# DS – Seminar 5
# – Hash Tables

1. **Set – reprezentation on a hash table – collision resolution with coalesced chaining**

- Assume:
  - The elements are integer numbers

**For ex:**

- Set with: 5, 18, 16, 15, 13, 31, 26
- HT:
  - m = 13
  - Hash function defined with the division method

| **Elem** | 5 | 18 | 16 | 15 | 13 | 31 | 26 |
|---|---|---|---|---|---|---|---|
| **h(elem)** | 5 | 5 | 3 | 2 | 0 | 5 | 0 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **t** | 18 | 13 | 15 | 16 | 31 | 5 | 26 | | | | | | |
| **next** | ~~-1~~ 1 | ~~-1~~ 4 | -1 | -1 | ~~-1~~ 6 | ~~-1~~ 0 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

firstFree = ~~0 1 4 6~~ 7

- firstFree is considered to be the first empty position from left to right (empty positions are no longer linked)
    - Why did we link the empty positions for a linked list on array? – to get an empty position in Θ(1)
    - Why are we not linking the empty positions here? – because sometimes we do not need just any empty position, we need to occupy a specific one (when the position given by the hash function is empty) and removing an empty position from the middle of the list is just as bad as not having a linked list and search for en empty position in O(m)
    - What would be a solution? – make it doubly linked
        - Would that make add Θ(1) in worst case? – No, you still need to find the end of list that starts from the position given by the hash function
        - But making the list doubly linked would obviously use extra memory (and we already have more positions in the array than actual elements and have an array for next as well).
- One „linked list" can contain elements belonging to different collisions: for ex. the list starting at position 5: 5 (5) – 18 (5) – 13 (0) – 31 (5) – 26 (0)
    - In separate chaining you knew that all elements in a singly linked list had the same value for the hash function

**Reprezentation:**

Set:

m: Integer
t: TElem[]
next:Integer[]
firstFree: Integer
h: Tfunction

```
subalgorithm init (set):
      @ initialize the hash function
      @ initialize the value of m
      for i ← 0, set.m-1 execute
            set.t[i] ← -1 //assume -1 means empty position
            set.next[i] ← -1
      end-for
      set.firstFree ← 0
end-subalgorithm
Complexity: Θ(m)

Function search(set, elem):
      i ← set.h(elem)
      while (i ≠ -1 and set.t[i] ≠ elem) execute
            i ← set.next[i]
      end-while
      if i = -1 then
            search ← false
      else
            search ← true
end-function
Complexity: Θ (n) in worst case, but on average Θ(1)
```

**Remove – that's the tricky operation**

Let's see some simple examples for remove (before discussing the complicated ones):

Hash function is the one by division

Example 1:

m = 5, insert (in this order) 11, 8, 3

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 3 | 11 | | 8 | |
| -1 | -1 | -1 | -1 | -1 |

Remove 11

- We can just simply remove 11 (make the position empty)

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 3 | | | 8 | |
| -1 | -1 | -1 | -1 | -1 |

Example 2:

m = 5, insert (in this order) 56, 8, 11, 12

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 11 | 56 | 12 | 8 | |
| -1 | 0 | -1 | -1 | -1 |

Remove 11

- Here we have a linked list (starting from position 1) made of two elements: 56 – 11. Removing the last element is similar to removing the last element from any linked list: we make the next of the previous element to be -1 and empty this position

| ~~0~~ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| | 56 | 12 | 8 | |
| -1 | ~~0~~ -1 | -1 | -1 | -1 |

Example 3:

m= 5, insert (in this order) 11, 20, 56

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 20 | 11 | 56 | | |
| -1 | 2 | -1 | -1 | -1 |

Remove 11

- This time we want to remove the first element of a two-element list. Normally in linked lists we do this by simply saying that the list starts from now on with the second element. But if we empty position 1 (to remove 11), we will not be able to find 56 anymore (search for 56 starts from position 1). So position 1 is not allowed to become an empty position, this is why we need to move 56 to position 1.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 20 | ~~11~~ 56 | | | |
| -1 | ~~2~~ -1 | -1 | -1 | -1 |

Example 4

m= 5, insert (in this order) 56, 11, 12, 1

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 11 | 56 | 12 | 1 | |
| 3 | 0 | -1 | -1 | |

Remove 11

- 11 is in the middle of a linked list with 3 elements (56 – 11 – 1). We can remove it as we remove any element from the middle of a linked list (set the next of the previous element to the next of this element)

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| | 56 | 12 | 1 | |
| ~~3~~ -1 | ~~0~~ 3 | -1 | -1 | |

Example 5

m= 5, insert (in this order) 56, 11, 20, 13

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 11 | 56 | 20 | 13 | |
| 2 | 0 | -1 | -1 | |

Remove 11

- Although it is similar to the previous case, we cannot just set the next of position 1 to position 2 and put a -1 to position 0, because then 20 will never be found (search for 20 starts from position 0). So we need to move 20 to position 0 and make position 2 an empty position

- We could set the next of 56 to -1 as well (no other elements that hash to 1 can be found, but in general checking this is not so easy).

