DATA STRUCTURES AND ALGORITHMS
LECTURE 10

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2024 - 2025

In Lecture 9...

- Direct address table
- Hash table
    - Hash function
    - Collision resolution through separate chaining

Today

- Collision resolution through separate chaining
- Collision resolution through coalesced chaining
- Collision resolution through open addressing

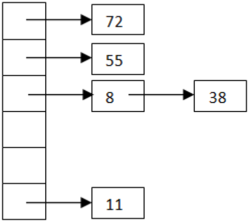
Example

- Assume we have a hash table with $m = 6$ that uses separate chaining for collision resolution, with the following policy: if the load factor of the table after an insertion is greater than or equal to 0.7, we double the size of the table

- Using the division method, insert the following elements, in the given order, in the hash table: 38, 11, 8, 72, 57, 29, 2.

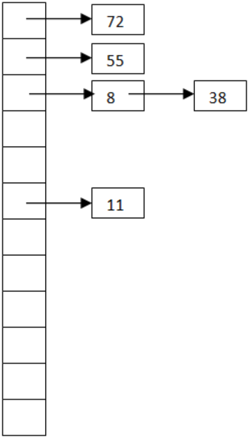
Example

- $h(38) = 2$ (load factor will be $1/6$)
- $h(11) = 5$ (load factor will be $2/6$)
- $h(8) = 2$ (load factor will be $3/6$)
- $h(72) = 0$ (load factor will be $4/6$)
- $h(55) = 1$ (load factor will be $5/6$ - greater than 0.7)
- The table after the first five elements were added:



Example

- Is it OK if after the resize this is our hash table?

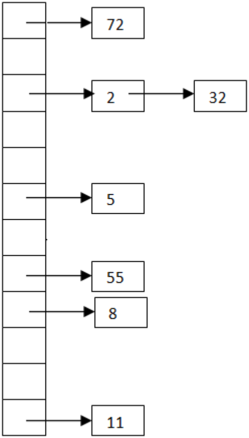


Example

- The result of the hash function (i.e. the position where an element is added) depends on the size of the hash table. If the size of the hash table changes, the value of the hash function changes as well, which means that search and remove operations might not find the element.
- After a resize operation, we have to add all elements again in the hash table, to make sure that they are at the correct position $\rightarrow$ rehash

Example

- After rehash and adding the other two elements:



- What do you think, which containers cannot be represented on a hash table?

Iterator

- What do you think, which containers cannot be represented on a hash table?
- How can we define an iterator for a hash table with separate chaining?

Iterator

- What do you think, which containers cannot be represented on a hash table?
- How can we define an iterator for a hash table with separate chaining?
- Since hash tables are used to implement containers where the order of the elements is not important, our iterator can iterate through them in any order.
- For the hash table from the previous example, the easiest order in which the elements can be iterated is: 72, 2, 32, 5, 55, 8, 11

Iterator

- Iterator for a hash table with separate chaining is a combination of an iterator on an array (table) and on a linked list.
- We need a current position to know the position from the table that we are at, but we also need a current node to know the exact node from the linked list from that position.

<u>IteratorHT:</u>
  ht: HashTable
  currentPos: Integer
  currentNode: $\uparrow$ Node

Iterator - init

- How can we implement the *init* operation?

Iterator - init

- How can we implement the *init* operation?

```
subalgorithm init(ith, ht) is:
//pre: ith is an IteratorHT, ht is a HashTable
   ith.ht ← ht
   ith.currentPos ← 0
   while ith.currentPos < ht.m and ht.T[ith.currentPos] = NIL execute
      ith.currentPos ← ith.currentPos + 1
   end-while
   if ith.currentPos < ht.m then
      ith.currentNode ← ht.T[ith.currentPos]
   else
      ith.currentNode ← NIL
   end-if
end-subalgorithm
```

- Complexity of the algorithm:

Iterator - init

- How can we implement the *init* operation?

subalgorithm init(ith, ht) **is**:
//pre: ith is an IteratorHT, ht is a HashTable
    ith.ht ← ht
    ith.currentPos ← 0
    while ith.currentPos < ht.m **and** ht.T[ith.currentPos] = NIL **execute**
        ith.currentPos ← ith.currentPos + 1
    end-while
    if ith.currentPos < ht.m **then**
        ith.currentNode ← ht.T[ith.currentPos]
    else
        ith.currentNode ← NIL
    end-if
end-subalgorithm

- Complexity of the algorithm: $O(m)$

Iterator - other operations

- How can we implement the *getCurrent* operation?

Iterator - other operations

- How can we implement the *getCurrent* operation?
- How can we implement the *next* operation?

