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.

<table>
<thead>
<tr>
<th>Algorithm 1 Map Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> List ( L ) of ( n ) elements, Comparator ( c )</td>
</tr>
<tr>
<td><strong>Output:</strong> Collection of elements in sorted order</td>
</tr>
<tr>
<td>1: Sorted Map ( M \leftarrow \emptyset ) (uses ( c ) as the comparator)</td>
</tr>
<tr>
<td>2: for all ( e \in L ) do</td>
</tr>
<tr>
<td>3: ( M\text{.put}(e, e) )</td>
</tr>
<tr>
<td>4: return ( M\text{.keySet()} )</td>
</tr>
</tbody>
</table>

**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. Suppose we are given two $n$-element sorted sequences $A$ and $B$ each with distinct elements, but potentially some elements that are in both sequences. Describe an $O(n)$-time method for computing a sequence representing the union $A \cup B$ (with no duplicates) as a sorted sequence.

**Algorithm.**

The algorithm is described in Algorithm 2. Essentially, the algorithm proceeds like the merge step of merge sort, except that when two elements are compared only unique element are added to the final sequence.

---

**Algorithm 2 Union**

**Input:** Sorted sequences $A$ and $B$

**Output:** Union $A \cup B$

1: List $U \leftarrow \emptyset$
2: while $\neg A$.isEmpty() $\land \neg B$.isEmpty() do
3:   $a \leftarrow A$.front(); $b \leftarrow B$.front(); $u \leftarrow U$.back()
4:   if $a = b$ then
5:     if $a \neq u$ then
6:       $U$.add($a$)
7:       $A$.removeFront(); $B$.removeFront()
8:   else if $a < b$ then
9:     if $a \neq u$ then
10:        $U$.add($a$);
11:        $A$.removeFront()
12:   else if $a > b$ then
13:     if $b \neq u$ then
14:        $U$.add($b$)
15:        $B$.removeFront()
16: while $\neg A$.isEmpty() do
17:   if $A$.front() $\neq U$.front() then
18:     $U$.add($A$.front());
19:     $A$.removeFront()
20: while $\neg B$.isEmpty() do
21:   if $B$.front() $\neq U$.front() then
22:     $U$.add($B$.front());
23:     $B$.removeFront()
24: return $U$

---
Time Complexity.

**Theorem 3.** Algorithm runs in $O(n)$ time, where $n$ is the size of each of the input sets.

*Proof.* This is almost identical to the merge function of merge sort. For similar reasons, it takes $O(n)$ time – each element is visited once ($2n$ elements) and each visit takes $O(1)$ time.

Memory Complexity.

**Theorem 4.** Algorithm uses $O(n)$ additional memory.

*Proof.* While the algorithm presented only moves elements around, it is reasonable to duplicate the input elements instead. Thus the operation would be bound by at most $O(n)$ elements appearing in the final union.

Note: $O(1)$ is an acceptable memory complexity.
3. **Bonus.** Describe and analyze an efficient method for removing all duplicates from a collection $A$ of $n$ elements.

**Algorithm.**

Algorithm 3 first sorts the list of elements. Then, while iterating over the collection any duplicates are removed. It is assumed that an efficient implementation of iteration and removal is given, e.g., if the collection is represented by a linked list, iteration and removal will occur efficiently.

### Algorithm 3 Remove duplicates

**Input:** Collection $A$ of $n$ elements

1: $\text{sort}(A)$
2: for $i \leftarrow 2$ to $|A|$ do
3: if $A.\text{get}(i - 1) = A.\text{get}(i)$ then
4: $A.\text{remove}(i)$

**Time Complexity.**

**Theorem 5.** Assuming, the collection and removal step is implemented efficiently, e.g., by a linked-list, Algorithm 3 runs in $O(n \log n)$ time, where $n$ is the number of elements.

**Proof.** The sort will occur in $O(n \log n)$ time. Since the collection is implemented efficiently, the iteration and removal step occurs in $O(n)$ time. In total the algorithm takes $O(n \log n)$ time. \hfill $\square$

**Memory Complexity.**

**Theorem 6.** Assuming an in place sort, Algorithm 3 uses $O(1)$ additional memory.

**Proof.** Since the sort is in place, it requires no additional memory. Removal requires no additional memory. Thus in total, $O(1)$ excess memory is used. \hfill $\square$