INFS7901
Database Principles

Hashing

Hassan Khosravi & Ash Rahimi

Notes

- Anonymous feedback link on Blackboard.
- Project specification marks and feedback posted.
  - Please come to consultation hours if you need more feedback.
- RiPPLE round 4 and Project final submission due 26th May 3pm.
  - Don’t postpone everything for the last week.
  - Have regular group meetings on Zoom (maybe twice a week).
  - Come to consultation hours for more feedback.

Notes

- Lecture: Wrapping on Module 2. Interested in a more in-depth discussion? See the online algorithm course by Tim Roughgarden from Stanford.
- Module 3: Indexing and Query Processing to be covered during weeks 11 and 12

The Tree Jupyter Notebook

Learning Outcomes

Description	Tag
Demonstrate how hash tables can be used for implementation of a dictionary ADT.	Hashing
Describe characteristics of a good hash function	
Explain and apply the pigeonhole principle.	
Explain collision in hashing.	
Describe the properties of chaining for hashing.	Chaining
insert, delete, and find using chaining algorithms.	
Describe the properties of open addressing.	Open-addressing
Insert, delete and find using linear probing.	
Insert, delete and find using quadratic probing.	
Insert, delete and find using double hashing.	
Explain the notion of primary clustering in hashing.	
Explain the notion of secondary clustering in hashing.	
Compare and contrast different hashing algorithms.	Hashing
Reflect on how hashing can be used in DBMS.	

High-level view of Hashing

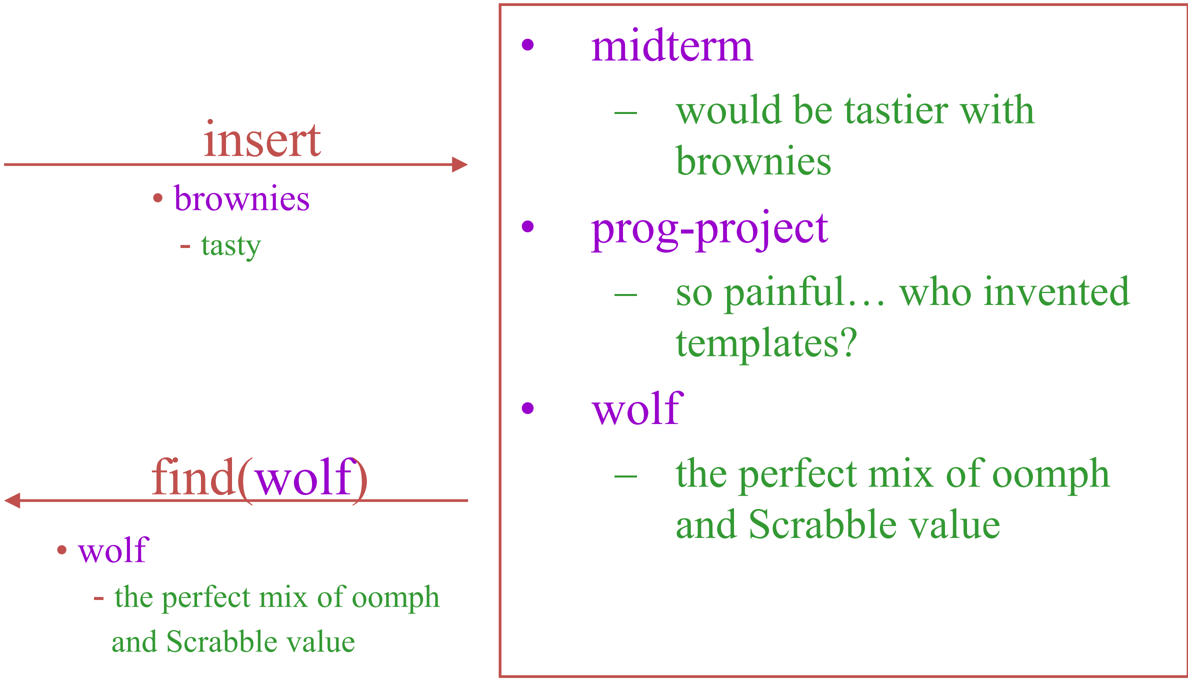
Collision and Hash Functions

Chaining

Open Addressing

Reminder: Dictionary ADT

- Dictionary operations
  - create
  - destroy
  - insert
  - find
  - delete



insert

- brownies
  - tasty

find(wolf)

- wolf
  - the perfect mix of oomph and Scrabble value

- midterm
  - would be tastier with brownies
- prog-project
  - so painful… who invented templates?
- wolf
  - the perfect mix of oomph and Scrabble value

- Stores values associated with user-specified keys
  - values may be any (homogenous) type
  - keys may be any (homogenous) comparable type

Dictionary Implementations

	insert	*find*	*delete* *By value*	*delete* *By address*
• Linked list				
– Unsorted	$O(1)$	$O(n)$	$O(n)$	$O(1)$
– Sorted	$O(n)$	$O(n)$	$O(n)$	$O(1)$
• Array				
– Unsorted	$O(1)$	$O(n)$	$O(n)$	$O(1)$
– Sorted	$O(n)$	$O(\lg n)$	$O(n)$	$O(n)$
• Tree				
– BST	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$

Can we do better? O(1)?

Example 1 (natural, numeric keys)

- In a small company of 100 employees, each employee is assigned an Emp_ID number in the range 0 – 99.
- To store the employee’s records in an array, each employee’s Emp_ID number acts as an index into the array where this employee’s record will be stored as shown in figure

KEY	ARRAY OF EMPLOYEE’ S RECORD
Key 0 [0]	**Record of employee having Emp_ID 0**
Key 1 [1]	**Record of employee having Emp_ID 1**
…………………………..	**…………………………………………………..**
Key 99 [99]	**Record of employee having Emp_ID 99**

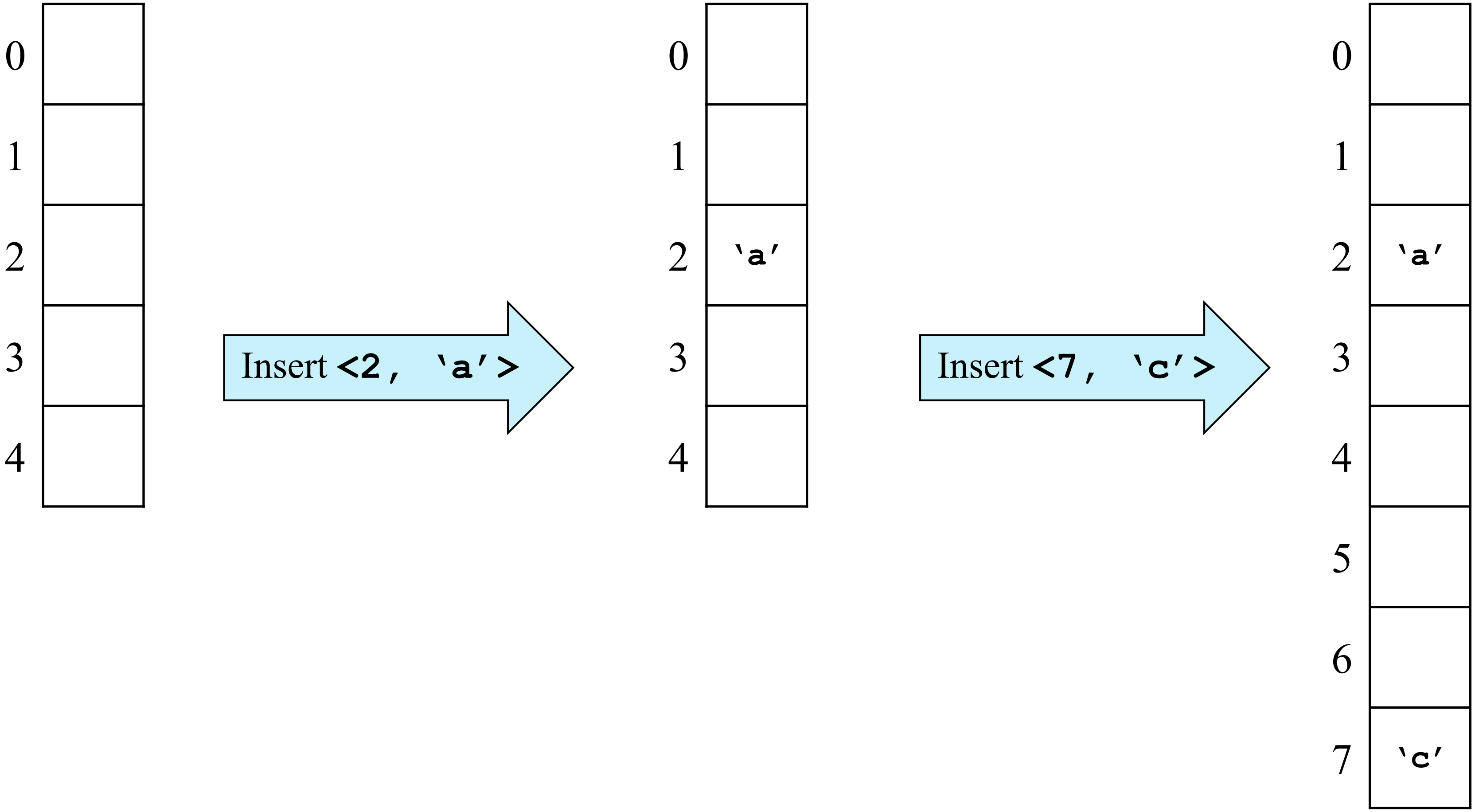
Follow-up example

- Let's assume that the same company uses a five digit Emp_ID number as the primary key. In this case, key values will range from 00000 to 99999. If we want to use the same technique as above, we will need an array of size 100,000, of which only 100 elements will be used.

KEY	ARRAY OF EMPLOYEE' S RECORD
Key 00000 [0]	Record of employee having Emp_ID 00000
…………………………	…………………………………………………
Key n [n]	Record of employee having Emp_ID n
…………………………	………………………………………………...
Key 99999 [99999]	Record of employee having Emp_ID 99999

- It is impractical to waste that much storage just to ensure that each employee's record is in a unique and predictable location.

