Ben-Gurion University of the Negev

Data Structures

202.1.1031

Assignment No. 4

Responsible staff members:
Michal Shemesh (shemeshm@post.bgu.ac.il)
Nadav Keren (nadavker@post.bgu.ac.il)
Matan Malaci Goren (gorenm@post.bgu.ac.il)

Publish date: 20.5.2023
Submission date: 10.06.2023

אוניברסיטת בן-גוריון בנגב
Ben-Gurion University of the Negev

Contents

Tasks and Questions

0 Integrity Statement

To sign and submit the integrity statement (presented below), fill in your name/s in the method signature in the class IntegrityStatement. If you submit the assignment alone, write your full name; if you submit the assignment in pairs, write your full names separated by “and”. 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."

1 Preface

1.1 Assignment Structure

In this assignment, we discuss two main data structures seen in class, with some modifications presented below. This assignment includes both practical and theoretical parts. The practical assignment is to be implemented using Java (version 17) using the included skeleton files. The theoretical questions are to be answered in the given skeleton file of the expected pdf.

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

You **must** read the entire assignment **at least once** before starting work on it!

1.2 General Instructions

The following instructions apply to all of the questions and tasks in this assignment:

- 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 (i.e., everything the implements `Collection<E>`) except for the classes `List`, `LinkedList` and `ArrayList`.
- You may use built-in utility classes of Java: `Math`, `Random`, `Collections`, etc.
- In each of the theoretical questions, you are expected of short and concise answers, that **do not exceed 5 lines each**. This does not include the pseudo-code or the description of an algorithm, that may require more than 5 lines. This does not include questions that specify other limits.
- Replace the question marks in the tables provided in the assignment with your answers.
- All of the theoretical questions must be answered in a pdf file based on the pdf skeleton file (found here) which uses the same formatting and numbering of the assignment.

  In this skeleton file, you are supplied with a code for each of the tables in the assignment. In addition, you are required to write pseudo-codes according to the example of the `Algorithmic` environment given to you in the skeleton file. **You should remove this example from the final submission**, and use it only as a reference for writing pseudo-code in the assignment.
- **It is strongly recommended that you submit an updated version of your assignment after each section you solve. Avoid submissions in the last hour and preferably avoid submissions in the last evening.**

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 abstract implementation of Skip-List, in the class **AbstractSkipList**.

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

- An array of pointers to the previous links, each position correlates to the previous link in the respective level of the skip list.
- An array of pointers to the next links, each position correlates to the next link in the respective level of the skip list.
- A height indicator.
- The value of the node.

Therefore, this implementation consists of several **doubly-linked** lists, and contains two sentinel nodes; The ***Head*** node represents the $-\infty$ value using the value `Integer.MIN_VALUE`, and the ***Tail*** node represents the $+\infty$ value 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.

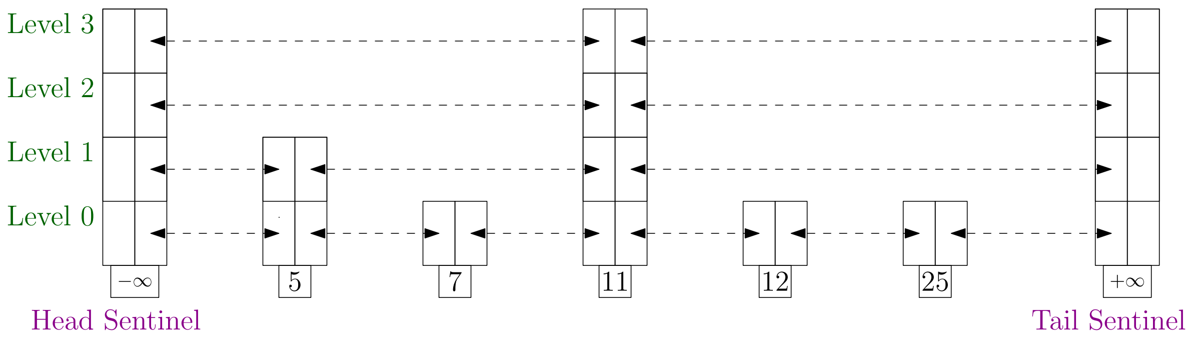

Figure 1: An example of an AbstractSkipList

2.1.1 Given Implementations

In the class **AbstractSkipList** you are given implementations of the operations:

- **insert(key)** - Inserts the key *key* into the DS - $\Theta(\log n)$ **expected**.
- **delete(node)** - Removes the node *node* from the DS - $\Theta(1)$ **expected**.
- **search(key)** - Returns the node with key *key* from the DS if exists, or *null* otherwise - $\Theta(\log n)$ **expected**.
- **minimum()** - Returns the minimal non $-\infty$ value in the DS - $\Theta(1)$ **worst-case**.
- **maximum()** - Returns the maximal non $+\infty$ value in the DS - $\Theta(1)$ **worst-case**.
- **successor(node)** - Returns the successor of the key of the node *node* from the DS - $\Theta(1)$ **worst-case**.
- **predecessor(node)** - Returns the predecessor of the key 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 ADT. They are dependent on two ***abstract*** functions that needed to be implemented by you:

- **find(key)** - Returns the node of 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: (3 point) Implement the two functions mentioned above 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 will test the behavior of the height values created by the geometric process. To do so, we wish to test the average height generated by a sequence of calls to **generateHeight** you have implemented in the previous section.

