

# Extended Introduction to Computer Science

## CS1001.py

### Module G:

## Open Indexing and Cuckoo Hashing (for reference only)

Instructors: Elhanan Borenstein, Michal Kleinbort

Teaching Assistants: Noam Parzanchevsky,

Omri Porat, Moshe Shechner

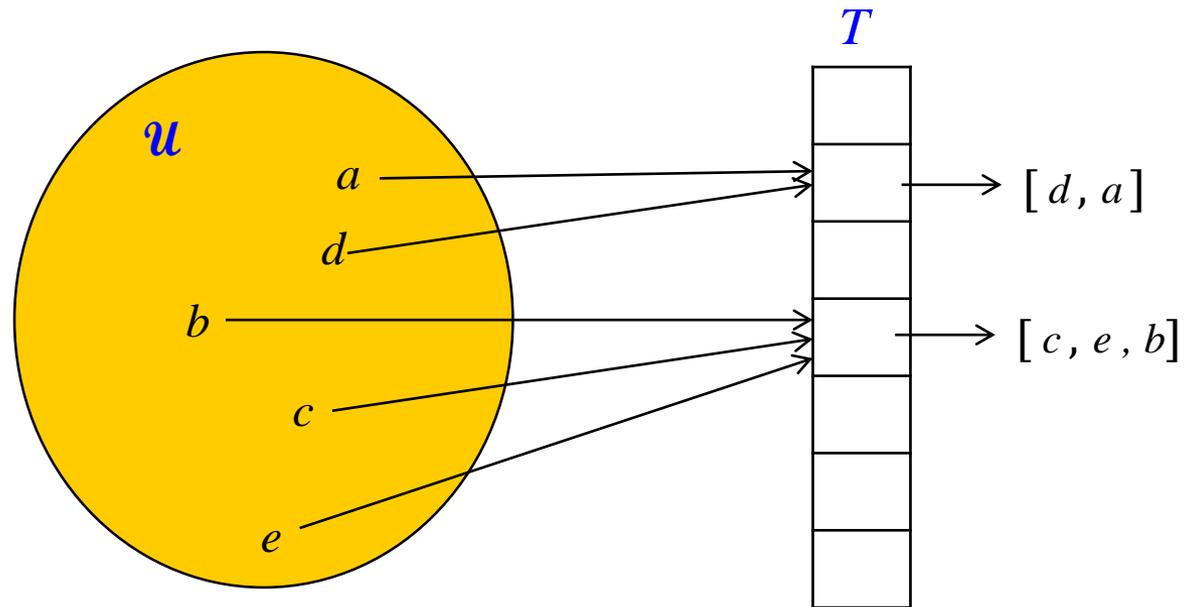
School of Computer Science

Tel-Aviv University

<http://tau-cs1001-py.wikidot.com>

# Dealing with Collisions: **Chaining** Method

- Each **cell** in the table will contain a **list** (can be linked or not), with all the elements  $h$  maps to this cell



- How do we **insert**, **search** and **delete** elements?

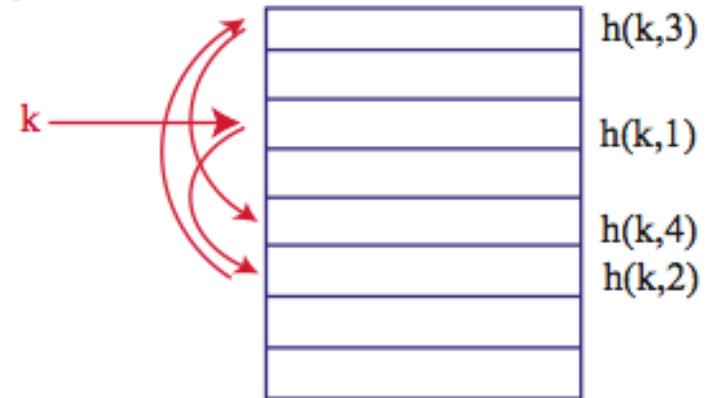
# Two Approaches for Dealing with Collisions

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

# Cuckoo Hashing

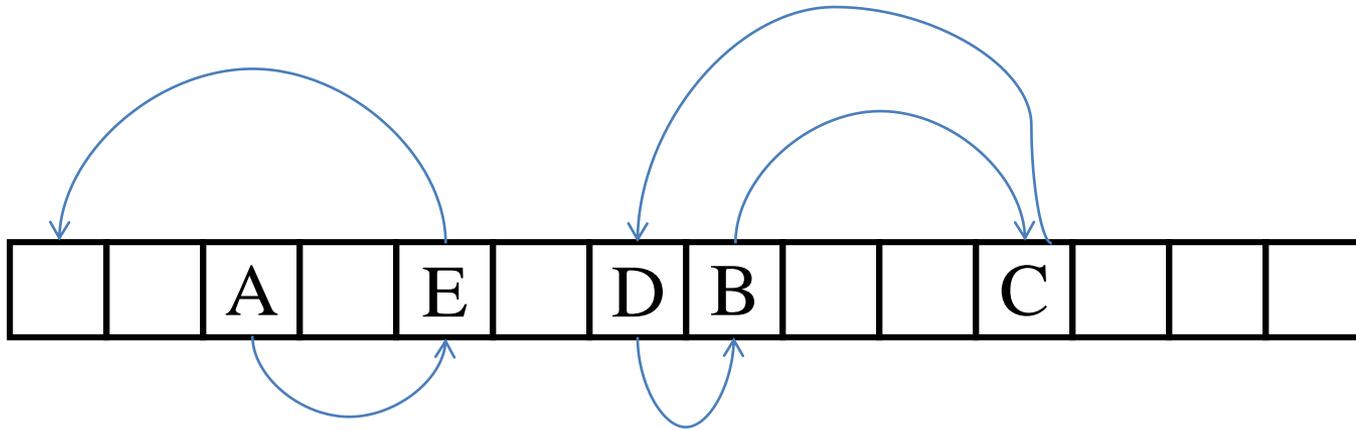
- **Cuckoo hashing** uses two distinct hash functions,  $h_1$  and  $h_2$  (improved versions use four, six, or eight, but the idea is the same).
- Each key,  $k$ , has **two potential slots** in the hash table,  $h_1(k)$  and  $h_2(k)$ . If we search for  $k$ , all we have to do is look for it in these 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  $k$  .

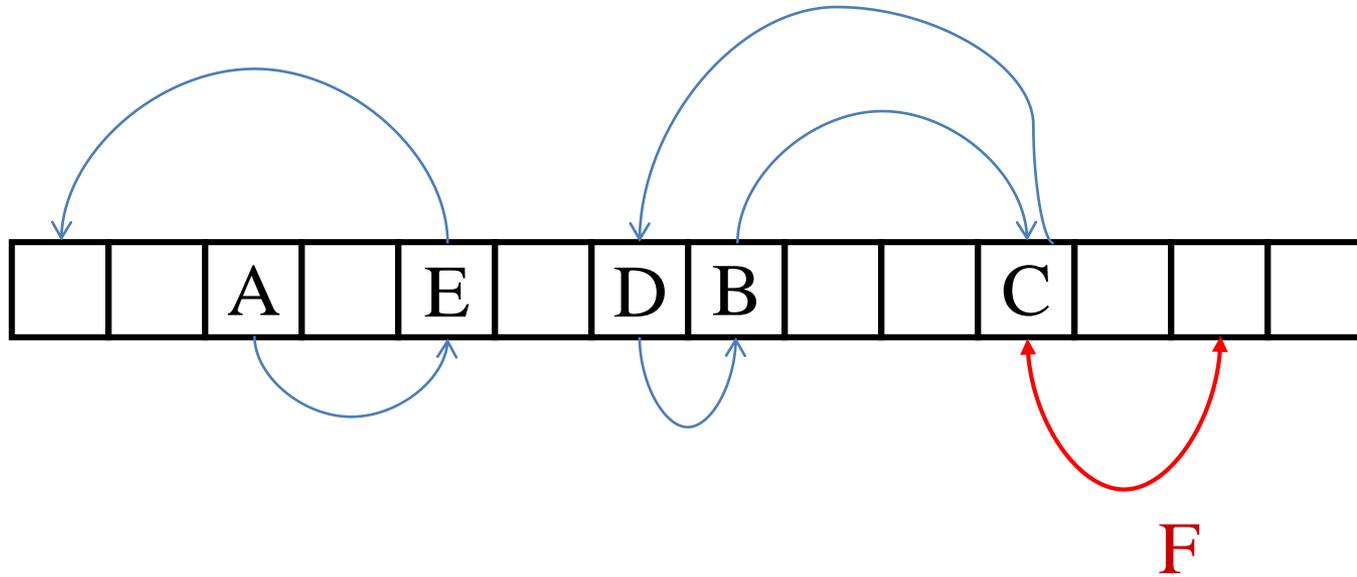
- If any of the two slots,  $h_1(k)$  or  $h_2(k)$  is empty,  $k$  is inserted there.
- If both slots are full, pick one of the two occupants, say  $x$ . Place  $k$  in  $x$ 's current slot.
- Assume this was location  $h_1(x)$ . Place  $x$  in its other slot,  $h_2(x)$ .
- If that slot was empty, we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



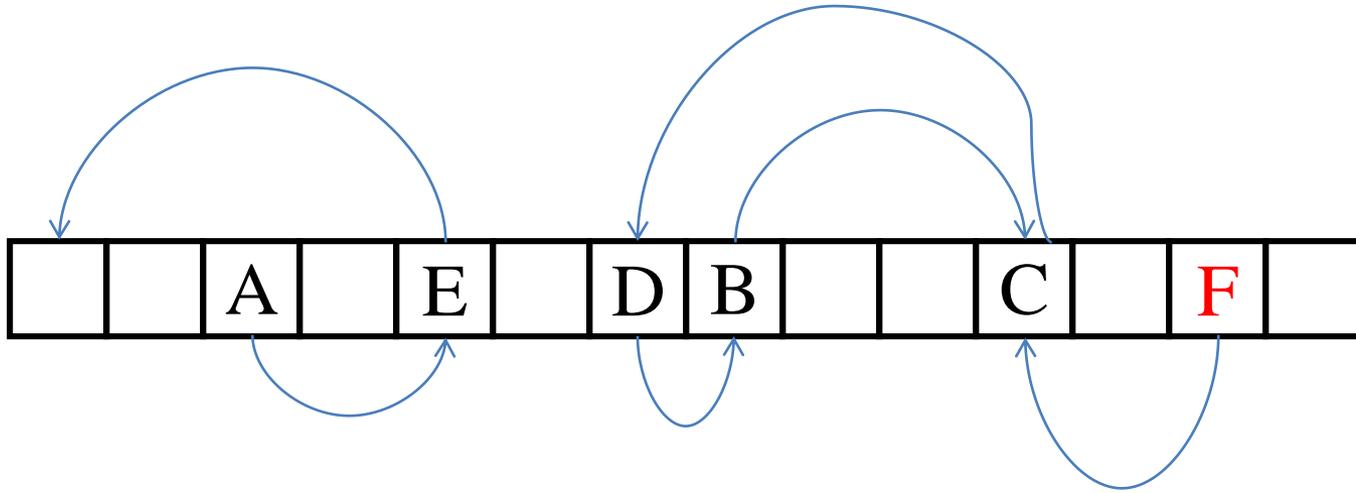
= The other potential slot for an item

# Cuckoo Hashing: Examples



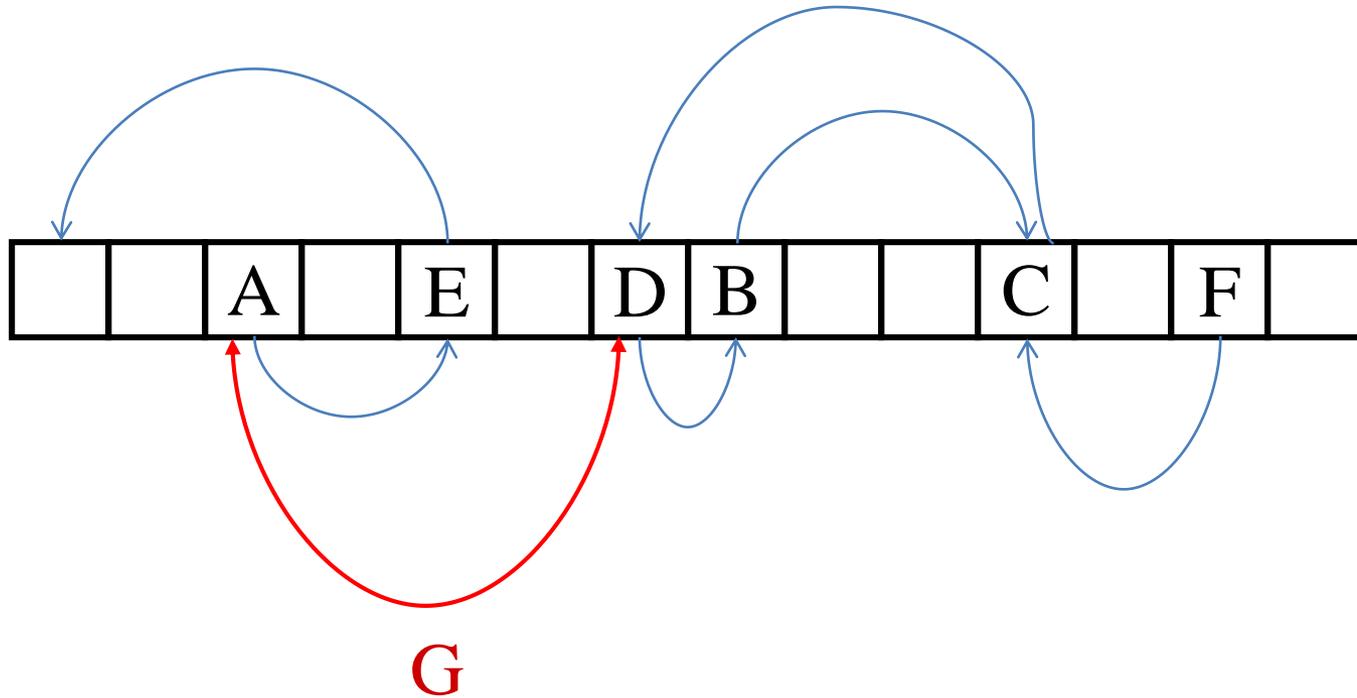
= The other potential slot for an item

# Cuckoo Hashing: Examples



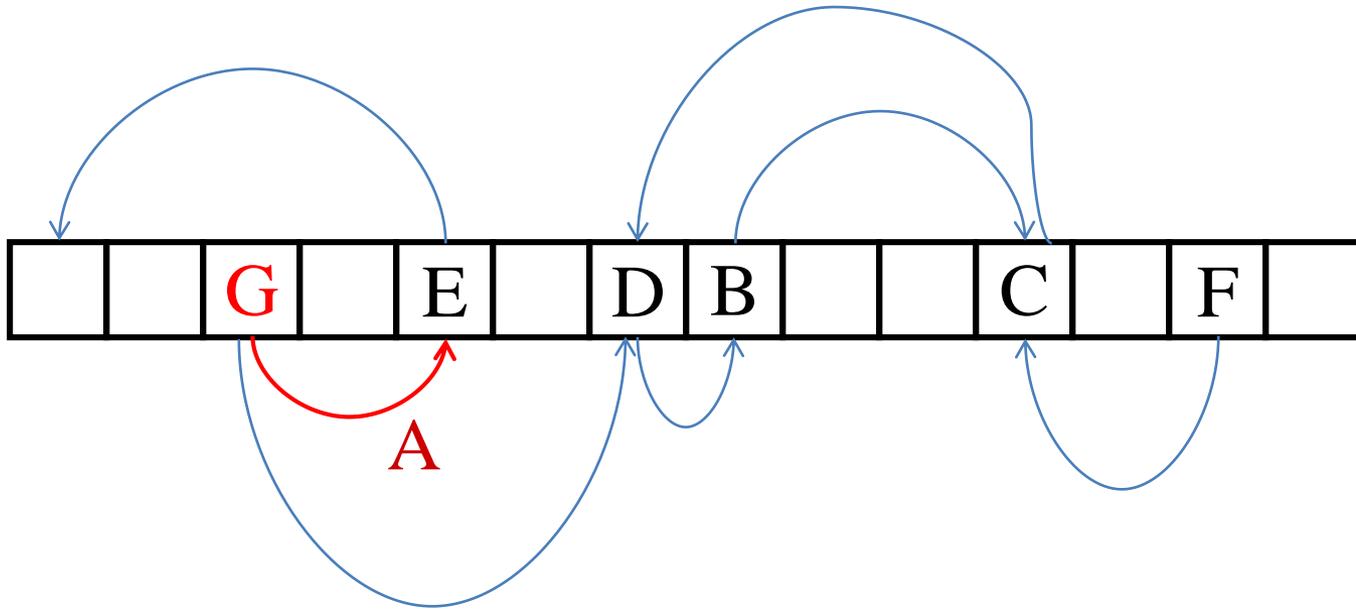
= The other potential slot for an item

# Cuckoo Hashing: Examples



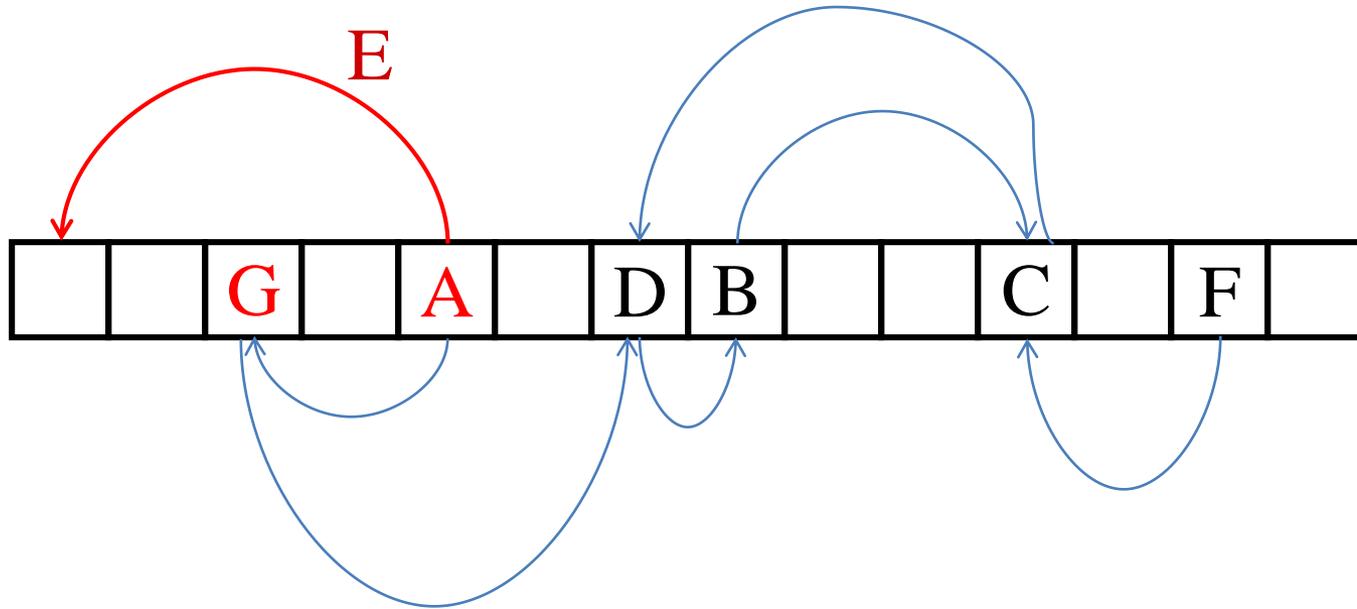
= The other potential slot for an item

# Cuckoo Hashing: Examples



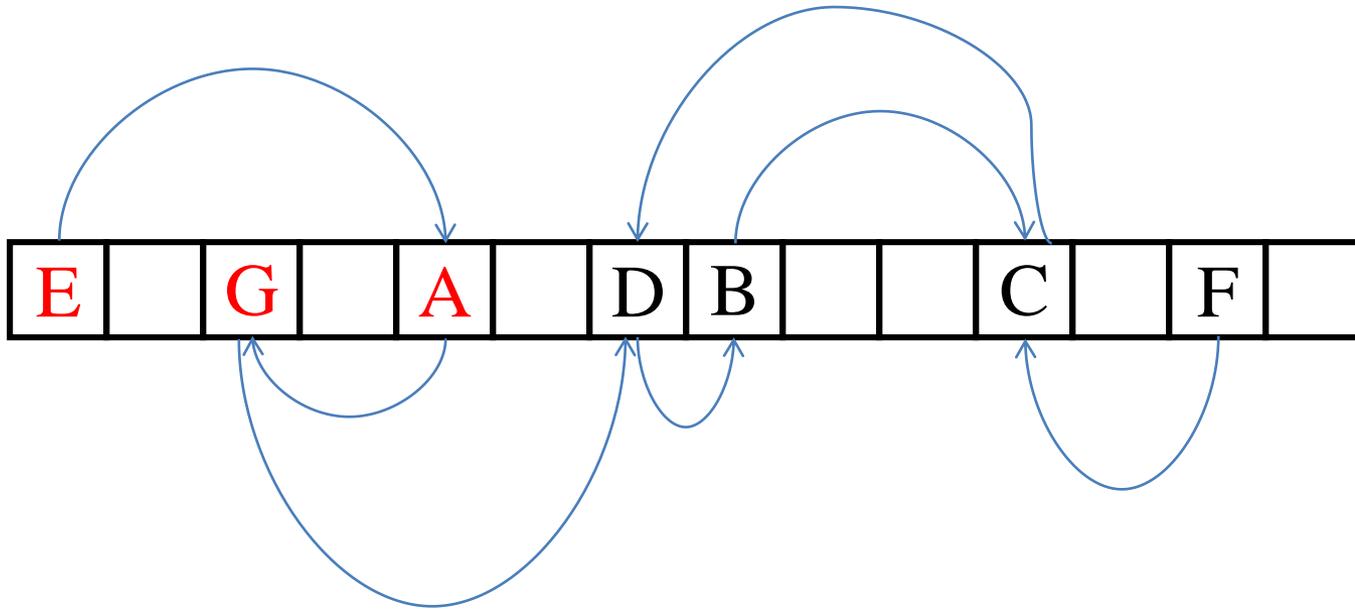
= The other potential slot for an item

# Cuckoo Hashing: Examples



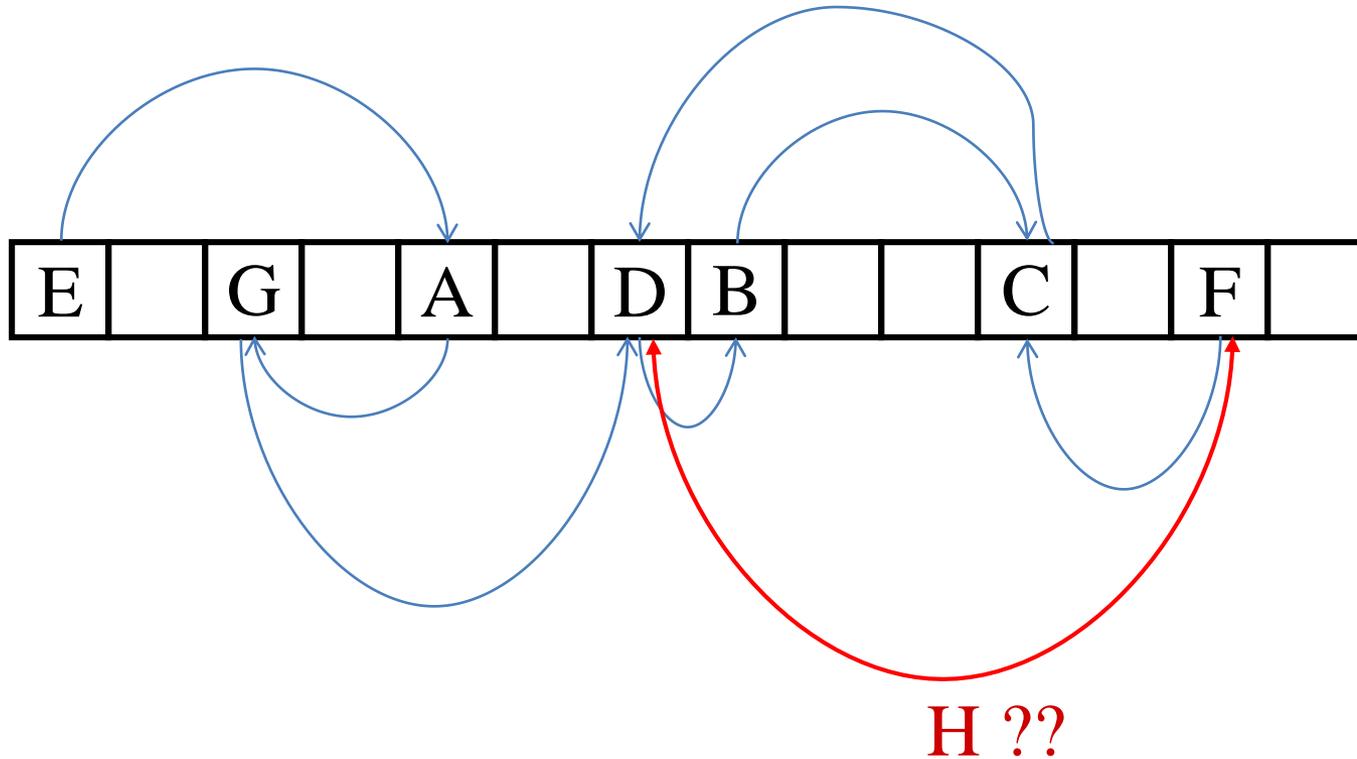
= The other potential slot for an item

# 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 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 ( $4+4\cdot 3=16$  different locations, typically), we go one more level down this search tree ( $12\cdot 3 = 36$  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.