Iterator - other operations

- How can we implement the *getCurrent* operation?
- How can we implement the *next* operation?
- How can we implement the *valid* operation?

- How can we define a sorted container on a hash table with separate chaining?

Sorted containers

- How can we define a sorted container on a hash table with separate chaining?
  - Hash tables are in general not very suitable for sorted containers.
  - However, if we have to implement a sorted container on a hash table with separate chaining, we can store the individual lists in a sorted order and for the iterator we can return them in a sorted order.

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 $m = 13$ that uses coalesced chaining for collision resolution and a hash function with the division method
- Insert into the table the following elements: 5, 18, 16, 15, 13, 31, 26.
- Let's compute the value of the hash function for every key:

Key	5	18	16	15	13	31	26
Hash	5	5	3	2	0	5	0

Example

- Initially the hash table is empty. All next values are -1 and the first empty position is position 0.
- 5 will be added to position 5. But 18 should also be added there. Since that position is already occupied, we add 18 to position firstEmpty and set the next of 5 to point to position 0. Then we reset firstEmpty to the next empty position.
- We keep doing this, until we add all elements.

Example

- The final table:

pos	0	1	2	3	4	5	6	7	8	9	10	11	12
T	18	13	15	16	31	5	26						
next	1	4	-1	-1	6	0	-1	-1	-1	-1	-1	-1	-1

- $firstEmpty = 7$

Coalesced chaining - representation

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

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

<u>HashTable:</u>
  T: TKey[]
  next: Integer[]
  m: Integer
  firstEmpty: Integer
  h: TFunction

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

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

Coalesced chaining - insert

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

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

subalgorithm changeFirstEmpty(ht) **is:**
//pre: ht is a HashTable
//post: the value of ht.firstEmpty is set to the next free position
  ht.firstEmpty ← ht.firstEmpty + 1
  while ht.firstEmpty < ht.m **and** ht.T[ht.firstEmpty] ≠ -1
execute
    ht.firstEmpty ← ht.firstEmpty + 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?

- How would you search for an element in a hash table with coalesced chaining?

Coalesced chaining - search

- How would you search for an element in a hash table with coalesced chaining?
- Even if it is an array, we are not going to search as in an array (i.e., start from position 0 and go until you find the element)
- We compute the value of the hash function and check the linked list which starts from that position. If the element is in the table, it should be in this list.

- Remove is a tricky operation for coalesced chaining and at first it might not even be clear what situations make it complicated. So let's take 5 simple examples where we will add a few elements in a hash table with coalesced chaining with $m = 5$ and then we will remove element 11. For every example we will only focus on how to do the removal so that the result is correct for that particular hash table.
- *firstEmpty* is not going to be marked on the following examples, simply assume that it is the first empty position from left to right.

Coalesced chaining - simple examples of remove II

- Example 1: insert 11, 8, 3

elems	3	11		8	
next	-1	-1	-1	0	-1

- Remove 11

elems	3			8	
next	-1	-1	-1	0	-1

- In this case we just mark the position as empty.

- Example 2: insert 56, 8, 11, 12

elems	11	56	12	8	
next	-1	0	-1	-1	-1

- Remove 11

elems		56	12	8	
next	-1	-1	-1	-1	-1

- In this case we remove 11 in the same way as we remove an element from the end of a linked list on array.

- Example 3: insert 11, 20, 56

elems	20	11	56		
next	-1	2	-1	-1	-1

- Remove 11

elems	20	56			
next	-1	-1	-1	-1	-1

- Now we need to remove the first element of the linked list. But position 1 cannot be empty (because a search for 56 would start from position 1), so we move 56 to replace 11.

- Example 4: insert 56, 11, 12, 1

elems	11	56	12	1	
next	3	0	-1	-1	-1

- Remove 11

elems		56	12	1	
next	-1	3	-1	-1	-1

- We remove 11 in the same way in which we remove an element from the middle of a linked list on array.

- Example 5: insert 56, 11, 20, 13

elems	11	56	20	13	
next	2	0	-1	-1	-1

- Remove 11

elems	20	56		13	
next	-1	0	-1	-1	-1

- Position 0 cannot become empty, since then 20 would not be found, so we move 20 to replace 11.

- Let's see a few more complicated example (on the one used previously).

