Lecture 11 - Hashing, Hash Tables and Map (ADT)

Computer Systems, Data Structures, and Data Management (4CM508)

Dr Sam O'Neill

We will not do a detailed analysis on this topic, if you are interested there is plenty online or can look at Introduction to Algorithms (See References at the end of slides)

Sorting and Searching

We have already seen that searching is expensive. We will also see that sorting is expensive too.

Using an array or linked list requires searching and/or sorting to retrieve items based on a value.

Alternative - Don't Sort or Search!

- One option is to not spend time searching or sorting.
- Provide a key that gives access to some data.

Example

Ignoring Google Maps etc...

- **Address without a postcode** - You have to do some searching on a map.
- **With postcode** - You can find it easier.
- **Latitude and longitude** - You can find it quickly and precisely.

Map (ADT)

- An abstract data type (ADT) that stores key-value `(k,v)` pairs
- No duplicate keys

Name	Description
`create(X)`	create a map from a sequence `X` of key-value pairs `(k,v)`
`size()`	return the size of the map
`get(k)`	return the entry stored with key `k`
`put(k,v)`	add `v` to the map, stored with key `k` if key already exists replace item with `v`
`remove(k)`	delete the item stored at key `k` and delete the key `k`

Python Dictionaries

Python dictionaries are examples of the map (ADT).

```python
student_dict = {
    12345: "Chris Windmall",
    54321: "Patrick Marritt"
  }  # Create(X) - this is one line of code

student_dict[12345]  # get(k) - here get(12345), returns "Chris Windmall"

student_dict[32145] = "Sam O'Neill"  # put(k,v) - here put(32145, "Sam O'Neill")

del student_dict[12345] # remove(k) - here remove(12345)
```

Also known as associative array, hashmap and symbol table depending on language.

C# Dictionaries

C# dictionaries are examples of the map (ADT).

```
Dictionary studentDict = <int, string>{
    {12345: "Chris Windmall"},
    {54321: "Patrick Marritt"}
  }  // Create(X) - this is one line of code

studentDict[12345]  // get(k) - here get(12345), returns "Chris Windmall"

studentDict[32145] = "Sam O'Neill"  // put(k,v) - here put(32145, "Sam O'Neill")

student_dict.remove(12345) // remove(k) - here remove(12345)
```

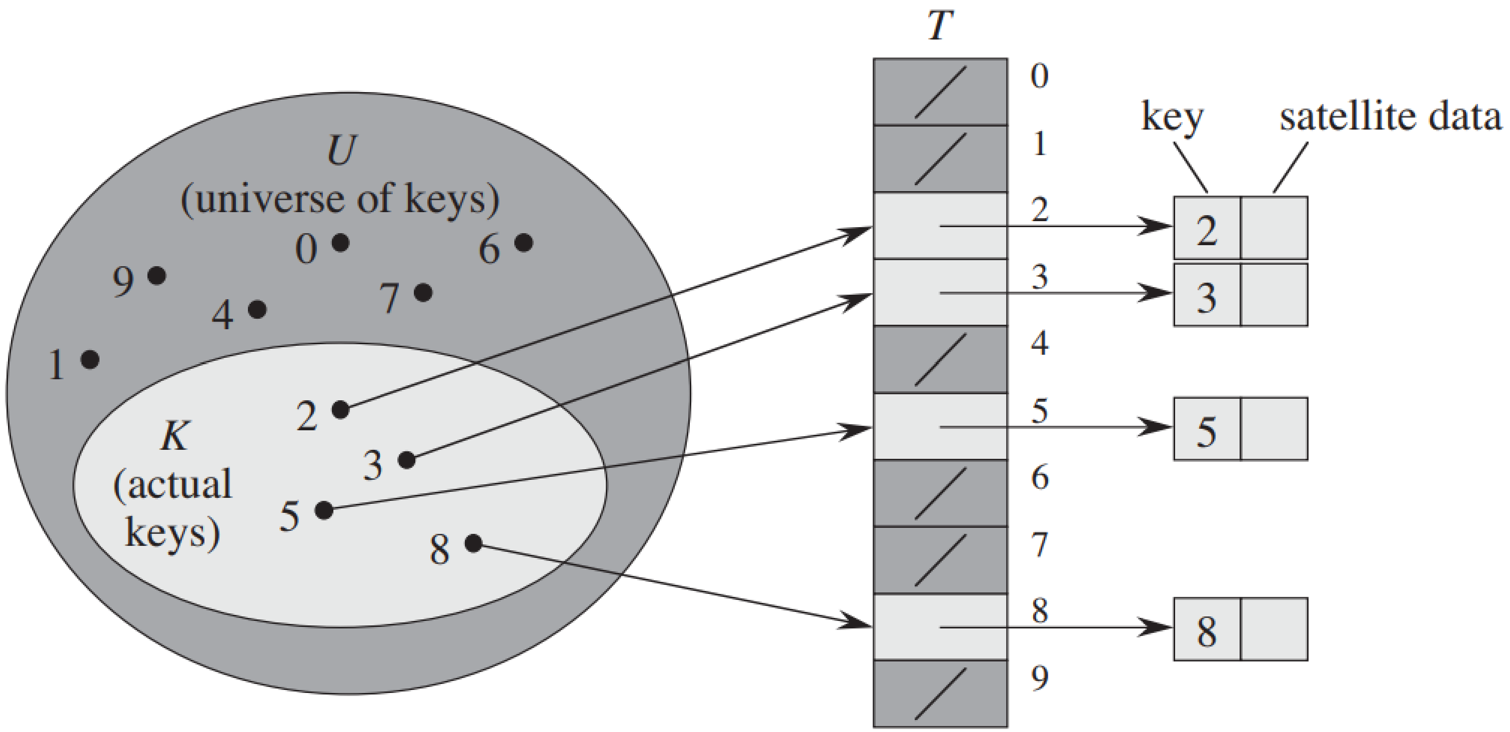
How? Storing Values with Keys

We need some way of storing a value with a key.

How could we implement this?

Direct Addressing

We can simply use an array. What is the problem here?



Issues with Direct Addressing

- Do we know all the keys?
  - Unlikely unless it is a set of very particular circumstances
- Are the keys of the same type? e.g. Integer, String, Boolean
  - Perhaps, and you could easily enforce this. In fact, that is what C# does. Python doesn't!
- Is the set of all keys finite?
  - Probably not, we might be using co-ordinates, names, set of integers

Thus we need a way of storing an unknown size of keys in a finite(fixed-size) data structure. How?

