CS1Ah Lecture Note 20

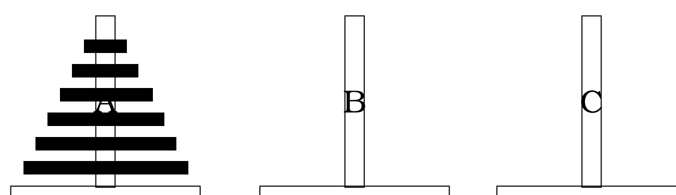
Recursion and stacks

In this lecture note we discuss a programming technique called *recursion* and the related data structure called the *stack*. We have seen many examples of programs where one method invokes another, which invokes another, which invokes another, and so on. One question which we might eventually ask is "can a method in a Java program invoke *itself*?" The answer to this question is "Yes". If a method includes an invocation of itself then it is said to be defined recursively. Similarly, a class `C` would be said to have a recursive description if it had fields which were themselves objects of class `C`.

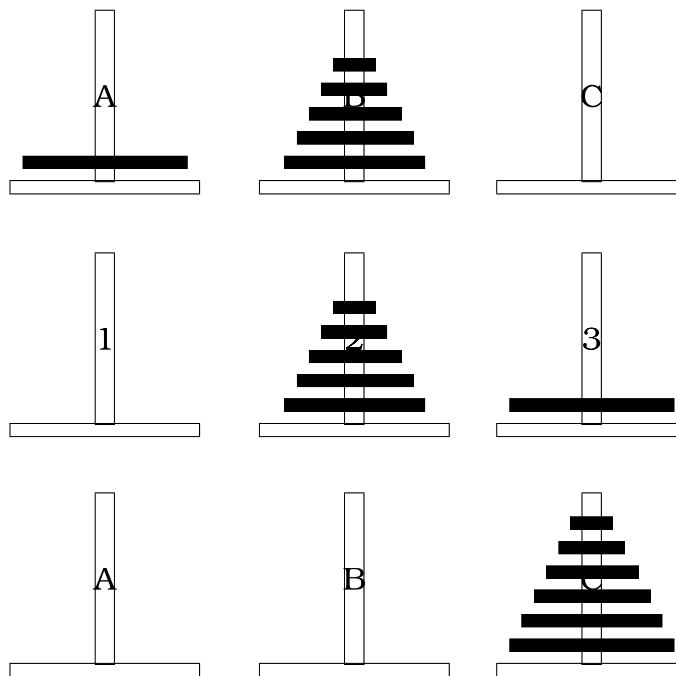
When a method (the *caller*) invokes another (the *callee*) it is because the caller is delegating some of its work to the callee. The caller is suspended until the callee has completed its work, at which point the caller resumes its work from the point where it was suspended. The same reasoning applies even if the caller and the callee are the same method (and therefore the invocation is a recursive invocation). When the caller is suspended, the values held in its local variables must be stored until it is resumed. This information is stored on a data structure in computer memory called a *stack*. The storage space associated with a method is called an *activation record* and thus a stack of these records is built up as one method invokes another. We say more about stacks later. Before then we consider a problem for which a recursive solution seems well suited.

20.1 The Towers of Hanoi

The *Towers of Hanoi* problem was invented by the French mathematician Edouard Lucas and sold as a toy in 1883. The problem is to transfer a tower of discs from one peg to another moving one disc at a time and never placing a disc on top of a smaller one. (We have to assume that entertainment was much harder to come by in 1883 than it is today.) Here is the starting state of the problem, with all of the discs in order on peg A.



The problem is solved by moving every disc except the largest one onto the spare peg; then move the largest across; then move the others back on top. In the case where we are moving six discs from peg A to peg C we must reach an intermediate stage where we have only the largest disc on peg A and we have five discs on peg B. At this point we can move the largest disc on to peg C and we then move the five discs on top of it (in many steps, observing the rules that we can move only one disc at a time and that we can never place a larger disc on top of a smaller one).



Of course, moving every disc but the largest is nothing other than a smaller instance of the problem so the solution naturally lends itself to a *recursive* description.

