Ben-Gurion University of the Negev

Data Structures

202.1.1031

Assignment No. 2

Responsible staff members:
Michal Shemesh
Dor Amzaleg
Nimrod Berman

Publish date: 20.04.2025
Submission date: 11.05.2025

Contents

0 Integrity Statement

To sign and submit the integrity statement (presented below), fill in your 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 the Abstrct Data Type of Dynamic Set, and three main data structures seen in class: Array (sorted and unsorted), Linked List (sorted and unsorted), and AVL Tree. The assignment is to be implemented using Java using the included skeleton files. The assignment consists of three main parts:

- Warm-Up and Familiarization.
- Implementing Dynamic Set.
- Designing Data Structures according to specifications.

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

1.2 General Instructions

- The skeleton files for the assignment are available in the VPL system of the assignment. You should download the skeleton files to your computer, implement your code, and upload your submission to the VPL system.
- You are expected to test your code locally and on the VPL system and sign your name and ID in the IntegretyStatement file as mentioned above.
- 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 **must not edit the given implementation** in the skeleton files. You are **permitted** (and actually should) to add fields and/or auxiliary methods to some of the classes as you wish for your implementations.
- You **may not** use any built-in Java data structure or collection.
- You **can** use:
  - Primitive variables.
  - Basic arrays.
  - The classes given in the skeleton files.
- You **can assume that the input is legal** for all the functions in the assignment. There is **no need to check and throw exceptions** for illegal arguments.
- Your code will be tested automatically for correctness, and manually for **efficiency and clarity**.

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 (9 points) Warm-Up and Familiarization

2.1 Given Implementations

In the skeleton files you are given five classes, of the following data structures:

- **MyArray** (unsorted)
- **MySortedArray**
- **MyLinkedList** (unsorted)
- **MySortedLinkedList**
- **MyAVLTree**

In addition, you are given a class **Element** representing a general element in a general data structure with key and satellite data, and three extend classes representing specific elements in specific data structures:

- **ListLink** (a link in a Linked-List).
- **TreeNode** (a node in an AVL-Tree).
- **ArrayElement**[1] (an element in an Array).

Read those classes carefully and make sure you understand the given implementations!

2.2 Implementation of specific methods

Task 2.1: In the class **MyArray**, implement the method **reverse()**. The method should reverse the order of the elements in the array.

Task 2.2: In the class **MyLinkedList**, implement the method **reverse()**. The method should reverse the order of the elements in the linked-list.

Task 2.3: In the class **MyAVLTree**, implement the method **depthOfMin()**. The method should return the depth of the element with the smallest key in the tree (return (-1) if the tree is empty).

[1] As you will see, in the class **ArrayElement** there is a field *index* that should be maintained. One might ask, why to complicate such a simple and direct implementation such an array with extra fields. The reason is to be consistent with the operations of Dynamic-Set. Some of them receives pointer to an element as input, and should have the information about the element's position in the data-structure.

3 (16 points) Implementing Dynamic Set

As learned in class, Dynamic Set is an abstract data type supporting the following operations: **search**, **insert**, **delete**, **successor**, **predecessor**, **minimum**, **maximum**.
In this section, you are required to implement a Dynamic Set with the given time complexities for each operation:

- **search(k)** - Returns the element in the DS whose key is k if exists (return *null* if no element in the DS has key k) - $O(n)$.
- **insert(x)** - Add the element x to the DS - $O(n)$.
- **delete(x)** - Removes the element x from the DS (the operation receives a pointer to x) - $O(1)$.
- **minimum()** - Returns the element in the DS with the smallest key (return *null* if no such exists) - $O(1)$.
- **maximum()** - Returns the element in the DS with the largest key (return *null* if no such exists) - $O(1)$.
- **successor(x)** - Returns the element in the DS with smallest key that is larger than $x.key$ (return *null* if no such exists) - $O(1)$.
- **predecessor(x)** - Returns the element in the DS with largest key that is smaller than $x.key$ (return *null* if no such exists) - $O(1)$.

