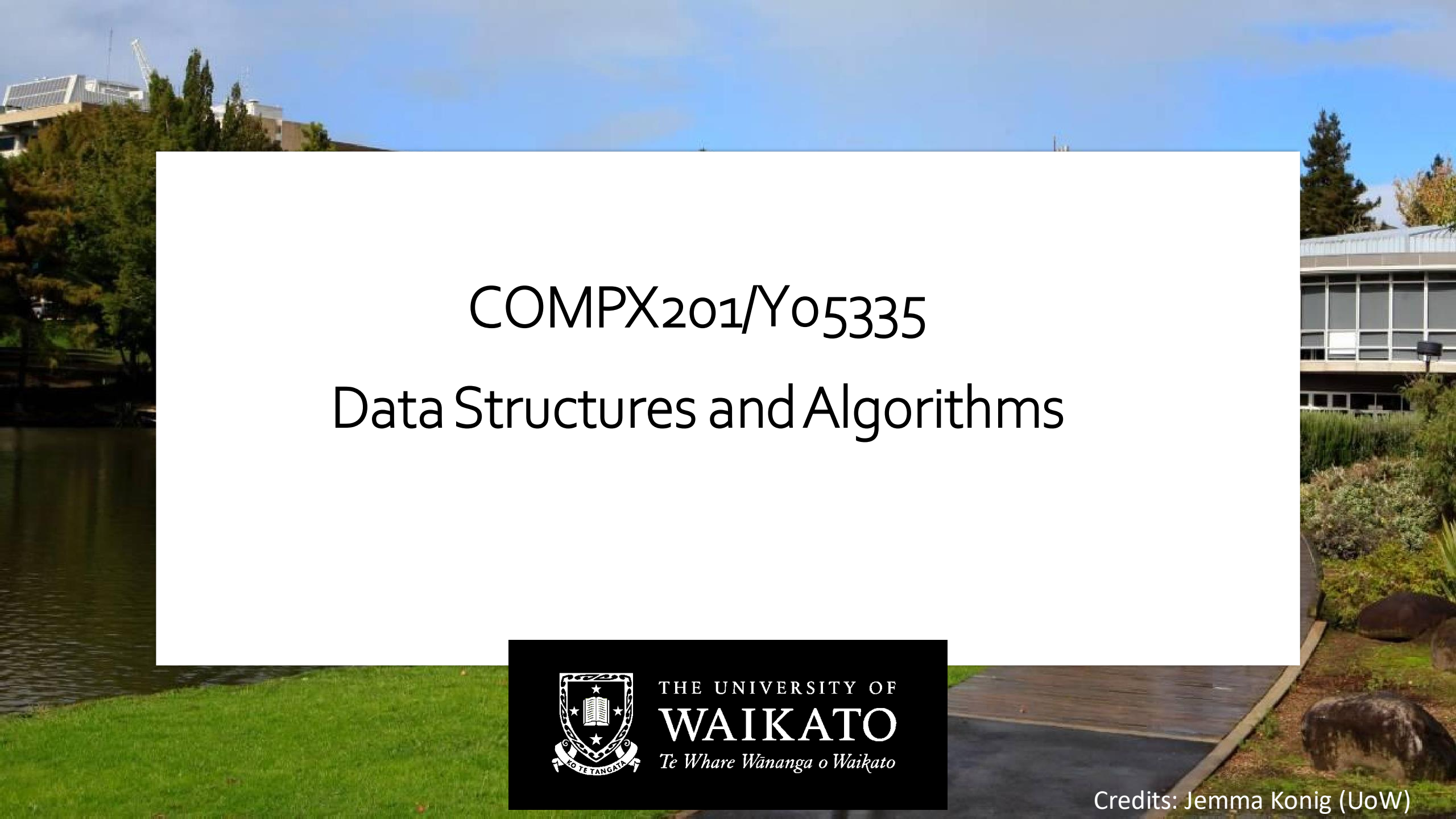


COMPX201/Y05335

Data Structures and Algorithms

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Credits: Jemma Konig (UoW)

