Computer Science 1001.py, Lecture 13[†]

The Dictionary Problem
Hash Functions and Hash Tables

Instructors: Haim Wolfson, Amiram Yehudai
Teaching Assistants: Yoav Ram, Amir Rubinstein

School of Computer Science
Tel-Aviv University
Spring Semester, 2012-13
http://tau-cs1001-py.wikidot.com

Lecture 12 Highlights

- The Newton–Raphson iteratative root finding method.
- A geometric interpretation.
- Cases where Newton–Raphson succeed, and cases where it fails.
- Running Newton–Raphson in "floats mode" vs. "rational mode".

Lectures 1–12 Topics, a Bird's Eye View

- Integer arithmetic and applications (primality checking, cryptographic key exchange, Euclid's gcd algorithm).
- Recursion.
- Lambda expressions and higher order functions.
- Classes and methods.
- Numerical algorithms (numerical derivative and integral, root finding).

Lecture 13, Plan

- The dictionary problem (find, insert, delete).
- Python `hash` and `<class 'dict'>`.
- Hash functions and hash tables.

And Now to Something Completely Different:
Hash Functions, Hash Tables, and Search

"A loaf of bread," the Walrus said,
"Is what we chiefly need:
Pepper and vinegar besides
Are very good indeed–
Now if you're ready, Oysters dear,
We can begin to feed."

"But not on us!" the Oysters cried,
Turning a little blue.
"After such kindness, that would be
A dismal thing to do!"
"The night is fine," the Walrus said.
"Do you admire the view?

Through the Looking-Glass and What Alice Found There:
Lewis Carroll, 1871.

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

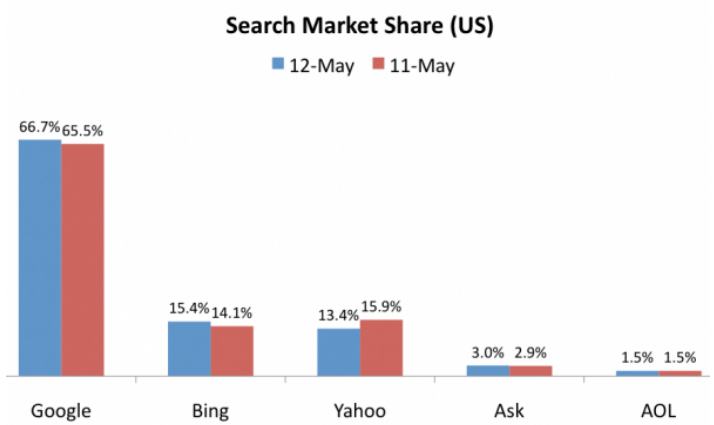
In computer science, hashing has multiple meaning, often unrelated. For example, universal hashing, perfect hashing, cryptographic hashing, and geometric hashing, have very different meanings. Common to all of them is a mapping from a large space into a smaller one.

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-flat-yahoo-hits-new-low-124519)

And, while at that



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

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 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 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 reorganize the whole list – requiring $O(n)$ 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 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, 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 time $O(\log n)$. They are fairly involved, and studied extensively in the data structures course.

