Lecture 2:
Arrays & Linked Lists

김수경

이화여자대학교 인공지능융합전공 소속

저작권 안내

(주)업스테이지가 제공하는 모든 교육 콘텐츠의 지식재산권은 운영 주체인 (주)업스테이지 또는 해당 저작물의 적법한 관리자에게 귀속되어 있습니다.

콘텐츠 일부 또는 전부를 복사, 복제, 판매, 재판매 공개, 공유 등을 할 수 없습니다.
유출될 경우 지식재산권 침해에 대한 책임을 부담할 수 있습니다.

유출에 해당하여 금지되는 행위의 예시는 다음과 같습니다.

- 콘텐츠를 재가공하여 온/오프라인으로 공개하는 행위
- 콘텐츠의 일부 또는 전부를 이용하여 인쇄물을 만드는 행위
- 콘텐츠의 전부 또는 일부를 녹취 또는 녹화하거나 녹취록을 작성하는 행위
- 콘텐츠의 전부 또는 일부를 스크린 캡쳐하거나 카메라로 촬영하는 행위
- 지인을 포함한 제3자에게 콘텐츠의 일부 또는 전부를 공유하는 행위
- 다른 정보와 결합하여 Upstage Education의 콘텐츠임을 알아볼 수 있는 저작물을 작성, 공개하는 행위
- 제공된 데이터의 일부 혹은 전부를 Upstage Education 프로젝트/실습 수행 이외의 목적으로 사용하는 행위

01

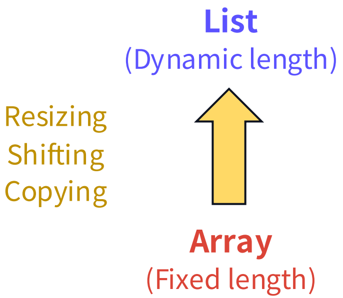
Arrays

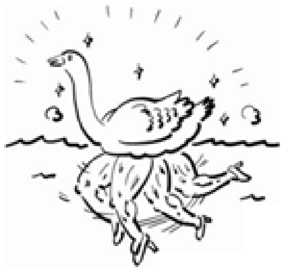
upstage Education

Arrays

- An **array** is an object comprising a numbered sequence of memory boxes
  - More fundamental data structure that Python lists are built on.
  - This is why we can easily access the *i*-th element of list A by using A[i].

- An array comprises
  - **Fixed** integer **length** (*N*) – should be set when initializing it
  - A sequence of *N* memory boxes (numbered 0 through *N* - 1)





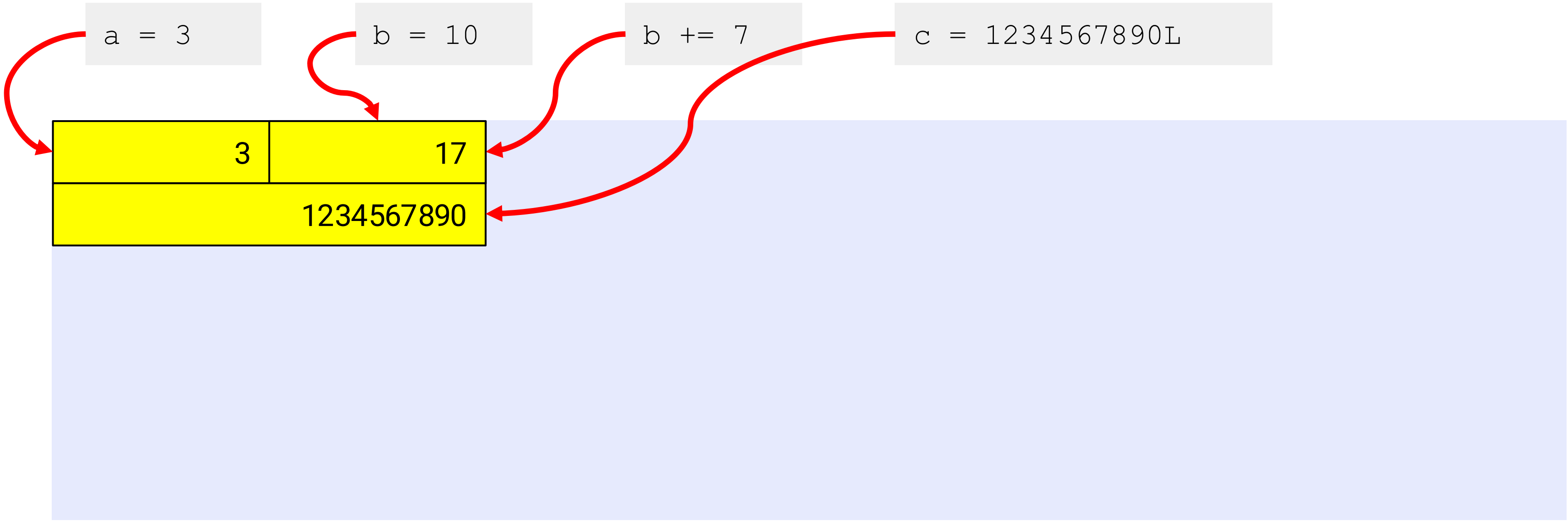
[1]