Hash Tables

COMPX201/Y05335

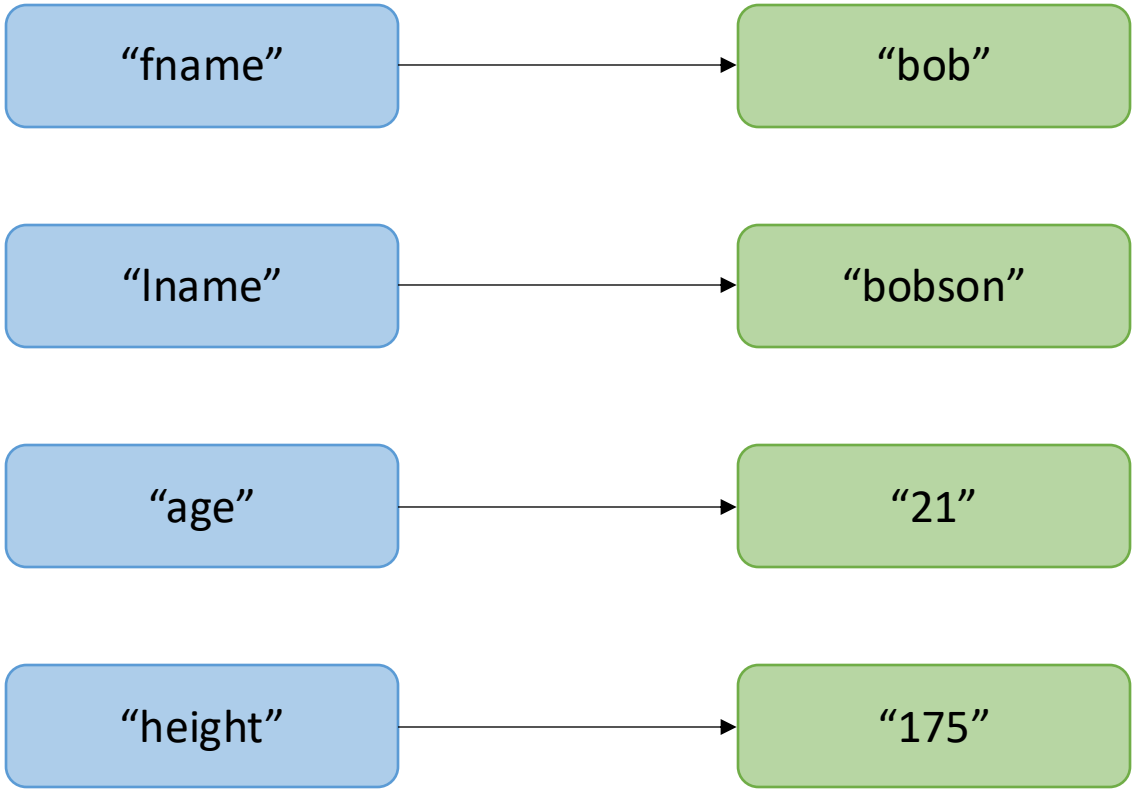
Overview

- What is a Hash Table
- Hash functions
- Rehashing
- Performance

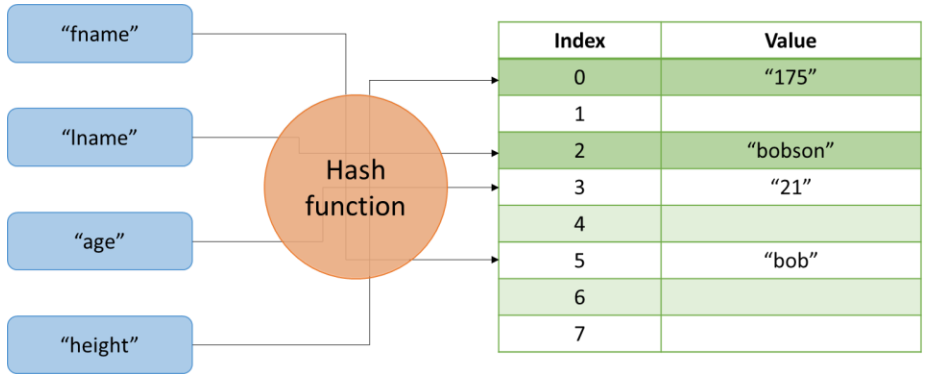
Hash Tables

What is a Hash Table

- A data structure that maps keys to values



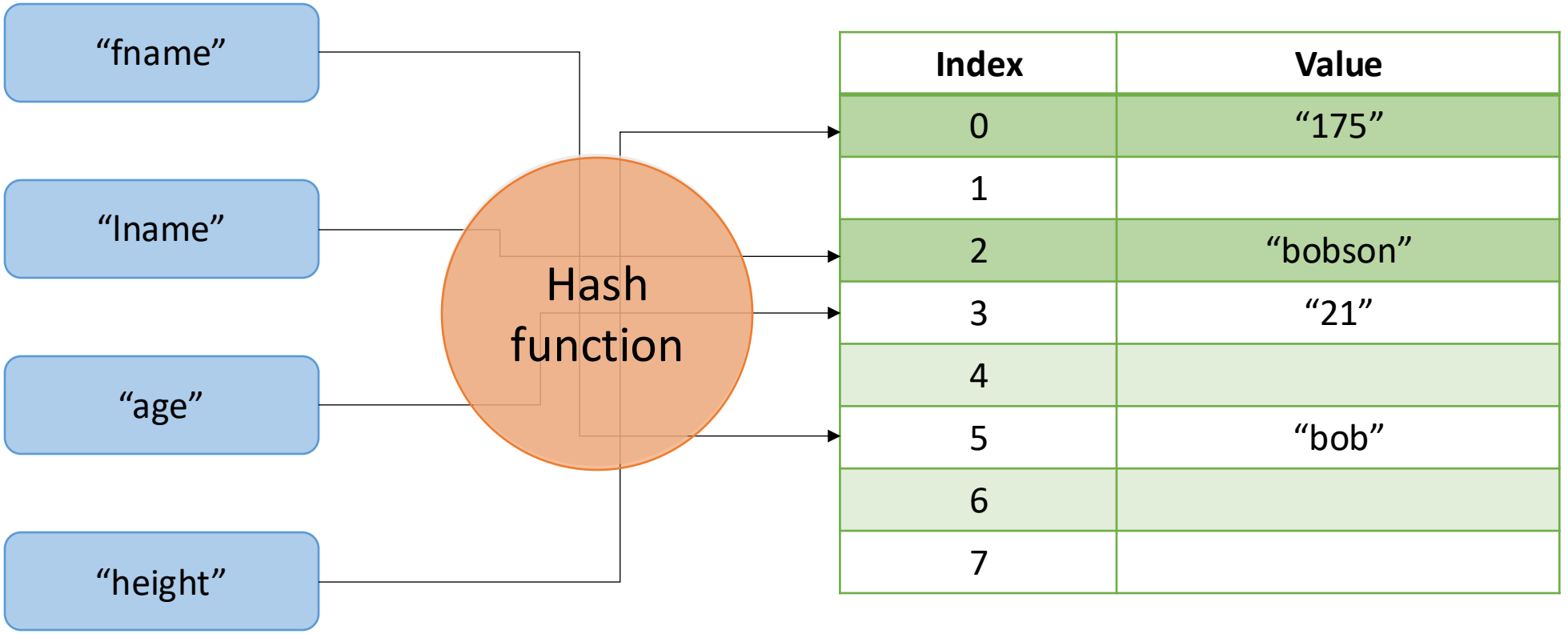
What is a Hash Table



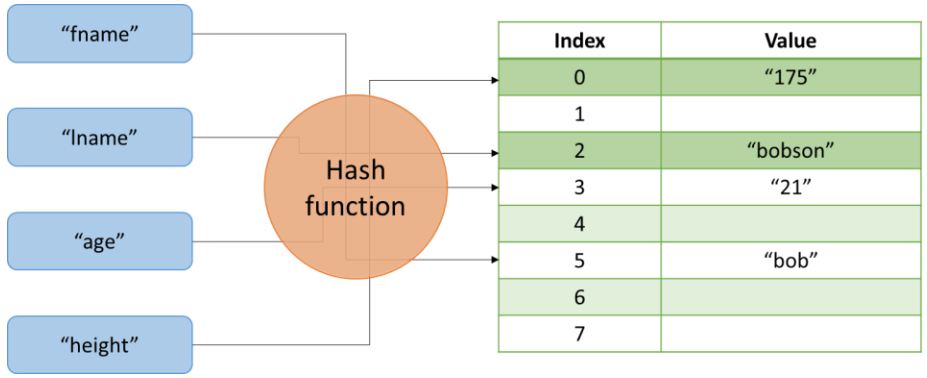
- A data structure that maps keys to values
- Each item has some unique key that identifies it.
- The key is transformed into an integer (the hash code) using a hash-code function.
