Computer Science 1001.py, Lecture 13[†]

Floating Point vs. Rational Numbers
Hash Functions and Hash Tables

Instructor: Benny Chor

Teaching Assistant (and Python Guru): Rani Hod

School of Computer Science
Tel-Aviv University
Fall Semester, 2011/12
http://tau-cs1001-py.wikidot.com

Thanks to Prof. Sivan Toledo for helpful suggestions and discussions.

Lecture 12 Highlights

- The Newton–Raphson iteratative root finding method.
- A geometric interpretation.

- Floating point (bounded precision) numbers.

Lecture 13, Plan

- Defining a class for (infinite precision) rationals.
- Newton–Raphson iteration for floating point vs. infinite precision rationals.

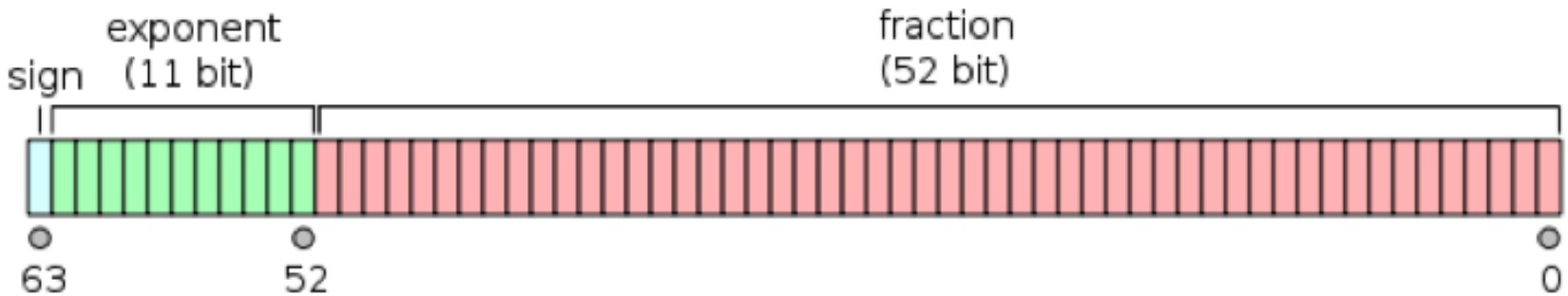
- The dictionary problem (find, insert, delete).
- Hash functions and hash tables.

Representation of Floating Point Numbers

Suppose we deal with a machine with 64 bit words. A floating point number is typically represented by

$\texttt{sign} \cdot \texttt{fraction} \cdot 2^{\texttt{exponent}}$.

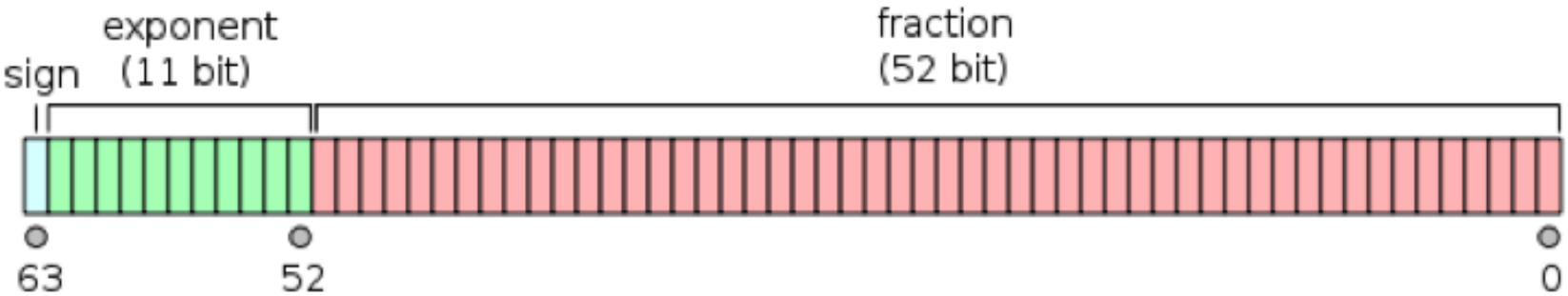
- The `sign` is ± 1 (1 indicates negative, 0 indicates non-negative).
- The `fraction` is a sum of negative powers of 2, represented by 52 bits, $0 \leq \texttt{fraction} \leq 1 - 2^{-52}$.
- The exponent is an integer, represented in a sneaky way by 11 bits (one used for the sign), $-1023 \leq \texttt{exponent} \leq 1023$.



(figure from Wikipedia)

In particular, the largest floating point number in a 64 bit word is smaller than 2^{1024}, and larger than 2^{1023}.

