1. (a) Each leaf of \( T_A \) is characterized by some reordering of the elements in \( A \). That is to say, if \( A = \{a_1, a_2, \ldots, a_n\} \), then implicit in the leaf of \( T_A \) is some permutation of that so that the elements in \( A \) are sorted.

It is quite possible for there to be multiple leaves of \( T_A \) that impose the same ordering of \( A \). I want to show that only one of those leaves could possibly be visited. The proof is by contradiction: suppose there do exist two leaves \( L \) and \( M \) that impose the same ordering of \( A \) and have nonzero probability of being visited. Since the probabilities of those leaves being visited are nonzero, there are two arrays \( A_1 \) and \( A_2 \) so that \( A_1 \) reaches leaf \( L \) and \( A_2 \) reaches leaf \( M \). Since different leaves are reached some specific comparison must have evaluated differently. Let’s call that comparison \( a_i > a_j \). Let \( A_1 \) be the array for which that evaluated to true, and let \( A_2 \) be the array for which that evaluated to false. Because of that comparison, the sorted version of \( A_1 \) must have the \( i \)th element after the \( j \)th element, while the sorted version of \( A_2 \) must have the \( i \)th element before the \( j \)th element. This contradicts our earlier assertion that \( L \) and \( M \) imposed the same ordering. Therefore, though multiple leaves may impose a reordering, at most 1 can ever be visited.

Now we must show that at least 1 can be visited, that is, there is no reordering that cannot be reached. This is slightly simpler. Suppose we have a reordering of \( A = \{a_1, a_2, \ldots, a_n\} \) as \( \hat{A} = \{a_{i_1}, a_{i_2}, \ldots, a_{i_n}\} \), where all \( i_j \) are distinct and are integers in the range \([1, n] \). Then, we can construct an \( A \) where the \( i_j \) element of \( A \) is \( a_{i_j} \). Therefore, some leaf with the reordering \( \hat{A} \) must be accomplished, so it is possible to reach every reordering.

All leaves of \( A \) that imply the same reordering must have exactly one leaf among them that is the “right” leaf to be reached. Since there are \( n! \) re-orderings, and all permutations of \( A \) are equally likely, each permutation must have \( 1/n! \) probability of showing up. Since there is exactly one visitable leaf for each permutation, there must be \( n! \) visitable leaves, each with the same probability \( 1/n! \) of being visited. The rest of the leaves are unvisitible as shown, and so have probability 0 of being visited.

(b) \( T \) is the immediate parent of \( LT \) and \( RT \). Therefore, depth from \( T \) will be exactly one more for each node (including leaves) than from \( LT \) and \( RT \). \( T \) has \( k \) leaves, all of which must be represented either in \( LT \) or \( RT \), so \( D(T) = D(LT) + D(RT) + \sum_{i=1}^{k} 1 = D(LT) + D(RT) + k \).

(c) Suppose we have a decision tree \( T \) with \( k \) leaves that achieves this minimum, so that \( D(T) = d(k) \). From our work in (b) we know that \( D(T) = D(LT) + D(RT) + k \). Suppose \( i_0 \) is the number of leaves in \( LT \) and \( k - i_0 \) the number of leaves in \( RT \). Since \( D(T) \) is the smallest it can possibly be, both the terms \( D(LT) \) and \( D(RT) \) should be as small as they can possibly be, specifically, \( d(k) = D(T) = D(LT) + D(RT) + k = d(i_0) + d(k - i_0) + k \). Naturally \( i_0 \) must be chosen so that both the left and right subtrees have at least one leaf, so whatever our choice of \( i_0 \) is, it must range from between 1 and \( k - 1 \). Since we’re trying to minimize \( d(k) \), that will be accomplished
by minimizing $d(k) = d(i) + d(k - i) + k$ with respect to $i$, or \( \min_{1 \leq i \leq k-1} \{d(i) + d(k - i) + k\} \). This will select $i$ as the desired minimizing $i_0$.

(d) We have the function \( f(i) = i \lg i + (k - i) \lg (k - i) \). We want roots of the derivative.

\[
\frac{df(i)}{di} = \lg i + \frac{i}{i} - \lg (k - i) - \frac{k - i}{k - i} = \lg i - \lg(k - i) = 0
\]

\[
\frac{d^2f(i)}{di^2} = \frac{1}{i} + \frac{1}{k - i}
\]

With $\lg i = \lg(k - i)$, as $\lg$ is surjective $i = k - i$, $2i = k$, or $i = \frac{k}{2}$ is our single root. Further, evaluating $f''(i) = \frac{d^2f(i)}{di^2}$ at $i = \frac{k}{2}$ we see $f''\left(\frac{k}{2}\right) = \frac{1}{2} + \frac{1}{k - \frac{k}{2}} = \frac{2}{k} + \frac{2}{k} = \frac{4}{k}$, which is $> 0$ so this point is indeed a minimum. $f(i)$ is minimized at $i = \frac{k}{2}$.

We can rewrite $d(k)$ as $d(k) = d(\frac{k}{2}) + d(k - \frac{k}{2}) + k = 2d(\frac{k}{2}) + k$. We have seen this recursion equation many times: it is of order $\Theta(k \lg k)$. $k$ may not be evenly divisible, but we are still left with a lower bound of $d(k) = \Omega(k \lg k)$.

(e) We want a lower bound on $D(T_A)$. As shown earlier, $T_A$ may have many leaves, but there must be at least $n!$ to represent all possible rearrangements of a list of length $n$. Moreover, we know the external path length has a minimum bound of $d(n!) = \Omega(n! \lg(n!))$, so $D(T_A) = \Omega(n! \lg(n!))$. That is the lower bound on operations necessary to go from the root of $T_A$ to every leaf. Each leaf is reachable with probability $1/n!$, so the expected time to sort a given array $A_{\text{input}}$ is $(1/n!) \cdot \Omega(n! \lg(n!)) = \Omega(\lg(n!))$. We know $\lg(n!) = \Theta(n \lg n)$, so the expected time is $\Omega(\lg(n!)) = \Omega(n \lg n)$.

(f) To prove this, I construct the deterministic sort $A$ using the randomized sort $B$. Let us find some randomization node $\text{RANDOM}(1,r)$ in the tree of $B$ so there are no other randomization nodes in the subtree at that randomization node. Then, for each of the $r$ children of that node, in each of those subtrees there must be one that has the fewest average number of comparisons for this lists that could make it to that point. We then replace the randomization subtree with the subtree of the lowest average comparison choice. We repeat this substitution process until there are no more randomization nodes in the comparison tree of $B$, at which point we have a deterministic comparison tree for $A$, which by its construction must make no more comparisons on average than $B$.

2. (a) The probability that any particular hash maps to a specific entry is $\frac{1}{n}$ (and similarly the probability that it doesn’t is $1 - \frac{1}{n}$). For $k$ of the $n$ hashes to map to an entry, $n-k$ must not, and there are $\binom{n}{k}$ ways to chose the $k$ entries that get mapped, so $Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}$.

(b) In order for $M = k$, there must be one slot that has exactly $k$ keys, and every other slot must have $k$ or fewer keys. Let $R_k$ be the probability that every other slot other than the “max” slot has $k$ or fewer keys: its exact value does not matter, save for the fact that, as a probability, it is $\leq 1$. The probability that this max slot has $k$ keys is, as we know, exactly $Q_k$, and there are $\binom{n}{1} = n$ ways to choose which of the $n$ slots is the “max” slot, so $P_k = nQ_k R_k \leq nQ_k \cdot 1 = nQ_k$. 

