Extended Introduction to Computer Science
CS1001.py

Module G Data Structures:
Hash Functions and Hash Tables

Instructors: Elhanan Borenstein, Michal Kleinbort

Teaching Assistants: Noam Parzanchevsky,

Asaf Cassel, Shaked Dovrat, Omri Porat

School of Computer Science
Tel-Aviv University
Spring Semester 2020-21
http://tau-cs1001-py.wikidot.com

* Slides based on a course designed by Prof. Benny Chor

Data Structures

1. Linked Lists
2. Binary Search Trees
3. Hash tables
4. Generators

Lecture Plan

We'll introduce an additional, very efficient data structure: hash table.

- Hash functions
- Hash tables
  - Resolving collisions with chaining
  - "Average" time complexity
  - Implementation in Python

“Hash”?

- Definition (from the Merriam-Webster dictionary):

  hash - transitive verb
  1 a: to chop (as meat and potatoes) into small pieces
  b: confuse, muddle
  2 : to talk about: review -- often used with over or out
  ***Synonyms**: dice, chop, mince*
  ***Antonyms**: arrange, array, dispose, draw up, marshal (also marshall), order, organize, range, regulate, straighten (up), tidy*

- In computer science, hashing has multiple meanings, often unrelated.
  - For example, universal hashing, perfect hashing, cryptographic hashing, and geometric hashing, have very different meanings.

- Common to all of them is a mapping from a large space into a smaller one.

- Today, we will study hashing in the context of hash tables

Hash Functions

- Hash function: a function that maps a large (possibly infinite) set to a smaller set of a fixed size.

- Example for a hash function from integers to integers:

```
def hash4int(n):
    m = 1000
    c = (5**0.5-1)/2 #some irrational, 0<c<1
    return int(m*((n*c)%1))
```

- Executions in class

- Note that this function spreads the (infinite) set of integers over a small finite range (0-999).

- But what can such a function be possibly good for? soon…

Hash Functions (cont.)

- Hash function: a function that maps a large (possible infinite) set to a smaller set of a fixed size.

- Example for a hash function from strings to integers:

```
def hash4strings(st):
    p = 2**120+451 # some arbitrary prime number
    s = 0
    for c in st:
        s = (128*s + ord(c)) % p
    return s
```

- Note that this function spreads the (infinite) set of strings over a finite range (0…p-1).

- But what can such a function be possibly good for? soon…

Python's Built-in hash Function

- Python comes with its own hash function, from any immutable type to integers (both negative and positive):

```
>>> hash("Michal")
5551611717038549197
>>> hash("Amir")
-6654385622067491745
>>> hash((3,4))
3713083796997400956


>>> hash([3,4])
Traceback (most recent call last):
    File "<pyshell #16 >", line 1, in <module >
          hash([3,4])
TypeError: unhashable type: 'list'
```

- But what can such a function be possibly good for? soon…

Hashing with a Random Seed

- If you run this code yourself, you will probably encounter different outputs from those in the last slide.
- This is because when IDLE starts, it randomly generates a number called seed, which is used to compute the built-in hash function
  - This is intended to provide protection against denial-of-service attacks caused by carefully-chosen inputs, designed to exploit a worst case scenarios (which will be explained and analyzed soon).
- But as long as you work under the same instance of IDLE, hash is consistent and deterministic. So consistency is kept for the lifetime of an IDLE session

But what can such a function be possibly good for?

…and now for the answer

Hash Tables: Definition

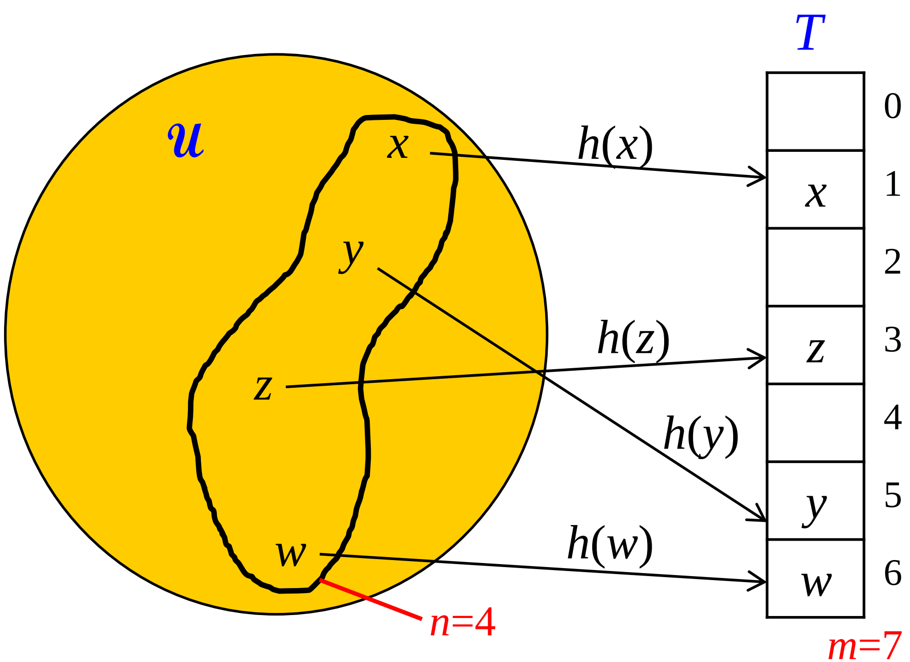
- Suppose elements belong to a large set (possibly infinite), called the "universe", denoted $\mathcal{U}$
  - for example: all possible ID numbers, all possible strings, etc.

