CSE 373: Data Structures and Algorithms

Hash Tables: Handling Collisions

Autumn 2018

Shrirang (Shri) Mare
shri@cs.washington.edu

Thanks to Kasey Champion, Ben Jones, Adam Blank, Michael Lee, Evan McCarty, Robbie Weber, Whitaker Brand, Zora Fung, Stuart Reges, Justin Hsia, Ruth Anderson, and many others for sample slides and materials ...

Announcements

- HW3 due Friday Noon
- Office hours for next week have changed. Please see the calendar for the correct info
- We made a mistake in a comment in HW4. We'll push a commit to your repo to correct that. (So expect one more git commit from us.)

Today

- Review Hashing
- Separate Chaining
- Open addressing with linear probing
- Open addressing with quadratic probing

Problem (Motivation for hashing)

How can we implement a dictionary such that dictionary operations are efficient?

Idea 1: Create a giant array and use keys as indices.

(This approach is called direct-access table or direct-access map)

Two main problems:

1. Can only work with integer keys?
2. Too much wasted space

Idea 2: What if we

(a) convert any type of key into a non-negative integer key

(b) map the entire key space into a small set of keys (so we can use just the right size array)

Solution to problem 1: Can only work with integer keys?

Idea: Use functions that convert a non-integer key into a non-negative integer key

Solution to problem 1: Can only work with integer keys?