- The hash code is used to index into an array where the item can be stored and found.

What is a Hash Table

- A data structure that maps keys to values



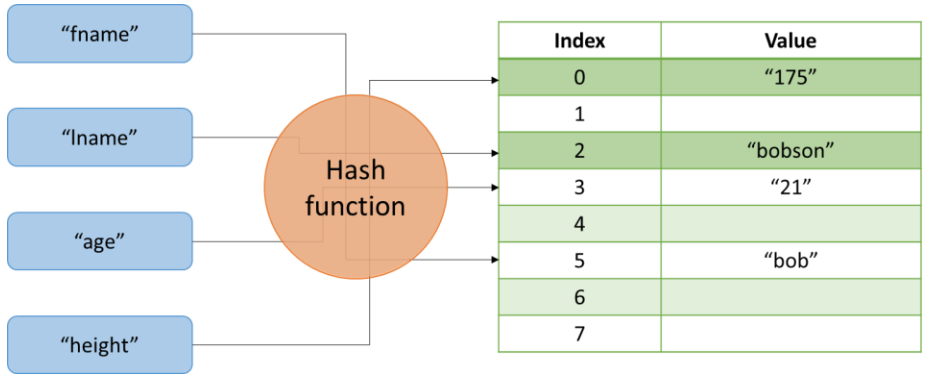
What is a Hash Table



- Used when fast look-up is required for single items.
- Do not (usually) support sequential processing or more general types of search.

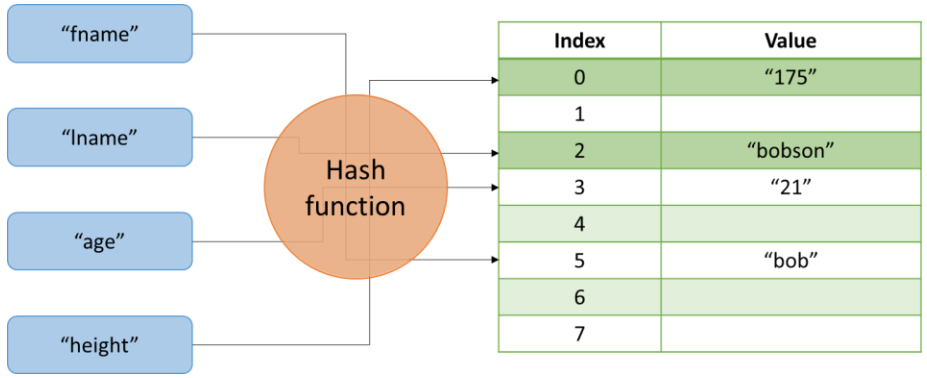
Hash functions

What is a hash function



- The hash-code function transforms a key into an integer (the hash code).
- The hash code is used to index into an array where the item can be stored and found.

