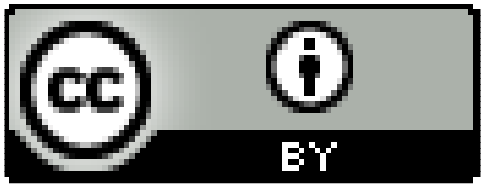
Sets and Dictionaries

Storage



Let's try an experiment

Let's try an experiment

```
>>> things = set()
>>> things.add('a string')
>>> print things
set(['a string'])
```

Let's try an experiment

```
>>> things = set()
>>> things.add('a string')
>>> print things
set(['a string'])

>>> things.add([1, 2, 3])
TypeError: unhashable type: 'list'
```

Let's try an experiment

```
>>> things = set()
>>> things.add('a string')
>>> print things
set(['a string'])

>>> things.add([1, 2, 3])
TypeError: unhashable type: 'list'
```

What's wrong?

Let's try an experiment

```
>>> things = set()
>>> things.add('a string')
>>> print things
set(['a string'])

>>> things.add([1, 2, 3])
TypeError: unhashable type: 'list'
```

What's wrong?

And what does the error message mean?

How are sets stored in a computer's memory?

How are sets stored in a computer's memory?

Could use a list

How are sets stored in a computer's memory?

Could use a list

```
def set_create():
  return []
```

How are sets stored in a computer's memory?

Could use a list

```
def set_create():
  return []


def set_in(set_list, item):
  for thing in set_list:
    if thing == item:
      return True
  return False
```

```
def set_add(set_list, item):
  for thing in set_list:
    if thing == item:
      return
  set.append(item)
```

```
def set_add(set_list, item):
  for thing in set_list:
    if thing == item:
      return
  set.append(item)
```

How efficient is this?

```
def set_add(set_list, item):
  for thing in set_list:
    if thing == item:
      return
  set.append(item)
```

How efficient is this?

With N items in the set, `in` and `add` take 1 to N steps

```
def set_add(set_list, item):
  for thing in set_list:
    if thing == item:
      return
  set.append(item)
```

How efficient is this?

With N items in the set, `in` and `add` take 1 to N steps

"Average" is N/2

```
def set_add(set_list, item):
  for thing in set_list:
    if thing == item:
      return
  set.append(item)
```

How efficient is this?

With N items in the set, `in` and `add` take 1 to N steps

"Average" is N/2

It's possible to do *much* better

```
def set_add(set_list, item):
  for thing in set_list:
    if thing == item:
      return
  set.append(item)
```

How efficient is this?

With N items in the set, `in` and `add` take 1 to N steps

"Average" is N/2

It's possible to do *much* better

But the solution puts some constraints on programs

Start simple: how do we store a set of integers?

Start simple: how do we store a set of integers?

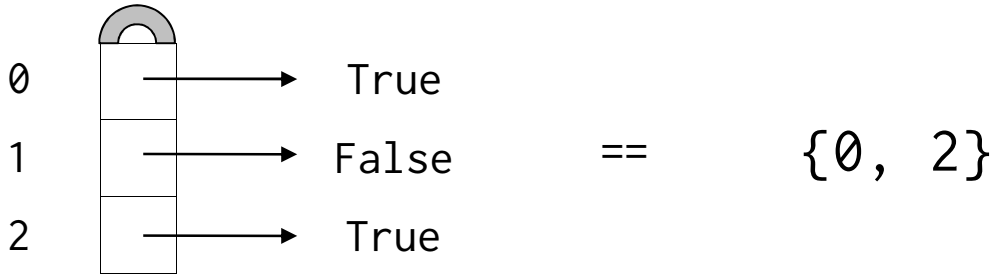
If the range of possible values is small and fixed,

use a list of Boolean flags ("present" or "absent")

Start simple: how do we store a set of integers?

If the range of possible values is small and fixed,

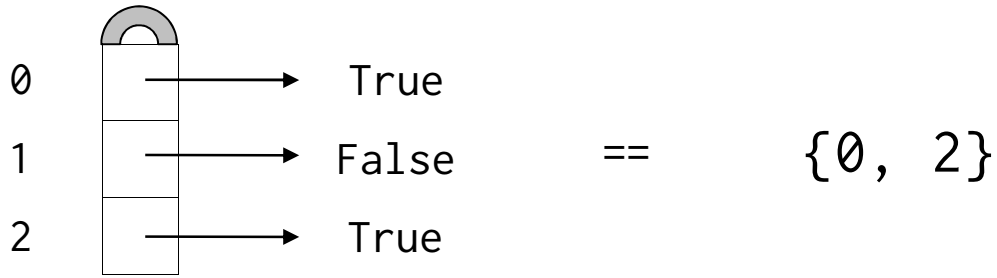
use a list of Boolean flags ("present" or "absent")



Start simple: how do we store a set of integers?

If the range of possible values is small and fixed,

use a list of Boolean flags ("present" or "absent")



But what if the range of values is large, or can

change over time?

Use a fixed-size *hash table* of length L

Use a fixed-size *hash table* of length L

Store the integer I at location I % L

Use a fixed-size *hash table* of length L

Store the integer I at location I % L

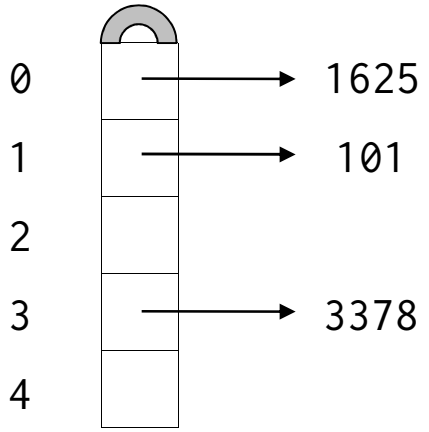
'%' is the remainder operator

Use a fixed-size *hash table* of length L

Store the integer I at location I % L

'%' is the remainder operator

```
{3378, 1625, 101}      ==
```



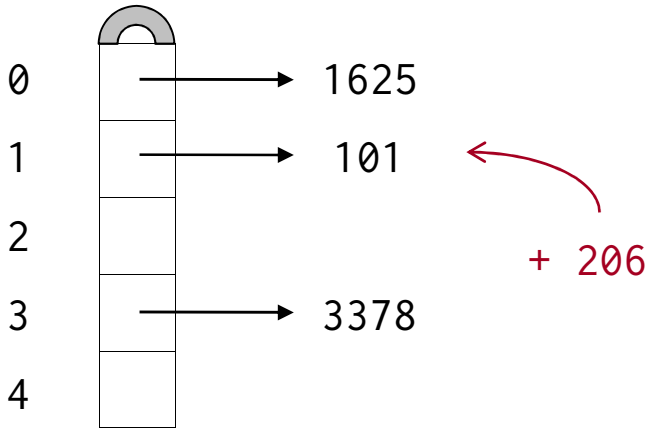
Time to insert or look up is constant (!)

Time to insert or look up is constant (!)

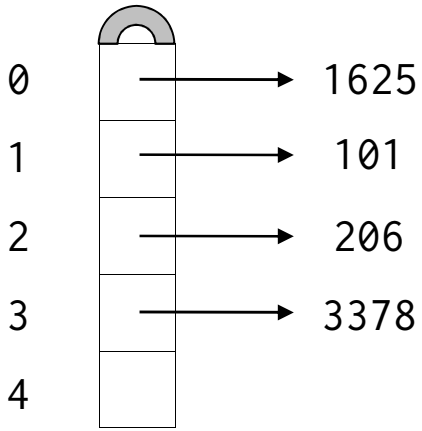
But what do we do when there's a collision?

Time to insert or look up is constant(!)

