CSCI 241

Lecture 18

HashMap, Rehashing, Hash Functions, Open Addressing

Announcements

- Midterm grading is underway
- Lab 7 is forthcoming (out today or tomorrow)

Goals

- Know how to implement Set and Map using hash tables.
- Know how to respond to large hash table load factors by resizing the array and **rehashing.**
- Know how to avoid linked list buckets using **open addressing** with **linear** or **quadratic probing**.
- Know how to use the hashCode method of java objects.

Origins of the term "hash"

History [edit]

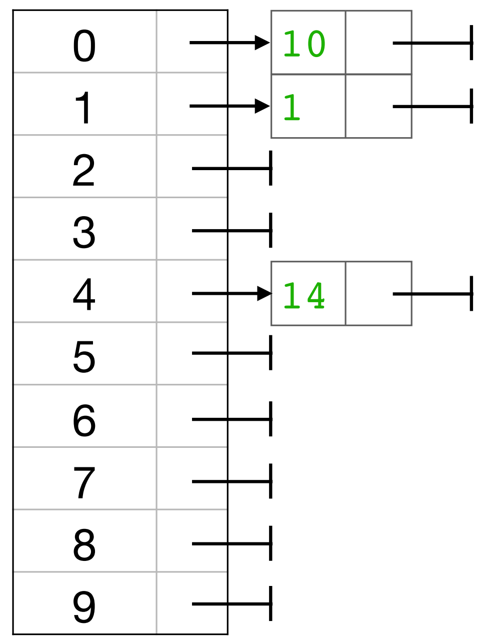
The term "hash" offers a natural analogy with its non-technical meaning (to "chop" or "make a mess" out of something), given how hash functions scramble their input data to derive their output.[19] In his research for the precise origin of the term, Donald Knuth notes that, while Hans Peter Luhn of IBM appears to have been the first to use the concept of a hash function in a memo dated January 1953, the term itself would only appear in published literature in the late 1960s, on Herbert Hellerman's *Digital Computer System Principles*, even though it was already widespread jargon by then.[20]

https://en.wikipedia.org/wiki/Hash_function#History

Implementing Set<V>

$h(k) = k \% A.length$

- Use a HashTable!
- Hash the key to determine array index
- Store values in array
    - add(14): (14 % 10) => 4
    - add(10): (10 % 10) => 0
    - add(1): (1 % 10) => 1
    - **add(11): (11 % 10) => 1**

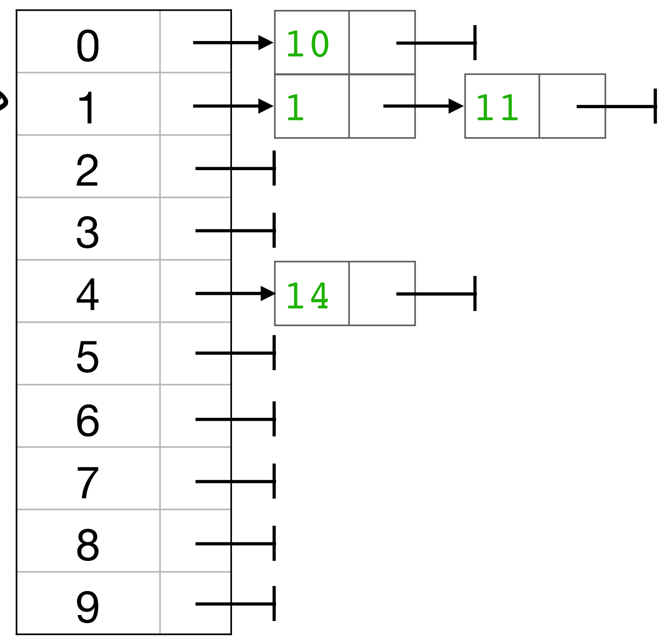


Implementing Set<V>

h(k) = k % A.length

- Use a HashTable!
- Hash the key to determine array index
- Store values in array
  - add(14): (14 % 10) => 4
  - add(10): (10 % 10) => 0
  - add(1): (1 % 10) => 1
  - **add(11): (11 % 10) => 1**

(collision)

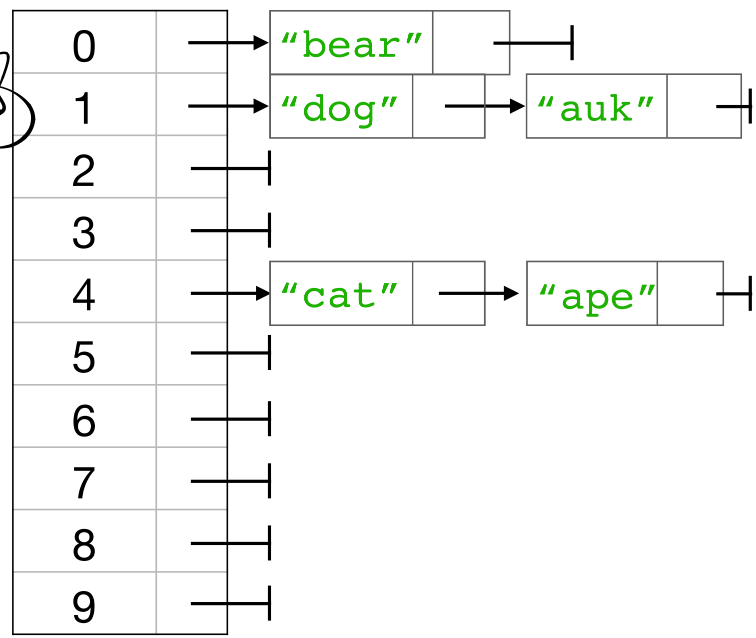


The Map Interface

```
public interface Map<K,V> {
    /** Returns the value to which the specified key
      * is mapped, or null if this map contains no
      * mapping for the key. */
    V get(Object key);

    /** Associates the specified value with the
      * specified key in this map */
    V put(K key, V value);

    /** Removes the mapping for a key from this map
      * if it is present */
    V remove(Object key);

    // more methods
}
```

Map<Integer,String>

- Use a HashTable!
- Hash the key to determine array index
- Store values in array

