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 $m = 16$ that uses coalesced chaining for collision resolution and a hash function with the division method
- 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:

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

Example

- Initially the hash table is empty. All next values are -1 and the first empty position is position 0.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

firstEmpty = 0

- 76 will be added to position 12. But 12 should also be added there. Since that position is already occupied, we add 12 to position firstEmpty and set the next of 76 to point to position 0. Then we reset firstEmpty to the next empty position

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12												76			
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1

firstEmpty = 1

Example

- And we continue in the same manner. We have no collisions up to 81, but we need to reset firstEmpty when we *accidentally* occupy it.

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

firstEmpty = 3

- When adding 91, we put it to position firstEmpty and set the next link of position 11 to position 3.

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

firstEmpty = 4

- The final table:

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

firstEmpty = 10

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?

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

Coalesced chaining - ChangeFirstEmpty

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?

Coalesced chaining - search

- 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.

Coalesced chaining - remove

- 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.

```
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 $\Theta(1)$ 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.

Coalesced chaining - iterator

- 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?