

Assignment

SUBJECT: DATA STRUCTURES AND ALGORITHMS

SUBJECT ID: 504008

Receipt management, finding relationship between customers and products

(Students should read the instruction carefully before starting your work)

I. Overview

A grocery store need to manage their receipts. After each receipt, customer need to rate products with stars, the grocery store want to find rules from the relationship between customers and products for appropriate business strategy.

Features that students need to implement:

- Build a balance binary search tree (AVL) to store the receipts.
- Build a graph to represent the relationship between customers and products to find the result of queries.

II. Provided resources

Source code are provided by default, included in these files:

- Input files and expected output files:
 - o Folder *data* includes these files: *receipt.txt* contains list of receipts, *customer.txt* contains list of customers, *product.txt* contains list of products and *rating.txt* contains products' rating of customers.
 - o Folder *output* includes these 5 files: *Req1.txt*, *Req2.txt*, *Req3.txt*, *Req4.txt*, *Req5.txt* contain sample results from provided **main** method which calls to different requirements in the assignment.
- Source code files:
 - o *Main.java*: creates objects and calls to methods that students will implement.

- *Receipt.java*, *Customer.java*, *Product.java*: orderly contain class **Receipt**, **Customer**, **Product** which is defined initially. **Students must not edit these files.**
- *AVL.java*: contains class **AVL** and methods that are used to build an AVL tree. Students will implement some methods in this file.
- *ReceiptManagement.java*: contains class **ReceiptManagement** which includes a property “AVL tree” to store list of receipts. In this file, the read file method, write file methods and other methods to build tree, traverse tree, search, based on calls to methods of class **AVL** are provided by default. Students will implement methods of class **AVL** for these methods to be usable. **Students must not edit this file.**
- *RatingQuery.java*: contains class **RatingQuery** which include read file and write file methods provided by default, students will implement the code into this file following some requirements.

