CHAPTER

8 Stacks and Queues

Linear lists in which insertions, deletions, and accesses to values occur almost always at the first or the last node are very frequently encountered, and we give them special names . . .

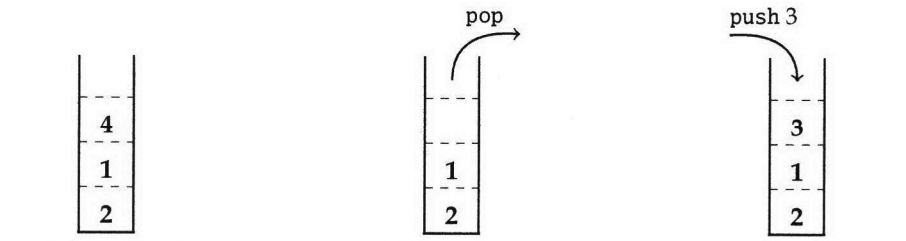
— *"The Art of Computer Programming, Volume 1,"* D. E. KNUTH, 1997

Stacks support last-in, first-out semantics for inserts and deletes, whereas queues are first-in, first-out. Stacks and queues are usually building blocks in a solution to a complex problem. As we will soon see, they can also make for stand-alone problems.

Stacks

A *stack* supports two basic operations—push and pop. Elements are added (pushed) and removed (popped) in last-in, first-out order, as shown in Figure 8.1. If the stack is empty, pop typically returns null or throws an exception.

When the stack is implemented using a linked list these operations have $O(1)$ time complexity. If it is implemented using an array, there is maximum number of entries it can have—push and pop are still $O(1)$. If the array is dynamically resized, the amortized time for both push and pop is $O(1)$. A stack can support additional operations such as peek, which returns the top of the stack without popping it.



(a) Initial configuration. (b) Popping (a). (c) Pushing 3 on to (b).

Figure 8.1: Operations on a stack.

Stacks boot camp

The last-in, first-out semantics of a stack make it very useful for creating reverse iterators for sequences which are stored in a way that would make it difficult or impossible to step back from a given element. This a program uses a stack to print the entries of a singly-linked list in reverse order.

```
def print_linked_list_in_reverse(head):
    nodes = []
    while head:
        nodes.append(head.data)
        head = head.next
    while nodes:
        print(nodes.pop())
```

The time and space complexity are $O(n)$, where n is the number of nodes in the list.

As an alternative, we could form the reverse of the list using Solution 7.2 on Page 85, iterate through the list printing entries, then perform another reverse to recover the list—this would have $O(n)$ time complexity and $O(1)$ space complexity.

Learn to recognize when the stack **LIFO** property is **applicable**. For example, **parsing** typically benefits from a stack. Consider **augmenting** the basic stack or queue data structure to support additional operations, such as finding the maximum element.

Table 8.1: Top Tips for Stacks

Know your stack libraries

Some of the problems require you to implement your own stack class; for others, use the built-in `list`-type.

- `s.append(e)` pushes an element onto the stack. Not much can go wrong with a call to push.
- `s[-1]` will retrieve, but does not remove, the element at the top of the stack.
- `s.pop()` will remove and return the element at the top of the stack.
- `len(s) == 0` tests if the stack is empty.

When called on an empty list `s`, both `s[-1]` and `s.pop()` raise an `IndexError` exception.

8.1 Implement a stack with max API

Design a stack that includes a max operation, in addition to push and pop. The max method should return the maximum value stored in the stack.

Hint: Use additional storage to track the maximum value.

Solution: The simplest way to implement a max operation is to consider each element in the stack, e.g., by iterating through the underlying array for an array-based stack. The time complexity is $O(n)$ and the space complexity is $O(1)$, where n is the number of elements currently in the stack.

The time complexity can be reduced to $O(\log n)$ using auxiliary data structures, specifically, a heap or a BST, and a hash table. The space complexity increases to $O(n)$ and the code is quite complex.

Suppose we use a single auxiliary variable, M, to record the element that is maximum in the stack. Updating M on pushes is easy: $M = \max(M, e)$, where e is the element being pushed. However, updating M on pop is very time consuming. If M is the element being popped, we have

no way of knowing what the maximum remaining element is, and are forced to consider all the remaining elements.

