CSCI 1933 Lab 11

Using a Stack for a Runtime Stack

Rules

You may work individually or with *one* other partner on this lab. All steps should be completed primarily in the lab time; however, we require that at least the starred steps (★) to be completed and checked off by a TA during lab time. Starred steps cannot be checked off during office hours. You have **until the last office hours on Monday** to have non-starred steps checked by ANY TA, but we suggest you have them checked before then to avoid crowded office hours on Monday. There is nothing to submit via Moodle for this lab. This policy applies to all labs.

Introduction

Methods that generate recursive processes make extensive use of a stack, specifically the system stack. Even non-recursive method calls use a stack so that the runtime system can keep track of the order of method calls. For example, if `methodA()` calls `methodB()` which in turn calls `methodC()`, all three calls will cause an **"activation record"** to be pushed onto the system stack. The activation record for the method called *most recently*, which is also the currently executing method, will appear on the *top* of the stack. In this case, the activation record for `methodC()` will be on top of the stack, followed by the activation record for `methodB()` and then the activation record for `methodA()`.

On the system stack, an activation record (sometimes called stack frame) exists for *each* instance of a method that is active or *waiting* for another method to complete. In a recursive calling situation, there will be many activation records for a single method, with one activation record corresponding to each of the method. All activation records on the system stack belong to method calls that are waiting for methods with activation records higher on the stack to complete (except, of course, for the top one which is executing).

Other than the text of a method's code, activation records contain all the needed information about a specific instance of a method call. When the method is finished, its activation record is "popped" off the system stack because it is no longer needed, and execution resumes with the method corresponding to the activation record next on the stack.

Typical information contained within an activation record are:

- **values of all formal parameters**
- **values of all local (method) variables**

- **return location - where to resume when returning to the calling method**
- **some type of reference to the method's code (or an internal copy)**

Note that the return value from a method call should be kept in a global place where it is available to this activation record, or the previous activation record, or for processing the final value. Before operating systems used a system stack, it was difficult for languages to allow methods to call themselves. While recursive calls to procedures (whether for an iterative or recursive process) are naturally implemented today using the system stack, they can also be implemented by constructing your own stack. That is what we will do in this week's lab. For this lab, you may use the Java's built-in Stack which we already import for you in the resource files.

1 Factorials Recursively Without a Recursive Call ★

First, recall the recursive factorial method:

```
public static int rFactorial(int n) {
   if (n == 0) {
       return 1;
   else {
       return n * rFactorial(n - 1);
   }
}
```

We will implement a similar method that does *not* use a recursive call–that is, nowhere in Java code will a method call itself. To start, we will build a class called `FactRecord` which extends the `ActivationRecord` class. `FactRecord` will be an `ActivationRecord` specific to the factorial method. A `FactRecord` will be instantiated for each call to factorial in place of a recursive call. Each instance will go onto a stack in a manner similar to what the Java runtime system does for the recursive method calls on the system stack.

Part 1 Resources

- `FactRecord.java`
- `FactR.java`
- `ActivationRecord.java`

The rest of this page is intentionally left blank

2 Building a Tower ★

Familiarize yourself with the Tower of Hanoi problem. A recursive approach to listing out the steps required to solve the problem for a tower of `n` is shown below.

1. If there is 1 disk to move, move it from the `source` tower to the `destination` tower. Otherwise, do steps 2,3, and 4.
2. Recursively move n-1 disks from the `source` tower to the `temp` tower.
3. Move disk n from the `source` tower to the `destination` tower.
4. Recursively move the n-1 disks on the `temp` tower to the `destination` tower.

Create a class `Hanoi.java` and implement the following method using the algorithm described above.

```
public static void hanoi(int n, char source, char dest, char temp);
```

Convince yourself that it works by running it for small values of N (3 or less), and following the code through its recursive calls. Note that the Tower of Hanoi solution is inherently recursive. However, in contrast to the factorial problem above, `Hanoi` contains "two" recursive calls within it rather than a single call. Also, be prepared to answer the following questions.

- Which call to `hanoi()` is the first to finish?
- Which call to `hanoi()` is the second to finish?
- Which call to `hanoi()` is the last to finish?
- **Challenge:** Using Big-O notation, what do you think is the time complexity of the `hanoi` method?

> **Hint:** You may want to create a `count` variable to keep track of the number of recursive calls to `hanoi()`.

The rest of this page is intentionally left blank

3 A Building Block

Create a `HanoiR.java` class and `main` method similar to `FactR` above except that this class will be used for `Hanoi`. Include a local stack for activation records using one of the generic stacks from lecture. Note that while you can include a `returnValue` variable in your class similar to that included in `FactR`, you will **not** need to use it because the method for solving Tower of Hanoi does not return anything. Rather, it prints out instructions with `System.out.println()` directly from the method. The `while-loop` that iterates until the stack is empty will be the same except that you will be pushing and running `HanoiRecords` instead of `FactRecords`. Make sure `HanoiR.java` compiles.

4 The Long and Short of It

Write an implementation of the `ActivationRecord` interface that is specific to the Tower of Hanoi problem. Call it `HanoiRecord.java`. In it, you will create local variables to implement the parameters:

- `from` to represent the source tower
- `to` to represent the destination tower
- `tmp` to represent the temporary tower
- `count` to represent the disk number

Note that with the `FactRecord`, there was only one parameter to implement. However, neither `HanoiRecord` nor `FactRecord` have any local variables to represent. Then, write a constructor that accepts the four parameters and have it initialize your local variables accordingly.

Now, implement the `run()` method. Note that in addition to the initial entry point, there will be two return points in `HanoiRecord`'s `run()` method rather that one in `FactRecord`'s `run()` method. Be careful about where you `push()` and `pop()` `HanoiRecords` as well as how you update the return locations.

Test your solution on a variety of Tower sizes to verify that it works.