CSC111 Lecture 2: Introduction to Linked Lists

Hisbaan Noorani

January 13, 2021

Contents

Here are the implementations of the `LinkedList` class and the `_Node` class you saw in lecture.

```python
from __future__ import annotations
from dataclasses import dataclass
from typing import Any, Callable, Iterable, Optional, Union


@dataclass
class _Node:
    """A node in a linked list.

    Note that this is considered a "private class", one which is only meant
    to be used in this module by the LinkedList class, but not by client code.

    Instance Attributes:
      - item: The data stored in this node.
      - next: The next node in the list, if any.
    """
    item: Any
    next: Optional[_Node] = None  # By default, this node does not link to any other


class LinkedList:
    """A linked list implementation of the List ADT.
    """
    # Private Instance Attributes:
    #   - _first: The first node in this linked list, or None if this list is empty.
    _first: Optional[_Node]

    def __init__(self) -> None:
        """Initialize an empty linked list.
        """
        self._first = None
```

1 Exercise 1: The `LinkedList` and `_Node` classes

The following Python code creates three `_Node` objects and an empty `LinkedList` object.

```
>>> node1 = _Node('a')
>>> node2 = _Node('b')
>>> node3 = _Node('c')
>>> linky = LinkedList()  # linky is empty
```

1. Write code below that uses the above four variables to make `linky` refer to a linked list containing the elements `'a'`, `'b'`, and `'c'`, in that order.

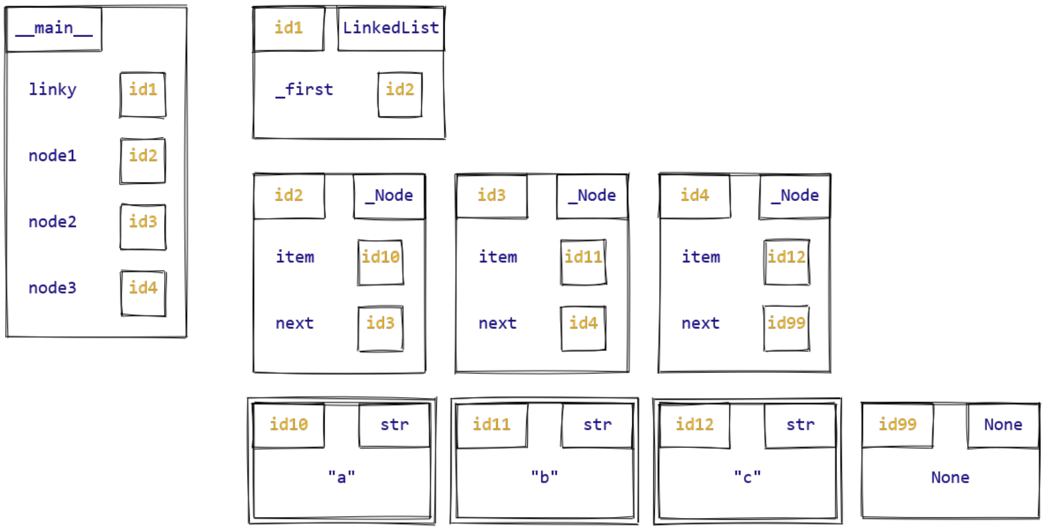
```
>>> node1.next(node2)
>>> node2.next(node3)
>>> linky._first(node1)
```

2. Assume you have executed your code from the previous question. Write the value of each of the following Python expressions. If there would be an error raised, describe the error, but don't worry about matching the exact name/wording as the Python interpreter.

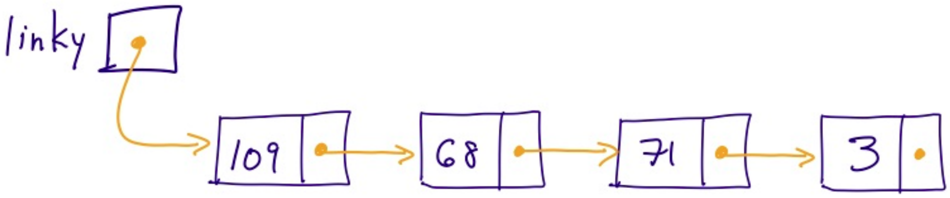
```
>>> node1.item
'a'

>>> node1.next is node2
True

>>> node1.next.item
'b'

>>> node1.next.next.item
'c'

>>> node1.item + node2.item + node3.item
'abc'

>>> node1 + node2 + node3
TypeError: unsupported operand type(s) for +: '_Node' and '_Node'

>>> type(linky)
<class 'LinkedList'>

>>> type(linky._first)
```

```
<class '_Node'>

>>> type(linky._first.item)
<class 'str'>

>>> linky._first.next.next.item
'c'

>>> linky._first.next.next.next.item
Error
```

3. Complete the memory model diagram below to show the state of memory from the above example. Remember that you can make up your own id values, as long as they're all unique. (If you are working on paper, you'll need to redraw the diagram. That's okay, take your time!)



4. Draw an abstract diagram of the linked list `linky`, using the style from lecture (with boxes and arrows).



2 Exercise 2: Linked list traversal

Recall the basic *linked list traversal pattern*:

```python
curr = self._first

while curr is not None:
    ... curr.item ...  # Do something with curr.item
    curr = curr.next
```

In this exercise, you'll implement three new methods using this pattern.

1. Implement the following linked list method.

```python
import math


class LinkedList:
    def maximum(self) -> int:
        """Return the maximum element in this linked list.

        Preconditions:
            - every element in this linked list is a float
            - this linked list is not empty

        >>> linky = LinkedList()
        >>> node3 = _Node(30.0)
        >>> node2 = _Node(-20.5, node3)
        >>> node1 = _Node(10.1, node2)
        >>> linky._first = node1
        >>> linky.maximum()
        30.0
        """
        largest_so_far = -math.inf
        curr = self._first

        while curr is not None:
            if largest_so_far < curr.item:
                largest_so_far = curr.item

            curr = curr.next

        return largest_so_far
```

2. David has attempted to implement the LinkedList.__contains__ method below, but unfortunately his program has a bug.

```python
class LinkedList:
    def __contains__(self, item: Any) -> bool:
        """Return whether item is in this linked list.

        >>> linky = LinkedList()
        >>> linky.__contains__(10)
        False
        >>> node2 = _Node(20)
        >>> node1 = _Node(10, node2)
        >>> linky._first = node1
        >>> linky.__contains__(20)
        True
        """
        curr = self._first

        while curr is not None:
            if curr == item:
                # We've found the item and can return early.
                return True

            curr = curr.next

        # If we reach the end of the loop without finding the item,
        # it's not in the linked list.
        return False
```

(a) What is the error in the above implementation?

`if curr == item:` → `if curr.item == item:`

(b) Which doctest example(s) will fail because of this error?

Anything that is expecting `True` will return false as `curr` will almost never be equal to `item`.

(c) How should we fix this error?

`if curr == item:` → `if curr.item == item:`

3. Finally, let's look at one more `LinkedList` method, `__getitem__`:

```
class LinkedList:
    def __getitem__(self, i: int) -> Any:
        """Return the item stored at index i in this linked list.

        Raise an IndexError if index i is out of bounds.

        Preconditions:
            - i >= 0
        """
```

To implement this method, we're going to need two variables: `curr`, to keep track of the current `_Node`, and `curr_index`, to keep track of the current *index*.

Implement this method below by using an *early return* inside the loop body, similar to `LinkedList.__contains__` above. We've started the implementation for you.

```python
def __getitem__(self, i: int) -> Any:
    """Return the item stored at index i in this linked list.

    Raise an IndexError if index i is out of bounds.

    Preconditions:
        - i >= 0
    """
    curr = self._first
    curr_index = 0

    while curr is not None:
        if curr_index == i:
            # If we reach the right index, return the item
            return curr.item

        curr_index += 1
        curr = curr.next

    # If we reach this point, the list has ended BEFORE
    # we reach index i.
    raise IndexError
```

3 Additional exercises

1. Implement the following linked list method, which is very similar to `LinkedList.sum`.

```python
class LinkedList:
    def __len__(self) -> int:
        """Return the number of elements in this linked list.

        >>> linky = LinkedList()
        >>> linky.__len__()
        0
        >>> node3 = _Node(30)
        >>> node2 = _Node(20, node3)
        >>> node1 = _Node(10, node2)
        >>> linky._first = node1
        >>> linky.__len__()
        3
        """
        curr = self._first
        len_so_far = 0

        while curr is not None:
            curr = curr.next
            len_so_far += 1

        return len_so_far
```

Note: this is yet another Python special method. Unsurprisingly, it gets called when you call the built-in function `len` on a `LinkedList` object. Try it!

2. Here is another linked list method, which allows you to compare the items in two different linked lists.

```
class LinkedList:
    def __eq__(self, other: LinkedList) -> bool:
        """Return whether this list and the other list are equal.

        Two lists are equal when each one has the same number of items, and
        each corresponding pair of items are equal (using == to compare).
        """
        curr_1 = self._first
        curr_2 = other._first

        # Ensure that the lengths are then same using the above len method.
        # If they're not the same length, they cannot be equal.
        # Also the loop condition only works as intended if they're the same len.
        if len(curr_1) != len(curr_2):
            return False

        while curr_1 is not None and curr_2 is not None:
            if curr_1.item != curr_2.item:
                return False

        return True
```

Implement this method by using the linked list traversal pattern, except use *two* loop variables `curr1` and `curr2`, one for each list.

For extra practice, implement this method twice: once using an early return, and once using a compound while loop condition. Which approach do you like better?