Stack using Linked List

**Implement a stack using a linked list**

Previously, we looked at how to implement a stack using an array. While that approach does work, we saw that it raises some concerns with time complexity. For example, if we exceed the capacity of the array, we have to go through the laborious process of creating a new array and moving over all the elements from the old array.

What if we instead implement the stack using a linked list? Can this improve our time complexity? Let's give it a try.

**1. Define a Node class**

Since we'll be implementing a linked list for this, we know that we'll need a Node class like we used earlier in this lesson.

See if you can remember how to do this, and implement it in the cell below.

**Note:** If you've forgotten, that's completely OK—simply take a look at the solution in order to remind yourself. Then hide the solution, take a short break, and see if you can remember how to do it. Throughout this course, you will find yourself building on concepts you learned earlier. Whenever this is the case, it's good to take this same approach (try to remember, then check the solution, then hide the solution and try to remember again). This effort will help the ideas stick better.

In [1]:



**class** Node:

**def** \_\_init\_\_(self,val):

self.value**=**val

self.next**=None**

Show Solution

**2. Create the Stack class and its \_\_init\_\_ method**

In the cell below, see if you can write the \_\_init\_\_ method for our Stack class. It will need two attributes:

* A head attribute to keep track of the first node in the linked list
* A num\_elements attribute to keep track of how many items are in the stack

In [2]:



**class** Stack:

**def** \_\_init\_\_(self):

self.head**=None**

self.num\_elements**=**0

Show Solution

**3. Add the push method**

Next, we need to define our push method, so that we have a way of adding elements to the top of the stack. First, have a look at the walkthrough:

Walkthrough

In [ ]:



​

Now give it a try for yourself. In the cell below, add the push method:

* The method will need to have a parameter for the value that you want to push
* You'll then need to create a new Node object with this value and link it to the list
* Remember that we want to add new items at the head of the stack, not the tail!
* Once you've added the new node, you'll want to increment num\_elements

In [3]:



**class** Stack:

**def** \_\_init\_\_(self):

self.head **=** **None**

self.num\_elements **=** 0

**def** push(self, value):

new\_node **=** Node(value)

*# if stack is empty*

**if** self.head **is** **None**:

self.head **=** new\_node

**else**:

new\_node.next **=** self.head *# place the new node at the head of the linked list (top)*

self.head **=** new\_node

​

self.num\_elements **+=** 1

Hide Solution

In [4]:



**class** Stack:

**def** \_\_init\_\_(self):

self.head **=** **None**

self.num\_elements **=** 0

**def** push(self, value):

new\_node **=** Node(value)

*# if stack is empty*

**if** self.head **is** **None**:

self.head **=** new\_node

**else**:

new\_node.next **=** self.head *# place the new node at the head of the linked list (top)*

self.head **=** new\_node

​

self.num\_elements **+=** 1

**4. Add the size and is\_empty methods**

When we implemented a stack using an array, we had these same methods. They'll work exactly the same way here—they aren't affected by the use of a linked list versus an array.

* Add a size method that returns the current size of the stack
* Add an is\_empty method that returns True if the stack is empty and False otherwise

In [5]:



**class** Stack:

**def** \_\_init\_\_(self):

self.head **=** **None**

self.num\_elements **=** 0

**def** push(self, value):

new\_node **=** Node(value)

*# if stack is empty*

**if** self.head **is** **None**:

self.head **=** new\_node

**else**:

new\_node.next **=** self.head *# place the new node at the head (top) of the linked list*

self.head **=** new\_node

​

self.num\_elements **+=** 1

**def** size(self):

**return** self.num\_elements

**def** is\_empty(self):

**return** self.num\_elements**==**0

​

Show Solution

**5. Add the pop method**

The last thing we need to do is add the pop method. First, here's a walkthrough that describes how the method works:

Walkthrough

In [ ]:



​

Now try it yourself. The method needs to:

* Check if the stack is empty and, if it is, return None
* Get the value from the head node and put it in a local variable
* Move the head so that it refers to the next item in the list
* Return the value we got earlier

In [6]:



**class** Stack:

**def** \_\_init\_\_(self):

self.head **=** **None**

self.num\_elements **=** 0

**def** push(self, value):

new\_node **=** Node(value)

*# if stack is empty*

**if** self.head **is** **None**:

self.head **=** new\_node

**else**:

new\_node.next **=** self.head *# place the new node at the head (top) of the linked list*

self.head **=** new\_node

​

self.num\_elements **+=** 1

**def** pop(self):

**if** self.head **is** **None**:

**return** **None**

**else**:

t**=**self.head.value

self.head.next**=**self.head.next.next

**return** t

**def** size(self):

**return** self.num\_elements

**def** is\_empty(self):

**return** self.num\_elements **==** 0

File "<ipython-input-6-f707bd4b28e1>", line 19

if self.headis None:

^

SyntaxError: invalid syntax

Show Solution

**Test it!**

Now that we've completed our Stack class, let's test it to make sure it all works as expected.

In [6]:



*# Setup*

stack **=** Stack()

stack.push(10)

stack.push(20)

stack.push(30)

stack.push(40)

stack.push(50)

​

*# Test size*

print ("Pass" **if** (stack.size() **==** 5) **else** "Fail")

​

*# Test pop*

print ("Pass" **if** (stack.pop() **==** 50) **else** "Fail")

​

*# Test push*

stack.push(60)

print ("Pass" **if** (stack.pop() **==** 60) **else** "Fail")

print ("Pass" **if** (stack.pop() **==** 40) **else** "Fail")

print ("Pass" **if** (stack.pop() **==** 30) **else** "Fail")

stack.push(50)

print ("Pass" **if** (stack.size() **==** 3) **else** "Fail")

Pass

Pass

Pass

Pass

Pass

Pass

**Time complexity of stacks using linked lists**

Notice that if we pop or push an element with this stack, there's no traversal. We simply add or remove the item from the head of the linked list, and update the head reference. So with our linked list implementaion, pop and push have a time complexity of O(1).

Also notice that using a linked list avoids the issue we ran into when we implemented our stack using an array. In that case, adding an item to the stack was fine—until we ran out of space. Then we would have to create an entirely new (larger) array and copy over all of the references from the old array.

That happened because, with an array, we had to specify some initial size (in other words, we had to set aside a contiguous block of memory in advance). But with a linked list, the nodes do not need to be contiguous. They can be scattered in different locations of memory, an that works just fine. This means that with a linked list, we can simply append as many nodes as we like. Using that as the underlying data structure for our stack means that we never run out of capacity, so pushing and popping items will always have a time complexity of O(1).