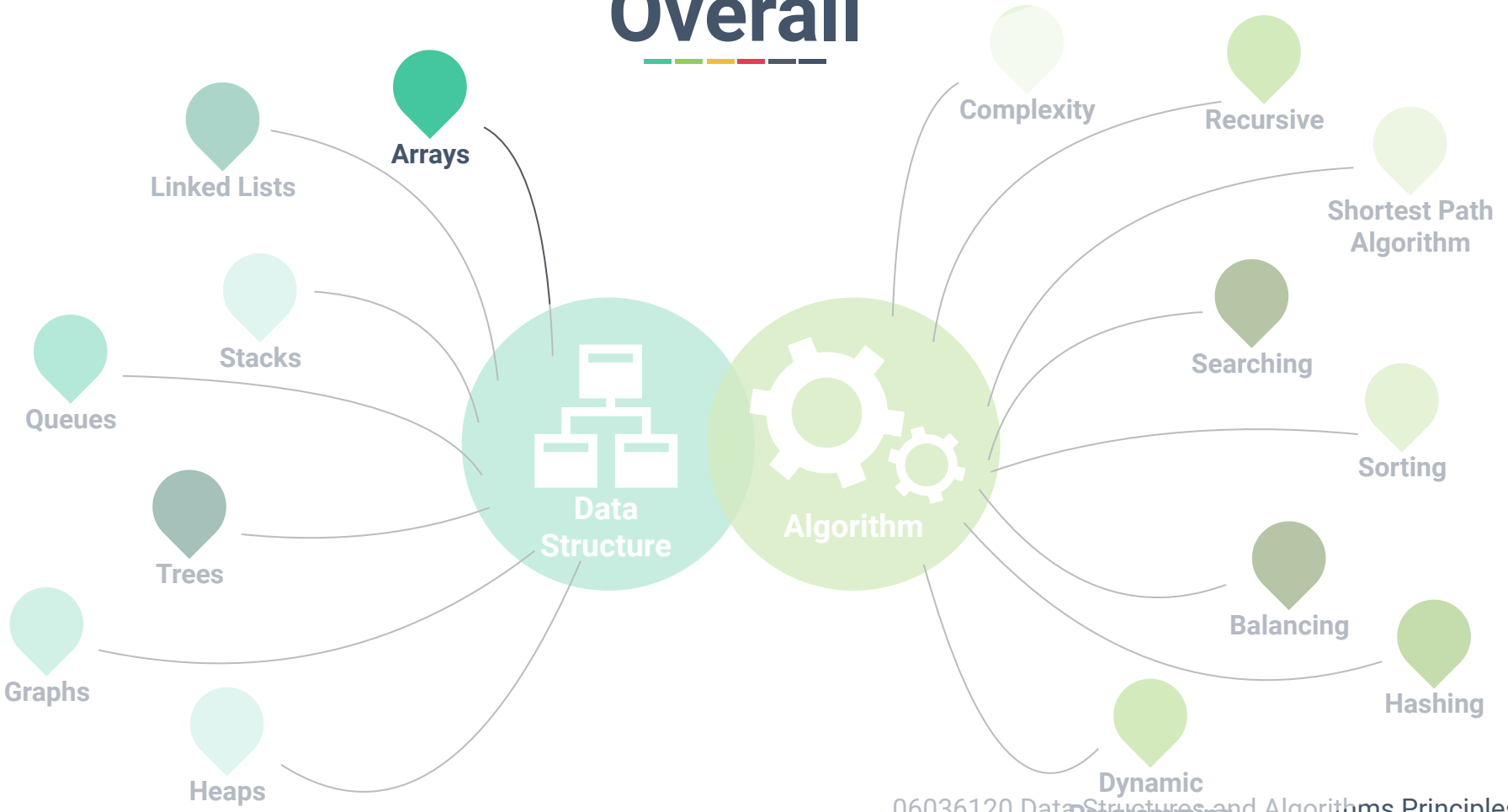
Chapter 2: Arrays

Dr. Sirasit Lochanachit

Overall


Arrays
Linked Lists
Stacks
Queues
Trees
Graphs
Heaps
Data Structure
Algorithm
Complexity
Recursive
Shortest Path Algorithm
Searching
Sorting
Balancing
Hashing
Dynamic Programming

Today's Outline

1. What is an Array?
2. 2D Arrays
3. 3D Arrays
4. Array Operations and Asymptotic Performance

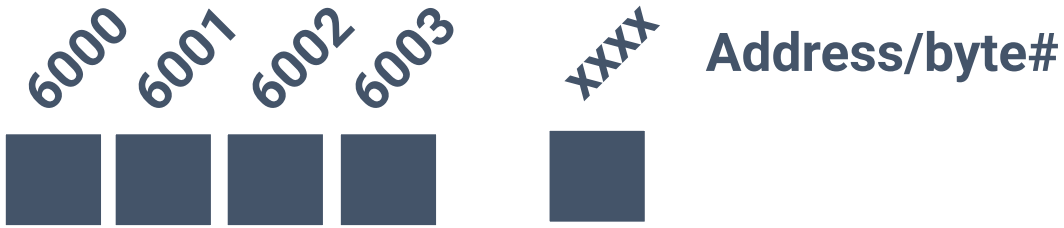
Python Sequence Types

- Built-in class
  - list
  - tuple
  - str

What is an Array?

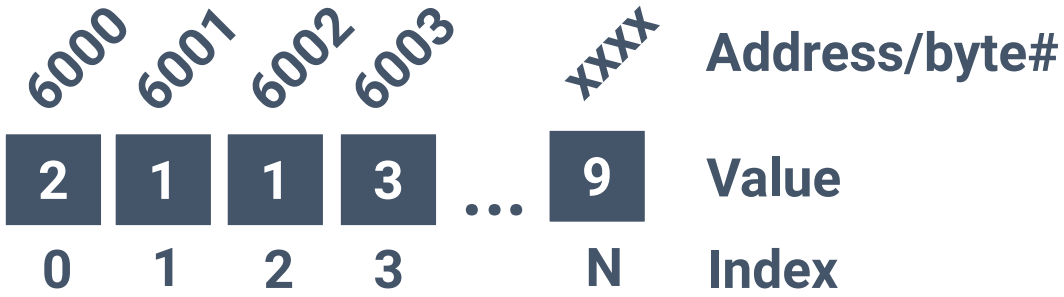
- A computer will have a large number of bytes of memory.

What is an Array?



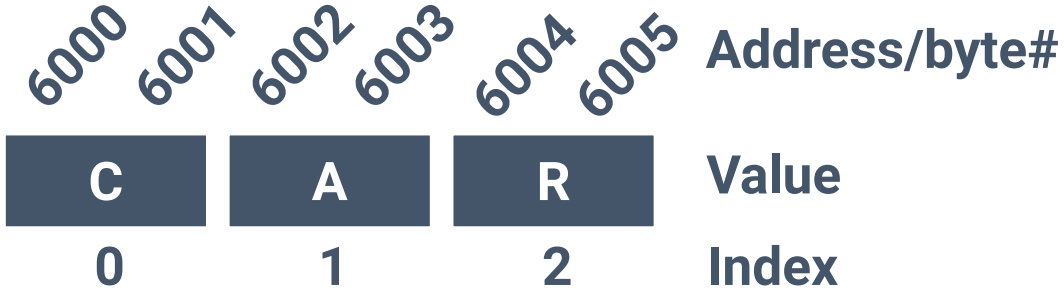
- It has a memory address to keep track of where a data is stored.
  - Each byte has a unique number as its address.
- Although the number is sequential, any byte/element in a RAM can be accessed to read or write with a constant time O(1).

What is an Array?



- An array is a chunk of memory, consisting of equal-size elements.
- Each of those elements has an integer index, which uniquely refers to the value stored.
- The values are all of the same type (integer, character, etc.).

Array of Characters



- In Python, it represents a unicode character with 16 bits (i.e. 2 bytes).
- Since each cell has an equal-size bytes, any element can be accessed constantly with this formula:
  - start_address + cell_size * index

Exercise

Given an array:

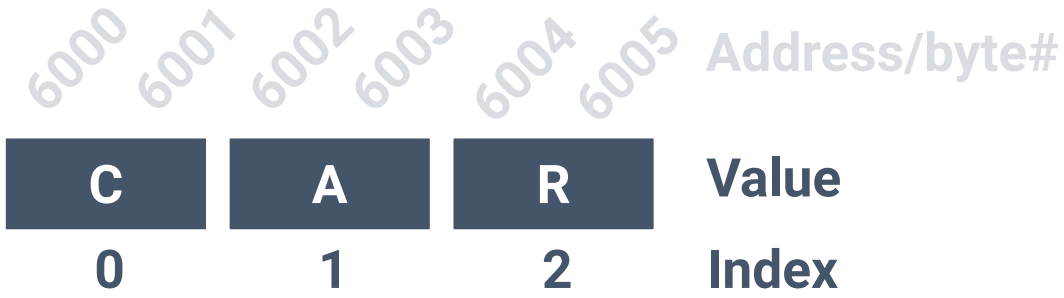
Start Address is 6000,

cell size is 8,

What is the address of the element at index 6?

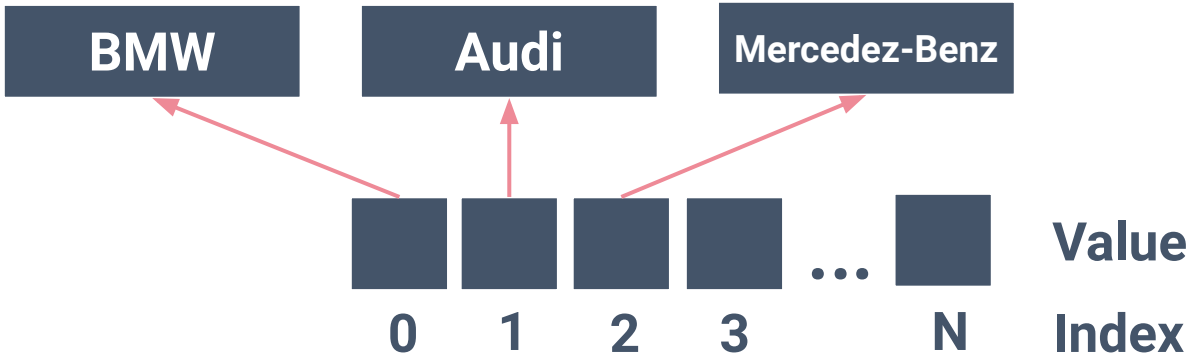
- start_address + cell_size * index

Array of Characters



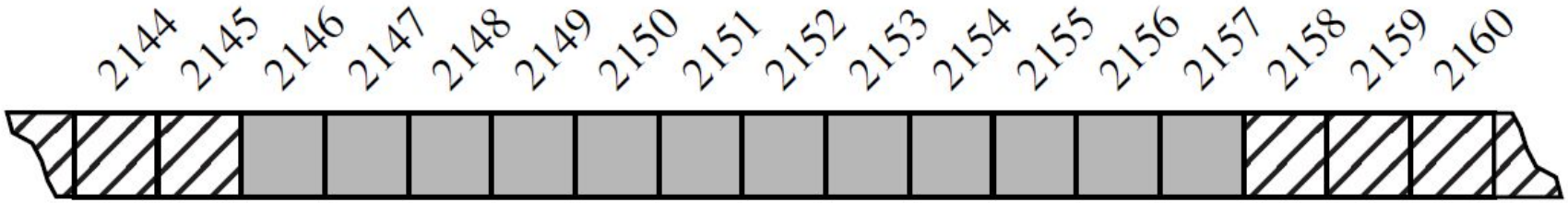
- Luckily, a programming language calculate memory addresses of an array automatically.

Python List



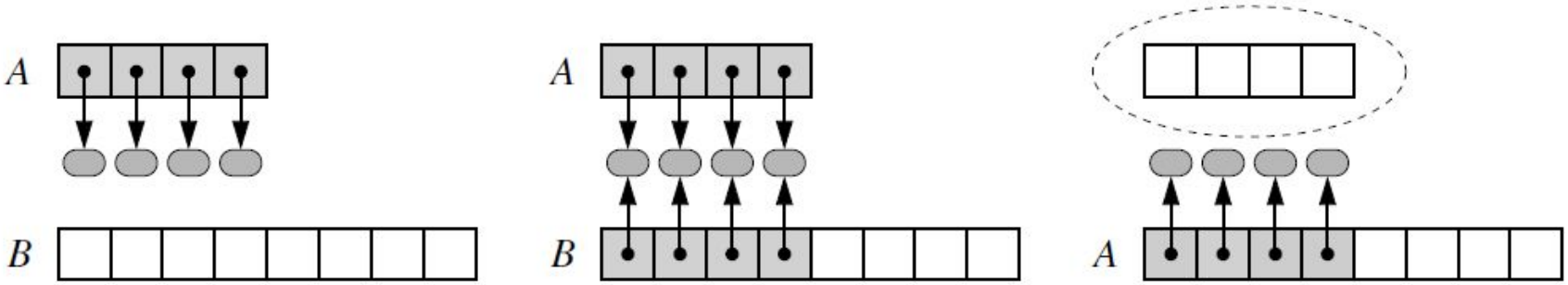
- Python list is a referential-type array that stores the memory addresses (references) of a value instead of the value itself.
- Strings can be in any length, but memory addresses are fixed-size.

Static Array



- Once an array in C or Java has been created, it's size is fixed.
- Python tuple and str instances are immutable.
  - Unable to change size and value

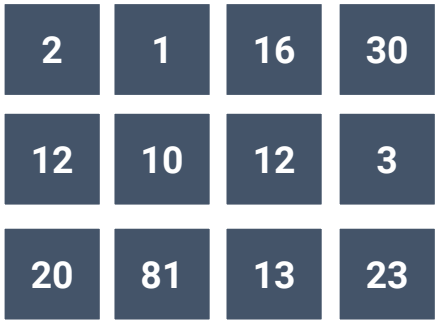
Dynamic Array



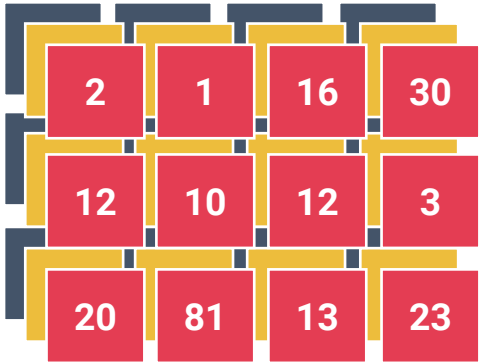
- Python's list instance maintains an underlying array that often has greater capacity than the current length of the list.
- If a capacity is exhausted, the list class requests a new, larger array.

Arrays

(a) One dimension

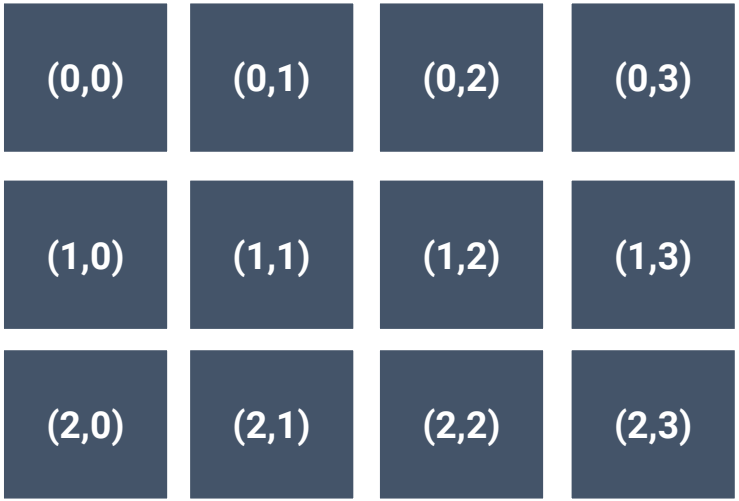


(b) Two dimensions

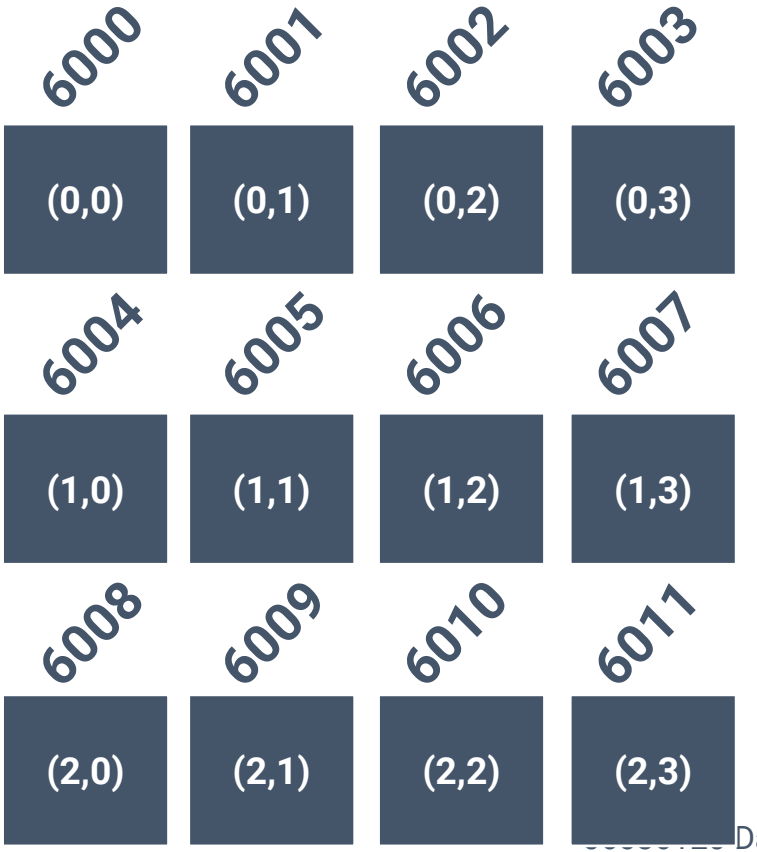


(c) Three dimensions

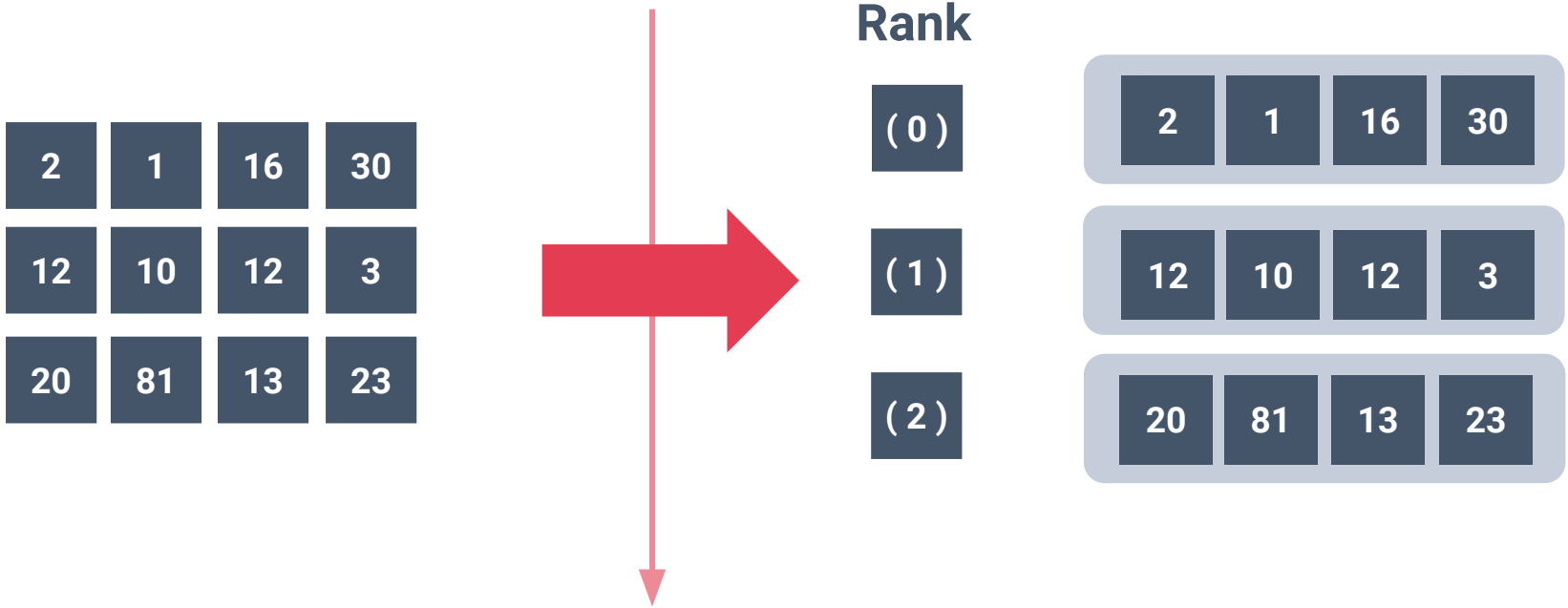
2-Dimensional Arrays



2-Dimensional Arrays



2-D Arrays



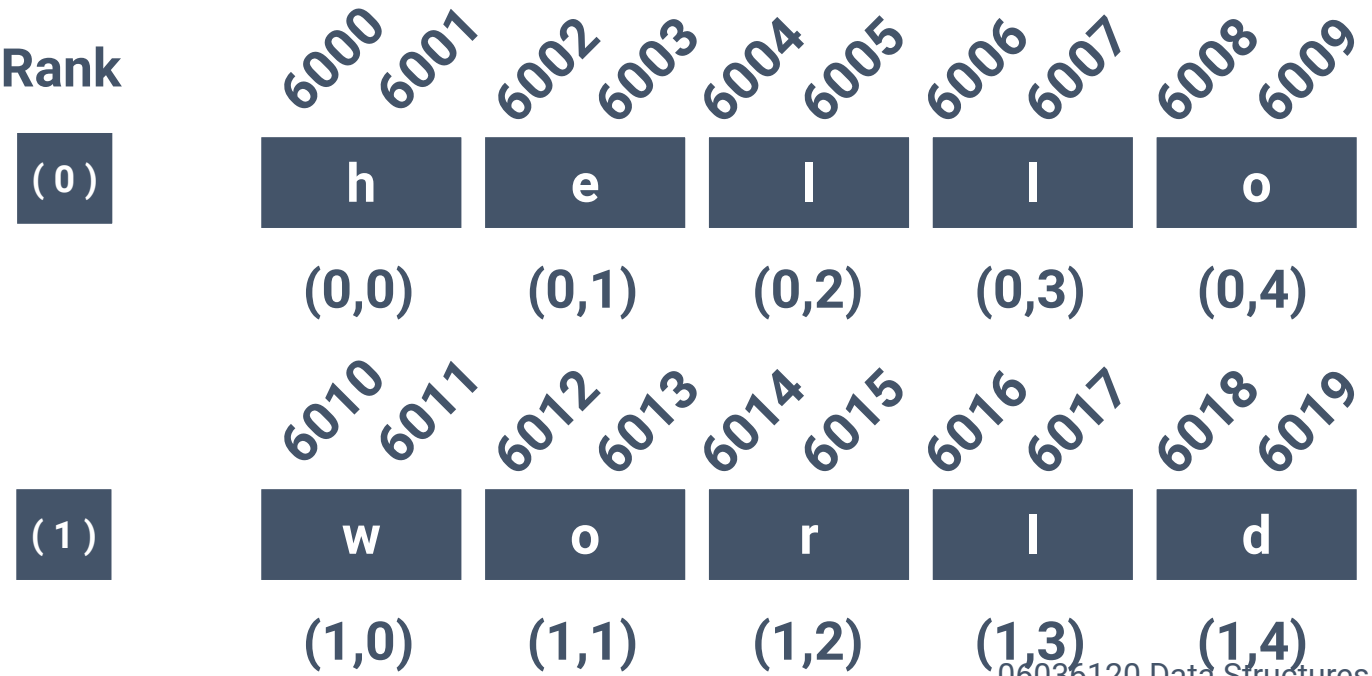
(a) Two dimensions

(b) Nested-one dimension

Example

Python Code: 2-D Arrays

```
c = np.array([['h','e','l','l','o'],['w','o','r','l','d']])
```

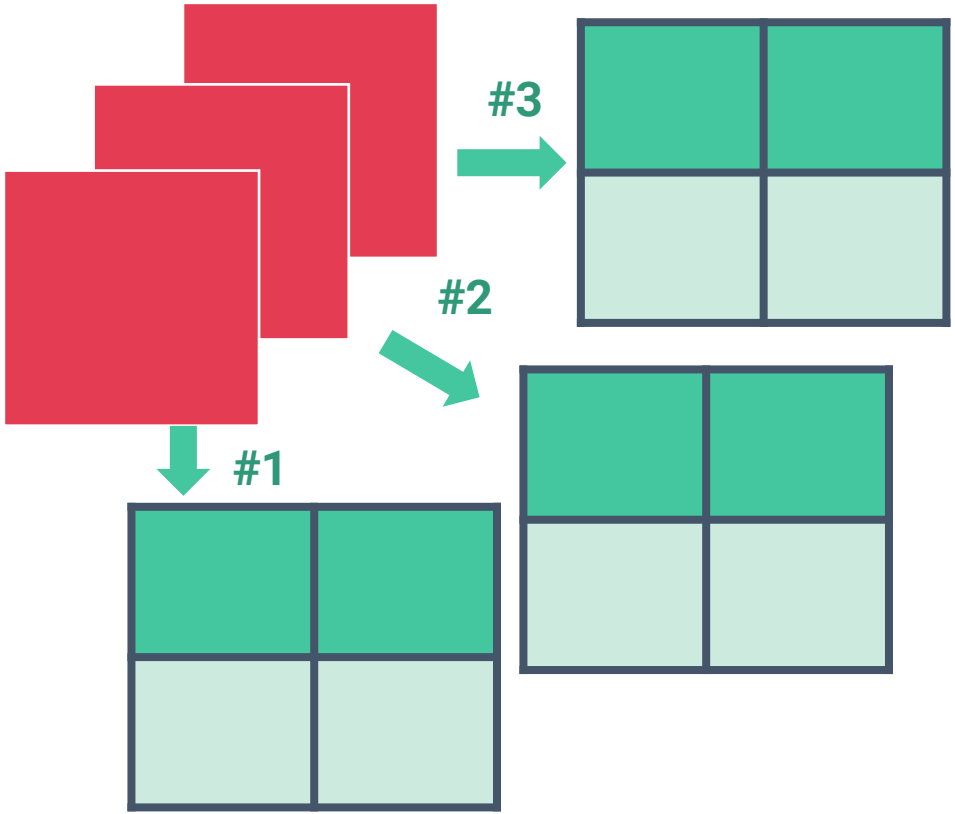


Exercise

Based on previous slide, suppose start address is 6000, find the address of index (1,4)

- start_address + cell_size * index
- Where index = (target_rank * array_length) + target_index

3D Arrays

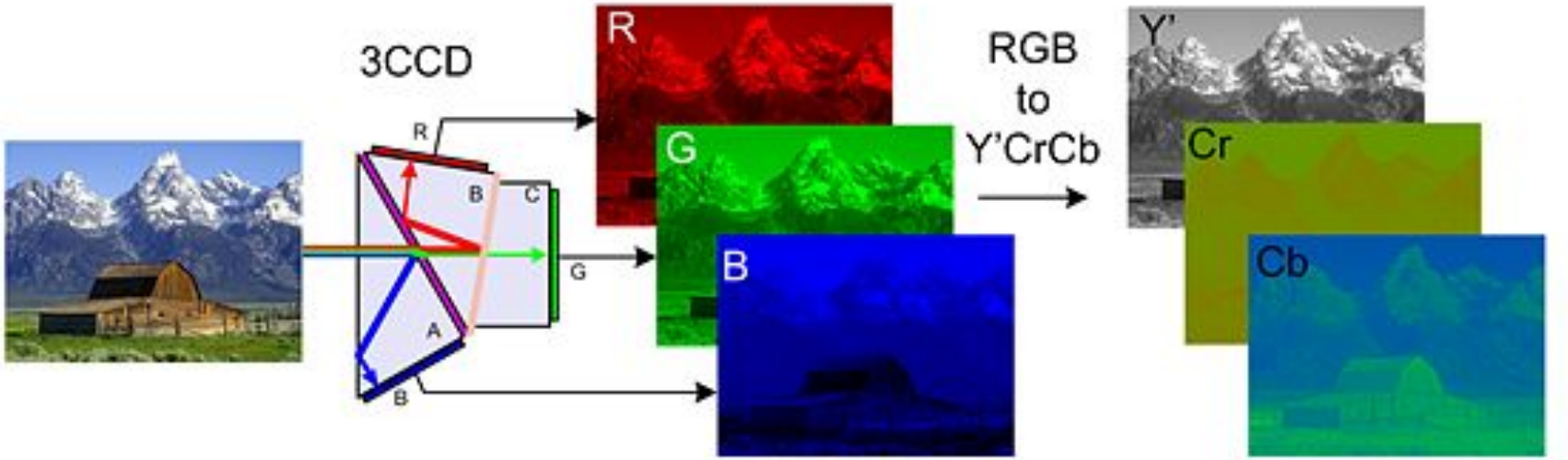


Python Code: 3D Arrays

```
e = np.array([
[[1,2,3],[4,5,6]],
[[7,8,9],[10,11,12]],
[[13,14,15],[16,17,18]]])
```

Dimension
#1
#2
#3

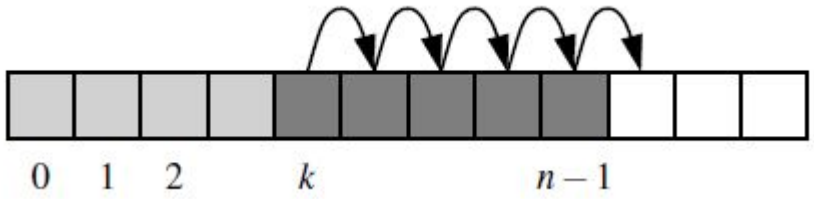
3D Arrays



List Operations

Operation	Running Time
len(data)	*O*(1)
data[i]	*O*(1)
data[i] = val	*O*(1)
c * data	*O*(n)
data.reverse()	*O*(n)
data.sort()	*O*(n log n)

