CSC 2720 – Data Structures

Dynamic Arrays, Stacks

Course Announcements

- **Lab Schedule**

- **Section 20**

Time: 4:00 – 4:50 PM

Location: Langdale 405

- **Section 24**

Time: 3:45 – 4:35 PM

Location: Classroom South 305

All students are kindly reminded to attend only the lab section they are officially enrolled in.

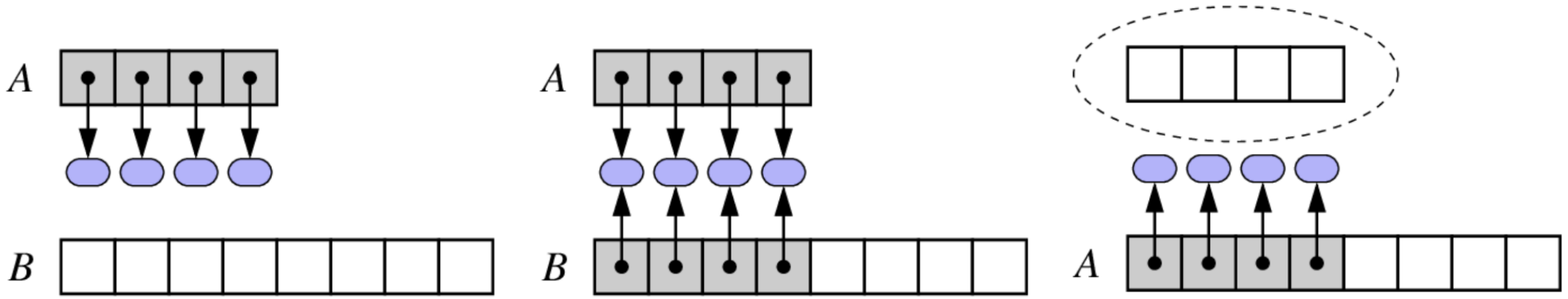
Dynamic arrays

- A list instance maintains an underlying array that often has greater capacity than the current length of the list.
- For example, while a user may have created a list with **five** elements, the system may have reserved an underlying array capable of storing **eight** object references.
- This extra capacity makes it easy to append a new element to the list by using the next available cell of the array.

Dynamic arrays

- If a user continues to append elements to a list, any reserved capacity will eventually be exhausted
- In that case, the class requests a new, larger array from the system, and initializes the new array using the data from old array.
- At that point in time, the old array is no longer needed, so it is reclaimed by the system.

