Ben-Gurion University of the Negev

Data Structures
202.1.1031

Assignment No. 2

Responsible staff members:
Paz Carmi (carmip@cs.bgu.ac.il) Idan Tomer (idantom@post.bgu.ac.il)
Matan Malaci Goren (gorenm@post.bgu.ac.il)

Publish date: April 13th, 2023
Submission date: May 4th, 2023

0 Integrity Statement

I, <Insert your name>, assert that the work I submitted is entirely my own. I have not received any part from any other person, nor did I give parts of it for others to use. I realize that if my work is found to contain code that is not originally my own, a formal case will be opened against me with the BGU disciplinary committee.

To sign and submit the integrity statement, fill your name/s in the method **signature** in the class *IntegrityStatement*. If you submit the assignment alone, write your full name, and if you submit the assignment in pairs, write your full names separated by "and". See the comment in the method **signature**.

1 Assignment structure

1.1. Hands-on programming and Theoretical questions

- Design an efficient data structures for solving the given problem (elaborated in this document) using the data structure taught in this course.
- Implement and test your data structure in Java programming language.
- Provide a time complexity analysis of your implementation.

1.2. Bonus

- Utilize your suggested data structure to solve a problem with a different setup. Provide input data and detailed explanations. (5 points)
- Improve the time complexity of some specific methods. (5 points)

Both parts of the bonus will be discussed in more detail below.

2 Introduction - Problem setup

The commercial airline company **NoCrash Flying** approached you to get help on planning an air collision prevention system. The main goal of the system is to reliably assure that the distance, in the Euclidean sense, between any two given planes will be at least a specific threshold. In addition, the system will assure that the density of the planes in the given flight space is not high. Clearly, your advised solution should be efficient as possible.
A detailed explanation of the system is presented in the following sections.

3 Auxiliary Classes & Interface

For this assignment, you are provided with the following:

- class *Point*
- class *Container*
- class *DataStructure*
- interface *DT*

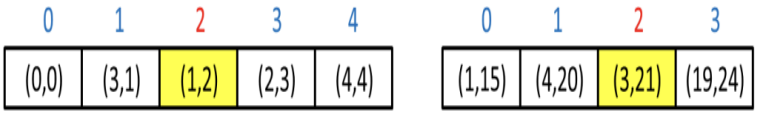
The *Point* class contains three fields: *Name*, X, and Y. Indicating the flight's serial number, X-axis coordinate, and Y-axis coordinate, respectively. **This class should NOT be changed!**

The *Container* class initially contains a field *Point data*. **You can add fields and methods to this class as you wish, but the initial *Point data* field and the corresponding getData() method must remain**.

The *DataStructure* class will contain your implementation of the non-collision system. The methods and run-time requirements for each method of the system are presented in the next section. **Note** that this class implements the *DT* interface. Therefore, you must implement all of the interface's methods. **You must not change DT!** - including the signature of the methods. Some methods are called with *Container* and some *Point*, you cannot modify the signatures of those methods. However, you are encouraged to add your own methods to the class implementation.

Important Notes and Remarks

1. As said above, you **MUST NOT** change *Point* and *DT*.
2. You may add methods and fields to *Container* and *DataStructure*. As well as any auxiliary class you might need.
3. You can assume that there will be no two different planes (i.e. Points) with the same X value or the same Y value. Meaning, the X and Y values are unique.
4. In some methods, as you will see, a boolean argument is passed - *axis*. Assume that a **true** value indicates the X-axis and a **false** value indicates the Y-axis.
5. The distance between two planes with the coordinates (x_1, y_1), (x_2, y_2) is given by the Euclidean distance between the two points, which is given by
$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$
6. The median value for a given sorted array of size n, indexed $0, \ldots, n-1$, is the element at index $\lfloor \frac{n}{2} \rfloor$. For example, for the median value according to the Y-axis (*axis*=**false**):



   Note that the Y-axis coordinate is the second element in each tuple.
7. You are allowed to use the classes *Math, Object, Comparable, Comparator*. You are also allowed to use the static method **Arrays.sort**.

4 System Requirements - DataStructure Methods

In the following, we present the methods required in the system and their worst-case time complexity, where n is the number of planes in the system. As aforementioned, these methods should be implemented as a part of the *DataStructure* class.

#	Returns	Operation	Time Complexity
1	(constructor)	**DataStructure**()	$O(1)$
2	void	**addPoint**(Point point)	$O(n)$
3	Point[ ]	**getPointsInRangeRegAxis**(int min, int max, Boolean axis)	$O(n)$
4	Point[ ]	**getPointsInRangeOppAxis**(int min, int max, Boolean axis)	$O(n)$
5	double	**getDensity**()	$O(1)$
6	void	**narrowRange**(int min, int max, Boolean axis)	$(\lvert A \rvert)$ (see Section 4.1.6)
7	Boolean	**getLargestAxis**()	$O(1)$
8	Container	**getMedian**(Boolean axis)*	$O(n)$
9	Point[ ]	**nearestPairInStrip**(Container container, double width, Boolean axis)*	$O\big(\min\{n, \lvert B \rvert \log \lvert B \rvert\}\big)$ (see Section 4.1.9)
10	Point[ ]	**nearestPair**()	see Section 4.1.10

* You can assume that the method **nearestPairInStrip** will only be called with a *Container* that was returned from the **getMedian** method.

4.1 Methods Description

For all methods, n is the number of planes in the system.

4.1.1 DataStructure()

Constructor. Creates an empty data structure with no planes.
Time complexity: $O(1)$.

4.1.2 void addPoint(*Point* point)

Method for adding a new plane to the data structure. You can assume that there is no other plane in the data structure with the same X or Y value, thus, the uniqueness assumption still holds.
Parameters: point - the new plane we want to add.
Time complexity: $O(n)$.

4.1.3 Point[ ] getPointsInRangeRegAxis(*int* min, *int* max, *Boolean* axis)

The method returns an array of all planes that have an **axis** value in the range of **[min,max]** (note that this is an inclusive range, i.e., including **min** and **max** values) **sorted in non-decreasing order by the axis value**. The axis considered in the method is based on the given **axis** parameter - **true** for X-axis and **false** for Y-axis.
Parameters:

1. min - lower bound of the range.
2. max - upper bound of the range.
3. axis - the desired axis, **true** for X-axis and **false** for Y-axis.

Time complexity: $O(n)$.

4.1.4 Point[ ] getPointsInRangeOppAxis(*int* min, *int* max, *Boolean* axis)

The method returns an array of all planes that have an **axis** value in the range of **[min,max]** (note that this is an inclusive range, i.e., including **min** and **max** values) **sorted in non-decreasing order by the other axis value**. The axis considered in the method is based on the given **axis** parameter - **true** for X-axis and **false** for Y-axis. Thus, the other axis can be achieved by simply performing **!axis** where ! represents not.
Parameters:

1. min - lower bound of the range.
2. max - upper bound of the range.
3. axis - the desired axis, **true** for X-axis and **false** for Y-axis.

Time complexity: $O(n)$.

4.1.5 double getDensity()

The method returns the density of the points in the structure. That is, the number of points in the structure, divided by the area of the smallest axis-aligned rectangle containing all points. The formula to calculate the density is

$$\frac{n}{(X_{max} - X_{min})(Y_{max} - Y_{min})}$$

where X_{max} is the maximal X-value of a point in the structure and $X_{min}, Y_{max}, Y_{min}$ defined similarly.
Note: We will not test the case where the density is undefined (e.g. when there is only one point in the data structure).
Time complexity: $O(1)$.

4.1.6 void narrowRange(*int* min, *int* max, *Boolean* axis)

The method deletes from the data structure all the points that have an **axis** value in the range of $[-\infty, \textbf{min})$ or in the range of $(\textbf{max}, \infty]$(note that these are exclusive ranges, i.e., not including **min** and **max** values)

Parameters:

1. min - lower bound of the range.
2. max - upper bound of the range.
3. axis - the desired axis, **true** for X-axis and **false** for Y-axis.

