Extended Introduction to Computer Science CS1001.py

Lecture 18: Hash Functions and Hash Tables (cont.)
(The Dictionary Problem)

Instructors: Benny Chor, Amir Rubinstein

Teaching Assistants: Michal Kleinbort, Amir Gilad

School of Computer Science
Tel-Aviv University
Spring Semester 2017
http://tau-cs1001-py.wikidot.com

Lecture 18

- OOP (lec 15)

- Data Structures
  1. Linked Lists (lec 16)
  2. Binary Search Trees (lec 17)
  3. Hash tables (lec 17-18)
  4. Iterators and generators (lec 19)

Lecture 18: Plan

We'll introduce an additional, very efficient data structure: hash table.

- hash functions (reminder)
- hash tables: storing and searching elements efficiently
- Resolving collisions:
  - Chaining
  - Open addressing (with the example of cuckoo hashing)

A hash function example - Reminder

- Hash function: a function that maps a large (possible infinite) set to a smaller set of a fixed size.
- example for a hash function from strings to integers:

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

- Executions in class

- Note that this function spreads the (infinite) set of strings over a finite range (albeit large).

- But what can such a function be possibly good for? soon…

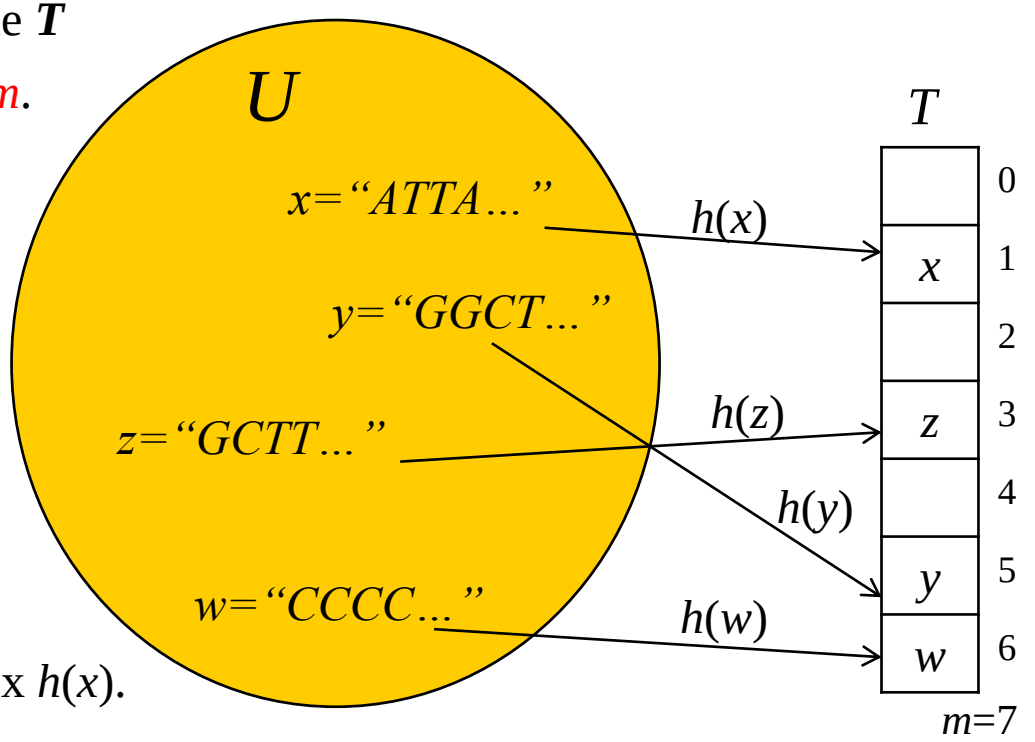
Hash Tables - Reminder

- Let us now define this mechanism more formally.
- Suppose elements belong to a large set (possibly infinite), called the "**universe**", denoted ***U*** (for example, all possible strings, all possible ID numbers etc.)

- Suppose we need to store *n* elements from ***U***, and ***n*** << |***U***|.
  (for example, genes of an organism, ID's of students in class right now)

- We can keep the elements in a table ***T*** called **hash table**, whose size is *m*.
  $|T| = m << |U|$

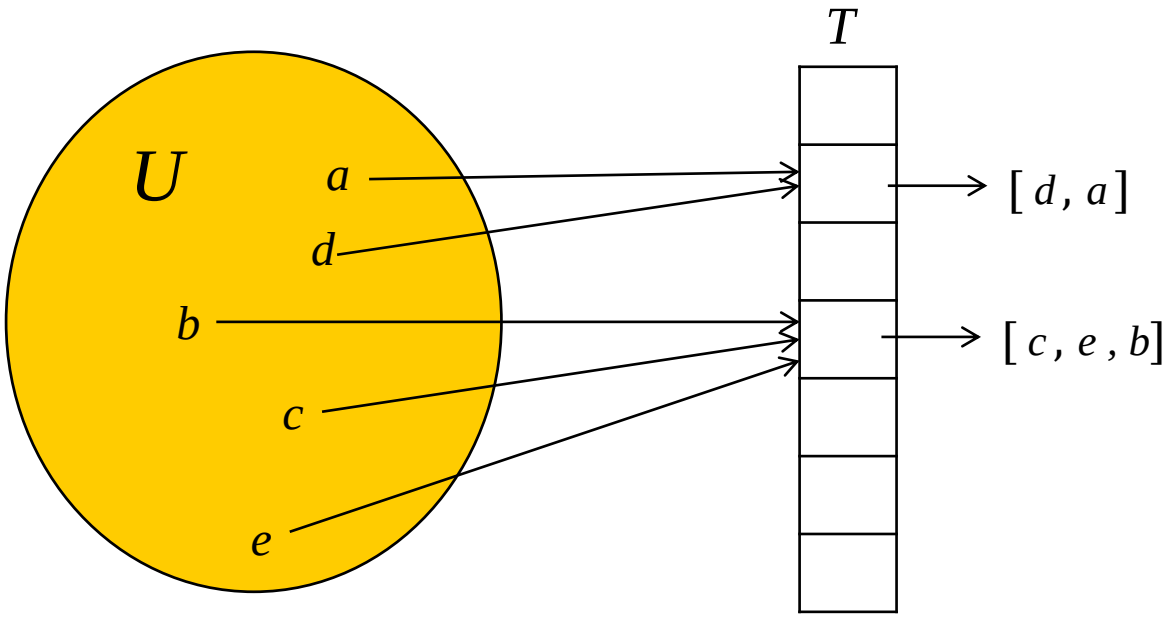
- To map elements from *U* to *T*
  we will use a **hash function**
  $h: U \rightarrow \{0, 1, \ldots, m\text{-}1\}$



Element $x \in U$ will be stored at index $h(x)$.

