CURTIN UNIVERSITY (CRICOS number: 00301J)
Faculty of Science and Engineering
Department of Computing
Data Structures and Algorithms 120 (Index No. 1922)

Assignment, Semester 2, 2013

1. Overview and Background

You will be provided with a large file containing thousands of records representing a warehouse's inventory of current stock. The records are not currently in any particular order. Each record is identified by a unique key comprising of the category (table, chair, computer, etc) and a sequence number, eg: Computer:14. Each record contains both the key and information associated with the item (type, price, etc – see section 2.1 for details). Your task is to write a program that can load the records into memory and allow users to look up and display records via the item key (since the user knows the record key and wants to find out the associated information).

Since this is a unit on data structures, you will be required to implement two search methods based on two data structures. Initially, you are to load the records into an array, and the user may then request to search the array using a linear O(N) search. The user must also be able to build a binary search tree from the array and subsequently choose to search using the O(log N) find of a binary search tree. Finally, the user must be able to store the data to a file in key-sorted order.

When the program starts up, it should show the following main menu choices:

```
(1) – Load records into array
(2) – Build binary search tree from array
(3) – Search array
(4) – Search tree
(5) – Save data to file
(6) – Quit
Choice:>
```

A major aspect of this assignment is that your program must be bulletproof – crashes, ungraceful behaviour when given bad data, premature exits and infinite recursions are bugs that must be eliminated or you will lose marks. Hence testing is essential.

2. Specifications

The system should loop, requesting the user enter in their choice at which point the system executes the desired option (sometimes involving further user input) and loops back to re-display the main menu. If the user chooses to Quit the program should exit. Detailed specifications for the behaviour of each of the above menu options follows.

2.1 Load Records into Array

On choosing Option 1, the user must be asked to enter in the path to the data file containing the records. Two example data files will be provided to you, a smaller one containing 1,000 records and a larger one containing a million records. However, although these files do not contain errors your code must still handle the possibility that the data gets corrupted and so must check for errors in formatting (see Input Validation below). Data files are in CSV format and each record is formatted as follows:

<Key>,<Brand>,<Model>,<WeightInKg>,<Price>

- Key – Structured as Category:ID, where ID is a sequence number. There is no need (or use) to split this into its two pieces, just treat the key as a single unique value
- Brand – Manufacturer of the product
- Model – Model name of the product
- WeightInKg – Weight of the product, rounded to the nearest kg
- Price – Selling price of the product to 2 decimal places (2dp)

- *Input validation:* If the path and/or files do not exist, an error should be shown to the user and the system should return to the main menu.
- *Input validation:* WeightInKg should be an integer.
- *Input validation:* Price should be a double. You may choose to either round it to 2dp *or* reject the value completely if the user enters in a value that is 3dp or greater precision.
- Ensure you close the file once finished.

DO NOT SORT THE ARRAY YET. It is important that you leave it in unsorted order for the binary search tree build (option 2). *If you sort it, the tree will be degenerate* (as it would be built from a sorted list) and its search performance will be awful.

HINTS:
- Create a class to store a single WarehouseItem. Then create an array of these objects to store the full list of these items.

- You may assume that will never be more than 1,000,000 records.
- Don’t forget to keep track of the count of items as you are loading them: it is different to the capacity of the array!
- In order to run the program for 1,000,000 items you will need to increase the size of memory that Java can use. To do this, run java with the –Xmx option, ie:

  ```
  $ java –Xmx1024M MainClass
  ```

2.2 Build Binary Search Tree from Array

On selecting this option, the program must build a binary search tree of the data by inserting the items from the array one at a time into a new binary search tree.

- Keys in the provided datasets are unique hence they will not cause duplicates. However your code must handle the possibility that non-unique keys exist (eg: corrupted data file).
- The user should be able to choose this option multiple times and have it rebuild to binary search tree from scratch rather than append to an existing tree (the latter would fail with duplicate key errors).

2.3 Search Array (Linear Search)

On selecting this option, the program must ask the user for a key (CategoryName:ID) to search for. The program must then perform a linear search through the array looking for the record containing that key. If found, the Warehouse record should be displayed to the user; if not found an appropriate error message should be displayed to the user. In both cases, the time taken to perform the search (in milliseconds) must be displayed.

Ensure the record is displayed in a neat format with each field name indicated. Weight must have ‘kg’ appended, and price have ‘$’ prefixed. Display each field on its own line, ie:

  - Key: XXXXXXX
  - Brand: XXXXXXX
  - Model: XXXXXXX
  - Weight: XX kg
  - Price: $XXX.XX

- Ensure the values (XXXX above) line up by padding spaces after the field names. Alternatively, you can use string formatting to do this for you (eg: to print the a string in a field 10 characters wide, use sPaddedStr = String.format(“%25s”, sStr); – extra chars will be spaces – see the JavaDocs on String.format for more information).
- Search must be case-SENSITIVE.
- *Validation:* The array must already be loaded.

HINTS:

- To calculate the time taken to perform a search, use System.nanoTime(). This returns the current system time in nanoseconds, returning a ‘long’ variable (a ‘long’ is a 64-bit int).
- So call System.nanoTime() just before you start the search (but *after* the user enters the search key) and store this value. Then immediately after the search completes, call System.nanoTime() again. The difference in the two values is the search time.
- To convert nanoseconds to milliseconds, divide by 1000.0 (as a double). **Display milliseconds to the full 3 decimal places** (3dp)

2.4 Search Tree

On selecting this option, the program must ask the user for a key (CategoryName:ID) to search for. The program must then perform a search through the *tree* looking for the record containing that key. If found, the Warehouse record should be displayed to the user; if not found an appropriate error message should be displayed to the user. In both cases, the time taken to perform the search (in milliseconds) must be displayed.

- As with searching the array, ensure the record is displayed in the same neat format.
- It would thus be a good idea to make a separate function that displays a given record!
- Note that the tree will be roughly balanced rather than degenerate, since I have ensured that the provided datasets are randomly ordered. Hence search times will be O(log N) and you will *not* need to implement complex rebalancing algorithms like Red-Black trees.
- Do not forget to display the time taken to do the search. You should find that the search of the tree is *much* faster than the linear search of the array (especially with the million-item dataset).
- *Validation:* The tree must already be built.

2.5 Save Data to File

When selecting this option, the user is asked for the name of the file to save the data. This must save data to a file in comma-separated format *and in key-sorted ascending order*. You have two options on how to do this:

1. Sort the array before saving it. You must use an O(N log N) algorithm to do this. An O(N^2) algorithm will only get you half marks.
2. Perform an in-order traversal of the binary search tree (which will visit nodes in ascending key order) and write out the records in the order visited.

- *Validation:* The data must already be loaded.

2.6 REPORT

You are also required to submit a small ‘report’ (this may be submitted electronically). You must also provide a *printed and signed* assignment cover sheet (see unit pages for the cover sheet).

The report is to have the following parts to it:

- Simple UML class diagram showing all of your classes, their fields (names only), their methods (names only) and (optionally) the relationships between classes. Do not show Java’s standard library classes since this will clutter up your diagram.
- A list of known defects in your application.

3. Marks Breakdown

Below is a breakdown of how your assignment will be marked

• Report – class diagram	5 marks	
• Code consistency and clarity	10 marks	
• Good commenting	5 marks	
• Simplicity of design	5 marks	
• Loading file into array	10 marks	
• Binary search tree [re]building	15 marks	
• Search array	10 marks	
• Search tree	15 marks	
• Save data to CSV file in sorted order	15 marks	
• ***General error handling/recovery***	***10 marks***	***Do not ignore this!***
Grand Total:	**100 marks**	

4. Submission

You are required to submit your assignment by 5:00pm on 29 October 2013 (week 13). Upload your submission (report and software) electronically via Blackboard, under the Assessments section.

- You may choose to submit the report as a physical printed copy. If you submit it electronically, the report must be either a Word .doc or .docx, PDF, or possibly even an image file (PNG, JPEG). Avoid other formats since if I can’t open it you will lose the marks!
- Please use zip or .tar.gz to group all of your application’s required source code and compiled classes when uploading it via Blackboard.
- DO NOT USE RAR. Only zip or .tar.gz files are acceptable.
- TRY DOWNLOADING AND OPENING THE ZIP YOU HAVE SUBMITTED to make sure that Blackboard uploaded it OK
- PRESS SUBMIT ONLY WHEN YOU ARE READY – “Save” doesn’t finalise the submission, so you can use “Save” to allow later updates. But be aware that you must press Submit before the deadline.

If you fail to upload your data before the deadline and you have a copy of your code already on the Computing Linux home area, DO NOT TOUCH ANYTHING. I will talk to you individually and ensure the timestamps of the assignment files in your home area predate the assignment due date and copy them manually. If you have it all at home then I cannot help you.

Appendix A: Notes

- *Commenting*: Ideally a reader should be able to skim through the comments in your code to understand what the code is supposed to be doing. Good commenting should:
  - Briefly describe the purpose of blocks of code, rather than restating every line of code.
  - Explain not just *what*, but also summarise *why* and *how* for the more difficult parts.
  - Accurately reflect the code.

- *Clarity*: Names should be meaningful. Moreover, avoid unnecessarily complicated code since it's unmaintainable and makes it hard for people to understand your code. Good indentation and the use of blank lines to block related code together is also a major aspect of clarity.

- *Consistency*: Variables, classes and methods should be named according to a standard – choose a standard and stick with it. Also, similar structures in different parts of the code (eg: loops through an array) should follow similar styles – since you are writing all the code this shouldn't be too hard to ensure.

THESE ARE EASY MARKS! If you do them as you go you won't even notice the effort.

Appendix B: General Hints

- *Array size*: If you are loading the data into an array, you may assume a maximum of 1,000,000 WarehoueItems. Note that you won't necessarily *store* 1,000,000 though (eg: if you are loading the smaller dataset); the program must be able to handle any arbitrary length up to 1,000,000. In other words, you need to keep a count of the stored total.

- *Sorting in the Prac vs Sorting in the Assignment*: Beware that in the prac, you worked directly on an array of numbers and those numbers are *both* the elements to sort *and* the sorting criteria. In the assignment, the elements to sort are WarehouseItem objects (records), and the sorting criteria is the key field *inside* the object. So you won't be sorting a nice simple array of numbers, but rather an array of objects, and your checks on sorting order will be on the appropriate getter *in* the WarehouseItem (ie: getKey()) rather than the whole object itself.

- *Efficiency*: Note that there are ZERO marks for efficient code. The purpose of this assignment is for you to learn how to write an application, with emphasis on the elegance of your approach. This is a bit of a golden rule in applications programming – elegance first, efficiency later. Why? Because (1) elegant code will be less buggy, easier to fix and easier to extend; and (2) it's usually not hard to make elegant code faster, but you rarely can make efficient code more maintainable.