CSC148 Week 2

Larry Zhang

Admin

Discussion board is up (link on the course website).

Outline for this week

- Abstract Data Type
- Stack
- Queue

Abstract Data Type (ADT)

What is an abstract data type (ADT)

- It's a **data type**, so
  - it stores **data**, and
  - it has **operations** on those data
- It is **abstract**, so
  - we ignore how it's implemented.
- We describe it from a **user's** point of view, NOT an **implementer's** POV.
- ADT allows us to directly describe the data used in our problems, independent of implementation details.
  - It makes problem-solving easier.

Real life example of ADT

ADT: Ice Cream Machine

Data stored: three flavours of ice cream

Operations:

get_chololate()

get_vanilla()

get_twist()

We don’t need to know the implementation details

What ADT did we learn in CSC108

Example: Dictionary (dict)

Data stored: a collection of key-value pairs

Operations:

- lookup value by key
- change value by key
- insert a new key-value pair
- deleting a key-value pair
- etc

Stack ADT

Let’s start by doing an OO analysis for it

> *A stack contains items of various sorts. New items are pushed on to the top of the stack, items may only be popped from the top of the stack. It’s a mistake to try to remove an item from an empty stack. We can tell how big a stack is, and what the top item is.*

Take a few minutes to identify the main noun, verb, and attributes of the main noun, to guide our class design. Remember to be flexible about alternate names and designs for the same class

Stack ADT

The order of adding and removing is Last In, First Out (LIFO).

Data stored: a sequence of objects

Operations:

- `push(o)`: Add a new item o to the top of the stack.
- `pop()`: Remove and return top item
- `is_empty()`: Test if stack is empty

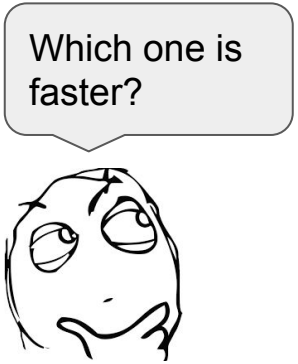
Could also have:

- `peek()`: Return top item (without removing it)
- `size()`: Return number of items in stack

DEMO
use a stack

- Start with empty stack []
- Push 5: [5]
- Push 8: $[5, 8]$
- Pop: [5] (and returns 8)
- Pop: [] (and returns 5)
- Pop: error!

How to implement a stack?



Option 1: Use a Python **list** L, add and remove at the **end** of the list.

- `push(o): L.append(o)`
- `pop(): return L.pop()`
- `is_empty(): return len(L) == 0`

Option 2: Use a Python **list** L, add and remove at the **beginning** of the list

- `push(o): L.insert(0, o)`
- `pop(): return L.pop(0)`

Option 3: Use a Python **dictionary** with integer keys 0,1,..., where the highest-numbered key is the most-recently pushed item

DEMO
compare different
implementations of stack

Summary of stack

Stack is one of the most useful ADT

- Keep track of pages visited in a browser tab (“Back Button”)
- Keep undo/redo history in a text editor or word processor
- Keep track of function calls in a running program (makes recursion possible)
    - the memory used by every process running on any computer has a stack segment in it (more about this in CSC209 and CSC369)
- Check for balanced parentheses (we will do an exercise on this)
- and lots more!

DEMO
function calls & stack

Queue ADT

Queue ADT

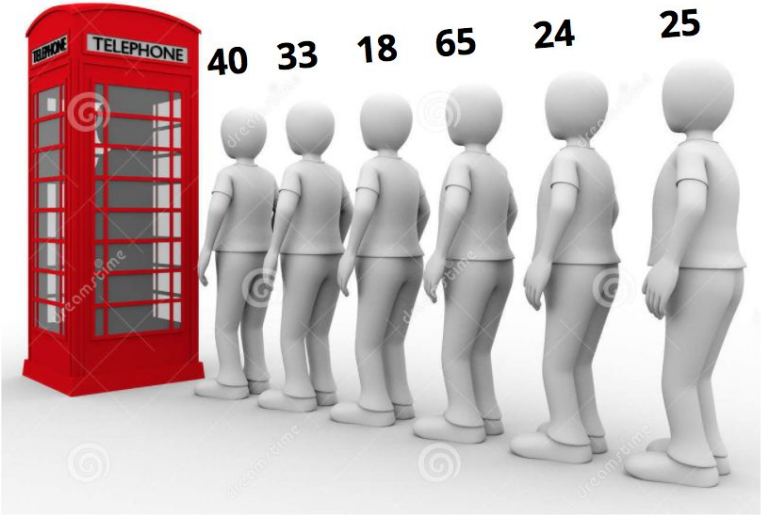
- Similar to Stack, a Queue also stores a sequence of objects.
- Different from Stack, a Queue's order for adding and removing is **First In, First out (FIFO)**.
- Operations:
  - **`enqueue()`**: Add o to the **end** of the queue
  - **`dequeue()`**: Remove and return object at the **front** of queue
  - **`front()`**: Return object at the front of queue
  - **`is_empty()`**: test if queue is empty
  - **`size()`**: return number of items in queue

A Queue

First in first serve

A Priority Queue!

