Chapter 2
Hide contents

2.1 Introduction
2.1.1 The Object Metaphor
2.1.2 Native Data Types

2.2 Data Abstraction
2.2.1 Example: Arithmetic on Rational Numbers
2.2.2 Pairs
2.2.3 Abstraction Barriers
2.2.4 The Properties of Data

2.3 Sequences
2.3.1 Tuples
2.3.2 Sequence Iteration
2.3.3 Sequence Abstraction
2.3.4 Nested Pairs
2.3.5 Recursive Lists
2.3.6 Strings
2.3.7 Sequence Processing

2.4 Mutable Data
2.4.1 Lists
2.4.2 Dictionaries
2.4.3 Local State
2.4.4 The Benefits of Non-Local Assignment
2.4.5 The Cost of Non-Local Assignment
2.4.6 Implementing Lists and Dictionaries
2.4.7 Dispatch Dictionaries
2.4.8 Propagating Constraints

2.5 Object-Oriented Programming
2.5.1 Objects and Classes
2.5.2 Defining Classes
2.5.3 Message Passing and Dot Expressions
2.5.4 Class Attributes
2.5.5 Inheritance
2.5.6 Using Inheritance
2.5.7 Multiple Inheritance
2.5.8 Functions as Objects
2.5.9 The Role of Objects

2.6 Implementing Classes and Objects
2.6.1 Instances
2.6.2 Classes
2.6.3 Using Implemented Objects

2.7 Recursive Data Structures
2.7.1 A Recursive List Class
2.7.2 Hierarchical Structures
2.7.3 Memoization
2.7.4 Orders of Growth
2.7.5 Example: Exponentiation
2.7.6 Sets

2.8 Generic Operations

2.7 Recursive Data Structures

Objects can have other objects as attribute values. When an object of some class has an attribute value of that same class, the result is a recursive data structure.

2.7.1 A Recursive List Class

A recursive list, introduced earlier in this chapter, is an abstract data type composed of a first element and the rest of the list. The rest of a recursive list is itself a recursive list. The empty list is treated as a special case that has no first element or rest. A recursive list is a sequence: it has a finite length and supports element selection by index.

We can now implement a class with the same behavior. In this version, we will define its behavior using special method names that allow our class to work with the built-in `len` function and element selection operator (square brackets or `operator.getitem`) in Python. These built-in functions invoke special method names of a class: length is computed by `__len__` and element selection is computed by `__getitem__`.

```
>>> class Rlist:
        """A recursive list consisting of a first element and the rest."""
        class EmptyList(object):
            def __len__(self):
                return 0
        empty = EmptyList()
        def __init__(self, first, rest=empty):
            self.first = first
            self.rest = rest
        def __getitem__(self, i):
            if i == 0:
                return self.first
            else:
                return self.rest[i-1]
        def __len__(self):
            return 1 + len(self.rest)

>>> s = Rlist(3, Rlist(4, Rlist(5)))
>>> len(s)
3
>>> s[1]
4
```

The definitions of `__len__` and `__getitem__` are in fact recursive. The built-in Python function `len` invokes a method called `__len__` when applied to a user-defined object argument. Likewise, the element selection operator invokes a method called `__getitem__`. Thus, bodies of these two methods will call themselves indirectly.

Our implementation is complete, but an instance of the `Rlist` class is currently difficult to inspect. To help with debugging, we can also define a function to convert an `Rlist` to a string expression.

```
>>> def rlist_expression(s):
        """Return a string that would evaluate to s."""
        if s.rest is Rlist.empty:
            rest = ''
        else:
            rest = ', ' + rlist_expression(s.rest)
        return 'Rlist({0}{1})'.format(s.first, rest)

>>> rlist_expression(s)
'Rlist(3, Rlist(4, Rlist(5)))'
```

This way of displaying an `Rlist` is so convenient that we would like to use it whenever an `Rlist` instance is displayed. We can ensure this behavior by setting the `rlist_expression` function as the value of a special class attribute `__repr__`. Python displays instances of user-defined classes by invoking their `__repr__` method.

```
>>> Rlist.__repr__ = rlist_expression
>>> s
Rlist(3, Rlist(4, Rlist(5)))
```

