Computer Science I Linked Structures

CSCI-141 Lecture

10/25/2020

1 Problem Statement

A *sequence* is a collection of elements arranged in a specific order. We have already seen a data structure for sequences: Python lists. Our problem is duplicating this essential functionality without relying on Python lists. We will do so, but we will not be creating identical functionality. Our alternative approaches will have interesting differences concerning both how we write our programs and how we can use this new data structure. We will develop simple algorithms on our new sequence data structure to illustrate some of these ideas.

We start by applying the design of linkable nodes that contain values, and that also can be chained together, to the general concept of a `sequence`, or list. These nodes were the building blocks of the stack and queue data structures you have already seen.

2 The Linked Node for Lists

2.1 A Logical Perspective

We begin with the simplest possible description of a list using recursion. We have already seen examples of recursive functions. It is useful to characterize data recursively as well. We start with the intuitive notion of sequence: either a sequence is empty or it is not. A non-empty sequence must begin with a data element. If we ignore that element, we observe that the remainder is also a sequence. Thus we have the following recursive formal definition of a sequence, which we call a *linked sequence*.

Definition 1 *A* linked sequence *is one of the following.*

- *An empty linked sequence.*
- *A non-empty linked sequence, which has the following parts.*
  - *A* ***value****, which is a data element, and,*
  - *The* ***rest****, which is a linked sequence.*

2.1.1 Examples

A sequence could be empty: $\langle\rangle$. Or a sequence could be non-empty: $\langle 99, 100, 101\rangle$. The non-empty sequence begins with 99. If we ignore 99, we observe that the remainder is also a sequence: $\langle \cancel{99,}100, 101\rangle = \langle 100, 101\rangle$.

If we now start with the definition, empty corresponds to $\langle\rangle$. A non-empty linked sequence $\mathsf{nonEmpty}(101, \mathsf{empty})$ corresponds to the *singleton sequence* $\langle 101\rangle$. We can construct longer sequences by placing a linked sequence in the rest: $\mathsf{nonEmpty}(100, \mathsf{nonEmpty}(101, \mathsf{empty}))$

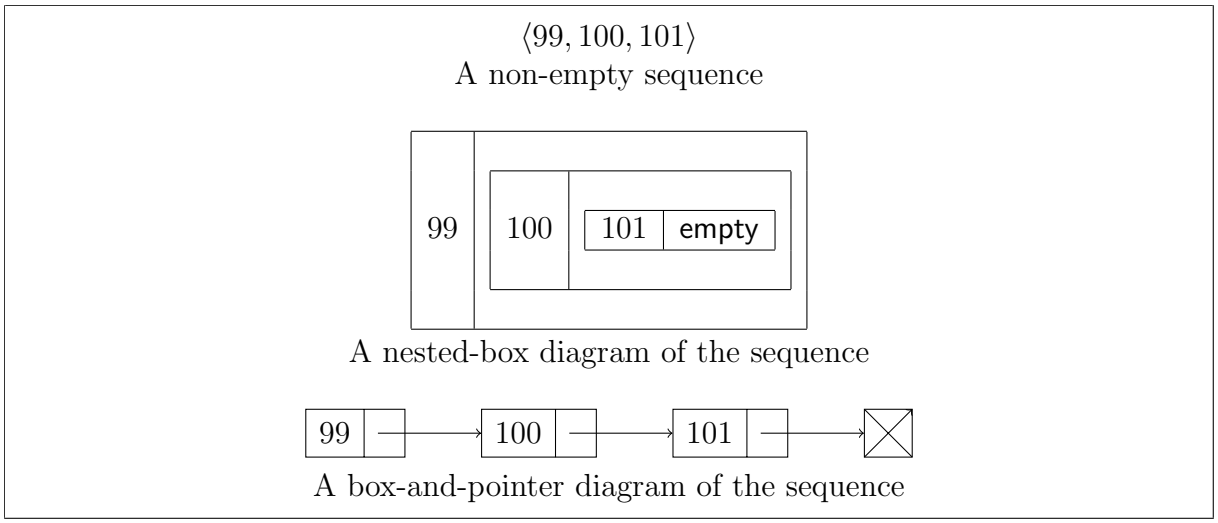

Figure 1: Sequence diagrams

which corresponds to $\langle 100, 101 \rangle$. Similarly, with 99 for the value and the previous sequence for the rest we have $\mathsf{nonEmpty}(99, \mathsf{nonEmpty}(100, \mathsf{nonEmpty}(101, \mathsf{empty})))$ which corresponds to $\langle 99, 100, 101 \rangle$.

Sometimes all this nesting becomes annoying and authors use box-and-pointer diagrams to characterize these linked sequences. See Figure 1.

Interestingly, the internal memory layout of what we have described more closely matches the box-and-pointer diagram. But the syntax used in Python to build such a sequence looks more like the nested-box diagram. Let's set that up now.

2.2 Python Implementation of Linked Nodes: A Review

To implement the linked sequence data structure in Python we will need concrete Python representations of both empty and non-empty linked sequences. We will use the Python value `None` to represent the empty linked sequence.

To implement a non-empty linked sequence, we will make use of the Python `class` construct with its `dataclass` decorator. For the time being we will enforce the rule that these nodes, once set up, cannot be modified. They are *frozen*, or *immutable*.

```
@dataclass( frozen=True )
class FrozenNode:
    """
    An immutable link node containing a value and a link to the next
                                        node
    """
    value: Any
    next: Union[ "FrozenNode", None ]
```

2.2.1 Examples

We now reproduce the examples from section 2.1.1. Recall that when the Python interpreter is given an expression that evaluates to `None`, nothing is displayed in the output.

```
>>> from node_types import FrozenNode
>>> print( None ) # empty list
None
>>> FrozenNode( 101, None )
FrozenNode(value=101, next=None)
>>> FrozenNode( 99, FrozenNode( 101, None ) )
FrozenNode(value=99, next=FrozenNode(value=101, next=None))
>>> FrozenNode(99,FrozenNode(100,FrozenNode(101,None)))
FrozenNode(value=99, next=FrozenNode(value=100, next=FrozenNode(value=101, next=None)))
>>> lnk = FrozenNode(99,FrozenNode(100,FrozenNode(101,None)))
>>> lnk.value
99
>>> lnk.next
FrozenNode(value=100, next=FrozenNode(value=101, next=None))
>>> lnk.next.value
100
>>> lnk.next.next
FrozenNode(value=101, next=None)
>>> print( lnk.next.next.next )
None
```

Figure 2 illustrates how these linked structures look in memory.

