Grade Database

Read in *N* student records that consist of an ID number plus a grade, and provide a function to find the grade for a given ID number

Grade Database

Read in *N* student records that consist of an ID number plus a grade, and provide a function to find the grade for a given ID number

Solution: Use a binary search tree keyed on the ID number:

- Read and sort is O(*N* log *N*)
- Lookup is O(log *N*)

Grade Database

Read in *N* student records that consist of an ID number *between 0 and N-1* plus a grade, and provide a function to find the grade for a given ID number

Grade Database

Read in *N* student records that consist of an ID number *between 0 and N-1* plus a grade, and provide a function to find the grade for a given ID number

Solution: Use an array of size *N*

- Read and sort is O(*N*)
- Lookup is O(1)

Grade Database

Read in *N* student records that consist of an ID number *between 0 and 2N* plus a grade, and provide a function to find the grade for a given ID number

Grade Database

Read in *N* student records that consist of an ID number *between 0 and 2N* plus a grade, and provide a function to find the grade for a given ID number

Solution: Still best to use an array of size 2*N*

- Read and sort is O(*N*)
- Lookup is O(1), but an array slot might be empty

Grade Database

Read in *N* student records that consist of an ID number *between 0 and 100N* plus a grade, and provide a function to find the grade for a given ID number

Grade Database

Read in *N* student records that consist of an ID number *between 0 and 100N* plus a grade, and provide a function to find the grade for a given ID number

Solution: An array of size 100*N* is probably too wasteful...

Grade Database

Read in *N* student records that consist of an ID number *evenly distributed in the range 0 to 100N* plus a grade, and provide a function to find the grade for a given ID number

Grade Database

Read in *N* student records that consist of an ID number *evenly distributed in the range 0 to 100N* plus a grade, and provide a function to find the grade for a given ID number

Solution: Use an array of size *N*, and lookup by dropping the last two digits

Each array element is a (short) linked list, in case of collisions

- Read and sort is O(*N*)
- Lookup is O(1)

Hash Tables

General strategy:

- A ***hash function*** converts a value to a number:
  - posn: multiply the x and y numbers
  - name: treat it as a base-26 number
  - snake: convert fields to numbers and add
- Use the number modulo array size as an index
  - handle collisions somehow

Chained Hashing

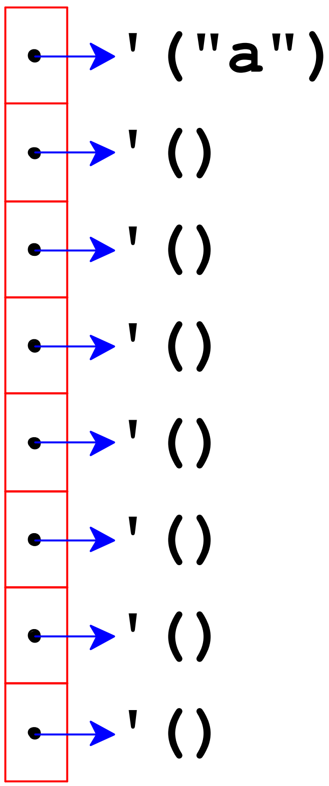
Chained hashing handles collisions by keeping a list of values at each slot

If the distributiion of hash codes is fairly uniform, the lists are very short

Chained Hashing

`(hash "a")` = 0

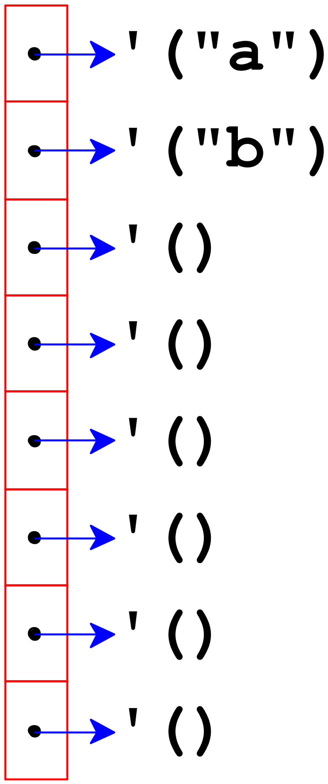
`(modulo 0 8)` = 0



Chained Hashing

`(hash "b")` = 1

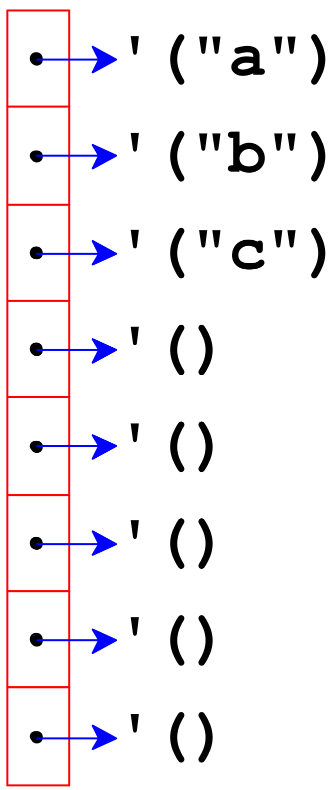
`(modulo 1 8)` = 1



Chained Hashing

`(hash "c")` = 2

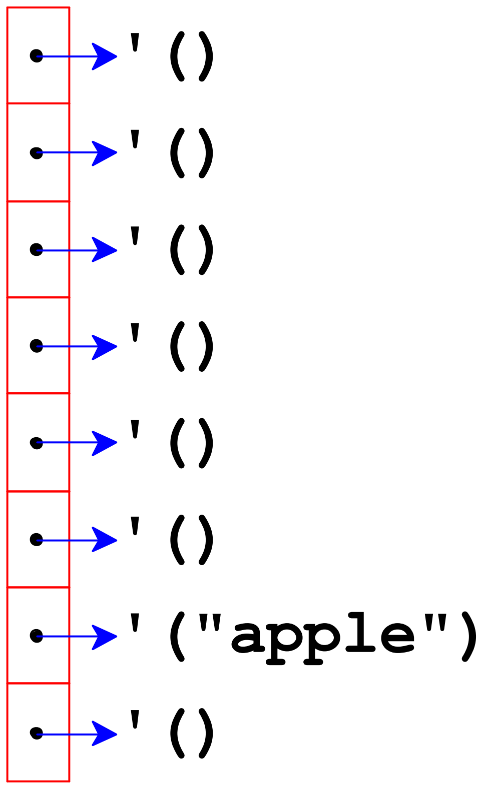
`(modulo 2 8)` = 2



Chained Hashing

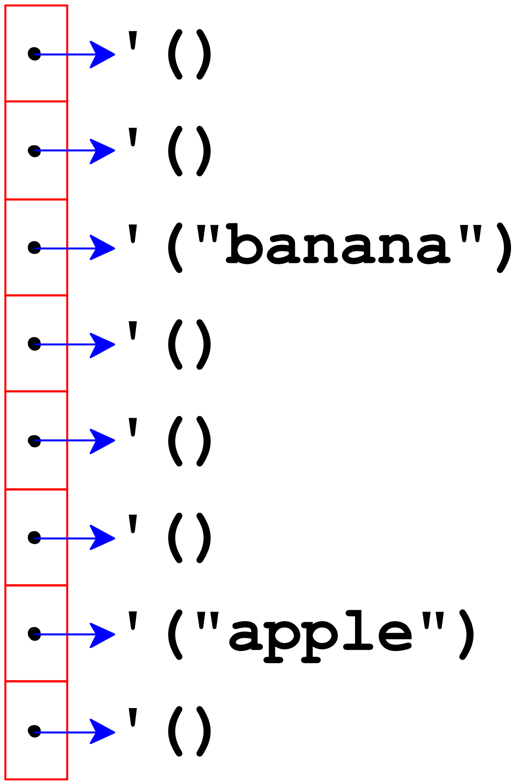
`(hash "apple")` = 274070

`(modulo 274070 8)` = 6



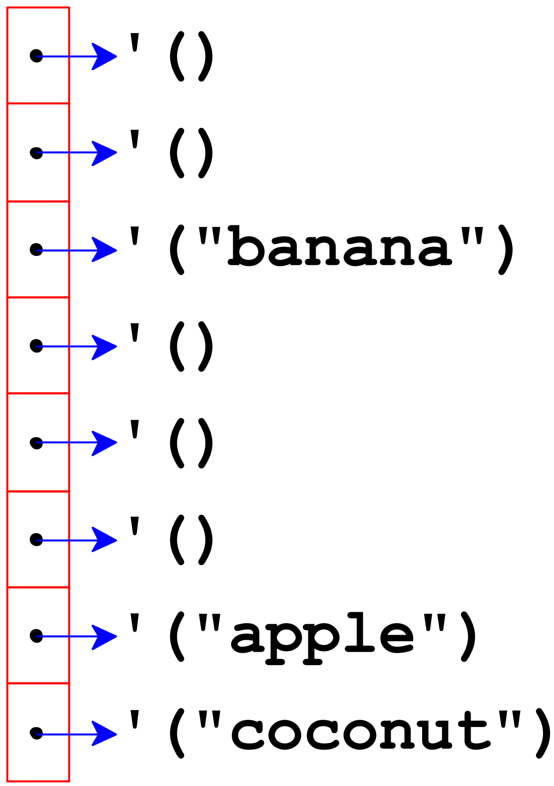
Chained Hashing

```
(hash "banana") = 12110202

(modulo 12110202 8) = 2
```



Chained Hashing

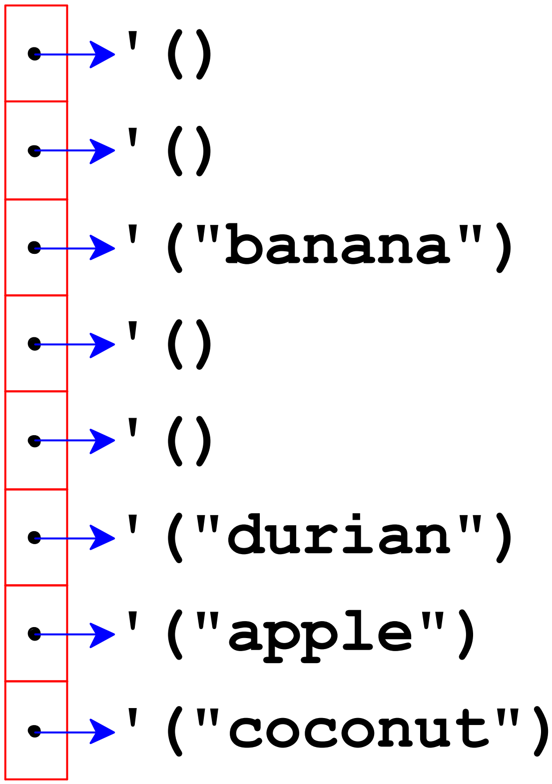
```
(hash "coconut") = 785340159

(modulo 785340159 8) = 7
```



Chained Hashing

`(hash "durian")` = 45087861

`(modulo 45087861 8)` = 5



Chained Hashing

`(hash "eggplant")` = 34059071949

`(modulo 34059071949 8)` = 5

```
• → '()
• → '()
• → '("banana")
• → '()
• → '()
• → '("durian" "eggplant")
• → '("apple")
• → '("coconut")
```

Chained Hashing

`(hash "fig")` = 3594

`(modulo 3594 8)` = 2

```
'()
'()
'("banana" "fig")
'()
'()
'("durian" "eggplant")
'("apple")
'("coconut")
```