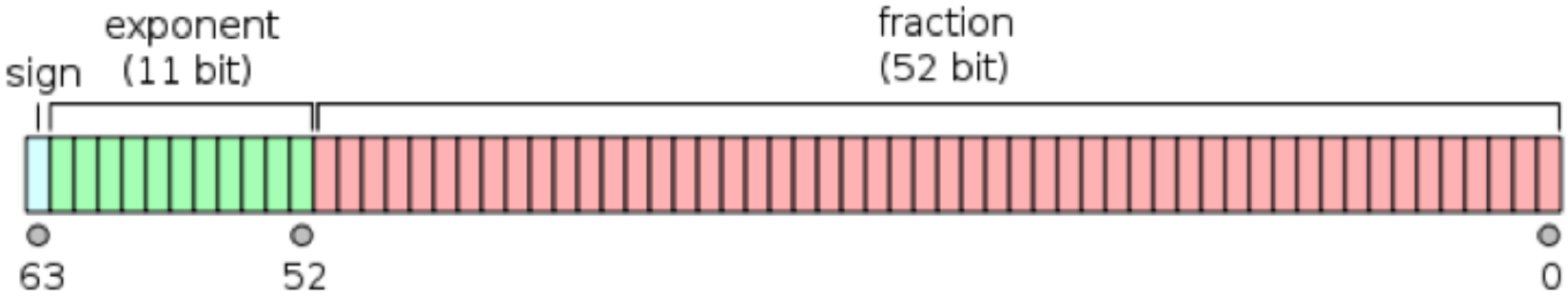
Representation of Floating Point Numbers



In particular, the largest floating point number in a 64 bit machine is smaller than 2^{1024}, and larger than 2^{1023}. We can try looking for this maximum,

```
>>> sum (2.0**i for i in range(971,1024))
1.7976931348623157e+308
>>> sum (2.0**i for i in range(970,1024))
inf
```

Representation of Floating Point Numbers



We can also probe Python for exact values of various floating points constants (and even understand most of them).

```
>>> sys.float_info
sys.floatinfo(
max=1.7976931348623157e+308,
max_exp=1024,
max_10_exp=308,
min=2.2250738585072014e-308,
min_exp=-1021,
min_10_exp=-307,
dig=15,
mant_dig=53,
epsilon=2.220446049250313e-16,
radix=2,
rounds=1)
```

Representation of Floating Point Numbers: The Full Monty

The following (rather obscure) code displays all 64 bits of a floating point number. It is brought to you as a courtesy of the Python's guru in the course staff (hint: not your lecturer):

```
import  struct

def display_float(x):
    """ prints 64 bits in the representation of the float x """
    if isinstance(x,float):
        q, = struct.unpack('Q', struct.pack("d", x))
        full_bin="{:064b}".format(q)
        sign= full_bin[0]
        exponent_plus=full_bin[1:12]
        fraction=full_bin[12:]
        return sign+" "+exponent_plus+" "+fraction
    else:
        return None
```

The true exponent is determined by `exponent_plus - 1022`. This enables us to get both positive and negative values.

The Full Monty: Specific examples

```
>>> display_float(1.)
'0 01111111111
      0000000000000000000000000000000000000000000000000000'
>>> display_float(1/2)
'0 01111111110
      0000000000000000000000000000000000000000000000000000'
>>> display_float(1/4)

'0 01111111101
      0000000000000000000000000000000000000000000000000000'
>>> display_float(2.0)
'0 10000000000
      0000000000000000000000000000000000000000000000000000'
>>> display_float(0.3)
'0 01111111101
      0011001100110011001100110011001100110011001100110011'
```

For example, the `exponent_plus` for 1.0 is `01111111111`, and `int("01111111111",2)` is 1023.
Similarly, the `exponent_plus` for 1/4 is `01111111101`, and `int("01111111101",2)` is 1021.

Arithmetic of Floating Point Numbers

Multiplication is done by multiplying the two fractions, and adding the two exponents.

Addition is done by first representing both numbers with the same exponent, then adding the weighted fractions, then converting back so that the fraction is smaller than 1.

Division and subtraction are analogous to addition and multiplication, correspondingly.

Addition and subtraction often lead to substantial loss of precision.

```
>>> a=11111111111111111111111111111111111111111111111.0
>>> b=11111111111111111111111111111.0
>>> a/b
1.0000000000000001e+18
>>> a-b==a
True
```

Many of you have encountered such behavior while solving problem 1 of assignment 3.

Relaxed Equality for Floating Point Arithmetic

We may solve some non intuitive issues with floating point numbers by redefining equality to mean equality up to some epsilon. For example

```
def float_eq(num1,num2,epsilon=10**(-10)):
    """ re-defines equality of floating point numbers """
    assert(isinstance(num1,float) and isinstance(num2,float))
    return abs(num1-num2)<epsilon
```

This indeed solves one problem

```
>>> float_eq(0.1+0.1+0.1,0.3)
True
```

But now this new equality relation is not transitive

```
>>> float_eq(0.,2*11**(-10))
True
>>> float_eq(2*11**(-10),4*11**(-10))
True
>>> float_eq(0.,4*11**(-10))
False
```

Floating Point Arithmetic vs. Rational Arithmetic

So, no matter how we turn it around, floating point arithmetic introduces some challenges and problems we are not very used to in "everyday life".

Questions:

1. Can we do something about this?
2. Should we bother?

Floating Point Arithmetic vs. Rational Arithmetic

So, no matter how we turn it around, floating point arithmetic introduces some challenges and problems we are not very used to in "everyday life".

Questions:

1. Can we do something about this?
2. Should we bother?

Answers:

1. Yes, we can! (Unbounded precision, rational arithmetic.)
2. Usually not: Typically the results of rational (unbounded precision) arithmetic for numerical computations are very similar to results from floating point arithmetic. Yet rational arithmetic is not for free – huge numerators and denominators tend to form, slowing down computation significantly, for no good reason.
3. Still, in some singular (and fairly rare) cases, numerical computations can be unstable, and the outcomes of floating point vs. rational arithmetic can be very different.

Rational Arithmetic in Python

Python comes in with unbounded precision integers (`type int`). We have secretly defined an unbounded precision Rational Class (and you will do so in a forthcoming assignment).

A rational number is nothing but a numerator and a denominator, both integers. The denominator cannot be zero.

Such class should support at the very least the four basic unbounded precision arithmetic operations (addition, subtraction, multiplication, and division), as well as the relations equal (==), and greater than (>).

Rational Class in Python – Some Examples

```
>>> a=Rational(43,58)
>>> a
43/58
>>> b=Rational(65,90)
>>> b
13/18
>>> a+b
382/261
>>> a/b
387/377
>>> a==b
False
>>> float(a*b)
0.5354406130268199
>>> c=Rational(0,6)
>>> c
0/1
>>> c/a
0/1
>>> b/c
Traceback (most recent call last):
        ### output truncated  for lack of space
    assert type(n)==int and type(d)==int and d!=0
AssertionError
```

Rational vs. Float Point Arithmetic: NR as a Benchmark

We will modify our NR procedure, so that it concurrently runs two executions of Newton-Raphson, starting from the same initial point, x_0.

One execution will use floating point numbers. The other will employ rational arithmetic, using the `Rational` class and methods.

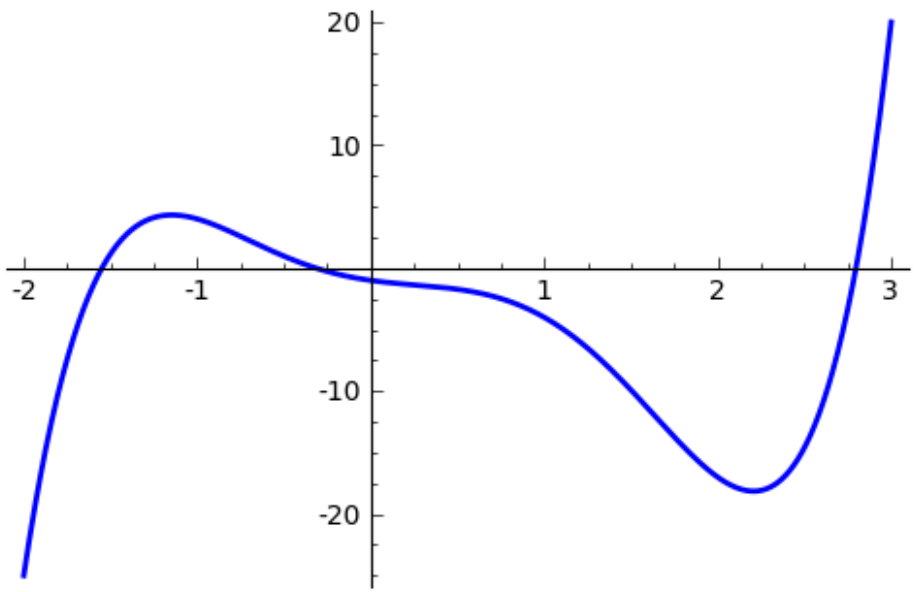
We will examine whether the unbounded precision makes a substantial difference.

Both executions will start from the initial point 0. The "rational thread" will use `r0=Rational(0,1)`. The "floating point thread" will use `r0=float(r0)`.

Rational vs. Float Point Arithmetic: NR as a Benchmark (2)

We will use the function $f(x) = x^5 - 2x^4 - 3x^3 + 3x^2 - 2x - 1$, whose derivative is $g(x) = 5x^4 - 8x^3 - 9x^2 + 6x - 2$. Its plot is shown below.

This is a fairly tame, degree 5 univariate polynomial. The plot shows three real roots.



The example is taken from Trefethen and Bau's book, "Numerical Linear Algebra", Exercise 13.4. on page 101.

