executes.

Even though we use the family metaphor for many of the naming conventions, a family tree is not actually a tree. The reason is that these violate the one parent rule.

When we draw trees, we take an Australian convention of putting the root on the top and the children below. Although this is backwards with respect to the trees we see in nature, it does correspond to how we generally think of most other hierarchies. Thankfully, there will be sufficiently little conceptual overlap between your experience of trees in the park and the trees in this book, that the metaphoric breakdown should not be confusing.

16.1 Some more definitions

A **path** in a tree is a sequence of nodes in which each node is a child of the previous node. We say it is a path from x to y if the first node is x and the last node is y. There exists a path from the root to every node in the tree. The **length of the path** is the number of hops or edges which is one less than the number of nodes in the path. The **descendants** of a node x are all those nodes y for which there is a path from x to y. If y is a descendant of x, then we say x is an **ancestor** of y. Given a tree T and a node n in that tree, the **subtree rooted at** n is the tree whose root is n that contains all descendants of n.

The **depth** of a node is the length of the path to the node from the root. The **height** of a tree is the maximum depth of any node in the tree.

This is a lot of definitions to ingest all at once. It will help to draw some examples and identify check that you can apply the definitions to the examples, i.e. identify the parent, children, descendants, and ancestors of a node.

16.2 A recursive view of trees

A tree can be defined recursively as a root with zero or more children, each of which is tree. Also, the root of a tree stores some data. This will be convenient at first and will be important later on as we think about doing more elaborate algorithms on trees, many of which are easiest to describe recursively. Be warned however, that our object-oriented instincts will later kick in as we separate the notions of trees and nodes. Having two different

kinds of things will lead us to use two different classes to represent them (but not yet).

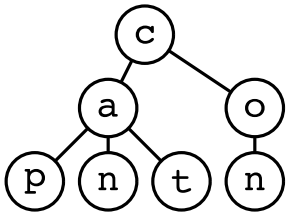
We can use lists to represent a hierarchical structure by making lists of lists. To make this a little more useful, we will also store some data in each node of the tree. We'll use the convention that a list is a tree in which the first item is the data and the remaining items are the children (each should be a list).

Here is a simple example.

```
['a', ['p'], ['n'], ['t']]
```

This is a tree with `'a'` stored in the root. The root has 3 children storing respectively, `'p'`, `'n'`, and `'t'`. Here is a slightly bigger example that contains the previous example as a subtree.

```
T = ['c', ['a', ['p'], ['n'], ['t']], ['o', ['n']]]
```



We can print all the nodes in such a tree with the following code.

```
def printtree(T):
    print(T[0])
    for child in range(1, len(T)):
        printtree(T[child])

printtree(T)
```

```
c
a
p
n
t
o
n
```

Whenever I see code like that above, I want to replace it with something that can work with any iterable collection. This would involve getting rid of all the indices and using an iterator. Here's the equivalent code that uses an iterator.

```
def printtree(T):
    iterator = iter(T)
    print(next(iterator))
    for child in iterator:
        printtree(child)

printtree(T)
```

```
c
a
p
n
t
o
n
```

Here we use the iterator to extract the first list item, then loop through the rest of the children. So, when we get to the `for` loop, the iterator has already yielded the first value and it will start with the second, i.e., the first child.

16.3 A Tree ADT

The information in the tree is all present in the *list of lists* structure. However, it can be cumbersome to read, write, and work with. We will package it into a class that allows us to write code that is as close as possible to how we think about and talk about trees. As always, we will start with an ADT that describes our expectations for the data structure and its usage. The **Tree ADT** is as follows.

- `__init__(L)` : Initialize a new tree given a list of lists. The convention is that the first element in the list is the data and the later elements (if they exist) are the children.
- `height()` : Return the height of the tree.

- `__str__()` : Return a string representing the entire tree.
- `__eq__(other)` : Return `True` if the tree is equal to `other`. This means that they have the same data and their children are equal (and in the same order).
- `__contains__(k)` : Return `True` if and only if the tree contains the data `k` either at the root or at one of its descendants. Return `False` otherwise.
- `preorder()` Return an iterator over the data in the tree that yields values according to the **preorder** traversal of the tree.
- `postorder()` : Return an iterator over the data in the tree that yields values according to the **postorder** traversal of the tree.
- `__iter__()` : An alias for preorder.
- `layerorder()` : Return an iterator over the data in the tree that yields values according to the **layer order** traversal of the tree.

16.4 An implementation

```
class Tree:
    def __init__(self, L):
        iterator = iter(L)
        self.data = next(iterator)
        self.children = [Tree(c) for c in iterator]
```

The initializer takes a *list of lists* representation of a tree as input. A `Tree` object has two attributes, `data` stores data associated with a node and `children` stores a list of `Tree` objects. The recursive aspect of this tree is clear from the way the children are generated as `Tree`'s. This definition does not allow for an empty tree (i.e. one with no nodes).

Let's rewrite our print function to operate on an instance of the `Tree` class.

```
def printtree(T):
    print(T.data)
    for child in T.children:
        printtree(child)
```

```
T = Tree(['a', ['b', ['c', ['d']]],['e',['f'], ['g']]])
printtree(T)
```

```
a
b
c
d
e
f
g
```

This is the most common pattern for algorithms that operate on trees. It has two parts; one part operates on the data and the other part applies the function recursively on the children.

One unfortunate aspect of this code is that although it prints out the data, it doesn't tell us about the structure of the tree. It would be nicer if we used indentation to indicate the depth of the node as we print the data. It turns out that this is not too difficult and we will implement it as our __str__ method.

```
    def __str__(self, level = 0):
        treestring = "  " * level + str(self.data)
        for child in self.children:
            treestring += "\n" + child.__str__(level + 1)
        return treestring

T = Tree(['a', ['b', ['c', ['d']]],['e',['f'], ['g']]])
print(str(T))
```

```
a
  b
    c
      d
  e
    f
    g
```

To "see" the tree structure in this print out, you find the parent of a node by finding the lowest line above that node that is not at the same

indentation level. This is a task that you have some training for; it is how you visually parse Python file to understand their block structure.

Although the code above seems to work, it does something terrible. It builds up a string by iterative adding more strings (i.e. with concatenation). This copies and recopies some part of the string for every node in the tree. Instead, we would prefer to just keep a nice list of the trees and then join them into a string in one final act before returning. To do this, it is handy to use a helper method that takes the level and the current list of trees as parameters. With each recursive call, we add one to the level, and we pass down the same list to be appended to.

```
    def _listwithlevels(self, level, trees):
        trees.append("  " * level + str(self.data))
        for child in self.children:
            child._listwithlevels(level + 1, trees)

    def __str__(self):
        trees = []
        self._listwithlevels(0, trees)
        return "\n".join(trees)

T = Tree(['a', ['b', ['c', ['d']]],['e',['f'], ['g']]])
print(str(T))
```

```
a
  b
    c
      d
  e
    f
    g
```

The pattern involved here is called a tree traversal and we will discuss these in more depth below. We implemented this one first, because it will help us to see that our trees look the way we expect them to. Naturally, it is vital to write good tests, but when things go wrong, you can learn a lot by seeing the tree. It can give you hints on where to look for bugs.

Here is another example of a similar traversal pattern. Let's check if two trees are equal in the sense of having the same shape and data. We use the

__eq__ method so this method will be used when we use == to check equality between Tree's.

```
def __eq__(self, other):
    return self.data == other.data and self.children == other.children
```

Here, it is less obvious that we are doing recursion, but we are because self.children and other.children are lists and list equality is determined by testing the equality of the items. In this case, the items in the lists are Tree's, so our __eq__ method will be invoked for each one.

Here's another example. Let's write a function that computes the height of the tree. We can do this by computing the height of the subtrees and return one more than the max of those. If there are no children, the height is 0.

```
def height(self):
    if len(self.children) == 0:
        return 0
    else:
        return 1 + max(child.height() for child in self.children)
```

Hopefully, you are getting the hang of these generator expressions.

Recall that the __contains__ magic method allows us to use the in keyword for membership testing in a collection. The desired behavior of this function as described in the ADT tells us how to implement it.

```
def __contains__(self, k):
    return self.data == k or any(k in ch for ch in self.children)
```

The any function takes an iterable of booleans and return True if any of them are True. It handles short-circuited evaluation, so it can stop as soon as it finds that one is true. If the answer is False, then this will iterate over the whole tree. This is precisely the behavior we saw with

16.5 Tree Traversal

Previously, all the collections we stored were either sequential (i.e., list, tuple, and str) or non-sequential (i.e., dict and set). The tree structure seems to lie somewhere between the two. There is some structure, but it's not linear. We can give it a linear (sequential) structure by iterating through

all the nodes in the tree, but there is not a unique way to do this. For trees, the process of visiting all the nodes is called **tree traversal**. For ordered trees, there are two standard traversals, called **preorder** and **postorder**, both are naturally defined recursively.

