Implementations II

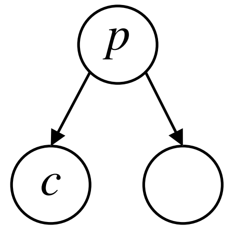
Agenda

- **Trees**
  - Terminology
  - Binary trees
  - Tree traversal

- **Binary search trees**
  - The basic binary search tree
  - Balanced binary search trees
    - AVL-trees
    - Red-black trees
    - AA-trees

- **B-trees**

Trees

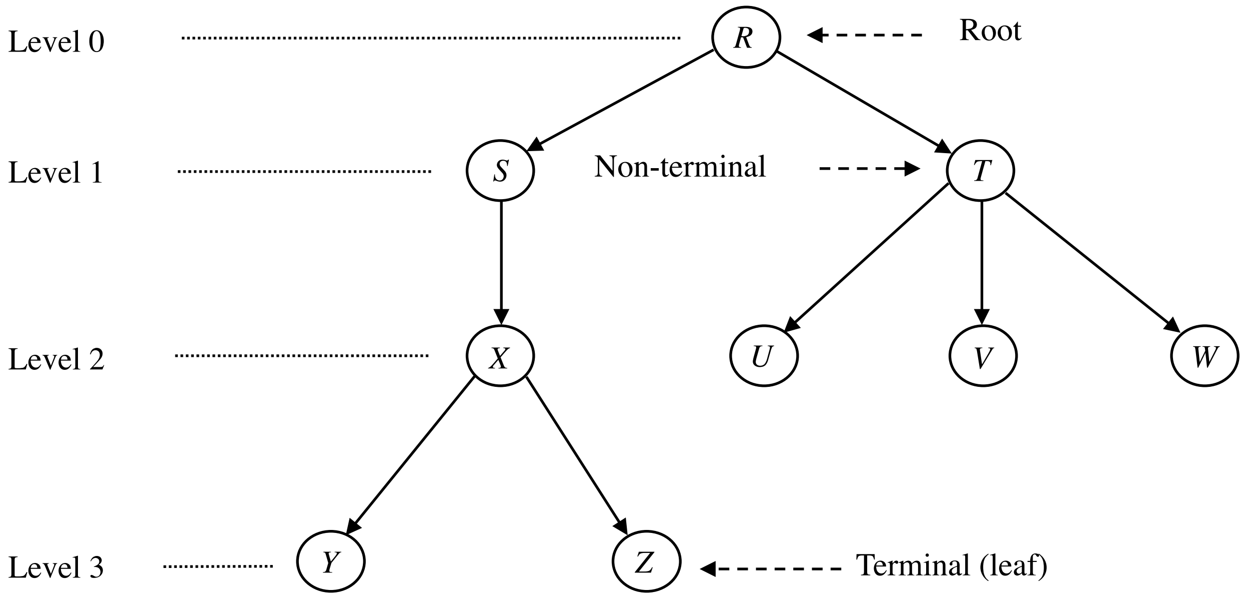


Non-recursive definition of a tree

A tree consists of a set of **nodes** and a set of directed **edges** that connect pairs of nodes. A **rooted tree** has the following properties:

- One node is distinguished as the *root*.
- Every node *c*, except the root, is connected by an edge from exactly one other node *p*.
  Node *p* is *c*'s *parent*, and *c* is one of *p*'s *children*.
- A unique path traverses from the root to each node.
  The number of edges that must be followed is the *path length*.

Terminology

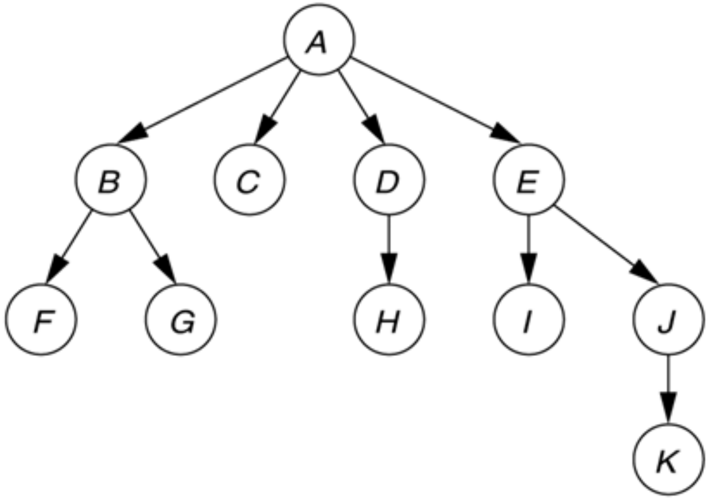


Root: *R*
X is a **parent** of *Y*
Y is a **child** of *X*
U, *V*, and *W* are **children** of *T*
S is a **grandparent** of *Z*
S is an **ancestor** of *Y*
Y is a **descendent** of *S*
Terminal nodes (**leaves**): *Y, Z, U, V, W*
Non-terminal nodes: *R, S, X, T*

Height, depth, and size

figure 18.1

A tree, with height and depth information



Node	Height	Depth	Size
A	3	0	11
B	1	1	3
C	0	1	1
D	1	1	2
E	2	1	4
F	0	2	1
G	0	2	1
H	0	2	1
I	0	2	1
J	1	2	1
K	0	3	1

Leaf: a node that has no children
Height of a node: length of the path from the node to the deepest leaf
Depth of a node: length of the path from the root to the node
Size of a node: Number of descendants the node has (including the node itself)

Recursive definition of a tree

Either a tree T is empty or it consists of a root and zero or more nonempty subtrees T_1, T_2, ..., T_k, each of whose roots are connect by an edge from the root of T.

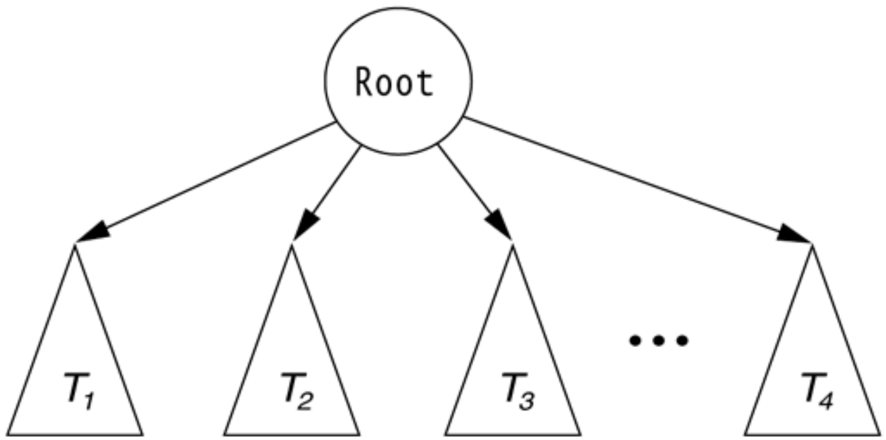


figure 18.2

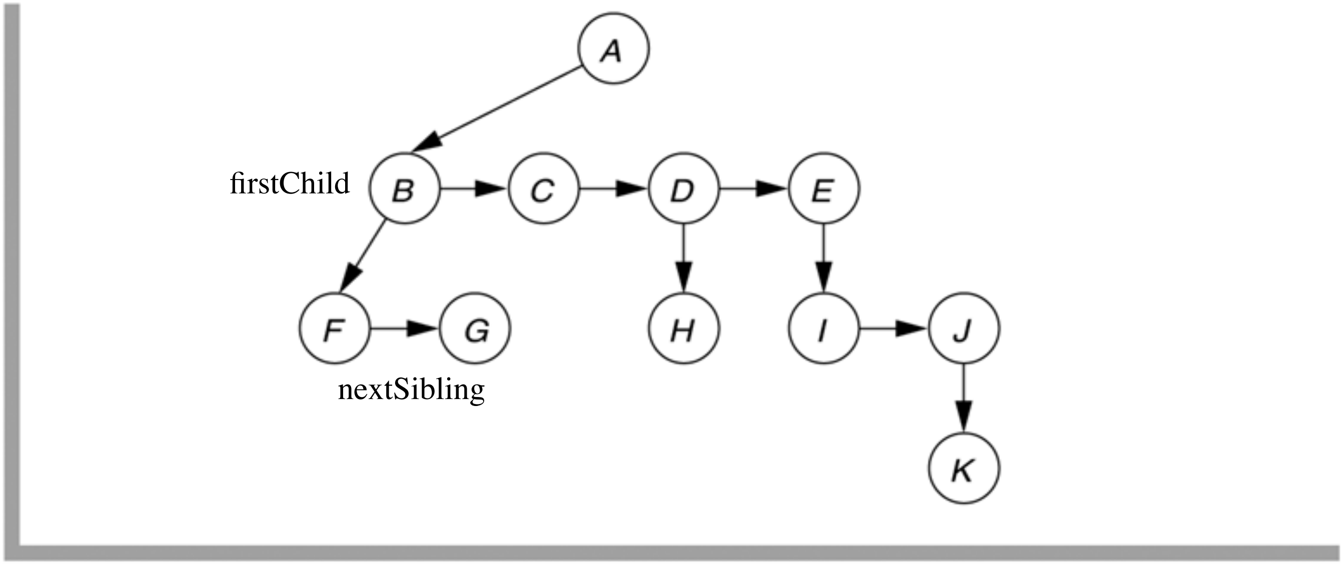
A tree viewed recursively

In certain cases (most notably, the *binary trees*) we may allow some subtrees to be empty.

Implementation

figure 18.3

First child/next sibling representation of the tree in Figure 18.1

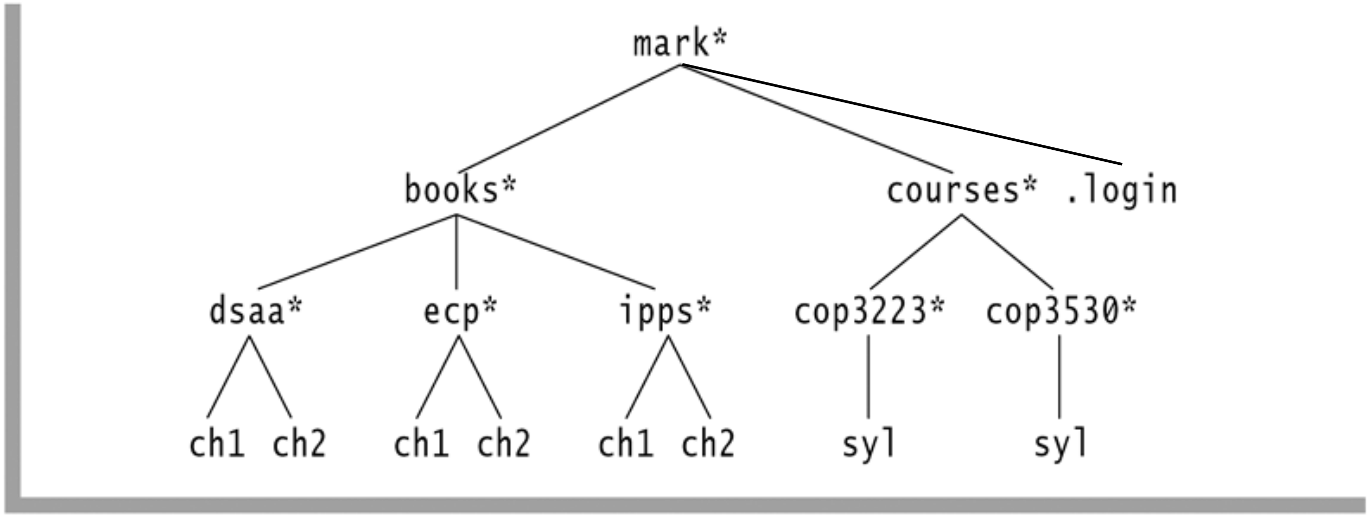



```
class Node {
    Node firstChild, nextSibling;
}
```

An application: file systems

figure 18.4

A Unix directory



Listing a directory and its subdirectories

```
void listAll( int depth = 0 ) // depth is initially 0
{
    printName( depth );      // Print the name of the object
    if( isDirectory( ) )
        for each file c in this directory (for each child)
            c.listAll( depth + 1 );
}
```

figure 18.5

A routine for listing a directory and its subdirectories in a hierarchical file system

pseudocode

`printName(depth)` prints the name of the object indented by `depth` tab characters

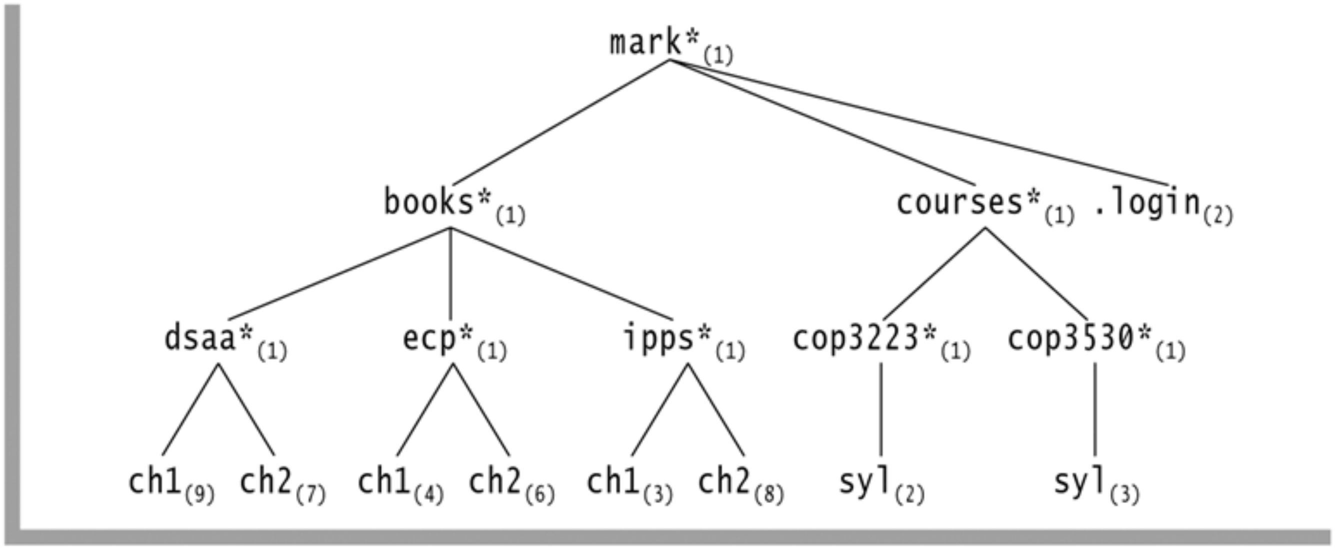
```
mark
      books
            dsaa
                    ch1
                    ch2
            ecp
                    ch1
                    ch2
            ipps
                    ch1
                    ch2
      courses
            cop3223
                    syl
            cop3530
                    syl
      .login
```

figure 18.6

The directory listing for the tree shown in Figure 18.4

figure 18.7

The Unix directory with file sizes



File size is measured in number of blocks
Typical block size: 4 KB

Calculating the total size of all files

```
int size( )
{
    int totalSize = sizeOfThisFile( );

    if( isDirectory( ) )
        for each file c in this directory (for each child)
            totalSize += c.size( );

    return totalSize;
}
```

figure 18.8

A routine for calculating the total size of all files in a directory

```
                ch1                        9
                ch2                        7
            dsaa                          17
                ch1                        4
                ch2                        6
            ecp                           11
                ch1                        3
                ch2                        8
            ipps                          12
        books                             41
                syl                        2
            cop3223                        3
                syl                        3
            cop3530                        4
        courses                            8
        .login                             2
mark                                      52
```

figure 18.9

A trace of the `size` method

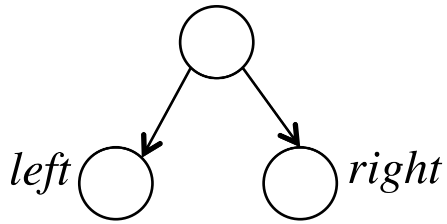
```java
import java.io.File;

public class FileSystem extends File
{
        // Constructor
    public FileSystem( String name )
    {
        super( name );
    }

        // Output file name with indentation
    public void printName( int depth )
    {
        for( int i = 0; i < depth; i++ )
            System.out.print( "\t" );
        System.out.println( getName( ) );
    }

        // Public driver to list all files in directory
    public void listAll( )
    {
        listAll( 0 );
    }

        // Recursive method to list all files in directory
    private void listAll( int depth )
    {
        printName( depth );

        if( isDirectory( ) )
        {
            String [ ] entries = list( );

            for( String entry : entries)
            {
                FileSystem child = new FileSystem( getPath( )
                             + separatorChar + entry );
                child.listAll( depth + 1 );
            }
        }
    }

        // Simple main to list all files in current directory
    public static void main( String [ ] args )
    {
        FileSystem f = new FileSystem( "." );
        f.ListAll( );
    }
}
```

figure 18.10

Java implementation for a directory listing

`getName`, isDirectory, list, and getPath are methods in `java.io.File`

`separatorChar`, defined in `java.io.File`, is the system-dependent name-separator character



Binary trees

A **binary tree** is a tree in which no node has more than two children

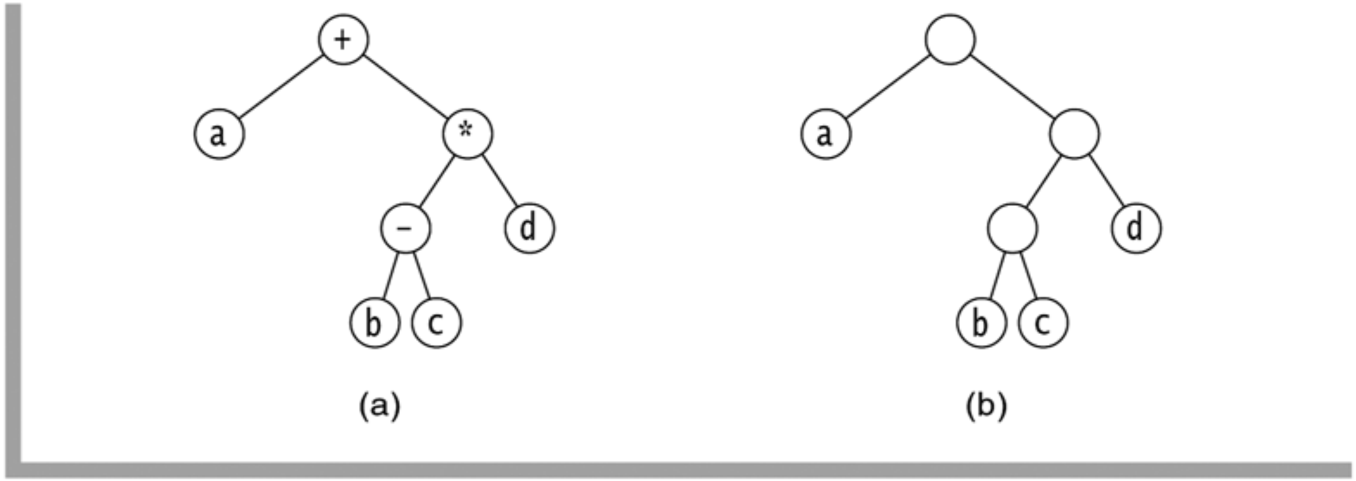
We name the children of a node *left* and *right*

Recursively, a binary tree is either empty or consists of a root, a left binary tree, and a right binary tree.

Uses of binary trees

figure 18.11

Uses of binary trees: (a) an expression tree and (b) a Huffman coding tree



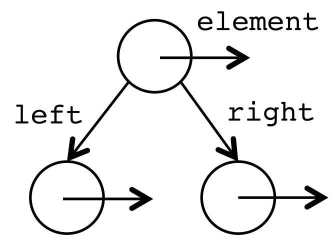
a + (b − c) * d

a: 0
b: 100
c: 101
d: 11

```java
// BinaryNode class; stores a node in a tree.
//
// CONSTRUCTION: with no parameters, or an Object,
//     left child, and right child.
//
// *******************PUBLIC OPERATIONS**********************
// int size( )           --> Return size of subtree at node
// int height( )         --> Return height of subtree at node
// void printPostOrder( ) --> Print a postorder tree traversal
// void printInOrder( )   --> Print an inorder tree traversal
// void printPreOrder( )  --> Print a preorder tree traversal
// BinaryNode duplicate( )--> Return a duplicate tree

class BinaryNode<AnyType>
{
    public BinaryNode( )
      { this( null, null, null ); }
    public BinaryNode( AnyType theElement,
                       BinaryNode<AnyType> lt, BinaryNode<AnyType> rt )
      { element = theElement; left = lt; right = rt; }

    public AnyType getElement( )
      { return element; }
    public BinaryNode<AnyType> getLeft( )
      { return left; }
    public BinaryNode<AnyType> getRight( )
      { return right; }
    public void setElement( AnyType x )
      { element = x; }
    public void setLeft( BinaryNode<AnyType> t )
      { left = t; }
    public void setRight( BinaryNode<AnyType> t )
      { right = t; }

    public static <AnyType> int size( BinaryNode<AnyType> t )
      { /* Figure 18.19 */ }
    public static <AnyType> int height( BinaryNode<AnyType> t )
      { /* Figure 18.21 */ }
    public BinaryNode<AnyType> duplicate( )
      { /* Figure 18.17 */ }

    public void printPreOrder( )
      { /* Figure 18.22 */ }
    public void printPostOrder( )
      { /* Figure 18.22 */ }
    public void printInOrder( )
      { /* Figure 18.22 */ }

    private AnyType             element;
    private BinaryNode<AnyType> left;
    private BinaryNode<AnyType> right;
}
```

figure 18.12

The `BinaryNode` class skeleton



static methods
`t` may be `null`

figure 18.13

The `BinaryTree` class, except for `merge`

```
// BinaryTree class; stores a binary tree.
//
// CONSTRUCTION: with (a) no parameters or (b) an object to
//    be placed in the root of a one-element tree.
//
// ********************PUBLIC OPERATIONS***********************
// Various tree traversals, size, height, isEmpty, makeEmpty.
// Also, the following tricky method:
// void merge( Object root, BinaryTree t1, BinaryTree t2 )
//                          --> Construct a new tree
// ********************ERRORS**********************************
// Error message printed for illegal merges.

public class BinaryTree<AnyType>
{
    public BinaryTree( )
      { root = null; }
    public BinaryTree( AnyType rootItem )
      { root = new BinaryNode<AnyType>( rootItem, null, null ); }

    public BinaryNode<AnyType> getRoot( )
      { return root; }
    public int size( )
      { return BinaryNode.size( root ); }
    public int height( )
      { return BinaryNode.height( root ); }

    public void printPreOrder( )
      { if( root != null ) root.printPreOrder( ); }
    public void printInOrder( )
      { if( root != null ) root.printInOrder( ); }
    public void printPostOrder( )
      { if( root != null ) root.printPostOrder( ); }

    public void makeEmpty( )
      { root = null; }
    public boolean isEmpty( )
      { return root == null; }

    public void merge( AnyType rootItem,
                       BinaryTree<AnyType> t1, BinaryTree<AnyType> t2 )
      { /* Figure 18.16 */ }

    private BinaryNode<AnyType> root;
}
```

Creates a new tree, with `rootItem` at the root, and `t1` and `t2` as left and right subtrees

Naive implementation of the `merge` operation

`root = new BinaryNode(rootItem, t1.root, t2.root)`

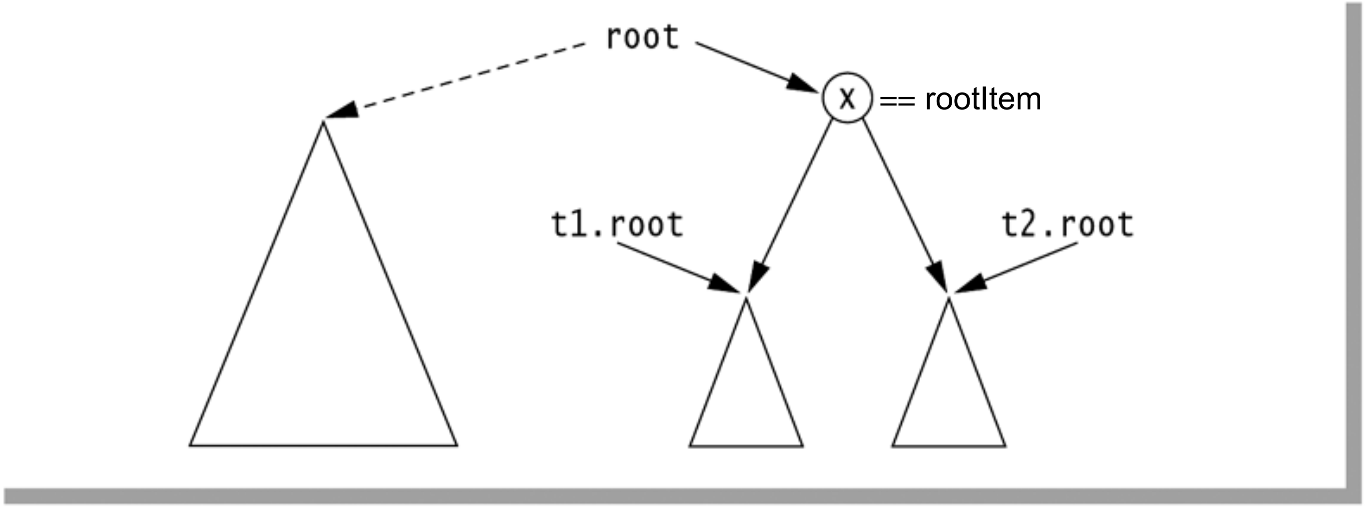


figure 18.14

Result of a naive `merge` operation: Subtrees are shared.

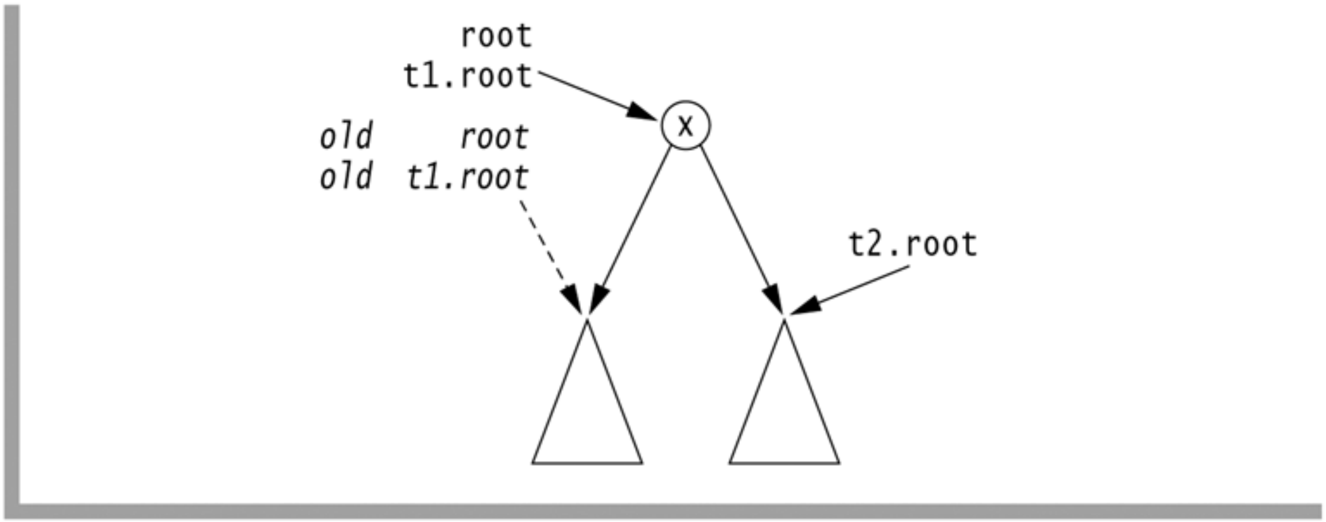
Nodes in `t1` and `t2`'s trees are now in two trees (their original trees and the merged result). This is a problem if we want to remove or otherwise alter subtrees.

Solution: set `t1.root` and `t2.root` to `null`.

t1.merge(x, t1, t2)

figure 18.15

Aliasing problems in the `merge` operation; `t1` is also the current object.



`t1` is an alias for the current object (`this`).
If we execute `t1.root = null`, we change `this.root` to `null`, too.

Solution: check for aliasing (`this == t1` and `this == t2`)

```java
/**
 * Merge routine for BinaryTree class.
 * Forms a new tree from rootItem, t1 and t2.
 * Does not allow t1 and t2 to be the same.
 * Correctly handles other aliasing conditions.
 */
public void merge( AnyType rootItem,
                   BinaryTree<AnyType> t1, BinaryTree<AnyType> t2 )
{
    if( t1.root == t2.root && t1.root != null )
        throw new IllegalArgumentException( );

        // Allocate new node
    root = new BinaryNode<AnyType>( rootItem, t1.root, t2.root );

        // Ensure that every node is in one tree
    if( this != t1 )
        t1.root = null;
    if( this != t2 )
        t2.root = null;
}
```

figure 18.16

The `merge` routine for the `BinaryTree` class

Copying a tree

```java
/**
 * Return a reference to a node that is the root of a
 * duplicate of the binary tree rooted at the current node.
 */
public BinaryNode<AnyType> duplicate( )
{
    BinaryNode<AnyType> root =
               new BinaryNode<AnyType>( element, null, null );

    if( left != null )              // If there's a left subtree
        root.left = left.duplicate( );    // Duplicate; attach
    if( right != null )            // If there's a right subtree
        root.right = right.duplicate( );  // Duplicate; attach
    return root;                         // Return resulting tree
}
```

figure 18.17

A routine for returning a copy of the tree rooted at the current node

Calculating the size of a tree

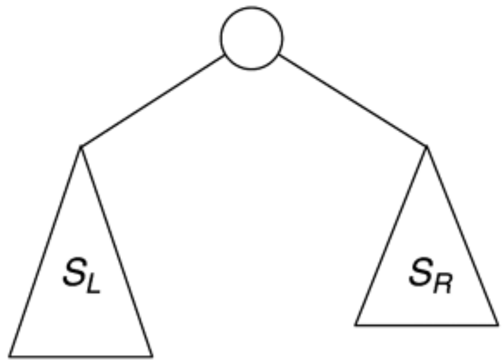


figure 18.18

Recursive view used to calculate the size of a tree: $S_T = S_L + S_R + 1$.

figure 18.19

A routine for computing the size of a node

```
/**
 * Return the size of the binary tree rooted at t.
 */
public static <AnyType> int size( BinaryNode<AnyType> t )
{
    if( t == null )
        return 0;
    else
        return 1 + size( t.left ) + size( t.right );
}
```

Calculating the height of a tree

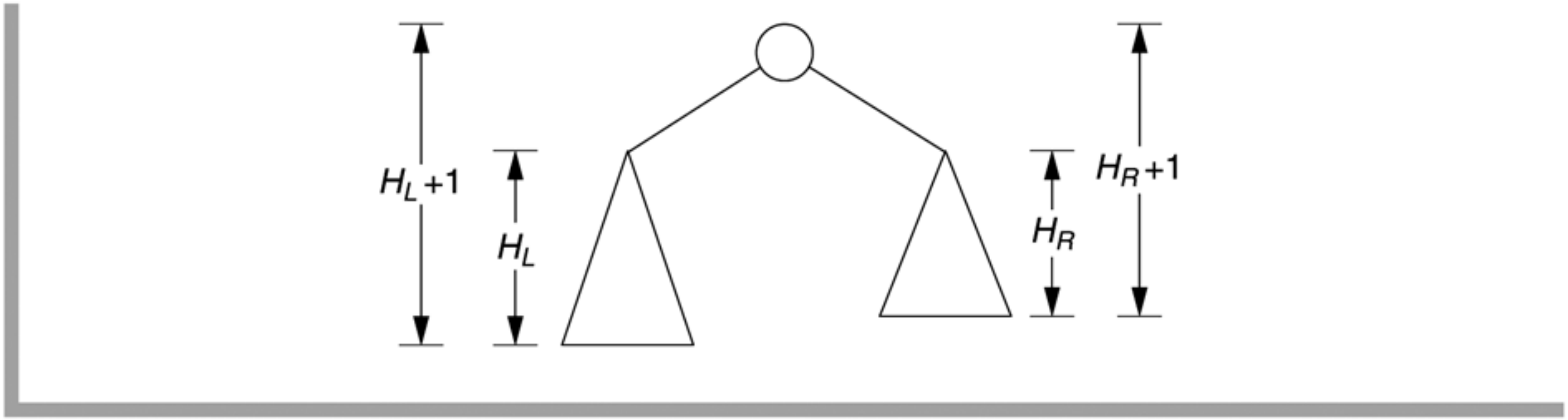


figure 18.20

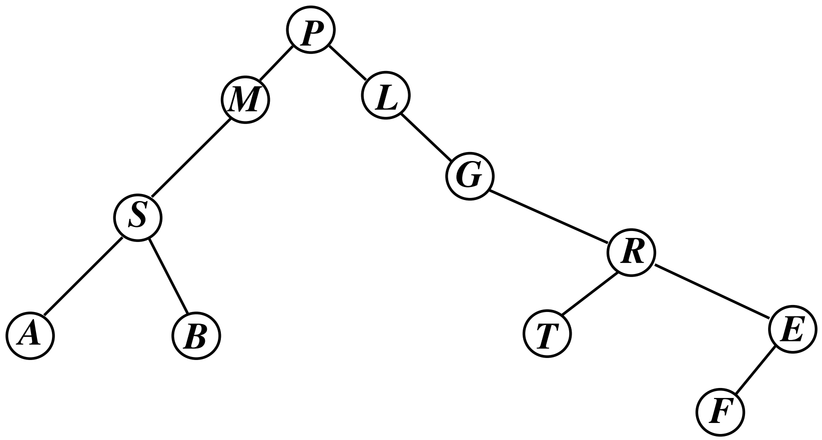
Recursive view of the node height calculation:
$H_T = \text{Max}\,(H_L + 1, H_R + 1)$

```java
/**
 * Return the height of the binary tree rooted at t.
 */
public static <AnyType> int height( BinaryNode<AnyType> t )
{
    if( t == null )
        return -1;
    else
        return 1 + Math.max( height( t.left ), height( t.right ) );
}
```

figure 18.21

A routine for computing the height of a node

Recursive traversal of binary trees



(1) **Preorder**: Visit the root. Visit the left subtree. Visit the right subtree.

P M S A B L G R T E F

(2) **Inorder**: Visit the left subtree. Visit the root. Visit the right subtree.

A S B M P L G T R F E

(3) **Postorder**: Visit the left subtree. Visit the right subtree. Visit the root.

A B S M T F E R G L P

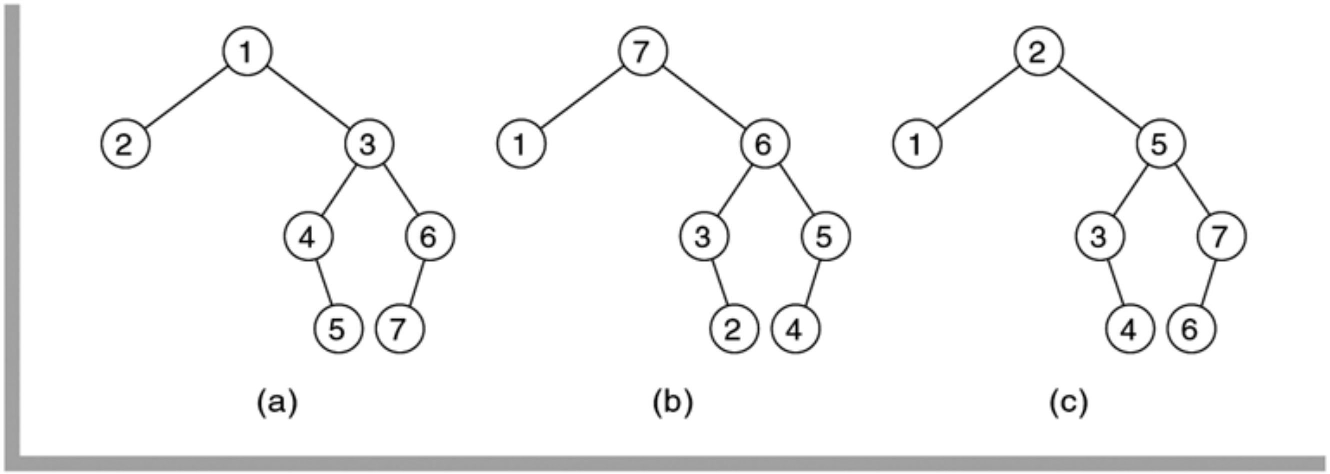
figure 18.22

Routines for printing nodes in preorder, postorder, and inorder

```java
// Print tree rooted at current node using preorder traversal.
public void printPreOrder( )
{
    System.out.println( element );       // Node
    if( left != null )
        left.printPreOrder( );           // Left
    if( right != null )
        right.printPreOrder( );          // Right
}

// Print tree rooted at current node using postorder traversal.
public void printPostOrder( )
{
    if( left != null )                   // Left
        left.printPostOrder( );
    if( right != null )                  // Right
        right.printPostOrder( );
    System.out.println( element );       // Node
}

// Print tree rooted at current node using inorder traversal.
public void printInOrder( )
{
    if( left != null )                   // Left
        left.printInOrder( );
    System.out.println( element );       // Node
    if( right != null )
        right.printInOrder( );           // Right
}
```

figure 18.23

(a) Preorder,
(b) postorder, and
(c) inorder visitation
routes



TreeIterator

abstract class

```
import java.util.NoSuchElementException;

// TreeIterator class; maintains "current position"
//
// CONSTRUCTION: with tree to which iterator is bound
//
// ******************PUBLIC OPERATIONS*********************
//     first and advance are abstract; others are final
// boolean isValid( )    --> True if at valid position in tree
// AnyType retrieve( )   --> Return item in current position
// void first( )         --> Set current position to first
// void advance( )       --> Advance (prefix)
// ******************ERRORS*******************************
// Exceptions thrown for illegal access or advance

abstract class TreeIterator<AnyType>
{
    /**
     * Construct the iterator. The current position is set to null.
     * @param theTree the tree to which the iterator is bound.
     */
    public TreeIterator( BinaryTree<AnyType> theTree )
      { t = theTree; current = null; }

    /**
     * Test if current position references a valid tree item.
     * @return true if the current position is not null; false otherwise.
     */
    final public boolean isValid( )
      { return current != null; }

    /**
     * Return the item stored in the current position.
     * @return the stored item.
     * @exception NoSuchElementException if the current position is invalid.
     */
    final public AnyType retrieve( )
    {
        if( current == null )
            throw new NoSuchElementException( );
        return current.getElement( );
    }

    abstract public void first( );
    abstract public void advance( );

    protected BinaryTree<AnyType> t;          // The tree root
    protected BinaryNode<AnyType> current;    // The current position
}
```

abstract methods

// The binary tree to be traversed

figure 18.24

The tree iterator abstract base class

Subclasses of TreeIterator

```
class PreOrder<AnyType>  extends TreeIterator<AnyType>

class InOrder<AnyType>   extends TreeIterator<AnyType>

class PostOrder<AnyType> extends TreeIterator<AnyType>
```

Example of use:

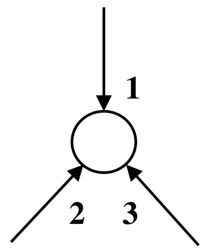
```
TreeIterator<Integer> itr = new InOrderIterator<>(theTree);
for (itr.first(); itr.isValid(); itr.advance())
    System.out.print(itr.retrieve() + " ");
```

Traversals implemented by using a stack

(of nodes not yet ready to be visited)

The type of traversal is determined by how many times a node is popped from the stack before it is visited:

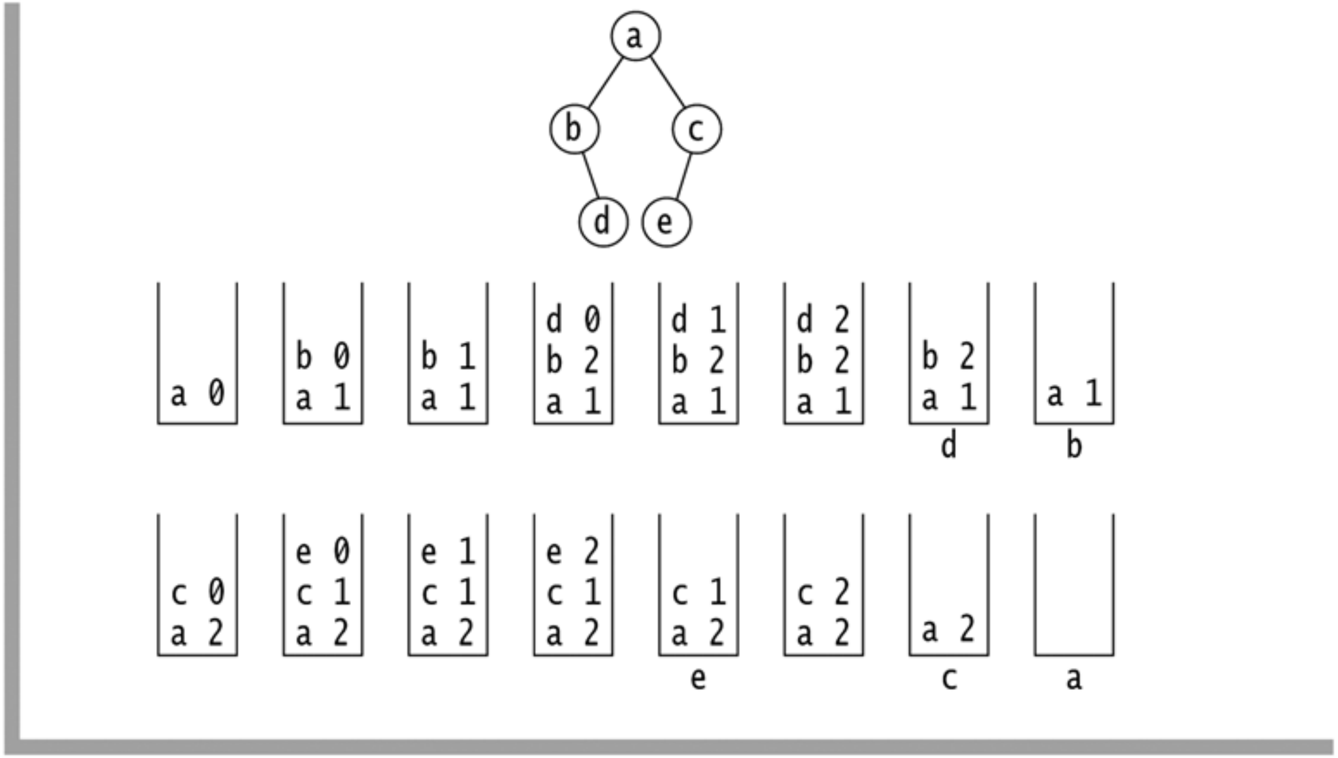
Once:	preorder
Twice:	inorder
Thrice:	postorder




```
class StNode<AnyType> {
    StNode(BinaryNode<AnyType> n)
      { node = n; timesPopped = 0; }

    BinaryNode<AnyType> node;
    int timesPopped;
}
```

figure 18.25

Stack states during postorder traversal



A node is popped thrice before it is visited

```
import weiss.nonstandard.Stack;
import weiss.nonstandard.ArrayStack;

// PostOrder class; maintains "current position"
//     according to a postorder traversal
//
// CONSTRUCTION: with tree to which iterator is bound
//
// ******************PUBLIC OPERATIONS**********************
// boolean isValid( )   --> True if at valid position in tree
// AnyType retrieve( )  --> Return item in current position
// void first( )        --> Set current position to first
// void advance( )      --> Advance (prefix)
// ******************ERRORS*********************************
// Exceptions thrown for illegal access or advance

class PostOrder<AnyType> extends TreeIterator<AnyType>
{
    protected static class StNode<AnyType>
    {
        StNode( BinaryNode<AnyType> n )
          { node = n; timesPopped = 0; }
        BinaryNode<AnyType> node;
        int timesPopped;
    }

    /**
     * Construct the iterator. The current position is set to null.
     */
    public PostOrder( BinaryTree<AnyType> theTree )
    {
        super( theTree );
        s = new ArrayStack<StNode<AnyType>>( );
        s.push( new StNode<AnyType>( t.getRoot( ) ) );
    }

    /**
     * Set the current position to the first item.
     */
    public void first( )
    {
        s.makeEmpty( );
        if( t.getRoot( ) != null )
        {
            s.push( new StNode<AnyType>( t.getRoot( ) ) );
            advance( );
        }
    }

    protected Stack<StNode<AnyType>> s; // The stack of StNode objects
}
```

figure 18.26

The PostOrder class (complete class except for advance)

Postorder `advance`

```java
/**
 * Advance the current position to the next node in the tree,
 *     according to the postorder traversal scheme.
 * @throws NoSuchElementException if the current position is null.
 */
public void advance( )
{
    if( s.isEmpty( ) )
    {
        if( current == null )
            throw new NoSuchElementException( );
        current = null;
        return;
    }

    StNode<AnyType> cnode;

    for( ; ; )
    {
        cnode = s.topAndPop( );

        if( ++cnode.timesPopped == 3 )
        {
            current = cnode.node;
            return;
        }

        s.push( cnode );
        if( cnode.timesPopped == 1 )
        {
            if( cnode.node.getLeft( ) != null )
                s.push( new StNode<AnyType>( cnode.node.getLeft( ) ) );
        }
        else  // cnode.timesPopped == 2
        {
            if( cnode.node.getRight( ) != null )
                s.push( new StNode<AnyType>( cnode.node.getRight( ) ) );
        }
    }
}
```

// A node is popped thrice before it is visited

figure 18.27

The `advance` routine for the `PostOrder` iterator class

Inorder `advance`

```java
// InOrder class; maintains "current position"
//     according to an inorder traversal
//
// CONSTRUCTION: with tree to which iterator is bound
//
// ******************PUBLIC OPERATIONS**********************
// Same as TreeIterator
// ******************ERRORS*********************************
// Exceptions thrown for illegal access or advance

class InOrder<AnyType> extends PostOrder<AnyType>
{
    public InOrder( BinaryTree<AnyType> theTree )
      { super( theTree ); }

    /**
     * Advance the current position to the next node in the tree,
     *     according to the inorder traversal scheme.
     * @throws NoSuchElementException if iteration has
     *     been exhausted prior to the call.
     */
    public void advance( )
    {
        if( s.isEmpty( ) )
        {
            if( current == null )
                throw new NoSuchElementException( );
            current = null;
            return;
        }

        StNode<AnyType> cnode;
        for( ; ; )
        {
            cnode = s.topAndPop( );

            if( ++cnode.timesPopped == 2 )
            {
                current = cnode.node;
                if( cnode.node.getRight( ) != null )
                    s.push( new StNode<AnyType>( cnode.node.getRight( ) ) );
                return;
            }
                // First time through
            s.push( cnode );
            if( cnode.node.getLeft( ) != null )
                s.push( new StNode<AnyType>( cnode.node.getLeft( ) ) );
        }
    }
}
```

// A node is popped twice before it is visited

figure 18.28

The complete `InOrder` iterator class

```java
// PreOrder class; maintains "current position"
//
// CONSTRUCTION: with tree to which iterator is bound
//
// ******************PUBLIC OPERATIONS**********************
// boolean isValid( )   --> True if at valid position in tree
// AnyType retrieve( )  --> Return item in current position
// void first( )        --> Set current position to first
// void advance( )      --> Advance (prefix)
// ******************ERRORS*********************************
// Exceptions thrown for illegal access or advance

class PreOrder<AnyType> extends TreeIterator<AnyType>
{
    /**
     * Construct the iterator. The current position is set to null.
     */
    public PreOrder( BinaryTree<AnyType> theTree )
    {
        super( theTree );
        s = new ArrayStack<BinaryNode<AnyType>>( );
        s.push( t.getRoot( ) );
    }

    /**
     * Set the current position to the first item, according
     * to the preorder traversal scheme.
     */
    public void first( )
    {
        s.makeEmpty( );
        if( t.getRoot( ) != null )
        {
            s.push( t.getRoot( ) );
            advance( );
        }
    }

    public void advance( )
      { /* Figure 18.30 */ }

    private Stack<BinaryNode<AnyType>> s; // Stack of BinaryNode objects
}
```

figure 18.29

The `PreOrder` class skeleton and all members except `advance`

Preorder `advance`

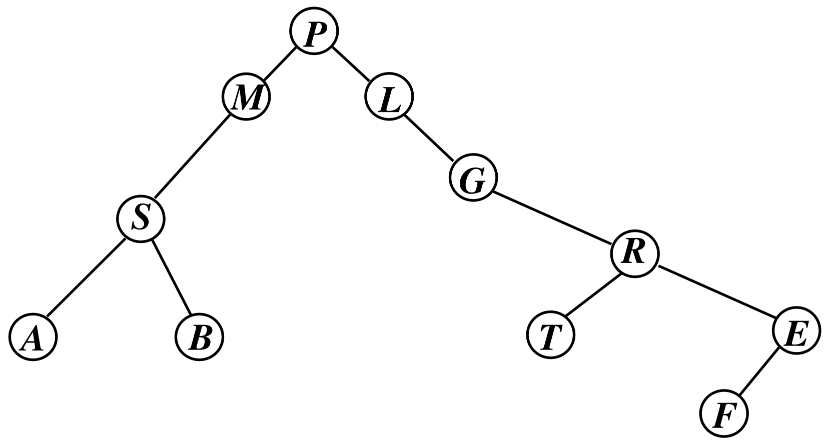
figure 18.30

The `PreOrder` iterator class `advance` routine

```
/**
 * Advance the current position to the next node in the tree,
 *     according to the preorder traversal scheme.
 * @throws NoSuchElementException if iteration has
 *     been exhausted prior to the call.
 */
public void advance( )
{
    if( s.isEmpty( ) )
    {
        if( current == null )
            throw new NoSuchElementException( );
        current = null;
        return;
    }

    current = s.topAndPop( );

    if( current.getRight( ) != null )
        s.push( current.getRight( ) );
    if( current.getLeft( ) != null )
        s.push( current.getLeft( ) );
}
```

We need no longer maintain a count of the number of times an object has been popped. Note the order: The right child is pushed onto the stack before the left child.

Level-order traversal



(4) **Level-order**: Visit the nodes starting at the root and going from top to bottom, left to right.

P M L S G A B R T E F

Implemented using a queue of nodes.
Note that the queue can get very large! Possibly, *N*/2 objects.

```
// LevelOrder class; maintains "current position"
//     according to a level-order traversal
//
// CONSTRUCTION: with tree to which iterator is bound
//
// ******************PUBLIC OPERATIONS**********************
// boolean isValid( )   --> True if at valid position in tree
// AnyType retrieve( )  --> Return item in current position
// void first( )        --> Set current position to first
// void advance( )      --> Advance (prefix)
// ******************ERRORS*********************************
// Exceptions thrown for illegal access or advance

class LevelOrder<AnyType> extends TreeIterator<AnyType>
{
    /**
     * Construct the iterator.
     */
    public LevelOrder( BinaryTree<AnyType> theTree )
    {
        super( theTree );
        q = new ArrayQueue<BinaryNode<AnyType>>( );
        q.enqueue( t.getRoot( ) );
    }

    public void first( )
      { /* Figure 18.32 */ }

    public void advance( )
      { /* Figure 18.32 */ }

    private Queue<BinaryNode<AnyType>> q; // Queue of BinaryNode objects
}
```

figure 18.31

The `LevelOrder` iterator class skeleton

figure 18.32

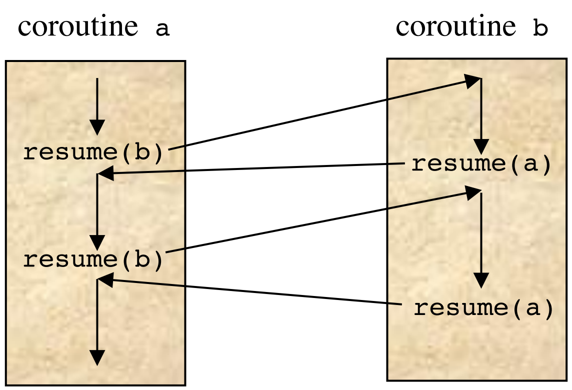
The `first` and `advance` routines for the `LevelOrder` iterator class

```java
/**
 * Set the current position to the first item, according
 * to the level-order traversal scheme.
 */
public void first( )
{
    q.makeEmpty( );
    if( t.getRoot( ) != null )
    {
        q.enqueue( t.getRoot( ) );
        advance( );
    }
}

/**
 * Advance the current position to the next node in the tree,
 *     according to the level-order traversal scheme.
 * @throws NoSuchElementException if iteration has
 *     been exhausted prior to the call.
 */
public void advance( )
{
    if( q.isEmpty( ) )
    {
        if( current == null )
            throw new NoSuchElementException( );
        current = null;
        return;
    }

    current = q.dequeue( );

    if( current.getLeft( ) != null )
        q.enqueue( current.getLeft( ) );
    if( current.getRight( ) != null )
        q.enqueue( current.getRight( ) );
}
```

Traversals implemented by using coroutines

A **coroutine** is a routine that may temporarily suspend itself.
In the meantime other coroutines may be executed.
A suspended coroutine may later be resumed at the point where it was suspended.

This form of sequencing is called *alternation*.



Class **Coroutine**

implemented by Keld Helsgaun

```
public class abstract Coroutine {
    protected abstract void body();

    public static void resume(Coroutine c);
    public static void call(Coroutine c);
    public static void detach();
}
```

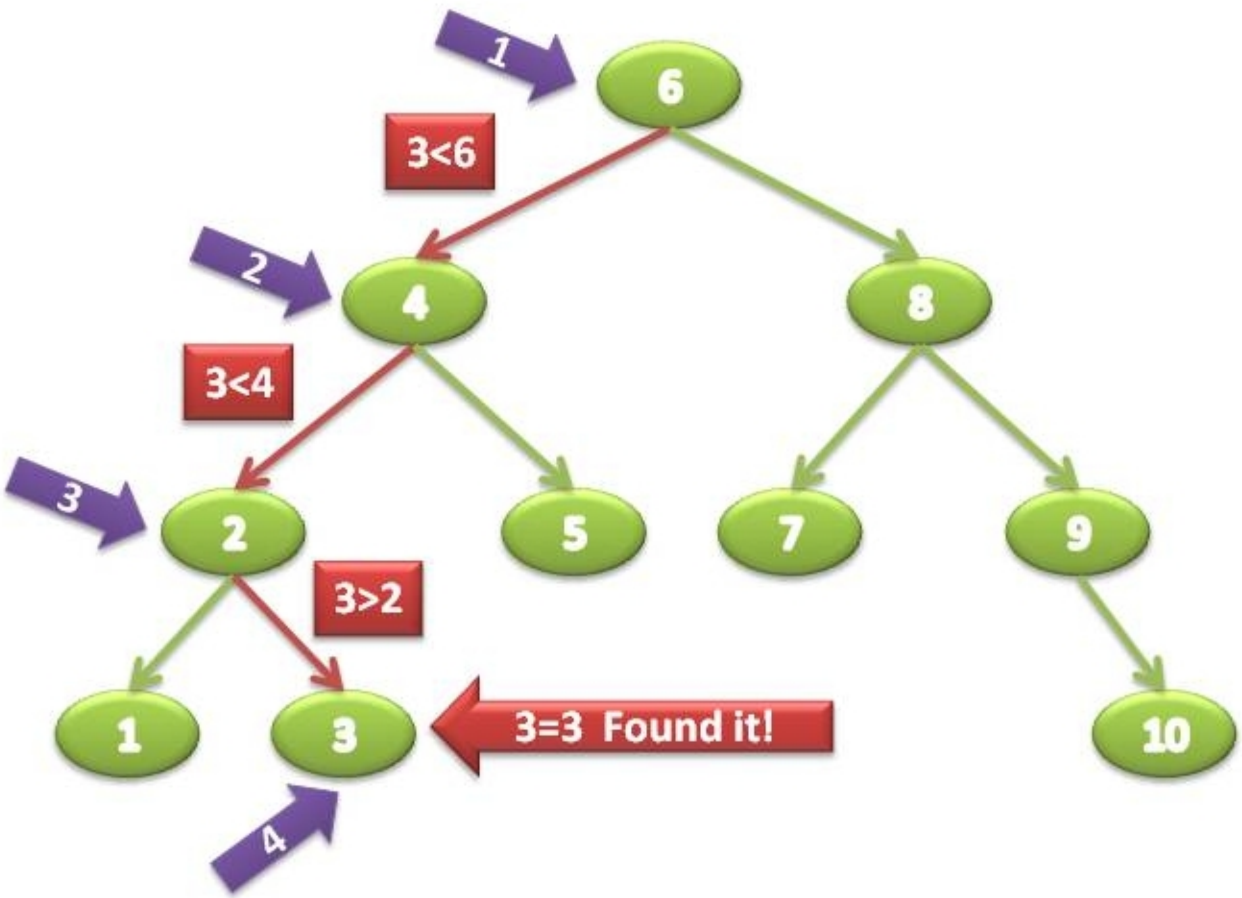
Recursive inorder traversal by using coroutine sequencing

```
public class InOrderIterator<T> extends TreeIterator<T> {
    public InOrderIterator(BinaryTree<T> tree) { super(tree); }

    @Override void traverse(BinaryNode<T> t) {
        if (t != null) {
            traverse(t.left);
            current = t;
            detach();
            traverse(t.right);
        } else
            current = null;
    }
}
```

```
public abstract class TreeIterator<T> extends Coroutine
```

```
public abstract class TreeIterator<T> extends Coroutine {
    public TreeIterator(BinaryTree<T> theTree) {
        t = theTree; current = null;
    }

    abstract void traverse(BinaryNode<T> n);

    protected void body() { traverse(t.root); }

    public void first() { call(this); }

    public boolean isValid() { return current != null; }

    public void advance() {
        if (current == null)
            throw new NoSuchElementExcption();
        call(this);
    }

    public T retrieve() {
        if (current == null)
            throw new NoSuchElementExcption();
        return current.element;
    }

    protected BinaryTree<T> t;
    protected BinaryNode<T> current;
}
```

Binary search trees



Binary search

Requires that the input array is **sorted**

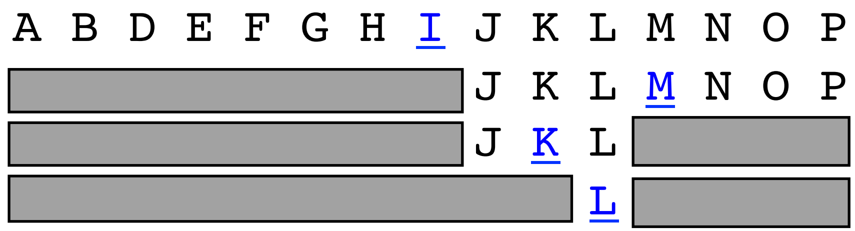
Algorithm:

Split the array into two parts of (almost) equal size

Determine which part may contain the search item.

Continue the search in this part in the same fashion.

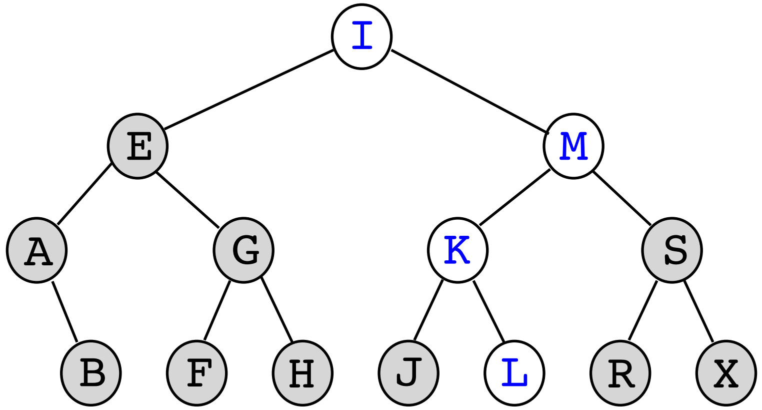
Example: Searching for **`L`**.



Worst-case running time is $O(\log N)$.
Average-case running time is $O(\log N)$.

Binary search tree

Binary search may be described using a **binary search tree**:



A **binary search tree** is a binary tree that satisfies the *search order property*: for every node *X* in the tree, all keys in the left subtrees are *smaller* than the key in *X*, and all keys in the right subtree are *larger* than the key in *X*.

`find`

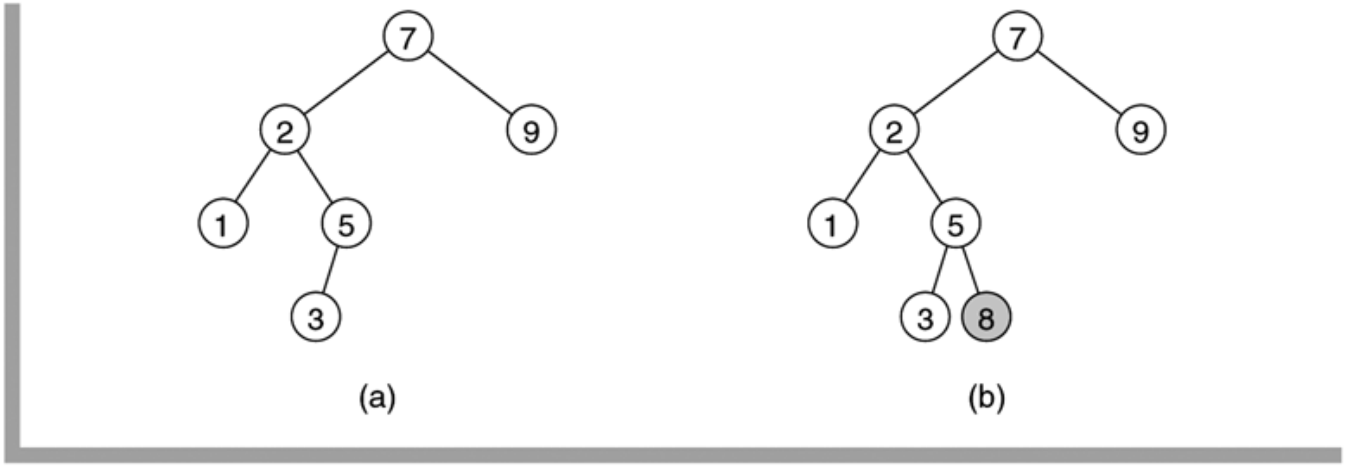


figure 19.1

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

The `find` operation is performed by repeatedly branching either left or right, depending on the result of the comparison.
The `findMin` operation is performed by following left nodes as long as there is a left child. The `findMax` operation is similar.

insert

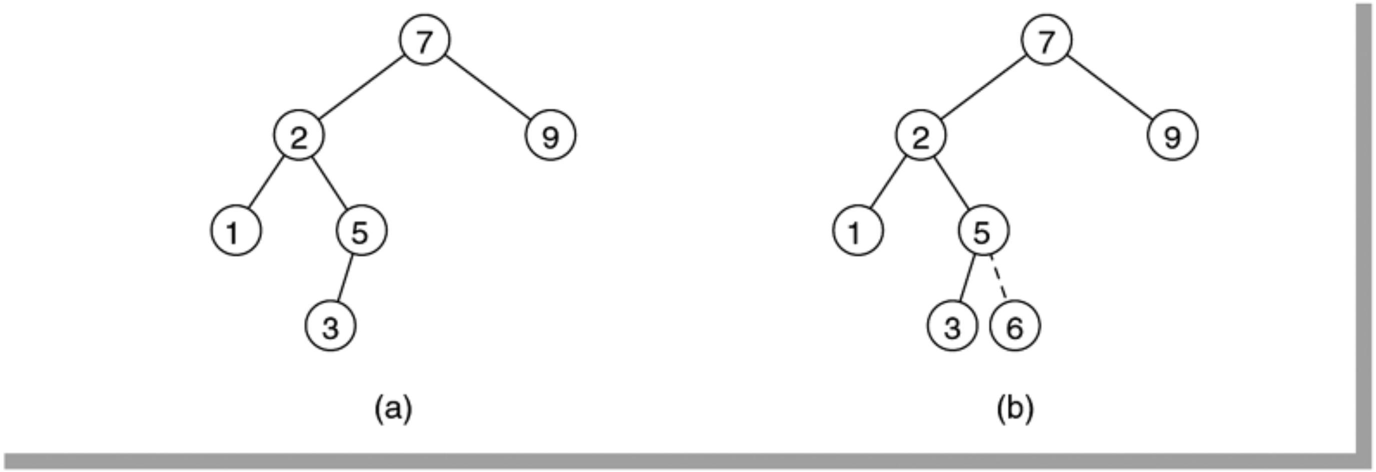


figure 19.2

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

The `insert` operation can be performed by inserting a node at the point at which an unsuccessful search terminated.

remove

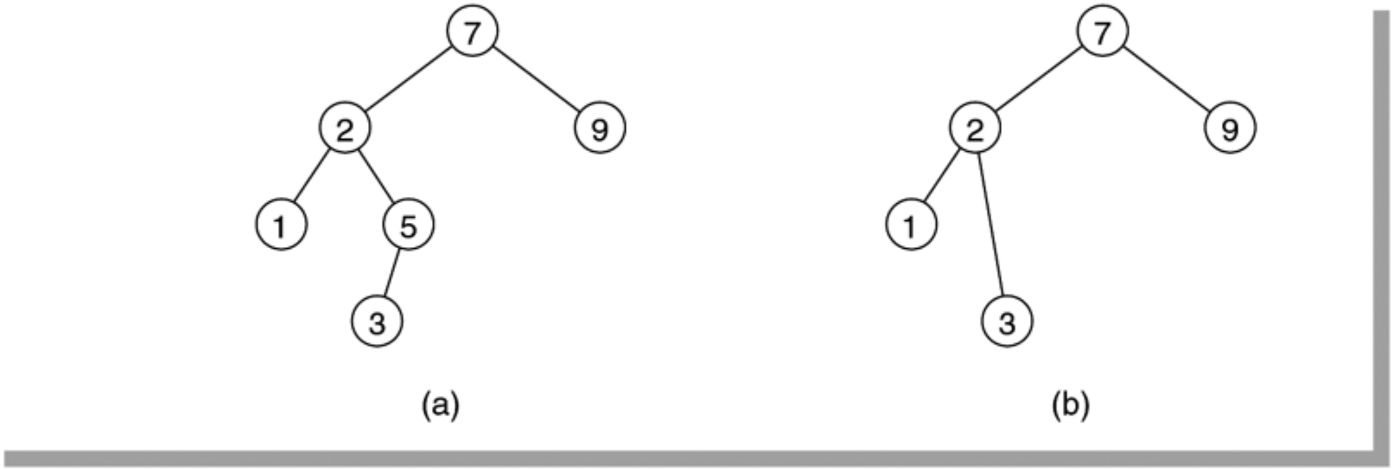


figure 19.3

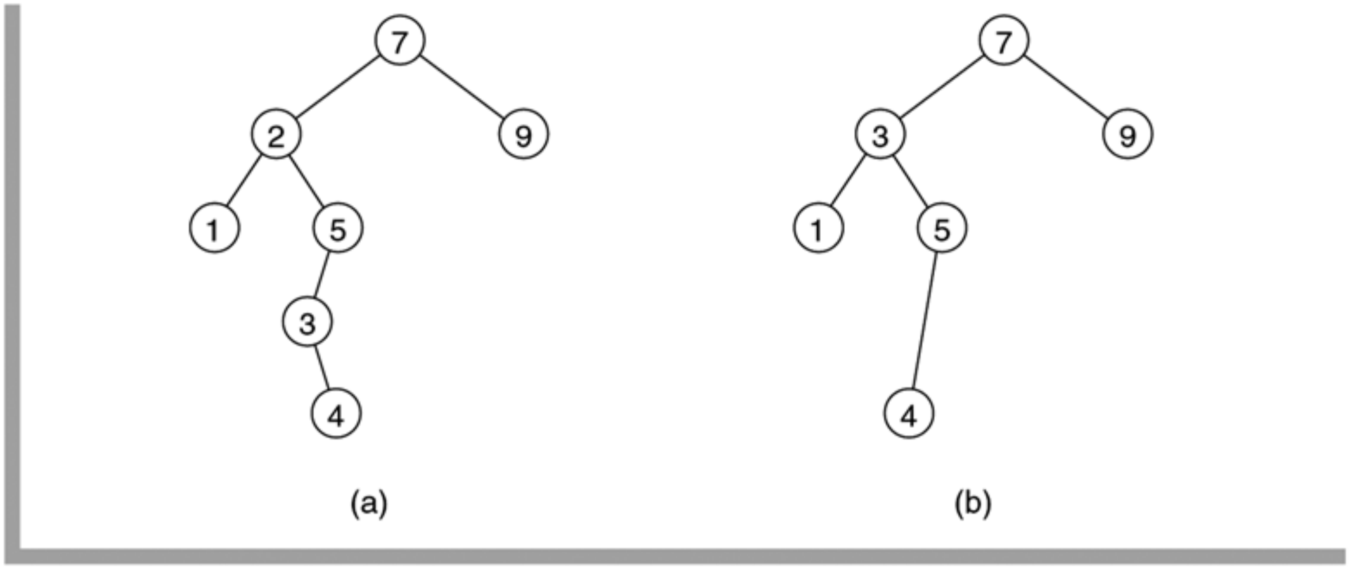
Deletion of node 5 with one child: (a) before and (b) after

If the node to be deleted is a leaf, it can be deleted immediately.
If the node has only one child, we adjust the parent's child link to bypass the node.

remove

figure 19.4

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



If the node has two children, replace the item in this node with the smallest item in the right subtree and then remove that node. The second remove is easy.

```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 ); }
```

overloaded

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*)

```java
/**
 * 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

```java
/**
 * 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

figure 19.9

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

```
/**
 * 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;
}
```

```java
/**
 * 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

figure 19.11

The `removeMin` method for the `BinarySearchTree` class

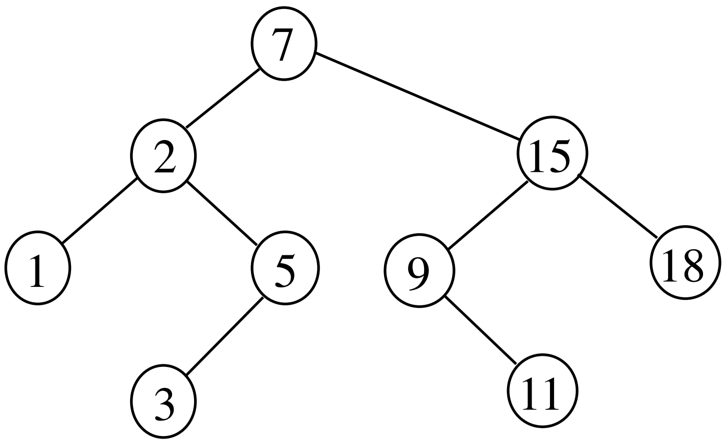
```
/**
 * Internal method to remove minimum item from a subtree.
 * @param t the node that roots the tree.
 * @return the new root.
 * @throws ItemNotFoundException if t is empty.
 */
protected BinaryNode<AnyType> removeMin( BinaryNode<AnyType> t )
{
    if( t == null )
        throw new ItemNotFoundException(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

```java
/**
 * 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

Printing the elements in sorted order



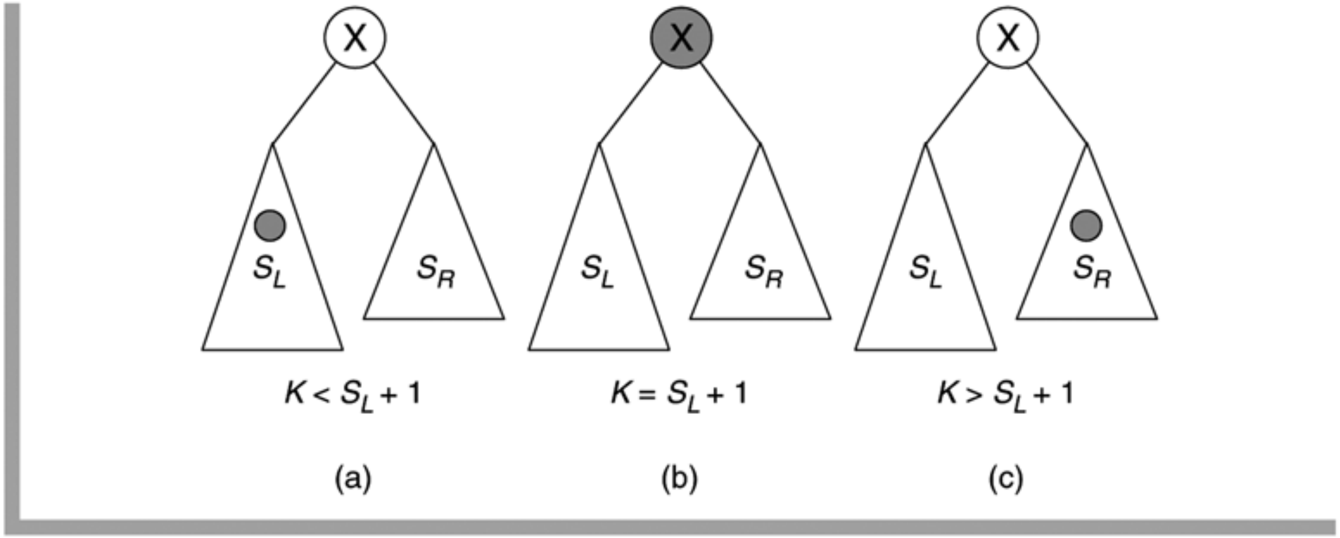
Inorder traversal of the tree

```
void printSorted(BinaryNode t) {
    if (t != null) {
        printSorted(t.left);
        System.out.println(t.element);
        printSorted(t.right);
    }
}
```

findKth

figure 19.13

Using the `size` data member to implement `findKth`



```
package weiss.nonstandard;

// BinarySearchTreeWithRank class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// Comparable findKth( k )--> Return kth smallest item
// All other operations are inherited
// ******************ERRORS********************************
// IllegalArgumentException thrown if k is out of bounds

public class BinarySearchTreeWithRank<AnyType extends Comparable<? super AnyType>>
                    extends BinarySearchTree<AnyType>
{
    private static class BinaryNodeWithSize<AnyType> extends BinaryNode<AnyType>
    {
        BinaryNodeWithSize( AnyType x )
          { super( x ); size = 0; }

        int size;
    }

    /**
     * Find the kth smallest item in the tree.
     * @param k the desired rank (1 is the smallest item).
     * @return the kth smallest item in the tree.
     * @throws IllegalArgumentException if k is less
     *     than 1 or more than the size of the subtree.
     */
    public AnyType findKth( int k )
      { return findKth( k, root ).element; }

    protected BinaryNode<AnyType> findKth( int k, BinaryNode<AnyType> t )
      { /* Figure 19.15 */ }
    protected BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> tt )
      { /* Figure 19.16 */ }
    protected BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> tt )
      { /* Figure 19.18 */ }
    protected BinaryNode<AnyType> removeMin( BinaryNode<AnyType> tt )
      { /* Figure 19.17 */ }
}
```

figure 19.14

The `BinarySearchTreeWithRank` class skeleton

```java
/**
 * Internal method to find kth smallest item in a subtree.
 * @param k the desired rank (1 is the smallest item).
 * @return the node containing the kth smallest item in the subtree.
 * @throws IllegalArgumentException if k is less
 *     than 1 or more than the size of the subtree.
 */
protected BinaryNode<AnyType> findKth( int k, BinaryNode<AnyType> t )
{
    if( t == null )
        throw new IllegalArgumentException( );
    int leftSize = ( t.left != null ) ?
                   ((BinaryNodeWithSize<AnyType>) t.left).size : 0;

    if( k <= leftSize )
        return findKth( k, t.left );
    if( k == leftSize + 1 )
        return t;
    return findKth( k - leftSize - 1, t.right );
}
```

figure 19.15

The `findKth` operation for a search tree with order statistics

```java
/**
 * Internal method to insert into a subtree.
 * @param x the item to insert.
 * @param tt 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> tt )
{
    BinaryNodeWithSize<AnyType> t = (BinaryNodeWithSize<AnyType>) tt;

    if( t == null )
        t = new BinaryNodeWithSize<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( ) );
    t.size++;
    return t;
}
```

figure 19.16

The `insert` operation for a search tree with order statistics

```java
/**
 * Internal method to remove the smallest item from a subtree,
 *     adjusting size fields as appropriate.
 * @param t the node that roots the tree.
 * @return the new root.
 * @throws ItemNotFoundException if the subtree is empty.
 */
protected BinaryNode<AnyType> removeMin( BinaryNode<AnyType> tt )
{
    BinaryNodeWithSize<AnyType> t = (BinaryNodeWithSize<AnyType>) tt;

    if( t == null )
        throw new ItemNotFoundException( );
    if( t.left == null )
        return t.right;

    t.left = removeMin( t.left );
    t.size--;
    return t;
}
```

figure 19.17

The `removeMin` operation for a search tree with order statistics

```java
/**
 * 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> tt )
{
    BinaryNodeWithSize<AnyType> t = (BinaryNodeWithSize<AnyType>) tt;

    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
        return ( t.left != null ) ? t.left : t.right;

    t.size--;
    return t;
}
```

figure 19.18

The `remove` operation for a search tree with order statistics

Complexity

The cost of `insert`, `find`, and `remove` is proportional to the number of nodes accessed. This number depends upon the structure of the tree.

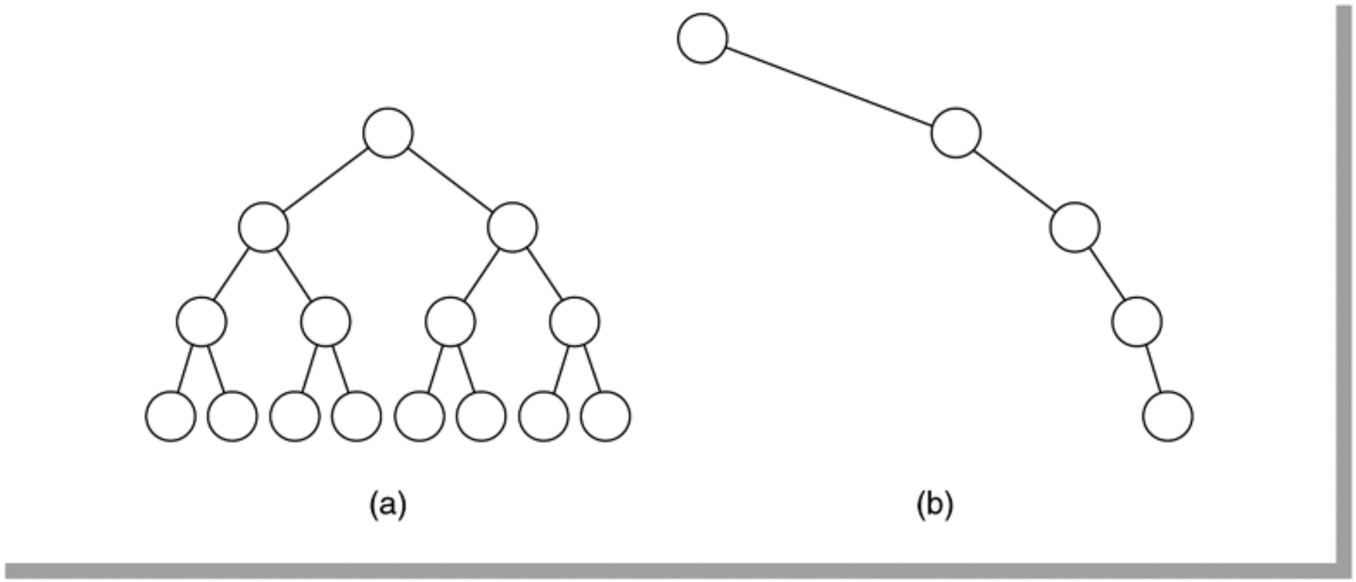


figure 19.19

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

Best case
(Perfectly balanced tree)

Worst case
(Linear list)

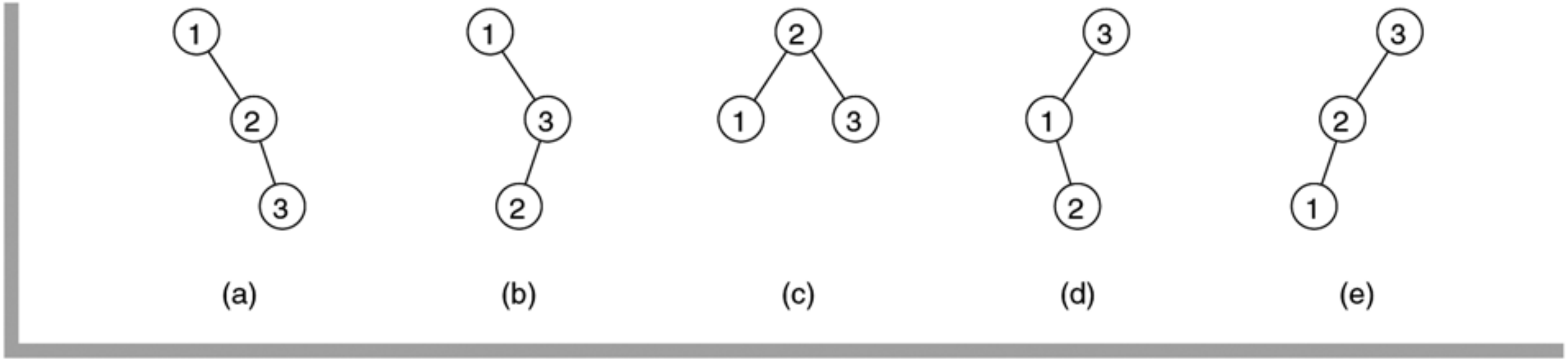


figure 19.20

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.

The tree with root 2 is formed from either the insertion sequence (2, 3, 1) or the sequence (2, 1, 3).

Average case complexity

If N elements are inserted in random order in an initially empty binary search tree, then the average search path length is $1.38 \log_2 N$.

Note that the worst case occurs when the elements are inserted in sorted order.

Balanced binary search trees

Balancing is a technique that **guarantees** that the worst cases do not occur.

The idea is to reorganize the tree during insertion and deletion so that it becomes *perfectly balanced* (i.e., the two subtrees of every node contains the same number of nodes), or almost perfectly balanced.

The following presents some data structures and algorithms that guarantee $O(\log N)$ running time for search, insertion and deletion.

Building a perfectly balanced binary search tree from an array

```
BinarySearchTree buildTree(Comparable[] a) {
    Arrays.sort(a);
    BinarySearchTree bst = new BinarySearchTree();
    bst.root =  buildTree(a, 0, a.length - 1);
    return bst;
}
```

```
BinaryNode buildTree(Comparable[] a, int low, int high) {
    if (low > high)
       return null;
    int mid = (low + high) / 2;
    return new BinaryNode(a[mid], buildTree(a, low, mid - 1),
                                  buildTree(a, mid + 1, high));
 }
```

AVL trees

(**A**delson-**V**elskii and **L**andis, 1962)

An AVL tree is a binary search tree that satisfies the condition:

> For any node in the tree, the height of the left and right subtrees can differ by at most 1.

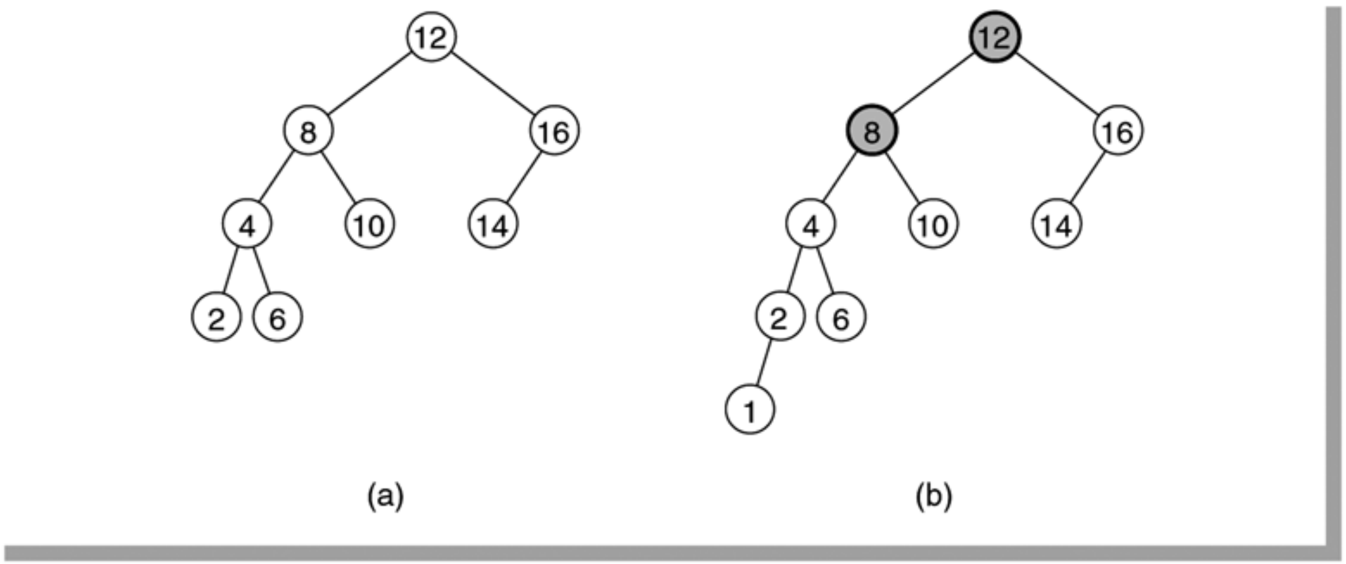


figure 19.21

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

An AVL tree has logarithmic height

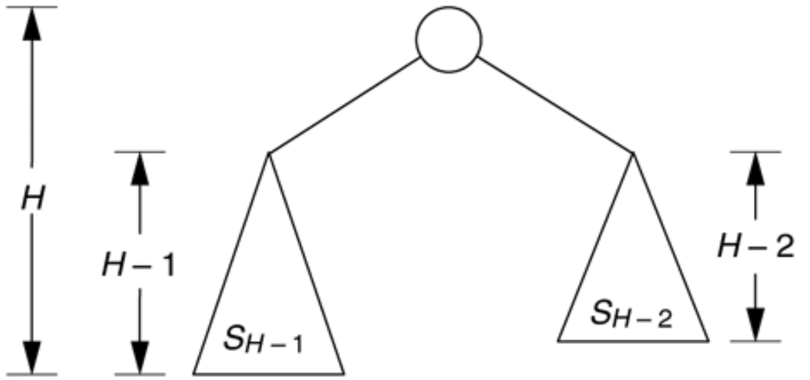


figure 19.22

Minimum tree of height H

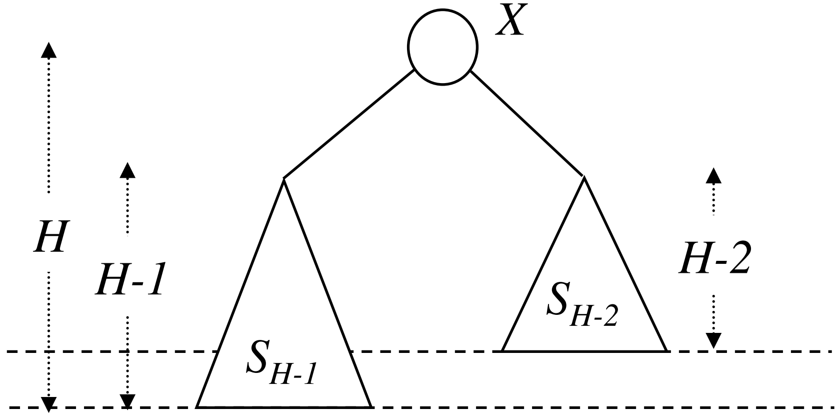
The minimum number of nodes S_H in an AVL tree of height H satisfies
$S_H = S_{H-1} + S_{H-2} + 1$, $S_0 = 1$, and $S_1 = 2$.
$S_H = F_{H+3} - 1$, where F_i is the i'th Fibonacci number.

$$S_H + 1 \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{H+3}$$

$$H < 1.44\log_2(N+2) - 1.328$$

The worst case height is 44% more than minimum possible for binary trees.

Insertion into an AVL tree



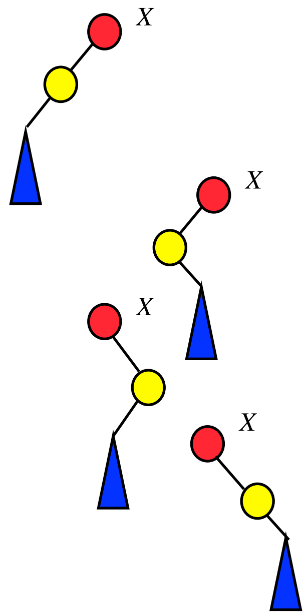
Insertion into *X*'s left subtree may violate the AVL balance condition.

In the following *X* denotes the deepest node to be balanced.

Insertion into an AVL tree

A violation of might occur in four cases:

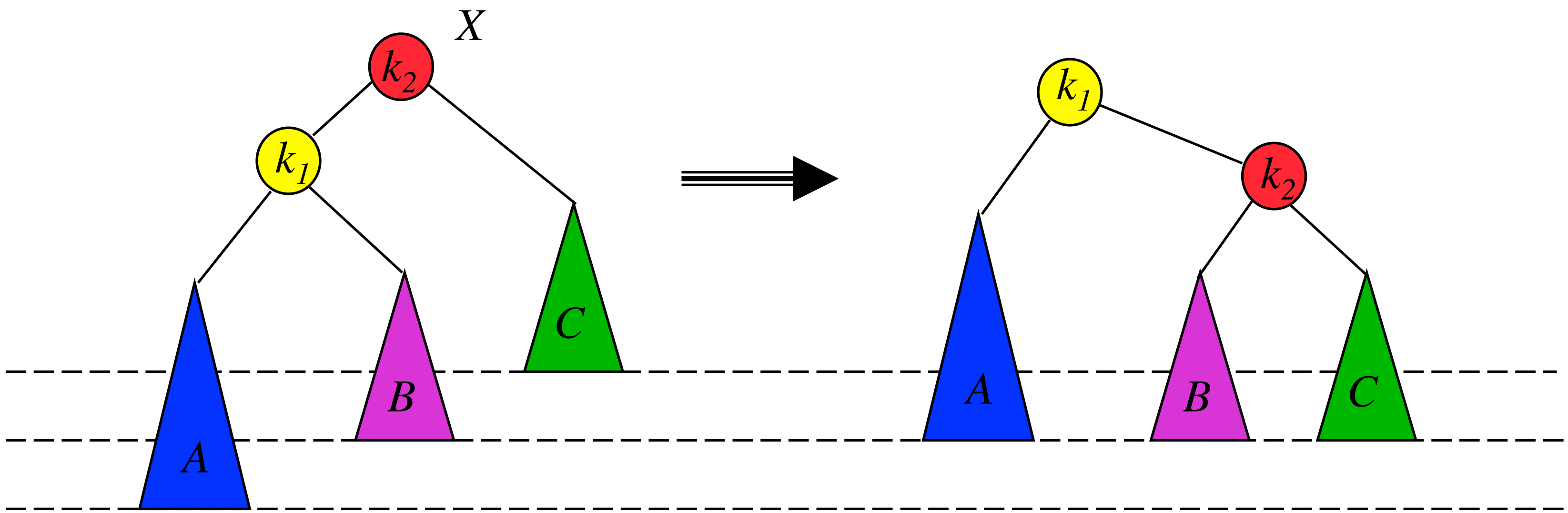
1. An insertion in the *left* subtree of the *left* child of X.
2. An insertion in the *right* subtree of the *left* child of X.
3. An insertion in the *left* subtree of the *right* child of X.
4. An insertion in the *right* subtree of the *right* child of X.



Cases 1 and 4 are symmetric with respect to X.
Cases 2 and 3 are symmetric with respect to X.

Case 1

left-left

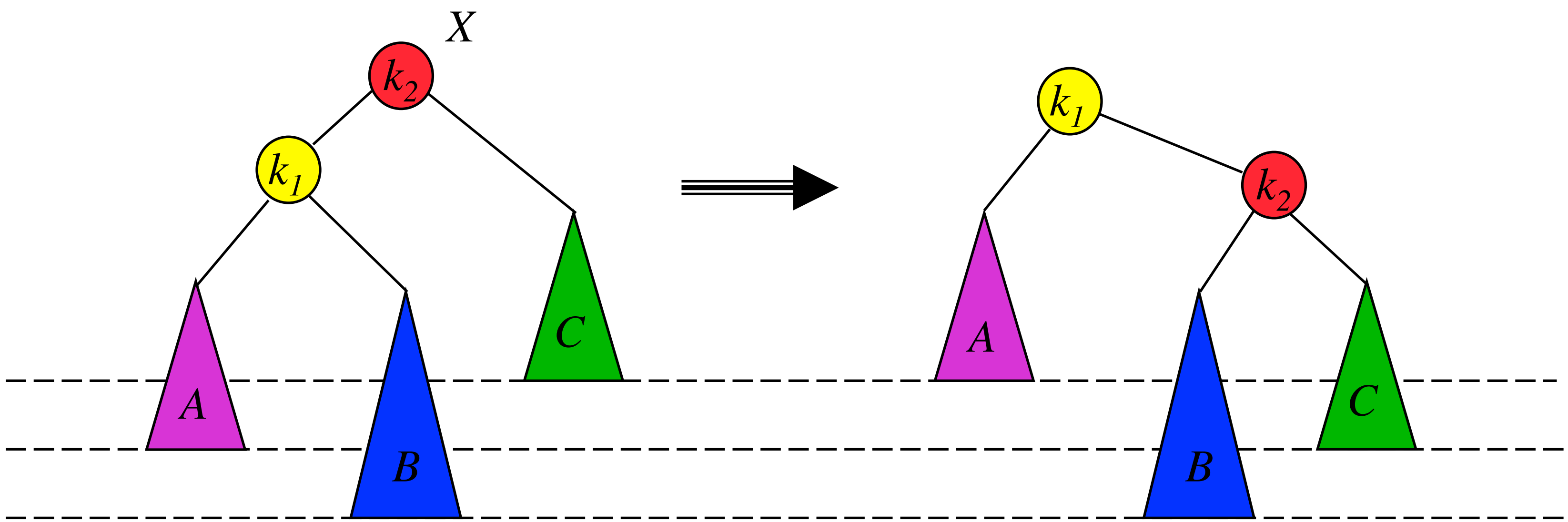


A right rotation restores the balance.

No further rotations are needed.

Case 2

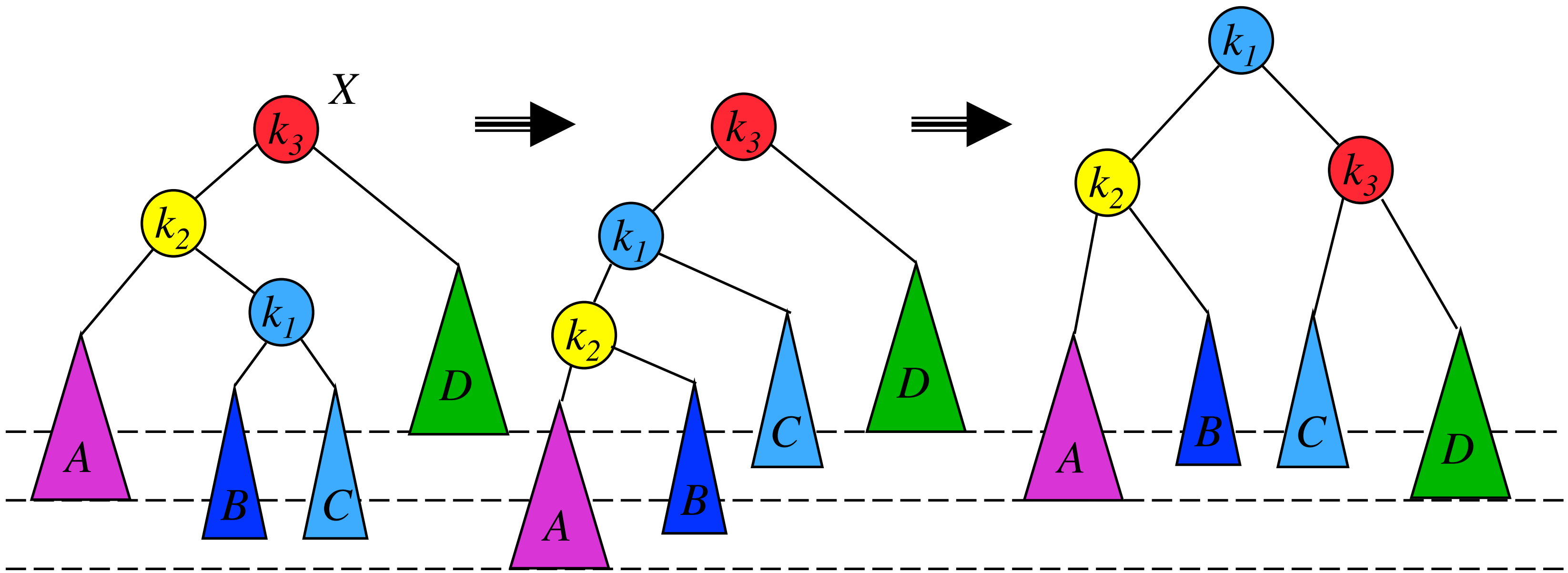
left-right



A right rotation **does not** restore the balance.

Case 2

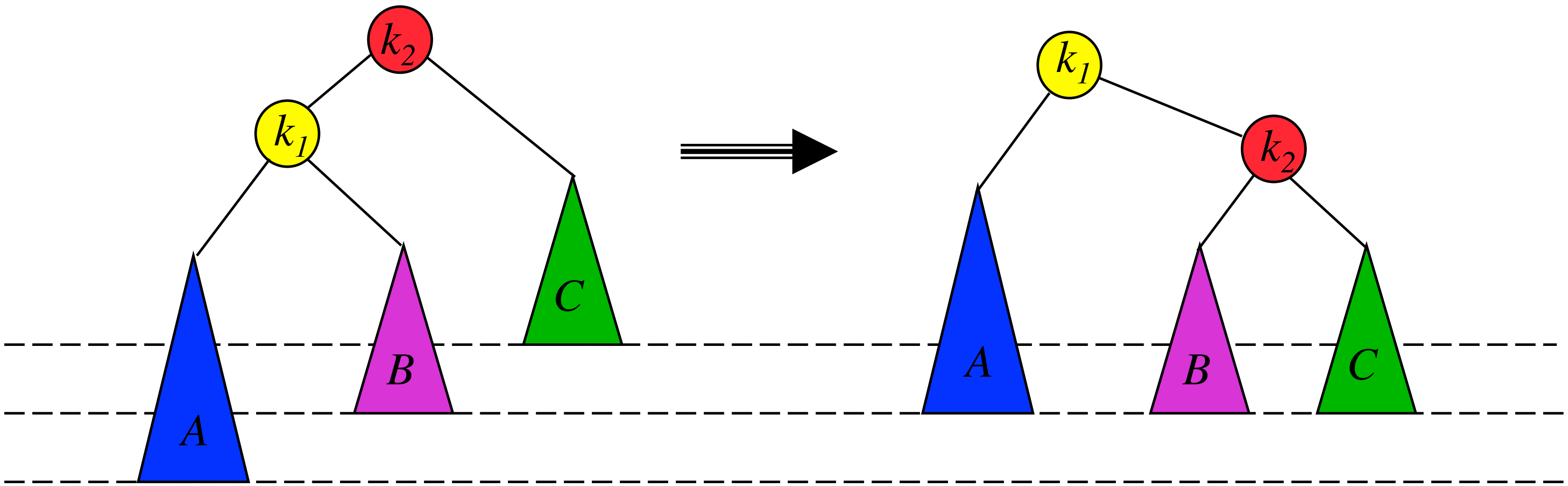
left-right



A left-right double rotation restores the balance.

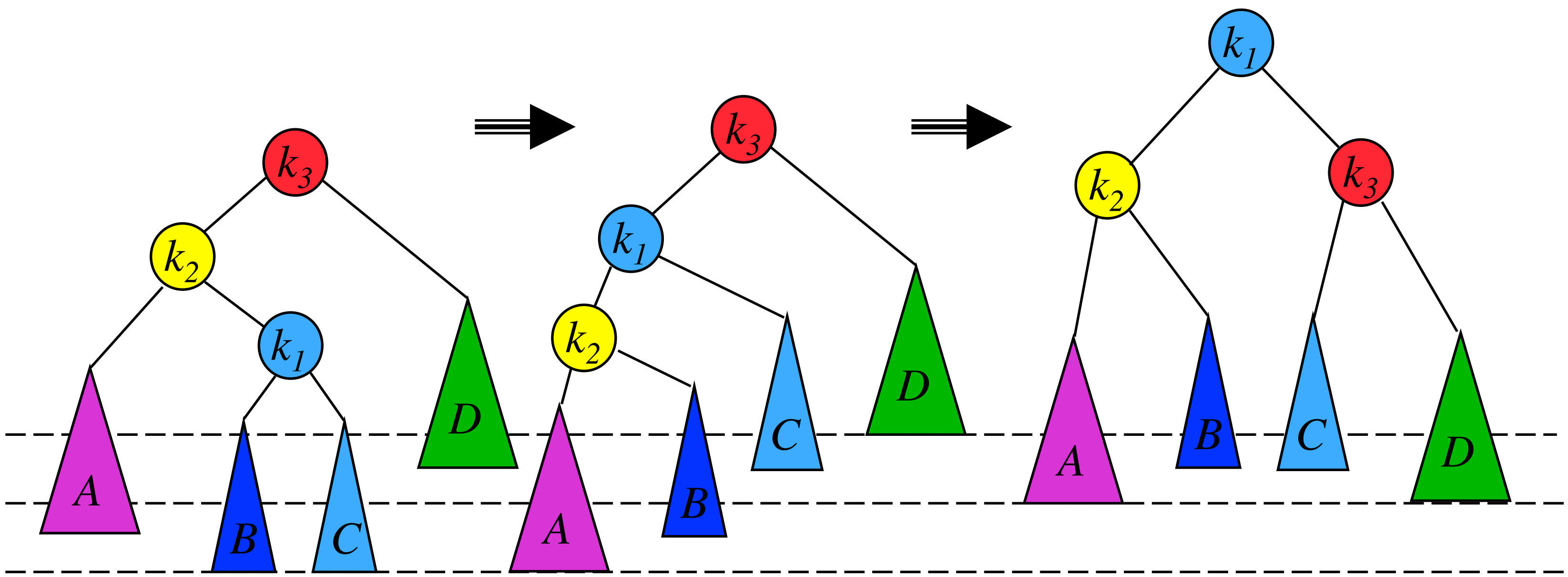
No further rotations are needed.

Implementation of single right rotation




```
BinaryNode rotateWithLeftChild(BinaryNode k2) {
    BinaryNode k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    return k1;
}
```

Implementation of double left-right rotation




```
BinaryNode doubleRotateWithLeftChild(BinaryNode k3) {
    k3.left = rotateWithRightChild(k3.left);
    return rotateWithLeftChild(k3);
}
```

Height information

An extra integer field, `height`, is added in class `BinaryNode`.

`height` stores the height of the tree that has the current node as root.

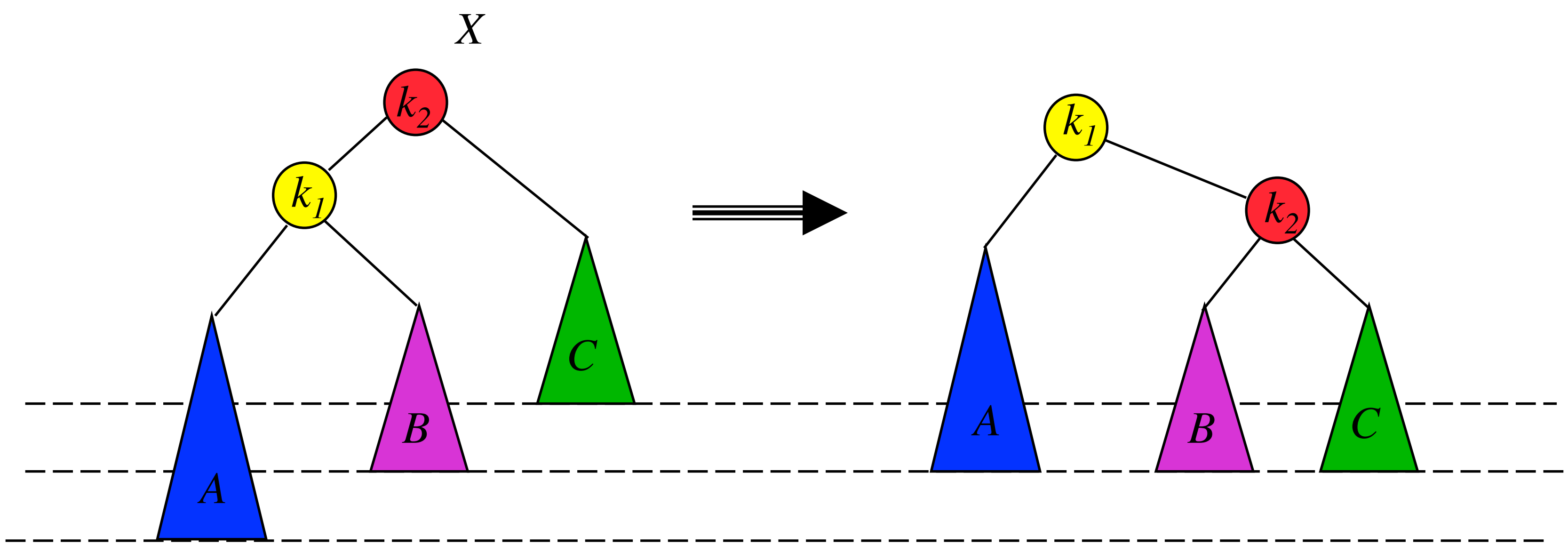
Auxiliary method in `BinaryNode`:

```
static int height(BinaryNode t) {
    return t == null ? -1 : t.height;
}
```

Implementation of `insert`

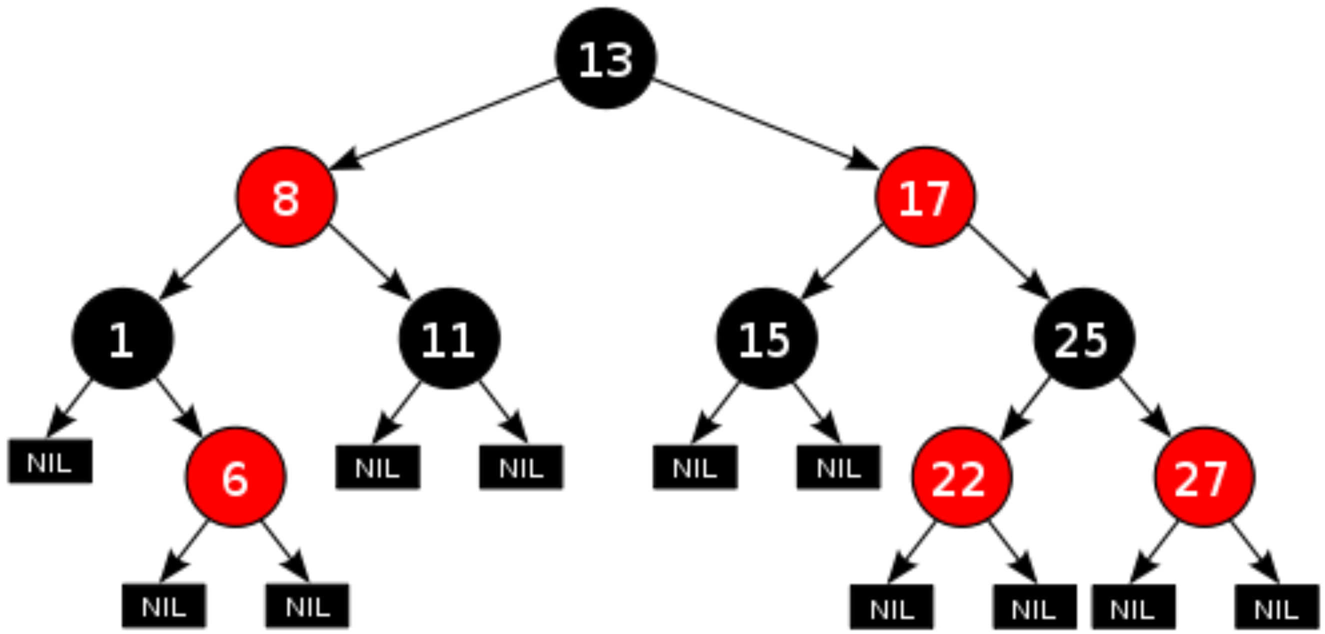
```
BinaryNode insert(Comparable x, BinaryNode t) {
    if (t == null)
        t = new BinaryNode(x, null, null);
    else if (x.compareTo(t.element) < 0) {
        t.left = insert(x, t.left);
        if (height(t.left) - height(t.right) == 2)
            if (x.compareTo(t.left.element) < 0)
                t = rotateWithLeftChild(t); // case 1
            else
                t = doubleRotateWithLeftChild(t); // case 2
    } else if (x.compareTo(t.element) > 0 ) {
        ... // case 3 or 4
    } else
        throw new DuplicateItemException();
    t.height = Math.max(height(t.left), height(t.right)) + 1;
    return t;
}
```

Maintenance of `height`




```
BinaryNode RotateWithLeftChild(BinaryNode k2) {
    BinaryNode k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    k2.height = Math.max(height(k2.left), height(k2.right)) + 1;
    k1.height = Math.max(height(k1.left), k2.height) + 1;
    return k1;
}
```

Red-black trees



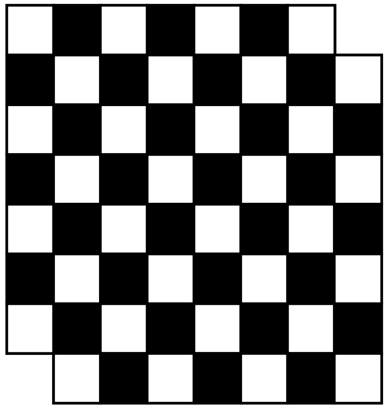
Problem

The damaged chessboard

A chess board with 8x8 squares can be covered by 32 domino pieces where each domino piece covers 2 squares.

The bottom-left and top-right corners are now taken away. Can 31 pieces cover the board?

Domino piece



Coloring solves the problem easily. Why?

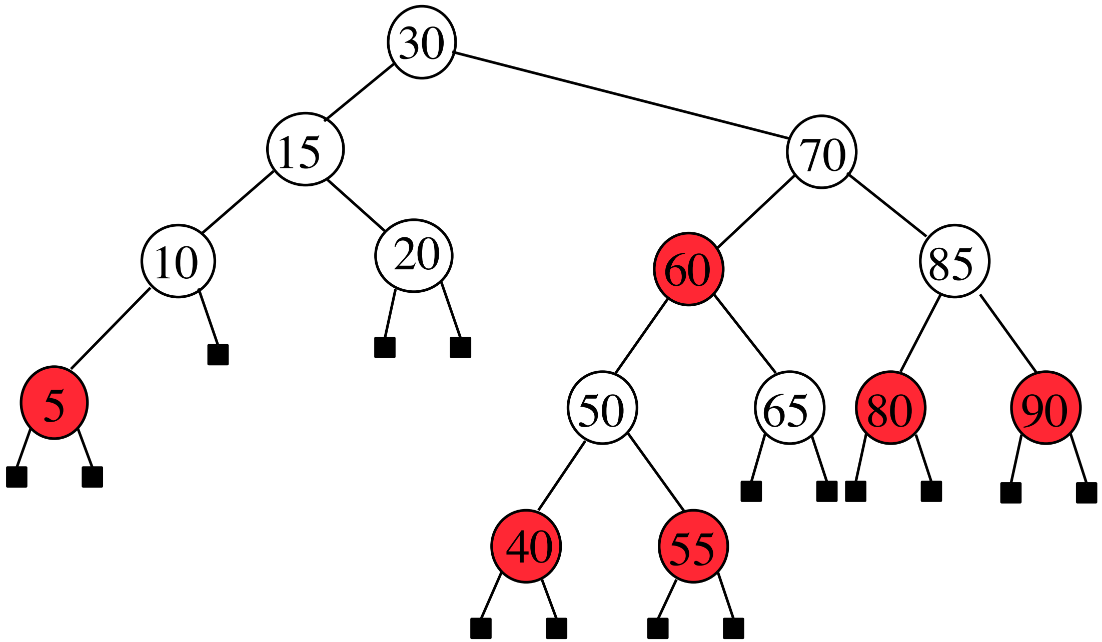
Red-black tress

(R. Bayer, 1972)

A **red-black tree** is a binary search tree having the following ordering properties:

1. Every node is colored either red or **black**.
2. The root is **black**.
3. If a node is red, its children must be **black**.
4. Every path from a node to a `null` link must contain the same number of **black** nodes.

A red-black tree



0. The tree is a binary search tree.
1. Every node is colored either red or **black**.
2. The root is **black**.
3. If a node is red, its children must be **black**. (`null` nodes are **black**)
4. Every path from a node to a `null` link must contain the same number of **black** nodes: 3.

The height of red-black trees

The height H of a red-**black** tree satisfies

$H \leq 2 \log_2(N+1)$

In a randomly generated tree H is very close to $\log_2 N$:
$1.002 \log_2 N$.

Insertion into a red-black tree

A new item is always inserted as a leaf in the tree.

If we color the new node **black**, we violate property 4, and it is not immediately clear how this property may be restored.

If we color the new node red and its parent is **black**, we are done.

If its parent is red, we violate property 3.

We must see to that the new red node gets a **black** parent

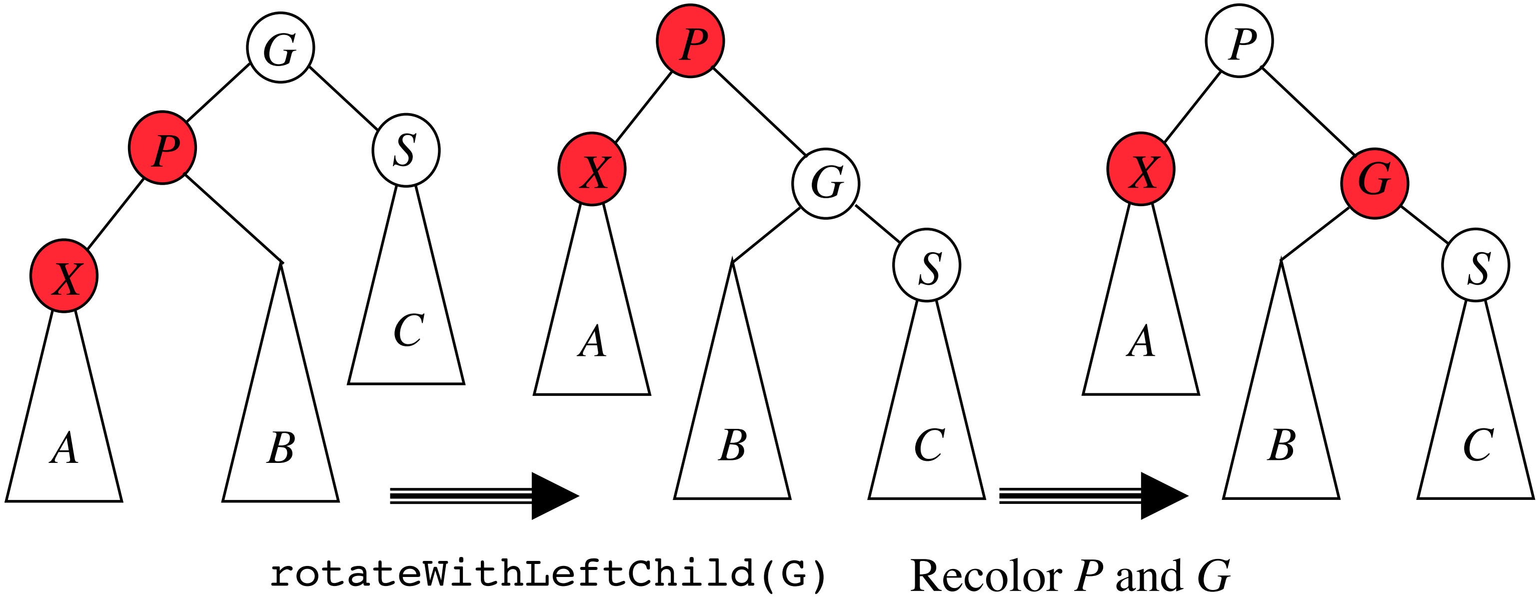
How?

Answer: Use rotations and color changes.

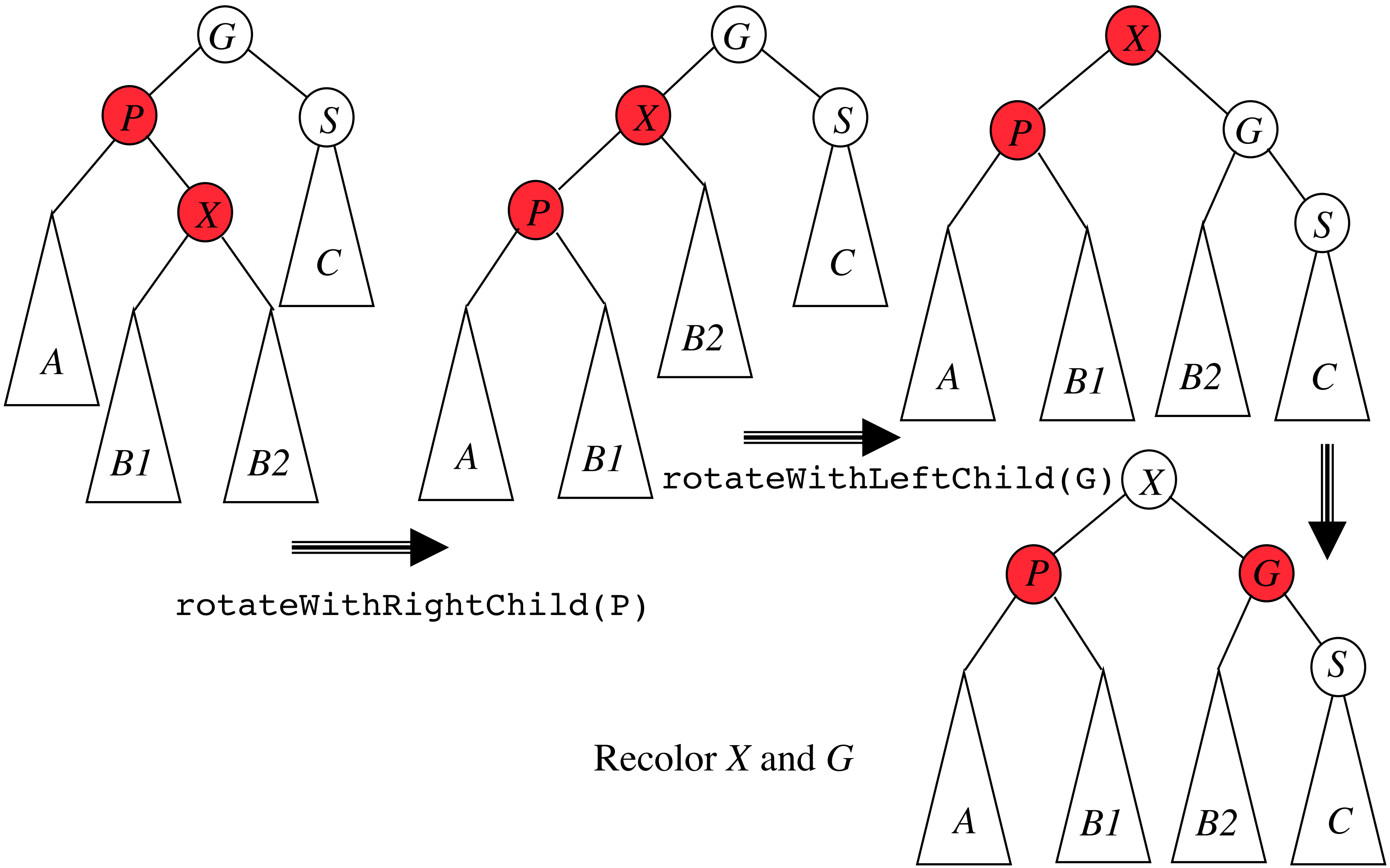
Five cases

X: the current red node
P: *X*'s red parent
G: *X*'s grandparent (has to be **black**)
S: Sibling of *X*'s parent

Case 1: *X* is *left* child of *P*, *P* is *left* child of *G*, and *S* is **black**.



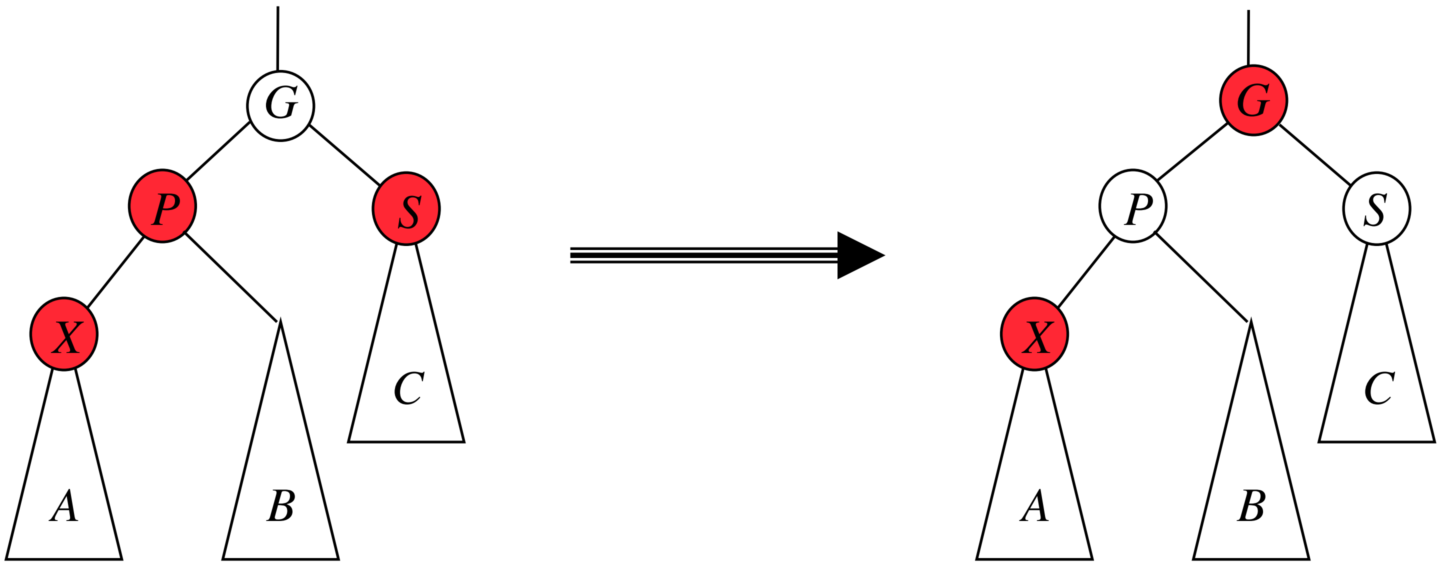
Case 2: *X* is *right* child of *P*, *P* is *left* child of *G*, and *S* is **black**.



Case 3: *X* is *right* child of *P*, *P* is *right* child of *G*, and *S* is **black**.
Symmetric to case 1

Case 4: *X* is *left* child of *P*, *P* is *right* child of *G*, and *S* is **black**.
Symmetric to case 2

Case 5: Both *P* and *S* are red.



Perform a color flip on *P*, *S* and *G*.
If *G* is the root, then color *G* **black**.

After a case 5 repair

The red coloring of G in case 5 may violate property 3 (if G's parent is red).

This violation can be fixed by a single rotation (in cases 1 and 3), and a double rotation (in cases 2 and 4). Again, case 5 may arise.

We could percolate this *bottom-up* procedure until we reach the root. To avoid the possibility of having to rotate up the tree, we apply a *top-down* procedure. Specifically, we guarantee that when we arrive at a leaf and insert a node, S is not red.

Top-down insertion

The rotations after a type 5 repair requires access to *X*'s great-grandparent. The implementation in the textbook maintain the following references:

`current`:	the current node (*X*)
`parent`:	the parent of the current node (*P*)
`grand`:	the grandparent of the current node (*G*)
`great`:	the great-grandparent of the current node

These references can be omitted when insertion is performed top-down. The top-down code is given in the following.

Java implementation

In class `BinaryNode` we add the field

```
boolean color;
```

In class `RedBlackTree` we define the constants

```
static final boolean RED = false;
static final boolean BLACK = true;
```

We use a `BinaryNode` object in place of `null` links:

```
static BinaryNode nullNode;
static {
    nullNode = new BinaryNode(null);
    nullNode.left = nullNode.right = nullNode;
    nullNode.color = BLACK;
}
```

```
public void insert(Comparable x)
  { root = insert(x, root, true); root.color = BLACK; }
```

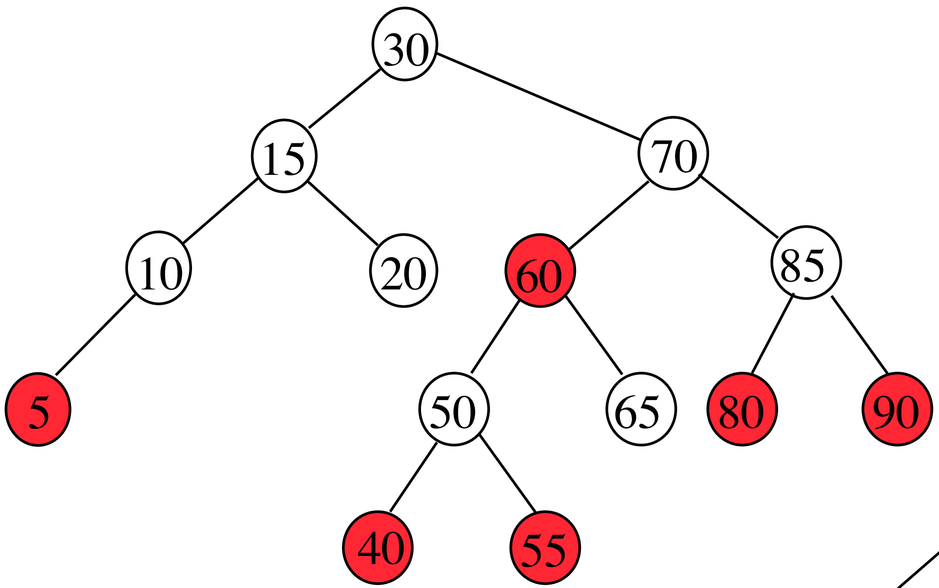
```
BinaryNode insert(Comparable x, BinaryNode t, boolean rightChild) {
    if (t == nullNode) {
        t = new BinaryNode(x, nullNode, nullNode); t.color = RED;
    } else {
        if (t.left.color == RED && t.right.color == RED) {
            t.color = RED; t.left.color = t.right.color = BLACK;
        }
        if (x.compareTo(t.element) < 0) {
            t.left = insert(x, t.left, false);
            if (rightChild && t.color == RED && t.left.color == RED)
                t = rotateWithLeftChild(t);
            if (t.left.color == RED && t.left.left.color == RED) {
                t = rotateWithLeftChild(t);
                t.color = BLACK; t.right.color = RED;
            }
        }
        else if (x.compareTo(t.element) > 0) { ... }
        else throw new DuplicateItemException(x.toString());
    }
    return t;
}
```

Handling the symmetric case

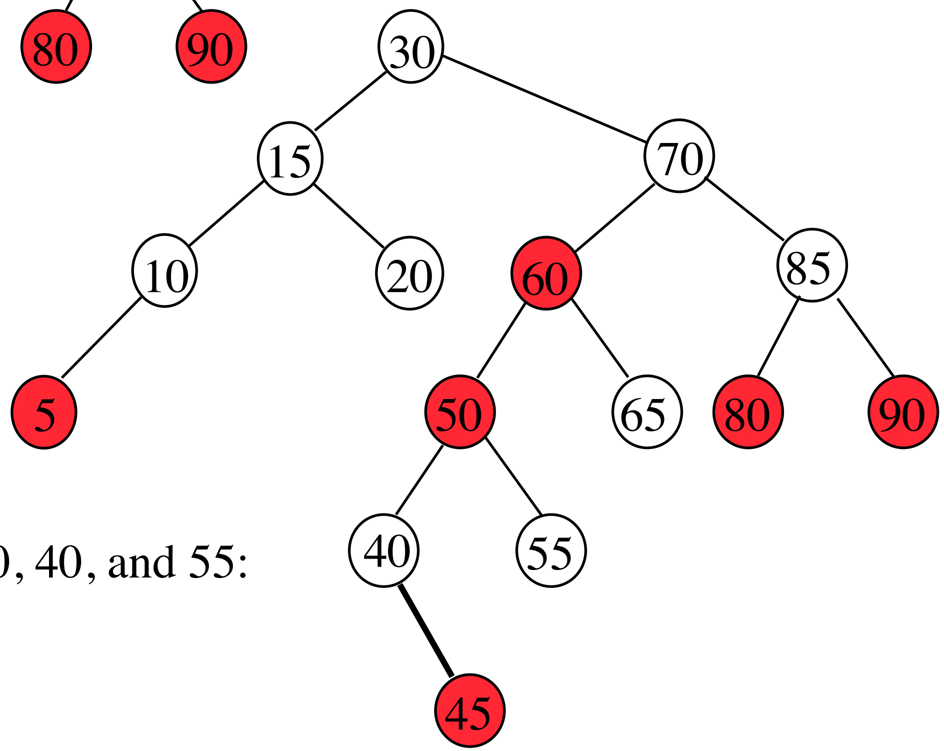
Replace `left` by `right`
Replace `right` by `left`
Replace `rightChild` by `!rightChild`
Replace `rotateWithLeftChild` by `rotateWithRightChild`

```
t.right = insert(x, t.right, true);
if (!rightChild && t.color == RED && t.right.color == RED)
    t = rotateWithRightChild(t);
if (t.right.color == RED && t.right.right.color == RED) {
    t = rotateWithRightChild(t);
    t.color = BLACK; t.left.color = RED;
}
```

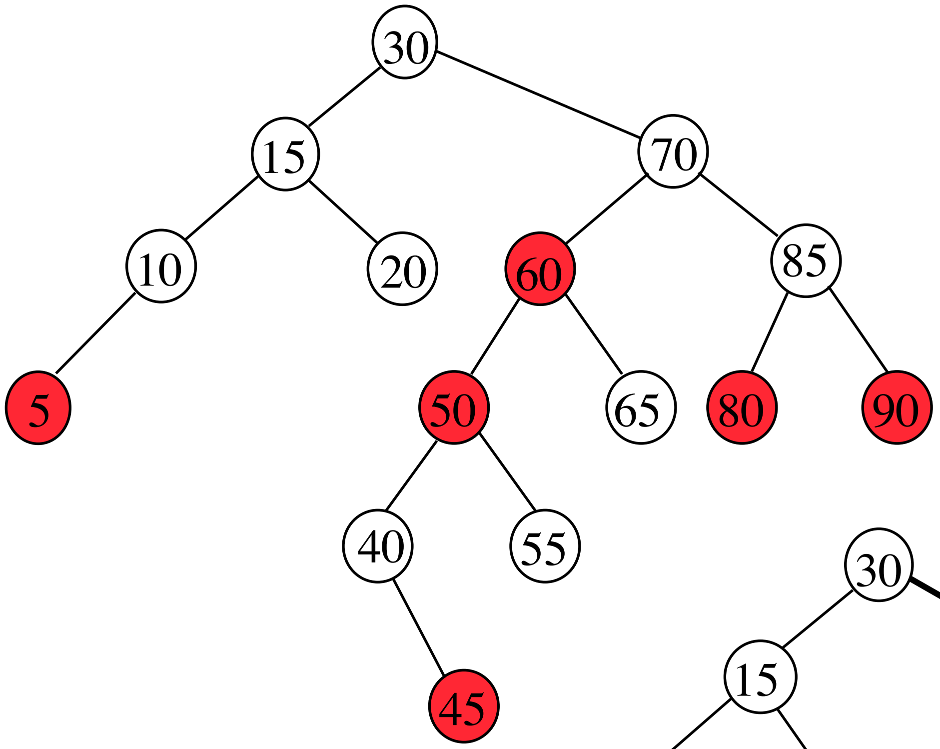
Insertion example

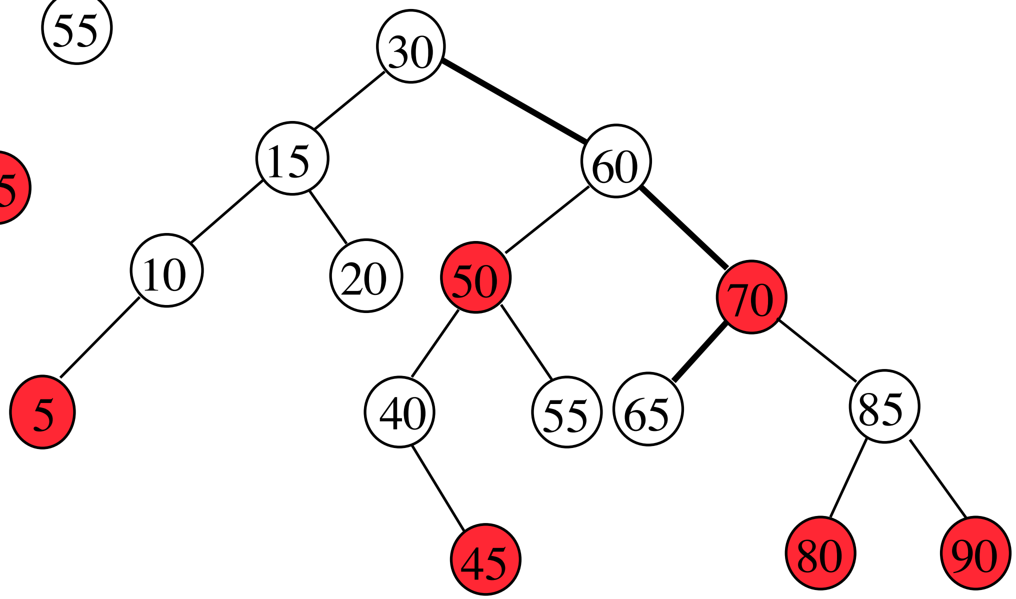


45 to be inserted:



Flip color of 50, 40, and 55:





Right rotation at 70 and color flip of 60 and 70:

end

AVL trees versus red-black trees

Although the red-**black** tree balancing properties are slightly weaker than the AVL tree balancing properties experiments suggest that the number of nodes traversed during a search is almost identical.

However, updating a red-**black** tree requires lower overhead than AVL trees. Insertion into an AVL tree requires (in worst case) two passes on a path (from the root to a leaf and up again), whereas insertion into a red-**black** tree can be performed in one pass.

Implementation of deletion is complicated for both AVL trees and red-**black** trees.

AA-trees

(Arne Andersson, 1993)

An AA-tree is a red-**black** tree that has one extra property:

(5) A *left* child must not be red.

This property simplifies implementation:

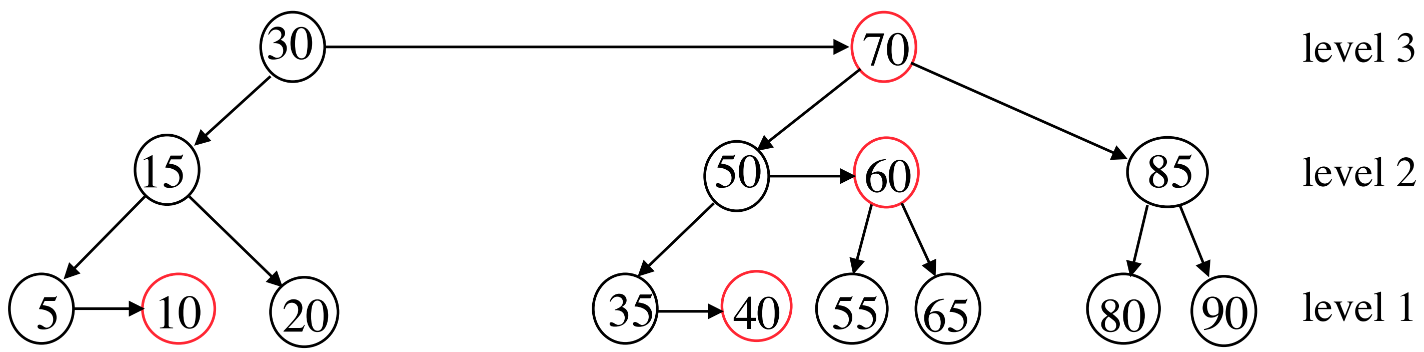
(1) it eliminates half of the restructuring cases;
(2) it removes an annoying case for the deletion algorithm

Properties of AA-trees

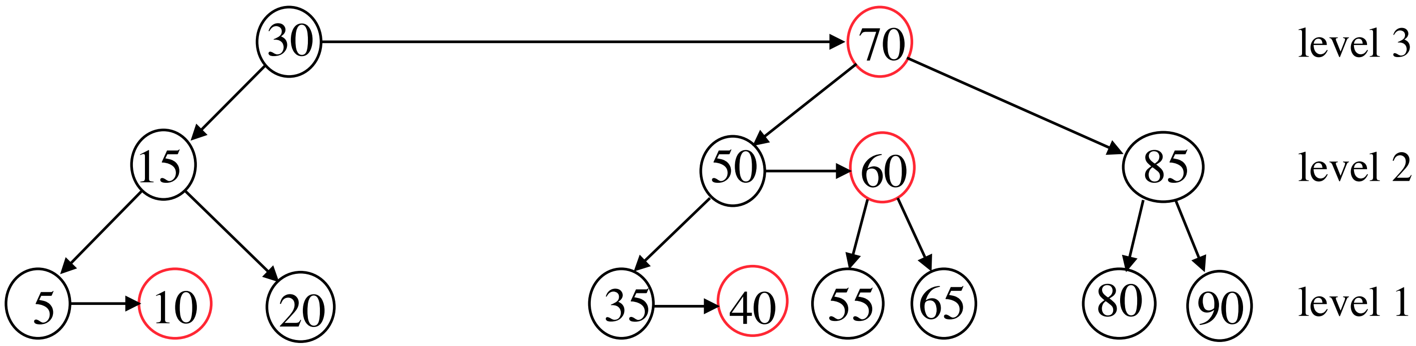
Level replaces color

The *level* of a node is

- 1, if the node is a leaf
- the level of its parent, if the node is red
- one less than the level of its parent, if the node is **black**



Properties

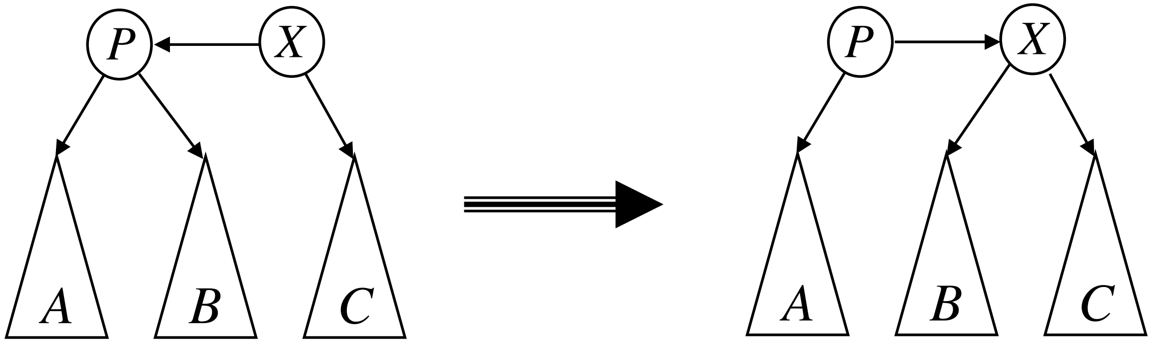


1. Horizontal links are right links
2. There may not be two consecutive horizontal links
3. Nodes at level 2 or higher must have two children
4. If a node at level 2 does not have a right horizontal link, its two children are at the same level

Rotations can be used to maintain the AA-tree properties

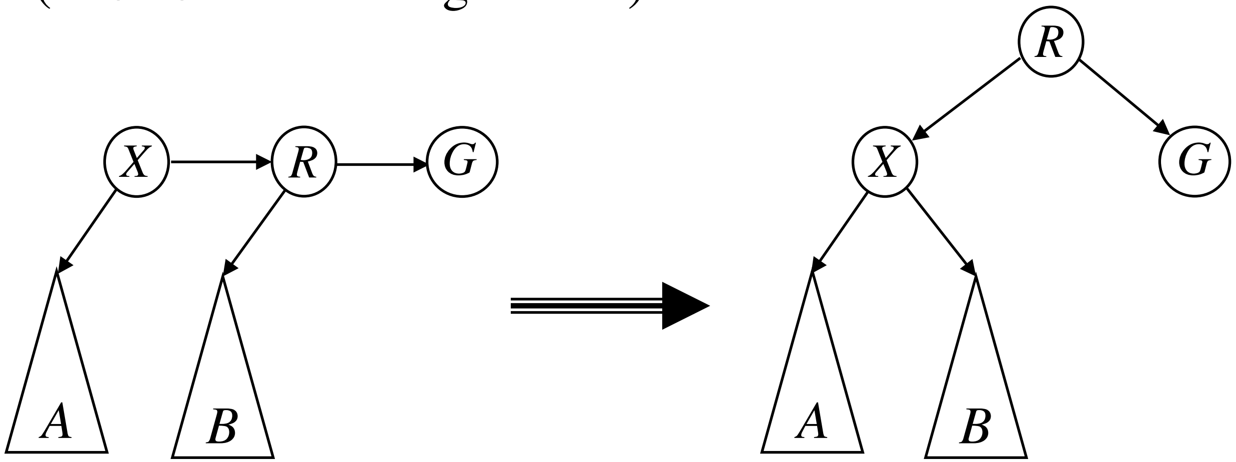
There are only 2 cases that require restructuring:

Case 1 (horizontal left link):




```
BinaryNode skew(BinaryNode t) {
    if (t.left.level == t.level)
        t = rotateWithLeftChild(t);
    return t;
}
```

Case 2 (two consecutive right links):




```
BinaryNode split(BinaryNode t) {
    if (t.right.right.level == t.level) {
        t = rotateWithRightChild(t);
        t.level++;
    }
    return t;
}
```

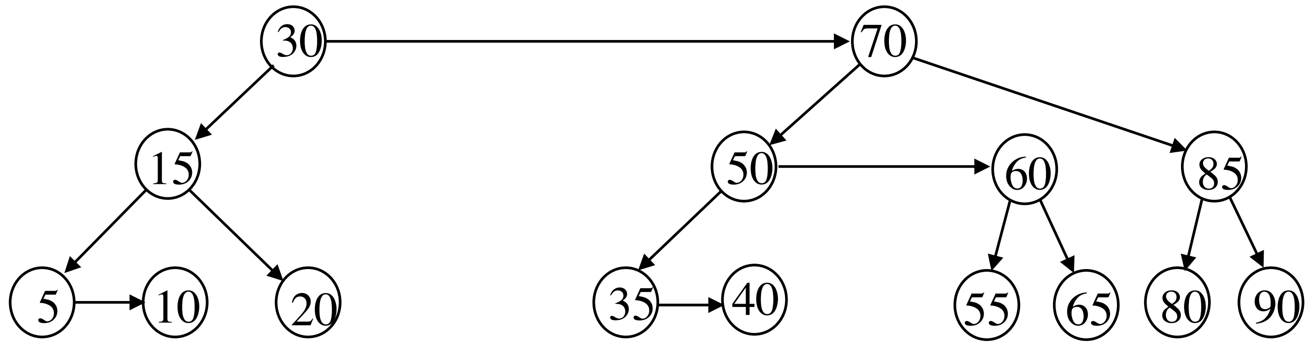
After a `skew`, a `split` may be required

Implementation of `insert`

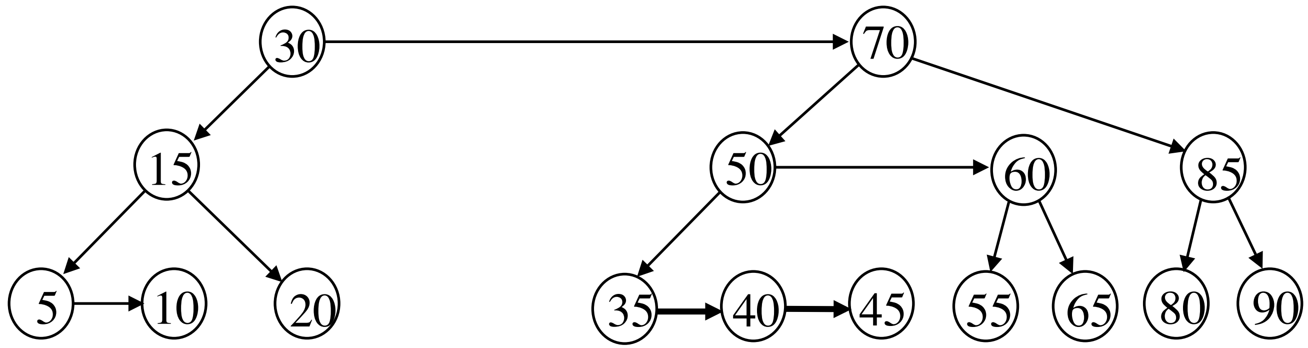
Use recursion and call `skew` and `split` on the way back.

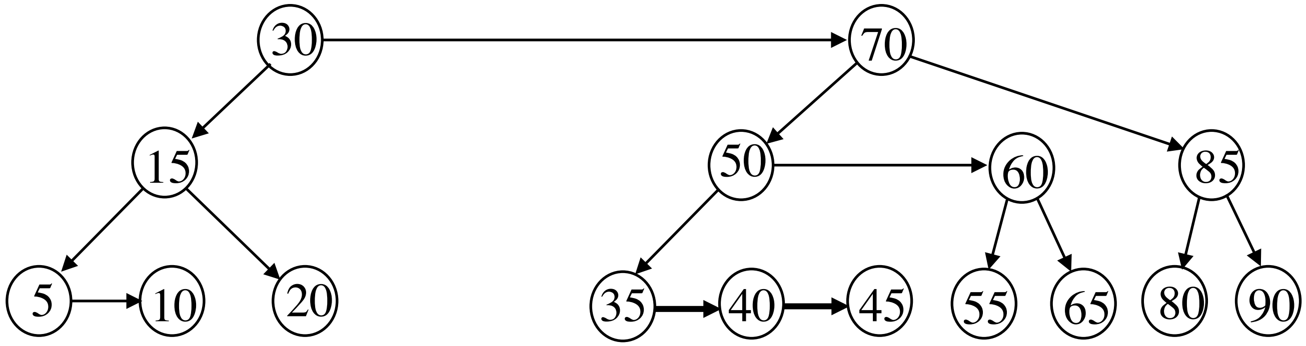
```
BinaryNode insert(Comparable x, BinaryNode t) {
    if (t == nullNode)
        t = new BinaryNode(x, nullNode, nullNode);
    else if (x.compareTo(t.element) < 0)
        t.left = insert(x, t.left);
    else if (x.compareTo(t.element) > 0)
        t.right = insert(x, t.right);
    else
        throw new DuplicateItemException();
    return split(skew(t));
}
```

Insertion example

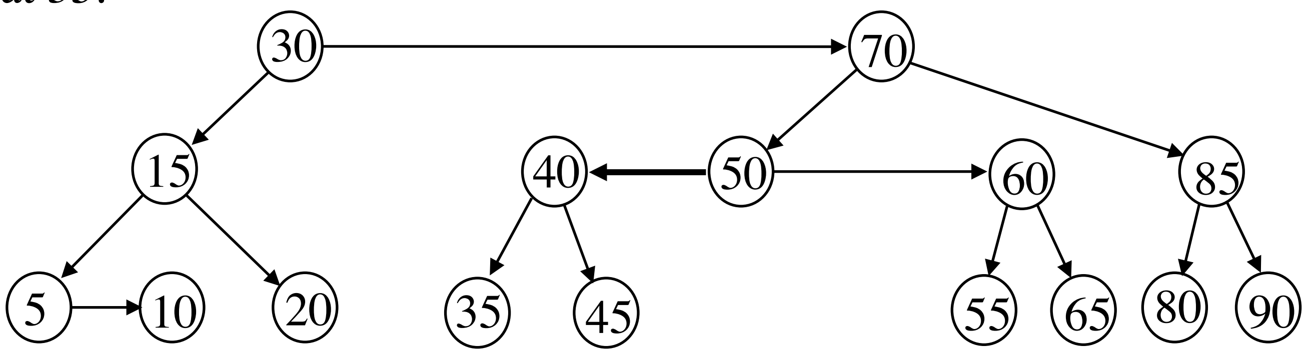


45 to be inserted:

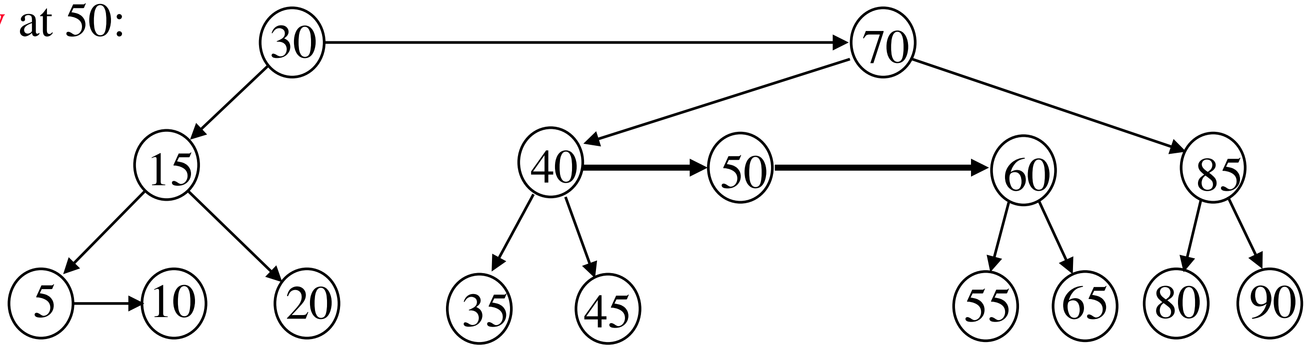


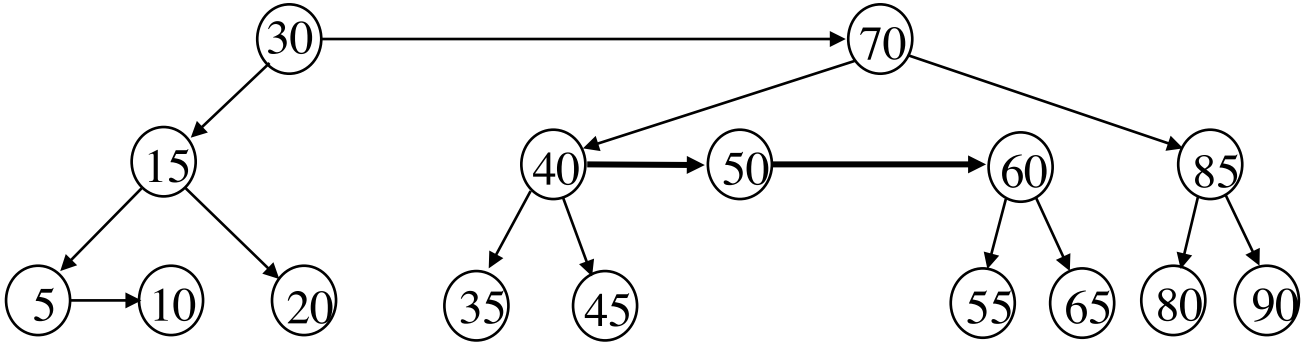


split at 35:

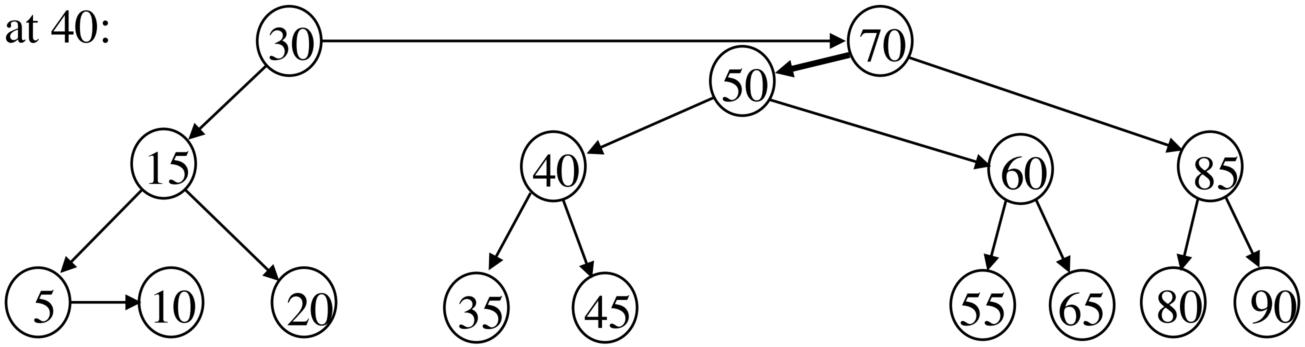


skew at 50:

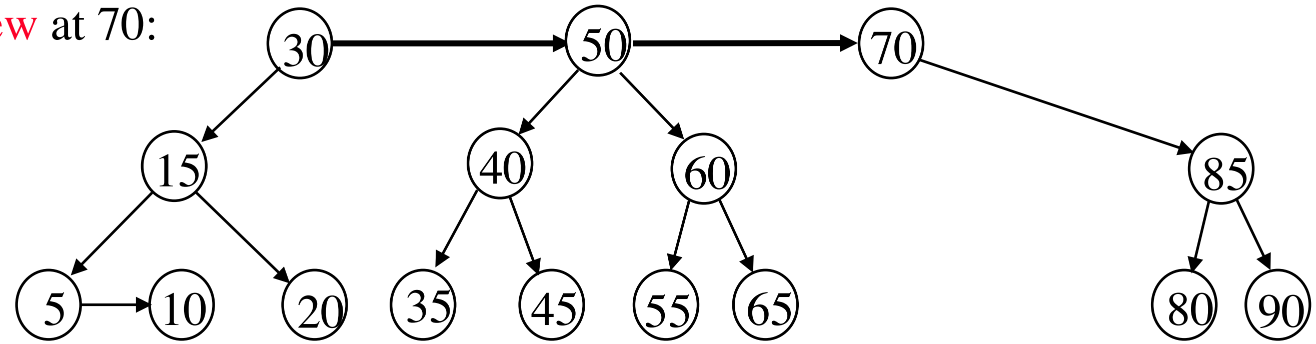


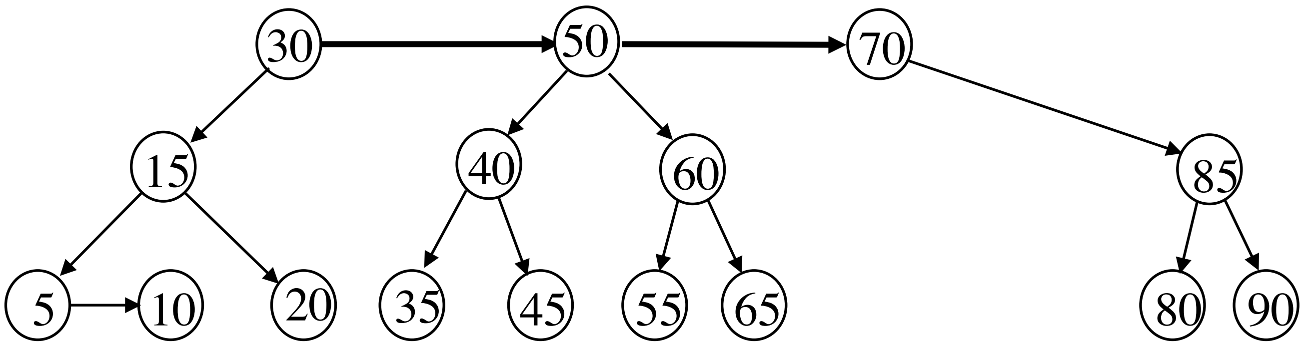


split at 40:

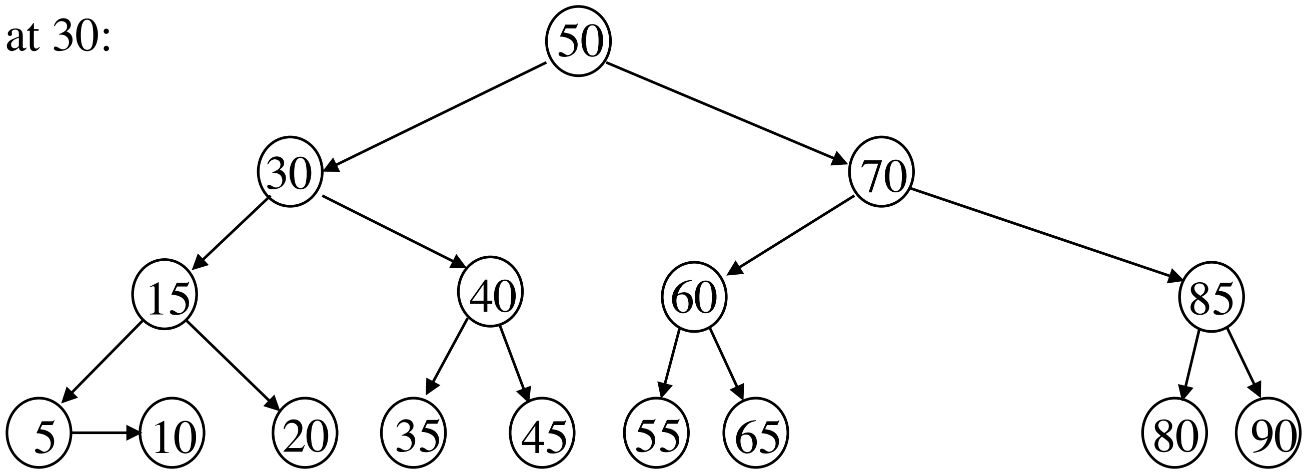


skew at 70:




30
50
70
15
40
60
85
5
10
20
35
45
55
65
80
90

split at 30:


50
30
70
15
40
60
85
5
10
20
35
45
55
65
80
90

Implementation of `remove`

Use recursion and call `skew` and `split` on the way back.

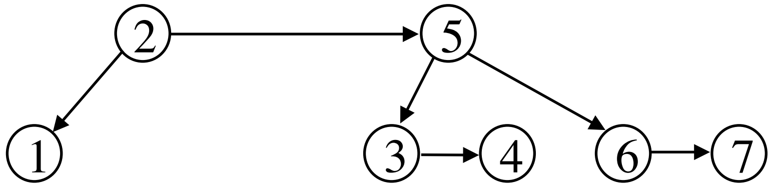
```
BinaryNode remove(Comparable x, BinaryNode t) {
    if (t == nullNode)
        return nullNode;
    lastNode = t;
    if (x.compareTo(t.element) < 0)
        t.left = remove(x, t.left);
    else {
       deletedNode = t;
       t.right = remove(x, t.right);
    }
    if (t == lastNode) {   // at level 1
        if (deletedNode == nullNode ||
            x.compareTo(deletedNode.element) != 0)
            throw new ItemNotFoundException();
        deletedNode.element = t.element;
        t = t.right;
    } else {
        /* see next slide */
    }
    return t;
}
```

Maintaining the AA-tree properties

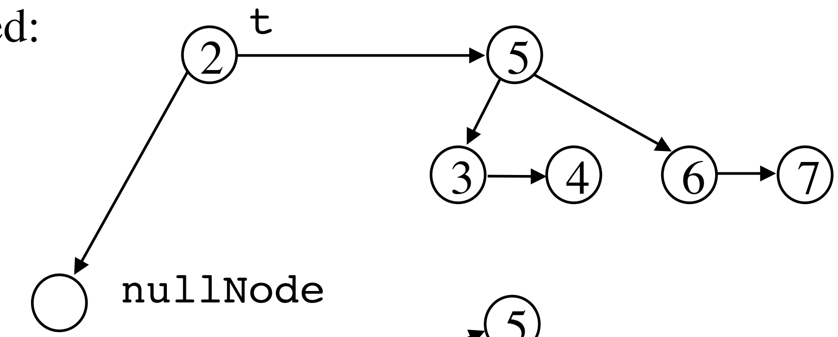
```
if (t.left.level  < t.level - 1 ||
    t.right.level < t.level - 1) {
    t.level--;
    if (t.right.level > t.level)
        t.right.level = t.level;
    t = skew(t);
    t.right = skew(t.right);
    t.right.right = skew(t.right.right);
    t = split(t);
    t.right = split(t.right);
}
```

See the textbook for an explanation

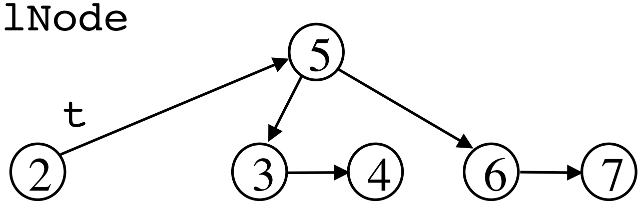
Deletion example



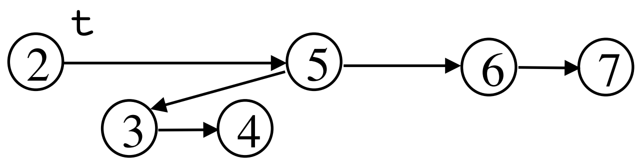
1 to be deleted:

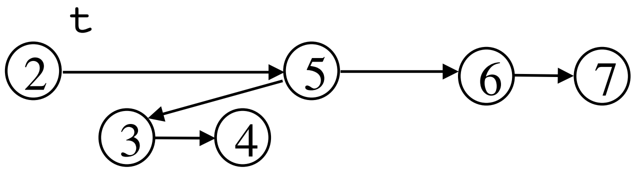


`t.level--`:



`t.right.level--`:

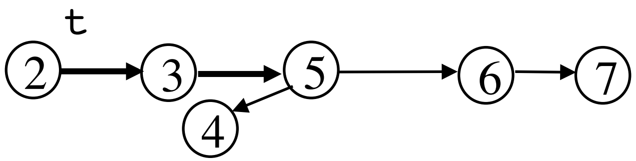




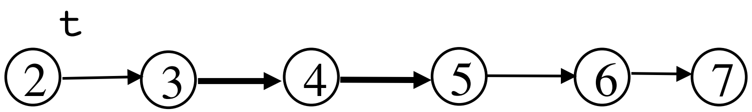
`t = skew(t):`

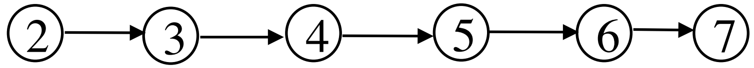
no effect

`t.right = skew(t.right):`

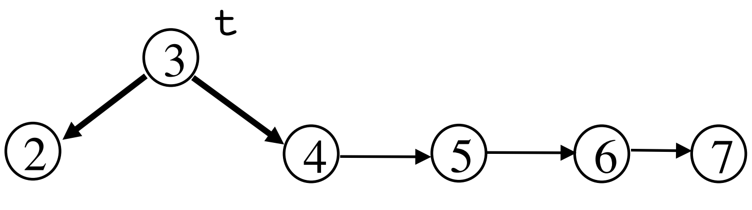


`t.right.right = skew(t.right.right):`

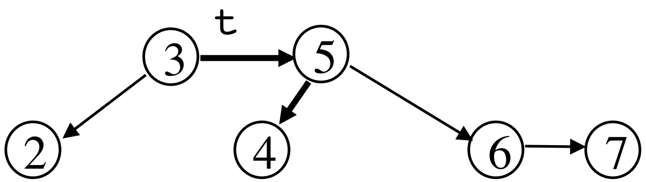




```
t = split(t):
```




```
t.right = split(t.right):
```



Red-black trees versus AA-trees

Experimental results

Insertion of 10,000,000 different integers into an initially empty tree, followed by deletion of each element, in random order.
2.8 GHz MacBook Pro.

Red-**black** tree (`java.util.TreeSet`)	57.9 seconds
AA-tree (`weiss.util.TreeSet`)	65.0 seconds

`java -Xmx1G`

B-tree

a data structure for external search

(Bayer and McCraight, 1970)

Suppose 10,000,000 records must be stored on a disk in such a way that search time is minimized.

An ordinary binary search tree:

Average case: $1.38*\log_2(10{,}000{,}000) \approx 32$ disk accesses

Worst case: 10,000,000 disk accesses!

Perfectly balanced search tree: $\log_2(10{,}000{,}000) \approx 24$ accesses

This in unacceptable. We want to reduce the number of disk accesses to a very small number, such as three or four.
Use a **B-tree** – a balanced *M-ary* search tree. A B-tree allows M-way branching in a tree, which has a height that is roughly $\log_M N$.

A 5-ary tree

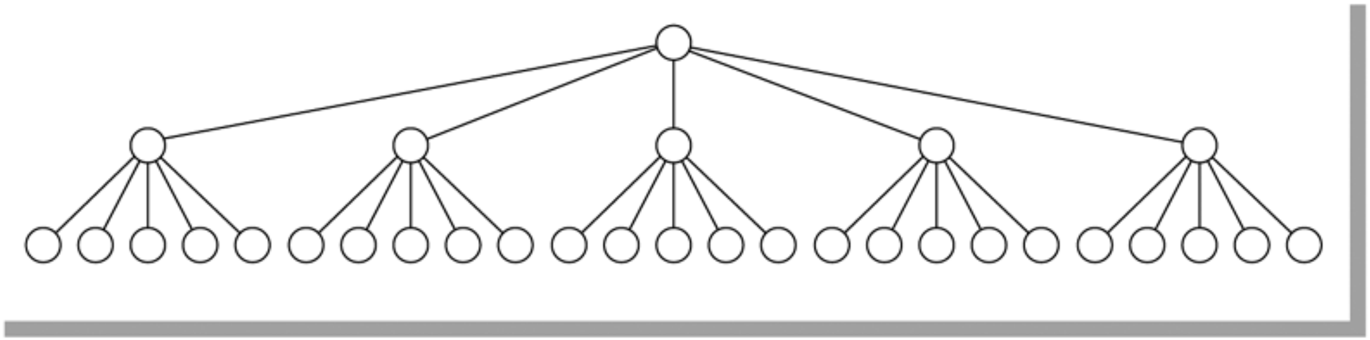
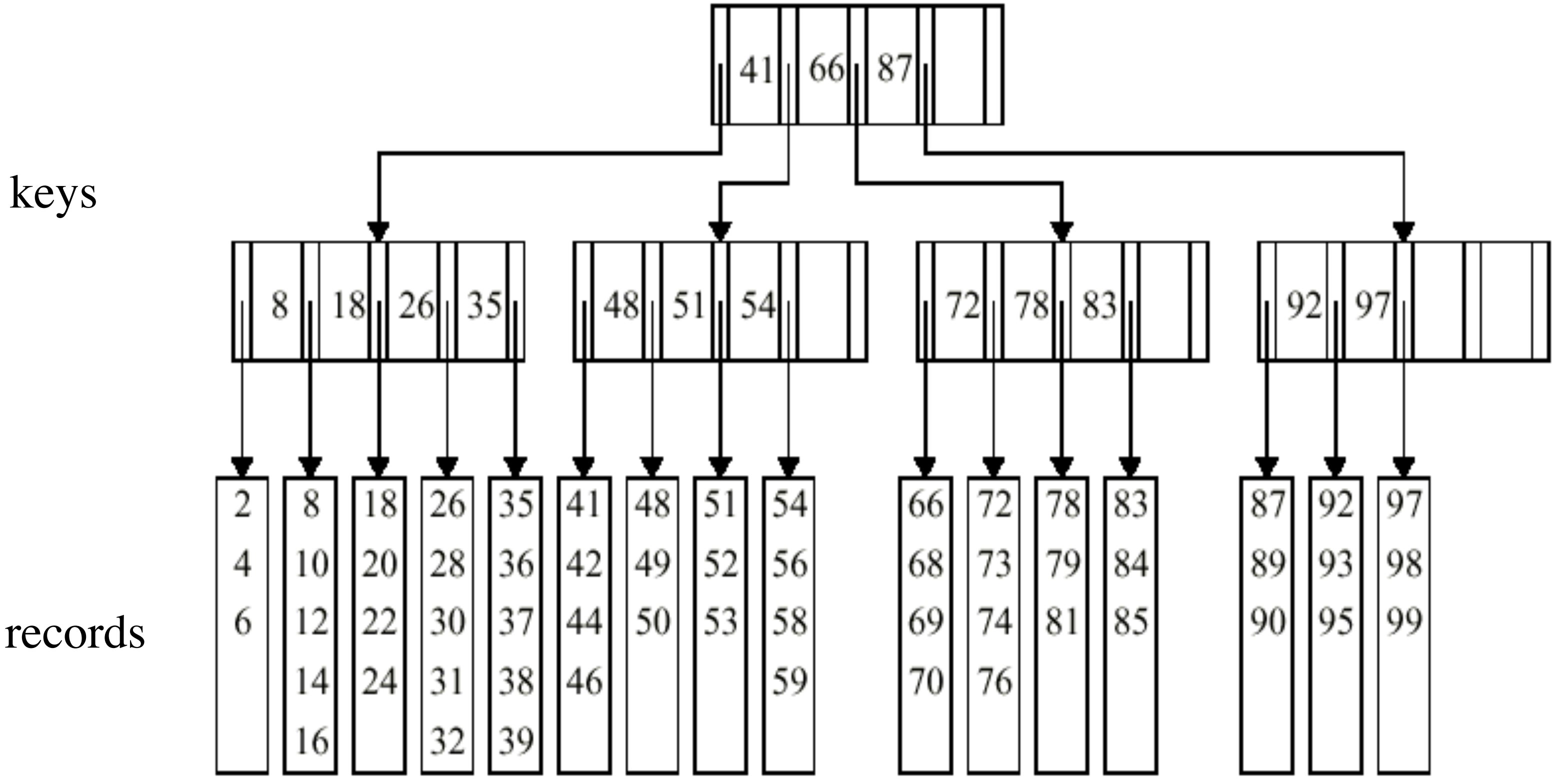


figure 19.83

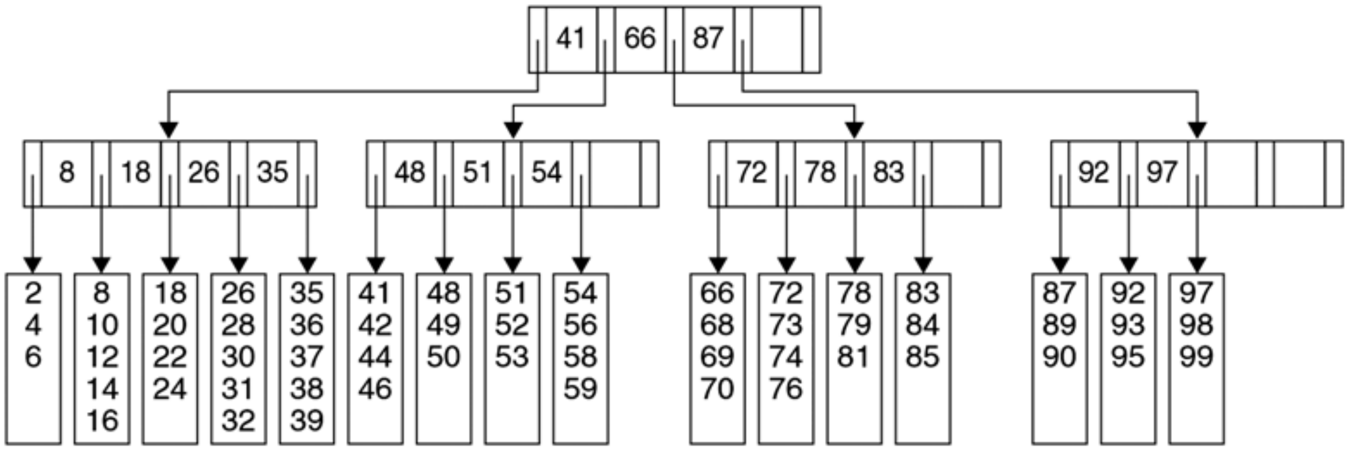
A 5-ary tree of 31 nodes has only three levels

B-tree of order 5



When all records are stored at the leaf level, the data structure is called a B^+-tree.

Definition of B-tree



A B-tree of order M is a M-ary tree with the following properties:

1. The data items are stored at leaves.
2. The nonleaf nodes store as many as M-1 keys to guide the searching; key i represents the smallest key in subtree $i+1$.
3. The root is either a leaf or has between 2 and M children.
4. All nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children.
5. All leaves are at the same depth and have between $\lceil L/2 \rceil$ and L data items, for some L.

B-tree example

Assume

- each of 10,000,000 records uses 256 bytes
- each key uses 32 bytes
- each branch uses 4 bytes
- each block holds 8,192 bytes

Choose M as large as possible: $(M\text{-}1)*32 + M*4 \leq 8192$. So we choose $M = 228$.
Each nonleaf node has at least $M/2 = 114$ children.

Choose L as large as possible: $L = 8192/256 = 32$.

Number of leaves: At most $10{,}000{,}000/(L/2) = 10{,}000{,}000/16 = 625{,}000$.

Height of the tree: $\log_{114}(625{,}000) \approx 3.4$. If the root is in RAM, only **3** disk accesses are required to find a record.

Insertion into a B-tree

simple case

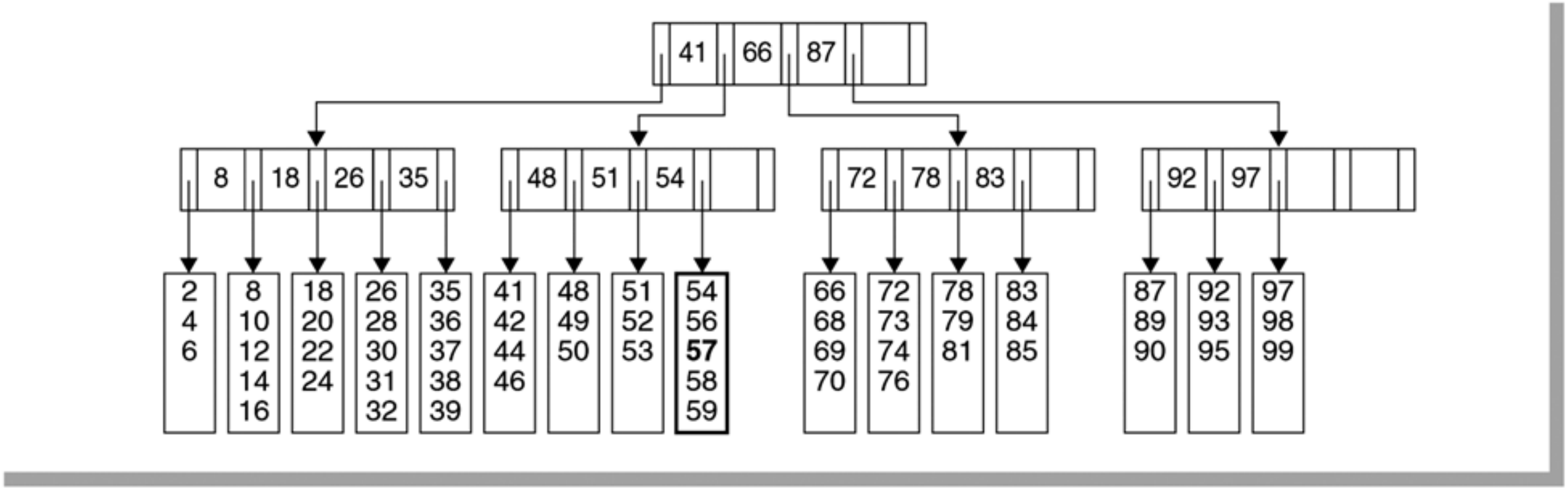


figure 19.85

The B-tree after insertion of 57 in the tree shown in Figure 19.84.

Insertion into a B-tree

node splitting

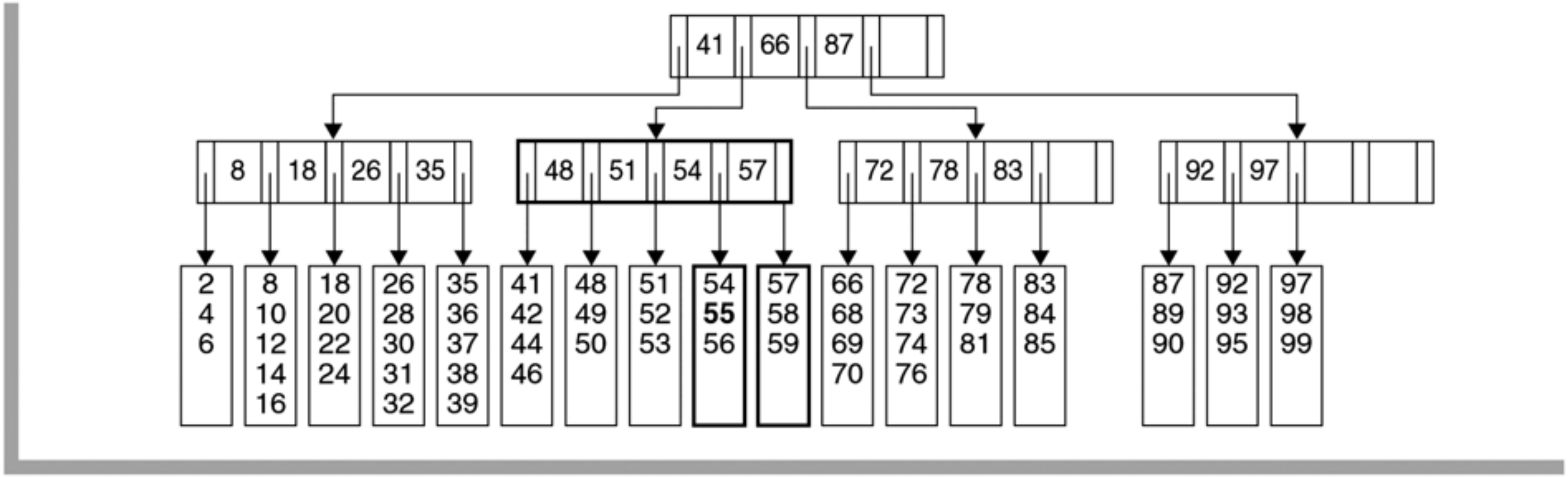


figure 19.86

Insertion of 55 in the B-tree shown in Figure 19.85 causes a split into two leaves.

Insertion into a B-tree

extra node splitting

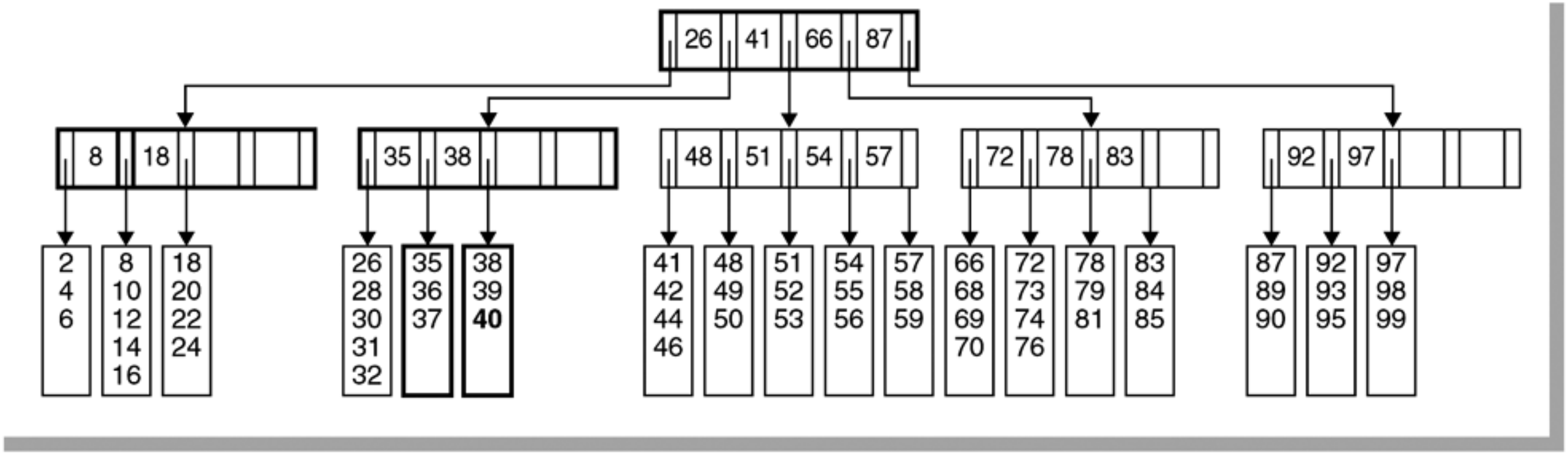


figure 19.87

Insertion of 40 in the B-tree shown in Figure 19.86 causes a split into two leaves and then a split of the parent node.

Deletion in a B-tree

combining two nodes

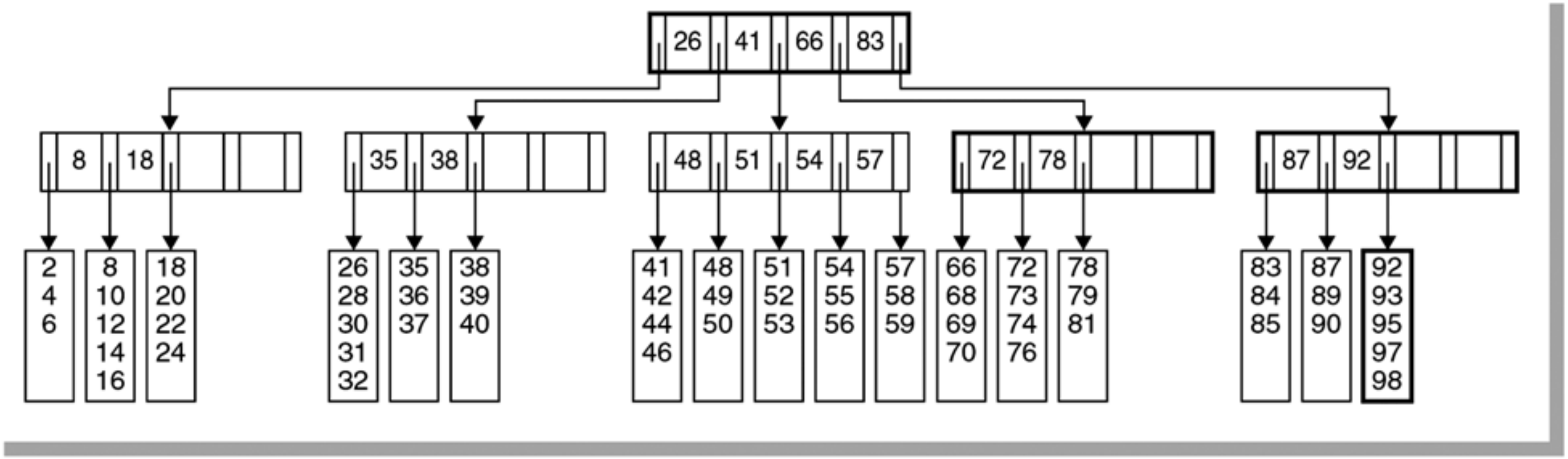


figure 19.88

The B-tree after deletion of 99 from the tree shown in Figure 19.87.

Sketch of a B-tree implementation

```
public class BTree {
   private int M, height;
   private Node root;
   private class Node {...}
   private class Entry {...}

   public BPTree(int order) { M = order; }

   public Object find(Comparable key) {...}
   public void insert(Comparable key, Object data) {...}
   public boolean remove(Comparable key) {...}
}
```

```
class Node {
    Entry[] entry = new Entry[M + 1];
    int size;

    Object find(Comparable key, int ht) {...}
    Node insert(Comparable key, Object pointer, int ht) {...}
    Node split() { ... }
}
```

```
class Entry {
    Comparable key;
    Object pointer;
    Entry(Comparable k, Object p)
      { key = k; pointer = p; }
}
```

find

```
Object find(Comparable key) {
    return root != null ? root.find(key, height) : null;
}
```

```
Object find(Comparable key, int ht) {
    if (ht == 0) {
        for (int i = 0; i < size; i++)
            if (key.compareTo(entry[i].key) == 0)
                return entry[i].pointer;
    } else
        for (int i = 0; i < size; i++)
            if (i + 1 == size || key.compareTo(entry[i + 1].key) < 0)
                return ((Node) entry[i].pointer).find(key, ht - 1);
    return null;
}
```

```
void insert(Comparable key, Object data) {
    if (root == null)
        root = new Node();
    Node t = root.insert(key, data, height);
    if (t != null) { // split root
        Node newRoot = new Node();
        newRoot.entry[0] = new Entry(root.entry[0].key, root);
        newRoot.entry[1] = new Entry(t.entry[0].key, t);
        root = newRoot;
        root.size = 2;
        height++;
    }
}
```

```
Node insert(Comparable key, Object data, int ht) {
    Entry newEntry = new Entry(key, pointer);
    int i;
    if (ht == 0) {
        for (i = 0; i < size; i++)
            if (key.compareTo(entry[i].key) < 0)
                break;
    } else
        for (i = 0; i < size; i++)
            if (i + 1 == size || key.compareTo(entry[i + 1].key) < 0) {
                Node t = ((Node) entry[i++].pointer).
                                insert(key, data, ht - 1);
                if (t == null)
                    return null;
                newEntry.key = t.entry[0].key;
                newEntry.pointer = t;
                break;
            }
    for (int j = size; j > i; j--)
        entry[j] = entry[j - 1];
    entry[i] = newEntry;
    return ++size <= M ? null : split();
}
```

```
Node split() {
    Node t = new Node();
    for (int i = 0, j = M / 2; j <= M; i++, j++)
        t.entry[i] = entry[j];
    t.size = M - M / 2 + 1;
    size = M / 2;
    return t;
}
```

Quote

Alan J. Perlis

"You think you know when you learn,
are more sure when you can write,
even more when you can teach,
but certain when you can program."