Chapter 14 Solusion

frc6.com
i@frc6.com
https://github.com/frc123/CLRS-code-solution

10/20/2021

14.1

14.1-1

recursion	$x.key$	r	i
1	26	13	10
2	17	8	10
3	21	3	2
4	19	1	2
5	20	1	1

The result is 20.

14.1-2

iteration	$y.key$	r
1	35	1
2	38	1
3	30	3
4	41	3
5	26	16

The result is 16.

14.1-3

```
template <class Key, class T>
typename OrderStatisticsTree<Key, T>::Node* OrderStatisticsTree<Key, T>::Select
    (Node* subtree_root_node, size_t rank)
{
    size_t root_rank;
    root_rank = subtree_root_node->left->size + 1;
    while (rank != root_rank)
```

```
        {
            if (rank < root_rank)
            {
                subtree_root_node = subtree_root_node->left;
            }
            else
            {
                subtree_root_node = subtree_root_node->right;
                rank -= root_rank;
            }
            root_rank = subtree_root_node->left->size + 1;
        }
        return subtree_root_node;
    }
```

14.1-4

```
template <class Key, class T>
size_t OrderStatisticsTree<Key, T>::Rank(Node* node)
{
    if (node == root_)
        return node->left->size + 1;
    else if (node == node->parent->left)
        return Rank(node->parent) - node->right->size - 1;
    else
        return Rank(node->parent) + node->left->size + 1;
}
```

14.1-5

$r = \text{OS-R}\textsc{ank}(T, x)$
$succ = \text{OS-S}\textsc{elect}(T.root, r + i)$

The result is *succ*.

14.1-6

For RB-INSERT(T, z), set $z.rank = 1$, and change the while loop to the following code:

while $x \neq T.nil$
    $y = x$
    if $z.key < x.key$
        $x.rank = x.rank + 1$
        $x = x.left$
    else $x = x.right$

For RB-DELETE(T, z), add the following code right befire line 18 (in the else branch):

$y.rank = z.rank$

And invoke RB-DELETE-FIX-RANK(T, x) right before line 21.

RB-DELETE-FIX-RANK(T, x)

while $x \neq T.root$
    if $x == x.p.left$
        $x.p.rank = x.p.rank - 1$
    $x = x.p$

For LEFT-ROTATE(T, x), add the following code to the end of the procedure:

$y.rank = y.rank + x.rank$

For RIGHT-ROTATE(T, y), add the following code to the end of the procedure:

$y.rank = y.rank - x.rank$

14.1-7

```cpp
template <typename Key>
size_t CountInversions(std::vector<Key> array)
{
    size_t inversions, i, rank;
    OrderStatisticsTree<Key, int> tree;
    std::pair<typename OrderStatisticsTree<Key, int>::Iterator, bool> insert_result;
    inversions = 0;
    for (i = 0; i < array.size(); ++i)
    {
        insert_result = tree.Insert({array[i], 0});
        rank = tree.Rank(insert_result.first);
        inversions += (1 + i - rank);
    }
    return inversions;
}
```

14.1-8

```
size_t CountIntersections(std::vector<Chord> chords)
{
    size_t intersections, i, rank_a, rank_b, rank_diff_1, rank_diff_2;
    OSTree tree;
    std::pair<typename OSTree::Iterator, bool> insert_result_a, insert_result_b;
    intersections = 0;
    for (i = 0; i < chords.size(); ++i)
    {
        insert_result_a = tree.Insert({chords[i].endpoint_a, 0});
        insert_result_b = tree.Insert({chords[i].endpoint_b, 0});
        rank_a = tree.Rank(insert_result_a.first);
        rank_b = tree.Rank(insert_result_b.first);
        if (rank_a > rank_b) std::swap (rank_a, rank_b);
        // rank_a must smaller than rank_b
        rank_diff_1 = rank_b - rank_a - 1;
        rank_diff_2 = tree.Size() - rank_b + rank_a - 1;
        intersections += std::min(rank_diff_1, rank_diff_2);
    }
    return intersections;
}
```

14.2

14.2-1

Add *prev* and *succ* attributes to each node in the tree. Let *prev* points to predecessor of the node, and let *succ* points to successor of the node. Let $T.nil.succ$ points to the minumum element in the tree, and let $T.nil.prev$ points to the maximum element in the tree. A circular doubly linked list is formed.

In order to maintain these informations, we just need to modify RB-Insert and RB-Delete.