We can dramatically improve on the time complexity of popping by caching, in essence, trading time for space. Specifically, for each entry in the stack, we cache the maximum stored at or below that entry. Now when we pop, we evict the corresponding cached value.

```
class Stack:
    ElementWithCachedMax = collections.namedtuple('ElementWithCachedMax',
                                                   ('element', 'max'))

    def __init__(self):
        self._element_with_cached_max = []

    def empty(self):
        return len(self._element_with_cached_max) == 0

    def max(self):
        if self.empty():
            raise IndexError('max(): empty stack')
        return self._element_with_cached_max[-1].max

    def pop(self):
        if self.empty():
            raise IndexError('pop(): empty stack')
        return self._element_with_cached_max.pop().element

    def push(self, x):
        self._element_with_cached_max.append(
            self.ElementWithCachedMax(x, x if self.empty() else max(
                x, self.max())))
```

Each of the specified methods has time complexity $O(1)$. The additional space complexity is $O(n)$, regardless of the stored keys.

We can improve on the best-case space needed by observing that if an element e being pushed is smaller than the maximum element already in the stack, then e can never be the maximum, so we do not need to record it. We cannot store the sequence of maximum values in a separate stack because of the possibility of duplicates. We resolve this by additionally recording the number of occurrences of each maximum value. See Figure 8.2 on the following page for an example.

```
class Stack:
    class MaxWithCount:
        def __init__(self, max, count):
            self.max, self.count = max, count

    def __init__(self):
        self._element = []
        self._cached_max_with_count = []

    def empty(self):
        return len(self._element) == 0

    def max(self):
        if self.empty():
```

```
            raise IndexError('max(): empty stack')
        return self._cached_max_with_count[-1].max

    def pop(self):
        if self.empty():
            raise IndexError('pop(): empty stack')
        pop_element = self._element.pop()
        current_max = self._cached_max_with_count[-1].max
        if pop_element == current_max:
            self._cached_max_with_count[-1].count -= 1
            if self._cached_max_with_count[-1].count == 0:
                self._cached_max_with_count.pop()
        return pop_element

    def push(self, x):
        self._element.append(x)
        if len(self._cached_max_with_count) == 0:
            self._cached_max_with_count.append(self.MaxWithCount(x, 1))
        else:
            current_max = self._cached_max_with_count[-1].max
            if x == current_max:
                self._cached_max_with_count[-1].count += 1
            elif x > current_max:
                self._cached_max_with_count.append(self.MaxWithCount(x, 1))
```

The worst-case additional space complexity is $O(n)$, which occurs when each key pushed is greater than all keys in the primary stack. However, when the number of distinct keys is small, or the maximum changes infrequently, the additional space complexity is less, $O(1)$ in the best-case. The time complexity for each specified method is still $O(1)$.

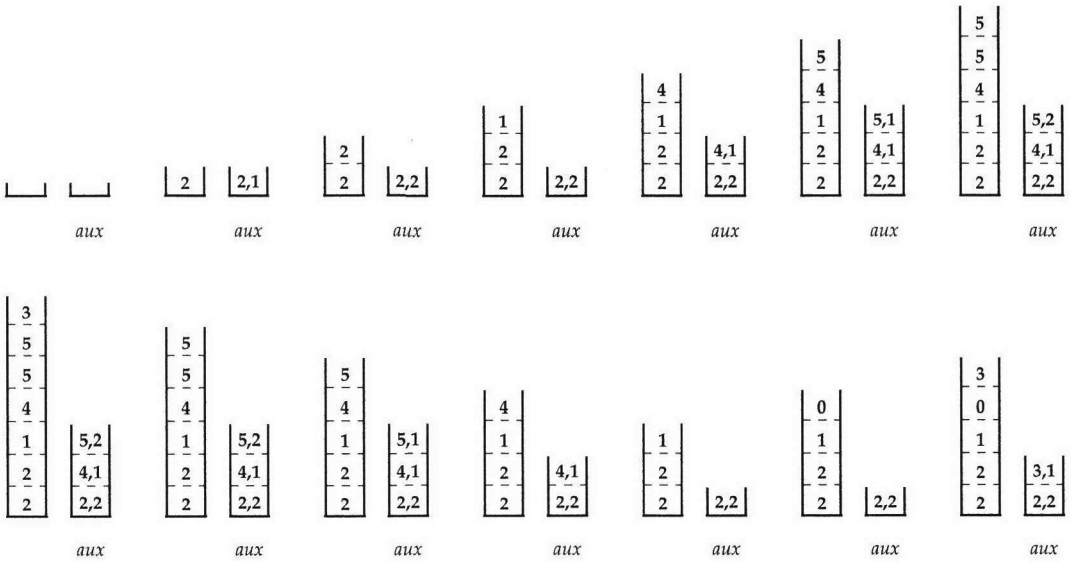


Figure 8.2: The primary and auxiliary stacks for the following operations: push 2, push 2, push 1, push 4, push 5, push 5, push 3, pop, pop, pop, pop, push 0, push 3. Both stacks are initially empty, and their progression is shown from left-to-right, then top-to-bottom. The top of the auxiliary stack holds the maximum element in the stack, and the number of times that element occurs in the stack. The auxiliary stack is denoted by *aux*.

8.2 Evaluate RPN expressions

A string is said to be an arithmetical expression in Reverse Polish notation (RPN) if:

(1.) It is a single digit or a sequence of digits, prefixed with an option $-$, e.g., "6", "123", "-42".

(2.) It is of the form "$A, B, \circ$" where A and B are RPN expressions and $\circ$ is one of $+, -, \times, /$.

For example, the following strings satisfy these rules: "1729", "$3, 4, +, 2, \times, 1, +$", "$1, 1, +, -2, \times$", "$-641, 6, /, 28, /$".

An RPN expression can be evaluated uniquely to an integer, which is determined recursively. The base case corresponds to Rule (1.), which is an integer expressed in base-10 positional system. Rule (2.)corresponds to the recursive case, and the RPNs are evaluated in the natural way, e.g., if A evaluates to 2 and B evaluates to 3, then "$A, B, \times$" evaluates to 6.

Write a program that takes an arithmetical expression in RPN and returns the number that the expression evaluates to.

Hint: Process subexpressions, keeping values in a stack. How should operators be handled?

Solution: Let's begin with the RPN example "$3, 4, +, 2, \times, 1, +$". The ordinary form for this is $(3 + 4) \times 2 + 1$. To evaluate this by hand, we would scan from left to right. We record 3, then 4, then applying the + to 3 and 4, and record the result, 7. Note that we never need to examine the 3 and 4 again. Next we multiply by 2, and record the result, 14. Finally, we add 1 to obtain the final result, 15.

Observe that we need to record partial results, and as we encounter operators, we apply them to the partial results. The partial results are added and removed in last-in, first-out order, which makes a stack the natural data structure for evaluating RPN expressions.

```
def evaluate(RPN_expression):
    intermediate_results = []
    DELIMITER = ','
    OPERATORS = {
        '+': lambda y, x: x + y, '-': lambda y, x: x - y, '*':
        lambda y, x: x * y, '/': lambda y, x: int(x / y)
    }

    for token in RPN_expression.split(DELIMITER):
        if token in OPERATORS:
            intermediate_results.append(OPERATORS[token](
                intermediate_results.pop(), intermediate_results.pop()))
        else:  # token is a number.
            intermediate_results.append(int(token))
    return intermediate_results[-1]
```

Since we perform $O(1)$ computation per character of the string, the time complexity is $O(n)$, where n is the length of the string.

Variant: Solve the same problem for expressions in Polish notation, i.e., when $A, B, \circ$ is replaced by $\circ, A, B$ in Rule (2.).

8.3 Test a string over "{,},(,),[,]" for well-formedness

A string over the characters "{,},(,),[,]" is said to be well-formed if the different types of brackets match in the correct order.

For example, "([]){()}" is well-formed, as is "[()[]{()()}]". However, "{)", "()", and "[()[]{()()" are not well-formed,

Write a program that tests if a string made up of the characters '(', ')', '[', ']',"{' and"}' is well-formed.

Hint: Which left parenthesis does a right parenthesis match with?

Solution: Let's begin with well-formed strings consisting solely of left and right parentheses, e.g., "()(())". If such a string is well-formed, each right parenthesis must match the closest left parenthesis to its left. Therefore, starting from the left, every time we see a left parenthesis, we store it. Each time we see a right parenthesis, we match it with a stored left parenthesis. Since there are not brackets or braces, we can simply keep a count of the number of unmatched left parentheses.