[1] http://1moment.tistory.com

upstage Education

Internal Implementation: Memory

- Internally, all variables and constants we use in our program should be stored somewhere in **memory**.
  - For a single variable of a primitive type (int, float, ⋯), we know its size (how many bits are needed).



upstage Education

Array Resizing

- Two problems of an array due to its fixed length
  - **Memory wastage**: if it contains only $n \ll N$ valid elements
  - **Memory shortage**: if it wants to contain more than N elements
- Array resizing: create another larger array and copy all the elements
  - L.append(3) when the current array is full.

L

Index	0	1	2	3	4	5	6	7	8	9
Value	-7	15	2	6	-1	5	4	10	-4	21

Index	0	1	2	3	4	5	6	7	8	9	10
Value	-7	15	2	6	-1	5	4	10	-4	21	3

Create
a new long array O(1)

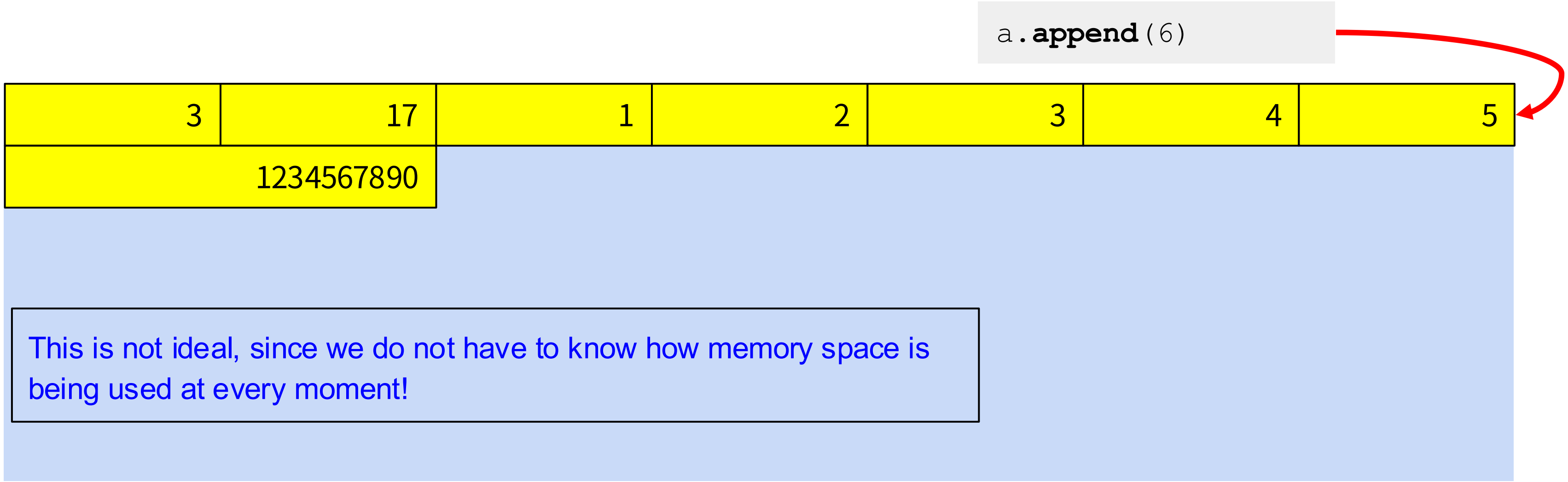
Create(copy)
a new long array **$O(N)$**

Add
a new element O(1)

upstage Education

Internal Implementation: Memory

- Now, let's think about the time complexity!
  - **Theoretical time complexity** for `a.append(6)`? O(1)
  - **Actual time complexity** for `a.append(6)`, if there's no enough space after it? O(*N*)



upstage Education

Array Resizing

- Array resizing is expensive: new memory boxes and copy operation
  - Increasing size by one every time is not efficient (too many resizing)
  - Increasing size too much at once is not efficient either (memory wastage)

- To resize fewer, Python list size grows as 0, 4, 8, 16, 25, 35, 46, 58, ···
  - Mild over-allocation proportional to the current size

- Anyway, is there any better way of organizing a collection of data to support append and pop easily?

