Algorithms and Data Structures 2

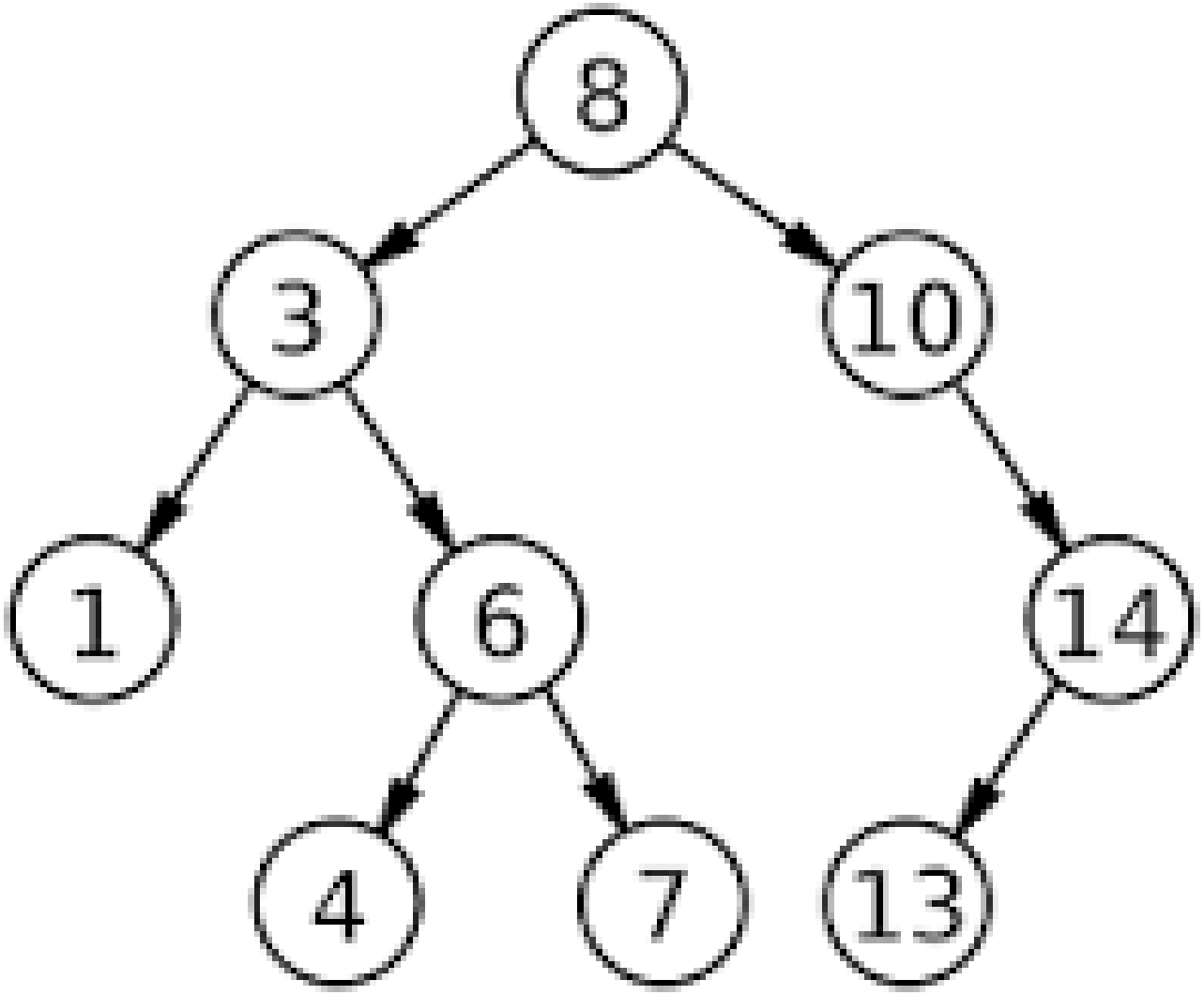
5. Binary Search Trees

Binary Search Trees

- A binary search tree is a binary tree with the added property that for each node, the left child is less than the parent is less than or equal to the right child.

- Given this refinement of our earlier definition of a binary tree, we can now include additional operations.

- A binary search tree (BST) is the most common type of tree and is regularly used when a large amount of data needs to be regularly accessed in memory.


8
3
10
1
6
14
4
7
13

The operations on a binary search tree

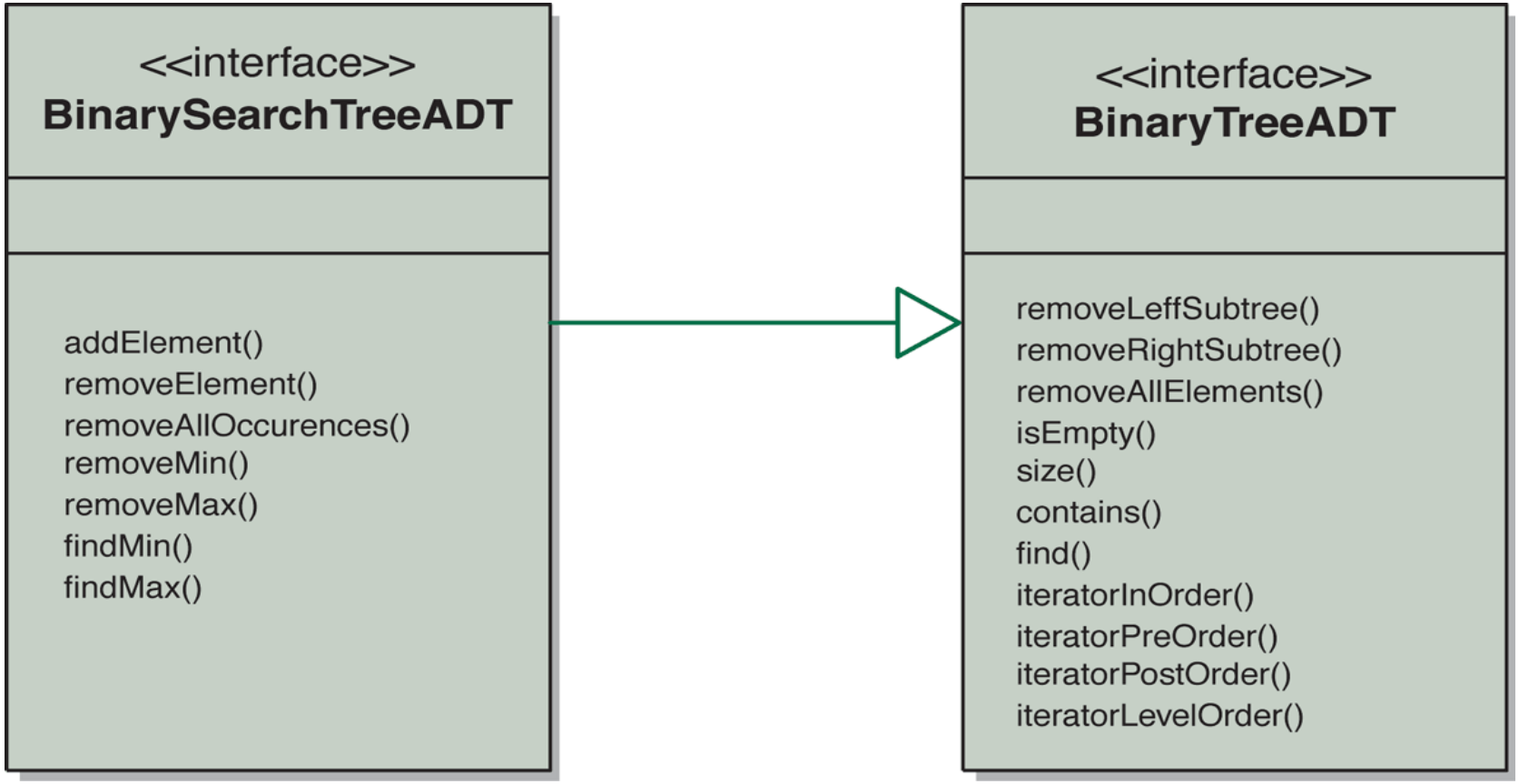
Operation	Description
`addElement`	Add an element to the tree.
`removeElement`	Remove an element from the tree.
`removeAllOccurrences`	Remove all occurrences of element from the tree.
`removeMin`	Remove the minimum element in the tree.
`removeMax`	Remove the maximum element in the tree.
`findMin`	Returns a reference to the minimum element in the tree.
`findMax`	Returns a reference to the maximum element in the tree.