Idea: Use functions that convert a non-integer key into a non-negative integer key

- Everything is stored as bits in memory and can be represented as an integer.
- But the representation can be much simpler (nothing to do with memory).
- For example (just for illustration; this is not how strings, images, and videos are hashed in practice):
  - Strings can be represented with number of characters in the string, ascii value of the first char, last char
  - Image can be represented with resolution, size of image, value of the 5th pixel in the image, 100th pixel
  - Similarly, video can be represented resolution, size, frame rate, size of the 10th frame

Solution to problem 1: Can only work with integer keys?

Idea: Use functions that convert a non-integer key into a non-negative integer key

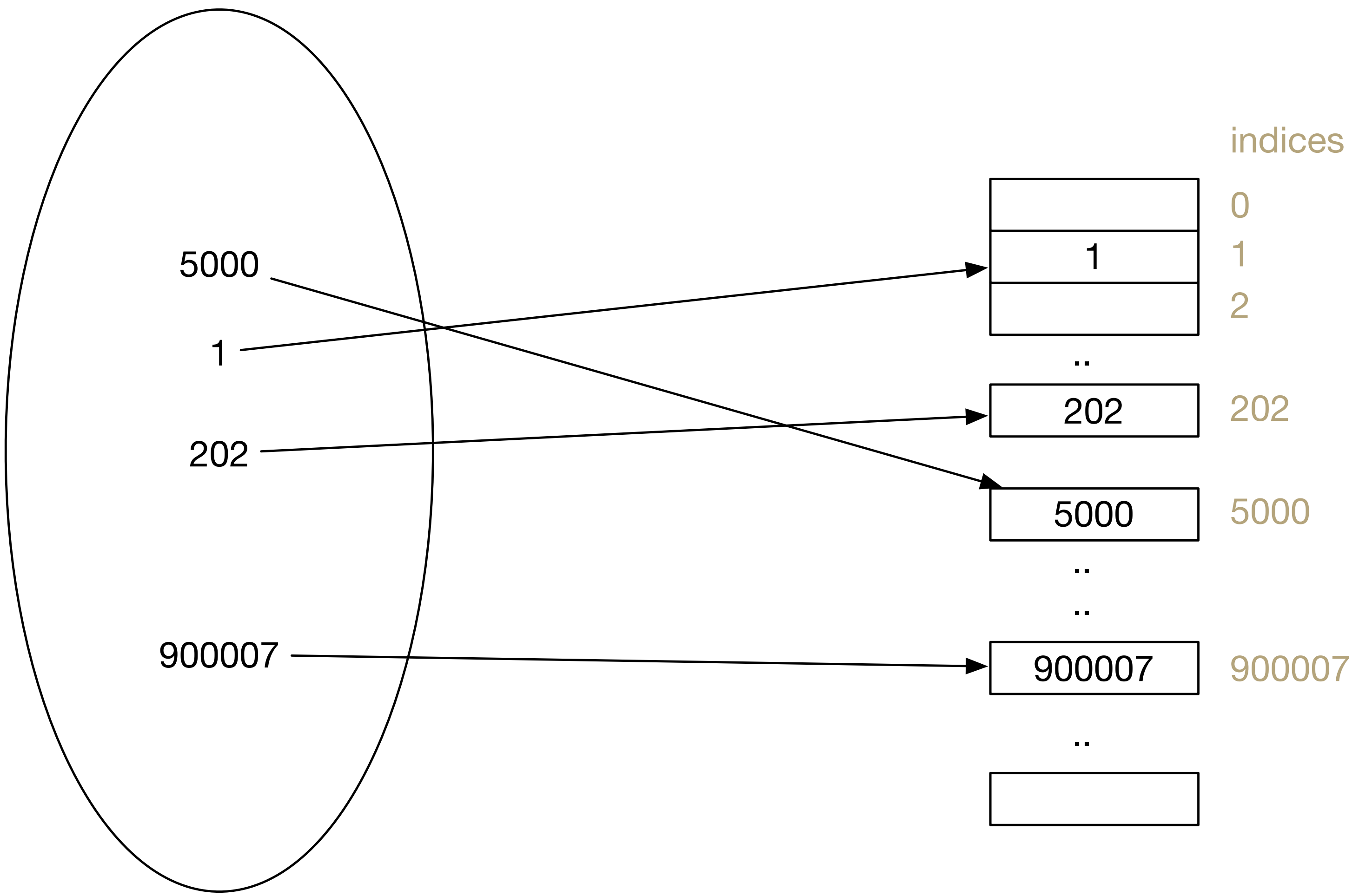
- Everything is stored as bits in memory and can be represented as an integer.
- But the representation can be much simpler (nothing to do with memory).
- For example (just for illustration; this is not how strings, images, and videos are hashed in practice):
  - Strings can be represented with number of characters in the string, ascii value of the first char, last char
  - Image can be represented with resolution, size of image, value of the 5th pixel in the image, 100th pixel
  - Similarly, video can be represented resolution, size, frame rate, size of the 10th frame