02

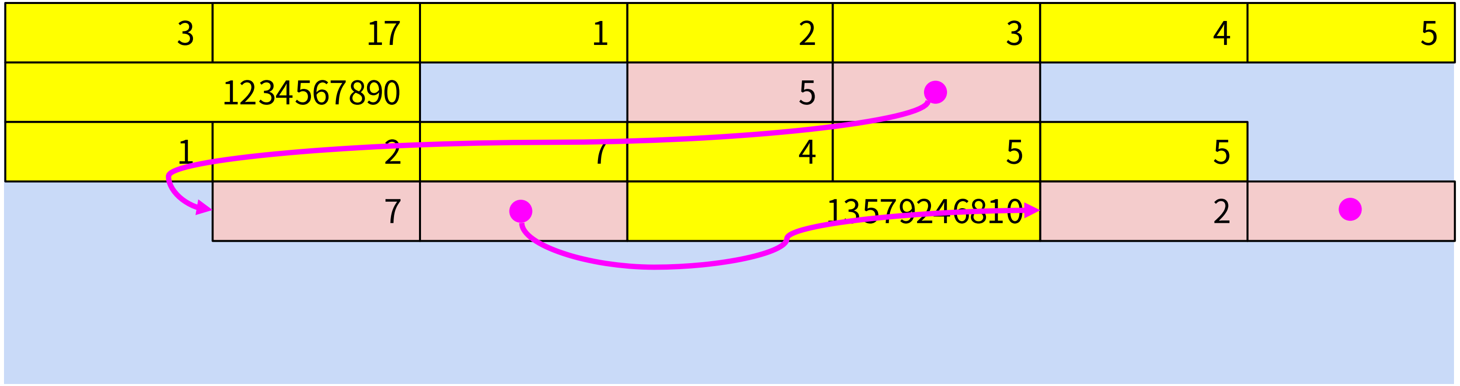
Linked Lists

upstage Education

Linked Lists

- Main idea:
  - Allow each element in the list to be **scattered in the memory**.
  - Instead, each element **points to the next one**.

l.**append**(5)　　l.**append**(7)　　l.**append**(2)



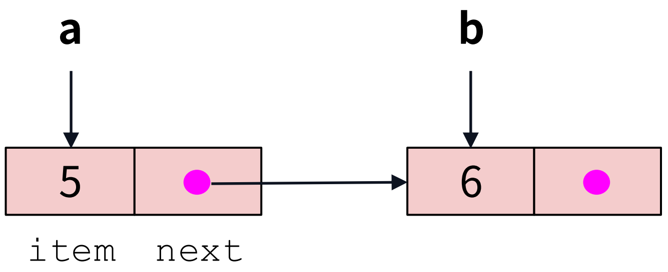
upstage Education

Linked Lists

- Class Node
  - Because we always need to store the **item** and the pointer to the **next** node, let's make this as a class!

```
class Node():
  def __init__(self, x):
    self.item = x
    self.next = None
```

```
a = Node(5)
b = Node(6)
a.next = b
```




```
print(a.item)
print(a.next.item)
```

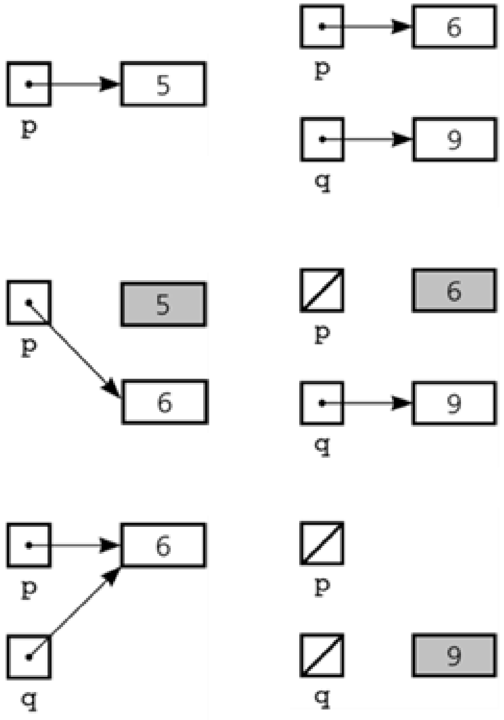
5

6

upstage Education

Review: Python Object Reference

	p.item	q.item
`p = Node(5)`	5	
`p = Node(6)`	6	
`q = p`	6	6
`q = Node(9)`	6	9
`p = None`	Error!	9
`q = p`	Error!	Error!