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| ~~11~~ 20 | 56 | | 13 | |
| ~~2~~ -1 | 0 | -1 | -1 | |

Let's see now how a more complex case, on our first example:

Remove: remove key 5 from the initial example

- **Problem:** we might lose links to other elements
- Cannot just do a remove like in case of a linked list on array, because not every element can be at any position in the table. No element can be "before" (considering the links) the position to which it hashes. For example, we cannot move 26 to replace 5 (because 26 hashes to 0, and a search starting from position 0 does not go through position 5). Secondly, not any position can become empty (we cannot just remove 5 by making that position empty, because 18 and 31 will never be found).

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **T** | ~~18~~ 13 | 13 | 15 | 16 | 31 | ~~5~~ 18 | 26 | | | | | | |
| **Next** | ~~1~~ 4 | ~~4~~ -1 | -1 | -1 | 6 | 0 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

firstFree: 7

Steps:

1. Compute the value of the hash function => h(5) = 5
2. Start searching for element 5, starting from the position given by the hash function and following the next links. We find 5 right on position 5.
3. We cannot just set t[5] = -1 and next[5] = -1 because we lose the link to 18 (and a search for 18 or 31 will not find these elements)
4. Search for elements (following the links) that hash to the position from which I am removing an elements (position 5 in our example)
    a. If no element is found, we remove the element as we remove an element from a singly linked list on array.
    b. If an element is found, it is moved to the position where we delete from, and the process of removal is repeated with the position from which we moved the element.

- Remove element 5, which is at position 5
- Search for the first element that hashes to position 5 => 18
- Move 18 to position 5
- Now we want to remove element 18, which is at position 0
- Search for the first element that hashes to position 0 => 13
- Move 13 on position 0
- Now we want to remove element 13, which is at position 1
- Search for the first element that hashes to position 1 => no such element
- Remove element 13, modifying the links

```
subalgorithm remove(set, elem) is
    pos ← set.h(elem)
    prevpos ← -1 //previous of pos, when we want to remove an element from position pos, we need its previous element, since it is a singly linked list
    //find the element to be removed. Set its previous as well
    while pos ≠ -1 and set.t[pos] ≠ elem execute
        prevpos ← pos
        pos ← set.next[pos]
    end-while
    if pos = -1 then
        @element does not exist
    else
        //find another element that hashes to pos. We might have to execute this part of finding another element for a position several times
        over ← false //becomes true when nothing hashes to pos
        repeat
            p ← set.next[pos] //first position to be checked
            pp ← pos //previous of p
            while p ≠ -1 and set.h(set.t[p]) ≠ pos execute
                pp ← p
                p ← set.next[p]
            end-while
            if p = -1 then
                over ← true
            else
                set.t[pos] ← set.t[p] //move element from position p to position pos
                prevpos ← pp
                pos ← p
            end-if
        until over
        //now element from position pos can be removed (no other element hashes to it).
        //if prevpos = -1, it means that prevpos was never changed (initially it was -1) so the first while loop was not executed because elem was right on the position where it hashes and in the repeat loop no other element was moved on the initial position returned by the hashfunction. So it looks like pos is the first element of the linked list and it has no previous. But this is not true. Just because an element is on the position where it hashes, it might still have a previous. Look at our example: after removing 5, element 13 will be on the position where it hashes (position 0), but it has a previous element (position 5). So it prevpos is -1 at this point, we might still not be 100% sure that pos has no previous, so we need to go through the hashtable and check if there is any element, whose next is pos. If we find such an element, that is going to be prevpos.
        if prevpos = -1 then
            //parse the table to check if pos has any previous element
             idx ← 0
             while (idx < set.m and prevpos = -1) execute
                  if set.next[idx] = pos then
                        prevpos ← idx
                  else
                        idx ← idx + 1
                  end-if
             end-while
```

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

2. **Iterator for a SortedMap represented on a hash table, collision resolution with separate chaining.**

**Note 1:** Why would we implement a SortedMap on a hash table?

- Normally hash tables are not very suitable for sorted containers, but we might have a situation when we need a SortedMap in which search operations are very frequent and we want to get a good average complexity for it (even if some other operations will have a slightly worse complexity).

