GRAPH THEORY
[2]

Documents are here:

https://www-l2ti.univ-paris13.fr/~viennet/ens/2024-USTH-Graphs

Abstract Data Types: Stacks, Queues, and Dictionaries

Slides adapted from Champion & Chun

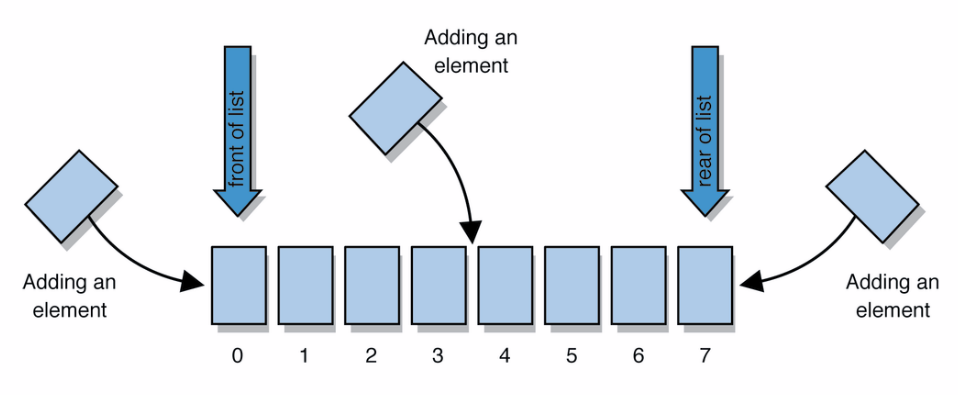
Abstract Data Types (ADT)

Abstract Data Types

- An abstract definition for expected operations and behavior
- Defines the input and outputs, not the implementations

Review: List - a collection storing an ordered sequence of elements

- each element is accessible by a 0-based index
- a list has a size (number of elements that have been added)
- elements can be added to the front, back, or elsewhere
- in Java, a list can be represented as an ArrayList object



Review: Interfaces

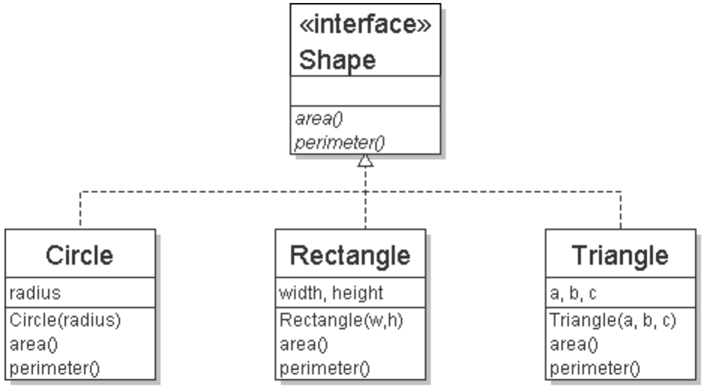
interface: A construct in Java that defines a set of methods that a class promises to implement

- Interfaces give you an is-a relationship *without* code sharing.
  - A `Rectangle` object can be treated as a `Shape` but inherits no code.
- Analogous to non-programming idea of roles or certifications:
  - "I'm certified as a CPA accountant.
    This assures you I know how to do taxes, audits, and consulting."
  - "I'm 'certified' as a Shape, because I implement the Shape interface.
    This assures you I know how to compute my area and perimeter."

```
public interface name {
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
    ...
    public type name(type name, ..., type name);
}
```

Example

```
// Describes features common to all
// shapes.
public interface Shape {
        public double area();
        public double perimeter();
}
```



Review: Java Collections

Java provides some implementations of ADTs for you!

ADTs	Data Structures
Lists	`List<Integer> a = new ArrayList<Integer>();`
Stacks	`Stack<Character> c = new Stack<Character>();`
Queues	`Queue<String> b = new LinkedList<String>();`
Maps	`Map<String, String> d = new TreeMap<String, String>();`

We can build other data structures from scratch, e.g.

Linked Lists - LinkedIntList a collection of ListNode

Review: Python Collections

Python provides some implementations of ADTs as native types

ADTs	Data Structures
Lists	`list [1, 2, 3]`
Stacks	use `list` or `collection.deque`
Queues	`collection.deque`
Maps	`dict, {}, { key : value }`

We can build other data structures from scratch by defining new classes.

Full Definitions

Abstract Data Type (ADT)

- *A definition for expected operations and behavior*
- A mathematical description of a collection with a set of supported operations and how they should behave when called upon
- Describes what a collection does, not how it does it
- Can be expressed as an interface
- Examples: List, Map, Set

Data Structure

- *A way of organizing and storing related data points*
- An object that implements the functionality of a specified ADT
- Describes exactly how the collection will perform the required operations
- Examples: LinkedIntList, ArrayIntList

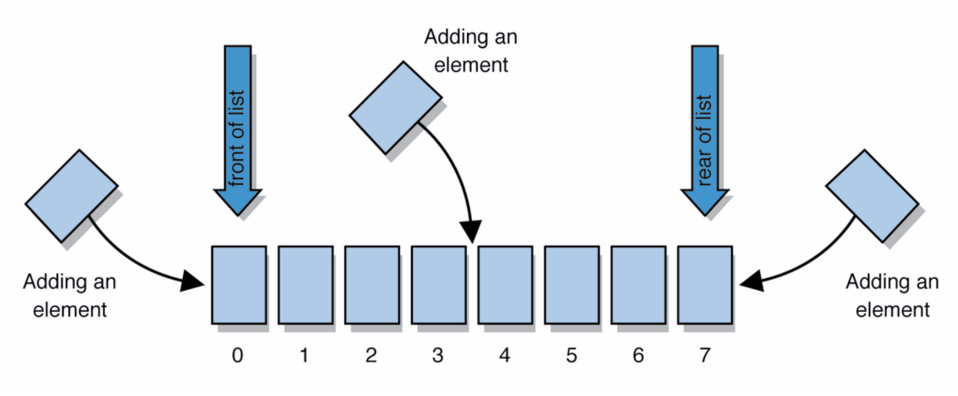
Case Study: The List ADT

list: a collection storing an ordered sequence of elements.

- Each item is accessible by an index.
- A list has a size defined as the number of elements in the list

Expected Behavior:

- **get(index):** returns the item at the given index
- **set(value, index):** sets the item at the given index to the given value
- **append(value):** adds the given item to the end of the list
- **insert(value, index):** insert the given item at the given index maintaining order
- **delete(index):** removes the item at the given index maintaining order
- **size():** returns the number of elements in the list



Case Study: List Implementations

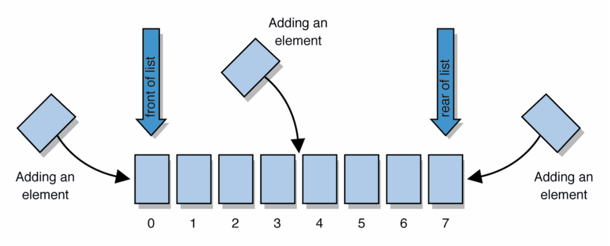
List ADT

state

Set of ordered items
Count of items

behavior

get(index) return item at index
set(item, index) replace item at index
append(item) add item to end of list
insert(item, index) add item at index
delete(index) delete item at index
size() count of items



