**Stacks**

# **The Adapter Pattern**

The ***adapter*** design pattern applies to any context where we effectively want to modify an existing class so that its methods match those of a related, but different, class or interface. One general way to apply the adapter pattern is to define a new class in such a way that it contains an instance of the existing class as a hidden field, and then to implement each method of the new class using methods of this hidden instance variable. By applying the adapter pattern in this way, we have created a new class that performs some of the same functions as an existing class, but repackaged in a more cconvenient way

|  |  |
| --- | --- |
| **Stack Method** | Realization with Python List |
| S.push(e) | **L.append(e)** |
| **S.pop()** | **L.pop()** |
| **S.top()** | **L[-1]** |
| **S.is\_empty()** | **len(L) == 0** |
| **len(S)** | **len(L)** |

# **Explanation of the Table:**

* **S.push(e) / L.append(e)**:
  + **Purpose**: Adds an element **e** to the top of the stack.
  + **Python List Implementation**: The **append()** method adds an element to the end of the list, which corresponds to the top of the stack in this analogy.
* **S.pop() / L.pop()**:
  + **Purpose**: Removes and returns the top element of the stack.
  + **Python List Implementation**: The **pop()** method removes and returns the last element from the list, which represents the top of the stack.
* **S.top() / L[-1]**:
  + **Purpose**: Returns the top element of the stack without removing it.
  + **Python List Implementation**: Accessing the element at index **-1** retrieves the last element in the list, which is the top of the stack.
* **S.is\_empty() / len(L) == 0**:
  + **Purpose**: Checks if the stack is empty.
  + **Python List Implementation**: Checking if the length of the list is 0 (**len(L) == 0**) determines if the list (stack) is empty.
* **len(S) / len(L)**:
  + **Purpose**: Returns the number of elements in the stack.
  + **Python List Implementation**: The **len()** function returns the number of elements in the list, which corresponds to the size of the stack.

# **Determining the Initial Size of a List**

In Python, when you create a list, the initial size of the underlying array is determined by the Python interpreter. The Python language abstracts away the details of how memory is allocated for lists, but the CPython implementation (the most common implementation of Python) starts with a list that has space for a small number of elements and then resizes it as needed when elements are added or removed.

# **Time Complexity of Array Based Stack Implementation**

* **Push and Pop:** These operations generally run in constant time, O(1), meaning they take the same amount of time regardless of the number of items in the stack. However, because a stack may need to increase or decrease the size of its underlying array, these operations can occasionally take O(n) time, where n is the number of items in the stack. This leads us to the concept of amortized time.
* **Top, Is\_Empty, and Len**: These operations also run in constant time, O(1), as they involve returning a value without modifying the stack's structure.

# **Space Complexity**

The space complexity of a stack refers to the amount of memory it uses in relation to the number of items stored in the stack:

* **Space Complexity**: For a stack, this is O(n), which means the space used by the stack grows linearly with the number of items. If there are n items, the stack uses an amount of memory proportional to n.

# **Amortized Time Complexity**

**Amortized Time Complexity for push and pop**: The **push** and **pop** operations are generally O(1) as well, thanks to the way Python's list is implemented. However, because the list that represents the stack in Python may occasionally need to resize its internal array when it runs out of space (during a **push**) or it's too large for the current number of elements (potentially during a **pop**), these operations can sometimes take O(n) time in the worst case.

Now, the concept of **amortized time** is best understood by looking at the long-term average performance of an operation over a sequence of operations. Even though a single **push** or **pop** might occasionally take O(n) time due to resizing, most of the time, these operations complete in O(1) time. If you average this out over a large number of operations, the rare expensive operations become negligible, and the average time complexity per operation stays O(1).

To explain **amortized time** to a high school student: Imagine you have a chore that takes just a minute to do each day, but once every month, it takes an hour. If you spread out that one long hour across the whole month, each day's chore time is still pretty short when you average it out. Even though once in a while you have a long day, most days are quick and easy. That's like the **push** and **pop** in a stack: usually quick, but every so often a bit longer when you have to get a bigger box to hold all your stuff. The average time, with those long days included, is still pretty short, and that's what we mean by amortized time.

# **Amortized Time Complexity**

"Amortized time" is a term used in computer science to describe an average time per operation over a worst-case sequence of operations. Here's how it applies to push and pop operations:

* **Amortized Time for Push**: In a dynamic array (like Python's list), when you push items onto the stack, it occasionally needs to allocate a larger array to accommodate more items when it runs out of space. This resizing can take O(n) time because it may involve copying all the existing items to the new array. However, this doesn't happen with every push — only after a certain number of operations when the current array is full. When it does occur, the cost is spread out, or "amortized," over all the pushes, and the average time per operation remains O(1).
* **Amortized Time for Pop**: Similarly, the pop operation might sometimes trigger a decrease in the array size to save space when many items have been popped. This resizing is less common in practice but can also be considered in the same amortized sense.

The key takeaway is that while each individual operation could potentially take up to O(n) time due to resizing, the likelihood of that happening is low, and when averaged over a sequence of operations, the time per operation is O(1). This makes push and pop operations very efficient on average, despite the occasional costly operation due to resizing.