Ben-Gurion University of the Negev

Data Structures
202.1.1031

Assignment No. 3

Responsible staff members:
Michal Shemesh (shemeshm@post.bgu.ac.il)
Dan Zlotnikov (danzlot@post.bgu.ac.il)
Dor Amzaleg (amzalegd@post.bgu.ac.il)

Publish date: 06.06.2024
Submission date: 27.06.2024

Contents

Tasks

0 Integrity Statement

To sign and submit the integrity statement (presented below), fill in your **full name and ID** in the method signature in the class IntegrityStatement. See the comment in the method signature for example.

If you have relied on or used an external source, **you must cite** that source at the end of your integrity statement. External sources include shared file drivers, Large Language Models (LLMs) including ChatGPT, forums, websites, books, etc.

"I certify that the work I have submitted is entirely my own. I have not received any part of it from any other person, nor have I given any part of it to others for their use. I have not copied any part of my answers from any other source, and what I submit is my own creation. I understand that formal proceedings will be taken against me before the BGU Disciplinary Committee if there is any suspicion that my work contains code/answers that is not my own."

Assignments without a signed integrity statement will get 0 grade!

1 Preface

1.1 Assignment Structure

In this assignment, we discuss two main data structures seen in class, with some modifications presented below. The assignment is to be implemented using Java using the included skeleton files.

For your convenience, all the tasks are listed in the table of contents above and start with Task X.Y.

You **must** read the entire assignment **at least once** before beginning your work!

1.2 General Instructions

- You are expected to test your code locally and on the VPL system and sign your names in the IntegretyStatement file as mentioned above. You should not perform the experiments on the VPL system.
- In this assignment, you **must not add any additional `.java` files** to the assignment. Your code should be submitted in the already existing files.
- You **may not** use any built-in Java data structure or collection except for the classes `List`, `LinkedList` and `ArrayList`.
- You may use built-in utility classes of Java: `Math`, `Random`, `Collection`, etc.
- Some of the tasks are marked as **Food for thought** - it is highly recommended that you discuss and try to solve them, but they **should not be included** anywhere in your submission.

Before starting your work, read the following instructions carefully:

- With each version of your work, submit it so that with each submission a larger part of your work is complete. This way, you won't lose major parts of your work due to a technical issue or unexpected circumstances.
- DO NOT wait until the final hour for submitting your work, because there might be a power stoppage/computer issues/internet issues/Moodle issues/etc.
- Save a continuous backup of your work in some cloud platform, so that you can access it from any device if your computer decides to malfunction.

2 Skip-List

2.1 Introduction

In this section, we will extend the basic Skip-List by adding additional functionality not defined originally in class. In order to do so, you are supplied with an implementation of Skip-List, in the abstract class **AbstractSkipList**.

This skip list is composed of nodes, each of these represents a single element in the data structure, and contains the fields:

- The key and the satellite data of the node [1].
- An array of pointers to the previous nodes, each position correlates to the previous node in the respective level of the skip list.
- An array of pointers to the next nodes, each position correlates to the next node in the respective level of the skip list.
- A height indicator.

The implementation contains two sentinel nodes; The ***Head*** node with $key = -\infty$ (using the value `Integer.MIN_VALUE`), and the ***Tail*** node with $key = +\infty$ (using `Integer.MAX_VALUE`), as can seen in Figure (1). This implementation assumes that there are no duplicate keys, therefore, you can ignore the case where several items have the same key.

