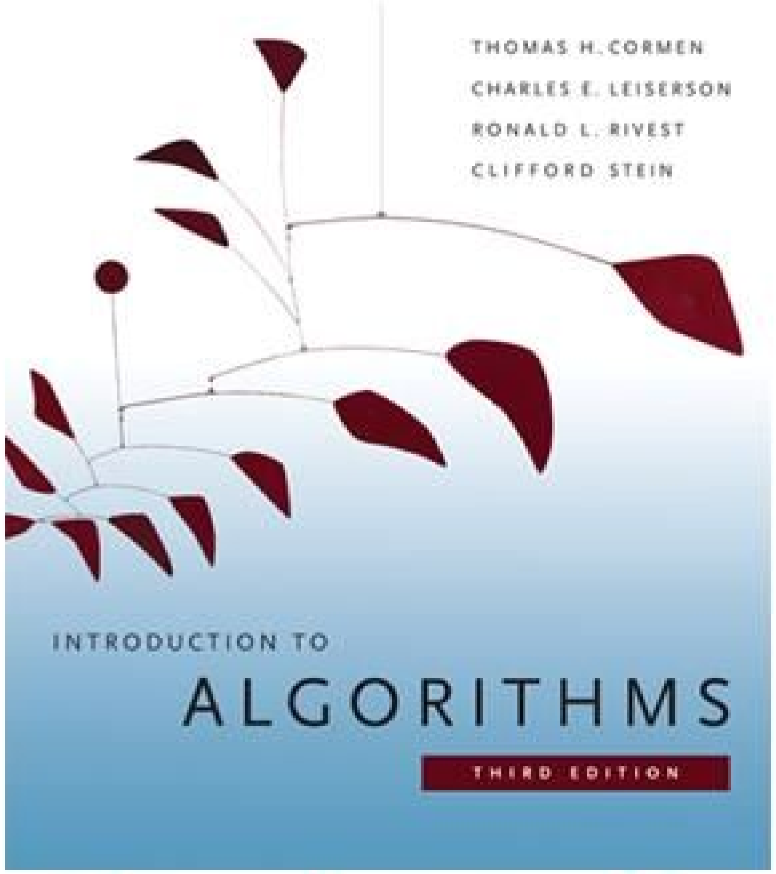
6.006- *Introduction to Algorithms*



Lecture 5

Prof. Constantinos Daskalakis

Today’s Topic

“*Optimist pays off!*”

a.k.a. The ubiquity and usefulness of ***dictionaries***

Dictionaries

- It is a **set** containing **items**; each item has a **key**
- what keys and items are is quite flexible
- Supported Operations:
  - **Insert(*item*):** add given *item* to set
  - **Delete(*item*):** delete given *item* to set
  - **Search(*key*):** return the item corresponding to the given *key*, if such an item exists
- **Assumption**: every item has its own key (or that inserting new item clobbers old
- Application (and origin of name): Dictionaries
  - *Key* is word in English, *item* is word in French

Dictionaries are everywhere

- Spelling correction
  - *Key* is misspelled word, *item* is correct spelling
- Python Interpreter
  - Executing program, see a variable name (*key*)
  - Need to look up its current assignment (*item*)
- Web server
  - Thousands of network connections open
  - When a packet arrives, must give to right process
  - *Key* is source IP address of packet, *item* is handler

Implementation

- use BSTs!
  - can keep keys in a BST, keeping a pointer from each key to its value
  - O(log n) time per operation

- Often not fast enough for these applications!

- Can we beat BSTs?

if only we could do all operations in O(1) ...

