Algorithms and Data Structures

Binary Search Trees

dr Szymon Murawski

Comarch SA

May 2, 2019

Table of contents I

1. Binary Search Trees
   - Implementation
2. AVL Trees
3. Red-Black Trees
4. Other types of search trees

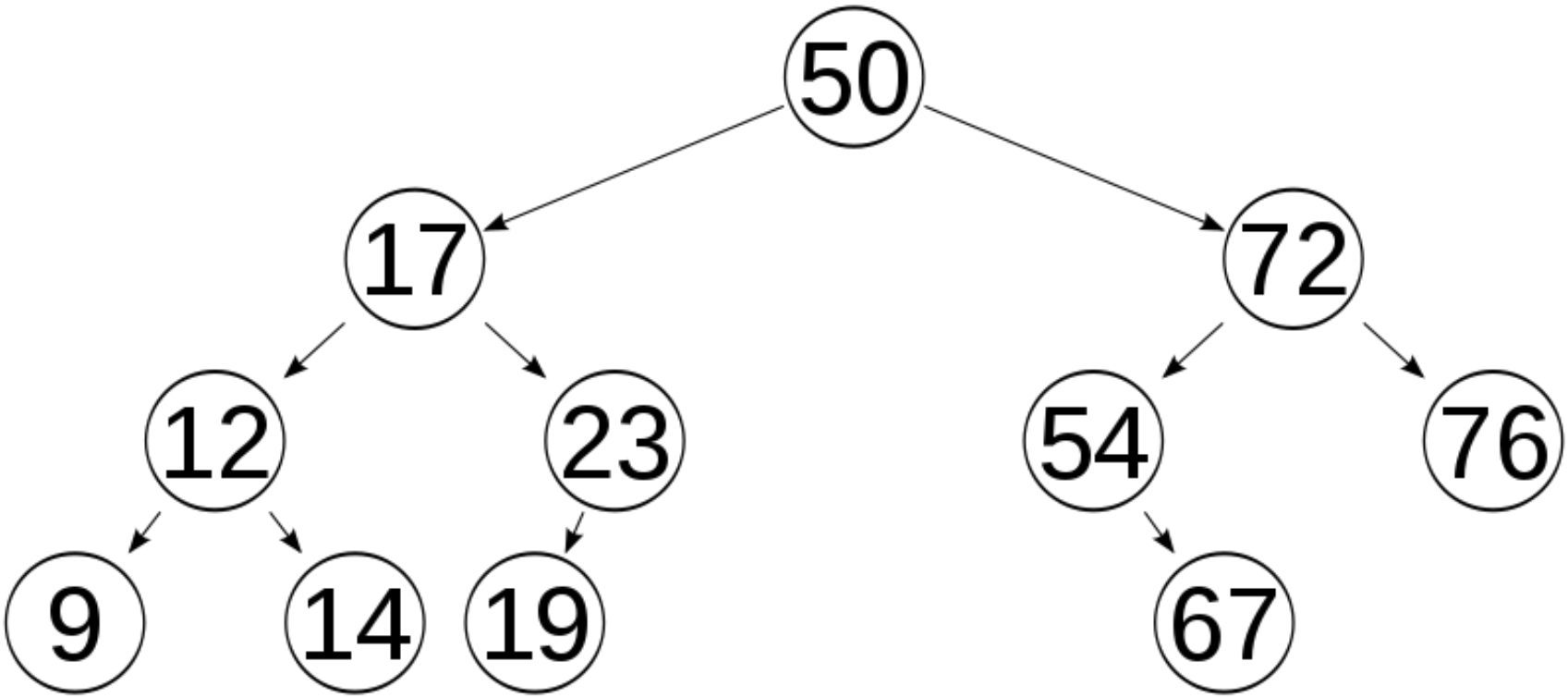
Course plan

1. Binary Search Trees
   - Implementation
2. AVL Trees
3. Red-Black Trees
4. Other types of search trees

Binary Search Trees

- Binary Search Trees (BST) are a type of binary tree in which children of each internal node are also trees
- For each node binary search property is satisfied
  - All values in the left subtree are lesser or equal to the node value
  - All values in the right subtree are greater then the node value
- Subtrees represented in BST are also binary search trees - the property is still valid!
- BSTs primary advantage is searching for an element in the tree and inserting new elements into this data structure
- They combine power of arrays and linked lists - arrays have fast search and slow updates, linked lists have slow search, but fast updates
- They are basis for other data structures: sets, multisets, associative arrays, that provide fast access to objects
- Inorder (LNR) traversal of BST returns elements in ascending order

BST example



Course plan

1. Binary Search Trees
   - Implementation
2. AVL Trees
3. Red-Black Trees
4. Other types of search trees

BST implementation

Properties

- `root` - root node of the tree
- `count` - number of elements in the tree

Methods

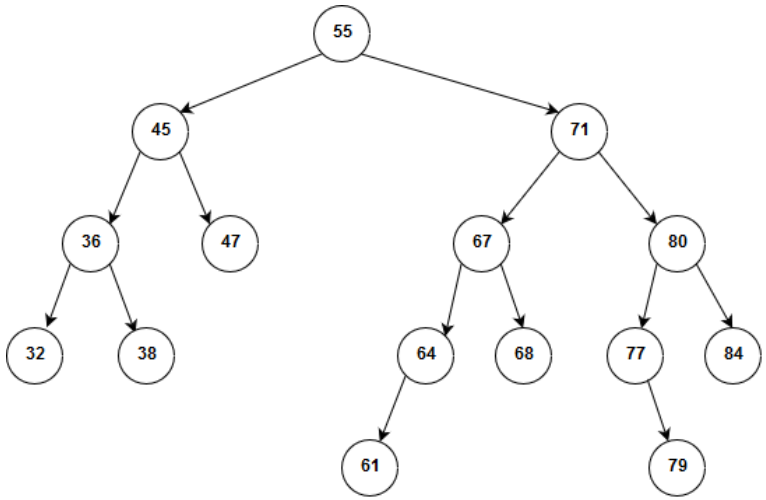
- `node search(key)` - returns node for which key is equal to searched value, null otherwise
- `node insert(key)` - inserts new node into BST and returns pointer to it
- `void delete(key)` - removes node with a given key from BST
- `string printSorted()` - prints elements in BST in sorted order using inorder traversal
- `node max()` - returns node with maximum key
- `node min()` - returns node with minimum key
- `node successor(node)` - returns next node in BST following tree order
- `node predecessor(node)` - returns previous node in BST

BST implementation details

- All operations take advantage of BST property - all nodes in left subtree are lesser or equal and all nodes in right subtree are greater than the value of specified node
- After every modification operation (delete and insert) the BST property must be maintained

Visualization of BST algorithms

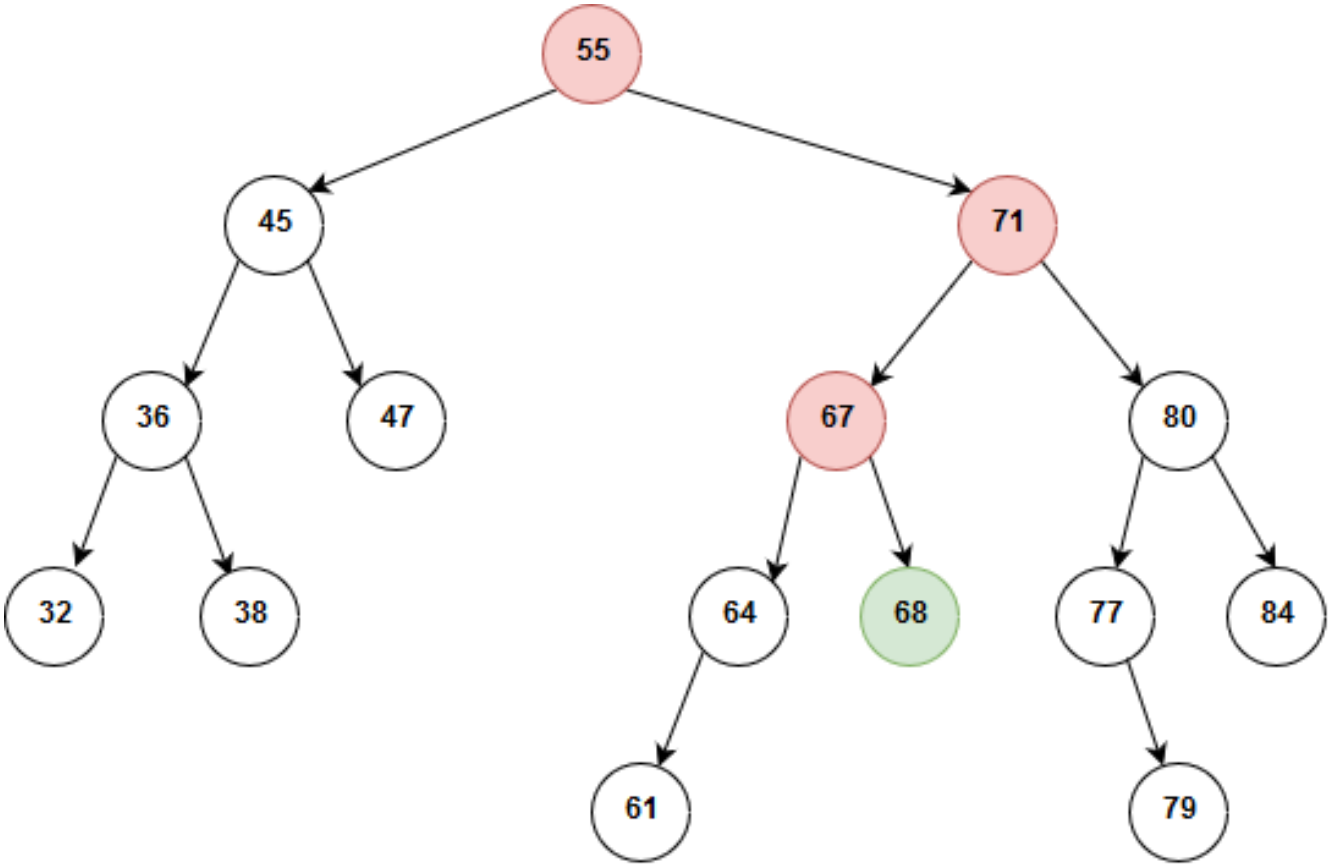
http://btv.melezinek.cz/binary-search-tree.html



BST search

```
1  BSTSearchRecursive(key, currentNode (default root))
2    if currentNode == NULL OR currentNode.key = key
3      return currentNode
4    if currentNode.key < key
5      return BSTSearch(key, currentNode.left)
6    else
7      return BSTSearch(key, currentNode.right)
8
9  BSTSearchIterative(key)
10   currentNode = root
11   while currentNode != NULL AND currentNode.key != key
12     if k < currentNode.key
13       currentNode = currentNode.left
14     else
15       currentNode = currentNode.right
16   return currentNode
17
```

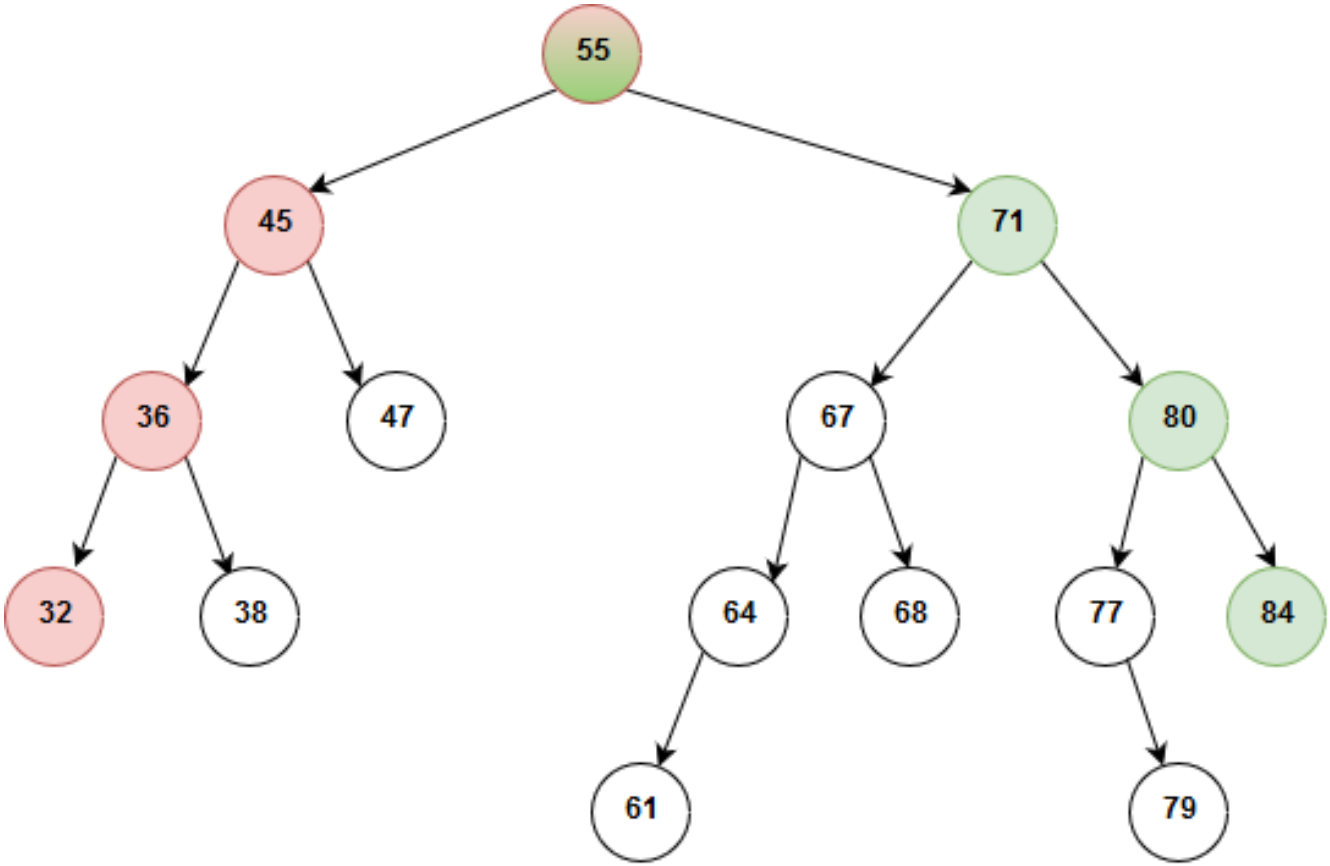
BST search



BST Min Max

```
1 BSTMin(node (default root))
2   while node.left != NULL
3     node = node.left
4   return node
5
6 BSTMax(node (default root))
7   while node.right != NULL
8     node = node.right
9   return node
```

BST Min Max



BST Successor

```
1 Successor(node)
2   if node.right != NULL
3     return BSTMin(node.right)
4   currentParent = node.parent
5   while currentParent != NULL AND node == currentParent.right
6     node = currentParent
7     currentParent = currentParent.parent
8   return currentParent
```

If a nodes right subtree is empty, then a nodes successor is its lowest ancestor, whose left child is also an ancestor of that node (recall that every node is its own ancestor).

Sorting binary tree

```
1 printSorted()
2   Inorder(root)
3
4 Inorder(node)
5   return if node == null
6   Inorder(node.leftChild)
7   print node.data
8   Inorder(node.rightChild)
```

- Inorder traversal visits each node only once in Left-Node-Right order
- Complexity of inorder traversal (and print sorted) is then $O(n)$
- If we have already built BST getting elements back in order are quite fast operation!

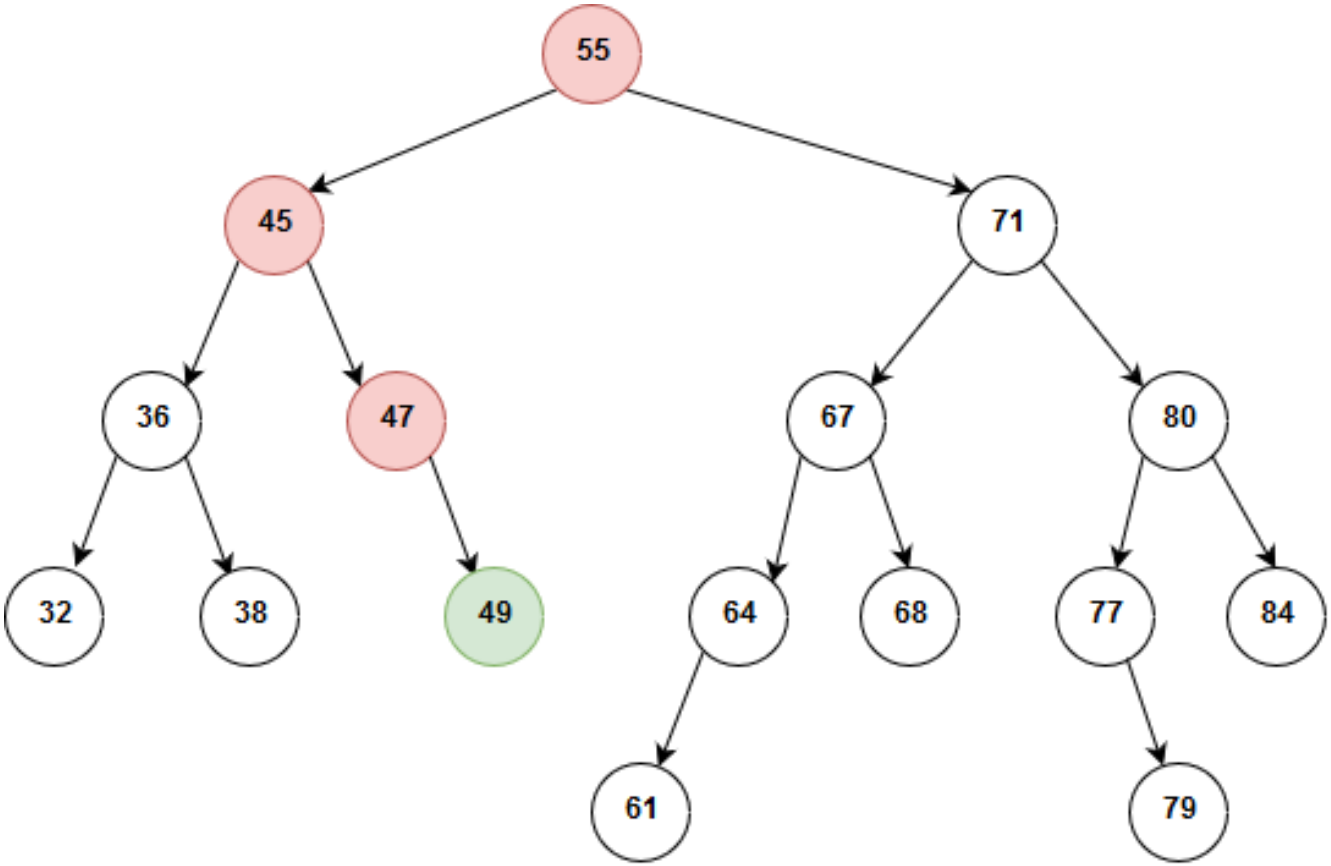
BST Insert

- We always insert at the bottom of tree
- First we search for an empty place (leaf) to put new key following BST property
- When such a place is found new node at that location is created and child property of parent is updated

BST Insert pseudocode

```
1  BSTInsert(key)
2    currentParent = NULL
3    currentNode = root
4    while currentNode != NULL
5        currentParent = currentNode
6        if key <= currentNode.key
7          currentNode = currentNode.left
8        else
9          currentNode = currentNode.right
10   currentNode = new Node(key)
11   currentNode.parent = currentParent
12   if currentParent == NULL //Tree was empty
13     root = currentNode
14   else if key <= currentParent.key
15     currentParent.left = currentNode
16   else
17     currentParent.right = currentNode
18   return currentNode
```

BST Insert example



BST delete

- Deletion of a node in BST is a little bit tricky, we need to distinguish three possible cases:
  - Node does not have any children - we simply remove the node
  - Node has exactly one child - we elevate that child to take the node's position in the tree
  - If node has two children we need to find a successor to the node, which must be in right subtree, and have that successor take place of node in the tree.
- We define a function `Transplant(node, node)` to help us with node deletion - given two nodes it will put the second one in the place of first