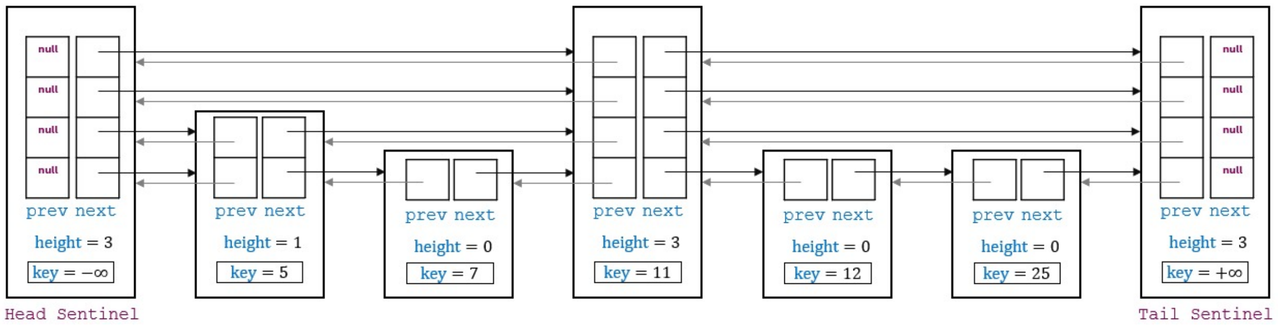

Figure 1: An example of an AbstractSkipList

Remark: The class **AbstractSkipList** is one of the possible implementations of the **Dynamic Set ADT** seen in class.

2.1.1 Given Implementations

In the class **AbstractSkipList** you are given implementations of the following operations and their time complexities:

- **insert(key)** - Inserts a node with key *key* into the DS - $\Theta(\log n)$ **expected**.
- **delete(node)** - Removes the node *node* from the DS - $\Theta(1)$ **expected**.
- **increaseHeight()** - Increases the height of the DS - $\Theta(1)$ **worst-case**.
- **search(key)** - Returns the node with key *key* from the DS if exists, or *null* otherwise - $\Theta(\log n)$ **expected**.
- **minimum()** - Returns the node with the minimal non $-\infty$ key in the DS - $\Theta(1)$ **worst-case**.

[1] In this section, the keys are integers, and the satellite data is null (we don't store actual data, just keys).

- **maximum()** - Returns the node with the maximal non $+\infty$ key in the DS - $\Theta(1)$ **worst-case**.
- **successor(node)** - Returns the successor of the node *node* from the DS - $\Theta(1)$ **worst-case**.
- **predecessor(node)** - Returns the predecessor of the node *node* from the DS - $\Theta(1)$ **worst-case**.
- **toString** - A basic toString implementation that prints all the values in each level.

2.2 Warm-Up and Familiarization

2.2.1 Implementation of Abstract Functions

In this section, we will familiarize ourselves with the structure of **AbstractSkipList** and the probabilistic process of determining the nodes' height.

The above-mentioned methods are almost complete implementations of the **Dynamic Set ADT**. They are dependent on two ***abstract*** functions that needed to be implemented by you:

- **find(key)** - Returns the node with the key *key*, or the node ***previous*** to the supposed location of such a node, if not in the data structure.
- **generateHeight()** - Returns a height generated by the result of a geometric process (defined below) with probability $p \in (0, 1)$, where p is a parameter of the data-structure.

Definition: The result of a ***Geometric Process*** with probability p is the number of coin tosses until the first success ("head") (including), with probability p for success.

Task 2.1: (6 points) Implement the function **find(key)** in the class **IndexableSkipList** given to you. In order to do so, you should consult the slides from lecture 8.

Task 2.2: (6 points) Implement the function **generateHeight()** in the class **IndexableSkipList** given to you. In order to do so, you should consult the slides from lecture 8.

Remark: The function **generateHeight** should use the method `Math.random()`.

Remark: This implementation assumes that both `Integer.MAX_VALUE` and `Integer.MIN_VALUE` aren't valid values in the Skip-List.

2.2.2 Analysis of the Probabilistic Process

In this section, we analyse the **expected** height of a node, generated by the geometric process defined above.

Task 2.3: (2 points) Implement the function **calculateExpectedHeight** in the class **SkipListUtils**. The function receives a parameter $p \in (0, 1)$, which is the probability for success in a coin toss, and should return the expected height of a node.

Remark: As defined above, the meaning of *"success"* in a coin toss, is stop increasing the node's height.