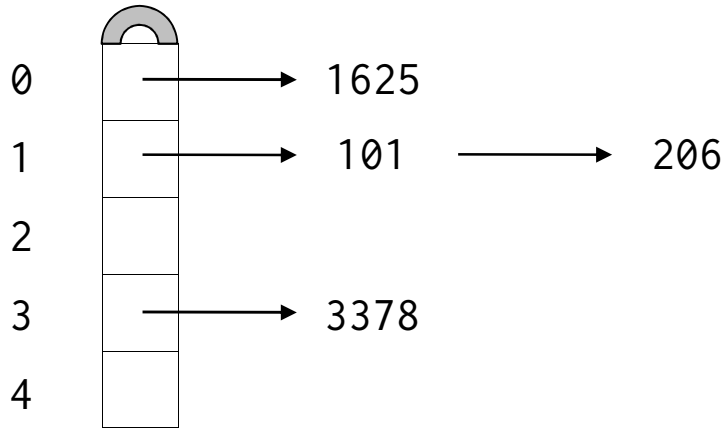
But what do we do when there's a collision?



Option #1: store it in the next empty slot



Option #2: chain values together



Either works well until the table is about 3/4 full

Either works well until the table is about 3/4 full

Then average time to look up/insert rises rapidly

Either works well until the table is about 3/4 full

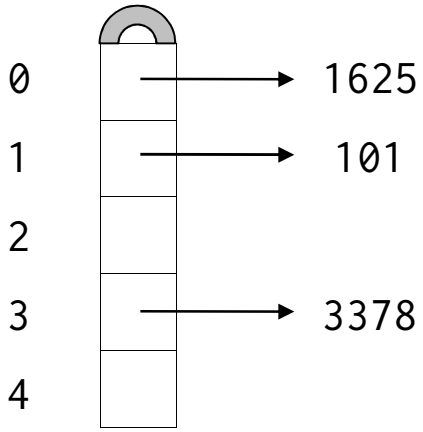
Then average time to look up/insert rises rapidly

So enlarge the table

Either works well until the table is about 3/4 full

Then average time to look up/insert rises rapidly

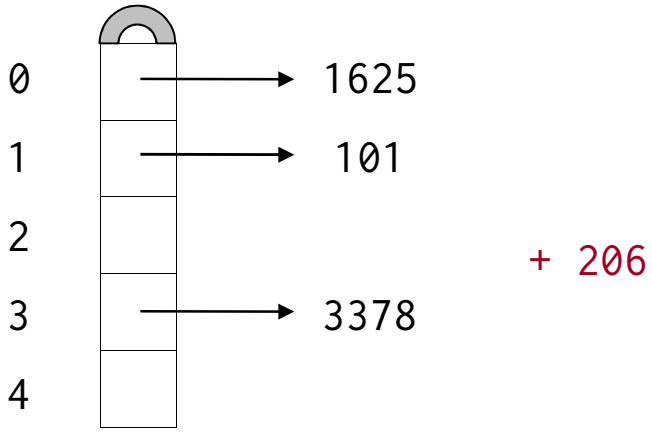
So enlarge the table



Either works well until the table is about 3/4 full

Then average time to look up/insert rises rapidly

So enlarge the table

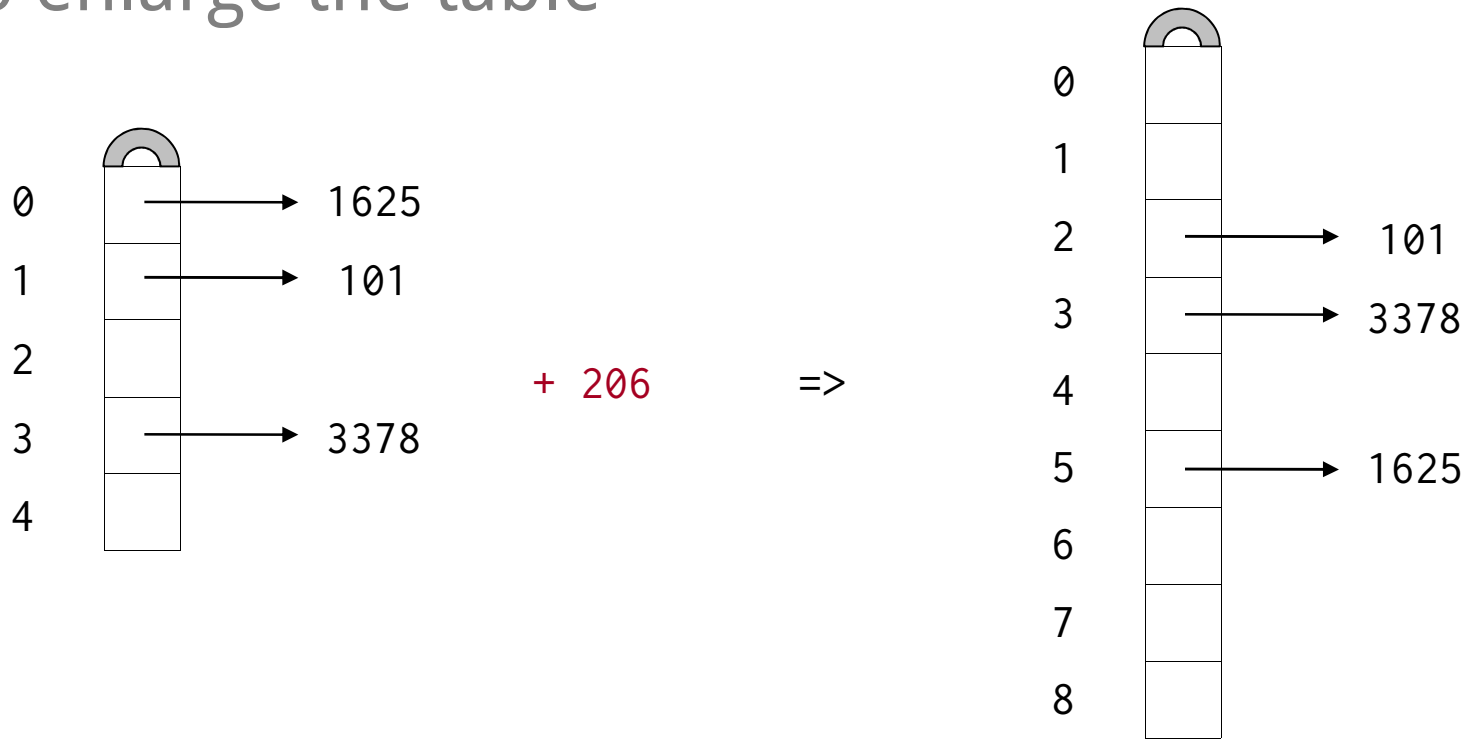



```
+ 206
```

Either works well until the table is about 3/4 full

Then average time to look up/insert rises rapidly

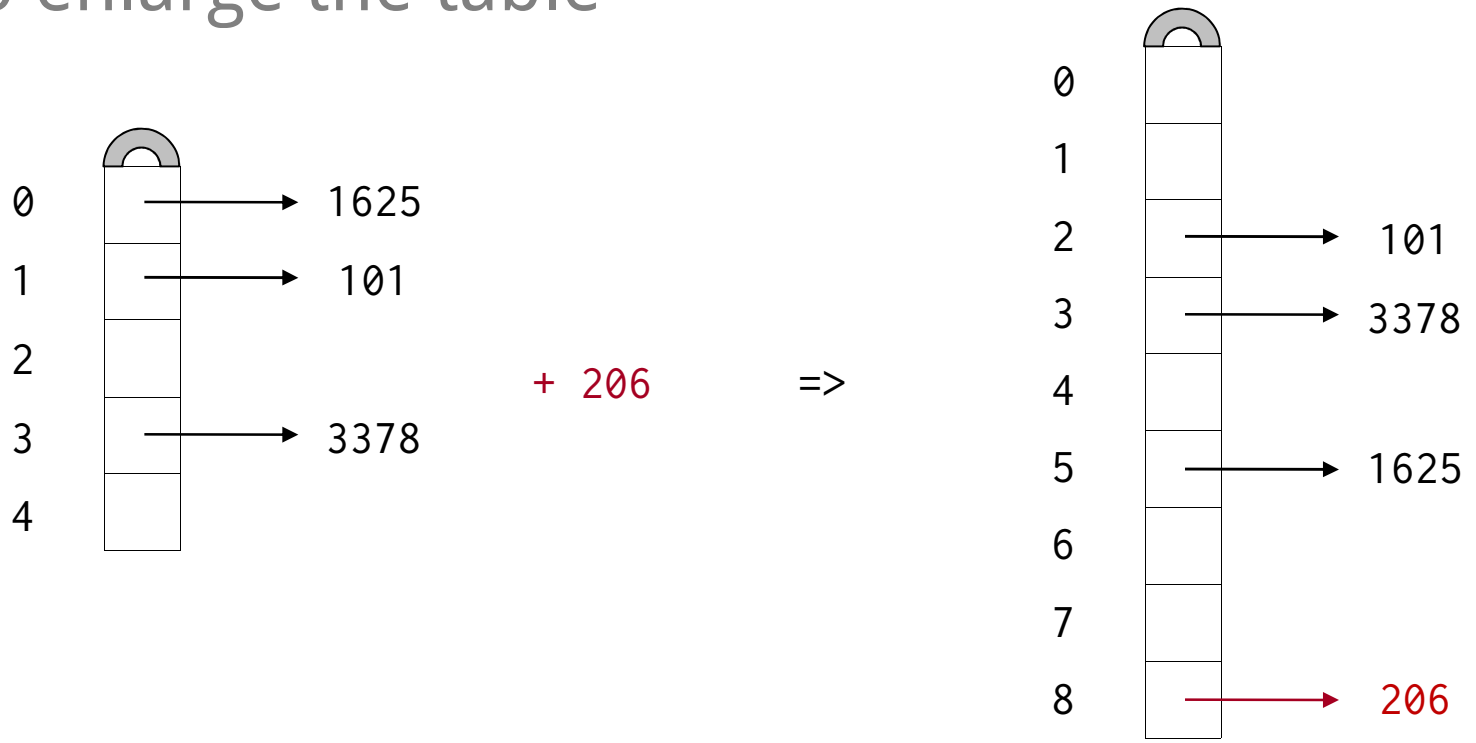
So enlarge the table



Either works well until the table is about 3/4 full

Then average time to look up/insert rises rapidly

So enlarge the table



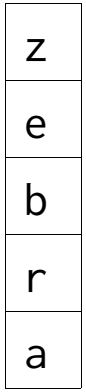
How do we store strings?

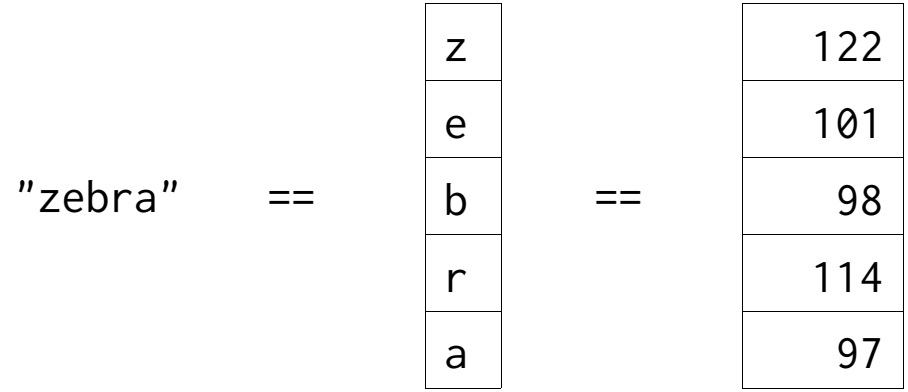
How do we store strings?

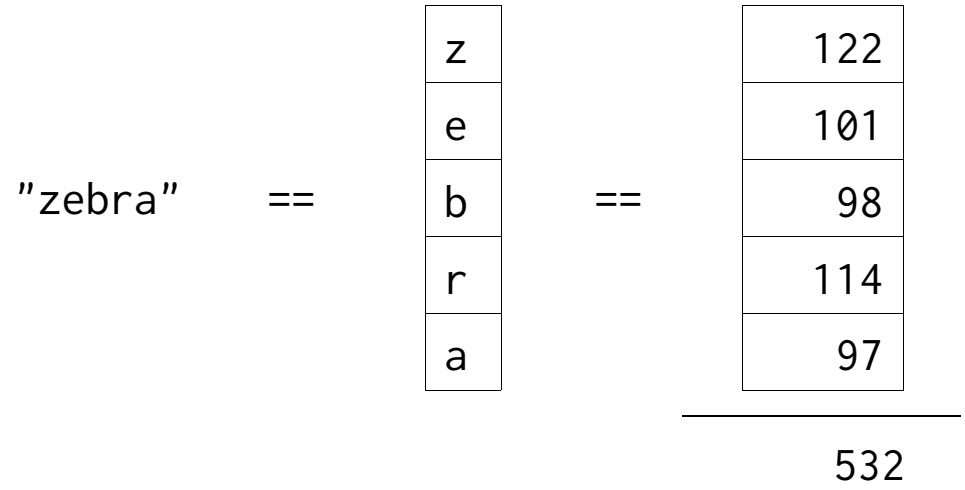
Use a *hash function* to generate an integer index based on the characters in the string

