Count Occurrences

In this lesson, we will learn how to count occurrences of a data element in a linked list.

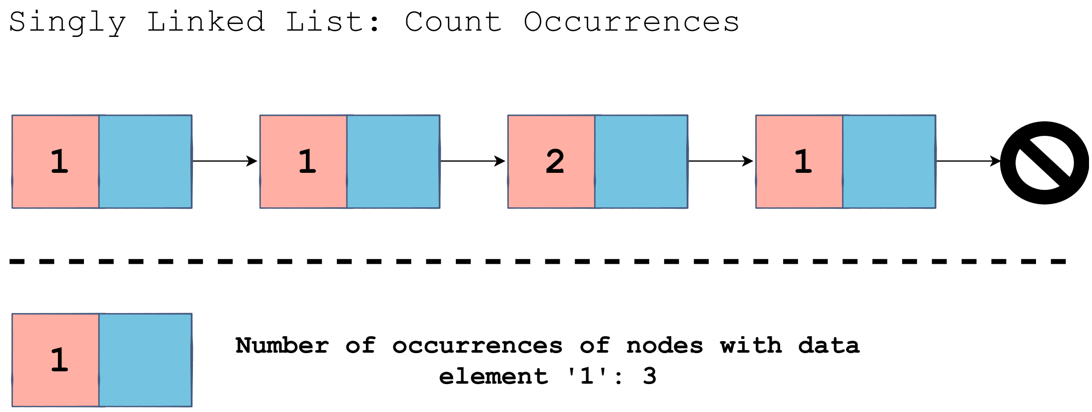
WE'LL COVER THE FOLLOWING

In this lesson, we investigate how to count the occurrence of nodes with a specified data element. We will consider how one may solve this problem in both an iterative and recursive manner, and we will code the solution to both of these approaches in Python.

As an example, have a look at the illustration below where we have a linked list with the following elements:

```
1 - 1 - 2 - 1
```

You can see that the number of occurrences of `1` in the linked list is `3`.



Iterative Implementation

Our approach to iteratively solving this problem is straightforward. We'll traverse the linked list and count the occurrences as we go along. Let's go ahead and code it in Python!

```
def count_occurences_iterative(self, data):
  count = 0
  cur = self.head
  while cur:
      if cur.data == data:
          count += 1
      cur = cur.next
  return count
```

count_occurences_iterative(self, data)

Explanation

`count_occurences_iterative` takes in `data` as one of the input parameters. On **line 2** and **line 3**, `count` and `cur` are initialized to `0` and `self.head` respectively. `count` will keep count of the number of occurrences of `data` while `cur` is used for the linked list traversal. The `while` loop on **line 4** runs until `cur` becomes equal to `None`. `cur` is updated to `cur.next` in each iteration (**line** 7).

On **line 5**, there is a simple check to see if `cur.data` is equal to `data`. If the data of the current node is equal to `data` that we are looking for, `count` is incremented by `1`. This is how the number of occurrences will be counted. On **line 8**, `count` after being updated in the `while` loop, is returned from the method.

Hope the iterative implementation is clear.

Recursive Implementation

Let's turn to the recursive implementation now.

```
def count_occurences_recursive(self, node, data):
  if not node:
      return 0
  if node.data == data:
      return 1 + self.count_occurences_recursive(node.next, data)
  else:
      return self.count_occurences_recursive(node.next, data)
```

count_occurences_recursive(self, node, data)

Explanation

The general idea is nearly the same except that it has been implemented recursively. The `count_occurences_recursive` method takes in `self`, `node`, and `data` as input parameters. `self` is passed because it is a class method. `node` will be the current node that we are on while `data` is the data that we have to check the occurrences of.

The base case written on **lines 2-3** is an empty linked list. An empty linked list has no number of occurrences of any value. If we hit a `None` node, we return `0` from the method.

However, if the current node (`node`) is not `None`, then we check if `node.data` is equal to `data`. If it is, then we make a recursive call to `count_occurences_recursive` and pass the next node of the current node (`node.next`) and `data` to it. Also, we add `1` to whatever will be returned from the recursive call and return it from the method. This is because we have to make the occurrence of the current node count as `node.data` match the `data`. However, if `node.data` does not match `data`, we don't add `1` and simply return whatever is returned from `self.count_occurences_recursive(node.next, data)` (**line** 7).

Overall, for a non-empty linked list, we only look at the first node in the linked list. If it has that value, we've found one match. We let the recursive calls count the number of occurrences of the desired value in the remaining linked list, i.e., one that starts at `cur.next`. We either add one to it, or we don't.

Let's test both the implementations in the code widget below. We pass the head node as an argument to the recursive implementation.

```
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

            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

    def count_occurences_iterative(self, data):
        count = 0
        cur = self.head
        while cur:
            if cur.data == data:
                count += 1
            cur = cur.next
        return count

    def count_occurences_recursive(self, node, data):
        if not node:
            return 0
        if node.data == data:
            return 1 + self.count_occurences_recursive(node.next, data)
        else:
            return self.count_occurences_recursive(node.next, data)


llist = LinkedList()
llist.append(1)
llist.append(2)
llist.append(3)
llist.append(4)
llist.append(5)
llist.append(6)

llist_2 = LinkedList()
llist_2.append(1)
llist_2.append(2)
llist_2.append(1)
llist_2.append(3)
llist_2.append(1)
llist_2.append(4)
llist_2.append(1)
print(llist_2.count_occurences_iterative(1))
print(llist_2.count_occurences_recursive(llist_2.head, 1))
```

Hope you were able to understand the lesson! See you in the next one.