Stack and Queue

A stack is a linear data structure that can be accessed only at one of its ends for storing and retrieving data, which follows the rule "Last In, First Out." That is to say, the last element pushed into a stack will be the first one to be popped. Because of this rule, stacks are very useful when data have to be stored and then retrieved in reverse order. The stack is a common data structure in computer-related domains. For instance, operating systems create a stack for each thread to store function parameters, return addresses, and local variables.

We will discuss the characteristics of push and pop sequences in the interview question "Push and Pop Sequence of Stacks" (Question 56). Moreover, stacks are closely related to recursion. More details will be covered in the section *Recursion and Iteration*.

A queue is another important data structure where both ends are used: one for inserting new elements and the other for removing them. Different from stacks, queues follow the rule "First in, First Out," where the first enqueued element will be the first one to be dequeued. It is necessary to utilize queues for breadth-first traversal algorithms.

Usually, elements in stacks or queues are not sorted and it costs O(n) time to get the minimal or maximal element. Special design and implementation are needed if it is required to get the minimum or maximum in O(1) time. A detailed discussion is available in the sections *Stack with min Function* and *Maximum in a Queue.*

Even though the rules for stacks and queues are opposite, what is interesting is that a stack can be implemented with two queues, and a queue can be implemented with two stacks.

Build a Queue with Two Stacks

Question 21 Please design a queue with two stacks and implement methods to enqueue and dequeue items.

It is necessary to implement a queue that follows the rule "First In, First Out" with two stacks that follow the rule of "Last In, First Out." The two stacks inside a queue are denoted as *stack1* and *stack2*.

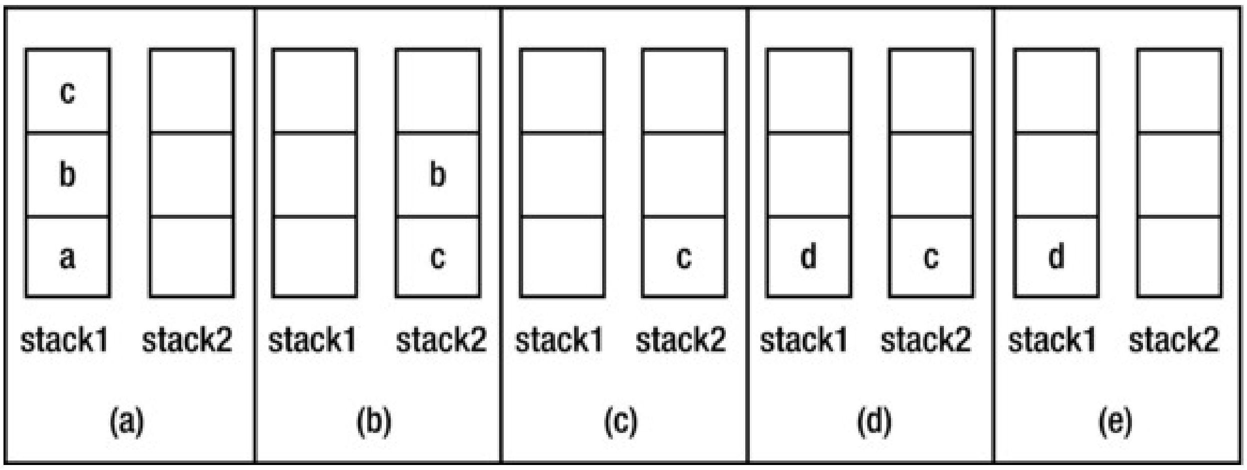

Figure 3-15. *The operations on a queue with two stacks. (a) Enqueue the three elements* a, b, c *one by one. (b) Dequeue the element* a*. (c) Dequeue the element* b *from the front end. (d) Enqueue another element* d*. (e) Dequeue the element* c.

Let us analyze the process required to enqueue and dequeue elements via some examples. First, an element a is appended. Let's push it into *stack1*. There is an element $\{a\}$ in *stack1* and *stack2* that is empty. It continues to append two more elements, b and c, pushing them into *stack1*. There are three elements $\{a, b, c\}$ in *stack1* now, where c is on top and *stack2* is still empty (as shown in Figure 3-15(a)).

