20 Jan 2020

Array and Python List

CPE111-Programming with Data Structures

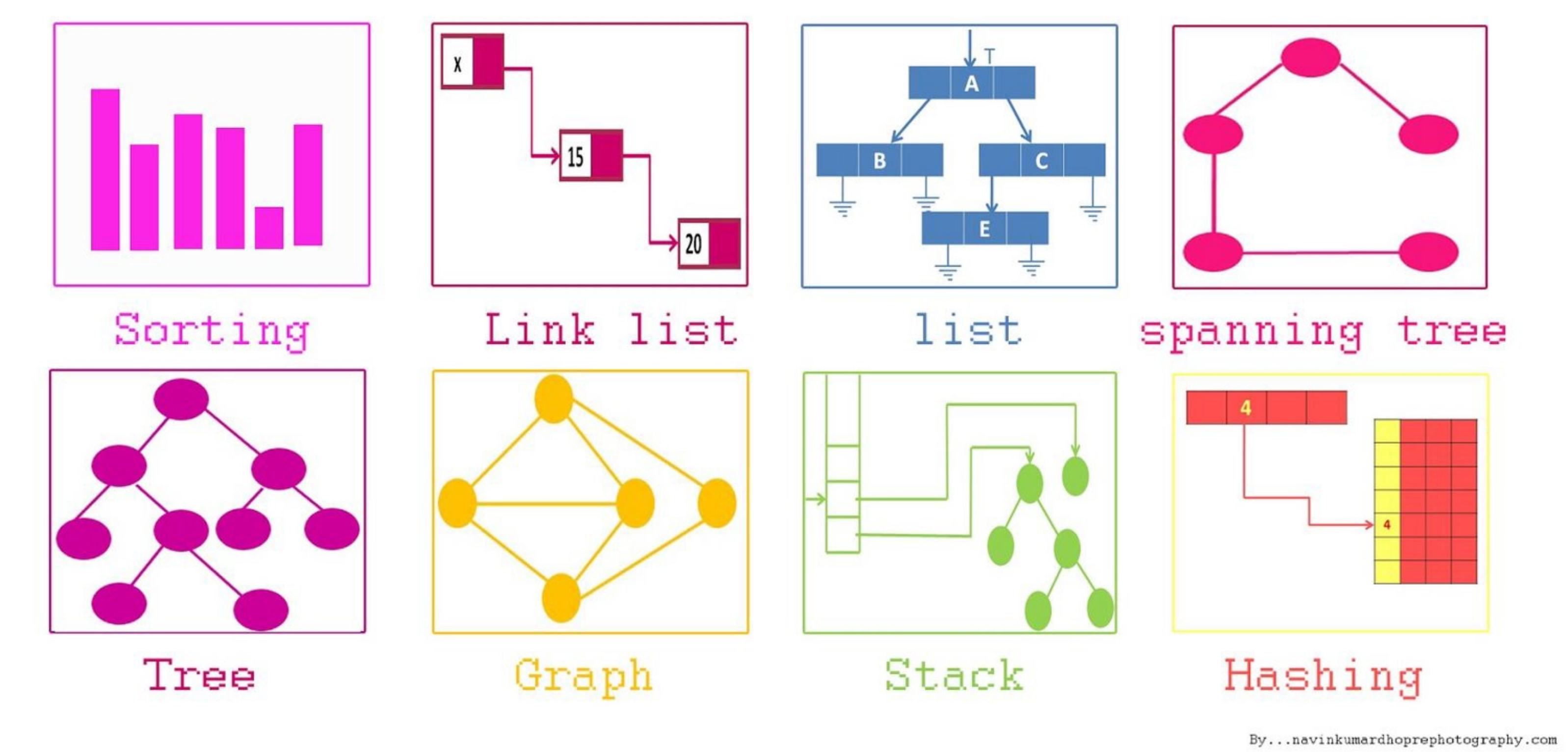
Reference

- Rance D. Necaise. Data Structures and algorithms using python. Chapter2. John Wiley&Sons,Inc., 2011
- Michael T.Goodrich, Roberto Tamassia, Michael H. Goodwasser. Data Structures and Algorithms in python. Chapter5. John Wiley&Sons,Inc. 2013

Abstract Data Type (ADT) =

A definition for a data type

- a set of values
- a set of Operations allowed on data type

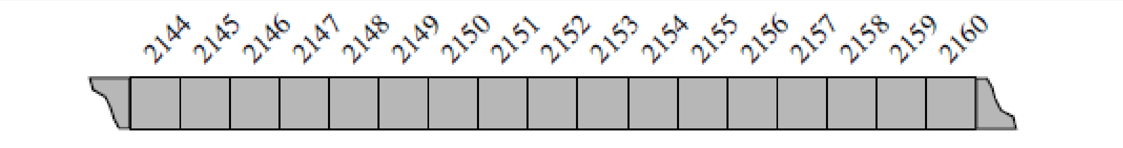


Array ADT

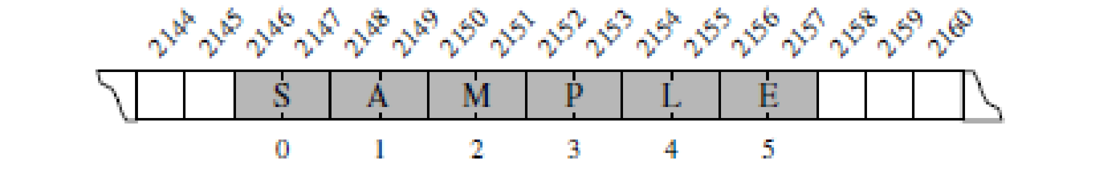
A one-dimensional array is a collection of contiguous elements in which individual elements are identified by a unique integer subscript starting with zero. Once an array is created, its size cannot be changed.

- **Array(size):** Creates a one-dimensional array consisting of size elements with each element initially set to None. size must be greater than zero.
- **length ():** Returns the length or number of elements in the array.
- **getitem (index):** Returns the value stored in the array at element position index. The index argument must be within the valid range. Accessed using the subscript operator.
- **setitem (index, value):** Modifies the contents of the array element at position index to contain value. The index must be within the valid range. Accessed using the subscript operator.
- **clearing(value):** Clears the array by setting every element to value.
- **iterator ():** Creates and returns an iterator that can be used to traverse the elements of the array.

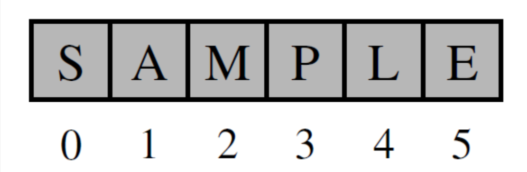
Data stored in Memory address



an array of six characters, requires 12 bytes of memory. We will refer to each location within an array as a cell, and will use an **integer index** to describe its location within the array



sequential data type

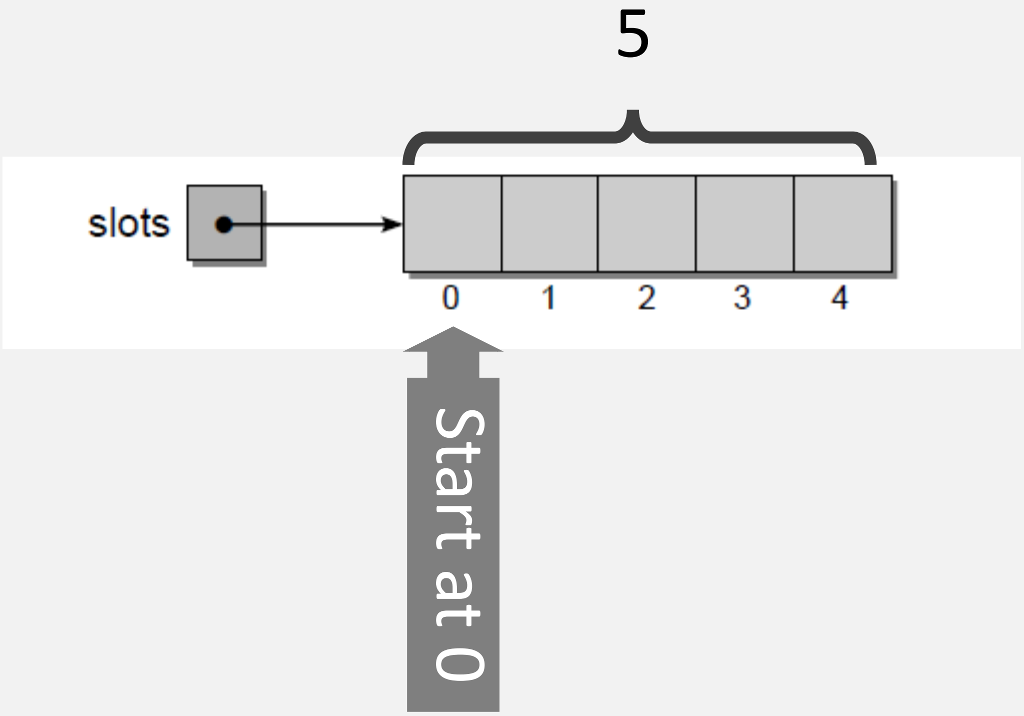


higher-level abstraction

array(size) - Create array with size elements

```
import ctypes

ArrayType = ctypes.py_object * 5
slots = ArrayType()
```



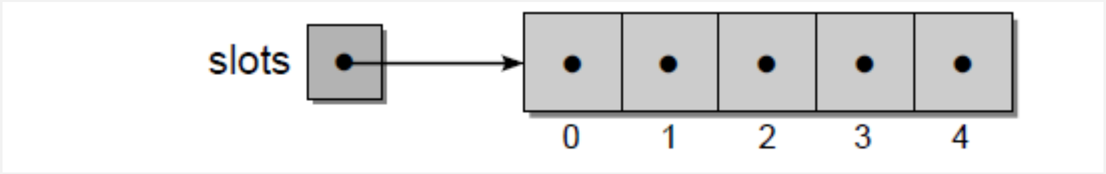
getitem(index) - recall the value at slots[index]

```
print( slots[0] )
```

→ Error

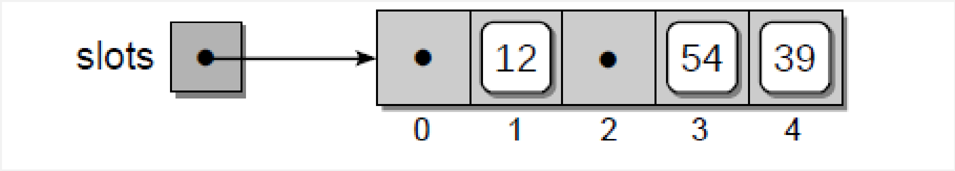
clear(None) - assign all value to **N o n e**

```
for i in range( 5 ) :
  slots[i] = None
```



setitem(index,value) - assign other values

```
slots[1] = 12
slots[3] = 54
slots[4] = 37
```



This module provides access to the diverse set of data types available in the C language and the complete functionality provided by a wide range of C libraries.

```python
# Implements the Array ADT using array capabilities of the ctypes module.
import ctypes

class Array :
   # Creates an array with size elements.
  def __init__( self, size ):
    assert size > 0, "Array size must be > 0"
    self._size = size
     # Create the array structure using the ctypes module.
    PyArrayType = ctypes.py_object * size
    self._elements = PyArrayType()
     # Initialize each element.
    self.clear( None )

   # Returns the size of the array.
  def __len__( self ):
    return self._size
```

check status, “Error message”

