CS151 Intro to Data Structures

Maps

Announcements

HW5 Autograder fixed

Lab8 and HW6 due Sunday

Outline

- Warm up - review
- Maps
  - Operations:
    - get
    - put (insert)
    - remove

- ArrayMap Implementation

- Intro to Hash Maps

Warm up: Choosing the Right Data Structure...

You are building a task management system that needs to handle various operations efficiently. Each task has an associated importance level, and tasks can be completed, added, or queried frequently. The operations that your system needs to support are:

1. **Add a task**: Insert a new task with a specified importance level.
2. **Remove most important task**: Complete and remove the task with the highest importance.
3. **Get most important task**: View the task with the highest importance without removing it.
4. **Check if tasks are available**: Determine if there are any tasks left to complete.
5. **Update the importance of an existing task**: Change the importance level of a specific task.

Warm up: Choosing the Right Data Structure...

Priority Queue

1. **Add a task**:
   a. O(logn)
2. **Remove most important task**:
   a. O(logn)
3. **Get most important task**:
   a. O(1)
4. **Check if tasks are available**:
   a. O(1)
5. **Update the importance of an existing task**:
   a. O(logn) assuming you have the location of the given task

Maps

Map Motivation

Suppose I have an array of Students. What is the runtime complexity of the following operations?

Update the midterm grade of the second student
O(1)

Update the midterm grade for Liam
O(n) I need to search!

Student:

String name;
double[] hwGrades;
double[] labGrades;
double midtermGrade;
...

Student name = Emily	**Student** name = Aiden	**Student** name = Sophia	**Student** name = Liam	**Student** name = Isabella	**Student** name = Noah	**Student** name = Ava ....

Maps

- Also called “dictionaries” or “associative arrays”
- Similar syntax to an array:
  - `m[key]` retrieves a value
  - `m[key] = value` assigns a value
  - keys need not be ints
- data structure that stores a collection of key-value pairs

Key-Value Pairs

- Each element in a map consists of (K, V)

- The key is used to identify the value
  - In an array, this would be the index

- **Examples**: what are the keys and values here?
  - Dictionary
  - Phone Book
  - Student grades

