CS120: Intro. to Algorithms and their Limitations Hesterberg & Anshu

Problem Set 2

Harvard SEAS - Fall 2023 *Due: Tue Sep. 27, 2023 (11:59pm)*

Your name: Samuel Osa-Agbontaen
Collaborators: Office Hours
No. of late days used on previous psets: 1
No. of late days used after including this pset: 1

Please review the Syllabus for information on the collaboration policy, grading scale, revisions, and late days.

1. (reductions) The purpose of this exercise is to give you practice formulating reductions and proving their correctness and runtime. Consider the following computational problem:

Input	**:** Points $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ in the $\mathbb{R}^2$ plane that are the vertices of a convex polygon (in an arbitrary order) whose interior contains the origin
Output	**:** The area of the polygon formed by the points

Computational Problem AreaOfConvexPolygon

(a) Show that AreaOfConvexPolygon$\leq_{O(n),n}$ Sorting. Be sure to analyze both the correctness and runtime of your reduction. In this part and the next one, you may assume that a point $(x, y) \in \mathbb{R}^2$ can be converted into polar coordinates (r, θ) in constant time.

You may find the following useful:

- The polar coordinates (r, θ) of a point (x, y) are the unique real numbers $r \geq 0$ and $\theta \in [0, 2\pi)$ such that $x = r\cos\theta$ and $y = r\sin\theta$. Or, more geometrically, $r = \sqrt{x^2 + y^2}$ is the distance of the point from the origin, and θ is the angle between the positive x-axis and the ray from the origin to the point.
- The area of a triangle is $A = \sqrt{s(s-a)(s-b)(s-c)}$ where a, b, c are the side lengths of the triangle and $s = \frac{a+b+c}{2}$ (Heron's Formula).

To reduce *AreaofConvexPolygon* to the Sorting problem in linear time, we want to form triangles from each of the given coordinate pairs because our convex polygon can always be formed with triangles, so if we can calculate the area of each triangle in our convex polygon and find their sum, we will have the area of our convex polygon.

First, we start with a list of coordinate pairs $[(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})]$. Next, we want to convert each coordinate pair to a polar coordinate pair $[(\theta_0, r_0), \ldots, (\theta_{n-1}, r_{n-1})]$. Since the problem tells us that we can convert coordinate pairs to polar coordinates in constant time, and we are applying this conversion across n coordinates, then converting the entire list of pairs can be done in $O(n)$ steps.

Next, we will sort our n-length list of polar coordinates by each of their θ_i keys to ensure that adjacent vertices are next to each other. θ_i must be in the keys because our definition of the Sorting Problem introduced in class sorts by keys, and not values.

Afterwards, we will initialize an accumulator (acc) at acc $= 0$. For each polar coordinate (θ_i, r_i) in our list, we will traverse through each pair and increment our accumulator by A_i, where A_i represents the area of the triangle formed by the vertices at (θ_i, r_i), (θ_{i+1}, r_{i+1}), and the origin. Formally,

$$A_i = \frac{1}{2} \times r_i \times r_{i+1} \times \sin(\theta_{i+1} - \theta_i).$$

which follows from the formula to calculate the area of a triangle given two sides and the angle between the two sides. In this case, r_i and r_{i+1} represent the side lengths of our triangle, while $\theta_{i+1} - \theta_i$ represents the angle between those two side lengths.

Iterating across each polar coordinate and incrementing its corresponding area can be done in $O(n)$ steps. We now need to consider the edge case where the final triangle in our polygon is formed from the origin, (θ_{n-1}, r_{n-1}), and $(\theta_0 + 2\pi, r_0)$. To do this, we simply increment A_{edgecase} to our accumulator, where

$$A_{\text{edgecase}} = \frac{1}{2} \times r_{n-1} \times r_0 \times \sin(\theta_0 + 2\pi - \theta_{n-1}).$$

Finally, we just need to return the value of the accumulator, and this value will be equivalent to the size of all the triangles, and consequently, the size of our convex polygon.

When calculating runtime of this reduction, we must consider that it took $\mathrm{O}(n)$ steps to convert each of our coordinate points to polar coordinates and it also took $\mathrm{O}(n)$ steps to iterate through each polar coordinate to find its corresponding triangle's area. So in total it should take $\mathrm{O}(n + n)$ which is $\mathrm{O}(n)$ steps. This reuduction runs in linear time.

