knopp_daniel_final_exam_Q2

December 15, 2023

1 Final Exam Part B: Question 2

Daniel Knopp

Comprehensive guide to CRUD operations of common data structures in Python.

1.0.1 Section 1: Sets

Sets Intro

Sets are a very useful data structure in Python when dealing with unique groupings of items. Sets are stored in memory as dynamic arrays, but with a key improvement over that standard datatype: hash funciton indexing. Sets are extremely fast, on average, with data retrieval due to their usage of hash tables. Hash tables are special funcitons which translate the data being stored from it's original data to an index that is used to directly loop up a specific element in the dynamic array. This generally means that read operations have a time complexity of O(1), but is not always the case. When dealing with hash tables, there is always the issue of if a new value would end up being stored in the same dynamic array index due to the output of the hash function returning the same value for different inputs. This event is known as a collision. Collisions increase the time complexity of the data structure operations and are typically stored as linked lists starting from the index output by the hash function. These linked lists must then be traveled sequentially until the desired element is found. In the code below I will give an overview of set definition and usage.

Practical Example: - One example of how you could use sets in the real-world would be to keep track of different levels of access within a company network or system. For exmaple, if you had a set of users who had Admin privilages, read access, and write access to files on a file system then you could use sets to quickly check if a particular user is in one of those groups and should have a certain level of access to a file with restricted privilages. This data structure would be better than something like a list because checking if an item is in a set is typically a constant time operation O(1), while checking if an item is in a list can be up to O(n) time complexity.

[ ]:
```
# Sets can be both defined with curly braces or with the set() function, first␣
 ↪let's take a look at defining an empty set:
# Time  Complexity: O(1)
# Space Complexity: O(1)
my_set = set()
print(my_set)
```

```
set()
```

[ ]:
```
# Alternatively, we can define a set with values in it:
# Time  Complexity: O(n) -> n is length of set
# Space Complexity: O(n) -> n is length of set
my_set = {1, 2, 3, 4, 5}
print(my_set)
```

```
{1, 2, 3, 4, 5}
```

[ ]:
```
# To add an element to a set, we can use the built in add() method:
# Time  Complexity: O(1)
# Space Complexity: O(1)
my_set.add(6)
print(my_set)
```

```
{1, 2, 3, 4, 5, 6}
```

[ ]:
```
# Note, since sets only contain unique elements, adding an element that already␣
↪exists in the set will not change the set
# Time  Complexity: O(1)
# Space Complexity: O(1)
my_set.add(6)
print(my_set)
```

```
{1, 2, 3, 4, 5, 6}
```

[ ]:
```
# To remove an element from a set, we can use the built in remove() method:
# Time  Complexity: O(1)
# Space Complexity: O(1)
my_set.remove(6)
print(my_set)
```

```
{1, 2, 3, 4, 5}
```

[ ]:
```
# Often we want to know if a specific element is in a set, we can do this using␣
↪the in keyword as a conditional statement:
# Time  Complexity: O(1)
# Space Complexity: O(1)
print(6 in my_set)
```

```
False
```

[ ]:
```
# There are other set operations we can do, such as union, intersection, and␣
↪difference. Let's take a look at these by first defining two sets:
# Time  Complexity: O(n) -> n is length of set
# Space Complexity: O(n) -> n is length of set
set_a = {1, 2, 3, 4, 5}
set_b = {4, 5, 6, 7, 8}

# Union
# Time  Complexity: O(n) -> n is length of largest set
```

```
# Space Complexity: O(n) -> n is length of largest set
print(f'{"Union":>12} of set_a and set_b is: {set_a | set_b}')

# Intersection
# Time  Complexity: O(n) -> n is length of smallest set
# Space Complexity: O(n) -> n is length of smallest set
print(f'{"Intersection":>12} of set_a and set_b is: {set_a & set_b}')

# Difference
# Time  Complexity: O(n) -> n is length of first set
# Space Complexity: O(n) -> n is length of first set
print(f'{"Difference":>12} of set_a and set_b is: {set_a - set_b}')
```

```
      Union of set_a and set_b is: {1, 2, 3, 4, 5, 6, 7, 8}
Intersection of set_a and set_b is: {4, 5}
  Difference of set_a and set_b is: {1, 2, 3}
```

1.0.2 Practice Exercises: Sets

Exercise 1: Write a Python function that takes a list as an input and uses a set to remove any duplicate elements, returning a list of unique values as output. (Hint: a fundamental property of sets will be very helpful here)

Exercise 2: Write a Python function that takes multiple lists as inputs and uses a set to find all common elements between the lists, returning a list of the common values as output. (Hint: set operations provide good tools for finding common elements between multiple sets)

1.0.3 Section 2: Tuples

Tuples Intro

Tuples are another useful data structure available in Python. Tuples are similar to static arrays in that their size cannot be changed, but they are also immutable objects - meaning that once they are defined they cannot be edited. Often, tuples are used to return multiple values in the output of a function. Another example of a useful implementation of a tuple is to use the tuple as a key to a dictionary. Since tuples are immutable, they are compatible with dictionary keys and often can simply certain dictionary structures. Dictionaries will be discussed in more detail in Section 3.

Practical Example: - One example of where you might use Tuples in the real-world would be when working with GPS coordinates. GPS coordinates that represent locations on Earth always come in pairs of 2 values. If, for example, you wanted to create a lookup dictioary of places based on their GPS coordinates, you could use the latitude and longitude coordinates stored in the form of a Tuple as a key to the dicitonary (more on dictionaries below). This is because Tuples are immutable objects, and this allows them to be used as keys in dictionaries even though the Tuples may contain more than one element. Other data structures such as lists would be clunky for storing and comparing GPS coordinates as Tuples allow for direct logical comparison to see if 2 sets of coordinates are identical (i.e. (1, 2) == (1, 2) -> True).

