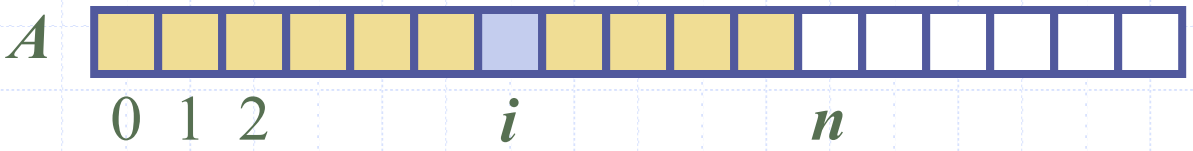
Array-Based Sequences

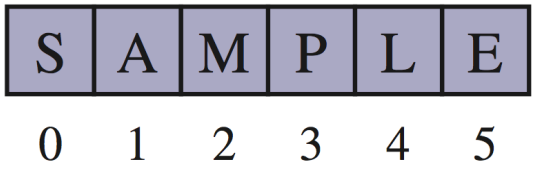
Python Sequence Classes

- 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.

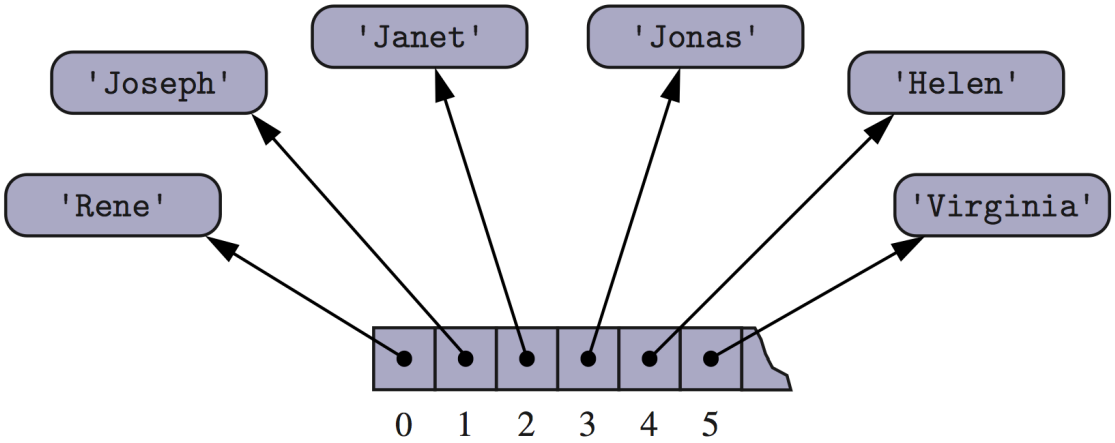


Arrays of Characters or Object References

- 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])
```

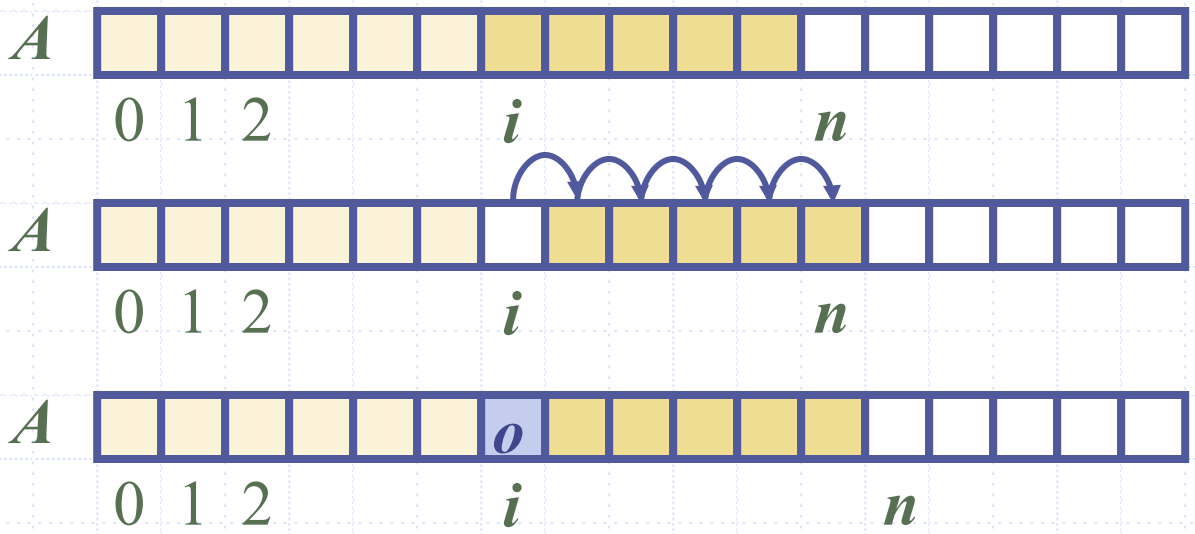
Type Codes in the array Class

- 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

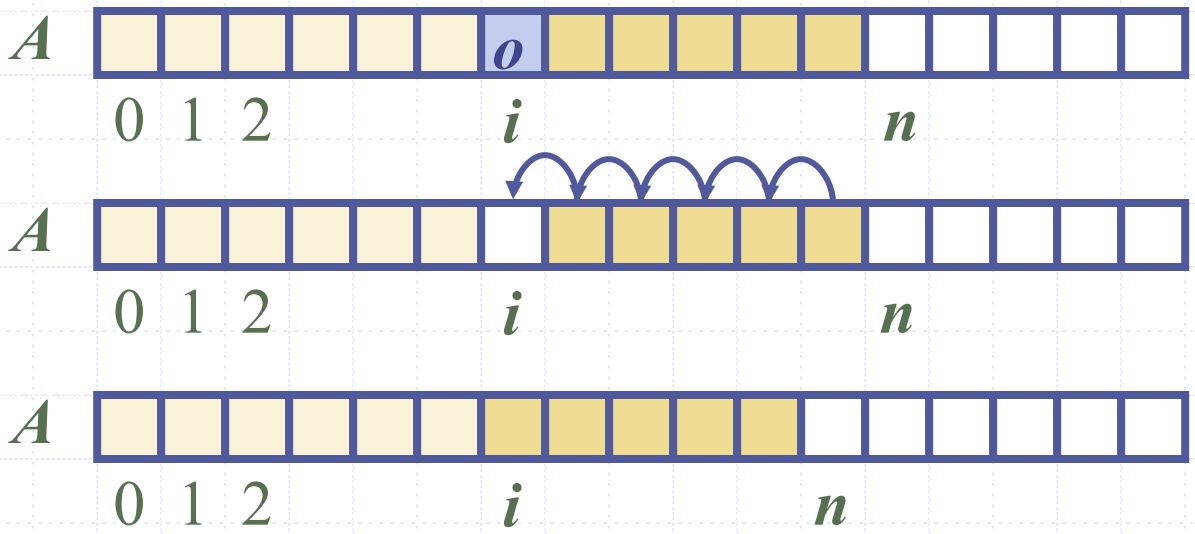
Insertion

- In an operation $add(i, o)$, we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n-1]$
- 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 $A[i+1], \ldots, A[n-1]$
- 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 n = S.length then
    A = new array of
          size …
    for i = 0 to n−1 do
      A[i] = S[i]
    S = A
  S[n] = o
  n = n + 1
```