ArrayList

uses an Array as underlying storage

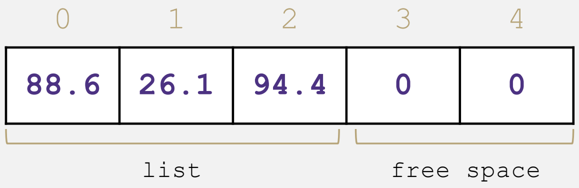
ArrayList<E>

state

data[]
size

behavior

get return data[index]
set data[index] = value
append data[size] = value, if out of space grow data
insert shift values to make hole at index, data[index] = value, if out of space grow data
delete shift following values forward
size return size



LinkedList

uses nodes as underlying storage

LinkedList<E>

state

Node front
size

behavior

get loop until index, return node's value
set loop until index, update node's value
append create new node, update next of last node
insert create new node, loop until index, update next fields
delete loop until index, skip node
size return size

Implementing ArrayList

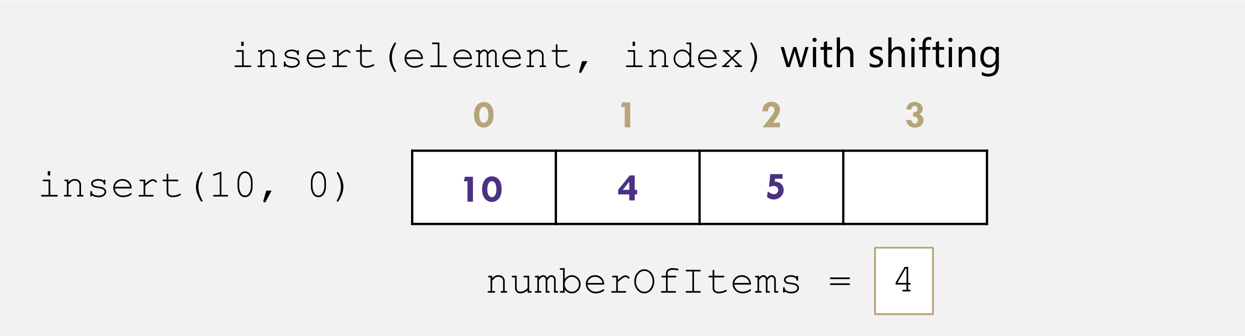
ArrayList<E>

state

`data[]`
`size`

behavior

`get` return data[index]
`set` data[index] = value
`append` data[size] = value, if out of space grow data
`insert` shift values to make hole at index, data[index] = value, if out of space grow data
`delete` shift following values forward
`size` return size

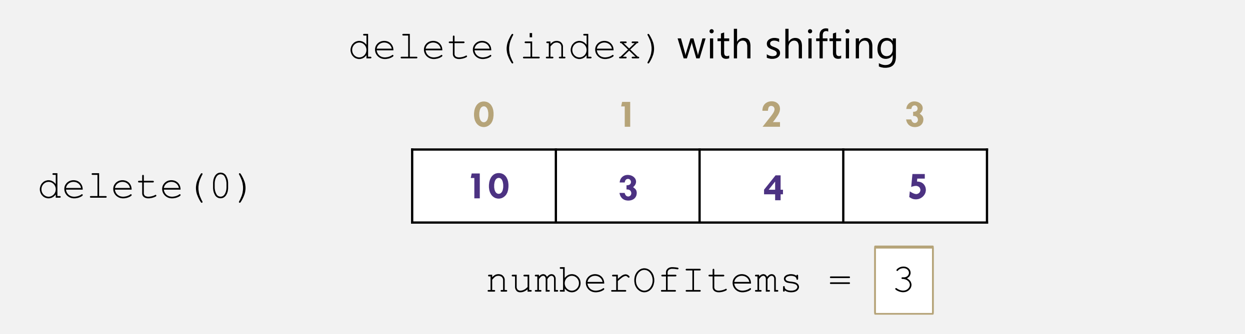


`insert(element, index)` with shifting

`insert(10, 0)`

0	1	2	3
10	4	5	

`numberOfItems = 4`



`delete(index)` with shifting

`delete(0)`

0	1	2	3
10	3	4	5

`numberOfItems = 3`

Implementing ArrayList

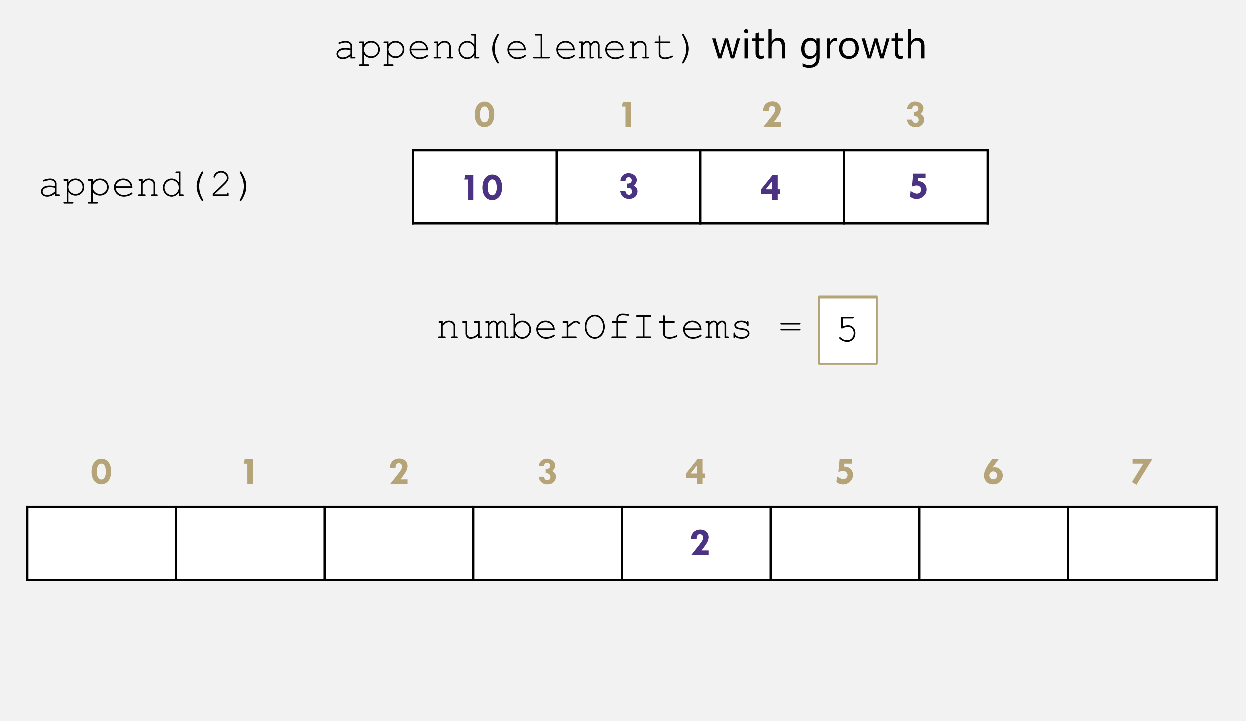
ArrayList<E>

`state`

`data[]`
`size`

`behavior`

`get` return data[index]
`set` data[index] = value
`append` data[size] = value, if out of space grow data
`insert` shift values to make hole at index, data[index] = value, if out of space grow data
`delete` shift following values forward
`size` return size



`append(element)` with growth

`append(2)`

0	1	2	3
10	3	4	5

`numberOfItems` = 5

0	1	2	3	4	5	6	7
				2			

Design Decisions

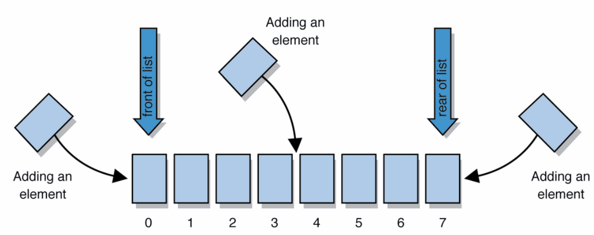
For every ADT there are lots of different ways to implement them

Based on your situation you should consider:

- Memory vs Speed
- Generic/Reusability vs Specific/Specialized
- One Function vs Another
- Robustness vs Performance

This class is all about implementing ADTs based on making the right design tradeoffs!