BinarySearchTree ADT

```
public interface BinarySearchTreeADT extends BinaryTreeADT
{

   public void addElement (Object element);

   public Object removeElement (Object targetElement);

   public void removeAllOccurrences (Object targetElement);

   public Object removeMin();

   public Object removeMax();

   public Object findMin();

   public Object findMax();

}  // interface BinarySearchTreeADT
```

Again, I'm using Java here purely for clarity in showing what valid operations are allowed for a BST ADT. As we will be implementing the tree in C, we won't be using an interface.

UML description
of the BinarySearchTreeADT



Implementing Binary Search Trees With Pointers

- The definition of the node for a BST is the same as for a binary tree.
- The only difference between the two trees is in how the new nodes are added.
- Pointers are always used in the creation of BSTs. An array can be used to simulation the creation of a tree, but its very difficult to visualise and maintain.

BinarySearchTree - Data Structures

```
struct BinarySearchTreeNode {
        struct Node data;
        struct BinarySearchTreeNode *left;
        struct BinarySearchTreeNode *right;
}

struct BinarySearchTreeNode *root = NULL;

int treeCounter = 0;
```

Implementing Binary Search Trees With Links

- Now that we know more about how this tree is to be used (and structured) it is possible to define a method to add an element to the tree

- The `addElement()` method finds the proper location for the given element and adds it there as a leaf.
  1. Always start at the root i.e. let it be the current element.
  2. Compare the new element with the current element. If it is less than the current element, make the left child the new current element otherwise make the right node the new current element.
  3. Continue from 2. until the current element has no left or right children, then add the new element to the left or right of the current element depending on its value.

BinarySearchTree - addElement

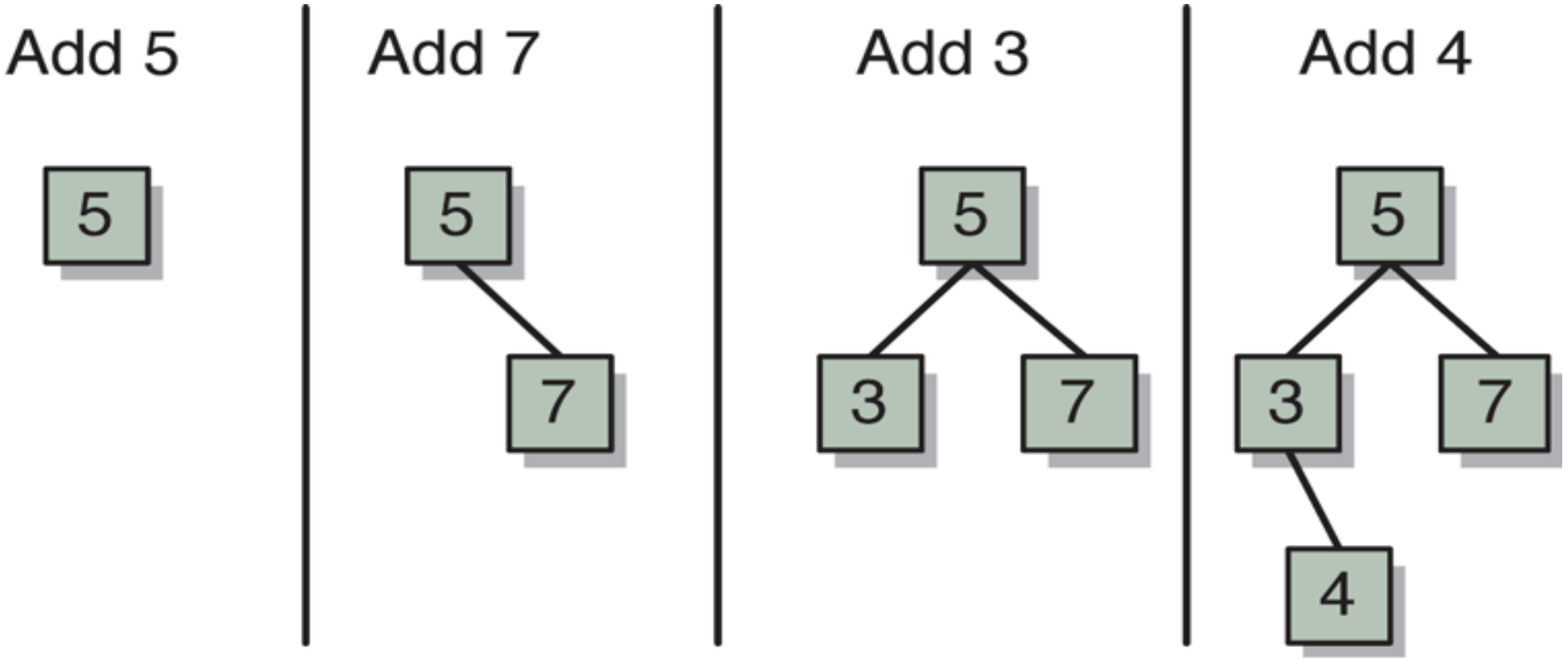
```
void addElement (struct Node element)
   {
      struct BinarySearchTreeNode *pointer, *current;
      pointer = createNode (element);

      if (isEmpty())  //set the root pointer
         root = pointer;
      else
      {
         current = root;
```

```
    while (current != NULL) {
      if (element.Operator < current->data.Operator) {
              if (current->left == NULL)
              {
                 current-> left = pointer;
                  current = NULL;
              }
              else
                 current = current->left;
      }
      else {
            if (current->right == NULL)
            {
                current->right = pointer;
               current = NULL;
              }
              else
                 current = current->right;
       } //end else
    }//end while
  } //end else
    treeCounter++;

}  // addElement
```