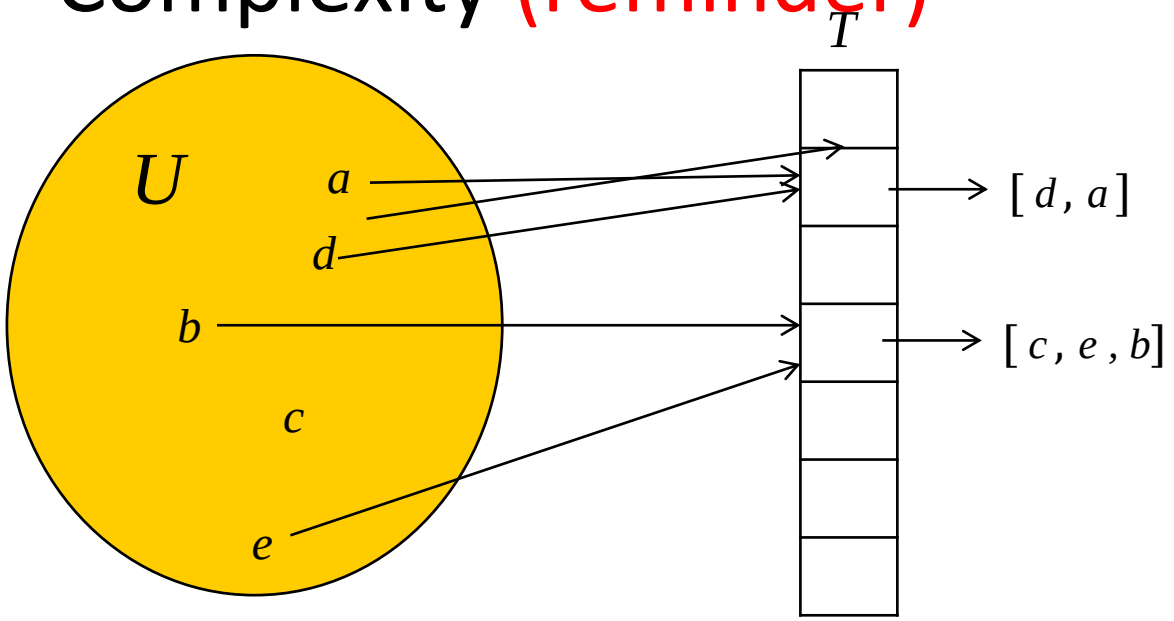
Approaches for Dealing with Collisions: The First Approach – Chaining (reminder)

Chaining:



- Each cell in the table will contain a list (can be linked or not)
- How do we search an element in the table then?
- insert?
- delete?

Chaining – Complexity (reminder)



- The average length of a chain is n/m. This is called the "load factor", denoted α.
- We don't want α to be too large or too small (why?)
- if we choose m such that n = O(m), then α=O(1).
- This requires some estimation of the number of element we expect to be in the table, or a mechanism to dynamically update the table size
- what is the average time complexity of search, insert, delete?
- worst case?

Good hash functions?

- How rare is the worst case? can we trust to fall within a reasonable deviation from the average case most of the time?

- The answer to this depends mainly on how uniformly the hash function spreads the elements in the table.

- A good hash function is one that:
  - Distributes element in the table uniformly (and of course deterministically!)
  - Is easy to compute (O(|key|) where |key| is the size of key)

- Is $h(n) = n\%100$ a good hash function for integers?

- When we have some apriori knowledge on the keys, their distribution and properties, etc., we can tailor a specific hash function, that will improve spread-out among table cells.
- Otherwise, we can expect Python's hash will do a good job (trust but check).

Pyhon's classes dict and set

- Python's class **dict** and class **set** (same as dict but only keys, no values) are both implemented behind the scenes as hash tables.

- This explains why they are such good choices for storing and searching elements. Indeed, we used them for memoization.

- dict and set however do not use chaining to solve collisions. They use another approach called open addressing.
  - We will see later an example for open addressing called cuckoo hashing (not used by dict and set, however, which use a more sophisticated approach).

- In addition, dict and set are dynamic hash tables – they grow and shrink as the load factor becomes too large or too small.

- The exact details may change between versions (e.g. 3.5 and 3.6), due to optimization efforts by the language developers. We will not delve into those details, however.

Implementation in Python

- Let us implement our own class `Hashtable` in Python now.
- We will use chaining for resolving collisions.
- We will demonstrate its usage with elements which are simple strings first. Later on we will show another example with class `Student`.

Initializing the Hash Table

```
class Hashtable:

 def __init__(self, m, hash_func=hash):
     """ initial hash table, m empty entries """
     self.table = [ [] for i in range(m)]
     self.hash_mod = lambda key: hash_func(key) % m



 def __repr__(self):
     L = [self.table[i] for i in range(len(self.table))]

     return "".join([str(i) + " " + str(L[i]) + "\n"
                     for i in range(len(self.table))])
```

Initializing the Hash Table

```
>>> ht = Hashtable (11)
>>> ht
0 []
1 []
2 []
3 []
4 []
5 []
6 []
7 []
8 []
9 []
10 []
```

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: a Bogus Code

Consider the following alternative initialization:

```
class Hashtable:
 def __init__(self, m, hash_func=hash):
     """ initial hash table, m empty entries """
     self.table = [[]]*m
```

```
>>> ht = Hashtable(11)
>>> ht.table[0] == ht.table[1]
True
>>> ht.table[0] is ht.table[1]
True
```

The entries produced by this bogus __init__ are identical.
Therefore, mutating one mutate all of them:

```
>>> ht.table[0].append(5)
>>> ht
0 [5]
1 [5]
…
```

Initializing the Hash Table, cont.

But this one will work fine, as different copies of the empty list will be created:

```
class Hashtable:

  def __init__(self, m, hash_func=hash):
     """ initial hash table, m empty entries """
     empty = []
     self.table = [ list(empty) for i in range(m)]
```

Dictionary Operations: Python Code

```
class Hashtable:
…

 def find(self, item):
        """ returns True if item in hashtable, False otherwise  """
        i = self.hash_mod(item)
        chain = self.table[i]
        if item in chain: # a hidden loop
              return True
        else:
            return False

 def insert(self, item):
        """ insert an item into table, if not there """
        i = self.hash_mod(item)
        chain = self.table[i]
        if item not in chain: # a hidden loop
            chain.append(item)
```

```
return item in chain
```

Example: A Very Small Table (n = 14, m = 7)

In the following slides, there are executions construct a hash table with m = 7 entries. We'll insert n = 14 string record in it and check how insertions are distributed, and in particular what the maximum number of collisions per cell is.