Task 3.1: In the class **MyDynamicSet**, implement a dynamic set that supports the requirements mentioned above.

Remark: The class constructor, **MyDynamicSet(int N)**, receives an int N as input. This N is the maximum number of elements in the data structure at each time. If you choose to use an array to implement your dynamic set, you can use an array of size N without worrying about the need to expand the array. If you choose to use a linked-list or an AVL-tree, this N has no use.
Note the difference between n (the actual number of elements currently in the dynamic set, which is relevant for the time complexity requirements) and N.
This remark is relevant for all data structures in this assignment.

Remark: When your code will be tested, the input x for the function **insert** will be a general element of instance type **Element**. You should create a new instance of a specific element according to the data structure you choose to use to implement your dynamic set (with the same key and satellite-data as x), and insert it to the dynamic set. The input for the functions **delete**, **successor**, **predecessor** will be an element which is **in** the dynamic set, that **you** inserted. So once you inserted elements of a specific instance type, you can be perfectly sure what is the instance type of the input to those functions.

4 (75 points) Designing Data Structures according to specifications

In this section, we will design new data structures according to a specified ADT and their time complexity requirements.

4.1 (30 points) First Data Structure

The data structure should support the following operations in the given time complexities:

Operation	Description	Time complexity
$Init(N)$[2]	Initiate an empty data structure. N is the maximum number of elements in the data structure at each time.	$O(1)$
$Insert(x)$	Add the element x to the DS.	$O(\log n)$
$FindAndRemove(k)$	Find the element in the DS with key k (if exists), and removes it from the DS. If no element in the DS has key k do nothing.	$O(\log n)$
$Maximum()$	Returns the element in the DS with the largest key (return *null* if no such exists)	$O(1)$
$First()$	Returns the element in the DS inserted first among all the elements currently in the DS. (return *null* if no such exists).	$O(1)$
$Last()$	Returns the element in the DS inserted last among all the elements currently in the DS. (return *null* if no such exists).	$O(1)$

Task 4.1: In the class **MyFirstDataStructure**, implement a data structure that supports the requirements mentioned above.

[2] The initiation in the code is by implementing the constructor $MyFirstDataStructure(N)$. Note the remark in Section 3 about the role of N.

4.2 (45 points) Second Data Structure

We need to Implement, for a client who owns a products factory, a data structure to store his products. Each product is characterized by a unique identification number, quality, price, and name, and is represented by the class **Product**. The optional qualities for a product are: 0/1/2/3/4/5. The data structure should support the following operations in the given time complexities:

Operation	Description	Time complexity
$Init(N)$[3]	Initiate an empty data structure. N is the maximum number of products in the data structure at each time.	$O(1)$
$Insert(product)$	Insert the product *product* to the data structure.	$O(1)$
$FindAndRemove(id)$	Find the product with the unique identification number *id* (if exists), and removes it from the DS. If no product in the DS has identification number *id* do nothing.	$O(n)$
$MedianQuality()$	Return the median quality of the products currently stored in the data structure. The median is defined as the $\lceil \frac{n}{2} \rceil$-th smallest element. (Return -1 if the data structure is empty)	$O(1)$
$AvgQuality()$	Return the average quality of the products currently stored in the data structure. (Return -1 if the data structure is empty)	$O(1)$
$RaisePrice(raise, quality)$	Raise the price of all the products **currently in** the DS with quality *quality* in the amount *raise*. The input *raise* is a positive integer.	$O(1)$
$MostExpensive()$	Return the most expensive product in the DS.	$O(1)$

Task 4.2: In the class **MySecondDataStructure**, implement a data structure that supports the requirements mentioned above.

Good Luck!

[3] The initiation in the code is by implementing the constructor $MySecondDataStructure(N)$. Note the remark in Section 3 about the role of N.