Create a new Node

```
struct BinarySearchTreeNode *createNode (struct Node aNode)
   {
   struct BinarySearchTreeNode *pointer;

   pointer = (struct BinarySearchTreeNode *)malloc(sizeof
   (struct BinarySearchTreeNode));
       pointer->data = aNode;
   pointer->left = NULL;
   pointer->right = NULL;

   return pointer;
}
```

Adding elements to a binary search tree



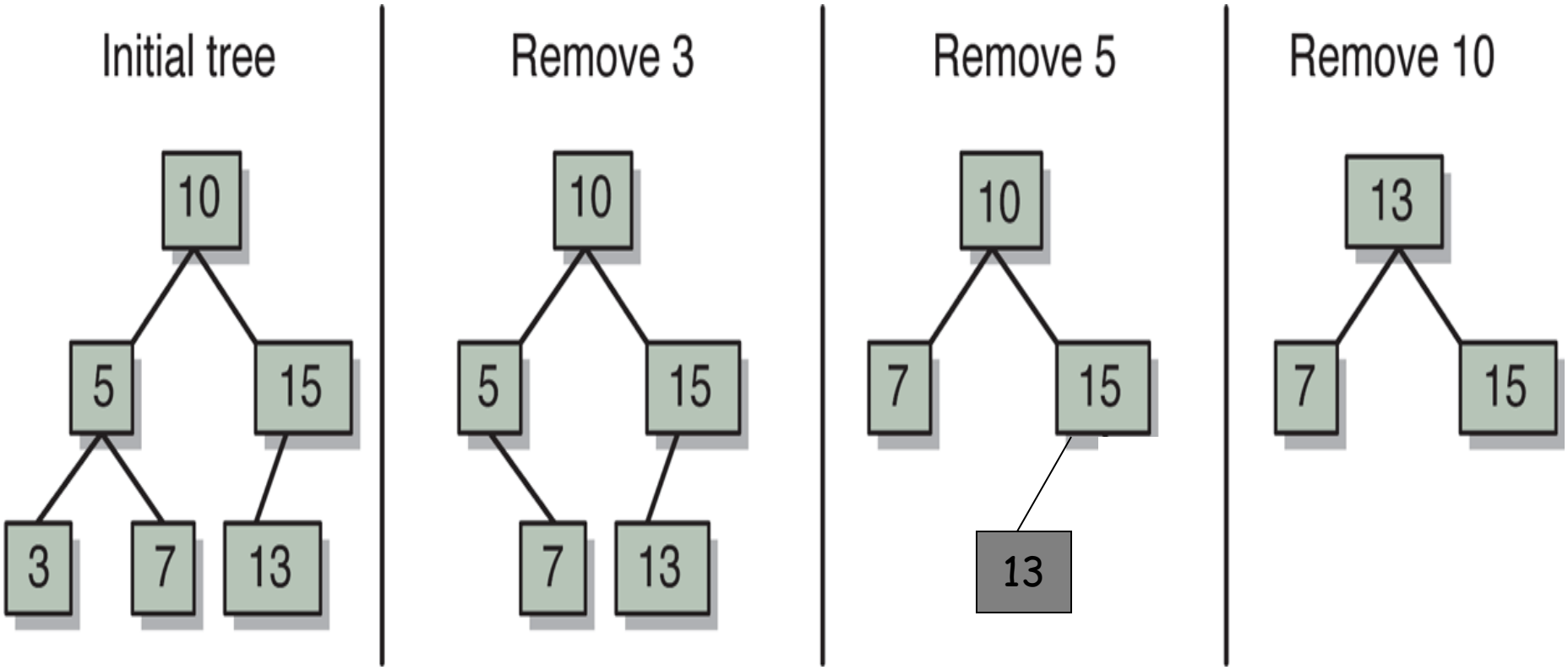
Removing Elements

- Removing elements from a binary search tree requires
  - Finding the element to be removed.
  - If that element is not a leaf, then replace it with its inorder successor.
  - Return the removed element.

•The `removeElement` method makes use of a `replacement` method to find the proper element to replace a non-leaf element that is removed.

- replacement() returns a reference to a node that will replace the the one specified for removal. There are three cases for selecting the replacement node :
  - If the node has no children, replacement returns NULL.
  - if the node has only one child, replacement returns that child.
  - If the node to be removed has two children, replacement returns the inorder successor of the node to be removed.
    - Inorder successor is the next highest (in terms of value) element in the tree. Returned by going right and then going left as far as you can.

