Lecture 3: Stacks & Queues

김수경

이화여자대학교 인공지능융합전공 소속

저작권 안내

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

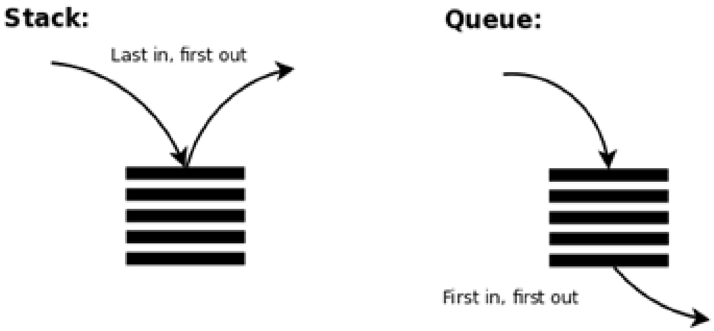
콘텐츠 일부 또는 전부를 복사, 복제, 판매, 재판매 공개, 공유 등을 할 수 없습니다.
유출될 경우 지식재산권 침해에 대한 책임을 부담할 수 있습니다.

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

Today: Stacks & Queues

- Stack
  - Last In, First out (**LIFO**)
  - Access only to the most-recently added item.
- Queue
  - First In, First out (**FIFO**)
  - Access only to the item that was added earliest.



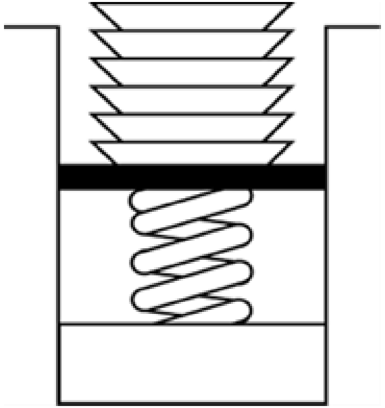
[1]

[1] https://gohighbrow.com/

01

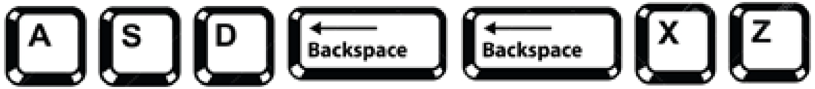
Stacks

Stack Examples



Stack of cafeteria dishes

[1]