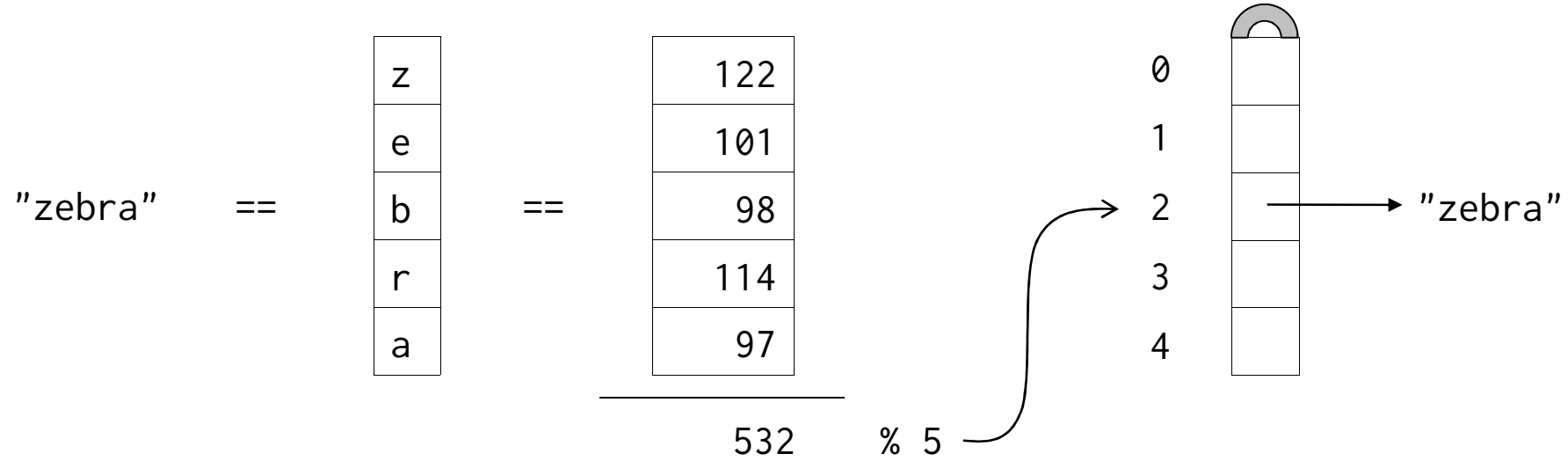
```
"zebra"
```

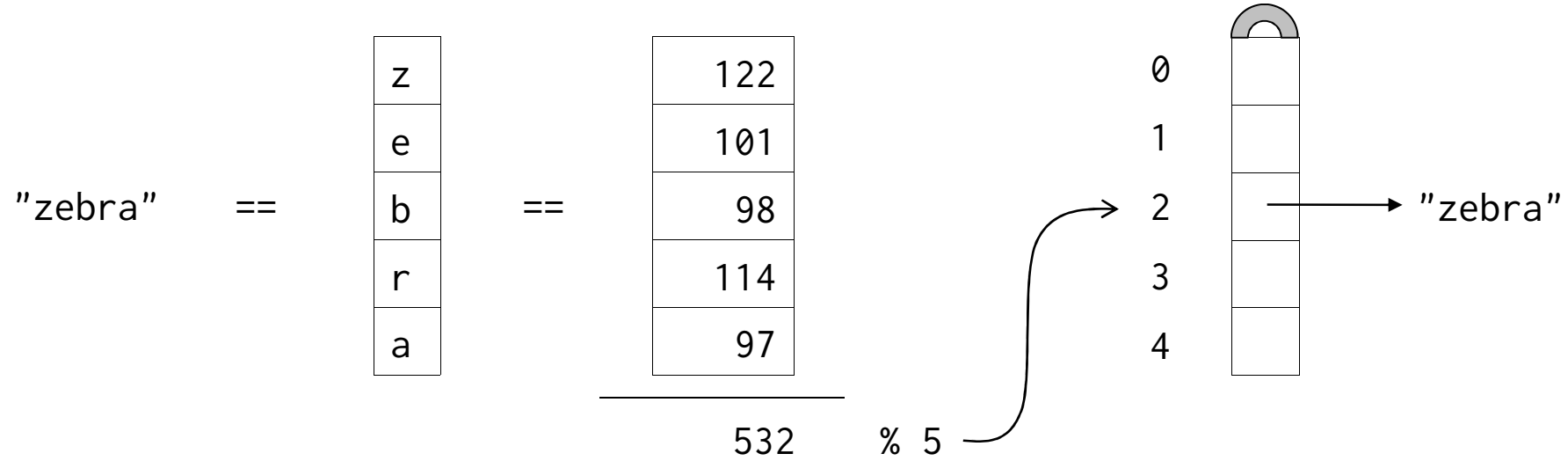
"zebra" ==




"zebra"
==
z
e
b
r
a
==
122
101
98
114
97

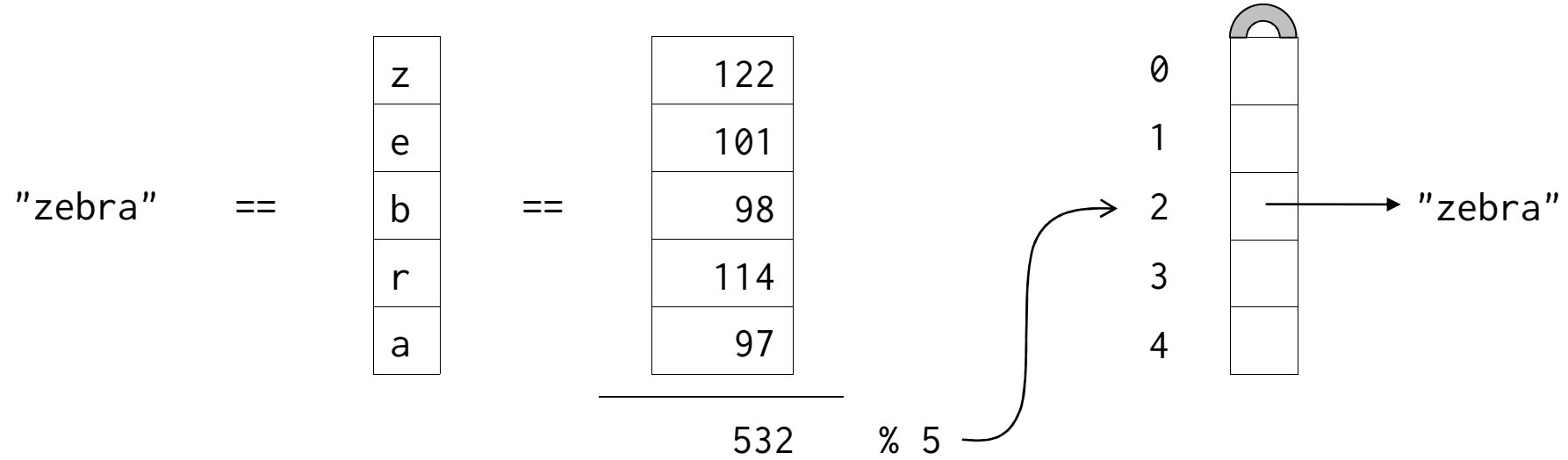

"zebra"
==
z
e
b
r
a
==
122
101
98
114
97
532


"zebra"
==
z
e
b
r
a
==
122
101
98
114
97
532
% 5
0
1
2
3
4
"zebra"



If we can define a hash function for something,

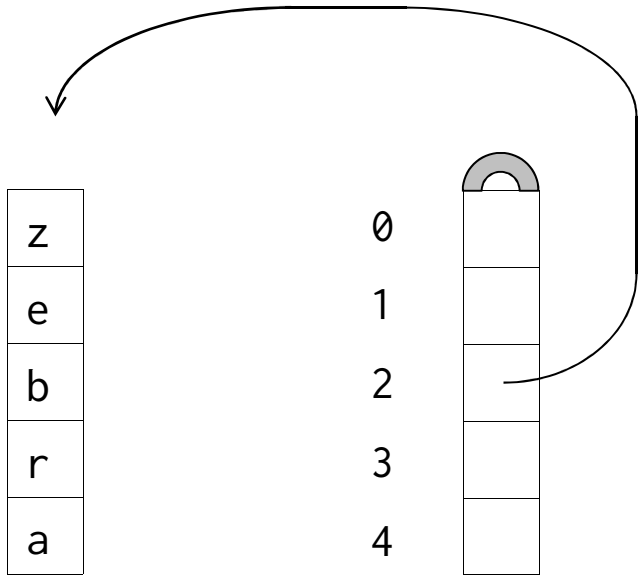
we can store it in a set



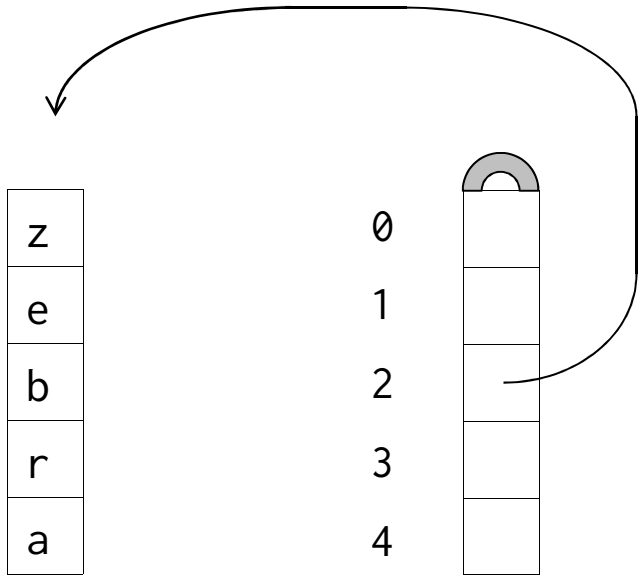
If we can define a hash function for something,

we can store it in a set

So long as nothing changes behind our back



This is what the previous example really looks like in memory

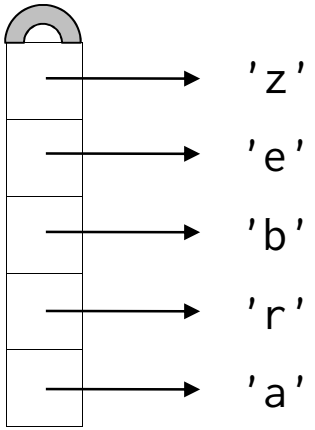


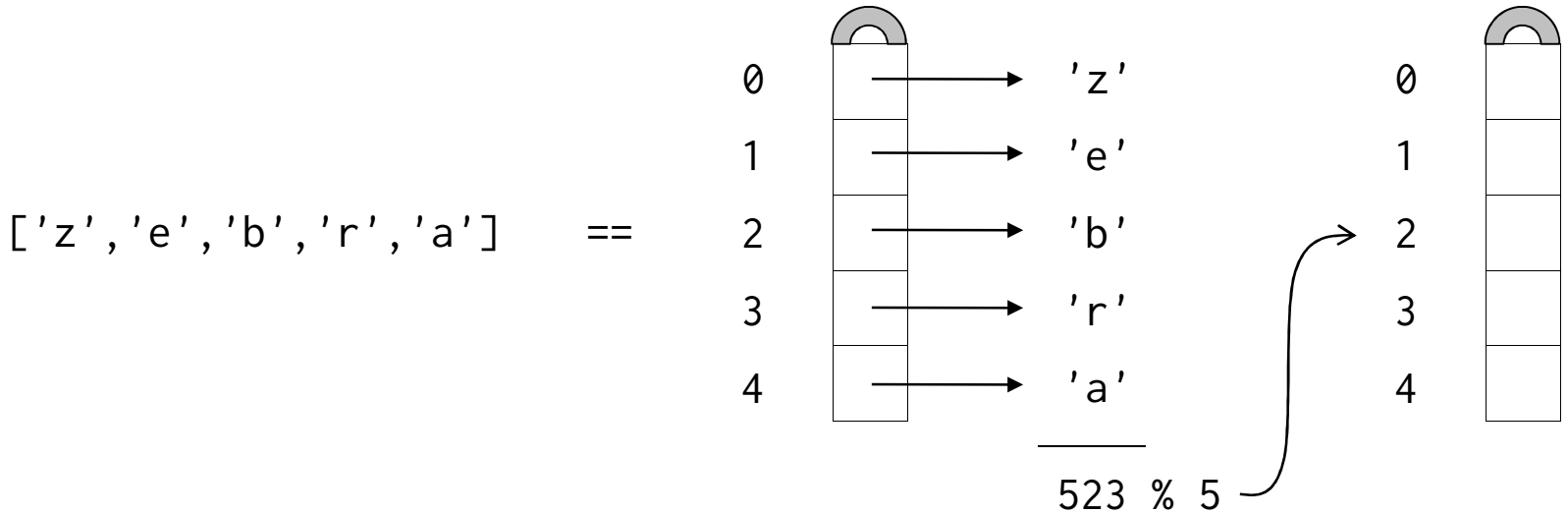
This is what the previous example really looks like in memory

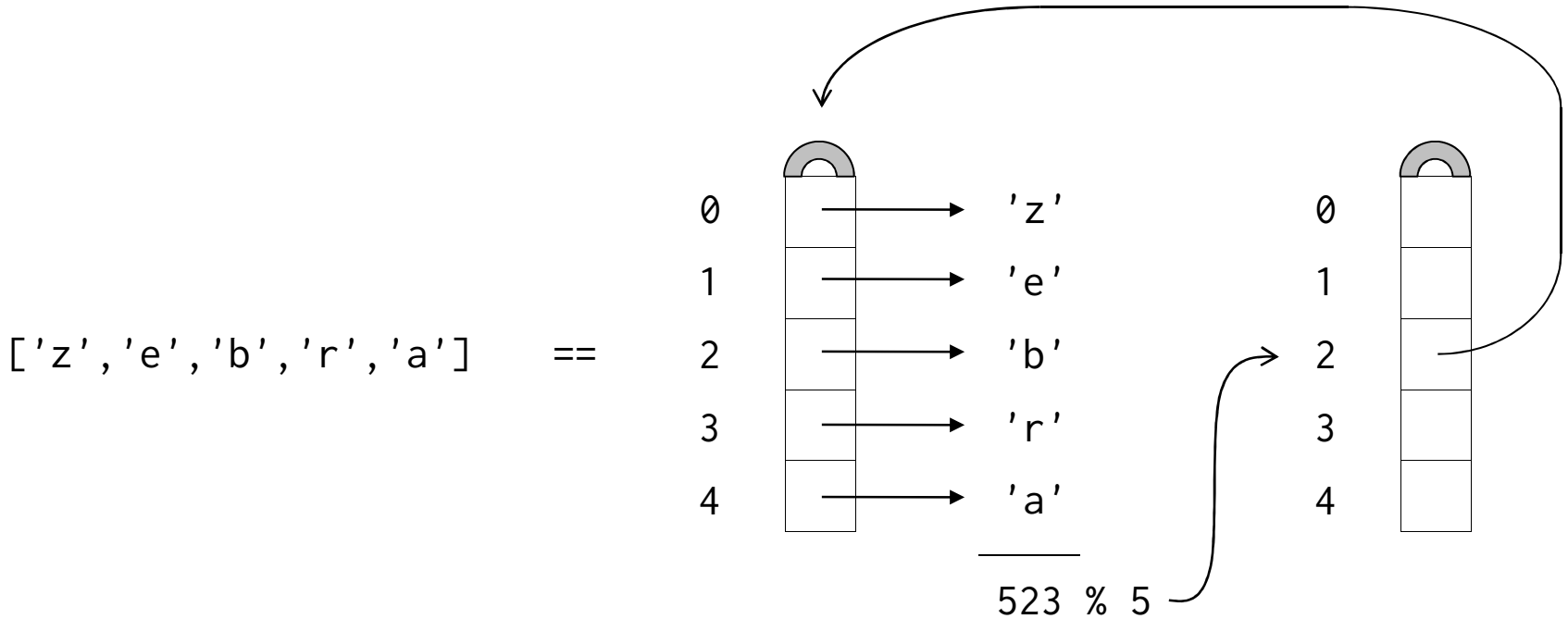
Let's take a look at what happens if we use a list

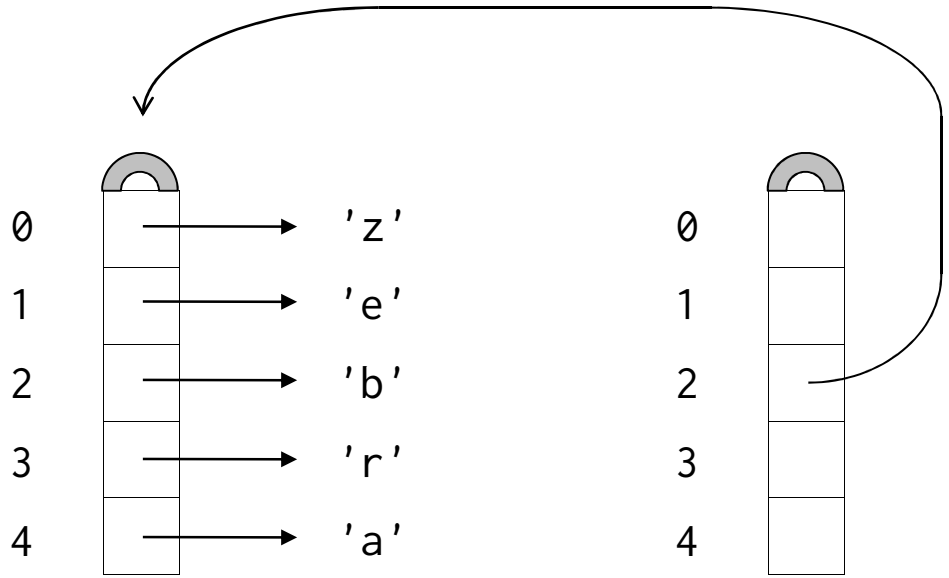
```
['z','e','b','r','a']
```

```
['z','e','b','r','a']    ==
```