Chained Hashing

`(hash "grape")` = 3041042

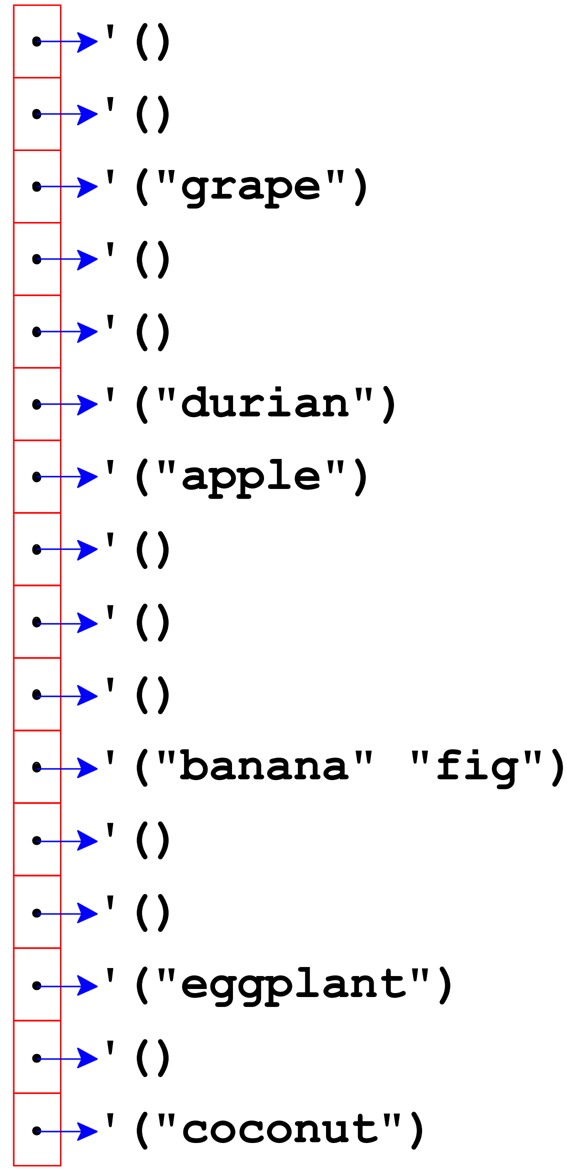
`(modulo 3041042 8)` = 2

```
'()
'()
'("banana" "fig" "grape")
'()
'()
'("durian" "eggplant")
'("apple")
'("coconut")
```

Chained Hashing

Re-hash when the table's count is more than *k* times the array's size

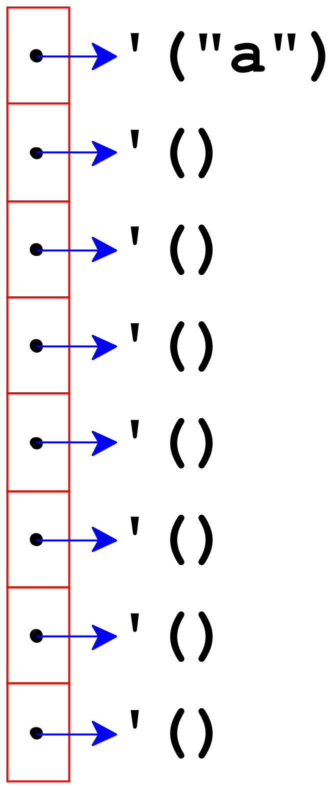
The value *k* is the ***load factor*** or ***fill factor***



Chained Hashing

`(hash "a")` = 0

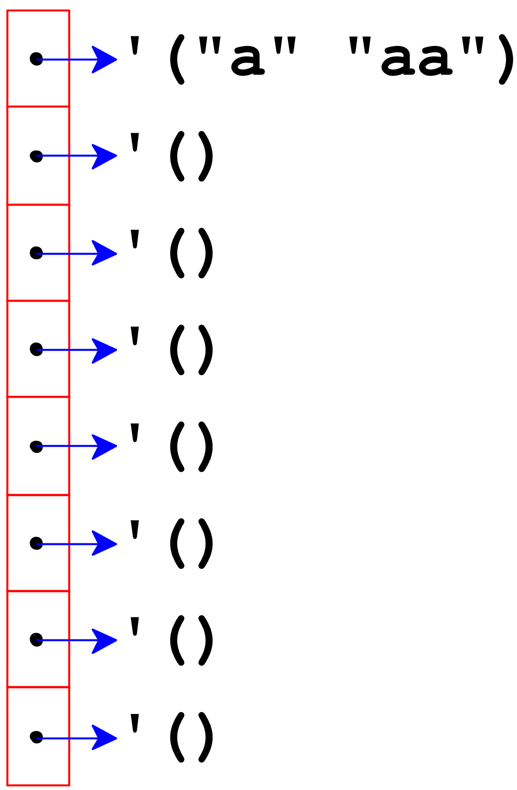
`(modulo 0 8)` = 0



Chained Hashing

`(hash "aa")` = 0

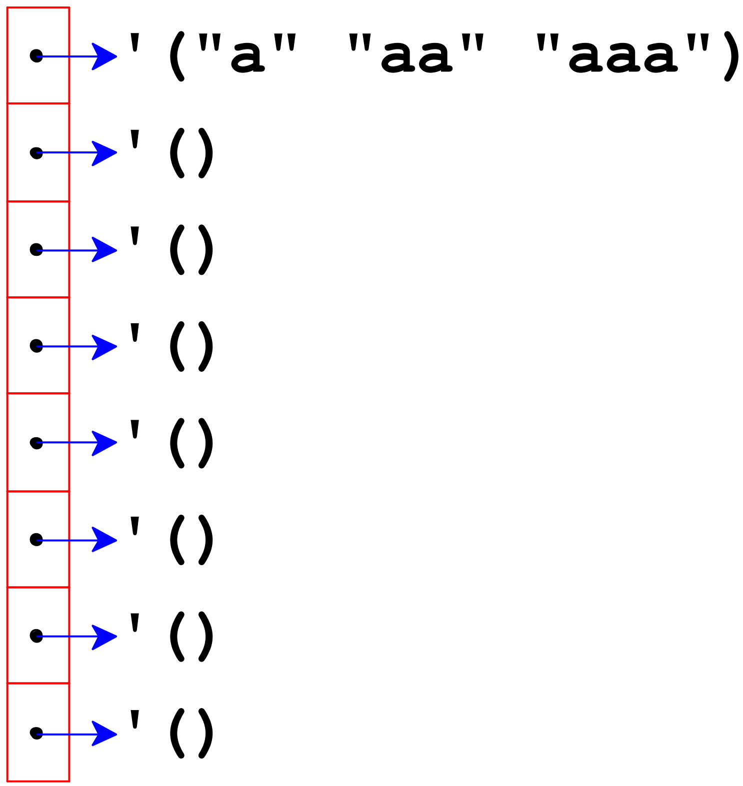
`(modulo 0 8)` = 0



Chained Hashing

`(hash "aaa")` = 0

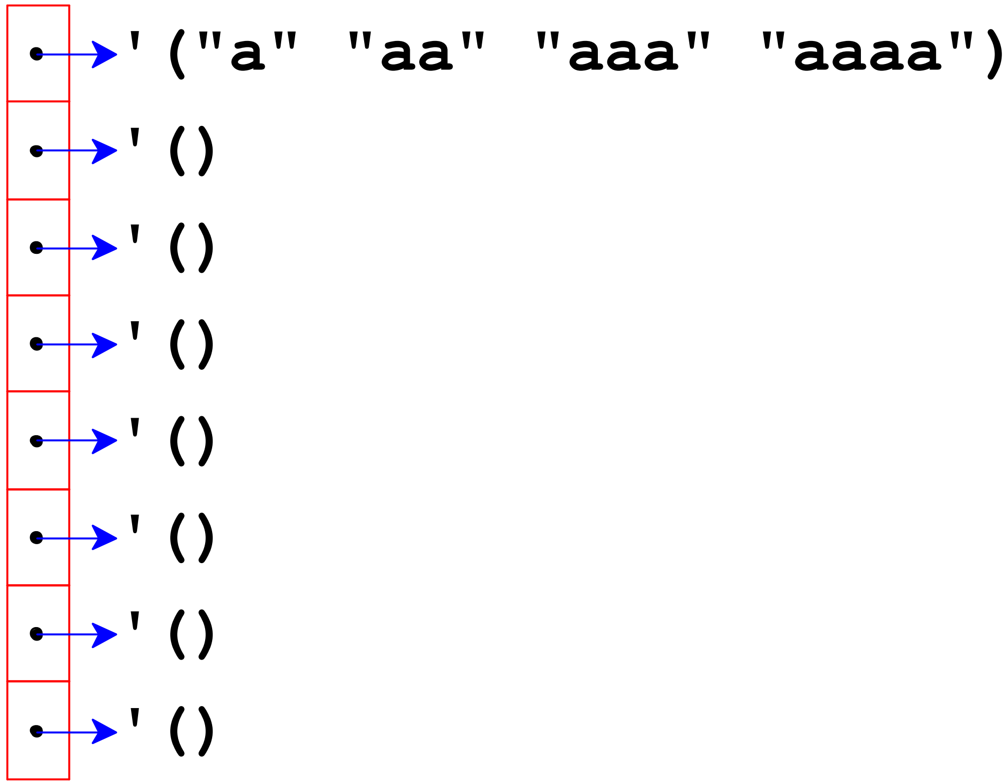
`(modulo 0 8)` = 0



Chained Hashing