Rational Arithmetic for NR: Code

```
def R_NR(func,deriv,epsilon=Rational(1,10)**8,n=8,r0=Rational(0,1)):
    """ Similar to NR, only it runs a sequence of length n of
    rational numbers, r_i. R_NR returns the resulting sequences r_i

    ration=["*"]*n
    ration[0]=r0

    for i in range(n-1):
        if abs(deriv(ration[i]))<epsilon:
            break
        else:
            ration[i+1]=ration[i]- func(ration[i])/deriv(ration[i])

    return ration
```

Floating Point Arithmetic for NR: Code

```
def F_NR(func,deriv,epsilon=10**(-8),n=8,f0=0.):
    """ Similar to NR, only it runs a sequence of length n
    of floating point numbers, f_i.
    F_NR returns the resulting sequences f_i """

    floats=["*"]*n
    floats[0]=float(f0)

    for i in range(n-1):
        if abs(deriv(floats[i]))<epsilon:
            break
        else:
            floats[i+1]=floats[i]- func(floats[i])/deriv(floats[i])
            yr=func(floats[i+1])

    return floats
```

Convergence of x-Values: Rational vs. Floating Point

```
>>> floats=F_NR(lambda x: x**5-2*x**4-3*x**3+3*x**2-2*x-1,
                          lambda x:5*x**4-8*x**3-9*x**2+6*x-2)
>>> ration=R_NR(lambda x: x**5-2*x**4-3*x**3+3*x**2-2*x-1,
                          lambda x:5*x**4-8*x**3-9*x**2+6*x-2)s
>>> for i in range(8):
        print ("i=",i,float(ration[i]),floats[i])
        print()
i= 0 0.0 0.0

i= 1 -0.5 -0.5

i= 2 -0.336842105263 -0.336842105263

i= 3 -0.315728448396 -0.315728448396

i= 4 -0.315301162703 -0.315301162703

i= 5 -0.315300986459 -0.315300986459

i= 6 -0.315300986459 -0.315300986459

i= 7 -0.315300986459 -0.315300986459
```

So far, the floating point displays of the two runs look identical.

Convergence of Function Values

```
>>> def f(x):
return x**5-2*x**4-3*x**3+3*x**2-2*x-1

>>> for i in range(8):
print ("i=",i,float(f(ration[i])),f(floats[i]))
print()
i= 0 -1.0 -1.0

i= 1 0.96875 0.96875

i= 2 0.0986450190239 0.0986450190239

i= 3 0.00191853843117 0.00191853843117

i= 4 7.90693465994e-07 7.90693466257e-07

i= 5 1.34537754922e-13 1.34559030585e-13

i= 6 3.89508143967e-27 0.0

i= 7 3.26483528197e-54 0.0
```

This indicates that under the rug, the unbounded precision sequence keeps converging to better and better approximations of a root.

Displaying the Rational Numbers

When comparing the values in the rationals vs. floating point sequences, we only displayed the `float()` of a rational. Is this a bug or a feature? Lets try

```
i= 0 0/1
i= 1 -1/2
i= 2 -32/95
i= 3 -11414146527/36151783550
i= 4 -437115663193076384403256764909499867587929989600855 36/
13863433279008761611840812755838900332126896609091862 5
i= 5 -724391479176820176129001381878925973035003883604754393117
804119434357926010580274469629922882064184585670017703551996316
651611596343634562735299921308664663139405767412052875538201240
642484300698212354536105198270689471522317606875456902898519837
650550434545296779 21/
229746023731575873333990816664320035147759847208021088660066874
783249488750988451982247975822897844718084679832592257179299176
854789444915362215689722609358654955195182168763169315683704659
081440024954196748041166750181397522783471619066874148005355642
10785107754125 0
```

Displaying the Rational Numbers

We have not continued with displaying the list of rationals, simply because the representation is too long

```
>>> len(str(ration[7].denom))
6662
```

Namely the denominator of `ration[7]` is 6662 decimal digits long.

Rational vs. Floating Point Numbers: Interim Conclusions

With respect to the Newton-Raphson root finding method on the specific polynomial function $f(x) = x^5 - 2x^4 - 3x^3 + 3x^2 - 2x - 1$, and the specific starting point $x_0 = 0$:

- Both sequences of rational and floating point numbers converged quickly to a root.
- The sequence of floating point numbers reached "equilibrium" after the fifth iteration.
- The sequence of rational numbers kept evolving. The function values kept getting closer to zero.
- Handling the sequence of rational required many more computational resources than its floating point counter part – both in terms of computing time and memory/space.
- For this specific application, this (using unbounded precision rationals) is not worth the effort.
- We might as well have used only floating point numbers in this numerical context.

Rational vs. Floating Point Numbers: General Conclusions

We will boldly jump to the following conclusion: Use just floating point numbers for all your numeric needs.

Rational vs. Floating Point Numbers: General Conclusions

We will boldly jump to the following conclusion: Use just floating point numbers for all your numeric needs.

Such blunt generalization from a single instance is unjustified on any scientific or experimental ground. There are cases of completely tame functions that become unstable / chaotic / wild when certain conditions hold. In such cases, there could be a huge difference between rational and floating point arithmetic.

You will even work on such case (matrix inversion using Gaussian elimination) in a forthcoming homework assignment. And the binary search in assignment 3 also required unbounded precision (but was not very tame).

However, as a rule of thumb, most applications you will encounter "normally" will not be of that form, and floating point arithmetic will be the way to go initially. These topics are covered in the Scientific Computing course given in TAU (recommended from the 3rd year).

