Project 2 for CSE 332 (Spring 2011)

Dictionary Face-Off

Phase A due Wednesday, April 27 at 11:00 PM via Catalyst CollectIt.
Phase B due Tuesday, May 3 at 11:00 PM via Catalyst CollectIt.

CSE 332: Data Abstractions
The University of Washington, Seattle, Spring 2011
© Steve Tanimoto, April, 2011

Overview: In this project you will create two or more implementations of the Dictionary ADT. In Phase A, each partner will create an applet that features one dictionary implementation. In Phase B, the team will work together to combine the dictionary implementations in a special way. There will be a new data structure, called a Redundant Array of Different Dictionary Implementations ("RADDI") that functions like a list of dictionaries but responds to commands as described in this document. Teams of either two or three students are acceptable.

Purposes: (a) gain implementation experience with dictionary ADTs, (b) learn how to instrument a data structure to measure its performance, (c) perform actual experiments that reveal the different behavior of alternative approaches to dictionary implementation, and (d) have fun predicting which data structures will do better in various use cases.

Resources Provided: You'll start with the following resources:
1. The Visual Stack Applet and associated files from Project 1.
2. A "Visual Binary Tree Applet" that handles the graphical layout of binary trees. This will simplify the graphical part of the implementation of any of the tree-based dictionary implementations.

Required Functionality for Phase A: You will select two data structures (one per partner) from the following list. Note that if one partner chooses hashtable, then the other partner(s) choose among AVL trees and B-Trees, etc. If a team has three partners, then all three types of data structures will be covered.

Choice list:
AVL Trees
B-Trees with any fixed M>2
Hashtable with given capacity.

Then you'll implement the following new required functionality:

a. Basic dictionary operations – INSERT, FIND and DELETE. However, each operation will have a slightly special meaning. It will take a single argument (a key) which is a word (a string of letters – any other character after the first letter signifies the end of the word). The word is then converted to all lower-case characters (if it wasn't already). Your dictionaries will be used to count occurrences of words (again, actually just strings). The command “INSERT apple” will first do a FIND on “apple”. If there is no entry in the dictionary for “apple” then INSERT puts a new entry into the dictionary and gives it a value of 1. If there already is an entry for “applet”, then its value is incremented by 1. A FIND operation will return 0 if there is no entry for the given word; otherwise, it returns the numeric value associated with the word. DELETE causes the entry corresponding to its argument to be removed from the tree. (You may use lazy deletion to implement the DELETE operation --- by setting the value to 0, but extra credit will be awarded for implementing the real, non-lazy method(s). If you implement non-lazy deletion, then also implement lazy deletion and include support for a command LAZY_DELETION that is used with an argument that is either “on” or “off” to control what form of deletion the DELETE command will then use.)

b. Extended dictionary operations KEYS, PAIRS, SIZE, INSERT-FILE, INSERT-TEXT and STATS.

The KEYS operation outputs a list of all the keys in the dictionary. The order should be ascending, and the keys should be separated by `“, ”` (a comma followed by a space). The PAIRS operations outputs a list of all the (key, value) pairs in the dictionary, again sorted by key and separated by a comma and a space. For this assignment, the values are the counts of occurrences of the keys. SIZE should return the number of valid entries in the dictionary. (It should not count any deleted elements, and it should count 1 for each key, even if it was inserted multiple times.)

INSERT-FILE should take one argument, a path including a valid file name, and it should read the file and process it by finding all the words in it and inserting them into the dictionary. For purposes of this assignment, consider a word to be a string of alphanumeric characters (letters and digits). You do not have to handle Unicode, special characters, etc. Any non-alphabetic character in the file can be treated as a space or word separator. An actual string such as “<i>Don't bug me.<i>” would turn into 6 instances of words: “i”, “don”, “t”, “bug”, “me”,”i”. Notice again that each word is converted to lower-case.

The command INSERT-TEXT provides a way to “mass-insert” some textual material without having to have a file. It should cause successive lines in the command text area to be processed (the same way as INSERT-FILE does) until it reaches a line that starts with “END-TEXT”. Any semicolons within this block of text should be ignored and thus a semicolon does not signify a comment when it occurs between an INSERT-TEXT command and and “END-TEXT” command.

c. The STATS command. It should cause your data structure to report its useful statistics. There will be three kinds of statistics:
 – dictionary statistics (which should be independent of the implementation),
 – cost information (where the numbers do tend to depend on the implementation) and
 – implementation statistics where the numbers themselves only have meaning for the particular data structure.

The dictionary statistics are the counts of the basic dictionary operations performed. The cost statistics include average cost per INSERT, etc. in terms of “elementary operations” that you define precisely[1]. The implementation statistics are those specific to the implementation. For a hash table, the current load factor is one such statistic. For a binary search tree, the average node depth and greatest node depth are of interest.

d. Display of your data structure within the applet.

e. Put your applet on the web.

Phase B Functionality: You'll implement the following functionality for an advanced version of your player:

f. Support for multiple dictionaries in the form of a list of dictionaries. (We'll call this a “RADDI”.)

g. Support for a command CREATE that takes as a first argument a name that controls the type of implementation. For example, CREATE AVL sets up an empty AVL tree in the next available position in the list of dictionaries. Additional arguments may be used, if your implementation can benefit from them. For example, CREATE HASHTABLE 15 QUADRATIC would be used to specify a new empty hash table with capacity 15, and using quadratic probing for resolving collisions. A default hashing function might be used here. (However, you are welcome to offer various alternative hash functions and have an argument to the CREATE command for specifying what hash function to use.) Note: *you* as a designer specify the number, names, and meanings of the arguments to your CREATE command so that you can offer options when instantiating your dictionaries.

h. Support for a command REMOVE_DICT_AT k. This command takes as its argument an integer greater than or equal to 0 and less than the number of dictionary instances in your RADDI. It should delete the one in position k. If k is not valid, then it should report an error.

i. In the Phase B applet, all the Phase A commands (except STATS) should not only work, but they should be invoked on ALL of the data structures in your list of dictionaries. For example, INSERT “apple” should cause the word “apple” to be inserted in all of the dictionaries. However, the STATS command will have a slightly different interpretation (explained below).

Recommended Development Sequence: The following steps are recommended.
1. Choose your partner(s).
2. Select your data structures for Phase A, coordinating with each other.
3. Start coming up with a good “IDIA” (Individual Dictionary Implementation Applet). That is, individually implement your chosen data structures in separate applets, getting the commands and the visualizations to work in the Visual Data Structure Applet Framework. Your experience from Project 1, Phase A should come in handy for this.
4. Set up your applet to run the basic demo sequence (to be announced).
5. Post your applet on the web.

1 Expect further discussion about this on GoPost or in class.

6. Turn in your code and link file for Phase A.

7. “Get RADDI for fun.” Work with your partner on a new applet that features a new “RADDI” visible data structure, which is a “Redundant Array of Different Dictionary Implementations”. Actually, it's just a list of dictionary implementations. Set it up so that it will display each dictionary in the list by calling each dictionary's own renderDS method. But have that method take a starting x coordinate, so that the dictionaries can be drawn side-by-side, one after another. Optional: have the applet draw a filled, colored rectangle behind each data structure, using distinct colors, so that its easier to navigate within the scrolled panel to one data structure or another.
8. Each partner now individually take the new RADDI applet and adapt your dictionary implementation to fit into the new structure. If you and your partner(s) have used a similar interface, abstract class, or set of calling conventions for your separate dictionary implementations, then it should be relatively easy to allow your structure to be instantiated any number of times in the new applet. Your experience with Phase B of Project 1 should come in handy here, where you adapted your queue from Phase A to fit into a list of queues in Phase B. The new twist in the current project is that the dictionaries can be of two or more implementation types. However, it also should also work to have several dictionaries of the same type. Having five identical AVL trees might not be very interesting, but five hash tables, each with a different capacity and/or hash function and/or collision-resolution policy could be quite interesting.

9. Adapt the processing so that all the commands except CREATE, REMOVE_DICT_AT and STATS are applied to all the data structures in the list. The results should generally be reported as follows:
If all dictionaries report the same result (call it R), the RADDI structure should report “All data structures respond: “ followed by R. If the results differ, then RADDI should report them separately, as in the following responses to “INSERT apple”:
  AVL Tree in position 0 reports “OK”
  Hash Table in position 1 reports: “Error: could not insert 'apple' because table is full.”
  Hash Table in position 2 reports: “OK”

10. The STATS command should cause each data structure to report its statistics, and the answers should be concatenated in a readable fashion. Each structure should report the following information: (a) what kind of structure it is, possibly including the values of parameters specified when it was created such as hashtable capacity; (b) the counts of the basic dictionary operations performed, (c) the average cost of each insert, find, and delete operation so far in the current session (i.e., since the most recent RESET operation or, if none, since the program was started), (d) total cost of all operations so far for this data structure, and (e) the interesting statistics for that structure, such as load factor, average node depth, etc.

11. Implement a new command MONITOR that (a) creates a new window (JFrame) containing a panel (you'll typically subclass JPanel) and (b) draws a running graph of the performance of each of the dictionary instances in the RADDI. The graph should have on its horizontal axis the number of basic operations. Basic operations are insertions, finds, and deletes. If the monitor is open, then the display should be updated automatically as operations happen (somewhat like the history window). Make it easy for a person watching

your monitor window to tell which curves go with which dictionaries, and make it easy to compare performance.

12. Get your applet to correctly run the canonical demo for Phase B.
13. Jar-up your applet and put it on the web.
14. Turn in your advanced applet code and link file.

Further Instructions and Information: Updates and clarifications to this project will be posted in the Catalyst GoPost discussion topic “Project 2” and/or in the “clarifications” box on the Projects webpage.