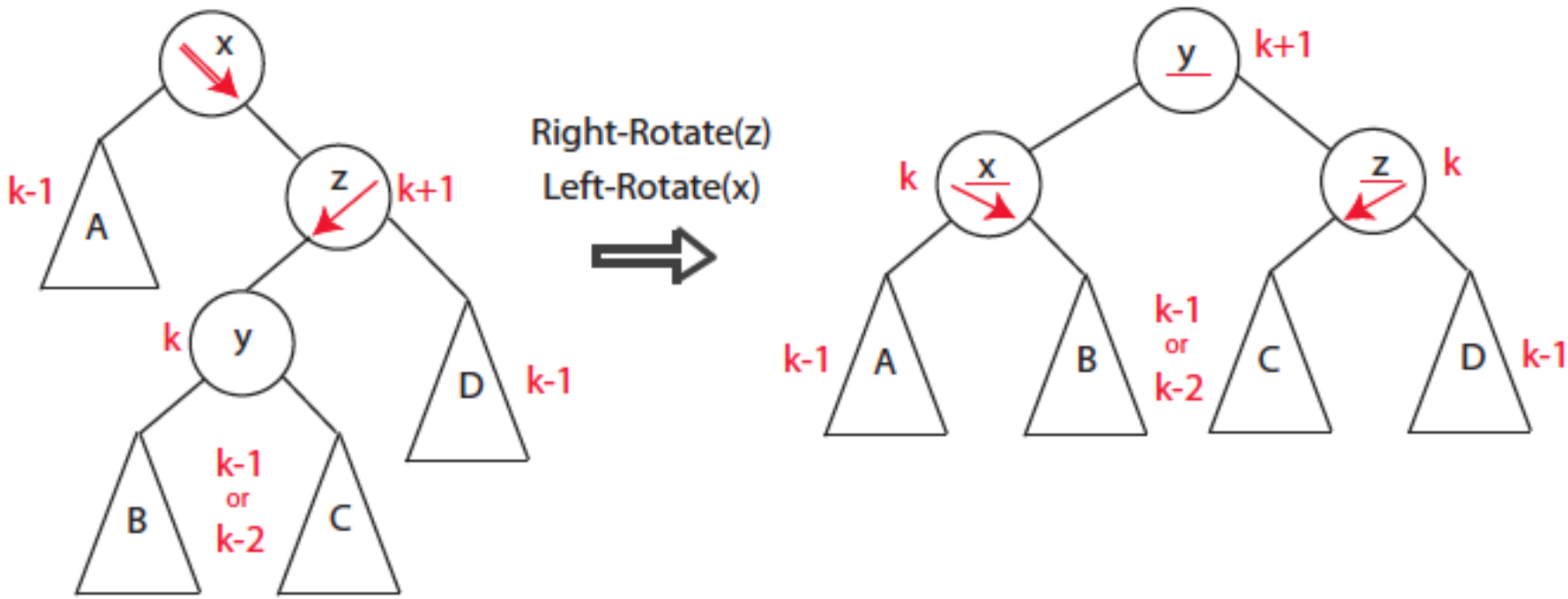
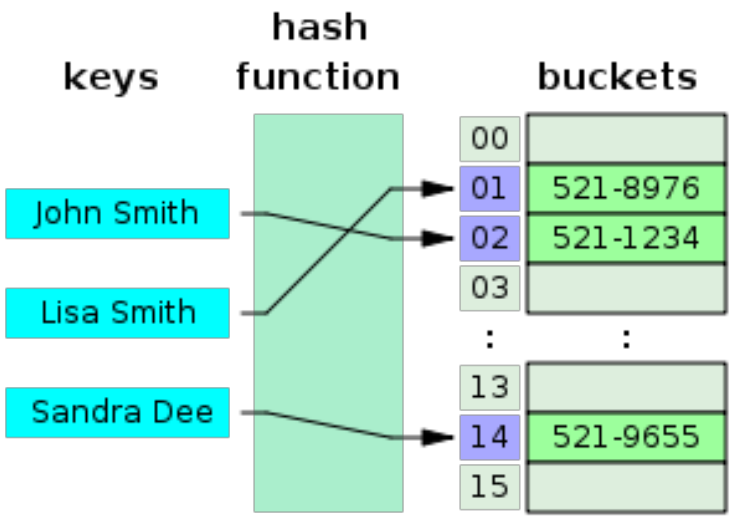


Figure from MIT algorithms course, 2008. Shows item insertion in an AVL tree.

