CS5001 Synthesis Assignment 3

Shang Xiao

May 2, 2022

Contents

1 Dictionaries & Sets

1.1 Associative Arrays

Explanation

Although lists and tuples can be beneficial in programming, it is not convenient to use them in some scenarios. We need to introduce other data structures to deal with some situation more efficiently. For example, when referring to something specific, we need to use associative arrays.

An associative array is an array with a unique indexing method. It can be indexed not only by integers but also by strings or other types of values.

With associative arrays, we can access other data more easily. Although it is not impossible to solve such problems using what we have learned earlier, using associative arrays can make our programming easier. If we use lists or tuples to link two sets of data together, then modifying one will affect the other, and there is no way to update both, or it is very tedious to do so. Now, if we can use a dictionary to maintain this data, the relationship between the two kinds of data is so tight that we can delete or modify one of the data without worrying about the above situation.

Once we understand the dictionary, we will be able to model various natural objects more accurately. We can create a dictionary that represents a person and then store as much information as you want: name, age, address, occupation, and any aspect to be described. We can also store any two kinds of related information, such as a series of words and their meanings, a series of people's names and their preferred numbers, and a series of mountains and their elevations.

Exercise

1. List the types of variables that can be used as a key in dictionary.
2. List the types of variables that can be used as a value in dictionary.
3. number = {"China", "US", "Japan", "Russia", "Germany"}; letter = "Q", "W", "E", "R", "T". Turn them into a dictionary with number being the key.

Solution

1. float, int, str
2. list, str, int, float
3.

```
number = {"China", "US", "Japan", "Russia", "Germany"}
letter = {"Q", "W", "E", "R", "T"}

number1 = ["China", "US", "Japan", "Russia", "Germany"]
```

```
letter2 = ["Q", "W", "E", "R", "T"]

key = {"China": "Q", "US": "W", "Japan": "E", "Russia": "R",
    "Germany": "T"}

key2 = dict(China = "Q", US = "W", Japan = "E", Russia = "R",
                    Germany = "T")

key3 = dict([["China", "Q"], ["US", "W"], ["Japan","E"],
                      ["Russia", "R"], ["Germany","T"]])

key4 = dict(zip(number, letter))

key5 = dict(zip(number1, letter2))

def main():
    print(key)
    print(key2)
    print(key3)
    print(key4)
    print(key5)

main()
```

1.2 Dictionaries

Explanation

Dictionary is a common data structure provided by Python, consisting of pairs of keys and values, separated by colons between the keys and values and commas between the items, with the whole dictionary enclosed by curly braces .

The format is as follow:

```
dic = {key1 : value1, key2 : value2 }
```

Dictionaries are also known as associative arrays or hash tables. The following are a few common ways dictionaries are created.

```
# approach1
dic1 = { 'Author' : 'Tom' , 'age' : 99 , 'sex' : 'man' }

# approach2
lst = [('Author', 'Tom'), ('age', 99), ('sex', 'man')]
dic2 = dict(lst)

# approach3
dic3 = dict( Author = 'Tom', age = 99, sex = 'man')
```

```
10
11 # approach4
12 list1 = ['Author', 'age', 'sex']
13 list2 = ['Tom', 99, 'man']
14 dic4 = dict(zip(list1, list2))
```

There are many other ways to create a dictionary, so I won't go into them here.

The dictionary is represented by the dict class, and we can use dir(dict) to see what methods the class contains by entering the command and seeing the following output:

```
1 methods = dir(dict)
2 print('methods = ',methods)
```

There are various methods and properties of dictionaries, and here we focus on the following methods.

Method	Description
dict.clear()	clear() is used to clear all elements (key-value pairs) in a dictionary. After clear() is executed on a dictionary, the dictionary becomes an empty dictionary.
dict.get()	get() is used to return the value of the specified key, that is, to get the value based on the key, to return None if the key does not exist, or to specify the return value.
dict.keys()	keys() returns all the keys of a dictionary.
dict.values()	values() returns all the values of a dictionary.
dict.items()	items() gets all the key-value pairs in the dictionary, and in general, the result can be transformed into a list for subsequent processing.
dict.pop()	pop() returns the value corresponding to the specified key and removes this key-value pair from the original dictionary.
dict.popitem()	popitem() deletes the last pair of keys and values in the dictionary.
dict.update(dic1)	update() update the dictionary, update the key-value pair of dict1 to dict; if the updated dictionary contains the corresponding key-value pair, then the original key-value team will be overwritten if the updated dictionary does not have the corresponding key-value pair, then the key-value couple will be added.

Table 1: Dictionaries Methods

Exercise

1. Create a dictionary so that English numbers can be mapped to Arabic numbers. Try to implement it in multiple ways.
2. Create a dictionary where key is the students' id and the value is the student's age.
3. Create a dictionary where key is the course name and the value is the time stamp of the beginning of the course.

Solution

1.
```
number = {'one':1, 'two':2, 'three':3, 'four':4, 'five':5,
    'six':6, 'seven':7, 'eight':8, 'nine':9}

number = dict(one = 1, two = 2)

lst = [('one',1),('two',2)]
number = dict(lst)
```

2.
```
number = {'Tom': 19, 'Jack': 17}
```

3.
```
number = {'Physics':385994334, 'Math':70173743487}
```

1.3 Iterating Over a Dictionary

Explanation

With a dictionary, how do we traverse this dictionary? I have summarized these methods of traversing the dictionary.

- Use the keys of the dictionary for traversal.

```
dict={"Tom":9,'Jack':87,'Wang':99}
for key in dict:
    print("key:%s value:%s"%(key,dict[key]))
```

- Use the enumerate() function to take out the serial number and key name of the key.

```
for index1,key in enumerate(dict):
    print("index:%s key: %s value:%s"%(index1,key,dict[key]))
```

- Use the dict.keys() method to get all the keys in the dictionary. (Similar to method 1).

```
for key in dict.keys():
    print("key: %s value:%s"%(key,dict[key]))
```

- Use the dict.values() method to get all the values in the dictionary. Unfortunately, you cannot get the dictionary name from the dictionary value.

```
for value in dict.values():
    print(value)
```

- Use the dict.items() method to get all the key-value pairs in the dictionary.

```
for key,value in dict.items():
    print(key,value)
```

Exercise

1. Given a dictionary, find the key whose value is the string "ok".
2. Given a dictionary where the key is id and the value is the age,find the person whose age is more than other people.
3. Given a dictionary where the key is the course name and the value is the time when the course is to be started,find the course which is the latest started.

Solution

1.
```
for key,value in dict.items():
    if value == "ok":
        print(key,value)
```

```
max_age = 0
for key,value in dict.items():
    if value > max_age:
        max_age = value
```

3.
```
time = 0
course = ""
for key,value in dict.items():
    if value < time:
        time = value
```

```
        course = key
```

1.4 Sets

Explanation

A set is an unordered sequence of non-duplicated elements. A set can be created using curly braces or the set() function. Note: To create an empty set we must use set() and not because is used to create an empty dictionary. Creation format:

```
parame = {value01,value02,...}
# Or
set(value)
```

Here are some common features I've found when working with set.

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # Here is a demonstration of the
    non-duplication function
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # Quickly determine if an element is
    in the set
True
>>> 'crabgrass' in basket
False

>>> # The following shows the operation between two sets.
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # The elements contained in set a
    but not in set b
{'r', 'd', 'b'}
>>> a | b                              # All elements contained in the set
    a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # The elements contained in both
    sets a and b
{'a', 'c'}
>>> a ^ b                              # Elements not contained in both a
    and b
{'r', 'd', 'b', 'm', 'z', 'l'}
```

There are many other functions on set that we can look up by checking the relevant documentation.

Exercise

Given two sets x and y, compute the intersection, union, difference and complement of the two sets.

Solution

```
>>> x = set('eleven')
>>> y = set('twelve')
>>> x,y
({'l', 'e', 'n', 'v'}, {'e', 'v', 'l', 't', 'w'})
>>> x & y #intersection
{'l', 'e', 'v'}
>>> x | y #union
{'e', 'v', 'n', 'l', 't', 'w'}
>>> x - y #difference
{'n'}
>>> y -x  #difference
{'t', 'w'}
>>> x ^ y #complement
{'t', 'n', 'w'}
>>> y ^ x #complement
{'w', 'n', 't'}
>>>
```

2 Stacks and Queues

2.1 Data Structures vs ADT

Explanation

Unlike data structures, ADTs do not tell you HOW to implement certain tasks, rather they tell you what kind of things and opertions can be completed. ADTs are not implementation and can be immediately used. Think ADTs as outlines for data structures can be the best way to distinguish these two.

Another difference between these two is ADTs do not care about actual data that we will be using in it, the data type can rather be strings, objects, or something else, we only care about these data types when we get to data structures.

Abstract Data Type (ADT) is a mathematical model of a specific class of data structures with similar behavior in computer science or data types with similar semantics for one or more programming languages. Abstract data types are defined indirectly through the executable operations and the mathematical constraints on the effects of those operations.

Below are some data structure examples we have learned:

Data Structures	Description
List	A type of data structure that is indexable, ordered list which supports enforcing strict order
Dictionary	An abstract data structure that supports key-value pairs
Set	A collection of abstract data that will de-duplicate the internal elements

Table 2: Previous Data Structures

Abstract data types (ADTs) are purely theoretical entities used to simplify the description of abstract algorithms, classify and evaluate data structures, and formally describe the type system of programming languages. An ADT can be implemented with a specific data type or data structure and there are many implementations in many programming languages.

Exercise

1. Explain the relationship between ADT and specific data structures.
2. Write a tree ADT

Solution

1. ADT defines how to store data, call data, extract data, and be able to make specific selections about data. The data structure specifically implements calls to the data.
2.

```
#Graph
class Graph:
    def __init__(self):
        self.vertList={}
        self.numVertices=0
    #New Plus Vertex
    def addVertex(self,key):
        self.numVertices=self.numVertices+1# Add 1 to the number of
            vertices
        newVertex=Vertex(key)
        self.vertList[key]=newVertex
        return newVertex
    #Find vertices by key
    def getVertex(self,n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None
    def __contains__(self,n):
        return n in self.vertList
```

```
20      #Add side
21      def addEdge(self,f,t,cost=0):
22          if f not in self.vertList:# Non-existent vertices are added
                first
23              nv=self.addVertex(f)
24          if t not in self.vertList:# Non-existent vertices are added
                first
25              nv=self.addVertex(t)
26          self.vertList[f].addNeighbor(self.vertList[t],cost)#
                Adjacent edges are added by calling the method of the
                starting vertex
27      def getVertices(self):
28          return self.vertList.keys()
29      def __iter__(self):
30          return iter(self.vertList.values())
```

2.2 Stack ADT

Explanation

A stack is a linear table in computer science that is restricted to insert or delete operations only at the end of the table. A stack is an ADT that stores data on a last-in, first-out basis, with the first-in data being pressed into the bottom of the stack and the last date at the top, and the information being popped out from the top of the stack when it needs to be read. Stacks are unique linear tables that can only be inserted and deleted at one end. When stacking items in a bucket, the things that come in first are pressed to the bottom and then stacked to the top. When you take away, you can only assume from the top one piece at a time. Reading and fetching are done at the top, and the bottom is usually left untouched. A stack is a data structure similar to a bucket stacking items, where one end for deletion and insertion is called the top of the stack, and the other end is called the bottom. Insertions are generally called in-stock, and deletions are called out-stack. A stack is also known as a last-in, first-out table.

Because of the stack's first-in, last-out nature, we can apply it to recursive calls. In recursion, the first layer of data is stored at the top of the stack, and then the next layer of information is recursively put on the pile again, and so on. Eventually, when we backtrack, we can retrieve the memory data of the original recursive call in turn.

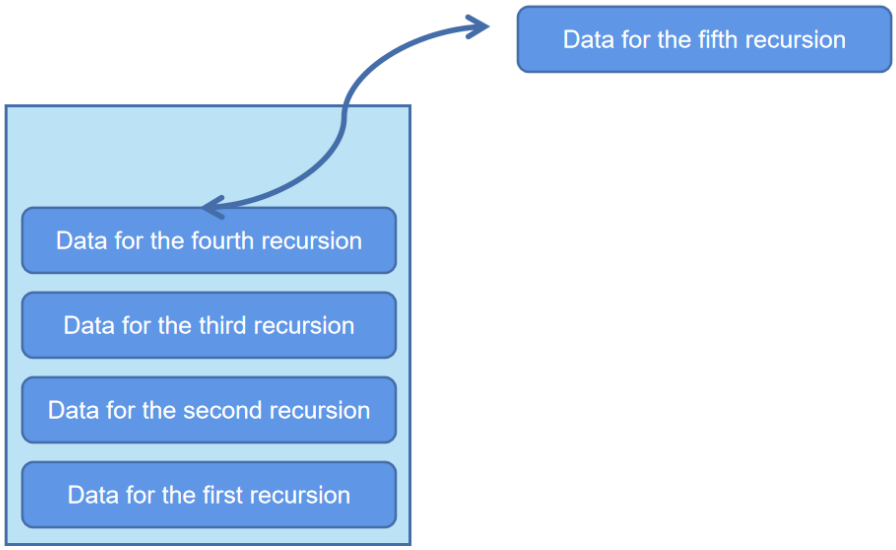

Figure 1: Stack Memory in Recursion

Exercise

1. Use the stack to find the element of an array that is larger than the current number and closest to the current position.

2. Given a string s containing only '(', ')', '', '', '[', ']', determine whether the string is valid.
   Valid strings need to satisfy:

   - The left parentheses must be closed with right parentheses of the same type.
   - The left parentheses must be closed in the correct order.

Solution

1. In this problem, we can use a stack to maintain a sequence of data within this stack that satisfies decreasing, so that we can find the nearest one that is larger than the current number by popping out of the stack. Taking the sequence 1, 3, 7, 2 as an example, the following figure demonstrates the action we perform when we reach 2.

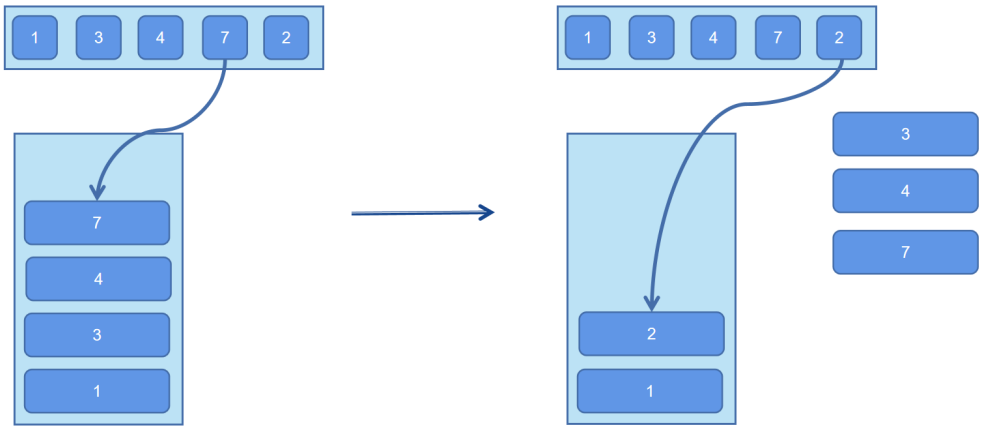

Figure 2: The Nearest Bigger Number

2. Determining the validity of parentheses can be solved using the data structure "stack".

   We iterate over a given string s. When we encounter a left bracket, we expect a right bracket of the same type to close it in subsequent iterations. Since the later encountered left bracket has to be closed first, we can put this left bracket on the top of the stack.

   When we encounter a right bracket, we need to close a left bracket of the same type. At this point, we can remove the left bracket at the top of the stack and determine if they are brackets of the same type. If they are not of the same type, or if there is no left bracket on the stack, then the string ss is invalid and False is returned. To quickly determine the type of the parentheses, we can use a hash table to store each type of parentheses. The keys of the hash table are the right brackets and the values are the left brackets of the same type.

   At the end of the traversal, if there are no left brackets in the stack, it means we close all left brackets in the string ss and return True, otherwise return FalseFalse.

   Notice that the length of a valid string must be even, so if the length of the string is odd, we can return False directly and save the subsequent traversal judgment process.

2.3 Stack Operations

Explanation

The stack is a linear data structure that stores data in a first-in-first-out or last-in-first-out manner. The insertion and deletion of data in the stack are performed at the top of the stack, and common stack functions include:

Operation	Description
pop()	Pulls off top element of the stack
push()	Adding elements on top of the stack
empty()	Determine if the stack is empty
size()	Return the length of the stack
top()	View top-of-stack elements
peek()	Take a look at what is currently at the top of the stack without taking it off
count()	Sum up how many things are on the stack
dump()	Print out all current elements that are on the stack

Table 3: Stack ADT Methods

Exercise

Given two sequences of pushed and popped, each with non-duplicated values returns true only if they could result from a series of forced push and popped operations on an initially empty stack; otherwise, it returns false.

Solution

Push each number in the pushed queue onto the stack, check if it is the next value to be popped in the popped sequence, and pop it out.

Finally, check that not all the values that should be popped are popped out.

