CHAPTER

14 Binary Search Trees

The number of trees which can be formed with $n+1$ given knots $\alpha, \beta, \gamma, \ldots = (n+1)^{n-1}$.

— *"A Theorem on Trees,"* A. CAYLEY, 1889

BSTs are a workhorse of data structures and can be used to solve almost every data structures problem reasonably efficiently. They offer the ability to efficiently search for a key as well as find the *min* and *max* elements, look for the successor or predecessor of a search key (which itself need not be present in the BST), and enumerate the keys in a range in sorted order.

BSTs are similar to arrays in that the stored values (the "keys") are stored in a sorted order. However, unlike with a sorted array, keys can be added to and deleted from a BST efficiently. Specifically, a BST is a binary tree as defined in Chapter 9 in which the nodes store keys that are comparable, e.g., integers or strings. The keys stored at nodes have to respect the BST property—the key stored at a node is greater than or equal to the keys stored at the nodes of its left subtree and less than or equal to the keys stored in the nodes of its right subtree. Figure 14.1 on the following page shows a BST whose keys are the first 16 prime numbers.

Key lookup, insertion, and deletion take time proportional to the height of the tree, which can in worst-case be $O(n)$, if insertions and deletions are naively implemented. However, there are implementations of insert and delete which guarantee that the tree has height $O(\log n)$. These require storing and updating additional data at the tree nodes. Red-black trees are an example of height-balanced BSTs and are widely used in data structure libraries.

A common mistake with BSTs is that an object that's present in a BST is not updated. The consequence is that a lookup for that object returns false, even though it's still in the BST. As a rule, avoid putting mutable objects in a BST. Otherwise, when a mutable object that's in a BST is to be updated, always first remove it from the tree, then update it, then add it back.

The BST prototype is as follows:

```
class BSTNode:
    def __init__(self, data=None, left=None, right=None):
        self.data, self.left, self.right = data, left, right
```

Binary search trees boot camp

Searching is the single most fundamental application of BSTs. Unlike a hash table, a BST offers the ability to find the min and max elements, and find the next largest/next smallest element. These operations, along with lookup, delete and find, take time $O(\log n)$ for library implementations of BSTs. Both BSTs and hash tables use $O(n)$ space—in practice, a BST uses slightly more space.

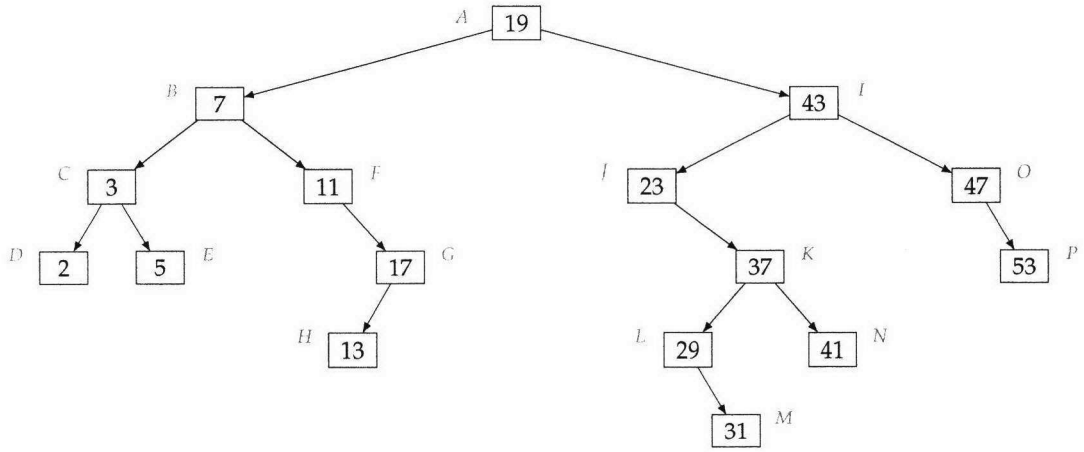

Figure 14.1: An example of a BST.

The following program demonstrates how to check if a given value is present in a BST. It is a nice illustration of the power of recursion when operating on BSTs.

```
def search_bst(tree, key):
    return (tree
            if not tree or tree.data == key else search_bst(tree.left, key)
            if key < tree.data else search_bst(tree.right, key))
```

Since the program descends tree with in each step, and spends $O(1)$ time per level, the time complexity is $O(h)$, where h is the height of the tree.

With a BST you can **iterate** through elements in **sorted order** in time $O(n)$ (regardless of whether it is balanced).
Some problems need a **combination of a BST and a hashtable**. For example, if you insert student objects into a BST and entries are ordered by GPA, and then a student's GPA needs to be updated and all we have is the student's name and new GPA, we cannot find the student by name without a full traversal. However, with an additional hash table, we can directly go to the corresponding entry in the tree.
Sometimes, it's necessary to **augment** a BST to make it possible to manipulate more complicated data, e.g., intervals, and efficiently support more complex queries, e.g., the number of elements in a range (on Page 214).
The BST property is a **global property**—a binary tree may have the property that each node's key is greater than the key at its left child and smaller than the key at its right child, but it may not be a BST.

Table 14.1: Top Tips for Binary Search Trees

Know your binary search tree libraries

Some of the problems in this chapter entail writing a BST class; for others, you can use a BST library. Python does not come with a built-in BST library.

The `sortedcontainers` module the best-in-class module for sorted sets and sorted dictionaries—it is performant, has a clean API that is well documented, with a responsive community. The underlying data structure is a sorted list of sorted lists. Its asymptotic time complexity for inserts and deletes is $O(\sqrt{n})$) since these operations entail insertion into a list of length roughly $\sqrt{n}$, rather than the $O(\log n)$) of balanced BSTs. In practice, this is not an issue, since CPUs are highly optimized for block data movements.

In the interests of pedagogy, we have elected to use the `bintrees` module which implements sorted sets and sorted dictionaries using balanced BSTs. However, any reasonable interviewer should accept `sortedcontainers` wherever we use `bintrees`.

Below, we describe the functionalities added by `bintrees`.

- `insert(e)` inserts new element e in the BST.
- `discard(e)` removes e in the BST if present.
- `min_item()/max_item()` yield the smallest and largest key-value pair in the BST.
- `min_key()/max_key()` yield the smallest and largest key in the BST.
- `pop_min()/pop_max()` remove the return the smallest and largest key-value pair in the BST.

It's particularly important to note that these operations take $O(\log n)$, since they are backed by the underlying tree.