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

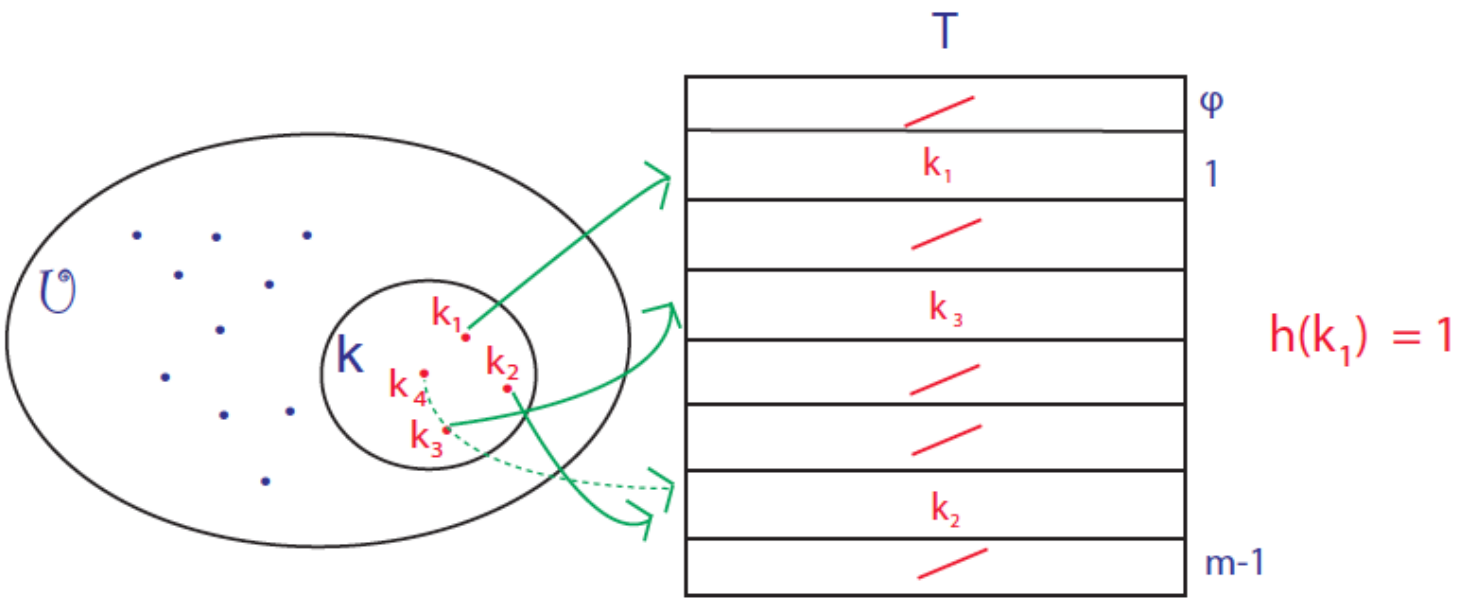
As noted earlier, Python's dictionary, storing key:value pairs (keys should be immutable), supports efficient insert, delete, and search operations. It is indeed implemented using a hash table.

```
>>> table={"Or":(51467286,92),"Barak":(43052060,85),
             "Roee":(34430444,23)}   # creating the initial table
>>> table["Or"]   # search
(51467286, 92)
>>> table["Shady"]
Traceback (most recent call last):  # output truncated for lack of
KeyError: 'Shady'
>>> table["Shady"]=(36352520,79)  # inserting a new key
>>> table
{'Shady':(36352520,79),'Or':(51467286,92),'Barak':(43052060,85),
              'Roee':(34430444,23)}
>>> del table["Or"]; del table["Barak"] # deleting two keys
>>> table
{'Shady':(36352520,79),'Roee':(34430444,23)}
>>> table["Shady"]=(5555555,81)  # re-inserting an existing key
>>> table
{'Shady':(5555555,81),'Roee':(34430444,23)}
```

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).

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

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.

- 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 $O(n)$ 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, if we are really unlucky, h will map all n keys in $\mathcal{K}$ to the same value. Sorting this out will take $O(n)$ 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 $h(k)$ 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 $\mathcal{S} \subset \mathcal{K}$ 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.

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 seems 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

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

```
>>> hash("Benny")
5551611717038549197
>>> 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'
```

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 (23 in our case).

```
def hash_p(key,p=23):
    return hash(key) % p
```

Constructing the Hash Table: A Very Small Example

We'll construct a hash table with $p = 23$ entries. We'll insert 14 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 $p = 23$ entries.

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 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_p(name)) for name in names]
[('Reuben', 7),('Simeon',0),('Levi',16),('Judah',2),('Dan',21),
('Naphtali',7),('Gad',6),('Asher',3),('Issachar',18),('Zebulun',15)
,('Benjamin',22),('Joseph',5),('Ephraim',3),('Manasse',14)]
```

In the example above, with $n = 14, m = 23$, we see that there is two collisions: Both Reuben and Naphtali are mapped to the same entry in the hash table, $\ell = 7$, while both Asher and Ephraim are mapped to the same entry in the hash table, $\ell = 3$.

Question is, how shall we deal with such collisions?

An Alternative Hash Functions

There is nothing sacred about Python's hash function. Here is an alternative, from strings and integers to non-negative integers, up to $2^{120} + 450$ (a prime number, naturally).

