AVL Trees

Hussein Suleman <hussein@cs.uct.ac.za>

Department of Computer Science
School of IT
University of Cape Town

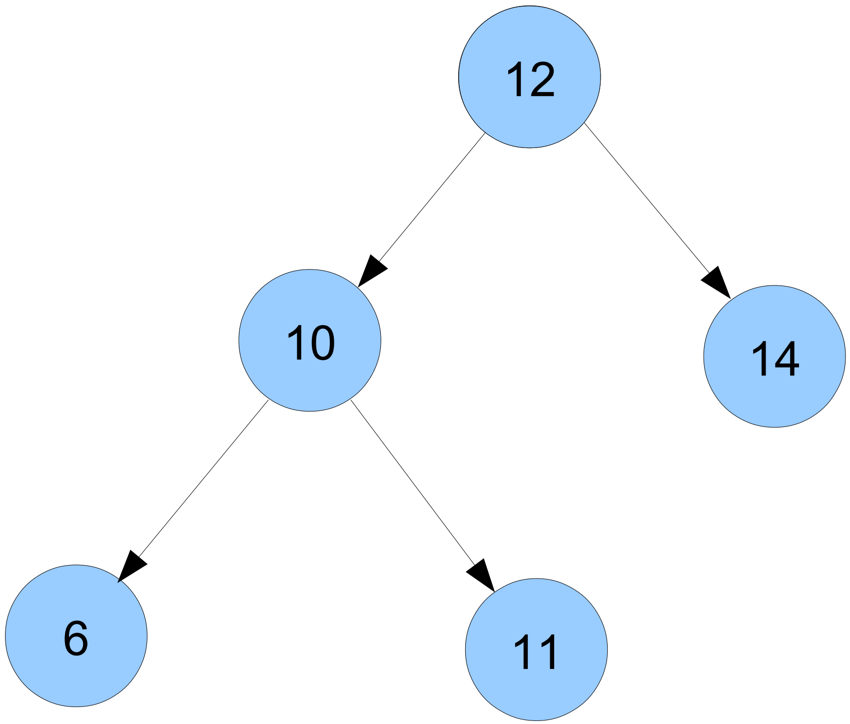
2020

AVL Trees

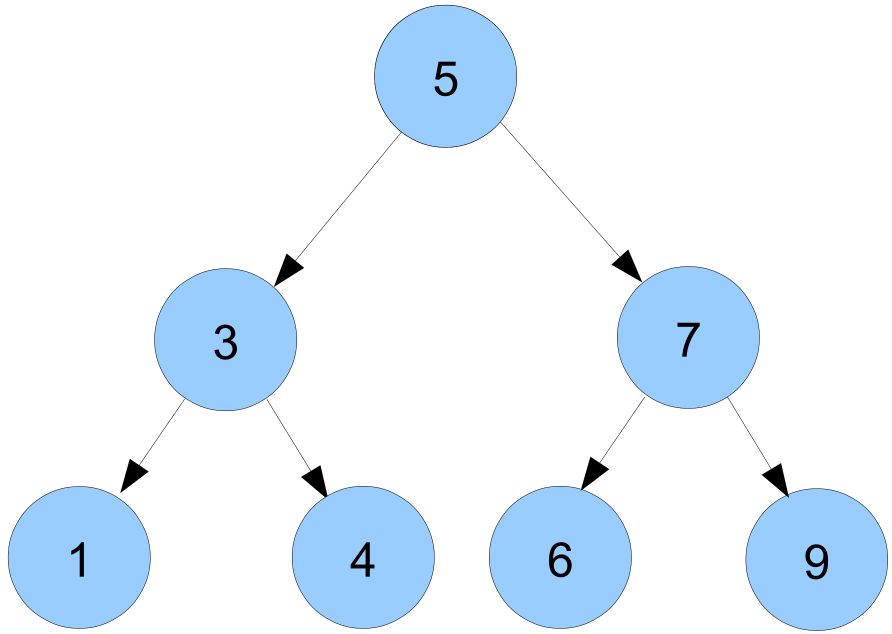
- Binary Search Tree with additional balance properties:
  - Every node is balanced (in AVL terms).
    - Height of left and right sub-trees are at most different by 1.
- Advantage: Balanced tree in worst case; Faster operations.
- Disadvantage: More work for adding/removing nodes.

