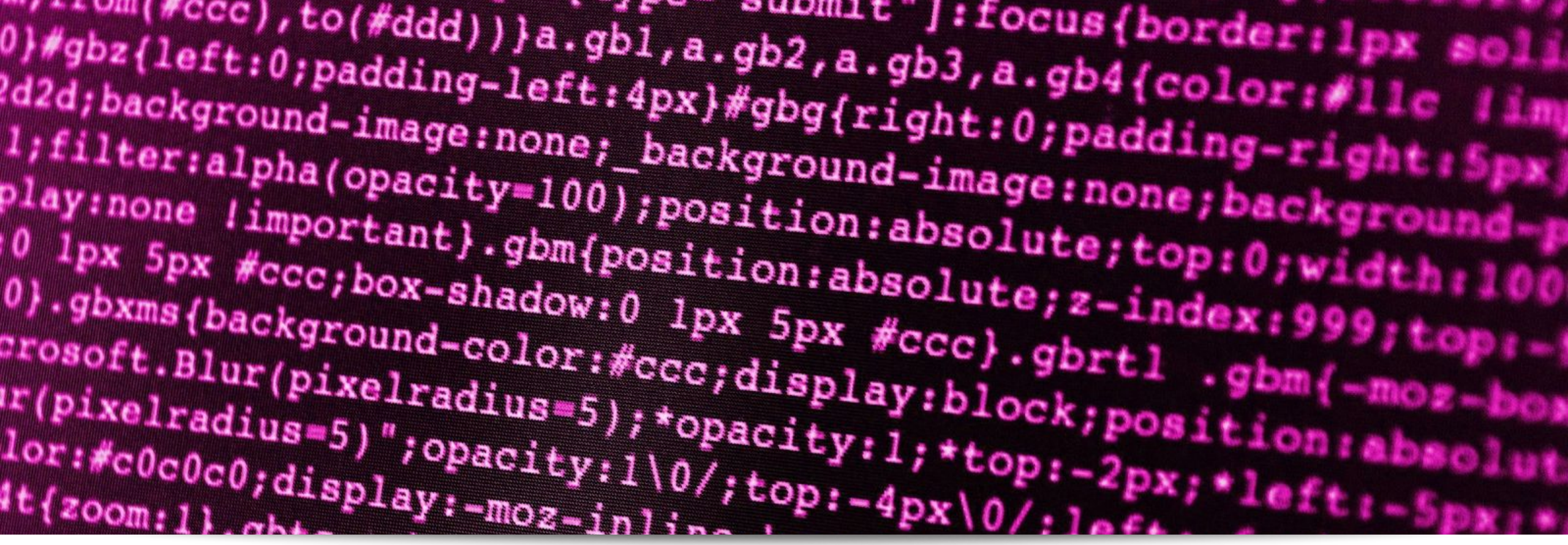


07

October 29th-November 4th, 2024

Collision Resolution Methods

Announcements

- Midterm results are released
  - Check Gradescope for multiple choice and free response
  - Come to OH with questions
  - Regrade Requests are open, close **11/1 at 12 PM**
- Project 3 due **Tuesday, November 12th** at **11:59pm**
- Lab 7 (handwritten problem) due IN LAB by **Monday, November 4th**
- Lab 7 (autograder and quiz) due **November 11th** at **11:59pm**
- A reminder, the handwritten problem can be found in separate file on the Canvas (lab/lab07/Replace Words Written Problem/EECS 281 Lab 7 Written Problem.pdf)

Agenda

- Recap on Hash Tables and Hashing Functions
- Why we need Collision Resolution?
- Methods of Collision Resolution
  - Separate Chaining
  - Open Addressing (e.g. forms of probing)
- Examples and Exercises for Handling Collisions
- Analyzing the Performance of a Hash Table
- Dynamic Hashing
- Handwritten Problem

Lab 6 Handwritten Solution

Handwritten Problem

Given an array of **distinct** integers, find if there are two pairs (a, b) and (c, d) such that a+b=c+d, and a, b, c, and d are distinct elements. If there are multiple elements, the function should print **all** pairs that work. You may assume that for any pair (a, b), there is at most one other pair (c, d) that sums to a+b.

Examples:

Input: `[3, 4, 7, 1, 2, 8]`

Output:

```
(3, 2) and (4, 1)
(3, 7) and (2, 8)
(3, 8) and (4, 7)
(7, 2) and (1, 8)
```

Input: `[65, 30, 7, 90, 1, 9, 8]`

Output (nothing):

Expected Runtime: $O(n^2)$

```
// Prints out all different pairs in input_vec that have same sum.
void two_pair_sums(const vector<int> &input_vec, ostream& out);
```

Solution

```
void two_pair_sums(const vector<int> &input_vec, ostream &out) {
    unordered_map<int, pair<int, int>> sum_to_pair_map;

    for (int c = 0; c < input_vec.size(); ++c) {
        // `d` is (c + 1) to stop d == c scenarios.
        for (int d = c + 1; d < input_vec.size(); ++d) {
            int sum = input_vec[c] + input_vec[d];
            auto iter = sum_to_pair_map.find(sum);
            // Case where sum was NOT found. Insert into map.
            if (iter == sum_to_pair_map.end()) {
                sum_to_pair_map[sum] = make_pair(input_vec[c], input_vec[d]);
            }
            else {  // Case where sum was found.
                int a, b;
                tie(a, b) = iter->second;
        // erase
                out << a << " + " << b << " = " << input_vec[c] << " + " << input_vec[d]
                    << " = " << sum << '\n';
            }
        }
    }
```

Use `.find()` instead of `[]` to prevent inserting empty elements

Solution

```
void two_pair_sums(const vector<int> &input_vec, ostream &out) {
    unordered_map<int, pair<int, int>> sum_to_pair_map;

    for (int c = 0; c < input_vec.size(); ++c) {
        // `d` is (c + 1) to stop d == c scenarios.
        for (int d = c + 1; d < input_vec.size(); ++d) {
            int sum = input_vec[c] + input_vec[d];
            // Case where sum was NOT FOUND. Insert into map.
            if (sum_to_pair_map.count(sum) == 0) {
                sum_to_pair_map[sum] = make_pair(input_vec[c], input_vec[d]);
            }
            else {  // Case where sum was FOUND.
                int a, b;
                tie(a, b) = sum_to_pair_map[sum];
                out << a << " + " << b << " = " << input_vec[c] << " + " << input_vec[d]
                    << " = " << sum << '\n';
            }
        }
    }
}
```

Alternative:
`.count()` returns how many times the element is in the map (either 1 or 0)

Solution

```
void two_pair_sums(const vector<int> &input_vec, ostream &out) {
    unordered_map<int, pair<int, int>> sum_to_pair_map;

    for (int c = 0; c < input_vec.size(); ++c) {
        // `d` is (c + 1) to stop d == c scenarios.
        for (int d = c + 1; d < input_vec.size(); ++d) {
            int sum = input_vec[c] + input_vec[d];
            // Case where sum was NOT FOUND. Insert into map.
            if (not sum_to_pair_map.contains(sum)) {
                sum_to_pair_map[sum] = make_pair(input_vec[c], input_vec[d]);
            }
            else {  // Case where sum was FOUND.
                auto [a, b] = sum_to_pair_map[sum];
                out << a << " + " << b << " = " << input_vec[c] << " + " << input_vec[d]
                    << " = " << sum << '\n';
            }
        }
    }
}
```

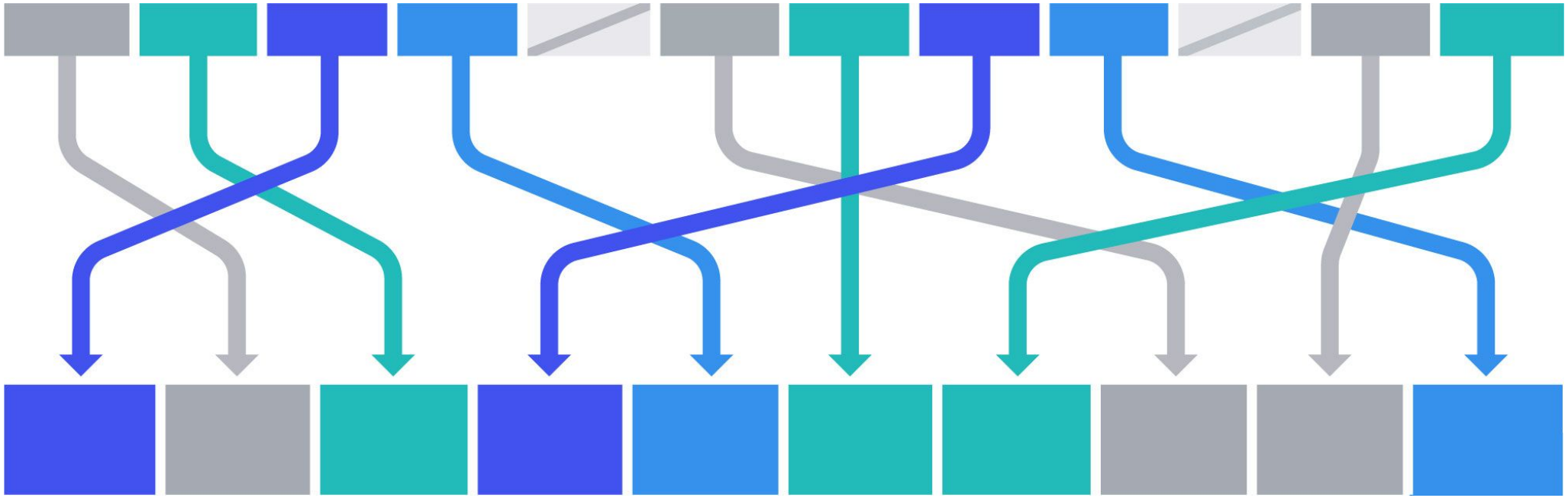
Better alternative (since C++20):
`.contains()` does exactly what you think it does

Another improvement (since C++17): use a structured binding

Hash Functions

What is a Hash Table?

- Implements the unordered_map and unordered_set abstract data types.
- A data structure that aims for **average** case $\Theta(1)$ insert, lookup, delete.
- Uses a **hash function** to map keys to an associated hash value (an integer).

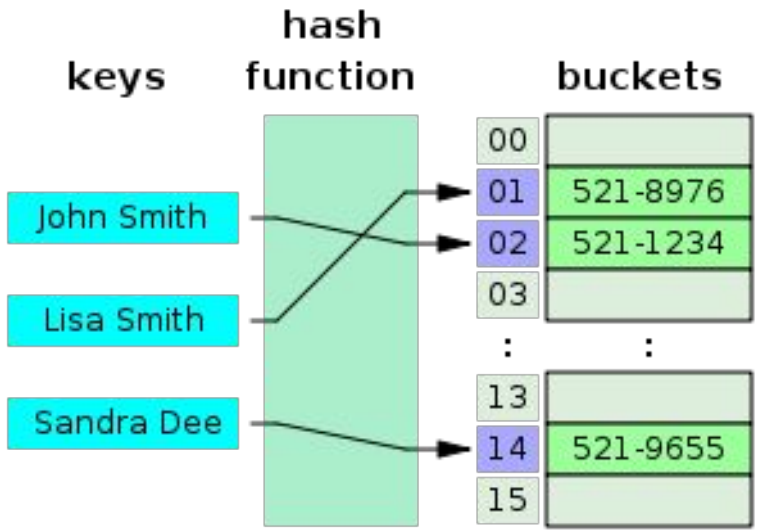


Hash Functions

WE WANT YOU

YOU need to use the modulo operator (%) to keep hash values within the capacity of the hash table. The hash function just returns a hash, but does not guarantee it fits within your range:

```
size_t bucket_index = hash_of(key) % number_buckets;
```



TO USE %

Hash Function Invariants

- **Consistency:** If two keys are equal, they must hash to the same thing!
  - Otherwise, you'll look in different places for each of them while you try to find them in the hash table!