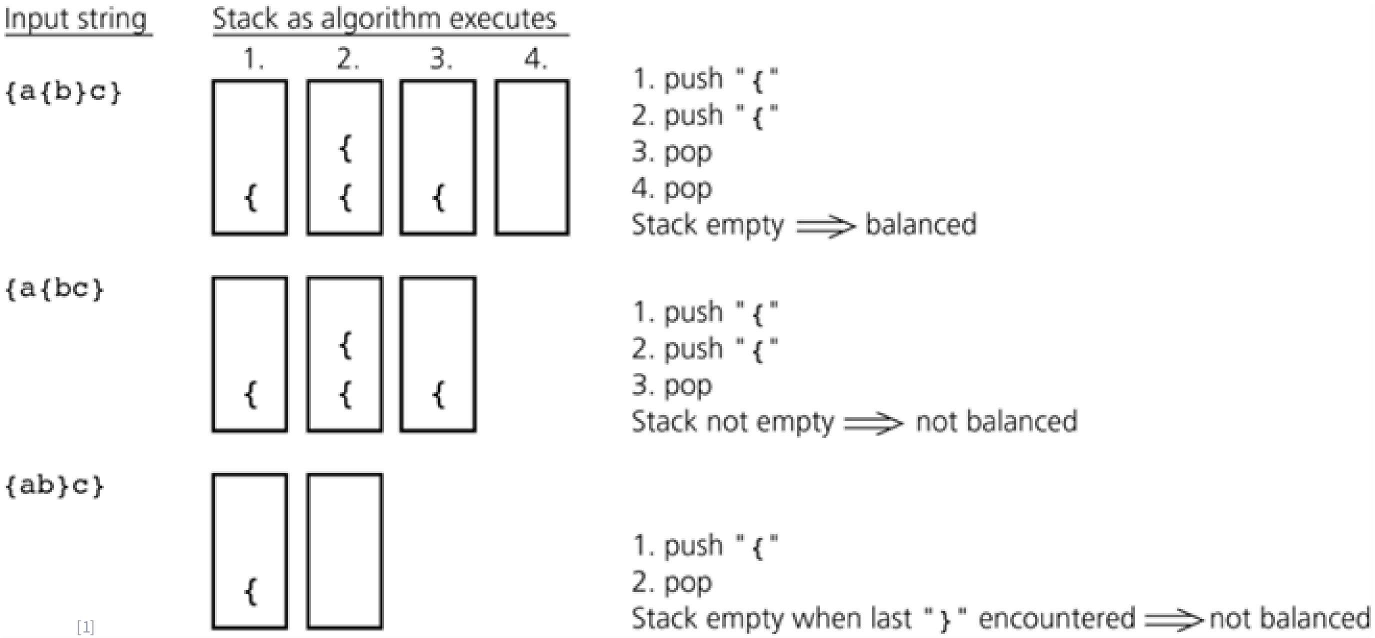


Backspacing with keyboard

[2]

[1] https://www.cs.vassar.edu/~cs125/lectures/lect9-Stacks/ch07.pdf
[2] https://kr.123rf.com/photo_15834863_키보드-키-버튼의-아이콘을-설정-문자-백-스페이스-삭제.html

Stack Example: Checking Balances of Braces



{a{b}c}

1. push "{"
2. push "{"
3. pop
4. pop
Stack empty ⟹ balanced

{a{bc}

1. push "{"
2. push "{"
3. pop
Stack not empty ⟹ not balanced

{ab}c}

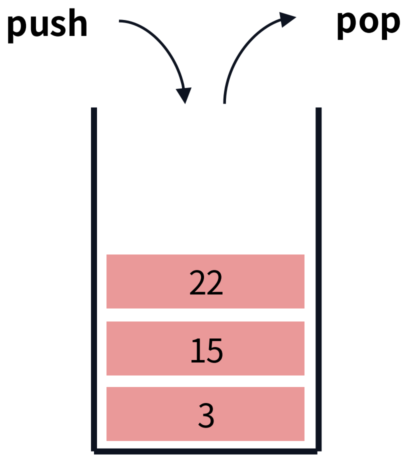
1. push "{"
2. pop
Stack empty when last "}" encountered ⟹ not balanced

[1]

[1] https://www.slideshare.net/slideshow/stack-10033613/10033613

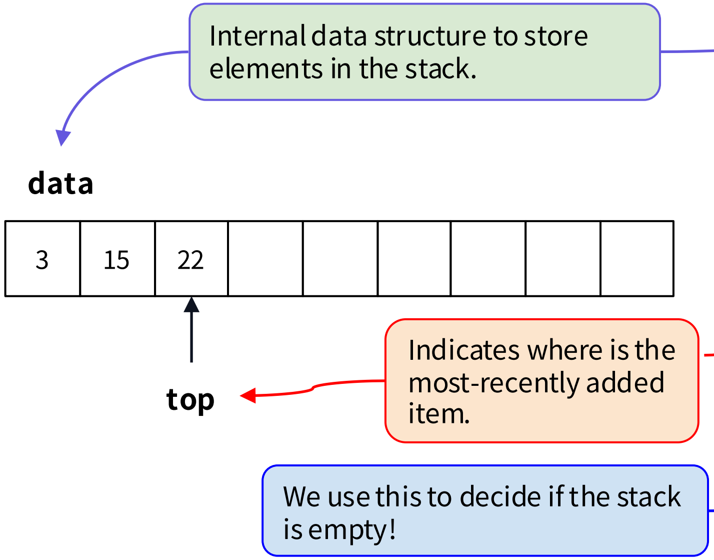
Stack Terminologies

- What functionality do we want to have with Class Stack?
  - Adding a new element (**push**)
  - Retrieving the most recent item (**peek**)
  - Deleting an item (**pop**)



Stack Class Design

- Array-based Implementation
  - We use Python `List` for simplicity here.



Internal data structure to store elements in the stack.

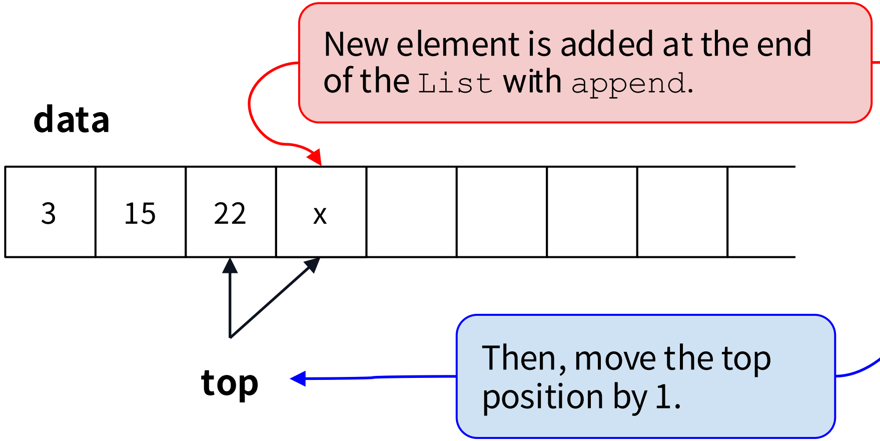
Indicates where is the most-recently added item.

We use this to decide if the stack is empty!

```
class Stack():
  def __init__(self):
    self.data = []
    self.top = -1

  def push(self, x):
    # insert x

  def peek(self):
    # get item

  def pop(self):
    # delete an item

  def is_empty(self):
    return (self.top == -1)
```

Stack Class Implementation

- Push
  - Do NOT specify where to insert.
  - The new element is added only at the top.



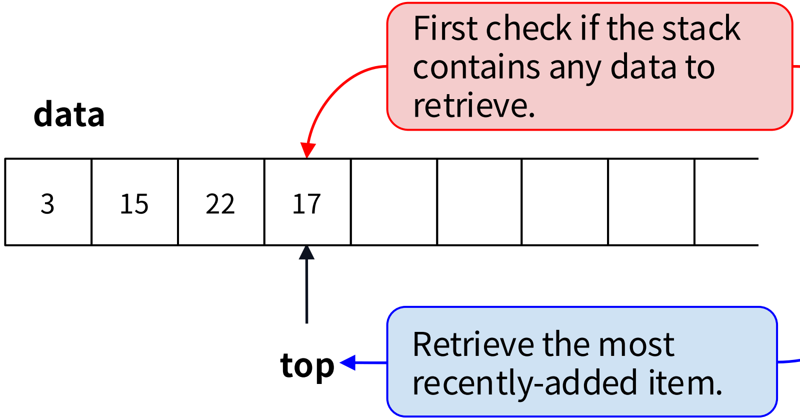
New element is added at the end of the `List` with `append`.

Then, move the top position by 1.

```
class Stack():
  def __init__(self):
    self.data = []
    self.top = -1

  def push(self, x):
    self.data.append(x)
    self.top += 1

  def peek(self):
    # get item

  def pop(self):
    # delete an item

  def is_empty(self):
    return (self.top == -1)
```

Stack Class Implementation

- Peek
  - Again, do NOT specify where to retrieve.
  - Stack always retrieves only the top element.



3	15	22	17					

First check if the stack contains any data to retrieve.

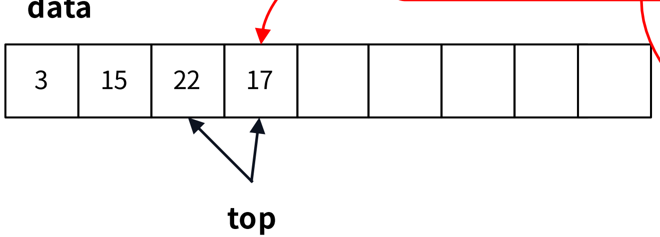
Retrieve the most recently-added item.

```
class Stack():
  def __init__(self):
    self.data = []
    self.top = -1

  def push(self, x):
    self.data.append(x)
    self.top += 1

  def peek(self):
    if not self.is_empty():
     return self.data[self.top]
    else: return None

  def pop(self):
    # delete an item

  def is_empty(self):
    return (self.top == -1)
```

Stack Class Implementation

- Pop
  - ○ Again, do NOT specify from where to delete.
  - ○ Stack always pops only the top element.

Remove the top item, then move the `top` pointer by 1!

