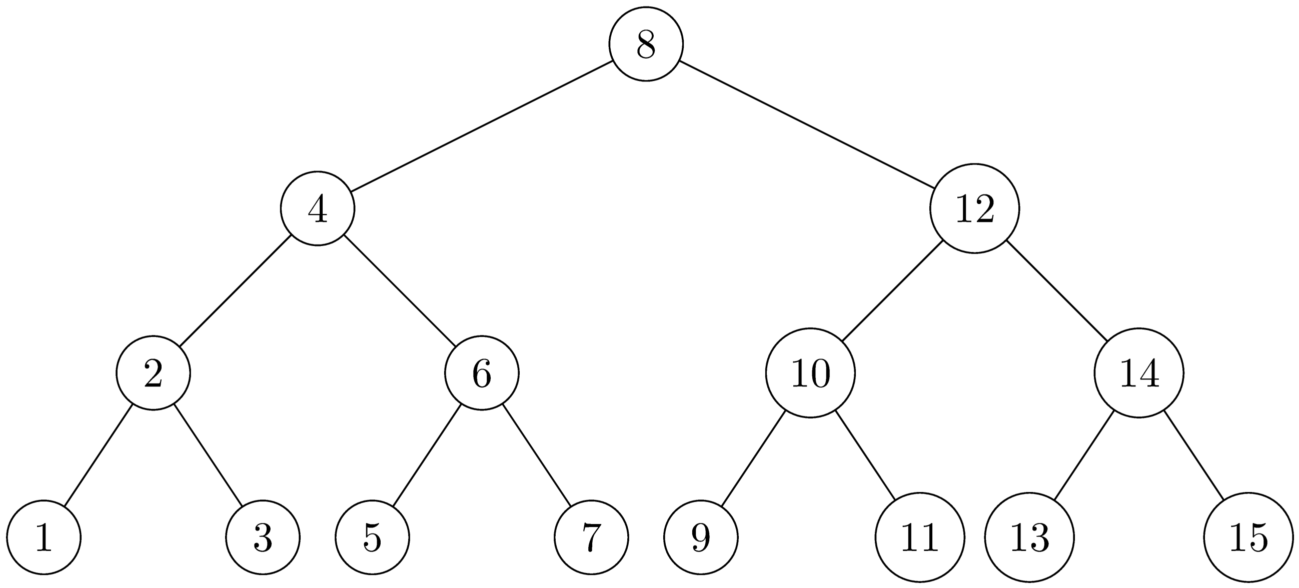
13 Red-Black Trees

13.1 Properties of red-black trees

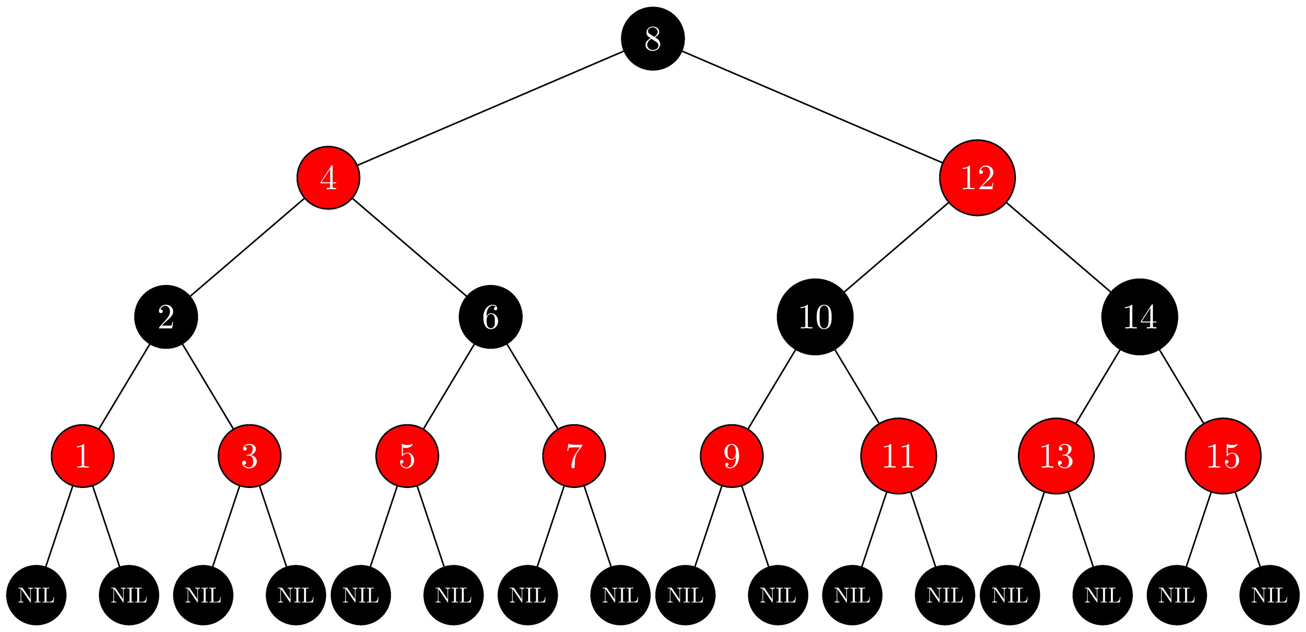
13.1-1

> In the style of Figure 13.1(a), draw the complete binary search tree of height 3 on the keys $\{1, 2, \ldots, 15\}$. Add the NIL leaves and color the nodes in three different ways such that the black-heights of the resulting red-black trees are 2, 3, and 4.

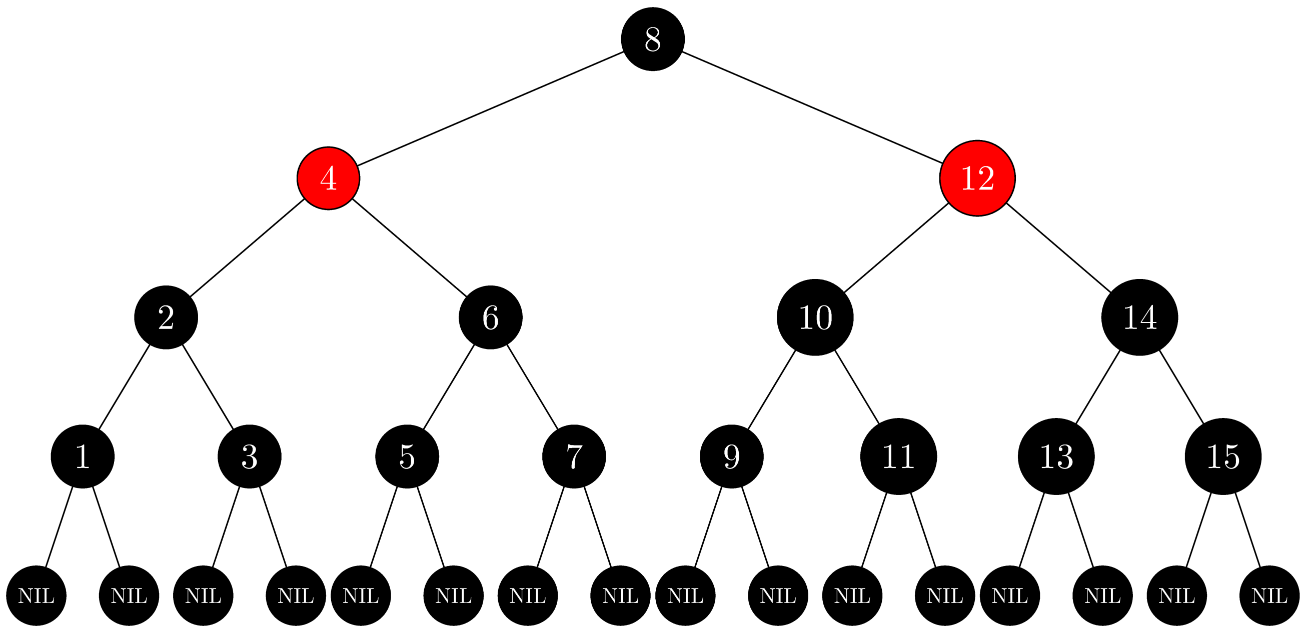
- Complete binary tree of $\text{height} = 3$:



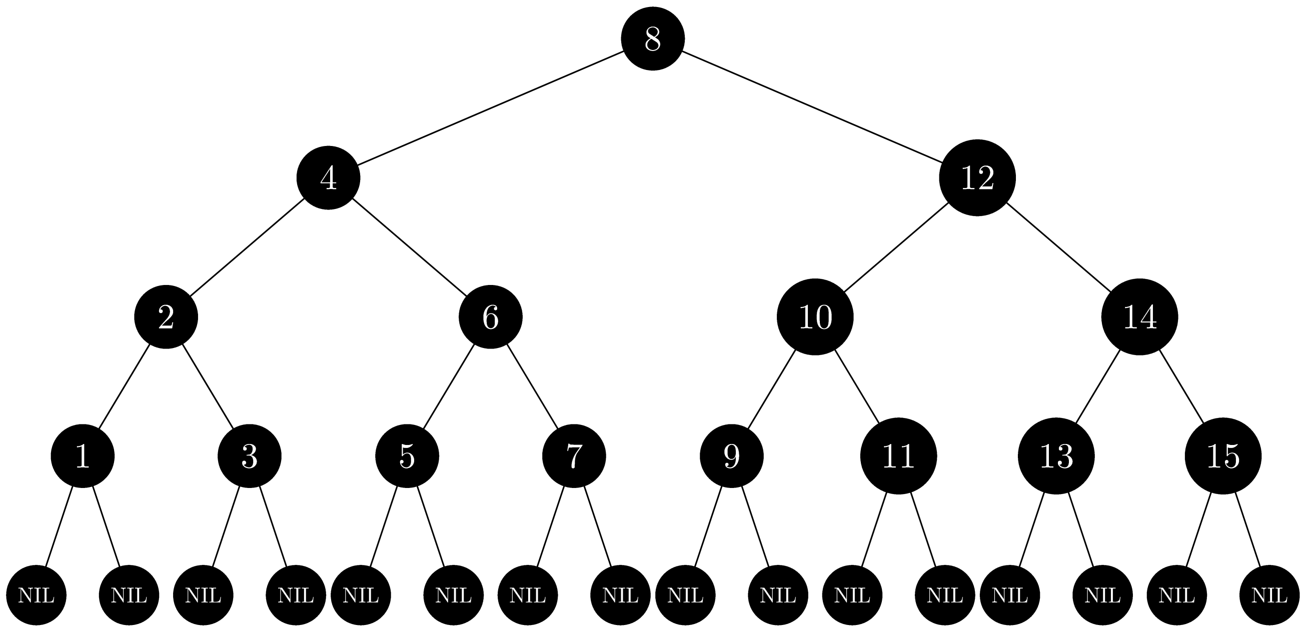
- Red-black tree of $\text{black-heights} = 2$:



- Red-black tree of $\text{black-heights} = 3$:



- Red-black tree of $\text{black-heights} = 4$:

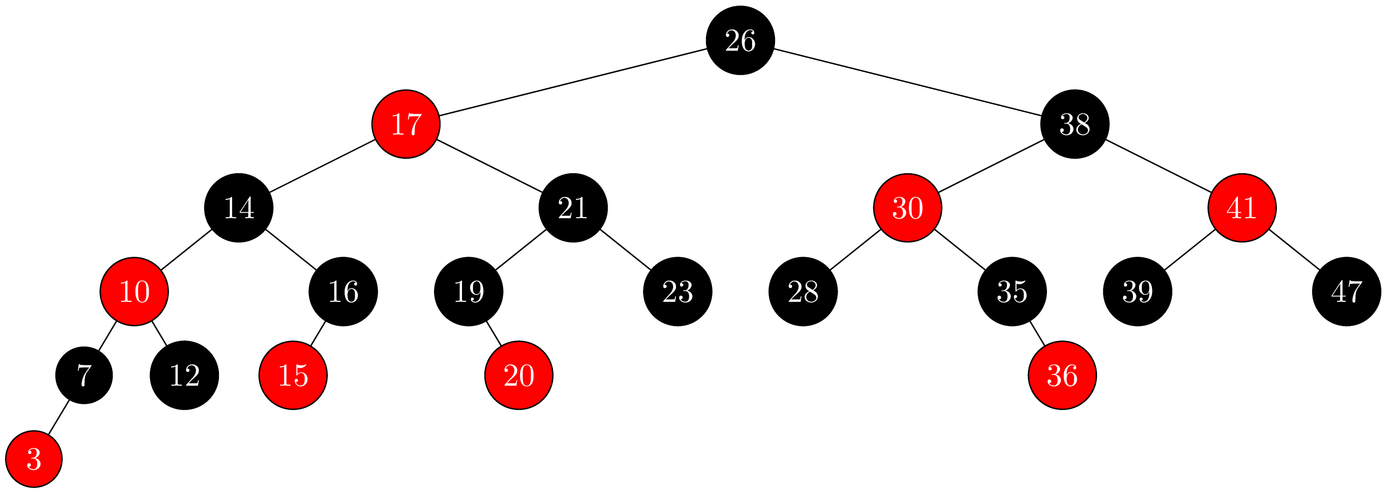


13.1-2

> Draw the red-black tree that results after TREE-INSERT is called on the tree in Figure 13.1 with key 36. If the inserted node is colored red, is the resulting tree a red-black tree? What if it is colored black?

- If the inserted node is colored red, the tree doesn't satisfy property 4 because 35 will be the parent of 36, which is also colored red.
- If the inserted node is colored black, the tree doesn't satisfy property 5 because there will be two paths from node 38 to $T.\text{nil}$ which contain different numbers of black nodes.

We don't draw the *wrong* red-black tree; however, we draw the adjusted correct tree:



13.1-3

> Let us define a ***relaxed red-black tree*** as a binary search tree that satisfies red-black properties 1, 3, 4, and 5. In other words, the root may be either red or black. Consider a relaxed red-black tree T whose

> root is red. If we color the root of T black but make no other changes to T, is the resulting tree a red-black tree?

Yes, it is.

- Property 1 is trivially satisfied since only one node is changed and it is not changed to some mysterious third color.
- Property 3 is trivially satisfied since no new leaves are introduced.
- Property 4 is satisfied since there was no red node introduced, and root is in every path from the root to the leaves, but no others.
- Property 5 is satisfied since the only paths we will be changing the number of black nodes in are those coming from the root. All of these will increase by 1, and so will all be equal.

13.1-4

> Suppose that we "absorb" every red node in a red-black tree into its black parent, so that the children of the red node become children of the black parent. (Ignore what happens to the keys.) What are the possible degrees of a black node after all its red children are absorbed? What can you say about the depths of the leaves of the resulting tree?

The degree of a node in a rooted tree is the number of its children (see Section B.5.2). Given this definition, the possible degrees are 0, 2, 3 and 4, based on whether the black node had zero, one or two red children, each with either zero or two black children. The depths could shrink by at most a factor of $1/2$.

13.1-5

> Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

Suppose we have the longest simple path $(a_1, a_2, \dots, a_s)$ and the shortest simple path $(b_1, b_2, \dots, b_t)$. Then, by property 5 we know they have equal numbers of black nodes. By property 4, we know that neither contains a repeated red node. This tells us that at most $\left\lfloor \frac{s-1}{2} \right\rfloor$ of the nodes in the longest path are red. This means that at least $\left\lceil \frac{s+1}{2} \right\rceil$ are black, so, $t \ge \left\lceil \frac{s+1}{2} \right\rceil$. Therefore, if, by way of contradiction, we had that $s > t \cdot 2$, then $t \ge \left\lceil \frac{s+1}{2} \right\rceil \ge \left\lceil \frac{2t+2}{2} \right\rceil = t + 1$ results a contradiction.

13.1-6

> What is the largest possible number of internal nodes in a red-black tree with black-height k? What is the smallest possible number?

- The largest is a path with half black nodes and half red nodes, which has $2^{2k} - 1$ internal nodes.
- The smallest is a path with all black nodes, which has $2^k - 1$ internal nodes.

13.1-7

> Describe a red-black tree on n keys that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio?

- The largest ratio is 2, each black node has two red children.
- The smallest ratio is 0.

13.2 Rotations

13.2-1

> Write pseudocode for RIGHT-ROTATE.

```
RIGHT-ROTATE(T, y)
    x = y.left
    y.left = x.right
    if x.right != T.nil
        x.right.p = y
    x.p = y.p
    if y.p == T.nil
        T.root = x
    else if y == y.p.right
        y.p.right = x
    else y.p.left = x
    x.right = y
    y.p = x
```

13.2-2

> Argue that in every n-node binary search tree, there are exactly $n - 1$ possible rotations.

Every node can rotate with its parent, only the root does not have a parent, therefore there are $n - 1$ possible rotations.

13.2-3

> Let a, b, and c be arbitrary nodes in subtrees α, β, and γ, respectively, in the left tree of Figure 13.2. How do the depths of a, b, and c change when a left rotation is performed on node x in the figure?

- a: increase by 1.
- b: unchanged.
- c: decrease by 1.

13.2-4

> Show that any arbitrary n-node binary search tree can be transformed into any other arbitrary n-node binary search tree using $O(n)$ rotations. ($Hint$: First show that at most $n - 1$ right rotations suffice to transform the tree into a right-going chain.)

Consider transforming an arbitrary n-node binary tree into a right-going chain as follows:

Let the root and all successive right children of the root be the elements of the chain initial chain. For any node x which is a left child of a node on the chain, a single right rotation on the parent of x will add that node to the chain and not remove any elements from the chain. Thus, we can convert any binary search tree to a right chain with at most $n - 1$ right rotations.

Let $r_1, r_2, \dots, r_k$ be the sequence of rotations required to convert some binary search tree T_1 into a right-going chain, and let $s_1, s_2, \dots, s_m$ be the sequence of rotations required to convert some other binary search tree T_2 to a right-going chain. Then $k < n$ and $m < n$, and we can convert T_1 to T_2 by performing the sequence $r_1, r_2, \dots, r_k, s'_m, s'_{m-1}, \dots, s'_1$ where s'_i is the opposite rotation of s_i. Since $k + m < 2n$, the number of rotations required is $O(n)$.

13.2-5 ⋆

> We say that a binary search tree T_1 can be ***right-converted*** to binary search tree T_2 if it is possible to obtain T_2 from T_1 via a series of calls to RIGHT-ROTATE. Give an example of two trees T_1 and T_2 such that T_1 cannot be right-converted to T_2. Then, show that if a tree T_1 can be right-converted to T_2, it can be right-converted using $O(n^2)$ calls to RIGHT-ROTATE.

We can use $O(n)$ calls to rotate the node which is the root in T_2 to T_1's root, then use the same operation in the two subtrees. There are n nodes, therefore the upper bound is $O(n^2)$.

13.3 Insertion

13.3-1

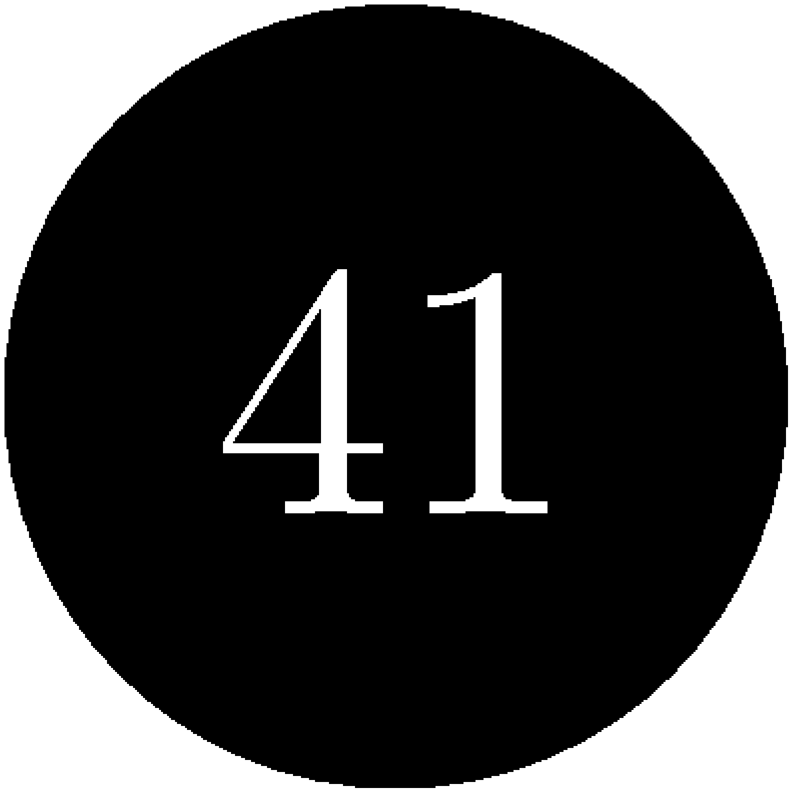
> In line 16 of RB-INSERT, we set the color of the newly inserted node z to red. Observe that if we had chosen to set z's color to black, then property 4 of a red-black tree would not be violated. Why didn't we choose to set z's color to black?

If we chose to set the color of z to black then we would be violating property 5 of being a red-black tree. Because any path from the root to a leaf under z would have one more black node than the paths to the other leaves.

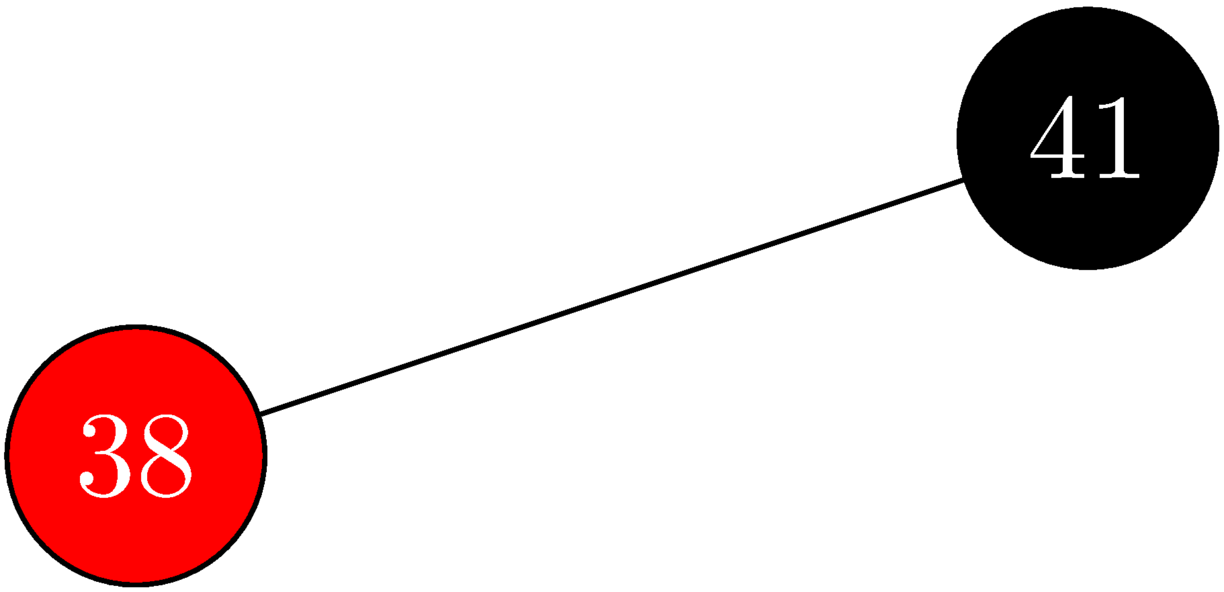
13.3-2

> Show the red-black trees that result after successively inserting the keys $41, 38, 31, 12, 19, 8$ into an initially empty red-black tree.

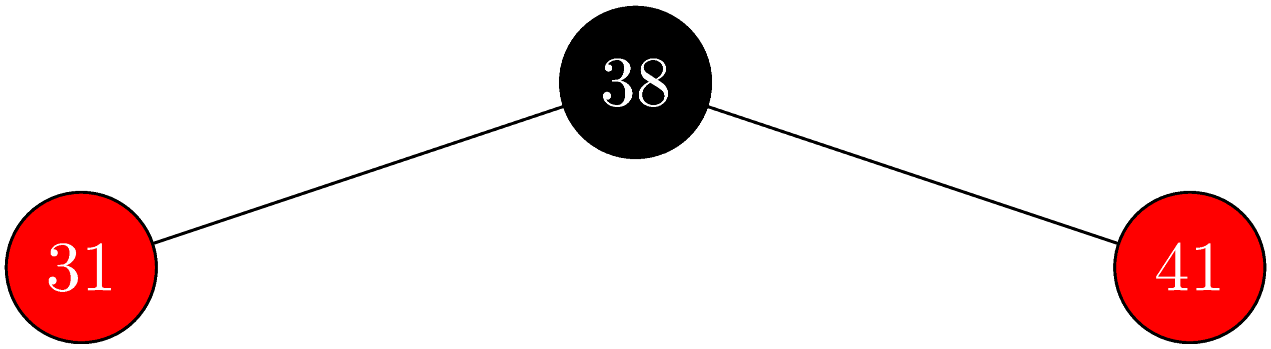
- insert 41:



