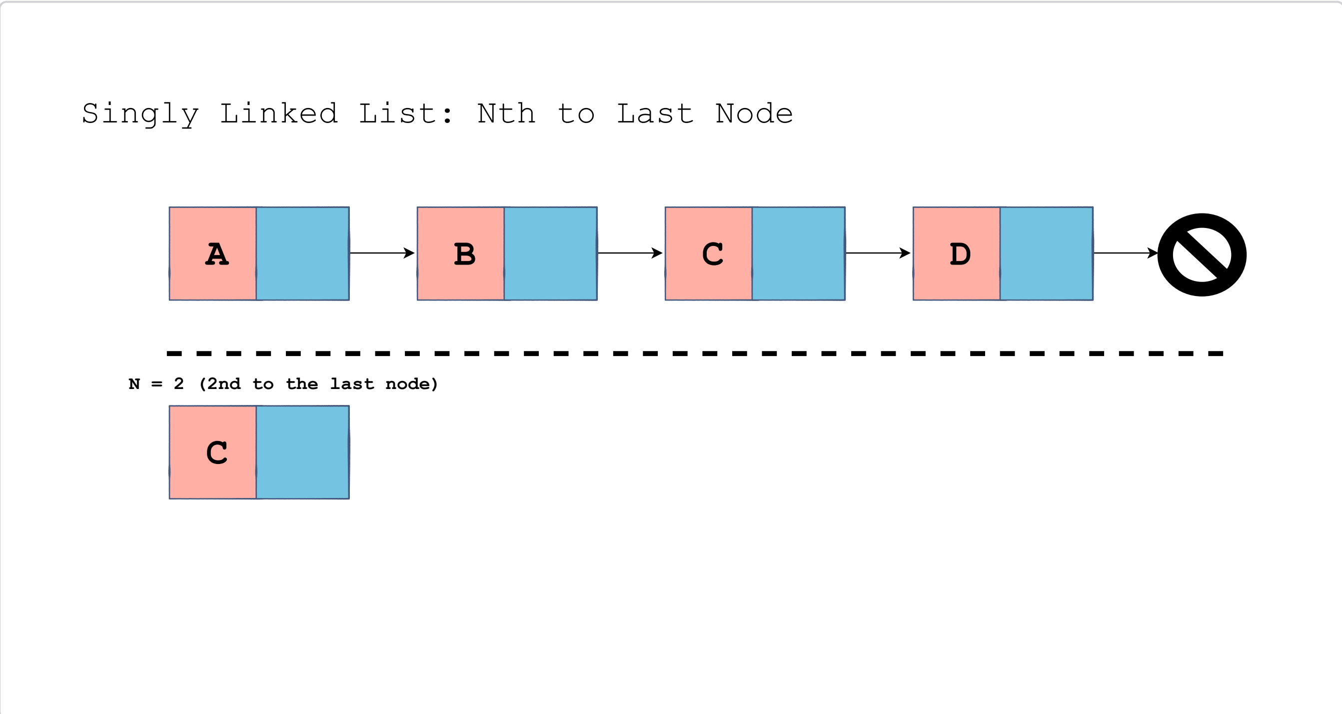
Nth-to-Last Node

In this lesson, we will learn how to get the Nth-to-Last Node from a given linked list.

We'll cover the following

In this lesson, we are going to find how to get the *Nth-to -Last Node* from a linked list. First of all, we'll clarify what we mean by *Nth-to -Last Node* in the illustration below:



As you can see from the illustration above, if `N` equals `2`, we want to get the second to last node from the linked list.

We will be using two solutions to solve this problem.

Solution 1

We'll break down this solution in two simple steps:

1. Calculate the length of the linked list.
2. Count down from the total length until `n` is reached.

For example, if we have a linked list of length four, then we'll begin from the head node and decrement the calculated length of the linked list by one as we traverse each node in the linked list. We'll only stop on the node when our count becomes equal to `n`.

