DSA – Seminar 6

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

- Assume
  - We memorize only the keys from the Map
  - Keys are integer numbers

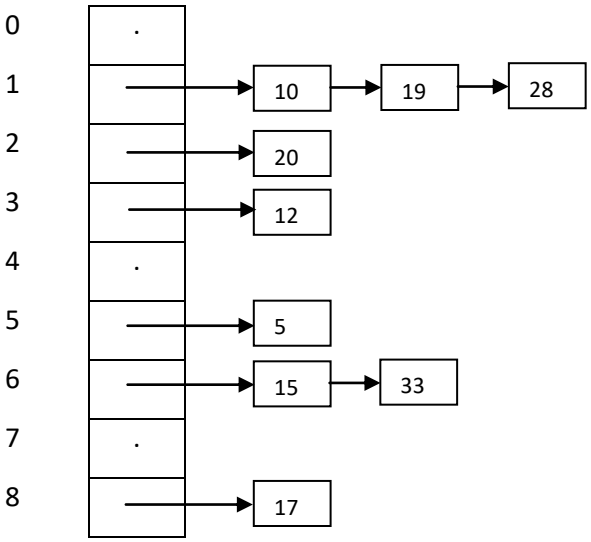
For ex:

- Keys from the map: 5, 28, 19, 15, 20, 33, 12, 17, 10 – Keys have to be unique!
- HT
  - 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

SM:



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 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
- …

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

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
  - Θ(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)

2. Map – reprezentation on a hash table – collision resolution with coalesced chaining

- Assume:
  - We memorize only the keys
  - The keys are integer numbers

For ex:

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

K	5	18	16	15	13	31	26
h(k)	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 the 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
- 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:

Map:
m: Integer
t: TKey[]
next:Integer[]
firstFree: Integer
h: Tfunction

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

Function search(map, k):
// for simplicity we return the position where the key was found, or -1
// in case of a real map, you return the value associated to the key
      i ← map.h(k)
      while (i ≠ -1 and map.t[i] ≠ k) execute
            i ← map.next[i]
      end-while
      if i = -1 then
            search ← -1
      else
            search ← i
end-function
Complexity: Θ (m) in worst case, but on average Θ(1)
```

Remove – that's the tricky operation

Simple examples for remove (before discussing the complicated ones):

Hash function for all the elements that appear in the examples:

<u>Example 1:</u>

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 -1)

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

<u>Example 2:</u>

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

- We can simply remove 11 (make the position -1 and the next of position 1 -1 as well)

~~0~~	1	2	3	4
~~11~~ -1	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

- If we put a -1 at position 1 (to remove 11), we will not be able to find 56 anymore (search for 56 starts from position 1). So we need to move 56 to position 1.

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

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).
-

	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. 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)
2. 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 key 5, which is at position 5

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

```
subalgorithm remove(map, k) is
    i ← map.h(k)
    j ← -1 {previous of i, when we want to remove node from pos i, we need
its previous node}

    {find the key to be removed. Set its previous as well}
    while i ≠ -1 and map.t[i] ≠ k execute
        j ← i
        i ← map.next[i]
    end-while
    if i= -1 then
        @key does not exist
    else
        {find another key that hashes to i}
        over ← false {becomes true when nothing hashes to i}
        repeat
            p ← map.next[i] {first position to be checked}
            pp ← i {previous of p}
            while p ≠ -1 and map.h(map.t[p]) ≠ i execute
                pp ← p
                p ← map.next[p]
            end-while
            if p = -1 then
                over ← true
            else
                map.t[i] ← map.t[p]
                j ← pp
                i ← p
            end-if
        until over
        {remove key from position i}
        if j = -1 then
            {parse the table to check if i has any previous element}
             idx ← 0
             while (idx < map.m and j = -1) execute
                  if map.next[idx] = i then
                        j ← idx
                  else
                        idx ← idx + 1
                  end-if
             end-while
        end-if
        if j ≠ -1 then
            map.next[j] ← map.next[i]
        end-if
        map.t[i] ← -1
```

```
        map.next[i] ← -1
        if map.firstFree > i then
            map.firstFree ← i
        end-if
    end-if
end-subalgorithm
```