1 Binary search trees

A binary search tree is a data structure composed of nodes. Each node has a key, which determines the node's position in the tree. (The node may also have a "value" field, where additional data is stored.)

The top of the tree is the "root," and the nodes contain pointers to other nodes. Specifically, each node has a left child, a right child, and a parent (some of which may be NIL). In Figure 12.1(b), the left child of 7 is 6 and the left child of 5 is NIL. Also, the parent of 5 is 2, and since 2 is the root of the tree, the parent of 2 is NIL.

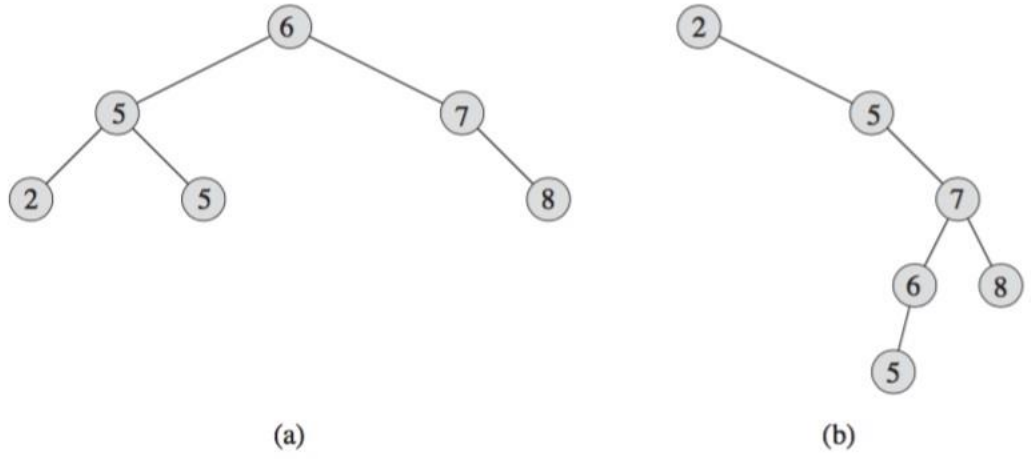
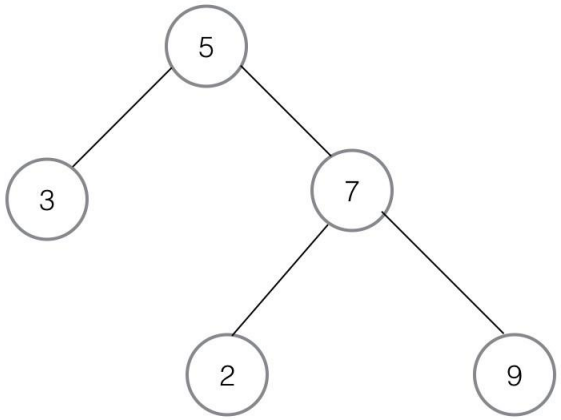


Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

All nodes in a binary search tree must satisfy the binary search tree property:

Binary-search-tree property: Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.

This means, for example, that the following tree is **not** a binary search tree:



Even though 2 ≤ 7 ≤ 9 and 3 ≤ 5 ≤ 7, this tree does not satisfy the binary search tree property, because 2 is in the right subtree of 5, despite being smaller than 5.

1.0.1 Runtime

Binary search trees support several operations, including Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete. These operations run in time proportional to the height of the tree. In the best case scenario, the tree is a complete binary tree, and the height of the tree is $\Theta(\log n)$. In the worst case scenario, the tree is a linear chain, so the height of the tree is $\Theta(n)$.

Later we will talk about red black trees, which are a type of **balanced binary search tree**. In red black trees, the height of the tree is guaranteed to be logarithmic, and we modify the Insert and Delete operations so the logarithmic height is maintained at all times.

1.0.2 Inorder tree walk

Example: Given a binary search tree, write a program that prints the keys in the binary search tree in sorted order.

Answer: To run the program, execute Inorder-Tree-Walk(T.root).

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

Runtime: This program runs in $\Theta(n)$ time. InorderTreeWalk visits all n nodes of the subtree, so $T(n) = \Omega(n)$. To prove $T(n) = O(n)$, we use the substitution method.

InorderTreeWalk takes constant time on an empty subtree, so $T(0) = c$.

Suppose that InorderTreeWalk is called on a node x whose left subtree has k nodes and whose right subtree has $n-k-1$ nodes. Then the time to perform InorderTreeWalk(x) is bounded by $T(n) \le T(k) + T(n-k-1) + d$.