For each sequence length $x \in \{10, 100, 1000, 10000\}$, we wish to run 5 experiments and:

1. Calculate the expected number of levels of **a node** (that is $E[\ell] = 1/p$, as shown in practical session 6) in the skip-list.
2. For each run, calculate the **average** number of levels generated (that is the average height of items + **1**), marked as $\hat{\ell}_i$ for the i'th run.
3. Calculate the average difference (delta) between the results of each run and the expected height.

In order to perform the experiment, you are given a skeleton of a utility class named **SkipListExperimentUtils** which includes the following functions:

- **measureLevels(p, x)** - This function generates x random heights with probability p. It calculates the average number of levels ($\hat{\ell}$) described above and returns its value.
- **main(args)** - This is a standard main function, in which you should perform the experiments.

The remaining functions in this class are explained in the next section.

Task 2.2: (3 points) Implement the function **measureLevels** in **SkipListExperimentUtils**, and for each $p \in \{0.33, 0.5, 0.75, 0.9\}$ fill in the following table:

p = <Insert the value of p here>							
x	$\hat{\ell}_1$	$\hat{\ell}_2$	$\hat{\ell}_3$	$\hat{\ell}_4$	$\hat{\ell}_5$	**Expected Level** ($E[\ell]$)	**Average delta** ($\frac{1}{5} \cdot \sum_{i=1}^{5} \lvert \hat{\ell}_i - E[\ell] \rvert$)
1 (example)	1	1	2	3	2	2	0.6
5 (example)	2.6	1.8	2.0	1.6	2.6	2	0.36
10	?	?	?	?	?	?	?
100	?	?	?	?	?	?	?
1000	?	?	?	?	?	?	?
10000	?	?	?	?	?	?	?

Remark: In this section, you should have 4 tables (for the 4 probabilities) that sum up 20 experiments each (that is 4 sizes of x and 5 runs for each size).

Question 2.3: (2 points) How does the value of p affects the average number of levels ($\hat{\ell}_i$) generated by **generateHeight**?

Question 2.4: (2 points) How does the value of x effects the average delta generated in each experiment? What can we deduce from that?

2.2.3 Analysis of the operations

Now, we want to test the average time for each operation type on the data structure. In the class **SkipList-ExperimentUtils**, you are given the skeleton of the following functions:

- **measureInsertions(p, size)** - This function returns a skip list of size $size + 1$ and the average time (in nano-seconds) for each insertion in the process as a **Pair**. The values inserted are the values in $\{0, 2, 4, \ldots, 2 \cdot size\}$ **in random order**, and the data structure is with the parameter p.
- **measureSearch(DS, size)** - This function returns the average time (in nano-seconds) of a search in the data structure DS of an item from the values in $\{0, 1, 2, \ldots, 2 \cdot size\}$ **in random order**.
- **measureDeletions(DS, size)** - This function returns the average time (in nano-seconds) for each deletion in the data structure DS of an item from the values in $\{0, 2, 4, \ldots, 2 \cdot size\}$ **in random order**.
- **main(args)** - This is a standard main function, in which you should perform the experiments.

Remark: For **each experiment** we perform the flow:

1. We call **measureInsertions** with the paremeters p and $size$. This creates a new skip-list, populates it according to the definition above, and returns the average time of insertions, i.e.:

$$\texttt{Average Insertion Time} = \frac{\texttt{Total Time of Insertions}}{\texttt{Number of Items Inserted}}$$

2. We call **measureSearch** with the already populated skip-list from the previous step, and perform both **successful** searches (of items in the skip-list) and **unsuccessful** searches (of items that aren't in the skip-list), and return the average search time, that is:

$$\texttt{Average Search Time} = \frac{\texttt{Total Time of Search Operations}}{\texttt{Number of Items Searched}}$$

3. Finally, we call **measureDeletions** with the skip-list from the first step. This function removes all the items from the skip-list, and returns the average time of deletion, that is:

$$\texttt{Average Deletion Time} = \frac{\texttt{Total Time of Deletions}}{\texttt{Number of Items Deleted}}$$

After each experiment, we get three measurements: Average Insertion Time; Average Search Time; Average Deletion Time.

