

# Chapter 14 Solusion

frc6.com

i@frc6.com

<https://github.com/frc123/CLRS>

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

```
1     template <class Key, class T>
2     typename OrderStatisticsTree<Key, T>::Node* OrderStatisticsTree<Key, T>::Select
3         (Node* subtree_root_node, size_t rank)
4     {
5         size_t root_rank;
6         root_rank = subtree_root_node->left->size + 1;
7         while (rank != root_rank)
```

```

8      {
9          if (rank < root_rank)
10         {
11             subtree_root_node = subtree_root_node->left;
12         }
13         else
14         {
15             subtree_root_node = subtree_root_node->right;
16             rank -= root_rank;
17         }
18         root_rank = subtree_root_node->left->size + 1;
19     }
20     return subtree_root_node;
21 }

```

#### 14.1-4

```

1     template <class Key, class T>
2     size_t OrderStatisticsTree<Key, T>::Rank(Node* node)
3     {
4         if (node == root_)
5             return node->left->size + 1;
6         else if (node == node->parent->left)
7             return Rank(node->parent) - node->right->size - 1;
8         else
9             return Rank(node->parent) + node->left->size + 1;
10    }

```

#### 14.1-5

```

1   $r = \text{OS-RANK}(T, x)$ 
2   $\text{succ} = \text{OS-SELECT}(T.\text{root}, r + i)$ 

```

The result is *succ*.

#### 14.1-6

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

```

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

```

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

```
 $y.rank = z.rank$ 
```

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

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

```

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

```

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

```

1      template <typename Key>
2      size_t CountInversions(std::vector<Key> array)
3      {
4          size_t inversions, i, rank;
5          OrderStatisticsTree<Key, int> tree;
6          std::pair<typename OrderStatisticsTree<Key, int>::Iterator, bool> insert_result;
7          inversions = 0;
8          for (i = 0; i < array.size(); ++i)
9              {
10                 insert_result = tree.Insert({array[i], 0});
11                 rank = tree.Rank(insert_result.first);
12                 inversions += (1 + i - rank);
13             }
14          return inversions;
15      }

```

## 14.1-8

```

1      size_t CountIntersections(std::vector<Chord> chords)
2      {
3          size_t intersections, i, rank_a, rank_b, rank_diff_1, rank_diff_2;
4          OSTree tree;
5          std::pair<typename OSTree::Iterator, bool> insert_result_a, insert_result_b;
6          intersections = 0;
7          for (i = 0; i < chords.size(); ++i)
8          {
9              insert_result_a = tree.Insert({chords[i].endpoint_a, 0});
10             insert_result_b = tree.Insert({chords[i].endpoint_b, 0});
11             rank_a = tree.Rank(insert_result_a.first);
12             rank_b = tree.Rank(insert_result_b.first);
13             if (rank_a > rank_b) std::swap (rank_a, rank_b);
14             // rank_a must smaller than rank_b
15             rank_diff_1 = rank_b - rank_a - 1;
16             rank_diff_2 = tree.Size() - rank_b + rank_a - 1;
17             intersections += std::min(rank_diff_1, rank_diff_2);
18         }
19         return intersections;
20     }

```

## 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 minimum 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:

```

1  if  $y == T.nil$ 
2       $T.root = z$ 
3       $z.succ = T.nil$ 
4       $z.prev = T.nil$ 
5  elseif  $z.key < y.key$ 
6       $y.left = z$ 
7       $z.succ = y$ 
8       $z.prev = y.prev$ 
9  else  $y.right = z$ 
10      $z.prev = y$ 
11      $z.succ = y.succ$ 
12  $z.succ.prev = z$ 
13  $z.prev.succ = z$ 

```

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

```

1   $z.prev.succ = z.succ$ 
2   $z.succ.prev = z.prev$ 

```

## 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:

```
1   $x.p.f = x.f$ 
2   $x.f = x.left.f \otimes x.right.f \otimes x.a$ 
```

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

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

### 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( $x, a, b$ )

```
1  if  $a \leq x.key$  and  $x.key \leq b$ 
2      OUTPUT( $x$ )
3  if  $a \leq x.key$  and  $x.left \neq T.nil$ 
4      RB-ENUMERATE( $x.left, a, b$ )
5  if  $x.key \leq b$  and  $x.right \neq T.nil$ 
6      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( $T, a, b$ )

```
1   $k = a$ 
2  OUTPUT( $k$ )
3  repeat
4       $k = k.succ$ 
5      OUTPUT( $k$ )
6  until  $k == b$ 
```

## 14.3

### 14.3-1

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

```
1   $y.max = x.max$ 
2   $x.max = \max(x.int.high, x.left.max, x.right.max)$ 
```

**14.3-2**INTERVAL-SEARCH( $T, i$ )

```

1   $x = T.root$ 
2  while  $x \neq T.nil$  and  $i$  does not overlap  $x.int$ 
3      if  $x.left \neq T.nil$  and  $x.left.max > i.low$ 
4           $x = x.left$ 
5      else  $x = x.right$ 
6  return  $x$ 

```

**14.3-3**

Iterative version:

INTERVAL-SEARCH-MIN( $T, i$ )

```

1   $x = T.root$ 
2   $smallest = T.nil$ 
3  while  $x \neq T.nil$ 
4      if  $x.left \neq T.nil$  and  $x.left.max \geq i.low$ 
5          if  $i$  overlaps  $x.int$ 
6               $smallest = x$ 
7               $x = x.left$ 
8      else if  $i$  overlaps  $x.int$ 
9          return  $x$ 
10      $x = x.right$ 
11 return  $smallest$ 

```

Recursive version:

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

```

1  if  $x.left \neq T.nil$  and  $x.left.max \geq i.low$ 
2       $smallest = \text{INTERVAL-SEARCH-MIN}(T, x.left, i)$ 
3      if  $smallest \neq T.nil$ 
4          return  $smallest$ 
5      elseif  $i$  overlaps  $x.int$ 
6          return  $x$ 
7      else return  $T.nil$ 
8  else if  $i$  overlaps  $x.int$ 
9      return  $x$ 
10 else return  $\text{INTERVAL-SEARCH-MIN}(T, x.right, i)$ 

```

**14.3-4**LIST-OVERLAP-INTERVAL( $T, x, i$ )

```

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

```

**14.3-5**INTERVAL-SEARCH-EXACTLY( $T, i$ )

```

1   $x = T.root$ 
2  while  $x \neq T.nil$  and  $(x.int.low \neq i.low$  or  $x.int.high \neq i.high)$ 
3      if  $i.high > x.max$ 
4           $x = T.nil$ 
5      elseif  $i.low < x.low$ 
6           $x = x.left$ 
7      elseif  $i.low > x.low$ 
8           $x = x.right$ 
9      else  $x = T.nil$ 
10 return  $x$ 

```

**14.3-6**

Consider to augment the red-black tree by adding attribute *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  $Q.root.min-gap$ , 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  $x.min-gap$  depends on only the information in nodes  $x$ ,  $x.left$ , and  $x.right$ .

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

$$Q.nil.min-gap = \infty$$

$$Q.nil.min = \infty$$

$$Q.nil.max = -\infty$$

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  $x.min-gap$  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

```

1      bool DetermineOverlapRectangles(const std::vector<Rectangle>& rectangles)
2      {
3          bool found_overlap;
4          RectangleWithXCoordIndex *alloc_ptr, *now;
5          std::allocator<RectangleWithXCoordIndex> alloc;
6          std::vector<RectangleWithXCoordIndex*> heap;
7          Tree tree; // interval tree
8          found_overlap = false;
9          // sort the x-coordinates with the minimum heap
10         alloc_ptr = ConstructHeap(rectangles, heap);
11         while (heap.size() > 0)
12         {
13             now = HeapExtractMin(heap);
14             if (now->coord_pos == RectangleWithXCoordIndex::LEFT)
15             {
16                 if (tree.Find(now->rectangle->y_int) != tree.End())
17                 {
18                     found_overlap = true;
19                     break;
20                 }
21                 now->relate.right->relate.left =
22                     tree.Insert({now->rectangle->y_int, 0}).first;
23             }
24             else
25             {
26                 tree.Delete(now->relate.left);
27             }
28         }
29         DestructHeap(alloc_ptr, rectangles.size());
30         return found_overlap;
31     }

```

## 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.  $\square$

(b)

```

1      struct Node
2      {
3          Node* parent;
4          Node* left;
5          Node* right;
6          enum { BLACK, RED } color;
7          T mapped_value; // only lower endpoint
8          const Key key;
9          enum { LOWER = +1, HIGHER = -1 } endpoint_type;
10         Node* related_endpoint; // other endpoint of the current interval
11         /**
12          * x.type_sum = x.left.type_sum + x.endpoint_type + x.right.type_sum
13          */
14         int type_sum;
15         /**
16          * x.left_max_overlap_num = x.left.max_overlap_num
17          * x.overlap_num = x.left.type_sum + x.endpoint_type
18          * x.right_max_overlap_num = x.overlap_num + x.right.max_overlap_num
19          * x.max_overlap_num =
20          * max(x.left_max_overlap_num, x.overlap_num, x.right_max_overlap_num)
21          *
22          * Notice that overlap_num only consider local overlap
23          * (in the subtree root at the node)
24          * This says overlap_num is relatively
25          */
26         // node with endpoint which has maximum number of overlap segments
27         Node* max_overlap_node;
28         // maximum number of overlap segments
29         int max_overlap_num;

```