- **Efficiency:** Hash Functions should also be computable in $\Theta(1)$ time, since otherwise the purpose of using a hash table is defeated.
  - They might depend on the size of the key, but they shouldn't depend on the size of the whole table.

Hash Function Invariants

```
size_t hash1(string s){
  return 1;
}
```

```
size_t hash2(string s){
  if (s.empty())
     return 0;
  return (s[0]-'A');
}
```

```
size_t hash3(string s){
  size_t result = 0;
  for (size_t i = 0; i < s.size(); i++) {
     int a = i*(s.at(i)-'A')*100;
     result += size_t(a);
  }
  return result;
}
```

What is the problem with these hash functions?

Hash Function Invariants

```
size_t hash1(string s){
  return 1;
}
```

```
size_t hash2(string s){
  if (s.empty())
     return 0;
  return (s[0]-'A');
}
```

```
size_t hash3(string s){
  size_t result = 0;
  for (size_t i = 0; i < s.size(); i++) {
     int a = i*(s.at(i)-'A')*100;
     result += size_t(a);
  }
  return result;
}
```

What is the problem with these hash functions?

Many *collisions*: different keys get the same hash

Hash Tables Exercise

N=10, with hash function:

```
int hash2(string s){
  if(s.empty())
     return 0;
  return (s[0]-'A');
}
```

What does the hash function return for the following keys?

“Zebra”
“Dog”
“Ferret”

Hash Tables Exercise

N=10, with hash function:

```
int hash2(string s){
  if(s.empty())
     return 0;
  return (s[0]-'A');
}
```

What does the hash function return for the following keys?

"Zebra" - **25**
"Dog" - **3**
"Ferret" - **5**

Hash Tables Exercise

N=10, with hash function:

```
int hash2(string s){
  if(s.empty())
     return 0;
  return (s[0]-'A');
}
```

Which bucket would the following keys be inserted at?

"Zebra"
"Dog"
"Ferret"

Hash Tables Exercise

N=10, with hash function:

```
int hash2(string s){
  if(s.empty())
     return 0;
  return (s[0]-'A');
}
```

Which bucket would the following keys be inserted at?

"Zebra" - **5**
"Dog" - **3**
"Ferret" - **5**

Since "Zebra" and "Ferret" map to the same bucket, we have a **hash collision**!

Collision Resolution

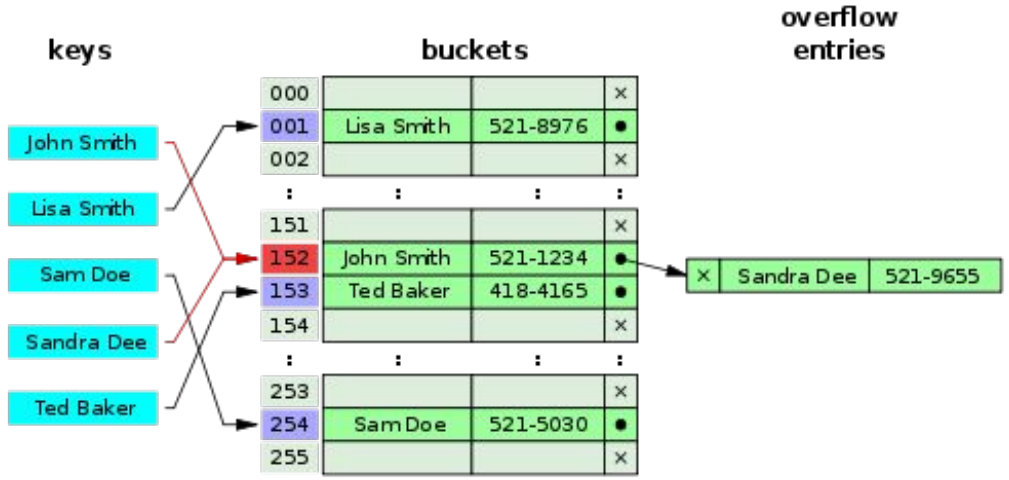
Collision Resolution

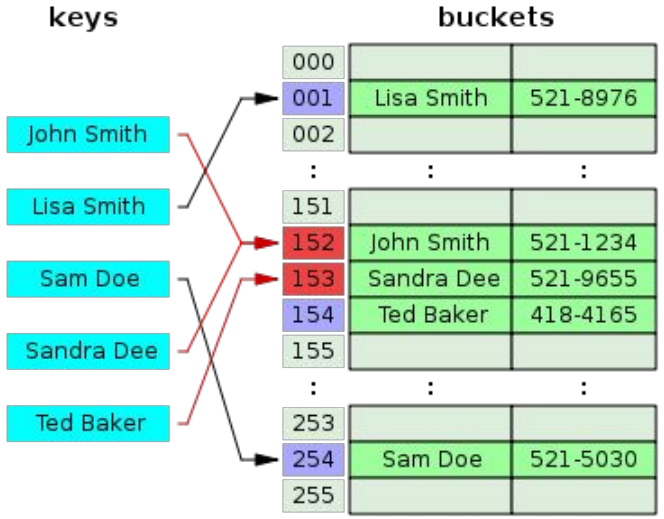
Separate Chaining:

- Store colliding key-value pairs in a linked list for that bucket

Open Addressing:

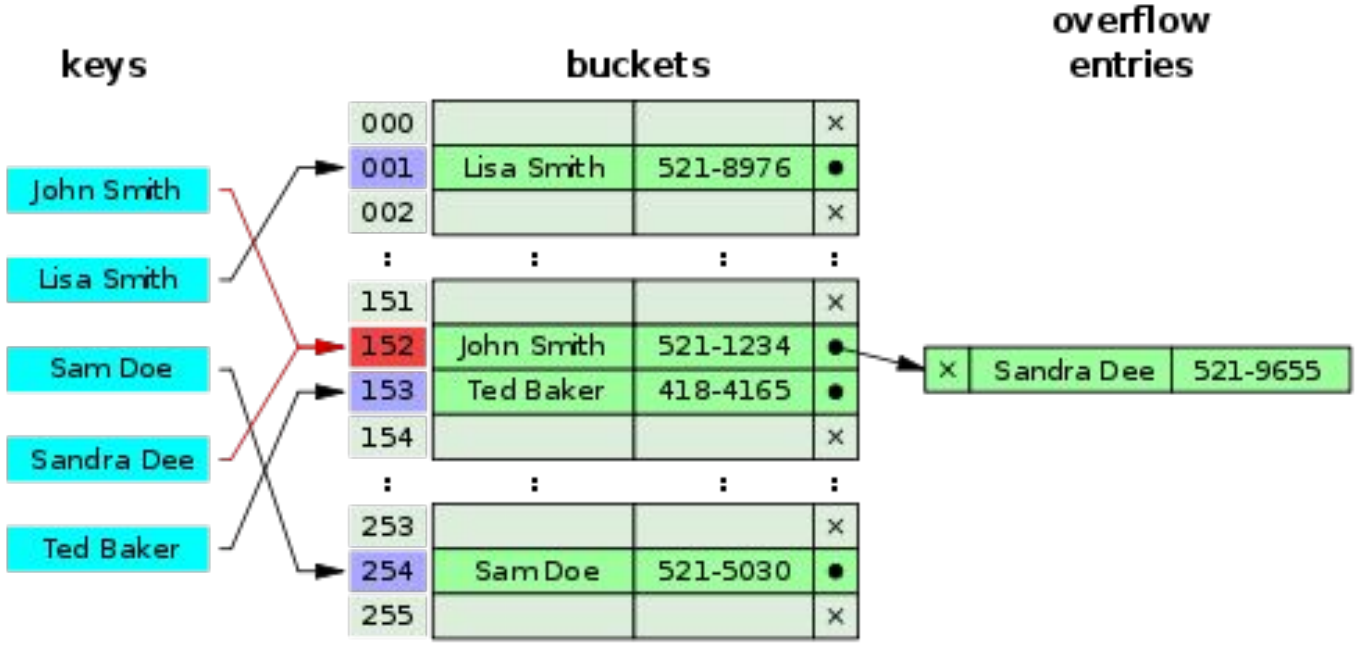
- Store colliding key-value pairs in another bucket/location





Separate Chaining

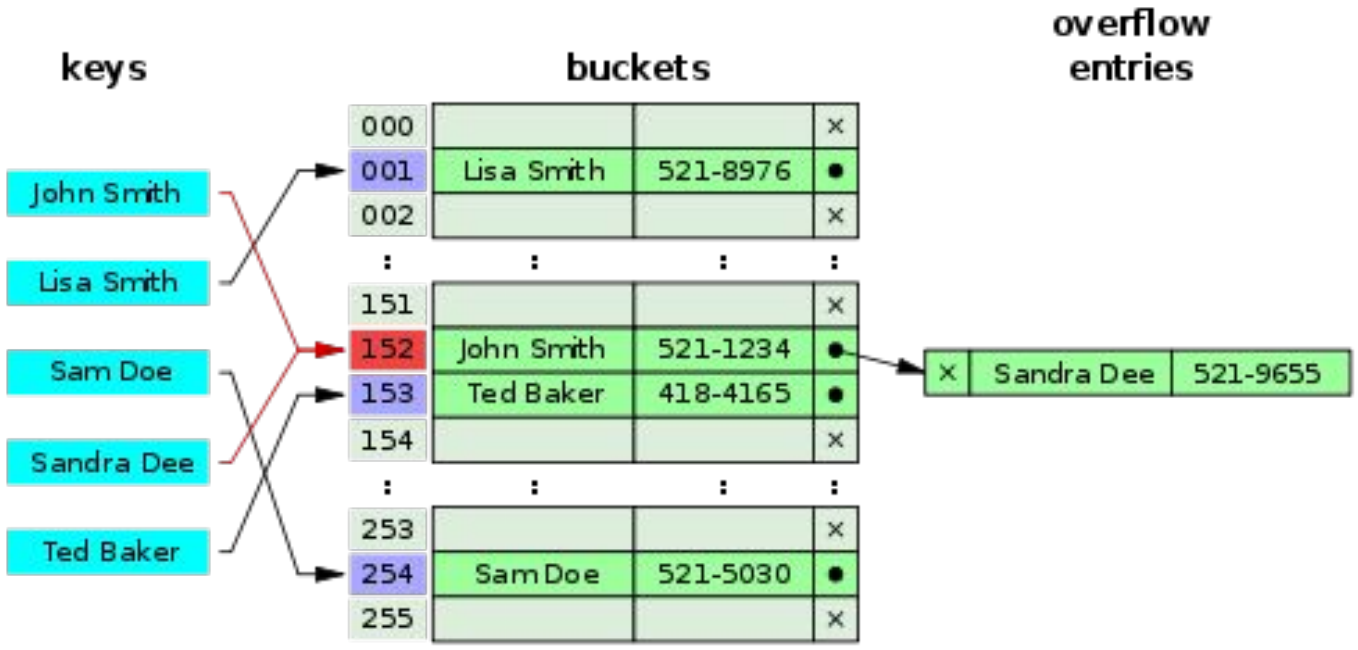
- **Chaining** uses linked lists to allow buckets to hold multiple elements.
- Upon a collision, the new element is added to the bucket's linked list.



What is the best case, average case, worst case of insert or look up?

Separate Chaining

- **Chaining** uses linked lists to allow buckets to hold multiple elements.
- Upon a collision, the new element is added to the bucket's linked list.



What is the best case, average case, worst case of insert or look up? Θ(1), Θ(1), Θ(n)

Open Addressing: Linear Probing

Let k be the key, H(k) be the hash value

Linear Probing

- Interval *j* between probes is fixed (usually, j = 1)
- Try buckets H(k), H(k) + 1, H(k) + 2, H(k) + 3, H(k) + 4, ...
  (wrap around if needed)

$$(H(k) + ij) \% N$$

H(k) - hash function
i - collision number (starts at 0)
N - size of our hash table
j - constant stride size, always 1 (as far as 281 is concerned)

Linear Probing Exercise

number of buckets N=6

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

If we insert these 4 items in this order, using **linear probing** to resolve collisions, where do the keys end up?

“cat”
“cop”
“ear”
“cartographer”

Linear Probing Exercise

number of buckets N=6

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

If we insert these 4 items in this order, using **linear probing** to resolve collisions, where do the keys end up?

“cat”	**hash: 2**
“cop”	**hash: 2**
“ear”	**hash: 4**
“cartographer”	**hash: 2**

Linear Probing Exercise

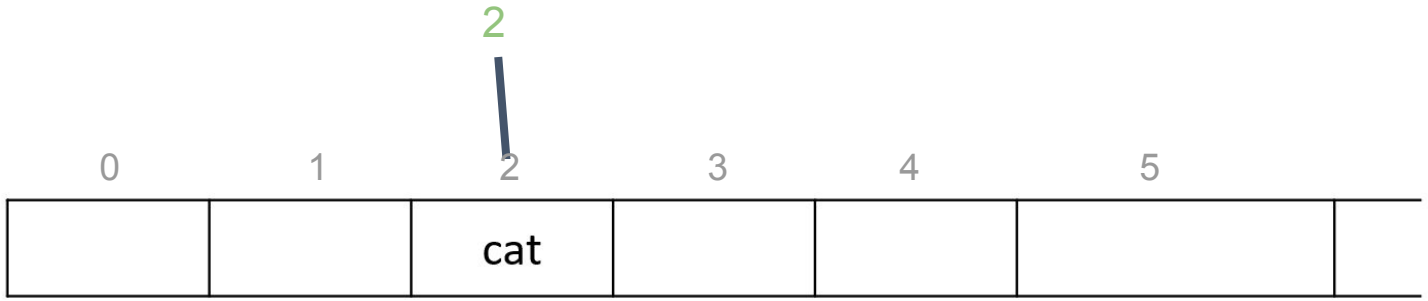
number of buckets N=6

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

If we insert these 4 items in this order, using **linear probing** to resolve collisions, where do the keys end up?

"cat"	**hash: 2**
"cop"	**hash: 2**
"ear"	**hash: 4**
"cartographer"	**hash: 2**



Linear Probing Exercise

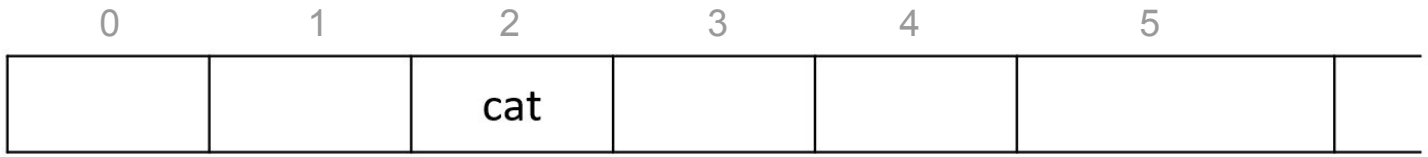
number of buckets N=6

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

If we insert these 4 items in this order, using **linear probing** to resolve collisions, where do the keys end up?

“cat”	**hash: 2**
“cop”	**hash: 2**
“ear”	**hash: 4**
“cartographer”	**hash: 2**



Linear Probing Exercise

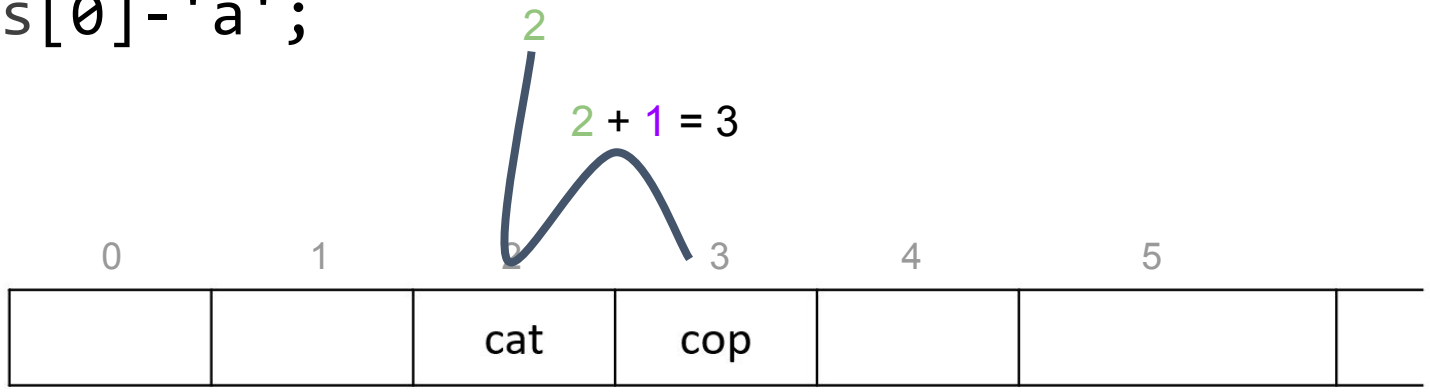
number of buckets N=6

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

If we insert these 4 items in this order, using **linear probing** to resolve collisions, where do the keys end up?

"cat"	**hash: 2**
"cop"	**hash: 2**
"ear"	**hash: 4**
"cartographer"	**hash: 2**



Linear Probing Exercise

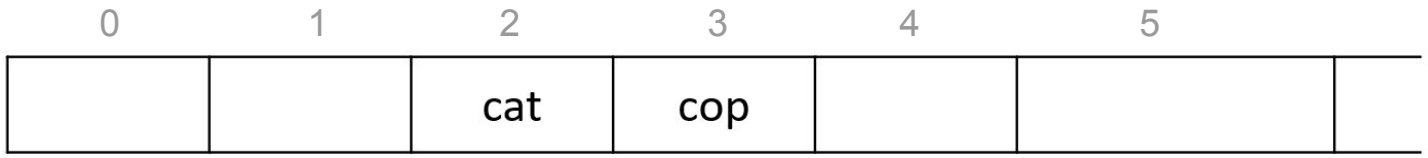
number of buckets N=6

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

If we insert these 4 items in this order, using **linear probing** to resolve collisions, where do the keys end up?

“cat”	**hash: 2**
“cop”	**hash: 2**
“ear”	**hash: 4**
“cartographer”	**hash: 2**



Linear Probing Exercise

number of buckets N=6

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

If we insert these 4 items in this order, using **linear probing** to resolve collisions, where do the keys end up?

"cat"	**hash: 2**
"cop"	**hash: 2**
"ear"	**hash: 4**
"cartographer"	**hash: 2**

Linear Probing Exercise

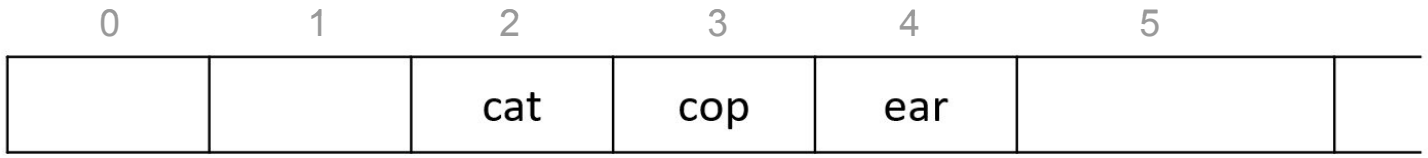
number of buckets N=6

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

If we insert these 4 items in this order, using **linear probing** to resolve collisions, where do the keys end up?

“cat”	**hash: 2**
“cop”	**hash: 2**
“ear”	**hash: 4**
“cartographer”	**hash: 2**



Linear Probing Exercise

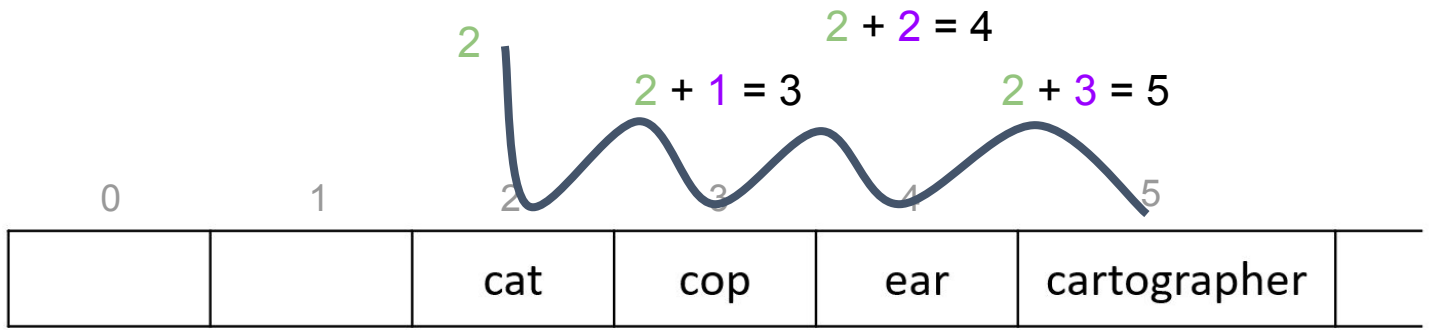
number of buckets N=6

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

If we insert these 4 items in this order, using **linear probing** to resolve collisions, where do the keys end up?

"cat"	**hash: 2**
"cop"	**hash: 2**
"ear"	**hash: 4**
"cartographer"	**hash: 2**



Linear Probing Exercise

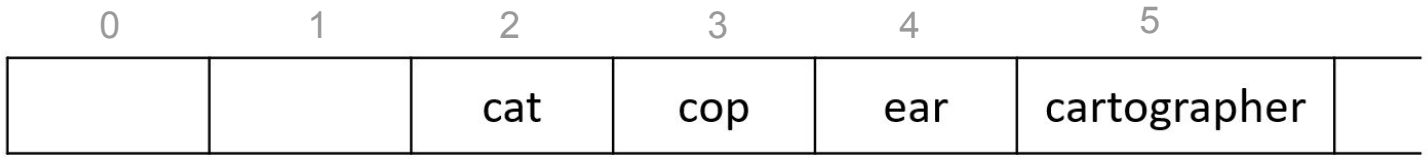
number of buckets N=6

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

If we insert these 4 items in this order, using **linear probing** to resolve collisions, where do the keys end up?

“cat”	**hash: 2**
“cop”	**hash: 2**
“ear”	**hash: 4**
“cartographer”	**hash: 2**



Linear Probing Exercise

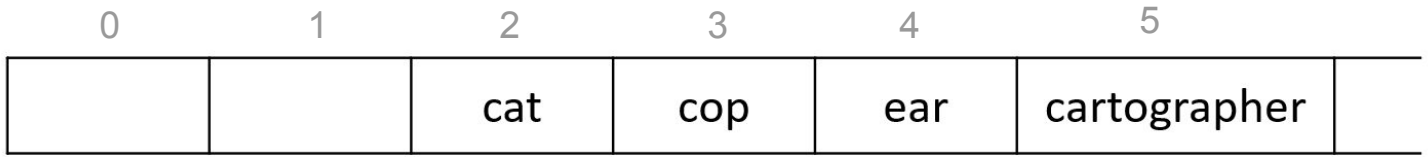
number of buckets N=6

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

Your turn!