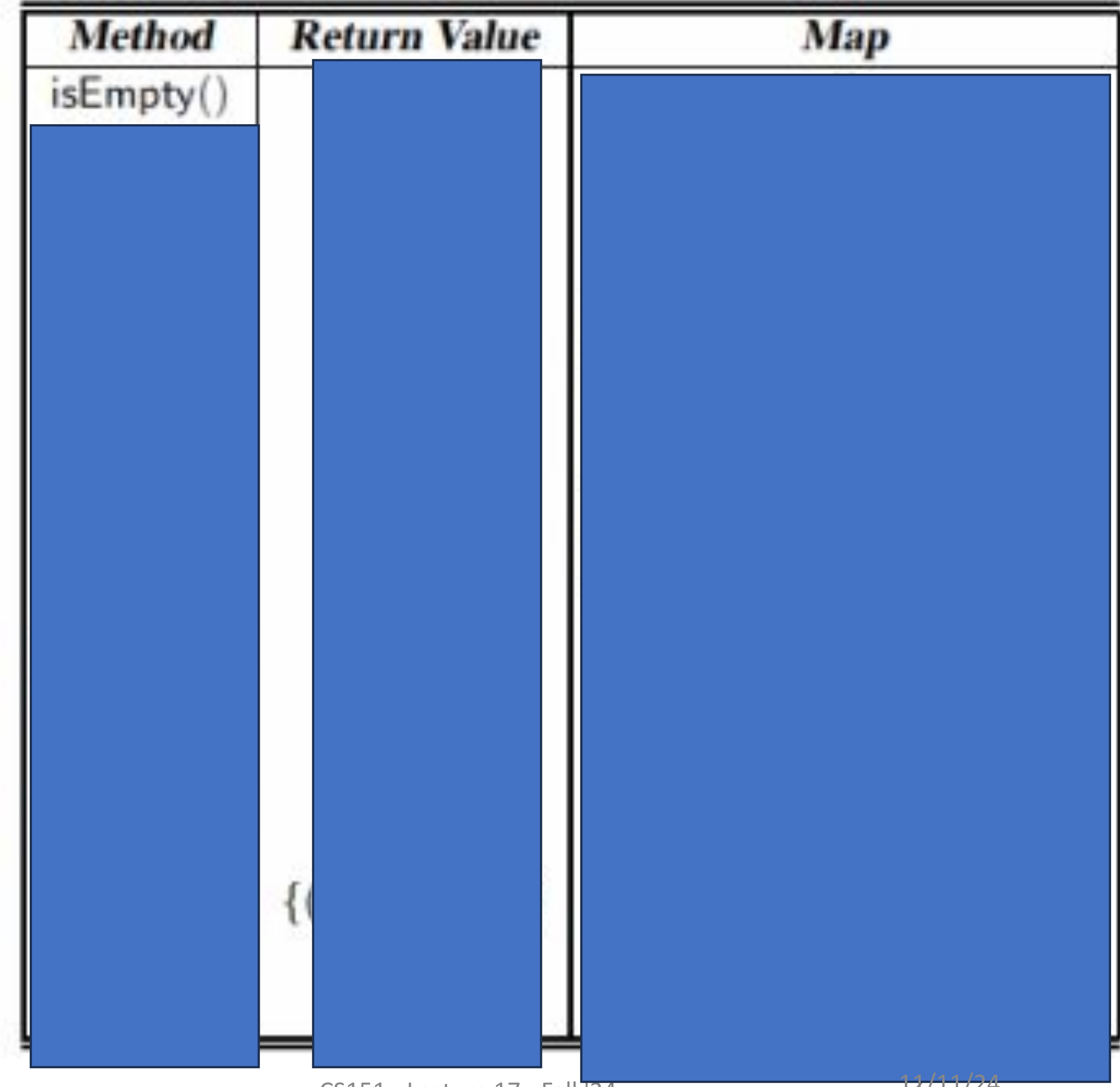
Map

- A indexable collection of key-value pairs
- Multiple entries with the same key are not allowed

Map ADT

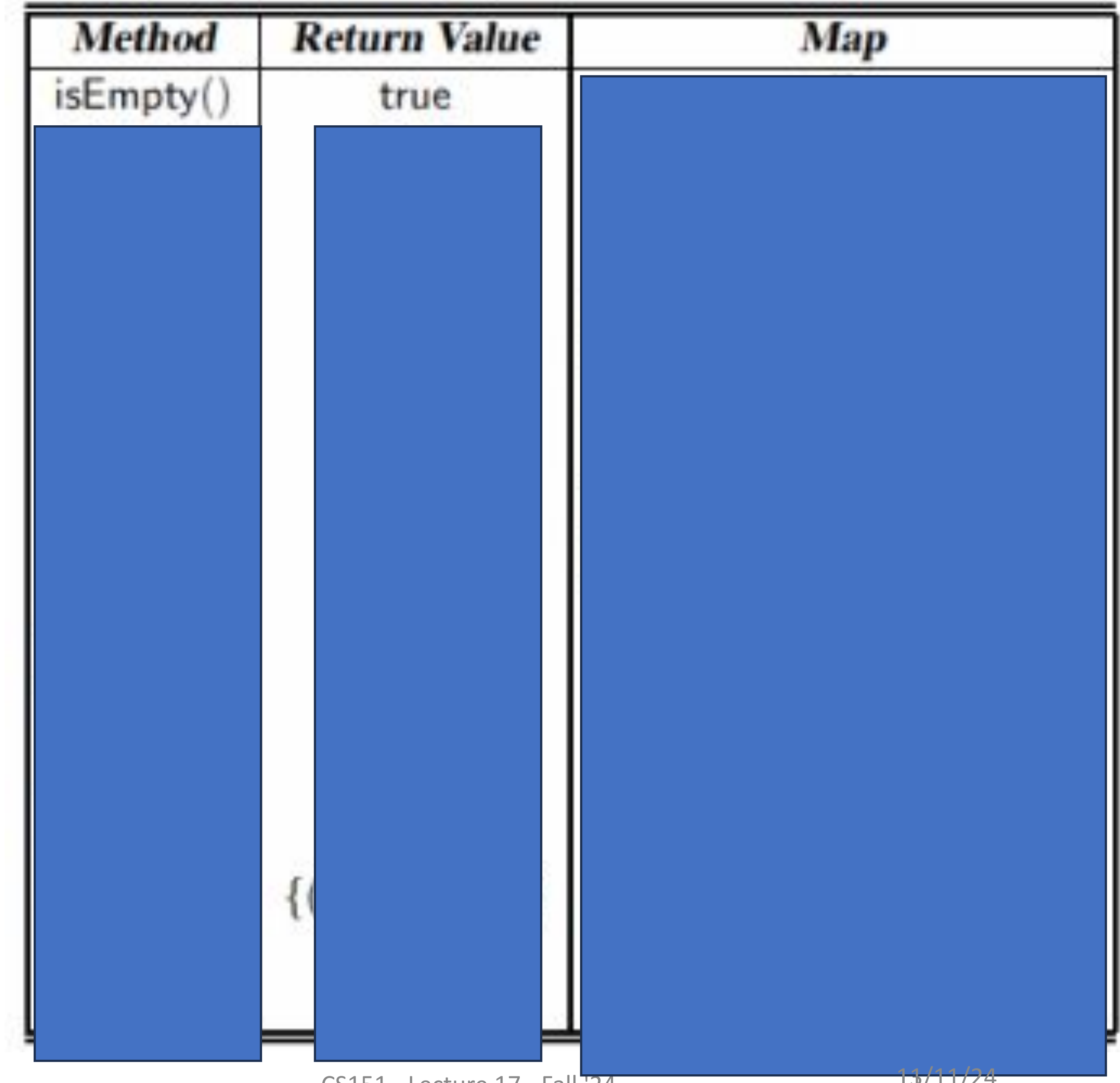
- `get(k)`: if the map `M` has an entry with key `k`, return its associated value; else, return null
- `put(k, v)`: insert entry `(k, v)` into the map `M`; if key `k` is not already in M, then return null; else, replace old value with `v` and return old value associated with `k`
- `remove(k)`: if the map `M` has an entry with key `k`, remove it from `M` and return its associated value; else, return null
- `size()`, `isEmpty()`
- `keySet()`: return an iterable collection of the keys in `M`
- `values()`: return an iterator of the values in `M`
- `entrySet()`: return an iterable collection of the entries in `M`