Questions !



Q: Would you use a LinkedList or ArrayList implementation for each of these scenarios?

List ADT

state
Set of ordered items
Count of items

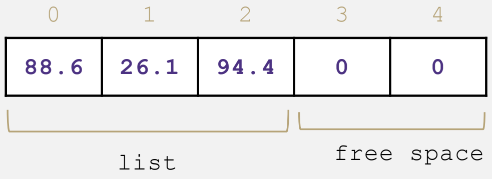
behavior
get(index) return item at index
set(item, index) replace item at index
append(item) add item to end of list
insert(item, index) add item at index
delete(index) delete item at index
size() count of items

ArrayList
uses an Array as underlying storage

ArrayList

state
data[]
size

behavior
get return data[index]
set data[index] = value
add data[size] = value, if out of space grow data
insert shift values to make hole at index, data[index] = value, if out of space grow data
delete shift following values forward
size return size



LinkedList
uses nodes as underlying storage

LinkedList

state
Node front
size

behavior
get loop until index, return node's value
set loop until index, update node's value
add create new node, update next of last node
insert create new node, loop until index, update next fields
delete loop until index, skip node
size return size

Situation #1: Choose a data structure that implements the List ADT that will be used to store a list of songs in a playlist.

Situation #2: Choose a data structure that implements the List ADT that will be used to store the history of a bank customer's transactions.

Situation #3: Choose a data structure that implements the List ADT that will be used to store the order of students waiting to speak to a teacher at a tutoring center

Design Decisions : **Situation #1** playlist

Write a data structure that implements the List ADT that will be used to store a list of songs in a playlist.

Common operations:

- add/delete a song (rare)
- play (iterate through the playlist)
- shuffle play

ArrayList – I want to be able to shuffle play on the playlist

Design Decisions **Situation #2** bank

Write a data structure that implements the List ADT that will be used to store the history of a bank customer's transactions.

Common operations:

- add a record (frequent, large amount)
- access (iterate through the list)

ArrayList – optimize for addition to back and accessing of elements

Design Decisions **Situation #3** students

Write a data structure that implements the List ADT that will be used to store the order of students waiting to speak to a teacher at a tutoring center

Common operations:

- add a student to back
- remove a student from front

LinkedList - optimize for removal from front

ArrayList – optimize for addition to back

List ADT tradeoffs

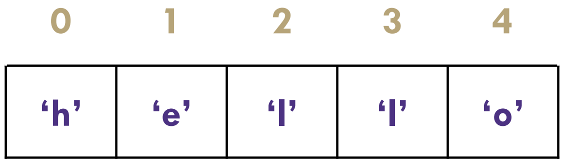
Last time: we used “slow” and “fast” to describe running times. Let’s be a little more precise.

Recall these basic Big-O ideas from : Suppose our list has N elements
- If a method takes a constant number of steps (like 23 or 5) its running time is O(1)
- If a method takes a linear number of steps (like 4N+3) its running time is O(N)

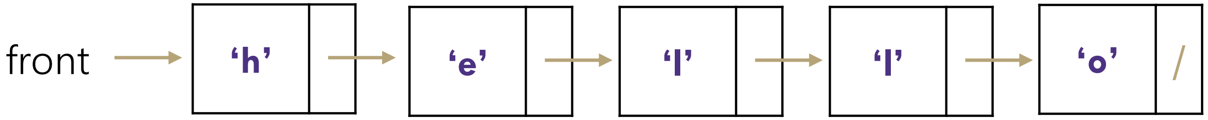
For ArrayLists and LinkedLists, what is the O() for each of these operations?
- Time needed to access N^{th} element:
- Time needed to insert at end (the array is full!)

What are the memory tradeoffs for our two implementations?
- Amount of space used overall
- Amount of space used per element

`ArrayList<Character> myArr`



`LinkedList<Character> myLl`



List ADT tradeoffs

Time needed to access N^{th} element:

- ArrayList: O(1) constant time
- LinkedList: O(N) linear time

Time needed to insert at N^{th} element (the array is full!)

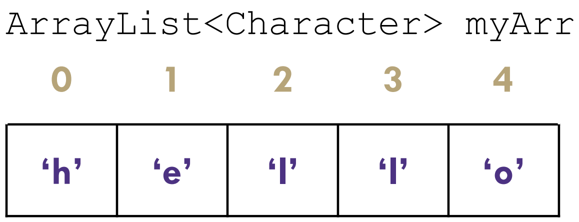
- ArrayList: O(N) linear time
- LinkedList: O(N) linear time