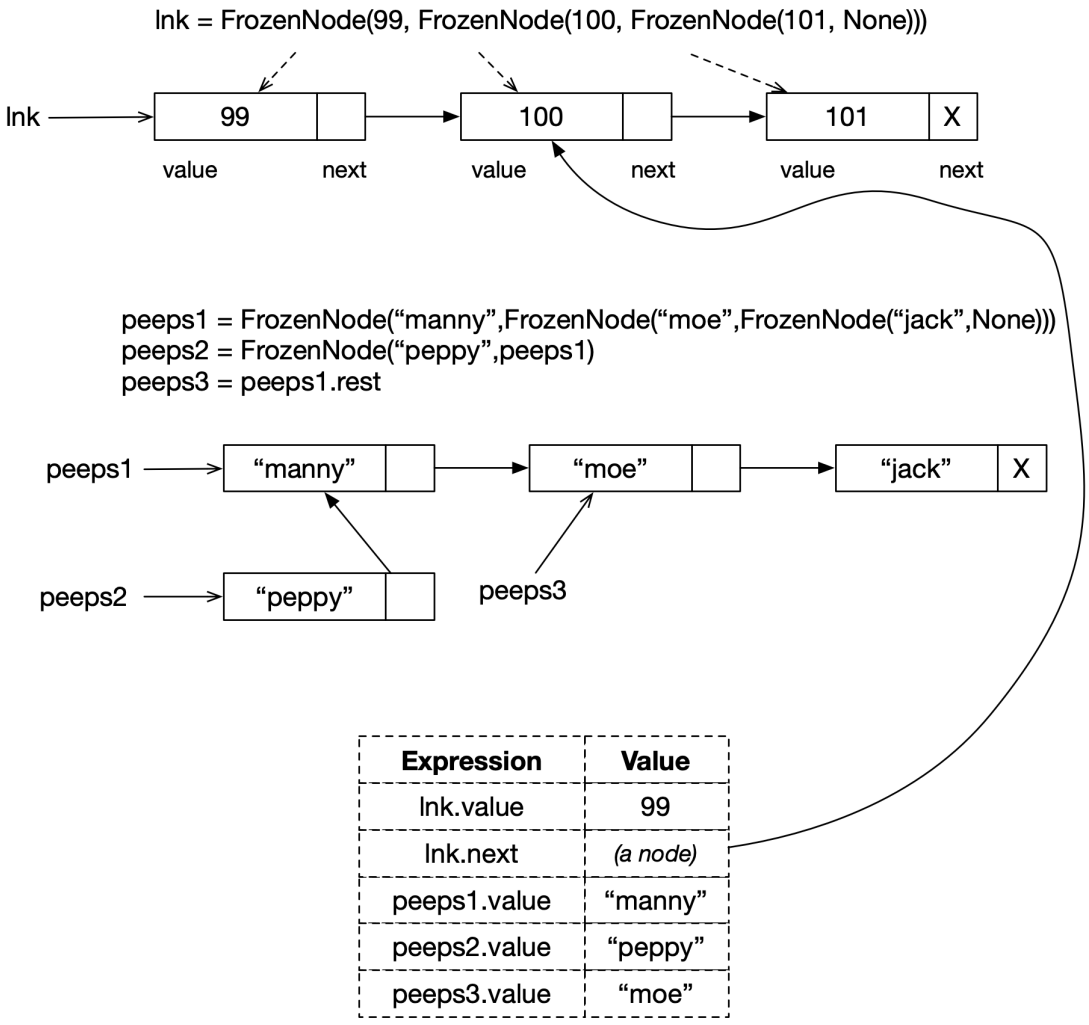

Figure 2: Sequence Examples, Using Boxes and Pointers

2.3 The Structural Recursive Template

Writers are known to suffer the Blank Screen Syndrome, in which they are stuck without a clue on how to begin. Programmers may experience the same feeling, but often there is a pattern, or template, that can help for recursive problems.

When a data structure is recursively defined, there is a function template for that data structure. One can immediately start by writing the template code, and then filling it in by asking a couple of questions. No more blank screens! The template for linked sequences follows. The word "`head`" is used to indicate a linked sequence. A linked sequence can be found by identifying its first node; the rest are all chained after it.

```
def f_template(head, ...):
    if head is None:
        result = ...
    else:
```

```
        result = ... head.value ... f_template(head.next, ...) ...
    return result
```

To fill in the template for a particular function, we ask, "What is the result when the linked sequence is empty?" The result replaces the dots after the first `return` statement. We then assume that the recursive call works (it is helpful to consider a concrete sequence). Given that we have the result for the rest of the sequence we ask, "How can the result for the rest of the sequence (`head.next`) help us construct the result for the original sequence?" The answer helps us fill in the dots in the recursion statement block.

The idea of the *structural recursive template* can be generalized to other recursive data structures. This generalization entails that base cases of the data structure correspond to base cases in the function and that recursive data correspond to recursive functions on sub-structure in the function. We will see other examples of recursive data structures and their corresponding templates in future lectures.

2.4 Applying the Structural Recursive Template

We will now use the structural recursive template to write functions that perform the following operations.

- convert the list to a string
- append another value to the list
- insert into the list based on an index position
- remove a value from the list

Note that, unlike the built-in list type, our structure is currently immutable. That means that any modifications to a linked sequence must create a new sequence that looks like the old one with the modifications. (The second half of this lecture covers modifying *mutable* sequences in place.)