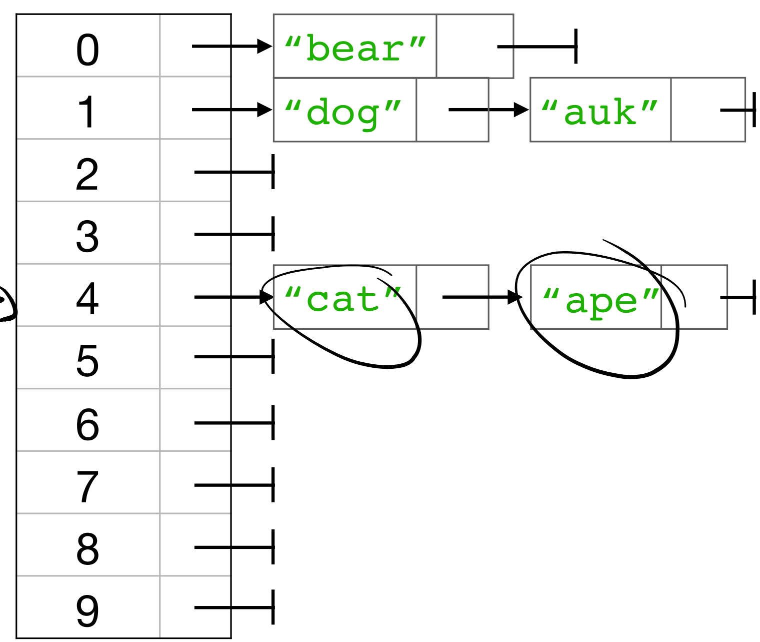


h(k) = k % A.length

```
put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);
```

Map<Integer,String>

- Use a HashTable!
- Hash the key to determine array index
- Store values in array



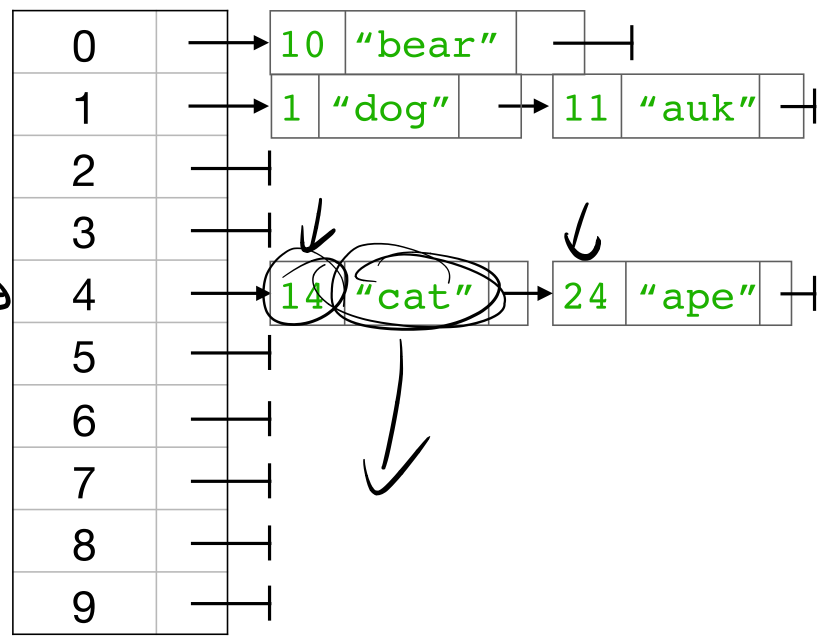
h(k) = k % A.length

put(1, "dog");
put(11, "auk");
put(10, "bear");
put(14, "cat");
put(24, "ape");

get(14)

Map<Integer,String>

- Use a HashTable (or a HashSet of Key-Value pairs)
- Hash the key to determine array index
- ~~Store values in array~~
- Store (K,V) pairs in the array.



put(1, "dog");
put(11, "auk");
put(10, "bear");
put(14, "cat");
put(24, "ape");

get(14)

Hash Tables: Load Factor

$$\frac{\text{\# entries in table}}{\text{size of the array}}$$

Hash Tables: Load Factor

How full is your hash table?

$$\text{Load factor } \lambda = \frac{\text{\# entries in table}}{\text{size of the array}}$$

The average bucket size is λ.

Average-case runtime is O(λ).

Hash Tables: Load Factor

$$\frac{\text{\# entries in table}}{\text{size of the array}}$$

Hash Tables: Load Factor

$$\text{Load factor } \lambda = \frac{\text{\# entries in table}}{\text{size of the array}}$$

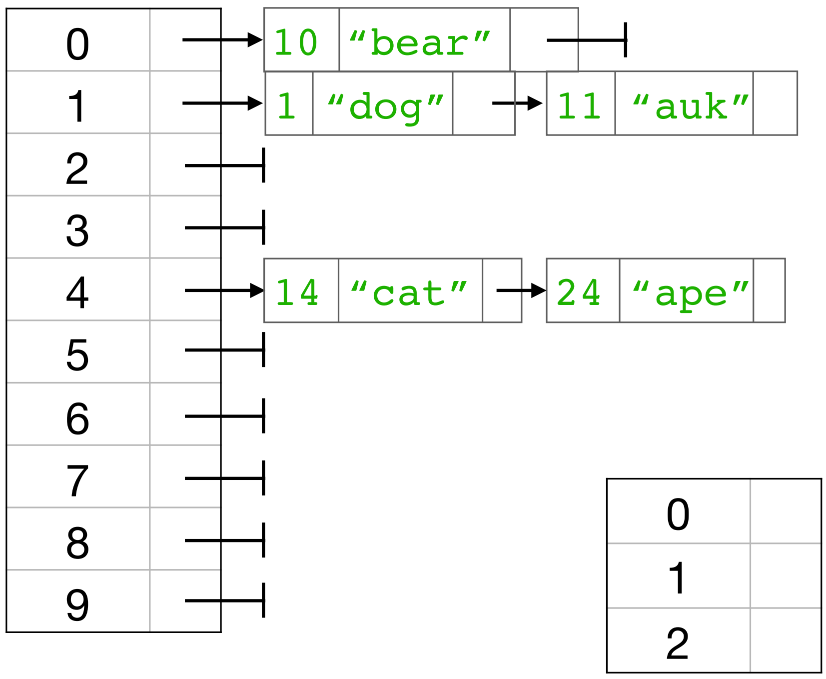
Average-case runtime is $O(\lambda)$.

- If λ is large, runtime is slow.
- If λ is small, memory is wasted.

Strategy: grow or shrink array when λ gets too large or small.

Shrinking the array

Requires **rehashing**: put each element where in belongs in the new array.