Constructor

The __len__method, which returns the number of elements in the array, simply returns the value of size that was saved in the constructor.

the __getitem__ operator method takes the array index as an argument and returns the value of the corresponding element.

```
# Gets the contents of the index element.
def __getitem__( self, index ):
  assert index >= 0 and index < len(self), "Array subscript out of range"
  return self._elements[ index ]

# Puts the value in the array element at index position.
def __setitem__( self, index, value ):
  assert index >= 0 and index < len(self), "Array subscript out of range"
  self._elements[ index ] = value

# Clears the array by setting each element to the given value.
def clear( self, value ):
  for i in range( len(self) ) :
    self._elements[i] = value
```

check status, “Error message”

the __setitem__operator method is used to set or change the contents of a specific element of the array

Difference between _, __ and __xx__ in Python

- **One underline in the beginning**

Python doesn't have real private methods, so one underline in the beginning of a method or attribute means you shouldn't access this method, because it's not part of the API.
e.g. _spam, _name, _data

- **Two underlines in the beginning**

This one causes a lot of confusion. It should not be used to mark a method as private, the goal here is to avoid your method to be overridden by a subclass.
e.g. __methodA( ), __bite( )

Private Variables

“Private” instance variables that cannot be accessed except from inside an object don’t exist in Python.

However, there is a convention that is followed by most Python code: a name prefixed with an underscore

e.g. _spam, _data, _name

should be treated as a non-public part of the API
(whether it is a function, a method or a data member).

```
class A(object):
    def __method(self):
        print ('I'm a method in A')

    def method(self):
        self.__method()

a = A()
a.method()
```

answer: I'm a method in A

```
class B(A):
    def __method(self):
        print ('I'm a method in B')

b = B()
b.method()
```

answer:

a) I’m a method in A

b) I'm a method in B

- **Two underlines in the beginning and in the end**

When you see a method like __this__, the rule is simple: don't call it.
Why? Because it means it's a method python calls, not you.

  - __init__( )
  - __len__( )
  - __add__( )
  - __sub__( )
  - __repr__( )
  - __str__( )
  - ………..
  - ………..

http://igorsobreira.com/2010/09/16/difference-between-one-underline-and-two-underlines-in-python.html

Python List →Dynamic Array

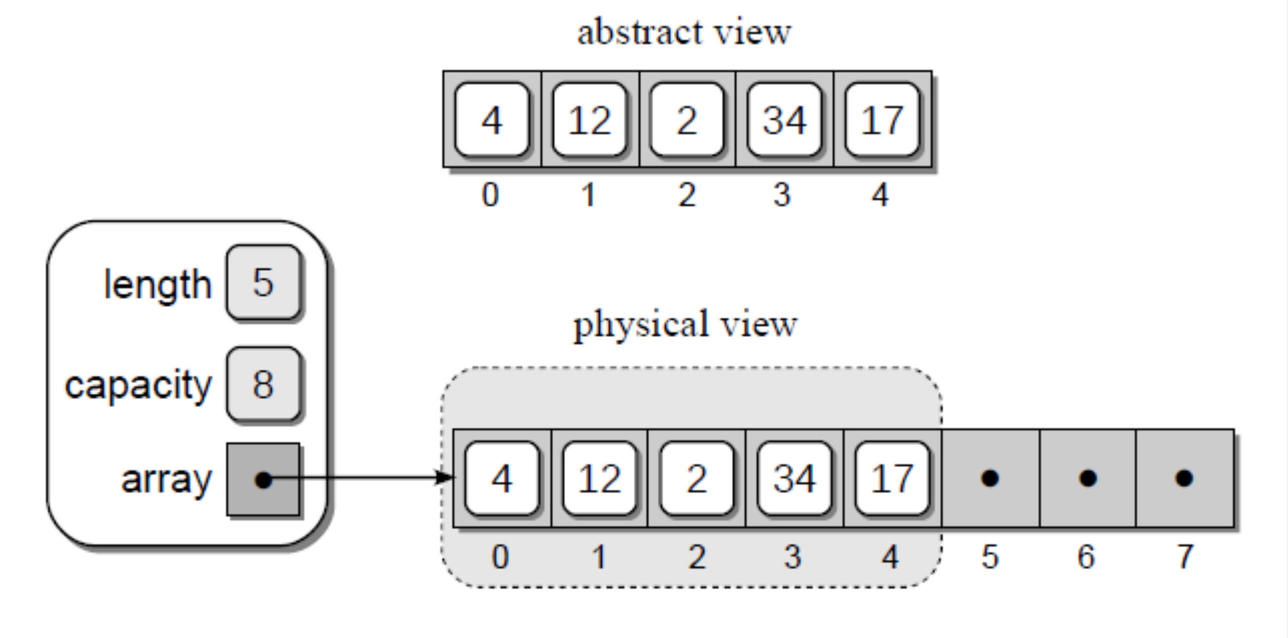
```
import sys                                           # provides getsizeof function
data = [ ]
for k in range(n):                                   # NOTE: must fix choice of n
  a = len(data)                                      # number of elements
  b = sys.getsizeof(data)                     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

Array vs Python List

- an array has a limited number of operations
- The array is best suited for problems requiring a sequence in which the maximum number of elements are known up front
- list is the better choice when the size of the sequence needs to change after it has been created
- Python list will use four to five time as much memory

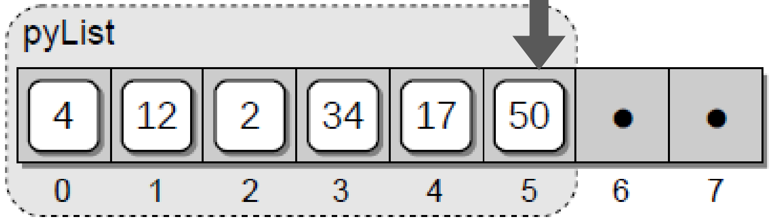
The Python List

