Jonathan Lam
Prof. Sable
DSA I
10 / 31 / 18

Data Structures and Algorithms I
Fall 2018
Homework #2

1. Describe where the listed data are stored in memory.

For activation records, the names used correspond to the following numbered sections (to match the wording of the assignment):

Name	Section
parameters	1
machine status info	2
local and temporary variables	3

a)	static memory	global variable
b)	activation record — 1	function parameter
c)	activation record — 1	function parameter
d)	activation record — 3	local variable of function
e)	activation record — 1	function parameter
f)	heap	dynamically allocated
g)	heap	dynamically allocated
h)	activation record — 3	local variable of function
i)	heap	large string (dynamically allocated)
j)	heap	element (dynamically allocated) of vector
k)	heap	element (dynamically allocated) of vector
l)	activation record — 3	local variable of function
m)	heap	element (dynamically allocated) of list
n)	heap	dynamically allocated
o)	activation record — 2	holds register data
p)	static memory	global variable
q)	activation record — 3	local variable of function

Jonathan Lam
Prof. Sable
DSA I
10 / 31 / 18

2. Answer questions about implementations of lists, stacks, and queues.

a) One stack would be the primary stack, and the second a temporary stack used only for pushing. The enqueue operation would be accomplished by pushing to the primary stack (retains constant time of push operation to stack). The dequeue operation would simulate popping out the bottom-most value in the stack; this would be accomplished by popping each value from the stack and pushing all of those values (except for the last one) into the temporary stack (at this point, the temporary stack is almost a reverse version of what the primary stack was before the pop operation, and the primary stack only contains the last element). The last element can then be dequeued with a pop operation, and all of the items from the temporary stack can be returned to the main stack by popping from the temporary stack and pushing to the primary stack again. This is linear because the number of constant-time operations is proportional to the size of the list (in this case, number of operations = (2 push + 2 pop operations) * (n-1) + 1 pop, or roughly 4*n).

b) This approach would also have a primary and temporary stack. In 2a), the primary stack had the most recent item on top of the stack and the oldest on bottom; this design is reversed for this exercise. To dequeue, a pop operation is performed on the primary stack (because the most recent item is on top). To enqueue, the primary stack would have to be emptied into the temporary stack by repeatedly popping from the primary stack and pushing into the temporary stack for each element of the stack, pushing the new value onto the (now empty) stack, and then returning all of the values from the temporary stack by the same series popping and pushing. This method uses roughly 4*n operations like the pop operation in 2a), and therefore has worst-case linear time.

c) One stack would be the primary stack (stack 1), and the other stack (stack 2) would only be used for holding all of the values that are lower than or equal to all of the values that come before it in the stack.

Push operation: Push to stack 1. If stack 2 is empty, push value to stack 2 as well (it is by default the minimum value). If not, pop from stack 2, save its value, and push it back to stack 2. If the value to be pushed is lower than or equal to the value at the top of stack 2 (if it is the minimum value of the stack or equal to the current minimum value), push to stack 2; otherwise, don't.

Get minimum value operation: Pop from stack 2. The top value is the minimum of the values in stack 1.

Pop operation: Pop from stack 1, and return this value. Pop from stack 2. If the two popped elements are not same (i.e., if the popped element is not the minimum of the stack), push the value popped from stack 2 back onto stack 2.

d) There is no clearly better choice. These two solutions are analogous to 2a) and 2b): each would have one constant-time operation to enqueue/dequeue from the beginning of the linked-list, and a constant-time operation to perform the opposite operation at the end of the linked-list. The only difference is the what is considered the "beginning" and "end" of the linked-list, which should not change its ability.

e) While the time complexity of a tree-recursive function such as fibonacci ("branching" out twice for every function call) might be exponential (the number of function calls, or nodes of the tree, is exponential), the stack space is only the maximum depth of the tree; in the case of Fib(50), this would be a depth of 50 recursive calls until a base case is obtained, not an unreasonable number that would run a computer out of stack space. It is the depth of the tree because every time a function returns, it gets removed from the call stack; the recursive fibonacci implementation is slow because it branches out into many nodes and has to travel up and down the call stack many times. An (annotated) illustrative diagram courtesy of SICP is shown below: