SCC.121 Fundamentals of Computer Science Week 11, Lecture 1: Hashing

Amit Chopra

Emma Wilson

Today's Lecture: Overview

- Introduction and term ahead
- Hashing
  - Review of collisions (recap from lecture on indexed retrieval)
  - Hashing

Plan for this term

- This lecture (week 11, lecture 1): Data Structures, Hashing with Dr Emma Wilson (covering for Dr. Amit Chopra)
- Next lecture till end of week 15: Algorithms and Complexity with Dr Emma Wilson
- Week 16 – Week 21: Abstract data types with Dr Fabien Dufoulon

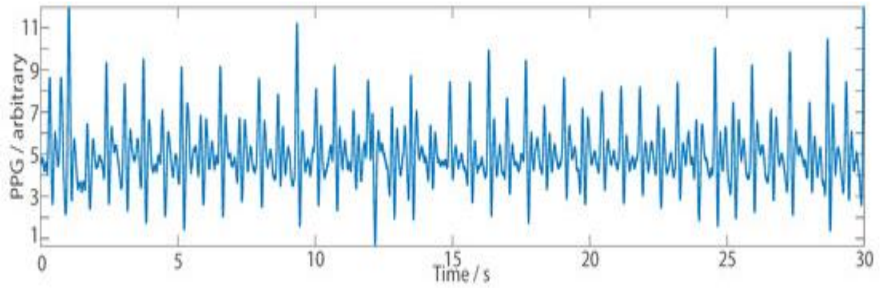
About Me

Emma Wilson

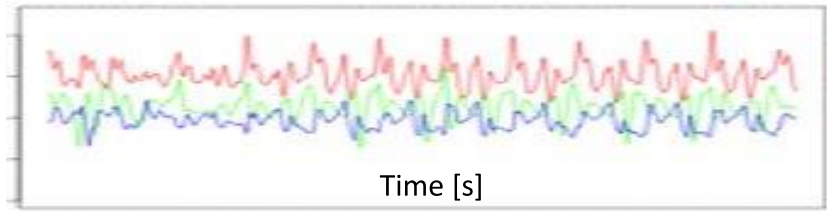
Data Science Research Group

- Time series data
- Real time signal processing, control and optimisation
  - Health data, e.g. heart rate or accelerometer signals
  - Robotics and bioinspired control
- Efficiency of algorithms key

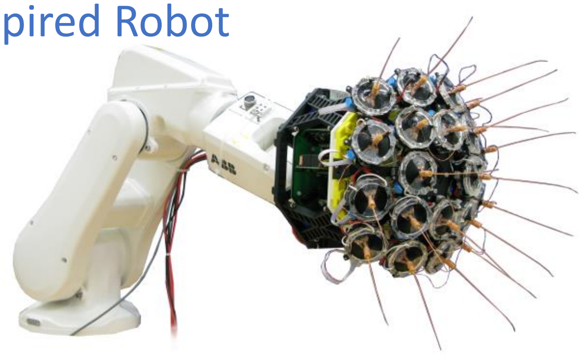
Heart Rate PPG Signal



Accelerometer signals during walking



Bioinspired Robot



Support

- FastHub – open every weekday afternoon in B77 Infolab (next to computer labs on B-floor. Drop in support. https://www.lancaster.ac.uk/scc/study/facilities/fast-hub/
- Labs
- My contact: e.d.wilson1@lancaster.ac.uk , office c50 Infolab (best to email first). Note I work part time and only on Mon, Tues, Thurs
- Note: All slides now on Moodle, but may be slightly updated

Slido

I often use Slido in my lectures

- Anonymous – please use responsibly
- To check understanding – e.g. multiple-choice questions
- To enable audience questions (time permitting)

slido

Please download and install the Slido app on all computers you use

Audience Q&A

ⓘ Start presenting to display the audience questions on this slide.

SCC.121: **Indexed Retrieval and Hashing**

Amit K. Chopra

amit.chopra@lancaster.ac.uk

Today

- Review of collisions and collision-free storage

- Hashing

Collisions

Recap from last lecture on indexed retrieval

Collision-free storage

Test whether a word is in a set of words:

break case data for if make null quit return save test value while

They all start with different letters

- Hold the set in an array of 26 elements with the index of each word determined by its first letter: e.g. A[1]="break", A[2]="case", A[22]="while"
- Take the first letter of the word to test
- Check it in the appropriate array element.
- It *either* matches *or* the word is definitely not in the set.

Collision-free storage: algorithm

Given ***name***, look for record in ***STORE*** with matching key:

```
take first letter L from name;
convert L to numeric value V;
if (STORE[V].key == name)
{
 return STORE[V];  // record found
}
else ......         // no matching key
```

Note: only *one* element of *STORE* is tested; if it does not match, the retrieval *fails*. Hence, this is a fast, *constant time*, O(1), algorithm.

Dealing with collisions using indexed retrieval

Solution 1 – Indexed retrieval (as above)

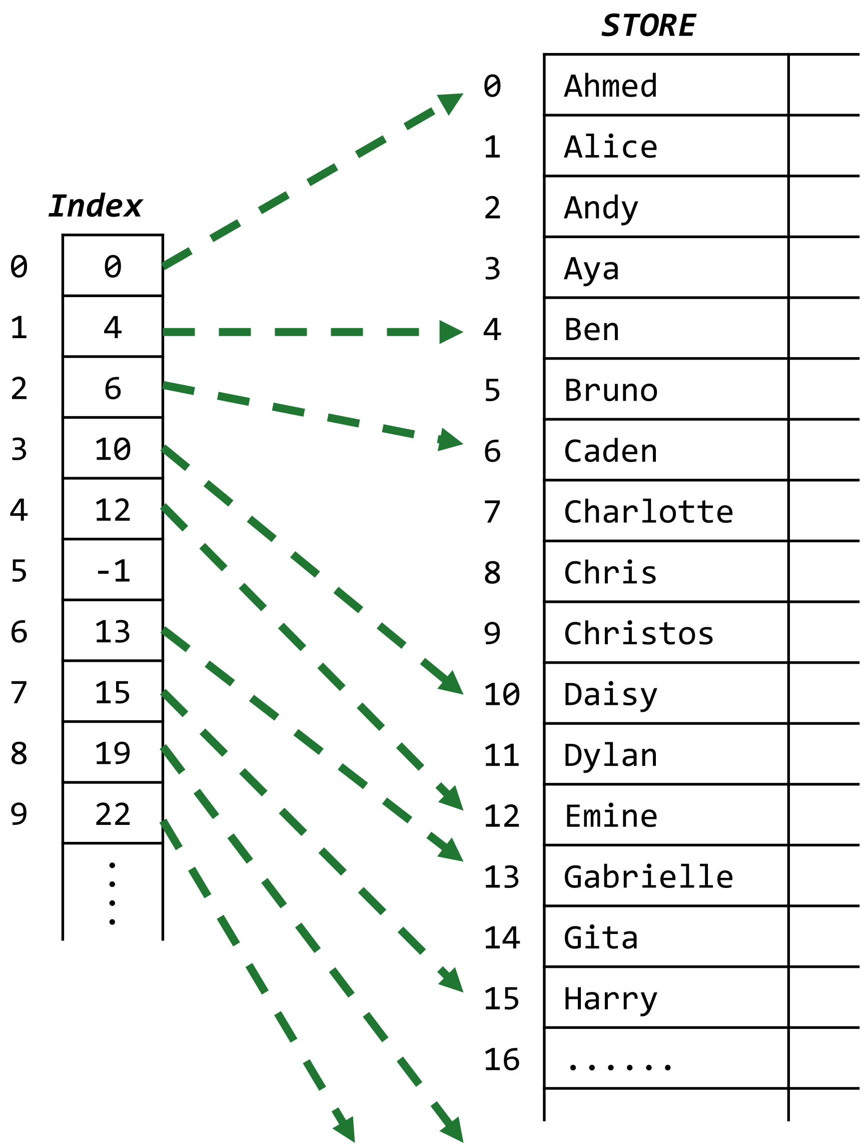
- put keys into a sorted array
- use an index array to find relevant section

but

to insert extra data, the index array must be recomputed

Indexed retrieval: adding an entry (1)

What if we need to add an entry for Christophe?