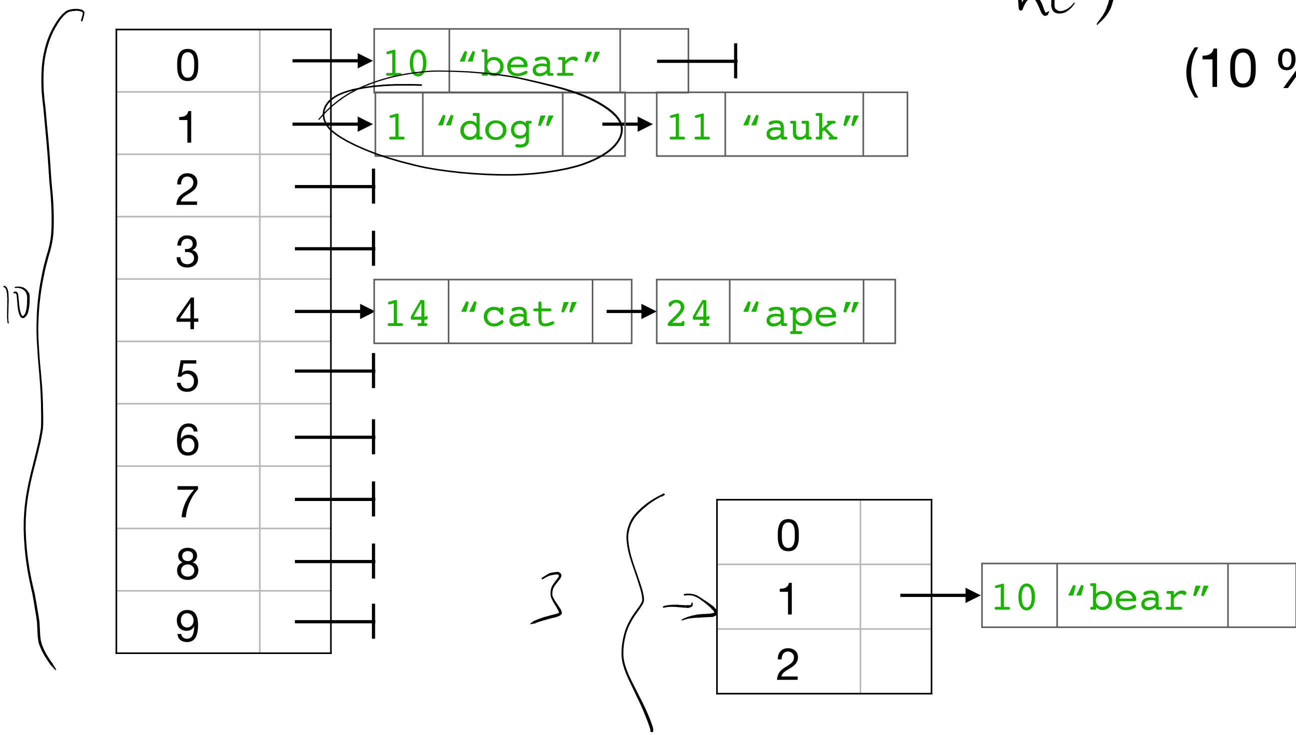


Shrinking the array

Requires **rehashing**: put each element where in belongs in the new array.

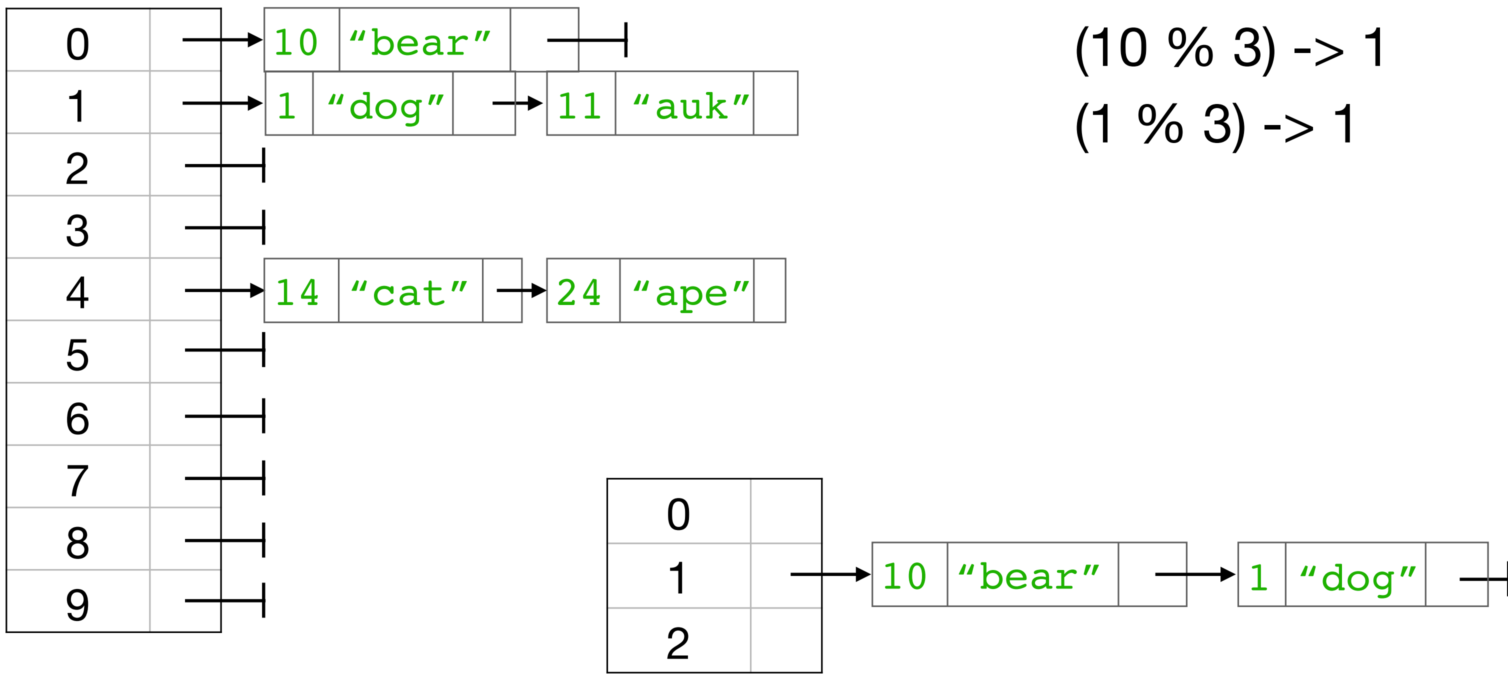
$h(x) = x \,\%\, \cancel{10}\ 3$

(10 % 3) -> 1



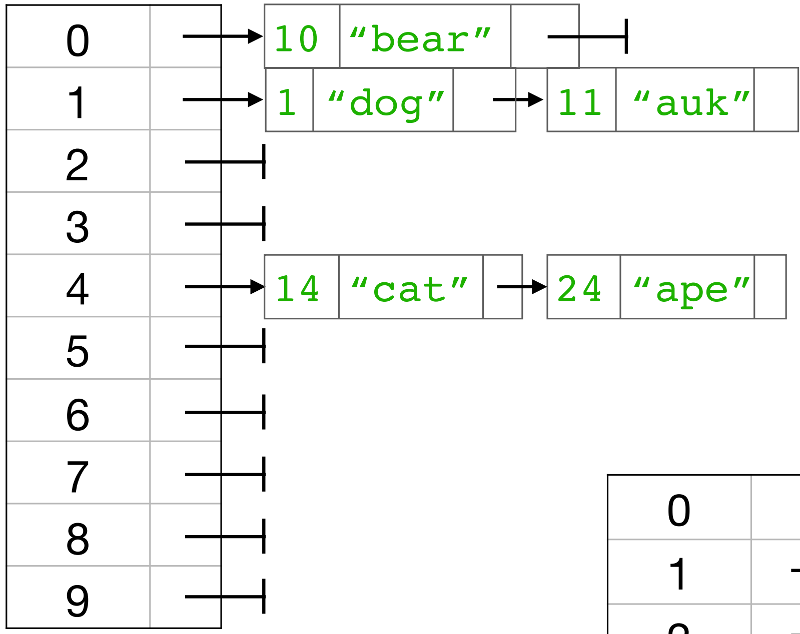
Shrinking the array

Requires **rehashing**: put each element where in belongs in the new array.



