1. Show that any \( n \)-node binary tree can be converted to any other \( n \)-node binary tree using \( O(n) \) rotations. Provide a statement and proof by construction.

Solution.

**Theorem 1.** Any \( n \)-node binary tree can be converted to any other \( n \)-node binary tree using \( O(n) \) rotation.

**Proof.** We break this into two parts. First, we need to show that any binary tree can be converted to an intermediate structure. If this is possible, then we know, that we can convert between any two trees, which we show in the second part.

We can convert any \( n \)-node binary tree into an \( n \)-node left-linear tree. Start at the left-most node. While the node has a right-child, rotate it, otherwise go to the nodes parent. Repeat until you finish rotating the new root. By rotating, we mean that convert a node that has a left and right child into the structure of the left being a child of the node and the node being a child of the right (a line). This will work on any structure of a binary tree.

Second, take two \( n \)-node binary trees, \( A \) and \( B \). Convert them both to \( n \)-node left-linear trees and record the rotations that occurred. Then to the \( n \)-node left-linear trees apply the reverse of the recorded rotations. This will convert the structure of one onto the other.
2. Write and analyze an algorithm to find and construct a list of all elements within a range of keys \((k_1, k_2)\) for a binary tree \(T\). The algorithm must run in \(O(s + h)\) time, where \(s\) is the number of elements in the range and \(h\) is the height of the tree.

**Algorithm.**

The algorithm is described in Algorithm 1. Essentially, begin by searching for the smallest node greater than or equal to \(k_1\), this is called \texttt{ceilingNode}. Then, from this node, walk through the tree in an inorder fashion constructing the list of keys between \(k_1\) and \(k_2\). Return the list of values.

**Algorithm 1 Elements in Range**

**Input:** Keys \(k_1\) and \(k_2\)

**Output:** List of elements whose keys are in the range provided

1. List \(l \leftarrow \emptyset\)
2. Node \(n \leftarrow \texttt{ceilingNode}(k_1)\)
3. while \(n\).getKey() \(\leq\) \(k_2\) do
4. \(l\).add(n.getValue())
5. \(n \leftarrow \texttt{inOrderNext}(n)\)
6. return \(l\)

**Time Complexity.**

**Theorem 2.** Algorithm 1 runs in \(O(s + h)\) time, where \(s\) is the number of elements in the range \([k_1, k_2]\) and \(h\) is the height of the tree.

**Proof.** The search for the ceiling node takes \(O(h)\) time with a standard tree search algorithm. Then we do an inorder walk through \(s\) tree nodes. The function \texttt{inOrderNext} will run in amortized constant time over the \(s\) nodes. Thus, in total the algorithm takes \(O(s + h)\) time.

**Memory Complexity.**

**Theorem 3.** Algorithm 1 uses \(O(s)\) additional memory.

**Proof.** The ceiling search function and inorder next functions can both be implemented iteratively and thus take \(O(1)\) extra memory. Constructing a list of \(s\) items is the only extra memory. Thus, the algorithm uses \(O(s)\) extra memory.
3. Let's assume your algorithm for the previous problem is modified to remove all elements within the range. State and prove the time complexity for this new algorithm for (1) a binary search tree and (2) and AVL tree.

**Theorem 4.** The algorithm runs in \( O(s + h) \) time for a binary search tree, where \( s \) is the number of elements in the range \([k_1, k_2]\) and \( h \) is the height of the tree.

*Proof.* The search for the ceiling node takes \( O(h) \) time with a standard tree search algorithm. Then we do an inorder walk through \( s \) tree nodes performing a modified removal function for the node. Let its modification be to return the next node after pulling it out. This clearly will run in amortized constant time over the \( s \) nodes. Thus, in total the algorithm takes \( O(s + h) \) time.

**Theorem 5.** Algorithm \( \square \) runs in \( O(s \log n) \) time for an AVL tree, where \( s \) is the number of elements in the range \([k_1, k_2]\).

*Proof.* The search for the ceiling node takes \( O\log n \) time with a standard tree search algorithm. Then we do an inorder walk through \( s \) tree nodes performing a modified removal function for the node. Let its modification be to return the next node after pulling it out. This function must rebalance the tree each time, requiring at most \( \log n \) restructurings. So, doing this for \( s \) nodes will take \( O(s \log n) \) time for an AVL tree.
4. **Bonus.** Show that the nodes of any AVL tree $T$ can be colored “red” and “black” so that $T$ becomes a red-black tree.

**Solution.**

**Theorem 6.** The nodes of any AVL tree $T$ can be colored “red” and “black” so that $T$ becomes a red-black tree.

*Proof.* Label the nodes in the following way. The root is black. Then for each pair of siblings, if the height of the left is less than the height of the right or the height of the left is even, color it black. If the height of the right is less than the height of the left or the height of the right is even, color it black. Otherwise color it red.

This labeling is correct and maintains the properties of a red-black tree. (1) The root is black. (2) The leaves are black because their heights are 0. (3) All children of red nodes are black because children of red nodes will have even height (black) or will be shorter than their sibling (black). (4) Based on an inductive argument. If the subtree has even height - when both subtrees have odd height they are red and their parent is black or the parent is black and the children are different colors. If the subtree has odd height - when both subtrees have even height, they are both black and the parent is red, or they are different and the parent is red and again both children are black. In all cases the black depth will be the same.