And Now for Something Completely Different: Brazil

To prepare for the next context switch in our course, you may

- want to get acquainted with Harry Tuttle, Heating Engineer, at your service!
- see why small typos (Buttle instead of Tuttle) may be pretty bad for your health,
- listen to Django Reinhardt's Brazil,
- or simply watch Terry Gilliam's hilarious movie, Brazil (1985).

What Have We Done So Far

Among other things, we

- Learnt some Python (more than just "some", in fact).
- Saw how operations on large integers are relevant to primality testing and secret key exchange.
- Investigated the problem of root finding in numerical mathematics.
- Became familiar with floating point numbers as wel as unbounded precision rationals.

What Have We Done So Far

Among other things, we

- Learnt some Python (more than just "some", in fact).
- Saw how operations on large integers are relevant to primality testing and secret key exchange.
- Investigated the problem of root finding in numerical mathematics.
- Became familiar with floating point numbers as wel as unbounded precision rationals.

We are now going to have yet another context switch, and study how hash function and tables can be used for search in O(1) expected time.

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

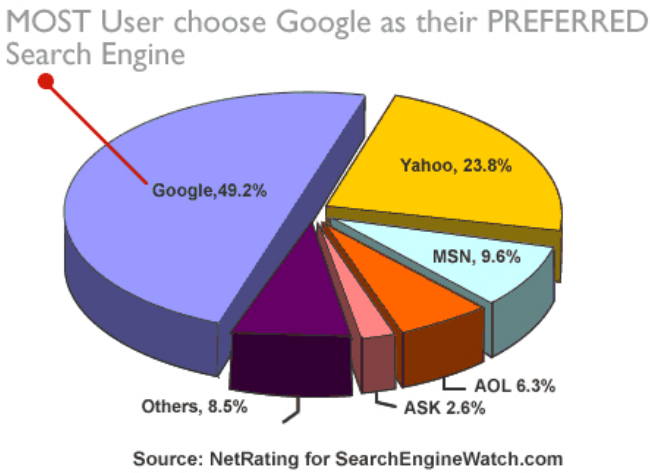
In computer science, hashing has multiple meaning, often unrelated. For example, universal hashing, perfect hashing, cryptographic hashing, and geometric hashing, have very different meanings. Common to all of them is a mapping from a large space into a smaller one.

Today, we will study hashing in the context of the dictionary problem.

Hash Functions, Hash Tables, and Search

(figure from www.searchengineguide.com/stoney-degeyter/comprehensive-guide-to-keyword-research.php)

And, while at that



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

Search (reminder from lecture 7)

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

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

In lecture 7 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 resort the whole list.

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 dictionaries.

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, and the keys are students' names.

We remark that in our setting, keys of different items need not be unique.

