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 \textbf{pop} \( \frac{n}{4} \) more elements before resizing the array. Each of these \textbf{pops} takes \( O(1) \) time. Once we have \textbf{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 \( \frac{3n}{4} \) \textbf{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 \( \frac{1}{4} \left( \frac{n}{2} \right) = \frac{n}{8} \) \textbf{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 \( \frac{n}{4} \) elements into the smaller array. Thus, the total cost for the first \( \frac{2n}{8} \) \textbf{pop} operations is 

\[ T\left( \frac{2n}{8} \right) = \frac{n}{4} + \left( \frac{n}{4} + \frac{n}{4} + \frac{n}{4} \right). \]

Are you noticing a pattern?

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

\[ T(n) = \frac{n}{2} + \left( \frac{1}{4} \left( \frac{n}{2^0} \right) + \frac{1}{2} \left( \frac{n}{2^0} \right) + \frac{1}{4} \left( \frac{n}{2^0} \right) \right) + \left( \frac{1}{4} \left( \frac{n}{2^1} \right) + \frac{1}{2} \left( \frac{n}{2^1} \right) + \frac{1}{4} \left( \frac{n}{2^1} \right) \right) + \left( \frac{1}{4} \left( \frac{n}{2^2} \right) + \frac{1}{2} \left( \frac{n}{2^2} \right) + \frac{1}{4} \left( \frac{n}{2^2} \right) \right) + \cdots \]

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

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

\[ = \frac{n}{2} + n \sum_{i=0}^{\infty} \frac{1}{2^i} \]

\[ = \frac{n}{2} + 2n \]

\[ \leq 3n \]

\[ = O(n) \]

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

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

\section*{Introduction: Heaps}

A heap is a tree-like data structure that satisfies the heap-order property.

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

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 subtrees 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
    ▷ One of children is larger. Swap and recurse.
    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?

  Solution. The worst case running time is \(O(\log n)\). For our algorithm, we do \(\Theta(1)\) work at each level of the recurrence (comparisons/swap) and recurse on either of the subtrees. Therefore, our recurrence will look like this:

  \[ T(n) = T(\text{size of subtree}) + \Theta(1) \]

  In the worst case, we want to examine the case where the size of the subtree we recurse on is maximal \textit{with respect to} \(n\). That is, at each level of the recurrence, we want to choose the largest possible fraction of the \(n\) nodes to maximize \(T(n)\). This case occurs when the bottom level of \(T\) is \textit{half full} (the right subtree's bottom level is empty).
To determine the maximum size of the subtree chosen, we use the following theorem:

**Theorem 1.** Let $T$ be a nonempty, full binary tree. Then the number of leaf nodes in $T$ is one more than the number of internal nodes in $T$.

Let $|R| = k$ be the number of nodes in the right subtree of $T$. Then we have $|L| = k + (k + 1)$ by the above theorem (as $|R|$ would be the number of internal nodes in $L$). Then, $|T| = n = |R| + |L| = 3k + 1$, and $|L|/|T| < 2/3$.

Therefore, we have that the worst case for $T(n) \leq T(\frac{2n}{3}) + \Theta(1) = O(\log n)$.

- 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?

**Solution.** We first observe that the loop in BUILD-MAX-HEAP begins halfway in $A$ because the latter half of the heap represents the leaves (individual nodes are already heaps). Because of the shape property, the heap contains $2^h - j$ nodes with height $j$ at each level of the tree. A node at height $j$ can be swapped down at most $j$ levels. Counting with respect to the number of swap operations, we have at most $T(n) = \sum_{j=0}^{h} j2^{h-j}$ swaps.

Therefore,

$$T(n) = \sum_{j=0}^{h} j2^{h-j} = \sum_{j=0}^{h} j \frac{2^h}{2^j} < \frac{n}{2} \sum_{j=0}^{h} j \frac{1}{2^j}$$

since $n < 2^{h+1}$. (We can assume for simplicity that $n$ is a power of 2).

$$T(n) < \frac{n}{2} \sum_{j=0}^{h} j \frac{1}{2^j} \leq \frac{n}{2} \sum_{j=0}^{\infty} j \frac{1}{2^j} = O(n)$$

If we try to build a heap by running INSERT given an input of size $n$, we will end up with a $O(n \log n)$ running time:

$$T(n) = c \sum_{i=1}^{n} \log i = c \left[ \log 1 + \log 2 + \log 3 + \cdots + \log n \right] = c \log n! = O(n \log n)$$

- Given that both BUILD-MAX-HEAP and HEAPSORT call MAX-HEAPIFY at least $n/2$ times, why does HEAPSORT run in $\Theta(n \log n)$ time and not BUILD-MAX-HEAP?
Solution. Intuitively, the amount of work performed by BUILD-MAX-HEAP is less than that of HEAPSORT. For most nodes $i$ being swapped down the tree in BUILD-MAX-HEAP, the total number of swaps will not be $\Theta(h) = \Theta(\log n)$. No work will be done for half the nodes in the tree at the leaf-level, and at higher heights, the number of nodes that have to do more work decreases exponentially ($/2$ at each level, to be precise). The root is the only node that might have to be swapped down $\log n$ times.