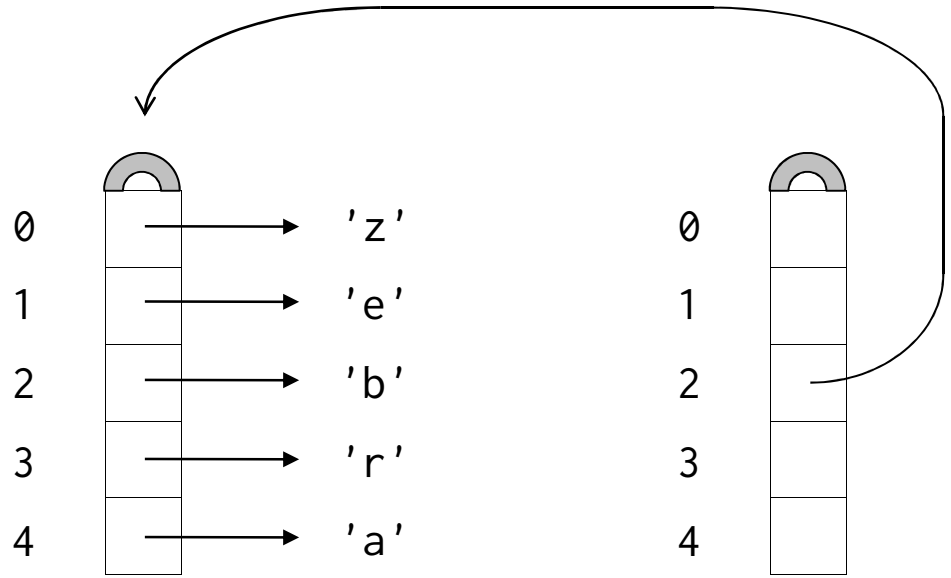



['z','e','b','r','a'] ==
0
1
2
3
4
'z'
'e'
'b'
'r'
'a'
523 % 5
0
1
2
3
4


['z','e','b','r','a'] ==
0
1
2
3
4
'z'
'e'
'b'
'r'
'a'
523 % 5
0
1
2
3
4

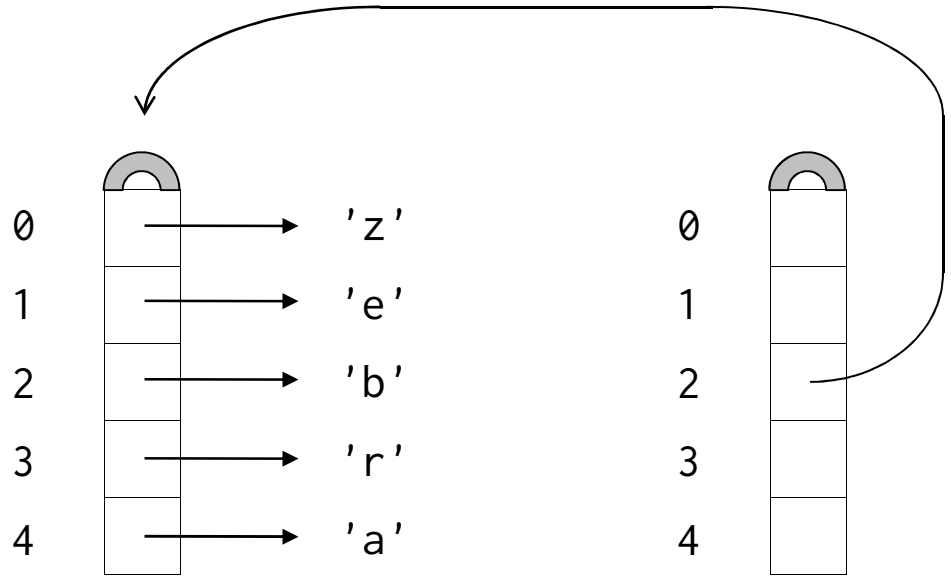


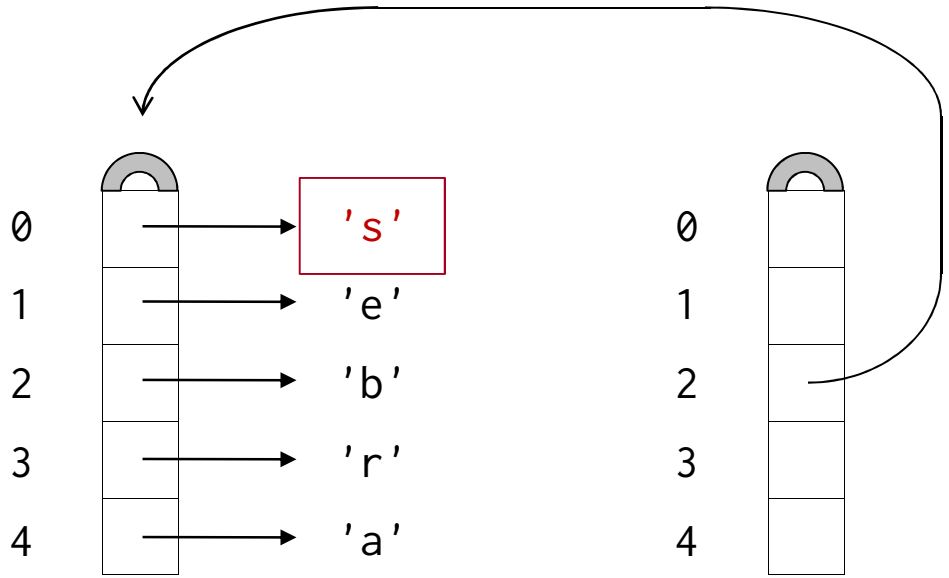
This is what's actually in memory

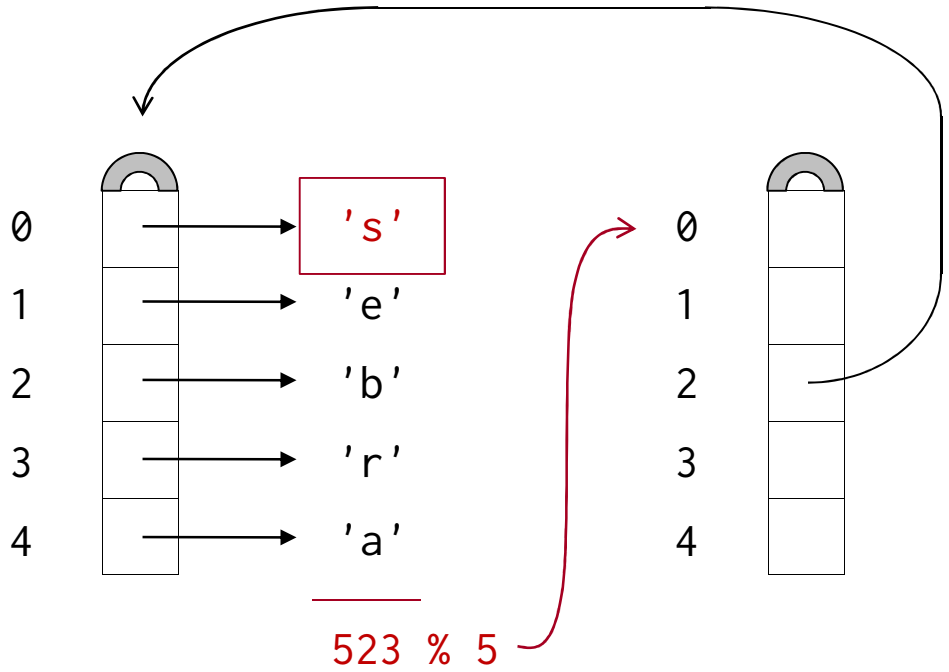


This is what's actually in memory

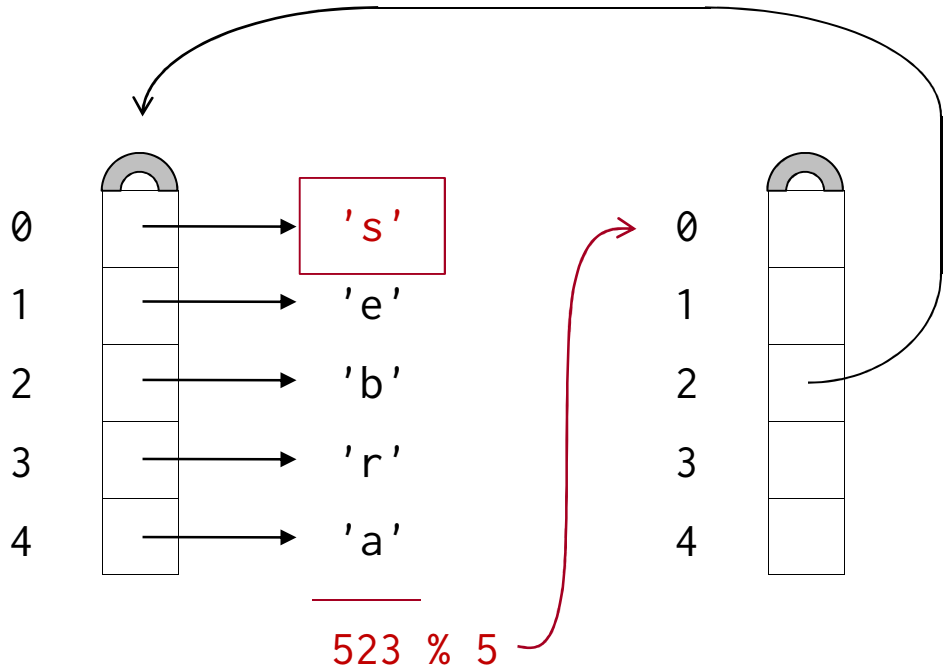
What happens if we change the values in the list?