Notice that since the order of operations is random, and due to the fact that time measurements on computers tend to be ***noisy*** (as explained in Practical Session 1), a single test may return unreliable results. To make our experimentation more reliable, we perform this flow **30 times**, and calculate the average of each type of measurement.

That means that each table entry is of the form:

$$\texttt{Average Measurement} = \frac{1}{30} \cdot \sum_{i=1}^{30} \texttt{measurement}_i$$

Task 2.5: (2 points) Implement the functions given in **SkipListExperimentUtils**, which includes the main function to perform the experiments described in the next question.

Notice: You may add additional methods to the class as you please.

Question 2.6: (2 points) Fill the following table for each $p \in \{0.33, 0.5, 0.75, 0.9\}$ with the average of **30 experiments** for each value of $x \in \{1000, 2500, 5000, 10000, 15000, 20000, 50000\}$:

p = <Insert the value of p here>

x	Average Insertion	Average Search	Average Deletion
1000	?	?	?
2500	?	?	?
5000	?	?	?
10000	?	?	?
15000	?	?	?
20000	?	?	?
50000	?	?	?

Remark: In this section, you should have 4 tables (for the 4 probabilities) that sum 210 experiments each (that is 7 sizes of x and 30 runs for each size).

Task 2.7: (2 points) After you have completed the experiments above, plot (using your favorite method, such as Microsoft Excel, Google Sheets, Libre Office, etc.) a graph for each operation type (Insertion, Search and Deletion), depicting the average time of operation as a function of x. Each graph should include **4** plot lines - each representing a different value of p.

Question 2.8: (2 points) Using the graph created above, how does the value of p effects each of the operations?

In questions (2.9) (2.10) (2.11), we will revise the calculation of the expected height/size/Insertion/Search cost as seen in Lecture 8, and extend it to any value $0 < p < 1$. In your proofs you will show **all the mathematical developments** done in order to prove your result. Therefore, the answers may be longer than 5 lines.

Question 2.9: (3 points) What is the expected number of nodes in a conceptual **Skip-List** containing n elements, not including the Sentinels, as a function of p? Prove your answer.

Question 2.10: (4 points) What is the expected height of a **Skip-List** as a function of p? Prove your answer.

Question 2.11: (4 points) What is the expected time complexity of Insertion/Search as a function of p? Prove your answer. How does it compare with the empirical results in Question (2.6)?

2.3 Order Statistics

2.3.1 Introduction

In Practical Session 5 we have defined a new set of operations on a data structure by the following definitions:

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 the 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.

Task 2.12: (15 points) Implement both **Select(i)** and **Rank(val)** in the class **IndexableSkipList**.

To implement the functions mentioned previously, you may change some implementation details in the class **AbstractSkipList**. These changes should be minimal. You are allowed to increase the time complexity of the original operations (listed at 2.1.1), to at most $\Theta(\log n)$ Expected.

The changes must be documented in the theoretical section as follows:

- What additional space (number of memory cells) has been added? What is the purpose of this memory addition? What is the space complexity added to the data structure?
- What operations differ from the original implementation? For each of these changes, you must include a Pseudo-Code of the new algorithm **highlighting the changes over the original** (see an example for highlighting in the Pseudo-Code example in the beginning of the pdf skeleton file).

  If the time complexity has changed, explain the new time complexity,and why it has changed compared to the original operation.
- What new methods were added, a short explanation of each method and its algorithm, a Pseudo-Code of the operation, and a time complexity analysis of the method.

Hint: Think about the process we have seen on AVL-Trees. Skip-Lists' operations are similar (but not identical).

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, perform one of the following variants:

1. We don't remove the items from the table. Instead, we use a special flag to denote whether an item is valid or deleted.

2. On deletion, we perform rehashing to the entire block.

Remark: In this assignment, we exclusively refer to the variant of Linear Probing that uses deleted signs (Variant 1).

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 (1979)

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 **Crater-Wegman** (1979) 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 such that:

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

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

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

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:

$$-m < x \,\%\, m < m$$

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 `<<`.

Question 3.1: (5 points) Explain why the computation in equation (1) is truly equal to the one in equation (2) in a 64-bit machine.

Remark: In order to answer question (3.1), you should understand how the elementary operations (that is, $+$ and $\times$) are calculated within the limited-sized types `int` and `long`.

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 $[q] \to [m]$.

Question 3.2: (5 points) Suggest a way to use the Carter-Wegman hashing for strings in order to hash any object type in Java.

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. **genUniqueStrings(num, stringMinLength, stringMaxLength)** - Returns a list of *num* uniquely generated items of type `String` of length $stringMinLength \leq length \leq stringMaxLength$.
6. **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 3.3: (2 points) Implement the Carter-Wegman hash family for `int` $\to [m]$ in the class **ModularHash**.

