Readings

- [Lecture Notes Chapter 13: Stacks & Queues](#)
- [Lecture Notes Chapter 14: Binary Heaps and Heapsort](#)

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 a Java interface. 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 less than $\frac{n}{4}$ 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 $\frac{n}{4}$ elements in it (instead of when it has $\frac{n}{2}$ 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, every 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 since we’re just setting a value at an index of the array.

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 bounded by the total costs of the amortized runtimes of the operations.

Review: Heaps

A heap is a tree-like data structure that implements the priority queue ADT, which allows us to maintain a set of elements, each with an associated key, and select the element with highest/lowest priority. Heaps satisfy the two following properties:

**Heap Property:** In a max-heap, for each node $i$, we have $A[\text{Parent}(i)] \geq A[i]$, so the maximum value is stored at the root. In a min-heap, for each node $i$, we have $A[\text{Parent}(i)] \leq A[i]$, so the minimum value is stored at the root.

**Shape Property:** A heap is an almost complete binary tree, meaning that every level of the tree is completely filled except for the last, which must be filled from left to right.

Implementation Details

Because a heap is an almost complete binary tree, we are able implement it using an array with 1-indexing as shown below:

Observe that we can populate the array from left to right by doing a level-order traversal of the tree, where we start from the root and go through each level of the tree from left to right. Additionally, because of the shape property, if the root is stored at index 1 of the array, given a node at index $i$, its left child can be found at index $2i$, its right child can be found at index $2i + 1$, and its parent can be found at index $\lfloor i/2 \rfloor$. 
Operations (Max-Heaps)

**Max-Heapify** maintains the max-heap property at the node called on, so the entire subtree rooted at the node will now be a max-heap. It assumes the node’s left and right subtrees are both valid max-heaps, and then allows the node to “float-down,” swapping it with its larger child or terminating if the max-heap property holds. It runs in $O(h)$ time, where $h$ is the height of the node, since in the worst case, the node must “float down” to the bottom of the tree. Since the height of any node is upper bounded by $\log n$, **Max-Heapify** runs in $O(\log n)$ for any node (though this bound may not be tight for some nodes, which is a property we leverage when analyzing the runtime of **Build-Max-Heap**).

**Build-Max-Heap** constructs a max-heap from an unsorted array by repeatedly calling **Max-Heapify** on nodes from the “bottom-up”, starting at the nodes right above the leaves (which by definition are max-heaps!). It runs in $O(n)$ time; the mathematical proof of this upper-bound can be found [here](#).

**Extract-Max** removes and returns the element with the maximum key. We remove the root, replace it with the right-most element in the bottom level/last element in the array, and then call **Max-Heapify** on the “new” temporary root to maintain the max-heap property. We perform constant work besides calling **Max-Heapify**, so it runs in $O(\log n)$ time.

**Insert** adds an element by first adding it to the end of the array/max-heap, and then allowing it to “float-up” to its correct position by repeatedly swapping it with its parent as necessary to maintain the max-heap property. It runs in $O(\log n)$ time, since the path it takes while it “floats-up” has length $O(\log n)$.

**Peek** returns the maximum element in the heap stored at the root. Since we implement a heap with an array, this runs in $O(1)$ time because we just index into the array.

**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**

**enqueue**(x):

1. push x into $S_1$.

**dequeue**:

1. If $S_2$ is empty, pop all elements from $S_1$ and push them into $S_2$. If $S_2$ is still empty, return Nil.
2. Otherwise, pop an element from $S_2$ and return it.

**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**. Since a stack is LIFO, any elements that are pushed into $S_2$ must be in reverse order relative to how they were pushed into $S_1$, so popping off $S_2$ guarantees that the correct element in the queue is retrieved at any time. 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.

**Runtime Analysis:** The amortized running time of **enqueue** is $O(1)$, same as a regular stack **push** operation. For the amortized running time of **dequeue**, observe that for each element we **dequeue**, we **push** exactly once (into $S_2$) and **pop** 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 (as push and pop are amortized $O(1)$). 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 $S_2$.

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

**Problem 2**

You are given a full stack $S_1$ with distinct elements and an empty stack $S_2$, each of size $n$. Design an algorithm to sort the $n$ elements in increasing order from the top 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:**

```
4 3 1 5 2 → 1 2 3 4 5
```

*Hint:* Start with a smaller example:

```
3 2 1 → 1 2 3
```

**Solution**

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

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

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 $\text{max}$.
2. pop elements from $S_2$ (until only $\text{size}$ elements remain in $S_2$) and push all of these elements, except the maximum element stored in $\text{max}$, back into $S_1$.
3. push the maximum element (stored in $\text{max}$) into $S_2$.
4. Increment $\text{size}$ 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 $\text{size}$ elements (the number of elements sorted) remain in $S_2$. While we pop from $S_1$, we correctly update $\text{max}$ to be the max element that is currently unsorted and then “sort” this element by push'ing $\text{max}$ into $S_2$ in Step 3. Our algorithm terminates when $\text{size} = n$ (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. $n$ push/pop operations take $O(n)$ time, and for each of the $n$ elements we sort, we push and pop at most $n$ elements. Therefore, our sorting algorithm runs in $O(n^2)$ time.

**Space Analysis:** Beyond the given stacks, we use two variables $\text{max}$ and $\text{size}$ for $O(1)$ additional space.
Problem 3

Given a data stream of $n$ test scores, design an $O(n \log k)$ time algorithm to find the $k$-th highest test score. Since PEFS provides minimal monetary resources, CIS 121 Staff has limited access to storage space and can only afford you $O(k)$ space, where $k \ll n$.

Solution

Algorithm: Construct a min-heap from the first $k$ tests, where tests are ordered by their score, by calling Build-Min-Heap. For each remaining test in the data stream: if its score is greater than the score at the root of the heap, remove the root by calling Extract-Min and then Insert the current test; otherwise, the score of the current test is less than or equal to the score at the root, so do nothing. After processing all tests in the data stream, return the score at the root of the heap by calling Peek.

Proof of Correctness: We will prove the correctness of our algorithm by showing that our min-heap contains the $k$-highest scores at the termination of our algorithm. Let $s$ be the $k$-th highest score. Assume for the sake of contradiction that a test $a_{bad}$ with a score lower than $s$ remains in the heap when our algorithm terminates. Since we maintain a heap of size $k$, this implies that some test $a_{good}$ with a score higher than $s$ is excluded from the heap. However, by design, it is impossible for $a_{good}$ to have been excluded, since its score is higher than $a_{bad}$, so it would have been included by transitivity. This is a contradiction, so by returning the score at the root, our algorithm is correct.

Runtime Analysis: Constructing a min-heap from the first $k$ tests by calling Build-Min-Heap takes $O(k)$ time. For each remaining test, we either do nothing, or we maintain the heap at size $k$ by calling Extract-Min and then Insert on the current score, which takes $O(\log k)$ time. Each test is inserted into the heap at most once, and since the data stream has $n$ scores, our overall running time is $O(k + n \log k)$. Since $k \ll n$, our final running time is $O(n \log k)$.

Space Analysis: As a “pre-processing” step, we first construct a min-heap with the first $k$ tests. We maintain the heap at size $k$ because for each remaining test that we insert, we remove the root. Therefore, the space complexity of our algorithm is $O(k)$.