Using the `Rlist` class, we can define some common operations on recursive lists. For example, we can create a new `Rlist` that contains all elements of two input `Rlists`.

```
>>> def extend_rlist(s1, s2):
        """Return an Rlist with the elements of s1 followed by those of s2."""
        if s1 is Rlist.empty:
            return s2
        else:
            return Rlist(s1.first, extend_rlist(s1.rest, s2))
```

Then, for example, we can extend the rest of `s` with `s`.

```
>>> extend_rlist(s.rest, s)
Rlist(4, Rlist(5, Rlist(3, Rlist(4, Rlist(5)))))
```

The implementation for mapping a function over a recursive list has a similar structure.

2.8.1 String Conversion
2.8.2 Multiple Representations
2.8.3 Special Methods
2.8.4 Generic Functions

```
>>> def map_rlist(s, fn):
        if s is Rlist.empty:
            return s
        else:
            return Rlist(fn(s.first), map_rlist(s.rest, fn))

>>> map_rlist(s, square)
Rlist(9, Rlist(16, Rlist(25)))
```

Filtering includes an additional conditional statement, but also has a similar recursive structure.

```
>>> def filter_rlist(s, fn):
        if s is Rlist.empty:
            return s
        else:
            rest = filter_rlist(s.rest, fn)
            if fn(s.first):
                return Rlist(s.first, rest)
            else:
                return rest

>>> filter_rlist(s, lambda x: x % 2 == 1)
Rlist(3, Rlist(5))
```

Recursive implementations of list operations do not, in general, require local assignment or `while` statements. Instead, recursive lists can be taken apart and constructed incrementally as a consequence of function application.

2.7.2 Hierarchical Structures

Hierarchical structures result from the closure property of data, which asserts for example that tuples can contain other tuples. For instance, consider this nested representation of the numbers 1 through 5. This tuple is a length-three sequence, of which the first two elements are themselves tuples. A tuple that contains tuples or other values is a tree.

```
1 t = ((1, 2), (3, 4), 5)
```

Edit code in Online Python Tutor

< Back Program terminated Forward >

Frames Objects

Global frame t

tuple 0 1 2 5 | tuple 0 1 1 2 | tuple 0 1 3 4

In a tree, each subtree is itself a tree. As a base condition, any bare element that is not a tuple is itself a simple tree, one with no branches. That is, the numbers are all trees, as is the pair `(1, 2)` and the structure as a whole.

Recursion is a natural tool for dealing with tree structures, since we can often perform operations on trees by performing the same operations on each of their branches, and likewise on the branches of the branches until we reach the leaves of the tree. As an example, we can implement a `count_leaves` function, which returns the total number of leaves of a tree. Step through this function to see how the leaves are counted.

```
1 def count_leaves(tree):
2     if type(tree) != tuple:
3         return 1
4     else:
5         return sum(map(count_leaves, tree))
6
7 t = ((1, 2), (3, 4), 5)
8 result = count_leaves(t)
```

Edit code in Online Python Tutor

< Back Step 3 of 27 Forward >

Frames Objects

Global frame count_leaves t

func count_leaves(tree)

tuple 0 1 2 5 | tuple 0 1 1 2 | tuple 0 1 3 4

Just as `map` is a powerful tool for dealing with sequences, mapping and recursion together provide a powerful general form of computation for manipulating trees. For instance, we can square all leaves of a tree using a higher-order recursive function `map_tree` that is structured quite similarly to `count_leaves`.

```
>>> def map_tree(tree, fn):
        """Map fn over all leaves in tree."""
        if type(tree) != tuple:
            return fn(tree)
        else:
            return tuple(map_tree(branch, fn) for branch in tree)

>>> t = ((1, 2), (3, 4), 5)
>>> map_tree(t, square)
```

```
((1, 4), (9, 16), 25)
```

