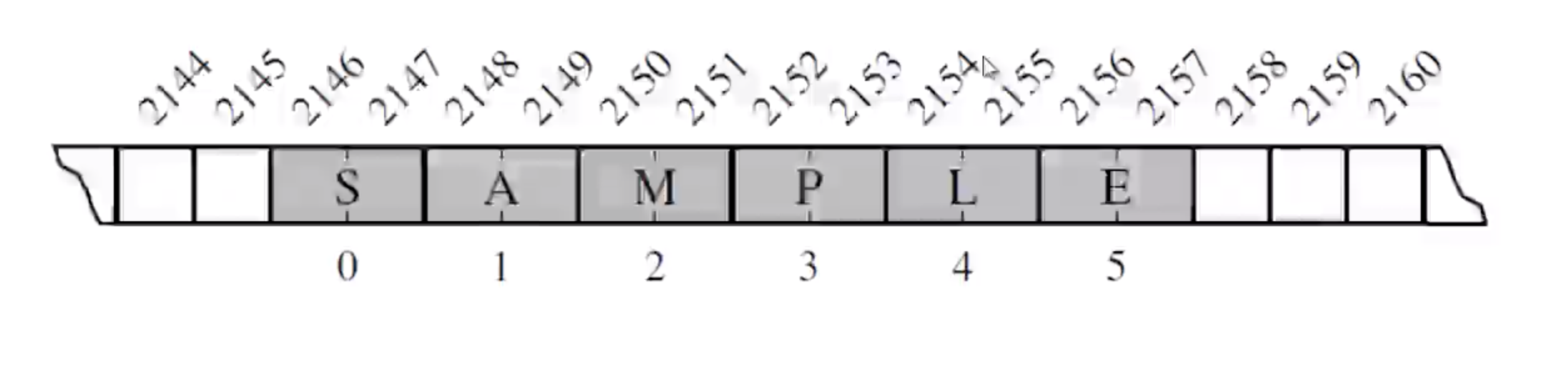
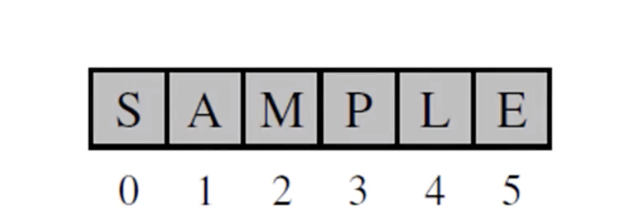
* Computer hardware is designed, in theory, so that any byte of the main memory can be efficiently accessed
* Computer’s main memory performs as a **random access memory (RAM)**
* **Individual byte of memory can be stored or retrieved in O(1) time**
* A text string is stored as an ordered sequence of individual characters
* Python internally represents each Unicode character with 16bits (i.e. 2 bytes)
* Each cell of an array uses the same number of bytes
* Allow any cell to be accessed in constant time
* Appropriate memory address can be computed by: **start + (cellsize)(index)**

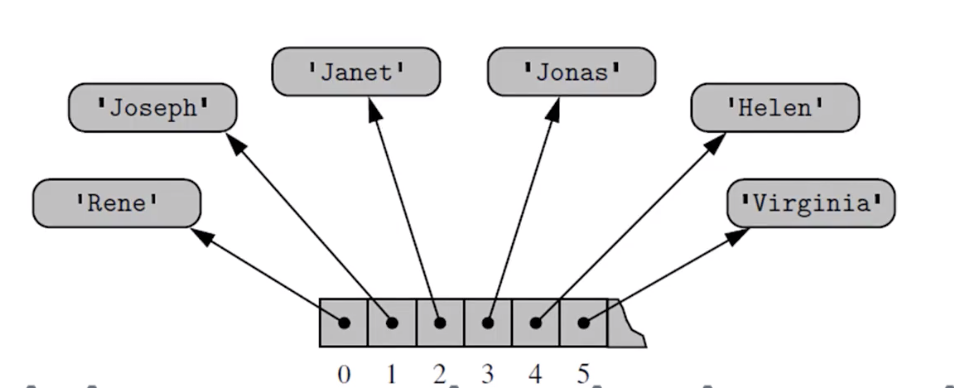


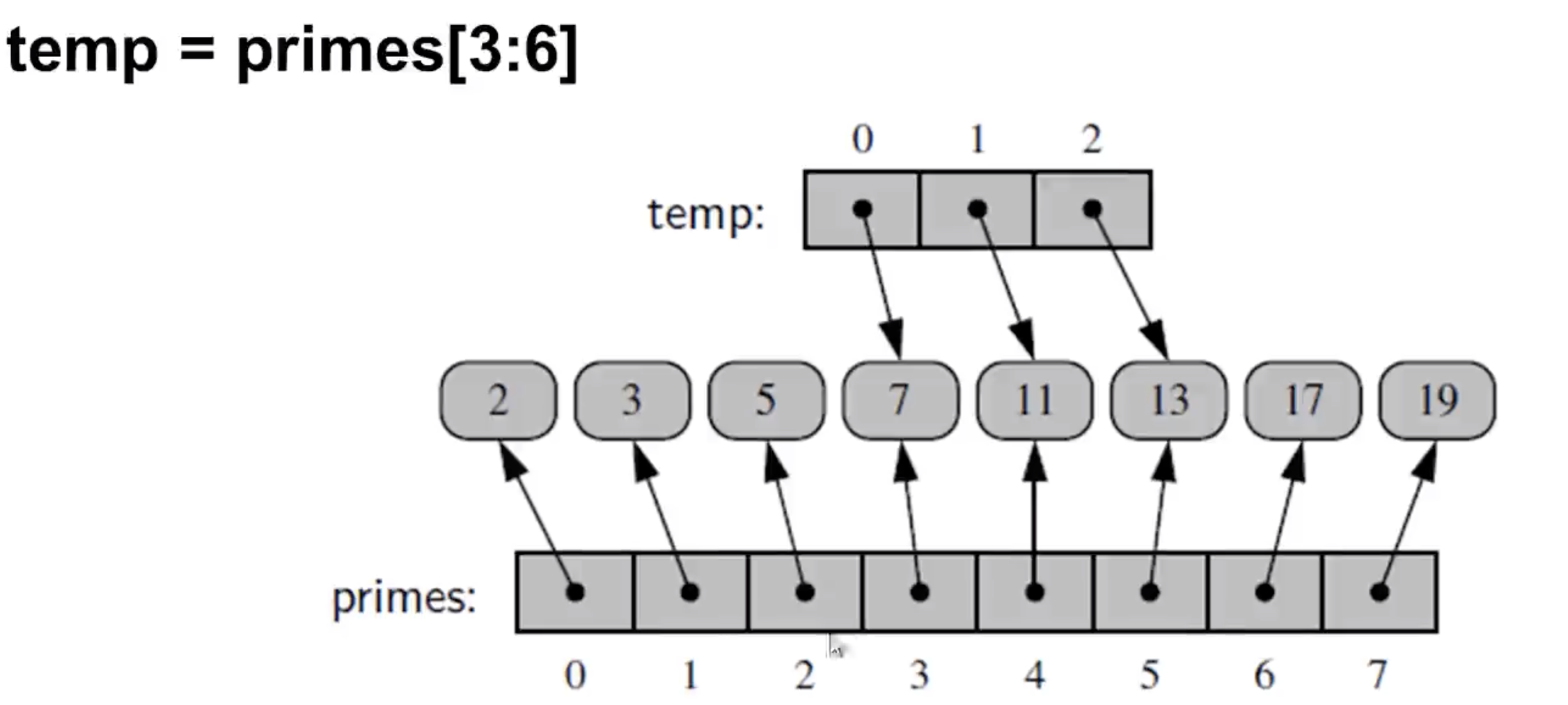
