DSA - Seminar 5

1. Consider the following problem: Determine the sum of the largest *k* elements from a vector containing *n* distinct numbers. For example, if the array contains the following 10 elements [6, 12, 91, 9, 3, 5, 25, 81, 11, 23] and k = 3, the result should be: 91 + 81 + 25 = 197.

   I. Find the maximum k times (especially good if k is small)
   - If we just call the maximum function 3 times for our example, it will return 91 each time, so we need a solution where we also have an upper bound, and we are searching for the maximum which is less than that value.
   - First, maximum is 91. At the second call we want the maximum which is less than 91, we will get 81.
   - At the second call we want the maximum which is less than 81, we will get 25.
   - Complexity of the approach: $\Theta(k*n)$ – finding the maximum is $\Theta(n)$ and we do this k times.

   II. Sort the array in a descending order and pick the first k elements (especially good is k is large).
   - Sorting can be done in $\Theta(n*\log_2 n)$ time
   - Computing the sum of the first k elements: $\Theta(k)$
   - In total $\Theta(n*\log_2 n) + \Theta(k) \in \Theta(n*\log_2 n)$

   III. Use a binary max-heap. Add all the elements to the heap and remove the first k.
   - Adding an element to a heap with n elements is $O(\log_2 n)$.
   - Removing an element from a heap with n elements is $O(\log_2 n)$.
   - In total we have $O(n*\log_2 n) + O(k*\log_2 n)$. Since n >= k, this is $O(n*\log_2 n)$

```
function sumOfK(elems, n, k) is:
//elems is an array of unique integer numbers
//n is the number of elements from elems
//k is the number of elements we want to sum up. Assume k <= n
       init(h, "≥") //assume we have the Heap data structure implemented. We
       initialize a heap with the relation "≥" (a max-heap)
       for i ← 1, n execute
              add(h, elems[i]) //add operation was discussed at Lecture 8
       end-for
       sum ← 0
       for i ← 1, k execute
              elem ← remove(h) //remove operation was discussed at Lecture 8
              sum ← sum + elem
       end-for
       sumOfK ← sum
end-function
```

   IV. How can we reduce the complexity? Well, we do not need all the elements in the heap, we are always interested in the k largest ones. If we consider the example from above,

we can work in the following way, always keeping just the k maximum elements up until now:

- Initially we keep 6, 12, 91
- When we get to 9, we can drop 6, because we know for sure that it is not going to be part of the 3 maximum numbers (we already have 3 numbers greater than this). So we keep 12, 91, 9.
- When we get to 3, we know it is not going to be part of the 3 maximum elements (We already have 3 elements greater than that). Similar with 5.
- When we get to 25, we can drop 9, and go on with 12, 91, 25.
- Etc.

We can keep the k elements that we consider in a heap. Should it be a min-heap or max-heap?

When we have the k largest elements at a given point, we will be interested in the minimum of these elements (this is what we compare to the new element that is considered and this is what we remove if we find a larger one) so it should be a min-heap.

```
function sumOfK2(elems, n, k) is:
//elems is an array of unique integer numbers
//n is the number of elements from elems
//k is the number of elements we want to sum up. Assume k <= n
       init(h, "≤") //assume we have the Heap data structure implemented. We
       initialize a heap with the relation "≤" (a min-heap)
       for i ← 1, k execute //the first k elements are added "by default"
              add(h, elems[i])
       end-for
       for i ← k+1, n execute
              if elems[i] > getFirst(h) then //getFirst is an operation which returns
the first element from the heap.
                     remove(h)//it returns the removed element, but we do not need it
                     add(h, elems[i])
              end-if
       sum ← 0
       for i ← 1, k execute
              elem ← remove(h) //remove operation was discussed at Lecture 8
              sum ← sum + elem
       end-for
       sumOfK2 ← sum
end-function
```

- Complexity? Our heap has maximum k elements, so operations have a complexity of $O(\log_2 k)$. We call add at most n times (worst case, when every element is greater than the root of the heap) and remove at most n times. So in total we have $O(n*\log_2 k)$
- If you do not use an already implemented heap, but have access to the representation, you can make the previous implementation slightly more efficient (complexity will not change though):
  - the last *for* will not have to remove elements, just simply to add up the sum of the elements from the heap array (but for this you need access to the array)
  - the middle for loop will not have to do a remove and an add. You can just overwrite the element from position 1 (this is what would be removed anyway) with the newly added element and do a bubble-down on it.

V. Can we improve complexity even more? We know that if we have an array we can transform it into a heap in a complexity O(n) – it was discussed at heapsort. So we can do something similar to version III, but instead of adding the elements one by one to the heap we could transform the array into a max-heap and then remove k elements from it. This approach has a complexity of $O(n + k*\log_2 n)$

2. 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

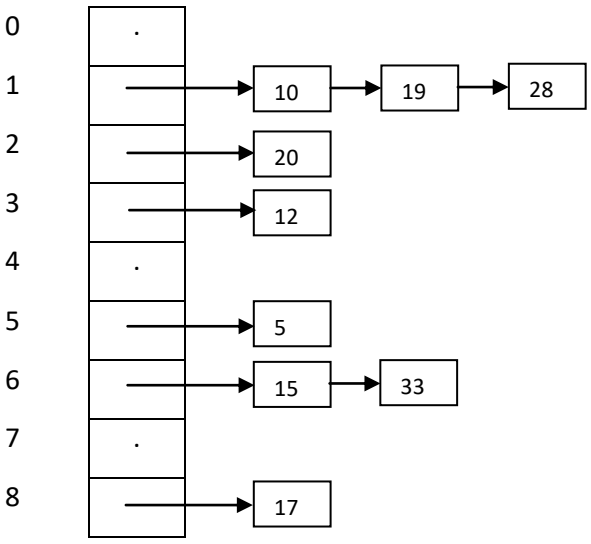
For ex:

- 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)

Standard iteration code (for example, for printing the content of the sortedmap)

```
subalgorithm print(sm) is:
	iterator(sm, smit)
	while valid(smit) execute
		e ← getCurrent (smit)
		print(e)
		next(smit)
	end-while
end-subalgorithm
```

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:

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

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)