```
pyList = [ 4, 12, 2, 34, 17 ]
```

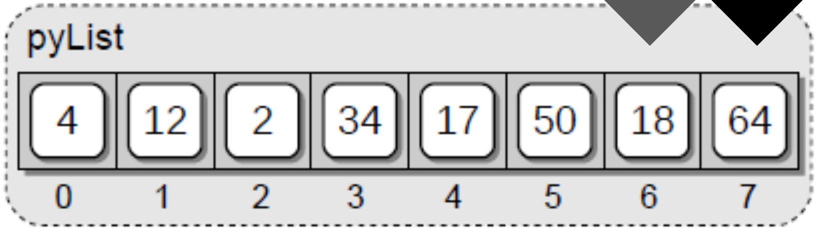


Append the list

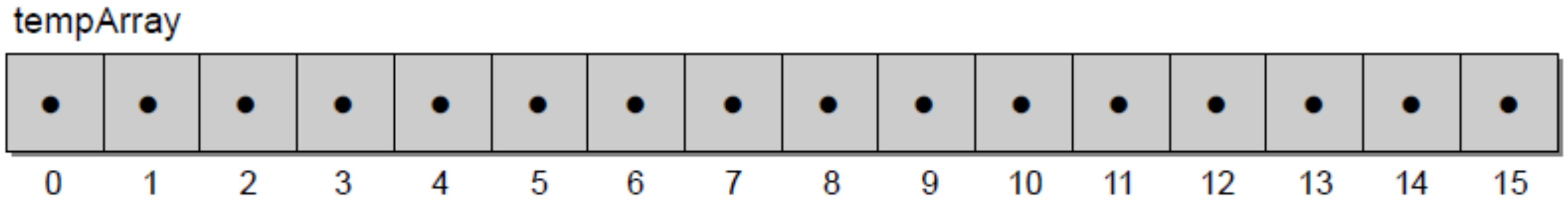
```
pyList.append( 50 )
```



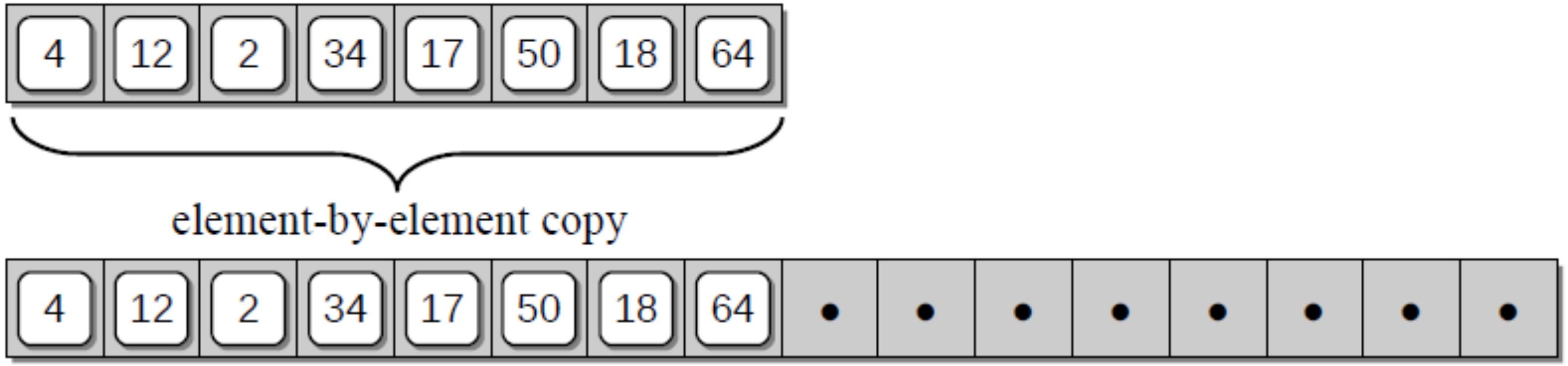