Our hash table will be a list with m = 7 entries. Each entry will contain a list with a variable length. Initially, each entry of the hash table is an empty list.

Example: A Very Small Table (n = 14, m = 7)

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


>>> ht = Hashtable(7)


>>> for name in tribes:
        ht.insert(name)

>>> ht #calls __repr__
```

(next slide)

Example: A Very Small Table (n = 14, m = 7)

```
>>> ht #calls __repr__
0 []
1 ['Reuben', 'Judah', 'Dan']
2 ['Naphtali']
3 ['Gad', 'Ephraim']
4 ['Levi']
5 ['Issachar', 'Zebulun']
6 ['Simeon', 'Asher', 'Benjamin', 'Joseph', 'Manasse']
```

Example: A slightly larger table (n = 14, m = 21)

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


>>> ht = Hashtable(21)


>>> for name in tribes:
        ht.insert(name)

>>> ht #calls __repr__
```

(next slide)

Example: A slightly larger table (n = 14, m = 21)

```
>>> ht #calls __repr__
0 []
1 []
2 []
3 ['Ephraim']
4 []
5 ['Issachar']
6 ['Benjamin']
7 []
8 ['Judah']
9 ['Naphtali']
10 []
11 []
12 ['Zebulun']
13 ['Manasse']
14 []
15 ['Reuben', 'Dan']
16 []
17 ['Gad']
18 ['Levi']
19 []
20 ['Simeon', 'Asher', 'Joseph']
```

Hashing random seed

- If you run this code yourself, you will probably encounter results different from those in the last few slides.
- This is because when IDLE starts, it randomly generates a number called seed, which is used to compute it built-in hash function
- This is intended to provide protection against a denial-of-service attacks caused by carefully-chosen inputs designed to collide. Such attacks exploit the worst case performance when using hash.
- Of course, as long as you work under the same instance of IDLE, hash is consistent. But re-opening IDLE will break this consistency.

2 issues with Collisions of Hashed Values

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

- Let $\mathcal{K}$ be the set of stored elements, $|\mathcal{K}| = n$ and $|T| = m$, and assume that the values h(k) for $k \in \mathcal{K}$ are distributed in T at random.

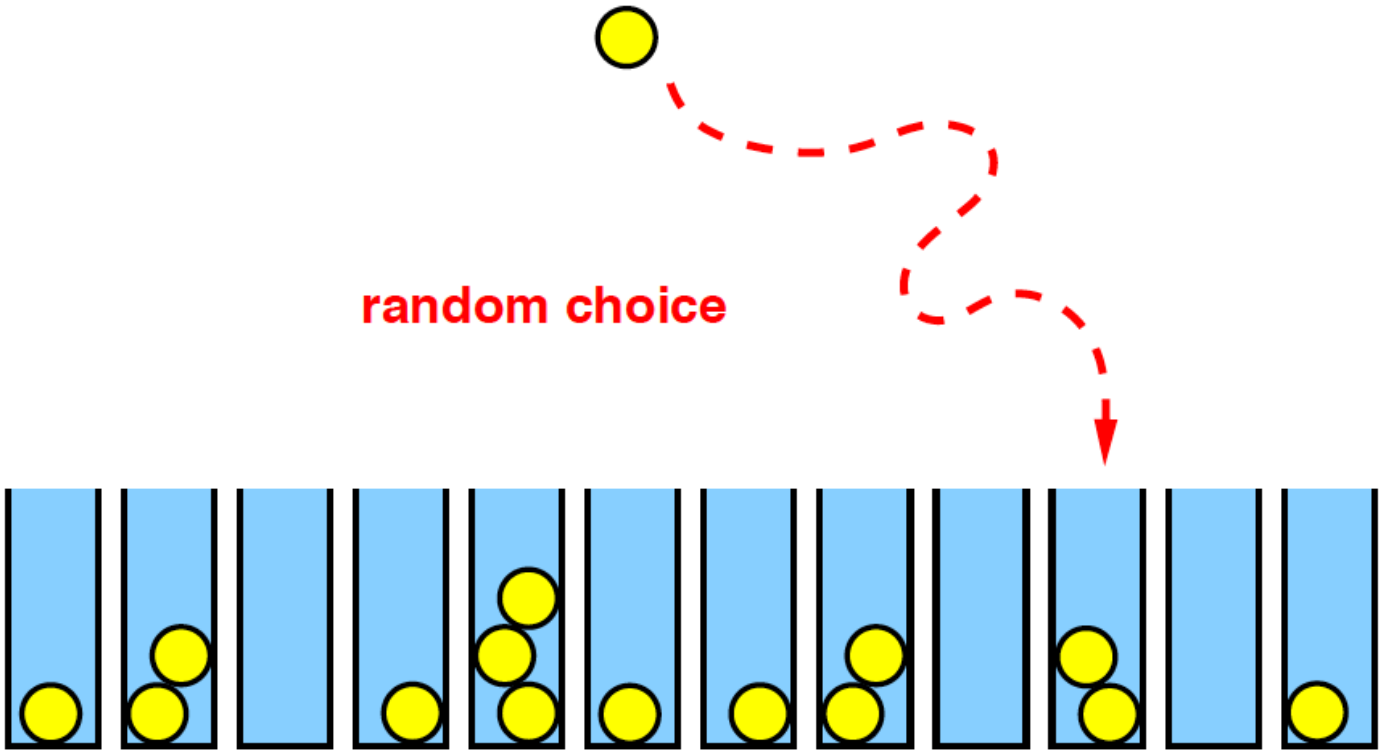
  - Q1: What is the probability that a collision exists?
  - Q2: What is the size of the largest colliding set (a maximal set $S \subset \mathcal{K}$ whose elements are all mapped to the same target by h)?

- The answer to these questions depend on the ratio $\alpha = n/m$ (the load factor = average number of keys per chain in the table)

  - If $\alpha > 1$, then clearly there is at least one collision (pigeon hole principle).
  - If $\alpha \leq 1$, and if we could tailor h to $\mathcal{K}$, then we could avoid collisions. However, such tinkering is not possible in our context, since we have no control over which elements will be stored.

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

- 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.

Collision Size – for reference only

- Let $|\mathcal{K}|$ = n and |T| = m.
- The expected maximal capacity in a single slot is known for the following cases (proof omitted):

load factor	condition	expected maximal capacity is a single slot
sublinear	$n < \sqrt{m}$	1 (=no collisions)
	$n = m^{1-\epsilon} \quad , 0 < \epsilon < 1/2$	$O(1/\varepsilon)$
linear	$n = m$	$\frac{ln(n)}{lnln(n)}$
superlinear	$n > m$	$\frac{n}{m} + \frac{ln(n)}{lnln(n)}$

- This provides some intuition as to the rareness of the worst case.

