5

BINARY SEARCH TREES

Binary search trees use the concepts underpinning the binary search algorithm to create a dynamic data structure. The key word here is *dynamic*. Unlike sorted arrays, binary search trees support the efficient addition and removal of elements in addition to searches, making them the perfect blend of the algorithmic efficiency of binary search and the adaptability of dynamic data structures. They also make for wonderful decorative mobiles for any room.

In addition to introducing binary search trees, this chapter discusses algorithms for searching for values, adding new values, and deleting values. It shows how to use pointers to create branching structures more powerful than the list-based structures in previous chapters. You'll learn how, by carefully structuring the relationships among the values, we can encode the approach used for binary search into the very structure of the data itself.

Binary Search Tree Structure

Trees are hierarchical data structures composed of branching chains of nodes. They are a natural extension of linked lists, where each tree node is permitted two `next` pointers that point to subsequent nodes in disjoint lists. Figure 5-1 shows a sample binary search tree.

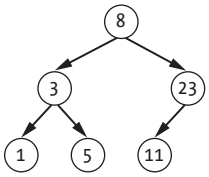

Figure 5-1: An example binary search tree

A node contains a value (of a given type) and up to two pointers to lower nodes in the tree, as shown in Figure 5-2. We call nodes with at least one child *internal nodes* and nodes without any children *leaf nodes*.

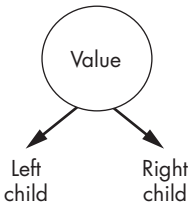

Figure 5-2: The required components of a binary search tree node

Tree nodes may contain other information, depending on their use. We often store a pointer back to the node's parent, for instance. This single piece of additional information allows us to traverse the tree from the bottom up as well as from the top down, which comes in handy when we consider removing nodes.

Formally, we specify a *binary search tree node* as a data structure with this minimal information: a value (or key), pointers to two child nodes (either of which can be set to `null` if no corresponding child exists), and an optional pointer to the parent node.

```
TreeNode {
    Type: value
    TreeNode: left
    TreeNode: right
    TreeNode: parent
}
```

We might also want to store auxiliary data. Storing and searching for individual values are useful, but using these values as keys for looking up

more detailed information greatly extends the power of the data structure. For example, we could use the names of our favorite coffees as the node's values, allowing us to efficiently look up records for any coffee. In this case, our auxiliary data would be a detailed record of everything we know about that coffee. Or our values could be timestamps, and the nodes could contain indications of which coffee we brewed at that time, allowing us to efficiently search our historical coffee consumption. The tree node data structure can either store this auxiliary data directly or include a pointer to a composite data structure located somewhere else in memory.

Binary search trees start at a single *root* node at the top of the tree and branch into multiple paths as they descend, as shown in Figure 5-3. This structure allows programs to access the binary search tree through a single pointer—the location of its root node.

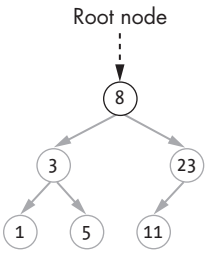

Figure 5-3: The root node indicates the top of the binary search tree and is the starting location for operations.

Botanical purists may draw trees with the root node at the bottom of the tree and nodes branching upward, instead of starting from the top as in Figure 5-3. However, the representations are equivalent. In truth, both the top-down and bottom-up illustrations hide the actual complexity of the binary search tree. Like a linked list, a search tree's individual nodes can be scattered throughout the computer's memory. Each node is only linked to its children and parents through the power and flexibility of pointers.

The power of the binary search tree stems from how values are organized within the tree. The *binary search tree property* states:

> For any node *N*, the value of any node in *N*'s left subtree is less than *N*'s value, and the value of any node in *N*'s right subtree is greater than *N*'s value.

In other words, the tree is organized by the values at each node, as shown in Figure 5-4. The values of the data in the left node and all nodes below it are less than the value of the current node. Similarly, the values of the data in the right node and all nodes below it are greater than the value of the current node. The values thus serve two roles. First, and most obviously, they indicate the value stored at that node. Second, they define the tree's structure below that node by partitioning the subtree into two subsets.

The above definition implicitly restricts the binary search tree to contain unique values. It is possible to define binary search trees that allow

duplicate values by modifying the binary search tree property accordingly. Other references may vary in whether they allow duplicate values and thus how they handle equality in the binary search tree property. This chapter focuses on the case of non-duplicate values to stay consistent with other indexing data structures we will explore in the book, such as skiplists and hash tables, though the algorithms presented can be adapted to handle duplicates.

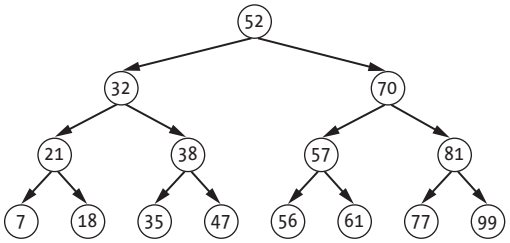

Figure 5-4: The values of the nodes in a binary search tree are ordered by the binary search tree property.

