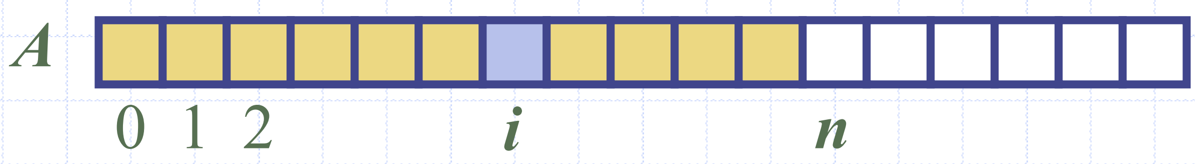
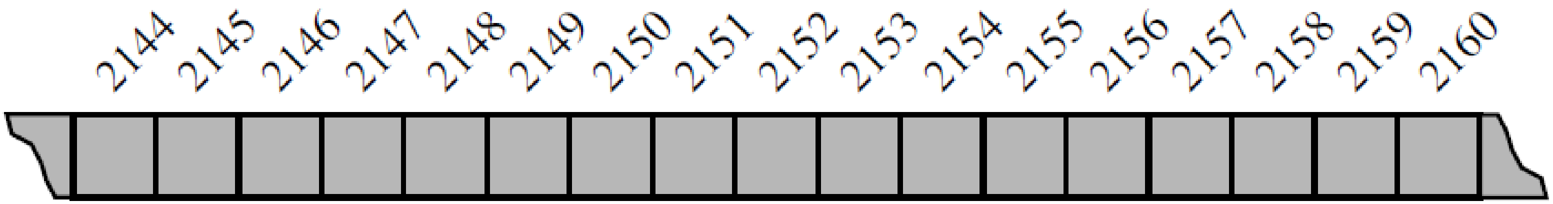
Array-Based Sequences

Python Sequence Classes

- Python has built-in types, **list**, **tuple**, and **str**.
- Each of these **sequence** types supports indexing to access an individual element of a sequence, using a syntax such as A[i]
- Each of these types uses an **array** to represent the sequence.
  - An array is a set of memory locations that can be addressed using consecutive indices, which, in Python, start with index 0.



Low-Level Arrays



A representation of a portion of a computer's memory, with individual bytes labeled with consecutive memory addresses

Random Access Memory (RAM): Theoretically we need O(1) time to access any location if we already know the Address of the location.

Python String Representation ("SAMPLE")

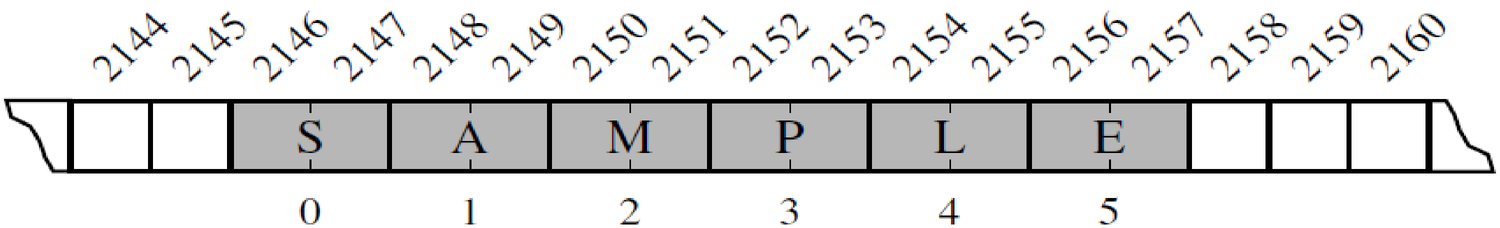


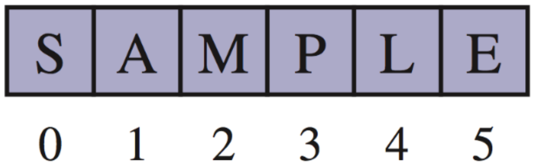
Figure 5.2: A Python string embedded as an array of characters in the computer's memory. We assume that each Unicode character of the string requires two bytes of memory. The numbers below the entries are indices into the string.

- If we know
  - memory address at which array start
  - number of bytes per element
  - index position within the array

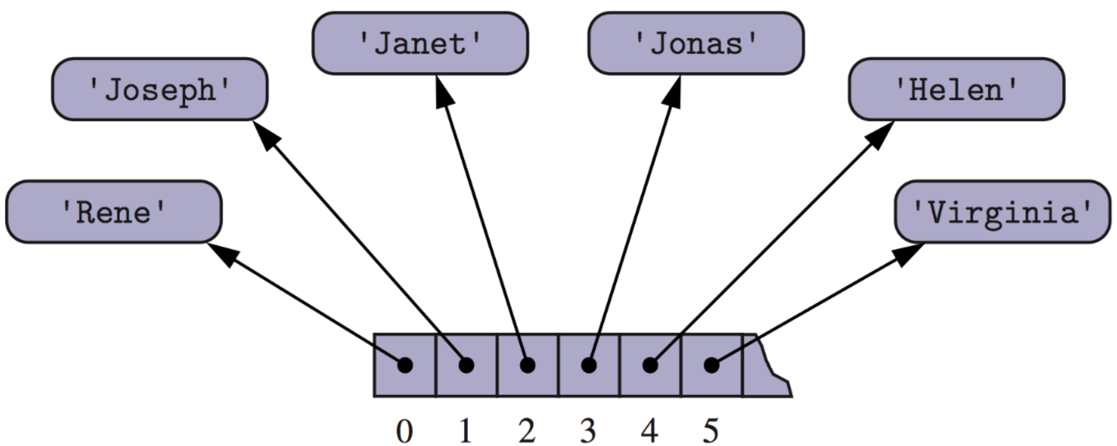
Desired memory address = start + cell_size * index

Arrays of Characters or Object References

- An array can store primitive elements, such as characters, giving us a **compact array.**



- An array can also store references to objects.