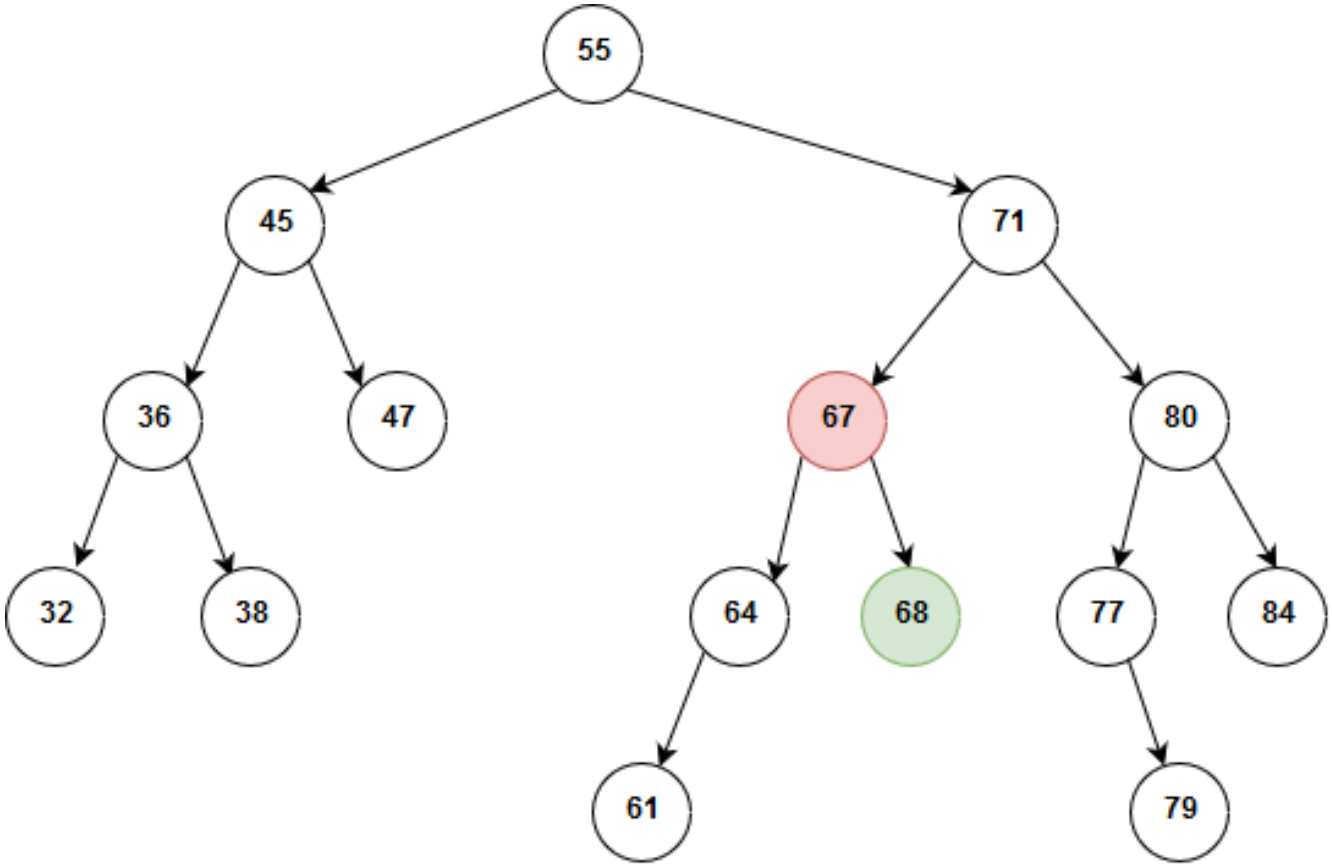
BST Transplant pseudocode

```
1  Transplant(originalNode, replacementNode)
2    if originalNode.parent = NULL
3      root = replacementNode
4    else if originalNode == originalNode.parent.left
5      originalNode.parent.left = replacementNode
6    else
7      originalNode.parent.right = replacementNode
8    if replacementNode != NULL
9      replacementNode.parent = originalNode.parent
10   return replacementNode
```

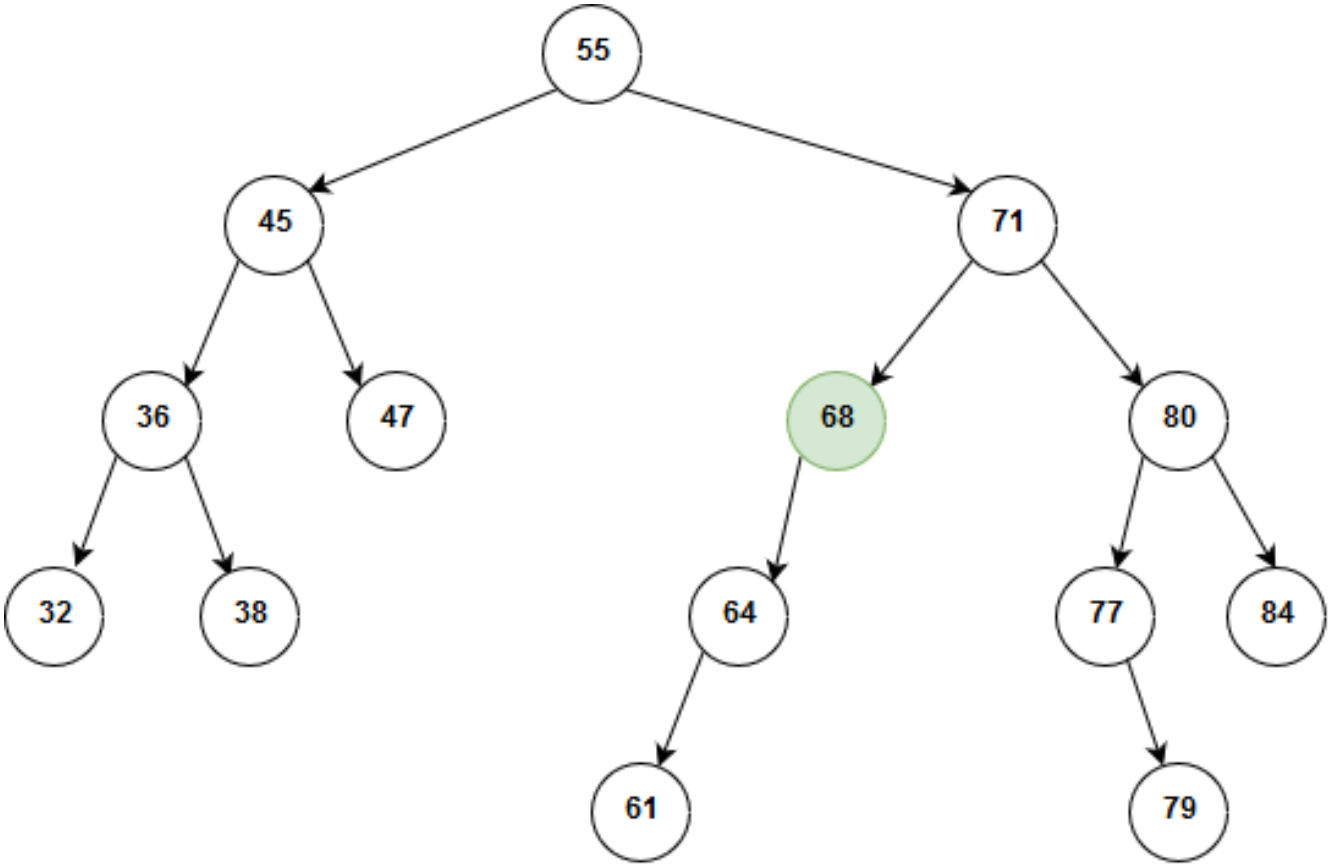
BST delete pseudocode

```
1  BSTDelete(node)
2    if node.left == NULL
3      return Transplant(node, node.right)
4    else if node.right == NULL
5      return Transplant(node, node.left)
6    else
7      successor = BSTMin(node.right)
8      if successor.parent != node
9        Transplant(successor, successor.right)
10       successor.right = node.right
11       successor.right.parent = successor
12     Transplant(node, successor)
13     successor.left = node.left
14     successor.left.parent = successor
15     return successor
```

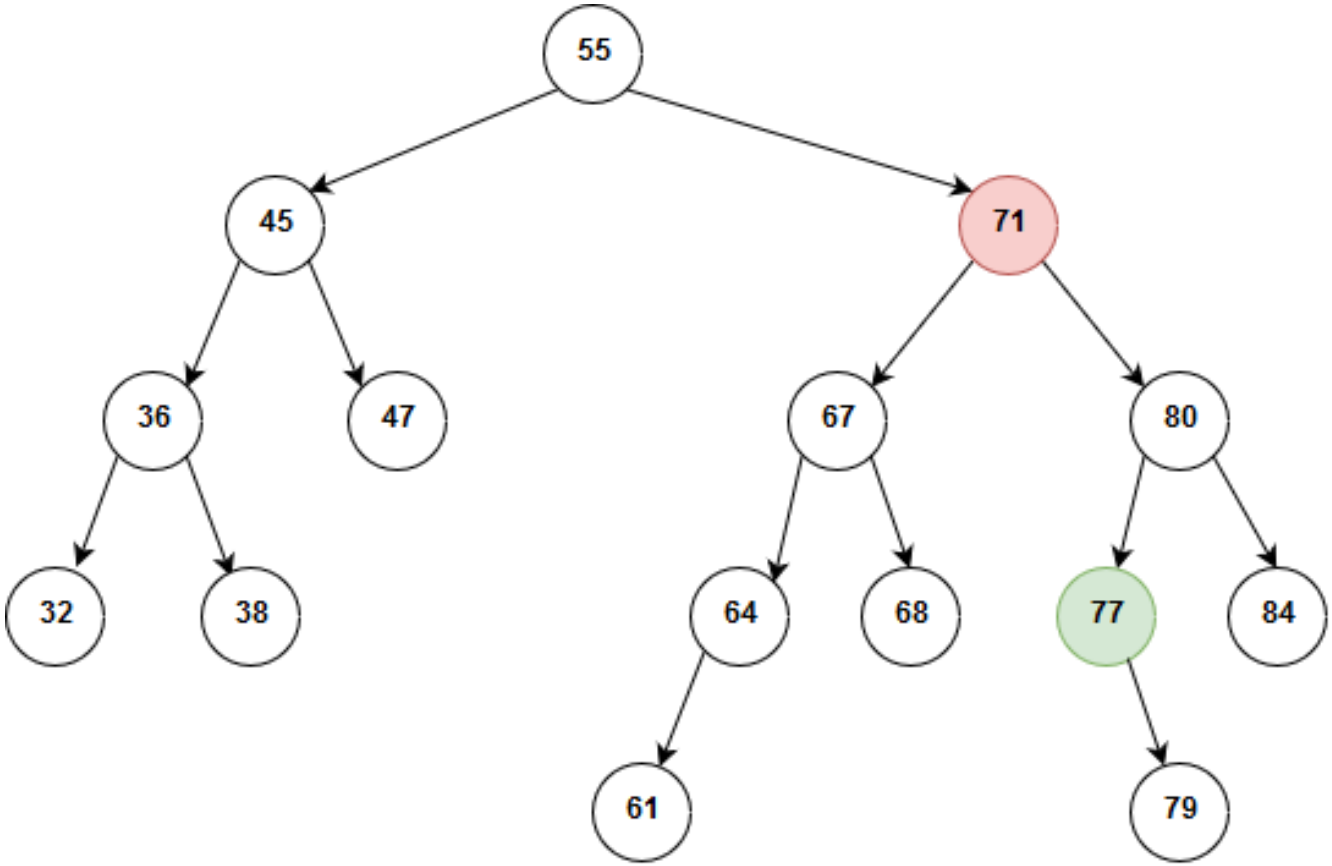
BST delete - example 1



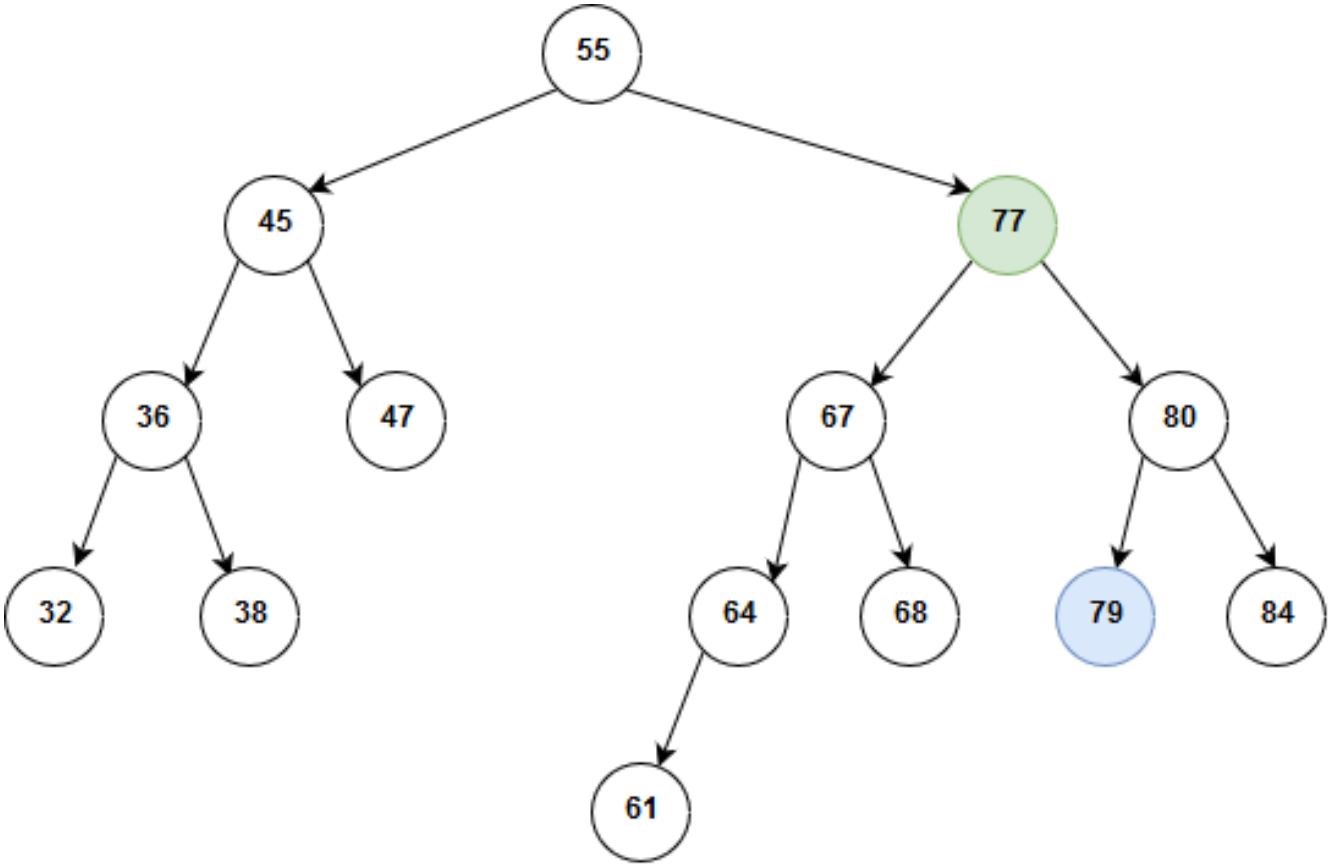
BST delete - example 1



BST delete - example 2



BST delete - example 2



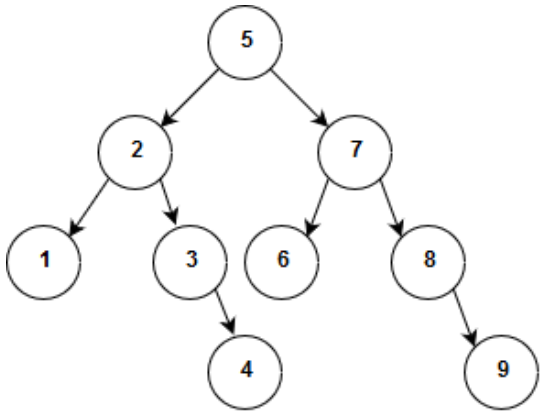
BST performance

- Performance of all operations in BST is related to height of the tree
- Based on binary trees properties we can then say, that complexity is $O(lgn)$ in best case, but $O(n)$ in worst case
- Base case scenario is tree fully filled at every level
- Worst case scenario is only one node at each level - unbalanced (degenerate) tree
- Worst case scenario can occur if we add new keys in incorrect order

Balanced vs unbalanced BST

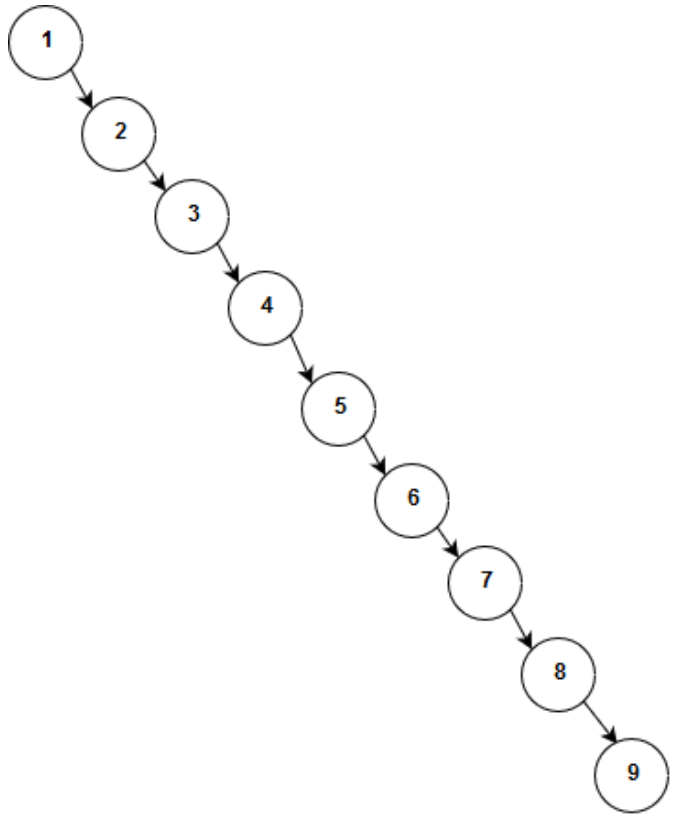
Balanced

Insertion sequence: $5, 2, 7, 1, 3, , 6, 8, 4, 9$



Unbalanced

Insertion sequence: $1, 2, 3, 4, 5, 6, 7, 8, 9$



Course plan

1. Binary Search Trees
   - Implementation
2. AVL Trees
3. Red-Black Trees
4. Other types of search trees

AVL Trees

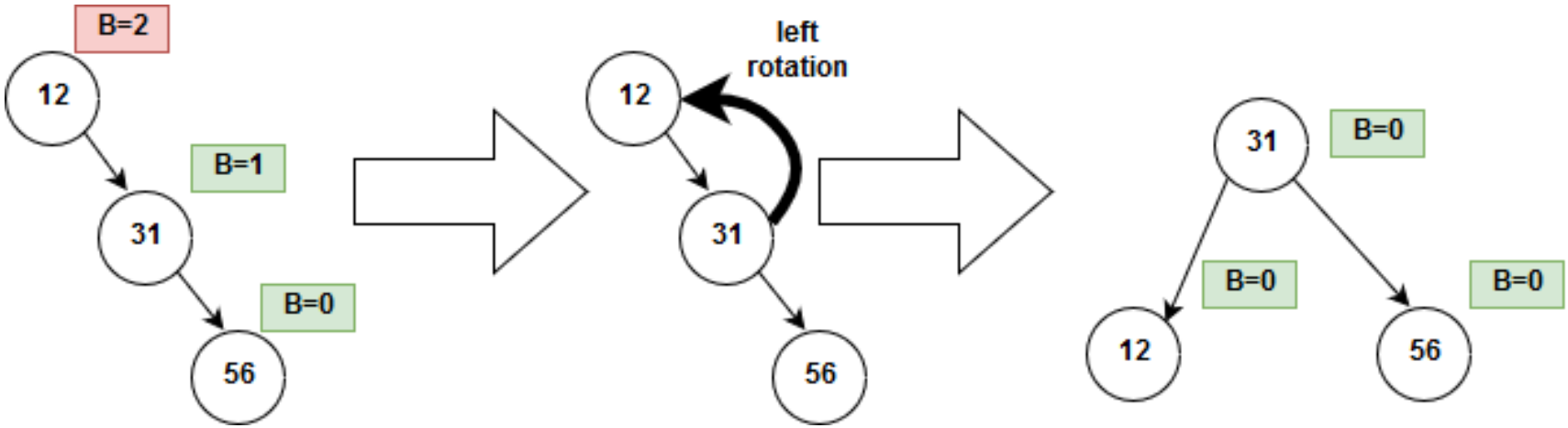
- Performance of operations on BST depend on height of a tree.
- For a given number of nodes multiple different trees can be created, often resulting in suboptimal (unbalanced) trees
- There are however tree data structures that are self-balanced
- One of them is AVL (Adelson-Velsky and Landis) Tree, developed in 1962.
- AVL tree is upgraded version of BST, that maintains balance while inserting/deleting nodes
- We say, that a binary search tree is balanced if any two sibling subtrees do not differ in height by more than one
- Let's introduce balance factor for a node as height difference between left and right subtree$B = node.right.heigh() - node.left.height()$
- In AVL Tree $B \in \{-1, 0, 1\}$
- AVL are first self-balancing trees introduced in computer science

