Object-Oriented Programming

Overview of Python OOP

```
d = defaultdict(lambda : {
        "count": 0,
        "lines": set()
    })
```

- Motivations of OOP
  - complicated data structures
  - modularity
- Perl does not have built-in OOP
  - Perl + OOP ==> Ruby (pure OO, like Java)
- Python has OOP from the very beginning
  - hybrid approach (like C++ but unlike Java)
  - nearly everything inside a class is public
  - explicit argument `self`

Classes

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def norm(self):
        return self.x ** 2 + self.y ** 2
```

- "self" is like "this" pointer in C++/Java/C#/PHP
- constructor `__init__(self, ...)`
- every (new-style) class is a subclass of Object like Java
  - we will only use new-style classes in this course

```
>>> p = Point (3, 4)
>>> p.x
3
>>> p.norm()
25
```

why do we need this "self"?

```
class Point(object):
    def norm(self):
        return self.x ** 2 + self.y ** 2
```

- unlike implicit "this", the first argument is explicit in Python
  - thus doesn't need to be "self" (but usually, after Smalltalk)

```
float Point::norm() {
  return this->x**2 + this->y**2;
}

is semantically equivalent to:

float norm(const Point *this) {
  return this->x**2 + this->y**2;
}

but C++/Java can't transform a
global function into a method
while Python can!
```

```
def norm_global(point):
   return point.x**2 + point.y**2

>>> Point.norm2 = norm_global
>>> p = Point(3, 4); print p.norm2()
25

>>> pnorm = p.norm      # functional!
>>> pnorm() == p.norm()
True
```

calling `p.xxx(...)` is internally transformed to `Point.xxx(p, ...)`

Member variables

- each instance has its own hashmap for variables!
- you can add new fields on the fly (weird... but handy...)

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "(%s, %s)" % (self.x, self.y)
```

```
>>> p = Point (5, 6)
>>> p.z = 7
>>> print p
(5, 6)
>>> p.z
7
>>> print p.w
AttributeError - no attribute 'w'
>>> p["x"] = 1
AttributeError - no attribute 'setitem'
```

More efficient: __slots__

- like C++/Java: fixed list of member variables
- class-global hash: all instances of this class share this hash
  - can't add new variables to an instance on the fly

```
class Point(object):
    __slots__ = "x", "y"

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        " like toString() in Java "
        return "(%s, %s)" % (self.x, self.y)

>>> p = Point(5, 6)
>>> p.z = 7
AttributeError!
```

Special function `__str__`

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def norm(self):
        return self.x ** 2 + self.y ** 2

    def __str__(self):
        return "(%s, %s)" % (self.x, self.y)
```

```
>>> P = Point(3, 4)
>>> p.__str__()
'(3, 4)'
>>> Point.__str__(p)
'(3, 4)'
>>> str(p)
'(3, 4)'
>>> print p
(3, 4)
```

```
print p
=> str(p)
=> p.__str__()
=> Point.__str__(p)
```

Special functions: str vs repr

```
class Point(object):
   def __init__(self, x, y):
      self.x = x
      self.y = y

   def __str__(self):
      return "(%s, %s)" % (self.x, self.y)
```

```
>>> p = Point(3,4)
>>> print p
(3, 4)
>>> p
<__main__.Point instance at 0x38be18>
```

Special functions: str vs repr

```
class Point (object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "(%s, %s)" % (self.x, self.y)

    def __repr__(self):
        return self.__str__()
```

```
>>> p = Point(3,4)
>>> print p
(3, 4)
>>> p
<__main__.Point instance at 0x38be18>
```

```
>>> p
(3, 4)
>>> repr(p)
(3, 4)
```

Special functions: str vs repr

```
class Point (object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

when __str__ is not defined, __repr__ is used
if __repr__ is not defined, Object.__repr__ is used

```
>>> p = Point(3,4)
>>> print p
<__main__.Point instance at 0x38be18>
>>> p
<__main__.Point instance at 0x38be18>
```

Special functions: str vs repr

```
class Point (object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "(%s, %s)" % (self.x, self.y)

    __repr__ = __str__
```

(functional assignment)

```
>>> p = Point(3,4)
>>> print p
(3, 4)
>>> p
<__main__.Point instance at 0x38be18>
```

```
>>> p
(3, 4)
>>> repr(p)
(3, 4)
```

Special functions: cmp

```
class Point (object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "(%s, %s)" % (self.x, self.y)
```

by default,
Python class object comparison
is by pointer! define __cmp__!

```
>>> p = Point(3,4)
>>> Point (3,4) == Point (3,4)
False
```

