Chapter 13. Red-Black Trees

13.1-1.
All of those trees can be constructed in same manner, despite possibly different colouring:
Root 8, First level 4, 12 (left from right), Second level 2, 6, 10, 14, Third level 1, 3, 5, 7, 9, 11, 13, 15.
Red-black tree of black-heights 2 could be constructed by colouring nodes in the first and the third level as red.
Red-black tree of black-heights 3 could be constructed by colouring nodes in the first level as red.
Red-black tree of black-heights 4 could be constructed by colouring all nodes as black.

13.1-2.
No, becuase 36(red) will be a right child of 35(red).
No again, because the black-height from 38(black) will not be well-defined.

13.1-3.
Yes, because the only adjusted node is the root, and the black-height from the root is still well-defined, increased by 1.

13.1-4.
The degree of each black node can varies from 2 to 4, depending on the number of red nodes in its former children. The black-height remains same for all nodes.

13.1-5.
The longest path should alternate black-red-black-red-... nodes. The shortest path should consist of all black nodes. Since the two paths contain equal numbers of black nodes, the length of the longest path is at most twice the

length of the shortest path.

13.1-6.
The largest possible number comes from the complete binary tree with red layer and black layer alternating, which has $2^{2k} - 1$ internal nodes.
The smallest possible number comes from the complete binary tree with all black nodes, which has $2^k - 1$ internal nodes.

13.1-7.
The largest ratio is 2.
The smallest ratio is 0.

13.2-1.

Algorithm 1: RIGHT-ROTATE(T, x)

```
y = x.left;
x.left = y.right;
if $y.right \neq T.nil$ then
    y.right.p = x;
y.p = x.p;
if $x.p == T.nil$ then
    T.root = y;
else if $x == x.p.right$ then
    x.p.right = y;
else
    x.p.left = y;
y.right = x;
x.p = y;
```

13.2-2.
Every rotation with respect to some node corresponds to the parent of the node. Since there is only one node that does not have a parent, there are exactly $n - 1$ distinct rotations.

13.2-3.
a : Increase by 1.
b : Unchanged.
c : Decrease by 1.

13.2-4.
We first show that we can convert any binary search tree into a single right-going chain with at most $n - 1$ right rotations. Define the right spine as the root and all descendants of the root that are reachable by following only right pointers from the root. A right-going chain is a binary search tree with all nodes in the right spine. If the tree is not a right-going chain, we can find some node on the right spine that has a left child so that we can perform a right rotation. This right rotation increases the number of nodes in the right spine by 1. By repeating the sequence of at most $n - 1$ right rotations, it is guaranteed to convert any binary search tree to a right-going chain. This procedure can be reversed, so we can transform any binary search tree to any other binary search tree with at most $2n - 2$ rotations.

13.2-5.
Counterexample : T_1 is a binary search tree with two nodes that its root have the right child. T_2 is the one with the left child.
If the root nodes of T_1 and T_2 are different, then the root node of T_1 can be changed to the root node of T_2 by $O(n)$ right rotations. This can be recursively applied by the right subtree of the root. We have $T(n) = T(n-1) + O(n)$, so $T(n) = O(n^2)$.

13.3-1.
Because doing so violates the property 5.

13.3-2.
Root 38 (black), Level 1 - left to right - 19 (red), 41 (black), Level 2 - child of 19 - 12 (black), 31(black), Level 3 - child of 12 - 8 (red).

13.3-3.
In figure 13.5, A, B and D have black-height $k + 1$. C has black-height $k + 1$ on the left and $k + 2$ on the right.
In figure 13.6, A, B and C have black-height $k + 1$.

13.3-4.
Colors are set to red only in cases 1 and 3, where $z.p.p$ is coloured. If $z.p.p$ is $T.nil$, then $z.p$ is the root, so in this case we don't enter the while loop. In case 2 before falling through to case 3, LEFT-ROTATE doesn't change $z.p.p$, so there's no problem.

13.3-5.
For $n = 2$, there is exactly one red node and one black node (root). For $n > 2$, if the loop terminates with case 1, the inserted node must be red. For case 2 and 3, the number of red nodes does not decrease. Hence, there must be at least one red node if $n > 1$.

13.3-6.
We can use stack to record the path to the inserted node.