List Class Representation in Memory

- Primes=[2,3,5,7,11,13,17,19]
- Temp = Primes[3:6]

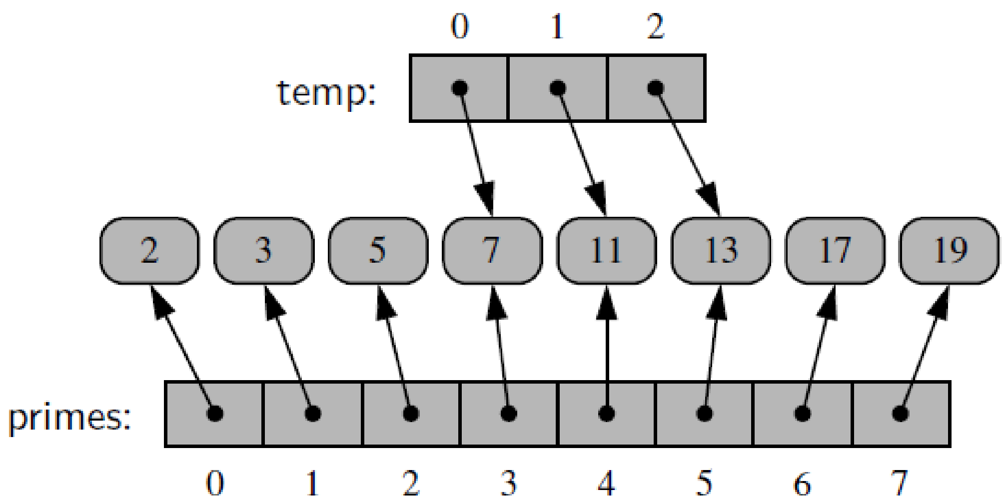
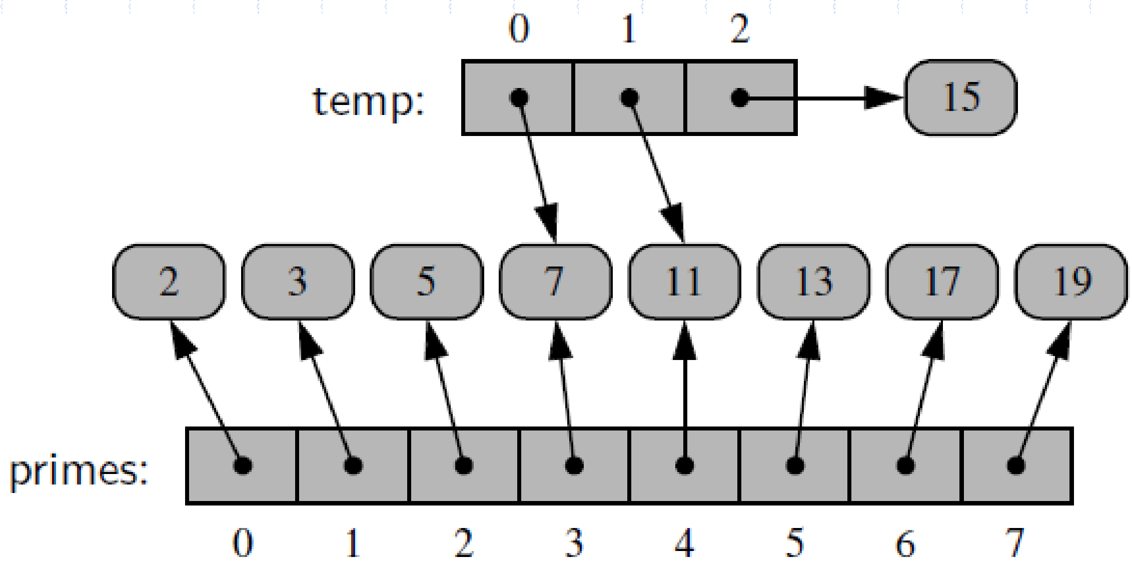


Figure 5.5: The result of the command temp = primes[3:6].