AVL rotations

- During insertion or deletion of a node, AVL tree might become unbalance
- Balance is restored by performing rotations on nodes, either single or double.
- We distinguish between the following rotations:
  - Left rotation
  - Right rotation
  - Left-right rotation
  - Right-left rotation

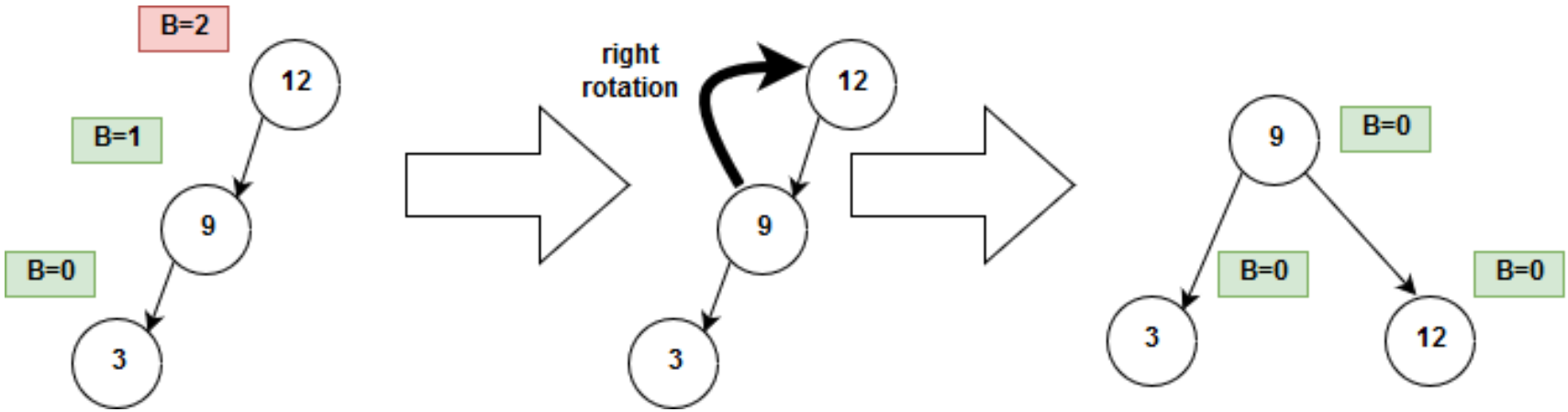
AVL left rotation

- Left rotation is performed, if a node is inserted as a right child of right subtree and it causes the tree to become unbalanced



AVL right rotation

- Right rotation is performed, if a node is inserted as a left child of left subtree and it causes the tree to become unbalanced

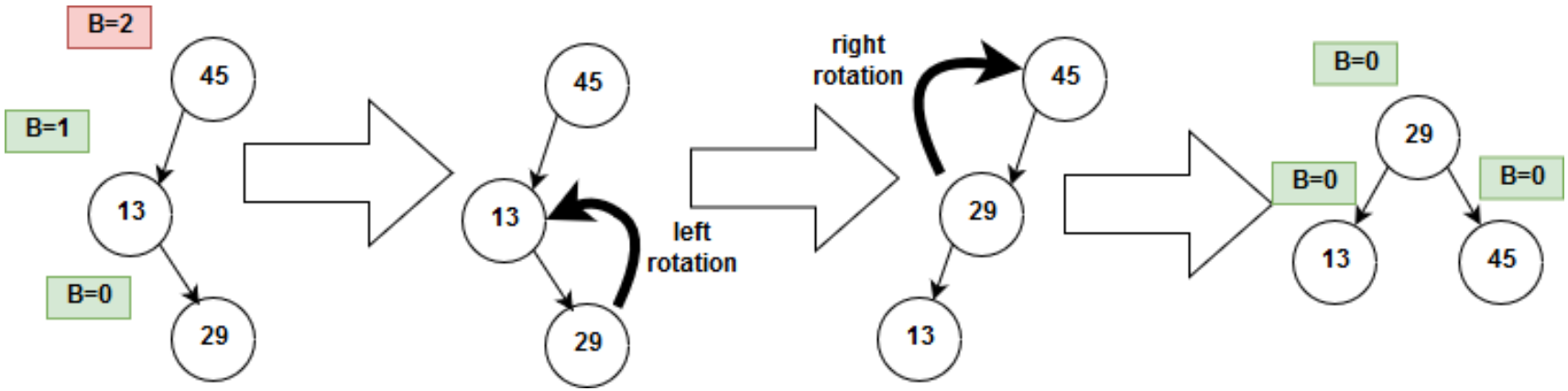


AVL single rotation pseudocode

```
1  RightRotate(node)  {
2     newRoot = node.left
3     node.left = newRoot.right
4     newRoot.right = node
5        return newRoot
6  }
7
8  LeftRotate(node){
9     newRoot = node.right
10    node.right = newRoot.left
11    newRoot.left = node
12    return newRoot
13 }
```

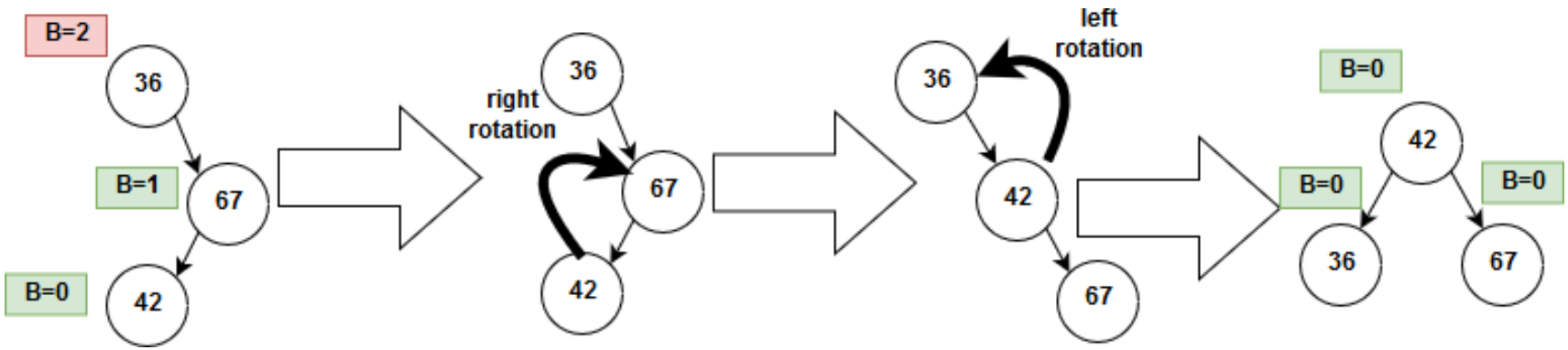
AVL left-right rotation

- If imbalance is introduced by inserting node as right child in left subtree we perform left-right rotation
- First we do left rotation to obtain still unbalanced, but familiar situation
- Then we restore balance by performing right rotation



AVL right-left rotation

- If imbalance is introduced by inserting node as left child in right subtree we perform right-left rotation
- First we do right rotation to obtain still unbalanced, but familiar situation
- Then we restore balance by performing left rotation



AVL double rotations pseudocode

```
1 RightLeftRotate(node)  {
2   node.right = RightRotate(node.right)
3   return LeftRotate(node)
4 }
5
6 LeftRightRotate(node){
7   node.left = LeftRotate(node.left)
8   return RightRotate(node)
9 }
```

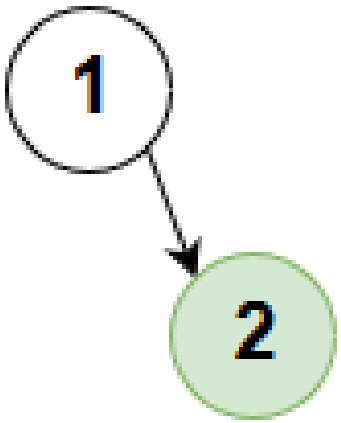
AVL Insert

- Inserting a node into AVL tree begins with simple BST insert followed by restoring balance

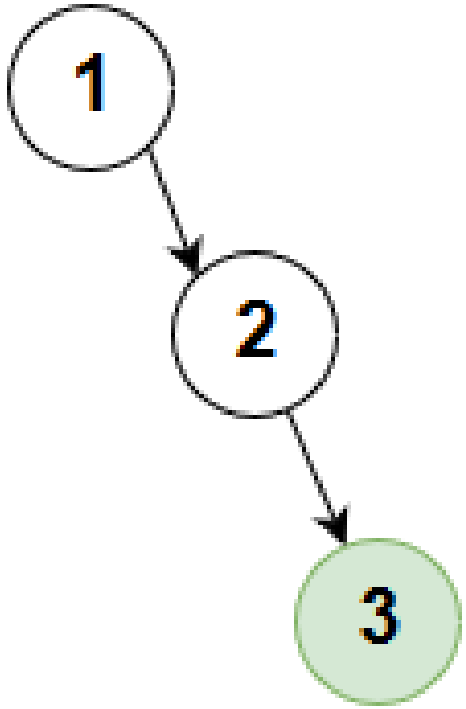
```
1  AVLInsert(key){
2    node = BSTInsert(key)
3    balance = GetBalance(node)
4    if (balance > 1 && key < node.left.key)
5      return RightRotate(node);
6    if (balance < -1 && key > node.right.key)
7      return LeftRotate(node);
8    if (balance > 1 && key > node.left.key){
9      node.left = LeftRotate(node.left);
10     return RightRotate(node);
11   }
12   if (balance < -1 && key < node.right.key){
13     node.right = RightRotate(node.right);
14     return LeftRotate(node);
15   }
16   return node;
17 }
```