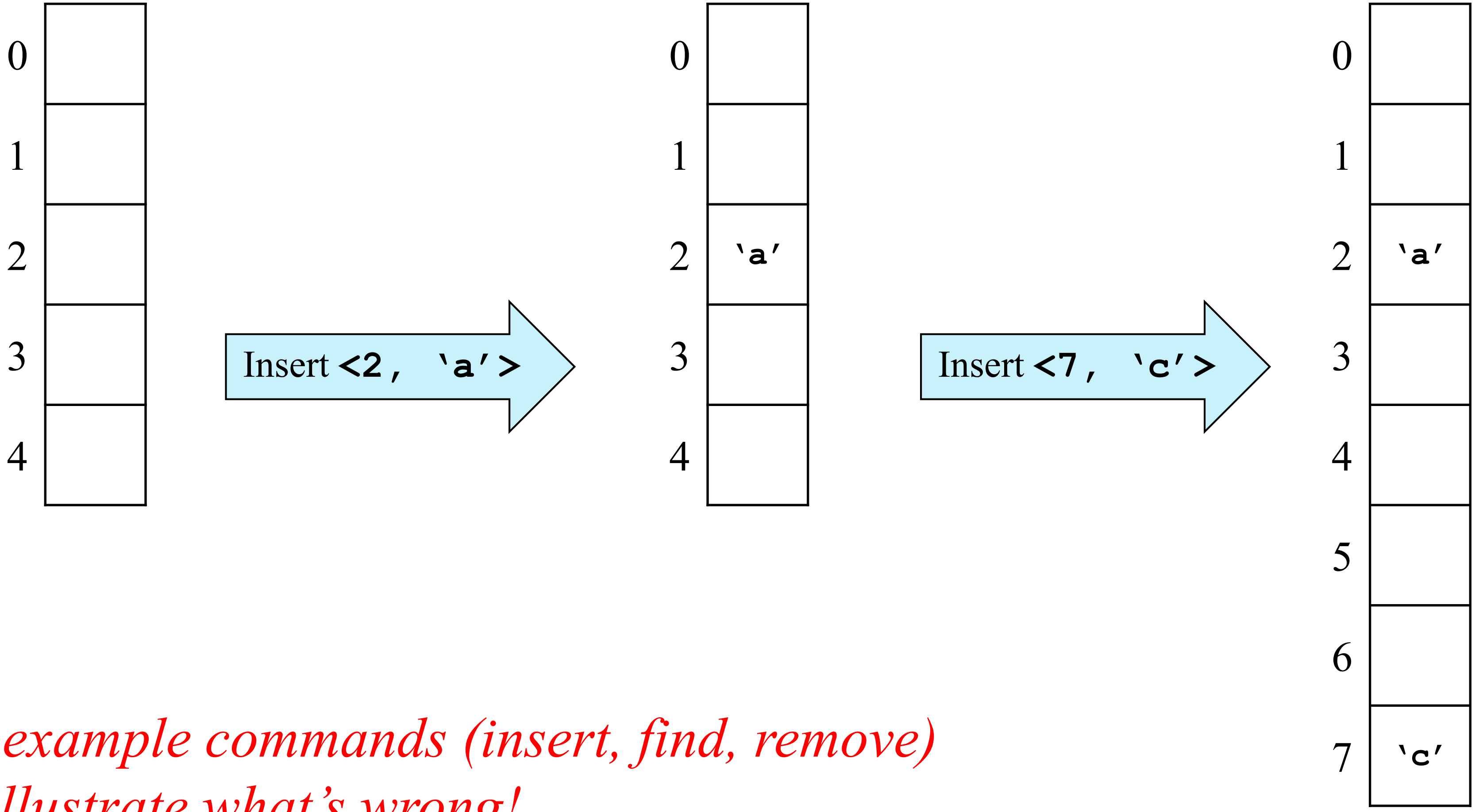
First Pass: Resizable Vectors



Insert **<2, 'a'>**

Insert **<7, 'c'>**

What's Wrong with Our First Pass?

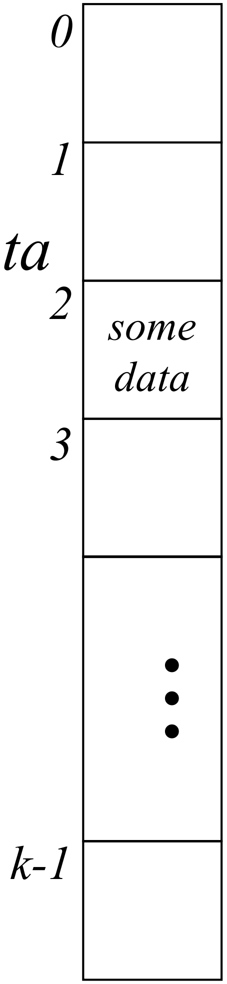


Give example commands (insert, find, remove) that illustrate what's wrong!

Hash Table Goal

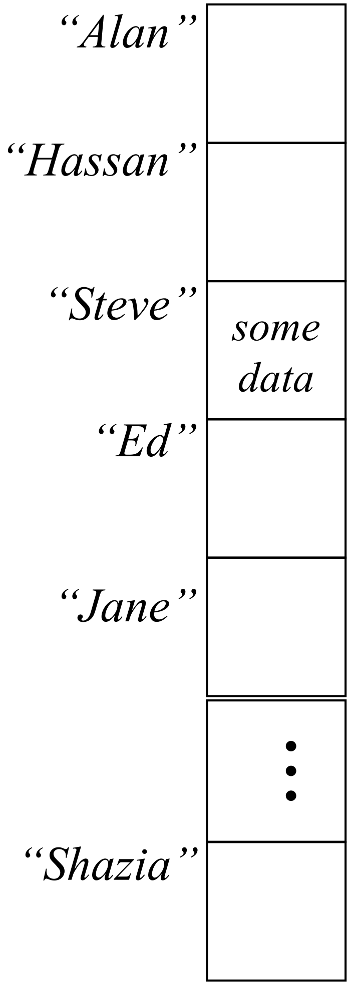
We can do:

a[2] = some data



We want to do:

a["Steve"] = some data

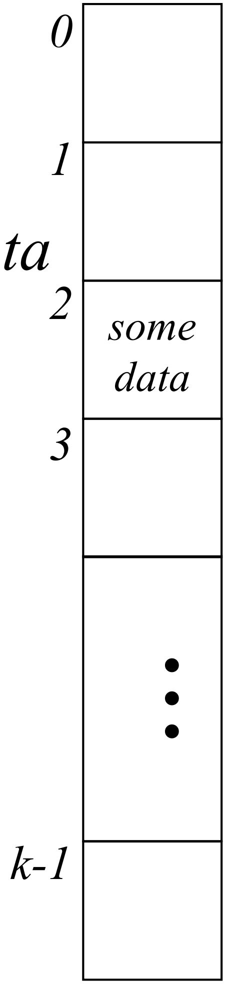


How will insert, find, and delete work?

Aside: How do arrays do that?

We can do:

a[2] = some data



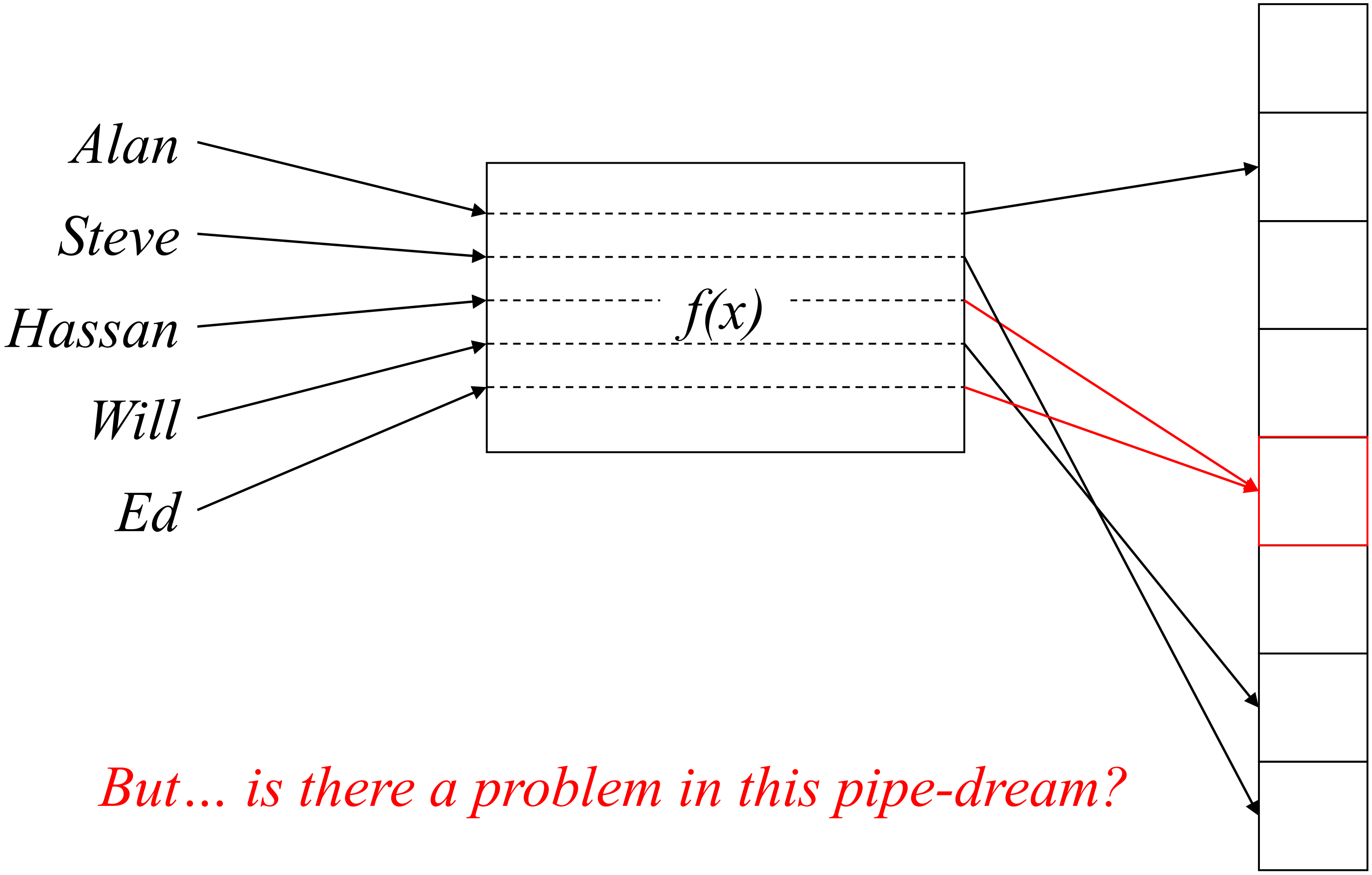
Q: If I know houses on a certain block on 33-foot-wide lots, where is the 5th house?
A: It's from (5-1)*33 to 5*33 feet from the start of the block.

element_type a[SIZE];

Q: Where is a[i]?
A: start of a + i*sizeof(element_type)

Aside: This is why array elements have to be the same size, and why we start the indices from 0.

Hash Table Approach

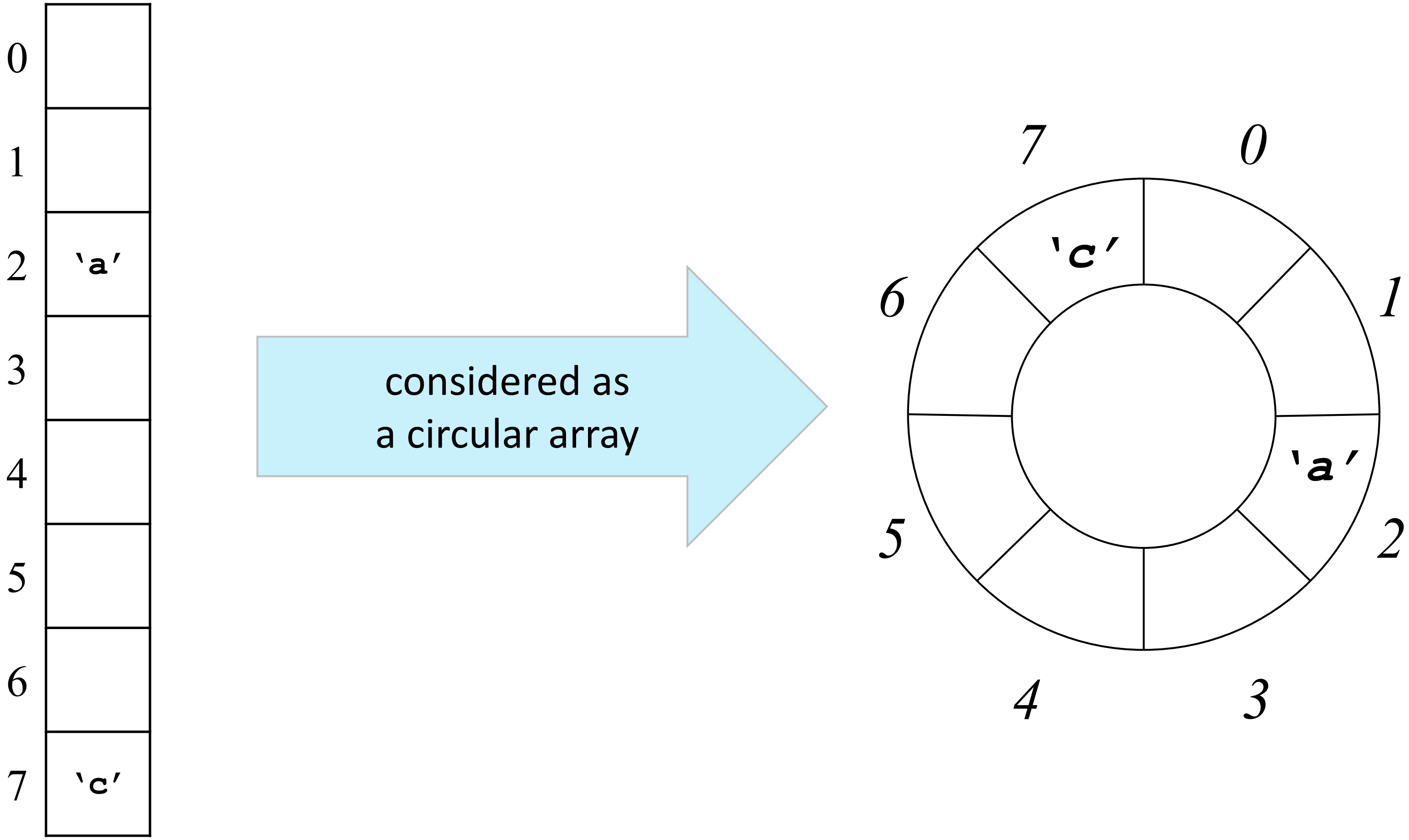


But… is there a problem in this pipe-dream?

What is the 25th Element?

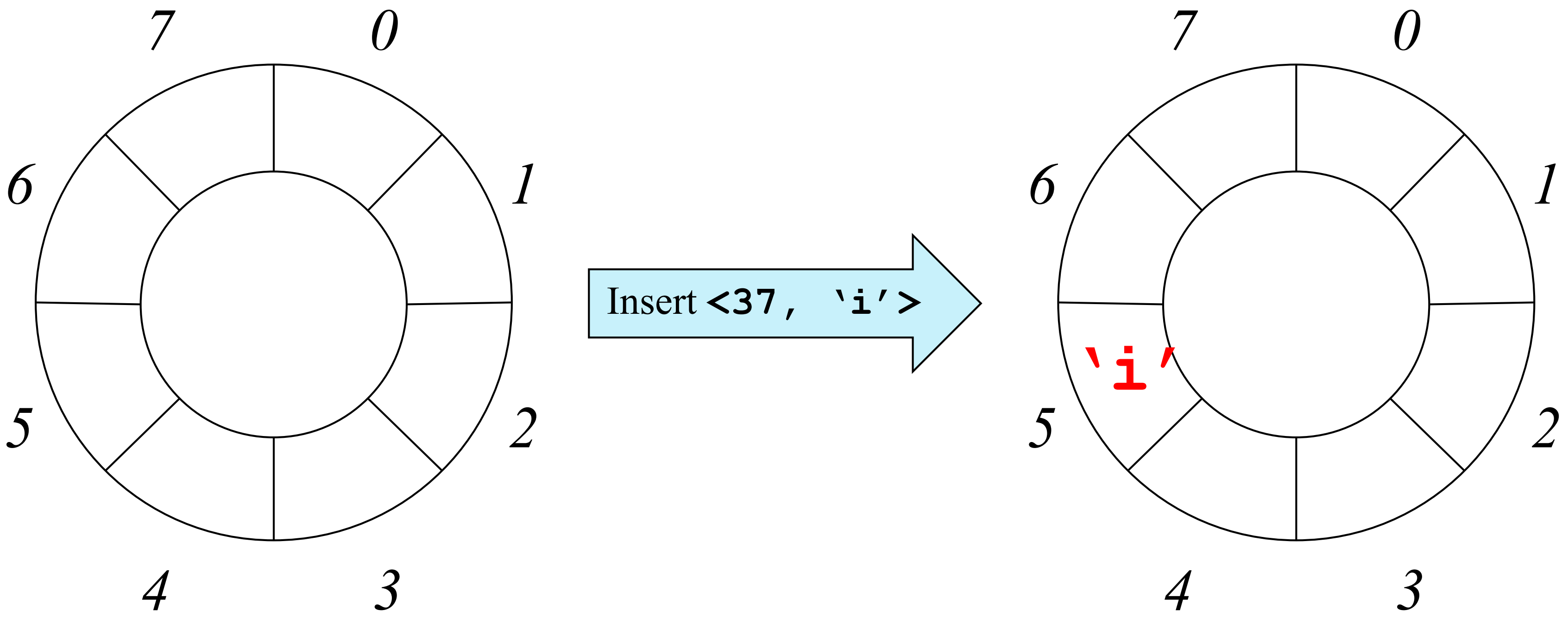
0	
1	
2	`'a'`
3	
4	
5	
6	
7	`'c'`

What is the 25th Element Now?



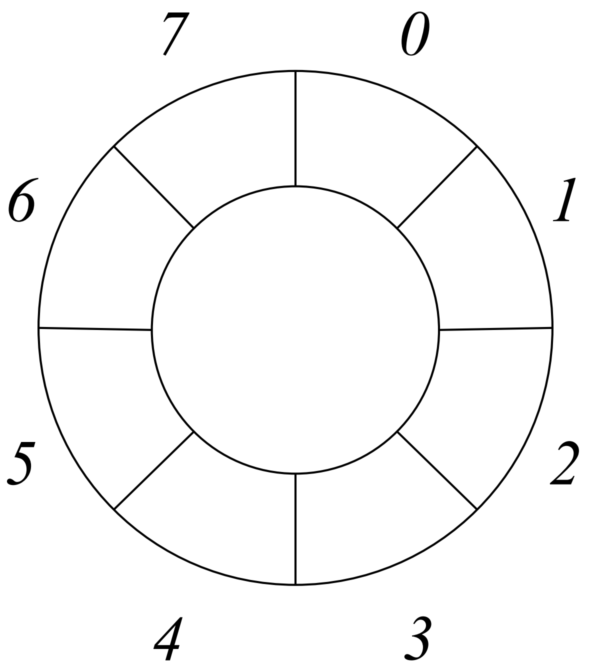
considered as
a circular array

Second Pass: Circular Array (For the Win?)



Does this solve our memory usage problem?

What's Wrong with our **Second** Pass?



Let's insert 2 and 258?

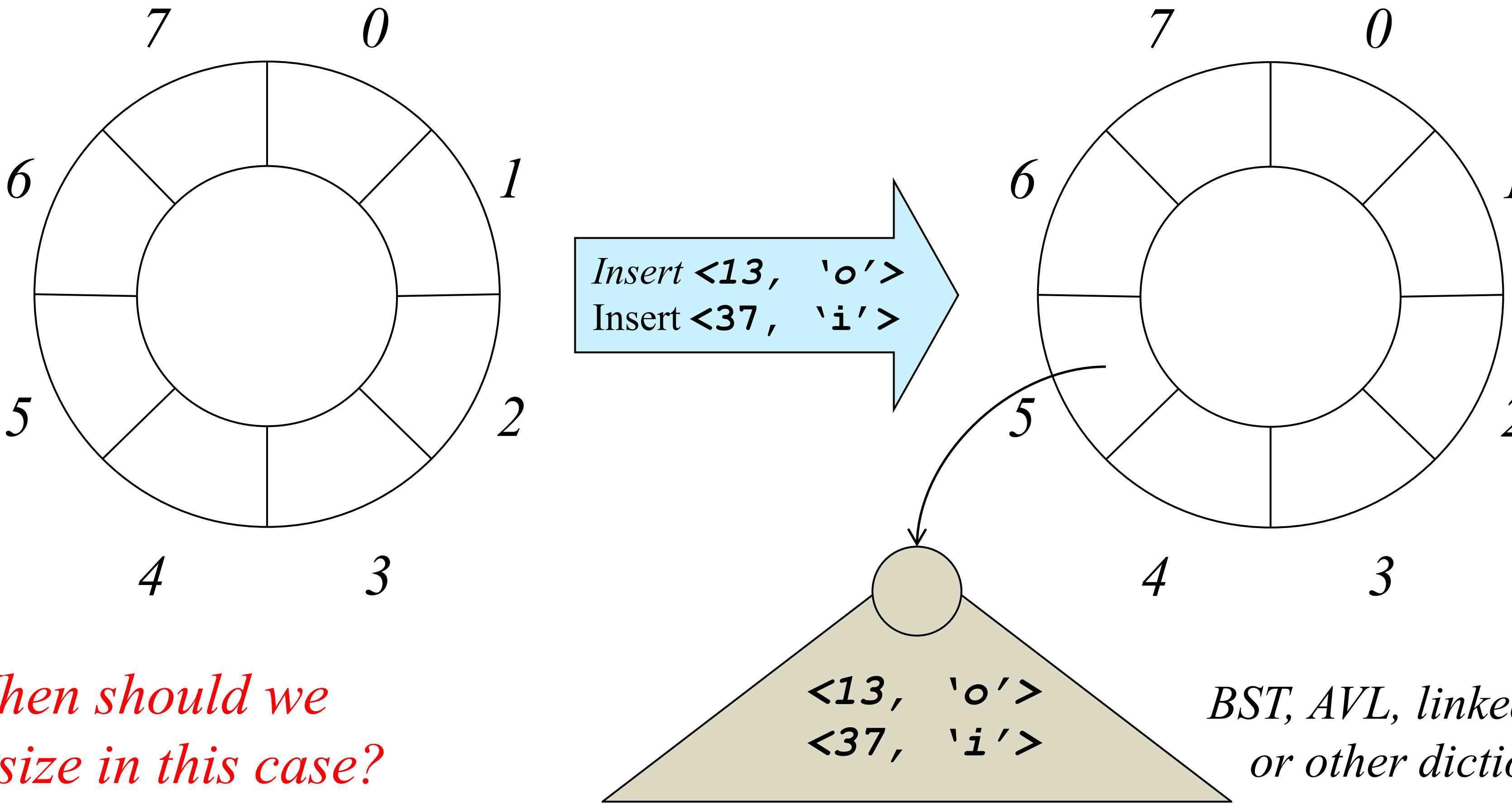
258 % 8 = 2
258 % 16 = 2
258 % 32 = 2
258 % 64 = 2
258 % 128 = 2
258 % 256 = 2

Resize until they don't?

Solutions:

- Prime table sizes helps
- Some way to handle these collisions without resizing?

Third Pass: Punt to Another Dictionary?