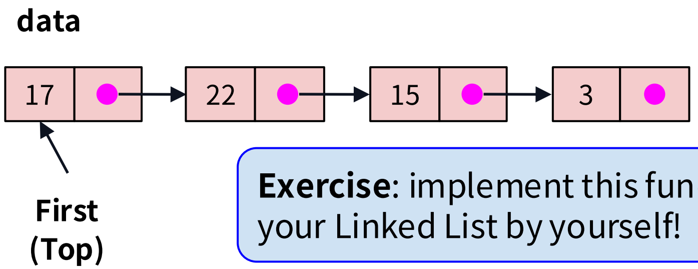


Note that with Python `list`, we may not explicitly need the variable `top`; instead, we may simply use `len(self.list)` to figure out the top position.

```
class Stack():
  def __init__(self):
    self.data = []
    self.top = -1

  def push(self, x):
    self.data.append(x)
    self.top += 1

  def peek(self):
    if not self.is_empty():
      return self.data[self.top]
    else: return None
  def pop(self):
    if not self.is_empty():
      del self.data[self.top]
      self.top -= 1
    else: return None
  def is_empty(self): (omitted)
```

Stack Class Design

- Reference-based Implementation
  - We may implement this by using a **Linked List**.
  - Recall that the singly linked list is accessed from the **first element**, sequentially.
    - We may naturally use the first element as the **top** element!
    - Thus, we don't have to maintain the top index.



Exercise: implement this function in your Linked List by yourself!

```
class Stack():
 def __init__(self):
  self.data = LinkedList()

 def push(self, x):
  # insert x

 def peek(self):
  # get item

 def pop(self):
  # delete an item

 def is_empty(self):
  return self.data.is_empty()
```

Stack Class Implementation

- How to implement push, peek, pop?
  - Use the functions of the Linked List!

```
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]
```

```
class Stack():
  def __init__(self):
    self.data = LinkedList()

  def push(self, x):
    self.data.insert(x, 0)

  def peek(self):
    return self.data.get(0)

  def pop(self):
    self.data.delete(0)

  def is_empty(self):
    return self.data.is_empty()
```

Time Complexity

- Time complexity of Stack?

Task	Array-based	Reference-based
Insertion	O(1)	O(1)
Retrieval	O(1)	O(1)
Deletion	O(1)	O(1)

= Best cases only in linked list

Stack is more efficient than (more general) array or linked list, if the data and problem satisfy stack's condition!

02

Applications of Stacks (Homework)

Application Questions

- Use stack(s) to check if a string with parentheses is well-formed.
  - "`(3+4)*(2+5)`" is well-formed.
  - "`((2*2)*3+1`" is not well-formed.
  - "`)(2+2`" is not well-formed.

- What if we have more than one types of parentheses?
  - "`{(2+1)*(3+2)-22}*7`" is well-formed.
  - "`{(7+2}*3)`" is not well-formed.

03

Queues

Queue Examples

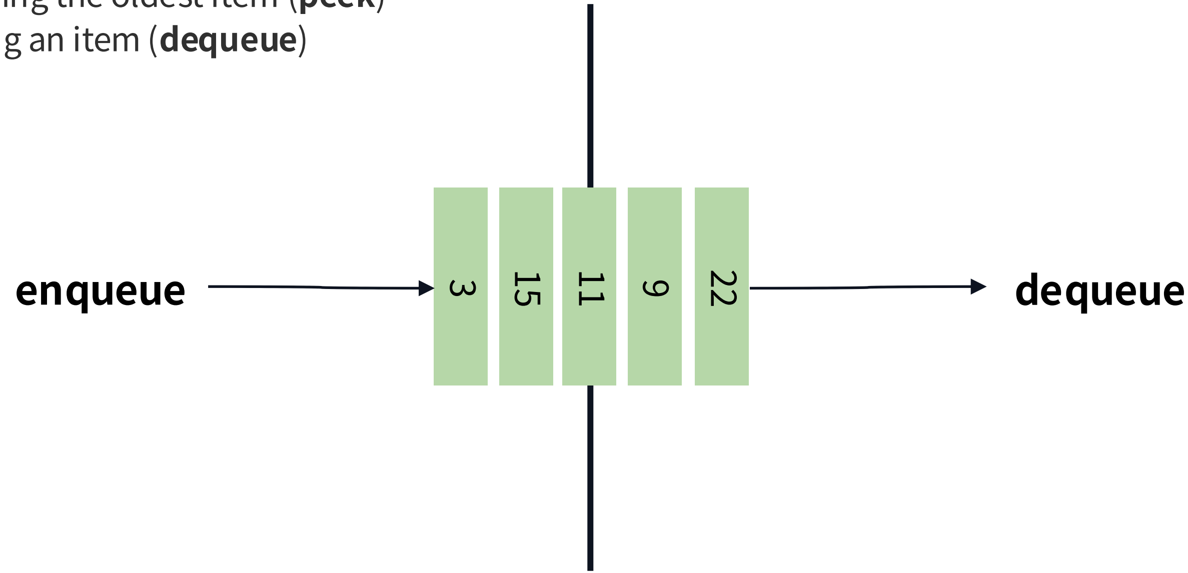
[1] Line of passengers at airport security

