Data Structures
Lesson 7

BSc in Computer Science
University of New York, Tirana

Assoc. Prof. Dr. Marenglen Biba

Binary Search Trees

- For large amounts of input, the linear access time of linked lists is prohibitive.
- We now look at an alternative to the linked list.
- The binary search tree, a simple data structure that can be viewed as extending the binary search algorithm to allow insertions and deletions.
- The running time for most operations is O(logN) on average.
- Unfortunately, the worst-case time is O(N) per operation.

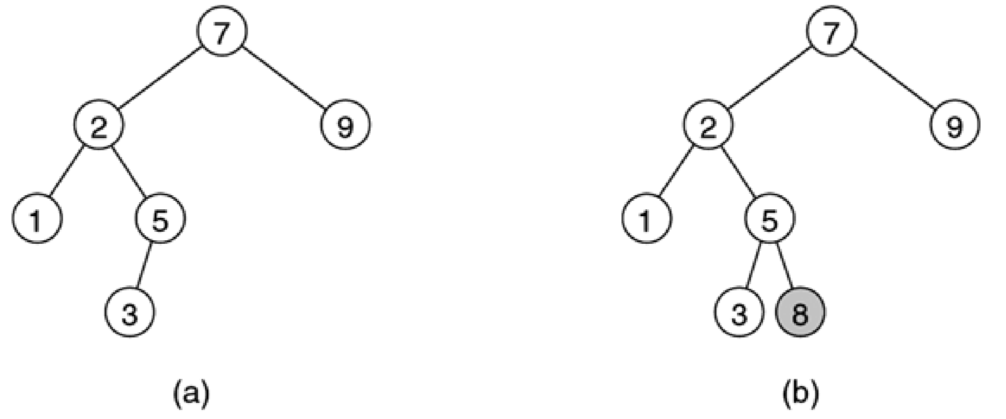
Basic idea

- In the general case, we search for an item (or element) by using its key.
- For instance, a student transcript could be searched on the basis of a student ID number.
- In this case, the ID number is referred to as the item's key.
- The binary search tree satisfies the search order property; that is, for every node X in the tree, the values of all the keys in the left subtree are smaller than the key in X and the values of all the keys in the right subtree are larger than the key in X.

Binary search trees

figure 19.1

Two binary trees: (a) a search tree; (b) not a search tree



Insertion in binary search trees

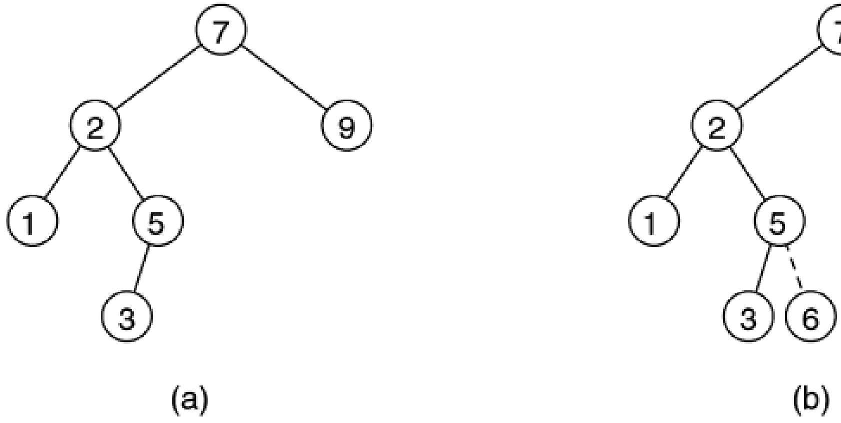
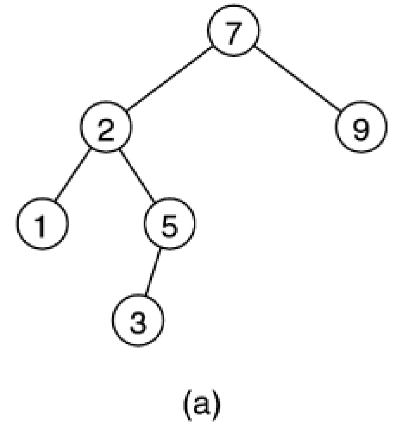


