CompSci 105

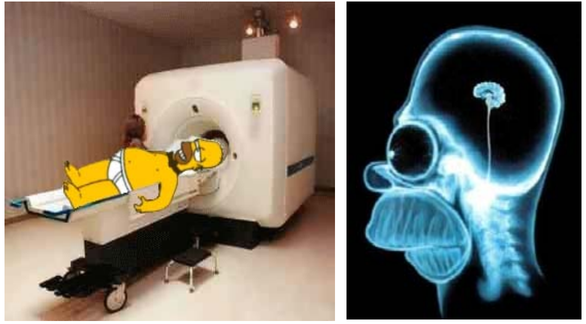
Part 3: Hashing, Sorting and Trees

Lecturer: Burkhard Wuensche

burkhard@cs.auckland.ac.nz
Phone 373-7599 83705
Office 303-529
Office hour: Open door policy, but better contact me to make sure I am around

Who is Burkhard?

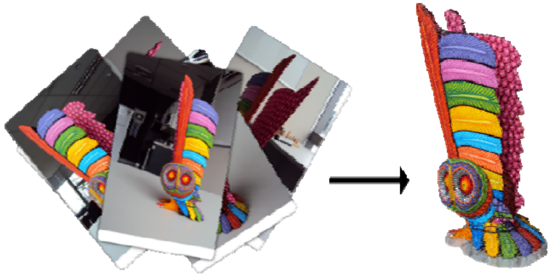
- Born in München (Germany)

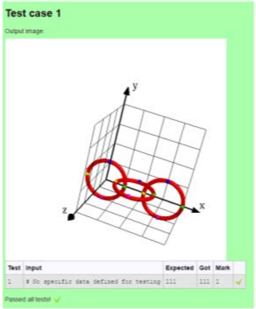


- Studied 3 years in Kaiserslautern (Germany)
- PhD in Biomedical Visualization
- Research Interests:
  - Computer Graphics, Biomedical Imaging, Scientific Visualization, Game Technology, Exergaming, Simulation Algorithms, Information Visualization, Human-Computer Interfaces, Human-Robot Interfaces, Augmented and Virtual Reality, Image-based modelling, Sketch-based modelling, CS Education

Burkhard Wünsche

Graphics Group
Department of Computer Science
The University of Auckland, New Zealand



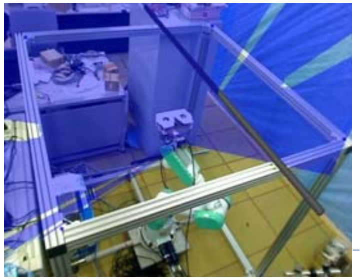


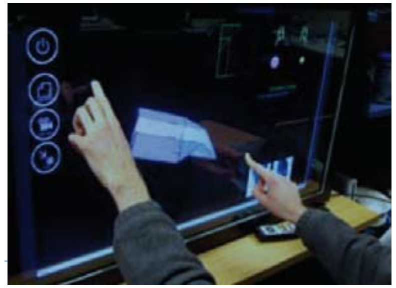
Test case 1

Output image:

Test	Input	Expected	Got	Mark	
1	# No specific data defined for testing	111	111	1	✓

Passed all tests! ✓





CompSci 105
Lecture 25-27 Content

Hashing

Motivation

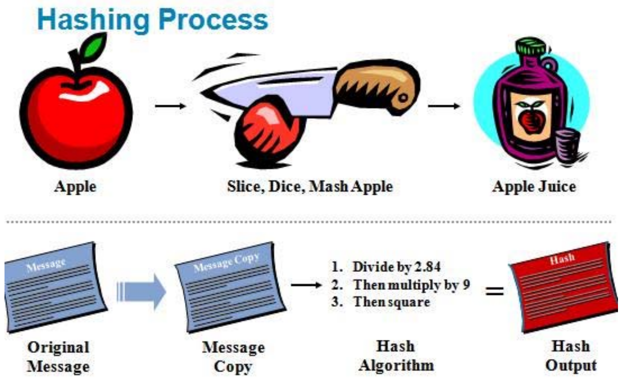
Hash Functions

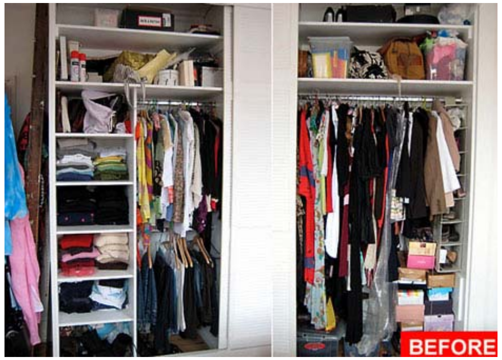
Collision Reduction

ADT & Implementation

Textbook: Chapter 5

(section 5.2.3)





Hashing = arrange data so that it can be accessed in constant time … like a perfectly sorted wardrobe

Agenda – Hashing (Lecture 1)

- Agenda
  - Hashing – Why?
  - Load Factor
  - Hash Functions – folding, mid-square
  - Hash Functions – keys that are strings
  - Collisions and Collision Resolution – introduction

Why hashing?

- For unsorted data it takes O(n) time to find or delete items (and O(1) to add items)

- For sorted data it takes O(log n) time to find items (and O(log n) to O(n) time to add or delete items depending on data structure)

- Is there a data structure where inserting, deleting and searching for items is more efficient?
  - Using a hash table we can, on average, insert, delete and search for items in constant time – O(1) !! ☺
    BUT: need extra memory, works best if size of data structure can be predicted, "encoding" data often non-trivial ["A good hash function is more an art than a science"], unsuitable for complex queries, e.g. "find k largest values" or "find closest value to X", often causes problems when using "caching" or "out of core computing", worst case O(n) [=> Can be exploited for denial of service attacks]

What is a Hash Table?

- A collection of items which are stored in such a way that the items are easy to access.
- Each position (slot) in the hash table can hold one item and is named (indexed) by an integer value starting from 0.
- Initially every slot is empty.

0	1	2	3	4	5	6	7	8	9	10	11	12
None	None	None	None	None	None	None	None	None	None	None	None	None

What is a Hash Function?

- Takes an item in the collection and returns a slot (i.e. an integer).