Good hash function characteristics



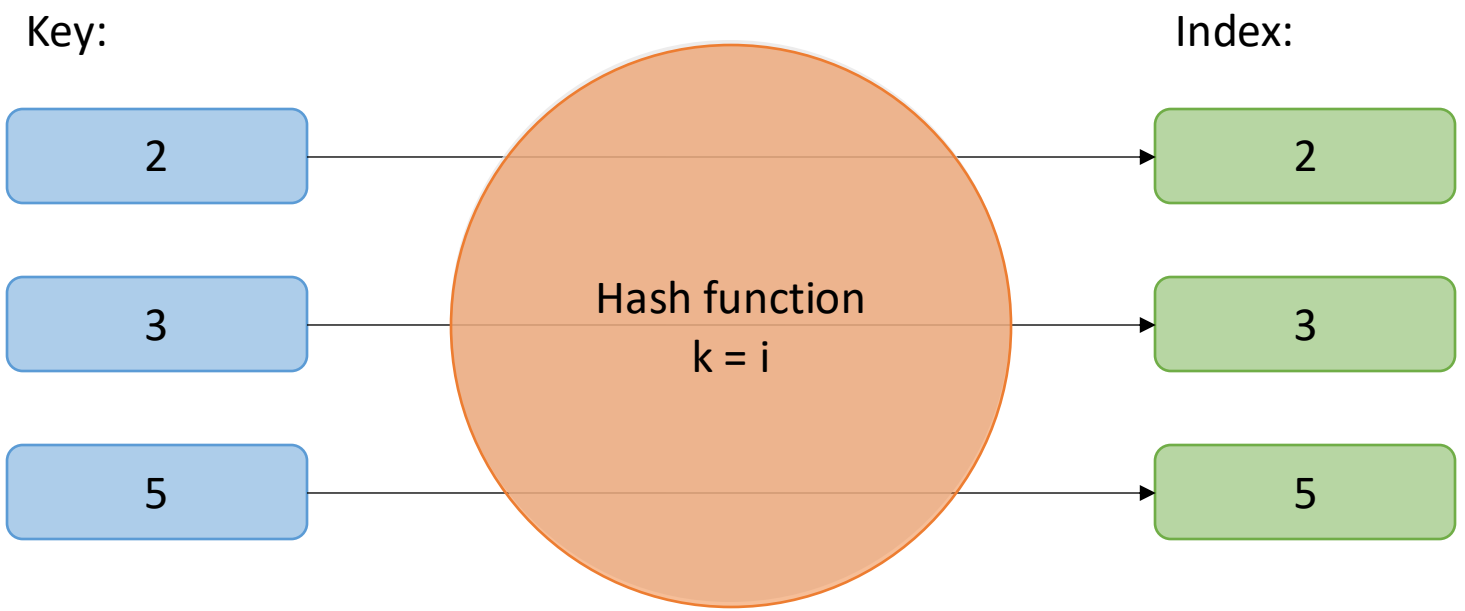
- The hash function should not generate keys that are too large in comparison with space as space will be wasted.
- Keys generated should not be too close or too far in range.
- Collision must be minimised as much as possible.

Hash functions

- Direct
  - The integer key is the index (only possible if the hash table is big enough to hold data for every possible value).
- Modulo-division
  - Almost always the final step in any hash function, but can be used by itself.
- Folding
  - Cut up the value into smaller numbers which are added together to form the hash code (usually with other operations). One folding technique is Folding with Strings.