Algorithm 2: RB-INSERT(T, z)

```
Let S be a stack;
y = T.nil;
x = T.root;
while $x \neq T.nil$ do
    y = x;
    S.push(x);
    if $z.key$ ¡ $x.key$ then
        x = x.left;
    else
        x = x.right;
if $y == T.nil$ then
    T.root = z;
else if $z.key$ ¡ $y.key$ then
    y.left = z;
else
    y.right = z;
z.left = T.nil;
z.right = T.nil;
z.color = RED;
RB-INSERT-FIXUP(T, S, z);
```

Algorithm 3: RB-INSERT-FIXUP(T, S, z)

```
while S.top().color == RED do
    p = S.top();
    S.pop();
    if p == S.top().left then
        y = S.top().right;
        if y.color == RED then
            p.color = BLACK;
            y.color = BLACK;
            S.top().color = RED;
            z = S.top();
            S.pop();
        else
            if z == p.right then
                z = p;
                LEFT-ROTATE(T, z);
            p.color = BLACK;
            S.top().color = RED;
            RIGHT-ROTATE(T, S.top());
    else
        y = S.top().left;
        if y.color == RED then
            p.color = BLACK;
            y.color = BLACK;
            S.top().color = RED;
            z = S.top();
            S.pop();
        else
            if z == p.left then
                z = p;
                RIGHT-ROTATE(T, z);
            p.color = BLACK;
            S.top().color = RED;
            LEFT-ROTATE(T, S.top());
```

13.4-1.
It suffices to check all four cases in RB-DELETE-FIXUP.
Case 1 converts into one of case 2, 3, or 4.

Case 3 converts into case 4.
If the loop terminates with case 2 or 4, it means that the new x is T.root. It is set to black at the end of the algorithm.

13-4.2.
If in RB-DELETE, both x and $x.p$ are red, then the first two cases of if/elseif clauses cannot hold, because z and z.left (or z.right) cannot be both red at the beginning of RB-DELETE. Therefore, this can only happen when y-original-color is black, z.color is red and x.color is red. Since y-original-color is black, y's original sibling (which becomes x's sibling right after RB-TRANSPLANT(T, y, y.right)) must be black. Therefore, x is the only red child of x.p. When calling RB-DELETE-FIXUP, the whole algorithm just sets x to be black, restoring the property 4.

13.4-3.
Initial: 38(black) / 19(red), 41(black) / Children of 19 - 12(black), 31(black) / Children of 12 - 8(red)
Delete 8: 38(black) / 19(red), 41(black) / Children of 19 - 12(black), 31(black)
Delete 12: 38(black) / 19(black), 41(black) / Children of 19 - 31(red)
Delete 19: 38(black) / 31(black), 41(black)
Delete 31: 38(black) / 41(red)
Delete 38: 41(black)
Delete 41: NIL(black)

13.4-4.
When the node y in RB-DELETE has no children, we have $x = T.nil$ when calling RB-DELETE-FIXUP(T, x).
When the root is deleted, we have $x = T.nil$, and we restore the color of x back to black.

13.4-5.
Case 1 : All subtrees have count of 1.
Case 2 : For α and β, the count is count(c). For the rest, the count is count$(c) + 1$.
Case 3 : For ϵ and ζ, the count is count$(c) + 1$. For the rest, the count is count(c).
Case 4 : For α and β, the count is count(c). For γ and δ, it is count$(c) +$ count(c'). For ϵ and ζ, it is count(c).

13.4-6.
Case 1 occurs only if x's sibling is red. If x.p were red, then we have both x.p and x.sibling as red at the beginning of RB-DELETE, a contradiction.

13.4-7.
No. Consider the red-black tree with two nodes with 3 (root, black) - 2 (left, red). If we insert 1 and delete 1, the tree becomes 2 (root, black) - 3 (right, red).