Oldest person first



Priority Queue ADT

- A sequence of objects, again!
- Objects are removed in order of their priority
- Operations (max-priority queue):
  - **`insert(o)`:** Add **`o`** to the priority queue
  - **`extractMax()`:** Remove and return object with maximum value
  - **`max()`:** Return object with maximum value
  - **`is_empty(), size()`:** Same as previously
- There are **max**-priority queue and **min**-priority queues
  - **`extractMin() and min()`** instead

DEMO

`queue.py`

Home exercise

Implement the Priority Queue ADT using a list

and think about how to make it more efficient

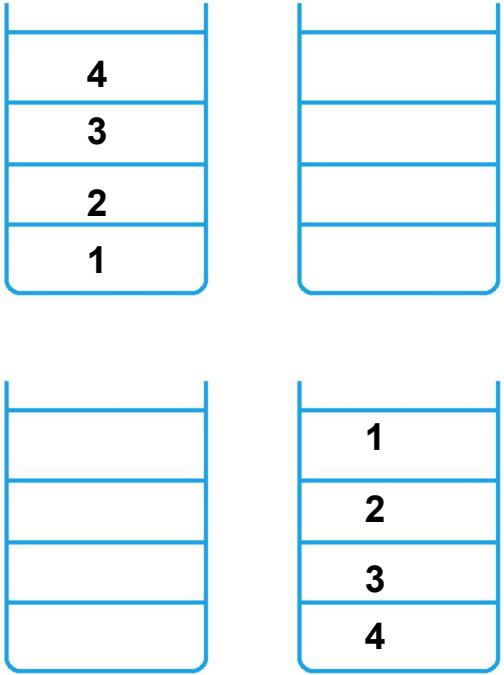
Problem Solving Time!

Reverse a stack

We want to reverse the order of the items in a stack. Below is an attempt.

```
def reverse_stack1(s: Stack):
    '''Reverse items on s.'''

    temp = Stack()
    while not s.is_empty():
        item = s.pop()
        temp.push(item)
    # s is empty
    while not temp.is_empty():
        s.push(temp.pop())
```



How to **slightly** modify this code to make it work?

DEMO

`reverse_stack.py`

ADT Puzzle

You're given a list of integers; you're goal is to transform the list into a new list according to the following rule:
Find the leftmost pair of consecutive numbers in the list whose values are x and $x + 1$, replace them by the single element whose value is $2x + 1$ and repeat the process using this new list. If no pair of integers satisfies this property, the process is complete.

- Example: list $[1, 2, 3, 4]$ is transformed first to $[3, 3, 4]$, and then to $[3, 7]$

```
Example: [1,2,4,5]  → [3,4,5] → [7,5]

Example: [4,2,1,2]  → [4,2,3] → [4,5] → [9]
```

Round 1: Naive Solution

Check the list over and over again until there is no more consecutive pairs.

```
def puzzle(lst):
    ''' (list of int) -> list of int

    Solve the puzzle.
    '''
    i = 0
    while i < len(lst) - 1:
        if lst[i] + 1 == lst[i + 1]:
            lst[i] = 2 * lst[i] + 1
            lst.pop(i + 1)
            i = 0  # go back to beginning
        else:
            i = i + 1
    return lst
```

What's the problem?

Consider this input:

`[32, 16, 8, 4, 2, 1, 2]`

Repeating the same work multiple times!

Not lazy enough!

Round 2: Use a stack

DEMO

See puz_track.txt

then see puz3.py

Balanced Brackets

Having balanced brackets means that

- Each opening bracket has a closing one of the same type, and
- brackets are properly nested

```
def is_balanced(s):
  '''(str) -> bool
  s is composed of characters in "()[]{}"
  Return True iff s is balanced.
  '''
  # implementation needed
```

- `([]){}`
  - `yes`
- `([)]`
  - `no`
- `(][)`
  - `no`
- `((()))`
  - `no`
- `[[[]]]`
  - `no`

Balanced Brackets

- When we see a closing bracket, we must have seen an earlier opening bracket with the same type.
  - Which one? `[[[][]]]`
  - The most recent one that has not been closed yet
- When we're finished, there must not be any remaining unmatched open parentheses
- Matching is LIFO
- Use a stack!

DEMO

`balanced_brackets.py`

Balanced Brackets

Solution idea:

- go through the input string, character by character
- if it is an opening bracket, push onto stack
- if it is a closing bracket, pop from stack, check if the type matches
    - if not matching return False
    - if nothing to pop (empty stack) return False
- After going through the whole string, the stack must be empty

Next week

- Exceptions