The following program illustrates the use of `bintrees`.

```
t = bintrees.RBTree([(5, 'Alfa'), (2, 'Bravo'), (7, 'Charlie'), (3, 'Delta'),
                     (6, 'Echo')])

print(t[2])  # 'Bravo'

print(t.min_item(), t.max_item())  # (2, 'Bravo'), (7, 'Charlie')

# {2: 'Bravo', 3: 'Delta', 5: 'Alfa', 6: 'Echo', 7: 'Charlie', 9: 'Golf'}
t.insert(9, 'Golf')
print(t)

print(t.min_key(), t.max_key())  # 2, 9

t.discard(3)
print(t)  # {2: 'Bravo', 5: 'Alfa', 6: 'Echo', 7: 'Charlie', 9: 'Golf'}

# a = (2: 'Bravo')
a = t.pop_min()
print(t)  # {5: 'Alfa', 6: 'Echo', 7: 'Charlie', 9: 'Golf'}

# b = (9, 'Golf')
b = t.pop_max()
print(t)  # {5: 'Alfa', 6: 'Echo', 7: 'Charlie'}
```

14.1 Test if a binary tree satisfies the BST property

Write a program that takes as input a binary tree and checks if the tree satisfies the BST property.

Hint: Is it correct to check for each node that its key is greater than or equal to the key at its left child and less than or equal to the key at its right child?

Solution: A direct approach, based on the definition of a BST, is to begin with the root, and compute the maximum key stored in the root's left subtree, and the minimum key in the root's right subtree. We check that the key at the root is greater than or equal to the maximum from the left subtree and less than or equal to the minimum from the right subtree. If both these checks pass, we recursively check the root's left and right subtrees. If either check fails, we return false.

Computing the minimum key in a binary tree is straightforward: we take the minimum of the key stored at its root, the minimum key of the left subtree, and the minimum key of the right subtree. The maximum key is computed similarly. Note that the minimum can be in either subtree, since a general binary tree may not satisfy the BST property.

The problem with this approach is that it will repeatedly traverse subtrees. In the worst-case, when the tree is a BST and each node's left child is empty, the complexity is $O(n^2)$, where n is the number of nodes. The complexity can be improved to $O(n)$ by caching the largest and smallest keys at each node; this requires $O(n)$ additional storage for the cache.

We now present two approaches which have $O(n)$ time complexity.

The first approach is to check constraints on the values for each subtree. The initial constraint comes from the root. Every node in its left (right) subtree must have a key less than or equal (greater than or equal) to the key at the root. This idea generalizes: if all nodes in a tree must have keys in the range $[l, u]$, and the key at the root is w (which itself must be between $[l, u]$, otherwise the requirement is violated at the root itself), then all keys in the left subtree must be in the range $[l, w]$, and all keys stored in the right subtree must be in the range $[w, u]$.

As a concrete example, when applied to the BST in Figure 14.1 on Page 198, the initial range is $[-\infty, \infty]$. For the recursive call on the subtree rooted at B, the constraint is $[-\infty, 19]$; the 19 is the upper bound required by A on its left subtree. For the recursive call starting at the subtree rooted at F, the constraint is $[7, 19]$. For the recursive call starting at the subtree rooted at K, the constraint is $[23, 43]$. The binary tree in Figure 9.1 on Page 112 is identified as not being a BST when the recursive call reaches C—the constraint is $[-\infty, 6]$, but the key at F is 271, so the tree cannot satisfy the BST property.

```
def is_binary_tree_bst(tree, low_range=float('-inf'), high_range=float('inf')):
    if not tree:
        return True
    elif not low_range <= tree.data <= high_range:
        return False
    return (is_binary_tree_bst(tree.left, low_range, tree.data)
            and is_binary_tree_bst(tree.right, tree.data, high_range))
```

The time complexity is $O(n)$, and the additional space complexity is $O(h)$, where h is the height of the tree.

Alternatively, we can use the fact that an inorder traversal visits keys in sorted order. Furthermore, if an inorder traversal of a binary tree visits keys in sorted order, then that binary tree must be a BST. (This follows directly from the definition of a BST and the definition of an inorder walk.) Thus we can check the BST property by performing an inorder traversal, recording the key stored at the last visited node. Each time a new node is visited, its key is compared with the key of the previously visited node. If at any step in the walk, the key at the previously visited node is greater than the node currently being visited, we have a violation of the BST property.

All these approaches explore the left subtree first. Therefore, even if the BST property does not hold at a node which is close to the root (e.g., the key stored at the right child is less than the key

stored at the root), their time complexity is still $O(n)$.

We can search for violations of the BST property in a BFS manner, thereby reducing the time complexity when the property is violated at a node whose depth is small.

Specifically, we use a queue, where each queue entry contains a node, as well as an upper and a lower bound on the keys stored at the subtree rooted at that node. The queue is initialized to the root, with lower bound $-\infty$ and upper bound ∞. We iteratively check the constraint on each node. If it violates the constraint we stop—the BST property has been violated. Otherwise, we add its children along with the corresponding constraint.

For the example in Figure 14.1 on Page 198, we initialize the queue with $(A, [-\infty, \infty])$. Each time we pop a node, we first check the constraint. We pop the first entry, $(A, [-\infty, \infty])$, and add its children, with the corresponding constraints, i.e., $(B, [-\infty, 19])$ and $(I, [19, \infty])$. Next we pop $(B, [-\infty, 19])$, and add its children, i.e., $(C, [-\infty, 7])$ and $(D, [7, 19])$. Continuing through the nodes, we check that all nodes satisfy their constraints, and thus verify the tree is a BST.

If the BST property is violated in a subtree consisting of nodes within a particular depth, the violation will be discovered without visiting any nodes at a greater depth. This is because each time we enqueue an entry, the lower and upper bounds on the node's key are the tightest possible.

```
def is_binary_tree_bst(tree):
    QueueEntry = collections.namedtuple('QueueEntry', ('node', 'lower',
                                                       'upper'))
    bfs_queue = collections.deque(
        [QueueEntry(tree, float('-inf'), float('inf'))])

    while bfs_queue:
        front = bfs_queue.popleft()
        if front.node:
            if not front.lower <= front.node.data <= front.upper:
                return False
            bfs_queue += [
                QueueEntry(front.node.left, front.lower, front.node.data),
                QueueEntry(front.node.right, front.node.data, front.upper)
            ]
    return True
```

The time complexity is $O(n)$, and the additional space complexity is $O(n)$.

14.2 Find the first key greater than a given value in a BST

Write a program that takes as input a BST and a value, and returns the first key that would appear in an inorder traversal which is greater than the input value. For example, when applied to the BST in Figure 14.1 on Page 198 you should return 29 for input 23.

Hint: Perform binary search, keeping some additional state.

Solution: We can find the desired node in $O(n)$ time, where n is the number of nodes in the BST, by doing an inorder walk. This approach does not use the BST property.

A better approach is to use the BST search idiom. We store the best candidate for the result and update that candidate as we iteratively descend the tree, eliminating subtrees by comparing the keys stored at nodes with the input value. Specifically, if the current subtree's root holds a value less than or equal to the input value, we search the right subtree. If the current subtree's root stores

a key that is greater than the input value, we search in the left subtree, updating the candidate to the current root. Correctness follows from the fact that whenever we first set the candidate, the desired result must be within the tree rooted at that node.

For example, when searching for the first node whose key is greater than 23 in the BST in Figure 14.1 on Page 198, the node sequence is A, I, J, K, L. Since L has no left child, its key, 29, is the result.

```
def find_first_greater_than_k(tree, k):
    subtree, first_so_far = tree, None
    while subtree:
        if subtree.data > k:
            first_so_far, subtree = subtree, subtree.left
        else:  # Root and all keys in left subtree are <= k, so skip them.
            subtree = subtree.right
    return first_so_far
```

The time complexity is $O(h)$, where h is the height of the tree. The space complexity is $O(1)$.

Variant: Write a program that takes as input a BST and a value, and returns the node whose key equals the input value and appears first in an inorder traversal of the BST. For example, when applied to the BST in Figure 14.2, your program should return Node B for 108, Node G for 285, and null for 143.

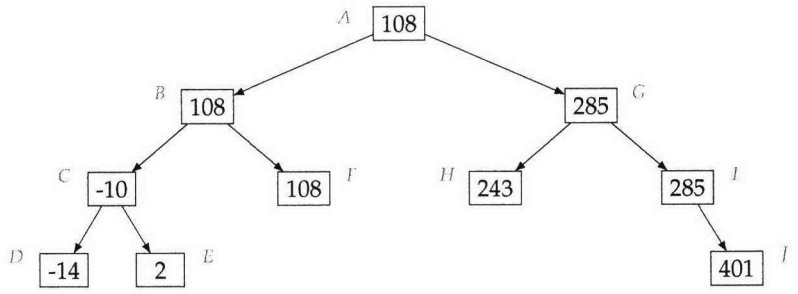

Figure 14.2: A BST with duplicate keys.

14.3 Find the k largest elements in a BST

A BST is a sorted data structure, which suggests that it should be possible to find the k largest keys easily.

Write a program that takes as input a BST and an integer k, and returns the k largest elements in the BST in decreasing order. For example, if the input is the BST in Figure 14.1 on Page 198 and $k = 3$, your program should return $\langle 53, 47, 43 \rangle$.

Hint: What does an inorder traversal yield?

Solution: The brute-force approach is to do an inorder traversal, which enumerates keys in ascending order, and return the last k visited nodes. A queue is ideal for storing visited nodes, since it makes it easy to evict nodes visited more than k steps previously. A drawback of this approach is

that it potentially processes many nodes that cannot possibly be in the result, e.g., if k is small and the left subtree is large.

A better approach is to begin with the desired nodes, and work backwards. We do this by recursing first on the right subtree and then on the left subtree. This amounts to a reverse-inorder traversal. For the BST in Figure 14.1 on Page 198, the reverse inorder visit sequence is $\langle P, O, I, N, K, M, L, J, A, G, H, F, B, E, C, D \rangle$.

As soon as we visit k nodes, we can halt. The code below uses a dynamic array to store the desired keys. As soon as the array has k elements, we return. We store newer nodes at the end of the array, as per the problem specification.

To find the five biggest keys in the tree in Figure 14.1 on Page 198, we would recurse on A, I, O, P, in that order. Returning from recursive calls, we would visit P, O, I, in that order, and add their keys to the result. Then we would recurse on J, K, N, in that order. Finally, we would visit N and then K, adding their keys to the result. Then we would stop, since we have five keys in the array.

```
def find_k_largest_in_bst(tree, k):
    def find_k_largest_in_bst_helper(tree):
        # Perform reverse inorder traversal.
        if tree and len(k_largest_elements) < k:
            find_k_largest_in_bst_helper(tree.right)
            if len(k_largest_elements) < k:
                k_largest_elements.append(tree.data)
                find_k_largest_in_bst_helper(tree.left)

    k_largest_elements = []
    find_k_largest_in_bst_helper(tree)
    return k_largest_elements
```

The time complexity is $O(h + k)$, which can be much better than performing a conventional inorder walk, e.g., when the tree is balanced and k is small. The complexity bound comes from the observation that the number of times the program descends in the tree can be at most h more than the number of times it ascends the tree, and each ascent happens after we visit a node in the result. After k nodes have been added to the result, the program stops.

14.4 Compute the LCA in a BST

Since a BST is a specialized binary tree, the notion of lowest common ancestor, as expressed in Problem 9.4 on Page 118, holds for BSTs too.

In general, computing the LCA of two nodes in a BST is no easier than computing the LCA in a binary tree, since structurally a binary tree can be viewed as a BST where all the keys are equal. However, when the keys are distinct, it is possible to improve on the LCA algorithms for binary trees.

Design an algorithm that takes as input a BST and two nodes, and returns the LCA of the two nodes. For example, for the BST in Figure 14.1 on Page 198, and nodes C and G, your algorithm should return B. Assume all keys are distinct. Nodes do not have references to their parents.

Hint: Take advantage of the BST property.

Solution: In Solution 9.3 on Page 117 we presented an algorithm for this problem in the context of binary trees. The idea underlying that algorithm was to do a postorder traversal—the LCA is the

first node visited after the two nodes whose LCA we are to compute have been visited. The time complexity was $O(n)$, where n is the number of nodes in the tree.

This approach can be improved upon when operating on BSTs with distinct keys. Consider the BST in Figure 14.1 on Page 198 and nodes C and G. Since both C and G hold keys that are smaller than A's key, their LCA must lie in A's left subtree. Examining B, since C's key is less than B's key, and B's key is less than G's key. B must be the LCA of C and G.

Let s and b be the two nodes whose LCA we are to compute, and without loss of generality assume the key at s is smaller. (Since the problem specified keys are distinct, it cannot be that s and b hold equal keys.) Consider the key stored at the root of the BST. There are four possibilities:

- If the root's key is the same as that stored at s or at b, we are done—the root is the LCA.
- If the key at s is smaller than the key at the root, and the key at b is greater than the key at the root, the root is the LCA.
- If the keys at s and b are both smaller than that at the root, the LCA must lie in the left subtree of the root.
- If both keys are larger than that at the root, then the LCA must lie in the right subtree of the root.

```
# Input nodes are nonempty and the key at s is less than or equal to that at b.
def find_LCA(tree, s, b):
    while tree.data < s.data or tree.data > b.data:
        # Keep searching since tree is outside of [s, b].
        while tree.data < s.data:
            tree = tree.right  # LCA must be in tree's right child.
        while tree.data > b.data:
            tree = tree.left  # LCA must be in tree's left child.
    # Now, s.data <= tree.data && tree.data <= b.data.
    return tree
```

Since we descend one level with each iteration, the time complexity is $O(h)$, where h is the height of the tree.

14.5 Reconstruct a BST from traversal data

As discussed in Problem 9.12 on Page 125 there are many different binary trees that yield the same sequence of visited nodes in an inorder traversal. This is also true for preorder and postorder traversals. Given the sequence of nodes that an inorder traversal sequence visits and either of the other two traversal sequences, there exists a unique binary tree that yields those sequences. Here we study if it is possible to reconstruct the tree with less traversal information when the tree is known to be a BST.