Internal values. The trees described above have values only at the leaves. Another common representation of tree-structured data has values for the internal nodes of the tree as well. An internal value is called an `entry` in the tree. The `Tree` class below represents such trees, in which each tree has at most two branches `left` and `right`.

```
>>> class Tree:
        def __init__(self, entry, left=None, right=None):
            self.entry = entry
            self.left = left
            self.right = right
        def __repr__(self):
            if self.left or self.right:
                args = self.entry, self.left, self.right
                return 'Tree({0}, {1}, {2})'.format(*args)
            else:
                return 'Tree({0})'.format(self.entry)
```

The `Tree` class can represent, for instance, the values computed in an expression tree for the recursive implementation of `fib`, the function for computing Fibonacci numbers. The function `fib_tree(n)` below returns a `Tree` that has the nth Fibonacci number as its `entry` and a trace of all previously computed Fibonacci numbers within its branches.

```
>>> def fib_tree(n):
        """Return a Tree that represents a recursive Fibonacci calculation."""
        if n == 1:
            return Tree(0)
        elif n == 2:
            return Tree(1)
        else:
            left = fib_tree(n-2)
            right = fib_tree(n-1)
            return Tree(left.entry + right.entry, left, right)

>>> fib_tree(5)
Tree(3, Tree(1, Tree(0), Tree(1)), Tree(2, Tree(1), Tree(1, Tree(0), Tree(1))))
```

Trees represented in this way are also processed using recursive functions. For example, we can sum the entries of a tree. As a base case, we return that an empty branch has no entries.

```
>>> def sum_entries(t):
        """Sum the entries of a Tree instance, which may be None."""
        if t is None:
            return 0
        else:
            return t.entry + sum_entries(t.left) + sum_entries(t.right)

>>> sum_entries(fib_tree(5))
10
```

2.7.3 Memoization

The `fib_tree` function above can generate very large trees. For example, `fib_tree(35)` consists of 18,454,929 instances of the `Tree` class. However, many of those trees are identical in structure. For example, both the `left` tree and the `right` of the `right` tree are the result of calling `fib_tree(33)`. It would save an enormous amount of time and memory to create one such subtree and use it multiple times.

Tree-recursive data structures and computational processes can often be made more efficient through *memoization*, a powerful technique for increasing the efficiency of recursive functions that repeat computation. A memoized function will store the return value for any arguments it has previously received. A second call to `fib_tree(33)` would not build an entirely new tree, but instead return the existing one that has already been constructed. Within the enormous structure computed by `fib_tree(35)`, there are only 35 unique trees (one for each Fibonacci number).

Memoization can be expressed naturally as a higher-order function, which can also be used as a decorator. The definition below creates a *cache* of previously computed results, indexed by the arguments from which they were computed. The use of a dictionary requires that the argument to the memoized function be ammutable.

```
>>> def memo(f):
        """Return a memoized version of single-argument function f."""
        cache = {}
        def memoized(n):
            if n not in cache:
                cache[n] = f(n)
            return cache[n]
        return memoized
```

We can apply `memo` in a recursive computation of Fibonacci numbers.

```
>>> @memo
    def fib(n):
        if n == 1:
            return 0
        if n == 2:
            return 1
        return fib(n-2) + fib(n-1)
```

```
>>> fib(35)
5702887
```

We can also apply `memo` to construct a Fibonacci tree, where repeated subtrees are only created once by the memoized version of `fib_tree`, but are used multiple times as branches of different larger trees.

```
>>> fib_tree = memo(fib_tree)
>>> big_fib_tree = fib_tree(35)
>>> big_fib_tree.entry
5702887
>>> big_fib_tree.left is big_fib_tree.right.right
True
>>> sum_entries = memo(sum_entries)
>>> sum_entries(big_fib_tree)
142587180
```

The amount of computation time and memory saved by memoization in these cases is substantial. Instead of creating 18,454,929 different instances of the `Tree` class, we now create only 35.