**Note 2:** In a hash table we use a hash function to provide a position for an element. In a SortedMap, elements are key-value pairs. What do we compute the hash function for: just the key, just the value or some combination of the key and value?

- If we look at the most important operations of the SortedMap, we can observe that only add receives as parameter both the key and the value, search and remove receive as parameter only the key. This means that the hash function has to be created only for the key, since this is the information that is provided to all functions.

Let's take an example. In this example we will assume:

- We memorize only the keys from the Map (values are not used for the hash function, anyway)
- Keys are integer numbers

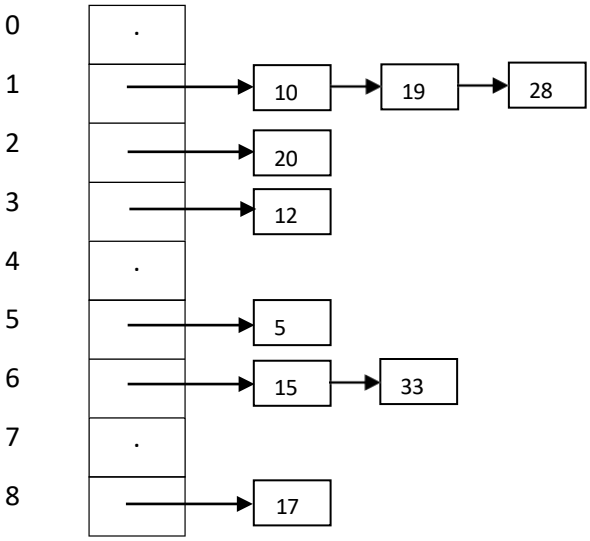
- Keys from the SortedMap: 5, 28, 19, 15, 20, 33, 12, 17, 10 – Keys have to be unique!
- Hash Table
    - m = 9
        - Hash function defined with the division method
            - h(k) = k mod m

| **k** | 5 | 28 | 19 | 15 | 20 | 33 | 12 | 17 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| **h(k)** | 5 | 1 | 1 | 6 | 2 | 6 | 3 | 8 | 1 |

- h(k) can contain duplicates – they are called collisions

**Sorted Map after adding all elements**

We cannot make the table completely sorted, but at least we can make the individual lists sorted.




Iterator:

- If we iterate through the elements using the iterator, they should be visited in the following order: 5, 10, 12, 15, 17, 19, 20, 28, 33
- If we use the iterator -> complexity of the whole iteration to be Θ(n) (or as close as possible)

**V1. Merge the singly linked lists into one single sorted singly linked list in the init of the iterator and iterate over that list.**

- mergeLists merges the separate linked lists:
    - first with the second, the result with the third, etc.
    - all lists using a binary heap
- Operations *valid, next, getCurrent* have a complexity of Θ(1)
- Complexity of iterating through the hash table is the complexity of merging the lists.

Complexity of merging lists one by one:

$$\left.\begin{matrix}\text{HT with m positions}\\ SortedMap\ with\ n\ elems\end{matrix}\right\} \Rightarrow average\ number\ of\ elems\ in\ a\ list: \frac{n}{m} = \alpha\ (load\ factor)$$

Merge the first list with the second, the result with the third, etc.

- list1 + list2 => list12 => α + α = 2α
- list12 + list3 => list123 => 2α + α = 3α
- list123 + list4 => list1234 => 3 α + α = 4 α
- …

Total merging: $2\alpha + 3\alpha + \cdots + m\alpha \approx \left.\begin{matrix}\frac{m*(m+1)}{2}\ \alpha\\ \alpha = \frac{n}{m}\end{matrix}\right\} \rightarrow \frac{m\ (m+1)}{2}\ \frac{n}{m} \Rightarrow \in \theta\ (n * m)$

All lists using a binary heap:

- Add from each list the first node to the heap
- Remove the minimum from the heap, and add to the heap the next of the node (if exists)
- The heap will contain at most k elements at any given time (k is the number of the listst, $1 \leq k \leq m$) => height of the heap is $O(\log_2 k)$

- Merge complexity:
  - $O(n \log_2 k)$, if $k > 1$
  - $\Theta(n)$, if $k = 1$

**V2. Create a copy of the hashtable (just the table, the nodes stay the same) and find the minimum**

init – create a new table and copy in it the nodes from the hashtable (just the first one) and find the position of the minimum – Θ(m) complexity

getCurrent – return the information from the node from the minimum position – Θ(1) complexity

next – remove the minimum node (replace it with its next) and find the position of the next minimum - Θ(m) complexity

valid – next should set the position of the minimum to a special value when there are no more elements. Valid should just check for this special value - Θ(1) complexity

Complexity of iterating through the entire hashtable: Θ(n*m)