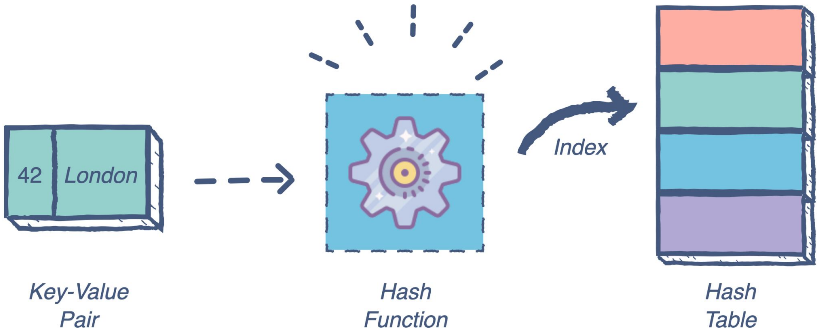


Introduction to Hash Tables

Hammad Ahmad
University of Michigan, Ann Arbor

Today’s Learning Goals

- Explain why hash tables are useful
- Contrast hash tables against an alternative strategy that would support the same operations
- Explain an approach to dealing with collisions
- Describe at least one hash function and some challenges of designing a “good” hash function
- Describe an application where a hash table would be appropriate

Setting The Stage…

- What is an array?
- What is a linked list?
- What is time complexity (big-O)?
- What are ASCII values?

Motivating Problem: The Big Picture

“How many likes and dislikes does a video have?”

“What quantity of a certain item is left in stock?”

Motivating Problem: A Closer Look

Word Count: Given a text file, return the number of times each word appears.

Example: “this is some text”

Result: “this”→1, “is”→1, “some”→1, “text”→1

How should we represent this data?

Motivating Problem: More Complex Inputs

Word Count: Given a text file, return the number of times each word appears.

Example: <The Wikipedia Page for Harvey Mudd College>

Result: ???

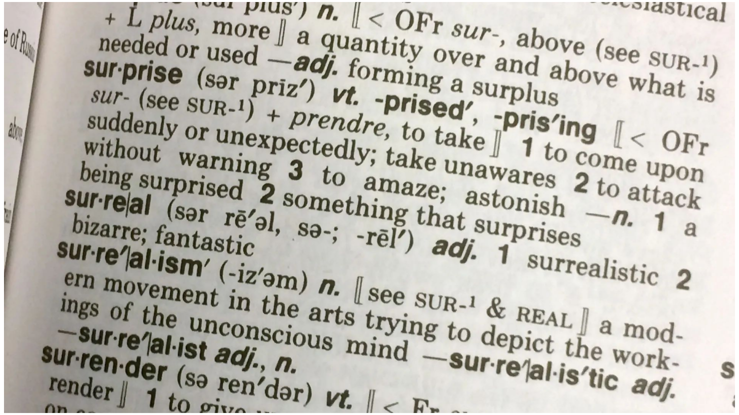
How should we represent this data?

Dictionary Abstract Data Type

Dictionary: A container of key-value pairs that supports three basic operations

Key: “thing” used to look up data

Value: some information about the “thing”

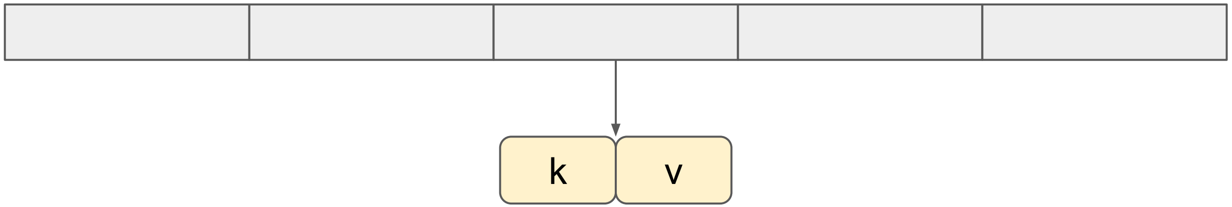


Desired Operations:

- **Insert** a new key-value pair
- **Search** (or retrieve) the value associated with a key
- **Remove** a key-value pair

A (Basic) Implementation of Dictionaries

Unsorted array storing the entries (i.e., key-value pairs)



A (Basic) Implementation of Dictionaries

Unsorted array storing the entries (i.e., key-value pairs)

Word Count Example: “this is some text”

<this, 1>	<is, 1>	<some, 1>

A (Basic) Implementation of Dictionaries

Unsorted array storing the entries (i.e., key-value pairs)

Word Count Example: “this is some text”

<this, 1>	<is, 1>	<some, 1>	<text, 1>

Adding a new entry

A (Basic) Implementation of Dictionaries

Unsorted array storing the entries (i.e., key-value pairs)

Word Count Example: “this is some text”

<this, 1>	<is, 1>	<some, 1>	<text, 1>

Searching for a key

A (Basic) Implementation of Dictionaries

Unsorted array storing the entries (i.e., key-value pairs)

Word Count Example: “this is some text”

<this, 1>	<some, 1>	<text, 1>

Removing an entry

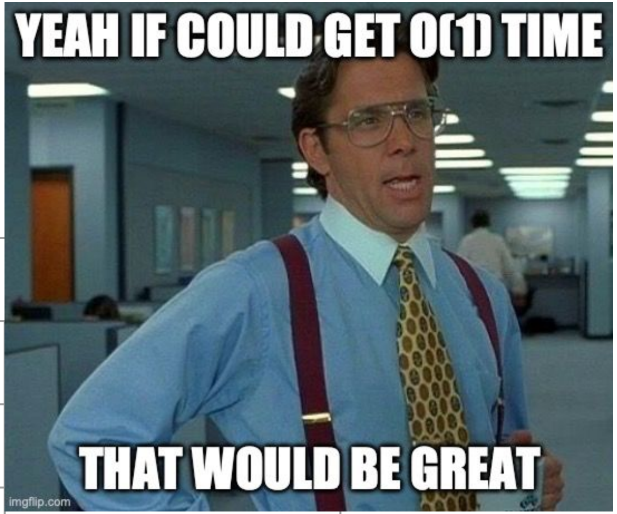
Thoughts?
Comments?
Questions?