Question: What are some good strategies to pick a hash function? (This is important)

1. Quick: Computing hash should be quick (constant time).
2. Deterministic: Hash value of a key should be the same hash table.
3. Random: A good hash function should distribute the keys uniformly into the slots in the table.

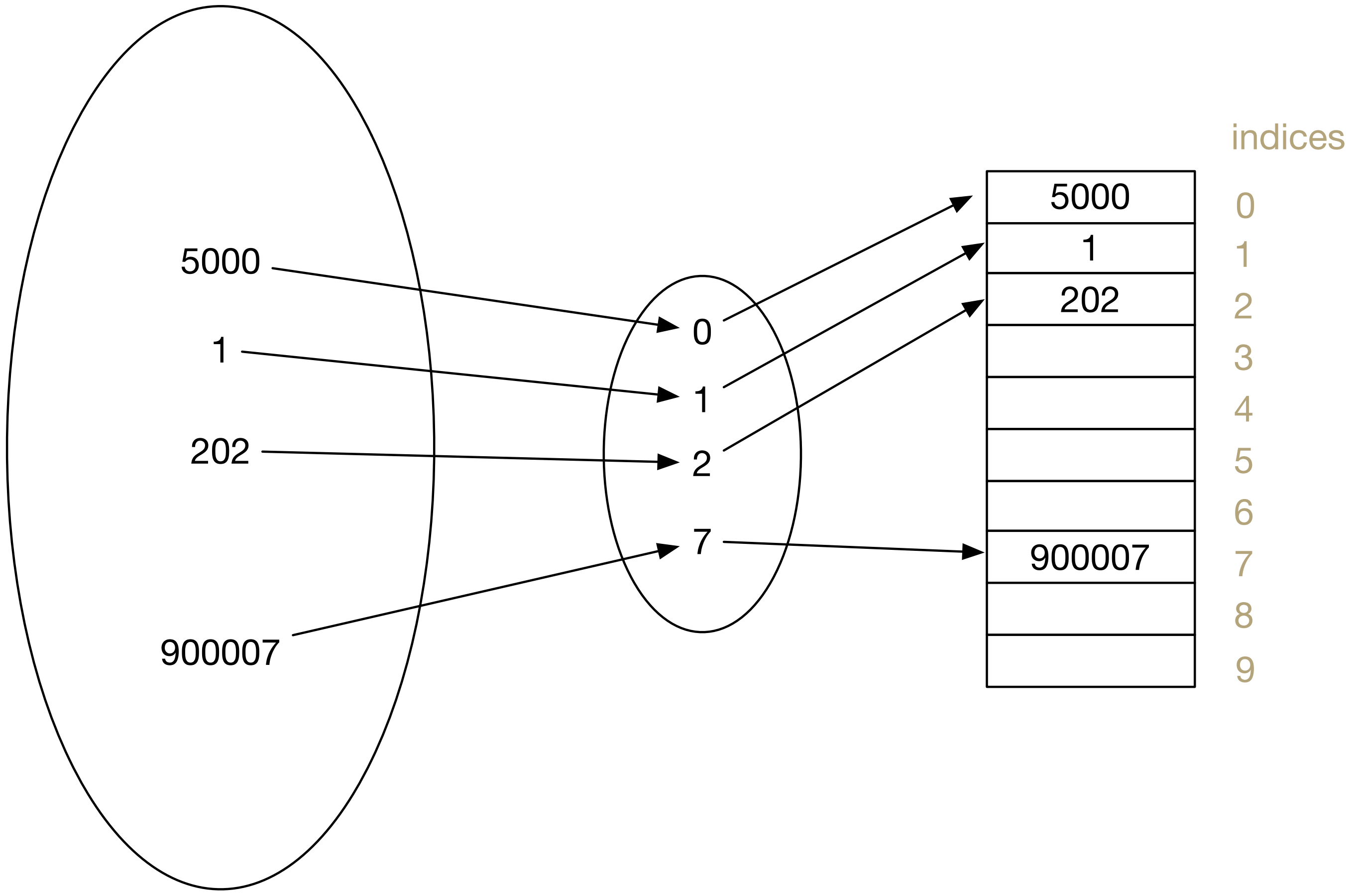
Solution to problem 2: Too much wasted space

Idea: Map the entire key space into a small set of keys (so we can use just the right sized array)



Solution to problem 2: Too much wasted space

Idea: Map the entire key space into a small set of keys (so we can use just the right sized array)



Review: The "modulus" (mod) operation

The "modulus" (mod) operation

The modulus (or mod) operation gives the remainder of a division of one number by another. Written as $x \bmod n$ or $x \;\%\; n$.

Examples:

1 % 10 = 1

11 % 10 = 1

10 % 10 = 0

5746 % 10 = 6

71 % 7 = 1

For more review/practice, check out https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/what-is-modular-arithmetic

Review: The “modulus” (mod) operation

The “modulus” (mod) operation

The modulus (or mod) operation gives the remainder of a division of one number by another. Written as $x \bmod n$ or $x \ \%\ n$.

Examples:

1 % 10 = 1

11 % 10 = 1

10 % 10 = 0

5746 % 10 = 6

71 % 7 = 1

Common applications of the mod operation:

- finding last digit (% 10)
- whether a number is odd/even (% 2)
- wrap around behavior (% wrap limit)

The application we are interested in is the wrap around behavior.
It lets us map any large integer into an index in our array of size m (using % m)

For more review/practice, check out https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/what-is-modular-arithmetic

Implementing a simple hash table (assume no collisions)

