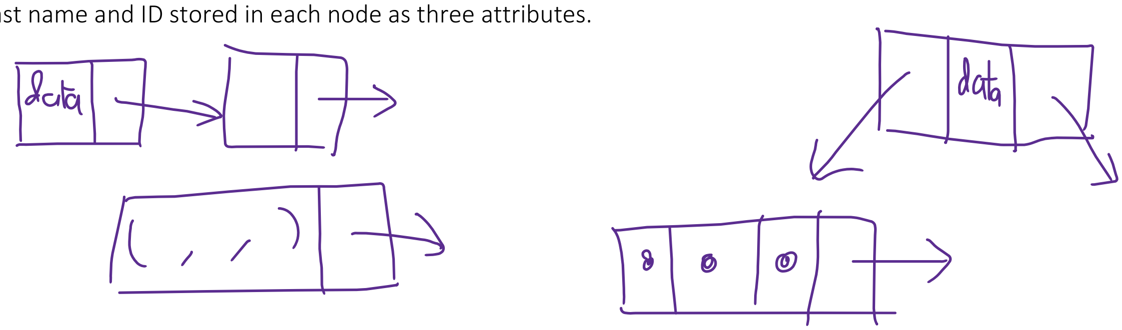
Lecture 7

Linked implementation and the Node class

- In the previous class, we discussed the implementation of a list using array. There is another way to implement a list: using nodes. An implementation with nodes is called a linked implementation, for example, a list that is implemented using nodes is called a LinkedList.
- A most simple node class includes two attributes: a piece of data and a pointer that points to the next node. You may add more data or more pointers as needed.
  - For example, we have a list of students' information. You may have each student's First name, Last name and ID stored in a tuple, and each node contains one tuple; or you may have each student's First name, Last name and ID stored in each node as three attributes.



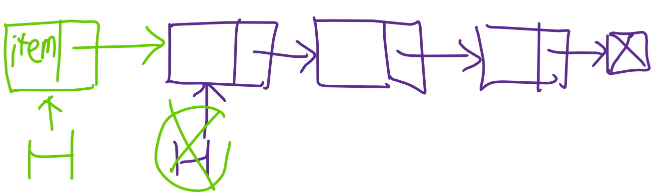
- As an aside, what are the differences between lists, tuples, and sets?
  - A list: [3, 4, 3, 7, 5, 2, 2], may have duplicate items, items are ordered, and the location of each item doesn't have any meaning.
  - A tuple: ("Xiaolang", "Wang", "A20XXXXXX"), items are ordered, location of each item has a certain meaning.
  - A set: {3, 4, 3, 7, 5, 2, 2} = {2, 4, 7, 5, 3} doesn't allow duplicate items, items are not ordered.

A basic design of LinkedList

- Construction method
  - We start with an empty list. A pointer "head" points to None and an integer attribute "size", which records the number of items in the LinkedList has value 0.
  - Whenever we are given a LinkedList, it is like we are given the head of the list together with its size.

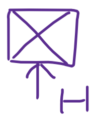
- __len__()
  - So that we can call len(), we need to implement this dunder function. Simply return size. This operation takes $O(1)$ time.

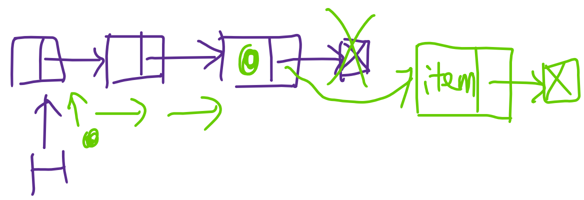
- prepend(item)
  - Add the item to the head of list. This operation *takes* $O(1)$ time.



- __iter__()
  - While the list has next item, we yield the data inside of that item.
- __repr__()
  - So that we can print out the items in a LinkedList, we need to implement this dunder function.

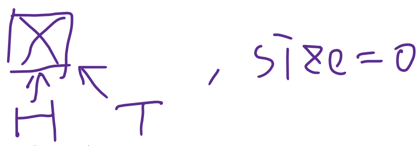
- append(item)
  - Add item to the tail of the LinkedList. We need to scan the list to find the tail node first. This procedure takes $O(n)$ time.



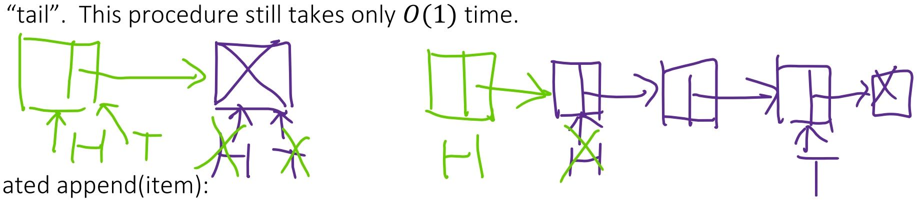


<u>LinkedList with tail pointer</u>

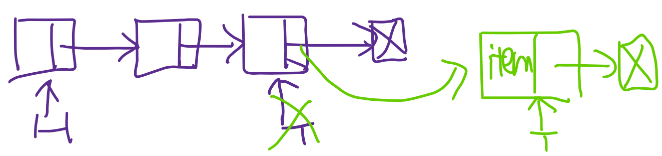
- To reduce the time complexity of append(item), we maintain an extra pointer in the LinkedList that points to the tail of the list.
- Updated construction method:
  - We start with "head" and "tail" both pointing to None.



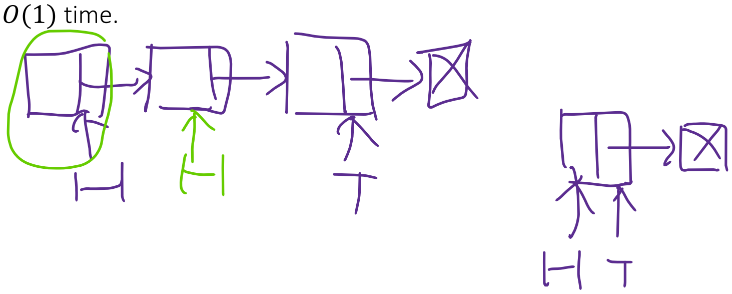
- Updated prepend(item):
  - We now need to check whether we just prepended the first *item* to the list, if yes, we also need to move "tail". This procedure still takes only $O(1)$ time.



- Updated append(item):
  - We can now simply append the item after the "tail", and the item becomes the new tail. This operation takes only $O(1)$ time.



- Deletion
  - delete_head()
    - Here, we use a new keyword "assert". Assert is usually used while studying or debugging, it follows by a condition, when the condition is not satisfied, it returns an error massage.
    - So that we don't delete from an empty LinkedList, we assert len(self) > 0 or self.size > 0 here.
    - We can simply let "head" points to "head.next"; if we deleted the last item, set "tail" to None too. This operation takes only $O(1)$ time.



  - delete_tail()

- If there is only one item, we can simply call delete_head() to delete the last item.
- Otherwise, we can use a while loop to find the node before "tail" and set that node to be the new tail. This operation takes $\Theta(n)$ time.