Flattening a nested linked list

May 14, 2020

0.1 Flattening a nested linked list

Suppose you have a linked list where the value of each node is a sorted linked list (i.e., it is a *nested* list). Your task is to *flatten* this nested list—that is, to combine all nested lists into a single (sorted) linked list.

First, we'll need some code for generating nodes and a linked list:

In [3]:
```
# Helper code

# A class behaves like a data-type, just like an int, float or any other built-in ones.
# User defined class
class Node:
    def __init__(self, value): # <-- For simple LinkedList, "value" argument will be an
        self.value = value
        self.next = None

    def __repr__(self):
        return str(self.value)

# User defined class
class LinkedList:
    def __init__(self, head): # <-- Expects "head" to be a Node made up of an int or Lin
        self.head = head

    '''
    For creating a simple LinkedList, we will pass an integer as the "value" argument
    For creating a nested LinkedList, we will pass a LinkedList as the "value" argument
    '''
    def append(self, value):

        # If LinkedList is empty
        if self.head is None:
            self.head = Node(value)
            return

        # Create a temporary Node object
        node = self.head
```

```
        # Iterate till the end of the currrent LinkedList
        while node.next is not None:
            node = node.next

        # Append the newly createaed Node at the end of the currrent LinkedList
        node.next = Node(value)


    '''We will need this function to convert a LinkedList object into a Python list of i
    def to_list(self):
        out = []           # <-- Declare a Python list
        node = self.head   # <-- Create a temporary Node object

        while node:        # <-- Iterate untill we have nodes available
            out.append(int(str(node.value))) # <-- node.value is actually of type Node,
            node = node.next

        return out
```

0.1.1 Exercise - Write the two function definitions below

Now, in the cell below, see if you can solve the problem by implementing the `flatten` method.

Hint: If you first create a `merge` method that merges two linked lists into a sorted linked list, then there is an elegant recursive solution.

```
In [ ]: def merge(list1, list2):
            # TODO: Implement this function so that it merges the two linked lists in a single,
            '''
            The arguments list1, list2 must be of type LinkedList.
            The merge() function must return an instance of LinkedList.
            '''
            new_list = LinkedList()
            current_node1 = list1.head
            current_node2 = list2.head
            while current_node1 or current_node2:
                if current_node1 > current_node2:
                    new_list = current_node2
                else:
                    new_list = current_node1
                current_node1 = current_node1.next
                current_node2 = current_node2.next
            return new_list
            pass


        ''' In a NESTED LinkedList object, each node will be a simple LinkedList in itself'''
        class NestedLinkedList(LinkedList):
            def flatten(self):
```

```
        # TODO: Implement this method to flatten the linked list in ascending sorted ord
        if node.next is None:
            return merge(node.value, None) # <-- First argument is a simple LinkedList

        # _flatten() is calling itself untill a termination condition is achieved
        return merge(node.value, self._flatten(node.next))
        pass
```

0.1.2 Test - Let's test your function

Here's some code that will generate a nested linked list that we can use to test the solution:

```
In [3]: # First Test scenario
        ''' Create a simple LinkedList'''
        linked_list = LinkedList(Node(1)) # <-- Notice that we are passing a Node made up of an
        linked_list.append(3) # <-- Notice that we are passing a numerical value as an argument
        linked_list.append(5)

        ''' Create another simple LinkedList'''
        second_linked_list = LinkedList(Node(2))
        second_linked_list.append(4)

        ''' Create a NESTED LinkedList, where each node will be a simple LinkedList in itself'''
        nested_linked_list = NestedLinkedList(Node(linked_list)) # <-- Notice that we are passin
        nested_linked_list.append(second_linked_list) # <-- Notice that we are passing a LinkedL
```

Structure of the nested linked list to be tested `nested_linked_list` should now have 2 nodes. The head node is a linked list containing `1, 3, 5`. The second node is a linked list containing `2, 4`.

Calling `flatten` should return a linked list containing `1, 2, 3, 4, 5`.

```
In [ ]: solution = nested_linked_list.flatten() # <-- returns A LinkedList object

        expected_list = [1,2,3,4,5] # <-- Python list

        # Convert the "solution" into a Python list and compare with another Python list
        assert solution.to_list() == expected_list, f"list contents: {solution.to_list()}"
```

0.1.3 Solution

First, let's implement a `merge` function that takes in two linked lists and returns one sorted linked list. Note, this implementation expects both linked lists to be sorted.

```
In [4]: def merge(list1, list2):
            merged = LinkedList(None)
            if list1 is None:
                return list2
            if list2 is None:
```

```
            return list1
        list1_elt = list1.head
        list2_elt = list2.head
        while list1_elt is not None or list2_elt is not None:
            if list1_elt is None:
                merged.append(list2_elt)
                list2_elt = list2_elt.next
            elif list2_elt is None:
                merged.append(list1_elt)
                list1_elt = list1_elt.next
            elif list1_elt.value <= list2_elt.value:
                merged.append(list1_elt)
                list1_elt = list1_elt.next
            else:
                merged.append(list2_elt)
                list2_elt = list2_elt.next
        return merged
```

Let's make sure merge works how we expect:

```
In [5]: ''' Test merge() function'''
        linked_list = LinkedList(Node(1))
        linked_list.append(3)
        linked_list.append(5)

        second_linked_list = LinkedList(Node(2))
        second_linked_list.append(4)

        merged = merge(linked_list, second_linked_list)
        node = merged.head
        while node is not None:
            #This will print 1 2 3 4 5
            print(node.value)
            node = node.next

        # Lets make sure it works with a None list
        merged = merge(None, linked_list)
        node = merged.head
        while node is not None:
            #This will print 1 3 5
            print(node.value)
            node = node.next
```

```
1
2
3
4
5
```

```
1
3
5
```

Now let's implement `flatten` recursively using merge.

```
In [7]: ''' In a NESTED LinkedList object, each node will be a simple LinkedList in itself'''
        class NestedLinkedList(LinkedList):
            def flatten(self):
                return self._flatten(self.head) # <-- self.head is a node for NestedLinkedList

            '''  A recursive function '''
            def _flatten(self, node):

                # A termination condition
                if node.next is None:
                    return merge(node.value, None) # <-- First argument is a simple LinkedList

                # _flatten() is calling itself untill a termination condition is achieved
                return merge(node.value, self._flatten(node.next)) # <-- Both arguments are a si
```

```
In [8]: ''' Test flatten() function'''
        nested_linked_list = NestedLinkedList(Node(linked_list))
        nested_linked_list.append(second_linked_list)
        flattened = nested_linked_list.flatten()

        node = flattened.head
        while node is not None:
            #This will print 1 2 3 4 5
            print(node.value)
            node = node.next
```

```
1
2
3
4
5
```

0.1.4 Computational Complexity

Lets start with the computational complexity of `merge`. Merge takes in two lists. Let's say the lengths of the lists are N_1 and N_2. Because we assume the inputs are sorted, `merge` is very efficient. It looks at the first element of each list and adds the smaller one to the returned list. Every time through the loop we are appending one element to the list, so it will take $N_1 + N_2$ iterations until we have the whole list.

The complexity of `flatten` is a little more complicated to calculate. Suppose our `NestedLinkedList` has N linked lists and each list's length is represented by $M_1, M_2, ..., M_N$.

We can represent this recursion as:

$merge(M_1, merge(M_2, merge(..., merge(M_{N-1}, merge(M_N, None)))))$

Let's start from the inside. The inner most merge returns the *nth* linked list. The next merge does $M_{N-1} + M_N$ comparisons. The next merge does $M_{N-2} + M_{N-1} + M_N$ comparisons.

Eventually we will do N comparisons on all of the M_N elements. We will do $N - 1$ comparisons on M_{N-1} elements.

This can be generalized as:

$$\sum_{n}^{N} n * M_n$$

In [ ]: