CS2510: Data Structures and Algorithms – Assignment 1

Muhammad Yaseen Khan
DCS, MAJU

Fall – September 28, 2018

Instructions.

(i) Do not cheat. (ii) The assignment is consisting of four questions. (iii) You have to submit one `***.java` file that accommodate answers to all questions. (iv) Those questions where a descriptive answer is required, you can do in the same `***.java` file, as/in multi-line comments. (v) Read the first instruction again. (vi) This is an individual assignment. (vii) `***` in the instructions iii and iv means your student id. (viii) Failing to comply the instructions is same as failing to get assignment graded.

Questions.

1. THE LISTS. In class we have learnt about the list as an Abstract Data Types (ADT). The main operations involved in the lists are:

   (a) `isEmpty(item)` – Informs whether a list is empty or not? The cost of this operation is $\mathcal{O}(1)$.

   (b) `add(item)` – Adds/Appends the given item at the end of the list. It has the internal ability to grow itself upon the situation when the list if full. The cost of this operation is $\mathcal{O}(1)$ when the list has space. And, when there is requirement of growing the list, the cost of amortisation i.e. $\mathcal{O}(n)$ is added, hence, it will be $\mathcal{O}(n)+\mathcal{O}(1)$ which is $\approx\mathcal{O}(n)$.

   (c) `add(item, position)` – Inserts the given item on specified position. Doing this involves the skipping of pre-existing items one block ahead. This operation also has the intrinsic ability to grow itself upon the situation when the list is full. The cost of this operation is $\mathcal{O}(m)+\mathcal{O}(1)$ when the list has space, such that $\mathcal{O}(m)$ indicates the cost of skipping m items forward for adding the item at the specified position. Further, if there is no space left for adding the item, the growth of list has to be taken place, for which the cost will be $\mathcal{O}(n)+\mathcal{O}(m)+\mathcal{O}(1)$, which is nearly equal to $\mathcal{O}(m+n)$, provided that $m \leq n$.

   (d) `remove(item)` – Checks, whether the given item exists in the list or not? If ‘yes’, then removes/deletes the given item at the position it found first. Doing this involves the skipping of items one block backwards. It has the internal ability to shrink the list when removal of one item makes the more than half of the list empty. The cost of removing an item from the list is $\mathcal{O}(m)$, such that m items have to be skipped backwards. If on removal, the list has space more than the half of it, then, the amortisation cost of shrinking the list is added, hence the cost will be $\mathcal{O}(n) + \mathcal{O}(m)$, which is $\approx \mathcal{O}(m+n)$.

(e) `update(new_item, position)` – Locates the position in the list, and assigns the new item. The cost of the operation is $\mathcal{O}(1)$.

(f) `search(item)` – Sequentially searches the given item in the list. On seeking the item in the list it returns the position where it is located, and in the failure, it returns -1. The cost of operation is $\mathcal{O}(n)$.

Task 1: Implement the list data structure in Java. Make a class with the name `MyList` that has the implementation of list operations, as stated above. Essentially, you have to utilise the generics in implementation.

Task 2: Synthesise the operation `remove(item)` by altering its functionality. Make a function `removeAll(item)` which implements the logic of removing all instances of the given item in the list.

Task 3: Make a new operation `print()` to display all items in the list.

Task 4: Write a `Test` class to verify your implementation. Use `print()` after using other functions.

2. THE SORTED LISTS. Synthesise your list's implementation to add items in a sorted way or in the ascending order. Typically all operations will have ditto implementation of the list, except `add(item)`, where you have to search the position for the item to be added.

   Task 5: Implement the synthesised operation of list by making a class `MySortedList`. Ideally, you can utilise your own implementation of `search(item)` for locating the position, followed by calling `add(item, position)` internally. Remember, external call of the function `add(item, position)` must be restricted, because, the operation of `add(item, position)` no longer useful. You have to utilise generics in this implementation as well. Although its generics, you can leave its usage with the `String` type.

   Task 6: Write a `Test` class to verify your implementation. Use `print()` after using other functions.

3. AN DIE FREUDE – ODE TO JOY.

   Task 7: Implement a function `addNumbers(start, end)` which can add all numbers exist in the range `[start, end]`.

   Task 8: Calculate its asymptotic complexity for the worst case.

   Task 9: Can the worst case be good as $\mathcal{O}(1)$?

4. ASYMPTOTIC COMPLEXITY ANALYSIS. In class, we have discussed the comparison of operations in two data structures in a great detail. For example, we argued the worst

cases of adding items in the first data structure is more/less costly than adding them in the second one. We also, summarise our arguments in the form of Big-O notations.

Task 10: Now consider (do not implement) the `Linked-Lists` and it's synthesised version `Sorted Linked-Lists`, and provide the comparative analysis (description) of their operations. Also summarised them in Big-O notations.

(a) adding an item in `Linked-Lists` vs. adding it in `Sorted Linked-Lists`.

(b) removing an item in `Linked-Lists` vs. removing it in `Sorted Linked-Lists`.

(c) searching an item in `Linked-Lists` vs. searching it in `Sorted Linked-Lists`.

(d) searching an item in `Sorted Singly Linked-Lists` vs. searching it in `Sorted Doubly Linked-Lists`.

Submit timely on your Google Classroom
The due date is October 7, 2018