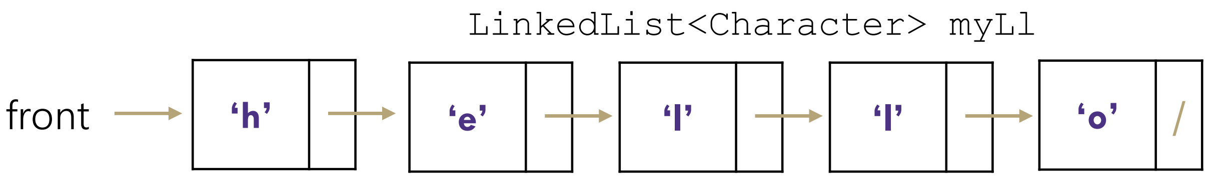
Amount of space used overall

- ArrayList: sometimes wasted space
- LinkedList: compact

Amount of space used per element

- ArrayList: minimal
- LinkedList: tiny extra



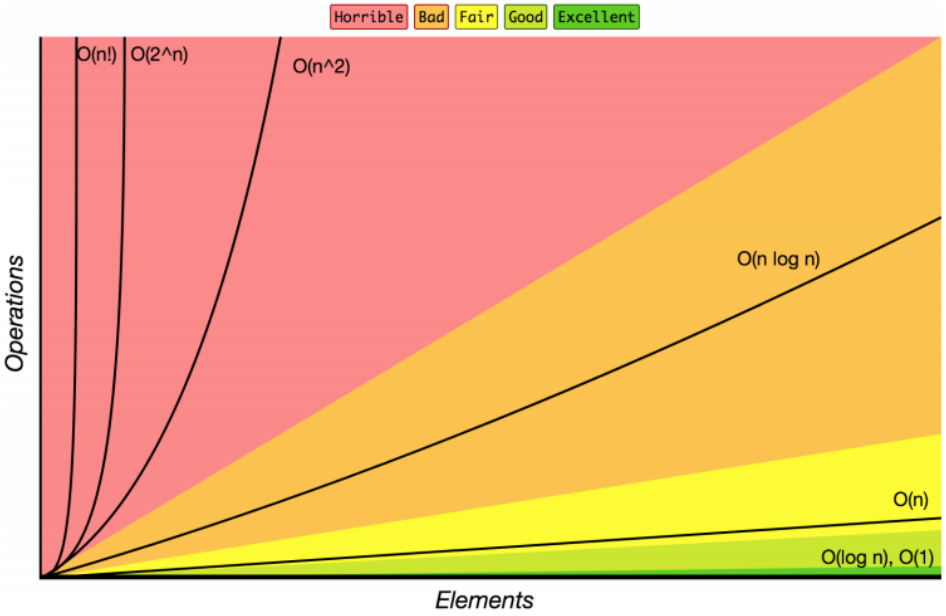


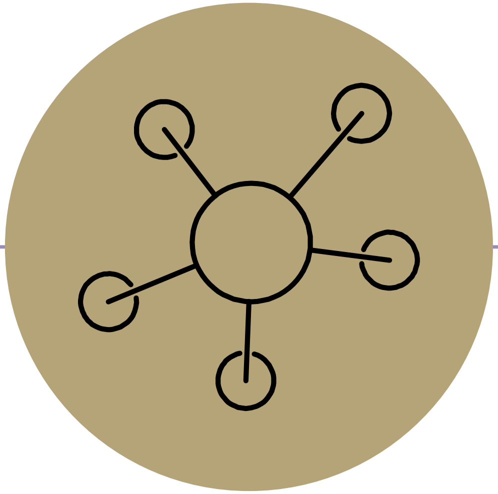
Review: Complexity Class

Note: You don't have to understand all of this right now – we'll dive into it soon.

complexity class: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

Complexity Class	Big-O	Runtime if you double N	Example Algorithm
constant	O(1)	unchanged	Accessing an index of an array
logarithmic	$O(\log_2 N)$	increases slightly	Binary search
linear	O(N)	doubles	Looping over an array
log-linear	$O(N \log_2 N)$	slightly more than doubles	Merge sort algorithm
quadratic	$O(N^2)$	quadruples	Nested loops!
...	...	...	...
exponential	$O(2^N)$	multiplies drastically	Fibonacci with recursion



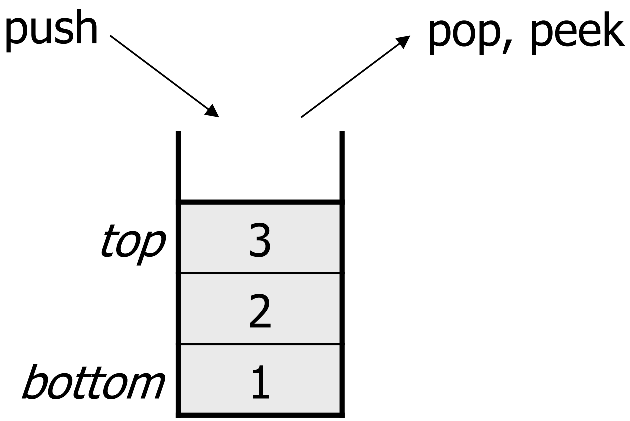


Questions?

Review: What is a Stack?

stack: A collection based on the principle of adding elements and retrieving them in the opposite order.

- Last-In, First-Out ("LIFO")
- Elements are stored in order of insertion.
  - We do not think of them as having indexes.
- Client can only add/remove/examine the last element added (the "top").



Stack ADT

state
Set of ordered items
Number of items

behavior
push(item) add item to top
pop() return and remove item at top
peek() look at item at top
size() count of items
isEmpty() count of items is 0?

supported operations:

- **push(item)**: Add an element to the top of stack
- **pop()**: Remove the top element and returns it
- **peek()**: Examine the top element without removing it
- **size()**: how many items are in the stack?
- **isEmpty()**: true if there are 1 or more items in stack, false otherwise

Implementing a Stack with an Array

Stack ADT

state

Set of ordered items
Number of items

behavior

push(item) add item to top
pop() return and remove item at top
peek() look at item at top
size() count of items
isEmpty() count of items is 0?

ArrayStack<E>

state

```
data[]
size
```

behavior

```
push data[size] = value, if
out of room grow data
pop return data[size - 1],
size-1
peek return data[size - 1]
size return size
isEmpty return size == 0
```

Big O Analysis

`pop()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`push()`	O(N) linear if you have to resize O(1) otherwise

```
push(3)
push(4)
pop()
push(5)
```

0	1	2	3
3	5		

`numberOfItems = 2`

Question

What do you think the worst possible runtime of the “push()” operation will be?

Implementing a Stack with Nodes

Stack ADT

state

Set of ordered items
Number of items

behavior

push(item) add item to top
pop() return and remove item at top
peek() look at item at top
size() count of items
isEmpty() count of items is 0?

LinkedStack<E>

state

```
Node top
size
```

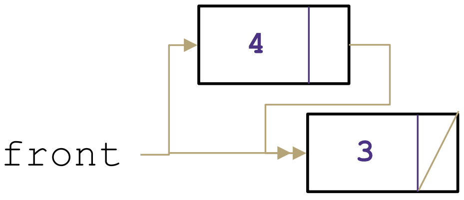
behavior

```
push add new node at top
pop return and remove node at top
peek return node at top
size return size
isEmpty return size == 0
```

Big O Analysis

`pop()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`push()`	O(1) Constant

```
push(3)
push(4)
pop()
```



numberOfItems = 2

Question

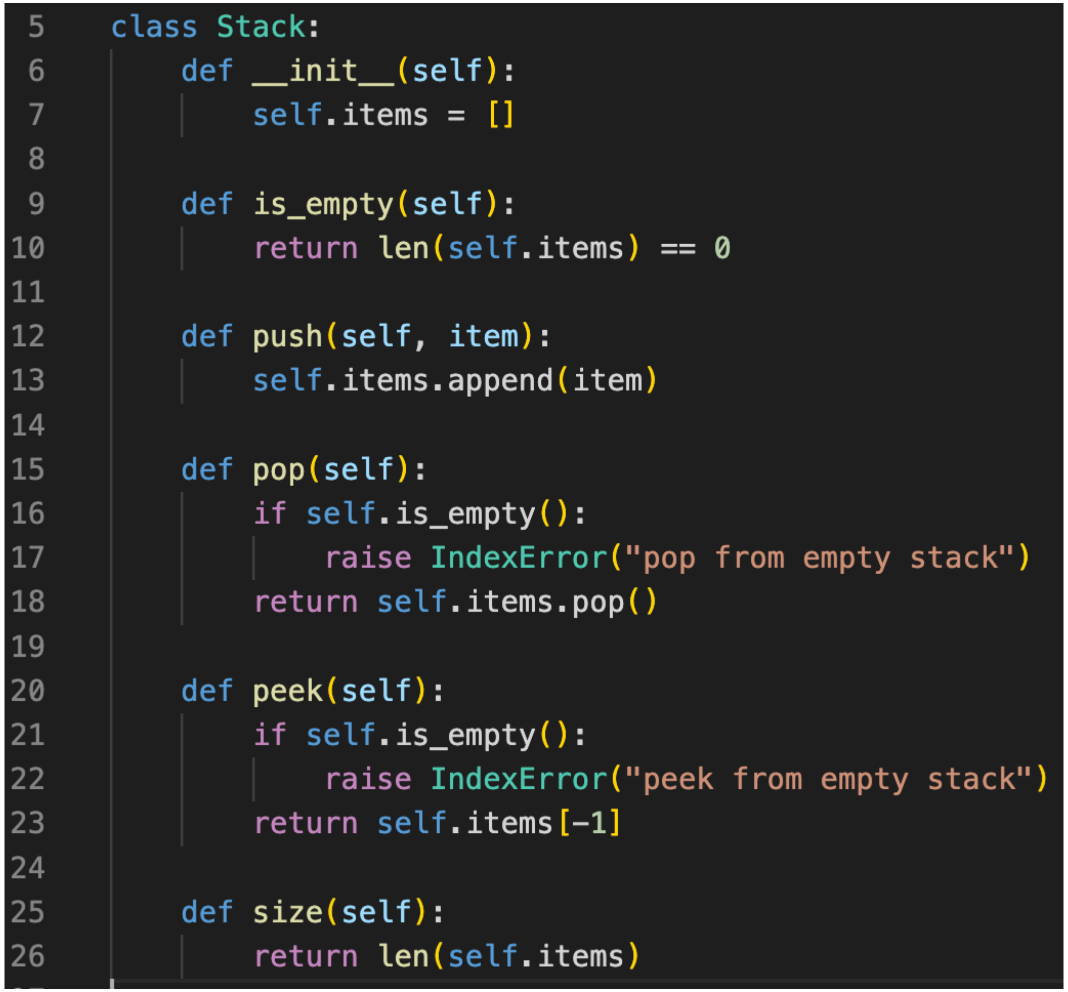
What do you think the worst possible runtime of the “push()” operation will be?