[ ]:
```
# Tuples can be defined with parentheses or with the tuple() function, first␣
↪let's take a look at defining an empty tuple:
```

```
# Time  Complexity: O(1)
# Space Complexity: O(1)
my_tuple = ()
print(my_tuple)
```

```
()
```

[ ]:
```
# Alternatively, we can define a tuple with values in it:
# Time  Complexity: O(n) -> n is length of tuple
# Space Complexity: O(n) -> n is length of tuple
my_tuple = (1, 2, 3, 4, 5)
print(my_tuple)
```

```
(1, 2, 3, 4, 5)
```

[ ]:
```
# To demonstrate the immutability of tuples, let's try to change the first␣
 ↪element of the tuple:
# Time  Complexity: O(1)
# Space Complexity: O(1)
try:
    my_tuple[0] = 6
except TypeError as e:
    print(f'Error: {e}')
```

```
Error: 'tuple' object does not support item assignment
```

[ ]:
```
# To access elements of the tuple, we can use an index-based syntax (with␣
 ↪zero-based indexing):
# Time  Complexity: O(1)
# Space Complexity: O(1)
print(my_tuple[2]) # print the 3rd element of the tuple
```

```
3
```

[ ]:
```
# It's also possible to access a range of elements in a tuple using the slice␣
 ↪syntax:
# Time  Complexity: O(x) -> x is the size of the slize
# Space Complexity: O(x) -> x is the size of the slize
print(my_tuple[2:4]) # print the 3rd and 4th elements of the tuple
```

```
(3, 4)
```

[ ]:
```
# Sometimes it's useful to combine multiple tuples together by concetenate them:
# Time  Complexity: O(n + m) -> n is length of original tuple and m the length␣
 ↪of the tuple being added
# Space Complexity: O(n + m) -> n is length of original tuple and m the length␣
 ↪of the tuple being added
my_tuple = my_tuple + (6, 7, 8)
print(my_tuple)
```

```
(1, 2, 3, 4, 5, 6, 7, 8)
```

```
[ ]: # If we need to unpack a tuple into multiple variables, we can do so using the␣
      ↪following syntax:
     # Time  Complexity: O(1)
     # Space Complexity: O(1)
     my_tuple = (1, 2, 3)
     a, b, c = my_tuple
     print(f'Unpacked tuple: {my_tuple} into variables: a={a}, b={b}, c={c} using␣
      ↪the code: a, b, c = my_tuple')
```

```
Unpacked tuple: (1, 2, 3) into variables: a=1, b=2, c=3 using the code: a, b, c
= my_tuple
```

1.0.4 Practice Exercises: Tuples

Exercise 1: Write a Python function that takes in the dimensions of a rectangle stored as a Tuple (width, height) and computes the area, returning the area of the rectangle as an output. (Suggestion: unpack the elements of the tuple before trying to compute the area)

Exercise 2: Write a Python function that takes in 2 lists of equal length and combines the lists element-wise into a list of tuples, returning the list of tuples as an output. (Hint: check out information about the build-in 'zip' function)

1.0.5 Section 3: Dictionaries

Dictionaries Intro

Dictionaries are very useful in Python when you need to store lookup information, but are more generally used any time you need to store data in a key, value pair format. Dictionaries are also very fast, on average, data structures and use hashing functions in a similar way as sets. To lookup data from a dictionary, one must specify the key that is assosiated with the values stored within. This key is translated into a memmory address similar to what happens with sets and then in that memory location there is also a pointer which directs the system to the other region of memory where the data structure that matches the key resides. This data structure can also suffer from performance loss in the same way as sets - if your hashing function creates a lot of collusions for the keys you are using to store the data, then when you go to look up something by a key with many collisions the system will have to iterate over the collided items until it gets the desired one.

Practical Example: - A very common practical example for using dictionaries is creating lookup tables. For example, you could create a contact dictionary of phone numbers where the dictionary keys are contact names and the corresponding value pairs are phone numbers. Then, if you wanted to call a friend, you can very quickly lookup the phone number using the name as the key. These are very useful for accessing information quickly from large datasets. Because the key, value pairs are stored using a hash table, then typically have close to constant-time lookup operations. Compare that to using something like a list that could have O(n) time complexity.

```
[ ]: # Dictionaries can be defined with curly braces or with the dict() function,␣
      ↪first let's take a look at defining an empty dictionary:
     # Time  Complexity: O(1)
```

```
# Space Complexity: O(1)
my_dict = {}
print(my_dict)
```

```
{}
```

[ ]:
```
# Alternatively, we can define a dictionary with key, value pairs in it:
# Time  Complexity: O(n) -> n is the number of key, value pairs
# Space Complexity: O(n) -> n is the number of key, value pairs
my_dict = {'a': 1, 'b': 2, 'c': 3}
print(my_dict)
```

```
{'a': 1, 'b': 2, 'c': 3}
```

[ ]:
```
# To add a new key, value pair to the dictionary, we can use the following␣
 ↪syntax:
# Time  Complexity: O(1)
# Space Complexity: O(1)
my_dict['d'] = 4 # where d is the key and 4 is the corresponding value
print(my_dict)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

[ ]:
```
# It is also possible to store any type of object as a value in a dictionary,␣
 ↪including other dictionaries:
# Time  Complexity: O(m)     -> m is the number of key, value pairs in the␣
 ↪nested dictionary
# Space Complexity: O(n + m) -> m is the number of key, value pairs in the␣
 ↪nested dictionary while n is the number in the original dictionary
my_dict['e'] = {'x': 1, 'y': 2, 'z': 3}
print(my_dict)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': {'x': 1, 'y': 2, 'z': 3}}
```

[ ]:
```
# To access a value in a dictionary, we can use the following syntax:
# Time  Complexity: O(1)
# Space Complexity: O(1)
print(my_dict['a'])
```

