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: Cory Zimmerman
Collaborators: None
No. of late days used on previous psets: 0
No. of late days used after including this pset: 0

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).

Consider the following algorithm for $AreaOfConvexPolygon$:

```
function $AreaOfConvexPolygon$(points: $(x_k, y_k)$ list): returns $r \in \mathbb{R}$
let $sortable$ be an empty list of $(K, V)$ pairs where $K \in \mathbf{R}$ and $V \in \mathbf{R}^2$
foreach $i = 0, \dots, length(points) - 1$ do
    let $polarAngle_i = calcPolarAngle(points_i)$;
    $sortable_i = (polarAngle_i, points_i)$;
$sort(sortable)$;
let $area = 0$;
foreach $i = 0, \dots, length(points) - 1$ do
    let $point1 = sortable_{i,V}$;
    let $point2 = sortable_{i+1,V}$ if $i < points - 1$ else $sortable_{0,V}$;
    let $point3 = (0, 0)$;
    let $a_i = distance(point1, point2)$;
    let $b_i = distance(point2, point3)$;
    let $c_i = distance(point3, point1)$;
    let $s_i = \frac{a_i + b_i + c_i}{2}$;
    let $triangleArea = \sqrt{s_i(s_i - a_i)(s_i - b_1)(s_i - c_i)}$;
    $area = area + triangleArea$;
return $area$;
```

This algorithm performs the following:

i. Indexes every point by its polar angle.
ii. Sorts the input by polar angle such that the output lists the coordinates in counter clockwise order.
iii. For every pair of neighboring coordinates (relating the last to the first), calculates the angle of the triangle formed by the first point, its neighbor, and the origin. Adds that area to the total area.
iv. Returns the total area.

Proof of correctness:
If the input is a valid list of points in $\mathbb{R}^2$ forming the vertices of a convex polygon containing the origin, this algorithm returns the area of the polygon formed by the points
Assert by definition of polygon that the input has at-least three vertices.
Assume the correctness of oracles *length*, *calcPolarAngle*, and *distance*. The formulas for *calcPolarAngle* and *distance* are given above and are guaranteed to be constant-time calculations. Assert *length* is also a known constant time operation.
Let the correctness of $AreaOfConvexPolygon$ reduce to the correctness of *sort*. If the inputs are sorted correctly by polar angle, the resulting (x, y) coordinate values are sorted in counterclockwise order.
Assert that the convex polygon can be entirely partitioned into triangles where each triangle contains the origin, a vertex (x_i, y_i), and its sorted neighbor, (x_{i+1}, y_{i+1}) where the $n - 1$th vertex is instead neighbored by the 0th vertex.

By calculating and summing the area of every triangle in the partition, the correct area of the polygon is always returned.
Because every valid input to the algorithm produces a valid output that solves the *AreaOfConvexPolygon* problem, this is a correct solution as long as the solution to *sort* is also correct.

Runtime: The algorithm for *AreaOfConvexPolygon* iterates a list of length n twice, so its runtime is at-least $O(n)$. I assert from in-class discussion and the information given by the question that the oracles for (1) determining the length of a list, (2) calculating the polar angle of an (x, y) coordinate, and (3) calculating the distance between points are known constant time algorithms.
Thus, the algorithm for *AreaOfConvexPolygon* runs in time complexity $O(n)$ plus the time complexity for the oracle for sorting, which *AreaOfConvexPolygon* reduces to. This allows the conclusion that $AreaOfConvexPolygon \leq_{O(n),n} Sorting$.

(b) Deduce that AreaOfConvexPolygon can be solved in time $O(n \log n)$.

Assert from the reasoning above that the algorithm for *AreaOfConvexPolygon* is a linear-time function that reduces to the sorting problem.
On an infinite universe of keys, the sorting problem has a known algorithmic time complexity of $O(n \log n)$. For example, the merge sort algorithm solves the sorting problem and runs in $O(n \log n)$ time.
Thus, the runtime of *AreaOfConvexPolygon* can be expressed as $O(n) + O(n \log n)$. Because the growth rate of $n \log n$ has mathematically greater magnitude than n as n approaches infinity, the $O(n)$ term can be dropped from the complexity expression, leaving the algorithm with $O(n \log n)$ time complexity.

(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.

To show the claim, work backwards, constructing the runtime from the given information:

- (Given) Γ can be solved in $T(s)$ time where s is the input size to T.
- Because T's input s as a reduction from Π will never exceed $h(n)$ where n is the input size to Π, $T(s) \leq T(h(n))$. So, Γ can be solved in time $T(h(n))$.
- Π calls Γ at most $k(n)$ times. Because the time for Γ is at most $T(h(n))$, the time for Γ when used as a reduction from Π is at most $k(n) \cdot T(h(n))$.
- Ignoring the reduction's time, Π itself runs in at most $g(n)$ time.
- Adding these together, the upper time bound of Π and its reduction Γ is at most $g(n) + k(n) \cdot T(h(n))$.
- Thus the runtime of Π is $O(g(n) + k(n) \cdot T(h(n)))$.

(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

[1]This problem is meant to be done based on your enjoyment/interest and only if you have time. It won't make a

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.

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.

[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).

`select`:
Select had a correctness error. It was properly determining if the current node was the target node or if the left child tree should be searched. However, when deciding to recursively search the right child, the target index was being not updated.
When descending to the right child, the target index should be decreased by the size of the left tree and the current node. That way, `select` looks for the proper index in the reduced subtree. Otherwise, `select` never finds a target belonging to the right child tree.

```
    def select(self, ind: int) -> Self | None:
        left_size = 0
        if self.left is not None:
            left_size = self.left.size
        if ind == left_size:
            return self
        if left_size > ind and self.left is not None:
            return self.left.select(ind)
        if left_size < ind and self.right is not None:
            return self.right.select(ind - left_size - 1)
        # ^ Fix here
        return None
```

`search`:
Search is correct.

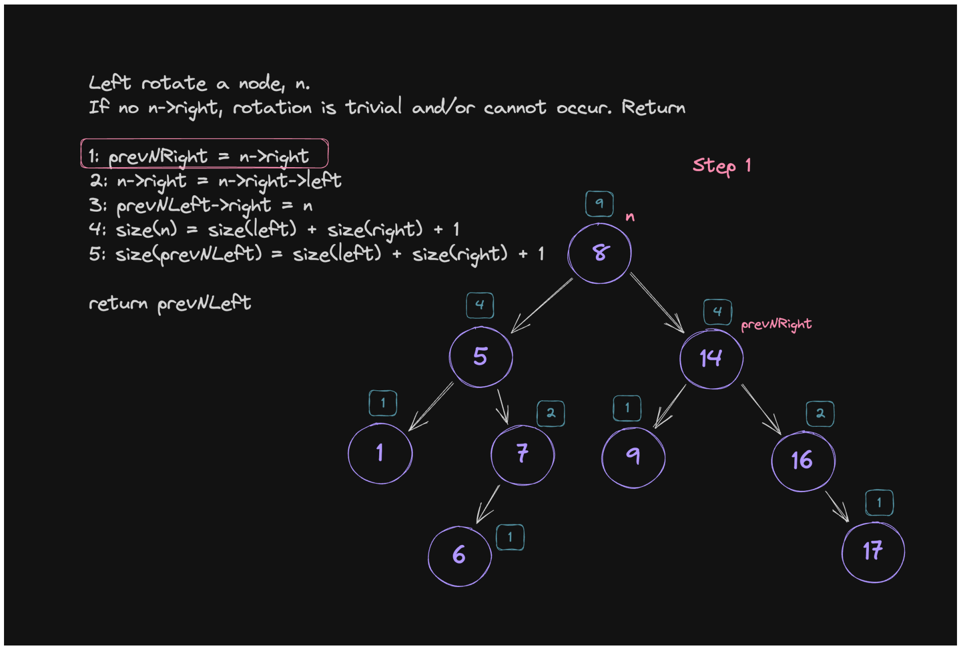
`insert`:
The old insert updated sizes by recalculating the size of every subtree at the end of insertion. However, we know that every time we insert a child node into the current subtree, we're increasing the current subtree's size by one. So, I cut out the recalculation at the end and added increments every time we insert a node into a subtree. (We don't increment when assigning a key to a leaf because the tree definition defaults size to 1 on instantiation).

```
    def insert(self, key: int) -> Self | None:
        if self.key is None:
            self.key = key
        elif self.key > key:
            if self.left is None:
                self.left = BinarySearchTree(self.debugger)
            self.left.insert(key)
            self.size += 1
            # ^ added
        elif self.key < key:
            if self.right is None:
                self.right = BinarySearchTree(self.debugger)
            self.right.insert(key)
            self.size += 1
            # ^ added
        # self.calculate_sizes()
        # ^ removed
        return self
```

(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.)