Task 3.4: (2 points) Implement the Dietzfelbinger et al. hash family for `long` $\to [2^k]$ using the shortened version defined in equation (2) in the class **MultiplicativeShiftingHash**.

Task 3.5: (2 points) Implement the Carter-Wegman hash for strings assuming the strings are ASCII in the class **StringHash**.

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.
- An 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<T>** 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<T>** 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 $T \in \{$`Integer`, `Long`, `String`$\}$. 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.

Remark: You can assume no duplicate items are inserted, and you do not need to handle such an event.

Task 3.6: (6 points) Implement the Chaining-based hash table in the class **ChainedHashTable<K, V>**.

Task 3.7: (6 points) Implement the Linear Probing-based hash table in the class **ProbingHashTable<K, V>**.

In the class **HashingExperimentUtils** you are given the skeleton of the methods **measureOperationsChained(max-load-factor)** and **measureOperationsProbing(max-load-factor)** 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!)
- Perform $2^{16} \cdot$ `max-load-factor` searches - where 50% of all searches are successful and the rest are unsuccessful.
- Returns a Pair with the average insertion and search times (in nano-seconds).
- The inserted and searched items should be chosen at random.

Question 3.8: (2 points) Run the measurements on **ProbingHashTable** for **30 times**, and complete the average Insertion and Search time (in nano-seconds) for the max load factors $\alpha \in \{1/2, 3/4, 7/8, 15/16\}$ in the following table:

Linear Probing		
max α	Average Insertion	Average Search
1/2	?	?
3/4	?	?
7/8	?	?
15/16	?	?

Question 3.9: (2 points) What can we deduce regarding the relation of the load-factor to the average operation time in Linear-Probing based hash tables?

Question 3.10: (2 points) Run the measurements on **ChainingHashTable** for **30 times**, and complete the average Insertion and Search time (in nano-seconds) for the max load factors $\alpha \in \{1/2, 3/4, 1, 3/2, 2\}$ in the following table:

Chaining		
max α	Average Insertion	Average Search
1/2	?	?
3/4	?	?
1	?	?
3/2	?	?
2	?	?

Question 3.11: (2 points) What can we deduce regarding the relation of the load-factor to the average operation time in Chaining-based hash tables?

In **HashingUtils** you are given functions that return a list of *num* unique random items of type `Integer`, `Long` and `String` respectively. Using these functions, we wish to test the cost of the hashing functions in our implementations of hash tables. In order to do so, we will perform the same experiment executed in **measureOperationChained(max-load-factor)** for max-load-factor $= 1$. We want to run the experiment 10 times using a table that accepts `Long` keys, and 10 times for a table accepting `String` keys.

To do so, you are requested to implement the following functions in **HashingExperimentUtils** and run the experiment:

- **measureLongOperations()** - Measures the cost of a table using Dietzfelbinger et al. for `Long` items.
- **measureStringOperations()** - Measures the cost of a table using Carter-Wegman for Strings, with strings of length 10 to 20.

Question 3.12: (2 points) What are the results of the experiment described above? What can we deduce from these results?

3.4 Theoretical Questions

The ADT ***Dictionary*** includes three operations: **Insert(x)**, **Search(key)** and **Delete(x)**. It can be implemented using Hash Tables with Chaining, with time complexity of $\Theta(1)$ worst case, $\Theta(1)$ expected, and $\Theta(1)$ expected, respectively. In this section, we will look at the implementation of additional operations using Chaining-based Hash Tables.

In each of the following questions, you must describe an algorithm, write a Pseudo-Code, and analyze its time complexity.

Remark: Notice that you **may not change the time complexity of the original Insert/Search/Delete operations**.

Question 3.13: (2 points) Describe the operation **Successor(val)**, where *val* exists in the hash table.

Question 3.14: (2 points) Describe the operation **Minimum()**.

Question 3.15: (2 points) Describe the operation **Rank(val)**, where *val* might not appear in the hash table.

Question 3.16: (2 points) Describe the operation **Select(i)**, where $1 \leq i \leq n$.

4 Designing a data structure according to given 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: $item.value \leq val$	$\Theta(\log n)$ Expected
`Select(index)`	Returns the item of size *index* in the DS for $1 \leq index \leq DS.size$	$\Theta(\log n)$ Expected
`Range(low, high)`	Returns a list $\boldsymbol{L}$ containing all items in the DS such that $low \leq item.value \leq high$ in an ascending order if *low* is contained in the DS, and *null* otherwise.	$\Theta(\lvert L \rvert)$ Expected

Task 4.1: (10 points) Implement a data structure that supports the requirements mentioned previously in the class **MyDataStructure**. Provide a **short** (5 lines each) explanation of each of the implementations, and explain the time complexity of each of the operations.

Good Luck!