Special functions: cmp

```
class Point (object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "(%s, %s)" % (self.x, self.y)

    def __cmp__(self, other):
        if self.x == other.x:
            return self.y - other.y
        return self.x - other.x
```

if __eq__ is not defined, __cmp__ is used;
if __cmp__ is not defined, Object.__cmp__ is used (by reference)

```
>>> p = Point(3,4)
>>> p
<__main__.Point instance at 0x38be18>
>>> Point (3,4) == Point (3,4)
False
```

```
>>> cmp(Point(3,4), Point(4,3))
-1

=> Point(3,4).__cmp__(Point(4,3))
=> Point.__cmp__(Point(3,4), Point(4,3))

>>> Point (3,4) == Point (3,4)
True
```

Special functions: cmp

```
class Point (object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "(%s, %s)" % (self.x, self.y)

    def __cmp__(self, other):
        c = cmp(self.x, other.x)
        return c if c != 0 else cmp(self.y, other.y)
```

or `__cmp__ = lambda p, q: cmp((p.x, p.y), (q.x, q.y))`

or
```
tuple = lambda p: (p.x, p.y)
__cmp__ = lambda p, q: cmp(p.tuple(), q.tuple())
```

```
>>> p = Point(3,4)
>>> p
<__main__.Point instance at 0x38be18>
>>> Point (3,4) == Point (3,4)
False
```

```
>>> cmp(Point(3,4), Point(4,3))
-1

=> Point(3,4).__cmp__(Point(4,3))
=> Point.__cmp__(Point(3,4), Point(4,3))

>>> Point (3,4) == Point (3,4)
True
```

Special functions: cmp

```
class Point (object):
    def __init__(self, x, y):
        self.x, self.y = x, y
    norm = lambda p: p.x**2 + p.y**2
    __len__ = norm
    __tuple__ = lambda p: (p.x, p.y)
    __cmp__ = lambda p, q: cmp(p.__tuple__(), q.__tuple__())
```

```
>>> cmp(Point(3,4), Point(4,3))
-1

>>> len(Point(3,4))
25

>>> Point(3,4).__tuple__()
(3, 4)

>>> tuple(Point(3,4))
TypeError: iteration over non-sequence
```

```
>>> p = Point(3,4)
>>> p
<__main__.Point instance at 0x38be18>
>>> Point (3,4) == Point (3,4)
False
```

__tuple__ is not a special function

unique signature for each method

```
class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __init__(self, (x, y)):
        self.x, self.y = x, y
```

- no polymorphic functions (earlier defs will be shadowed)
  - ==> only one constructor (and no destructor)
  - each function can have only one signature
    - because Python is dynamically typed

class docstring

```
class Point(object):
    '''A 2D point'''
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __str__(self):
        '''like toString() in Java'''
        return "(%s, %s)" % (self.x, self.y)
```

- like Javadoc (Python pydoc)

```
>>> Point.__doc__
"A 2D point"
>>> help(Point)
"A 2D point"

>>> p.__str__.__doc__
"like toString() in Java"
>>> help(p.__str__)
"like toString() in Java"
```

Inheritance

```
class Point (object):
    ...
    def __str__(self):
        return str(self.x) + ", " + str(self.y)
    ...

class Point3D (Point):
    "A 3D point"
    def __init__(self, x, y, z):
        Point.__init__(self, x, y)
        self.z = z

    def __str__(self):
        return Point.__str__(self) + ", " + str(self.z)

    def __cmp__(self, other):
        tmp = Point.__cmp__(self, other)
        return tmp if tmp != 0 else self.z - other.z
```

super-class, like C++
(multiple inheritance allowed)

__slots__ in inheritance

- like C++/Java: fixed list of member variables
- class-global hash: can't add new field on the fly

```
class Point(object):
    __slots__ = "x", "y"

    def __init__(self, x, y):
        self.x, self.y = x, y

class Point3D(Point):
    __slots__ = "z"

    def __init__(self, x, y, z):
        Point.__init__(self, x, y)
        self.z = z

>>> p = Point3D(5, 6, 7)
>>> p.z = 7
```

super()

- cast self into an instance of its super class type
- i.e. super returns an instance, not a type or class!

  `Point.__init__(self, x, y)` vs `super(Point3D, self).__init__(x, y)`

```
class Point(object):
    __slots__ = "x", "y"

    def __init__(self, x, y):
        self.x, self.y = x, y

class Point3D(Point):
    __slots__ = "z"

    def __init__(self, x, y, z):
        super(Point3D, self).__init__(x, y)
        self.z = z

>>> p = Point3D(5, 6, 7)
>>> p.z = 7
```

HW 2

- out by the end of today (Oct 18)
  - a rational number class: supports all arithmetic operators
  - a BST-map class: insertion, deletion, replace, iteration...
  - a priority queue class (will be reused in HW 3)
  - a wrapper/decorator question (functional programming)
- due in 2 weeks on Friday Nov 2
- Discussions/Review on Tuesday Nov 6
- Midterm (15%) on Thursday Nov 8 -- mostly from HW2

HW 2 rational numbers class

- all results are automatically reduced to the smallest terms

```
>>> from rational import Rational
>>> Rational(1,2) - Rational(2,1)
-3/2
>>> Rational(1,2) * Rational(2,3)
1/3                             # automatically reduced
>>> print Rational(1,2) / ~Rational(2,-4)  # __inv__
-1/4
>>> p = Rational(1,2)
>>> p *= Rational(2,5)
>>> print p
1/5                             # automatically reduced
>>> 5 + (-Rational(1,2))        # __neg__
9/2
>>> Rational(2,4) ** 10         # __pow__
1/1024
```

HW 2 BST-map class

- BST as a "map" like C++ STL's map (red-black tree)
- self-balancing is optional in HW2 (extra credit)
- automatic insertion of missing key is required (defaultdict)

```
>>> from bst import BST
>>> t = BST(missing=int)
>>> t["a"] = 5
>>> print t["b"]
0
>>> len(t)
2
>>> del t["a"]
>>> t["a"]
0
>>> [x for x in t]    # iteration, or t.keys()
["a", "b"]
```

Operator Overloading

- like operator overloading in C++
- special functions like `__add__()`, `__mul__()`

```
class Point (object):
  # previously defined methods here...

  def __add__(self, other):
    return Point(self.x + other.x, self.y + other.y)

  def __mul__(self, other):
    return self.x * other.x + self.y * other.y
```

dot-product

```
Point & Point::operator+(const Point &other) {
  return Point(this->x + other.x, this.y + other.y);
}
```

```
>>> p = Point (3, 4)
>>> q = Point (5, 6)
>>> print (p + q)
(8, 10)
>>> print (p * q)
39
```

mul vs. rmul

```
class Point (object):

    def __mul__(self, other):
        return self.x * other.x + self.y * other.y

    def __rmul__(self, other):
        return Point(other * self.x,  other * self.y)
```

scalar-multiplication

```
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> print p1 * p2
43
>>> print 2 * p2
(10, 14)

>>> print p2 * 2
AttributeError: 'int' object has no attribute 'x'
```

When is __rmul__ invoked?

- when the left object doesn't define __mul__
  - or when the left object's __mul__ returns `NotImplemented`
- and right object defines __rmul__

```
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> print p1 * p2
43
>>> print 2 * p2
(10, 14)
>>> dir(2)
...  __mul__ ...
```

Note: int.__mul__ exists, but returns NotImplemented when the right argument is not int or float

```
class Foo(object):
    def __init__(self, val):
        self.val = val

class Bar(object):
    def __init__(self, val):
        self.val = val

    def __rmul__(self, other):
        return Bar(self.val * other.val)

f = Foo(4)
b = Bar(6)

obj = f * b    # Bar [24]
obj2 = b * f   # ERROR
```

__mul__ v2: check operand type

```
    def __mul__(self, other):                                  # v1
        return self.x * other.x + self.y * other.y

    def __rmul__(self, other):
        return Point(other * self.x,  other * self.y)
```

```
    def __mul__(self, other):                                  # v2
        if type(other) is Point:
            return self.x * other.x + self.y * other.y
        return NotImplemented
```

```
... using __mul__ v1 ...
>>> p1 = Point(3, 4)
>>> print p1 * 2
AttributeError: 'int' object has no attribute 'x'

... after __mul__ v2 ...
>>> p1 = Point(3, 4)
>>> print p1 * 2
unsupported operand type(s)
```

__mul__ v3: call __rmul__

```
    def __mul__(self, other):                          # v3
        if type(other) is Point:
            return self.x * other.x + self.y * other.y
        return self.__rmul__(other)

    def __rmul__(self, other):                         # v3
        if type(other) in [int, float]:
            return Point(other * self.x,  other * self.y)
        return NotImplemented
```

```
... after __mul__ v2 ...
>>> p1 = Point(3, 4)
>>> print p1 * 2
unsupported operand type(s)
```

```
... after __mul__ v3 ...
>>> print p1 * 2
(6, 8)
>>> print p2 * [2]
([2, 2, 2], [2, 2, 2, 2])      ???
```

```
... after __rmul___ v3 ...
>>> print p2 * [2]
unsupported operand type(s)
```