2.3 Order Statistics

2.3.1 Introduction

In Practical Session 5 we have seen an extended Dynamic Set ADT called **OrderStatisticDynamicSet ADT**. The additional operations are based on the following definition:

Definition: Given a set of values S and a value v, the ***index*** of v in S is the number of values $a \in S$ such that $a \leq v$.

Definition: The ***Order-Statistic*** ADT defines two additional operations:

- **rank(S, v)** - Returns the **index** of the value v in the set S. Notice, the value v might not appear in S.
- **select(S, i)** - Returns the value $v \in S$ whose index is i. Here we assume that if $|S| = n$, then $1 \leq i \leq n$.

We have also presented an extension of an AVL Tree to support these operations in $\Theta(\log n)$ worst-case time. In this part, we wish to extend the Skip List data structure to support these operations in $\Theta(\log n)$ **expected** time.

Remark: The common name for an order statistics skip list is **"Indexable SkipList"**. Those of you who would like to wander online, will find that this is a widely accepted term.

Task 2.4: (20 points) Implement both **Select(i)** and **Rank(val)** in the class **IndexableSkipList**.

To implement the functions mentioned previously, you have to change **exactly three** method implementations of the 9 given implementations: [**insert**, **delete**, **increaseHeight**, **search**, **minimum**, **maximum**, **successor**, **predecessor**, **toString**] in the class AbstractSkipList.
You are allowed to increase the time complexity of the original operations (listed at 2.1.1), to at most $\Theta(\log n)$ Expected.
You are allowed (and encouraged) to add new fields to the DS as you see fit.
You are allowed to change the implementation of the inner class **SkipListNode**.

Task 2.5: (3 points) Implement the function **changedMethodsArray** in the class **SkipListUtils**. This method should return a boolean array of size 9, where each value corresponds to one of the functions in the class **AbstractSkipList**, e.g. index 0 for **insert**, index 1 for **delete** and so on. The value should be **true** if the function was changed in the previous task, and **false** if not.

For example, if you changed the functions **search**, **minimum**, **maximum**, the returned array should look like this: **[false, false, false, true, true, true, false, false, false]**.

Reminder: **exactly three** methods should be flagged as **true**.

3 Hashing

3.1 Introduction

In this section, we will deep-dive into one of the more popular data structures, to better understand the different implementations seen in class, and get familiar with the costs associated with some of the operations of this data structure. To do so, we will implement several different hashing families, and two distinct collision-resolving schemes.

Definition: A ***Hash Table*** for the universe U is a pair (A, h) of an array A of size m, and a mapping $h : U \to [m]$ called ***Hash Function***, that is, a function that for each item in the universe, map it into a specific index of A.

In order to indicate how *occupied* the hash table is, we also defined the following term:

Definition: The ***Load Factor*** of a table of size m with n items, **including deleted signs**, is:

$$\alpha = \frac{n}{m}$$

As mentioned above, since we assume that $|U| \gg m$ (much greater than), according to the ***pigeonhole principle*** any mapping $h : U \to [m]$ isn't one-to-one, therefore, we anticipate that some items will collide in the same address. In order to cope with such collisions, we must define the behavior of the data structure in such events.

Definition: In ***Closed Addressing*** collision-resolving scheme, each item resides in the address given by the hashing function. Each address may contain several items and must keep track of all those using a different data structure.

The most common Closed Addressing scheme is putting all items mapped to an address in a singly linked-list, which is known as ***Chaining*** (Luhn 1953, Dumey 1956).

Definition: In ***Open Addressing*** collision resolving scheme, each item may be remapped from the address given by the hashing function. The scheme must define where to put items that suppose to reside in occupied addresses. In this scheme, since all items reside directly in the array A, we must enforce that $\alpha < 1$ and is ***a constant***. The most commonly used open addressing scheme is the ***Linear Probing*** (Amdahl, McGraw and Samuel 1954, Knuth 1963).