Hashing and User-defined Classes

- So far we have stored in our class Hashtable Python's built-in classes such as str and int.
- We will now use class Hashtable on our own class Student.
- As we will see, this will raise certain issues.

The Student Class

```
class Student:
    def __init__(self, name, surname, ID):
        self.name = name
        self.surname = surname
        self.id = ID
        self.grades = dict()

    def __repr__(self): #must return a string
        return "<" + self.name + ", " + str(self.id) + ">"


    def update_grade(self, course, grade):
        self.grades[course] = grade


    def avg(self):
        s = sum([self.grades[course] for course in self.grades])
        return s / len(self.grades)
```

Hashing Students

```
>>> st1 = Student("Grace", "Hopper", 123456789)
>>> st2 = Student(st1.name, st1.surname, st1.id)
>>> st1
<Grace, 123456789>
>>> st2
<Grace, 123456789>
>>> hash(st1)
-9223372036851698786
>>> hash(st2)
3077117
```

From Wikipedia:
Grace Brewster Murray Hopper (1906 –1992), was an American computer scientist and United States Navy Rear Admiral.
She was one of the first programmers of the Harvard Mark I computer in 1944, invented the first compiler for a computer programming language, and was one of those who popularized the idea of machine-independent programming languages which led to the development of COBOL, one of the first high-level programming languages.

- This should not be a surprise (by default, Python uses the memory address of an object to compute the value of hash on it).

The __hash__ method

- We will add to class Student the special method __hash__.
- It defines the result of calling Python's hash on an object of this class.

```
class Student :
    …
    def __hash__(self): #so we can use hash(st) on a student
        return hash(self.id)   #assume id is a unique identifier
```

- Notes:
  1) __hash__ of Student class calls __hash__ of int class
  2) We used merely the id to compute a student's hash, under the assumption that it is unique. We could have used more fields of a Student object.

Hashing Students – almost done

```
>>> st1 = Student("Grace", "Hopper", 123456789)
>>> st2 = Student(st1.name, st1.surname, st1.id)

>>> hash(st1) == hash(st2) == hash(st1.id)
True ☺
```

Hashing Students – almost done

- Can you explain why the following search fails?

```
>>> ht = Hashtable(7)
>>> ht.insert(st1)
>>> ht
0 []
1 [<Grace, 123456789>]
2 []
3 []
4 []
5 []
6 []

>>> ht.find(st2) # recall st2 holds same data as st1
False ☹
```

hashing requires __eq__ as well

- Indeed, no much point in having __hash__ without __eq__, for comparing elements (within a chain inside a table's index).

```
class Student :
    …
    def __eq__(self, other): #so we can search for students in the table
        return self.name == other.name and \
                self.surname == other.surname  and \
                self.id == other.id
```

```
>>> ht.find(st2) # recall st2 holds same data as st1
True ☺
```

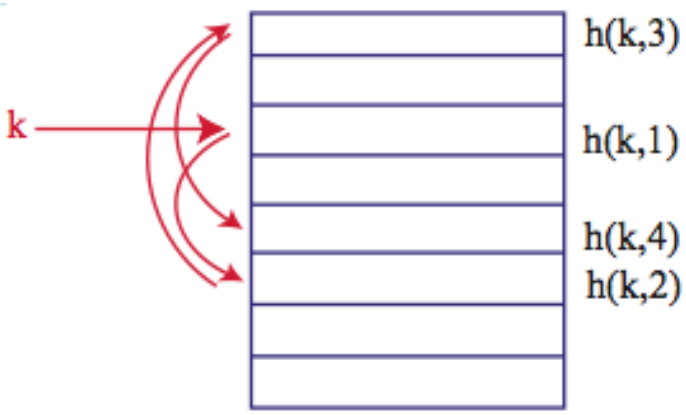
Two Approaches for Dealing with Collisions

1) Chaining – explained and implemented this 

2) Open addressing – we will briefly discuss it now

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 (e.g. fast switches and high capacity routers). It is also used in python dictionaries and sets.
- There are many approaches to open addressing. We will describe a fairly recent one, termed cuckoo hashing (Pagh and Rodler, 2001).

Cuckoo Hashing: Motivation

- We saw that if $n \le m$, hashing with chaining guarantees that insertion, deletion, and find are carried out in expected time O(1) per operation, and with high probability (probability is over choices of inputs) O(log n/log log n) per operation. (The worst case time is O(n) per operation.)

- In certain scenarios (e.g. fast routers in large internet nodes) we want find to run with high probability in O(1) time. (The worst case time is still O(n) per operation.)

- Compare O(1) time with high probability to O(1) expected time of hashing with chaining.

- Cuckoo hashing is one way to achieves this, but there are two prices to pay:
  1) Instead of $n \le m$, we require $n \le 7m/8$, or $n \le 3m/4$, or $n \le m/2$, or even $n \le m/3$.
     - That is, we pay a price in terms of memory
  2) insert may take somewhat longer time.

Cuckoo Hashing

- Cuckoo hashing uses two distinct hash functions, h_1 and h_2 (improved versions use four, six, or eight, but the idea is the same).

- Each key, k , has two potential slots in the hash table, $h_1(k)$ and $h_2(k)$. If we search for k, all we have to do is look for it in these two locations (no chains here -- at most one item per slot).

- It is slightly more involved to insert a record whose key is k .

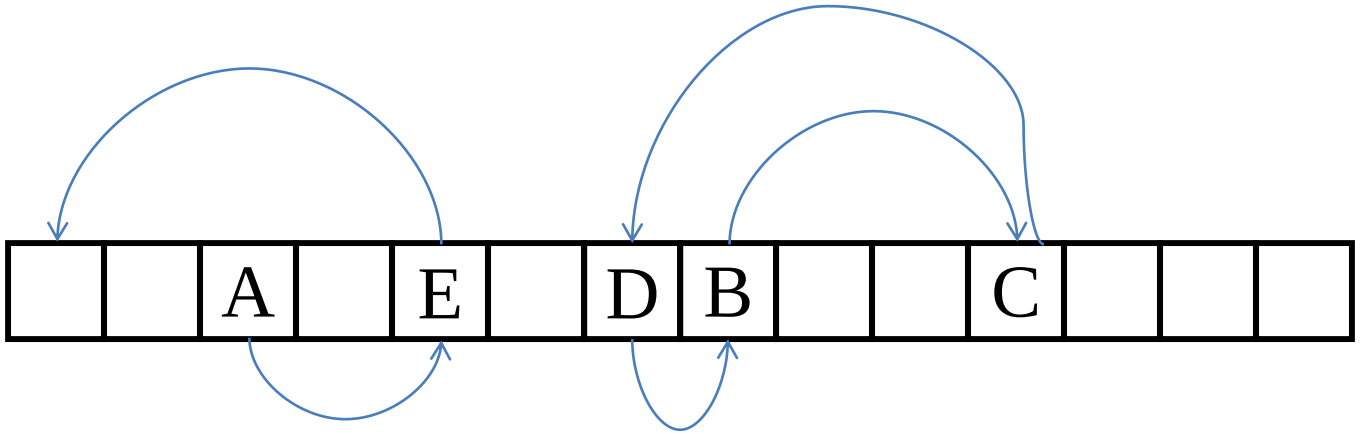
Cuckoo Hashing

It is slightly more involved to insert a record whose key is k .