```
1
```

[ ]:
```
# This format continues to work even if the value is another dictionary as in␣
 ↪the case with the value for the key 'e':
# Time  Complexity: O(1)
# Space Complexity: O(1)
print(my_dict['e'])
```

```
{'x': 1, 'y': 2, 'z': 3}
```

[ ]:
```
# And we can access a value in the nested dictionary using the same syntax:
# Time  Complexity: O(1)
# Space Complexity: O(1)
print(my_dict['e']['x'])
```

```
1
```

[ ]:
```
# Modifying values stored in the dictionary is also possible and we use a␣
 ↪similar syntax to adding a new key, value pair:
# Time  Complexity: O(1)
# Space Complexity: O(1)
my_dict['a'] = 5
print(my_dict)
```

```
{'a': 5, 'b': 2, 'c': 3, 'd': 4, 'e': {'x': 1, 'y': 2, 'z': 3}}
```

[ ]:
```
# To remove a key, value pair from a dictionary, we can use the built in del␣
 ↪keyword:
# Time  Complexity: O(1)
# Space Complexity: O(1)
del my_dict['e']
print(my_dict)
```

```
{'a': 5, 'b': 2, 'c': 3, 'd': 4}
```

[ ]:
```
# Often it is useful to loop over the key, value pairs within a dicitonary to␣
 ↪do something with the data, we can do this using the items() method:
# Time  Complexity: O(n) -> n is the number of key, value pairs
# Space Complexity: O(n) -> n is the number of key, value pairs
for key, value in my_dict.items():
    print(f'Key: {key}, Value: {value}')
```

```
Key: a, Value: 5
Key: b, Value: 2
Key: c, Value: 3
Key: d, Value: 4
```

[ ]:
```
# We can also just loop over only the keys or only the values using the keys()␣
 ↪and values() methods respectively:
# Time  Complexity: O(n) -> n is the number of key, value pairs
# Space Complexity: O(n) -> n is the number of key, value pairs
for key in my_dict.keys():
    print(f'Key: {key}')
for value in my_dict.values():
    print(f'Value: {value}')
```

```
Key: a
Key: b
Key: c
Key: d
```

```
Value: 5
Value: 2
Value: 3
Value: 4
```

1.0.6 Practice Exercises: Dictionaries

Exercise 1: Create a dictionary that uses X, Y coordinates stored as Tuples for keys and names of interesting locations on a fictional 2-D map as the corresponding values. Write a Python function that takes in this lookup dictionary and a randomly generated X, Y coordinate as inputs and finds the point of interest in the dictionary nearest to the randomly generated point. Return the point of interest and the distance between as a Tuple.

Exercise 2: Write a Python function that creates a contact lookup table where each key represents the name of a contact, but the corresponding value is itself another dictionary that contains key, value pairs for phone number, email address, and street address. Return the nested dictionary as an output.

1.0.7 Section 4: Lists

Lists Intro

Lists are arguably one of the simplest but also one of the most commonly used data structures in Python. Lists take the place of an array which would be used in many other programming languages, and they are stored in momory similar to how a dynamic array is stored. A certain chunk of continuous memory is reserved, and elements in the list are stored sequentially in that chunk of memory. Since you can store both immutable and mutable objects in each element of the list, often the list stores only a pointer to the data at each element instead of the data object itself. They are not as space efficient as static arrays because unused segments of memory are often left empty when the list is not fully populated to capacity. However, they offer the convenience of being dynamic in size and you can just add or remove elements without needing to re-reserve a new place in system memory.

Practical Example: - A real-world example of using a list would be to process a very large amount of text line-by-line. This is often done when looking out text output from a simulation. You can read in the file and parse the text for carriage returns break up the text into individual lines. Then, you can loop over each line and do some sort of processing or searching, depending on the application. The mutibility of lists can also be quite convenient in this scenario if you, for example, needed to modify some content on individual lines and then write the new text back to a file. Sets could technically be usable in this example, but there is no way of knowing if you might have any duplicate lines that you would want to preserve and storing it in a set would exclude those duplicate lines. Also sets are inherently unordered, while lists are ordered - so if you tried to use a set when you wrote the file back out all your lines would be in some random order instead of the order you read them in as.

[ ]:
```
# Lists can be defined with square brackets or with the list() function, first␣
  ↪let's take a look at defining an empty list:
# Time  Complexity: O(1)
# Space Complexity: O(1)
my_list = []
```

```
print(my_list)
```

```
[]
```

[ ]:
```
# Alternatively, we can define a list with values in it:
# Time  Complexity: O(n) -> n is length of list
# Space Complexity: O(n) -> n is length of list
my_list = [1, 2, 3, 4, 5]
print(my_list)
```

```
[1, 2, 3, 4, 5]
```

[ ]:
```
# To add an element to a list, we can use the built in append() method, note␣
 ↪that this will add the element to the end of the list:
# Time  Complexity: O(1)
# Space Complexity: O(1)
my_list.append(6)
print(my_list)
```

```
[1, 2, 3, 4, 5, 6]
```

[ ]:
```
# To remove an element from a list, we can use the built in remove() method,␣
 ↪note that this will remove the first instance of the value in the list:
# Time  Complexity: O(n) -> n is length of list
# Space Complexity: O(1)
my_list.remove(6)
print(my_list)
```