Removing elements from a binary tree

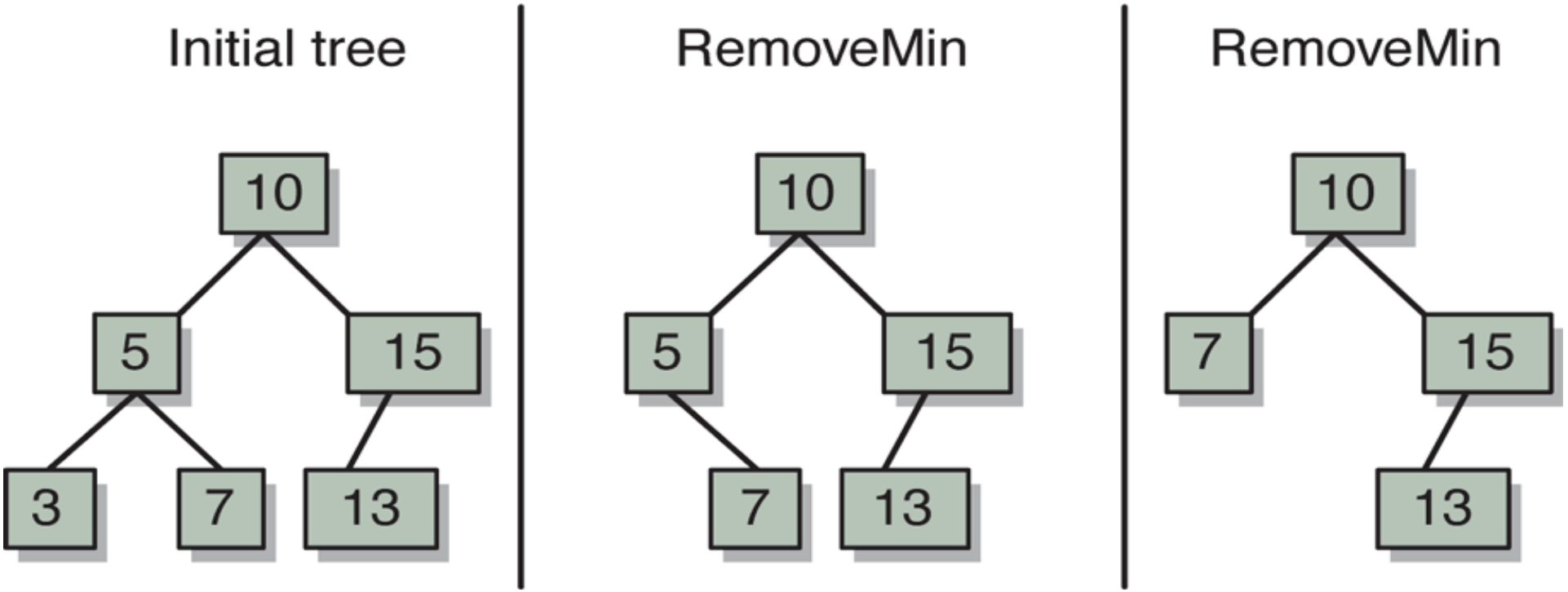


- `current` points to 5
- `parent` points to 10
- `parent->right` points to the replacement for 5

The removeMin Operation

- There are three cases for the location of the minimum element in a binary search tree:
  - If the root has no left child, then the root is the minimum element and the right child of the root becomes the new root.
  - If the leftmost node of the tree is a leaf, then we set its parent's left child reference to null.
  - If the leftmost node of the tree is an internal node, then we set its parent's left child reference to point to the right child of the node to be removed.

Removing the minimum element from a binary search tree



Using Binary Search Trees: Implementing Ordered Lists

- A binary search tree is frequently used to implement an **ordered list**.
- The elements are added to the tree using the process described on slide 8.
- Then an `inorder traversal` (see lecture 4) can be used to retrieve the elements in the correct order.
- The advantage of storing element in a BST as opposed to a linear list is that no explicit sorting is required and searching is frequently faster.

Analysis of linked list and binary search tree implementations of an ordered list

Operation	LinkedList	BinarySearchTreeList
`removeFirst`	O(1)	O(log n)
`removeLast`	O(n)	O(log n)
`remove`	O(n)	O(log n)*
`first`	O(1)	O(log n)
`last`	O(n)	O(log n)
`contains`	O(n)	O(log n)
`isEmpty`	O(1)	O(1)
`size`	O(1)	O(1)
`add`	O(n)	O(log n)*
*both the `add` and `remove` operations may cause the tree to become unbalanced		

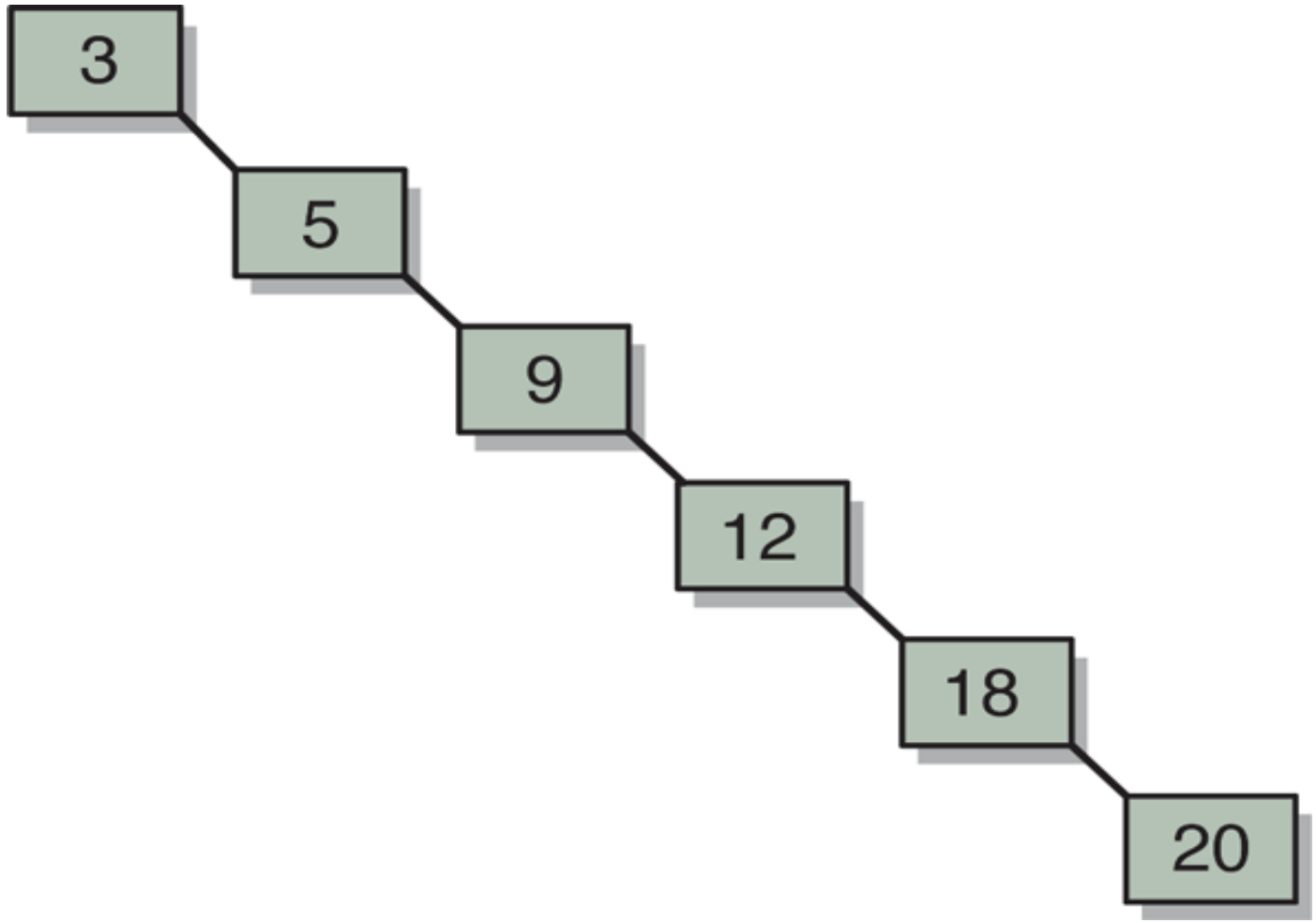
Balanced Binary Search Trees

- Why is our balance assumption so important?

