DSA – Seminar 6

1. **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

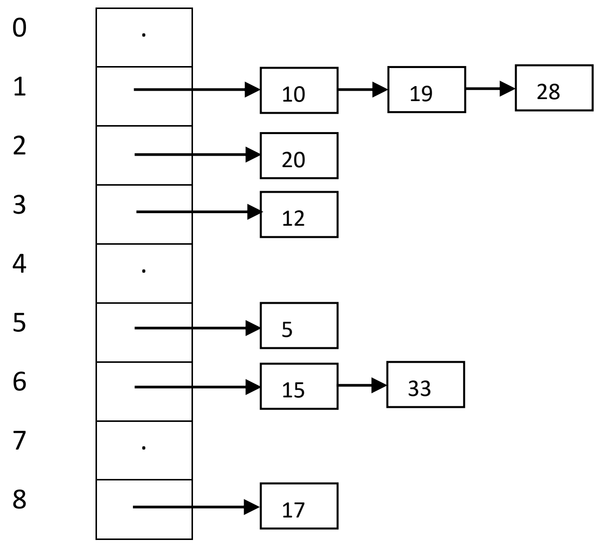
- Keys from the SortedMap: 5, 28, 19, 15, 20, 33, 12, 17, 10 – Keys have to be unique!
- Hash Table
  - m = 9
  - Hash function defined with the division method
    - h(k) = k mod m

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.

<u>Complexity of merging lists one by one:</u>

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

<u>Merge the first list with the second, the result with the third, etc.</u>

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

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

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 – $\Theta(m)$ complexity

getCurrent – return the information from the node from the minimum position – $\Theta(1)$ complexity

next – remove the minimum node (replace it with its next) and find the position of the next minimum - $\Theta(m)$ 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 - $\Theta(1)$ complexity

Complexity of iterating through the entire hashtable: $\Theta(n*m)$

2. A Bidirectional Map is a Map which supports bidirectional lookup: given a key you can find the corresponding value and given a value, you can find the corresponding key. This implies that values are unique as well. The interface of ADT BidirectionalMap contains the following operations:

- Init(bdm) – create a new, empty BidirectionalMap
- insert(bdm, k, v) – add a new key – value pair. If the key or the value already exists, the old pair is removed. The operation does not return anything.
- search (bdm, k) – returns the value associated to a key. If the key is not in the map, it returns NULL_TVALUE
- remove(bdm, k) – removes a key and its associated value. It returns the value associated to the key or NULL_TVALUE if the key is not in the BidirectionalMap.
- reverseSearch(bdm, v) – returns the key associated to a value. If the value is not in the map, it returns NULL_TKEY.

Implement ADT BidirectionalMap in such a way that its operations have the following complexities:

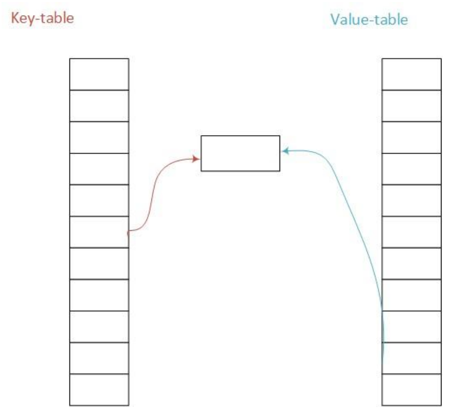
- init – $\Theta(1)$
- search, delete and reverseSearch – $\Theta(1)$ – on average
- insert – $\Theta(1)$ – on average and amortized

The – $\Theta(1)$ average complexity (especially for search) suggests that the solution for the problem is to use a hash table, which makes sense, since regular maps are in general implemented on hash tables. We also know that in case of a regular map the key of a pair is passed to the hash function to get the position of the pair. This is convenient, since the main operations (insert, search and delete) receive the key as parameter. But how to find the value efficiently? In a regular map, if you want to find a value, you need to keep checking pairs, until you find one with the given value. This is definitely not Θ(1) on average.

One simple solution would be to use two hash tables – one in which we keep key-value pairs (let's call it *key-table*) and another one in which we keep value – key pairs (let's call it *value-table*). Obviously, both of them would have their own hash function, since the type of keys and values might be different (and even if it is the same, we might want to use different functions), let's call them *h_key* and *h_value*. In init we initialize both hash tables. When we insert a new pair, we insert in both. When we search, we only search in the corresponding hash table (operation search searches in *key-table*, reverseSearch searches in *value-table)*. When we delete, we get the key as parameter, so we first delete from *key-table* and by this we get the value, so then we can delete from *value-table* as well. The only problem with this approach is that each key-value pair is stored in two different places, which is potentially a waste of memory. How could we make this more efficient?

