are used, towers 1 and 2 must have a capacity of n disks each, while tower 3 must have a capacity of $n-1$. Therefore, we need space for a total of $3n-1$ disks. As our earlier analysis has shown, the time complexity of the Towers of Hanoi problem is exponential in n. So using a reasonable amount of computer time, the problem can be solved only for small values of n (say $n \leq 30$). For these small values of n, the difference in space required by the array and linked representations is sufficiently small that either may be used. Since the array implementations of a stack run faster than the linked implementations, we use an array implementation.

The code of Program 9.8 uses array stacks. `towersOfHanoi(n)` is just a preprocessor for the recursive method `showTowerStates`, which is modeled after the method of Program 9.7. The preprocessor creates the three stacks `tower[1:3]` that will store the states of the three towers. The disks are numbered 1 (smallest) through n (largest). Since the disks are modeled as integers, the disk numbers cannot be stored on a stack unless their data type is converted from `int` to a wrapper type for `int`. We can use either of the types `Integer` or `MyInteger` for this purpose. The initial configuration has all n disks in `tower[1]`; the remaining two towers have no disk. After constructing this initial configuration, the preprocessor invokes the method `showTowerStates`.

9.5.3 Rearranging Railroad Cars

Problem Description

A freight train has n railroad cars. Each is to be left at a different station. Assume that the n stations are numbered 1 through n and that the freight train visits these stations in the order n through 1. The railroad cars are labeled by their destination. To facilitate removal of the railroad cars from the train, we must reorder the cars so that they are in the order 1 through n from front to back. When the cars are in this order, the last car is detached at each station. We rearrange the cars at a shunting yard that has an *input track*, an *output track*, and k holding tracks between the input and output tracks. Figure 9.6(a) shows a shunting yard with $k = 3$ holding tracks $H1$, $H2$, and $H3$. The n cars of the freight train begin in the input track and are to end up in the output track in the order 1 through n from right to left. In Figure 9.6(a), $n = 9$; the cars are initially in the order 5, 8, 1, 7, 4, 2, 9, 6, 3 from back to front. Figure 9.6(b) shows the cars rearranged in the desired order.

Solution Strategy

To rearrange the cars, we examine the cars on the input track from front to back. If the car being examined is the next one in the output arrangement, we move it directly to the output track. If not, we move it to a holding track and leave it there until it is time to place it in the output track. The holding tracks operate in a LIFO manner as cars enter and leave these tracks from the top. When rearranging cars, only the following moves are permitted:

```
public class TowersOfHanoiShowingStates
{
   // data member
   private static ArrayStack [] tower;  // the towers are tower[1:3]

   /** n disk Towers of Hanoi problem */
   public static void towersOfHanoi(int n)
   {// Preprocessor for showTowerStates

      // create three stacks, tower[0] is not used
      tower = new ArrayStack[4];
      for (int i = 1; i <= 3; i++)
         tower[i] = new ArrayStack();

      for (int d = n; d > 0; d--) // initialize
         tower[1].push(new Integer(d));  // add disk d to tower 1

      // move n disks from tower 1 to 2 using 3 as
      // intermediate tower
      showTowerStates(n, 1, 2, 3);
   }

   public static void showTowerStates(int n, int x, int y, int z)
   {// Move the top n disks from tower x to tower y.
    // Use tower z for intermediate storage.
       if (n > 0)
       {
          showTowerStates(n-1, x, z, y);
          Integer d = (Integer) tower[x].pop();  // move d from top
                                                 // of tower x to
          tower[y].push(d);                      // top of tower y
          System.out.println("Move disk " + d + " from tower "
                             + x + " to top of tower " + y);
          // output statement should be replaced by showState() when
          // showState method has been implemented
          showTowerStates(n-1, z, y, x);
       }
   }
}
```

Program 9.8 Towers of Hanoi using stacks

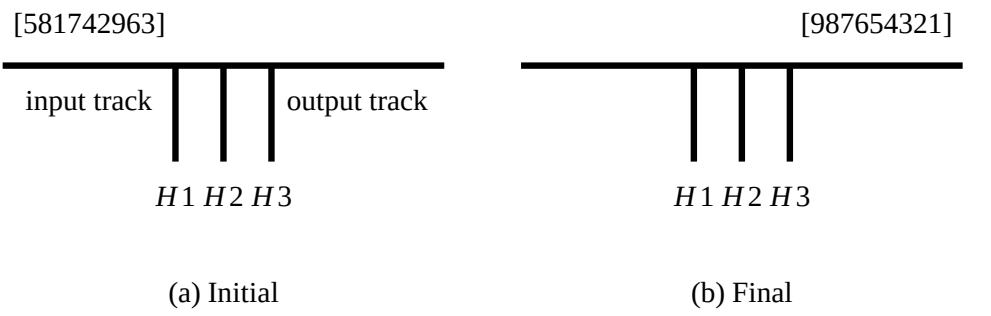

Figure 9.6 A three-track example

- A car may be moved from the front (i.e., right end) of the input track to the top of one of the holding tracks or to the left end of the output track.
- A car may be moved from the top of a holding track to the left end of the output track.

Consider the input arrangement of Figure 9.6(a). Car 3 is at the front and cannot be output yet, as it is to be preceded by cars 1 and 2. So car 3 is detached and moved to the holding track $H1$. The next car, car 6, is also to be moved to a holding track. If car 6 is moved to $H1$, the rearrangement cannot be completed because car 3 will be below car 6. However, car 3 is to be output before car 6 and so must leave $H1$ before car 6 does. So car 6 is put into $H2$. The next car, car 9, is put into $H3$ because putting it into either $H1$ or $H2$ will make it impossible to complete the rearrangement. *Notice that whenever the car labels in a holding track are not in increasing order from top to bottom, the rearrangement cannot be completed.* The current state of the holding tracks is shown in Figure 9.7(a).

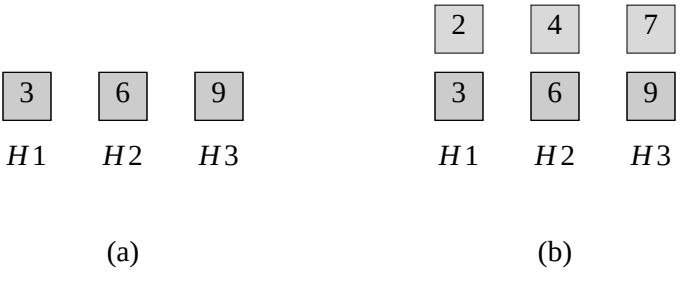

Figure 9.7 Track states

Car 2 is considered next. It can be moved into any of the holding tracks while satisfying the requirement that car labels in any holding track be in increasing order, but moving it to $H1$ is preferred. If car 2 is moved to $H3$, then we have no place to move cars 7 and 8. If we move it to $H2$, then the next car, car 4, will have to be moved to $H3$ and we will have no place for cars 5, 7, and 8. *The least restrictions on future car placement arise when the new car u is moved to the holding track that has at its top a car with smallest label v such that $v > u$.* We will use this **assignment rule** to select the holding track.

When car 4 is considered, the cars at the top of the three holding tracks are 2, 6, and 9. Using our assignment rule, car 4 is moved to $H2$. Car 7 is then moved to $H3$. Figure 9.7(b) shows the current state of the holding tracks. The next car, car 1, is moved to the output track. It is now time to move car 2 from $H1$ to the output track. Next car 3 is moved from $H1$, and then car 4 is moved from $H2$. No other cars can be moved to the output at this time.

The next input car, car 8, is moved to $H1$. Then car 5 is moved from the input track to the output track. Following this move, car 6 is moved from $H2$. Then car 7 is moved from $H3$, car 8 from $H1$, and car 9 from $H3$.

While three holding tracks are sufficient to rearrange the cars from the initial ordering of Figure 9.6(a), other initial arrangements may need more tracks. For example, the initial arrangement 1, n, $n-1$, ..., 2 requires $n-1$ holding tracks.