Other Dynamic Data Structure

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

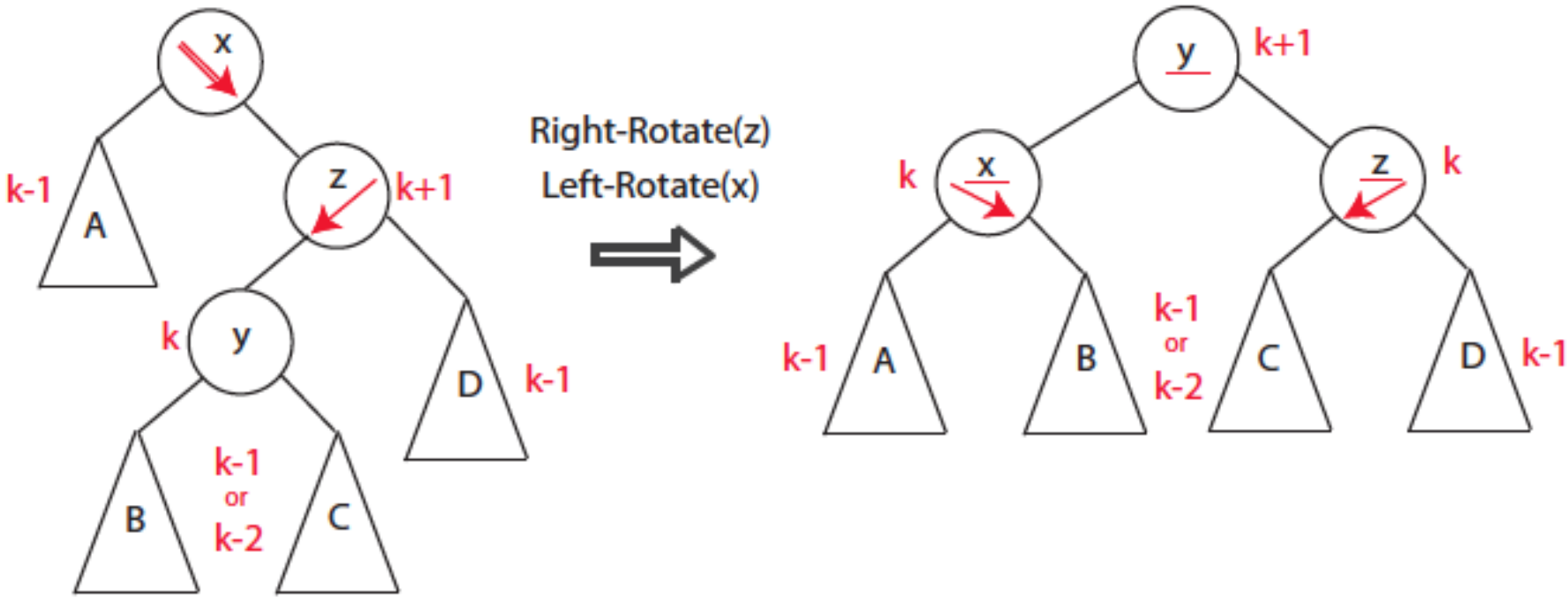
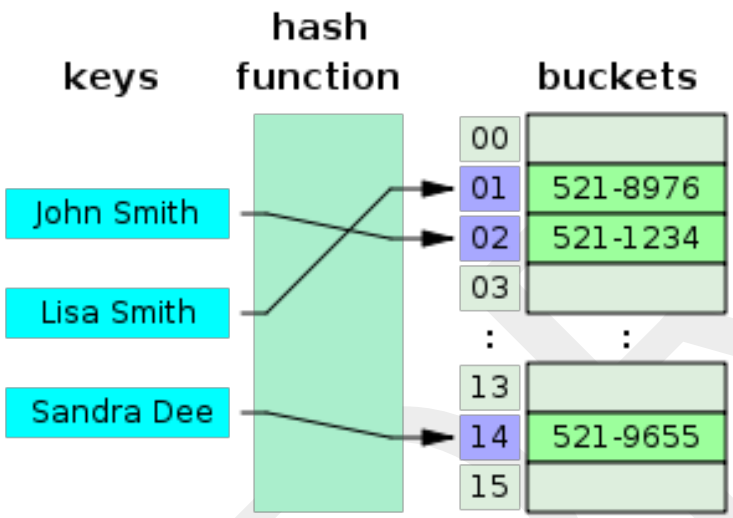


Figure from MIT algorithms course, 2008. Shows item insertion in an AVL tree.

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 essentially be achieved 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. So are Python's `sets`, which are simply dictionaries where entries are `keys` without `values`.

Python's dict vs. Our Planned Dictionary

As noted earlier, Python's dictionary, storing key:value pairs (keys should be immutable), supports efficient insert, delete, and search operations. It is indeed implemented using a hash table.

```
>>> table={"Or":51467286,"Barak":43052060,"Yuval":34430444}
>>> table["Or"]    # search
51467286
>>> table["Shady"]=36352520   # insertion
>>> table
{'Shady':36352520,'Barak':43052060,'Or':51467286,'Yuval':34430444}
>>> del table["Or"]   # deletion
>>> table
{'Shady':36352520,'Barak':43052060,'Yuval':34430444}
>>> table["Shady"]=5555555   # inserting existing key
>>> table
{'Shady':5555555,'Barak':43052060,'Yuval':34430444}
```

We see that Python's dict does not support having different items with the same keys (it keeps only the most recent item with a given key).

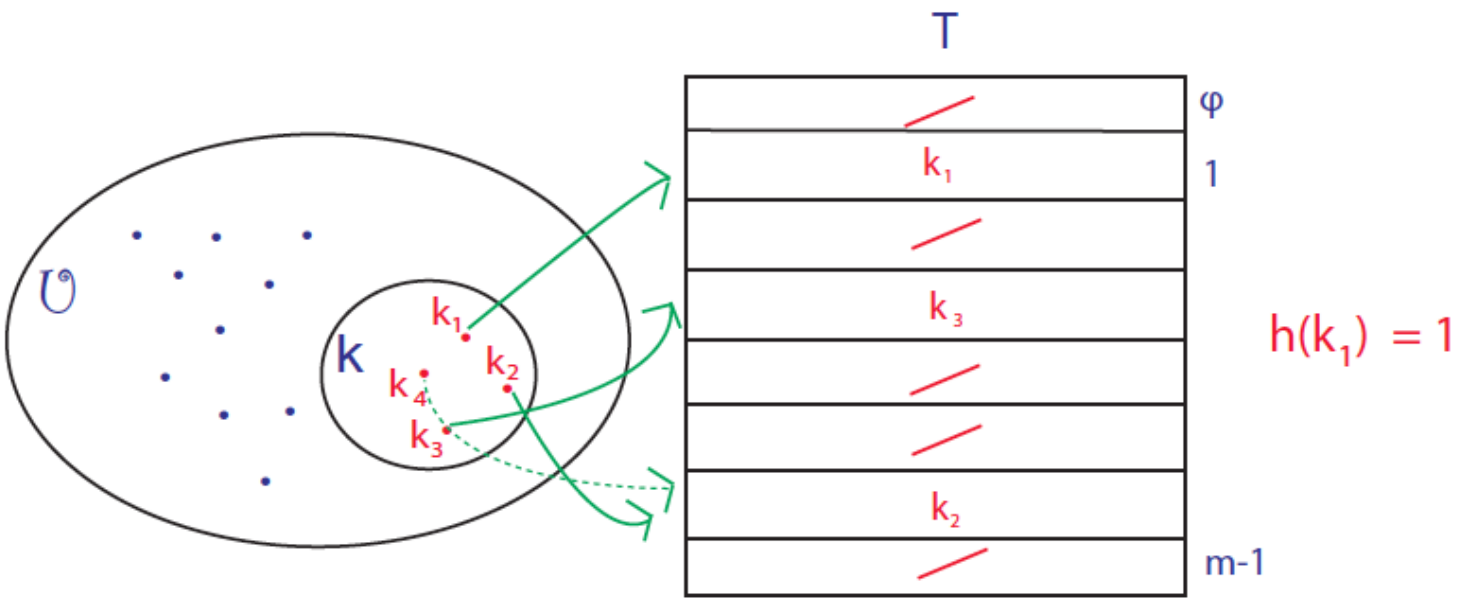
To understand what is really going on, we cannot merely use dict.

