Data structures. ADT.
Stacks and queues

Data structures. Abstract data types (ADT)

- **Data structure.** A collection of data values, the relationships among them, and the functions or operations that can be applied to the data
- Data structures serve as the basis for **abstract data types (ADT)**
- **ADT.** Defines the **logical form** of the data type. A mathematical model for data types, where a **data type is defined by its behavior (semantics)** from the **point of view** of a **user** of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations
- **Data structure.** Defines the **physical form** of the data type. Data structures are concrete representations of data, and are the **point of view** of an **implementer**, not a user

Data types. Applications programming interface (API)

- **Data types.** A data type is a set of values and a set of operations on those values
- **Abstract data types.** An abstract data type is a data type whose **internal representation is hidden from the client**
- **Applications programming interface (API).** To specify the behavior of an abstract data type, we use an application programming interface (API), which is a list of **constructors** and **instance methods (operations)**

Separation between interface and implementation

- Definitions:
  - **Client:** program using operations defined in interface
  - **Implementation:** actual code implementing operations
  - **Interface:** description of data type, basic operations
- Benefits:
  - Client can't know details of implementation ⇒ client has many implementation from which to choose
  - Implementation can't know details of client needs ⇒ many clients can re-use the same implementation
  - **Design:** creates modular, reusable libraries
  - **Performance:** use optimized implementation where it matters

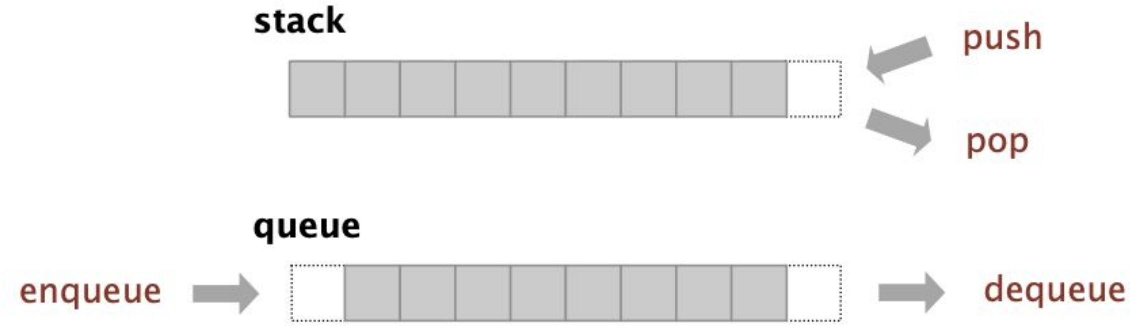
Example. List in Python

- Non-exhaustive list of methods:
    - append (add single element)
    - extend (add elements from another list to this list)
    - clear (remove all elements)
    - index (find the given element, return its index)
- A user doesn't need to know how the list is implemented
- However:
    - It's important to know and understand the **complexity of operations** (which depends on their implementation and the physical data structure)
    - It might be useful to know the internal implementation to take **memory usage** into account
    - Besides asymptotic complexity, there are other factors that impact performance and stem from the implementation internals: constant time, whether elements are stored contiguously or there is a need to follow links (pointers), etc.

Stacks and queues

- Both are **collections of objects**
- Support a pretty standard set of operations: insert, remove, iterate over values, check size / if empty, etc.
- Insertion is the same for both
- The difference is in how elements are removed

Stacks and queues



- **Stack.** Examine the item most recently added (**LIFO = "last in first out"**)
- **Queue.** Examine the item least recently added (**FIFO = "first in first out"**)

Stack implementation (backed by linked list)

- Start with defining the general structure
- The stack would contain instances of *Node*
- *Node* contains: 1) reference to *data* value; 2) reference to the *next Node*
- Stack only points to the current *head* element (of type *Node*)

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

class Stack:
    def __init__(self):
        self.head = None

    def is_empty(self):
        return self.head is None
```

Adding and removing elements

- **Push (insert):** create new *Node* that points to the current *head* (since the new *Node* is on top of the stack), and reassign the *head* reference to point to this new *Node*
- **Pop (remove from the top of the stack):** save the *Node* we're removing, reassign the *head* to point to the next element, and return *data* contained in the *Node* we remove