Implementing a Stack with Nodes - Python

Using the native python `list` type

See https://docs.python.org/3/tutorial/datastructures.html

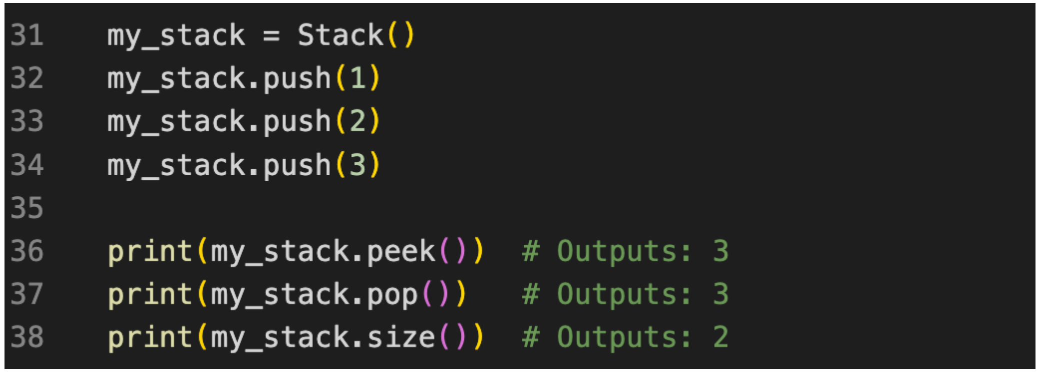
For serious work, use provided efficient types, like `collections.deque` https://docs.python.org/3/library/collections.html#collections.deque



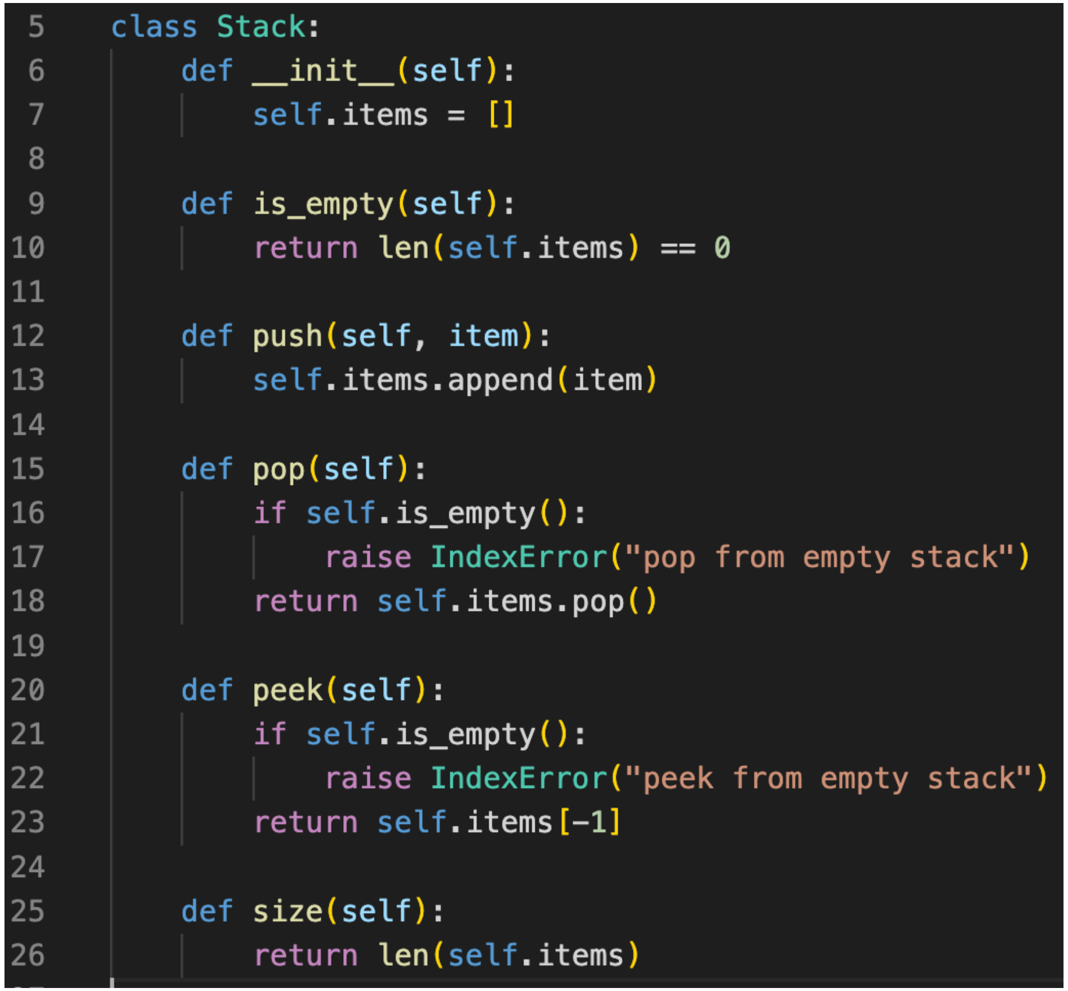
```python
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self.items.pop()

    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self.items[-1]

    def size(self):
        return len(self.items)
```

Implementing a Stack with Nodes - Python

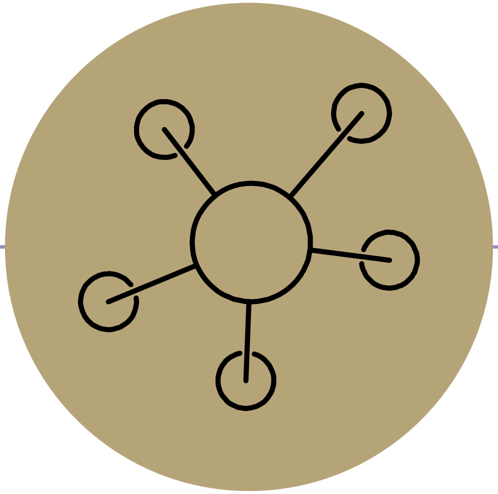
Example usage :



```python
my_stack = Stack()
my_stack.push(1)
my_stack.push(2)
my_stack.push(3)

print(my_stack.peek())  # Outputs: 3
print(my_stack.pop())   # Outputs: 3
print(my_stack.size())  # Outputs: 2
```



```python
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self.items.pop()

    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self.items[-1]

    def size(self):
        return len(self.items)
```



Question Break

Review: What is a Queue?

queue: Retrieves elements in the order they were added.

- First-In, First-Out ("FIFO")
- Elements are stored in order of insertion but don't have indexes.
- Client can only add to the end of the queue, and can only examine/remove the front of the queue.

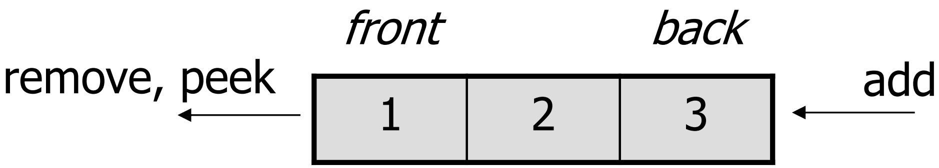
Queue ADT

state

Set of ordered items
Number of items

behavior

add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count of items is 0?



front *back*

remove, peek

1	2	3

add

supported operations:

- **add(item):** aka “enqueue” add an element to the back.
- **remove():** aka “dequeue” Remove the front element and return.
- **peek()**: Examine the front element without removing it.
- **size():** how many items are stored in the queue?
- **isEmpty():** if 1 or more items in the queue returns true, false otherwise

Implementing a Queue with an Array

Queue ADT

state

Set of ordered items
Number of items

behavior

add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count of items is 0?

ArrayQueue<E>

state

```
data[]
Size
front index
back index
```

behavior

```
add – data[size] = value, if
out of room grow data
remove – return data[size -
1], size-1
peek – return data[size – 1]
size – return size
isEmpty – return size == 0
```

Big O Analysis

`remove()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	O(N) linear if you have to resize O(1) otherwise

Question

What do you think the worst possible runtime of the "add()" operation will be?

```
add(5)
add(8)
add(9)
remove()
```

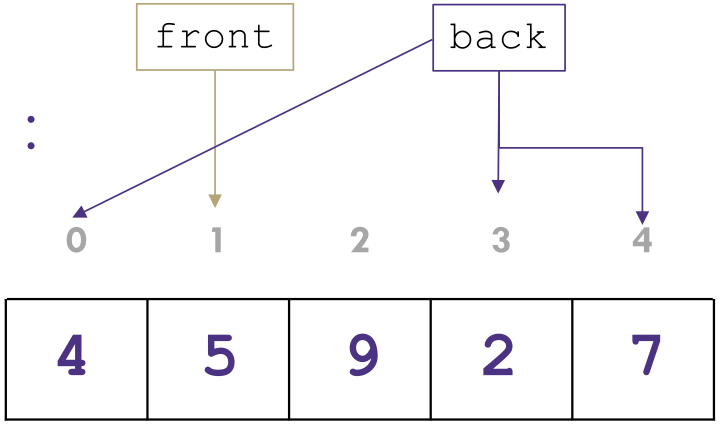
0	1	2	3	4
5	8	9		

```
numberOfItems = 3
front = 1
back = 2
```

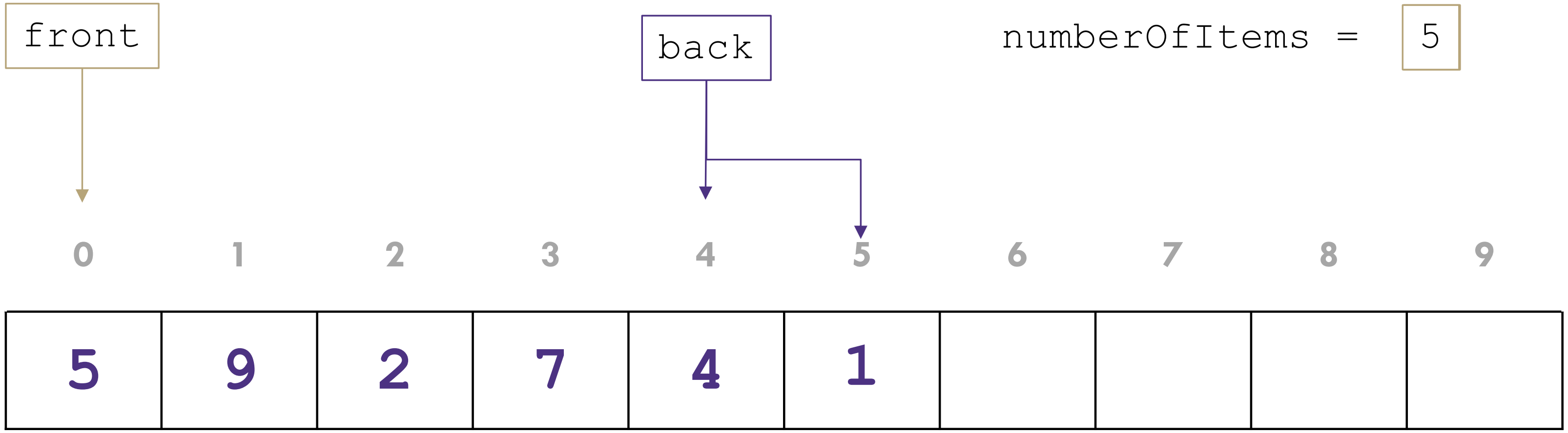
Implementing a Queue with an Array

add(7)
add(4)
add(1) *ouch* ?

Wrapping Around :



numberOfItems = 5



Implementing a Queue with Nodes

Queue ADT

state

Set of ordered items
Number of items

behavior

add(item) add item to **back**
remove() remove and return item at **front**
peek() return item at **front**
size() count of items
isEmpty() count of items is 0?

LinkedQueue<E>

state

```
Node front
Node back
size
```

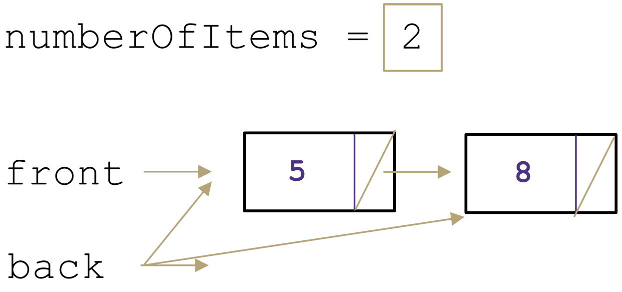
behavior

```
add - add node to back
remove - return and remove
node at front
peek - return node at front
size - return size
isEmpty - return size == 0
```

Big O Analysis

`remove()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	O(1) Constant

```
add(5)
add(8)
remove()
```