```
pyList.append( 18 )
pyList.append( 64 )
pyList.append( 6 )
```



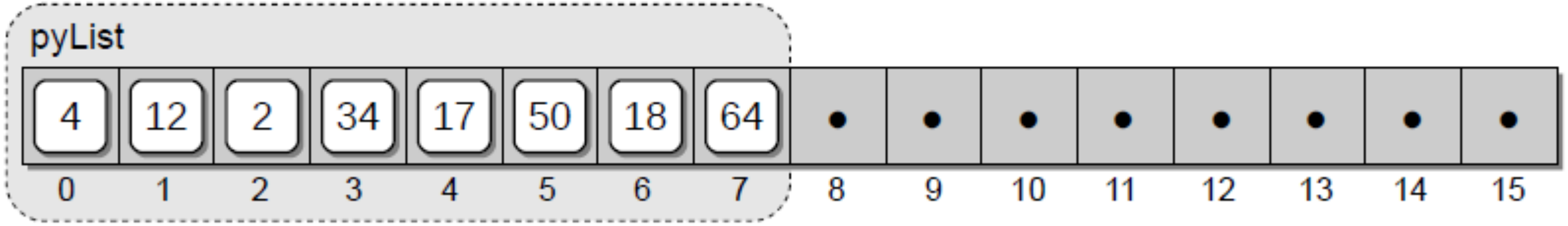
(1) A new array, double the size of the original, is created.



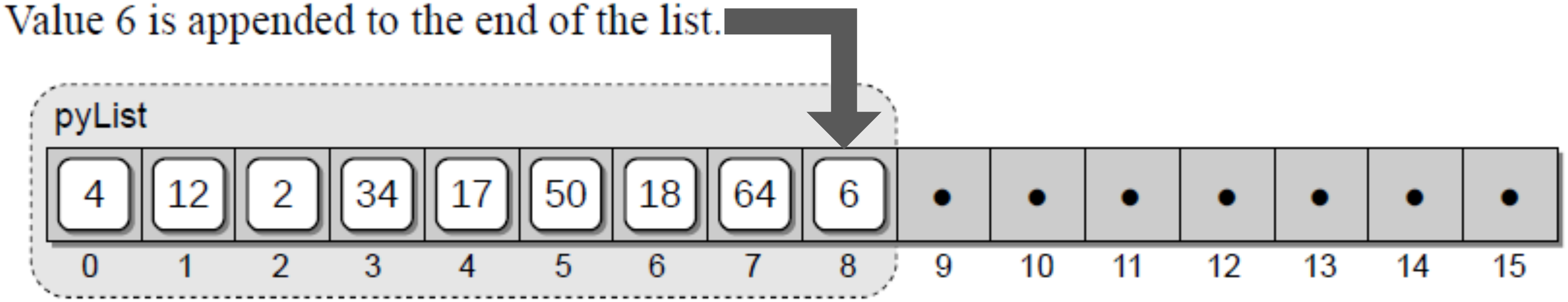
(2) The values from the original array are copied to the new larger array.



(3) The new array replaces the original in the list.

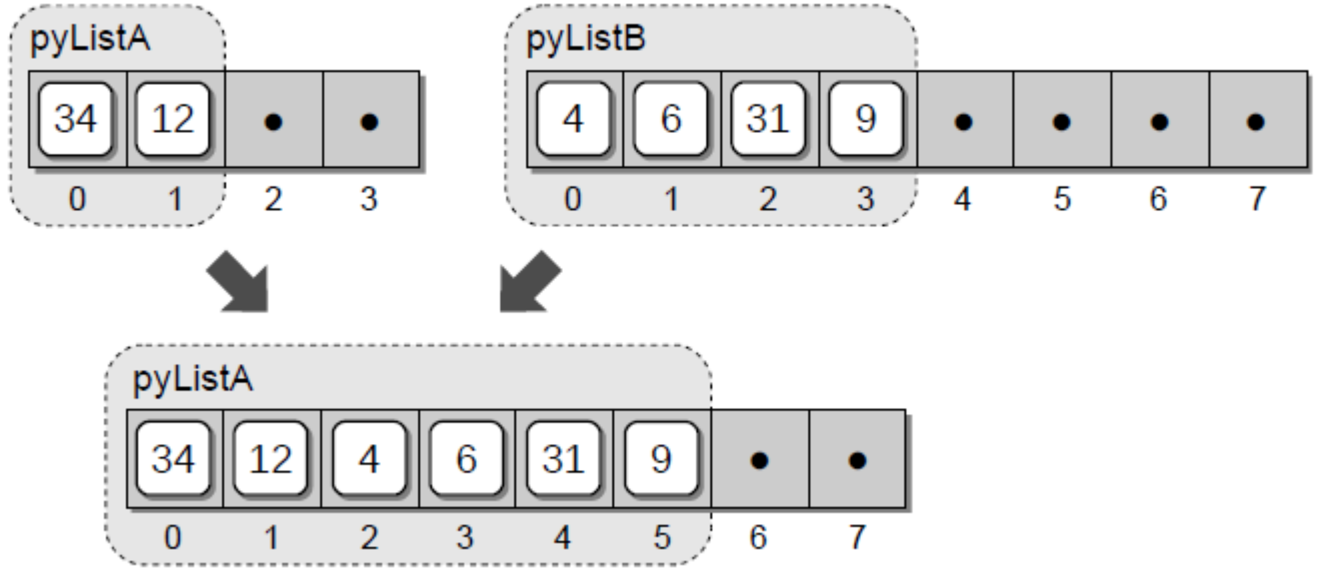


