How to search in a sorted list

```
public class BinarySearch // Fig 5.11, pg168
{
    public static final int NOT_FOUND = -1;
    public static int binarySearch
              ( Comparable [ ] a, Comparable x )
    {
        int low = 0;
        int high = a.length - 1;
        int mid;
        while( low <= high )
        {
            mid = ( low + high ) / 2;
            if( a[ mid ].compareTo( x ) < 0 )
                low = mid + 1;
            else if( a[ mid ].compareTo( x ) > 0 )
                high = mid - 1;
            else
                return mid;
        }
        return NOT_FOUND;    // NOT_FOUND = -1
    }
```

```
    // Test program
    public static void main( String [ ] args )
    {
        int SIZE = 8;
        Comparable [ ] a = new Integer [ SIZE ];
        for( int i = 0; i < SIZE; i++ )
            a[ i ] = new Integer( i * 2 );

        for( int i = 0; i < SIZE * 2; i++ )
            System.out.println( "Found " + i + " at " +
                  binarySearch( a, new Integer( i ) ) );
    }
}
```

Stacks and Queues

```
public interface Stack
{ // Fig 6.21, p206
    public Object push( Object x );
    public Object pop( );
    public boolean isEmpty( );
}

public interface Queue
{ // Fig 6.23, p209
    public boolean isEmpty( );
    public void enqueue( Object x );
    public Object dequeue( );
}
```

Stacks & Queues – Implementations

```
public class Stack implements Serializable
{  // Fig 16.28, p532
   public Object push( Object x )
   {
      items.add( x );
      return x;
   }
   public Object pop( )
   {
      if( isEmpty( ) )
         throw new EmptyStackException( );
      return items.remove( items.size( ) - 1 );
   }
   public boolean isEmpty( )
      {  return size( ) == 0;  }

   private ArrayList items;
   // LinkedList????
}
```

```
public class ListQueue implements Queue
{  // Fig 16.25, p529
   public boolean isEmpty( )
   {  return front == null; }
   public void enqueue( Object x )
   {  if( isEmpty( ) )
         back = front = new ListNode( x );
      else              // Regular case
         back = back.next = new ListNode( x );
   }
   public Object dequeue( )
   {  if( isEmpty( ) )
         throw new UnderflowException( "" );
      Object returnValue = front.element;
      front = front.next;
      return returnValue;
   }

   private ListNode front;
   private ListNode back;
}
```

Stacks: Application 1

- Check balanced parentheses
  - (())()(()(()))
  - ((())()))()(()

```
While (expr.nextToken())
{
   if next token is "("
          push "(" on stack;
   else
          if stack is not empty
                 pop "(" from stack;
          else report error;
}
If stack is not empty
          report error;
```

Stacks: Application 2

Evaluate Postfix Expressions

1 2 3 + *

= (1* (2 + 3))

4 1 2 2 3 * ^ + -1 * +

= ?

```
While (expr.nextToken())
{
   if next token is an operand
      push operand on stack;
   else if next token is an operator Op
      {
         pop Val1 from stack;
         pop Val2 from stack;
         compute Val1 Op Val2;
         push result on stack;
      }
   if stack has only one item
      pop value and return as Value of expr;
   else report error;
}
```