For the general case, we do the same, except that we need to explicitly store the unmatched left characters, i.e., left parenthesis, left brackets, and left braces. We cannot use three counters, because that will not tell us the last unmatched one. A stack is a perfect option for this application: we use it to record the unmatched left characters, with the most recent one at the top.

If we encounter a right character and the stack is empty or the top of the stack is a different type of left character, the right character is not matched, implying the string is not matched. For example, if the input string is "(()][)", when we encounter the first ']', the character at the top of the stack is '(', so the string is not matched. Conversely, if the input string is "(()[])", when we encounter the first ']', the character at the top of the stack is '[', so we continue on. If all characters have been processed and the stack is nonempty, there are unmatched left characters so the string is not matched.

```
def is_well_formed(s):
    left_chars, lookup = [], {'(': ')', '{': '}', '[': ']'}
    for c in s:
        if c in lookup:
            left_chars.append(c)
        elif not left_chars or lookup[left_chars.pop()] != c:
            # Unmatched right char or mismatched chars.
            return False
    return not left_chars
```

The time complexity is $O(n)$ since for each character we perform $O(1)$ operations.

8.4 Normalize pathnames

A file or directory can be specified via a string called the pathname. This string may specify an absolute path, starting from the root, e.g., `/usr/bin/gcc`, or a path relative to the current working directory, e.g., `scripts/awkscripts`.

The same directory may be specified by multiple directory paths. For example, `/usr/lib/../bin/gcc` and `scripts//./../scripts/awkscripts/././` specify equivalent absolute and relative pathnames.

Write a program which takes a pathname, and returns the shortest equivalent pathname. Assume individual directories and files have names that use only alphanumeric characters. Subdirectory names may be combined using forward slashes (/), the current directory (.), and parent directory (..).

Hint: Trace the cases. How should . and .. be handled? Watch for invalid paths.

Solution: It is natural to process the string from left-to-right, splitting on forward slashes (/s). We record directory and file names. Each time we encounter a .., we delete the most recent name, which corresponds to going up directory hierarchy. Since names are processed in a last-in, first-out order, it is natural to store them in a stack. Individual periods (.s) are skipped.

If the string begins with /, then we cannot go up from it. We record this in the stack. If the stack does not begin with /, we may encounter an empty stack when processing .., which indicates a path that begins with an ancestor of the current working path. We need to record this in order to give the shortest equivalent path. The final state of the stack directly corresponds to the shortest equivalent directory path.

For example, if the string is sc//./../tc/awk/././, the stack progression is as follows: ⟨sc⟩, ⟨⟩, ⟨tc⟩, ⟨tc, awk⟩. Note that we skip three .s and the / after sc/.

```
def shortest_equivalent_path(path):
    if not path:
        raise ValueError('Empty string is not a valid path.')

    path_names = []  # Uses list as a stack.
    # Special case: starts with '/', which is an absolute path.
    if path[0] == '/':
        path_names.append('/')

    for token in (token for token in path.split('/') if token not in ['.', '']):
        if token == '..':
            if not path_names or path_names[-1] == '..':
                path_names.append(token)
            else:
                if path_names[-1] == '/':
                    raise ValueError('Path error')
                path_names.pop()
        else:  # Must be a name.
            path_names.append(token)

    result = '/'.join(path_names)
    return result[result.startswith('//'):]  # Avoid starting '//'.
```

The time complexity is $O(n)$, where n is the length of the pathname.

8.5 Compute buildings with a sunset view

You are given a series of buildings that have windows facing west. The buildings are in a straight line, and any building which is to the east of a building of equal or greater height cannot view the sunset.

Design an algorithm that processes buildings in east-to-west order and returns the set of buildings which view the sunset. Each building is specified by its height.

Hint: When does a building not have a sunset view?

Solution: A brute-force approach is to store all buildings in an array. We then do a reverse scan of this array, tracking the running maximum. Any building whose height is less than or equal to the running maximum does not have a sunset view.

The time and space complexity are both $O(n)$, where n is the number of buildings.

Note that if a building is to the east of a taller building, it cannot view the sunset. This suggests a way to reduce the space complexity. We record buildings which potentially have a view. Each new building may block views from the existing set. We determine which such buildings are blocked by comparing the new building's height to that of the buildings in the existing set. We can store the existing set as a hash set—this requires us to iterate over all buildings each time a new building is processed.