Dynamic arrays – Implementation



(a) creating a new array B

(b) storing elements of A in B

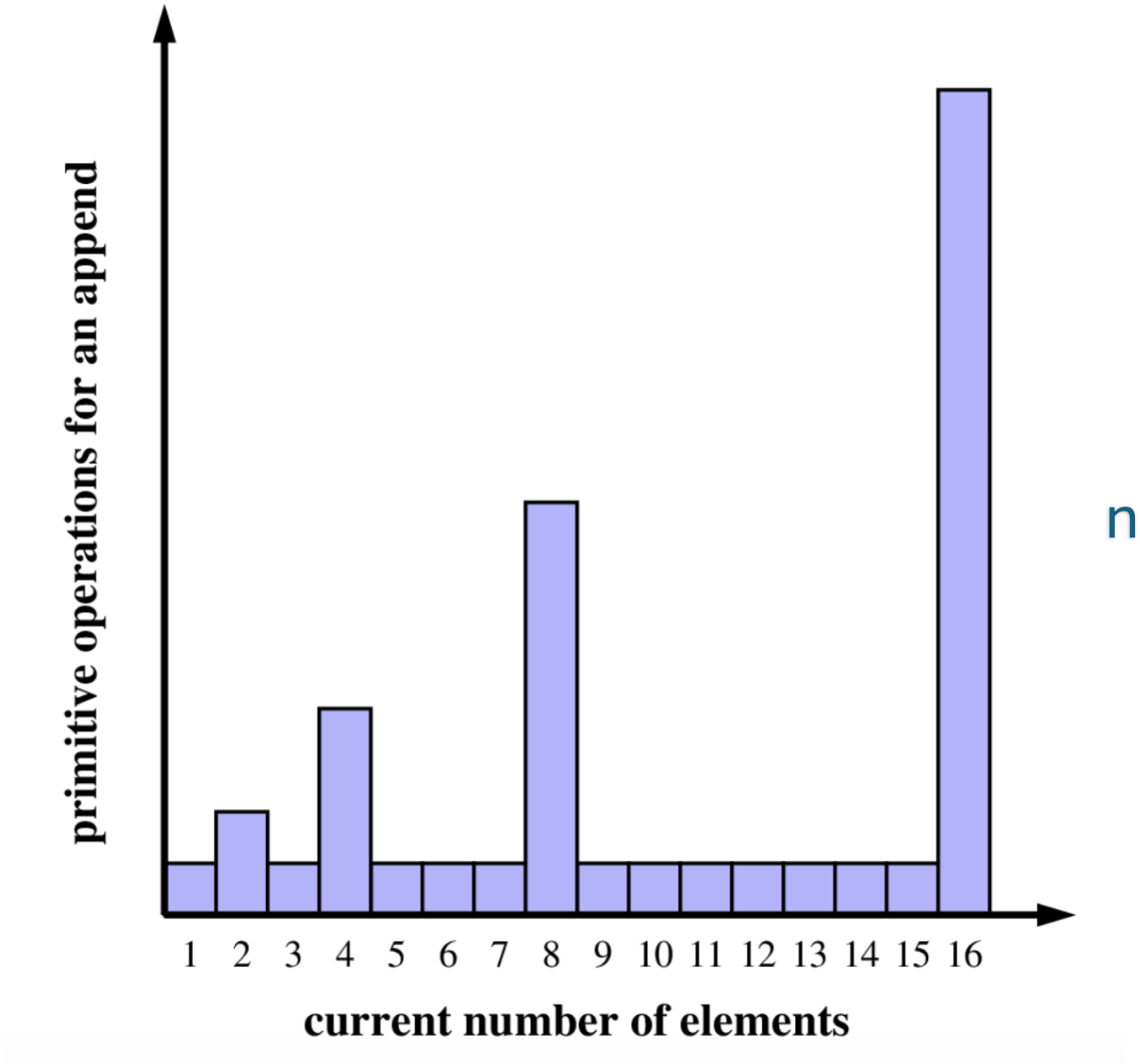
(c) reassigning reference A to the new array

Implementation

```
import ctypes

class DynamicArray:
    """A dynamic array class akin to a simplified Python list."""

    def __init__(self):
        """Create an empty array."""
        self._n = 0  # count actual elements
        self._capacity = 1  # default array capacity
        self._A = self._make_array(self._capacity)  # low-level array

    def __len__(self):
        """Return number of elements stored in the array."""
        return self._n

    def __getitem__(self, k):
        """Return element at index k."""
        if not 0 <= k < self._n:
            raise IndexError('invalid index')
        return self._A[k]  # retrieve from array
```

```
    def append(self, obj):
        """Add object to end of the array."""
        if self._n == self._capacity:  # not enough room
            self._resize(2 * self._capacity)  # so double the capacity
        self._A[self._n] = obj
        self._n += 1

    def _resize(self, c):  # nonpublic utitity
        """Resize internal array to capacity c."""
        B = self._make_array(c)  # new (bigger) array
        for k in range(self._n):  # for each existing value
            B[k] = self._A[k]
        self._A = B  # use the bigger array
        self._capacity = c

    def _make_array(self, c):  # nonpublic utility
        """Return new array with capacity c."""
        return (c * ctypes.py_object)()  # see ctypes documentation
```

Amortized analysis of dynamic arrays



new_capacity = current_capacity * 2

Amortized analysis of dynamic arrays

- Growth Strategy: Doubling the array capacity

  The amortized running time of each append operation is O(1); hence, the total running time of n append operations is O(n)