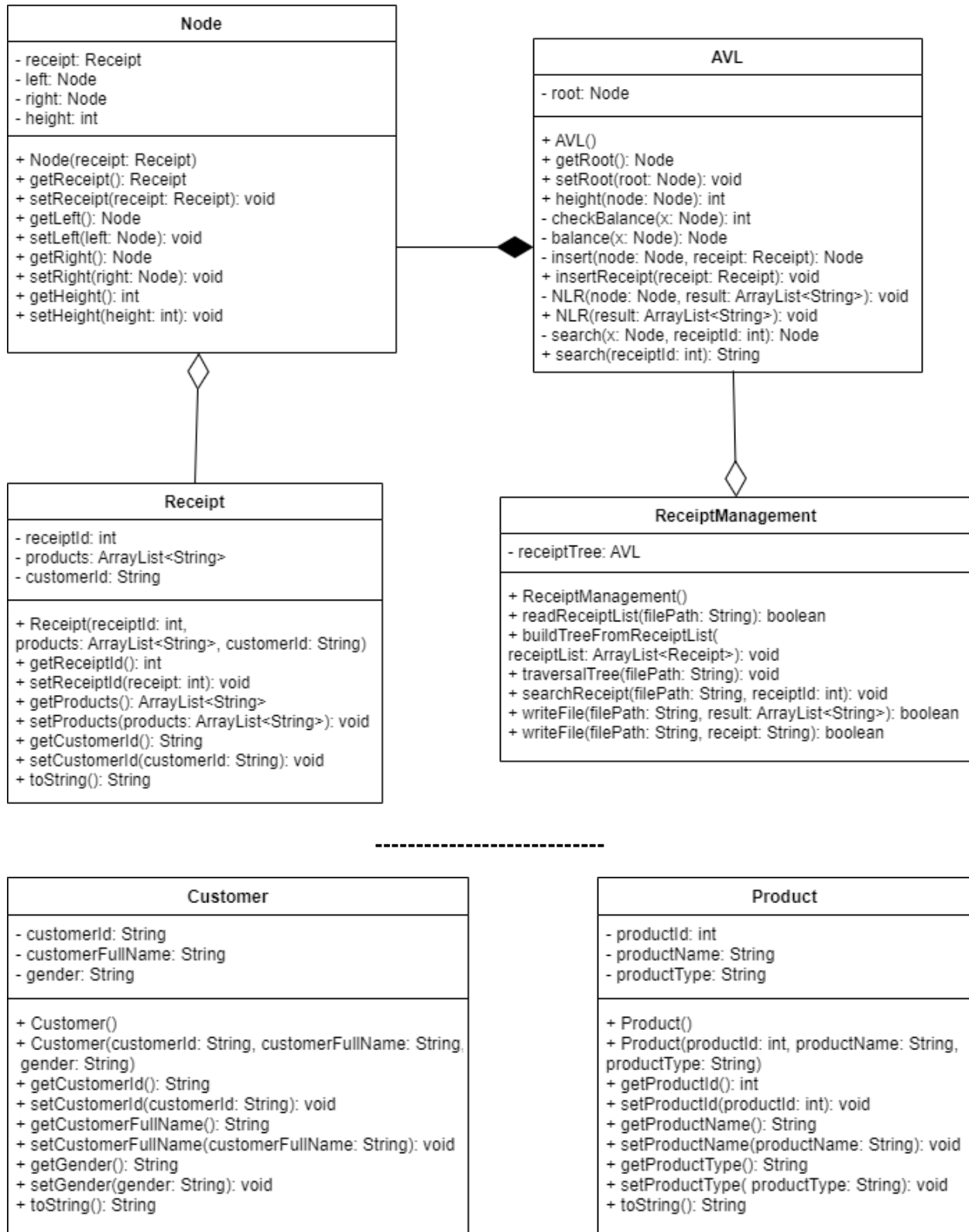
III. The order to do the assignment

- Students download and extract provided resources.
- The assignment contains 5 requirements. The requirements are symbolize:
 - Requirement 1: YC1
 - Requirement 2: YC2
 - Requirement 3: YC3
 - Requirement 4: YC4
 - Requirement 5: YC5
- In YC1 and YC2, students implement an AVL tree by creating file *Node.java* which contains class **Node** and completing the missing methods in *AVL.java* file.
- If students implement the AVL tree correctly then the methods in class **ReceiptManagement** will work correctly.
- After implementing the methods in class **AVL**, students compile and run the **main** method in file *Main.java*
- In YC3, YC4 and YC5, students implement class **Rating** to represent the graph in the form of an edge list, implement the methods in class **RatingQuery** to find information in the data following the requirements.
- After implementing the methods in class **RatingQuery**, students compile and run the **main** method in file *Main.java*.
- Students compare your output to the expected output in the *output* folder.
- For requirements that students can't implement, please do not remove methods that are related to those requirements and make sure that your program can run with the **main** method in the provided *Main.java* file.
- The Google Drive link will also contain a *version.txt* file. Students should usually check this file. If there are new changes, students should read the changes carefully and download

the newest file(s). This *version.txt* file is used to notify what was changed and the date when it was changed in case of error in the assignment.

IV. Product, Customer, Receipt, ReceiptManagement, Node, AVL, Rating, RatingQuery classes

- Explanation of properties and methods for these classes:
 - Class **Receipt** (students only read and understand this class and must not edit this class):
 - *receiptId*: receipt unique identifier
 - *products*: list of products identifier
 - *customerId*: customer unique identifier
 - Class **Node**:
 - *receipt*: the receipt
 - *left, right*: left, right
 - *height*: height of the node
 - Class **AVL**:
 - *root*: root node
 - **checkBalance(x: Node)**: check the balance factor of a node.
 - **balance(x: Node)**: balance the tree at one node.
 - **insert(node: Node, receipt: Receipt)**: recursively insert a node into the tree.
 - **NLR(node:Node, result: ArrayList<String>)**: recursively traverse tree by NLR order and store the result into the **result** list.
 - **search(x: Node, receiptId: int)**: recursively search for node that is storing the receipt has **receiptId**.
 - Class **ReceiptManagement** (students only read and understand this class and must not edit this class):
 - *receiptTree*: an AVL tree containing receipts
 - **readReceiptList(filePath: String)**: read list of receipts from file *receipt.txt* and build the tree from this list.
 - **traversalTree(filePath: String)**: traverse tree and write to file.
 - **searchReceipt(filePath: String, receiptId: int)**: search for receipt by **receiptId** and write to file.