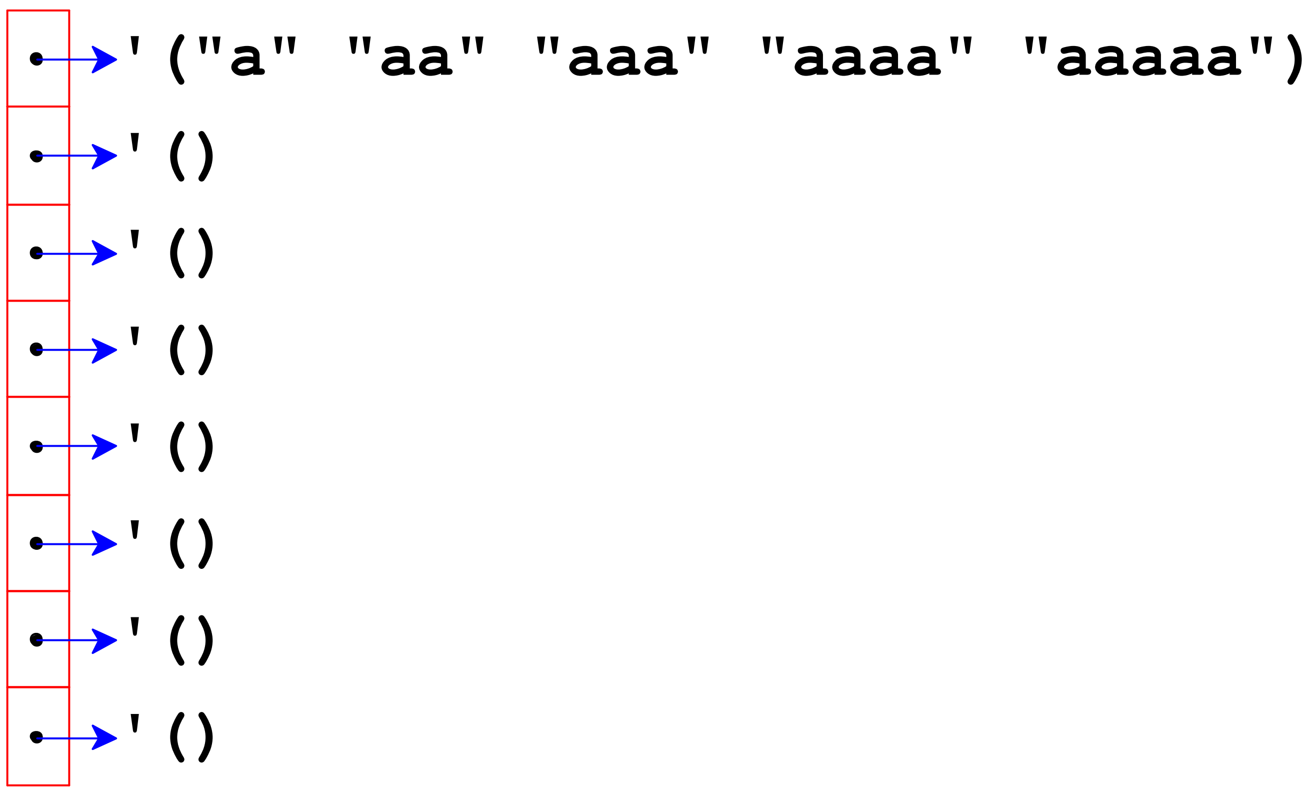
`(hash "aaaa")` = 0

`(modulo 0 8)` = 0

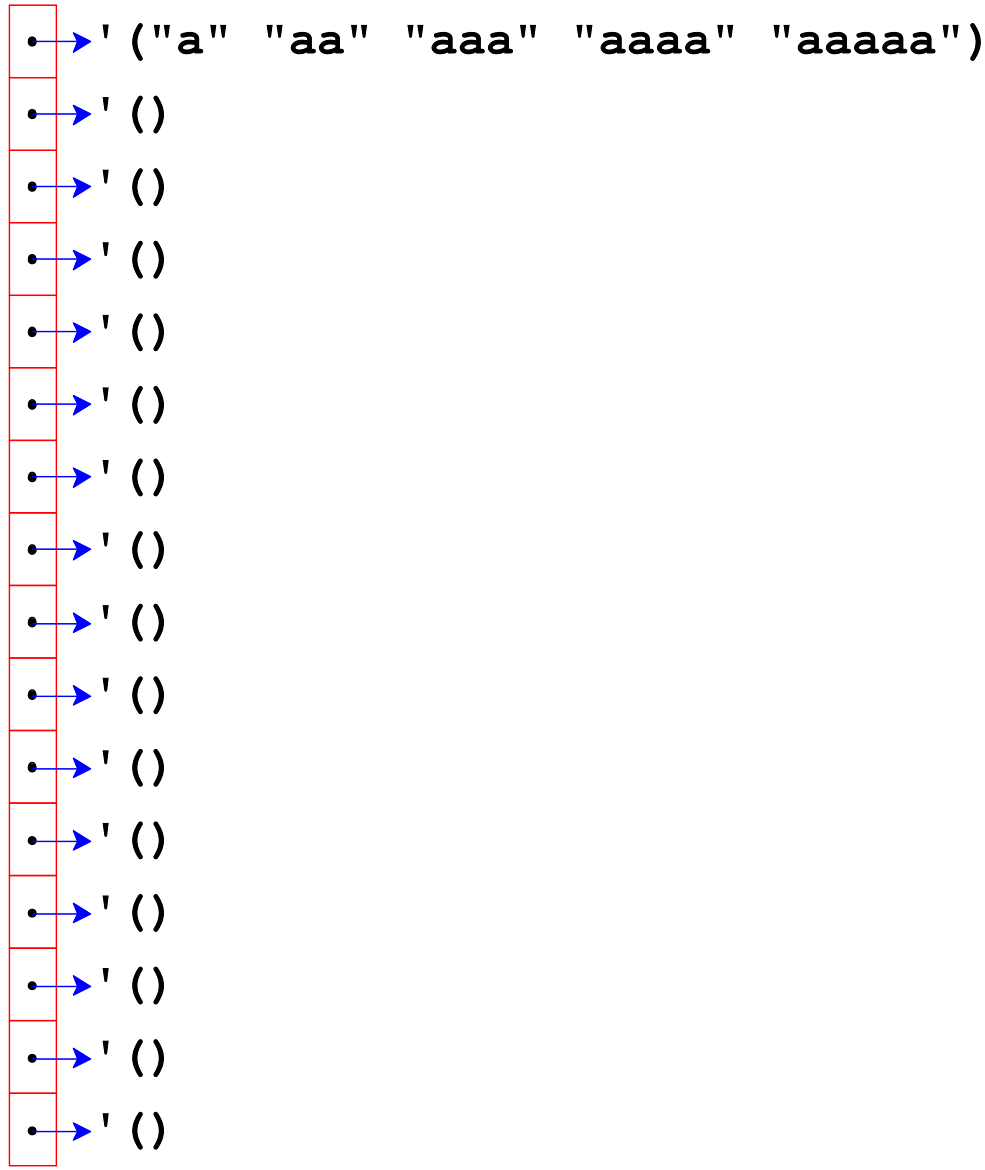


Chained Hashing

```
(hash "aaaaa") = 0

(modulo 0 8) = 0
```

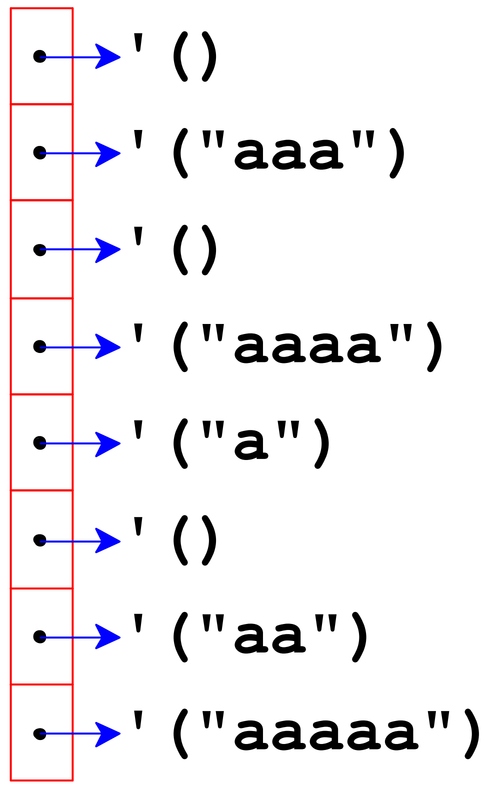


Chained Hashing

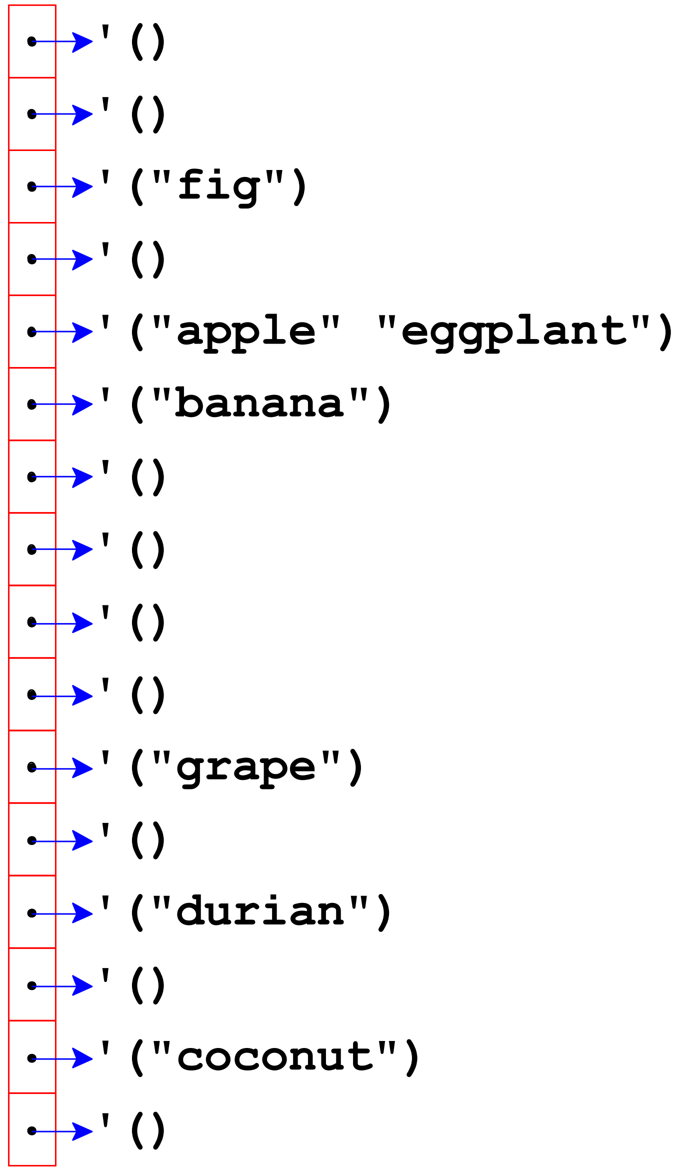


Picking the right hash function is sometimes difficult

Using a Better Hash Function



Using a Better Hash Function



Chained Hashing

See **`chain-hash.c`**

Linear Probing

Linear probing handles collisions by trying the next slot

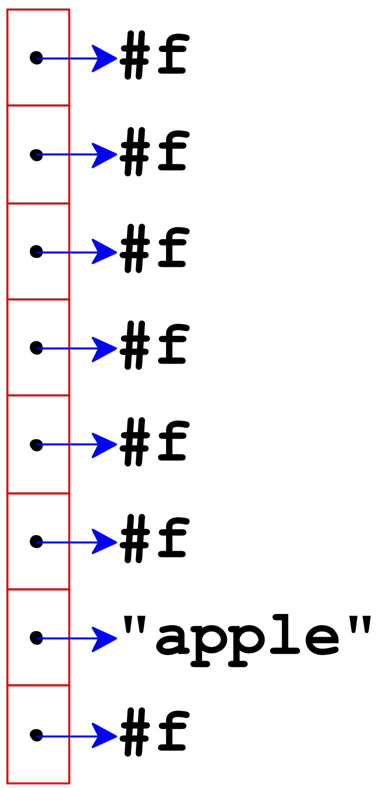
Linear probing avoids linked lists at the potential expense of worse cluster effects