```

30
31     Node() : key(Key()) {}
32     Node(const Key& key) : key(key) {}
33 };
34
35 template <class Key, class T>
36 bool IntervalTreePOM<Key, T>::MaintainAugmentedAttributesOfSingleNode(Node *node)
37 {
38     int type_sum, max_overlap_num, overlap_num, right_max_overlap_num;
39     Node *max_overlap_node;
40     bool modified;
41     /* maintain type_sum */
42     type_sum = node->left->type_sum + node->endpoint_type + node->right->type_sum;
43     /* maintain max_overlap */
44     overlap_num = node->left->type_sum + node->endpoint_type;
45     if (node->left != nil_ && node->left->max_overlap_num > overlap_num)
46     {
47         max_overlap_num = node->left->max_overlap_num;
48         max_overlap_node = node->left->max_overlap_node;
49     }
50     else
51     {
52         max_overlap_num = overlap_num;
53         max_overlap_node = node;
54     }
55     if (node->right != nil_)
56     {
57         right_max_overlap_num = overlap_num + node->right->max_overlap_num;
58         if (right_max_overlap_num > max_overlap_num)
59         {
60             max_overlap_num = right_max_overlap_num;
61             max_overlap_node = node->right->max_overlap_node;
62         }
63     }
64     /* do modify */
65     modified = false;
66     if (node->type_sum != type_sum)
67     {
68         node->type_sum = type_sum;
69         modified = true;

```

```

70     }
71     if (node->max_overlap_num != max_overlap_num)
72     {
73         node->max_overlap_num = max_overlap_num;
74         modified = true;
75     }
76     if (node->max_overlap_node != max_overlap_node)
77     {
78         node->max_overlap_node = max_overlap_node;
79         modified = true;
80     }
81     return modified;
82 }
83
84 // start to check augmented attributes from node->parent to root_
85 template <class Key, class T>
86 void IntervalTreePOM<Key, T>::FixAugmentedAttributes(Node* node)
87 {
88     Key max;
89     while (node != root_)
90     {
91         node = node->parent;
92         if (MaintainAugmentedAttributesOfSingleNode(node) == false) break;
93     }
94 }
95
96 template <class Key, class T>
97 std::pair<typename IntervalTreePOM<Key, T>::Iterator, bool>
98     IntervalTreePOM<Key, T>::Insert(const ValueType& value)
99 {
100     Node **node_ptr, *lower_node, *higher_node, *parent;
101     node_ptr = FindNodePtrByLowerKey(value.interval.low, &parent);
102     if (*node_ptr == nil_)
103     {
104         /* insert lower endpoint */
105         lower_node = new Node(value.interval.low);
106         *node_ptr = lower_node;
107         lower_node->parent = parent;
108         lower_node->left = nil_;
109         lower_node->right = nil_;

```

```

110         lower_node->color = Node::RED;
111         lower_node->mapped_value = value.mapped_value;
112         lower_node->endpoint_type = Node::LOWER;
113         MaintainAugmentedAttributesOfSingleNode(lower_node);
114         FixAugmentedAttributes(lower_node);
115         InsertFixup(lower_node);
116         /* insert higher endpoint */
117         node_ptr = FindLeafNodePtrToInsertByKey(value.interval.high, &parent);
118         higher_node = new Node(value.interval.high);
119         *node_ptr = higher_node;
120         lower_node->related_endpoint = higher_node;
121         higher_node->related_endpoint = lower_node;
122         higher_node->parent = parent;
123         higher_node->left = nil_;
124         higher_node->right = nil_;
125         higher_node->color = Node::RED;
126         higher_node->endpoint_type = Node::HIGHER;
127         MaintainAugmentedAttributesOfSingleNode(higher_node);
128         FixAugmentedAttributes(higher_node);
129         InsertFixup(higher_node);
130         return std::make_pair(Iterator(lower_node, this), true);
131     }
132     else
133     {
134         return std::make_pair(Iterator(nil_, this), false);
135     }
136 }
137
138 template <class Key, class T>
139 void IntervalTreePOM<Key, T>::Delete(Iterator pos)
140 {
141     DeleteNode(pos.node->related_endpoint);
142     DeleteNode(pos.node_);
143 }
144
145 template <class Key, class T>
146 void IntervalTreePOM<Key, T>::DeleteNode(Node *node)
147 {
148     Node *replaced, *replaced_replaced;
149     bool is_black_deleted;

```