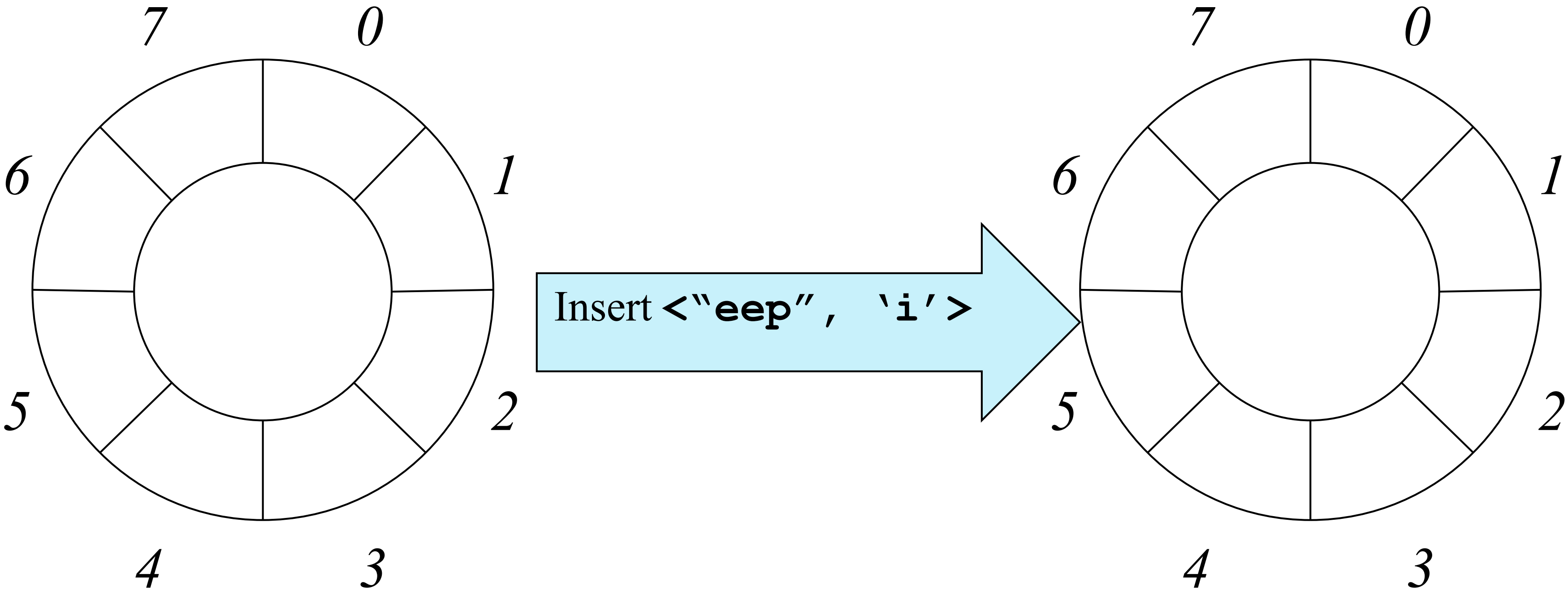
Insert **`<13, 'o'>`**
Insert **`<37, 'i'>`**

`<13, 'o'>`
`<37, 'i'>`

BST, AVL, linked list, or other dictionary

When should we resize in this case?

How Do We Turn Strings into Numbers?



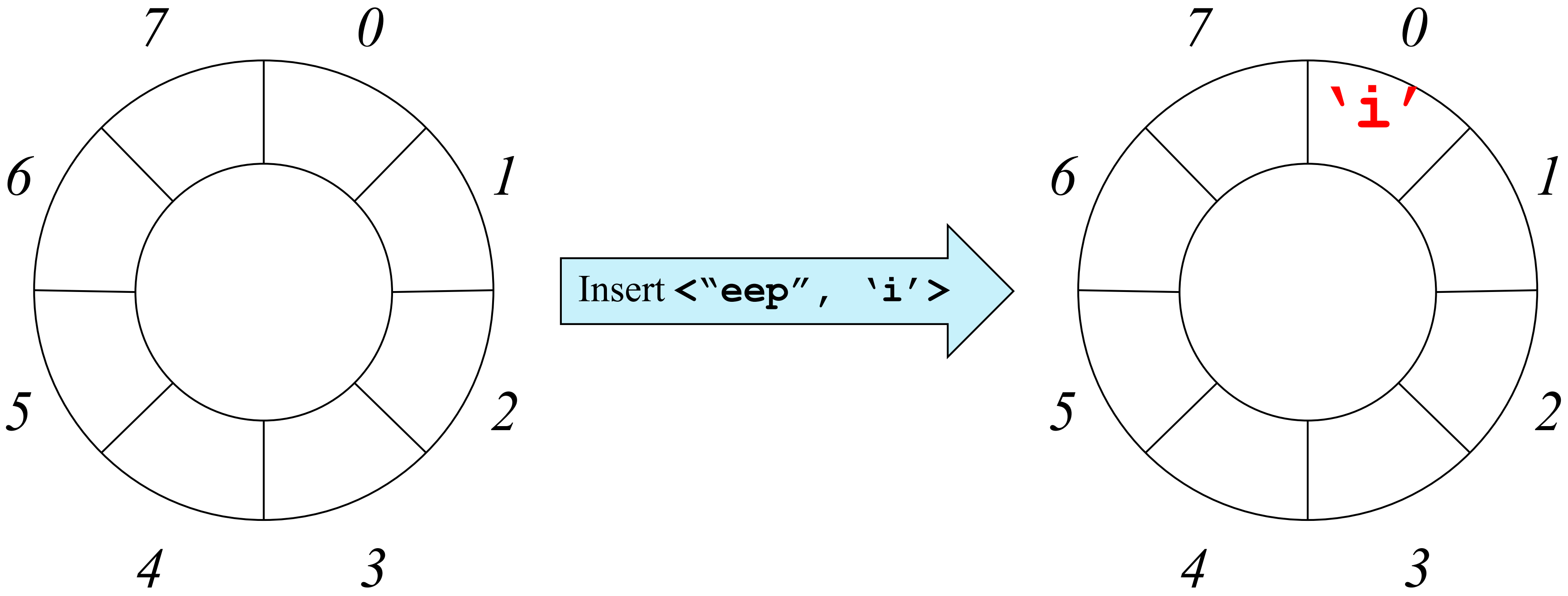
Insert **<“eep”, ‘i’>**

What should we do?

Fourth Pass: Strings **ARE** Numbers

e	e	p
01100101	01100101	01110000

= 6,645,104



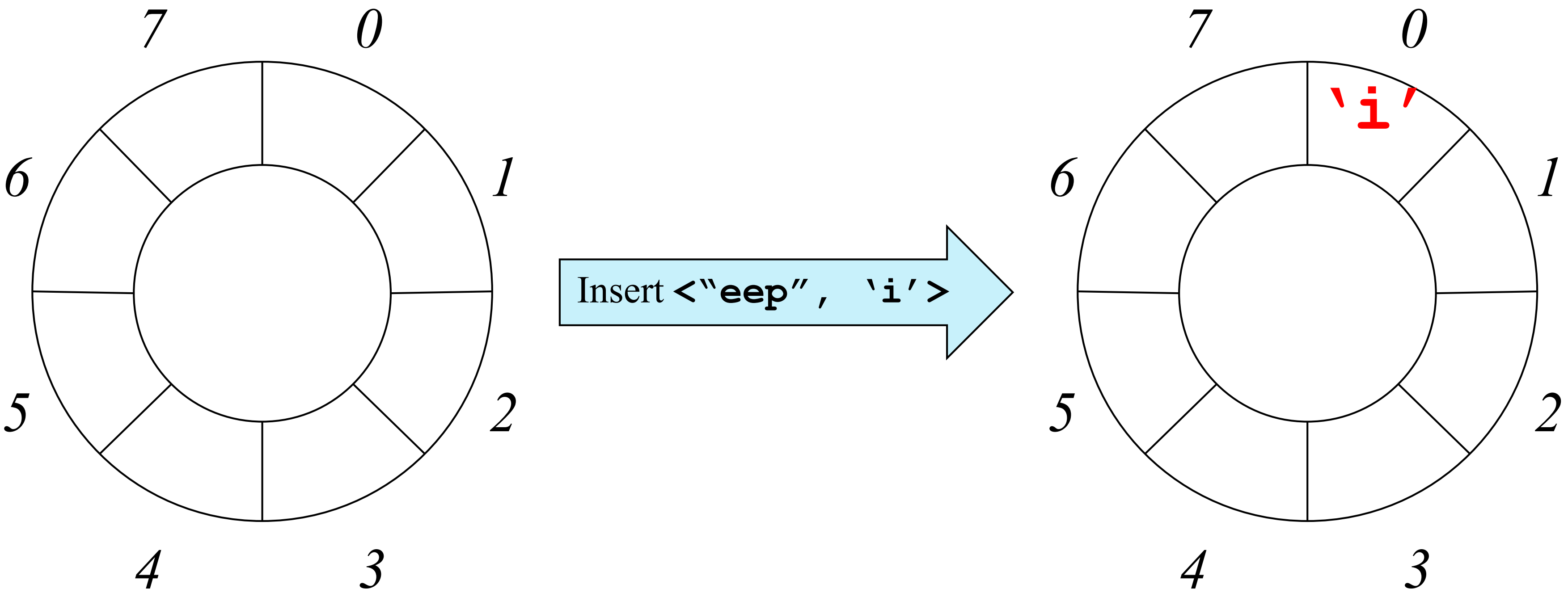
Insert **<"eep", 'i'>**

6,645,104 % 8 = 0

Fourth Pass: Strings **ARE** Numbers

e	**e**	**p**
01100101	01100101	01110000

= *6,645,104*



Insert <“eep”, ‘i’>

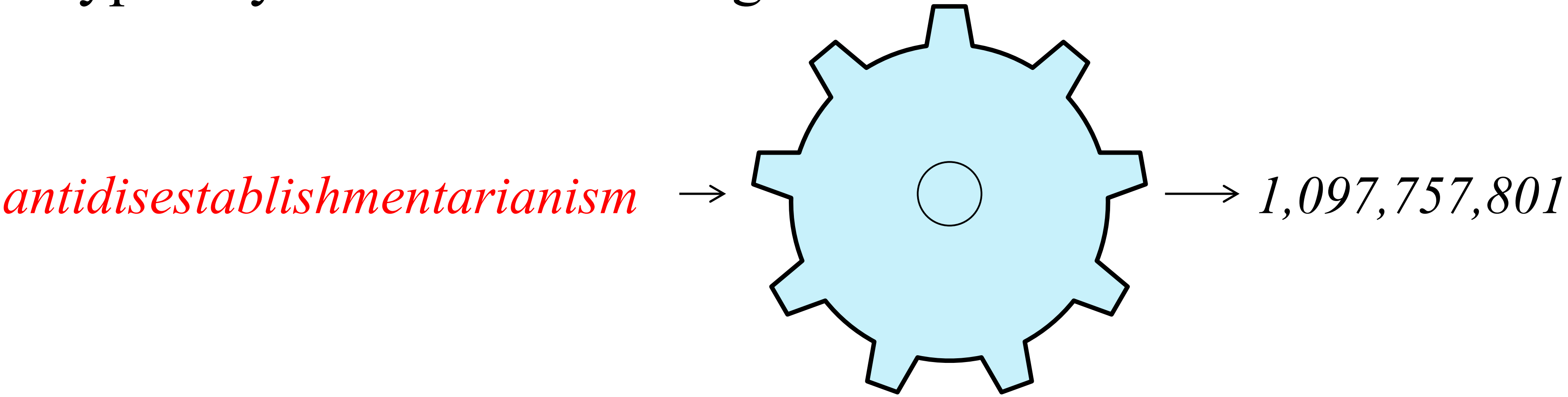
Those numbers get REALLY big
antidisestablishmentarianism. Just saying.

6,645,104 % 8 = 0

Fifth Pass: Hashing!

- We only need perhaps a 64 (128?) bit number. There’s no point in forming a **huge** number.

- We need a function to turn the strings into numbers, typically on a bounded range…



antidisestablishmentarianism → → *1,097,757,801*

Maybe we can only use some parts of the string

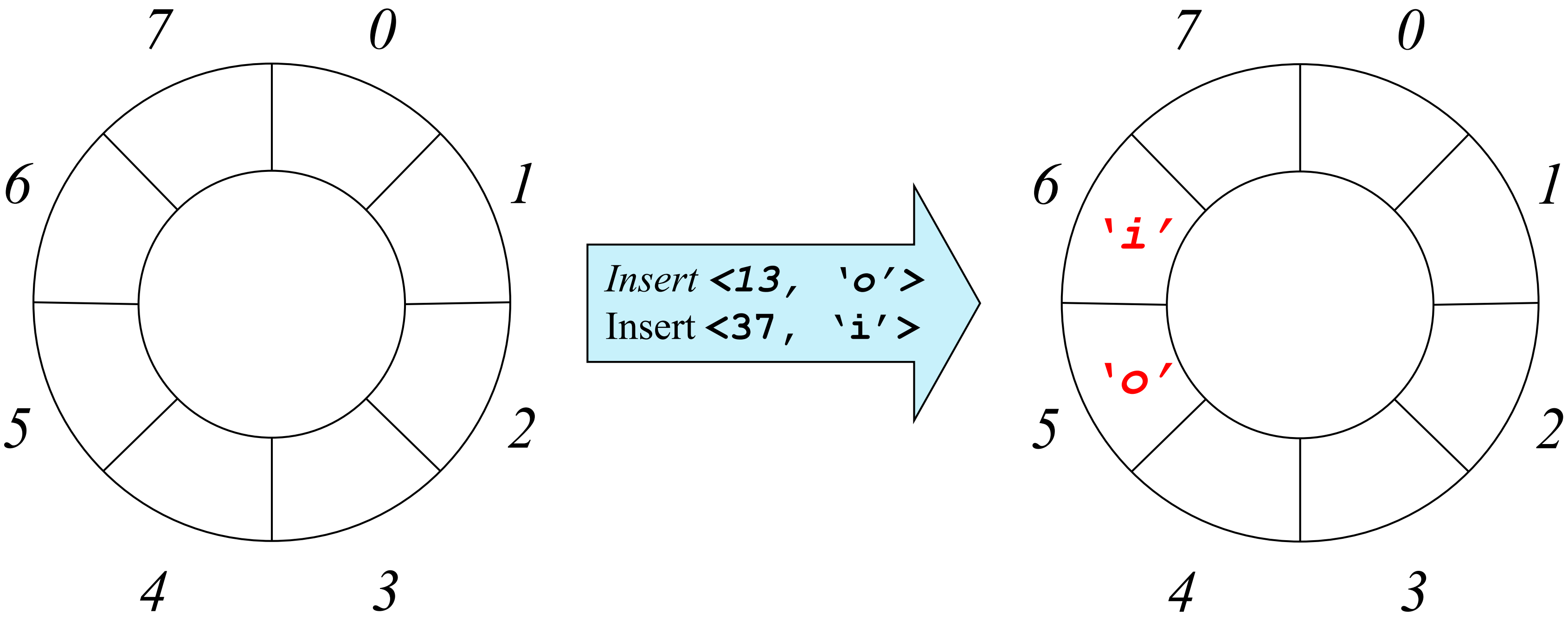
Schlemiel, Schlemazel, Trouble for Our Hash Table?

- Let’s try out:
  - “schlemiel” and “schlemazel”?
  - “microscopic” and “telescopic”?
  - “abcdefghijklmnopqrstuvwxyzyxwvutsrqponmlkjihgfedcba” and “abcdefghijklmnopqrstuvwxyzzyxwvutsrqponmlkjihgfedcba”

- Which bits of the string should we keep? Does our hash table care?

That’s hashing! Take our data and turn it into a sorta-random number, ideally one that spreads out similar strings far apart!

Third Pass, Take Two: Punt to Another Slot?



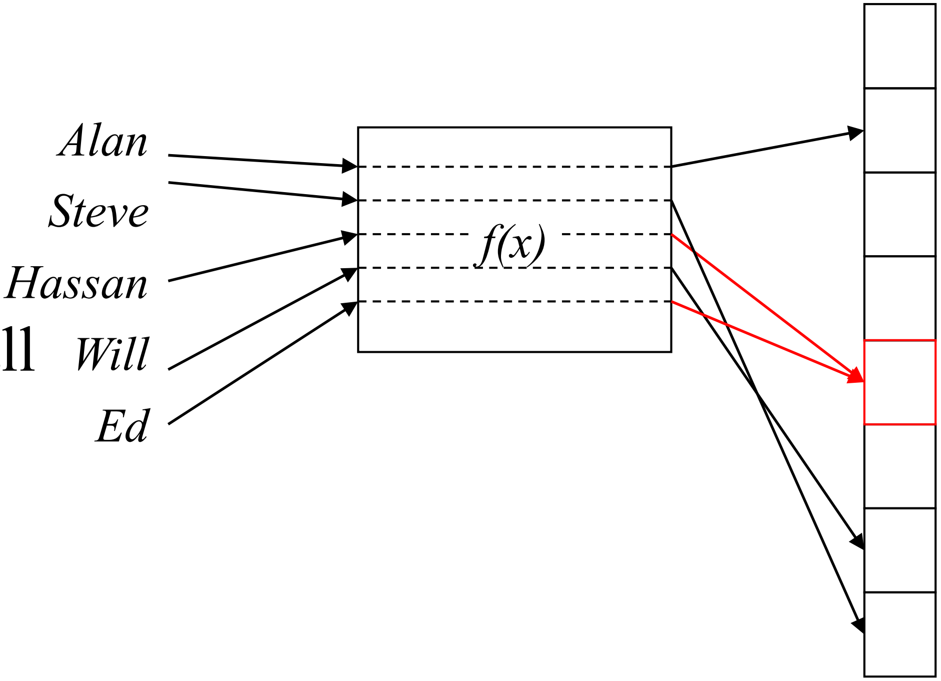
Insert **`<13, 'o'>`**

Insert **`<37, 'i'>`**

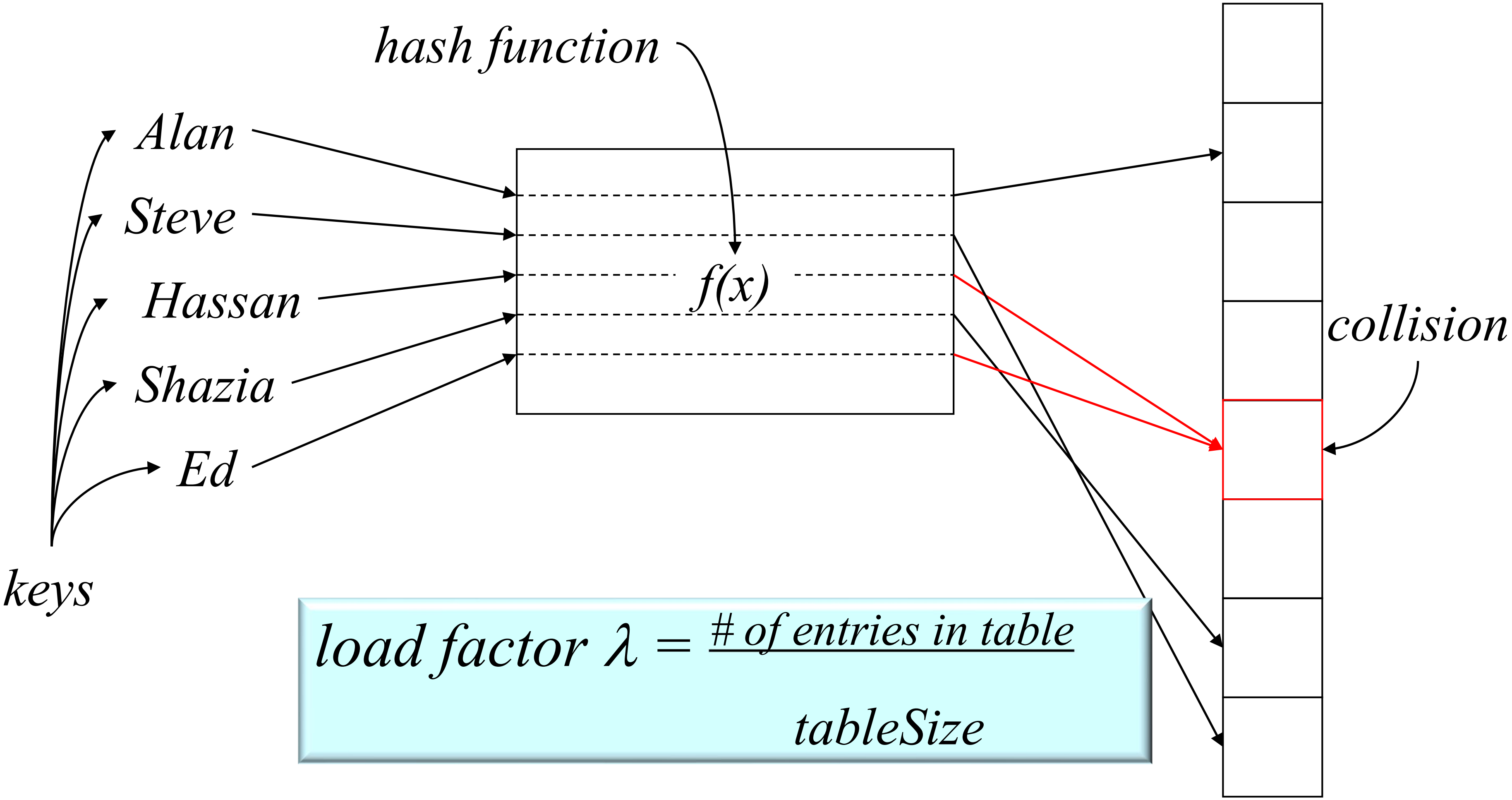
Slot 5 is full, but no "dictionaries in each slot" this time.
Overflow to slot 6? When should we resize?

Hash Table Dictionary Data Structure

- Hash function: maps keys to integers
  - result: can quickly find the right spot for a given entry
- Unordered and sparse table
  - result: cannot efficiently list all entries, ***definitely*** cannot efficiently list all entries in order or list entries between one value and another (a "range" query)



Hash Table Terminology



$$\textit{load factor } \lambda = \frac{\textit{\# of entries in table}}{\textit{tableSize}}$$

Hash Class

```
class HashTable:

    def __init__(self):
        self.size = 11
        self.slots = [None] * self.size
        self.data = [None] * self.size
```

Hash Table Code First Pass

```
Def get(self, Key):
  int index = hash(key) % self.size
  return self.data[index];
```

- What should the hash function be?
- What should the table size be?
- How should we resolve collisions?

High-level view of Hashing

Collision and Hash Functions

Chaining

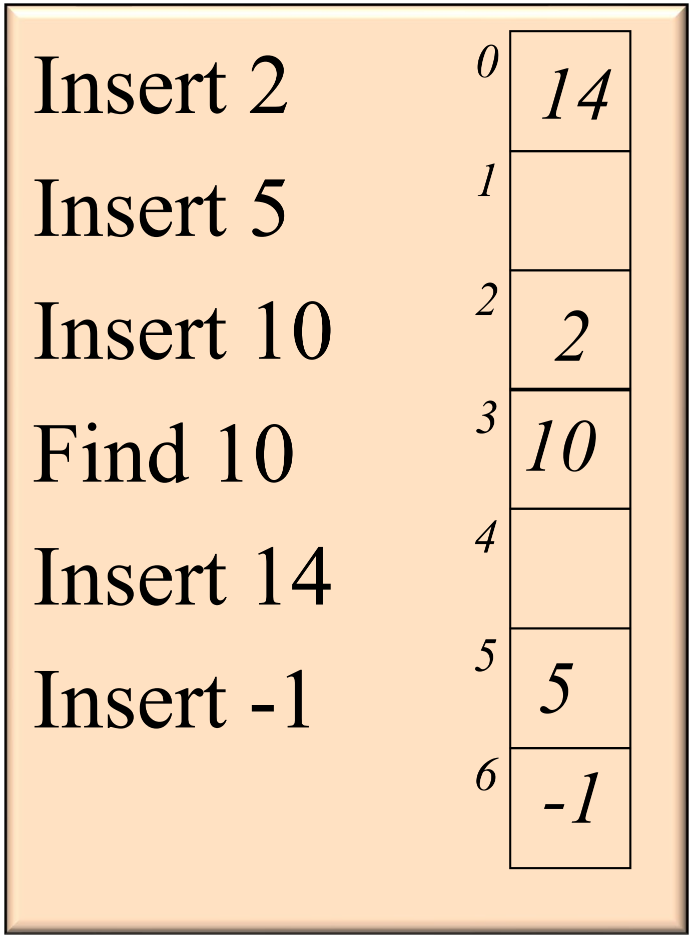
Open Addressing

A Good Hash Function…

- is easy (fast) to compute
  - O(1) *and* fast in practice.
- uses the whole hash table. for all $0 \leq k < size$, there’s an i such that
  - hash(i) % size = k.
- distributes the data evenly as much as possible

Good Hash Function for Integers

- Choose
  - tableSize is
    - **prime** for good spread
    - **power of two** for fast calculations/convenient size
  - hash(n) = n (fast and good enough?)



Insert 2

Insert 5

Insert 10

Find 10

Insert 14

Insert -1

0	*14*
1	
2	*2*
3	*10*
4	
5	*5*
6	*-1*

Good Hash Function for Strings?

Suppose we have a table capable of holding 5000 records, and whose keys consist of strings that are 6 characters long. We can apply numeric operations to the ASCII codes of the characters in the string in order to determine a hash index:

```
def hash(key):
   hashCode =0
   for str in key:
               # ord turns char to ASCII code
      hashCode = hashCode + ord(str)

   return hashCode % 5000
```

C	A	M	E	C	O	/0

$C = 67$

$A = 65$

$M = 77$

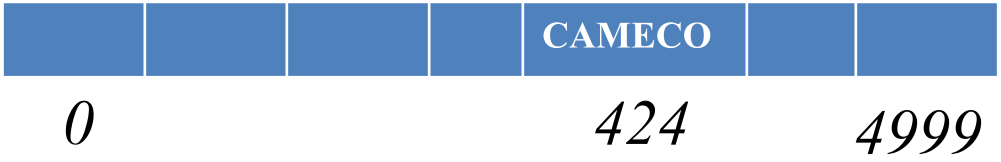
$E = 69$

$C = 67$

$O = 79$

$= 424$

Is this a good idea?



The ASCII table has 128 characters

Good Hash Function for Strings?

- What is a significant problem with this approach?
  - Hash of any string with the same 6 letters is the same
  - The ASCII table has 128 characters
    - 6*128= 768, which means [769– 4999] are wasted
- Alternative approach
- Let s = $s_1 s_2 s_3 s_4 \ldots s_5$: choose
  - – hash(s) = $s_1 + s_2 128 + s_3 128^2 + s_4 128^3 + \ldots + s_n 128^n$
- Problems:
  - – hash("really, really big") is really, really big!
  - – hash("one thing") % 128 = hash("other thing") % 128

Improved Hash Function for Strings

```
def hash2(key):
    hashCode =0
    index = 0
    for str in key:
        # ord turns char to ASCII code
        hashCode = hashCode + pow(128,index)*ord(str)
        index = index +1

    return hashCode % 5000
```

Hash Function Summary

- Goals of a hash function
  - reproducible mapping from key to table entry
  - evenly distribute keys across the table
  - separate commonly occurring keys (neighbouring keys?)
  - complete quickly

How to Design a Hash Function

- Know what your keys are *or*
  Study how your keys are distributed.
- Try to include all important information in a key in the construction of its hash.
- Try to make “neighbouring” keys hash to very different places.
- Balance complexity/runtime of the hash function against spread of keys (very application dependent).

The Pigeonhole Principle (informal)

You can't put k+1 pigeons into k holes without putting two pigeons in the same hole.

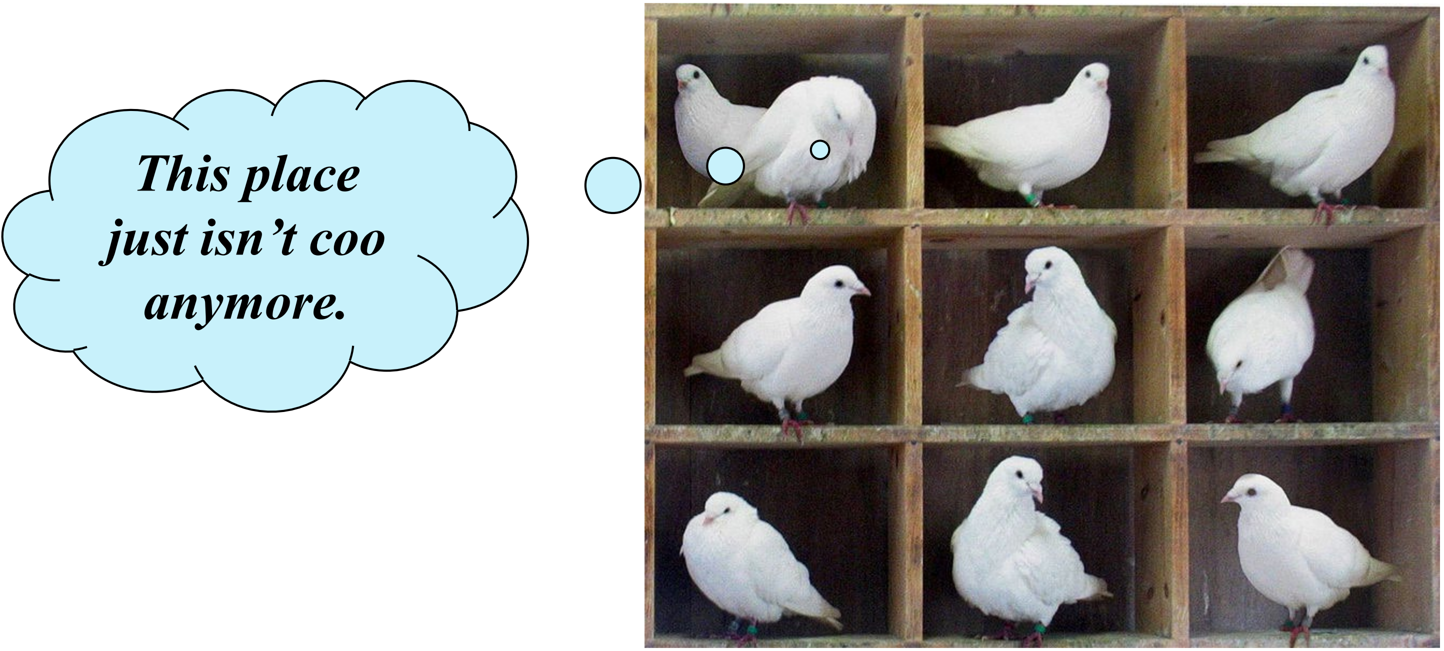


Image by [en:User:McKay](), used under CC attr/share-alike.

Clicker question

Suppose we have 5 colours of Halloween candy, and that there's lots of candy in a bag. How many pieces of candy do we have to pull out of the bag if we want to be sure to get 2 of the same colour?

a. 2
b. 4
c. 6
d. 8
e. None of these

Clicker question (answer)

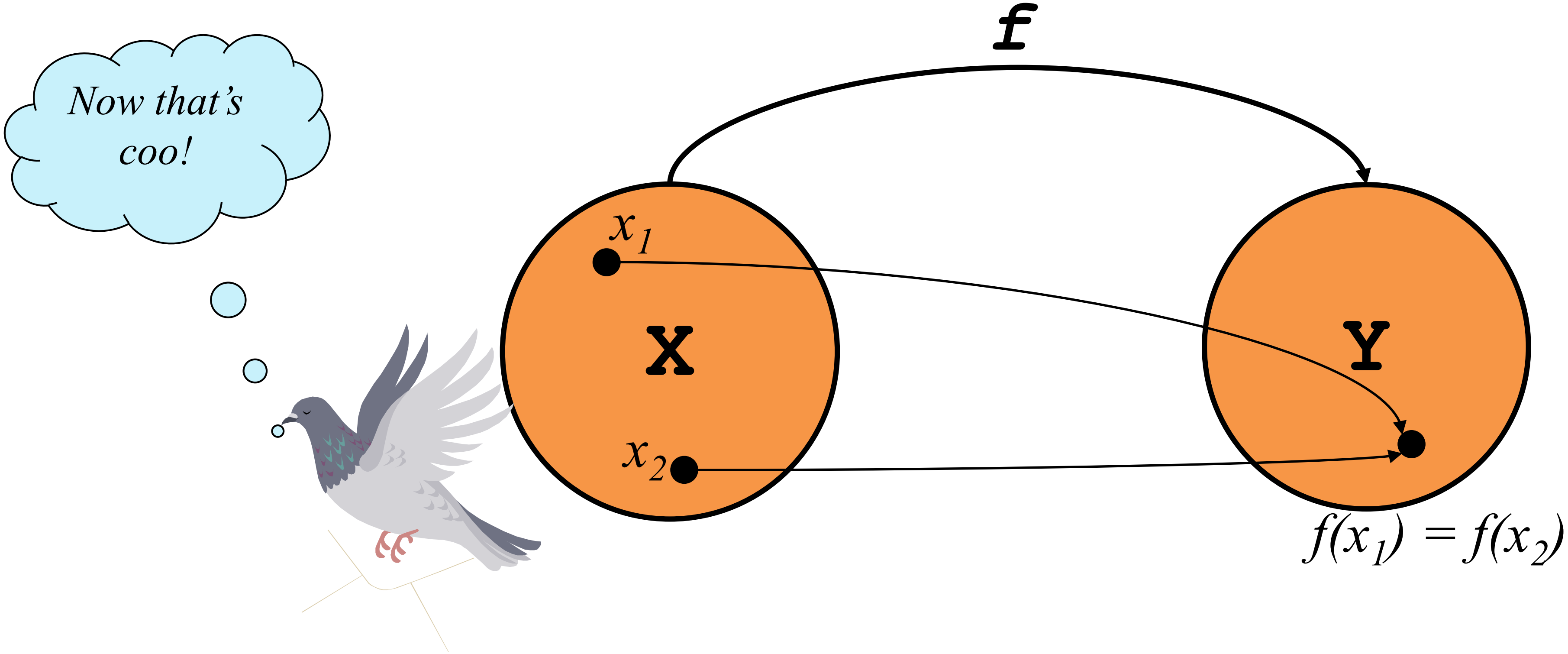
Suppose we have 5 colours of Halloween candy, and that there's lots of candy in a bag. How many pieces of candy do we have to pull out of the bag if we want to be sure to get 2 of the same colour?

a. 2
b. 4
c. 6
d. 8
e. None of these

The Pigeonhole Principle (formal)

Let X and Y be finite sets where $|X| > |Y|$.

If $f : X \rightarrow Y$, then $f(x_1) = f(x_2)$ for some $x_1, x_2 \in X$, where $x_1 \neq x_2$.



The Pigeonhole Principle (Example #2)

If there are 1000 pieces of each colour, how many do we need to pull to guarantee that we'll get 2 *black* pieces of candy (assuming that black is one of the 5 colours)?

a. 2

b. 6

c. 4002

d. 5001

e. None of these

The Pigeonhole Principle (Example #2)

If there are 1000 pieces of each colour, how many do we need to pull to guarantee that we'll get 2 *black* pieces of candy (assuming that black is one of the 5 colours)?

a. 2

b. 6

c. 4002

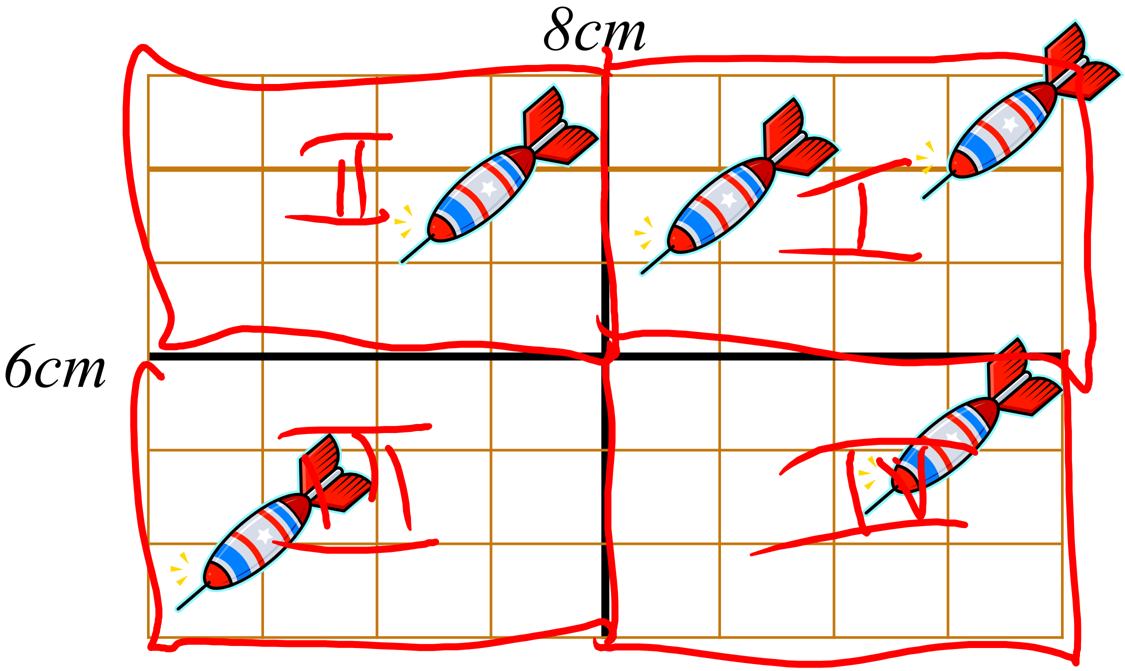
d. 5001

e. None of these

This is not an appropriate problem for the pigeonhole principle! We don't know **which** hole has two pigeons!

The Pigeonhole Principle (Example #3)

If 5 points are placed in a 6cm x 8cm rectangle, argue that there are two points that are not more than 5 cm apart.



Hint: How long is the diagonal?

Example revisited

- In a small company of 100 employees, each employee is assigned an Emp_ID number in the range 00000 - 99999.
  - U (number of potential keys)=100,000
  - m (number of keys) = 100
  - n (space allocated) =?
    - Hopefully not much bigger than m
    - Maybe 200 or 300
- By the Pigeonhole Principle(PHP) multiple potential keys are mapped to the same slot, which introduces the possibility of collisions.
- As m gets larger there is a higher probability of collision.

Clicker question

- Consider n people with random birthdays (i.e., with each day of the year equally likely). How large does n need to be before there is at least a 50% chance that two people have the same birthday.

A: 23

B: 57

C: 184

D: 367

E: None of the above

Clicker question (Birthday Paradox)

- Consider n people with random birthdays. How large does n need to be before there is at least a 50% chance that two people have the same birthday.

A: 23 → 50%

B: 57 → 99%

C: 184

D: 367 → 100%

E: None of the above

- Corollary: Even if we randomly hash only $\sqrt{2m}$ keys into m slots, we get a collision with probability > **0.5**.

Pathological Data Sets

- For good hash performance, we need a good hash function
  - Spreads data evenly across buckets
- Ideal: Use super-clever hash function guaranteed to spread every data set out evenly
- Problem: Such a hash function does not exist
  - For every hash function, there is a pathological data set

Pathological Data Sets

- Reason
  - Fix a hash function h
  - Let U be the potential number of keys
  - Let m be the table size
- There exists an array cell i, such that at least U/m elements hash to i under h
- If data set drawn only from these elements, then everything collides.
- This data set could be quite large since $U >> m$

Overview of Universal Hashing

- For every deterministic hash function, there is a pathological data set.
  - Solution: Do not commit to a specific hash function

- Use randomization
  - Design a family H of hash functions, such that for every data set S, most functions $h \in H$ spread S out "pretty evenly"

Collision Resolution

- What do we do when two keys hash to the same entry?
  - chaining: put little dictionaries in each entry

shove extra pigeons in one hole!

  - open addressing: pick a next entry to try

High-level view of Hashing

Collision and Hash Functions

Chaining

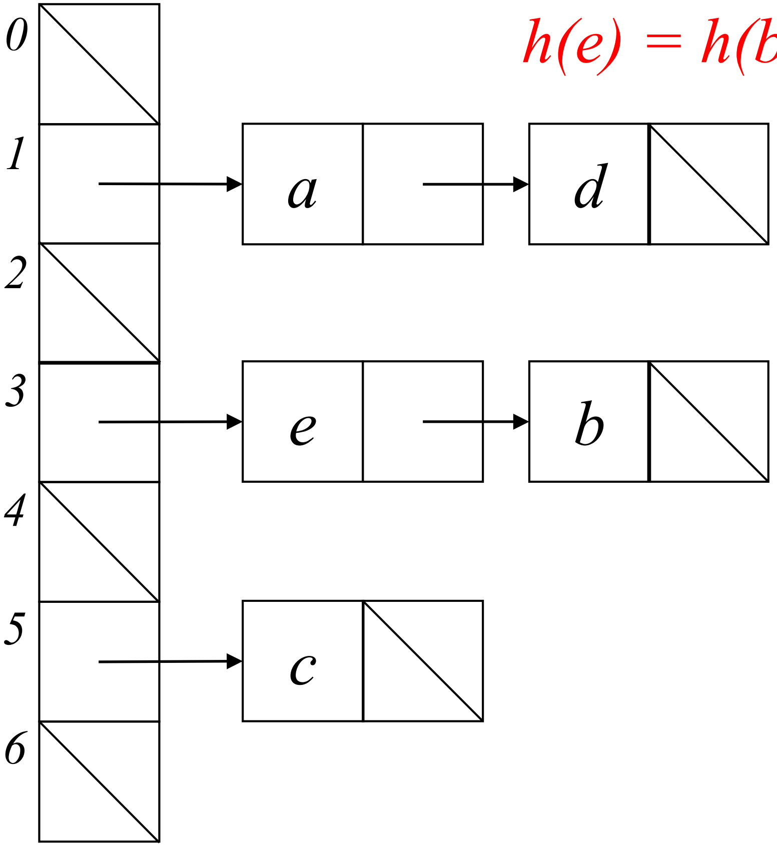
Open Addressing

Hashing with Chaining

- Put a little dictionary at each entry
  - choose type as appropriate
  - common case is unordered move-to-front linked list (chain)
- Properties
  - λ can be greater than 1
  - performance degrades with length of chains

$h(a) = h(d)$

$h(e) = h(b)$

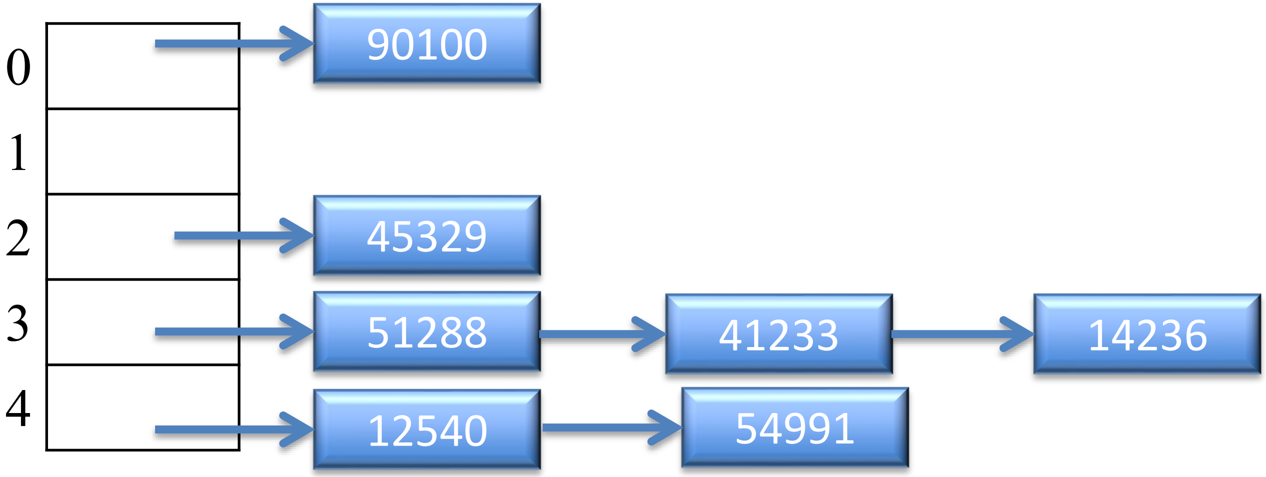


$$\text{load factor } \lambda = \frac{\#\text{ of entries in table}}{\text{tableSize}}$$

In-class exercise

Example: Suppose $h(x) = \lfloor x/10 \rfloor \bmod 5$

Hash: 12540, 51288, 90100, 41233, 54991, 45329, 14236

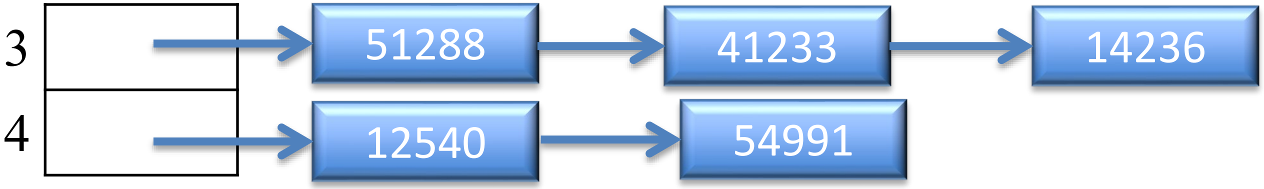


Example: find node with key 14236

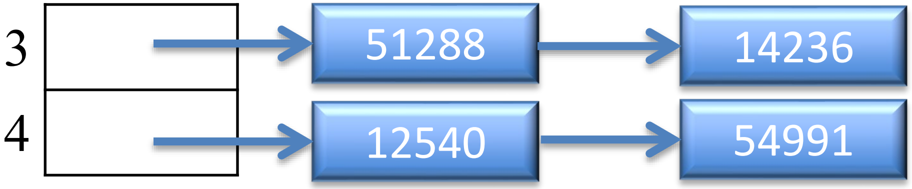
Deleting when using chaining

Example: Suppose $h(x) = \lfloor x/10 \rfloor \bmod 5$

Hash: 12540, 51288, 41233, 54991, 14236



- Delete 41233
- Remove 41233 from the linked list



Load Factor in Chaining

$$\textit{load factor } \lambda = \frac{\#\textit{ of entries in table}}{\textit{tableSize}}$$

- Search cost
  - unsuccessful search:
    - On average λ
  - successful search:
    - On average ~ λ/2
- Desired load factor:
  - between 1/2 and 1.