Shrinking the array

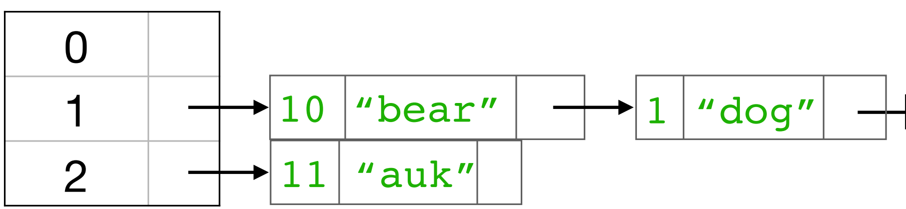
Requires **rehashing**: put each element where in belongs in the new array.



(10 % 3) -> 1

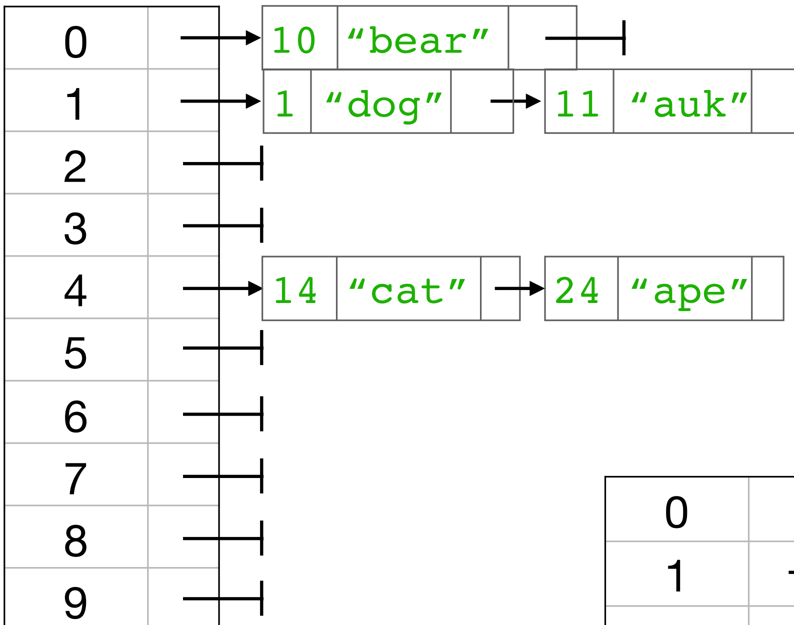
(1 % 3) -> 1

(11 % 3) -> 2



Shrinking the array

Requires **rehashing**: put each element where in belongs in the new array.

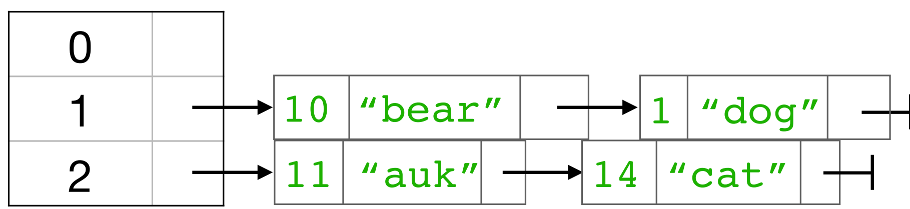


(10 % 3) -> 1

(1 % 3) -> 1

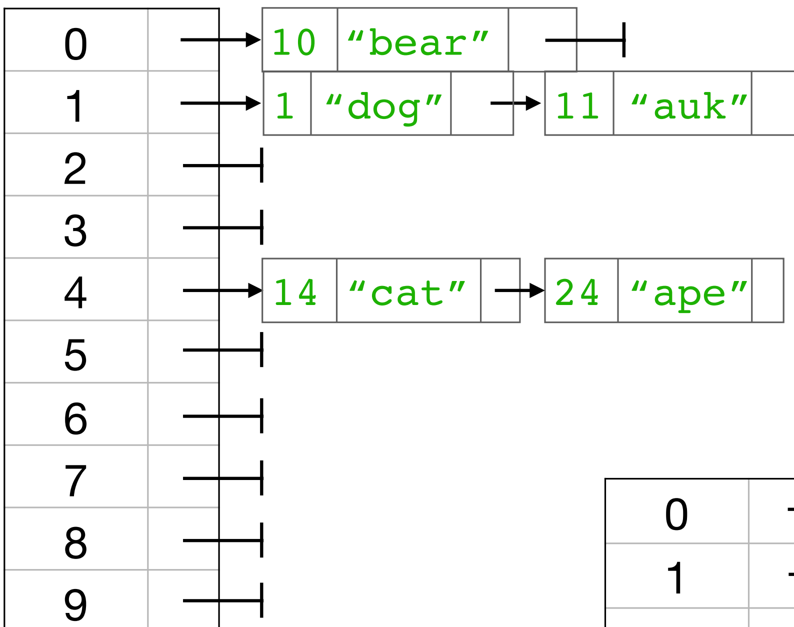
(11 % 3) -> 2

(14 % 3) -> 2



Shrinking the array

Requires **rehashing**: put each element where in belongs in the new array.