If a new building is shorter than a building in the current set, then all buildings in the current set which are further to the east cannot be blocked by the new building. This suggests keeping the buildings in a last-in, first-out manner, so that we can terminate earlier.

Specifically, we use a stack to record buildings that have a view. Each time a building b is processed, if it is taller than the building at the top of the stack, we pop the stack until the top of the stack is taller than b—all the buildings thus removed lie to the east of a taller building.

Although some individual steps may require many pops, each building is pushed and popped at most once. Therefore, the run time to process n buildings is $O(n)$, and the stack always holds precisely the buildings which currently have a view.

The memory used is $O(n)$, and the bound is tight, even when only one building has a view—consider the input where the west-most building is the tallest, and the remaining $n - 1$ buildings decrease in height from east to west. However, in the best-case, e.g., when buildings appear in increasing height, we use $O(1)$ space. In contrast, the brute-force approach always uses $O(n)$ space.

```
def examine_buildings_with_sunset(sequence):
    BuildingWithHeight = collections.namedtuple('BuildingWithHeight',
                                                ('id', 'height'))
    candidates = []
    for building_idx, building_height in enumerate(sequence):
        while candidates and building_height >= candidates[-1].height:
            candidates.pop()
        candidates.append(BuildingWithHeight(building_idx, building_height))
    return [candidate.id for candidate in reversed(candidates)]
```

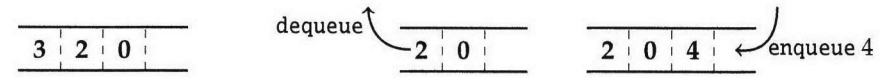
Variant: Solve the problem subject to the same constraints when buildings are presented in west-to-east order.

Queues

A *queue* supports two basic operations—enqueue and dequeue. (If the queue is empty, dequeue typically returns null or throws an exception.) Elements are added (enqueued) and removed (dequeued) in first-in, first-out order. The most recently inserted element is referred to as the tail

or back element, and the item that was inserted least recently is referred to as the head or front element.

A queue can be implemented using a linked list, in which case these operations have $O(1)$ time complexity. The queue API often includes other operations, e.g., a method that returns the item at the head of the queue without removing it, a method that returns the item at the tail of the queue without removing it, etc. A queue can also be implemented using an array; see Problem 8.7 on Page 107 for details.



(a) Initial configuration. (b) Queue (a) after dequeue. (c) Queue (b) after enqueuing 4.

Figure 8.3: Examples of enqueuing and dequeuing.

A *deque*, also sometimes called a double-ended queue, is a doubly linked list in which all insertions and deletions are from one of the two ends of the list, i.e., at the head or the tail. An insertion to the front is commonly called a push, and an insertion to the back is commonly called an inject. A deletion from the front is commonly called a pop, and a deletion from the back is commonly called an eject. (Different languages and libraries may have different nomenclature.)

Queues boot camp

In the following program, we implement the basic queue API—enqueue and dequeue—as well as a max-method, which returns the maximum element stored in the queue. The basic idea is to use composition: add a private field that references a library queue object, and forward existing methods (enqueue and dequeue in this case) to that object.

```
class Queue:
    def __init__(self):
        self._data = collections.deque()

    def enqueue(self, x):
        self._data.append(x)

    def dequeue(self):
        return self._data.popleft()

    def max(self):
        return max(self._data)
```

The time complexity of enqueue and dequeue are the same as that of the library queue, namely, $O(1)$. The time complexity of finding the maximum is $O(n)$, where n is the number of entries. In Solution 8.9 on Page 109 we show how to improve the time complexity of maximum to $O(1)$ with a more customized approach.

Learn to recognize when the queue FIFO property is applicable. For example, queues are ideal when order needs to be preserved.

Table 8.2: Top Tips for Queues

Know your queue libraries

Some of the problems require you to implement your own queue class; for others, use the `collections.deque` class.

- `q.append(e)` pushes an element onto the queue. Not much can go wrong with a call to push.
- `q[0]` will retrieve, but not remove, the element at the front of the queue. Similarly, `q[-1]` will retrieve, but not remove, the element at the back of the queue.
- `q.popleft()` will remove and return the element at the front of the queue.

