Chapter 14. Augmenting Data Structures

14.1-1.
$26 \to 17 \to 21 \to 19 \to 20$.

14.1-2.
$35 \to 38 \to 30 \to 41 \to 26$, $r = 1 \to 1 \to 3 \to 3 \to 16$.

14.1-3.

Algorithm 1: OS-SELECT(x, i)

```
r = x.left.size + 1;
while $r \neq i$ do
    if $i < r$ then
        x = x.left;
    else
        x = x.right;
        i -= r;
    r = x.left.size + 1;
return x;
```

14.1-4.

Algorithm 2: OS-KEY-RANK(T, k)

```
if $k == T.root.key$ then
    return T.root.left.size + 1;
else if $k < T.root.key$ then
    return OS-KEY-RANK(T.left, k);
else
    return T.root.left.size + 1 + OS-KEY-RANK(T.right, k);
```

14.1-5.
The following procedure retrieves the ith successor of x in the linear order of

T:

Algorithm 3: OS-SUCCESSOR(T, x, i)

```
r = OS-RANK(T, x);
s = r + i;
return OS-SELECT(T.root, s);
```

Since both OS-RANK and OS-SELECT take $O(\lg n)$ time, so does OS-SUCCESSOR.

14.1-6.
When inserting a node, for each node on the according path, add 1 to rank of the node only if the node is inserted within its left subtree. Similarly when deleting, subtract 1 to rank of the node only if the spliced-out node had been in its left subtree.
For rotations, consider a left rotation on node x, and denote the right child of x before the rotation as y. Then we add $x.rank$ to $y.rank$ after the rotation, because after rotation x becomes the left child of y. Right rotations are handled in a similar manner.

14.1-7.
The number of inversions in the given array can be obtained by summing up the numbr of larger elements that precede an element for each element in the array. Here, we have the number of inversion for j-th element is equal to $j - OS - RANK(T, j)$. so we can compute the number of inversions by inserting the elements of the array into an order statistic tree and using OS-RANK. Since insertion and OS-RANK each take $O(\lg n)$ time, so the total time is $O(n \lg n)$.

14.1-8.
First we sort the vertices in clockwise order and for each vertex assign a distinct value (since no two chords share an endpoint). Next, label each chord's endpoints as A_i and B_i where $A_i < B_i$. The number of chords which intersect a particula chord is equal to the number of chords which have only one endpoint between A_i and B_i. This is indeed a double-counting, we just count those intersections such that $A_i < A_j < B_i$. We can then use an order-statistic tree, inserting each vertex into the tree from the smallest. If it is a start endpoint, insert it into the tree. If it is an end endpoint, remove the corresponding start endpoint from the tree and count the number

of start endpoints we have seen before. Summing up gives the desired count. The running time for initial sorting is $\Theta(n \lg n)$, inserting each point takes $\Theta(\lg n)$, deleting each point takes $\Theta(\lg n)$, counting takes $\Theta(1)$, so the total running time is $\Theta(n \lg n)$.

14.2-1.
We can find the SUCCESSOR and PREDECESSOR of any node in $\Theta(1)$ time by storing corresponding pointers. Note that these values do not change during rotations. When inserting a node x, we can call the regular TREE-SUCCESSOR(x) and TREE-PREDECESSOR(x) functions (which takes $O(\lg n)$) to set its fields, and then update the predecessor of x.successor and the successor of x.predecessor (which takes $\Theta(1)$). When we delete a node, we update its successor and predecessor to point each other rather than x. MINIMUM and MAXIMUM are global properties of the tree and can be maintained directly: when we insert a node, update the MINIMUM or the MAXIMUM if necessary. When we delete a node, we may need to set MINIMUM to its successor or MAXIMUM to its predecessor if we are deleting the minimum or the maximum element in the tree. Thus, these operations run and the properties can be maintained in $\Theta(1)$.

14.2-2.
Yes, by Theorem 14-1, because the black-height of a node can be computed from the information at the node and its two children. It remains to show that within the RB-INSERT-FIXUP and RB-DELETE-FIXUP, the color changes cause only local black-height changes.
For RB-INSERT-FIXUP, there are three cases:
Case 1: z's uncle is red. In this case, the only node whose black height changed is $z.p.p$, which is increased by 1.
Case 2/3 : z's uncle is black. In this case, the black heights are unchanged.
For RB-DELETE-FIXUP, there are 4 cases:
Case 1: x's sibling is red. In this case, the black heights are unchanged.
Case 2: x's sibling is black, and the sibling's both children are black. In this case, the only node whose black height changed is $x.p$, which is decreased by 1.
Case 3 : x's sibling is black, and the sibling's left child is red and right child is black. In this case, the black heights are unchanged.
Case 4 : x's sibling is black, and the sibling's right child is red. In this case, we update the black heights of $x.p.bh = x.bh + 1$ and $x.p.p.bh = x.p.bh + 1$.

Therefore, we can conclude that the black heights of nodes can be maintained as fields in red-black trees without affecting the asymptotic performance of operations.
For the depths, this is not true, because the depth of a node depends on the depth is parent. When the depth of a node changes, the depths of all nodes in its subtree must be updated.

14.2-3.
After a rotation, starting at the deeper node x of the two nodes that were moved by the rotate, set $x.f = x.left.f \otimes x.a \otimes x.right.f$. Do the same thing for the higher node in the rotation. For size, set $x.size = x.left.size + x.right.size + 1$ and do the same thing for the higher node in the rotation.

14.2-4.
We should slightly modify TREE-SEARCH(x, a) to return the biggest element smaller than or equal to a rather than NIL if a is not in the tree. Adopting that, the following algorithm runs in $\Theta(m + \lg n)$ time:

Algorithm 4: RB-ENUMERATE(x, a, b)

```
begin = TREE-SEARCH(x, a);
if $begin < a$ then
    begin = start.successor;
while $begin \leq b$ do
    print begin;
    begin = begin.successor;
```

14.3-1.

Algorithm 5: LEFT-ROTATE(T, x)

```
y = x.right;
x.right = y.left;
if $y.left \neq T.nil$ then
    y.left.p = x;
y.p = x.p;
if $x.p == T.nil$ then
    T.root = y;
else if $x == x.p.left$ then
    x.p.left = y;
else
    x.p.right = y;
y.left = x;
x.p = y;
y.max = x.max;
x.max = MAX(x.high, x.left.max, x.right.max);
```

14.3-2.

Algorithm 6: INTERVAL-SEARCH(T, i)

```
x = T.root;
while $x \neq T.nil$ and i does not overlap x.int do
    if $x.left \neq T.nil$ and $x.left.max > i.low$ then
        x = x.left;
    else
        x = x.right;
return x;
```

14.3-3.

Algorithm 7: MIN-INTERVAL-SERACH(T, x, i)

```
if $x.left \neq T.nil$ and $x.left.max \geq i.low$ then
    y = MIN-INTERVAL-SEARCH(T, x.left, i);
    if $y \neq T.nil$ then
        return y;
    else if $i$ overlaps $x.int$ then
        return x;
    else
        return T.nil;
else if $i$ overlaps $x.int$ then
    return x;
else
    return MIN-INTERVAL-SEARCH(T, x.right, i);
```

14.3-4.

Algorithm 8: OVERLAPPING-INTERVALS(T, i)

```
if $T.root$ overlaps $i$ then
    print T.root;
if $T.root.left \neq T.nil$ and $T.root.left.max \geq i.low$ then
    OVERLAPPING-INTERVALS(T.root.left, i);
if $T.root.right \neq T.nil$ and $T.root.right.max \geq i.low$ and $T.int.low \leq$
  $i.high$ then
    OVERLAPPING-INTERVALS(T.root.right, i);
```

This algorithm examines at most twice and performs constant operation, so the runtime cannot exceed $O(n)$. If a recursive call is made on a branch of the tree, then that branch must contain an overlapping interval, so the runtime cannot also exceed $O(k \lg n)$. Therefore, the runtime is $O(\min(n, k \lg n))$.

14.3-5.

Algorithm 9: INTERVAL-SERACH-EXACTLY(T, i)

```
x = T.root;
while $x \neq T.nil$ and i not exactly overlap x do
    if $i.high > x.max$ then
        x = T.nil;
    else if $i.low < x.low$ then
        x = x.left;
    else if $i.low > x.low$ then
        x = x.right;
    else
        x = T.nil;
return x;
```

14.3-6.
Store the elements in the red-black tree, the number as the key itself. The additional attributes needed would be: the minimum gap in the subtree rooted at the node, the minimum and the maximum value in the subtree rooted at the node, These three fields can be computed from information in the node and the children, in the following manner:
$x.min = x.left.min$ if there is a left subtree, $x.min = x.key$ otherwise.
$x.max = x.right.max$ if there is a right subtree, $x.max = x.key$ otherwise.
$x.mingap = MIN(x.left.mingap, x.right.mingap, key - x.left.max, x.right.min - x.key)$
MIN-GAP simply return the mingap attribute in $O(1)$ time.

14.3-7.
Sort the rectangles by their x-coordinates, and scan the sorted list from the lowest to the highest x-coordinate. When an x-coordinate of a left edge is found, check whether the rectangle's y-coordinate interval overlaps an interval in the tree, and insert the rectangle into the tree. When an x-coordinate of a right edge is found, delete the rectangle from the interval tree. If an overlap is ever found in the interval tree, there are overlapping rectangles.
Time : $O(n \lg n)$ to sort rectangles, $O(n \lg n)$ for interval tree operations, making total time $O(n \lg n)$.

14.1.
a. If the maximum overlap point is in the interior of some segments, it is in the interior of the intersection of those segments. Either one of endpoints of

the intersection of those segments is indeed the maximum overlap points, a contradiction.
b. Set a red-black tree of the endpoints. For each left endpoint x, associate $x.v = +1$. For each right endpoint x, associate $x.v = -1$. When multiple endpoints have the same value, insert left ones first.
Assuming the intervals are in half-open form $[a, b)$, if we have $x_1, \cdots, x_n$ a sorted endpoints, summing the assigned values from $i = 1$ to j gives the number of intervals which overlap endpoint j. So, the point of maximum overlap is the value of j which has the maximum value of the prefix sum. Each node x stores three new attributes: the sum of the values of all nodes in the subtree rooted at the node, the maximum prefix sum in the subtree rooted at the node, the index which attains the maximum prefix sum in the subtree rooted at the node. Clearly, the index which attains the maximum prefix sum in the subtree rooted at the root will be the desired point.
By Theorem 14.1, it suffices to show that these attributes can be calculated using only information at each node and its children.
The sum of the values of all nodes in the subtree is simply $x.sum = x.left.sum + x.v + x.right$.
The maximum prefix sum can either be in the left subtree, x itself, or in the right subtree. Therefore it is $x.max = \max(x.left.max, x.v + x.left.sum, x.v + x.left.sum + x.right.max)$.
The index which attains the maximum prefix sum can be figuring out by examining where $x.max$ comes from.
Therefore, each operations INTERVAL-INSERT, INTERVAL-DELETE runs in $\Theta(\lg n)$ time, maintaining the mentioned properties. FIND-POM takes $O(1)$ time.

14.2.
a. Make a doubly linked circular list and manually construct the sequence. This takes $O(mn)$ time. Since m is constant, we have $O(n)$ time.
b. Use an order-statistic tree T and the following algorithm:

Algorithm 10: JOSEPHUS(n, m)

```
Initialize T to be empty;
for $j$ = 1 to $n$ do
    OS-INSERT(T, j);
j = 1;
for $k$ = $n$ downto 1 do
    j = ((j + m - 2)  mod  k) + 1;
    x = OS-SELECT(T.root, j);
    print x;
    OS-DELETE(T, x);
```

It takes $O(n \lg n)$ time to build up the order-statistic tree T, and we make $O(n)$ calls to the order-statistic tree procedures, which takes $O(\lg n)$ time. Thus, the total time is $O(n \lg n)$.