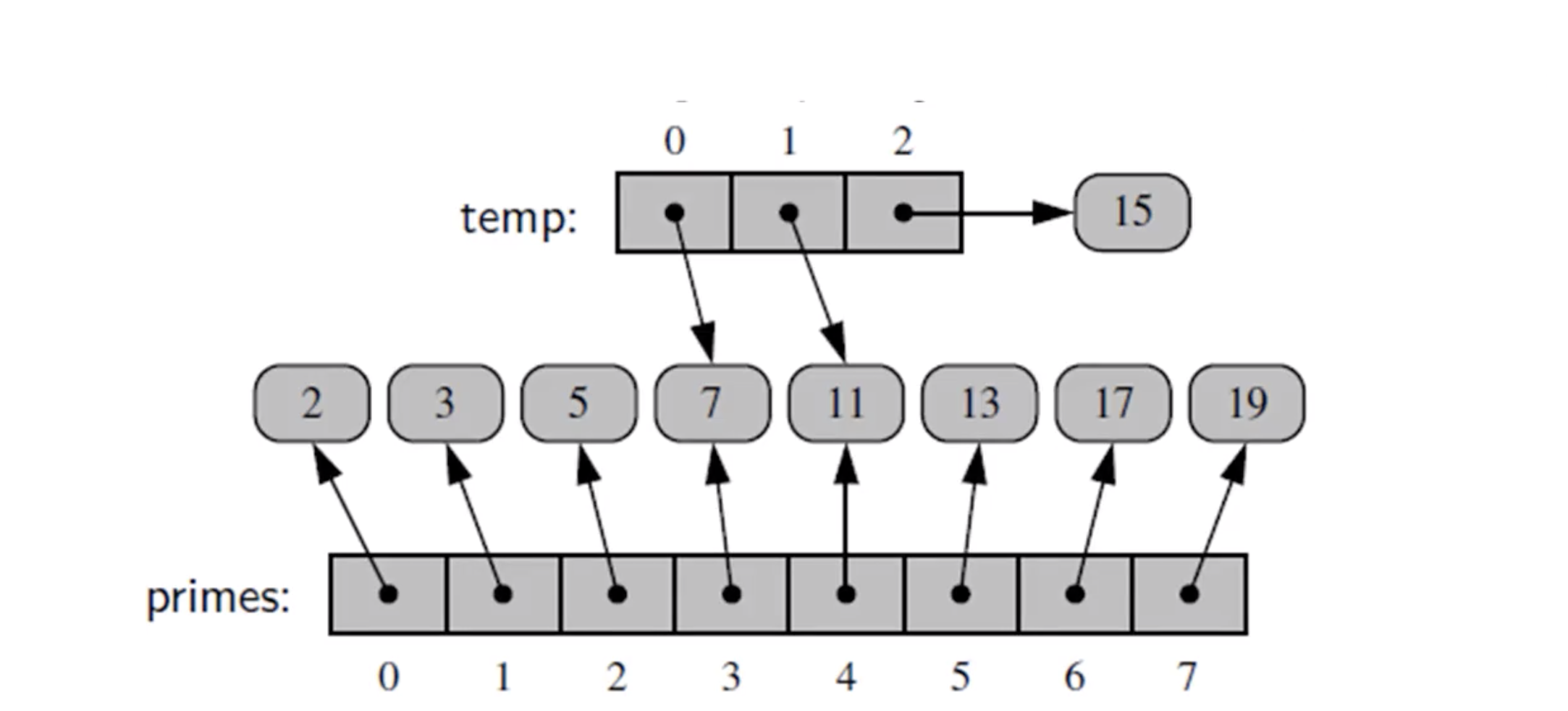
**Higher level abstraction**