Time complexity: $O(|A|)$ where $|A|$ is the number of points to be deleted.

4.1.7 Boolean getLargestAxis()

The method returns the largest axis for the points in the data structures. That is, **true** if $(X_{max} - X_{min} > Y_{max} - Y_{min}$ and **false** otherwise, where X_{max} is the maximal X-value of a point in the structure and $X_{min}, Y_{max}, Y_{min}$ defined similarly.

Time complexity: $O(1)$.

4.1.8 Container getMedian(*Boolean* axis)

The method returns the median point in the chosen axis among all the points in the structure (see Section 3.

Parameters:

1. axis - the desired axis, **true** for X-axis and **false** for Y-axis.

Time complexity: $O(n)$.

4.1.9 Point[ ] nearestPairInStrip(*Container* container, *double* width, *Boolean* axis)

The method returns the two nearest points in a strip centered in **container** with width **width** by the axis **axis** (X-axis for **true** and Y-axis for **false**). If there aren't two points in the strip, the function should return an empty array. You may assume every two points on the same side of the container by the direction **axis** is at least $\frac{width}{2}$ apart. The strip is defined by all the points with value $Z_p \pm \frac{width}{2}$ in the axis **axis**, where Z_p is the value of the point in **container** in the axis **axis**.

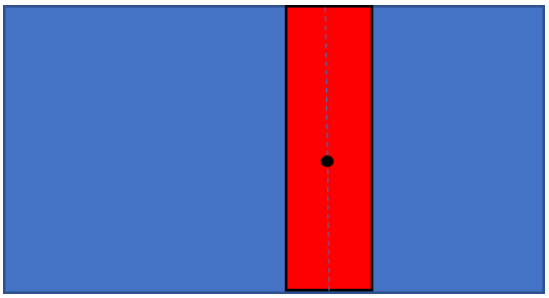


Figure 1: nearestPairInStrip example. the black point represents the container point, and the red rectangle represents the width

Parameters:

1. container - containing a point from the range (you may assume nearestPairInStrip will be called only with container returned from the method getMedian(Section 4.1.8) with the parameter **axis**. that is getMedian(**axis**).
2. width - the width of the strip.
3. axis - the desired axis, **true** for X-axis and **false** for Y-axis.

Time complexity: $O\big(\min\{n, |B| \log |B|\}\big)$ where $|B|$ is the number of points in the strip.

4.1.10 Point[ ] nearestPair()

The method returns the two closest points in the data structure. You may assume there are at least two points in the structure.

Time complexity: You should calculate the time complexity of this method, and explain your calculation in the theoretical questions part.

Guidance: The method performs the calculation using the following steps:

i If there are only two points return them. If there are less return empty set.

ii Find the largest axis **axis**.

iii Find the median in **axis**

iv Calculate the nearest pairs for all the points larger than the median recursively, and similarly (and recursively) for all the points smaller than the median.

v Choose the closer pair between the two pairs from Item iv and calculate the distance between them *minDist*.

vi Check if the strip centered in the median with width $2 \cdot minDist$ contains two points with a distance smaller than minDist.

(a) If there exists such pair, return it

(b) Else, return the pair from Item v

4.2 The method split

You should supply a pseudo-code for the method **split**(*int* value, *Boolean* axis) which returns two collections of points - one containing all the points larger than **value** in the axis **axis**, and the second one containing all the points smaller than **value** in the axis **axis**. The time complexity of the function should be $O(|C|)$ where $|C|$ is the number of points in the **smaller** collection (the one with fewer points). The time complexity of $O(n)$ will get a partial score, while worse time complexity solutions won't get any score.

Instructions:

a. You should explain what is returned from the method **split** and how can one access each of the points in the collection (the access doesn't need to be efficient).

b. You should write a pseudo-code for the method **split**.

c. You should analyze the time complexity of the method **split**.

d. The data structure can be modified after calling **split** (e.g. by calling **addPoint** or **narrowRange**), but the return value of **split** does not need to exist or perform correctly anymore.