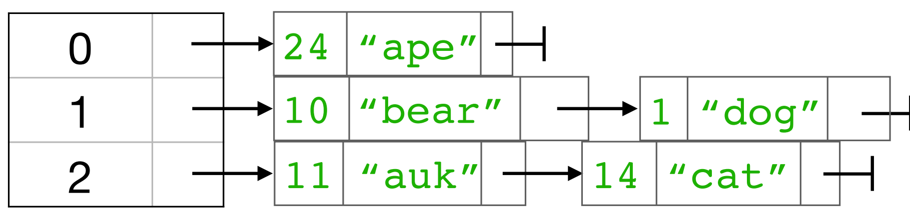
(10 % 3) -> 1

(1 % 3) -> 1

(11 % 3) -> 2

(14 % 3) -> 2

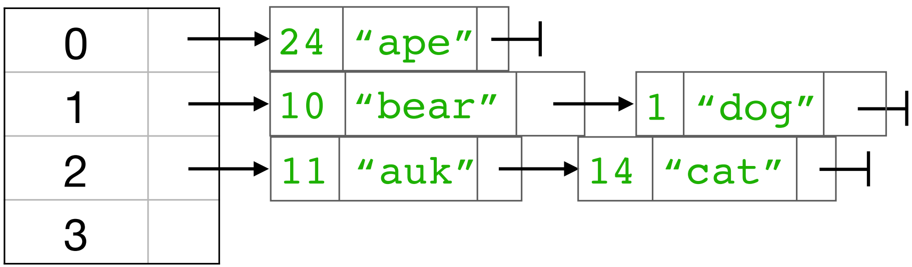
(24 % 3) -> 0

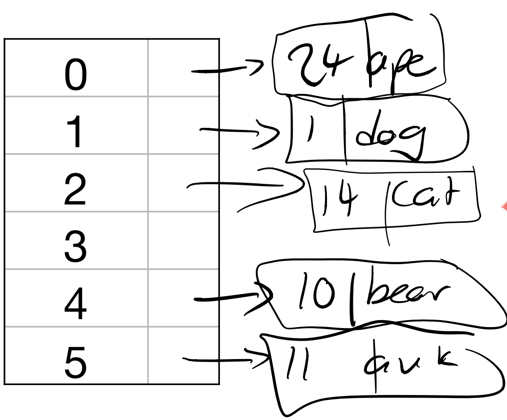


Growing the array

Also requires **rehashing**: put each element where in belongs in the new array.

Exercise: **Grow the array to size 6 and rehash:**





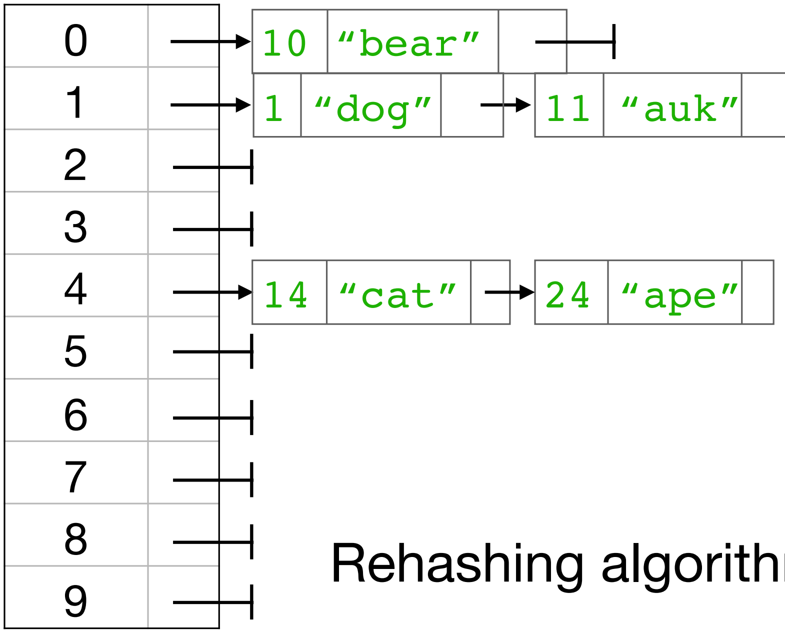
How many elements are in the most full bucket?

a. 1
b. 2
c. 3
d. 4

Rehashing: Runtime

Let N = array size

Let n = number of entries



$N + n(\text{runtime of put})$

Rehashing algorithm:

visits N buckets

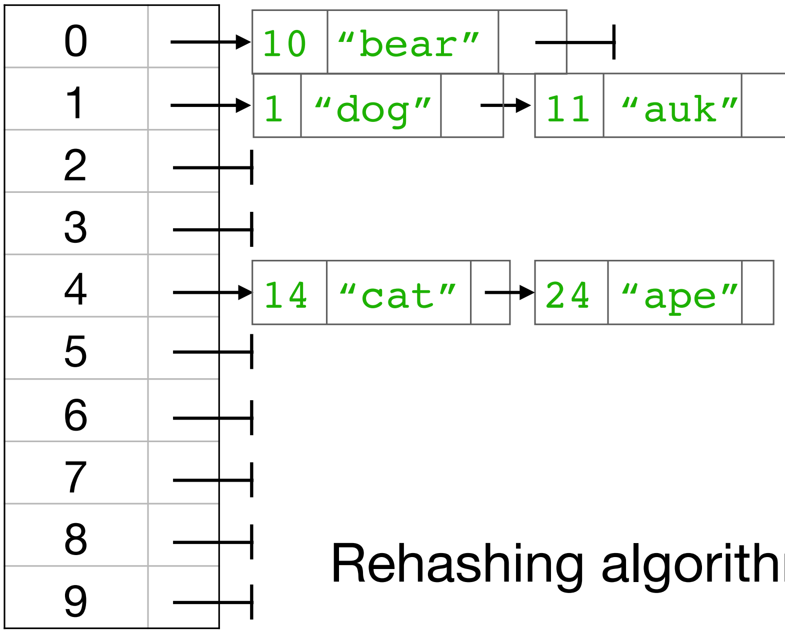
visits n entries (total)

could be O(n) =(

```
for each bucket b:
    for each element e in b:
+n      put e into the new array
```

n times

Rehashing: Runtime, take 1



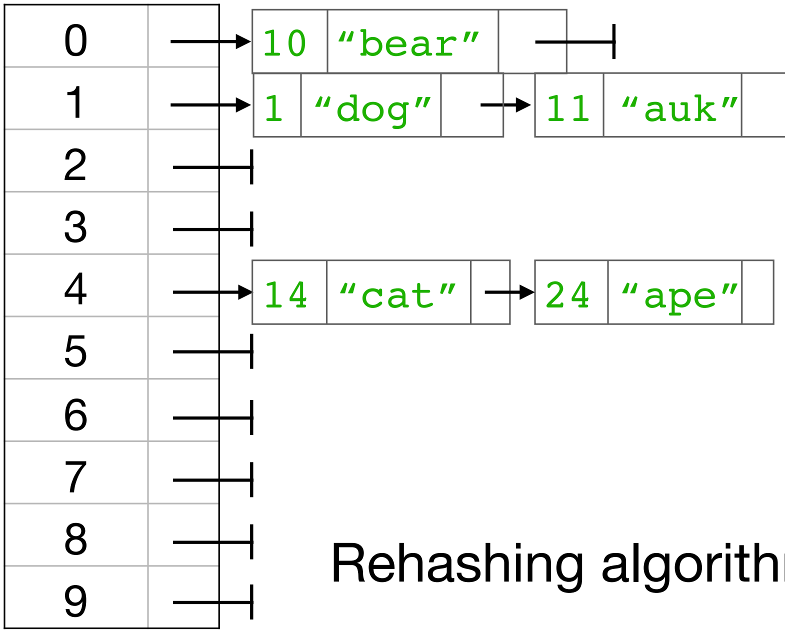
Let C = array size

Let n = number of entries

Rehashing algorithm:

```
for each bucket b:
  for each element e in b:
    put e into the new array
```

Rehashing: Runtime, take 1



Let C = array size

Let n = number of entries

visits C buckets

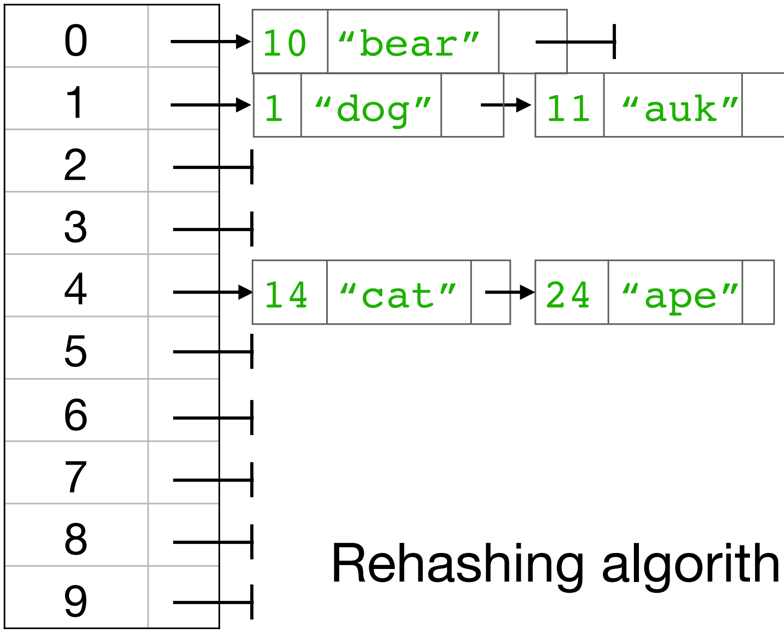
Rehashing algorithm:

```
for each bucket b:
   for each element e in b:
      put e into the new array
```

Rehashing: Runtime, take 1

Let C = array size

Let n = number of entries



visits `c` buckets

Rehashing algorithm:

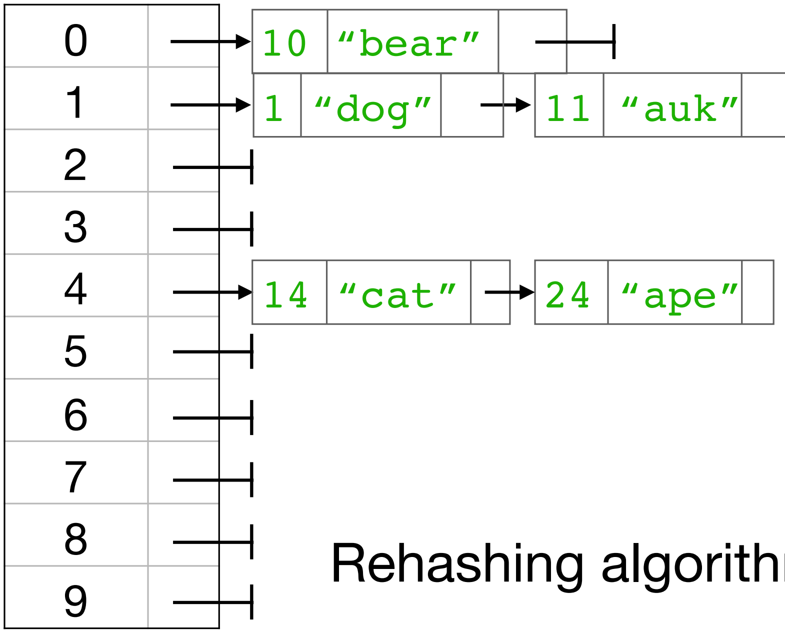
visits `n` entries (total)

```
for each bucket b:
  for each element e in b:
    put e into the new array
```

Rehashing: Runtime, take 1

Let C = array size

Let n = number of entries



visits C buckets

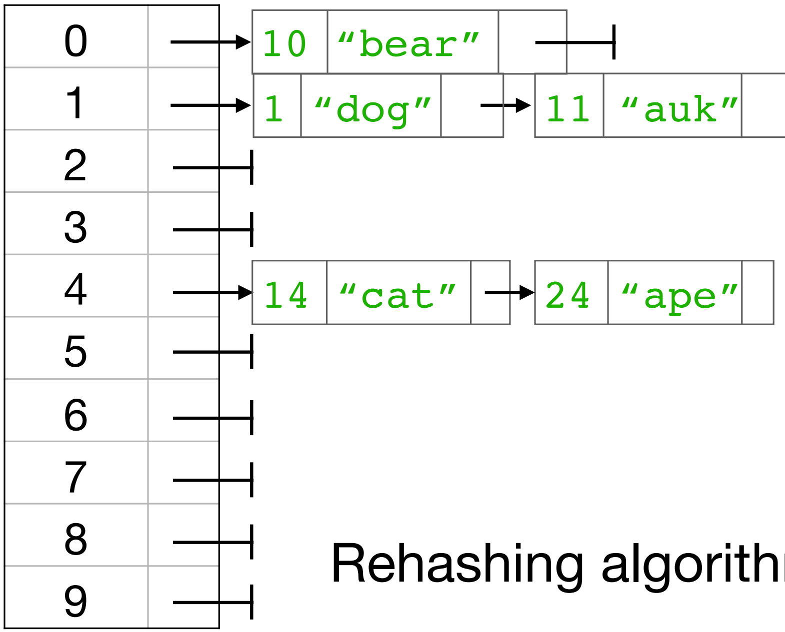
visits n entries (total)

could be O(n) =(

Rehashing algorithm:

```
for each bucket b:
  for each element e in b:
    put e into the new array
```

Rehashing: Runtime, take 1



Let C = array size

Let n = number of entries

Overall runtime is:

- worst-case $O(C + n^2)$
- average-case $O(C + n)$

visits C buckets

Rehashing algorithm: visits n entries (total)