```
def solve( pushed, popped):
    j = 0
    stk = []
    for x in pushed:
        stk.append(x)

        while stk and j < len(popped) and stk[-1] == popped[j]:
            stk.pop()
            j += 1

    return j == len(popped)
```

2.4 Queue ADT

Explanation

A queue is a special kind of linear table, unique in that it only allows deletion operations at the front end of the table and insertion operations at the back end of the table. Like a stack, a queue is a linear table with restricted functions. The end of the insertion operation is the end of the queue, and the end of the deletion operation is the head of the queue. When there are no elements in the queue, it is called an empty queue.

The data elements of a queue are also called queue elements. Inserting a queue element into a queue is called entering the queue, and removing a queue element from a queue is reached leaving the queue. Because the queue only allows insertion at one end and deletion at the other end, only the earliest element to enter the queue can be removed first, so the queue is also known as the first-in, first-out linear table.

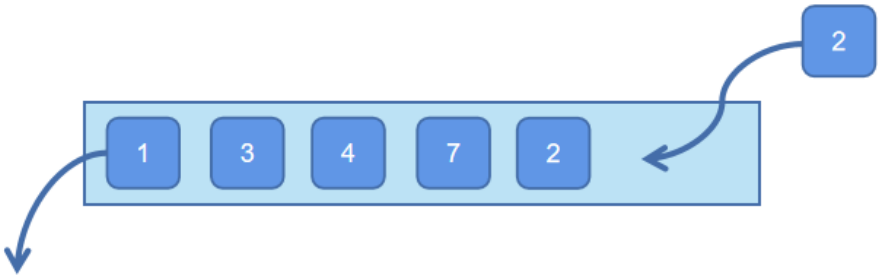

Figure 3: Queue ADT

Exercise

1. Give a scenario for the use of a queue and explain why.
2. Determine the correctness or incorrectness of the following statements: The one who goes out in the queue is always the one who stays in the queue for the shortest time.

Solution

1. The problem of waiting in line for everything in ordinary life, such as borrowing books, buying breakfast, applying for the dole, etc. Because no one likes to be cut in line by others.

2. False. Because the queue always pop the value in the front of the queue which stays the longest in the items.

2.5 Queue Operations

Explanation

The queue can be stored as an array Q[1...m], and the upper bound m of the array is the maximum capacity allowed for the queue. Two pointers are needed in the queue operation: head, the head pointer, which points to the actual head element; tail, the tail pointer, which points to the next position of the actual tail element. In general, the initial value of the two pointers is set to 0 when the queue is empty and there are no elements. Q(i) i=3,4,5,6,7,8. head=2, tail=8. the number of elements in the queue is: L=tail-head. to get the head element out of the queue, add 1 to the head pointer. head=head+1. the head pointer is moved up one position and points to Q(3). The head pointer moves up one position to Q(3), which means Q(3) is out of the queue. If you want to add a new element to the queue, you need to move the tail pointer up by one position. That is, tail=tail+1, which means Q(9) is in the queue.

Of course, just like stack, python provides already implemented queues for us to use. In python, we can use queue for its implementation. We can declare a queue in the following way:

```
1 q = queue.Queue()
```

And the following methods:

Operation	Description
enqueue()	Similar to push(), enters an element into the line
dequeue()	Similar to pop(), gets the earliest enqueue element out of the line
is_full()	Check if the queue has space
is_empty()	Check if the queue has something
peek()	Take a look at what is currently at the front of the line
count()	Count the number of elements in the line
put()	Adding tasks to the queue
get()	Get the tasks in the queue
qsize()	Get the number of tasks in the queue
task_done()	Queue tasks are executed to completion

Table 4: Queue ADT Methods

Exercise

Given an array of nums and a sliding window of size k, find the maximum value in all sliding windows.

Solution

When the sliding window moves to the right, as long as i is still in the window, then j must also still be in the window, which is guaranteed by the fact that i is to the left of j. Therefore, due to *nums*[j]nums[j], *nums*[i] must not be the maximum value in the sliding window anymore, and we can remove *nums*[i] permanently.

So we can use a queue to store all the subscripts that have not been removed yet. In the queue, these subscripts are stored in order from smallest to largest, and their corresponding values in the array *nums* are strictly monotonically decreasing. Because if there are two adjacent subscripts in the queue that correspond to equal or increasing values, then making the former i and the last j corresponds to the case described above, i.e., *nums*[i] will be removed, which creates a contradiction.

When the sliding window moves to the right, we need to put a new element into the queue. To maintain the nature of the queue, we keep comparing the new element with the element at the end of the queue. If the former is greater than or equal to the latter, then the element at the end of the queue can be permanently removed, and we pop it out of the queue. We need to keep doing this operation until the queue is empty or the new element is smaller than the element at the end of the queue.

Since the elements corresponding to the subscripts in the queue are strictly monotonically decreasing, the element corresponding to the subscript at the top of the queue is the maximum value in the sliding window at this point. However, as in method 1, the maximum value may be to the left of the left boundary of the sliding window, and as the window moves to the right, it can never appear in the sliding window. Therefore we also need to keep popping elements from the head of the queue until the head element is in the window.

```python
import collections
from typing import List


def solve(lst: List[int], k: int) -> List[int]:
    n = len(lst)
    q = collections.deque()
    for i in range(k):
        while q and lst[i] >= lst[q[-1]]:
            q.pop()
        q.append(i)

    rlt = [lst[q[0]]]
    for i in range(k, n):
```

```
        while q and lst[i] >= lst[q[-1]]:
            q.pop()
        q.append(i)
        while q[0] <= i - k:
            q.popleft()
        rlt.append(lst[q[0]])

