Readings

- Lecture Notes Chapter 10: Selection Problem
- Lecture Notes Chapter 13: Stacks & Queues

Review: Selection Algorithm

SELECT is an $O(n)$ algorithm for finding the $i$th order statistic (the $i$th smallest element) from a set of $n$ elements. It is commonly used to find the median of a set of elements. For example, we can use SELECT to find the median element and use it as the pivot to make DETERMINISTIC QUICKSORT run in worst-case $O(n \log n)$ time instead of in worst-case $O(n^2)$ time. A high-level overview of the algorithm is as follows:

The SELECT algorithm determines the $i$th smallest of an input array of $n > 1$ distinct elements by executing the following steps. (If $n = 1$, then SELECT merely returns its only input value as the $i$th smallest.)

1. Divide the $n$ elements of the input array into $\lceil n/5 \rceil$ groups of 5 elements each and at most one group made up of the remaining $n \mod 5$ elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups by first INSERTION-SORTING the elements of each group (of which there are at most 5) and then picking the median from the sorted list of group elements.
3. Use SELECT recursively to find the median $x$ of the $\lceil n/5 \rceil$ medians found in step 2. (If there are an even number of medians, then by our convention, $x$ is the lower median.)
4. Partition the input array around the median-of-medians $x$ using the modified version of PARTITION.
   Let $k$ be one more than the number of elements on the low side of the partition, so that $x$ is the $k$th smallest element and there are $n - k$ elements on the high side of the partition.
5. If $i = k$, then return $x$. Otherwise, use SELECT recursively to find the $i$th smallest element on the low side if $i < k$, or the $(i - k)$th smallest element on the high side if $i > k$.

Runtime Analysis

Step 1 takes $O(n)$ time to divide the elements into groups of 5 elements each. For Step 2, note that when we call INSERTION-SORT, we call it on a group of constant size (the maximum size of a group is 5), so each call to INSERTION-SORT takes $O(1)$ time. Since we call INSERTION-SORT on each of the $\lceil n/5 \rceil$ groups and each call takes $O(1)$ time, Step 2 takes $O(\lceil n/5 \rceil) = O(n)$ time as well. In Step 3, we call SELECT recursively on the medians found in Step 1 to find the median-of-medians $x$; since we have $\lceil n/5 \rceil$ groups of elements and each group has 1 median, we recurse on $\lceil n/5 \rceil$ elements, so Step 3 takes $T(\lceil n/5 \rceil)$. Step 4 takes $O(n)$ time because we call PARTITION on the input array.

To calculate the running time of Step 5, we need to calculate the number of elements we recurse on in the worst-case. Observe that we recurse on either the number of elements greater than $x$ or on the number of elements less than $x$. To upper-bound the number of elements SELECT is recursively called on in the worst-case, we will first find a lower-bound on the number of elements greater than $x$ and the number of elements less than $x$. Subtracting these lower-bounds from $n$ will yield the maximum number of elements SELECT is called on if we recurse on the lower side of $x$ or on the higher side of $x$, respectively. For the number of elements greater than $x$, note that at least half of the medians are greater than or equal to the median-of-medians $x$. So, at least half of the $\lceil n/5 \rceil$ groups will contribute 3 elements that are greater than $x$ (the median and the 2 elements in its group greater than or equal to the median), except for the group that
contains \( x \) and the group that may have fewer than 5 elements if 5 does not divide \( n \) exactly. Therefore, the number of elements greater than \( x \) is at least

\[
3 \left( \left\lceil \frac{n}{5} \right\rceil - 2 \right) \geq \frac{3n}{10} - 6
\]

Similarly, at least \( \frac{3n}{10} - 6 \) elements are less than \( x \). From this analysis, in the worst-case, there are at most \( n - (3n/10 - 6) = 7n/10 + 6 \) elements less than \( x \) and \( n - (3n/10 - 6) = 7n/10 + 6 \) elements greater than \( x \), so in the worst-case, we recurse on at most \( 7n + 10 \) elements. So, Step 5 takes \( T(7n/10 + 6) \) time.