Dictionary Setting

- There is a very large universe of keys, $\mathcal{U}$.
- Within this universe, we should process a set of keys, $\mathcal{K}$, containing up to n keys.
- The keys in the set $\mathcal{K}$ are initially unknown.
- We wish to map the set $\mathcal{K}$ to a table, $T = \{0, \ldots, m-1\}$ of size m, where $m \approx n$.
- The mapping is by a (fixed) hash function, $h : \mathcal{U} \mapsto \mathcal{T}$.
- Note that h does not depend on $\mathcal{K}$.

Dictionary Setting

- There is a very large universe of keys, $\mathcal{U}$.
- Within this universe, we should process a set of keys, $\mathcal{K}$, containing up to n keys.
- The keys in the set $\mathcal{K}$ are initially unknown.
- We wish to map the set $\mathcal{K}$ to a table, $T = \{0, \ldots, m-1\}$ of size m, where $m \approx n$.
- The mapping is by a (fixed) hash function, $h : \mathcal{U} \mapsto \mathcal{T}$.
- Note that h does not depend on $\mathcal{K}$.



Implementing Insert, Delete, Search

There universe of all keys, $\mathcal{U}$, is much larger than the set of actual keys, $\mathcal{K}$, whose size is up to n.

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

Implementing Insert, Delete, Search

There universe of all keys, $\mathcal{U}$, is much larger than the set of actual keys, $\mathcal{K}$, whose size is up to n.

- Given an item with key $k \in \mathcal{U}$.
- Compute $h(k)$ and check if in T (this is search).
- If not, can insert item to cell $h(k)$ in T.
- If it is, can delete item from cell $h(k)$ in 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 $\theta(n)$ steps, rather than $O(1)$ steps.

Luck, and Distribution of Hashed Values

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.

Indeed, we are really unlucky, h will map all n keys in $\mathcal{K}$ to the same value. Sorting this out will take $\theta(n)$ steps rather than $O(1)$ steps.

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.

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 $h(k), k \in \mathcal{K}$ are distributed in $\mathcal{T}$ at random. What is the probability that there exist a collision? What is the size of the largest colliding set (a set $\mathcal{S} \subset \mathcal{K}$ 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.

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 $n \approx \sqrt{\pi m/2}$, 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 ($m = 365$ here, and $\sqrt{\pi \cdot 365/2} = 23.94$).

Thus if our set of keys is of size $n \approx \sqrt{\pi m/2}$, two keys are likely to create a collision.
It is also known that if $n = m$, the size of the largest colliding set is $\ln n / \ln \ln n$.

Maximum Collision Size

More generally, 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 $n = m^{1-\varepsilon},\ 0 < \varepsilon < 1$, the expected maximal capacity (in a single slot) is $O(1/\varepsilon)$.
- 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) is $n/m \ + \ \ln n / \ln \ln n$.

A Very Small Example

We'll construct a hash table with $m = 23$ entries, and insert the records of 13 students in it.

```
names=["Or","Yana","Amir","Roee","Noa","Gal","Barak",
       "Rina","Tal","Lielle","Shady","Yuval","Walt Disney"]

students_list=students(13)

for i in range(13):
    students_list[i].name=names[i]
```

The Hash Function

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

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

The Hash Function

Python comes with its own hash function, from everything 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 very random. However, what concerns us mostly in our context is that the range of Python's hash function is too large. To take care of this, we simply reduce its outcome modulo m, the size of the hash table (23 in our case).

```
def hash_mod(key,m=23):
    return hash(key) % m
```

Constructing the Hash Table: A Very Small Example

We'll construct a hash table with $m = 23$ entries. We'll insert 13 students' record in it and check how insertions are distributed, and in particular what the maximum number of collisions.