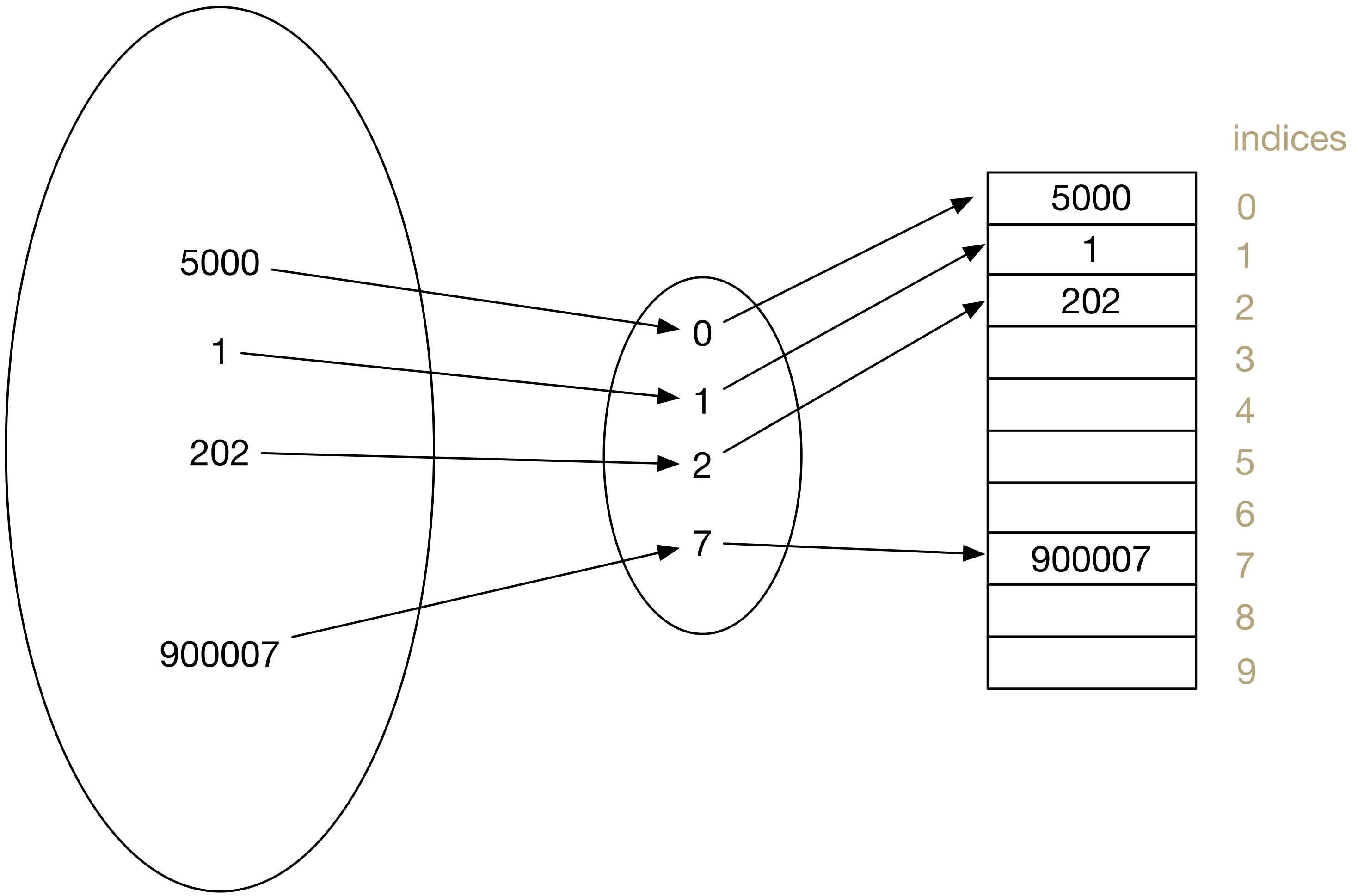
```
public V get(int key) {

   return this.array[key].value;
}

public void put(int key, V value) {

    this.array[key] = value;
}

public void remove(int key) {

   this.array[key] = null;
}
```

Implementing a simple hash table (assume no collisions)

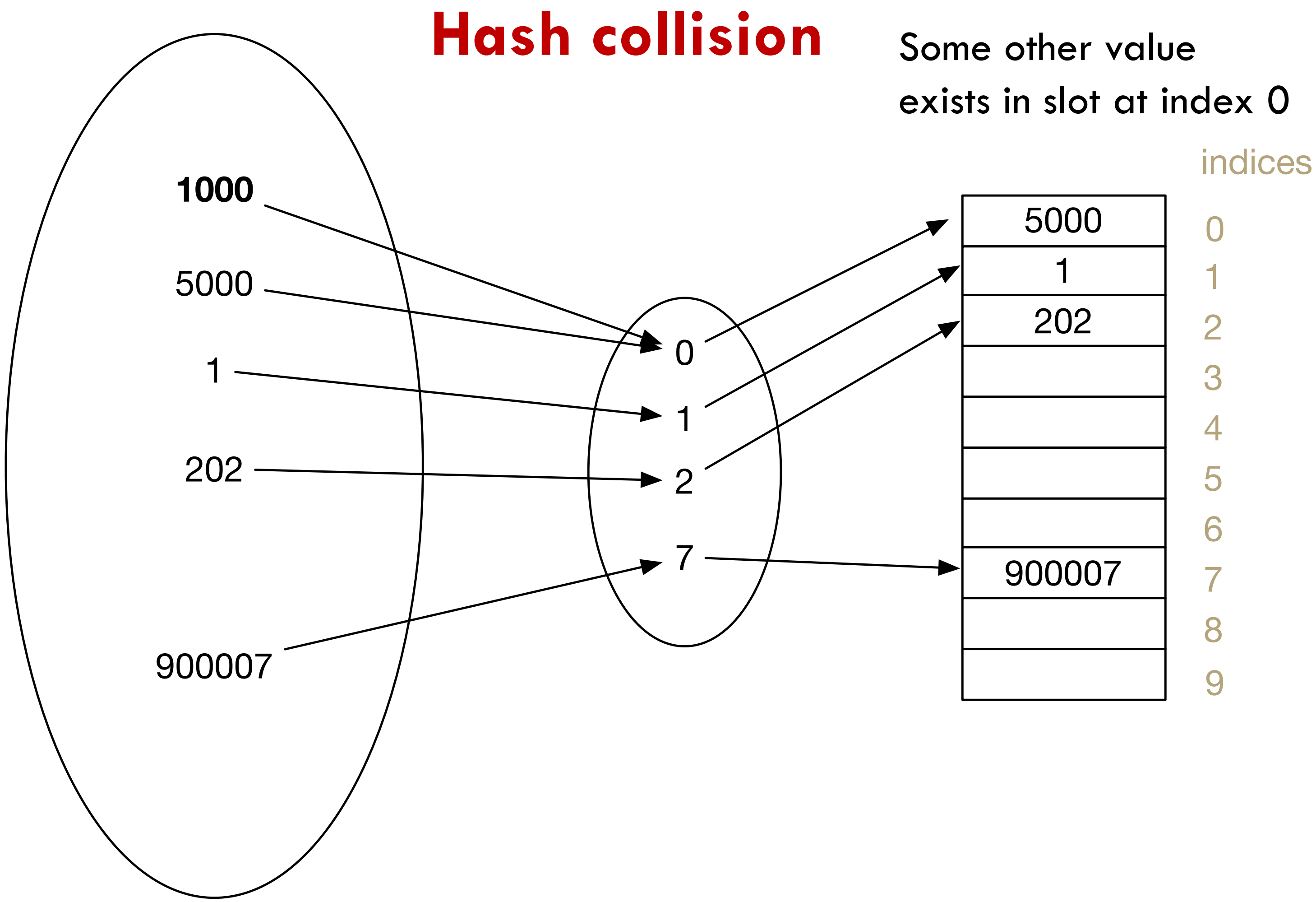
```java
public V get(int key) {
   key = getHash(key)
   return this.array[key].value;
}

public void put(int key, V value) {
    key = getHash(key)
    this.array[key] = value;
}

public void remove(int key) {
   key = getHash(key)
   this.array[key] = null;
}
```

```java
public int getHash(int a) {
    return a % this.array.length;
}
```

Our simple hash table: insert (1000)



Our simple hash table: insert (1000)



Hash collision

What is a hash collision?

It's a case when two different keys have the same hash value.
Mathematically, $h(k1) = h(k2)$ when $k1 \neq k2$

Hash collision

What is a hash collision?

It's a case when two different keys have the same hash value.
Mathematically, $h(k1) = h(k2)$ when $k1 \neq k2$

Why is this a problem?

- We put keys in slots determined by the hash function. That is, we put k1 at index h(k1),
- A collision means the natural choice slot is taken
- We cannot replace k1 with k2 (because the keys are different)
- So the problem is where do we put k2?

Strategies to handle hash collision

Strategies to handle hash collision

There are multiple strategies. In this class, we'll cover the following three:

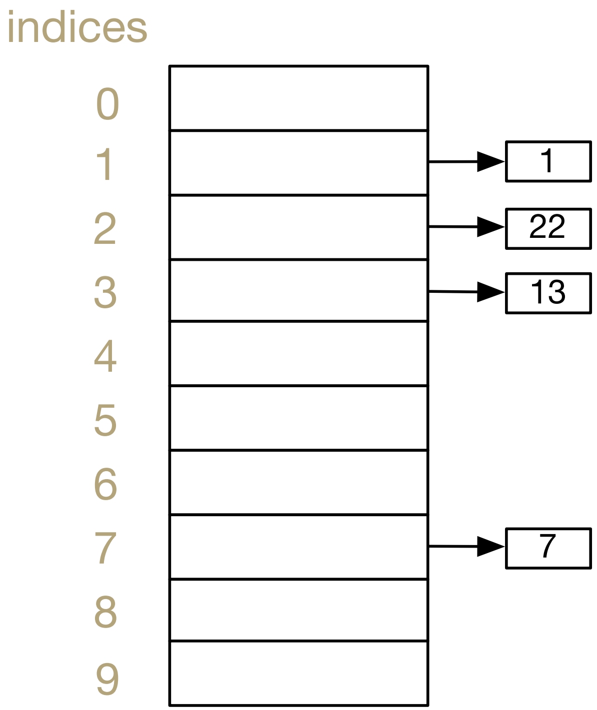
1. Separate chaining
2. Open addressing
   - Linear probing
   - Quadratic probing
3. Double hashing

Separate chaining

- Separate chaining is a collision resolution strategy where collisions are resolved by storing all colliding keys in the same slot (using linked list or some other data structure)

- Each slot stores a pointer to another data structure (usually a linked list or an AVL tree)

```
put(21, value21)

put(44, value44)
```

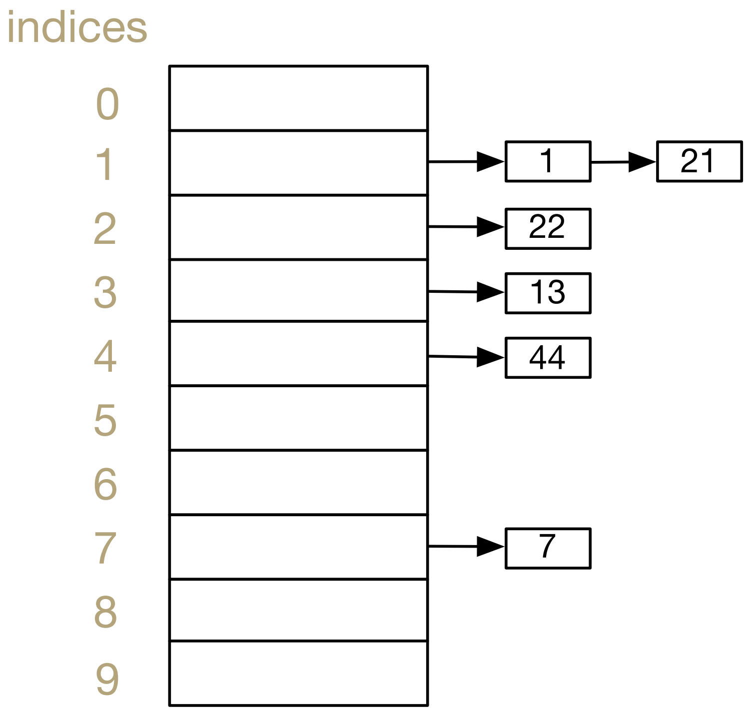


Note: For simplicity, the table shows only keys, but in each slot/node both, key and value, are stored.

Separate chaining

- Separate chaining is a collision resolution strategy where collisions are resolved by storing all colliding keys in the same slot (using linked list or some other data structure)
- Each slot stores a pointer to another data structure (usually a linked list or an AVL tree)

```
put(21, value21)

put(44, value44)
```



Note: For simplicity, the table shows only keys, but in each slot/node both, key and value, are stored.

