Stacks and Queues

Recall the stack and queue ADTs (abstract data types) from lecture. Each is characterized by a specific way of removing elements and has a set of supported operations.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>• LIFO (last-in-first-out)—the most recent element that has been added to the stack will be removed first.</td>
<td>• FIFO (first-in-first-out)—the least recent element that has been added to the queue will be removed first.</td>
</tr>
<tr>
<td>• Supported operations:</td>
<td>• Supported operations:</td>
</tr>
<tr>
<td>– push</td>
<td>– enqueue</td>
</tr>
<tr>
<td>– pop</td>
<td>– dequeue</td>
</tr>
<tr>
<td>– peek</td>
<td>– peek</td>
</tr>
<tr>
<td>– isEmpty</td>
<td>– isEmpty</td>
</tr>
<tr>
<td>– size</td>
<td>– size</td>
</tr>
</tbody>
</table>

Implementation Details

Stacks and queues can be implemented “under the hood” with almost any data structure. In this course, we will implement stacks and queues using expandable arrays. 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 $\frac{n}{4}$ elements in it, create a new array of size $\frac{n}{2}$, and copy all elements into the new array.

Amortized Analysis

*Amortized analysis* refers to finding the time-averaged cost for a sequence of operations. In other words, it is the time required to perform a sequence of operations averaged over all the operations performed. 

Since amortized analysis for the stack *push* operation was covered in lecture, we are going to take a closer look at the stack *pop* operation.

The worst case running time for a single *pop* operation is $O(n)$, since we may need to resize the array and copy the elements into it. Based on this running time, we might conclude that a tight bound for the worst case running time for $n$ *pop* operations is $O(n^2)$, since there are $n$ operations and each operation takes worst case $O(n)$ time; however, we can find a tighter bound through some careful analysis.

If we start from a full stack of size $n$, what is the total cost of a sequence of $n$ *pop* operations?

Initially, the array is of size $n$ and contains $n$ elements. To make our analysis simpler, let’s immediately *pop* the first $\frac{n}{2}$ elements. Each of these *pops* takes $O(1)$ time. Now our array is of size $n$ but contains only $\frac{n}{2}$ elements.

---

2. The analysis for *enqueue* and *dequeue* is similar to that of *push* and *pop*, respectively.
In accordance with our rules, we can \( \text{pop} \frac{n}{4} \) more elements before resizing the array. Each of these \( \text{pops} \) takes \( O(1) \) time. Once we have \( \text{pop}'d \) those elements (leaving us with \( \frac{n}{4} \) elements in our array), we must reduce the size of our array to \( \frac{n}{2} \), and copy the remaining \( \frac{n}{4} \) elements into the new array. Thus, the total cost for the first \( \frac{3n}{4} \) \( \text{pop} \) operations is \( T\left(\frac{3n}{4}\right) = \frac{n}{2} + \left(\frac{n}{4} + \frac{n}{2} + \frac{n}{4}\right) \).

We can apply identical analysis to the new array of size \( \frac{n}{4} \) that contains \( \frac{n}{4} \) elements. We get \( \frac{1}{4} \left(\frac{n}{2}\right) = \frac{n}{4} \) \( \text{pops} \) “for free”, after which we resize the array to be of size \( \frac{1}{2} \left(\frac{n}{2}\right) = \frac{n}{4} \) and copy the remaining \( \frac{1}{4} \left(\frac{n}{2}\right) = \frac{n}{8} \) elements into the smaller array. Thus, the total cost for the first \( \frac{7n}{8} \) \( \text{pop} \) operations is \( T\left(\frac{7n}{8}\right) = \frac{n}{4} + \left(\frac{n}{4} + \frac{n}{2} + \frac{n}{4}\right) \).

Are you noticing a pattern?