Reminder: In the Linear-Probing scheme, we try to put the item x in the address $h(x)$. If it is occupied, we continue to the next address (mod m) and continue to do so as long as the addresses are occupied.

On deletion, we don't remove the items from the table. Instead, we use a special flag (what we referred to in class as "D") to denote whether an item was deleted from a certain address.

3.1.1 Hash Functions

In addition to the collision-resolution scheme, we must also choose a hash function, hopefully getting a good distribution of the items in the different addresses of the table. However, finding such a function is a difficult task; In most cases, there is no **one** good function, and we instead pick at random a function from a family of functions that promise to ***usually*** provide a good distribution of the items.

Definition: A family of functions $\mathcal{H} \subseteq U \to [m]$ is called ***Universal*** if for each $x, y \in U$ the probability of picking at random a function $h \in \mathcal{H}$ that map both into the same address is at most $\frac{1}{m}$. Formally:

$$P_{h \in \mathcal{H}}\big(h(x) = h(y)\big) \leq \frac{1}{m}$$

The research on such families is vast, and different function families have different qualities. However, finding a truly universal family is quite difficult, and many of the well-known Universal Hashing Families are hard to compute on the standard computers in use.

3.1.1.1 Carter-Wegman (1977)

The first family of functions we are going to use in this task is the Carter-Wegman ***Modular*** hash family, seen in class:

Definition: The **Carter-Wegman** universal hashing family is defined for integer items. In this task, we limit the values to all items that fit the `int` data-type of Java, meaning $U = \texttt{int}$. In this family, we **randomly** choose two integers a, b and a prime long integer p such that:

$$a \in \{1, 2, 3, ..., \texttt{Integer.MAX\_VALUE} - 1\}$$

$$b \in \{0, 1, 2, ..., \texttt{Integer.MAX\_VALUE} - 1\}$$

$$p \in \{\texttt{Integer.MAX\_VALUE} + 1, \dots, \texttt{Long.MAX\_VALUE}\}$$

Given those, the hash function selected is:

$$h_{a,b}(x) = ((a \cdot x + b) \bmod p) \bmod m$$

Remark: Notice that in order to calculate the value $a \cdot x + b$, we save the outcome in a `long` variable. Since we enforce $m < 2^{31} - 1$, the result of the second mod ensures us the end result can be safely stored in an `int` variable.

Remark: In Java, the operator % isn't a one-to-one replacement of `mod`, and may return **a negative number**, but must comply with:

$$-m \leq (x \bmod) \, m \leq m$$

Therefore, you are given a function **mod** in **HashingUtils** in order to correct the result for negative numbers.

Reminder: In Java:

- The `int` data-type (32 bit variable) contains all values between -2^{31} and $2^{31} - 1$.
- The `long` data-type (64 bit variable) contains all values between -2^{63} to $2^{63} - 1$.

Remark: While this family guarantees universal hashing of integers, there are two major setbacks in this family:

1. In order to use this family, we must find a large prime number p.
2. The operations `mod` and `div` are immensely expensive operations in modern computer architecture.

3.1.1.2 Dietzfelbinger et al. (1997)

Definition: An ***Almost-Universal*** family of functions is a family that promises the same as a Universal family, with a higher probability for collisions; Instead of a probability of $\frac{1}{m}$, that probability of identical mapping of two items is $\frac{2}{m}$. That is: for each $x, y \in U$

$$P_{h \in \mathcal{H}}\big(h(x) = h(y)\big) \leq \frac{2}{m}$$

Definition: The **Dietzfelbinger-Hagerup-Katajainen-Penttonen (1997)** ***Multiplicative-Shift*** (for convenience: DHKP) *Almost Universal* hash family is defined for integers. In this task we limit the values to the `long` data-type of Java. In this family, we assume the size of the hash table m is of the form 2^k for some $k \in \mathbb{N}$, and we randomly pick an integer $a > 1$ and get the following hash function:

$$h_a(x) = ((a \cdot x) \bmod 2^w) \operatorname{div} 2^{w-k} \tag{1}$$

where w is the length of a computer word (in modern **64 bit** computers, that is $w = 64$).