[2] Drink older milk first

[1] https://www.nbcnews.com/business/travel/tsa-replaces-head-security-airport-lines-keep-getting-longer-n579021
[2] https://brunch.co.kr/@myolivenote/1974

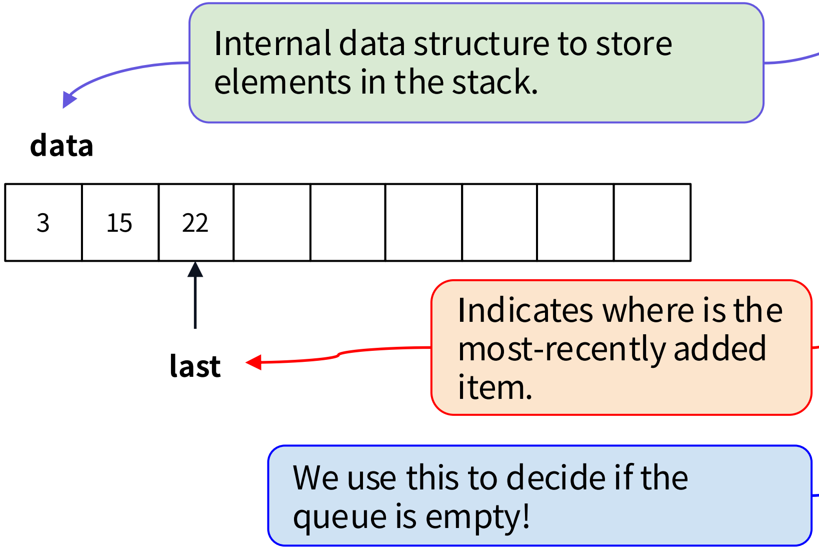
Queue Terminologies

- Similarly to the stack, queue also uses its own jargons:
    - Adding a new element (**enqueue**)
    - Retrieving the oldest item (**peek**)
    - Deleting an item (**dequeue**)



Queue Class Design

- Array-based Implementation
  - ○ We use Python `List` for simplicity here.



Internal data structure to store elements in the stack.

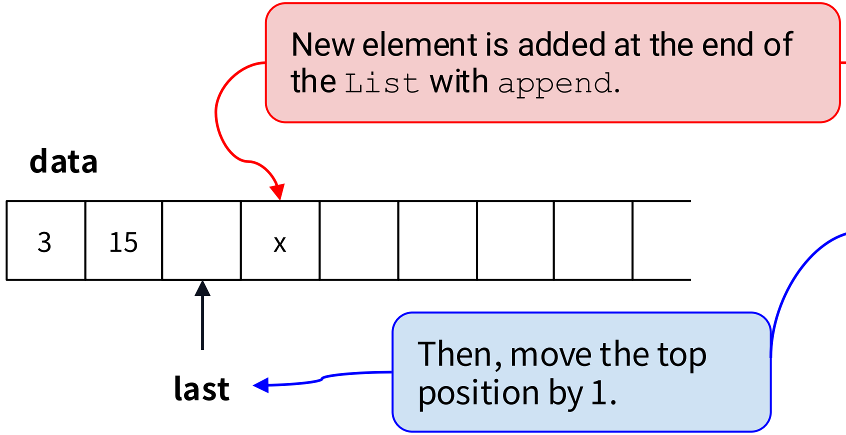
Indicates where is the most-recently added item.