Dequeing or accessing the head/tail of an empty collection results in an `IndexError` exception being raised.

8.6 Compute binary tree nodes in order of increasing depth

Binary trees are formally defined in Chapter 9. In particular, each node in a binary tree has a depth, which is its distance from the root.

Given a binary tree, return an array consisting of the keys at the same level. Keys should appear in the order of the corresponding nodes' depths, breaking ties from left to right. For example, you should return $\langle\langle 314\rangle, \langle 6, 6\rangle, \langle 271, 561, 2, 271\rangle, \langle 28, 0, 3, 1, 28\rangle, \langle 17, 401, 257\rangle, \langle 641\rangle\rangle$ for the binary tree in Figure 9.1 on Page 112.

Hint: First think about solving this problem with a pair of queues.

Solution: A brute force approach might be to write the nodes into an array while simultaneously computing their depth. We can use preorder traversal to compute this array—by traversing a node's left child first we can ensure that nodes at the same depth are sorted from left to right. Now we can sort this array using a stable sorting algorithm with node depth being the sort key. The time complexity is dominated by the time taken to sort, i.e., $O(n \log n)$, and the space complexity is $O(n)$, which is the space to store the node depths.

Intuitively, since nodes are already presented in a somewhat ordered fashion in the tree, it should be possible to avoid a full-blow sort, thereby reducing time complexity. Furthermore, by processing nodes in order of depth, we do not need to label every node with its depth.

In the following, we use a queue of nodes to store nodes at depth i and a queue of nodes at depth $i + 1$. After all nodes at depth i are processed, we are done with that queue, and can start processing the queue with nodes at depth $i + 1$, putting the depth $i + 2$ nodes in a new queue.

```
def binary_tree_depth_order(tree):
    result = []
    if not tree:
        return result

    curr_depth_nodes = [tree]
    while curr_depth_nodes:
        result.append([curr.data for curr in curr_depth_nodes])
        curr_depth_nodes = [
            child
            for curr in curr_depth_nodes for child in (curr.left, curr.right)
            if child
        ]
    return result
```

Since each node is enqueued and dequeued exactly once, the time complexity is $O(n)$. The space complexity is $O(m)$, where m is the maximum number of nodes at any single depth.

Variant: Write a program which takes as input a binary tree and returns the keys in top down, alternating left-to-right and right-to-left order, starting from left-to-right. For example, if the input is the tree in Figure 9.1 on Page 112, your program should return $\langle\langle 314\rangle, \langle 6, 6\rangle, \langle 271, 561, 2, 271\rangle, \langle 28, 1, 3, 0, 28\rangle, \langle 17, 401, 257\rangle, \langle 641\rangle\rangle$.

Variant: Write a program which takes as input a binary tree and returns the keys in a bottom up, left-to-right order. For example, if the input is the tree in Figure 9.1 on Page 112, your program should return $\langle\langle 641\rangle, \langle 17, 401, 257\rangle, \langle 28, 0, 3, 1, 28\rangle, \langle 271, 561, 2, 271\rangle, \langle 6, 6\rangle, \langle 314\rangle\rangle$.

Variant: Write a program which takes as input a binary tree with integer keys, and returns the average of the keys at each level. For example, if the input is the tree in Figure 9.1 on Page 112, your program should return $\langle 314, 6, 276.25, 12, 225, 641\rangle$.

8.7 Implement a circular queue

A queue can be implemented using an array and two additional fields, the beginning and the end indices. This structure is sometimes referred to as a circular queue. Both enqueue and dequeue have $O(1)$ time complexity. If the array is fixed, there is a maximum number of entries that can be stored. If the array is dynamically resized, the total time for m combined enqueue and dequeue operations is $O(m)$.

Implement a queue API using an array for storing elements. Your API should include a constructor function, which takes as argument the initial capacity of the queue, enqueue and dequeue functions, and a function which returns the number of elements stored. Implement dynamic resizing to support storing an arbitrarily large number of elements.

Hint: Track the head and tail. How can you differentiate a full queue from an empty one?

Solution: A brute-force approach is to use an array, with the head always at index 0. An additional variable tracks the index of the tail element. Enqueue has $O(1)$ time complexity. However dequeue's time complexity is $O(n)$, where n is the number of elements in the queue, since every element has to be left-shifted to fill up the space created at index 0.