In contrast, in HEAPSORT, at each iteration of the loop we extract the maximum of the heap and have to sift down a value at the root each time. As a result, the amount of comparisons MAX-HEAPIFY will need is always going to be $\Theta(h)$. The tree will shrink as we remove elements, but it doesn’t shrink nearly as fast! The height only decreases by 1 once half of the nodes have been removed.

For the exact math, you can read CLRS for detailed explanations.

• Discuss INSERTION-SORT, MERGESORT, QUICKSORT, and HEAPSORT. What are their relative advantages? When might one sorting algorithm be preferred over the others?

Solution. INSERTION-SORT is simple to implement, efficient for (very) small inputs, adaptive (efficient for mostly sorted inputs), stable, in-place, and online. Terribly inefficient for large inputs (like most quadratic-time sorts). $O(n^2)$ worst case running time, $O(n)$ best case running time. $O(1)$ additional space.

MERGESORT is guaranteed $\Theta(n \log n)$ running time and stable. Better at handling inputs that are slower to access than quicksort. $O(n \log n)$ best and worst case running time. $O(n)$ additional space.

QUICKSORT can be implemented in-place. Randomized quicksort performs (perhaps surprisingly) very well in practice. $O(n \log n)$ best and average case running time, $O(n^2)$ worst case running time. $O(1)$ additional space if in-place.

HEAPSORT is in-place and directly competes with QUICKSORT. Slower in practice than well-implemented QUICKSORT but has better guaranteed worst case running time of $O(n \log n)$. $O(1)$ additional space. Used more frequently in cases with limited memory or systems with real-time constraints/security concerns.

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.

Solution. Take the first $k$ packets of the input stream, and construct a min-heap of size $k$, where IP addresses are inserted into the min-heap and ordered by their frequency. For each IP address in the input, if the frequency is greater than the frequency of the address at the root of the heap, remove the root, insert the new address as the new root, and perform MIN-HEAPIFY. Else, if the frequency of the new address is less than or equal to the frequency of the root, do nothing. After processing all input, return the address at the root of the heap.

Proof of correctness. We want to show that the algorithm, as described above, returns the IP address of the input that is the $k$-th most frequent. Consider any address that enters the heap. By construction, any address that enters the heap must have a frequency that is greater than some other address. Assume for the sake of contradiction that an address $a_{\text{bad}}$ with a frequency greater than order $k$ (i.e., less frequent than the $k$-th most frequent) remains on the heap at the termination of the algorithm. But because we always maintain a heap that has a maximum size of $k$, this implies that some address $a_{\text{good}}$ with frequency order less than or equal to $k$ (more frequent than the $k$-th most-frequent) is excluded from the heap. But by construction, it is impossible for $a_{\text{good}}$ to have been excluded, since it would have compared more frequent...
than $e_{bad}$, which, in turn, would also be more frequent than the root element by transitivity! This is a contradiction, so our algorithm must be correct.

Running time analysis. Constructing a min-heap from the first $k$ elements (unsorted) takes time $O(k)$. We maintain the heap at size $k$ by removing the root in constant time and inserting a new address at the root and percolating downwards. Since each address can be inserted in the heap as the root at most once, each address is percolated downwards to its final position in time $O(\log k)$. Since the input has $n$ addresses, our overall running time is $O(k + n \log k)$. But since $n \gg k$, we have a final complexity of $O(n \log k)$.

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

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

Example: see below

![Binary Tree Diagram](image1)

Figure 1: For this tree, your function should print 1, 2, 3, 7, 6, 5, 4.

Solution

Algorithm: We use a queue to hold nodes that are to be visited. We first start with the queue containing the root node of the tree. While the queue is not empty, we \textbf{dequeue} an element from the queue, mark it as visited, and then \textbf{enqueue} its children into the queue.

- for the tree above, we first start with node 1 in the queue. We remove 1, mark it as visited, and add 2, 3 to the queue.
- We then remove 2 and 7, 6 to the queue. We remove 3 and add 5, 4 to the queue.
- Since all nodes in the queue at this point are leaves, we remove each node one by one until the queue is empty.


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

Example:

![Binary Tree Diagram](image2)

Figure 2: For this tree, your function should print 1, 2, 3, 4, 5, 6, 7.

