Stack and Queue

Xie Haoxuan

The Chinese University of Hong Kong, Shenzhen

October 25, 2021

Difference Between Stack and Queue

- Stack: "LIFO" (Last in first out);
- Queue: "FIFO" (First in first out).

E.g. Push the list [0, 1, 2, 3, 4] into a stack and a queue, respectively. Then pop the stack and queue. What is the order of the popped elements?

Implementation

- Stack: push(), pop(), top(), isEmpty(), size();
- Queue: push(), pop(), front(), tail(), isEmpty(), size();

Refer to the lecture slides.

Applications

The searching algorithms depth-first search (DFS) and breadth-first search (BFS) adopt the principles of stack and queue, respectively. They are widely used in many situations. You may learn them in the later lectures.

Here, we will introduce some advanced models of queue and stack.

Monotonic Queue

The elements in the queue are monotonic. Let's take an advanced sliding window model as an example.
`https://leetcode.com/problems/sliding-window-maximum/`

- Given an array A of n integers. A sliding window of size k is moving from left to right. Find the maximum number in the sliding window at every moment, i.e. calculate the set:

$$\left\{ \max_{i \le j \le i+k-1} \{A_j\} \right\}_{i=1}^{|A|-k+1}$$

Naive Algorithm

Construct a queue for the sliding window. Each time the window moves, pop the queue and push the new element into the queue. Then iterate the queue to find the maximum. Time: $O(|A|k)$.

E.g. A = [1, 3, 2, 4, 5], k = 3;

- Initialize: Queue.push(1), Queue.push(3), Queue.push(2), Queue = [1, 3, 2] with maximum 3;
- First move: Queue.pop(), Queue.push(4), Queue = [3, 2, 4] with maximum 4;
- Second move: Queue.pop(), Queue.push(5), Queue = [2, 4, 5] with maximum 5;
- End: The answer is 3, 4, 5.

Optimized Indexing Structure

Observation: If an element x appears after x' and $x > x'$, then once x is pushed into the queue, the maximum in the current window will never be x'.

Proof: Since x' is in front of x, x' will be popped before x. Therefore, before x' is popped, x, x' exist simultaneously. The maximum will be $\geq x' > x$.

Actually, this observation can be extended to multi-dimensional space. In mathematics, we define it as **partial order**.

Optimized Indexing Structure

With the observation above, we can construct a monotonic queue. The queue should be in decreasing order.

Consider the above example again, A = [1, 3, 2, 4, 5], k = 3.

- Initialize: Queue.push(1), Queue.push(3), since $3 > 1$, Queue.pop(), Queue.push(2), Queue = [3, 2] with maximum 3;
- First move: Queue.push(4), since $4 > 3 > 2$, Queue.pop() twice, Queue = [4] with maximum 4;
- Second move: Queue.push(5), since $5 > 4$, Queue = [5] with maximum 5;
- End: The answer is 3, 4, 5.

Optimized Indexing Structure

Note that we do not need to pop for every move, since the front of the queue is not necessarily at the edge of the window. The optimized part is that we do not need to iterate the whole queue to find the maximum. Since it is monotonic, the front of the queue is already the maximum.

What is the time complexity of this algorithm?

Amortized Analysis

A direct way to think about it: each element is popped and pushed into the queue only once. Therefore, the time complexity is $O(|A|)$.

Actually, there is a rigorous proof for such kind of problems. We can define a potential function:

$$\phi : \bigcup_{i=1}^{|A|} \mathbb{Z}^i \to \mathbb{Z}^+, \phi(\text{queue}) = \text{No. elements in the queue}$$

Then the amortized cost of each move is:

$$\Theta\left(\sum_{a \in \text{queue}} 1(a < x) + \Delta_x \phi(\text{queue})\right) = \Theta(1)$$

Applications

An application of monotonic queue is to optimize dynammic programming. With monotonic queue, we can ignore many unnecessary states.

Follow the definition of monotonic queue, we can similarly define monotonic stack. If you are interested, refer to a sample problem "next greater element" at `https://labuladong.gitbook.io/algo-en/ii.-data-structure/monotonicstack`.