A better approach is to keep one more variable to track the head. This way, dequeue can also be performed in $O(1)$ time. When performing an enqueue into a full array, we need to resize the array. We cannot only resize, because this results in queue elements not appearing contiguously. For example, if the array is $\langle e, b, c, d\rangle$, with e being the tail and b the head, if we resize to get $\langle e, b, c, d, \sqcup, \sqcup, \sqcup, \sqcup\rangle$, we cannot enqueue without overwriting or moving elements.

```
class Queue:
    SCALE_FACTOR = 2

    def __init__(self, capacity):
        self._entries = [None] * capacity
        self._head = self._tail = self._num_queue_elements = 0

    def enqueue(self, x):
```

```
        if self._num_queue_elements == len(self._entries):  # Needs to resize.
            # Makes the queue elements appear consecutively.
            self._entries = (
                self._entries[self._head:] + self._entries[:self._head])
            # Resets head and tail.
            self._head, self._tail = 0, self._num_queue_elements
            self._entries += [None] * (
                len(self._entries) * Queue.SCALE_FACTOR - len(self._entries))

        self._entries[self._tail] = x
        self._tail = (self._tail + 1) % len(self._entries)
        self._num_queue_elements += 1

    def dequeue(self):
        if not self._num_queue_elements:
            raise IndexError('empty queue')
        self._num_queue_elements -= 1
        ret = self._entries[self._head]
        self._head = (self._head + 1) % len(self._entries)
        return ret

    def size(self):
        return self._num_queue_elements
```

The time complexity of dequeue is $O(1)$, and the amortized time complexity of enqueue is $O(1)$.

8.8 Implement a queue using stacks

Queue insertion and deletion follows first-in, first-out semantics; stack insertion and deletion is last-in, first-out.

How would you implement a queue given a library implementing stacks?

Hint: It is impossible to solve this problem with a single stack.

Solution: A straightforward implementation is to enqueue by pushing the element to be enqueued onto one stack. The element to be dequeued is then the element at the bottom of this stack, which can be achieved by first popping all its elements and pushing them to another stack, then popping the top of the second stack (which was the bottom-most element of the first stack), and finally popping the remaining elements back to the first stack.

The primary problem with this approach is that every dequeue takes two pushes and two pops of *every* element, i.e., dequeue has $O(n)$ time complexity, where n is the number of stored elements. (Enqueue takes $O(1)$ time.)

The intuition for improving the time complexity of dequeue is that after we move elements from the first stack to the second stack, any further dequeues are trivial, until the second stack is empty. This is true even if we need to enqueue, as long as we enqueue onto the first stack. When the second stack becomes empty, and we need to perform a dequeue, we simply repeat the process of transferring from the first stack to the second stack. In essence, we are using the first stack for enqueue and the second for dequeue.

```
class Queue:
```

```
def __init__(self):
    self._enq, self._deq = [], []

def enqueue(self, x):
    self._enq.append(x)

def dequeue(self):
    if not self._deq:
        # Transfers the elements in _enq to _deq.
        while self._enq:
            self._deq.append(self._enq.pop())

    if not self._deq:  # _deq is still empty!
        raise IndexError('empty queue')
    return self._deq.pop()
```

This approach takes $O(m)$ time for m operations, which can be seen from the fact that each element is pushed no more than twice and popped no more than twice.

8.9 Implement a queue with max API

Implement a queue with enqueue, dequeue, and max operations. The max operation returns the maximum element currently stored in the queue.

Hint: When can an element never be returned by max, regardless of future updates?

Solution: A brute-force approach is to track the current maximum. The current maximum has to be updated on both enqueue and dequeue. Updating the current maximum on enqueue is trivial and fast—just compare the enqueued value with the current maximum. However, updating the current maximum on dequeue is slow—we must examine every single remaining element, which takes $O(n)$ time, where n is the size of the queue.

Consider an element s in the queue that has the property that it entered the queue before a later element, b, which is greater than s. Since s will be dequeued before b, s can never in the future become the maximum element stored in the queue, regardless of the subsequent enqueues and dequeues.

The key to a faster implementation of a queue-with-max is to eliminate elements like s from consideration. We do this by maintaining the set of entries in the queue that have no later entry in the queue greater than them in a separate deque. Elements in the deque will be ordered by their position in the queue, with the candidate closest to the head of the queue appearing first. Since each entry in the deque is greater than or equal to its successors, the largest element in the queue is at the head of the deque.