- Lets look at what happens if we insert the following numbers in order without rebalancing the tree:

  3 5 9 12 18 20

A degenerate binary tree



Degenerate Binary Trees

- The resulting tree is called a degenerate binary tree

- Degenerate binary search trees are far less efficient than balanced binary search trees (O(n) on find as opposed to O(logn))

Balancing Binary Trees

- There are many approaches to balancing binary trees

- One method is brute force
  - Write an inorder traversal to a file
  - Use a recursive binary search of the file to rebuild the tree

Balancing Binary Trees

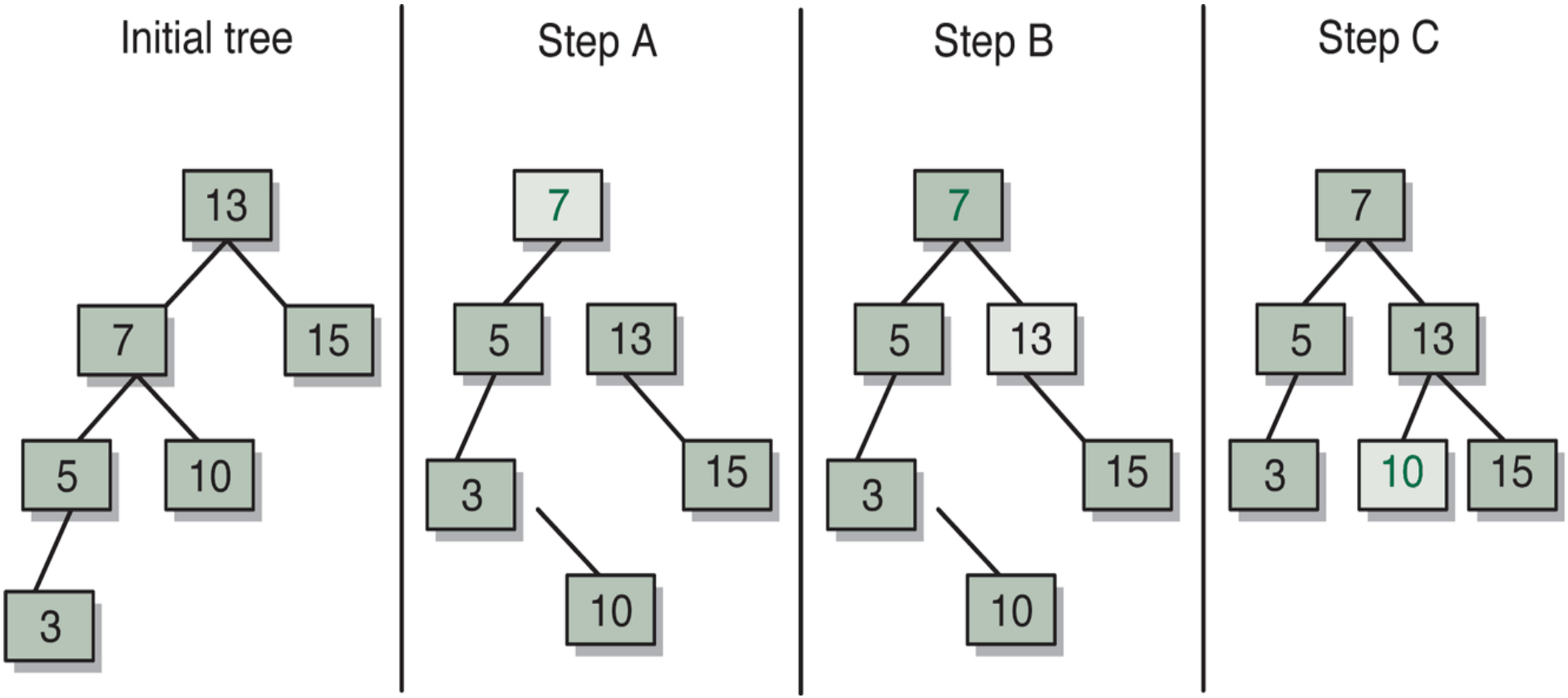
- Better solutions involve algorithms such as red-black trees and AVL trees that persistently maintain the balance of the tree.
- Most all of these algorithms make use of rotations to balance the tree.
- Before looking at one of these algorithms (for AVL trees), lets first examine each of the possible rotations.

Right Rotation

- Right rotation will solve an imbalance if it is caused by a long path in the left sub-tree of the left child of the root.

  - Make the left child element of the root the new root element.
  - Make the former root element the right child element of the new root.
  - Make the right child of what was the left child of the former root the new left child of the former root.