Let’s rewrite the expression slightly and continue to expand it:

\[
T(n) = \frac{n}{2} + \left(\frac{1}{4} \left(\frac{n}{2^n}\right) + \frac{1}{2} \left(\frac{n}{2^n}\right) + \frac{1}{4} \left(\frac{n}{2^n}\right)\right) + \left(\frac{1}{4} \left(\frac{n}{2^n}\right) + \frac{1}{2} \left(\frac{n}{2^n}\right) + \frac{1}{4} \left(\frac{n}{2^n}\right)\right) + \cdots + \left(\frac{1}{4} \left(\frac{n}{2^n}\right) + \frac{1}{2} \left(\frac{n}{2^n}\right) + \frac{1}{4} \left(\frac{n}{2^n}\right)\right)
\]

We can now calculate the total cost of \( n \) \( \text{pop} \) operations:

\[
T(n) \leq \frac{n}{2} + \sum_{i=0}^{\infty} \left(\frac{1}{4} \left(\frac{n}{2^n}\right) + \frac{1}{2} \left(\frac{n}{2^n}\right) + \frac{1}{4} \left(\frac{n}{2^n}\right)\right) = \frac{n}{2} + n \sum_{i=0}^{\infty} \frac{1}{2^i} = \frac{n}{2} + 2n \leq 3n = O(n)
\]

(The first term in the summation is the \textit{cost of the initial \text{pops}}, the second term is the \textit{cost of allocating} a new array, and the third term is the \textit{cost of copying} the remaining elements into the new array.)

Thus, the \textit{amortized} time complexity of a \( \text{pop} \) operation is \( 3 = O(1) \), even though the worst case time complexity of a single \( \text{pop} \) operation is \( O(n) \).

**Introduction: Heaps**

A \textit{heap} is a tree-like data structure that satisfies the heap-order property.

**Definition** (Heap-Order Property). A tree has the \textit{heap-order property} if for any parent node \( P \) with a child \( C \), the key of \( P \) is ordered with respect to the child \( C \).

Common examples of orderings on a heap would be \( \geq \) (max-heap) or \( \leq \) (min-heap). For \( \geq \), the key in each node in the heap \( T \) is greater than or equal to the keys of all nodes in its subtree.
An example binary max-heap. Note that the root contains the maximum key.

Notice that this definition immediately implies that the root must contain either the “maximum” or the “minimum” of the ordering relationship that we define, since the root is the parent/ancestor of every other node. Specializing this definition to keys that act like natural numbers, or keys that implement Comparable, we have our classic min-heap and max-heap. This basic idea is really powerful, as the heap data structure maintains the “maximum” or “minimum” element whenever we add to it or remove from it. This means that we can retrieve the max/min element quickly!

Binary Heaps

A binary heap is a binary tree, but with the heap-order property. A binary heap is most commonly implemented by flattening a tree in level order into an array. It satisfies the following property:

**Definition** (Shape Property). A tree has the heap-shape property if the tree is a complete binary tree. That is, all levels of the tree are fully filled, except for possibly the last, where all nodes are as far left as possible.

With the shape property, we can easily index into a binary heap, since we will not have to worry about “gaps.”
A max-heap visualized as both a tree and an array.

For an element at index $i$ of $A$, its left and right children can be found at indices $2i$ and $2i + 1$ respectively. Conversely, an element at index $i$ has its parent at index $\lfloor i/2 \rfloor$.

This property holds true only if the heap begins at index 1 of the array (or if the array is one-indexed).

**Running time of Operations**

Running times are given with respect to $n$, where $n$ is the number of elements in the binary heap.

- **insert** $(x, k)$: An element $x$ with key $x$ may be inserted in $O(\log n)$ time.
- **find-min/max()**: Finding the min/max of a binary heap takes $O(1)$ time.
- **extract-min/max()**: Removing the root and restoring the min/max heap property takes $O(\log n)$ time.
- **decrease/increase-key** $(x, k)$: Changing the key of an element can be done in $O(\log n)$ time. Note that the Java implementation of a priority queue does not support this operation.

**Partial Ordering**

We say that the heap-order property induces a partial order over its elements. Intuitively, a partial order means that not every pair of elements are related. Even though we know that 17 is less than 23, when we insert these numbers into the heap, we cannot determine which number is “greater” solely by its position in the heap. Compare this to inserting both elements in a binary search tree, where we can determine the order by examining their relative positions. We say that the binary search tree establishes a total order.