We use this to decide if the queue is empty!

```
class Queue():
  def __init__(self):
    self.data = []
    self.last = -1

  def enqueue(self, x):
    # insert x

  def peek(self):
    # get item

  def dequeue(self):
    # delete an item

  def is_empty(self):
    return (self.last == -1)
```

Queue Class Implementation

- Enqueue
  - Insert always at the end (last).
  - Same as push in stack.



New element is added at the end of the `List` with `append`.

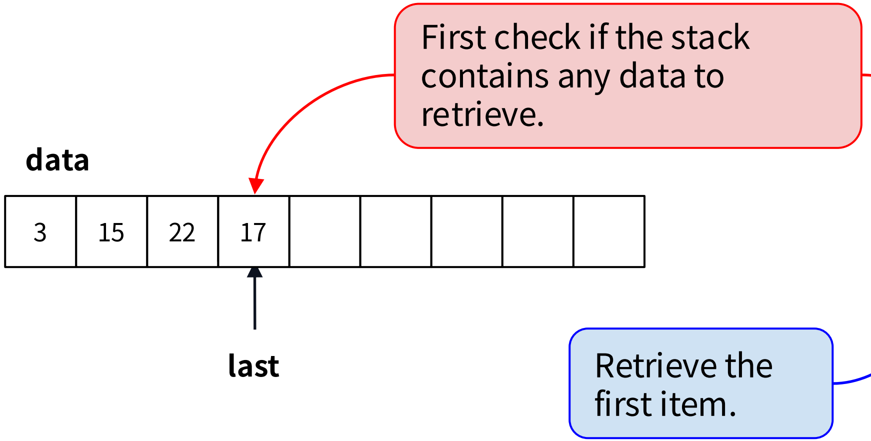
Then, move the top position by 1.

```
class Queue():
  def __init__(self):
    self.data = []
    self.last = -1

  def enqueue(self, x):
    self.data.append(x)
    self.last += 1

  def peek(self):
    # get item

  def dequeue(self):
    # delete an item

  def is_empty(self):
    return (self.last == -1)
```

Time complexity? **O(1)**

Queue Class Implementation

- Peek
  - We retrieve always the oldest item, which is located at the first.



First check if the stack contains any data to retrieve.

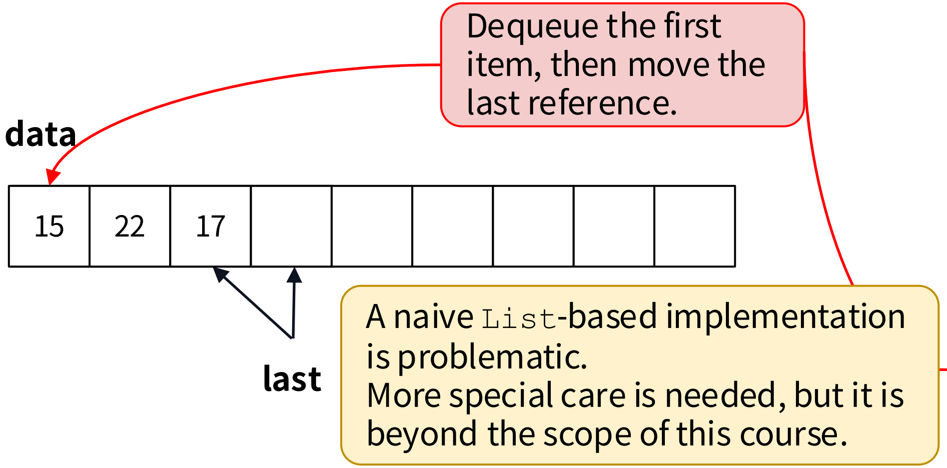
Retrieve the first item.

Time complexity? **O(1)**

```
class Queue():
  def __init__(self):
    self.data = []
    self.last = -1

  def enqueue(self, x):
    self.data.append(x)
    self.last += 1

  def peek(self):
    if not self.is_empty():
      return self.data[0]
    else: return None

  def dequeue(self):
    # delete an item

  def is_empty(self):
    return (self.last == -1)
```

Queue Class Implementation

- Dequeue
  - We dequeue always the oldest item, located at the first.