0
1
2
3
4
'z'
'e'
'b'
'r'
'a'
0
1
2
3
4

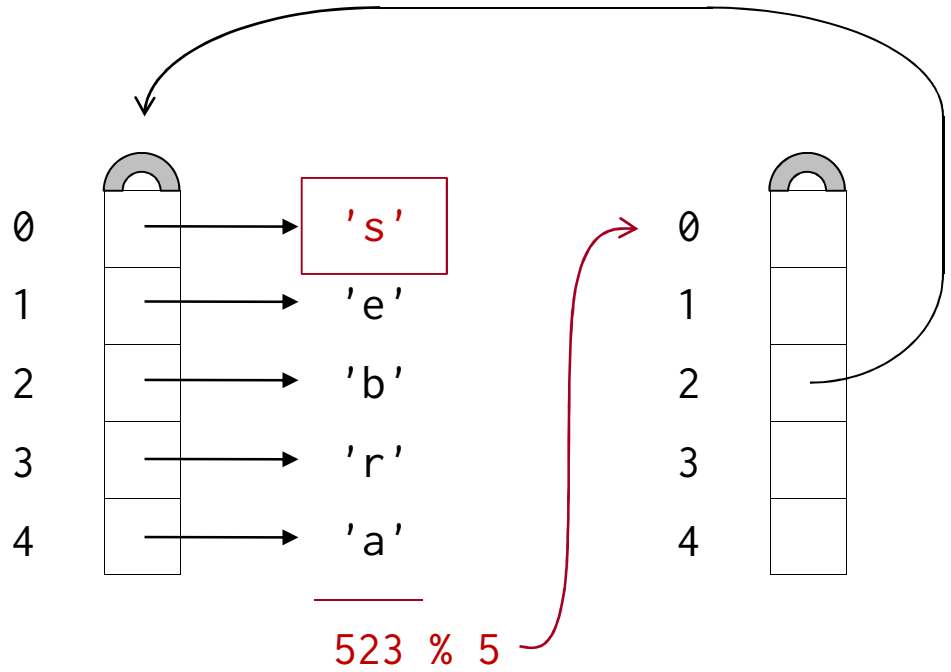

0
1
2
3
4
's'
'e'
'b'
'r'
'a'
0
1
2
3
4


0
1
2
3
4
's'
'e'
'b'
'r'
'a'
523 % 5
0
1
2
3
4

The list is stored in the wrong place!

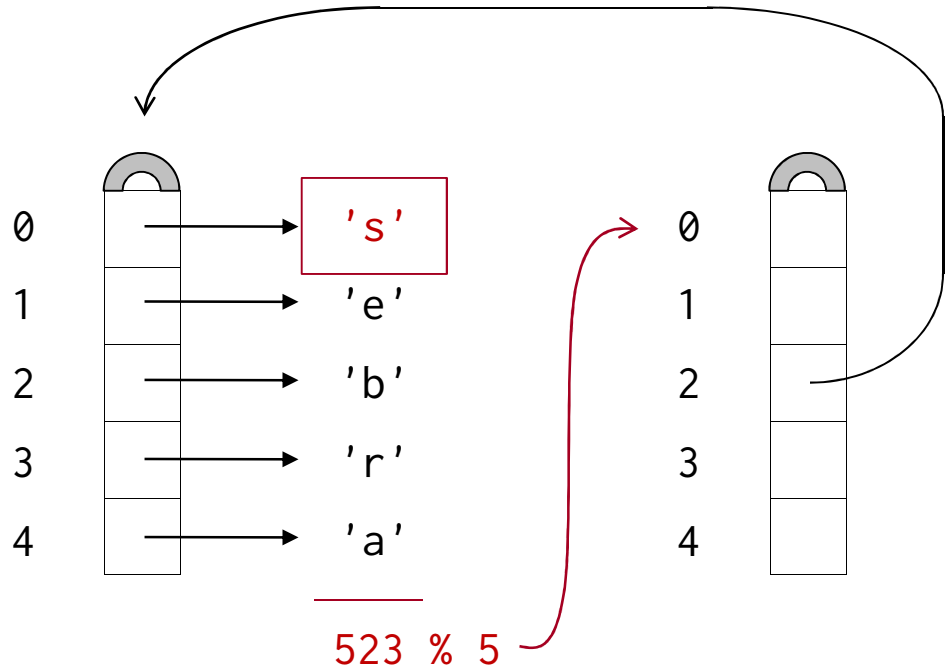


The list is stored in the wrong place!




```
['s','e','b','r','a'] in S
```

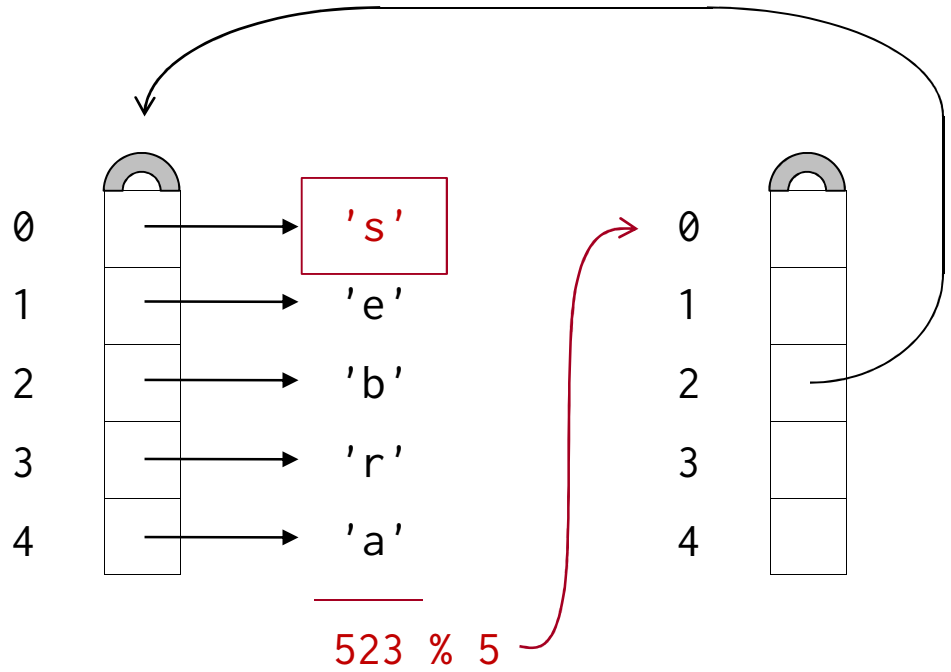
The list is stored in the wrong place!



`['s','e','b','r','a'] in S`

looks at index 0 and says `False`

The list is stored in the wrong place!