- The hash function is the mapping between an item and the slot where the item is stored
  - Ideally a hash function maps an item to a unique slot

0	1	2	3	4	5	6	7	8	9	10	11	12
None	None	None	None	None	None	None	None	None	None	None	None	None

Mapping an Item into a Hash Table Slot

Example:
- use a hash table of size 13
- items are integers (i.e. key is equal to item).
- Hash function is the key modulo the size m of the table

Key	Hash code (slot where to store)
54	54 mod 13 = 2
26	26 mod 13 = 0
94	94 mod 13 = 3
17	17 mod 13 = 4
77	77 mod 13 = 12
31	31 mod 13 = 5

0	1	2	3	4	5	6	7	8	9	10	11	12
26	None	54	94	17	31	None	None	None	None	None	None	77

This mapping uses the remainder method (i.e., key % 13).

Mapping an Item into a Hash Table Slot

- A hash function takes the key (which must be unique) of an item and returns a slot number in the hash table.
- Typically, hash functions are more complex than just the remainder function, and have ”% table_size (m)” as part of the formula since the resulting slot number must be within the range of the table size, i.e. in general:

  hash(item_key) = F(item_key) % m

  for some function F.

- The result of applying the hash function to the key is an index into the table.

Load Factor of the Hash Table

- The load factor (λ) of the hash table is the number of items in the table divided by the size of the table.

- The example hash table below has a load factor of

$$\lambda = 6 / 13$$

0	1	2	3	4	5	6	7	8	9	10	11	12
26	None	54	94	17	31	None	None	None	None	None	None	77

Search an Item

- Use the hash function to compute the slot of a given item and check whether or not it is present.
- This can be done in O(1)!
- E.g. For item with key 14, we have 14 mod 13 = 1. Since slot 1 is unoccupied, we conclude that 14 is not present.

0	1	2	3	4	5	6	7	8	9	10	11	12
26	None	54	94	17	31	None	None	None	None	None	None	77

Collisions

- Hash function:

$$hash(item\_key) = item\_key \ \% \ 13$$

- 6 items are mapped into the table below:

0	1	2	3	4	5	6	7	8	9	10	11	12
26	None	54	94	17	31	None	None	None	None	None	None	77

- Insert the item 44:

$$hash(44) = 44 \ \% \ 13 = 5$$

- Problem!
  - There is an item already in this slot!

- This is referred to as a collision (or a clash)

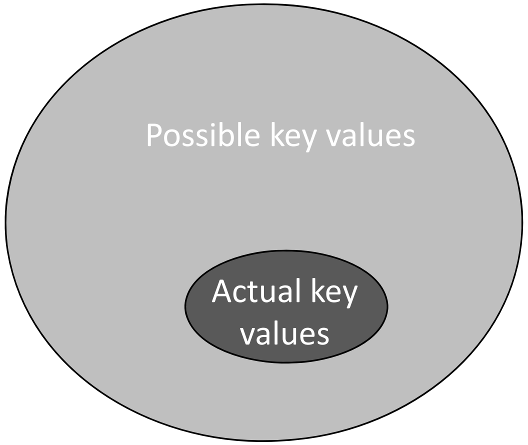
Perfect Hash Functions

- A hash function which uniformly distributes items over the whole hash table is a perfect hash function.
  - I.e. a "perfect hash function" is able to map m distinct items into a table of size n (≥m) with no collisions

- One way to achieve this is to have a hash table which is big enough to accommodate the full range of keys. If the keys were eight digit student ID numbers we would need an 10^8 sized table (from 00000000 to 99999999)
- This is usually very inefficient and often even infeasible

Perfect Hash Functions

- Trying to find a perfect hash function can be very wasteful as the number of items to be stored (and retrieved) may be much smaller than the actual table size.



We need some sort of compression from the full range of the keys into the number of hash table slots.

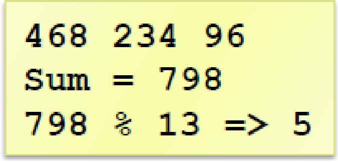
Good Hash Functions

- A good hash function should:
  - Be easy and fast to compute
  - Achieves even distribution of items (uniformity)
  - Ideally have a 1:1 correspondence between the number of items and the number of slots (i.e. size) of the hash table

- General requirements of a hash function:
  - The calculation of the hash function should involve the item value in its entirety
  - If a hash function uses modulo arithmetic, the base should be a prime number to help ensure even distribution of items

Hash Functions – The Folding Method

- Divides key into equal-size pieces (the last piece may not be of equal size).
  - Can compute the sum of these pieces or perform some computation on them.
- Example:
  - Keys are 8 digit phone numbers: 468-23496
  - Split into 3 numbers – 3 digits, 3 digits, and 2 digits
  - Find the sum of these numbers and use with hash function (% table_size).



```
468 234 96
Sum = 798
798 % 13 => 5
```

  - Note: we use all parts of the key in the calculation in case some parts of the key are very similar (which can result in collisions).

Hash Functions – The Mid Square Method

- Square the key and take some portion of the result.
- Example:
  - Square the item
  - Take all digits apart from the first
  - Take the modulus of the remaining number with the size of the table (13)

For keys:

key	key^2	Remove first	% 13
655	429025	29025	9
654	427716	27716	0
653	426409	26409	6