**Referential Arrays**

* Imagine 100 student names with ID numbers
* Each cell of the array need to have the same number of bytes
* We can use an array of object references
* **Each element is a reference to the object**



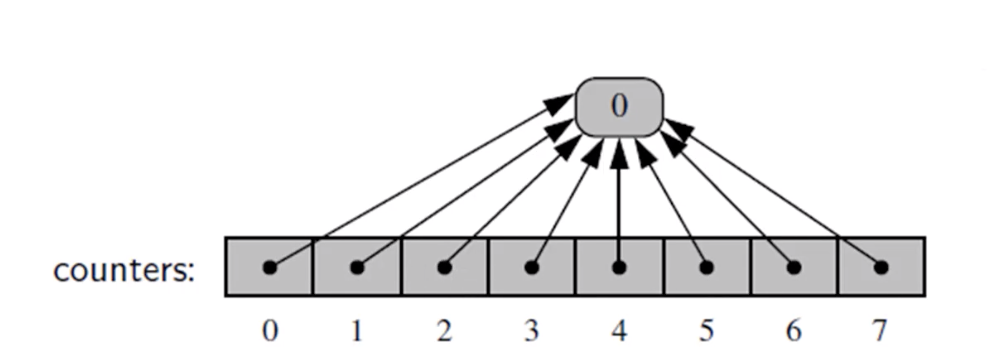
* A single list instance may include multiple references to the same object as elements of the list
* Single object can be an element of two or more lists
* When computing the slice of a list, the result is a new list instance
* New list has references to the same elements that are in the original list
* 
* **List is immutable**. It doesn’t affect the original list when we reassign the temp list. Just point the index to another object
* Ex: temp[2] = 15
* **Copying arrays**
* backup = list(primes)

🡺 This procedures a new list that is a **shallow copy** in that **it references the same elements as in the first list**

* If the contents of the list were of a mutable type**, a deep copy**, meaning a new list with new elements, can be produced by using deepcopy function the copy module
* Ex:

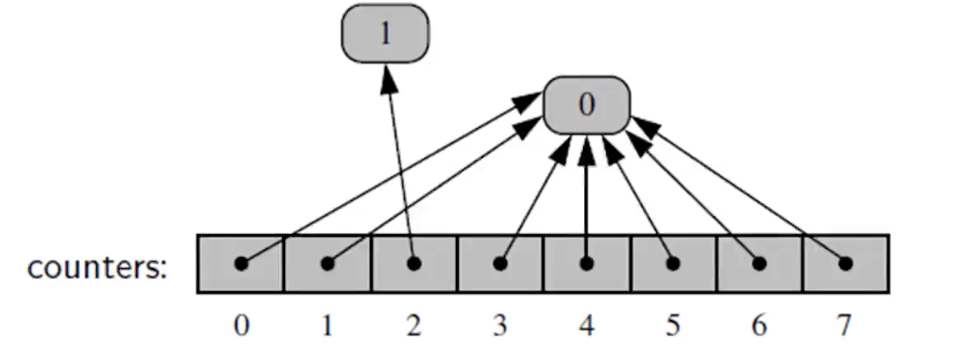
**counters = [0] \* 8**

🡪 All eight cells reference the same object

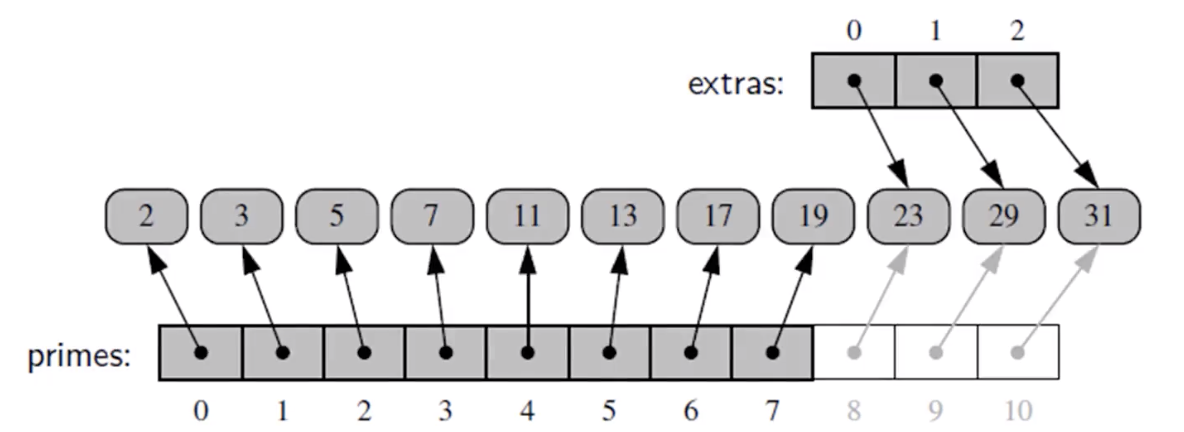


**counters[2] += 1**

🡪 Does not technically change the value of the existing integer instance. This computes a new integer



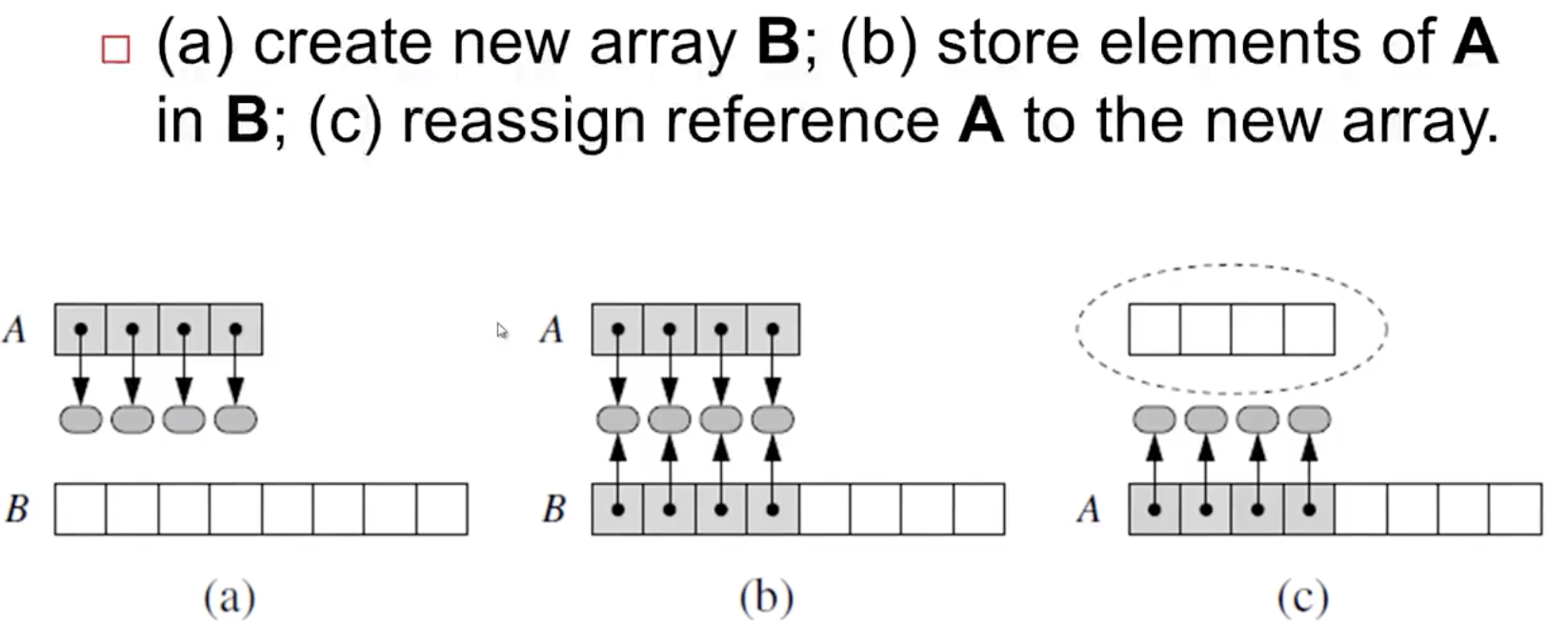
**Extend**

**Primes.extend(extras)** ****

**DYNAMIC ARRAYS**

* The key is to provide means to grow the array A that stores the elements of a list
* We can’t actually grow that array, its capacity is fixed
* If an element is appended to a list at a time when the underlying array is full, we’ll need to perform the following steps:

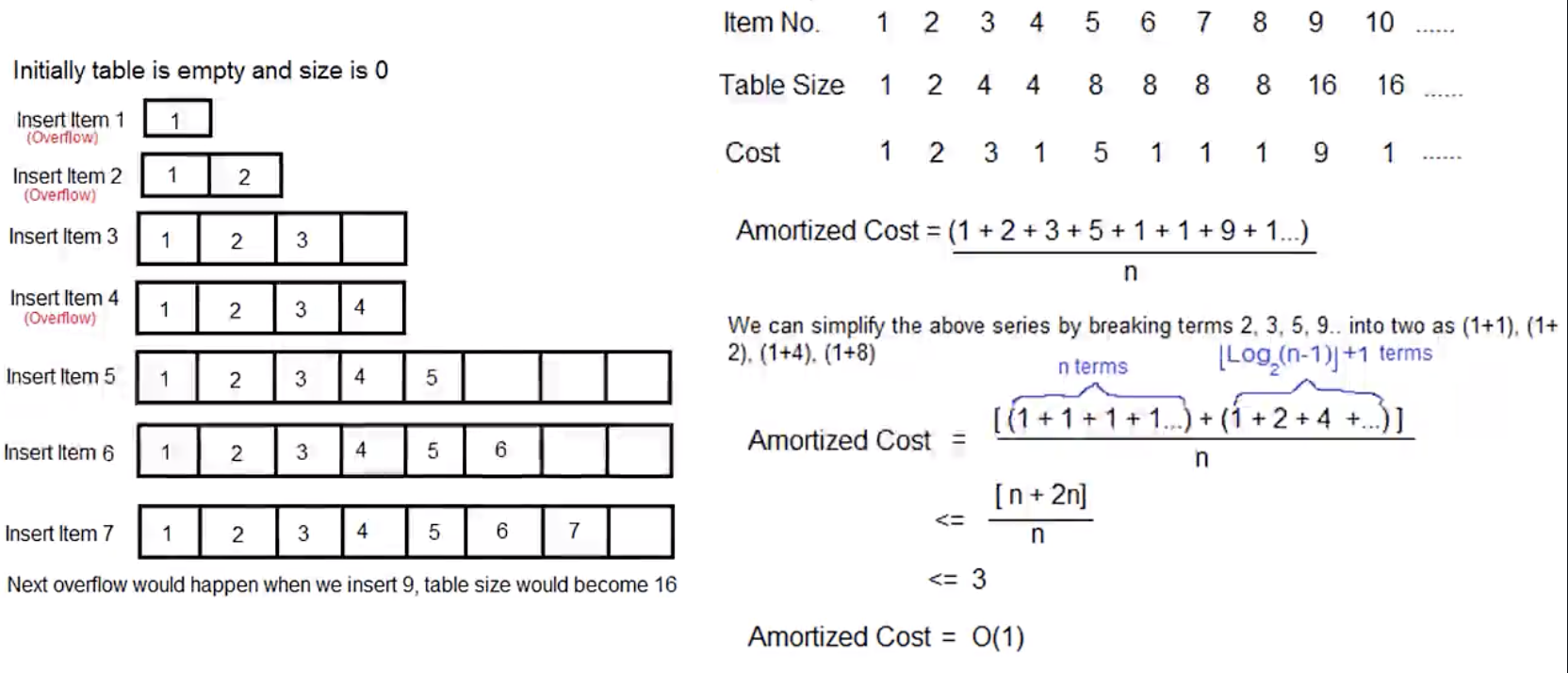
1. Allocate a new array B with larger capacity
2. Set B[i] = A[i], for i = 0 -> i = n-1 ; n: current number of items
3. Set A = B 🡺 use B as the array supporting the list
4. Insert the new element in the new array

****

**🡪 How large of a new array to create?: Common rule is the new array has twice as the capacity of the existing array that has been filled**

**AMORTIZED ANALYSIS**

**Initially table is empty and size = 0**

****