Example AVL Tree

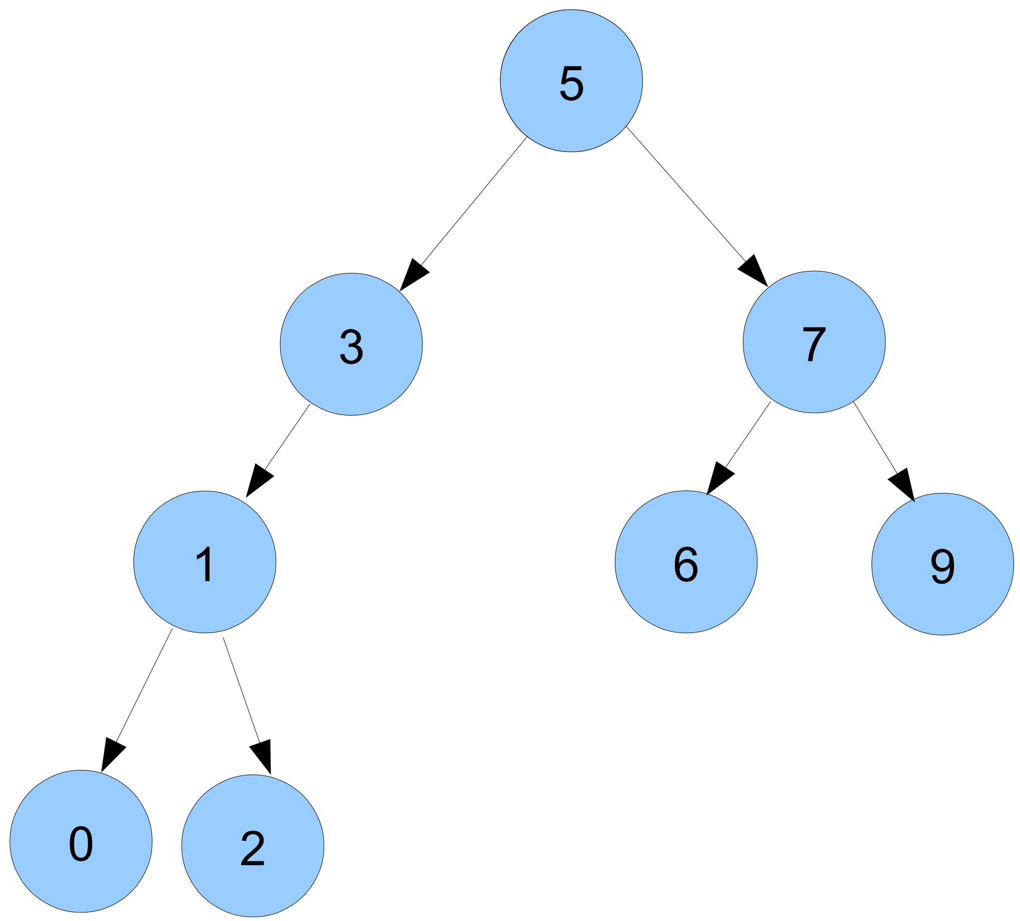
- 12: lhs = 1, rhs = 0
- 10: lhs = 0, rhs = 0
- 14: lhs = -1, rhs = -1
- 6: lhs = -1, rhs = -1
- 11: lhs = -1, rhs = -1



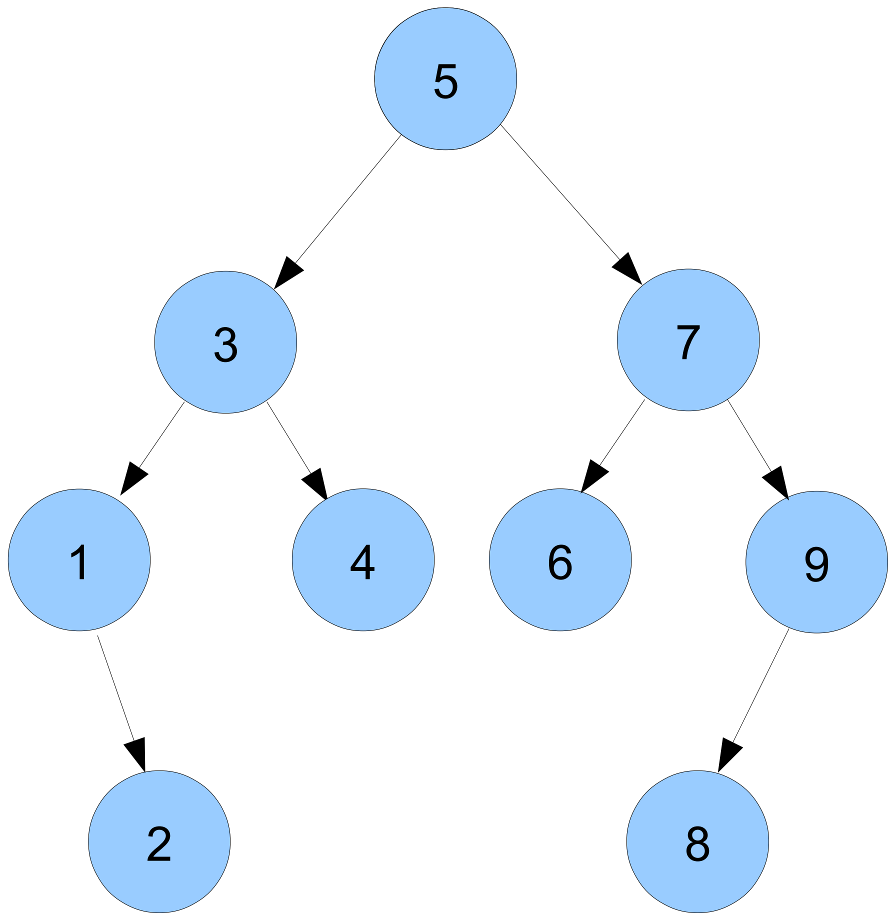
Am I AVL or Not?



Am I AVL or Not?



Am I AVL or Not?



Basic Operations

- insert
- delete

- auxiliary routines:
  - manage height
    - precalculate it
    - fix it after changes
  - rebalance
  - rotate left/right

Additional Information

- Maintain height at each subtree
  - Use an instance variable in the node
- Determine the difference in heights between left and right sub-trees
  - Balance factor

height (precalculated)

```
public int height ( BinaryTreeNode<dataType> node )
{
   if (node != null)
      return node.height;
   return -1;
}
```

balanceFactor and fixHeight

```
public int balanceFactor ( BinaryTreeNode<dataType> node )
{
   return height (node.right) - height (node.left);
}

public void fixHeight ( BinaryTreeNode<dataType> node )
{
   node.height = Math.max (height (node.left),
                           height (node.right)) + 1;
}
```