2.7.4 Orders of Growth

The previous examples illustrate that processes can differ considerably in the rates at which they consume the computational resources of space and time. For some functions, we can exactly predict the number of steps in the computational process evolved by those functions. For example, consider the function `count_factors` below that counts the number of integers that evenly divide an input `n`, by attempting to divide it by every integer less than or equal to its square root. The implementation takes advantage of the fact that if k divides n and $k < \sqrt{n}$, then there is another factor $j = n/k$ such that $j > \sqrt{n}$.

```
 1  from math import sqrt
 2  def count_factors(n):
 3      sqrt_n = sqrt(n)
 4      k, factors = 1, 0
 5      while k < sqrt_n:
 6          if n % k == 0:
 7              factors += 2
 8          k += 1
 9      if k * k == n:
10          factors += 1
11      return factors
12
13  result = count_factors(576)
```

Edit code in Online Python Tutor

< Back   Program terminated   Forward >

Frames | Objects

Global frame: sqrt → func sqrt(...); count_factors → func count_factors(n); result 21

count_factors: n 576; sqrt_n 24.0; k 24; factors 21; Return value 21

The total number of times this process executes the body of the `while` statement is the greatest integer less than $\sqrt{n}$. Hence, we can say that the amount of time used by this function, typically denoted $R(n)$, scales with the square root of the input, which we write as $R(n) = \sqrt{n}$.

For most functions, we cannot exactly determine the number of steps or iterations they will require. One convenient way to describe this difference is to use the notion of *order of growth* to obtain a coarse measure of the resources required by a process as the inputs become larger.

Let n be a parameter that measures the size of the problem to be solved, and let $R(n)$ be the amount of resources the process requires for a problem of size n. In our previous examples we took n to be the number for which a given function is to be computed, but there are other possibilities. For instance, if our goal is to compute an approximation to the square root of a number, we might take n to be the number of digits of accuracy required. In general there are a number of properties of the problem with respect to which it will be desirable to analyze a given process. Similarly, $R(n)$ might measure the amount of memory used, the number of elementary machine operations performed, and so on. In computers that do only a fixed number of operations at a time, the time required to evaluate an expression will be proportional to the number of elementary machine operations performed in the process of evaluation.

We say that $R(n)$ has order of growth $\Theta(f(n))$, written $R(n) = \Theta(f(n))$ (pronounced "theta of $f(n)$"), if there are positive constants k_1 and k_2 independent of n such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

for any sufficiently large value of n. In other words, for large n, the value $R(n)$ is sandwiched between two values that both scale with $f(n)$:

- A lower bound $k_1 \cdot f(n)$ and
- An upper bound $k_2 \cdot f(n)$

For instance, the number of steps to compute $n!$ grows proportionally to the input n. Thus, the steps required for this process grows as $\Theta(n)$. We also saw that the space required for the recursive implementation `fact` grows as $\Theta(n)$. By contrast, the iterative

implementation `fact_iter` takes a similar number of steps, but the space it requires stays constant. In this case, we say that the space grows as $\Theta(1)$

The number of steps in our tree-recursive Fibonacci computation `fib` grows exponentially in its input n. In particular, one can show that the nth Fibonacci number is the closest integer to

$$\frac{\phi^{n-2}}{\sqrt{5}}$$

where ϕ is the golden ratio:

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.6180$$

We also stated that the number of steps scales with the resulting value, and so the tree-recursive process requires $\Theta(\phi^n)$ steps, a function that grows exponentially with n.

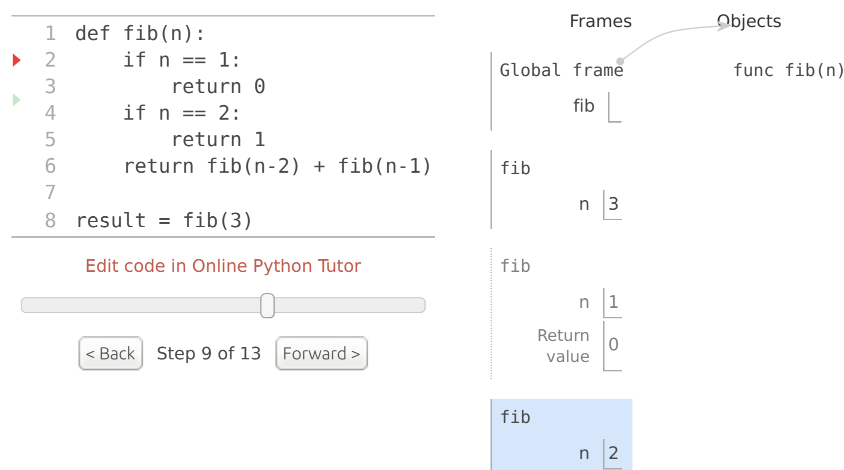
Orders of growth provide only a crude description of the behavior of a process. For example, a process requiring n^2 steps and a process requiring $1000 \cdot n^2$ steps and a process requiring $3 \cdot n^2 + 10 \cdot n + 17$ steps all have $\Theta(n^2)$ order of growth. There are certainly cases in which an order of growth analysis is too coarse a method for deciding between two possible implementations of a function.

However, order of growth provides a useful indication of how we may expect the behavior of the process to change as we change the size of the problem. For a $\Theta(n)$ (linear) process, doubling the size will roughly double the amount of resources used. For an exponential process, each increment in problem size will multiply the resource utilization by a constant factor. The next example examines an algorithm whose order of growth is logarithmic, so that doubling the problem size increases the resource requirement by only a constant amount.

Space. To understand the space requirements of a function, we must specify generally how memory is used, preserved, and reclaimed in our environment model of computation. In evaluating an expression, we must preserve all *active* environments and all values and frames referenced by those environments. An environment is active if it provides the evaluation context for some expression being evaluated.

For example, when evaluating `fib`, the interpreter proceeds to compute each value in the order shown previously, traversing the structure of the tree. To do so, it only needs to keep track of those nodes that are above the current node in the tree at any point in the computation. The memory used to evaluate the rest of the branches can be reclaimed because it cannot affect future computation. In general, the space required for tree-recursive functions will be proportional to the maximum depth of the tree.

The diagram below depicts the environment created by evaluating `fib(3)`. In the process of evaluating the return expression for the initial application of `fib`, the expression `fib(n-2)` is evaluated, yielding a value of 0. Once this value is computed, the corresponding environment frame (grayed out) is no longer needed: it is not part of an active environment. Thus, a well-designed interpreter can reclaim the memory that was used to store this frame. On the other hand, if the interpreter is currently evaluating `fib(n-1)`, then the environment created by this application of `fib` (in which `n` is 2) is active. In turn, the environment originally created to apply `fib` to 3 is active because its return value has not yet been computed.



In the case of `memo`, the environment associated with the function it returns (which contains `cache`) must be preserved as long as some name is bound to that function in an active environment. The number of entries in the `cache` dictionary grows linearly with the number of unique arguments passed to `fib`, which scales linearly with the input. On the other hand, the iterative implementation requires only two numbers to be tracked during computation: `prev` and `curr`, giving it a constant size.

Memoization exemplifies a common pattern in programming that computation time can often be decreased at the expense of increased use of space, or vis versa.

2.7.5 Example: Exponentiation

Consider the problem of computing the exponential of a given number. We would like a function that takes as arguments a base `b` and a positive integer exponent `n` and computes b^n. One way to do this is via the recursive definition

$$b^n = b \cdot b^{n-1}$$
$$b^0 = 1$$

which translates readily into the recursive function

```
>>> def exp(b, n):
        if n == 0:
            return 1
        return b * exp(b, n-1)
```

This is a linear recursive process that requires $\Theta(n)$ steps and $\Theta(n)$ space. Just as with factorial, we can readily formulate an equivalent linear iteration that requires a similar number of steps but constant space.

```
>>> def exp_iter(b, n):
        result = 1
        for _ in range(n):
            result = result * b
        return result
```

We can compute exponentials in fewer steps by using successive squaring. For instance, rather than computing b^8 as

$$b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot b))))))$$

we can compute it using three multiplications:

$$b^2 = b \cdot b$$
$$b^4 = b^2 \cdot b^2$$
$$b^8 = b^4 \cdot b^4$$

This method works fine for exponents that are powers of 2. We can also take advantage of successive squaring in computing exponentials in general if we use the recursive rule

$$b^n = \begin{cases} (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

We can express this method as a recursive function as well:

```
>>> def square(x):
        return x*x

>>> def fast_exp(b, n):
        if n == 0:
            return 1
        if n % 2 == 0:
            return square(fast_exp(b, n//2))
        else:
            return b * fast_exp(b, n-1)

>>> fast_exp(2, 100)
1267650600228229401496703205376
```

