

5

Stacks and Queues

In this chapter, we will discuss two very important data structures: stacks and queues. Stacks and queues have many important applications, such as form operating system architecture, arithmetic expression evaluation, load balancing, managing printing jobs, and traversing data. In stack and queue data structures, the data is stored sequentially, like arrays and linked lists, but unlike arrays and linked lists, the data is handled in a specific order with certain constraints, which we will be discussing in detail in this chapter. Moreover, we will also examine how we can implement stacks and queues using linked lists and arrays.

In this chapter, we will discuss constraints and methods to handle the data in stacks and queues. We will also implement these data structures and learn how to apply different operations to these data structures in Python.

In this chapter, we will cover the following:

- How to implement stacks and queues using various methods
- Some real-life example applications of stacks and queues

Stacks

A stack is a data structure that stores data, similar to a stack of plates in a kitchen. You can put a plate on the top of the stack, and when you need a plate, you take it from the top of the stack.

The last plate that was added to the stack will be the first to be picked up from the stack:

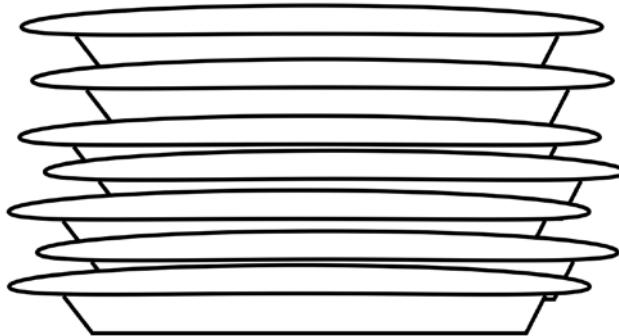


Figure 5.1: Example of a stack

The preceding diagram depicts a stack of plates. Adding a plate to the pile is only possible by leaving that plate on top of the pile. To remove a plate from the pile of plates means to remove the plate that is on top of the pile.

A stack is a data structure that stores the data in a specific order similar to arrays and linked lists, with several constraints:

- Data elements in a stack can only be inserted at the end (push operation)
- Data elements in a stack can only be deleted from the end (pop operation)
- Only the last data element can be read from the stack (peek operation)

A stack data structure allows us to store and read data from one end, and the element which is added last is picked up first. Thus, a stack is a **last in first out (LIFO)** structure, or **last in last out (LILO)**.

There are two primary operations performed on stacks – push and pop. When an element is added to the top of the stack, it is called a push operation, and when an element is to be picked up (that is, removed) from the top of the stack, it is called a pop operation. Another operation is peek, in which the top element of the stack can be viewed without removing it from the stack. All the operations in the stack are performed through a pointer, which is generally named top. All these operations are shown in *Figure 5.2*:

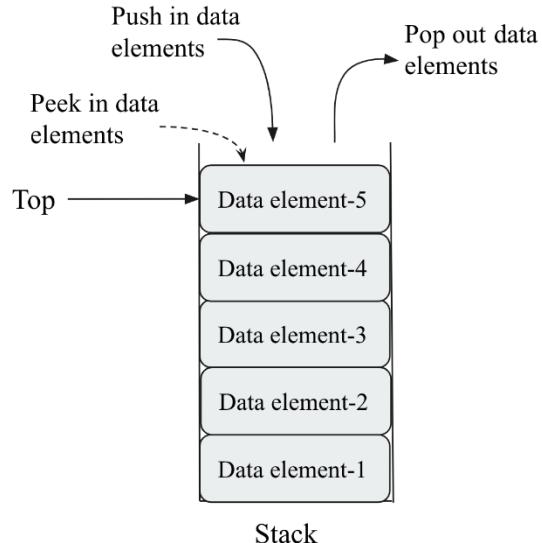


Figure 5.2: Demonstration of push and pop operations in a stack

The following table demonstrates the use of two important stack operations (push and pop) in the stack:

Stack operation	Size	Contents	Operation results
stack()	0	[]	Stack object created, which is empty.
push "egg"	1	['egg']	One item egg is added to the stack.
push "ham"	2	['egg', 'ham']	One more item, ham, is added to the stack.
peek()	2	['egg', 'ham']	The top element, ham, is returned.
pop()	1	['egg']	The ham item is popped off and returned. (This item was added last, so it is removed first.)
pop()	0	[]	The egg item is popped off and returned. (This is the first item added, so it is returned last.)

Table 5.1: Illustration of different operations in a stack with examples

Stacks are used for a number of things. One common usage for stacks is to keep track of the return address during function calls. Let's imagine that we have the following program:

```
def b():
    print('b')

def a():
    b()

a()
print("done")
```

When the program execution gets to the call to `a()`, a sequence of events will be followed in order to complete the execution of this program. A visualization of all these steps is shown in *Figure 5.3*:

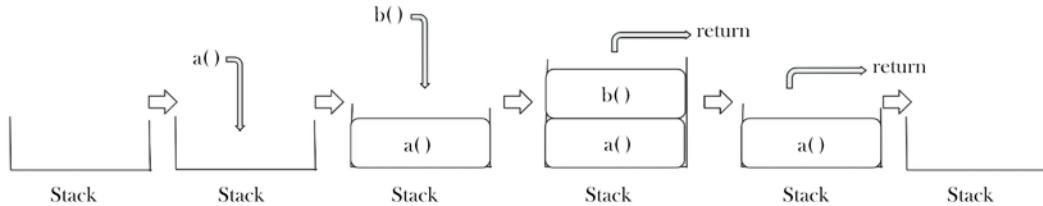


Figure 5.3: Steps for a sequence of events during function calls in our sample program

The sequence of events is as follows:

1. First, the address of the current instruction is pushed onto the stack, and then execution jumps to the definition of `a`
2. Inside function `a()`, function `b()` is called
3. The return address of function `b()` is pushed onto the stack. Once the execution of the instructions and functions in `b()` are complete, the return address is popped off the stack, which takes us back to function `a()`
4. When all the instructions in function `a()` are completed, the return address is again popped off the stack, which takes us back to the main program and the `print` statement

The output of the above program is as follows:

```
b
done
```

We have now discussed the concept of the stack data structure. Now, let us understand its implementation in Python using array and linked list data structures.

Stack implementation using arrays

Stacks store data in sequential order like arrays and linked lists, with a specific constraint that the data can only be stored and read from one end of the stack following the **last in first out (LIFO)** principle. In general, stacks can be implemented using arrays and linked lists. Array-based implementations will have fixed lengths for the stack, whereas linked list-based implementations can have stacks of variable lengths.

In the case of the array-based implementation of a stack (where the stack has a fixed size), it is important to check whether the stack is full or not, since trying to push an element into a full stack will generate an error, called an overflow. Likewise, trying to apply a pop operation to an empty stack causes an error known as an underflow.

Let us understand the implementation of a stack using an array with an example in which we wish to push three data elements, “egg”, “ham”, and “spam”, into the stack. Firstly, to insert new elements into a stack using the push operation, we check the overflow condition, which is when the top pointer is pointing to the end index of the array. The top pointer is the index position of the top element in the stack. If the top element is equal to the overflow condition, the new element cannot be added. This is a stack overflow condition. If there is free space in the array to insert new elements, new data is pushed into the stack. An overview of the push operation on a stack using an array is shown in *Figure 5.4*:

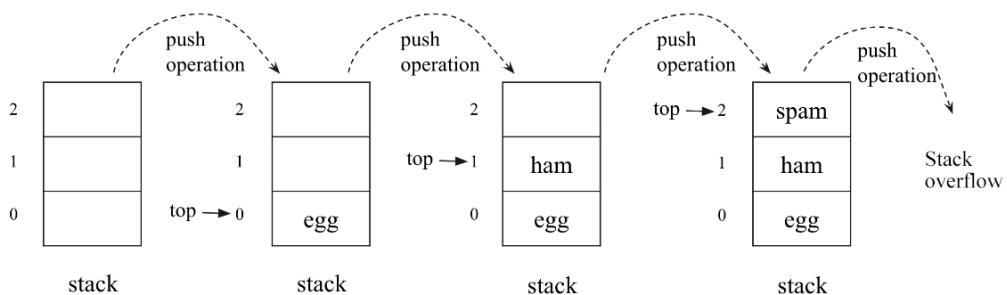


Figure 5.4: Sequence of push operations in a stack implementation using an array

The Python code for the push operation is as follows:

```
size = 3
data = [0] * (size) #Initialize the stack
```

```

top = -1

def push(x):
    global top
    if top >= size - 1:
        print("Stack Overflow")
    else:
        top = top + 1
        data[top] = x

