Extended Introduction to Computer Science
CS1001.py

Lecture 18:
Hash Tables (cont.)
Finite and Infinite Iterators

Instructors: Daniel Deutch, Amir Rubinstein
Teaching Assistants: Michal Kleinbort, Amir Gilad

Founding instructor: Benny Chor

School of Computer Science
Tel-Aviv University
Winter Semester, 2017-18
http://tau-cs1001-py.wikidot.com

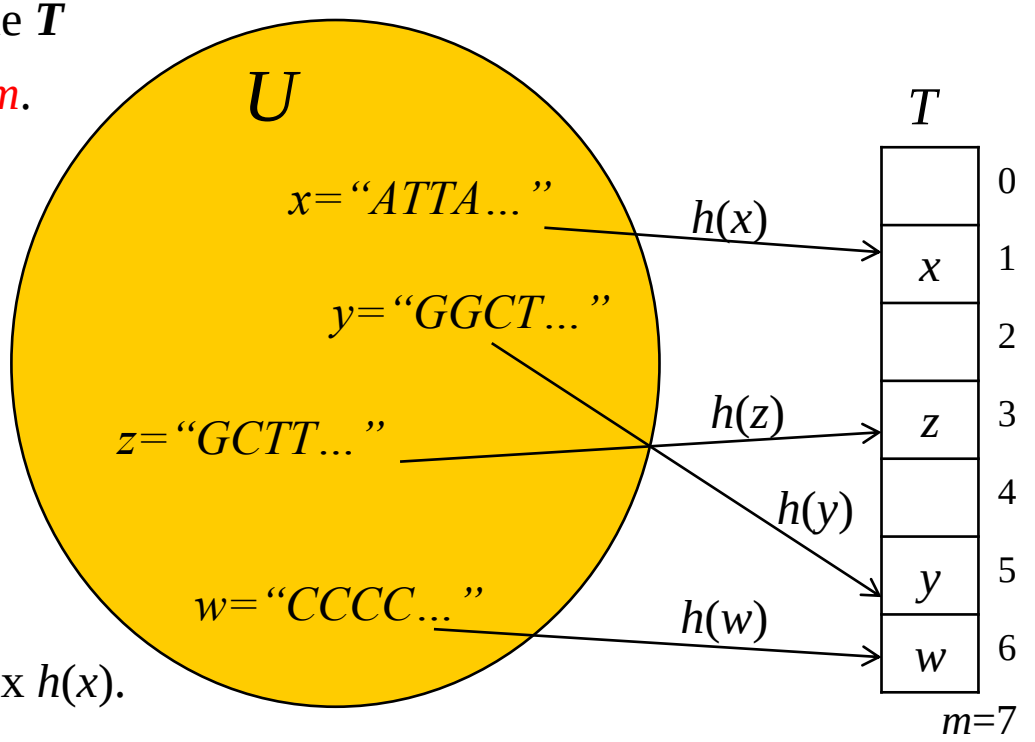
Hash Tables - Reminder

- 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 $h(x)$.

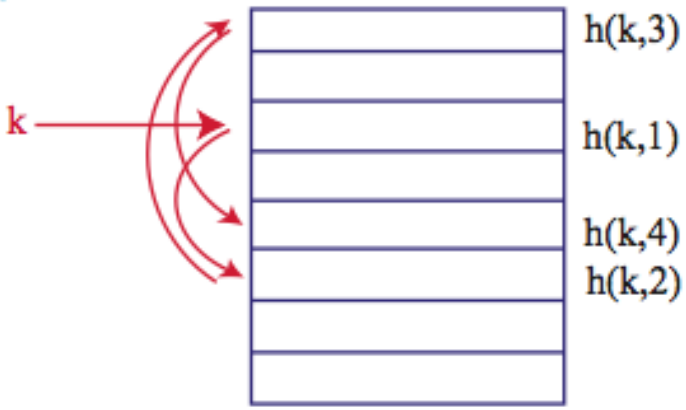
Two Approaches for Dealing with Collisions

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 .