Comparing Dictionary Implementations

Implementation	Time Complexity †		
	Insert	**Search**	**Delete**
Unsorted Array	O(1)	O(n)	O(n)
Linked List	O(1)	O(n)	O(n)
Sorted Array	O(n)	O(log n)	O(n)
Balanced Binary Search Tree	O(log n)	O(log n)	O(log n)

† Number of comparisons / steps needed to perform an operation

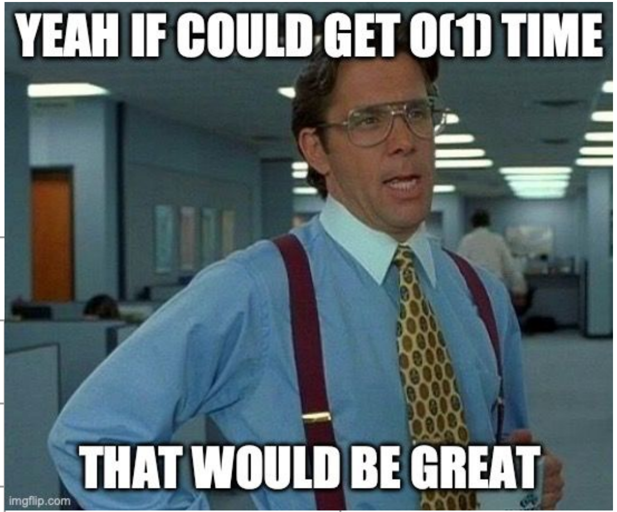
Comparing Dictionary Implementations



Implementation	Time Complexity		
	Insert	**Search**	
Unsorted Array	O(1)	O(*n*)	
Linked List	O(1)	O(*n*)	O(*n*)
Sorted Array	O(*n*)	O(log *n*)	O(*n*)
Balanced Binary Search Tree	O(log *n*)	O(log *n*)	O(log *n*)

Can we do better?

Comparing Dictionary Implementations



Implementation	Time Complexity		
	Insert	**Search**	[illegible]
Unsorted Array	O(1)	O(*n*)	[illegible]
Linked List	O(1)	O(*n*)	O(*n*)
Sorted Array	O(*n*)	O(log *n*)	O(*n*)
Balanced Binary Search Tree	O(log *n*)	O(log *n*)	O(log *n*)

Can we do better?

Yes, by storing the entries in a special key-indexed table, where the index is a function of the key!

Introducing: Hash Tables

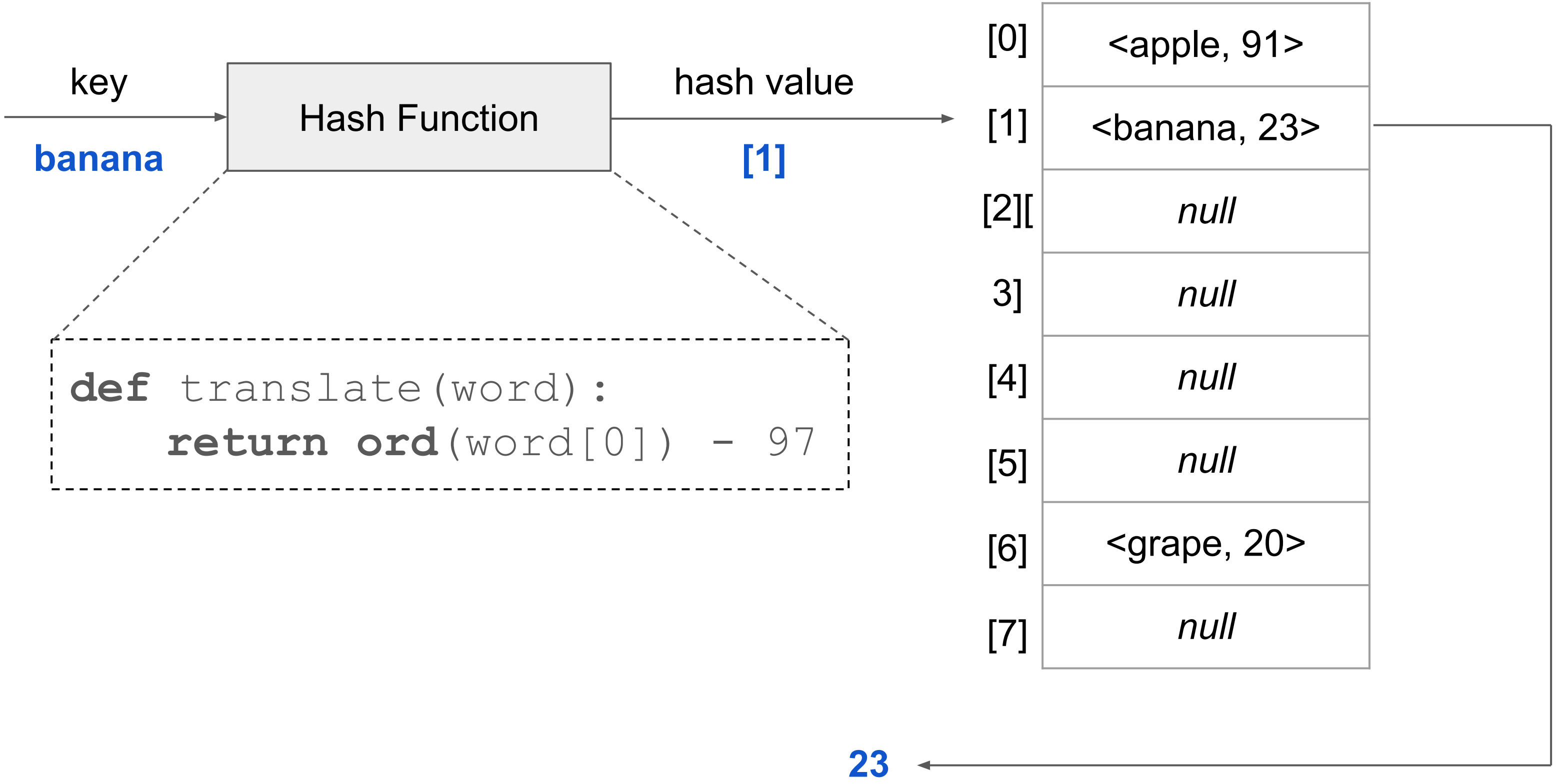
Idea: Use arithmetic operations to locate items in a table (i.e., an array of fixed size) given a key

Three components to hashing:

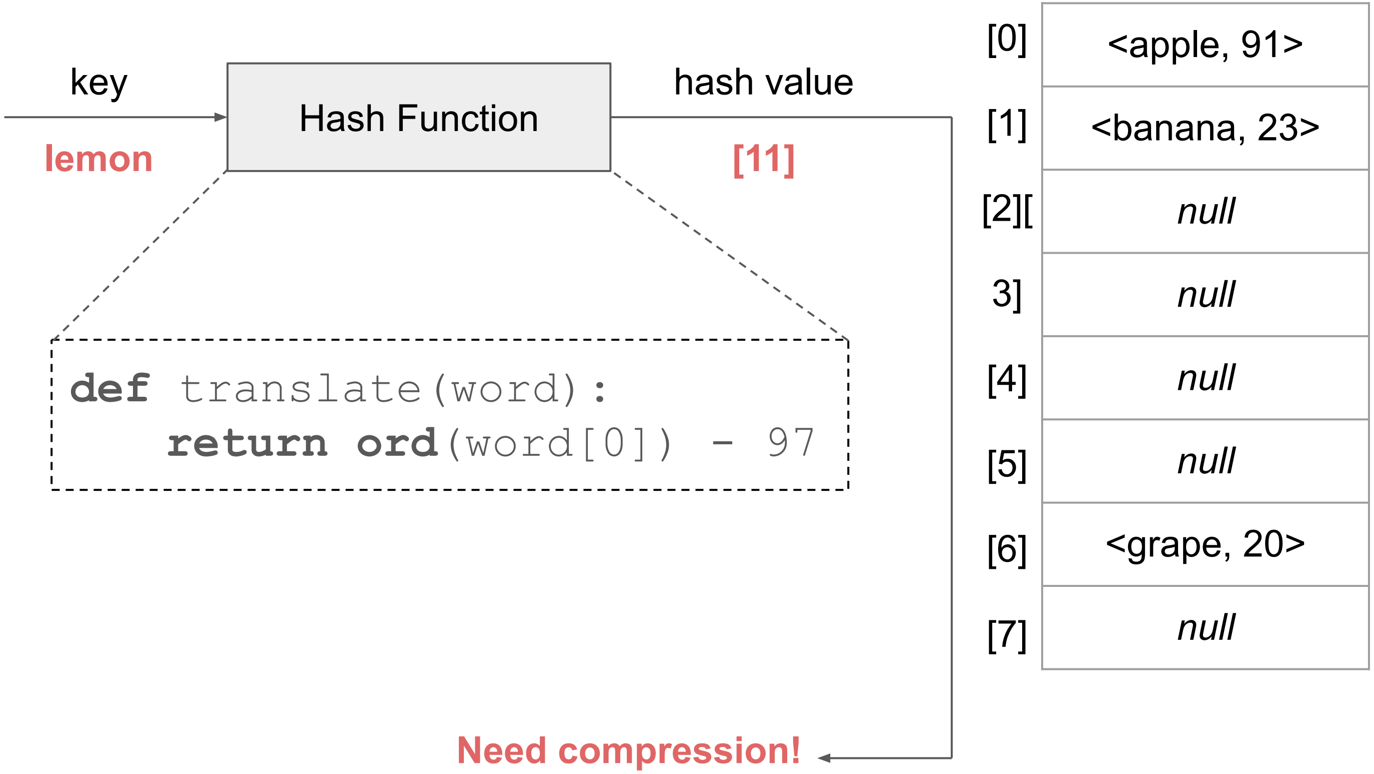
- **Translation**: converts a search key into an integer; **t(***key***)** ⇒ *hashint*
- **Compression**: convert an integer into a valid index; **c(***hashint***)** ⇒ index
- (and one more!)