AVL Insert example

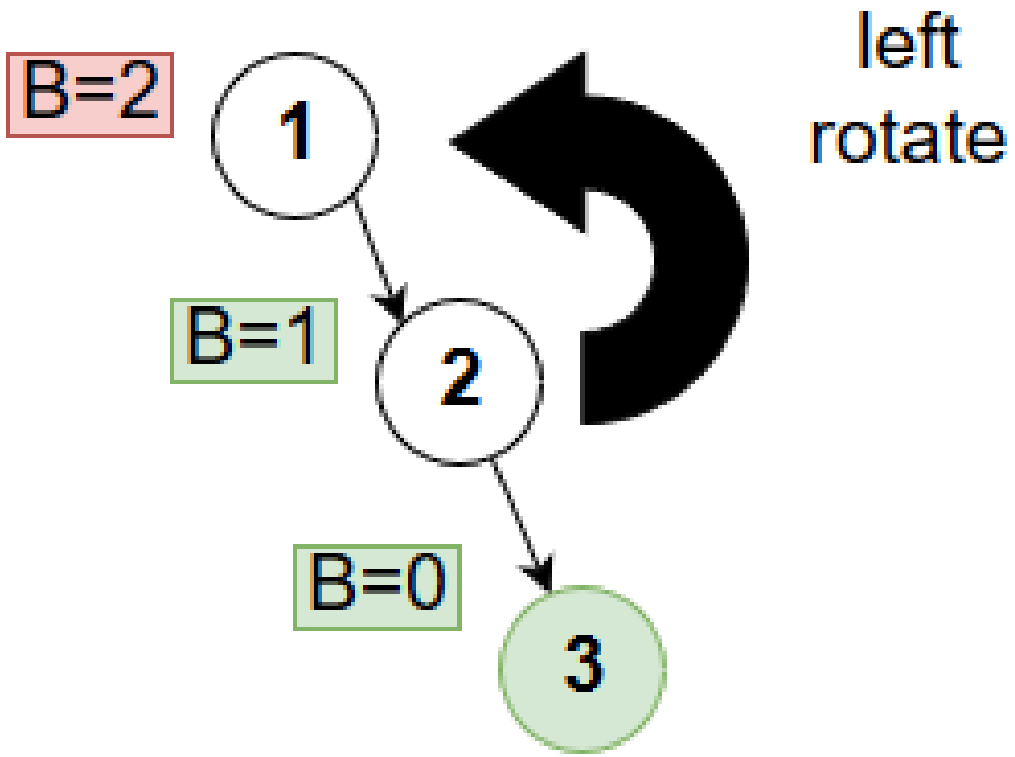
AVL Insert example



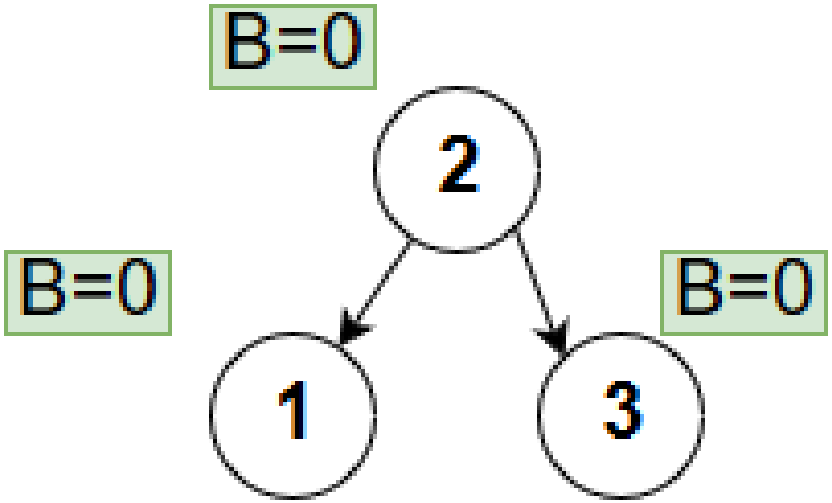
AVL Insert example



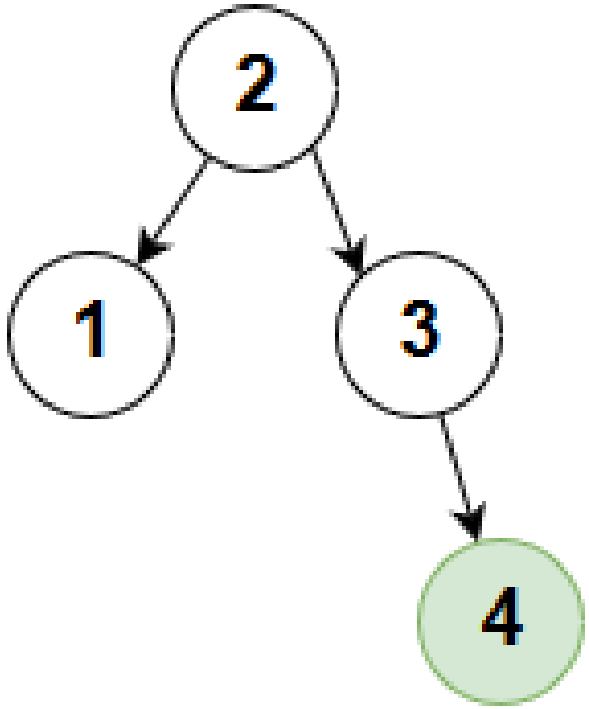
AVL Insert example



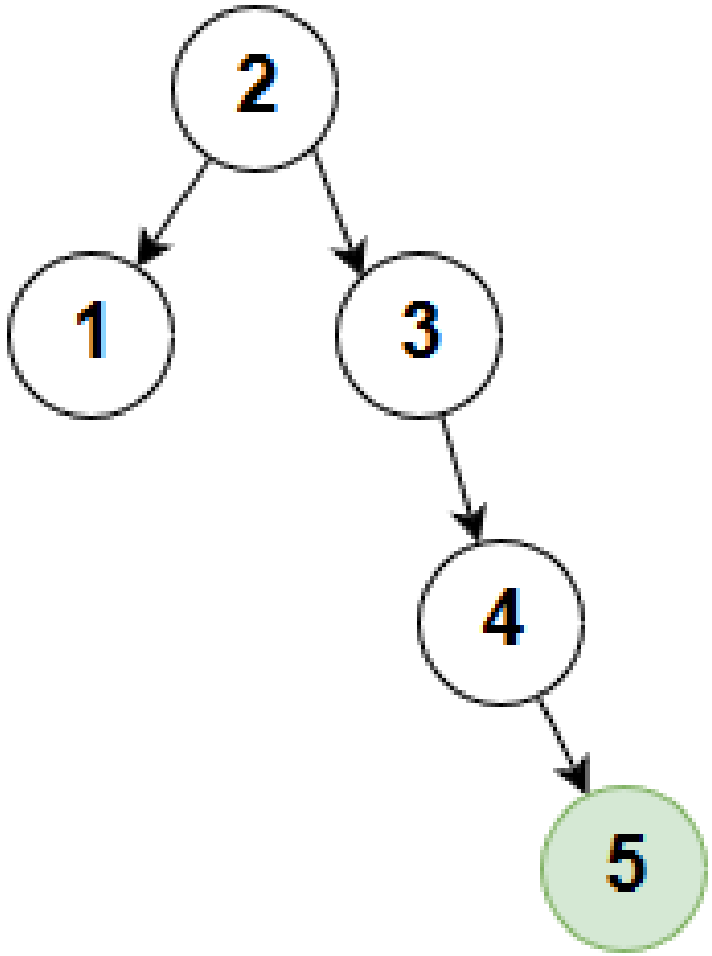
AVL Insert example



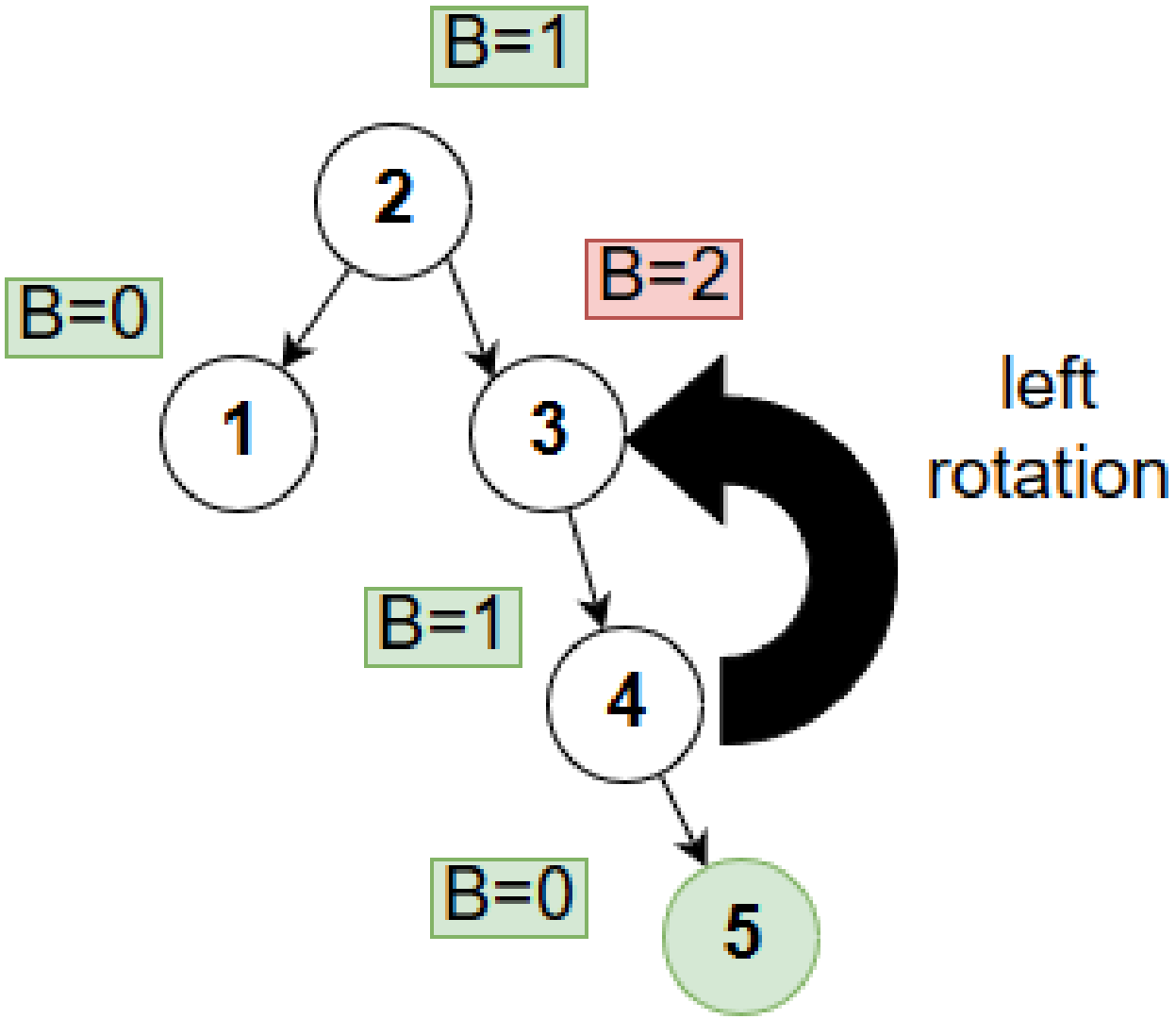
AVL Insert example



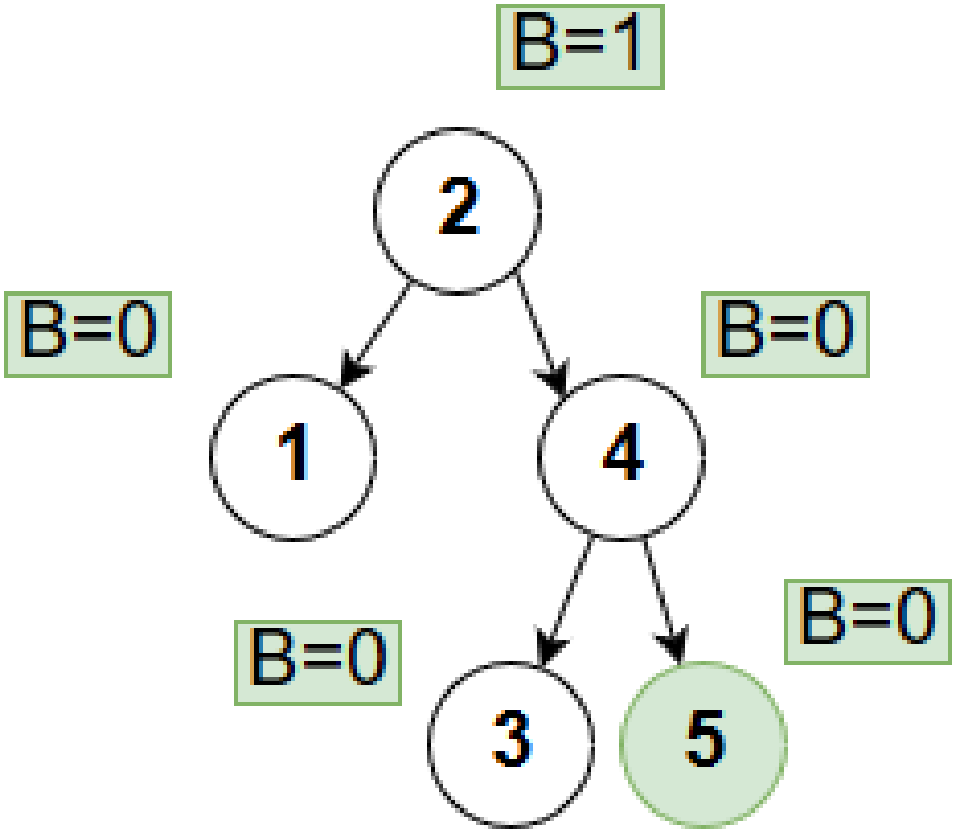
AVL Insert example



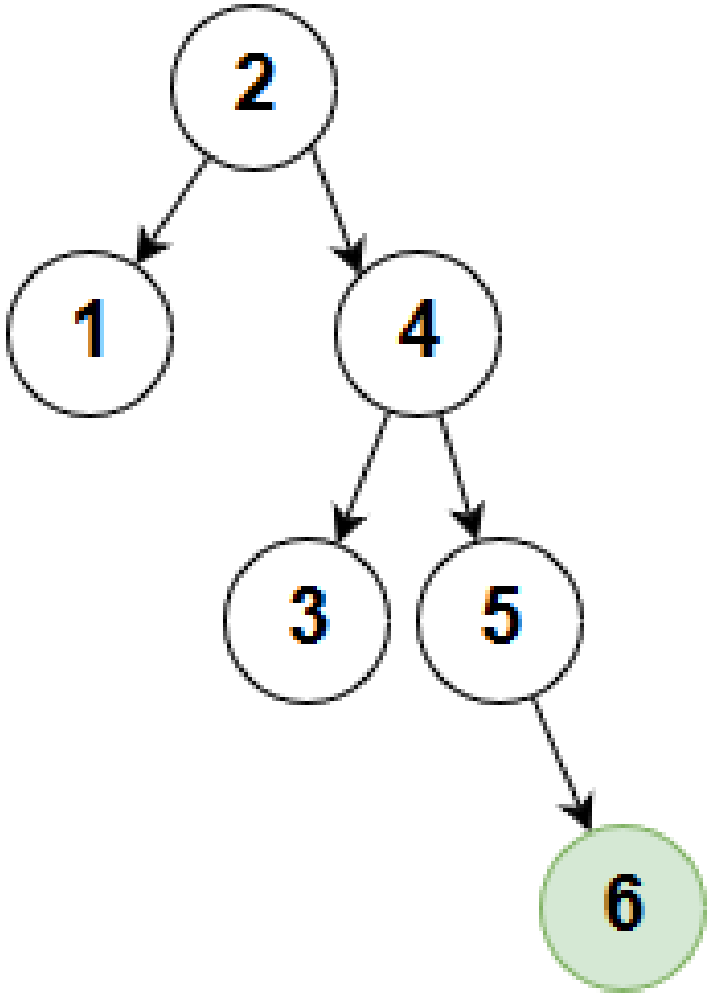
AVL Insert example



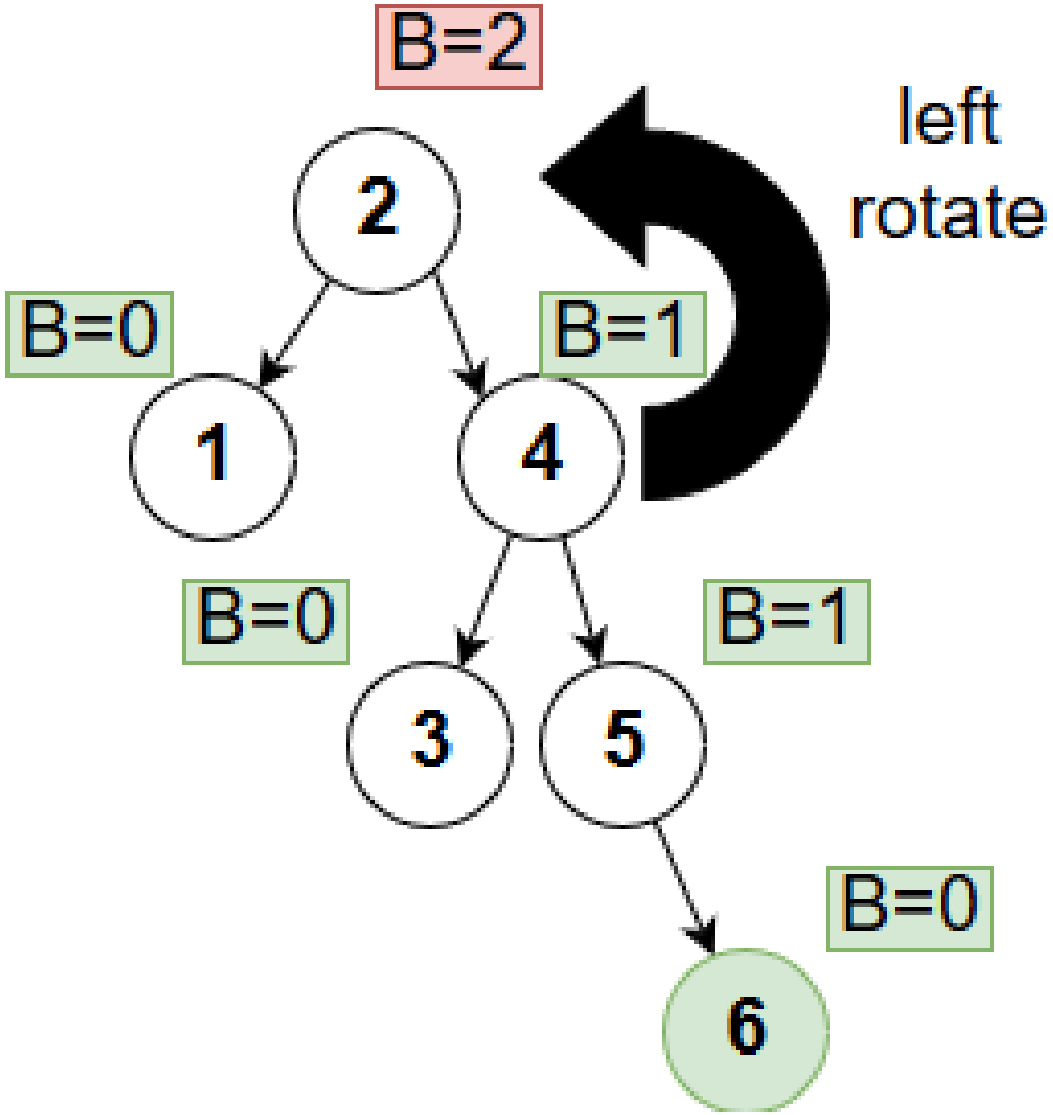
AVL Insert example



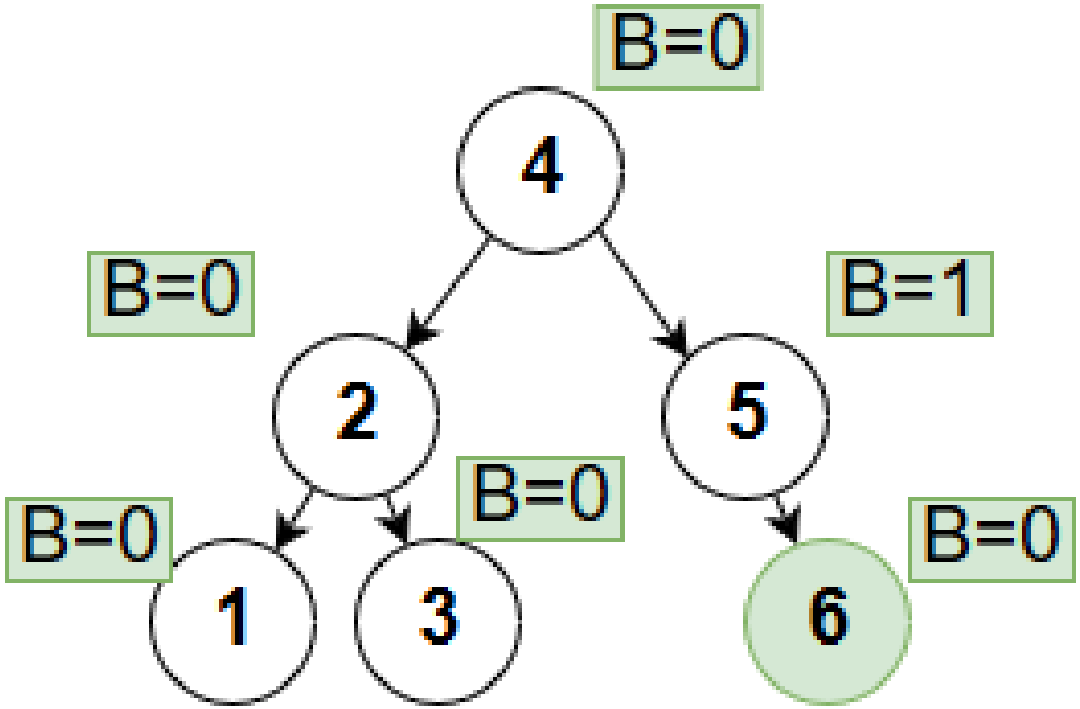
AVL Insert example



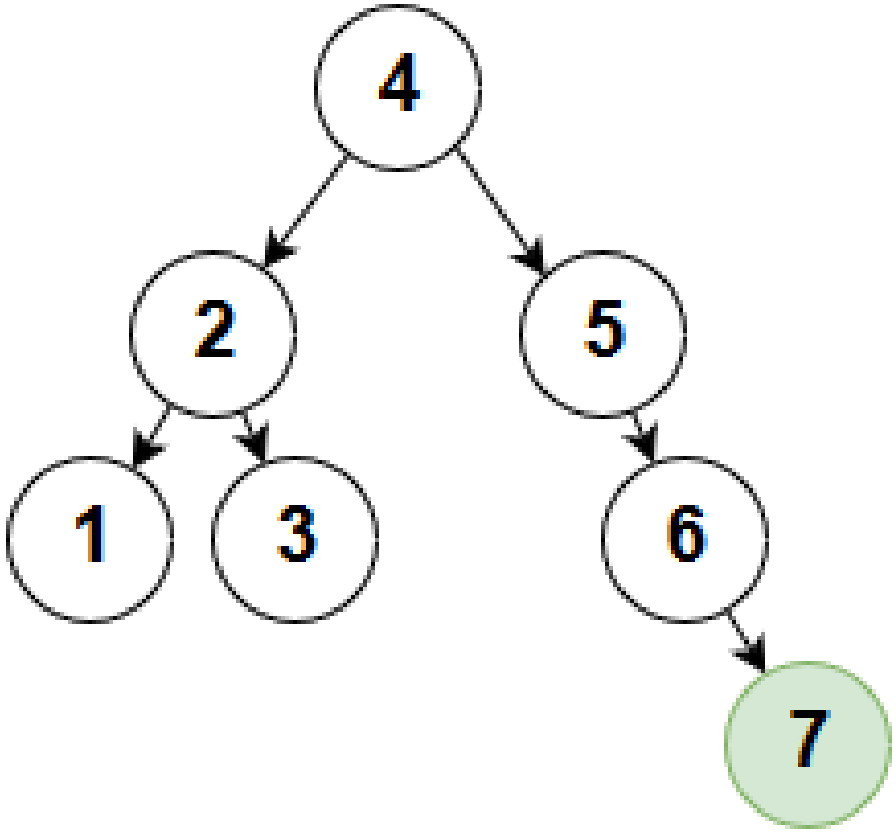
AVL Insert example



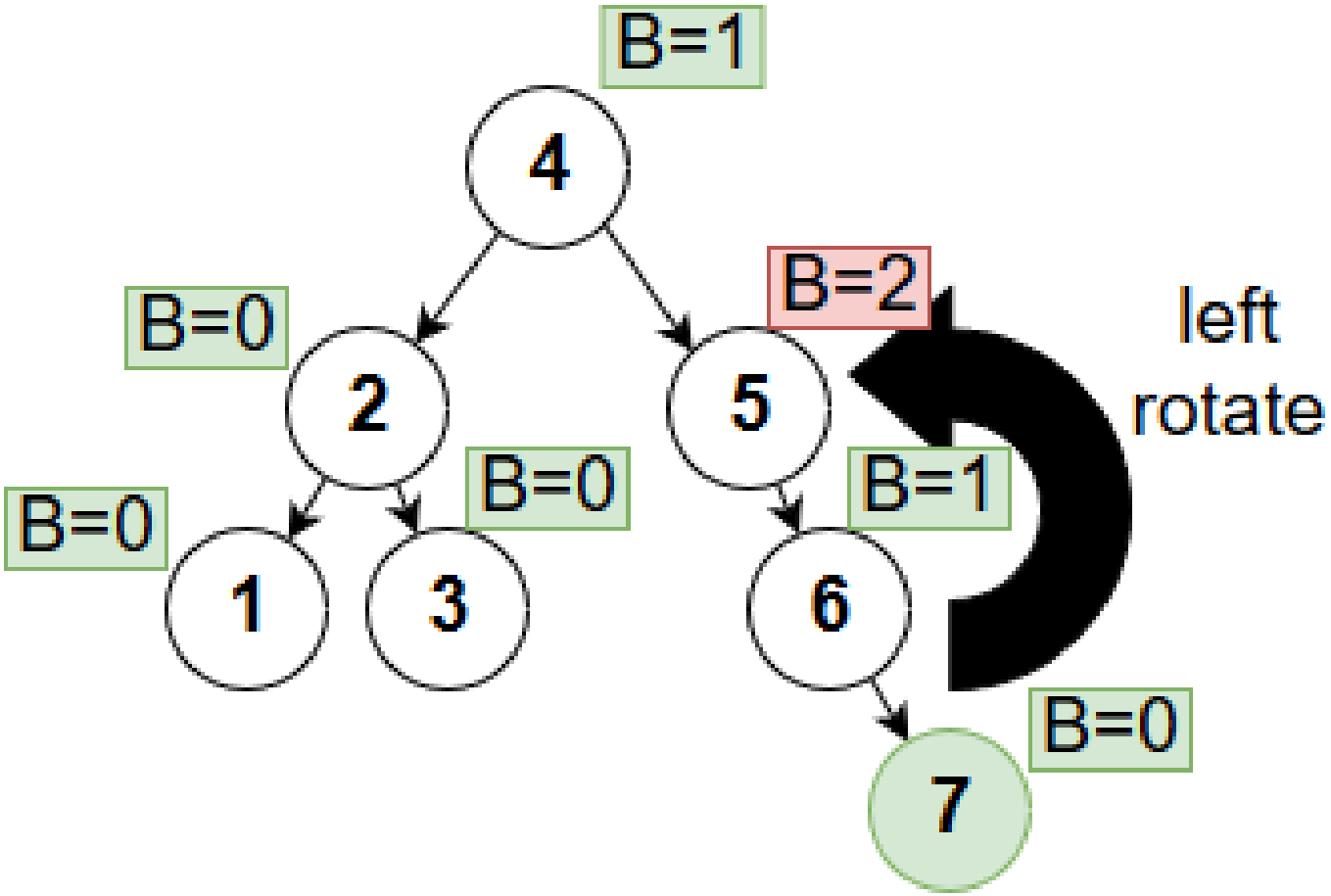
AVL Insert example



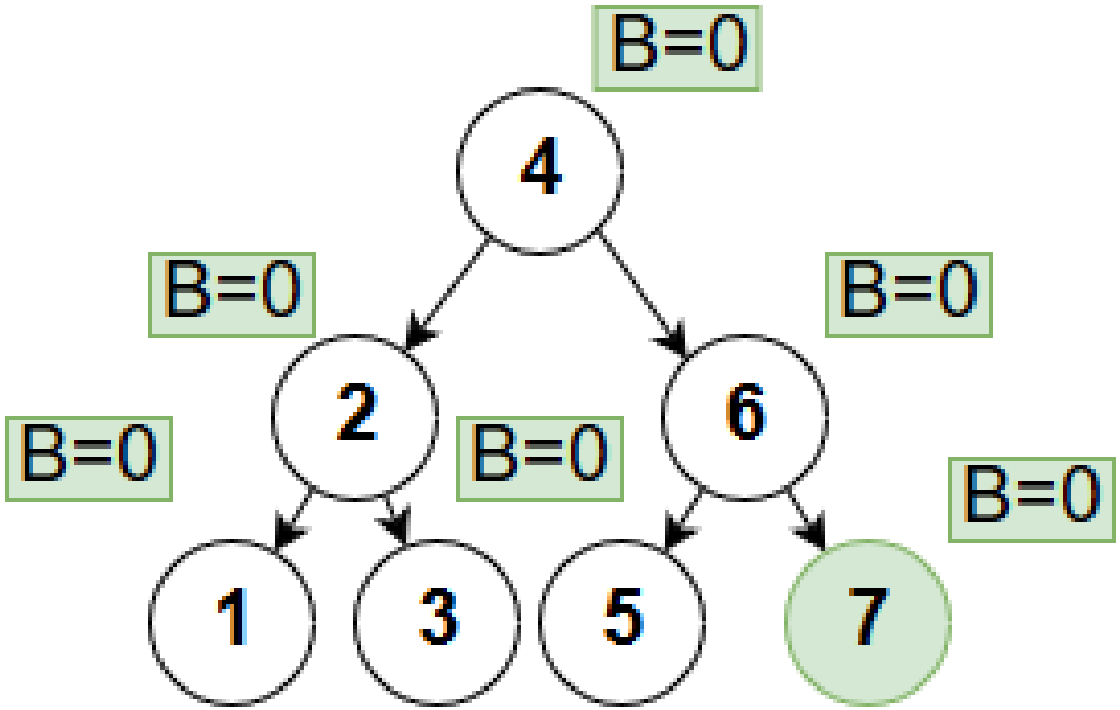
AVL Insert example



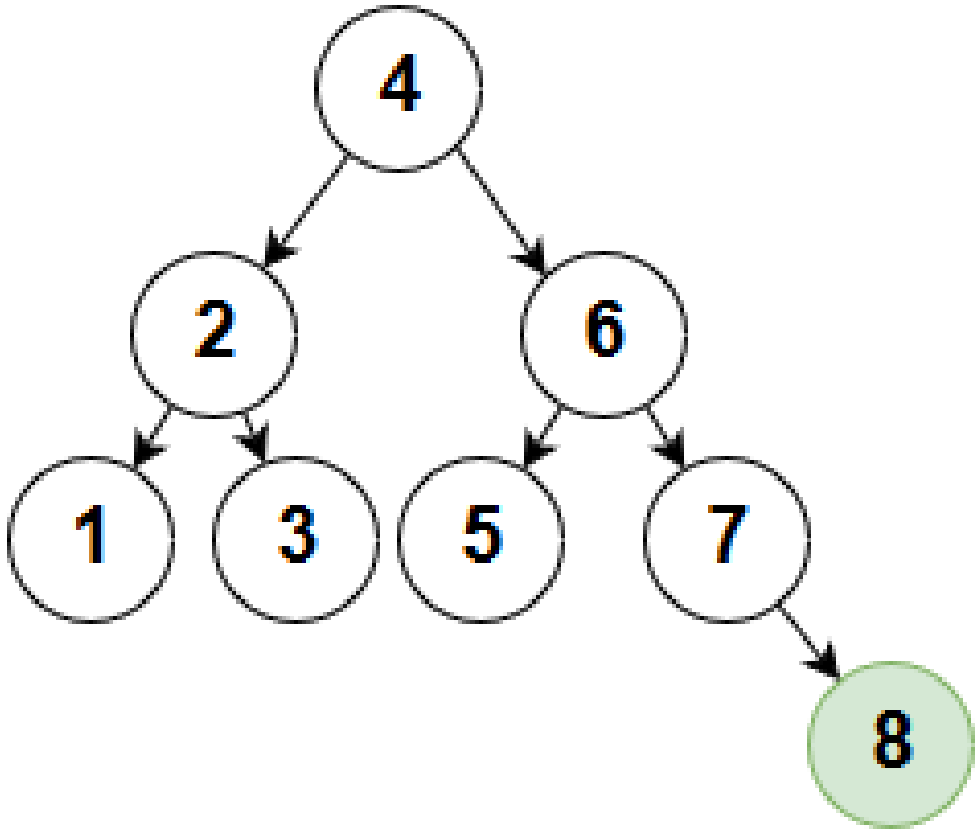
AVL Insert example



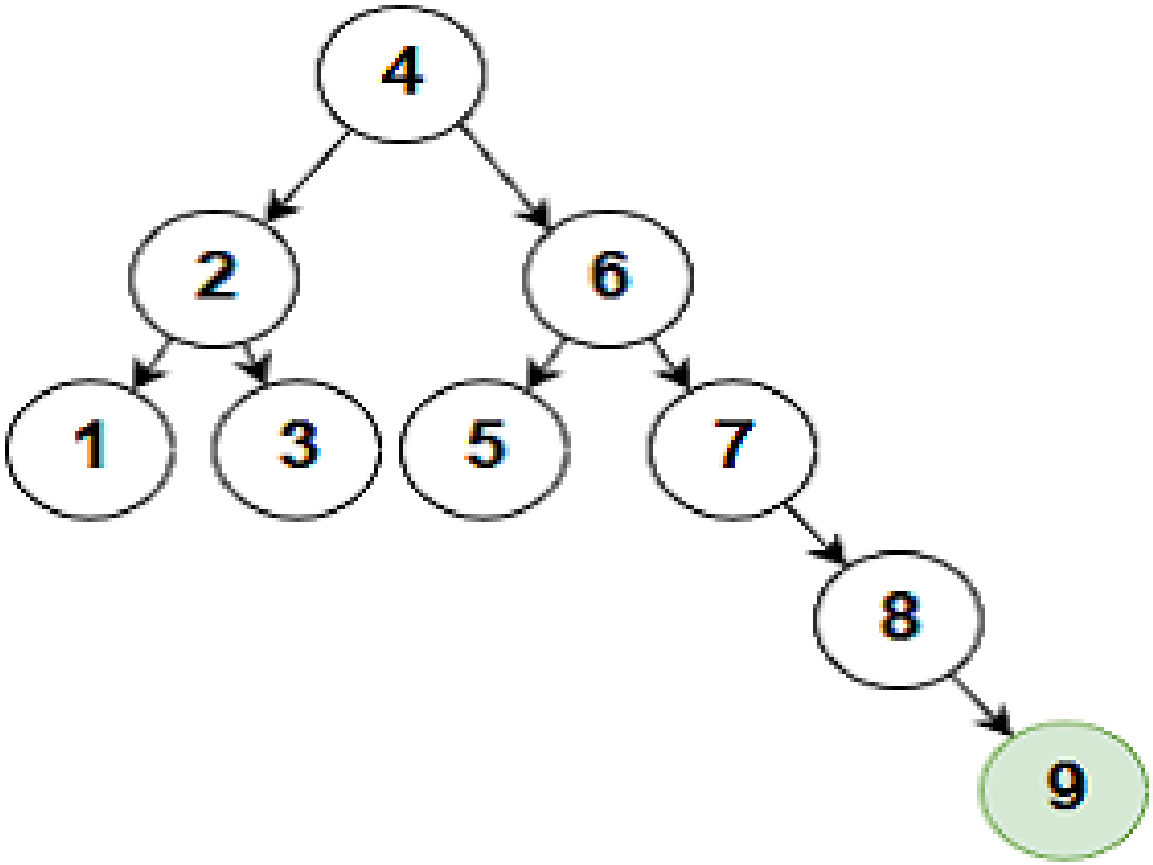
AVL Insert example



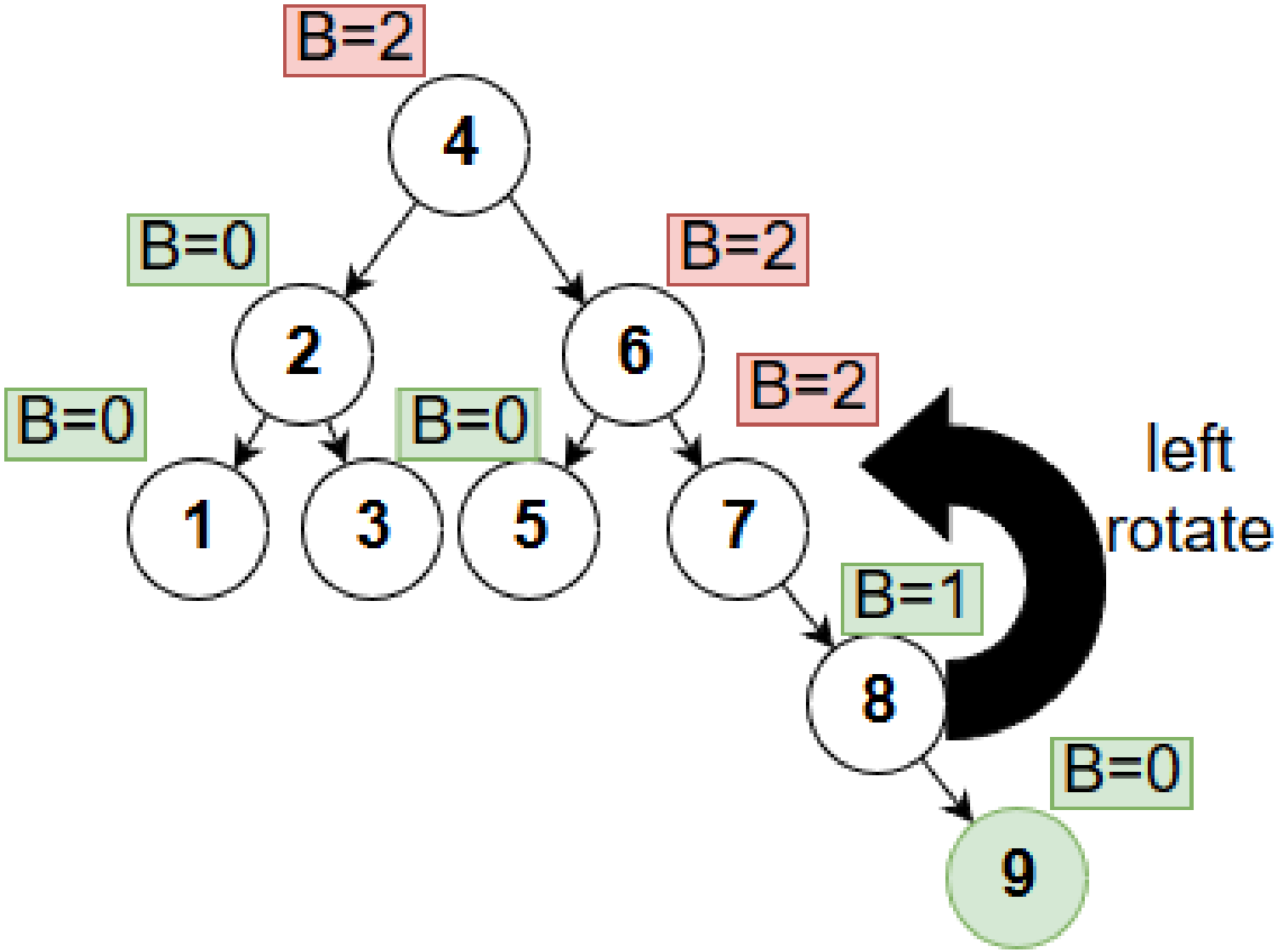
AVL Insert example



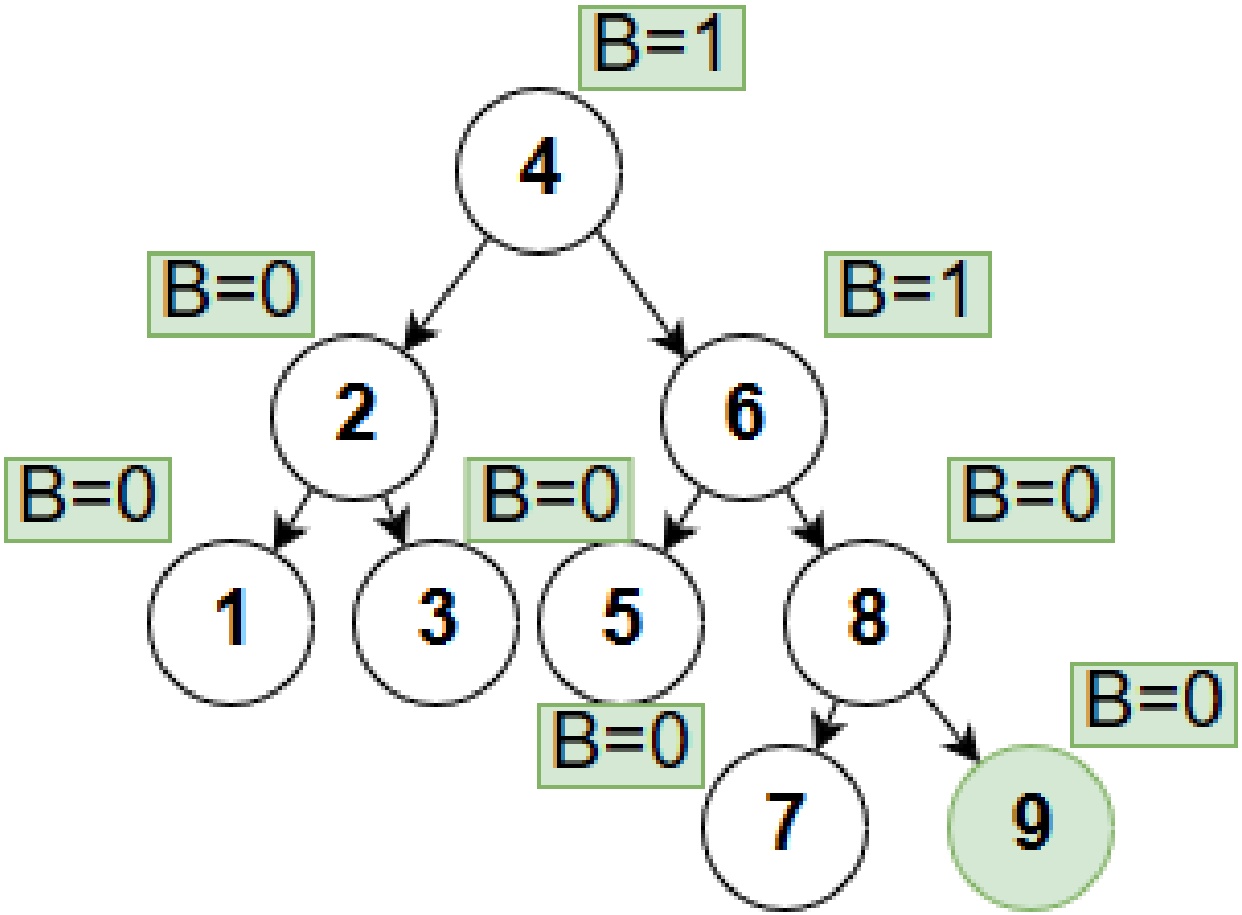
AVL Insert example



AVL Insert example



AVL Insert example

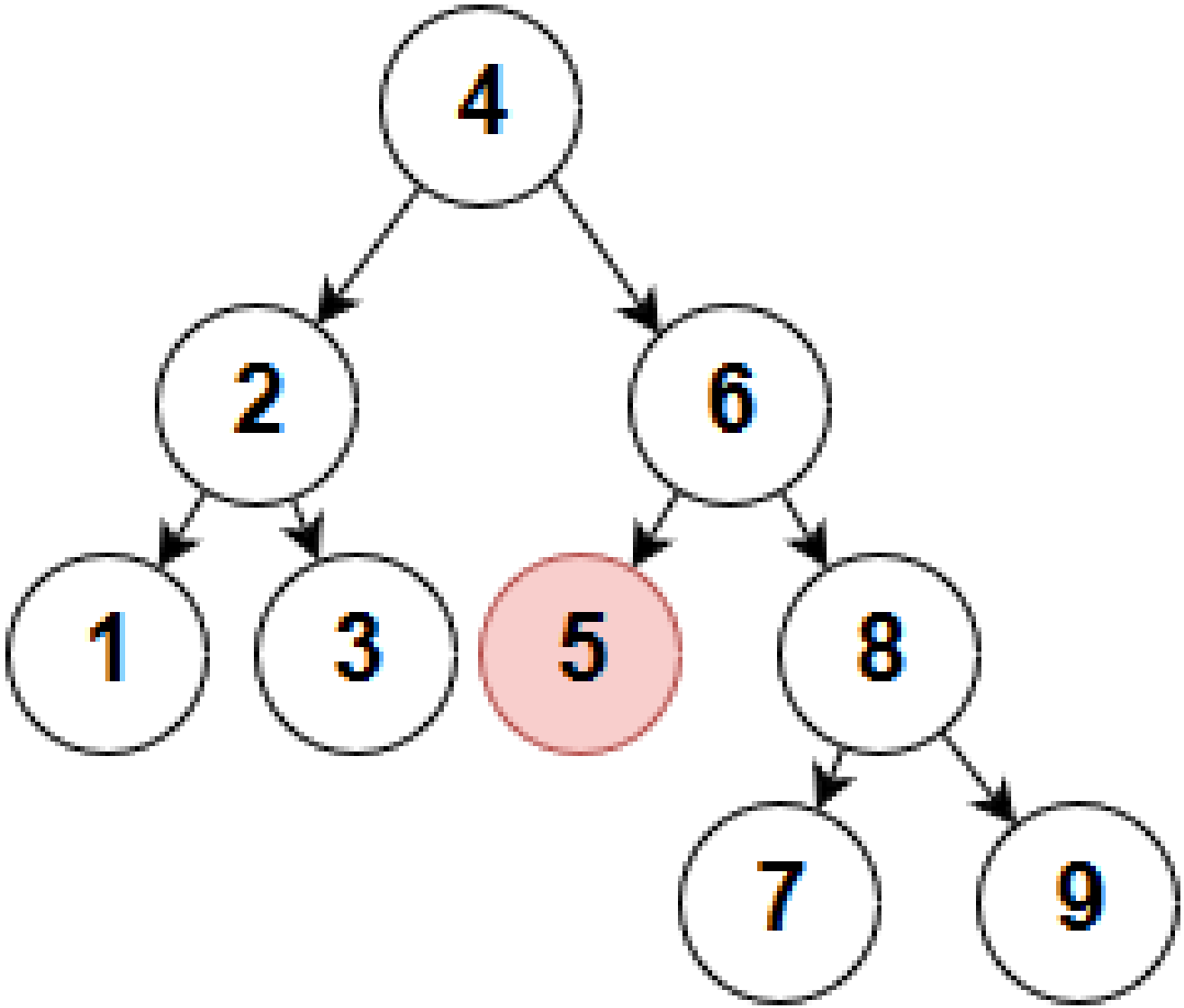


AVL Delete

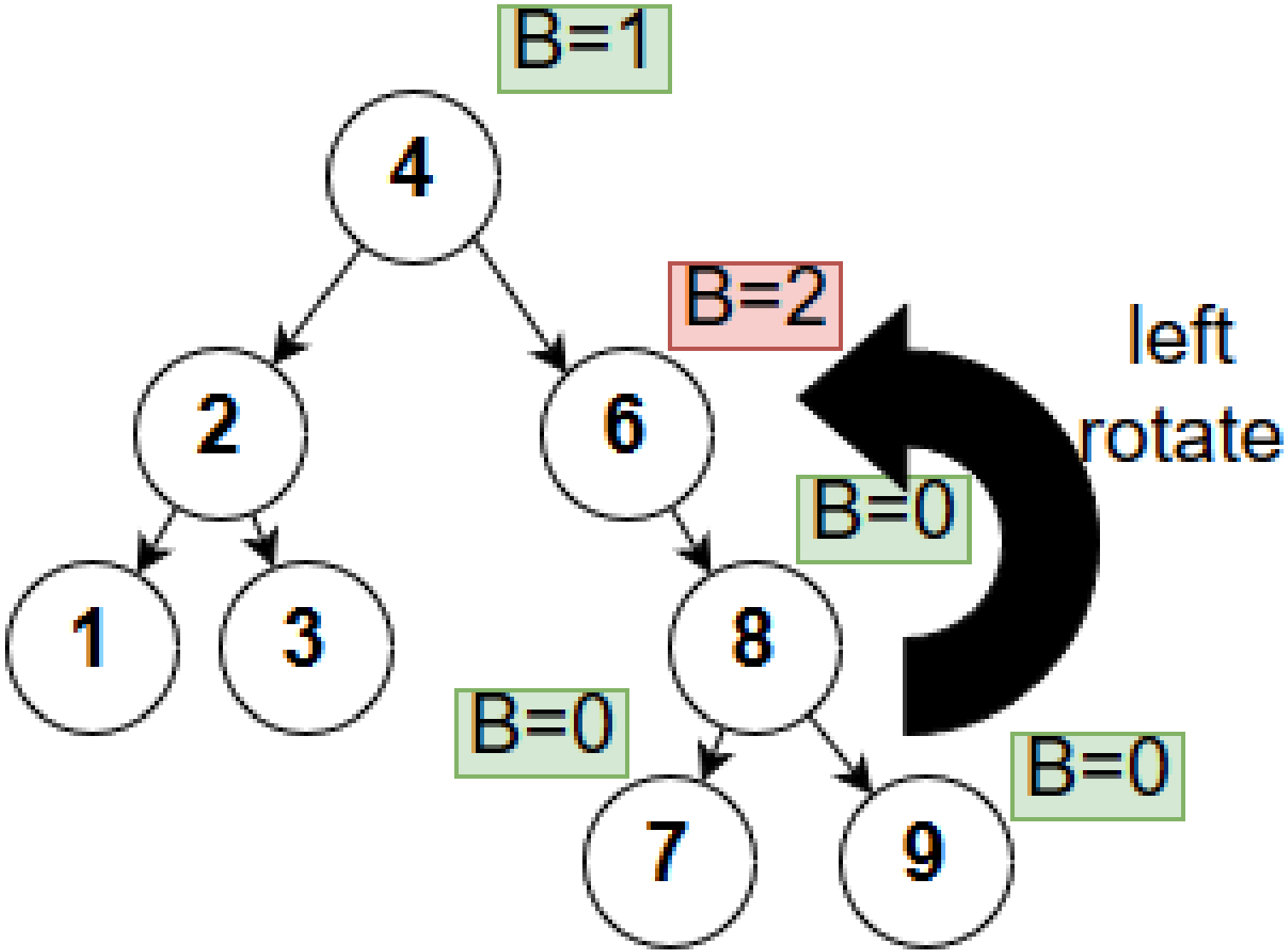
- Deleting a node from AVL tree begins with simple BST delete followed by restoring balance

```
1  AVLDelete(node){
2    newRoot = BSTDelete(node)
3    return if newRoot == NULL //Tree is empty
4    balance = GetBalance(newRoot)
5    if (balance > 1 && getBalance(newRoot.left) >= 0){
6      return RightRotate(newRoot);
7    }
8    if (balance > 1 && getBalance(newRoot.left) < 0){
9      return LeftRightRotate(newRoot);
10   }
11   if (balance < -1 && getBalance(newRoot.right) <= 0){
12     return LeftRotate(newRoot);
13   }
14   if (balance < -1 && getBalance(newRoot.right) > 0){
15     return RightLeftRotate(newRoot);
16   }
17   return newRoot
18 }
```