Separate chaining: Running Times

What are the running times for:

`insert`

Best:

Worst:

`find`

Best:

Worst:

`delete`

Best:

Worst:

Separate chaining: Running Times

What are the running times for:

`insert`

Best: $O(1)$

Worst: $O(n)$ (if insertions are always at the end of the linked list)

`find`

Best: $O(1)$

Worst: $O(n)$

`delete`

Best: $O(1)$
Worst: $O(n)$

Load Factor

Load Factor (λ)

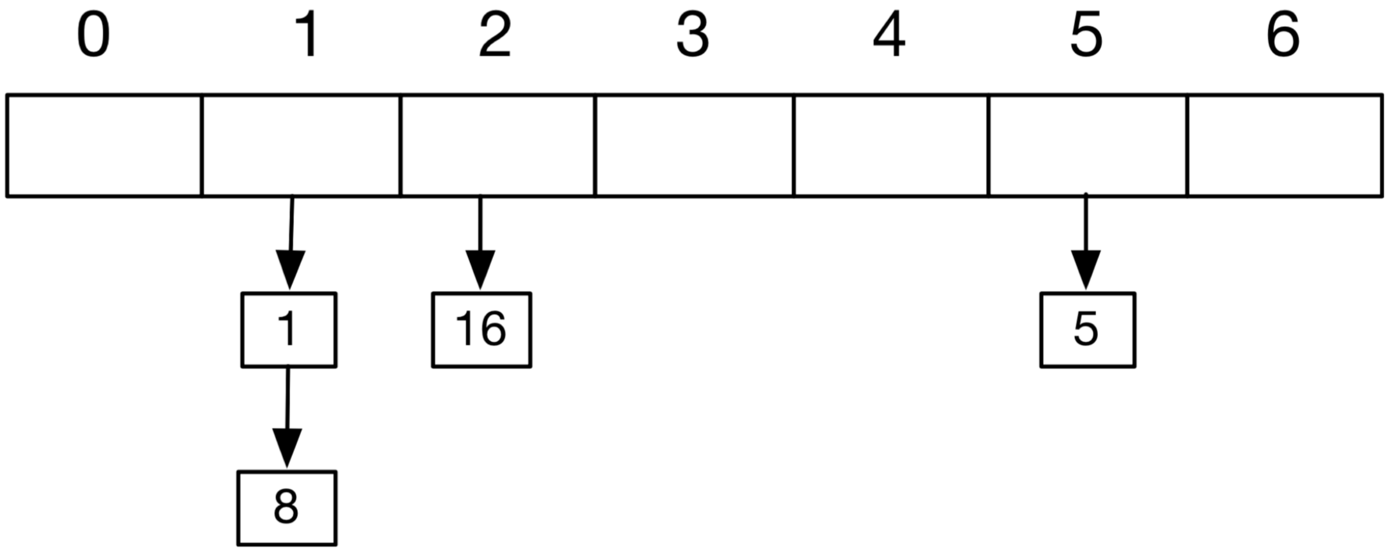
Ratio of number of entries in the table to table size.

If n is the total number of (key, value) pairs stored in the table and c is capacity of the table (i.e., array),