Our hash table will be a list with a fixed number of $m = 23$ 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 $\{0, 1, \ldots, 22\}$ (indices in the hash table). Given a student, we apply the hash function to its name, which is the key in our case.

Constructing the Hash Table: A Very Small Example

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 named `hash_mod`.
If the result is ℓ, we will map (the record of) this student to entry number ℓ in the hash table.

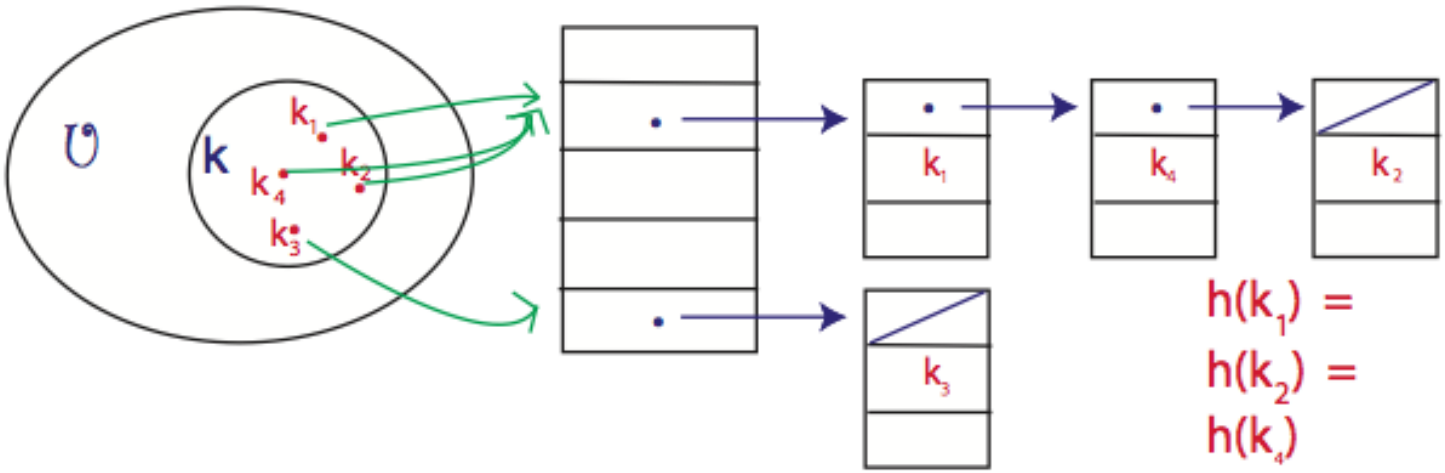
```
>>> print([(student.name,hash_mod(student.name)) \
        for student in students_list])
[('Or', 6), ('Yana', 12), ('Amir', 21), ('Roee', 14), ('Noa', 5),
('Gal', 10), ('Barak', 20), ('Rina', 10), ('Tal', 22),
('Lielle', 0), ('Shady', 13), ('Yuval', 4), ('Walt Disney', 2)]
```

In the example above, with $n = 13, m = 23$, we see that there is one collision: Both Gal and Rina are mapped to the same entry in the hash table, $\ell = 10$.

Question is, how shall we deal with such collisions?

Two Approaches for Dealing with Collisions

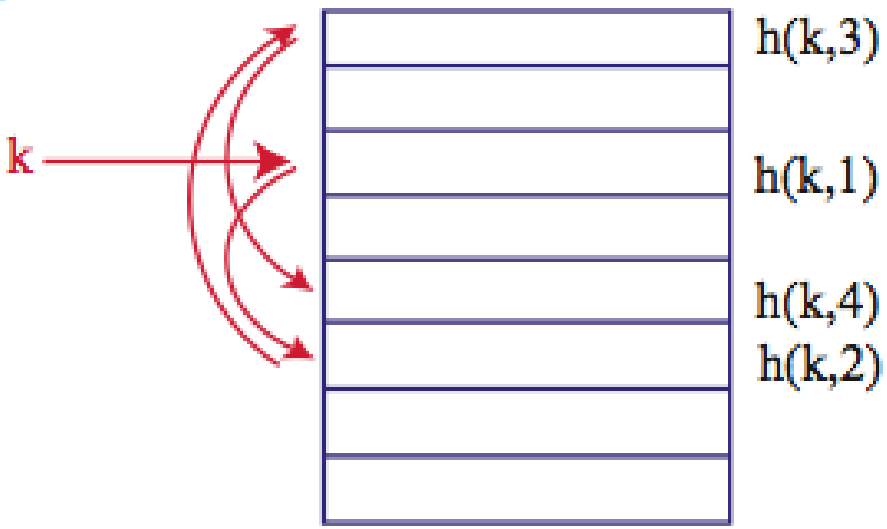
1) Chaining:



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.

Two Approaches for Dealing with Collisions

2) Open Addressing (each slot is populated by at most one item):



We will not discuss open addressing in this course.

However, we will later discuss a related approach, using more than one different hash functions (two, three, or four), known as cuckoo hashing.