CS2040 Data Structures and Algorithms: Problem Set 1

Due: Fri, Aug 31, 2018

Harold Soh

This assignment is GRADED. Please take note of the due date and submit your solutions on Coursemology.

Overview Sometimes you have an idea for an algorithm. It seems like a really good idea. It *should* result in much better performance. Unfortunately, it is hard to know for sure whether it really does work. What do you do? In Task 1, we consider the problem of "List Maintenance." It seems like a very natural idea to move items that are frequently accessed to the front, leaving infrequently accessed items at the back of the list. Your job is to simulate this process and see how well it works.

Collaboration Policy. You are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. We may, randomly, ask you questions about your solution, and if you cannot answer them, we will assume you have cheated. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person that you talked to about the problem (even if you only discussed it briefly). Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

Task 1: List Maintenance

(Total: 30 points + 3 bonus points) The problem with lists is that they are either expensive to access or expensive to modify. For example, consider two possible designs for a list that is stored in an array:

1. *Sorted:* The list is kept in sorted order. Searching a list of n elements takes time $O(\log n)$ using binary search. Inserting a new element takes time $O(n)$, as you have to make room for the new element in the array.
2. *Unsorted:* The list is not sorted. Searching a list of n elements takes time $O(n)$ using a linear search of the array. Inserting a new element takes time $O(1)$, as you simply add it to the end of the array.

The Boss wants you to speed up the current version of your software that heavily uses a list. You can safely assume that the list will only to store non-negative integers. Your colleague Naruto has kindly coded up a `List` interface and a sample class `ArrayList` which implements `List` using an array data structure. Naruto has also provided a simulator class `ListSimulator` that simulates operations on lists. The following are features of the provided code:

- The `ArrayList` class stores the list in an array of fixed length.
- The `final` (constant) variable `LISTSIZE` defines the size of the list to simulate, while the `final` variable `NUMQUERIES` defines the number of times the list will be searched during the simulation.
- For the purpose of the simulation, the list consists of exactly the integers

$$0, 1, 2, 3, \ldots, \texttt{LISTSIZE} - 1.$$

- Each element in the list is accessed according to a fixed distribution.
- The simulator can be configured to run three different experiments, depending on the initial ordering of the items in the list. For the `DECREASING` experiment, items are initially added to the list in order of decreasing probability: the most accessed elements are at the beginning of the list. For the `INCREASING` experiment, items are initially added to the list in order of increasing probability. For the `RANDOM` experiment, items are initially added to the list in a random order. Intuitively, the `DECREASING` order is the best order for the list.
- The `main` method demonstrates how to use the simulator to run the three experiments for the `ArrayList` class. Notice in particular how the `StopWatch` class is used to measure time.
- The distribution is stored in the variable `m_ListProb`, and is generated by `generateListProbs`. The probability of accessing integer j is stored in `m_ListProb[j]`. For example, the probability of accessing 2 is stored in `m_ListProb[2]`.

Some extra information:

- The distribution is stored in the variable `m_ListProb`, and is generated by `generateListProbs`. The probability of accessing integer j is stored in `m_ListProb[j]`. For example, the probability of accessing 2 is stored in `m_ListProb[2]`.
- For the purpose of simulation, we assume that if $i < j$, then i is accessed with larger probability than j. For example, the integer 0 is accessed with the highest probability, and the integer `LISTSIZE` $- 1$ is accessed with the lowest probability.

Here is one idea for improving the performance of a list: when you insert an item in the list, add it to the end of the list; when you access an item in the list, move it to the front of the list

How much does this strategy help? If the same element is accessed repeatedly, then this strategy should help a lot. Many real-world systems have exactly this type of *locality*, i.e., the same elements are used over and over again, and hence you would expect this strategy to help.

Problem 1.1 (7 points) The class `ArrayList` implements `add` by adding an element to the end of the list, and `search` by performing a linear search through the list.

Write a new class `MoveToFrontArrayList` that extends `ArrayList` and overrides `search`. The new version of `search` should:

- Perform a linear search for the specified item.
- If the item is found, then move that item to the front of the list (i.e., to slot 0 in the array) by shifting every intervening item back one slot.
- Return `true` if the item is found and `false` if it is not.

Your new class *must* extend `ArrayList`, i.e., be a child class of the fixed-length list. (Otherwise, the simulator will not simulate it!) Use the `MoveToFrontArrayListTest` JUnit test to test your solution. (Of course, you can create your own tests if you want!)

Problem 1.2 (7 points) You can see that adding an item to the front of the array is quite expensive since we have the push back all the other items. Perhaps a linked list will perform better? First, let's implement a class `LinkedList` based on the provided `List` interface. We have provided a code skeleton for you to get started in file `LinkedList.java`. You'll also notice a *nested* class called Node. Implement the following methods:

- `add(int key)` to add a key to the end of the linked list
- `search(int key)` which finds a key
- `toString(int key)` which returns the linked list in a `String` format. The string should be the elements of the linked list separated by a space. If the linked list holds 3→4→5, the output should be "3 4 5".

To test your solution, use the `LinkedListTest` TestNG test file locally. Once you are happy with your solution, try the tests on Coursemology. **Note!** this should be a basic LinkedList class that **does not** perform the move to front strategy.

Problem 1.3 (7 points) Create a new class `MoveToFrontLinkedList` that extends `LinkedList` and overrides `search`. The new version of `search` should perform the same strategy used in Problem 1.1:

- Perform a linear search for the specified item.
- If the item is found, then move that item to the front of the list.
- Return `true` if the item is found and `false` if it is not.

Your new class *must* extend `LinkedList`, i.e., be a child class of your linked list class. Again, you are provided with a simple test file, `MoveToFrontLinkedListTest`, to test your solution.

Use the simulator to compare `MoveToFrontLinkedList` and `MoveToFrontArrayList`. Is there a performance difference? Why or why not? Use your knowledge of computational complexity to justify what you observe.

Problem 1.4 (5 points) Implement one other list maintenance heuristic of your choosing. That is, extend either `ArrayList` or `LinkedList` such that it implements search in a different manner (perhaps using some alternate rule as to how to move items when they have been searched for). Note that it should still be a list; replacing the list with other data structures (such as trees) breaks the spirit of *list* management. Call your new class `MyFastList`. (Who can come up with the fastest algorithm?) Explain clearly how your list algorithm works. *Remember that the list class is not allowed to know anything about the distribution being used by the simulator.*

Problem 1.5 (4 points) Run several experiments comparing the performance of classes you've implemented: `ArrayList`, `MoveToFrontArrayList`, `LinkedList`, `MoveToFrontLinkedList`, and `MyFastList`. (Again, look at how the simulator uses the StopWatch class to measure time.) Which one performs best? What is your conclusion on the capability of the *Move-to-Front* heuristic?

Problem 1.6 (Optional Bonus. 3 points) One major problem with the current strategy is that searching for an item *not* in the list always takes $O(n)$ time for a list of size n. That is, to discover that an item is not in the list requires exhaustively scanning the entire list. Let's try solving this problem as follows: whenever a client searches for item k and it is not found, insert $-k$ in the list and move it to the front. Now, whenever a search for k finds $-k$, it can stop searching—and move that item to the front of the list. Notice that you also have to update the `add` routine to make this work (and it will not work for $k = 0$).

Try this idea out and see how much improvement you get. What are the advantages and disadvantages of this strategy?

Using the negation of a key as a signal is, in fact, a bad idea from a software engineering perspective. What is the right way of implementing this strategy?

— END OF PROBLEM SET 1 —