We then delete an element from the queue. According to the "First in, First out" rule, the first element to be deleted is a since it was appended before b and c. The element a is stored in *stack1*, and it is not on the top of a stack. Therefore, it cannot be popped off directly. Note that *stack2* has not been used, and it is time for us to utilize it. If elements are popped from *stack1* and pushed into *stack2* one by one, the order of elements in *stack2* is in the reverse order of *stack1*. After three popping and pushing operations, *stack1* becomes empty and there are three elements $\{c, b, a\}$ in *stack2*. The element a can be popped off now since it is on top of *stack2*. Now there are two elements left $\{c, b\}$ in *stack2* and b is on top (as shown in Figure 3-15(b)).

How do we continue deleting more elements from the tail of the queue? The element b was inserted into the queue before c, so it should be deleted when there are two elements b and c left in queue. It can be popped off since it is on top of *stack2*. After the popping operation, *stack1* remains empty and there is only an element c in *stack2*, as shown in Figure 3-15(c).

It is time to summarize the steps to delete an element from a queue. The top of *stack2* can be popped off when *stack2* is not empty since it is the first element inserted into the queue. When *stack2* is empty, we pop all elements from *stack1* and push them into *stack2* one by one. The first element in a queue is pushed into the bottom of *stack1*. It can be popped out directly after popping and pushing operations since it is on top of *stack2*.

Let us enqueue another element, d, and it is pushed into *stack1*, as shown in Figure 3-15(d). If we are going to dequeue an element, the element on top of *stack2*, which is c, needs to be deleted. The element c is indeed inserted into the queue before the element d, so it is a reasonable operation to delete c before d. The final status of the queue is shown in Figure 3-15(e).

We can write code after we have gotten clear ideas about the process necessary to enqueue and dequeue elements. Some sample code in C# is shown in Listing 3-28.

Listing 3-28. *C# Code to Implement a Queue with Two Stacks*

```
public class QueueWithTwoStacks<T> {

 public void Enqueue(T item) {

     stack1.Push(item);
```

```
    }

        public T Dequeue() {
        if (stack2.Count == 0) {
            while (stack1.Count > 0) {
                T item = stack1.Peek();
                stack1.Pop();
                stack2.Push(item);
            }
        }

        if (stack2.Count == 0)
            throw new InvalidOperationException("Queue is Empty");

        T head = stack2.Peek();
        stack2.Pop();

        return head;
    }

    private Stack<T> stack1 = new Stack<T>();
    private Stack<T> stack2 = new Stack<T>();
}
```

Source Code:

```
021_QueueWithTwoStacks.cs
```

Test Cases:

- Insert elements into an empty queue and then delete them
- Insert elements into a non-empty queue and then delete them
- Enqueue and dequeue multiple elements continuously

Build a Stack with Two Queues

Question 22 Please design a stack with two queues and implement the methods to push and pop items.

Similar to the analysis for the previous question, we employ some examples to simulate a stack with two queues, denoted as *queue1* and *queue2*.

An element *a* is pushed into the stack at first. Since its two queues are empty, we could choose any one of them to hold the first element. Supposing *a* is enqueued into *queue1*. It continues to enqueue two more elements, *b* and *c*, into *queue1*. *queue1* has three elements {*a*, *b*, *c*}, and *a* is at its head, *c* is at its tail, as shown in Figure 3-16(a).