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 list 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

$$n + c + 2c + 3c + 4c + \ldots + kc =$$

$$n + c(1 + 2 + 3 + \ldots + k) =$$

$$n + ck(k + 1)/2$$

- 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 $O(n)$

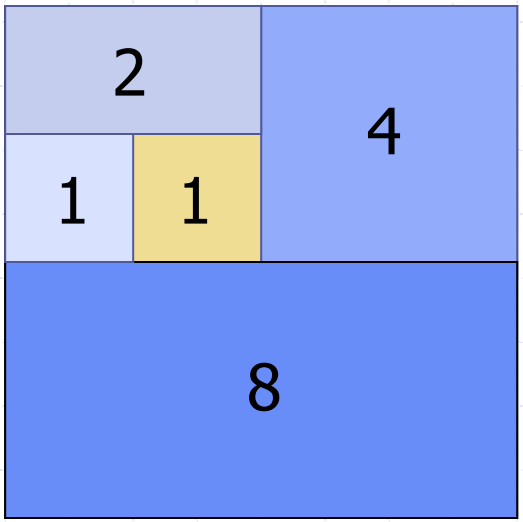
Doubling Strategy Analysis

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

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

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

geometric series



Python Implementation…

```
#Python's module to create arrays
import ctypes

class DynamicArray:

    ...
```

_makeArray & __init__

```
#return a pointer to a memory area
#that can store c contiguous python objects
def _makeArray( self, c ):
    return( c * ctypes.py_object )()

#create an array of 1 element
def __init__( self ):
    self._n = 0
    self._capacity = 1
    self._A = self._makeArray( self._capacity )
```

__str__

```
#pretty print the array
def __str__( self ):
    if self._n == 0:
        return "[](size = 0; capacity = " + str( self._capacity ) + ")"
    pp = "[" + str( self._A[0] )
    for k in range( 1, self._n ):
        pp += ", " + str( self._A[k] )
        pp += "](size = " + str( self._n )
        pp += "; capacity = " + str( self._capacity ) + ")"
    return pp
```

__len__ & __getitem__

```
#returns the number of elements in the array
def __len__( self ):
    return self._n

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

append

```
#append at the end of the list
def append( self, obj ):
    #if there is no space in the array
    if self._n == self._capacity:
        #double its size
        self._resize( 2 * self._capacity )
        #put obj at the end of the array
    self._A[self._n] = obj
    #increment n by 1
    self._n += 1
```

resize

```
#resize extends the array to c
def _resize( self, c ):
    #create the new array for c elements
    B = self._makeArray( c )
    #copy the elements of the old array in the new array
    for k in range( self._n ):
        B[k] = self._A[k]
    #make the self array point to the new array
    self._A = B
    #adjust the capacity of the new array
    self._capacity = c
```

remove