Our "guess" will be $T(n) \le (c+d)n + c$. (We make the guess a bit more complicated than normal to make the math work out.) Plugging in the base case, $n = 0$, we get $T(0) \le c$, as required. For $n > 0$, we have

$$\begin{aligned} T(n) &\le T(k) + T(n-k-1) + d \\ &= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\ &= (c+d)n + c - (c+d) + c + d \\ &= (c+d)n + c \end{aligned}$$

which completes the proof that $T(n) = O(n)$.

1.1 Querying a binary search tree

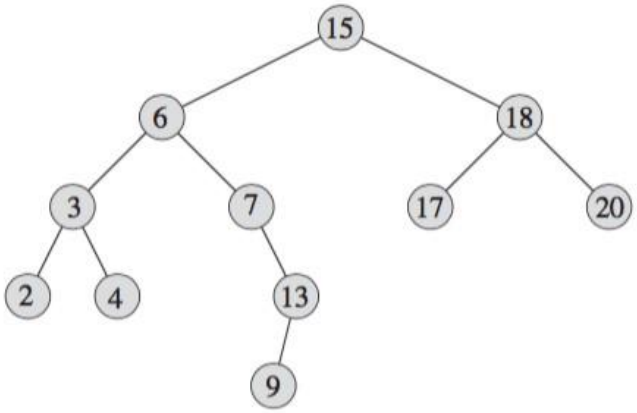


Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path $15 \to 6 \to 7 \to 13$ from the root. The minimum key in the tree is 2, which is found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

1.1.1 Search

The Search routine searches for a node in the tree with a given key. First it compares the node to the root. If the keys are the same, it just returns that node. If the node's key is smaller, then the result (if it exists) will be in the left subtree, by the binary search tree property. If the node's key is larger, then the result (if it exists) will be in the right subtree.

Eventually, we either find the node, or we reach NIL, which tells us that the node can't be found in the tree.

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2      if k < x.key
3          x = x.left
4      else x = x.right
5  return x
```

Exercise: Trace the execution of the program when we search for the key 13 in Figure 12.2.

The Search routine can also be written iteratively, if we "unroll" the recursion and replace it with a while loop. Here we are literally moving down the tree, instead of calling Search on smaller and smaller subproblems.

The Search routine runs in time $O(h)$, where h is the height of the tree, because the loop is executed at most $O(h)$ times.

1.1.2 Minimum/maximum

To find the minimum element in a tree, we start at the root, and take its left child. Then we look at the left child of that left child, etc. We keep going until we reach NIL, and then we return the last non-NIL node in the sequence.

This pseudocode generalizes this procedure to find the minimum element in a subtree rooted at a given non-NIL node x.

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

This is correct because on any iteration of the loop:

1. If x has no left subtree, then x has the smallest key in the subtree rooted at x.
2. If x has a left subtree, then the smallest key in the subtree rooted at x must be in the left subtree of x, by the binary search tree property.

The Minimum and Maximum functions also run in $O(h)$ time.

1.1.3 Successor/predecessor

The successor of a node is the smallest node greater than that node (when ordered by key). To find the successor s of x, we look at the InorderTreeWalk routine, which prints s right after printing x.

From the code you can see there are two cases.

x has a right subtree: If x has a right subtree, we call InorderTreeWalk on the right child immediately after printing x. The first element it will print is the minimum element in the right subtree of x, so we just return TreeMinimum(x.right).

x has no right subtree: If x has no right subtree, we exit the recursive call and go back to the parent node's recursive call. If x was the left child of its parent, InorderTreeWalk immediately prints the parent's key, so the parent is the successor. If x was the right child of its parent, then the parent's recursive call is done, and we proceed to the grandparent's recursive call. In fact we proceed all the way up the tree until we find a node that is the left child of its parent, and then we return that parent. (If we don't find any such nodes, then x had no successor.)

This code once again runs in $O(h)$ time, because TreeMinimum runs in $O(h)$ time, and it also takes $O(h)$ time to follow all of the parent pointers.

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2       return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5       x = y
6       y = y.p
7  return y
```

1.2 Insertion

To insert a node into the tree, we start at the root, and progress downwards until we find a blank space to put the node into. When we find an appropriate space, we replace the NIL with the node, and update pointers.

