Extended Introduction to Computer Science CS1001.py

Lecture 16: Hashing continued
Finite and Infinite Iterators

Instructors: Daniel Deutch, Amiram Yehudai

Teaching Assistants: Michal Kleinbort, Amir Rubinstein

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

Hash table (motivation, reminder)

Looking for an efficient data structure for insert/delete/find

Recall that with python's list L, we may access the k'th item in O(1) time: simply L[k]

So, if we happen to know that all objects that will ever be inserted to our structure is accompanied with consecutive integer identifying numbers (keys) {0,…m-1} for some m, then we can use a list of size m and we gain O(1) complexity for all operations.

This is unrealistic! Object identifiers are e.g. ID number, maybe name, etc. which are not nearly consecutive numbers

We could in principle insert by ID numbers ,

is that a good idea?

L[65777888] = student1

L[32333333]=student2

Hash table (reminder)

A very large universe of keys, $\mathcal{U}$

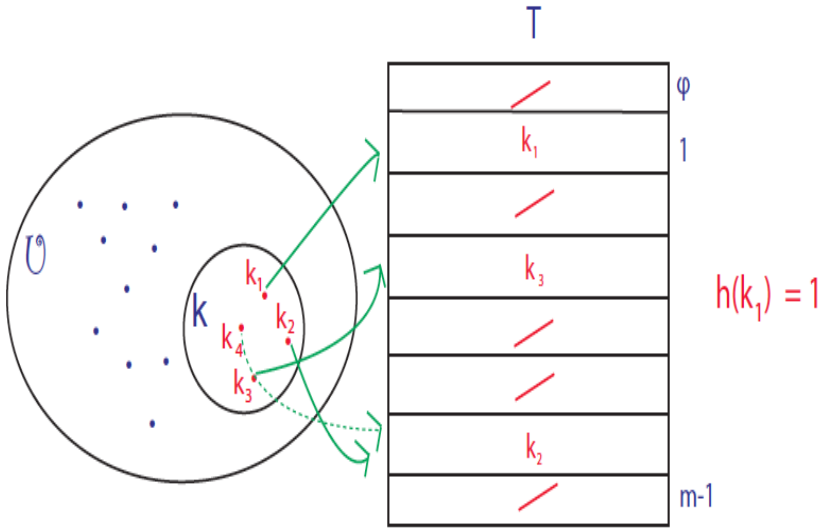
A much smaller set of keys, $\mathcal{K}$, containing up to n keys.

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 $\mathcal{K}$).

Example: h(k) = k mod m

In an ideal world, h is chosen to be one-to-one

In practice, we cannot, and will have collisions



 .Figure from MIT algorithms course, 2008

Hash tables (reminder, cont.)

The hash function h is chosen one and then used in all insert/delete/find operation for the structure

When inserting an object with key value k (e.g. student with ID/name k), it is inserted to the hash table (implemented by python's list) in location (index) h(k)