For RB-Insert(T, z), modify line 9 - 13 to the following code:

```
if y == T.nil
    T.root = z
    z.succ = T.nil
    z.prev = T.nil
elseif z.key < y.key
    y.left = z
    z.succ = y
    z.prev = y.prev
else y.right = z
    z.prev = y
    z.succ = y.succ
z.succ.prev = z
z.prev.succ = z
```

For RB-DELETE(T, z), add the following code right before line 21:

```
z.prev.succ = z.succ
z.succ.prev = z.prev
```

14.2-2

We can maintain black-heights of nodes without affecting the asymptotic performance since a change to $x.bh$ propagates only to ancestors of x in the tree.
For RB-INSERT(T, z), add the following code right before line 17:

```
z.bh = 1
```

For RB-INSERT-FIXUP(T, z), add the following code right before line 8 (in the if branch) (case 1):

```
z.p.p.bh = z.p.p.bh + 1
```

For RB-DELETE-FIXUP(T, z), add the following code right before line 11 (in the if branch) (case 2):

```
x.p.bh = x.p.bh - 1
```

And add the following code right before line 21 (in the else branch) (case 4):

```
x.p.bh = x.p.bh - 1
x.p.p.bh = x.p.p.bh + 1
```

We cannot maintain depths of nodes without affecting the asymptotic performance since a change to $x.bh$ propagates to descendants of x in the tree.

14.2-3

After the rotation on x is performed, run the following code:

```
x.p.f = x.f
x.f = x.left.f ⊗ x.right.f ⊗ x.a
```

Apply to the *size* attributes in order-statistic trees, we just need to change f to *size*, change $\otimes$ to $+$, and attibute a of each node will be 1; the code will be:

```
x.p.size = x.size
x.size = x.left.size + x.right.size + 1
```

14.2-4

The following procedure takes $\Theta(m + \lg n)$ time (to understand this asymptotic performance, refer to theorem 12.1 and exercise 12.2-8):

```
RB-ENUMERATE(x, a, b)
if a ≤ x.key and x.key ≤ b
    OUTPUT(x)
if a ≤ x.key and x.left ≠ T.nil
    RB-ENUMERATE(x.left, a, b)
if x.key ≤ b and x.right ≠ T.nil
    RB-ENUMERATE(x.right, a, b)
```

Note that we need to implement RB-ENUMERATE(x, a, b) in $\Theta(m + \lg n)$ time, so it does not meet the requirement of the question if we implement the procedure in the following ways (augment the tree in the way of exercise 14.2-1) since it takes $\Theta(m)$ time only:

```
RB-ENUMERATE(T, a, b)
k = a
OUTPUT(k)
repeat
    k = k.succ
    OUTPUT(k)
until k == b
```

14.3

14.3-1

Add the following lines to the end of LEFT-ROTATE(T, x):

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

14.3-2

```
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

Iterative version:

```
INTERVAL-SEARCH-MIN(T, i)
x = T.root
smallest = T.nil
while x ≠ T.nil
    if x.left ≠ T.nil and x.left.max ≥ i.low
        if i overlaps x.int
            smallest = x
        x = x.left
    else if i overlaps x.int
            return x
        x = x.right
return smallest
```

Recursive version:

Invoke INTERVAL-SEARCH-MIN($T, T.root, i$)

```
INTERVAL-SEARCH-MIN(T, x, i)
if x.left ≠ T.nil and x.left.max ≥ i.low
    smallest = INTERVAL-SEARCH-MIN(T, x.left, i)
    if smallest ≠ T.nil
        return smallest
    elseif i overlaps x.int
        return x
    else return T.nil
else if i overlaps x.int
        return x
    else return INTERVAL-SEARCH-MIN(T, x.right, i)
```

14.3-4

LIST-OVERLAP-INTERVAL(T, x, i)

if $x.int$ overlaps i
    OUTPUT(x)
if $x.left \neq T.nil$ and $x.left.max \geq i.low$
    LIST-OVERLAP-INTERVAL$(T, x.left, i)$
if $x.right \neq T.nil$ and $x.right.max \geq i.low$ and $x.int.low < i.high$
    LIST-OVERLAP-INTERVAL$(T, x.right, i)$

14.3-5

INTERVAL-SEARCH-EXACTLY(T, i)

$x = T.root$
while $x \neq T.nil$ and $(x.int.low \neq i.low$ or $x.int.high \neq i.high)$
    if $i.high > x.max$
        $x = T.nil$
    elseif $i.low < x.low$
        $x = x.left$
    elseif $i.low > x.low$
        $x = x.right$
    else $x = T.nil$
return x

14.3-6

Consider to augment the red-black tree by adding attibute *min-gap*, *min*, and *max* to every node.

min-gap is the minimum gap in the subtree rooted at the node.

min is the minimum key in the subtree rooted at the node.

max is the maximum key in the subtree rooted at the node.

When MIN-GAP(Q) is called, we just need to return $Q.root.min\text{-}gap$, which takes $O(1)$ time.

Let x be arbitrary node in the red-black tree Q. In order to maintain *min-gap* in $O(\lg n)$, we want $x.min\text{-}gap$ depends on only the information in nodes x, $x.left$, and $x.right$.

$x.min\text{-}gap = \min(x.left.min\text{-}gap, x.right.min\text{-}gap, x.key - x.left.max, x.right.min - x.key)$

$Q.nil.min\text{-}gap = \infty$

$Q.nil.min = \infty$

$Q.nil.max = -\infty$

Notice when there is no left subtree of x ($x.left == Q.nil$), $x.prev$ will be the ancestor of x, and the gap between $x.prev$ and x will be compared by $x.prev$, so the gap will not be neglected. (recall that $x.min\text{-}gap$ only contains the minimum gap in the subtree rooted at x)

We can maintain *min* and *max* in $O(\lg n)$ also since $x.min$ and $x.max$ depends only on only the information in nodes x, $x.left$, and $x.right$.

$$x.min = \min(x.left.min, x.key)$$
$$x.max = \max(x.right.max, x.key)$$

Notice that if $x.left \neq Q.nil$, $x.left.min < x.key$ must be true; if $x.right \neq Q.nil$, $x.right.max > x.key$ must be true.

14.3-7