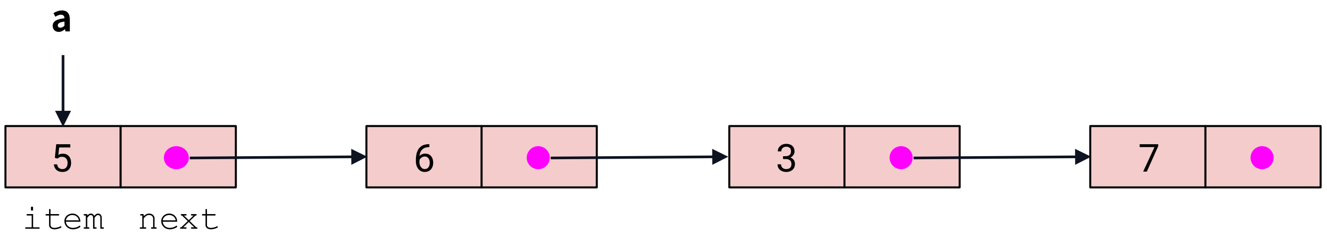


upstage Education

Singly Linked List

- Let's design the **singly linked list** data structure. What functionalities do we need?
  - **Creating** an empty list
  - **Adding / inserting** a new item
  - **Retrieving** an item
  - **Deleting / removing** an existing item

at position *i*

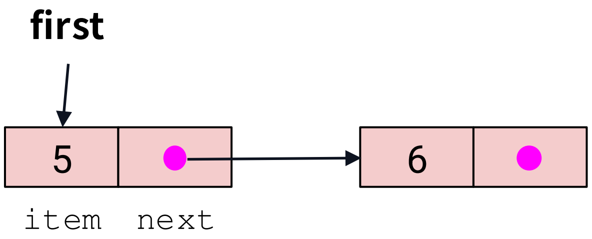


upstage Education

Singly Linked List

- Class `LinkedList`
  - We keep only the reference to the **first node**.
  - At creation, first node is `None`, having no element in the list.

```
class Node():
  def __init__(self, x):
    self.item = x
    self.next = None
```



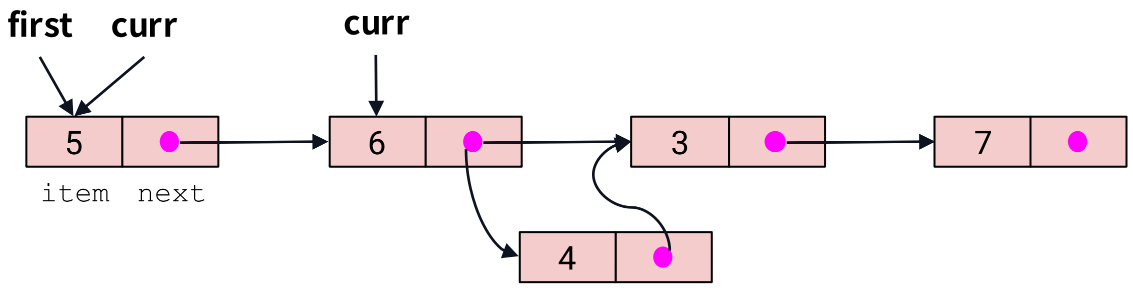

```
class LinkedList():
  def __init__(self):
    self.first = None

  def insert(self, x, i):
    # insert x at [i]

  def get(self, i):
    # get item at [i]

  def delete(self, i):
    # delete item at [i]
```

upstage Education

Inserting an Item at position *i*

- Step 1: Creating a new node with the given item.
- Step 2: Traverse to the (*i* - 1)-th position.
- Step 3: Set the new node's next as the original *i*-th node.
- Step 4: Update the (*i* - 1)-th node's next as the new node.

Example:
insert "4" at position 2.

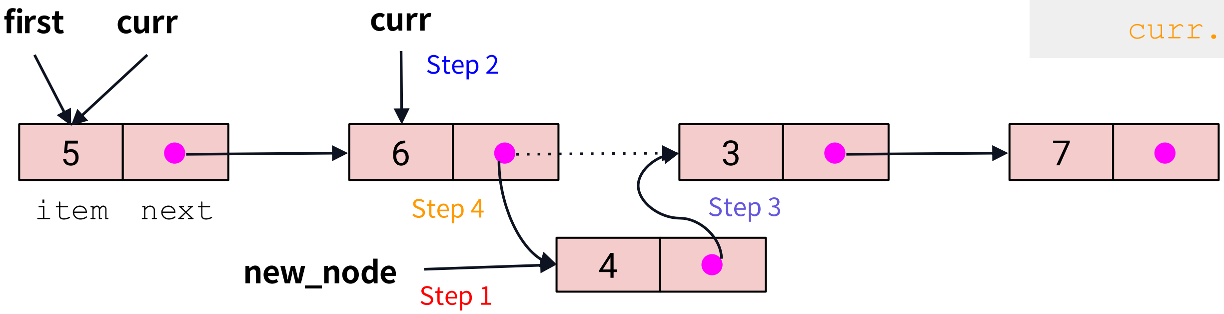


upstage Education

Inserting an Item at position *i*

- Step 1: Creating a new node with the given item.
- Step 2: Traverse to the (*i* - 1)-th position.
- Step 3: Set the new node's next as the original *i*-th node.
- Step 4: Update the (*i* - 1)-th node's next as the new node.

```
class Node():
  def __init__(self, x):
    self.item = x
    self.next = None
```