```python
def push(self, data):
    new_node = Node(data)
    new_node.next = self.head
    self.head = new_node

def pop(self):
    if self.is_empty():
        return None

    popped_node = self.head
    self.head = self.head.next
    popped_node.next = None
    return popped_node.data
```

Convert the stack to string

- Useful for debugging
- Implementation: iterate until we find the last element (by following *next* references), and add every data value we find to the result string
- **Note the method naming:** in Python, methods that start and end with two underscores are called dunder (“double under (scores)”) or “magic” methods

```
def __str__(self):
    if self.is_empty():
        return '[]'

    current_node = self.head
    string = ''
    while current_node is not None:
        if string:
            string += ', '
        else:
            string += '['
        string += str(current_node.data)
        current_node = current_node.next
    string += ']'
    return string
```

Dunder methods

- These methods let you emulate the behavior of built-in types with a special calling convention. However, these are just normal methods
- Example:
  - You have a class *MyObject* and it contains a constructor (__init__ method) that receives a single argument
  - An instance of this class can be created as follows: *MyObject(123)*
  - Actually, this call is “translated” to *MyObject.*__init__*(123)* (you can call it this way as well)
  - Similarly, when you get the length of a collection (*len(my_collection)*), the __len__ dunder method gets invoked
  - Whenever an instance of an object that implements __str__ gets passed to a method that expects a string (like *print(my_object)*), the dunder method __str__ gets invoked

Let’s use this stack

```
stack = Stack()

stack.push(11)
stack.push(22)
stack.push(33)
stack.push(44)

print(stack)  # outputs [44, 33, 22, 11]

stack.pop() # returns 44
stack.pop() # returns 33

print(stack)  # outputs [22, 11]
```

Something is missing

With a built-in list, we can do this:

```
l = [1, 2, 3]
print(len(l))  # outputs 3
for value in l:
    # outputs 'Value: 1' and so on
    print(f'Value: {value}')
```

This doesn’t work with our stack:

```
print(len(stack))  # throws TypeError: object of type 'Stack' has no len()
for value in stack:  # throws TypeError: 'Stack' object is not iterable
    print(f'Value: {value}')
```

Let’s implement __len__

```python
# inside Stack class
def __len__(self):
    length = 0
    current_node = self.head
    while current_node is not None:
        length += 1
        current_node = current_node.next
    return length
```

Now we can do *print(len(stack))*

Let's implement iterator

```python
# inside Stack class
class StackIterator:
    def __init__(self, head):
        self.current_node = head

    def __next__(self):
        if self.current_node is None:
            raise StopIteration

        value = self.current_node.data
        self.current_node = self.current_node.next
        return value

def __iter__(self):
    return self.StackIterator(self.head)
```

```python
for value in stack:
    print(f'Value: {value}')
```

This works now as well

Complexity

Operation	Complexity
construct	O(1)
push	O(1)
pop	O(1)
size	O(n)
iterate	O(n)

Array-backed stack

- This implementation uses the built-in Python list
- Python list is a dynamic array (meaning it resizes automatically)
- In this code, we’ve used the available methods (append and pop) to implement the logic we need
- However, using (almost) readily available code is of no interest to us as we aim to understand the internals

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

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

    def pop(self):
        return self.items.pop()

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

Array-backed stack

- We’ll use the array from **numpy** library instead
- numpy is a widely used library for numerical computing
- Can be installed as follows (in terminal): *pip install numpy*
- It contains highly optimized data structured and methods for linear algebra and other mathematical operations
- numpy arrays are **fixed size** and conceptually similar to the ones in C programming language

Let’s start

- Here we import the **numpy** library, assigning it **np** alias (for brevity)
- In constructor, we create an empty (filled with *None*) numpy array with a capacity for 1 element
- *n* contains the current number of elements in this stack

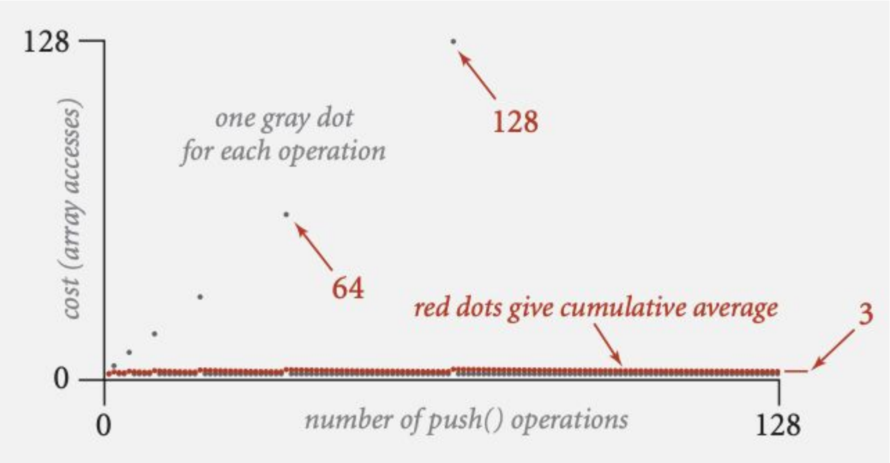
```python
import numpy as np

