Basic Data Structures in Python

CS 280, Spring 2019
Basic Data Structure Taxonomy

Main basic data structures

- Lists (fast random access, sortable)
- Deque (like a list but fast access to front and back)
- Sets (collection of unique items)
- Dictionaries (a.k.a. maps, associate a set of keys to a set of values)

Python's Ethos

There should only be one obvious way of doing anything. Hence there aren't multiple implementations of list like in Java (ArrayList, LinkedList, etc.) so we won't worry too much about the distinction between ADT and implementation.
Python Lists

• Creating:

```python
empty = []                        # empty list
myList = [1, '2', 3]             # List literal
evens = [2 * x for x in range(10)] # list comprehension
```

(note: items need not have the same type)

• Typical List Operations

```python
list.append(item) # Add item to end of the list O(1)
list.insert(idx, item) # Add item at index idx. O(n)
list[idx] # Access item at index idx. O(1)
list[idx] = val # Replace object at idx with val. O(1)
list.pop() # Remove and return last item. O(1)
list.pop(idx) # Remove and return item at idx. O(n)
item in list # Tests whether item is in the list O(n)
len(list) # number of items in list
```
More advanced list operations

- Nice built-ins:

```python
list.sort()  # Sorts list in place
list.reverse()  # Reverses the list in place
list[::-1]  # Reverses and returns the list
list.extend(iterable)  # appends all items from iterable
# examples
list.extend([3, 4, 5])  # appends 3, 4, and 5.
list.extend(range(20))  # appends 0 through 19
list.remove(item)  # removes first instance of item
list1 + list2  # Concatenate the two lists
```
Using lists

- Pros:
  - Fast access to all items.
  - Fast append and pop (remove from end).
  - Fast sorting.

- Cons:
  - Slow to search for an item / find duplicates.
  - Slow to insert unless near the back.
  - Slow to remove an item unless near the back.
Deques

Part of the python `collections` library. You'll need to include it:

```python
from collections import deque
```

• Creating

```python
myDeque = deque() # Empty deque
myDeque = deque([1, 2, 3]) # Initialize from a list
```

• Typical operations

```python
myDeque.append(x) # Append to right side O(1)
myDeque.appendLeft(x) # Append to left side O(1)
myDeque.pop() # Remove and return rightmost item O(1)
myDeque.popLeft() # Remove and return leftmost item O(1)
myDeque.clear() # Empty out the deque
len(myDeque) # Number of items in the deque
myDeque[0] # peek at left-most item
myDeque[-1] # peek at right-most item
```
Deques: Additional Operations

```python
myDeque.extend(iterable)  # Extend on right side O(k)
myDeque.extendLeft(iter)  # Extend on the left side O(k)
myDeque.rotate()          # Equiv to d.appendleft(d.pop()) O(1)
myDeque.rotate(-1)        # Equiv to d.append(d.popLeft()) O(1)
myDeque.rotate(k)         # Like above, but k times. O(k)
myDeque.reverse()         # Reverse elements in place O(n)
```
Using Deques: As queues

```
d.append(x) # queue is now [x]
d.append(y) # queue is now [x, y]
d.append(z) # queue is now [x, y, z]
d.popLeft() # returns x, queue is [y, z]
```

We'll use this in tree and graph traversals later.
Using Deques: As stacks (LIFO)

Canonical problem: Check that parentheses are properly nested.

```python
from collections import deque

def checkString(str):
    d = deque()
    for c in str:
        if c == "(":
            d.append(")")
        elif c == "[":
            d.append("]")
        elif c == "{":
            d.append("}")
        elif c == ")" or c == "]" or c == "}":
            if len(d) == 0 or d[-1] != c:
                return False
            else:
                return True
    return len(d) == 0

checkString("[[x+2]\{3\}[[({9}\{9\})](x)]"]") # True
checkString("[[[]]]") # False
```
Using Deques: Pros / Cons

- Pros:
  - Implements both queue and stack interfaces.
  - Fast modification of both sides, list only has fast modification at the end.

- Cons:
  - Slow access to the middle. `myDeque[idx]` works in $O(n)$
Solved Problem

Backspace

Shortly before the programming contest started, Bjarki decided to update his computer. He didn’t notice anything strange until he started coding in his favorite editor, Bim (Bjarki IMproved). Usually when he’s writing in an editor and presses the *backspace* key a single character is erased to the left. But after the update pressing that key outputs the character `<`. He’s tested all the editors on his machine, Bmags, Neobim, bjedit, NoteBjad++ and Subjark Text, but they all seem to have the same problem. He doesn’t have time to search the web for a solution, and instead decides to temporarily circumvent the issue with a simple program.

Help Bjarki write a program that takes as input the string that was written in the text editor, and outputs the string as Bjarki intended to write it. You can assume that Bjarki never intended to write the character `<`, and that Bjarki never pressed the `backspace` key in an empty line.
**Input**

One line containing the string that was written in the text editor. The length of the string is at most $10^6$, and it will only contain lowercase letters from the English alphabet as well as the character `<.`

**Output**

One line containing the string as Bjarki intended to write it.

<table>
<thead>
<tr>
<th>Sample Input 1</th>
<th>Sample Output 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a&lt;bc&lt;</td>
<td>b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample Input 2</th>
<th>Sample Output 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>foss&lt;&lt;rritun</td>
<td>forritun</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample Input 3</th>
<th>Sample Output 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a&lt;a&lt;a&lt;aa&lt;&lt;</td>
<td></td>
</tr>
</tbody>
</table>
Python solution

```python
from collections import deque

line = input().strip()
chars = deque()

for i in range(len(line)):
    if line[i] == '<':
        chars.pop()
    else:
        chars.append(line[i])

print(''.join(chars))
```
Python Sets

Maintains a set of objects without storing duplicates.

- Creating

```python
s = set()  # Empty set
s = set([1, 2, 3])  # Initialize with values from list
```

- Typical Operations (All very efficient)

```python
s.add(item)  # Add item to the set.
s.remove(item)  # Remove item from the set.
item in s  # Test if item is in the set.
s.clear()
```
Sets: Additional operations

- Set operations (All efficient.)

```python
s1.union(s2)  # Union of the two sets.
s1.intersection(s2)  # Intersection of the two sets.
s1.issubset(s2)  # True if s1 is a subset of s2 (or =).
s1.issuperset(s2)  # True if s1 is a superset of s2 (or =)
s1.isdisjoint(s2)  # True if s1 and s2 are disjoint
```
Using sets

- **Pros:**
  - Very useful to maintain a collection of items you've seen while ignoring duplicates.
  - Very efficient set operations.

- **Cons:**
  - Order of iteration not guaranteed.

Commonly dealt with via converting to a list and sorting:

```python
s = set()
...
...
l = list(s)
l.sort()
for item in l:
  ...
```
Solved Problem

Boat Parts

Boating season is over for this year, and Theseus has parked his boat on land. Of course, the boat looks nothing like it did as of the beginning of the season; it never does. You see, Theseus is constantly looking for ways to improve his boat.

At every day of the boating season, Theseus bought exactly one type of item at his local supply store, and replaced the existing part on his boat with it. Now, as the season has ended, Theseus wonders what day he replaced all the parts from the previous season.

Input

The first line of the input consists of two space-separated integers $P$ and $N$, representing the number of parts the boat consists of, and the number of days in the boating season respectively. Then follows $N$ lines, each line has a single word $w_i$, the type of boat part that Theseus bought on day $i$.

Output

Output the day Theseus ended up replacing the last existing part from the previous season, or paradox avoided if Theseus never ended up replacing all the different parts.
Limits

- $1 \leq P \leq N \leq 1000$.
- Each word $w_i$ will consist only of the letters a–z and _ (underscore).
- Each word $w_i$ will be between 1 and 20 characters long.
- The number of distinct $w_i$s will be at most $P$.

Sample Input 1

```
3 5
left_oar
right_oar
left_oar
hull
right_oar
```

Sample Output 1

```
4
```

Sample Input 2

```
4 5
motor
hull
left_oar
hull
motor
```

Sample Output 2

```
paradox avoided
```
P, N = [int(x) for x in input().split()]
replaced_parts = set()

for i in range(1, N+1):
    replaced_parts.add(input())
    if len(replaced_parts) == P:
        print(i)
        exit()

print("paradox avoided")
Python Dictionaries

Maintains a mapping from a set of keys to a set of values. For example, employee IDs to names.

- Creating

```python
d = {}
d = dict()
d = {key1: val1, key2: val2, ...}  # O(n)
d = dict([[key1, val1], [key2, val2], ...])  # O(n)
```

- Typical Operations

```python
# O(1) time:
d[key] = value  # Add/overwrite key to value association
d[key]  # Returns the value associated to key
key in d  # True if there is a value associated to key

# O(n) time:
list(d.keys())  # List of all keys
list(d.values())  # List of all values
```
Source: interactivepython.org
Using dictionaries

Common use case:

Frequency-counting:

- keys are the things you want to count.
- values are the counts

```python
names = ['Bowers', 'Lam', 'Bowers', 'Bowers', 'Lam']
freq = dict([[name, 0] for name in set(names)])

for name in names:
    freq[name] = freq[name] + 1

for key in freq.keys():
    print(key + ' appears ' + str(freq[key]) + ' times.')```
Your problem

Babelfish

You have just moved from Waterloo to a big city. The people here speak an incomprehensible dialect of a foreign language. Fortunately, you have a dictionary to help you understand them.

Input

Input consists of up to 100 000 dictionary entries, followed by a blank line, followed by a message of up to 100 000 words. Each dictionary entry is a line containing an English word, followed by a space and a foreign language word. No foreign word appears more than once in the dictionary. The message is a sequence of words in the foreign language, one word on each line. Each word in the input is a non-empty sequence of at most 10 lowercase letters.

Output

Output is the message translated to English, one word per line. Foreign words not in the dictionary should be translated as “eh”.
Note: You need to be able to read arbitrary length input from \texttt{stdin} until EOF is reached. (We aren't given an \texttt{N} for the number of lines.) \texttt{input()} is not so good at this, but it can be achieved simply via:

```python
from sys import stdin  # Loads stdin as a file

# Two options:
lines = stdin.readlines()  # reads all lines into a list

# Or, if we want to strip off the newlines:
lines = [s.strip() for s in stdin.readlines()]
```