```
[1, 2, 3, 4, 5]
```

[ ]:
```
# You can also remove an element by index using the del keyword:
# Time  Complexity: O(n) -> n is length of list
# Space Complexity: O(1)
del my_list[0]
print(my_list)
```

```
[2, 3, 4, 5]
```

[ ]:
```
# To access elements of the list, we can use an index-based syntax (with␣
 ↪zero-based indexing):
# Time  Complexity: O(1)
# Space Complexity: O(1)
print(my_list[2]) # print the 3rd element of the list
```

```
4
```

[ ]:
```
# It's also possible to access a range of elements in a list using the slice␣
 ↪syntax:
# Time  Complexity: O(m) -> m is the length of the slice
# Space Complexity: O(m) -> m is the length of the slice
```

```
print(my_list[2:4]) # print the 3rd and 4th elements of the list (note that the␣
 ↪end value is non-inclusive)
```

```
[4, 5]
```

[ ]:
```
# You can also use slice a list by specifying a start and end index with a step␣
 ↪size:
# Time  Complexity: O(m) -> m is the length of the slice
# Space Complexity: O(m) -> m is the length of the slice
print(my_list[0:4:2]) # print every other element in the list, starting at the␣
 ↪first element
print(my_list[::-1]) # print the list in reverse order
```

```
[2, 4]
[5, 4, 3, 2]
```

[ ]:
```
# Similar to other data structures, you can loop over all elements in a list␣
 ↪using a for loop:
# Time  Complexity: O(n) -> n is length of list
# Space Complexity: O(n) -> n is length of list
for element in my_list:
    print(element)
```

```
2
3
4
5
```

[ ]:
```
# Sometimes it's useful to combine multiple lists together by concetenate them:
# Time  Complexity: O(n + m) -> n is length of orig list, m the length of the␣
 ↪new list being added
# Space Complexity: O(n + m) -> n is length of orig list, m the length of the␣
 ↪new list being added
my_list = [5, 6, 7] + my_list
print(my_list)
```

```
[5, 6, 7, 2, 3, 4, 5]
```

[ ]:
```
# You can extend a list with another list using the extend() method:
# Time  Complexity: O(m) -> m is the number of added elements
# Space Complexity: O(m) -> m is the number of added elements
my_list.extend([1, 8, 9, 10])
print(my_list)
```

```
[5, 6, 7, 2, 3, 4, 5, 1, 8, 9, 10]
```

[ ]:
```
# If we need the list sorted, we can use the built in sort() method:
# Time  Complexity: O(n log n) -> n is length of list, sort() uses Timsort␣
 ↪algorithm
```

```
# Space Complexity: O(log n)   -> n is length of list, sort() uses Timsort␣
 ↪algorithm and has a maximum recursive call stack of log n
my_list.sort()
print(my_list)
```

```
[1, 2, 3, 4, 5, 5, 6, 7, 8, 9, 10]
```

1.0.8 Practice Exercises: Lists

Exercise 1: List comprehension is a useful and powerful tool when writing Python code. Write a function that takes in a list of numbers as an input and uses list comprehension to square each element of the list. Return the new list of squared elements.

Exercise 2: Write a function that takes in a large list of words as an input and filters the list for any element that begin with a randomly generated letter. Return the filtered list as an output. (Note: feel free to also use list comprehension in this exmaple as well)

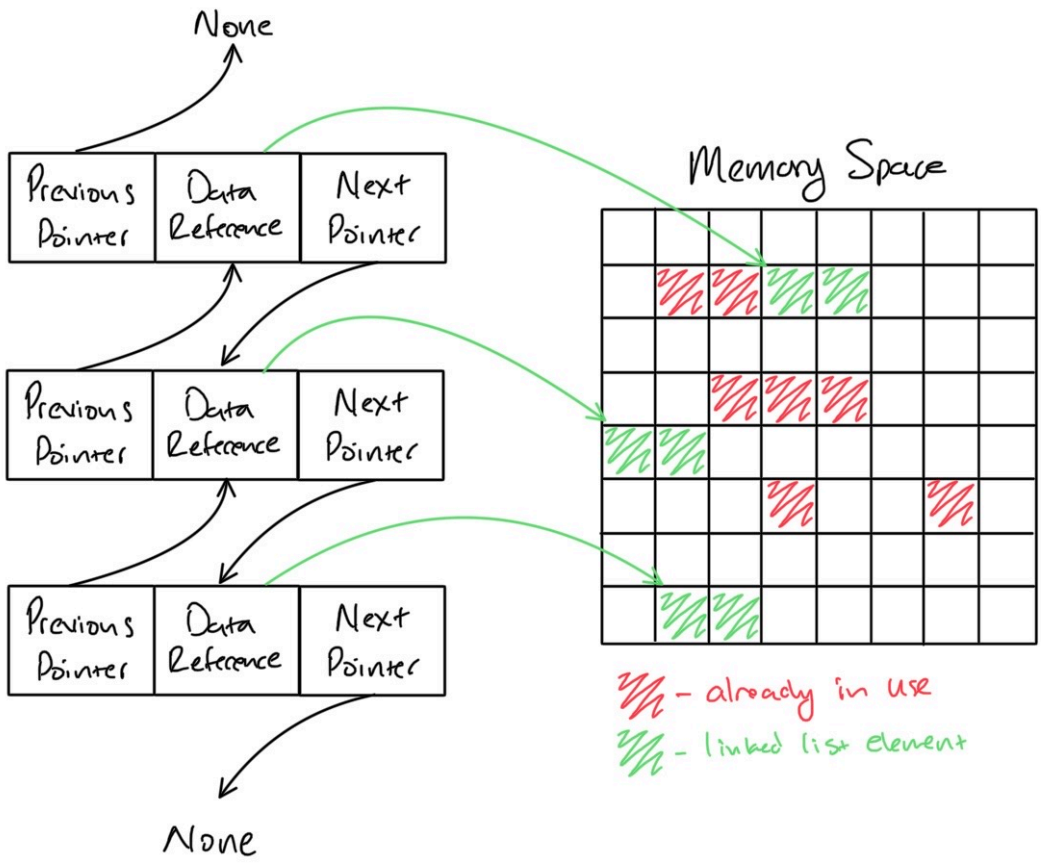
1.0.9 Section 5: Linked Lists

Linked Lists Intro

Linked lists are quite different from Lists even though they share a similar name. They are similar in that they represent data in a specific order, but the way they are stored in memory is very different. Linked lists are a series of individual objects that contain specific data as well as a reference to the following object next in the list. Each of these individual objects are not required to be stored in one contiguous chunk of memory, but are instead stored anywhere that the individual objects can fit and have a pointer that tells the system where else in memory to find the next object in the list. This type of data structure is useful for creating systems such as a music playlist - where each object is a song and also contains a reference to the next song upcoming in the playlist. There are a number of types of lists, depending on the implementation method: singly-linked lists, doubly-linked lists, and circularly-linked lists. These distinctions represent how the pointers are structured in the list as a hole. Singly-linked lists have a pointer at each element that only points one direction for the whole list. Doubly-linked lists have pointers that point from the current element in both directions to adjacent elements in the list. Lastly, circularly-linked lists can be either singly- or doubly-linked lists with the caveat that the 1st and last elements (head and tail of the list) must point to each other according to the structure of the list (singly or doubly, where directionality must be adhered to in singly). For the sake of brevity, I will only demonstrate a singly linked list below - but note that the only difference between these types is the Node class definition. This definition will have a next and prev attribute to keep track of elements in both directions for a doubly-linked list and for a circularly linked list the head and tail will point to each other instead of pointing to None.

Practical Example: - Linked lists have many useful applicaitons. One example is the undo/redo functionality of a text editor. As changes are made to a document, those sequential changes can be stored as a sequence of operations in a linked list. If you want to have both undo and redo functionality, then you would use a doubly linked list to move back and forth between operations. Often linked lists are used for their space complexity scalability. Since these are list-like data structures who do not require contigous pieces of memory for storage, they can accomodate very large sequences of items where each item is dynamically stored in memory in smaller pieces where it can more easily find an opening and fit.

Diagram: - See the diagram below that provides a visualization of a doubly-linked list. Note that the memory visualization is simplified for the sake of the diagram, but the important part is to show that each element in the linked list can be stored in non-contiguous memory. This provides great scalability for very large lists.



[ ]:
```python
# As an example, lets create a node class and a SinglyLinkedList class:

# Clase for node in a singly-linked list
class Node:

    # The __init__ method is called when a new instance of the class is␣
 ↪created, it is used to initialize the instance with any values we want
    # Time  Complexity: O(1)
    # Space Complexity: O(1)
    def __init__(self, data):
        self.data = data
        self.next = None

# Class for singly-linked list
class SinglyLinkedList:
```

```
    # The __init__ method is called when a new instance of the class is␣
↪created, it is used to initialize the instance with any values we want
    # Time  Complexity: O(1)
    # Space Complexity: O(1)
    def __init__(self):
        self.head = None
        self.tail = None
        self.__size = 0 # The double underscore before the variable name makes␣
↪it a private variable, meaning that it can only be accessed from within the␣
↪class

    # The __str__ method is called when we want to print the object, it should␣
↪return a string representation of the object
    # This method starts at the first node and subsequently prints the data␣
↪stored in each node until it reaches the end of the list
    # Time  Complexity: O(n) -> n is the size of the list
    # Space Complexity: O(n) -> n is the size of the list
    def __str__(self):
        cur = self.head
        out = ""
        while cur != None:
            out += "{}->".format(cur.data)
            cur  = cur.next
        out += "None"
        return out

    # The __len__ method is called when we want to get the length of the␣
↪object, it should return an integer representing the length of the object
    # Note that the __size variable is a private variable, meaning that it can␣
↪only be accessed from within the class
    # Time  Complexity: O(1)
    # Space Complexity: O(1)
    def __len__(self):
        return self.__size

    # The push method adds a new node to the beginning of the list
    # Time  Complexity: O(1)
    # Space Complexity: O(1)
    def push(self, data):
        new_node      = Node(data)
        new_node.next = self.head
        if self.head == None:
            self.tail = new_node
        self.head     = new_node
        self.__size  += 1
```

```
    # The append method adds a new node to the end of the list
    # Time  Complexity: O(1)
    # Space Complexity: O(1)
    def append(self, data):
        new_node       = Node(data)
        self.tail.next = new_node
        self.tail      = new_node
        self.__size   += 1

    # The get_at method returns the data stored in the node at the specified index (similar to indexing a list)
    # Time  Complexity: O(n) -> n is the size of the list
    # Space Complexity: O(1)
    def get_at(self, index):
        cur = self.head
        count = 0
        while count < index:
            cur = cur.next
            count += 1
        return cur.data

    # The remove_at method removes the node at the specified index
    # Time  Complexity: O(n) -> n is the size of the list
    # Space Complexity: O(1)
    def remove_at(self, index):
        prev = None
        cur  = self.head
        count = 0
        while count <= index:
            if count == index:
                if count == 0:
                    self.head = cur.next
                elif count == self.__size:
                    prev.next = None
                else:
                    prev.next = cur.next
            prev = cur
            cur  = cur.next
            count += 1
        self.__size -= 1
        return self.__str__()