Adding Elements to a List



- Operation **insert**(*k*, *value*) requires creating a room to insert a new element at index *k* of a dynamic array
  - Make room by shifting forward the (*n* - 1) - *k* elements *data*[*k*], … , *data*[*n*-1]
  - O(*n* - *k* + 1) for inserting at index *k*

Assignment 3

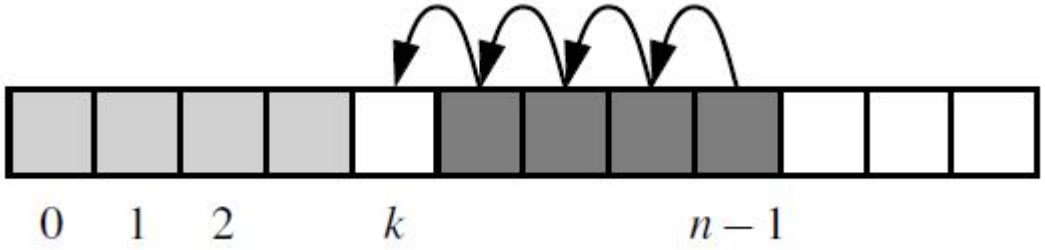
Record the average running time of insert(k, 20) in seconds with three different inserting patterns for each of the *N* calls:

1. Repeatedly insert at the beginning of a list
2. Repeatedly insert near the middle of a list
3. Repeatedly insert at the end of the list
   - Each pattern starts from an empty list[] -> data = []

Assignment 3

	N				
Patterns	100	1,000	10,000	100,000	1,000,000
First Case					
Second Case					
Third Case					

Removing Elements from a List



- Operation **pop**(*k*) removes the element that is at index *k* < *n* of a list
  - Then shifting backward the (*n* - 1) - (*k + 1)* elements *data*[*k*+1], … , *data*[*n*-1]
  - O(*n* - *k*) for removing element at index *k*

Asymptotic Performance

	Add Operation	Running Time	Remove Operation	Running Time
Beginning				
End				
Between				

Data								
	10	5	8	7	1			

Asymptotic Performance

	Add Operation	Running Time	Remove Operation	Running Time
Beginning				
End	data.append(val)	$O(1)$		
Between				

Data	10	5	8	7	1	20		

Asymptotic Performance

	Add Operation	Running Time	Remove Operation	Running Time
Beginning				
End	data.append(val)	$O(1)$	data.pop()	$O(1)$
Between				

Data

10	5	8	7	1			

Asymptotic Performance

	Add Operation	Running Time	Remove Operation	Running Time
Beginning			data.pop(0) Del data[0]	*O*(n)
End	data.append(val)	*O*(1)	data.pop()	*O*(1)
Between				

Data		5	8	7	1			

Asymptotic Performance

	Add Operation	Running Time	Remove Operation	Running Time
Beginning			data.pop(0) Del data[0]	*O*(n)
End	data.append(val)	*O*(1)	data.pop()	*O*(1)
Between				

Data							
5	8	7	1				

Asymptotic Performance

	Add Operation	Running Time	Remove Operation	Running Time
Beginning	data.insert(0, val)	*O*(n)	data.pop(0) Del data[0]	*O*(n)
End	data.append(val)	*O*(1)	data.pop()	*O*(1)
Between				

Data								
	9	5	8	7	1			

Asymptotic Performance

	Add Operation	Running Time	Remove Operation	Running Time
Beginning	data.insert(0, val)	*O*(n)	data.pop(0) Del data[0]	*O*(n)
End	data.append(val)	*O*(1)	data.pop()	*O*(1)
Between			data.remove(val)	*O*(n)

Data

9	5		7	1			

Asymptotic Performance

	Add Operation	Running Time	Remove Operation	Running Time
Beginning	data.insert(0, val)	*O*(n)	data.pop(0) Del data[0]	*O*(n)
End	data.append(val)	*O*(1)	data.pop()	*O*(1)
Between			data.remove(val)	*O*(n)

Data								
	9	5	7	1				

Asymptotic Performance

	Add Operation	Running Time	Remove Operation	Running Time
Beginning	data.insert(0, val)	*O*(n)	data.pop(0) Del data[0]	*O*(n)
End	data.append(val)	*O*(1)	data.pop()	*O*(1)
Between	data.insert(index, val)	*O*(n)	data.remove(val)	*O*(n)

Data	9	0	5	7	1			