14 Augmenting Data Structures

14.1 Dynamic order statistics

14.1-1

> Show how $\text{OS-SELECT}(\text{T}.\text{root}, 10)$ operates on the red-black tree T of Figure 14.1.

- $26 : \text{r} = 13, \text{i} = 10$, go left.
- $17 : \text{r} = 8, \text{i} = 10$, go right.
- $21 : \text{r} = 3, \text{i} = 2$, go left.
- $19 : \text{r} = 1, \text{i} = 2$, go right.
- $20 : \text{r} = 1, \text{i} = 1$, choose 20.

14.1-2

> Show how $\text{OS-RANK}(\text{T}, \text{x})$ operates on the red-black tree T of Figure 14.1 and the node x with $\text{x}.\text{key} = 35$.

- $35 : \text{r} = 1$.
- $38 : \text{r} = 1$.
- $30 : \text{r} = \text{r} + 2 = 3$.
- $41 : \text{r} = 3$.
- $26 : \text{r} = \text{r} + 13 = 16$.

14.1-3

> Write a nonrecursive version of OS-SELECT.

```
OS-SELECT(x, i)
    r = x.left.size + 1
    while r ≠ i
        if i < r
            x = x.left
        else x = x.right
            i = i - r
        r = x.left.size + 1
    return x
```

14.1-4

> Write a recursive procedure $\text{OS-KEY-RANK}(\text{T}, \text{k})$ that takes as input an order-statistic tree T and a key k and returns the rank of k in the dynamic set represented by T. Assume that the keys of T are distinct.

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

14.1-5

> Given an element x in an n-node order-statistic tree and a natural number i, how can we determine the ith successor of x in the linear order of the tree in $\text{O}(\lg \text{n})$ time?

The desired result is $\text{OS-SELECT}(T, \text{OS-RANK}(T, x) + i)$. This has runtime $O(h)$, which by the properties of red black trees, is $O(\lg n)$.

14.1-6

> Observe that whenever we reference the size attribute of a node in either OS-SELECT or OS-RANK, we use it only to compute a rank. Accordingly, suppose we store in each node its rank in the subtree of which it is the root. Show how to maintain this information during insertion and deletion. (Remember that these two operations can cause rotations.)

First perform the usual BST insertion procedure on z, the node to be inserted. Then add 1 to the rank of every node on the path from the root to z such that z is in the left subtree of that node. Since the added node is a leaf, it will have no subtrees so its rank will always be 1.

When a left rotation is performed on x, its rank within its subtree will remain the same. The rank of $x.\text{right}$ will be increased by the rank of x, plus one. If we perform a right rotation on a node y, its rank will decrement by $y.\text{left.rank} + 1$. The rank of $y.\text{left}$ will remain unchanged.

For deletion of z, decrement the rank of every node on the path from z to the root such that z is in the left subtree of that node. For any rotations, use the same rules as before.

14.1-7

> Show how to use an order-statistic tree to count the number of inversions (see Problem 2-4) in an array of size n in time $O(n \lg n)$.

The runtime to build a red-black tree is $O(n \lg n)$, so we need to calculate inversions while building trees.

Every time INSERT, we can use OS-RANK to calculate the rank of the node, thus calculating inversions.

14.1-8 ⋆

> Consider n chords on a circle, each defined by its endpoints. Describe an $O(n \lg n)$-time algorithm to determine the number of pairs of chords that intersect inside the circle. (For example, if the n chords are all diameters that meet at the center, then the correct answer is $\binom{n}{2}$.) Assume that no two chords share an endpoint.

Sort the vertices in clock-wise order, and assign a unique value to each vertex. For each chord its two vertices are u_i, v_i and $u_i < v_i$. Add the vertices one by one in clock-wise order, if we meet a u_i, we add it to the order-statistic tree, if we meet a v_i, we calculate how many nodes are larger than u_i (which is the number of intersects with chord i), and remove u_i.

14.2 How to augment a data structure

14.2-1

> Show, by adding pointers to the nodes, how to support each of the dynamic-set queries MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $O(1)$worstcase time on an augmented order-statistic tree. The asymptotic performance of other operations on order-statistic trees should not be affected.

- **MINIMUM:** A pointer points to the minimum node, if the node is being deleted, move the pointer to its successor.
- **MAXIMUM:** Similar to MINIMUM.
- **SUCCESSOR:** Every node records its successor, the insertion and deletion is similar to that in linked list.
- **PREDECESSOR:** Similar to MAXIMUM.

14.2-2

> Can we maintain the black-heights of nodes in a red-black tree as attributes in the nodes of the tree without affecting the asymptotic performance of any of the redblack tree operations? Show how, or argue why not. How about maintaining the depths of nodes?

Since the black height of a node depends only on the black height and color of its children, Theorem 14.1 implies that we can maintain the attribute without affecting the asymptotic performance of the other red-black tree operations. The same is not true for maintaining the depths of nodes. If we delete the root of a tree we could potentially have to update the depths of $O(n)$ nodes, making the DELETE operation asymptotically slower than before.

14.2-3 ⋆

> Let $\otimes$ be an associative binary operator, and let a be an attribute maintained in each node of a red-black tree. Suppose that we want to include in each node x an additional attribute f such that $x.f = x_1.a \otimes x_2.a \otimes \cdots \otimes x_m.a$, where $x_1, x_2, \ldots, x_m$ is the inorder listing of nodes in the subtree rooted at x. Show how to update the f attributes in $O(1)$ time after a rotation. Modify your argument slightly to apply it to the size attributes in order-statistic trees.

$x.f = x.left.f \otimes x.a \otimes x.right.f$.

14.2-4 ⋆

> We wish to augment red-black trees with an operation $\text{RB-ENUMERATE}(x, a, b)$ that outputs all the keys k such that $a \le k \le b$ in a red-black tree rooted at x. Describe how to implement RB-ENUMERATE in $\Theta(m + \lg n)$ time, where m is the number of keys that are output and n is the number of internal nodes in the tree. (Hint: You do not need to add new attributes to the red-black tree.)

- $\Theta(\lg n)$: Find the smallest key that larger than or equal to a.
- $\Theta(m)$: Based on Exercise 14.2-1, find the m successor.

14.3 Interval trees

14.3-1

> Write pseudocode for LEFT-ROTATE that operates on nodes in an interval tree and updates the $\max$ attributes in $O(1)$ time.

Add 2 lines in LEFT-ROTATE in 13.2

```
    y.max = x.max
    x.max = max(x.high, x.left.max, x.right.max)
```

14.3-2

> Rewrite the code for INTERVAL-SEARCH so that it works properly when all intervals are open.

```
INTERVAL-SEARCH(T, i)
    x = T.root
    while x ≠ T.nil and i does not overlap x.int
        if x.left ≠ T.nil and x.left.max > i.low
            x = x.left
        else x = x.right
    return x
```

14.3-3

> Describe an efficient algorithm that, given an interval i, returns an interval overlapping i that has the minimum low endpoint, or $T.nil$ if no such interval exists.

Consider the usual interval search given, but, instead of breaking out of the loop as soon as we have an overlap, we just keep track of the overlap that has the minimum low endpoint, and continue the loop. After the loop terminates, we

return the overlap stored.

14.3-4

> Given an interval tree T and an interval i, describe how to list all intervals in T that overlap i in $O(\min(n, k \lg n))$ time, where k is the number of intervals in the output list. (Hint: One simple method makes several queries, modifying the tree between queries. A slightly more complicated method does not modify the tree.)

```
INTERVALS-SEARCH(T, x, i)
    let list be an empty array
    if i overlaps x.int
        list.APPEND(x)
    if x.left ≠ T.nil and x.left.max > i.low
        list = list.APPEND(INTERVALS-SEARCH(T, x.left, i))
    if x.right ≠ T.nil and x.int.low ≤ i.high and x.right.max ≥ i.low
        list = list.APPEND(INTERVALS-SEARCH(T, x.right, i))
    return list
```

14.3-5

> Suggest modifications to the interval-tree procedures to support the new operation INTERVAL-SEARCH-EXACTLY(T, i), where T is an interval tree and i is an interval. The operation should return a pointer to a node x in T such that $x.\text{int}.\text{low} = i.\text{low}$ and $x.\text{int}.\text{high} = i.\text{high}$, or $T.\text{nil}$ if T contains no such node. All operations, including INTERVAL-SEARCH-EXACTLY, should run in $O(\lg n)$ time on an n-node interval tree.

Search for nodes which has exactly the same low value.

```
INTERVAL-SEARCH-EXACTLY(T, i)
    x = T.root
    while x ≠ T.nil and i not exactly overlap x
        if i.high > x.max
            x = T.nil
        else if i.low < x.low
            x = x.left
        else if i.low > x.low
            x = x.right
        else x = T.nil
    return x
```

14.3-6

> Show how to maintain a dynamic set Q of numbers that supports the operation MIN-GAP, which gives the magnitude of the difference of the two closest numbers in Q. For example, if $Q = \{1, 5, 9, 15, 18, 22\}$, then MIN-GAP$(Q)$ returns $18 - 15 = 3$, since 15 and 18 are the two closest numbers in Q. Make the operations INSERT, DELETE, SEARCH, and MIN-GAP as efficient as possible, and analyze their running times.

Store the elements in a red-black tree, where the key value is the value of each number itself. The auxiliary attribute stored at a node x will be the min gap between elements in the subtree rooted at x, the maximum value contained in subtree rooted at x, and the minimum value contained in the subtree rooted at x. The min gap at a leaf will be ∞. Since we can determine the attributes of a node x using only the information about the key at x, and the attributes in $x.\text{left}$ and $x.\text{right}$, Theorem 14.1 implies that we can maintain the values in all nodes of the tree during insertion and deletion without asymptotically affecting their $O(\lg n)$ performance. For MIN-GAP, just check the min gap at the root, in constant time.

14.3-7 ⋆

> VLSI databases commonly represent an integrated circuit as a list of rectangles. Assume that each rectangle is rectilinearly oriented (sides parallel to the x- and y-axes), so that we represent a rectangle by its minimum and maximum x and y-coordinates. Give an $O(n \lg n)$-time algorithm to decide whether or not a set of n rectangles so represented contains two rectangles that overlap. Your algorithm need not report all intersecting pairs, but it must report that an overlap exists if one rectangle entirely covers another, even if the boundary lines do not intersect. (Hint: Move a "sweep" line across the set of rectangles.)

Let L be the set of left coordinates of rectangles. Let R be the set of right coordinates of rectangles. Sort both of these sets in $O(n \lg n)$ time. Then, we will have a pointer to L and a pointer to R. If the pointer to L is smaller, call interval search on T for the up-down interval corresponding to this left hand side. If it contains something that intersects the up-down bounds of this rectangle, there is an intersection, so stop.

Otherwise add this interval to T and increment the pointer to L. If R is the smaller one, remove the up-down interval that that right hand side corresponds to and increment the pointer to R. Since all the interval tree operations used run in time $O(\lg n)$ and we only call them at most $3n$ times, we have that the runtime is $O(n \lg n)$.

Problem 14-1 Point of maximum overlap

> Suppose that we wish to keep track of a ***point of maximum overlap*** in a set of intervals—a point with the largest number of intervals in the set that overlap it.
>
> **a.** Show that there will always be a point of maximum overlap that is an endpoint of one of the segments.
>
> **b.** Design a data structure that efficiently supports the operations INTERVAL-INSERT, INTERVAL-DELETE, and FIND-POM, which returns a point of maximum overlap. (Hint: Keep a red-black tree of all the endpoints. Associate a value of $+1$ with each left endpoint, and associate a value of -1 with each right endpoint. Augment each node of the tree with some extra information to maintain the point of maximum overlap.)

(Removed)

Problem 14-2 Josephus permutation

> We define the ***Josephus problem*** as follows. Suppose that n people form a circle and that we are given a positive integer $m \leq n$. Beginning with a designated first person, we proceed around the circle, removing every mth person. After each person is removed, counting continues around the circle that remains. This process continues until we have removed all n people. The order in which the people are removed from the circle defines the (n, m)***-Josephus permutation*** of the integers $1, 2, \ldots, n$. For example, the $(7, 3)$-Josephus permutation is $\langle 3, 6, 2, 7, 5, 1, 4 \rangle$.
>
> **a.** Suppose that m is a constant. Describe an $O(n)$-time algorithm that, given an integer n, outputs the (n, m)-Josephus permutation.
>
> **b.** Suppose that m is not a constant. Describe an $O(n \lg n)$-time algorithm that, given integers n and m, outputs the (n, m)-Josephus permutation.

(Removed)