Recursion

- **Example 1:** Fibonacci Numbers
  1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

```
public static long fib(int n)
{
        if (n <= 1)
                return n;
        else
                return fib(n-1) + fib(n-2);
}
```

- **Example 2:** Towers of Hanoi

Figure 2.11

Recursive calls that ***rabbit(7)*** generates

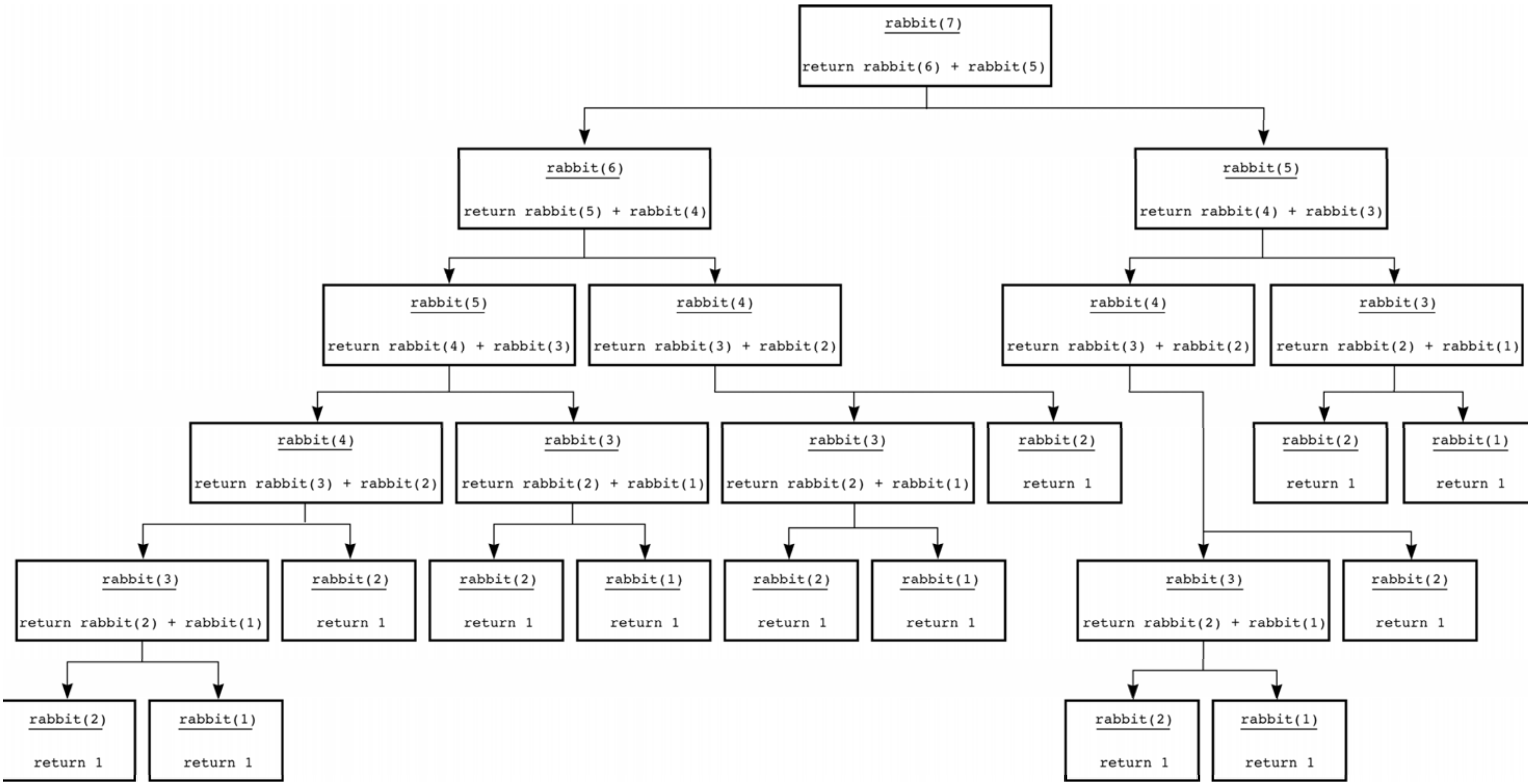