List Class Representation in Memory

- Primes=[2,3,5,7,11,13,17,19]
- Temp = Primes[3:6]
- Temp[2] = 15



counters =[0] * 8

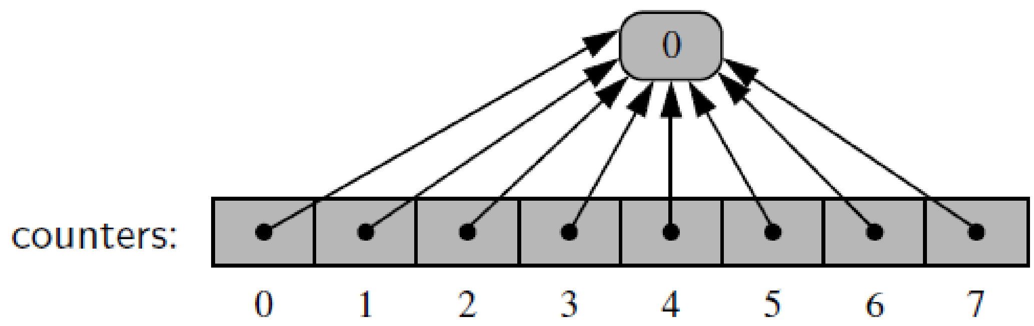
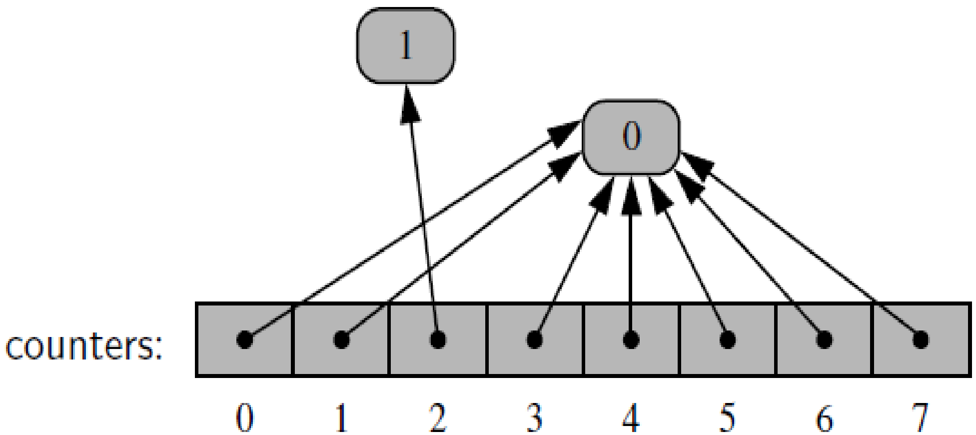


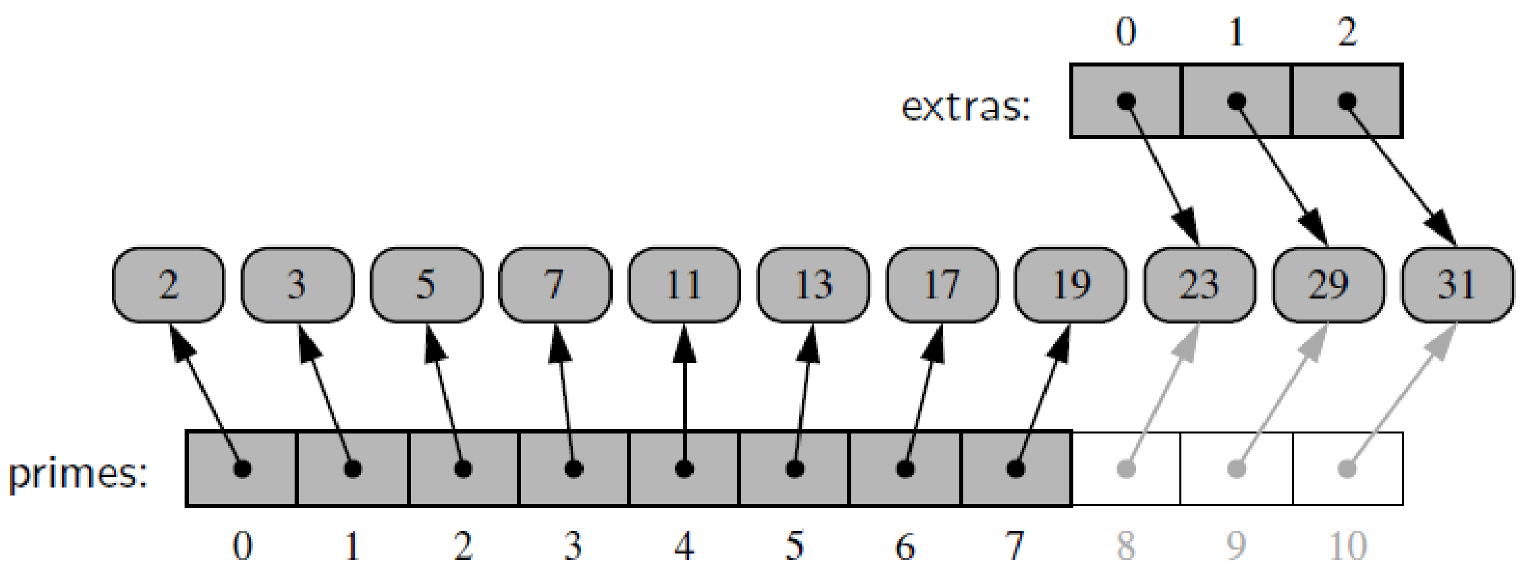
Figure 5.7: The result of the command data = [0] * 8.

After executing counters[2] +=1



primes.extend(extras)

- primes=[2,3,5,7,11,13,17,19]
- extras = [23,29,31]



Shallow Copy, Copy using Slice Operator and Deep Copy of List

```
>>> colours1 = ["red", "green"]
>>> colours2 = colours1
>>> colours2 = ["rouge", "vert"]
>>> print(colours1)
```

Shallow Copy, Copy using Slice Operator and Deep Copy of List

```
>>> colours1 = ["red", "green"]
>>> colours2 = colours1
>>> colours2 = ["rouge", "vert"]
>>> print(colours1)
```

Output: [‘red’, ‘green’]

```
>>> colours1 = ["red", "green"]
>>> colours2 = colours1
>>> colours2[1] = "blue"
>>> print(colours1)
```

```
>>> colours1 = ["red", "green"]
>>> colours2 = colours1
>>> colours2[1] = "blue"
>>> print(colours1)
```

Output: [‘red’, ‘blue’]

Copy with the Slice Operator

```
>>> list1 = ['a','b','c','d']
>>> list2 = list1[:]
>>> list2[1] = 'x'
>>> print(list2)
>>> print(list1)
```

Copy with the Slice Operator

```
>>> lst1 = ['a','b',['ab','ba']]
>>> lst2 = lst1[:]
>>> lst2[0] = 'c'
>>> lst2[2][1] = 'd'
>>> print(lst1)
```

Copy with the Slice Operator

```
>>> lst1 = ['a','b',['ab','ba']]
>>> lst2 = lst1[:]
```

What is the memory image?

Copy with the Slice Operator

```
>>> lst1 = ['a','b',['ab','ba']]
>>> lst2 = lst1[:]
>>> lst2[0] = 'c'
>>> lst2[2][1] = 'd'
>>> print(lst1)
```

What is the memory image?

Using deepcopy from copy module

```
from copy import deepcopy
lst1 = ['a','b',['ab','ba']]
lst2 = deepcopy(lst1)
lst2[2][1] = "d"
lst2[0] = "c"
print(lst2)
print(lst1)
```

What is the memory image?

Compact Arrays

- Primary support for compact arrays is in a module named **array**.
  - That module defines a class, also named array, providing compact storage for arrays of primitive data types.
- The constructor for the array class requires a type code as a first parameter, which is a character that designates the type of data that will be stored in the array.

```
primes = array('i', [2, 3, 5, 7, 11, 13, 17, 19])
```

Type Codes in the array Class

- Python’s array class has the following type codes:

Code	C Data Type	Typical Number of Bytes
'b'	signed char	1
'B'	unsigned char	1
'u'	Unicode char	2 or 4
'h'	signed short int	2
'H'	unsigned short int	2
'i'	signed int	2 or 4
'I'	unsigned int	2 or 4
'l'	signed long int	4
'L'	unsigned long int	4
'f'	float	4
'd'	float	8

What are the outputs?

```
from array import array
primes = array('i', [2,3,5,7,11,13]) # 4 bytes for each signed integer
--NORMAL--
```

```
primes[0] = "hey"
```

```
[15]
primes1 = [2, 3, 5, 7, 11, 13]
primes1[0] = "hey"
print(primes1)
```

Dynamic Array and Amortization

- When creating a low-level array in a computer system, the precise size of that array must be explicitly declared in order for the system to properly allocate a consecutive piece of memory for its storage.
- Otherwise system might dedicate neighboring memory locations to store other data, the capacity of an array cannot trivially be increased by expanding into subsequent cells.
- For Immutable classes like tuple or str instance, this is no problem (since they cannot be resized once instantiated)

Python List Class

- Dynamic Array
  - Underlying array has more capacity than current length of the list.
  - If full, create a new array with larger capacity, and then copy the elements from old array to new array.
  - Old array is no longer needed. System can claim that memory space.

Experimental Analysis (open experiment_list_size.py file)

```
import sys
try:
    n = int(sys.argv[1])
except:
    n = 100
data = []
for k in range(n):                  # NOTE: must fix choice of n
    a = len(data)                   # number of elements
    b = sys.getsizeof(data)         # actual size in bytes
print('Length: {0:3d}; Size in bytes:{1:4d}'.format(a,b))
    data.append(None)               # increase length by one
```

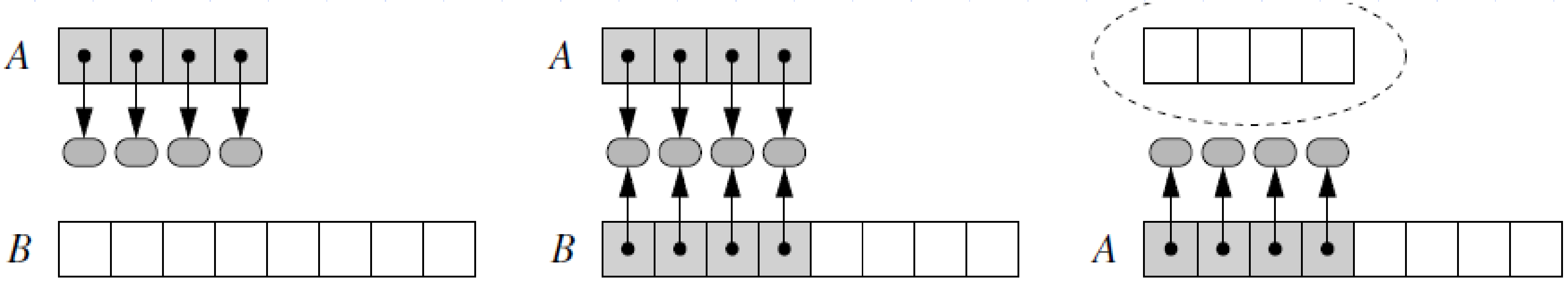
Output

```
Length:   0; Size in bytes:   64
Length:   1; Size in bytes:   96
Length:   2; Size in bytes:   96
Length:   3; Size in bytes:   96
Length:   4; Size in bytes:   96
Length:   5; Size in bytes:  128
Length:   6; Size in bytes:  128
Length:   7; Size in bytes:  128
Length:   8; Size in bytes:  128
Length:   9; Size in bytes:  192
Length:  10; Size in bytes:  192
Length:  11; Size in bytes:  192
Length:  12; Size in bytes:  192
```

```
Length:  13; Size in bytes:  192
Length:  14; Size in bytes:  192
Length:  15; Size in bytes:  192
Length:  16; Size in bytes:  192
Length:  17; Size in bytes:  264
Length:  18; Size in bytes:  264
Length:  19; Size in bytes:  264
Length:  20; Size in bytes:  264
Length:  21; Size in bytes:  264
Length:  22; Size in bytes:  264
Length:  23; Size in bytes:  264
Length:  24; Size in bytes:  264
Length:  25; Size in bytes:  264
Length:  26; Size in bytes:  344
```

Implementation of a Dynamic Array

1. Allocate a new array *B* with larger capacity.
2. Set $B[i] = A[i]$, for $i = 0, \ldots, n-1$, where n denotes current number of items.
3. Set $A = B$, that is, we henceforth use *B* as the array supporting the list.
4. Insert the new element in the new array.



(a) (b) (c)

Python Implementation

```python
import ctypes                                    # provides low-level arrays

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 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 utitity
     """Return new array with capacity c."""
     return (c * ctypes.py_object)( )            # see ctypes documentation
```

Growable Array-based Array List

- In an add(o) operation (without an index), we could always add at the end
- When the array is full, we replace the array with a larger one
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

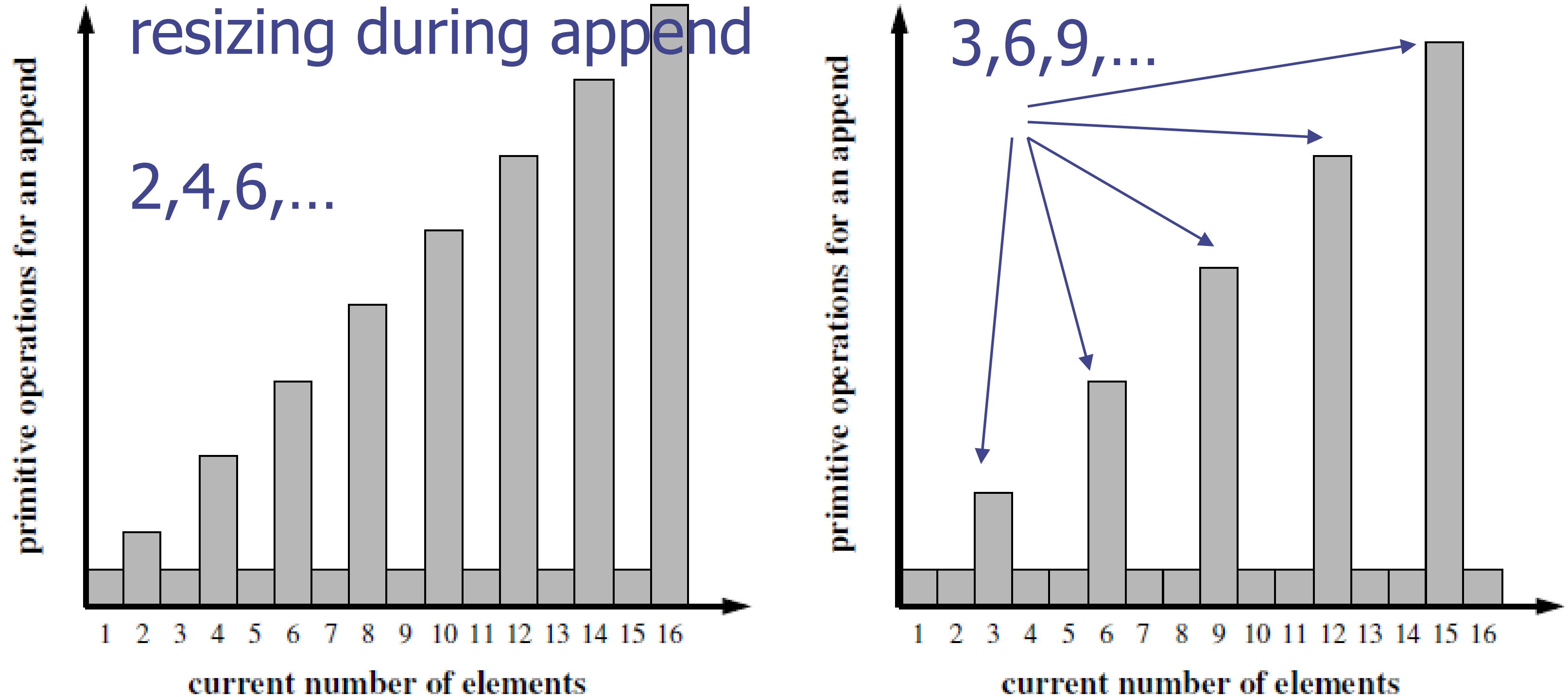
Algorithm $add(o)$
  if $n = S.length - 1$ **then**
    $A \leftarrow$ **new array of size …**
    for $i \leftarrow 0$ **to** $n-1$ **do**
      $A[i] \leftarrow S[i]$
    $S \leftarrow A$
  $n \leftarrow n + 1$
  $S[n-1] \leftarrow o$

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n add(o) operations
- We assume that we start with an empty stack represented by an array of size 1
- We call amortized time of an add operation the average time taken by an add over the series of operations, i.e., $T(n)/n$

Incremental Strategy

When array size reaches current capacity, array needs resizing during append



(a) (b)

Figure 5.15: Running times of a series of append operations on a dynamic array using arithmetic progression of sizes. (a) Assumes increase of 2 in size of the array, while (b) assumes increase of 3.

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
- The total time $T(n)$ of a series of n add operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$
$$n + c(1 + 2 + 3 + \ldots + k) =$$
$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an add operation is $O(n)$

Always Doubling Strategy

- When array size reaches current capacity, array needs resizing during append

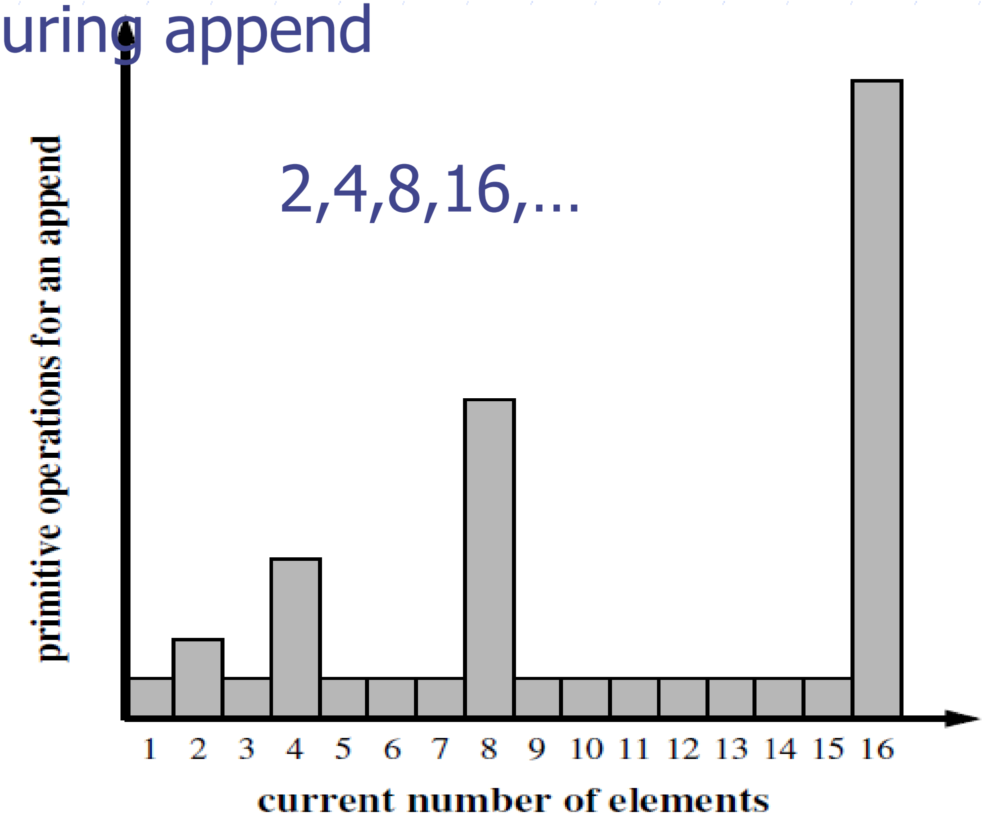


Figure 5.13: Running times of a series of append operations on a dynamic array.

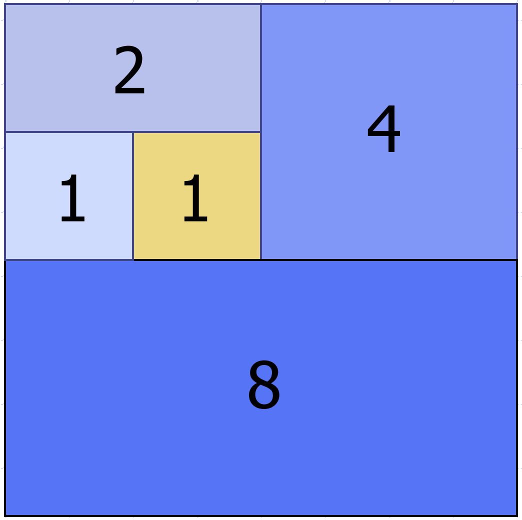
Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n add operations is proportional to

$$n + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 =$$
$$3n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of an add operation is $O(1)$

geometric series



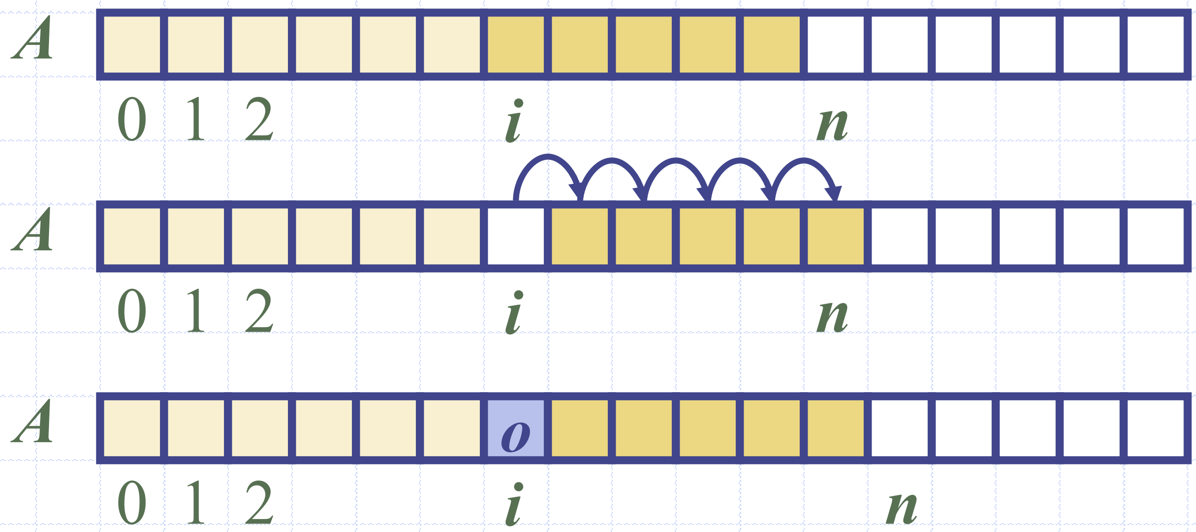
Average Running Time Append Operation over n call

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

Table 5.2: Average running time of append, measured in microseconds, as observed over a sequence of *n* calls, starting with an empty list.

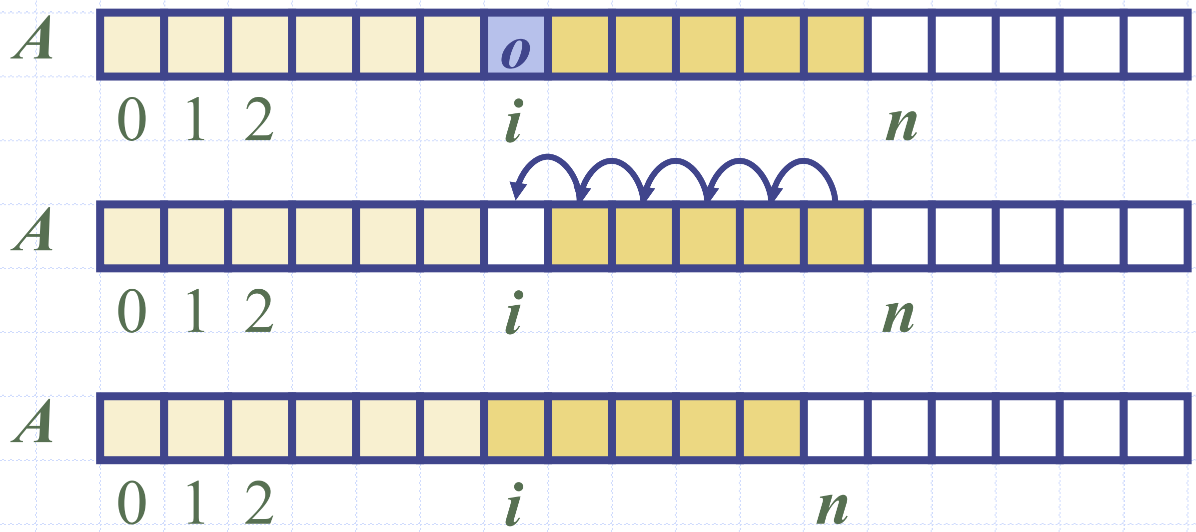
Insertion

- In an operation $add(i, o)$, we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Element Removal

- In an operation ***remove***(i), we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \dots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Performance

- In an array based implementation of a dynamic list:
  - The space used by the data structure is $O(n)$
  - Indexing the element at i takes $O(1)$ time
  - ***add*** and ***remove*** run in $O(n)$ time in worst case
- In an ***add*** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one…

Asymptotic Performance of Nonmutating Behaviors

Operation	Running Time
len(data)	$O(1)$
data[j]	$O(1)$
data.count(value)	$O(n)$
data.index(value)	$O(k+1)$
value **in** data	$O(k+1)$
data1 == data2 (similarly !=, <, <=, >, >=)	$O(k+1)$
data[j:k]	$O(k-j+1)$
data1 + data2	$O(n_1+n_2)$
c * data	$O(cn)$

Asymptotic Performance of mutating behavior

Operation	Running Time
data[j] = val	$O(1)$
data.append(value)	$O(1)^*$
data.insert(k, value)	$O(n-k+1)^*$
data.pop()	$O(1)^*$
data.pop(k) **del** data[k]	$O(n-k)^*$
data.remove(value)	$O(n)^*$
data1.extend(data2) data1 += data2	$O(n_2)^*$
data.reverse()	$O(n)$
data.sort()	$O(n \log n)$

*amortized

Composing Large Strings

```
letters = ''
for c in document:
  if c.isalpha( ):
    letters += c
```

```
temp = [ ]
for c in document:
  if c.isalpha( ):
    temp.append(c)
letters = ''.join(temp)
```

Creating a Multidimensional list

22	18	709	5	33
45	32	830	120	750
4	880	45	66	61

data = [[22, 18, 709, 5, 33], [45, 32, 830, 120, 750], [4, 880, 45, 66, 61]]

data[1][3] represents the value at row number 1 and column number 3.

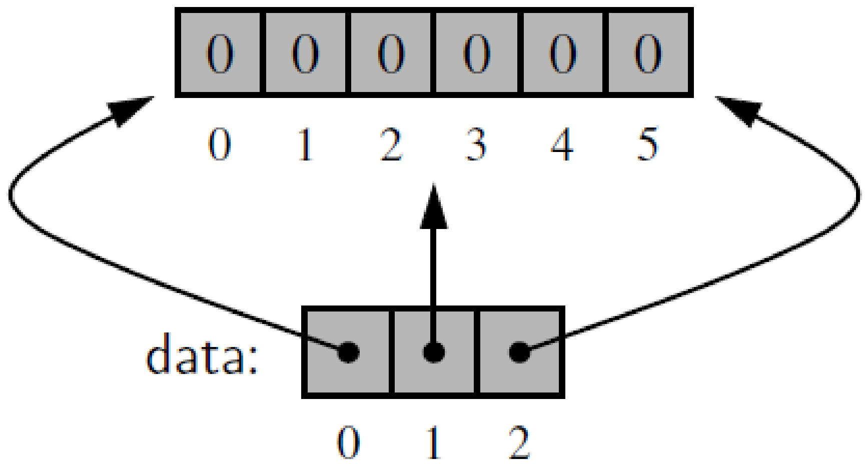
Creating a Two Dimensional List

- Two dimensional list of integers with r rows and c columns with initialize all values to 0.
- data = ([0] * c) * r

  Flawed Approach. Why???
- data = [[0]*c]*r

  Flawed Approach too. Why???

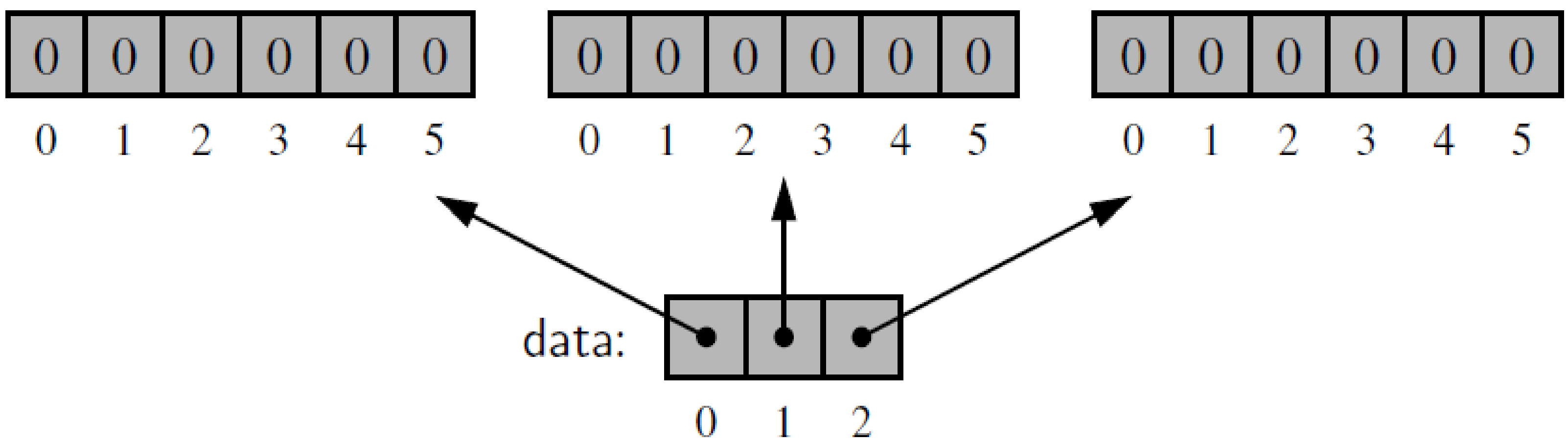
Creating a Two Dimensional List



A flawed representation of a 3×6 data set as a list of lists, created with the command data = [[0]*6]*3.
(For simplicity, we overlook the fact that the values in the secondary list are referential.)

Creating a Two Dimensional List

```
data = [ [0]*c for j in range(r) ]
```



A valid representation of a 3×6 data set as a list of lists.

In-class exercise

- Creating our own dynamic array.
- Open the UserDefinedDynamicArray.py file.
- Implement is_empty, __iter__, & __setitem__ functions
- Submit your code to Gradescope

Using Array Based Sequences

- Storing High Scores of a game
- Simple Cryptography
- Multidimensional Array
- Tic Tac Toe game