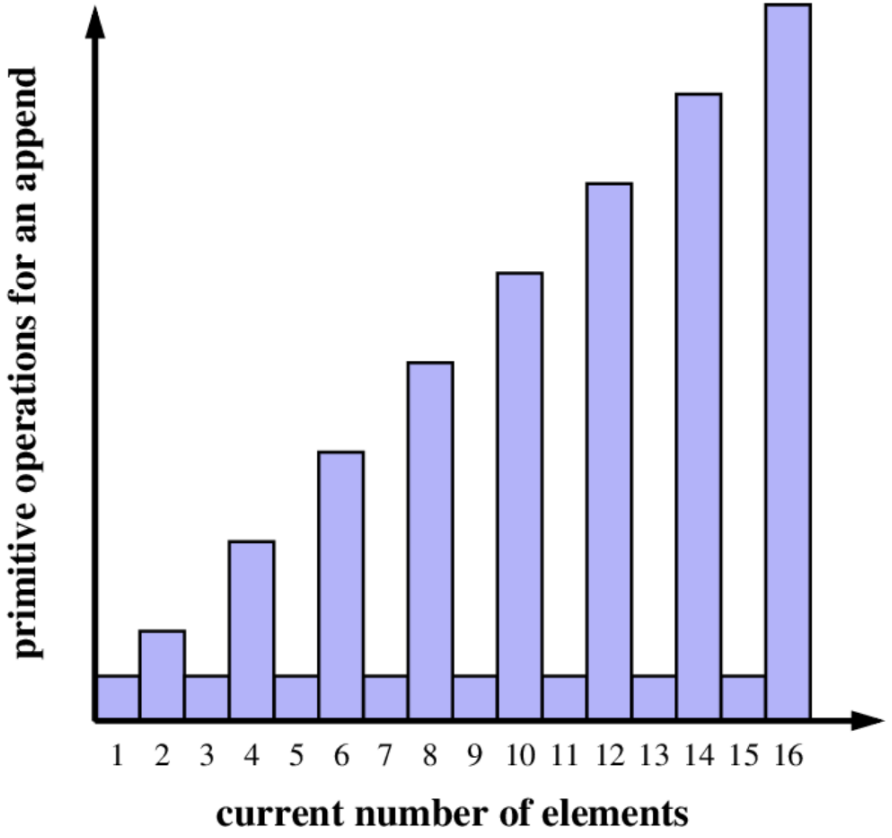
- Growth Strategy: Geometric increase in capacity (E.g. 1.25, 1.75, 2.5 etc.)

  The amortized running time of each append operation is still O(1); hence, the total running time of n append operations is O(n) as well.

Amortized analysis of dynamic arrays

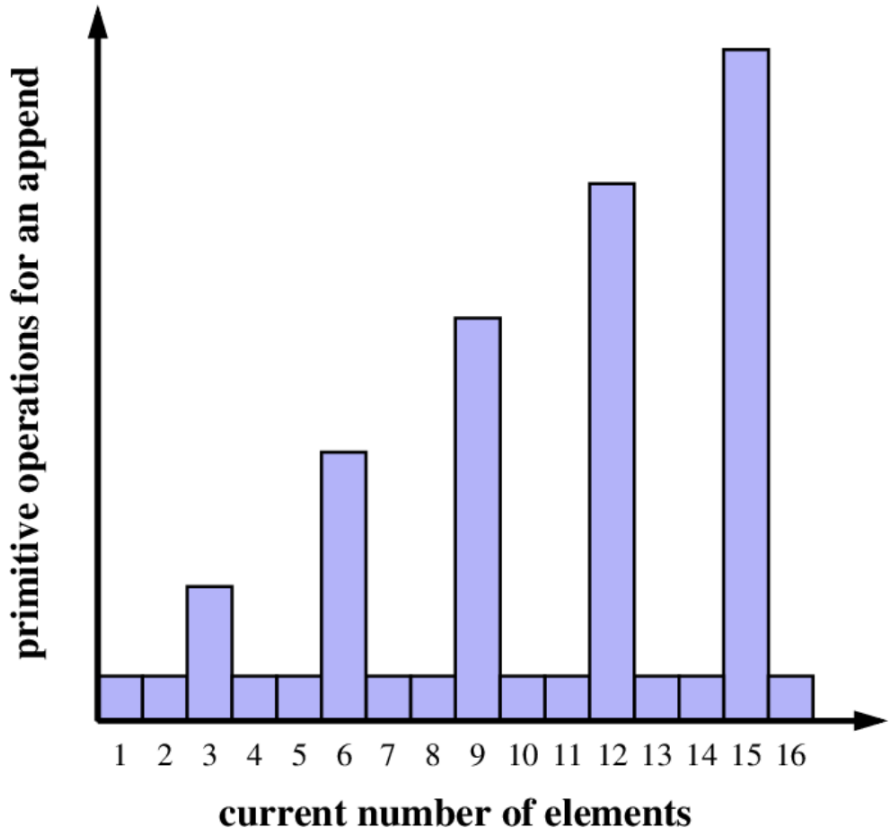
Beware of arithmetic progression

new_capacity = current_capacity + 2



(a)

new_capacity = current_capacity + 3



(b)

Experimentation with Python List's append() method

n	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
μs	0.219	0.158	0.164	0.151	0.147	0.147	0.149

Average running time of append

Suggested Reading:

[ZyBook] 5.3 Dynamic Arrays and Amortization