Stacks – Applications 3

- Convert Infix Expressions to Postfix

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

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

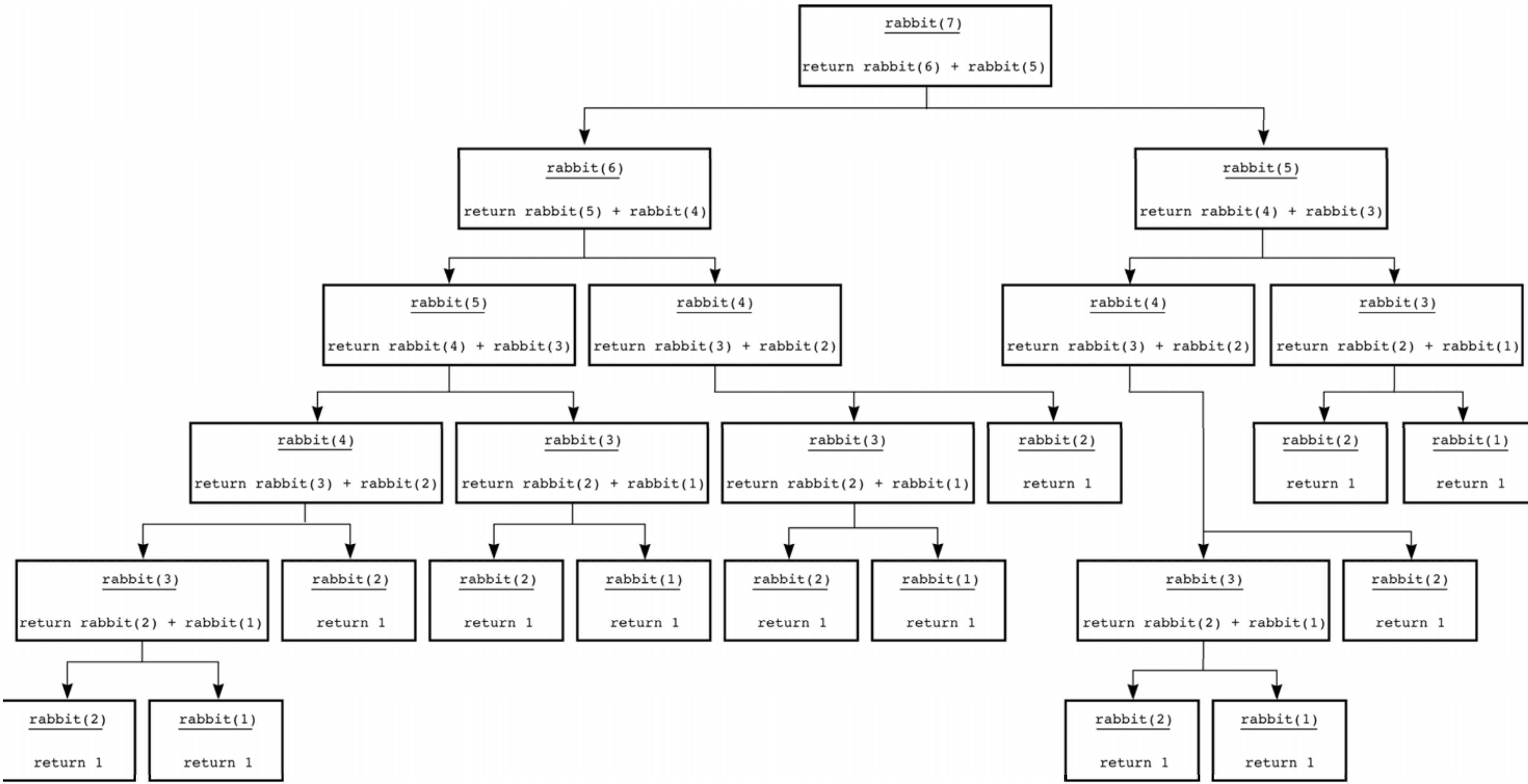


Figure 2.19a and b

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

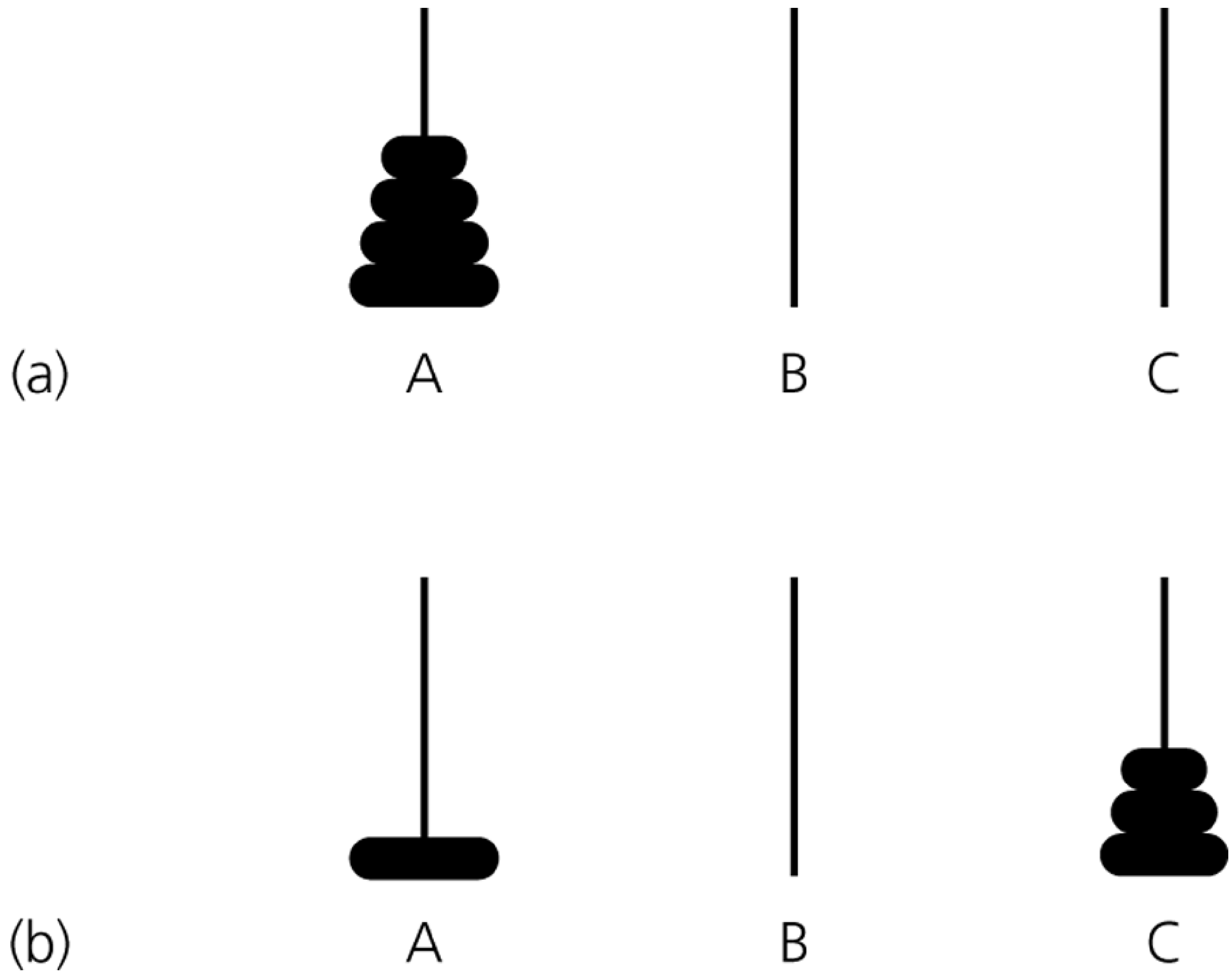
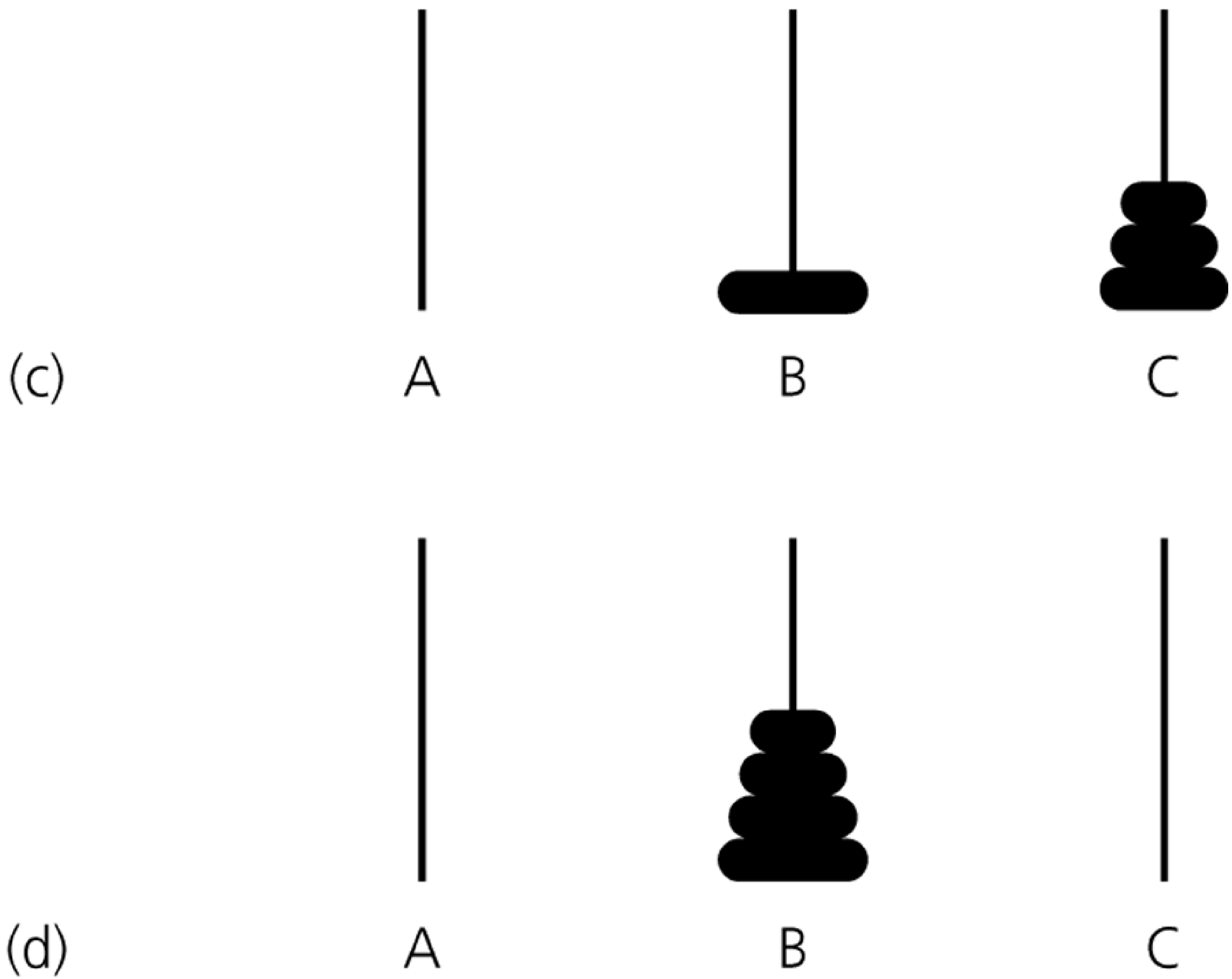