Unbalanced tree and balanced tree after a right rotation

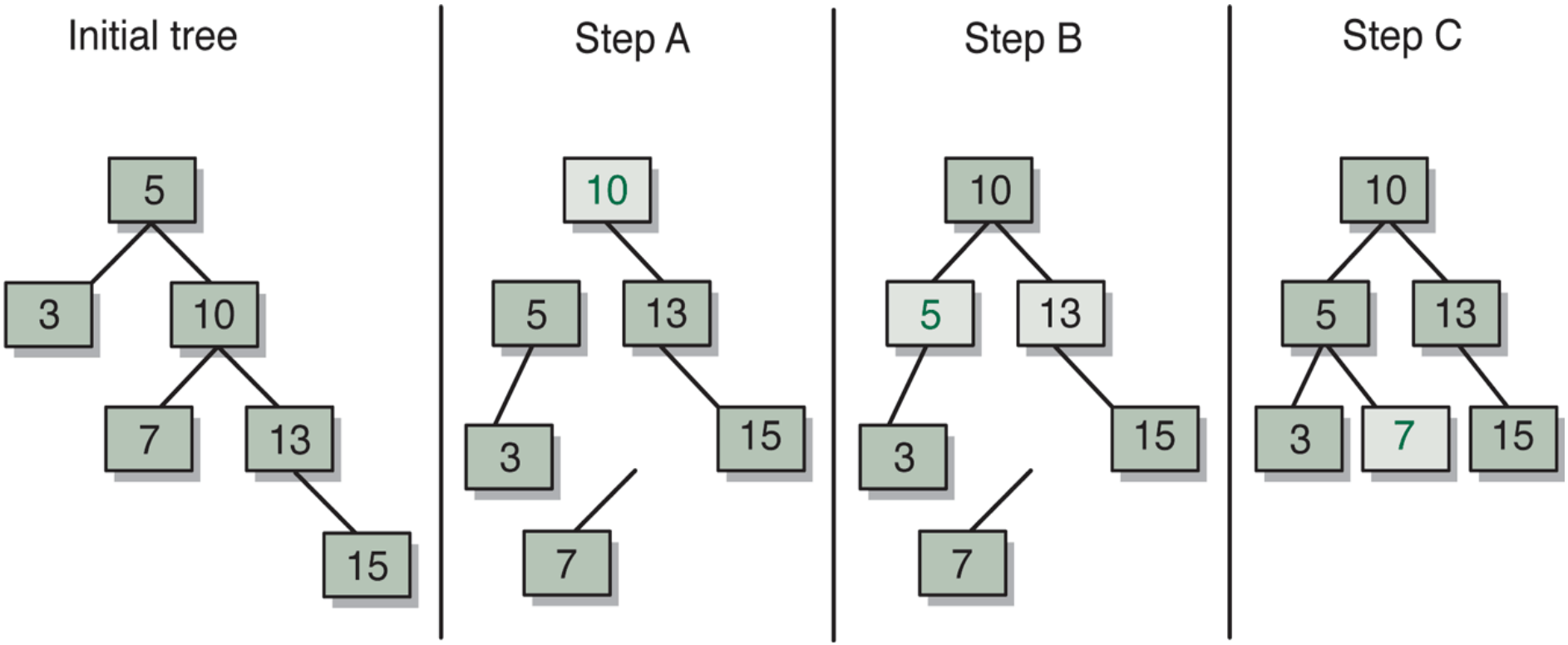


Left Rotation

- Left rotation will solve an imbalance if it is caused by a long path in the right sub-tree of the right child of the root.

- Make the right child element of the root the new root element.
- Make the former root element the left child element of the new root.
- Make the left child of what was the right child of the former root the new right child of the former root.

Unbalanced tree and balanced tree after a left rotation

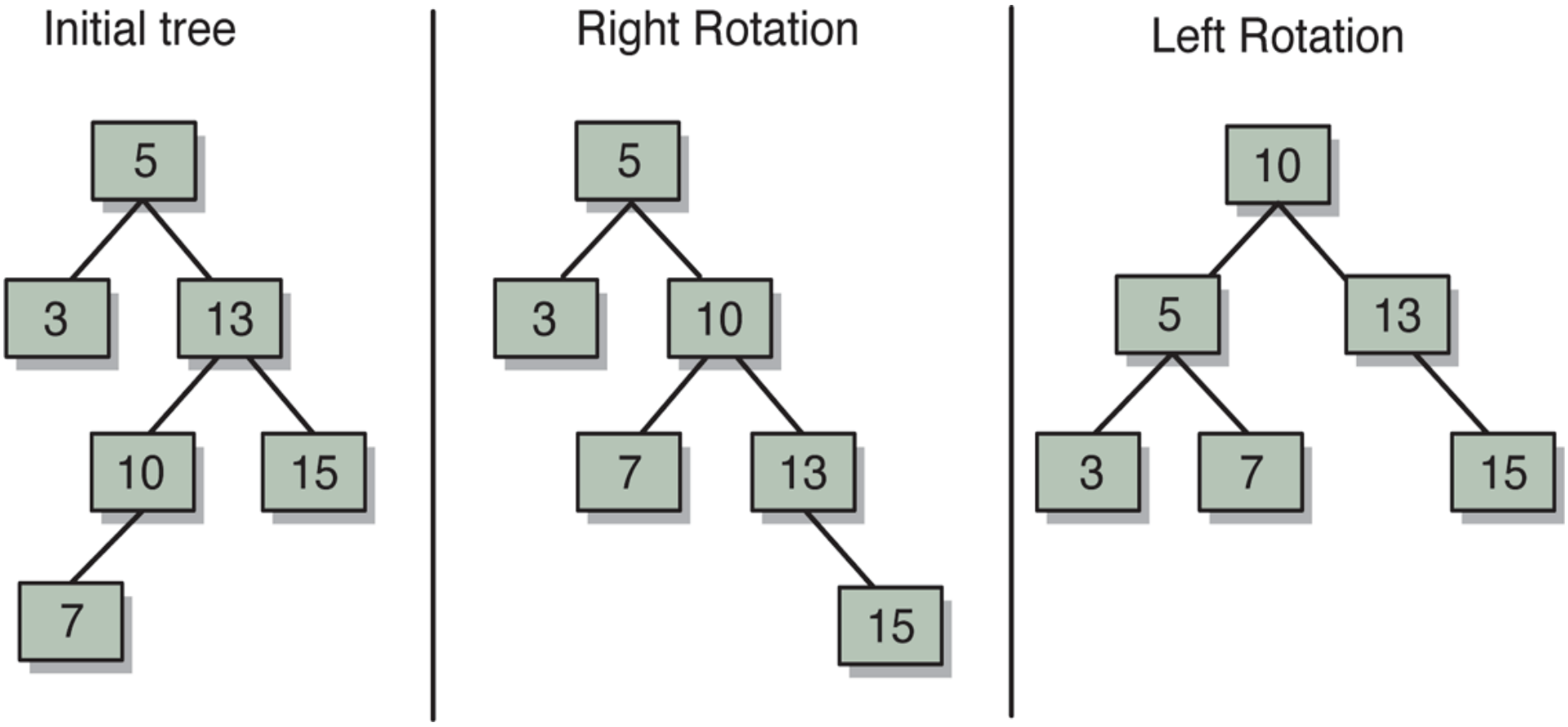


Rightleft Rotation

- Rightleft rotation will solve an imbalance if it is caused by a long path in the left sub-tree of the right child of the root.

- Perform a right rotation of the left child of the right child of the root around the right child of the root, and then perform a left rotation of the resulting right child of the root around the root.