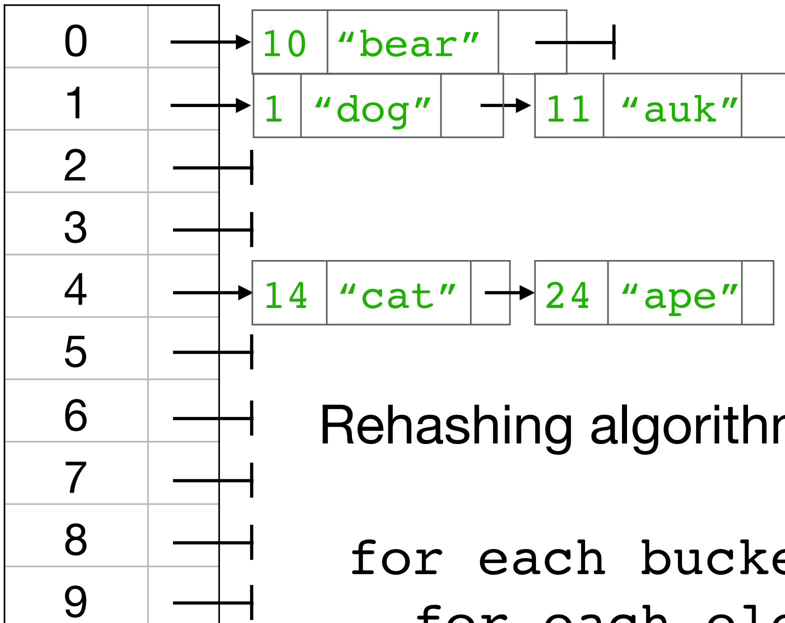
could be O(n) =(

```
for each bucket b:
  for each element e in b:
    put e into the new array
```

Rehashing: Runtime, take 2

Let C = array size (capacity)

Let n = number of entries



visits C buckets

Rehashing algorithm:

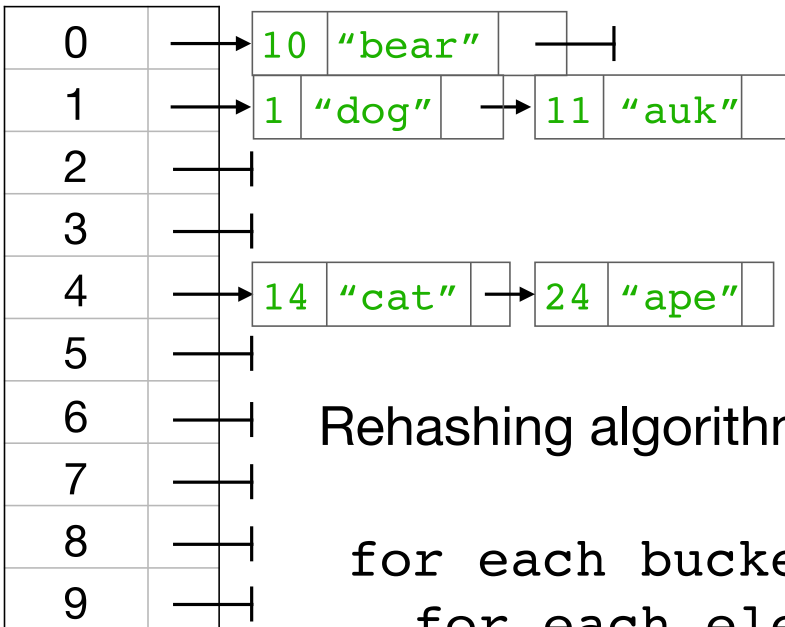
visits n entries (total)

```
for each bucket b:
   for each element e in b:
      put e into the new array
```

Rehashing: Runtime, take 2

Let C = array size (capacity)

Let n = number of entries



Rehashing algorithm:

visits C buckets

visits n entries (total)

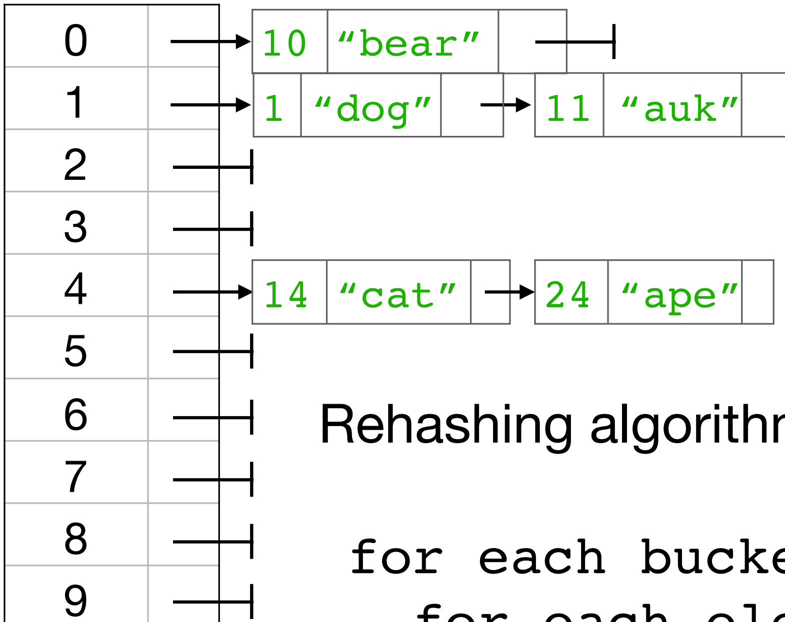
could it be O(n)?

```
for each bucket b:
    for each element e in b:
        put e into the new array
```

Rehashing: Runtime, take 2

Let C = array size (capacity)

Let n = number of entries



visits C buckets

Rehashing algorithm:

visits n entries (total)

could it be O(n)?

```
for each bucket b:
   for each element e in b:
      put e into the new array
```

We **can't** have duplicate keys: all (k,v) pairs were already in the map!
Consequence: we don't need to search the bucket when rehashing

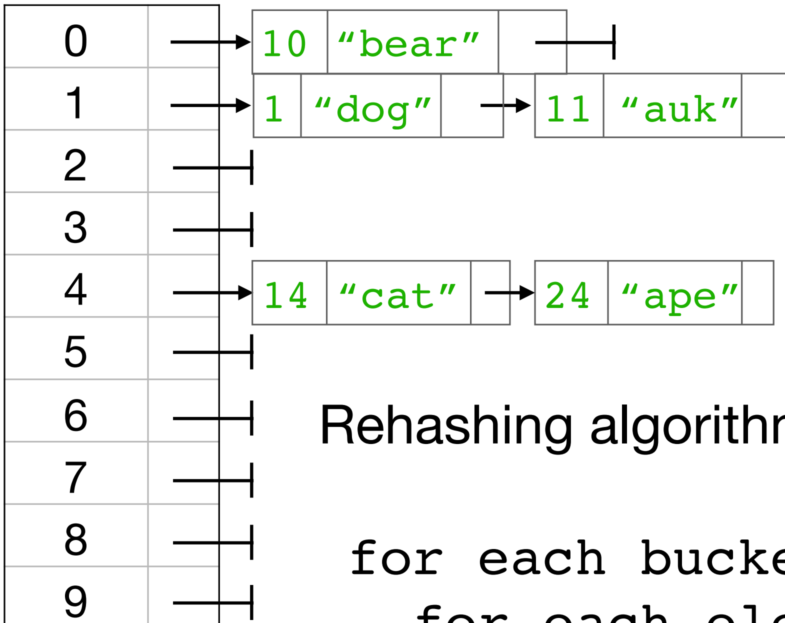
Rehashing: Runtime, take 2

Let C = array size (capacity)

Let n = number of entries

Overall runtime is:

- worst-case O(C + n)



Rehashing algorithm:

visits C buckets

visits n entries (total)

could it be O(n)?

```
for each bucket b:
  for each element e in b:
    put e into the new array
```

We **can’t** have duplicate keys: all (k,v) pairs were already in the map!
Consequence: we don’t need to search the bucket when rehashing

How do we hash things that aren't integers?