In a preorder traversal, we *visit* the node first followed by the traversal of its children. In a postorder traversal, we traverse all the children and then visit the node itself. The *visit* refers to whatever computation we want to do with the nodes. The `printtree` method given previously is a classic example of a preorder traversal. We could also print the nodes in a postorder traversal as follows.

```
def printpostorder(T):
    for child in self.children:
        printpostoder(child)
    print(T.data)
```

It is only a slight change in the code, but it results in a different output.

16.6 If you want to get fancy...

It was considered a great achievement in Python to be able to do this kind of traversal with a generator. Recursive generators seem a little mysterious, especially at first. However, if you break down this code and walk through it by hand, it will help you have a better understanding of how generators work.

```
    def preorder(self):
        yield self.data
        for child in self.children:
            for data in child.preorder():
                yield data

    # Set __iter__ to be an alias for preorder.
    __iter__ = preorder
```

You can also do this to iterate over the trees (i.e. nodes). I have made this one private because the user of the tree likely does not want or need access to the nodes.

```
def _preorder(self):
    yield self
```

```
    for child in self.children:
        for descendant in child._preorder():
            yield descendant
```

16.6.1 There's a catch!

This recursive generator does not run in linear time! Recall that each call to the generator produces an object. The object does not go away after yielding because it may need to yield more values. If one calls this method on a tree, each value yielded is passed all the way from the node to the root. Thus, the total running time is proportional to the sum of the depths of all the nodes in the tree. For a degenerate tree (i.e. a single path), this is $O(n^2)$ time. For a perfectly balanced binary tree, this is $O(n \log n)$ time.

Using recursion and the call stack make the tree traversal code substantially simpler than if we had to keep track of everything manually. It would not be enough to store just the stack of nodes in the path from your current node up to the root. You would also have to keep track of your place in the iteration of the children of each of those nodes. Remember that it is the job of an iterator object to keep track of where it is in the iteration. Thus, we can just push the iterators for the children onto the stack too.

```
    def _postorder(self):
        node, childiter = self, iter(self.children)
        stack = [(node, childiter)]
        while stack:
            node, childiter = stack[-1]
            try:
                child = next(childiter)
                stack.append((child, iter(child.children)))
            except StopIteration:
                yield node
                stack.pop()

    def postorder(self):
        return (node.data for node in self._postorder())
```

In the above code, I don't love the fact that I am reassigning `node` and `childiter` at every iteration of the loop. Can you fix that so that it still works?

16.6.2 Layer by Layer

Starting with our non-recursive postorder traversal, we can modify it to traverse the tree layer by layer. We will consider this traversal when we study heaps later on.

```
    def _layerorder(self):
        node, childiter = self, iter(self.children)
        queue = Queue()
        queue.enqueue((node, childiter))
        while queue:
            node, childiter = queue.peek()
            try:
                child = next(childiter)
                queue.enqueue((child, iter(child.children)))
            except StopIteration:
                yield node
                queue.dequeue()

    def layerorder(self):
        return (node.data for node in self._layerorder())
```

Chapter 17

Binary Search Trees

Let's start with an ADT. An **ordered mapping** is a mapping data type for which the keys are ordered. It should support all the same operations as any other mapping as well as some other operations similar to those in an `OrderedList` such as predecessor search.

17.1 The Ordered Mapping ADT

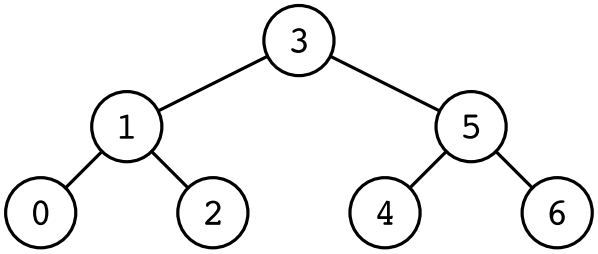
An **ordered mapping** stores a collection of key-value pairs (*with comparable keys*) supporting the following operations.

- `get(k)` - Return the value associate to the key `k`. An error (`KeyError`) is raised if the given key is not present.
- `put(k, v)` - Add the key-value pair `(k,v)` to the mapping.
- `floor(k)` - Return a tuple `(k,v)` corresponding to the key-value pair in the mapping with the largest key that is less than or equal to `k`. If there is no such tuple, it returns `(None, None)`.
- `remove(k)` - Remove the key-value pair with key `k` from the ordered mapping. An error (`KeyError`) is raised if the given key is not present.

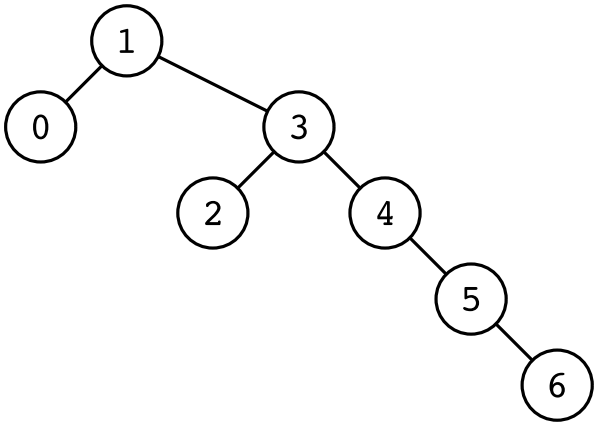
17.2 Binary Search Tree Properties and Definitions

A tree is called a **binary tree** if every node has at most two children. We will continue to assume that we are working with ordered trees and so we

call the children `left` and `right`. We say that a binary tree is a **binary search tree** if for every node `x`, all the keys in the subtree `x.left` are less than the key at `x` and all the keys in the subtree `x.right` are greater than the key of `x`. This ordering property, also known as **the BST property** is what makes a binary search tree different from any other kind of binary tree.



The same set of nodes can be arranged into a different binary search tree.



The BST property is related to a new kind of tree traversal, that was not possible with other trees. Previously we saw *preorder* and *postorder* traversal of trees. These traversals visit all the nodes of the tree. The preorder traversal visits the root of each subtree before to visiting any of its descendants. The postorder traversal visits all the descendants before visiting the root. The new traversal we introduce here is called **inorder traversal** and it visits all the nodes in the left child prior to visiting the root and then visits all the nodes in the right child after visiting the root. This order results in a traversal of the nodes *in sorted order according to the ordering of the keys*.

As a result, different BSTs with the same nodes, will have the same

inorder traversal.

17.3 A Minimal implementation

We're going to implement an ordered mapping using a binary search tree. As we have already written an abstract class for packaging up a lot of the magic methods we expect from a mapping, we will inherit from that class to get started.

Also, unlike in the previous section, we're going to distinguish between a class for the tree (`BSTMapping`) and a class for the nodes (`BSTNode`). We will maintain a convention that the operations on the `BSTNode` class will operate on and return other `BSTNode` objects when appropriate, whereas the `BSTMapping` class will abstract away the existence of the nodes for the user, only returning keys and values.

Here is just the minimum requirements to be a `Mapping`. It's a top down implementation, so it delegates all the real work to the `BSTNode` class.

```
from ds2.mapping import Mapping

class BSTMapping(Mapping):
    def __init__(self):
        self._root = None

    def get(self, key):
        if self._root:
            return self._root.get(key).value
        raise KeyError

    def put(self, key, value):
        if self._root:
            self.root = self._root.put(key, value)
        else:
            self._root = BSTNode(key, value)

    def __len__(self):
        return len(self._root) if self._root is not None else 0

    def _entryiter(self):
        if self._root:
            yield from self._root
```

```
    def floor(self, key):
        if self._root:
            floornode = self._root.floor(key)
            if floornode is not None:
                return floornode.key, floornode.value
        return None, None

    def remove(self, key):
        if self._root:
            self._root = self._root.remove(key)
        else:
            raise KeyError

    def __delitem__(self, key):
        self.remove(key)
```

The code above gives us almost everything we need. There are a couple of mysterious lines to pay attention to. One is the line in the `put` method that updates the root. We will use this convention extensively. As methods may rearrange the tree structure, such methods return the node that ought to be the new root of the subtree. The same pattern appears in the `remove` function.

One other construct we haven't seen before is the `yield from` operation in the iterator. This takes an iterable and iterates over it, yielding each item. So, `yield from self._root` is the same as `for item in iter(self._root): yield item`. It implies that our `BSTNode` class will have to be iterable.

Let's see how these methods are implemented. We start with the initializer and some handy other methods.

```
class BSTNode:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.left = None
        self.right = None
        self._length = 1

    def __len__(self):
```

```
        return self._length

    def __str__(self):
        return str(self.key) + " : " + str(self.value)
```

The `get` method uses binary search to find the desired key.

```
    def get(self, key):
        if key == self.key:
            return self
        elif key < self.key and self.left:
            return self.left.get(key)
        elif key > self.key and self.right:
            return self.right.get(key)
        else:
            raise KeyError
```

Notice that we are using `self.left` and `self.right` as booleans. This works because `None` evaluates to `False` and `BSTNode`'s always evaluate to `True`. We could have implemented `__bool__` to make this work, but it suffices to implement `__len__`. Objects that have a `__len__` method are `True` if and only if the length is greater than `0`. This is the default way to check if a container is empty. So, for example, it's fine to write `while L: L.pop()` and it will never try to pop from an empty list. In our case, it will allow us to write `if self.left` to check if there is a left child rather than writing `if self.left is not None`.