The process evolved by `fast_exp` grows logarithmically with `n` in both space and number of steps. To see this, observe that computing b^{2n} using `fast_exp` requires only one more multiplication than computing b^n. The size of the exponent we can compute therefore doubles (approximately) with every new multiplication we are allowed. Thus, the number of multiplications required for an exponent of `n` grows about as fast as the logarithm of `n` base 2. The process has $\Theta(\log n)$ growth. The difference between $\Theta(\log n)$ growth and $\Theta(n)$ growth becomes striking as n becomes large. For example, `fast_exp` for `n` of 1000 requires only 14 multiplications instead of 1000.

2.7.6 Sets

In addition to the list, tuple, and dictionary, Python has a fourth built-in container type called a `set`. Set literals follow the mathematical notation of elements enclosed in braces. Duplicate elements are removed upon construction. Sets are unordered collections, and so the printed ordering may differ from the element ordering in the set literal.

```
>>> s = {3, 2, 1, 4, 4}
>>> s
{1, 2, 3, 4}
```

Python sets support a variety of operations, including membership tests, length computation, and the standard set operations of union and intersection

```
>>> 3 in s
True
>>> len(s)
4
>>> s.union({1, 5})
{1, 2, 3, 4, 5}
>>> s.intersection({6, 5, 4, 3})
```

```
{3, 4}
```

In addition to `union` and `intersection`, Python sets support several other methods. The predicates `isdisjoint`, `issubset`, and `issuperset` provide set comparison. Sets are mutable, and can be changed one element at a time using `add`, `remove`, `discard`, and `pop`. Additional methods provide multi-element mutations, such as `clear` and `update`. The Python documentation for sets should be sufficiently intelligible at this point of the course to fill in the details.

Implementing sets. Abstractly, a set is a collection of distinct objects that supports membership testing, union, intersection, and adjunction. Adjoining an element and a set returns a new set that contains all of the original set's elements along with the new element, if it is distinct. Union and intersection return the set of elements that appear in either or both sets, respectively. As with any data abstraction, we are free to implement any functions over any representation of sets that provides this collection of behaviors.

In the remainder of this section, we consider three different methods of implementing sets that vary in their representation. We will characterize the efficiency of these different representations by analyzing the order of growth of set operations. We will use our `Rlist` and `Tree` classes from earlier in this section, which allow for simple and elegant recursive solutions for elementary set operations.

Sets as unordered sequences. One way to represent a set is as a sequence in which no element appears more than once. The empty set is represented by the empty sequence. Membership testing walks recursively through the list.

```
>>> def empty(s):
        return s is Rlist.empty

>>> def set_contains(s, v):
        """Return True if and only if set s contains v."""
        if empty(s):
            return False
        elif s.first == v:
            return True
        else:
            return set_contains(s.rest, v)

>>> s = Rlist(3, Rlist(4, Rlist(5)))
>>> set_contains(s, 2)
False
>>> set_contains(s, 5)
True
```

This implementation of `set_contains` requires $\Theta(n)$ time to test membership of an element, where n is the size of the set `s`. Using this linear-time function for membership, we can adjoin an element to a set, also in linear time.

```
>>> def adjoin_set(s, v):
        """Return a set containing all elements of s and element v."""
        if set_contains(s, v):
            return s
        else:
            return Rlist(v, s)

>>> t = adjoin_set(s, 2)
>>> t
Rlist(2, Rlist(3, Rlist(4, Rlist(5))))
```

In designing a representation, one of the issues with which we should be concerned is efficiency. Intersecting two sets `set1` and `set2` also requires membership testing, but this time each element of `set1` must be tested for membership in `set2`, leading to a quadratic order of growth in the number of steps, $\Theta(n^2)$, for two sets of size n.

```
>>> def intersect_set(set1, set2):
        """Return a set containing all elements common to set1 and set2."""
        return filter_rlist(set1, lambda v: set_contains(set2, v))

>>> intersect_set(t, map_rlist(s, square))
Rlist(4)
```

When computing the union of two sets, we must be careful not to include any element twice. The `union_set` function also requires a linear number of membership tests, creating a process that also includes $\Theta(n^2)$ steps.

```
>>> def union_set(set1, set2):
        """Return a set containing all elements either in set1 or set2."""
        set1_not_set2 = filter_rlist(set1, lambda v: not set_contains(set2, v))
        return extend_rlist(set1_not_set2, set2)

>>> union_set(t, s)
Rlist(2, Rlist(3, Rlist(4, Rlist(5))))
```

Sets as ordered tuples. One way to speed up our set operations is to change the representation so that the set elements are listed in increasing order. To do this, we need some way to compare two objects so that we can say which is bigger. In Python, many different types of objects can be compared using `<` and `>` operators, but we will concentrate on numbers in this example. We will represent a set of numbers by listing its elements in increasing order.

One advantage of ordering shows up in `set_contains`: In checking for the presence of an

object, we no longer have to scan the entire set. If we reach a set element that is larger than the item we are looking for, then we know that the item is not in the set:

```
>>> def set_contains(s, v):
        if empty(s) or s.first > v:
            return False
        elif s.first == v:
            return True
        else:
            return set_contains(s.rest, v)

>>> set_contains(s, 0)
False
```

How many steps does this save? In the worst case, the item we are looking for may be the largest one in the set, so the number of steps is the same as for the unordered representation. On the other hand, if we search for items of many different sizes we can expect that sometimes we will be able to stop searching at a point near the beginning of the list and that other times we will still need to examine most of the list. On average we should expect to have to examine about half of the items in the set. Thus, the average number of steps required will be about $\frac{n}{2}$. This is still $\Theta(n)$growth, but it does save us, on average, a factor of 2 in the number of steps over the previous implementation.

We can obtain a more impressive speedup by re-implementing `intersect_set`. In the unordered representation, this operation required $\Theta(n^2)$steps because we performed a complete scan of `set2` for each element of `set1`. But with the ordered representation, we can use a more clever method. We iterate through both sets simultaneously, tracking an element `e1` in `set1` and `e2` in `set2`. When `e1` and `e2` are equal, we include that element in the intersection.

Suppose, however, that `e1` is less than `e2`. Since `e2` is smaller than the remaining elements of `set2`, we can immediately conclude that `e1` cannot appear anywhere in the remainder of `set2` and hence is not in the intersection. Thus, we no longer need to consider `e1`; we discard it and proceed to the next element of `set1`. Similar logic advances through the elements of `set2` when `e2 < e1`. Here is the function:

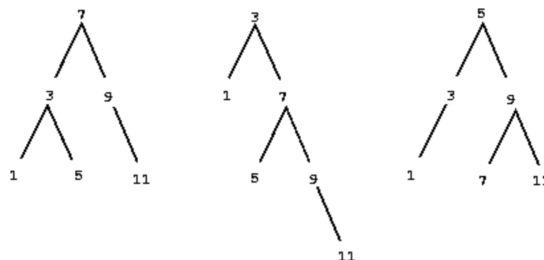
```
>>> def intersect_set(set1, set2):
        if empty(set1) or empty(set2):
            return Rlist.empty
        else:
            e1, e2 = set1.first, set2.first
            if e1 == e2:
                return Rlist(e1, intersect_set(set1.rest, set2.rest))
            elif e1 < e2:
                return intersect_set(set1.rest, set2)
            elif e2 < e1:
                return intersect_set(set1, set2.rest)

>>> intersect_set(s, s.rest)
Rlist(2, Rlist(3))
```

To estimate the number of steps required by this process, observe that in each step we shrink the size of at least one of the sets. Thus, the number of steps required is at most the sum of the sizes of `set1` and `set2`, rather than the product of the sizes, as with the unordered representation. This is $\Theta(n)$growth rather than $\Theta(n^2)$-- a considerable speedup, even for sets of moderate size. For example, the intersection of two sets of size 100 will take around 200 steps, rather than 10,000 for the unordered representation.

Adjunction and union for sets represented as ordered sequences can also be computed in linear time. These implementations are left as an exercise.

Sets as binary trees. We can do better than the ordered-list representation by arranging the set elements in the form of a tree. We use the `Tree` class introduced previously. The `entry` of the root of the tree holds one element of the set. The entries within the `left` branch include all elements smaller than the one at the root. Entries in the `right` branch include all elements greater than the one at the root. The figure below shows some trees that represent the set `{1, 3, 5, 7, 9, 11}`. The same set may be represented by a tree in a number of different ways. The only thing we require for a valid representation is that all elements in the `left` subtree be smaller than the tree `entry` and that all elements in the `right` subtree be larger.



The advantage of the tree representation is this: Suppose we want to check whether a value `v` is contained in a set. We begin by comparing `v` with `entry`. If `v` is less than this, we know that we need only search the `left` subtree; if `v` is greater, we need only search the `right` subtree. Now, if the tree is "balanced," each of these subtrees will be about half the size of the original. Thus, in one step we have reduced the problem of searching a tree of size n to searching a tree of size $\frac{n}{2}$. Since the size of the tree is halved at each step, we should expect that the number of steps needed to search a tree grows as *Theta(log n)*. For large sets, this will be a significant speedup over the previous representations. This

`set_contains` function exploits the ordering structure of the tree-structured set.

```
>>> def set_contains(s, v):
        if s is None:
            return False
        elif s.entry == v:
            return True
        elif s.entry < v:
            return set_contains(s.right, v)
        elif s.entry > v:
            return set_contains(s.left, v)
```

Adjoining an item to a set is implemented similarly and also requires $\Theta(\log n)$ steps. To adjoin a value `v`, we compare `v` with `entry` to determine whether `v` should be added to the `right` or to the `left` branch, and having adjoined `v` to the appropriate branch we piece this newly constructed branch together with the original `entry` and the other branch. If `v` is equal to the `entry`, we just return the node. If we are asked to adjoin `v` to an empty tree, we generate a `Tree` that has `v` as the `entry` and empty `right` and `left` branches. Here is the function:

```
>>> def adjoin_set(s, v):
        if s is None:
            return Tree(v)
        elif s.entry == v:
            return s
        elif s.entry < v:
            return Tree(s.entry, s.left, adjoin_set(s.right, v))
        elif s.entry > v:
            return Tree(s.entry, adjoin_set(s.left, v), s.right)

>>> adjoin_set(adjoin_set(adjoin_set(None, 2), 3), 1)
Tree(2, Tree(1), Tree(3))
```

Our claim that searching the tree can be performed in a logarithmic number of steps rests on the assumption that the tree is "balanced," i.e., that the left and the right subtree of every tree have approximately the same number of elements, so that each subtree contains about half the elements of its parent. But how can we be certain that the trees we construct will be balanced? Even if we start with a balanced tree, adding elements with `adjoin_set` may produce an unbalanced result. Since the position of a newly adjoined element depends on how the element compares with the items already in the set, we can expect that if we add elements "randomly" the tree will tend to be balanced on the average.

But this is not a guarantee. For example, if we start with an empty set and adjoin the numbers 1 through 7 in sequence we end up with a highly unbalanced tree in which all the left subtrees are empty, so it has no advantage over a simple ordered list. One way to solve this problem is to define an operation that transforms an arbitrary tree into a balanced tree with the same elements. We can perform this transformation after every few `adjoin_set` operations to keep our set in balance.

Intersection and union operations can be performed on tree-structured sets in linear time by converting them to ordered lists and back. The details are left as an exercise.

Python set implementation. The `set` type that is built into Python does not use any of these representations internally. Instead, Python uses a representation that gives constant-time membership tests and adjoin operations based on a technique called *hashing*, which is a topic for another course. Built-in Python sets cannot contain mutable data types, such as lists, dictionaries, or other sets. To allow for nested sets, Python also includes a built-in immutable `frozenset` class that shares methods with the `set` class but excludes mutation methods and operators.

Continue: 2.8 Generic Operations

Composing Programs by John DeNero, based on the textbook Structure and Interpretation of Computer Programs by Harold Abelson and Gerald Jay Sussman, is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

File failed to load: https://s3.amazonaws.com/saylordotorg-resources/CS/CS102/2.7%20Recursive%20Data%20Structures_files/extensions/jsMath2jax.js