# Algorithms and Data Structures

## Hashing (Ch. 5)

Morgan Ericsson

# Today

- Hashing
  - Seperate chaining
  - Probing
  - Double hashing
  - Rehashing

# Hashing

# Question

- Assume we have a list of integers
- We can find a given integer
  - $O(n)$ if we use sequential search
  - $O(\log_2 n)$ if we use a binary search tree
- Can we do better?
  - Think back to union find/disjoint set

# Binary list

- Let’s try to adopt the idea from union find
  - But now we only care if a number is in the list or not
- So, if we want to insert a number
  - `lst[6433] = True`
- If we want to check
  - `lst[2137] is not None`
- Reasonable idea, but how larger should the list be?
- How much space is wasted?

# Flawed but good idea

- We use the integer value to map a key to a position
- The problem is the range of the random integers
- If we wanted to store all numbers 1 to 100 it would not be a problem
- But in the general case, the numbers might be to large and the list would be very sparse

# Flawed but good idea

```python
import numpy as np

lst = np.zeros(7630+1, dtype=bool)

for v in [7630, 3275, 6433, 5913, 2137]:
    lst[v] = True

print(len(lst[lst == True]))
print(len(lst[lst == False]))
```

```
5
7626
```

# General idea

- We use the key to map to an element
- If unlimited space, we can just use the value
- If less then limited space, we need a “better” mapping function
  - What if we use %?

# Another way to map

```
lst_sz = 5
lst = np.zeros(lst_sz, dtype=bool)

for v in [7630, 3275, 6433, 5913, 2137]:
    lst[v % lst_sz] = True

print(len(lst[lst == True]))
print(len(lst[lst == False]))
```

```
3
2
```

# Hash

- The mapping is called a hashing function or hash(Code)
- The problem we observed is a collison
  - When two (or more) keys map to the same position
- A perfect hashing function should never produce collisions
  - but it can be difficult to define
- Especially since it should also be efficient to compute

# Another try

```
lst_sz = 31
lst = np.zeros(lst_sz, dtype=bool)

def hashf(v):
    return v % lst_sz

for v in [7630, 3275, 6433, 5913, 2137]:
    lst[hashf(v)] = True

print(len(lst[lst == True]))
```

5

# More problems

- Without perfect hashing, we can never find the key
  - Hashing is a one-way function
  - `99 % 10 == 9 % 10`
- There is no order
- What if they keys are not integers?

# Another example

```python
lst_sz = 31
lst = np.zeros(lst_sz, dtype=bool)

def hashf(v:str) -> int:
    pass

for v in ['Liam', 'Olivia', 'Charlotte', 'Lucas', 'Mia']:
    lst[hashf(v)] = True
```

# Options

- Use the first (or last) $k$ letters
- Can be a bad idea, many domains have common pre- and suffixes
  - Names
  - Phone numbers
  - ...
- Use the whole key?

# Example

```python
def hashf(key:str) -> int:
  hv = 0
  for c in key:
    hv += ord(c)

  return hv
```

# Example

```
lst_sz = 31
lst = np.zeros(lst_sz, dtype=bool)

for v in ['Liam', 'Olivia', 'Charlotte', 'Lucas', 'Mia']:
    lst[hashf(v) % lst_sz] = True

print(len(lst[lst == True]))
```

# All is good?

```
print(hashf('abc') == hashf('acb'))
```

```
True
```

> Problem? Depends on our use... Not great for phone numbers, for example...

# What can we do?

```python
def hashf(key:str) -> int:
  hv = 0
  for ix,c in enumerate(key, start=2):
    hv += ix * ord(c)

  return hv
```

# Solved?

```python
print(hashf('abc') == hashf('acb'))
print(hashf('abc') == hashf('-10['))
```

```
False
True
```

> It is not easy to create a perfect (good) hash function.