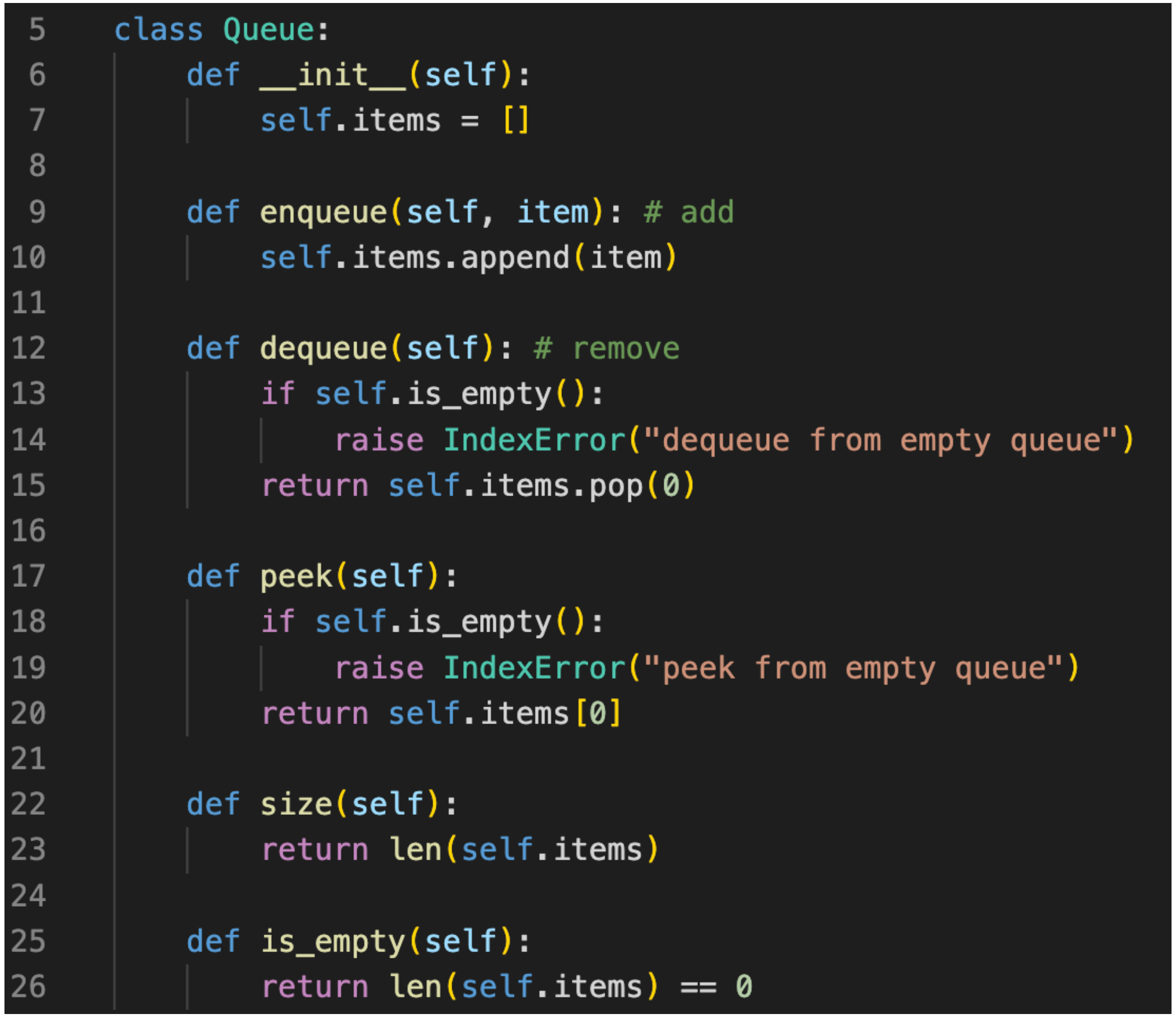


numberOfItems = 2

Implementing a Queue in Python (simple)

Using the native python `list` type

Not recommended for real applications due to the time complexity append() and pop()

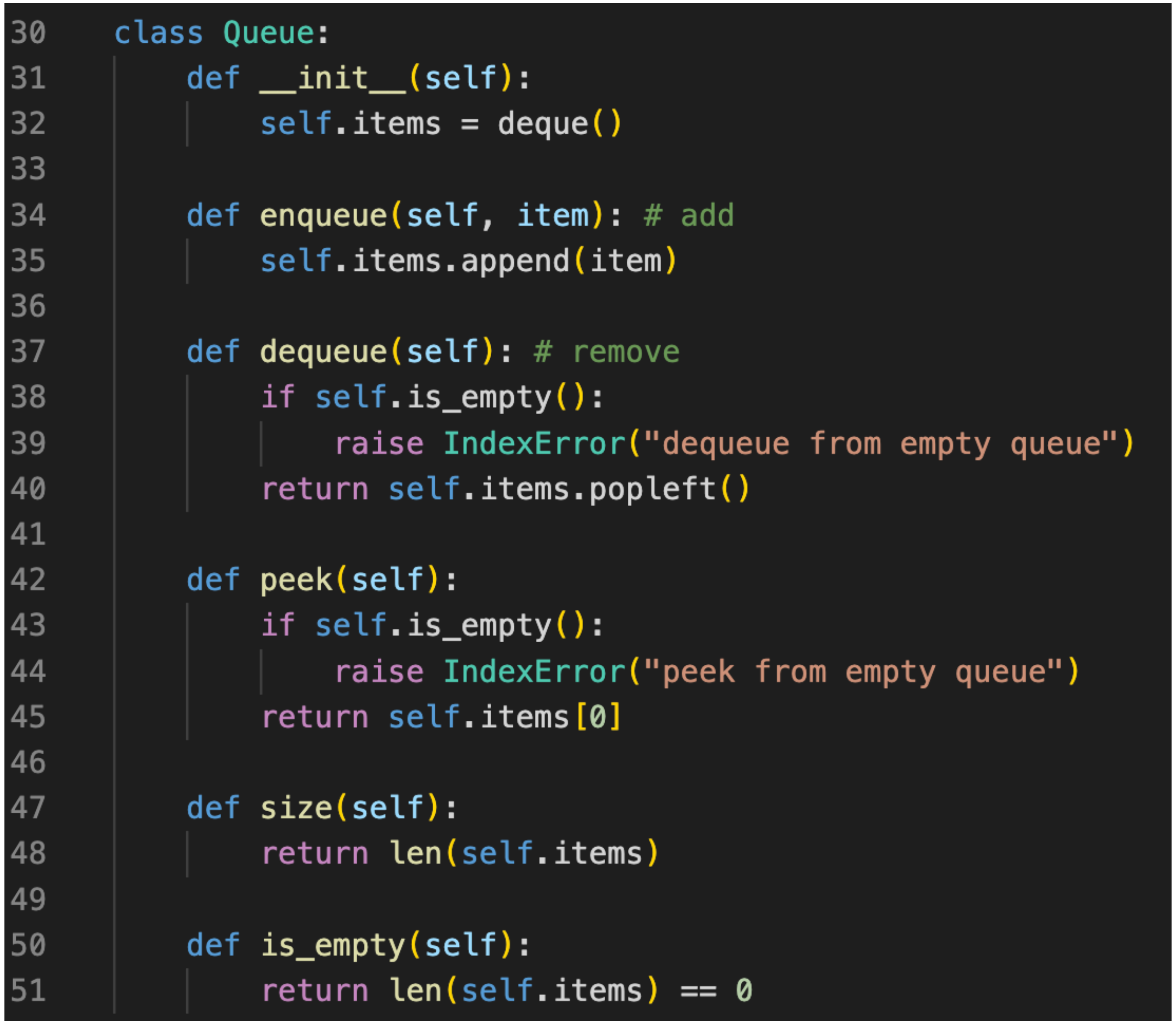


```python
class Queue:
    def __init__(self):
        self.items = []

    def enqueue(self, item): # add
        self.items.append(item)

    def dequeue(self): # remove
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self.items.pop(0)

    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty queue")
        return self.items[0]

    def size(self):
        return len(self.items)

    def is_empty(self):
        return len(self.items) == 0
```

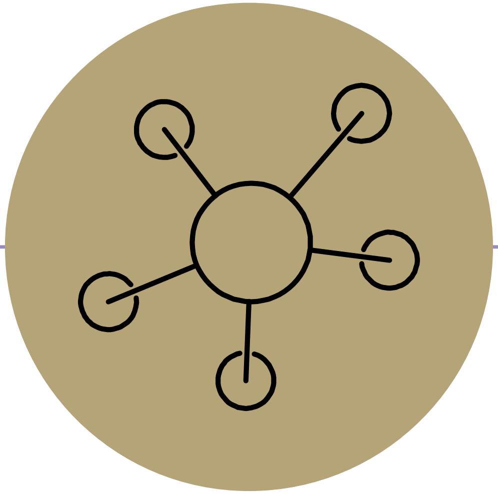
Implementing a Queue in Python (better)

Using the python’s `collections.deque` type

Efficient because `append()` and `popleft()` are O(1)



```python
class Queue:
    def __init__(self):
        self.items = deque()

    def enqueue(self, item): # add
        self.items.append(item)

    def dequeue(self): # remove
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self.items.popleft()

    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty queue")
        return self.items[0]

    def size(self):
        return len(self.items)

    def is_empty(self):
        return len(self.items) == 0
```



Questions?