- Class **Customer** (students only read and understand this class and must not edit this class):
 - *customerId*: customer unique identifier
 - *customerFullName*: customer fullname
 - *gender*: gender (“M” stands for male and “F” is for female)
- Class **Product** (students only read and understand this class and must not edit this class):
 - *productId*: product unique identifier
 - *productName*: product name
 - *productType*: product type

RatingQuery
- edges: ArrayList<Rating> - productList: ArrayList<Product> - customerList: ArrayList<Customer>
+ RatingQuery(edges: ArrayList<Rating>, productList: ArrayList<Product>, customerList: ArrayList<Customer>) + printGraph(): void + readCustomerFile(filePath: String): boolean + readProductFile(filePath: String): boolean + addEdge(u: Customer, v: Product, w: int): void + buildGraph(filePath: String): boolean + query1(customerId: String): ArrayList<Product> + query2(productId: int): Integer + query3(): ArrayList<Product> + writeFile(filePath: String, list: ArrayList<E>): boolean + writeFile(filePath: String, data: <E>): boolean

- Class **RatingQuery**:
 - *edges*: an edge list representing the graph
 - *productList*: list of products
 - *customerList*: list of customers
 - **Constructor method**: Call to the read file methods and build the graph method.
 - **printGraph()**: print the edge list.
 - **addEdge(u: Customer, v: Product, w: int)**: add an edge to the edge list with vertex *u* being a customer, vertex *v* being a product and *w* is the star that the customer has rated this product.
 - **buildGraph(filePath: String)**: build the graph from *rating.txt* file, this method is defined by default. Students just need to correctly implement the method **addEdge** then this method will work correctly.

- **query1, query2, query3:** are the methods that matches with the Requirement 3, Requirement 4 and Requirement 5.
- **writeFile methods:** methods that are used to write file. It used in the **main** method for writing the result of Requirement 3, Requirement 4 and Requirement 5.

V. Input and output files overview

- Folder *data* contains 4 input files.
- Input file *receipt.txt* contains list of receipts, one row being one receipt and the properties of one receipt is delimited by a comma “,”:

Receipt ID,List of products (delimited by an underscore “_”),Customer ID

```
1    1,[1_2],Q102
2    2,[1_4],Q902
3    3,[1_7_10],Q701
4    4,[2_3_4_9],Q702
```

- Input file *customer.txt* contains list of customers, one row being one customer and the properties of one customer is delimited by a comma “,”:

Customer ID,Customer name,Gender (“M” for male and “F” for female)

```
1    Q101,Vu Thi Hanh,F
2    Q102,Nguyen Quang Duy,M
3    Q1101,Tran Tuan Kiet,M
4    Q801,Pham Thi Nhu,F
5    Q802,Nguyen Duy An,M
```

- Input file *product.txt* contains list of products, one row being one product and the properties of one product is delimited by a comma “,”:

Product ID,Product name,Product type

```
1    1,Vinamilk,Milk
2    2,Chicken,Meat
3    3,Pork,Meat
4    4,Beef,Meat
5    5,Coca,Beverage
```

Product ID will always start with 1 and increase one by one.

- Input file *rating.txt* contains list of ratings for different products, one row being one rating and the properties of one rating is delimited by a comma “,”:

Customer ID,Product ID,number of stars rated

```
1 Q102,1,3
2 Q102,2,4
3 Q902,1,5
4 Q902,4,1
5 Q701,1,2
```

For example: The first line is customer with ID Q102 rates 3 stars for the product with ID 1.

- Output folder contains 5 files with expected result for each requirements:
 - o *Req1.txt*: result of traversing the AVL tree with the traversal order NLR.
 - o *Req2.txt*: result of search for receipt with productId is 3 on the AVL tree.
 - o *Req3.txt*: result of products that are rated with 3 stars upper by customer with ID Q601.
 - o *Req4.txt*: result of the count of male customers that rated product with ID 1 with 3 stars upper.
 - o *Req5.txt*: result of products that are rated by greater than or equal to 50% female customers.

Lưu ý:

- Students can add more data to the input file to test more cases but remember that the added data must follow the format that are defined above.
- Students should carefully read the **main** method to correctly define the order to implement classes and methods.
- Students can add methods to serve the requirements but make sure that your program can still run with the default provided **main** method.
- Students can add code to the **main** method to test your implementation but make sure that your program **can run on the default provided main method**.
- Students **must not** in any circumstances edit **the class name and method name** that are defined initially (the naming always follow the class diagram above).
- Students do not make changes to files that are not required to be submit.