If you now insert “duck” into the hash table, where does it end up?



Linear Probing Exercise

number of buckets N=6

Hash function:

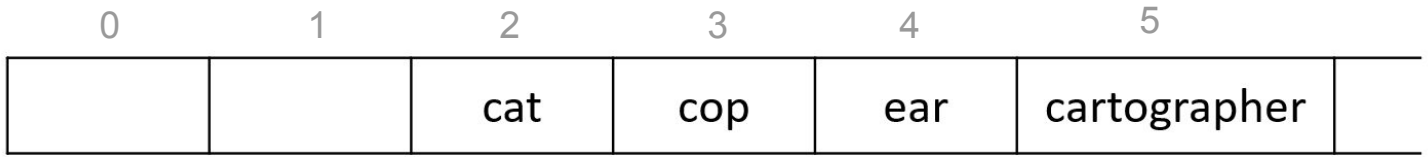
```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

Your turn!

If you now insert “duck” into the hash table, where does it end up?

hash2(“duck”) = 3

(3 + 0) % 6 = 3 = FULL
(3 + 1) % 6 = 4 = FULL
(3 + 2) % 6 = 5 = FULL
(3 + 3) % 6 = 0 = EMPTY! Insert here



Open Addressing: Quadratic Probing

Let k be the key, H(k) be the hash value

Quadratic Probing

- Interval between probes increases quadratically until open spot is found
- Try buckets H(k), H(k) + 1, H(k) + 4, H(k) + 9, H(k) + 16, ...
  (wrap around if needed)

$$(H(k) + i^2 j) \% N$$

H(k) - hash function

i - collision number (starts at 0)

N - size of our hash table

j - constant stride size, always 1 (as far as 281 is concerned)

Quadratic Probing Exercise

number of buckets N=10

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

If we insert these 4 items in this order, using **quadratic probing** to resolve collisions, where do the keys end up?

“cat”	**hash: 2**
“cop”	**hash: 2**
“ear”	**hash: 4**
“cartographer”	**hash: 2**

Quadratic Probing Exercise

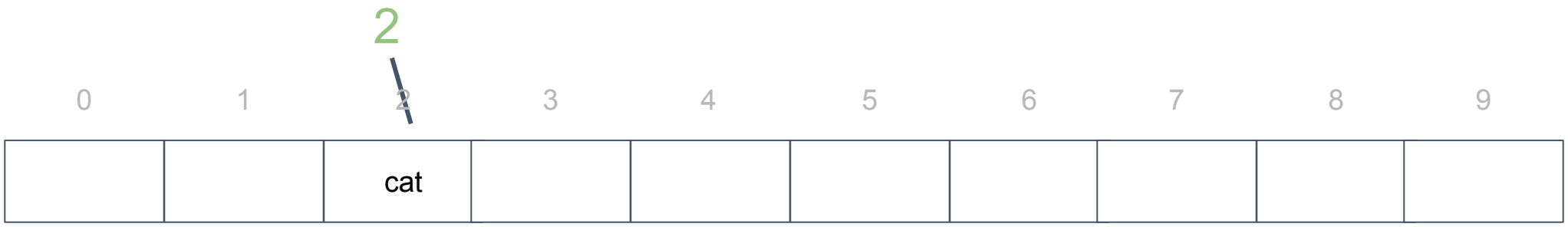
number of buckets N=10

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

If we insert these 4 items in this order, using **quadratic probing** to resolve collisions, where do the keys end up?

"cat"	**hash: 2**
"cop"	**hash: 2**
"ear"	**hash: 4**
"cartographer"	**hash: 2**



Quadratic Probing Exercise

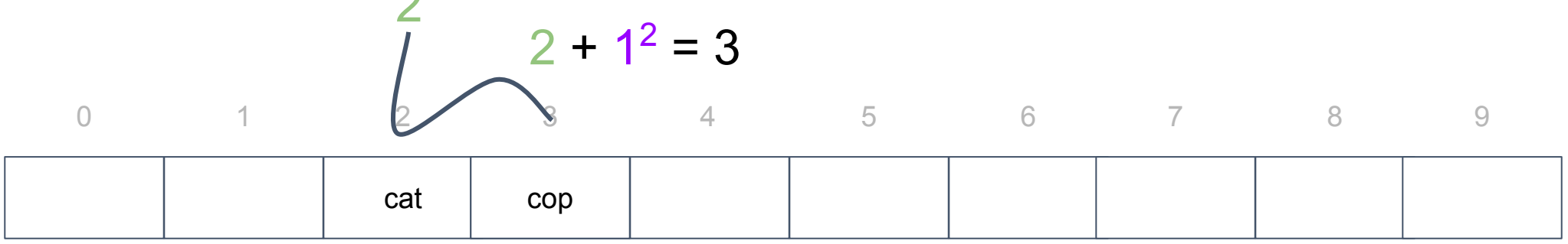
number of buckets N=10

Hash function:

```
int hash2(string s) {
   if(s.empty())
      return 0;
   else
      return s[0]-'a';
}
```

If we insert these 4 items in this order, using **quadratic probing** to resolve collisions, where do the keys end up?

"cat"	**hash: 2**
"cop"	**hash: 2**
"ear"	**hash: 4**
"cartographer"	**hash: 2**



Quadratic Probing Exercise

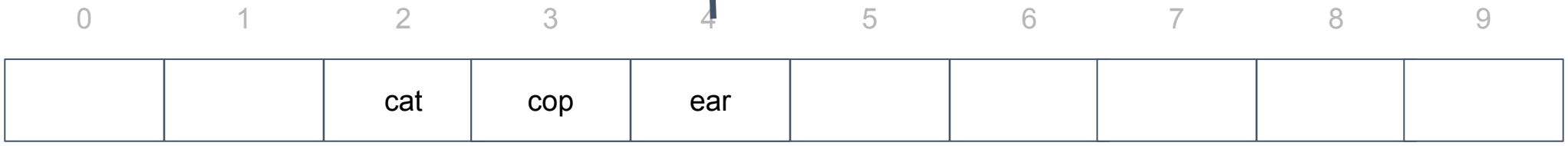
number of buckets N=10

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

If we insert these 4 items in this order, using **quadratic probing** to resolve collisions, where do the keys end up?

"cat"	**hash: 2**
"cop"	**hash: 2**
"ear"	**hash: 4**
"cartographer"	**hash: 2**



Quadratic Probing Exercise

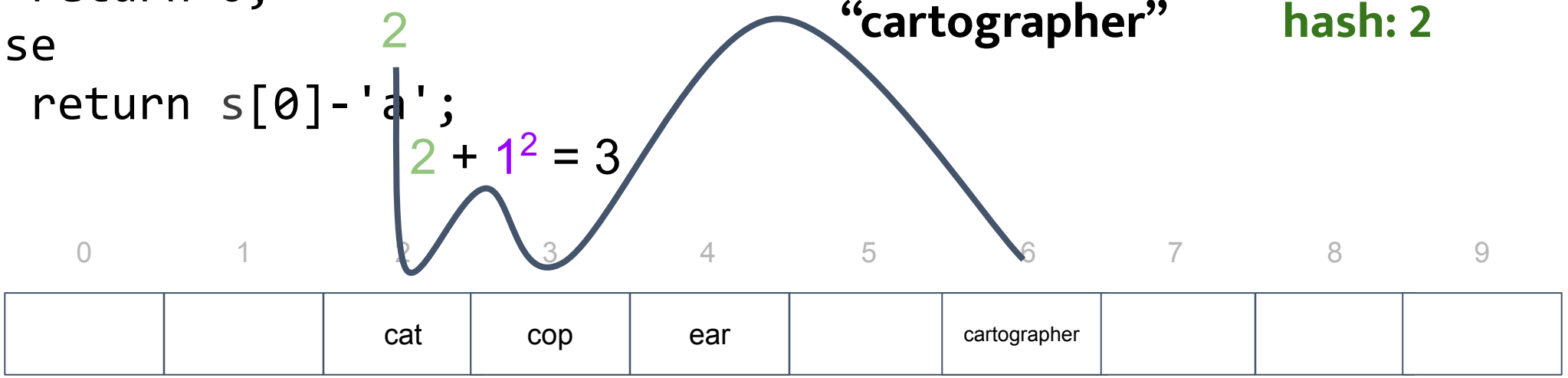
number of buckets N=10

Hash function:

```
int hash2(string s) {
   if(s.empty())
      return 0;
   else
      return s[0]-'a';
}
```

If we insert these 4 items in this order, using **quadratic probing** to resolve collisions, where do the keys end up?

"cat"	**hash: 2**
"cop"	**hash: 2**
"ear"	**hash: 4**
"cartographer"	**hash: 2**



Quadratic Probing Exercise

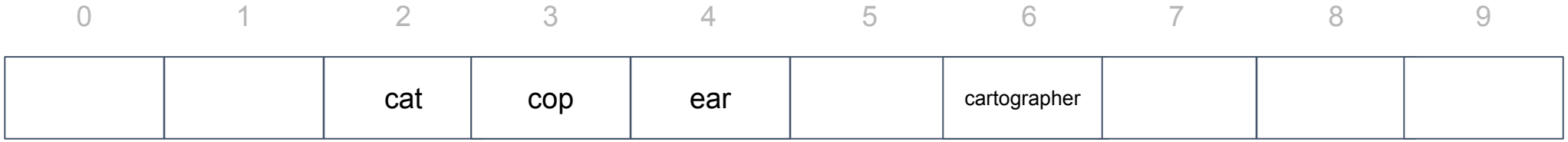
number of buckets N=10

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

If we insert these 4 items in this order, using **quadratic probing** to resolve collisions, where do the keys end up?

“cat”	**hash: 2**
“cop”	**hash: 2**
“ear”	**hash: 4**
“cartographer”	**hash: 2**



Quadratic Probing Exercise

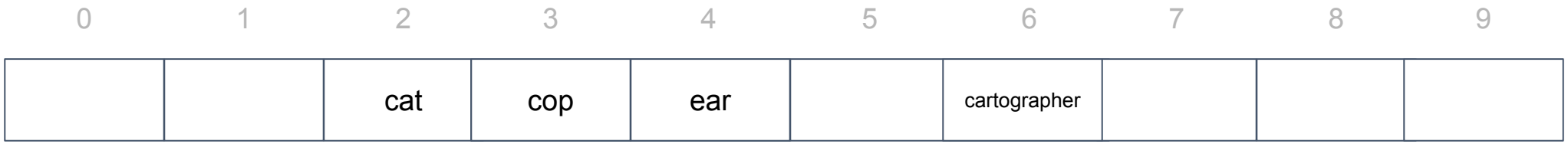
number of buckets N=10

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

Again!

If you now insert “duck” into the hash table, where does it end up?



Quadratic Probing Exercise

number of buckets N=10

Hash function:

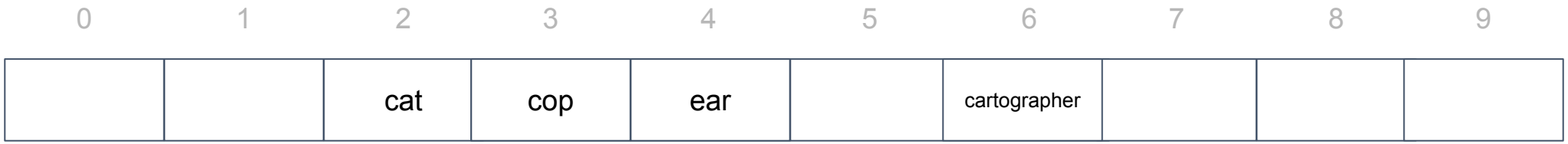
```
int hash2(string s) {
   if(s.empty())
      return 0;
   else
      return s[0]-'a';
}
```

Again!

If you now insert “duck” into the hash table, where does it end up?

hash2(“duck”) = 3

$(3 + 0^2)$ % 10 = 3 = FULL
$(3 + 1^2)$ % 10 = 4 = FULL
$(3 + 2^2)$ % 10 = 7 = EMPTY! Insert here



Open Addressing: Double Hashing

Let k be the key, H(k) be the hash value, and F(k) be the 2nd hash value

Double Hashing

- Interval probes is computed using another hash function
- Try buckets H(k), H(k) + F(k), H(k) + 2F(k), H(k) + 3F(k), H(k) + 4F(k), … (wrap around if needed)

$$(H(k) + i * F(k)) \% N$$

H(k) - hash function

i - collision number (starts at 0)

F(k) - second hash function

N - size of the hash table

Open Addressing: Double Hashing

Let k be the key, H(k) be the hash value, and F(k) be the 2nd hash value

Double Hashing

- Interval probes is computed using another hash function
- Try buckets H(k), H(k) + F(k), H(k) + 2F(k), H(k) + 3F(k), H(k) + 4F(k), … (wrap around if needed)

$$(H(k) + i * F(k)) \% N$$

H(k) - hash function

i - collision number (starts at 0)

F(k) - second hash function

N - size of the hash table

Notice: exactly like linear probing, except stride size is dependent on key rather than constant! Colliding keys will usually not collide as badly.

Double Hashing Exercise

number of buckets N=10