```

In the above code, we initialize the stack with a fixed size (say, 3 in this example), and also the top pointer to -1 , which indicates that the stack is empty. Further, in the push method, the top pointer is compared with the size of the stack to check the overflow condition and, if the stack is full, the stack overflow message is printed. If the stack is not full, the top pointer is incremented by 1, and the new data element is added to the top of the stack. The following code is used to insert data elements into the stack:

```

push('egg')
push('ham')
push('spam')

print(data[0 : top + 1] )

push('new')
push('new2')

```

In the above code, when we try to insert the first three elements, they are added since there was enough space, but when we try to add the data elements new and new2, the stack is already full, hence these two elements cannot be added to the stack. The output of this code is as follows:

```

['egg', 'ham', 'spam']
Stack Overflow
Stack Overflow

```

Next, the pop operation returns the value of the top element of the stack and removes it from the stack. Firstly, we check if the stack is empty or not. If the stack is already empty, a stack underflow message is printed. Otherwise, the top is removed from the stack. An overview of the pop operation is shown in *Figure 5.5*:

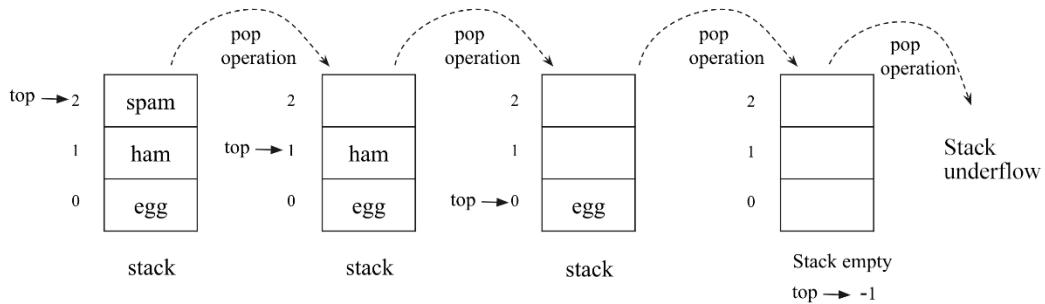


Figure 5.5: Sequence of the pop operation in a stack implementation using an array

The Python code for the pop operation is as follows:

```
def pop():
    global top
    if top == -1:
        print("Stack Underflow")
    else:
        top = top - 1
        data[top] = 0
        return data[top+1]
```

In the above code, we first check the underflow condition by checking whether the stack is empty or not. If the `top` pointer has a value of `-1`, it means the stack is empty. Otherwise, the data elements in the stack are removed by decrementing the `top` pointer by 1, and the `top` data element is returned to the main function.

Let's assume we already added three data elements to the stack, and then we call the `pop` function four times. Since there are only three elements in the stack, the initial three data elements are removed, and when we try to call the `pop` operation a fourth time, the stack underflow message is printed. This is shown in the following code snippet:

```
print(data[0 : top + 1])
pop()
pop()
pop()
pop()
print(data[0 : top + 1])
```

The output of the above code is as follows:

```
['egg', 'ham', 'spam']
Stack Underflow
[]
```

Next, let us see an implementation of the peek operation in which we return the value of the top element of the stack. The Python code for this is as follows:

```
def peek():
    global top
    if top == -1:
        print("Stack is empty")
    else:
        print(data[top])
```

In the above code, firstly, we check the position of the top pointer in the stack. If the value of the top pointer is -1 , it means that the stack is empty, otherwise, we print the value of the top element of the stack.

We have discussed the Python implementation of a stack using an array, so next let us discuss stack implementation using linked lists.

Stack implementation using linked lists

In order to implement the stacks using linked lists, we will write the `Stack` class in which all the methods will be declared; however, we will also use the `node` class similar to what we discussed in the previous chapter:

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None
```

As we know, a node in a linked list holds data and a reference to the next item in the linked list. Implementing the stack data structure using a linked list can be treated as a standard linked list with some constraints, including that elements can be added or removed from the end of the list (push and pop operations) through the top pointer. This is shown in *Figure 5.6*:

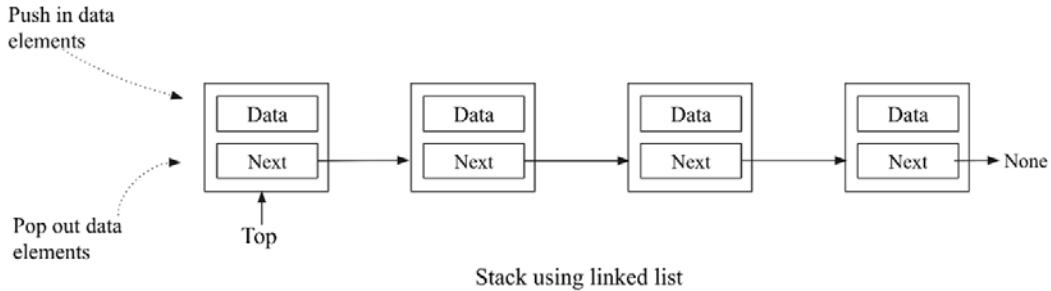


Figure 5.6: Representation of the stack using a linked list

Now let us look at the **stack** class. Its implementation is quite similar to a singly linked list. In addition, we need two things to implement a stack:

1. We first need to know which node is at the top of the stack so that we can apply the push and pop operations through this node
2. We would also like to keep track of the number of nodes in the stack, so we add a **size** variable to the **Stack** class

Consider the following code snippet for the **Stack** class:

```
class Stack:
    def __init__(self):
        self.top = None
        self.size = 0
```

In the above code, we have declared the **top** and **size** variables, which are initialized to **None** and **0**. Once we have initialized the **Stack** class, next, we will implement different operations in the **Stack** class. First, let us start with a discussion of the push operation.

Push operation

The push operation is an important operation on a stack; it is used to add an element at the top of the stack. In order to add a new node to the stack, firstly, we check if the stack already has some items in it or if it is empty. We are not required here to check the overflow condition because we are not required to fix the length of the stack, unlike the stack implementation using arrays.

If the stack already has some elements, then we have to do two things:

1. The new node must have its next pointer pointing to the node that was at the top earlier
2. We put this new node at the top of the stack by pointing **self.top** to the newly added node

See the two instructions in the following *Figure 5.7*:

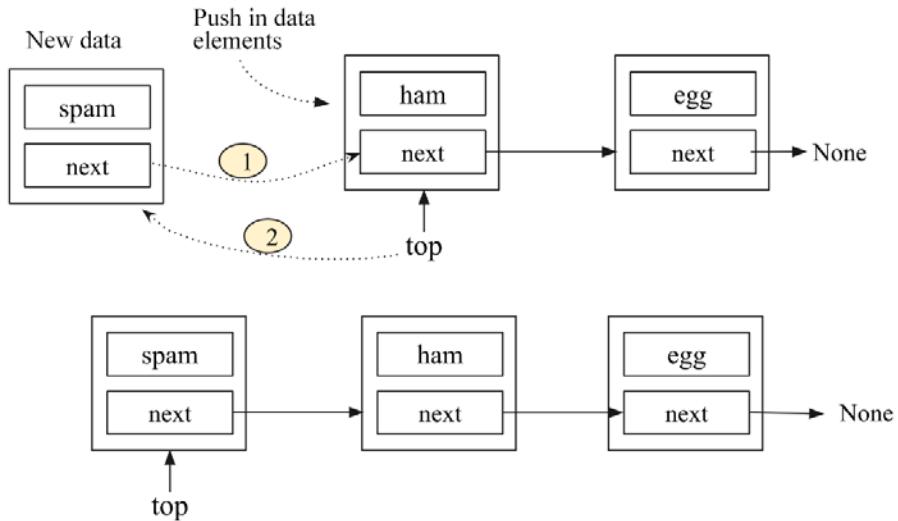


Figure 5.7: Workings of the push operation on the stack

If the existing stack is empty, and the new node to be added is the first element, we need to make this node the top node of the element. Thus, `self.top` will point to this new node. See the following *Figure 5.8*:

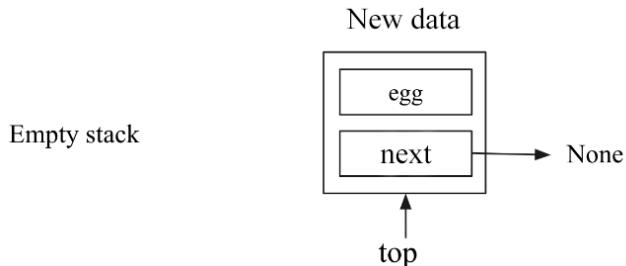


Figure 5.8: Insertion of the data element "egg" into an empty stack

The following is the complete implementation of the push operation, which should be defined in the `Stack` class:

```
def push(self, data):
    # create a new node
    node = Node(data)
    if self.top:
```

```
        node.next = self.top
        self.top = node
    else:
        self.top = node
    self.size += 1
```

In the above code, we create a new node and store the data in that. Then we check the position of the top pointer. If it is not null, that means the stack is not empty, and we add the new node, updating two pointers as shown in *Figure 5.7*. In the else part, we make the top pointer point to the new node. Finally, we increase the size of the stack by incrementing the self.size variable.

To create a stack of three data elements, we use the following code:

```
words = Stack()
words.push('egg')
words.push('ham')
words.push('spam')

#print the stack elements.
current = words.top
while current:
    print(current.data)
    current = current.next
```

The output of the above code is as follows:

```
spam
ham
egg
```

In the above code, we created a stack of three elements – egg, ham, and spam. Next, we will discuss the pop operation in stack data structures.

Pop operation

Another important operation that is applied to the stack is the pop operation. In this operation, the topmost element of the stack is read, and then removed from the stack. The pop method returns the topmost element of the stack and returns None if the stack is empty.

To implement the pop operation on a stack, we do following:

1. First, check if the stack is empty. The pop operation is not allowed on an empty stack.

2. If the stack is not empty, check whether the top node has its next attribute pointing to some other node. If so, it means the stack contains elements, and the topmost node is pointing to the next node in the stack. To apply the pop operation, we have to change the top pointer. The next node should be at the top. We do this by pointing `self.top` to `self.top.next`. See the following *Figure 5.9* to understand this:

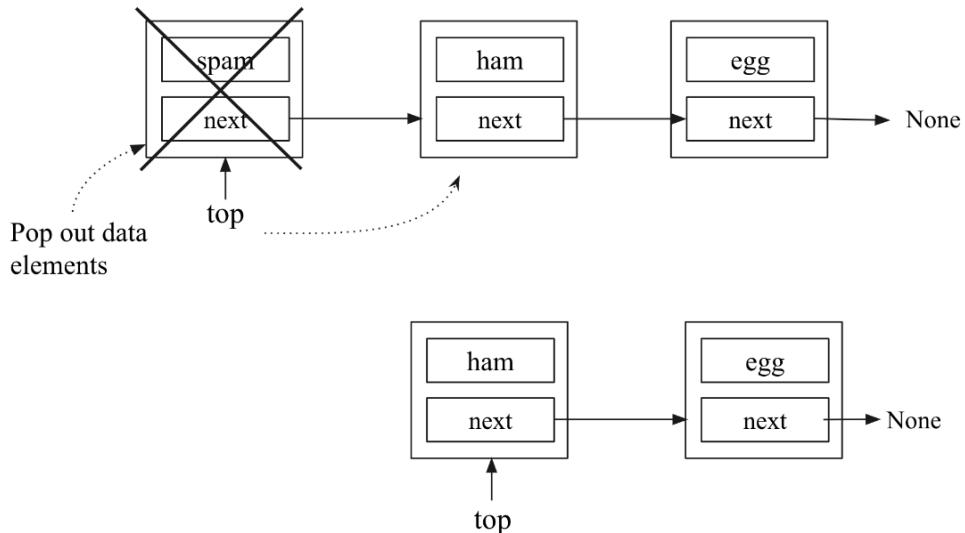


Figure 5.9: Workings of the pop operation on the stack

3. When there is only one node in the stack, the stack will be empty after the pop operation. We have to change the top pointer to None. See the following *Figure 5.10*:

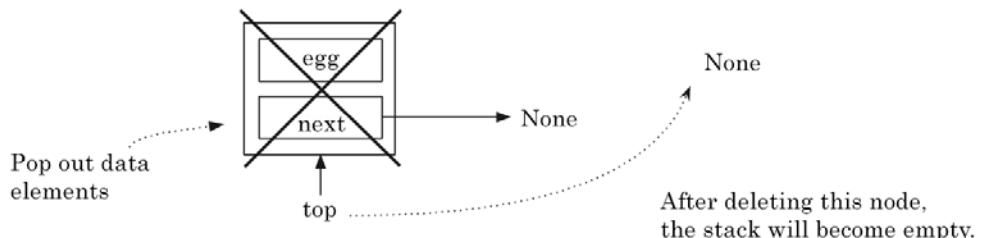


Figure 5.10: The pop operation on a stack with one element

4. Removing this node results in `self.top` pointing to None, as shown in *Figure 5.10*.
5. We also decrement the size of the stack by 1 if the stack is not empty.

Here is the code for the pop operation for the stack in Python, which should be defined in the Stack class:

```
def pop(self):
    if self.top:
        data = self.top.data
        self.size -= 1
        if self.top.next: #check if there is more than one node.
            self.top = self.top.next
        else:
            self.top = None
        return data
    else:
        print("Stack is empty")
```

In the above code, firstly, we check the position of the top pointer. If it is not null, it means the stack is not empty, and we can apply the pop operation such that if there is more than one data element in the stack, we move the top pointer to point to the next node (see *Figure 5.9*), and if that is the last node, we make the top pointer point to None (see *Figure 5.10*). We also decrease the size of the stack by decrementing the self.size variable.

Let's say we have three data elements in a stack. We can use the following code to apply the pop operation to the stack:

```
words.pop()
current = words.top
while current:
    print(current.data)
    current = current.next
```

The output of the above code is as follows:

```
ham
egg
```

In the above code, we popped off the top element from the stack of three elements – egg, ham, spam.

Next, we will discuss the peek operation used on stack data structures.

Peek operation

There is another important operation that can be applied to stacks—the peek method. This method returns the top element from the stack without deleting it from the stack. The only difference between peek and pop is that the peek method just returns the topmost element; however, in the case of a pop method, the topmost element is returned, and that element is also deleted from the stack.

The peek operation allows us to look at the top element without changing the stack. This operation is very straightforward. If there is a top element, return its data; otherwise, return None (thus, the behavior of peek matches that of pop). The implementation of the peek method is as follows (this should be defined in the Stack class):

```
def peek(self):
    if self.top:
        return self.top.data
    else:
        print("Stack is empty")
```

In the above code, we first check the position of the top pointer using `self.top`. If it is not null, this means the stack is not empty, and we return the data value of the topmost node, otherwise, we print the message that the stack is empty. We can use the peek method to get the top element of the stack through the following code:

```
words.peek()
```

The output of the above code is:

```
spam
```

As per our original example of the three data elements being added to the stack, if we use the peek method, we get the top element, `spam`, as an output.

Stacks are an important data structure with several real-world applications. To better understand the concept of the stack, we will discuss one of these applications: bracket matching utilizing stacks.

Applications of stacks

As we know, array and linked list data structures can do whatever the stack or queue data structures (that we will discuss shortly) can do.

Despite this, these data structures are important because of their many applications. For example, in any application, it may be required to add or delete any element in a particular order. stack and queues can be used for this to avoid any potential bug in the program, perhaps accessing/deleting an element from the middle of the list (which can happen in the cases of arrays and linked lists).

Now let us look at an example bracket-matching application and see how we can use our stack to implement it.

Let us write a function `check_brackets` that will verify whether a given expression containing brackets—(), [], or { }—is balanced or not, that is, whether the number of closing brackets matches the number of opening brackets. Stacks can be used for traversing a list of items in reverse order since they follow the **LIFO** rule, which makes them a good choice for this problem.

The following code is for a separate `check_brackets` method defined outside the `Stack` class. This method will use the `Stack` class that we discussed in the previous section. The method takes an expression consisting of alphabetical characters and brackets as input and produces `True` or `False` as output for whether the given expression is valid or not, respectively. The code for the `check_brackets` method is as follows:

```
def check_brackets(expression):
    brackets_stack = Stack()      #The stack class, we defined in previous
    section.
    last = ''
    for ch in expression:
        if ch in ('{', '[', '('):
            brackets_stack.push(ch)
        if ch in ('}', ']', ')'):
            last = brackets_stack.pop()
            if last == '{' and ch == '}':
                continue
            elif last == '[' and ch == ']':
                continue
            elif last == '(' and ch == ')':
                continue
            else:
                return False
        if brackets_stack.size > 0:
            return False
    else:
        return True
```

The above function parses each character in the expression passed to it. If it gets an open bracket, it pushes it onto the stack. If it gets a closing bracket, it pops the top element off the stack and compares the two brackets to make sure their types match- (should match), [should match], and { should match }. If they don't, we return `False`; otherwise, we continue parsing.

Once we reach the end of the expression, we need to do one last check. If the stack is empty, then it is fine and we can return `True`. But if the stack is not empty, then we have an opening bracket that does not have a matching closing bracket and we will return `False`.

We can test the bracket-matcher with the following code:

```
sl = (
    "{{(foo)(bar)}[hello](((this)is)a)test",
    "{{(foo)(bar)}[hello](((this)is)atest",
    "{{(foo)(bar)}[hello](((this)is)a)test))"
)
for s in sl:
    m = check_brackets(s)
    print("{}: {}".format(s, m))
```

Only the first of the three statements should match. When we run the code, we get the following output:

```
{{(foo)(bar)}[hello](((this)is)a)test: True
{{(foo)(bar)}[hello](((this)is)atest: False
{{(foo)(bar)}[hello](((this)is)a)test)): False
```

In the above sample three expressions, we can see that the first expression is valid, while the other two are not valid expressions. Hence, the output of the preceding code is `True`, `False`, and `False`.

In summary, the push, pop, and peek operations of the stack data structure have a time complexity of $O(1)$ since the addition and deletion operations can be directly performed in constant time through the top pointer. The stack data structure is simple; however, it is used to implement many functionalities in real-world applications. For example, the back and forward buttons in web browsers are implemented using stacks. Stacks are also used to implement the undo and redo functionalities in word processors.

We have discussed the stack data structure and its implementations using arrays and linked lists. In the next section, we will discuss the queue data structure and the different operations that can be applied to queues.

Queues

Another important data structure is the queue, which is used to store data similarly to stacks and linked lists, with some constraints and in a specific order. The queue data structure is very similar to the regular queue you are accustomed to in real life. It is just like a line of people waiting to be served in sequential order at a shop. Queues are a fundamentally important concept to grasp since many other data structures are built on them.

A queue works as follows. The first person to join the queue usually gets served first, and everyone will be served in the order in which they joined the queue. The acronym **FIFO** best explains the concept of a queue. **FIFO** stands for **f**irst **i**n, **f**irst **o**ut. When people are standing in a queue waiting for their turn to be served, service is only rendered at the front of the queue. Therefore, people are dequeued from the front of the queue and enqueued from the back where they wait their turn. The only time people exit the queue is when they have been served, which only occurs at the very front of the queue. Refer to the following diagram, where people are standing in the queue, and the person at the front will be served first:

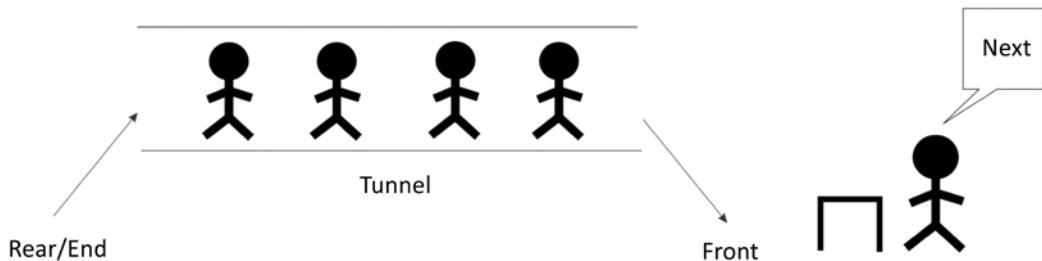


Figure 5.11: Illustration of a queue

To join the queue, participants must stand behind the last person in the queue. This is the only legal or permitted way the queue accepts new entrants. The length of the queue does not matter.

A queue is a list of elements stored in sequence with the following constraints:

1. Data elements can only be inserted from one end, the rear end/tail of the queue.
2. Data elements can only be deleted from the other end, the front/head of the queue.
3. Only data elements from the front of the queue can be read.

The operation to add an element to the queue is known as enqueue. Deleting an element from the queue uses the dequeue operation. Whenever an element is enqueued, the length or size of the queue increments by 1, and dequeuing an item reduces the number of elements in the queue by 1.

We can see this concept in the doubly linked list shown in *Figure 5.12*, in which we can add new elements to the tail/rear end and elements can only be deleted from the head/front end of the queue:

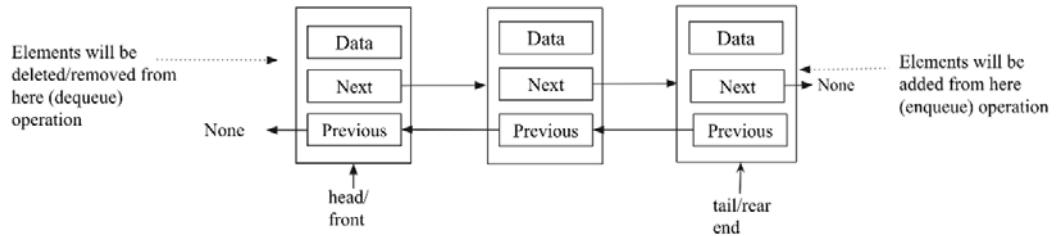


Figure 5.12: Queue implementation using the stack data structure

The reader is advised to not confuse the notation: the enqueue operation will be performed only at the **tail/rear** end and the dequeue operation will be performed from the **head/front** end. It should be fixed that one end will be used for enqueue operations and the other end will be used for dequeue operations; however, either end can be used for each of these operations. It is good in general practice to fix that we perform enqueue operations from the **rear** end and dequeue operations from the **front** end. To demonstrate these two operations, the following table shows the effects of adding and removing elements from a queue:

Queue operation	Size	Contents	Operation results
queue()	0	[]	Queue object created, which is empty.
enqueue- "packt"	1	['packt']	One item, packt, is added to the queue.
enqueue "publishing"	2	['packt', 'publishing']	One more item, publishing, is added to the queue.
Size()	2	['packt', 'publishing']	Return the number of items in the queue, which is 2 in this example.
dequeue()	1	['publishing']	The packt item is dequeued and returned. (This item was added first, so it is removed first.)

<code>dequeue()</code>	0	[]	The publishing item is dequeued and returned. (This is the last item added, so it is returned last.)
------------------------	---	-----	--

Table 5.2: Illustration of different operations on an example queue

Queue data structures in Python have a built-in implementation, `queue.Queue`, and can also be implemented using the `deque` class from the `collections` module. Queue data structures can be implemented using various methods in Python, namely, (1) Python's built-in list, (2) stacks, and (3) node-based linked lists. We will discuss them one by one in detail.

Python's list-based queues

Firstly, in order to implement a queue based on Python's list data structure, we create a `ListQueue` class, in which we declare and define the different functionalities of `queue`. In this method, we store the actual data in Python's list data structure. The `ListQueue` class is defined as follows:

```
class ListQueue:
    def __init__(self):
        self.items = []
        self.front = self.rear = 0
        self.size = 3      # maximum capacity of the queue
```

In the `__init__` initialization method, the `items` instance variable is set to `[]`, which means the queue is empty when created. The size of the queue is also set to 4 (as an example in this code), which is the maximum capacity for the number of elements that can be stored in the queue. Moreover, the initial position of the rear and front indices are also set to 0. `enqueue` and `dequeue` are important methods in queues, and we will discuss them next.

The enqueue operation

The `enqueue` operation adds an item at the end of the queue. Consider the example of adding elements to the queue to understand the concept shown in *Figure 5.13*. We start with an empty list. Initially, we add an item 3 at index 0.

Next, we add an item 11 at index 1, and move the rear pointer every time we add an element:

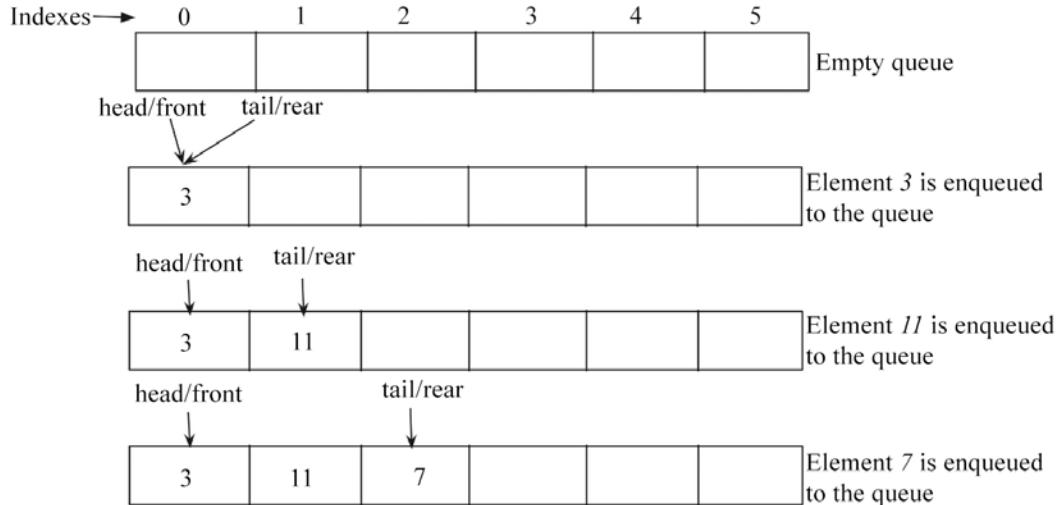


Figure 5.13: Example of an enqueue operation on the queue

In order to implement the enqueue operation, we use the `append` method of the `List` class to append items (or data) to the end of the queue. See the following code for the implementation of the `enqueue` method. This should be defined in the `ListQueue` class:

```
def enqueue(self, data):
    if self.size == self.rear:
        print("\n Queue is full")
    else:
        self.items.append(data)
        self.rear += 1
```

Here, we first check whether the queue is full by comparing the maximum capacity of the queue with the position of the `rear` index. Further, if there is space in the queue, we use the `append` method of the `List` class to add the data at the end of the queue and increase the `rear` pointer by 1.

To create a queue using the `ListQueue` class, we use the following code:

```
q= ListQueue()
q.enqueue(20)
q.enqueue(30)
q.enqueue(40)
```

```
q.enqueue(50)
```

```
print(q.items)
```

The output of the above code is as follows:

```
Queue is full
```

```
[20, 30, 40]
```

In the above code, we add can a maximum of three data elements since we have set the maximum capacity of the queue to be 3. After adding three elements, when we try to add another new element, we get a message that the queue is full.

The dequeue operation

The dequeue operation is used to read and delete items from the queue. This method returns the front item from the queue and deletes it. Consider the example of dequeuing elements from the queue shown in *Figure 5.14*. Here, we have a queue containing elements {3, 11, 7, 1, 4, 2}. In order to dequeue any element from this queue, the element inserted first will be removed first, so the item 3 is removed. When we dequeue any element from the queue, we also decrease the rear pointer by 1:

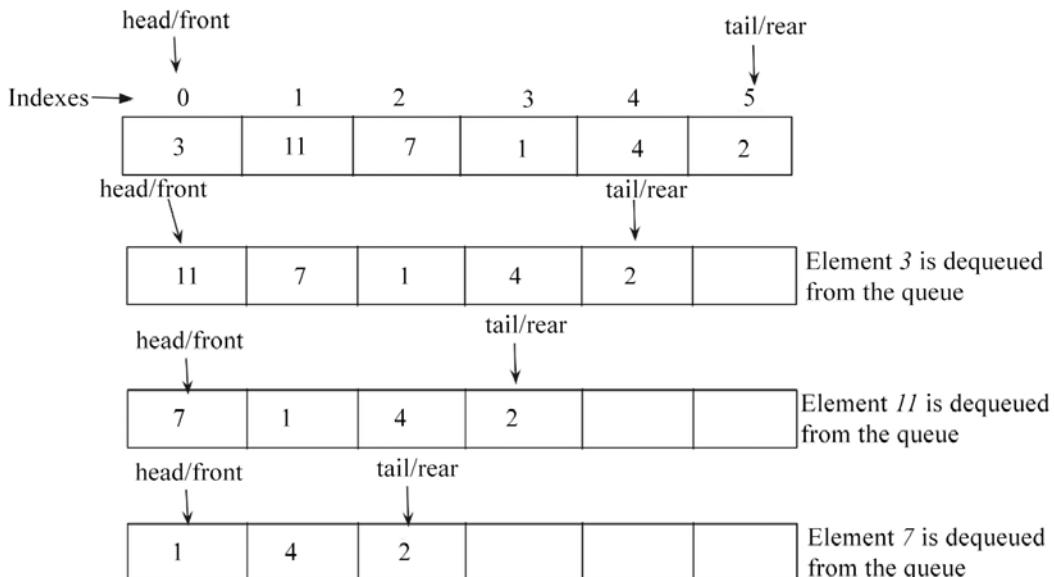


Figure 5.14. Example of a dequeue operation on a queue

The following is the implementation of the `dequeue` method, which should be defined in the `ListQueue` class:

```
def dequeue(self):
    if self.front == self.rear:
        print("Queue is empty")
    else:
        data = self.items.pop(0)      # delete the item from front end
        of the queue
        self.rear -= 1
    return data
```

In the above code, we firstly check whether the queue is already empty by comparing the front and rear pointers. If both rear and front pointers are same, it means the queue is empty. If there are some elements in the queue, we use the `pop` method to dequeue an element. The Python `List` class has a method called `pop()`. The `pop` method does the following:

1. Deletes the last item from the list
2. Returns the deleted item from the list back to the user or code that called it

The item at the first position pointed to by the `front` variable is popped and saved in the `data` variable. We also decrease the `rear` variable by 1, since one data item has been deleted from the queue. Finally, in the last line of the method, the `data` is returned.

To dequeue any element from an existing queue (say items `{20, 30, 40}`), we use the following code:

```
data = q.dequeue()
print(data)
print(q.items)
```

The output of the above code is as follows:

```
20
[30, 40]
```

In the above code, when we dequeue an element from the queue, we get the element 20, which was the first added.

The limitation of this approach to queue implementation is that the length of the queue is fixed, which may be not desirable for an efficient implementation of a queue. Now, let's discuss the linked list-based implementation of queues.

Linked list based queues

A queue data structure can also be implemented using any linked list, such as singly-linked or doubly-linked lists. We already discussed the implementation of singly or doubly linked lists in the previous *Chapter 4, Linked Lists*. We implement queues using linked lists that follow the **FIFO** property of the queue data structure.

Let us discuss the implementation of a queue using a doubly-linked list. For this, we start with the implementation of the node class the same as the node we defined when we discussed doubly-linked lists in the previous *Chapter 4, Linked Lists*. Moreover, the Queue class is very similar to that of the doubly-linked list class. Here, we have head and tail pointers, where tail points to the end of the queue (the rear end) that will be used for adding new elements, and the head pointer points to the start of the queue (the front end) that will be used for dequeuing the elements from the queue. The implementation of the Queue class is shown in the following code:

```
class Node(object):
    def __init__(self, data=None, next=None, prev=None):
        self.data = data
        self.next = next
        self.prev = prev

class Queue:
    def __init__(self):
        self.head = None
        self.tail = None
        self.count = 0
```

Initially, the `self.head` and `self.tail` pointers are set to `None` upon creation of an instance of the Queue class. To keep a count of the number of nodes in Queue, the `count` instance variable is also maintained here and initially set to `0`.

The enqueue operation

Elements are added to a Queue object via the `enqueue` method. The data elements are added through nodes. The `enqueue` method code is very similar to the `append` operation of the doubly-linked list that we discussed in *Chapter 4, Linked Lists*.

The `enqueue` operation creates a node from the data passed to it and appends it to the `tail` of the queue.

Firstly, we check if the new node to be enqueued is the first node, and whether the queue is empty or not. If it is empty, the new node becomes the first node of the queue, as shown in *Figure 5.15*:

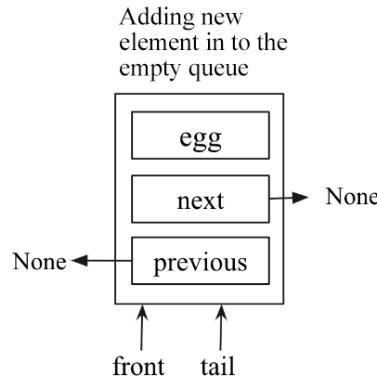


Figure 5.15: Illustration of enqueueing a new node in an empty queue

If the queue is not empty, the new node is appended to the rear end of the queue. In order to do this and enqueue an element to an existing queue, we append the node by updating three links: (1) the previous pointer of the new node should point to the tail of the queue, (2) the next pointer of the tail node should point to the new node, and (3) the tail pointer should be updated to the new node. All these links are shown in *Figure 5.16*:

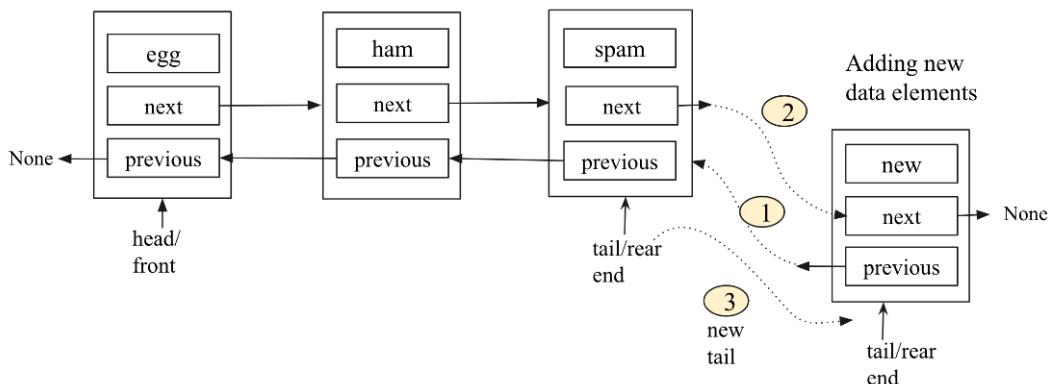


Figure 5.16: Illustration of links to be updated for an enqueue operation in a queue

The enqueue operation is implemented in the Queue class, as shown in the following code:

```
def enqueue(self, data):
    new_node = Node(data, None, None)
    if self.head == None:
```

```

        self.head = new_node
        self.tail = self.head
    else:
        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node

    self.count += 1

```

In the above code, we first check whether the queue is empty or not. If head points to None, this means the queue is empty. If it is empty, the new node is made the first node of the queue, and we make both `self.head` and `self.tail` point to the newly created node. If the queue is not empty, we append the new node to the rear of the queue by updating the three links shown in *Figure 5.16*. Finally, the total count of elements in the queue is increased by the line `self.count += 1`.

The worst-case time complexity of an enqueue operation on the queue is $O(1)$, since any item can be appended directly through the `tail` pointer in constant time.

The dequeue operation

The other operation that makes a doubly-linked list behave like a queue is the `dequeue` method. This method removes the node at the front of the queue, as shown in *Figure 5.17*. Here, we first check whether the dequeuing element is the last node in the queue, and if so, we will make the queue empty after the dequeue operation. If this is not the case, we dequeue the first element by updating the front/head pointer to the next node and the previous pointer of the new head to None, as shown in *Figure 5.17*:

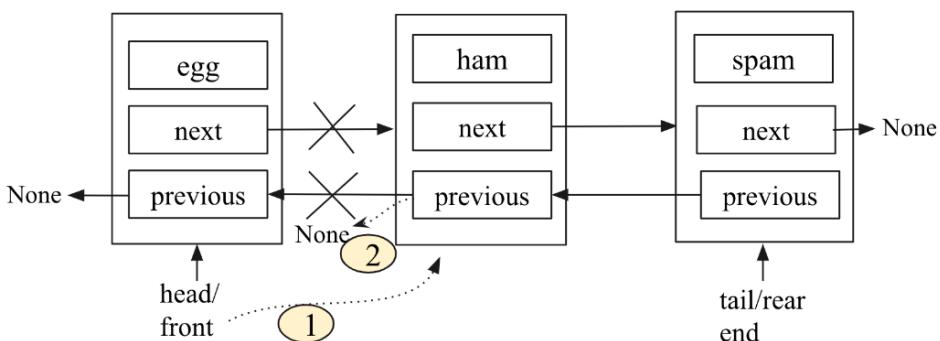


Figure 5.17: Illustration of the dequeue operation on a queue

The implementation of the dequeue operation on a queue is very similar to deleting the first element from the given doubly-linked list, as the following code for the dequeue operation shows:

```
def dequeue(self):
    if self.count == 1:
        self.count -= 1
        self.head = None
        self.tail = None
    elif self.count > 1:
        self.head = self.head.next
        self.head.prev = None
    elif self.count < 1:
        print("Queue is empty")
    self.count -= 1
```

In order to dequeue any element from the queue, we firstly check the number of items in the queue using the `self.count` variable. If the `self.count` variable is equal to 1, it means the dequeuing element is the last element, and we update the head and tail pointers to None.

If the queue has many nodes, then the head pointer is shifted to point to the next node after `self.head` by updating the two links shown in *Figure 5.17*. We also check whether there is an item left in the queue, and if not, then a message is printed that the queue is empty. Finally, the `self.count` variable is decremented by 1.

The worst-case time complexity of a dequeue operation in the queue is $O(1)$, since any item can be directly removed via the head pointer in constant time.

Stack-based queues

A queue is a linear data structure in which enqueue operations are performed from one end and deletion (dequeue) operations are performed from the other end following the **FIFO** principle. There are two methods to implement queues using stacks:

- When the dequeue operation is costly
- When the enqueue operation is costly

Approach 1: When the dequeue operation is costly

We use two stacks for the implementation of the queue. In this approach, the enqueue operation is straightforward. A new element can be enqueued in the queue using the push operation on the first of the two stacks (in other words, Stack-1) used for the implementation of the queue.

The enqueue operation is depicted in *Figure 5.18*, showing an example of enqueueing elements {23, 13, 11} to the queue:

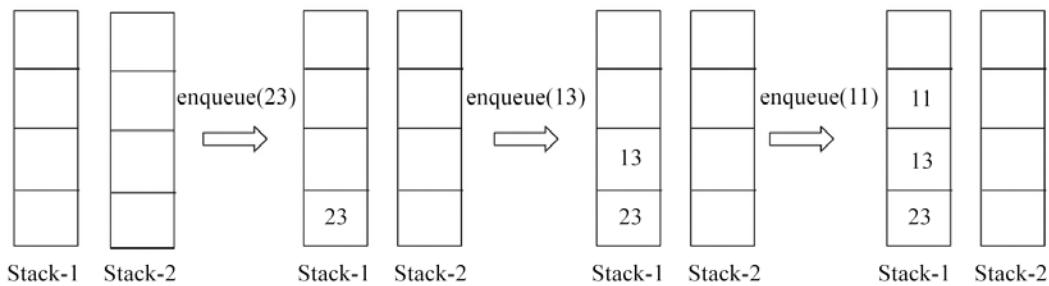


Figure 5.18: Illustration of an enqueue operation in the queue using approach 1

Further more, the dequeue operation can be implemented with two stacks (Stack-1 and Stack-2) using the following steps:

1. Firstly, the elements are removed (popped off) from Stack-1, and then one by one all the elements are added (pushed) to Stack-2.
2. The topmost data element will be popped off Stack-2 and will be returned as the desired element.
3. Finally, the remaining elements are popped off Stack-2 one by one and then pushed again to Stack-1.

Let's look at an example to help understand this concept. Let's say we have three elements stored in the queue {23, 13, 11}, and now we want to dequeue an element from this queue. The complete process is shown in *Figure 5.19* following the above three steps. As you might notice, this implementation follows the **FIFO** property of queues and hence 23 is returned, as it was added first:

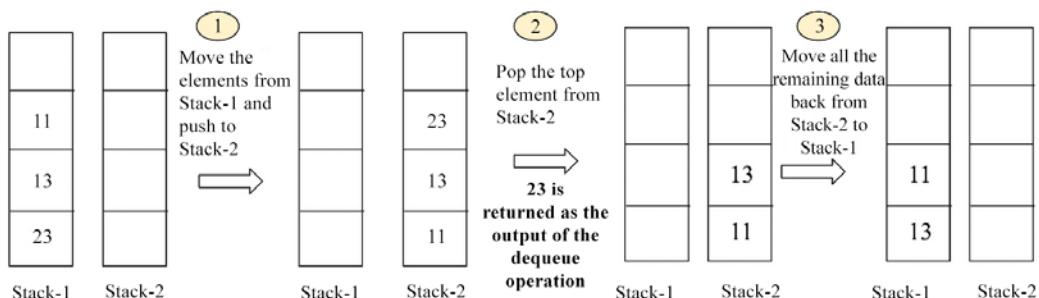


Figure 5.19: Illustration of a dequeue operation in the queue using approach 1

The worst-case time complexity of enqueue operations is $O(1)$, since any element can be added directly to the first stack, and the time complexity of the dequeue operation is $O(n)$, since all elements are accessed and transferred from Stack-1 to Stack-2.

Approach 2: When the enqueue operation is costly

In this method, the enqueue operation is quite similar to the dequeue operation of the previous approach we just discussed, and the dequeue operation is likewise similar to the previous enqueue operation.

In order to implement the enqueue operation, we follow the steps:

1. Move all the elements from Stack-1 to Stack-2.
2. Push the element we want to enqueue to Stack-2.

Move all the elements from Stack-2 to Stack-1 one by one. Pop the elements from Stack-2 and push them to Stack-1.

Let's take an example to understand this concept. Let's say we want to enqueue three elements $\{23, 13, 11\}$ in the queue one by one. We do this by following the above three steps, as shown in *Figure 5.20*, *Figure 5.21*, and *Figure 5.22*:

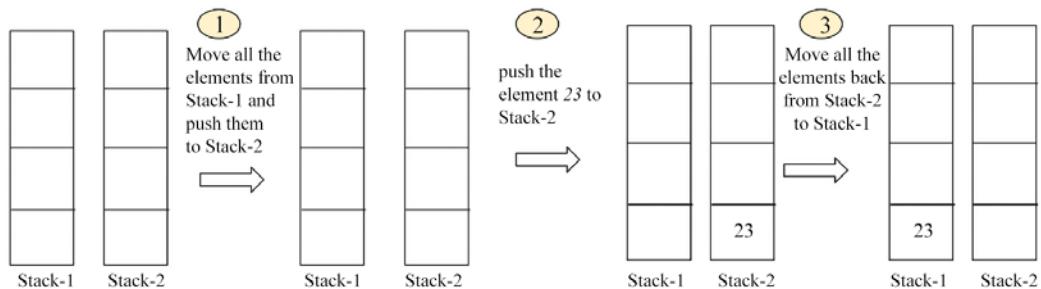


Figure 5.20: Enqueueing element 23 to an empty queue using approach 2

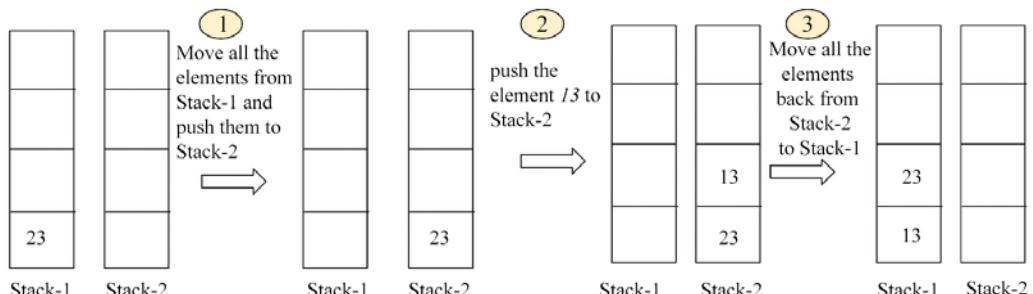


Figure 5.21: Enqueueing element 13 to the existing queue using approach 2

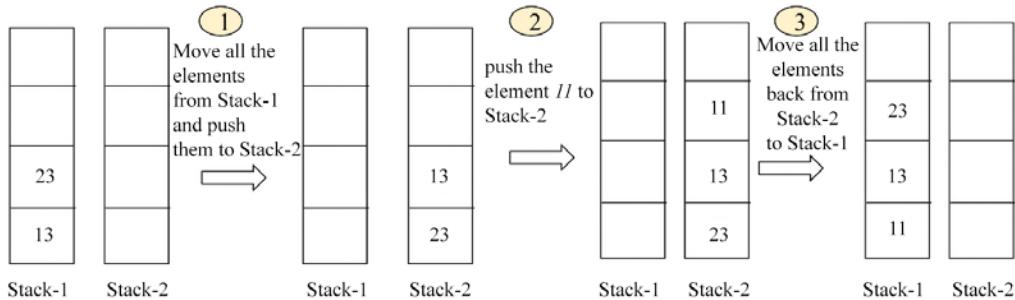


Figure 5.22: Enqueueing element 11 to the queue using approach 2

The dequeue operation can be directly implemented by applying a pop operation to Stack-1. Let's take an example to understand this. Assuming we have already enqueued three elements, and we want to apply the dequeue operation, we can simply pop the top element off the stack, as shown in *Figure 5.23*:

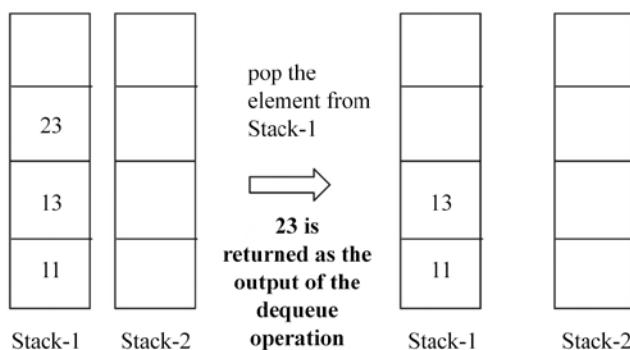


Figure 5.23: Illustration of a dequeue operation on a queue using approach 2

In this second approach, the time complexity for the enqueue operation is $O(n)$, and for the dequeue operation, it is $O(1)$.

Next, we discuss the implementation of a queue using two stacks using approach-1, in which the dequeue operation is costly. In order to implement queues using two stacks, we initially set two stack instance variables to create an empty queue upon initialization. The stacks, in this case, are simply Python lists that allow us to call the push and pop methods on them, which allow us to get the functionality of the enqueue and dequeue operations. Here is the Queue class:

```
class Queue:
    def __init__(self):
        self.Stack1 = []
        self.Stack2 = []
```

Stack1 is only used to store elements that are added to the queue. No other operation can be performed on this stack.

Enqueue operation

The enqueue method is used to add items to the queue. This method only receives the data that is to be appended to the queue. This data is then passed to the append method of Stack1 in the Queue class. Further, the append method is used to mimic the push operation, which pushes elements to the top of the stack. The following code is the implementation of enqueue using the stack in Python, which should be defined in the Queue class:

```
def enqueue(self, data):
    self.Stack1.append(data)
```

To enqueue data onto Stack1, the following code does the job:

```
queue = Queue()
queue.enqueue(23)
queue.enqueue(13)
queue.enqueue(11)
print(queue.Stack1)
```

The output of Stack1 on the queue is as follows:

```
[23, 13, 11]
```

Next, we will examine the implementation of the dequeue operation.

Dequeue operation

The dequeue operation is used to delete the elements from the queue in the same order in which the items were added, according to the **FIFO** principle. New elements are added to the queue in Stack1. Further, we use another stack, that is, Stack2, to delete the elements from the queue. The delete (dequeue) operation will only be performed through Stack2. To better understand how Stack2 can be used to delete the items from the queue, let us consider the following example.

Initially, assume that Stack2 was filled with the elements 5, 6, and 7, as shown in *Figure 5.24*:



Figure 5.24. Example of Stack1 in a queue

Next, we check if the Stack2 is empty or not. As it is empty at the start, we move all the elements delete from Stack1 to Stack2 using the pop operation on Stack1 for all the element and then push them to Stack2. Now, Stack1 becomes empty and Stack2 has all the elements. We show this in *Figure 5.25* for more clarity:

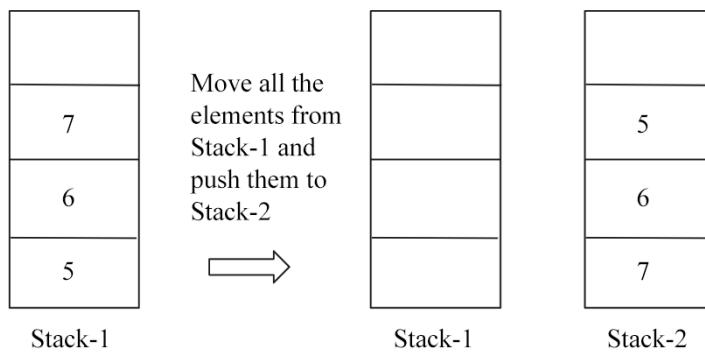


Figure 5.25. Demonstration of Stack1 and Stack2 in a queue

Now, if the Stack is not empty, in order to pop an element from this queue, we apply the pop operation to Stack2, and we get the element 5, which is correct as it was added first and should be the first element to be popped off from the queue.

Here is the implementation of the dequeue method for the queue, which should be defined in the Queue class:

```
def dequeue(self):
    if not self.Stack2:
        while self.Stack1:
            self.Stack2.append(self.Stack1.pop())
    if not self.Stack2:
        print("No element to dequeue")
        return
    return self.Stack2.pop()
```

The `if` statement first checks whether `Stack2` is empty. If it is not empty, we proceed to remove the element at the front of the queue using the `pop` method, as follows:

```
return self.Stack2.pop()
```

If `Stack2` is empty, all the elements of `Stack1` are moved to `Stack2`:

```
while self.Stack1:
    self.Stack2.append(self.Stack1.pop())
```

The `while` loop will continue to be executed as long as there are elements in `Stack1`.

The `self.Stack1.pop()` statement will remove the last element added to `Stack1` and immediately pass the popped data to the `self.Stack2.append()` method.

Let us consider some example code to understand the operations on the queue. We firstly use the `Queue` implementation to add three items to the queue, that is, 5, 6, and 7. Next, we apply `dequeue` operations to remove items from the queue using the following code:

```
queue = Queue()
queue.enqueue(23)
queue.enqueue(13)
queue.enqueue(11)
print(queue.Stack1)

queue.dequeue()
print(queue.Stack2)
```

The output for the preceding code is as follows:

```
[23, 13, 11]  
[13, 11]
```

The preceding code snippet firstly adds elements to a queue and prints out the elements within the queue. Next, the dequeue method is called, after which a change in the number of elements is observed when the queue is printed out again.

The enqueue and dequeue operations on the queue data structure using a stack with approach 1 have time complexities of $O(1)$, and $O(n)$ respectively. The reason for this is that the enqueue operation is straightforward as a new element can be appended directly, whereas in the dequeue operation, all the n elements need to be accessed and moved to the other stack.

Overall, the linked list-based implementation is the most efficient since both the enqueue and dequeue operations can be performed in $O(1)$ time and there is no constraint on the size of the queue. In the stack-based implementation of queues, one of the operations is costly, be it enqueue or dequeue.

Applications of queues

Queues can be used to implement a variety of functionalities in many real computer-based applications. For instance, instead of providing each computer on a network with its own printer, a network of computers can be made to share one printer by queuing what each computer wants to print. When the printer is ready to print, it will pick one of the items (usually called jobs) in the queue to print out. It will print the command from the computer that has given the command first and will choose the following jobs in the order in which they were submitted by the different computers.

Operating systems also queue processes to be executed by the CPU. Let's create an application that makes use of a queue to create a bare-bones media player.

Most music player software allows users to add songs to a playlist. Upon hitting the play button, all the songs in the main playlist are played one after the other. Sequential playing of the songs can be implemented with queues because the first song to be queued is the first song that is to be played. This aligns with the **FIFO** acronym. We will implement our own playlist queue to play songs in the **FIFO** manner.

Our media player queue will only allow for the addition of tracks and a way to play all the tracks in the queue. In a full-blown music player, threads would be used to improve how the queue is interacted with, while the music player continues to be used to select the next song to be played, paused, or even stopped.

The track class will simulate a musical track:

```
from random import randint
class Track:
    def __init__(self, title=None):
        self.title = title
        self.length = randint(5, 10)
```

Each track holds a reference to the title of the song and also the length of the song. The length of the song is a random number between 5 and 10. The `random` module in Python provides the `randint` function to enable us to generate random numbers. The class represents any MP3 track or file that contains music. The random length of a track is used to simulate the number of seconds it takes to play a track.

To create a few tracks and print out their lengths, we do the following:

```
track1 = Track("white whistle")
track2 = Track("butter butter")
print(track1.length)
print(track2.length)
```

The output of the preceding code is as follows:

```
6
7
```

Your output may be different depending on the random length generated for the two tracks.

Now, let's create our queue using inheritance. We simply inherit from the `Queue` class:

```
import time
class MediaPlayerQueue(Queue):
```

To add tracks to the queue, an `add_track` method is created in the `MediaPlayerQueue` class:

```
def add_track(self, track):
    self.enqueue(track)
```

The method passes a track object to the enqueue method of the queue super class. This will, in effect, create a Node using the track object (as the node's data) and point either the tail if the queue is not empty, or both the head and tail if the queue is empty, to this new node.

Assuming the tracks in the queue are played sequentially, from the first track added to the last (FIFO), then the play function has to loop through the elements in the queue:

```
def play(self):
    while self.count > 0:
        current_track_node = self.dequeue()
        print("Now playing {}".format(current_track_node.data.title))
        time.sleep(current_track_node.data.length)
```

self.count keeps count of when a track is added to our queue and when tracks have been dequeued. If the queue is not empty, a call to the dequeue method will return the node (which houses the track object) at the front of the queue. The print statement then accesses the title of the track through the data attribute of the node. To further simulate the playing of a track, the time.sleep() method halts program execution till the number of seconds for the track has elapsed:

```
time.sleep(current_track_node.data.length)
```

The media player queue is made up of nodes. When a track is added to the queue, the track is hidden in a newly created node and associated with the data attribute of the node. That explains why we access a node's track object through the data property of the node returned by the call to dequeue.

You can see that, instead of our node object just storing any data, it stores tracks in this case.

Let's take our music player for a spin:

```
track1 = Track("white whistle")
track2 = Track("butter butter")
track3 = Track("Oh black star")
track4 = Track("Watch that chicken")
track5 = Track("Don't go")
```

We create five track objects with random words as titles, as follows:

```
print(track1.length)
print(track2.length)
```

The output is as follows:

```
8  
9
```

The output may be different from what you get on your machine due to the random length.

Next, an instance of the `MediaPlayerQueue` class is created using the following code snippet:

```
media_player = MediaPlayerQueue()
```

The tracks will be added, and the output of the `play` function should print out the tracks being played in the same order in which we queued them:

```
media_player.add_track(track1)  
media_player.add_track(track2)  
media_player.add_track(track3)  
media_player.add_track(track4)  
media_player.add_track(track5)  
media_player.play()
```

The output of the preceding code is as follows:

```
Now playing white whistle  
Now playing butter butter  
Now playing Oh black star  
Now playing Watch that chicken  
Now playing Don't go
```

Upon execution of the program, it can be seen that the tracks are played in the order in which they were queued. When playing each track, the system also pauses for the number of seconds equal to the length of the track.

Summary

In this chapter, we discussed two important data structures, namely, stacks and queues. We have seen how these data structures closely mimic stacks and queues in the real world. Concrete implementations, together with their varying types, were explored. We later applied the concepts of stacks and queues to write real-life programs.

We will consider trees in the next chapter. The major operations on trees will be discussed, along with the different spheres of application of this data structure.

Exercises

1. Which of the following options is a true queue implementation using linked lists?
 - a. If, in the enqueue operation, new data elements are added at the start of the list, then the dequeue operation must be performed from the end.
 - b. If, in the enqueue operation, new data elements are added to the end of the list, then the enqueue operation must be performed from the start of the list.
 - c. Both of the above.
 - d. None of the above.
2. Assume a queue is implemented using a singly-linked list that has head and tail pointers. The enqueue operation is implemented at the head, and the dequeue operation is implemented at the tail of the queue. What will be the time complexity of the enqueue and dequeue operations?
3. What is the minimum number of stacks required to implement a queue?
4. The enqueue and dequeue operations in a queue are implemented efficiently using an array. What will be the time complexity for both of these operations?
5. How can we print the data elements of a queue data structure in reverse order?

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://packt.link/MEvK4>

