Extended Introduction to Computer Science CS1001.py

Lecture 15: The Dictionary Problem Hash Functions and Hash Tables

Instructors: Amir Rubinstein, Amiram Yehudai

Teaching Assistants: Yael Baran, Michal Kleinbort,

School of Computer Science
Tel-Aviv University
Winter Semester, 2015-16
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 5 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 5, 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.

There are two variations of this data structure, according to the type of the elements stored in the data structure.

- pairs key: value, or
- Just keys

In any case, we assume all keys are distinct

Most of the discussion will be relevant to both variations, but examples may be from one or the other variations.

Dynamic Data Structure: Dictionary

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 (with elements that are pairs key:value)

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 may be 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.

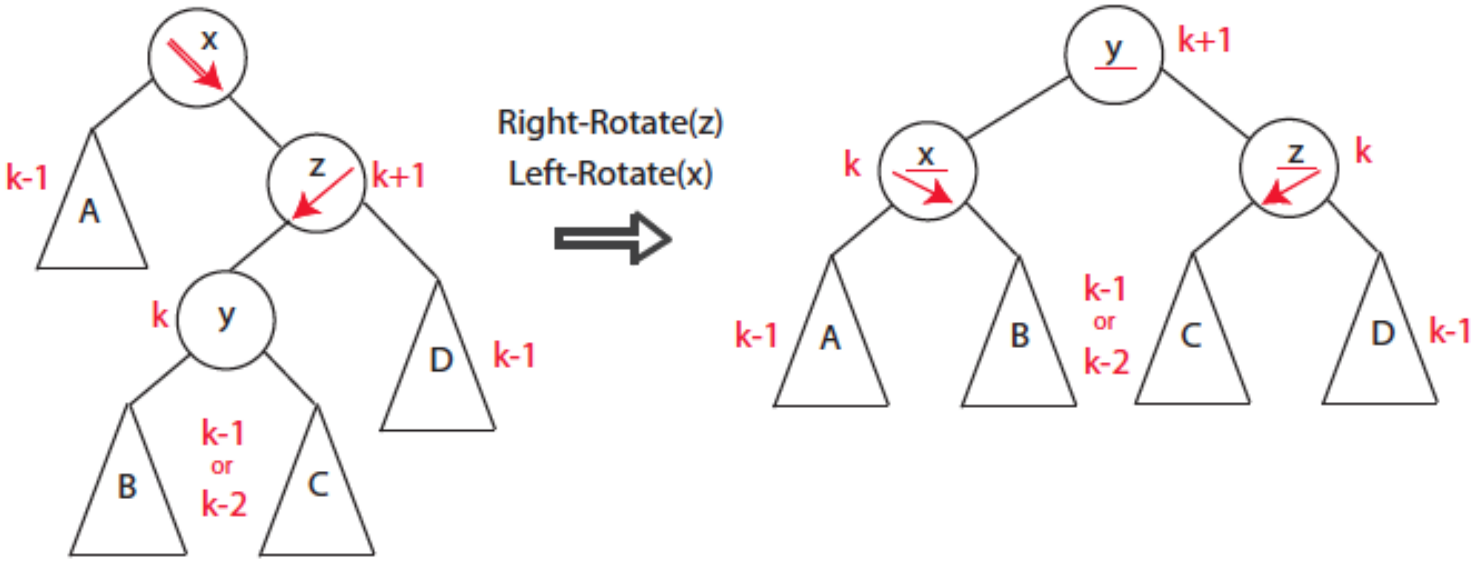
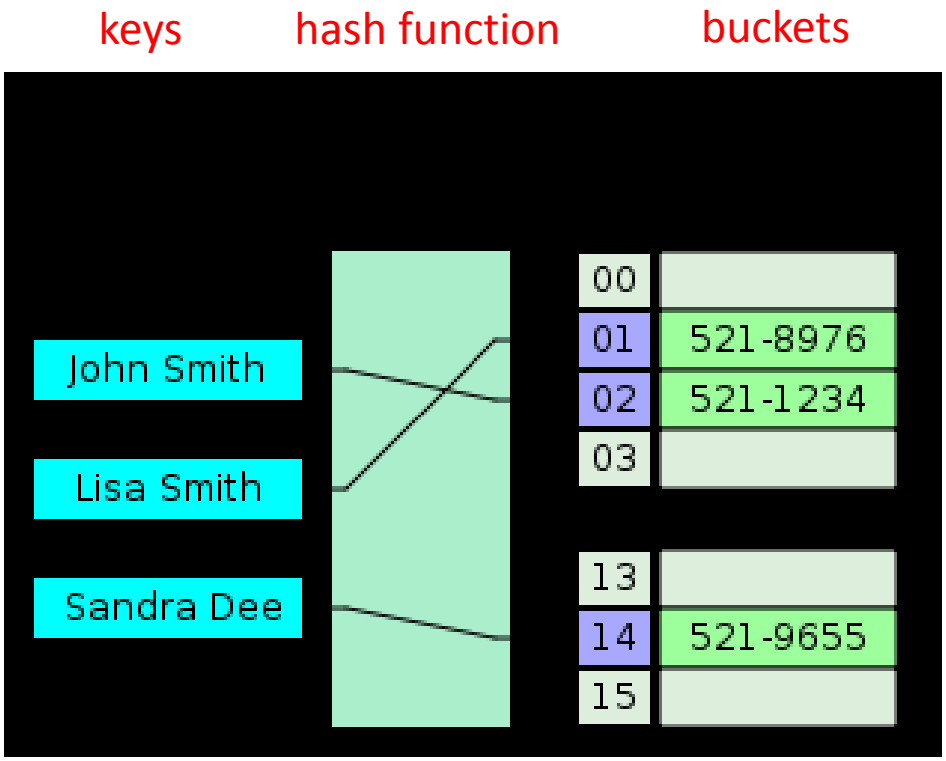


Figure from MIT algorithms course, 2008. Shows item insertion in an AVL tree.

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.

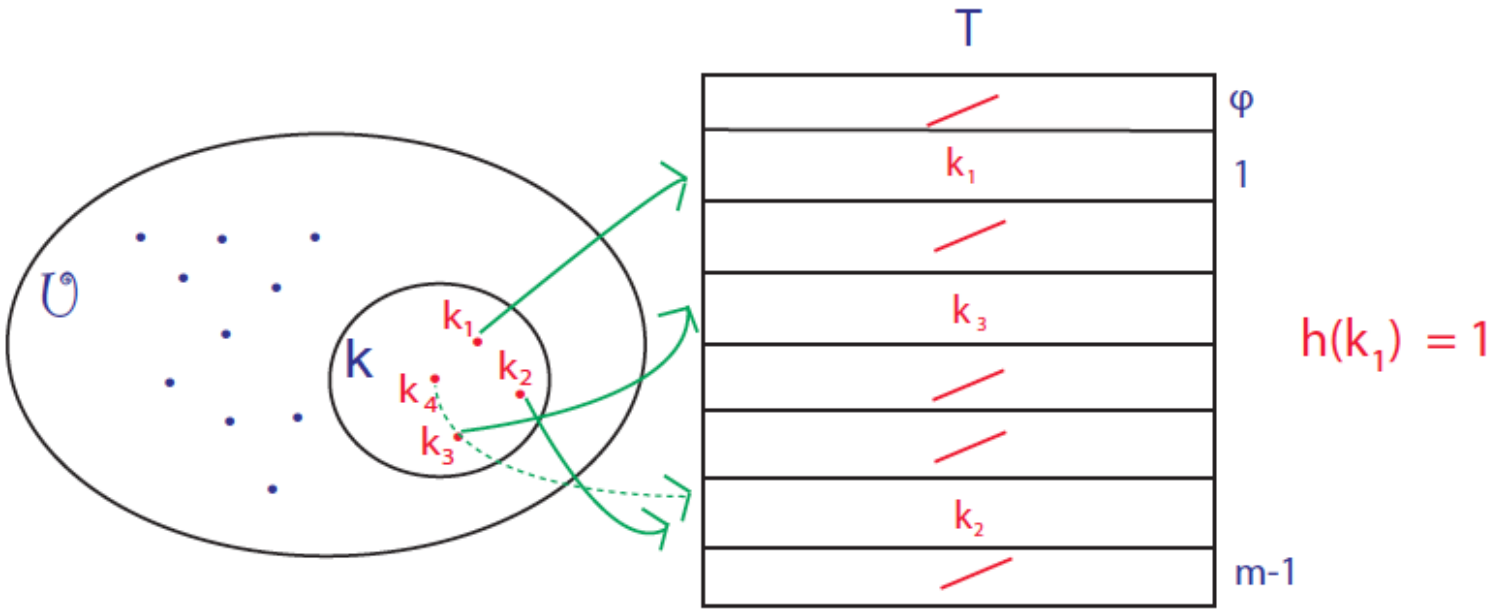
Dictionary Setting

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 $\mathcal{K}$).



