DATA STRUCTURES AND ALGORITHMS

LECTURE 10

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2019 - 2020

In Lecture 9...

- Hash tables
    - Separate chaining
    - Coalesced chaining
    - Open addressing

Today

- Open addressing
- Perfect hashing
- Cuckoo hashing
- Linked Hash Table

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.

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

- 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.
- We would also like to be able to generate all the possible $m!$ permutations as probe sequences

Open addressing - Linear probing

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

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

- 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 - 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 - 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 probing, 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 $\{0$, m-1$\}$ set (actually $\{1$, m-1$\}$ set, because $h''(k)$ should never return 0).

Open addressing - Double hashing

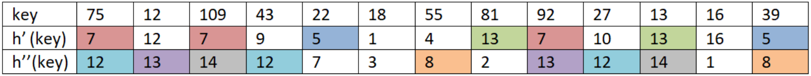
- Choose m as a prime number and design h'' in such a way that it always return a value from the {0, m-1} set.
- For example:
  $h'(k) = k\%m$
  $h''(k) = 1 + (k\%(m-1))$.
- For $m = 11$ and $k = 36$ we have:
  $h'(36) = 3$
  $h''(36) = 7$
- The probe sequence is: $< 3, 10, 6, 2, 9, 5, 1, 8, 4, 0, 7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

Open addressing - Double hashing - example

- Consider a hash table of size $m = 17$ that uses open addressing with double hashing for collision resolution, with $h'(k) = k\%m$ and $h''(k) = (1 + (k\%16))$.

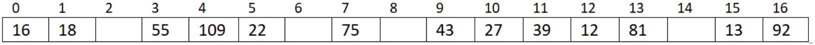
- Insert into the table the following elements: 75, 12, 109, 43, 22, 18, 55, 81, 92, 27, 13, 16, 39.

- Values of the two hash functions for each element:



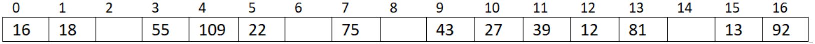
key	75	12	109	43	22	18	55	81	92	27	13	16	39
h' (key)	7	12	7	9	5	1	4	13	7	10	13	16	5
h''(key)	12	13	14	12	7	3	8	2	13	12	14	1	8

Open addressing - Double hashing - example



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	18		55	109	22		75		43	27	39	12	81		13	92

Open addressing - Double hashing - example



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	18		55	109	22		75		43	27	39	12	81		13	92

- 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:
  - 75: $< 7, 2, 14, 9, 4, 16, 11, 6, 1, 13, 8, 3, 15, 10, 5, 0, 12 >$
  - 109: $< 7, 4, 1, 15, 12, 9, 6, 3, 0, 14, 11, 8, 5, 2, 16, 13, 10 >$
- 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 some of the basic dictionary operations for collision resolution with open addressing.

- 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?

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?

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] $\neq$ -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 - other operations

- What should the *search* operation do?

Open addressing - other operations

- What should the *search* operation do?
- How can we *remove* an element from the hash table?

Open addressing - other operations

- What should the *search* operation do?
- 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

Open addressing - other operations

- What should the *search* operation do?
- 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 $\Theta(n)$

Perfect hashing

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 expected to be at most $2N$ (if it is larger, just pick a different hash function).

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 values 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39
- Since we are inserting $N = 13$ elements, we will take $m = 13$.

Perfect hashing - example

- p has to be a prime number larger than the maximum key ⇒ 151

- 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).

Value	76	12	109	43	22	18	55	81	91	27	13	16	39
H(Value)	1	12	1	1	3	4	3	3	7	5	2	11	2

Perfect hashing - example

- Collisions:
    - position 1 - 76, 109, 43
    - position 2 - 13, 39
    - position 3 - 22, 55, 81
    - position 4 - 18
    - position 5 - 27
    - position 7 - 91
    - position 11 - 16
    - position 12 - 12
- Sum of the sizes of the secondary hash tables: $9 + 4 + 9 + 1 + 1 + 1 + 1 + 1 = 27$

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 2, we can define $a = 4$ and $b = 11$ and we will have:
  $h(13) = 3$
  $h(39) = 0$

Perfect hashing - example

- Assume that for the secondary hash table from position 1 we will choose $a = 14$ and $b = 1$.

- Positions for the elements will be:
  $h(76) = ((14 * 76 + 1)\%151)\%9 = 8$
  $h(109) = h(24) = ((14 * 109 + 1)\%151)\%9 = 8$
  $h(43) = h(24) = ((14 * 43 + 1)\%151)\%9 = 6$

- 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 $\mathrm{h}(76) = 5$, $\mathrm{h}(109) = 8$, $\mathrm{h}(43) = 0$.

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 the 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.

Dynamic Perfect Hashing

- Traditionally, perfect hashing is said to work in a static environment (you need to know all the keys in advance).
- It is easy to see why: you can build a table to be collision free, by picking new hash functions. But if you allow new additions you might get a collision after you have built the table.
- However, dynamic perfect hashing was also introduced in 1994.
- It obviously implies a lot of rebuilding when a new element is added (the *small* hash table is rebuilt more often, but the first-level hash table is also rebuilt after any M operations)

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 element.

- 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.

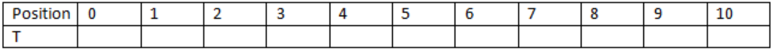
- The element that was kicked off, will be placed at its position in the second hash table. If that position is occupied, we will kick off the element from there and place it into its position in the first hash table.

- We repeat the above process until we will get an empty position for an element.

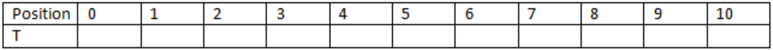
- If we get back to the same location with the same key we have a cycle and we cannot add this element $\Rightarrow$ resize, rehash

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



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

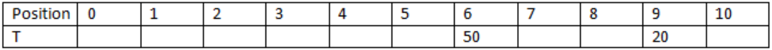
- Insert key 20

Cuckoo hashing - example

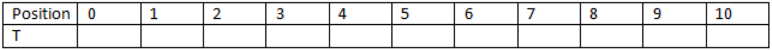
- Insert key 20
    - h1(20) $= 9$ - empty position, element added in the first table
- Insert key 50

Cuckoo hashing - example

- 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	



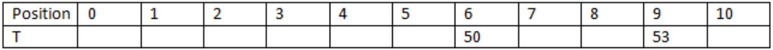
Position	0	1	2	3	4	5	6	7	8	9	10
T											

Cuckoo hashing - example

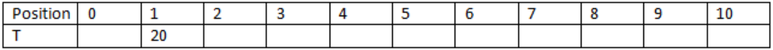
- Insert key 53

Cuckoo hashing - example

- 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	



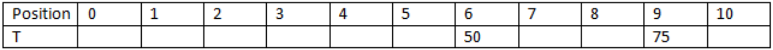
Position	0	1	2	3	4	5	6	7	8	9	10
T		20									

Cuckoo hashing - example

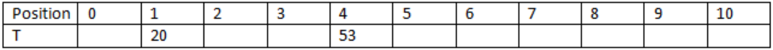
- Insert key 75

Cuckoo hashing - example

- 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

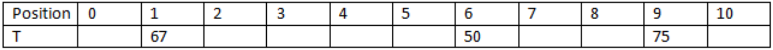
- Insert key 100

Cuckoo hashing example

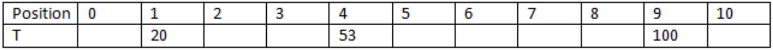
- Insert key 100
    - h1(100) = 1 - empty position
- Insert key 67

Cuckoo hashing example

- 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	



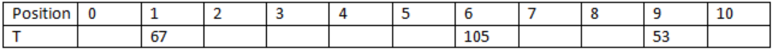
Position	0	1	2	3	4	5	6	7	8	9	10
T		20			53					100	

Cuckoo hashing example

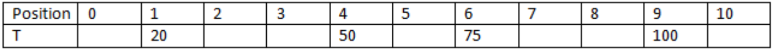
- Insert key 105

Cuckoo hashing example

- 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

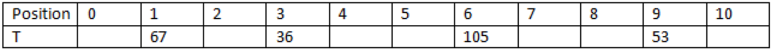
- Insert key 3

Cuckoo hashing example

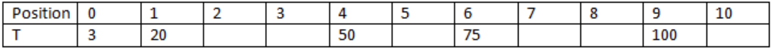
- Insert key 3
    - h1(3) = 3 - empty position
- Insert key 36

Cuckoo hashing example

- 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	

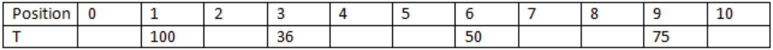
Cuckoo hashing example

- Insert key 39

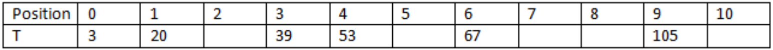
Cuckoo hashing example

- 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



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

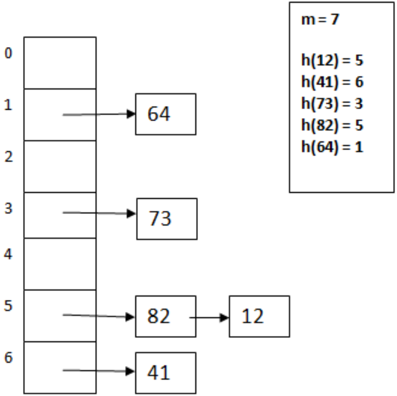
- If we use two tables and each position from a table holds one element at most, the tables have to have load factor below 0.5 to work well.
- If we use three tables, the tables can have load factor of 0.91 and for 4 tables we have 0.97

Linked Hash Table

Linked Hash Table

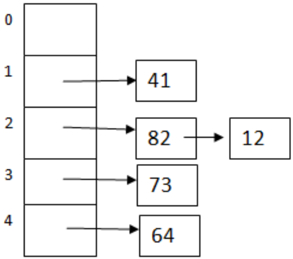
- Assume we build a hash table using separate chaining as a collision resolution method.
- We have discussed how an iterator can be defined for such a hash table.
- When iterating through the elements of a hash table, the order in which the elements are visited is *undefined*
- For example:
    - Assume an initially empty hash table (we do not know its implementation)
    - Insert one-by-one the following elements: 12, 41, 73, 82, 64
    - Use an iterator to display the content of the hash table
    - In what order will the elements be displayed?

Linked Hash Table



- Iteration order: 64, 73, 82, 12, 41

Linked Hash Table



m = 5

h(12) = 2
h(41) = 1
h(73) = 3
h(82) = 2
h(64) = 4

- Iteration order: 41, 82, 12, 73, 64

Linked Hash Table

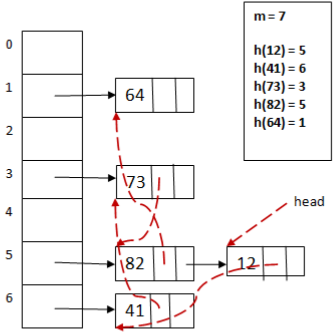
- A *linked hash table* is a data structure which has a *predictable* iteration order. This order is the order in which elements were inserted.
- So if we insert the elements 12, 41, 73, 82, 64 (in this order) in a linked hash table and iterate over the hash table, the iteration order is guaranteed to be: 12, 41, 73, 82, 64.
- How could we implement a linked hash table which provides this iteration order?

Linked Hash Table

- A linked hash table is a combination of a hash table and a linked list. Besides being stored in the hash table, each element is part of a linked list, in which the elements are added in the order in which they are inserted in the table.

- Since it is still a hash table, we want to have, on average, $\Theta(1)$ for insert, remove and search, these are done in the same way as before, the *extra* linked list is used only for iteration.

Linked Hash Table



- Red arrows show how the elements are linked in insertion order, starting from a *head* - the first element that was inserted, 12.

Linked Hash Table

- Do we need a doubly linked list for the order of elements or is a singly linked list sufficient? (think about the operations that we usually have for a hash table).

Linked Hash Table

- Do we need a doubly linked list for the order of elements or is a singly linked list sufficient? (think about the operations that we usually have for a hash table).

- The only operation that cannot be efficiently implemented if we have a singly linked list is the *remove* operation. When we remove an element from a singly linked list we need the element before it, but finding this in our linked hash table takes $O(n)$ time.

Linked Hash Table - Implementation

- What structures do we need to implement a Linked Hash Table?

```
Node:
    info: TKey
    nextH: ↑ Node //pointer to next node from the collision
    nextL: ↑ Node //pointer to next node from the insertion-order list
    prevL: ↑ Node //pointer to prev node from the insertion-order list
```

```
LinkedHT:
    m:Integer
    T:(↑ Node)[]
    h:TFunction
    head: ↑ Node
    tail: ↑ Node
```