When analyzing correctness, we consider that each iteration calculated the area of one of n non-overlapping triangles that make up the convex polygon. Since every convex polygon can be constructed by triangles, we know that the sum of each triangle will equal the sum of the entire convex polygon. Additionally, we must also consider the significance of our oracle call to Sorting. This matters because sorting the polar coordinates by their angular distance to the from the positive x-axis ensures that the triangles formed are non-overlapping.

In the event that trigonometric functions like *sin* are not regarded as basic operations, we would have to use Heron's formula to calculate our area, leaving all the other steps of the aforementioned algorithm the same. To start, we calculate $A = \sqrt{s(s-a)(s-b)(s-c)}$, where $s = \frac{a+b+c}{2}$. In this problem, $a = r_i, B = r_{i+1}$, and we can find the third side, c, by finding the distance between the two coordinate points (x_i, y_i) and (x_{i+1}, y_{i+1}) using the distance formula, $c = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}$, where x_a and x_b respectively represent the Cartesian x coordinate for a and b and y_a and y_b represent the Cartesian

y coordinate for a and b. One might be skeptical of how we would have access to the Cartesian coordinates after already converting to polar coordinates, but this can be resolved by modifying our original input right before we convert to polar coordinates. Specifically, unlike the previous approach where we converted (x_i, y_i) to (θ_i, r_i), instead we will convert to $(\theta_i, (x_i, y_i))$. Since we haven't changed the keys, we can still call our oracle and sort by θ_i just like in our original solution. Any reference made to r_i in the original solution can simply be substitued with $\sqrt{x^2 + y^2}$ in this solution. Since the only thing that was changed was the formula to calculate the area, and these calculations are done in constant time (as was assumed in the original solution), the runtime and correctness reasoning remains the same.

(b) Deduce that AreaOfConvexPolygon can be solved in time $O(n \log n)$.
Let's say our sorting algorithm is Mergesort, which runs in $O(n \log n)$ steps. Since *AreaOfConvexPolygon* is reduced to the Sorting Problem (with an $O(n)$ reduction), and our sorting algorithm, Mergesort, solves the Sorting Problem, the runtime of the sorting algorithm will dominate the runtime of all the other steps involved in solving *AreaOfConvexPolygon*, meaning that the algorithm is also $O(n \log n)$.

Formally, we can also consider Corollary 3.5 discussed in Lecture 3, where *IntervalScheduling-Decision* in that problem is analogous to *AreaOfConvexPolygon* in this problem.

(c) Let Π and Γ be arbitrary computational problems, and suppose that there is a reduction from Π to Γ that runs in time at most $g(n)$ and makes at most $k(n)$ oracle calls, all on instances of size at most $h(n)$. Show that if Γ can be solved in time at most $T(n)$, then Π can be solved in time at most $O(g(n) + k(n) \cdot T(h(n)))$. Note that the case $k(n) = 1$ was stated in class; the case $k(n) > 1$ is useful as well, such as in Part 1d below.
If Π can be reduced to Γ, this means that the runtime to solve Π can be represented by the runtime of the reduction to Γ (we will denote as x) summed with the runtime of the algorithm that solves the oracle (in this case Γ) called during the reduction (we will denote as y) (multiplied across however many times we call the oracle, denoted as z). Formalized, we can say that the runtime of Π is $O(x + y \cdot z)$.

First, we calculate x, the runtime of the reduction, which is the sum of all the steps involved in transforming the inputs before and after the reduction. In this problem, we are told that this process takes $O(g(n))$ time, so $x = O(g(n))$.

Next, we calculate y, the runtime of the algorithm that solves Γ. In this problem, it's given that Γ can be solved in $T(n)$ time, but it should be noted that inputs of size $h(n)$ are fed to the oracle, so we say that our $y = T(h(n))$.

Finally, we calculate z, the amount of oracle calls we need to make. In this problem, it is given that we make $k(n)$ oracle calls, so $z = k(n)$.

The runtime to solve Π is no more than the runtime to solve each of the aforementioned steps, which, when formalized, can be written as $O(x+y+z) = O(g(n)+k(n)\cdot T(h(n)))$.