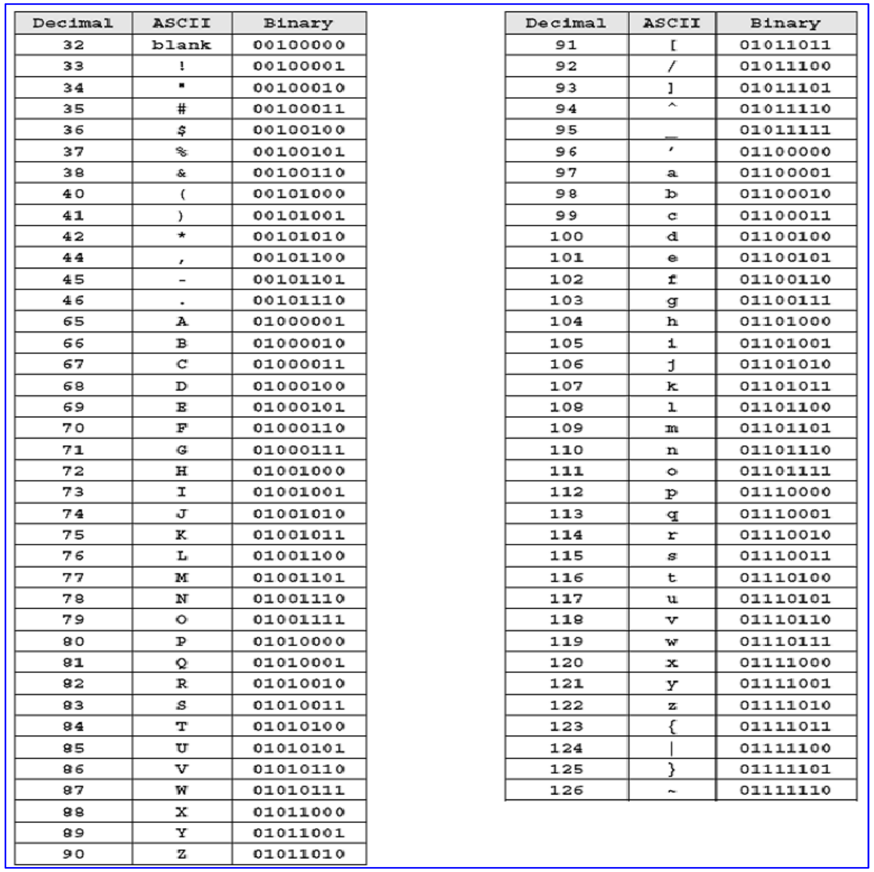
Hash Functions – Keys which are Strings

The ASCII table on the right shows the numerical representation of each character.

```
ord('a') is 97

ord('b') is 98

ord('c') is 99
```



Decimal	ASCII	Binary
32	blank	00100000
33	!	00100001
34	"	00100010
35	#	00100011
36	$	00100100
37	%	00100101
38	&	00100110
40	(	00101000
41	)	00101001
42	*	00101010
44	,	00101100
45	-	00101101
46	.	00101110
65	A	01000001
66	B	01000010
67	C	01000011
68	D	01000100
69	E	01000101
70	F	01000110
71	G	01000111
72	H	01001000
73	I	01001001
74	J	01001010
75	K	01001011
76	L	01001100
77	M	01001101
78	N	01001110
79	O	01001111
80	P	01010000
81	Q	01010001
82	R	01010010
83	S	01010011
84	T	01010100
85	U	01010101
86	V	01010110
87	W	01010111
88	X	01011000
89	Y	01011001
90	Z	01011010

Decimal	ASCII	Binary
91	[	01011011
92	/	01011100
93	]	01011101
94	^	01011110
95	_	01011111
96	'	01100000
97	a	01100001
98	b	01100010
99	c	01100011
100	d	01100100
101	e	01100101
102	f	01100110
103	g	01100111
104	h	01101000
105	i	01101001
106	j	01101010
107	k	01101011
108	l	01101100
109	m	01101101
110	n	01101110
111	o	01101111
112	p	01110000
113	q	01110001
114	r	01110010
115	s	01110011
116	t	01110100
117	u	01110101
118	v	01110110
119	w	01110111
120	x	01111000
121	y	01111001
122	z	01111010
123	{	01111011
124	|	01111100
125	}	01111101
126	~	01111110

Can we store String items?

- The ASCII values of the characters of the string can be used to compute the slot number into which the item is mapped.
- Example:
  - Add the ASCII value of each character in the key
  - Take the modulus of the result with the size of the table (13)

For key:

key	Add ASCII codes	Sum	% 13
"cat"	99 + 97 + 116	312	0

Exercise 1 – Hash function for string: Sum of ASCII codes

```
def hash1(key_word, table_size):
        sum = 0
        for pos in range(len(key_word)):
                sum = sum + ord(key_word[pos])
        return sum % table_size

def main():
  print("table size is 13")
  for key_wd in ["cat","dog","god","abracadabra","abraabracad"]:
        print(key_wd, hash1(key_wd, 13))
```

```
table size is 13
cat 0
dog 2
god 2
abracadabra 3
abraabracad 3
```

Using the above hashing algorithm, which kind of keys will cause collisions?

Exercise 2 – Hash function for string: Weighted sum of ASCII codes

- Improve the previous algorithm by adding a weighting to each character (1 for the first, 2 for the second, …).

```
def hash2(key_word, table_size):
        sum = 0
        for pos in range(len(key_word)):
                sum = sum + (pos+1) * ord(key_word[pos])
        return sum % table_size

def main():
  print("table size is 13")
  for key_wd in ["cat","dog","god","abracadabra","abraabracad"]:
        print(key_wd, hash2(key_wd, 13))
```

```
table size is 13
cat 4
dog 7
god 1
abracadabra 9
abraabracad 1
```

Hashing – Collisions

- Perfect hash functions are hard to come by, especially if you do not know the input keys beforehand.

- If multiple keys map to the same hash value this is called collision.
  - For non-perfect hash functions we need systematic way to handle collisions (=> collision resolution)

Exercise 3

- Insert the following items into the hash table below and indicate any collisions:
  - 11, 25, 63, 99, 12, 35, 54, 87, 66, 75, 91

- Hashing function:

$$h(item) = item \; \% \; 11$$

c c			c							c
11	**12**	**35**	**25**					**63**	**75**	**54**
0	1	2	3	4	5	6	7	8	9	10
99 **66**			**91**							**87**

Summary

- Using a hash table we can, on average (if table large enough and hash function suitable), insert, delete and search for items in constant time – O(1).
- The hash function is the mapping between an item and the slot where the item is stored.
- A collision occurs when an item is mapped to an occupied slot.
- A perfect hash function is able to map m items into a table of size m with no collisions.
- Perfect hash functions are hard to come by. Handling collisions systematically is required – collision resolution.

Agenda – Hashing (Lecture 2 & 3)

- Agenda
  - Collisions and Collision Resolution – open addressing methods, separate chaining
  - Map Abstract Data Type
  - Implementation of the Map Abstract Data Type
  - Using the **[]** syntax
  - Using the **del** Operator
  - Rehashing

Hashing – Collision Resolution

- Perfect hash functions are hard to come by, especially if you do not know the input keys beforehand.

- If multiple keys map to the same hash value this is called collision.
  - For non-perfect hash functions we need systematic way to handle collisions (=> collision resolution)

- One method is to systematically find an empty slot in the table, and put the value in this slot. This technique is called 'open addressing'. For example, start at the original hash value position (slot), look sequentially until you find a slot which is empty.

"open addressing" refers to the fact that the location ("address") of the item is not determined by its hash value.