Figure 2.19c and d

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)***

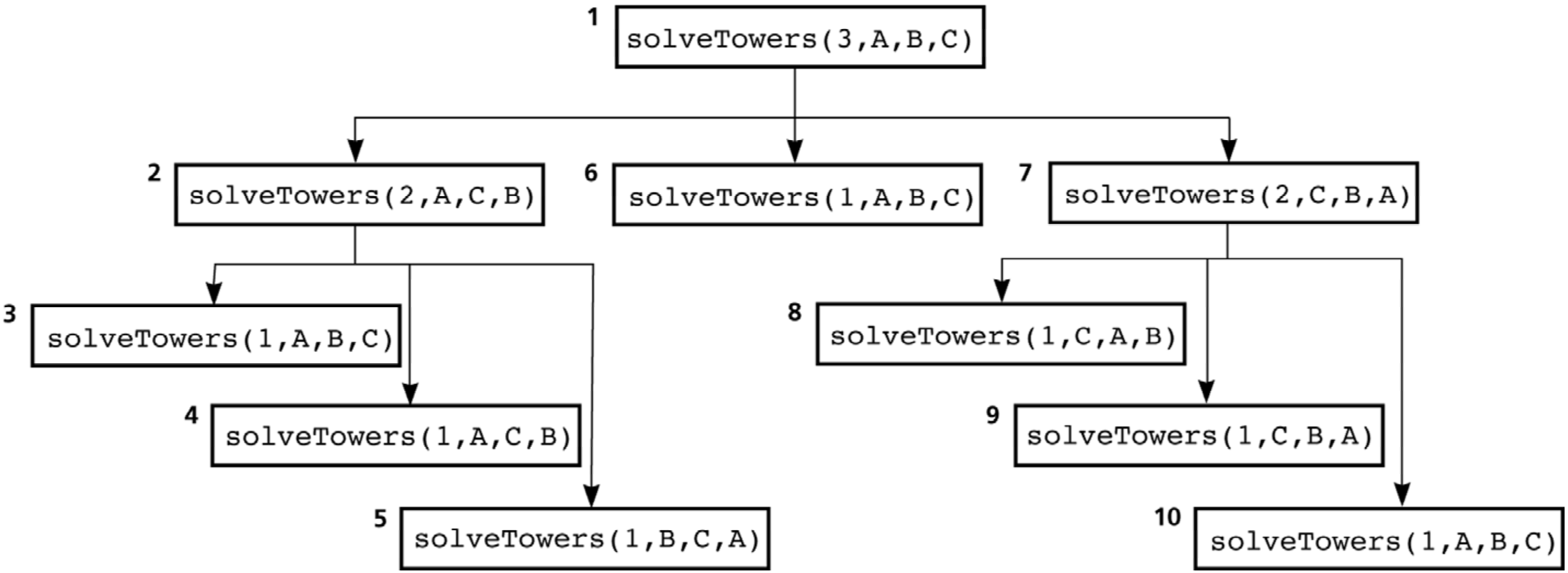
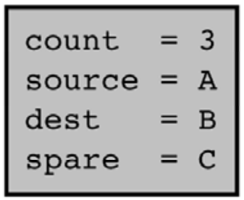