Implementation

Let's try implementing this solution in Python:

```
def print_nth_from_last(self, n):
  total_len = self.len_iterative()

  cur = self.head
  while cur:
    if total_len == n:
      print(cur.data)
      return cur.data
    total_len -= 1
    cur = cur.next
  if cur is None:
    return
```

print_nth_from_last(self, n)

Explanation

The method `print_nth_from_last` only takes in `n` as an input parameter. You are already familiar with how to calculate the length of a linked list from a previous lesson. We'll use the class method `self.len_iterative()` that we have implemented before to calculate the length of the linked list and store it in the variable named `total_len` on **line 2**.

So far, we have completed step 1 of solution 1. Let's move on to the second step. `cur` is initialized to `self.head` on **line 4**. Next, we have a `while` loop on

line 5 which will traverse the entire linked list or stop after traversing n nodes. In the body of the `while` loop, we check if `total_len` equals `n` which is our primary goal for this method. If `total_len` equals `n`, we print the `data` of the current node (`cur`) and return the data of the current node from the method. Otherwise, we decrement `1` from `total_len` on **line 9** and update `cur` to the next node on **line 10** to traverse the linked list.

If, however, we reach the end of the linked list but `total_len` never becomes equal to `n`, then we handle this on **lines 11-12**. In this case, we check if `cur` is `None` and we return from the method.

Solution 2

That was all about Solution 1. Let's proceed with Solution 2, which can be described as follows:

There will be a total of two pointers `p` and `q`:

- `p` will point to the head node.
- `q` will point `n` nodes beyond head node.

Next, we'll move these pointers along with the linked list one node at a time. When `q` will reach `None`, we'll check where `p` is pointing, as that is the node we want.

Implementation

Let's make it clearer by implementing it in Python:

```
def print_nth_from_last(self, n):
    p = self.head
    q = self.head

    count = 0
    while q:
        count += 1
        if(count>=n):
            break
        q = q.next

    if not q:
        print(str(n) + " is greater than the number of nodes in list.")
        return

    while p and q.next:
        p = p.next
        q = q.next
```

```
        return p.data
```

print_nth_from_last(self, n)

Explanation

We initialize `p` and `q` to `self.head` on **line 2** and **line 3** respectively. According to the algorithm, we have to make `q` point to `n` nodes beyond the head. Therefore, we initialize `count` to `0` on **line 5** and using the `while` loop on **line 6**, we keep updating `q` to the next node (**line 10**) until and unless `count`, which is being incremented in each iteration (**line** 7), becomes equal to or greater than `n`. Additionally, the `while` loop will terminate if `q` reaches `None`. To handle this case, we put up a condition on **line 12** which checks if `q` is `None` or not. If `q` is `None`, then this implies that `n` is greater than the length of the linked list and getting the *Nth-to-last* node is not possible. We return from the method on **line 14**.

Now let's jump to the main part of the implementation. In the `while` loop on **line 16**, we keep updating `p` and `q` to the next nodes. This `loop` will terminate when either `p` or `q.next` equals `None`. As a result, we return `p.data` which will be the *Nth-to-last* node according to our algorithm.

Both the solutions have been made part of the `LinkedList` class in the coding widget below. You can call the solution of your choice by passing in `1` or `2` as `method` to `print_nth_from_last(self, n, method)`. Feel free to play around with any of the methods in the `LinkedList` implementation below:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None


class LinkedList:
    def __init__(self):
        self.head = None

    def print_list(self):
        cur_node = self.head
        while cur_node:
            print(cur_node.data)
            cur_node = cur_node.next

    def append(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = new_node
            return
```

```
        last_node = self.head
        while last_node.next:

            last_node = last_node.next
        last_node.next = new_node

    def prepend(self, data):
        new_node = Node(data)

        new_node.next = self.head
        self.head = new_node

    def insert_after_node(self, prev_node, data):

        if not prev_node:
            print("Previous node does not exist.")
            return

        new_node = Node(data)

        new_node.next = prev_node.next
        prev_node.next = new_node

    def delete_node(self, key):

        cur_node = self.head

        if cur_node and cur_node.data == key:
            self.head = cur_node.next
            cur_node = None
            return

        prev = None
        while cur_node and cur_node.data != key:
            prev = cur_node
            cur_node = cur_node.next

        if cur_node is None:
            return

        prev.next = cur_node.next
        cur_node = None

    def delete_node_at_pos(self, pos):
        if self.head:
            cur_node = self.head

            if pos == 0:
                self.head = cur_node.next
                cur_node = None
                return

            prev = None
            count = 1
            while cur_node and count != pos:
                prev = cur_node
                cur_node = cur_node.next
                count += 1

            if cur_node is None:
                return
```

```python
            prev.next = cur_node.next
            cur_node = None

    def len_iterative(self):

        count = 0
        cur_node = self.head

        while cur_node:
            count += 1
            cur_node = cur_node.next
        return count

    def len_recursive(self, node):
        if node is None:
            return 0
        return 1 + self.len_recursive(node.next)

    def swap_nodes(self, key_1, key_2):

        if key_1 == key_2:
            return

        prev_1 = None
        curr_1 = self.head
        while curr_1 and curr_1.data != key_1:
            prev_1 = curr_1
            curr_1 = curr_1.next

        prev_2 = None
        curr_2 = self.head
        while curr_2 and curr_2.data != key_2:
            prev_2 = curr_2
            curr_2 = curr_2.next

        if not curr_1 or not curr_2:
            return

        if prev_1:
            prev_1.next = curr_2
        else:
            self.head = curr_2

        if prev_2:
            prev_2.next = curr_1
        else:
            self.head = curr_1

        curr_1.next, curr_2.next = curr_2.next, curr_1.next

    def print_helper(self, node, name):
        if node is None:
            print(name + ": None")
        else:
            print(name + ":" + node.data)

    def reverse_iterative(self):

        prev = None
        cur = self.head
        while cur:
            nxt = cur.next
```

```
            cur.next = prev

            self.print_helper(prev, "PREV")
            self.print_helper(cur, "CUR")
            self.print_helper(nxt, "NXT")
            print("\n")

            prev = cur
            cur = nxt
        self.head = prev

    def reverse_recursive(self):

        def _reverse_recursive(cur, prev):
            if not cur:
                return prev

            nxt = cur.next
            cur.next = prev
            prev = cur
            cur = nxt
            return _reverse_recursive(cur, prev)

        self.head = _reverse_recursive(cur=self.head, prev=None)

    def merge_sorted(self, llist):

        p = self.head
        q = llist.head
        s = None

        if not p:
            return q
        if not q:
            return p

        if p and q:
            if p.data <= q.data:
                s = p
                p = s.next
            else:
                s = q
                q = s.next
            new_head = s
        while p and q:
            if p.data <= q.data:
                s.next = p
                s = p
                p = s.next
            else:
                s.next = q
                s = q
                q = s.next
        if not p:
            s.next = q
        if not q:
            s.next = p
        return new_head

    def remove_duplicates(self):

        cur = self.head
```

```
        prev = None

        dup_values = dict()

        while cur:
            if cur.data in dup_values:
                # Remove node:
                prev.next = cur.next
                cur = None
            else:
                # Have not encountered element before.
                dup_values[cur.data] = 1
                prev = cur
            cur = prev.next

    def print_nth_from_last(self, n, method):
        if method == 1:
            #Method 1:
            total_len = self.len_iterative()
            cur = self.head
            while cur:
                if total_len == n:
                   #print(cur.data)
                    return cur.data
                total_len -= 1
                cur = cur.next
            if cur is None:
                return

        elif method == 2:
            # Method 2:
            p = self.head
            q = self.head

            count = 0
            while q:
                count += 1
                if(count>=n):
                    break
                q = q.next

            if not q:
                print(str(n) + " is greater than the number of nodes in list.")
                return

            while p and q.next:
                p = p.next
                q = q.next
            return p.data

llist = LinkedList()
llist.append("A")
llist.append("B")
llist.append("C")
llist.append("D")

print(llist.print_nth_from_last(4,1))
print(llist.print_nth_from_last(4,2))
```

Hope you are clear about both the solutions. Let's move on to another problem about singly linked lists in the next lesson. See you there!