Pros and cons of chaining

Advantages of Chaining:

- The size s of the hash table can be smaller than the number of items n hashed. Why is this often a good thing?
    - Fewer blank/wasted cells (especially in the case where the number of cells greatly exceeds the number of keys).
    - Collision handling can be O(1).
    - Can accommodate overflows

Disadvantages of Chaining:

- Search time can become O(n) due to long chains.

High-level view of Hashing

Collision and Hash Functions

Chaining

Open Addressing

Open Addressing

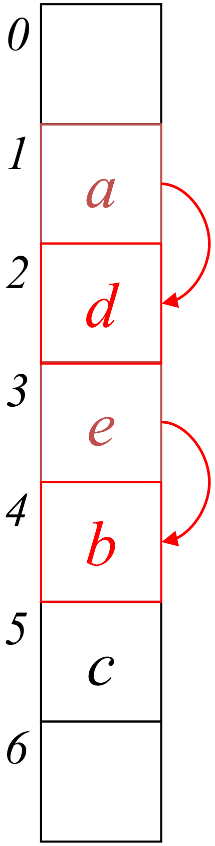
What if we only allow one Key at each entry?

- two objects that hash to the same spot can't both go there
- first one there gets the spot
- next one must *go in another spot*

- Properties
  - $\lambda \leq 1$
  - performance degrades with difficulty of finding right spot

$h(a) = h(d)$

$h(e) = h(b)$



$$load\ factor\ \lambda = \frac{\#\ of\ entries\ in\ table}{tableSize}$$

Probing

- Probing how to:
  - First probe - given a key k, hash to h(k)
  - Second probe - if h(k) is occupied, try h(k) + f(1)
  - Third probe - if h(k) + f(1) is occupied, try h(k) + f(2)
  - And so forth
- Probing properties
  - the i^{th} probe is to (h(k) + f(i)) mod size where f(0) = 0
  - if i reaches size, the insert has failed
  - depending on f(), the insert may fail sooner
  - long sequences of probes are costly!

f() will be defined later.

Linear Probing, f(i) = i

- Probe sequence is
    - h(k) mod size
    - h(k) + 1 mod size
    - h(k) + 2 mod size
    - …
- findEntry using linear probing:

```
def findEntryLinear(self, key):
    probpoint = hash(key)
        for i in range self.size:
            if self.slots[probpoint] = key:
                return self.data[probpoint]
            else:
                probpoint = (probpoint + 1)%size
    return None
```

In-class exercise

- Using the hash function *h*(*x*) = x % 7 insert the following values using **linear probing**: *76, 93, 40, 47, 10, 55*

	insert(76)	*insert(93)*	*insert(40)*	*insert(47)*	*insert(10)*	*insert(55)*
	76%7 = 6	*93%7 = 2*	*40%7 = 5*	*47%7 = 5*	*10%7 = 3*	*55%7 = 6*
0				*47*	*47*	*47*
1						*55*
2		*93*	*93*	*93*	*93*	*93*
3					*10*	*10*
4						
5			*40*	*40*	*40*	*40*
6	*76*	*76*	*76*	*76*	*76*	*76*
probes:	*1*	*1*	*1*	*3*	*1*	*3*

Load Factor in Linear Probing

$$\textit{load factor } \lambda = \frac{\textit{\# of entries in table}}{\textit{tableSize}}$$

- For *any* λ < 1, linear probing will find an empty slot
- Search cost (for large table sizes)
  - successful search: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$

	λ=0.25	λ=0.5	λ=0.75	λ=0.9
Avg # slots searched	1.17	1.5	2.5	5.5

  - unsuccessful search: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$
    - How performance degrades as λ gets bigger

	λ=0.25	λ=0.5	λ=0.75	λ=0.9
Avg # slots searched	1.4	2.5	8.5	50.5

Load Factor in Linear Probing

$$\text{load factor } \lambda = \frac{\#\text{ of entries in table}}{\text{tableSize}}$$

Values hashed close to each other probe the same slots.

- Linear probing suffers from ***primary clustering***
- Performance quickly degrades for λ > 1/2

Quadratic Probing, f(i) = i^2

- Probe sequence is
  - h(k) mod size
  - (h(k) + 1) mod size
  - (h(k) + 4) mod size
  - (h(k) + 9) mod size
  - …
- findEntry using quadratic probing:

```
def findEntryQuadratic(self, key):
    probpoint = hash(key)
        for i in range self.size:
            if self.slots[probpoint] = key:
                return self.data[probpoint]
            else:
                probpoint = (probpoint + i*i)%size
    return None
```

Quadratic Probing Example ☺

- Using the hash function *h*(*x*) = x % 7 insert the following values using **quadratic probing**: *76, 40, 48, 5, 55*

	insert(76) *76%7 = 6*	*insert(40)* *40%7 = 5*	*insert(48)* *48%7 = 6*	*insert(5)* *5%7 = 5*	*insert(55)* *55%7 = 6*
0			*48*	*48*	*48*
1					
2				*5*	*5*
3					*55*
4					
5		*40*	*40*	*40*	*40*
6	*76*	*76*	*76*	*76*	*76*
probes:	*1*	*1*	*2*	*3*	*3*

Quadratic Probing Example ☹

- Using the hash function *h*(*x*) = x % 7 insert the following values using quadratic probing: *76, 93, 40, 35, 47*

	insert(76)	*insert(93)*	*insert(40)*	*insert(35)*	*insert(47)*
	76%7 = 6	*93%7 = 2*	*40%7 = 5*	*35%7 = 0*	*47%7 = 5*
0				*35*	*35*
1					
2		*93*	*93*	*93*	*93*
3					
4					
5			*40*	*40*	*40*
6	*76*	*76*	*76*	*76*	*76*
probes:	*1*	*1*	*1*	*1*	*∞*

Load Factor in Quadratic Probing

- For *any* $\lambda \leq \frac{1}{2}$, quadratic probing will find an empty slot; for greater λ, quadratic probing *may* find a slot
- Quadratic probing does not suffer from primary clustering
- Quadratic probing *does* suffer from ***secondary clustering***
  - How could we possibly solve this?

Values hashed to the SAME index probe the same slots.

Double Hashing, f(i) = i . hash2(k)

- Probe sequence is
  - $h_1(k)$ mod size
  - $(h_1(k) + 1 \cdot h_2(k))$ mod size
  - $(h_1(k) + 2 \cdot h_2(k))$ mod size
  - …

```
def findEntryDoubleHashing(self, key):
    probpoint = hash(key)
    hashInc = hash2(key)
        for i in range self.size:
            if self.slots[probpoint] = key:
                return self.data[probpoint]
            else:
                probpoint = (probpoint + hashInc)%size
return None
```

A Good Double Hash Function…

- is quick to evaluate.
- differs from the original hash function.
- never evaluates to 0 (mod size).

- One good choice is to choose a prime R < size
  - $hash_2(k) = R - (k \bmod R)$

Double Hashing Example

- Using the hash functions $h_1(x) = x \% 7$ and $h_2(x)=5 - (x \% 5)$ insert the following values using **double hashing** *76, 93, 40, 47, 10, 55*

	insert(76)	*insert(93)*	*insert(40)*	*insert(47)*	*insert(10)*	*insert(55)*
	76%7 = 6	*93%7 = 2*	*40%7 = 5*	*47%7 = 5*	*10%7 = 3*	*55%7 = 6*
				5 - (47%5) = 3		*5 - (55%5) = 5*
0						
1				*47*	*47*	*47*
2		*93*	*93*	*93*	*93*	*93*
3					*10*	*10*
4						*55*
5			*40*	*40*	*40*	*40*
6	*76*	*76*	*76*	*76*	*76*	*76*
probes:	*1*	*1*	*1*	*2*	*1*	*2*

Clicker question

The primary hash function is: $\mathbf{h_1(k) = (2k + 5) \bmod 11}$.
The secondary hash function is: $\mathbf{h_2(k) = 7 - (k \bmod 7)}$

Hash these keys, in this order: *12, 44, 13, 88, 23, 94, 11.*
Which cell in the array does key 11 hash to?

A. 0
B. 2
C. 3
D. 4
E. 10

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Clicker question (answer)

$h_1(k) = (2k + 5) \bmod 11$. $h_2(k) = 7 – (k \bmod 7)$

12, 44, 13, 88, 23, 94, 11. Which cell in the array does key 11 hash to?

$h(12) = (2(12) + 5) \% 11 = 7$

$h(44) = (2(44) + 5) \% 11 = 5$

$h(13) = (2(13) + 5) \% 11 = 9$

$h(88) = (2(88) + 5) \% 11 = 5 + 7 – 88\%7 = 8$

$h(23) = (2(23) + 5) \% 11 = 7 + 7 – 23\%7 = 12$

$h(94) = (2(94) + 5) \% 11 = 6$

$h(11) = (2(11) + 5) \% 11 = 5 + 2(7 – 11\%7) = 11$

A. 0
B. 2
C. 3
D. 4
E. 10

0	*11*
1	*23*
2	
3	
4	
5	*44*
6	*94*
7	*12*
8	*88*
9	*13*
10	

Load Factor in Double Hashing

- For *any* $\lambda < 1$, double hashing will find an empty slot (given appropriate table size and hash_2)
- Search cost appears to approach optimal (random hash):
  - successful search:

$$\frac{1}{\lambda}\ln\frac{1}{1-\lambda}$$

	λ=0.25	λ=0.5	λ=0.75	λ=0.9
Avg # slots searched	1.5	1.4	1.8	2.6

  - unsuccessful search:

$$\frac{1}{1-\lambda}$$

	λ=0.25	λ=0.5	λ=0.75	λ=0.9
Avg # slots searched	1.3	2	4	10

- No primary clustering and no secondary clustering
- One extra hash calculation

The Squished Pigeon Principle

- An insert using open addressing *cannot* work with a load factor of 1 or more.
- An insert using open addressing with quadratic probing may not work with a load factor of ½ or more.
- Whether you use chaining or open addressing, large load factors lead to poor performance!
- How can we relieve the pressure on the pigeons?

Hint: think resizable arrays!

Rehashing

- When the load factor gets “too large” (over a constant threshold on λ), rehash all the elements into a new, larger table:
  - takes O(n), but amortized O(1) as long as we (just about) double table size on the resize
  - spreads keys back out, may drastically improve performance
  - gives us a chance to retune parameterized hash functions
  - avoids failure for open addressing techniques
  - allows arbitrarily large tables starting from a small table
  - clears out lazily deleted items

Practice: Open Addressing

(Try linear, quadratic and double hashing.)

h1(x) = x % 7 and h2(x)= 5 - (x % 5)

Insert 2
Insert 5
Insert 4
Insert 10
Insert 73
Find 10
Insert 14
Resize/Rehash
Insert -1
Insert 3

0	
1	
2	
3	
4	
5	
6	

Learning Outcomes Revisited1

Description	Tag
Demonstrate how hash tables can be used for implementation of a dictionary ADT.	Hashing
Describe characteristics of a good hash function	
Explain and apply the pigeonhole principle.	
Explain collision in hashing.	
Describe the properties of chaining for hashing.	Chaining
insert, delete, and find using chaining algorithms.	
Describe the properties of open addressing.	Open-addressing
Insert, delete and find using linear probing.	
Insert, delete and find using quadratic probing.	
Insert, delete and find using double hashing.	
Explain the notion of primary clustering in hashing.	
Explain the notion of secondary clustering in hashing.	
Compare and contrast different hashing algorithms.	Hashing
Reflect on how hashing can be used in DBMS.	