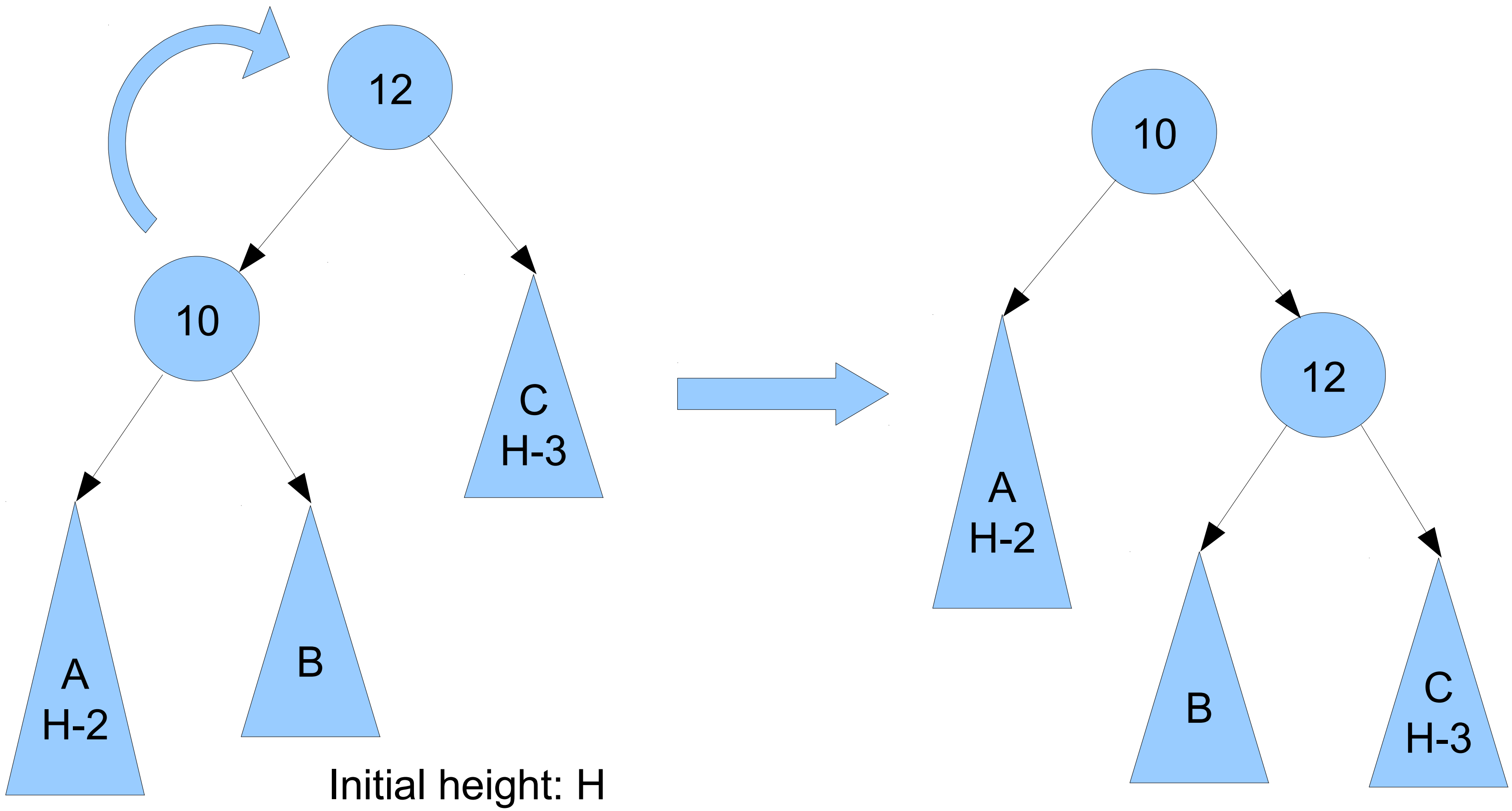
insert Algorithm

- Use same BST insertion algorithm.
- Rebalance all nodes potentially affected.
  - Apply to all nodes from insertion point to root.
  - Use tree rotations at each node as necessary.

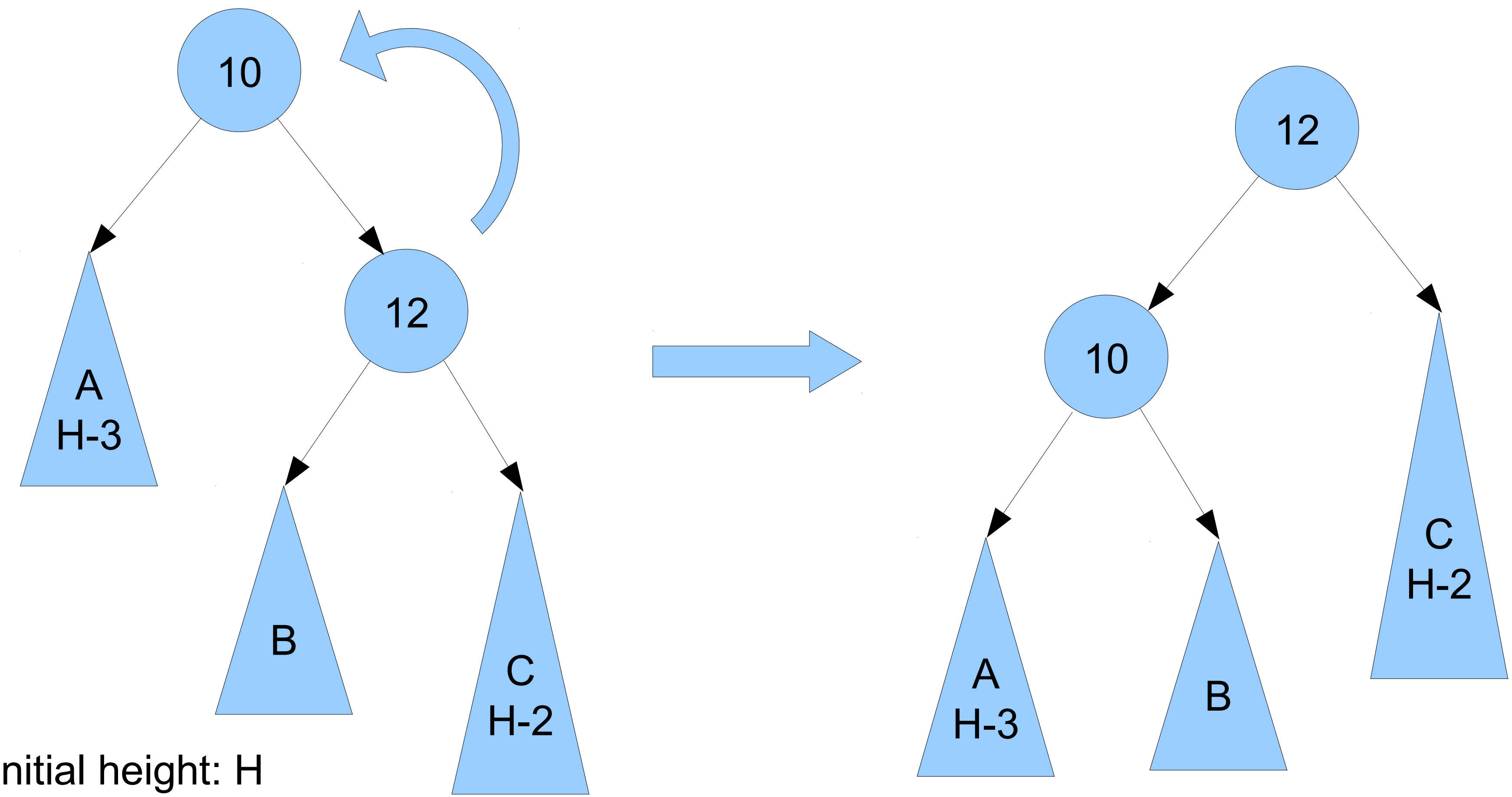
Tree Rotations

- Single rotations:
  - insertion into left subtree of left child (rotateWithLeftChild - rotate right)
  - insertion into right subtree of right child (rotateWithRightChild - rotate left)
- Double rotations:
  - insertion into right subtree of left child
  - insertion into left subtree of right child