In practice, an equivalent Java computation is:

$$h_a(x) = (a \cdot x) >>> (w - k) \tag{2}$$

You are invited to search for the difference between >>>, >> and << operators online. Notice that there is no need for the equivalent <<< operator for <<.

Remark: For those who wonder, the two computations are equal because in 64-bit computers, numbers are stored in 64 or less bits. Thus, performing a modulo (%) operation on them will return the same number, making sure that the number is 64 bits or less. Shifting right by x bits is identical to dividing by 2^x, because in a binary representation, each bit represents a different power of 2, which increments by 1 with each step to the left.

3.1.1.3 Hashing Strings

In this section we will discuss hashing objects that are more complex than integers. In our task, we will represent such objects using strings. For simplicity, our strings must hold only ASCII characters. In order to do so, we transform the string into an integer; We choose a random prime, q, such that

$$\texttt{Integer.MAX\_VALUE}/2 < q \leq \texttt{Integer.MAX\_VALUE}$$

and choose $2 \leq c < q$.

Now we look at each string as a sequence of characters $(x_1, x_2, x_3, \ldots, x_k)$, and transform it using the following mapping:

$$(x_1, x_2, x_3, \ldots, x_k) \mapsto \left(\sum_{i=1}^{k} \left((x_i \cdot (c^{k-i} \bmod q)) \bmod q \right) \right) \bmod q$$

We feed the result of this mapping into the simpler Carter-Wegman hash for $[q] \to [m]$.

(Food for Thought) Suggest a way to use the Carter-Wegman hashing for strings in order to hash any object type in Java. Keep in mind that there is more than one correct way to do so.

3.2 Hash Implementations

In this task, you are given two interfaces. The interface **HashFactory** represents a family of hash functions and is capable of returning a new ***random*** hash function when given a table size m. The second is the interface **HashFunctor** which represents a hash function, and is implemented as a nested-class inside each HashFactory implementation.

The **HashFactory** interface contains only one function:

- **pickHash(k)** - Given some $k \in \mathbb{N}$, returns a randomly chosen hash function from the family of hash functions, where $m = 2^k$.

The **HashFunctor** interface also contains a single function:

- **hash(key)** - Given a **valid** key *key*, calculates its mapping into $[m]$. m, the size of the matching hash table, is a field of the class (and defined at the constructor of this Functor).

Remark: The ***Factory*** design pattern is very common when it comes to creating multiple instances of some class, not needing to know what class to instantiate.

Remark: A Function-Object (commonly called ***Functor***) is a class that wraps a function. In this assignment, we wish to wrap our hash functions in a wrapper that allows us to perform the **hash(key)** method while being able to observe the internals of each hash function chosen.

In order to implement each of the following tasks, you need to implement the HashFactory classes. Each of these HashFactory classes, given a table size, m, returns an HashFunctor that represents the hash function from U (defined by the generic type) to m.

Notice that for each call of **pickHash(k)** we should get a randomly chosen hash function out of the family of functions the **HashFactory** represents. That means, picking at random the different parameters of the function. In order to do so, you should use the `Random` class of Java.

Remark: For your convenience, you are supplied with a utility class named **HashingUtils** that contains the following public methods:

1. **runMillerRabinTest(long suspect, int rounds)** - A function that runs the Miller-Rabin Primality Test for the number *suspect* for *rounds* times.
2. **genLong(long lower, long higher)** - A function that uniformly generates numbers between lower (inclusive) and higher (exclusive).
3. **genUniqueIntegers(num)** - Returns an array of size *num* filled with uniquely generated items of type `Integer`.
4. **genUniqueLongs(num)** - Returns an array of size *num* filled with uniquely generated items of type `Long`.
5. **fastModularPower(long x, long y, long mod)** - A function that calculates $x^y \bmod mod$.

You may find these functions useful in your implementations.