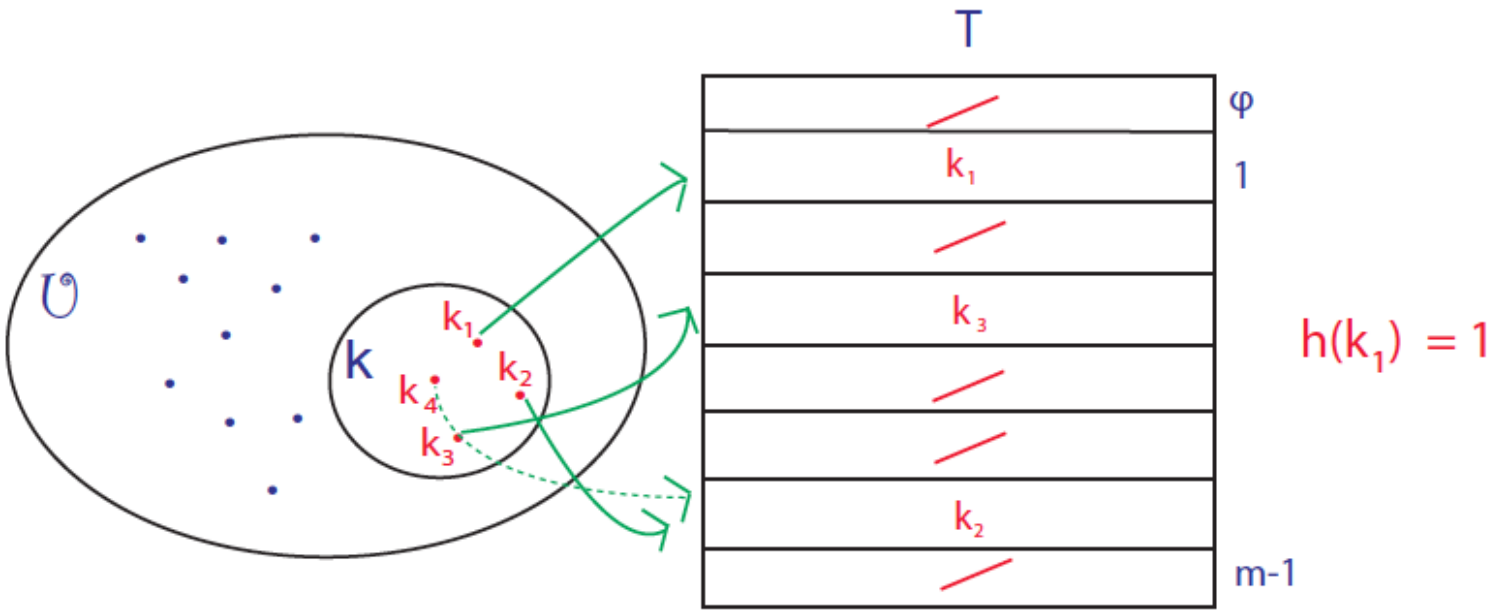
Then, when the user wishes to find an object with value k, we simply access the list at index h(k)

Problem: it may be the case that h(k)=h(k')

Hash functions

What is a good hash function?

- distributes elements uniformly in the table (and in particular covers the whole range).
- simple to calculate



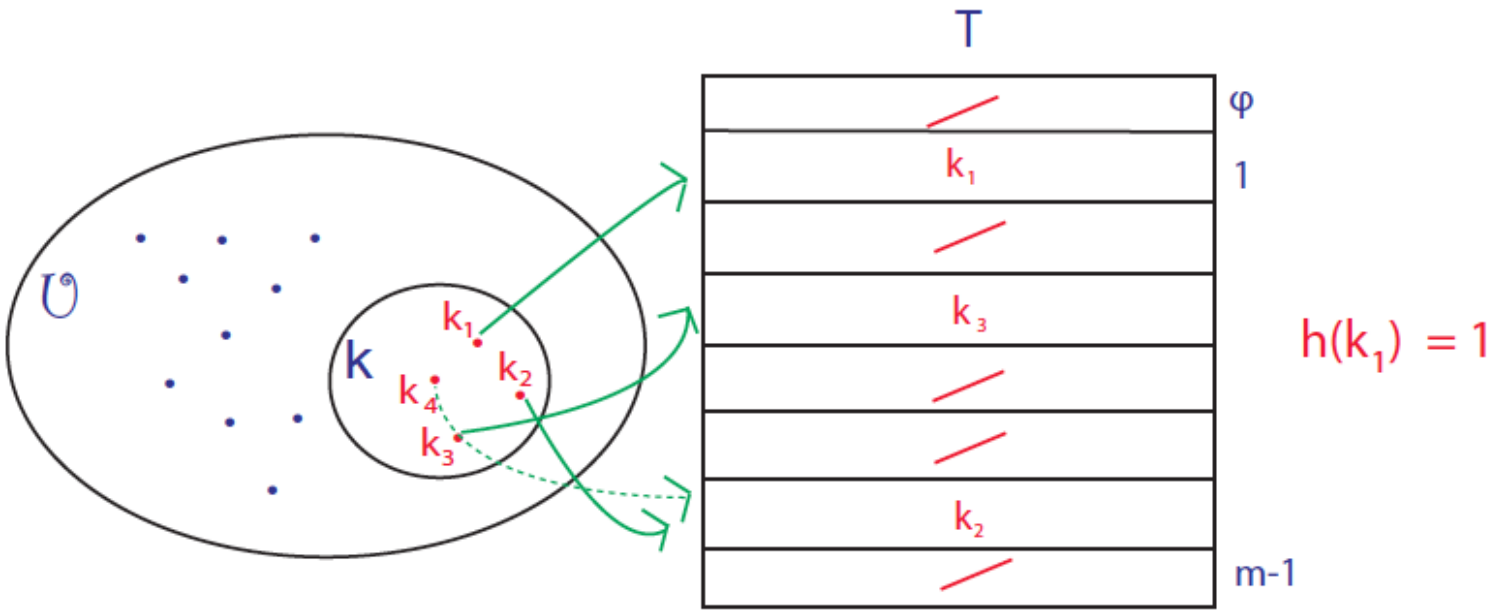
.Figure from MIT algorithms course, 2008