A **hash function** combines translation and compression: **h(***key***) = c(t(**key**))**

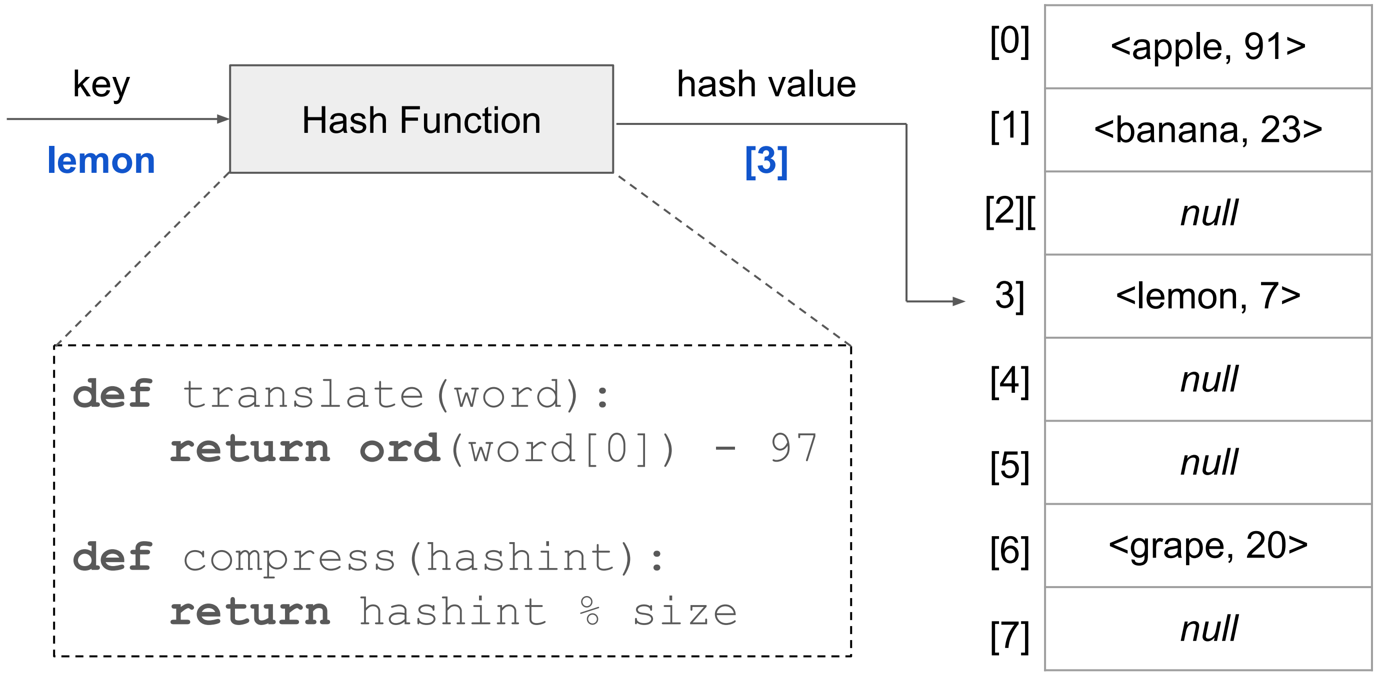
Let’s Hash Strings!