```
def str_to_int(string):
    """ converts a string to int, a sum of powers of 128,
    modulo  2**120+451"""
    if isinstance(string,str):
        partial_sum=0
        for i in range(len(string)):
          partial_sum=(128*partial_sum+ord(string[i]))%(2**120+451)
        return partial_sum
    else:
        return None

def benny_hash(input):
        """ a new hash, applicable to integers and strings """
        if isinstance(input,str):
            return (str_to_int(input))**2 % (2**120+451)
        elif isinstance(input,int):
            return input**2 % (2**120+451)
        else:
            return None
```

Python's Hash Function vs. Benny's Hash Function

Let us run the two functions on some arbitrary test values:

```
>>> hash(1),benny_hash(1)
(1, 1)
>>> hash(0),benny_hash(0)
(0, 0)
>>> hash(10000000),benny_hash(10000000)
(10000000, 100000000000000)
>>> hash(-468864544),benny_hash(-468864544)
(-468864544, 2198339606203279 36)
>>> hash("b"),benny_hash("b")
(-340864157, 9604)
>>> hash("cs1001.py"),benny_hash("cs1001.py")
(564361474, 131001634198281260281814593555 0572095)
```

We make `hash_mod` a high order function, taking the hash function as one of its arguments, with the default value being Python's hash.

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

>>> hash_mod(4567),hash_mod(4567,func=benny_hash)
(13, 8)
```

Alternative Hash Table: A Small Example

Given a student, we apply the alternative 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.

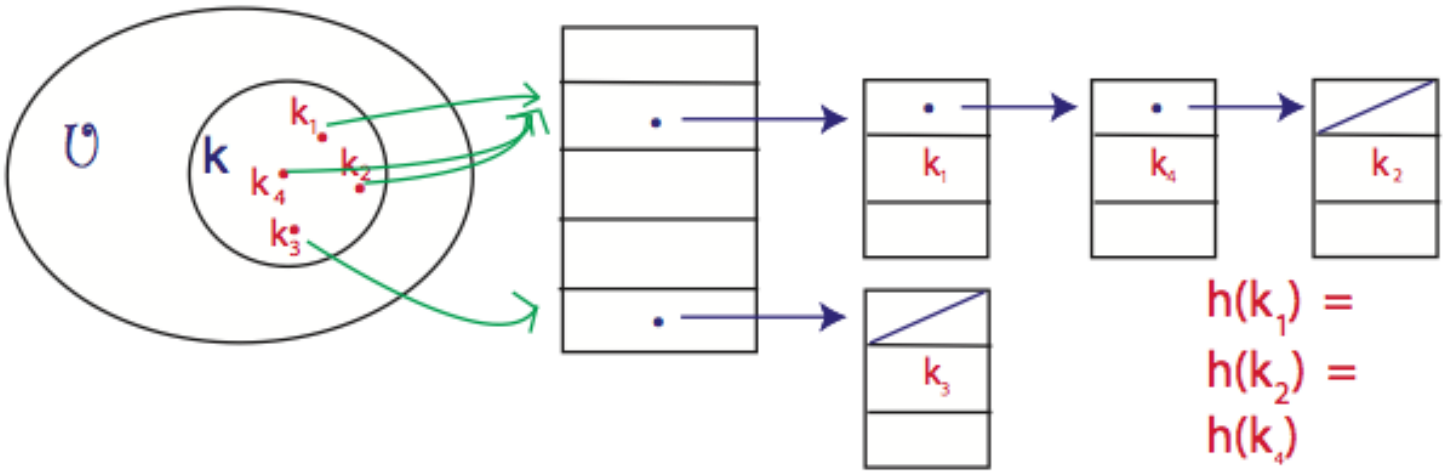
```
>>> [(name,hash_mod(name,func=benny_hash)) for name in names]
[('Reuben',3),('Simeon',9),('Levi',1),('Judah',4),('Dan',13),
('Naphtali',8),('Gad',9),('Asher',4),('Issachar',4),('Zebulun',13),
('Benjamin',4),('Joseph',3),('Ephraim',16),('Manasse',1)]
```

In the example above, with $n = 14, m = 23$, we see that there are four collision: Judah, Asher, Issachar, and Benjamin are all mapped to the same entry in the hash table, $\ell = 4$. Reuben and Joseph are mapped to $\ell = 3$. Levi and Manasse are both mapped to $\ell = 1$. Finally, Dan and Zebulun are are both mapped to $\ell = 13$

This hash function performs much much worse on this small example, compared to Python's built-in hash function.

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.