Hash functions

Are these hash functions good?

- h(n) = random.randint(0,n) (for ints)
- h(x) = 7 (for ints, strs,…)
- h(s) = $\left(\sum_i ord(s_i)\right)$ (for strs)
- h(n) = n%100 (for ints)

Ord returns the ascii code of a character



.Figure from MIT algorithms course, 2008

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.

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, $\ell = 2$, Naphtali, Zebulun and Joseph 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?

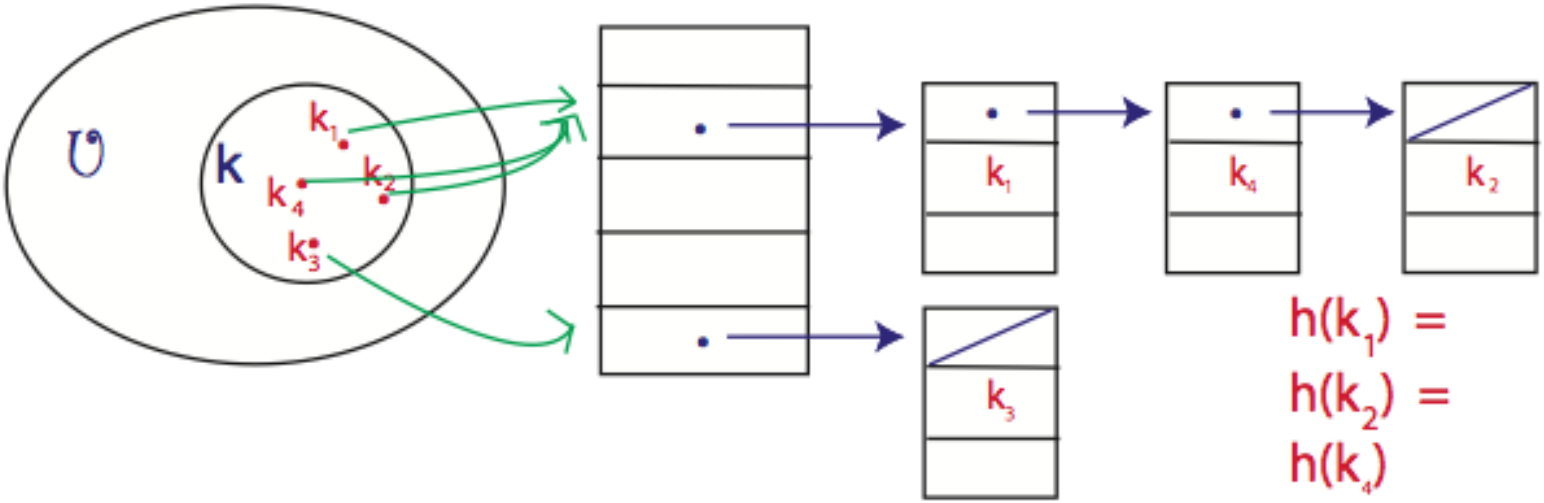
```
>>> [( name , hash_mod (name ,31)) for name in names ]
[( 'Reuben ', 18) , ('Simeon ', 30) , ('Levi ', 24) , ('Judah ', 23) ,
('Dan ', 16) , ('Naphtali ', 9), ('Gad ', 26) , ('Asher ', 23) ,
('Issachar ', 20) , ('Zebulun ', 5), ('Benjamin ', 18) , ('Joseph ', 25) ,
('Ephraim ', 1), ('Manasse ', 4)]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

Chaining for Dealing with Collisions



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)]
```

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] #access to hash table at hashed location
     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

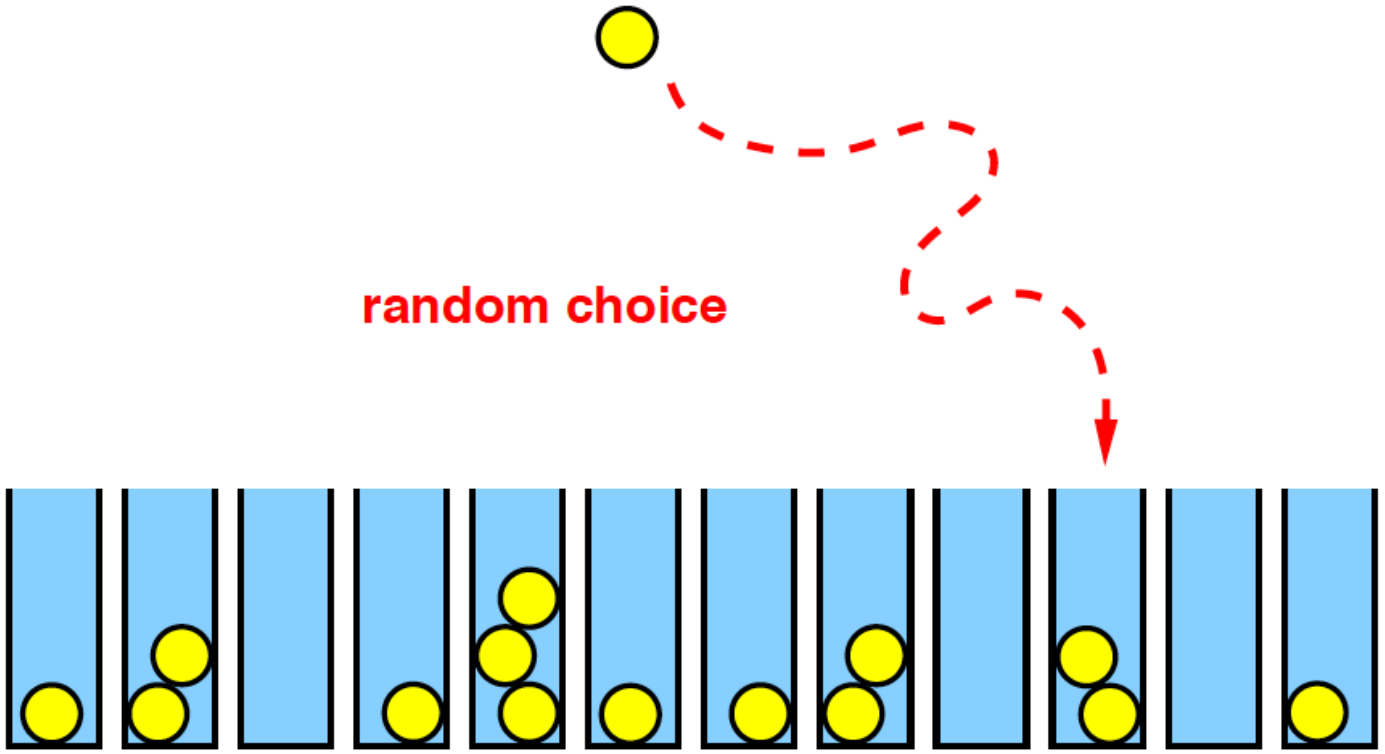
```
def bogus_hash_table (m):
      empty =[]
      return [ empty for i in range (m)]
# initial hash table , m empty entries

>>> 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]]
```

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).

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.

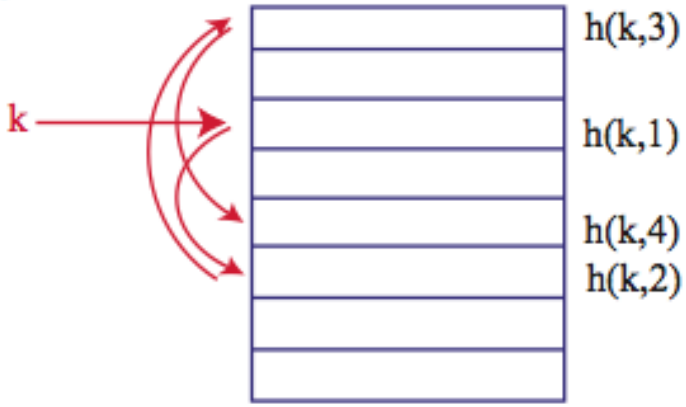
Collision Size

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.

Alternative Approach for Dealing with Collisions: Open Addressing (for reference only)

In open addressing, each slot in the hash table contains at most one item. This obviously implies that n cannot be larger than m . Furthermore, an item will typically not stay statically in the slot where it “tried'' to enter, or where it was placed initially. Instead, it may be moved a few times around.



Open addressing is important in hardware applications where devices have many slots but each can only store one item (eg. Fast switches and high capacity routers). It is also used in python dictionaries.

Hash Functions: Wrap Up

- Hash functions map large domains X to smaller ranges Y.
- Another Example:

  $h : \{0,1,\ldots,p^2\} \rightarrow \{0,1,\ldots,p-1\}$,
  defined by $h(x) = a \cdot x + b \bmod p$.

- Hash tables are extensively used for searching.
- If the range is larger than the domain, there will surely be collisions ($x \neq y$ with $h(x) = h(y)$). For example, in the example above, if $x_1 = x_2 \pmod p$ then $h(x_1) = h(x_2)$.
- If the range size is larger than the square root of the domain size, there will be collisions with high probability .
- A good hash function should create few collisions for most subsets of the domain ("few" is relative to size of subset).

And Now to Something Completely Different:
Finite and Infinite Iterators

Iterators and Generators

- Linked lists and Python's built-in lists (arrays) are two ways to represent a collection of elements. There are others, such as trees, hash tables, and more.
- It is desirable that functions that use the data as part of a computation should be as oblivious as possible to such internal representation, which may change over time .
- This general idea is captured in a concrete way by Python's iterators .
- Iterators will provide a generic access to a collection of items. So generic that it will even allow us to access an infinite collection (also known as stream)!
- Python's generators are tools to create iterators .

Iterables

An iterable is an object capable of returning its members one at a time. Examples of iterables include all sequence types (such as list, str , and tuple), some non-sequence types like dict and file, and objects of any user defined classes with an __iter__() or __getitem__() method.

(see http://docs.python.org/dev/glossary.html#term-iterable)

range is a special iterable class.

```
>>> a=range(10)
>>> type(a)
<class 'range'>
>>> a
range(0, 10)
>>> for i in a:
        …
```

Iterators

An iterator is an object representing a stream of data. Repeated calls to the iterator's __next__() method (or passing it to the built-in function next()) return successive items in the stream. When no more data are available a StopIteration exception is raised instead. At this point, the iterator object is "exhausted", and any further calls to its __next__() method just raise StopIteration exception again.

(see http://docs.python.org/dev/glossary.html#term-iterator)

```
>>> it = iter([0,1,2])
>>> next(it)
0
>>> next(it)
1
>>> next(it)
2
>>> next(it)
Traceback (most recent call last):
    File "<pyshell#26>", line 1, in <module>
        next(it)
StopIteration
```

Iterables and Iterators

We can create an iterator by calling the function iter with an iterable object argument (like list, tuple, str, dict, range , etc.) This function does not modify the original iterable object. In fact, when we loop over an iterable using for, under the hood an iterator is created first, and then the items are called, one by one, using next() .

```
>>> table={"benny":72,"rani":82,"raanan":92}
>>> next(table)
Traceback (most recent call last):
    File "<pyshell#13>", line 1, in <module>
        next(table)
TypeError: dict object is not an iterator
>>> it = iter(table)
>>> next(it)
'rani'
>>> next(it)
'benny'
```

Iterables and Iterators, cont.

```
>>> next(it)
'raanan'
>>> next(it)
Traceback (most recent call last):
    File "<pyshell#18>", line 1, in <module>
        next(it)
StopIteration

>>> table= {"benny":72,"rani":82,"raanan":92}
>>> for key in table:              # an iterator is created
        print(key)                 # “under the hood”
                                    # more details later
rani
benny
raanan
```

Iterables and Iterators, cont.

As we see from this example, a dictionary (when transformed into an iterator), returns the keys one by one.
Files return the lines one by one, etc.

We can turn an iterator into a list as well. This list will reflect the current state of the iterator, not its original state.

```
>>> table = {"benny":72,"rani":82,"raanan":92}
>>> it = iter(table)
>>> next(it)
'rani'
>>> list(it)
['benny', 'raanan']
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#82>", line 1, in <module>
    next(it)
StopIteration
```

Thou Shalt Not Modify an iterable during Iteration

If we add or remove elements from an iterable during iteration, strange things may happen. For example

```
>>> elems = ['a','b','c']
>>> for e in elems:
    print(e)
    elems.remove(e)

a
c
>>> elems
['b']
>>>
```

adapted from

http://unspecified.wordpress.com/2009/02/12/thou-shalt-not-modify-a-list-during-iteration/

Iterables, Iterators, and Generators: More Examples

```
>>> mylist = [x for x in range(10**8)]
>>> it1 = iter(mylist)
>>> it2 = (x for x in range(10**8))

>>> type(mylist)
<class 'list'>
>>> type(it1)
<class 'list_iterator'>
>>> type(it2)
<class 'generator'>

>>> # mylist
        # typing this without the comment will clobber your screen
        # and most likely will cause your Python shell to crash
>>> it1
<list_iterator object at 0x17027d0>
>>> it2
<generator object <genexpr> at 0x1704f08>
```

Iterables, Iterators, and Generators, cont.

it1 and it2 were iterators representing the first 10^8 integers.

These integers can fit in just under 1GB RAM. But, can we have iterators representing even more

items?

```
>>> it3 = (x for x in range(2**100))
>>> next(it3)
0
>>> next(it3)
1
```

An iterable with 2^{100} elements will not fit in Amazon, Google, and NASA computers, even if taken together.

Iterators and generators represent streams, but produce only one element at a time. Therefore, there is no problem representing a 2^{100} long stream!

Generators for Infinite Streams

In fact, there is no problem representing streams with countably many elements.

To do that, we will introduce generator functions.

So far, our functions contained no state, or memory. Successive calls to the function with the same arguments produced the same results (assuming the function does not refer to a global variable, which may have changed). This is now going to change .

```
def naturals():
    """ a generator for all natural numbers """
    n=1
    while True:
        yield n
        n+=1
```

Generators for Infinite Streams, cont.

A function that contains a yield statement is termed a generator function. When a generator function is called, the actual arguments are bound to function -- local formal argument names in the usual way, but no code in the body of the function is executed. Instead, a generator object is returned.

```
>>> naturals()
<generator object naturals at 0x16f60d0>
>>> nat = naturals()
>>> nat
<generator object naturals at 0x16f60a8
```

Generators, cont.

nat is a generator. To get its "returned value'', which is specified by the yield statement, we invoke next .

```
>>> next(nat)
1
>>> next(nat)
2
>>> [next(nat) for i in range(10)]
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

We see that nat has a state , which is retained, unchanged, between successive calls.

We can have additional instances of the same generator function.

```
>>> nat2 = naturals()
>>> next(nat2)
1
>>> next(nat)
13
```

Lazy Evaluation

In programming language theory, lazy evaluation or call-by-need is an evaluation strategy, which delays the evaluation of an expression until its value is actually required, and also avoids repeated evaluations by memoization (caching).

The "opposite'' of lazy actions is eager evaluation, sometimes known as strict evaluation. Eager evaluation is the evaluation behavior used in most cases in most programming languages.

Python's iterators and generators employ lazy evaluation. The next item is evaluated only when it is required, by means of executing next(). We remark that it would not be possible to handle finite but very large iterators/generators, or infinite iterators/generators, without the lazy evaluation mechanism.

Good old Scheme has a special syntax, enabling the delay and force of an evaluation of an expression. These make the use of very large and of infinite streams in Scheme possible.

A Fibonacci Numbers Generator

```
def fib():
    """ a generator for all Fibonacci numbers"""
    a, b = 0, 1
    while True:
        yield b
        a, b = b, a+b

>>> Fib = fib()
>>> Fib
<generator object fib at 0x1704fa8>
```

Again, Fib is a generator, so to get its “returned value'', which is specified by the yield statement, we invoke next() .

```
>>> next(Fib)
1
>>> next(Fib)
1
>>> next(Fib)
2
>>> [next(Fib) for i in range(10)]
[3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
```

Execution Specification

When a yield statement

yield expression_list

is encountered, the state of the function is frozen , and the value of expression_list is returned to the caller of __next__() .

By "frozen" we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, and the internal evaluation stack: enough information is saved so that the next time next() is invoked, the function can proceed exactly as if the yield statement were just another external call.

(see http://www.python.org/dev/peps/pep-0255/)

Merging Sorted, Infinite Iterators

Suppose iter1 and iter2 are sorted iterators, and both are infinite . We wish to produce a new sorted iterator which is the merge of both.

```
def merge(iter1,iter2):
    """ on input iter1, iter2, two infinite sorted iterators,
    produces the sorted merge of the two iterators """“

    left = next(iter1)
    right = next(iter2)
    while True:
        if left < right:
            yield(left)
            left = next(iter1)
        else:
            yield(right)
            right = next(iter2)
```

Merging Sorted, Infinite iterators: Execution

```
>>> nat1 = naturals()
>>> nat2 = naturals()
>>> nat3 = merge(Nat1, Nat2)
```

nat3 , too is a generator, so to get its “returned value'', which is specified by the yield statement, we invoke next .

```
>>> next(nat3)
1
>>> next(nat3)
1
>>> next(nat3)
2
>>> next(nat3)
2
>>> [next(nat3) for i in range(10)]
[3, 3, 4, 4, 5, 5, 6, 6, 7, 7]
```

An Attempt to Merge Sorted, Finite iterators

Should the iterators in merge really be infinite ?

```
>>> nat1 = natural()
>>> nat2 = (n-2 for n in range(3))
>>> nat3 = merge(Nat1,Nat2)
>>> next(nat3)
-2
>>> next(nat3)
-1
>>> next(nat3)
0
>>> next(nat3)
Traceback (most recent call last):
    File "<pyshell#48>", line 1, in <module>
        next(Nat3)
    File "/Users/benny/Documents/InttroCS2011/Code/intro17/lecture17.py",
line 30, in merge
        right = next(iter2)
StopIteration
```

What went wrong is that the merged iterator was not yet exhausted, yet one of the arguments to merge, Nat2 was exhausted. The merging procedure still invoked next(iter2) . This has caused a StopIteration error.

Handling Errors: try and except

Python provides an elaborate mechanism to handle run time errors.
For example, division by zero causes a ZeroDivisionError .

```
>>> 5/0
Traceback (most recent call last):
    File "<pyshell#37>", line 1, in <module>
        5/0
ZeroDivisionError: int division or modulo by zero
```

Such errors disrupt the flow of control in a program execution.
We may want to detect such error and allow the flow of control to continue. This may not be so important in the small programs written in this course, but becomes meaningful in large software projects.
Python enables such detection, using the keywords try and except.

Handling Errors: try and except: example

```
def division(a,b):
    try:
        return a/b
    except ZeroDivisionError:
        print("division by zero")
```

Let us now apply this function in two different cases:

```
>>> division(5,6)
0.8333333333333334

>>> division(5,0)
division by zero
```

We will employ this error handling mechanism to enable merging any non-empty sorted iterators, finite or infinite .

More on try and except

The example in the previous slide is not so good – we can solve this problem with an if statement. The following example shows a situation where we would need to write many if statements, so try/except is better

```
def compute(…):
    try
        # a long computation, with several steps
        # that may cause  zero divide
    except ZeroDivisionError:
        print("division by zero")
```

We will use try/except when it is either impossible or expensive to check for the condition in advance. Example – inverting a matrix, and detecting singularity.

We can have multiple except clauses; a list of exceptions to be handled in each clause; and the last clause may omit exception names (to handle all others)

For loop

We mentioned that a for loop over an iterable using for, actually uses an iterator. We now show an example:

```
>>> elems = ['a','b','c']
```

```
>>> for e in elems:
    print(e)

a
b
c
```

Is the same as

```
>>> it = iter(elems)
>>> while True:
    try:
        print(next(it))
    except StopIteration:
        break

a
b
c
```

Merging Any Non-Empty, Sorted iterators

```
def merge3(iter1,iter2):
    """ on input iter1, iter2, two non-empty sorted iterators, not
     necessarily infinite, produces sorted merge of the two iterators """

    left=next(iter1)
    right=next(iter2)
    while True:
        if left<right:
            yield(left)
            try:
                left=next(iter1)
            except StopIteration:        # iter1 is exhausted
                yield(right)
                remaining=iter2
                break
```

merge3 : cont.

```
    else:
        yield(right)
        try:
            right=next(iter2)
        except StopIteration:        # iter2 is exhausted
            yield(left)
            remaining=iter1
            break
# end of the while loop
for elem in remaining:    # protects against StopIteration
    yield(elem)
```

Merge3: Examples of Executions

```
>>> iter1=(x**2 for x in range(4))
>>> iter2=natural()
>>> merged=merge3(iter1,iter2)
>>> [next(merged) for i in range(14)]
[0, 1, 1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10]

>>> iter1=(x**2 for x in range(5))
>>> iter2=(x**3 for x in range(6))
>>> merged=merge3(iter1,iter2)
>>> [next(merged) for i in range(11)]
[0, 0, 1, 1, 4, 8, 9, 16, 27, 64, 125]
```

Finally, lets see what happens when the original iterators/generators are not sorted .

```
>>> iter1=((-1)**x*x**2 for x in range(5))
>>> iter2=(x**3 for x in range(6))
>>> merged=merge3(iter1,iter2)
>>> [next(merged) for i in range(11)]
[0, 0, -1, 1, 4, -9, 8, 16, 27, 64, 125]
# garbage in, garbage out
```