CSC110 Lecture 26: Abstract Data Types and Stacks

Print this handout

Exercise 1: Using Stacks

Before we get into implementing stacks, we are going to put ourselves in the role of a stack *user*, and attempt to implement the following top-level function (*not* method):

```python
def size(s: Stack) -> int:
    """Return the number of items in s.

    >>> s = Stack()
    >>> size(s)
    0
    >>> s.push('hi')
    >>> s.push('more')
    >>> s.push('stuff')
    >>> size(s)
    3
    """
```

1. Each of the following four implementations of this function has a problem. For each one, explain what the problem is.

   Note: some of these functions may seem to work correctly, but do not exactly follow the given docstring because they mutate the stack `s` as well!

   ```python
   def size(s: Stack) -> int:
       """Return the number of items in s."""
       count = 0
       for _ in s:
           count = count + 1
       return count
   ```

   ```python
   def size(s: Stack) -> int:
       """Return the number of items in s."""
       count = 0
       while not s.is_empty():
           s.pop()
           count = count + 1
       return count
   ```

   ```python
   def size(s: Stack) -> int:
       """Return the number of items in s."""
       return len(s._items)
   ```

   ```python
   def size(s: Stack) -> int:
       """Return the number of items in s."""
       s_copy = s
       count = 0
       while not s_copy.is_empty():
           s_copy.pop()
           count += 1
       return count
   ```

2. Write a correct implementation of the `size` function. You can use the same approach as (b) from the previous question, but use a second, temporary stack to store the items popped off the stack.

```python
def size(s: Stack) -> int:
    """Return the number of items in s.

    >>> s = Stack()
    >>> size(s)
    0
    >>> s.push('hi')
    >>> s.push('more')
    >>> s.push('stuff')
    >>> size(s)
    3
    """
    temp_stack = Stack()

    # Count the items in s by popping them off, but store them in temp_stack








    # Restore the items in s by popping them off of temp_stack








    # Return the count
```

Exercise 2: Stack implementation and running-time analysis

1. Consider the implementation of the Stack we just saw in lecture:

```python
from typing import Any


class Stack1:
    """A last-in-first-out (LIFO) stack of items.

    Stores data in first-in, last-out order. When removing an item from the
    stack, the most recently-added item is the one that is removed.

    >>> s = Stack1()
    >>> s.is_empty()
    True
    >>> s.push('hello')
    >>> s.is_empty()
    False
    >>> s.push('goodbye')
    >>> s.pop()
    'goodbye'
    """
    # Private Instance Attributes:
    #   - _items: The items stored in the stack. The end of the list represents
    #     the top of the stack.
    _items: list

    def __init__(self) -> None:
        """Initialize a new empty stack.
        """
        self._items = []

    def is_empty(self) -> bool:
        """Return whether this stack contains no items.
        """
        return self._items == []

    def push(self, item: Any) -> None:
        """Add a new element to the top of this stack.
        """
        self._items.append(item)

    def pop(self) -> Any:
        """Remove and return the element at the top of this stack.

        Preconditions:
            - not self.is_empty()
        """
        return self._items.pop()
```

   Analyse the running times of the `Stack1.push` and `Stack1.pop` operations in terms of n, the size of the stack.

2. Our implementation of `Stack1` uses the back of its list attribute to store the top of the stack. In the space below, complete the implementation of `Stack2`, which is very similar to `Stack1`, but now uses the *front* of its list attribute to store the top of the stack.

```python
class Stack2:
    """A last-in-first-out (LIFO) stack of items.

    Stores data in first-in, last-out order. When removing an item from the
    stack, the most recently-added item is the one that is removed.

    >>> s = Stack2()
    >>> s.is_empty()
    True
    >>> s.push('hello')
    >>> s.is_empty()
    False
    >>> s.push('goodbye')
    >>> s.pop()
    'goodbye'
    """
    # Private Instance Attributes:
    #   - _items: The items stored in the stack. The FRONT of the list represents
    #     the top of the stack.
    _items: list

    def __init__(self) -> None:
        """Initialize a new empty stack.
        """



    def is_empty(self) -> bool:
        """Return whether this stack contains no items.
        """



    def push(self, item: Any) -> None:
        """Add a new element to the top of this stack.
        """



    def pop(self) -> Any:
        """Remove and return the element at the top of this stack.

        Preconditions:
            - not self.is_empty()
        """
```

3. Analyse the running time of the `Stack2.push` and `Stack2.pop` methods.

4. Based on your answers to Questions 1 and 3, which stack implementation should we use, `Stack1` or `Stack2`?

Additional exercises

Each of the following functions takes at least one stack argument. Analyse the running time of each function *twice*: once assuming it uses `Stack1` as the stack implementation, and again using `Stack2`. (We use the type annotation `Stack` as a placeholder for either `Stack1` or `Stack2`.)

```python
def extra1(s: Stack) -> None:
    s.push(1)
    s.pop()
```

```python
def extra2() -> None:
    s = Stack1()  # Or, s = Stack2()

    for i in range(0, 5):
        s.push(i)
```

```python
def extra3(s: Stack, k: int) -> None:
    """Precondition: k >= 0"""
    for i in range(0, k):
        s.push(i)
```

```python
def extra4(s1: Stack) -> None:
    s2 = Stack1()  # Or, s2 = Stack2()

    while not s1.is_empty():
        s2.push(s1.pop())

    while not s2.is_empty():
        s1.push(s2.pop())
```