We now briefly describe how to update the deque on queue updates. If the queue is dequeued, and if the element just dequeued is at the deque's head, we pop the deque from its head; otherwise the deque remains unchanged. When we add an entry to the queue, we iteratively evict from the deque's tail until the element at the tail is greater than or equal to the entry being enqueued, and then add the new entry to the deque's tail. These operations are illustrated in Figure 8.4 on the following page.

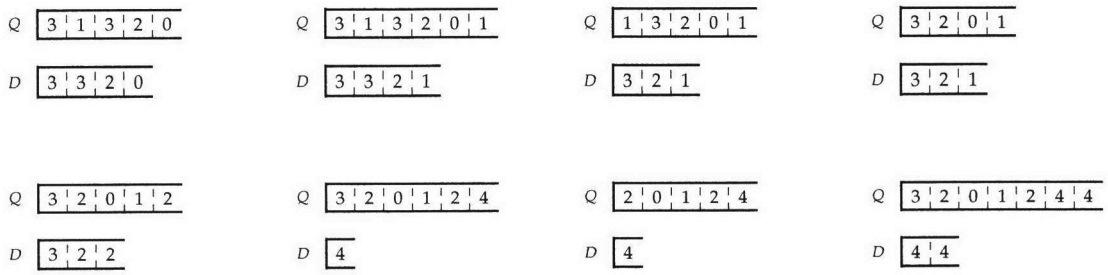


Figure 8.4: The queue with max for the following operations: enqueue 1, dequeue, dequeue, enqueue 2, enqueue 4, dequeue, enqueue 4. The queue initially contains $3, 1, 3, 2$, and 0 in that order. The deque D corresponding to queue Q is immediately below Q. The progression is shown from left-to-right, then top-to-bottom. The head of each queue and deque is on the left. Observe how the head of the deque holds the maximum element in the queue.

```
class QueueWithMax:
    def __init__(self):
        self._entries = collections.deque()
        self._candidates_for_max = collections.deque()

    def enqueue(self, x):
        self._entries.append(x)
        # Eliminate dominated elements in _candidates_for_max.
        while self._candidates_for_max and self._candidates_for_max[-1] < x:
            self._candidates_for_max.pop()
        self._candidates_for_max.append(x)

    def dequeue(self):
        if self._entries:
            result = self._entries.popleft()
            if result == self._candidates_for_max[0]:
                self._candidates_for_max.popleft()
            return result
        raise IndexError('empty queue')

    def max(self):
        if self._candidates_for_max:
            return self._candidates_for_max[0]
        raise IndexError('empty queue')
```

Each dequeue operation has time $O(1)$ complexity. A single enqueue operation may entail many ejections from the deque. However, the amortized time complexity of n enqueues and dequeues is $O(n)$, since an element can be added and removed from the deque no more than once. The max operation is $O(1)$ since it consists of returning the element at the head of the deque.

An alternate solution that is often presented is to use reduction. Specifically, we know how to solve the stack-with-max problem efficiently (Solution 8.1 on Page 98) and we also know how to efficiently model a queue with two stacks (Solution 8.8 on Page 108), so we can solve the queue-with-max design by modeling a queue with two stacks-with-max. This approach feels unnatural compared to the one presented above.

```
class QueueWithMax:
    def __init__(self):
```

```
        self._enqueue = Stack()
        self._dequeue = Stack()

    def enqueue(self, x):
        self._enqueue.push(x)

    def dequeue(self):
        if self._dequeue.empty():
            while not self._enqueue.empty():
                self._dequeue.push(self._enqueue.pop())
        if not self._dequeue.empty():
            return self._dequeue.pop()
        raise IndexError('empty queue')

    def max(self):
        if not self._enqueue.empty():
            return self._enqueue.max() if self._dequeue.empty() else max(
                self._enqueue.max(), self._dequeue.max())
        if not self._dequeue.empty():
            return self._dequeue.max()
        raise IndexError('empty queue')
```

Since the stack-with-max has $O(1)$ amortized time complexity for push, pop, and max, and the queue from two stacks has $O(1)$ amortized time complexity for enqueue and dequeue, this approach has $O(1)$ amortized time complexity for enqueue, dequeue, and max.