(A more challenging solution will be really splitting the data structure into two different data structures with the corresponding points, while the original data structure ceases to exist).

5 Hands-on programming

The assignment includes several source files with:

- *Point* class and *DT* interface which you **must not** change.
- *Container* and *DataStructure* classes which you need to implement.
- *GUI* class you will use if you choose to implement the bonus (see Section 7)

You are allowed (and encouraged) to change *Container* class and add files and classes as you need, but you **must not** change any existing code (i.e., class name, *data* field and **getData** method).

You are supplied with an empty implementation of the class *DataStructure*. You are required to implement *DataStructure* so that it supports all the required methods described in Section 4.1 in the required run time. You are allowed to add methods, constructors, and fields as you need. Please notice our tests will call only the constructor and methods described in this file.

6 Theoretical questions

6.1. Describe your implementation shortly in a typed document. Explain which data structures you used and describe **with words** the algorithms you used for implementing the methods of *DS* interface.

6.2. Explain **shortly** the run time of **each** of the methods (for most methods, 1-2 lines will be sufficient).

6.3. Give a pseudo-code for the function **split**(*int* value, *Boolean* axis) and analyze the run time of your solution. All points will be given for a solution running in $O(|C|)$ time where $|C|$ is the number of points in the smaller group in the partition. A solution running in $O(n)$ time will get only some of the points, and slower solutions won't get points at all.

6.4. Determine and explain the run time of the method **nearestPair()** (your answer should be as optimal as possible).

6.5. **Bonus** (up to 5 points) – Explain how can one build from an existing *DataStructure* instance, two *DataStructure* instances, one containing all the points with X value larger than the median (included), and one containing all the points with X value smaller than the median. The build should take $O(n)$ time. The instances must of course support all *DataStructure* methods correctly.

Optional – Explain how can one use the above for improving **nearestPair()** run time.

The document must be typed and in PDF format.

7 Bonus: GUI - Graphical User Interface

In this section, you will be able to examine a different possible use for your built data structure. In the assignment's files, you will find a class named *GUI*. Running it will open a graphical user interface. After implementing Section 5 you will be able to use it for checking your code.

The application can load a JPG image and a TXT file describing which object is present in the image. Using the solution of Section 5, the app allows you to mark a certain area on the image and return the number of objects in the selected area (or the objects themselves). The app is straightforward to use. You should load the image and its matching special-format TXT file (format described below) first. Then you should choose the desired area by a continued left-click on the mouse followed by dragging to mark the area. The output will be shown below upon releasing the left-click. In some of the methods, the matching points in the data structure will be printed (notice in the standard state, points are not shown on the image).

In the assignment's files, you will find two sets containing a JPG image and a TXT file describing the objects in the image. You can use them to verify your data structures. You can also add your own sets.

The format of the TXT files is:

<name>;<x-coordinate>;<y-coordinate>

for example:

Moshe; 30; 55

You are encouraged to submit your own sets you used to validate your implementation, which doesn't have to match the plane's problem.

A **5 points bonus** will be given for submitting 2 original, meaningful, and different sets with at least 10 objects each (please note in your PDF file you submitted sets for bonus).