```
#remove the ith element of the list
def remove( self, i ):
    #array indices starts at 0
    i -= 1
    #if index not within bounds
    if i < 0 or i >= self._n:
        raise IndexError( 'index out of bound' )
    #fill the hole left by the removed element
    for k in range( i + 1, self._n ):
        self._A[k-1] = self._A[k]
    #decrement n by 1
    self._n -= 1
```

find

```
#find obj in the list and return its rank
#or return False otherwise
def find( self, obj ):
    #iterate until obj is not found
    for k in range( self._n ):
        if self._A[k] == obj:
        #if found return its rank
        #array indices starts at 0
            return k+1
    #obj was not found so return False
    return False
```

Unit testing

```
if __name__ == '__main__':

    data = DynamicArray()
    print( data )

    data.append( 'titi' )
    data.append( 'toto' )
    data.append( 'tata' )
    print( data )

    idx = data.find( 'titi' )
    if idx:
        print( "found titi ranked", idx )
    else:
          print( "titi not found"        )
    idx = data.find( 'cece' )
    if idx:
        print( "found cece ranked", idx )
    else:
        print( "cece not found" )

    data.remove( 1 )
    print( data )
    data.remove( 2 )
    print( data )
    data.remove( 1 )
    print( data )
    try:
        data.remove( 1 )
    except IndexError:
          print( "No element to remove" )
```

```
[](size = 0; capacity = 1)
[titi, toto, tata](size = 3; capacity = 4)
found titi ranked 1
cece not found
[toto, tata](size = 2; capacity = 4)
[toto](size = 1; capacity = 4)
[](size = 0; capacity = 4)
No element to remove
```

A first implementation of List using ArrayList

```
from DynamicArray import DynamicArray

class ArrayList:

    #implements the ADT List (List.py)
    #uses the DynamicArray class (DynamicArray.py)
    def __init__( self ):
        self._A = DynamicArray()
```

List methods using a DynamicArray

```
def __len__( self ):
    return len( self._A )

def __str__( self ):
    return str( self._A )

def __getitem__( self, k ):
    return self._A[k]

#append at the end of the list
def append( self, obj ):
    self._A.append( obj )

#remove the ith element of the list
def remove( self, i ):
    self._A.remove( i )

#return the rank of obj in the list
def find( self, obj ):
    return self._A.find( obj )
```

Unit testing

```
if __name__ == '__main__':

    data = ArrayList()
    print( data )

    data.append( 'titi' )
    data.append( 'toto' )
    data.append( 'tata' )
    print( data )

    idx = data.find( 'titi' )
    if idx:
        print( "found titi ranked", idx )
    else:
          print( "titi not found"        )
    idx = data.find( 'cece' )
    if idx:
        print( "found cece ranked", idx )
    else:
        print( "cece not found" )

    data.remove( 1 )
    print( data )
    data.remove( 2 )
    print( data )
    data.remove( 1 )
    print( data )
    try:
        data.remove( 1 )
    except IndexError:
          print( "No element to remove" )
```

```
[](size = 0; capacity = 1)
[titi, toto, tata](size = 3; capacity = 4)
found titi ranked 1
cece not found
[toto, tata](size = 2; capacity = 4)
[toto](size = 1; capacity = 4)
[](size = 0; capacity = 4)
No element to remove
```

A second implementation of List using a Python's list…

```
class ListList:

    #implements the ADT List (List.py)
    #uses the python default List
    def __init__( self ):
        self._A = []

    def __len__( self ):
        return len( self._A )

    #no access to a python's list capacity
    def __str__( self ):
        pp = str( self._A )
        pp += "(size = " + str( len( self._A ) ) + ")"
        return pp

    def __getitem__( self, k ):
        return self._A[k]
```

A second implementation of List using a Python's list

```
#append at the end of the list
def append( self, obj ):
    self._A.append( obj )

#remove the ith element of the list
def remove( self, i ):
    #indices in a python's list starts at 0
    self._A.pop( i-1 )

#return the rank of obj in the list
#or False otherwise
def find( self, obj ):
    try:
        idx = self._A.index( obj )
    except ValueError:
        return False
    #obj is in the list
    #indices in a python's list starts at 0
    return 1 + self._A.index( obj )
```

Unit testing

```
if __name__ == '__main__':

    data = ListList()
    print( data )

    data.append( 'titi' )
    data.append( 'toto' )
    data.append( 'tata' )
    print( data )

    idx = data.find( 'titi' )
    if idx:
        print( "found titi ranked", idx )
    else:
          print( "titi not found"        )
    idx = data.find( 'cece' )
    if idx:
        print( "found cece ranked", idx )
    else:
        print( "cece not found" )

    data.remove( 1 )
    print( data )
    data.remove( 2 )
    print( data )
    data.remove( 1 )
    print( data )
    try:
        data.remove( 1 )
    except IndexError:
          print( "No element to remove" )
```

```
[](size = 0)
[titi, toto, tata](size = 3)
found titi ranked 1
cece not found
[toto, tata](size = 2)
[toto](size = 1)
[](size = 0)
No element to remove
```