Binary Trees

A binary tree is a rooted tree where every node has at most two children. When a node has only one child, we still distinguish whether this is the left child or the right child of the parent. We already met binary trees in the form of rank trees. A binary heap is also a binary tree (but we used an array to implement this tree).

We will now see how to use a binary tree to implement the set and map (dictionary) ADTs. Since we can think of a set as a special case of a map that maps elements of the set to `True`, in the following we will only discuss the map ADT (called `dict` in Python).

The two basic method of `dict` are `__contains__` and `__getitem__`. The method `__contains__(k)` determines if the key k is present in the dictionary, the method `__getitem__(k)` returns the value stored with the key k (and raises an exception if k is not present in the map). The `__setitem__(k, v)` method adds a new mapping for the given key and value. The `__delitem__(k)` method removes the mapping for the key k. We add methods `firstkey()` and `lastkey()` that return the smallest and largest key present in the `dict` (this is not possible with standard Python dictionaries).

Binary Search Trees

A *binary search tree* is a binary tree where every node stores one key (and the value that corresponds to this key), and that has the *binary search tree property*: For any node v, all the keys in the left subtree of v are less than the key of v, and all the keys in the right subtree of v are larger than the key of v.

As a result, an *in-order traversal* of a binary search tree returns the keys in sorted order.

Here is the node class of our binary search tree:

```
class _Node():
  def __init__(self, key, value, left=None, right=None):
    self.key = key
    self.value = value
    self.left = left
    self.right = right
```

Finding keys. To implement `__contains__(k)` and `__getitem__(k)`, we need to find the node containing a given key k. We start at the root and compare k to the key s of the root node. If $k = s$, then the node we are looking for is the root node itself. If $k < s$ then by the search tree property, we know that k can only be in the left subtree of the root. If $k > s$, then k can only be in the right subtree of the root. We proceed in this manner until we either find a node with key k, or we reach an empty subtree:

```
  def _find(self, key):
    if key == self.key:
      return self
    if key < self.key:
      return self.left._find(key) if self.left else None
    else:
      return self.right._find(key) if self.right else None
```

Smallest and largest keys. To implement `firstkey()`, we need to find the smallest key in the search tree. For every node, its left subtree contains all the keys smaller than the key in the node itself, so the smallest key in the tree can be found by always walking down to the left child until there is no more left child:

```
  def _find_first(self):
    p = self
    while p.left is not None:
```

```
    p = p.left
  return p
```

You may recognize this from method `findfirst()` in our rank tree. Implementing `lastkey()` works the same, walking down to the right child.

Inserting a key. To add a mapping (k, v), we first need to find the correct place for the key k. Since we need to maintain the search tree property, this needs to be done exactly as in `_find(k)`.

If we find a node n that already contains the key k, then we update the tree by changing the value at n to v. Otherwise, we reach an empty subtree, and we replace this empty subtree by a single node with key k and value v.

```
def _insert(self, key, value):
  if key == self.key:
    self.value = value
  elif key < self.key:
    if self.left is None:
      self.left = _Node(key, value)
    else:
      self.left._insert(key, value)
  else:
    if self.right is None:
      self.right = _Node(key, value)
    else:
      self.right._insert(key, value)
```

Removing a key. Removing a mapping is the hardest part. Fortunately, we already learnt how to remove a node from a binary tree when we implemented rank trees, and we can use the exact same method here:

We first find the node n containing the key k to be deleted. If n is a leaf or has only one child, we can remove the node n easily, by changing the link from n's parent. If n has two children, we do not remove the node n. Instead, we find the leftmost node u in the right subtree of n. The key of u is the smallest key in the tree that is larger than k. That means that we can move the key and value information from u to n, and delete the node u. Since u has no left child, this is again easy.

