VVKS

Changwen Yu

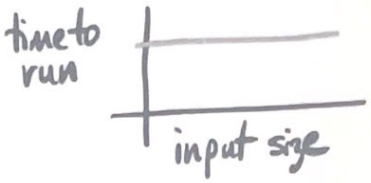
CSC148 - Running time efficiency: Lists, Stacks, and Queues

We have now seen that Python lists have the following running times for key operations:

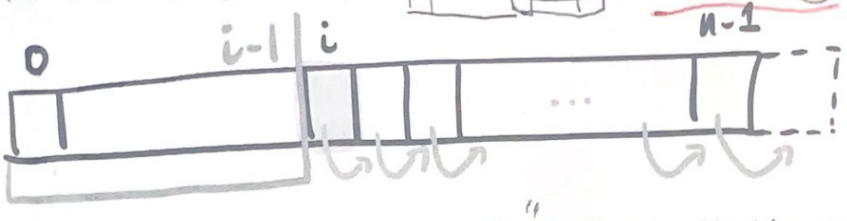
- Accessing or assigning to any element by index takes *constant time*.
- Inserting or removing an item at a given index takes time *proportional to the number of items after the index.*

1. Answer the following questions to make sure you understand the key concepts before moving on.

   (a) What do we mean by "constant time" above?

   (b) Suppose we have a list of length n. If we want to insert a new item at position i, how many list elements must be moved? (Be careful about off-by-one errors!)

   (c) Suppose we have a list of length n. If we want to remove the existing item at position i, how many list elements must be moved? Do not include the item being removed.

   (d) Suppose we have two lists: one of length 100, and one of length 1,000,000. Give an example of each of the following:

   (i) An operation that would be faster on the smaller list than the larger list.

   insert at front

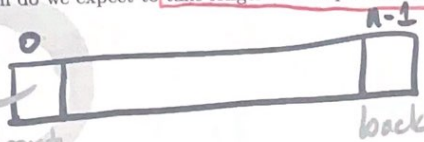
   Add an element in the middle of the list

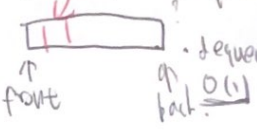
   (ii) An operation that would take roughly the same amount of time on the two lists.

   index, append

   Append to ~~last~~ end of the list.

   (e) Suppose we implement a Queue using a Python list, where the front of the list represents the front of the queue. Which operation do we expect to take longer as the queue size increases? `Queue.enqueue` or `Queue.dequeue`?

2. Now let's look at some code. Suppose we have two implementations of the Stack ADT: `Stack1` has `push` and `pop` operations that take 1 step, regardless of stack size, while `Stack2` has `push` and `pop` operations that take $n + 1$ steps, where n is the number of items currently on the stack. (You might argue it's "n steps", but that difference doesn't matter here.)

   For each of the code snippets on the next page, calculate the number of steps taken in total by all `push` and `pop` operations that are performed by the code. Do each calculation twice: once assuming we use the `Stack1` implementation, and once assuming we use the `Stack2` implementation. Ignore all other operations for this exercise—you're only counting steps for `push` and `pop` here.

(a)
```
# s starts as a stack of size n
s.push(1)
s.pop()
```
Stack gets bigger

Stack1 $1+1=2$.

Stack2 push: $(n+1)$ +

pop: $(n+1+1) = 2n+3$

(b)
```
# s starts as an empty stack
for i in range(5)
    s.push(i)
```

Stack1 $1*5 = 5$.

Stack2 $(n+1) + \cdots + (n+4+1)$

$= \frac{(2n+6)\times 5}{2} = 5n+15$.

(c)
```
# s starts as an empty stack, k is a positive integer.
# Calculate the number of steps in terms of k.
# Hint: 1 + 2 + 3 + ... + k = k * (k + 1) / 2
for i in range(k)
    s.push(i)
```

Stack1 $1*k = k$.

Stack2 $\frac{k(k+1)}{2}$

(d)
```
# s1 starts as a stack of size n, and s2 starts as an empty stack
while not s1.is_empty():
    s2.push(s1.pop())

while not s2.is_empty():
    s1.push(s2.pop())
```
push $+(n+1)$ $\quad \frac{n(n+1)}{2}$