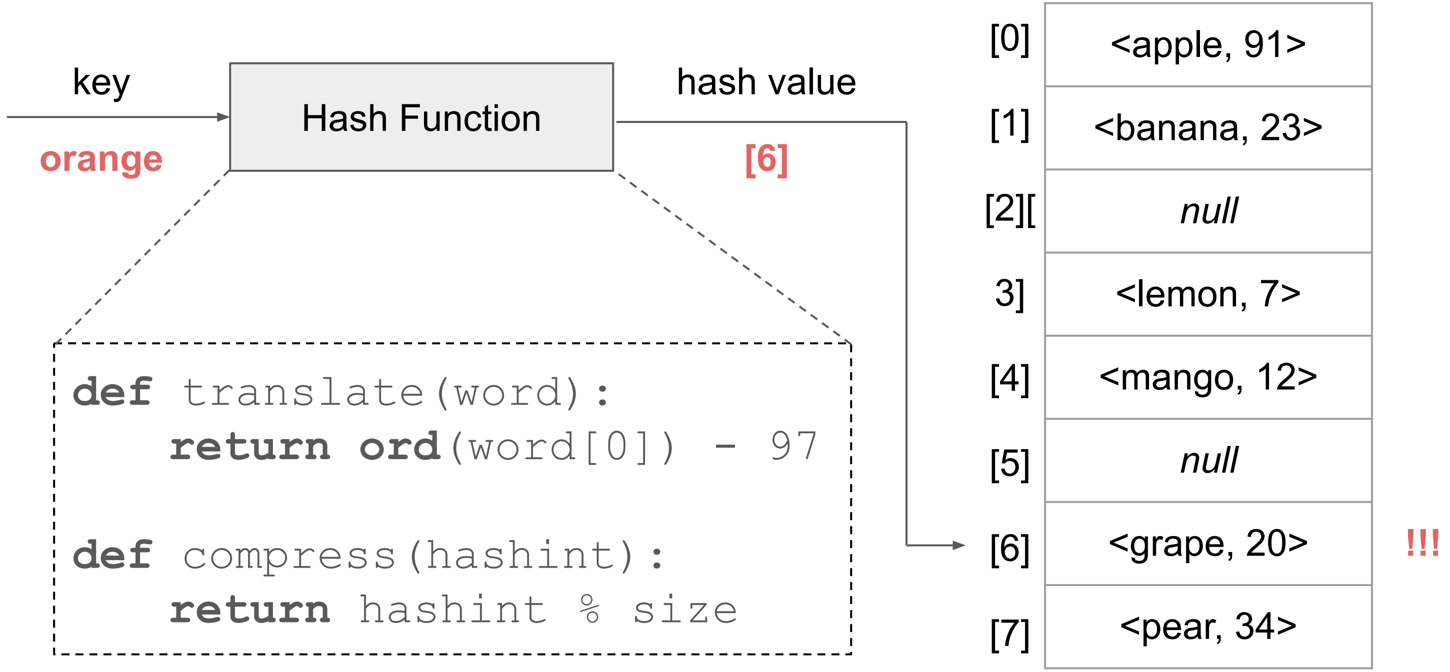
A Potential Problem…



Adding Compression



Yet Another Problem…



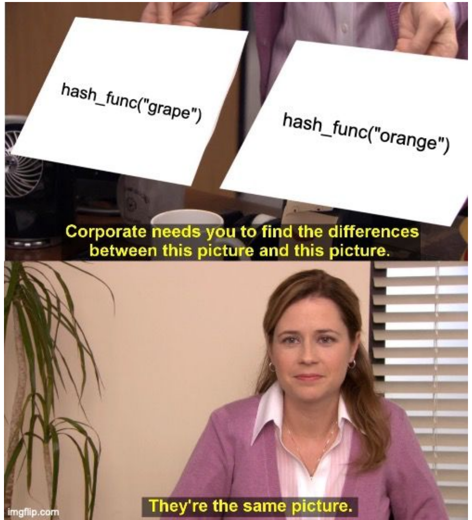
Dealing with Collisions

Collision: Two (or more) distinct keys hashing to the same index

key → Hash Function → hash value

grape
orange

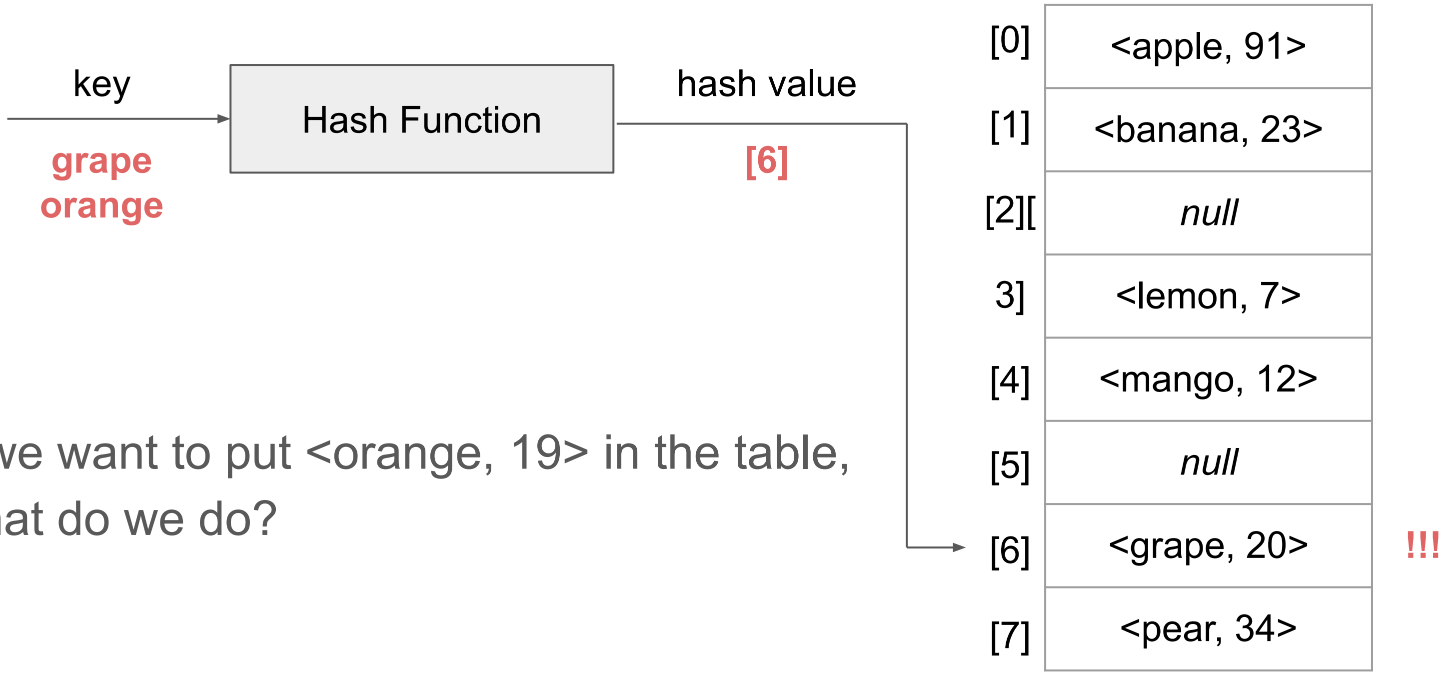
[6]



[0]	<apple, 91>
[1]	<banana, 23>
[2][	*null*
3]	<lemon, 7>
[4]	<mango, 12>
[5]	*null*
[6]	<grape, 20> !!!
[7]	<pear, 34>

Dealing with Collisions

Collision: Two (or more) distinct keys hashing to the same index



If we want to put <orange, 19> in the table, what do we do?

Collision Resolution: Separate Chaining

Imagine: You are shopping for rice at a grocery store. If you don't want to go over the thousands of items in the grocery store, how do you find the rice *quickly*?

