

# 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

```
1     template <class Key, class T>
2     typename AugmentRedBlackTree<Key, T>::Node* AugmentRedBlackTree<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 AugmentRedBlackTree<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      AugmentRedBlackTree<Key, int> tree;
6      std::pair<typename AugmentRedBlackTree<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  $\Theta(m + \log(n))$  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  $\text{RB-ENUMERATE}(x, a, b)$  in  $\Theta(m + \log(n))$  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  $k == b$ 

```