(4) Value 6 is appended to the end of the list.



Extend the list

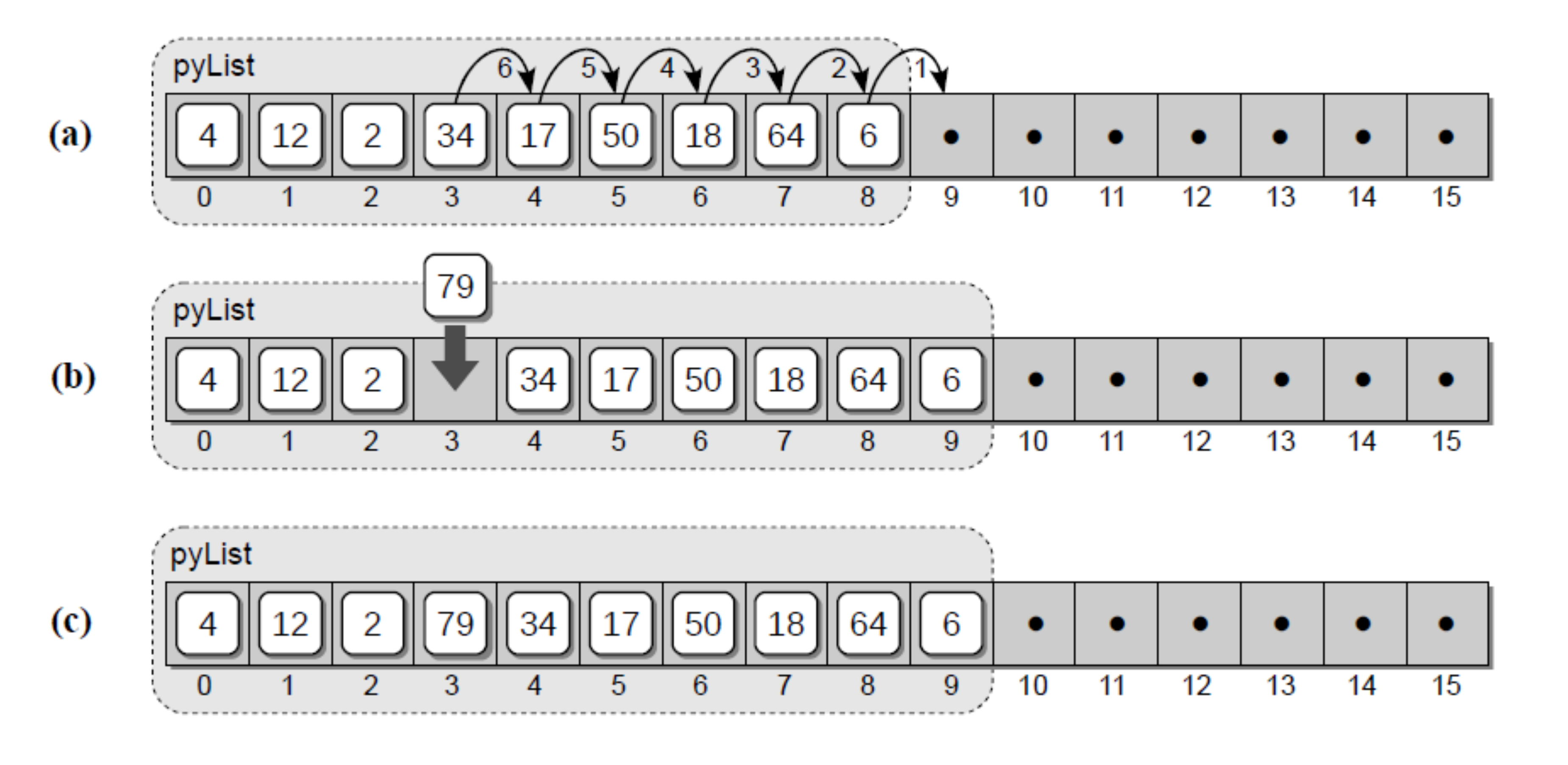
```
pyListA = [ 34, 12 ]
pyListB = [ 4, 6, 31, 9 ]
pyListA.extend( pyListB )
```



Insert an element to the list

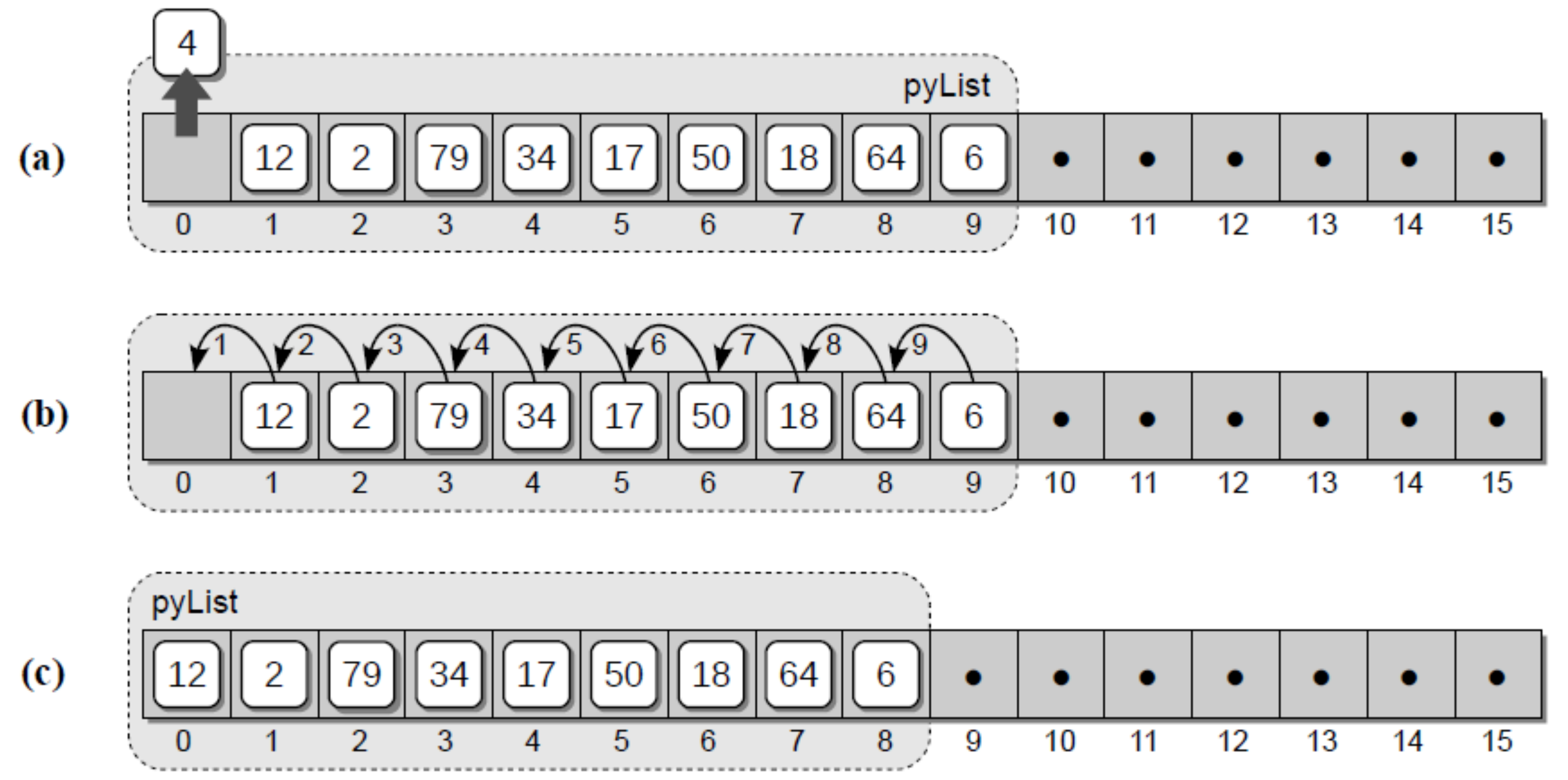
```
pyList.insert( 3, 79 )
```

at position:3 with value:79



pop an element from the list

