CSC263 Summer 2022 Problem Set 1

Due June 3, 2022 23:59 ET on MarkUs

- Your problem sets are graded on both correctness and clarity of communication. Solutions that are correct but poorly written will not receive full marks.
- Each problem set may be completed individually or in a group of two students.
- Solutions must be typeset electronically and submitted to MarkUs as a pdf with the filename `ps1.pdf`. Handwritten submissions will receive a grade of zero.
- The work you submit must be that of your group. You may not use or copy from the work of other groups or from external sources like websites or textbooks. All the information you need to answer these questions has been covered in lecture and/or in the textbook.
- If you consult any external sources (i.e. other than the lecture notes, CLRS, or documents posted to the course website), then you must clearly indicate these additional sources on the first page of your submission.

Question 1 [6 marks]

Consider the unsorted array implementation of a dictionary that was discussed during the week 1 lecture. The following block of code is a possible implementation of `delete(D, k)`. The `D.arr` variable points to the underlying unsorted array, and the `D.size` variable is an integer denoting the number of elements currently stored by the dictionary. For the sake of this question, assume that the elements stored in the dictionary are simply keys with no associated values.

```
1  def delete(D, k):
2      i = 0
3      while i < D.size:
4          if D.arr[i] == k:
5              break
6          i += 1
7
8      if i == D.size:
9          return False
10
11     for j in range(i, D.size - 1):
12         D.arr[j] = D.arr[j+1]
13     D.size -= 1
14     return True
```

The `insert(D, k)` operation first searches for the key k in the dictionary and, if it is not already present, sets `D.arr[D.size] = k` and then increments `D.size`. (The *precise* implementation of `insert` is not important for this question).

Consider the following sequence of operations performed on an initially empty dictionary called `D`: `insert(D,` k_1`)`, ..., `insert(D,` k_n`)`, `delete(D,` ℓ`)`. Suppose that every key $k_1, \ldots, k_n, \ell$ is sampled independently and uniformly at random from $\{1, \ldots, n\}$. In this question, we will count the number of times that the `delete(D,` ℓ`)` operation performs the comparison `i == D.size` on line 8 and the assignment `D.arr[j] = D.arr[j+1]` on line 12 in the best case, worst case, and average case.

(a) Compute a *tight bound* (using asymptotic notation) on the sum of the number of times line 8 and line 12 are executed in the *best case*. Justify your answer.

(b) Compute a *tight bound* (using asymptotic notation) on the sum of the number of times line 8 and line 12 are executed in the *worst case*. Justify your answer.

(c) Compute the expected sum of the number of times line 8 and line 12 are executed. Show all of your work, and give an exact number as your solution (i.e. no asymptotic notation).

Question 2 [4 marks]

We have seen one possible notion of a "balanced" binary search tree during class: A BST T is *AVL balanced* if and only if, for every node x in T, the height of the subtrees rooted at $x.left$ and $x.right$ differ by at most 1.

In this question, we consider a different notion of balance: A BST is *depth-balanced* if and only if, for every node x in T, the height of the subtrees rooted at $x.left$ and $x.right$ differ by at most $d(x)+1$, where $d(x)$ denotes the depth of node x in T. Recall that a node's depth is equal to its distance from the root. In particular, $T.root$ has depth 0, the children of $T.root$ have depth 1, and so on.

(a) Draw an example of a BST that is depth-balanced but *not* AVL balanced. Indicate the key and the balance factor of each node. (You may choose the keys of the tree arbitrarily, as long as the tree is a BST.)

(b) Give the number j such that,

i. for all depth-balanced BSTs T with j or fewer nodes, T is also AVL balanced, and

ii. there exists a depth-balanced BST T' with $j+1$ nodes that is *not* AVL balanced.

Prove that your answer is correct.

Question 3 [10 marks]

We wish to simulate a robotic arm that cooks a sequence of pancakes on an infinitely long griddle. Each pancake has the following set of attributes: flavour (e.g. plain, blueberry, or chocolate chip) and mass (in grams). If there is at least one pancake in the sequence, then the robotic arm holds exactly one of these pancakes on its spatula, which is considered to be the *current* pancake. The following ADT represents this scenario:

- Data: a sequence of pancakes. If the sequence is nonempty, then one of the pancakes is considered to be the *current* pancake.
- Operations:
  - `Forward()`: Move the robotic arm to the next pancake in the sequence, updating the *current* pancake. If the *current* pancake is the last one in the sequence before `Forward()` is applied, or the sequence is empty, then do nothing.
  - `Backward()`: Move the robotic arm to the previous pancake in the sequence, updating the *current* pancake. If the *current* pancake is the first one in the sequence before `Backward()` is applied, or the sequence is empty, then do nothing.
  - `Move(`j`)`: Move the arm to the j-th pancake in the sequence, setting it to be the *current* pancake. (You may assume that a j-th pancake exists.)
  - `Serve()`: Remove the *current* pancake from the sequence. Set the new *current* pancake to be the previous one in the sequence. If the *current* pancake was already the first one in the sequence, then set the *current* pancake to the next one in the sequence. If the sequence was empty before `Serve()` was applied, then do nothing.
  - `Cook(`p`)`: Add the new pancake p to the end of the sequence.
  - `Examine()`: Return the attributes of the *current* pancake.

Design a data structure to implement this ADT. When the sequence contains n pancakes, `Forward()`, `Backward()`, and `Examine()` should have worst case runtime $O(1)$, and `Move(`j`)`, `Serve()`, and `Cook(`p`)` should have worst-case runtime $O(\log n)$.

When asked to provide an implementation of an operation, you *must* give a high-level description of your algorithm in English! You may reference algorithms seen in class without describing them, and simply describe the modifications made, if any. You may supplement your description with pseudo-code, but it is not necessary.

YOUR ANSWER FOR ALL PARTS OF THIS QUESTION MUST NOT EXCEED 3 PAGES COMBINED (using 1 in. margins and 11 pt. font). Any content beyond the first 3 pages will not be marked.

(a) Describe the data structure you will use to implement this ADT. Explain any augmentations that are needed to implement the operations.

(b) Provide an implementation of `Forward()`. Briefly justify why its worst case runtime is $O(1)$.

(c) Provide your implementation of `Backward()`. Briefly justify why its worst case runtime is $O(1)$.

(d) Provide your implementation of `Move(`j`)`. Briefly justify why its worst case runtime is $O(\log n)$.

(e) Provide your implementation of `Serve()`. Briefly justify why its worst case runtime is $O(\log n)$.

(f) Provide your implementation of `Cook(`p`)`. Briefly justify why its worst case runtime is $O(\log n)$.

(g) Provide your implementation of `Examine()`. Briefly justify why its worst case runtime is $O(1)$.