

*#6*

# CPT 187—OBJECT-ORIENTED LOGIC & DESIGN

Spring-Program #6

**PROGRAM DESCRIPTION**

You’re going to implement two common algorithms for searching arrays: sequential search and binary search. The array you will search will be very similar to the discussion in the class/tutorial: an array of inventory part numbers loaded from a sequential data file. The purpose for the search will be to determine what the price is for any particular part in the inventory. To support your data requirements, I have included a sorted list of inventory ID numbers and associated prices in one file, *inventoryPricing.dat*. Each record in the file contains two fields: an integer representing one inventory ID number; and a double representing that item’s price. The records are sorted by ID number, the first field. You’ll implement each search algorithm in a separate method and (optionally) see how many comparisons are needed to find, or not find, each part the user searches for.

## **SPECIFIC DIRECTIONS**

Develop a class named **InventoryData** that will be used to analyze just that: certain data about every part in the company inventory. Here is a more formal description of the records in the file *inventoryPricing.dat*:

**RECORD DESCRIPTION (inventoryPricing.dat):**

|  |  |
| --- | --- |
| partNum | int |
| price | double |

The class needs to contain the following methods to manage the data.

InventoryData The constructor. The filename should be passed to the constructor from the MainClass. The constructor will save the filename in an instance variable to be used in the loadArrays method.

loadArrays Load the arrays from the inventory file. The file will have no more than 200 records.

seqSearch This method must follow the sequential search flowchart logic covered in class or in the tutorial for online students. Do not copy a search from some other reference or use a library module. The method needs to return the following answer: if the part is found, return the subscript number where it was found in the array; if not found, return the value -1.

binSearch This method must follow the binary search logic presented in class. Again, do not copy a search from another reference or use a library module. The method needs to return the same type of information as the sequential search: if the part is found, return its subscript number; if not found, return the value -1.

If you want to incorporate other methods, that’s fine as long they make logical sense to include inside the class (remember, that’s always gradable!). You can simply make all methods public and call them from the main method. Hint: you will definitely need at least one other method not listed here.

Use main as the driver to make sure all of your logic works. First, call loadArrays to get the data loaded into the appropriate arrays, one for each of the two types of data. Then, start asking the user for part numbers to search for. For each part number the user is looking for, search the part number array using both search methods, one at a time, and report whether the part was found after each search. If the part was found, also report the price for that part, which of course is easily accessible in the other array. Display these answers in main, not in the search methods—that’s not their job! All output should be in the MainClass.

The user must be able to search for as many ID numbers as desired. For example, the program might run all day in a Parks Auto Parts™ store, so don’t stop asking for part numbers until the user wants to shut down at the end of the day. Be sure to keep reminding the store clerk how to quit the program: at the end of the day they are not going to remember your instructions from when they started your program early in the morning!

Be careful to report the search results separately for each of the two search algorithms, even if the part was not found! (This is so that you can double-check that both algorithms are working correctly and reporting the same answers in all cases.) For example, say I request part number 28751. You should have two separate results:

Sequential found part number 28751, and its price is $9.27.

Binary found part number 28751, and its price is $9.27.

However, if the part is not in the file, then you should have two independent results declaring so, perhaps like this:

Sequential did not find part number 28751.

Binary did not find part number 28751.

Both searches must report the same results no matter what they are!

Do not quit until the user is done searching for part numbers. But, you don’t need to accumulate any data, only report search results from both search methods.

## As always, be sure that you have named the project, package, and .Java files correctly. See the Coding Standards handout for the required names.

## **TURN IN**

A complete Class Description and a zipped copy of the complete project directory.

**Extra Credit** (5 pts): incorporate a counter in each search method to determine how many array comparisons were needed to find the number’s location, or how many comparisons were needed to determine that it wasn’t there. By this I mean the comparisons we were using in class to analyze the efficiency of each algorithm. This will give you a gross count of how much work was required, and sort of support the Big O analysis we discussed. (For the extra credit only, you may display the number of comparisons inside each of the search methods, rather than trying to return it.)