We now consider a Java method, `hanoi(int n, char src, char dst)`, which prints the moves required to transfer n discs from source peg *src* to destination peg *dst*. A call of the `hanoi()` method is either trivial (we can move the disk immediately) or it sets up three recursive calls: moving $n-1$ discs out of the way, moving a single disk, moving $n-1$ discs back on top. To compute the letter of the spare peg (on line 5) we subtract the integer values of the other two pegs from the integer sum of the pegs.

```
1     static void hanoi(int n, char src, char dst) {
2       if (n == 1) System.out.println
3         ("Move disk from " + src + " to " + dst);
4       else {
5         char spare =
6           (char) (('A' + 'B' + 'C') - (src + dst));
7         hanoi (n - 1, src, spare);
8         hanoi (1, src, dst);
9         hanoi (n - 1, spare, dst);
10      }
```

20.2 Recursion and iteration

Every program which can be implemented using a recusive method can also be implemented using an *iterative* method (using only **while** or **for** loops). The converse is also true, every program which can be implemented using an iterative method can also be implemented using a recursive method. There are some cases where a recursive implementation seems more appropriate and may also be simpler to understand. This can be because the use of a stack (of activation records) is *implicit* in the recursive case whereas it needs to be made *explicit* in the iterative case.

Sometimes we can devise an iterative version of a recursive method that uses a stack data structure in a different way than the recursive method uses activation record stack. We will see such example shortly with an iterative version of the recursive `hanoi()` method above.

20.3 The stack data structure

A stack is a *dynamic* data structure. It grows in size when a new item is added and it shrinks in size when an item is removed. In this way it differs from the *array*, which is a collection of a fixed size. A stack differs from an array in another way also. A stack has an *access discipline* which dictates that items may only be added to the top and they may only be removed from the top. In contrast, an array is a *random access* data structure.

We could also contrast a stack with another dynamic data structure, the *queue*. The stack discipline is last in, first out (LIFO) whereas the queue discipline is first in, first out (FIFO).

Java has a built-in stack datatype with the following interface.

```
1   package java.util;
2   public class Stack extends Vector {
3     // Public constructors
4     public Stack();
5
6     // Public instance methods
7     public boolean empty();
8     public Object peek();
9     public Object pop();
10    public Object push(Object o);
11    public int search(Object o);
12  }
```

For reasons which are lost in the mists of time, the operation which puts a new item onto the top of the stack is named `push()` although the method which removes an item from the top of the stack is called `pop()` (and not “`pull()`”). To inspect the top of the stack without actually removing the top item is to `peek()`.

20.4 Hanoi iteratively

To develop an iterative solution to the Towers of Hanoi problem we need to define the class of object which is pushed onto our stack. These will be objects of class `Move`, shown below.

```
1   static class Move {
2     int n;
3     char src;
4     char dst;
5
6     Move (int n, char src, char dst) {
7       this.n = n;
8       this.src = src;
9       this.dst = dst;
10    }
11  }
```

Finally, it only remains to rewrite our recursive solution to push moves onto the stack and repeatedly to pop them off and break them down again and again to smaller moves. Moving a single disc is trivial and does not increase the number of moves on the stack.

One point about the implementation may need a little more explanation. On line 5 when we pop a move off the stack it is returned to us as an `Object`. We *down-cast* it to a `Move` in order that we can access the `n`, `i` and `j` fields.

```
1   static void hanoi(int n, char src, char dst) {
2     Stack s = new Stack();
3     s.push (new Move(n, src, dst));
4     while (!s.empty()) {
5       Move m = (Move) s.pop();
6       if (m.n == 1)
7         System.out.println ("Move disk from "
8                               + m.src + " to " + m.dst);
9       else {
10        char spare =
11          (char) (('A' + 'B' + 'C') - (m.src + m.dst));
12        // Add last move to do first on stack
13        s.push(new Move(m.n - 1, spare, m.dst));
14        s.push(new Move(1, m.src, m.dst));
15        s.push(new Move(m.n - 1, m.src, spare));
16      }
17    }
18  }
```

Stephen Gilmore.
Updated by Paul Jackson, November 21st, 2002.