

# Lecture #11: Sequences

## Announcements

- HKN review session for Midterm 1 in 145 Dwinelle from 5-8 PM TONIGHT.
- Rooms for midterm to be assigned by login. Please watch website and Piazza.
- Please watch Piazza for news about TA review session on Monday.
- Alternative exams will be given in the labs on Wednesday.
- No labs next week. Also no Wednesday lecture.

# Sequences

- The term *sequence* refers generally to a data structure consisting of an *indexed collection of values*.
- That is, there is a first, second, third value (which CS types call #0, #1, #2, etc).
- A sequence may be *finite* (with a length) or *infinite*.
- As an object, it may be *mutable* (elements can change) or *immutable*.
- There are numerous alternative interfaces (i.e., sets of operations) for manipulating it.
- And, of course, numerous alternative implementations.
- Today: immutable, finite sequences, recursively defined.

# A Recursive Definition

- A possible definition: A sequence consists of
  - An empty sequence, or
  - A first element and a sequence consisting of the rest of the elements of the sequence other than the first (its *tail*).
- The definition is clearly recursive ("a sequence consists of ... and a sequence ..."), so let's call it an *rlist* for now.
- Suggests the following ADT interface:

```
# The empty rlist (unique).
empty_rlist = ...
def rlist(first, rest = empty_rlist):
    """A recursive list, r, such that first(R) is FIRST and
    rest(R) is REST, which must be an rlist."""
def first(r):
    """The first item in R."""
def rest(r):
    """The tail of R: the sequence consisting of items 1, 2,...,
    renumbered from 0."""
```

# Implementation With Pairs

- An obvious implementation uses two-element tuples (pairs). The result is called a *linked list*.

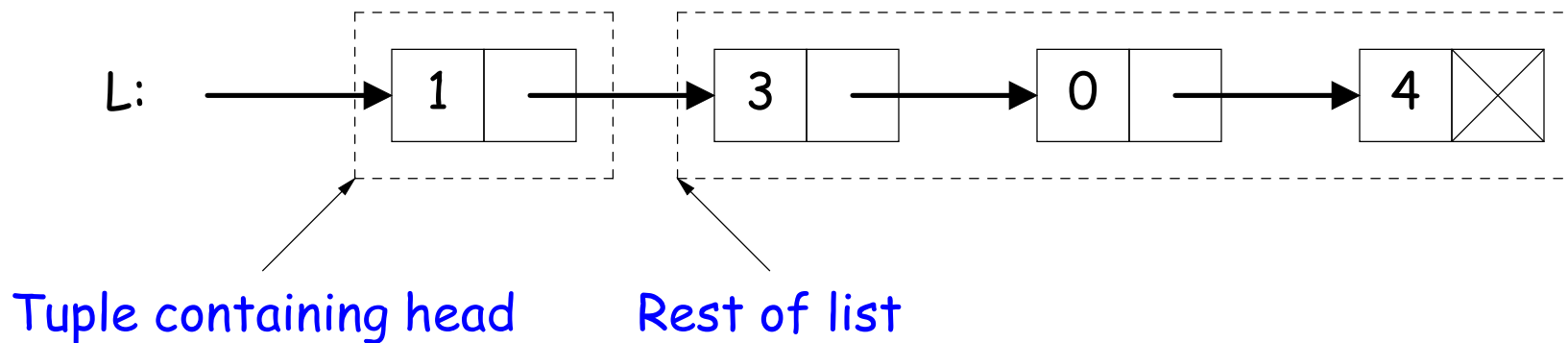
```
empty_rlist = None
def rlist(first, rest = empty_rlist):
    return first, rest
def first(r):
    return r[0]
def rest(r):
    return r[1]
```

# Box-and-Pointer Diagrams for Linked Lists

- Diagrammatically, one gets structures like this:

```
# The sequence 1, 3, 0, 4
```

```
L = rlist(1, rlist(3, rlist(0, rlist(4, empty_rlist))))
```

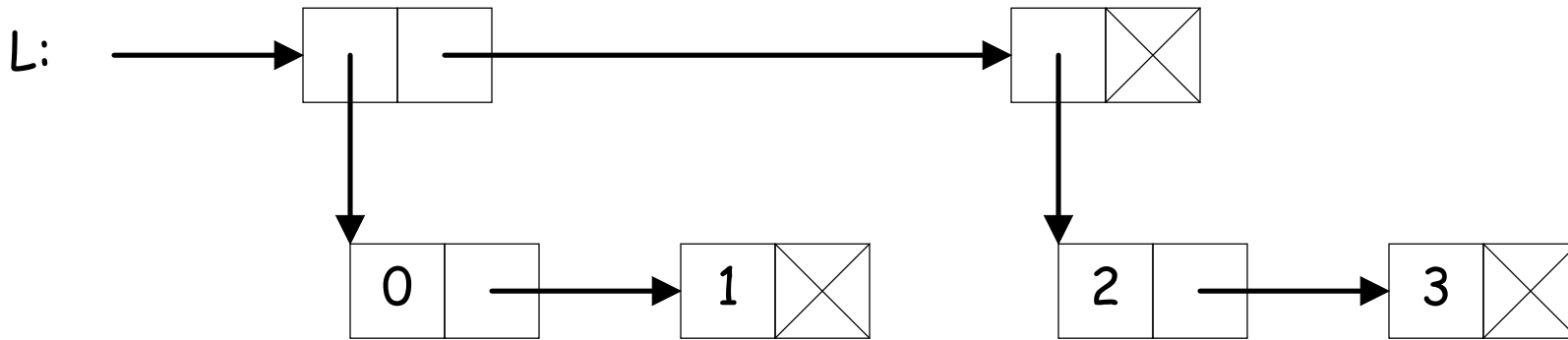


# Adding Dimensions

Our **rlists** can contain anything, including other **rlists**:

# The sequence containing sequences (0, 1) and (2, 3)

```
L = rlist(rlist(0, rlist(1, empty_list)),  
          rlist(rlist(2, rlist(3, empty_list)),  
                empty_rlist))
```



# Recursive Lists vs. Python Tuples

- In Python, tuples are not limited to pairs.
- Could have used `(1, 3, 0, 4)` or `((0, 1), (2, 3))`.
- But there are advantages to `rlists`:
  - For tuples, `rest(L)` corresponds to `L[1:]`.
  - The time and space required for this operation increases linearly with the length of `L`.
  - But `rest(L)` on an `rlist` takes constant time and no additional space.
- On the other hand,
  - Computing the length or the  $k$ th element of an `rlist` takes time proportional to the length of the sequence,
  - But for tuples, these are constant-time operations.

# From Recursive Structure to Recursive Algorithm

- The cases in the recursive definition of list often suggest a recursive approach to implementing functions on them.
- Example: length of an rlist:

```
def len_rlist(s):                                # A sequence is:
    """The length of rlist S."""
    if s == empty_rlist:                         # Empty or...
        return 0
    else:
        return 1 + len_rlist(rest(s))
                                                # A first element and
                                                # the rest of the list
```

- **Q:** Why do we know the comment is accurate?
- **A:** Recursive thinking: Because we assume the comment is accurate! (For “smaller” arguments, that is).
- Not tail recursive: can't directly make `len_rlist` iterative.



## Tail Recursion (Again)

- But a slight modification makes iteration possible:

```
def len_rlist(s):  
    def len(sofar, s):  
        """Return SOFAR + the length of rlist S."""  
        if s == empty_rlist:  
            return sofar  
        else:  
            return len(sofar + 1, rest(s))  
    len(0, s)
```

- We simply return the value of the recursive call to `len` directly, so this version is *tail recursive*, and can become a loop:

```
def len_rlist(s):  
    sofar = 0  
    while s != empty_rlist:  
        sofar, s = sofar+1, rest(s)  
    return sofar
```

## Another Example: Selection

- Want to extract item #k from an rlist (number from 0).
- Recursively:

```
def getitem_rlist(s, k):  
    """Return the element at index K of recursive list S.  
    Assumes K >= 0.  
    >>> getitem_rlist(rlist(2, rlist(3, rlist (4))), 1)  
    3"""  
  
    if _____:  
        return _____  
    else:  
        return _____
```

## Another Example: Selection

- Want to extract item #k from an rlist (number from 0).
- Recursively:

```
def getitem_rlist(s, k):  
    """Return the element at index K of recursive list S.  
    Assumes K >= 0.  
    >>> getitem_rlist(rlist(2, rlist(3, rlist (4))), 1)  
    3"""  
  
    if k == 0:  
        return _____  
    else:  
        return _____
```

## Another Example: Selection

- Want to extract item #k from an rlist (number from 0).
- Recursively:

```
def getitem_rlist(s, k):  
    """Return the element at index K of recursive list S.  
    Assumes K >= 0.  
    >>> getitem_rlist(rlist(2, rlist(3, rlist (4))), 1)  
    3"""  
  
    if k == 0:  
        return first(s)  
    else:  
        return _____
```

## Another Example: Selection

- Want to extract item #k from an rlist (number from 0).
- Recursively:

```
def getitem_rlist(s, k):  
    """Return the element at index K of recursive list S.  
    Assumes K >= 0.  
    >>> getitem_rlist(rlist(2, rlist(3, rlist (4))), 1)  
    3"""  
  
    if k == 0:  
        return first(s)  
    else:  
        return getitem_rlist(rest(s), k-1)
```

# Iterative getitem\_rlist

- From the previous version:

```
def getitem_rlist(s, k):  
    if k == 0:  
        return first(s)  
    else:  
        return getitem_rlist(rest(s), k-1)
```

- Can transform into an iterative version:

```
def getitem_rlist(s, k):  
    """Return the element at index K of recursive list S.  
    Assumes K >= 0."""  
  
    while _____:  
        s, k = _____  
    return _____
```

# Iterative getitem\_rlist

- From the previous version:

```
def getitem_rlist(s, k):  
    if k == 0:  
        return first(s)  
    else:  
        return getitem_rlist(rest(s), k-1)
```

- Can transform into an iterative version:

```
def getitem_rlist(s, k):  
    """Return the element at index K of recursive list S.  
    Assumes K >= 0."""  
  
    while k != 0:  
        s, k = _____  
    return _____
```

# Iterative getitem\_rlist

- From the previous version:

```
def getitem_rlist(s, k):  
    if k == 0:  
        return first(s)  
    else:  
        return getitem_rlist(rest(s), k-1)
```

- Can transform into an iterative version:

```
def getitem_rlist(s, k):  
    """Return the element at index K of recursive list S.  
    Assumes K >= 0."""  
  
    while k != 0:  
        s, k = rest(s), k-1  
    return _____
```



# Iterative getitem\_rlist

- From the previous version:

```
def getitem_rlist(s, k):  
    if k == 0:  
        return first(s)  
    else:  
        return getitem_rlist(rest(s), k-1)
```

- Can transform into an iterative version:

```
def getitem_rlist(s, k):  
    """Return the element at index K of recursive list S.  
    Assumes K >= 0."""  
  
    while k != 0:  
        s, k = rest(s), k-1  
    return first(s)
```

## Applying to All Elements

- Given an rlist, I'd like to create the list of the squares of its elements:

```
def square_rlist(s):  
    """The list of squares of the elements of rlist S."""  
    if _____:  
        return _____:  
    else:  
        return _____
```

## Applying to All Elements

- Given an rlist, I'd like to create the list of the squares of its elements:

```
def square_rlist(s):  
    """The list of squares of the elements of rlist S."""  
    if s == empty_rlist:  
        return _____:  
    else:  
        return _____
```

## Applying to All Elements

- Given an rlist, I'd like to create the list of the squares of its elements:

```
def square_rlist(s):  
    """The list of squares of the elements of rlist S."""  
    if s == empty_rlist:  
        return empty_rlist:  
    else:  
        return _____
```

## Applying to All Elements

- Given an rlist, I'd like to create the list of the squares of its elements:

```
def square_rlist(s):  
    """The list of squares of the elements of rlist S."""  
    if s == empty_rlist:  
        return empty_rlist:  
    else:  
        return rlist(first(s)**2, square_rlist(rest(s)))
```

# On to Higher Orders!

```
def map_rlist(f, s):  
    """The list of values F(x) for each element x of S in order."""  
    if s == empty_rlist:  
        return empty_rlist  
    else:  
        return rlist(f(first(s)), map_rlist(f, rest(s)))
```

- So `square_rlist(L)` is `map_rlist(lambda x:x**2, L)`.
- [Python 3 produces a different kind of result from its `map` function; we'll get to it.]
- Iterative version difficult here!

# Extending rlists

- Joining two lists together is called “appending” in most languages. Python uses “append” to mean “add an item,” and uses the term “extend” for joining lists.

```
def extend_rlist(left, right):  
    """The sequence of items of rlist 'left'  
    followed by the items of 'right'."""  
    if _____:  
        return _____  
    else:  
        return _____
```

---

- Again, iterative version is difficult.

# Extending rlists

- Joining two lists together is called “appending” in most languages. Python uses “append” to mean “add an item,” and uses the term “extend” for joining lists.

```
def extend_rlist(left, right):  
    """The sequence of items of rlist 'left'  
    followed by the items of 'right'."""  
    if left == empty_rlist:  
        return _____  
    else:  
        return _____
```

---

- Again, iterative version is difficult.



# Extending rlists

- Joining two lists together is called “appending” in most languages. Python uses “append” to mean “add an item,” and uses the term “extend” for joining lists.

```
def extend_rlist(left, right):  
    """The sequence of items of rlist 'left'  
    followed by the items of 'right'."""  
    if left == empty_rlist:  
        return right  
    else:  
        return
```

---

- Again, iterative version is difficult.

# Extending rlists

- Joining two lists together is called “appending” in most languages. Python uses “append” to mean “add an item,” and uses the term “extend” for joining lists.

```
def extend_rlist(left, right):  
    """The sequence of items of rlist 'left'  
    followed by the items of 'right'."""  
    if left == empty_rlist:  
        return right  
    else:  
        return rlist(first(left), extend_rlist(rest(left), right))
```

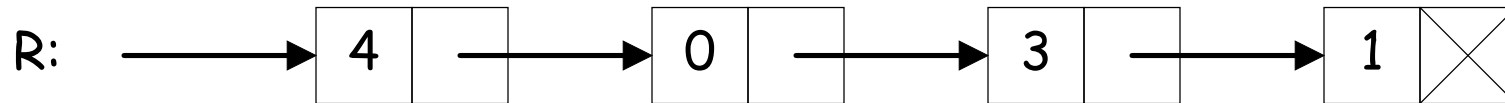
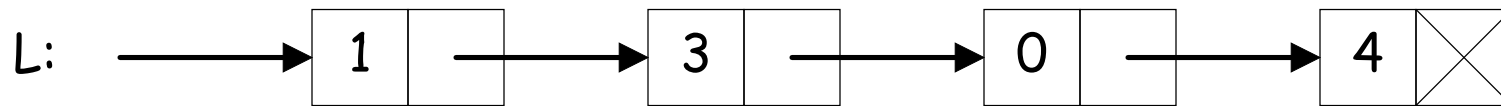
- Again, iterative version is difficult.