figure 19.2

Binary search trees (a) before and (b) after the insertion of 6

Deletion in binary search trees



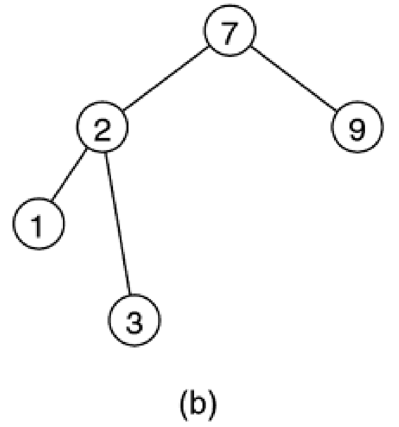


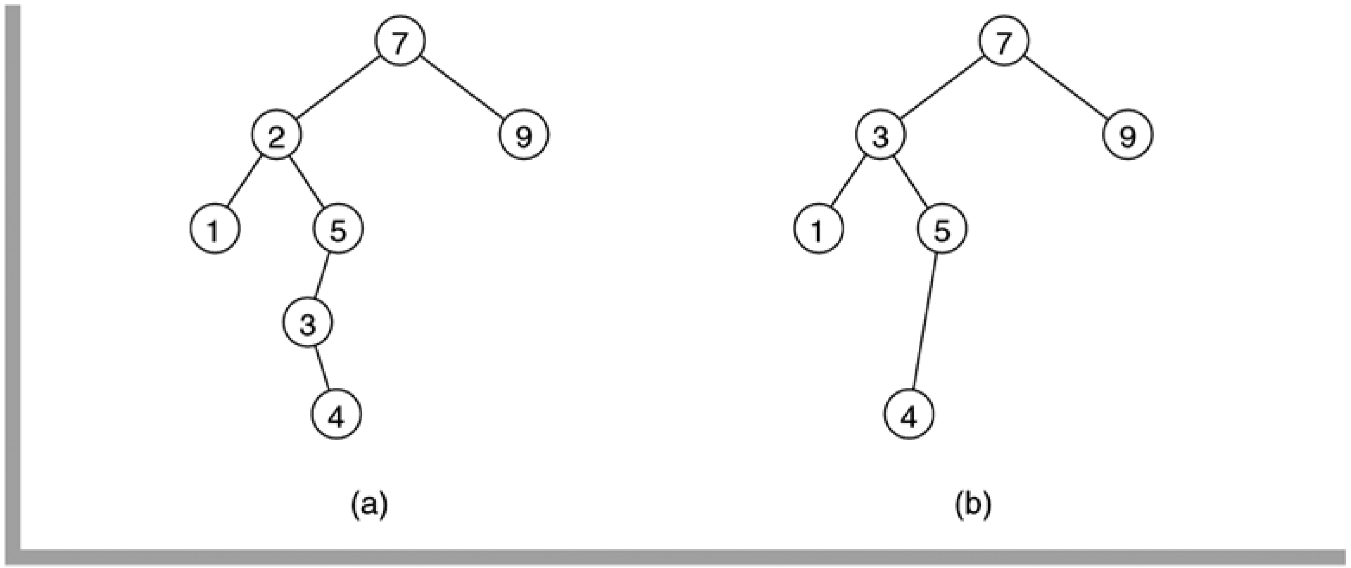
figure 19.3

Deletion of node 5 with one child: (a) before and (b) after

Deletion in binary search trees

figure 19.4

Deletion of node 2 with two children: (a) before and (b) after



The general strategy is to replace the item in node 2 with the smallest item in the right subtree and then remove that node

Implementation in Java

```
package weiss.nonstandard;

// Basic node stored in unbalanced binary search trees
// Note that this class is not accessible outside
// this package.

class BinaryNode<AnyType>
{
        // Constructor
    BinaryNode( AnyType theElement )
    {
        element = theElement;
        left = right = null;
    }

      // Data; accessible by other package routines
    AnyType             element;  // The data in the node
    BinaryNode<AnyType> left;     // Left child
    BinaryNode<AnyType> right;    // Right child
}
```

figure 19.5

The `BinaryNode` class for the binary search tree

```java
package weiss.nonstandard;

// BinarySearchTree class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x
// void removeMin( )      --> Remove minimum item
// Comparable find( x )   --> Return item that matches x
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// ******************ERRORS********************************
// Exceptions are thrown by insert, remove, and removeMin if warranted

public class BinarySearchTree<AnyType extends Comparable<? super AnyType>>
{
    public BinarySearchTree( )
      {  root = null; }

    public void insert( AnyType x )
      { root = insert( x, root ); }
```

figure 19.6a

The BinarySearchTree class skeleton (*continues*)

```
    public void remove( AnyType x )
      { root = remove( x, root ); }
    public void removeMin( )
      { root = removeMin( root ); }
    public AnyType findMin( )
      { return elementAt( findMin( root ) ); }
    public AnyType findMax( )
      { return elementAt( findMax( root ) ); }
    public AnyType find( AnyType x )
      { return elementAt( find( x, root ) ); }
    public void makeEmpty( )
      { root = null; }
    public boolean isEmpty( )
      { return root == null; }

    private AnyType elementAt( BinaryNode<AnyType> t )
      { /* Figure 19.7 */ }
    private BinaryNode<AnyType> find( AnyType x, BinaryNode<AnyType> t )
      { /* Figure 19.8 */ }
    protected BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
      { /* Figure 19.9 */ }
    private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
      { /* Figure 19.9 */ }
    protected BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )
      { /* Figure 19.10 */ }
    protected BinaryNode<AnyType> removeMin( BinaryNode<AnyType> t )
      { /* Figure 19.11 */ }
    protected BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
      { /* Figure 19.12 */ }

    protected BinaryNode<AnyType> root;
}
```

figure 19.6b

The `BinarySearchTree` class skeleton (*continued*)

```
/**
 * Internal method to get element field.
 * @param t the node.
 * @return the element field or null if t is null.
 */
private AnyType elementAt( BinaryNode<AnyType> t )
{
    return t == null ? null : t.element;
}
```

figure 19.7

The `elementAt` method

```
/**
 * Internal method to find an item in a subtree.
 * @param x is item to search for.
 * @param t the node that roots the tree.
 * @return node containing the matched item.
 */
private BinaryNode<AnyType> find( AnyType x, BinaryNode<AnyType> t )
{
    while( t != null )
    {
        if( x.compareTo( t.element ) < 0 )
            t = t.left;
        else if( x.compareTo( t.element ) > 0 )
            t = t.right;
        else
            return t;    // Match
    }

    return null;         // Not found
}
```

figure 19.8

The `find` operation for binary search trees

findMin and findMax

figure 19.9

The `findMin` and `findMax` methods for binary search trees

```java
/**
 * Internal method to find the smallest item in a subtree.
 * @param t the node that roots the tree.
 * @return node containing the smallest item.
 */
protected BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
{
    if( t != null )
        while( t.left != null )
            t = t.left;

    return t;
}

/**
 * Internal method to find the largest item in a subtree.
 * @param t the node that roots the tree.
 * @return node containing the largest item.
 */
private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
{
    if( t != null )
        while( t.right != null )
            t = t.right;

    return t;
}
```

Insert

```
/**
 * Internal method to insert into a subtree.
 * @param x the item to insert.
 * @param t the node that roots the tree.
 * @return the new root.
 * @throws DuplicateItemException if x is already present.
 */
protected BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )
{
    if( t == null )
        t = new BinaryNode<AnyType>( x );
    else if( x.compareTo( t.element ) < 0 )
        t.left = insert( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
        t.right = insert( x, t.right );
    else
        throw new DuplicateItemException( x.toString( ) );  // Duplicate
    return t;
}
```

