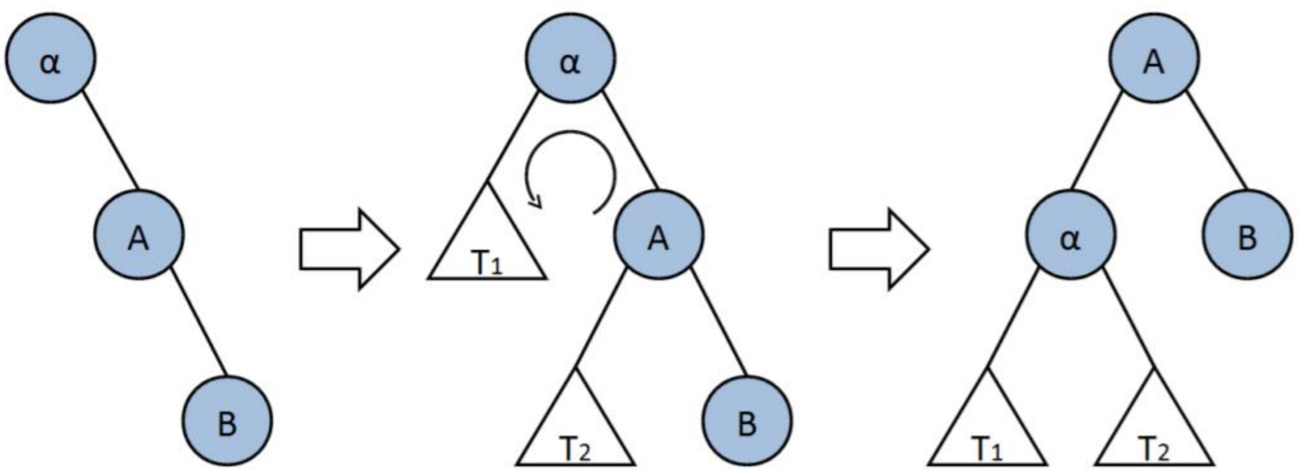
Insert into **right subtree** of **right child**

Introducing AVL Trees /5

- **Single Left Rotation – Specific Example:**



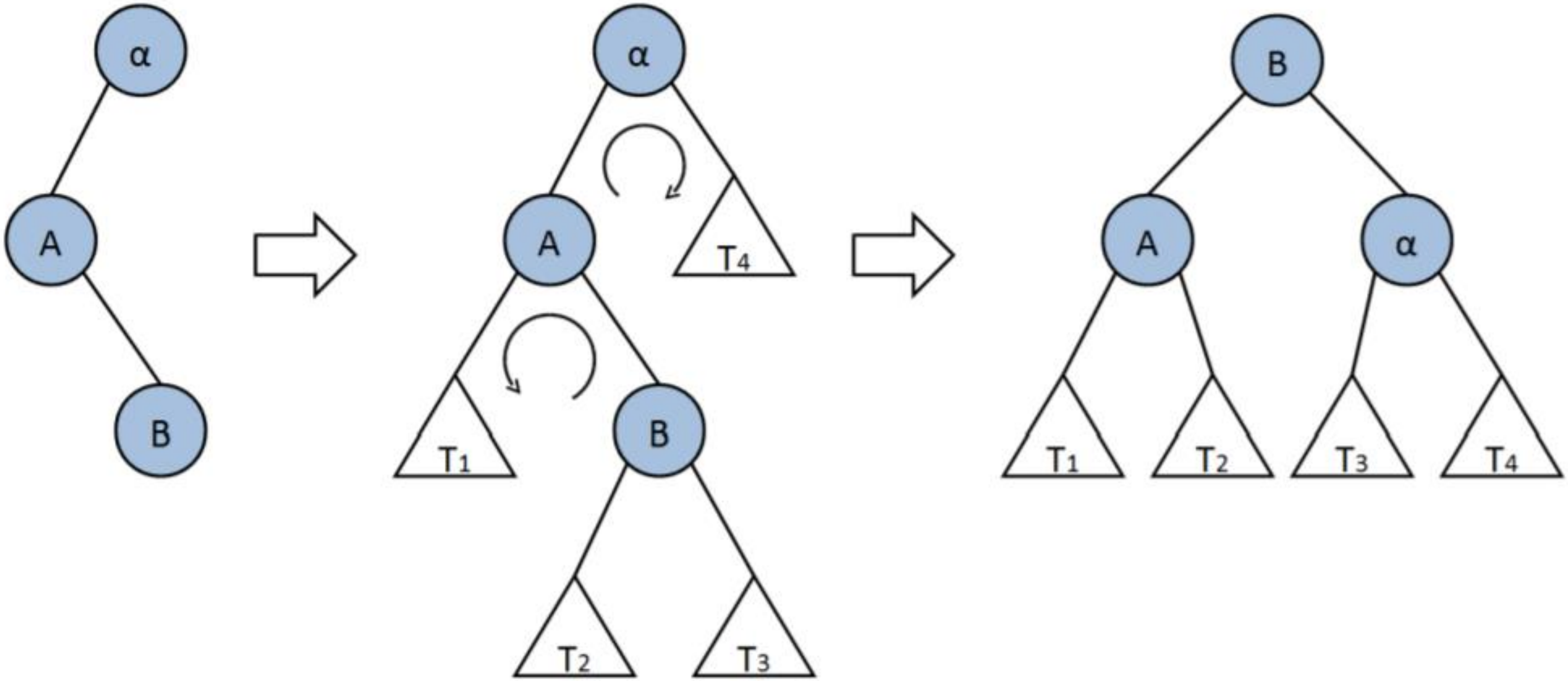
- **Single Left Rotation – Generic Example:**



Introducing AVL Trees /6

Insert into **right subtree** of **left child**

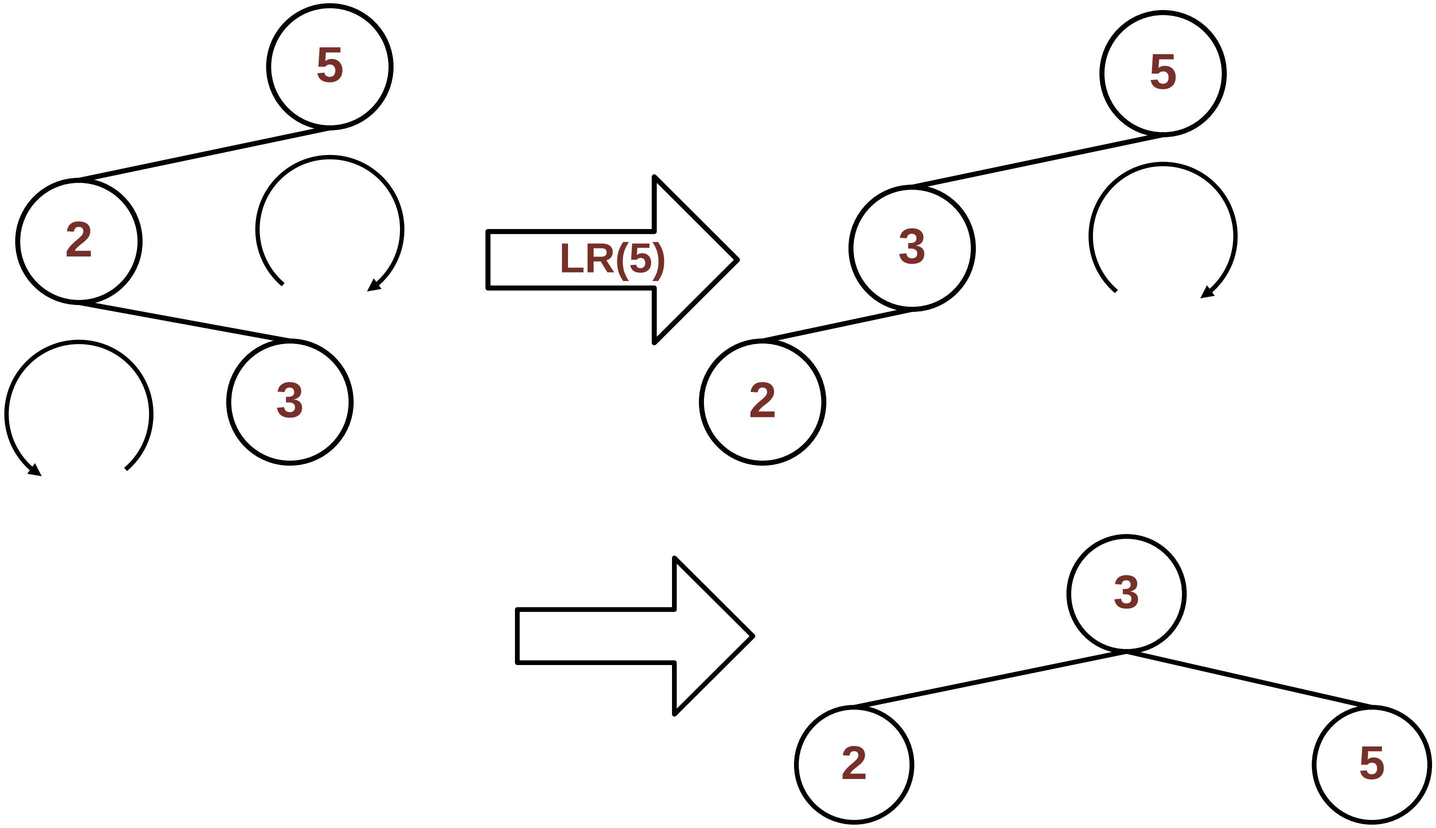
- **Double Left-Right Rotation – Generic Example:**



Insert into **right subtree** of **left child**

Introducing AVL Trees /7

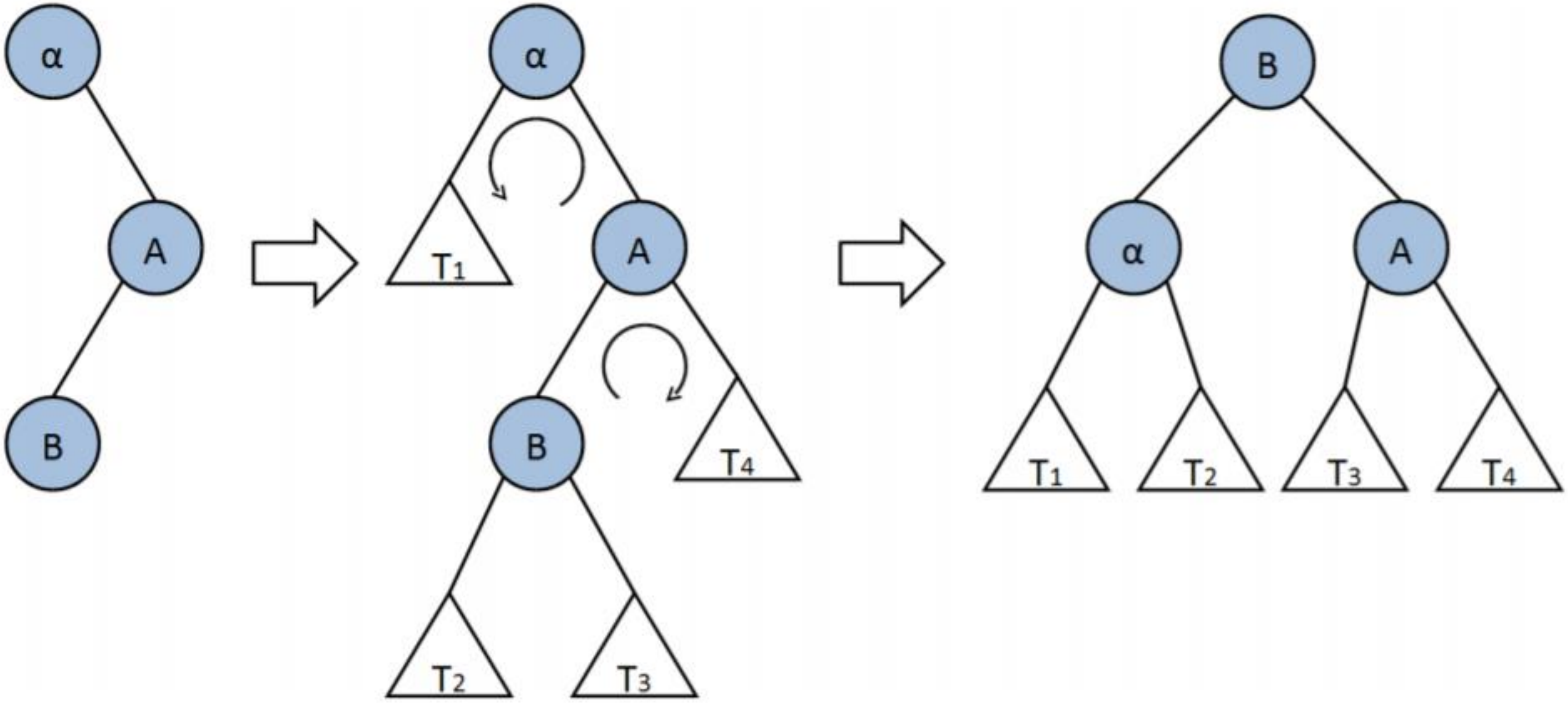
- **Double Left-Right Rotation – Specific Example:**



Insert into **left subtree** of **right child**

Introducing AVL Trees /8

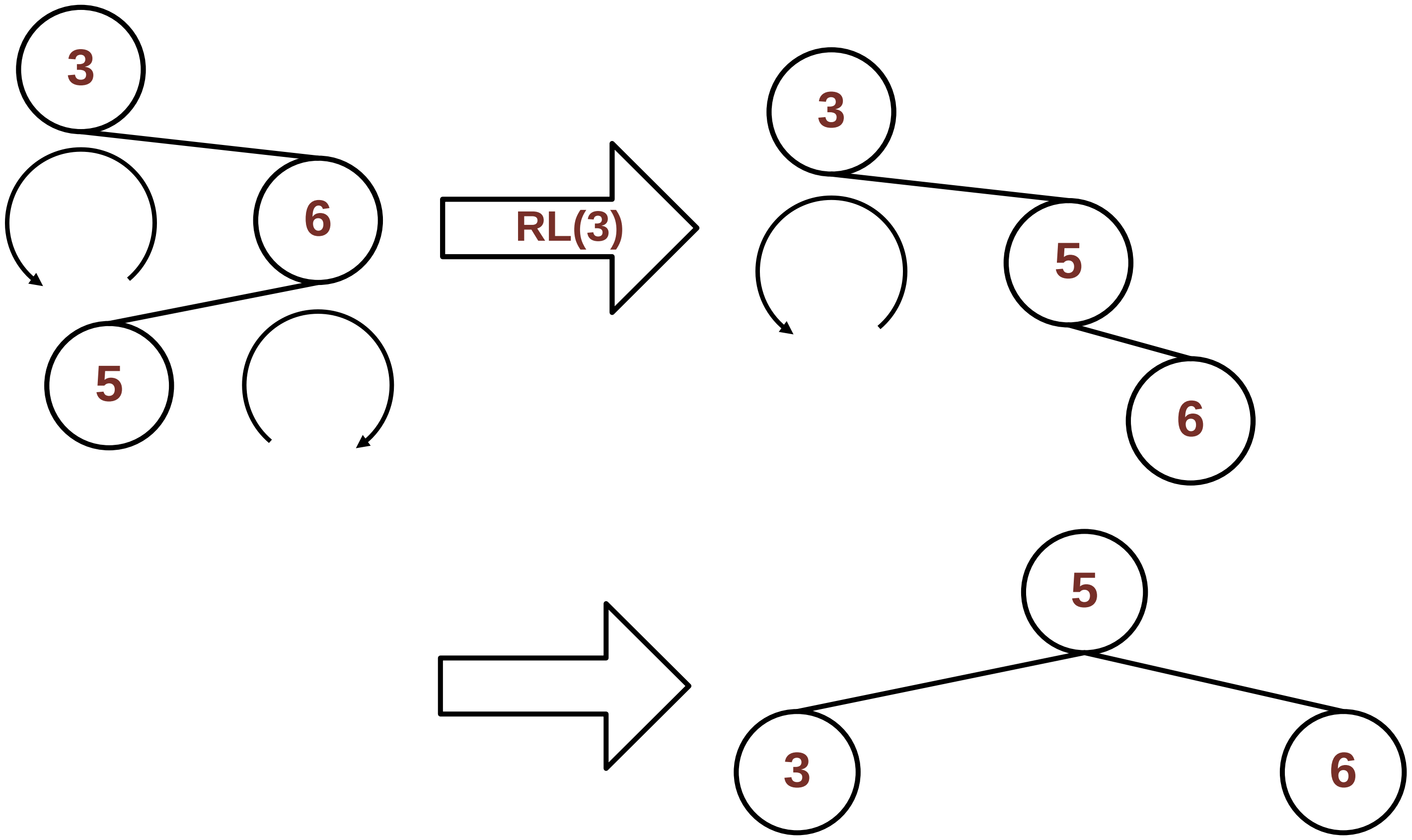
- **Double Right-Left Rotation – Generic Example:**



Insert into **left subtree** of **right child**

Introducing AVL Trees /9

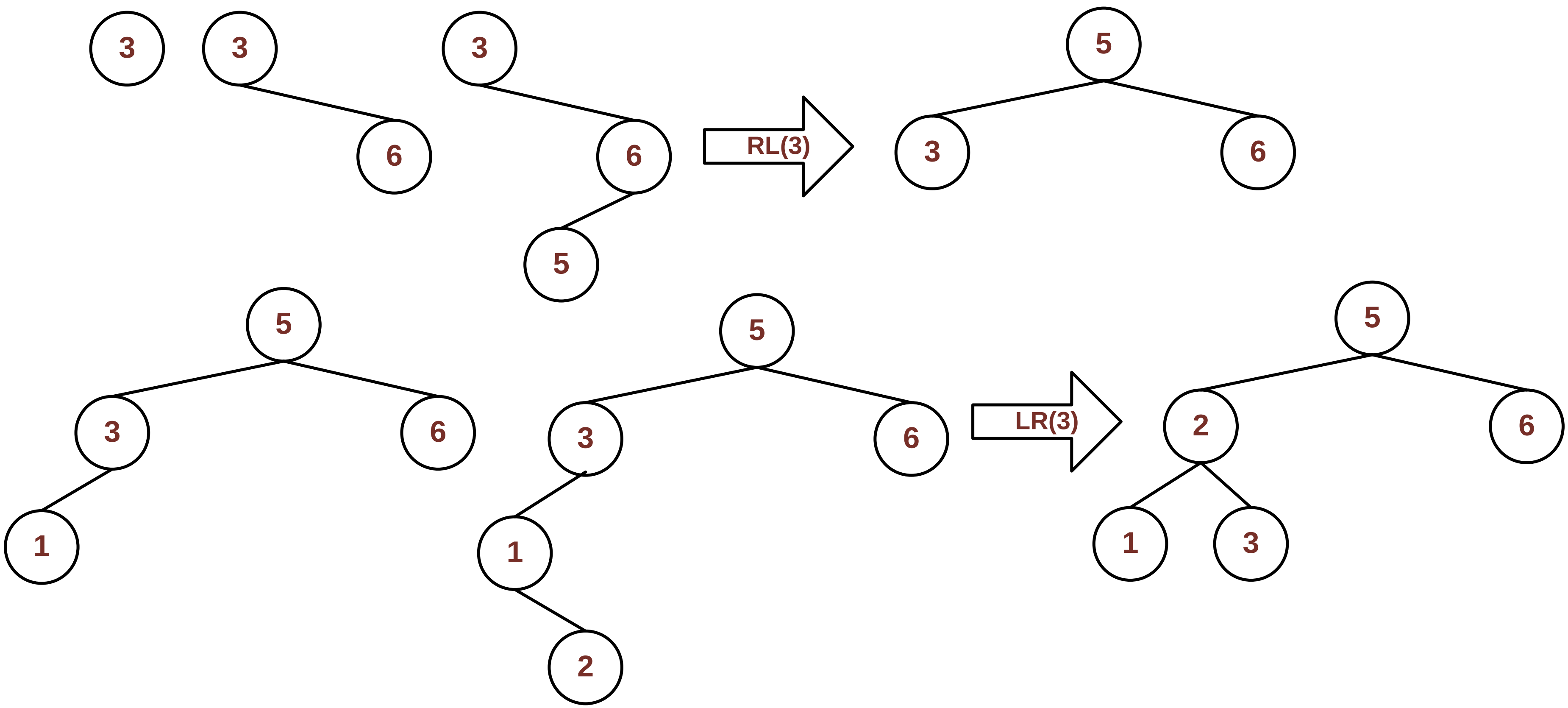
- **Double Right-Left Rotation – Specific Example:**



