Readings

- Lecture Notes Chapter 13: Stacks & Queues
- Lecture Notes Chapter 14: Binary Heaps & Heapsort

Problems

Problem 1
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

Algorithm: We maintain a min-heap and a max-heap simultaneously. At any given time, 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, if the element is smaller than the root of the max-heap, insert it into the max-heap, otherwise insert it into the min-heap. If invariant (1) is violated, remove the root from the larger heap and insert that newly-removed element into the smaller heap. To retrieve the median at any given time, if the number of total elements is odd, take the root of the larger heap; 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 into heaps of size at most $\lceil n/2 \rceil$, 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. Hence, 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 2

Given: A binary tree $T$.
Objective: Print the level order traversal of the tree $T$.
Example:

![Figure 1](image)

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 dequeue an element from the queue, mark it as visited, and then 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.

Time and space complexity: If the tree $T$ contains $n$ nodes, this solution takes $O(n)$ time since we are enqueuing and dequeuing each of the $n$ nodes once and $O(n)$ space for the queue.
Problem 3

Given a full stack $S_1$ of size $n$ and an empty stack $S_2$ of size $n$, 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:

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

$\rightarrow$

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

*Hint: Start with a simpler example:*

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

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.

**Running Time:** 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.
Problem 4
You are given two stacks $S_1$ and $S_2$, each of size $n$.

Implement a queue using $S_1$ and $S_2$. Your queue’s `enqueue` and `dequeue` methods should be implemented using only your stacks’ `push`, `pop`, and/or `peek` methods. What are the running times of your new queue’s `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 $S_2$.
2. If $S_2$ is still empty, return `Nil`.
3. Else pop an element from $S_2$ and return it.

Running Time: The running time of `enqueue(x)` is clearly $O(1)$. The running time for `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 `dequeue` is $O(1)$. 

Additional Practice Problems

Problem 5

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.

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.