It is critical that the elements stored in the tree be unique. If the root contains key v and the tree contains more occurrences of v, we cannot always identify from the sequence whether the subsequent vs are in the left subtree or the right subtree. For example, for the tree rooted at G in Figure 14.2 on Page 202 the preorder traversal sequence is 285, 243, 285, 401. The same preorder traversal sequence is seen if 285 appears in the left subtree as the right child of the node with key 243 and 401 is at the root's right child.

Suppose you are given the sequence in which keys are visited in an inorder traversal of a BST, and all keys are distinct. Can you reconstruct the BST from the sequence? If so, write a program to do so. Solve the same problem for preorder and postorder traversal sequences.

Hint: Draw the five BSTs on the keys 1, 2, 3, and the corresponding traversal orders.

Solution: First, with some experimentation, we see the sequence of keys generated by an inorder traversal is not enough to reconstruct the tree. For example, the key sequence $\langle 1, 2, 3 \rangle$ corresponds to five distinct BSTs as shown in Figure 14.3.

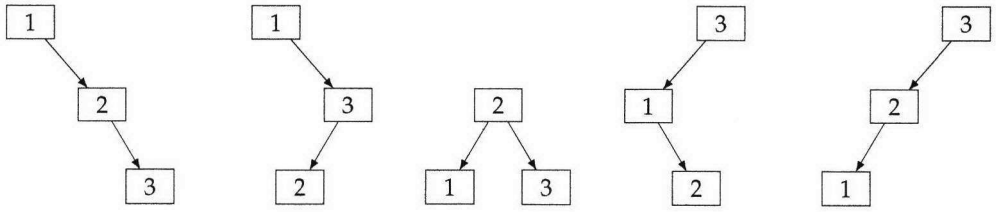

Figure 14.3: Five distinct BSTs for the traversal sequence $\langle 1, 2, 3 \rangle$.

However, the story for a preorder sequence is different. As an example, consider the preorder key sequence $\langle 43, 23, 37, 29, 31, 41, 47, 53 \rangle$. The root must hold 43, since it's the first visited node. The left subtree contains keys less than 43, i.e., 23, 37, 29, 31, 41, and the right subtree contains keys greater than 43, i.e., 47, 53. Furthermore, $\langle 23, 37, 29, 31, 41 \rangle$ is exactly the preorder sequence for the left subtree and $\langle 47, 53 \rangle$ is exactly the preorder sequence for the right subtree. We can recursively reason that 23 and 47 are the roots of the left and right subtree, and continue to build the entire tree, which is exactly the subtree rooted at Node I in Figure 14.1 on Page 198.

Generalizing, in any preorder traversal sequence, the first key corresponds to the root. The subsequence which begins at the second element and ends at the last key less than the root, corresponds to the preorder traversal of the root's left subtree. The final subsequence, consisting of keys greater than the root corresponds to the preorder traversal of the root's right subtree. We recursively reconstruct the BST by recursively reconstructing the left and right subtrees from the two subsequences then adding them to the root.

```
def rebuild_bst_from_preorder(preorder_sequence):
    if not preorder_sequence:
        return None

    transition_point = next((i for i, a in enumerate(preorder_sequence)
                             if a > preorder_sequence[0]),
                            len(preorder_sequence))
    return BSTNode(
        preorder_sequence[0],
        rebuild_bst_from_preorder(preorder_sequence[1:transition_point]),
        rebuild_bst_from_preorder(preorder_sequence[transition_point:]))
```

The worst-case input for this algorithm is the pre-order sequence corresponding to a left-skewed tree. The worst-case time complexity satisfies the recurrence $W(n) = W(n-1) + O(n)$, which solves to $O(n^2)$. The best-case input is a sequence corresponding to a right-skewed tree, and the corresponding time complexity is $O(n)$. When the sequence corresponds to a balanced BST, the time complexity is given by $B(n) = 2B(n/2) + O(n)$, which solves to $O(n \log n)$.

The implementation above potentially iterates over nodes multiple times, which is wasteful. A better approach is to reconstruct the left subtree in the same iteration as identifying the nodes which lie in it. The code shown below takes this approach. The intuition is that we do not want to iterate from first entry after the root to the last entry smaller than the root, only to go back and

partially repeat this process for the root's left subtree. We can avoid repeated passes over nodes by including the range of keys we want to reconstruct the subtrees over. For example, looking at the preorder key sequence $\langle 43, 23, 37, 29, 31, 41, 47, 53 \rangle$, instead of recursing on $\langle 23, 37, 29, 31, 41 \rangle$ (which would involve an iteration to get the last element in this sequence). We can directly recur on $\langle 23, 37, 29, 31, 41, 47, 53 \rangle$, with the constraint that we are building the subtree on nodes whose keys are less than 43.

```
def rebuild_bst_from_preorder(preorder_sequence):
    def rebuild_bst_from_preorder_on_value_range(lower_bound, upper_bound):
        if root_idx[0] == len(preorder_sequence):
            return None

        root = preorder_sequence[root_idx[0]]
        if not lower_bound <= root <= upper_bound:
            return None
        root_idx[0] += 1
        # Note that rebuild_bst_from_preorder_on_value_range updates root_idx[0].
        # So the order of following two calls are critical.
        left_subtree = rebuild_bst_from_preorder_on_value_range(
            lower_bound, root)
        right_subtree = rebuild_bst_from_preorder_on_value_range(
            root, upper_bound)
        return BSTNode(root, left_subtree, right_subtree)

    root_idx = [0]  # Tracks current subtree.
    return rebuild_bst_from_preorder_on_value_range(float('-inf'), float('inf'))
```

The worst-case time complexity is $O(n)$, since it performs a constant amount of work per node. Note the similarity to Solution 24.20 on Page 377.

A postorder traversal sequence also uniquely specifies the BST, and the algorithm for reconstructing the BST is very similar to that for the preorder case.

14.6 Find the closest entries in three sorted arrays

Design an algorithm that takes three sorted arrays and returns one entry from each such that the minimum interval containing these three entries is as small as possible. For example, if the three arrays are $\langle 5, 10, 15 \rangle$, $\langle 3, 6, 9, 12, 15 \rangle$, and $\langle 8, 16, 24 \rangle$, then 15, 15, 16 lie in the smallest possible interval.

Hint: How would you proceed if you needed to pick three entries in a single sorted array?

Solution: The brute-force approach is to try all possible triples, e.g., with three nested for loops. The length of the minimum interval containing a set of numbers is simply the difference of the maximum and the minimum values in the triple. The time complexity is $O(lmn)$, where l, m, n are the lengths of each of the three arrays.

The brute-force approach does not take advantage of the sortedness of the input arrays. For the example in the problem description, the smallest intervals containing (5, 3, 16) and (5, 3, 24) must be larger than the smallest interval containing (5, 3, 8) (since 8 is the maximum of 5, 3, 8, and $8 < 16 < 24$).