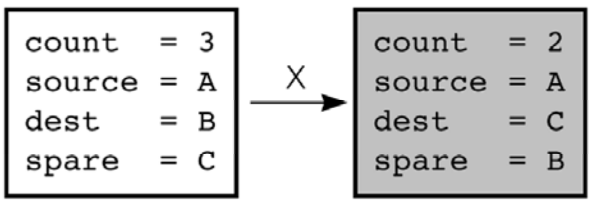
Figure 2.21a

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

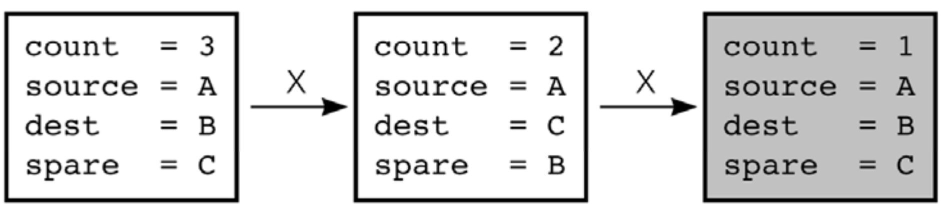
The initial call 1 is made, and `solveTowers` begins execution:



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.

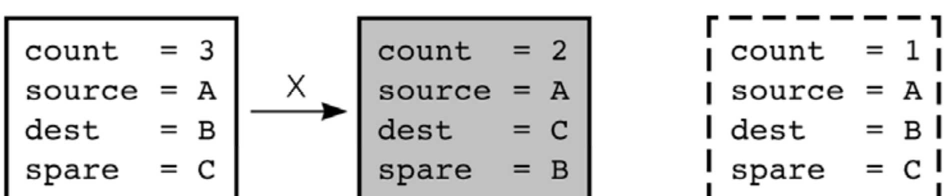
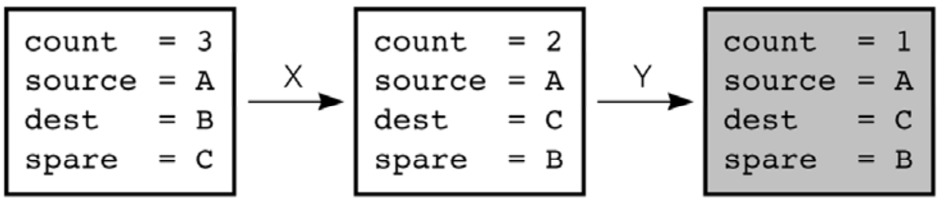


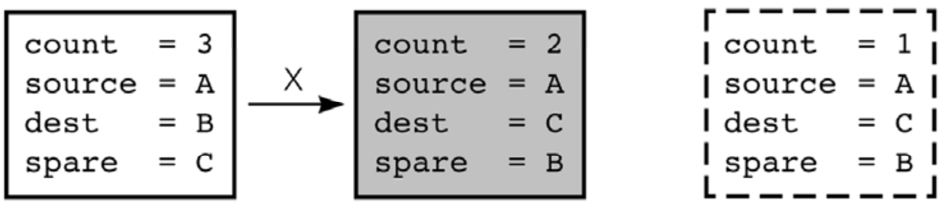
Figure 2.21b

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

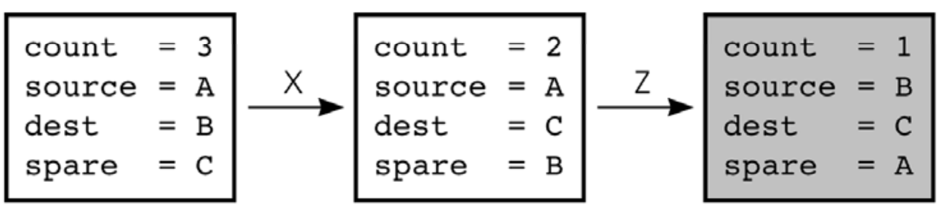
At point Y, recursive call 4 is made, and the new invocation of the method begins execution:



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.

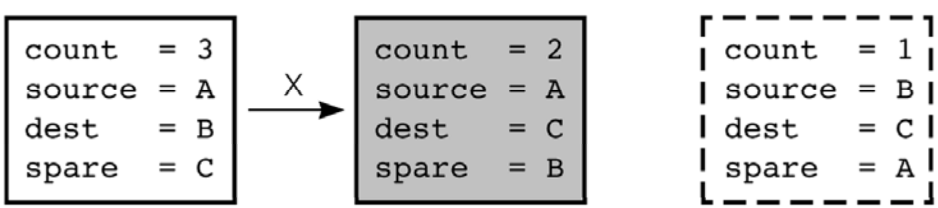
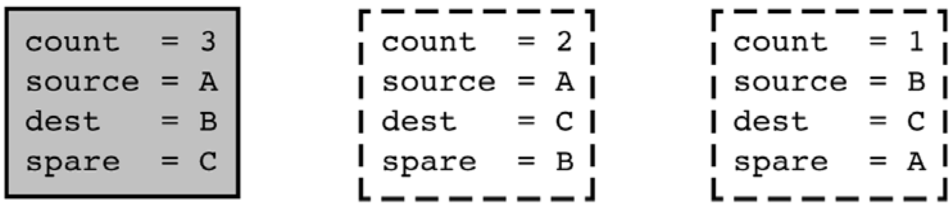


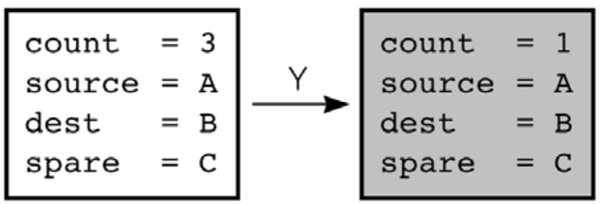
Figure 2.21c

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

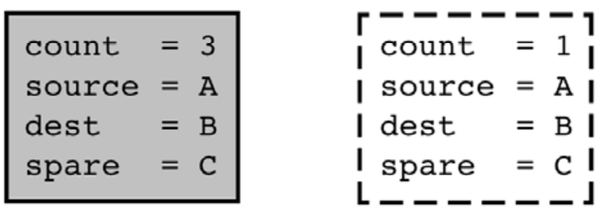
This invocation completes, the return is made, and the method continues execution.



At point Y, recursive call 6 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 7 is made, and the new invocation of the method begins execution:

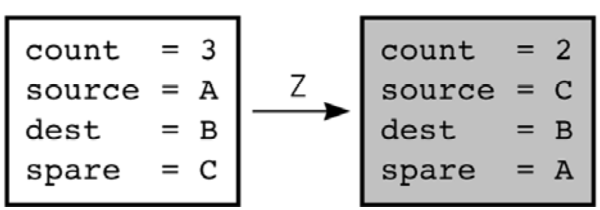
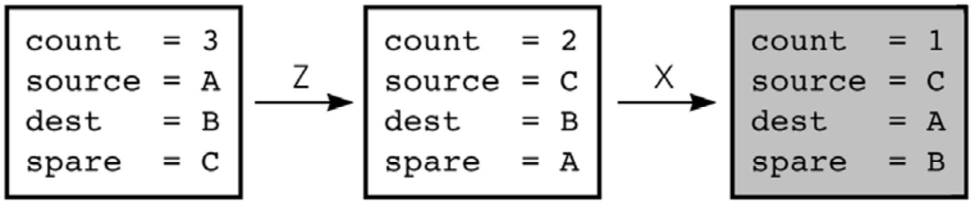