Example



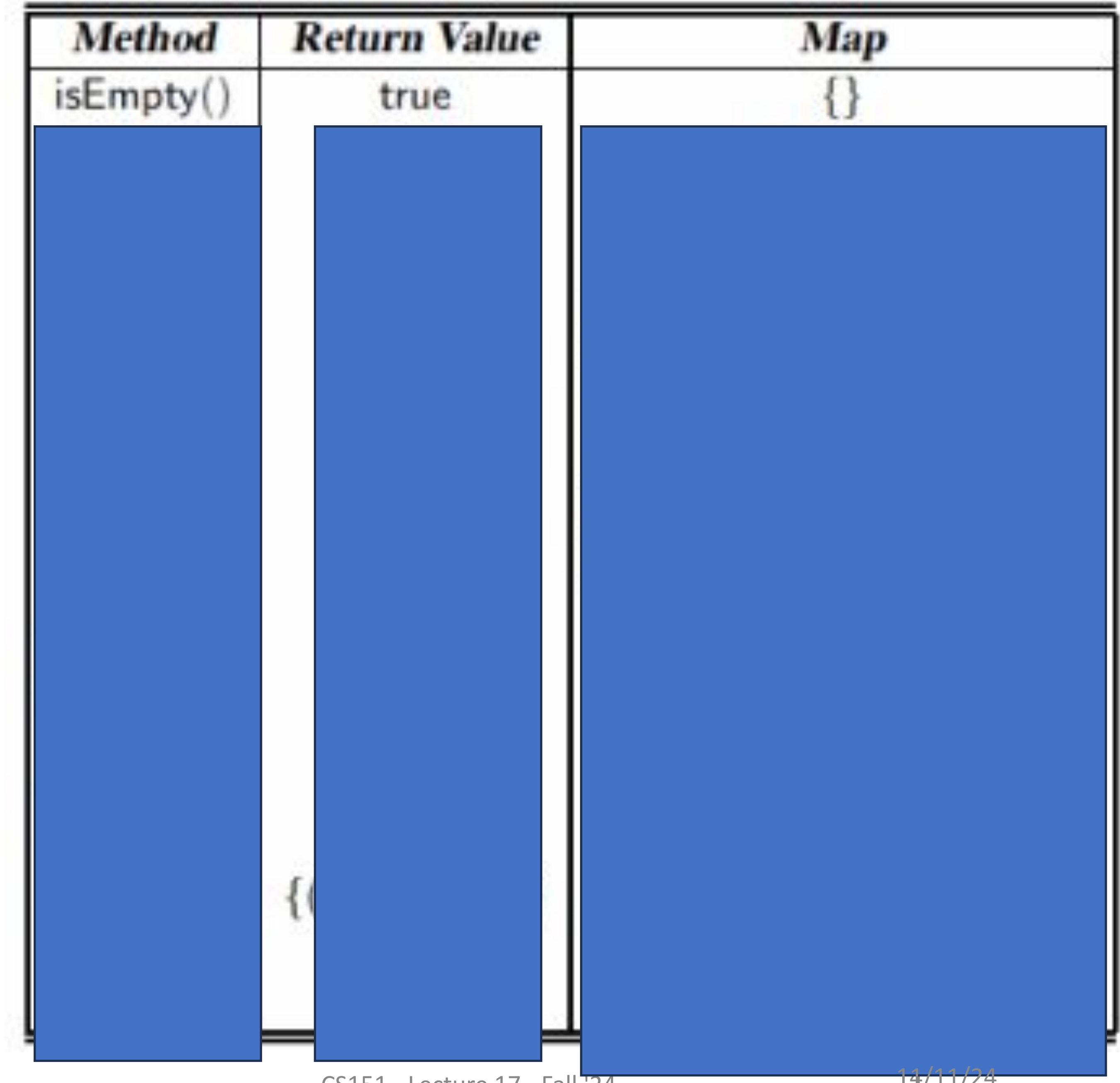
Method	Return Value	Map
isEmpty()		