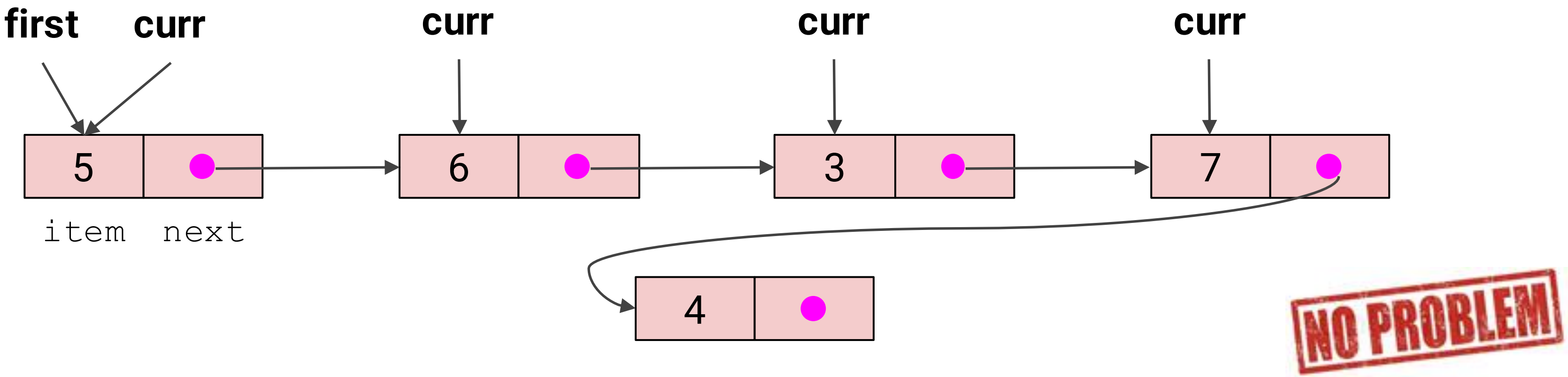

```
def insert(self, x, i):
  # insert x at [i]
  new_node = Node(x)

  pos = 0
  curr = self.first
  while pos < i - 1:
    curr = curr.next
    pos += 1

  new_node.next = curr.next

  curr.next = new_node
```

upstage Education

Does it work at the end?

- Step 1: Creating a new node with the given item.
- Step 2: Traverse to the (i - 1)-th position.
- Step 3: Set the new node's next as the original i-th node.
- Step 4: Update the (i - 1)-th node's next as the new node.

Example:
insert "4" at position **4.**



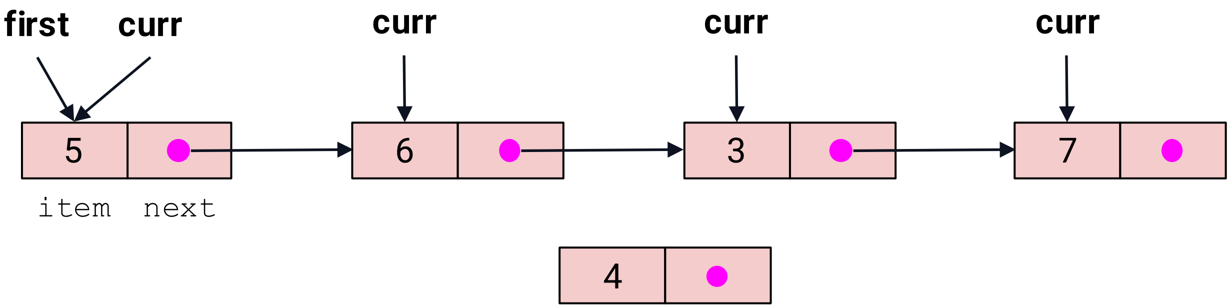
upstage Education

Does this work at the beginning?

- Step 1: Creating a new node with the given item.
- Step 2: Traverse to the (*i* - 1)-th position.
- We need special treatment when we insert at position 0!

Example:
insert “4” at position **0**.

i - 1 = -1 😨



upstage Education

Inserting an Item at position *i*

- If inserting at the first position:
  - Step 1: Creating a new node with the given item.
  - Step 2: Set the new node's next as the original **first node**.
  - Step 3: **Update the first node** reference to the new node.
- else:
  - Step 1: Creating a new node with the given item.
  - Step 2: Traverse to the (*i* - 1)-th position.
  - Step 3: Set the new node's next as the original *i*-th node.
  - Step 4: Update the (*i* - 1)-th node's next as the new node.