“Next” can mean *k* slots later for any fixed *k*

Linear Probing

`(hash "apple")` = 274070

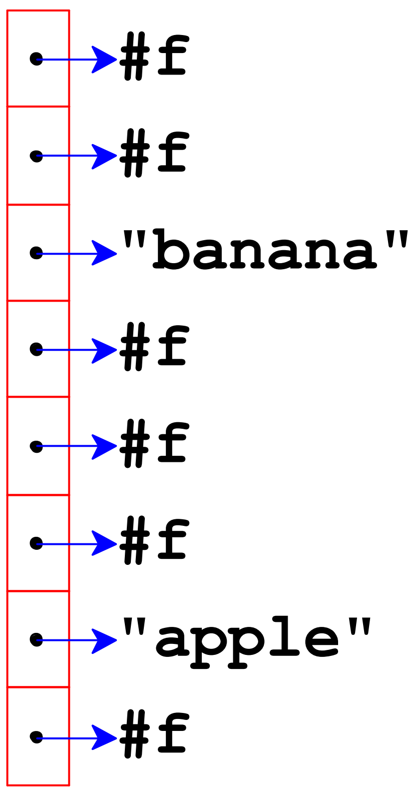
`(modulo 274070 8)` = 6



Linear Probing

`(hash "banana")` = 12110202

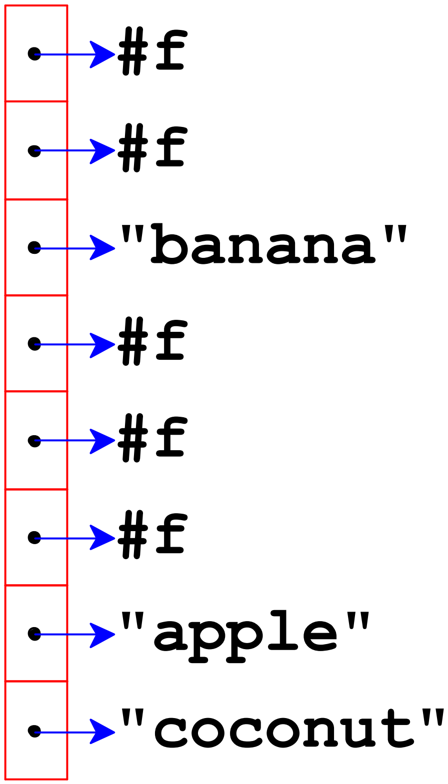
`(modulo 12110202 8)` = 2



Linear Probing

`(hash "coconut")` = 785340159

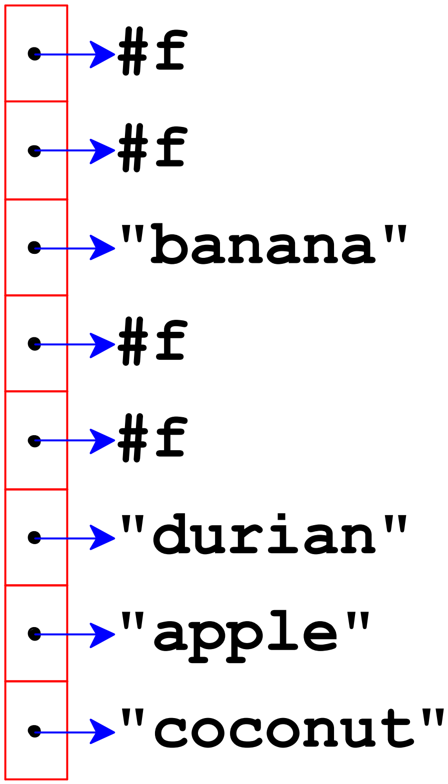
`(modulo 785340159 8)` = 7



Linear Probing

`(hash "durian")` = 45087861

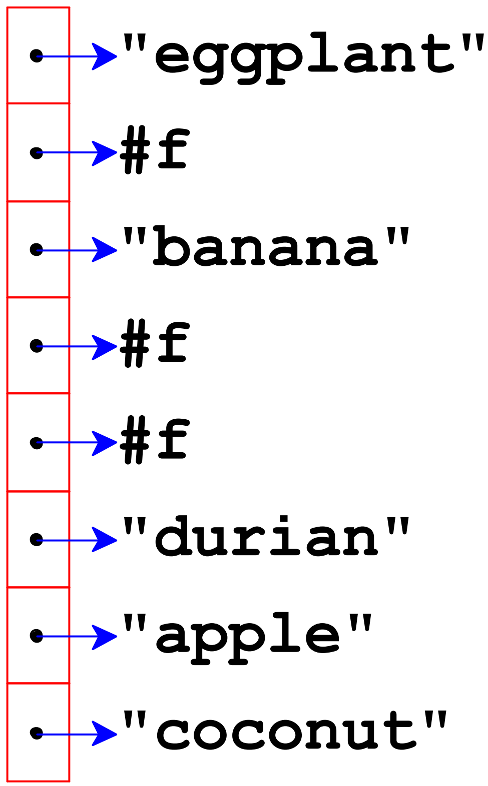
`(modulo 45087861 8)` = 5



Linear Probing

`(hash "eggplant")` = 34059071949

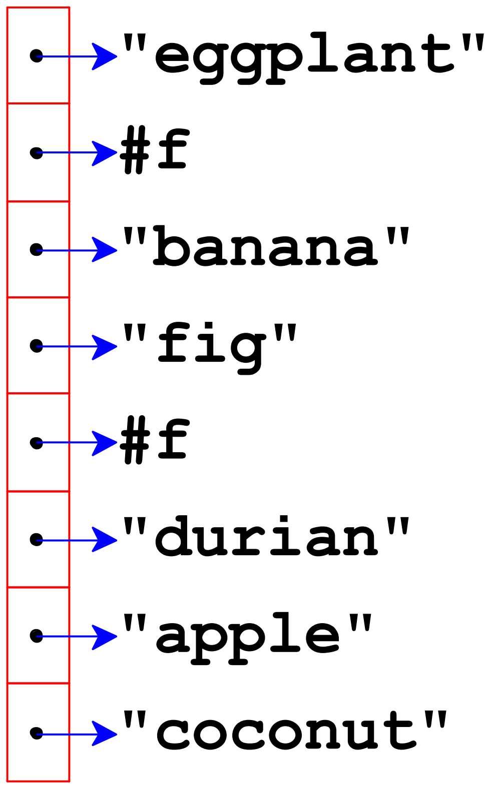
`(modulo 34059071949 8)` = 5



Linear Probing

`(hash "fig")` = 3594

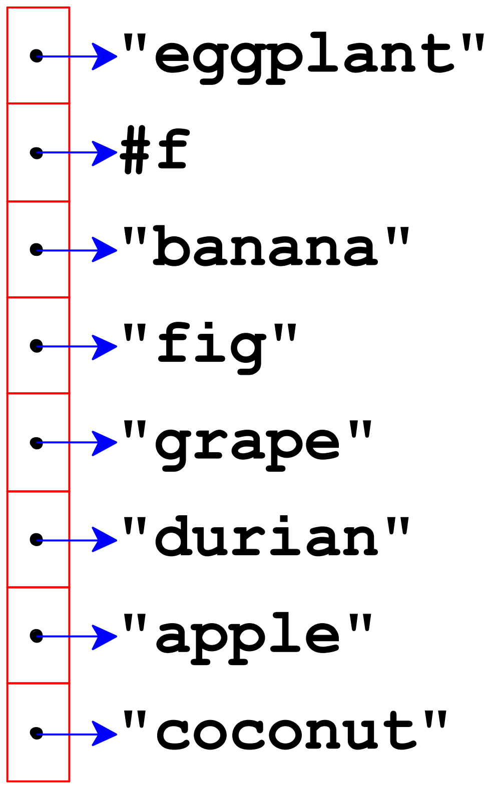
`(modulo 3594 8)` = 2



Linear Probing

`(hash "grape")` = 3041042

`(modulo 3041042 8)` = 2



Chained Hashing

See **`double-hash.c`** with

`#define DOUBLE_HASH 0`

Double Hashing

Double hashing generalizes linear probing by making "next" depend on the key

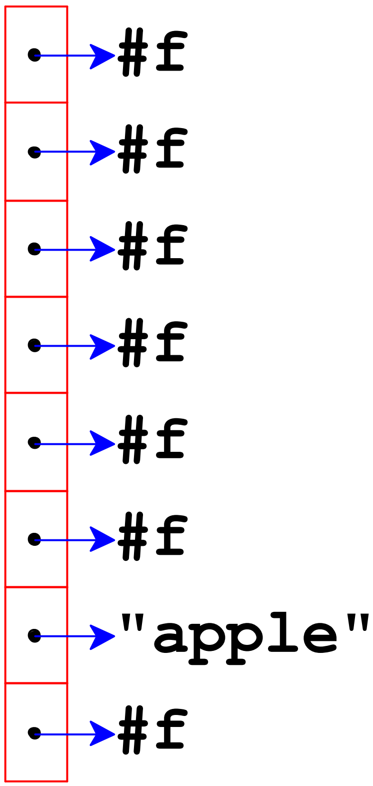
By using two different hash functions for the primary and secondary hash codes, double hashing limits the damage of a bad hashing function

Double Hashing

`(hash "apple")` = 274070

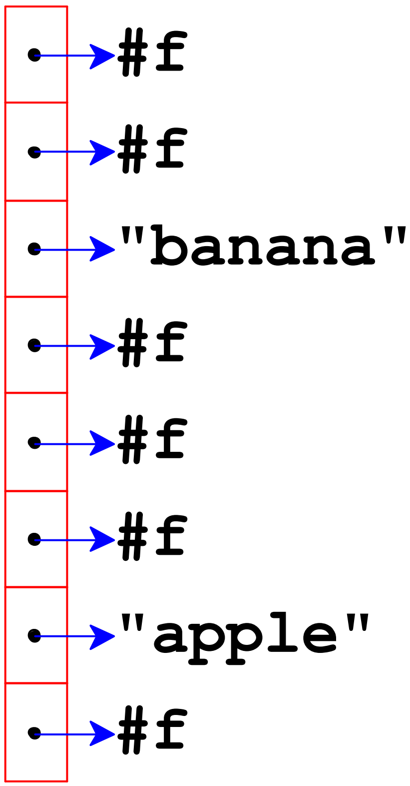
`(modulo 274070 8)` = 6

`(modulo (hash2 "apple") 8)` = 5



Double Hashing

```
(hash "banana") = 12110202

(modulo 12110202 8) = 2

(modulo (hash2 "banana") 8) = 3
```