- We need to store some n elements from $\mathcal{U}$, and $n << |\mathcal{U}|$.
  - for example: ID's of students in class right now, genes of an organism

- We store the elements in a table T called hash table, whose size is $m \approx n$.

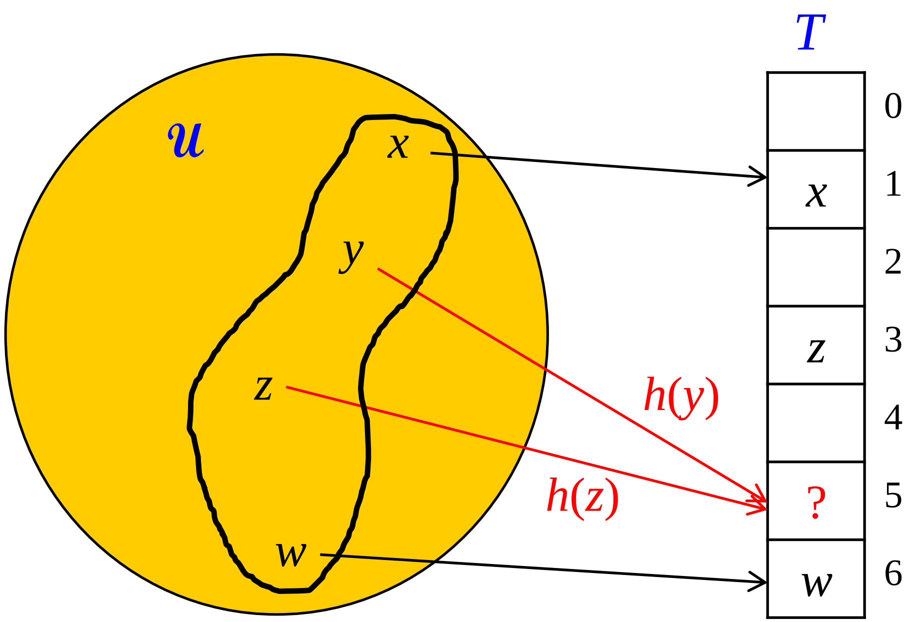
- To map elements from $\mathcal{U}$ to T we use a hash function h: $\mathcal{U} \rightarrow \{0, 1, \ldots, m-1\}$
For example: $h =$ `hash(key)%m`

Element with key $k \in \mathcal{U}$ is stored (and searched for) at index $h(k)$ in T.