# Keep in mind

- Try to avoid repetition and “round” numbers
  - It is generally a bad idea to use sizes of even 10s
  - Or powers of two
  - Use prime numbers to break patterns
  - Preferably close to powers of two
- Use as much of the key as possible
  - More bits means more variation

# Hashing in Python (and Java)

- Types define a hash value
  - `hash('Olivia')`
  - `hash((1, 3))`
- Similar in Java
- Required for certain things to work, e.g., sets

# Simple example

```python
class Person:
  def __init__(self, n:str, a:int) -> None:
    self.name = n
    self.age = a

  def __str__(self) -> str:
    return f'{self.name} ({self.age})'

p1 = Person('Olivia', 34)
print(hash(p1))
```

```
8785737328619
```

# For free?

```python
p1 = Person('Olivia', 34)
p2 = Person('Olivia', 34)

print(hash(p1) == hash(p2))
```

```
False
```

> Hash is based on object identity. Problem? Depends on our use...

# For free?

```python
from fastcore.basics import patch

@patch
def __hash__(self:Person) -> int:
    hv = 17
    hv = 31 * hv + hash(self.name)
    hv = 31 * hv + hash(self.age)
    return hv
```

> We can define our own hash function

# New try

```
p1 = Person('Olivia', 34)
p2 = Person('Olivia', 34)

print(hash(p1) == hash(p2))
```

```
True
```

> Based on object values rather than identify

# Using our function

```
plst = [None] * 31

p1 = Person('Olivia', 34)
p2 = Person('Mia', 11)

plst[hash(p1) % 31] = p1
plst[hash(p2) % 31] = p2

print(plst[hash(p1) % 31])
```

```
Olivia (34)
```

# Suppose Olivia had a birthday ...

```
p1.age += 1

print(plst[hash(p1) % 31])
```

```
None
```

> Custom hash functions and mutability is a problem...

# Simple hash table

```
class HT:
  def __init__(self):
    self.sz = 31
    self.table = [None] * 31

  def insert(self, key):
    self.table[hash(key) % self.sz] = key

  def contains(self, key):
    return self.table[hash(key) % self.sz] == key

  def __len__(self):
    return len([v for v in self.table \
                if v is not None])
```

# Testing it

```
import random

ht = HT()
for i in range(10):
    v = random.randint(1, 100_000)
    ht.insert(v)

print(len(ht))
```

9

# Seperate chaining

# Collisions

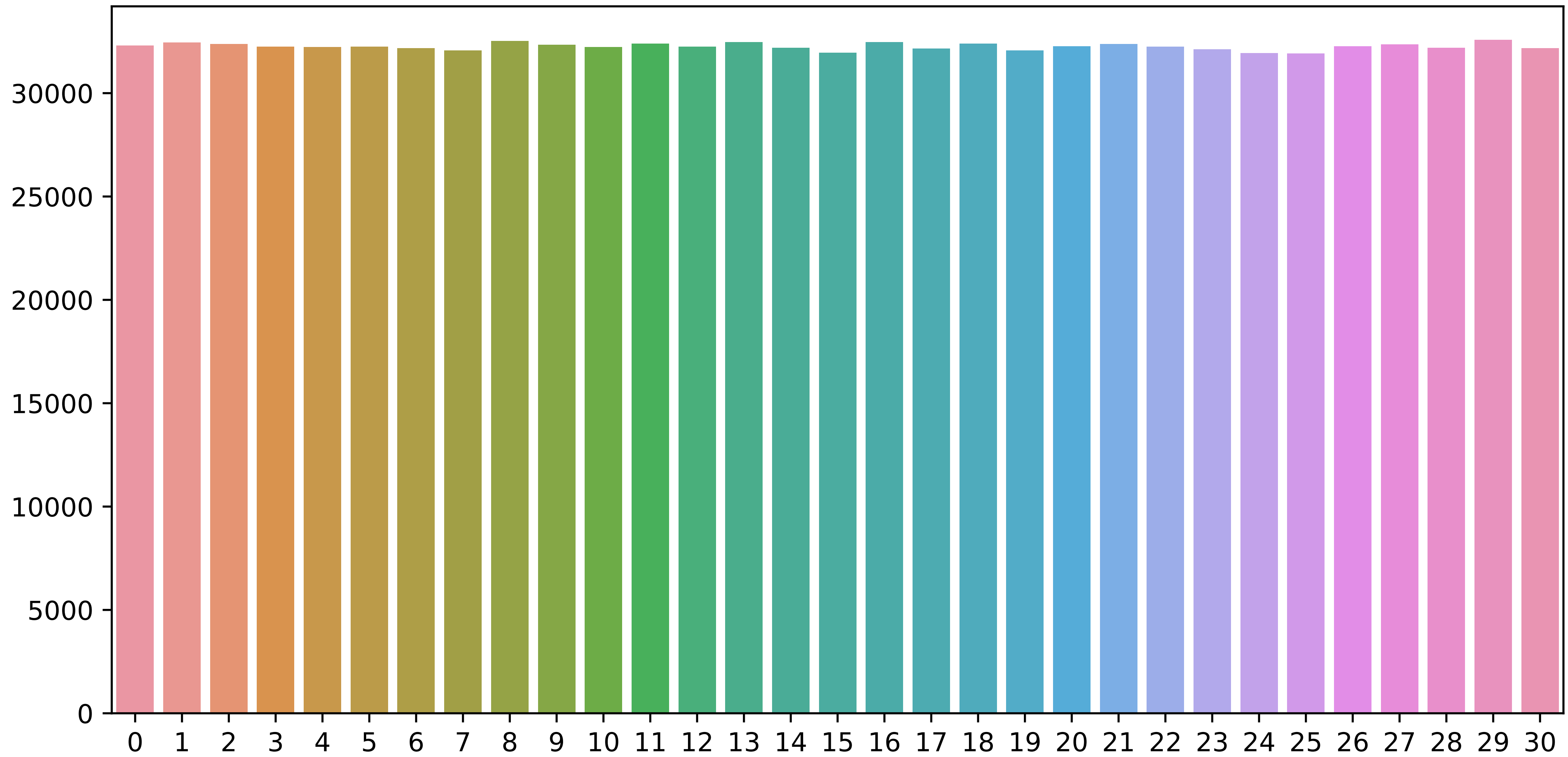
- We have seen that some hashing functions can result in collisions
- Can we manage the collisions?