Index position	Index value
0	0
1	4
2	6
3	10
4	12
5	-1
6	13
7	15
8	19
9	22
	⋮

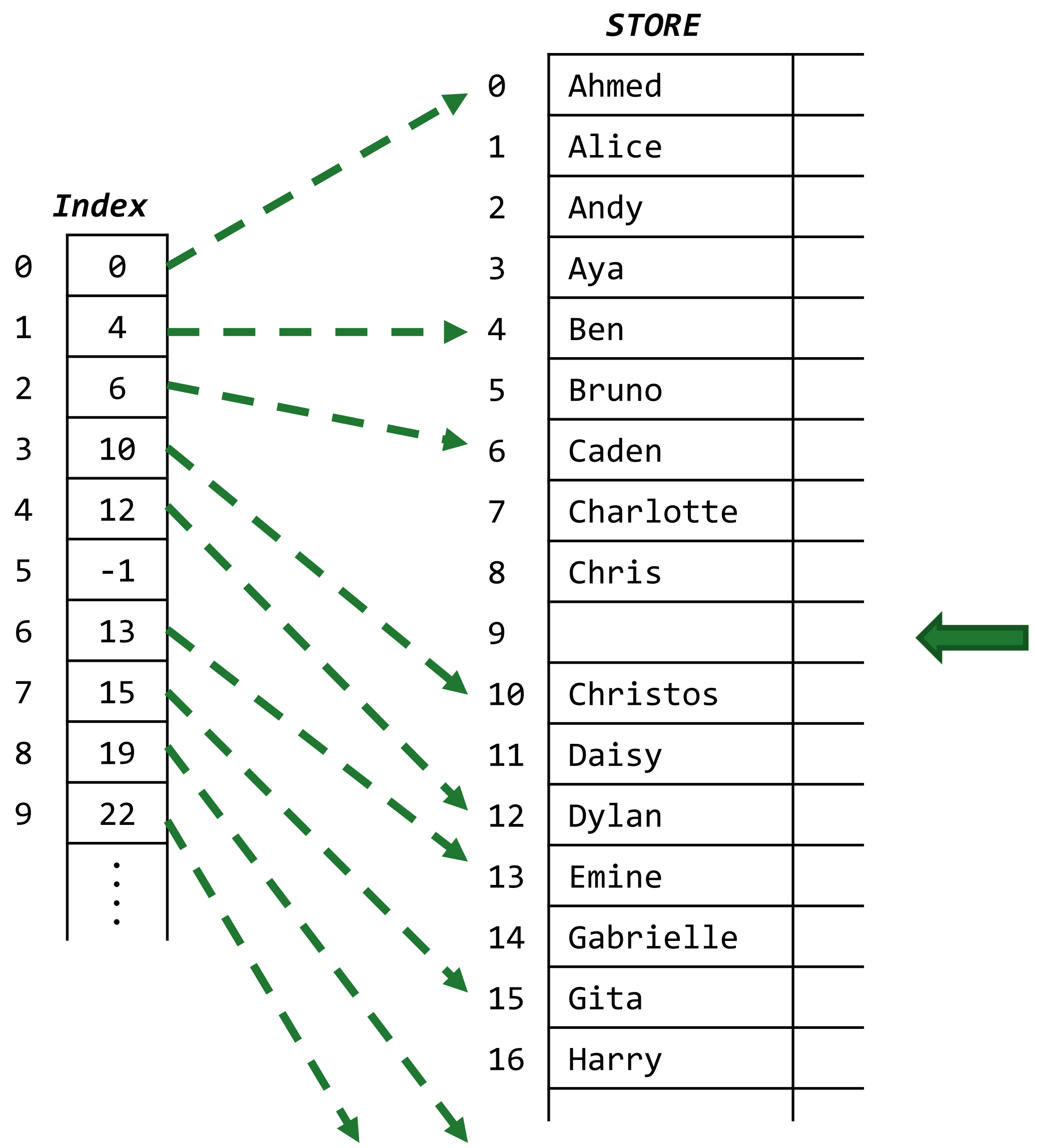
Store position	Store entry
0	Ahmed
1	Alice
2	Andy
3	Aya
4	Ben
5	Bruno
6	Caden
7	Charlotte
8	Chris
9	Christos
10	Daisy
11	Dylan
12	Emine
13	Gabrielle
14	Gita
15	Harry
16	......

Indexed retrieval: adding an entry (2)

What if we need to add an entry for Christophe?

First we have to make room for the entry in the store.

This means all the entries from position 9 onwards have to be shifted into the next slot.



Index position	Index value
0	0
1	4
2	6
3	10
4	12
5	-1
6	13
7	15
8	19
9	22
	⋮

Store position	Entry
0	Ahmed
1	Alice
2	Andy
3	Aya
4	Ben
5	Bruno
6	Caden
7	Charlotte
8	Chris
9	
10	Christos
11	Daisy
12	Dylan
13	Emine
14	Gabrielle
15	Gita
16	Harry

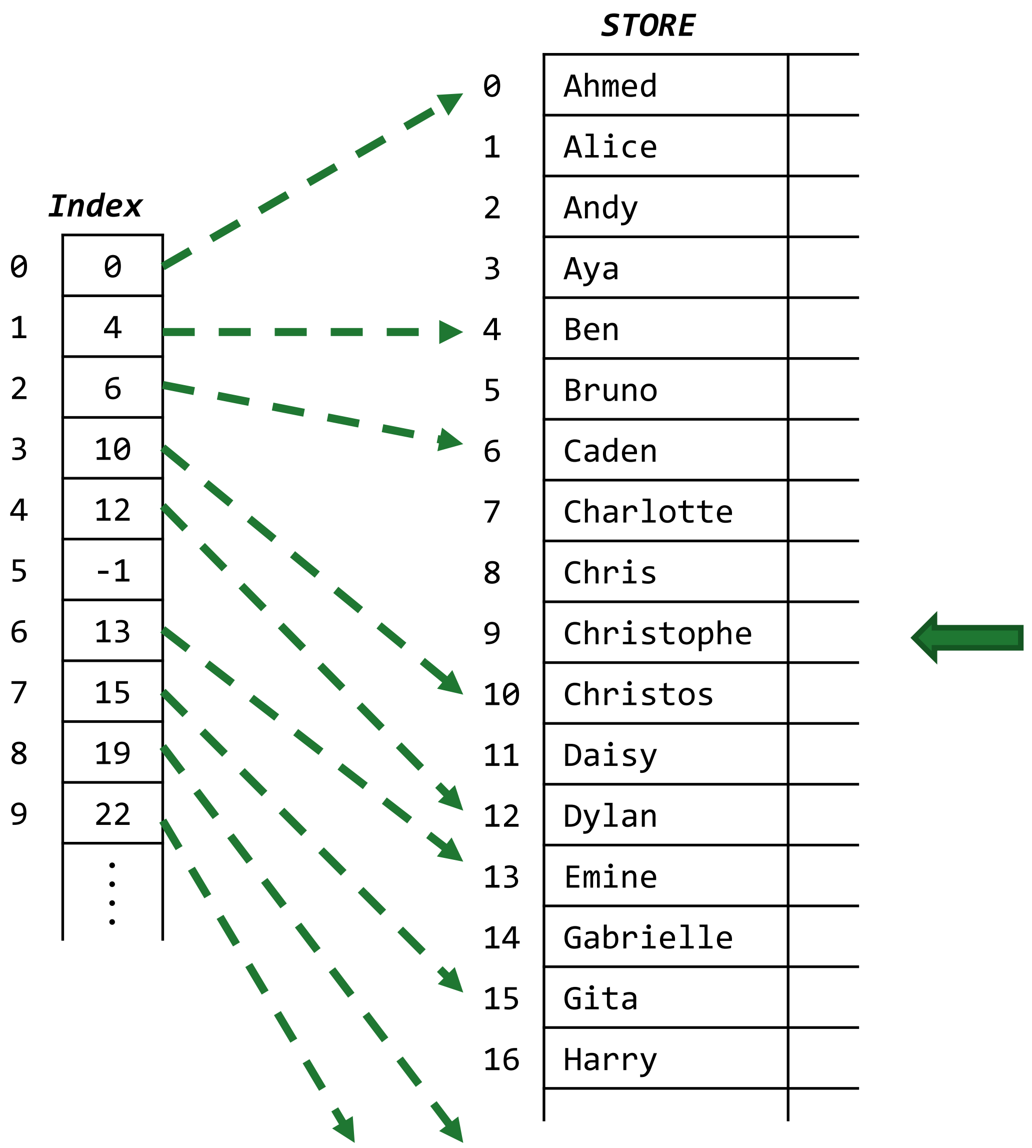
Indexed retrieval: adding an entry (3)

What if we need to add an entry for Christophe?

First we have to make room for the entry in the store.

This means all the entries from position 9 onwards have to be shifted into the next slot.

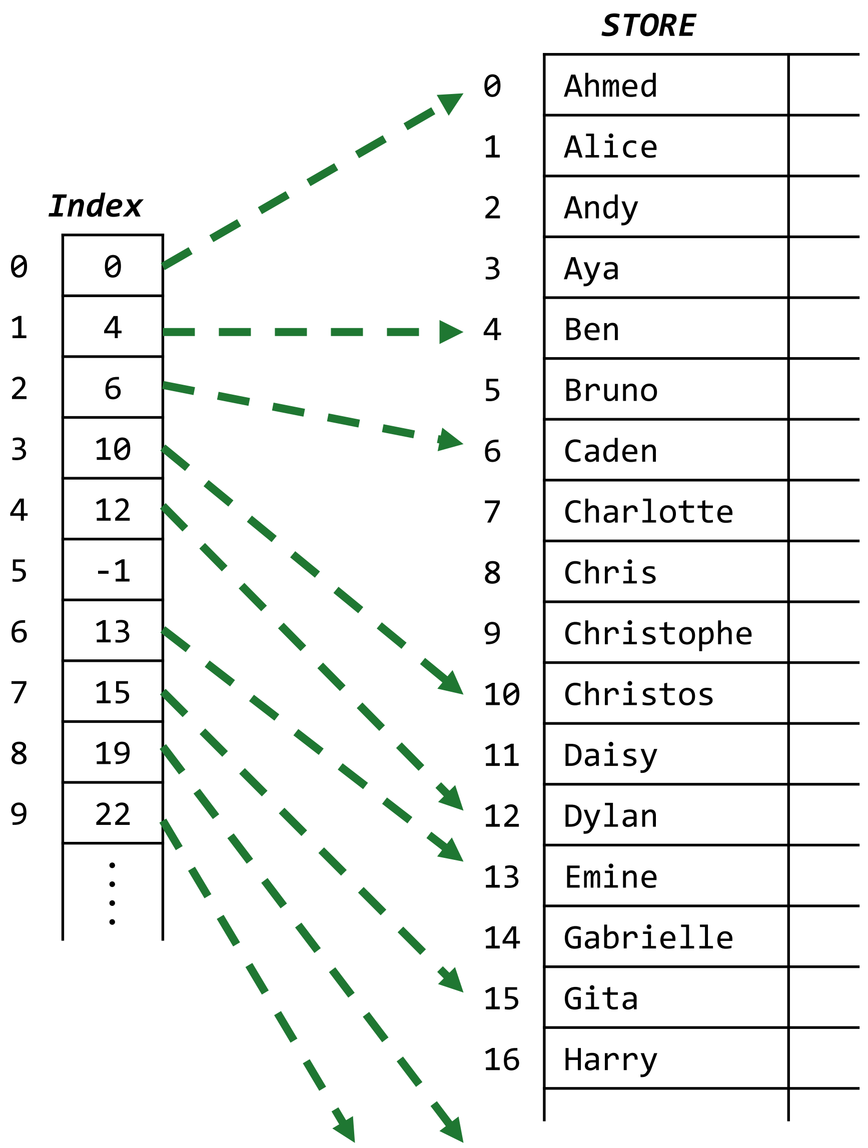
Now we can add Christophe.