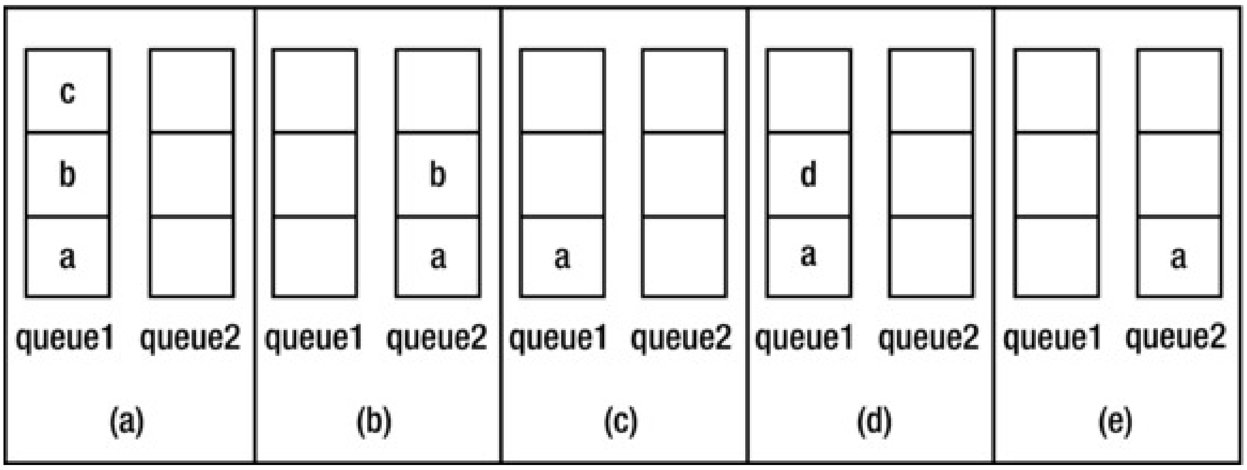

Figure 3-16. *The operations on a stack with two queues. (a) Push the three elements* a, b *and* c *one by one. (b) Pop the element* c. *(c) Pop the element* b *from the top. (d) Push another element* d. *(e) Pop the element* d.

Let us pop an element from the stack. The last element *c* should be popped according to the "Last In, First Out" rule. Only an element at the head of a queue can be deleted, but *c* is at the tail of *queue1*, so it can't be dequeued directly. The two elements *a* and *b* are deleted from queue1 one by one and inserted into *queue2*. Because elements before *c* have been dequeued from *queue1* and *c* is at the head, it can be dequeued now. The status of the stack after *c* is popped is shown in Figure 3-16(b). Similarly, the status of the stack after *b* is popped is shown in Figure 3-16(c).

How about pushing the element *d* into the stack? The item *d* is at the tail of *queue1* after it is enqueued (Figure 3-16(d)). The item *d* is inserted into *queue1* because it is not empty. We should keep one queue empty to accommodate elements of the other queue during popping operations.

If we are going to pop another element, the last pushed element *d* will be popped. Since *d* is not at the head of *queue1*, elements before it have to be dequeued and then enqueued into *queue2*. When *d* is at the head of *queue1*, it is deleted, as shown in Figure 3-16(e).

After we have clear ideas about how to handle pushing and popping operations in a stack with two queues, we can implement such a data structure. The sample code in C# is shown in Listing 3-29.

Listing 3-29. *C# Code to Implement a Stack with Two Queues*

```
public class StackWithTwoQueues<T> {

 public void Push(T item) {

     if (queue2.Count != 0)

         queue2.Enqueue(item);

     else

         queue1.Enqueue(item);

 }


 public T Pop() {

     if (queue1.Count == 0 && queue2.Count == 0)

         throw new InvalidOperationException("Stack is Empty");


     Queue<T> emptyQueue = queue1;

     Queue<T> nonemptyQueue = queue2;

     if (queue1.Count > 0) {

         emptyQueue = queue2;

         nonemptyQueue = queue1;

     }


        while (nonemptyQueue.Count > 1)

         emptyQueue.Enqueue(nonemptyQueue.Dequeue());


     return nonemptyQueue.Dequeue();

 }


 private Queue<T> queue1 = new Queue<T>();

 private Queue<T> queue2 = new Queue<T>();

}
```

Source Code:

```
022_StackWithTwoQueues.cs
```

Test Cases:

- Insert elements into an empty stack and then delete them
- Insert elements into a non-empty stack and then delete them
- Push and pop multiple elements continuously