Java Implementation

To implement the preceding rearrangement scheme, we define the class `RailroadWithStacks`. This class uses k array stacks, `track[1:k]`, to represent the k holding tracks. We use array stacks because they are faster than linked stacks; and regardless of whether we use array or linked stacks, the space required by the k stacks is expected to be well within the capacity of even the most modest computers. Program 9.9 gives the data members of `RailroadWithStacks`.

```
private static ArrayStack [] track;  // array of holding tracks
private static int numberOfCars;
private static int numberOfTracks;
private static int smallestCar; // smallest car in any holding track
private static int itsTrack;    // holding track with car smallestCar
```

Program 9.9 Data members of `RailroadWithStacks`

The method `railroad` (Program 9.10) determines a sequence of moves that results in rearranging cars with initial ordering `inputOrder[1:theNumberOfCars]` using at most `theNumberOfTracks` holding tracks. If such a sequence does not exist, `railroad` returns `false`. Otherwise, it returns `true`.

```
/** rearrange railroad cars beginning with the initial order
  * inputOrder[1:theNumberOfCars]
  * @return true if successful, false if impossible. */
public static boolean railroad(int [] inputOrder,
                      int theNumberOfCars, int theNumberOfTracks)
{
   numberOfCars = theNumberOfCars;
   numberOfTracks = theNumberOfTracks;

   // create stacks track[1:numberOfTracks] for use as holding tracks
   track = new ArrayStack [numberOfTracks + 1];
   for (int i = 1; i <= numberOfTracks; i++)
      track[i] = new ArrayStack();

   int nextCarToOutput = 1;
   smallestCar = numberOfCars + 1;  // no car in holding tracks

   // rearrange cars
   for (int i = 1; i <= numberOfCars; i++)
      if (inputOrder[i] == nextCarToOutput)
      {// send car inputOrder[i] straight out
          System.out.println("Move car " + inputOrder[i]
                             + " from input track to output track");
          nextCarToOutput++;

          // output from holding tracks
          while (smallestCar == nextCarToOutput)
          {
             outputFromHoldingTrack();
             nextCarToOutput++;
          }
      }
      else
      // put car inputOrder[i] in a holding track
         if (!putInHoldingTrack(inputOrder[i]))
            return false;

   return true;
}
```

Program 9.10 The method `RailroadWithStacks.railroad`

Method `railroad` begins by creating an array `track` of stacks. `track[i]` represents holding track `i`, $1 \leq$ `i` $\leq$ `numberOfTracks`. The `for` loop maintains the invariant: *at the start of this loop, the car with label* `nextCarToOutput` *is not in a holding track.*

In iteration `i` of the `for` loop, car `inputOrder[i]` is moved from the input track. This car is to move to the output track only if `inputOrder[i]` equals `nextCarToOutput`. If car `inputOrder[i]` is moved to the output track, `nextCarToOutput` increases by one, and it may be possible to move one or more of the cars in the holding tracks. These cars are moved to the output by the `while` loop. If car `inputOrder[i]` cannot be moved to the output, then no car can be so moved. Consequently, car `inputOrder[i]` is added to a holding track using the stated track assignment rule.

Programs 9.11 and 9.12, respectively, give the methods `outputFromHoldingTrack` and `putInHoldingTrack` utilized by `railroad`. `outputFromHoldingTrack` outputs instructions to move a car from a holding track to the output track. It also updates `smallestCar` and `itsTrack`. The method `putInHoldingTrack` puts car `c` into a holding track using the stated track assignment rule. It also outputs instructions to move the car to the chosen holding track and updates `smallestCar` and `itsTrack` if necessary.

```
/** output the smallest car from the holding tracks */
private static void outputFromHoldingTrack()
{
   // remove smallestCar from itsTrack
   track[itsTrack].pop();
   System.out.println("Move car " + smallestCar + " from holding "
                      + "track " + itsTrack + " to output track");

   // find new smallestCar and itsTrack by checking top of all stacks
   smallestCar = numberOfCars + 2;
   for (int i = 1; i <= numberOfTracks; i++)
      if (!track[i].empty() &&
          ((Integer) track[i].peek()).intValue() < smallestCar)
      {
         smallestCar = ((Integer) track[i].peek()).intValue();
         itsTrack = i;
      }
}
```