13.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 $\mathcal{T}$.
- (When elements are pairs, we can retrieve the value of the item)

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 $|\mathcal{U}| >> n$ 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.

hash (key) % p

Other hash functions?

Are these hash functions good?

- h(n) = random.randint(0,n) (for ints)
- h(x) = 7 (for ints, strs,…)
- h(n) = n%100 (for ints)

- A good hash function is one that:
  - Distributes element in the table uniformly (and deterministically!)
  - Is easy to compute (O(m) for an element of size m)

Other hash functions (cont.)

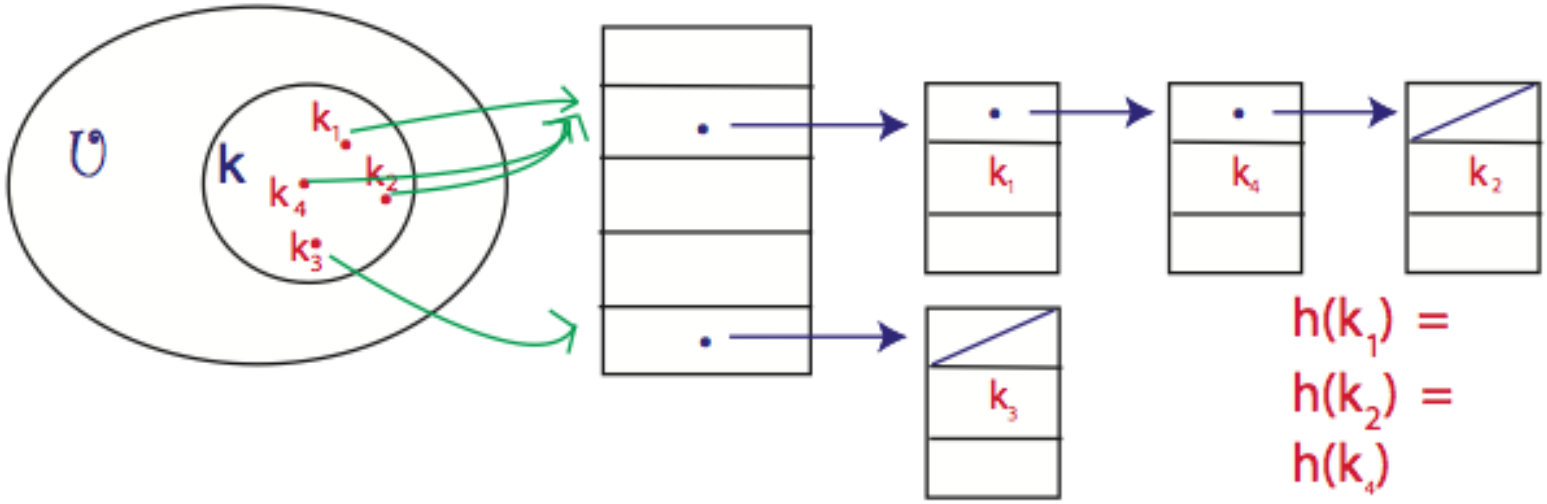
- An example for a hash function for strings:

```
def hash4strings(s):
    """ ord(c) is the ascii value of character c
        2**120+451 is a prime number """
    sum = 0
    for i in range(len(s)):
        sum = (128*sum + ord(s[i])) % (2**120+451)
    return sum**2 % (2**120+451)
```

- When we have some apriori knowledge on the keys, their distribution, etc., we may prefer another hash function, tailored for our context.
- Normally, Python's hash should do the job.

Approaches for Dealing with Collisions: Two Approaches

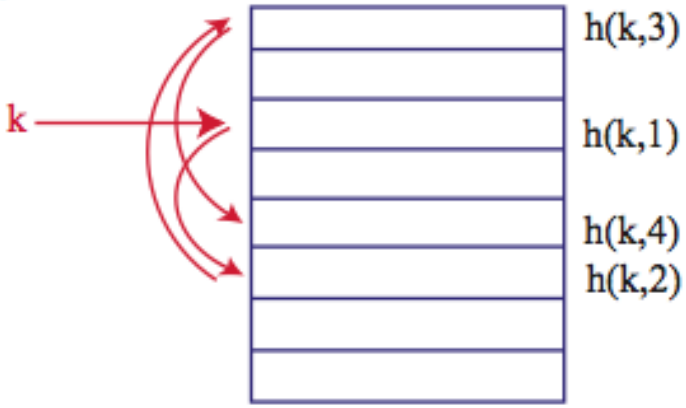
1) Chaining:



In chaining, each entry in the table contains a link to a list that will include all the items that have been mapped to this index.

Two Approaches for Dealing with Collisions: (2) Open Addressing

In open addressing, each slot in the hash table contains at most one item. This obviously implies that n cannot be larger than m . 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.

Collisions: an experiment

Before we discuss in detail the implementation of the hash table (including dealing with collisions), we will run small examples to see how many items will collide (be mapped by the hash function to the same entry).

So for now, we do not actually create a hash table (we do not actually resolve the collisions), just note the collisions.

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

We assume a hash table with m = 23 entries, and a set of n = 14 students' names that are to be inserted into the hash table. In order to insert a record into the hash table, the hash value of the key is computed, and determines the index of the table entry. Here we only check how the names would be mapped, and in particular what is the number of collisions.

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 hash (key) % p to its name, and obtain an index ℓ, $(0 \leq \ell \leq m-1)$.

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 ']

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

We create a python dictionary whose elements are pairs: an index ℓ, $(0 \leq \ell \leq m-1)$, and all the names mapped to it. We then print the results

```
>>> mappedTo = {i : [ ] for i in range(m)}
>>> for name in names:
        mappedTo[hash(name) % m] += [name]
>>> for i in mappedTo:
        print ('Mapped to ' + str(i) + ' Names: ' + str(mappedTo[i]))
```

The printout appears in the next slide

Mapped to 0 Names: []
Mapped to 1 Names: ['Issachar ', 'Joseph ']
Mapped to 2 Names: []
Mapped to 3 Names: []
Mapped to 4 Names: ['Zebulun ']
Mapped to 5 Names: []
Mapped to 6 Names: []
Mapped to 7 Names: []
Mapped to 8 Names: ['Reuben ', 'Simeon ', 'Naphtali ', 'Benjamin ']
Mapped to 9 Names: []
Mapped to 10 Names: ['Gad ']
Mapped to 11 Names: []
Mapped to 12 Names: ['Ephraim ']
Mapped to 13 Names: []
Mapped to 14 Names: []
Mapped to 15 Names: []
Mapped to 16 Names: []
Mapped to 17 Names: []
Mapped to 18 Names: ['Levi ', 'Dan ']
Mapped to 19 Names: ['Judah ']
Mapped to 20 Names: ['Manasse ']
Mapped to 21 Names: ['Asher ']
Mapped to 22 Names: []

Hash collisions for n = 14, m = 23

So we have 3 collisions, two of size 2 ($\ell = 1$ and $\ell = 18$) and one of size 4 ($\ell = 8$)

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

So with n=14, m=23, we have 3 collisions, two of size 2 and one of size 4.

Let us take a somewhat bigger table (m = 31 slots) for the same keys (n = 14). Are there fewer collisions? The results in the next slide show just 1 collision, of size 2.

We should remember that these are just examples, and this does not prove that a larger table (with the same number of keys) will always yield fewer collisions.

The question that remains is, how shall we deal with such collisions?

Mapped to 0 Names: ['Benjamin ']
Mapped to 1 Names: []
Mapped to 2 Names: []
Mapped to 3 Names: ['Simeon ']
Mapped to 4 Names: []
Mapped to 5 Names: ['Manasse ']
Mapped to 6 Names: []
Mapped to 7 Names: []
Mapped to 8 Names: ['Asher ']
Mapped to 9 Names: ['Zebulun ']
Mapped to 10 Names: []
Mapped to 11 Names: []
Mapped to 12 Names: ['Gad ']
Mapped to 13 Names: []
Mapped to 14 Names: ['Issachar ']
Mapped to 15 Names: ['Levi ']
Mapped to 16 Names: []
Mapped to 17 Names: ['Judah ']

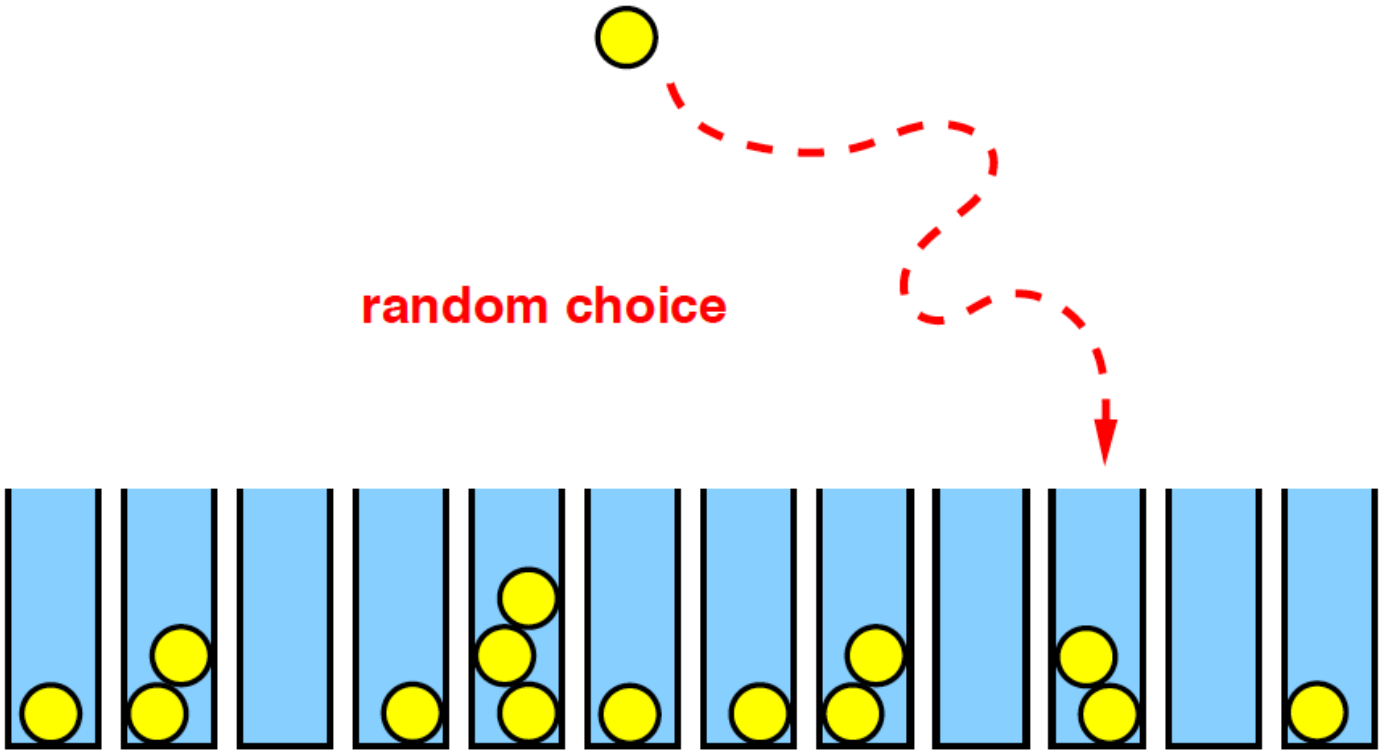
Hash collisions for n = 14, m = 31

Mapped to 16 Names: []
Mapped to 17 Names: ['Judah ']
Mapped to 18 Names: []
Mapped to 19 Names: ['Reuben ']
Mapped to 20 Names: []
Mapped to 21 Names: ['Dan ']
Mapped to 22 Names: ['Naphtali ', 'Ephraim ']
Mapped to 23 Names: []
Mapped to 24 Names: []
Mapped to 25 Names: []
Mapped to 26 Names: ['Joseph ']
Mapped to 27 Names: []
Mapped to 28 Names: []
Mapped to 29 Names: []
Mapped to 30 Names: []

Here we have just 1 collision, of size 2 ($\ell = 22$)

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 – for reference only

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

Next time we will discuss in detail how to deal with collisions, and we will see a detailed implementation.