DATA STRUCTURES AND ALGORITHMS

LECTURE 10

Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017

In Lecture 9...

- Hash tables

Today

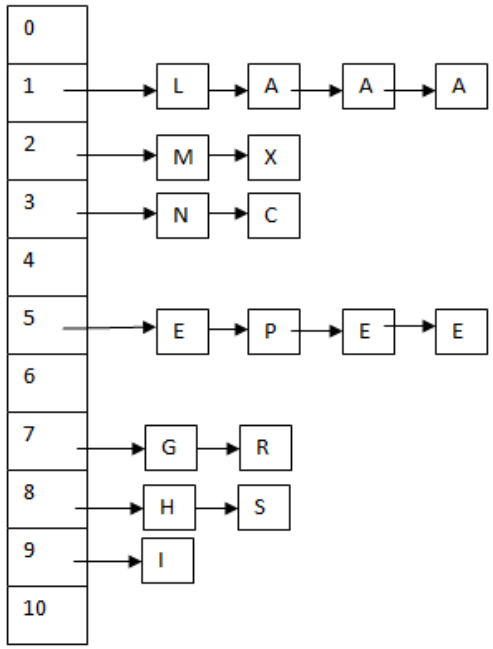
1. Hash tables

Hash table - reminder

- Hash tables are tables (arrays) - one consecutive memory zone - but they are not filled with elements from left to right as regular arrays are.
- For every element to be added a unique position is generated in the table using a *hash function*.
- This position is generated for the element when it has to be removed or when we search for it.
- When the hash function generates the same position for two distinct elements we have a *collision* and we need a method to resolve it.

Hash table - reminder II

- In case of *separate chaining* at each position from the table we have a linked list.



- An important value for hash tables is the *load factor*, α, which is computed as: n/m.
- For separate chaining α can be larger than 1.

Coalesced chaining

- Collision resolution by coalesced chaining: each element from the hash table is stored inside the table (no linked lists), but each element has a *next* field, similar to a linked list on array.
- When a new element has to be inserted and the position where it should be placed is occupied, we will put it to any empty position, and set the *next* link, so that the element can be found in a search.
- Since elements are in the table, α can be at most 1.

Coalesced chaining - example

- Consider a hash table of size $m = 19$ that uses coalesced chaining for collision resolution and a hash function with the division method
- Insert into the table the letters from *A SEARCHING EXAMPLE* (space is ignored)
- For each letter, the *hashCode* is the index of the letter in the alphabet.

Letter	A	S	E	R	C	H	I	N	G	X	M	P	L
HashCode	1	19	5	18	3	8	9	14	7	24	13	16	12
h(Letter)	1	0	5	18	3	8	9	14	7	5	13	16	12

Coalesced chaining - example

position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T	S	A	A	C	E	E	X	G	H	I	A	E	L	M	N		P		R
next	-1	2	10	-1	6	4	11	-1	-1	-1	-1	-1	-1	-1	-1		-1		-1

- $m = 19$
- $\alpha = 0.89$
- $firstFree = 15$

Coalesced chaining - representation

- What fields do we need to represent a hash table where collision resolution is done with coalesced chaining?

HashTable:
T: TKey[]
next: Integer[]
m: Integer
firstFree: Integer
h: TFunction

- For simplicity, in the following, we will consider only the keys.

Coalesced chaining - insert

```
subalgorithm insert (ht, k) is:
//pre: ht is a HashTable, k is a TKey
//post: k was added into ht
    pos ← ht.h(k)
    if ht.T[pos] = -1 then //-1 means empty position
        ht.T[pos] ← k
        ht.next[pos] ← -1
    else
        current ← pos
        while ht.next[current] ≠ -1 execute
            current ← ht.next[current]
        end-while
        if ht.firstFree = ht.m then
            @resize and rehash
        else
//continued on the next slide...
```

Coalesced chaining - insert

```
        ht.T[ht.firstFree] ← k
        ht.next[ht.firstFree] ← - 1
        ht.next[current] ← ht.firstFree
        changeFirstFree(ht)
    end-if
  end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$ on average, $\Theta(n)$ - worst case

Coalesced chaining - ChangeFirstFree

subalgorithm changeFirstFree(ht) **is:**
//pre: ht is a HashTable
//post: the value of ht.firstFree is set to the next free position
  ht.firstFree $\leftarrow$ ht.firstFree + 1
  while ht.firstFree $<$ ht.m **and** ht.T[ht.firstFree] = -1 **execute**
    ht.firstFree $\leftarrow$ ht.firstFree + 1
  end-while
end-subalgorithm

- Complexity: $O(m)$
- *Think about it:* Should we keep the free spaces linked in a list as in case of a linked lists on array?
- What it we kept the empty spaces in a doubly linked list?

Coalesced chaining

- *Remove* and *search* operations for coalesced chaining will be discussed in Seminar 6.

- How can we define an iterator for a hash table with coalesced chaining? What should the following operations do?
  - init
  - getCurrent
  - next
  - valid

Open addressing

- In case of open addressing every element of the hash table is inside the table, we have no pointers, no next links.
- When we want to insert a new element, we will successively generate positions for the element, check (*probe*) the generated position, and place the element in the first available one.

Open addressing

- In order to generate multiple positions, we will extend the hash function and add to it another parameter, i, which is the *probe number* and starts from 0.

$$h : U \times \{0, 1, ..., m-1\} \rightarrow \{0, 1, ..., m-1\}$$

- For an element k, we will successively examine the positions $< h(k, 0), h(k, 1), h(k, 2), ..., h(k, m-1) >$ - called the *probe sequence*
- The *probe sequence* should be a permutation of a hash table positions $\{0, ..., m-1\}$, so that eventually every slot is considered.

Open addressing - Linear probing

- One version of defining the hash function is to use linear probing:

$$h(k,i) = (h'(k) + i) \ mod \ m \ \ \forall i = 0, ..., m-1$$

- where $h'(k)$ is a *simple* hash function (for example: $h'(k) = k \ mod \ m$)

- the *probe sequence* for linear probing is: $< h'(k), h'(k)+1, h'(k)+2, ..., m-1, 0, 1, ..., h'(k)-1 >$

Open addressing - Linear probing - example

- Consider a hash table of size $m = 19$ that uses open addressing with linear probing for collision resolution (h'(k) is a hash function defined with the division method)
- Insert into the table the letters from *A SEARCHING EXAMPLE* (space is ignored)
- For each letter, the *hashCode* is the index of the letter in the alphabet.

Letter	A	S	E	R	C	H	I	N	G	X	M	P	L
HashCode	1	19	5	18	3	8	9	14	7	24	13	16	12
h'(Letter)	1	0	5	18	3	8	9	14	7	5	13	16	12

Open addressing - Linear probing - example

position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T	S	A	A	C	A	E	E	G	H	I	X	E	L	M	N		P		R

- $\alpha = 0.89$

Open addressing - Linear probing - example

position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T	S	A	A	C	A	E	E	G	H	I	X	E	L	M	N		P		R

- $\alpha = 0.89$
- Disadvantages of linear probing:
  - There are only m distinct probe sequences (once you have the starting position everything is fixed)
  - *Primary clustering* - long runs of occupied slots

Open addressing - Quadratic probing

- In case of quadratic probing the hash function becomes:

$$h(k,i) = (h'(k) + c_1 * i + c_2 * i^2) \ mod \ m \ \ \forall i = 0, ..., m-1$$

- where $h'(k)$ is a *simple* hash function (for example: $h'(k) = k \ mod \ m$) and c_1 and c_2 are constants initialized when the hash function is initialized. c_2 should not be 0.

- Considering a simplified version of $h(k,i)$ with $c_1 = 0$ and $c_2 = 1$ the probe sequence would be: $< k, k+1, k+4, k+9, k+16, ... >$

Open addressing - Quadratic probing

- One important issue with quadratic probing is how can we choose the values of m, c_1 and c_2 so that the probe sequence is a permutation.
- If m is a prime number, only the first half of the probe sequence is unique, so, once the hash table is half full, there is no guarantee that an empty space will be found.
  - For example, for $m = 17$, $c_1 = 3$, $c_2 = 1$ and $k = 13$, the probe sequence is $< 13, 0, 6, 14, 7, 2, 16, 15, 16, 2, 7, 14, 6, 0, 13, 11, 11 >$
  - For example, for $m = 11$, $c_1 = 1$, $c_2 = 1$ and $k = 27$, the probe sequence is $< 5, 7, 0, 6, 3, 2, 3, 6, 0, 7, 5 >$

Open addressing - Quadratic probing

- If m is a power of 2 and $c_1 = c_2 = 0.5$, the probe sequence will always be a permutation. For example for $m = 8$ and $k = 3$:
  - $h(3, 0) = (3 \,\%\, 8 + 0.5 * 0 + 0.5 * 0^2) \,\%\, 8 = 3$
  - $h(3, 1) = (3 \,\%\, 8 + 0.5 * 1 + 0.5 * 1^2) \,\%\, 8 = 4$
  - $h(3, 2) = (3 \,\%\, 8 + 0.5 * 2 + 0.5 * 2^2) \,\%\, 8 = 6$
  - $h(3, 3) = (3 \,\%\, 8 + 0.5 * 3 + 0.5 * 3^2) \,\%\, 8 = 1$
  - $h(3, 4) = (3 \,\%\, 8 + 0.5 * 4 + 0.5 * 4^2) \,\%\, 8 = 5$
  - $h(3, 5) = (3 \,\%\, 8 + 0.5 * 5 + 0.5 * 5^2) \,\%\, 8 = 2$
  - $h(3, 6) = (3 \,\%\, 8 + 0.5 * 6 + 0.5 * 6^2) \,\%\, 8 = 0$
  - $h(3, 7) = (3 \,\%\, 8 + 0.5 * 7 + 0.5 * 7^2) \,\%\, 8 = 7$

Open addressing - Quadratic probing

- If m is a prime number of the form $4 * k + 3$, $c_1 = 0$ and $c_2 = (-1)^i$ (so the probe sequence is +0, -1, +4, -9, etc.) the probe sequence is a permutation. For example for $m = 7$ and $k = 3$:
  - $h(3, 0) = (3 \,\%\, 7 + 0^2) \,\%\, 7 = 3$
  - $h(3, 1) = (3 \,\%\, 7 - 1^2) \,\%\, 7 = 2$
  - $h(3, 2) = (3 \,\%\, 7 + 2^2) \,\%\, 7 = 0$
  - $h(3, 3) = (3 \,\%\, 7 - 3^2) \,\%\, 7 = 1$
  - $h(3, 4) = (3 \,\%\, 7 + 4^2) \,\%\, 7 = 5$
  - $h(3, 5) = (3 \,\%\, 7 - 5^2) \,\%\, 7 = 6$
  - $h(3, 6) = (3 \,\%\, 7 + 6^2) \,\%\, 7 = 4$

Open addressing - Quadratic probing - example

- Consider a hash table of size $m = 16$ that uses open addressing with quadratic probing for collision resolution (h'(k) is a hash function defined with the division method), $c_1 = c_2 = 0.5$.
- Insert into the table the letters from *HASHTABLE*
- For each letter, the *hashCode* is the index of the letter in the alphabet.

Letter	H	A	S	T	B	L	E
HashCode	8	1	19	20	2	12	5
h'(Letter)	8	1	3	4	2	12	5

Open addressing - Quadratic probing - example

position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T		A	A	S	T	B	E		H	H			L			

- Disadvantages of quadratic probing:
  - The performance is sensitive to the values of m, c_1 and c_2.
  - *Secondary clustering* - if two elements have the same initial probe positions, their whole probe sequence will be identical: $h(k_1, 0) = h(k_2, 0) \Rightarrow h(k_1, i) = h(k_2, i)$.
  - There are only m distinct probe sequences (once you have the starting position the whole sequence is fixed).

Open addressing - Double hashing

- In case of double hashing the hash function becomes:

$$h(k,i) = (h'(k) + i * h''(k)) \ \% \ m \ \forall i = 0, ..., m-1$$

- where $h'(k)$ and $h''(k)$ are *simple* hash functions, where $h''(k)$ should never return the value 0.

- For a key, k, the first position examined will be $h'(k)$ and the other probed positions will be computed based on the second hash function, $h''(k)$.

Open addressing - Double hashing

- Similar to quadratic hashing, not every combination of m and $h''(k)$ will return a complete permutation as a probe sequence.
- In order to produce a permutation m and all the values of $h''(k)$ have to be relatively primes. This can be achieved in two ways:
  - Choose m as a power of 2 and design h'' in such a way that it always returns an odd number.
  - Choose m as a prime number and design h'' in such a way that it always returns a value from the $\{0, \text{m-}1\}$ set.

Open addressing - Double hashing

- Choose m as a prime number and design h'' in such a way that it always returns a value from the {0, m-1} set.
- For example:
  $h'(k) = k\%m$
  $h''(k) = 1 + (k\%(m-1))$.
- For $m = 11$ and $k = 36$ we have:
  $h'(36) = 3$
  $h''(36) = 7$
- The probe sequence is: $< 3, 10, 6, 2, 9, 5, 1, 8, 4, 0, 7 >$

Open addressing - Double hashing - example

- Consider a hash table of size $m = 13$ that uses open addressing with double hashing for collision resolution, with $h'(k) = k\%m$ and $h''(k) = 1 + (k\%(m-1))$.
- Insert into the table the letters from *HASHTABLE*
- For each letter, the *hashCode* is the index of the letter in the alphabet.

Letter	H	A	S	T	B	L	E
HashCode	8	1	19	20	2	12	5
h'(Letter)	8	1	3	4	2	12	5
h''(Letter)	9	2	8	9	3	1	6

Open addressing - Double hashing - example

position	0	1	2	3	4	5	6	7	8	9	10	11	12
T		A	B	S	H	A			H	T		E	L

- Main advantage of double hashing is that even if $h(k_1, 0) = h(k_2, 0)$ the probe sequences will be different if $k_1 \neq k_2$.
- For example:
  - Letter A, hashCode 1: $< 1, 3, 5, 7, 9, 11, 0, 2, 4, 6, 8, 10, 12 >$
  - Letter N, hashCode 14: $< 1, 4, 7, 10, 0, 3, 6, 9, 12, 2, 5, 8, 11 >$
- Since for every $(h'(k), h''(k))$ pair we have a separate probe sequence, double hashing generates $\approx m^2$ different permutations.

Open addressing - operations

- In the following we will discuss the implementation of the basic dictionary operations for collision resolution with open addressing.
- In the following, we will use the notation $h(k, i)$ for a hash function, without mentioning whether we have linear probing, quadratic probing or double hashing (code is the same for each of them, implementation of h is different only).

Open addressing - representation

- What fields do we need to represent a hash table with collision resolution with open addressing?

HashTable:
T: TKey
m: Integer
h: TFunction

- For simplicity we will consider that we only have keys.

Open addressing - insert

- What should the *insert* operation do?

```
subalgorithm insert (ht, e) is:
//pre: ht is a HashTable, e is a TKey
//post: e was added in ht
    i ← 0
    pos ← ht.h(e, i)
    while i < ht.m and ht.T[pos] ≠ -1 execute
    //-1 means empty space
        i ← i + 1
        pos ← ht.h(e, i)
    end-while
    if i = ht.m then
        @resize and rehash
    else
        ht.T[pos] ← e
   end-if
end-subalgorithm
```

Open addressing - search

- What should the *search* operation do?

```
function search (ht, e) is:
//pre: ht is a HashTable, e is a TKey
//post: e was added in the ht
    i ← 0
    pos ← ht.h(e, i)
    while i < ht.m and ht.T[pos] ≠ -1 and ht.T.[pos] ≠ e execute
    //-1 means empty space
        i ← i + 1
        pos ← ht.h(e, i)
    end-while
    if i < ht.m and ht.T[pos] = e execute
        search ← True
    else
        search ← False
    end-if
end-function
```

Open addressing - remove

- How can we remove an element from the hash table?

Open addressing - remove

- How can we remove an element from the hash table?
- Removing an element from a hash table with open addressing is not simple:
  - we cannot just mark the position empty - *search* might not find other elements
  - you cannot move elements - *search* might not find other elements
- Remove is usually implemented to mark the deleted position with a special value, *DELETED*.
- How does this special value change the implementation of the *insert* and *search* operation?

Open addressing - Performance

- In a hash table with open addressing with load factor $\alpha = n/m$ ($\alpha < 1$), the *average* number of probes is at most
  - for *insert* and *unsuccessful search*

$$\frac{1}{1-\alpha}$$

  - for *successful search*

$$\frac{1}{\alpha} * ln\frac{1}{1-\alpha}$$

- If α is constant, the complexity is $\Theta(1)$
- Worst case complexity is $O(n)$

Perfect hashing

- Assume that we know all the keys in advance and we use *separate chaining* for collision resolution $\Rightarrow$ the more lists we make, the shorter the lists will be (reduced number of collisions) $\Rightarrow$ if we could make a large number of list, each would have one element only (no collision).
- How large should we make the hash table to make sure that there are no collisions?
- If $M = N^2$, it can be shown that the table is collision free with probability at least $1/2$.
- Start building the hash table. If you detect a collision, just choose a new hash function and start over (expected number of trials is at most 2).

Perfect hashing

- Having a table of size N^2 is impractical.
- Solution instead:
  - Use a hash table of size N (*primary* hash table).
  - Instead of using linked list for collision resolution (as in separate chaining) each element of the hash table is another hash table (*secondary hash table*)
  - Make the secondary hash table of size ${n_j}^2$, where n_j is the number of elements from this hash table.
  - Each secondary hash table will be constructed with a different hash function, and will be reconstructed until it is collision free.
- This is called **perfect hashing**.
- It can be shown that the total space needed for the secondary hash tables is at most $2N$.

Perfect hashing

- Perfect hashing requires multiple hash functions, this is why we use *universal hashing*.

Perfect hashing

- Perfect hashing requires multiple hash functions, this is why we use *universal hashing*.
- Let p be a prime number, larger than the largest possible key.
- The universal hash function family $\mathcal{H}$ can be defined as:

$$\mathcal{H} = \{H_{a,b}(x) = ((a * x + b) \ \% \ p) \ \% \ m),$$

$$\text{where } 1 \leq a \leq p-1, 0 \leq b \leq p-1$$

- a and b are chosen randomly when the hash function is initialized.

Perfect hashing - example

- Insert into a hash table with perfect hashing the letters from "PERFECT HASHING EXAMPLE". Since we want no collisions at all, we are going to consider only the unique letters: "PERFCTHASINGXML"
- Since we are inserting $N = 15$ elements, we will take $m = 15$.
- For each letter, the *hashCode* is the index of the letter in the alphabet.

Letter	P	E	R	F	C	T	H	A	S	I	N	G	X	M	L
HashCode	16	5	18	6	3	20	8	1	19	9	14	7	24	13	12

Perfect hashing - example

- p has to be a prime number larger than the maximum key $\Rightarrow$ 29
- The hash function will be:

$$H_{a,b}(x) = ((a * x + b) \ \% \ p) \ \% \ m$$

- where a will be 3 and b will be 2 (chosen randomly).

Letter	P	E	R	F	C	T	H	A	S	I	N	G	X	M	L
HashCode	16	5	18	6	3	20	8	1	19	9	14	7	24	13	12
H(HashCode)	6	2	12	5	11	4	11	5	1	0	0	8	1	12	9

Perfect hashing - example

- Collisions:
  - position 0 - I, N
  - position 1 - S, X
  - position 2 - E
  - position 4 - T
  - position 5 - F, A
  - position 6 - P
  - position 8 - G
  - position 9 - L
  - position 11 - C, H
  - position 12 - R, M

Perfect hashing - example

- For the positions where we have no collision (only one element hashed to it) we will have a secondary hash table with only one element, and hash function $h(x) = 0$
- For the positions where we have two elements, we will have a secondary hash table with 4 positions and different hash functions, taken from the same universe, with different random values for a and b.
- For example for position 0, we can define $a = 4$ and $b = 11$ and we will have:
  $h(I) = h(9) = 2$
  $h(N) = h(14) = 1$

Perfect hashing - example

- Assume that for the secondary hash table from position 1 we will choose $a = 5$ and $b = 2$.
- Positions for the elements will be:
  $h(S) = h(19) = ((5 * 19 + 2)\%29)\%4 = 2$
  $h(X) = h(24) = ((5 * 24 + 2)\%29)\%4 = 2$
- In perfect hashing we should not have collisions, so we will simply chose another hash function: another random values for a and b. Choosing for example $a = 2$ and $b = 13$, we will have h(S) = 2 and h(X) = 3.

Perfect hashing

- When perfect hashing is used and we search for an element we will have to check at most 2 positions (position in the primary and in the secondary table).
- This means that worst case performance of the table is $\Theta(1)$.
- But in order to use perfect hashing, we need to have static keys: once the table is built, no new elements can be added.

Cuckoo hashing

- In cuckoo hashing we have two hash tables of the same size, each of them more than half empty and each hash table has its hash function (so we have two different hash functions).
- For each element to be added we can compute two positions: one from the first hash table and one from the second. In case of cuckoo hashing, it is guaranteed that an element will be on one of these positions.
- Search is simple, because we only have to look at these two positions.
- Delete is simple, because we only have to look at these two positions and set to empty the one where we find the elements.

Cuckoo hashing

- When we want to insert a new element we will compute its position in the first hash table. If the position is empty, we will place the element there.
- If the position in the first hash table is not empty, we will kick out the element that is currently there, and place the new element into the first hash table.
- The element that was kicked off will be placed at its position in the second hash table. If that position is occupied, we will kick off the element from there and place it into its position in the first hash table.
- We repeat the above process until we will get an empty position for an element.
- If we get back to the same location with the same key we have a cycle and we cannot add this element $\Rightarrow$ resize, rehash

Cuckoo hashing - example

- Assume that we have two hash tables, with $m = 11$ positions and the following hash functions:
  - h1(k) = k % 11
  - h2(k) = (k div 11) % 11
- We will see the step-by-step insertion of the following elements: 20, 50, 53, 75, 100, 67, 105, 3, 36, 39

Position	0	1	2	3	4	5	6	7	8	9	10
T											

Position	0	1	2	3	4	5	6	7	8	9	10
T											

Cuckoo hashing - example I

- Insert key 20
  - h1(20) = 9 - empty position, element added in the first table
- Insert key 50
  - h1(50) = 6 - empty position, element added in the first table

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			20	

Position	0	1	2	3	4	5	6	7	8	9	10
T											

Cuckoo hashing - example II

- Insert key 53
  - h1(53) = 9 - occupied
  - 53 goes in the first hash table, and it sends 20 in the second to position h2(20) = 1

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20									

Cuckoo hashing - example III

- Insert key 75
  - h1(75) = 9 - occupied
  - 75 goes in the first hash table, and it sends 53 in the second to position h2(53) = 4

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			53						

Cuckoo hashing - example IV

- Insert key 100
  - h1(100) = 1 - empty position
- Insert key 67
  - h1(67) = 1 - occupied
  - 67 goes in the first hash table, and it sends 100 in the second to position h2(100) = 9

Position	0	1	2	3	4	5	6	7	8	9	10
T		67					50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			53					100	

Cuckoo hashing - example V

- Insert key 105
  - h1(105) = 6 - occupied
  - 105 goes in the first hash table, and it sends 50 in the second to position h2(50) = 4
  - 50 goes in the second hash table, and it sends 53 to the first one, to position h1(53) = 9
  - 53 goes in the first hash table, and it sends 75 to the second one, to position h2(75) = 6

Position	0	1	2	3	4	5	6	7	8	9	10
T		67					105			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			50		75			100	

Cuckoo hashing - example VI

- Insert key 3
  - h1(3) = 3 - empty position
- Insert key 36
  - h1(36) = 3 - occupied
  - 36 goes in the first hash table, and it sends 3 in the second to position h2(3) = 0

Position	0	1	2	3	4	5	6	7	8	9	10
T		67		36			105			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T	3	20			50		75			100	

Cuckoo hashing - example VII

- Insert key 39
  - h1(39) = 6 - occupied
  - 39 goes in the first hash table and it sends 105 in the second to position h2(105) = 9
  - 105 goes to the second hash table and it sends 100 in the first to position h1(100) = 1
  - 100 goes in the first hash table and it sends 67 in the second to position h2(67) = 6
  - 67 goes in the second hash table and it sends 75 in the first to position h1(75) = 9
  - 75 goes in the first hash table and it sends 53 in the second to position h2(53) = 4
  - 53 goes in the second hash table and it sends 50 in the first to position h1(50) = 6
  - 50 goes in the first hash table and it sends 39 in the second to position h2(39) = 3

Cuckoo hashing - example VIII

Position	0	1	2	3	4	5	6	7	8	9	10
T		100		36			50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T	3	20		39	53		67			105	

Cuckoo hashing

- It can happen that we cannot insert a key because we get in a cycle. In these situation we have to increase the size of the tables and rehash the elements.
- While in some situation insert moves a lot of elements, it can be shown that if the load factor of the tables is below 0.5, the probability of a cycles is low and it is very unlikely that more than $O(log_2 n)$ elements will be moved.

Cuckoo hashing

- If we use two tables and each position from a table holds one element at most, the tables have to have load factor below 0.5 to work well.
- If we use three tables, the tables can have load factor of 0.91 and for 4 tables we have 0.97