Note that in addition to the node we are considering (x), we also keep track of x's parent node (y) at all times. This is because once x = NIL we must update y's pointers. Ordinarily we would

be able to make a call to x.parent, but in this case, NIL.parent does not point to a unique parent node.

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4       y = x
 5       if z.key < x.key
 6            x = x.left
 7       else x = x.right
 8  z.p = y
 9  if y == NIL
10       T.root = z          // tree T was empty
11  elseif z.key < y.key
12       y.left = z
13  else y.right = z
```

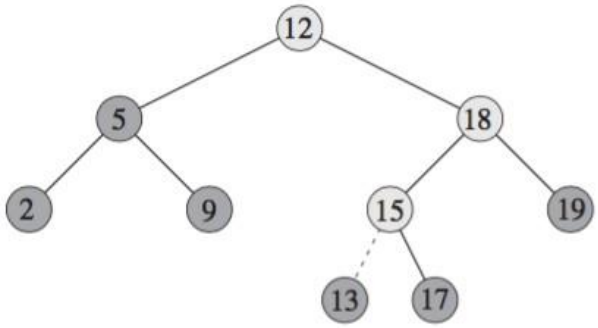


Figure 12.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

1.3 Deletion

To delete a node *z* from the tree, we consider several cases. (Note that in the figure, the successor *s* is labeled *y*.)

1. *z* has no children. In this case, we just remove *z*, by having its parent point to NIL.
2. *z* has one child. Here we switch things so the parent points to *z*'s child, instead of pointing to *z*.

3. z has two children. Here we want to replace z with its successor s (which must be somewhere in the right subtree) because s is smaller than all the other nodes in z's right subtree, so the ordering of nodes would be preserved. However, doing so might mess with the connections of s.

   Fortunately, s cannot have a left child, because then that child would be greater than z (since it's in the right subtree of z) but less than s (since it's in the left subtree of s). This means that if we mess around with s, we only have one child to worry about, and not two. To deal with the child, we consider two cases:

   - If s is the direct right child of z, then we lose nothing by moving s and s's right child "up by one". Here s takes z's place in the tree, and s's left subtree becomes z's left subtree, while s's right subtree stays the same.
   - Otherwise, we pretend we're deleting s (replacing s with s's right child). And then we allow s to take z's place in the tree.

The deletion routine takes $O(h)$ time, because everything takes constant time, except for the part where we find the successor of z.

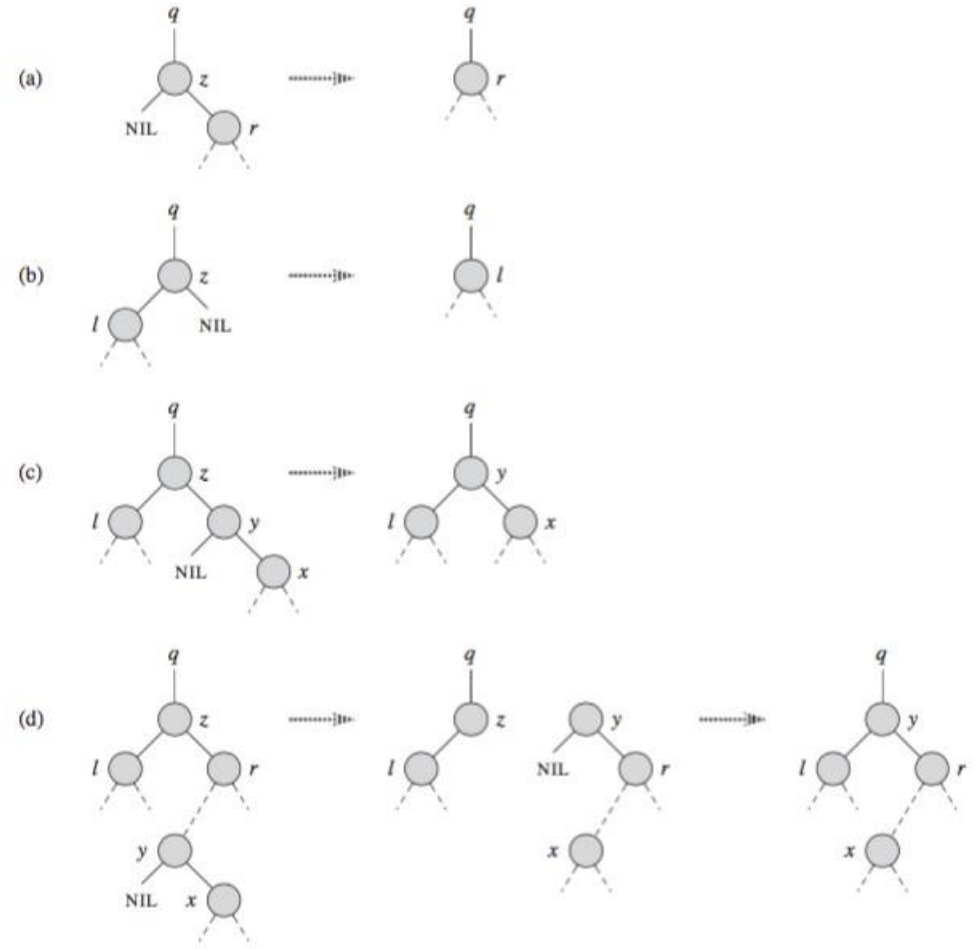


Figure 12.4 Deleting a node z from a binary search tree. Node z may be the root, a left child of node q, or a right child of q. **(a)** Node z has no left child. We replace z by its right child r, which may or may not be NIL. **(b)** Node z has a left child l but no right child. We replace z by l. **(c)** Node z has two children; its left child is node l, its right child is its successor y, and y's right child is node x. We replace z by y, updating y's left child to become l, but leaving x as y's right child. **(d)** Node z has two children (left child l and right child r), and its successor $y \neq r$ lies within the subtree rooted at r. We replace y by its own right child x, and we set y to be r's parent. Then, we set y to be q's child and the parent of l.

To write the pseudocode for this routine, we consider the routine Transplant which replaces the subtree rooted at node *u* with the subtree rooted at node *v*.

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