Single rotation: rotateWithLeftChild

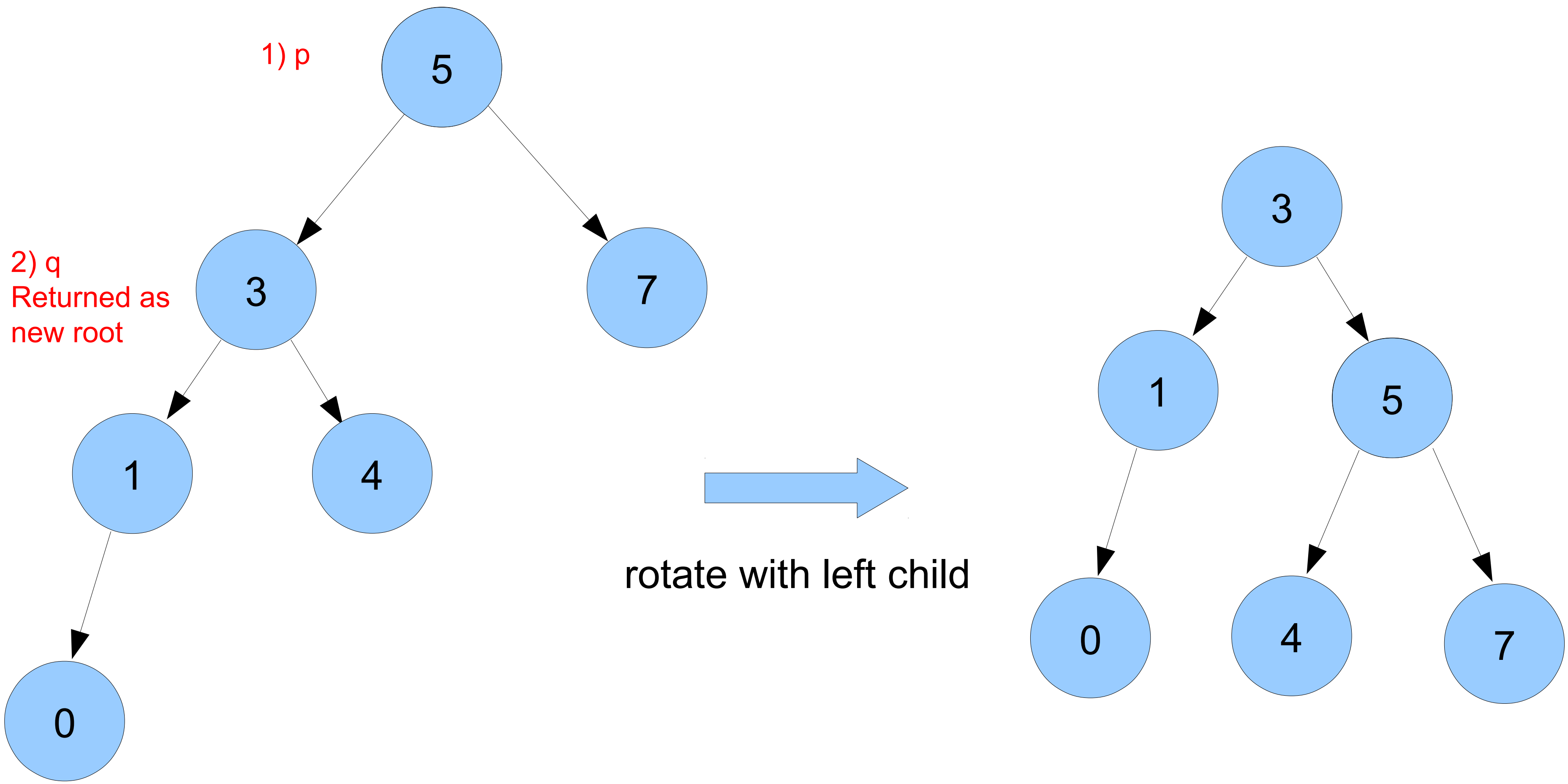


Single rotation: rotateWithRightChild



Initial height: H

Example: Single rotation



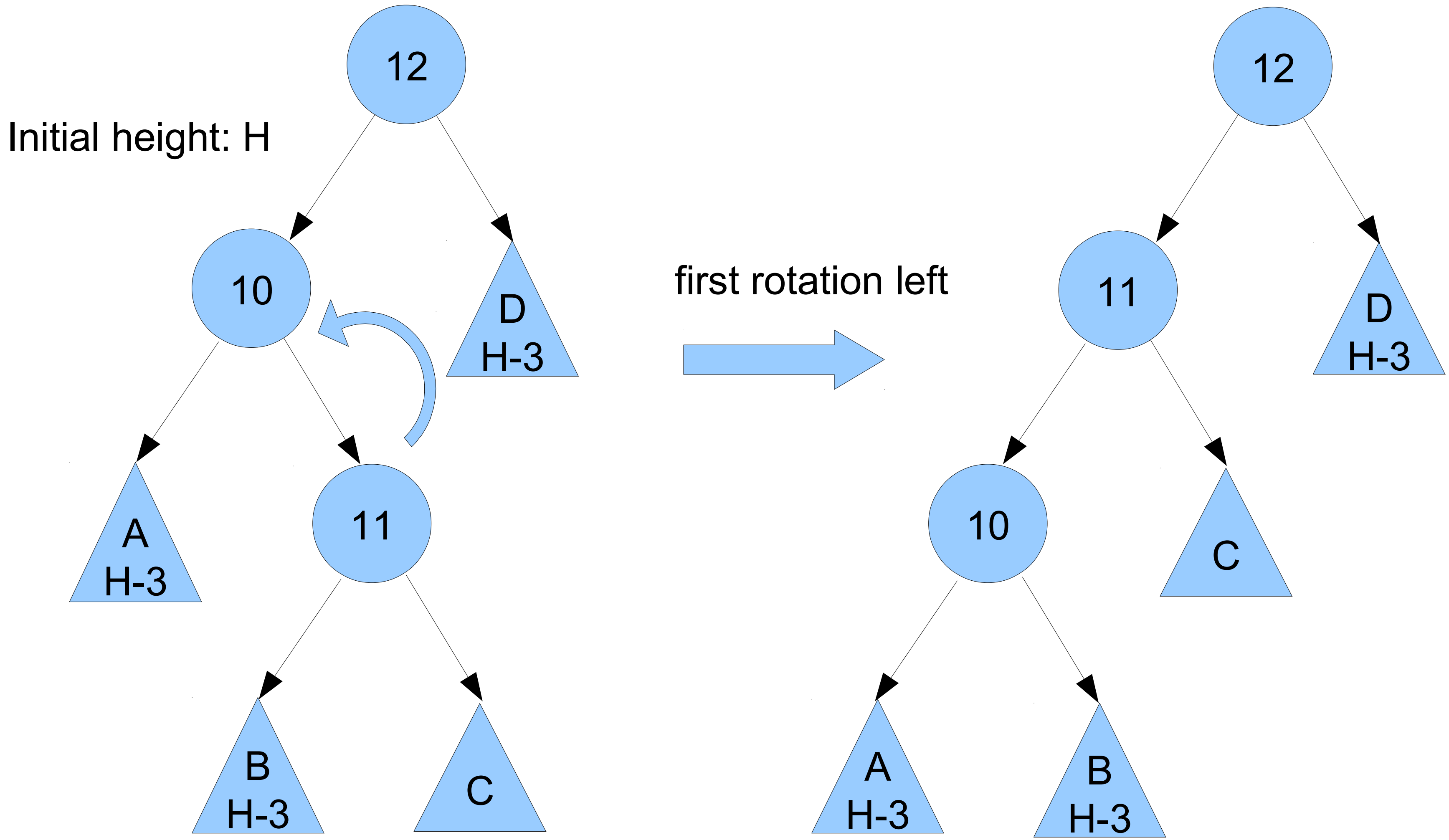
rotateRight

```
   public BinaryTreeNode<dataType> rotateRight
( BinaryTreeNode<dataType> p )
   {
      BinaryTreeNode<dataType> q = p.left;
      p.left = q.right;
      q.right = p;
      fixHeight (p);
      fixHeight (q);
      return q;
   }
```