Example



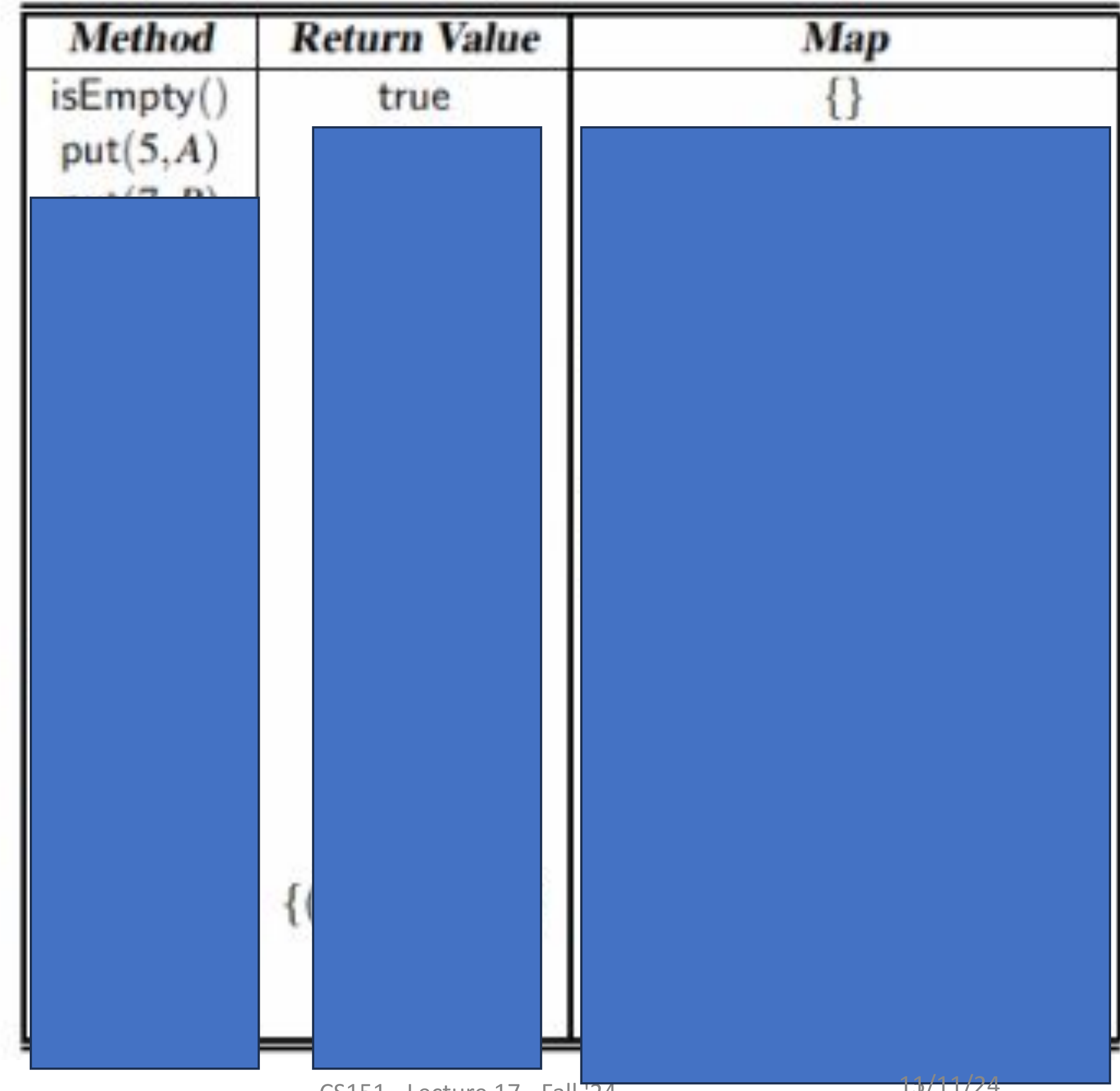
Method	Return Value	Map
isEmpty()	true	

Example



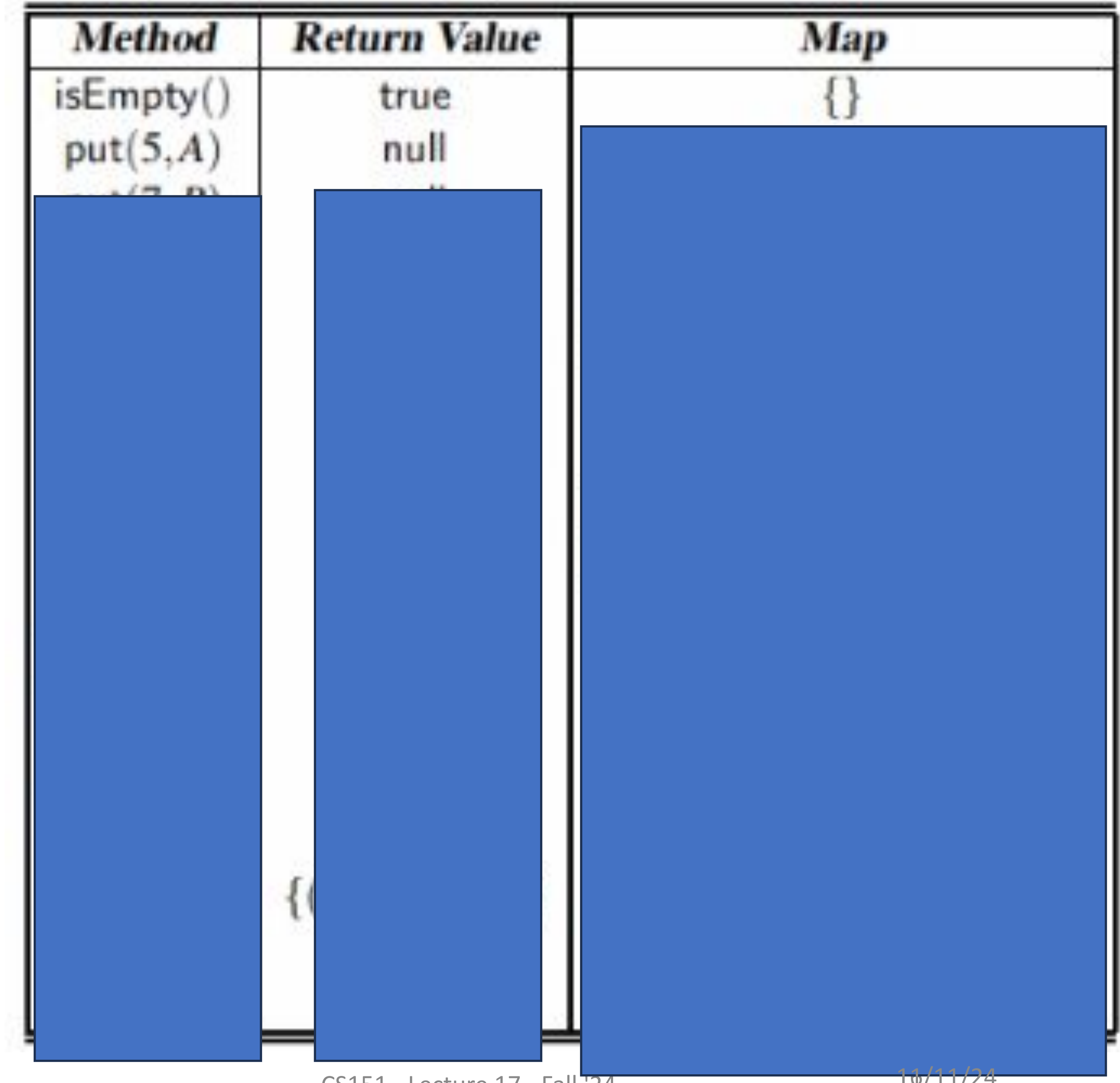
Method	Return Value	Map
isEmpty()	true	{}

