Lecture 2

Part 2:

Unordered Data

Unordered Data

Unordered Data

Set ADT

S.Add(x)

S.Remove(x)

x in S?

Unordered Data

Set ADT

S.Add(x)

S.Remove(x)

x in S?

Key/Value Store

Hashmap

Associative Array

S.Add(K,V) S.Replace(K,V)

S.Remove(K)

x in S

value stored with key K

Unordered Data

Set ADT

S.Add(x)

S.Remove(x)

x in S?

Key/Value Store

Hashmap

Associative Array

Key/Value Store	Python Dictionary S={}
S.Add(k,v) S.Replace(k,v)	S[k]=v
S.Remove(k)	del S[k]
x in S	k in S
value stored with key k	S[k]

se

Unordered Data

Set ADT

S.Add(x)

S.Remove(x)

x in S?

Key/Value Store Hashmap Associative Array	Python Dictionaries S={}
S.Add(k,v) S.Replace(k,v)	S[k]=v
S.Remove(k)	del S[k]
x in S	k in S
value stored with key k	S[k]

Allow: Big keys, values

Unordered Data

Set ADT

S.Add(x)

S.Remove(x)

x in S?

Key/Value Store

Hashmap

Associative Array

	Python Dictionary S={}
S.Add(k,v) S.Replace(k,v)	S[k]=v
S.Remove(k)	del S[k]
x in S	k in S
value stored with key k	S[k]

Allow: Big keys, values

Don't Allow: Any other queries, eg proximity, range

Today's Plan

Today's Plan

- Hash Tables

  General-purpose way to store unordered data

Today's Plan

- Hash Tables

  General-purpose way to store unordered data

- Bit Vectors

  Compact way to store an array of 0/1 values

Today's Plan

- Hash Tables

  General-purpose way to store unordered data

- Bit Vectors

  Compact way to store an array of 0/1 values

- Bloom Filters

  Compact way of storing General-purpose data at the cost of allowing errors

Hash Tables

Hash Tables

- Support all operations in O(1) time (expected)

Hash Tables

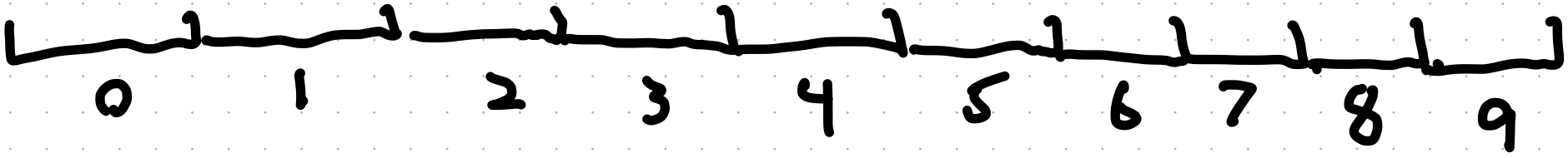
- Support all operations in O(1) time (expected)
- Use Hash Functions

Hash Tables

- Support all operations in O(1) time (expected)
- Use Hash Functions
- The best structure to use in many situations