The `_Node`'s internal method `_remove(k)` returns the root of a new subtree that is identical to the subtree rooted at n, except that the mapping for k has been removed:

```
def _remove(self, key):
  if key < self.key and self.left is not None:
    self.left = self.left._remove(key)
  elif key > self.key and self.right is not None:
    self.right = self.right._remove(key)
  elif key == self.key:
    if self.left is not None and self.right is not None:
      # Need to remove self, but has two children
      n = self.right._find_first()
      self.key = n.key
      self.value = n.value
      self.right = self.right._remove_first()
    else:
      # Need to remove self, which has zero or one child
      return self.left if self.left else self.right
  return self
```

`_remove` makes use of another internal method `_remove_first(n)`, which again returns a new subtree, identical to the subtree at n except that the leftmost node has been removed:

```
def _remove_first(self):
  if self.left is None:
    return self.right
  else:
    self.left = self.left._remove_first()
    return self
```

Analysis Finding, inserting, and removing nodes from a binary search tree takes time $O(h)$, where h is the current height of the tree. In a perfectly balanced binary tree we would have $h = \log n$. Unfortunately, it is quite easy to make very unbalanced trees. For instance, this code

```
import bst
t = bst.dict()
for i in range(n):
  t[i] = i+1
```

creates a tree of height $n - 1$. Essentially, this tree is a linked list: nodes have no left child, only a right child.

It can be shown that binary search trees offer $O(\log n)$ performance on insertions of randomly chosen or randomly ordered keys with high probability. As long as the trees remain "reasonably" well-balanced, search operations will run in $O(\log n)$ time.

In practice, you may need to resort to experiments to determine whether your particular application uses binary search trees in a way that tends to generate somewhat balanced trees or not.

Unfortunately, there are occasions where you might fill a tree with entries that happen to be already sorted. In this circumstance, a disastrously imbalanced tree will be the result. Technically, all operations on binary search trees have $O(n)$ worst-case running time.

For this reason, researchers have developed a variety of algorithms for keeping search trees balanced. "Balanced" means that the left and right subtree of every node should have roughly equal "size". Many different kinds of balanced search trees have been developed—we will discuss AVL-trees, 2-3-4 trees, and red-black trees.

AVL Trees

AVL trees are *height-balanced* trees, and are the oldest balanced search trees (1962). They are named after their inventors, Georgy Adelson-Velsky and Evgenii Landis. An AVL-tree is a binary search tree with an additional *balance property*: For every node of the tree, the *height* of the left subtree and the right subtree differ by at most one. Here, we count the height of the empty subtree as -1.

Height of AVL-trees. We first show that AVL-trees have height $O(\log n)$. More precisely, we ask the opposite question: For a given height h, what is the *smallest* number of nodes $N(h)$ that an AVL-tree with height h can have?

A bit of drawing quickly shows that $N(0) = 1$, $N(1) = 2$, $N(2) = 4$, and $N(3) = 7$.

For $h \geqslant 2$, we have $N(h) = N(h-1) + N(h-2) + 1$. Indeed, consider an AVL-tree of height h. One of its subtrees, let's say the left subtree, must be an AVL-tree of height $h-1$, and so it has at least $N(h-1)$ nodes. The right subtree must have either height $h-1$ or height $h-2$, since otherwise the balance property would be violated at the root. It follows that the right subtree has at least $N(h-2)$ nodes, so the tree has at least $N(h-1) + N(h-2) + 1$ nodes. This shows that $N(h) \geqslant N(h-1) + N(h-2) + 1$.

We actually have equality in this recursive formula, because we can take a smallest possible AVL-tree of height $h-1$, a smallest possible AVL-tree of heigth $h-2$, and merge them into an AVL-tree of height h with $N(h-1) + N(h-2) + 1$ nodes. The trees constructed this way are called *Fibonacci-trees*.

Since clearly $N(h-1) \geqslant N(h-2)$, we have $N(h) \geqslant 2N(h-2)$, and we immediately get that $N(h) \geqslant 2^{\lceil h/2 \rceil}$. If we know that a tree has n nodes and height h, then this implies that $n \geqslant N(h) \geqslant 2^{\lceil h/2 \rceil}$, and so $\log n \geqslant h/2$, and so $h \leqslant 2 \log n$.

We can get a better bound by using the Fibonacci numbers. Remember that $F_0 = 0$, $F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$ for $i \geqslant 2$. We now prove by induction that $N(h) = F_{h+3} - 1$. This is true for $h = 0$ and $h = 1$, so consider $h \geqslant 2$. Then $N(h) = N(h-1) + N(h-2) + 1 = F_{h+2} - 1 + F_{h+1} - 1 + 1 = F_{h+3} - 1$.