Also, since we are talking about hash tables, what kind of collision resolution should we use?

In case of coalesced chaining and open addressing the key-value pairs are actually stored in the hash table (in the array), so if we have two tables, we must store the same pair twice. However, in case of separate chaining the information is actually stored in nodes (and the address of the nodes is in the array). So, if we choose to use separate chaining as collision resolution method, we can avoid storing the key-value pairs twice by doing the following: every key-value pair is stored in one single node, but that node is stored in both hash tables (probably at different positions / as part of different lists) based on the value returned by the two hash functions.



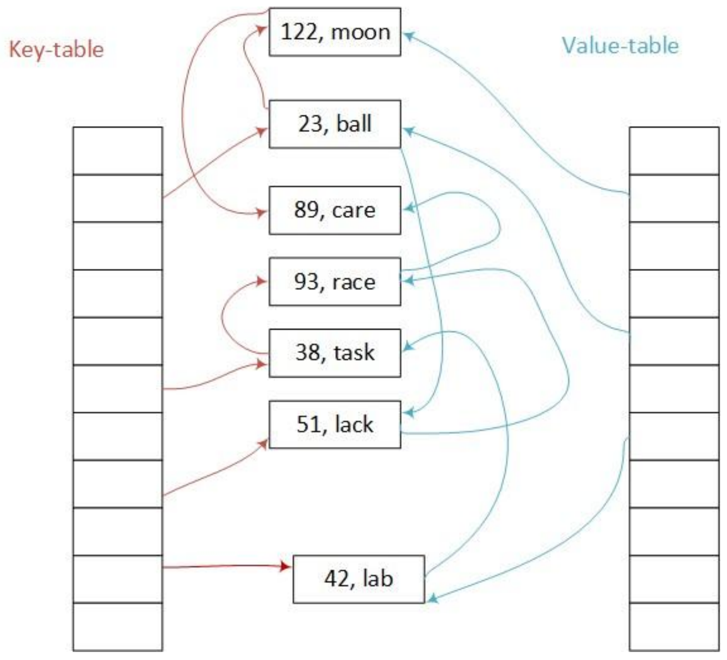
Since every node is part of two hash tables, it means that they need to have two *next* links, one for the next node whose key hashes to the same position where the current key hashes (*next_key*), and one for the next node whose value hashes to the same position where the current value hashes (*next_value*).
To see the situation better, let's take an example:

- m = 11 (the two tables can have the same size, they will store the same number of elements (we have the same number of keys and values) and will have the same load factor, so they will be resized in the same time as well.
- keys – integer numbers, the corresponding hash function will he:
    - h_key(k) = k % m

- values – strings, the corresponding hash function will be:
  - hash(v) = sum of the ascii code of the characters in the string
  - h_value(v) = hash(v) % m

Let's add the following pairs to the table:

Key	h_key(key)	value	hash(value)	h_value(value)
23	1	ball	411	4
42	9	lab	303	6
51	7	lack	411	4
122	1	moon	441	1
38	5	task	435	6
93	5	race	411	4
89	1	care	411	4



Let's see how the operations would be implemented in this case

- init – just create the two empty hash tables, set all pointers to NIL
- search – compute the value of the hash function for the key, and search in *key-table* for the element using the *next_key* links
- reverseSearch – same as search, but with the other table and hash function and the *next_value* links.
- remove – we have the key. Compute its hash and search for the key in the *key-table.* If you found it, remove it from both linked lists (the one with keys and the one with values). Removing it from the *key-list* is simple, since you traversed that list, you have the previous node. But how do you remove it from the *value-list?* Since at this point you have the value, you could do a search for the value as well starting from the *value-table*, and while searching for the node you could keep the previous, but this means extra traversal. We can avoid this, if we keep a doubly-linked lists, at least for the values. So, every node will have a *prev_value* link as well.

- <u>insert</u> – we want to insert a key-value pair. Before actually inserting it, we need to check if the given key and/or value is already in the map, because in that case, they need to be removed. So first we search for the key and if found we remove it (with whatever value it had associated). This is basically the same as remove. Then we search for the value and if found, we remove it (with whatever key it was associated to). This is like a reverse remove, we get to the node following the *next_value* links and now we want to remove it from both linked lists. As discussed for remove, it is easy to remove it from the *value-list*, but how to remove it from the *key-list* efficiently. The solution is that the *key-list* needs to be doubly linked as well. Once we have removed the required nodes, we need to add a new node to the map. This new node will be inserted in the *key-list* and in the *value-list* as well. Since the order of the nodes is not important, the easiest part is to insert it at the beginning of each list.

Representation:

<u>BDMNode:</u>

key: Tkey
value: Tvalue
next_key: ↑BDMNode
next_value: ↑BDMNode
prev_key: ↑BDMNode
prev_value: ↑BDMNode

<u>BDMap:</u>

key_table: (↑BDMNode)[]
value_table: (↑BDMNode)[]
m: Integer
size: Integer //the map does not have a size operation, but nr of pairs is needed to decide when to resize the tables. This is needed to keep the average Θ(1) complexity
h_key: TFunction
h_value: TFunction

Implementation of operations

```
Subalgorithm init(bdm):
    bdm.m ← 11
    bdm.key_table ← @ an array with bdm.m pointers to BDMNode
    bdm.value_table ← @ an array with bdm.m pointers to BDMNode
    for i ← 0, bdm.m-1 execute:
        bdm.key_table[i] ← NIL
        bdm.value_table[i] ← NIL
    end-for
    bdm.size ← 0
    @initialize h_key and h_value
end-subalgorithm

Subalgorithm destroy(bdm):
    for i ← 0, bdm.m-1 execute:
        while bdm.key_table[i]!= NIL execute:
            current ← bdm.key_table[i]
            bdm.key_table[i] ← [bdm.key_table[i]].next_key
```

```
                free(current)
            end-while
        end-for
        free(bdm.key_table)
        free(bdm.value_table)
end-subalgorithm

//auxiliary function, given a pointer to a node, removes that node both from the
//key- and the value-list.
subalgorithm removeNode(bdm, node):
        keypos ← bdm.h_key([node].key)
        valuepos ← bdm.h_value([node].value)

        //deal with the key_list
        if bdm.key_table[keypos] = node then // if it is the first in the key-list
            bdm.key_table[keypos] ← [bdm.key_table[keypos]].next_key
            if bdm.key_table[keypos] != NIL then
                [bdm.key_table[keypos]].prev_key ← NIL
            end-if
        else if [node].next_key = NIL then //it is the last in the keylist
            [[node].prev_key].next_key ← NIL
        else // it is between two existing nodes
            [[node].prev_key].next_key ← [node].next_key
            [[node].next_key].prev_key ← [node].prev_key
        end-if

        //deal with the value_list similarly
        if bdm.value_table[valuepos] = node then
            //it is the first in the valuelist
            bdm.value_table[valuepos] ← [node].next_value
            if bdm.value_table[valuepos] != NIL then
                [bdm.value_table[valuepos]].prev_value ← NIL
            end-if
        else if [node].next_value = NIL then
            [[node].prev_value].next_value ← NIL
        else
            [[node].prev_value].next_value ← [node].next_value
            [[node].next_value].prev_value ← [node].prev_value
        end-if
        //deallocate the node
        free(node)
        bdm.size ← bdm.size-1
end-subalgorithm

//auxiliary function, inserts a node in the map. It just includes the node in the
//map, it does not check keys and values for uniqueness.
subalgorithm insertNode(bdm, node):
        //add at the beginning of the key-list
        keypos ← bdm.h_key([node].key)
        valuepos ← bdm.h_value([node].value)
        [node].next_key ← bdm.key_table[keypos]
        [node].prev_key ← NIL
        if bdm.key_table[keypos] != NIL then
            [bdm.key_table[keypos]].prev_key ← node
        end-if
        bdm.key_table[keypos] ← node

        //similarly, add at the beginning of the valuelist
        [node].next_value ← bdm.value_table[valuepos]
        [node].prev_value ← NIL
        if bdm.value_table[valuepos] != NIL then
                [bdm.value_table[valuepos]].prev_value ← node
        end-if
        bdm.value_table[valuepos] ← node
        bdm.size ← bdm.size + 1
end-subalgorithm


//auxiliary function, handles the resizing
subalgorithm resize(bdm):
        oldM ← bdm.m
        //need to change m now so that the hash functions use the new m
        bdm.m ← bdm.m * 2 + 1
        oldKeyTable ← bdm.key_table
        bdm.key_table ← @ an array with bdm.m pointers to BDMNode
        oldValueTable ← bdm.value_table
        bdm.value_table ← @ an array with bdm.m pointers to BDMNode
        for i ← 0, bdm.m-1 execute
                bdm.key_table[i] ← NIL
                bdm.value_table[i] ← NIL
        end-for
        for i ← 0, oldM-1 execute
                while oldKeyTable[i] != NIL execute
                        currentNode ← oldKeyTable[i]
                        oldKeyTable[i] ← [oldKeyTable[i]].next_key
        //no need to set prev for oldKeyTable[i], it will be changed anyway
        //reuse the existing node, set its links to nullptr and insert in the resized
table
                        [currentNode].next_key ← NIL
                        [currentNode].prev_key ← NIL
                        [currentNode].next_value ← NIL
                        [currentNode].prev_value ← NIL
                        insertNode(bdm, currentNode)
                end-while
        end-for
        //free just the arrays, the actual nodes are already joined in the new table
        free(oldKeyTable)
        free(oldValueTable)
end-subalgorithm

function findKey(bdm, k):
        keypos ← bdm.h_key(k)
        current ← bdm.key_table[keypos]
        while current != NIL && [current].key != k execute
                current ← [current].next_key
        end-while
        findKey ← current
end-function

function findValue(bdm, v):
        valuepos ← bdm.h_value(v)
        current ← bdm.value_table[valuepos]
        while current != NIL && [current].value != v execute
                current ← [current].next_value
        end-while
        findValue ← current
```

```
end-function


subalgorithm insert(bdm, k, v):
      if bdm.size / bdm.m >= 0.75 then
            resize(bdm)
      end-if

      //check if the key exists
      current ← findKey(bdm, k)
      if current != NIL then // found the key
            //check if it happens to have the same value. Why remove and re-insert?
            if [current].value = v then
                  @ return
            end-if
            //not the same value. We need to remove it.
            removeNode(bdm, current)
      end-while
      //check if the value exists
      current ← findValue(v)
      if current != NIL then
            removeNode(current)
      end-if
      //now we can add
      newNode ← allocate()
      [newNode].key ← k
      [newNode].value ← v

      insertNode(bdm, newNode)

end-subalgorithm


function search(bdm, k):
      current ← findKey(k)
      if current != NIL then
            search ← [current].value
      else
            search ← NULL_TVALUE
      end-if
end-function

function reverseSearch(bdm, v):
      current ← findValue(v)
      if current != NIL then
            reverseSearch ← [current].key
      else
            reverseSearch ← NULL_TKEY
      end-if
end-function


function remove(bdm, k):
      current ← findKey(k)
      result ← NULL_TVALUE
      if current != NIL then
            result ← [current].value
            removeNode(bdm, current)
```

```
        end-if
        remove ← result
end-function
```