```
int hash(string s){
   if(s.empty())
      return 0;
   else
      return s.front()-'a';
}

int hash2(string s){
   if(s.empty())
      return 0;
   else
      return s.back()-'a';
}
```

If we insert these 4 items in this order, using **double hashing** to resolve collisions, where do the keys end up?

“cat”	**hash: 2**	**hash2: 19**
“cop”	**hash: 2**	**hash2: 15**
“ear”	**hash: 4**	**hash2: 17**
“cartographer”	**hash: 2**	**hash2: 17**

Double Hashing Exercise

number of buckets N=10

```
int hash(string s){
   if(s.empty())
      return 0;
   else
      return s.front()-'a';
}

int hash2(string s){
   if(s.empty())
      return 0;
   else
      return s.back()-'a';
}
```

If we insert these 4 items in this order, using **double hashing** to resolve collisions, where do the keys end up?

“cat”	**hash: 2**	**hash2: 19**
“cop”	**hash: 2**	**hash2: 15**
“ear”	**hash: 4**	**hash2: 17**
“cartographer”	**hash: 2**	**hash2: 17**

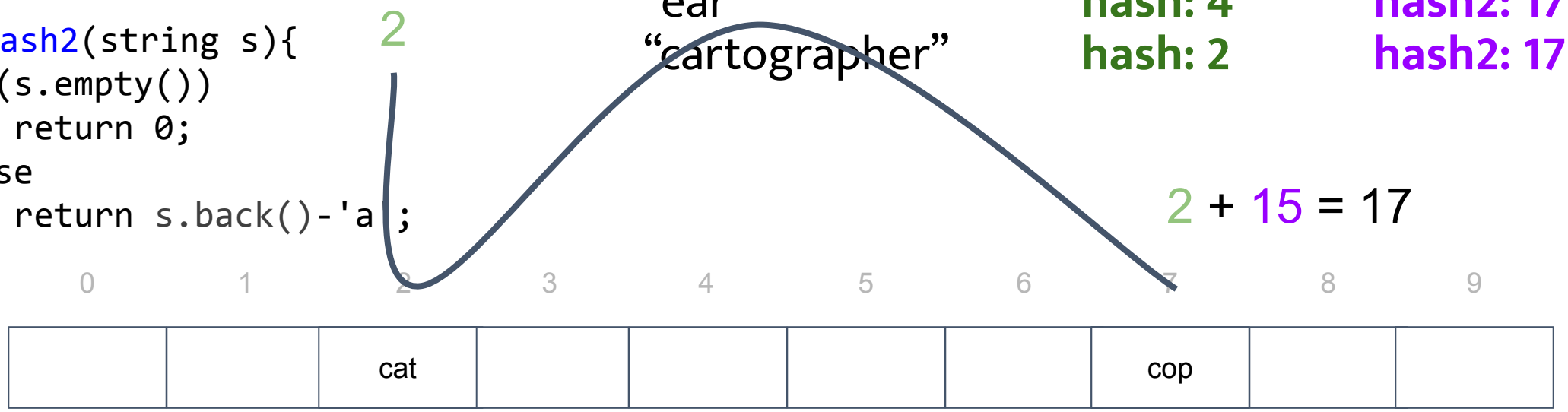
Double Hashing Exercise

number of buckets N=10

```
int hash(string s){
   if(s.empty())
      return 0;
   else
      return s.front()-'a';
}

int hash2(string s){
   if(s.empty())
      return 0;
   else
      return s.back()-'a';
}
```

If we insert these 4 items in this order, using **double hashing** to resolve collisions, where do the keys end up?

"cat"	hash: 2	hash2: 19
"cop"	hash: 2	hash2: 15
"ear"	hash: 4	hash2: 17
"cartographer"	hash: 2	hash2: 17



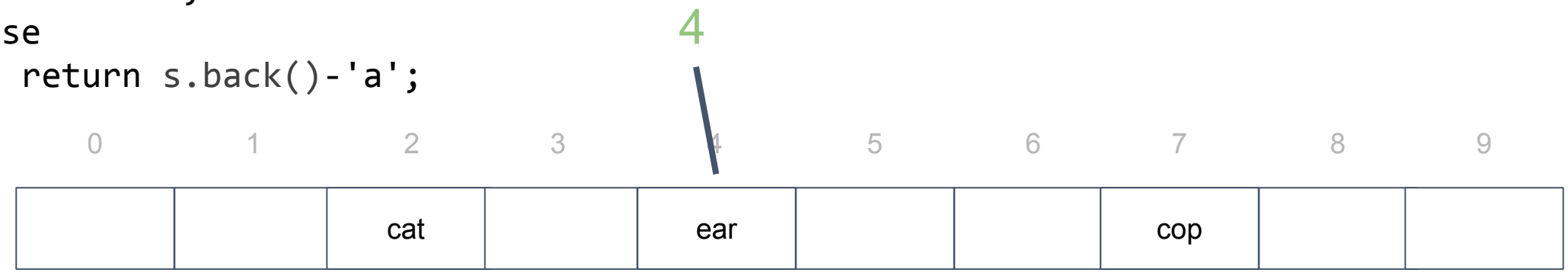
Double Hashing Exercise

number of buckets N=10

```
int hash(string s){
   if(s.empty())
      return 0;
   else
      return s.front()-'a';
}

int hash2(string s){
   if(s.empty())
      return 0;
   else
      return s.back()-'a';
}
```

If we insert these 4 items in this order, using **double hashing** to resolve collisions, where do the keys end up?

“cat”	**hash: 2**	**hash2: 19**
“cop”	**hash: 2**	**hash2: 15**
“ear”	**hash: 4**	**hash2: 17**
“cartographer”	**hash: 2**	**hash2: 17**



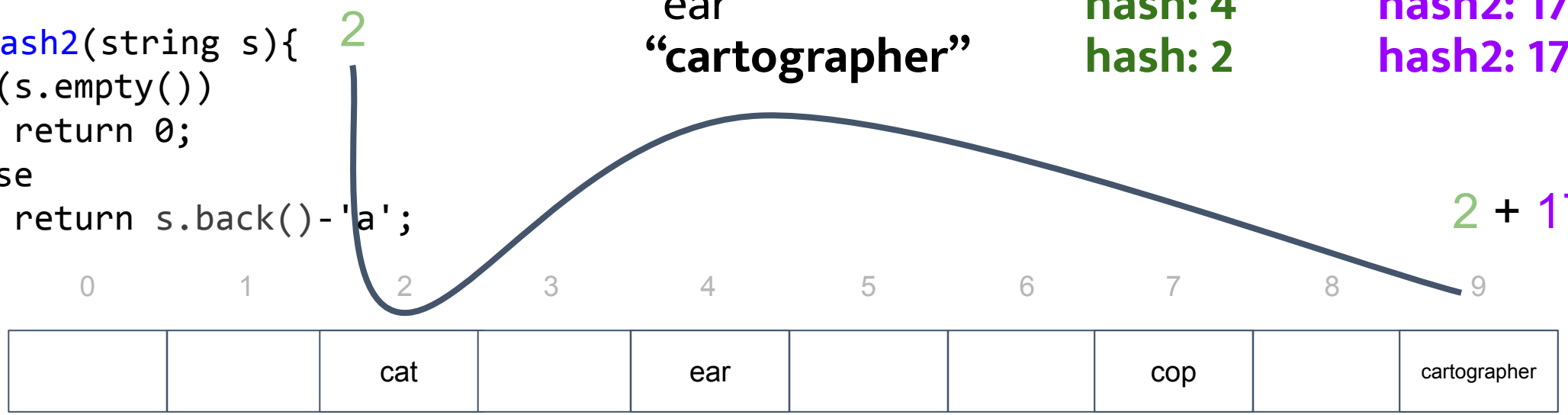
Double Hashing Exercise

number of buckets N=10

```
int hash(string s){
   if(s.empty())
      return 0;
   else
      return s.front()-'a';
}