2
(c) We can use Stirling’s equation here. Also used is the fact that $1 + x \leq e^x$.

\[
Q_k = \left( \frac{1}{n} \right)^k \binom{n-1}{n-k} \frac{n^{n-k}}{(n-k)!}
\]

\[
= \frac{(n-1)^{n-k}}{n^n} \frac{n!}{(n-k)!k!}
\]

\[
\sim \frac{(n-1)^{n-k}}{n^n} \frac{\sqrt{2\pi n} \frac{n^n}{e^n}}{\sqrt{2\pi (n-k)} \frac{(n-k)^{n-k}}{e^{n-k}} \sqrt{2\pi k} \frac{k^k}{e^k}}
\]

\[
= \frac{(n-1)^{n-k}}{n^n} \frac{\sqrt{n}}{\sqrt{2\pi (n-k)k} \frac{n^{n-k}}{k^k}}
\]

\[
\leq \left( \frac{n}{n-k} \right)^{n-k} \cdot 1 \cdot \frac{1}{k^k}
\]

\[
= \left( 1 + \frac{k}{n-k} \right)^{n-k} \frac{1}{k^k}
\]

\[
\leq e^{\frac{k}{n-k}} \frac{n-k}{k} \frac{1}{k^k}
\]

\[
= e^k \frac{1}{k^k}
\]

So, $Q_k < \frac{e^k}{k^k}$.

(d) For $k_0 = c \lg n / \lg \lg n$:

\[
Q_{k_0} < \left( \frac{e}{c \lg n / \lg \lg n} \right)^{c \lg n / \lg \lg n}
\]

\[
= \frac{c^{\frac{c}{\lg n / \lg \lg n}}}{\frac{c \lg n / \lg \lg n}{\lg \lg n}}
\]

\[
= \frac{c^{\frac{c}{\lg n / \lg \lg n}} \frac{c \lg n / \lg \lg n}{\lg \lg n}}{n^{\frac{c \lg n / \lg \lg n}{\lg \lg n}}}
\]

\[
= \frac{c^{\frac{c}{\lg n / \lg \lg n}} \frac{c \lg n / \lg \lg n}{\lg \lg n}}{n^{\frac{c \lg n / \lg \lg n}{\lg \lg n}}}
\]

\[
= \frac{c^{\frac{c}{\lg n / \lg \lg n}} \frac{c \lg n / \lg \lg n}{\lg \lg n}}{n^{\frac{c \lg n / \lg \lg n}{\lg \lg n}}}
\]

\[
= \frac{c^{\frac{c}{\lg n / \lg \lg n}} \frac{c \lg n / \lg \lg n}{\lg \lg n}}{n^{\frac{c \lg n / \lg \lg n}{\lg \lg n}}}
\]

\[
= \frac{c^{\frac{c}{\lg n / \lg \lg n}} \frac{c \lg n / \lg \lg n}{\lg \lg n}}{n^{\frac{c \lg n / \lg \lg n}{\lg \lg n}}}
\]

\[
= \frac{c^{\frac{c}{\lg n / \lg \lg n}} \frac{c \lg n / \lg \lg n}{\lg \lg n}}{n^{\frac{c \lg n / \lg \lg n}{\lg \lg n}}}
\]

\[
= \frac{c^{\frac{c}{\lg n / \lg \lg n}} \frac{c \lg n / \lg \lg n}{\lg \lg n}}{n^{\frac{c \lg n / \lg \lg n}{\lg \lg n}}}
\]

\[
= \frac{c^{\frac{c}{\lg n / \lg \lg n}} \frac{c \lg n / \lg \lg n}{\lg \lg n}}{n^{\frac{c \lg n / \lg \lg n}{\lg \lg n}}}
\]

\[
= \frac{c^{\frac{c}{\lg n / \lg \lg n}} \frac{c \lg n / \lg \lg n}{\lg \lg n}}{n^{\frac{c \lg n / \lg \lg n}{\lg \lg n}}}
\]

\[
= \frac{c^{\frac{c}{\lg n / \lg \lg n}} \frac{c \lg n / \lg \lg n}{\lg \lg n}}{n^{\frac{c \lg n / \lg \lg n}{\lg \lg n}}}
\]