Check: does this work when we insert the very first item (that is, does it work when `self.first = None`)?

```
def insert(self, x, i):
  # insert x at [i]
  if i == 0:
    new_node = Node(x)
    new_node.next = self.first
    self.first = new_node
  else:
    new_node = Node(x)

    pos = 0
    curr = self.first
    while pos < i - 1:
      curr = curr.next
      pos += 1

    new_node.next = curr.next

    curr.next = new_node
```

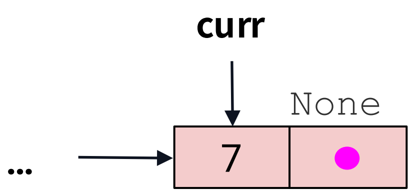
upstage Education

Inserting an Item at position *i*

Check: what happens with our code if *i* > last position?

It will crash here,

when it tries to access `None.next`




```
def insert(self, x, i):
  # insert x at [i]
  if i == 0:
    new_node = Node(x)
    new_node.next = self.first
    self.first = new_node
  else:
    new_node = Node(x)

    pos = 0
    curr = self.first
    while pos < i - 1:
      curr = curr.next
      pos += 1

    new_node.next = curr.next

    curr.next = new_node
```

We should prevent this, instead of letting the users to be responsible!

upstage Education

Size Variable

- First try:
  - Let's add a check at the beginning, if *i* is within the valid range.
  - Valid range?

    From 0 to current length (item count)

- But, how do we know the size?
  - A naive way: traverse from the `first` until we meet `None`.
  - This is not efficient, since we need to traverse *N* items whenever we insert a new item, regardless of the target position. 😨
  - Any better way?

```
  def insert(self, x, i):
    # insert x at [i]
    if i > size: return
    if i == 0:
      new_node = Node(x)
      new_node.next =
self.first
      self.first = new_node
    else:
      new_node = Node(x)
    elif i <= size:
      pos = 0
      curr = self.first
      while pos < i - 1:
        curr = curr.next
        pos += 1

      new_node.next = curr.next
      curr.next = new_node
```

upstage Education

Size Variable

- Solution: Let's keep the `size` variable in the class, and maintain it whenever we insert or delete an element.
- Time complexity?

O(1)

```
class LinkedList():
  def __init__(self):
    self.first = None
    self.size = 0

  def insert(self, x, i):
    # insert x at [i]

  def get(self, i):
    # get item at [i]

  def delete(self, i):
    # delete item at [i]
```

```
def insert(self, x, i):
  # insert x at [i]
  if i == 0:
    new_node = Node(x)
    new_node.next = self.first
    self.first = new_node
    self.size += 1
  elif i <= self.size:
    new_node = Node(x)

    pos = 0
    curr = self.first
    while pos < i - 1:
      curr = curr.next
      pos += 1

    new_node.next = curr.next
    curr.next = new_node
    self.size += 1
```

upstage Education

Retrieving an Item at position *i – Homework*

- Basic logic
  - Step 1: Traverse to the *i*-th position.
  - Step 2: Return the item in the node.
- Any special cases to consider?
  - Check if your implementation works when
    - `i = 0`
    - `i > self.size`
    - `self.size = 0`
    - …

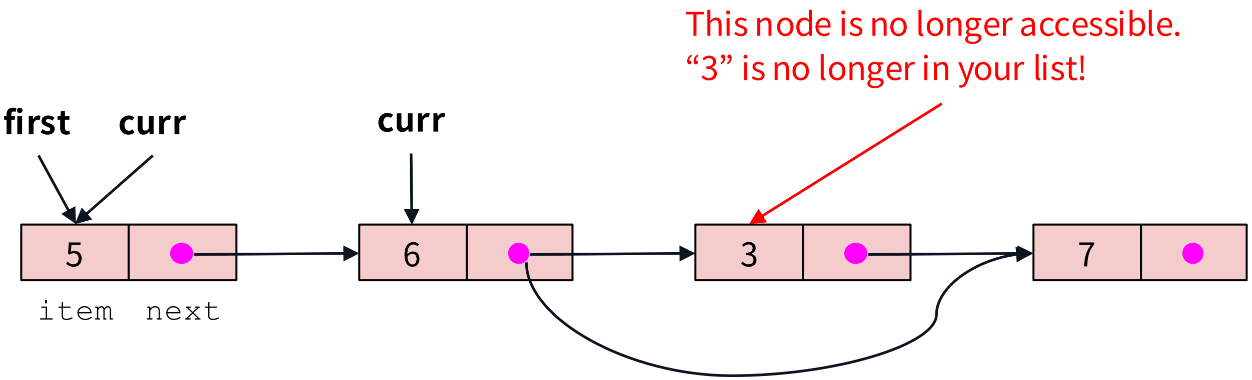
```
def get(self, i):
  # get item at [i]

  # TODO(students): implement!

  return ?
```

upstage Education

Deleting an Item at position *i*

- Step 1: Traverse to the (*i* - 1)-th position.
- Step 2: Set the (*i* - 1)-th node's next as the target's next.

