

i117: Basic of Programming

4. User-defined data structures (1)

Kazuhiro Ogata, Canh Minh Do

Roadmap

- How to define data structures
- User-defined lists
- Stacks
- Queues
- Peano natural numbers

How to define data structures

- What are called “structure,” “record,” etc. are used to define data structures in some other programming languages, such as C and Pascal.
- “*class*” is used to define data structures in Python, which has an aspect of object-oriented programming like Java in which “class” is used to define data structures (or the structure of objects)

User-defined lists

Inductively defined as follows:

(1) nil is a list.

(2) If e is an element and l is a list, then $e \mid l$ is a list.

nil

the empty list

$0 \mid nil$

the list that only consists of 0

$1 \mid 0 \mid nil$

the list that consists of 1 and 0 in this order

$2 \mid 1 \mid 0 \mid nil$

the list that consists of 2, 1 and 0 in this order

User-defined lists

```
class List(object):  
    def cons(self,e):  
        pass  
    def append(self,l):  
        pass  
    def len(self):  
        pass  
    def __str__(self):  
        pass
```

Honestly, we do not need this class.

This class is used as the role of “*interface*” of Java.

A procedure defined inside a class is generally called a “*method*.”

In the Python terminology, the name of such a procedure is called an “*attribute*.”

Every method must take **self** (a receiver object of the class) as the first parameter.

`__str__` will be explained on the next page.

User-defined lists

```
class Nil(List):
    def cons(self,e):
        return NnList(e,self)
    def append(self,l):
        return l
    def len(self):
        return 0
    def __str__(self):
        return 'nil'
```

`Nil()` makes an object of class `Nil`, representing the empty list `nil`.

Let `nil` refer to the object.

`nil.len()` returns `0`.

Note that `len()` does not take any parameters, even if its definition has one (`self`).

`nil` (the receiver of the message) is used as the first parameter.

`nil.__str__()` returns `'nil'`.

`str(nil)` can also be used and returns `'nil'`.

User-defined lists

```

class NnList(List):
    head = None
    tail = Nil()
    def __init__(self,h,t):
        self.head = h
        self.tail = t
    def cons(self,e):
        return NnList(e,self)
    def append(self,l):
        return NnList(self.head,self.tail.append(l))
    def len(self):
        return 1 + self.tail.len()
    def __str__(self):
        return str(self.head) + ' | ' + str(self.tail)

```

There are two *instance variables* (or attributes) in class NnLIST.

Let *lst* be an object (a list) of class NnList. *lst.head* refers to the head (or top) element of the list, and *lst.tail* refers to the remaining part (tail or bottom) of the list.

None is used to represent nothing.

User-defined lists

```
....
def __init__(self,h,t):
    self.head = h
    self.tail = t
....
```

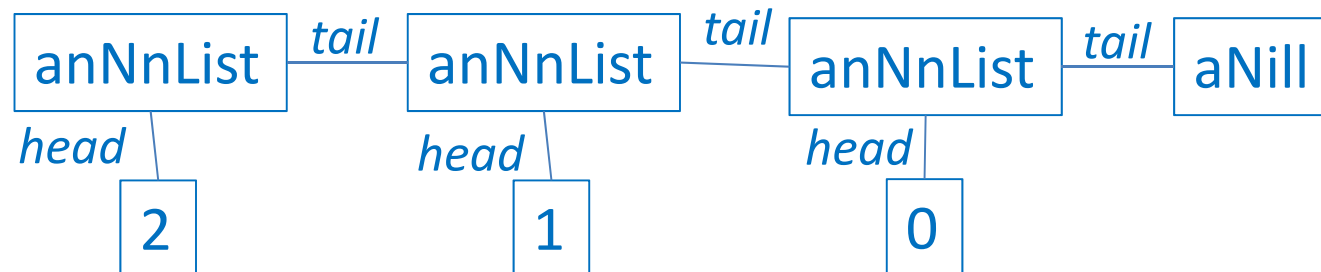
This is called a **constructor** with which an object (or an instance) of class **NnList** is made.

NnList(*elt*, *lst*)

It makes the list whose head is *elt* and whose tail is *lst*.



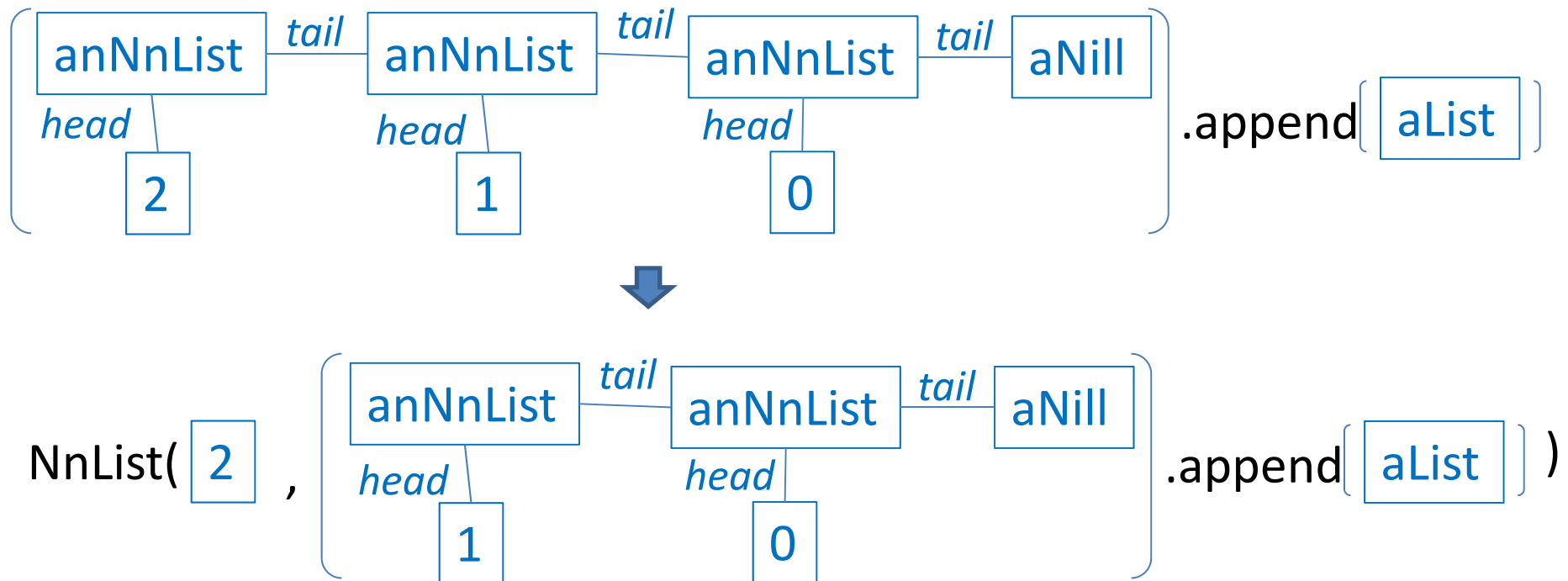
NnList(2, NnList(1, NnList(0, Nil()))) makes the following:



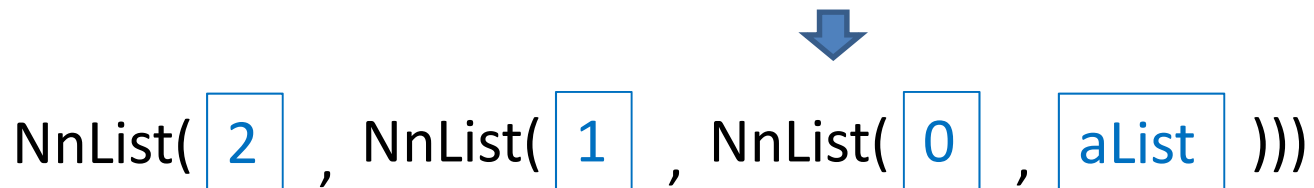
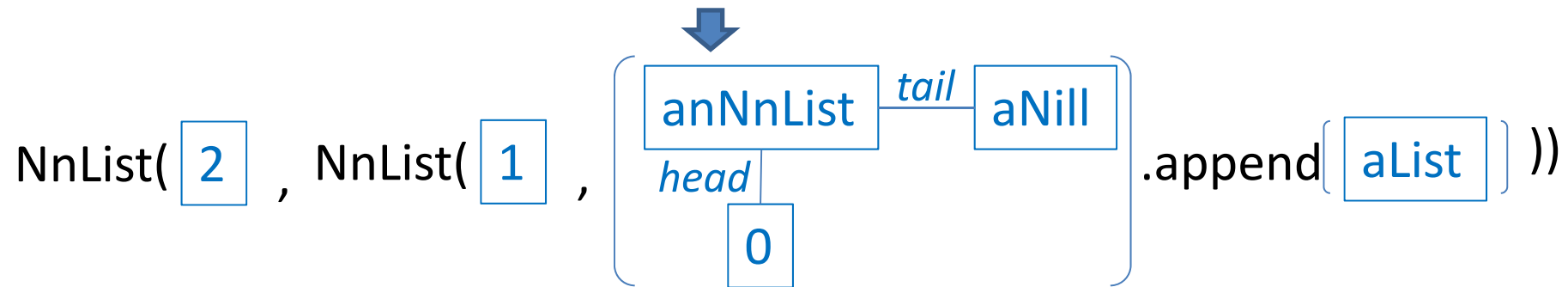
User-defined lists

```
....
def append(self,l):
    return NnList(self.head,self.tail.append(l))
....
```

It concatenates
two lists.

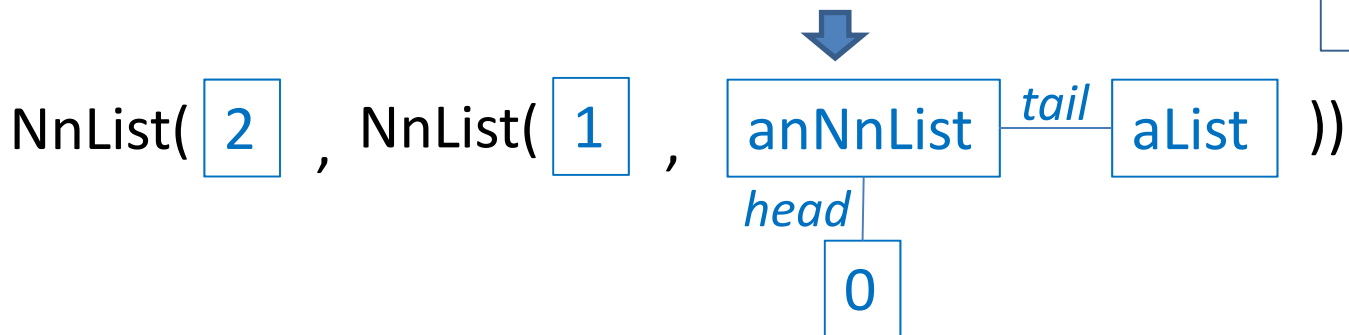


User-defined lists

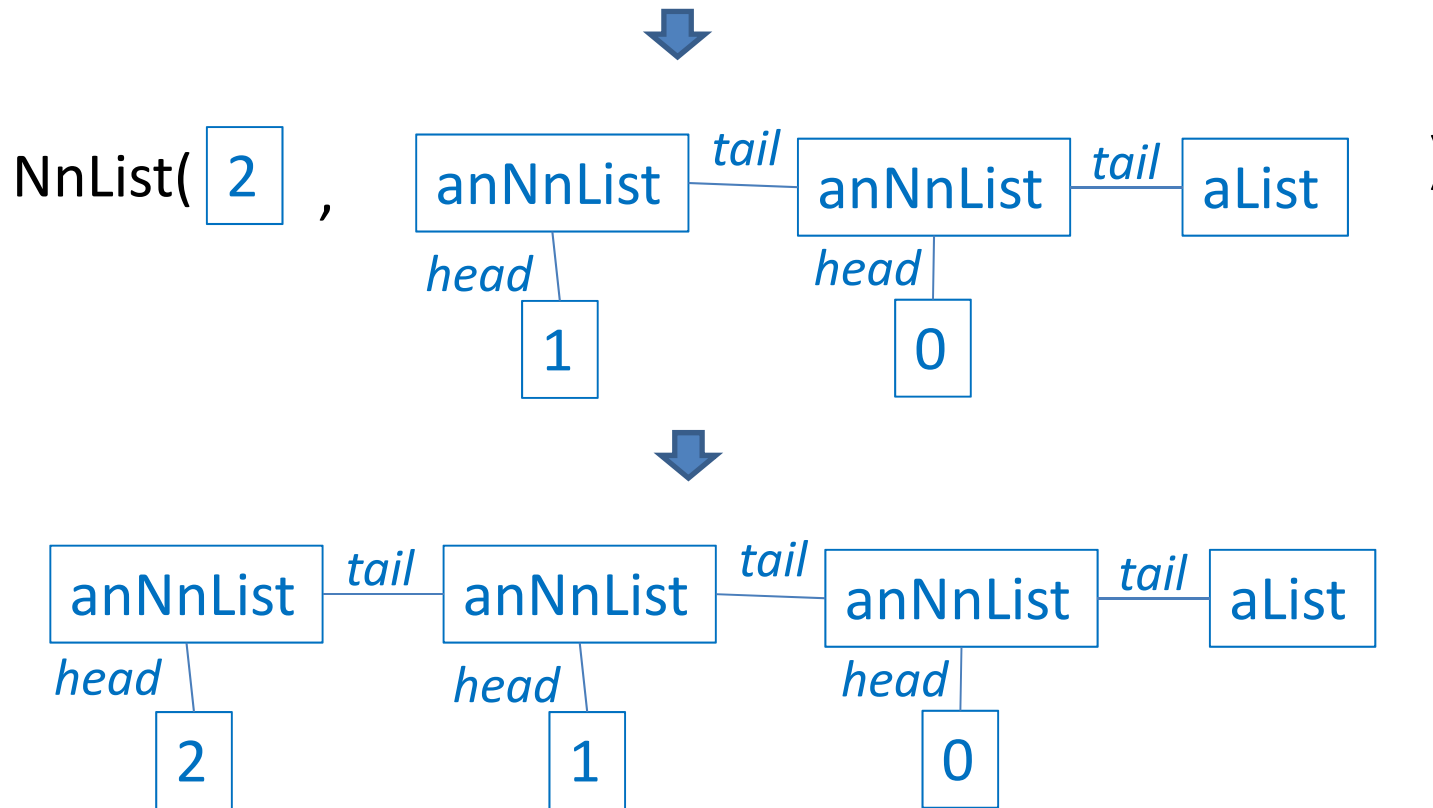


```

class Nil(List): ...
    def append(self, l):
        return l
    ...
  
```

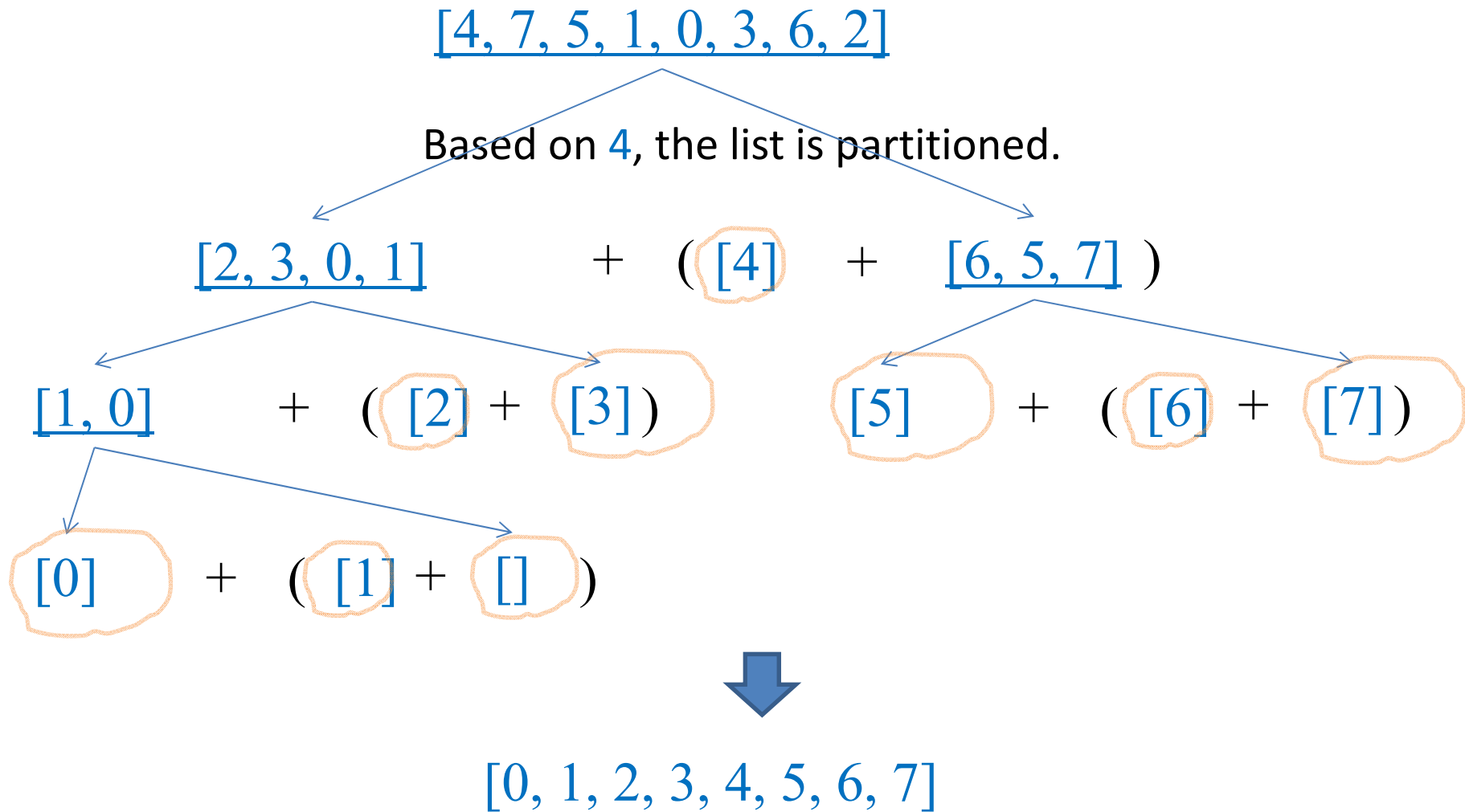


User-defined lists



User-defined lists

Sorting with Quicksort



User-defined lists

```
def qsort(lst):  
    if lst.len() <= 1:  
        return lst  
    else:  
        pair = partition(lst.head,lst.tail)  
        return qsort(pair[0]).append(qsort(pair[1]).cons(lst.head))
```

```
def partition(pvt,lst):  
    pair = (Nil(), Nil())  
    while isinstance(lst,NnList):  
        e = lst.head  
        lst = lst.tail  
        fl = pair[0]  
        sl = pair[1]
```

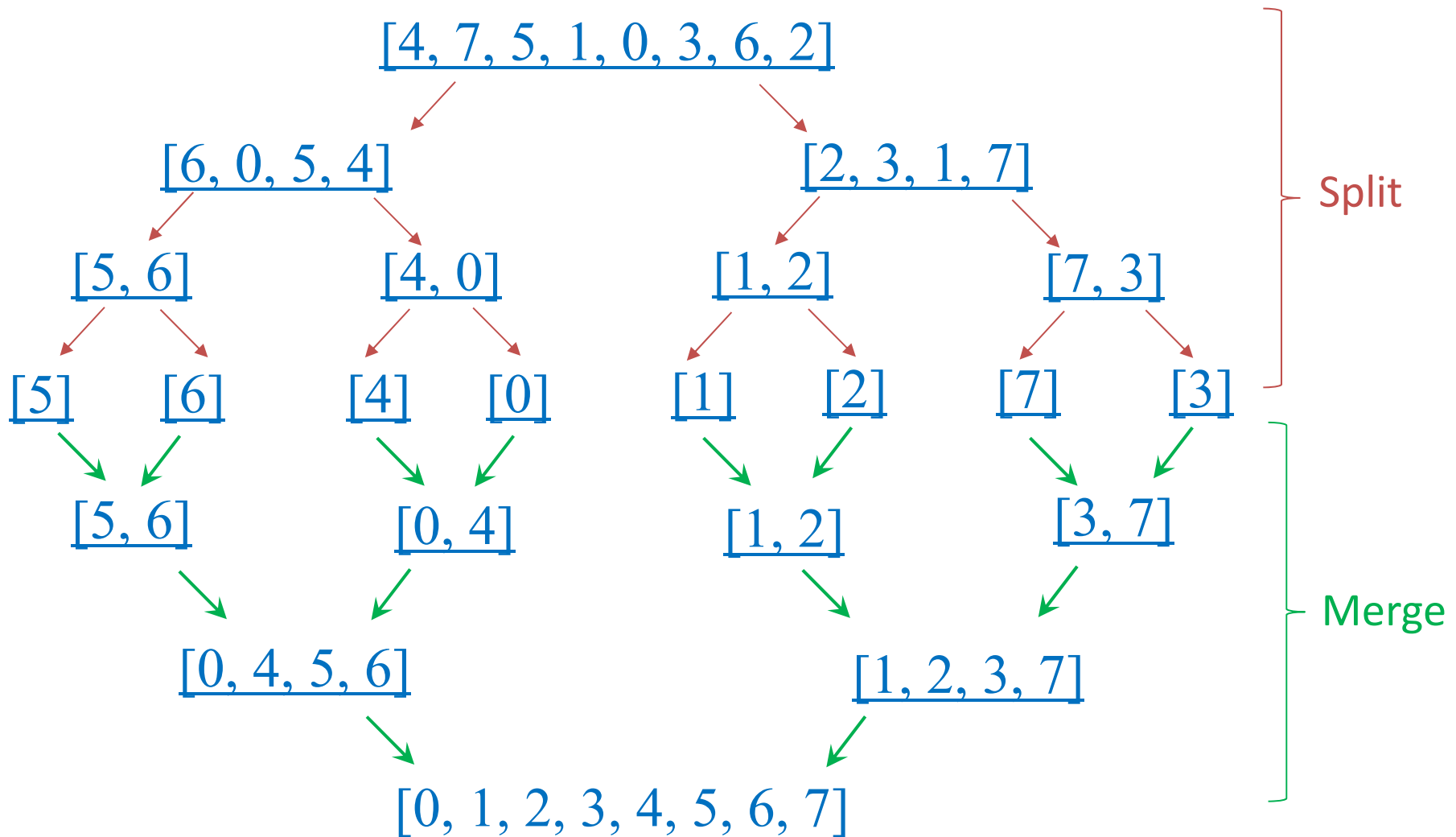
```
        if e < pvt:  
            pair = (fl.cons(e), sl)  
        else:  
            pair = (fl, sl.cons(e))  
    return pair
```

User-defined lists

```
l0 = Nil()  
l1 = NnList(2,l0)  
l2 = NnList(6,l1)  
l3 = NnList(3,l2)  
l4 = NnList(0,l3)  
l5 = NnList(1,l4)  
l6 = NnList(5,l5)  
l7 = NnList(7,l6)  
l8 = NnList(4,l7)  
print(str(l8))  
print(str(qsort(l8)))
```

User-defined lists

Sorting with Mergesort



User-defined lists

```
def msort(lst):  
    if lst.len() <= 1:  
        return lst  
    else:  
        l1 = Nil()  
        l2 = Nil()  
        flag = True  
        while isinstance(lst, NnList):  
            if flag:  
                l1 = l1.cons(lst.head)  
            else:  
                l2 = l2.cons(lst.head)  
            lst = lst.tail  
            flag = not flag  
        return merge(msort(l1),msort(l2))
```

```
def merge(l1,l2):  
    if isinstance(l1, Nil):  
        return l2  
    elif isinstance(l2, Nil):  
        return l1  
    else:  
        if l1.head < l2.head:  
            return NnList(l1.head,merge(l1.tail,l2))  
        else:  
            return NnList(l2.head,merge(l1,l2.tail))
```


User-defined lists

```
l0 = Nil()  
l1 = NnList(2,l0)  
l2 = NnList(6,l1)  
l3 = NnList(3,l2)  
l4 = NnList(0,l3)  
l5 = NnList(1,l4)  
l6 = NnList(5,l5)  
l7 = NnList(7,l6)  
l8 = NnList(4,l7)  
print(str(l8))  
print(str(msort(l8)))
```

Stacks

- A collection of data.
- Used in the *Last-In-First-Out (LIFO)* way.
- Three basic operations:
 - *push* puts an element in a stack at the top
 - *pop* deletes the top from a stack (if any)
 - *top* returns the top of a stack (if any)

Stacks

```
class Stack(object):  
    def push(self,e):  
        pass  
    def pop(self):  
        pass  
    def top(self):  
        pass  
    def __str__(self):  
        pass
```

```
class EmptyStack(Stack):  
    def push(self,e):  
        return NeStack(e,self)  
    def pop(self):  
        return self  
    def top(self):  
        return None  
    def __str__(self):  
        return 'es'
```

Stacks

Python does not allow us to use the same name for instance variables and methods.

```
class NeStack(Stack):  
    topElt = None  
    botom = EmptyStack()  
    def __init__(self,t,b):  
        self.topElt = t  
        self.botom = b  
    def push(self,e):  
        return NeStack(e,self)  
    def pop(self):  
        return self.botom  
    def top(self):  
        return self.topElt  
    def __str__(self):  
        return str(self.topElt) + ' ; ' + str(self.botom)
```

Stacks

```
s1 = EmptyStack()  
s2 = s1.push(0)  
s3 = s2.push(1)  
s4 = s3.push(2)  
print(str(s4))  
print(s4.top())  
s5 = s4.pop()  
print(str(s5))  
print(s5.top())
```

Queues

- A collection of data.
 - Used in the *First-In-First-Out (FIFO)* way.
 - Three basic operations:
 - *enqueue* puts an element in a queue at the end
 - *dequeue* deletes the top from a queue (if any)
 - *top* returns the top of a queue (if any)
 - Implemented with a built-in list in this course.
- (Note that there are multiple ways to implement each data structure, such as queues.)

Queues

```
class Queue(object):  
    elements = []  
    def __init__(self):  
        self.elements = []  
    def isEmpty(self):  
        if self.elements == []:  
            return True  
        else:  
            return False  
    def top(self):  
        if self.isEmpty():  
            return None  
        else:  
            return self.elements[0]
```

```
def dequeue(self):  
    if not self.isEmpty():  
        self.elements = self.elements[1:]  
    def enqueue(self,e):  
        self.elements = self.elements + [e]  
    def str(self):  
        return str(self.elements)
```

Queues

```
q = Queue()
print(q.str())
print(q.isEmpty())
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.str())
print(q.isEmpty())
print(q.top())
q.dequeue()
print(q.str())
q.dequeue()
print(q.str())
```


Peano natural numbers

Natural numbers have been formalized by Giuseppe Peano (1858 – 1932), an Italian mathematician.

Inductively defined as follows:

(1) 0 is a natural number.

(2) If n is a natural number, then the successor of n , denoted $s(n)$, is also a natural number.

0 $s(0)$ $s(s(0))$ $s(s(s(0)))$ $s(s(s(s(0))))$

0 1 2 3 4

Peano natural numbers

Addition is defined as follows:

$$0 + y = y$$

$$s(x) + y = s(x + y)$$

Multiplication is defined as follows:

$$0 * y = 0$$

$$s(x) * y = y + (x * y)$$

Peano natural numbers

```
class Nat(object):  
    def plus(self,y):  
        pass  
    def multiply(self,y):  
        pass  
    def str(self):  
        pass
```

```
class Zero(Nat):  
    def plus(self,y):  
        return y  
    def multiply(self,y):  
        return self  
    def str(self):  
        return '0'
```

Peano natural numbers

```
class NzNat(Nat):  
    prev = Nat()  
    def __init__(self,p):  
        self.prev = p  
    def plus(self,y):  
        return NzNat(self.prev.plus(y))  
    def multiply(self,y):  
        return y.plus(self.prev.multiply(y))  
    def str(self):  
        return 's(' + self.prev.str() + ')'
```

Peano natural numbers

```
zero = Zero()
one = NzNat(zero)
two = NzNat(one)
three = NzNat(two)
four = NzNat(three)
print('zero: ', zero.str())
print('one: ', one.str())
print('two: ', two.str())
print('three: ', three.str())
print('four: ', four.str())
print(three.str(), '+', four.str(), ' = ', three.plus(four).str())
print(three.str(), '*', two.str(), ' = ', three.multiply(two).str())
```