int hash2(string s){
   if(s.empty())
      return 0;
   else
      return s.back()-'a';
}
```

If we insert these 4 items in this order, using **double hashing** to resolve collisions, where do the keys end up?

"cat"	**hash: 2**	**hash2: 19**
"cop"	**hash: 2**	**hash2: 15**
"ear"	**hash: 4**	**hash2: 17**
"cartographer"	**hash: 2**	**hash2: 17**



Double Hashing: The Second Hash Function

Recall: $(H(k) + i * F(k)) \% N$
What do we want from $F(k)$ to make this work well?

- Nonzero - probing wouldn't ever get anywhere otherwise
- Not equal to N - same problem, since $N \equiv 0 \pmod{N}$
- More generally, coprime to N

How? Easy: Use F' instead, where $F'(k) = 1 + (F(k) \% (N-1))$
(when N is prime)
(the example from the last few slides doesn't do this)

Deleted Elements

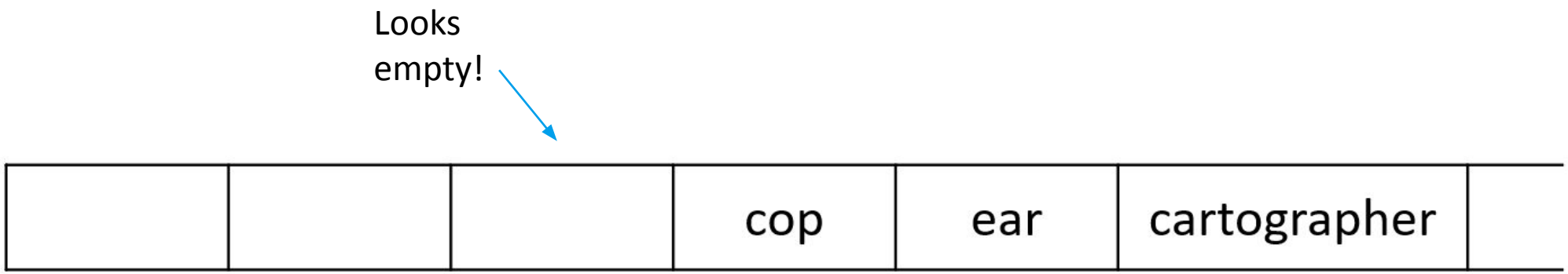
Open Addressing: Erasing Elements

Assume that we inserted “cat” into bucket 2 and then inserted “cop” into bucket 3 after a collision. What if we tried to find “cop” **after *removing*** “cat”?

Open Addressing: Erasing Elements

Assume that we inserted “cat” into bucket 2 and then inserted “cop” into bucket 3 after a collision. What if we tried to find “cop” **after *removing*** “cat”?

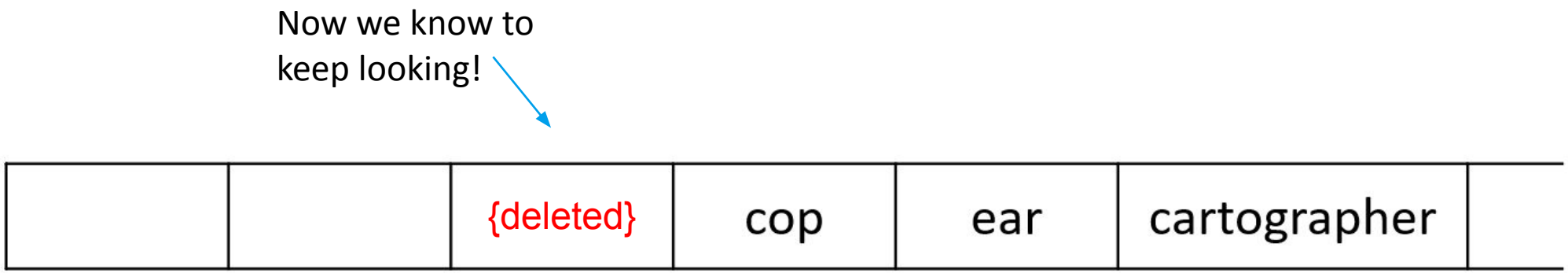
In order to implement probing correctly, we need a deleted element to show that an element used to be here. This way, when we are trying to find/remove items, we know to check further when we run into deleted elements.



Open Addressing: Erasing Elements

Assume that we inserted “cat” into bucket 2 and then inserted “cop” into bucket 3 after a collision. What if we tried to find “cop” **after *removing*** “cat”?

In order to implement probing correctly, we need a deleted element to show that an element used to be here. This way, when we are trying to find/remove items, we know to check further when we run into deleted elements.



Linear Probing Erasing Exercise

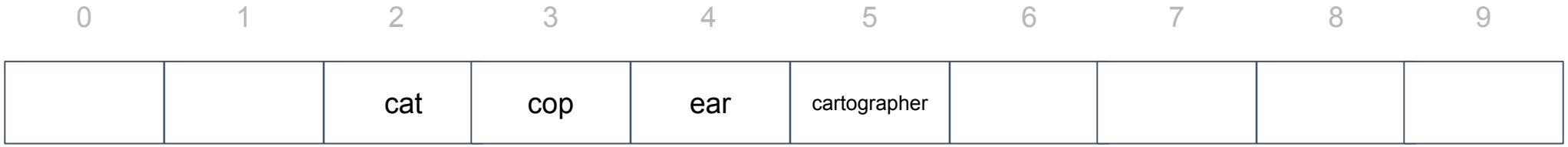
number of buckets N=10

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

How would we complete these three operations in this order, using **linear probing** to resolve collisions?

erase "cat" **hash: 2**
erase "cartographer" **hash: 2**
insert "cartographer" **hash: 2**



Linear Probing Erasing Exercise

number of buckets N=10

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

How would we complete these three operations in this order, using **linear probing** to resolve collisions?

erase "cat" **hash: 2**
erase "cartographer" **hash: 2**
insert "cartographer" **hash: 2**

Linear Probing Erasing Exercise

number of buckets N=10

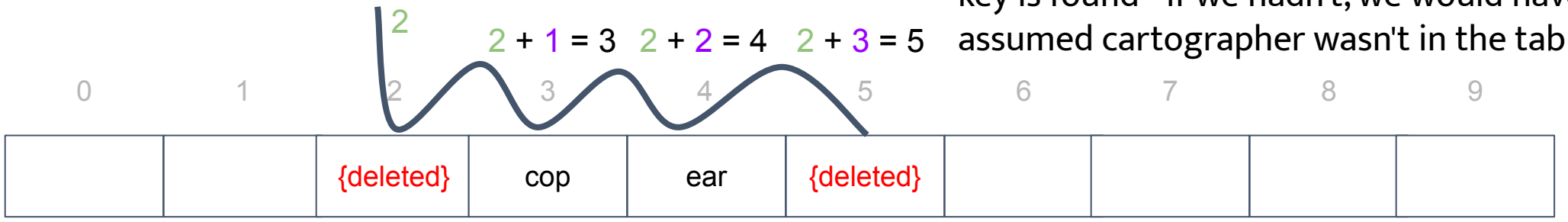
Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

How would we complete these three operations in this order, using **linear probing** to resolve collisions?

erase "cat" **hash: 2**
erase "cartographer" **hash: 2**
insert "cartographer" **hash: 2**

Must continue looking until an empty spot or the key is found - if we hadn't, we would have wrongly assumed cartographer wasn't in the table.



Linear Probing Erasing Exercise

number of buckets N=10

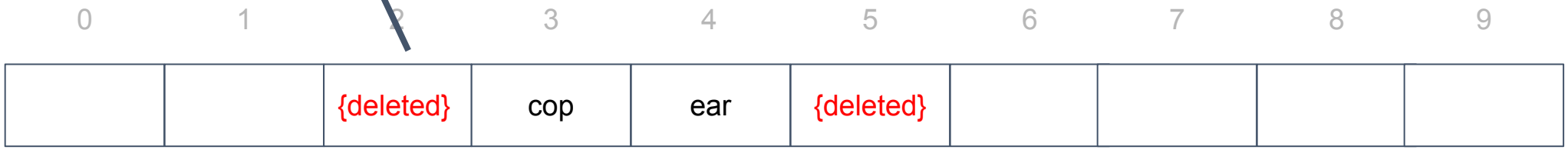
Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

How would we complete these three operations in this order, using **linear probing** to resolve collisions?

erase "cat" **hash: 2**
erase "cartographer" **hash: 2**
insert "cartographer" **hash: 2**

We can't just insert cartographer at the first deleted space! It's possible that cartographer already exists in the map, so we first need to probe until the first empty spot and make sure we don't see it.



Linear Probing Erasing Exercise

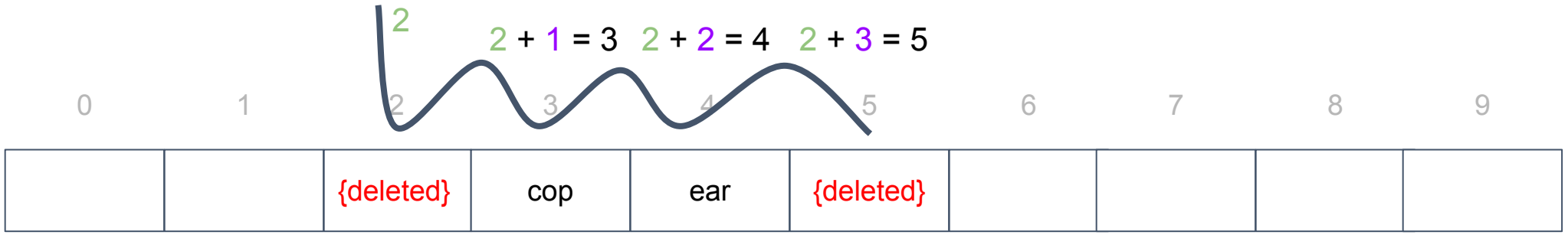
number of buckets N=10

Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

How would we complete these three operations in this order, using **linear probing** to resolve collisions?

erase "cat" **hash: 2**
erase "cartographer" **hash: 2**
insert "cartographer" **hash: 2**



Linear Probing Erasing Exercise

number of buckets N=10

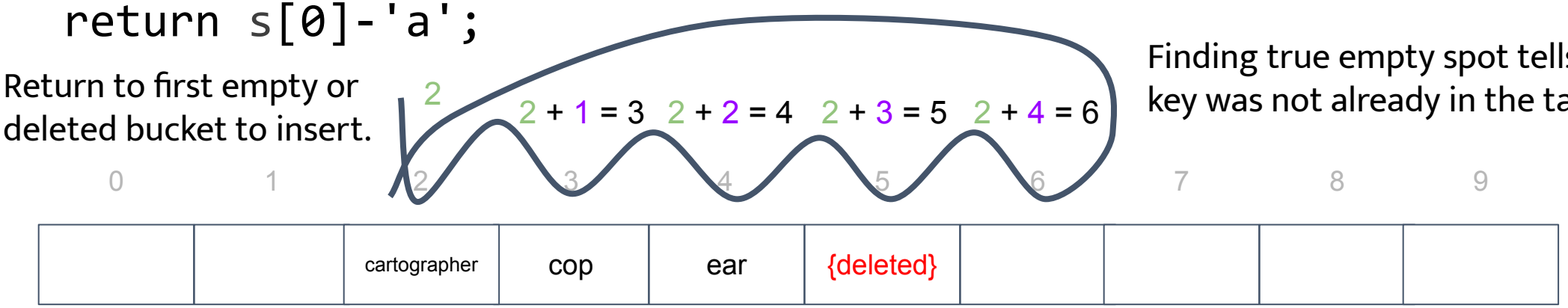
Hash function:

```
int hash2(string s) {
  if(s.empty())
     return 0;
  else
     return s[0]-'a';
}
```

Return to first empty or deleted bucket to insert.

How would we complete these three operations in this order, using **linear probing** to resolve collisions?

erase "cat" **hash: 2**
erase "cartographer" **hash: 2**
insert "cartographer" **hash: 2**



Finding true empty spot tells us the key was not already in the table.

Performance and Load Factor

Hash Table Performance

With Linear Probing, we inserted “cat,” “cop,” “ear,” and “cartographer” and obtained these buckets:

“cat” - 2
“cop” - 3 (collision!)
“ear” - 4
“cartographer” - 5 (double collision!)

What does this show about the performance of our hash table as it gets filled up?

Hash Table Performance

With Linear Probing, we inserted “cat,” “cop,” “ear,” and “cartographer” and obtained these buckets:

“cat” - 2
“cop” - 3 (collision!)
“ear” - 4
“cartographer” - 5 (double collision!)

What does this show about the performance of our hash table as it gets filled up?

It gets worse!

Load Factor and Performance

A key metric for a hash table's performance is its **load factor**

load factor = # elements / # buckets

The higher the load factor, the more likely you are to have a collision!

How to increase performance / reduce collisions:

- Use prime numbers for table sizes - this is because hash functions often give rise to big numbers that are multiples of each other that will often collide if they have a factor in common with the table size (since it decides what the modulus will be)
- Use good hash functions that avoid collisions
- Keep load factor low: < 0.75 (some would argue < .5 or even < .3 - these take more memory)

Hash Table Complexities

k = number of keys, n = table size

Time Complexity

- Search or Insert or Delete: O(1 + k/n)* average, O(k) worst
  - If you **know** that a key **isn't** in the table, you can insert in O(1) time with separate chaining
  - Cannot do the above with STL, since there is no way to tell it that a key definitely isn't in the table
- Amortized if rehashing/resizing is done - similar to vector's amortization

Space Complexity

- O(n) average, O(n) worst (this assumes you keep a good load factor so k < n)

Fundamentally, still O(1) assuming a reasonable load factor (α <= ~0.5)

Comparison of Collision Resolution Methods

Separate Chaining:

- Extra memory is in form of linked list ptrs
- Uses more dynamic memory
- Handling collisions is faster (appending to list is faster than math + reindexing)
- Can have a load factor greater than 1
- Linked lists used due to need for fast insert and delete in middle of list, and no need for random access
- Most common implementation

Open Addressing:

- Extra memory is in form of empty spaces
- Simpler storage
- Requires deleted elements for erase
- Cache locality improves runtime
- Linear probing suffers from clusters
- Load factor must be < 1
    - Quadratic probing needs load factor < .5
- Double hashing wastes time on 2nd hash

Perfect Hashing

If all keys that could be inserted are known ahead of time, you can make a custom hash function that will be guaranteed to have 0 collisions.

Once you have done this, your table will never have any hash collisions!

If the actual number of items that will be inserted into the hash table at a given time is much smaller than the number of possible keys, a lot of memory would be wasted.

TIP: If you find yourself using integers in a fixed range as keys, and that fixed range is relatively small, a vector is usually better. (You can think of the indices as the keys)

Resizing and Rehashing

Resizing Hash Tables

- Reduces load factor
- Make new container, **rehash** every element, and delete old container
- Only needed when load factor is high enough; this maintains amortized O(1) insert

Resizing Hash Tables

- Reduces load factor
- Make new container, **rehash** every element, and delete old container
- Only needed when load factor is high enough; this maintains amortized O(1) insert

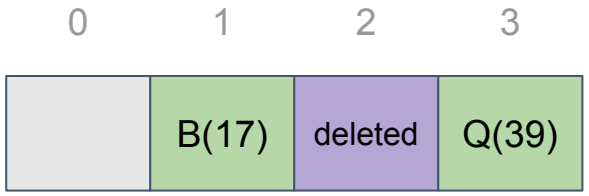
How to rehash:

Create a larger, empty hash table.

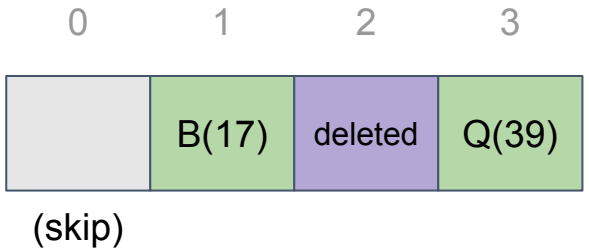
Insert all non-deleted elements.

Some keys will stop colliding in this new, larger table. Some new collisions may also occur.

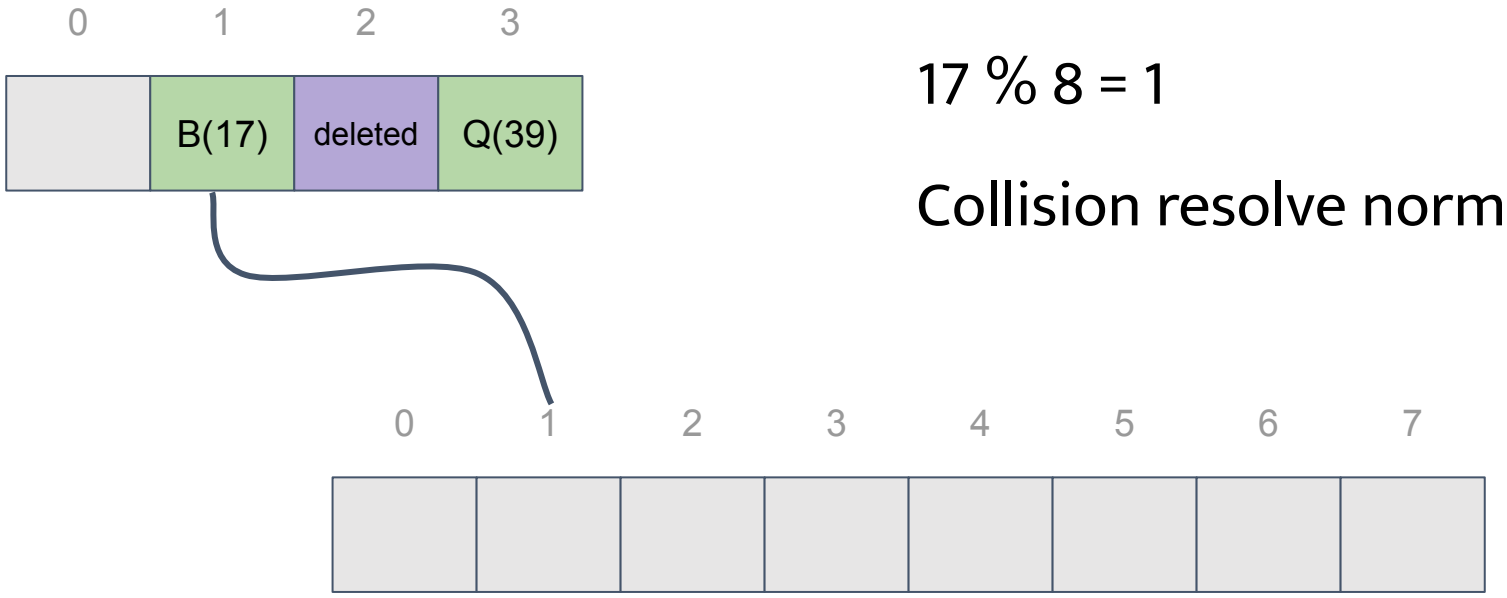
Resizing Hash Tables



Resizing Hash Tables



Resizing Hash Tables

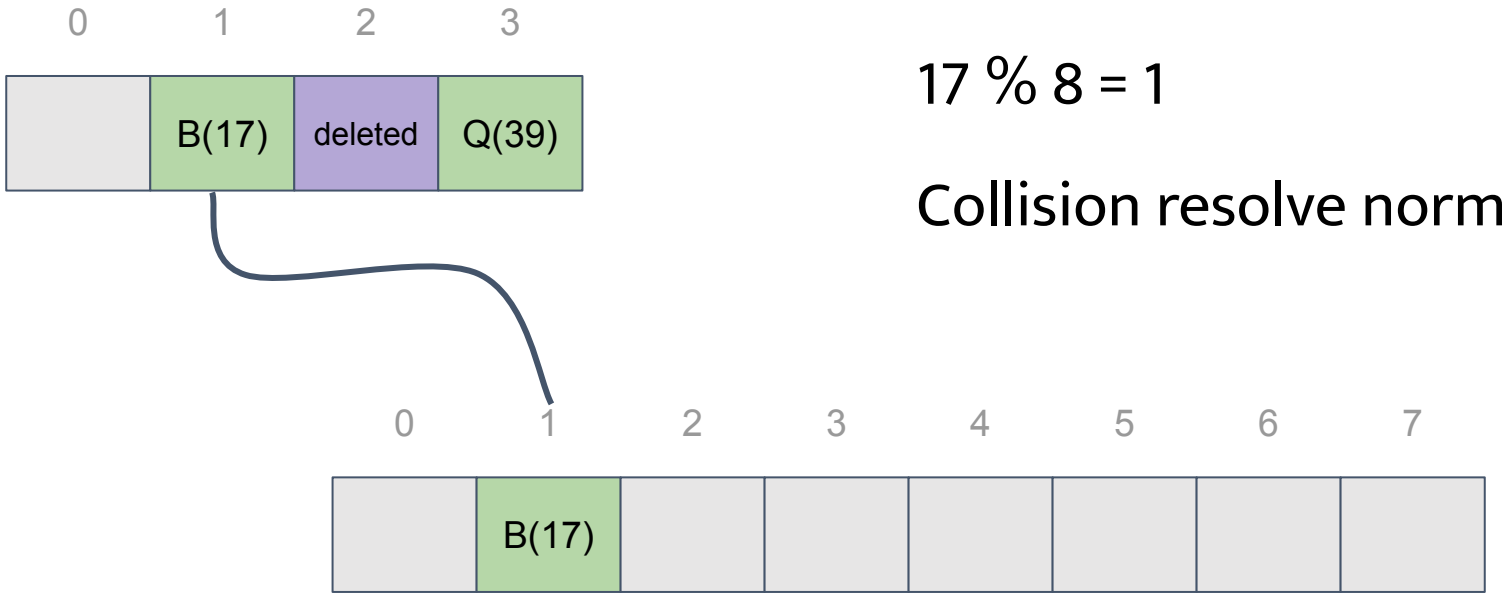


Take the original hash value and mod it by the size of the new hash table.

17 % 8 = 1

Collision resolve normally from there.

Resizing Hash Tables

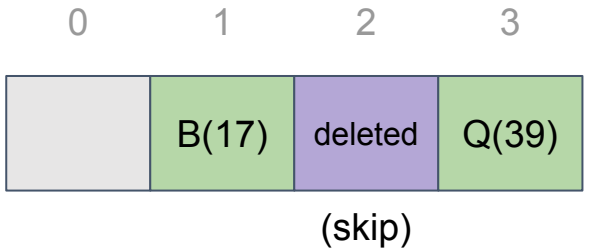


Take the original hash value and mod it by the size of the new hash table.

17 % 8 = 1

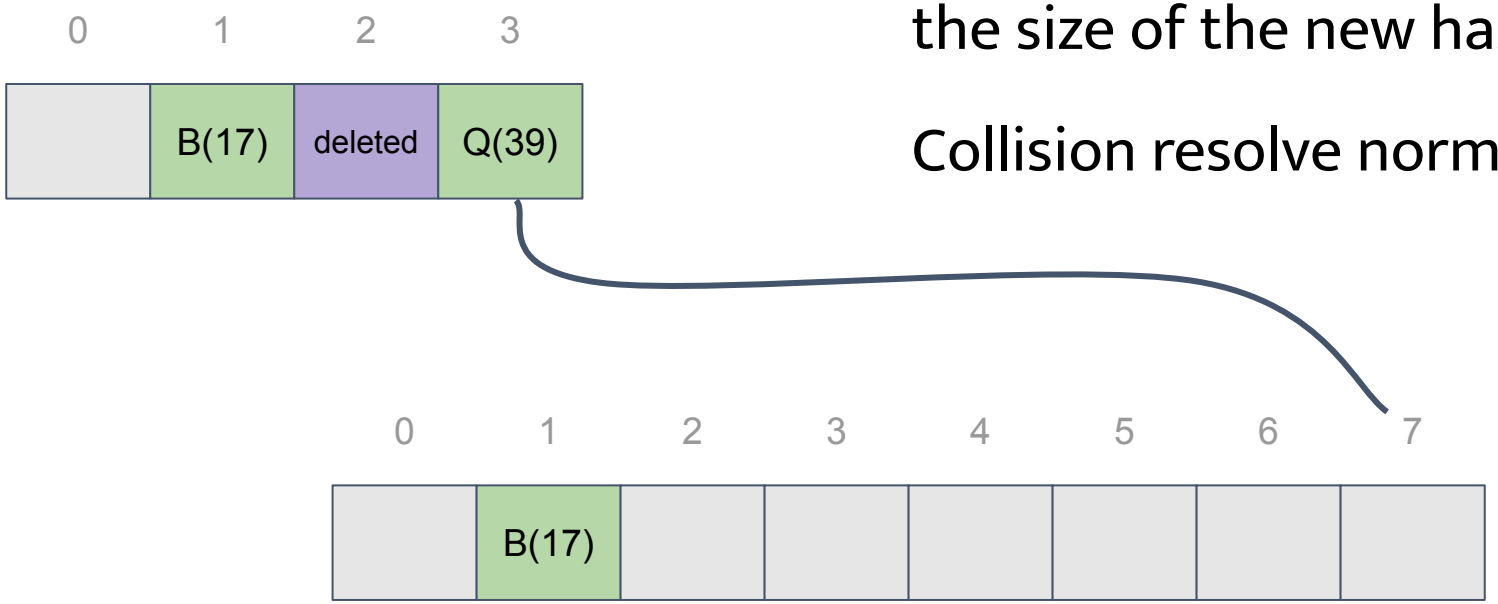
Collision resolve normally from there.

Resizing Hash Tables



Skip any deleted elements! We don't want to insert those into the new hash table, because we're trying to break up contiguous chunks of filled buckets.

Resizing Hash Tables

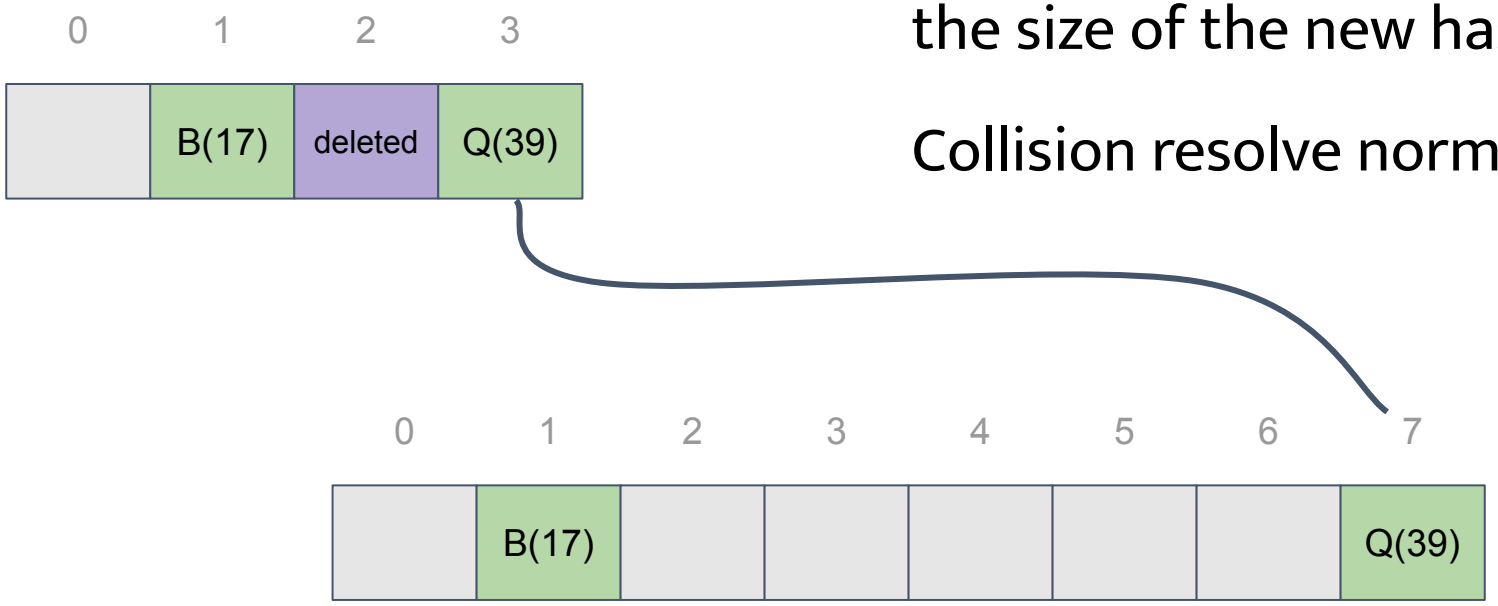


Take the original hash value and mod it by the size of the new hash table: **39 % 8 = 7**

Collision resolve normally from there.

Notice, we did NOT just copy the value at index 3 in the original hash table to index 3 in the new hash table.

Resizing Hash Tables

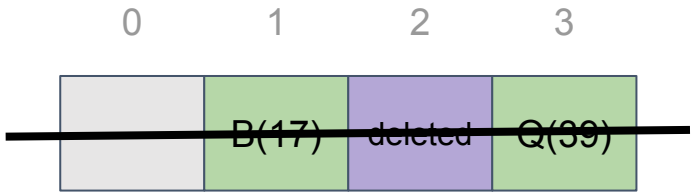


Take the original hash value and mod it by the size of the new hash table: **39 % 8 = 7**

Collision resolve normally from there.

Notice, we did NOT just copy the value at index 3 in the original hash table to index 3 in the new hash table.

Resizing Hash Tables



Then this old array can be destructed.

Hash Tables Summary

Pros

- Good time complexity (O(1) average case) for core operations, many uses
- Great when amount of data is known ahead of time

Cons

- Cost of good hash function can make lookup slow
- Poor spatial locality (similar data is distributed randomly in memory)
- Usually uses an excess of memory
- Unsorted - if you would need to repeatedly sort, use an (ordered) map/set

Hash Tables Practice Problem

Suppose you're given a set of N distinct words, all of the same length M. We want to find a pair of words that are "similar", meaning that you can obtain one from the other by changing one letter. For example, "cart" and "cast" are similar, and "cast" and "cost" are similar (but "cart" and "cost" are not).

We know that N is much, much bigger than M. You can also assume that $M \geq 3$. The algorithm may use any amount of memory. (Hint: Use a hash table.)

Hash Tables Practice Problem

Go through each word and replace each letter with a “blank” letter (*). Search for this “blankified” word in the hash table.

- If you find it, that’s your pair of similar words.
- Otherwise, insert it and keep going! For each i = 0, 1, ..., M-1, and for each string s, set s[i] = ‘*’ and put it into the hash set.

What’s the best possible worst-case running time of this algorithm to find a pair of similar words in the provided set, in terms of length M and number of distinct words N?

(Note that it takes O(M) look up a string of length M in a hash table because it must be hashed and compared for equality).

Hash Tables Practice Problem

Go through each word and replace each letter with a "blank" letter (*). Search for this "blankified" word in the hash table.

- If you find it, that's your pair of similar words.
- Otherwise, insert it and keep going! For each i = 0, 1, ..., M-1, and for each string s, set s[i] = '*' and put it into the hash set.

What's the best possible worst-case running time of this algorithm to find a pair of similar words in the provided set, in terms of length M and number of distinct words N? $\Theta(M^2N)$

For every letter (M) in every word (N), you must do an O(M) look up.

Lab 07 AG

Lab 07 AG

You have to write 4 functions:

- `insert()`
- `operator[]()`
- `erase()`
- `rehash_and_grow()`

The first two will end up being very similar. The last one can call `insert()` if you're careful! It's not super fast, but it's fast enough and will save on lines of code you need to write.

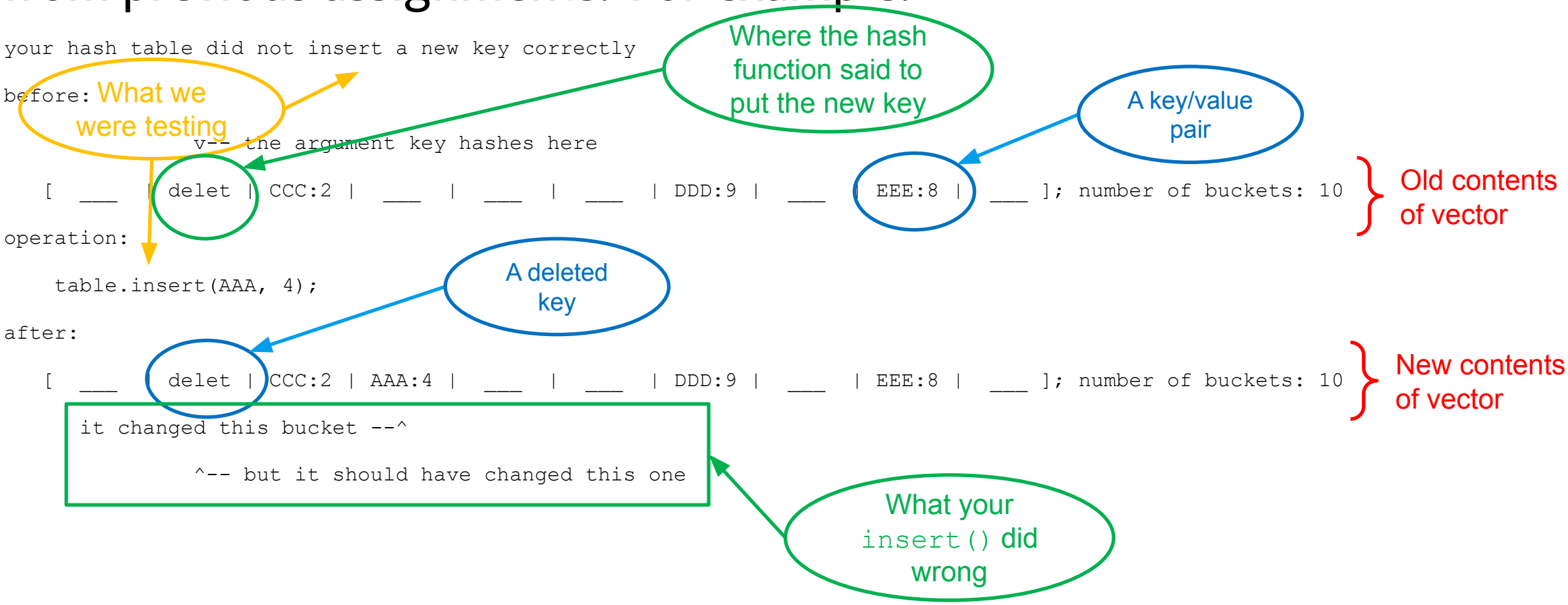
Lab 07 AG Feedback

When you get feedback from the autograder, it'll be very different from previous assignments. For example:

```
your hash table did not insert a new key correctly

before:

               v-- the argument key hashes here

   [  ___  | delet | CCC:2 |  ___  |  ___  |  ___  | DDD:9 |  ___  | EEE:8 |  ___ ]; number of buckets: 10

operation:

    table.insert(AAA, 4);

after:

   [  ___  | delet | CCC:2 | AAA:4 |  ___  |  ___  | DDD:9 |  ___  | EEE:8 |  ___ ]; number of buckets: 10

      it changed this bucket --^

               ^-- but it should have changed this one
```

Lab 07 AG Feedback

When you get feedback from the autograder, it'll be very different from previous assignments. For example:



Lab 07 Verifier

Local debugging can be tricky. There's a tool to help with that :)

https://github.com/khuldraeseth/eecs281-lab07-verifier

Documentation is there in the README, along with example usage.

You will still have to write your own tests! This is only a framework to make that easier.

Handwritten Problem

Handwritten Problem

Prefixes are words that can be followed by some other letters to form a longer word - let's call this final word the successor. For example, the prefix “an” followed by “other” forms the word “another”.

Now, given a dictionary consisting of many prefixes and a sentence, you need to replace all the successors in the sentence with the prefix forming it. If a successor has many prefixes that can form it, replace it with the prefix with the shortest length.

The input will only have lower-case letters. Return the new sentence in a vector of strings.
P prefixes**, N** words**, M** length**: O(PM + NM^2) (Hashing/looking up a string of length M costs O(M))**

Example:
Prefixes: ["cat", "bat", "rat"]
Sentence:[“the”, “cattle”, “was”, “rattled”, “by”, “the”, “battery”]
Output: ["the”, “cat”, “was”, “rat”, “by”, “the”, “bat"]

```
replace_words(const vector<string>& prefixes,
                              const vector<string>& sentence);
```