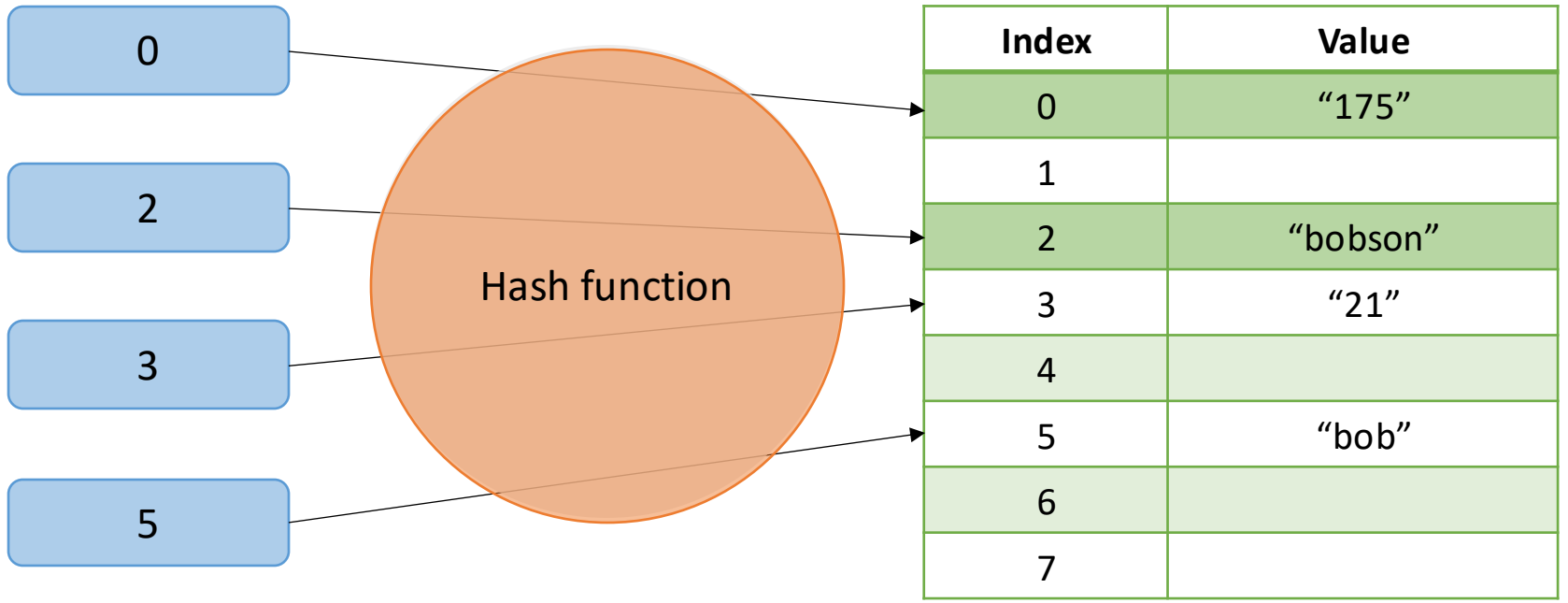
Direct Hash Function

The integer key is the index.



Direct Hash Function

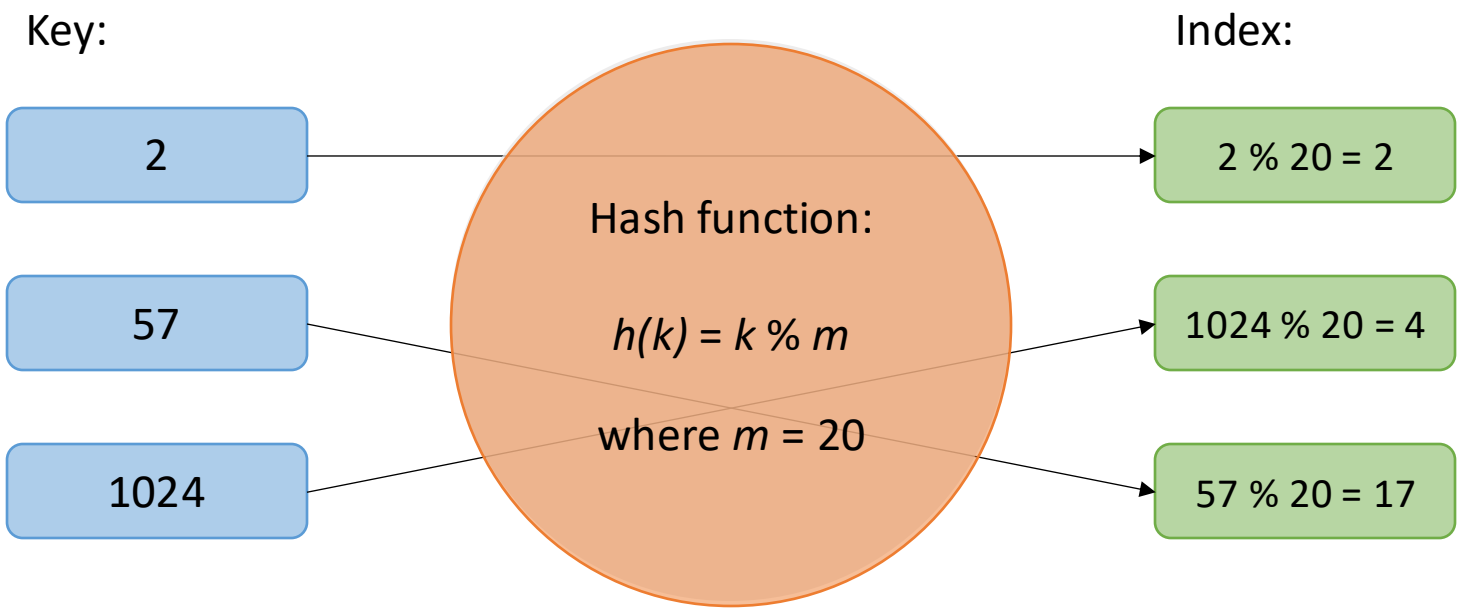
The integer key is the index. This assumes that keys are small, number of keys is not too large, and no two data have the same key.



Modulo-division

For a key *k* and size *m* of a hash table, the hash function *h()* is calculated as:

$$h(k) = k \% m$$



Modulo-division

Note, that there can be collisions! For example, any number that is a multiple of *m* will cause a collision.

i.e. let m = 20

20 % 20 = 0
100 % 20 = 0
320 % 20 = 0
etc.

If list size (*m*) is a prime number, then hash function produces fewer collisions.

Disadvantage is that consecutive keys map to consecutive hash codes leading to poor performance.