S_2 $1+2+\cdots$ $=$

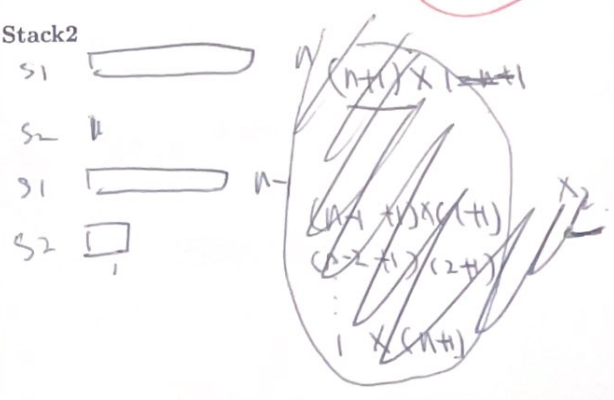
S_1 pop $(n+1+n+\cdots+2) = \frac{(n+3)n}{2}$

X_2

$= n^2+n+n^2+3n = 2n^2+4n$

Stack1 $2n+2n=4n$

Stack2



CSC148 - More practice with linked list traversals

Recall the basic Linked List traversal pattern:

```
curr = self._first          # Variable initialization
while curr is not None:     # Traversal loop
    ... curr.item ...
    curr = curr.next
```

l1 = LinkedList()
l2 = LinkedList()

On this worksheet, you'll work on developing two different methods that modify this basic pattern. In particular, both will involve changing the *while loop condition*, and you'll practice developing non-trivial loop conditions in a logical way.

1. Here is the docstring and implementation sections for the special method `LinkedList.__eq__`

special method

u v

```
def __eq__(self, other: LinkedList) -> bool:
    """Return whether this list and the other list are equal.

    Two lists are equal when each one has the same number of items,
    and each corresponding pair of items are equal (using == to compare).
    """
    # (1) Variable initialization

    # (2) Traversal loop

    # (3) Post-loop code
```

if l1 == l2
automatically call eq method

Our goal is to implement this method. Obviously, using just one variable `curr` is not enough to iterate through both lists. Instead, use two variables: `curr1` and `curr2`, one for each list. Show how to initialize these variables in the space below:

curr

```
# (1) Variable initialization
curr1 = self._first
curr2 = other._first
```

first

curr2

Next, let's work on **(2)**, the traversal loop. The loop condition is a bit subtle. To make sure we get this right, we're going to go slow.

先想什么时候停？有个比较关键。

(a) First, let's think about when we want the loop to *stop*. This should be when we reach the end of `self` or the end of `other`. Write down a Python expression involving `curr1` and `curr2` that expresses this *stopping condition*. (By "expresses", we mean your expression should evaluate to `True` when the condition is true, and `False` otherwise.)

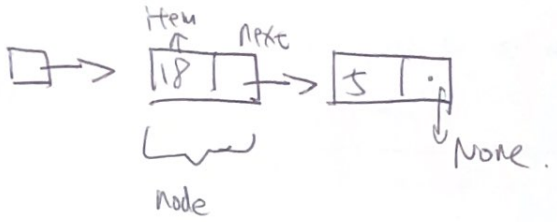
(curr1 is None) ~~and~~ (curr2 is None) or both

or

(b) The while loop condition should always be the *negation* of the stopping condition. Write down a Python expression for the while loop condition.

not [(curr1 is None) or (curr2 is None)]

(curr1 is not None) and (curr2 is not None)



Code summary

```
class _Node:
    item: Any
    next: Optional[_Node]

class LinkedList:
    _first: Optional[_Node]
```

private class

Using this, write down the while loop you would use to traverse the two lists. At each iteration, you should compare `curr1.item` against `curr2.item`; if this is False, you should be able to stop and return, without checking any other values!

```
# (2) Traversal loop
```

while (not curr1) and (not curr2):

While (curr1 is not None) and (curr2 is not None)
if curr1.item = curr2.item:
curr1 = curr1.next
curr2 = curr2.next
else: return False.

都是 both are None
return curr1 == curr2.

return (curr1 is None) and (curr2 is None)

Finally, suppose we reach the end of the loop. We need some code to handle this case as well.

(a) What do we know about `curr1` and/or `curr2` after the loop ends? (Hint: look up at your stopping condition.)

(b) How can we use the values of `curr1` and `curr2` to check whether the lists have the same length?

(c) Write the code that should go after the end of our loop. Remember that it should return `True` or `False`.

```
# (3) Post-loop code
```

2. Now, we're going to repeat the same process for the special method `__getitem__`, which enables indexing using square brackets: `lst[i]` is equivalent to `lst.__getitem__(i)` in Python!

magic method

```
def __getitem__(self, index: int) -> Any:
    """Return the item at position <index> in this list.

    Raise an IndexError if the <index> is out of bounds.

    Precondition: index >= 0.
    """
```

For this implementation, you'll need to use two variables: `curr` for the current node in the list, and `i` for the current index in the list. Use the following strategy to implement `__getitem__` on a separate sheet of paper.

(a) Write down how to initialize `curr` and `i`.

(b) Write down the stopping condition for the loop, and then the while loop condition (negate the stopping condition).

(c) Implement the loop body.

(d) After the loop ends, use the stopping condition to remind yourself what you know about the values of `curr` and `i`. Use this to implement the post-loop code.

```
curr1 =
curr2 = 2
while (curr1 is not None) and (curr2      ):
    if curr1.item ==      :
                    .next
                    .next
    else:
        return False
assert (curr1 is None) or (curr2 is None)
```

思路 ① 想什么时候停

② 停的 negation 是 while loop condition

③ 想好，一步一步分析，之后再写

2. ~~self.first~~ ~~current item~~ try to find ith item in linked list.

```
curr = self.first ← current item
i = 0 ← cur index
# iterate to reach ith node, stop when curr. ~~next~~ is None or
                                              i == index ← 到了.
while curr ~~or~~ and (i != index):
    (curr is not None)
    i +=
    curr = curr.next.
# (curr is None) or (i == index) or both
~~return i == index~~
~~if i == index:~~
if curr is None:
    raise IndexError:
else: 当cur不是None, i must be index
    return curr.item
           ↓        (. dot之前的)
永远要考虑 curr是不是有可能是 None
```

CSC148 - Linked List Insertion

Our goal for this worksheet is to extend our `LinkedList` class by implementing one of the standard mutating List ADT methods: inserting into a list by index. Here's the docstring of such a method:

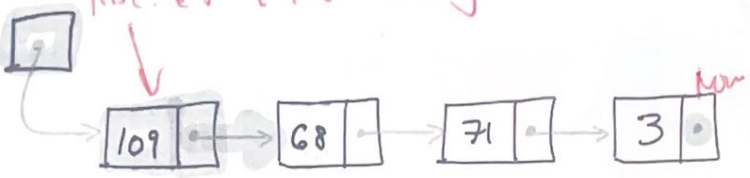
```
def insert(self, index: int, item: Any) -> None:
    """Insert a the given item at the given index in this list.

    Raise IndexError if index > len(self) or index < 0.
    Note that adding to the end of the list is okay.
    """
```

1. Before diving into any code at all, we'll gain some useful intuition by generating some test cases for this method based on two key input properties: the length of the list, and the relationship between index and the length of the list. We *won't* care about what item we're inserting (since it could be anything).

   In the table below, we've described some inputs based on these properties.

   (a) For each row of the table, draw a linked list of the specified length, using the abstract diagram style shown below. The leftmost box represents the `LinkedList` object itself, with its `_first` attribute referring the the first node in the linked list.



   To represent an empty list, we draw a box with a single dot in it (its `_first` attribute is `None`).

   (b) For each row of the table, now show what happens if you insert the number 148 into the list at the given index. You can edit your existing diagram or draw a brand-new one.

Input description	Linked list diagram
`len(self) == 0, index == 0`	
`len(self) == 1, index == 0`	
`len(self) == 1, index == 1`	
`len(self) == 4, index == 0`	
`len(self) == 4, index == 2`	
`len(self) == 4, index == 4`	

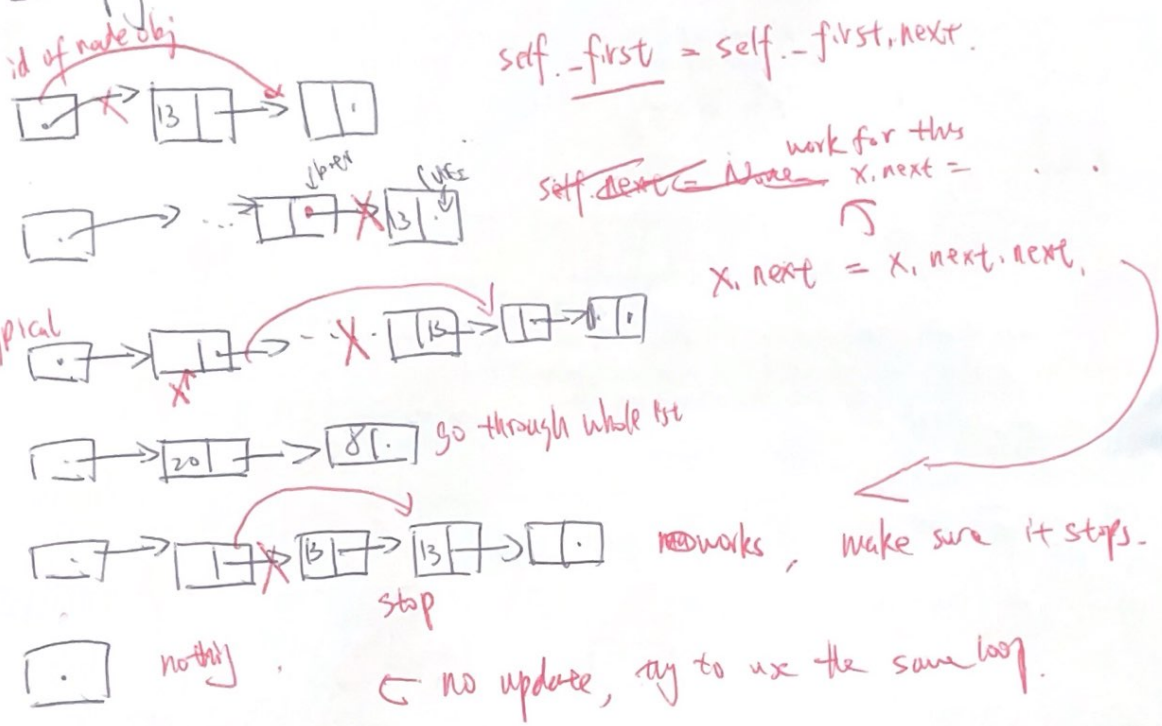
Remove From LinkedList

parameters: self, item (13)

Scenarios

(1) At front
At end
"middle"
not in list
in list more than once

LL empty



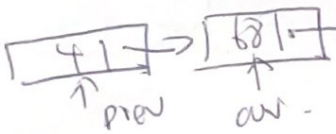
先从 common case 想起

Note: we need to change the Node befor the Node 13.

2 sol

1) look ahead & stop if following node has 13.
if curr.next == item

2) keep 2 variable: curr + prev



prev, curr = curr, curr.next
~~curr, prev~~

we are inserting at end. change (except inserting at front) is at index - 1

2. Using your diagrams as a guide, answer the following questions:

   (a) For what values of `len(self)` and/or `index` would we need to re-assign `self._first` to something new?

   index = 0

   (b) What is the relationship between `len(self)` and `index` that makes `insert` behave the same as `LinkedList.append` from this week's prep? ==

   (c) In the `len(self) == 4, index == 2`, which *existing* node was actually mutated? Write down the index of this node in the list; hint, it's not the one at index 2!

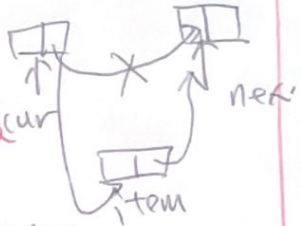
   index - 1 index 之前的 node

3. Finally, using these ideas, implement the `insert` method in the space below. Note that you should have two cases: one for when you need to mutate `self._first`, and one where you don't. Also, you'll want to use the same approach as `LinkedList.__getitem__` and keep two parallel variables, `curr` and `i`.

```
curr = self._first.
i = 0.                          new_node = _Node(item)
# loop to right spot
if index < 0 :
    raise IndexError !          if index == 0:
                                    self._first = new_node.
                                    new_node.next = curr.
curr.
# loop - 找到 index -1 (curr在i前面).
while (i != index-1) and (cur.next is not None):
    cur = cur.next
    i += 1.

# 找到 i == index -1 or cur.next is None.
if cur.next is None:
    raise IndexError.
else:
    next = cur.next
    cur.next = item.          new_node
    item.next = next / cur.next.next = next
    new_node
```

Stop condition
i == index - 1
curr is None.



Code

```
# initize prev & cur.
curr = self._first
prev = None

# stopping condition: found item / curr is None
# continue condition: (curr.item != item) and (curr
                    -> (curr is not None) and (curr.item != item)

while (curr is not None) and (curr.item != item):
    prev = curr
    curr = curr.next

# decide if we found item
if curr is not None:
    # we found item (curr.item == item)
    # determine whether we are at front.
    if prev == None:
        self._first = self._first.next
    else:
        prev.next = prev.next.next
```

```
def eq_v2(self, other: LinkedList) -> bool:
    # use a while loop that stops loop
      if 2 cur node aren't equivalent.

    cur1 = self._first
    cur2 = other._first
    # whileloop stop condition: reaches End (cur1 is None or cur2 is None)
    # continue 是反过来
    while (cur1 is not None) and (cur2 is not None):
        if cur1.item != cur2.item:
            return False
    # 当结束有一个 or 二个是 None, 二个是 None return True
    return (not cur1) and (not cur2)
```

```
def pop(self, index: int) -> Any:
    """remove & return the item at position<index>. 改变 index 有的
    index<0 or index>len(self), pop index Error.
    """
    prev = None
    cur = self._first
    while position != index and (cur is not None):
        prev = cur
        cur = cur.next
        position += 1
    # 在头:
    if prev is None:
        self._first = self._first.next
    # 在中:
    elif position == index:
        prev.next = cur.next
    # 在尾:
    else:
        raise IndexError
```