13-1.
a. When inserting a node k, all nodes on the path from the root to the newly added node must be changed.
When deleting a node y, if y has at most one child, then it is enough to splice out y from the path from the root. All ancestors in the path must be changed. If y has two children, let z be the transplanted node instead of y. In this case, z is the node actually removed. y and all ancestors of y must be changed.
b. First, we set two subroutines: MAKE-NEW-NODE(k) creates a new node with key k, returning a pointer to the new node. COPY-NODE(x) creates a clone of node x, returning a pointer to the new node. PERSISTENT-TREE-INSERT goes as follows:

Algorithm 4: PERSISTENT-TREE-INSERT(root, k)

```
x = NIL;
if root == NIL then
    x = MAKE-NEW-NODE(k);
else
    x = COPY-NODE(root);
    if $k <$ root.key then
        x.left = PERSISTENT-TREE-INSERT(root.left, k);
    else
        x.right = PERSISTENT-TREE-INSERT(root.right, k);
return x;
```

c. PERSISTENT-TREE-INSERT does a constant amount of work for each node along the path from the root to the new leaf, taking $O(h)$ time.
Since it newly allocates all of ancestors of the inserted node, the consumed

space is $O(h)$.
d. If there were parent fields, since the root is changed, every descendant of the root should be changed, that is, a whole set of nodes in the tree should be cloned, so it takes $\Omega(n)$ space and $\Omega(n)$ time.
e. To maintain $O(\lg n)$ insertion time, we track the path from the root to the newly inserted node using a stack, tracking the pointers to the ancestors without storing parents directly. To distinguish nodes with the same key, we can introduce a second part to each key and to use this as a tiebreaker. For rotation and recoloring in RB-INSERT-FIXUP, it performs at most 2 rotations and each rotation changes at most 3 nodes, so at most 6 nodes are directly modified. The changed nodes share a single $O(\lg n)$ length path of ancestors, so it takes $O(\lg n)$ time. The recolored nodes are uncles along the path, so the number of them are still $O(\lg n)$.
We can maintain $O(\lg n)$ deletion time in a similar manner.

13-2.
a. For RB-INSERT, if at the end of the iteration we reach the red root, we color it as black and increase bh by 1. For RB-DELETE, if at the end of the iteration we reach the root, we decrease bh by 1. While descending through T, we can increase bh by 1 each time we find a black node.
b. If the node has a right child, move to its right child. Otherwise, move to the left child. If the node is black, decrease bh by 1. Repeat this until bh equals $T_2.bh$.
c. Let x be the new root and T_y be the left subtree of x, T_2 be the right subtree of x. The time complexity is $O(1)$.
d. Red. Then we call RB-INSERT-FIXUP(T_1, x) to enforce red-black properties in $O(\lg n)$ time.
e. The algorithm in part b. can be applied in exactly symmetric manner.
f. We combine part c. and part d. to get $O(\lg n)$ time.

13-3.
a. Let $T(h)$ be the minimum number of nodes of an AVL tree of height h. To have the minimum number of nodes, one child of the root should have height $h-1$ and one child should have height $h-2$. So we have that $T(h) \geq T(h-1) + T(h-2) + 1$. Therefore, $T(h) \geq F_h$, where F_h is the h-th Fibonacci number. Since $F_h = \Theta((\frac{1+\sqrt{5}}{2})^n)$, we have $h = O(\lg n)$.
b.

Algorithm 5: BALANCE(x)

```
if $|x.left.h - x.right.h| \leq 1$ then
    return x;
else if $x.left.h > x.right.h$ then
    y = x.left;
    if $y.left.h > y.right.h$ then
        LEFT-ROTATE(y);
    return RIGHT-ROTATE(x);
else
    y = x.right;
    if $y.left.h > y.right.h$ then
        RIGHT-ROTATE(y);
    return LEFT-ROTATE(x);
```

c.

Algorithm 6: AVL-INSERT(x, z)

```
if $x == NIL$ then
    z.h = 0;
    return z;
if $z.key \leq x.key$ then
    y = AVL-INSERT(x.left, z);
    x.left = y;
    y.p = x;
    x.h = y.h + 1;
else
    y = AVL-INSERT(x.right, z);
    x.right = y;
    y.p = x;
    x.h = y.h + 1;
x = BALANCE(x);
```

d. Since the height of the AVL tree is $O(\lg n)$, the insertion takes $O(\lg n)$ time. In addition, there will be only a single rotation in BALANCE(x) at the end of AVL-INSERT, because when we do it, we decrease the height of the originally unbalanced subtree by 1. Therefore, all of its ancestors' heights are unaffected, so, no further rotation will be required.