class ArrayStack:
    def __init__(self):
        self.arr = np.empty(1, dtype=object)
        self.n = 0

    def __len__(self):
        return self.n
```

Resizing the array

- We’ve allocated an array of size 1
- Therefore, we can push one element, and then we’ll need to resize this array (that is, increase its size)
- **Resizing operation.** Create a new array with a different size, and then copy the elements from the current array to the new one
- **Resizing is expensive.** If we increase the size of the array by 1 on every push, and decrease the size by 1 on every pop, it would be very slow: inserting first **N** items would take time proportional to $\mathbf{1 + 2 + \ldots + N \sim N^2 / 2}$
- Hence we need to ensure that resizing happens infrequently

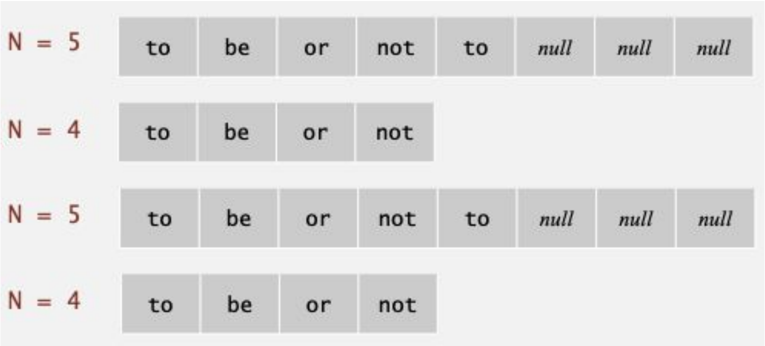
Growing array

- **If array is full, create a new array of twice the size, and copy items**
- As a result, inserting first **N** items takes time proportional to **N** (not $\mathbf{N^2}$)
- Cost of inserting first N items: **N (1 array access per push) + <cost for doubling the size> = N + (2 + 4 + 8 + … + N) ~ 3N**



Shrinking array

- We could halve size of array when its 50% full
- **Worst case:**
  - Push-pop-push-pop-... sequence when array is full
  - Each operation takes time proportional to N
- **Solution: halve size when array is 25% full**
- **Invariant.** Array is between 25% and 100% full



push/pop implementation

- Use n for indexing and keeping track of the current number of elements in the stack
- Grow/shrink to maintain the invariant: keep array between 25% and 100% full

```
def push(self, data):
    if self.n == len(self.arr):
        self.resize(2 * len(self.arr))
    self.arr[self.n] = data
    self.n += 1

def pop(self):
    if self.n == 0:
        return None

    self.n -= 1
    data = self.arr[self.n]
    self.arr[self.n] = None
    if self.n > 0 and self.n == len(self.arr) / 4:
        self.resize(int(len(self.arr) / 2))
    return data
```

resize implementation

- Create a new array with the given capacity
- Copy elements from the old array to the new one
- Reassign the field