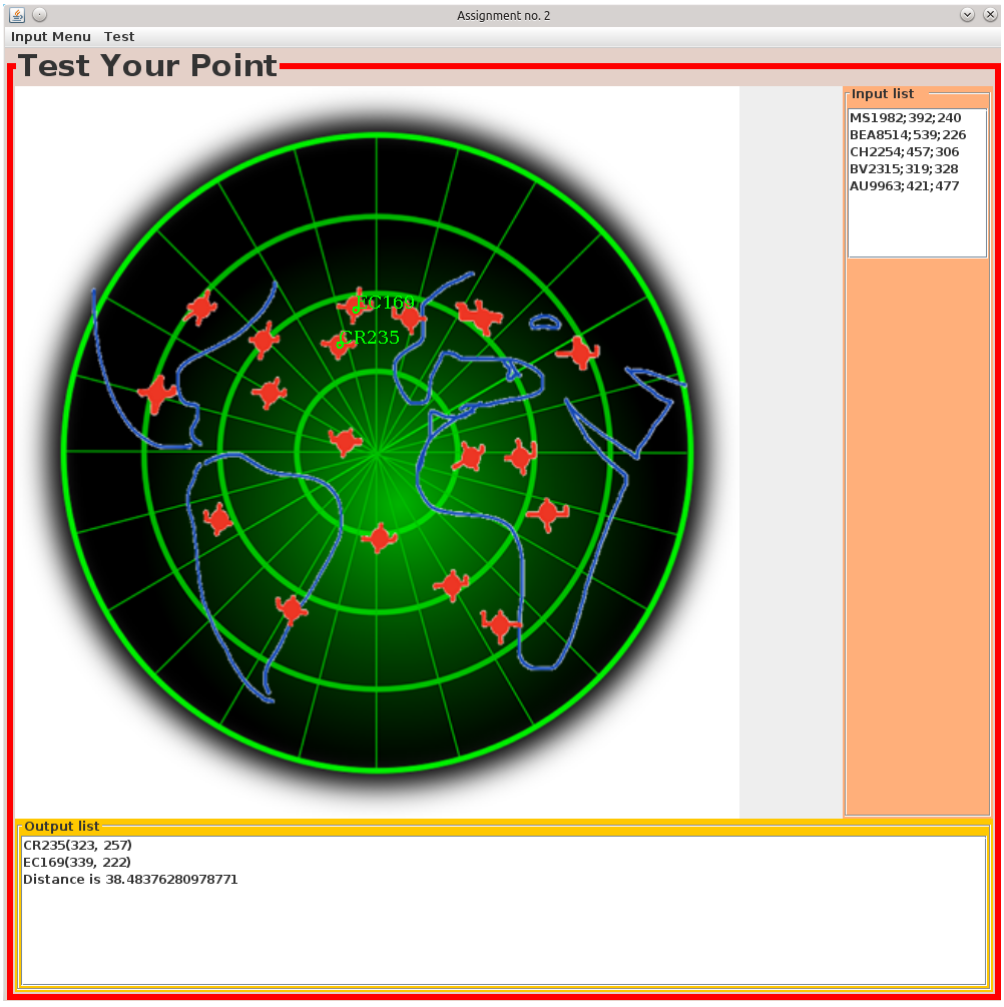


Figure 2: An example for GUI with planes on radar and selected areas

8 Important comments and submission requirements

1. It is strongly recommended to read the entire assignment and FAQ section (in the moodle) before you start writing your solutions.
2. The assignment may be submitted either by pairs of students or by a sole student.
3. You may not use generic data structures implemented by others (the developers of Java, Git projects, and so on).
4. Your code should be neat and well-documented.
5. When testing your code, you may use whatever tools you want, including classes and data structures created by others. The restriction above applies only to the code you submit to us.
6. Your implementation should be as efficient as possible. Inefficient implementations will receive a partial score depending on the magnitude of the complexity.
7. As you have learned, in this course in general and specifically in this assignment the analysis of runtime complexity is always a worst-case analysis.
8. Your code will be tested in the VPL environment, and therefore you must make sure that it compiles and runs in that environment. Code that will not compile will receive a grade of 0. We provide some basic sanity checks for you to make sure that your code compiles.

9. Don't forget to sign the statement in Section 0. Your code will be checked for plagiarism using automated tools and manually. The course faculty, CS department, and the university regard plagiarism with all seriousness, and severe actions will be taken against anyone that was found to have plagiarized. A submitted assignment without a signed statement will receive a grade of 0.

10. You can always assume the input is valid. You will not:

    - Be asked to add an existing point.
    - Be asked to delete points from an empty structure.
    - Receive from our tests *null* as a parameter to any of the methods of the interface.

11. Please follow the submission guidelines in the VPL environment strictly. For testing, we will use our own *Point*, *GUI* classes.

Good Luck!