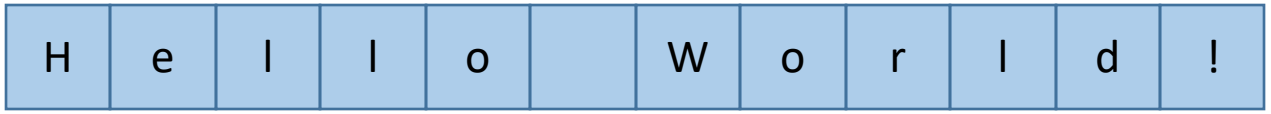
Folding with Strings

For a key *k* and size *m* of a hash table, the hash function *h()* is calculated by:

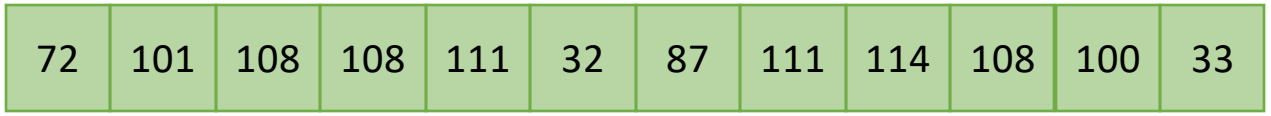
1. Converting the key k (which is a string) to its ascii values.
2. Grouping the values to a set size s.
3. Concatenate the numbers in each group.
4. Sum the numbers and then modulo by the size m of the table.

Folding with Strings

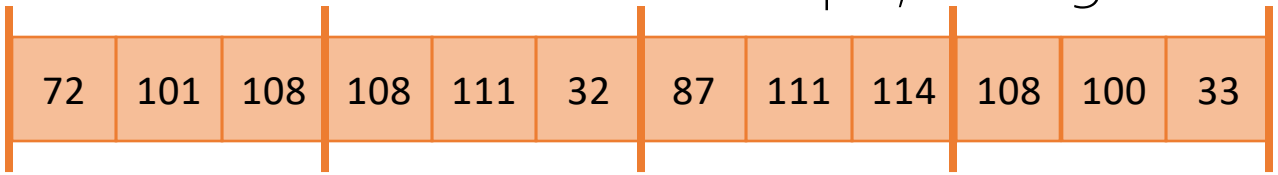
For example, for key *k* "Hello World" and size *m* of 100:



1. Converting the key k (which is a string) to its ascii values.



2. Grouping the values to a set size s. For example, *let s = 3*



3. Concatenate the numbers in each group.

4. Sum the numbers and then modulo by the size *m* of the table.

72101108 + 10811132 + 87111114 + 10810033 = 180833387 % 100 = 87

Key = "Hello World!"
Index = 87

More Hash functions

- Digit extraction
  - Selected digits are taken and recombined to form the hash code (usually with other operations).
- Rotation
  - When data tend to be sequential, their hash codes then tend to be sequential. You can spread them out more by moving the least significant digit of the code into the most significant position.
- Mid-square
  - A kind of randomisation, where the middle digits of the code are extracted and squared to produce a more or less random new number (note: this can also be used in a double has by taking the middle digits of the last hash code and squaring them to get a new one).

Rehashing

Rehashing

- Hash tables do not grow dynamically. They have a fixed size.
- So, what happens if the hash table is full? (or almost full)
  - Like an array or Set, we need to resize it
  - This is called resizing/rehashing
- This involves three steps
  1. Calculating the load factor
  2. Resizing the hash table
  3. Rehashing the values

Rehashing

Load factor:

- We do not wait for a hash table to be full before resizing/rehashing
- Instead, we calculate a *load factor*
- When the load factor hits a threshold, we resize/rehash
- Load factor = number of full rows / size of hash table
- We resize/rehash when the load factor is > 0.75

Rehashing

Resizing the hash table:

- When we have a load factor > 0.75, we resize/rehash the table
- General practice is to increase the size of the table two-fold (x 2)
- We do this by creating a new table twice the size of the original
- We then copy all of the items across, making sure to rehash each as we go

Rehashing

Rehashing items:

- When we create a new, larger table, if we simply copy the values over at their current index, they will not conform to the new hash code
- For example:

h(k) = k % m
where *m* = 5

Key	Index	Value
6	1	“a”
7	2	“b”
8	3	“c”

h(k) = k % m
where *m* = 10

Key	Index	Value
6	6	“a”
7	7	“b”
8	8	“c”

Rehashing

Rehashing items:

- When we create a new, larger table, if we simply copy the values over at their current index, they will not conform to the new hash code
- So, we need to pass their key through the hash function again before inserting them into the new table

Note, this means that, even though we convert their key to an index before inserting them, we still need to keep a copy of their key in case they need to be rehashed

Performance

Performance

- The goal of a hash table is to locate an item (datum) quickly, ideally in O(1) time i.e. one comparison.