    return rlt
```

2.6 Circular Buffers in Queues

Explanation

To make full use of the vector space, a way to overcome the overflow phenomenon is to consider the vector space as a circle with the beginning and the end connected, and call this vector a circular vector. The queue stored in it is called Circular Queue. A circular queue is a sequential queue connected at the beginning and end, and the table storing the elements of the queue is logically viewed as a ring, which becomes a circular queue.

The circular queue is a queue storage space of the last position around the first position, forming a logical ring space for the queue cycle. In the circular queue structure, when the last location of the storage space has been used and then to enter the queue operation, only the first location of the storage space is free, then the elements can be added to the first location, that is, the first location of the storage space as the end of the queue. The circular queue can prevent the occurrence of pseudo overflow more simply, but the queue size is fixed.

In a circular queue, there is front=rear when the queue is empty and front=rear when all queue spaces are fully occupied. to distinguish between these two cases, it is specified that a circular queue can have at most MaxSize - 1 queue elements, and the queue is full when there is only one empty storage unit left in the circular queue. Thus, the condition for a queue to be judged empty is front=rear, while the condition for a queue to be judged full is front=(rear+1)%MaxSize.

We can represent it graphically.

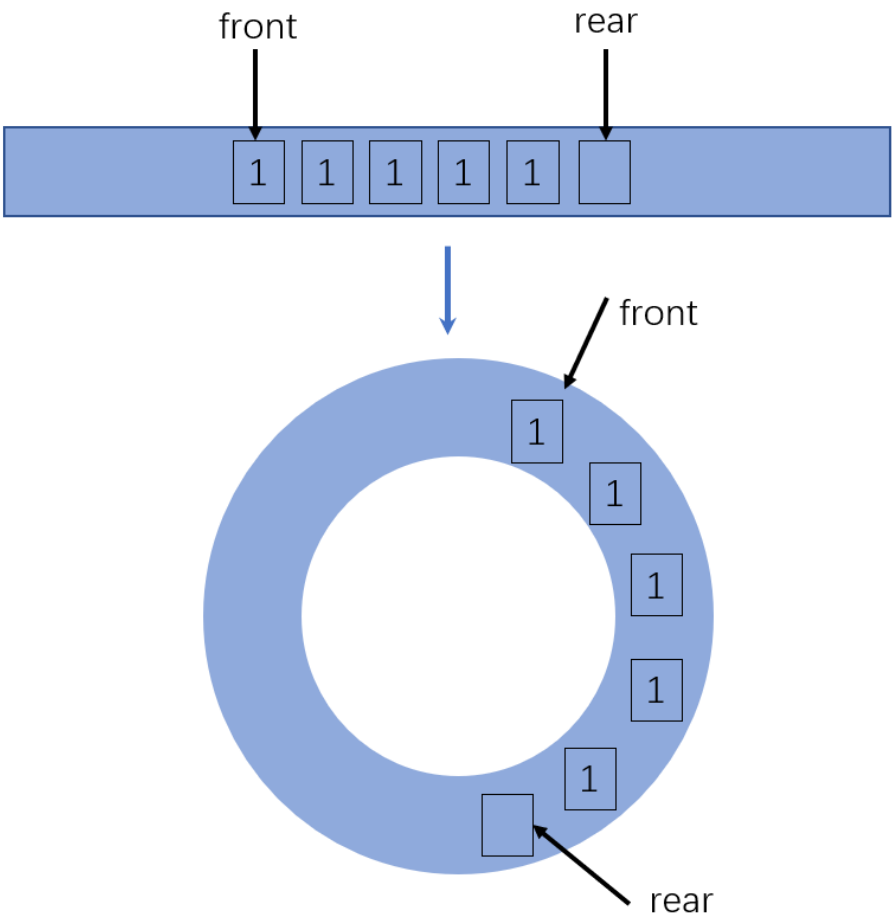

Figure 4: Circular Buffers in Queues

Exercise

1. Explain how to determine how a circular queue determines whether the current queue is full or not.

2. How to determine if a circular queue is empty.

Solution

1. Since the back pointer points to empty data, there is no way to tell if the queue is empty or full when ah two pointers are summed. To distinguish whether the stack is empty or not, we leave one position empty for the

back pointer so that we can determine whether the current queue is empty by the above condition.

```
def full(self):
    return (self.rear + 1) % len(self.data) == self.front
```

2.

```
def empty(self):
    return self.rear == self.front
```