Collision Resolution – Linear Probing

Look sequentially until an empty slot is found.

hash(key, 0) = key % m   #may be a different hash function
hash(key, 1) = (**hash(key, 0)** + 1) % m
hash(key, 2) = (**hash(key, 0)** + 2) % m
hash(key, 3) = (**hash(key, 0)** + 3) % m
…
hash(key, i) = (**hash(key, 0)** + i) % m

The number of probes is the number of attempts made until an empty slot position is found.

The probe sequence is the sequence of slots which are checked until an available slot is found.

Collision Resolution – Linear Probing

Example:

hash(key) = key % 13

0	1	2	3	4	5	6	7	8	9	10	11	12
26	None	54	94	17	31	None	None	None	None	None	None	77

Insert keys 44, 51 into the above hash table:

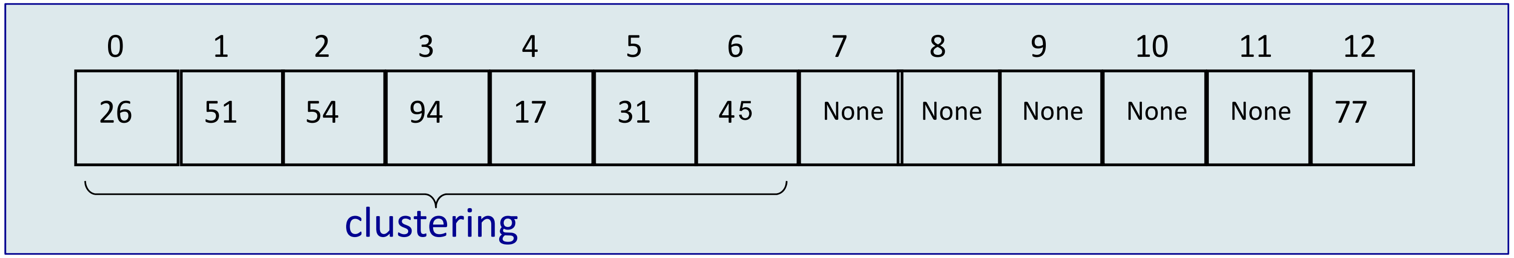
44 mod 13 = 5 Collision!
→ (5+1) mod 13 = 6

51 mod 13 = 12 Collision!
→ (12+1) mod 13 = 0 Collision!
→ (12+2) mod 13 = 1

0	1	2	3	4	5	6	7	8	9	10	11	12
26	51	54	94	17	31	44	None	None	None	None	None	77

Collision Resolution – Clustering

Clustering happens when regions of the table become very full and there are long runs of filled slots. Clustering slows down performance.



0	1	2	3	4	5	6	7	8	9	10	11	12
26	51	54	94	17	31	45	None	None	None	None	None	77

clustering

Another 'open addressing' approach: instead of looking for an empty slot sequentially, we skip slots, e.g. look at every third slot.

hash(k, i) = (hash(k, 0) + **3** * i) % m

"plus 3 probe"

Exercise. Repeat example from last slide with a plus 3 probe.

Collision Resolution – Linear Probing

Exercise:

hash(key) = key % 13

0	1	2	3	4	5	6	7	8	9	10	11	12
26	None	54	94	17	31	None	None	None	None	None	None	77

Insert keys 44, 51 into the above hash table using the “plus 3 probe”:

44 mod 13 = 5 Collision!
→ (5+3) mod 13 = 8

51 mod 13 = 12 Collision!
→ (12+3) mod 13 = 2 Collision!
→ (12+6) mod 13 = 5 Collision!
→ (12+9) mod 13 = 8 Collision!
→ (12+12) mod 13 = 11

0	1	2	3	4	5	6	7	8	9	10	11	12
26	None	54	94	17	31	None	None	44	None	None	51	77

Collision Resolution – Quadratic Probing

Another method of resolving collisions using 'open addressing‘. Instead of adding 1,2,3 etc. to the first hash result, add 1^2, 2^2, 3^2 etc.

hash(key , 0) = key % m   #may be different
hash(key , 1) = (hash(key, 0) + 1^2) % m
hash(key , 2) = (hash(key, 0) + 2^2) % m
hash(key , 3) = (hash(key, 0) + 3^2) % m
…
hash(key , i) = (hash(key , 0) + i^2) % m

The probe sequence is not a sequential list of numbers
⟶ reduces clustering

Collision Resolution – Quadratic Probing

Exercise:

hash(key) = key % 13

0	1	2	3	4	5	6	7	8	9	10	11	12
26	None	54	94	17	31	None	None	None	None	None	None	77

Insert keys 44, 51 into the above hash table using quadratic probing:

44 mod 13 = 5 Collision!
→ $(5+1^2)$ mod 13 = 6

51 mod 13 = 12 Collision!
→ $(12+1^2)$ mod 13 = 0 Collision!
→ $(12+2^2)$ mod 13 = 3 Collision!
→ $(12+3^2)$ mod 13 = 8

0	1	2	3	4	5	6	7	8	9	10	11	12
26	None	54	94	17	31	44	None	51	None	None	None	77

Collision Resolution – Double Hashing

We first looked at sequential linear probing (look sequentially until we find an empty slot).

→ prone to clustering

Improved ‘open addressing’ methods skip some slots (e.g. "plus-3 probing“) or use non-linear probing, e.g. quadratic probing.

→ clustering reduced, but still problem if many keys map to the same hash value

IDEA: Apply second hash function to key and use resulting value as our skip number for probing.

→ different keys have different probing sequences, even if initial slot was the same.

Collision Resolution – Double Hashing

Example: Use these two hash functions on the table below:

$hash_1(key) = key \% 13$
$hash_2(key) = 7 – key \% 7$

0	1	2	3	4	5	6	7	8	9	10	11	12
26	None	54	94	17	31	None	None	None	None	None	None	77

Inserting keys 43, 25 into the above hash table:

$h_1(43) = 4$ Collision
$h_2(43) = 6$ ⟶ next slot to try is 4+6=10 OK
(probe sequence is 4, 10)
$h_1(25) = 12$ Collision
$h_2(25) = 3$ ⟶ probe sequence is 12, 2, 5, 8 OK

0	1	2	3	4	5	6	7	8	9	10	11	12
26	None	54	94	17	31	None	None	25	None	43	None	77

Collision Resolution – Separate Chaining

Another way of handling collisions is to use chaining where every element of the hash table is a list and any items which are hashed to a slot are added to the list.