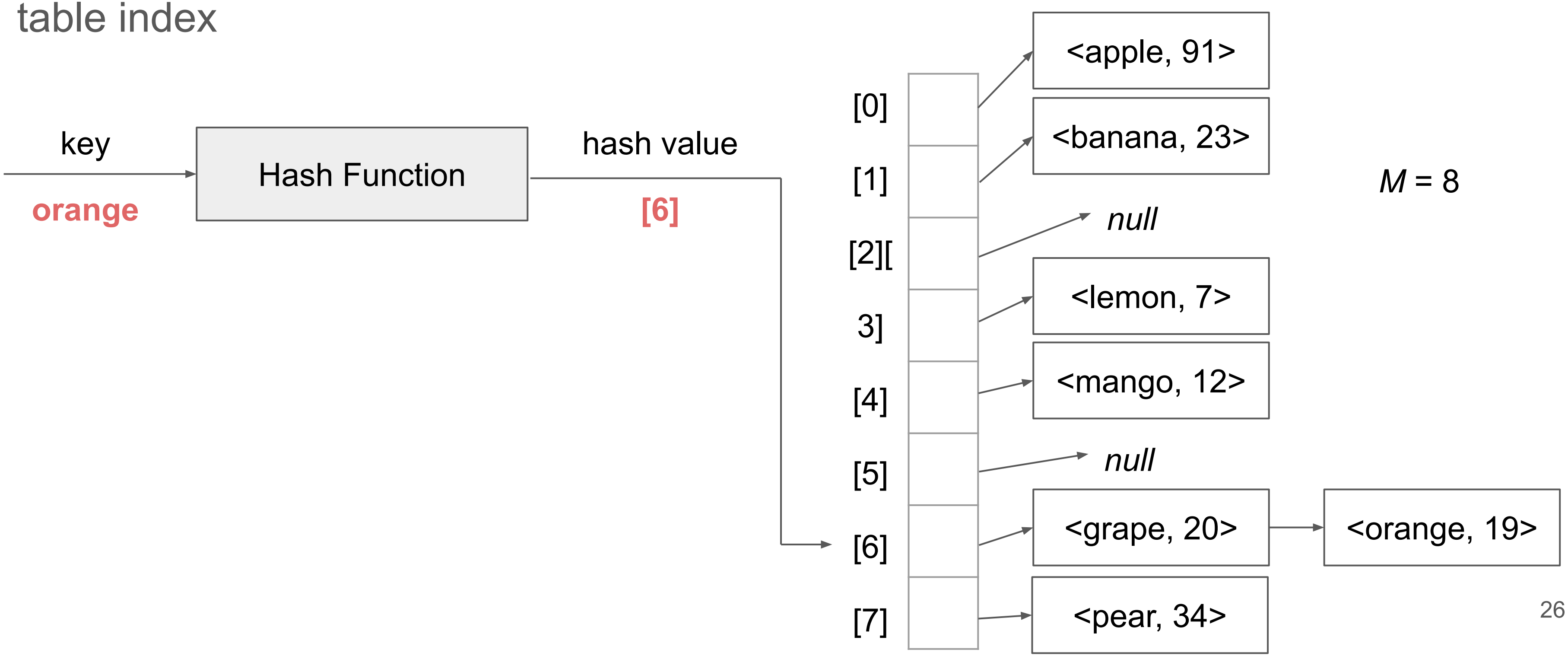
Collision Resolution: Separate Chaining

Separate Chaining: Resolve collisions by maintaining *M* linked lists, one for each table index

- Hashing maps key to integer *i* between 0 and M-1 (inclusive)
- Insertion puts the key-value pair at the front of the i^{th} chain (if not already there)
- Searching requires only looking at the i^{th} chain

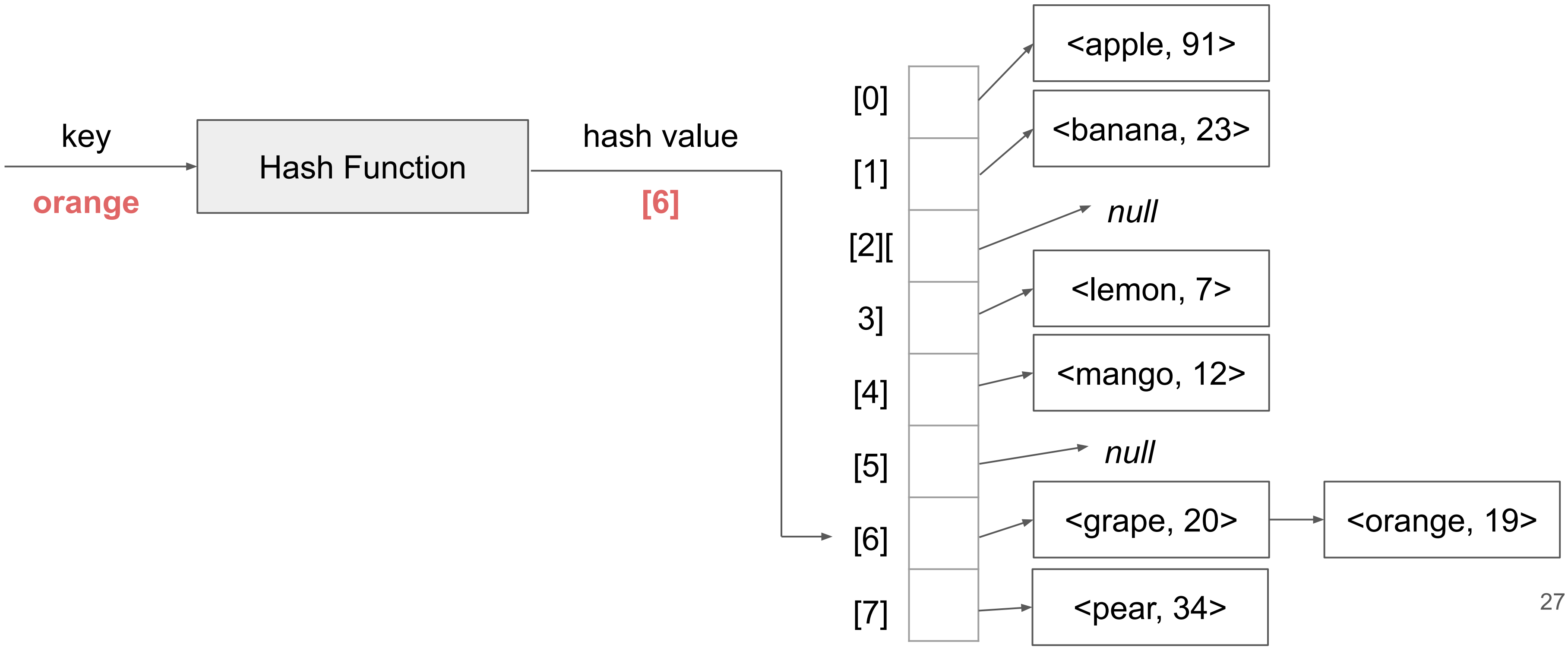
Collision Resolution: Separate Chaining

Separate Chaining: Resolve collisions by maintaining *M* linked lists, one for each table index