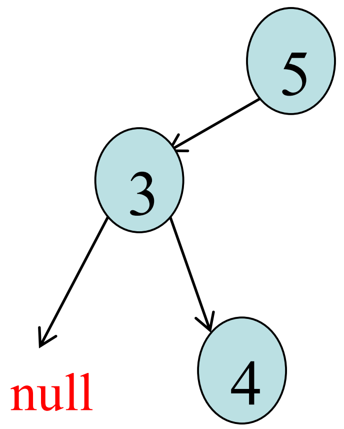
figure 19.10

The recursive `insert` for the `BinarySearchTree` class

removeMin

figure 19.11

The removeMin method for the BinarySearchTree class

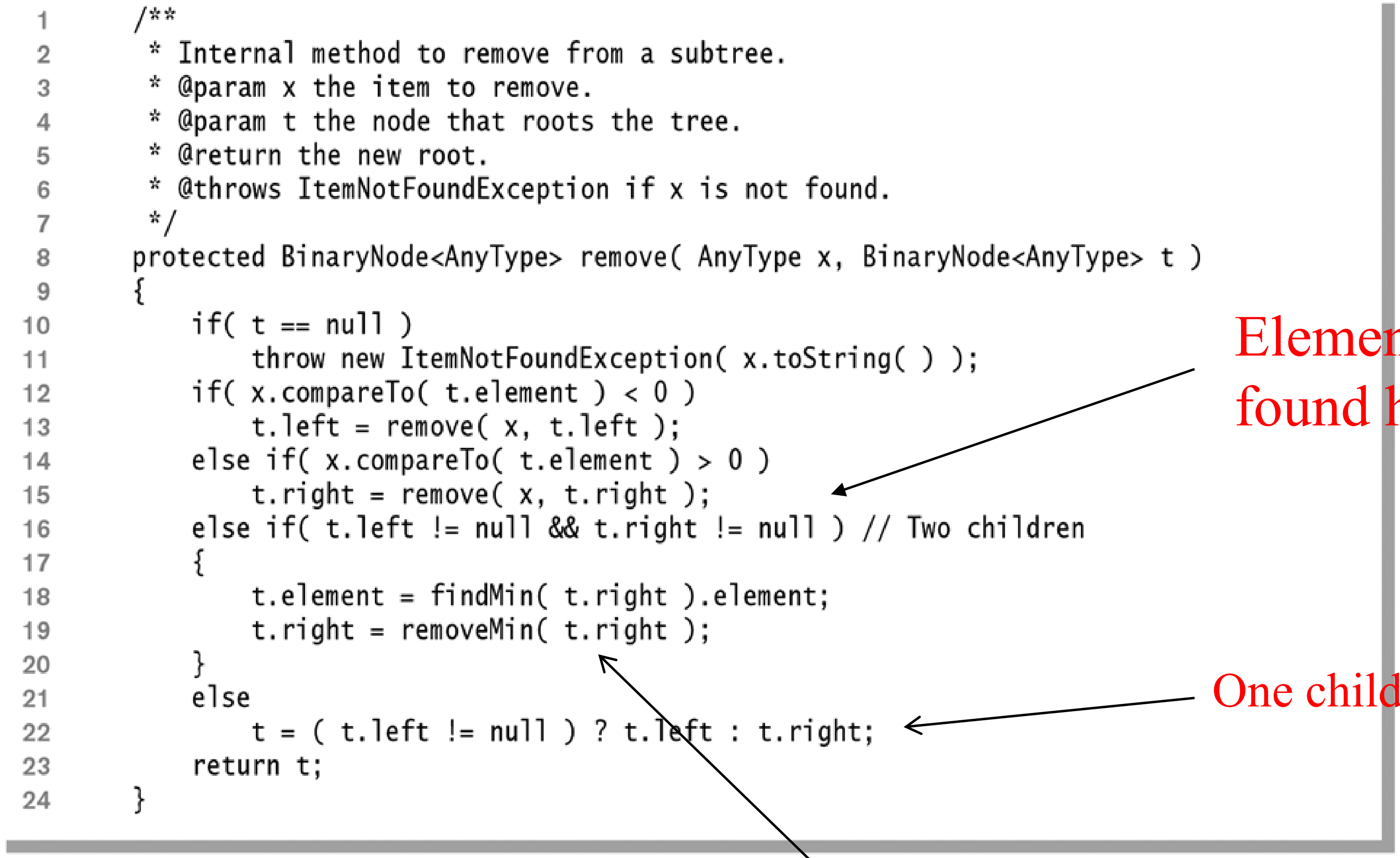


```
/**
 * Internal method to remove minimum item from a subtree.
 * @param t the node that roots the tree.
 * @return the new root.
 * @throws ItemNotFoundException if t is empty.
 */
protected BinaryNode<AnyType> removeMin( BinaryNode<AnyType> t )
{
    if( t == null )
        throw new ItemNotFoundException( );
    else if( t.left != null )
    {
        t.left = removeMin( t.left );
        return t;
    }
    else
        return t.right;
}
```

Recursive call to left

Delete the minimum. We go left until finding a node that has a null left link and then replace the link to that node by its right link. The symmetric method works for delete the maximum.

Remove



```
/**
 * Internal method to remove from a subtree.
 * @param x the item to remove.
 * @param t the node that roots the tree.
 * @return the new root.
 * @throws ItemNotFoundException if x is not found.
 */
protected BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
{
    if( t == null )
        throw new ItemNotFoundException( x.toString( ) );
    if( x.compareTo( t.element ) < 0 )
        t.left = remove( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
        t.right = remove( x, t.right );
    else if( t.left != null && t.right != null ) // Two children
    {
        t.element = findMin( t.right ).element;
        t.right = removeMin( t.right );
    }
    else
        t = ( t.left != null ) ? t.left : t.right;
    return t;
}
```

figure 19.12
The remove method for the BinarySearchTree class

Element is found here

One child

The general strategy is to replace the item in node X with the smallest item in the right subtree and then remove that node

Analysis of binary search tree operations

- The cost of each binary search tree operation (insert, find, and remove) is proportional to the number of nodes accessed during the operation.

- We can thus charge the access of any node in the tree a cost of 1 plus its depth.
  - Recall that the depth measures the number of edges on a path rather than the number of nodes, which gives the cost of a successful search.