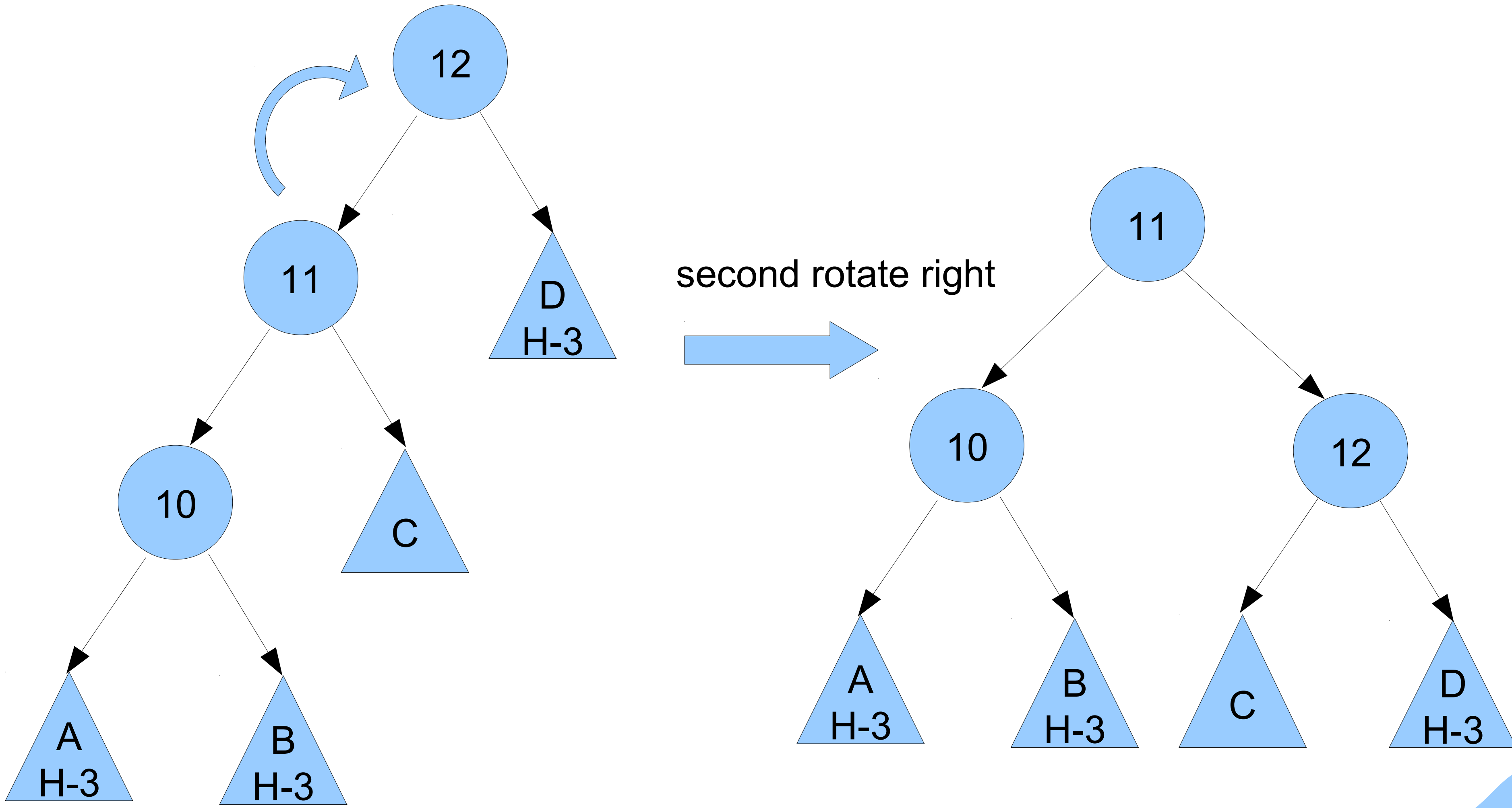
rotateLeft

```
   public BinaryTreeNode<dataType> rotateLeft
( BinaryTreeNode<dataType> q )
   {
      BinaryTreeNode<dataType> p = q.right;
      q.right = p.left;
      p.left = q;
      fixHeight (q);
      fixHeight (p);
      return p;
   }
```

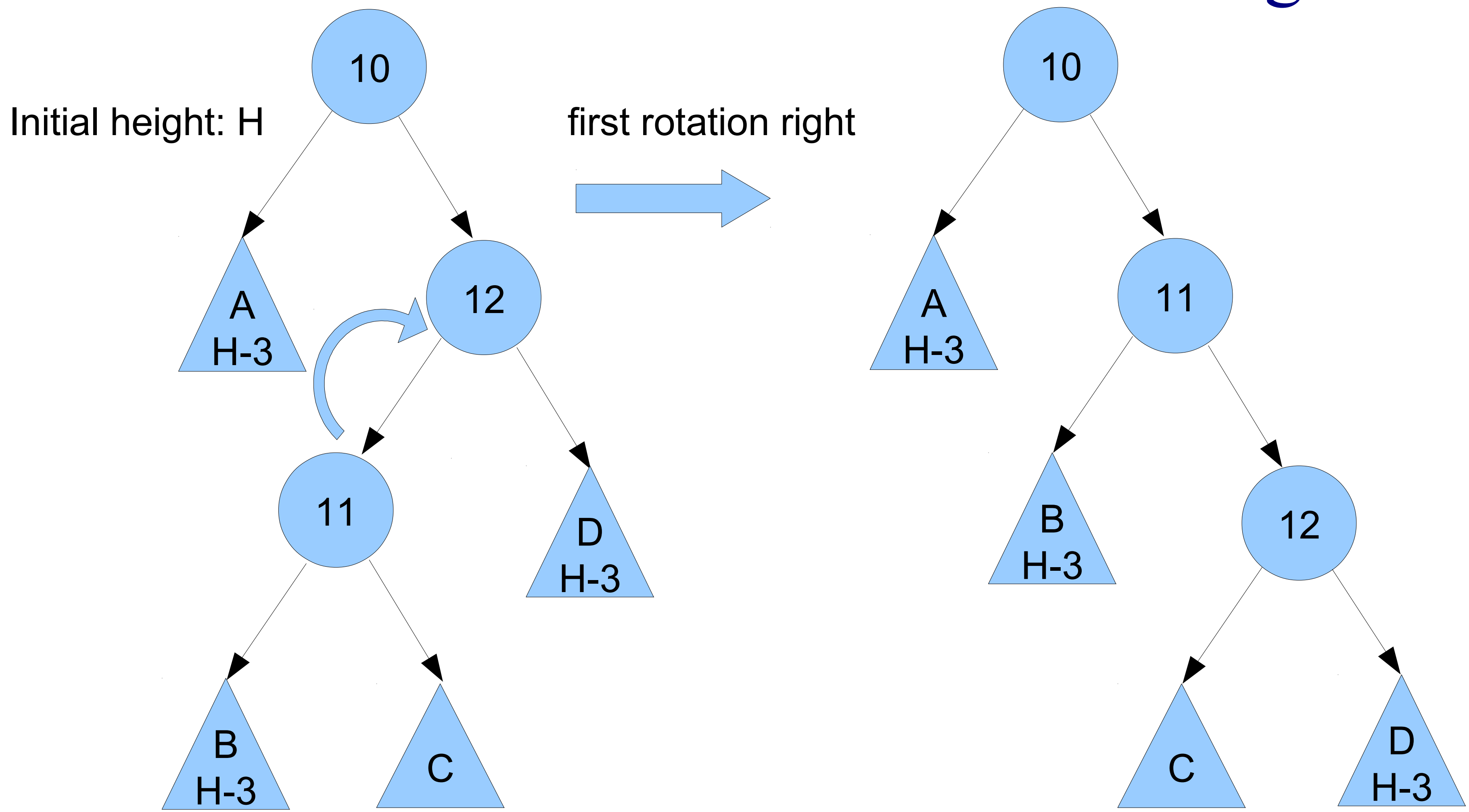
Double rotation: doubleRotateWithLeftChild 1