Deletion in Separate Chaining Tables

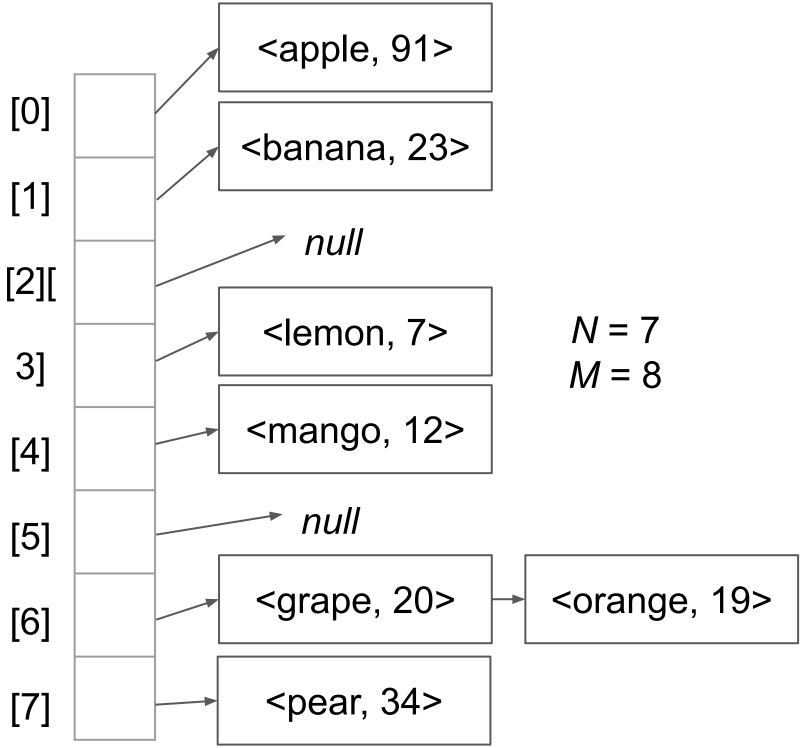
How do we delete <grape, 20> from the table?



Separate Chaining Properties

Reduces the number of comparisons for a sequential search by a factor of M (on average), using extra space for M linked lists

For a table with N keys, need to linear search N/M items on average if a collision occurs



Separate Chaining Properties

If the hash function is good, in a table with M lists (table indices) and N keys, the probability that the number of keys in each list is within a constant factor of N/M is extremely close to 1 (i.e., the average size of each linked list is close to 1)

Implications: Number of probes for search and insert is proportional to N/M

- M too large → too many empty chains (wasted memory)
- M too small → chains too long (wasted time)
- Want $M \approx N$ → constant-time operations on average

Separate Chaining Time Complexity

- Search: **O(*N/M*)**
- Insertion: **O(*N/M*)**
  - What if duplicate keys are allowed?
- Removal: dependent upon search, **O(*N/M*)**

Recall: If we have a good hash function, $N/M \approx 1 \rightarrow$ **O(1)** operations on average

Demo: Separate Chaining in Action

Desirable Properties in a Hash Function

Talk to your neighbors:

- What are properties that **any** hash function **must** satisfy?
- What are properties that a **good** hash function **should** satisfy?

Desirable Properties in a Hash Function

Any hash function must:

- Compute a hash value for every key
- Compute the same hash value for the same key every time
- Be very efficient to compute (!!!)

A “good” hash function should:

- Distribute keys evenly across the hash table
- Use all of the input data
- Generate very different hash value for similar keys

A Better String Hash

- Sum up the ASCII values of characters
  - What could go wrong?

A Better String Hash

- Sum up the ASCII values of characters
  - What could go wrong?

- Better still, take into account the position of each character
  - Recall: The decimal numbers 123 and 321 have different values
    - **123** = **1** * 100 + **2** * 10 + **3** * 1
    - **321** = **3** * 100 + **2** * 10 + **1** * 1
  - Similarly, we want “spot” and “stop” to hash to different values
    - “**spot**” = “**s**” * 1000 + “**p**” * 100 + “**o**” * 10 + “**t**” * 1
    - “**stop**” = “**s**” * 1000 + “**t**” * 100 + “**o**” * 10 + “**p**” * 1

Hash Tables Are Everywhere!

Databases

Password Verification

Compilers

Collision Resolution: Open Addressing

Imagine: You are trying to park a car, and your “usual spot” (closest to your classroom building) is taken. Where do you park? How do you find the car when returning?