```
pyList.pop( 0 )    # remove the first item
pyList.pop()       # remove the last item
```

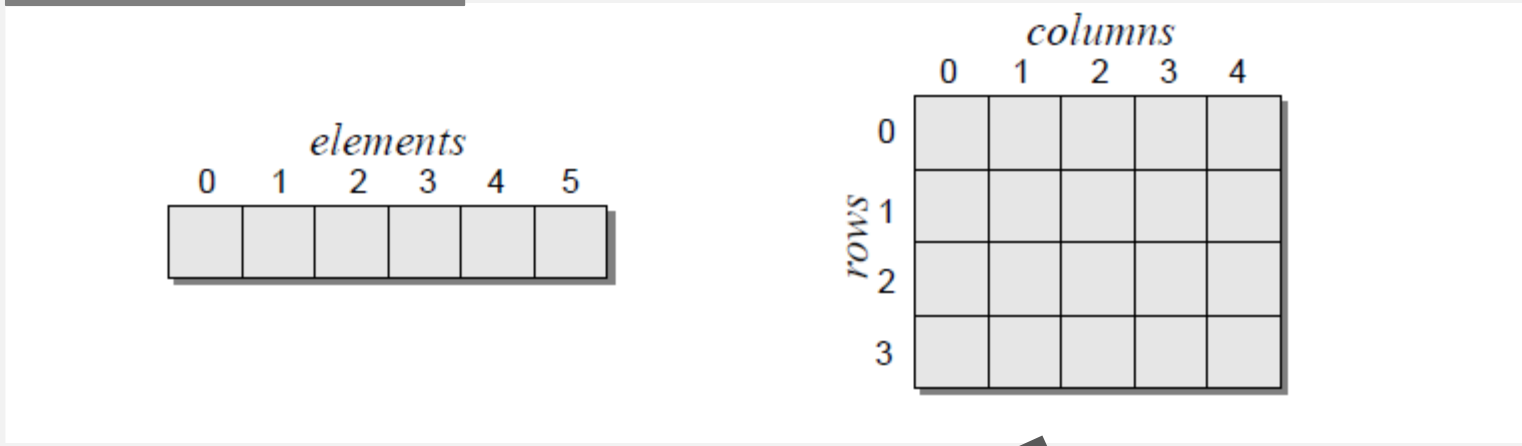


Array2D ADT

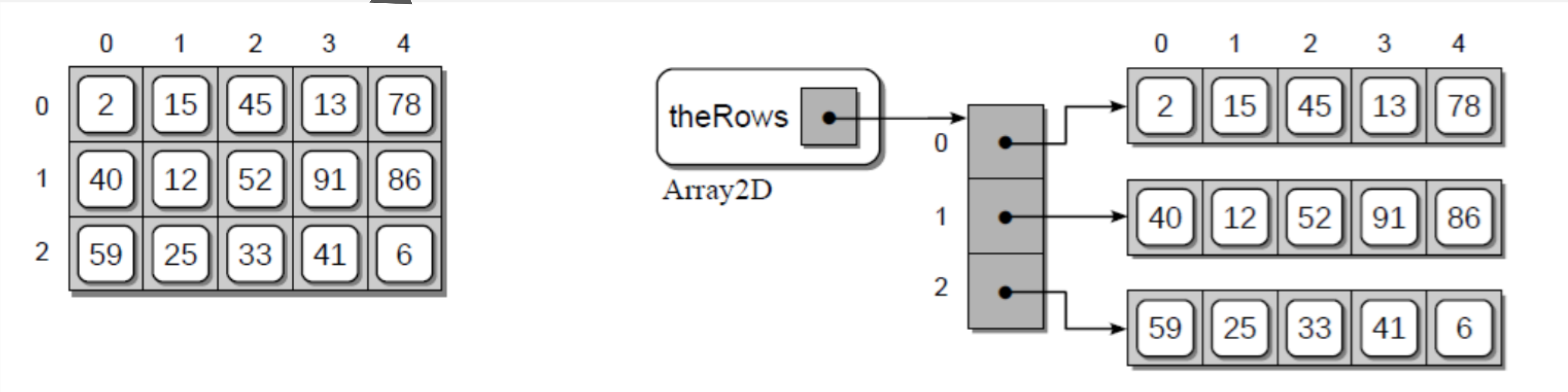
A two-dimensional array consists of a collection of elements organized into rows and columns. Individual elements are referenced by specifying the specific row and column indices (r; c), both of which start at 0.

- **Array2D(nrows, ncols):** Creates a two-dimensional array organized into rows and columns. The nrows and ncols arguments indicate the size of the table. The individual elements of the table are initialized to None.
- **numRows():** Returns the number of rows in the 2-D array.
- **numCols():** Returns the number of columns in the 2-D array.
- **clear(value):** Clears the array by setting each element to the given value.
- **getitem(i1, i2):** Returns the value stored in the 2-D array element at the position indicated by the 2-tuple (i1; i2), both of which must be within the valid range. Accessed using the subscript operator: y = x[1,2].
- **setitem(i1, i2, value):** Modifies the contents of the 2-D array element indicated by the 2-tuple (i1; i2) with the new value. Both indices must be within the valid range. Accessed using the subscript operator: x[0,3] = y.

The 2D-Array



2-dimensional array creates from a number of 1-dimensional arrays (theRows)



Matrix ADT

A matrix is a collection of scalar values arranged in rows and columns as a rectangular grid of a fixed size. The elements of the matrix can be accessed by specifying a given row and column index with indices starting at 0.

- **Matrix(rows, ncols):** Creates a new matrix containing nrows and ncols with each element initialized to 0.
- **numRows()**: Returns the number of rows in the matrix.
- **numCols():** Returns the number of columns in the matrix.
- **getitem (row, col):** Returns the value stored in the given matrix element. Both row and col must be within the valid range.
- **setitem (row, col, scalar):** Sets the matrix element at the given row and col to scalar. The element indices must be within the valid range.

Additional operations

- **scaleBy(scalar):** Multiplies each element of the matrix by the given scalar value. The matrix is modified by this operation.
- **transpose():** Returns a new matrix that is the transpose of this matrix.
- **add (rhsMatrix):** Creates and returns a new matrix that is the result of adding this matrix to the given rhsMatrix. The size of the two matrices must be the same.
- **subtract (rhsMatrix):** The same as the add() operation but subtracts the two matrices.
- **multiply (rhsMatrix):** Creates and returns a new matrix that is the result of multiplying this matrix to the given rhsMatrix. The two matrices must be of appropriate sizes as defined for matrix multiplication.

Scaling in Matrix

$$3\begin{bmatrix} 6 & 7 \\ 8 & 9 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 3*6 & 3*7 \\ 3*8 & 3*9 \\ 3*1 & 3*0 \end{bmatrix} = \begin{bmatrix} 18 & 21 \\ 24 & 27 \\ 3 & 0 \end{bmatrix}$$