- … Side note … what is O(1)? …

Side note: Big O Notation in a nutshell

- Big O Notation is a mathematical notation
- We are using it to describe the performance of a data structure
- Where we use *n* to represent the size of our data structure

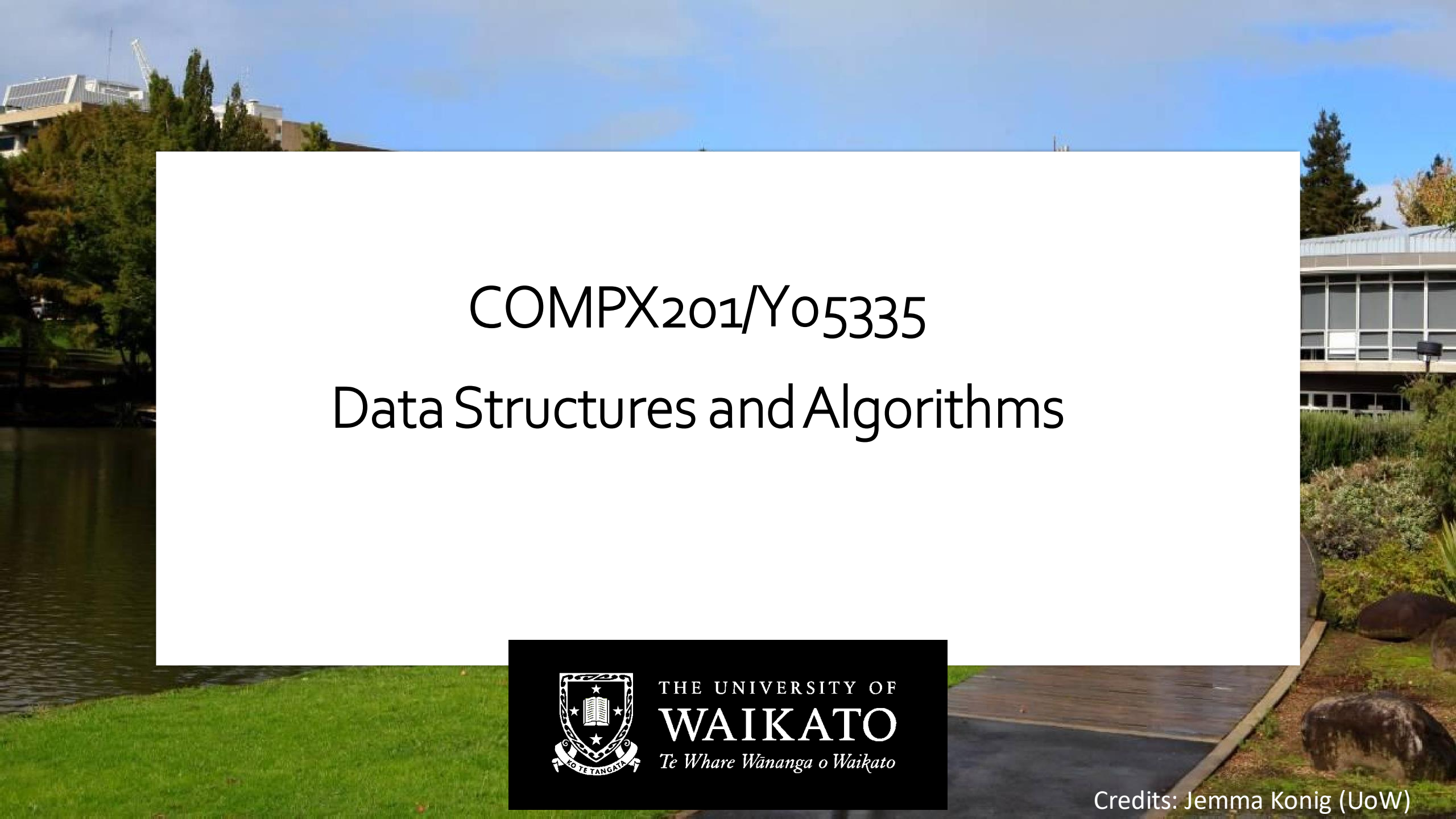
- i.e. Searching a BST is O(log(n)) because the number of nodes that are visit is halved with every step (i.e. logarithmic)
- Searching a linked-list would be O(n), because at most, we would need to visit every node in the list to find our value.
- O(1) means we only need to visit one item i.e. it is a direct look-up.

- More about Big O later in the course, but for now, let's get back to our Hash Tables ...

Performance

- The goal of a hash table is to locate an item (datum) quickly, ideally in O(1) time i.e. one comparison.
- Collisions may require additional comparisons, so minimising these improves performance.
- If a table becomes full a new bigger table is created and all items must be re-hashed. This can be time consuming so we don't want it to happen often.

Note: more on collisions and performance later!



COMPX201/Y05335

Data Structures and Algorithms

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Credits: Jemma Konig (UoW)

Hash Tables in Java

COMPX201/Y05335

Overview

- Defining our Hash Table
- Operations

Defining our Hash Table

Defining our custom Hash Table

- We are going to use an array of nodes for our Hash Table
- We will need to give the array an initial size
- Each node will have a key and a value

- Also need to keep track of:
  - The number of full entries
  - The number of collisions * *for later*
  - The number of times we have rehashed the table * *for later*

Defining our custom Hash Table

Okay, let's try it ...

Operations

Operations

- Print
- Hash Function
- Insert
- Delete
- Rehash/Resize

Print

- Print() = print the array
- Print(key) = print the value that is associated with given key
- … let's try it …

Hash Function

- Takes a key
- Returns an index
- But which Hash Function should we use?
- For simplicity, we'll use Modulo-division

- For a key *k* and size *m* of a hash table, the hash function *h()* is calculated as:
  $h(k) = k \% m$
- ... let's try it ...

Insert

- Given a key and a value
- Calculate the index
- Insert the value into the table at given index

- BUT what if the index is greater than the size of the table?
  - It can't be, that's how modulus works. Modulo will always return a value between 0 and *m*-1
- BUT what happens if there's already something there? i.e., we have a collision
  - For now, just override the value and increment collision count
- BUT what if the table is full?
  - For now, just increment the rehashes count
- … let's try it …

Delete

- Given a key
- Calculate the index
- Delete the value from the table at given index

- Make sure to decrement the number of full rows
- ... let's try it ...

Resize/Rehash

- What should we do if the table is full?
  - Create a new, larger table – twice the size of the original
  - Move all values from the old table to the new table
  - Note, will need to rehash the keys because the size of the table has changed!

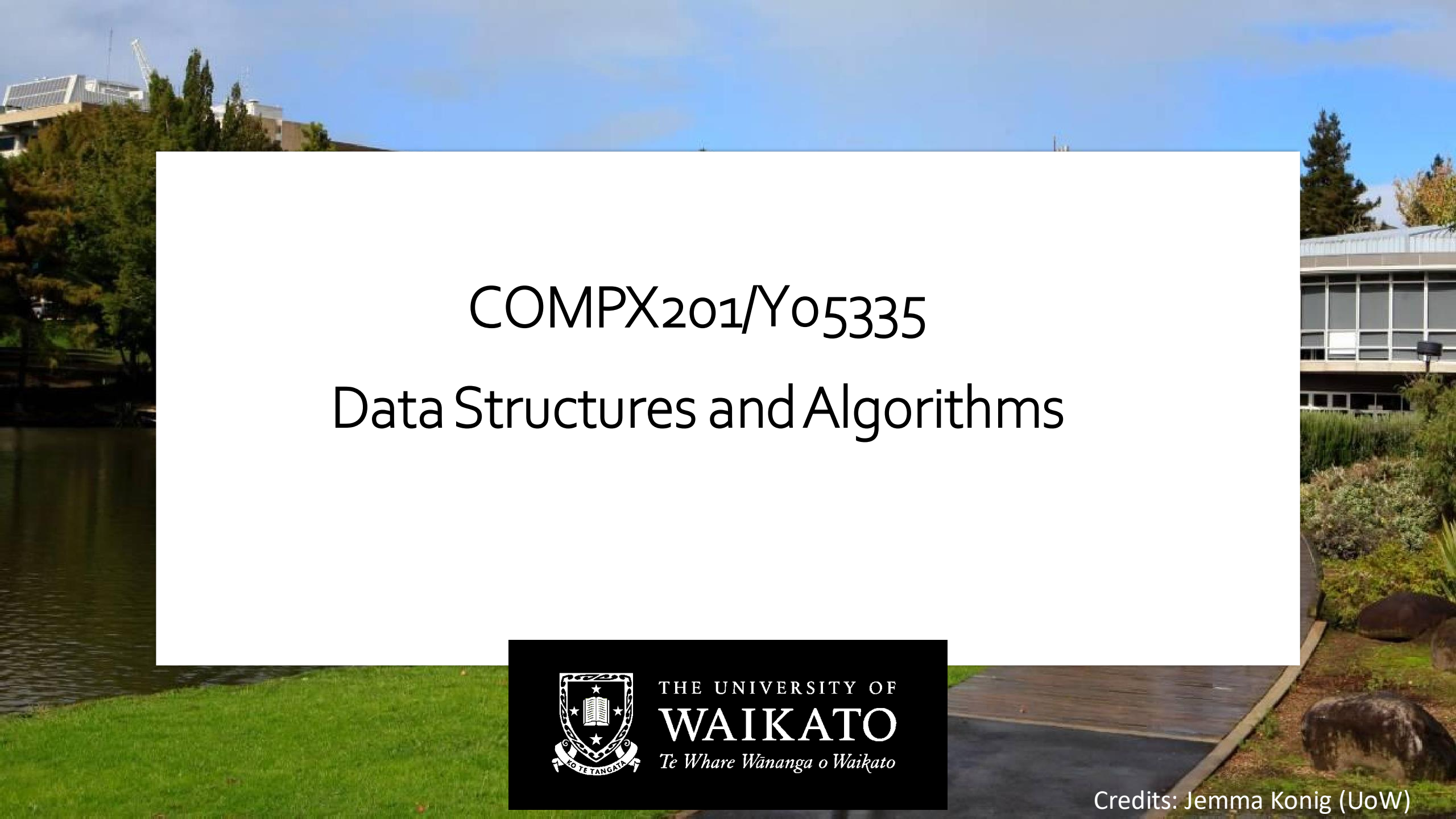
- How do we rehash the keys?
  - Use the insert method, passing the original key i.e., Node key

- Note, we shouldn't actually wait for the table to be completely full. This will result in a much higher chance of collisions. Instead, resize/rehash when the load factor gets to 75% capacity (0.75).

- … let's try it …

Collisions

- Next time …



COMPX201/Y05335

Data Structures and Algorithms

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Credits: Jemma Konig (UoW)