VI. Run the main method

- The provided **main** method in the *Main.java* file creates objects and calls to check requirements implementation.

- To check requirements implementation, students must correctly define all the classes follow the requirements, after successful compilation, students calls by the format below to get the result of the selected requirement:

java Main X Y

- YC1: X = 1, Y = 1
- YC2: X = 1, Y = 2
- YC3: X = 2, Y = 1
- YC4: X = 2, Y = 2
- YC5: X = 2, Y = 3

For example: Students want to check Requirement 1, students input *java Main 1 1*, the result will be written to the *Req1.txt* file.

- After checking successfully with the **main** method, students can check other requirements by changing the passed arguments.

VII. Requirement

Students are not allowed to use any external API.

Students must do this assignment on Java 8.

1. REQUIREMENT 1 (3 POINTS)

Students implement class **Node** according to class diagram above and implement the methods in class **AVL**. The AVL tree build based on the **receiptId** of the **Receipt**.

Students implement method **private Node insert(Node node, Receipt receipt)** to recursively insert a receipt into the AVL tree. Remember that students need to balance the tree whenever a new node is inserted into the tree. (Students need to define methods **rotateLeft**, **rotateRight** and complete the **balance** method).

NLR traversal method and read *receipt.txt* file to get list of receipts to add to the tree method are defined by default and called in the class **ReceiptManagement**.

After successfully define the **insert** method, students compile and check this requirement with the provided **main** method by calling the command *java Main 1 1* to output the file *Req1.txt*. Students compare this file to the expected output file *Req1.txt* in the *output* folder.

Note: This is the requirement students must define for Requirement 2 to be scored.

2. REQUIREMENT 2 (1 POINT)

In class **AVL**, students implement the method:

private Node search(Node x, int receiptId)

to recursively search for a node contains the receipt with **receiptId**.

Class **ReceiptManagement** has the call to the **search** method by default. Students compile and check this requirement with the provided **main** method by calling the command *java Main 1 2* to output the file *Req2.txt*. Students compare this file to the expected output file *Req2.txt* in the *output* folder.

3. REQUIREMENT 3 (3 POINTS)

To implement Requirement 3, 4, and 5, students need to build a two-side graph, undirected, weighted based on the file *rating.txt*. The graph vertices will be customers and products, edge with weight will be the number of stars that the customer has rated the product.

Example:

With these rows of data:

Q102,1,3

Q102,2,4

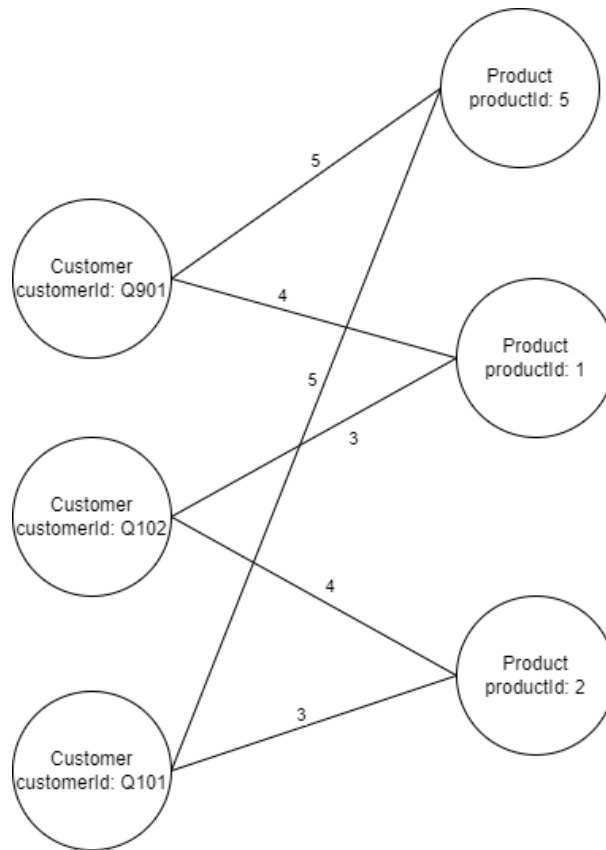
Q101,2,3

Q901,1,4

Q101,5,5

Q901,5,5

We will be able to build this graph:



To store the above graph, students will use an edge list with each edge object having the vertex u is Customer, vertex v is Product and weight w is the number of stars.