Figure 2.19a and b

a) The initial state; b) move *n* - 1 disks from *A* to *C*

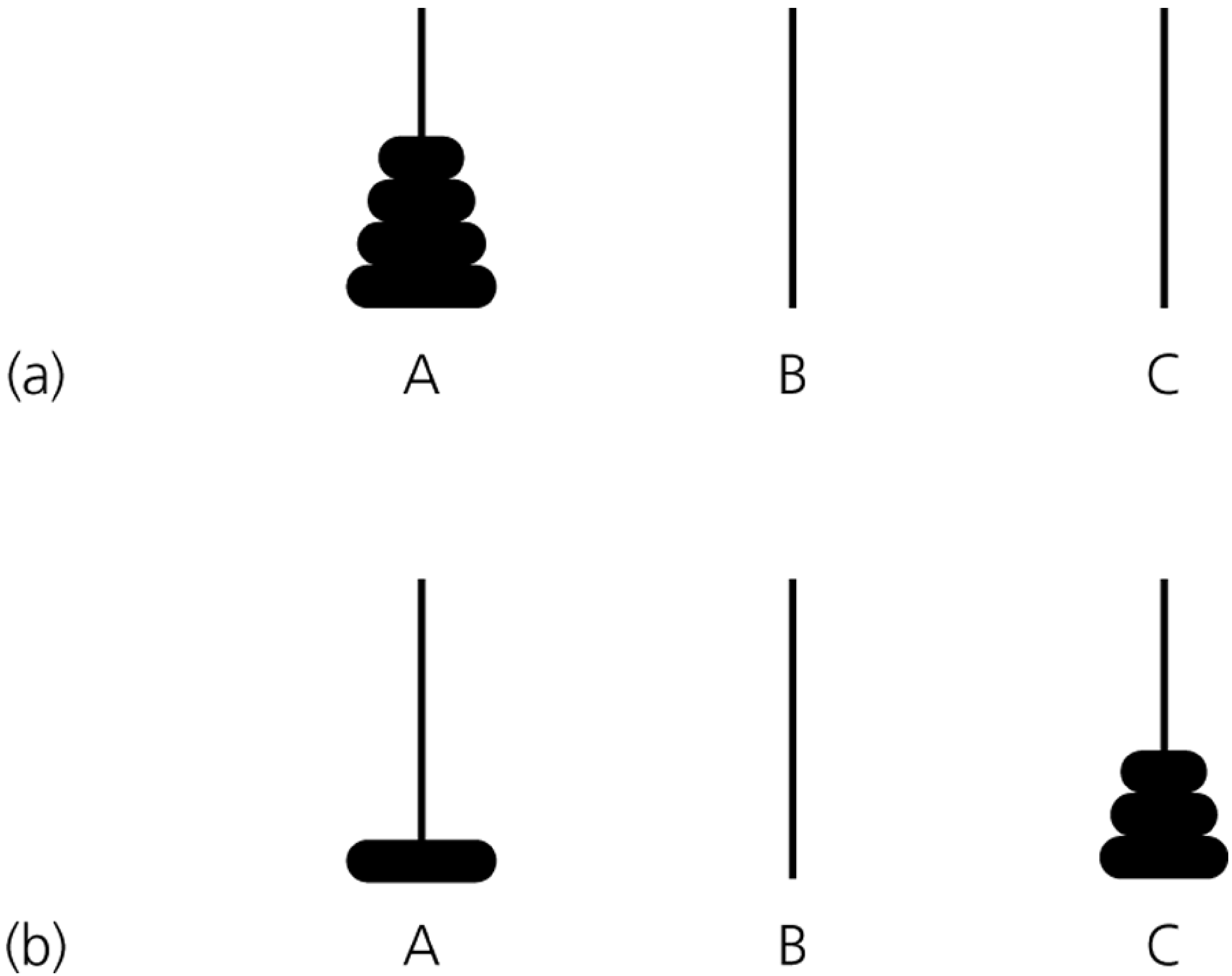
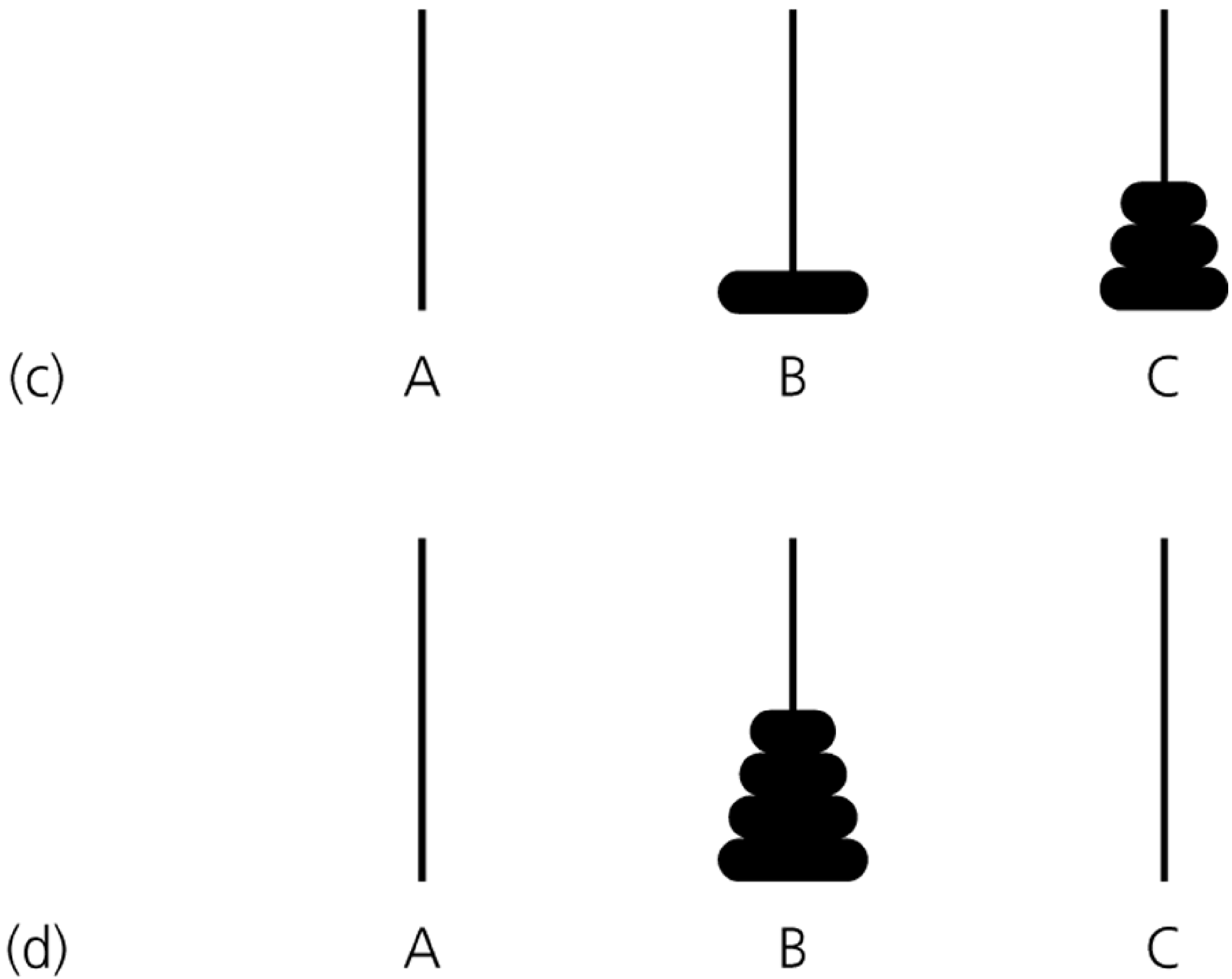