- A hash table with coalesced chaining is essentially an array, in which we have multiple singly linked lists. Can we remove an element like we remove from a regular singly linked list? Just set the next of the previous element to *jump over* it?
- For example, if from the previously built hash table I want to remove element 18, can we just do it like that?

pos	0	1	2	3	4	5	6	7	8	9	10	11	12
T		13	15	16	31	5	26						
next	-1	4	-1	-1	6	1	-1	-1	-1	-1	-1	-1	-1

- $firstEmpty = 0$

- If we remove 18 simply by setting the next of 5 to be 13, we will never be able to find 13 and 26, because a search for them is going to start from position 0, and that position being empty, we will never check any other position.
- **Obs 1**: Some positions from the linked list of elements are not allowed to become empty (specifically, the ones which are equal to the value of the hash function of any element from the linked list).

- Would then be a solution to move every element to the previous position in the linked list?

- For example, if we remove 18, we would have:

pos	0	1	2	3	4	5	6	7	8	9	10	11	12
T	13	31	15	16	26	5							
next	1	4	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	-1

- $firstEmpty = 6$

- Would then be a solution to move every element to the previous position in the linked list?

- For example, if we remove 18, we would have:

pos	0	1	2	3	4	5	6	7	8	9	10	11	12
T	13	31	15	16	26	5							
next	1	4	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	-1

- $firstEmpty = 6$

- For this example, it would work. This hash table is now correct and every element can be found in it. But what if now we remove 5? Is the hash table below correct?

pos	0	1	2	3	4	5	6	7	8	9	10	11	12
T	31	26	15	16		13							
next	1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	-1

- $firstEmpty = 4$

- Now element 13 is not going to be found, because a search for 13 starts from position 0, but 13 is currently on a position *before* 0 in the linked list.
- **Obs 2:** Not any element can get to any position in the linked list (specifically, no element is allowed to be on a position which is *before* the position to which it hashes)

- Considering the cases discussed previously, we can describe how remove should look like:
  - Compute the value of the hash function for the element, let's call it p.
  - Starting from p follow the links in the hash table to find the element.
  - If element is not found, we want to remove something which is not there, so nothing to do. Assume we do find it, on position *elem_pos*.
  - Starting from position *elem_pos* search for another element in the linked list, which should be on that position. If you find one, let's say on position *other_pos*, move the element from *other_pos* to *elem_pos* and restart the remove process for *other_pos*.
  - If no element is found which hashes to *elem_pos*, you can simply remove the element, like in case of a singly linked list, setting its previous to point to its next.

Coalesced chaining - remove

```
subalgorithm remove(ht, elem) is:
   pos ← ht.h(elem)
   prevpos ← -1 //find the element to be removed and its previous
   while pos ≠ -1 and ht.t[pos] ≠ elem execute:
      prevpos ← pos
      pos ← ht.next[pos]
   end-while
   if pos = -1 then
      @element does not exist
   else
      over ← false //becomes true when nothing hashes to pos
      repeat
         p ← ht.next[pos]
         pp ← pos //previous of p
         while p ≠ -1 and ht.h(ht.t[p]) ≠ pos execute
            pp ← p
            p ← ht.next[p]
         end-while
//continued on the next slide
```

```
        if p = -1 then
            over ← true //no element hashes to pos
        else
            ht.t[pos] ← ht.t[p] //move element from position p to pos
            prevpos ← pp
            pos ← p
        end-if
    until over
//now element from pos can be removed (no element hashes to it)
    if prevpos = -1 then //see next slide for explanation
        idx ← 0
        while (idx < ht.m and prevpos = -1) execute
            if ht.next[idx] = pos then
                prevpos ← idx
            else
                idx ← idx + 1
            end-if
        end-while
    end-if
//continued on the next slide...
```

```
    if prevpos ≠ -1 then
        ht.next[prevpos] ← ht.next[pos]
    end-if
    ht.t[pos] ← −1
    ht.next[pos] ← -1
    if ht.firstFree > pos then
        ht.firstFree ← pos
    end-if
  end-if
end-subalgorithm
```

- Complexity:

```
        if prevpos ≠ -1 then
            ht.next[prevpos] ← ht.next[pos]
        end-if
        ht.t[pos] ← −1
        ht.next[pos] ← -1
        if ht.firstFree > pos then
            ht.firstFree ← pos
        end-if
    end-if
end-subalgorithm
```

- Complexity: O(m), but Θ(1) on average

- What happens when the element you need to remove is right on the position where it should be? In this case we might assume that the element has no previous (and in the above implementation its *prev* will be -1), but this is not true. It might happen that an element is on its position, but it still has a previous element. For example, element 13:

pos	0	1	2	3	4	5	6	7	8	9	10	11	12
T	13	31	15	16	26	5							
next	1	4	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	-1

- $firstEmpty = 6$

- If we wanted to remove 13, it would be ok, because 26 would be moved in its place, but if no other element hashed to position 0 and we just made its next -1, element 31 would never be found.