\[
= \frac{c^{\frac{c}{\lg n / \lg \lg n}} \frac{c \lg n / \lg \lg n}{\lg \lg n}}{n^{\frac{c \lg n / \lg \lg n}{\lg \lg n}}}
\]

\[
= \frac{c^{\frac{c}{\lg n / \lg \lg n}} \frac{c \lg n / \lg \lg n}{\lg \lg n}}{n^{\frac{c \lg n / \lg \lg n}{\lg \lg n}}}
\]
Past a certain $n$ certainly $\frac{\log e}{\log \log n} \leq \frac{1}{4}$, and also past $\log \log n \geq 16$ we know that $\frac{\log \log \log n}{\log \log n} \leq \frac{1}{4}$.

\[
\leq \frac{n^{c(\frac{1}{4} + \frac{1}{4})}}{cn^c} = \frac{1}{cn^2}
\]

Suppose $c = 6$. Then, $\frac{1}{cn^2} = \frac{1}{6n^2} < \frac{1}{n^4}$. From out work in part (b) we know $P_k \leq nQ_k < n\frac{1}{n^3} = \frac{1}{n^2}$ for $k \geq k_0$.

(e) Casually speaking of $k$ as a continuous variable, we know $E[M] = \int_0^n kP_k dk$. Moreover, $P_k$ is a probability distribution function for the random variable $M$, and we can treat it as such.

\[
E[M] = \int_0^n kP_k dk
= \int_0^{c\frac{\log n}{\log \log n}} kP_k dk + \int_{c\frac{\log n}{\log \log n}}^n kP_k dk
\leq \int_0^{c\frac{\log n}{\log \log n}} \frac{c\log n}{\log \log n} P_k dk + \int_{c\frac{\log n}{\log \log n}}^n nP_k dk
\leq \frac{c\log n}{\log \log n} \int_0^{c\frac{\log n}{\log \log n}} P_k dk + n \int_{c\frac{\log n}{\log \log n}}^n P_k dk
\leq \frac{c\log n}{\log \log n} \cdot 1 + n \frac{1}{n^2}
= \frac{c\log n}{\log \log n} + \frac{1}{n}
\]

So $E[M] = O(\log n/\log \log n)$.

3. Start with an empty binary radix tree. Insert each string into the radix tree. Since each insertion takes time length of the string and the strings total have length $n$, insertion takes $n$ time. Then, suppose we have a list structure that can be appended to in $O(1)$ time. We then do a preorder traversal through the tree, and for any node that has a key, we insert that key as a binary string. The traversal will take unit time per each edge in the tree, and from the insertion there will be at most $n$ edges in the tree, so traversal will takes $O(n)$ time, and insertion into the list will takes $O(n)$ time.

```
SORT(S)
1: T ← initialized radix tree
2: for each s ∈ S do
3: T.insert(s)
4: end for
5: L ← initialized list {this is assumed global}
6: TRAVERSE_APPEND(T.root) {needs to access L}
```
We have the TraverseAppend helper function below that takes a node in the radix tree. The list \( L \) is assumed to be global, so it can be modified in the helper function with changes reflecting themselves in the sort function.

**TraverseAppend**

1: \[ \text{if } N.\text{has0child} \text{ then} \]
2: \[ \text{TraverseAppend}(N.0\text{child}) \]
3: \[ \text{end if} \]
4: \[ \text{if } N.\text{hasKey} \text{ then} \]
5: \[ L.\text{append}(N.\text{key}) \]
6: \[ \text{end if} \]
7: \[ \text{if } N.\text{has1child} \text{ then} \]
8: \[ \text{TraverseAppend}(N.1\text{child}) \]
9: \[ \text{end if} \]

At the end of this nonsense, sort should return a list with the elements of the set in sorted order. The insertion takes \( O(n) \) time, the traversal takes \( O(n) \) time, and each append to the list we assume takes \( O(1) \) time which makes it \( O(n) \) time if each binary string has length 1, so all this totals to \( O(n) \) time.

4. (a) The nodes that need to be changed for insertion are the nodes that are traversed as the point to insert the new node is found, or in other words, the only new nodes in insertion are the node being inserted, and all of it’s would be ancestors.

Deletion is less trivial. To be sure all ancestors of the deleted node must be changed. If the deleted node had zero or one children, then nothing else must be changed: the changed parent of deleted node \( d \) will link to nothing or the one child of \( d \). However, if \( d \) has two children, then the successor of \( d \) must be removed. We know the successor will have at most one right child, so this will change only those nodes down from \( d \) to its successor. In short, if \( d \) has zero or one child those nodes on the path from the root down to \( d \) will be changed, and if \( d \) has two children those nodes on the path from the root down to the successor of \( d \) will be changed (since \( d \) has two children this path will include \( d \)).

(b) The modified insert procedure is a recursive insert.

**Persistent-Tree-Insert** \((T, k)\)

1: \[ x \leftarrow \text{root}[T] \]
2: \[ x' \leftarrow \text{a new uninitialized node} \]
3: \[ T' \leftarrow \text{a new empty tree} \]
4: \[ \text{root}[T'] \leftarrow x' \]
5: \[ \text{if } x = \text{NIL} \text{ then} \]
6: \[ \text{key}[x'] \leftarrow k \]

(c) Until it reaches some node that does not exist, the procedure takes unit time at each level. At each stage of recursion the position of the algorithm, so to speak, goes down one level. As said there are \( h \) levels. At every level visited a node is created. At worst, one additional level will be created if the traversal goes to the deepest nodes, so a maximum of \( h + 2 \) space is taken (\( h + 1 \) for the root plus the maximum \( h \) descendants, plus 1 for the additional node for key \( k \) created). Time takes some multiple of \( h + 1 \), as each level of recursion requires about 10 operations, except for the last which takes 7. In any event, insertion takes \( O(h) \) time and space.

(d) In every insertion, the root of the main tree is one of the changed nodes. This is inescapable. If a parent field is included in the node, then a changed root requires that its children be changed to indicate the new parent, which require that their children be changed to indicate their new parents, etc etc. In short, the root and every descendant of the root must be changed... i.e., every node must be changed, in effect creating a copy of the tree with the only structural difference being the new insertion! At the very least this changeover requires that we visit every node and go through the reinitialization of a changed node, which requires at best \( n \) visitations of a node, and \( n \) space for the new nodes. The best case time and space is \( \Omega(n) \).

(e) It is not necessary to rewrite the entire book on RB trees, but only to argue that the rotations can be done with a constant number of change nodes being created. We know that rotations are done in constant time. According to the persistent set definition of modifies, three nodes are modified; subtrees, though they may be shuffled among those three nodes, remain unchanged. In the insertion and deletion, we know that \( O(h) \) rotations are required. The total number of

7. \( \text{left}[x'] \leftarrow \text{NIL} \)
8. \( \text{right}[x'] \leftarrow \text{NIL} \)
9. \textbf{else}
10. \( x' \leftarrow \text{a new node} \)
11. \( \text{key}[x'] \leftarrow \text{key}[x] \)
12. \textbf{if} \( \text{key}[x] < k \) \textbf{then}
13. \( R \leftarrow \text{treeWithRoot}[\text{right}[x]] \)
14. \( R' \leftarrow \text{Persistent-Tree-Insert}(R, k) \)
15. \( \text{left}[x'] \leftarrow \text{left}[x] \)
16. \( \text{right}[x'] \leftarrow \text{root}[R'] \)
17. \textbf{else}
18. \( L \leftarrow \text{treeWithRoot}[\text{left}[x]] \)
19. \( L' \leftarrow \text{Persistent-Tree-Insert}(L, k) \)
20. \( \text{left}[x'] \leftarrow \text{root}[L'] \)
21. \( \text{right}[x'] \leftarrow \text{right}[x] \)
22. \textbf{end if}
23. \textbf{end if}
24. \textbf{return} \( T' \)
space created is \(3 \cdot O(h) = O(h)\), as the problem works its way up from the problem area up to the root (or wherever the RB properties are no long violated). Time is no different save for the constant extra time needed for each new initialized node, so \(O(1)O(h) = O(h)\) time. Operations in the non-persistent RB tree take \(O(h)\) time and \(O(1)\) space, so this new persistent variation takes \(O(h) + O(h) = O(h)\) time and \(O(1) + O(h) = O(h)\) space worst case. The entire point of the red black tree is that \(h = O(\lg n)\), so this is really \(O(\lg n)\) worst case space and time.

5. (a) We start a counter at 0. As we descend through a tree, each time we encounter a black node we increment the counter by one (surely an \(O(1)\) operation). The current node being visited has black height equal to the current value of the counter.

Maintaining \(bh[T]\) is simple. In insertion, if we never make our way up to the root in the process of our fixup, we know the black height hasn’t changed since we have only changed elements of one subtree of the root while the other and its black height has remained the same. If we do make it all the way up to the root and we must color the root black, then \(bh[T]\) must have increased by one.

Deletion is similar. As before, if the root is not reached then the black height could not have changed. If the root is reached, that is, if the node we want to delete makes its way to the root with its single child, then the node can be deleted, causing \(bh[T]\) to decrease by one. Otherwise, it remains the same.

(b) To find the largest node in \(T_1\) with black height \(bh[T_2]\): Set a counter equal to \(b \leftarrow 0\). Set \(y \leftarrow \text{root}[T_1]\). Then, repeat the following: If \(b = bh[T_1] - bh[T_2]\), then return \(y\), otherwise we continue with our loop. \(y \leftarrow \text{right}[y]\). If \(y\) is black set \(b \leftarrow b + 1\). At this point we repeat. In going right, we can never have \(b\) be nil since that would imply that we ran to a leaf of the tree where the black height was less than \(bh[T_2]\), a contradiction of the RB properties. \(y\) must be black since the first time the condition to break out of the loop is met is right after \(y\) is set to something black. This only visits nodes in the right-most branch of the tree which, like any other branch in the tree, has order \(O(\lg n)\) height, so the algorithm takes \(O(\lg n)\) time.

(c) Since all elements of \(T_1\) are less than \(x\), all elements of \(T_y\) are less than \(x\). We also know all elements of \(T_2\) are greater than \(x\). We can therefore have a tree built simply by taking a node with key \(x\) as the root, with subtrees set to \(T_y\) on the left and \(T_2\) on the right. All elements less than \(x\) are in the left subtree, and all elements greater than \(x\) are in the right subtree, so that condition of the binary search tree is maintained, and our invariant of \(T_y\) and \(T_2\) being proper binary search trees in their own right takes care of the rest. The time involved is just the time to create the node with key \(k\) and set its children, so this takes \(O(1)\) time.

(d) Once this new \(T_y\) is in place, we have the problem of the extra \(x\) node. If the parent of \(x\) is black, then we can color \(x\) red with no problem, and the black height of this new \(T_1\) remains the same. However, if the parent is red, then we have, in terms of red black trees, case 3 to deal with. It can be handled by the same RB-INSERT-FIXUP method starting at node \(x\), which will take at most time \(O(bh[T_1] - bh[T_2]) = O(\lg n)\) if the fixup procedure has to go all the way to

\[7\]
If $bh[T_1] \leq bh[T_2]$, we can make the following changes to the algorithm: If we simply change the
definition of right and left to mean left and right (respectively) and change the comparison
operators to mean their reverse as it pertains to comparing keys and values in the tree, we wind
up with the same algorithm. Since right and left have been reversed along with the comparison
operators, the binary search tree properties still hold, as do properties of the red black trees.

Putting it all together, we have $O(\lg n)$ time to find $y$, $O(1)$ time to create the new $T_y$ subtree,
and $O(\lg n)$ time to make sure it maintains the RB properties, all for a total of an $O(\lg n)$
algorithm.

6. (a) The **Find** function can be done as a regular binary tree search, with the stipulation that
the *addend* is subtracted from $x$ for every node we traverse.

**Find**($x, T$)
1: $n \leftarrow \text{root}[T]$
2: **while** $n \neq \text{nil}$ **do**
3: $x \leftarrow x - \text{addend}[n]$
4: **if** $x = \text{key}[n]$ **then**
5: \hspace{1em} **return** YES
6: **else if** $x < \text{key}[n]$ **then**
7: \hspace{1em} $n \leftarrow \text{left}[n]$
8: **else**
9: \hspace{1em} $n \leftarrow \text{right}[n]$
10: **end if**
11: **end while**
12: **return** NO

• The **Insert** function works quite similarly. As we traverse down, we decrement the key of
the node we’re inserting by every addend we encounter, until we find an appropriate nil leaf
to insert it at.

**Insert**($x, T$)
1: $n \leftarrow \text{root}[T]$
2: $v \leftarrow$ new initialized node
3: $\text{key}[v] \leftarrow x$
4: $\text{addend}[v] \leftarrow 0$
5: $\text{left}[v] \leftarrow \text{nil}$
6: $\text{right}[v] \leftarrow \text{nil}$
7: **while** $n \neq \text{nil}$ **do**
8: \hspace{1em} $\text{key}[v] \leftarrow \text{key}[v] - \text{addend}[n]$
9: **if** $x < \text{key}[n]$ **then**
10: \hspace{1em} **if** $\text{left}[n] = \text{nil}$ **then**

8
11: left[n] ← v
12: n ← nil
13: else
14: n ← left[n]
15: end if
16: else
17: if right[n] = nil then
18: right[n] ← v
19: n ← nil
20: else
21: n ← right[n]
22: end if
23: end if
24: end while

• By adding to the addend of a node, we add to all keys of the subtree rooted at that node. By subtracting from the root of a subtree of that node, that subtree has its keys remain unchanged. The general algorithm is that, starting at the root, it repeatedly goes right until it encounters a node with key greater than x, at which point it adds k to the addend, and then goes left repeatedly until it finds a node with key less than x, at which point it subtracts, then goes right again, goes left again, right, left, etc., until a nil node is encountered.

Push(x, k, T)
1: n ← root[T]
2: needMax ← true
3: while n ≠ nil do
4: x ← x − addend[n] \{adjust x\}
5: if needMax then
6: if x ≥ key[n] then
7: addend[x] ← addend[x] + k
8: n ← left[n]
9: needMax ← false
else
10: n ← right[n] \{keep searching right\}
end if
12: else
13: if x < key[n] then
14: addend[x] ← addend[x] − k
15: n ← right[n]
16: end if
17: end if
Each of these involve a loop that pushes the algorithm one level deeper in the tree so the loop repeats at most $h$ times, and the operations at each level take $O(1)$ time, for a grand total of $O(h) \cdot O(1) = O(h)$ time.

(b) The short answer is use a red-black tree to force $h = O(\lg n)$, though it is not initially clear that this can be done without error. The thing that makes rotations initially not seem to work is the existence of the addend. If $\text{addend} = 0$ on the two nodes we are rotating, then the rotations would work just dandily as they do now. It is very easy to force $\text{addend}$ to be 0 for the two nodes. We start with the higher node, (1) add its $\text{addend}$ to its key, (2) add the $\text{addend}$ to the $\text{addend}$ of its at most two children, and (3) set $\text{addend}$ to be zero; we have a node with $\text{addend} = 0$. This process can be repeated for the either the right or left child (for left and right rotate respectively). Then, both nodes have $\text{addend} = 0$ and can be rotated without fear of messing anything up. Each of these $\text{addend}$-resolutions takes at most 4 total additions, subtractions, and variable settings, so this does not affect the asymptotic running time of the algorithm at all. If we use these modified rotations in a modified red-black fixer methods and put them in our INSERT method, that will force the tree to always have $h = O(\lg n)$ so that all three of our methods then take $O(h) = O(\lg n)$ time.