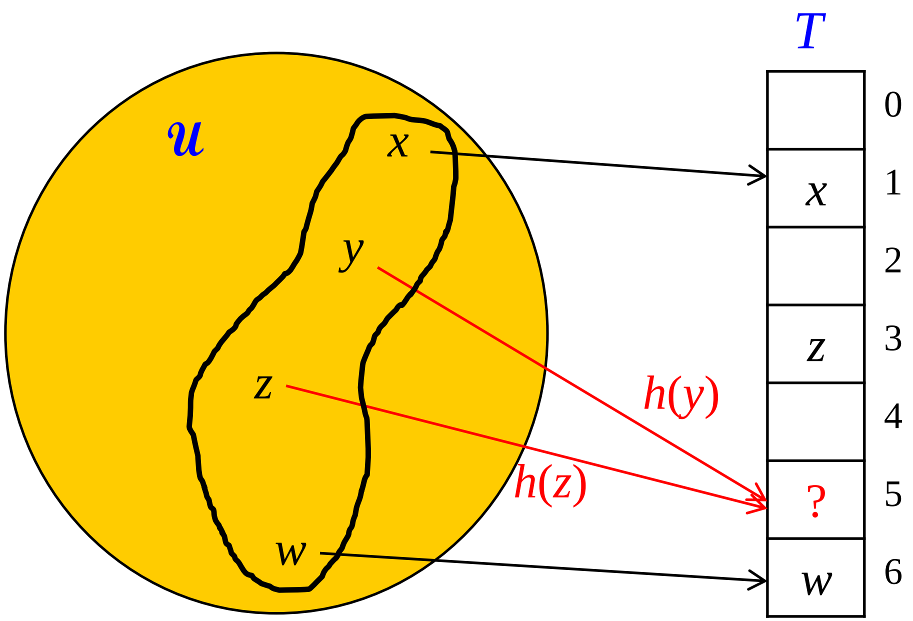
Problem

- Handle collisions while providing efficient insert, delete, search



Collisions

- Collision: $h(k_1) = h(k_2)$ for $k_1 \neq k_2$



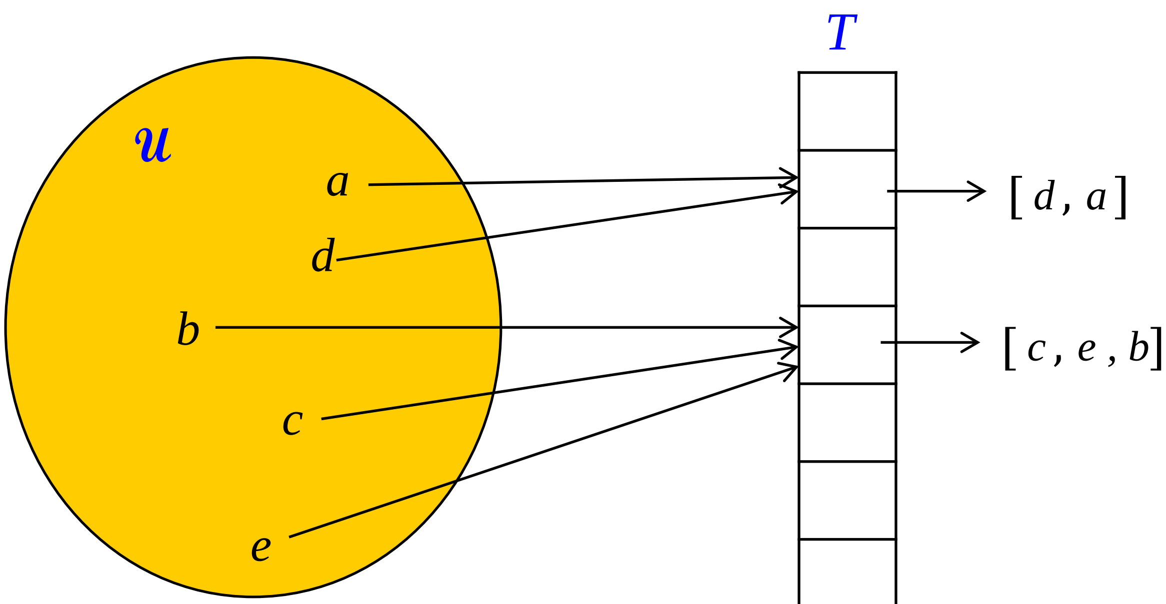
- Can we totally avoid collisions?

> Pigeonhole principle:
>
> *if n+1 pigeons enter n holes,*
>
> *at least 1 hole will contain at least 2 pigeons*

- How can we decrease the probability for collisions?
  - Larger T
  - "Better" h (more on that soon)

Dealing with Collisions: Chaining Method

- Each cell in the table will contain a list (can be linked or not), with all the elements h maps to this cell



- How do we insert, search and delete elements?

Implementing Insert, Delete, Search

- Initialization: create a table T with m empty lists

- Given an element with key $k \in \mathcal{U}$:

  - Search: compute $i = h(k)$ and check if list $T[i]$ contains the key k.

  - Insert: compute $i = h(k)$
    if k not in list $T[i]$, insert element to list $T[i]$.
    otherwise? replace element or make no change.

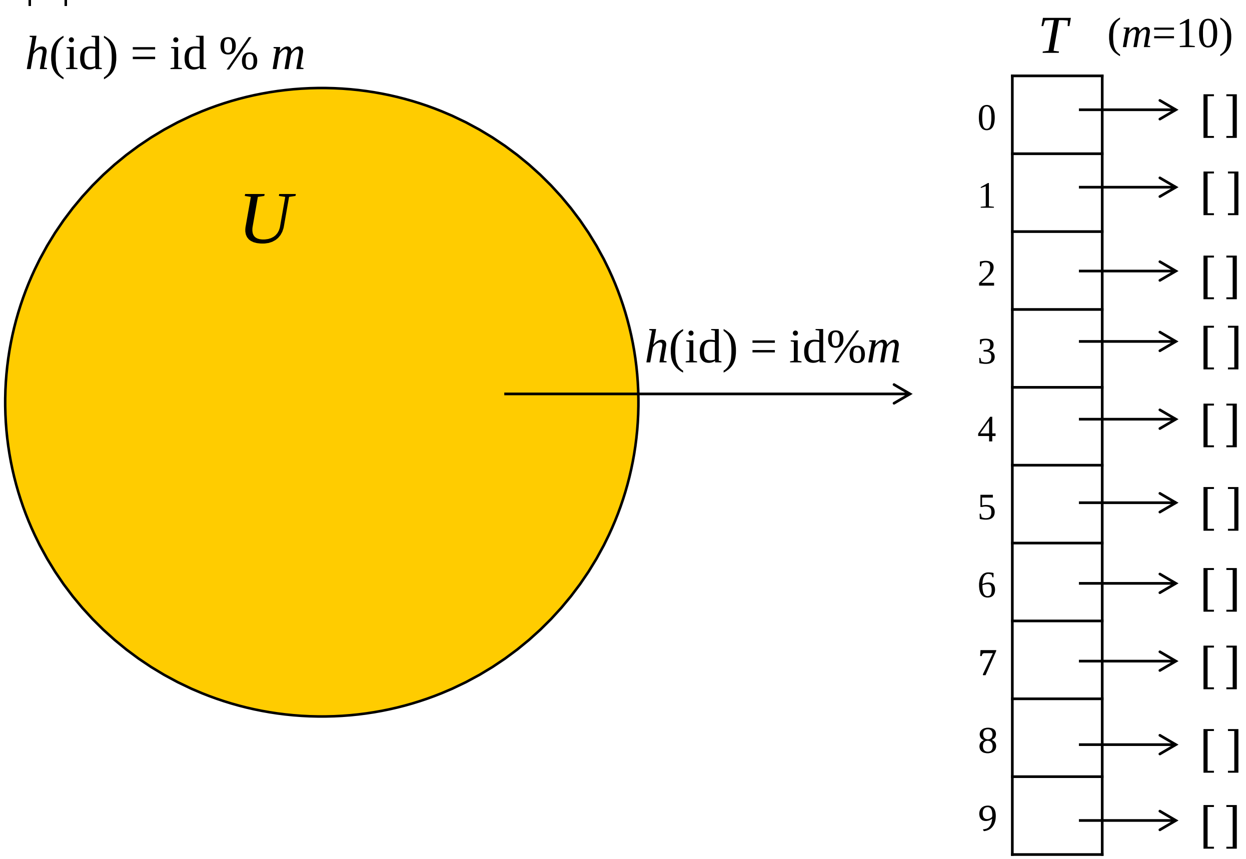
  - Delete: compute $i = h(k)$
    if k in list $T[i]$, remove element from list $T[i]$.

Simple (interactive) Example for ID's

We want to store all students who attended class today, by their ID.

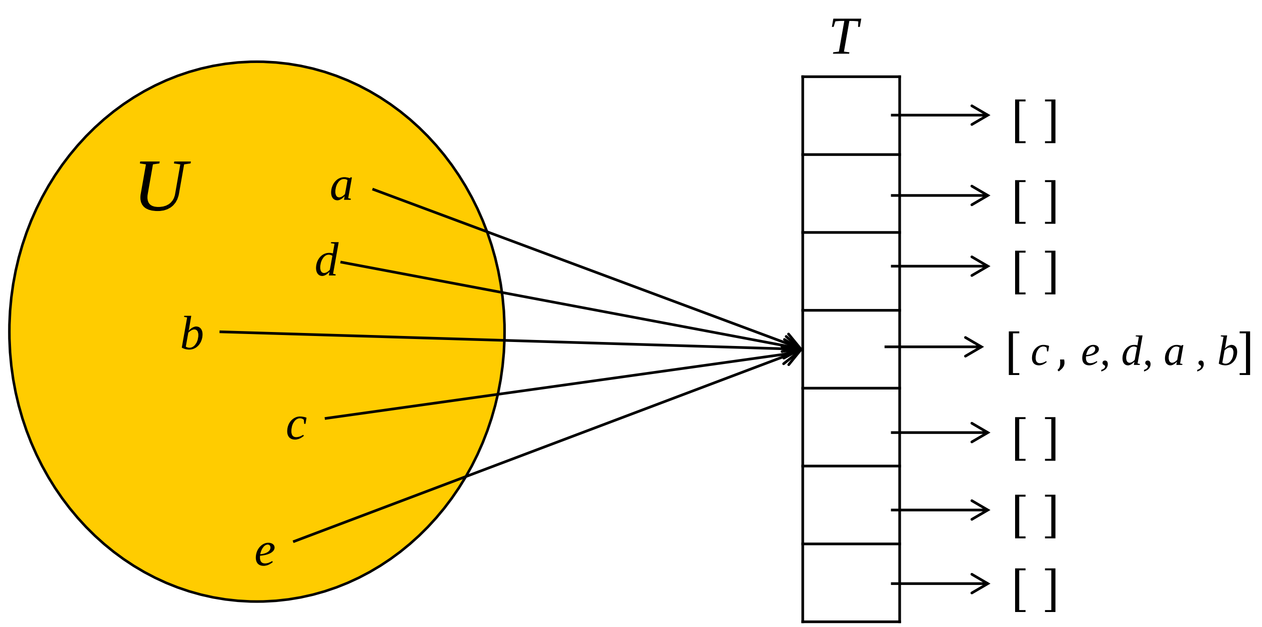
- $\mathcal{U}$ = {all possible Israeli ID numbers }

  $|\mathcal{U}|$ = ?
- n = ?
- $|T| = m = 10$
- $h(\text{id}) = \text{id} \% m$



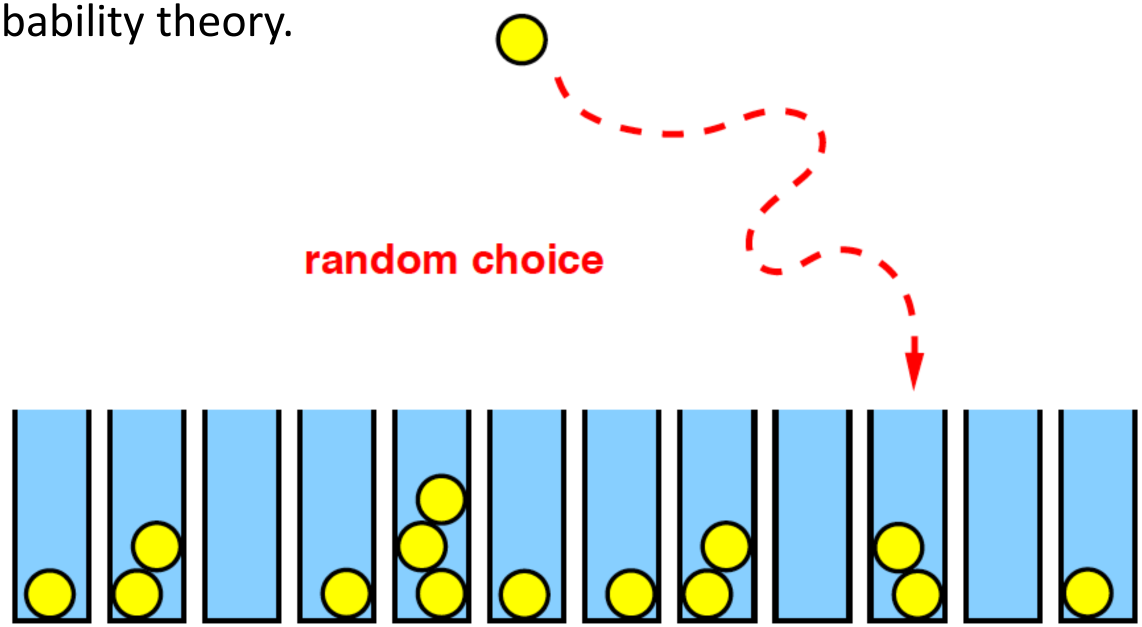
Chaining – Time Complexity: Worst Case

- In each operation we compute $h(k)$ and then iterate over a single chain
- The worst case time complexity, for the three operations search, insert and delete in terms of n is $O(n)$.
  - This happens when all the elements inserted were hashed to the same single cell
  - Assumption: computing h and comparing 2 elements both take O(1) time



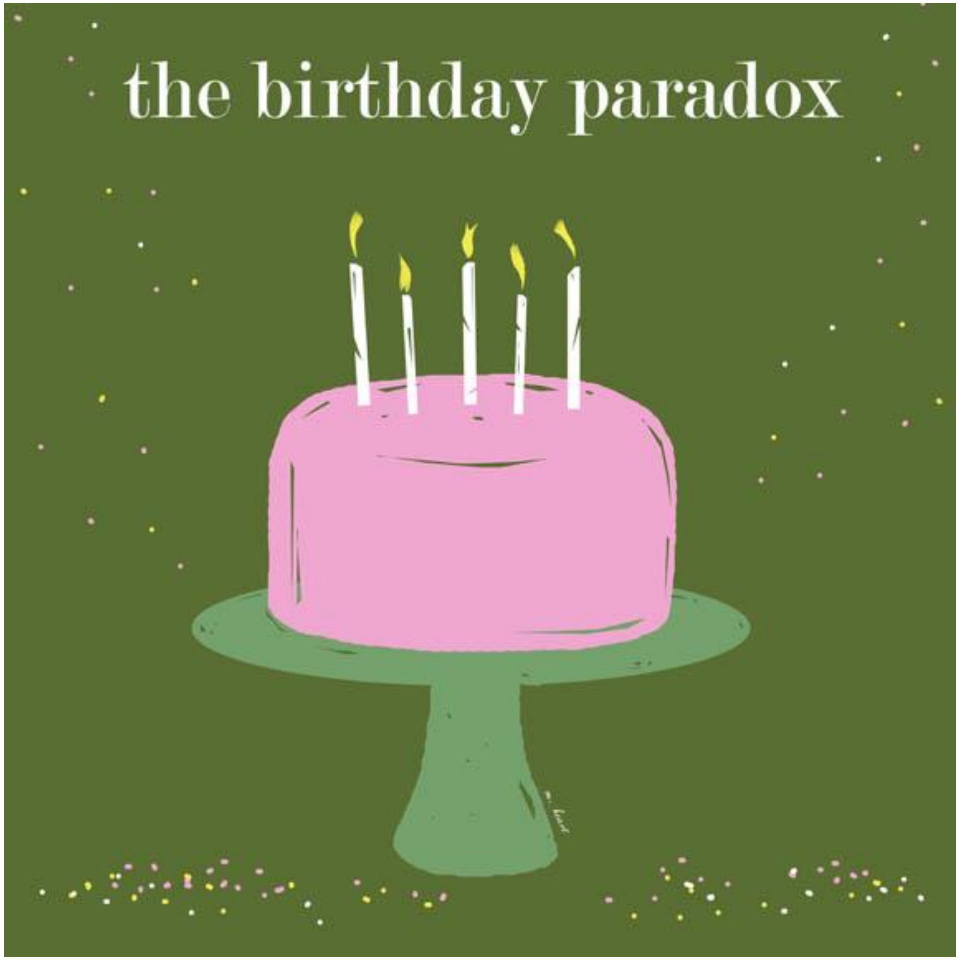
Chaining – Time Complexity: Average

- The worst case may indeed occur. But assuming h was chosen carefully and spreads elements rather uniformly and independently, the worst case is very rare!
  - The definitions of “uniformly” and “independently” will be taught in a probability course.
  - The scenario is often described as throwing n balls into m bins. The distribution of balls in the bins (maximum load, number of empty bins, etc.) is a well studied topic in probability theory.



The figure is taken from a manuscript titled “Balls and Bins -- A Tutorial",
by Berthold Vöcking (Universität Dortmund).

A Related Issue: The Birthday Paradox



(figure taken from http://thenullhypodermic.blogspot.co.il/2012_03_01_archive.html)

The Birthday Paradox

- A well known (and not too hard to prove) result is that if we throw n balls at random into m distinct slots, and $n \approx \sqrt{\pi \cdot m/2}$ then with probability about 0.5, two balls will end up in the same slot.
- For $m = 365$ we get $\sqrt{\pi \cdot 365/2} \approx 23.94$
- This gives rise to the so called "birthday paradox" - given 24 people with random birth dates (month and day of month), with probability > 0.5 two will have the same birth date
- Thus if our set of keys is of size $n > \sqrt{\pi \cdot m/2}$ most likely there will be a collision

Chaining – Time Complexity: Average

(for reference only)

- Numerous additional results from probability theory are well known. For example, we can ask what the expected maximal capacity of a cell is.
  - Maximal capacity = size of the largest colliding set

case	expected maximal capacity in a single slot
$n < \sqrt{m}$	1 (=no collisions)
$n = m^{1-\epsilon} \quad , 0 < \epsilon < 1/2$	$O(1/\varepsilon)$
$n = m$	$\frac{ln(n)}{lnln(n)}$
$n > m$	$\frac{n}{m} + \frac{ln(n)}{lnln(n)}$

- Bottom line: worst case is rare.

Chaining – Time Complexity: Average

- Assuming h indeed "spreads elements well", as mentioned above, it makes sense to measure complexity in terms of the average length of a chain (average here is on the various inputs).
- Average chain length is $\alpha = \frac{n}{m}$ (α is termed the load factor).
- If we choose m (table size) such that $n = O(m)$, then $\alpha = O(1)$.
- Therefore, all operations run in $O(1)$ "on average"
- Note: assuring $n = O(m)$ requires prior estimation of the number of elements n we expect to be inserted into the table, or a mechanism to dynamically update the table size m

Python's dict and set

- Python's class dict and class set are both implemented behind the scenes as hash tables.

- This explains why they are such good choices for storing and searching elements. Indeed, we used them (rather than lists for example) for memoization.