Hashing

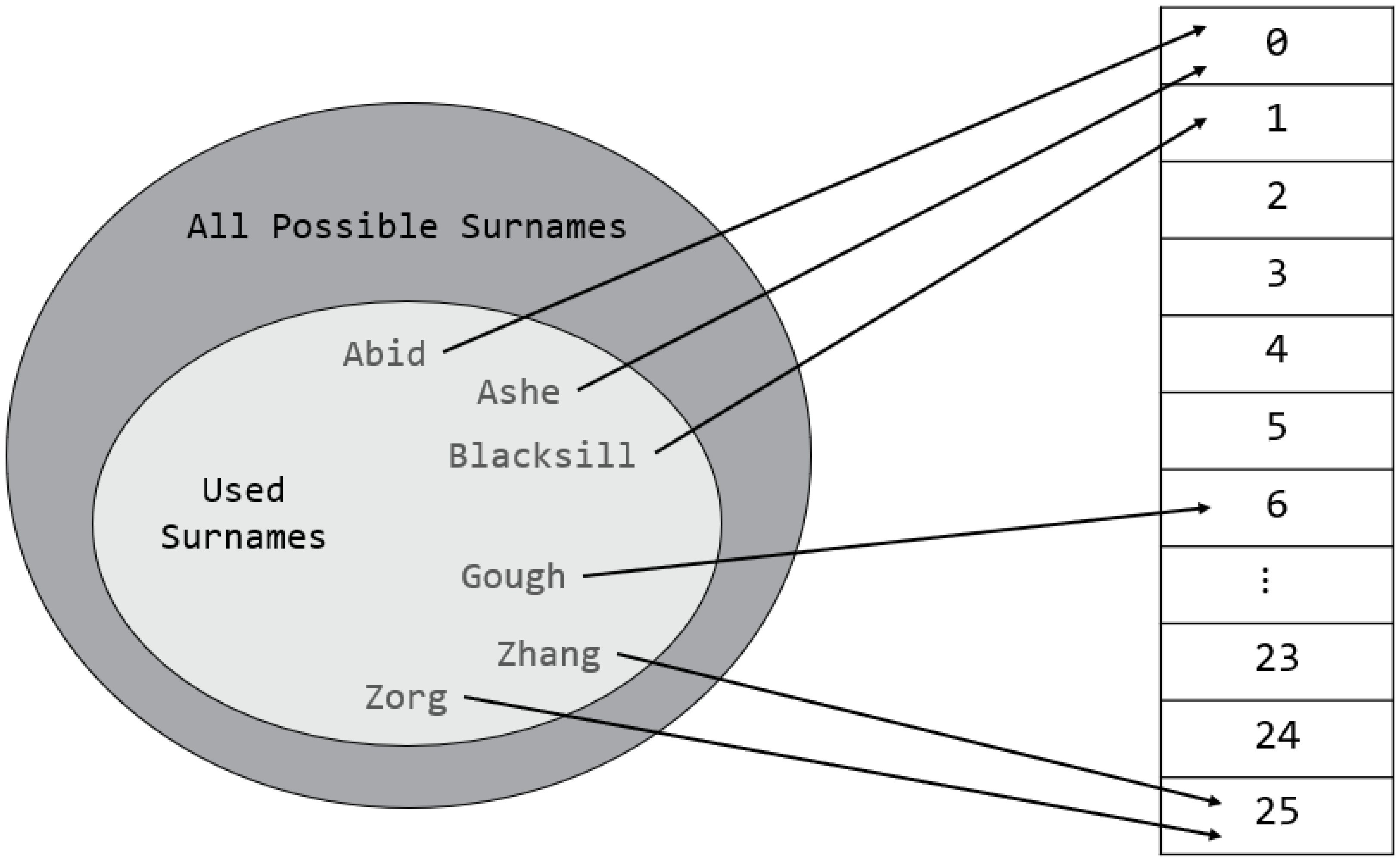
Hashing or the use of **Hash functions** is a way of taking a set of items of any size and mapping them to a fixed-size set of values.

Example

We can take all students in this room and map them to 26 items. How?

- Take the first letter of their surname.
- Only 26 letters in the alphabet.

A hash function normally maps a larger set of keys to a smaller set.


All Possible Surnames
Used
Surnames
Abid
Ashe
Blacksill
Gough
Zhang
Zorg
0
1
2
3
4
5
6
⋮
23
24
25

Simple example

Map all the integers to 10 values. This is an infinite set of keys!

Simple, let x be the integer.

Then,

$h(x) = x \ \% \ 10$ where $\%$ is the modulus operator.

