CSc 120
Introduction to Computer Programming II

11: Hashing

Hashing

Searching

We have seen two search algorithms:

- linear (sequential) search O(n)
  - the items are not sorted
- binary search O(log n)
  - the items are sorted
  - must consider the cost of sorting

- Can we do better?
- Have you considered how a Python dictionary might be implemented?
- Let's think about writing a Dictionary ADT

ADT - Dictionary

- A dictionary is an ADT that holds key/value pairs and provides the following operations:
  - put(key, value)
    - makes an entry for a key/value pair
    - same as d[key] = value
    - simplification: assumes key is not already in the dictionary
  - get(key) looks up key in the dictionary
    - returns the value associated with key (and None if not found)
    - same as d[key]

EXERCISE-ICA-32 p.4

Problem:

Implement the Dictionary ADT

Simplification: dictionary is a fixed sized, specified when created

Usage:

```
>>> d = Dictionary(7)
>>>
>>> d.put('five', 5)
>>> d.put('three', 3)
```

Hint:

```
>>> d._pairs
[['five', 5], ['three', 3], None, None, None, None, None]
```

*The class must have an attribute for the next ”unused” slot

ADT – Dictionary solution 1

```
class Dictionary:
    def __init__(self,capacity):
        # each element will be a key/value pair
        self._pairs = [None] * capacity
        self._nextempty = 0

    def put(self, k, v):
        self._pairs[self._nextempty] = [k,v]
        self._nextempty += 1

    def get(self, k):
        for pair in self._pairs[0:self._nextempty]:
            if pair[0] == k:
                return pair[1]
        return None
```

Performance

- What is big-O of the Dictionary's methods?
  - put()
  - get()

  get('three’)

  [['five', 5], ['three', 3], None, None, None, None, None]

Performance

- What is big-O of the Dictionary's methods?
  - put()   O(1)
  - get()   O(n)

  get('three’)

  [['five', 5], ['three', 3], None, None, None, None, None]

Performance

- What is big-O of the Dictionary's methods?
  - put() O(1)
  - get() O(n)

  get('three’)

  [['five', 5], ['three', 3], None, None, None, None, None]

- This is no better than linear search. Can we do better than O(n) for get()? If so, how?
- Consider indexing into a Python list:

  alist[3] # this "get" or "lookup" is O(1)

- Why is this O(1)?

  values in a Python list are contiguous

  location of alist in memory plus an offset of 3

- Can we 'transform' keys into integers that fall into a small, contiguous range?

Beating O(n)

Can we 'transform' keys into integers that fall into a small range?

"hello" -> 147

"a" -> 422 (an arbitrary integer will not do)

How could we turn a key (string) into an integer?

- We need some kind of method/algo
- i.e., perform a computation on the key to get an integer

“Hash” the key (colloquial meaning)

- Chop up/scramble
- i.e., perform a computation on the key to get an integer
- (but a very specific kind of computation…)

Hashing

- A hash function is a function that can be used to map data of arbitrary size (and of various types) to a integer value in a fixed range
- Simple idea for "hashing" a string: use the length (?)
- Is the following a hash function?

```
def hash(key):
    return len(key)
```

- Strings are arbitrary length
  - Modify `hash(key)` to return a value in a fixed range
  - How would we map any string length in to an integer in a fixed range, say, a range of 0 to 6?

EXERCISE-ICA-33 p.1

Problem:

Modify the Dictionary to use a hash function to compute the index for a new key/value pair.

Use the following hash function:

```
def _hash(self, k):
    return len(k) % len(self._pairs)
```

In put() and get(), use this:

```
index = self._hash(k)
```

ADT – Dictionary solution w/hashing

```
class Dictionary:
    def __init__(self, capacity):
      # each element will be a key/value pair
        self._pairs = [None] * capacity

    def _hash(self, k):
        return len(k) % len(self._pairs)

    def put(self, k, v):
        index = self._hash(k)
        self._pairs[index] = [k,v]      #use the hash function

    def get(self, k):
        index = self._hash(k)
        return self._pairs[index][1] #use the hash function
```

Performance

- What is big-O of the Dictionary's methods?
  - put()
  - get()

- put() and get() are each O(1)

- But we have already identified issues with this simple implementation.