[A parenthesis: DNA Matching

Application: DNA matching

- Given two DNA sequences
  - Strings over 4-letter alphabet
- Find largest substring that appears in both
  - Algorithm vs. Arithmetic
  - Algorithm vs. Arithmetic
- Also useful in plagiarism detection

- Say strings S and T of length n

Naïve Algorithm

- For L = n downto 1
- for all length L substrings X1 of S
- for all length L substrings X2 of T
- if X1=X2, return L

- Runtime analysis
  - n candidate lengths
  - n strings of that length in each of X1, X2
  - L time to compare the strings
  - Total runtime: $\Omega(n^4)$

Improvement 1: binary search

- Start with L=n/2
- for all length L substrings X1 of S
- for all length L substrings X2 of T
- if X1=X2, success, try larger L
  if failed, try smaller L

- Runtime analysis
  $\Omega(n^4) \rightarrow \Omega(n^3 \log n)$

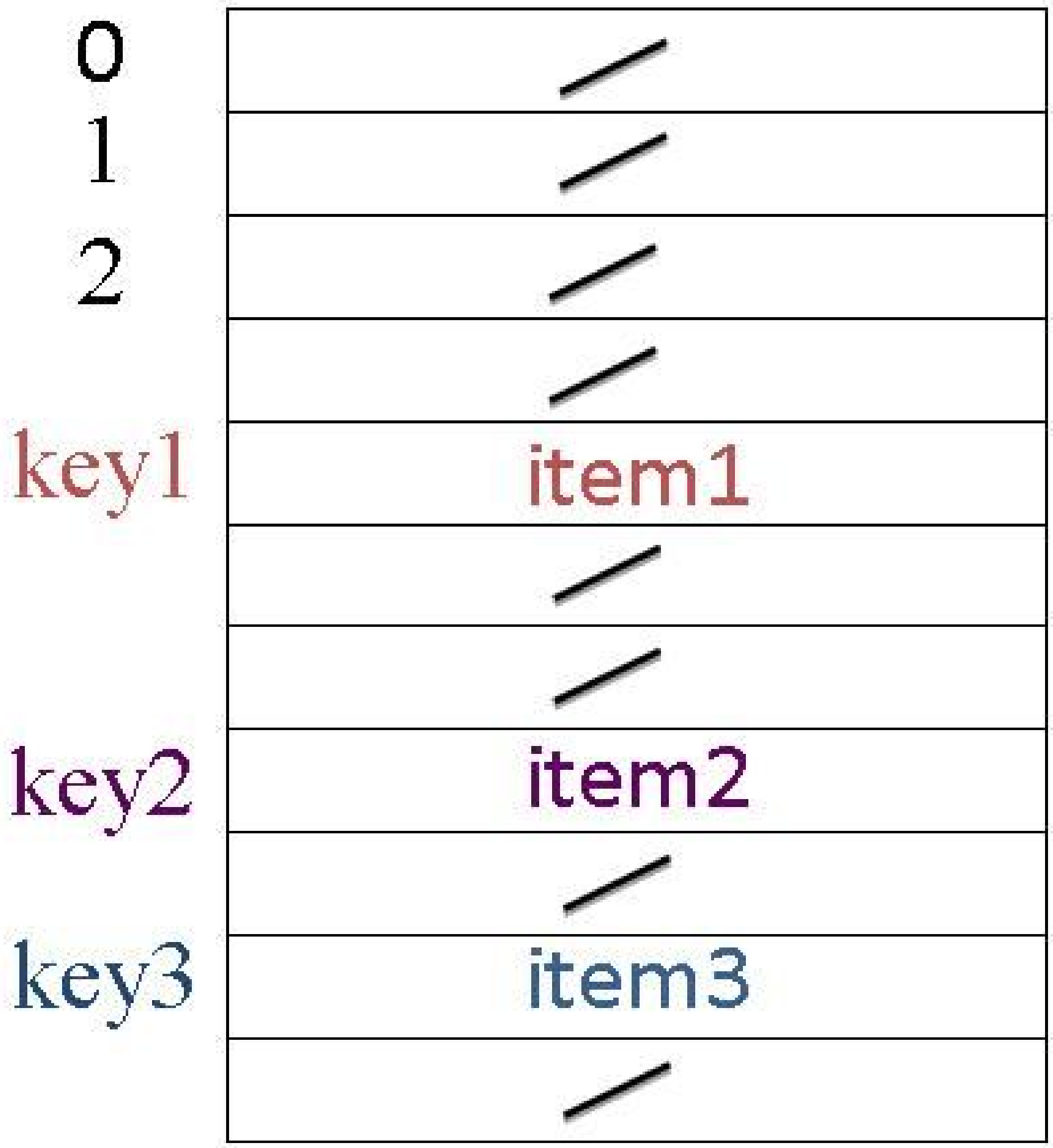
Improvement 2: Dictionary

- For every possible length L=n,…,1
  - Insert all length L substrings of S into a dictionary
  - For each length L substring of T, check if it exists in dictionary

- Possible lengths for outer loop: n
- For each length:
  - at most n substrings of S inserted into dictionary, each insertion takes time O(1) * L (L is paid because we have to read string to insert it)
  - at most n substrings of T checked for existence inside dictionary, each check takes time O(1) * L
  - Overall time spent to deal with a particular length L is O(Ln)
- Hence overall $O(n^3)$
- With binary search on length, total is $O(n^2 \log n)$
- "Rolling hash" dictionaries improve to O(n log n) (next time)

…end of parenthesis]

Dictionaries: Attempt #1



- Forget about BSTs..
- Use table, indexed by keys!

Problems…

- What if keys aren't numbers?

  How can I then index a table?

Problems…

- What if keys aren't numbers?

How can I then index a table?

Pythagoras

Problems…

- What if keys aren't numbers?

"Everything is number."

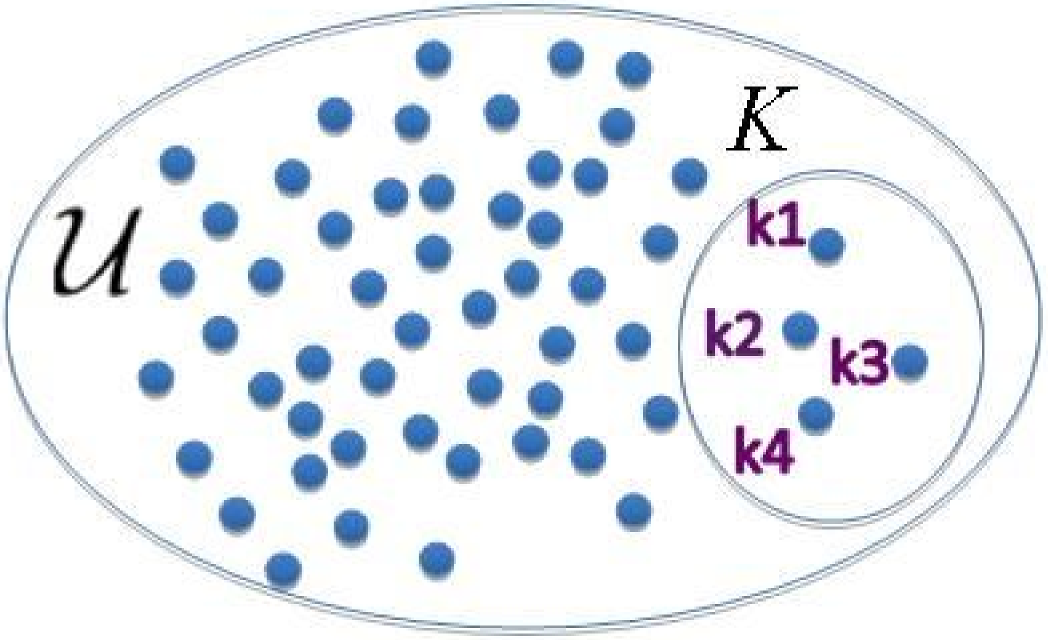
  - Anything in the computer is a sequence of bits
  - So we can pretend it's a number
- Example: English words
  - 26 letters in alphabet

    $\Rightarrow$ can represent each with 5 bits
  - Antidisestablishmentarianism has 28 letters
  - 28*5 = 140 bits
  - So, store in array of size 2^{140} ….oops
- Isn't this too much space for 100,000 words?