Let's suppose we begin with the triple consisting of the smallest entries in each array. Let s be the minimum value in the triple and t the maximum value in the triple. Then the smallest interval

with left endpoint s containing elements from each array must be $[s, t]$, since the remaining two values are the minimum possible.

Now remove s from the triple and bring the next smallest element from the array it belongs to into the triple. Let s' and t' be the next minimum and maximum values in the new triple. Observe $[s', t']$ must be the smallest interval whose left endpoint is s': the other two values are the smallest values in the corresponding arrays that are greater than or equal to s'. By iteratively examining and removing the smallest element from the triple, we compute the minimum interval starting at that element. Since the minimum interval containing elements from each array must begin with the element of *some* array, we are guaranteed to encounter the minimum element.

For example, we begin with $(5, 3, 8)$. The smallest interval whose left endpoint is 3 has length $8 - 3 = 5$. The element after 3 is 6, so we continue with the triple $(5, 6, 8)$. The smallest interval whose left endpoint is 5 has length $8 - 5 = 3$. The element after 5 is 10, so we continue with the triple $(10, 6, 8)$. The smallest interval whose left endpoint is 6 has length $10 - 6 = 4$. The element after 6 is 9, so we continue with the triple $(10, 9, 8)$. Proceeding in this way, we obtain the triples $(10, 9, 16)$, $(10, 12, 16)$, $(15, 12, 16)$, $(15, 15, 16)$. Out of all these triples, the one contained in a minimum length interval is $(15, 15, 16)$.

In the following code, we implement a general purpose function which finds the closest entries in k sorted arrays. Since we need to repeatedly insert, delete, find the minimum, and find the maximum amongst a collection of k elements, a BST is the natural choice.

```
def find_closest_elements_in_sorted_arrays(sorted_arrays):
    min_distance_so_far = float('inf')
    # Stores array iterators in each entry.
    iters = bintrees.RBTree()
    for idx, sorted_array in enumerate(sorted_arrays):
        it = iter(sorted_array)
        first_min = next(it, None)
        if first_min is not None:
            iters.insert((first_min, idx), it)

    while True:
        min_value, min_idx = iters.min_key()
        max_value = iters.max_key()[0]
        min_distance_so_far = min(max_value - min_value, min_distance_so_far)
        it = iters.pop_min()[1]
        next_min = next(it, None)
        # Return if some array has no remaining elements.
        if next_min is None:
            return min_distance_so_far
        iters.insert((next_min, min_idx), it)
```

The time complexity is $O(n \log k)$, where n is the total number of elements in the k arrays. For the special case $k = 3$ specified in the problem statement, the time complexity is $O(n \log 3) = O(n)$.

14.7 Enumerate numbers of the form $a + b\sqrt{2}$

Numbers of the form $a + b\sqrt{q}$, where a and b are nonnegative integers, and q is an integer which is not the square of another integer, have special properties, e.g., they are closed under addition and multiplication. Some of the first few numbers of this form are given in Figure 14.4 on the following page.

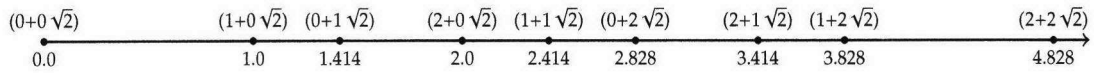

Figure 14.4: Some points of the form $a + b\sqrt{2}$. (For typographical reasons, this figure does not include all numbers of the form $a + b\sqrt{2}$ between 0 and $2 + 2\sqrt{2}$, e.g., $3 + 0\sqrt{2}, 4 + 0\sqrt{2}, 0 + 3\sqrt{2}, 3 + 1\sqrt{2}$ lie in the interval but are not included.)

Design an algorithm for efficiently computing the k smallest numbers of the form $a + b\sqrt{2}$ for nonnegative integers a and b.

Hint: Systematically enumerate points.

Solution: A key fact about $\sqrt{2}$ is that it is irrational, i.e., it cannot equal to $\frac{a}{b}$ for any integers a, b. This implies that if $x + y\sqrt{2} = x' + y'\sqrt{2}$, where x and y are integers, then $x = x'$ and $y = y'$ (since otherwise $\sqrt{2} = \frac{x-x'}{y-y'}$).

Here is a brute-force solution. Generate all numbers of the form $a + b\sqrt{2}$ where a and b are integers, $0 \le a, b \le k - 1$. This yields exactly k^2 numbers and the k smallest numbers must lie in this collection. We can sort these numbers and return the k smallest ones. The time complexity is $O(k^2 \log(k^2)) = O(k^2 \log k)$.

Intuitively, it is wasteful to generate k^2 numbers, since we only care about a small fraction of them.

We know the smallest number is $0 + 0\sqrt{2}$. The candidates for next smallest number are $1 + 0\sqrt{2}$ and $0 + 1\sqrt{2}$. From this, we can deduce the following algorithm. We want to maintain a collection of real numbers, initialized to $0 + 0\sqrt{2}$. We perform k extractions of the smallest element, call it $a + b\sqrt{2}$, followed by insertion of $(a + 1) + b\sqrt{2}$ and $a + (b + 1)\sqrt{2}$ to the collection.

The operations on this collection are extract the minimum and insert. Since it is possible that the same number may be inserted more than once, we need to ensure the collection does not create duplicates when the same item is inserted twice. A BST satisfies these operations efficiently, and is used in the implementation below. It is initialized to contain $0 + 0\sqrt{2}$. We extract the minimum from the BST, which is $0 + 0\sqrt{2}$, and insert $1 + 0\sqrt{2}$ and $0 + 1\sqrt{2}$ to the BST. We extract the minimum from the BST, which is $1 + 0\sqrt{2}$, and insert $2 + 0\sqrt{2}$ and $1 + 1\sqrt{2}$ to the BST, which now consists of $0 + 1\sqrt{2} = 1.414, 2 + 0\sqrt{2} = 2, 1 + 1\sqrt{2} = 2.414$. We extract the minimum from the BST, which is $0 + 1\sqrt{2}$, and insert $1 + 1\sqrt{2}$ and $0 + 2\sqrt{2}$. The first value is already present, so the BST updates to $2 + 0\sqrt{2} = 2, 1 + 1\sqrt{2} = 2.414, 0 + 2\sqrt{2} = 2.828$. (Although it's not apparent from this small example, the values we add back to the BST may be smaller than values already present in it, so we really need the BST to hold values.)

```
class ABSqrt2:
    def __init__(self, a, b):
        self.a, self.b = a, b
        self.val = a + b * math.sqrt(2)

    def __lt__(self, other):
        return self.val < other.val

    def __eq__(self, other):
        return self.val == other.val
```

```
def generate_first_k_a_b_sqrt2(k):
    # Initial for 0 + 0 * sqrt(2).
    candidates = bintrees.RBTree([(ABSqrt2(0, 0), None)])

    result = []
    while len(result) < k:
        next_smallest = candidates.pop_min()[0]
        result.append(next_smallest.val)
        # Adds the next two numbers derived from next_smallest.
        candidates[ABSqrt2(next_smallest.a + 1, next_smallest.b)] = None
        candidates[ABSqrt2(next_smallest.a, next_smallest.b + 1)] = None
    return result
```

In each iteration we perform a deletion and two insertions. There are k such insertions, so the time complexity is $O(k \log k)$. The space complexity is $O(k)$, since there are not more than $2k$ insertions.

Now we describe an $O(n)$ time solution. It is simple to implement, but is less easy to understand than the one based on BST. The idea is that the $(n+1)$th value will be the sum of 1 or $\sqrt{2}$ with a previous value. We could iterate through all the entries in the result and track the smallest such value which is greater than nth value. However, this takes time $O(n)$ to compute the $(n+1)$th element.

Intuitively, there is no need to examine all prior values entries when computing $(n+1)$th value. Let's say we are storing the result in an array A. Then we need to track just two entries—i, the smallest index such that $A[i] + 1 > A[n-1]$, and j, smallest index such that $A[j] + \sqrt{2} > A[n-1]$. Clearly, the $(n+1)$th entry will be the smaller of $A[i]+1$ and $A[j] + \sqrt{2}$. After obtaining the $(n+1)$th entry, if it is $A[i] + 1$, we increment i. If it is $A[j] + \sqrt{2}$, we increment j. If $A[i] + 1$ equals $A[j] + \sqrt{2}$, we increment both i and j.

To illustrate, suppose A is initialized to $\langle 0 \rangle$, and i and j are 0. The computation proceeds as follows:

(1.) Since $A[0] + 1 = 1 < A[0] + \sqrt{2} = 1.414$, we push 1 into A and increment i. Now $A = \langle 0, 1 \rangle$, $i = 1, j = 0$.

(2.) Since $A[1] + 1 = 2 > A[0] + \sqrt{2} = 1.414$, we push 1.414 into A and increment j. Now $A = \langle 0, 1, 1.414 \rangle$, $i = 1, j = 1$.

(3.) Since $A[1] + 1 = 2 < A[1] + \sqrt{2} = 2.414$, we push 2 into A and increment i. Now $A = \langle 0, 1, 1.414, 2 \rangle$, $i = 2, j = 1$.

(4.) Since $A[2] + 1 = 2.414 = A[1] + \sqrt{2} = 2.414$, we push 2.414 into A and increment both i and j. Now $A = \langle 0, 1, 1.414, 2, 2.414 \rangle$, $i = 3, j = 2$.

(5.) Since $A[3] + 1 = 3 > A[2] + \sqrt{2} = 2.828$, we push 2.828 into A and increment j. Now $A = \langle 0, 1, 1.414, 2, 2.828 \rangle$, $i = 3, j = 3$.

(6.) Since $A[3] + 1 = 3 < A[3] + \sqrt{2} = 3.414$, we push 3 into A and increment i. Now $A = \langle 0, 1, 1.414, 2, 2.828, 3 \rangle$, $i = 4, j = 3$.

```
def generate_first_k_a_b_sqrt2(k):
    # Will store the first k numbers of the form a + b sqrt(2).
    cand = [ABSqrt2(0, 0)]
    i = j = 0
    for _ in range(1, k):
        cand_i_plus_1 = ABSqrt2(cand[i].a + 1, cand[i].b)
        cand_j_plus_sqrt2 = ABSqrt2(cand[j].a, cand[j].b + 1)
        cand.append(min(cand_i_plus_1, cand_j_plus_sqrt2))
        if cand_i_plus_1.val == cand[-1].val:
```

```
            i += 1
        if cand_j_plus_sqrt2.val == cand[-1].val:
            j += 1
    return [a.val for a in cand]
```

Each additional element takes $O(1)$ time to compute, implying an $O(n)$ time complexity to compute the first n values of the form $a + b\sqrt{2}$.

14.8 Build a minimum height BST from a sorted array

Given a sorted array, the number of BSTs that can be built on the entries in the array grows enormously with its size. Some of these trees are skewed, and are closer to lists; others are more balanced. See Figure 14.3 on Page 205 for an example.

How would you build a BST of minimum possible height from a sorted array?

Hint: Which element should be the root?

Solution: Brute-force is not much help here—enumerating all possible BSTs for the given array in search of the minimum height one requires a nontrivial recursion, not to mention enormous time complexity.

Intuitively, to make a minimum height BST, we want the subtrees to be as balanced as possible—there's no point in one subtree being shorter than the other, since the height is determined by the taller one. More formally, balance can be achieved by keeping the number of nodes in both subtrees as close as possible.

Let n be the length of the array. To achieve optimum balance we can make the element in the middle of the array, i.e., the $\lfloor \frac{n}{2} \rfloor$th entry, the root, and recursively compute minimum height BSTs for the subarrays on either side of this entry.

As a concrete example, if the array is $\langle 2, 3, 5, 7, 11, 13, 17, 19, 23 \rangle$, the root's key will be the middle element, i.e., 11. This implies the left subtree is to be built from $\langle 2, 3, 5, 7 \rangle$, and the right subtree is to be built from $\langle 13, 17, 19, 23 \rangle$. To make both of these minimum height, we call the procedure recursively.

```
def build_min_height_bst_from_sorted_array(A):
    def build_min_height_bst_from_sorted_subarray(start, end):
        if start >= end:
            return None
        mid = (start + end) // 2
        return BSTNode(A[mid],
                       build_min_height_bst_from_sorted_subarray(start, mid),
                       build_min_height_bst_from_sorted_subarray(mid + 1, end))

    return build_min_height_bst_from_sorted_subarray(0, len(A))
```

The time complexity $T(n)$ satisfies the recurrence $T(n) = 2T(n/2) + O(1)$, which solves to $T(n) = O(n)$. Another explanation for the time complexity is that we make exactly n calls to the recursive function and spend $O(1)$ within each call.

14.9 Test if three BST nodes are totally ordered

Write a program which takes two nodes in a BST and a third node, the "middle" node, and determines if one of the two nodes is a proper ancestor and the other a proper descendant of the middle. (A proper ancestor of a node is an ancestor that is not equal to the node; a proper descendant is defined similarly.) For example, in Figure 14.1 on Page 198, if the middle is Node J, your function should return true if the two nodes are $\{A, K\}$ or $\{I, M\}$. It should return false if the two nodes are $\{I, P\}$ or $\{J, K\}$. You can assume that all keys are unique. Nodes do not have pointers to their parents

Hint: For what specific arrangements of the three nodes does the check pass?

Solution: A brute-force approach would be to check if the first node is a proper ancestor of the middle and the second node is a proper descendant of the middle. If this check returns true, we return true. Otherwise, we return the result of the same check, swapping the roles of the first and second nodes. For the BST in Figure 14.1 on Page 198, with the two nodes being $\{L, I\}$ and middle K, searching for K from L would be unsuccessful, but searching for K from I would succeed. We would then search for L from K, which would succeed, so we would return true.

Searching has time complexity $O(h)$, where h is the height of the tree, since we can use the BST property to prune one of the two children at each node. Since we perform a maximum of three searches, the total time complexity is $O(h)$.

One disadvantage of trying the two input nodes for being the middle's ancestor one-after-another is that even when the three nodes are very close, e.g., if the two nodes are $\{A, J\}$ and middle node is I in Figure 14.1 on Page 198, if we begin the search for the middle from the lower of the two nodes, e.g., from J, we incur the full $O(h)$ time complexity.

We can prevent this by performing the searches for the middle from both alternatives in an interleaved fashion. If we encounter the middle from one node, we subsequently search for the second node from the middle. This way we avoid performing an unsuccessful search on a large subtree. For the example of $\{A, J\}$ and middle I in Figure 14.1 on Page 198, we would search for I from both A and J, stopping as soon as we get to I from A, thereby avoiding a wasteful search from J. (We would still have to search for J from I to complete the computation.)

```
def pair_includes_ancestor_and_descendant_of_m(possible_anc_or_desc_0,
                                               possible_anc_or_desc_1, middle):
    search_0, search_1 = possible_anc_or_desc_0, possible_anc_or_desc_1

    # Perform interleaved searching from possible_anc_or_desc_0 and
    # possible_anc_or_desc_1 for middle.
    while (search_0 is not possible_anc_or_desc_1 and search_0 is not middle
           and search_1 is not possible_anc_or_desc_0 and search_1 is not middle
           and (search_0 or search_1)):
        if search_0:
            search_0 = (search_0.left
                        if search_0.data > middle.data else search_0.right)
        if search_1:
            search_1 = (search_1.left
                        if search_1.data > middle.data else search_1.right)

    # If both searches were unsuccessful, or we got from
    # possible_anc_or_desc_0 to possible_anc_or_desc_1 without seeing middle,
    # or from possible_anc_or_desc_1 to possible_anc_or_desc_0 without seeing
    # middle, middle cannot lie between possible_anc_or_desc_0 and
```

```
    # possible_anc_or_desc_1.
    if ((search_0 is not middle and search_1 is not middle)
            or search_0 is possible_anc_or_desc_1
            or search_1 is possible_anc_or_desc_0):
        return False

    def search_target(source, target):
        while source and source is not target:
            source = source.left if source.data > target.data else source.right
        return source is target

    # If we get here, we already know one of possible_anc_or_desc_0 or
    # possible_anc_or_desc_1 has a path to middle. Check if middle has a path
    # to possible_anc_or_desc_1 or to possible_anc_or_desc_0.
    return search_target(middle, possible_anc_or_desc_1
                         if search_0 is middle else possible_anc_or_desc_0)
```

When the middle node does have an ancestor and descendant in the pair, the time complexity is $O(d)$, where d is the difference between the depths of the ancestor and descendant. The reason is that the interleaved search will stop when the ancestor reaches the middle node, i.e., after $O(d)$ iterations. The search from the middle node to the descendant then takes $O(d)$ steps to succeed. When the middle node does not have an ancestor and descendant in the pair, the time complexity is $O(h)$, which corresponds to a worst-case search in a BST.

14.10 The range lookup problem

Consider the problem of developing a web-service that takes a geographical location, and returns the nearest restaurant. The service starts with a set of restaurant locations—each location includes X and Y-coordinates. A query consists of a location, and should return the nearest restaurant (ties can be broken arbitrarily).

One approach is to build two BSTs on the restaurant locations: T_X sorted on the X coordinates, and T_Y sorted on the Y coordinates. A query on location (p, q) can be performed by finding all the restaurants whose X coordinate is in the interval $[p - D, p + D]$, and all the restaurants whose Y coordinate is in the interval $[q - D, q + D]$, taking the intersection of these two sets, and finding the restaurant in the intersection which is closest to (p, q). Heuristically, if D is chosen correctly, the subsets are small and a brute-force search for the closest point is fast. One approach is to start with a small value for D and keep doubling it until the final intersection is nonempty.

There are other data structures which are more robust, e.g., Quadtrees and k-d trees, but the approach outlined above works well in practice.

Write a program that takes as input a BST and an interval and returns the BST keys that lie in the interval. For example, for the tree in Figure 14.1 on Page 198, and interval $[16, 31]$, you should return $17, 19, 23, 29, 31$.

Hint: How many edges are traversed when the successor function is repeatedly called m times?

Solution: A brute-force approach would be to perform a traversal (inorder, postorder, or preorder) of the BST and record the keys in the specified interval. The time complexity is that of the traversal, i.e., $O(n)$, where n is the number of nodes in the tree.

The brute-force approach does not exploit the BST property—it would work unchanged for an arbitrary binary tree.

We can use the BST property to prune the traversal as follows:

- If the root of the tree holds a key that is less than the left endpoint of the interval, the left subtree cannot contain any node whose key lies in the interval.
- If the root of the tree holds a key that is greater than the right endpoint of the interval, the right subtree cannot contain any node whose key lies in the interval.
- Otherwise, the root of the tree holds a key that lies within the interval, and it is possible for both the left and right subtrees to contain nodes whose keys lie in the interval.

For example, for the tree in Figure 14.1 on Page 198, and interval [16, 42], we begin the traversal at A, which contains 19. Since 19 lies in [16, 42], we explore both of A's children, namely B and I. Continuing with B, we see B's key 7 is less than 16, so no nodes in B's left subtree can lie in the interval [16, 42]. Similarly, when we get to I, since $43 > 42$, we need not explore I's right subtree.

```
Interval = collections.namedtuple('Interval', ('left', 'right'))


def range_lookup_in_bst(tree, interval):
    def range_lookup_in_bst_helper(tree):
        if tree is None:
            return

        if interval.left <= tree.data <= interval.right:
            # tree.data lies in the interval.
            range_lookup_in_bst_helper(tree.left)
            result.append(tree.data)
            range_lookup_in_bst_helper(tree.right)
        elif interval.left > tree.data:
            range_lookup_in_bst_helper(tree.right)
        else:  # interval.right > tree.data
            range_lookup_in_bst_helper(tree.left)

    result = []
    range_lookup_in_bst_helper(tree)
    return result
```

The time complexity is tricky to analyze. It makes sense to reason about time complexity in terms of the number of keys m that lie in the specified interval. We partition the nodes into two categories—those that the program recurses on and those that it does not. For our working example, the program recurses on $A, B, F, G, H, I, J, K, L, M, N$. Not all of these have keys in the specified interval, but no nodes outside of this set can have keys in the interval. Looking more carefully at the nodes we recurse on, we see these nodes can be partitioned into three subsets—nodes on the search path to 16, nodes on the search path to 42, and the rest. All nodes in the third subset must lie in the result, but some of the nodes in the first two subsets may or may not lie in the result. The traversal spends $O(h)$ time visiting the first two subsets, and $O(m)$ time traversing the third subset—each edge is visited twice, once downwards, once upwards. Therefore the total time complexity is $O(m + h)$, which is much better than $O(n)$ brute-force approach when the tree is balanced, and very few keys lie in the specified range.