Example



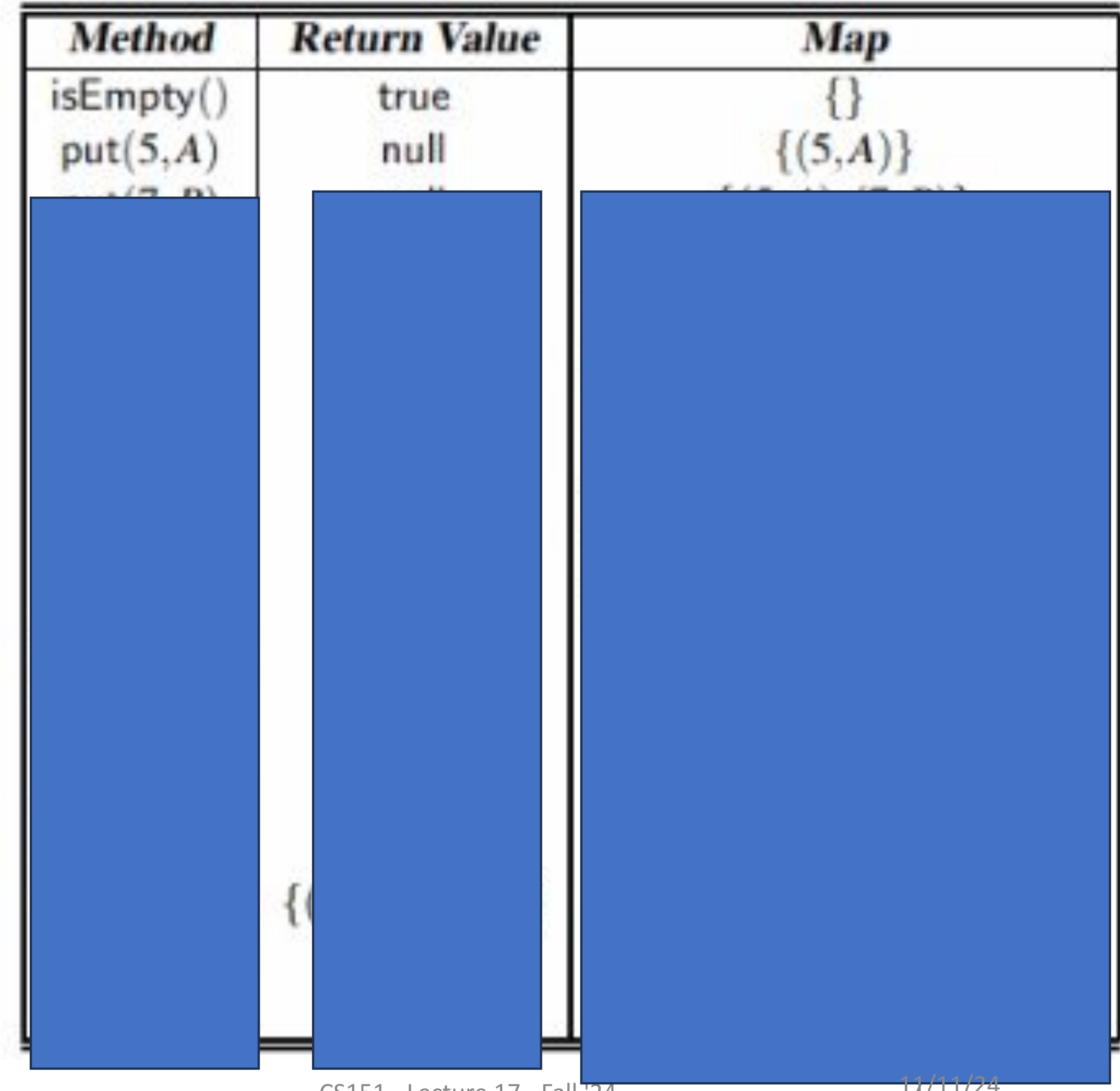
Method	Return Value	Map
isEmpty()	true	{}
put(5,A)		

