(1) (a) (4 points) Define the BST (Binary Search Tree) property.

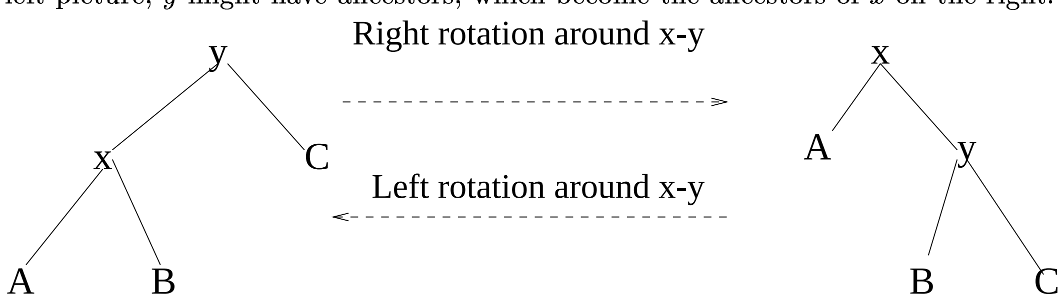
Solution: A binary tree is a BST if every node x has a value $key(x)$ such that

$$key(left(x)) \leq key(x) \qquad \text{and } key(right(x)) \geq key(x).$$

Of course, this is assuming that $left(x)$ and $right(x)$ exist. If either one does not, then the corresponding inequality is not applicable.

(b) (4 points) Show that if you perform a rotation around any edge of a BST tree then the resulting tree is a BST tree. You may want to use a picture.

Solution: Consider the following rotation, where x, y are nodes and A, B, C are subtrees. We assume the tree on the left is a BST tree and prove that the tree on the right is also a BST tree. The picture focuses on a subtree of a potentially larger tree; that is, in the left picture, y might have ancestors, which become the ancestors of x on the right.



The subtrees A, B, C don't change, so they retain the BST property. Let a, b, c be the roots of A, B, C respectively. From the left tree, we know $key(a) \leq key(x), key(b) \leq key(y), key(c) \geq key(y)$. Therefore, it is ok to have A as the left subtree of x, B as the left subtree of y and C as the right subtree of y. We also know $key(x) \leq key(y)$ so it is ok to have y is the right child of x. Finally, the whole subtree in the picture contains exactly the same elements after the rotation as it did before (they just get rearranged); therefore the parent of y retains the BST property after the rotation.

(2) A *ternary counter* is a string of k "trits" $t_{k-1}t_k \dots t_0$, each of which can be 0, 1, or 2. As with a binary counter, we can perform the operation `INCREMENT` on a ternary counter. If we start with every trit equal to 0, then after n `INCREMENT`s, the counter holds the number n written in base 3. For example, if $k = 4$ and $n = 6$, we have

t_3	t_2	t_1	t_0
0	0	0	0
0	0	0	1
0	0	0	2
0	0	1	0
0	0	1	1
0	0	1	2
0	0	2	0

The cost of each `INCREMENT` is the number of trits that change. We are interested in the worst-case sequence complexity, $WCSC(n)$, of performing n `INCREMENT`s starting form all 0's.

(a) (8 points) Compute $WCSC(n)$ using the aggregate method. You may use the fact that $\sum_{i=0}^{\infty} 1/3^i = 3/2$.

Solution: As in lecture, notice that t_i changes every 3^i increments. Therefore, after n increments, we have

$$WCSC(n) = \sum_{i=0}^{\ell} n/3^i,$$

where ℓ is the index of the largest trit that ever becomes non-zero. Therefore,

$$WCSC(n) \leq n \sum_{i=0}^{\infty} 1/3^i = 3n/2.$$

(b) (8 points) Compute $WCSC(n)$ using the accounting method. Make sure to specify the charge for each `INCREMENT` and the credit invariant.

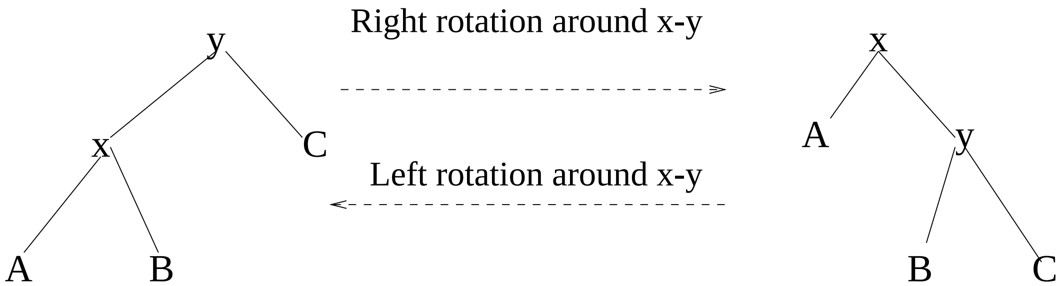
Solution: We'll charge 3/2 for each increment. The credit invariant will be that each trit with value 1 will have credit 1/2 and every trit with value 2 will have credit 1. We can achieve this credit invariant as follows: in each increment, exactly one trit will increase in value. We use 1 unit of the charge to pay for increasing this trit, and store the extra 1/2 with the trit. When we need to change a trit with value 2 to 0, we can use the 1 unit of credit stored at that trit.

Therefore, $WCSC(n) \leq$ Total Charge $\leq 3n/2$.

(3) We want to augment Red-Black Trees so that each node x stores a number $x.height$, the height of the subtree rooted at x. Briefly explain how to modify the following standard operations to maintain this information at every node. The modifications should not change their running times (in Θ-notation).

(a) (5 points) Rotation:

Solution: Consider the following picture again (going from left to right):



Again, let a, b, c be the roots of A, B, C, respectively. We simply reset $y.height$ to $\max\{b.height, c.height\} + 1$ and reset $x.height$ to $\max\{a.height, y.height\} + 1$. This takes constant time since we look at only a constant number of nodes.

(b) (5 points) `BST-INSERT`:

Solution: If we insert a new node x, it gets added to the tree as a leaf. Assign $x.height := 0$. Starting with x's parent, visit each of the ancestors of x. For each such ancestor y, set $y.height$ to $\max\{left(y).height, right(y).height\} + 1$. This takes time $O(\log n)$ since we follow one path from a leaf to the root.

(c) (5 points) `BST-DELETE`:

Solution: Let x be the node that gets removed by `BST-DELETE`. Again, starting with x's parent, visit each of the ancestors of x. For each such ancestor y, set $y.height$ to $\max\{left(y).height, right(y).height\} + 1$. This takes time $O(\log n)$ since we follow one path from a leaf to the root.

(4) Consider the following procedure for testing whether a given array of integers is sorted:

```
boolean IsSorted ( integer A[], integer n )
  For i = 1 to n-1 do
    If (A[i] > A[i+1]) then
        Return False
  Return True
End
```

Throughout this question, we will measure the running time in terms of the number of comparisons that IsSorted performs.

(a) (4 points) What is the worst-case running time, $T_{wc}(n)$, of IsSorted on an array of length n? Justify your answer.

Solution: $T_{wc}(n) = n - 1 \in \Theta(n)$. If the array is sorted, then the loop will never break, so we'll execute the comparison $n - 1$ times.

(b) (2 points) Consider the sample space S_n of all permutations of $(1, 2, \ldots, n)$, with the uniform distribution (that is, each permutation is equally likely). Let A be a random array from S_n. Let $B_{i,j}$ be the event that $A[i] > A[j]$. What is the value of $\Pr(B_{i,j})$?

Solution: $\Pr(B_{i,j}) = 1/2$.

(c) (6 points) Let $t(A)$ be the running time of IsSorted on array A. Express $\Pr(t(A) = k)$ in terms of the events $B_{1,2}, B_{2,3}, \ldots, B_{k,k+1}$. Explain why this is at most $1/2^{k-1}$. Is it strictly less than $1/2^{k-1}$?

Solution:

$$\Pr(t(A) = k) = \Pr(\neg B_{1,2} \cap \neg B_{2,3} \cap \ldots \cap \neg B_{k-1,k} \cap B_{k,k+1}).$$

First notice that

$$\begin{aligned} \Pr(t(A) = k) &< \Pr(\neg B_{1,2} \cap \neg B_{2,3} \cap \ldots \cap \neg B_{k-1,k}) \\ &< \Pr(\neg B_{1,2}) \cdot \Pr(\neg B_{2,3} | \neg B_{1,2}) \cdots \Pr(\neg B_{k-1,k} | \neg B_{1,2}, \neg B_{2,3}, \ldots, \neg B_{k-2,k-1}). \end{aligned}$$

Intuitively, if we know that $A[i]$ is bigger than all the previous elements, then that makes it more likely to be bigger than $A[i+1]$. More formally, this means that $\Pr(\neg B_{i,i+1} | \neg B_{1,2}, \ldots, \neg B_{i-1,i}) < 1/2$. Hence, $\Pr(t(A) = k) < 1/2^{k-1}$.

(d) (4 points) Compute $T_{avg}(n)$, the average-case running time of IsSorted over the sample space S_n. You may use the fact that $\sum_{k=1}^{\infty} k/c^{k-1} = O(1)$ for any constant $c > 1$.

Solution: We just need to calculate

$$\begin{aligned} T_{avg}(n) &= \textstyle\sum_{k=1}^{n-1} k \Pr(t(A) = k) \\ &\leq \textstyle\sum_{k=1}^{n-1} k/2^{k-1} \\ &\leq \textstyle\sum_{k=1}^{\infty} k/2^{k-1} \\ &= O(1). \end{aligned}$$

Since it obviously takes at least 1 comparison to test if A is sorted, $T_{avg}(n)$ is $\Theta(1)$.