Data Structures: Lab 2

Following the American Psychological Association’s Guidelines

Markham M. Shofner

Johns Hopkins University

Data Structures: Lab 2

Analysis Document

The Towers of Hanoi problem is well suited for a stack-based solution, since the problem space essentially outlines the definition of a stack (LIFO structure where items are only inserted on or removed from the top of a stack, where common methods like push() and pop() will be needed to move discs around between the trio of stacks). The stack implementation I went with is fairly straightforward, a DiscStack class that has two separate constructors – one that takes a number of discs, and one that initializes an empty stack. The stack class keeps track of its head node, its size, and its name. The DiscNode keeps track of its data (the size of the disc) and a pointer to the next node in the stack.

I found recursion, as opposed to iteration, a much more straightforward way to think about the problem space and to implement a solution. The Towers of Hanoi is a good candidate for recursion because the problem can be broken down into simpler/smaller versions of itself until it reaches the base case (of having a stack with one disc on it). Recursively here, we call the function once on our set of three stacks, and pass in the number of discs into Stack A – so basically, we are recursively calling against the biggest disc in that stack, and solving up and down the three stacks until we move that disc, and then solving up and down the stacks again until we move the largest remaining discs onto the initial largest disc (and so on).

Finding the right code to match these patterns was a bit hard initially and required some experimentation and adjustments. But what helped put me on the right path was watching the lecture video example of the discs moving, but on 1.5 speed to see if I could identify any patterns. From there it became apparent that the discs were following some basic movement patterns centered around having an output stack, and input stack, and an auxiliary stack – and then movement conditions based within that.

Disc 1 cycled from A→B→C→A the whole time, never returning to the stack that it came off of, as its direct next move. Also disc 1 moved every other move. Disc 2 cycled A→C→B→A in the same fashion as disc 1, but in reverse (and also not moving as frequently, as it's disc size is larger and does not get called off the top of the stacks as frequently). Disc 3 followed disc 1's cycle, but again on a much lower frequency following the principle outlined for disc 2 moving less than disc 1. Disc 4 matched disc 2, again with the lower frequency. So with some playing with the base case, and then the next move, a recursive function emerged that could be tweaked/optimized and produce results in line with observational outcomes.

The main method handles the command line arguments and passing those forward to the initializeStacks method. The initializeStacks method sets up three stacks (stackA initialized to n dics, stackB and stackC initialized to empty), declares variables to track operational time, sets up a file writer, and then passes those all forward to either the recursive or iterative moveStack function.

The below table outlines the time requirements for the respective calls. It looks like the recursive solution tended to require somewhere between 50%-75% of the time required by the iterative solution for a corresponding stack of discs. But an n of 25 on the recursive function crashed my computer due to RAM limits, which makes sense because the recursive solution puts far more strain on memory. The run at 25 discs succeeded iteratively, but I stopped moving forward after this ~11 minute run because the fans on my computer were running full tilt and it felt like the device was melting by the end of it.

N	Recursive Time (nano seconds)	Iterative Time (nano seconds)
4	660,177	1,186,001
5	1,575,288	2,828,968
6	2,334,816	3,871,710
7	5,546,505	6,600,802
8	9,547,224	16,581,950
9	17,965,422	30,883,820
10	26,302,168	33,533,054
…	…	…
15	390,359,899	590,177,207
…	…	…
20	14,558,708,404 [14 seconds rounding down]	21,648,933,354 [21 seconds rounding down]
…	…	…
25	Crashed computer	701,212,386,868 [701 seconds rounding down]

Efficiency wise, dropping stacks and just using bare bones local variables would probably save space and time. Since we are simply writing to a file and not tracking other data, a stack is not necessary to solve the problem (as we can track stack names and disc size within the iterative or recursive call). Simplifying the solution would increase speed of operation and also reduce the size requirements of the application.

Enhancements included command line arguments that allow you to name your output file and (directly related to the above paragraph) actually using a stack to solve the problem. I say this because a stack would/could be really useful if we wanted to track more information on each node than just it's size. Say we wanted to give unique names to the nodes, or store passwords on the nodes, or give them colors. The stack based solution will more easily accept any of these potential adjustments.

Similar to the last lab, I wish I had spent more time at the drawing board before diving into the code. Eventually I got there, but I also misused a decent amount of time writing and rewriting parts of the application that could/should have been simpler.

References

zyBooks. 605.202: Data Structures. Module 3 – Recursion.