or in python

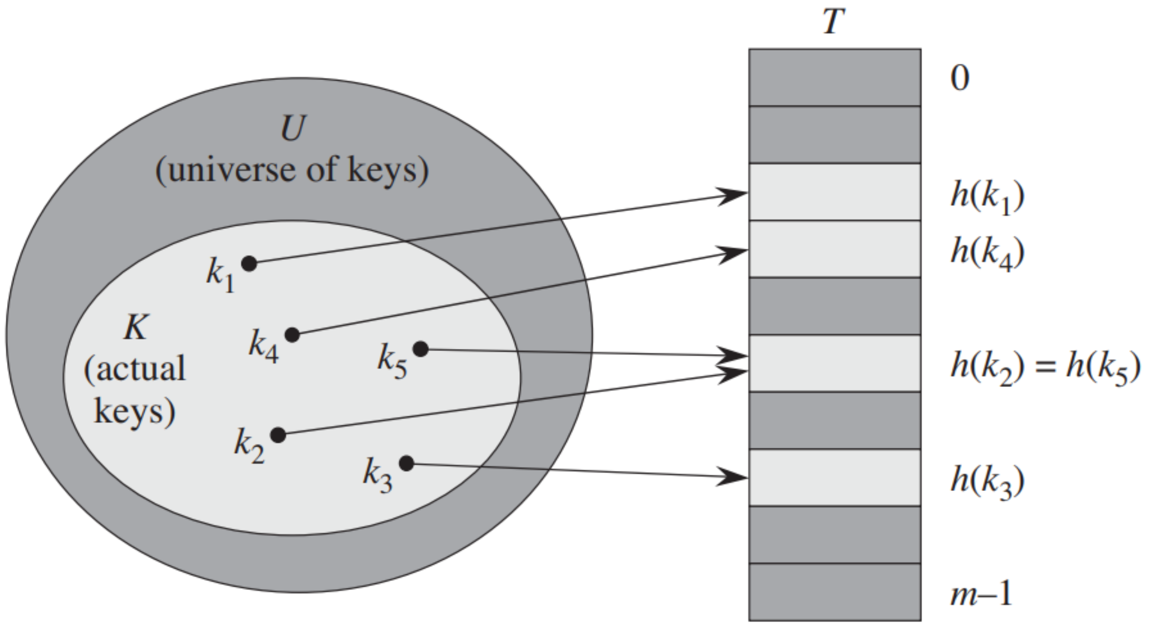
```python
def h(x):
  return x % 10

h(97) # returns 7
h(7869) # returns 9
```

Give me any whole number and I'll give you back a value between 0 and 9.

Hash Function Pictorially

In general you can take a set of keys U of size $|U|$ and map it to another set $0, 1, 2, \ldots, m-1$

Formally $h : U \to \{0, 1, 2, \ldots, m-1\}$



Hash Tables

A Hash Table is a way of implementing the Map (ADT).

If you like, it's how we create dictionaries in Python, C# etc...

Phone Book Example

- Surname used as key
- Phone Number stored as value
- Array of size 26
- We will hash the Surname using our previous idea
  - Take the first letter of the surname and map to an integer 0-25.

```python
def hash_name(name):
  """ Takes the first letter of a name and maps to a number 0 - 26"""
  first_letter = name[0].lower()
  return ord(first_letter) - 97
```

- e.g. O'Neill -> take 'O' which maps to 14. So store my address at index 14.

What is the issue here?

Collisions

Sometimes two keys will map to the same value. Let's take our surname example.

Clearly both `"O'Neill"` and `"Olson"` map to `14`.

If we store the phone numbers in an array with size 26 then we can't store both of their phone numbers!

Note if a hash function h maps a set of keys U of size $|U|$ to a set containing m keys.

If $m < |U|$, then collisions are guaranteed. Why?

Question

What should we do?

Dealing with Collisions

To create a useful hash table we need to deal with the following:

- We need a **hash function** that **avoids collisions**
- We need something **big enough** to store data
- We need something **not too big** that we waste space
- We need something that can **manage collisions** when they happen

Let's start by addressing:

- We need a hash function that avoids collisions

What Makes a Good Hash Function?

Uniformity! Avoid as many collisions as possible!

If a lot of your keys map to a given value, then you have something that becomes inefficient.

What about using this hash function to store items in an array of size 10?

```python
def rubbish_hash(x):
  return 1
```

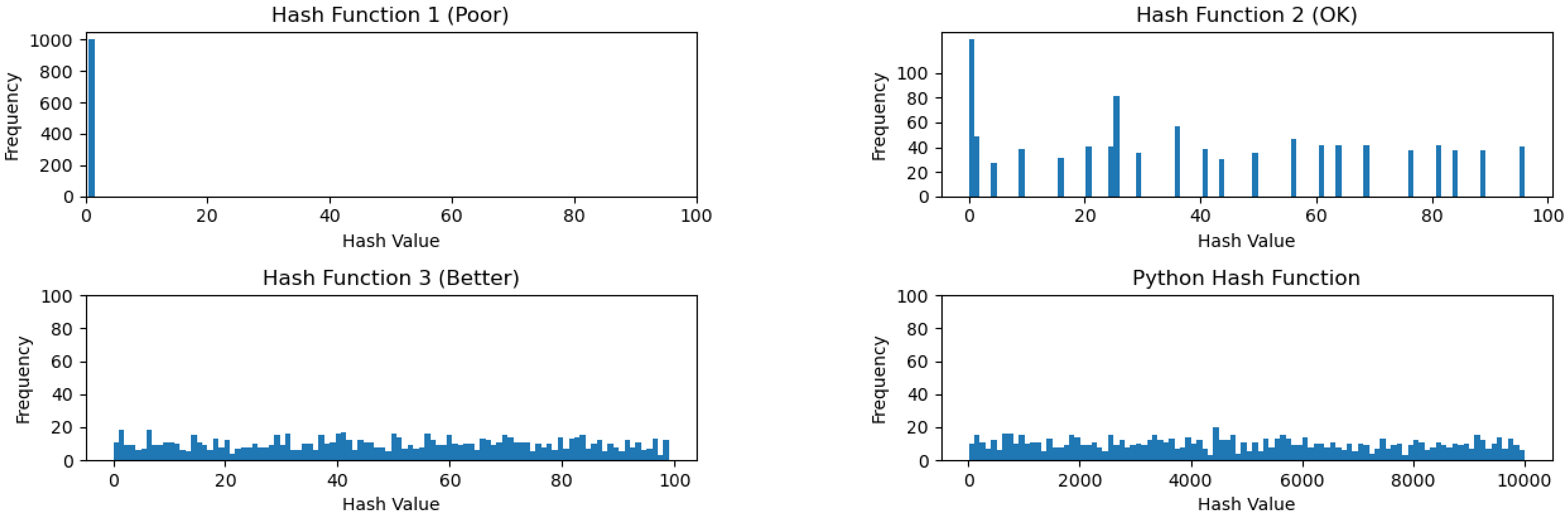
This only ever returns `1`. Thus we store every item at index 1.

Told you it was bad!

Ideally, you want the keys to be distributed evenly across the locations in the array.

Comparison of Hash Functions

- numbers 0-9999 randomly generated 1000 times.
- 4 hash functions compared.

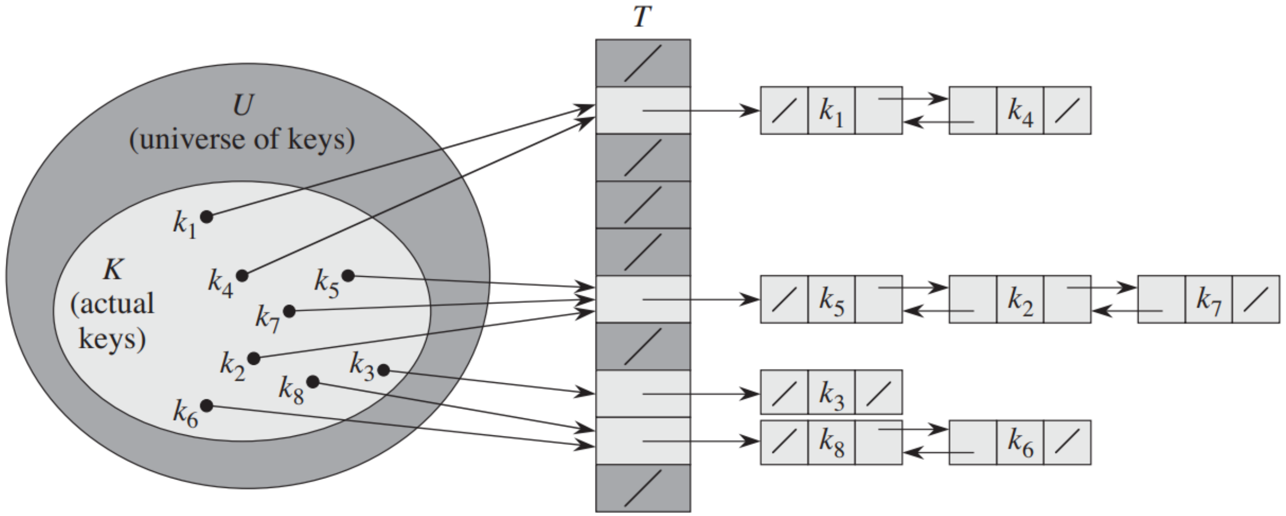


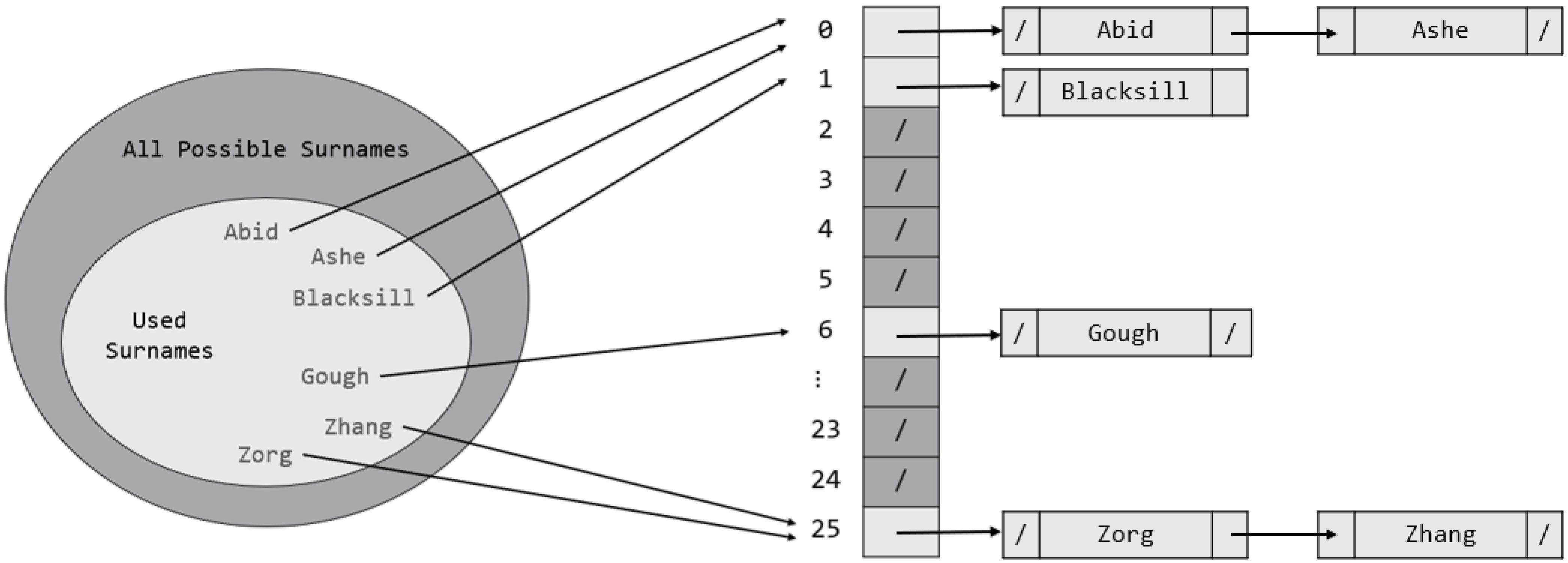
How do we address the following?

- We need something big enough to store data
- We need something not too big that we to waste space
- We need something that can manage collisions

Closed Addressing (Chaining)

- Each location in the array is called a bucket.
- Maintain a chain of items whose keys map to the bucket. Known as **separate chaining**.
- Normally done using a linked list.




All Possible Surnames
Used Surnames
Abid
Ashe
Blacksill
Gough
Zhang
Zorg
0
1
2
3
4
5
6
⋮
23
24
25
/
Abid
Ashe
Blacksill
Gough
Zorg
Zhang

Cost of Closed Addressing (Linked-list)

- It takes $O(1)$ to lookup (access) a key.
- Worst-case time complexity of searching the chain (linked list) is $O(n)$

Therefore a worst-case time complexity of $O(n)$ for `get`, `put` and `remove`.

However, keep the linked lists small and you will get on average, constant time - $O(1)$.

Open Addressing

- Each location in the array is a bucket.
- Hash the key to find it's bucket, if occupied find the next free bucket.

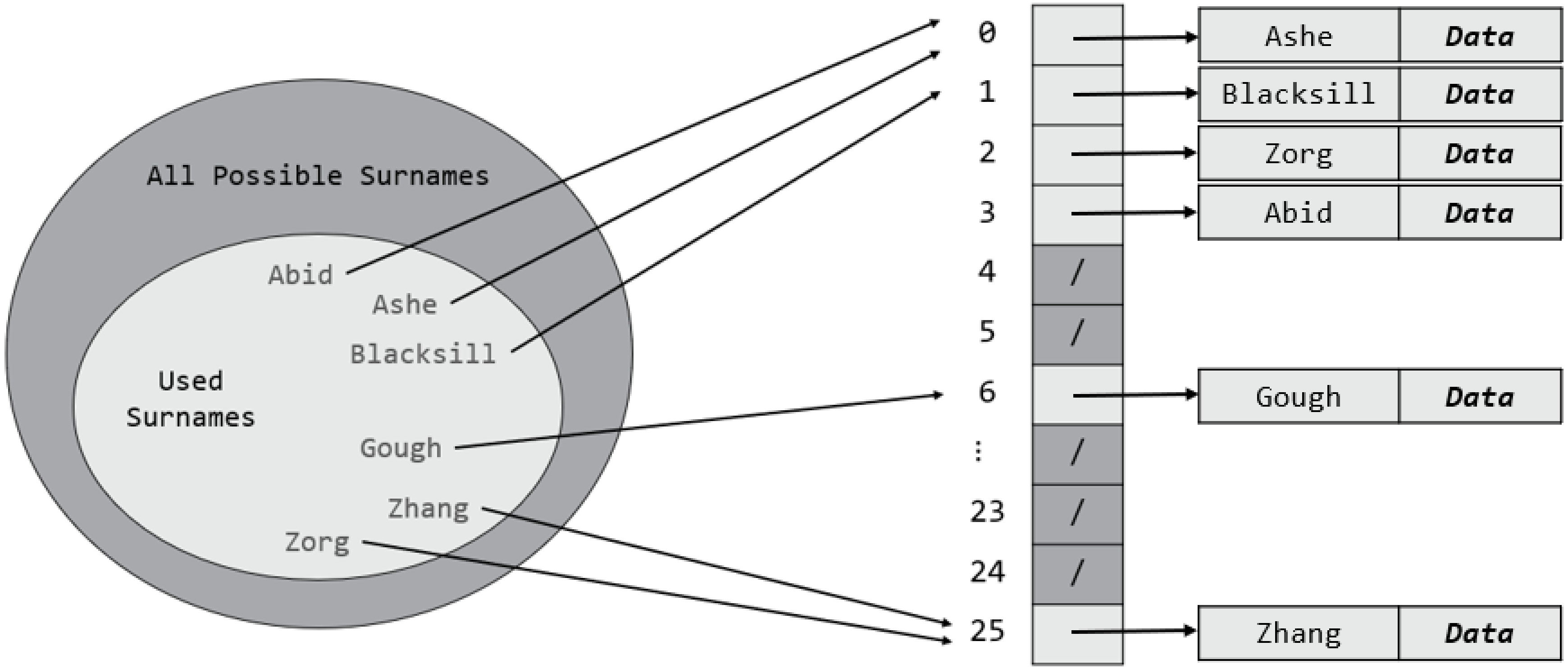
What do we mean by next free bucket?