Hashing Multiple Integers

- Various heuristic methods:
  - (a + b + c + d) % N
  - (ak^1 + bk^2 + ck^3 + dk^4) % N

Hashing Strings

- Interpret ASCII (or unicode) representation as an integer.
- Java String uses:
  `s[0]*31^(n-1) + s[1]*31^(n-2)+ … +s[n-1]`

Hashing in Java

- Object has a hashCode method.

  By default, this returns the object's address in memory.

- **Scenario 1: You are using a class** that someone else wrote.
  - All Java objects (i.e., non-primitive types) inherit from Object.
  - If you want to put an instance of the class in a hash table, you don't need to know how to hash it!
  - Just call its `hashCode` method.

Collision Resolution

- **Chaining** - use a LinkedList to store multiple elements per bucket.
- **Open Addressing** - use empty buckets to store things that belong in other buckets.
  - Need some scheme for deciding which buckets to look in.

Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.
- Which empty bucket? Using the next empty one is called **Linear Probing**

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	
1	
2	
3	
4	

```
put(key):
   h = hash(key);
   while A[h] is full:
      h = (h+1) % N
   A[h] = value
```

Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	
1	(1, dog)
2	
3	
4	

```
put(key):
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```

Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	
1	(1, dog)
2	(11, auk)
3	
4	

```
put(key):
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```

Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	(10, bear)
1	(1, dog)
2	(11, auk)
3	
4	

```
put(key):
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```

Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	(10, bear)
1	(1, dog)
2	(11, auk)
3	
4	(14, cat)

```
put(key):
   h = hash(key);
   while A[h] is full:
      h = (h+1) % N
   A[h] = value
```

Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	(10, bear)
1	(1, dog)
2	(11, auk)
3	(24, ape)
4	(14, cat)

```
put(key):
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```

Open Addressing with Linear Probing

- Problem with linear probing:
  - Hashing clustered values (e.g., 1, 1, 3, 2, 3, 4, 6, 4, 5) will result in a lot of searching.

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	(10, bear)
1	(1, dog)
2	(11, auk)
3	(24, ape)
4	(14, cat)

```
put(key):
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```

Open Addressing with Quadratic Probing

- **Quadratic Probing:** Jump further ahead to avoid clustering of full buckets.

Linear probing looks at H, H+1, H+2, H+3, H+4, …

Quadratic probing looks at H, H+1, H+4, H+9, H+16, …

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	(10, bear)
1	(1, dog)
2	(11, auk)
3	(24, ape)
4	(14, cat)

```
put(key):
  H = hash(key);
  i = 0;
  while A[h] is full:
    h = (H + i²) % N
    i++;
  A[h] = value
```

Open Addressing with Quadratic Probing

- **Quadratic Probing:** Jump further ahead to avoid clustering of full buckets.

Linear probing looks at H, H+1, H+2, H+3, H+4, …

Quadratic probing looks at H, H+1, H+4, H+9, H+16, …

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	(10, bear)
1	(1, dog)
2	(11, auk)
3	(24, ape)
4	(14, cat)

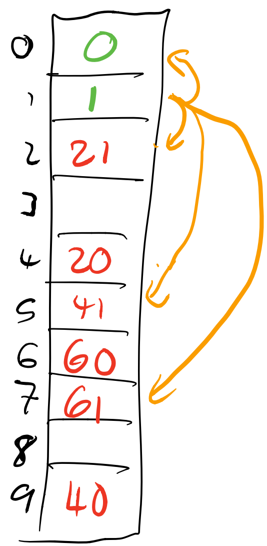
```
put(key):
   H = hash(key);
   i = 0;
   while A[h] is full:
      h = (H + i²) % N
      i++;
   A[h] = value
```

Open Addressing with Quadratic Probing

- **Quadratic Probing:** Jump further ahead to avoid clustering of full buckets.

Exercise: Which buckets are full after the following insertions into an array size of 10 using quadratic probing?

put(0, “ape”);
put(1, “dog”);
put(20, “elf”);
put(21, “auk”);
put(40, “bear”);
put(41, “cat”);
put(60, “elk”);
put(61, “imp”);




```
put(key):
    H = hash(key);
    i = 0;
    while A[h] is full:
        h = (H + i^2) % N
        i++;
    A[h] = value
```

Open Addressing with Quadratic Probing

- **Quadratic Probing:** Jump further ahead to avoid clustering of full buckets.

Exercise: Which buckets are full after the following insertions into an array size of 10 using quadratic probing?

put(0, “ape”);	0
put(1, “dog”);	1
put(20, “elf”);	0, 1, 4
put(21, “auk”);	1, 2
put(40, “bear”);	0, 1, 4, 9
put(41, “cat”);	1, 2, 5
put(60, “elk”);	0, 1, 4, 9, 6
put(61, “imp”);	1, 2, 5, 10, 7

```
put(key):
   H = hash(key);
   i = 0;
   while A[h] is full:
      h = (H + i²) % N
      i++;
   A[h] = value
```

Open Addressing: Runtime

- May be faster, but may not be. Depends on keys.

- There’s no free lunch: worst-case is always O(n).

- In practice, average-case is O(1) if you make good design decisions and insertions are not done by an adversary.

Further Reading

- CLRS 11.5: Perfect Hashing
  - You can guarantee O(1) lookups and insertions if the set of keys is fixed
- C++ implementations from Google:
  - sparse_hash_map - optimized for memory overhead
  - dense_hash_map - optimized for speed

Map and HashMap

- Map is an ADT
- HashMap is an implementation of a Map using a Hash Table.
- TreeMap is a thing too - some of you already wrote one!
  - AVL tree: store a key and a value in each node; BST property applies to **keys** only
  - Example: TreeMap<String, Integer> maps words to the number of times they have been seen

TreeMap vs HashMap

- Runtime of put, get, and remove:
  - TreeMap has O(log n) worst and expected
  - HashMap has O(1) expected, O(n) worst; better in practice
- Other considerations:
  - TreeMaps enable sorted traversal of keys
  - HashMaps are space-inefficient if load factor is small