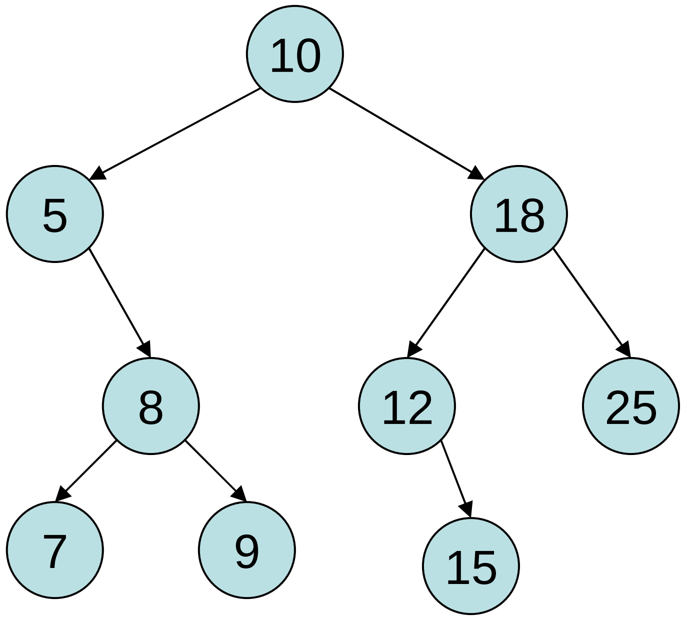
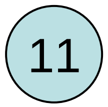
Bucknell University
Computer Science

CSCI 311 - Data Structures

Randomized Binary Search Trees

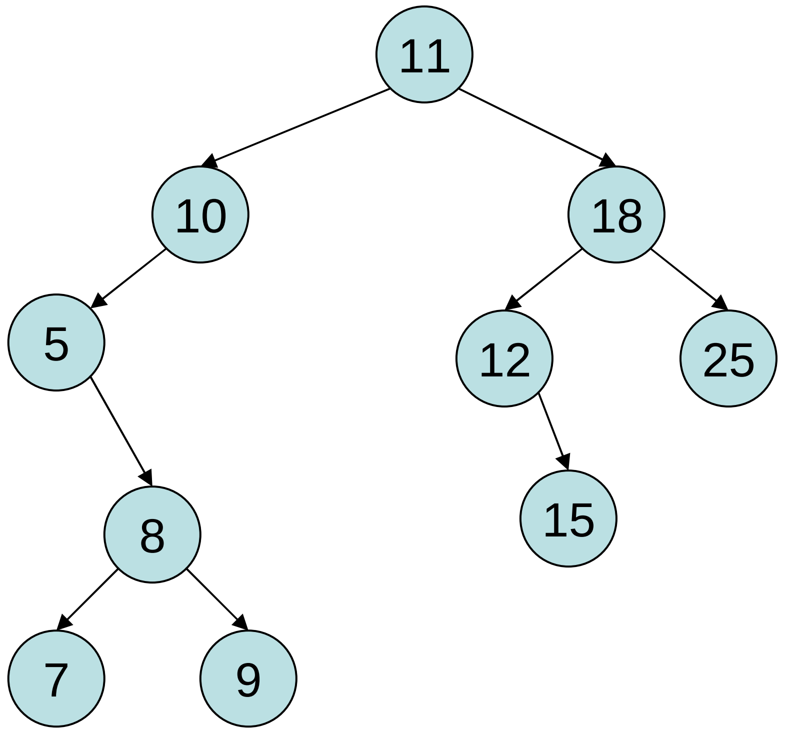
Insertion at the root





Make the new item the root node of a new tree. The old root will be the left subtree of the new item. The right subtree of the old root will be the right subtree of the new root.

Insertion at the root

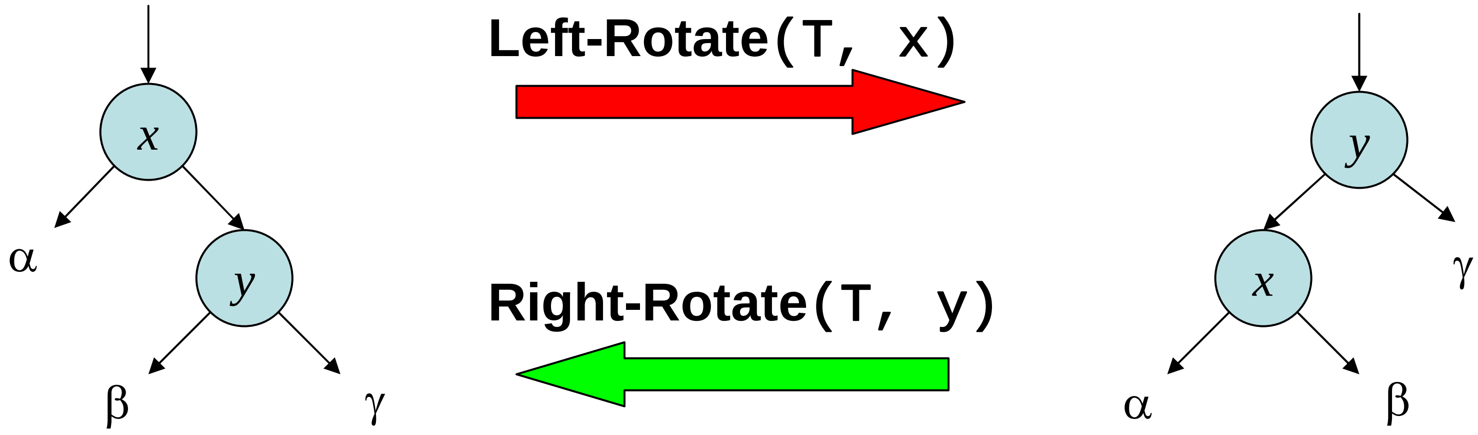


A potential problem with this rationale is that you may end up with a tree that is getting needlessly deeper and deeper, more and more *unbalanced*.

What is needed is a mechanism to restructure the tree after each insertion so that it doesn't ***degenerate***.

It is inefficient to do this globally, but perhaps we can do something locally that is not bad for performance.

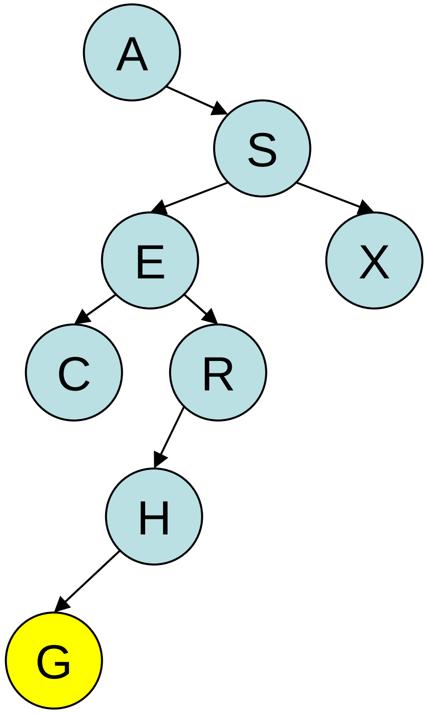
Rotations



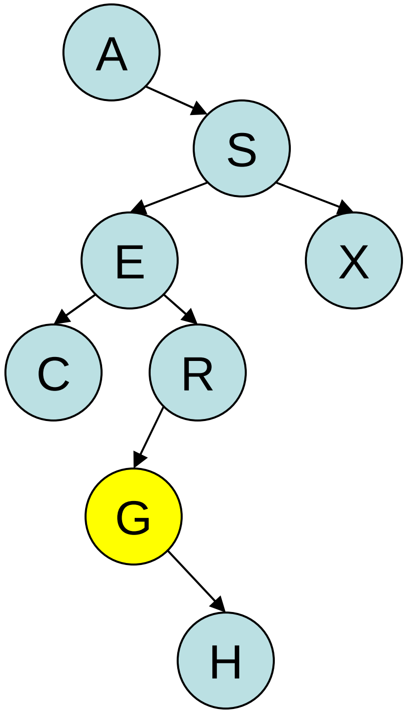
A rotation is a *local* change involving two nodes and three links. Note that although it restructures a portion of the tree, it does not change the tree's global properties.

It is easy to verify that the ***inorder traversal*** of the rotated tree is the same as the original's.

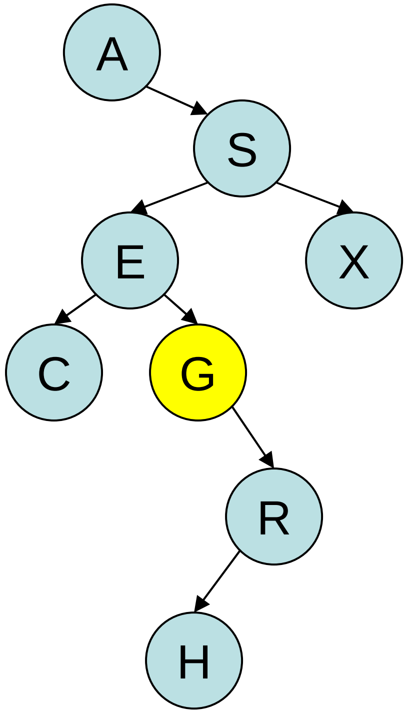
Insertion at the root



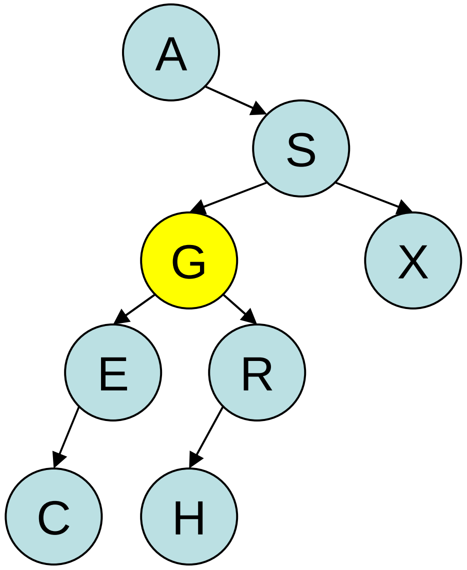
Right-Rotate(T, H)



Right-Rotate(T, R)

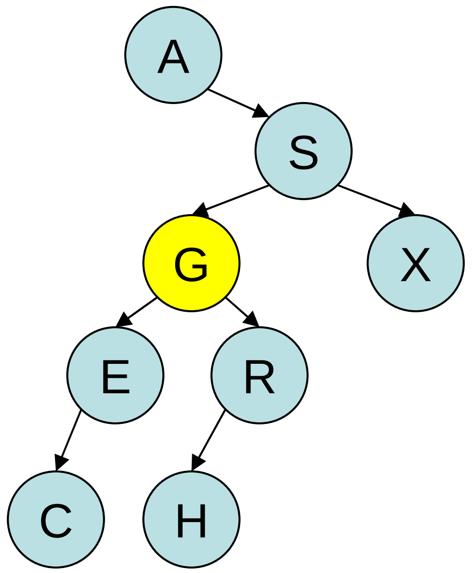


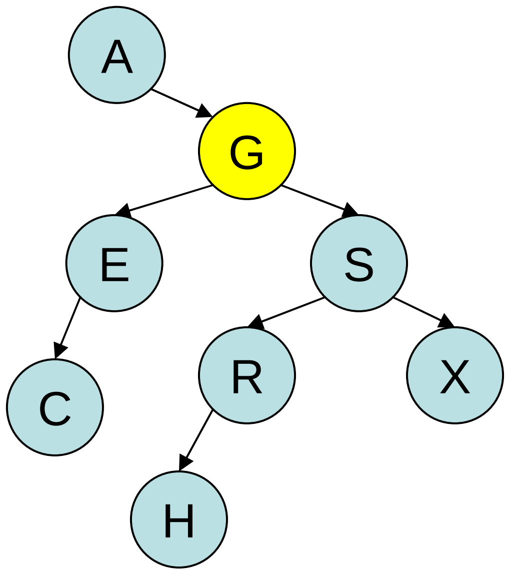
Left-Rotate(T, E)

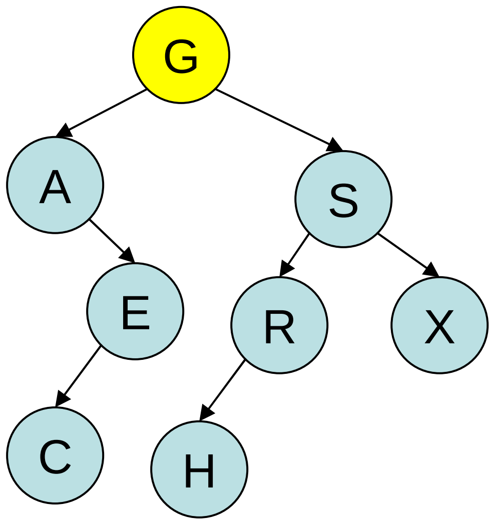


Right-Rotate(T, S)

Insertion at the root







Right-Rotate(T, S)

Left-Rotate(T, A)

Randomly Built BSTs

The basic BST operations run in O(*h*), where *h* is the height of the tree. It is important to note that h depends on the order in which items are inserted in the tree.

Question: What would the tree look like if the keys were inserted in strictly increasing order?

If the order of item insertion were to follow equally likely permutations of the possible (distinct) keys, the expected height of the tree with *n* nodes would be lg *n*. In the next slide we will show how to make any insertion order look random, and thus give good expected (average) performance.

Note, however, that even when the key insertion order is random, there is ***no guarantee*** that the height of the tree will be lg *N* - we may get a bad permutation. Therefore we cannot guarantee that operations on the BST are O(lg *n*).

Randomized Insertion

We can make it look like the order of key insertion is random by choosing the insertion point at random. Say that the number of nodes in the current subtree is *k* before the insertion.

Simple recursive procedure: When inserting at the root of a subtree, toss a ***biased*** coin:

- With probability 1/(*k*+1), insert new key at the root of the current subtree using the algorithm given above (there is no further randomization for this insertion).
- With probability (*k*/*k*+1), recursively apply randomized insertion in the appropriate subtree.

$$\Pr\{k\text{th element (of } n\text{) ends up at root}\} = \underbrace{\frac{1}{k}}_{\text{heads for this element}} \underbrace{\frac{k}{k+1}\,\frac{k+1}{k+2}\cdots\frac{n-2}{n-1}\,\frac{n-1}{n}}_{\text{tails for all other elements}} = \frac{1}{n}$$

Performance: a mix of *N* insertions and searches will take O(*N* lg *N*) on average and O(N^2) in the worst case.