# Hash functions

- We want the hash functions to distribute keys uniformely across the integer interval
- Bins and balls
  - Assume that each key is a ball and each position is a bin
  - If we randomly toss a random ball, it should be equally likely to end up in any of the bins
  - If have $m$ bins and toss $n$ balls, we would expect there to be $n/m$ balls in each bin after a while

# Uniformity

## Tossing 1 000 000 balls into 31 bins



# We also know

- We can expect two balls in the same bin after $\sim \sqrt{\pi \frac{m}{2}}$ tosses
- Every bin has $\geq 1$ balls after $\sim m \ln m$ tosses
- After $m$ tosses, the most loaded bin has $\Theta(\frac{\log m}{\log \log m})$ balls

# So, how can we deal with collisions?

- Seperate chaining
- We make each bin a linked list
- And place keys that collide in the same bin

# A simple linked list

```python
from dataclasses import dataclass

@dataclass
class LLNode:
  key: int
  nxt: 'LLNode|None' = None
```

# Seperate chaining

```
class HTSC:
  def __init__(self, m=5):
    self.sz = m
    self.table = [None] * self.sz
```

# Inserting

```
@patch
def insert(self:HTSC, key):
  hv = hash(key) % self.sz
  self.table[hv] = self._inschain(self.table[hv], key)

@patch
def _inschain(self:HTSC, n:LLNode|None, key) -> LLNode:
  if n is None:
    return LLNode(key)
  n.nxt = self._inschain(n.nxt, key)
  return n
```

# Finding

```
@patch
def find(self:HTSC, key) -> bool:
  hv = hash(key) % self.sz
  if self.table[hv] is not None:
    p = self.table[hv]
    while p is not None:
      if p.key == key:
        return True
      p = p.nxt
  return False
```

# Len

```python
@patch
def __len__(self:HTSC) -> int:
  l = 0
  for t in self.table:
    l += self._lchain(t)
  return l

@patch
def _lchain(self:HTSC, n:LLNode|None) -> int:
  l = 0
  while n is not None:
    l += 1
    n = n.nxt
  return l
```