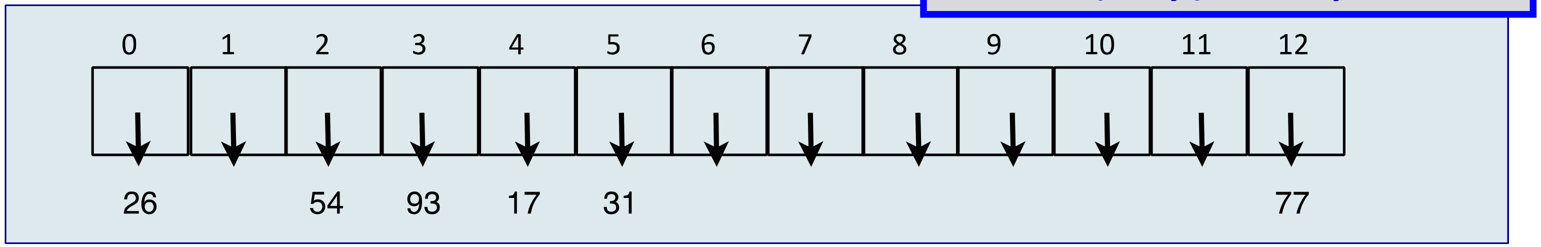
If the hash function is good and if the table has a load factor which is reasonable, the lists in each node of the hash table will be quite small. Therefore the Big O for inserting, deleting or searching for an item will be close to O(1).

Each element of the hash table could be a linked list or a Python list object.

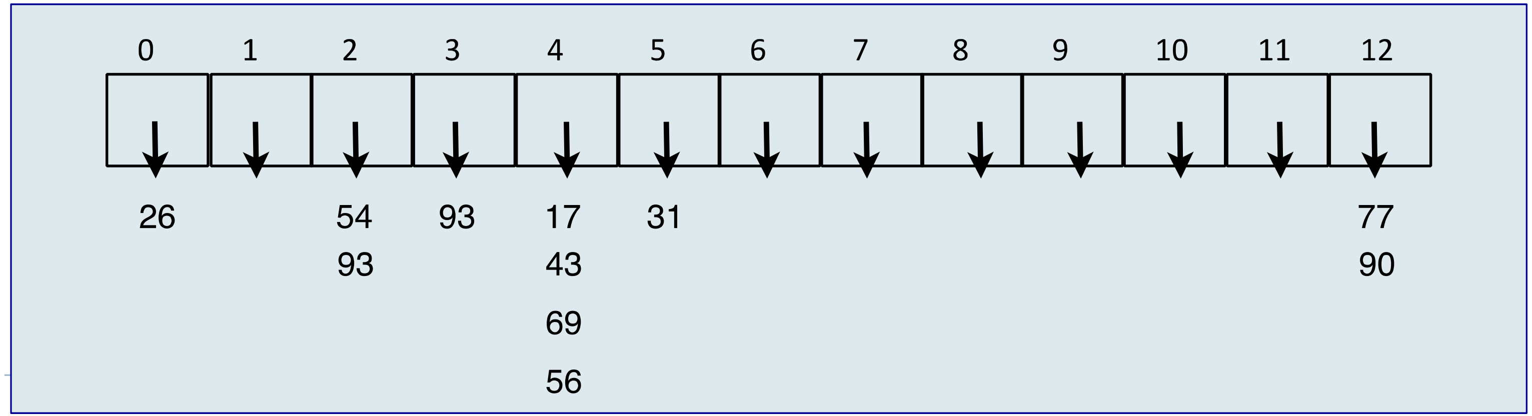
Collision Resolution – Separate Chaining

Example:

hash(key) = key % 13



Insert the keys: 43, 69, 93, 56, 90



Map Abstract Type

Operations of a Map ADT:

```
put(key, value)
get(key)
del map[key]
len()
in          #contains a given key
```

The Python dictionary stores key-data pairs where the key is unique. The key is used to look up the associated data value. The Python dictionary is an implementation of the Map ADT. Example:

```
phone_ext = {'David':1410,'Brad':1137,"Sarah":2830,"Chika":1345}
phone_ext["Lia"] = 1123
print('Brad' in phone_ext)                  # Output: True
print(phone_ext["Sarah"])                   # Output: 2830
del phone_ext["Brad"]
print(len(phone_ext))                       # Output: 4
```

Map ADT – An Implementation

We will use two parallel Python lists, one for the slot numbers corresponding to the keys and one for the associated data.

We are using linear probing to resolve collisions. Initially the table size is 11:

hash(key) = key % 11

	0	1	2	3	4	5	6	7	8	9	10
slots	None	None	None	None	None	None	None	None	None	None	None

	0	1	2	3	4	5	6	7	8	9	10
data	None	None	None	None	None	None	None	None	None	None	None

After all the items have been inserted:

	0	1	2	3	4	5	6	7	8	9	10
slots	77	44	55	20	26	93	17	None	None	31	54

	0	1	2	3	4	5	6	7	8	9	10
data	'bird'	'goat'	'pig'	'chicken'	'dog'	'lion'	'tiger'	None	None	'cow'	'cat'

```
h = HashTable()
h[54] = "cat"
h[26] = "dog"
h[93] = "lion"
h[17] = "tiger"
h[77] = "bird"
h[31] = "cow"
h[44] = "goat"
h[55] = "pig"
h[20] = "chicken”
```

Map ADT – An Implementation

1. In this implementation we are using the hash function:

hash(key) = key % size

```
def hash_function(self, key, size):
    return key % size
```

Whenever we add an item we need to call the hash function:

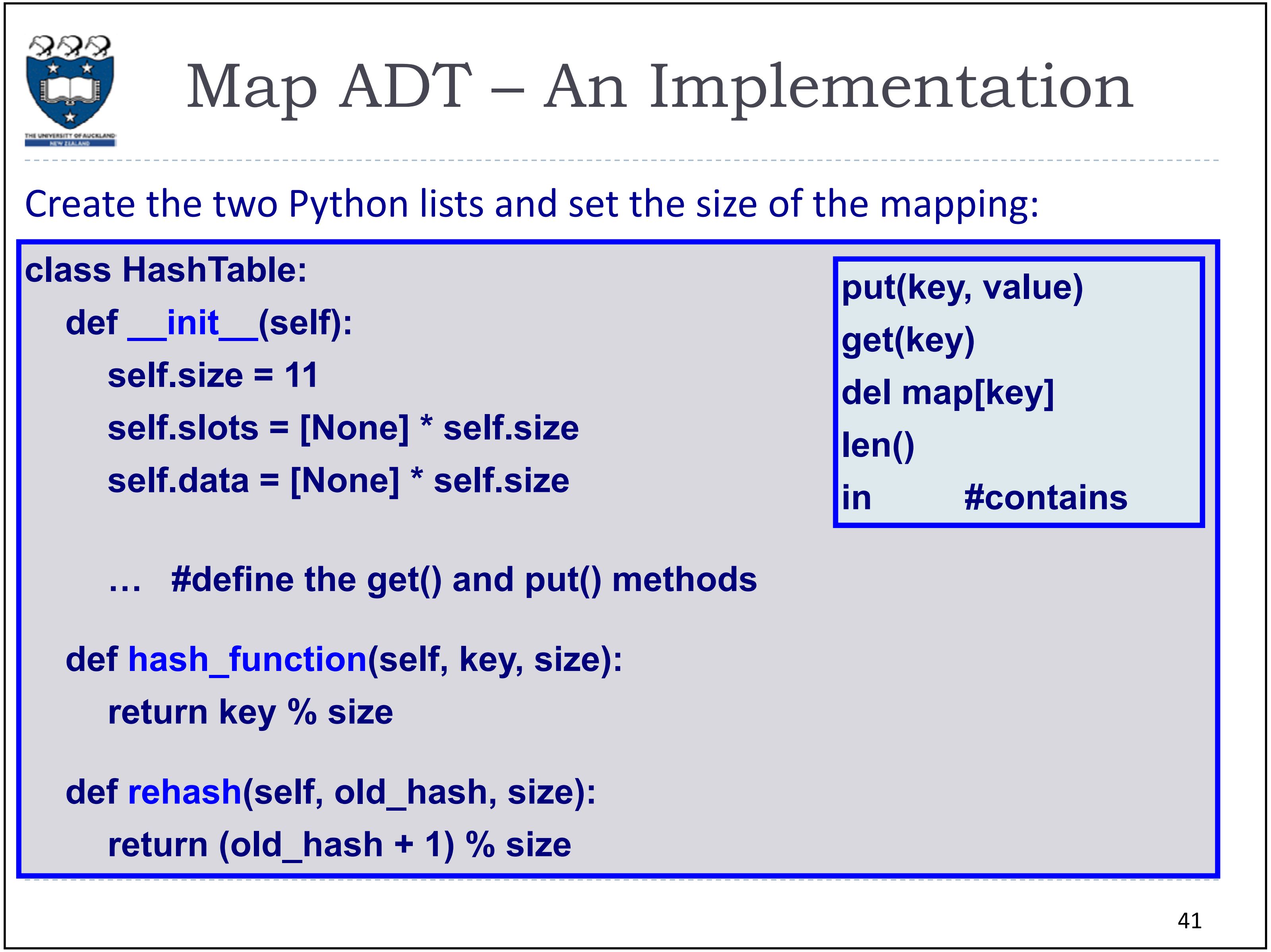
```
hash_value = self.hash_function(key, len(self.slots))
```

2. We will resolve collisions using linear probing, i.e., a step size of 1.

```
def rehash(self, old_hash, size):
    return (old_hash + 1) % size
```

Whenever there is a collision we need to get the next slot to try:

```
next_slot = self.rehash(next_slot, size)
```



Map ADT – An Implementation

Create the two Python lists and set the size of the mapping:

```
class HashTable:
    def __init__(self):
        self.size = 11
        self.slots = [None] * self.size
        self.data = [None] * self.size

        …   #define the get() and put() methods

    def hash_function(self, key, size):
        return key % size

    def rehash(self, old_hash, size):
        return (old_hash + 1) % size
```

```
put(key, value)
get(key)
del map[key]
len()
in          #contains
```

Map ADT – An Implementation

Getting the associated value of an entry in the hash table:

```
def get(self, key):
    start_slot = self.hash_function(key, len(self.slots))
    position = start_slot

    while self.slots[position] != None:
        if self.slots[position] == key:        # key found
            return self.data[position]         # return associated data
        else:
            position = self.rehash(position, len(self.slots))
            if position == start_slot:         # all slots in hash table searched
                return None                    # → key not in table

    return None                                # empty slot → key not in table
```

Map ADT – An Implementation

Putting an entry (key-value pair) into the hash table:

```
def put(self, key, data):
    hash_value = self.hash_function(key, len(self.slots))
    if self.slots[hash_value] == None:
        self.slots[hash_value] = key
        self.data[hash_value] = data
    elif self.slots[hash_value] == key:
        self.data[hash_value] = data
    else:
        next_slot = self.rehash(hash_value, len(self.slots))
        while self.slots[next_slot] != None and self.slots[next_slot] != key:
            next_slot = self.rehash(next_slot, len(self.slots))
            if next_slot == hash_value:
                return
        if self.slots[next_slot] == None:
            self.slots[next_slot] = key
            self.data[next_slot] = data
        else:
            self.data[next_slot] = data
```

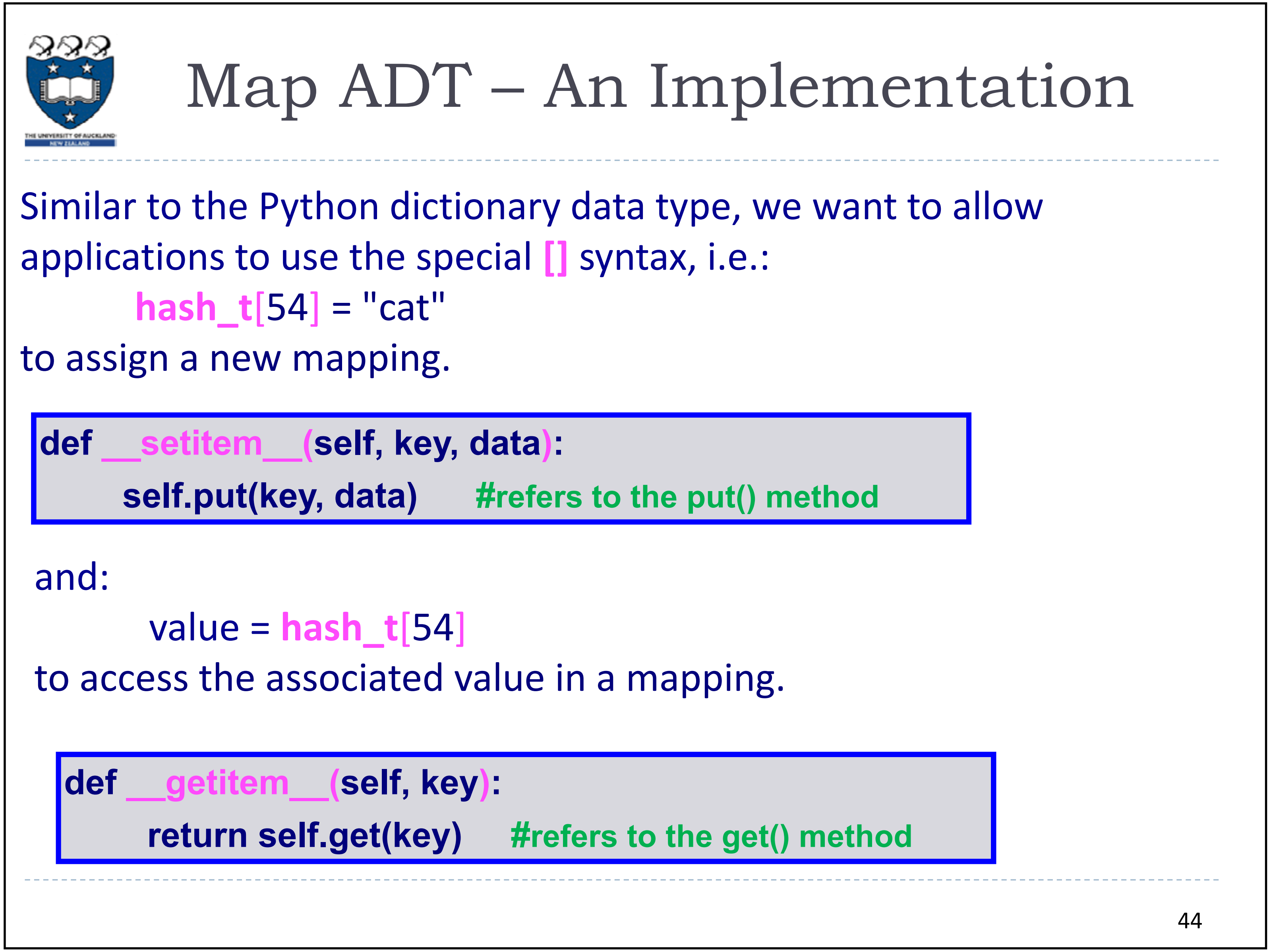
Put the key and associated data into the lists

Replace the associated data

Hash table full, cannot add data

Put the key and associated data into the lists

Replace the associated data



Map ADT – An Implementation

Similar to the Python dictionary data type, we want to allow applications to use the special [] syntax, i.e.:

hash_t[54] = "cat"

to assign a new mapping.

```
def __setitem__(self, key, data):
      self.put(key, data)      #refers to the put() method
```

and:

value = hash_t[54]

to access the associated value in a mapping.

```
def __getitem__(self, key):
      return self.get(key)     #refers to the get() method
```

Map ADT – An Implementation

The implementation now allows the use of the **[]** syntax.

```
class HashTable:
    def __init__(self):
        self.size = 11
        self.slots = [None] * self.size
        self.data = [None] * self.size
    def put(self, key, data):
        …
    def get(self, key):
        …
    def __setitem__(self, key, data):
        self.put(key, data)
    def __getitem__(self, key):
        return self.get(key)
  …
```

```
hash_t = HashTable()
hash_t[54] = "cat"
print(hash_t[54])
```

HashTable – Deleting a key-value pair

Deleting a value is non-trivial because of collisions (see next slides).

Case 1: key is NOT in the table:

Apply hash function. The field is either ‘None’ (we can return) or occupied by another key. In that case we look sequentially (linear probing) until we find an element which is ‘None’.

Example: look for hash[23], we apply the hash function and look in slot 1, then in slots 2, 3, 4, 5, 6, 7. Since slot 7 is 'None’ we know the key 23 is not in the table and we do not need to look any further.

	0	1	2	3	4	5	6	7	8	9	10
slots	77	44	55	20	26	93	17	None	None	31	54
data	'bird'	'goat'	'pig'	'chicken'	'dog'	'lion'	'tiger'	None	None	'cow'	'cat'

