CIS 1210—Data Structures and Algorithms—Spring 2025

Stacks, Queues, Heaps—Tuesday, February 18 / Wednesday, February 19

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:

Stack	**Queue**
• LIFO (Last-In-First-Out): the most recent element added to the stack will be removed first • Supported operations: – `push`: amortized $O(1)$ – `pop`: amortized $O(1)$ – `peek`: $O(1)$ – `isEmpty`: $O(1)$ – `size`: $O(1)$	• FIFO (First-In-First-Out): the oldest/least recent element added to the queue will be removed first • Supported operations: – `enqueue`: amortized $O(1)$ – `dequeue`: amortized $O(1)$ – `peek`: $O(1)$ – `isEmpty`: $O(1)$ – `size`: $O(1)$

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 runtime of the amortized operations.

Review: Heaps

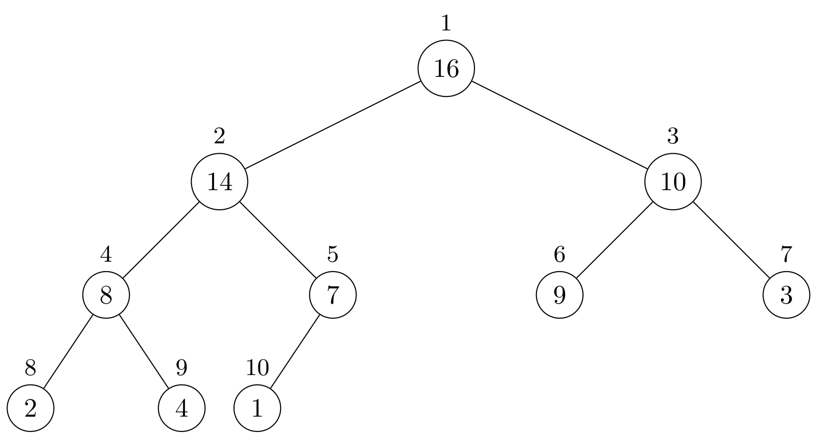
A heap is a tree-like data structure that implements the priority queue abstract data structure (ADT), which allows us to maintain a set of elements, each with an associated key, and select the element with the highest/lowest priority. Heaps satisfy the two following properties:

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



Index	0	1	2	3	4	5	6	7	8	9	10
Value	`null`	16	14	10	8	7	9	3	2	4	1

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 $O(\log n)$ time 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?

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				1
3				2
1		$\rightarrow$		3
5				4
2				5

Hint: Start with a smaller example:

3				1
2		$\rightarrow$		2
1				3

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 1210 Staff has limited access to storage space and can only afford you $O(k)$ space, where $k \ll n$.