Luckily for us, there is a formula for the Fibonacci numbers. We have $F_i = \varphi^i/\sqrt{5} + (1-\varphi)^i/\sqrt{5}$, where φ is the golden ratio $\varphi \approx 1.618034$. The second term is very small, so we have $N(h) = F_{h+3} - 1 \approx \varphi^{h+3}/\sqrt{5} - 1$. It follows that for a tree with n nodes we have $h + 3 \leqslant \log_\varphi(\sqrt{5}(n+1))$ and so $h \leqslant 1.4405 \log(n+1)$.

Node objects. To implement an AVL-tree, we store with each node v the height of the subtree rooted at v:

```
class _Node():
  def __init__(self, key, value, left=None, right=None, height=0):
    self.key = key
    self.value = value
    self.left = left
    self.right = right
    self.height = height

  def _left_height(self):
    return -1 if self.left is None else self.left.height

  def _right_height(self):
    return -1 if self.right is None else self.right.height

  def _recompute_height(self):
    left = self._left_height()
    right = self._right_height()
    self.height = max(left, right) + 1
    return abs(right - left) > 1
```

The methods `_left_height()` and `_right_height()` return the heights of the two subtrees, and correctly handle the case of an empty subtree, which we define to have height -1. The method `_recompute_height()` recomputes the height of a node from the heights of its two subtrees. It also returns `True` if the node is unbalanced, that is, if the height difference is larger than one.

Finding keys, smallest and largest keys. An AVL-tree *is* a binary search tree: It fulfills the search tree property. Therefore, all BST methods that do not modify the tree work unchanged on AVL trees: `_find(k)`, `_find_first()`, and `_find_last()`.

The big difference is: Since we have proven above that the height of an AVL-tree with n nodes is $O(\log n)$, we have a *guarantee* on the running time of these methods now. They take time $O(\log n)$ on an AVL tree that stores n elements.

Maintaining balance. To add a mapping (k, v), we first follow the path through the tree as in `_find(k)`. If we find a node n containing key k, then we simply update the tree by changing the value at n to v. Otherwise, we reach an empty subtree, and we replace this empty subtree by a single node w with key k and value v. The height of all subtrees that include w may change, and as a consequence the tree may become unbalanced.

When we remove a mapping for k, we again first find the node as in `_find(k)`. If the node is a leaf or has only one child, we can delete it easily, otherwise we instead delete the leftmost node in its right subtree. Let w be the node that was deleted. Again, the height of all subtrees that include w may change.

In both cases, we observe first that the nodes whose subtree includes w are exactly the ancestors of w. These ancestors form a path from the root to w, and some of them may have become unbalanced because of the insertion or deletion of w.

Let z be the *lowest* ancestor of w that is now unbalanced. Let y be the child of z with larger height (the two children cannot have equal height since z is unbalanced). Let x be the child of y with larger height (if both children of y have equal height, we chose x to be the right child of y if y is the right child of z, and chose x to be the left child of y if y is the left child of z).

We will now *restructure* the subtree with root z. We distinguish four cases, depending on the ordering of the keys stored at x, y, and z. Fig. 1 shows the four cases. If y is the right child of z, we are in the left

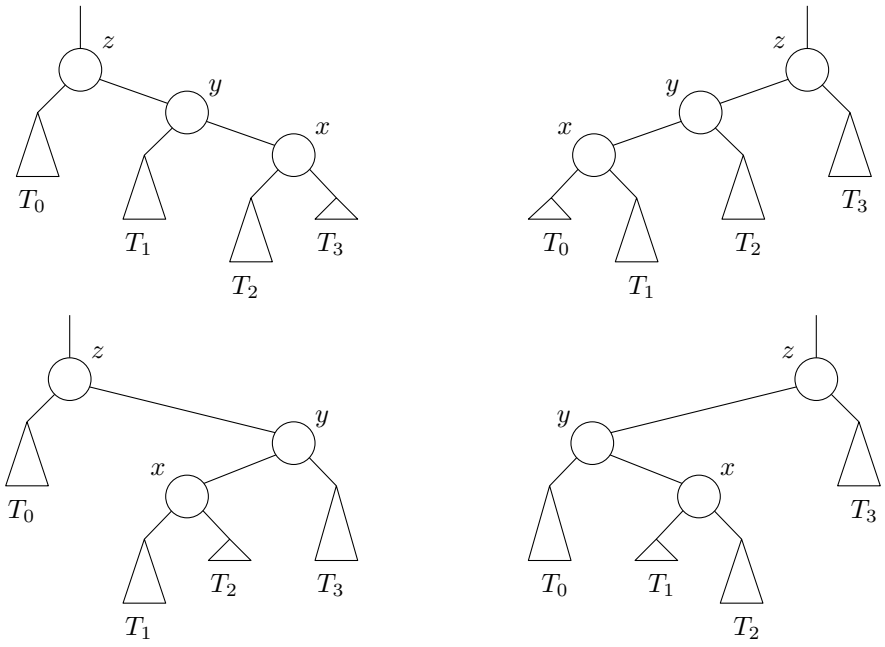

Figure 1: Four cases for restructuring.

column of Fig. 1. If the key of x lies inbetween the keys of y and z, we are in the bottom row, otherwise in the top row.

Let T_0, T_1, T_2, and T_3 be the four subtrees of z, y, and x, as in Fig. 1. By our choice of z, all four trees are balanced. Let $h(T)$ denote the height of a subtree T, and let $h(v)$ denote the height of the subtree with root v (before restructuring). Recall that by our choice of z, the nodes x and y are balanced, while z is unbalanced (before restructuring).

Left rotation. Let us first consider the case where y is the right child of z, and x is the right child of y (top left in Fig. 1). We restructure the subtree with root z by making y its new root, and moving z into the left subtree of y as in Fig. 2. This operation is called a *left rotation* about z.

We first argue that a left rotation maintains the search tree property of the tree. Let us write $T_0 \prec z$ to mean that all keys in T_0 are less than the key of z. Since the search tree property held before the left rotation, we have

$$T_0 \prec z \prec T_1 \prec y \prec T_2 \prec x \prec T_3.$$

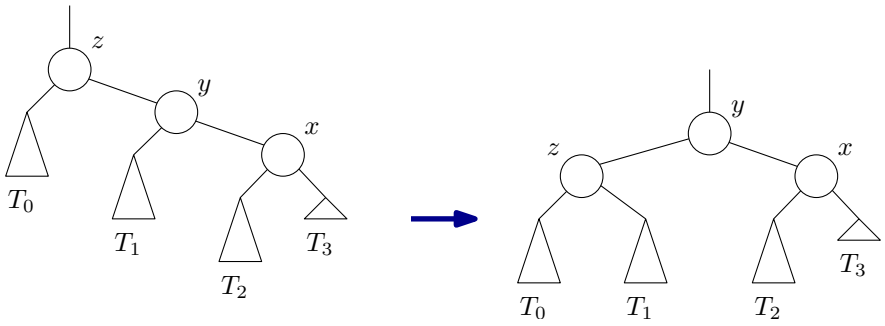

Figure 2: Left rotation about z.

This implies that the search tree property holds *after* the left rotation. For instance, all keys in the subtree formed by T_0, T_1, and z are less than the key of y.

We now argue that after the left rotation the subtree (with new root y) is balanced. Without loss of generality, let's assume that $h(T_2) \geqslant h(T_3)$. Since x is balanced, that means $h(T_2) \geqslant h(T_3) \geqslant h(T_2) - 1$. We have $h(x) = h(T_2) + 1$. Since y is balanced, and we have chosen x to be the child of y with larger height, we have $h(T_2) + 1 = h(x) \geqslant h(T_1) \geqslant h(x) - 1 = h(T_2)$. The node z, however, is unbalanced, and so $h(T_0) = h(y) - 2 = h(T_2)$.

After the left rotation, node x is still balanced, as its subtrees have not changed. Node z is now balanced, since we have

$$h(T_0) + 1 = h(T_2) + 1 \geqslant h(T_1) \geqslant h(T_2) = h(T_0.)$$

Finally, node y is balanced since $h(z) = h(T_1) + 1$ and $h(x) = h(T_2) + 1$, and so $h(x) + 1 \geqslant h(z) \geqslant h(x)$.

In this way, we have re-established the balance property for this subtree. There may still be ancestors of w (and therefore ancestors of the new node y) that are unbalanced, so we have to repeat the procedure there.

Right rotation. The case where y is the left child of z and x is the left child of y is entirely symmetric. We perform a *right rotation about* z. Fig. 3 shows an example of an insertion that causes a right rotation.