- This is why we have the while loop in the remove code when *prevpos* is -1: we go through the table and see if there is an element whose next is *pos*, because this element would then be the previous of *pos*.
- This *while* loop happens rarely, only when an element is found on the position where it hashes and no other element hashes to its position. Nevertheless, having a while loop which goes through all the elements of the table is not a very *hash table-like* operation and it increases the complexity of the function.

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

- How can we implement a sorted container on a hash table with coalesced chaining? How can we implement its iterator?

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 positions, 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 the hash table positions $\{0, ..., m-1\}$, so that eventually every slot is considered.
- We would also like to have a hash function which can generate all the $m!$ possible permutations (spoiler alert: we cannot)

- 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 $m = 16$ that uses open addressing and linear probing for collision resolution
- Insert into the table the following elements: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.
- Let's compute the value of the hash function for every key for i = 0:

Key	76	12	109	43	22	18	55	81	91	27	13	16	39
Hash	12	12	13	11	6	2	7	1	11	11	13	0	7

Open addressing - Linear probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
27	81	18	13	16		22	55	39			43	76	12	109	91

Open addressing - Linear probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
27	81	18	13	16		22	55	39			43	76	12	109	91

- 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
- Advantages of linear probing:
  - Probe sequence is always a permutation
  - Can benefit from caching

Open addressing - Linear probing - primary clustering

- Why is primary clustering a problem?
- Assume m positions, n elements and $\alpha = 0.5$ (so $n = m/2$)
- Best case arrangement: every second position is empty (for example: even positions are occupied and odd ones are free)
- What is the average number probes (positions verified) that need to be checked to insert a new element?

Open addressing - Linear probing - primary clustering

- Why is primary clustering a problem?
- Assume m positions, n elements and $\alpha = 0.5$ (so $n = m/2$)
- Best case arrangement: every second position is empty (for example: even positions are occupied and odd ones are free)
- What is the average number probes (positions verified) that need to be checked to insert a new element?
- Worst case arrangement: all n elements are one after the other (assume in the second half of the array)
- What is the average number of probes (positions verified) that need to be checked to insert a new element?

- 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: $< k, k + 1, k + 4, k + 9, k + 16, ... >$

- One important issue with quadratic probing is how we can 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 $k = 3$:
  - $h(3,0) = (3 \% 8 + 0.5 * 0 + 0.5 * 0^2) \% 8 = 3$
  - $h(3,1) = (3 \% 8 + 0.5 * 1 + 0.5 * 1^2) \% 8 = 4$
  - $h(3,2) = (3 \% 8 + 0.5 * 2 + 0.5 * 2^2) \% 8 = 6$
  - $h(3,3) = (3 \% 8 + 0.5 * 3 + 0.5 * 3^2) \% 8 = 1$
  - $h(3,4) = (3 \% 8 + 0.5 * 4 + 0.5 * 4^2) \% 8 = 5$
  - $h(3,5) = (3 \% 8 + 0.5 * 5 + 0.5 * 5^2) \% 8 = 2$
  - $h(3,6) = (3 \% 8 + 0.5 * 6 + 0.5 * 6^2) \% 8 = 0$
  - $h(3,7) = (3 \% 8 + 0.5 * 7 + 0.5 * 7^2) \% 8 = 7$

- 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 $k = 3$:
  - $h(3, 0) = (3 \% 7 + 0^2) \% 7 = 3$
  - $h(3, 1) = (3 \% 7 - 1^2) \% 7 = 2$
  - $h(3, 2) = (3 \% 7 + 2^2) \% 7 = 0$
  - $h(3, 3) = (3 \% 7 - 3^2) \% 7 = 1$
  - $h(3, 4) = (3 \% 7 + 4^2) \% 7 = 5$
  - $h(3, 5) = (3 \% 7 - 5^2) \% 7 = 6$
  - $h(3, 6) = (3 \% 7 + 6^2) \% 7 = 4$

- 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 following elements: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.

Open addressing - Quadratic probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	81	18	16		91	22	55	39		27	43	76	12	109	

Open addressing - Quadratic probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	81	18	16		91	22	55	39		27	43	76	12	109	

- 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 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 $\{0, \text{m-1}\}$ set (actually $\{1, \text{m-1}\}$ 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.

- 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?

- 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] ≠ -1 execute
    //-1 means empty space
        i ← i + 1
        pos ← ht.h(e, i)
    end-while
    if i = ht.m then
        @resize and rehash and compute the position for e again
    else
        ht.T[pos] ← e
   end-if
end-subalgorithm
```

- 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 $\Theta(n)$

Summary

- Today we have talked about:
  - Coalesced chaining
  - Open addressing