# Testing it

```
ht = HTSC()

for v in ['Liam', 'Olivia', 'Charlotte', 'Lucas', 'Mia']:
  ht.insert(v)

#print(len(ht))
print(len(ht))
```

5

# Testing it some more

```python
ht = HTSC()
for i in range(200):
    v = random.randint(1, 100_000)
    ht.insert(v)

print(len(ht))
```

```
200
```

# Linear again?

- With seperate chaining, we need to search the list to determine if the value exists or not
- We know that each list holds on average $n$ / $m$
  - Where $n$ is the number of keys and $m$ is the size
- So, can be significant better than than $O(n)$

# How should we pick *m*?

- If $m$ is too small, the lists will be too long
- If $m$ is too large, we will waste space
- A good rule of thumb is to set $m$ to $n$ / 5
    - Then access will be $O(1)$

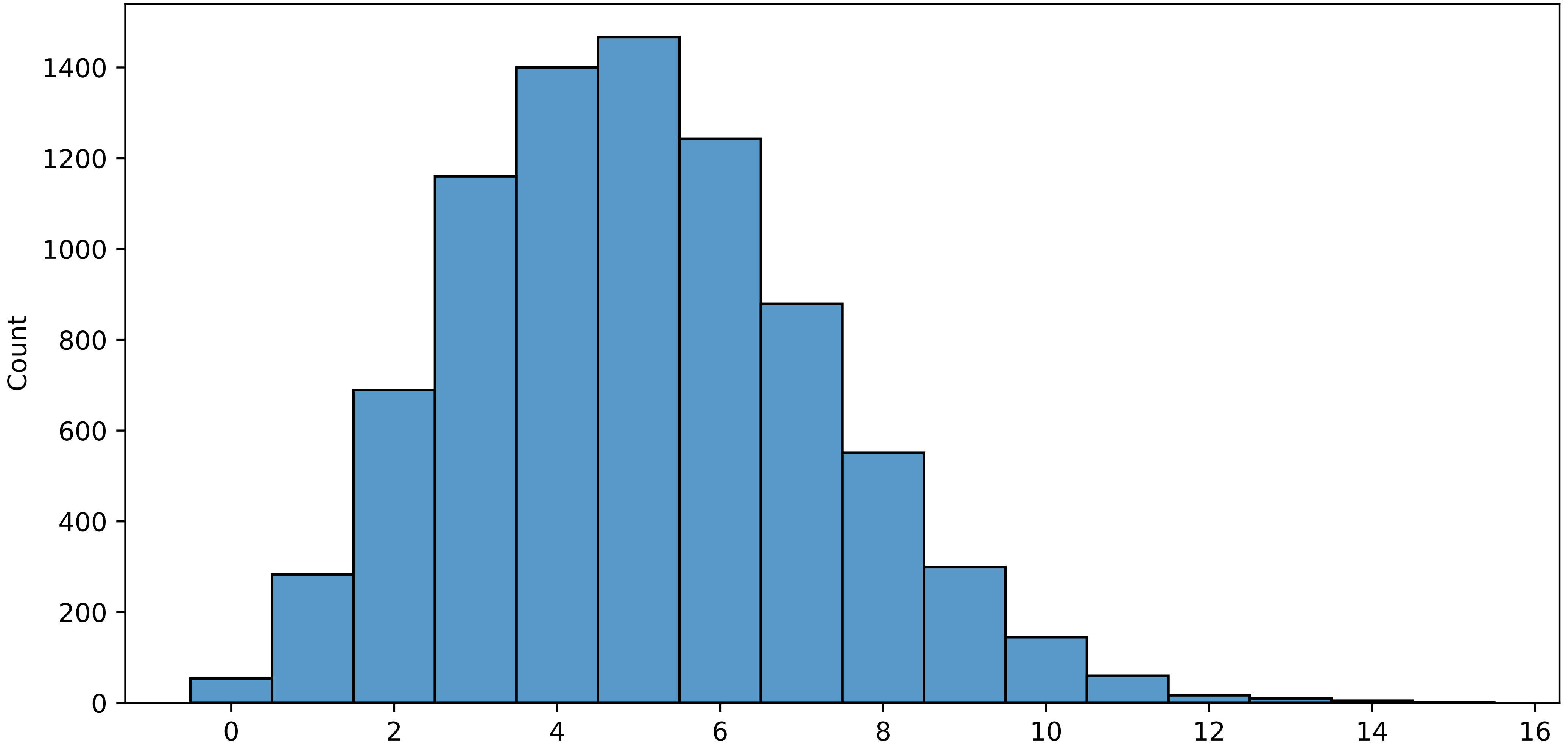
# Testing the idea

```
n = 8263 * 5
m = 8263

ht = HTSC(m)
for i in range(n):
    v = random.uniform(0, 1)
    ht.insert(v)

ll = [ht._lchain(n) for n in ht.table]
```