Insert: 3,6,5,1,2,10,15,13,12

Introducing AVL Trees /10

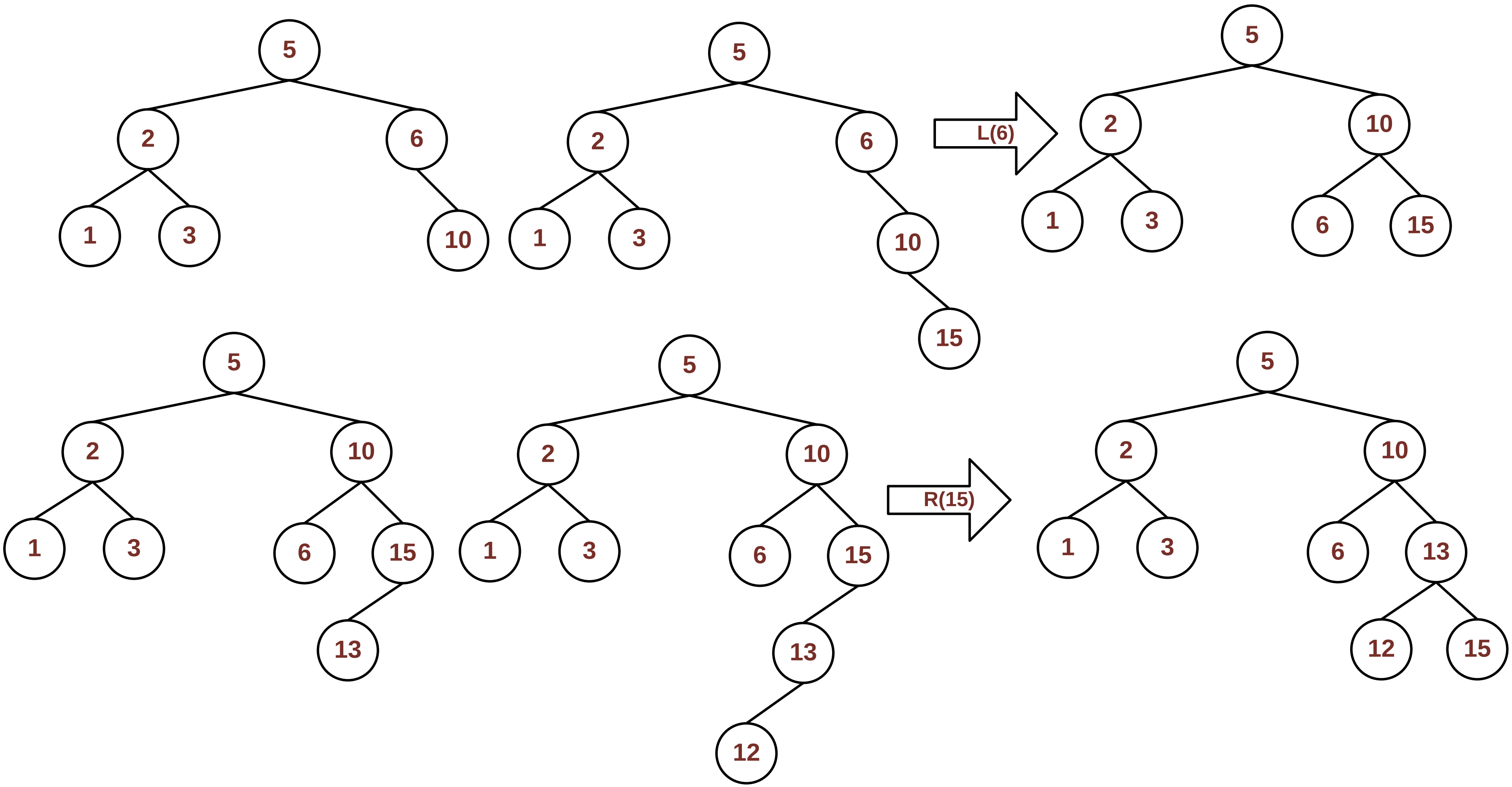
- **AVL Tree Construction Example:**



Insert: 3,6,5,1,2,10,15,13,12

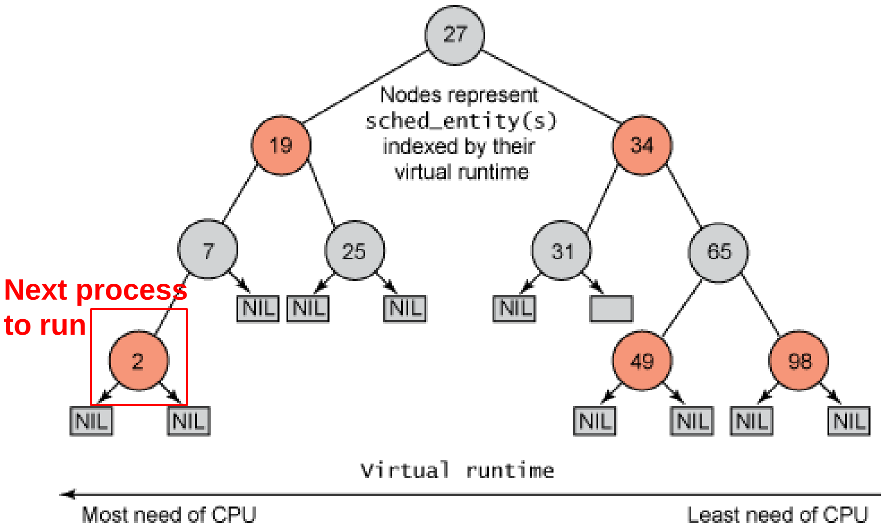
Introducing AVL Trees /11

- **AVL Tree Construction Example Continued:**



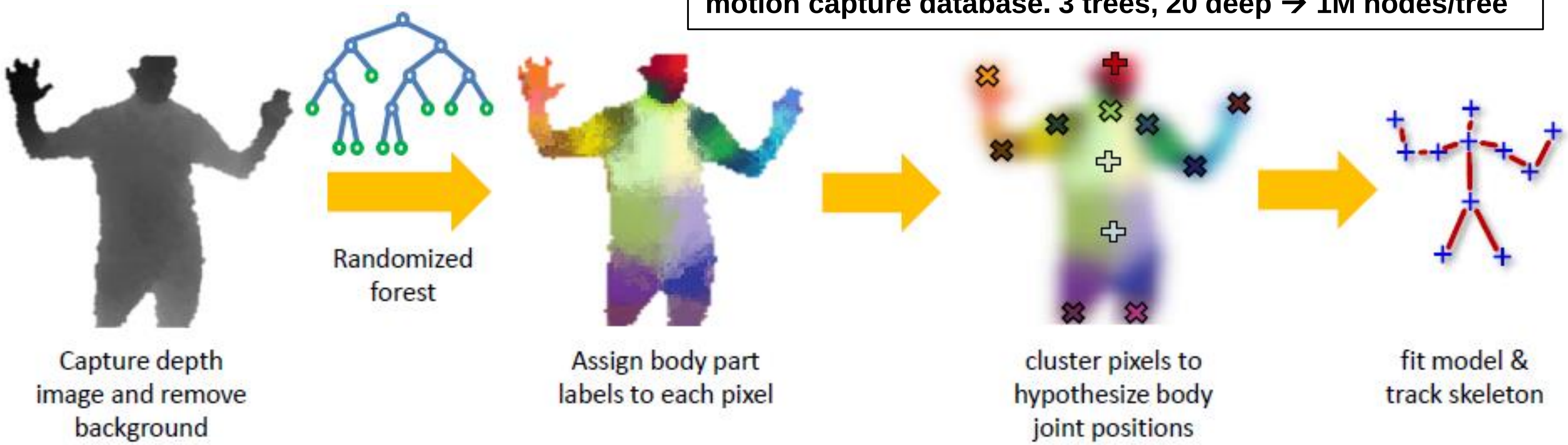
Aside: Applications of Trees

- Completely Fair Scheduler
  - Linux kernel process scheduler
  - Red-black tree (better for insert-intensive tasks)



- Machine learning: decision tree/random forest
  - Microsoft Kinect

500,000 frames from 300 sequences of exercises from motion capture database. 3 trees, 20 deep → 1M nodes/tree



Aside: Other Trees

- There are many other useful trees that we don’t cover in this course. Some popular ones include:
  - Red-Black tree
  - B tree / B+ tree
  - Splay tree
  - Huffman tree
- Also see the Visitor software design pattern

Electronic Course Evaluation

https://evaluate.uwaterloo.ca

- Login using your Quest credentials
- Answer all questions in one sitting
- Hit Submit

Difficulties? Contact kabecker@uwaterloo.ca

Lecture Notes Summary

- **What do you need to know?**
  - The basics of trees
  - Binary trees
  - Complete binary trees
  - Heaps
  - Pre/In/Post Tree traversals
  - DFT traversal
  - BFT traversal
  - Binary search tree (BST)
  - Searching using BST
  - Inserting into BST
  - Deleting from BST
  - BST efficiency
  - The basics of AVL trees
  - AVL tree rotations

Food for Thought

- **Read:**
  - Chapter 7 (Trees) from the course handbook

- **Additional Readings:**
  - Chapter 10 (Trees) from “Data Structures and Other Objects Using C++” by Main and Savitch
  - Chapter 11 (Balanced Trees) from “Data Structures and Other Objects Using C++” by Main and Savitch