Dequeue the first item, then move the last reference.

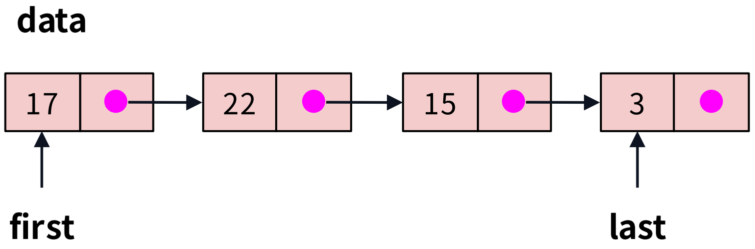
A naive `List`-based implementation is problematic.
More special care is needed, but it is beyond the scope of this course.

Time complexity? **O(*N*)** 😱

```
class Queue():
  def __init__(self):
    self.data = []
    self.last = -1

  def enqueue(self, x):
    self.data.append(x)
    self.last += 1

  def peek(self):
    if not self.is_empty():
      return self.data[0]
    else: return None

  def dequeue(self):
    if not self.is_empty():
      del self.data[0]
      self.last -= 1
  def is_empty(self): (omitted)
```

Queue Class Design

- Reference-based Implementation
  - Similarly to Stack, let's try **Linked List**.
  - As we enqueue and dequeue from different ends, we may keep references for both!
    - The beginning is naturally provided by the Linked List, so we only need to add the last reference.



```
class Queue():
  def __init__(self):
    self.data = LinkedList()
    self.last = None

  def enqueue(self, x):
    # insert x

  def peek(self):
    # get item

  def dequeue(self):
    # delete an item

  def is_empty(self):
    return self.data.is_empty()
```

Queue Class Implementation

- How to implement enqueue, peek, dequeue?
  - ○ Use the functions of the Linked List!

```
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]
```

```
class Queue():
  def __init__(self):
    self.data = LinkedList()
    self.last = None

  def enqueue(self, x):
    # insert x

  def peek(self):
    return self.data.get(0)

  def dequeue(self):
    self.data.delete(0)

  def is_empty(self):
    return self.data.is_empty()
```

Queue Class Implementation

- How to implement enqueue, peek, dequeue?
  - ○ `Enqueue` is not as simple as others!
  - ○ First, we do not know the last index.
  - ○ Even though we maintain it, the `insert` of `LinkedList` will traverse the entire list, taking O(*N*) 😱.
  - ○ To avoid this, we need to take advantage of the `self.last` reference directly!

```
new_node = Node(x)
if self.last is None:
  self.data.first = new_node
else:
  self.last.next = new_node
self.last = new_node
```

```
class Queue():
  def __init__(self):
    self.data = LinkedList()
    self.last = None

  def enqueue(self, x):
    self.data.insert(x, ?)

  def peek(self):
    return self.data.get(0)

  def dequeue(self):
    self.data.delete(0)

  def is_empty(self):
    return
self.data.is_empty()
```

Time Complexity

- Time complexity of Queue?

Task	Array-based	Reference-based
Insertion	O(1)	O(1)
Retrieval	O(1)	O(1)
Deletion	O(1)	O(1)

We need special implementation to make deletion in O(1).

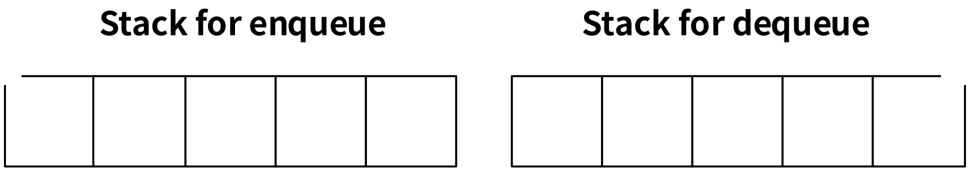
= Best cases only in linked list

04

Applications of Queues

Application Questions

- Implement Queue using two Stacks.
  - Main idea: use the first stack for enqueue, and the other for dequeue.
  - Whenever we get a dequeue request but the second stack is empty, pop all elements from the first stack and push them into the second stack.



upstage

Building intelligence for the future of work