```

150     replaced = node;
151     is_black_deleted = replaced->color == Node::BLACK;
152     if (replaced->left == nil_)
153     {
154         replaced_replaced = replaced->right;
155         Transplant(replaced, replaced_replaced);
156         FixAugmentedAttributes(replaced_replaced); // replaced_replaced will NOT be checked
157     }
158     else if (replaced->right == nil_)
159     {
160         replaced_replaced = replaced->left;
161         Transplant(replaced, replaced_replaced);
162         FixAugmentedAttributes(replaced_replaced); // replaced_replaced will NOT be checked
163     }
164     else
165     {
166         replaced = TreeMinimum(node->right);
167         is_black_deleted = replaced->color == Node::BLACK;
168         replaced_replaced = replaced->right;
169         if (replaced->parent == node)
170         {
171             replaced_replaced->parent = replaced;
172         }
173         else
174         {
175             Transplant(replaced, replaced_replaced);
176             replaced->right = node->right;
177             replaced->right->parent = replaced;
178         }
179         Transplant(node, replaced);
180         replaced->left = node->left;
181         replaced->left->parent = replaced;
182         replaced->color = node->color;
183         FixAugmentedAttributes(replaced_replaced); // replaced_replaced will NOT be checked
184         FixAugmentedAttributes(replaced->right); // so replaced will be checked
185     }
186     if (is_black_deleted)
187         DeleteFixup(replaced_replaced);
188     delete node;
189 }

```

```

190
191     template <class Key, class T>
192     Key IntervalTreePOM<Key, T>::FindPOM()
193     {
194         return root_->max_overlap_node->key;
195     }

```

## 14-2

(a)

```

1     /**
2     * running time: O(mn)
3     * caller is responsible to deallocate the return value
4     * n, m must greater than 0
5     */
6     std::vector<int>* GetJosephusPermutation(int n, int m)
7     {
8         Node *nodes, **ptr;
9         std::vector<int>* result;
10        int i;
11        /* allocate nodes */
12        nodes = new Node[n];
13        /* build circular singly linked list */
14        for (i = 0; i < n - 1; ++i)
15        {
16            nodes[i].key = i + 1;
17            nodes[i].next = &(nodes[i + 1]);
18        }
19        nodes[i].key = i + 1;
20        nodes[i].next = nodes; // &(nodes[0])
21        /* allocate vector */
22        result = new std::vector<int>;
23        result->reserve(n);
24        /* output */
25        /* find the m-th key */
26        ptr = &(nodes[n - 1].next);
27        for (i = 1; i < m; ++i) ptr = &((*ptr)->next);
28        while (true)
29        {
30            /* add the key into the permutation */

```

```

31         result->push_back((*ptr)->key);
32         /* remove the key */
33         if (*ptr == (*ptr)->next) break;
34         *ptr = (*ptr)->next;
35         /* find the next m-th key */
36         for (i = 1; i < m; ++i) ptr = &((*ptr)->next);
37     }
38     /* deallocate nodes */
39     delete[] nodes;
40     /* return */
41     return result;
42 }

```

(b)

```

1  /**
2   * running time: O(nlgn)
3   * caller is responsible to deallocate the return value
4   * n, m must greater than 0
5   */
6  std::vector<int>* GetJosephusPermutation(int n, int m)
7  {
8      int i, rank;
9      OrderStatisticsTree<int, int> ostree;
10     OrderStatisticsTree<int, int>::Iterator it;
11     std::vector<int>* result;
12     /* build order statistics tree: O(nlgn) */
13     for (i = 1; i <= n; ++i) ostree.Insert(std::make_pair(i, 0));
14     /* allocate vector */
15     result = new std::vector<int>;
16     result->reserve(n);
17     /* output: O(nlgn) */
18     rank = m;
19     for (i = n; i > 0; --i) // n times
20     {
21         rank = ((rank - 1) % i) + 1;
22         it = ostree.Select(rank); // O(lgn)
23         result->push_back(it->first);
24         ostree.Delete(it); // O(lgn)
25         rank += (m - 1);

```



```
26     }  
27     return result;  
28 }
```