Is Palindrome

In this lesson, we'll learn how to determine whether a singly linked list is a palindrome or not.

We'll cover the following

In this lesson, we investigate how to determine if a singly linked list is a palindrome, that is if the data held at each of the nodes in the linked list can be read forward from the head or backward from the tail to generate the same content. We will code the solution to both of these approaches in Python.

First of all, let's learn what a palindrome is. A palindrome is a word or a sequence that is the same from the front as from the back. For instance, *racecar* and *radar* are palindromes. Examples of non-palindromes are *ABC*, *hello*, and *test*.

For this lesson, we'll solve the *Is Palindrome* problem in Python by using three solutions.

Now let's go ahead and check out *Solution 1*:

Solution 1: Using a string

Solution 1: Using a string

Implementation

```
1  def is_palindrome(self):
2    # Solution 1:
3    s = ""
4    p = self.head
5    while p:
6      s += p.data
7      p = p.next
8    return s == s[::-1]
```

is_palindrome(self)

Explanation

The first solution makes use of a string, `s`, which is initialized to an empty string on **line 3**. Then we initialize `p` to `self.head` on **line 4**. On **line 5**, a `while` loop is set which will run until `p` becomes `None`. In the `while` loop, the data of the current node is concatenated to the string `s` on **line 6** while we keep updating `p` to its next node on **line** 7. Finally, we return a boolean value based on the following condition on **line 8**:

```
s == s[::-1]
```

The condition above checks if `s` is equivalent to the reverse of `s` and returns `True` or `False` accordingly. `s[::-1]` is just a shorthand in Python for reversing a string. As you can see the first method is fairly simple, let's move on to the next solution.

Solution 2: Using a stack

The second method is a little bit different than the first method, but it has a similar spirit. Solution 2 will involve a stack, but, we will not make use of a stack per se. Instead, we'll use a Python list and its `append` and `pop` methods. These operations are similar to the `push` and `pop` methods of the stack data structure.

In this method, we'll move along the list, but instead of concatenating onto a string, we'll push onto the stack.

Implementation

Let's code this up!

```
1  def is_palindrome(self):
2    # Solution 2:
3    p = self.head
4    s = []
5    while p:
6      s.append(p.data)
7      p = p.next
8    p = self.head
9    while p:
10     data = s.pop()
11     if p.data != data:
12       return False
13     p = p.next
14   return True
```

is_palindrome(self)

Explanation

On **lines 3-4**, we declare `p` and `s` as `self.head` and `[]` respectively. `s` is a Python list, but we'll use it as a stack. On **line 5**, we set up a `while` loop in which we append (push) the data of every node onto `s` (**line 6**) by traversing the linked list with the help of `p` which is updated to its next node in every iteration (**line** 7).

On **line 8**, we assign `p` to `self.head` as we want to traverse the list again using the `while` loop on **line 9**. In this `while` loop, we pop from `s` and save it into the `data` variable. Note that the `pop` method will return us the latest element pushed in. Therefore, we are going to get values starting from the last item of the linked list. We are traversing the list from the head while `s` gives us values from the end. We make use of this arrangement and check if `p.data` is equal to `data` on **line 11**. If it is, we continue to the next node by updating `p` to `p.next` on **line 13**. On the other hand, if `p.data` is not equal to `data` in any iteration, we'll return `False` from the method (**line 12**) as we've hit a case where the data from the front is not the same as the data when reading from the back. If we traverse the entire linked list and do not encounter any such case, we'll return `True` from the method after the `while` loop on **line 14**.

This was all about the second solution. Let's move on to the final solution!

Solution 3: Using Two Pointers

In this method, we'll have two pointers `p` and `q`. `p` will initially point to the head of the list and `q` to the last node of the list. Next, we'll check the data elements at each of these nodes that are being pointed to by `p` and `q` and see if they are equal to each other. If they are, we'll progress `p` by one, and we'll move `q` by one in reverse until we get to a point where `p` and `q` either meet or essentially can't move any further without crossing.

Implementation

Let's jump to the coding part now.

```
def is_palindrome(self):
  if self.head:
    p = self.head
    q = self.head
    prev = []

    i = 0
    while q:
      prev.append(q)
      q = q.next
      i += 1
    q = prev[i-1]

    count = 1

    while count <= i//2 + 1:
      if prev[-count].data != p.data:
        return False
      p = p.next
      count += 1
    return True
  else:
    return True
```

is_palindrome(self)

Explanation

On **line 2**, we check if the linked list is empty or not. If it is, execution jumps to **lines 22-23** and `True` is returned from the method. Otherwise, we initialize `p` and `q` by making them point to the head node on **lines 3-4** while another variable `prev` is initialized to an empty list on **line 5**. `prev` will help us in making `q` move to the previous node in each iteration in the `while` loop on

line 16.

On **line** 7, `i` is initialized to `0`. The `while` loop on **line 8** will help us in making `q` move to the end of the linked list. While doing so, we also append the nodes to `prev` list on **line 9** and update `q` to `q.next` to traverse the linked list on **line 10**. In the same loop, which runs until `q` is `None`, we increment `i` by `1`. `i` helps us in indexing on **line 12** where we set `q` to the last node in the `prev` list where `i-1` is the index of the last node.

Now that both `p` and `q` are rightly positioned, let's jump to the last `while` loop on **line 16**. The condition for the `while` loop makes use of the `count` variable which is initialized to `1` on **line 14**. The `while` loop will run until `count` is less than or equal to `i//2 + 1` (approximately half of the list). In the `while` loop on **line 17**, we check if `prev[-count].data` is not equal to `p.data`. Let's first see what we mean by `prev[-count].data`. It is basically the data of a node present in the `prev` list. The node depends on the index `-count` which, for the first iteration, will be `-1`, i.e., the last element in the `prev` list. This implies that we are comparing elements from the end of the linked list to the start of the list and checking if they are the same and form a palindrome. If they are not the same, we return `False` from the method on **line 18**. If they are the same, we proceed to the next iteration and update `p` to `p.next` on **line 19**. Additionally, we increment `count` by `1` on **line 20** so that the second iteration compares the second to last element with the second element and so on. If in all the iterations, we do not encounter the case where the compared elements are not the same, and we exit the `while` loop, we return `True` on **line 21** to confirm the linked list as a palindrome.

Below we have the `is_palindrome` method as part of the `LinkedList` class. It takes an integer as an input which specifies the solution we want to use. For instance, `is_palindrome(1)` will execute solution 1 to solve the problem. We verify the methods on two cases `radar` and `abc`. `is_palindrome` should return `True` in the first case while it should return `False` in the second case. Go ahead and use the solution of your choice to play with the methods!

```
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 rotate(self, k):
        if self.head and self.head.next:
            p = self.head
            q = self.head
            prev = None
            count = 0

            while p and count < k:
                prev = p
                p = p.next
                q = q.next
                count += 1
            p = prev
            while q:
                prev = q
                q = q.next
            q = prev

            q.next = self.head
            self.head = p.next
            p.next = None

    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)

    def is_palindrome_1(self):
        # Solution 1:
        s = ""
        p = self.head
        while p:
            s += p.data
            p = p.next
        return s == s[::-1]

    def is_palindrome_2(self):
        # Solution 2:
        p = self.head
        s = []
        while p:
             s.append(p.data)
             p = p.next
        p = self.head
        while p:
            data = s.pop()
```

```
            if p.data != data:
                return False
            p = p.next
        return True

    def is_palindrome_3(self):
        if self.head:
            p = self.head
            q = self.head
            prev = []

            i = 0
            while q:
                prev.append(q)
                q = q.next
                i += 1
            q = prev[i-1]

            count = 1

            while count <= i//2 + 1:
                if prev[-count].data != p.data:
                    return False
                p = p.next
                count += 1
            return True
        else:
            return True

    def is_palindrome(self,method):
        if method == 1:
            return self.is_palindrome_1()
        elif method == 2:
            return self.is_palindrome_2()
        elif method == 3:
            return self.is_palindrome_3()


# Example palindromes:
# RACECAR, RADAR

# Example non-palindromes:
# TEST, ABC, HELLO

llist = LinkedList()


llist_2 = LinkedList()
llist_2.append("A")
llist_2.append("B")
llist_2.append("C")

print(llist.is_palindrome(1))
print(llist.is_palindrome(2))
print(llist.is_palindrome(3))
print(llist_2.is_palindrome(1))
print(llist_2.is_palindrome(2))
print(llist_2.is_palindrome(3))
```

I hope you were able to comprehend and appreciate the three solutions. Next up we have a challenge for you, so brace yourself!