182 Week 2 Discussion Notes

Colin Ni

April 8, 2025

This week we will talk about arrays versus linked lists and the matrix problem on HW 2.

Arrays versus linked lists

Recall. An array A has a fixed size that must be specified at the time of construction. The upshot is that it takes $O(1)$ time to retrieve something from the array A: if we want the value at index i, we simply say $A[i]$.

Recall. A linked list can be arbitrary long, as we can always create a new node and add it to the end (or beginning) of the linked list. The consequence is that it takes $O(n)$ time, where n is the length of the linked list, to retrieve a value from the linked list: we are only given the head, so we must follow the trail of the linked list i times to get the value at index i.

Warmup. How can you delete the ith item in a linked list?

Solution. If $i = 0$, then simply define the new head to be the second item (possibly null). Otherwise, define the next item of the $(i-1)$th item to be the $(i+1)$th item (possibly null). □

HW 2 Problem 2 asks you to implement a queue using an array and using a linked list. Let us discuss the similar problem of how we can implement a stack using an array and using a linked list. Recall that a stack is a data structure that roughly speaking lets you push items onto the stack and pop items (the most recently pushed item is popped first) off of the stack.

Example. To implement a stack using an array, we need to first add the assumption that the stack will have a maximum size of M. We begin by initializing an array A of size M, and we keep track of the current size s of the stack, which starts at 0. To push an item x onto the stack, we set $A[s] = x$ and increase s by 1. To pop an item off of the stack, we store $t = A[s]$ temporarily, erase $A[s]$ from the array, decrease s by 1, then return t (alternatively, we can just decrease s by 1 and return $A[s+1]$, but this feels bad). Note that both pushing and popping take $O(1)$ time.

Example. Let us now implement a stack using a linked list. We begin by initializing an empty linked list (a node with no data and no next node).

Here is a dumb way of doing it. As before, we keep track of the current size s of the stack, which starts at 0. To push an item x onto the stack, we create a node n, store x in n, define the next node of the sth node in the linked list to be n, and increase s by 1. To pop an item off of the stack, we store the data in the sth node in the linked list in t temporarily, delete that node, decrease s by 1, and return t. This is dumb because getting the sth node in the linked list takes $O(s)$ time.

Let us describe two better ways of doing this. We can instead add new nodes to the head of the list, redefining the head after each push and pop. This is what was explained in Lecture 4. Alternatively, we can use a doubly linked list, which is the same idea as a linked list but where each node not only keeps track of its next node but also its previous node. Then we keep track of not only the head of the doubly linked list but also the tail, and we do not need to keep track of the size s. It is now easy to implement pushes and pops to work in $O(1)$ time.

We have seen that for stacks, we can do everything in $O(1)$ time with some small innovations. For queues, you will see that you can do things in $O(1)$ time with linked lists but not with arrays.

The matrix problem

Problem. Let A be a $n \times n$ matrix of zeros and ones in which every row consists of a sequence of all ones followed by a sequence of all zeros. Design an $O(n)$ time algorithm to find a row of the matrix which has the maximal number of ones.

Example. The third and fifth rows of the matrix

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

both have the maximal number of ones.

Observe that this can easily be done in $O(n^2)$ time: simply count the number of ones in each row and return a row with the maximal number. To improve this to $O(n)$ time, we need to take advantage of the structure of the rows. The key observation is as follows:

Observation. Having $A[i][j] = 1$ (one-indexed) is equivalent to row i having at least j ones. Similarly, having $A[i][j] = 0$ is equivalent to row i having fewer than j ones.

Since we are looking for a maximal number of ones, if we know there is a row with j ones, then we only need to check the jth column of other rows to

see if they could possibly beat our current best. Thus, we can start at the top left, travel right until we hit a zero, travel down until we hit a one, and repeat.

Let us give a solution to a slightly easier problem where we assume that A has at least one nonzero entry. Note that we are describing an algorithm precisely, proving its correctness, and rigorously analyzing its runtime. Your solution should include better pseudocode, handle the edge case where A is entirely zeros, and justify the last sentence of the solution

Solution outline. Our algorithm is as follows. Set $i = j = 1$. Repeat the following two while loops until i and j stop changing values. While $A[i][j] = 1$ and $j < n$, increase j by 1. While $A[i][j] = 0$ and $i < n$, increase i by 1. Finally, return any i such that $A[i][j-1] = 1$.

Let us first discuss the runtime. Since each step in the inner while-loops increase i or j by 1 and since A is an $n \times n$ matrix, the main while-loop terminates in $2n$ iterations. Finding i such that $A[i][j] = 1$ takes at most n iterations. Thus the overall runtime is $O(n)$.

Let us now prove that this algorithm is correct. Since we are assuming A has at least one nonzero entry, it is easy to see that upon termination of the main while-loop, we have $j > 1$. It follows that there exists a row with at least $j - 1$ ones: the only time that j is increases is when $A[i][j] = 1$, so our Observation says this means row i has at least $j - 1$ ones. Moreover, to see that it is exactly $j - 1$, we can prove the stronger statement that after each iteration of the main while loop, the maximal number of ones in rows $1, \ldots, i - 1$ is exactly $j - 1$; this follows somewhat easily by our Observation because as we increase j, we are moving down, and since we never run into a one until the second while loop terminates, all the previous rows had at most $j - 1$ ones.