Augmented BSTs

Thus far we have considered BSTs in which each node stores a key, a left child, a right child, and, possibly, the parent. Adding fields to the nodes can speed up certain queries. As an example, consider the following problem.

Suppose you needed a data structure that supports efficient insertion, deletion, lookup of integer keys, as well as range queries, i.e., determining the number of keys that lie in an interval.

We could use a BST, which has efficient insertion, deletion and lookup. To find the number of keys that lie in the interval $[U, V]$, we could search for the first node with a key greater than or equal to U, and then call the successor operation (9.10 on Page 123) until we reach a node whose key is greater than V (or we run out of nodes). This has $O(h + m)$ time complexity, where h is the height of the tree and m is the number of nodes with keys within the interval. When m is large, this become slow.

We can do much better by augmenting the BST. Specifically, we add a size field to each node, which is the number of nodes in the BST rooted at that node.

For simplicity, suppose we want to find the number of entries that are less than a specified value. As an example, say we want to count the number of keys less than 40 in the BST in Figure 14.1 on Page 198, and that each node has a size field. Since the root A's key, 19, is less than 40, the BST property tells us that all keys in A's left subtree are less than 40. Therefore we can add 7 (which we get from the left child's size field) and 1 (for A itself) to the running count, and recurse with A's right child.

Generalizing, let's say we want to count all entries less than v. We initialize count to 0. Since there can be duplicate keys in the tree, we search for the first occurrence of v in an inorder traversal using Solution 14.2 on Page 201. (If v is not present, we stop when we have determined this.) Each time we take a left child, we leave count unchanged; each time we take a right child, we add one plus the size of the corresponding left child. If v is present, when we reach the first occurrence of v, we add the size of v's left child, The same approach can be used to find the number of entries that are greater than v, less than or equal to v, and greater than or equal to v.

For example, to count the number of less than 40 in the BST in Figure 14.1 on Page 198 we would search for 40. Since A's key, 19 is less than 40, we update count to $7 + 1 = 8$ and continue from I. Since I's key, 43 is greater than 40, we move to I's left child, J. Since J's key, 23 is less than 40, update count to $8 + 1 = 9$, and continue from K. Since K's key, 37 is less than 40, we update count to $9 + 2 + 1 = 12$, and continue from N. Since N's key, 41, is greater than 40, we move to N's left child, which is empty. No other keys can be less than 40, so we return count, i.e., 12. Note how we avoided exploring A and K's left subtrees.

The time bound for these computations is $O(h)$, since the search always descends the tree. When m is large, e.g., comparable to the total number of nodes in the tree, this approach is much faster than repeated calling successor.

To compute the number of nodes with keys in the interval $[L, U]$, first compute the number of nodes with keys less than L and the number of nodes with keys greater than U, and subtract that from the total number of nodes (which is the size stored at the root).

The size field can be updated on insert and delete without changing the $O(h)$ time complexity of both. Essentially, the only nodes whose size field change are those on the search path to the added/deleted node. Some conditional checks are needed for each such node, but these add constant time per node, leaving the $O(h)$ time complexity unchanged.

14.11 Add credits

Consider a server that a large number of clients connect to. Each client is identified by a string. Each client has a "credit", which is a nonnegative integer value. The server needs to maintain a data structure to which clients can be added, removed, queried, or updated. In addition, the server needs to be able to add a specified number of credits to all clients simultaneously.

Design a data structure that implements the following methods:

- Insert: add a client with specified credit, replacing any existing entry for the client.
- Remove: delete the specified client.
- Lookup: return the number of credits associated with the specified client.
- Add-to-all: increment the credit count for all current clients by the specified amount.
- Max: return a client with the highest number of credits.

Hint: Use additional global state.

Solution: A hash table is a natural data structure for this application. However, it does not support efficient max operations, nor is there an obvious way to perform the simultaneous increment, short traversing all entries. A BST does have efficient max operation, but it too does not natively support the global increment.

A general principle when adding behaviors to an object is to wrap the object, and add functions in the wrapper, which add behaviors before or after delegating to the object. In our context, this suggests storing the clients in a BST, and having the wrapper track the total increment amount.

For example, if we have clients A, B, C, with credits $1, 2, 3$, respectively, and want to add 5 credits to each, the wrapper sets the total increment amount to 5. A lookup on B then is performed by looking up in the BST, which returns 2, and then adding 5 before returning. If we want to add 4 more credits to each, we simply update the total increment amount to 9.

One issue to watch out for is what happens to clients inserted after a call to the add-to-all function. Continuing with the given example, if we were to now add D with a credit of 6, the lookup would return $6 + 9$, which is an error.

The solution is simple—subtract the increment from the credit, i.e., add D with a credit of $6 - 9 = -3$ to the BST. Now a lookup for D will return $-3 + 9$, which is the correct amount.

More specifically, the BST keys are credits, and the corresponding values are the clients with that credit. This makes for fast max-queries. However, to perform lookups and removes by client quickly, the BST by itself is not enough (since it is ordered by credit, not client id). We can solve this by maintaining an additional hash table in which keys are clients, and values are credits. Lookup is trivial. Removes entails a lookup in the hash to get the credit, and then a search into the BST to get the set of clients with that credit, and finally a delete on that set.

```
class ClientsCreditsInfo:
    def __init__(self):
        self._offset = 0
        self._client_to_credit = {}
        self._credit_to_clients = bintrees.RBTree()

    def insert(self, client_id, c):
        self.remove(client_id)
        self._client_to_credit[client_id] = c - self._offset
        self._credit_to_clients.setdefault(c - self._offset,
                                           set()).add(client_id)
```

```
def remove(self, client_id):
    credit = self._client_to_credit.get(client_id)
    if credit is not None:
        self._credit_to_clients[credit].remove(client_id)
        if not self._credit_to_clients[credit]:
            del self._credit_to_clients[credit]
        del self._client_to_credit[client_id]
        return True
    return False

def lookup(self, client_id):
    credit = self._client_to_credit.get(client_id)
    return -1 if credit is None else credit + self._offset

def add_all(self, C):
    self._offset += C

def max(self):
    if not self._credit_to_clients:
        return ''
    clients = self._credit_to_clients.max_item()[1]
    return '' if not clients else next(iter(clients))
```

The time complexity to insert and remove is dominated by the BST, i.e., $O(\log n)$, where n is the number of clients in the data structure. Lookup and add-to-all operate only on the hash table, and have $O(1)$ time complexity. Library BST implementations uses caching to perform max in $O(1)$ time.