Exercise 13.1  
Segment trees.

a) We start with the empty skeleton of a segment tree with 16 leaves (each leaf represents an elementary interval of length 1, so 16 are needed to represent the interval [1, 17]). In every node, the represented interval is stored.

When a new segment is inserted, it is stored in the nodes that are closest to the root and represent intervals that are completely contained inside it. After the insertion of the segments, we obtain the following tree:
b) When performing a query for a value \( x \) in a node \( v \), we first check whether \( x \) is contained in the interval represented by \( v \). If so, then we output all the intervals contained in \( v \). Additionally, we forward the query to the children of \( v \) that represent intervals containing \( x \). Since we store closed intervals in the nodes, the query must be forwarded to both children if we ask for a boundary point of the interval. We must be careful not to output an interval twice. Each query starts from the root of the tree. For \( x = 11 \), the nodes that are marked in black in the following figure are visited.

The returned intervals are \( C \), \( E \) and \( F \).

c) Although it was not required, we give a formal proof for the sake of completeness. Let \( N \geq 4 \) be a power of 2. Consider a segment tree with interval boundaries from \( \{1, ..., N+1\} \). By induction over the depth \( T \), we show that in a segment tree of depth \( T \geq 2 \), every interval is stored in at most \( 2^T - 2 \) nodes.

**Base step** \((T = 2)\). To store intervals with boundaries from \( \{1, ..., N + 1\} \) for \( N \geq 4 \), a segment tree with a depth of at most 2 is required. It is easy to see that no interval is stored in more than \( 2^2 - 2 = 2 \) nodes.

**Inductive hypothesis.** Suppose that in every segment tree of depth \( T \), each interval is stored in at most \( 2^T - 2 \) nodes.

**Inductive step.** Consider an interval \( I \) and a segment tree of depth \( T + 1 \). We distinguish three cases.

- \( I \) is stored in the root. Then \( I \) is not stored in any other node of the segment tree.
- \( I \) is stored in one of the children of the root. Suppose that this was the left child \( v_1 \). Then the subtree rooted at \( v_3 \) has depth \( T \), and by the inductive hypothesis, \( I \) is stored in at most \( 2^T - 2 \) vertices. Overall, \( I \) is stored in at most \( 2^T - 2 + 1 \leq 2(T+1) - 2 \) nodes.
- \( I \) is neither stored in the root nor in the children of the root. At each level, a segment is inserted in at most two nodes: Suppose there is a level with three nodes \( v_1, v_2, v_3 \) (in this order) that represent an interval \( I \). If \( v_1, v_2 \) and \( v_3 \) represent contiguous intervals, the segment \( I \) would either be represented by the father of \( v_1 \) and \( v_2 \) (and not by \( v_3 \)), or by the father of \( v_2 \) and \( v_3 \) (and not by \( v_1 \)). If instead \( v_1, v_2 \) and \( v_3 \) represent intervals that are not contiguous, then \( I \) is not contiguous. Since there are \( T+1 \) levels, and the insertion of an interval in a node implies no insertions in the nodes below it, a segment can be inserted in a maximum of \( 2(T+1) - 2 = 2T - 2 \) nodes.

Since the depth of a segment tree is bounded by \( \log_2 N \), every interval is stored in at most \( 2 \log_2 N - 2 \) nodes.

The interval \([2, N]\) is stored in the nodes \([2^i + 1, 2^{i+1} + 1]\) as well as \([2^n - 2^{i+1} + 1, 2^n - 2^i + 1]\) for \( i = 0, 1, ..., \log_2 N - 2 \). These are exactly \( 2 \log_2 N - 2 \) nodes, i.e., exactly the upper bound. For \( N = 16 \) and the segment tree with the interval limits of \( \{1, ..., 17\} \), the segment \([2, 16]\) is stored at the nodes that are marked in black in the following figure.
Exercise 13.2  Piercing of orthogonal rectangles.

a) 1. **Stopping points.** We run the scanline from left to right, and we stop at beginnings of new rectangles and their ends. For this purpose, we sort the $x$-coordinates of the left and right sides of the rectangles in ascending order and use them as stopping points.

2. **Scanline data structure.** As currently active objects, we store the intervals that result in the intersection of the rectangles with the scanline. For this purpose, we use a segment tree. Additionally, we store the maximum number of overlapping intervals in each subtree in its root. This number can easily be computed recursively: If the two children of a node $v$ contain a maximum of $m_1$ and $m_2$ overlapping intervals, then the maximum number of overlapping segments in the interval represented by $v$ is exactly the sum of the number of intervals represented by $v$ and $\max\{m_1, m_2\}$ (all the intervals represented by $v$ overlap with the intervals represented by its children, so the maximum number of overlapping intervals must be increased accordingly). Additionally, at each node $v$ we store an elementary interval that is covered by a maximum number of intervals in the segment tree rooted at $v$.

It is important to note that the $y$-coordinates of the rectangles may be rational, but a segment tree with standard integer intervals $\{1, \ldots, 2n\}$ can be used anyway. We can sort the $y$-coordinates of the rectangles in advance and remove duplicates. We get $y_1, \ldots, y_k$ as possible coordinates, with $k \leq 2n$, and the corresponding elementary intervals are $[y_1, y_2], [y_2, y_3], \ldots, [y_{k-1}, y_k]$. If we store in each node of the tree its interval boundaries, then we can represent interval boundaries with values from $\{1, \ldots, k\}$.

3. **Update.** We distinguish two cases.

   **Case 1:** A new rectangle starts. In this case, we add the interval formed by the intersection of the scanline and the new rectangle to the segment tree. We also have to update the maximum number of overlapping intervals. To do this, we only have to compute new values for nodes that are visited during the insertion of the new interval (the values of the other nodes remain unchanged).

   **Case 2:** A rectangle ends. In this case, we delete the corresponding interval from the segment tree. The maximum number of overlapping rectangles has to be recalculated for the nodes that are visited during the deletion of the interval.

4. **Extracting the solution.** Every time the scanline data structure is updated, we check whether the overall number of maximum overlapping rectangles increases. This is the number stored in the root of the segment tree. For each position of the scanline, we keep the maximum number of overlapping rectangles, and the elementary interval that contains the most overlaps. At the end we can use this information to return a point that lies within the last remembered elementary interval inspected by the scanline.

b) The initial sorting of the stopping points can be done in time $O(n \log n)$. The same holds for the sorting of the $y$-coordinates. Insertion and deletion in the modified segment tree can be performed in time $O(\log n)$, since
we only have to update nodes that are visited during these operations. Since there are $2n$ stopping points, the resulting running time is $O(n \log n)$. Similarly, $O(n \log n)$ is an upper bound on the amount of memory required.

Exercise 13.3  Exposing line segments.

a) 1. Stopping points. As a scanline, we use a ray which moves in circular fashion around the light source. We stop it when we discover a new segment starting, or ending, or the intersection of two segments. At the beginning, the starting and ending points of all the segments are sorted according to the angle formed by the line that crosses the corresponding point and the light source, and the $x$-axis. These sorted angles are then used as the initial stopping points of the scanline. The intersection points are not initially known, but they can be discovered during the update step.

2. Scanline data structure. As active objects, we consider the segments that intersect with the scanline in ascending distance from the light source. The data structure must support insertion, removal and searching for successors and the predecessors. A suitable choice is an AVL-tree. It is important to observe that we are not interested in the absolute distances of the segments, but only on their relative order. We can proceed as follows. Each node of the AVL-tree stores a line segment. As keys we use the distance along the scanline from the light source to the corresponding segment. In every update of the scanline, the keys are recalculated on demand. The order of the elements can change only if two segments intersect.

We cannot generally start with an empty data structure as before. For the first stopping point, we have to identify all the segments that intersect with the scanline. These segments must then be inserted in ascending distance from the light source into the data structure, and for each pair of adjacent segments, we have to check whether they intersect somewhere, and insert the corresponding point as a stopping point of the scanline.

3. Update. We have to consider three cases.

Case 1: A new segment $s$ starts. Then $s$ must be inserted into the scanline data structure such that the order of the segments with respect to the current position of the scanline is preserved. The key of $s$ is the distance of the light source to the starting point of $s$. After inserting $s$, we have to check whether it is closest to the light source. In this case, $s$ is now exposed, and the previously exposed element is only exposed up to its current intersection with the scanline. We also have to identify the possible intersection points with adjacent segments. To do this, we check whether $s$ intersects with the immediately preceding element in the AVL tree. Similarly, we check whether $s$ intersects with the immediately following segment. This test can be performed easily using the intersection equation for the lines corresponding to each segment. If the intersection lies between the boundaries of both segments, then we have an intersection point. In this case, we insert a new stopping point for the scanline in the appropriate position.

Case 2: A segment $s$ ends. In this case, $s$ must be removed from the data structure. If $s$ is exposed, then the successor of $s$ is exposed starting from its current intersection with the scanline. We also have to check whether the segments that were previously the neighbors of $s$ (and are now directly next to each other) intersect in the future. If so, and if this intersection was not already found, the corresponding angle is inserted as a stopping point for the scanline.

Case 3: Two segments intersect. In this case, the corresponding elements in the data structure must be swapped. We also have to check whether one of the two segments was exposed: then it is exposed only up to the intersection point, and from there on the other segment is exposed. Finally, we have to check whether the segments intersect in the future with their new neighbors and that these intersections were not already found. In this case, we generate new stopping points for the scanline.

4. Extracting the solution. The exposed segment can always be found in the first position of the data structure. Once it is no longer exposed (because it ends, a new segment starts, or it is intersected by another segment), the previously exposed portion can be output.
b) The initial sorting of the start and end points can be performed in time $O(n \log n)$. The AVL-tree requires space $O(n)$, and allows searching, insertion and deletion in time $O(\log n)$. Checking whether two segments intersect and the calculation of the intersection point of a segment with the scanline can be done in constant time.

It is useful to manage the stopping points using a Minimum heap, because we have to determine the next stopping point and also have to add new stopping points dynamically if a new intersection point is detected. In total, there are $2n$ starting and ending points, and assume there are $k$ intersection points. Then, the additional memory needed is $O(n + k)$, and the intersection points can be inserted and deleted in time $O(\log(n + k))$. Note that $k$ is bounded from above by $O(n^2)$.

For each of the $O(n + k)$ stopping points, the required time is $O(\log(n + k))$. The overall running time of the algorithm hence is $O((n + k) \log(n + k))$. 