```
TREE-DELETE(T, z)
1  if z.left == NIL
2      TRANSPLANT(T, z, z.right)
3  elseif z.right == NIL
4      TRANSPLANT(T, z, z.left)
5  else y = TREE-MINIMUM(z.right)
6      if y.p ≠ z
7          TRANSPLANT(T, y, y.right)
8          y.right = z.right
9          y.right.p = y
10     TRANSPLANT(T, z, y)
11     y.left = z.left
12     y.left.p = y
```

2 Red black trees

So far all these operations have run in $O(h)$ time, where h is the height of the tree. This could potentially take $O(n)$ time, if the tree is a linear chain. We attempt to mitigate this by using red black trees. A red-black tree is a balanced binary search tree, which ensures that its height is logarithmic in n, and all of these operations run in $O(\log n)$ time. Red-black trees guarantee that no simple path from the root to a leaf is more than twice as long as any other, which is how the tree becomes balanced.

The query operations (Search, Minimum, Maximum, Successor, and Predecessor) work the same on red-black trees as they do on regular binary search trees. However, the Insert and Delete operations have to be modified to preserve the "red-blackness" of the tree. (These modifications don't add anything to the asymptotic runtime.)

2.1 Red black tree properties

A red-black tree is a binary search tree that satisfies the following properties:

1. Every node has a color, which is either red or black.
2. The root is black.
3. Every leaf is black. (Note that we connect the original leaves of the tree to NIL nodes to avoid boundary condition issues, so we are really just saying all of the NILs are black.)
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the samenumber of black nodes.

The following figure is a red black tree.

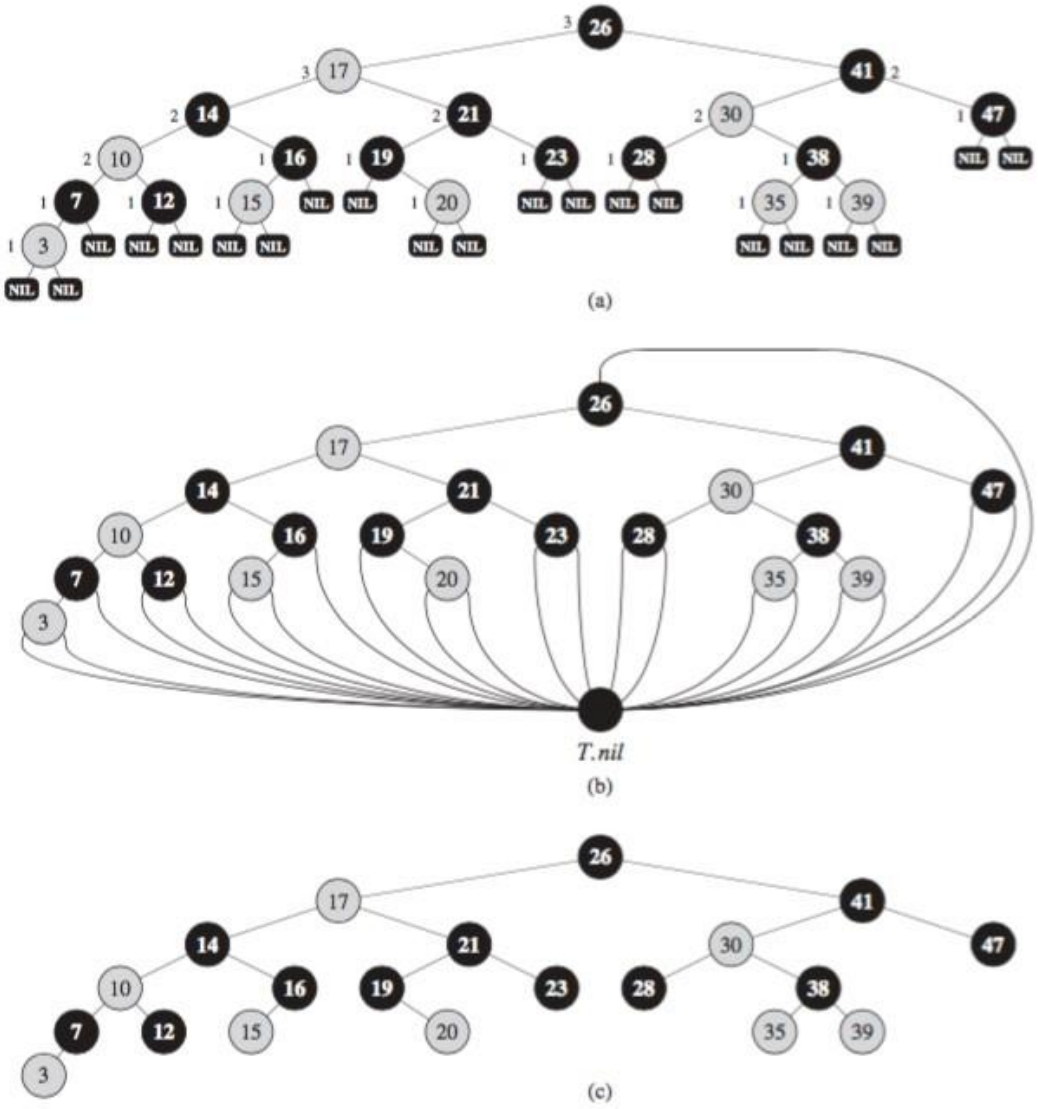
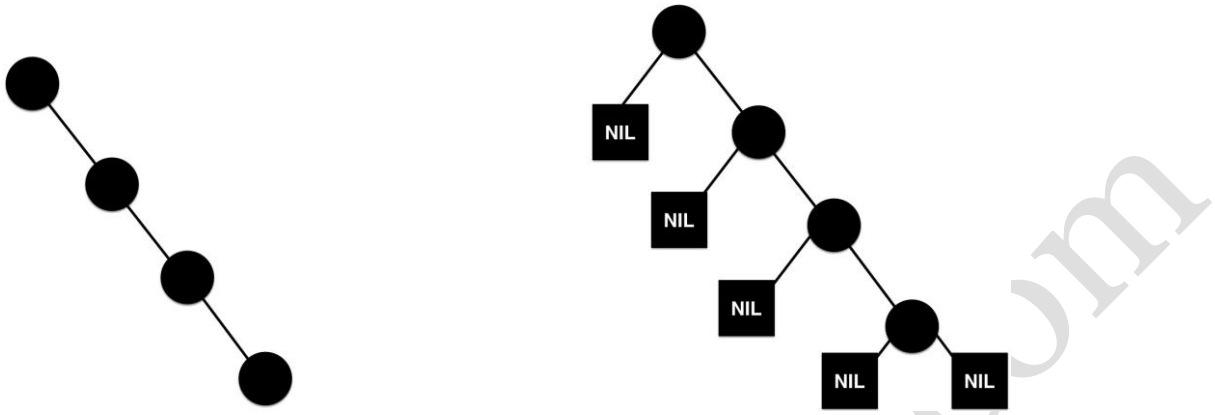


Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NILs have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel *T.nil*, which is always black, and with black-heights omitted. The root's parent is also the sentinel. **(c)** The same red-black tree but with leaves and the root's parent omitted entirely. We shall use this drawing style in the remainder of this chapter.

Note that we will generally avoid drawing NIL nodes (to save space), but we should remember that they exist. It becomes very important when we are trying to count the number of nodes on each path from the root to a leaf. The following figure is **not** a red-black tree. It looks like there is only one path from the root to a leaf, but in reality, the leaves are

the hidden NIL nodes shown on the right, and not all paths from the root to a leaf have the same number of black nodes.



Recall that all simple paths from a node to descendant leaves must contain the same number of black nodes. We call the number of black nodes on any simple path from, but not including, a node x down to a leaf the **black-height** of the node, denoted $bh(x)$. The black-height of the tree is just the black height of the root.

All NILs have black-height 0, and the black-height of the tree in Figure 13.1 is 3.

Because each red node must have all black children, the fact that all paths from the root to a leaf have the same number of black nodes also places a constraint on the total length of each path. Specifically, the lengths of each path have to be within a factor of 2 of each other. This gives rise to the logarithmic bound on the height of the tree:

Lemma: A red-black tree with n internal nodes has height at most $2\log(n + 1)$.

Proof: We claim that the subtree rooted at any node x has at least $2^{bh(x)}-1$ internal nodes. We show this by induction. If the height of x is 0, then x is a leaf (i.e. x is a NIL), and the subtree rooted at x contains at least $2^{bh(x)} - 1 = 0$ internal nodes.

For the inductive step, consider a node x that has positive height, and assume that the statement is true for all nodes with height less than x (strong induction!). Note that x must have two children because we use NILs to "fill the gaps" in the tree. Then each child has a black height of either $bh(x)$ (if x is red) or $bh(x) - 1$ (if x is black). By the inductive hypothesis, each child has at least $2^{bh(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains at least $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes, proving the claim.

Finally, since all red nodes must have two black children, at least half the nodes on any simple path from the root to a leaf (not including the root) must be black. Therefore, $bh(x) \geq h/2$, and $n \geq 2^{h/2} - 1$. Simplifying this, we get $h \leq 2\log(n + 1)$.