Task 3.1: (5 points) Implement the Carter-Wegman hash family for `int` $\rightarrow [m]$ in the class **ModularHash**.

Task 3.2: (5 points) Implement the Dietzfelbinger et al. hash family for `long` $\rightarrow [2^k]$ using the shortened version defined in equation (2) in the class **MultiplicativeShiftingHash**.

3.3 Hash Tables

3.3.1 Introduction

Recall that we defined the data structure hash-table as:

Definition: Let U be a universe of items. A ***hash table*** of size $m \ll |U|$ is:

- An array of size m.
- A hash-function $h : U \to [m]$.
- A collision-resolving scheme.

As explained in class, the collision-resolving schemes divide into two main groups:

1. ***Closed Addressing*** - each item is stored at the address given by h, but each address must hold all the items mapped to it in a different data-structure.
2. ***Open Addressing*** - only one item is stored at each address. On collision, the item must find an unoccupied space in the table, using a predefined rule.

The two main schemes we have discussed are ***Chaining*** and ***Linear Probing***:

- In the ***Chaining*** scheme, each address holds its items within a linked list (which may be a singly or doubly linked list).
- In the ***Linear Probing*** scheme, if an address is occupied, we continue to the subsequent address (modulo m) until finding an unoccupied address.

In both schemes, we define a **maximal load factor** upon creation of hash table, and when the table reaches this load factor, it performs ***Rehashing***. That is, creating a new table of size $2m$ with a **newly picked** hash function, and inserting all the items from the original table into the new one.

3.3.2 Implementation Details

In this section, we will implement hash tables that use both Chaining and Linear Probing. To do so, we will implement two classes:

1. The class **ChainedHashTable** that uses Chaining in order to resolve collisions. In this implementation, we allow any non-negative maximum load factor, $\alpha > 0$.
2. The class **ProbingHashTable** that uses Linear Probing in order to resolve collisions. In this implementation, we allow non-negative load factors that are smaller than 1, $0 < \alpha < 1$.

In both implementations, the constructor of the class gets two parameters:

- A value $k \in \mathbb{N}$ that represents the size of the table 2^k; Notice that in this assignment, all our tables are of a size of form 2^i.
- An instance of a **HashFactory**, that allows picking a hash function (a **Functor**) for any table size. The table picks a new hash function from this factory each time it performs rehashing.

Notice that we expect that on an Insert operation that causes the load factor to reach the maximal load factor defined in the table's creation, a rehashing will occur, and the table size doubles each time.

Remark: In this assignment, we enforce $Key \in \{$`Integer`, `Long`$\}$. Therefore, you may assume that:

- The **HashFactory** classes used are your implementations of the previous section.
- No additional types will be checked.

Remark: You can assume no rehashing will cause the table size to increase above `Integer.MAX_VALUE`, and there is no need to check for this case in your code.

Task 3.3: (12 points) Implement the Chaining-based hash table in the class **ChainedHashTable<K, V>**.

Task 3.4: (12 points) Implement the Linear Probing-based hash table in the class **ProbingHashTable<K, V>**.

3.3.3 Experiments

In the class **HashingExperimentUtils** you are given the skeleton of the methods **measureInsertionsProbing**, **measureSearchesProbing**, **measureInsertionsChaining** and **measureSearchesChaining** which should perform the following:

- Build a hash table of size 2^{16} (either Chaining-based or Probing-based).
- Insert items to fill the table up to the maximal load factor (without causing rehashing!) **or** perform searches - where 50% of all searches are successful and the rest are unsuccessful.
- Return an array with the average insertion and search times (in nano-seconds).
- The inserted and searched items should be chosen at random.

Remark: Read this explanation carefully before performing the next tasks.

We would like to perform measurements on **ProbingHashTable** and **ChainingHashTable**.
In each task, you are required to write a function which returns an array representing average computation times.
Each function you write should compute an average computation time of insertion **or** search for different values of a max load factor α which will be given in the task.