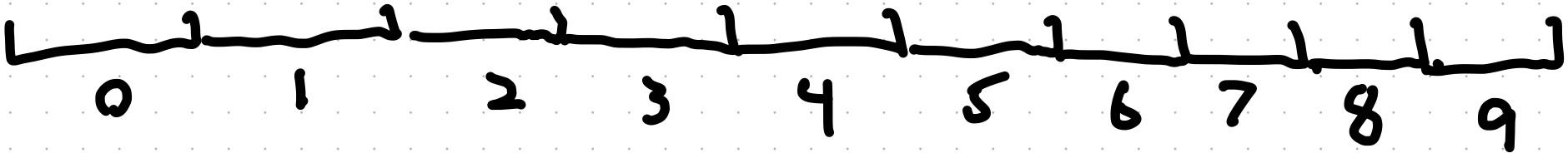
Hash Tables: Basic idea

Hash Tables: Basic idea



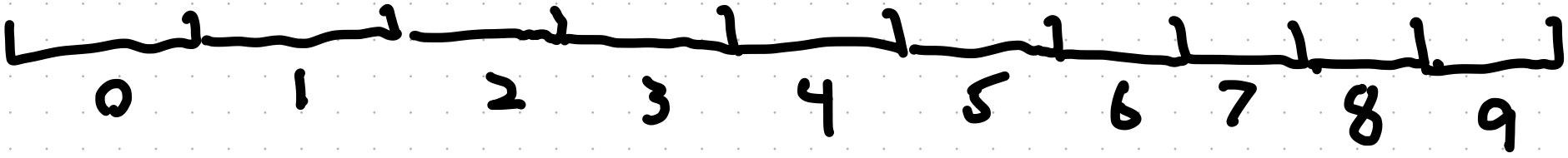
- Build a table with b buckets

Hash Tables: Basic idea



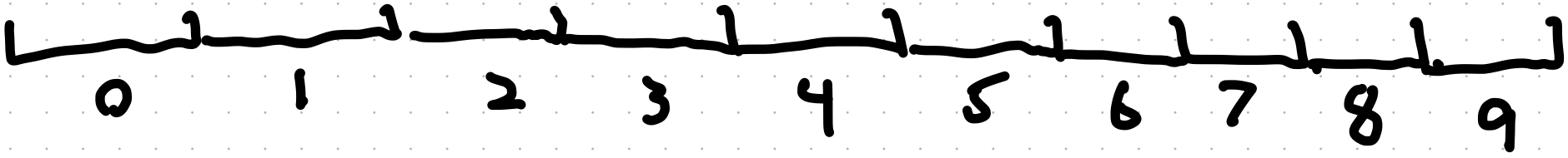
- Build a table with b buckets
- To insert K,V put it in bucket hash(K) mod 10

Hash Tables: Basic idea



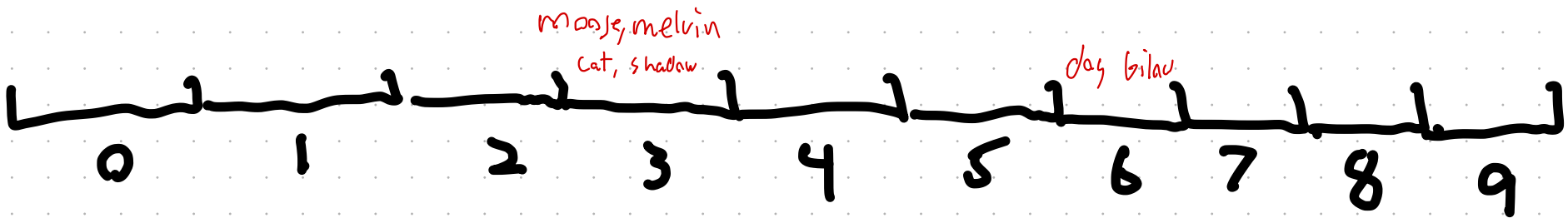
- Build a table with b buckets
- To insert K,V put it in bucket hash(K) mod 10
- To search for K, Look in bucket hash(K) mod 10

Hash Tables: Basic idea



- Build a table with b buckets
- To insert K,V put it in bucket hash(K) mod 10
- To search for K, Look in bucket hash(K) mod 10
- We assume hash functions generate uniformly distributed values, thus the expected size of a bucket is $O(\frac{n}{b})$

Hash Tables: Basic idea



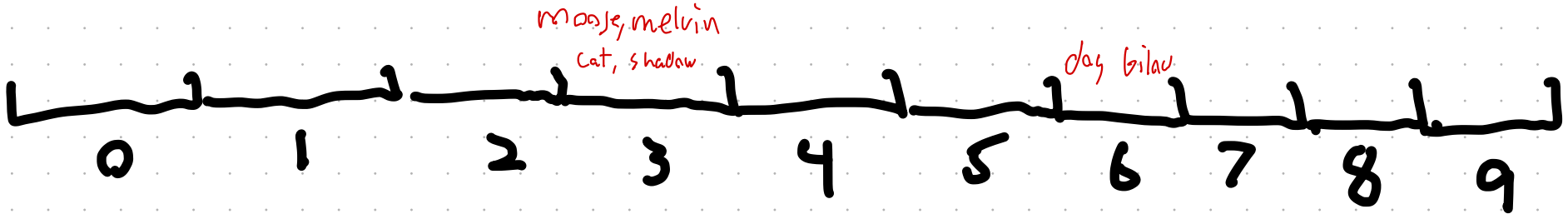
- Build a table with b buckets
- To insert K,V put it in bucket hash(K) mod 10
- To search for K, Look in bucket hash(K) mod 10
- We assume hash functions generate uniformly distributed values, thus the expected size of a bucket is $O(\frac{n}{b})$

hash("cat") = 123 hash("moose") = 313
hash("dog") = 456

H["cat"] = "shadow"
H["dog"] = "bilou"
H["moose"] = "melvin"

Hash Tables: Basic idea

Data structure:
Table: list
Bucket: list
K,V pair: list



- Build a table with b buckets
- To insert K,V put it in bucket hash(K) mod 10
- To search for K, Look in bucket hash(K) mod 10
- We assume hash functions generate uniformly distributed values, thus the expected size of a bucket is $O(\frac{n}{b})$

hash("cat") = 123 hash("moose") = 313
hash("dog") = 456

H["cat"] = "shadow"
H["dog"] = "bilou"
H["moose"] = "melvin"

Hash Tables: Coding

Hash Tables: Coding

Class called HashTable, works like a Dictionary

Hash Tables: Coding

Class called HashTable, works like a Dictionary

- A will store the table (A list of lists of lists)

Hash Tables: Coding

Class called HashTable, works like a Dictionary

- A will store the table (A list of lists of lists)

__init__ will be given the size of the table

Hash Tables: Coding

Class called HashTable, works like a Dictionary

- A will store the table (A list of lists of lists)

__init__ will be given the size of the table

__setitem__ H[K]=V

Hash Tables: Coding

Class called HashTable, works like a Dictionary

- self.A will store the table (A list of lists of lists)

__init__ will be given the size of the table

__setitem__ H[K]=V

__getitem__ H[K]

Hash Tables: Coding

Class called HashTable, works like a Dictionary

- A will store the table (A list of lists of lists)

__init__ will be given the size of the table

__setitem__ H[K]=V

__getitem__ H[K]

__delete__ del H[K]

Hash Tables: Coding

Class called HashTable, works like a Dictionary

- A will store the table (A list of lists of lists)

__init__ will be given the size of the table

__setitem__ H[K]=V

__getitem__ H[K]

__delete__ del H[K]

__contains__ K in H

Hash Tables: Coding

Class called HashTable, works like a Dictionary

- A will store the table (A list of lists of lists)

__init__ will be given the size of the table

__setitem__ H[K]=V

__getitem__ H[K]

__delete__ del H[K]

__contains__ K in H

__str__ str(H) shows the contents

Hash Tables: Coding

Class called HashTable, works like a Dictionary

- A will store the table (A list of lists of lists)

__init__ will be given the size of the table

__setitem__	H[K]=V
__getitem__	H[K]
__delete__	del H[K]
__contains__	K in H
__str__	str(H) shows the contents
__repr__	repr(H) shows the structure

Hash Tables: Coding

Class called HashTable, works like a Dictionary

- self.A will store the table (A list of lists of lists)

__init__ will be given the size of the table

__setitem__	H[K]=V
__getitem__	H[K]
__delete__	del H[K]
__contains__	K in H
__str__	str(H) shows the contents
__repr__	repr(H) shows the structure
_bucket	A helper method to determine which bucket a key belongs to

Code : __Contains__

k in H H.__contains__(k)

```
def __contains__(self, key)
    for k,v in self._A[self._bucket(key)]
        if k == key
            return True
    return False
```

Our Code vs Python Dictionaries

Our Code vs Python Dictionaries

- Python does not require initialization with the table size

Our Code vs Python Dictionaries

- Python does not require initialization with the table size
- Python will rebuild the table to keep $\frac{n}{b} = O(1)$ and thus keep the runtime $O(1)$

Our Code vs Python Dictionaries

- Python does not require initialization with the table size
- Python will rebuild the table to keep $\frac{n}{b} = O(1)$ and thus keep the runtime $O(1)$
- Python supports more operations
  e.g. `for x in H:` (iterator)

Why did I show you how to do something Python does already?

Why did I show you how to do something Python does already?

- Helps you understand how python works

Why did I show you how to do something Python does already?

- Helps you understand how Python works
- Shows you the power of hacking

Why did I show you how to do something Python does already?

- Helps you understand how python works
- Shows you the power of hashing
- Motivates what will come next