1. Explain how to use an AVL tree or a red-black tree to sort $n$ comparable elements in $O(n \log n)$ time in the worst case.

**Algorithm.**

Algorithm 1 shows the process. Simply, take each element and insert as an element, element entry into a sorted map. Then, return the key set, which is guaranteed to be in order as it is a sorted data structure.

**Algorithm 1** Map Sort

<table>
<thead>
<tr>
<th>Input: List $L$ of $n$ elements, Comparator $c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Collection of elements in sorted order</td>
</tr>
</tbody>
</table>

1: Sorted Map $M \leftarrow \emptyset$ (uses $c$ as the comparator)

2: for all $e \in L$ do

3: $M$.put($e, e$)

4: return $M$.keySet()

**Time Complexity.**

**Theorem 1.** Algorithm 1 runs in $O(n \log n)$ time, where $n$ is the number of elements.

*Proof.* Let the sorted map be implemented with a red-black tree. $n$ inserts into the map will take $O(n \log n)$, and then performing an inorder traversal to read the sorted elements will require $O(n)$ time. In total, this will take $O(n \log n)$ time.

**Memory Complexity.**

**Theorem 2.** Algorithm 1 uses $O(n)$ additional memory.

*Proof.* Items are copied from the original list to the sorted map, and then copied from the map into the return. The induced copy requires $O(n)$ additional memory.
2. Draw an example of a red-black tree that is not an AVL tree.

Solution.
Take the following in Figure 1. Clearly it satisfies all red-black tree properties, yet has a height imbalance at the root, according to the AVL tree definition.

Figure 1: Identical elements inserted in differing orders in an AVL search tree.
3. 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 2. 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 2 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 \(n.\text{getKey()} \leq k_2\) do
4. \(l.\text{add}(n.\text{getValue()}())\)
5. \(n \leftarrow \texttt{inOrderNext}(n)\)
6. return \(l\)

**Time Complexity.**

**Theorem 3.** Algorithm 2 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. \(\square\)

**Memory Complexity.**

**Theorem 4.** Algorithm 2 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. \(\square\)
4. 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) an AVL tree.

**Theorem 5.** 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 6.** Algorithm 2 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.

5. **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 7.** 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.