Rutgers Warehouse – 100 course points

The purpose of this assignment is to practice your understanding of the priority queue and hash table data structures.

Start your assignment early! You need time to understand the assignment and to answer the many questions that will arise as you read the description and the code provided.

Refer to our Programming Assignments FAQ for instructions on how to install VSCode, how to use the command line and how to submit your assignments.

Overview

Congratulations, you've just won the lottery and decided to use the money to start a Rutgers merch store! You have some warehouse space to store your inventory, but you are having trouble keeping track of what items you currently have in stock. Your warehouse can store up to 50 different product types, and being a new store, you will be frequently adding new products. You need a structure that can efficiently look up products and delete underperforming ones to make space for new items.

Design

- You settle on a Hash Table like structure, where each entry of the table stores a priority queue. Due to your limited space, you are unable to simply rehash to get more space. However, you can use your priority queue structure to delete less popular items and keep the space constant.
- Your warehouse is split into 10 sectors, each of which can hold 5 product types.
- The last digit of the product ID determines which sector it should go into. Then when we search for it, we can

immediately narrow down the search to at most 5 items rather than searching through the entire warehouse.
- You settle on a simple metric for how well an item is doing: *Popularity = Initial Demand + Total Amount Purchased + Date of Last Purchase*. The initial demand is obtained from a survey prior to product release, and the date of last purchase is simply the number of days since the store opening that the item was last purchased.
- You want to be able to delete the least popular item in a sector efficiently, so you decide to make each sector a min heap ranked by popularity.
- Even though each sector is a min heap, it can still function as a normal list, which you can search through sequentially.

Overview of Files

- Product.java
    - Contains the product ID, name, stock, date of last purchase, demand, and overall popularity. The demand is simply the sum of the initial demand for the product and the number purchased.
    - Getters and setters are provided, as well as special methods to update the stock and demand by some positive or negative amount.
    - You cannot manually set the popularity, but it is automatically calculated and updated as the sum of the last purchase day and current demand.
    - **DO NOT** edit this file.
- Sector.java
    - Contains an array of Products which forms a valid min heap based on popularities, as well as the current size of the sector (defined as the number of Products in it).
    - On Products array we ignore index 0 just like in class. Then at any moment, the indices containing valid Products range from 1 to currentSize inclusive.
    - Contains helpful methods to add a Product to the end of the sector, set some index, delete the last Product, get the Product at some index and get the current size.
    - The Sector class alone isn't a fully implemented min-heap. However, you're provided a method to swap the Products at 2 indices, a method to apply the swim algorithm from class on some index, and a method to apply the sink algorithm from class on some index.
    - **DO NOT** edit this file.
- Warehouse.java

    - Contains an array of 10 Sectors, with index i representing sector i.
    - Contains methods to fill in for every sub-problem in the assignment.
    - **DO** edit the empty methods in this file.
    - **DO NOT** edit the provided methods in this file.
    - **This file will be submitted for grading.**

- StdIn.java

    - Use StdIn.setFile(filename) to set the current input file you want to read from.
    - You can now use StdIn.readInt() and StdIn.readString() to operate on the file as if it were standard input. These methods ignore whitespace.

- StdOut.java

    - Use StdOut.setFile(filename) to set the current output file you want to write to.
    - You can now use StdOut.println() to operate on the file as if it were standard output.

- Method files

    - The structure of this assignment is significantly different from previous assignments.
    - Rather than being provided with a driver, you are provided with an empty java file for each graded method.
    - **DO** fill in the main methods in these files.
    - Each class is expected to take 2 command line arguments, an input file and an output file.
    - Use StdIn and StdOut to read from the input file and write the solution to the output file.
    - These files will not be graded, but they are necessary for testing of your methods.

- Input files

    - Each subtask has a corresponding .in file to help you test the method.
    - Feel free to write your own, as long as you adhere to the formats provided

Implementation Notes

- **DO NOT** use static variables on your code.
- **DO NOT** change the names of any of the given Java files, or the project structure itself (do not change directory names or create new directories).
- **DO NOT** remove the package statement from any of the given files.

- **DO NOT** use System.exit() in your code.
- **DO NOT** add any new files.