    # The traverse method just uses the __str__ method to print the list
    # Time  Complexity: O(n) -> n is the size of the list
    # Space Complexity: O(n) -> n is the size of the list
    def traverse(self):
```

```
        return self.__str__()
```

[ ]:
```
# First let's create a new SinglyLinkedList object:
# Time  Complexity: O(1)
# Space Complexity: O(1)
my_list = SinglyLinkedList()
```

[ ]:
```
# Now let's add some nodes to the beginning of the list using the push method:
# Time  Complexity: O(1)
# Space Complexity: O(1)
my_list.push(1)
my_list.push(2)

# Let's print the list to see what it looks like so far:
print(my_list)
```

```
2->1->None
```

[ ]:
```
# Let's add some nodes to the end of the list using the append method:
# Time  Complexity: O(1)
# Space Complexity: O(1)
my_list.append(3)
my_list.append(4)

# Let's print the list again to see how it's changed:
print(my_list)
```

```
2->1->3->4->None
```

[ ]:
```
# Let's get the data stored in the node at index 2 using the get_at method:
# Time  Complexity: O(n) -> n is the size of the list
# Space Complexity: O(1)
print(my_list.get_at(2))
```

```
3
```

[ ]:
```
# Let's remove the node at index 2 using the remove_at method:
# Time  Complexity: O(n) -> n is the size of the list
# Space Complexity: O(1)
my_list.remove_at(2)

# Let's print the list again to see how it's changed:
print(my_list)
```

```
2->1->4->None
```

[ ]:
```
# Note that printing the list is the same as using the traverse method in the␣
↪class above (because we created a custom __str__ method in the class)
# Time  Complexity: O(n) -> n is the size of the list
```

```
# Space Complexity: O(n) -> n is the size of the list
print(my_list.traverse())
```

```
2->1->4->None
```

1.0.10 Practice Exercises: Linked Lists

Exercise 1: Write a Python function that takes a singly-linked list as an input and returns the middle element in the list. Provide a visualization that proves the returned element is in-fact at the center of the list.

Exercise 2: Write a Python function that takes a doubly-linked list of characters as an input and returns if the word the characters form when strung together is a palandrome or not. (Hint: make sure your list has both a head and a tail defined)

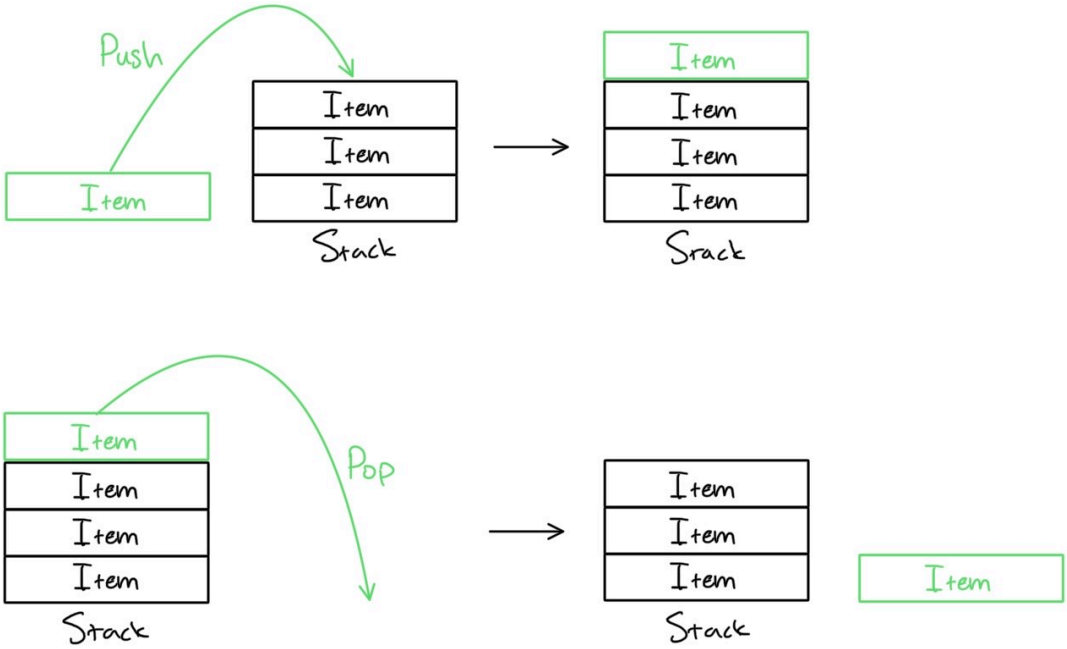
1.0.11 Section 6: Stacks

Stacks Intro

Stacks are useful data structures that can be implemented in a number of ways. Two common ways are to use a List or a Linked List to implement the concept. The generat concept of a Stack is first-in-last-out. To understand this concept, imagine stacking a sequence of boxes on top of one another on the ground. As you put boxes onto the pile, the first ones you place will be the last ones accessible to you because all subsequent boxes will be on top and blocking the previously added ones. This same concept applies to a stack. When you add items using the push() method, they go into the 1st place that the stack will pull from when you try to extract an item with the pop() method. You can also take a look at what's coming up next using the peek() method which doesn't remove the item from the stack but just returns the value of the next item. Stacks are used for a wide variety of applications, some examples include: undo/redo funcitonality in a program and evaluating mathematical expressions.

Practical Example: - A real-world example of using a stack is for navigating a maze. As you progress through the maze, you can push your current position onto the stack periodically. If you run into a dead end, then you simply need to pop items off the stack until you get back to where you could have chosen a different path. Eventually you will find the solution to the maze. A stack is particularly well suited for this because it keeps an ordered sequence of events as they occured. If you were to use something like a set that is unordered, you would fail misurably as you wouldn't be able to confidently know where you came from to be able to backtrack once you hit a dead end.

Diagram: - See the diagram below for a visualization of how the push and pop methods functionally work to add and remove items from the stack. The peek method doesn't move items on or off the stack, but instead just looks at the data stored at the next item that would be removed if you executed the pop method.



[ ]:
```
# Class for each element in the stack
class ElementStack:

    # The __init__ method is called when a new instance of the class is␣
 ↪created, it is used to initialize the instance with any values we want
    # Time  Complexity: O(1)
    # Space Complexity: O(1)
    def __init__(self, value):
        self.data = value
        self.next = None

# Class for stack
class Stack:

    # The __init__ method is called when a new instance of the class is␣
 ↪created, it is used to initialize the instance with any values we want
    # Time  Complexity: O(1)
    # Space Complexity: O(1)
    def __init__(self):
        self.head   = None
        self.__size = 0

    # Overwrite default method for print()
    # Time  Complexity: O(n) -> n is the size of the list
    # Space Complexity: O(n) -> n is the size of the list
    def __str__(self):
```

```
        if self.__size == 0: return 'None'
        cur = self.head
        out = ""
        while cur != None:
            out += f"{cur.data}->"
            cur  = cur.next
        out += "None"
        return out

    # Overwrite default method for len()
    # Time  Complexity: O(1)
    # Space Complexity: O(1)
    def __len__(self):
        return self.__size

    # Inserts an object at the top of the Stack.
    # Time  Complexity: O(1)
    # Space Complexity: O(1)
    def push(self, value):
        new_node      = ElementStack(value)
        new_node.next = self.head
        self.head     = new_node
        self.__size  += 1
        return self.__str__()

    # Removes and returns the object at the top of the Stack.
    # Time  Complexity: O(1)
    # Space Complexity: O(1)
    def pop(self):
        if self.__size == 0: return 'Error, stack is empty'
        out = self.head.data
        if self.__size == 1:
            self.head = None
        else:
            self.head = self.head.next
        self.__size -= 1
        return out

    # Returns the object at the top of the Stack without removing it.
    # Time  Complexity: O(1)
    # Space Complexity: O(1)
    def peek(self):
        return self.head.data
```

[ ]:
```
# First let's create a new Stack object:
my_stack = Stack()
```

```
# Now let's add some elements to the stack using the push method:
# Time  Complexity: O(1) each
# Space Complexity: O(1) each
my_stack.push(1)
my_stack.push(2)
my_stack.push(3)
my_stack.push(4)# Let's print the stack to see what it looks like so far:
print(my_stack)
```

```
4->3->2->1->None
```

[ ]:
```
# Let's get the next element from the stack using the pop method:
# Time  Complexity: O(1)
# Space Complexity: O(1)
print(my_stack.pop())

# Let's print the stack again to see how it's changed:
print(my_stack)
```

```
4
3->2->1->None
```

[ ]:
```
# Let's peek at the top element of the stack using the peek method:
# Time  Complexity: O(1)
# Space Complexity: O(1)
print(my_stack.peek())
```

```
3
```

[ ]:
```
# Now let's pop off elements until the stack is empty:
# Time  Complexity: O(1) each
# Space Complexity: O(1) each
print(my_stack.pop())
print(my_stack.pop())
print(my_stack.pop())
print(my_stack.pop())

# Let's print the stack again to see how it's changed:
print(my_stack)
```

```
3
2
1
Error, stack is empty
None
```

1.0.12 Practice Exercises: Stacks

Exercise 1: Implement a stack that has a max capacity limit. Demonstrate that this limit is working by adding elements until it "overflows".

Exercise 2: Use a stack to simplify a redundant Linux file system path (i.e. /home/user/./../user/jeff/../joe/documents/ -> /home/user/joe/documents). (Note: the '.' sequence represents 'current directory' while the '..' sequence represents 'previous directory')

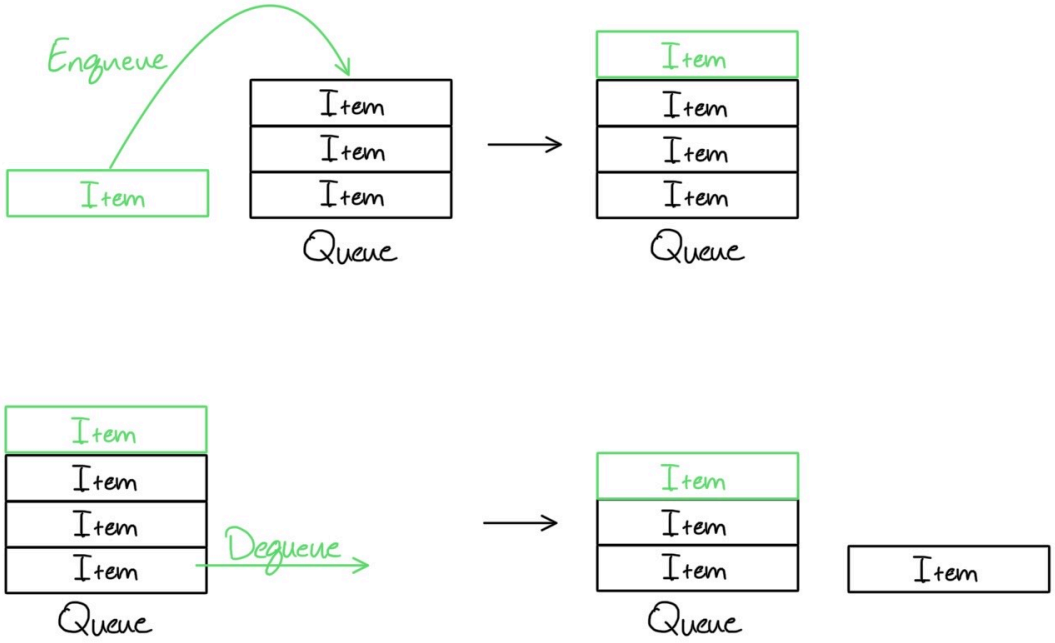
1.0.13 Section 6: Queues

Queues Intro

Queues are nearly identical to stacks except they store and return elements in the linked list in the opposite order. The fundamental behavior of a queue is first-in-first-out. This is easy to visualize as we see them nearly every day in our lives. Consider waiting in like at the drive through or standing in line to get tickets at a movie theater. These are all examples of queues, where the first person to arrive is the first person to get processes. Unlike stacks, the standard nomenclature for adding and removing items from the queue is enqueue (to add) and dequeue (to remove). Both Queues and Stacks use the same peek method to see what the next item is without removing it. Common examples of where Queues are used include: print job scheduling, simulation task scheduling for high performance computing jobs, buffers for networking applicaions.

Practical Example: - A common real-world example of using queues is how the print queue works when you send documents to the printer. Printing typically is addressed on a first-come-first-serve basis. This premise can be easily implemented as a first-in-first-out (FIFO) Queue where each document sent to the printer is added to the back of the queue and as the printer completes print jobs it grabs the next item from the front of the queue.

Diagram: - See the diagram below for a visualization of how the enqueue and dequeue methods functionally work to add and remove items from the queue. The peek method doesn't move items on or off the queue, but instead just looks at the data stored at the next item that would be removed if you executed the dequeue method.



[ ]:
```
# Class for each element in the queue
class ElementQueue:

    # The __init__ method is called when a new instance of the class is␣
 ↪created, it is used to initialize the instance with any values we want
    # Time  Complexity: O(1)
    # Space Complexity: O(1)
    def __init__(self, value):
        self.data = value
        self.next = None

# Class for queue
class Queue:

    # The __init__ method is called when a new instance of the class is␣
 ↪created, it is used to initialize the instance with any values we want
    # Time  Complexity: O(1)
    # Space Complexity: O(1)
    def __init__(self):
        self.head   = None
        self.tail   = None
        self.__size = 0

    # Overwrite default method for print()
    # Time  Complexity: O(n) -> n is the size of the list
    # Space Complexity: O(n) -> n is the size of the list
    def __str__(self):
        if self.__size == 0: return 'None'
        cur = self.head
        out = ""
        while cur != None:
            out += f"{cur.data}->"
            cur  = cur.next
        out += "None"
        return out

    # Overwrite default method for len()
    # Time  Complexity: O(1)
    # Space Complexity: O(1)
    def __len__(self):
        return self.__size

    # Adds an element to the start of the Queue.
    # Time  Complexity: O(1)
    # Space Complexity: O(1)
    def enqueue(self, value):
        new_node = ElementQueue(value)
```

```
        if self.__size == 0:
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            self.tail      = new_node
        self.__size += 1
        return self.__str__()

    # Removes the oldest element from the start of the Queue.
    # Time  Complexity: O(1)
    # Space Complexity: O(1)
    def dequeue(self):
        if self.__size == 0: return 'Error, queue is empty'
        out = self.head.data
        if self.__size == 1:
            self.head = None
        else:
            self.head = self.head.next
        self.__size -= 1
        return out

    # Returns the oldest element that is at the start of the Queue but does not␣
 ↪remove it from the Queue.
    # Time  Complexity: O(1)
    # Space Complexity: O(1)
    def peek(self):
        return self.head.data
```

[ ]:
```
# First let's create a new Queue object:
# Time  Complexity: O(1)
# Space Complexity: O(1)
my_queue = Queue()

# Now let's add some elements to the queue using the enqueue method:
# Time  Complexity: O(1) each
# Space Complexity: O(1) each
my_queue.enqueue(1)
my_queue.enqueue(2)
my_queue.enqueue(3)
my_queue.enqueue(4)

# Let's print the queue to see what it looks like so far:
print(my_queue)
```

```
1->2->3->4->None
```

[ ]:
```
# Let's get the next element from the queue using the dequeue method:
# Time  Complexity: O(1)
# Space Complexity: O(1)
print(my_queue.dequeue())

# Let's print the queue again to see how it's changed:
print(my_queue)
```

```
1
2->3->4->None
```

[ ]:
```
# Let's peek at the next element in the queue using the peek method:
# Time  Complexity: O(1)
# Space Complexity: O(1)
print(my_queue.peek())
```

```
2
```

[ ]:
```
# Now let's dequeue elements until the queue is empty:
# Time  Complexity: O(1) each
# Space Complexity: O(1) each
print(my_queue.dequeue())
print(my_queue.dequeue())
print(my_queue.dequeue())
print(my_queue.dequeue())
```

```
2
3
4
Error, queue is empty
```

[ ]:
```
# Let's print the queue again to see how it's changed:
print(my_queue)
```

```
None
```

1.0.14 Practice Exercises: Queues

Exercise 1: Use a queue to simulate the daily operations of a call center. As each call comes in, randomly assign a call duration between 1 and 5 minutes. Assume you have only 2 operators. Simulate 1 hour of operation of calls that come in randomly every 2-3 minutes.

Exercise 2: Couple your call center simulation with another simulation that models the printer inside the call center. Assume there is a 25% chance that during any given call, the operator must print off a packet of documents for later review. The packets vary in size but are quite length, taking 15-20 minutes to fully print. Simulate the same 1 hour of operation of the call center, but, once the call center work is done, continue the simulation until all print jobs have completed.