# Testing the idea



# Linear probing

# Linear probing

- Seperate chaining works, but:
  - Introduces a second data structure
  - Has overhead in creating nodes
- What if we “chain” in the existing list?

# Linear probing

- If a slot is taken, find the next empty one
  - If $hash(v) = i$ and $i$ is taken, try $i + 1$, $i + 2$, ... until an empty slot is found
- Must repeat the same when searching...
- The list must be larger than the number of keys

# Linear probing

```python
class HTLP:
  def __init__(self, m=5):
    self.sz = m
    self.table = [None] * self.sz
```

# Inserting

```
@patch
def insert(self:HTLP, key):
  hv = hash(key) % self.sz
  if self.table[hv] is None:
    self.table[hv] = key
  else:
    while self.table[hv] is not None:
      hv = (hv + 1) % self.sz
    self.table[hv] = key
```

# Finding

```
@patch
def find(self:HTLP, key) -> bool:
  hv = hash(key) % self.sz
  while self.table[hv] is not None:
    if self.table[hv] == key:
      return True
    hv = (hv + 1) % self.sz
  return False
```

# Len

```python
@patch
def __len__(self:HTLP) -> int:
  return len([v for v in self.table \
                  if v is not None])
```

# Testing it...

```
ht = HTLP(7)

for v in ['Liam', 'Olivia', 'Charlotte', 'Lucas', 'Mia']:
  ht.insert(v)

assert ht.find('Liam') == True
assert ht.find('John') == False
print(len(ht))
```

```
5
```

# Knuth's parking problem

- Cars arrive at a (one-way) street with $m$ parking spaces
- Each car desires a specific space $i$, but will try $i + 1$, $i + 2$, ... if $i$ is taken
- What is the average displacement?
  - with $m$ / $2$ cars, $\sim 3 / 2$
  - with $m$ cars, $\sim \sqrt{\pi\, m / 8}$

# Analysis of linear probing

- Assume we have a list of size $m$ and $n = \alpha m$ keys
- We can then determine the average number of probes if we have a search hit

$$\frac{1}{2}(1 + \frac{1}{1-\alpha})$$

# Analysis of linear probing

» And if we miss/insert

$$\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$$

# Analysis of linear probing

- If $m$ is too large, too much wasted space
- If $m$ is too small, search time blows up
- Rule of thumb, $\alpha = n / m \sim 1 / 2$
  - Probes for hit is about $3 / 2$
  - Probes for miss/insert is about $5 / 2$

# Testing it

```python
@patch
def insert(self:HTLP, key) -> int:
  hv = hash(key) % self.sz
  if self.table[hv] is None:
    self.table[hv] = key
    return 0
  else:
    off = 0
    while self.table[hv] is not None:
      hv = (hv + 1) % self.sz
      off += 1
    self.table[hv] = key
    return off
```