Tasks

1. **addProduct**
    - Notice that addProduct in the Warehouse class has been implemented in 3 pieces, and you are responsible for filling in each of the 3 pieces.
    - The method is designed to be incrementally tested. As you fill in each piece, the method takes on additional behavior.
    - The code you write in AddProduct.java to test the first piece **will work unchanged** to test your method after the second and then the third pieces are implemented. We provide multiple input files for you to test each of the 3 pieces of the addProduct() method.
    - **ONLY** the final addProduct method with all 3 parts implemented will be graded.

    a. **addToEnd**
        - Write a method in your Warehouse class that takes in a new product id, name, stock, day and initial demand, and adds a new Product object to the end of the correct sector.
        - The sector index you add to should be the last digit of the given product id.
        - The initial date of last purchase of your new Product should be set to the current day (which is passed in).
        - The input file will be formatted as follows:
            - An integer **n** representing the number of products to add
            - **n** lines, each containing the following in this order (space separated):
                - The current day
                - The product ID
                - The product name (Guaranteed to not contain spaces)
                - The initial item stock
                - The initial item demand
        - Fill in the AddProduct.java file to read from args[0] and write to args[1]. Create a new Warehouse object, then add each Product from the input file to your warehouse using the **addProduct()** method (and NOT the addToEnd() method. This will help with testing later). Finally, you can simply print out your Warehouse object to your output file. For

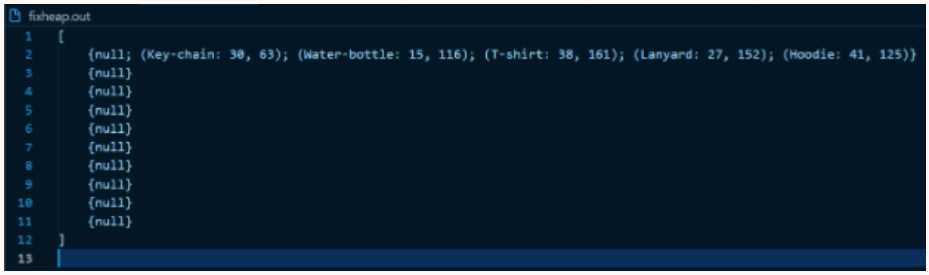
example, if your Warehouse object is named w, call StdOut.println(w).
- The output will be a text representation of your warehouse, showing the Products in each sector, specifically their names, stocks and popularities.
- Here is the correct “addtoend.out” file obtained from running the AddProduct.java file with the command line arguments “addtoend.in” and “addtoend.out” in that order.

addtoend.out

```
[
    {null; (Hoodie: 15, 6)}
    {null}
    {null}
    {null; (Water-bottle: 41, 12)}
    {null}
    {null; (Pants: 3, 19); (Notebook: 24, 24); (Flag: 47, 20)}
    {null}
    {null; (Lanyard: 23, 21)}
    {null; (T-shirt: 21, 9)}
    {null; (Key-chain: 6, 22); (Sunglasses: 8, 21); (Stickers: 19, 34)}
]
```

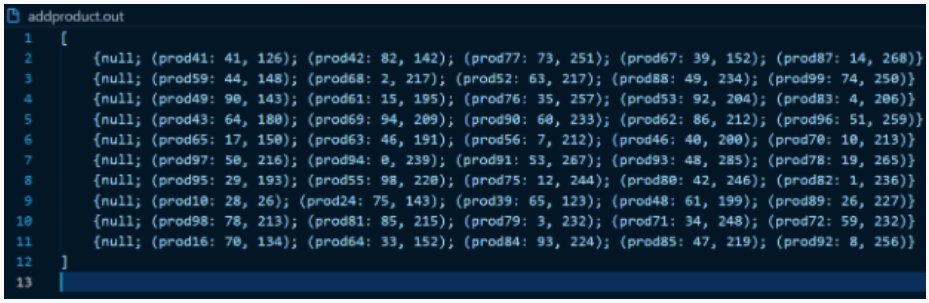
b. **fixHeap**

- Your current addProduct is fine, but as of now it just appends to the end of the sector. We want to maintain a min-heap structure based on popularity at all times.
- Write a method in your Warehouse class that takes in the id of a product which was just added to the end of its Sector, and fixes the heap structure of that Sector.
- Look into the Sector class to see what methods are provided to you. You do NOT have to implement all the heap operations from scratch.
- fixHeap does NOT call the addToEnd() method. It is assumed that the method has already been called, as can be seen in the template method for addProduct().
- The input file format is exactly the same as addToEnd.
- You do not have to change the AddProduct.java class in any way to test your updated addProduct() method.
- The output file format is exactly the same as addToEnd.
- Here is the correct “fixheap.out” file obtained from running the AddProduct.java file with the command line arguments “fixheap.in” and “fixheap.out” in that order.