2.4.1 Convert to String

We would like to write a function that computes a "human-readable" form of a linked sequence. For example, the call

`to_str(FrozenNode(99, FrozenNode(100, FrozenNode(101, None))))`

should return `"[99,100,101]"`. Let's start with the structural recursive template. For reasons that will become clear later, our initial task will be to construct the comma-separated list of values. Brackets will then be added.

```
def to_str( head ):
    """ to_str: Linked(T) -> str """
    if head is None:
        result = ...
    else:
        result = ... to_str(head.next) ...
    return result
```

Now we ask, "What is the result when the linked sequence is empty?" There are no characters in the string. Let's fill that in.

```
def to_str( head ):
    """ to_str: Linked(T) -> str """
    if head is None:
        result = ""
    else:
        result = ... to_str(head.next) ...
    return result
```

Answering the question, "How can the result for the rest help us construct the result for the original sequence?" is more challenging. Let's consider a concrete example. Suppose we have

`head` = `FrozenNode(99, FrozenNode(100, FrozenNode(101, None)))`

It follows that `head.next` = `FrozenNode(100, FrozenNode(101, None))`. *We assume* that `to_str` works on the rest; the result is `"100,101"`. We know that the result for `head` should be `"99,100,101"`. Therefore we ask a more concrete version of our question: "Given that the result for the rest is `"100,101"`, how can that be used to get `"99,100,101"` the result for the original sequence?" A reasonable answer is simply to prepend `"99,"` to the front of the previous result. Filling that in gives us the code.

```
def to_str( head ):
    """ to_str: Linked(T) -> str """
    if head is None:
        result = ""
    else:
        result = str( head.value ) + ',' + to_str( head.next )
```

This ends our first example of the application of our function template for our recursive data structure. However, the careful reader will have noticed that something is amiss. If

we actually call the function with `FrozenNode(99, FrozenNode(100, FrozenNode(101, None)))` as the argument, the string that is returned is `"99,100,101,"`.

Why? Because we have a special case.

If we are at the last node in the list, no comma is added after the value.

Therefore the recursive case needs a special test for is-last-node. The special text can be encoded by asking if the `next` slot in the current `head` node is `None` or not.

```
def to_str( head ):
    """ to_str: Linked(T) -> str """
    if head is None:
        result = ""
    else:
        result = str( head.value ) # Always add the new value.
        if head.next is not None:
            result += ','
        result += to_str( head.next )
    return result
```

Finally, we add the brackets. Were we to do this in the above function, the result would probably be `"[101,[101,[99]]]"` —not what we want[1]. Instead we will do a trick we have done before:

In a recursive algorithm, if something must be done initially that should not be done every step along the way, add a top-level function that does that work and also calls the recursive function to do the repetitive work.

Our final version of `to_str` is therefore two functions.

```
def to_str( head ) -> str:
    """ to_str: Linked(T) -> str """
    result = '['
    result += _to_str_rec( head )
    result += ']'
    return result

def _to_str_rec( head ) -> str:
    if head is None:
        result = ""
    else:
        result = str( head.value )
        if head.next is not None:
            result += ','
        result += _to_str_rec( head.next )
    return result
```

Let's call the time to execute the algorithm $time - for(N)$, This algorithm works on the head of the list, repeats itself for the rest of the list (`head.next`), and stops when the

1. although accurate, according to the structure!

list is empty. This means that $time - for(N)$ is proportional to $1 + time - for(N-1)$ Therefore the time complexity for `to_str` is $O(N)$.

Example Substitution Trace

To help with readability in the trace below we abbreviate `FrozenNode` as `FN`.

```
to_str(FN(99, FN(100, FN(101, None))))
   =  '[' + _to_str_rec(FN(99, FN(100, FN(101, None)))) + ']'
   =  '[' + '99' + ',' + _to_str_rec(FN(100, FN(101, None))) + ']'
   =  '[' + '99' + ',' + '100' + ',' + _to_str_rec(FN(101, None)) + ']'
   =  '[' + '99' + ',' + '100' + ',' + '101' + _to_str_rec(None) + ']'
   =  '[' + '99' + ',' + '100' + ',' + '101' + '' + ']'
   =  '[99,100,101]'
```

2.4.2 Recursive Extension

We would like to write a function that appends a new value to the end of a linked sequence. Note that we will name this function `append` to match the name of the equivalent method for Python `lists`.

Here is an example. We would like

```
append( FrozenNode(99, FrozenNode(100, FrozenNode(101, None))), 75 )
```

to produce

`FrozenNode(99, FrozenNode(100, FrozenNode(101, FrozenNode(75, None)))).`

We start with the structural recursive template, adjusting for an additional parameter.

```
def append( head, new_value ):
    """ append: Linked(T), T -> Linked(T) """
    if head is None:
        result = ...
    else:
        result = ... head.value ... append(head.next, ...) ...
    return result
```

Now we ask, "What is the result when the linked sequence is empty?" Simple! We make a single node out of the given value.

```
def append( head, new_value ):
    """ append: Linked(T), T -> Linked(T) """
    if head1 is None:
        result = FrozenNode( new_value, None )
    else:
        result = ... head1.value ... append(head1.next, ...) ...
    return result
```

How do we employ the head of the list in the answer, and how do we use a recursive call to `append` to help out? Observe that, to append a value onto a list, you append the value on the tail of the list. Then once we have done that, we "put the head back on". This gives us our solution.

Let's consider a concrete example. In the following text we will again abbreviate "`FrozenNode`" to "`FN`" to keep the line length under control and to concentrate on the arrangement of the sequences' values.

Suppose we have the argument

`head` = `FN(99, FN(100, FN(101, None)))`

and the argument

`new_value` = 75.

And so it follows that

`head.next` = `FN(100, FN(101, None))`.

But note that the sequence $[100, 101]$ plus 75 contain the last three elements of the top level solution. So we append 75 to the rest of, or *tail* of, the `head` sequence. Trusting in recursion, we assume that `append` works on the rest. The result is

`FN(100, FN(101, FN(75, None))).`

We know that the top-level result for the full `head` sequence should be

`FN(99, FN(100, FN(101, FN(75, None)))).`

which is simply connecting the first value in `head` to the result of the recursive `append` operation.

Filling that in gives us the full solution.

```
def append( head, new_value ):
    """ append: Linked(T), T -> Linked(T) """
    if head1 is None:
        result = FrozenNode( new_value, None )
    else:
        result = FrozenNode( head.value, append(head.next,new_value) )
    return result
```

NOTE Observe that there is an alternative way of expressing an algorithm like this in Python. We could simply put `return` statements where we currently assign our `result` variable.

```
def append( head, new_value ):
    """ append: Linked(T), T -> Linked(T) """
    if head1 is None:
        return FrozenNode( new_value, None )
    else:
        return FrozenNode( head1.value, append(head.next,new_value) )
```

This difference is largely a matter of personal preference, although for some it is a matter of principle.[2]

The time complexity for `append` is $O(N)$ because the list in the recursive call keeps getting shortened by 1 until it reaches 0 ($\Rightarrow$ linear time), and then things just build back up again in a linear fashion.

Example: Extension Trace

```
append(FN(99, FN(100, None)), 75)
                    =  FN(99, append(FN(100, None), 75))
                    =  FN(99, FN(100, append(None, 75)))
                    =  FN(99, FN(100, FN(75, None)))
```

Figures 3 and 4 illustrate how these linked structures look in memory.

2. If you are curious, do a web search with "Is it bad to have more than one return statement in a function?".

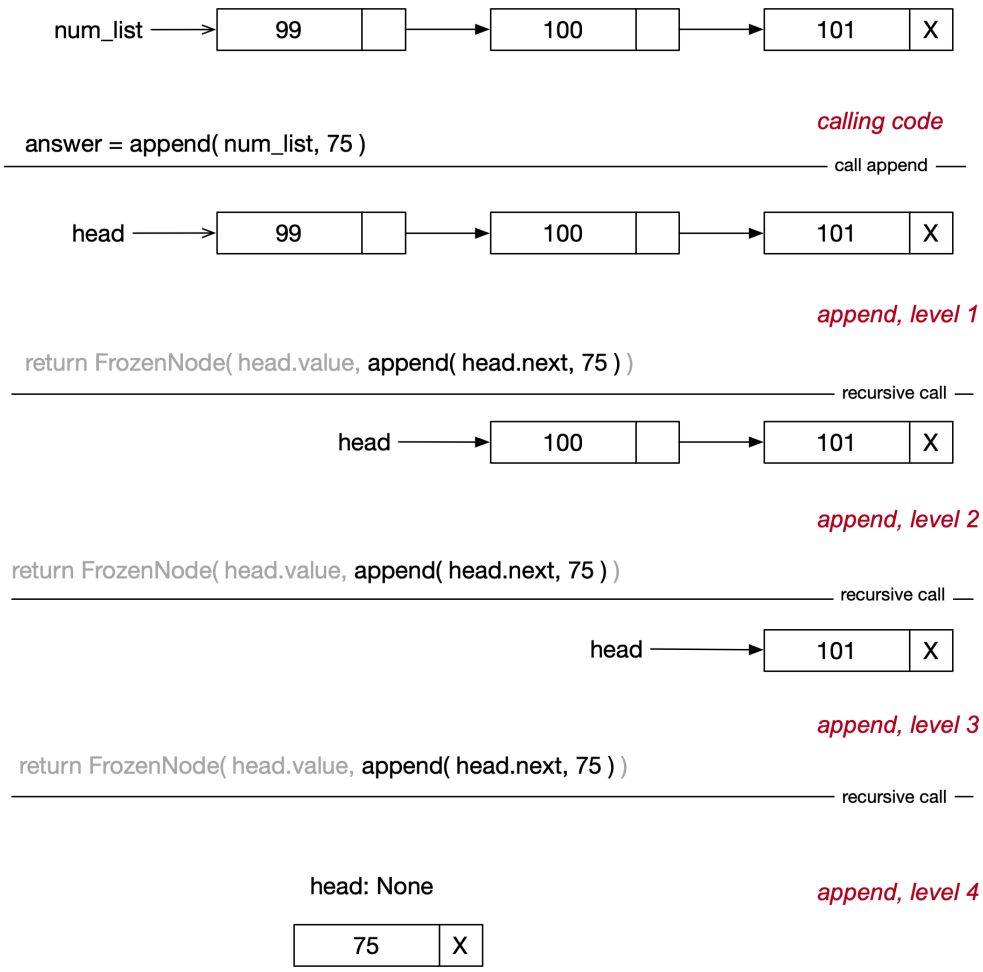

Figure 3: The append Function, Downward Recursive Calls

2.4.3 Insertion by Index

The third problem we will cover is how to insert a new value into the middle of a list, where the target position is specified as an index. Instead of using the long-form `FrozenNode` construction syntax, we will borrow from Python `list` syntax. The idea is as follows.

```
insert( [a,b,c,d], p, 3 ) = [a,b,c,p,d]
```

We start with the structural recursive template, again, adding the needed extra parameters

```
def insert_before_index( head, new_value, index ):
    """ insert_before_index: Linked(T), Any, Integer -> Linked(T) """
    if head1 is None:
        result = ...
    else:
        result = ... head1.value ... insert_before_index(head1.next, ...) ...
    return result
```

Now we ask, "What is the result when the linked sequence is empty?"

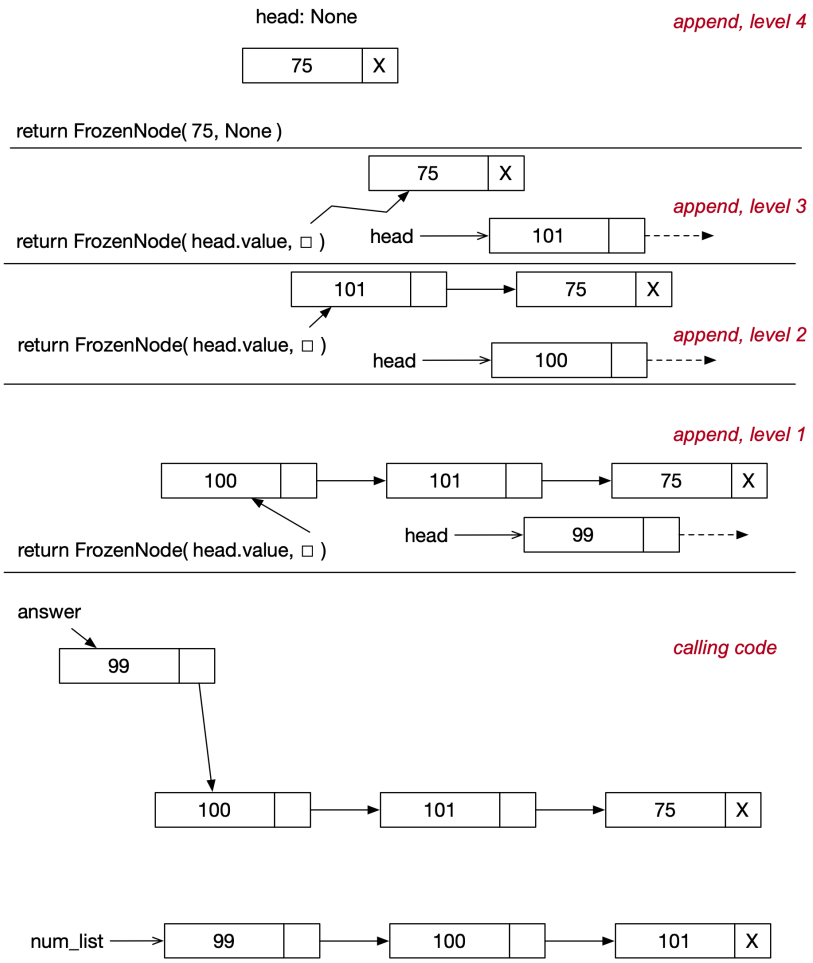

Figure 4: The append Function, Returned Values

Unfortunately our answer is, "It depends." If the index is 0, that's fine. In general it is perfectly reasonable to insert a new value before index N, when N is the length of the list, even 0; it means we should put the new value at the end of the list. But if the index is greater than 0, and there are no elements left, this is an error.[3] This thinking suggests the following code.

```
if head is None:
    if index == 0:
        # Result is a single node containing the new value.
    else:
        # Raise an error stating that the index is unreasonable.
```

We will write it this way for now, since that's what the template tells us. But later we will observe that things can be written more efficiently. Note the precondition added below to

3. We are using a special kind of statement called `raise`, which signals to the caller that there is a problem. In this course you don't have to worry about the details of what happens when exceptions are raised, or how to handle them.

the function's document string.

```
def insert_before_index( head, new_value, index ):
    """
    insert_before_index: Linked(T), Any, Integer -> Linked(T)
    :pre: 0 <= index <= length of list
    """
    if head is None:
        if index == 0:
            return FrozenNode( new_value, None )
        else:
            raise IndexError( "List is shorter than index " + str( index ) + "!" )
    else:
        result = ... head1.value ... insert_before_index(head1.next, ...) ...
    return result
```

For the recursive case, how do we leverage the information shown in the penultimate line of the template? We make this observation.

Inserting a value into a sequence at position n is the same as appending the value onto the tail of the sequence.

This is because it is possible to define the given `index` parameter in two different ways.

1. A current position in the sequence, in front of which the new value will be placed, or
2. The index of the new value once it's in the sequence.

We therefore build the following prototype complete function.

```
def insert_before_index( head, new_value, index ):
    """
    insert_before_index: Linked(T), Any, Integer -> Linked(T)
    :pre: 0 <= index <= length of list
    """
    if head is None:
        if index == 0:
            return FrozenNode( new_value, None )
        else:
            raise IndexError( "List is shorter than index " + str( index
                                                    ) + "!" )
    else:
        result = FrozenNode(
                    head.value,
                    insert_before_index( head.next, new_value, index-1
                                                                )
        )
    return result
```

But hold your horses! Checking if the index has been decremented to zero is also a base case in its own right, and clearly we are currently not checking for that case all the time

—only if the list is empty. Let's redo the base case by observing that the *fundamental* test is actually if the index has reached 0. It tells us that this is the place where the new value should go. Alternatively, if we know the index is not zero, we can ask, "Well, is there any more of the list to go through?" If the answer is "yes" we can go on to do the recursion.

Our algorithm is completely protected from indices being past the end of the list in the following revised and final form of the function.

```
def insert_before_index( head, new_value, index ):
    """
    insert_before_index: Linked(T), Any, Integer -> Linked(T)
    :pre: 0 <= index <= length of list
    """
    if index == 0:
        return FrozenNode( new_value, head ) # head None or not None
                                               works fine.
    elif head is None: # index > 0
        raise IndexError( "List is shorter than index " + str( index ) +
                                                "!" )
    else:
        return FrozenNode(
                    head.value,
                    insert_before_index( head.next, new_value, index - 1
                                                                  )
        )
```

The time complexity of insertion by index is $O(N)$.

Example: Insertion Trace

```
insert_before_index(FN(99, FN(100, FN(101, None))), 35, 2)
       =  FN(99, insert_before_index(FN(100, FN(101, None)), 35, 1)
       =  FN(99, FN(100, insert_before_index(FN(101, None), 35, 0)
       =  FN(99, FN(100, FN(35, FN(101, None))))
```

2.4.4 Removal by Value

Our last basic recursive operation on linked sequences is removal by value. That is, we want to create a new linked sequence that is the same as the original one, except that one value has been removed. What is being removed is identified by its own value rather than a positional index.

`remove( [a,b,c,d], c ) = [a,b,d]}`

Two points:

- If there is more than one such value, remove only the first instance.
- If the value is not present, it is an error.

Here is our starting template.

```
def remove_value( head, value ):
    """ remove_value: Linked(T), Any -> Linked(T) """
    if head1 is None:
        result = ...
    else:
        result = ... head1.value ... remove_value(head1.next, ...) ...
    return result
```

We have the standard base case that would tell us that if the list is empty we have an error situation. The additional base case is when the value in the current node matches the value being sought —just return the rest of the list.

```
def remove_value( head, value ):
    """ remove_value: Linked(T), Any -> Linked(T) """
    if head is None:
        raise ValueError( "No such value " + str( value ) + " in list!" )
    elif head.value == value:
        result = head.next
    else:
        result = ... head1.value ... remove_value(head1.next, ...) ...
    return result
```

For the recursive case, it's a simple matter of continuing the search, and rebuilding "on the way back".

```
def remove_value( head, value ):
    """ remove_value: Linked(T), Any -> Linked(T) """
    if head is None:
        raise ValueError( "No such value " + str( value ) + " in list!"
                                                )
    elif head.value == value:
        result = head.next
    else:
        result = FrozenNode( head.value, remove_value( head.next, value
                                                ) )
    return result
```

The code in the provided Python files again abbreviates the above solution by eliminating the `result` variable.

The time complexity of removal by value is $O(N)$.

Example: Removal Trace

```
remove_value(FN(99, FN(100, FN(101, None))), 100)
           =  FN(99, remove_value(FN(100, FN(101, None)), 100)
           =  FN(99, FN(101, None))
```

2.5 Accumulative Recursion: A Tail-Recursive Approach

Accumulative recursion is a technique that involves introducing an additional parameter, the *accumulator*, to be able to write the recursive code differently. Again, there is a template for this form of recursion. The template below has an additional accumulator parameter `acc` that passes partial results to subsequent recursive calls.

```
def f_accum_template(head, ..., acc):
    if head == None:
        return ...
    else:
        return f_accum_template(head.next, ..., ... head.value ... acc ...)
```

Note now that the *very last thing we do* in the non-base case is to execute a recursive call to the same function. Whatever that call returns is what the current instance of the function returns. In the other template, we'd have to do something with the returned value before the next return happened. You know a special name for this type of recursive function: it is a *tail recursive* function. You also know the advantage of the function being tail-recursive: it can easily be written using a loop instead of recursion.

Introducing this new parameter requires finding a suitable generalization of the original function; it is often difficult to find such a function. But once we've found the right generalization, filling in the template merely involves some calculation.

2.5.1 Accumulative String Conversion

This section shows a rewriting of the string conversion function that uses an accumulator.

We define the function `to_str_acc_rec` in a rather weird way:

$$\texttt{to\_str\_acc\_rec}(\textit{head}, \textit{acc}) = \textit{acc} + \texttt{to\_str}(\textit{head}).$$

That is, we think of the *head* as the rest of the linked sequence yet to be converted and the *acc* as the converted part of the sequence that came before *head*.

We start with the accumulative tail-recursive template.

```
def to_str_acc_rec( head, acc ):
    """ to_str_acc_rec: Linked(T), str -> str """
    if head == None:
        return ...
    else:
        return to_str_acc_rec( head.next, ... acc ... )
```

If there is nothing else, i.e., `head` is `None`, `acc` contains the full result (minus the brackets, for now).

Let's fill that in.

```
def to_str_acc_rec( head, acc ):
    """ to_str_acc_rec: Linked(T), str -> str """
    if head is None:
```

```
        return acc
    else:
        return to_str_acc_rec(head.next, ... acc ...)
```

For the non-empty sequence case, we must fulfill the assumption the base case is making: Travel down the linked sequence one more node and add text to the accumulator.

Filling that in gives us the code.

```
def to_str_acc_rec( head, acc ):
    """ to_str_acc_rec: Linked(T), str -> str """
    if head is None:
        return acc
    else:
        return to_str_acc_rec(head.next, acc + str(value))
```

For basic functionality that's fine. But we must tie up these loose ends.

1. Put the commas and blanks between the converted values.
2. Write a non-recursive wrapper function to initialize the accumulator and include the brackets.

```
def to_str_acc( head ):
    """ to_str_acc: Linked(T) -> str """
    return '[' + to_str_acc_rec( head, "" ) + ']'

def to_str_acc_rec( head, acc ):
    """ to_str_acc_rec: Linked(T), str -> str """
    if head is None:
        return acc
    else:
        suffix = str(value)
        if head.next is not None:
            suffix += ","
        return to_str_acc_rec(head.next, acc + suffix)
```

Example: to_str_acc Trace

```
to_str_acc(FN(3, FN(7, FN(11, None))))
      =  '[' + to_str_acc_rec( FN(3, FN(7, FN(11, None))), "" ) + ']'
      =  '[' + to_str_acc_rec( FN(7, FN(11, None)), "3," ) + ']'
      =  '[' + to_str_acc_rec( FN(11, None), "3,7," ) + ']'
      =  '[' + to_str_acc_rec( None, "3,7,11" ) + ']'
      =  '[' + "3,7,11" + ']'
      =  "[3,7,11]"
```

The time complexity for `to_str_acc_rec` is the same as the time complexity of `to_str`: $O(N)$.

3 Working with Mutable Linked Sequences

With an example of the accumulator pattern studied, we are ready to move to algorithms on linked sequences that use iteration instead of recursion. At the same time we will make another change that often goes along with loop algorithms: mutable sequences.

The first step in allowing linked sequences to be modified is to make the nodes themselves mutable.

```
@dataclass( frozen=False )
class MutableNode:
    """
    A mutable link node containing a value and a link to the next node
    """
    value: Any
    next: Union[ "MutableNode", None ]
```

We now observe that one of two things can happen if we modify a linked sequence.

- Something will change "inside" the sequence, but the head remains the same.
- The front of the sequence changes, yielding a new head.

We do not want our client code to be full of actions like the following.

```
my_list = . . .
result = do_action( my_list )
if result means that the head has changed:
    update my_list
```

Instead we will *encapsulate* our mutable linked list in another object, which we are naming `LinkedList`.

```
@dataclass( frozen=False )
class LinkedList:
    """
    A storage object for whatever the current head of the sequence is
    """
    head: Union[ MutableNode, None ] = None
    size: int = 0
```

The structure stores the size of the list too. This is an example of the tradeoff between computing something every time you need it and using extra memory to store the value so that it does not have to be recomputed. Here is our template for loop-based linked list operations.

```
def f_template( lst ):
    # If computing an answer, initialize accumulator here.
    node = lst.head
    while . . . node . . . :
        . . .
        . . . node . . .
        . . .
```

```
        # If using an accumulator, update its value.
        node = node.next
    # If modifying the list, update node.size.
    # If computing an answer, return the accumulator.
```

3.0.1 Conversion to a String

But let us begin with a non-modifying operation, string conversion. The only difference from the earlier solution is that we will accumulate the answer using a loop instead of recursion.[4]

The resulting string that slowly builds up is our accumulator.

```
def to_str( lst ):
    """
    to_str: Linked(T) -> None
    Construct a string that shows the contents of a linked list.
    """
    result = "["
    node = lst.head
    while node is not None:
        result += " " + str( node.value )
        if node.next is not None:
            result += ","
        node = node.next
    result += " ]"
    return result
```

4. Note that functions that don't modify their lists can be done recursively or with a loop, and it does not matter if the nodes are mutable or not. The same code works. This means for example you can apply either version of `to_str` to either kind of linked list.

3.0.2 Iterative Extension

In rewriting the `append` function for mutable sequences, the name becomes more appropriate. We are actually modifying (*mutating*) the first sequence. But something else remains the same: We must locate the end of the first (target) sequence. In the recursive case, we then built up a new sequence starting with the second argument sequence, building on to the front of it nodes containing values from the first sequence. With mutable lists, the job is the opposite: Add new nodes using the values from the second sequence on to the end of the first sequence.

Note: Remember that the parameters to these functions are not linked nodes; they are `LinkedList` objects that encapsulate the heads of their lists.

- `lst.head = node`
  The head of `lst`'s list is changing.
- `node1.next = node2`
  We are changing what node comes after `node1`.

```
def append( lst, new_value ):
    """
    append: Linked(T), T -> None
    Append a new value to the end of a list.
    :param lst: the LinkedList to whose node chain the value will be
                                          appended
    :param new_value: the value to put at the end of lst
    """
    new_node = MutableNode( new_value, None )
    node = lst.head
    if node is None:
        lst.head = new_node
    else:
        successor = node.next
        while successor is not None:
            node = successor
            successor = node.next
        node.next = new_node
    lst.size += 1
```

Figure 5 illustrates the behavior of `append` on our original example, [99,100,101] and 75.

Something to think about: How could you modify the above code so that it uses the size of `lst1` to locate its last node?

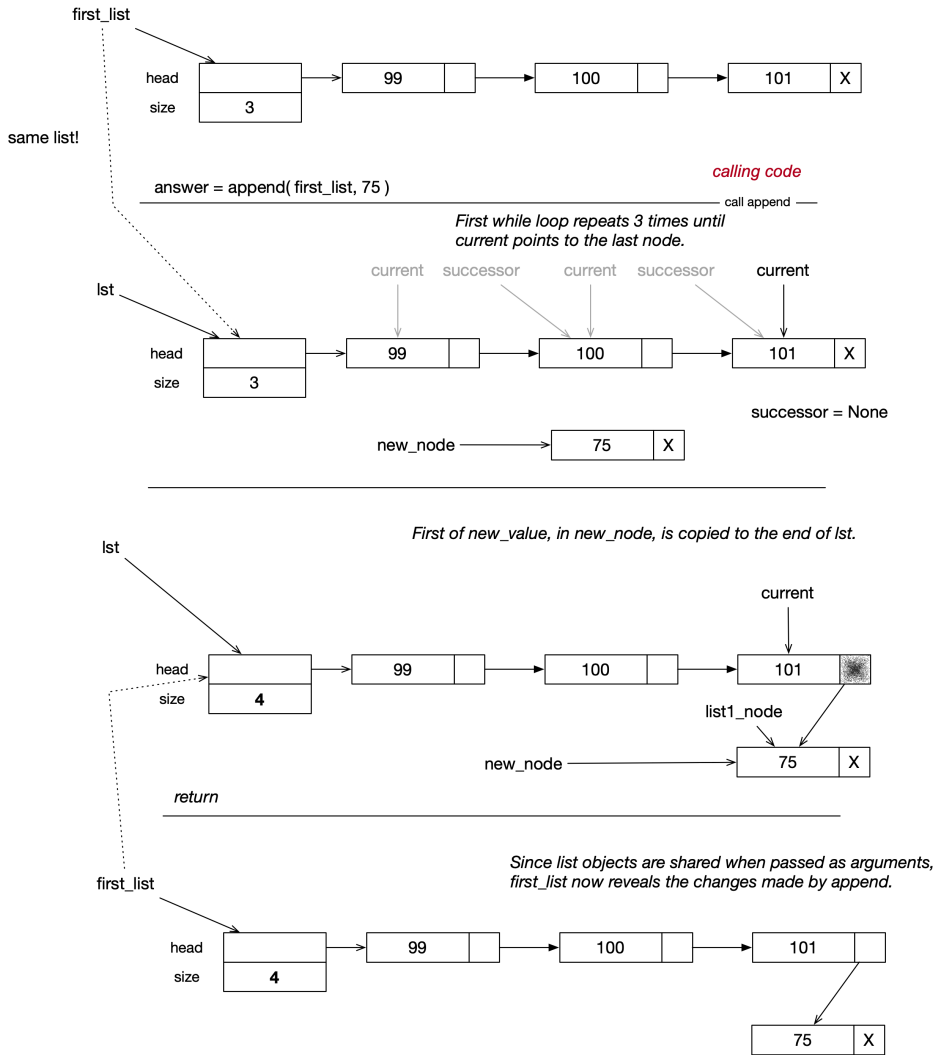

Figure 5: The Iterative append Function, Part 2

3.0.3 Insertion by Index

Insertion now involves patching a new node into an existing list:

1. Create a new node with the given value.
2. Find the node at the given index in the list.
3. Modify the node before that one (if it exists) to refer to the new node.
4. Modify the new node to refer to the found node.

You will notice that we had to remember the node we were looking for, *and* the node to which new nodes are attached. This is a common, and awkward, part of linked sequence mutation algorithms. Often when these algorithms are written down you will see names like *current* and *successor*, or *previous* and *current*.

Unless there is an error (illegal index), the while loop in the code below finishes when `node` is the node just before the insertion point and `successor` is the node just after the

insertion point.

```
def insert_before_index( lst, new_value, index ):
    """
    insert_before_index: Linked(T), Any, Integer -> None
    Stick a new value in front of the node at a certain ordinal
    position in a list.
    """
    new_node = MutableNode( new_value, None )

    # Check for special cases.
    #
    current = lst.head
    if index == 0: # We must modify the head of the LinkedList object.
        new_node.next = current
        lst.head = new_node
    elif current is None:
        raise IndexError( "List is shorter than index " + str( index ) +
                                                    "!" )
    else:
        # Locate the insertion point.
        #
        successor = current.next
        loc = 1
        while successor is not None and loc < index:
            current = successor
            successor = current.next
            loc += 1
        if loc < index: # The list ended prematurely.
            raise IndexError(
                "List is shorter than index " + str( index ) + "!" )
        else:
            # Splice in the new node.
            #
            new_node.next = successor
            current.next = new_node
    lst.size += 1
```

Figure 6 shows an example of insert-before-index.

How might you leverage the `LinkedList`'s `size` value to simplify error-checking in the code above?

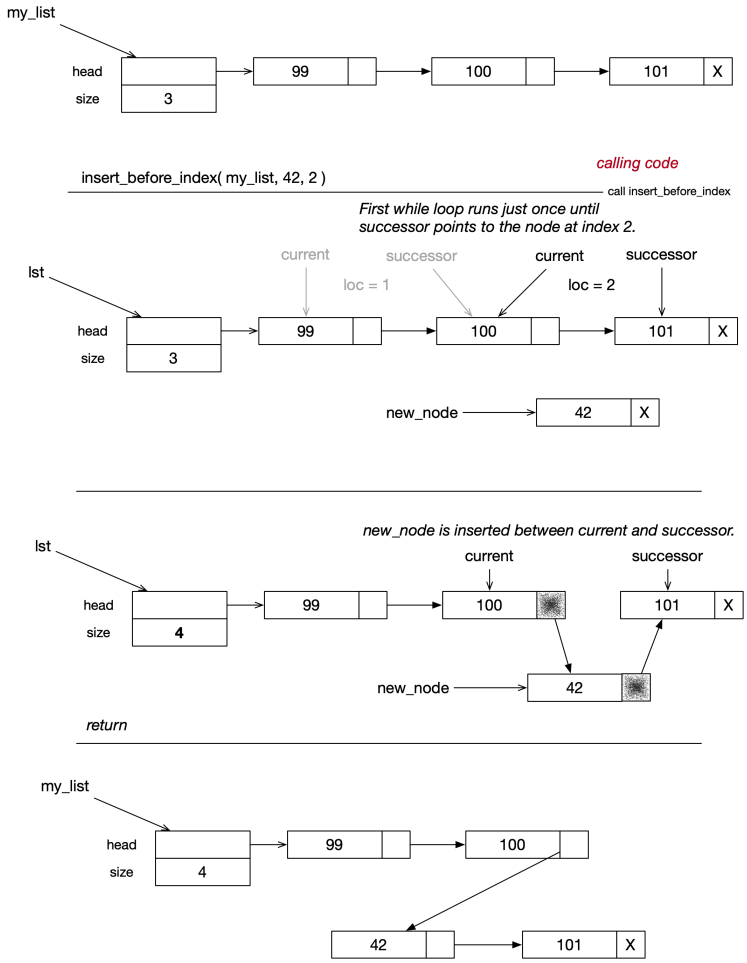

Figure 6: Iterative Insertion Before Index

3.0.4 Removal by Value

Removal of a value from the list is the opposite of insertion, but the algorithm is quite similar in that you have to find a node and also remember where the previous node is so you can perform the removal.

1. Find the node in the list that has the given value.
2. Modify the node before that one (if it exists) to refer to the node after that one.

```
def remove_value( lst, value ):
    """
    remove: Linked(T), Any -> None
    Locate a value in a list and remove it.
    """
    # Check for special cases.
    #
    node = lst.head
    if node is None:
        raise ValueError( "No such value " + str( value ) + " in list!"
                                                          )
```

```
    elif node.value == value:
        lst.head = node.next
    else:
        # Search for the value.
        #
        successor = node.next
        while successor is not None and successor.value != value:
            node = successor
            successor = node.next
        if successor is None:
            raise ValueError( "No such value " + str( value ) + " in
                                                    list!" )
        else:
            # Cut out the node containing the value.
            #
            node.next = successor.next
    lst.size -= 1
```

4 Additional Operations

In the file **immutable_list.py** you will also find other functions implemented.

- **node_types.py**: the node types
- **immutable_list.py**: the functions for `FrozenNode`s
- **linked_list.py**: the `LinkedList` type
- **mutable_list.py**: the functions for `MutableNode`s

5 Testing

Each of the above files contains a test function suitable for the structures and functions defined in it.

6 Why Bother with a New Kind of List?

We already have Python lists that implement sequences. Let's compare the implementations. We have seen that the constructor `FrozenNode` is analogous to `list`'s `insert`, at location 0; they are both basically $O(1)$ operations. We have seen that `value` corresponds to indexing to access the element at position 0; here too they are both $O(1)$. But what about an arbitrary position? Our implementation takes linear time for that, but Python lists only take constant time. We have seen that `next` corresponds to taking a specific slice to get all but the first element. `next` is $O(1)$ but a slice is $O(N)$. Also note that `to_str` corresponds to `str` for lists. Our function and Python's take $O(N)$ time. In some areas where Python is faster, the language runtime achieves this by keeping track of the lengths of lists rather than recomputing them each time. (We did do that in the second half of this lesson.) Finally, our `insert_before_index` and `remove_value` correspond to Python's `insert` and `remove`; all these operations are $O(N)$.

The bottom line is that if general indexing is needed, this implementation is inappropriate. Otherwise they appear to be about the same. Why bother creating our own? There is both pedagogical (teaching) and practical value in doing so.

From a pedagogical point of view, we observe the following.

- Our implementation of linked sequences emphasizes that high level programming language features such as list data structures can be explained in terms of lower level features.
- Linked sequences serve as an introduction to linked data structures in general.
- Linked lists are easier to implement in lower level languages than Python style lists.

From a practical point of view, we observe the following.

- Programming using recursively defined data structures ends writer's block for programmers.
- Linking makes it easy to dynamically extend a data structure.

- Linked sequences can provide a particularly efficient form of immutability, so that we can save old versions of the data structure in coexistence with newer versions. This trait is useful if one wishes to go back to a previous version of a data structure.