We could compare a binary search tree's structure to a public relations department that is organized by level of humor. Each employee measures their humor level with a single numerical value, the number of funny illustrations in a 30-minute presentation. A score of 0 represents the serious presenter who includes only technical diagrams. A score of 100 or above represents the aspiring comedian who adds multiple jokes to every slide. The entire department structures itself around this one metric. Internal nodes represent managers with either one or two direct reports. Each manager considers their own humor level and partitions their suborganization accordingly. Team members who include more jokes (a *larger* humor level) go in the right subteam. Those who include fewer jokes (a *smaller* humor level) go in the left subteam. Each manager thus provides both a partitioning function and a middle ground between the two subteams.

Although this ordering of nodes might not seem like a lot of structure, keep in mind the amount of power we got from using a similar property within binary search. The binary search tree property is effectively keeping the data within the tree sorted with respect to its position in the tree. As we will see, this allows us to not only efficiently find values in the tree but also efficiently add and remove nodes.

Searching Binary Search Trees

We search a binary search tree by walking down from the root node. At each step, we determine whether to explore the left or right subtree by comparing the value at the current node with the target value. If the target value is less than the current value, the search progresses to the left. If the target value is greater than the current value, the search progresses to the right. The node's value thus serves the same function as those helpful signs

in hotels that tell us rooms 500–519 are to the left and rooms 520–590 are to the right. With one quick check, we can make the appropriate turn and ignore the rooms in the other direction. The search ends when either the target value is found or it reaches a node with no children in the correct direction. In the latter case, we can definitively say that the target value is not in the tree.

Iterative and Recursive Searches

We implement this search with either an iterative or recursive approach. The following code uses a recursive approach, where the search function calls itself using the next node in the tree, initially called on the root node of the tree. The code returns a pointer to the node containing the value, allowing us to retrieve any auxiliary information from the node.

```
FindValue(TreeNode: current, Type: target):
 ❶ IF current == null:
       return null
 ❷ IF current.value == target:
       return current
 ❸ IF target < current.value AND current.left != null:
       return FindValue(current.left, target)
 ❹ IF target > current.value AND current.right != null:
       return FindValue(current.right, target)
 ❺ return null
```

This algorithm performs only a few tests at each node; if any of the tests pass, we end the function by returning a value. First, the code checks that the `current` node is not `null`, which can happen when searching an empty tree. If it is `null`, the tree is empty and, by definition, does not contain the value of interest ❶. Second, if the current node's value equals our target value, the code has found the value of interest and returns the node ❷. Third, the code checks whether it should explore the left subtree and, if so, returns whatever it finds from that exploration ❸. Fourth, the code checks whether it should explore the right subtree and, if so, return whatever it finds from that exploration ❹. Note that in both the left and right cases, the code also checks that the corresponding child exists. If none of the tests trigger, the code has made it to a node that doesn't match our target value and does not have a child in the correct direction. It has reached a dead end and is forced to admit defeat by returning a failure value such as `null` ❺. A dead end occurs whenever there is no child in the correct direction, so it is possible for an internal node with a single child to still be a dead end for a search.

Suppose we used this strategy to search Figure 5-5 for the value 63. We start at the root node and compare its value (50) to that of our target. Since 50 is less than 63, we know that the target value is not in the left-hand branch, where every node has a value less than 50. This simple fact allows us to *prune* the entire left-hand subtree from our search. We can avoid checking 11 of the 22 nodes in our tree with a single comparison. This test

is effectively the same as the pruning we did within the binary search algorithm from Chapter 2: we test a single element against our target value and use that to prune out a large section of our search space.

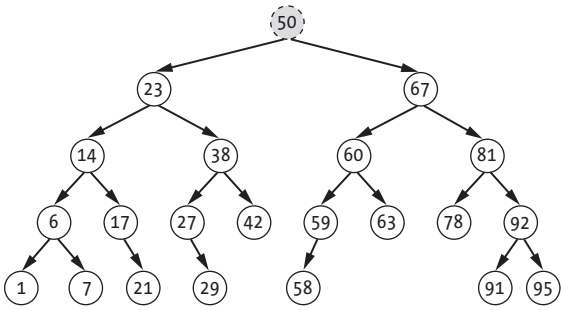

Figure 5-5: The first step in the search of a binary search tree. The search begins at the root node.

Our search progresses down the right-hand subtree to the node with value 67, as shown in Figure 5-6. We again employ the binary search tree property to rule out half the remaining search space. In this case 63 is less than 67, so we choose the left subtree. Anything in node 67's right-hand subtree must be larger than 67, and thus it cannot contain 63. We've pruned another 5 nodes.

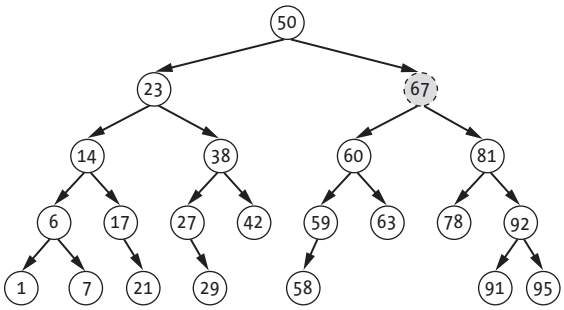

Figure 5-6: The second step in the search of a binary s earch tree

At this point, we can make definitive statements about the remaining search space underneath the current node. Since we branched right at 50 and left at 67, we know that all nodes in the new subtree will have values greater than 50 and less than 67. In fact, each time we take a right-hand branch, we're tightening the lower bound of the remaining search space. Whenever we take a left-hand branch, we're tightening the upper bound.

The search continues down the tree, traversing each of the shaded nodes as shown in Figure 5-7. The search passes through 4 of the 22 nodes before finding the target value.

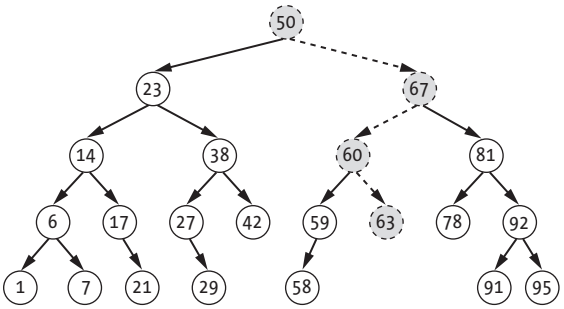

Figure 5-7: The complete path of a search of a binary search tree for value 63

Consider this search in the context of the public relations department organized by humor metric. Suppose the department head needs to find a speaker for an informal presentation at an industry conference. After some consideration, they determine a humor level of 63 jokes per 30 minutes will be optimal for this audience. The department head (root node) considers their own humor level, realize they are too serious, and therefore asks their right-hand report to find someone within the report's organization. Everyone in the right-hand subtree is more comedic than the department head. That manager repeats the same steps of comparing their own humor level (67) with the target value and delegating to their appropriate report.

Of course, the search does not need to progress all the way down to a leaf node. As shown in Figure 5-8, the node in question might sit in the middle of the tree. If we search the same tree for the value of 14, we take two left branches and end at the appropriate internal node. The manager at this intermediate level perfectly fits our humor criterion and can give the talk. Thus, as we descend the tree, we need to check whether the current node is equal to our target value and terminate the search early if we find a match.

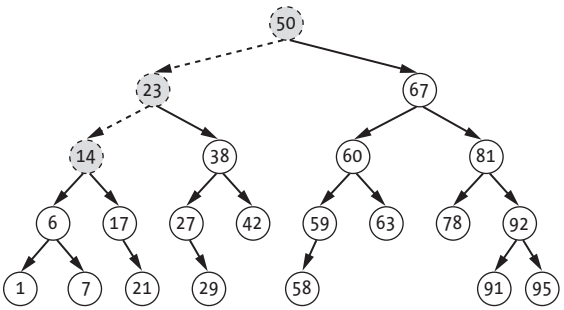

Figure 5-8: The search of a binary search tree can conclude at an internal node where the value matches our target value.

The iterative approach to searching a binary search tree replaces the recursion with a WHILE loop that iterates down the tree. The search again starts at the tree's root.

```
FindValueItr(TreeNode: root, Type: target):
  ❶ TreeNode: current = root
  ❷ WHILE current != null AND current.value != target:
      ❸ IF target < current.value:
            current = current.left
        ELSE:
            current = current.right
  ❹ return current
```

The code starts by creating a local variable current to point to the current node in the search ❶. Initially, this will be the root node, which may be null in an empty tree. Then a WHILE loop keeps iterating down the tree until it either hits a dead end (current == null) or finds the correct value (current.value == target) ❷. Within the loop, the code checks whether the next child should be to the left or right ❸ and reassigns current to point to the corresponding child. The function concludes by returning current, which is either the found node or, if the tree is empty or the value is not found, null ❹.

The computational cost of both the recursive and iterative searches is proportional to the depth of the target value in the tree. We start at the top of the tree and proceed down a single path. The deeper the tree, the more comparisons we need to perform. Structuring the tree to minimize its depth thus increases search efficiency.

Searching Trees vs. Searching Sorted Arrays

The skeptical reader might protest, "Chapter 2 already taught us how to do an efficient search on sorted data. Binary search scales logarithmically with the size of the data. You had illustrations and everything. Why bother putting the data in a tree rather than a sorted array? Are we adding unnecessary complexity and overhead with all these pointers?"

These concerns are reasonable. However, it's important to consider how the data structure and search will be used in a wider context. If our data is already in a sorted array and we want to search through it a single time, building a tree rather than simply performing a binary search does not help. In fact, building the tree itself is more expensive than a single linear scan. Similarly, if the data does not change, then sorting it once and using the sorted array may be preferable. We avoid the memory overhead of the tree structure itself. The tradeoffs change as our data becomes more dynamic.

Imagine the case where employees join or leave the PR department. In addition to the normal paperwork, the department needs to update its data structure of humor levels. Each new employee represents an addition to the list of humor levels. Each departure represents a deletion. Instead of using the reporting hierarchy, the department could use the office assignments to sort employees by humor level. The least humorous person is in office 1 and

the most humorous in office 100. The manager can still efficiently search for the correct speaker. However, they now need to fix the office assignments with each new addition or departure. For a large department or a high number of changes, the overhead increases. In highly dynamic environments, such as a list of pending restaurant orders, the costs can become significant.

The power of binary search trees, and dynamic data structures in general, arises in cases where the data is *changing*. As we will see in the next sections, binary search trees allow us to efficiently add and remove data points. In a sorted array, we would need to constantly update the array as we add and remove data, which can be expensive. In contrast, the binary search tree keeps the data in an easily searchable structure as the data itself changes. If we are doing many searches over a dynamic data set, this combination of efficiencies becomes critical.

Modifying Binary Search Trees

The root node always deserves special care when using or modifying a binary search tree. When searching for a node in the tree, we always start at the root node. When inserting the first node into the tree, such as the first person joining our PR department, we make that node our new root. And, as we will see later in the chapter, when removing a node from a binary tree, we must treat the root node as a special case.

We can simplify the logic for using binary search trees by wrapping the entire tree in a thin data structure that contains the root node:

```
BinarySearchTree {
    TreeNode: root
}
```

While this might seem like a waste (more complexity and an extra data structure), it provides an easy-to-use interface for the tree and greatly simplifies our handling of the root node. When using a wrapper data structure (or class) for our binary search tree, we also need to provide top-level functions to add or find nodes. These are relatively thin wrappers with a special case for handling a tree without any nodes.

To search a tree, the code again starts by checking whether the tree is empty (`tree.root == null`):

```
FindTreeNode(BinarySearchTree: tree, Type: target):
    IF tree.root == null:
        return null
    return FindValue(tree.root, target)
```

If the tree is empty, it immediately returns `null` to indicate the search failed to find a match. Otherwise, the code recursively searches the tree using `FindValue`. Performing the `null` check here can even take the place of the check at the start of `FindValue`, requiring us to perform it only once for the entire tree instead of at each node.

Adding Nodes

We use the same basic algorithm to add values to a binary search tree as we do to search it. We start at the root node, progress down the tree as if searching for the new value, and terminate once we hit a dead end: either a leaf node or an internal node with a single child in the wrong direction. The primary difference between our search and insertion algorithms comes after we hit the dead end, when the insertion algorithm creates a new node as a child of the current node: a left-hand child if the new value is less than that of the current node or a right-hand child otherwise.

Here we can see a clear difference in behavior between trees that allow duplicates and ones that do not. If our tree allows duplicate values, we keep going until we hit a dead end and then insert a new copy of that value into our tree. If the tree doesn't allow duplicates, we might replace or augment the data stored at the matching node. For example, one simple piece of auxiliary data we could track is a counter—the number of times the value has been added to the tree. Below we focus on the case of overwriting data to stay consistent with other indexing data structures we will explore in the book.

As with our search function, we start with a wrapper function for addition that handles the case of empty trees:

```
InsertTreeNode(BinarySearchTree: tree, Type: new_value):
    IF tree.root == null:
        tree.root = TreeNode(new_value)
    ELSE:
        InsertNode(tree.root, new_value)
```

First, the code checks whether the tree is empty (`tree.root == null`). If so, it creates a new root node with that value. Otherwise, it calls `InsertNode` on the root node, kicking off the recursive process from below. Thus, we can ensure that `InsertNode` is called with a valid (non-null) node.

Here is the `InsertNode` code:

```
InsertNode(TreeNode: current, Type: new_value):
  ❶ IF new_value == current.value:
        Update node as needed
        return
  ❷ IF new_value < current.value:
      ❸ IF current.left != null:
            InsertNode(current.left, new_value)
        ELSE:
            current.left = TreeNode(new_value)
            current.left.parent = current
    ELSE:
      ❹ IF current.right != null:
            InsertNode(current.right, new_value)
        ELSE:
            current.right = TreeNode(new_value)
            current.right.parent = current
```

The `InsertNode` code starts by checking whether it is at a node with a matching value and, if so, updating the node's data as needed ❶. Otherwise, the code searches for the correct location to insert the new value by following either the left- or right-hand paths, based on the comparison of the new value and the current node's value ❷. In either case, the code then checks that the next node along that path exists ❸ ❹. If the next node exists, the code follows the path, progressing deeper into the tree. Otherwise, the code has found a dead end, indicating the correct location to insert the new node. The algorithm inserts nodes by creating a new node, linking the parent's corresponding child pointer (`left` or `right`), and setting the `parent` link.

For example, if we want to add the number 77 to the binary search tree in Figure 5-9, we progress down through nodes 50, 67, 81, and 78 until we hit a dead end at the node with value of 78. At this point, we find ourselves without a valid child in the correct direction. Our search is at a dead end. We create a new node with the value of 77 and make it the node 78's left child.

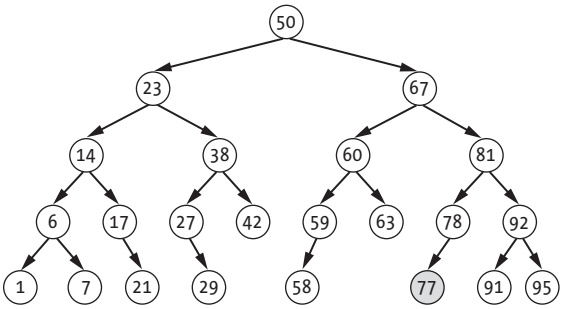

Figure 5-9: Inserting the value 77 into our binary search tree

The cost of inserting a new node into the tree is again proportional to the depth of the branch along which we insert the new node. We perform a single comparison for each node along the path until we hit a dead end, and, as with the search operation, we ignore all the nodes in other branches. Therefore, the worst-case cost of an insertion will scale linearly with the depth of the tree.

Removing Nodes

Removing nodes from a binary search tree is a more complicated process than adding them. There are three cases of node removal to consider: removing a leaf node (with no children), removing an internal node with a single child, and removing an internal node with two children. As you'd expect, the job becomes more complex as the number of children increases.

To remove a leaf node, we just delete that node and update its parent's child pointer to reflect the fact it no longer exists. This might make the parent node into a leaf. For example, to remove node 58 in Figure 5-10, we would just delete node 58 and set its parent's left child pointer to `null`.

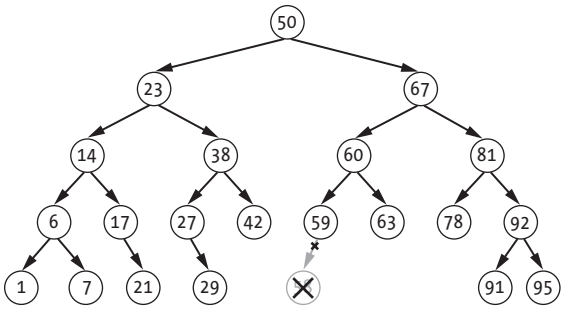

Figure 5-10: Remove a leaf node from a binary search tree by deleting it and updating the pointer from its parent node.

Removing leaf nodes shows the value of storing a pointer to the parent node: it allows us to search for the node to delete, follow the parent pointer back to that node's parent, and set the corresponding child pointer to `null`. Storing this single piece of additional data makes the deletion operation much simpler.

In the example of our public relations department, a leaf node that gets deleted corresponds to an employee with no direct reports leaving the company. After the farewell party and cake, the rest of the organization returns to work. The only change in the hierarchy is that the former employee's boss has one less person on their team. In fact, they might have no one reporting to them now.

If the target node has a single child, we remove it by promoting that single child to be the child of the deleted node's parent. This is like removing a manager from our reporting hierarchy without shuffling anyone else around. When the manager leaves, their boss assumes management of the former employee's single direct report. For example, if we wanted to remove node 17 from our example tree, we could simply shift node 21 up to take its place as shown in Figure 5-11. Node 14 now links directly to node 21.

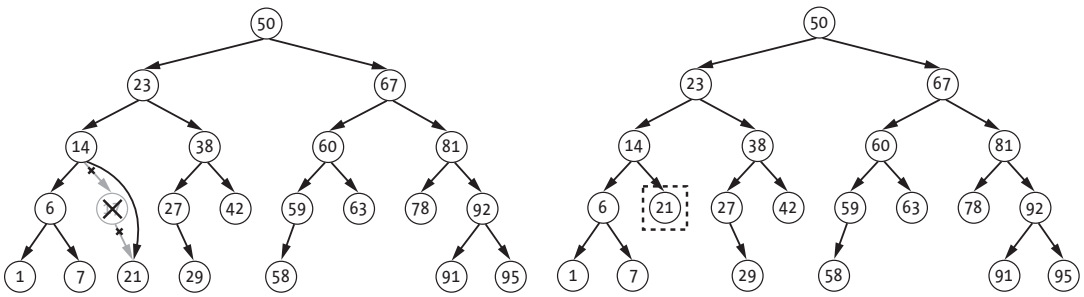

Figure 5-11: Remove an internal node with a single child by changing the pointers (left) and shifting that child up (right).

This way of removing a single-child node works even if the node we are shifting up has its own subtree. Since the node being moved up was already in the parent's subtree, all of its descendants will continue to respect the binary search tree property.

The complexity ramps up substantially when we try to remove an internal node with two children. It is no longer sufficient to just delete the node or shift a single child up. In our public relations department, a node's two children represent two distinct employees with different humor levels. We can't just choose one to promote and let the other accidentally disappear from the hierarchy, no longer anchored to the root node through the fragile chain of pointers. We must preserve the integrity of the rest of the tree and ensure it continues to follow the binary search tree property.

To remove a node with two children, we swap that node out for another node in the tree that will preserve the binary search tree property. We do this by finding the *successor* of the node to be deleted—the next node we would encounter if we traversed the nodes in sorted order. We swap the successor into the location of the deleted node. This swapped node might also have a child node that needs to be handled when it is removed from its old location. In order to remove the successor node from the binary tree without breaking any of its pointers, we reuse the delete procedure on the node to be swapped. We find the successor, save a pointer to that node, and then remove it from the tree.

For example, if we wanted to delete value 81 in Figure 5-12, we need to first swap in the node with a value of 91. We do this by saving pointers to the node to delete and the successor node (Figure 5-12(1)). Then we set the successor node to be the child of the deleted node's parent (Figure 5-12(2)). Finally, we update the successor node's children to those of the recently deleted node, effectively swapping it into place (Figure 5-12(3)).

In order to perform the deletion, we need to be able to efficiently find a node's successor. While this might seem like a daunting task, we have one critical advantage. Since we are only considering cases where the node in question has two children, we can always find the successor in the node's right-hand subtree. Specifically, the successor will be the minimum (or leftmost) node in the right-hand subtree. As a bonus, the successor node is guaranteed to have at most one (right-hand) child. If the candidate successor node had a child to the left, then that child (or a node down its own left-hand subtree) would be the true successor.

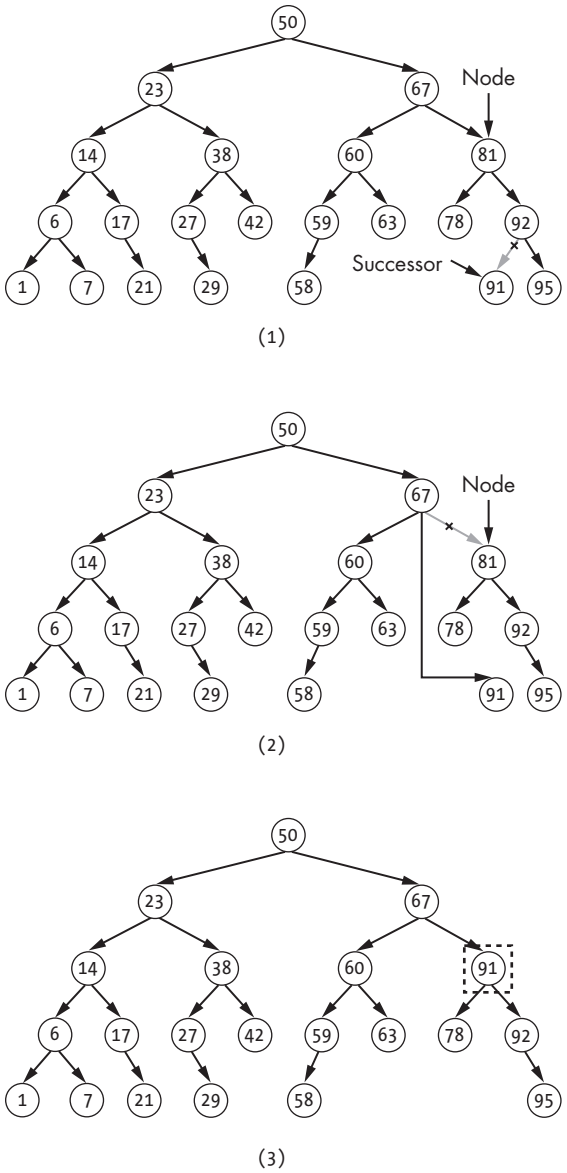

Figure 5-12: To remove an internal node with two children, we first swap the node's successor into that position.

Listing 5-1 provides (admittedly verbose) pseudocode to demonstrate removing the three types of nodes we've just discussed from a binary search tree. Shorter implementations are possible. However, breaking the cases out explicitly helps to illustrate the complexities involved. Also note that we delete a node using its pointer instead of its value. Thus, in order to delete the node with a given value, we would first find a pointer to the node using `FindTreeNode` and then call delete with that pointer.

```
RemoveTreeNode(BinarySearchTree: tree, TreeNode: node):
❶ IF tree.root == null OR node == null:
      return

  # Case A: Deleting a leaf node.
❷ IF node.left == null AND node.right == null:
      IF node.parent == null:
          tree.root = null
      ELSE IF node.parent.left == node:
          node.parent.left = null
      ELSE:
          node.parent.right = null
      return

  # Case B: Deleting a node with one child.
❸ IF node.left == null OR node.right == null:
    ❹ TreeNode: child = node.left
      IF node.left == null:
          child = node.right

    ❺ child.parent = node.parent
      IF node.parent == null:
          tree.root = child
      ELSE IF node.parent.left == node:
          node.parent.left = child
      ELSE:
          node.parent.right = child
      return

  # Case C: Deleting a node with two children.
  # Find the successor and splice it out of the tree.
❻ TreeNode: successor = node.right
  WHILE successor.left != null:
      successor = successor.left
  RemoveTreeNode(tree, successor)

  # Insert the successor in the deleted node's place.
❼ IF node.parent == null:
      tree.root = successor
  ELSE IF node.parent.left == node:
      node.parent.left = successor
  ELSE:
      node.parent.right = successor
❽ successor.parent = node.parent

❾ successor.left = node.left
  node.left.parent = successor

  successor.right = node.right
  IF node.right != null:
      node.right.parent = successor
```

Listing 5-1: Removal of a node from a binary search tree

As with the wrapper functions for insertion and search, the code starts by checking whether the tree is empty ❶ and, if so, returning `null`. It also checks whether there is a valid node to delete (`node != null`), which is useful in cases where we want to combine the search and deletion into a single line:

```
RemoveTreeNode(tree, FindTreeNode(tree, target))
```

Since `FindTreeNode` returns `null` if the node is not found, we handle this case explicitly.

The code then considers the three cases in order. In case A, where it is removing a leaf node ❷, the code only needs to change the correct child pointer of the removed node's parent. First, it checks whether the node to be deleted has a parent node. If not, the code is removing the root itself and modifies the root node pointer to `null`, effectively removing the root. If the removed node was the parent's left-hand child, the code sets that pointer to `null`. Likewise, if the removed node was the parent's right-hand child, the code sets that pointer to `null`. The code can then return, having successfully removed the target leaf node from the tree.

In case B, removing a node with a single child ❸, the code starts by identifying which of the node's two child pointers links to the child by checking which of the two pointers is not `null` ❹. The code stores a pointer to that child node for later use. Next, it fixes the pointers between the newly promoted node and its new parent ❺. The code sets the child's parent pointer to its previous grandparent, splicing the removed node out of the tree in the upward direction. Finally, the code fixes the correct child pointer within the removed node's parent, including handling changes to the root node as a special case. The code takes the pointer that previously pointed to the removed node and redirects it to point to that node's single child. If the removed node doesn't have a parent, the code is dealing with the root node and needs to modify that pointer accordingly. Once the code has spliced out the correct node, it returns.

In case C, where the node to be removed has two children, the code starts by identifying the successor node and removing that from the tree ❻. Note that, as described above, the recursive call to `RemoveTreeNode` cannot itself trigger case C because the successor will have at most a single (right-hand) child. The code maintains a pointer to this successor even after removing it from the tree because it will use this node to replace the deleted node. The code then replaces the deleted node with the successor through the following series of steps:

1. Modifying the deleted node's parent to set the correct child pointer to the successor ❼.
2. Modifying the successor's parent pointer to point to its new parent ❽.
3. Setting the links to and from the left and right children of the successor ❾. The code takes extra care when dealing with the right child, as it is possible that it has already deleted that child with the `RemoveTreeNode` operation above (in the case where the successor was the immediate right child of `node`). It therefore needs to check whether the right-hand child is `null` before trying to assign the right child's parent pointer.

Depending on the programming language and how the code will be used, we might also want to set `node`'s outgoing pointers to `null` as part of the deletion. This will clean up references from the deleted node to other nodes in the tree. We can do this by including the following lines at the end of each of the three cases (before the `return` statements in cases A and B and before the end of the function for case C):

```
node.parent = null
node.right = null
node.left = null
```

As with both search and insertion, the deletion operation involves at most traversing the tree from top to bottom along one path. In cases A and B, this trip happens before the `RemoveTreeNode` function (as part of the earlier call to `FindTreeNode` to get the pointer to the node itself). Case C adds an additional traversal from the internal node to be removed to its successor. Thus, the worst-case runtime of deletion is still proportional to the depth of the tree.

The Danger of Unbalanced Trees

The time it takes to perform searches, additions, and deletions on a *perfectly balanced* binary search tree is, in the worst case, proportional to the depth of the tree, making these operations highly efficient in trees that are not too deep. A perfectly balanced tree is one in which, at every node, the right subtree contains the same number of nodes as the left subtree. In this case, the depth of the tree grows by 1 each time we double the number of nodes in the tree. Thus, in balanced trees, the worst-case performance of all three operations grows proportionally to $\log_2(N)$, the logarithm of the number of elements N.

Binary search trees are still efficient as long as the trees are mostly, if not perfectly, balanced. But if the tree becomes highly unbalanced, its depth could grow linearly with the number of elements. In fact, in the extreme case, our splendid binary search tree becomes nothing more than a sorted linked list—all the nodes have a single child in the same direction as shown in Figure 5-13.

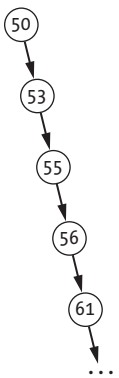

Figure 5-13: An example of an unbalanced binary search tree

Highly unbalanced trees can easily occur in many real-world applications. Imagine we are storing our coffee log in a binary search tree indexed by timestamp. Every time we drink a cup of coffee, we insert the relevant information into our tree. Things go bad quickly. Due to the monotonically increasing timestamps, we insert every entry in sorted order, and we create a linked list using only the right-hand child pointers.

Operations on an unbalanced tree can be extremely inefficient. Consider a tree with N nodes. If our tree is balanced, our operations take time logarithmic in N. In the opposite case, where our tree is a list, our operations can take time linearly proportional to N.

There are a variety of augmentations, such as red-black trees, 2-3 trees, and B-trees, that we can use to keep trees balanced while undergoing dynamic insertions and deletions. The tradeoff for any of these approaches is increased complexity in the tree operations. We consider B-trees in more detail in Chapter 12 and show how their structure keeps them balanced.

The next section introduces a straightforward approach to building a balanced binary search tree from an initial set of values. *Bulk construction* allows the algorithm to choose which nodes split the data so as to balance the number of nodes on each side. This is a good approach when we have many of the values up front but need to be careful with future insertions, as they could result in an unbalanced tree.

Bulk Construction of Binary Search Trees

We can easily construct a binary search tree by iteratively adding nodes: we create a single new node and label that our root, then for each remaining value, create a new node and add that node to the tree. This approach has the advantages of being simple and reusing the algorithms that we defined previously. Unfortunately, it can lead to unbalanced trees. As we saw above, if we add values in sorted order, we end up with a sorted linked list. We can do better when creating a tree from an initial set of numbers.

We create balanced binary search trees from a sorted array, such as the one shown in Figure 5-14, by recursively dividing the elements into smaller subsets. At each level, we choose the middle value to be the node at that level. If there is an even number of elements, we can use either of the two middle elements.

1	2	3	4	5	6	7	8	9	10	11	12

Figure 5-14: A sorted array used for bulk construction of a binary search tree

We create a new node with the value equal to the middle element in our array and split the remaining elements among the two child nodes, as shown in Figure 5-15. We recursively create subtrees for each of those child nodes using the same process. Values less than the middle element go on the left, while larger values go on the right.

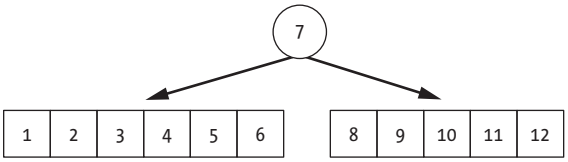

Figure 5-15: After the first split, we have a single node and two separate arrays.

We don't need to create new copies of the input array at each split. Instead, we can take a page from the binary search algorithm and just track the current range of the array under consideration, as shown in Figure 5-16. Each of our splits partitions the array into coherent halves, so we only need to worry about two bounds: the indices of the highest and lowest values that fall into the current branch.

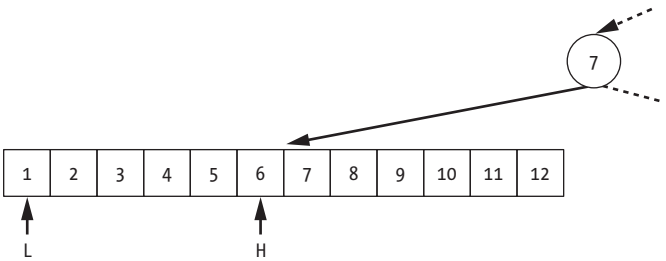

Figure 5-16: A high and low index can be used to track the subset of the array currently under consideration.

Once we've created the new node, we use the same approach to build each of the left and right subtrees independently. Specifically, we select the middle value, create a new node from that value, partition the remaining ranges, and use those ranges to create the subtrees. The process ends when there is only a single value left in our range. In that case, we create a new leaf node with that value and no children.

Why This Matters

Binary search trees demonstrate how we can adapt dynamic data structures to specific problems. The trees use a branching structure to capture and maintain ordering information in the data values. This allows them to facilitate efficient searches. Further, the pointer-based structure of binary search trees allows them to continuously adapt as new data is added. This interplay of data, problem, and computation provides the foundation that allows us to solve increasingly complex computational challenges.

Throughout later chapters, we will continue to build on the concepts of dynamic data structures, adapting the structure of the data to the problem itself and using branching data structures to allow efficient pruning. These techniques pop up in a range of different data structures. Understanding

the fundamentals, such as how these techniques allow efficient searches in dynamic binary search trees, is critical to understanding how to work with more advanced data structures and algorithms. The next chapter introduces the trie, showing how the tree-based concepts of the binary search tree can be extended to multiway branches in order to improve the efficiency of certain types of searches.