c. **evictIfNeeded**

- Your current addProduct() works fine until one of the sectors goes over capacity. We want to delete the least popular element when we are trying to add in a new one, so we can continue adding new products.
- Write a method in your Warehouse class that takes in the id of a product we want to add (hasn't been added yet), and makes room for it in the correct Sector. It will implement our deleteMin() algorithm from class which deletes from the min heap.
- The method **ONLY** performs this operation if necessary. In other words, it does nothing **UNLESS** the sector we want to insert into is currently at full capacity (has 5 products already).
- The popularity of the item to be inserted is irrelevant. This method still removes the least popular item in a full capacity sector, even if we're about to insert an even less popular new item.
- Look into the Sector class to see what methods are provided to you. You do NOT have to implement all the heap operations from scratch.
- The input file format is exactly the same as addToEnd.
- You do not have to change the AddProduct.java class in any way to test your updated addProduct() method.
- The output file format is exactly the same as addToEnd.
- Here is the correct "addproduct.out" file obtained from running the AddProduct.java file with the command line arguments "addproduct.in" and "addproduct.out" in that order.



2. **restockProduct**
    - Write a method in your Warehouse class that takes in a product ID and some amount to restock, and updates the stock of that item in the Warehouse accordingly. If the item does not exist in the Warehouse it does nothing.
    - This method does NOT affect the popularity of the item, and thus does not move around the products at all.
    - To test this method all 3 pieces of the addProduct() method are expected to have been implemented.
    - The input file is formatted as follows:
        - An integer n representing the number of queries
        - n lines, each containing either an add query or a restock query
        - Add queries:
            - An add query will start with the word “add”
            - It will then contain the following (space separated):
                - The current day
                - The product ID
                - The product name (Guaranteed to not contain spaces)
                - The initial item stock
                - The initial item demand
            - Add queries represent a new product to add to your warehouse
        - Restock queries:
            - A restock query will start with the word “restock”
            - It will then contain the following (space separated):
                - The product ID to restock
                - The amount to restock
            - Restock queries tell you to update the stock of some item
    - Fill in the Restock.java file to read from args[0] and write to args[1]. Create a new Warehouse object, then operate on your Warehouse object responding to each query. Finally, you can simply print out your Warehouse object to your output file. For example, if your Warehouse object is named w, call StdOut.println(w).

- The output will be a text representation of your warehouse, showing the Products in each sector, specifically their names, stocks and popularities.
- Here is the correct “restock.out” file obtained from running the Restock.java file with the command line arguments “restock.in” and “restock.out” in that order.

3. **deleteProduct**
   - Write a method in your Warehouse class that takes in a product ID, then removes that product from your Warehouse. The product will not necessarily be the least popular item in its sector. If the item does not exist in the Warehouse, do nothing.
   - While you may have to iterate through a sector to find the item, once it is found you must delete it in O(logn) time by first swapping it with the last element in the sector, reducing the sector size, and then fixing the heap accordingly.
   - To test this method, all 3 pieces of the addProduct() method are expected to have been implemented.
   - The input file is formatted as follows:
     - An integer n representing the number of queries
     - n lines, each containing either an add query or a delete query
     - Add queries are identical to the ones from Restock.
     - Delete queries:
       - A delete query will start with the word “delete”
       - It will then contain the following (space separated):
         - The product ID to delete
       - Delete queries tell you which product ID to delete.
   - Fill in the DeleteProduct.java file to read from args[0] and write to args[1]. Create a new Warehouse object,

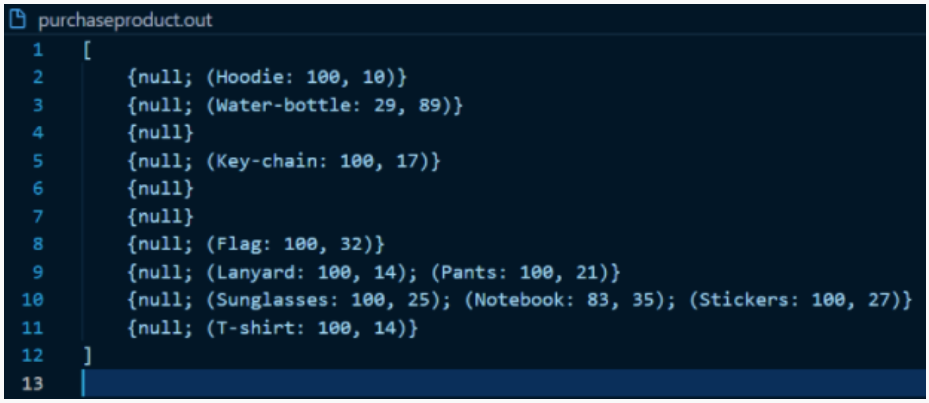
then operate on your Warehouse object responding to each query. Finally, you can simply print out your Warehouse object to your output file. For example, if your Warehouse object is named w, call StdOut.println(w).
- The output will be a text representation of your warehouse, showing the Products in each sector, specifically their names, stocks and popularities.
- Here is the correct “deleteproduct.out” file obtained from running the DeleteProduct.java file with the command line arguments “deleteproduct.in” and “deleteproduct.out” in that order.

deleteproduct.out

```
[
    {null}
    {null; (Lanyard: 13, 14); (Flag: 24, 21)}
    {null; (Notebook: 48, 23)}
    {null; (T-shirt: 36, 19)}
    {null}
    {null; (Pants: 39, 15)}
    {null; (Key-chain: 0, 19)}
    {null}
    {null; (Stickers: 29, 30)}
    {null}
]
```

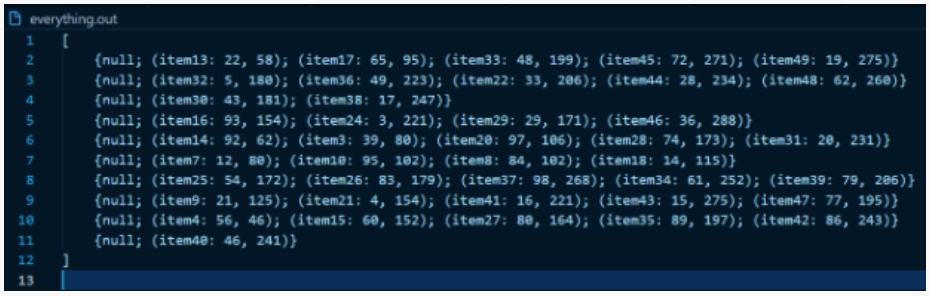
4. **purchaseProduct**
   - Write a method in your Warehouse class that takes in an ID, a day of purchase, and some amount purchased, then simulates the purchase order.
   - When an item is purchased, its last purchase day is updated to the current day, its stock is decreased by the amount purchased, and its demand is increased by the amount purchased. **Remember to maintain the min-heap structure based on popularity!**
   - If the item represented by the given ID doesn’t exist, do nothing.
   - If the purchase amount is greater than the amount on stock, do nothing.
   - To test this method, all 3 pieces of the addProduct() method are expected to have been implemented.
   - The input file is formatted as follows:
     - An integer n representing the number of queries
     - n lines, each containing either an add query or a purchase query
     - Add queries are identical to the ones from Restock.

        - Purchase queries:
            - A purchase query will start with the word “purchase”
            - It will then contain the following (space separated):
                - The current day
                - The product ID to purchase
                - The amount purchased
            - Purchase queries give you some purchased item and how many were purchased on what day.
    - Fill in the PurchaseProduct.java file to read from args[0] and write to args[1]. Create a new Warehouse object, then operate on your Warehouse object responding to each query. Finally, you can simply print out your Warehouse object to your output file. For example, if your Warehouse object is named w, call StdOut.println(w).
    - The output will be a text representation of your warehouse, showing the Products in each sector, specifically their names, stocks and popularities.
    - Here is an example of a correct “purchaseproduct.out” file obtained from running the PurchaseProduct.java file with the command line arguments “purchaseproduct.in” and “purchaseproduct.out” in that order.



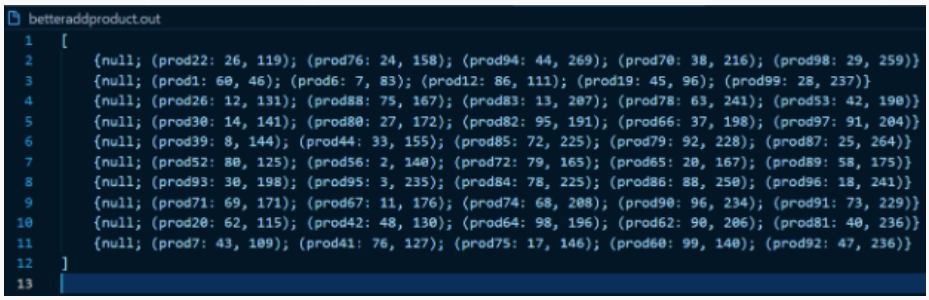
5. **Putting it all together**
    - Have all previous parts working to test this method.
    - DO NOT have to change any code in your Warehouse class to get the correct output. You are simply writing a main method which puts all the previous steps together and answers all types of queries at once.
    - The input file is formatted as follows:
        - An integer n representing the number of queries
        - n lines, each containing either an add, restock, purchase, or delete query
        - Add queries are identical to the ones from Restock.

        - Restock queries are identical to the ones from Restock.
        - Purchase queries are identical to the ones from PurchaseProduct
        - Delete queries are identical to the ones from DeleteProduct
    - Fill in the Everything.java file to read from args[0] and write to args[1]. Create a new Warehouse object, then operate on your Warehouse object responding to each query. Finally, you can simply print out your Warehouse object to your output file. For example, if your Warehouse object is named w, call StdOut.println(w).
    - The output will be a text representation of your warehouse, showing the Products in each sector, specifically their names, stocks and popularities.
    - Here is an example of a correct “everything.out” file obtained from running the Everything.java file with the command line arguments “everything.in” and “everything.out” in that order.



6. **betterAddProduct (20 points extra credit)**
    - Write a method in your Warehouse class to further optimize addProduct.
    - As of now, it’s possible that an item is removed from a full sector to make room for a new product, even if there are other sectors which are not full.
    - If the current sector is not full, add the product as normal.
    - Otherwise perform a linear probing-like operation to try to find a non-full sector (if the current one is full). Keep incrementing the sector until you either find one with space, or you return to your original sector. If you get to Sector 9, wrap around to Sector 0.
    - If you found a new sector with space, add the product into this sector. In a real-world scenario we would have to make sure to change the ID in our system and make sure it doesn’t conflict with an existing one. In this assignment that is not necessary since the output only contains the name.
    - If you returned to the original sector, perform eviction and add the product as normal.
    - The input file is formatted in exactly the same way as in AddProduct.

- Fill in the BetterAddProduct.java file to read from args[0] and write to args[1]. It should be identical to AddProduct.java, but call betterAddProduct() instead of addProduct().
- The output will be a text representation of your warehouse, showing the Products in each sector, specifically their names, stocks and popularities.
- Here is an example of a correct "betteraddproduct.out" file obtained from running the BetterAddProduct.java file with the command line arguments "betteraddproduct.in" and "betteraddproduct.out" in that order.



VSCode Extensions

You can install VSCode extension packs for Java. Take a look at this tutorial. We suggest:

- Extension Pack for Java
- Project Manager for Java
- Debugger for Java

Importing VSCode Project

1. Download the zip file from Autolab Attachments.
2. Unzip the file by double clicking it.
3. Open VSCode
    - Import the folder **RUWarehouse** to a workspace through **File** > **Open Folder**

Executing and Debugging

- You can run your program through VSCode or you can use the Terminal to compile and execute. We suggest running through VSCode because it will give you the option to debug.
- How to debug your code
- If you choose the Terminal, from PreReqChecker directory/folder:
    - to compile: **javac -d bin src/warehouse/*.java**
    - to execute addProduct: **java -cp bin warehouse.AddProduct addproduct.in addproduct.out**

Before submission

Collaboration policy. Read our collaboration policy here.

Submitting the assignment. Submit *Warehouse.java* separately via the web submission system called Autolab. To do this, click the *Assignments* link from the course website; click the *Submit* link for that assignment.

Getting help

If anything is unclear, don't hesitate to drop by office hours or post a question on Piazza.

- Find instructors and head TAs office hours *here*
- Find tutors office hours on Canvas -> Tutoring -> RU CATS
- In addition to office hours we have the CAVE (Collaborative Academic Versatile Environment), a community space staffed with lab assistants which are undergraduate students further along the CS major to answer questions.

Problem by Ishaan Ivaturi