Extended Introduction to Computer Science CS1001.py

Lecture 15: The Dictionary Problem
Hash Functions and Hash Tables

Instructors: Daniel Deutch , Amiram Yehudai

Teaching Assistants: Michal Kleinbort, Amir Rubinstein

School of Computer Science
Tel-Aviv University
Winter Semester, 2014-15
http://tau-cs1001-py.wikidot.com

Lecture 14: Highlights

Data Structures

Linked Lists

Trees

Binary search trees

Lecture 15 - Plan

The dictionary problem (find, insert, delete).

Python hash and <class 'dict'>.

Hash functions and hash tables.

Resolving collisions: Chaining and open addressing.

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

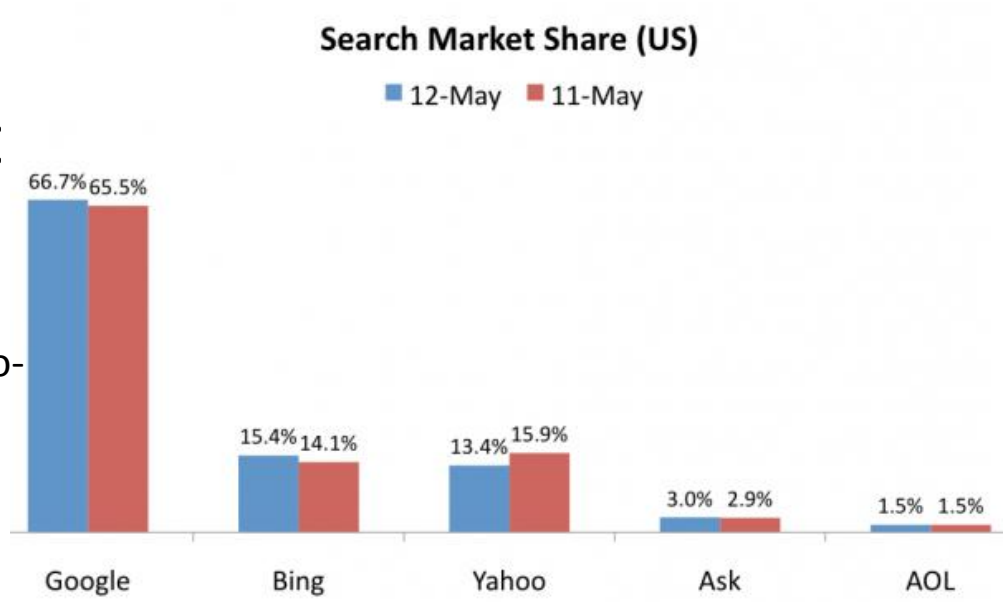
In computer science, hashing has multiple meaning, often unrelated. For example, universal hashing, perfect hashing, cryptographic hashing, and geometric hashing, have very different meanings. Common to all of them is a mapping from a large space into a smaller one.
Today, we will study hashing in the context of the dictionary problem

Hash Functions, Hash Tables, and Search

(figure from http://searchengineland.com/search-market-share-google-up-bing-at-yahoo-hits-new-low-124519)

And, while at that

(figure from http://www.designbaskets.com/services/seo-sem/) (May 2012 data.)



Search (reminder from lectures 6 and 14)

Search has always been a central computational task. The emergence and the popularization of the world wide web has literally created a universe of data, and with it the need to pinpoint information in this universe.

Various search engines have emerged, to cope with this big data challenge. They constantly collect data on the web, organize it, and store it in sophisticated data structures that support efficient (very fast) access, resilience to failures, frequent updates, including deletions, etc. etc.

In lecture 6, we have dealt with much simpler data structure that support search:

- unordered list
- ordered list

Sequential vs. Binary Search

For unordered lists of length n, in the worst case, a search operation compares the key to all list items, namely n comparisons.

On the other hand, if the n elements list is sorted, search can be performed much faster, in time O(log n).

One disadvantage of sorted lists is that they are static. Once a list is sorted, if we wish to insert a new item, or to delete an old one, we essentially have to reorganize the whole list -- requiring O(n) operations.

Linked lists also exhibit O(n) worst time performance for some insert, delete, and even search operations.

Dynamic Data Structure: Dictionary

A dictionary is a data structure supporting efficient insert, delete, and search operations.

We will introduce hash functions, and use them to build hash tables. These hash tables will be used here to implement the abstract data type dictionary.

The abstract data type dictionary should not be confused with Python's <class 'dict'>, although <class 'dict'> can be thought of as an implementation of the abstract data type dictionary

Dynamic Data Structure: Dictionary

In our setting, there is a dynamic (changing with time) collection of up to n items. Each item is an object that is identified by a key. For example, items may be instances of our Student class, the keys are students' names, and the returned values are the students' ID numbers and grades in the course.

We assume that keys are unique (different items have different keys).

Other Dynamic Data Structure

There are data structures, known as balanced search trees, which support these three operations in worst case time O(log n). They are fairly involved, and studied extensively in the data structures course.

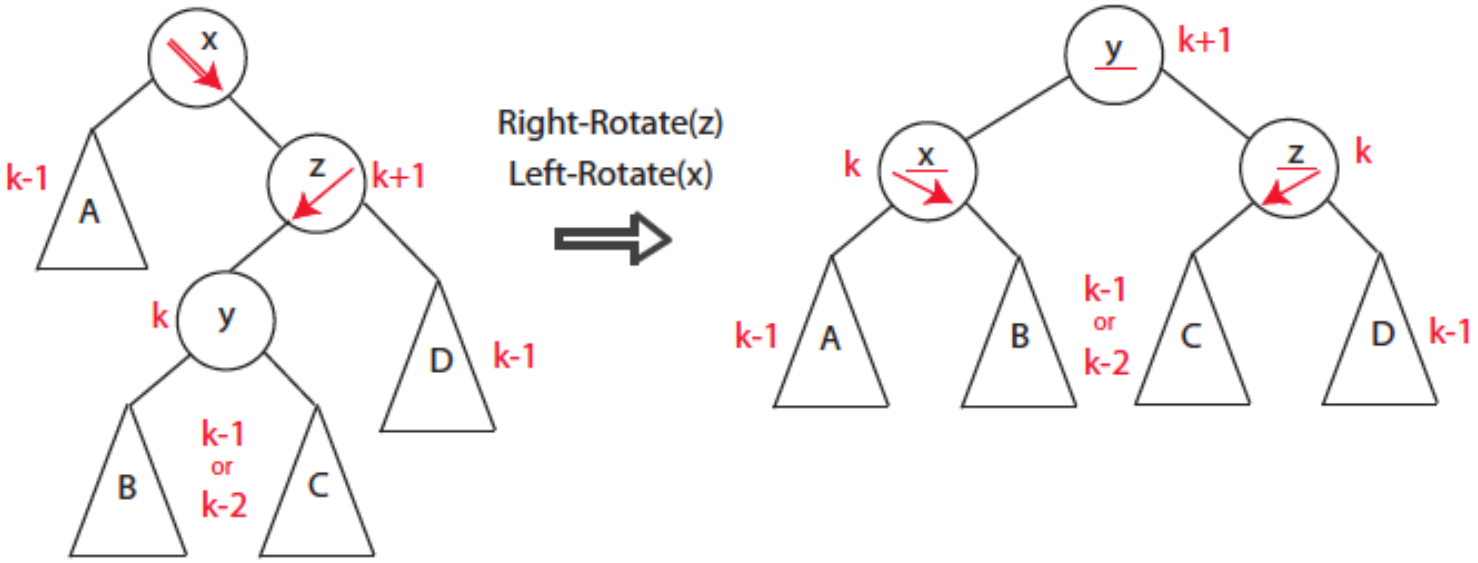
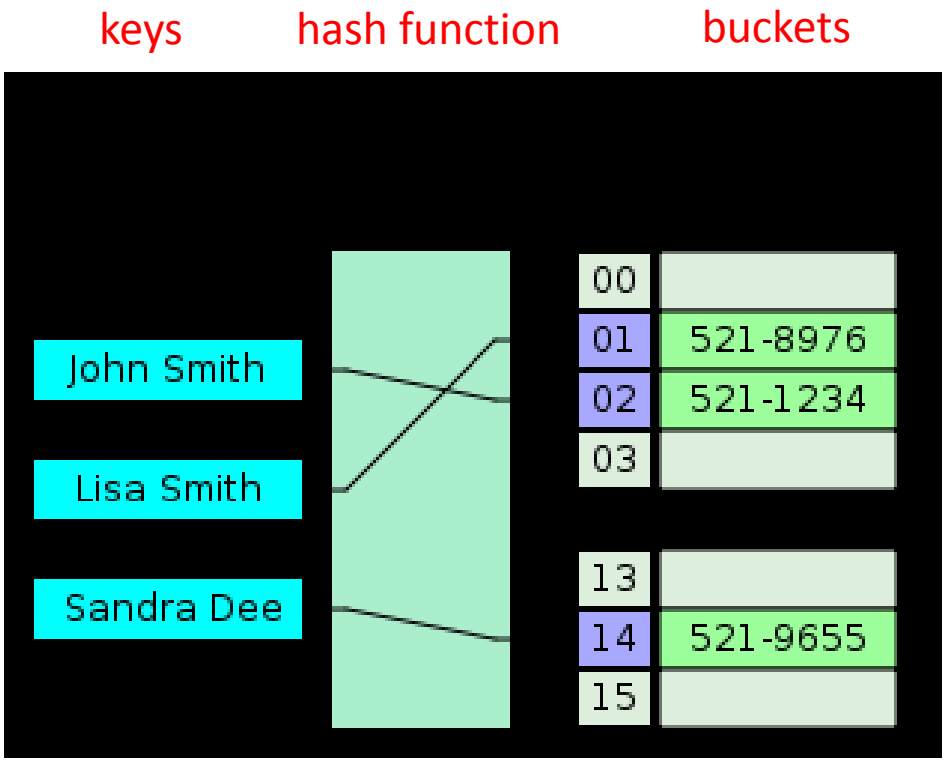


Figure from MIT algorithms course, 2008. Shows item insertion in an AVL tree.

Dynamic Data Structure: Dictionary

Question: Is it possible to implement these three operations, insert, delete, and search, in time O(1) (a constant, regardless of n)?
As we will shortly see, this goal can be achieved on average using the so called hash functions and a data structure known as a hash table.



(figure from Wikipedia)

We note that Python's dictionary (storing key:value pairs) is indeed implemented using a hash table.

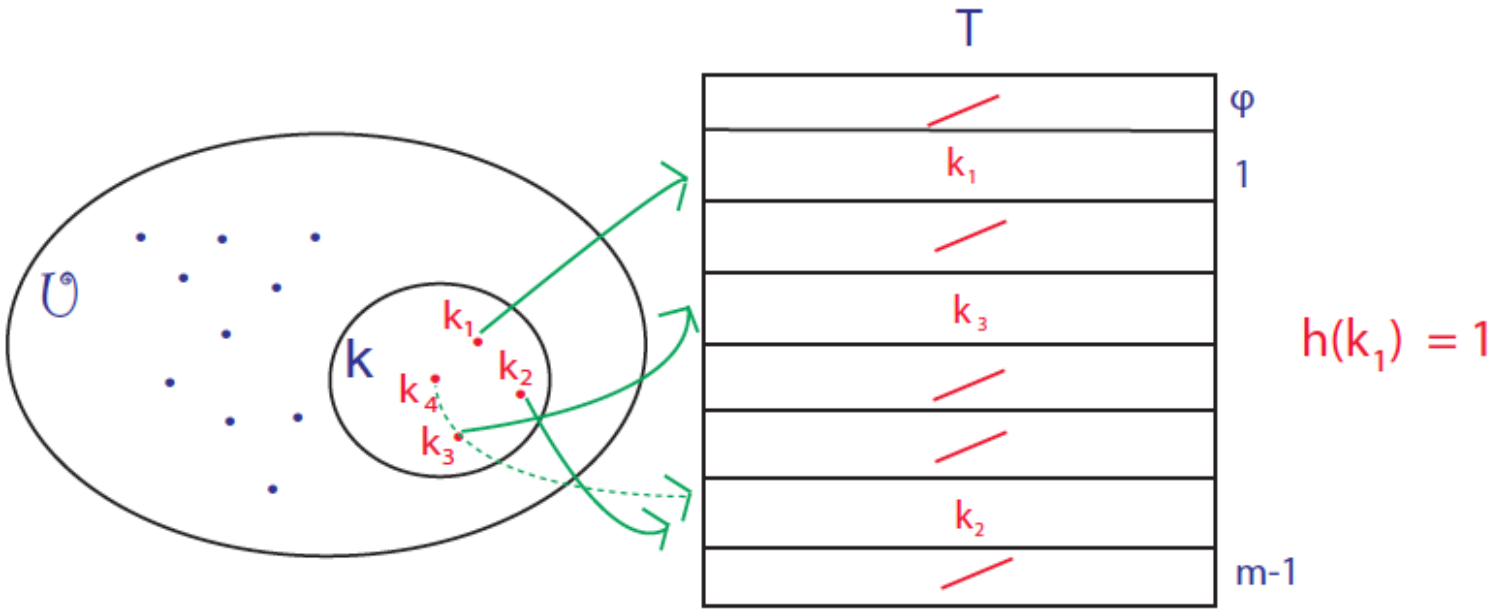
Dictionary Setting

A very large universe of keys, $\mathcal{U}$, Say students with their names.

A much smaller set of keys, $\mathcal{K}$, containing up to n keys.

The keys in $\mathcal{K}$ are initially unknown, and may change.

Map $\mathcal{K}$ to a table , $\mathcal{T}$={0,…,m-1} of size m, where m ≈ n, using hash function , h: $\mathcal{U} \rightarrow \mathcal{T}$ (h cannot depend on $\mathcal{K}$).



12.Figure from MIT algorithms course, 2008

Implementing Insert, Delete, Search

The universe of all possible keys, $\mathcal{U}$, is much much larger than the set of actual keys, $\mathcal{K}$, whose size is up to n. Mapping is by a (fixed) hash function, h: $\mathcal{U} \rightarrow \mathcal{T}$ that does not depend on $\mathcal{K}$.

- Given an item with key k $\in \mathcal{U}$.
- Compute h(k) and check if in $\mathcal{T}$ (this is search).
- If not, can insert item to cell h(k) in $\mathcal{T}$.
- If it is, can delete item from cell h(k) in $\mathcal{T}$.

If h(k) can be computed in constant time and insertion/deletion can be implemented in constant worst case time, we will achieve our goal.

Since $|\mathcal{U}| \gg$ n and h does not depend on $\mathcal{K}$, this last goal is clearly impossible.

If we are really unlucky, h will map all n keys in $\mathcal{K}$ to the same value. Going over all these items will take O(n) steps, instead of the desired O(1) steps.

Collisions of Hashed Values

We usually assume that the set of keys is generated independently of h, so that the values h(k) are randomly distributed in the hash table. We will analyze hashing under this assumption.

We say that two keys, $k_1, k_2 \in \mathcal{K}$ collide (under the function h) if $h(k_1)=h(k_2)$.

Let $|\mathcal{K}| = n$ and $|\mathcal{T}| = m$, and assume that the values h(k) for $k \in \mathcal{K}$ are distributed in $\mathcal{T}$ at random. What is the probability that a collision exists? What is the size of the largest colliding set (a set $S \subset \mathcal{K}$ whose elements are all mapped to the same target by h).

The answer to this question depends on the ratio $\alpha = n/m$. This ratio is the average number of keys per entry in the table, and is called the load factor.

If $\alpha > 1$, then clearly there is at least one collision (pigeon hall principle). If $\alpha \leq 1$, and we could tailor h to $\mathcal{K}$, then we could avoid collisions. However, such tinkering is not possible in our context.

Python's hash Function

Python comes with its own hash function, from everything immutable to integers (both negative and positive).

```
>>> hash (1)
1
>>> hash (0)
0
>>> hash (10000000)
10000000
>>> hash ("a")
 -468864544
>>> hash ( -468864544)
-468864544
>>> hash ("b")
-340864157
```

Note that Python's hash function is not “truly random". Yet what we care about is how it typically handles collisions, and it does seem to handle them well. We intend to employ Python's hash function for our needs. But we will have to make one important modifications to it.

Python's hash Function, cont.

Python comes with its own hash function, from everything immutable to integers (both negative and positive).

```
>>> hash (" Benny ")
5551611717038549197
>>> hash (" Amir ")
-6654385622067491745 # negative
>>> 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 '
```

Python's hash Function, cont. cont.

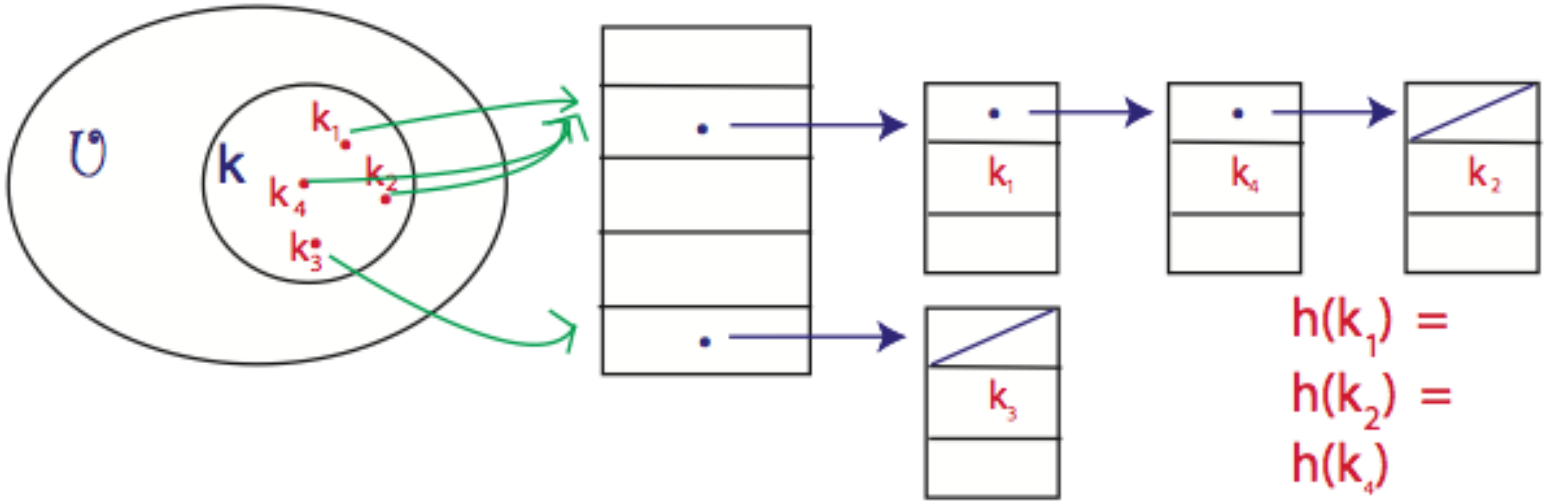
What concerns us mostly right now is that the range of Python's hash function is too large.
To take care of this, we simply reduce its outcome modulo p, the size of the hash table. It is recommended to use a prime modulus.

```
def hash_mod (key ,p, func = hash ):
    return func ( key ) % p
```

Note that our default parameter is using Python's built in hash. But we could (and would) employ other functions, (hopefully) good or bad.

Approaches for Dealing with Collisions: The First Approach

1) Chaining:



In our example of hashing students' names, we are going to use chaining. We will implement and analyze chaining on this small list and also on much larger examples.

Resolving Collisions Using Chaining

In our example we will hash students' names, and are going to use chaining for resolving collisions. We will implement and analyze chaining on this small list and then on much larger examples.

We process the 6 items (students' records) to be inserted into the hash table one by one. For each item, we apply the hash function, hash_mod, to its key (the student's name). The result is an integer, ℓ, which is an index of an entry in the hash table ($0 \leq \ell \leq m-1$).

We access the ℓ -th element in the hash table, which is a list. We search this list sequentially. If an equal item is not found in this list, we append “our student" at the end of the list.

Hash Table: A Very Small Example (n = 14, m = 23)

We'll construct a hash table with m = 23 entries. We'll insert n = 14 students' record in it and check how insertions are distributed, and in particular what is the maximum number of collisions.

Our hash table will be a list with m = 23 entries. Each entry will contain a list with a variable length. Initially, each entry of the hash table is an empty list.

We employ a hash function that maps strings (possible names of students) to the range {0, 1, … , 22} (indices in the hash table). Given a student, we apply the hash function to its name, which is the key in our case.

Hash Table: A Very Small Example (n = 14, m = 23)

Given a student, we apply the hash function to its name, which is the key in our case. The hash function in the code below is our hash_mod. If the result is ℓ , we will map (the record of) this student to entry number ℓ in the hash table.

Please welcome our 14 new students (new to this class, that is):

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

>>> [( name , hash_mod (name ,23)) for name in names ]
[( 'Reuben ', 14) , ('Simeon ', 12) , ('Levi ', 17) , ('Judah ', 2), ('Dan ', 7),
('Naphtali ', 11) , ('Gad ', 18) , ('Asher ', 7), ('Issachar ', 16) ,
('Zebulun ', 11) , ('Benjamin ', 2), ('Joseph ', 11) , ('Ephraim ', 3),
('Manasse ', 14)]
```

Hash Table: A Very Small Example

Let us sort for easier view of the collisions:

```
>>> sorted (_, key = lambda x:x [1]) # _ is the last value returned
[( 'Judah ', 2), ('Benjamin ', 2), ('Ephraim ', 3), ('Dan ', 7),
('Asher ', 7), ('Naphtali ', 11) , ('Zebulun ', 11) , ('Joseph ', 11) ,
('Simeon ', 12) , ('Reuben ', 14) , ('Manasse ', 14) , ('Issachar ', 16) ,
('Levi ', 17) , ('Gad ', 18)]
```

In the example above, with n = 14, m = 23, we see that there are three collisions: Judah and Benjamin are both mapped to the same entry in the hash table, $\ell = 2$, Asher, Naphtali and Zebulun are mapped to the same entry in the hash table, $\ell = 11$, and Reuben and Manasse are both mapped to entry $\ell = 14$.

In this case, hash_mod(name,23) performed rather poorly.

Question is, how shall we deal with such collisions?

A Slightly Larger Example (n = 14, m = 31)

Let us take a somewhat bigger table (m = 31 slots) for the same number of keys (n = 14). Are there fewer collisions?

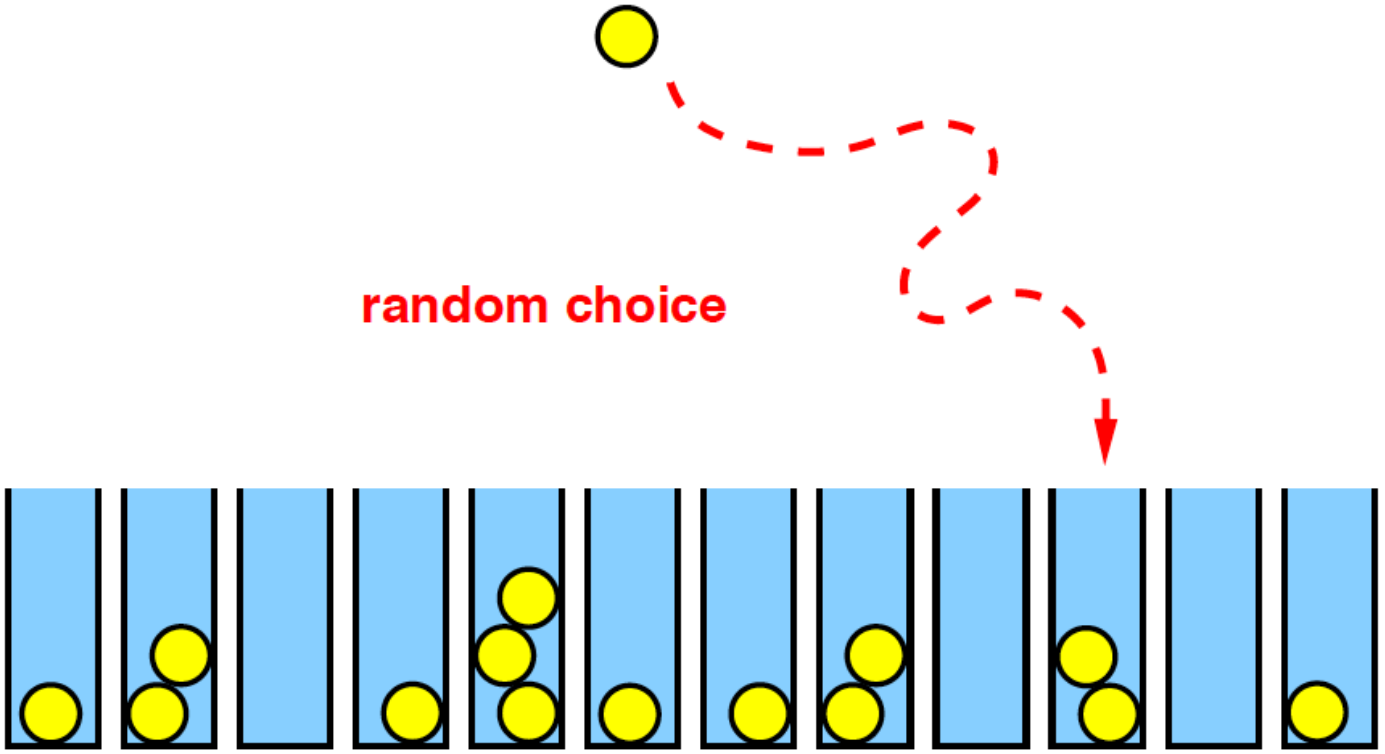
```
>>> [( name , hash_mod (name ,31)) for name in names ]
[( 'Reuben ', 18) , ('Simeon ', 30) , ('Levi ', 24) , ('Judah ', 23) ,
('Dan ', 16) , ('Naphtali ', 9), ('Gad ', 26) , ('Asher ', 23) ,
('Issachar ', 20) , ('Zebulun ', 5), ('Benjamin ', 18) , ('Joseph ', 25) ,
('Ephraim ', 1), ('Manasse ', 4)]
>>> sorted (_, key = lambda x:x [1])
[( 'Ephraim ', 1), ('Manasse ', 4), ('Zebulun ', 5), ('Naphtali ', 9),
('Dan ', 16) , ('Reuben ', 18) , ('Benjamin ', 18) , ('Issachar ', 20) ,
('Judah ', 23) , ('Asher ', 23) , ('Levi ', 24) , ('Joseph ', 25) ,
('Gad ', 26) , ('Simeon ', 30)]
```

We see that there are just two collisions: Reuben and Benjamin are both mapped to the same entry in the hash table, $\ell = 18$, Judah and Naphtali are mapped to the same entry in the hash table, $\ell = 23$. No size 3 collision this time! In this case, hash_mod(name,31) performed better than hash_mod(name,23).

Question still is, how shall we deal with such collisions?

Collisions' Sizes: Throwing Balls into Bins

We throw n balls (items) at random (uniformly and independently) into m bins (hash table entries). 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 and Maximum Collision Size

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.

This gives rise to the so called ”birthday paradox" -- given about 24 people with random birth dates (month and day of month), with probability exceeding 1/2, two will have the same birth date (here m = 365 and $\sqrt{\pi \cdot 365/2} = 23.94$)

Thus if our set of keys is of size $n \approx \sqrt{\pi \cdot m/2}$ two keys are likely to create a collision.

It is also known that if n = m, the expected size of the largest colliding set is ln n/ln ln n.

Collisions of Hashed Values

We say that two keys, $k_1, k_2 \in \mathcal{K}$ collide (under the function h) if $h(k_1)=h(k_2)$.

Let $|\mathcal{K}| = n$ and $|\mathcal{T}| = m$, and assume that the values h(k) for $k \in \mathcal{K}$ are distributed in $\mathcal{T}$ at random. What is the probability that a collision exists ? What is the size of the largest colliding set (a set $S \subset \mathcal{K}$ whose elements are all mapped to the same target by h).

The answer to this question depends on the ratio $\alpha = n/m$. This ratio is the average number of keys per entry in the table, and is called the load factor.

If $\alpha > 1$, then clearly there is at least one collision (pigeon hall principle). If $\alpha \leq 1$, and we could tailor h to $\mathcal{K}$, then we could avoid collisions. However, such tinkering is not possible in our context.

Collision Size

Let $|\mathcal{K}|$ = n and $|\mathcal{T}|$ = m. It is known that

- If $n < \sqrt{m}$, the expected maximal capacity (in a single slot) is 1, i.e. no collisions at all.
- If $n = m^{1-\varepsilon}, 0 < \varepsilon < 1/2$, the expected maximal capacity (in a single slot) is O(1/ε).
- If n = m, the expected maximal capacity (in a single slot) is ln n / ln ln n.
- If n > m, the expected maximal capacity (in a single slot) n/m + ln n/ ln ln n.

Resolving Collisions Using Chaining

In our example we will hash students' names, and are going to use chaining for resolving collisions. We will implement and analyze chaining on this small list and then on much larger examples.

We process the 6 items (students' records) to be inserted into the hash table one by one. For each item, we apply the hash function, hash_mod, to its key (the student's name). The result is an integer, ℓ, which is an index of an entry in the hash table ($0 \leq \ell \leq m-1$).

We access the ℓ -th element in the hash table, which is a list. We search this list sequentially. If an equal item is not found in this list, we append “our student" at the end of the list.

The Student Class

The key will be the name, while the value will be the pair israeli id,grade.

```
class Student :
    def __init__ ( self ):
        self . name = generate_name ()
        self .israeli_id = random . randint (2*10**7 ,6*10**7)
        self .grade = random . randint (19,99)

    def __repr__ ( self ):
        return str . format (" <{} ,{} ,{} >",
                self .name , self . israeli_id , self . grade )

    def __lt__ (self , other ):
        return self . name < other.name
```

Additional code (outside Student Class)

```
def students (n):
    return [ Student () for i in range (n)]

def next_prime ( start ):
    for i in range (start ,2* start ):
        if is_prime (i):
            return i
```

Dictionary Operations: Python Code (find)

```
def find ( candidate_key ,table , func =hash , mod =0):
    if mod ==0:
        mod = len ( table )
    i= hash_mod ( candidate_key ,mod , func )
    list_of_items = table [i]
    k= contained ( candidate_key , list_of_items )
#     print (i,k)          # diagnostic print
     if k != None :      # key exists in table
          return list_of_items [k]. israeli_id, \
                    list_of_items [k]. grade
      else :
          return None
```

Dictionary Operations: Python Code (insert)

```
def insert ( new_item ,table , func =hash , mod =0):
    """ insert an item into a hash table . If key already
    exists , the value is updated . Default mod for
    hash_mod is table 's size . Always returns None """
    if mod ==0:
      mod = len ( table )
     i= hash_mod ( new_item .name ,mod , func )
     list_of_items = table [i]
     k= contained ( new_item .name , list_of_items )
     if k != None :          # key exists in table
           print ( list_of_items [k])
           list_of_items [k]= new_item
           print ( list_of_items [k])
     else :
           list . append ( list_of_items , new_item )
     return None
```

Underlying Dictionary Operations (hash mod and contained)

```
def hash_mod (key ,p, func = hash ):
    return func ( key ) % p


def contained ( candidate_key , list_of_items ):
    """ checks if item is a member in list_of_items
    returns location in list (if found ), or None """
    for k in range ( len ( list_of_items )):
        if candidate_key == list_of_items [k]. name :
            return k # return index of candidate in list
    return None
```

Initializing the Hash Table

```
def hash_table (m):
        return [[] for i in range (m)]
# initial hash table , m different empty entries

>> ht= hash_table (11)
>>> ht
[[] ,[] ,[] ,[] ,[] ,[] ,[] ,[] ,[] ,[] , []]
>>> ht [0] == ht [1]
True
>>> ht [0] is ht [1]
False
```

Since our table is a list of lists, and lists are mutable, we should be careful even when initializing the list.

Initializing the Hash Table, cont.

The following code does produce the desired outcome, and is an alternative to the code presented in previous slide.

```
def fixed_hash_table (m):
      empty =[]
      return [ list ( empty ) for i in range (m)]

# initial hash table , m empty entries
>>> ht= fixed_hash_table (11)
>>> ht [0] is ht [1]
False

>>> list . append (ht [0] ,7)
>>> ht
[[] ,[] ,[] ,[] ,[] ,[] ,[] ,[] ,[] ,[] , [7]]
```

Initializing the Hash Table: a Bogus Code

Consider the following alternative initialization

```
def bogus_hash_table (m):
      empty =[]
      return [ empty for i in range (m)]
# initial hash table , m empty entries

>>> ht= bogus_hash_table (11)
>>> ht
[[] ,[] ,[] ,[] ,[] ,[] ,[] ,[] ,[] ,[] , []]
>>> ht [0] == ht [1]
True
>>> ht [0] is ht [1]
True
```

The entries produced by bogus hash table(n) are identical.
Therefore, mutating one mutate all of them:

```
>>> list . append (ht [0] ,7)
>>> ht
[[7] , [7] , [7] , [7] , [7] , [7] , [7] , [7] , [7] , [7]]
```