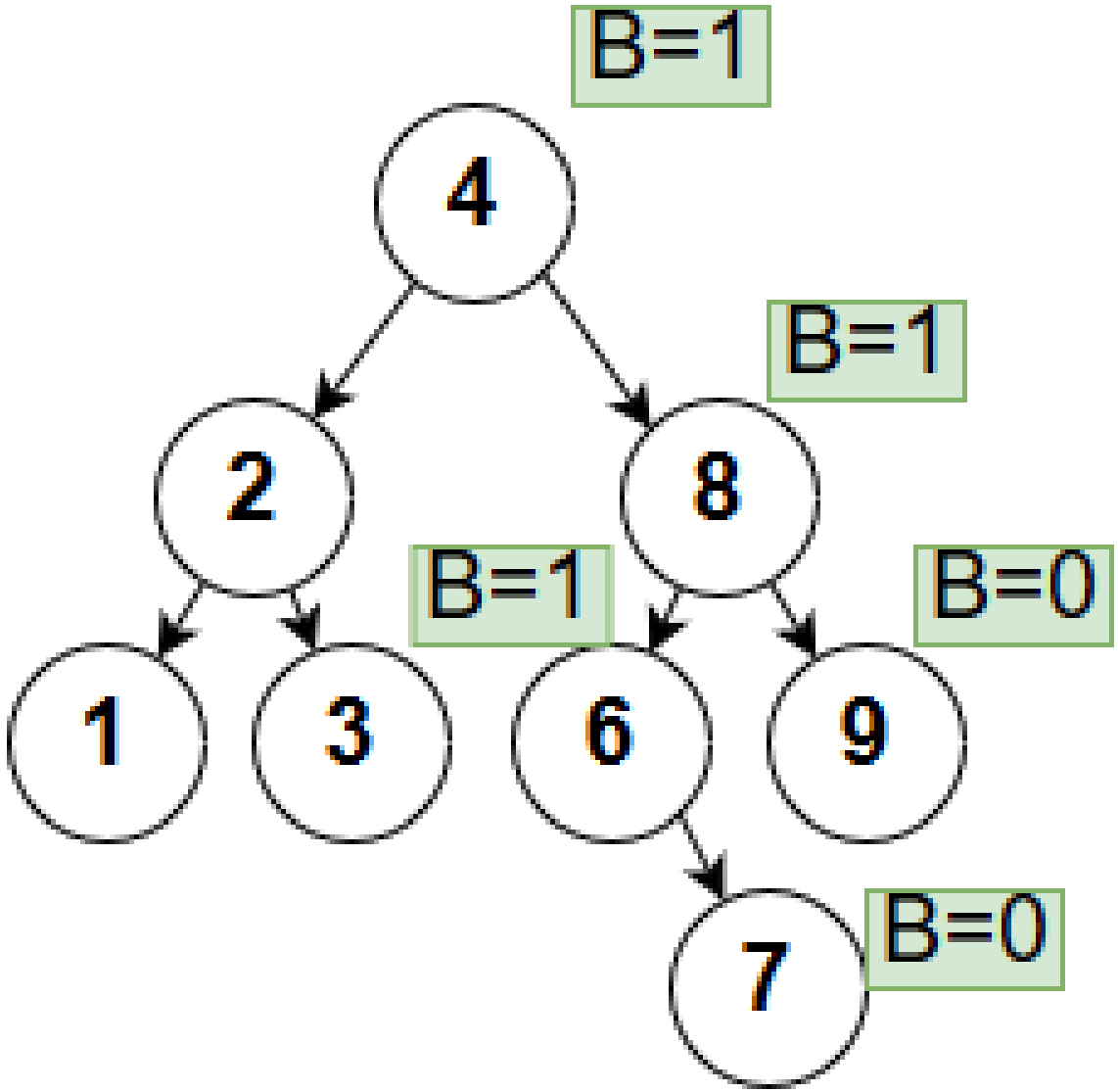
AVL Delete example 1



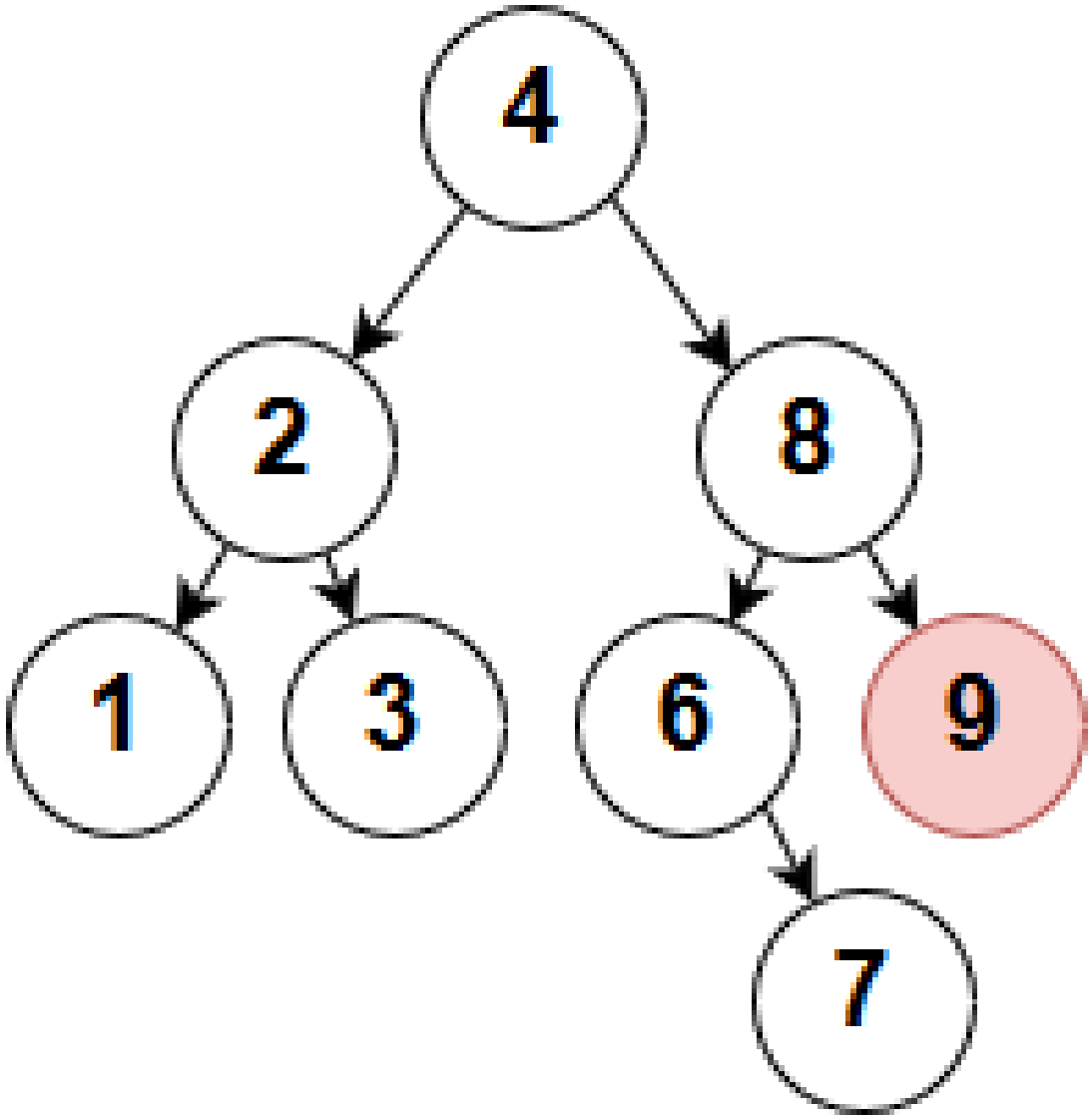
AVL Delete example 1



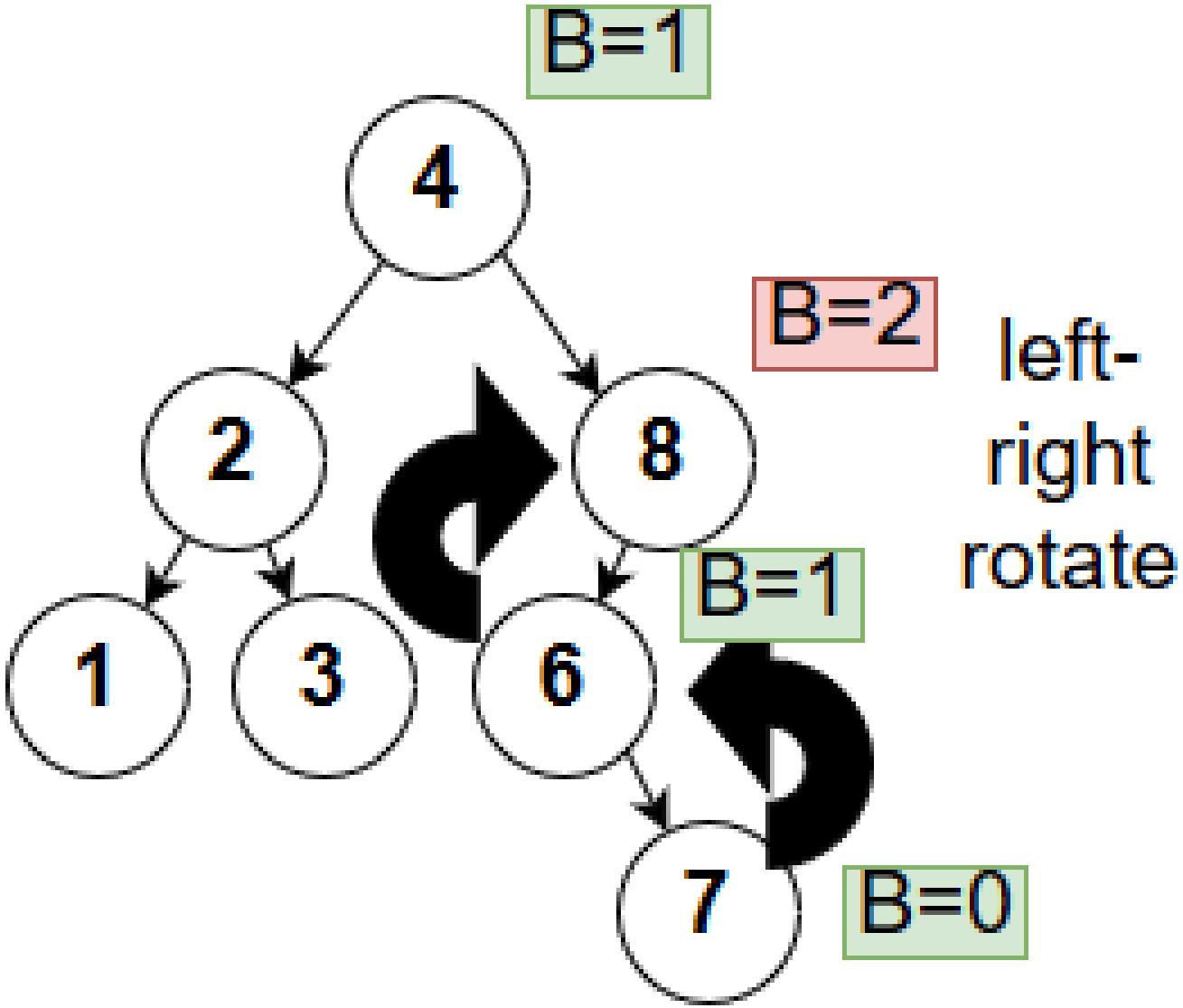
AVL Delete example 1



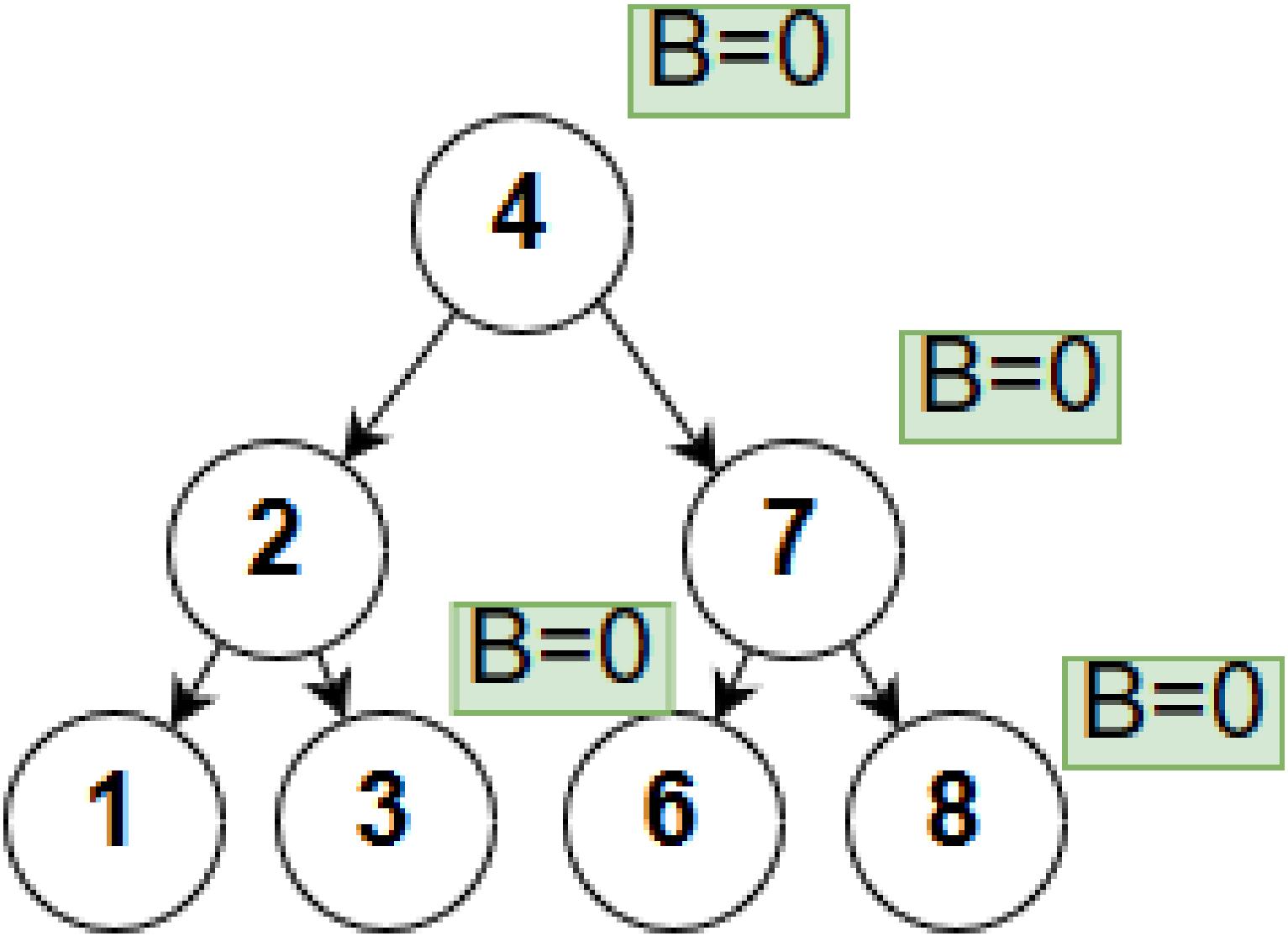
AVL Delete example 2



AVL Delete example 2



AVL Delete example 2



AVL analysis

- AVL trees are self-balancing, during each insert and delete operations some additional calculations are required
- Thankfully those additional operations always take constant time!
- In the end then both operations still take $O(lg\ n)$ time, just as with standard BST (but with worse coefficient)
- Big advantage of AVL tree is also $O(lg\ n)$ search complexity in the worst case, compared to $O(n)$ for binary trees
- Instead of calculating height and balance factor every time it can be stored in a node and updated during every modification operation. We sacrifice some memory for better computation time.