```
bool DetermineOverlapRectangles(const std::vector<Rectangle>& rectangles)
{
    bool found_overlap;
    RectangleWithXCoordIndex *alloc_ptr, *now;
    std::allocator<RectangleWithXCoordIndex> alloc;
    std::vector<RectangleWithXCoordIndex*> heap;
    Tree tree;// interval tree
    found_overlap = false;
    // sort the x-coordinates with the minimum heap
    alloc_ptr = ConstructHeap(rectangles, heap);
    while (heap.size() > 0)
    {
        now = HeapExtractMin(heap);
        if (now->coord_pos == RectangleWithXCoordIndex::LEFT)
        {
            if (tree.Find(now->rectangle->y_int) != tree.End())
            {
                found_overlap = true;
                break;
            }
            now->relate.right->relate.left =
                tree.Insert({now->rectangle->y_int, 0}).first;
        }
        else
        {
            tree.Delete(now->relate.left);
        }
    }
    DestructHeap(alloc_ptr, rectangles.size());
    return found_overlap;
}
```

Chapter 14 Problems

14-1

(a)

Proof. Let $[a, b]$ be the interval of maximum overlap. In other word, $[a, b]$ is the intersection of the overlap segments. This says all points in $[a, b]$ have the same number of overlap segments. Notice a and b must be one of the endpoint of a segment that is included in the intersection. Hence there will always be a point of maximum overlap that is an endpoint of the segments. □

(b)

```
struct Node
{
    Node* parent;
    Node* left;
    Node* right;
    enum { BLACK, RED } color;
    T mapped_value;// only lower endpoint
    const Key key;
    enum { LOWER = +1, HIGHER = -1 } endpoint_type;
    Node* related_endpoint;// other endpoint of the current interval
    /**
     * x.type_sum = x.left.type_sum + x.endpoint_type + x.right.type_sum
     */
    int type_sum;
    /**
     * x.left_max_overlap_num = x.left.max_overlap_num
     * x.overlap_num = x.left.type_sum + x.endpoint_type
     * x.right_max_overlap_num = x.overlap_num + x.right.max_overlap_num
     * x.max_overlap_num =
     *       max(x.left_max_overlap_num, x.overlap_num, x.right_max_overlap_num)
     *
     * Notice that overlap_num only consider local overlap
     *       (in the subtree root at the node)
     * This says overlap_num is relatively
     */
    // node with endpoint which has maximum number of overlap segments
    Node* max_overlap_node;
    // maximum number of overlap segments
    int max_overlap_num;
```

```

    Node() : key(Key()) {}
    Node(const Key& key) : key(key) {}
};

template <class Key, class T>
bool IntervalTreePOM<Key, T>::MaintainAugmentedAttributesOfSingleNode(Node *node)
{
    int type_sum, max_overlap_num, overlap_num, right_max_overlap_num;
    Node *max_overlap_node;
    bool modified;
    /* maintain type_sum */
    type_sum = node->left->type_sum + node->endpoint_type + node->right->type_sum;
    /* maintain max_overlap */
    overlap_num = node->left->type_sum + node->endpoint_type;
    if (node->left != nil_ && node->left->max_overlap_num > overlap_num)
    {
        max_overlap_num = node->left->max_overlap_num;
        max_overlap_node = node->left->max_overlap_node;
    }
    else
    {
        max_overlap_num = overlap_num;
        max_overlap_node = node;
    }
    if (node->right != nil_)
    {
        right_max_overlap_num = overlap_num + node->right->max_overlap_num;
        if (right_max_overlap_num > max_overlap_num)
        {
            max_overlap_num = right_max_overlap_num;
            max_overlap_node = node->right->max_overlap_node;
        }
    }
    /* do modify */
    modified = false;
    if (node->type_sum != type_sum)
    {
        node->type_sum = type_sum;
        modified = true;
```

```
    }
    if (node->max_overlap_num != max_overlap_num)
    {
        node->max_overlap_num = max_overlap_num;
        modified = true;
    }
    if (node->max_overlap_node != max_overlap_node)
    {
        node->max_overlap_node = max_overlap_node;
        modified = true;
    }
    return modified;
}

// start to check augmented attributes from node->parent to root_
template <class Key, class T>
void IntervalTreePOM<Key, T>::FixAugmentedAttributes(Node* node)
{
    Key max;
    while (node != root_)
    {
        node = node->parent;
        if (MaintainAugmentedAttributesOfSingleNode(node) == false) break;
    }
}

template <class Key, class T>
std::pair<typename IntervalTreePOM<Key, T>::Iterator, bool>
    IntervalTreePOM<Key, T>::Insert(const ValueType& value)
{
    Node **node_ptr, *lower_node, *higher_node,  *parent;
    node_ptr = FindNodePtrByLowerKey(value.interval.low, &parent);
    if (*node_ptr == nil_)
    {
        /* insert lower endpoint */
        lower_node = new Node(value.interval.low);
        *node_ptr = lower_node;
        lower_node->parent = parent;
        lower_node->left = nil_;
        lower_node->right = nil_;
```

```
            lower_node->color = Node::RED;
            lower_node->mapped_value = value.mapped_value;
            lower_node->endpoint_type = Node::LOWER;
            MaintainAugmentedAttributesOfSingleNode(lower_node);
            FixAugmentedAttributes(lower_node);
            InsertFixup(lower_node);
            /* insert higher endpoint */
            node_ptr = FindLeafNodePtrToInsertByKey(value.interval.high, &parent);
            higher_node = new Node(value.interval.high);
            *node_ptr = higher_node;
            lower_node->related_endpoint = higher_node;
            higher_node->related_endpoint = lower_node;
            higher_node->parent = parent;
            higher_node->left = nil_;
            higher_node->right = nil_;
            higher_node->color = Node::RED;
            higher_node->endpoint_type = Node::HIGHER;
            MaintainAugmentedAttributesOfSingleNode(higher_node);
            FixAugmentedAttributes(higher_node);
            InsertFixup(higher_node);
            return std::make_pair(Iterator(lower_node, this), true);
        }
        else
        {
            return std::make_pair(Iterator(nil_, this), false);
        }
    }

    template <class Key, class T>
    void IntervalTreePOM<Key, T>::Delete(Iterator pos)
    {
        DeleteNode(pos.node_->related_endpoint);
        DeleteNode(pos.node_);
    }

    template <class Key, class T>
    void IntervalTreePOM<Key, T>::DeleteNode(Node *node)
    {
        Node *replaced, *replaced_replaced;
        bool is_black_deleted;
```

```
        replaced = node;
        is_black_deleted = replaced->color == Node::BLACK;
        if (replaced->left == nil_)
        {
            replaced_replaced = replaced->right;
            Transplant(replaced, replaced_replaced);
            FixAugmentedAttributes(replaced_replaced);// replaced_replaced will NOT be check
        }
        else if (replaced->right == nil_)
        {
            replaced_replaced = replaced->left;
            Transplant(replaced, replaced_replaced);
            FixAugmentedAttributes(replaced_replaced);// replaced_replaced will NOT be check
        }
        else
        {
            replaced = TreeMinimum(node->right);
            is_black_deleted = replaced->color == Node::BLACK;
            replaced_replaced = replaced->right;
            if (replaced->parent == node)
            {
                replaced_replaced->parent = replaced;
            }
            else
            {
                Transplant(replaced, replaced_replaced);
                replaced->right = node->right;
                replaced->right->parent = replaced;
            }
            Transplant(node, replaced);
            replaced->left = node->left;
            replaced->left->parent = replaced;
            replaced->color = node->color;
            FixAugmentedAttributes(replaced_replaced);// replaced_replaced will NOT be check
            FixAugmentedAttributes(replaced->right);// so replaced will be checked
        }
        if (is_black_deleted)
            DeleteFixup(replaced_replaced);
        delete node;
    }
```

```
template <class Key, class T>
Key IntervalTreePOM<Key, T>::FindPOM()
{
    return root_->max_overlap_node->key;
}
```