CSC265 Homework 6 | Fall 2021
Name: Haojun Qiu | Student Number: 1006900622
Who I discussed with: Darcy Wang, Runshi Yang, Ted Deng

Notation for entire homework: For any node x in a binary search tree, we denote the number of nodes in the subtree rooted at x by $n(x)$, and the depth of x by $d(x)$.

Q1

Proof: We prove the contrapositive of the statement.

Let x be a node in the binary tree T. We assume that the binary tree is balanced at all proper ancestor of x. We will show that the depth of x is less than or equal to $\log_{3/2} n$.

Let $A = \{a \in T : a \text{ is an ancestor of } x\}$, and $P = \{x\} \cup A$. Firstly, we show a claim.

Claim: For all $y \in P$, $n(y) \leq (\frac{2}{3})^{d(y)} n$.

Base Case: We show the root $r \in P$ satisfies the statement. Since $d(r) = 0$, $n(r) = n = (\frac{2}{3})^0 n = (\frac{2}{3})^{d(r)} n$.

Induction step: Let p be any nodes in $P \setminus \{x\} = S$, we assume that $n(p) \leq (\frac{2}{3})^{d(p)} n$.
Let $u \in P$ be the child of p (u exists since $p \neq x$). We want to show that u satisfies $n(u) \leq (\frac{2}{3})^{d(u)} n$.

$$\begin{aligned}
n(u) &\leq \frac{2}{3} n(p) && \text{(since } T \text{ is balanced at } p\text{)} \\
&\leq \frac{2}{3}\left(\frac{2}{3}\right)^{d(p)} n && \text{(by induction hypothesis)} \\
&\leq \left(\frac{2}{3}\right)^{d(p)+1} n \\
&= \left(\frac{2}{3}\right)^{d(u)} n && \text{(since } u \text{ is } p\text{'s child)}
\end{aligned}$$

So, by the structural induction, the Claim holds. Therefore, in particular, $x \in P$ so $n(x) \leq (\frac{2}{3})^{d(x)} n$. Hence, since $1 \leq n(x)$, we know that $1 \leq (\frac{2}{3})^{d(x)} n$, which implies

$$\begin{aligned}
\frac{1}{n} &\leq \left(\frac{2}{3}\right)^{d(x)} \\
\log \frac{1}{n} &\leq d(x) \log \frac{2}{3} \\
-\log n &\leq -d(x) \log \frac{3}{2} \\
\frac{\log n}{\log \frac{3}{2}} &\geq d(x) \\
d(x) &\leq \log_{3/2} n,
\end{aligned}$$

and since depth is a natural number, the above inequality implies that $d(x) \leq \lfloor \log_{3/2} n \rfloor$. That is, the depth of x is less than or equal to $\lfloor \log_{3/2} n \rfloor$, as wanted. ■

Q2

(1) Firstly, we create an array A of size n, where n is number of nodes in the binary search tree T given (as stated in the question n is given).

(2) Then, we do an in-order tree walk, putting nodes into array A. After that, the array A stores (pointers to) all nodes in T sorted by key in non-decreasing order.

(3) Then, we use recursion to construct the desired tree using array A. The basic idea of my algorithm is that, we construct a tree by

- Setting the mid point of array A constructed in step 2, to be the root *root* of the new binary search tree

- Recursively calling the algorithm on the left-sub array to get the left subtree's root l (and set $root.left = l$), and similarly, recurse on the right-sub array to get the right subtree's root (and set $root.right = r$).
- The recursion stops (Base cases) when the sub-array being called on is empty or contains 1 node, then we return NIL or the node, respectively.

The above is actually the entire algorithm for this step. However, I think a pseudocode can help with the clarity of my algorithm so I also include it for this step as an kind of appendix. In the following code, instead of recursing on the array object, we pass the same array but change the index (s for start, e for end) to specify which part of the array is being recursed on.

```
CONSTRUCT-TREE(A, s, e)
 1  if s > e                  // recurse onto empty subarray
 2      return NIL
 3  elseif s == e             // recurse onto subarray with one node
 4      return A[s]
 5  else                      // s < e
 6      m = (s + e)//2        // the operator // is for integer division
 7      root = A[m]
 8      root.left = CONSTRUCT-TREE(A, s, m − 1)
 9      root.right = CONSTRUCT-tree(A, m + 1, e)
10      return root
```

Specification: Suppose array A is non-decreasingly sorted by keys of nodes in A, and s, e are natural numbers. Then, the algorithm will construct the binary search tree (and returns its root) containing nodes from $A[s : e]$ such that for each node in the tree, the number of nodes in its left and right subtree differ by at most one, and returns NIL if $s > e$.

Notice that if the specification holds, then the call CONSTRUCT-TREE$(A, 1, n)$, where A is constructed in step 2 and n is the number of nodes in the binary tree, will gives us the root of the desired tree.

I will briefly justify the specification holds.

- In the first base case $s > e$, NIL is returned as wanted. And in the other, $s = e$, the root of subtree of size 1 is returned - satisfying specification trivially. For the recursion step, assuming two recursion calls follow the specification, to show that all nodes has its left and right subtree size differ by at most 1, it suffices to show that *root* does. This follows directly from how we partitions array into subarrays $A[s : m-1]$ and $A[e : m+1]$ using $m = (s+e)//2$, combing with truth of specification for recursion calls (left and right subtrees with size $|A[s : m-1]|$ and $|A[m+1 : e]|$). Further, it is obvious that the resulting tree is a BST, as the array is sorted, and by specification the subtrees constructed in recursion call are BST. Therefore, the specification holds.

Runing Time Analysis: the first step takes constant time. For step 2, we traverse all nodes in T, and each perform constant operation (connect it to a pointer from array), so its runtime is in $O(n)$. For step 3, since our algorithm recurse on every nodes in the tree T/array A, and perform constant operation on each of them, the runtime is in $O(n)$. So, all together, the runtime of the entire algorithm is $O(n)$, as needed.

Q3

During the insertion, we keep track of the insertion path downwards to inserted node x. For example, we push nodes in the path into a stack, in order. So, after insertion performed, the first POP operation called to this stack gives us the parent of x, and the second gives us the grandparent of x, etc.

The algorithm is basically a while loop. Starting from x, at every iteration, going upward once to follow path of its proper ancestor, using a pointer u being updated each iteration. Notice that this can be done by calling POP to the stack S. That is, $u = x$ at the first iteration, $u = x$'parent at the second iteration, etc.

In the initial iteration, we count the number of nodes in the subtree rooted at $u = x$ using tree walk. And then find the sibling of $u = x$, call y, and count the number of nodes in the subtree rooted at y using a tree walk (by

x's parent that we stored in the stack S) this costs $O(n(x))$ and $O(n(y))$ respectively. After that, we check if one of $n(x)$ and $n(y)$ is greater than $2(n(x)+n(y)+1)/3$, if that is the case we found a, and loop terminates. Otherwise, we store $n(x)+n(y)+1$ into a variable, and proceed into next iteration by updating our pointer u to x's parent. So, after the first iteration, counting number of nodes in a subtree using tree walk is only needed for our node pointer u's sibiling, as we have variable that always keeps track of number of nodes in the subtree rooted at u. The checking of number of nodes is the same at every iteration onwards.

Runing Time Analysis: The loop terminates until we find a, and as our algorithm will traverse all nodes in the subtree rooted at a when terminates, and spend constant time on each node, so runtime for entire algorithm will be $O(n(a))$, as wanted.

Q4

Proof. We prove the statement using simple induction on n. We assume that $n \geq 1$, as when $n = 0$, $\lfloor \log_{3/2} n \rfloor = \lfloor \log_{3/2} 0 \rfloor$ is not defined.

Base Case: Let $n = 1$. After this sequence of 1 INSERT operation, the height of the tree is $0 \leq \lfloor \log_{3/2} 1 \rfloor$.

Induction Step: Let $n \in \mathbb{N}$. Assume that after a sequence of n INSERT operations are performed, beginning with an empty tree, the height of the tree is at most $\lfloor \log_{3/2} n \rfloor$. We will show that after a sequence of $n+1$ INSERT operations are performed, beginning with an empty tree the height of the tree is at most $\lfloor \log_{3/2} n+1 \rfloor$. After performed n operations we know that the height of the tree is at most $\lfloor \log_{3/2} n \rfloor$. Consider what happens after $(n+1)$-th INSERT operation, say the inserted node is x, there are two cases on depth of x.

- Case 1: x has depth less than or equal to $\lfloor \log_{3/2} size \rfloor$, i.e., $d(x) \leq \lfloor \log_{3/2}(n+1) \rfloor$.

  Then, there is no any rebalance will be performed, the INSERT operation is done fully. We know $d(x) \leq \lfloor \log_{3/2}(n+1) \rfloor$, and since the depth of other leaves do not change in this INSERT operation, by induction hypothesis, still less than or equal to $\lfloor \log_{3/2} n \rfloor < \lfloor \log_{3/2} n+1 \rfloor$. So, the height of the tree after this INSERT operation is at most $\lfloor \log_{3/2}(n+1) \rfloor$, as wanted.

- Case 2: x has depth greater than $\lfloor \log_{3/2} size \rfloor$, i.e., $d(x) > \lfloor \log_{3/2}(n+1) \rfloor$.

  Then, a rebalance is needed on the node a who is ancestor of x such that the BST is unbalanced on it. We will show that after the rebalance the height of the tree will be at most $\lfloor \log_{3/2} n \rfloor$ (thus at most $\lfloor \log_{3/2}(n+1) \rfloor$ as wanted).

  Since we know that before the INSERT the height is at most $\lfloor \log_{3/2}(n) \rfloor$, so is the depth of parent of x. So $d(x) \leq \lfloor \log_{3/2}(n) \rfloor + 1$. Combined with $d(x) > \lfloor \log_{3/2}(n+1) \rfloor > \lfloor \log_{3/2}(n) \rfloor$, we conclude that $d(x) = \lfloor \log_{3/2}(n) \rfloor + 1$. So, if we show that the subtree rooted at a decrease height by at least 1, combining with other leaves (those not in subtree rooted at a) has depth at most $\lfloor \log_{3/2} n \rfloor$ still, we conclude the result.

  In other words, denoting the height of subtree rooted at a before rebalance and after rebalance as h and h' respectively, we remains to show that $h > h'$.

  Suppose for a contradiction that $h \leq h'$. We first show two claims.

  Claim 1: Before rebalance, for any $h \in \mathbb{N}$ such that $h \geq 1$, if the height of the subtree rooted at a is h, then $n(a) \leq \frac{3}{2}(2^{h-1})$.

  Proof: Let $h \in \mathbb{N}$, $h \geq 1$. Notice that as stated above, $d(x) = \lfloor \log_{3/2}(n) \rfloor + 1$, so at depth of x, there is one node only (other leaves' height is at most $\lfloor \log_{3/2}(n) \rfloor$). Consider the left/right subtree of root a that contains x, call it U. Then, the size of subtree $U \setminus \{x\}$ is maximized when it is full, and we know its height is at most $h-2$ (no such tree when $h = 1$ or 2), so it has at most $2^{(h-2)+1} - 1 = 2^{h-1} - 1$ nodes. Hence, $|U| = |U \setminus \{s\}| + |\{x\}| \leq 2^{h-1}$. Since it is full and has higehr height than the other side subtree rooted from a, we know that it is the larger side subtree of the unbalance subtree rooted at a. So, the size of the other side subtree should plus root a, is at most half of the larger subtree, $\frac{1}{2}(2^{h-1})$. So, the entire subtree rooted at a, before REBALANCE, has at most $(2^{h-1}) + \frac{1}{2}(2^{h-1}) = \frac{3}{2}(2^{h-1})$ nodes, as wanted. ■

  Claim 2: After rebalance, for any $h' \in \mathbb{N}$ such that $h' \geq 1$, if the height of the subtree rooted at a is h', then $2^{h'} \leq n(a)$.

Proof: Initially, $h' = 1$. Then to satisfies property of "differ by at most 1", subtree rooted at a has at least $2 = 2^1$ nodes, a root and a leave.

Consider an arbitrary natural number $h' \geq 1$. Assume that if subtree rooted at a has height h, then $2^{h'} \leq n(a)$. Then, suppose that height of the subtree rooted at a is $h' + 1$. Then, one of a's subtree has height h' so by induction hypothesis, has at least $2^{h'}$ nodes. The other subtree, should have $2^{h'} - 1$ nodes to satisfies the "differ by at most 1" on the root a. So, altogether, with height $h' + 1$, the tree has at least $2^{h'} + (2^{h'} - 1) + 1 = 2^{h'+1}$ nodes, as wanted. ■

Since we know that height of subtree rooted at a is at least 1 (x with a) both before and after rebanlance, so $1 \leq h, h'$. Hence,

$$\begin{aligned} n(a) &\leq \frac{3}{2}(2^{h-1}) && \text{(by Claim 1)} \\ &\leq \frac{3}{2}(2^{h'-1}) && \text{(by assumption that } h \leq h', \text{ and } 1 \leq h, h') \\ &< 2(2^{h'-1}) \\ &= 2^{h'} \leq n(a) && \text{(by Claim 2)} \end{aligned}$$

This is a contradiction ($n(a) < n(a)$), so our inital assumption is false, i.e., we conclude $h > h'$, as wanted.

■

Q5

For this question, we note the algorithm in Q3 (finding node a) and the algorithm in Q2 (replacing a tree at a) together, and treat it as one algorithm REBALANCE.

1. Credit usage: We define 1 credit can pay for two different thing.

- 1 credit can be used to pay for all (which is a constant number) of constant operations during INSERT on any single node in the insertion/search path (i.e., basically 1 credit pay for a visit plus push into stack for future usage).
- 1 credit can also be used to pay for all (which is a constant number) constant operations during REBALANCE on any single node in subtree rooted at a (i.e., 1 credit can pay for a visit (for counting in Q3) plus put a node into array plus reconnect it to other nodes in the newly constructed tree).

2. Allocated Charge: Each INSERT operation has $4\lfloor \log_{3/2} n \rfloor$ credits. During insertion traversing downwards, we assign 4 credits to each nodes on the path, so every proper ancestor of inserted x gets 4 credits. For 4 credits on a node in the path, 1 will be used for the actual cost of INSERT, and the other 3 will be stored in them for maintaining the credit invariant. Notice that the total credits $4\lfloor \log_{3/2} n \rfloor$ is enough for assignment stated above since the height of tree is at most $\lfloor \log_{3/2} n \rfloor$ in prior and after every insertion (implied by Q4) so does the length of the insertion path (without x), and the unused credits, if any, can be donated to charity :).

3. Proof of Credit Invariant:

I will prove a stronger statement using induction, which directly implies the credit invariant given in the question.

Stronger Credit Invariant: Each node x in the tree contains at least $\max\{0, 3(|n(x.left) - n(x.right)| - 1)\}$ credits.

Proof: In prior to the first INSERT, the tree is empty. So, the invariant vacuously hold.

Consider an arbitrary INSERT operation in the sequence. Assume that the credit invariant hold before the INSERT operation, we want to show that it holds after the INSERT operation. We show that the credit invariant holds right after INSERT, and then if there is a REBALANCE it holds after REBANLANCE as well.

Right after INSERT:

- Notice that right after INSERT, every nodes u not in the insertion path (those other than inserted x and its proper ancestor) has $n(u.left)$ and $n(u.right)$ not change from before INSERT, so by the induction hypothesis, they still have at least $\max\{0, 3(|n(u.left) - n(u.right)| - 1)\}$ credits.
- For inserted x, it has no child, and it has 0 credits. So, the invariant holds on it since $0 \geq \max\{0, 3(|n(x.left) - n(x.right)| - 1)\} = \max\{0, -3\}$.
- For every nodes u in the path (x's proper ancestor), 4 credits are assigned to them during INSERT. 1 of 4 is used to pay for actual cost of constant operation that INSERT does to it, and 3 are stored in them. However, we either have $n(u.left)$ increase by 1 or $n(u.right)$ increase by 1. Since they are originally in the tree, and thus by induction hypothesis the invariant originally holds. And since the difference between left and right subtree sizes increase by 1, we need 3 more new credits stored into them to maintain the invariant, and there are actually 3 additional credits stored to them after INSERT as stated, so the invariant holds for x's any proper ancestor u.

So, right after INSERT, the credit invariant holds on every node.

Cases on whether perform REBALANCE:

- There is no REBALANCE needs to perform ($d(x) \leq \lfloor \log_{3/2} size \rfloor$), then the entire insertion is done. The credit invariant holds.
- There is REBALANCE needs to be perform on a proper ancestor a of x. Since right after insertion in last step, the credit invariant holds for all nodes, including a. We know that a contains at least $\max\{0, 3(|n(a.left) - n(a.right)| - 1)\}$ credits, thus contains at least $3(|n(a.left) - n(a.right)| - 1)$ credits. As the tree is unbalanced at a, we know that $n(a.left) > \frac{2}{3}n(a)$ (the analysis is the same if we switch $left$ and $right$ in here and following). Hence we know $n(a.right) = n(a) - n(a.left) - 1 < n(a) - \frac{2}{3}n(a) - 1 = \frac{1}{3}n(a) - 1$. So, $|n(a.left) - n(a.right)| > \frac{2}{3}n(a) - (\frac{1}{3}n(a) - 1) = \frac{1}{3}n(a) + 1$, so $3(|n(a.left) - n(a.right)| - 1) > 3(\frac{1}{3}n(a) + 1 - 1) = n(a)$. That is, there is at least $n(a) + 1$ credits stored in a. We use $n(a)$ of them for the REBALANCE operation. And afterwards, since nodes other than those in the subtree rooted at a has no change to their size of left and rigth subtrees, the credit invarint still holds on them. For those nodes u in the subtree rooted at a, since now they have left and right subtree size differ by at most 1 (0 or 1), so for any nodes $3(|n(u.left) - n(u.right)| - 1) \in \{0, -3\}$, since in prior to REBALANCE they originally has at least $\max\{0, 3(|n(u.left) - n(u.right)| - 1)\}$ credits, they have at least 0 or 3 credits for sure, so the credit invariant holds on any such u. ■

4. Conclusion

Our stronger credit invariant actually shows that we always have enough to pay for all kind of sequence of n INSERT operation. So, we conclude that the amortized cost of INSERT is at most the allocated charge of INSERT, which is $4 \log n \in O(\log n)$, as wanted.