Course plan

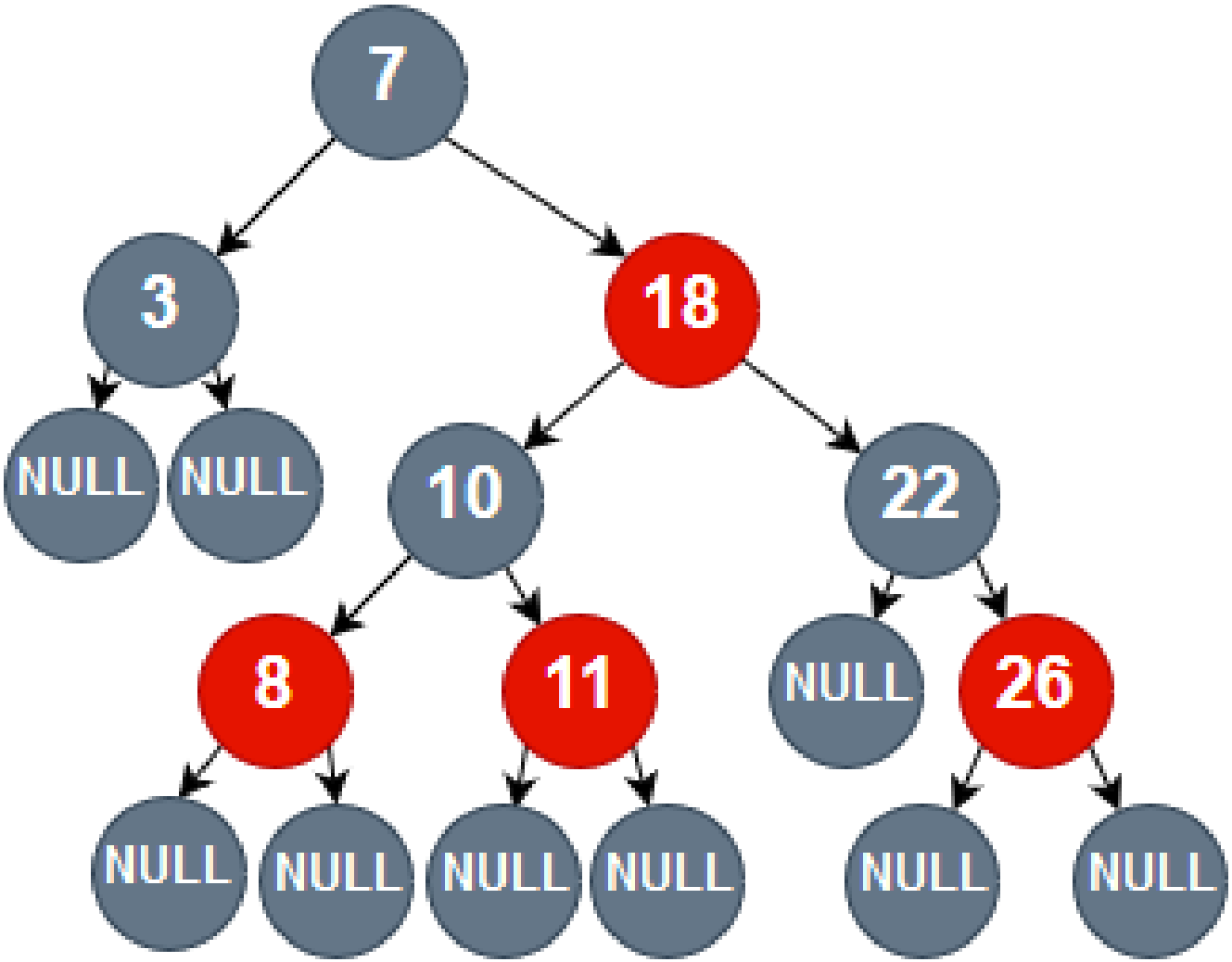
1. Binary Search Trees
   - Implementation
2. AVL Trees
3. Red-Black Trees
4. Other types of search trees

- Even though complexity of modification and query operations in AVL trees are $O(l\,n)$, they still require a lot of computations for inserts and deletes
- We can increase the effectiveness of those operations by trading some memory and balance
- Red-Black trees do exactly that
- In Red-Black trees we add one bit to every node that will store color of the node
- In RB trees are balanced in such a way, that every path from node to its leafs contain equal amount of black nodes - black height
- All AVL trees are RB trees, but not every RB tree is AVL tree!

Red-black tree properties

- Each node is either red or black.
- The root is black.
- All leaves are NULL and are black (we will skip it in visualizations).
- If a node is red, then both its children are black.
- Every path from a given node to any of its descendant leaf nodes contains the same number of black nodes.

Red-Black tree example



Balancing Red-Black tree

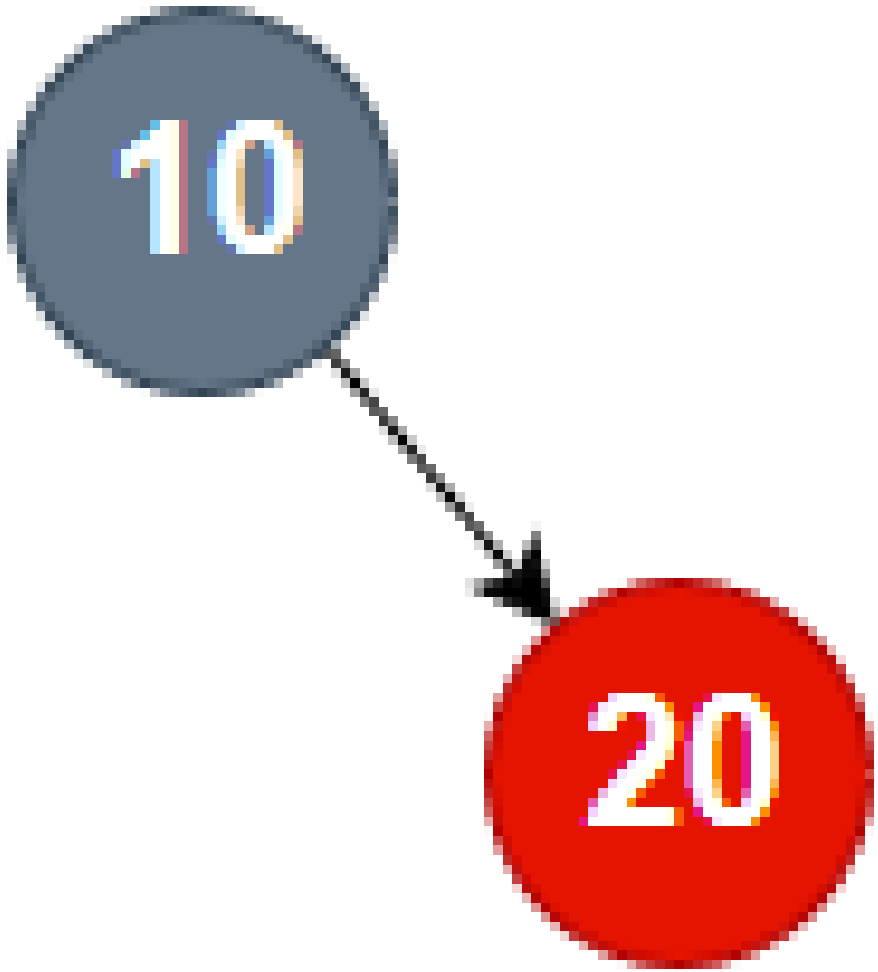
- Two operations are used to maintain balance in Red-Black trees: recoloring and rotation
- After every insertion we first try restoring balance by recoloring, if that doesn't help we do rotation

Insertion procedure

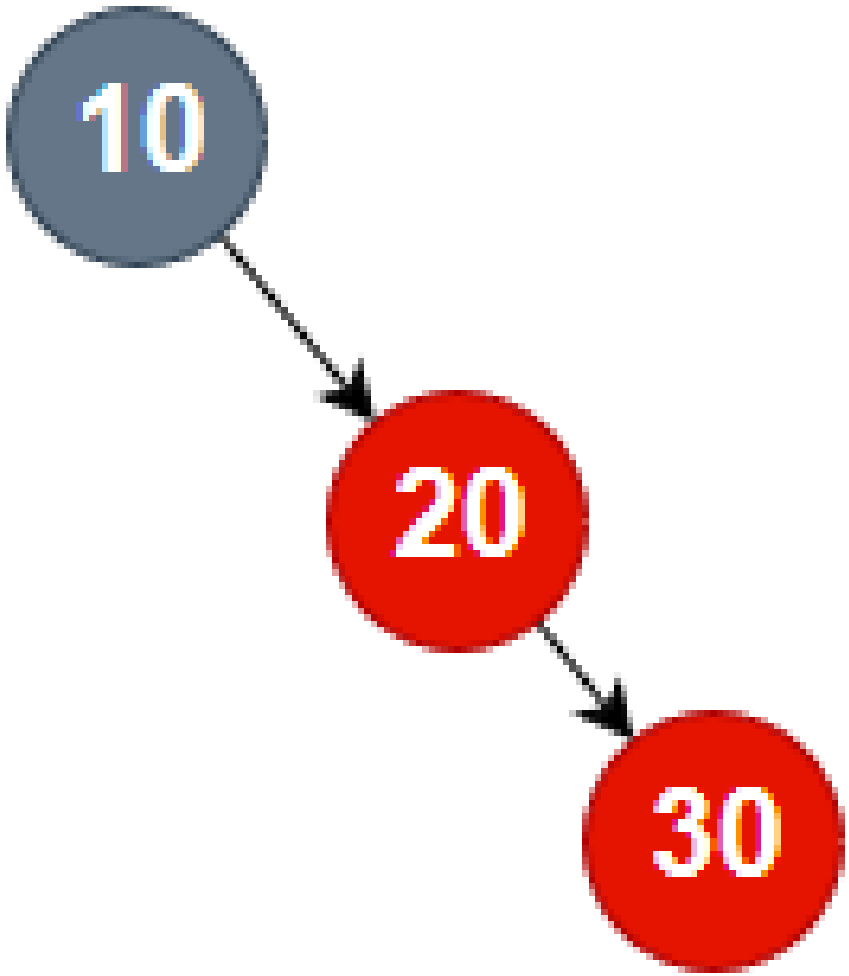
1. Start with standard BST insertion and color new node Red.
2. If new node is root, change it's color to Black.
3. End, unless new node parent color is Red or new node is not a root
4. If new node's uncle (sibling of parent) is Red:
   1. Change color of parent and uncle to Black.
   2. Change color of grandparent to Red.
   3. Repeat steps 2 and 3 for grandparent of new node.
5. Else, if new node's uncle is Black perform one of four (similar to AVL trees):
   - Right/Left rotate and swap colors of parent and grandparent of new node
   - Right-left/Left-right rotate and swap colors as above