# Reversing

- Given a sequence represented by an `rlist` `L`, how can I create the reverse sequence, `reverse_rlist(L)`?

```
L = rlist(1, rlist(3, rlist(0, rlist(4, empty_rlist))))
```

```
R = reverse_rlist(L)
```



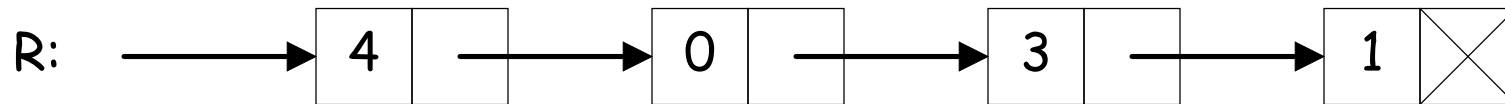
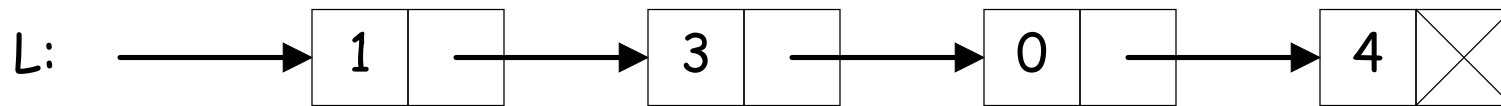
- What is the reverse of `empty_rlist`? \_\_\_\_\_.
  - Given an `rlist` `L`, what is the relationship between `first(L)`, `rest(L)`, and `R=reverse_rlist(L)`?
- \_\_\_\_\_
- \_\_\_\_\_

# Reversing

- Given a sequence represented by an `rlist` `L`, how can I create the reverse sequence, `reverse_rlist(L)`?

```
L = rlist(1, rlist(3, rlist(0, rlist(4, empty_rlist))))
```

```
R = reverse_rlist(L)
```



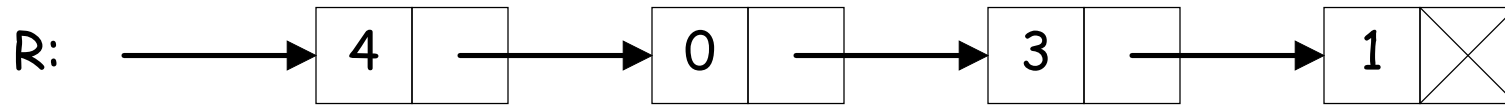
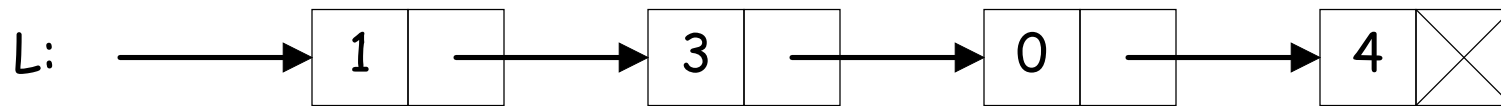
- What is the reverse of `empty_rlist`? `empty_rlist`.
  - Given an `rlist` `L`, what is the relationship between `first(L)`, `rest(L)`, and `R=reverse_rlist(L)`?
- 
-

# Reversing

- Given a sequence represented by an `rlist` `L`, how can I create the reverse sequence, `reverse_rlist(L)`?

```
L = rlist(1, rlist(3, rlist(0, rlist(4, empty_rlist))))
```

```
R = reverse_rlist(L)
```



- What is the reverse of `empty_rlist`? `empty_rlist`.
- Given an `rlist` `L`, what is the relationship between `first(L)`, `rest(L)`, and `R=reverse_rlist(L)`?

```
R = extend_rlist(reverse_rlist(rest(L)),  
                  rlist(first(L), empty_rlist))
```

# Iterative Reversing

- The iterative version of `rlist_reverse` is actually not bad.
- Rlists are most conveniently build from the end (because a tuple, once created, can't be changed).
- The *last* item of a reversed list is the *first* item of the original list.
- This leads to the following tail recursion:

```
def reverse_rlist(L):  
    def reverse_extend(to_do, already_done):  
        """The result of extending ALREADY_DONE with  
        the reverse of TO_DO."""  
        if to_do == empty_rlist:  
            return empty_rlist  
        else:  
            return reverse_extend(rest(to_do),  
                                  rlist(first(to_do), already_done))  
    reverse_extend(L, empty_rlist)
```

- Iterative version?