__iadd__ (+=) and __neg__ (-)

```
class Point(object):

   def __add__(self, other):
       return Point(self.x + other.x, self.y + other.y)

   def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self

    def __neg__(self):
        return Point(-self.x, -self.y)
```

must return `self` here!

```
>>> -Point(3,4)
(-3, -4)
>>> p = Point(3,4)
>>> p += p
>>> p
(6, 8)
```

```
add, sub, mul, div,
radd, rsub, rmul, rdiv,
iadd, isub, imul, idiv,
neg, inv, pow,
len, str, repr,
cmp, eq, ne, lt, le, gt, ge
```

Why __iadd__ must return self?

```
class Point(object):
  ...
  def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        # return self

>>> p = Point(3, 4)
>>> p += p
>>> p
>>> p is None
True
>>> p = Point(3, 4)
>>> p.__iadd__(p)
>>> p
(6, 8)
```

This shows a+=b *is* ***not*** *interpreted as* **a.__iadd__(b)**, *but rather* **a = a.__iadd__(b)**

```
// C++ tradition:

Point & Point::operator+=(const Point &other) {
    this -> x += other.x;
    this -> y += other.y;
    return *this;
}

Point a, b;
...
a += b;    // Same as a = a.operator+=(b)

(a += b) += b;
```

```
    # flexible += (return new copy):
    # great for immutable types

    class Point(object):
      ...
      def __iadd__(self, other):
          return Point(self.x + other.x,
                       self.y + other.y)

     # or just
     __iadd__ = __add__
```

Details of +=

- a += b => a = a.__iadd__(b) => a = a.__add__(b)
  => a = b.__radd__(a)

n-ary Trees

```
class Tree (object):
  __slots__ = “node”, “children”
  def __init__(self, node, children=[]):
    self.node = node
    self.children = children

def total(self):
  if self == None:
     return 0
  return self.node + sum([x.total() for x in self.children])

def pp(self, dep=0):
  print “ |“ * dep, self.node
  for child in self.children:
            child.pp(dep+1)

def __str__(self):
  return “(%s)” % “ “.join(map(str, \
          [self.node] + self.children))
```

```
left = Tree(2)
right = Tree(3)

>>> t = Tree(1, [Tree(2), Tree(3)])
>>> total(t)
6

>>> t.pp()
 1
 | 2
 | 3
>>> print t
(1 (2) (3))
```

Binary Search Tree v1

```
class BST(object):

    __slots__ = "val", "left", "right"

    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

    def __contains__(self, query):  # 5 in BST(...)
        c = cmp(query, self.val)
        if c == 0:
            return True
        elif c < 0:
            return self.left is not None and query in self.left
        else:
            return self.right is not None and query in self.right

    def list(self):
        llist = self.left.list() if self.left is not None else []
        rlist = self.right.list() if self.right is not None else []
        return llist + [self.val] + rlist

    def __str__(self):
        return "(%s %s %s)" % (self.left, self.val, self.right)
```

Using BST (optional args)

```
class BST(object):

    __slots__ = "val", "left", "right"

    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

   ...

>>> t = BST(3, BST(1, BST(2)), BST(4))
>>> 2 in t
False

........ why????

>>> t = BST(3, BST(1, None, BST(2)), BST(4))
>>> 2 in t
True

or

>>> t = BST(3, BST(1, right=BST(2)), BST(4))
>>> 2 in t
True
```

Binary Search Tree v2

```
class BST(object):

    __slots__ = "val", "left", "right"

    def __init__(self, val=None, left=None, right=None):
        if val is None:
            self.val = self.left = self.right = None
        else:
            self.val = val
            self.left = left if left is not None else BST()
            self.right = right if right is not None else BST()

    def __contains__(self, query):
        if self.val is None:
            return False
        c = cmp(query, self.val)
        if c == 0:
            return True
        return query in self.left if c < 0 else query in right

    def list(self):
        if self.val is None:
            return []
        return self.left.list() + [self.val] + self.right.list()

    def __str__(self):
        return "(%s %s %s)" % (self.left, self.val, self.right)
```

insertion in BST

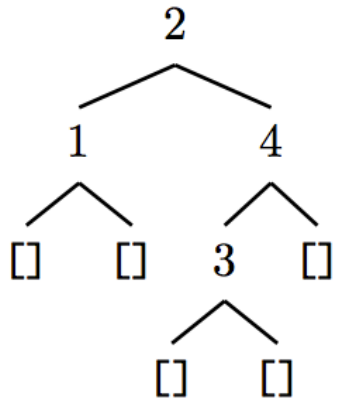
- modify __contains__ for both search and insert

```
class BST(object):

    __slots__ = "val", "left", "right"

    def __init__(self, val=None, left=None, right=None):
        if val is None:
            self.val = self.left = self.right = None
        else:
            self.val = val
            self.left = left if left is not None else BST()
            self.right = right if right is not None else BST()

    def __contains__(self, query):
        if self.val is None:
            return False
        c = cmp(query, self.val)
        if c == 0:
            return True
        return query in self.left if c < 0 else query in right
```

