

## Ordered Sets

---

## Announcements

# Sets

## Sets

never trust the order

One more built-in Python container type

- Set literals are enclosed in braces
- Duplicate elements are removed on construction
- Sets have arbitrary order, just like dictionary entries

```
>>> s = {'one', 'two', 'three', 'four', 'four'}
>>> s
{'three', 'one', 'four', 'two'}
>>> 'three' in s
True
>>> len(s)
4
>>> s.union({'one', 'five'})
{'three', 'five', 'one', 'four', 'two'}
>>> s.intersection({'six', 'five', 'four', 'three'})
{'three', 'four'}
>>> s
{'three', 'one', 'four', 'two'}
```

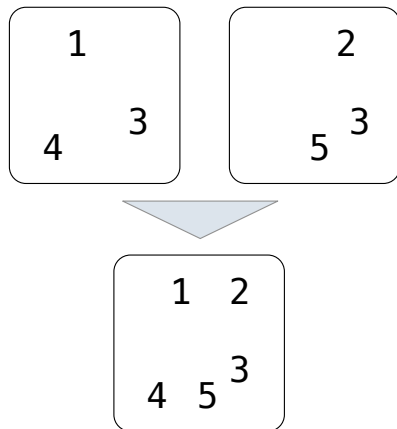
```
>>> s = {1, 2, 3}
>>> s
{1, 2, 3}
>>> s.add('four')
>>> s
{1, 2, 3, 'four'}
>>> s.intersection({3, 'four', 'five'})
{'four', 3}
>>> {[2]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
should not contain mutable values,
just like dict
```

## Implementing Sets

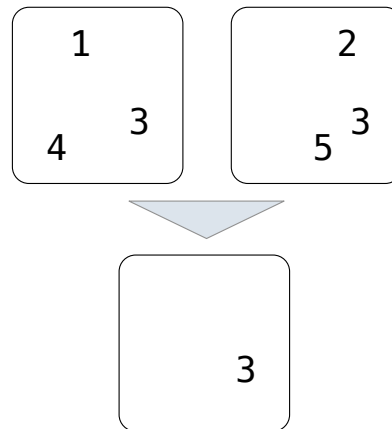
What we should be able to do with a set:

- **Membership testing:** Is a value an element of a set?
- **Union:** Return a set with all elements in set1 or set2
- **Intersection:** Return a set with any elements in set1 and set2
- **Adjoin:** Return a set with all elements in s and a value v

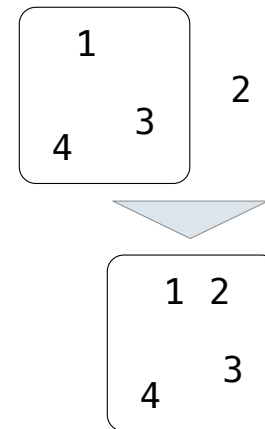
**Union**



**Intersection**



**Adjoin**



not applicable in the built-in set, which changes the original set (add) rather than creating a new set

## Sets as Linked Lists

## Sets as Unordered Sequences

**Proposal 1:** A set is represented by a linked list that contains no duplicate items.

**Time order of growth**

```
def empty(s):  
    return s is Link.empty
```

$\Theta(1)$

```
def contains(s, v):  
    """Return whether set s contains value v.
```

*Time depends on whether  
& where v appears in s.*

```
    >>> s = Link(1, Link(3, Link(2)))  
    >>> contains(s, 2)  
    True  
    """
```

$\Theta(n)$

*In the worst case: v  
does not appear in s*

**or**

*In the average case: appears  
in a uniformly distributed  
random location*

```
def contains(s, v):  
    """Return true if set s contains value v as an element.  
  
    >>> s = Link(1, Link(3, Link(2)))  
    >>> contains(s, 2)  
    True  
    >>> contains(s, 5)  
    False  
    """  
  
    if empty(s):  
        return False  
    elif s.first == v:  
        return True  
    else:  
        return contains(s.rest, v)
```

## Sets as Unordered Sequences

Time order of worst-case growth

```
def adjoin(s, v):  
    if contains(s, v):  
        return s  
    else:  
        return Link(v, s)
```

$\Theta(n)$

The size of the set

```
def intersect(s, t):  
    if s is Link.empty:  
        return Link.empty  
    rest = intersect(s.rest, t)  
    if contains(t, s.first):  
        return Link(s.first, rest)  
    else:  
        return rest
```

for every value in set1 ( $n$ ),  
there is a contains of  
value in set2 ( $n$ )

$\Theta(n^2)$

If sets are  
the same size

```
def intersect(set1, set2):  
    in_set2 = lambda v: contains(set2, v)  
    return filter_link(in_set2, set1)
```

Return elements  $x$  for which  
 $\text{in\_set2}(x)$  returns a true value

```
def union(set1, set2):  
    not_in_set2 = lambda v: not contains(set2, v)  
    set1_not_set2 = filter_link(not_in_set2, set1)  
    return extend_link(set1_not_set2, set2)
```

$\Theta(n^2)$   
 $\Theta(n)$

$\Theta(n^2)$   
 $\Theta(n^2+n) = \Theta(n^2)$



## Sets as Ordered Linked Lists

## Sets as Ordered Sequences

**Proposal 2:** A set is represented by a linked list with unique elements that is *ordered from least to greatest*

Parts of the program that...	Assume that sets are...	Using...
Use sets to contain values	Unordered collections	<code>empty, contains, adjoin, intersect, union</code>
Implement set operations	Ordered linked lists	<code>first, rest, &lt;, &gt;, ==</code>

*Different parts of a program may make different assumptions about data*

## Searching an Ordered List

```
>>> s = Link(1, Link(3, Link(5)))
>>> contains(s, 1)
True
>>> contains(s, 2)
False
>>> t = adjoin(s, 2)
```

**Operation**

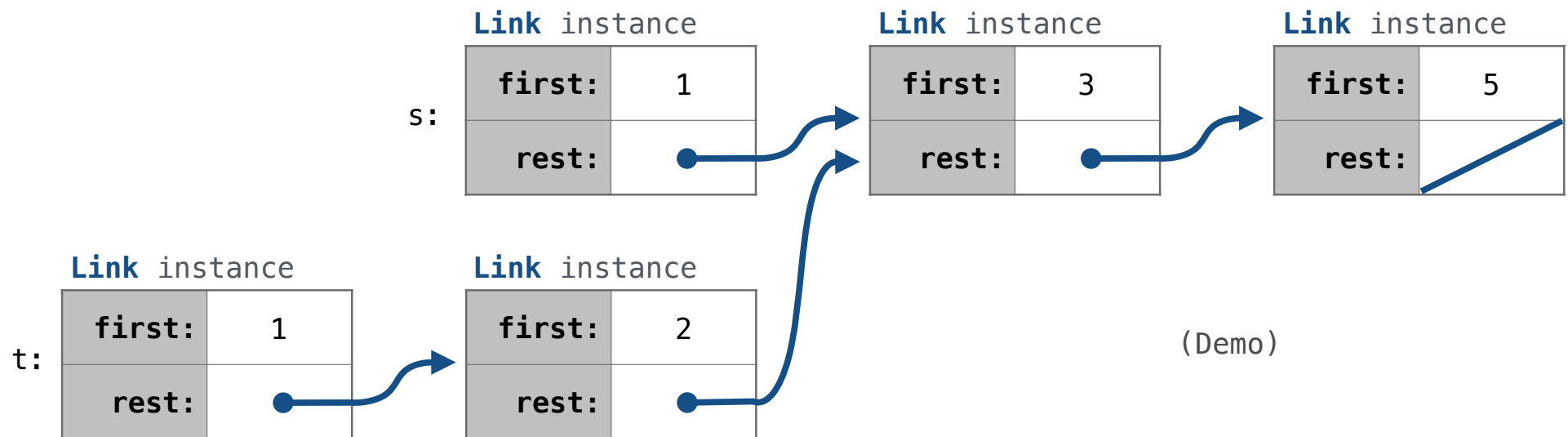
contains

adjoin

**Time order of growth**

$\Theta(n)$

$\Theta(n)$



## Set Operations

## Intersecting Ordered Linked Lists

**Proposal 2:** A set is represented by a linked list with unique elements that is *ordered from least to greatest*

```
def intersect(s, t):
    if empty(s) or empty(t):
        return Link.empty
    else:
        e1, e2 = s.first, t.first
        if e1 == e2:
            return Link(e1, intersect(s.rest, t.rest))
        elif e1 < e2:
            return intersect(s.rest, t)
        elif e2 < e1:
            return intersect(s, t.rest)
```

```
def union(set1, set2):
    if empty(set1):
        return set2
    elif empty(set2):
        return set1
    else:
        e1, e2 = set1.first, set2.first
        if e1 == e2:
            return Link(e1, union(set1.rest, set2.rest))
        elif e1 < e2:
            return Link(e1, union(set1.rest, set2))
        elif e2 < e1:
            return Link(e2, union(set1, set2.rest))
```

analogous

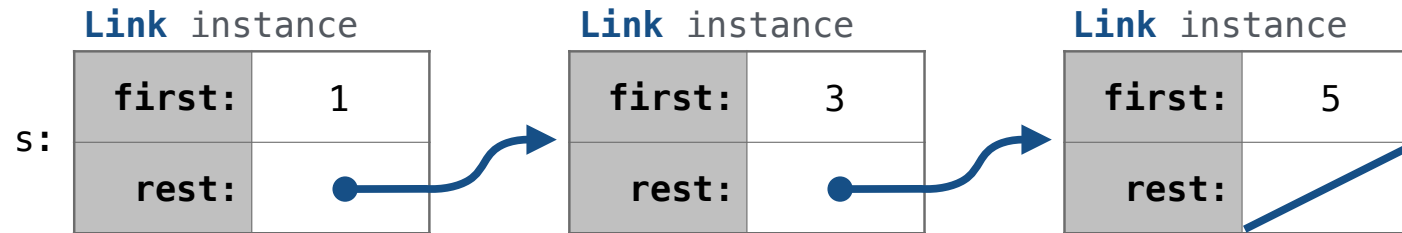
a similar union

Order of growth? If  $s$  and  $t$  are sets of size  $n$ , then  $\Theta(n)$

(Demo)

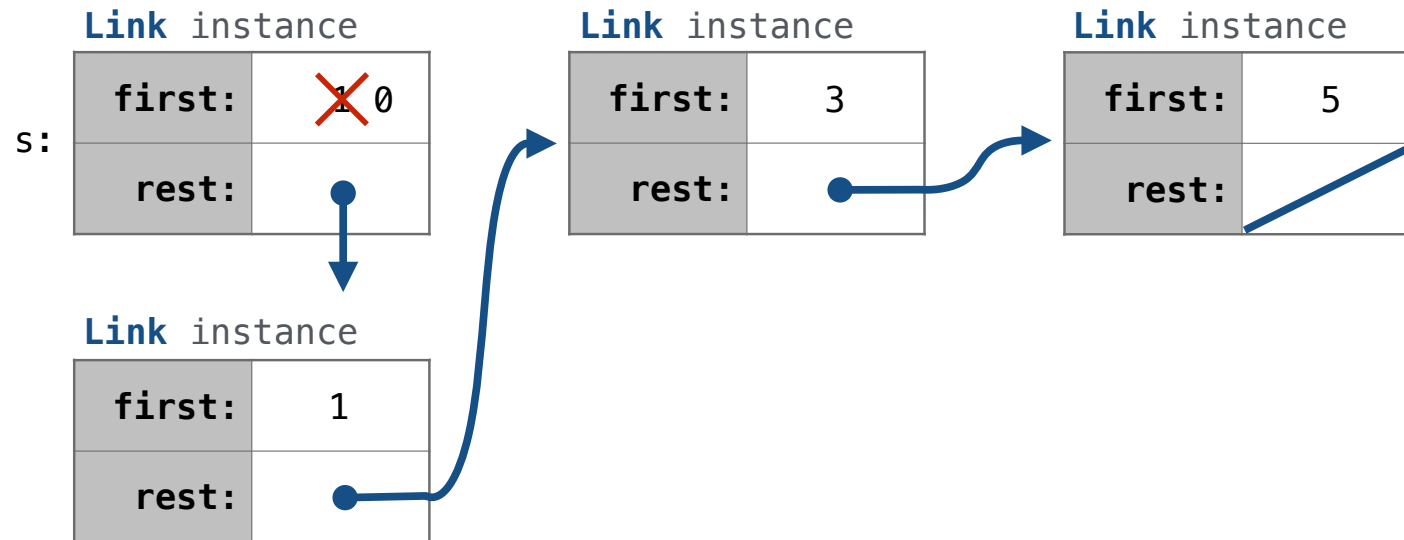
## Set Mutation

## Adding to an Ordered List



`add(s, 0)` Try to return the same object as input

## Adding to an Ordered List

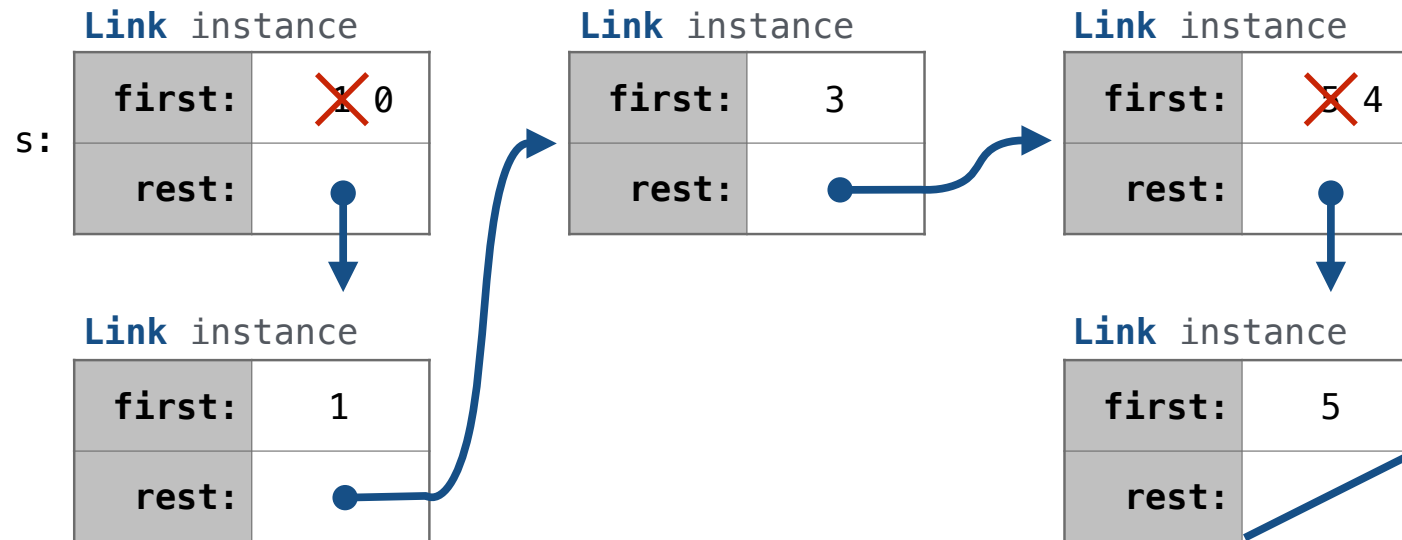


```
add(s, 3)
```

```
add(s, 4)
```

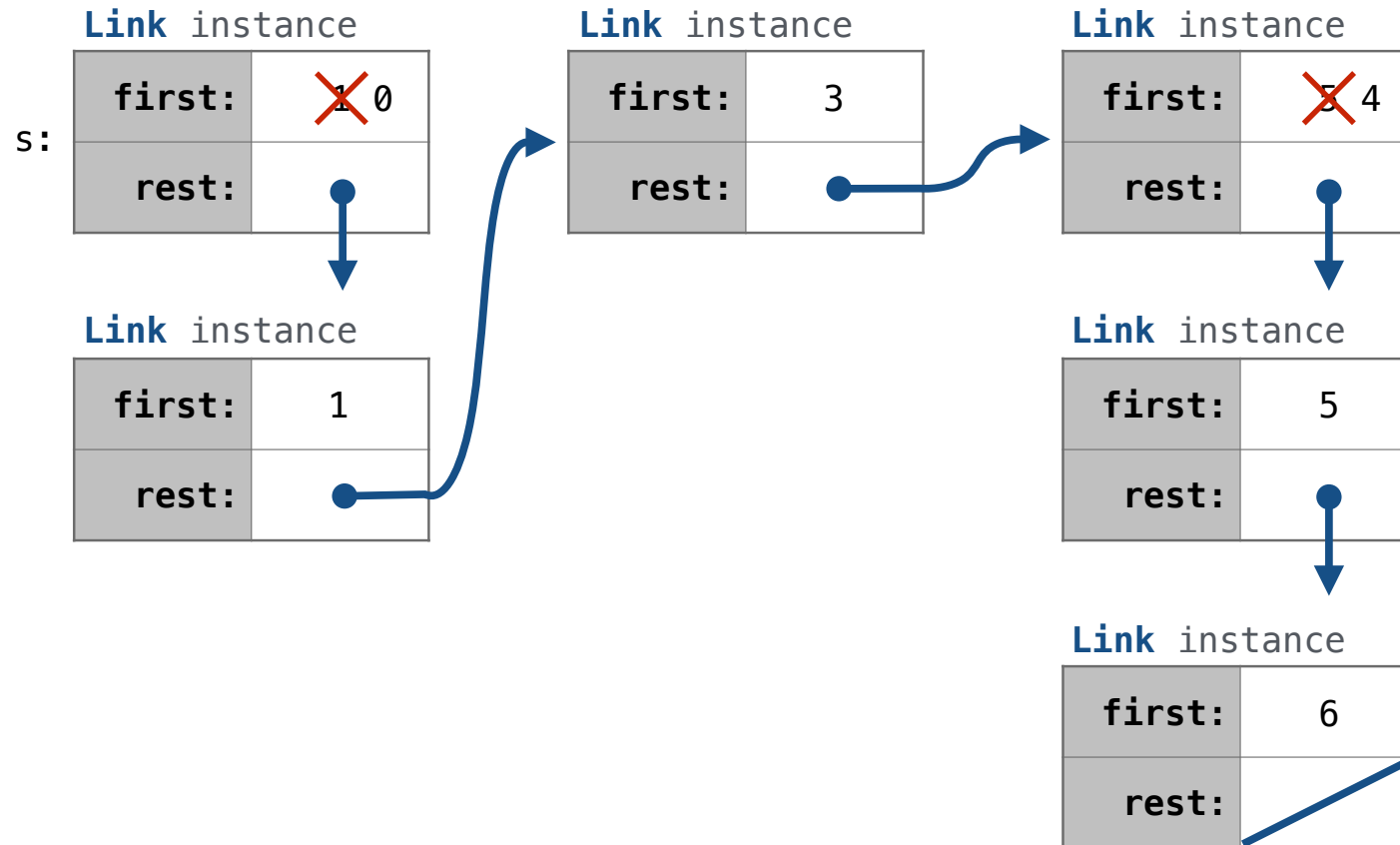


## Adding to an Ordered List



`add(s, 6)`

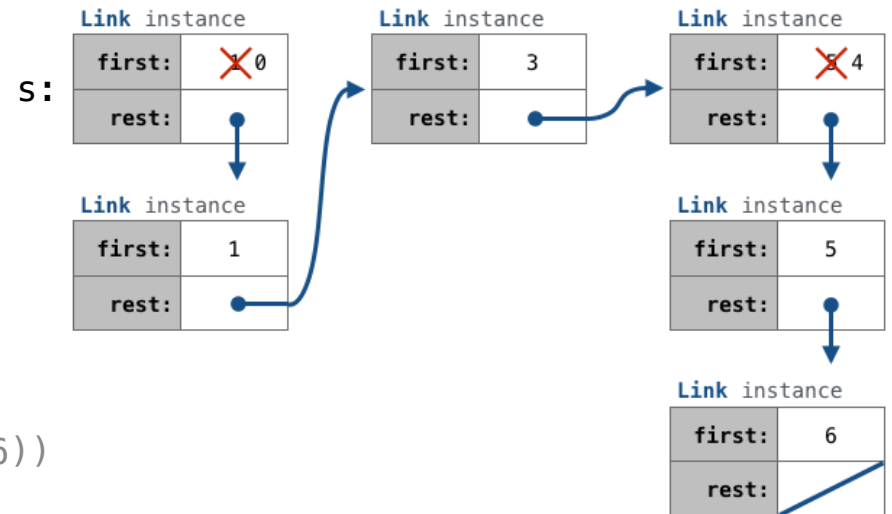
## Adding to an Ordered List



## Adding to a Set Represented as an Ordered List

```
def add(s, v):
    """Add v to a set s, returning modified s."""

    >>> s = Link(1, Link(3, Link(5)))
    >>> add(s, 0)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 3)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 4)
    Link(0, Link(1, Link(3, Link(4, Link(5)))))
    >>> add(s, 6)
    Link(0, Link(1, Link(3, Link(4, Link(5, Link(6)))))
    """
```



```
if empty(s): return Link(v)
```

```
if s.first > v:
```

```
    s.first, s.rest = v, Link(s.first, s.rest)
```

```
elif s.first < v and empty(s.rest):
```

```
    s.rest = Link(v, s.rest)
```

```
elif s.first < v:
```

```
    add(s.rest, v)
```

```
return s
```

Note: statements like `s.first = s` will cause error