Load factor $\lambda = \frac{n}{c}$

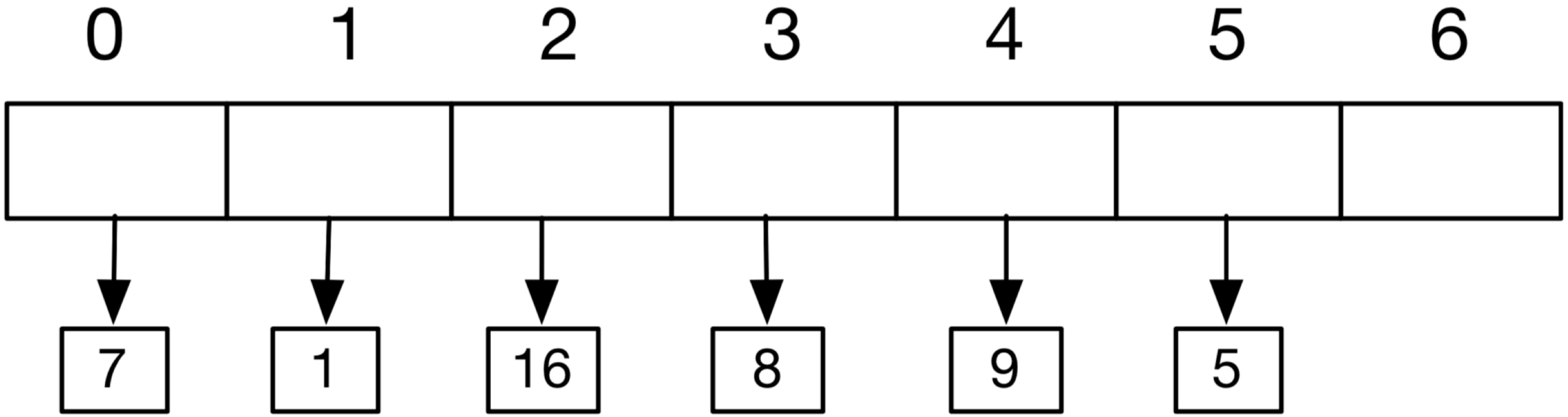
Worksheet Q1-Q3

Q1) The following table shows the resulting hash table after inserting keys 1, 16, 8, and 5. The hash table uses the hash function h(x) = x % 7 and separate chaining to avoid collisions. Now suppose **we insert keys 7 and 9** in this hash table. What would the resulting hash table look like (show where the values would be inserted).



Worksheet Q3

(Q3) What is the load factor of the following hash table?



Open Addressing

- Open addressing is a collision resolution strategy where collisions are resolved by storing the colliding key in a different location when the natural choice is full.

Open Addressing

- Open addressing is a collision resolution strategy where collisions are resolved by storing the colliding key in a different location when the natural choice is full.

`put(21, value21)`

indices	
0	
1	1
2	22
3	13
4	
5	
6	
7	7
8	
9	

Note: For simplicity, the table shows only keys, but in each slot both, key and value, are stored.

Open Addressing: Linear probing

- Open addressing is a collision resolution strategy where collisions are resolved by storing the colliding key in a different location when the natural choice is full.

`put(21, value21)`

indices	
0	
1	1
2	22
3	13
4	
5	
6	
7	7
8	
9	

Linear probing

Index = hash(k) + 0 (if occupied, try next i)
= hash(k) + 1 (if occupied, try next i)
= hash(k) + 2 (if occupied, try next i)
= ..
= ..
= ..

Note: For simplicity, the table shows only keys, but in each slot both, key and value, are stored.

Open Addressing: Quadratic probing

- Open addressing is a collision resolution strategy where collisions are resolved by storing the colliding key in a different location when the natural choice is full.

```
put(21, value21)
```

indices	
0	
1	1
2	22
3	13
4	
5	
6	
7	7
8	
9	

Quadratic probing

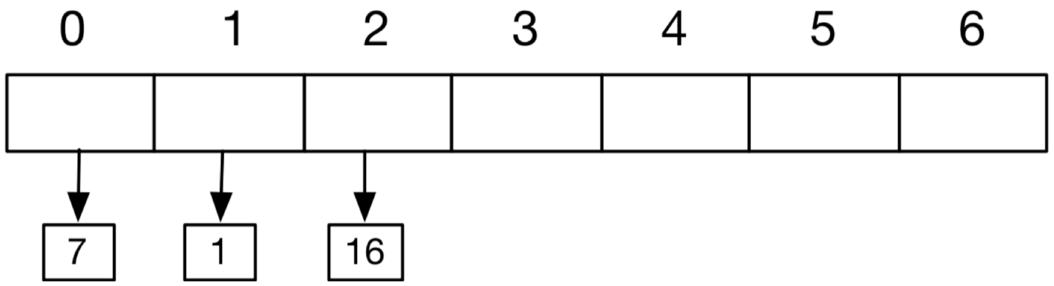
Index = hash(k) + 0 (if occupied, try next i^2)
= hash(k) + 1^2 (if occupied, try next i^2)
= hash(k) + 2^2 (if occupied, try next i^2)
= hash(k) + 3^2 (if occupied, try next i^2)
= ..
= ..

Note: For simplicity, the table shows only keys, but in each slot both, key and value, are stored.

Worksheet Q4

(Q4) Each table uses the hash function h(x) = x % 7, but different collision handling strategies. **Show where key 8 will be inserted** in the following hash tables.

(4a) The following hash tables uses separate chaining



(4b) The following hash tables uses open addressing with linear probing

0	1	2	3	4	5	6
7	1	16				

(4c) The following hash tables uses open addressing with quadratic probing.

0	1	2	3	4	5	6
7	1	16				

Worksheet Q5

(Q5) What is the worst case tight-O for the following operations:

(5a) Insert in a separate chaining hash table:

(5b) Insert in an open addressing hash table that uses linear probing to resolve collisions:

(5c) Find in a separate chaining hash table:

(5d) Find in an open addressing hash table that uses linear probing to resolve collisions: