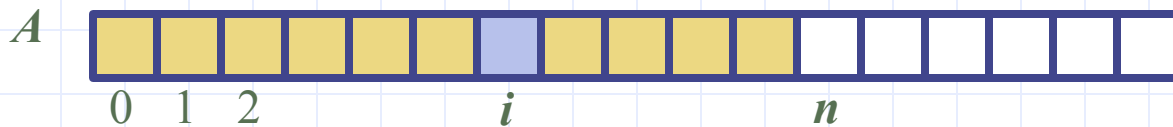


Array-Based Sequences



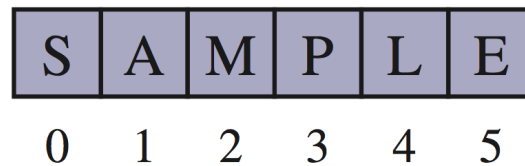
Python Sequence Classes

- ❑ 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.

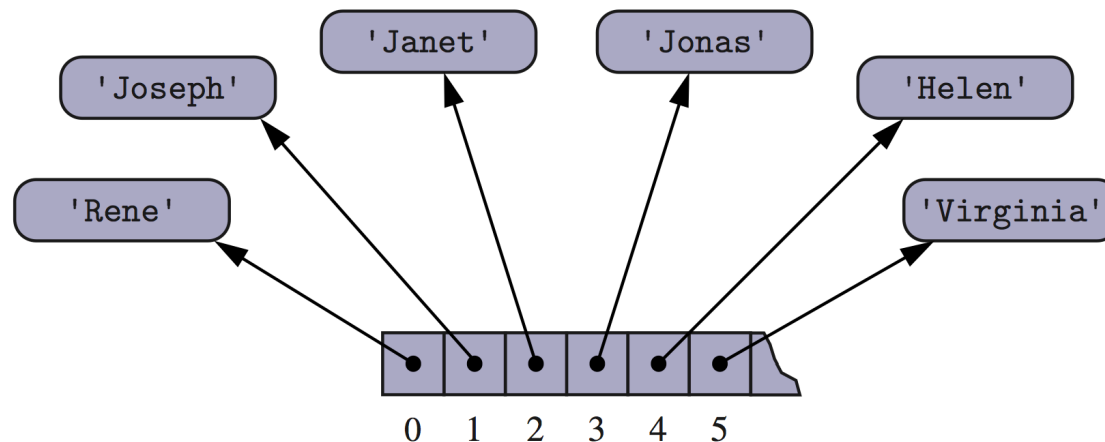


Arrays of Characters or Object References

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



- An array can also store references to objects.



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])
```

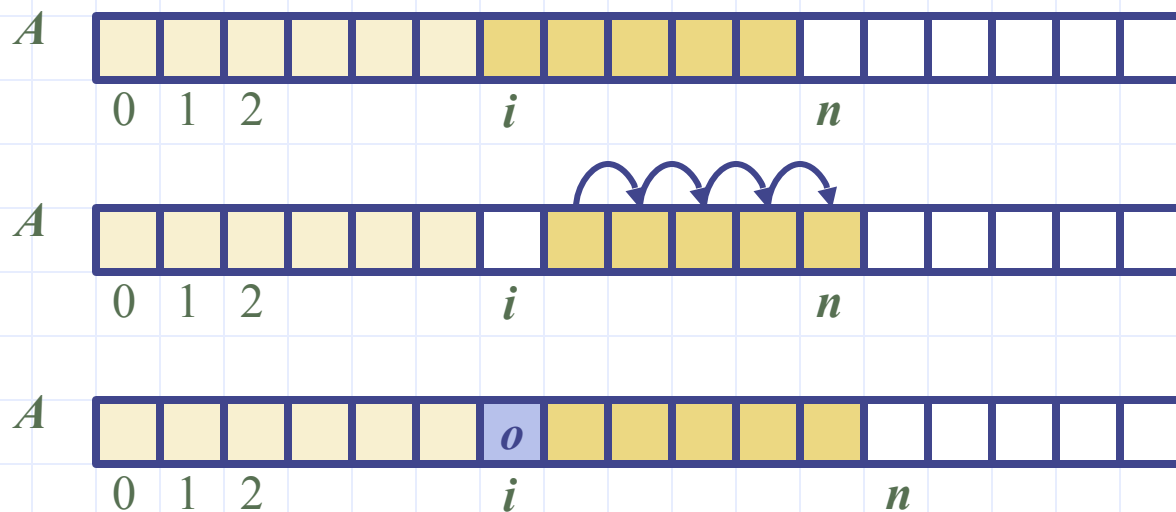
Type Codes in the array Class

- 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

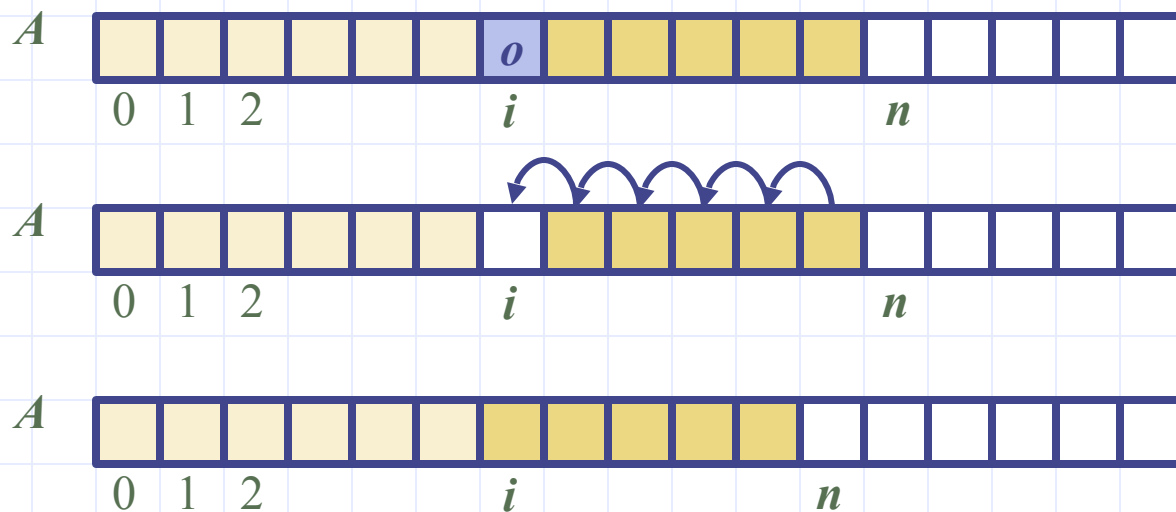
Insertion

- In an operation *add*(i, o), we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \dots, A[n - 1]$
- 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...

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  $t = S.length - 1$  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 $\text{add}(o)$ 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 Analysis

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

$$\begin{aligned} n + c + 2c + 3c + 4c + \dots + kc &= \\ n + c(1 + 2 + 3 + \dots + k) &= \\ n + ck(k + 1)/2 \end{aligned}$$

- 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 $O(n)$

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

$$\begin{aligned} n + 1 + 2 + 4 + 8 + \dots + 2^k &= \\ n + 2^{k+1} - 1 &= \\ 3n - 1 \end{aligned}$$

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

geometric series

