Warm-up

Problem 1. Suppose we implement a stack using a singly linked list. What would be the complexity of the push and pop operations? Try to be as efficient as possible.

Solution 1. To push an element we add it at be beginning of the list. To pop an element we delete and return the first element of the list. Both operations take $O(1)$ time.

Problem 2. Suppose we implement a queue using a singly linked list. What would be the complexity of the enqueue and dequeue operations? Try to be as efficient as possible.

Solution 2. To enqueue an element we add it at be end of the list. To dequeue an element we remove it from the front of the list. To keep the enqueue operations efficient, we can keep a pointer to the last element of the list, so that both operations take $O(1)$.

Problem solving

Problem 3. Given a singly linked list, we would like to traverse the elements of the list in reverse order.

a) Design an algorithm that uses $O(1)$ extra space. What is the best time complexity you can get?

b) Design an algorithm that uses $O(\sqrt{n})$ extra space. What is the best time complexity you can get?

You are not allowed to modify the list, but you are allowed to use position/cursors to move around the list.

Solution 3.

a) We keep a cursor (position) that initially points to the last element of the list. We iteratively scan the list until we find the node before the cursor, visit the element at the cursor, and update the cursor to the previous node. The time complexity is $\Theta(n + n - 1 + \cdots + 1) = \Theta(n^2)$.

b) We store $\sqrt{n}$ cursors evenly spaced along the list. We traverse the span between two of these cursors using the previous strategy. Each of these segments is $\sqrt{n}$ long, so each segment takes $\Theta(\sqrt{n}^2) = \Theta(n)$ time to traverse. There are $\sqrt{n}$ many such segments, so the total time is $\Theta(n^{3/2})$.
We can even do better if we are more aggressive on the data that we store. To scan between two cursors, we can traverse the chunk in the forward direction storing all the elements on a stack of size $\sqrt{n}$. Then we can use the smaller stack to traverse that segment in reverse order in $O(\sqrt{n})$ time. In this way, the space is still $O(\sqrt{n})$ and the overall time is $O(n)$.

Problem 4. Consider the problem of given an integer n, generating all possible permutations of the numbers $\{1, 2, \ldots, n\}$. Provide a recursive algorithm for this problem.

Solution 4. The helper function outputs permutations of the input array A that have the first i elements fixed.

The correctness of the algorithm hinges on the fact that while the helper function and its many recursive calls may modify the array during their execution, when a call to a helper function finally returns, the input array is always restored to the state it was in when the call started executing. For a formal argument we prove by induction the property that when we call `helper(A, i)` the algorithm outputs all of the array A that leaves all the entries $A[0 : i]$ fixed while trying all permutations of $A[i : n]$. The full proof is left to the reader.

```
1: function PERMUTATIONS-RECURSIVE(n)
2:     # input: integer n
3:     # do: print all permutations of order n
4:     A ← [1, 2, ..., n]
5:     HELPER(A, 0)
```

```
1: function HELPER(A, i)
2:     if i = size(A) then
3:         Print A
4:     for j ← i; j < n; j++ do
5:         Swap A[i] and A[j]
6:         HELPER(A, i + 1)
7:         Swap A[i] and A[j]
```

Problem 5. Consider the problem of given an integer n, generating all possible permutations of the numbers $\{1, 2, \ldots, n\}$. Provide a non-recursive algorithm for this problem using a stack.

Solution 5. For the non-recursive version we simulate the calls to the helper function with a stack. We use the tuple (c, i, j) to denote stages of a call. The tuple ("start", i, j) corresponds to the start of the for loop for some choice of (i, j) and the

tuple ("finish", i, j) to the part of the body of the for loop after the recursive call to `helper`.

```
1:  function PERMUTATIONS(n)
2:      # input: integer n
3:      # do: print all permutations of order n
4:      A ← [1, 2, ..., n]
5:      S ← a stack with the tuple ("start", 0, 0)
6:      while S is not empty do
7:          c, i, j ← S.pop()
8:          if c = "start" then
9:              if i = n then
10:                 Print A
11:             else
12:                 A[i], A[j] ← A[j], A[i]
13:                 S.push("finish", i, j)
14:                 S.push("start", i + 1, i + 1)
15:         if c = "finish" then
16:             A[i], A[j] ← A[j], A[i]
17:             if j < n − 1 then
18:                 S.push("start", i, j + 1)
```

Problem 6. Using only two stacks, provide an implementation of a queue. Analyze the time complexity of enqueue and dequeue operations.

Solution 6. The simplest solution is to push elements as they arrive into the first stack. When we are required to carry out a dequeue operation, we transfer all the elements to the second stack, pop once to later return the element on the top of the second stack, and then transfer back all the remaining elements back to the first stack.

This strategy works because when we transfer the elements from one stack to the next, we reverse the order of the elements. Before we transfer things, the most recent element to be queued is at the top of the first stack. After we transfer we have the oldest element queued at the top of the second stack. Finally, when we transfer the elements back to the first, we go back to the original stack order.

If the queue holds n elements each enqueue operation takes $O(1)$ time and each dequeue takes $O(n)$ time since we need to transfer all n elements twice.

Problem 7. We want to extend the queue that we saw during the lectures with an operation GETAVERAGE() that returns the average value of all elements stored in the queue. This operation should run in $O(1)$ time and the running time of the other queue operations should remain the same as those of a regular queue.

a) Design the GETAVERAGE() operation. Also describe any changes you make to the other operations, if any.

b) Briefly argue the correctness of your operation(s).

c) Analyse the running time of your operation(s).

Solution 7.

a) Recall that the average is the total sum of the elements divided by the number of elements. Since we already store the size of the queue, it suffices to add a single new variable that stores the sum of the elements, say *sum*. When a new element get enqueued or dequeued, the sum needs to be updated.

```
1: function getAverage()
2:     if isEmpty() then
3:         return "queue empty"
4:     else
5:         return sum/size
```

```
1: function newEnqueue(e)
2:     sum ← sum + e
3:     enqueue(e)
```

```
1: function newDequeue()
2:     e ← dequeue()
3:     sum ← sum − e
4:     return e
```

b) The correctness follows directly from the definition of average, assuming we maintain the sum correctly. Since we add the enqueued element's value to the sum and subtract it when it's dequeued, the sum is maintained correctly.

c) The getAverage operation checks if the queue is empty, which takes $O(1)$ time and either throws an error ($O(1)$ time) or returns the required division of two integers ($O(1)$ time). Hence, the total running time is $O(1)$ as required. We modified the enqueue and dequeue operations. Adding the new element to the *sum* takes $O(1)$ time, so enqueue still runs in $O(1)$ time. Similarly, subtracting the removed element from the *sum* takes $O(1)$ time, so dequeue still runs in $O(1)$ time.