The returned array will consist of average computation times, where each cell will contain the average computation time for the corresponding α, i.e. the value in cell 0 will correspond to the average insertion time with $\alpha = 1/2$.

Remark: In order to measure time, you can use the function **System.nanoTime()**.
From the official Java documentation: "**System.nanoTime()** *Returns the current value of the most precise available system timer, in nanoseconds.*"
You are not required to use this specific method, feel free to use any other function that is included in the built-in Java libraries of your project. It is encouraged that you explore and learn more about the language as you go.

Task 3.5: (2 points) Implement the function **measureInsertionsProbing** in the class **HashingExperimentUtils**.
The function should run the measurements on **ProbingHashTable**, and return the average insertion time array for $\alpha \in \{\frac{1}{2}, \frac{3}{4}, \frac{7}{8}, \frac{15}{16}\}$.
For example: if the measured times for the α values were $\{1.1, 1.6, 2.4, 3.0\}$, the returned array should look like this: **[1.1, 1.6, 2.4, 3.0]**.

Task 3.6: (2 points) Implement the function **measureSearchesProbing** in the class **HashingExperimentUtils**.
The function should run the measurements on **ProbingHashTable**, and return the average search time array for $\alpha \in \{\frac{1}{2}, \frac{3}{4}, \frac{7}{8}, \frac{15}{16}\}$.
As in the previous task, the returned array should correspond to the measure values.

Task 3.7: (2 points) Implement the function **measureInsertionsChaining** in the class **HashingExperimentUtils**.
The function should run the measurements on **ChainingHashTable** , and return the average insertion time array for $\alpha \in \{\frac{1}{2}, \frac{3}{4}, 1, \frac{3}{2}, 2\}$.
As in the previous task, the returned array should correspond to the measure values.

Task 3.8: (2 points) Implement the function **measureSearchesChaining** in the class **HashingExperimentUtils** .
The function should run the measurements on **ChainingHashTable**, and return the average search time array for $\alpha \in \{\frac{1}{2}, \frac{3}{4}, 1, \frac{3}{2}, 2\}$.
As in the previous task, the returned array should correspond to the measure values.

(Food for Thought) What can we deduce regarding the relation of the load-factor to the average operation time in Chaining-based hash tables?

(Food for Thought) What can we deduce regarding the relation of the load-factor to the average operation time in Probing-based hash tables?

4 Designing a Data Structure according to specifications

In this section, we will design a new data structure according to a specified ADT and its time complexity requirements. This DS will contain `int` values, thus, *val* is always of type `int`. The requirements of the ADT are:

Operation	**Description**	**Time Complexity**
`Init(N)` (in code: `MyDataStructure(N)`)	Initializes the DS, given that the maximal number of items that may reside in the DS is $N \in \mathbb{N}$	$\Theta(N)$ Worst Case
`Insert(val)`	Inserts the value *val* into the DS if it isn't contained already. Returns true if and only if the item was inserted.	$\Theta(\log n)$ Expected
`Delete(val)`	Removes the value *val* from the DS if exists, and returns true if and only if the item was removed.	$\Theta(\log n)$ Expected if exists, $\Theta(1)$ Expected if isn't
`Contains(val)`	Returns true if and only if the DS contains the value *val*	$\Theta(1)$ Expected
`Rank(val)`	Returns the number of items in the DS that: $item.value \le val$	$\Theta(\log n)$ Expected
`Select(index)`	Returns the item of size *index* in the DS for $1 \le index \le DS.size$	$\Theta(\log n)$ Expected
`Range(low, high)`	Returns a list $\boldsymbol{L}$ containing all items in the DS such that $low \le item.value \le high$ in an ascending order if *low* is contained in the DS, and *null* otherwise.	$\Theta(\lvert L \rvert)$ Expected

Task 4.1: (21 points) Implement a data structure that supports the requirements mentioned previously in the class **MyDataStructure**. Provide, in a comment, a **short** (5 lines each) explanation of each of the implementations, and explain the time complexity of each of the operations.

Good Luck!