Figure 2.19c and d

c) move one disk from *A* to *B*; d) move *n* - 1 disks from *C* to *B*



Sample output

Move top disk from pole A to pole B
Move top disk from pole A to pole C
Move top disk from pole B to pole C
Move top disk from pole A to pole B
Move top disk from pole C to pole A
Move top disk from pole C to pole B
Move top disk from pole A to pole B

SolveTowers Solution

```
public static void solveTowers(int count, char source,
                         char destination, char spare)
{
  if (count == 1) {
    System.out.println("Move top disk from pole " + source +
                      " to pole " + destination);
  }
  else {
    solveTowers(count-1, source, spare, destination); // X
    solveTowers(1, source, destination, spare);       // Y
    solveTowers(count-1, spare, destination, source); // Z
  }  // end if
}  // end solveTowers
```

Figure 2.20

The order of recursive calls that results from ***solveTowers(3, A, B, C)***

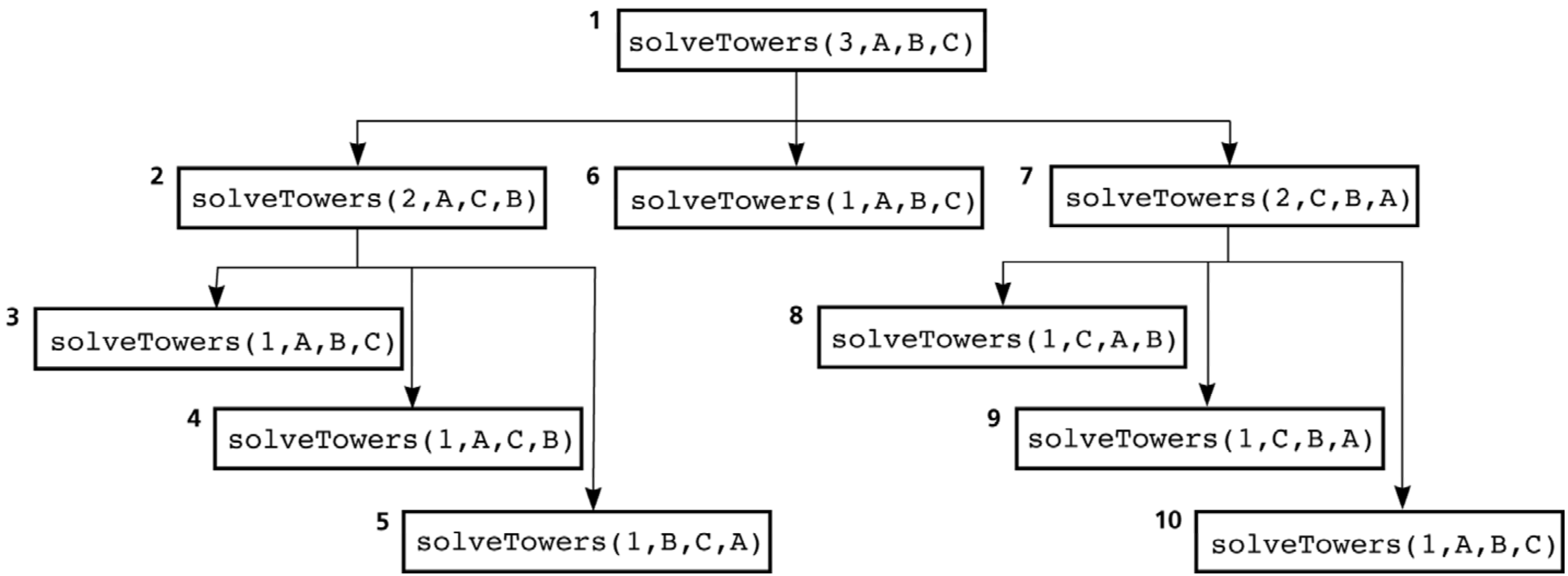
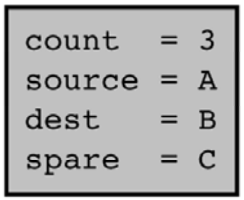


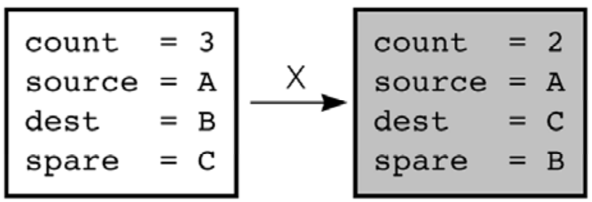
Figure 2.21a

Box trace of ***solveTowers(3, 'A', 'B', 'C')***

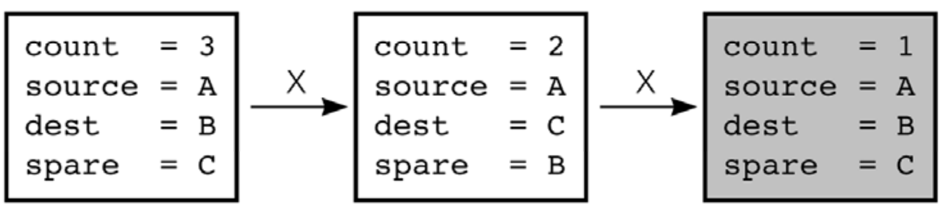
The initial call 1 is made, and `solveTowers` begins execution:



At point X, recursive call 2 is made, and the new invocation of the method begins execution:



At point X, recursive call 3 is made, and the new invocation of the method begins execution:



This is the base case, so a disk is moved, the return is made, and the method continues execution.