A rightleft rotation

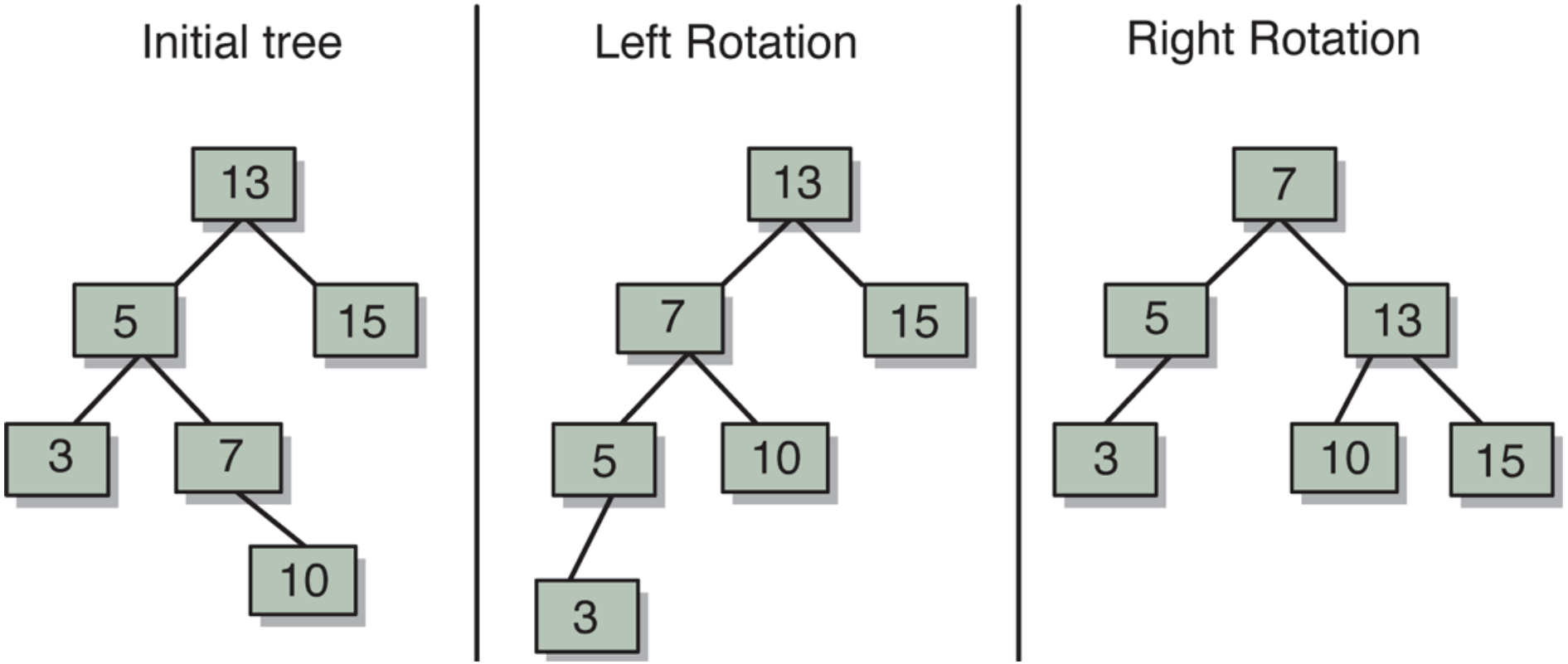


Leftright Rotation

- Leftright rotation will solve an imbalance if it is caused by a long path in the right sub-tree of the left child of the root.

- Perform a left rotation of the right child of the left child of the root around the left child of the root, and then perform a right rotation of the resulting left child of the root around the root.

A leftright rotation



AVL Trees

- AVL trees keep track of the difference in height between the right and left sub-trees for each node.

An easy way to calculate this is to subtract the height of the left subtree from the right subtree.

- This difference is called the balance factor.

- If the balance factor of any node is less than -1 or greater than 1, then that sub-tree needs to be rebalanced.

- The balance factor of any node can only be changed through either insertion or deletion of nodes in the tree.