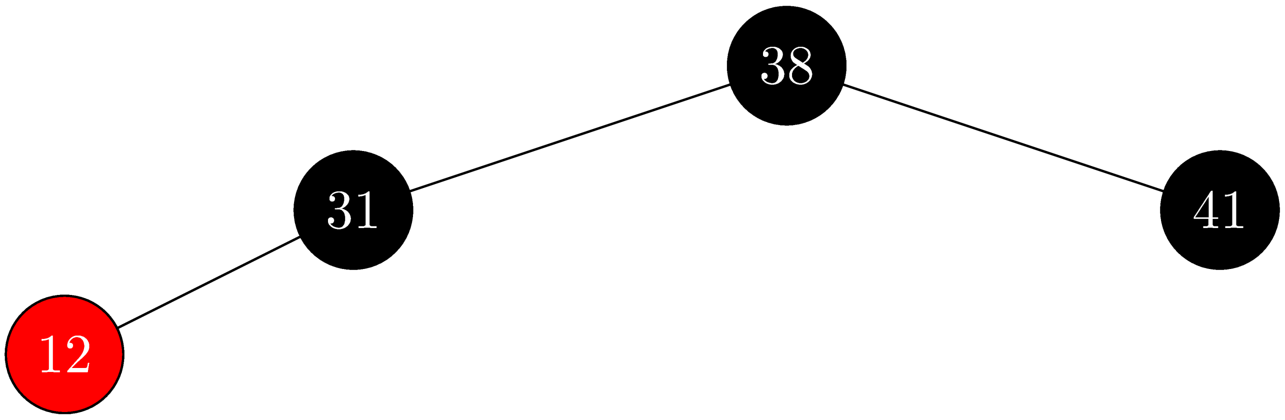
- insert 38:



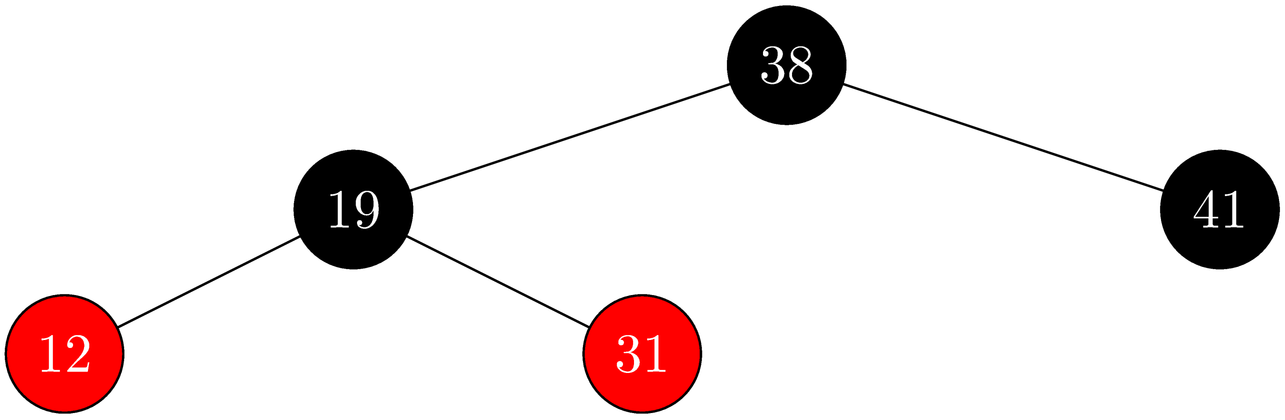
- insert 31:



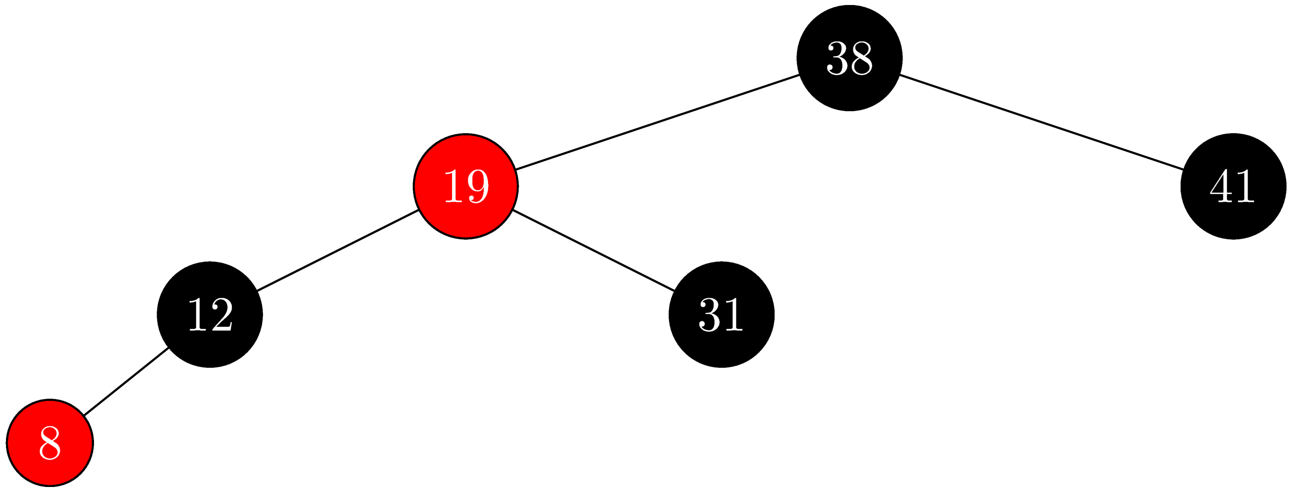
- insert 12:



- insert 19:



- insert 8:



13.3-3

> Suppose that the black-height of each of the subtrees $\alpha, \beta, \gamma, \delta, \epsilon$ in Figures 13.5 and 13.6 is k. Label each node in each figure with its black-height to verify that the indicated transformation preserves property 5.

(Removed)

13.3-4

> Professor Teach is concerned that RB-INSERT-FIXUP might set $T.nil.color$ to RED, in which case the test in line 1 would not cause the loop to terminate when z is the root. Show that the professor's concern is unfounded by arguing that RB-INSERT-FIXUP never sets $T.nil.color$ to RED.

First observe that RB-INSERT-FIXUP only modifies the child of a node if it is already RED, so we will never modify a child which is set to $T.nil$. We just need to check that the parent of the root is never set to RED.

Since the root and the parent of the root are automatically black, if z is at depth less than 2, the **while** loop will be broken. We only modify colors of nodes at most two levels above z, so the only case we need to worry about is when z is at depth 2. In this case we risk modifying the root to be RED, but this is handled in line 16. When z is updated, it will be either the root or the child of the root. Either way, the root and the parent of the root are still BLACK, so the **while** condition is violated, making it impossibly to modify $T.nil$ to be RED.

13.3-5

> Consider a red-black tree formed by inserting n nodes with RB-INSERT. Argue that if $n > 1$, the tree has at least one red node.

- **Case 1:** z and $z.p.p$ are RED, if the loop terminates, then z could not be the root, thus z is RED after the fix up.
- **Case 2:** z and $z.p$ are RED, and after the rotation $z.p$ could not be the root, thus $z.p$ is RED after the fix up.
- **Case 3:** z is RED and z could not be the root, thus z is RED after the fix up.

Therefore, there is always at least one red node.

13.3-6

> Suggest how to implement RB-INSERT efficiently if the representation for red-black trees includes no storage for parent pointers.

Use stack to record the path to the inserted node, then parent is the top element in the stack.

- **Case 1:** we pop $z.p$ and $z.p.p$.
- **Case 2:** we pop $z.p$ and $z.p.p$, then push $z.p.p$ and z.
- **Case 3:** we pop $z.p$, $z.p.p$ and $z.p.p.p$, then push $z.p$.

13.4 Deletion

13.4-1

> Argue that after executing RB-DELETE-FIXUP, the root of the tree must be black.

- **Case 1:** transform to 2, 3, 4.
- **Case 2:** if terminates, the root of the subtree (the new x) is set to black.
- **Case 3:** transform to 4.
- **Case 4:** the root (the new x) is set to black.

13.4-2

> Argue that if in RB-DELETE both x and $x.p$ are red, then property 4 is restored by the call to $\text{RB-DELETE-FIXUP}(T, x)$.

