CSCE 221 Problem Set 6

Jacob Purcell, Texas A&M, Student

Problem 1

Push appends to the end of a list, pop removes from the end of a list, and top reveals the element at the end of a list. Assuming this is a linked list with a pointer to the end node (tail), push and top are O(1), however pop would be O(N) without a doubly linked list, so this list is a doubly linked list. For push, have the tail node point to the next data member and reassign the tail pointer to the new element($2\ operations, O(1)$). For peek, it would be as simple as returning whichever value you are looking at without moving anything($1\ operation, O(1)$).

For pop, there needs to be a way for the tail pointer to know who is behind it on the chain. The simplest way would be to have an extra data member pointing to whomever is previous, then the tail can point to the previous pointer, and the next pointer (the current last element) is deleted($2\ operations, O(1)$). Since these are all independant of the list size, they run on constant time regardless of how big the list becomes. A simple implementation of what I mean is in stack.cpp.

Problem 2

Create array A, list 1 head points to A[0], list 2 head points to A[1]. To push, each list scans the array for an empty element, that element becomes the new value and the pointer is added to the end of that list, this is to make sure there is a clean organisation of space and that there even is space to use. An advantage of this is that it does not matter what order the array is in, the lists can still be organized and maintained effectively. To pop, each list does what is described in problem 1 as there is no need to interact with the array. In this way, N stacks may be created.

Problem 3

A.

As mentioned in problem 1, to push and pop one only needs the location of the tail. To push, have the tail next become the new element, then have the tail point to the new element. In order to pop, we must access the element before the tail, so a doubly linked list is assumed. Have the tail point to the new last element, delete the pointer next to it. findMin can be done in $O(1)$ for sorted lists, one just has to iterate to the zeroeth element or have the tail show what it's data is, both take 3 operations (access, dereference, and return) and therefore execute in constant time ($O(1)$).

B.

If one uses merge sort ($O(\log N)$) to sort the list, with the smallest value as the last element, then deleteMin becomes $O(\log N)$ since the only operation needed would be to pop the list. The entire algorthm bocomes $O(\log N) + O(1)$, and as $N \to \infty$, $O(\log N)$ becomes the dominating term.