insertion in BST

- modify __contains__ for both search and insert

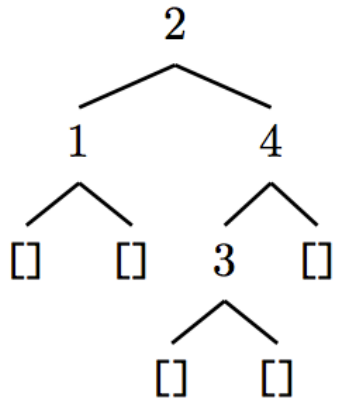
```
class BST(object):

    ...

    def search(self, query):
        if self.val is None:
            return False, self
        c = cmp(query, self.val)
        if c == 0:
            return True, self
        elif c < 0:
            return self.left.search(query)
        return self.right.search(query)

    def __contains__(self, query):      # 5 in t => bool(t.__contains__(5))
        found, _ =  self.search(query)
        return found

    def insert(self, query):
        found, node = self.search(query)
        if not found:
            node.val = query        # not node = BST(query), must be in-place!
            node.left = node.right = BST()  # wrong! why?
```



insertion in BST

- modify __contains__ for both search and insert

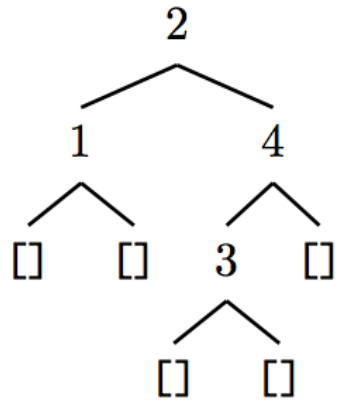
```
class BST(object):

    ...

    def search(self, query):
        if self.val is None:
            return False, self
        c = cmp(query, self.val)
        if c == 0:
            return True, self
        elif c < 0:
            return self.left.search(query)
        return self.right.search(query)

    def __contains__(self, query):      # 5 in t => bool(t.__contains__(5))
        found, _ =  self.search(query)
        return found

    def insert(self, query):
        found, node = self.search(query)
        if not found:
            node.val = query        # not node = BST(query), must be in-place!
            node.left, node.right = BST(), BST()   # correct, but lengthy
```



insertion in BST

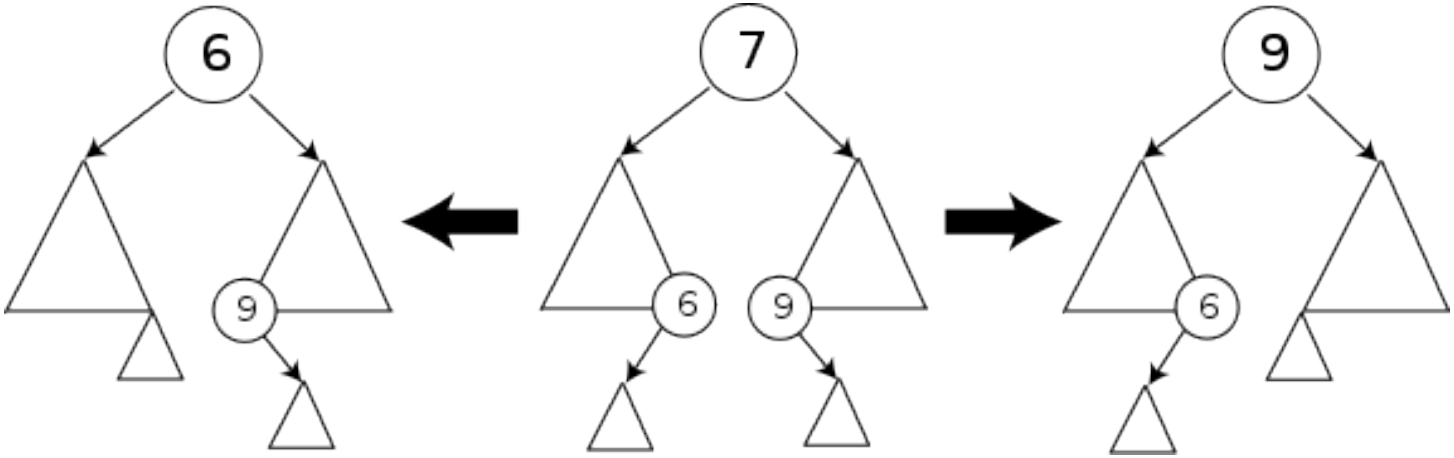
- modify __contains__ for both search and insert

```
class BST(object):

    ...

    def search(self, query):
        if self.val is None:
            return False, self
        c = cmp(query, self.val)
        if c == 0:
            return True, self
        elif c < 0:
            return self.left.search(query)
        return self.right.search(query)

    def __contains__(self, query):      # 5 in t => bool(t.__contains__(5))
        found, _ =  self.search(query)
        return found

    def insert(self, query):
        found, node = self.search(query)
        if not found:
            node.__init__(query)    # in-place re-initialization
```



deletion in BST

- easy case: node has no left or no right child
- hard case: node has both left and right child
  - choose either one neighbor in the inorder (sorted order) to replace the deleted node
  - i.e., smallest of the right or the largest of the left subtree



from unsorted list v1: global function

- "weird" qsort again
- how to make it a class method instead of a global func?

```
def fromlist(a):                          # v1: global function
    if a == []:
        return BST()                # or None if you use BST v1
    pivot = a[0]
    return BST(pivot,
               fromlist([x for x in a if x<pivot]),
               fromlist([x for x in a[1:] if x>=pivot]))
```

```
>>> from bst2 import BST, fromlist
>>> BST.fromlist = fromlist
>>> BST.fromlist([2,1,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unbound method fromlist() must be called with BST
instance as first argument (got list instance instead)
```

from unsorted list v2: instance method

- "weird" qsort again
- how to make it a class method instead of a global func?
  - it should be a static or class method, not instance method!

```
class BST(object):

   ...

   def fromlist(self, a):                          # v2: instance method
       if a == []:
           return BST()
       pivot = a[0]
       return BST(pivot,
                  fromlist(self, [x for x in a if x<pivot]),
                  fromlist(self, [x for x in a[1:] if x>=pivot]))

>>> print BST().fromlist([2,1,3])
((None 1 None) 2 (None 3 None))
```

from unsorted list v3: static method

- the @staticmethod "decorator"
- we'll see many more decorators later and we'll explain it

```
class BST(object):

   ...

   @staticmethod
   def fromlist(a):                              # v3: static method
       if a == []:
           return BST()
       pivot = a[0]
       return BST(pivot,
                  fromlist([x for x in a if x<pivot]),
                  fromlist([x for x in a[1:] if x>=pivot]))

>>> print BST.fromlist([2,1,3])
((None 1 None) 2 (None 3 None))
```

from unsorted list v4: class method

- static method does not take any implicit argument (Java/C++)
- class method takes the class as the first argument (Python!)
  - useful in subclassing, so class method is recommended!

```
class BST(object):

  ...

  @classmethod
  def fromlist2(cls, a):
      if a == []:
          return cls()
      pivot = a[0]
      return cls(pivot,
                 fromlist2([x for x in a if x<pivot]),
                 fromlist2([x for x in a[1:] if x>=pivot]))

>>> print BST.fromlist2([2,1,3])  # same usage, no need to pass cls
((None 1 None) 2 (None 3 None))
```

dict.fromkeys is a classmethod

```
>>> class DictSubclass(dict):
...     pass
...
>>> dict.fromkeys("abc")
{'a': None, 'c': None, 'b': None}
>>> type(DictSubclass.fromkeys("abc"))
DictSubclass
```

static attribute variable

- class-level information shared by all instances
- we have already seen one: __slots__

```
class Point(object):

    num = 0

    def __init__(self, x, y):
        self.x = x
        self.y = y
        Point.num += 1
```

good style

or

```
class Point(object):

    num = 0

    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.num += 1
```

bad style

What really is a decorator doing?

- decorator is nothing magic: it is a shorthand for a function that takes a function and returns a decorated function
- `staticmethod` and `classmethod` are built-in function mappers
- you can define your own decorator functions

```
class E(object):
     def f(x):
          print x
     f = staticmethod(f)
```

==

```
class E(object):
     @staticmethod
     def f(x):
          print x
```

```
>>> print E.f(3)
3
>>> print E().f(3)
3
```

```
>>> print E.f(3)
3
>>> print E().f(3)
3
```

custom decorator examples

- a decorator must return a function!

```
def enhanced(f):
    return lambda x: f(x)+1

def g(x):
    return x*2
g = enhanced(g)
```

```
def enhanced(f):
    return lambda x: f(x)+1

@enhanced
g = lambda x: x*2
```

```
enhanced = lambda f: lambda x: f(x)+1
g = enhanced(lambda x: x*2)
```

```
def enhanced(meth):
    def new(self, x):
        print "I am enhanced",
        return meth(self, x)
    return new

class C:
    def bar(self):
        print "bar method:", x
    bar = enhanced(bar)
```

```
def enhanced(meth):
    def new(self, x):
        print "I am enhanced",
        return meth(self, x)
    return new

class C:
    @enhanced
    def bar(self, x):
        print "bar method:", x
```

decorator chain

```
def makebold(f):
    def wrapped():
        return "<b>" + f() + "</b>"
    return wrapped

def makeitalic(f):
    def wrapped():
        return "<i>" + f() + "</i>"
    return wrapped

@makebold
@makeitalic
def hello():
    return "hello world"

print hello()
<b><i>hello world</i></b>
```

```
makebold   = lambda f: lambda: “<b>%s</b>” % f()
makeitalic = lambda f: lambda: “<i>%s</i>” % f()
hello = makebold(makeitalic(lambda: “hello”))
```

decorator with lexical closure

- functions can have "local storage" like static variables in C

```
def counter(f):
    num = 0
    def helper(x):
        num += 1       # num (local) = num + 1
        print "the function has been called", num, "times"
        return f(x)
    return helper

WRONG!!! local variable num referenced before assignment!

from collections import defaultdict
def counter2(f):
    num = defaultdict(int)
    def helper(x):
        num[f] += 1
        print "function", f.__name__, "called", num[f], "times"
        return f(x)
    return helper

fib = counter2(fib)
```

caution with global variable

```
a = 0
def count():
    a += 1

count()
UnboundLocalError: local variable 'a' referenced before assignment

a = []
def count():
    a += [1]

count()
UnboundLocalError: local variable 'a' referenced before assignment

a = 0
def count():
    global a
    a += 1
```

generic memoization decorator?

```
def fib(n, fibs={0:1, 1:1}):
    if n not in fibs:
        fibs[n] = fib(n-1, fibs) + fib(n-2, fibs)
    return fibs[n]
```

```
def memoized(f):
    cache = {}      # lexical closure (local storage)
    def helper(x):
        if x not in cache:
            cache[x] = f(x)
        return cache[x]
    return helper

def fib(n):
    return fib(n-1)+fib(n-2) if n>1 else 1

fib = memoized(fib) # not g = memoized(fib)

# or better ...
@memoized
def fib(n):
...
```

fib is recursive!

decorator with arbitrary list of args

- *args is a tuple of all arguments

```
>>> def addspam(fn):
...     def new(*args):
...         print "spam, spam, spam"
...         return fn(*args)
...     return new
...
>>> @addspam
... def useful(a, b):
...     print a**2 + b**2
...
>>> useful(3,4)
spam, spam, spam
25
```

*args is very useful

- packing/unpacking of arbitrary argument list

```
>>> def f(*a):
...   print a
...
>>> f(1)
(1,)
>>> f(1,2)
(1, 2)
>>> f(1,2,3)
(1, 2, 3)
```

```
>>> def f(*a):          # packing
...    print a
...    return zip(*a) # unpacking
...
>>> f([1],[2])
([1], [2])
[(1, 2)]
>>> f([1],[2],[3],[4])
([1], [2], [3], [4])
[(1, 2, 3, 4)
```

```
>>> theArray = [['a','b','c'],['d','e','f'],
['g','h','i']]
>>> zip(*theArray)   # unpacking
[('a', 'd', 'g'), ('b', 'e', 'h'), ('c', 'f', 'i')]
```

real generic memoizer

- combining lexical closure and *args

```
def memoized(f):
    cache = f.cache = {}
    def helper(*args):                  # packing
        if args not in cache:
            cache[args] = f(*args) # unpacking
        return cache[args]
    return helper

@memoized
def func(..., ..., ...):
...

>>> func(...)
>>> func.cache
{...}
```

decoration chain again

- stack multiple decorators together

```
@a @b @c
def f(...)
```

means

```
f=a(b(c(f)))
```

```
def memoized(f):
    # Same code as before...

def trace(f):
    def helper(x):
        print "Calling %s(%s) ..." % (f.__name__, x)
        result = f(x)
        print "... returning from %s(%s) = %s" % \
              (f.__name__, x, result)
        return result
    return helper

@memoized
@trace
def fib(n):
...
```

```
>>> fib(4)
Calling fib(4) ...
Calling fib(3) ...
Calling fib(2) ...
Calling fib(1) ...
... returning from fib(1) = 1
Calling fib(0) ...
... returning from fib(0) = 0
... returning from fib(2) = 1
... returning from fib(3) = 2
... returning from fib(4) = 3
3
```

decorator class

- decorators can be functions, or classes using __call__
- a class *instance* with __call__ method can act as a function

```
class memoize(dict):
    def __init__(self, func):
        self.func = func

    def __call__(self, *args):
        return self[args]

    def __missing__(self, key):
        result = self[key] = self.func(*key)
        return result

print memoize(fib)(30)  # WRONG (slow)

fib = memoize(fib)
print fib(30)           # CORRECT (fast)
```

```
>>> class A(object):
...     def __call__(self, x):
...       print x
...
>>> A()(5)
5
>>> A(5)
Error
```

getitem, setitem, and delitem

- __getitem__: a[x]
- __setitem__: a[x] = b
- __delitem__: del a[x]
- (implement these in HW2's BST)

__new__ and singleton pattern

- python object creation is in two steps
  - self = object.__new__(cls, ...) to create a new instance
  - self.__init__(...) to initialize member variables
- e.g. __new__ can check if an instance was created before

```
class Singleton(object):
    instance = None
    def __new__(cls):
        if not cls.instance:
            cls.instance = object.__new__(cls)
        return cls.instance
```

```
>>> a = Singleton(4)
>>> b = Singleton(5)
>>> a is b
True
```

__new__ and singleton pattern

- python object creation is in two steps
  - self = object.__new__(cls, ...) to create a new instance
  - self.__init__(...) to initialize member variables
- e.g. __new__ can check if an instance was created before
  - __new__ and __init__ must have the same signature!
  -

```
class Singleton(object):
    instance = None
    def __new__(cls, y):
        if not cls.instance:
            cls.instance = object.__new__(cls)
        return cls.instance

    def __init__(self, y):
        self.y = y
```

```
>>> a = Singleton(4)
>>> a.y
4
>>> b = Singleton(5)
>>> a is b
True
>>> a.y, b.y
5, 5
```

__new__ and singleton pattern

- python object creation is in two steps
  - self = object.__new__(cls, ...) to create a new instance
  - self.__init__(...) to initialize member variables
- e.g. __new__ can check if an instance was created before
  - __new__ and __init__ must have the same signature!
  - or do init in __new__ and leave __init__ blank or undefined

```
class Singleton(object):
    instance = None
    def __new__(cls, y):
        if not cls.instance:
            cls.instance = object.__new__(cls)
            cls.instance.y = y
        return cls.instance
```

```
>>> a = Singleton(4)
>>> a.y
4
>>> b = Singleton(5)
>>> a is b
True
>>> a.y, b.y
5, 5
```

__new__ and unique copy

- __new__ also useful for unique copy types (int, float, str)

```
class Unique(object):
    instances = {}
    def __new__(cls, y):
        if y not in cls.instances:
            new = object.__new__(cls)
            new.y = y
            cls.instances[y] = new
        return cls.instances[y]

class SubUnique(Unique):
    pass

c = Unique(5)
d = Unique(6)
e = Unique(5)
print c is e                # True
print c.y, d.y, e.y         # 5 6 5
print type(SubUnique(5))    # Unique
print type(SubUnique(7))    # SubUnique
```

unique copy:	int, float, str
immutable:	int, float, str, tuple
mutable:	list, dict, set, ...

```
>>> id(2)
4298191120
>>> a = 2
>>> id(a)
4298191120
>>> id(int.__new__(int, 2))
4298191120
>>> id("s")
4297516736
>>>  id("s" + "")
4297516736
```

__new__ for immutables

- __new__ also useful for unique copy types (int, float, str)
- the __init__ in immutables does nothing

unique copy:	int, float, str
immutable:	int, float, str, tuple
mutable:	list, dict, set, ...

```
>>> x = (1, 2)
>>> x
(1, 2)
>>> x.__init__([3, 4])
>>> x     # tuple.__init__ does nothing
(1, 2)
>>> a = "a"
>>> a.__init__(5)
>>> a
'a'
>>> y = [1, 2]
>>> y.__init__([3, 4])
>>> y     # list.__init__ reinitialises the object
[3, 4]
>>> tuple.__new__(tuple, [3, 4])
(3, 4)
```

iterator/generator

- what really is enumerate(a)? not a list!
- what is the difference b/w range(10) and xrange(10)?
- an iterator is a lazy list
- generator is the easiest way to implement an iterator

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

how to traverse a BST?

- lazy version of BST.list() (useful in HW2)

```
    def iteritems(self):
        if self.val is None:
            return
        for item in self.left.iteritems():
            yield item
        yield self.val
        for item in self.right.iteritems():
            yield item
```

generator expression

- lazy version of list comprehension
- memory efficient, but does not support list operations

```
# Generator expression
(x*2 for x in range(256))

# List comprehension
[x*2 for x in range(256)]
```

```
print enumerate(range(10))[:2]     # can’t index or slice
print [5,6] + enumerate(range(10)) # can't be added to lists
```

lazy versions

- range vs. xrange
- zip vs. itertools.izip
- dict.items() vs. dict.iteritems()
- dict.keys() vs. dict
- file.readlines() vs. file