For some problems, it is enough to have just a partial ordering. For example, if you want to get the $k$-largest elements of a list relatively fast, you can use a heap to achieve this. As you’ve seen with MERGESORT and some implementations of QUICKSORT, you can get a stronger, total ordering at the cost of a larger running time $[\Omega(n \log n)]$. However, building a heap only takes time linear in the number of elements. Therefore, we can get the maximum/minimum in linear time and the partial ordering!

**Building a (Max) Heap**

In order to build a heap, we define the following subroutine: MAX-HEAPIFY. Under the assumption that the left and right subkeys of the $i$’th vertex are valid max heaps, MAX-HEAPIFY ensures that the subtree rooted at $i$ is also a valid max heap. The running time analysis of MAX-HEAPIFY is left as a discussion topic. We can then write:
function Max-Heapify(A, i)
  
l ← left(i)
  r ← right(i)
    largest ← l
  else
    largest ← i
    largest ← r
  if largest ≠ i then
    swap(A[i], A[largest])
  max-heapify(A, largest)

function Build-Max-Heap(A)
  A.heapsize ← A.length
  for i ← ⌊A.length/2⌋ downto 1 do
    max-heapify(A, i)

The build-max-heap algorithm starts from the last internal node of the binary tree representation of A and converts each subtree to a max-heap, recursing upwards. As above, the running time analysis of BUILD-MAX-HEAP is left as a discussion topic.

Heapsort

function Heapsort(A)
  build-max-heap(A)
  for i ← A.length downto 2 do
    swap(A[1], A[i])
    A.heapsize ← A.heapsize − 1
    max-heapify(A, 1)

The HEAPSORT algorithm works by first converting the input array A to a max-heap. It grows the sorted subarray from right to left by swapping out the root (largest element at A[1]) to its proper place in the sorted subarray and restoring the max-heap property on the unsorted subarray. (Does this notion of dividing the input into an unsorted/sorted region remind you of another sorting algorithm...?) The running time analysis of HEAPSORT is also left as a discussion topic.

Discussion Topics

- What is the worst case running time of MAX-HEAPIFY? Why?

- Why does constructing a heap (BUILD-MAX-HEAP) take linear time? What happens if we try to build a heap by running INSERT n times instead?

- Given that both BUILD-MAX-HEAP and HEAPSORT call MAX-HEAPIFY at least n/2 times, why does HEAPSORT run in Θ(n log n) time and not BUILD-MAX-HEAP?
• Discuss **insertion-sort**, **mergesort**, **quicksort**, and **heapsort**. What are their relative advantages? When might one sorting algorithm be preferred over the others?

**Problems**

**Problem 1.** You have been hired to write an application for 121-CIS's new network router! 121-CIS plans to sell their routers to businesses with large corporate networks that need swift detection of network attacks. A network attack is characterized by a large amount of traffic from a single IP address. For the application, you are parsing a stream of packets containing an IP-address and their frequency. Routers have limited memory, and you can only maintain $O(k)$ space for your application, where $k \ll n$.

Design an $O(n \log k)$ time algorithm to find the $k$-th most frequent IP-address, where $n$ is the total number of IP addresses in the stream.

**Problem 2.** Given: A binary tree of size $n$

*Objective:* Print out the level order traversal of the binary tree

*Example:* see below

![Figure 1: For this tree, your function should print 1, 2, 3, 7, 6, 5, 4.](image)

**Problem 3.** Given: A binary tree $T$.

*Objective:* Print the spiral order traversal of the tree $T$.

*Example:*

![Figure 2: For this tree, your function should print 1, 2, 3, 4, 5, 6, 7.](image)

*Hint:* Try using 2 stacks.

**Problem 4.** Consider an indefinitely long stream of unsorted integers. We are interested in knowing the median (in sorted order) at any given time. How would we do this in an efficient manner?
Problem 5. Given: A full stack $S_1$ of size $n$ and an empty stack $S_2$ of size $n$.

Objective: Sort the $n$ elements in ascending order in $S_2$. You may only use the given 2 stacks $S_1$ and $S_2$ (each of size $n$) and $O(1)$ additional space. What is the running time of your sorting procedure?

Example:

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

Hint: Start with a simpler example:

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