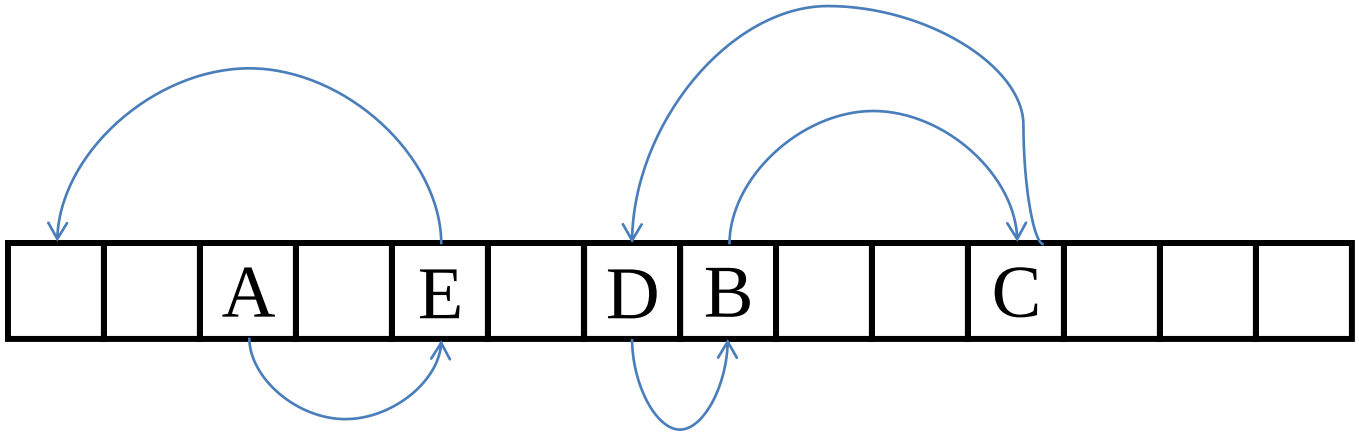
Cuckoo Hashing

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.

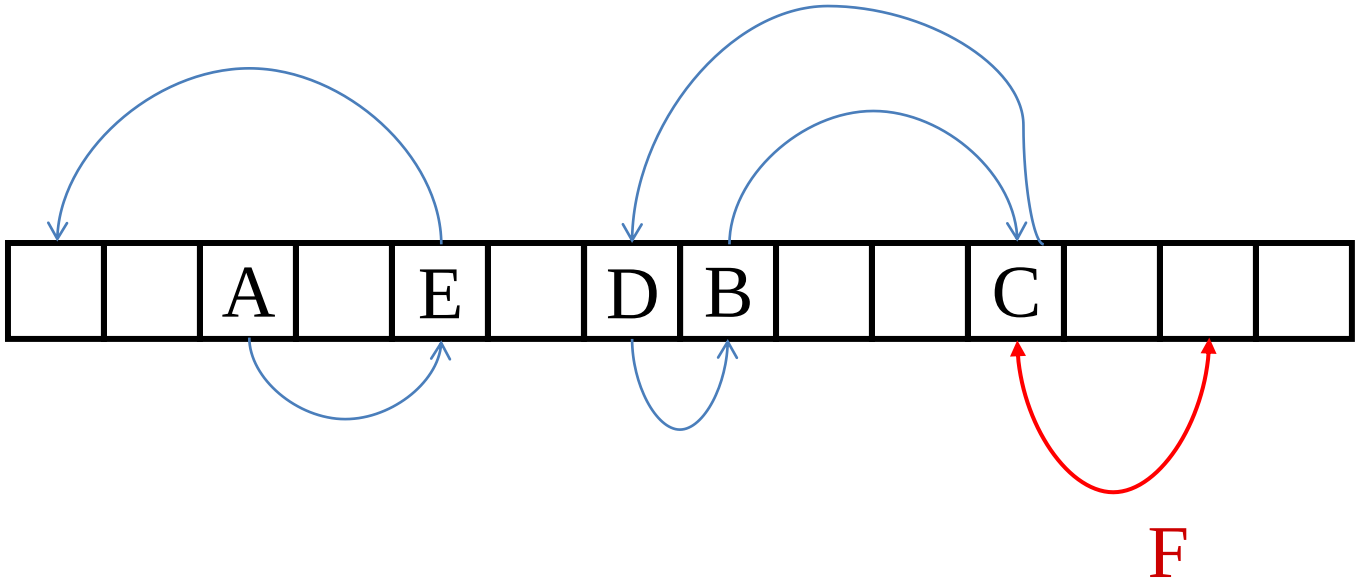
- Otherwise, the slot is occupied by some y . Place this y in its other slot, potentially kicking its present occupant, etc.,etc., until we find an empty slot.

Cuckoo Hashing: Examples



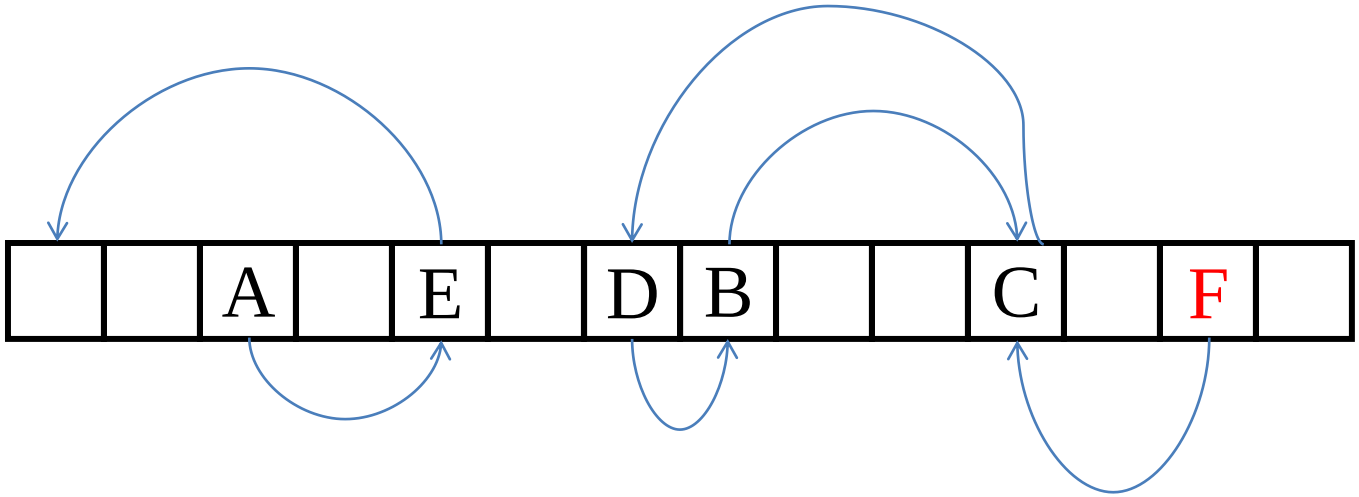
= The other potential slot for an item

Cuckoo Hashing: Examples



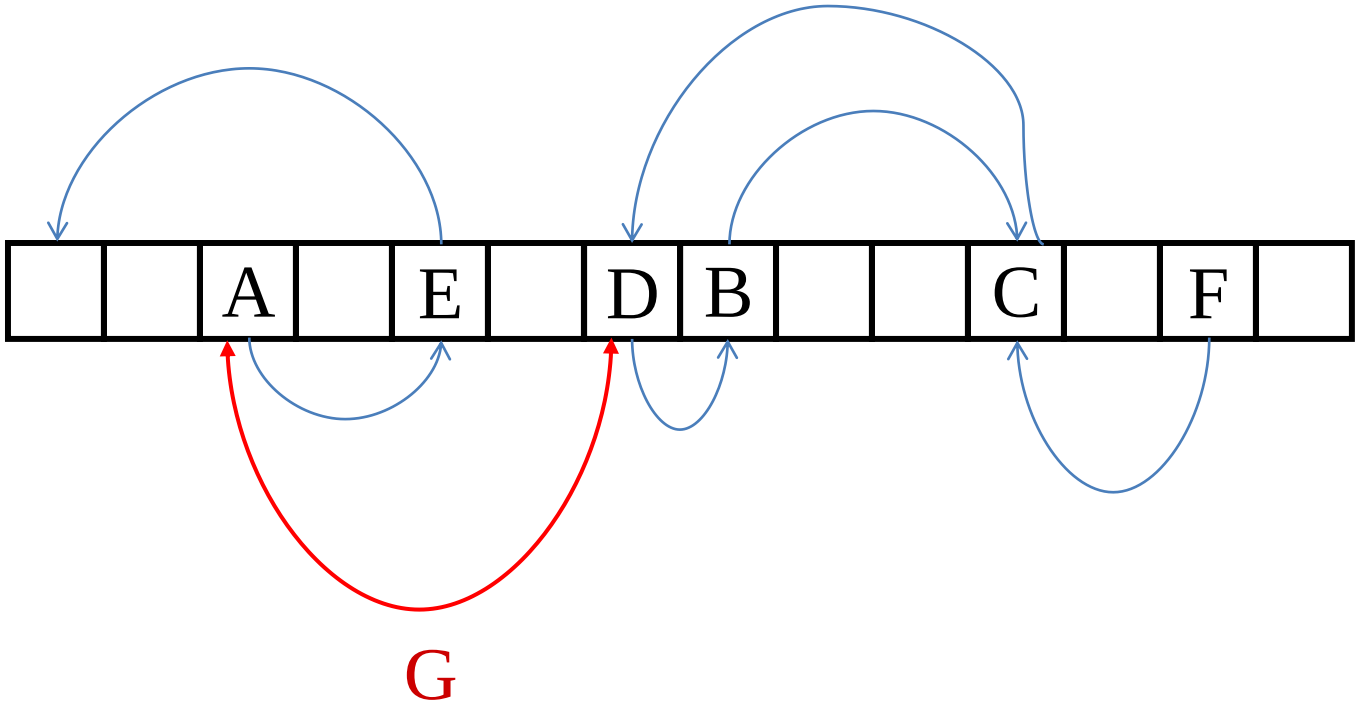
= The other potential slot for an item

Cuckoo Hashing: Examples



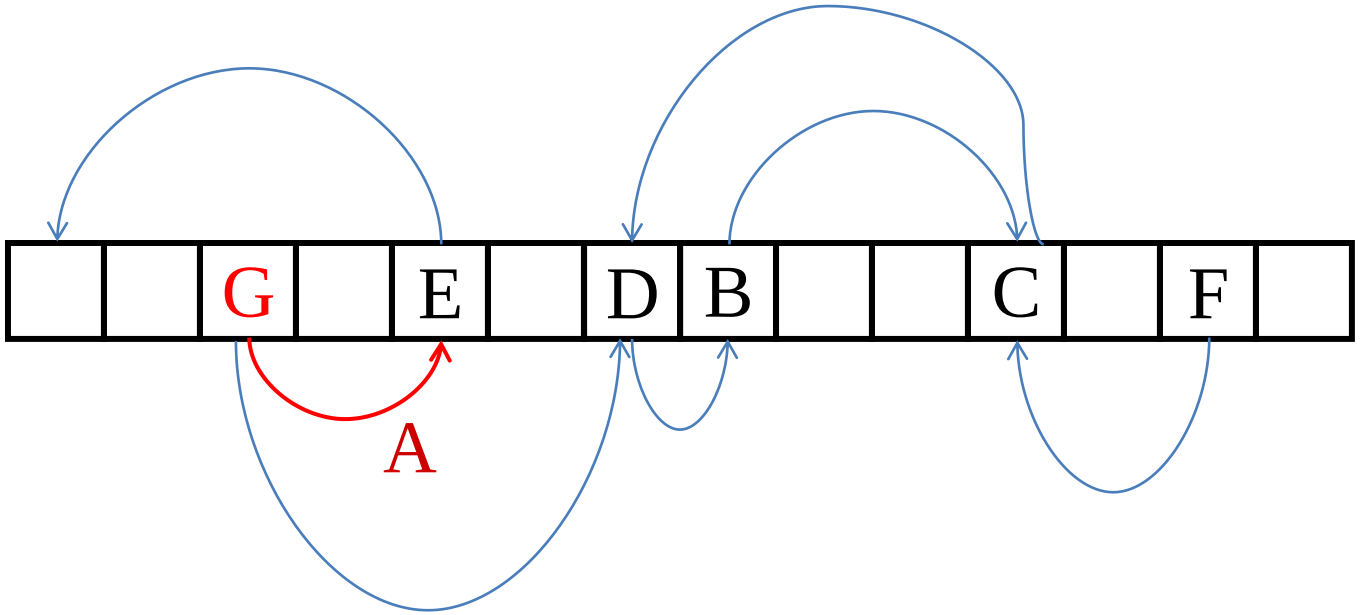
= The other potential slot for an item

Cuckoo Hashing: Examples



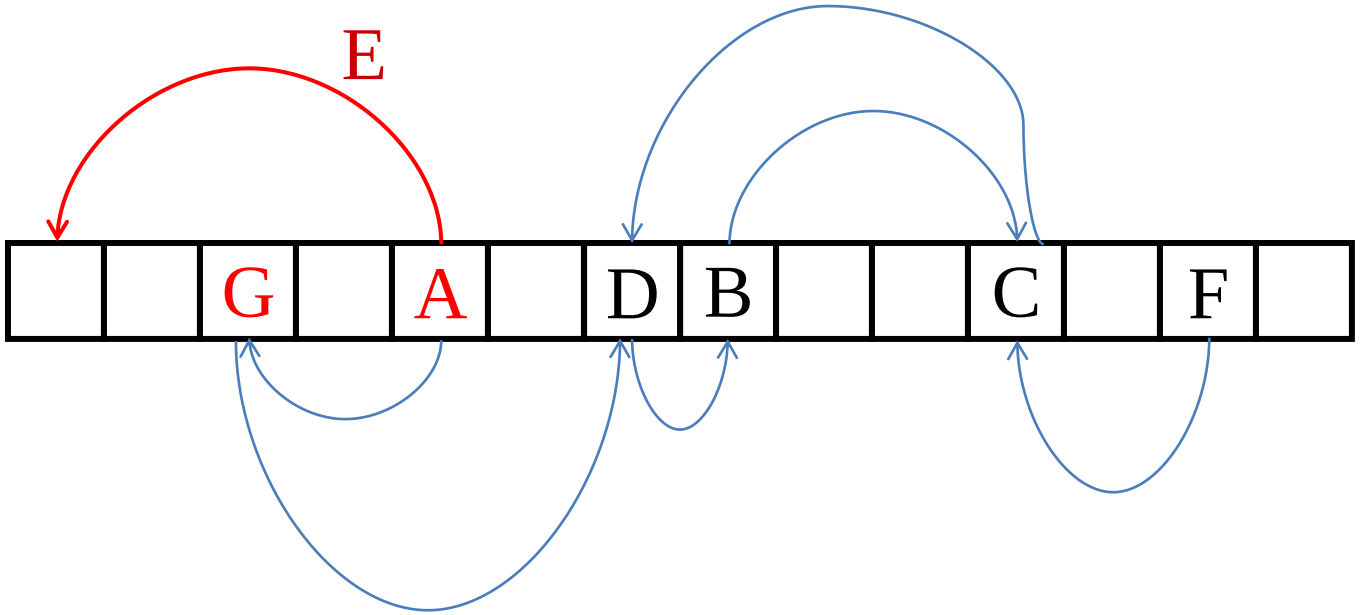
= The other potential slot for an item

Cuckoo Hashing: Examples



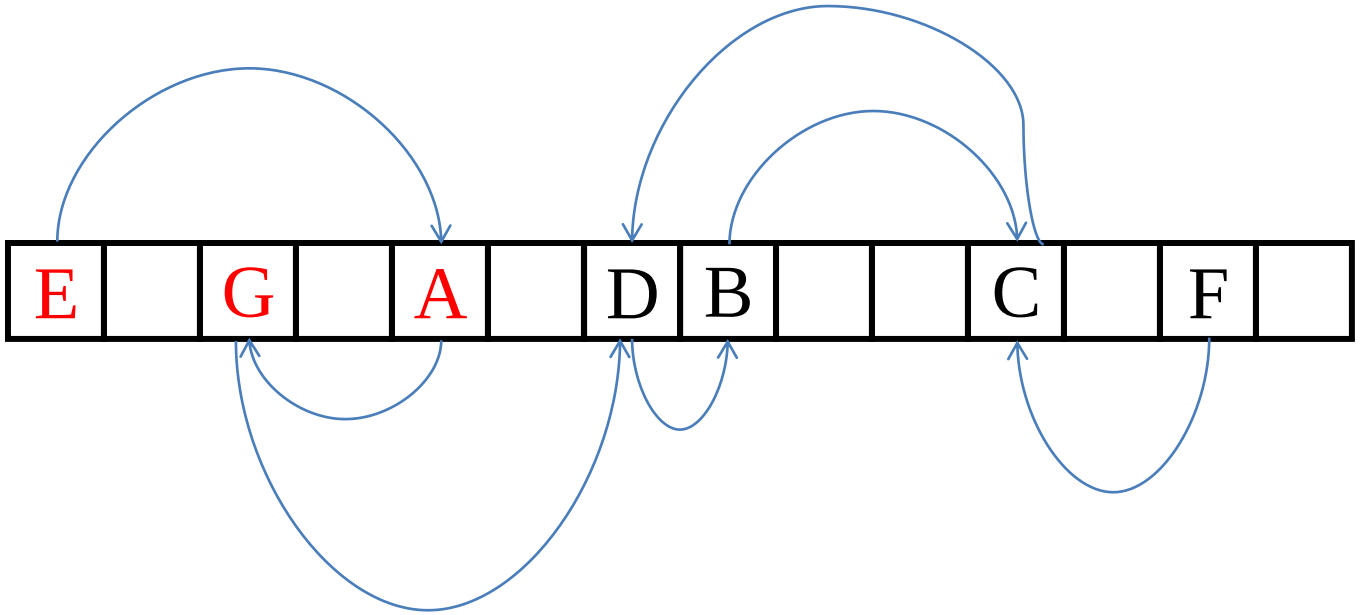
= The other potential slot for an item

Cuckoo Hashing: Examples



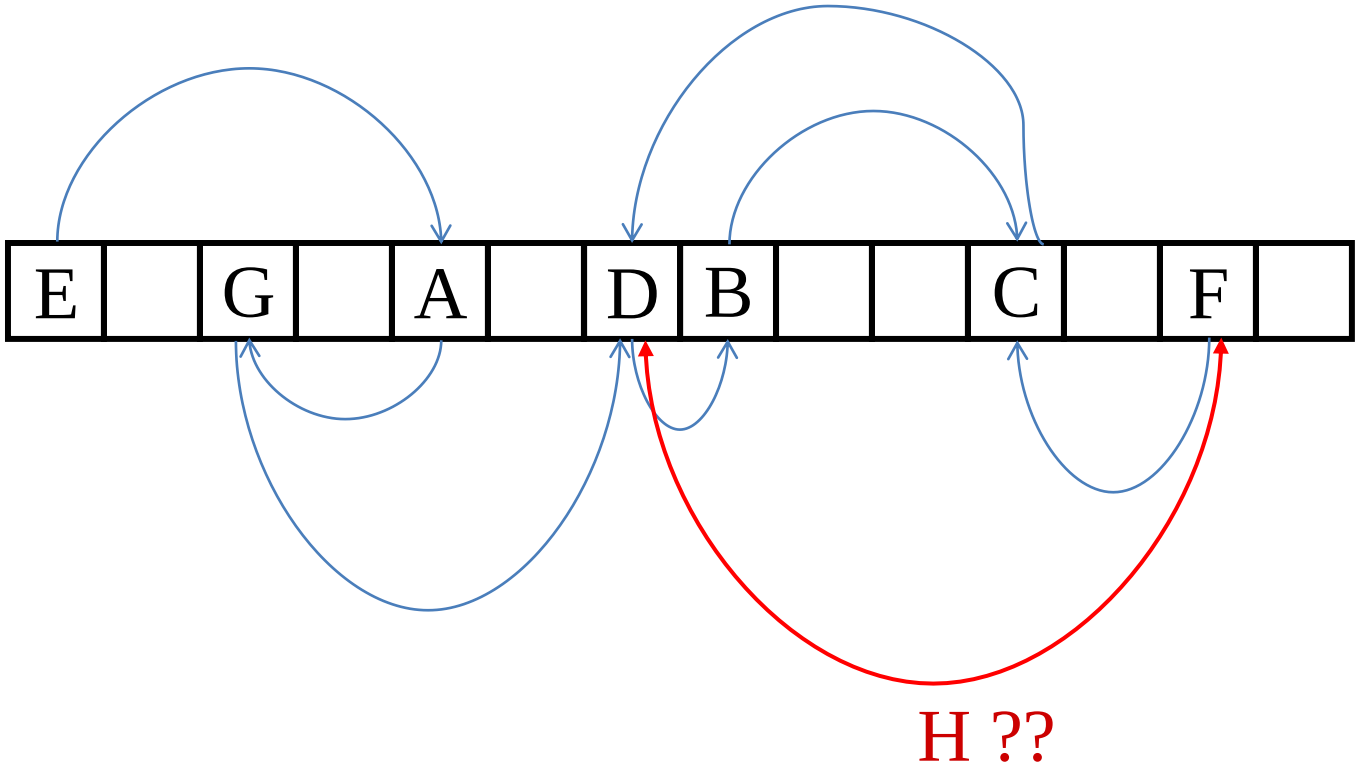
= The other potential slot for an item

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.

Iterators and Generators

- Linked lists and Python's built-in lists (arrays) are two ways to represent a collection of elements. There are others, such as trees, dictionaries, and more.

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

Iterables

- An iterable object is an object capable of returning its members one at a time.

- In particular, we can use a for loop on iterables

- Examples of iterables include:
  - all sequence types (such as list, str , tuple and range)
  - some non-sequence types like dict, set and File
  - and objects of any user defined classes with an __iter__() or __getitem__() method (that is, this is how you make your new class iterable. But we will not see this)

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

Iterables

- An iterable object is an object capable of returning its members one at a time.
- In particular, we can use a for loop on iterables
- Examples of iterables include:
  - all sequence types (such as list, str , tuple and range)
  - some non-sequence types like dict, set and File
  - and objects of any user defined classes with an __iter__() or __getitem__() method (that is, this is how you make your new class iterable. But we will not see this).

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

range is a special iterable class.

```
>>> a=range(10)
>>> type(a)
<class 'range'>
>>> a
range(0, 10)
>>> a[2]
2
```

Iterators

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

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

Iterables and Iterators

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

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

Iterables and Iterators, cont.

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

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

Iterables and Iterators, cont.

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

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

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

Thou Shalt Not Modify an iterable during Iteration

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

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

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

adapted from

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

Iterators as a Tool for Abstraction

- The use of iterators hides the implementation of data collections. For example, when we see the code

```
for x in SomeCollection:
    ….
```

- We do not know if SomeCollection is a list, a dict, or any user defined data collection. Furthemore, we can later modify the implementation of SomeCollection, for example change it from a list to a dict, and the code using it (called client code) will not have to be changed.

- Similarly when we use next(it), it may be an iterator of any kind of a data collection, with any order of traversal.

Iterables, Iterators, and Generators: More Examples

```
>>> mylist = [x for x in range(10**8)]
>>> it1 = iter(mylist)
>>> it2 = (x for x in range(10**8))  #note the () instead of []

>>> type(mylist)
<class 'list'>
>>> type(it1)
<class 'list_iterator'>
>>> type(it2)
<class 'generator'>     #list comprehension syntax inside () is one way to
                        #create a generator. The other will be shown next

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

Iterables, Iterators, and Generators, cont.

- it1 and it2 were iterators representing the first 10^8 integers.
- These integers can fit in just under 1GB RAM. But, can we have iterators representing even more items?

```
>>> it3 = (x for x in range(2**100))  #again, note the ()
>>> next(it3)
0
>>> next(it3)
1
```

- An iterable with 2^{100} elements will not fit in Amazon, Google, and NASA computers, even if taken together.
- Iterators and generators represent streams, but produce only one element at a time. Therefore, there is no problem representing a 2^{100} long stream!

Generators for Infinite Streams

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

To do that, we will introduce generator functions.

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

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

Generators for Infinite Streams, cont.

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

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

Generators, cont.

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

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

- We see that nat has a state , which is retained, unchanged, between successive calls.
- We can have additional instances of the same generator function.

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

A Fibonacci Numbers Generator

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

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

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

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

Execution Specification

When a yield statement

yield expression_list

is encountered, the state of the function is frozen , and the value of expression_list is returned to the caller of next.

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

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

This is similar to what is called co-routine, in contrast with a normal function call which is also called sub-routine.

Merging Sorted, Infinite Iterators

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

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

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

Merging Sorted, Infinite iterators: Execution

```
>>> nat1=natural()
>>> nat2=natural()
>>> nat3=merge(nat1,nat2)
```

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

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

An Attempt to Merge Sorted, Finite iterators

Should the iterators in merge really be infinite ?

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

What went wrong?
The merged iterator, nat3, was not yet exhausted, yet one of the arguments to merge, nat2 was exhausted. The merging procedure still invoked next(iter2) . This has caused a StopIteration error.

Handling Errors: try and except

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

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

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

Handling Errors: try and except: example

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

Let us now apply this function in two different cases:

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

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

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

More on try and except

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

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

We will also use try/except when it is either impossible or expensive to check for the condition in advance. Example – when we invert a matrix, checking in advance that it is not singular would take as much time as inverting, so it makes more sense to try to invert, and raise an exception if we discover that the matrix is singular while we do it.

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

For loop

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

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

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

```
a
b
c
```

Is the same as

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

```
a
b
c
```

Merging Any Non-Empty, Sorted iterators

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

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

merge3 : cont.

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

Merge3: Examples of Executions

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

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

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

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