- If any of the two slots, $h_1(k)$ or $h_2(k)$ is empty, k is inserted there.
- If both slots are full, pick one of the two occupants, say x. Place k in x's current slot.
- Assume this was location $h_1(x)$. Place x in its other slot, $h_2(x)$.
- If that slot was empty, we are done.

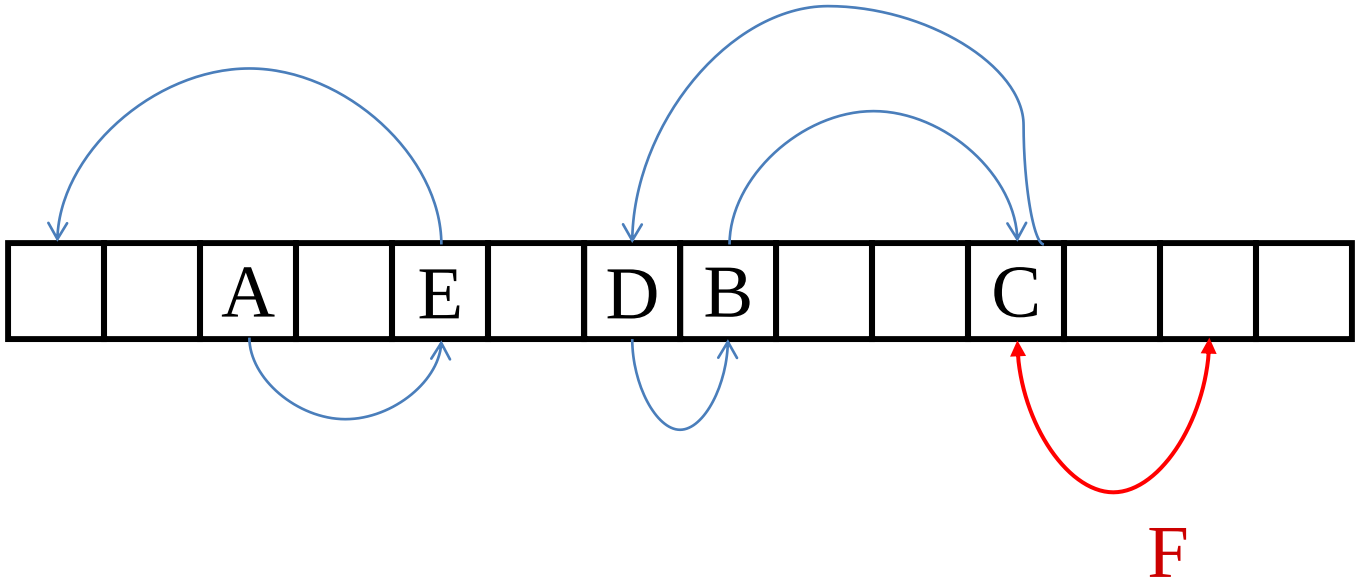
- Otherwise, the slot is occupied by some y . Place this y in its other slot, potentially kicking its present occupant, etc.,etc., until we find an empty slot.

Cuckoo Hashing: Examples



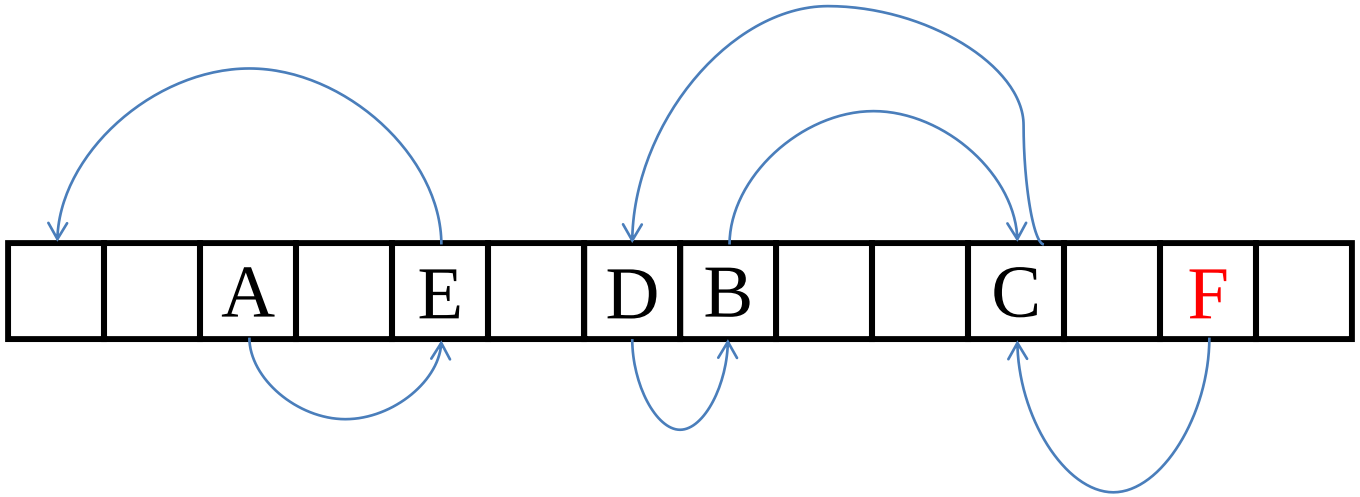
= The other potential slot for an item

Cuckoo Hashing: Examples



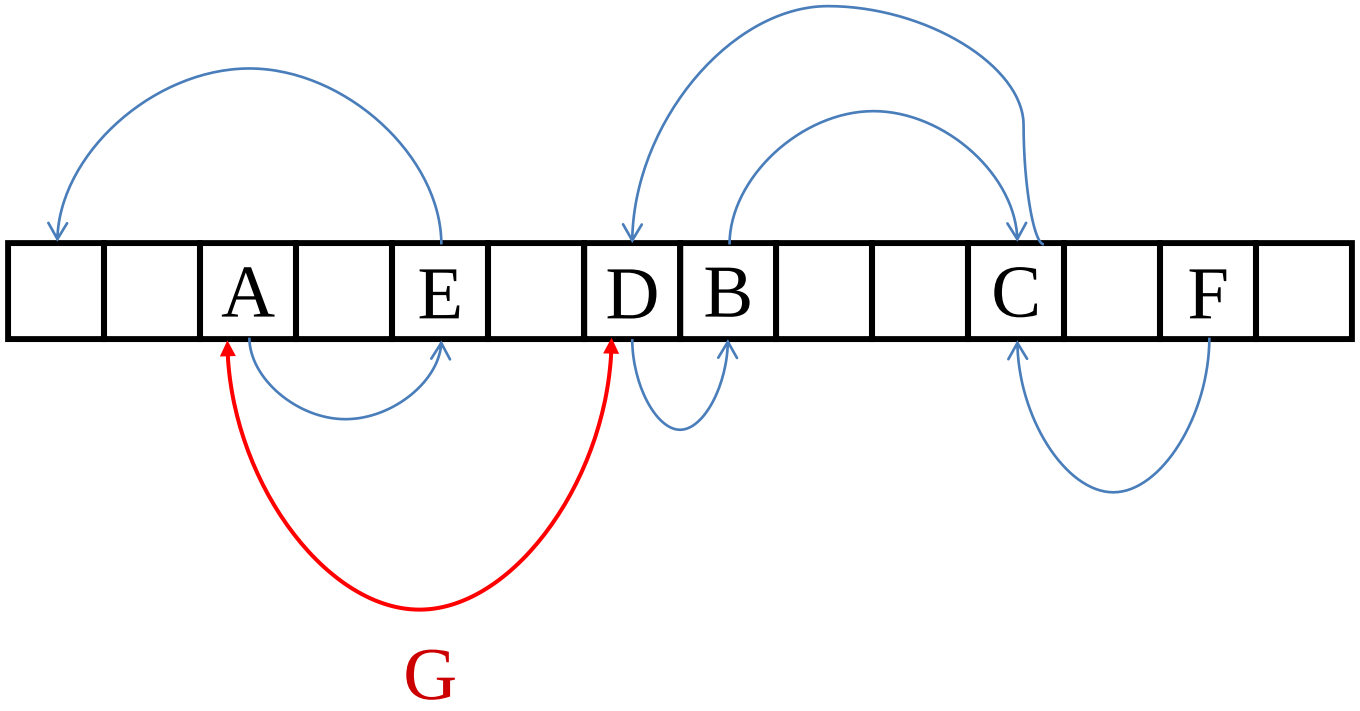
= The other potential slot for an item

Cuckoo Hashing: Examples



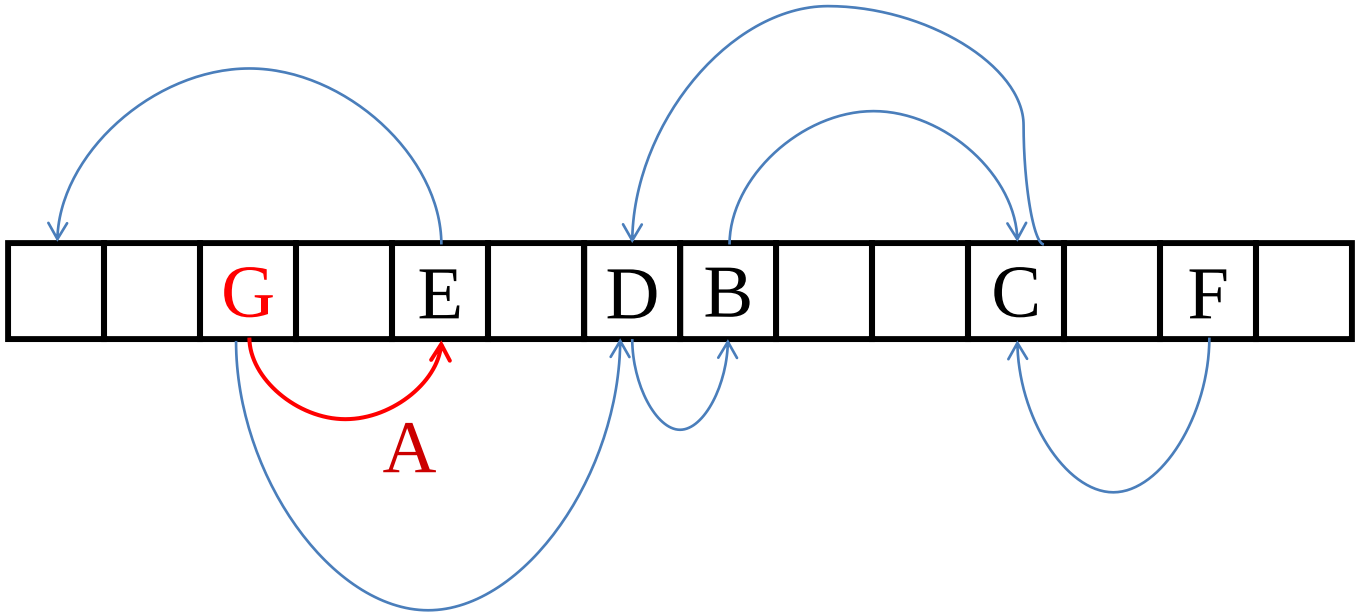
= The other potential slot for an item

Cuckoo Hashing: Examples



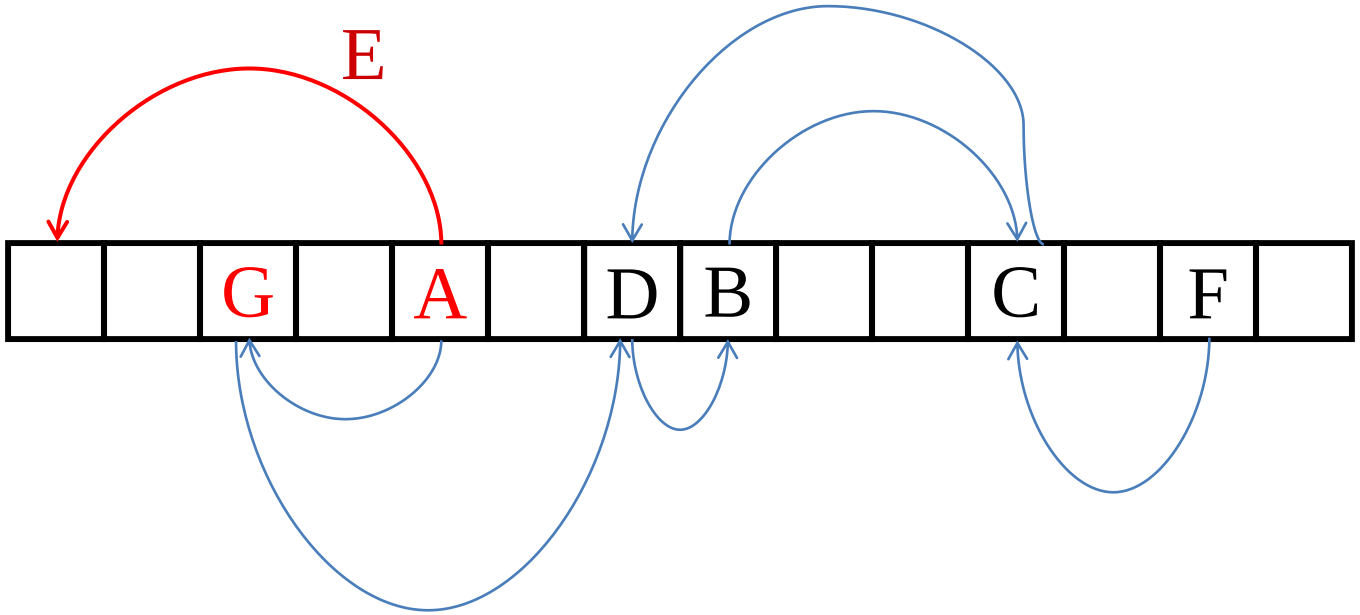
= The other potential slot for an item

Cuckoo Hashing: Examples



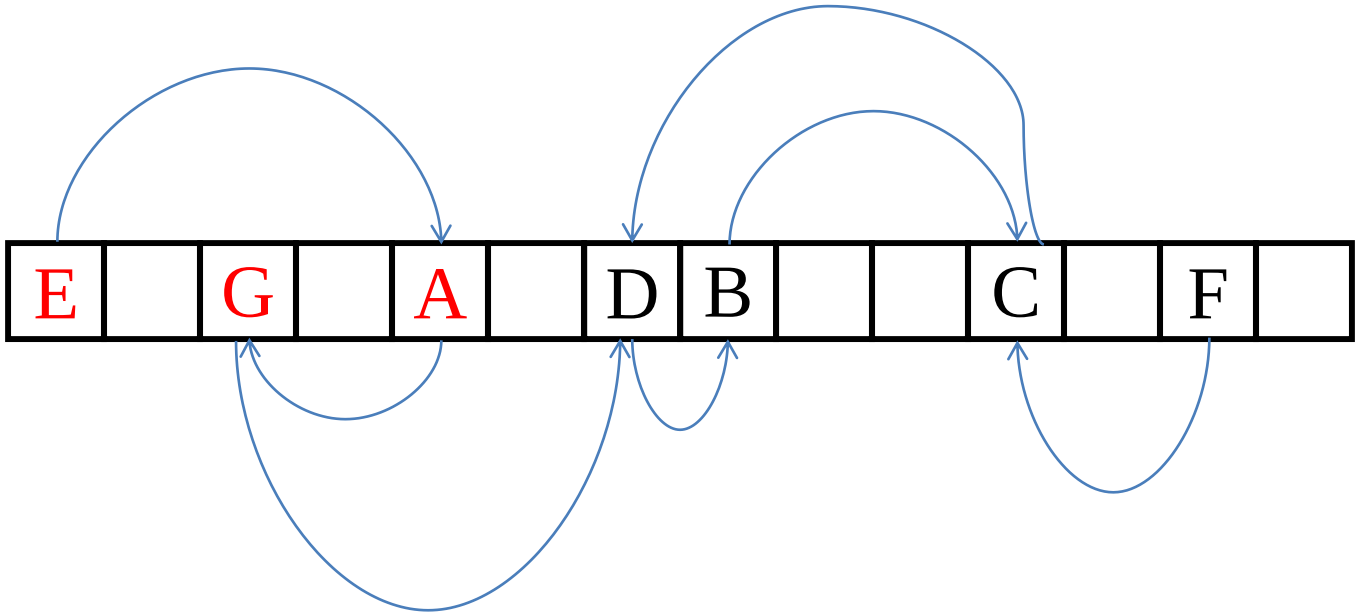
= The other potential slot for an item

Cuckoo Hashing: Examples



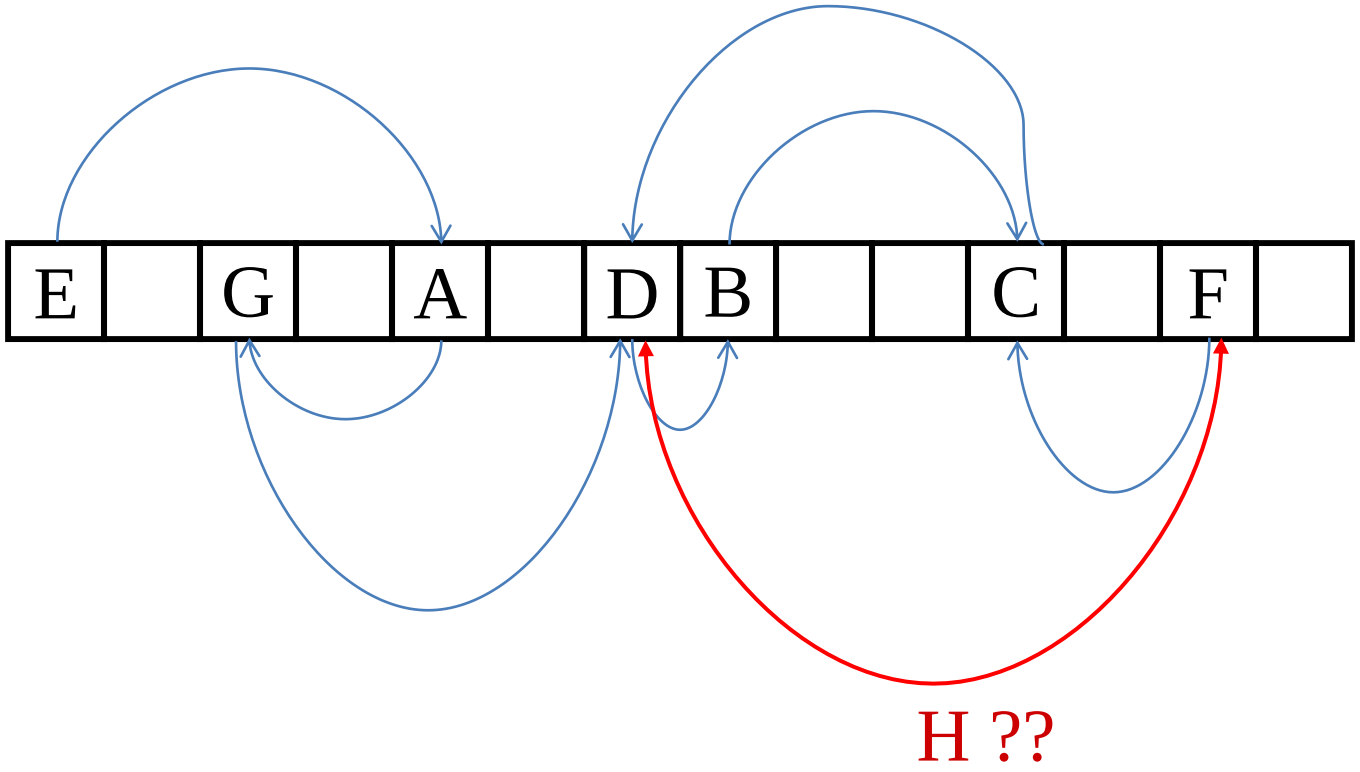
= The other potential slot for an item

Cuckoo Hashing: Examples



= The other potential slot for an item

Cuckoo Hashing: Examples



= The other potential slot for an item

Cuckoo Hashing - Deadlocks

- In the last example, we have reached a cycle, and we are in a non ending loop. This is called a deadlock.
- The union of the potential locations of 5 items (B, C, D, F, H) is just 4 slots.
- This obviously is very bad news for our cuckoo hashing.
- Notice that this is not a very likely event. With very high probability, the 10 potential locations ($10=5\cdot 2$) will attain more than just 4 distinct values (which is why we got stuck in that the last example).

Cuckoo Hashing – Solving Deadlocks

- Another possible problem is that there will be no cycle, but the path leading to the successful insertions will be very long.

- Fortunately, such unfortunate cases occur with very low probability when the load factor , i.e. n/m, is sufficiently low. The common recommendation for two hash functions , $h_1(\cdot)$, $h_2(\cdot)$, is to have n/m < 1/2 . (More hash functions enable a higher load factor).

- A theoretical solution: In case of failure (or very long path), rehash using “fresh hash functions."

- A more practical solution: Maintain a very small excess zone (e.g. 32 excess slots for a hash table with m=10000 slots) and place items “causing trouble'' there. If regular search (applying $h_1(x)$, $h_2(x)$) fails, search the excess zone as well.

Cuckoo Hashing in the Real World

- The load factor has to be smaller than 1. Yet a small load factor, say $n/m < 1/2$, is a waste of memory.