(d) (*challenge; extra credit; optional[1]) Come up with a way to avoid conversion to polar coordinates and any other trigonometric functions in solving AreaOfConvexPolygon in time $O(n \log n)$. Specifically, design an $O(n)$-time reduction that makes $O(1)$ calls to a Sorting oracle on arrays of length at most n, using only arithmetic operations $+$, $-$, $\times$, $\div$, and $\sqrt{\ }$, along with comparators like $<$ and $==$. (Hint: first partition the input points according to which quadrant they belong in, and consider the slope of the line from a vertex (x,y) to the origin.)

Similar techniques to what you are using in this problem are used in algorithms for other important geometric problems, like finding the Convex Hull of a set of points, which has applications in graphics and machine learning.

[1]This problem is meant to be done based on your enjoyment/interest and only if you have time. It won't make a difference between N, L, R-, and R grades (meaning it will only impact whether an R gets increased to an R+), and course staff will deprioritize questions about this problem at office hours and on Ed.

2. (augmented binary search trees) The purpose of this problem is to give you experience reasoning about correctness and efficiency of dynamic data-structure operations, on variants of binary-search trees.

   Specifically, we will work with *selection data structures*. We have seen how binary search trees can support min queries in time $O(h)$, where h is the height of the tree. A generalization is *selection* queries, where given a natural number q, we want to return the q'th smallest element of the set. So `DS.select(0)` should return the key-value pair with the minimum key among those stored by the data structure `DS`, `DS.select(1)` should return the one with the second-smallest key, `DS.select(n-1)` should return the one with the maximum key if the set is of size n, and `DS.select((n-1)/2)` should return the median element if n is odd.

   In the Roughgarden text (§11.3.9), it is shown that if we *augment* binary search trees by adding to each node v the size of the subtree rooted at v, then Selection queries can be answered in time $O(h)$.[2]

   (a) In the Github repository, we have given you a Python implementation of size-augmented BSTs supporting search, insertion, and selection, and with a stub for `rotate`. One of the implemented functions (`search`, `insert`, or `select`) has a correctness error, another one is too slow (running in time that's (at least) linear in the number of nodes of the tree rather than linear in the height of tree), and the third is correct. Identify and correct these errors. You should provide a text explanation of the errors and your corrections, as well as implement the corrections in Python.
   The incorrect algorithm is `select`. The error occurs when searching a right subtree. The algorithm fails to adjust the index it inputs to account for the first element in the right subtree being larger than every element in the left subtree (as well as the root node). The algorithm behaves correctly when searching left subtrees, so we can extend its functionality to the right subtree search but adjust the input to already account for all the nodes smaller than it (which would be found on the left subtree that we no longer need to search) since the value of a right node is greater than its parent and the left sub-tree. For example, if the size of the left subtree is four, but we are checking the first element in the right subtree, we can update the index to start at five instead of zero because we know the first element in the right subtree is greater than the four elements from that left subtree as well as the root element.

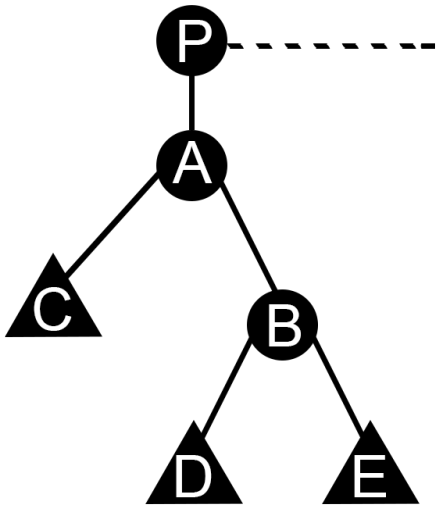
   The inefficient algorithm is `insert` because it calls `calculate_sizes`, which traverses the entire path to the lower level for every vertex, which is $O(n)$, not $O(h)$. To fix this, we should remove `calculate_sizes` and account for the sizes of each node by doing the following: when we insert a new node, we should increment the size of each node we're traversing (along the path of inserting the new node) by 1, since every ancestor of the new node will now account for the size of the new node as one of its descendants. Incrementing is a constant operation, and for each new insert, we will at most need to traverse the path from the root node to a leaf once, which is the height (h) of the tree, making this new algorithm $O(h)$.

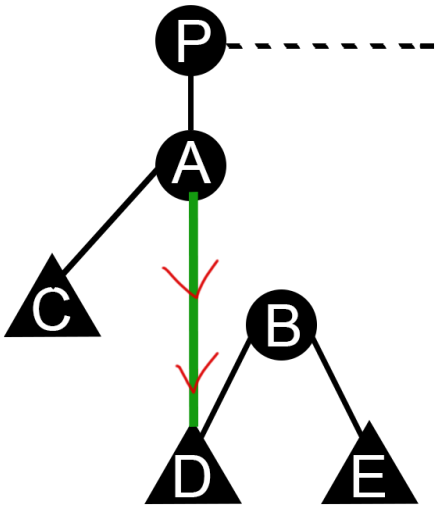
[2] Note that the Roughgarden text uses a different indexing than us for the inputs to Select. For Roughgarden, the minimum key is selected by Select(1), whereas for us it is selected by Select(0).

The correct algorithm is `search`. It solves the problem and only traverses at most the height of the tree.

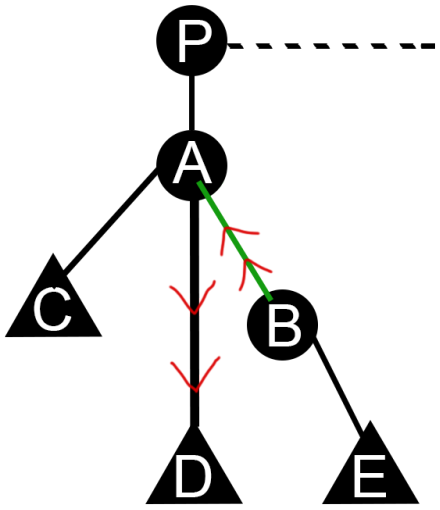
(b) Describe (in pseudocode or pictures) how to extend `rotate` to size-augmented BSTs, and argue that your extension maintains the runtime $O(1)$. Prove that your new rotation operation preserves the invariant of correct size-augmentations. (That is, if every node's size attribute had the correct subtree size before the operation, then the same is true after the operation.)
For context, the top four illustrations represent a left rotation, while the bottom four illustrations represent a right rotation. It should be noted that you might not always have a pointer to the nodes we need to retrieve size information from. For the pictures, I haven't reflected this because something like that is a low-level implementation detail in the code.

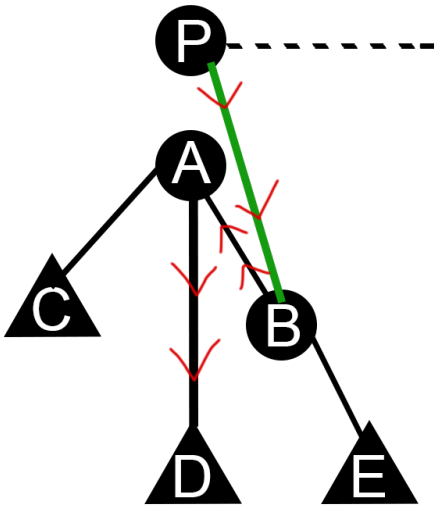
This process maintains the size invariant because A's size is always subtracting the size subtree of the child it detaches from and adding the size of the child subtree that its attaching to. The size property updates reflect the actual updates in the relationship between the nodes. We only ever need to worry about the nodes that are connected to the child being rotated since the rest of the nodes are accounted for in the size of the child's children's' subtrees, because of this, the rotation edge updates are always going to perform a constant number of operations and will not be dependent on the input size of the function.



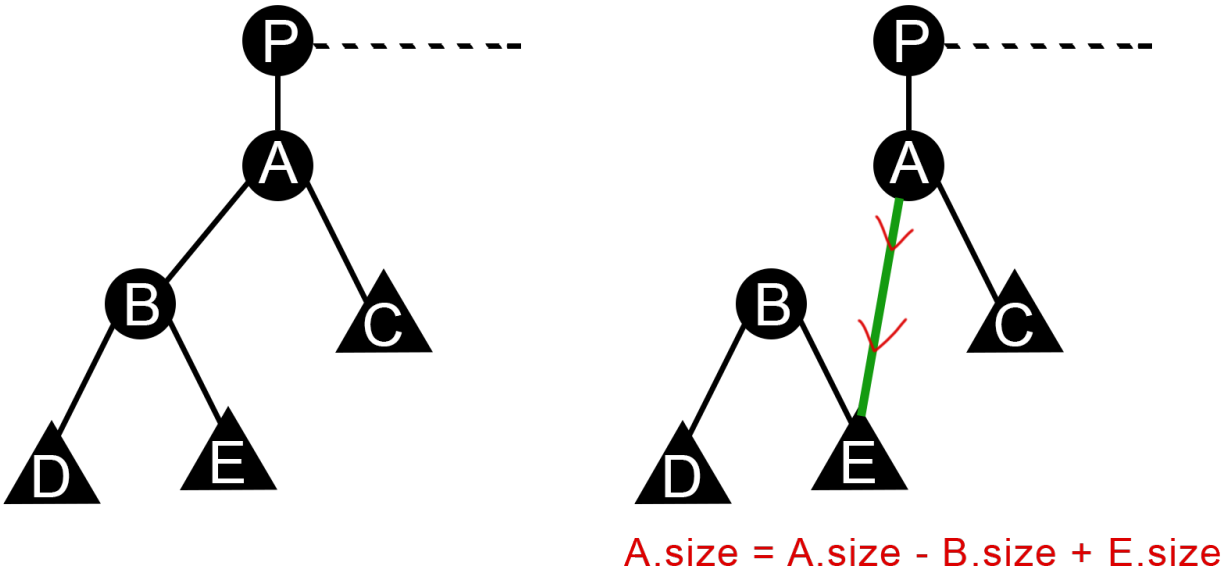


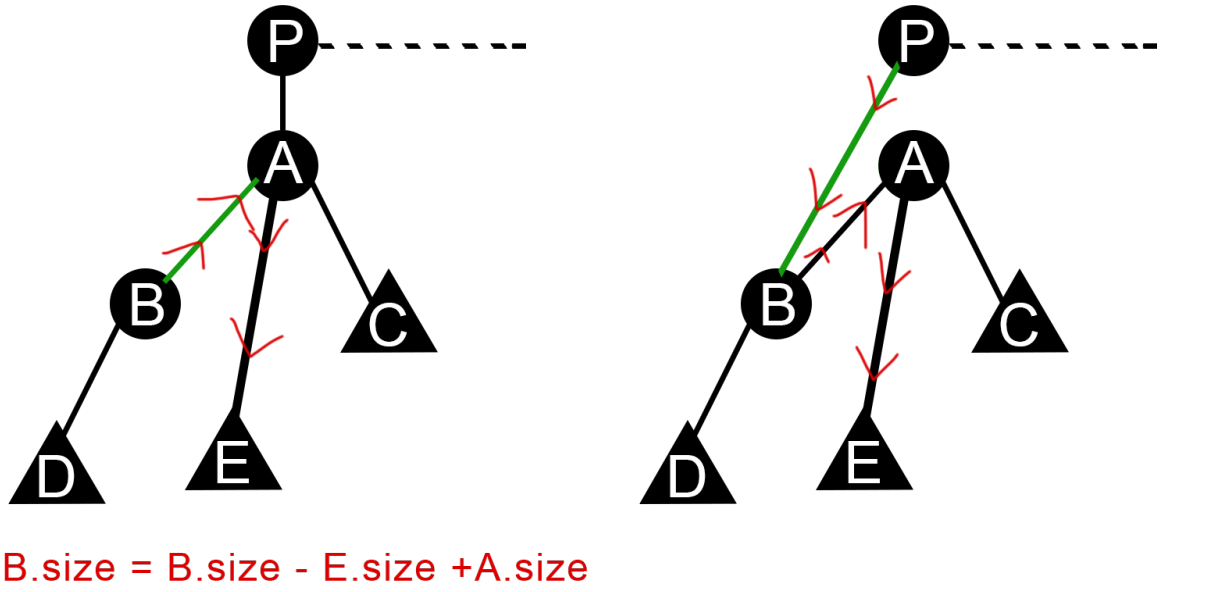
A.size = A.size - B.size + D.size





B.size = B.size - D.size + A.size





(c) Implement `rotate` in size-augmented BSTs in Python in the stub we have given you.
Implemented in coding portion

Food for thought (do read - it's an important take-away from this problem): This problem concerns size-augmented binary search trees. In lecture, we discussed AVL trees, which are balanced binary search trees where every vertex contains an additional *height* attribute containing the length of the longest path from the vertex to a leaf (height-augmented). Additionally, every pair of siblings in the tree have heights differing by at most 1, so the tree is height-balanced. Note that if we augment a binary search tree both by size (as in the above problem) and by height (and use it to maintain the AVL property), then we create a dynamic data structure able to perform `search`, `insert`, and `select` all in time $O(\log n)$.