Obs: Looking at the code you can see that in many cases we have duplicated code, we do something with the key-table and then we do the same with the value-table, and the code is almost the same, for example in the function *insertNode*. Removing such duplicated code seems like a good idea, but at first, it does not look simple: when we have duplicated code, in one part we work with *next_key* and *prev_key* while in other parts with *next_value* and *prev_value*. How to generalize this?
An interesting option is, if we store the pointers slightly differently in the node: instead of having 4 variables, we could have 2 vectors, each containing 2 pointers: one vector for *next* pointers and one vector for *prev* pointers. In both vectors in position 0 we would have the links for the keys and on position 1 the links for the values.
In this case, a significant part of the code could be parameterized, by passing it one hash table and an integer denoting whether we work with keys or values.

Obs2: Assuming that the keys and values take up a lot of space (this was the reason for using two hash tables with shared nodes instead of using two distinct hash tables with duplicated data), we can modify the insertion of a (*k, v*) pairs in the following way:

- if neither *k*, nor *v* exists in the map, insert a new node.
- If only *k* exists in the map (with a different value, *v_old*), remove the existing node from the current value list (the one according to *v_old*), change the value in the node to *v* and add it to the new value-list (the one according to the hash of *v*).
- If only *v* exists in the map (with a different key, *k_old*), remove the existing node from the current key list (the one according to *k_old*), change the key in it to *k* and add it to the new key-list (the one according to the hash of *k*).
- If both *k* and *v* exist but with different pairs (let's say *v_other* and *k_other*), remove the value node (the node with the pair *k_other* and *v*) completely, and remove the key node from its current value list (the one according to *v_other*), change the value in it to *v* and add it to the new value list (the one according to the hash function of *v*).

This avoids deallocating and recreating nodes as much as possible. However, the lists still need to be doubly linked, because in some cases we remove a node from a key list, in others from a value list.