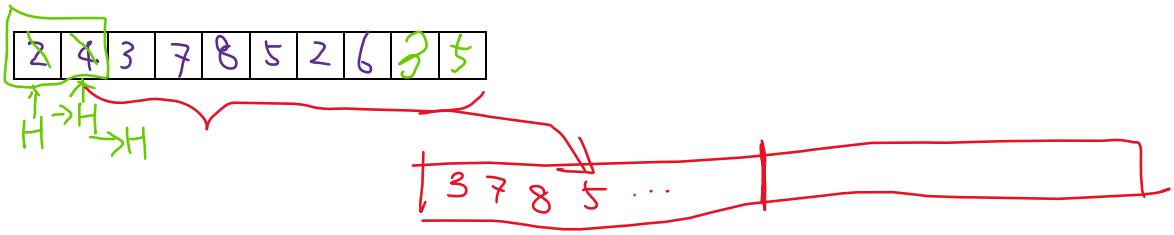
Queue

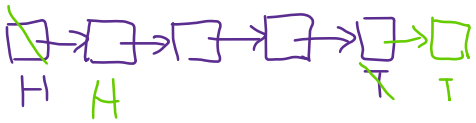
- As mentioned in the last lecture, queue is an ADT which only allows us to append (**enqueue**) items at the tail and remove (**dequeue**) items from the head. These two operations are the most important methods in a queue ADT.
- Similar to stack, we can use both ArrayList and LinkedList to implement a queue.

- In an ArrayList-Stack, we can simply simulate push and pop with operations "$append$" and "$del\ list[-1]$", respectively. Both operations only need constant time. But in a queue, one of the enqueue and dequeue operations has to be accessed from the head of the ArrayList, can we still have that both operations done in constant time?

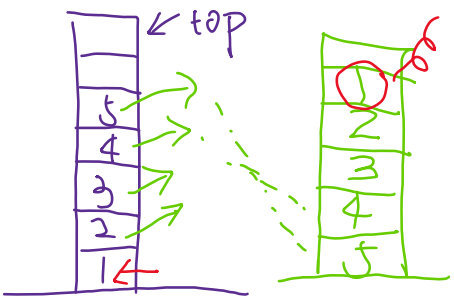


  - To ensure a $O(1)$ time complexity (for enqueue and dequeue), here we use a very space inefficient design: we enqueue to the tail of the list; and after we dequeue from the head of a list, we move the location of the head instead of deleting an item.
    - As an aside, if we are allowed to touch design of the ArrayList that's used here, we can do the follows, so it is not space inefficient anymore: whenever the array size is doubled, copy only those numbers that are still in the queue to the new array.

  - In a Linked-Queue, one of enqueue and dequeue has to be access to from the tail of a LinkedList: this is not a problem since we can use either a Doubly LinkedList or a Singly LinkedList with a tail pointer to achieve $O(1)$ running time in both enqueue and dequeue.

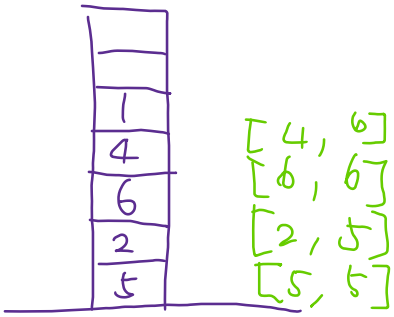


1. How to simulate the operations of a queue with a stack or several stacks (using only stack operation)?



  - If we enqueue from the top, then we need to dequeue from the bottom. While dequeuing, we need to pop out all items above the bottom one first and save them somewhere else then put them back later.
  - If we dequeue from the top, then we need to enqueue from the bottom. Now, we pop out all items while enqueuing.
  - In either design, one of enqueue and dequeue has $\Theta(n)$ time complexity, the other one has $O(1)$ time complexity.

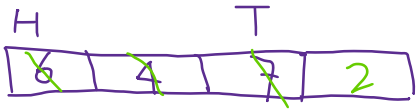
2. While keeping the time complexity of all other operations, how to update a stack (who stores numbers) so that we can find the maximum number stored in it in constant time? How about the same question in a queue?



- The most naïve idea to store an extra numeric attribute max in the data structure. However, it is not possible to update it after popping out the current max within constant time.
- We have the following observation, before the maximum number being pushed to a stack, the previous maximum must be in the stack already; and after we pop out the maximum number, the previous maximum number will become the new max. Thus, we want store a temporary max for each status of the stack. Where should we store it?

- Thus, we can implement the stack as follows:
  - $push$ = push the number and the current max as a length-2 array.
  - pop = regular pop but return the $stack.pop[0]$ instead.
  - $peek$ = return the $stack.peek[0]$.
  - max = return the $stack.peek[1]$.

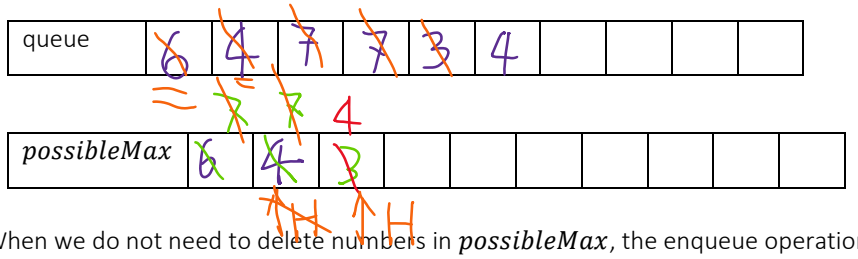
- Does the same implementation work for a queue? No.



- Here we the following observation: Whenever a large number is enqueued, the previous numbers that are smaller do not matter anymore: they won't be the max anyway and they will be dequeued earlier. On the other hand, if small number is enqueued, after the previous larger numbers are dequeued, this small number could be the new max.
- Using this idea, we should store a non-increasing list of possible maximum numbers. Here is our implementation:
  - Keep an extra (Linked)List in the structure called $possibleMax = [\,]$. The maximum number in the queue is always $possibleMax[0]$.
  - $enqueue$: If we enqueue a number k that is smaller than $possibleMax[-1]$ (or we enqueue the first number to the queue), append k to the tail of $possibleMax$; or else, we delete numbers smaller than k in $possibleMax$, then append k to the tail of $possibleMax$.

  Example: enqueue $6,4,7,7,3,4$

dequeue.



When we do not need to delete numbers in $possibleMax$, the enqueue operation is simply appending a number to two lists, it is constant time. When we need to delete m numbers from $possibleMax$,

enqueue + append

enqueue + m deletes + append

then this single enqueue takes $\Theta(m)$ time; however, there must be m appending operations happened in $possibleMax$ before, thus we can amortize this $\Theta(m)$ to those m operations and each enqueue has an amortized cost $O(1)$.

- $dequeue$: If we dequeue a number that is different from $possibleMax[0]$, simply dequeue it; else, delete the head of $possibleMax$ and then the new head of the $possibleMax$ become the new max. Both cases need only a constant time.

  Example: Try to call dequeue in the previous example.

- max: return $possibleMax[0]$.