Maps and Dictionaries

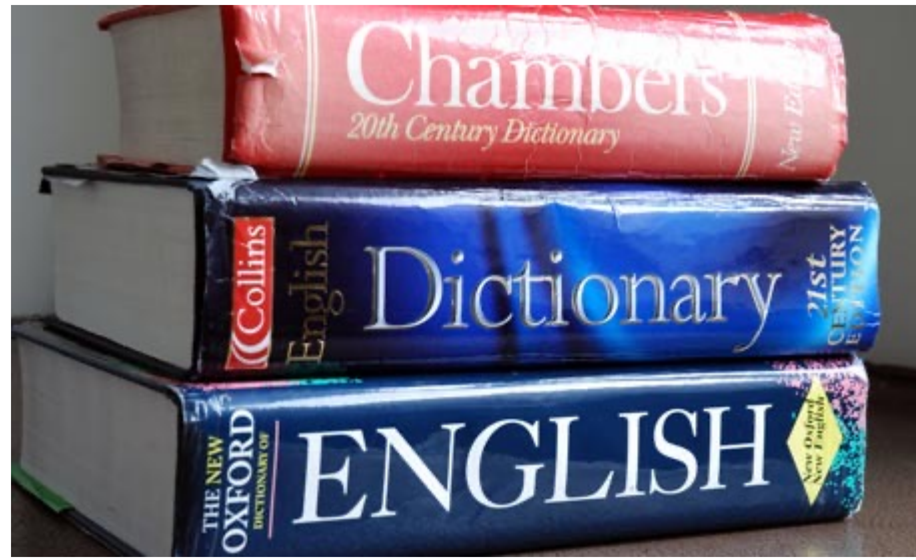
Storing and searching (key,value) pairs

An Introduction to Hash Tables

Dictionaries and Maps

A *dictionary*, or *map*, is a storage and look-up structure maintaining (key,value) pairs.

Formally, each *key* must be unique. Specifying the key allows us to retrieve the value from the structure.



Python provides the `dict` data type.

We will use the word *map* for the general concept.

From CS1112:

Let A and B be two non-empty sets. A function f from A to B specifies for each element of A exactly one element of B.

We write $f : A \to B$, and if f specifies b for a, we write $f(a)=b$.

Representing functions

T	M
mallow	Cork
killarney	Kerry
dungarvan	Waterford
limerick	Limerick
ennis	Clare
middleton	Cork
tralee	Kerry
bantry	Cork

... using a lookup table

Map ADT

Store elements (or (key,value) pairs)

getitem(key)	return the element with given key, or None if not there
setitem(key,value)	assign value to element with key; add new if needed
contains(key)	return True if map has some an element with key
delitem(key)	remove element with key, or return None if not there
length()	return the number of elements in the map

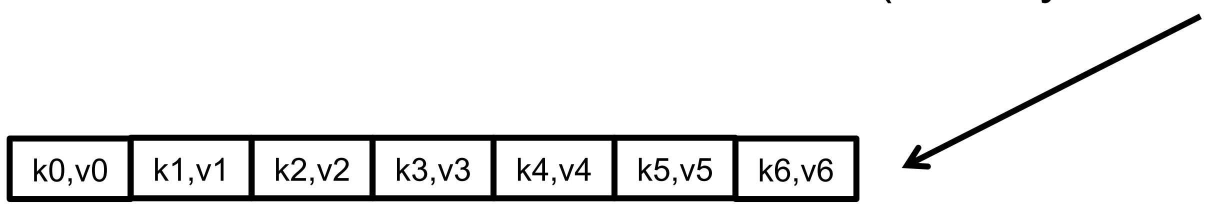
How should the Map ADT be implemented?

Store elements (i.e. (key,value) pairs)

getitem(key)	return the element with given key, or None if not there
setitem(key,value)	assign value to element with key; add new if needed
contains(key)	return True if map has some an element with key
delitem(key)	remove element with key, or return None if not there
length()	return the number of elements in the map

Unsorted list implementation of Map

append new elements here
(with Python list resizing)



getitem(k): search the unordered list to find Element with key k
contains(k): search the unordered list to find Element with key k

setitem(k,v): search the unordered list to find the key, and
change the value, or append Element(k,v) if the
key is not found
delitem(k): search the unordered list to find the key k, and pop the
element

Complexity

	getitem(k)	contains()	setitem(k,v)	delitem()	build full map
unsorted list	O(n)	O(n)	O(n)	O(n)	$O(n^2)$
others done on whiteboard ...					