HashTable – Deleting a key-value pair

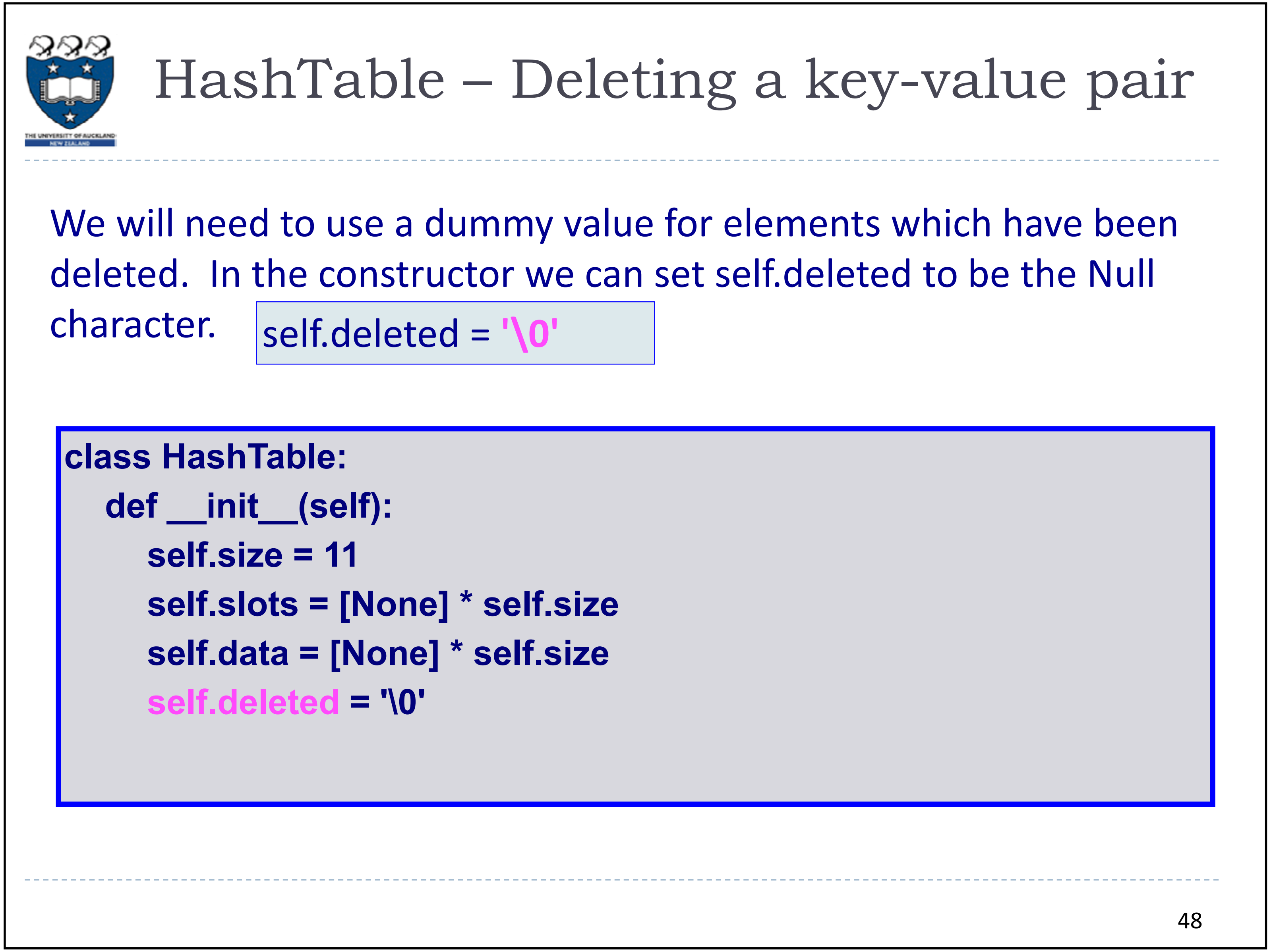
Case 2: key is in the table:

Assume we wish to delete hash[55]. We apply the hash function and look in slot 0, then we look in slots 1, 2. We find key 55 and delete it.

	0	1	2	3	4	5	6	7	8	9	10
slots	77	44	None	20	26	93	17	None	None	31	54

	0	1	2	3	4	5	6	7	8	9	10
data	'bird'	'goat'	None	'chicken'	'dog'	'lion'	'tiger'	None	None	'cow'	'cat'

BUT: What happens if we now wish to find key 20? (20%11=9)
Because of collisions it has been entered into slot 3. But because slot 2 is now empty (after deleting 55), we will not find key 20 anymore.



HashTable – Deleting a key-value pair

We will need to use a dummy value for elements which have been deleted. In the constructor we can set self.deleted to be the Null character.

`self.deleted = '\0'`

```
class HashTable:
    def __init__(self):
        self.size = 11
        self.slots = [None] * self.size
        self.data = [None] * self.size
        self.deleted = '\0'
```

HashTable – Deleting a key-value pair

The delete() method:

```
def delete(self, key):
    start_slot = self.hash_function(key, len(self.slots))
    position = start_slot
    key_in_slot = self.slots[position]

    while key_in_slot != None:
      if key_in_slot == key:
         self.slots[position] = self.deleted
         self.data[position] = self.deleted
         return None
      else:
         position = self.rehash(position, len(self.slots))
         key_in_slot = self.slots[position]
         if position == start_slot:
            return None
```

Will continue to search even if the slot contains self.deleted. Only stops if slot is None.

Key not in table – do nothing and return

HashTable – Deleting a key-value pair

The __delitem__(...) allows the use of the **del** operator.

```
def delete(self, key):
        # see previous slide

def __delitem__(self, key):
        return self.delete(key)
```

```
h = HashTable()
h[54] = "cat"
h[31] = "cow"
h[44] = "goat"
del h[44]
del h[54]
```

HashTable – Updating put() function

The **put() function** needs to be updated to take into account **self.deleted**

```
def put(self,key,data):
       hash_value = self.hash_function(key,len(self.slots))
       if self.slots[hash_value] == None or \
          self.slots[hash_value] == self.deleted:
           self.slots[hash_value] = key
           self.data[hash_value] = data
       elif self.slots[hash_value] == key:
           self.data[hash_value] = data
       else:
           next_slot = self.rehash(hash_value, len(self.slots))
           while self.slots[next_slot] != None\
                 and self.slots[next_slot] != self.deleted \
                 and self.slots[next_slot] != key:
               next_slot = self.rehash(next_slot,len(self.slots))
               if next_slot == hash_value:
                   return
           if self.slots[next_slot] == None or \
              self.slots[next_slot] == self.deleted:
               self.slots[next_slot] = key
               self.data[next_slot] = data
           else:

               self.data[next_slot] = data
```

The 'in' and 'len' Operators

The __len__(...) allows the use of the **len** operator.
The __contains__(...) allows the use of the **in** operator.

```
def __len__(self):
    count = 0
    for value in self.slots:
        if value != None and value != self.deleted:
            count += 1
    return count

def __contains__(self, key):
    return self.get(key) != None
```

Hashing Analysis

The load factor (λ) of the hash table is the number of items in the table divided by the size of the table.

If λ is small then keys are more likely to be mapped to slots where they belong and searching will be O(1).

If λ is large then collisions are more likely and more comparisons (is the slot available or not) are needed to find an empty slot.

Rehashing

The load factor (λ) of the hash table is the number of items in the table divided by the size of the table.

If the load factor gets to high performance slows down significantly. In that case the easiest solution is to copy the entire hash table into a larger table (rehashing).

For separate chaining the load factor should not exceed **0.75**.
For open addressing, the load factor should not exceed **0.5**.

NOTE 1: Rehashing a table is expensive (since elements must be inserted using the new hash function) – do only occasionally, e.g. double size of table each time, but make sure size is a prime number.

Rehashing - Exercise

0	1	2	3	4	5	6
56	43	30	None	None	26	13

Rehash the above table into the hash table below using the hash function: hash(key) = key % 13 and quadratic probing.

0	1	2	3	4	5	6	7	8	9	10	11	12
26	13	None	None	56	43	None	None	30	None	None	None	None