Therefore, in Steps 1, 2, and 4, we perform \( O(n) \) work, in Step 3, we recurse on an input of size \( \lceil n/5 \rceil \), and in Step 5, we recurse on an input of size at most \( T(7n/10 + 6) \). This yields a recurrence of

\[
T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)
\]

which is equal to \( O(n) \).

**Review: Stacks and Queues**

An abstract data type (ADT) is an abstraction of a data structure; it specifies the type of data stored and the operations that can be performed, similar to Java interfaces. Recall the Stack and Queue ADTs:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>• LIFO (Last-In-First-Out): the most recent element added to the stack will be removed first</td>
<td>• FIFO (First-In-First-Out): the oldest/least recent element added to the queue will be removed first</td>
</tr>
<tr>
<td>• Supported operations:</td>
<td>• Supported operations:</td>
</tr>
<tr>
<td>− push: amortized ( O(1) )</td>
<td>− enqueue: amortized ( O(1) )</td>
</tr>
<tr>
<td>− pop: amortized ( O(1) )</td>
<td>− dequeue: amortized ( O(1) )</td>
</tr>
<tr>
<td>− peek: ( O(1) )</td>
<td>− peek: ( O(1) )</td>
</tr>
<tr>
<td>− isEmpty: ( O(1) )</td>
<td>− isEmpty: ( O(1) )</td>
</tr>
<tr>
<td>− size: ( O(1) )</td>
<td>− size: ( O(1) )</td>
</tr>
</tbody>
</table>

**Implementation Details**

In this course, we implement stacks and queues using (dynamically resizing) arrays. In other words, we adjust the size of the array so that it is large enough to store all of its current elements but not large enough that it wastes space. The rules we will use for increasing or decreasing the size of a stack or queue’s underlying array are as follows:

1. If the array of size \( n \) is full, create a new array of size \( 2n \) and copy all elements into the new array.
2. If the array of size \( n \) has \( \frac{n}{2} \) elements in it, create a new array of size \( \frac{n}{2} \) and copy all elements into the new array.

Note that we resize “down” when the array has \( \frac{n}{2} \) elements in it (instead of when it has \( \frac{2n}{2} \) elements) to prevent “thrashing.” If we resized “down” when the array has \( \frac{n}{2} \) elements, consider the case where we push elements onto a stack until it resized “up.” If we were to pop a single element, then we would have to resize “down,” but then if we were to push another element, we would have to resize “up” again, so in the worst-case, each push/pop operation would require copying elements and creating new arrays, increasing our runtimes.
Amortized Analysis

When calculating the runtimes of operations for stacks and queues, we perform amortized analysis. In amortized analysis, the amortized runtime of a single operation is equal to the time needed to perform a series of operations divided by the number of operations performed. For example, let \( T(n) \) be the amount of time needed to perform \( n \) push operations. Then, the amortized runtime of a single push operation is equal to \( \frac{T(n)}{n} \). Observe that we often perform amortized analysis in situations where the occasional operation takes much longer than the rest of the operations. Considering a stack, in the worst-case, a push operation takes \( O(n) \) time because of array resizing, but otherwise most of the push operations take \( O(1) \) time.

Note: Amortized analysis is not the same as average-case analysis, since it does not depend at all on the probability distribution of inputs. Instead, the total running time of a series of operations is still bounded by the total costs of the amortized runtimes of the operations.

Problems

Problem 1

You are given two stacks \( S_1 \) and \( S_2 \) of size \( n \). Implement a queue using \( S_1 \), \( S_2 \), and a stack’s push, pop, and/or peek methods. What are the (amortized) running times of your new enqueue and dequeue methods?

Solution

\[
\text{enqueue}(x):
\begin{align*}
1. & \text{ push } x \text{ into } S_1. \\
\end{align*}
\]

\[
\text{dequeue}:
\begin{align*}
1. & \text{ If } S_2 \text{ is empty, pop all elements from } S_1 \text{ and push them into } S_2. \text{ If } S_2 \text{ is still empty, return Nil.} \\
2. & \text{ Otherwise, pop an element from } S_2 \text{ and return it.}
\end{align*}
\]

Proof of Correctness: We want to show that our enqueue and dequeue methods maintain a queue’s FIFO invariant. Since we enqueue an element by push’ing it onto \( S_1 \), to properly dequeue, we need to access the elements in \( S_1 \) in “reverse” order, from the bottom to the top of the stack. We maintain and ensure this by pop’ing elements from \( S_1 \) and push’ing them onto \( S_2 \) when necessary, so we can just pop from \( S_2 \) to dequeue. In the edge case where both \( S_1 \) and \( S_2 \) are empty, there are no elements in the queue, so we correctly return Nil. Otherwise, because a stack is LIFO, our algorithm is correct.

Runtime Analysis: The amortized running time of enqueue is \( O(1) \). For the amortized running time of dequeue, observe that overall, we push each element exactly twice (once into \( S_1 \) and once into \( S_2 \)) and pop each element exactly twice (once from \( S_1 \) and once from \( S_2 \)). Hence, since we have \( n \) elements, we still have an \( O(n) \) running time over all dequeue operations. When we average this over \( n \) operations, we see that dequeue still runs in \( O(1) \) amortized time. (At a high-level, when we dequeue, note that we only move elements to \( S_2 \) if \( S_2 \) is empty, and when we move elements onto \( S_2 \), we move many elements at once. So, the dequeue operation when \( S_2 \) is empty pays the “cost” so the following dequeue operations are faster, since in the future we can pop off the now non-empty stack.)

Space Analysis: Beyond the given stacks, we use \( O(1) \) additional space.

Problem 2

You are given a full stack \( S_1 \) and an empty stack \( S_2 \), each of size \( n \). Design an algorithm to sort the \( n \) elements in non-decreasing order in \( S_2 \), using only \( O(1) \) additional space beyond \( S_1 \) and \( S_2 \). What is the running time of your sorting algorithm?
Example:

\[
\begin{array}{c|c|c|c}
4 & & & 1 \\
3 & & & 2 \\
1 & & & 3 \\
5 & & & 4 \\
2 & & & 5 \\
\end{array}
\rightarrow
\begin{array}{c|c|c|c}
1 & & & 2 \\
2 & & & 3 \\
3 & & & 4 \\
4 & & & 5 \\
5 & & & 5 \\
\end{array}
\]

Hint: Start with a smaller example:

\[
\begin{array}{c|c|c}
3 & & 1 \\
2 & & 2 \\
1 & & 3 \\
\end{array}
\rightarrow
\begin{array}{c|c|c}
1 & & 2 \\
2 & & 3 \\
3 & & 3 \\
\end{array}
\]

Solution

We use the two given stacks, $S_1$ and $S_2$, and two extra variables max and size in our algorithm.

**Algorithm:** Initialize max to $-\infty$ and size to 0. Repeat these steps until size = n:

1. pop all elements from $S_1$ and push them onto $S_2$. While pop’ing, keep track of the maximum element we have seen so far in max.

2. pop elements from $S_2$ (until only size elements remain in $S_2$) and push all of these elements, except the maximum element stored in max, back into $S_1$.

3. push the maximum element (stored in max) into $S_2$.

4. Increment size by 1, so we can keep track of the number of sorted elements in $S_2$ and not pop them.

**Proof of Correctness:** The correctness of our algorithm follows from a stack’s LIFO invariant. $S_1$ starts with all (unsorted) elements, and we maintain this invariant that $S_1$ only contains elements that have not yet been sorted because in Step 2, we pop from $S_2$ into $S_1$ so only the bottom size elements (the number of elements sorted) remain in $S_2$. While we pop from $S_1$, we correctly update max to be the max element that is currently unsorted and then “sort” this element by push’ing max into $S_2$ in Step 3. Our algorithm terminates when size = n (when $S_1$ is empty), so all elements have been sorted. Because a stack is LIFO and we “sort” an element each iteration by push’ing the maximum unsorted element found into $S_2$, when our algorithm terminates, $S_2$ contains all elements sorted in non-decreasing order.

**Runtime Analysis:** Each iteration of our “loop” (Steps 1 to 4) sorts exactly 1 element. For each of the $n$ elements we sort, we push and pop at most $n$ elements. So, our sorting algorithm runs in $O(n^2)$ time.

**Space Analysis:** Beyond the given stacks, we use two variables max and size for $O(1)$ additional space.