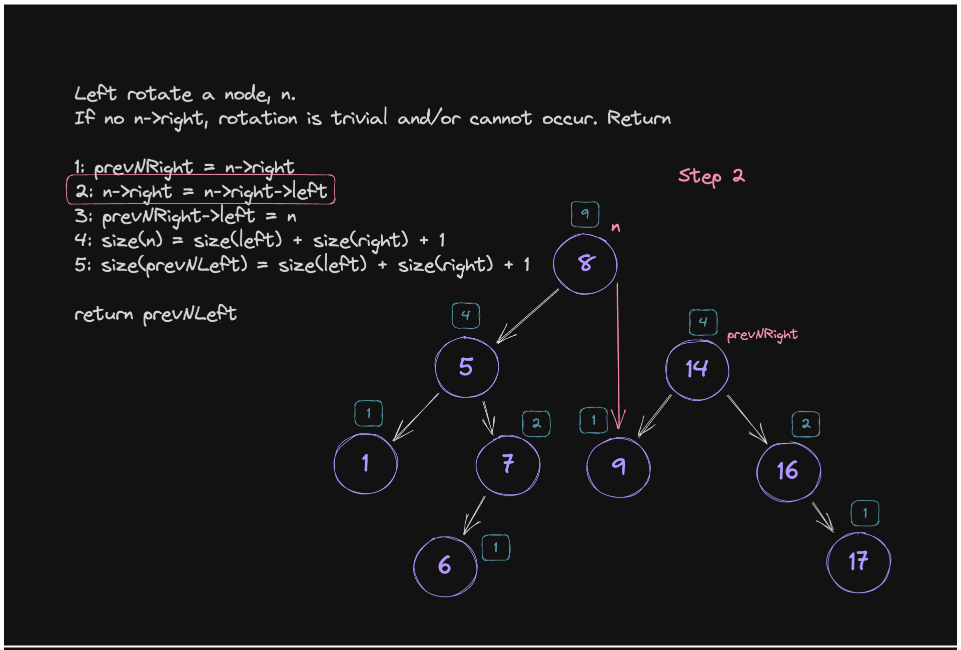
The pictures below illustrate two rotation algorithms, one for a left rotation and one for a right rotation. After the pictures, I'll analyze runtime and preservation of size augmentation.

Left rotation:



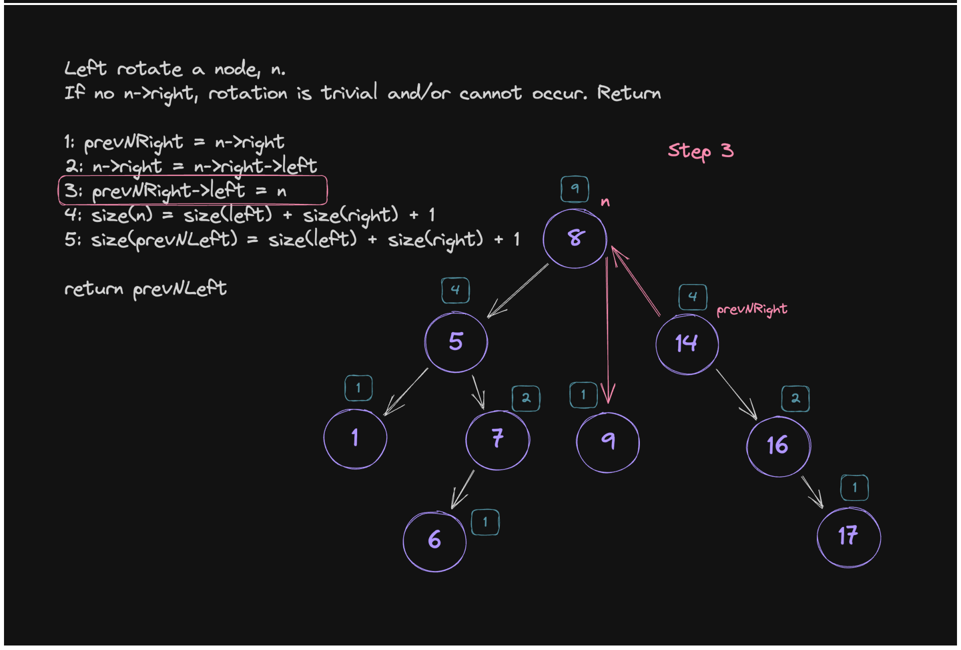
Left rotate a node, n.
If no n->right, rotation is trivial and/or cannot occur. Return

```
1: prevNRight = n->right
2: n->right = n->right->left
3: prevNLeft->right = n
4: size(n) = size(left) + size(right) + 1
5: size(prevNLeft) = size(left) + size(right) + 1

return prevNLeft
```



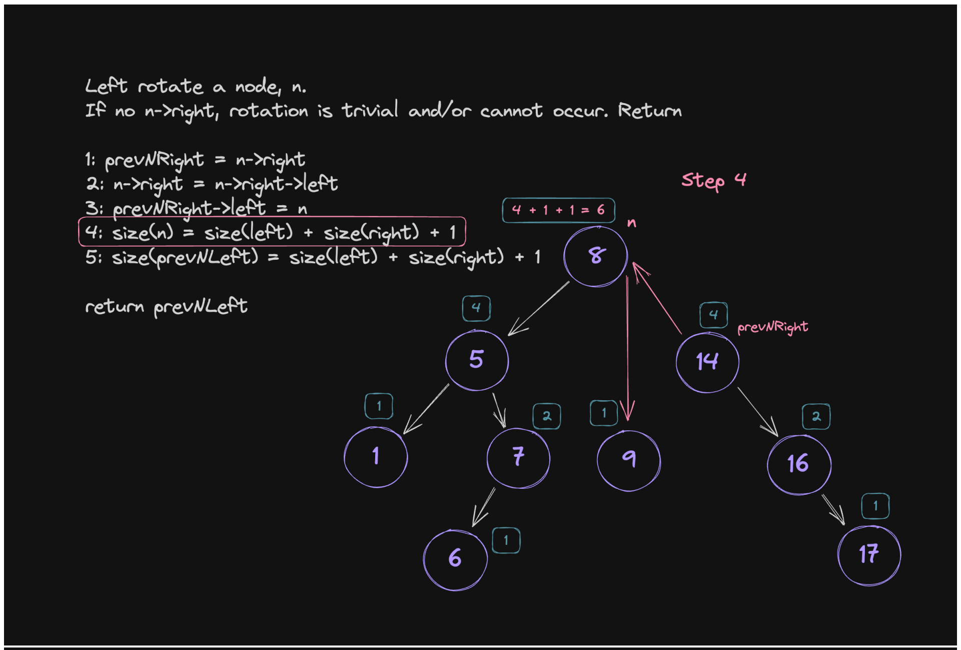
Left rotate a node, n.
If no n->right, rotation is trivial and/or cannot occur. Return

```
1: prevNRight = n->right
2: n->right = n->right->left
3: prevNRight->left = n
4: size(n) = size(left) + size(right) + 1
5: size(prevNLeft) = size(left) + size(right) + 1

return prevNLeft
```



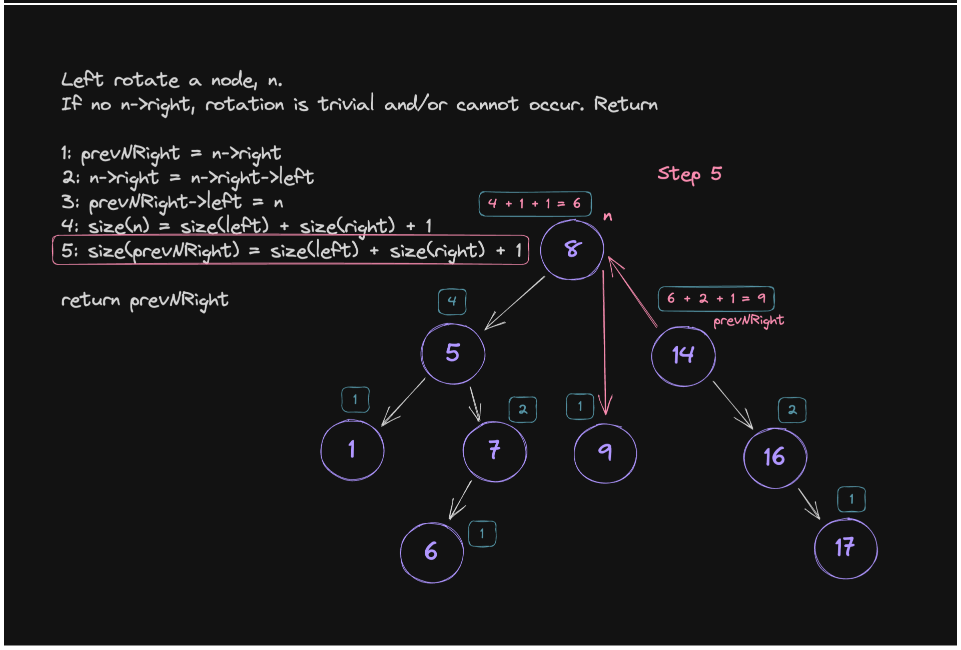
Left rotate a node, n.
If no n->right, rotation is trivial and/or cannot occur. Return

```
1: prevNRight = n->right
2: n->right = n->right->left
3: prevNRight->left = n
4: size(n) = size(left) + size(right) + 1
5: size(prevNLeft) = size(left) + size(right) + 1

return prevNLeft
```



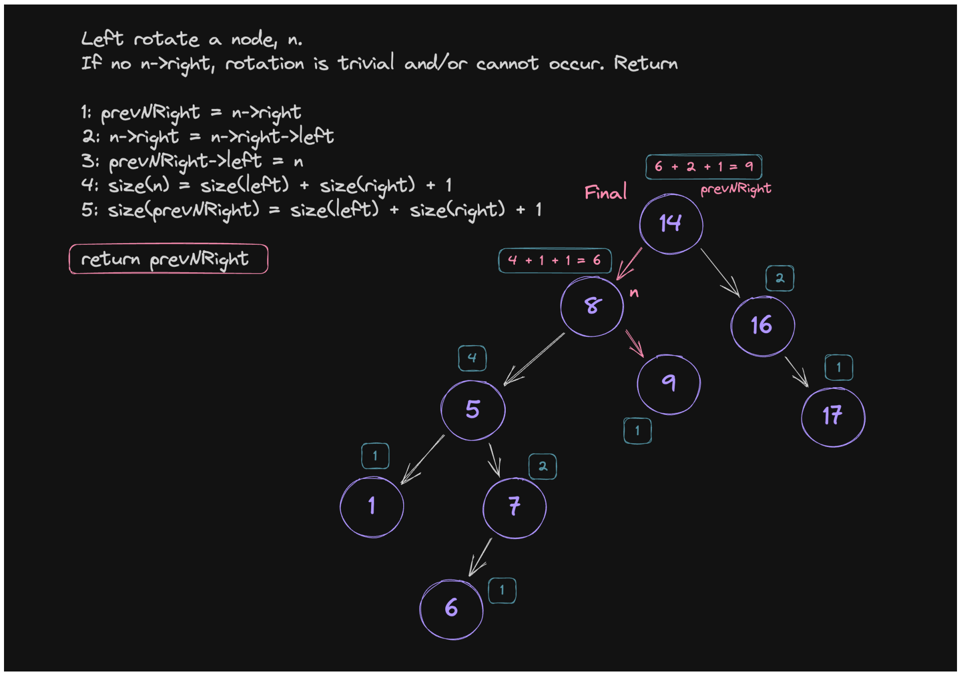
Left rotate a node, n.
If no n->right, rotation is trivial and/or cannot occur. Return

```
1: prevNRight = n->right
2: n->right = n->right->left
3: prevNRight->left = n
4: size(n) = size(left) + size(right) + 1
5: size(prevNLeft) = size(left) + size(right) + 1

return prevNLeft
```



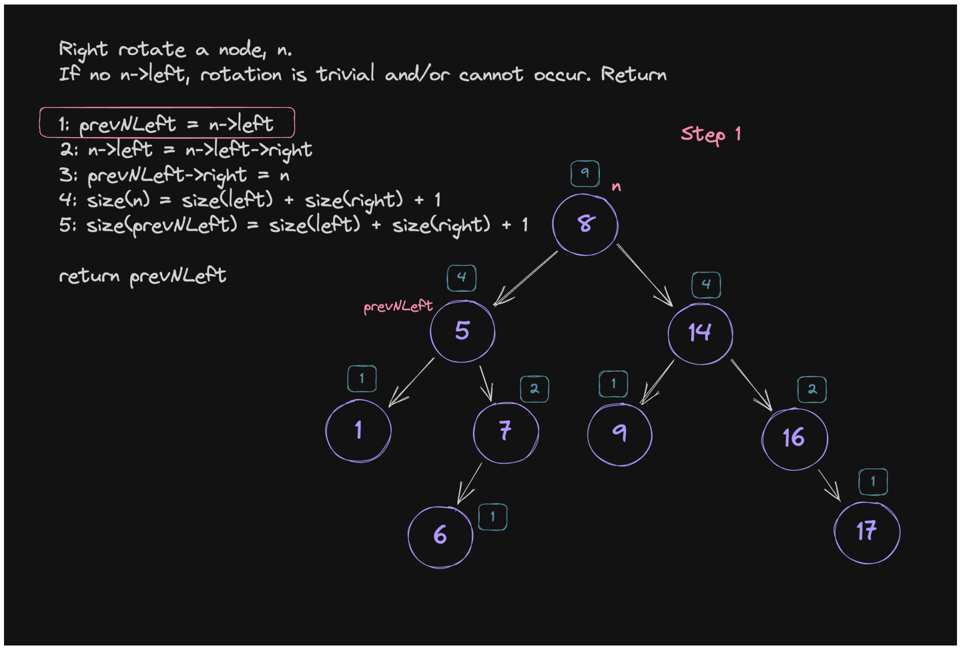
Left rotate a node, n.
If no n->right, rotation is trivial and/or cannot occur. Return

```
1: prevNRight = n->right
2: n->right = n->right->left
3: prevNRight->left = n
4: size(n) = size(left) + size(right) + 1
5: size(prevNRight) = size(left) + size(right) + 1

return prevNRight
```

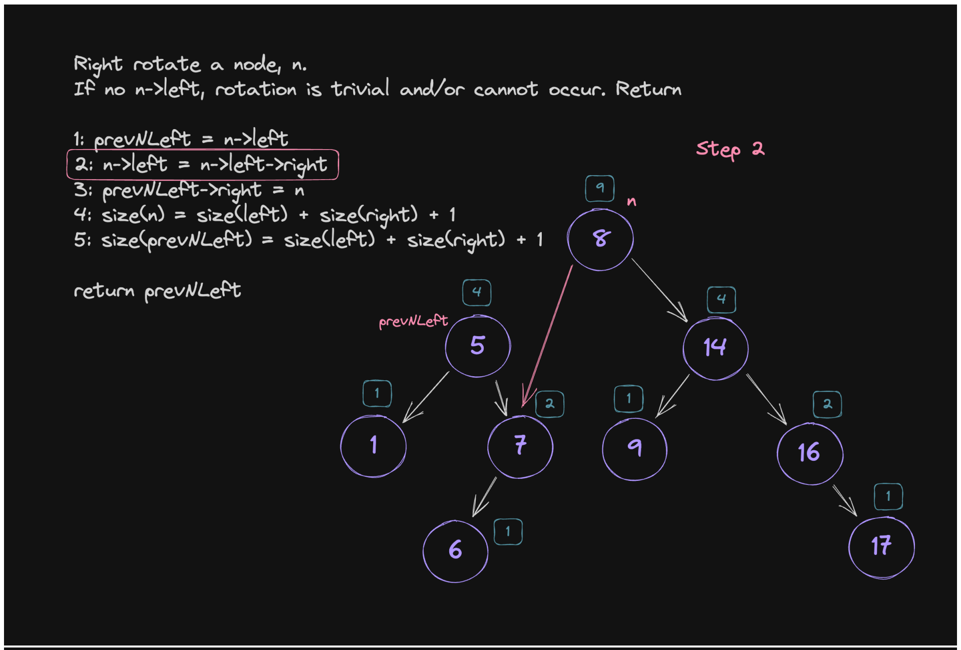


```
Left rotate a node, n.
If no n->right, rotation is trivial and/or cannot occur. Return

1: prevNRight = n->right
2: n->right = n->right->left
3: prevNRight->left = n
4: size(n) = size(left) + size(right) + 1
5: size(prevNRight) = size(left) + size(right) + 1

return prevNRight
```

Right rotation:



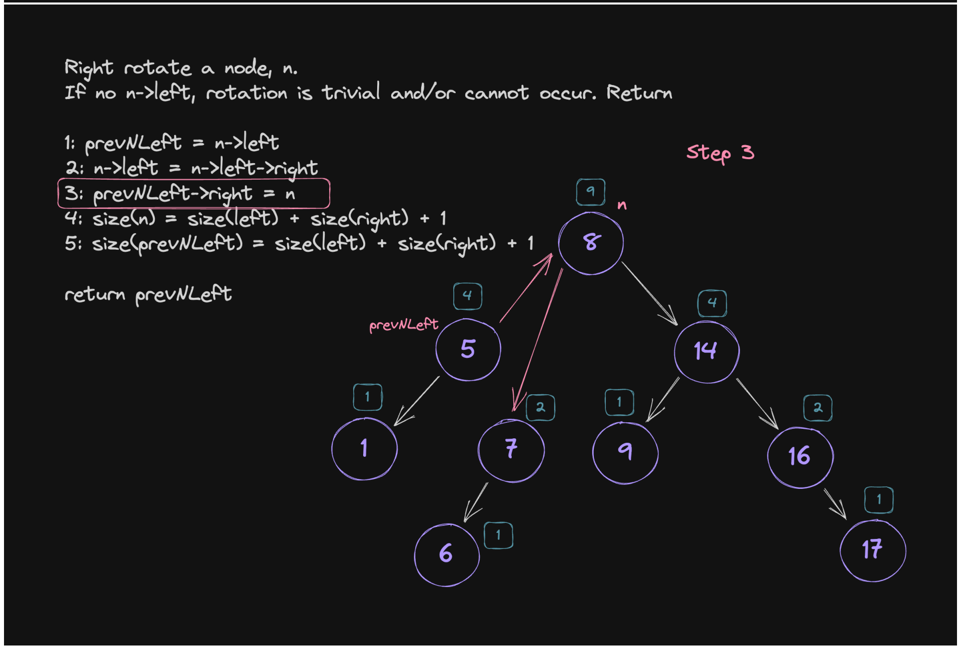
```
Right rotate a node, n.
If no n->left, rotation is trivial and/or cannot occur. Return

1: prevNLeft = n->left
2: n->left = n->left->right
3: prevNLeft->right = n
4: size(n) = size(left) + size(right) + 1
5: size(prevNLeft) = size(left) + size(right) + 1

return prevNLeft
```



Right rotate a node, n.
If no n->left, rotation is trivial and/or cannot occur. Return

```
1: prevNLeft = n->left
2: n->left = n->left->right
3: prevNLeft->right = n
4: size(n) = size(left) + size(right) + 1
5: size(prevNLeft) = size(left) + size(right) + 1

return prevNLeft
```

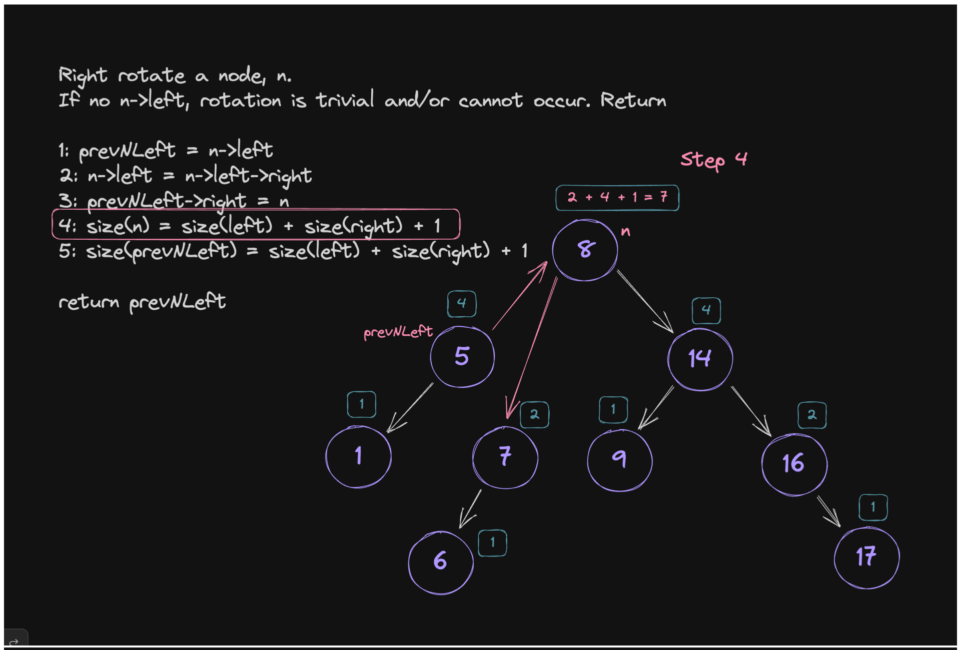
Step 2



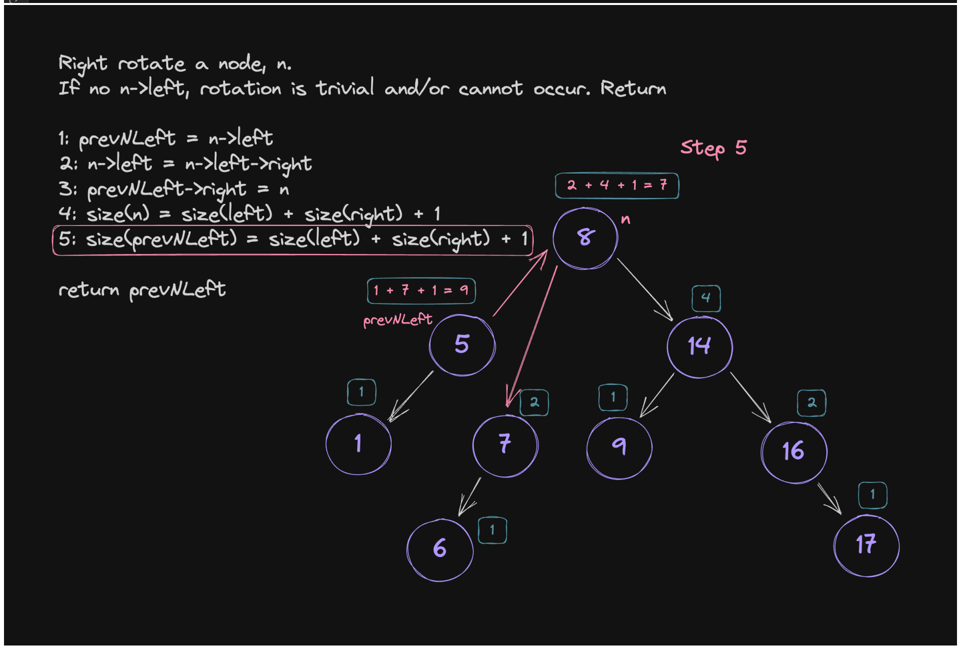
Right rotate a node, n.
If no n->left, rotation is trivial and/or cannot occur. Return

```
1: prevNLeft = n->left
2: n->left = n->left->right
3: prevNLeft->right = n
4: size(n) = size(left) + size(right) + 1
5: size(prevNLeft) = size(left) + size(right) + 1

return prevNLeft
```

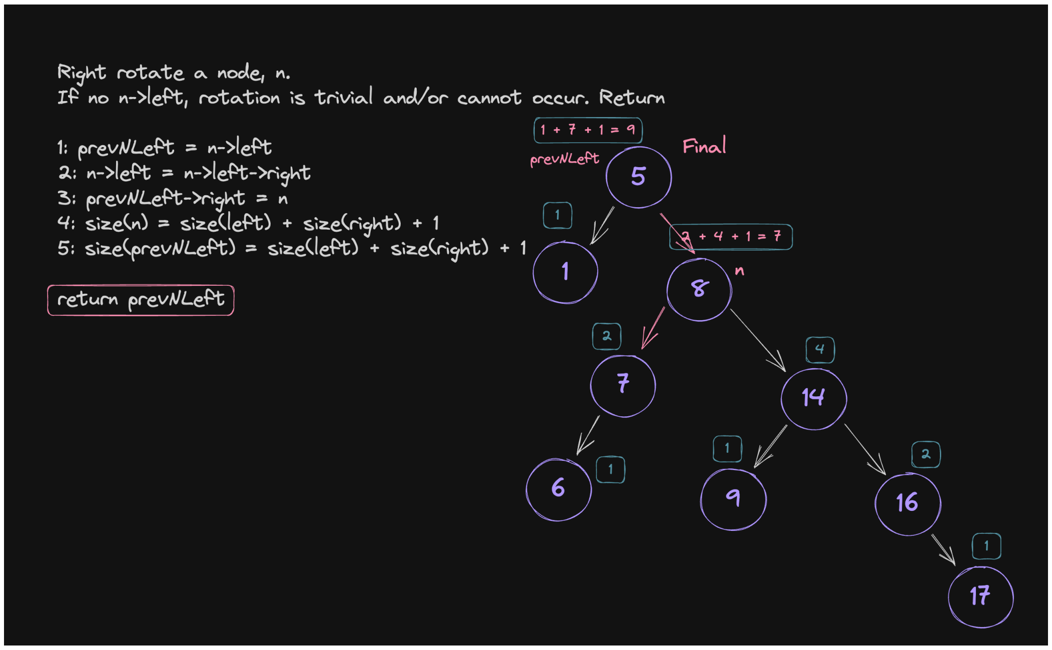
Step 3



```
Right rotate a node, n.
If no n->left, rotation is trivial and/or cannot occur. Return

1: prevNLeft = n->left
2: n->left = n->left->right
3: prevNLeft->right = n
4: size(n) = size(left) + size(right) + 1
5: size(prevNLeft) = size(left) + size(right) + 1

return prevNLeft
```



```
Right rotate a node, n.
If no n->left, rotation is trivial and/or cannot occur. Return

1: prevNLeft = n->left
2: n->left = n->left->right
3: prevNLeft->right = n
4: size(n) = size(left) + size(right) + 1
5: size(prevNLeft) = size(left) + size(right) + 1

return prevNLeft
```



```
Right rotate a node, n.
If no n->left, rotation is trivial and/or cannot occur. Return

1: prevNLeft = n->left
2: n->left = n->left->right
3: prevNLeft->right = n
4: size(n) = size(left) + size(right) + 1
5: size(prevNLeft) = size(left) + size(right) + 1

return prevNLeft
```

O(1) size updates:
Given a BST with correct sizes, if my right or left rotation algorithms are performed, prove the augmented tree size invariant is preserved:
To simplify reasoning, notice that both algorithms operate on n, a given subtree root node, and $prevN...$, a child of n where ... is either $Left$ or $Right$. Let $prevN$ be a substitute for either $prevNLeft$ or $prevNRight$, as the reasoning is identical.
In both algorithms, the only nodes whose child pointers are manipulated are n and $prevN$. Thus, the only modified subtrees are those rooted at n and $prevN$. Assuming the input trees held correct sizing, this algorithm retains the correctness of every subtree not rooted at n or $prevN$.
After rotation, n becomes a child of $prevN$. So, every left or right child tree of n is an unmodified subtree. Thus, the size of n can be calculated by adding the sizes of its left and right subtrees plus one for itself. An empty child tree has size zero, and a non-empty child tree already tracks its size, so this is a constant time arithmetic operation.
The node n is a child of $prevN$ in the output tree. Because one child of $prevN$ is an untouched input tree (possibly empty) and the other child is n, we can now assert n also has a correct size and perform the same arithmetic addition as before to determine the new size of the subtree rooted at $prevN$
After these two $O(1)$ time updates to n and $prevN$ are performed, the tree's size augmentation invariant is restored.

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

```python
def __recalc_size(self) -> int:
    sum = 0
    if self.left:
        sum += self.left.size
    if self.right:
        sum += self.right.size
    return sum

def __rotate_left(self) -> Self:
    prev_n_right = self.right
    assert prev_n_right is not None

    # redirect pointers
    self.right = self.right.left
    prev_n_right.left = self

    # recalculate sizes
    self.size = self.__recalc_size()
    prev_n_right.size = prev_n_right.__recalc_size()

    return prev_n_right

def __rotate_right(self) -> Self:
    prev_n_left = self.left
    assert prev_n_left is not None

    # redirect pointers
    self.left = self.left.right
    prev_n_left.right = self

    # recalculate sizes
    self.size = self.__recalc_size()
    prev_n_left.size = prev_n_left.__recalc_size()

    return prev_n_left

def rotate(self, direction, child_side):
    assert child_side == "L" or child_side == "R"
    assert direction == "L" or direction == "R"

    if child_side == "L":
        if direction == "L":
            self.left = self.left.__rotate_left()
        else:
                self.left = self.left.__rotate_right()
        else:
            if direction == "L":
                self.right = self.right.__rotate_left()
            else:
                self.right = self.right.__rotate_right()
        return self
```

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 $O(\log n)$.