Arrays

Overview

- `push()` and `pop()` amortized time complexity
- Finding memory addresses in Python
- Finding the size of items in bytes
- Shallow Copying vs Deep Copying

Push() and Pop() Time Complexity

We say that the time complexity of `O(1)`, but why?:

When we talk about push() (append) and pop() operations having O(1) time complexity, we're referring to their amortized time complexity. Here's what that means:

Push/Append Operation

- When a dynamic array is full and we try to append an element:
    1. A new, larger array is created (typically 2x the size)
    2. All existing elements are copied to the new array
    3. The new element is added
- While this resizing operation is O(n), it happens so infrequently that when we average out the cost over many operations, each append effectively becomes O(1)

Amortized Analysis

- Consider adding n elements:
    - Most pushes cost O(1)
    - Resizing occurs at elements 1, 4, 8, 16, ..., n
    - See the Python source code here
    - Total cost = n (regular pushes) + (1 + 4 + 8 + ... + n/2) (resizing costs)
    - This sums to O(n) operations for n pushes
    - Therefore, each push averages to O(1)

Pop Operation

- For pop() operations from the end of the list:
    - Usually O(1) as we just remove the last element
    - Occasionally O(n) when the array needs to shrink
    - Similar amortized analysis shows O(1) per operation

Finding Memory Addresses in Python

In CPython, the built-in `id()` function returns a unique identifier for an object, which is typically its memory address in the current process (though this is an implementation detail and may vary across different Python implementations).

```
my_list = [1, 2, 3]
print(id(my_list))         # Decimal representation
print(hex(id(my_list)))    # Hexadecimal representation
```

You can use these addresses to check whether two variables refer to the same underlying object:

```
a = [1, 2, 3]
b = a
print(id(a) == id(b))  # True, both refer to the same list object
```

If you need lower-level access to memory, you can use the ctypes module to cast an integer address back to a Python object reference, but this is rarely needed and can be unsafe in practice:

```
import ctypes

x = [10, 20, 30]
address_of_x = id(x)
x_ref = ctypes.cast(address_of_x, ctypes.py_object).value
print(x_ref is x) # prints True
```

Finding the Size of Items in Bytes

Python provides the built-in `sys.getsizeof()` function to determine the memory usage of an object in bytes. This function returns the size of the object itself and does not include the memory usage of any nested objects.

```
import sys

my_list = [1, 2, 3]
print(sys.getsizeof(my_list))

my_str = "Hello"
print(sys.getsizeof(my_str))

my_int = 100
print(sys.getsizeof(my_int))
```

Shallow Copying vs Deep Copying

In Python, a shallow copy creates a new object but does not recursively copy the objects that the original object references. A deep copy creates a new object and recursively copies all objects within the original.

```
import copy

original_list = [[1, 2], [3, 4]]
shallow_copied_list = copy.copy(original_list)
deep_copied_list = copy.deepcopy(original_list)

shallow_copied_list[0][0] = 100
deep_copied_list[1][1] = 400

print(original_list) # [[100, 2], [3, 4]]
print(shallow_copied_list) # [[100, 2], [3, 4]]
print(deep_copied_list) # [[1, 2], [3, 400]]
```

- Shallow Copy (copy.copy()):
    - A new container object is created.
    - References to the original container's elements are used.
- Deep Copy (copy.deepcopy()):
    - A new container object is created.
    - Recursively copies all contained objects rather than just references.

FAQ

At this point you might be wondering what about using a `LinkedList` for a stack, what might be the `push()` and `pop()` runtime. The answer is **always** O(1).

While Python doesn't provide a built-in linked list type with `push()` and `pop()`, you can roll your own. Here's a minimal example:

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

class LinkedStack:
    def __init__(self):
        self.head = None

    def push(self, value):
        new_node = Node(value)
        new_node.next = self.head
        self.head = new_node

    def pop(self):
        if self.head is None:
            raise IndexError("pop from empty stack")
        value = self.head.data
        self.head = self.head.next
        return value
```

Each time you call `push()`, a new node is allocated at a different memory address; there's no need for a contiguous block as in a dynamic array, so this insertion is O(1). Similarly, `pop()` just reassigns the head pointer to the next node, also O(1). This avoids any shifting of elements or extensive copying.```