Linear Probing (Open Addressing)

There are other types of probing

Next free bucket is just found linearly.

e.g. keep looking at the next bucket until you find a free bucket.



Order of entry: Ashe, Blacksill, Gough Zhang, Zorg, Abid

What is the cost of looking up Zhang and Zorg?

Primary Clustering

Linear probing can lead to long runs of slots built up.

$m = 10$ buckets

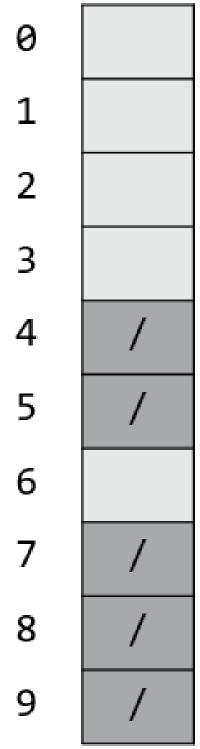
What is the probability of filling 4 next?

4 will get filled if something hashes to either of the buckets 0,1,2,3 or 4.

So 5 buckets out of 10 buckets, i.e. $\frac{5}{10}$ or 50%

In general an empty slot preceded by i full buckets is filled next with probability

$$\frac{i+1}{m}$$



Cost of Open Addressing (Linear Probing)

- It takes $O(1)$ to lookup (access) a key.
- Worst-case time complexity of searching the array is $O(n)$

Therefore a worst-case time complexity of $O(n)$ for `get`, `put` and `remove`.

However, on average, we get constant time - $O(1)$.

- Assuming we have a good hash function that distributes uniformly.

Resizing

- Clearly in open addressing, if the array fills up, then we have to resize!
- Also at about 70% full, things get inefficient. We see a lot more collisions in both open and closed addressing.

We resize based on the **load factor**.

$$\frac{n}{k}$$

where

- k is number of buckets in array
- n is number of occupied buckets

It is common to resize when $\frac{n}{k} \geq 0.7$. i.e. resize when 70% of the buckets are occupied.

Rehashing

As a result of a resize you will need a new hash function.

If you had the following hash function that worked for an array of size 10.

```
def h1(x):
  return x % 10
```

You could double the array to 20, but would need a new hash function, e.g.

```
def h2(x):
  return x % 20
```

So if we had hashed the key `15` with `h1` we would have tried to store it in bucket `5`.

Now we rehash, `h2(15) = 15`, so we store it in buket 15 of the new array.

Thus you have to rehash existing items and move them to the correct key location.

This is expensive.

We won't do the analysis in this course.

Closed 'vs' Open Addressing

Closed Addressing (Chaining)

- Typically performs better with high load factor.
- No issues with clustering.

Open Addressing

- No size overhead apart from the hash table array.
- Better memory locality and cache performance. All elements contiguous.
- Performs better than closed addressing when the number of keys is known

Map (ADT) Recipe

- Create a hash function that maps keys to an index
- Create an array the size of the set produced by the hash function
- Use either open or closed addressing to solve collisions

An Implementation in Python

I have implemented a basic hash table using a doubly linked list.

This is by no means efficient, but it is much better than direct addressing!

- Try it out
- Can you improve it?

Python Dictionaries

How are they implemented?

See the following link, but essentially they:

- Use Open Addressing
- Random Probing
- Resize when they are $\frac{2}{3}$ full.

https://stackoverflow.com/questions/327311/how-are-pythons-built-in-dictionaries-implemented

Other Uses for Hashing

- Crypotography
  - Digital signatures
  - Encryption
  - Authentication
- Databases
  - Retrieving via the index column
- Load balancing
- Fraud detection

Summary

- Hash function maps a set of keys U to another set of size m
  - A good hash function is uniform
- Hash Table is a data structure
  - Used to implement a Map (ADT)
- Open Addressing uses an array and probing
- Closed Addressing uses an array, a chain (linked list)
- Average time is $O(1)$ for `get`, `put` and `remove`

References

Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 2022. Introduction to algorithms. MIT press.