```
def resize(self, capacity):
    new_arr = np.empty(capacity, dtype=object)
    i = 0
    while i < self.n:
        new_arr[i] = self.arr[i]
        i += 1
    self.arr = new_arr
```

Complexity

- **Amortized analysis.** Average running time per operation over a worst-case sequence of operations
- O(n) operations in the table below correspond to doubling and halving operations

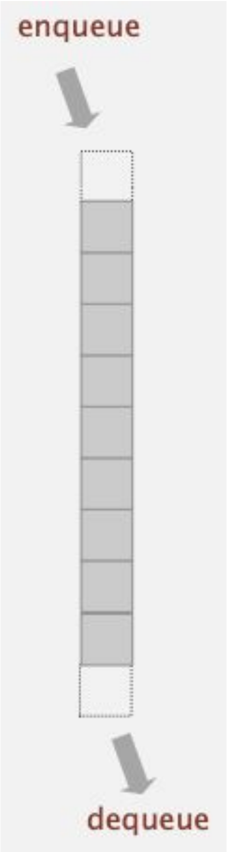
	best	worst	amortized
construct	O(1)	O(1)	O(1)
push	O(1)	O(n)	O(1)
pop	O(1)	O(n)	O(1)
size	O(1)	O(1)	O(1)

Resizing array vs. linked list

- Our API consists of a set of operations that a client can use: *push*, *pop*, __*len*__, and so on
- The semantics of these methods don’t depend on implementation details
- Can thus choose any implementation strategy
- **Linked list implementation:**
  - push/pop operations take **constant time in the worst case**
  - In practice, slightly slower and requires more memory (because of the need to store and deal with the references)
- **Array-based implementation:**
  - push/pop operations take **constant amortized time**
  - Usually faster in practice

Queue

- Queue implementation is very similar to stack
- We’ll only look at the one based on linked list
- Queue can also be implemented using resizable array



General structure

- Note that we now need two references: *head* and *tail*
- *head* points to the least recently added element, therefore the one that should be removed first
- Similarly, *tail* points to the most recently added element, to be removed last

```
class Queue:
    def __init__(self):
        self.head = None
        self.tail = None

    def is_empty(self):
        return self.head is None
```

enqueue/dequeue

- Very similar to stack, except for the two cases where we can whether the queue is empty
- These checks are needed to handle *head* and *tail* references properly

```
def enqueue(self, data):
    old_tail = self.tail
    self.tail = Node(data)
    if self.is_empty():
        self.head = self.tail
    else:
        old_tail.next = self.tail

def dequeue(self):
    if self.is_empty():
        return None

    head_node = self.head
    self.head = self.head.next
    if self.is_empty():
        self.tail = None
    return head_node.data
```

Array-backed queue

Use array *arr[]* to store items in queue.

- *enqueue*(): add new item at *arr[tail]*
- *dequeue*(): remove item from *arr[head]*
- Update *head* and *tail* modulo the capacity

Applications

- **Queue:**
  - Breadth-first search in graphs
  - Synchronization for input/output
  - Mostly used for queueing requests (servers, other data processing systems)
- **Stack:**
  - Parsing/evaluation of mathematical expressions (example: shunting-yard algorithm)
  - Function calls
  - Scheduling algorithms
  - Depth-first search in graphs
  - Can also be used for queueing requests (for example, when the goal is to first process the most recent requests)

Stack application example

- **Problem.** Given a string containing opening and closing braces, check if it represents a balanced expression or not
- Examples:
  - { [ ] { ( ) } } – balanced
  - [{ } { } ( ] – unbalanced
- **Solution:**
  - When an open parentheses is encountered push it onto the stack
  - When closed parenthesis is encountered, match it with the top of stack and pop it
  - If stack is empty at the end, return ‘balanced’. Otherwise, the expression is ‘unbalanced’

Balanced expressions. Code

```
def is_expression_balanced(expression):
    stack = Stack()
    opening_braces = ["[", "{", "("]
    closing_braces = ["]", "}", ")"]
    for char in expression:
        if char in opening_braces:
            stack.push(char)

        if char in closing_braces:
            if stack.is_empty():
                return False

            top_char = stack.pop()
            if opening_braces.index(top_char) != closing_braces.index(char):
                return False
    return stack.is_empty()


print(is_expression_balanced('{[]{()}}')) # True
print(is_expression_balanced('[{}{}(]')) # False
print(is_expression_balanced('(1+1){[2+4](3+5)}')) # True
print(is_expression_balanced('(1+1)[{2+4](3+5)}')) # False
```