Collision Resolution: Open Addressing

Open Addressing: When a new key collides, find the next empty index and store the entry there

Linear Probing: Look for the next empty index linearly

- Hashing maps key to integer *i* between 0 and M-1 (inclusive)
- Insertion puts the key-value pair at index *i* if free, and tries *i* + 1, *i* + 2 (etc.) to find the next empty index for insertion

Collision Resolution: Open Addressing

Linear Probing: Look for the next empty index linearly

- Searching (or probing) can result in three outcomes
  - Empty: No data found at index
  - Hit: Found occupied location with an entry matching the search key
  - Full: Found occupied location, but entry key does not match search key

  If a search results in “full”, then look at the next index – wrapping around if necessary – until we get a “hit” (search successful) or “empty” (search failed)

Collision Resolution: Open Addressing

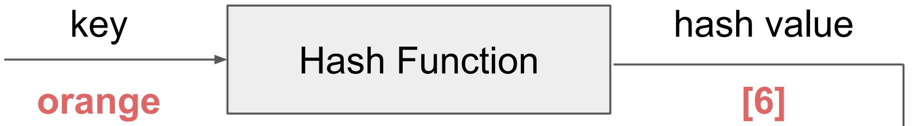
Linear Probing: Look for the next empty index linearly

- Searching (or probing) can result in three outcomes
  - Empty: No data found at index
  - Hit: Found occupied location with an entry matching the search key
  - Full: Found occupied location, but entry key does not match search key

  If a search results in “full”, then look at the next index – wrapping around if necessary – until we get a “hit” (search successful) or “empty” (search failed)

If the hash function is good, in a table of size M containing N entries, we want $N/M \approx 0.5$ for optimal performance

Collision Resolution: Linear Probing

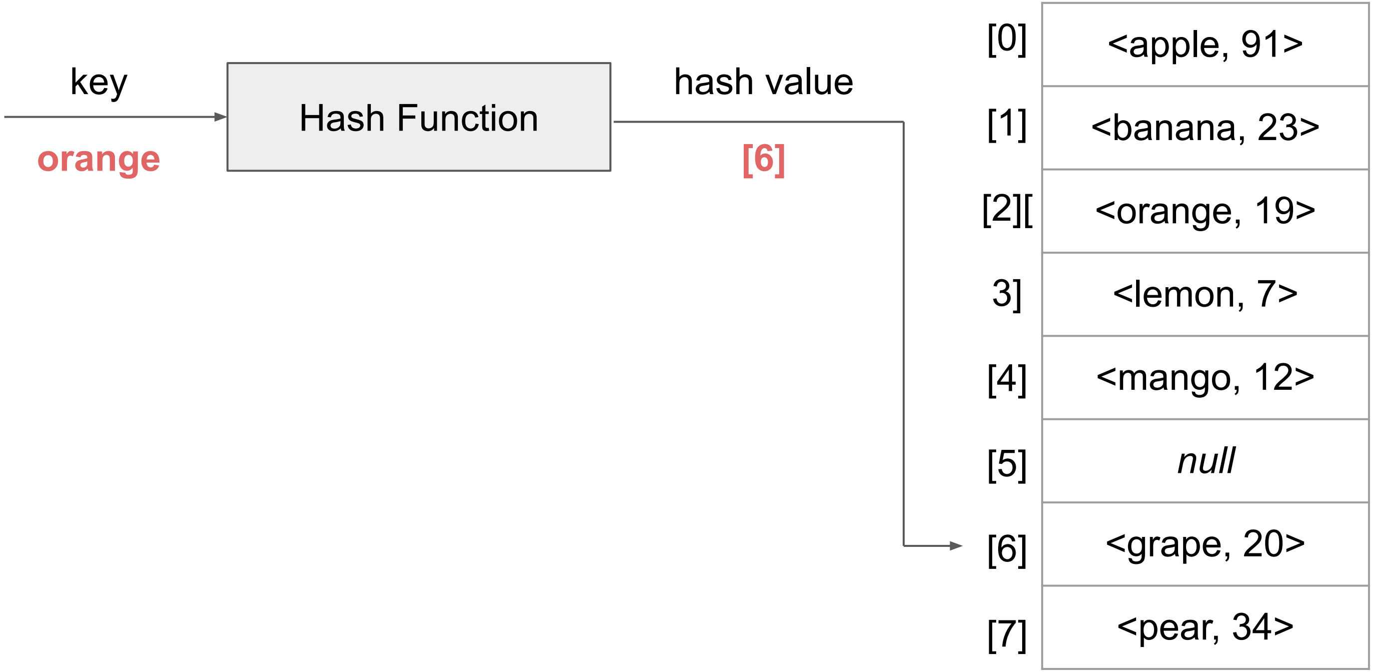


[0]	<apple, 91>
[1]	<banana, 23>
[2][	<orange, 19>
3]	<lemon, 7>
[4]	<mango, 12>
[5]	*null*
[6]	<grape, 20>
[7]	<pear, 34>

If we want to put <orange, 19> in the table, what do we do?

Deletion in Linear Probing Tables

How do we delete <grape, 20> from the table?



Deletion in Linear Probing Tables

Demo: Linear Probing in Action

Questions?

Thank you! :)

Motivating Problem

Integer Count: Given a list of integers, return the number of times each integer appears.

Example: [5, 7, 8, 8, 5, 0, 1, 0, 8]

Result: 0→2, 1→1, 5→2, 7→1, 8→3

```
def intCount(intList):
    counts = Dict()
    for num in intList:
        if num in counts: counts[num] += 1
        else: counts[num] = 0
    return counts
```

Linear Probing Properties

If the hash function is good, in a table of size *M* containing *N* = α * *M* keys, the average number of probes required for search hits and misses is about

$$\frac{1}{2}\left(1 + \frac{1}{1-a}\right) \quad \text{for search hits}$$

$$\frac{1}{2}\left(1 + \frac{1}{(1-a)^2}\right) \quad \text{for search misses}$$

Implications:

- *M* too large → too many empty array entries
- *M* too small → search time blows up
- Mathematically, want α = *N*/*M* ≈ 0.5 for optimal performance