13-4.

a. The root is the node with the smallest priority. We can divide the set of other nodes into two subsets based on the key. For each subtree, its root is again the node with the smallest priority among the subset. In this manner, given a set of nodes with keys and priorities all distinct, the associated treap is uniquely determined.
b. By part a, given a set of nodes with distinct keys, each treap corresponds to exactly one permutation of priorities. Hence the expected height of a treap is same with the expected height of a randomly built binary search tree, which equals $\Theta(\lg n)$.
c.

Algorithm 7: TREAP-INSERT(T, x)

```
TREE-INSERT(T, x);
while $x \neq T.root\ and\ x.priority < x.p.priority$ do
    if $x = x.p.left$ then
        RIGHT-ROTATE(T, x.p);
    else
        LEFT-ROTATE(T, x.p);
```

d. Since there are at most h rotations, and the ordinary binary search tree insertion takes $O(h)$ time, the expected running time is $\Theta(\lg n)$.
e. We prove it by induction. The base case is when x is the parent of y. We have a single rotation to make y as the new root with exactly one child, so $C + D = 1$. To prove the inductive step, observe that a left rotation increases C by one, and a right rotation increases D by one. Therefore, by the induction hypothesis, the total number of rotations is equal to $C + D$.
f. First we prove the if part. Assume that $X_{ik} = 0$.
- If y is in the right subtree of x, then $y.key \geq x.key$.
- If y is not in one of the subtrees of x, then there is a common ancestor z of x and y. Since $y.key < x.key$, we have $y.key < z.key < x.key$. We must have $z.priority < y.priority$, a contradiction.
- If y is in the left subtree of x but not on the right spine of the left subtree of x, then there is an ancestor z of y in the left subtree of x such that y is in the left subtree of z. Since we have $y.key < z.key < x.key$, we must have $z.priority < y.priority$, a contradiction.
Now we prove the only if part. Assume that $X_{ik} = 1$. Since y is in the left subtree of x, $y.priority > x.priority$ and $y.key < x.key$. If there is z such that $y.key < z.key < x.key$ but $z.priority < y.priority$, then z must be

inserted before y.
- If z is in the right spine of the left subtree of x, since y is in the right spine of the left subtree of x, y is in the right subtree of z. This contradicts to $y.key < z.key$.
- If z is in the left subtree of x but not in the right spine of the left subtree of x, then z is in the left subtree of some node w in the right spine of the left subtree of x. Again, y is in the right subtree of w, this contradicts to $y.key < z.key$.
- If z is not in one of the subtrees of x. Then z and x have a common ancestor w such that z is in the left subtree of w and x is in the right subtree of w. This implies $z.key < w.key < x.key$. Since $y.key < z.key$, y cannot be inserted into the right subtree of w, so it cannot be inserted in a subtree of x, a contradiction.

g. By part f, X_{ik} depends only on the relative ordering of the priorities of y, x and all z such that $y.key < z.key < x.key$. The keys of the items in question are $i, i+1, \cdots, k$. There are $(k-i+1)!$ permutations of these items, and there are $(k-i-1)!$ permutations where i has the minimum priority and k has the maximum priority. Therefore,

$$P(X_{ik} = 1) = \frac{(k-i-1)!}{(k-i+1)!} = \frac{1}{(k-i+1)(k-i)}.$$

h. $\mathbb{E}[C]$ is the expected number of nodes in the right spine of the left subtree of x, which equals the sum of $\mathbb{E}[X_{ik}]$ for all i in the tree. Therefore,

$$\mathbb{E}[C] = \sum_{i=1}^{k-1} \mathbb{E}[X_{ik}] = \sum_{i=1}^{k-1} P(X_{ik} = 1) = \sum_{i=1}^{k-1} \frac{1}{(k-i)(k-i+1)} = 1 - \frac{1}{k}.$$

i. For all items x, consider leaving $x.priority$ unchanged but replace $x.key$ with $n - x.key + 1$. Then we get the tree whose shape is exactly the mirror image of the original tree. The length of the left spine of the right subtree of x in the original tree is the length of the right spine of the left subtree of x in the mirrored tree. Therefore, we have

$$\mathbb{E}[D] = 1 - \frac{1}{n-k+1}.$$

j. The expected number of rotations is:

$$\mathbb{E}[C + D] = 1 - \frac{1}{k} + 1 - \frac{1}{n-k+1} \leq 2.$$