Students build the graph with these steps:

1. Create file **Rating.java** to implement class **Rating**. Class **Rating** will contain 3 properties being Customer, Product and stars number. Each Rating object will be a representation of two vertices that are connected by an edge.
2. In the file *RatingQuery.java*, students implement method **public void addEdge(Customer u, Product v, int w)** to add an edge to the list **edges**.
3. In class **RatingQuery**, the method **public boolean buildGraph(String filePath)** is provided by default to read the *rating.txt* file. If students correctly define the method **addEdge** then the method **buildGraph** will be able to create the edge list to represent the graph.
4. Students have the edge list **edges** to implement these requirements.

Implement method:

```
public ArrayList<Product> query1(String customerId)
```

to return a list of products that customer with **customerId** rated with 3 stars or more.

After implementing, students compile and check this requirement with the provided **main** method by calling the command *java Main 2 1* to output the file *Req3.txt*. Students compare this file to the expected output file *Req3.txt* in the *output* folder.

4. REQUIREMENT 4 (2 POINTS)

Implement method:

public Integer query2(int productId)

to return the count of male customers that rated product with **productId** with 3 stars or more.

After implementing, students compile and check this requirement with the provided **main** method by calling the command *java Main 2 2* to output the file *Req4.txt*. Students compare this file to the expected output file *Req4.txt* in the *output* folder.

5. REQUIREMENT 5 (1 POINT)

Implement method:

























public ArrayList<Product> query3()

to return the list of products that are rated by greater than or equal to 50% female customers.

After implementing, students compile and check this requirement with the provided **main** method by calling the command *java Main 2 3* to output the file *Req5.txt*. Students compare this file to the expected output file *Req5.txt* in the *output* folder.

VIII. Check carefully before submitting

- If students can't implement some of the requirements please leave the methods related to those requirements intact. Students **MUST NOT DELETE THE METHOD(S) OF THE REQUIREMENTS** which will lead to error when running the **main** method. Before submitting, students must check that the program are able to run with the provided **main** method.
- All of the files *ReqX.txt* ($X = \{1,2,3,4,5\}$) are written in the same level as the source code directory. For students that use IDE (Eclipse, Netbean, ...) make sure that the program can be run with commands in command prompt, make sure that the program does not exist inside a package, the position for written file *ReqX.txt* must be in the same level as the source code directory..
- Correct level of written file when running the program will follow this image:

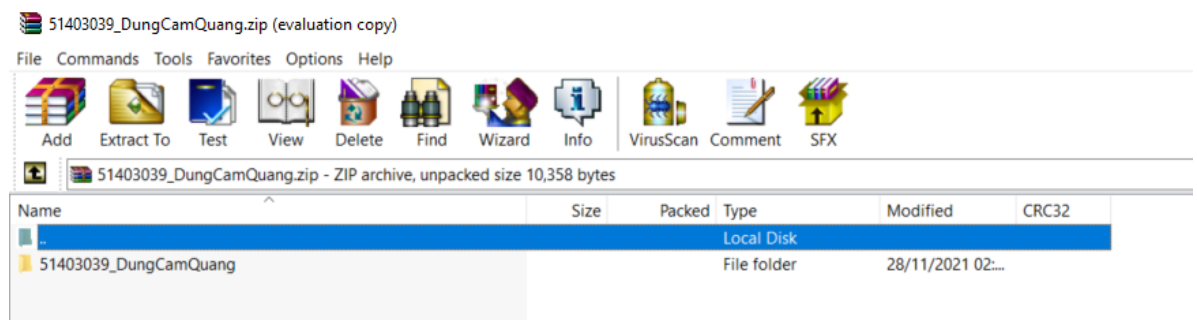
 data	22/11/2021 22:49	File folder	
 AVL.class	23/11/2021 03:22	CLASS File	3 KB
 AVL.java	22/11/2021 03:35	JAVA File	3 KB
 Customer.class	23/11/2021 03:22	CLASS File	2 KB
 Customer.java	23/11/2021 02:31	JAVA File	2 KB
 Main.class	23/11/2021 03:22	CLASS File	2 KB
 Main.java	23/11/2021 03:22	JAVA File	2 KB
 Node.class	23/11/2021 03:22	CLASS File	2 KB
 Node.java	21/11/2021 02:44	JAVA File	1 KB
 Product.class	23/11/2021 03:22	CLASS File	2 KB
 Product.java	23/11/2021 02:46	JAVA File	2 KB
 Rating.class	23/11/2021 03:22	CLASS File	2 KB
 Rating.java	23/11/2021 02:24	JAVA File	1 KB
 RatingQuery.class	23/11/2021 03:22	CLASS File	6 KB
 RatingQuery.java	23/11/2021 03:08	JAVA File	6 KB
 Receipt.class	23/11/2021 03:22	CLASS File	2 KB
 Receipt.java	22/11/2021 03:28	JAVA File	2 KB
 ReceiptManagement.class	23/11/2021 03:22	CLASS File	4 KB
 ReceiptManagement.java	22/11/2021 03:35	JAVA File	3 KB
 Req1.txt	23/11/2021 03:22	Text Document	2 KB
 Req2.txt	23/11/2021 03:22	Text Document	1 KB
 Req3.txt	23/11/2021 03:22	Text Document	1 KB
 Req4.txt	23/11/2021 03:22	Text Document	1 KB
 Req5.txt	23/11/2021 03:23	Text Document	1 KB

IX. Submission guideline

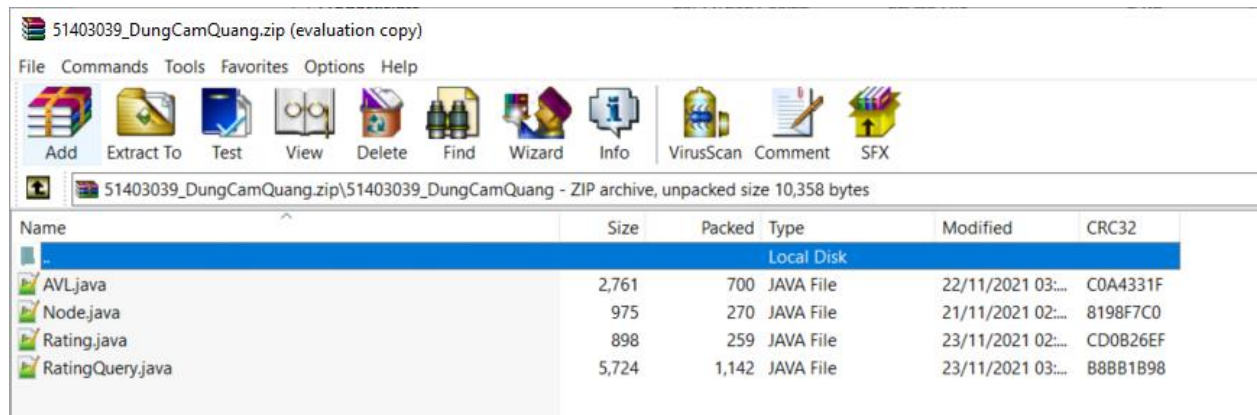
- When submitting, students submit file *Node.java*, *AVL.java*, *Rating.java* and *RatingQuery.java*, **do not include any other files and must not edit the name of these 4 files.**
- **Students place these 4 files into the folder StudentID_FullName** (Full name is written with no spaces, no signs) compressed with the **.zip** format and submit by the guidance of the practical lecturer.
- In case of wrong submission (wrong folder naming, not placing the files into the folder, more files than required, ...) then students will get **0 point**.
- Correct files submission will be:
 - o Compressed file:

 51403039_DungCamQuang.zip 28/11/2021 02:47 WinRAR ZIP archive 4 KB

- o Inside the compressed file:



- o Inside the folder:



X. Scoring and regulation

- Your implementation will be scored automatically through testcases (input file and output file will have the format as described above) so students will be responsible for not abiding to the submission guideline or changing the method name which lead to the program not be able to compile.
- Testcases which are used to score the program are files that have the same format as described with different content to the input files provided to students. Students will only receive points for the Requirement if your implementation output is exact completely.
- If the compilation of students implementation result in an error, you will receive **0 point** for all requirements.
- **All of your code will be checked for plagiarism. Any behaviors of copying code from the Internet, copying your friends code or allowing your friends to copy your code if detected you will receive 0 point for the whole Process 2 point or not being able to participate in the final exam.**
- If students implementation have sign of copying code from the Internet or copying each other, students will be call for a code interview to prove that the implementation is indeed their.
- **Deadline for submission: 23h00, December 18 2021.**

-- THE END --