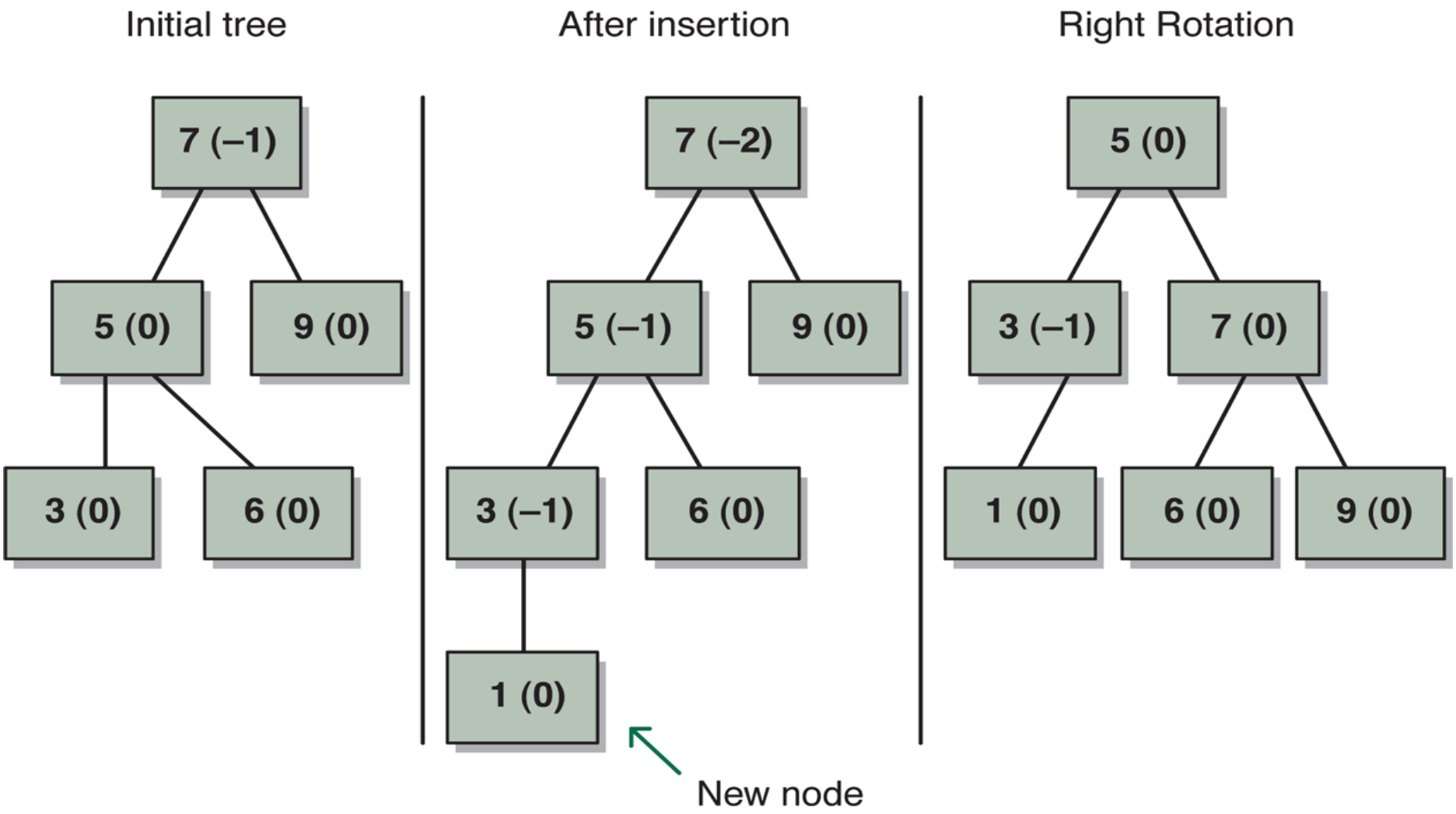
AVL Trees

- If the balance factor of a node is -2, this means the left sub-tree has a path that is too long.

- If the balance factor of the left child is -1, this means that the long path is the left sub-tree of the left child.

- In this case, a simple right rotation of the left child around the original node will solve the imbalance.

A right rotation in an AVL tree

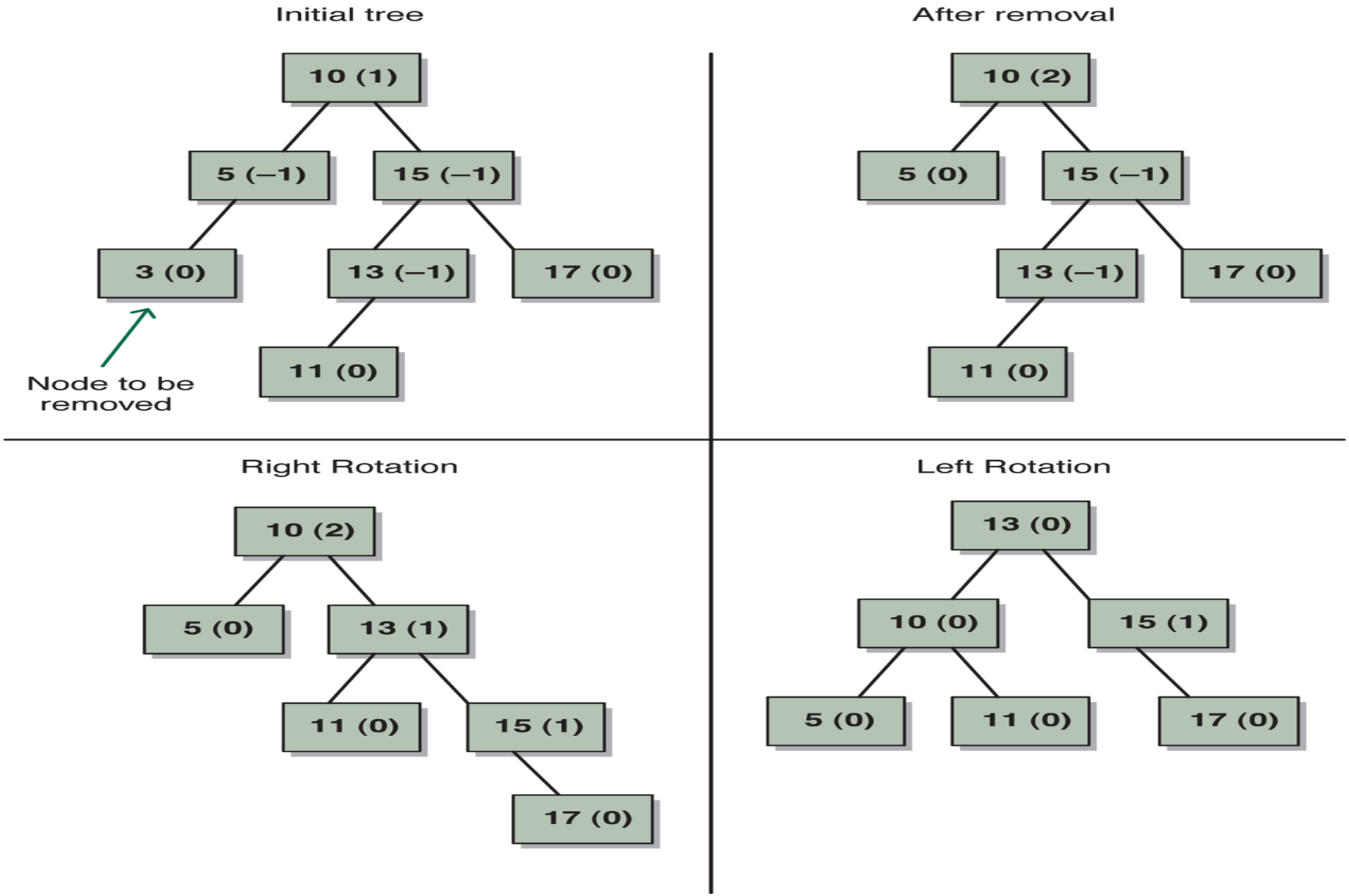


AVL Trees

- If the balance factor of a node is +2, this means the right sub-tree has a path that is too long.
- Then if the balance factor of the right child is +1, this means that the long path is the right sub-tree of the right child.
- In this case, a simple left rotation of the right child around the original node will solve the imbalance.

AVL Trees

- If the balance factor of a node is +2, this means the right sub-tree has a path that is too long.
- Then if the balance factor of the right child is -1, this means that the long path is the left sub-tree of the right child.
- In this case, a rightleft double rotation will solve the imbalance. For example, see the next slide.


Initial tree
10 (1)
5 (–1)
15 (–1)
3 (0)
13 (–1)
17 (0)
11 (0)
Node to be removed
After removal
10 (2)
5 (0)
15 (–1)
13 (–1)
17 (0)
11 (0)
Right Rotation
10 (2)
5 (0)
13 (1)
11 (0)
15 (1)
17 (0)
Left Rotation
13 (0)
10 (0)
15 (1)
5 (0)
11 (0)
17 (0)

AVL Trees

- If the balance factor of a node is -2, this means the right sub-tree has a path that is too long.
- Then if the balance factor of the left child is +1, this means that the long path is the right sub-tree of the left child.
- In this case, a leftright double rotation will solve the imbalance.

Its important to understand the difference between a binary search in a list (array or linked list) and a search in a BST.

Binary Search Trees in the Java Collections API

- Java provides two implementations of balanced binary search trees.
  - TreeSet
  - TreeMap