Example:
delete item at position 2.



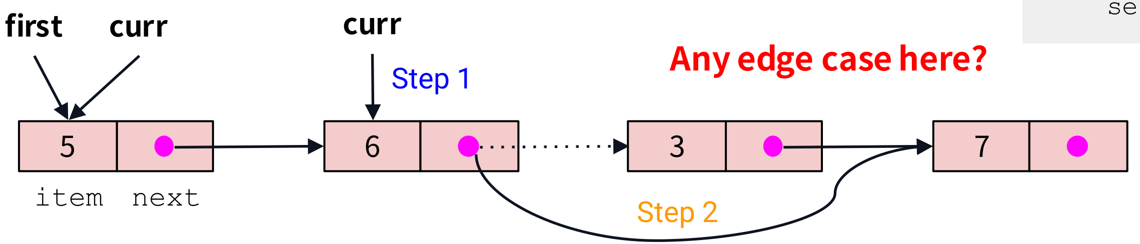
upstage Education

Deleting an Item at position *i*

- Step 1: Traverse to the (*i* - 1)-th position.
- Step 2: Set the (*i* - 1)-th node's next as the target's next.

```
class Node():
  def __init__(self, x):
    self.item = x
    self.next = None
```

```
def delete(self, i):
  # delete item at [i]
  pos = 0
  curr = self.first
  while pos < i - 1:
    curr = curr.next
    pos += 1

  curr.next = curr.next.next

  self.size -= 1
```



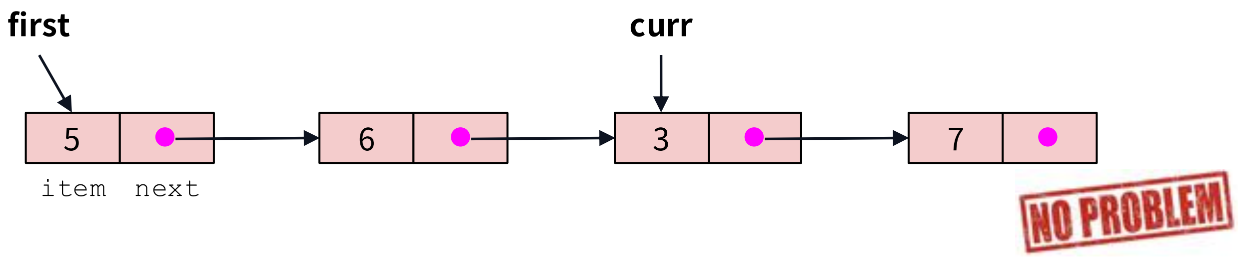
Any edge case here?

upstage Education

Does this work at the end?

- Step 1: Traverse to the (i - 1)-th position.
- Step 2: Set the (i - 1)-th node's next as the target's next.

Example:
delete item at position **3**.



upstage Education

Does this work at the beginning?

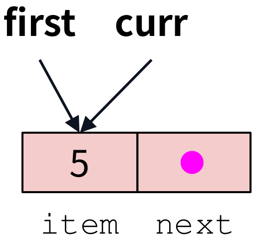
- Step 1: Traverse to the (*i* - 1)-th position.
- Step 2: Set the (*i* - 1)-th node's next as the target's next.

Example:
delete item at position **0**.

Again, we need special treatment when we delete the first one!

```
if i == 0:
  self.first = self.first.next
else:
  ...
```

i - 1 = -1



This condition is never satisfied.

```
def delete(self, i):
  # delete item at [i]
  pos = 0
  curr = self.first
  while pos < i - 1:
    curr = curr.next
    pos += 1

  curr.next = curr.next.next
  self.size -= 1
```