Map search time

The operation we will want to do most often is to find an item (which is needed to read its value or to update it).

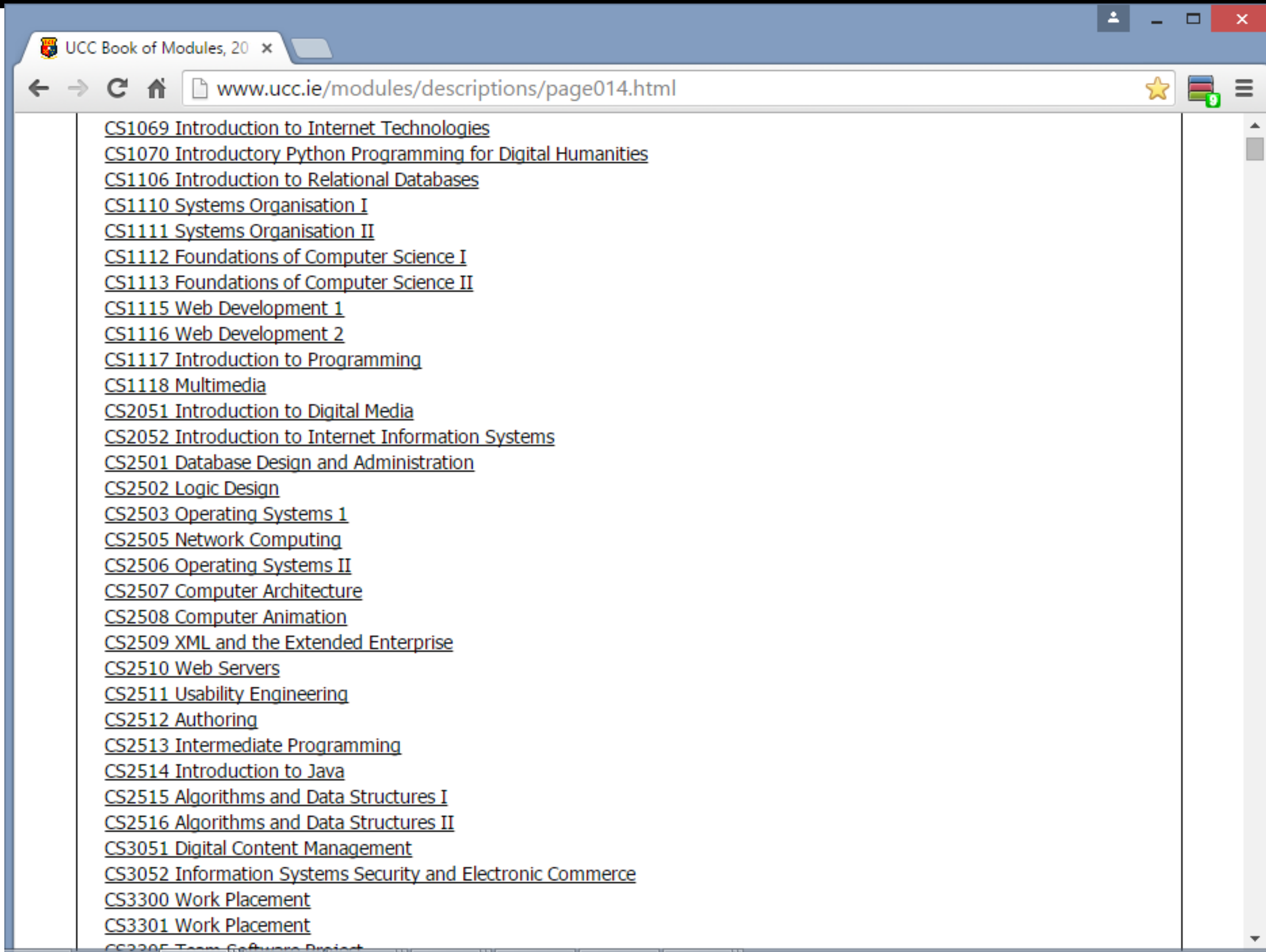
AVL trees offer O(log n) for searching. Can we do better?

In a standard Python list, we can access an item in O(1)

- if we know the item is in the list
- if we know the index of the item

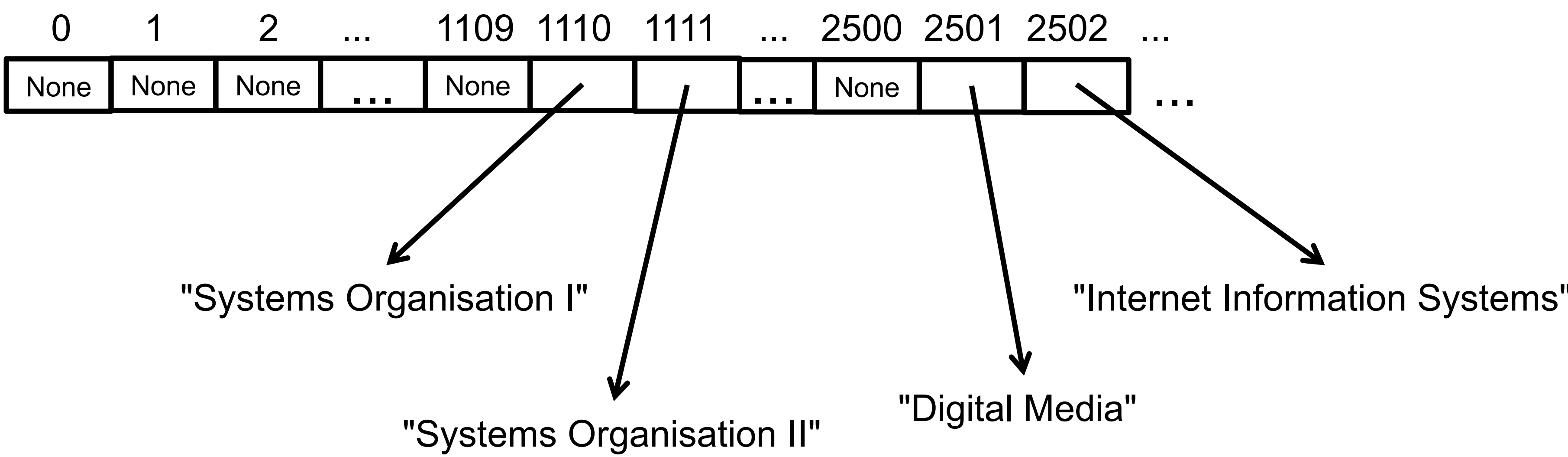
but in a standard map, we don't have this info.

Example: Book of Modules



Towards O(1) search time?

In the Book of Modules example, we could strip the letters off the front of the key, and use the numbers as the index.



Analysis

To represent just the 1st and 2nd year modules, we need up to 2999 cells in the array (to allow for new modules).

- but we only occupied 25 of them ...

For other applications, there won't be an obvious number for the key, and if there is, the number range might be massive compared to the elements.

Can we adapt the idea?

- use the structure of the key to identify the location
- use a more compact list, and use the size of the list to determine the location

Hash Tables

Maintain a list of known size, with explicit None items

To add a new item:
1. Compute an integer corresponding to the key
2. Compute an index in the list based on that integer and the list size
3. Store the item in the list at that index.

Line 1 requires a function from the set of keys to a subset of the integers.

Python's Hash function

The built-in function *hash()* will convert any hashable object into an integer.

Hash values must remain the same during an object's lifetime, and two objects which we want to evaluate as equivalent for dictionary lookup must have identical hash values.

All built-in immutable objects are hashable.

- strings, integers, floats, booleans, tuples

User-defined classes can include a __hash__ method, which should be defined on some immutable attribute(s)

- by default, Python will use the id or memory address of the object

Basic compression

Once we have computed a hash value, we need to decide on a location within our list.

Suppose our list has N cells:
- our location must be in range [0, N-1]

Suppose our key has hash value *hash(k)* = h

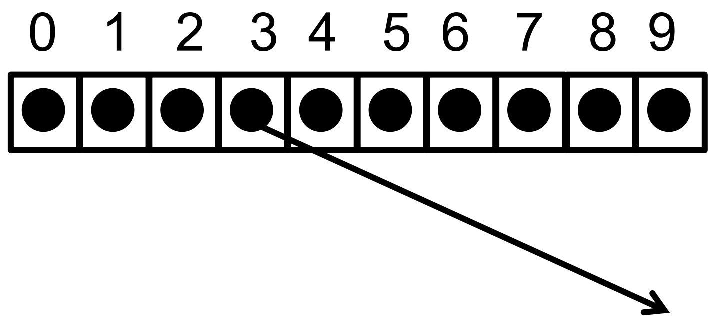
location(k) = h mod N (i.e. h % N in Python)

Determining this location is O(1), for a given list size N

```
location((k,v)) = hash(k) % N
```

```
item = Element('CS2515', 'Algorithms and Data Structures I')
```

hash('CS2515') evaluates to 1860694193
hash('CS2515') % 10 evaluates to 3




```
('CS2515','Algorithms and Data Structures I')
```

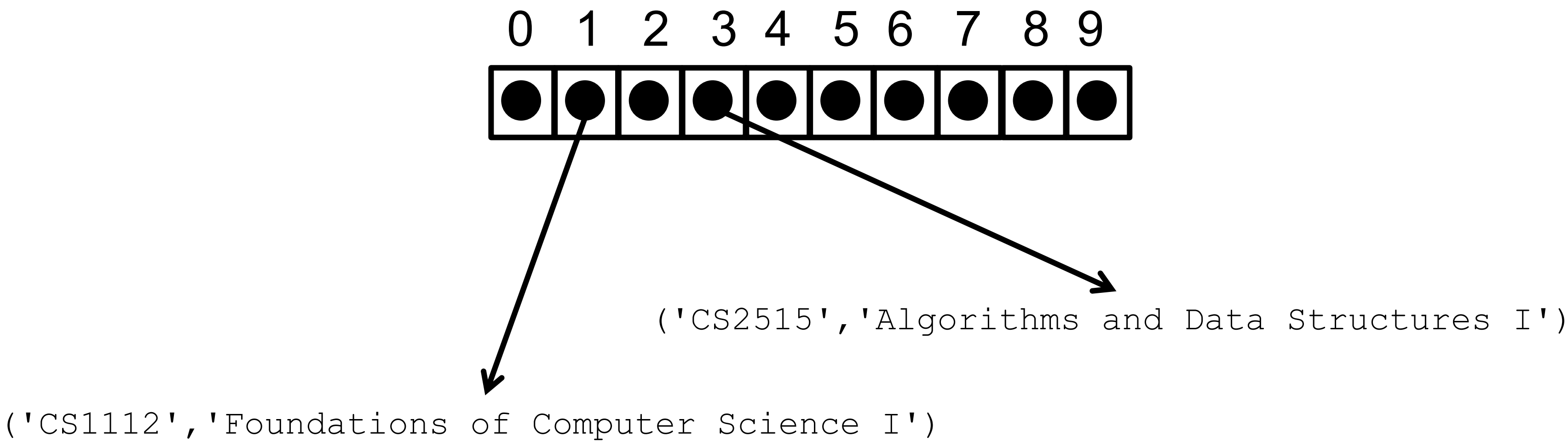
```
item = Element('CS1112', 'Foundations of Computer Science I')
```

hash('CS1112') evaluates to 2834091651

Where does Element('CS1112', 'Fou...') go?

```
location((k,v)) = hash(k) % N
```

Note if implementing in Python:
Python hash() uses a random seed, and so will give different values for each session (for security). If you want consistent hashing across different sessions, you need to set the seed (PYTHONHASHSEED) in your Python configuration.