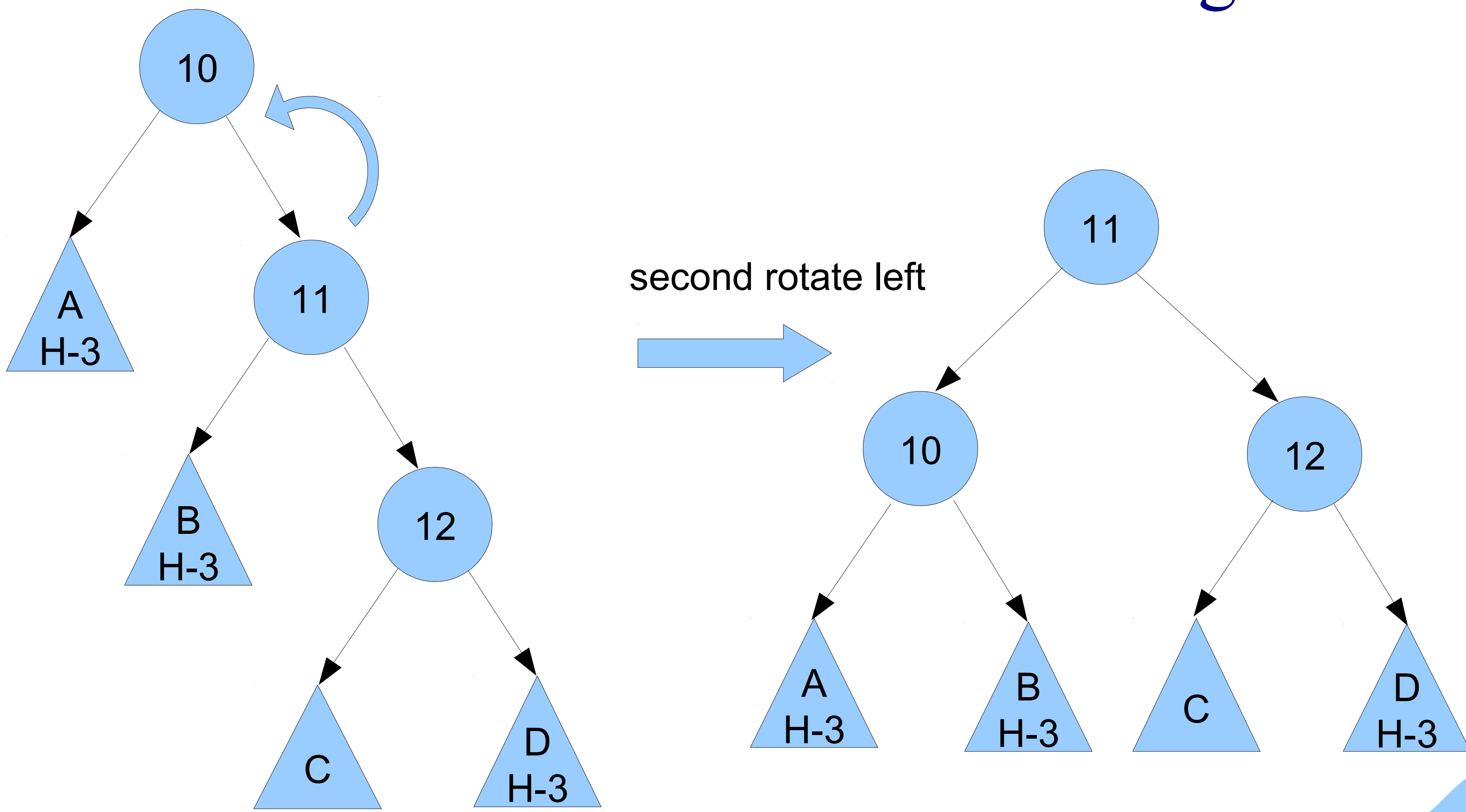
Double rotation: doubleRotateWithLeftChild 2