Next, we implement `put`. It will work by first doing a binary search in the tree. If it finds the key already in the tree, it overwrites the value (keys in a mapping are unique). Otherwise, when it gets to the bottom of the tree, it adds a new node.

```
    def put(self, key, value):
        if key == self.key:
            self.value = value
        elif key < self.key:
            if self.left:
                self.left.put(key, value)
            else:
                self.left = BSTNode(key, value)
        elif key > self.key:
            if self.right:
```

```
                self.right.put(key, value)
            else:
                self.right = BSTNode(key, value)
        self._updatelength()

    def _updatelength(self):
        len_left = len(self.left) if self.left else 0
        len_right = len(self.right) if self.right else 0
        self._length = 1 + len_left + len_right
```

The `put` method also keeps track of the length, i.e. the number of entries in each subtree.

17.3.1 The `floor` function

The `floor` function is just a slightly fancier version of `get`. It also does a binary search, but it has different behavior when the key is not found, depending on whether the last search was to the left or to the right. Starting from any node, if we search to the right and the result is `None`, then we return the node itself. If we search to the left and the result is `None`, we also return `None`.

```
    def floor(self, key):
        if key == self.key:
            return self
        elif key < self.key:
            if self.left is not None:
                return self.left.floor(key)
            else:
                return None
        elif key > self.key:
            if self.right is not None:
                floor = self.right.floor(key)
                return floor if floor is not None else self
            else:
                return self
```

17.3.2 Iteration

As mentioned above, binary search trees support inorder traversal. The result of an inorder traversal is that the nodes are yielded *in the order of*

their keys.

Here is an inorder iterator for a binary search tree implemented using recursive generators. This will be fine in most cases. There is a slightly more efficient way, but this way is perhaps the most readable.

```
def __iter__(self):
    if self.left is not None:
        yield from self.left
    yield self
    if self.right is not None:
        yield from self.right
```

17.4 Removal

To implement removal in a binary search tree, we'll use a standard algorithmic problem solving approach. We'll start by thinking about how to handle the easiest possible cases. Then, we'll see how to turn every case into an easy case.

The start of a removal is to find the node to be removed using binary search in the tree. Then, if the node is a leaf, we can remove it without any difficulty. It's also easy to remove a node with only one child because we can remove the node and bring its child up without violating the BST property.

The harder case come when the node to be removed has both a left and a right child. In that case, we find the node with the largest key in its left subtree (i.e. the rightmost node). We swap that node with our node to be removed and call `remove` again on the left subtree. The next time we reach that node, we know it will have at most one child, because the node we swapped it with had no right child (otherwise it wasn't the rightmost before the swap).

A note of caution. Swapping two node in a BST will cause the BST property to be violated. However, the only violation is the node to be removed will have a key greater than the node that we swapped it with. So, the removal will restore the BST property.

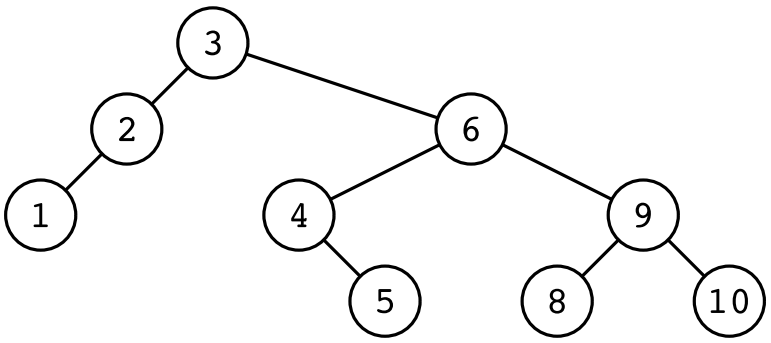
Here is the code to do the swapping and a simple recursive method to find the rightmost node in a subtree.

```
def _swapwith(self, other):
    self.key, other.key = other.key, self.key
    self.value, other.value = other.value, self.value
```

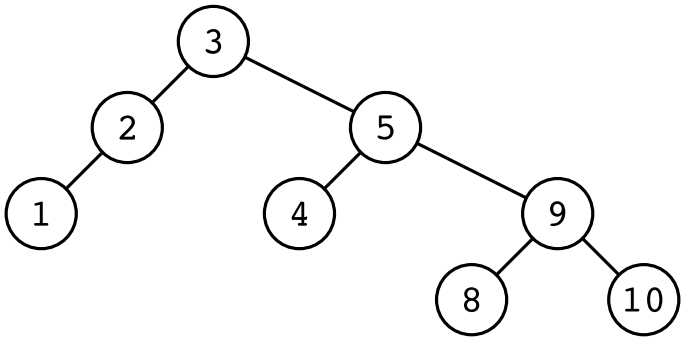
```
    def maxnode(self):
        return self.right.maxnode() if self.right else self
```

Now, we are ready to implement `remove`. As mentioned above, it does a recursive binary search to find the node. When it finds the desired key, it swaps it into place and makes another recursive call. This swapping step will happen only once and the total running time is linear in the height of the tree.

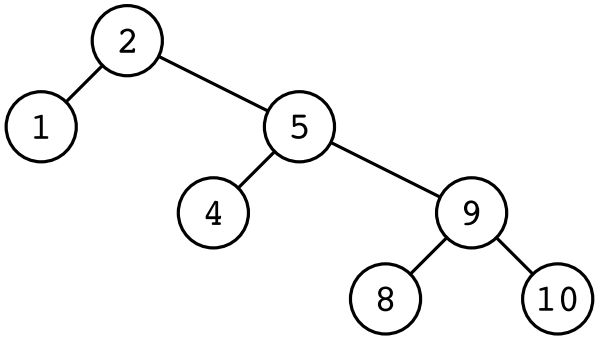
```
    def remove(self, key):
        if key == self.key:
            if self.left is None: return self.right
            if self.right is None: return self.left
            self._swapwith(self.left.maxnode())
            self.left = self.left.remove(key)
        elif key < self.key and self.left:
            self.left = self.left.remove(key)
        elif key > self.key and self.right:
            self.right = self.right.remove(key)
        else: raise KeyError
        self._updatelength()
        return self

T = BSTMapping()
for i in [3,2,1,6,4,5,9,8,10]:
    T[i] = None
```




```
T.remove(6)
```




```
T.remove(3)
```



Chapter 18

Balanced Binary Search Trees

In the previous chapter, we saw how to implement a Mapping using a BST. However, in the analysis of the main operations, it was clear that the running time of all the basic operations depended on the height of the tree.

A BST with n nodes can have height $n-1$. It's easy to get such a tree, just insert the keys in sorted order. On the other hand, there always exists a BST with height $O(\log n)$ for any set of n keys. This is achieved by inserting the median first, followed by the medians of each half and so on recursively. So, we have a big gap between the best case and the worst case. Our goal will be to get as close as possible to the best case while keeping the running times of all operations proportional to the height of the tree.

We will say that a BST with n nodes is **balanced** if the height of the tree is at most some constant times $\log n$. To balance our BSTs, we want to avoid rebuilding the whole tree. Instead, we'd like to make a minimal change to the tree that will preserve the BST property.

The most basic such operation is called a **tree rotation**. It comes in two forms, `rotateright` and `rotateleft`. Rotating a node to the right will move it to be the right child of its left child and will update the children of these nodes appropriately. Here it is in code.

```
def rotateright(self):
    newroot = self.left
    self.left = newroot.right
    newroot.right = self
    return newroot
```

Notice that `rotateright` returns the new root (of the subtree). This is a very useful convention when working with BSTs and rotations. Every method that can change the structure of the tree will return the new root of the resulting subtree. Thus, calling such methods will always be combined with an assignment. For example, if we want to rotate the left child of a node `parent` to the right, we would write `parent.left = parent.left.rotateright()`.

18.1 A `BSTMapping` implementation

Here is the code for the `BSTMapping` that implements rotations. We make sure to also update the lengths after each rotation. The main difference with our previous code is that now, all methods that can change the tree structure are combined with assignments. It is assumed that only `put` and `remove` will rearrange the tree, and so `get` and `floor` will keep the tree structure as is.

```
from ds2.orderedmapping import BSTMapping, BSTNode

class BalancedBSTNode(BSTNode):
    def newnode(self, key, value):
        return BalancedBSTNode(key, value)

    def put(self, key, value):
        if key == self.key:
            self.value = value
        elif key < self.key:
            if self.left:
                self.left = self.left.put(key, value)
            else:
                self.left = self.newnode(key, value)
        elif key > self.key:
            if self.right:
                self.right = self.right.put(key, value)
            else:
                self.right = self.newnode(key, value)
        self._updatelength()
        return self

    def rotateright(self):
        newroot = self.left
```

```
        self.left = newroot.right
        newroot.right = self
        self._updatelength()
        newroot._updatelength()
        return newroot

    def rotateleft(self):
        newroot = self.right
        self.right = newroot.left
        newroot.left = self
        self._updatelength()
        newroot._updatelength()
        return newroot

class BalancedBST(BSTMapping):
    Node = BalancedBSTNode

    def put(self, key, value):
        if self._root:
            self._root = self._root.put(key, value)
        else:
            self._root = self.Node(key, value)
```

18.1.1 Forward Compatibility of Factories

We are looking into the future a little with this code. In particular, we want this class to be easily extendable. To do this, we don't create new instances of the nodes directly. In the `BSTMapping` class, we have the assignment `Node = BSTNode` and create new nodes by writing `self.Node(key, value)` rather than `BSTNode(key, value)`. This way, when we extend this class (as we will several times), we can use a different value for `Node` in order to produce different types of nodes without having to rewrite all the methods from `BSTMapping`. Similarly, we use a `newnode` method in the `BSTNode` class. A method that generates new instances of a class is sometimes called a **factory**.

With regards to *forward compatibility*, it's a challenge to walk the fine line between, good design and premature optimization. In our case, the use of a factory to generate nodes only came about after the need was clear from writing subclasses. This should be a general warning to students when reading code in textbooks. One often only sees how the code ended up, but

not how it started. It's helpful to ask, "Why is the code written this way or that way?", but the answer might be that it will make later code easier to write.

18.2 Weight Balanced Trees

Now, we're ready to balance our trees. One way to be sure the tree is balanced is to have the key at each node `x` be the median of all the keys in the subtree at `x`. Then, the situation would be analogous to binary search in a sorted `list`. Moving down the tree, every node would have have as many nodes in its subtree than its parent. Thus, after $\log n$ steps, we reach a leaf and so the BST would be balanced.

Having medians in every node a nice ideal to have, but it is both too difficult to achieve and too strong an invariant for balancing BSTs. Recall that in our analysis of `quicksort` and `quickselect`, we considered *good pivots* to be those that lay in the middle half of the list (i.e. those with rank between $n/4$ and $3n/4$). If the key in each node is a good pivot among the keys in its subtree, then we can also guarantee an overall height of $\log_{4/3} n$.

We'll say a node `x` is **weight-balanced** if

`len(x) + 1 < 4 * (min(len(x.left), len(x.right)) + 1)`.

It's easy enough to check this condition with our `BSTMapping` implementation because it keeps track of the length at each node. If some change causes a node to no longer be weight balanced, we will recover the weight balance by rotations. In the easiest case, a single rotation suffices. If one is rotation is not enough, then two rotations will be enough. We'll have to see some examples and do some simple algebra to see why.

The `rebalance` method will check for the balance condition and do the appropriate rotations. We'll call this method after every change to the tree, so we can assume that the unbalanced node `x` is only just barely imbalanced. Without loss of generality, let's assume `x` has too few nodes in its left subtree. Then, when we call `x.rotateleft()`, we have to check that both `x` becomes weight balanced and also that its new parent `y` (the former right child of `x`) is weight balanced. Clearly, `len(x) == len(x.rotateleft())` as both contain the same nodes. So, there is an imbalance at `y` after the rotation only if its right child is too light. In that case, we rotate `y` right before rotating `x` left. This will guarantee that all the nodes in the subtree are weight balanced.

This description is not enough to be convincing that the algorithm is correct, but it is enough to write code. Let's look at the code first and see

the rebalance method in action. Then, we'll go back and do the algebra to prove it is correct.

```
from ds2.orderedmapping import BalancedBST, BalancedBSTNode

class WBTreeNode(BalancedBSTNode):
    def newnode(self, key, value):
        return WBTreeNode(key, value)

    def toolight(self, other):
        otherlength = len(other) if other else 0
        return len(self) + 1 >= 4 * (otherlength + 1)

    def rebalance(self):
        if self.toolight(self.left):
            if self.toolight(self.right.right):
                self.right = self.right.rotateright()
            newroot = self.rotateleft()
        elif self.toolight(self.right):
            if self.toolight(self.left.left):
                self.left = self.left.rotateleft()
            newroot = self.rotateright()
        else:
            return self
        return newroot

    def put(self, key, value):
        newroot = BalancedBSTNode.put(self, key, value)
        return newroot.rebalance()

    def remove(self, key):
        newroot = BalancedBSTNode.remove(self, key)
        return newroot.rebalance() if newroot else None

class WBTree(BalancedBST):
    Node = WBTreeNode
```

The `toolight` method is for checking if a subtree has enough nodes to be a child of a weight balanced node. We use it both to check if the current node is weight balanced and also to check if one rotation or two will be required.

The `put` and `remove` methods call the corresponding methods from the superclass `BSTNode` and then rebalance before returning. We tried to reuse as much as possible our existing implementation.

18.3 Height-Balanced Trees (AVL Trees)

The motivation for balancing our BSTs was to keep the height small. Rather than balancing by weight, we could also try to keep the heights of the left and right subtrees close. In fact, we can require that these heights differ by at most one. Similar to weight balanced trees, we'll see if the height balanced property is violated and if so, fix it with one or two rotations.

Such height balanced trees are often called AVL trees. In our implementation, we'll maintain the height of each subtree and use these to check for balance. Often, AVL trees only keep the balance at each node rather than the exact height, but computing heights is relatively painless.

```
from ds2.orderedmapping import BalancedBST, BalancedBSTNode

def height(node):
    return node.height if node else -1

def update(node):
    if node:
        node._updatelength()
        node._updateheight()

class AVLTreeNode(BalancedBSTNode):
    def __init__(self, key, value):
        BalancedBSTNode.__init__(self, key, value)
        self._updateheight()

    def newnode(self, key, value):
        return AVLTreeNode(key, value)

    def _updateheight(self):
        self.height = 1 + max(height(self.left), height(self.right))

    def balance(self):
        return height(self.right) - height(self.left)
```

```
    def rebalance(self):
        bal = self.balance()
        if bal == -2:
            if self.left.balance() > 0:
                self.left = self.left.rotateleft()
            newroot = self.rotateright()
        elif bal == 2:
            if self.right.balance() < 0:
                self.right = self.right.rotateright()
            newroot = self.rotateleft()
        else:
            return self
        update(newroot.left)
        update(newroot.right)
        update(newroot)
        return newroot

    def put(self, key, value):
        newroot = BalancedBSTNode.put(self, key, value)
        update(newroot)
        return newroot.rebalance()

    def remove(self, key):
        newroot = BalancedBSTNode.remove(self, key)
        update(newroot)
        return newroot.rebalance() if newroot else None

class AVLTree(BalancedBST):
    Node = AVLTreeNode
```

18.4 Splay Trees

Let's add in one more balanced BST for good measure. In this case, we won't actually get a guarantee that the resulting tree is balanced, but we will get some other nice properties.

In a **splay tree**, every time we get or put an entry, its node will get rotated all the way to the root. However, instead of rotating it directly, we consider two steps at a time. The `splayup` method looks two levels down the tree for the desired key. If its not exactly two levels down, it does nothing.

Otherwise, it rotates it up twice. If the rotations are in the same direction, the bottom one is done first. If the rotations are in opposite directions, it does the top one first.

At the very top level, we may do only a single rotation to get the node all the way to the root. This is handled by the `splayup` method in the `SplayTree` class.

A major difference from our previous implementations is that now, we will modify the tree on calls to `get`. As a result, we will have to rewrite `get` rather than inheriting it. Previously, get would return the desired value. However, we want to return the new root on every operation that might change the tree. So, which should we return? Clearly, we need that value to return, and we also need to not break the tree. Thankfully, there is a simple solution. The splaying operation conveniently rotates the found node all the way to the root. So, the `SplayTreeNode.get` method will return the new root of the subtree, and the `SplayTree.get` returns the value at the root.

```
from ds2.orderedmapping import BalancedBST, BalancedBSTNode

class SplayTreeNode(BalancedBSTNode):
    def newnode(self, key, value):
        return SplayTreeNode(key, value)

    def splayup(self, key):
        newroot = self
        if key < self.key:
            if key < self.left.key:
                newroot = self.rotateright().rotateright()
            elif key > self.left.key:
                self.left = self.left.rotateleft()
                newroot = self.rotateright()
        elif key > self.key:
            if key > self.right.key:
                newroot = self.rotateleft().rotateleft()
            elif key < self.right.key:
                self.right = self.right.rotateright()
                newroot = self.rotateleft()
        return newroot

    def put(self, key, value):
        newroot = BalancedBSTNode.put(self, key, value)
```

```
        return newroot.splayup(key)

    def get(self, key):
        if key == self.key:
            return self
        elif key < self.key and self.left:
            self.left = self.left.get(key)
        elif key > self.key and self.right:
            self.right = self.right.get(key)
        else:
            raise KeyError
        return self.splayup(key)

class SplayTree(BalancedBST):
    Node = SplayTreeNode

    def splayup(self, key):
        if key < self._root.key:
            self._root = self._root.rotateright()
        if key > self._root.key:
            self._root = self._root.rotateleft()

    def get(self, key):
        if self._root is None: raise KeyError
        self._root = self._root.get(key)
        self.splayup(key)
        return self._root.value

    def put(self, key, value):
        BalancedBST.put(self, key, value)
        self.splayup(key)
```