upstage Education

Time Complexity

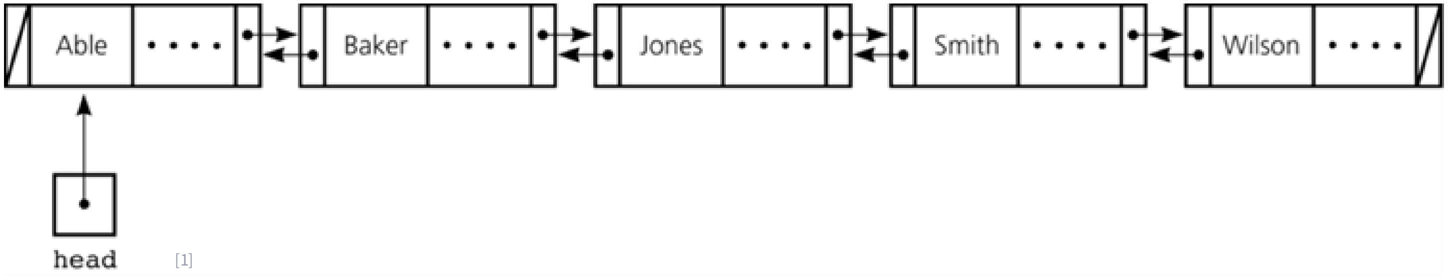
- Time complexity of Linked List?

Task	Worst case	Average case	Best case
Insertion	$O(N)$	$O(N)$	$O(1)$
Retrieval	$O(N)$	$O(N)$	$O(1)$
Deletion	$O(N)$	$O(N)$	$O(1)$

Happen when?

upstage Education

Doubly Linked List



[1]

- Sometimes it is useful to have ability to access *previous* items.
- No asymptotic benefit on complexity.

[1] https://maryash.github.io/235/projects/project_3/project_3.html

Comparison

- Array
  - **Consecutive** memory space is assigned.
  - Fixed length
  - **Random access** is supported in **O(1)**.
  - Suffers from **item shifting**.

- Linked list
  - **Scattered** in the memory space.
  - Additional space is needed for storing the next reference.
  - **No random access** allowed. (Linear traversal is required, taking $O(N)$.)
  - **No shifting** is needed even with size change.

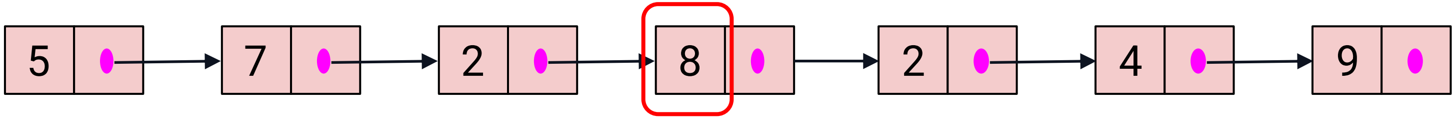
03

Applications of Linked Lists

upstage Education

Application Questions

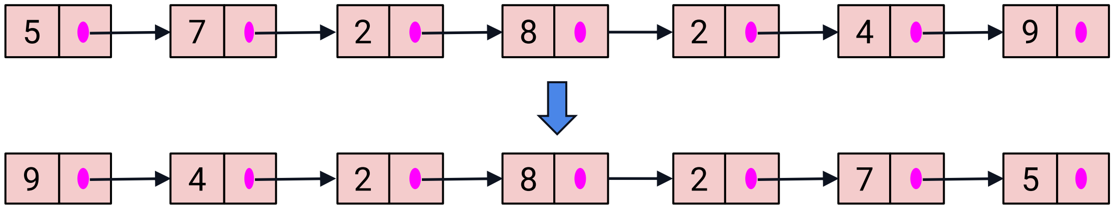
- Print the middle of a given linked list. (Assume list size is not maintained.)
- Brute-force solution
  - Traverse the entire list to count the number of elements.
  - Traverse half of them again.
  - $\rightarrow O(N)$



upstage Education

Application Questions

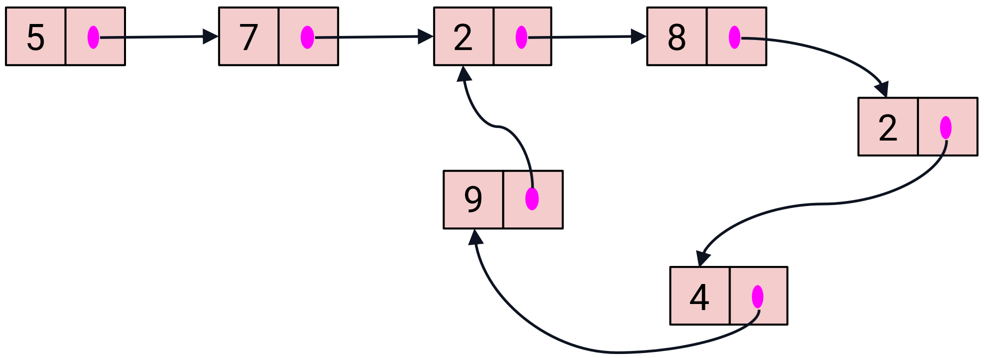
- Reverse a given linked list.



upstage Education

Application Questions

- Detect if there is a cycle in the given linked list.



Building intelligence for the future of work