Analysis of binary search tree operations

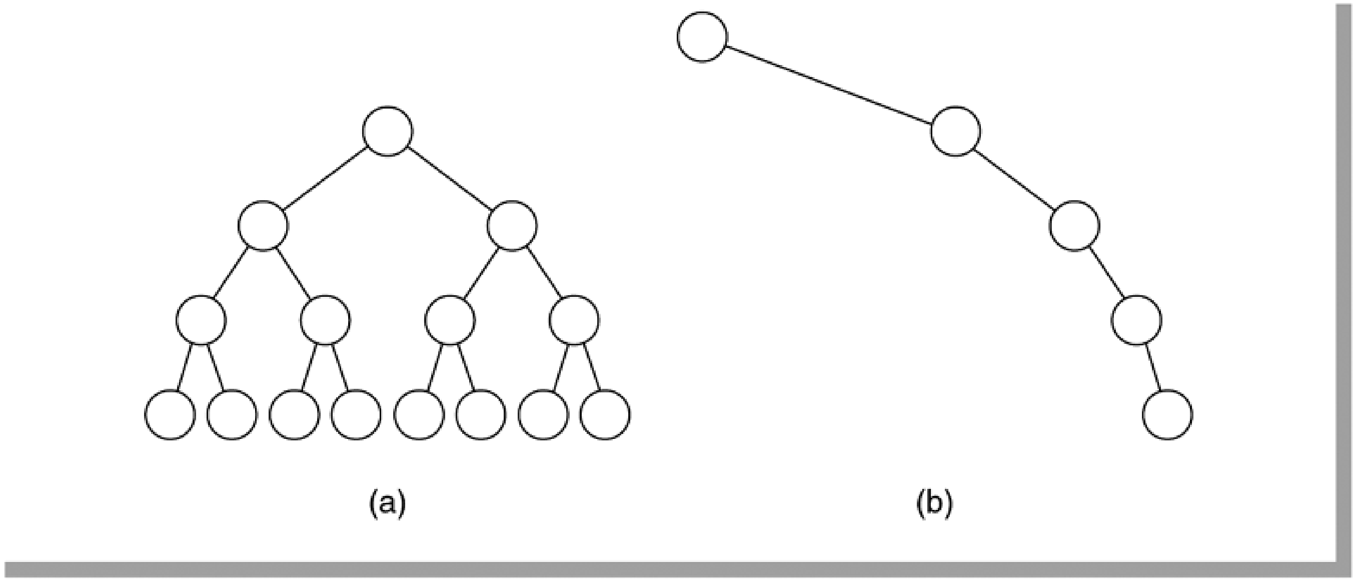


figure 19.19

(a) The balanced tree has a depth of $\lfloor \log N \rfloor$; (b) the unbalanced tree has a depth of $N - 1$.

Figure 19.19(a) shows a balanced tree of 15 nodes.
The cost to access any node is at most 4 units, and some nodes require fewer accesses.
This situation is analogous to the one that occurs in the binary search algorithm. If the tree is perfectly balanced, the access cost is logarithmic.

Unbalanced Trees

- Unfortunately, we have no guarantee that the tree is perfectly balanced.
- The tree shown in Figure 19.19(b) is the classic example of an unbalanced tree.
- Here, all N nodes are on the path to the deepest node, so the worst-case search time is O(N).
- Because the search tree has degenerated to a linked list, the average time required to search in this particular instance is half the cost of the worst case and is also O(N).

Average cost for most BSTs

- So we have two extremes: In the best case, we have logarithmic access cost, and in the worst case we have linear access cost.
- What is the average?
- Do most binary search trees tend toward the balanced or unbalanced case, or is there some middle ground?
- The answer is: The average is 38 percent worse than the best case.

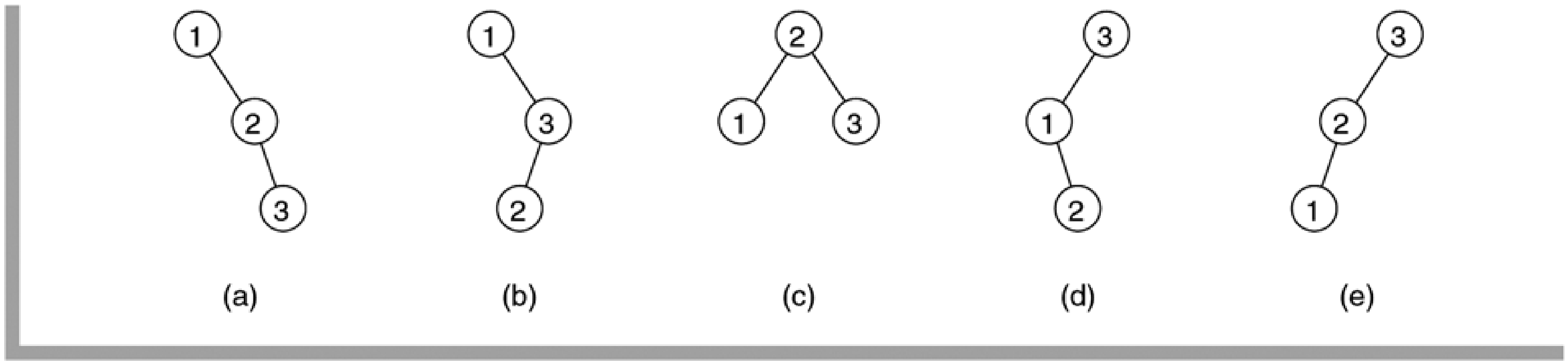


figure 19.20

Binary search trees that can result from inserting a permutation 1, 2, and 3; the balanced tree shown in part (c) is twice as likely to result as any of the others.

There are six possible insertion orders: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), and (3, 2, 1).

Note that the tree with root 2, shown in Figure 19.20(c), is formed from either the insertion sequence (2, 3, 1) or the sequence (2, 1, 3). Thus some trees are more likely to result than others, and balanced trees are more likely to occur than unbalanced trees

Average running time

- The average running time of all operations is O(logN).
- This implication is true in practice, but it has not been established analytically because the assumption used to prove this does not take into account the deletion algorithm.
- In fact, close examination suggests that we might be in trouble with our deletion algorithm because the remove operation always replaces a two-child deleted node with a node from the right subtree.
- This result would seem to have the effect of eventually unbalancing the tree and tending to make it left-heavy.

Logarithmic average time

- It has been shown that if we build a random binary search tree and then perform roughly N^2 pairs of random insert/remove combinations, the binary search trees will have an expected depth of $O(\sqrt{N})$.
- However, a reasonable number of random insert and remove operations (in which the order of insert and remove is also random) does not unbalance the tree in any observable way.
- In fact, for small search trees, the remove algorithm seems to balance the tree.

- *Consequently, we can reasonably assume that for random input all operations behave in logarithmic average time, although this result has not been proved mathematically.*

Balanced binary search trees

Putting Balance

- The most important problem is not the potential imbalance caused by the remove algorithm.
- Rather, it is that, if the input sequence is sorted, the worst-case tree occurs.
- When that happens, we are in deep trouble: We have linear time per operation (for a series of N operations) rather than logarithmic cost per operation.
- One solution to this problem is to insist on an extra structural condition called balance: No node is allowed to get too deep.

Balanced binary search tree

- Any of several algorithms can be used to implement a balanced binary search tree, which has an added structure property that guarantees logarithmic depth in the worst case.
- Most of these algorithms are much more complicated than those for the standard binary search trees, and all take longer on average for insertion and deletion.
- They do, however, provide protection against the embarrassingly simple cases that lead to poor performance for (unbalanced) binary search trees.
- Also, because they are balanced, they tend to give faster access time than those for the standard trees.

AVL trees

- The first balanced binary search tree was the AVL tree (named after its discoverers, Adelson-Velskii and Landis), which illustrates the ideas that are thematic for a wide class of balanced binary search trees.
- It is a binary search tree that has an additional balance condition.
- Any balance condition must be easy to maintain and ensures that the depth of the tree is O(log N).

AVL trees

- The simplest idea is to require that the left and right subtrees have the same height.
- Recursion dictates that this idea apply to all nodes in the tree because each node is itself a root of some subtree.
- This balance condition ensures that the depth of the tree is logarithmic.
- However, it is too restrictive because inserting new items while maintaining balance is too difficult.
- Thus the definition of an AVL tree uses a notion of balance that is somewhat weaker but still strong enough to guarantee logarithmic depth.

AVL Trees: Definition

- Definition: An AVL tree is a binary search tree with the additional balance property that, for any node in the tree, the height of the left and right subtrees can differ by at most 1. As usual, the height of an empty subtree is -1.

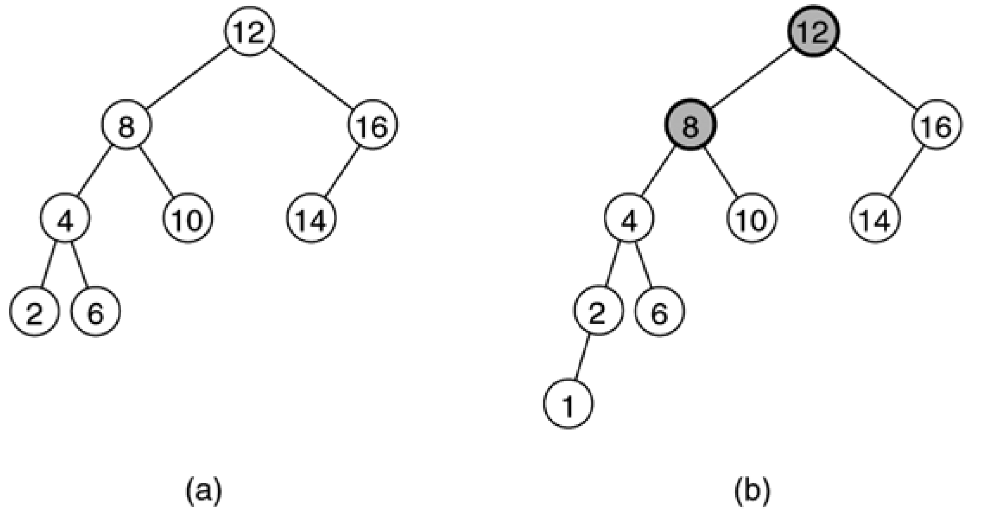


figure 19.21

Two binary search trees: (a) an AVL tree; (b) not an AVL tree (unbalanced nodes are darkened)

b) is not an AVL tree because the darkened nodes have left subtrees whose heights are 2 larger than their right subtrees.
If 13 were inserted, using the usual binary search tree insertion algorithm, node 16 would also be in violation.
The reason is that the left subtree would have height 1, while the right subtree would have height -1.

Logarithmic worst-case bound

- The depth of an average node in a randomly constructed AVL tree tends to be very close to logN.
- The exact answer has not yet been established analytically.
- All searching operations in an AVL tree have logarithmic worst-case bounds.
- The difficulty is that operations that change the tree, such as insert and remove, are not quite as simple as before.
- The reason is that an insertion (or deletion) can destroy the balance of several nodes in the tree.

Insertion

- A key observation is that after an insertion, only nodes that are on the path from the insertion point to the root might have their balances altered because only those nodes have their subtrees altered.
- This result applies to almost all the balanced search tree algorithms.
- As we follow the path up to the root and update the balancing information, we may find a node whose new balance violates the AVL condition.
- In this section we show how to rebalance the tree at the first (i.e., the deepest) such node and see that this rebalancing guarantees that the entire tree satisfies the AVL property.

Rebalancing

- The node to be rebalanced is X.
- Because any node has at most two children and a height imbalance requires that the heights of X's two subtrees differ by 2, a violation might occur in any of four cases:

1. An insertion in the left subtree of the left child of X
2. An insertion in the right subtree of the left child of X
3. An insertion in the left subtree of the right child of X
4. An insertion in the right subtree of the right child of X

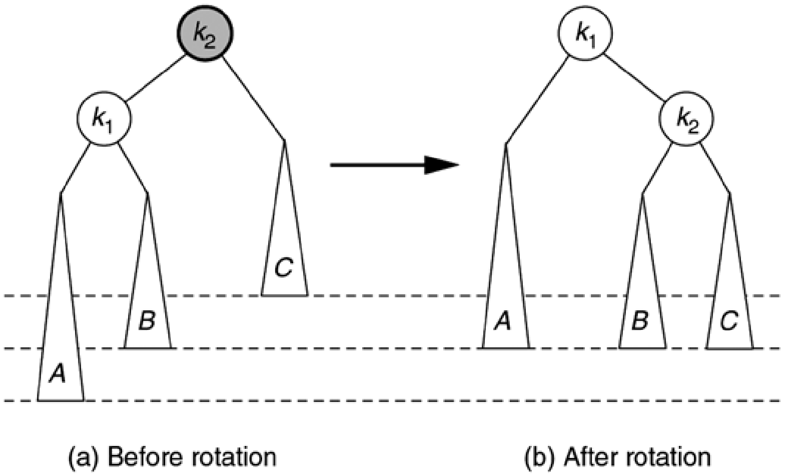
- Cases 1 and 4 are mirror-image symmetries with respect to X, as are cases 2 and 3. Consequently, there are theoretically two basic cases.
- From a programming perspective, of course, there are still four cases.

Tree Rotations

- The first case, in which the insertion occurs on the outside (i.e., left-left or right-right), is fixed by a single rotation of the tree.
- A single rotation switches the roles of the parent and child while maintaining search order.
- The second case, in which the insertion occurs on the inside (i.e., left-right or right-left), is handled by the slightly more complex double rotation.
- These fundamental operations on the tree are used several times in balanced tree algorithms.

figure 19.23

Single rotation to fix case 1



Node k2 violates the AVL balance property because its left subtree is two levels deeper than its right subtree (the dashed lines mark the levels).
The situation depicted is the only possible case 1 scenario that allows k2 to satisfy the AVL property before the insertion but violate it afterward.
Subtree A has grown to an extra level, causing it to be two levels deeper than C.
Subtree B cannot be at the same level as the new A because then k2 would have been out of balance before the insertion.
Subtree B cannot be at the same level as C because then k1 would have been the first node on the path that was in violation of the AVL balancing condition.

Rebalancing the tree

- Ideally, to rebalance the tree, we want to move A up one level and C down one level.
- Note that these actions are more than the AVL property requires.
- Here is an abstract scenario:
  - k1 will be the new root.
  - The binary search tree property tells us that in the original tree, k2 > k1, so k2 becomes the right child of k1 in the new tree.
  - Subtrees A and C remain as the left child of k1 and the right child of k2, respectively.

Single rotation: Case 1

figure 19.24

Pseudocode for a single rotation (case 1)

```
/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 */
static BinaryNode rotateWithLeftChild( BinaryNode k2 )
{
    BinaryNode k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    return k1;
}
```

Single rotation

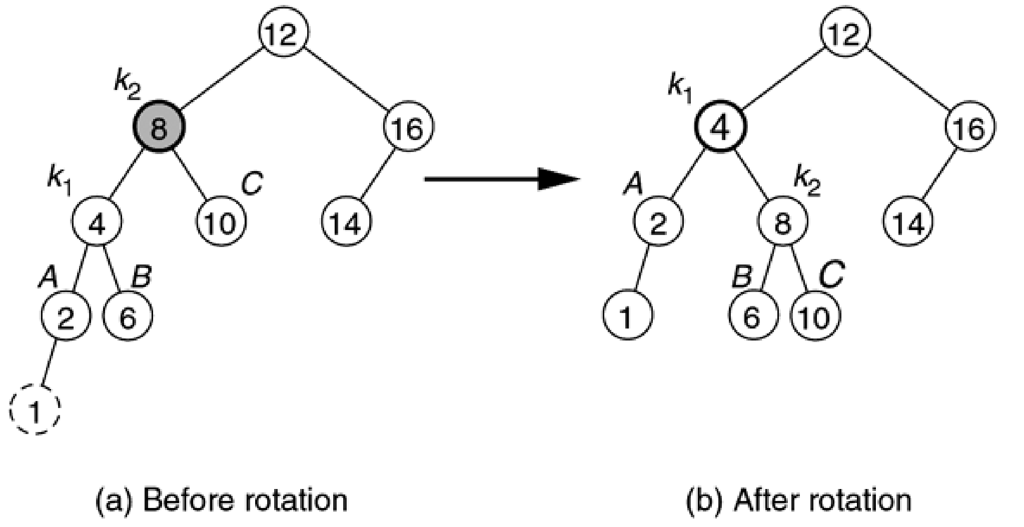


figure 19.25

Single rotation fixes an AVL tree after insertion of 1.

Single rotation: Case 4 (Symmetric of case 1)

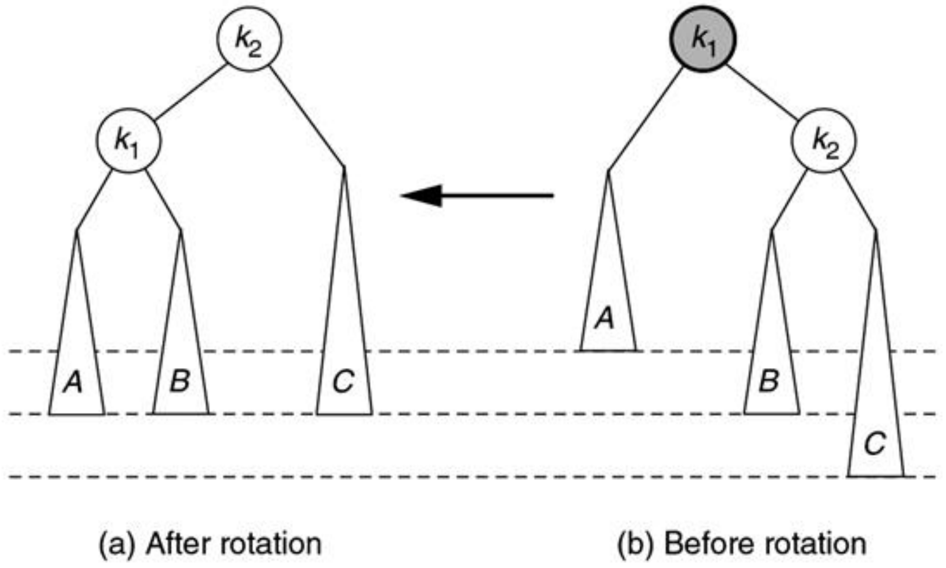


figure 19.26

Symmetric single rotation to fix case 4

Single rotation: Case 4

```
/**
 * Rotate binary tree node with right child.
 * For AVL trees, this is a single rotation for case 4.
 */
static BinaryNode rotateWithRightChild( BinaryNode k1 )
{
    BinaryNode k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    return k2;
}
```

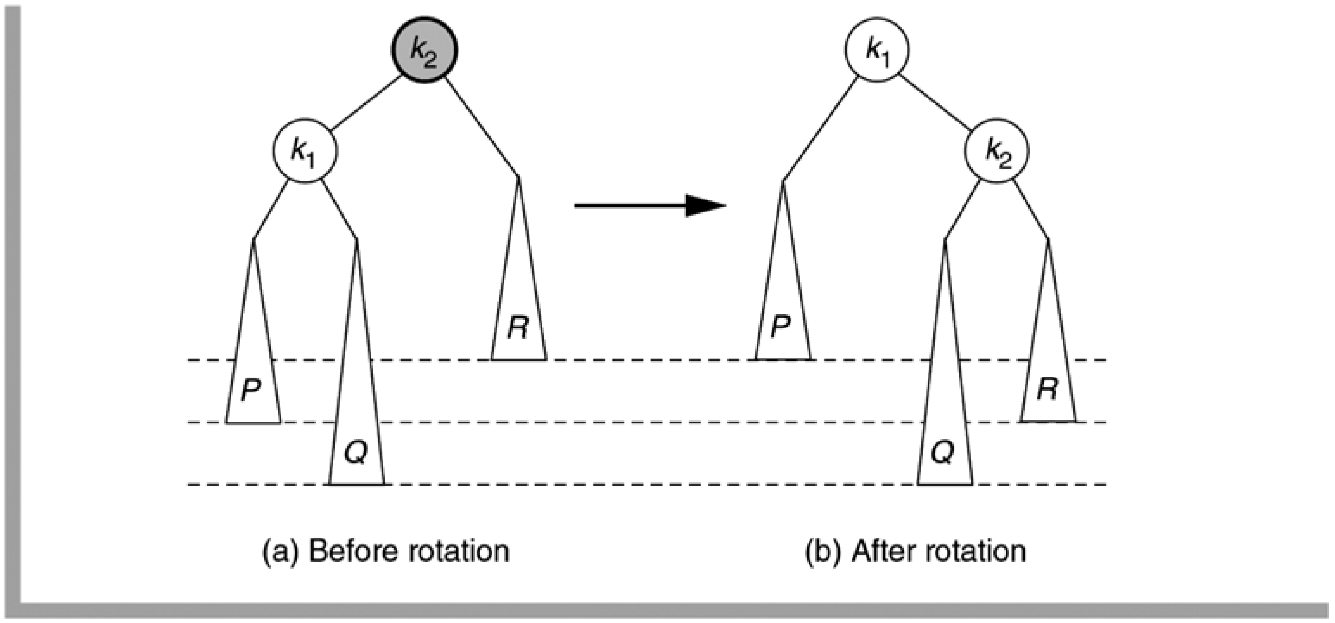
figure 19.27

Pseudocode for a single rotation (case 4)

Case 2

figure 19.28

Single rotation does not fix case 2.



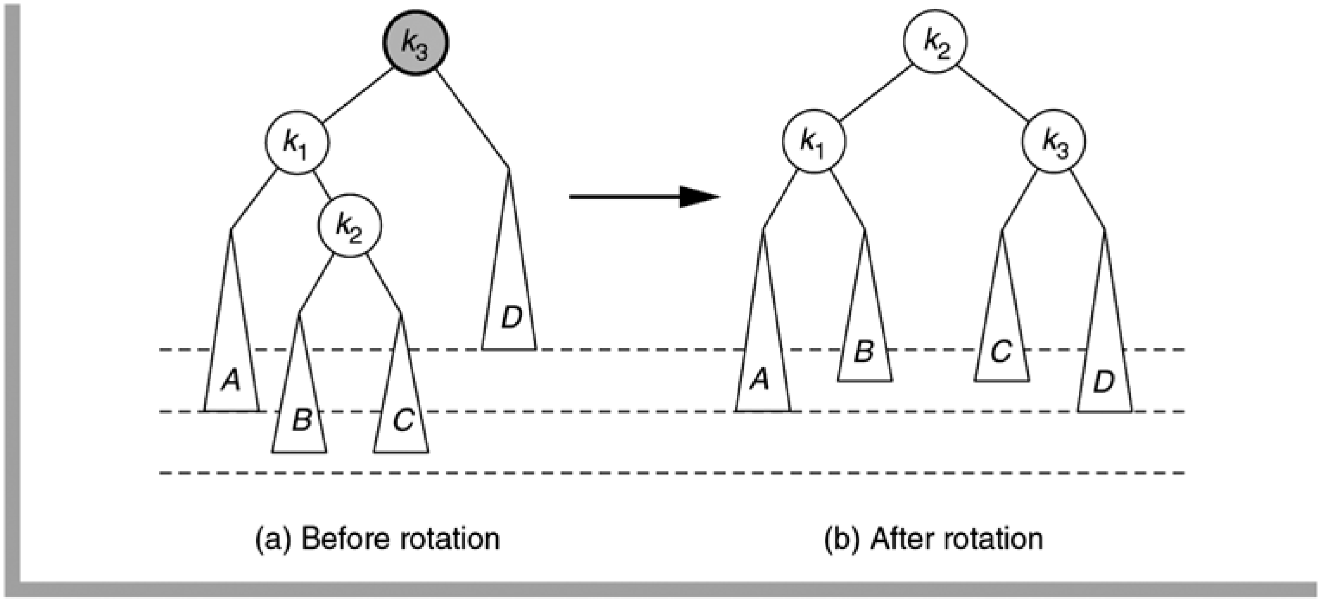
Cases:

2. An insertion in the right subtree of the left child of X
3. An insertion in the left subtree of the right child of X

Left-right double rotation

figure 19.29

Left–right double rotation to fix case 2



Left-right double rotation

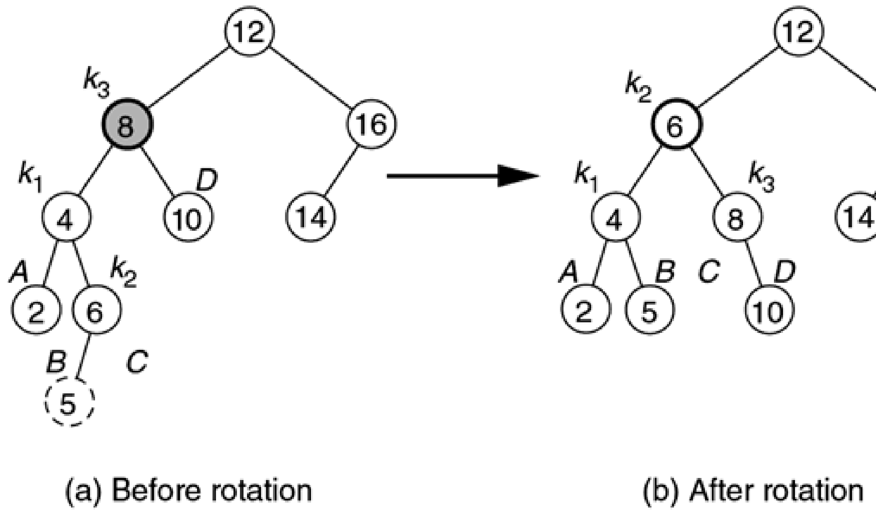


figure 19.30

Double rotation fixes AVL tree after the insertion of 5.

Double rotation operations

- Note that, although a double rotation appears complex, it turns out to be equivalent to the following sequence:
  - A rotation between X's child and grandchild
    - Ex. Rotation between 4 and 6
  - A rotation between X and its new child
    - Ex. Rotation between 6 and 8

Implementation of double rotation: case 2

figure 19.32

Pseudocode for a double rotation (case 2)

```
/**
 * Double rotate binary tree node: first left child
 * with its right child; then node k3 with new left child.
 * For AVL trees, this is a double rotation for case 2.
 */
static BinaryNode doubleRotateWithLeftChild( BinaryNode k3 )
{
    k3.left = rotateWithRightChild( k3.left );
    return rotateWithLeftChild( k3 );
}
```

Right-Left double rotation for case 3

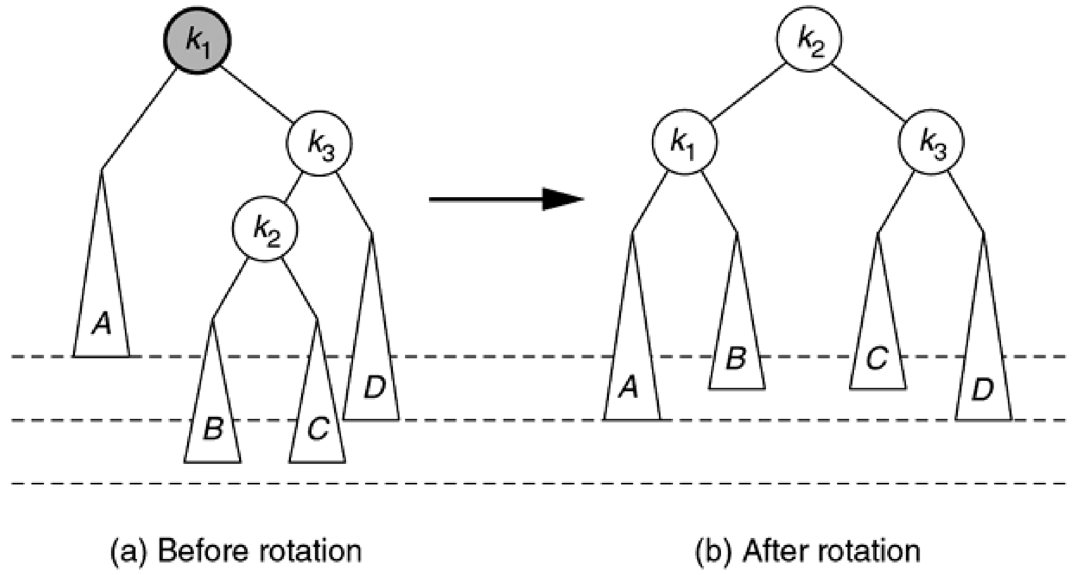


figure 19.31

Right–Left double rotation to fix case 3.

Implementation of double rotation: case 3

figure 19.33

Pseudocode for a double rotation (case 3)

```
/**
 * Double rotate binary tree node: first right child
 * with its left child; then node k1 with new right child.
 * For AVL trees, this is a double rotation for case 3.
 */
static BinaryNode doubleRotateWithRightChild( BinaryNode k1 )
{
    k1.right = rotateWithLeftChild( k1.right );
    return rotateWithRightChild( k1 );
}
```

Project: Part 2

- Implement a binary search tree to allow duplicates.
  - Hint: Have each node store a data structure of items that are considered duplicates (using the first item in this structure) to control branching.
- In addition to the existing operators of the tree, implement also the following:
  - Return number of duplicates of an element
  - Find and replace all duplicates of an element A with element B
  - Show all the tree together with the duplicates (pre-order, in-order, post-order, level-order)
  - Remove all existing duplicates (leaving only one copy) of an element
  - Remove only one copy of the existing duplicates of an element
  - Remove all existing duplicates (leaving only one copy) for all the duplicated elements in tree
  - Print all elements that have duplicates together with the number of duplicates
  - Print the number of all the duplicates in the tree
  - Show only the nodes that have/do not have duplicates

Project Part 2 Deadline

- Zip the project into a .zip document
- Rename the file in “Name Surname Project Part2.zip”
- Submit the file by email to: marenglenbiba@unyt.edu.al.
- Mail Subject: DSSPRING17 – Project Part 2 - Name Surname
- Deadline 28/06/2017 23:59.
- 10% penalty if the above rules are not respected as written.
- 10% penalty for each day of delay

End of Class

- Readings
  - Chapter 19 – Sections 19.1 – 19.4