- dict and set however do not use chaining to solve collisions. They use another approach called open addressing (more later and in the data structures course)

- In addition, dict and set are dynamic hash tables – they expand and shrink as the load factor becomes too large or too small, respectively.

- The exact details may change between versions (e.g. 3.7 and 3.8), due to optimization efforts by the language developers. We will not delve into those details.

Time – Space Tradeoff

- We don't want α to be neither too large (why?) nor too small (why?)

“Good” Hash Functions?

- You may wonder what it practically means to choose h “carefully”.
- Is $h(id) = id\%100$ a good hash function for id’s?
- When we have some apriori knowledge on the keys, their distribution and properties, etc., we can tailor a specific hash function, that will improve spread-out among table cells.
- However, such knowledge is not always at hand. In addition, as we mentioned, choosing h at random once in a while is a rather good idea.
  - In the *data structure* course you will define a mechanism called universal families to solve both problems
- Practically, we can expect Python's hash to do a good job.

Implementation in Python

- Let us implement our own `class Hashtable` in Python now.

- We will assume elements have only keys, so we are actually implementing something that resembles Python's sets.

- However, we will use chaining to resolve collisions.

Initializing the Hash Table

```
class Hashtable:

 def __init__(self, m, hash_func=hash):
     """ initial hash table, m empty entries """
     self.table = [ [] for i in range(m)]
     self.hash_mod = lambda key: hash_func(key) % m



 def __repr__(self):
     return "".join([str(i) + " " + str(self.table[i]) + "\n" \
                         for i in range(len(self.table))])
```

Initializing the Hash Table

```
>>> ht = Hashtable (11)
>>> ht
0 []
1 []
2 []
3 []
4 []
5 []
6 []
7 []
8 []
9 []
10 []
```

Initializing the Hash Table: a Bogus Code

Consider the following alternative initialization:

```
class Hashtable:
 def __init__(self, m, hash_func=hash):
     """ initial hash table, m empty entries """
     self.table = [[]]*m
```

```
>>> ht = Hashtable(11)
>>> ht.table[0].append(5)
>>> ht
0 [5]
1 [5]
…

>>> ht.table[0] == ht.table[1]
True
>>> ht.table[0] is ht.table[1]
True
```

The entries produced by this bogus __init__ are identical.
Therefore, mutating one mutate all of them.

Dictionary Operations: Python Code

```
class Hashtable:
…

 def find(self, item):
        """ returns True if item in hashtable, False otherwise  """
        i = self.hash_mod(item)
        chain = self.table[i]
        if item in chain: # a hidden loop
              return True
        else:
            return False
```

```
return item in chain
```

```
 def insert(self, item):
        """ insert an item into table, if not there """
        i = self.hash_mod(item)
        chain = self.table[i]
        if item not in chain: # a hidden loop
            chain.append(item)
```

Example: A Very Small Table
(n = 14, m = 7)

In the following slides, there are executions construct a hash table with m = 7 entries. We'll insert n = 14 string record in it and check how insertions are distributed, and in particular what the maximum number of collisions per cell is.

Our hash table will be a list with m = 7 entries. Each entry will contain a list with a variable length. Initially, each entry of the hash table is an empty list.

Example: A Very Small Table (n = 14, m = 7)

```
>>> tribes = ['Reuben', 'Simeon', 'Levi', 'Judah', 'Dan',
  'Naphtali', 'Gad', 'Asher', 'Issachar', 'Zebulun', 'Benjamin',
  'Joseph', 'Ephraim', 'Manasse']

>>> ht = Hashtable(7)

>>> for name in tribes:
        ht.insert(name)

>>> ht #calls __repr__
      (next slide)
```

Example: A Very Small Table (n = 14, m = 7)

```
>>> ht #calls __repr__
0 []
1 ['Reuben', 'Judah', 'Dan']
2 ['Naphtali']
3 ['Gad', 'Ephraim']
4 ['Levi']
5 ['Issachar', 'Zebulun']
6 ['Simeon', 'Asher', 'Benjamin', 'Joseph', 'Manasse']
```

Example: A slightly larger table (n = 14, m = 21)

```
>>> tribes = ['Reuben', 'Simeon', 'Levi', 'Judah', 'Dan',
   'Naphtali', 'Gad', 'Asher', 'Issachar', 'Zebulun', 'Benjamin',
   'Joseph', 'Ephraim', 'Manasse']


>>> ht = Hashtable(21)


>>> for name in tribes:
        ht.insert(name)


>>> ht #calls __repr__
```

(next slide)

Example: A slightly larger table (n = 14, m = 21)

```
>>> ht #calls __repr__
0 []
1 []
2 []
3 ['Ephraim']
4 []
5 ['Issachar']
6 ['Benjamin']
7 []
8 ['Judah']
9 ['Naphtali']
10 []
11 []
12 ['Zebulun']
13 ['Manasse']
14 []
15 ['Reuben', 'Dan']
16 []
17 ['Gad']
18 ['Levi']
19 []
20 ['Simeon', 'Asher', 'Joseph']
```

Hashing and User-defined Classes

- So far we used our Hashtable class to store Python's built-in types such as str and int.
- We will now use class Hashtable on our own **class** Student.
- As we will see, this will raise certain issues, which we will solve.