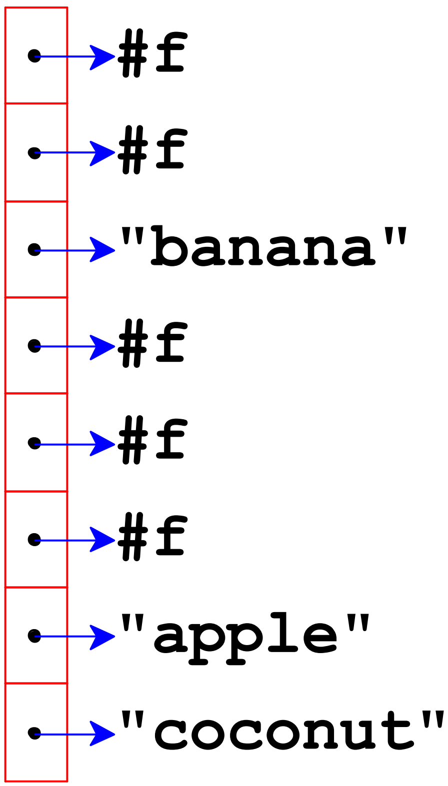


Double Hashing

`(hash "coconut")` = 785340159

`(modulo 785340159 8)` = 7

`(modulo (hash2 "coconut") 8)` = 4

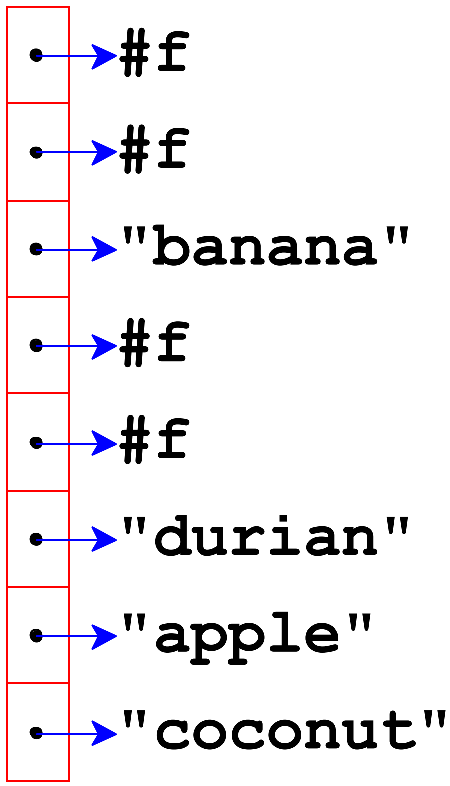


Double Hashing

`(hash "durian")` = 45087861

`(modulo 45087861 8)` = 5

`(modulo (hash2 "durian") 8)` = 5

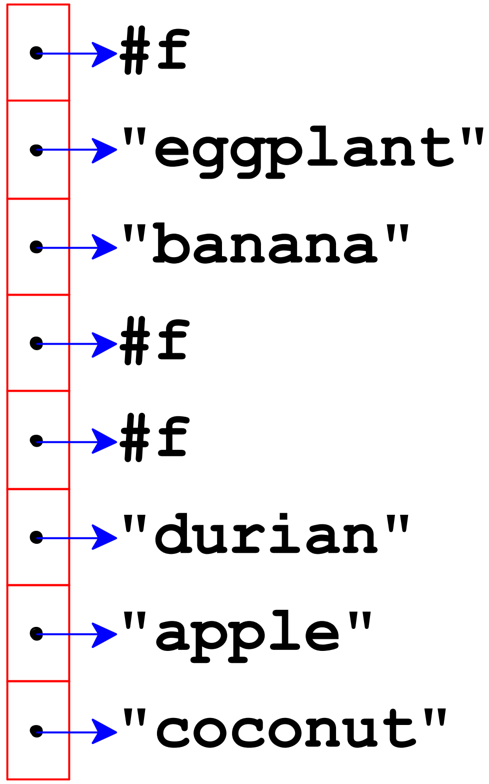


Double Hashing

`(hash "eggplant")` = 34059071949

`(modulo 34059071949 8)` = 5

`(modulo (hash2 "eggplant") 8)` = 2



Double Hashing

`(hash "fig")` = 3594

`(modulo 3594 8)` = 2

`(modulo (hash2 "fig") 8)` = 3

```
"fig"
"eggplant"
"banana"
#f
#f
"durian"
"apple"
"coconut"
```

Double Hashing

```
(hash "grape") = 3041042

(modulo 3041042 8) = 2

(modulo (hash2 "grape") 8) = 2
```

```
• → "fig"
• → "eggplant"
• → "banana"
• → #f
• → "grape"
• → "durian"
• → "apple"
• → "coconut"
```

Chained Hashing

See **`double-hash.c`** with

```
#define DOUBLE_HASH 1
```

Using Hash Table Libraries

Languages like Java and Racket provide built-in hash table support

- Java:

```
ht = new HashMap<Key, Val>()
ht.put(key, val)
ht.get(key)
```

- Racket:

```
(define ht (make-hash))
(hash-set! ht key val)
(hash-ref ht key [default])
```

Each built-in type has a built-in hashing function

Using Hash Table Libraries

For new classes in Java:

- To make equality work, implement

```
boolean equals(Object o)
```

- To make hashing work, implement

```
int hashCode()
```

Using Hash Table Libraries

For new structure types in Racket:

- Make the structure `#:transparent`

— or —

- Add a `prop:equal+hash` property to implement equality and hashing

Using Hash Table Libraries

For a general-purpose hash-table implementation in C, provide a hash function using a function pointer

See **`hash.h`** and **`hash.c`**