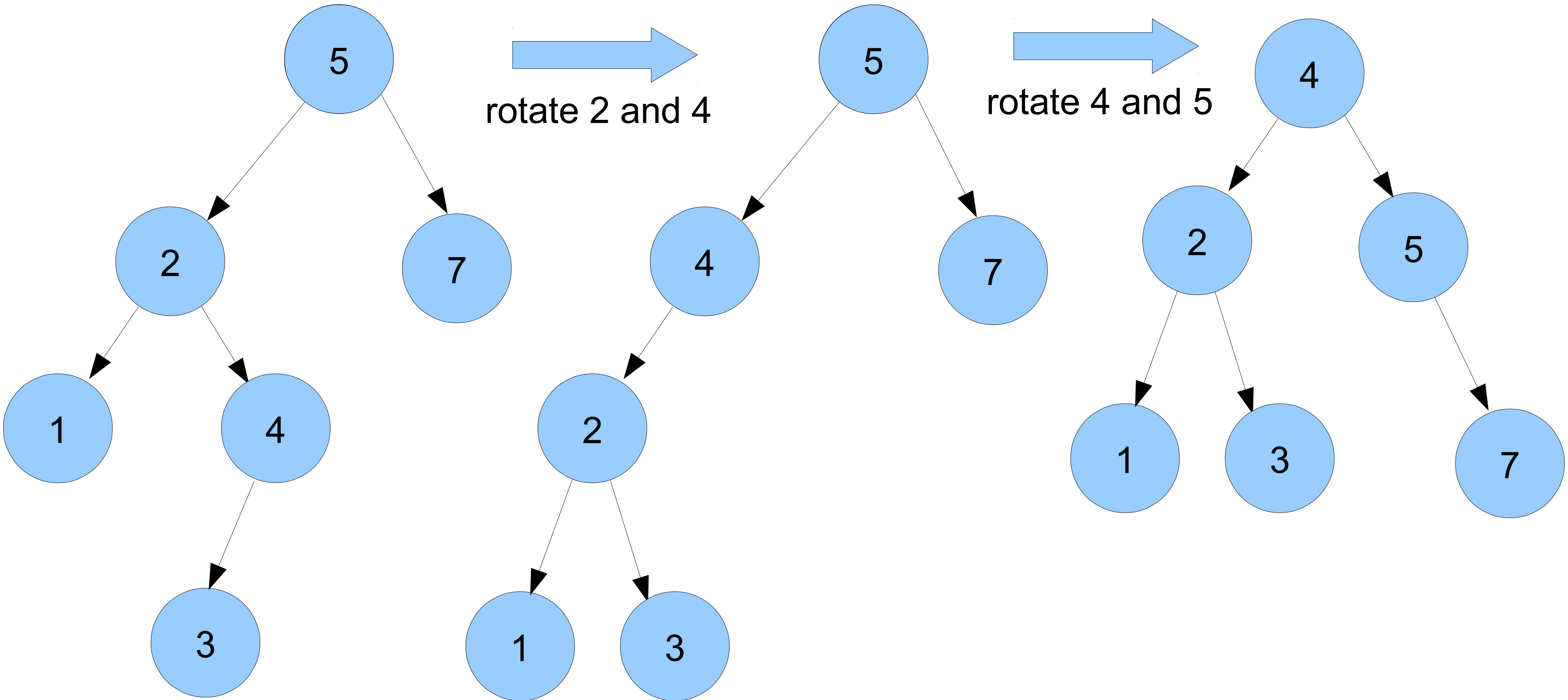
Double rotation: doubleRotateWithRightChild 1



Double rotation: doubleRotateWithRightChild 2



Example: Double rotation



balance

```
public BinaryTreeNode<dataType> balance ( BinaryTreeNode<dataType> p )
{
   fixHeight (p);
   if (balanceFactor (p) == 2)
   {
      if (balanceFactor (p.right) < 0)
         p.right = rotateRight (p.right);
      return rotateLeft (p);
   }
   if (balanceFactor (p) == -2)
   {
      if (balanceFactor (p.left) > 0)
         p.left = rotateLeft (p.left);
      return rotateRight (p);
   }
   return p;
}
```

insert

```
public void insert ( dataType d )
{
   root = insert (d, root);
}
public BinaryTreeNode<dataType> insert ( dataType d, BinaryTreeNode<dataType> node )
{
   if (node == null)
      return new BinaryTreeNode<dataType> (d, null, null);
   if (d.compareTo (node.data) <= 0)
      node.left = insert (d, node.left);
   else
      node.right = insert (d, node.right);
   return balance (node);
}
```

delete Algorithm

- Rebalance nodes all the way from node to root.
- Rebalance nodes also when removing the minimum.

- Use same balance function and rotations as before.

delete

```
public BinaryTreeNode<dataType> delete ( dataType d, BinaryTreeNode<dataType> node )
{
   if (node == null) return null;
   if (d.compareTo (node.data) < 0)
      node.left = delete (d, node.left);
   else if (d.compareTo (node.data) > 0)
      node.right = delete (d, node.right);
   else
   {
      BinaryTreeNode<dataType> q = node.left;
      BinaryTreeNode<dataType> r = node.right;
      if (r == null)
         return q;
      BinaryTreeNode<dataType> min = findMin (r);
      min.right = removeMin (r);
      min.left = q;
      return balance (min);
   }
   return balance (node);
}
```

removeMin

```
   public BinaryTreeNode<dataType> removeMin ( BinaryTreeNode<dataType>
node )
   {
      if (node.left == null)
         return node.right;
      node.left = removeMin (node.left);
      return balance (node);
   }
```

Complexity Analysis

- Worst Case:
  - search - O(log n)
  - insert - O(log n)
  - delete - O(log n)

- Maximum depth of n-item tree is O(log n).

that's all folks!