2.2 Rotations

These proofs only help if the tree always satisfies the red-black properties. If you insert nodes in the normal way, it will mess up the tree. So we need to find a way to fix the tree up after inserting a node in order to preserve the red-blackness of the red-black tree.

The key is an operation called **rotation**, which preserves the ordering of nodes in the tree while correcting an imbalance in height.

We have left rotations, which rotate a subtree "counterclockwise," and right rotations, which rotate the subtree "clockwise." Here we discuss left rotations on a node x, and in order to do a left rotation we assume that x's right child is not NIL.

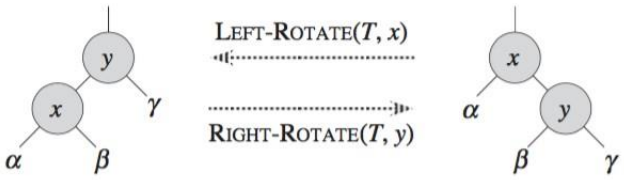


Figure 13.2 The rotation operations on a binary search tree. The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation RIGHT-ROTATE(T, y) transforms the configuration on the left into the configuration on the right. The letters α, β, and γ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in α precede $x.key$, which precedes the keys in β, which precede $y.key$, which precedes the keys in γ.

To understand left rotation, look at the right half of the figure. We know that

1. x is less than y.
2. Anything in x's left subtree α is less than both x and y.
3. Anything in y's left subtree β is greater than x, but less than y.
4. Anything in y's right subtree γ is greater than both x and y.

The left-rotation operation, depicted in the figure, preserves all these properties (and preserves the ordering of all the nodes in the tree). Specifically, since x is now y's left child, the tree still says x is less than y. And since β is now the right subtree of x, but is part of the left subtree of y, the tree still says everything in β is greater than x, and less than y.

The pseudocode for LeftRotate just hooks things up as depicted in the figure:

```
LEFT-ROTATE(T, x)
 1  y = x.right              // set y
 2  x.right = y.left         // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil
 4       y.left.p = x
 5  y.p = x.p                // link x's parent to y
 6  if x.p == T.nil
 7       T.root = y
 8  elseif x == x.p.left
 9       x.p.left = y
10  else x.p.right = y
11  y.left = x               // put x on y's left
12  x.p = y
```