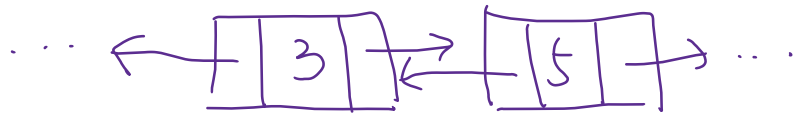
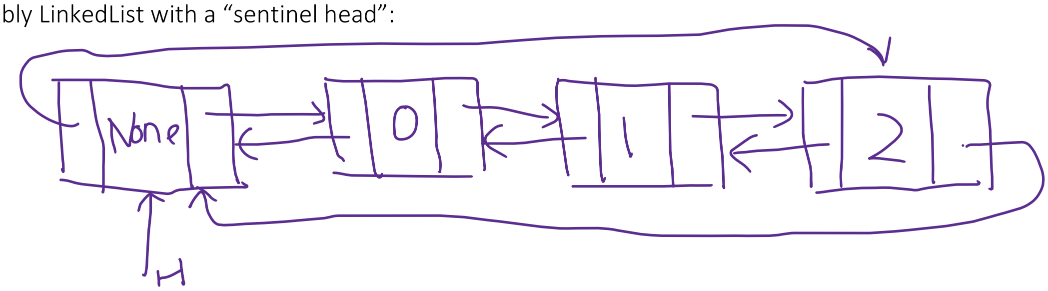
Doubly LinkedList

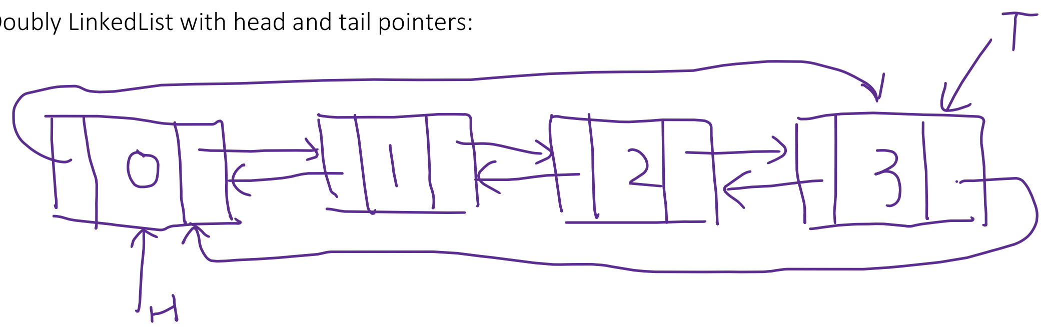
- Even with a pointer to the tail, we can see that it is still not easy to manipulate a LinkedList: it takes a long time to find the node that is previous to a given node.
- Each node in a Doubly LinkedList has two pointers: “prev” and “next”.



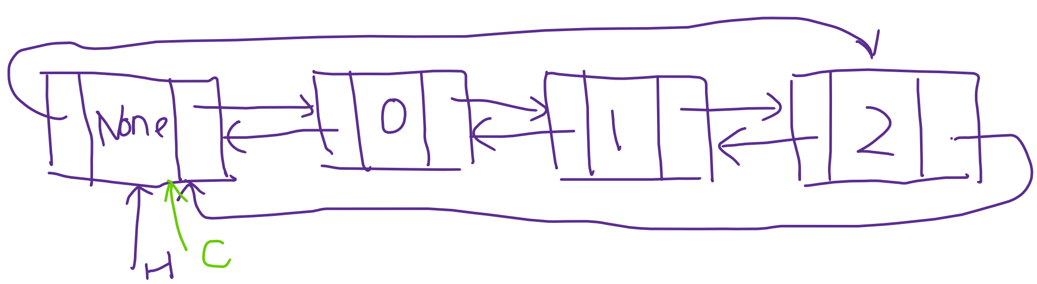
- There are several designs of a Doubly LinkedList:
  - Doubly LinkedList with a “sentinel head”:



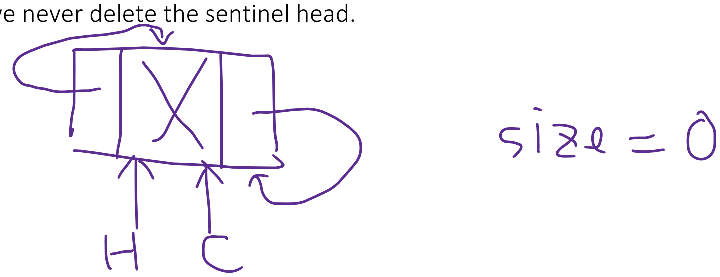
  - Doubly LinkedList with head and tail pointers:



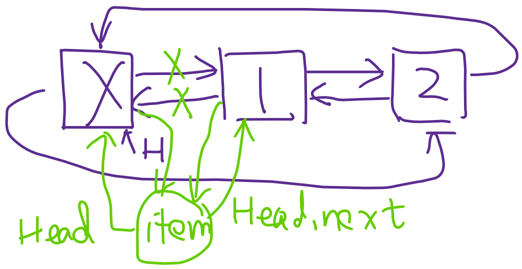
  - We will use the following design in our lecture. Doubly LinkedList with “sentinel head” and a cursor pointers:



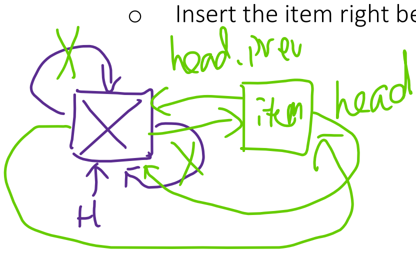
- __len__(), __repr__(), and __iter__() methods are basically the same as in a Singly LinkedList, be careful of the sentinel head.
- Construction method
  - Create a sentinel head. Set both “head” and “cursor” point to the sentinel head. Let attribute size equal to 0. Note that, we never delete the sentinel head.

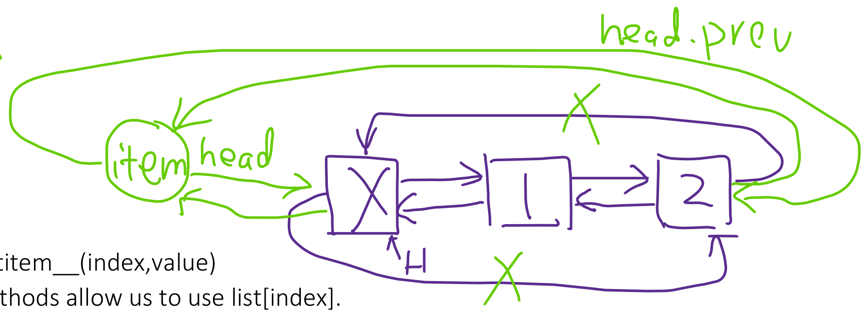


- prepend(item)
    - Insert the item right after the sentinel head. Increase size. Time complexity of this operation is $O(1)$.

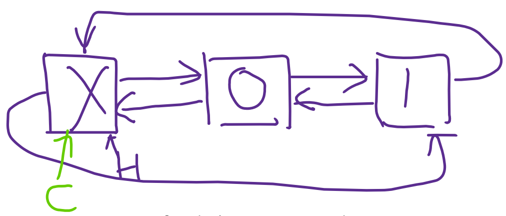


- append(item)
    - Insert the item right before the sentinel head. Increase size. Time complexity of this operation is $O(1)$.



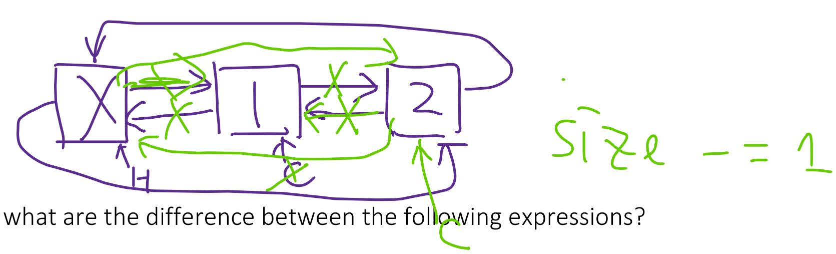


- __getitem__(index), __setitem__(index,value)
    - These two dunder methods allow us to use list[index].
    - Here, index is between 0 and len(list). To move to the given index, we move right $index + 1$ times from the sentinel head.



    - In our design, we use cursor pointer to find the given index. Here are some cursor functions that we need to implement:
        - cursor_set(index): let cursor move to the given index. This operation takes $O(n)$ time.
        - cursor_get(): return the item at cursor pointer. This operation takes $O(1)$ time.
        - cursor_update(value): update the item at cursor pointer to value. This operation takes $O(1)$ time.
    - With the help of above helper methods, __getitem__(index) can be implemented by cursor_set(index) + cursor_get(); __setitem__(index,value) can be implemented by cursor_set(index) + cursor_update(index, value).

- cursor_insert(value):
    - Insert the value after the cursor and set the new node as the cursor. This operation takes $O(1)$ time.
- cursor_delete():
    - Delete the node at cursor, then move cursor to the following node. Remind that, we never delete the sentinel head. This operation takes $O(1)$ time.



➢ As an aside, what are the difference between the following expressions?

1. This operation takes $O(n)$ time.
2. This operation takes $\Theta(n)$ time.
3. The worst-case time complexity of this operation is $\Theta(n)$.
   "operation" here actually means a set of simple operations

Number of operations	Expression 1	Expression 2	Expression 3
4	True	False	False
$[4, \lg n]$	True	False	False
$[2n, 100n]$	True	True	True
$[\lg n, 2n]$	True	False	True
$[4, n^2]$	False	False	False

- Search in a LinkedList
  - It is easy to see that we can use linear search in a LinkedList, and it takes $O(n)$ time.
  - If a LinkedList is sorted, can we use binary search to achieve $O(\lg n)$ running time?
    Here is the pseudo-code of recursive version of binary search in an array:

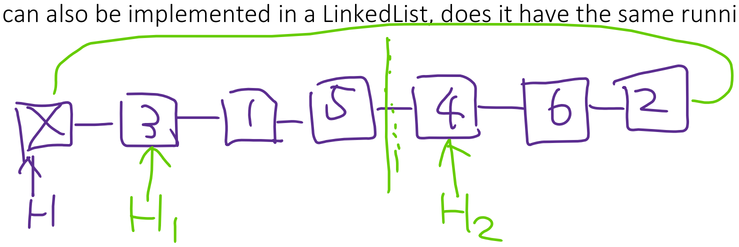
BINARY-SEARCH $(t, A[r \dots p])$

1 $low = r$
2 $high = p$
3 **if** $(low < high)$
4 $\quad mid = \lfloor (low + high)/2 \rfloor$
5 $\quad$ **if** $t == A[mid]$
6 $\quad\quad$ **return** mid
7 $\quad$ **else if** $t > A[mid]$
8 $\quad\quad$ **return BINARY-SEARCH** $(t, A[mid + 1 \dots p])$
9 $\quad$ **else** $\quad$ **return BINARY-SEARCH** $(t, A[r \dots mid - 1])$
10 **if** $t == A[high]$ $\quad$ **return** $high$
11 **else** $\quad$ **return** -1

As a review of Master Theorem, let's calculate its running time:
Let $T(n)$ be the worst-case time complexity to binary search in a subarray of length n, then $T(n) = T\left(\frac{n}{2}\right) + c$, here $a = 1, b = 2, f(n) = c$. Then $\frac{af\left(\frac{n}{b}\right)}{f(n)} = \frac{f\left(\frac{n}{2}\right)}{f(n)} = \frac{c}{c} = 1$, thus $T(n) = c \cdot \log_2 n$ which is $\Theta(\lg n)$.

In a LinkedList, line 5 cannot be achieved in $O(1)$; instead, it takes $\Theta(n)$ time. Thus, in a LinkedList, $T(n) = T\left(\frac{n}{2}\right) + \Theta(n)$ or $T(n) = T\left(\frac{n}{2}\right) + c \cdot n$. Here, $a = 1, b = 2, f(n) = cn$. Then $\frac{af\left(\frac{n}{b}\right)}{f(n)} = \frac{f\left(\frac{n}{2}\right)}{f(n)} = \frac{cn/2}{cn} < 1$, then $T(n) = cn$, which is $\Theta(n)$.

- Sort a LinkedList
  - All the $\Theta(n^2)$ sorting algorithms we have seen in an array also work well in a LinkedList.
  - A merge sort can also be implemented in a LinkedList, does it have the same running time $O(n \lg n)$ as in an array?



Here is the pseudo-code of merge sort in an array.

MERGE-SORT ($A[p \dots r]$)

if ($p < r$)

  $q = \left\lfloor \frac{p+r}{2} \right\rfloor$

  MERGE-SORT ($A[p \dots q]$)

  MERGE-SORT ($A[q+1 \dots r]$)

  MERGE ($A[p \dots q]$, $A[q+1 \dots r]$)

In a LinkedList, merging two sorted list takes $\Theta(n)$ time. We cannot split a list into two in constant time, we need to move a pointer to the index q first, and it takes $O(n)$ time. However, this $O(n)$ is not "larger than" the $\Theta(n)$ for merge, so a merge sort in a LinkedList still have time complexity $O(n \lg n)$.