CSC265 F18: Assignment 3

Due: October 24, by midnight

Guidelines: (read fully!!)

- Your assignment solution must be submitted as a *typed* PDF document. Scanned handwritten solutions, solutions in any other format, or unreadable solutions will **not** be accepted or marked. You are encouraged to learn the LaTeX typesetting system and use it to type your solution. See the course website for LaTeX resources. Solutions typed using LaTeX receive 2 bonus marks.
- Your submission should be no more than 8 pages long, in a single-column US Letter or A4 page format, using at least 9 pt font and 1 inch margins.
- To submit this assignment, use the MarkUs system, at URL `https://markus.teach.cs.toronto.edu/csc265-2018-09`
- This is a *group assignment.* This means that you can work on this assignment with *at most one other* student. You are *strongly encouraged* to work with a partner. Both partners in the group should work on and arrive at the solution together. Both partners receive the same mark on this assignment.
- Work on all problems together. For each problem, one of you should write the solution, and one should proof-read and revise it. The first page of your submission must list the *name*, *student ID*, and *UTOR email address* of both group members. It should also list, for each problem, which group member wrote the problem, and which group member proof-read and revised it.
- You **may not** consult any other resources except: your partner; your class notes; your textbook and assigned readings. *Consulting any other resource, or collaborating with students other than your group partner, is a violation of the academic integrity policy!*
- You may use any data structure, algorithm, or theorem previously studied in class, or in one of the prerequisites of this course, by just referring to it, and without describing it. This includes any data structure, algorithm, or theorem we covered in lecture, in a tutorial, or in any of the assigned readings. Be sure to give a *precise reference* for the data structure/algorithm/result you are using.
- Unless stated otherwise, you should justify all your answers using rigorous arguments. Your solution will be marked based both on its completeness and correctness, and also on the clarity and precision of your explanation.

Question 1. (18 marks)

The DOMINATING-POINTS ADT supports a set P of n points in $\mathbb{Z}^2$ (i.e. points in the plane with integer coordinates) under the following operations:

- INSERT(p) inserts a new point p into P, where p is specified by its x- and y-coordinates $p.x$ and $p.y$.
- DOMINATING(q) returns the coordinates of a point $p \in P$ which is above and to the right of the point q, if such exists. Assume q is specified by its x- and y-coordinates $q.x$ and $q.y$. Then the operation must return $p.x$ and $p.y$ for some $p \in P$ such that $p.x \geq q.x$ and $p.y \geq q.y$, if such exists, or return FAIL otherwise.

For this question, you can assume that at any point in time there are no two points in P with the same x or y coordinate. Your goal will be to implement both operations in worst-case running time $O(\log n)$.

Part a.

Describe an augmented AVL tree which allows you to implement INSERT and DOMINATING in the required worst-case time complexities. Specify what keys and auxiliary fields you are using for your AVL tree.

[Solution]

The data structure will be an AVL tree T, with the x-coordinates of the points in P as keys. Each node u of T corresponds to one point $p \in P$ and has key $u.key = p.x$, and additional fields:

- $u.y = p.y$;
- $u.max\text{-}y$ storing the maximum y-coordinate over the points whose x-coordinates are stored in the subtree rooted at u;
- $u.max\text{-}y\text{-}x$ storing the x-coordinate of the point whose y-coordinate is stored in $u.max\text{-}y$

Part b.

Describe an algorithm implementing the procedure DOMINATING(q) in clear and precise English, and, optionally, using pseudocode. Justify why the procedure correctly returns a point $p \in P$ dominating q, and why it runs in time $O(\log n)$.

[Solution]

Our algorithm is described by the following pseudocode. The algorithm takes q and a pointer to a node u of T. It returns a point p which dominates q, among those whose x-coordinate is stored under u, if such exists; otherwise it returns FAIL. Then we just need to call this function at the root of T.

```
DOMINATING($q, u$)
if $u$ == NIL
    return FAIL
if $u.key < q.x$
    return DOMINATING($q, u.right$)
else if $u.y \geq q.y$
        return $(u.key, u.y)$
    else $v = u.right$
        if $v.max\text{-}y \geq q.y$
            return $(v.max\text{-}y\text{-}x, v.max\text{-}y.)$
        else return DOMINATING($q, u.left$)
```

Suppose that u stores the x-coordinate of the point p. Then, if $p.x < q.x$, it must be the case that any point dominating q has to be to the right of p, and we recurse on the corresponding side of the tree. Otherwise, a point dominating q may be to the left or to the right of p. We check if p dominates q and return it if it does. Observe that, when $q.x \le p.x$, we have that a point to the right of p dominates q if and only if it has larger y coordinate than q. Therefore, if any point to the right of p dominates q, the point with the maximum y coordinate will. We check if this is the case by using the *max* field of u. If it is not the case, then the only points dominating q must be to the left of p, and we recurse on the corresponding side of the tree. The correctness of the procedure follows by induction on the height of u.

For the running time, notice that every call to DOMINATING(q, u) takes time $O(1)$. Every time we call DOMINATING(q, u), we do so on a node u which is one edge further from the root of T than the previous call. Therefore, the number of calls to the procedure is bounded by the height of T, which is $O(\log n)$. So, we have total running time $O(\log n)$.

Part c.

Describe an algorithm implementing the procedure INSERT(p) in clear and precise English. Justify why the procedure runs in time $O(\log n)$ and preserves the properties of the data structure, as you described them in the first subproblem.

[Solution]

We first insert $p.x$ into T as in any BST. Suppose that the new node inserted is u. We go up the path from u to the root of T, and for any v on this path we check if $p.y > v.\mathit{max\text{-}y}$. Whenever that's the case, we set $v.\mathit{max\text{-}y} = p.y$ and $v.\mathit{max\text{-}y\text{-}x} = p.x$. Then, we go back to u and start going up the path to the root again, updating the balance factors. We perform any necessary single or double rotation. Suppose that, after a rotation, some node v has new left and right subtrees T_1 and T_2, with roots w_1 and w_2, and that the auxiliary fields of w_1 and w_2 have already been updated. Then we set

$$v.\mathit{max\text{-}y} = \max\{w_1.\mathit{max\text{-}y}, w_2.\mathit{max\text{-}y}\},$$

and set $v.\mathit{max\text{-}y\text{-}x}$ to the corresponding x coordinate. I.e. if we set $v.\mathit{max\text{-}y}$ to $w_1.\mathit{max\text{-}y}$, then we set $v.\mathit{max\text{-}y\text{-}x}$ to $w_1.\mathit{max\text{-}y\text{-}x}$ and vice versa. At most two such nodes need to be updated per rotation.

To bound the total running time of INSERT(p) we note that

- the time to insert $p.x$ into T is $O(\log n)$ because T has height $O(\log n)$;
- the time to update $v.\mathit{max\text{-}y}$ and $v.\mathit{max\text{-}y\text{-}x}$ of any node v is $O(1)$; since this is done only for the nodes on the path from u to the root of the tree T, we need to perform at most $O(\log n)$ such updates, and the total running time is $O(\log n)$.
- updating the balance factors takes time $O(\log n)$
- any rotation takes time $O(1)$.

Question 2. (15 marks)

Let T be a complete ternary tree, i.e. a tree in which all leaf nodes are at the same distance from the root, and every tree node except the leaves has exactly three children: a left, a middle, and a right child. Then T has 3^h leaves, where h is its height. Suppose that each leaf of T is given a value 0 or 1. Then we determine values for the internal nodes of T as follows: the value of an internal node u of T equals 0 if at least two of its children have value 0, and its value equals 1 otherwise.

Assume that the values of the leaves, numbered from left to right, are given in an array $A[1\,..\,3^h]$. Using an adversary argument, show that every deterministic algorithm which computes the value of the root of T must query *every* entry of A in the worst case, and, therefore, has worst-case complexity $\Omega(3^h)$.

Describe the adversary's strategy clearly and precisely. Justify why the strategy forces any algorithm which correctly determines the value of the root of T to query every entry of A.

[Solution]

The adversary answers all the algorithm's queries for the values of the entries of A. Before we specify the adversary's strategy, let us give a crucial definition.

Definition 1. *We call a leaf of T* undetermined *if it is has not been queried yet, and we call an internal node of T undetermined if its value cannot be determined from the queries answered so far. I.e. an internal node u is undetermined if there are two ways to set the values of the undetermined leaves in the subtree rooted at u which result in two different values for u.*

Clearly, the algorithm has to keep making queries while the root of T is undetermined.

Suppose that the algorithm queries $A[i]$, and let the i-th leaf of T from the left be ℓ. Without loss of generality, $A[i]$ has not been queried before: otherwise we can just answer with the same value as the previous query. Let u be the closest node to ℓ on the path from ℓ to the root, such that at least two of u's children are undetermined. If no such node exists, we answer the query for $A[i]$ arbitrarily. Otherwise,

- if none of u's children are determined, we answer the query arbitrarily;
- if one of u's children is determined, and its value is $b \in \{0, 1\}$, then we answer $A[i] = 1 - b$.

This fully describes the adversary's strategy.

We claim that a node u of T is undetermined if and only if some leaf in the subtree rooted at u has not been queried yet. We prove this claim by induction on the height of u. In the base case, u has height 0, and is, therefore, a leaf: then the claim follows from the definition of being undetermined. For the inductive step, let u be an internal node of T, and let its children by v, w, x. It is clear that if all leaves under u are determined, then so is u. It remains to prove the converse. Without loss of generality, suppose that some leaf under x is undetermined. Then, by the induction hypothesis, x is also undetermined. Then u could be determined only if v and w are both determined, and both have the same value. Suppose, without loss of generality, that w got determined after v, and let the last undetermined leaf under w be ℓ. By the induction hypothesis, at the time ℓ got queried, all other leaves under v and w were already queried. Therefore, according to the description of the adversary's strategy, the adversary must have answered the query for ℓ with the value opposite to the value of v. Therefore, the value w is the opposite of the value of v, and u could not be determined before x is determined. This completes the proof.