The Student Class (reminder)

```
class Student:
    def __init__(self, name, surname, ID):
        self.name = name
        self.surname = surname
        self.id = ID
        self.grades = dict()

    def __repr__(self): #must return a string
        return "<" + self.name + ", " + str(self.id) + ">"


    def update_grade(self, course, grade):
        self.grades[course] = grade


    def avg(self):
        s = sum([self.grades[course] for course in self.grades])
        return s / len(self.grades)
```

Hashing Students

```
>>> st1 = Student("Grace", "Hopper", 123456789)
>>> st2 = Student("Grace", "Hopper", 123456789)

>>> st1
<Grace, 123456789>
>>> st2
<Grace, 123456789>

>>> hash(st1)
-9223372036851698786
>>> hash(st2)
3077117
```

From Wikipedia:
Grace Brewster Murray Hopper (1906 –1992), was an American computer scientist and United States Navy Rear Admiral.
She was one of the first programmers of the Harvard Mark I computer in 1944, invented the first compiler for a computer programming language, and was one of those who popularized the idea of machine-independent programming languages which led to the development of COBOL, one of the first high-level programming languages.

- This should not be a surprise to you: by default, Python uses the memory address of an object to compute the value of hash on it.

The __hash__ Method

- We will add to class Student the special method __hash__.
- It defines the result of calling Python's hash on an object of this class.

```
class Student:
       …
       def __hash__(self): #so we can use hash(st) on a student
            return hash(self.id) #assume student id number is a
                                   unique identifier
```

- Notes:
  1) __hash__ of Student class calls __hash__ of int class
  2) We used merely the student's id to compute a student's hash, under the assumption that it is unique. We could have used more fields of a Student object.

Hashing Students – almost done

```
>>> st1 = Student("Grace", "Hopper", 123456789)
>>> st2 = Student("Grace", "Hopper", 123456789)

>>> hash(st1) == hash(st2) == hash(st1.id)
True ☺
```

Hashing Students – almost done

- Can you explain why the following search fails?

```
>>> st1 = Student("Grace", "Hopper", 123456789)
>>> st2 = Student("Grace", "Hopper", 123456789)

>>> ht = Hashtable(7)
>>> ht.insert(st1)
>>> ht
0 []
1 [<Grace, 123456789>]
2 []
3 []
4 []
5 []
6 []

>>> ht.find(st2)
False ☹
```

Hash Tables Involve Comparisons

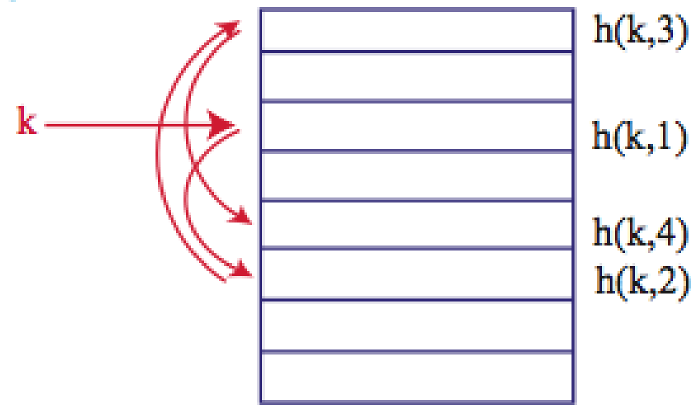
- Indeed, no much point in having __hash__ without __eq__, for comparing elements (within a chain inside a table's index).

```
class Student :
    …
    def __eq__(self, other):
        return self.name == other.name and \
               self.surname == other.surname  and \
               self.id == other.id
```

```
>>> ht.find(st2) # recall st2 holds same data as st1
True ☺
```

Open Addressing

- In open addressing, each slot in the hash table contains at most one item. This obviously implies that n cannot be larger than m.
- Each element enters the first vacant cell among a series of hash outputs:



- Open addressing is important in hardware applications where devices have many slots but each can only store one item (e.g. fast switches and high capacity routers). It is also used in python dictionaries and sets.
- There are many approaches to open addressing. A fairly recent one is termed cuckoo hashing (Pagh and Rodler, 2001).