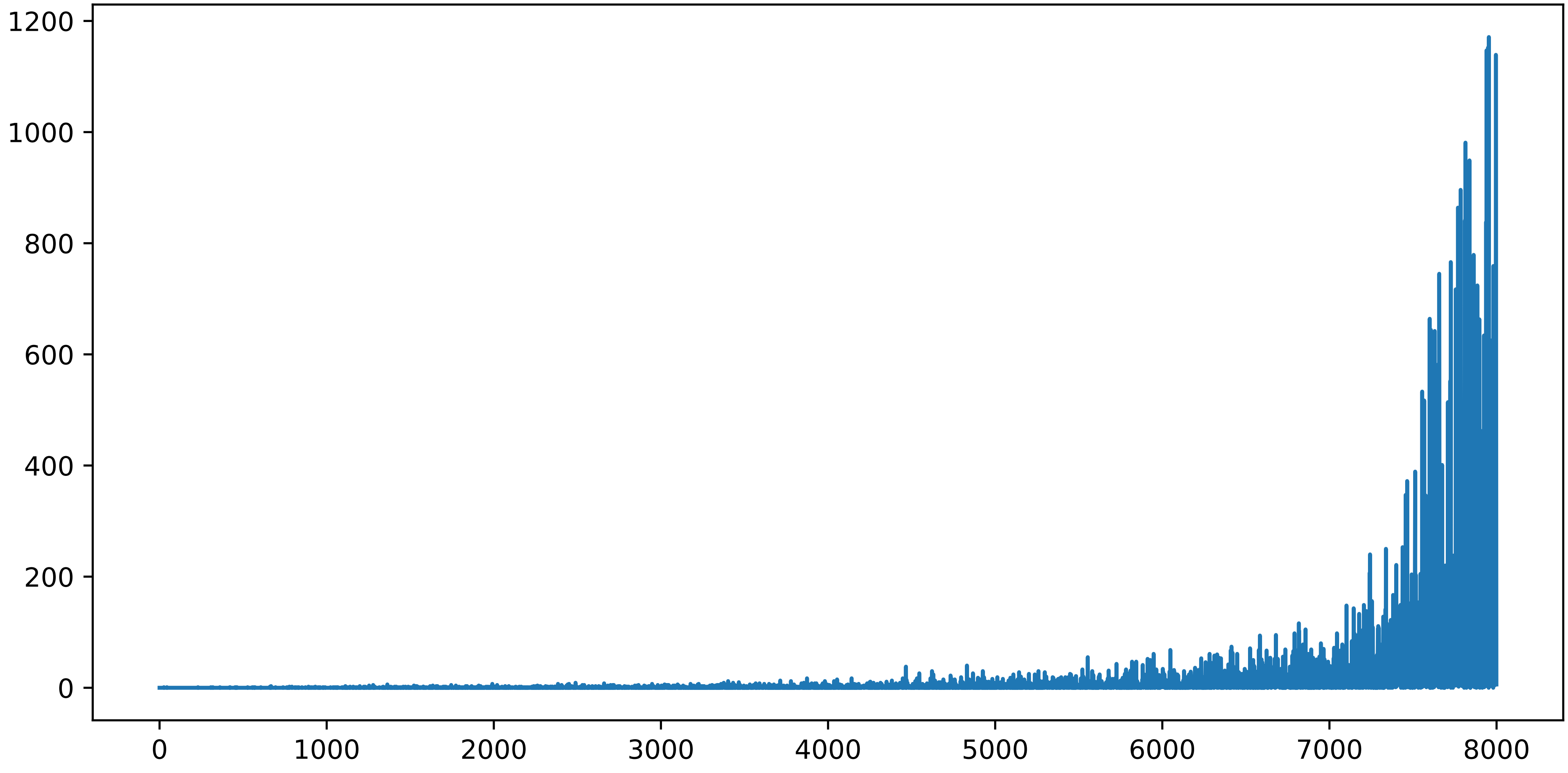
# Testing it

```
ht = HTLP(7)

for v in ['Liam', 'Olivia', 'Charlotte', 'Lucas', 'Mia']:
  print(ht.insert(v))
```

0
0
2
1
3

# Testing it some more



# So far

- We can manage collisions with seperate chaining or linear probing
- Within constant time on average
- But logarithmic time worst case
- Assuming uniform hasing

# We can do more

» Quadratic probing

» Double hashing

» ...

# Reading instructions

# Reading instructions

- Ch. 5.1 - 5.5
- Interesting, but not required
  - 5.6 discusses hashing in Java
  - 5.7 discusses more advanced versions of hasing
  - 5.8 discusses universal hash functions
  - 5.9 discusses hashing to secondary storage