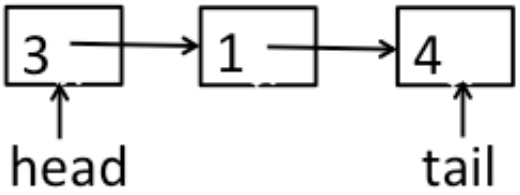
CS417 Lab 14: Linked List with Sentinel Nodes

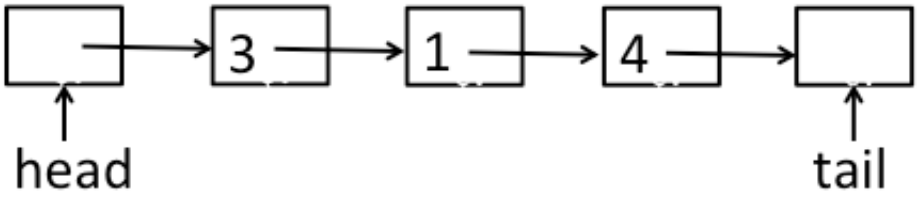
Introduction

Suppose you have an ordinary linked list, with both a `head` and `tail` pointer:



As you may have noticed, working with such a list can be awkward: many operations (`add_head`, `add_tail`, `insert`, `pop`) need special code to check if the list empty.

On the other hand, we could add two extra "sentinel" nodes at the head and tail. A sentinel node is an ordinary node, but it doesn't hold any data (its data field is `None`).



Using sentinels costs us a bit more memory, but it simplifies the code. **That** is the message of this lab.

Getting Started

Begin by creating a folder for your work. Then, go to canvas, find CS417, Modules, find the lab, and download these files:

- `list_node.py`
- `plain_linked_list.py`
- `sentinel_list.py`

Don't change `plain_linked_list.py` or `list_node.py`; they are just there for reference. Open `sentinel_list.py`, and make your changes.

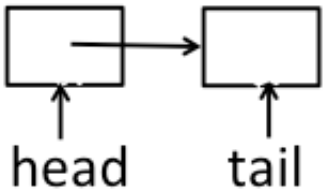
Exercises

When you start, the file `sentinel_list.py` pretty much implements an ordinary linked list without sentinels.

After you complete the exercises, it should use sentinels properly. Look at these methods in `sentinel_list.py`:

0. `__init__()`: this begins with an empty list, and fills it. **I already implemented this one**. It used to set both `self._head` and `self._tail`, to `None`.

   Notice that it now initially creates the two sentinel nodes, and links `head` to `tail`. That's an empty list:



---

1. `__str()___` : this method returns a string version of the list, by visiting all the nodes. Make these changes:

   a. start at `self._head._next` (skip the head sentinel)
   b. keep going while `current` isn't `self._tail` (skip the tail sentinel)
   c. change the test for adding `', '`

---

2. `first()` : this returns the first value. The first value is now **after** the head sentinel. Make this change:

   - get value of `self._head._next`, instead of `self._head`

---

3. `is_empty()` : tells you if the list is empty. Make this change:

   - compare `self._head._next == self._tail`, instead of `self._head == None`

---

4. `add_head()` : adds a value at the beginning of the list. Again, the first value is **after** the head sentinel. Make these changes:

   a. `new_node._next` should point to `self._head._next`, instead of `self._head`
   b. `self._head._next` should be changed, instead of `self._head`
   c. DELETE code that checks for empty tail (simpler code!).

---

5. `add_tail()` : add a value at the end of the list. Here, you must insert a node **before** the tail sentinel. You will need to walk down the list, from the head.

   Make these changes:

   a. `new_node._next` should be `self._tail`
   b. delete all code after that (it's all different)
   c. walk down the list:

- prev starts at `self._head`
- keep going while `prev._next` is not `self._tail`
- after the loop, `prev._next` should now be `new_node`

---

6. `size()` : get the length of the list. Here, we must skip **both** sentinels (don't count them). Modifications: you are visiting only the data nodes, so this is similar to modifications in `__str__()`.

---

7. `find_node()` : returns the node that contains the given value, or `None` if the value does not occur. Modifications: again, similar to modifications in `__str__()`.

---

8. `insert()` : adds a node with the given value, at the given index. Modifications:

    a. delete the special-case code that checks for an empty list
    b. since the head sentinel is an extra node, the `for` loop must do *one more* iteration.

---

9. `pop()` : delete the last node, and return its old value. Modifications:

    a. remove the special-case code for an empty list
    b. you want to stop at the victim's predecessor, so the loop runs while `prev._next._next` is not `self._tail`
    c. don't update the tail pointer! That tail sentinel never changes.

---

10. `remove(x)` : (*Bonus 10%*) remove the node that contains `x`. Modifications:

    Figure this one out.

IMPORTANT!

Notice that the fields begin with an underscore:

- `self._head` instead of `self.head`
- `current._next` instead of `current.next`

Careful! python will let you write `current.next = <something>`, which *adds a* **new** *attribute* `next` to the object `current`. But that doesn't modify `current._next`, which is what you want.

Because of the tail sentinel, the last **data** node doesn't have a `None` pointer. If you did your work correctly, you will **never** be comparing **any** pointers to `None`.

The Moral of the Story

After you finish, `sentinel_list.py` should be smaller and simpler than `plain_linked_list.py`. We are trading more memory (two extra nodes) for less complexity.

Turning in your Work

When you finish, go to `mycourses.unh.edu`, find CS417 and the lab, click “Submit”, and upload `sentinel_list.py`.