Pythagoras

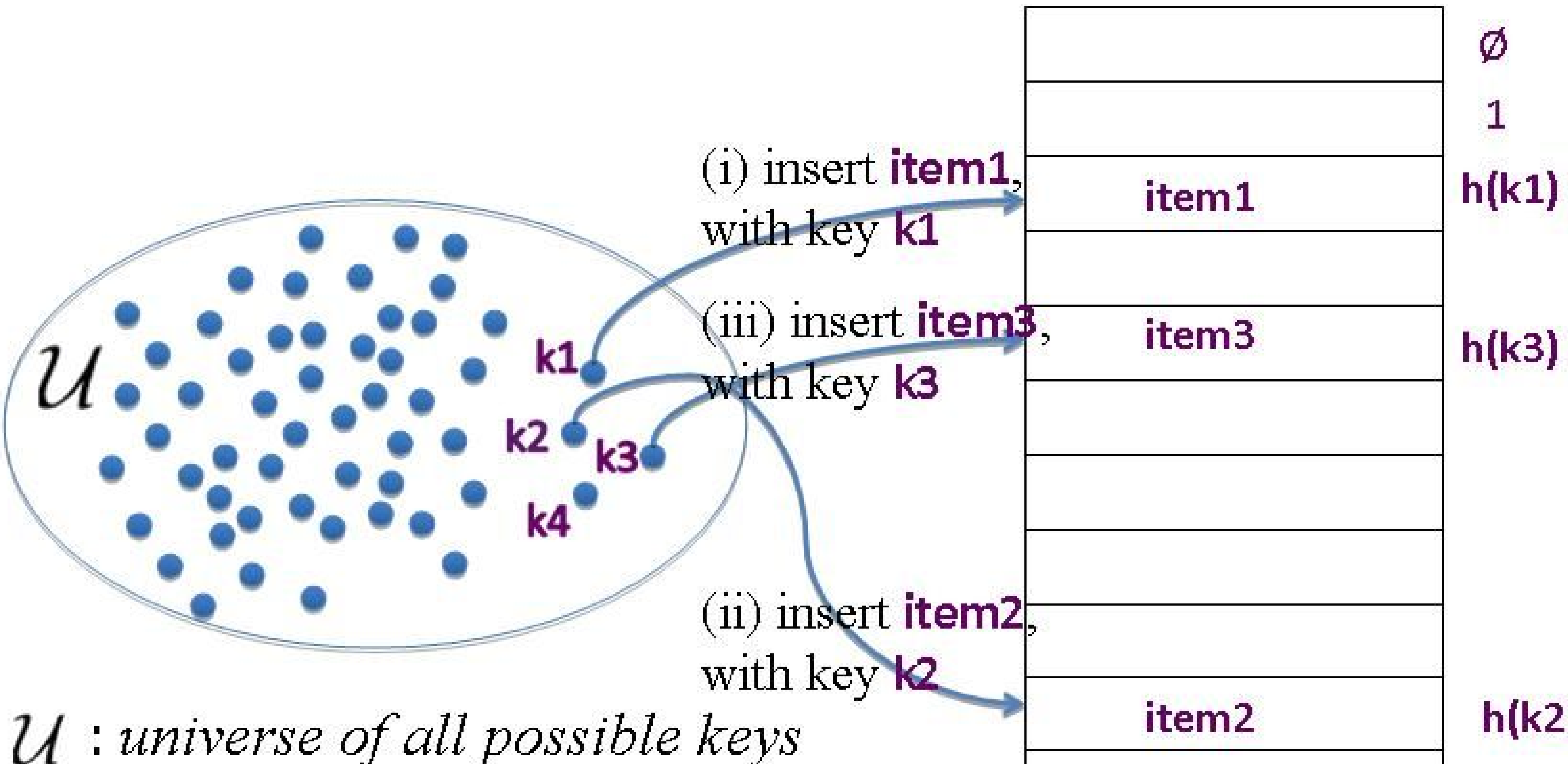
Hash Functions

- Exploit sparsity
  - Huge universe U of possible keys
  - But only n keys actually present
  - Want to store in table (array) of size m~n
- Define hash function $h:U \rightarrow \{1..m\}$
  - Filter key k through h( ) to find table position
  - Table entries are called buckets
- Time to insert/find key is
  - Time to compute h (generally length of key)
  - Plus one time step to look in array

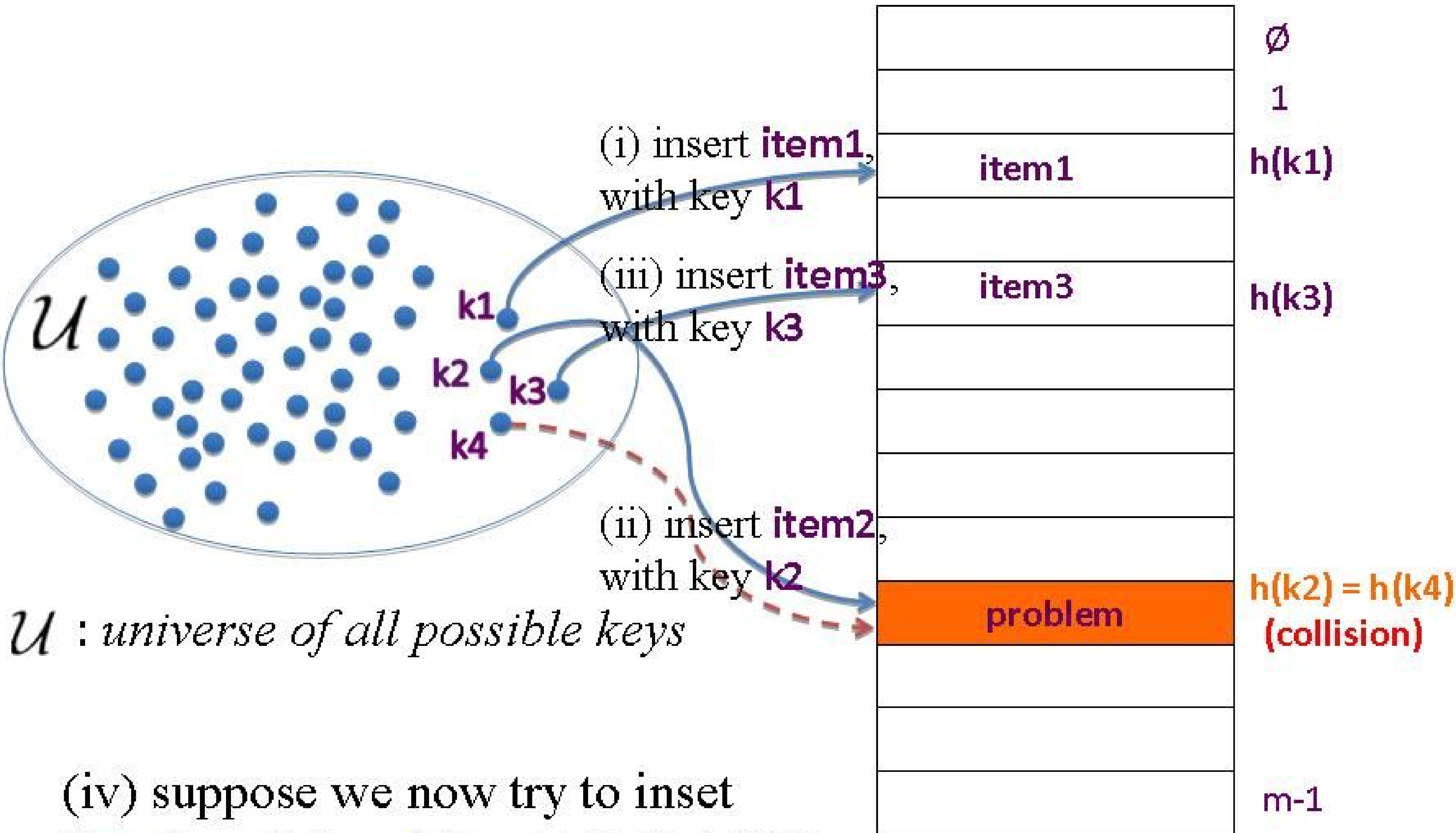


$\mathcal{U}$: *universe of all possible keys; huge set*

K : *actual keys; small set but not known in advance*



(iv) suppose we now try to inset **item4**, with key **k4** and h(k4)=h(k2)…



(iv) suppose we now try to inset **item4**, with key **k4** and h(k4)=h(k2)…

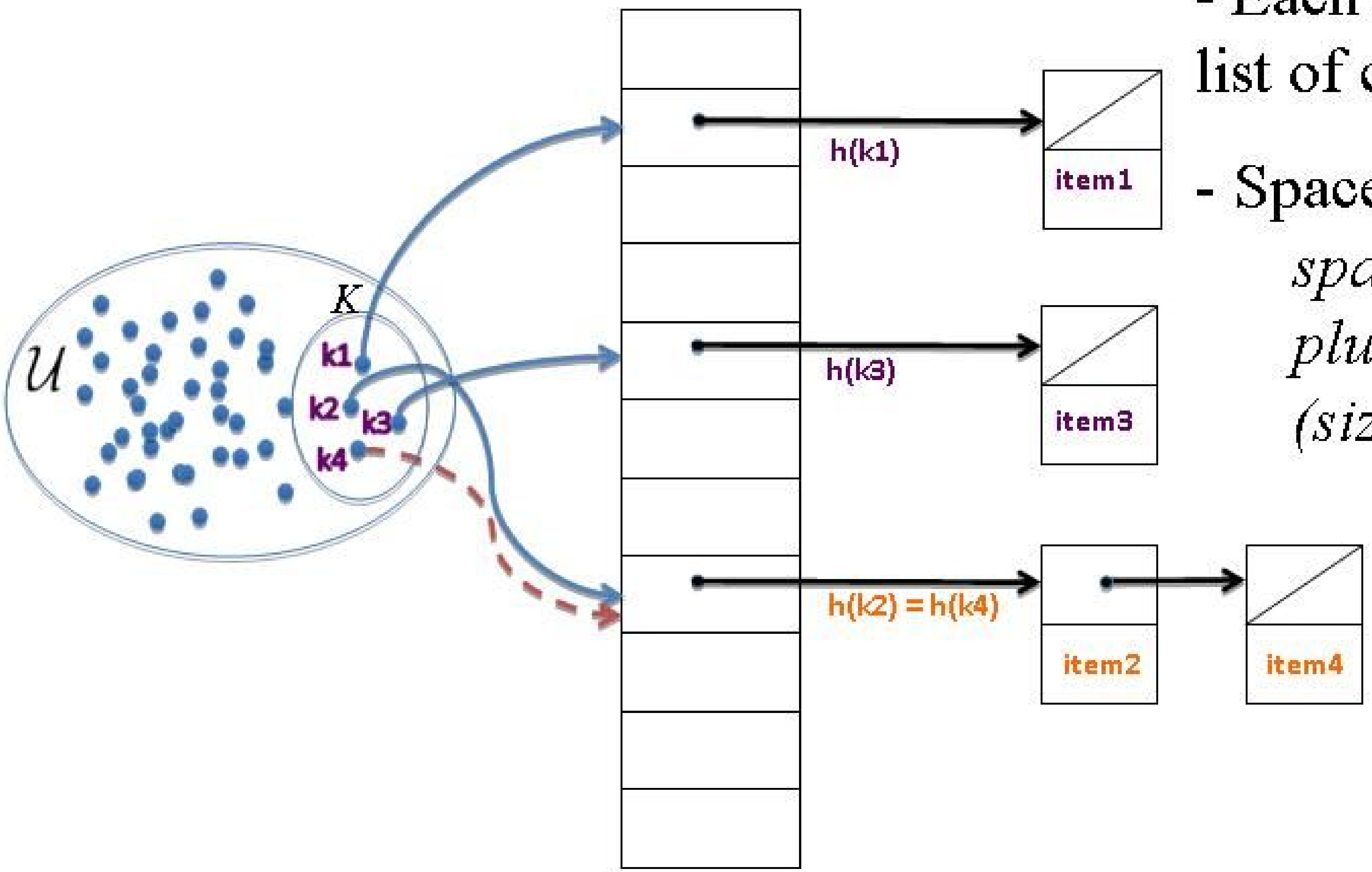
Collisions

- What went/can go wrong?
  - Distinct keys x and y
  - But h(x) = h(y)
  - Called a collision
- This is unavoidable: if table smaller than range, some keys must collide…
  - Pigeonhole principle
- What do you put in the bucket?

Coping with collisions

- **Idea1** : Change to a new "uncolliding" hash function
  - Hard to find, and takes time
- **Idea2**: Chaining
  - Put both items in same bucket (this lecture)
- **Idea3**: Open addressing
  - Find a different, empty bucket for y (next lecture)

Chaining



- Each bucket, linked list of contained items

- Space used is
  space of table
  plus one unit per item
  (size of key and item)

$\mathcal{U}$: *universe of all possible keys*

K : *actual keys, not known in advance*

Problem Solved?

- To find key, must scan whole list in key's bucket
- Length L list costs L key comparisons
- If all keys hash to same bucket, lookup cost $\Theta(n)$

Solution: Optimism

- Assume keys are **equally likely** to land in every bucket, independently of where other keys land
- Call this
  the "Simple Uniform Hashing" assumption
  – (why/when can we make this assumption?)

Average Case Analysis under SUHA

- n items in table of m buckets
- Average number of items/bucket is $\alpha = n/m$
- So expected time to find some key x is $1+\alpha$
- $O(1)$ if $\alpha = O(1)$, i.e. $m = \Omega(n)$

Problem: Reality

- Keys are often very nonrandom
  - Regularity (evenly spaced sequence of keys)
  - All sorts of mysterious patterns
- Solution: pick a hash function whose values "look" random
- Similar to pseudorandom generators
- Whatever function, always some set of keys that is bad
  - but hopefully not your set

Division Hash Function

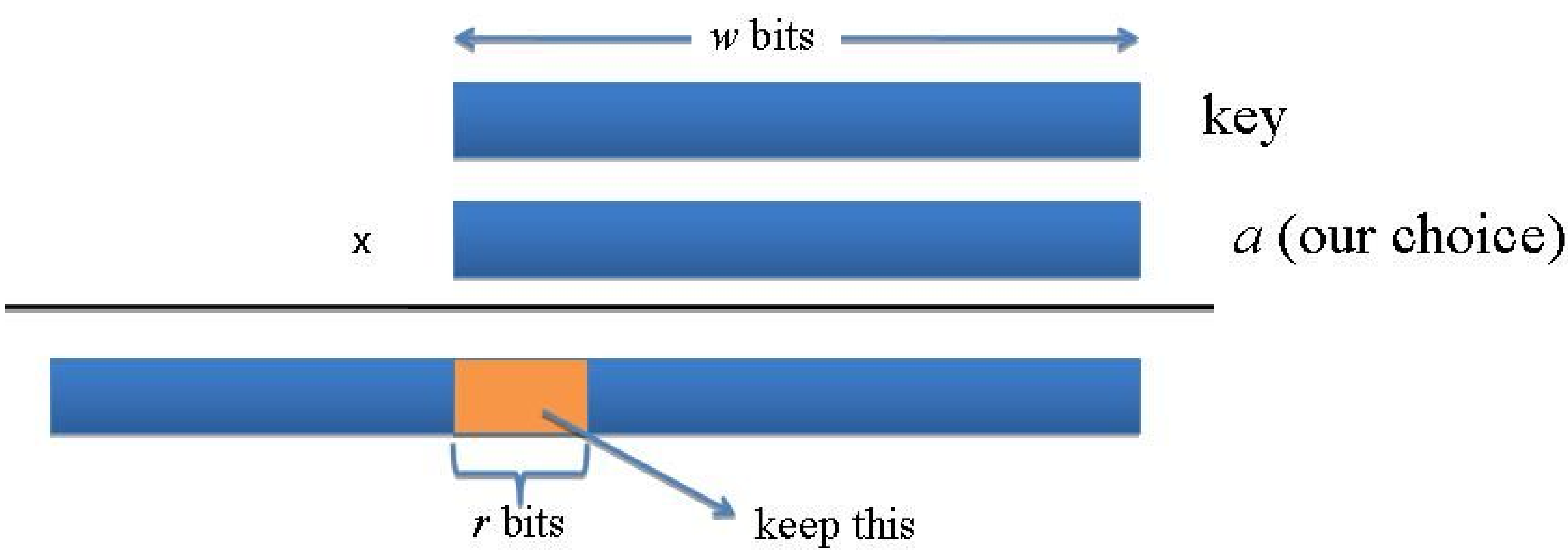
- $h(k) = k \bmod m$
- k_1 and k_2 collide when $k_1 = k_2 \pmod{m}$
  - Unlikely if keys are random
- e.g. if m is a power of 2, just take low order bits of key
  - Very fast (a mask)
  - And people care about very fast in hashing

Problems

- Regularity
  - Suppose keys are x,2x,3x,4x,....
  - Suppose x and chosen m have common divisor d
  - Then (m/d)x is a multiple of m
    - so $i \cdot x = (i+m/d)x \bmod m$
  - Only use 1/d fraction of table
    - E.g, m power of 2 and all keys are even
- So make m a prime number
  - But finding a prime number is hard
  - And now you have to divide (slow)

Multiplication Hash Function

- Suppose we're aiming for table size 2^r
- and keys are w bits long, where $w>r$ is the machine word
- Multiply k with some a (fixed for the hash function)
- then keep certain bits of the result as follows



Multiplication Hash Function

Bit shift

- The formula:

$$h(k) = [(a * k) \bmod 2^{w}] >> (w - r)$$

  - Multiply by a
  - When overflow machine word, wrap
  - Take high r bits of resulting machine word
  - (Assumes table size smaller than machine word)

Benefit: Multiplying and bit shifts faster than division

Good practice: Make a an odd integer (why?) $> 2^{w-1}$

Python Implementation

- Python objects have a hash method
  - Number, string, tuple, any object implementing __hash__
- Maps object to (arbitarily large) integer
  - So really, should be called prehash
- Take mod m to put in a size-m hash table
- Peculiar details
  - Integers map to themselves
  - Strings that differ by one letter don't collide
  - "better" than random for common sequences
    - 1,2,3,4,5 or var_a, var_b, var_c, var_d

Conclusion

- Dictionaries are pervasive
- Hash tables implement them efficiently
  - Under an optimistic assumption of random keys
  - Can be “made true” by choice of hash function
- How did we beat BSTs?
  - Used indexing
  - Sacrificed operations: previous, successor
- Next time: open addressing