Design Decisions

Take 5 Minutes

Discuss in your Breakouts: For each scenario select the appropriate ADT and implementation to best optimize for the given scenario.

Situation: You are writing a program to schedule jobs sent to a laser printer. The laser printer should process these jobs in the order in which the requests were received. There are busy and slow times for requests that can have large differences in the volume of jobs sent to the printer. Which ADT and what implementation would you use to store the jobs sent to the printer?

ADT options:
- List
- Stack
- Queue

Implementation options:
- array
- linked nodes

Dictionaries

Dictionaries (aka Maps)

Every Programmer's Best Friend

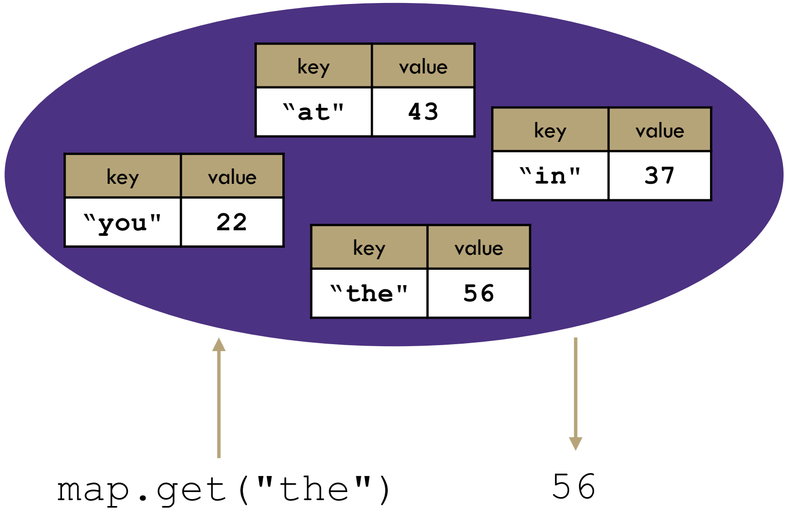
You'll probably use one in almost every programming project.

- Because it's hard to make a big project without needing one sooner or later.

Review: Maps

map: Holds a set of distinct *keys* and a collection of *values*, where each key is associated with one value.

- a.k.a. "dictionary"



Dictionary ADT

state

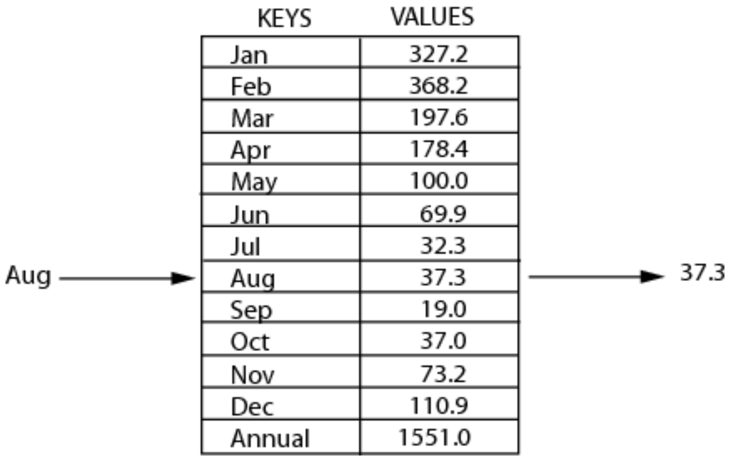
Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

supported operations:

- **put**(*key*, *value*): Adds a given item into collection with associated key,
  - **if the map previously had a mapping for the given key, old value is replaced.**
- **get**(*key*): Retrieves the value mapped to the key
- **containsKey**(key): returns true if key is already associated with value in map, false otherwise
- **remove**(*key*): Removes the given key and its mapped value



KEYS	VALUES
Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9
Annual	1551.0

2 Minutes

Implementing a Dictionary with an Array

Dictionary ADT

state

Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

ArrayDictionary<K, V>

state

`Pair<K, V>[] data`

behavior

`put` find key, overwrite value if there. Otherwise create new pair, add to next available spot, grow array if necessary
`get` scan all pairs looking for given key, return associated item if found
`containsKey` scan all pairs, return if key is found
`remove` scan all pairs, replace pair to be removed with last pair in collection
`size` return count of items in dictionary

```
containsKey('c')
get('d')
put('b', 97)
put('e', 20)
```

0	1	2	3	4
('a', 1)	('b' 97)	('c', 3)	('d', 4)	('e', 20)

Big O Analysis – (if key is the last one looked at / not in the dictionary)

`put()`	O(N) linear
`get()`	O(N) linear
`containsKey()`	O(N) linear
`remove()`	O(N) linear
`size()`	O(1) constant

Big O Analysis – (if the key is the first one looked at)

`put()`	O(1) constant
`get()`	O(1) constant
`containsKey()`	O(1) constant
`remove()`	O(1) constant
`size()`	O(1) constant

2 Minutes

Implementing a Dictionary with Nodes

Dictionary ADT

state

Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

LinkedDictionary<K, V>

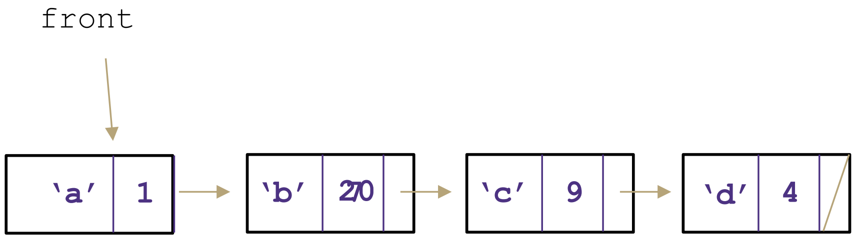
state

```
front
size
```

behavior

```
put if key is unused, create new with
pair, add to front of list, else
replace with new value
get scan all pairs looking for given
key, return associated item if found
containsKey scan all pairs, return if
key is found
remove scan all pairs, skip pair to be
removed
size return count of items in
dictionary
```

```
containsKey('c')
get('d')
put('b', 20)
```



Big O Analysis – (if key is the last one looked at / not in the dictionary)

`put()`	O(N) linear
`get()`	O(N) linear
`containsKey()`	O(N) linear
`remove()`	O(N) linear
`size()`	O(1) constant

Big O Analysis – (if the key is the first one looked at)

`put()`	O(1) constant
`get()`	O(1) constant
`containsKey()`	O(1) constant
`remove()`	O(1) constant
`size()`	O(1) constant