Program 9.11 The method `RailroadWithStacks.outputFromHoldingTrack`

```
/** put car c into a holding track
  * @return false iff there is no feasible holding track for this car */
private static boolean putInHoldingTrack(int c)
{
   // find best holding track for car c
   // initialize
   int bestTrack = 0,               // best track so far
       bestTop = numberOfCars + 1;  // top car in bestTrack

   // scan tracks
   for (int i = 1; i <= numberOfTracks; i++)
      if (!track[i].empty())
      {// track i not empty
          int topCar = ((Integer) track[i].peek()).intValue();
          if (c < topCar && topCar < bestTop)
          {
             // track i has smaller car at top
             bestTop = topCar;
             bestTrack = i;
          }
      }
      else // track i empty
         if (bestTrack == 0) bestTrack = i;

   if (bestTrack == 0) return false; // no feasible track

   // add c to bestTrack
   track[bestTrack].push(new Integer(c));
   System.out.println("Move car " + c + " from input track "
                      + "to holding track " + bestTrack);

   // update smallestCar and itsTrack if needed
   if (c < smallestCar)
   {
       smallestCar = c;
       itsTrack = bestTrack;
   }
   return true;
}
```

Program 9.12 The method `RailroadWithStacks.putInHoldingTrack`

Complexity

For the time complexity of `railroad` (Program 9.10), we first observe that both `outputFromHoldingTrack` and `putInHoldingTrack` have complexity O(`numberOfTracks`). Since at most `numberOfCars-1` cars can be output from the `while` loop of `railroad` and at most `numberOfCars-1` put into holding tracks from the `else` clause, the total time spent in methods `outputFromHoldingTrack` and `putInHoldingTrack` is O(`numberOfTracks` $*$ `numberOfCars`). The remainder of the `for` loop of `railroad` takes Θ(`numberOfCars`) time. So the overall complexity of Program 9.10 is O(`numberOfTracks` $*$ `numberOfCars`). This complexity can be reduced to O(`numberOfCars`$*\log($`numberOfTracks`$)$) by using a balanced binary search tree (such as an AVL tree) to store the labels of the cars at the top of the holding tracks (see Chapter 16). When a balanced binary search tree is used in this way, methods `outputFromHoldingTrack` and `putInHoldingTrack` can be rewritten to have complexity $O(\log($`numberOfTracks`$))$. The use of a balanced binary search tree for this application is recommended only when `numberOfTracks` is large.

9.5.4 Switch Box Routing

Problem Description

In the switch box routing problem, we are given a rectangular routing region with pins at the periphery. Pairs of pins are to be connected together by laying a metal path between the two pins. This path is confined to the routing region and is called a wire. If two wires intersect, an electrical short occurs. So wire intersections are forbidden. Each pair of pins that is to be connected is called a **net**. We are to determine whether the given nets can be routed with no intersections. Figure 9.8(a) shows a sample switch box instance with eight pins and four nets. The nets are (1, 4), (2, 3), (5, 6), and (7, 8). The wire routing of Figure 9.8(b) has a pair of intersecting wires (those for nets (1, 4) and (2, 3)), whereas the routing of Figure 9.8(c) has no intersections. Since the four nets can be routed with no intersections, the given switch box is a **routable switch box**. (In practice, we also require a minimum separation between adjacent wires. We ignore this additional requirement here.) Our problem is to input a switch box routing instance and determine whether it is routable.

While the wires in both Figures 9.8(b) and (c) are composed of straight line segments parallel to the x- and y-axes, segments that are not parallel to these axes as well as segments that are not straight lines are permissible.

Solution Strategy

To solve the switch box routing problem, we note that when a net is connected, the wire partitions the routing region into two regions. The pins that fall on the boundary of a partition do not depend on the wire path, but only on the pins of the net that was routed. For instance, when net (1, 4) is routed, we get two regions.