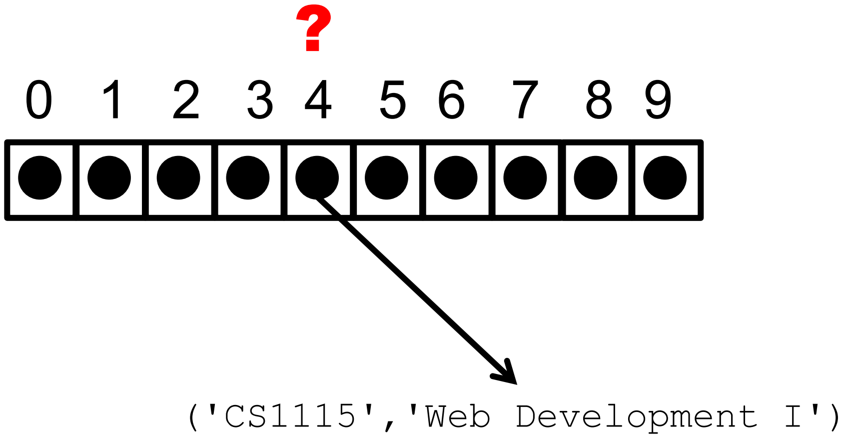
But ...

Our limited Book of Modules had 25 items, but we only have 10 cells in our list.

Some keys will be directed to the same location, and so if we try to insert them both, they will *collide*.

```
map.setitem('CS2514','Introduction to Java')
```

hash('CS2514') % 10 evaluates to 4



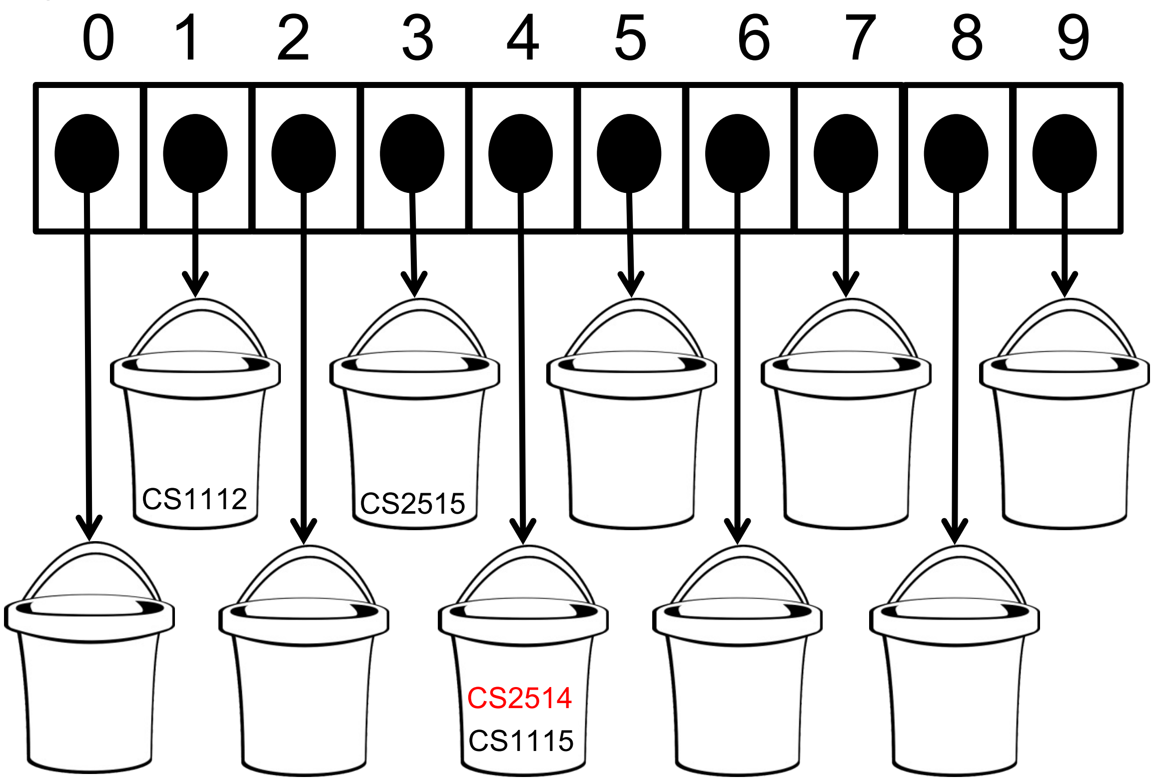
Separate Chaining

Each cell in the list will maintain its own list of values

- known as a *bucket* array

```
map.setitem('CS2514','Introduction to Java')
```

hash('CS2514') % 10 evaluates to 4



Next lecture

Collisions

Bucket arrays

More sophisticated compression

Using a flat structure while managing collisions