Hashing

What happens in this situation?

```
>>> d = Dictionary(7)

>>> d.put('hello', 14)              #  5 % 7 ->  5
>>> d.put('e', 351)                 #  1 % 7 ->  1
>>> d.put('hat', 8)                 #  3 % 7 ->  3
>>> d.put('conciousness', 1)        # 12 % 7 ->  5
```

Hashing

- Hash results:

key	hash value
'hello'	5
'e'	1
'hat'	3
'consciousness'	5

- *Collision*: two or more keys have the same hash value

Hashing

- Hash results:

key	hash value
'hello'	5
'e'	1
'hat'	3
'consciousness'	5

collision

- Dictionary implementation view:

0	1	2	3	4	5	6

['e', 351] ['hat', 8] ['hello', 14]

Need a place to put ['consciousness', 1]

Hashing and collisions

- *perfect hash function*: every key hashes to a unique value
  - most hash functions are not perfect

- Need a systematic method for placing keys in a Dictionary (hash table) when collisions occur.

0	1	2	3	4	5	6

	['e', 351]		['hat', 8]		['hello', 14]	

Need a place to put ['consciousness', 1]

Collision Resolution

- Methods for resolving collisions:
  - increase the table size (the list in our example)

    consider social security numbers: 333-55-8888

    9 digits / 10^9 entries (a billion)
  - open addressing: a method of collision resolution characterized by "probing"
    - on a collision, check other slots in a given order
  - linear probing
    - compute the hash value
    - on collision, start with the hash value
      - visit each slot by going 'lower' in the table (decrement by 1)
      - If empty, use it
      - If not, decrement by 1 again
      - wrap if necessary

Collision Resolution

- Simplify the example by using integers for keys
- Hash function

  h(key) = key % 7

- Hash values for the keys: 14, 2, 10, 19

key	hash value
14	0
2	2
10	3
19	5

- Hash table

0	1	2	3	4	5	6
14		2	10		19	

Collision Resolution

- keys: 14, 2, 10, 19
- Now add 24
  - h(key) = key % 7

    = 24 % 7

    = 3 ← collision, use open addressing
- Hash table

0	1	2	3	4	5	6
14		2	10		19	

h(24) = 3 – collision

Collision Resolution

- keys: 14, 2, 10, 19
- Now add 24
  - h(key) = key % 7

    = 24 % 7

    = 3 ← collision, use open addressing
- Hash table

0	1	2	3	4	5	6
14		2	10		19	

h(24) = 3 – collision

look lower – occupied

Collision Resolution

- keys: 14, 2, 10, 19
- Now add 24
  - h(key) = key % 7

    = 24 % 7

    = 3 ← collision, use open addressing
- Hash table

0	1	2	3	4	5	6
14		2	10		19	

h(24) = 3 – collision

look lower – occupied

look lower – empty

Collision Resolution

- *Probe sequence*: the locations examined when inserting a new key

  $$h(24) = 3$$

- The hash computation is the first "probe"
- Hash table

0	1	2	3	4	5	6
14		2	10		19	

Collision Resolution

- *Probe sequence*: the locations examined when inserting a new key

  h(24) = 3

- The hash computation is the first "probe"
- Hash table

0	1	2	3	4	5	6
14		2	10		19	

first probe – collision 3

Collision Resolution

- *Probe sequence*: the locations examined when inserting a new key

  h(24) = 3

- The hash computation is the first "probe"
- Hash table

0	1	2	3	4	5	6
14		2	10		19	

first probe – collision 3

second probe – occupied 2

Collision Resolution

- *Probe sequence*: the locations examined when inserting a new key

  h(24) = 3

- The hash computation is the first "probe"
- Hash table

0	1	2	3	4	5	6
14		2	10		19	

first probe – collision 3

second probe – occupied 2

third probe – empty 1

Collision Resolution

- *Probe sequence*: the locations examined when inserting a new key

  h(24) = 3

- The hash computation is the first "probe"
- Hash table

probe sequence: 3, 2, 1

0	1	2	3	4	5	6
14	**24**	2	10		19	

first probe – collision 3

second probe – occupied 2

third probe – empty 1

EXERCISE-ICA-33 p.2

Use open addressing with linear probing to insert the key 23 into the hash table below. Give the probe sequence.

The hash function is the key % 7

hash table

0	1	2	3	4	5	6
14	24	2	10		19	

- Do problem 3 next!

EXERCISE-ICA-33 p.3

Modify the put() method of the ADT below to implement open addressing with linear probing.

```
class Dictionary:
    def __init__(self, capacity):
        # each element will be a key/value pair
        self._pairs = [None] * capacity

    def _hash(self, k):
        return len(k) % len(self._pairs)

    def put(self, k, v):
        self._pairs[self._hash(k)] = [k,v]     #modify this to use
                                               # linear probing
   .....
```

Fill out the index card!

Write your birthday **month** and **day** on the index card.

If your birthday is April 20^{th}, then write it like this:

April 20

(This is anonymous! Don't write your name on the card.)

ICA-33 p.3 Solution

Modify the put() method of the ADT below to implement open addressing with linear probing.

```
class Dictionary:
...
    def put(self, k, v):
        i = self._hash(k)
        if self._pairs[i] != None:
            # the slot at i is taken – must "probe"
            # in a loop to find the next available slot
            while True:
                i -= 1
                if i < 0:
                    # wrap around to the end
                    i = len(self._pairs) - 1
                if self._pairs[i] == None:      # found an empty slot
                    break
        self._pairs[i] = [k,v]
```

Clusters

- *Cluster*: a sequence of adjacent, occupied entries in a hash table

- problems with open addressing with linear probing
  - colliding keys are inserted into empty locations below the collision location
  - on each collision, a key is added at the edge of a cluster
  - the edge of the cluster keeps growing
  - the edges begin to meet with other clusters
  - these combine to make *primary clusters*

Collision Resolution

open addressing with *linear probing* has serious performance problems

When two keys collide at the same hash value, they will follow the same initial probe sequence

- the probe sequence is linear
- the probe decrement is 1

Is there a better approach?

Hint: change the probe decrement.

Collision Resolution

open addressing

Linear probing uses the same decrement for all keys

- idea: need a probe decrement that is *different* for keys that hash to the same value

simple example

- the use mod for the hash
    - Suppose the hash function is hash(key) = key % 7
    - Ex: hash(19) = 19 % 7
        - 19 // 7 is 2 quotient 2, remainder 5
            - quotient 2
            - remainder 5
- use quotient for the probe
    - note: cannot use 0
- probe decrement function p(key)

  the quotient of key after division by 7 (if the quotient is 0, then 1)

  or

  probe(key) = max(1, key // 7)

called *open addressing with double hashing*

Collision Resolution – double hashing

- functions

  hash(key) = key % 7

  probe(key) = max(1, key // 7)

- values for the keys: 10, 2, 19, 14, 24, 23

key	hash value	probe decrement
10	3	?
2	2	?
19	5	?
14	0	?
24	3	?
23	2	?

Collision Resolution – double hashing

- functions

  hash(key) = key % 7

  probe(key) = max(1, key // 7)
- values for the keys: 10, 2, 19, 14, 24, 23

key	hash value	probe decrement
10	3	1
2	2	1
19	5	2
14	0	2
24	3	3
23	2	3

Collision Resolution – double hashing

key	hash value	probe decrement
10	3	1
2	2	1
19	5	2
14	0	2
24	3	3
23	2	3

Insert these keys into the hash table: 10, 2, 19, 14

0	1	2	3	4	5	6

Collision Resolution – double hashing

key	hash value	probe decrement
10	3	1
2	2	1
19	5	2
14	0	2
24	3	3
23	2	3

Insert 10 into the hash table below:

0	1	2	3	4	5	6

Collision Resolution – double hashing

key	hash value	probe decrement
10	3	1
2	2	1
19	5	2
14	0	2
24	3	3
23	2	3

Now insert 2 into the hash table:

0	1	2	3	4	5	6
			10			

Collision Resolution – double hashing

key	hash value	probe decrement
10	3	1
2	2	1
19	5	2
14	0	2
24	3	3
23	2	3

Now insert 19 into the hash table:

0	1	2	3	4	5	6
		2	10			

Collision Resolution – double hashing

key	hash value	probe decrement
10	3	1
2	2	1
19	5	2
14	0	2
24	3	3
23	2	3

Now insert 14 into the hash table:

0	1	2	3	4	5	6
		2	10		5	

Collision Resolution – double hashing

key	hash value	probe decrement
10	3	1
2	2	1
19	5	2
14	0	2
24	3	3
23	2	3

Now insert key 24:

0	1	2	3	4	5	6
14		2	10		19	

Collision Resolution – double hashing

key	hash value	probe decrement
10	3	1
2	2	1
19	5	2
14	0	2
24	3	3
23	2	3

Now insert key 24:

0	1	2	3	4	5	6
14		2	10		19	

h(24) = 3 collision

What is the decrement?
What is the probe sequence?

Collision Resolution – double hashing

key	hash value	probe decrement
10	3	1
2	2	1
19	5	2
14	0	2
24	3	3
23	2	3

Now insert key 24:

0	1	2	3	4	5	6
14		2	10	24	19	

h(24) = 3 collision

What is the decrement? 3
What is the probe sequence? 3, 0, 4

Collision Resolution – double hashing

key	hash value	probe decrement
10	3	1
2	2	1
19	5	2
14	0	2
24	3	3
23	2	3

Use double hashing to insert key 23:

0	1	2	3	4	5	6
14		2	10	24	19	

Collision Resolution – double hashing

key	hash value	probe decrement
10	3	1
2	2	1
19	5	2
14	0	2
24	3	3
23	2	3

Use double hashing to insert key 23:

0	1	2	3	4	5	6
14		2	10	24	19	23

EXERCISE-ICA-34 p.1 &2

In the ICA, you will do the same exercise with a different set of keys.

Collision Resolution

open addressing with double hashing

- compute the hash value
- on collision, use the probe decrement function to determine what slot to visit next
- wrap if necessary

improvement over linear probing

- when two keys collide, they will more frequently follow different probe sequences when a search is made for an empty location
  - hash(10) = 3 hash(24) = 3
  - probe(10) = 1 probe(24) = 3
- prevents primary clustering

Hash functions and collisions

- Consider an *ideal hash* function h(k)
  - it maps keys to hash values (slots) uniformly and randomly
- Suppose T is a hash table having M table entries from 0 to M-1
- An ideal hash function would imply that any slot from 0 to M -1 is equally likely
- All slots equally likely, implies collisions would be infrequent.
- Is that true?

collision phenomenon

- von Mises Birthday Paradox
  - if there are 23 or more people in a room, there is a > 50% chance that two or more will have the same birthday

collision phenomenon

Ball tossing model

Given

- a table T with 365 slots
  (each one represents a different day of the year)
- toss 23 balls at random into these 365 slots

then

- there is a > 50% chance we will toss 2 or more balls into the same slot

What?

- 23 balls in the table
- the table is only 6.3% full
  23/365 = .063
- and we have a 50% chance of a collision!

collision phenomenon

Ball tossing model

P(n) = probability that tossing n balls into 365 slots has at least one collision

$$P(n) = 1 - \frac{365!}{365^n(365-n)!}.$$

collision phenomenon

P(n) = probability that tossing n balls into 365 slots has at least one collision

n	P(n)
5	0.027
10	0.117
20	0.411
23	0.572
30	0.706
40	0.891
50	0.970
60	0.994
70	0.99915958
80	0.99991433
100	0.99999969

at 23, greater than 50% chance

collision phenomenon

P(n) = probability that tossing n balls into 365 slots has at least one collision

n	P(n)
5	0.027
10	0.117
20	0.411
23	0.572
30	0.706
40	0.891
50	0.970
60	0.994
70	0.99915958
80	0.99991433
100	0.99999969

at 23, greater than 50% chance

Let’s check the birthdays

There are 365 possible birthdays.

How many people are here?

How many collisions did we get?

Collision resolution

A collision resolution algorithm must be guaranteed to check every slot.

linear probing - yes (it sequentially walks through the slots)

double hashing - ?

Does the probe sequence used for double hashing cover the entire table? (I.e., is any slot ever missed?)

Collision resolution – double hashing

key	hash value	probe decrement
10	3	1
2	2	1
19	5	2
14	0	2
24	3	3
23	2	3

Question: Does the probe sequence cover the entire table? ICA-34-prob. 3

0	1	2	3	4	5	6

Use key 24. Show that the probe sequence visits each slot. (Keep wrapping.)

Collision resolution

key	hash value	probe decrement
10	3	1
2	2	1
19	5	2
14	0	2
24	3	3
23	2	3

The probe sequence covers every slot.

This is true for every key shown in the table

- *try it for other keys*

Why? The table size M and probe decrement are *relatively prime**. Guarantees that the probe sequence covers the table.

**relatively prime*

– have no common divisors other than 1

Collision resolution

The probe sequence covers every slot.

This is true for every key shown in the table

- *try it for other keys not shown in the slides*
- *any issues?*

Yes. The probe decrement function will give results that are not relatively prime with the table size M:

probe(key) = max(1, key // 7)

Ex:

probe(49) = 7

probe(98) = 14

Modify probe() to reduce the quotient to a range from 0 to M-1, i.e.,

probe(key) = max(1, (key // 7) % 7)

Collision resolution

Two policies

- open addressing
  - with linear probing
  - with double hashing

A third policy

- separate chaining

Collision Resolution

separate chaining

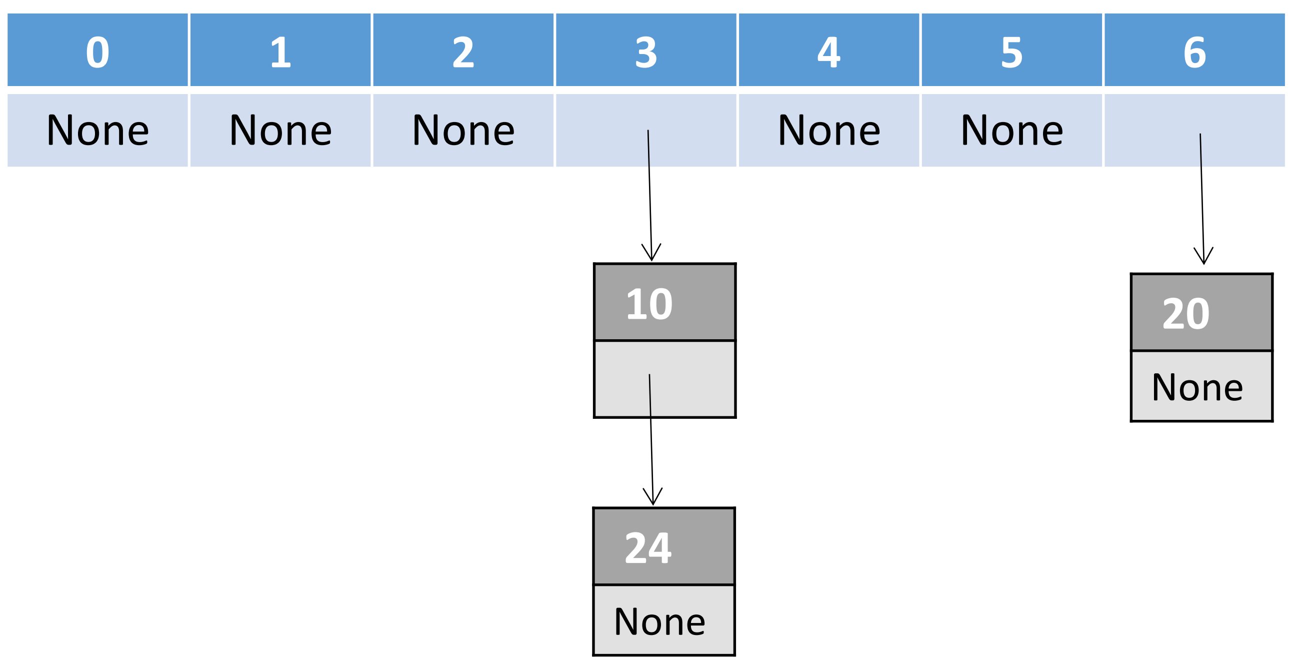
- each table location references a linked list
- on collision, add to the linked list, starting at the collision slot

Collision Resolution

separate chaining

- each table location references a linked list
- on collision, add to the linked list, starting at the collision slot

table with keys 24 and 10 (using %7 for the hash):



EXERCISE ICA-34 prob 4.

Use separate chaining to resolve collisions.

EXERCISE -- Whiteboards

What is a hash function?

What is a collision?

What are the names of the three policies we have covered to handle collisions?

1. ?
2. ?
3. ?

How are collisions handled in each?

EXERCISE -- Whiteboards

What is a hash function?

A function that maps data of arbitrary size (and of various types) to a value in a fixed range

What is a collision?

When two different keys hash to the same value

What are the names of the three policies we have covered to handle collisions?

1. Open addressing w/ Linear probing
2. Open addressing w/ Double hashing
3. Separate chaining

How are collisions handled in each?

1. Probe by decrementing the index by 1
2. Probe by using a probe function (we used quotient)
3. Each slot contains an empty LL. Add to the LL at the slot. On collision, add to the LL at the slot.

EXERCISE ICA-35 prob 1 .

1) Insert the given keys using the hash function provided.

Use linear probing to resolve collisions.

a) Give the probe sequence for inserting:
   - this is the put() method in the ADT

b) Give the probe sequence for lookup or search
   - this is the get() method in the ADT

Complexity

Analysis of separate chaining

If we have N keys, what is

- best case complexity for search:
  (the key is the first item in the linked-list) O(1)
- worst case complexity for search:
  (must exhaustively search one linked-list) O(n)

But whether we need to search depends on if there was a collision. We need to know what happens *on average.*

We will use known results for average case of the collision resolution policies.

Load factor

The load factor of a hash table with N keys and table size M is given by the following:

$$\lambda = N/M$$

load factor is a measure of how full the table is

Complexity is expressed in terms of the load factor.

Complexity

As load factor increases, the efficiency of inserting new keys and then finding those keys decreases

Inserting

- no collision: gets placed at hash value slot
- search for a slot by using the probe decrement
- or insert into the linked list (at the beginning)

Searching

- no collision: find it at the hash value slot
- search by using the probe decrement
- or search the linked list

We will use known results for the average cases of successful and unsuccessful *search* for the collision resolution policies

Assume a table with load factor: $\lambda = N/M$

Linear probing:

clusters form

leads to long probe sequences

It can be shown that the average number of probes is

$$\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$$ for successful search

$$\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$ for unsuccessful search

Bad when load factor is close to 1

Not too bad when load factor is .75 or less

Results

```
>>> # load factor is .75
>>>
>>> # linear probing - successful
>>>
>>> .5 * (1 + 1/.25)
2.5
>>> # linear probing - unsuccessful
>>>
>>> .5 * ( 1 + 1/(.25 *.25))
8.5
```

Assume a table with load factor:

$$\lambda = N/M$$

Double hashing:

clustering less common

It can be shown that the average number of probes is

$$\frac{1}{\lambda}\ln\left(\frac{1}{1-\lambda}\right)$$ for successful search

$$\left(\frac{1}{1-\lambda}\right)$$ for unsuccessful search

Very good when load factor is .75 or less

Results

```
>>> # load factor is .75
>>>
>>> # double hashing - successful
>>>
>>> import math
>>> 1/.75 * math.log(4)
1.8483924814931874
>>>
>>> # double hashing – unsuccessful
>>> 1/.25
4.0
```

Assume a table with load factor: $\lambda = N/M$

Separate chaining:

all keys that collide at a given location are on the same linked list

It can be shown that the average number of probes is

$1 + \frac{1}{2}\lambda$ for successful search

λ for unsuccessful search

Compare the three methods

Theoretical Results (number of probes)

Successful search

Load Factor	0.50	0.75	0.90	0.99
separate chaining	1.25	1.37	1.45	1.49
linear probing	1.50	2.50	5.50	50.5
double hashing	1.39	1.85	2.56	4.65

Unsuccessful search

Load Factor	0.50	0.75	0.90	0.99
separate chaining	0.50	0.75	0.90	0.99
linear probing	2.50	8.50	50.50	5000.00
double hashing	2.00	4.00	10.00	100.00

EXERCISE ICA-35 probs 2- 4

Do the remaining problems.

Hashing Functions

Good performance requires a good hashing function.

- the hash function should not cause clustering
- the hash function should be efficient
  - if too complicated, it takes over the computation and defeats the purpose of hashing

Hash functions typically

- map keys to numbers (if not already numbers)
- then reduce that using mod

Example:

'hello' → len('hello') % 7

Must be aware of properties of the hashing function.

Hashing Functions

Example: hashing function *hash(key,M) where key is a string*

- add the ord values of the characters in a string
- mod by the table size M

For the key 'bat':

- hash('bat', M) = (ord('b') + ord('a') + ord('t')) % M

```
def hash(key, M):
    sum = 0
    for c in key:
        sum += ord(c)
    return sum % M
```

What are the properties of this hash function?

Does it cause clustering?

Hashing Functions

```
def hash(key, M):
    sum = 0
    for c in key:
        sum += ord(c)
    return sum % M
```

Use:

```
>>> hash("bat", 7)
3
>>> hash("tab", 7)
3
>>> hash("atb", 7)
3
>>> hash("tide", 7)
2
>>> hash("tied", 7)
2
```

Hashing Functions

Example: hashing function *hash*

- add the ord values of a string
- mod by the table size M

hash('bat', M) = (ord('b') + ord('a') + ord('t')) % M

hash('tab', M) = (ord('t') + ord('a') + ord('b')) % M

What are the properties of this hash function?

- anagrams hash to the same value

Will that matter?

If it does, how would we fix that?

Hashing Functions

Example: hashing function *hash*

- add the ord values of a string
- mod by the table size M

Modify to multiply by character position, i.e.,

hash('bat', M) = (ord('b')*1 + ord('a')*2 + ord('t')*3) % M

hash('tab', M) = (ord('t')*1 + ord('a')*2 + ord('b')*3) % M

Hashing Functions

Due to properties of integers, there are some pitfalls using modulo:

h(k) = k % M

Example: If we use a power of 2 for M, then

- for $M = 2^{\mathbf{b}}$, $h(k) = k \% 2^b$
- elects the **b** low order bits of **k**

In general, when using modulo

avoid powers of 2

use prime numbers for M

Note:

You don't have to create a hash function for an exam or assignment

Other uses of Hashing

Message digest: ensure the integrity of a message transmitted over an insecure channel

- Given a message (a file), compute its cryptographic hash value
  - Results in a compressed image called Digest

- Send the message and the Digest (hash value)
- Once the message is sent, the receiver checks its hash value
- Must match
- Collisions must be rare

Other uses of Hashing

- SHA-1 (Secure Hash Algorithm 1)
  - cryptographic hash function designed by the NSA
  - takes a file as input
  - Outputs a 160 bit hash value (digest)
  - shown as hexadecimal number, 40 digits long

  https://wingware.com/downloads/wing-101

- SHA-512 (Secure Hash Algorithm 2)
  - a set of cryptographic hash functions designed by the NSA
  - built-on Merkle-Damgard construction

EXERCISE ICA-36 probs 1&2