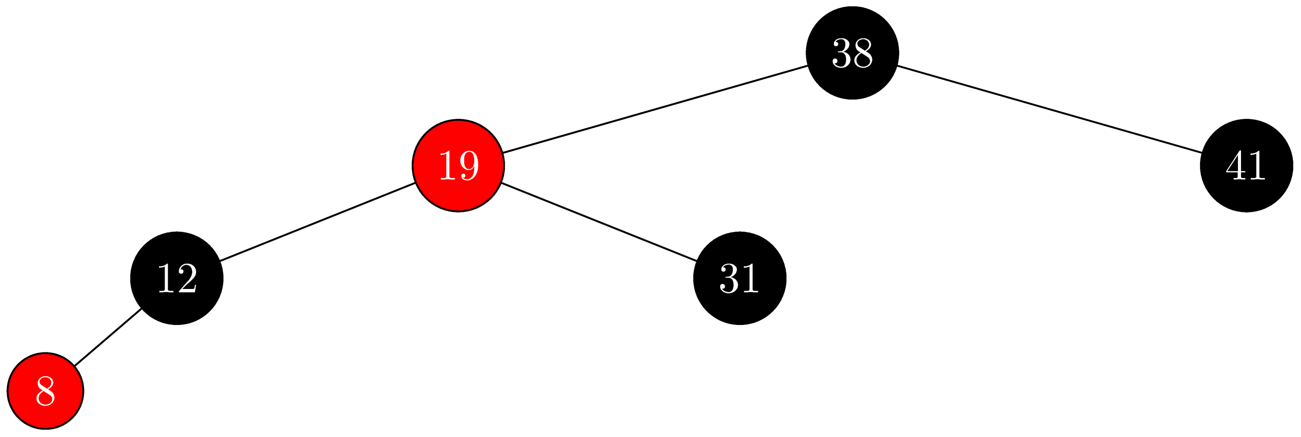
Suppose that both x and $x.p$ are red in RB-DELETE. This can only happen in the else-case of line 9. Since we are deleting from a red-black tree, the other child of y.p which becomes x's sibling in the call to

RB-TRANSPLANT on line 14 must be black, so x is the only child of $x.p$ which is red. The while-loop condition of $\text{RB-DELETE-FIXUP}(T, x)$ is immediately violated so we simply set $x.color = black$, restoring property 4.

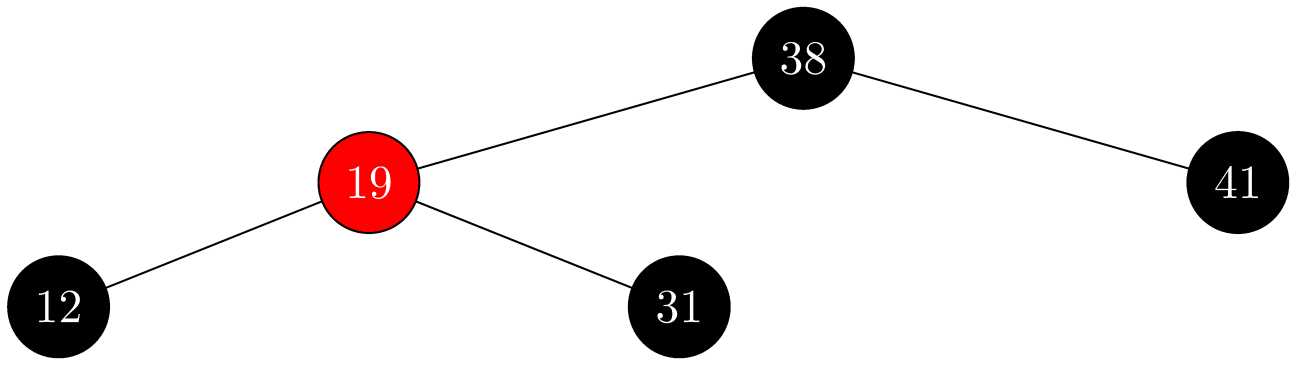
13.4-3

> In Exercise 13.3-2, you found the red-black tree that results from successively inserting the keys $41, 38, 31, 12, 19, 8$ into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order $8, 12, 19, 31, 38, 41$.

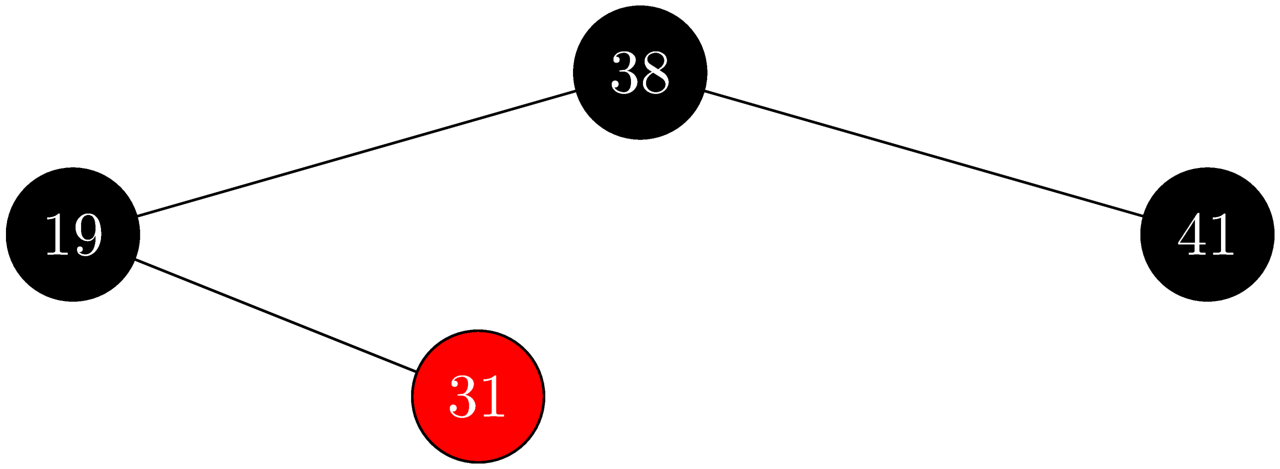
- initial:



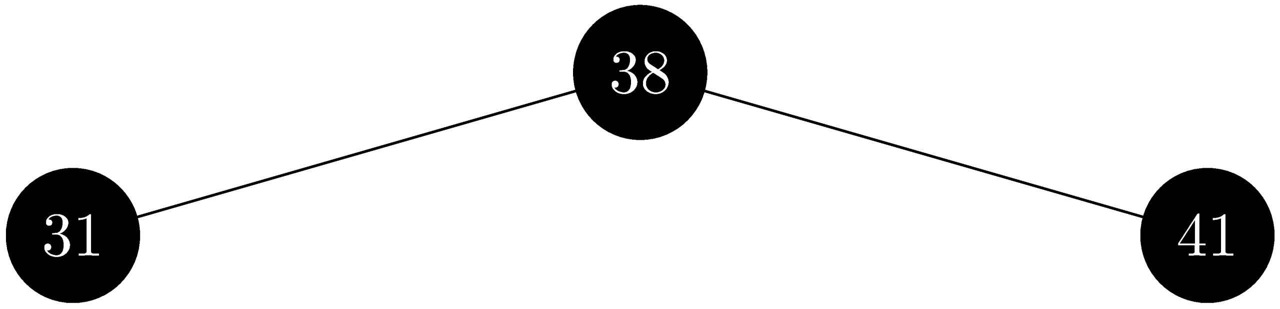
- delete 8:



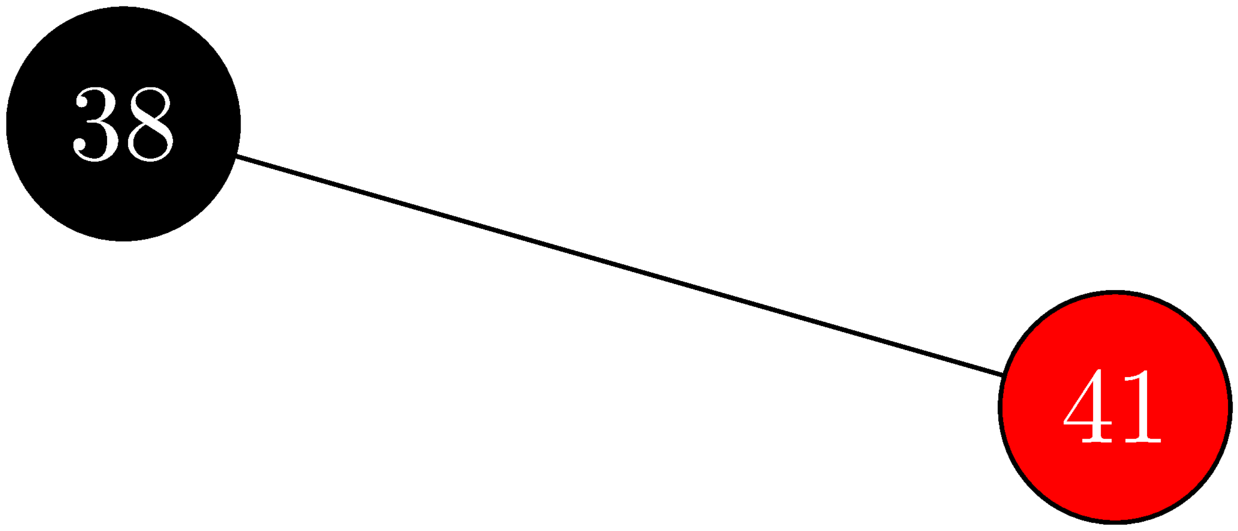
- delete 12:



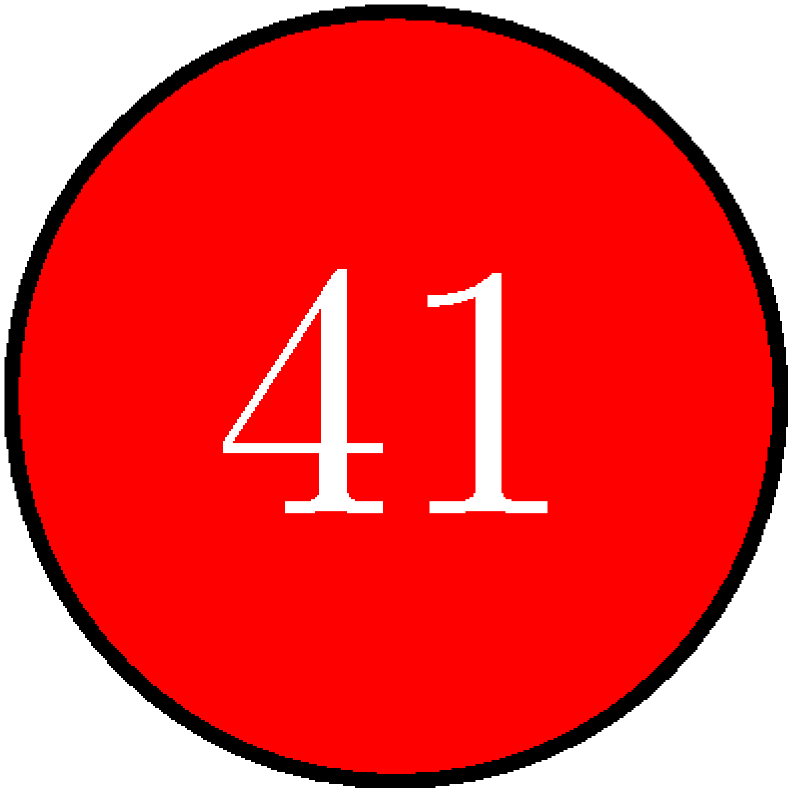
- delete 19:



- delete 31:



- delete 38:



- delete 41:

13.4-4

> In which lines of the code for RB-DELETE-FIXUP might we examine or modify the sentinel $T.nil$?

When the node y in RB-DELETE has no children, the node $x = T.nil$, so we'll examine the line 2 of RB-DELETE-FIXUP.

When the root node is deleted, $x = T.nil$ and the root at this time is x, so the line 23 of RB-DELETE-FIXUP will draw x to black.

13.4-5

> In each of the cases of Figure 13.7, give the count of black nodes from the root of the subtree shown to each of the subtrees $\alpha, \beta, \dots, \zeta$, and verify that each count remains the same after the transformation. When a node has a $color$ attribute c or c', use the notation $\text{count}(c)$ or $\text{count}(c')$ symbolically in your count.

Our count will include the root (if it is black).

- **Case 1:** For each subtree, it is 2 both before and after.
- **Case 2:**
  - For α and β, it is $1 + \text{count}(c)$ in both cases.
  - For the rest of the subtrees, it is from $2 + \text{count}(c)$ to $1 + \text{count}(c)$.

  This decrease in the count for the other subtreese is handled by then having x represent an additional black.
- **Case 3:**
  - For ϵ and ζ, it is $2 + \text{count}(c)$ both before and after.
  - For all the other subtrees, it is $1 + \text{count}(c)$ both before and after.
- **Case 4:**
  - For α and β, it is from $1 + \text{count}(c)$ to $2 + \text{count}(c)$.
  - For γ and δ, it is $1 + \text{count}(c) + \text{count}(c')$ both before and after.
  - For ϵ and ζ, it is $1 + \text{count}(c)$ both before and after.

This increase in the count for α and β is because x before indicated an extra black.

13.4-6

> Professors Skelton and Baron are concerned that at the start of case 1 of RB-DELETE-FIXUP, the node $x.p$ might not be black. If the professors are correct, then lines 5–6 are wrong. Show that $x.p$ must be black at the start of case 1, so that the professors have nothing to worry about.

At the start of case 1 we have set w to be the sibling of x. We check on line 4 that $w.color == red$, which means that the parent of x and w cannot be red. Otherwise property 4 is violated. Thus, their concerns are unfounded.

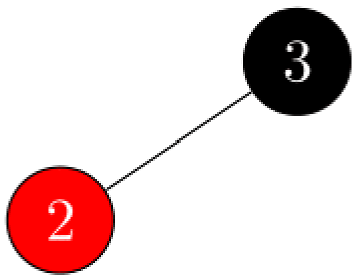
13.4-7

> Suppose that a node x is inserted into a red-black tree with RB-INSERT and then is immediately deleted with RB-DELETE. Is the resulting red-black tree the same as the initial red-black tree? Justify your answer.

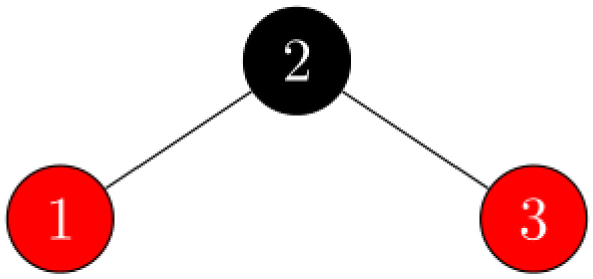
No, the red-black tree will not necessarily be the same.

- Example 1:

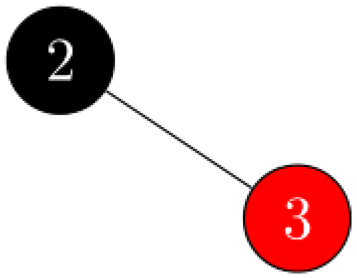
- initial:



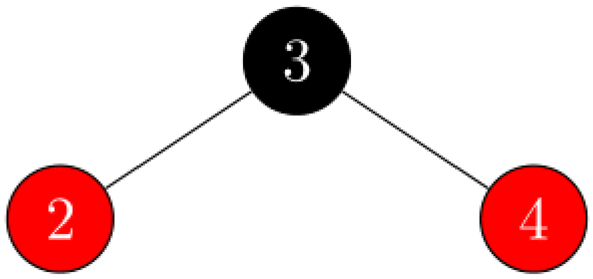
- insert 1:



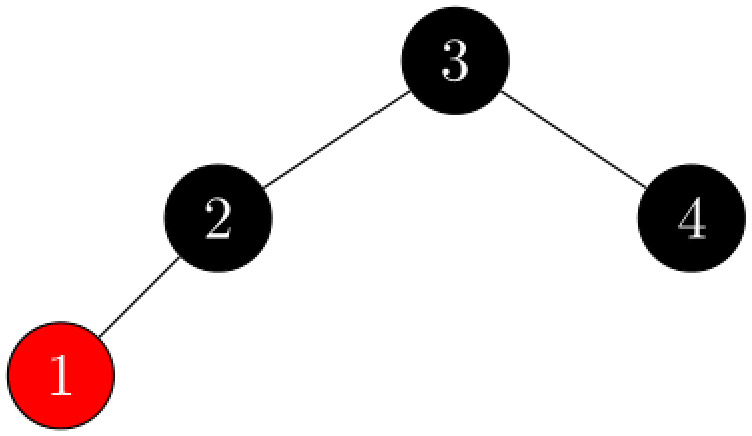
- delete 1:



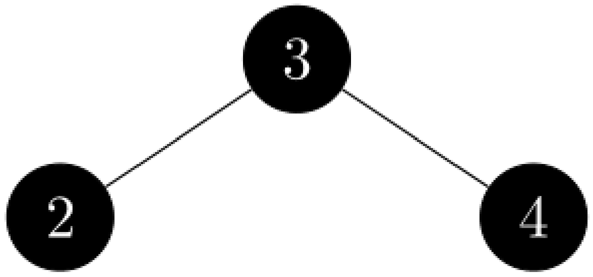
- Example 2:
  - initial:



  - insert 1:



- delete 1:



Problem 13-1 Persistent dynamic sets

> During the course of an algorithm, we sometimes find that we need to maintain past versions of a dynamic set as it is updated. We call such a set ***persistent***. One way to implement a persistent set is to copy the entire set whenever it is modified, but this approach can slow down a program and also consume much space. Sometimes, we can do much better.
>
> Consider a persistent set S with the operations INSERT, DELETE, and SEARCH, which we implement using binary search trees as shown in Figure 13.8(a). We maintain a separate root for every version of the set. In order to insert the key 5 into the set, we create a new node with key 5. This node becomes the left child of a new node with key 7, since we cannot modify the existing node with key 7. Similarly, the new node with key 7 becomes the left child of a new node with key 8 whose right child is the existing node with key 10. The new node with key 8 becomes, in turn, the right child of a new root r' with key 4 whose left child is the existing node with key 3. We thus copy only part of the tree and share some of the nodes with the original tree, as shown in Figure 13.8(b).
>
> Assume that each tree node has the attributes key, $left$, and $right$ but no parent. (See also Exercise 13.3-6.)
>
> **a.** For a general persistent binary search tree, identify the nodes that we need to change to insert a key k or delete a node y.

> **b.** Write a procedure $\text{PERSISTENT-TREE-INSERT}$ that, given a persistent tree T and a key k to insert, returns a new persistent tree T' that is the result of inserting k into T.
>
> **c.** If the height of the persistent binary search tree T is h, what are the time and space requirements of your implementation of $\text{PERSISTENT-TREE-INSERT}$? (The space requirement is proportional to the number of new nodes allocated.)
>
> **d.** Suppose that we had included the parent attribute in each node. In this case, $\text{PERSISTENT-TREE-INSERT}$ would need to perform additional copying. Prove that $\text{PERSISTENT-TREE-INSERT}$ would then require $\Omega(n)$ time and space, where n is the number of nodes in the tree.
>
> **e.** Show how to use red-black trees to guarantee that the worst-case running time and space are $O(\lg n)$ per insertion or deletion.

(Removed)

Problem 13-2 Join operation on red-black trees

> The ***join*** operation takes two dynamic sets S_1 and S_2 and an element x such that for any $x_1 \in S_1$ and $x_2 \in S_2$, we have $x_1.key \le x.key \le x_2.key$. It returns a set $S = S_1 \cup \{x\} \cup S_2$. In this problem, we investigate how to implement the join operation on red-black trees.
>
> **a.** Given a red-black tree T, let us store its black-height as the new attribute $T.bh$. Argue that RB-INSERT and RB-DELETE can maintain the bh attribute without requiring extra storage in the nodes of the tree and without increasing the asymptotic running times. Show that while descending through T, we can determine the black-height of each node we visit in $O(1)$ time per node visited.
>
> We wish to implement the operation $\text{RB-JOIN}(T_1, x, T_2)$, which destroys T_1 and T_2 and returns a red-black tree $T = T_1 \cup \{x\} \cup T_2$. Let n be the total number of nodes in T_1 and T_2.
>
> **b.** Assume that $T_1.bh \ge T_2.bh$. Describe an $O(\lg n)$-time algorithm that finds a black node y in T_1 with the largest key from among those nodes whose black-height is $T_2.bh$.
>
> **c.** Let T_y be the subtree rooted at y. Describe how $T_y \cup \{x\} \cup T_2$ can replace T_y in $O(1)$ time without destroying the binary-search-tree property.
>
> **d.** What color should we make x so that red-black properties 1, 3, and 5 are maintained? Describe how to enforce properties 2 and 4 in $O(\lg n)$ time.
>
> **e.** Argue that no generality is lost by making the assumption in part (b). Describe the symmetric situation that arises when $T_1.bh \le T_2.bh$.
>
> **f.** Argue that the running time of RB-JOIN is $O(\lg n)$.

a.

- Initialize: $\mathrm{bh} = 0$.
- $\mathrm{RB\text{-}INSERT}$: if in the last step the root is red, we increase bh by 1.
- $\mathrm{RB\text{-}DELETE}$: if x is root, we decrease bh by 1.
- Each node: in the simple path, decrease bh by 1 each time we find a black node.

b. Move to the right child if the node has a right child, otherwise move to the left child. If the node is black, we decease bh by 1. Repeat the step until $\mathrm{bh} = T_2.\mathrm{bh}$.

c. The time complexity is $O(1)$.

```
RB-JOIN'(T[y], x, T[2])
    TRANSPLANT(T[y], x)
    x.left = T[y]
    x.right = T[2]
    T[y].parent = x
    T[2].parent = x
```

d. Red. Call $\mathrm{INSERT\text{-}FIXUP}(T[1], x)$.

The time complexity is $O(\lg n)$.

e. Same, if $T_1.\mathrm{bh} \le T_2.\mathrm{bh}$, then we can use the above algorithm symmetrically.

f. $O(1) + O(\lg n) = O(\lg n)$.

Problem 13-3 AVL trees

> An ***AVL tree*** is a binary search tree that is ***height balanced***: for each node x, the heights of the left and right subtrees of x differ by at most 1. To implement an AVL tree, we maintain an extra attribute in each node: $x.h$ is the height of node x. As for any other binary search tree T, we assume that $T.\mathrm{root}$ points to the root node.
>
> **a.** Prove that an AVL tree with n nodes has height $O(\lg n)$. (Hint: Prove that an AVL tree of height h has at least F_h nodes, where F_h is the hth Fibonacci number.)
>
> **b.** To insert into an AVL tree, we first place a node into the appropriate place in binary search tree order. Afterward, the tree might no longer be height balanced. Specifically, the heights of the left and right children of some node might differ by 2. Describe a procedure $\mathrm{BALANCE}(x)$, which takes a subtree rooted at x whose left and right children are height balanced and have heights that differ by at most 2, i.e., $|x.\mathrm{right}.h - x.\mathrm{left}.h| \le 2$, and alters the subtree rooted at x to be height balanced. (Hint: Use rotations.)
>
> **c.** Using part (b), describe a recursive procedure $\mathrm{AVL\text{-}INSERT}(x, z)$ that takes a node x within an AVL tree and a newly created node z (whose key has already been filled in), and adds z to the subtree rooted at x, maintaining the property that x is the root of an AVL tree. As in $\mathrm{TREE\text{-}INSERT}$ from

> Section 12.3, assume that $z.key$ has already been filled in and that $z.left = \text{NIL}$ and $z.right = \text{NIL}$; also assume that $z.h = 0$. Thus, to insert the node z into the AVL tree T, we call $\text{AVL-INSERT}(T.root, z)$.
>
> **d.** Show that AVL-INSERT, run on an n-node AVL tree, takes $O(\lg n)$ time and performs $O(1)$ rotations.

a. Let $T(h)$ denote the minimum size of an AVL tree of height h. Since it is height h, it must have the max of it's children's heights is equal to $h - 1$. Since we are trying to get as few nodes total as possible, suppose that the other child has as small of a height as is allowed. Because of the restriction of AVL trees, we have that the smaller child must be at least one less than the larger one, so, we have that

$$T(h) \ge T(h-1) + T(h-2) + 1,$$

where the $+1$ is coming from counting the root node.

We can get inequality in the opposite direction by simply taking a tree that achieves the minimum number of number of nodes on height $h - 1$ and on $h - 2$ and join them together under another node.

So, we have that

$$T(h) = T(h-1) + T(h-2) + 1, \text{ where } T(0) = 0, T(1) = 1.$$

This is both the same recurrence and initial conditions as the Fibonacci numbers. So, recalling equation (3.25), we have that

$$T(h) = \left\lfloor \frac{\phi^h}{\sqrt{5}} + \frac{1}{2} \right\rfloor \le n.$$

Rearranging for h, we have

$$\begin{aligned} \frac{\phi^h}{\sqrt{5}} - \frac{1}{2} &\le n \\ \phi^h &\le \sqrt{5}(n + \frac{1}{2}) \\ h &\le \frac{\lg \sqrt{5} + \lg(n + \frac{1}{2})}{\lg \phi} \in O(\lg n). \end{aligned}$$

b. Let $\text{UNBAL}(x)$ denote $x.left.h - x.right.h$. Then, the algorithm BALANCE does what is desired. Note that because we are only rotating a single element at a time, the value of $\text{UNBAL}(x)$ can only change by at most 2 in each step.

Also, it must eventually start to change as the tree that was shorter becomes saturated with elements. We also fix any breaking of the AVL property that rotating may of caused by our recursive calls to the children.

```
BALANCE(x)
    while |UNBAL(x)| > 1
```

```
        if UNBAL(x) > 0
            RIGHT-ROTATE(T, x)
        else
            LEFT-ROTATE(T, x)
            BALANCE(x.left)
            BALANCE(x.right)
```

c. For the given algorithm $\text{AVL-INSERT}(x, z)$, it correctly maintains the fact that it is a BST by the way we search for the correct spot to insert z. Also, we can see that it maintains the property of being AVL, because after inserting the element, it checks all of the parents for the AVL property, since those are the only places it could of broken. It then fixes it and also updates the height attribute for any of the nodes for which it may of changed.

d. Both **for** loops only run for $O(h) = O(\lg(n))$ iterations. Also, only a single rotation will occur in the second while loop because when we do it, we will be decreasing the height of the subtree rooted there, which means that it's back down to what it was before, so all of it's ancestors will have unchanged heights, so, no further balancing will be required.

```
AVL-INSERT(x, z)
    w = x
    while w != NIL
        y = w
        if z.key > y.key
            w = w.right
        else w = w.left
    if z.key > y.key
        y.right = z
            if y.left = NIL
                y.h = 1
    else
        y.left = z
        if y.right = NIL
            y.h = 1
    while y != x
        y.h = 1 + max(y.left.h, y.right.h)
        if y.left.h > y.right.h + 1
            RIGHT-ROTATE(T, y)
        if y.right.h > y.left.h + 1
            LEFT-ROTATE(T, y)
            y = y.p
```

Problem 13-4 Treaps

If we insert a set of n items into a binary search tree, the resulting tree may be horribly unbalanced, leading to long search times. As we saw in Section 12.4, however, randomly built binary search trees tend to be balanced. Therefore, one strategy that, on average, builds a balanced tree for a fixed set of items would be to randomly permute the items and then insert them in that order into the tree.

What if we do not have all the items at once? If we receive the items one at a time, can we still randomly build a binary search tree out of them?

We will examine a data structure that answers this question in the affirmative. A ***treap*** is a binary search tree with a modified way of ordering the nodes. Figure 13.9 shows an example. As usual, each node x in the tree has a key value $x.\text{key}$. In addition, we assign $x.\text{priority}$, which is a random number chosen independently for each node. We assume that all priorities are distinct and also that all keys are distinct. The nodes of the treap are ordered so that the keys obey the binary-search-tree property and the priorities obey the min-heap order property:

- If v is a left child of u, then $v.\text{key} < u.\text{key}$.
- If v is a right child of u, then $v.\text{key} > u.\text{key}$.
- If v is a child of u, then $v.\text{priority} > u.\text{priority}$.

(This combination of properties is why the tree is called a "treap": it has features of both a binary search tree and a heap.)

It helps to think of treaps in the following way. Suppose that we insert nodes $x_1, x_2, \ldots, x_n$, with associated keys, into a treap. Then the resulting treap is the tree that would have been formed if the nodes had been inserted into a normal binary search tree in the order given by their (randomly chosen) priorities, i.e., $x_i.\text{priority} < x_j.\text{priority}$ means that we had inserted x_i before x_j.

a. Show that given a set of nodes $x_1, x_2, \ldots, x_n$, with associated keys and priorities, all distinct, the treap associated with these nodes is unique.

b. Show that the expected height of a treap is $\Theta(\lg n)$, and hence the expected time to search for a value in the treap is $\Theta(\lg n)$.

Let us see how to insert a new node into an existing treap. The first thing we do is assign to the new node a random priority. Then we call the insertion algorithm, which we call TREAP-INSERT, whose operation is illustrated in Figure 13.10.

c. Explain how TREAP-INSERT works. Explain the idea in English and give pseudocode. (Hint: Execute the usual binary-search-tree insertion procedure and then perform rotations to restore the min-heap order property.)

d. Show that the expected running time of TREAP-INSERT is $\Theta(\lg n)$.

TREAP-INSERT performs a search and then a sequence of rotations. Although these two operations have the same expected running time, they have different costs in practice. A search reads information from the treap without modifying it. In contrast, a rotation changes parent and child pointers within the treap. On most computers, read operations are much faster than write operations. Thus we would like

> TREAP-INSERT to perform few rotations. We will show that the expected number of rotations performed is bounded by a constant.
>
> In order to do so, we will need some definitions, which Figure 13.11 depicts. The ***left spine*** of a binary search tree T is the simple path from the root to the node with the smallest key. In other words, the left spine is the simple path from the root that consists of only left edges. Symmetrically, the ***right spine*** of T is the simple path from the root consisting of only right edges. The ***length*** of a spine is the number of nodes it contains.
>
> **e.** Consider the treap T immediately after TREAP-INSERT has inserted node x. Let C be the length of the right spine of the left subtree of x. Let D be the length of the left spine of the right subtree of x. Prove that the total number of rotations that were performed during the insertion of x is equal to $C + D$.
>
> We will now calculate the expected values of C and D. Without loss of generality, we assume that the keys are $1, 2, \dots, n$ since we are comparing them only to one another.
>
> For nodes x and y in treap T, where $y \ne x$, let $k = x.\text{key}$ and $i = y.\text{key}$. We define indicator random variables
>
> $$X_{ik} = \text{I}\{y \text{ is in the right spine of the left subtree of } x\}.$$
>
> **f.** Show that $X_{ik} = 1$ if and only if $y.\text{priority} > x.\text{priority}$, $y.\text{key} < x.\text{key}$, and, for every z such that $y.\text{key} < z.\text{key} < x.\text{key}$, we have $y.\text{priority} < z.\text{priority}$.
>
> **g.** Show that
>
> $$\begin{aligned}\Pr\{X_{ik} = 1\} &= \frac{(k-i-1)!}{(k-i+1)!} \\ &= \frac{1}{(k-i+1)(k-i)}.\end{aligned}$$
>
> **h.** Show that
>
> $$\begin{aligned}\text{E}[C] &= \sum_{j=1}^{k-1} \frac{1}{j(j+1)} \\ &= 1 - \frac{1}{k}.\end{aligned}$$
>
> **i.** Use a symmetry argument to show that
>
> $$\text{E}[D] = 1 - \frac{1}{n-k+1}.$$
>
> **j.** Conclude that the expected number of rotations performed when inserting a node into a treap is less than 2.

a. The root is the node with smallest priority, the root divides the sets into two subsets based on the key. In each subset, the node with smallest priority is selected as the root, thus we can uniquely determine a treap with a specific input.

b. For the priority of all nodes, each permutation corresponds to exactly one treap, that is, all nodes forms a BST in priority, since the priority of all nodes is spontaneous, treap is, essentially, randomly built binary search tress. Therefore, the expected height of a treap is $\Theta(\lg n)$.

c. First insert a node as usual using the binary-search-tree insertion procedure. Then perform left and right rotations until the parent of the inserted node no longer has larger priority.

d. Rotation is $\Theta(1)$, at most h rotations, therefore the expected running time is $\Theta(\lg n)$.

e. Left rotation increase C by 1, right rotation increase D by 1.

f. The first two are obvious.

The min-heap property will not hold if $y.\text{priority} > z.\text{priority}$.

g.

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

h.

$$\begin{aligned} E[C] &= \sum_{j=1}^{k-1} \frac{1}{(k-i+1)(k-i)} \\ &= \sum_{j=1}^{k-1} (\frac{1}{k-i} - \frac{1}{k-i+1}) \\ &= 1 - \frac{1}{k}. \end{aligned}$$

i.

$$\begin{aligned} E[D] &= \sum_{j=1}^{n-k} \frac{1}{(k-i+1)(k-i)} \\ &= 1 - \frac{1}{n-k+1}. \end{aligned}$$

j. By part (e), the number of rotations is $C + D$. By linearity of expectation, $E[C + D] = 2 - \frac{1}{k} - \frac{1}{n-k+1} \le 2$ for any choice of k.