Example



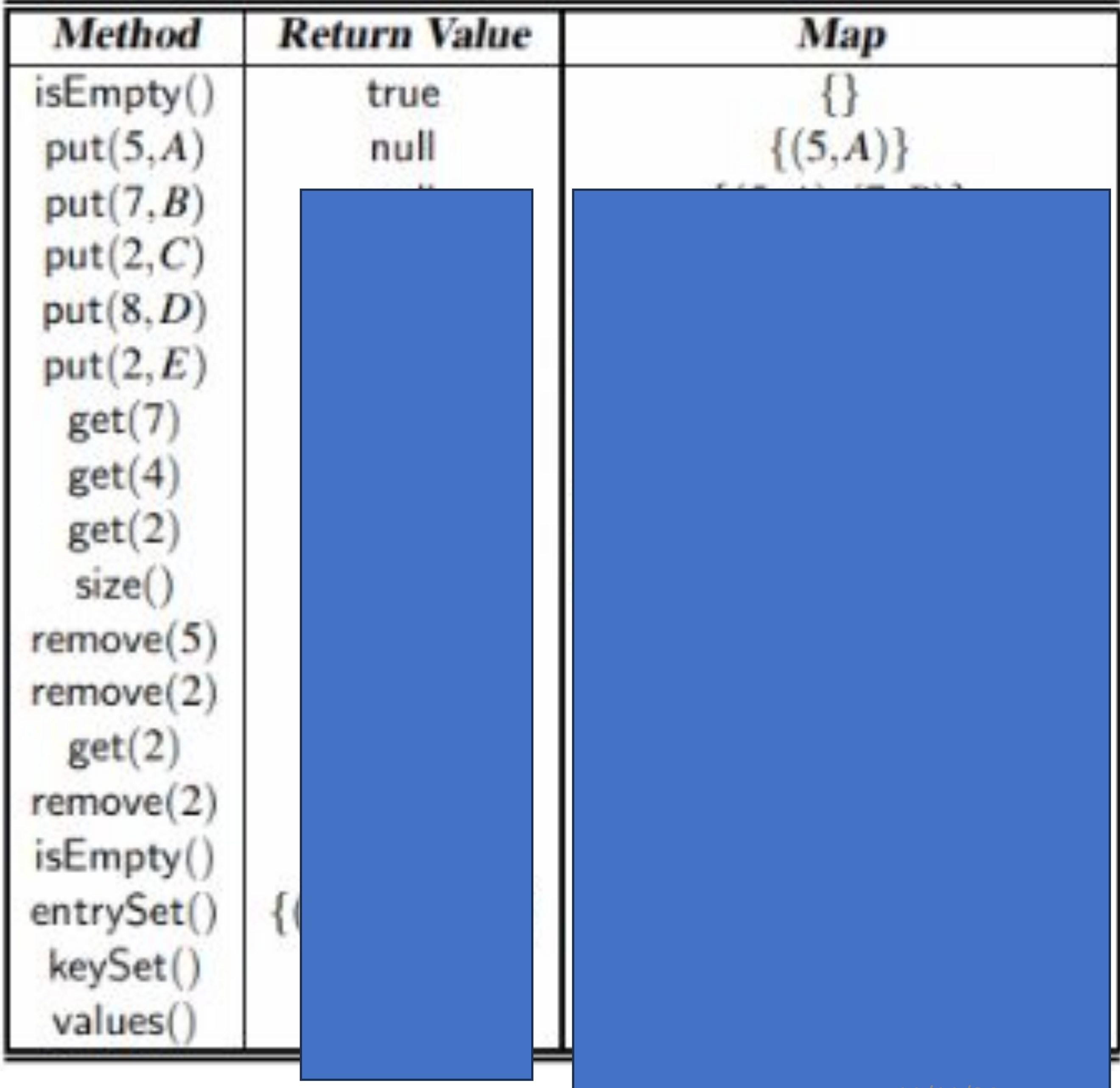
Method	Return Value	Map
isEmpty()	true	{}
put(5,A)	null	

Example



Method	Return Value	Map
isEmpty()	true	$\{\}$
put(5,A)	null	$\{(5,A)\}$

Example



Method	Return Value	Map
isEmpty()	true	{}
put(5,A)	null	{(5,A)}
put(7,B)		
put(2,C)		
put(8,D)		
put(2,E)		
get(7)		
get(4)		
get(2)		
size()		
remove(5)		
remove(2)		
get(2)		
remove(2)		
isEmpty()		
entrySet()		
keySet()		
values()		

Example

Method	Return Value	Map
isEmpty()	true	{}
put(5, A)	null	{(5, A)}
put(7, B)	null	{(5, A), (7, B)}
put(2, C)	null	{(5, A), (7, B), (2, C)}
put(8, D)	null	{(5, A), (7, B), (2, C), (8, D)}
put(2, E)	C	{(5, A), (7, B), (2, E), (8, D)}
get(7)	B	{(5, A), (7, B), (2, E), (8, D)}
get(4)	null	{(5, A), (7, B), (2, E), (8, D)}
get(2)	E	{(5, A), (7, B), (2, E), (8, D)}
size()	4	{(5, A), (7, B), (2, E), (8, D)}
remove(5)	A	{(7, B), (2, E), (8, D)}
remove(2)	E	{(7, B), (8, D)}
get(2)	null	{(7, B), (8, D)}
remove(2)	null	{(7, B), (8, D)}
isEmpty()	false	{(7, B), (8, D)}
entrySet()	{(7, B), (8, D)}	{(7, B), (8, D)}
keySet()	{7, 8}	{(7, B), (8, D)}
values()	{B, D}	{(7, B), (8, D)}

Map ADT

- Map class is abstract
- Concrete Implementations of Map:
  - `UnsortedTableMap`
  - `HashMap`

Map

- How can we implement a map?
  - Array !

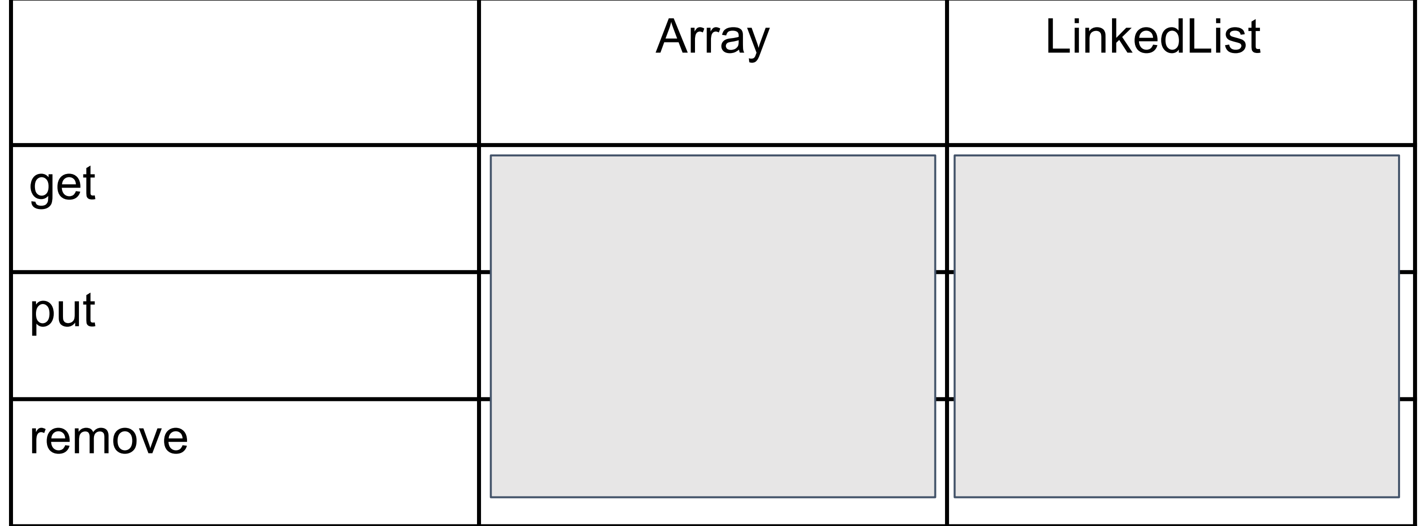
Map.Entry Interface

- A (Key, Value) pair
- Keys and Values can be any reference type
- Methods:
  - `getKey()`
  - `getValue()`
  - `setValue(V val)`
- Implementation: `SimpleEntry`

ArrayMap

Let's implement a `Map` as an array of `SimpleEntry` s

Performance Analysis



	Array	LinkedList
get		
put		
remove		

LinkedList Map

- `get(K key)`
- `put(K key, V value)`
  - If k is not in the map add it. If it is in the map, replace with the new value.
- `remove(K key)`

Performance Analysis

	Array	LinkedList
get	O(n)	
put	O(n)	
remove	O(n)	

HashMaps

Hash Functions

- A *hash function* maps an arbitrary length input to a fixed length *unique* output

- https://emn178.github.io/online-tools/sha256.html

- Applications
  - data structures
  - encryption / digital signatures
  - blockchain

- Properties of a good hash function:
  - one way
  - **collision resistant**
  - **uniformity**

Another Simple Hash Function

Given an int x...

h(x) = last 4 digits of x

- one way?
- collision resistant?
- uniform?

Another Simple Hash Function

$h(x) = x \% N$

- one way?
- collision resistant?
- uniform?

HashMaps

- How can we use hash functions to improve the performance of our ArrayMap implementation?

Summary

Maps:

- Associative DS with key-value pairs
- Can be implemented with an array or LL for O(n) operations

Hash Functions

- One way, collision resistant mathematical functions
- Maps an arbitrary length input to a fixed length *unique* output
- Can be used to implement efficient maps (next class..)

Start HW6 :)