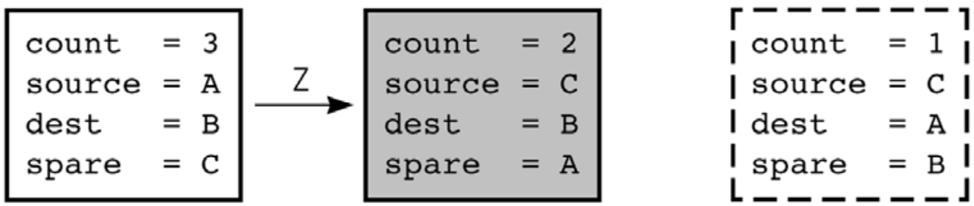
Figure 2.21d

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

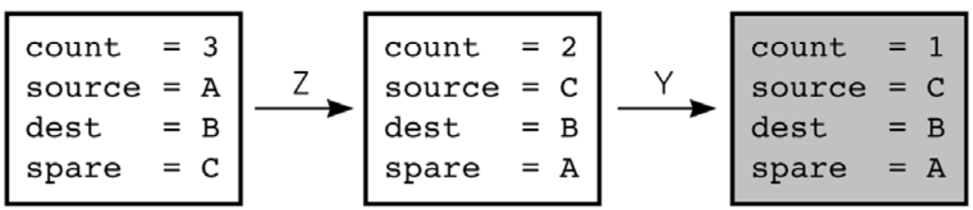
At point X, recursive call 8 is made, and the new invocation of the method begins execution:



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.

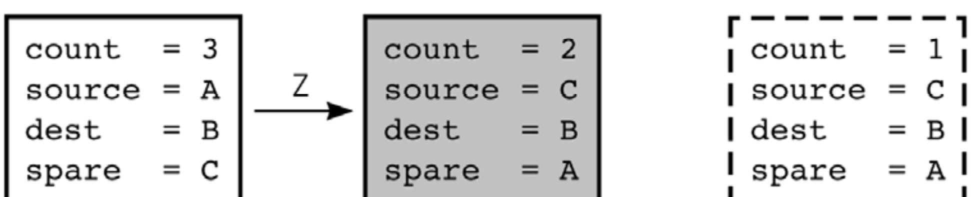
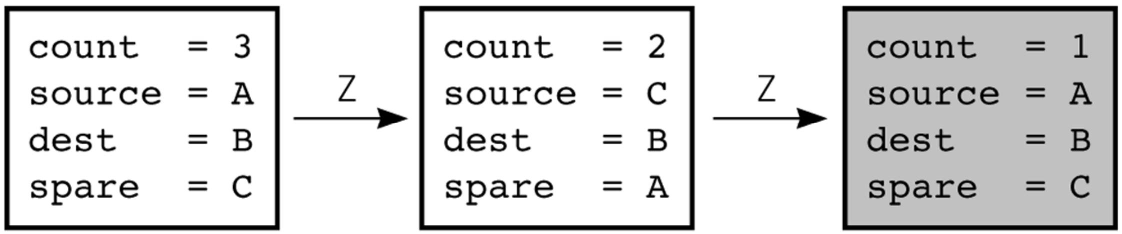


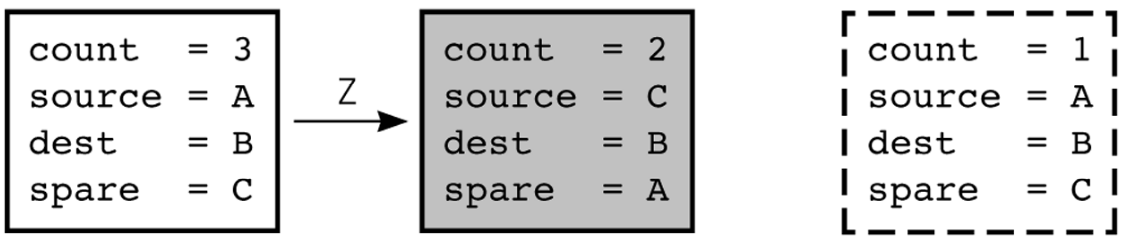
Figure 2.21e

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.