- In high performance routers, for example, most operations (including the hashing) are done in silico, by the hardware. The critical resource is memory area within the chip. Low load factor means wasted area.

- Instead of just 2 hash functions, 4 to 8 hash functions are utilized. This allows to increase the load factor to $n/m = 3/4$ or even $n/m = 7/8$.

- Suppose we use 4 hash functions, $h_1()$, $h_2()$, $h_3()$, $h_4()$. Given an element, x , that we wish to insert, we first check if any of the four locations $h_1(x)$, $h_2(x)$, $h_3(x)$, $h_4(x)$ is free.

Cuckoo Hashing in the Real World, cont.

- If these 4 locations are all taken, let *a,b,c,d* be the four elements in the above mentioned locations, respectively.

- Look, for example, at *a*. If one of the other 3 locations among $h_1(a)$, $h_2(a)$, $h_3(a)$, $h_4(a)$ is free, we move *a* there, and put x in its place. If not, we do the same with respect to *b*, then *c*, then *d*.

- If all these are taken ($4+4\cdot3=16$ different locations, typically), we go one more level down this search tree ($12\cdot3 = 36$ additional locations, typically).

- If all these are taken, we give up on x and put it in the garbage bin ("excess zone" table).

- With very high probability, the small excess zone does not fill up. After removing elements from the table, we could try re-inserting such x to the hash table.

Designing Distinct Hash Functions

- Recall that the goal of designing a hash functions is that they map most sets of keys such that the maximal number of collisions is small.

- When having more than one hash functions, we have the additional goal that the different functions map same keys approximately independently. In Python, we could try variants of good ole hash.

For example:

```
def hash0(x):
    return hash("0" + str(x))
def hash1(x):
    return hash("1" + str(x))
def hash2(x):
    return hash(str(x) + "2")
def hash3(x):
    return hash(str(x) + "3")
```

Designing Distinct Hash Functions

A reminder concerning str (mapping objects to representing strings):

```
>>> [str(i) for i in range(10,20)]
['10', '11', '12', '13', '14', '15', '16', '17', '18', '19']
>>> str(2.2)
'2.2'
>>> str("2.2")
'2.2'
```

And now applying the four functions on a small domain:

```
>>> for f in (hash0,hash1,hash2,hash3):
        print([f(i) %23 for i in range(10,20)])
[ 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
[ 3, 2, 5, 4, 22, 21, 1, 0, 11, 10]
[12, 5, 17, 10, 16, 9, 7, 0, 10, 22]
[13, 4, 18, 9, 17, 8, 8, 22, 11, 21]
```

Random? Independent? Mixing well? You be the judges.

Cuckoo Hashing:
Python Implementation and Analysis

Will not be done in class.

May appear in HW.

Hash Functions: Wrap Up

- Hash functions map large domains to smaller ranges.
- 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, we cannot avoid 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).

Wrap Up: Dictionary in Computer Science

- In computer science, a dictionary is a data structure supporting efficient insert, delete, and search operations.
- This is an abstract notion, and 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).
- 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 unique (different items have different keys).
- Hash tables provide an excellent way to implement dictionaries.

Using Hash Functions and Tables: Wrap Up

- We explained chaining as a way to resolve collisions.
- We also studied the paradigm of cuckoo hashing , using two hash functions $h_1()$, $h_2()$ (or four, or eight).

  Cuckoo hashing is aimed at a constant time find operation, with high probability, at the cost of a slightly longer insert operation.
- In the data structures course, you will see additional collisions resolution means, such as double hashing, etc.
- Python sets and dictionaries use hash tables, thus searching an element in a set / dict takes O(1) time on average. Collisions are solved using open addressing, in a more sophisticated manner. In addition, the size of the hash table is dynamic.

Search

- 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.

- Earlier in this course we have dealt with much simpler data structure that support search:
  - unordered list
  - ordered list (which allows binary search)

Static Data Structures

- 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 Structures

- A dynamic data structure in one that supports efficient insertion and deletion.

- examples:
  - linked list
  - (balanced) binary search tree
  - hash table