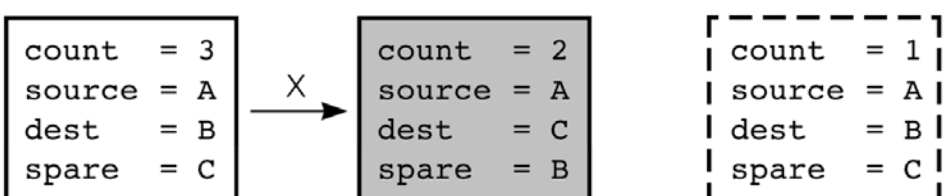
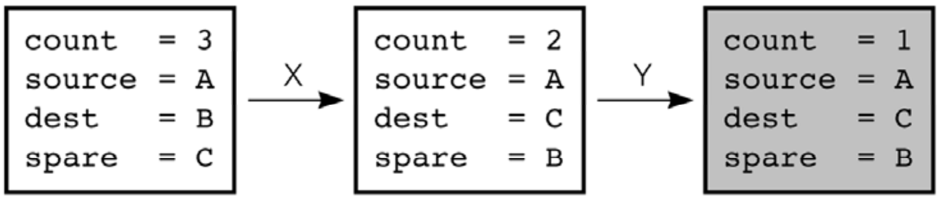


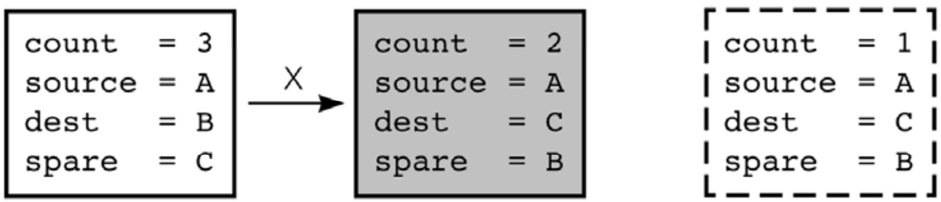
Figure 2.21b

Box trace of ***solveTowers(3, 'A', 'B', 'C')***

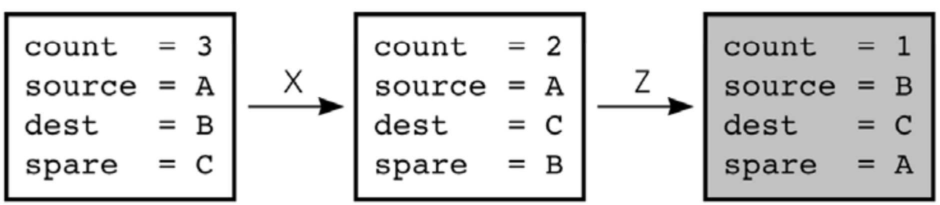
At point Y, recursive call 4 is made, and the new invocation of the method begins execution:



This is the base case, so a disk is moved, the return is made, and the method continues execution.



At point Z, recursive call 5 is made, and the new invocation of the method begins execution:



This is the base case, so a disk is moved, the return is made, and the method continues execution.

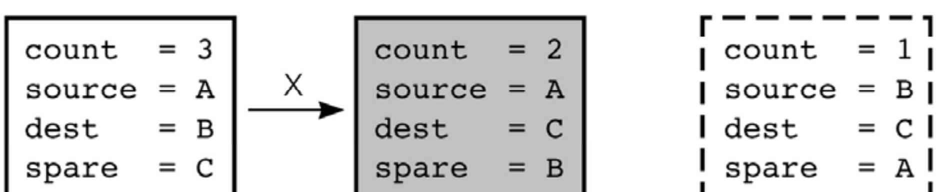


Figure 2.21c

Box trace of ***solveTowers(3, 'A', 'B', 'C')***

This invocation completes, the return is made, and the method continues execution.

```
count  = 3      count  = 2      count  = 1
source = A      source = A      source = B
dest   = B      dest   = C      dest   = C
spare  = C      spare  = B      spare  = A
```

At point Y, recursive call 6 is made, and the new invocation of the method begins execution:

```
count  = 3          count  = 1
source = A    Y     source = A
dest   = B  ----->  dest   = B
spare  = C          spare  = C
```

This is the base case, so a disk is moved, the return is made, and the method continues execution.

```
count  = 3      count  = 1
source = A      source = A
dest   = B      dest   = B
spare  = C      spare  = C
```

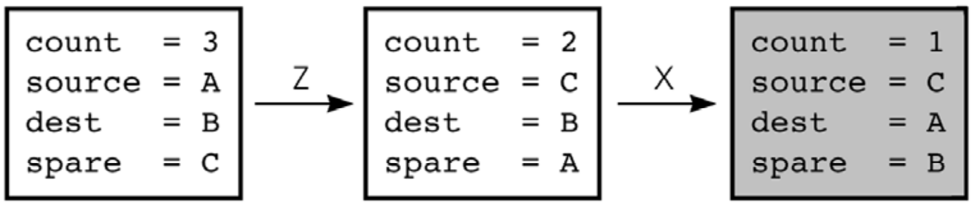
At point Z, recursive call 7 is made, and the new invocation of the method begins execution:

```
count  = 3          count  = 2
source = A    Z     source = C
dest   = B  ----->  dest   = B
spare  = C          spare  = A
```

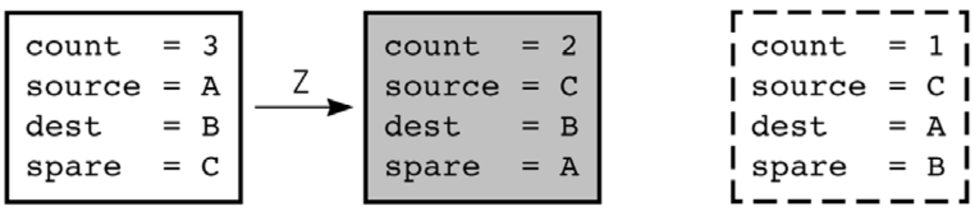
Figure 2.21d

Box trace of ***solveTowers(3, 'A', 'B', 'C')***

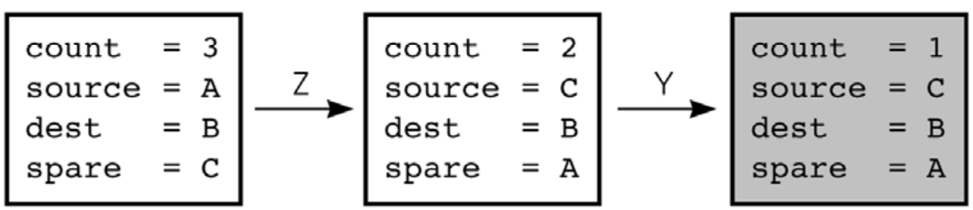
At point X, recursive call 8 is made, and the new invocation of the method begins execution:



This is the base case, so a disk is moved, the return is made, and the method continues execution.



At point Y, recursive call 9 is made, and the new invocation of the method begins execution:



This is the base case, so a disk is moved, the return is made, and the method continues execution.

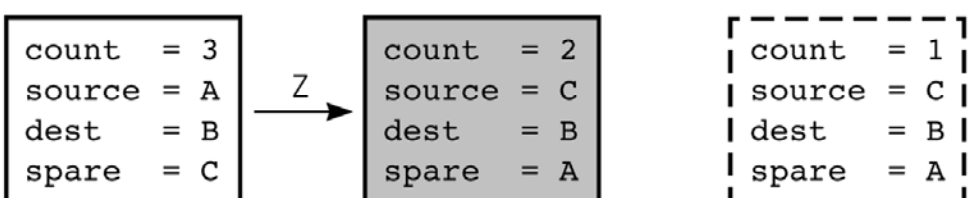
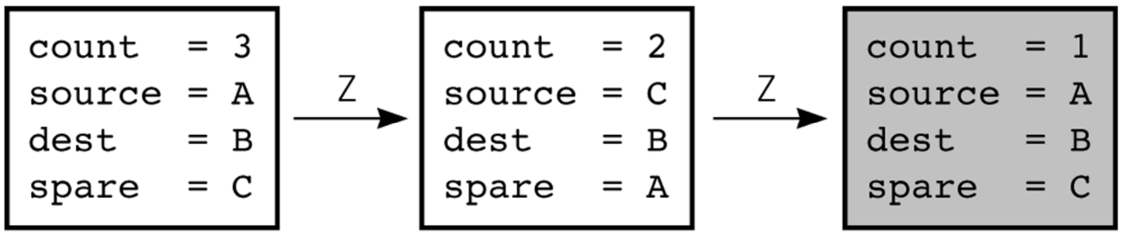


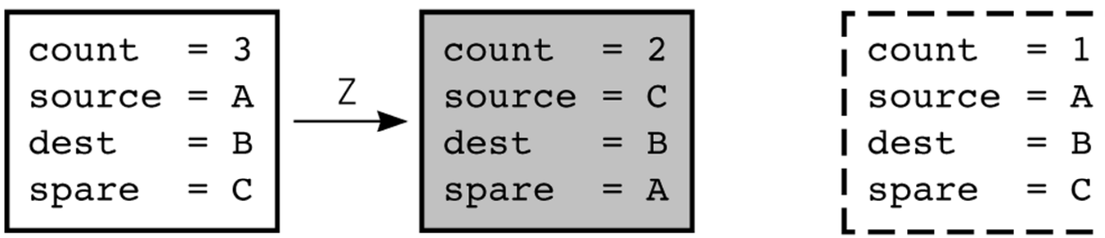
Figure 2.21e

Box trace of ***solveTowers(3, 'A', 'B', 'C')***

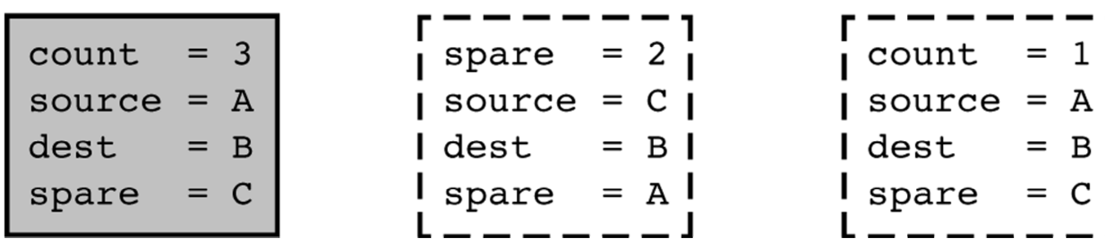
At point Z, recursive call 10 is made, and the new invocation of the method begins execution:



This is the base case, so a disk is moved, the return is made, and the method continues execution.



This invocation completes, the return is made, and the method continues execution.



Binary Trees

```
class BinaryNode
{
   public BinaryNode( )  {  this( null, null, null ); }
   public BinaryNode( Object theElement, BinaryNode lt, BinaryNode rt );
   public static int size( BinaryNode t );   // size of subtree rooted at t
   public static int height( BinaryNode t );
   public void printPreOrder( );
   public void printPostOrder( );
   public void printInOrder( );
   public BinaryNode duplicate( ); // make a duplicate tree and return root
   public Object getElement( );
   public BinaryNode getLeft( );
   public BinaryNode getRight( );
   public void setElement( Object x );
   public void setLeft( BinaryNode t );
   public void setRight( BinaryNode t );

   private Object     element;
   private BinaryNode left;
   private BinaryNode right;
}
```

Binary Trees

```
public class BinaryTree
{
    public BinaryTree( );
    public BinaryTree( Object rootItem );
    public void printPreOrder( );
    public void printInOrder( );
    public void printPostOrder( );
    public void makeEmpty( );
    public boolean isEmpty( );
   /** Forms a new tree from rootItem, t1 and t2. t1 not equal to t2. */
    public void merge( Object rootItem, BinaryTree t1, BinaryTree t2 );
    public int size( );
    public int height( );
    public BinaryNode getRoot( );

    private BinaryNode root;
}
```

Binary Trees

```
public class BinaryTree
{
   static public void main( String [ ] args )
   {
      BinaryTree t1 = new BinaryTree( "1" );   BinaryTree t3 = new BinaryTree( "3" );
      BinaryTree t5 = new BinaryTree( "5" );   BinaryTree t7 = new BinaryTree( "7" );
      BinaryTree t2 = new BinaryTree( );       BinaryTree t4 = new BinaryTree( );
      BinaryTree t6 = new BinaryTree( );
      t2.merge( "2", t1, t3 );   t6.merge( "6", t5, t7 );   t4.merge( "4", t2, t6 );

      System.out.println( "t4 should be perfect 1-7; t2 empty" );
      System.out.println( "----------------" );
      System.out.println( "t4" );
      t4.printInOrder( );
      System.out.println( "----------------" );
      System.out.println( "t2" );
      t2.printInOrder( );
      System.out.println( "----------------" );
      System.out.println( "t4 size: " + t4.size( ) );
      System.out.println( "t4 height: " + t4.height( ) );
   }
}
```

Binary Trees

```
public void printPreOrder( )
  {
     System.out.println( element );       // Node
     if( left != null )  left.printPreOrder( );          // Left
     if( right != null )  right.printPreOrder( );         // Right
  }
  public void printPostOrder( )
  {
     if( left != null )   left.printPostOrder( );         // Left
     if( right != null )   right.printPostOrder( );        // Right
     System.out.println( element );       // Node
  }
  public void printInOrder( )
  {
     if( left != null )  left.printInOrder( );          // Left
     System.out.println( element );       // Node
     if( right != null )  right.printInOrder( );         // Right
  }
```

Binary Trees

```
public void merge( Object rootItem, BinaryTree t1, BinaryTree t2 )
 {
    if( t1.root == t2.root && t1.root != null )  {
        System.err.println( "leftTree==rightTree; merge aborted" );
        return;
    }
   root = new BinaryNode( rootItem, t1.root, t2.root );
    if( this != t1 )    t1.root = null;
    if( this != t2 )    t2.root = null;
 }

public BinaryNode duplicate( )
 {
    BinaryNode root = new BinaryNode( element, null, null );
    if( left != null )    root.left = left.duplicate( );
    if( right != null )  root.right = right.duplicate( );
    return root;                        // Return resulting tree
 }
```

Binary Search Trees

```
// BinarySearchTree class
//
// 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
public class BinarySearchTree
{
   private Comparable elementAt( BinaryNode t ) { return t == null ? null : t.element; }
   protected BinaryNode insert( Comparable x, BinaryNode t )
   protected BinaryNode remove( Comparable x, BinaryNode t )
   protected BinaryNode removeMin( BinaryNode t )
   protected BinaryNode findMin( BinaryNode t )
   private BinaryNode findMax( BinaryNode t )
   private BinaryNode find( Comparable x, BinaryNode t )

   protected BinaryNode root;
}
```

Binary Search Trees

```
public static void main( String [ ] args ) {
        BinarySearchTree t = new BinarySearchTree( );
        final int NUMS = 4000;
        final int GAP  =   37;
        System.out.println( "Checking... (no more output means success)" );
        for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
            t.insert( new Integer( i ) );
        for( int i = 1; i < NUMS; i+= 2 )
            t.remove( new Integer( i ) );
        if( ((Integer)(t.findMin( ))).intValue( ) != 2 ||
            ((Integer)(t.findMax( ))).intValue( ) != NUMS - 2 )
            System.out.println( "FindMin or FindMax error!" );
        for( int i = 2; i < NUMS; i+=2 )
             if( ((Integer)(t.find( new Integer( i ) ))).intValue( ) != i )
                 System.out.println( "Find error1!" );
        for( int i = 1; i < NUMS; i+=2 )
        {
            if( t.find( new Integer( i ) ) != null )
                System.out.println( "Find error2!" );
        }
    }
}
```

Binary Search Trees

```
protected BinaryNode insert( Comparable x, BinaryNode t )   {
        if( t == null )
            t = new BinaryNode( 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;
    }
protected BinaryNode remove( Comparable x, BinaryNode 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 )    {
            t.element = findMin( t.right ).element;
            t.right = removeMin( t.right );
        }
        else
            t = ( t.left != null ) ? t.left : t.right;
        return t;
    }
```

Binary Search Trees

```
protected BinaryNode removeMin( BinaryNode t )
   {
      if( t == null )
         throw new ItemNotFoundException( );
      else if( t.left != null )
      {
         t.left = removeMin( t.left );
         return t;
      }
      else
         return t.right;
   }
```

```
protected BinaryNode findMin( BinaryNode t )
   {
      if( t != null )
         while( t.left != null )
            t = t.left;
      return t;
   }

private BinaryNode
            find( Comparable x, BinaryNode 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
   }
```