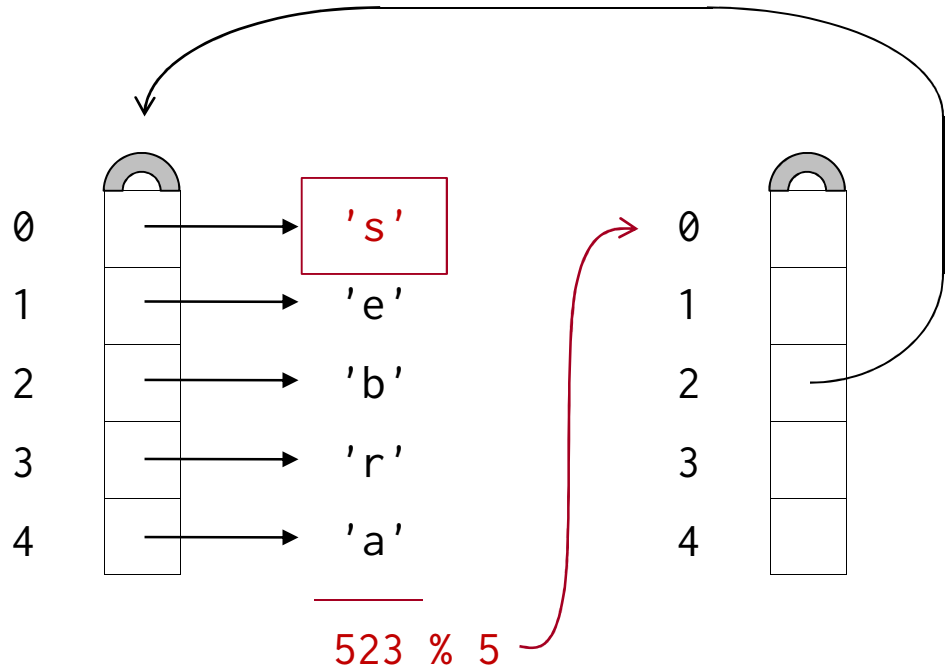


`['s','e','b','r','a'] in S`

looks at index 0 and says `False`

`['z','e','b','r','a'] in S`

The list is stored in the wrong place!



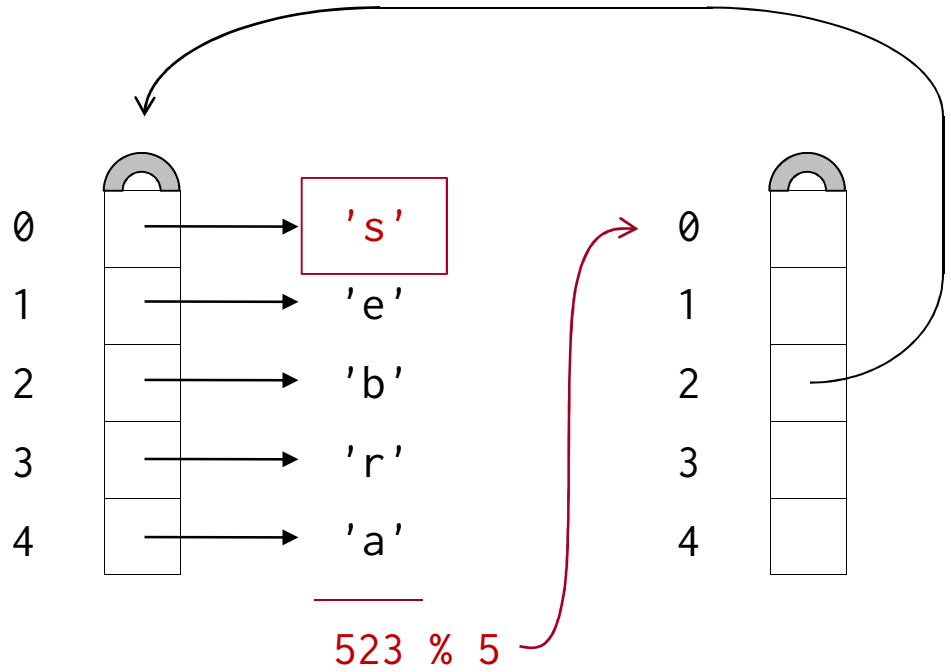
`['s','e','b','r','a'] in S`

looks at index 0 and says `False`

`['z','e','b','r','a'] in S`

looks at index 2 and says `True`

The list is stored in the wrong place!



`['s','e','b','r','a'] in S`

looks at index 0 and says `False`

`['z','e','b','r','a'] in S`

looks at index 2 and says `True` (or blows up)

This problem arises with any *mutable* structure

This problem arises with any *mutable* structure

Option #1: keep track of the sets an object is in,

and update pointers every time the object changes

This problem arises with any *mutable* structure

Option #1: keep track of the sets an object is in,

and update pointers every time the object changes

Very expensive

This problem arises with any *mutable* structure

Option #1: keep track of the sets an object is in,

and update pointers every time the object changes

Very expensive

Option #2: allow it, and blame the programmer

This problem arises with any *mutable* structure

Option #1: keep track of the sets an object is in,

and update pointers every time the object changes

Very expensive

Option #2: allow it, and blame the programmer

Very expensive when it goes wrong

This problem arises with any *mutable* structure

Option #1: keep track of the sets an object is in,

and update pointers every time the object changes

Very expensive

Option #2: allow it, and blame the programmer

Very expensive when it goes wrong

Option #3: only permit *immutable* objects in sets

This problem arises with any *mutable* structure

Option #1: keep track of the sets an object is in,

and update pointers every time the object changes

Very expensive

Option #2: allow it, and blame the programmer

Very expensive when it goes wrong

Option #3: only permit *immutable* objects in sets

(If an object can't change, neither can its hash value)

This problem arises with any *mutable* structure

Option #1: keep track of the sets an object is in,

and update pointers every time the object changes

Very expensive

Option #2: allow it, and blame the programmer

Very expensive when it goes wrong

Option #3: only permit *immutable* objects in sets

(If an object can't change, neither can its hash value)

Slightly restrictive, but never disastrous

So how do we store values that naturally have several parts, like first name and last name?

So how do we store values that naturally have several parts, like first name and last name?

Option #1: concatenate them

So how do we store values that naturally have several parts, like first name and last name?

Option #1: concatenate them

'Charles' and 'Darwin' stored as 'Charles|Darwin'

So how do we store values that naturally have several parts, like first name and last name?

Option #1: concatenate them

'Charles' and 'Darwin' stored as 'Charles|Darwin'

(Can't use space to join 'Paul Antoine' and 'St. Cyr')

So how do we store values that naturally have several parts, like first name and last name?

Option #1: concatenate them

'Charles' and 'Darwin' stored as 'Charles|Darwin'

(Can't use space to join 'Paul Antoine' and 'St. Cyr')

But data *always* changes...

So how do we store values that naturally have several parts, like first name and last name?

Option #1: concatenate them

'Charles' and 'Darwin' stored as 'Charles|Darwin'

(Can't use space to join 'Paul Antoine' and 'St. Cyr')

But data *always* changes...

Code has to be littered with joins and splits

Option #2 (in Python): use a *tuple*

Option #2 (in Python): use a *tuple*

An immutable list

Option #2 (in Python): use a *tuple*

An immutable list

Contents cannot be changed after tuple is created

```
>>> full_name = ('Charles', 'Darwin')
```

```
>>> full_name = ('Charles', 'Darwin')
```

Use '()' instead of '[]'

```
>>> full_name = ('Charles', 'Darwin')
>>> full_name[0]
Charles
```

```
>>> full_name = ('Charles', 'Darwin')
>>> full_name[0]
Charles

>>> full_name[0] = 'Erasmus'
TypeError: 'tuple' object does not support item
assignment
```

```
>>> full_name = ('Charles', 'Darwin)
>>> full_name[0]
Charles

>>> full_name[0] = 'Erasmus'
TypeError: 'tuple' object does not support item
assignment

>>> names = set()
>>> names.add(full_name)
>>> names
set([('Charles', 'Darwin')])
```

This episode has been about the science of computer science

This episode has been about the science of computer science

- Designs for hash tables

This episode has been about the science of computer science

- Designs for hash tables
- Mutability, usability, and performance

This episode has been about the science of computer science

- Designs for hash tables
- Mutability, usability, and performance

It's a lot to digest in one go...

This episode has been about the science of computer science

- Designs for hash tables
- Mutability, usability, and performance

It's a lot to digest in one go...

...but sometimes you need a little theory to make sense of practice

created by

Greg Wilson

July 2010