Indexed retrieval: adding an entry (4)

What if we need to add an entry for Christophe?

Now the Index is wrong, from element 3 onwards.

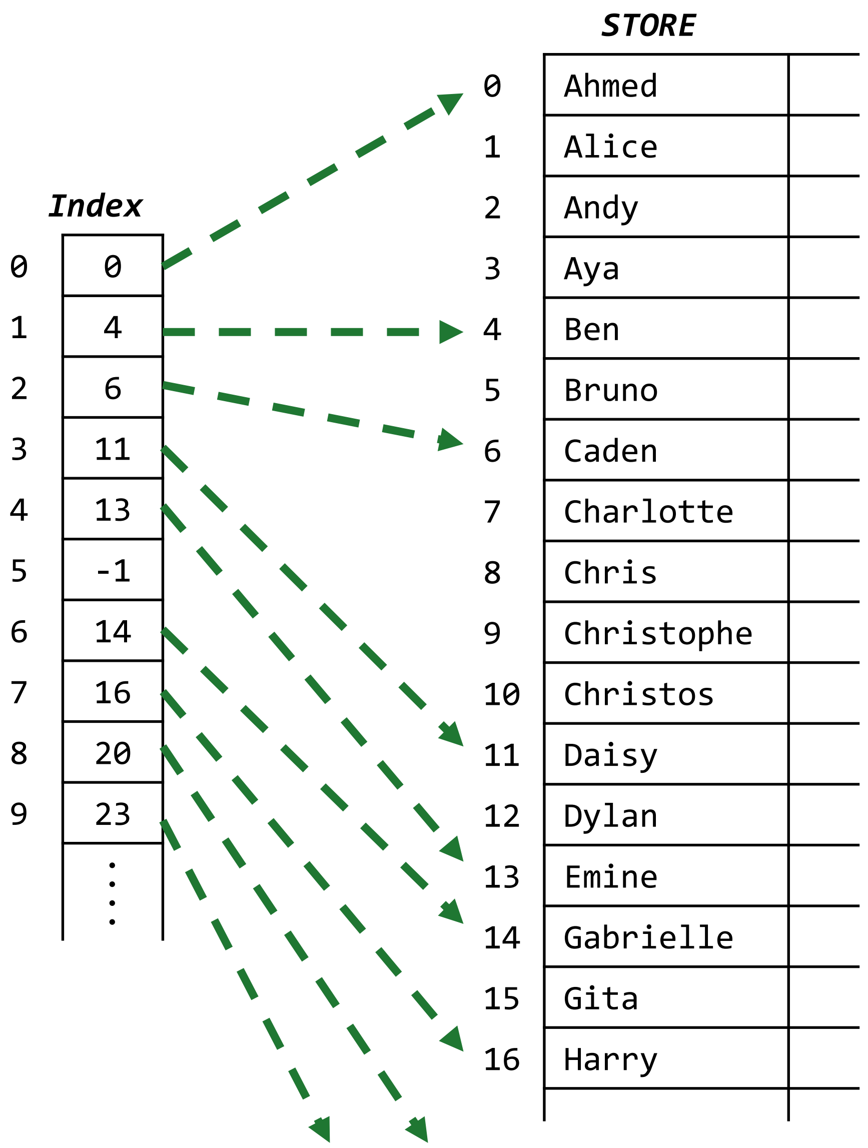


Indexed retrieval: adding an entry (5)

What if we need to add an entry for Christophe?

Now the Index is wrong, from element 3 onwards.

We have to recompute the Index. Adding 1 to every positive value, starting at element 3.



Dealing with collisions using chains (linked lists)

Solution 2 – Using chains

For storage based on first letter of each key:

use a linear array *STORE* of 26 elements representing the alphabet with each element pointing to the appropriate chain; i.e. *STORE*[0] points to a chain of objects whose keys starts with ‘A’

```
STORE[0] → Ahmed → Alice → Andy → Aya
STORE[1] → Ben → Bruno
STORE[2] → Caden → Charlotte → Chris → Christos
STORE[3] → Daisy → Dylan
STORE[4] → Emine
STORE[5]
STORE[6] → Gabrielle → Gita
 .......
```

Indexed retrieval algorithm using chains

Given ***name***, look for record in *STORE* with matching key:

```
take first letter L from name;
convert L to numeric value V;
obtain chain pointer  P  from  STORE[V];
while (P!=null)
{
 if (P.key==name)             //  record found!
  {  return P; }
 P=P.next;              //  move to next record
}
//  if loop exhausts, name was not found.
```