Constructing the Hash Table: A Very Small Example

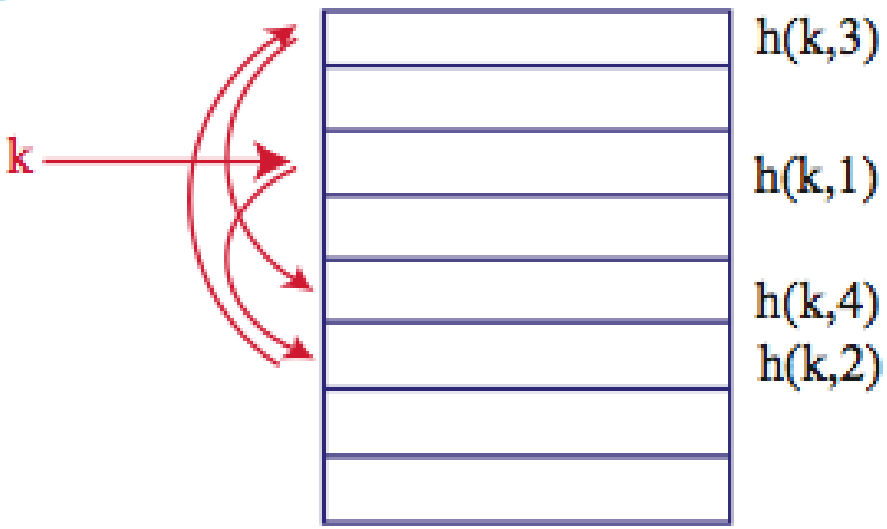
Initially, we'll insert 14 students' records in a hash table with $m = 23$ entries, using chaining. We'll check how insertions are distributed, and in particular what the maximum number of collisions.

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.

The hash function 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.

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.

A Short Detour: 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 $n \approx \sqrt{\pi m/2}$, 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 expected size of the largest colliding set is $\ln n / \ln \ln n$.

Collisions of Hashed Values (repeated)

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 $\mathcal{S} \subset \mathcal{K}$ 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.

Maximum 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 $O(1/\varepsilon)$.
- 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 $n/m \ + \ \ln n / \ln \ln n$.

Student Class, Revisited

We modify our Student class, by popular demand, to include grades in addition to IDs. FYI, Grades are uniformly distributed in [19,99].

```
class Student:
    def __init__(self):
        self.name = generate_name()
        self.id = random.randint(2*10**7,6*10**7)
        self.grade = random.randint(19,99)
        self.key=self.name  # anticipating dictionary ops
        self.value=self.id,self.grade # anticipating dictionary ops

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

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

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

>>> students(2)
[<Anst Irlmhqj, 24966788, 92>, <Icq Wifs, 26147350, 97>]
>>> students(2)
[<Iewsl Apzqd, 51868989, 76>, <Fbt Jwfcpehi, 39326078, 43>]
```

A Very Small Example

We'll construct a hash table with $m = 23$ entries, and insert the records of 14 students in it.

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

students_list=students(14)

for i in range(14):
    students_list[i].name=names[i]

>>> students_list
[<Reuben,20820721,70>, <Simeon,23803686,68>,
<Levi,20698074,67>, <Judah,56029794,52>,
<Dan,52650781,48>, <Naphtali,46802782,86>,
 <Gad,27792439,60>, <Asher,56119927,90>,
 <Issachar,49904136,73>, <Zebulun,29998597,58>,
 <Benjamin,49995918,76>, <Joseph,59159846,46>,
 <Ephraim,58373195,55>, <Manasse,36664945,24>]
```

Resolving Collisions Using Chaining

In our example of hashing students' names, we 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 14 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 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.

In particular, if the list was empty initially, it will contain one item after insertion. If an equal item is not found, but another item with the same `name` but with a different `id` was found, we will still insert "our student" at the end of the list.

A Note Regarding the List Methods append and remove

We will employ both append and remove, two list methods that modify (mutate) a list. The method remove removes the first occurrence of an item. If the item is not in the list, an error flag is raised.

```
>>> entry=[]
>>> id(entry)
4299878200
>>> entry.append(students_list[0])
>>> id(entry)
4299878200
>>> entry.append(students_list[5])
>>> entry
[<Reuben,20820721,70>, <Naphtali,46802782,86>]
>>> entry.remove(students_list[5])
>>> entry
[<Reuben,20820721,70>]
>>> entry.remove(students_list[5])
Traceback (most recent call last):
  File "<pyshell#76>", line 1, in <module>
    entry.remove(students_list[5])
ValueError: list.remove(x): x not in list
```

Dictionary Operations: Python Code

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

def contained(candidate,list_of_items):
    """ checks if item is a member in list_of_items """
    for item in list_of_items:
        if item.key == candidate:
            return item.value
    return None

def insert(s,table=small_hash_table,h=hash,mod=23):
   """ insert an item, s, into hash_table with m elements """
   i=hash_mod(s.key,func=h,m=mod)
   if not contained(s.key,table[i]):
      table[i].append(s)
   return None
```