Double rotations. Let's now consider the case where y is the right child of z and x is the left child of y (bottom left in Fig. 1). In this case, we first perform a right rotation about y, and then a left rotation about z, see Fig. 4. We call this operation a (right-left) *double rotation*.

Again, we first need to check that the resulting tree has the search tree property. This is true because

$$T_0 \prec z \prec T_1 \prec x \prec T_2 \prec y \prec T_3.$$

It remains to check the balance condition on the new subtree (with root x). Without loss of generality, let's assume that $h(T_1) \geqslant h(T_2)$. Since x is balanced, that means $h(T_1) \geqslant h(T_2) \geqslant h(T_1) - 1$ and we have $h(x) = h(T_1) + 1$. Since y is balanced, and we have chosen x to be the child of y with strictly larger height in this case, we have $h(T_3) = h(x) - 1 = h(T_1)$. Since z, is unbalanced, we have $h(T_0) = h(y) - 2 = h(T_1)$. To summarize, we have

$$h(T_0) = h(T_1) = h(T_3) \geqslant h(T_2) \geqslant h(T_1) - 1.$$

In the new tree, z is balanced since $h(T_0) = h(T_1)$. Similarly, y is balanced since $h(T_3) \geqslant h(T_2) \geqslant h(T_3) - 1$. The new root x is balanced since $h(z) = h(T_1) + 1 = h(T_3) + 1 = h(y)$.

Finally, the case where y is the left child of z and x is the right child of y completely symmetric, and we perform a left-right double rotation.

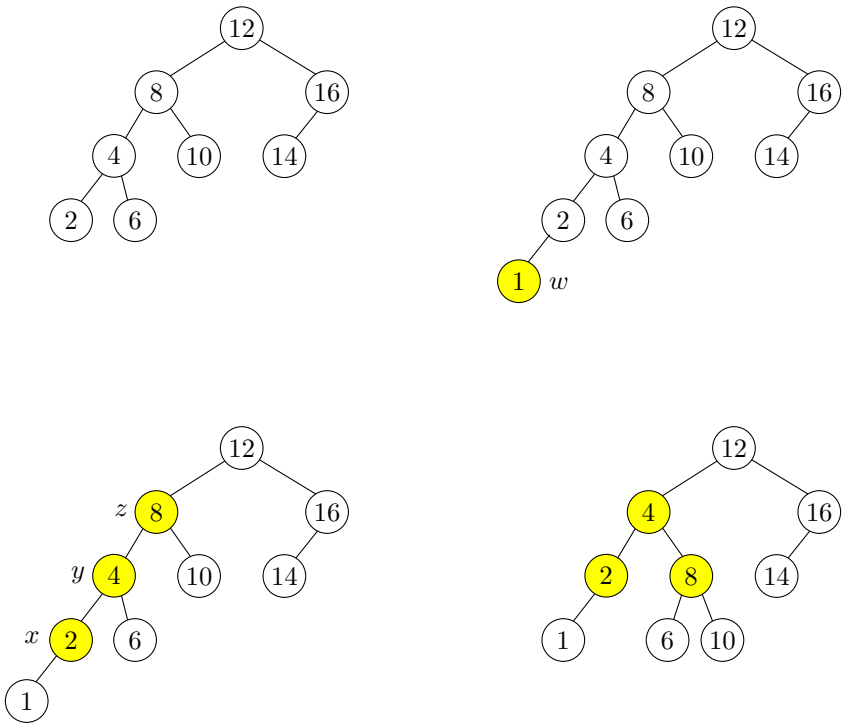

Figure 3: Insertion of 1.

Implementation. We start by two internal methods that implement left-rotations and right-rotations. `_rotate_left(t)` rotates about node t, and returns the new root of the subtree (which is the right child of t). The method already recomputes the height of all nodes whose height has changed:

```
def _rotate_left(self):
  root = self.right
  self.right = root.left
  self._recompute_height()
  root.left = self
  root._recompute_height()
  return root
```

The method `_rotate_right(t)` is exactly symmetric:

```
def _rotate_right(self):
  root = self.left
  self.left = root.right
  self._recompute_height()
  root.right = self
  root._recompute_height()
  return root
```

The following internal method implements the restructuring operation. z is the root of the subtree to be restructured. The method uses the heights of the children to determine which of the four cases we are in:

```
def _restructure(self):
  if self._right_height() > self._left_height():
    if self.right._left_height() > self.right._right_height():
```

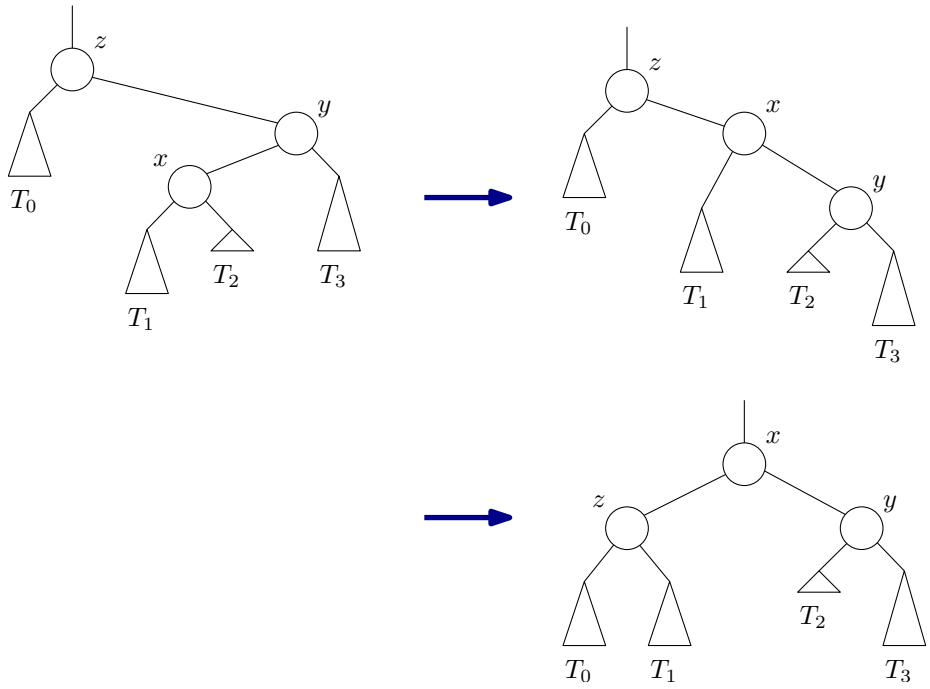

Figure 4: Double rotation

```
      self.right = self.right._rotate_right()
    return self._rotate_left()
  else:  # left height > right height
    if self.left._right_height() > self.left._left_height():
      self.left = self.left._rotate_left()
    return self._rotate_right()
```

Now we have all the pieces to implement the insert function. `t._insert(k, v)` adds the mapping $k \to v$ to the subtree rooted at t, and returns the new root of the subtree, guaranteed to be an AVL-tree again:

```
def _insert(self, key, value):
  if key == self.key:
    self.value = value
    return self
  if key < self.key:
    if self.left is None:
      self.left = _Node(key, value)
    else:
      self.left = self.left._insert(key, value)
  else:
    if self.right is None:
      self.right = _Node(key, value)
    else:
      self.right = self.right._insert(key, value)
  if self._recompute_height():
    return self._restructure()
  return self
```

Remember that after inserting the node w we have to walk up again in the tree until we find the first node

that is now unbalanced. The method `_insert` implements the "walking up" using recursion: It inserts the mapping into the correct subtree, then updates the height of the current node t, and restructures if necessary. A more careful analysis of the restructuring cases shows that in insertions, this can actually happen for one ancestor only (that is, after one restructuring operation the entire tree is balanced). We do not discuss this refined analysis, as our code does not use this fact.

It remains to discuss the remove function. The following internal function `t._remove_first()` removes the leftmost node (smallest key) from the subtree with root t, and returns the root of the new subtree. The returned tree is already balanced.

```
def _remove_first(self):
  if self.left is None:
    return self.right
  self.left = self.left._remove_first()
  if self._recompute_height():
    return self._restructure()
  return self
```

Finally, the following function removes the mapping for key k from the subtree, and returns a new, balanced subtree:

```
def _remove(self, key):
  if key < self.key and self.left is not None:
    self.left = self.left._remove(key)
  elif key > self.key and self.right is not None:
    self.right = self.right._remove(key)
  elif key == self.key:
    if self.left is not None and self.right is not None:
      # Need to remove self, but has two children
      n = self.right._find_first()
      self.key = n.key
      self.value = n.value
      self.right = self.right._remove_first()
    else:
      # Need to remove self, which has zero or one child
      # No restructuring needed in this case
      return self.left if self.left else self.right
  if self._recompute_height():
    return self._restructure()
  return self
```

Minimizing memory use. We use a integer field to store the height of each node. That wastes some memory—since AVL-trees are balanced, the height cannot be more than, say, 100, which would easily fit into one byte. In fact, it turns out that AVL-trees can be implemented using only *two bits* per node: Instead of storing the height in each node, we store the difference of the heights of the two subtrees. By the balance condition, this difference is either -1, 0, or $+1$, and can be stored in two bits. We leave it as an exercise to make such an implementation.

2-3-4 trees

A 2-3-4 tree is a perfectly balanced tree, where *all leaves* are on the bottom level. 2-3-4 trees are thus named because every node has 2, 3, or 4 children, except leaves. Each node stores 1, 2, or 3 keys (and the corresponding values), which determine how other entries are distributed among its children's subtrees.

Each internal (non-leaf) node has one more child than keys. For example, a node with keys $[20, 40, 50]$ has four children. Eack key k in the subtree rooted at the first child satisfies $k \leqslant 20$; at the second child, $20 \leqslant k \leqslant 40$; at the third child, $40 \leqslant k \leqslant 50$; and at the fourth child, $k \geqslant 50$.

2-3-4 trees are a type of B-tree, which you may learn about someday in connection with fast disk access for database systems.

Finding a key. Finding an entry is straightforward. Start at the root. At each node, check for the key k; if it's not present, move down to the appropriate child chosen by comparing k against the keys. Continue until k is found, or k is not found at a leaf node.

By the way, you can define an inorder traversal on 2-3-4 trees analogous to that on binary trees, and it visits the keys in sorted order.

Adding a mapping (bottom-up). To add the mapping $k \to v$, we first walk down the tree in search of the key k. If we find a node with key k, we update the mapping and are done.

If k is not yet in the tree, we find a leaf node v where k needs to be inserted. If v has at most 2 keys, we simply insert k and are done.

Otherwise, we say that v *overflows*. In this case we split the node v into two nodes v_1 and v_2, with two keys each (including the new key k). To be able to add both as children to v's parent p, we need to move one of the four keys up to p, increasing the number of keys in p. As a result, p may overflow, and we have to repeat the process.

The process ends when no overflow happens, or if we reach the root node. If the root node overflows, we simply create a new root, and the height of the tree increases by one.

Adding a mapping (top-down). The bottom-up method has the disadvantage that we need to walk down and up again in the tree. It is possible to avoid the second walk up, using *top-down* insertions. These can be implemented without recursion using a simple loop.

As we walk down the tree, we sometimes *modify* nodes of the tree. Specifically, whenever we encounter a 3-key node, the middle key is ejected and is placed in the parent node instead. Since the parent was previously treated the same way, the parent has at most two keys, and always has room for a third. The other two keys in the 3-key node are split into two separate 1-key nodes, which are divided underneath the old middle key.

The reasons why we split every 3-key node we encounter (and move its middle key up one level) are (1) to make sure there's room for the new key in the leaf node, and (2) to make sure that above the leaves, there's room for any key that gets kicked upstairs. Sometimes, an insertion operation increases the height of the tree by one by creating a new root.

Removing a key (top-down). Removing a key from a 2-3-4-tree is similar to removing a key in binary trees: you find the entry you want to remove (having key k). If it's in a leaf, you remove it. If it's in an internal node, you replace it with the entry with the next higher key. That entry is always in a leaf. In either case, you remove an entry from a leaf in the end.

Like insertions, deletions change nodes of the tree as we walk down. Whereas insertions eliminate 3-key nodes (moving nodes up the tree) to make room for new keys, deletions eliminate 1-key nodes (pulling keys down the tree) so that a key can be removed from a leaf without leaving it empty. There are three ways 1-key nodes (except the root) are eliminated.

Stealing. When we encounter a 1-key node (except the root), we try to steal a key from an adjacent sibling. But we can't just steal the sibling's key without violating the search tree invariant. We move a key from the sibling to the parent, and we move a key from the parent to the 1-key node. We also move a subtree S from the sibling to the 1-key node (now a 2-key node).

Note that we can't steal a key from a non-adjacent sibling.

Fusion. If no adjacent sibling has more than one key, we can't steal. In this case, the 1-key node steals a key from its parent. Since the parent was previously treated the same way (unless it's the root), it has at least two keys, and can spare one. The sibling is also absorbed, and the 1-key node becomes a 3-key node.

If the parent is the root and contains only one key, then the current 1-key node, its 1-key sibling, and the 1-key root are fused into one 3-key node that serves as the new root. The height of the tree decreases by one.

Eventually we reach a leaf. After processing the leaf, it has at least two keys (if there are at least two keys in the tree), so we can delete the key and still have one key in the leaf.

Analysis. A 2-3-4 tree with height h has between 2^h and 4^h leaves. If n is the total number of nodes (including internal nodes), then $n \geqslant 2^{h+1} - 1$. By taking the logarithm of both sides, we find that $h = O(\log n)$.

The time spent visiting a 2-3-4 node is typically longer than in a binary search tree (because the nodes and the rotation and fusion operations are complicated), but the time per node is still in $O(1)$.

The number of nodes visited is proportional to the height of the tree. Hence, the running times of searching, insertions, and deletions are in $O(h)$ and hence in $O(\log n)$, even in the worst case.

Red-black trees

The disadvantage of 2-3-4-trees is that managing nodes becomes rather complex, and depends on the number of elements stored in the node. A red-black tree is an implementation of 2-3-4-trees as a normal binary search tree, with the only addition of one bit per node, used to mark a node as either black or red.

To convert a 2-3-4-tree to a red-black tree, we replace each node w of the 2-3-4-tree by a small binary search tree T_w. If w contains one element, then T_w is simply one black node. If w contains two elements, then T_w consists of two nodes, a black root and a red child (note that there are two possible ways of doing this). If w contains three elements, then T_w consists of three nodes, a black root with two red children.

We observe that the resulting tree has the following properties:

- The root is black.
- The parent of a red node is black.
- Every path from the root to an empty subtree contains the same number of black nodes.

The last property follows from the fact that the number of black nodes on the path is the number of 2-3-4-nodes that the path passes through, and all the leaves in a 2-3-4-tree are on the same level.

We can define a red-black-tree as a binary search tree where each node is either red or black and which satisfies the three conditions above. It is easy to see that every such tree can be converted to an equivalent 2-3-4-tree. It follows that the height of a red-black tree is at most $2 \log n$.

Searching a red-black tree is easy, since it is a binary search tree. To implement insertions and deletions, we imitate the operations for 2-3-4-trees.

Bottom-up insertions. We first find the empty subtree where the new key belongs, and replace it by a red node w. This corresponds to inserting the key into a leaf of the 2-3-4-tree.

If w's parent is black, we are already done (the 2-3-4-node has enough space to store the new element). Otherwise, we first look at the *uncle* u of w. (The uncle is the sibling of w's parent.)

If the uncle is black, then we can fix the problem by a rotation. (This is the case where the 2-3-4-node had enough space for a new element, but we have to restructure its subtree to keep the correct representation.)

If the uncle is red, this means that the 2-3-4-node has overflowed, and we need to split it. We do this by changing the color of w's parent and w's uncle to black, and the color of w's grandparent v to red. In other words, the node v is "kicked up" into the parent 2-3-4-node. Of course this could mean that this node now overflows, and so we have to repeat the process, with v taking the role of w. Note that if v is the root, then we immediately change its color back to black, and we are done.

This algorithm can be explained without mentioning 2-3-4-trees at all. One can show its correctness simply by arguing that it maintains the three properties of red-black-trees.

Top-down insertions and deletions. Like for 2-3-4-trees, insertions and deletions can be implemented by walking through the tree from the root to an empty subtree. When a black node v has two red children, we immediately split it, by recoloring v red and its children black. If v's parent is red, its uncle must be black, and so we can use a rotation to fix the red-black-property. If v is the root, we immediately recolor it black.

To implement deletions, we can again imitate the algorithm for 2-3-4-trees. We leave this as an exercise for the reader.

Acknowledgments

These notes are based on the notes written by Jonathan Shewchuk for his course at UC Berkeley. All mistakes are mine, of course.