```
# Scales the matrix by the given scalar.
def scaleBy( self, scalar ):
  for r in range( self.numRows() ) :
    for c in range( self.numCols() ) :
      self[ r, c ] *= scalar
```

Matrix Addition

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} + \begin{bmatrix} 6 & 7 \\ 8 & 9 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0+6 & 1+7 \\ 2+8 & 3+9 \\ 4+1 & 5+0 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \\ 5 & 5 \end{bmatrix}$$

```
def __add__( self, rhsMatrix ):
  assert rhsMatrix.numRows() == self.numRows() and \
         rhsMatrix.numCols() == self.numCols(), \
    "Matrix sizes not compatible for the add operation."
   # Create the new matrix.
  newMatrix = Matrix( self.numRows(), self.numCols() )
   # Add the corresponding elements in the two matrices.
  for r in range( self.numRows() ) :
    for c in range( self.numCols() ) :
      newMatrix[ r, c ] = self[ r, c ] + rhsMatrix[ r, c ]
  return newMatrix
```

Matrix Subtraction

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} - \begin{bmatrix} 6 & 7 \\ 8 & 9 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0-6 & 1-7 \\ 2-8 & 3-9 \\ 4-1 & 5-0 \end{bmatrix} = \begin{bmatrix} -6 & -6 \\ -6 & -6 \\ 3 & 5 \end{bmatrix}$$

Matrix transposition

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}^T = \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix}$$

Matrix Multiplication

$$\begin{aligned} & \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} * \begin{bmatrix} 6 & 7 & 8 \\ 9 & 1 & 0 \end{bmatrix} \\ = & \begin{bmatrix} (0*6+1*9) & (0*7+1*1) & (0*8+1*0) \\ (2*6+3*9) & (2*7+3*1) & (2*8+3*0) \\ (4*6+5*9) & (4*7+5*1) & (4*8+5*0) \end{bmatrix} \\ = & \begin{bmatrix} 9 & 1 & 0 \\ 39 & 17 & 16 \\ 69 & 33 & 32 \end{bmatrix} \end{aligned}$$

$$A = \begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \\ A_{2,0} & A_{2,1} \end{bmatrix} \qquad B = \begin{bmatrix} B_{0,0} & B_{0,1} & B_{0,2} \\ B_{1,0} & B_{1,1} & B_{1,2} \end{bmatrix}$$

$$\begin{aligned} C_{0,0} &= A_{0,0} * B_{0,0} + A_{0,1} * B_{1,0} \\ C_{0,1} &= A_{0,0} * B_{0,1} + A_{0,1} * B_{1,1} \\ C_{0,2} &= A_{0,0} * B_{0,2} + A_{0,1} * B_{1,2} \\ C_{1,0} &= A_{1,0} * B_{0,0} + A_{1,1} * B_{1,0} \\ C_{1,1} &= A_{1,0} * B_{0,1} + A_{1,1} * B_{1,1} \\ C_{1,2} &= A_{1,0} * B_{0,2} + A_{1,1} * B_{1,2} \\ C_{2,0} &= A_{2,0} * B_{0,0} + A_{2,1} * B_{1,0} \\ C_{2,1} &= A_{2,0} * B_{0,1} + A_{2,1} * B_{1,1} \\ C_{2,2} &= A_{2,0} * B_{0,2} + A_{2,1} * B_{1,2} \end{aligned}$$