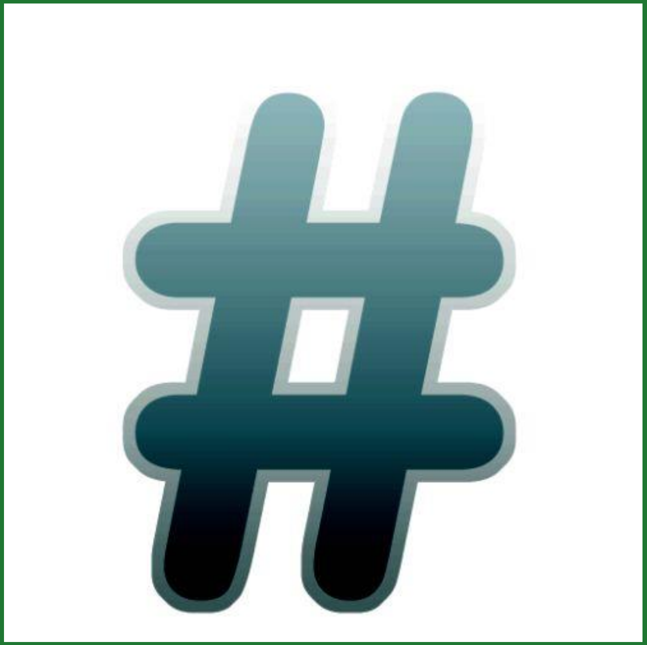
Dealing with collisions using chains: Efficiency

***Worst case*:**
Every object is on the same chain
Like linear search, O(*N*)

Practically: Much faster – if the objects are distributed across many chains, so same improvement as with using arrays.

Advantage

- Chains are dynamic, so extra objects can be inserted more easily
- No need to recompute index



HASHING

Using just the first letters is very limiting

So we need something better …

Hashing Function

- Converts a **key** (often a string) to an integer

- Integer is used for indexing a storage array.
- The same *deterministic* function is used for storing and retrieving data.

- Index integer is not just restricted to 0 to 25.
- Hence, data can be spread across a much longer array.

- Collisions can be dealt with by chaining.
- If collisions are few, high performance: O(1)

Hashing: Introduction

- A Hash Table is a data structure that is designed to be fast to work with.
- Hash Table elements are stored in what we often call **buckets** (e.g. index of array).
- Every Hash Table element has a part that is unique that is called the **key**.
- A **hash function** takes the **key** of an element to generate a **hash code**.
- The hash code says what bucket the element belongs to, so now we can go directly to that Hash Table element (assuming no collisions)

- The **hash function** maps keys to array indices (often called **buckets**).

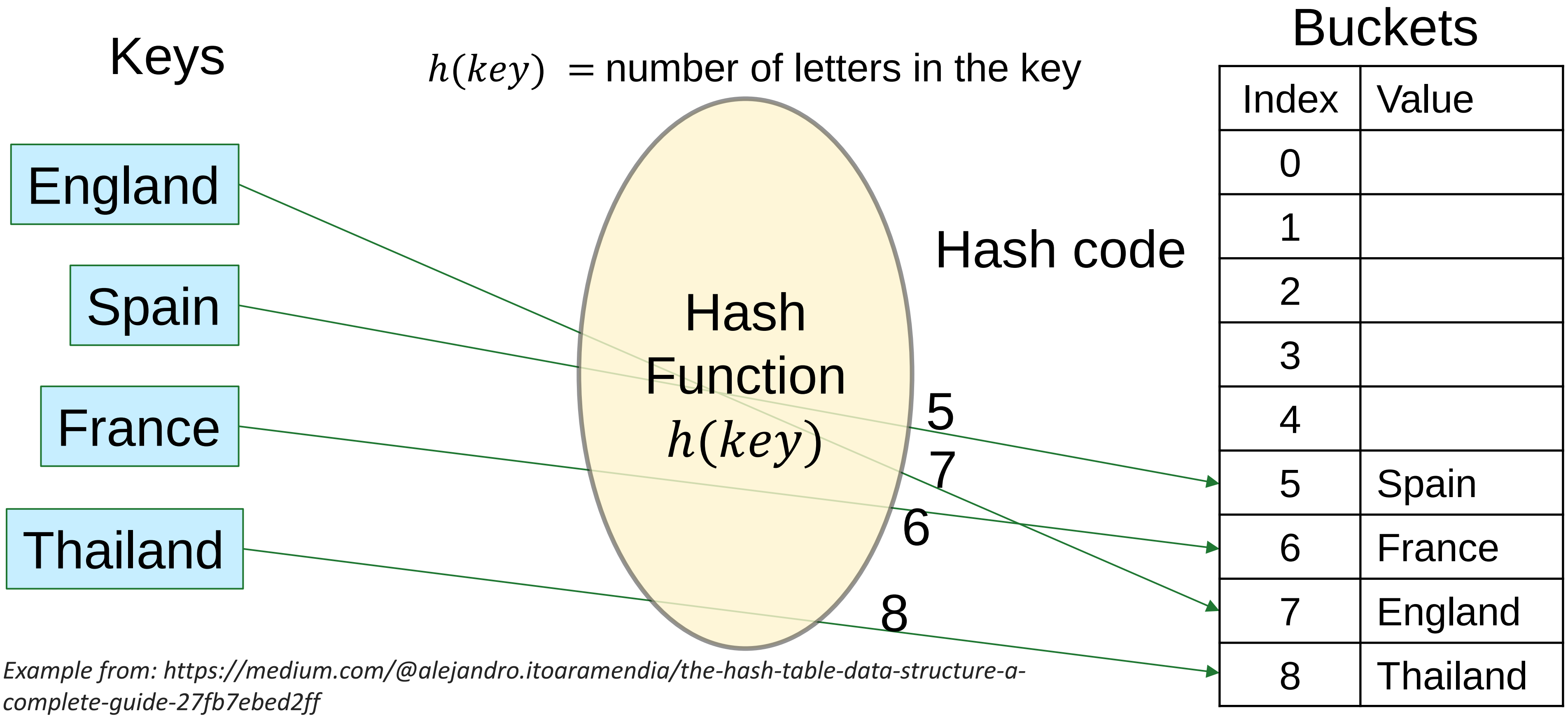
$$h: U \rightarrow \{0,1,2,\ldots,m-1\}$$

- Here h is the hash function, U is the universe of keys and m is the size of the array.

Hashing: Introduction

Given a set of keys, a hashing function computes indices that determine where the entry will be stored and can be found later.

Basic Example :



Index	Value
0	
1	
2	
3	
4	
5	Spain
6	France
7	England
8	Thailand

Example from: https://medium.com/@alejandro.itoaramendia/the-hash-table-data-structure-a-complete-guide-27fb7ebed2ff

Hashing: example

Simple hashing function:

sum the alphabetic index of the first 4 letters of key

a=0, b=1, …, z=25

key = ‘cart’	index = 2+0+17+19 = 38
key = ‘apple’	index = 0+15+15+11 = 41
key = ‘twine’	index = 19+22+8+13 = 62
key = ‘yours’	index = 24+14+20+17 = 75
key = ‘mine’	index = 12+8+13+4 = 37
key = ‘back’	index = 1+0+2+10 = 13
key = ‘tour’	index = 19+14+20+17 = 70

Sum first four letters hashing

Note:

Collisions soon start to occur, e.g.

key = ‘cart’	index = 2+0+17+19 = 38
key = ‘crater’	index = 2+17+0+19 = 38

Limitations

- The array is limited to 104 (4 * 26) elements.
- Indices above 75 may be rarely produced.
- Indices below 10 may be rarely produced.

Increasing range of indexes

- For more objects (more keys), need longer array to minimise collisions.

- Hashing function needs to produce a *large range* of integers.

- Use more information from each key.
  - e.g. all letters, and perhaps length

- But, just *adding* these together, a 15-letter key will rarely give an index, say, over 300.

Designing a good hashing function

To minimise collisions*:*

- a wide range of index values *and*
- unform hashing (each key is equally likely to map to an array location).

To get a wide range of index values:

- use multiplication as well as addition

A better hashing function

Using the first four letters, *L1, L2, L3* and *L4* of the key

$$index = L1 + L2 \times 26 + L3 \times 26^2 + L4 \times 26^3$$
$$= L1 + L2 \times 26 + L3 \times 676 + L4 \times 17576$$

key = ‘pace’: index = 15 + 0×26 + 2×676 + 4×17576
= 71671

key = ‘cart’: index = 2 + 0×26 + 17×676 + 19×17576
= 345438

but: if we have only 100 or so words to store, we don't want to have to use a 350,000 element array…

Dealing with large indices

If we want to use an array of length ***size***, we can use:

index = rawIndex%size:

where ***%*** is the integer remainder operation

e.g. to store 50 or so words, choose ***size*** = **100**:

key = "pace", rawIndex = 71671
index = 71671***%***100 = <u>71</u>

key = "cart", rawIndex = 345438
index = 345438***%***100 = <u>38</u>

The hashing pitfall

Note that hash retrieval has two stages:

1. Use hashing function to compute index;
2. Search chain in array element to see if there is a match

With a good hashing function and a large array, Stage 2 can be close to O(1).

but, Stage 1 should also be *fast*.

real-world hashing functions:

- avoid slow multiplication and division operations
- use fast logical shift and ‘OR’ operations

Multiplication vs bit shifting

$$X * 2^N == X << N$$

- Multiplication is a slow operation.
- Where the multiplier is a power of 2, we can left shift the bits in the multiplicand (X) by the power of 2 instead, and get exactly the same answer.
- So in our hash function, we replace 26 by the nearest power of 2, greater than or equal to 26; i.e. 32 (2^5).

Fast hashing with logical operations

Note: a 5-bit left shift multiplies an integer by 32, ie, 2^5

To encode the key ‘EOT’ using 5 bits per letter:

letter ‘T’, code 19:	$0000000000\underline{10011}$	
left-shift 5 places:	$00000\underline{10011}00000$	T x 32
letter ‘O’, code 14:	$0000000000\underline{01110}$	
combine by logical ‘OR’:	$00000\underline{1001101110}$	O + T x 32
left-shift 5 places:	$\underline{1001101110}00000$	O x 32 + T x 32^2
letter ‘E’, code 4:	$0000000000\underline{00100}$	
combine by logical ‘OR’:	$\underline{100110111000100}$	E + O x 32 + T x 32^2

Key-value pairs

- So far, we have just considered storing the key, but we can also store an associated value as a key-value pair
- Key-Value Pair: Each individual element of data (the value) is associated with a unique identifier. We use a label (the key) to identify a piece of information (the value) e.g. name (key) has associated phone number (value).

- **Key**: Unique input to hash function (can be anything e.g. integer, but often a string) that is translated by hash function
- **Hash** of the key: Determines the bucket where we store the value
- **Value** – what we put in the bucket associated with the key (in previous examples we just put the key itself in the bucket)

- We store key-value pair to distinguish between key-value pairs with the same hash.

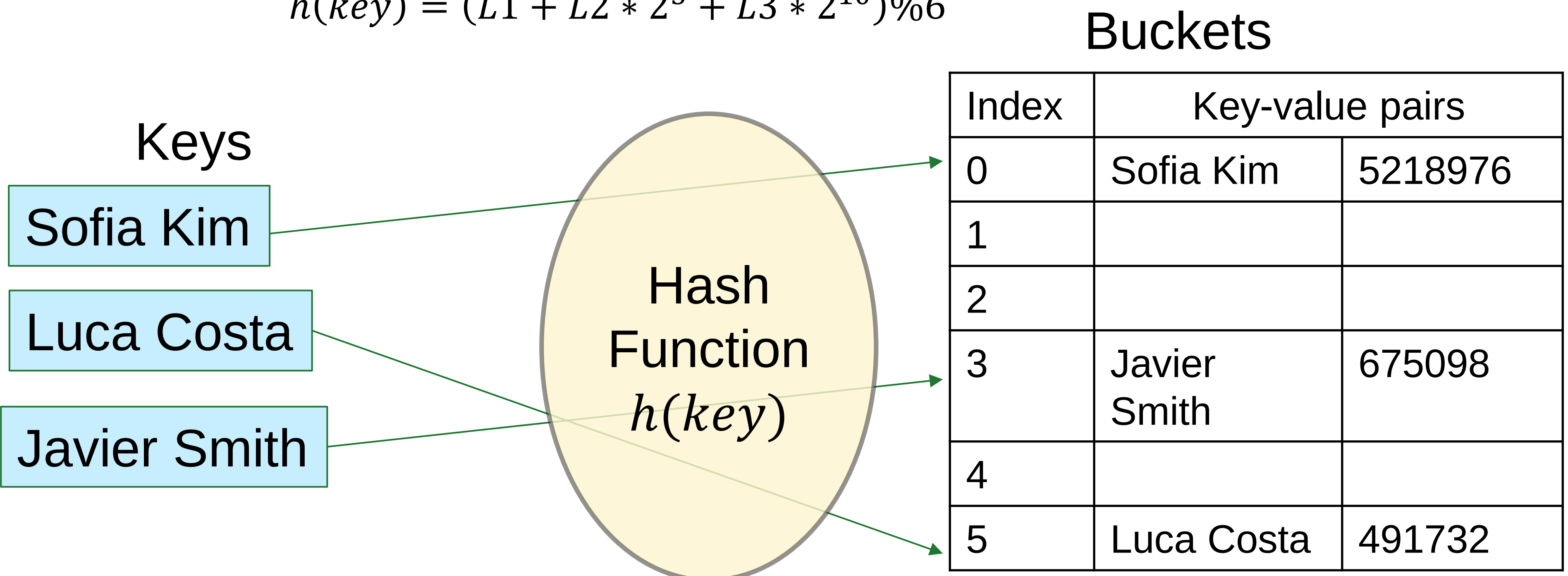
Key-value pair example

Name (key) with associated phone number (value)

- Sofia Kim - 5218976; Luca Costa - 491732; Javier Smith - 675098

For illustrative example use basic hash function

$$h(key) = (L1 + L2 * 2^5 + L3 * 2^{10})\%6$$



Index	Key-value pairs	
0	Sofia Kim	5218976
1		
2		
3	Javier Smith	675098
4		
5	Luca Costa	491732

Hash Table ADT

- Stores key-value <k,v> pairs
- Keys are unique and a handle to the associated value
- Operations
  - put(t, <k, v>) stores <k,v> in the table t
  - get(t, k) returns <k,v> if a <k,v> is in t else nil
  - delete(t, k) removes <k,v> from t

Pseudocode

Chains are lists; A is the hash table array

```
put(t, <k,v>) {
    chain = t.A[hash(k)]
    //C doesn’t have to be sorted
    add(chain, <k,v>)
}

get(t,k) {
    chain= t.A[hash(k)]
    return search(chain,k)
}
```

```
delete(t, k) {
    chain = t.A[hash(k)]
    remove(chain, k)
}
```

What happens if chains grow too long?

- Hash tables become less efficient
- To avoid this, hash table allocates an array double the size of current array
- Then, puts each <k,v> in current array into larger array
- Expensive operation (?)
- Invisible to user of ADT

Summary

- If collisions are rare, retrieval performance is O(1).

- To minimise collisions, use an array larger than the set of keys.

- Hashing function should minimise collisions by:
  - Having a wide range of index values,
  - With a uniform distribution.

- Hashing function needs to be quick
  - Using bit operations.

- Collisions can be dealt with by chaining (linked list).

slido

Please download and install the Slido app on all computers you use

Audience Q&A

ⓘ Start presenting to display the audience questions on this slide.