Hint: Try using 2 stacks.
Solution
We will use two stacks, \(S_1\) and \(S_2\). We will use \(S_1\) to hold elements in the same level that are being printed from left to right, and we will use \(S_2\) to hold elements in the same level that are being printed from right to left. We observe that these stacks are disjoint (i.e., they contain no overlapping elements), and if a given node \(n\) in \(T\) is in \(S_1\), then its two children should be in \(S_2\) (and vice versa).

**Algorithm:** First, push the root of the tree \(T\) onto stack \(S_2\). The following procedure will loop until both \(S_1\) and \(S_2\) are empty.

- While \(S_2\) is not empty, pop the top element \(n\) from \(S_2\). Print \(n\). If \(n\) has a right child, push it onto the other stack \(S_1\). Then, if \(n\) has a left child, push it onto \(S_1\). Continue this step until \(S_2\) is empty.
- While \(S_1\) is not empty, pop the top element \(n\) from \(S_1\). Print \(n\). If \(n\) has a left child, push it onto the other stack \(S_2\). Then, if \(n\) has a right child, push it onto \(S_2\). Continue this step until \(S_1\) is empty.

**Time and space complexity:** If the tree \(T\) contains \(n\) nodes, this solution takes \(O(n)\) time and \(O(n)\) extra space.

Solution
The solution for Problem 4 is very similar to that of Problem 3.

\(\text{enqueue}(x)\):

1. push \(x\) into \(S_1\).

\(\text{dequeue}\):

1. If \(S_2\) is empty, pop all elements from \(S_1\) and push them into \(S_2\).
2. If \(S_2\) is still empty, return Nil.
3. Else pop an element from \(S_2\) and return it.

**Time complexity:** The running time of \(\text{enqueue}(x)\) is clearly \(O(1)\). The running time for \(\text{dequeue}\) is a bit trickier. If we consider that each element will be in each Stack exactly once, then we realize that each element will be pushed exactly twice and popped exactly twice. Thus, the amortized running time of \(\text{dequeue}\) is \(O(1)\).

**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?

**Solution.** We can keep a min-heap and a max-heap simultaneously. The max-heap contains the smaller half of numbers and the min-heap contains the larger half of numbers. Maintain the following two invariants:

1. The difference in size of the max-heap and the size of the min-heap is at most 1.
2. The root of the max-heap is always less than or equal to the root of the min-heap.

For the first two elements of the stream, put the smaller element into the max-heap, and the larger element into the min-heap. Whenever a new element of the stream is encountered, compare it against the root of the max-heap (this is an arbitrary choice, we could have compared to the min-heap root). If it is smaller than the max-heap root, insert in the max-heap. Otherwise, insert it into the min-heap. If invariant \([1]\) is violated, remove the root from the heap of larger size and insert that newly-removed element into the heap of smaller size. To retrieve the median at any given time, if the number of total elements is odd, take the root of the heap with larger size; otherwise, take the average of the roots of both heaps.

**Proof of correctness.** The correctness of the computation of the median from the invariants is immediate. We want to show that our algorithm maintains these invariants. We leave justification of these facts to the reader.

**Running time analysis.** Let \(n\) be the number of elements seen in the stream. In this algorithm, we perform at most two insertions and at most one extraction from heaps of size at most \([n/2]\), which is a
running time that is $O(\log n)$. We can access the roots of the heaps for the median computation in constant time, so finding the median is $O(1)$. For every element of the stream, we maintain our data structures in $O(\log n)$ time. Since every element is stored internally, we use $O(n)$ space.

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:

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

$\rightarrow$

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

Hint: Start with a simpler example:

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

Solution

To solve this problem, we will use the two given stacks, $S_1$ and $S_2$, and two extra variables $\text{max}$ and $\text{size}$.

Algorithm: Initialize $\text{max}$ to $-\infty$ and $\text{size}$ to 0.

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 $\text{max}$. Once we have push’ed all elements into $S_2$, the absolute maximum element will be stored in $\text{max}$.

2. pop all elements from $S_2$ and push all except the maximum element $\text{max}$ back into $S_1$.

3. push the maximum element (stored in $\text{max}$) into $S_2$. Now $S_1$ contains $n - 1$ unsorted elements, and $S_2$ contains 1 sorted element.

4. Increment $\text{size}$ by 1. We will use $\text{size}$ to keep track of the number of sorted elements in $S_2$ so that we don’t pop them.

5. Repeat steps 1-4 until $\text{size} = n$. In Step 2, take care to only pop elements from $S_2$ until $S_2$ contains exactly $\text{size}$ elements. (The bottom $\text{size}$ elements in $S_2$ have already been sorted.)

When the procedure terminates, $S_1$ will be empty, and $S_2$ contains the elements in non-decreasing order.

Time complexity: The running time of our sorting procedure is $O(n^2)$, since for each element that we sort, we must push and pop at most $n$ elements.