Red-Black tree insert example

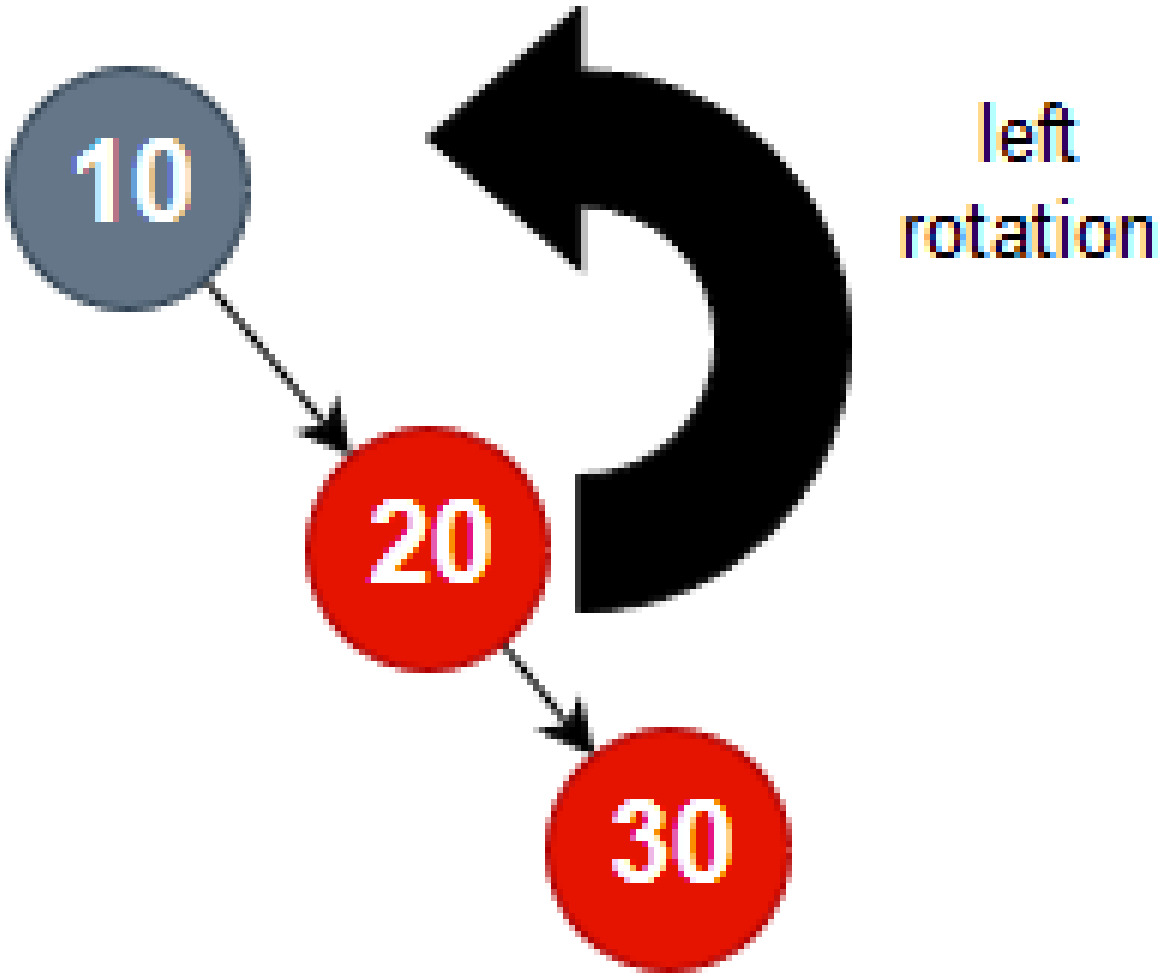
Red-Black tree insert example



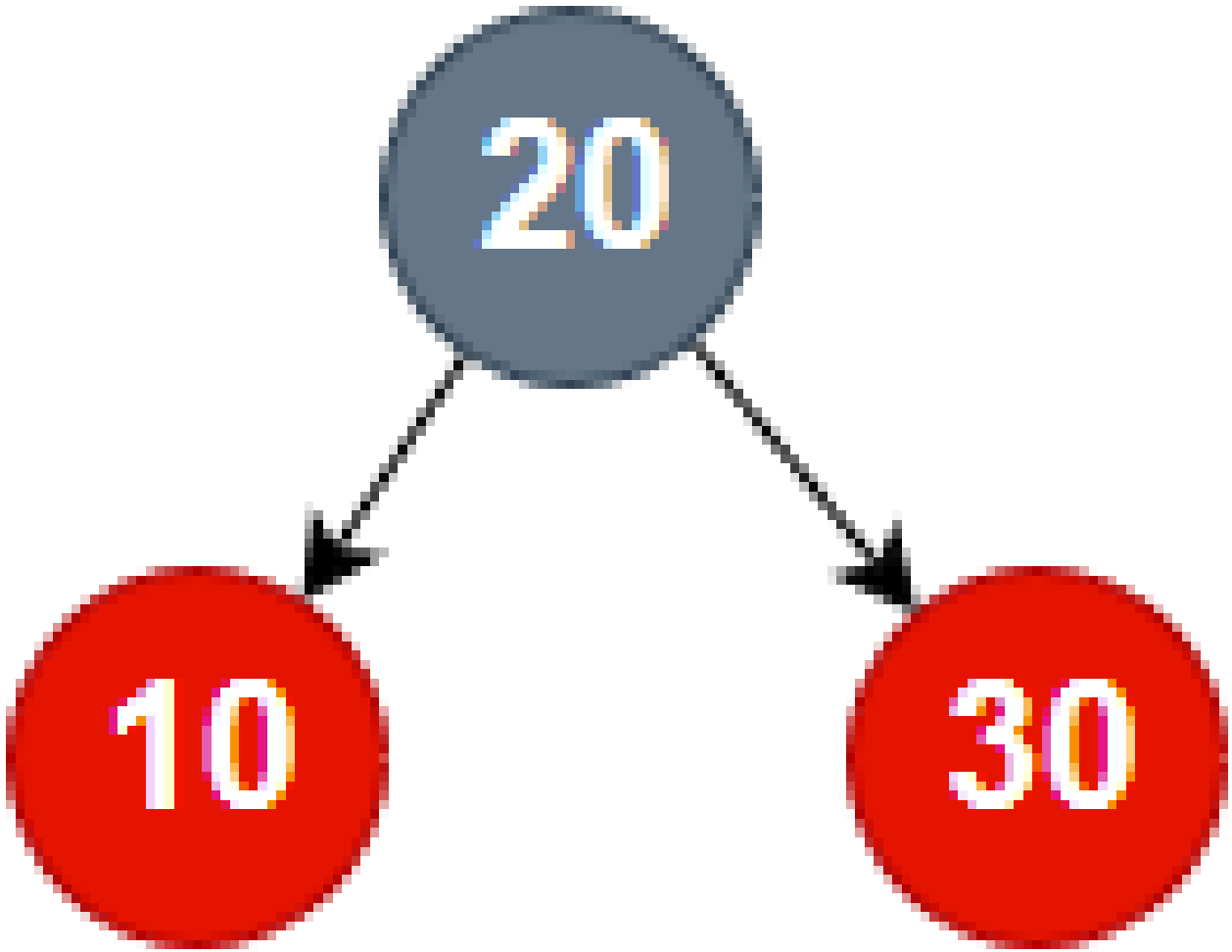
Red-Black tree insert example



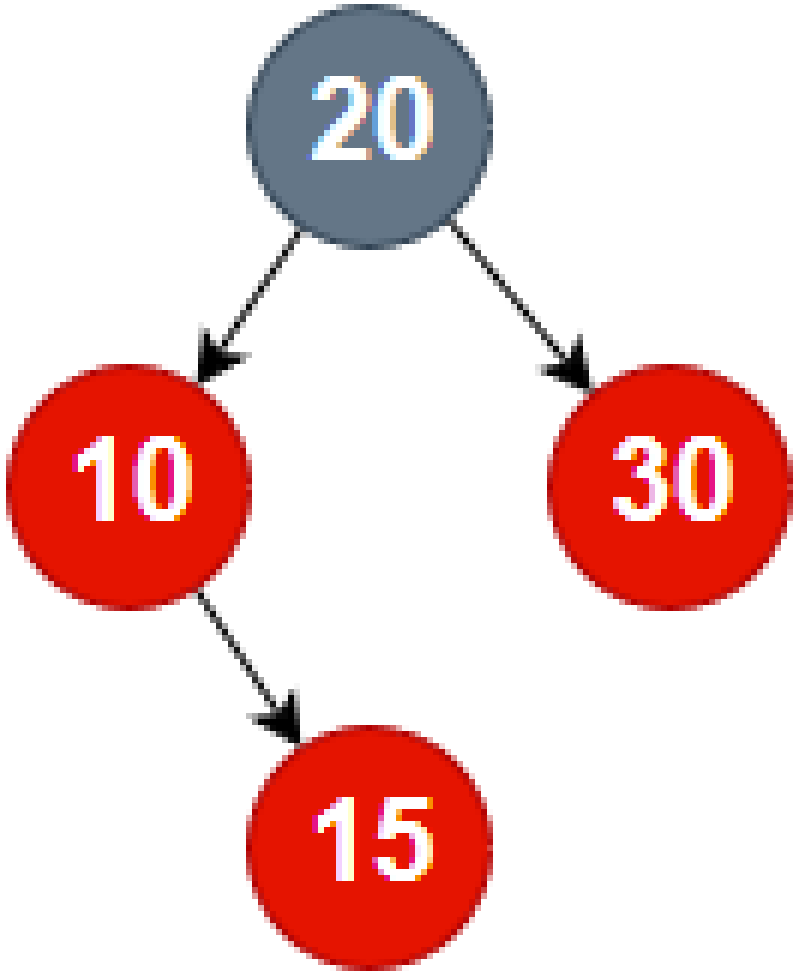
Red-Black tree insert example



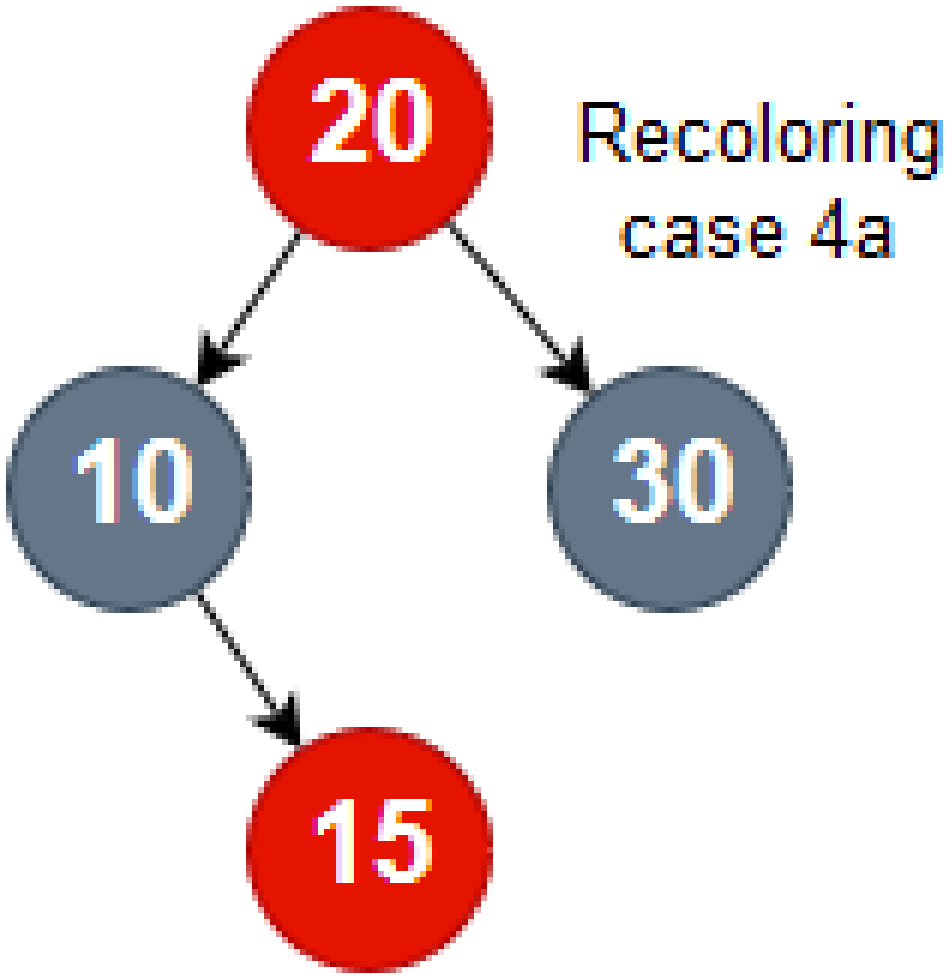
Red-Black tree insert example



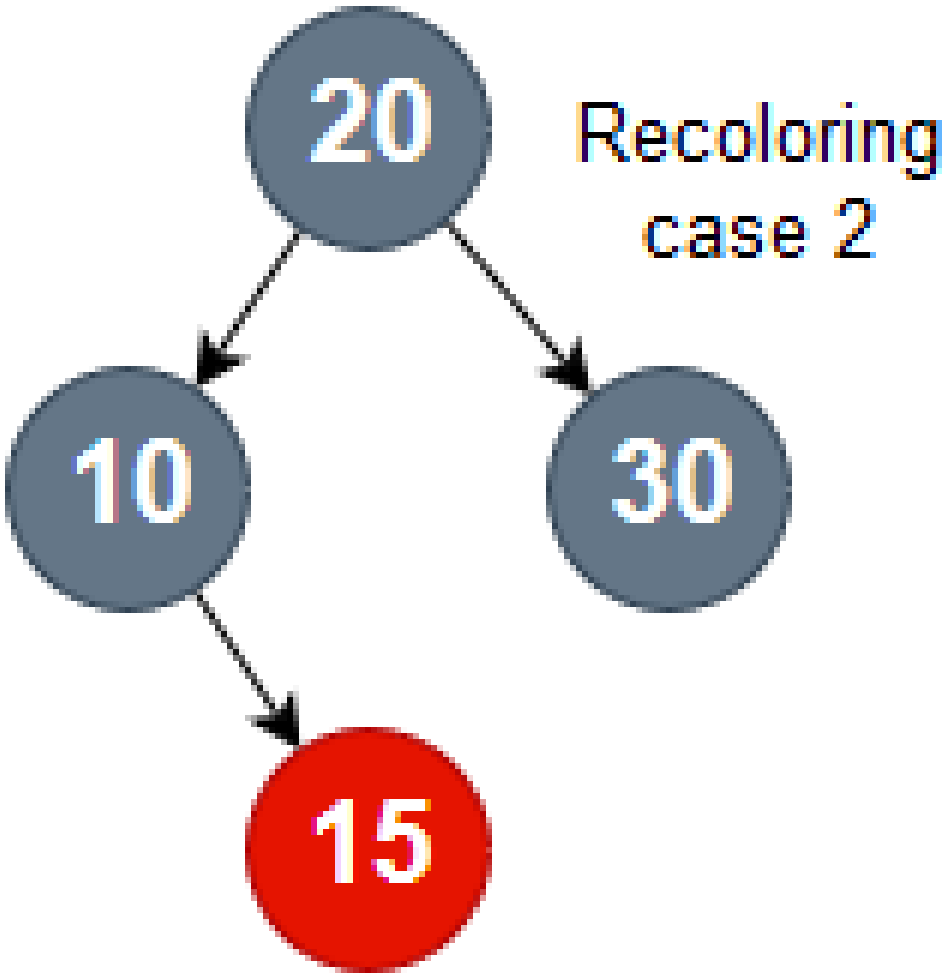
Red-Black tree insert example



Red-Black tree insert example



Red-Black tree insert example

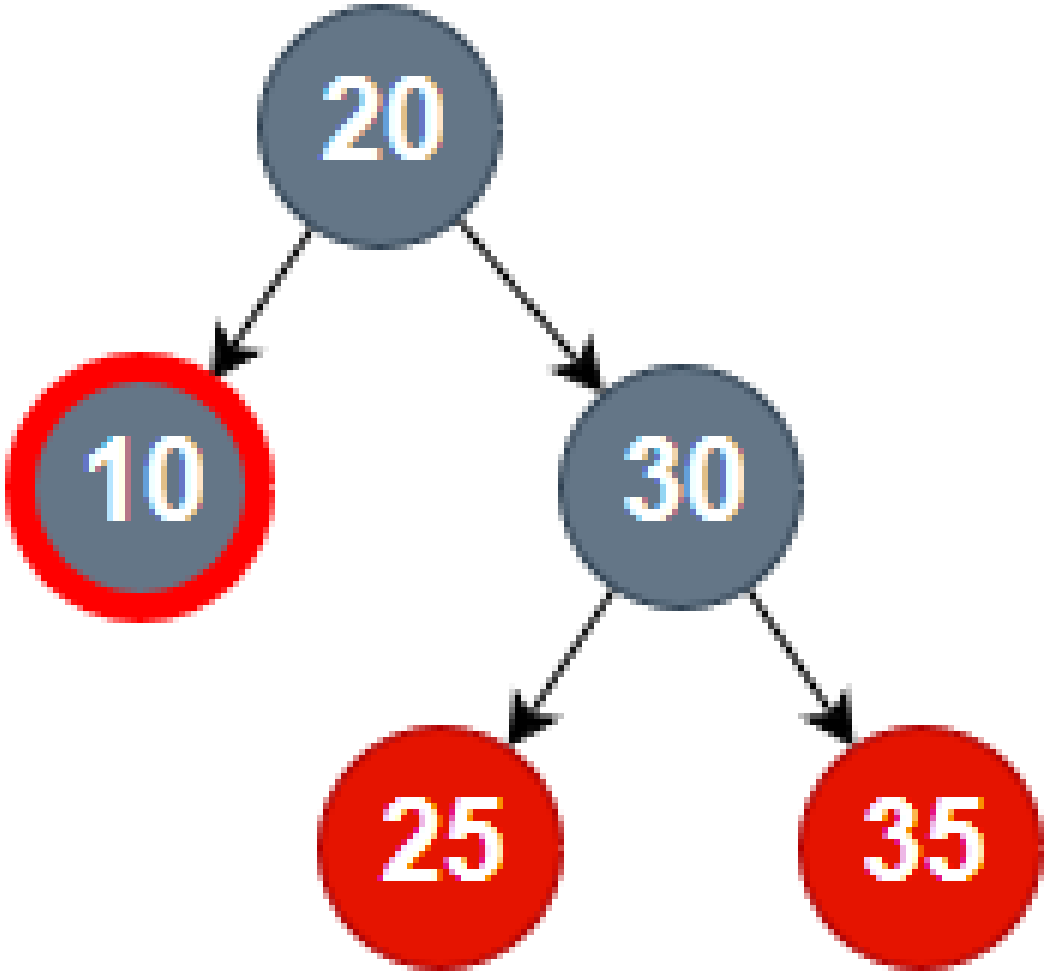


Red-Black tree deletion

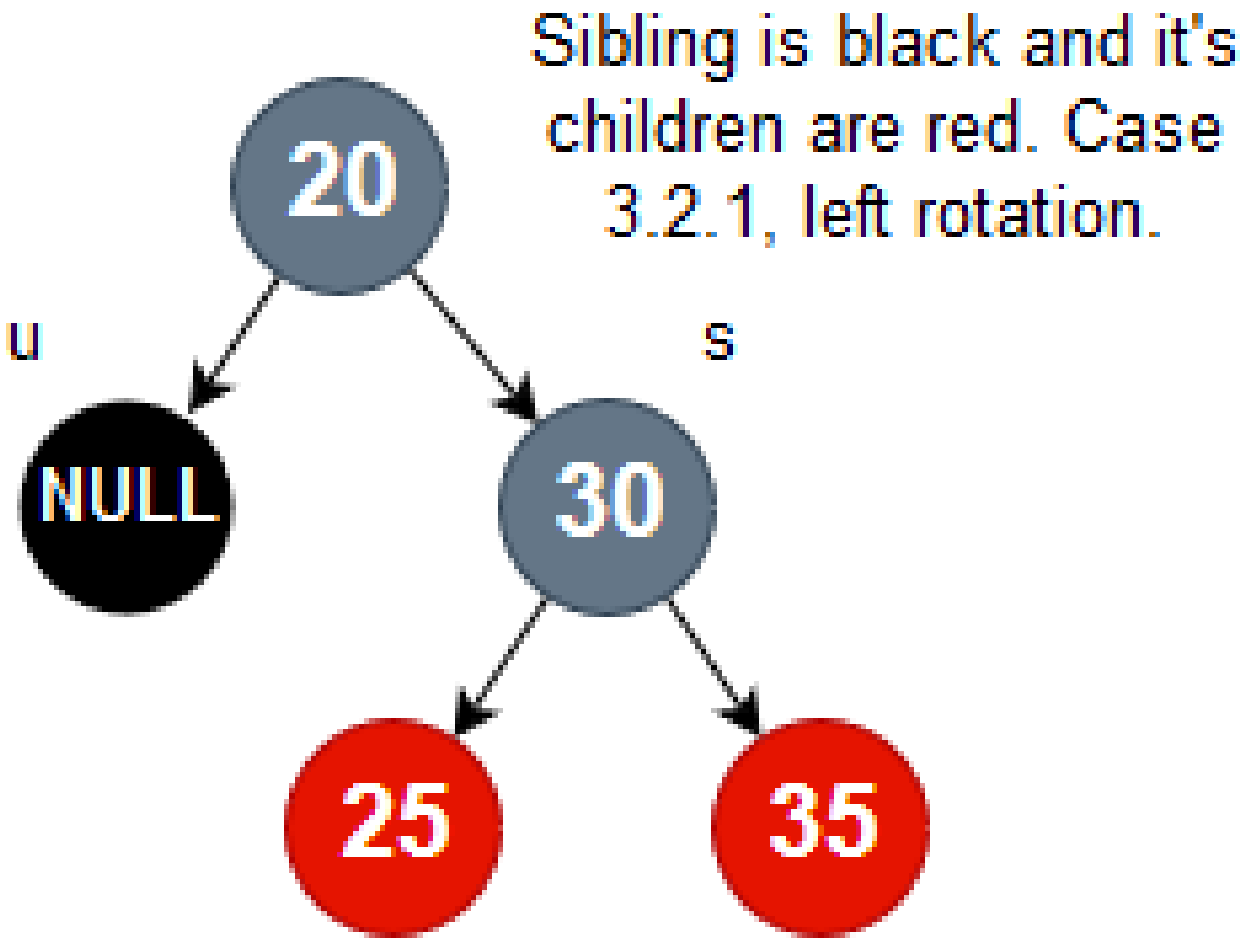
- Deletion from Red-Black tree is more complex process than insertion
- We use notion of double black, when a black node is replaced by black successor in deletion process

1. Perform standard BST deletion. Let v be the deleted node and u it's successor that replaced it in BST deletion
2. If either v or u are red mark u black
3. Else if both v and u are black:
   1. Change color of u to double black.
   2. Perform the following while u is double black and is not a root. Let s be a sibling of u:
      1. If s is black and at least one of s's children are red perform rotations (left, right, left-right, right-left)
      2. If s and it's children are black, perform recoloring. Repeat for parent if parent is black
      3. If s is red perform left or right rotation and recolor old sibling and parent
   3. If u is root color it single black

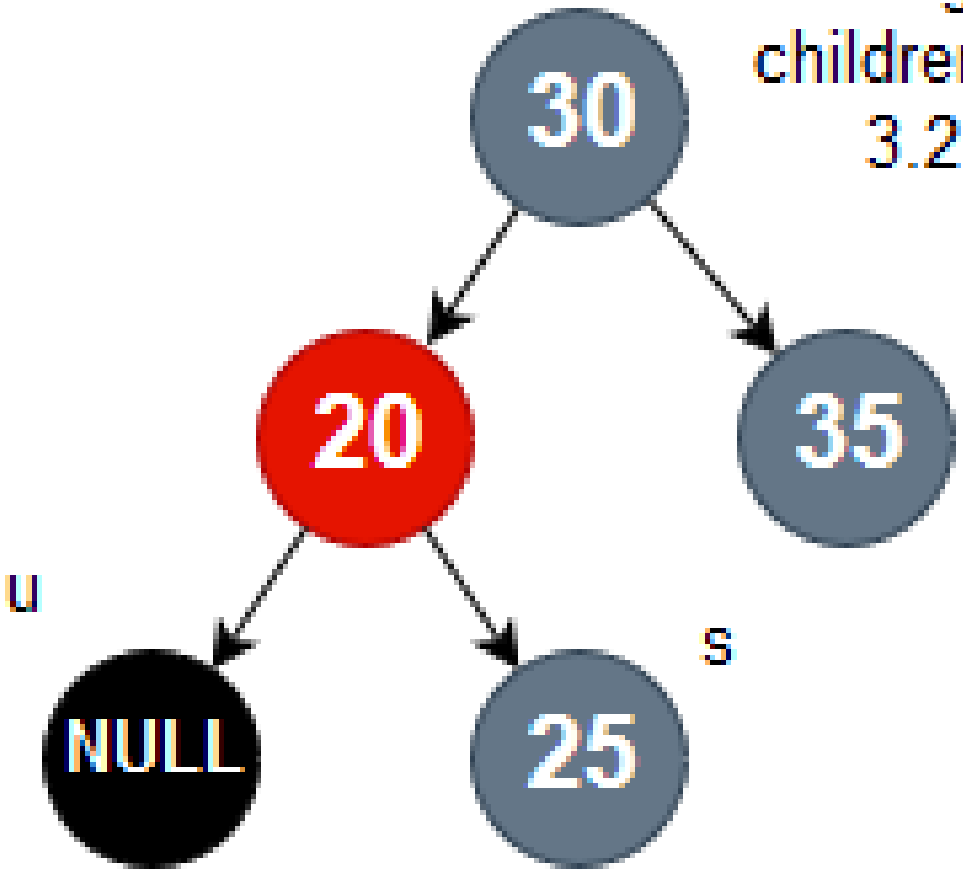
Red-Black tree delete example



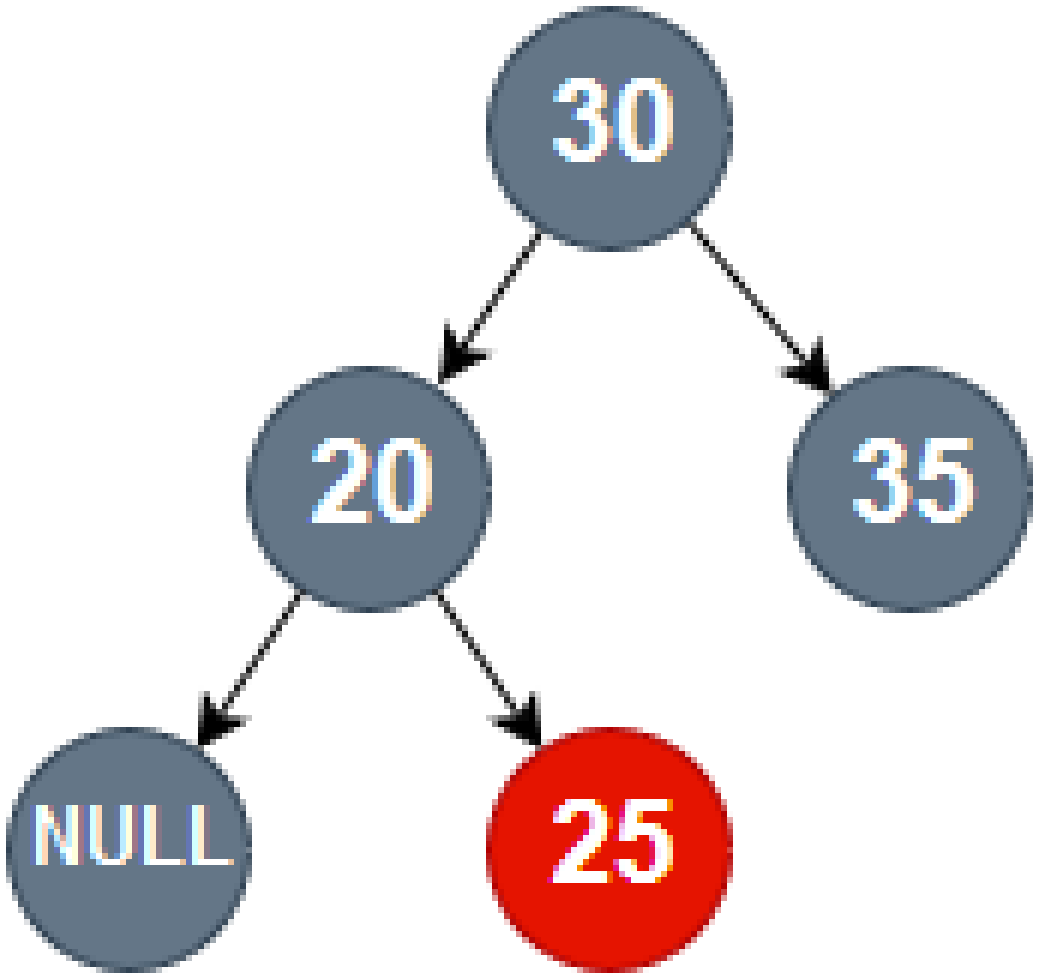
Red-Black tree delete example



Red-Black tree delete example



Red-Black tree delete example



Red-Black trees analysis

- Red-Black tree complexity of query and modifying operations is $O(lg\ n)$.
- This type of tree requires additional bit of memory for every node, but this is such a small amount that might be omitted in analysis.
- In comparison to AVL trees, Red-Black trees are slightly less balanced, but insertion and deletion are a little bit faster.
- Which tree to choose?
  - AVL tree if you will be querying tree a lot
  - Red-Black tree if you will be modifying tree a lot

Course plan

1. Binary Search Trees
   - Implementation
2. AVL Trees
3. Red-Black Trees
4. Other types of search trees

Other types of binary search trees

- **Splay trees** - if a node is accessed via search operations it is moved one level closer to the root. In splay trees commonly searched elements are near the top, while those that are accessed rarely are pushed to the bottom
- **Treaps** (tree+heap) - every node has key and randomly assigned priority. Keys follow BST property, while priorities follow heap property. As node's priority is completely random, the structure of a Treap is also random, but it can be shown that with very high probability it's height will be proportional to $lg\ n$, making all operations also $O(lg\ n)$.
- **Left-leaning red-black tree** - similar to standard red-black tree, but easier to implement
- **Dancing tree** - instead of balancing tree after every insert/delete, this tree only balances itself when it's written to the disk. It was mainly used in Raiser4 filesystem.
- **Scapegoat tree** - instead of small incremental rebalancing operations from time to time big overhaul is performed.