Data Structures and Algorithms

Louie Venegas

CEIS295

Introduction

This is an introduction to my class project for a course called Data Structures and Algorithms. Throughout the course of this presentation, I will introduce various data structures and searching algorithms. In each section, a real-life scenario will be applied using a data set to populate the specified data structure. These scenarios include a printer queue, a call service queue, a customer service center, and a call center, as well as inserting and deleting various amounts of records into and out of structures. The primary purpose of this exercise is to learn how these algorithms work and interact with the data structures, and to test the efficiency and speed of the operations performed using them.

Def Client Class:

In programming, when a class is defined, it becomes a model for a data type. Throughout the course of this project, I will use the `Client` class data type continuously. The `Client` class is defined in Python in the upper right photo. I will use this data type to run tests capturing the speeds of various data structures interacting with algorithms and performing operations defined by the classes.

The second photo is of the data I will use to fill parameters of the `Client` class and populate our arrays, lists, queues, etc. In other words, this class and data format will be the basis of our data I/O throughout the project.

```
        self.__phone = phone
        self.__email = email

    #classes that compare objects must implement __eq__ method and __lt__ methods
    #__lt__ means " less then" and must return boolean
    #__eq__ means "equals" and must return boolean
    def __le__(self, other):
        return self.__client_id <= other.__client_id

    def __lt__(self, other):
        return self.__client_id < other.__client_id

    def __eq__(self, other):
        return self.__client_id == other.__client_id

    # __str__() is automaticlly called when you print thr object
    def __str__(self):
        return str(self.__client_id)

    # getters and setters (shhh be berry berry quiet we are skipping validation)

    def get_client_id(self):
        return self.__client_id

    def set_client_id(self, client_id):
        self.__client_id = client_id

    def get_first_name(self):
        return self.__first_name

    def set_first_name(self, first_name):
        self.__first_name = first_name

    def get_last_name(self):
        return self.__last_name

    def set_last_name(self, last_name):
        self.__last_name = last_name

    def get_phone(self):
```

ClientData10 | httpsfl | define | Client

File Edit View

ClientData.CSV

```
102642,Bevis,Rivera,(406) 436-3341,elementum.dui@pellentesqueSed.edu
101078,Hadassah,Ashley,(525) 537-5797,odio.Phasellus@Nunc.com
109171,Kelly,Holt,(595) 430-9042,Curabitur.consequat@dignissim.org
100308,Armand,Hobbs,(639) 404-6293,est@risus.ca
109465,Mason,Bartlett,(146) 221-9639,Mauris.molestie@diameudolor.co.uk
107636,Keely,Mclaughlin,(628) 180-2883,mattis.Cras.eget@condimentum.com
109425,Kirby,Sullivan,(712) 813-6533,nec.cursus.a@odioAliquamvulputate.net
103678,Zahir,Dean,(237) 659-5962,sollicitudin@Namnullamagna.edu
104450,Benjamin,Zimmerman,(547) 393-9784,pede.Praesent@Aliquamornareliberc
107033,Carter,Richard,(699) 929-9968,orci.lacus.vestibulum@mienim.net
107111,Dora,Herman,(833) 433-5795,inceptos.hymenaeos.Mauris@cursus.co.uk
102092,Neil,Collins,(790) 903-2310,faucibus.leo.in@eunullaat.edu
109892,Ivory,Armstrong,(953)
200-6831,Phasellus.vitae@aptenttacitisociosqu.co.uk
106979,Dai,Townsend,(312) 149-1758,Integer@lectusa.net
```

Ln 1, Col 1 | 647,630 characters | 100% | Windows (CRLF) | UTF-8

Data IO:

Inside of each python code for each section you will find this basic structure for reading clientdata.csv and converting each line into objects based on the class defined in the previous slide.
The object is then placed into the list clients and used for whatever operations and test we may be performing. For this project it is mostly about speed. Though I won't mention it much, Big O notation was used

```
11    # display name and date in output
12    print("Name:", "Louie Venegas")
13    print("Date:", date.today())
14    print()
15
16    #create list
17    clients = []
18
19    #read the records from the client data file
20    #place clients data into clients listn
21    Input_File_Name = "clientData.csv"
22    with open(Input_File_Name) as infile:
23        for line in infile:
24            #split the lines based one the commas
25            s = line.split(',')
26            client_id = int(s[0])
27            f_name = s[1]
28            l_name = s[2]
29            phone = s[3]
30            email = s[4]
31            #create a client object using the data from the file
32            clt = Client(client_id, f_name, l_name, phone, email)
33            #add the client object to the list
34            clients.append(clt)
35
```

The ArrayList & Real-World Speeds

- ***Adding Client data type to the ArrayList***
- ***Developing a Python code that tests the ArrayList's real world speeds***
- ***Developing an Excel table that displays relevant real-world speeds***

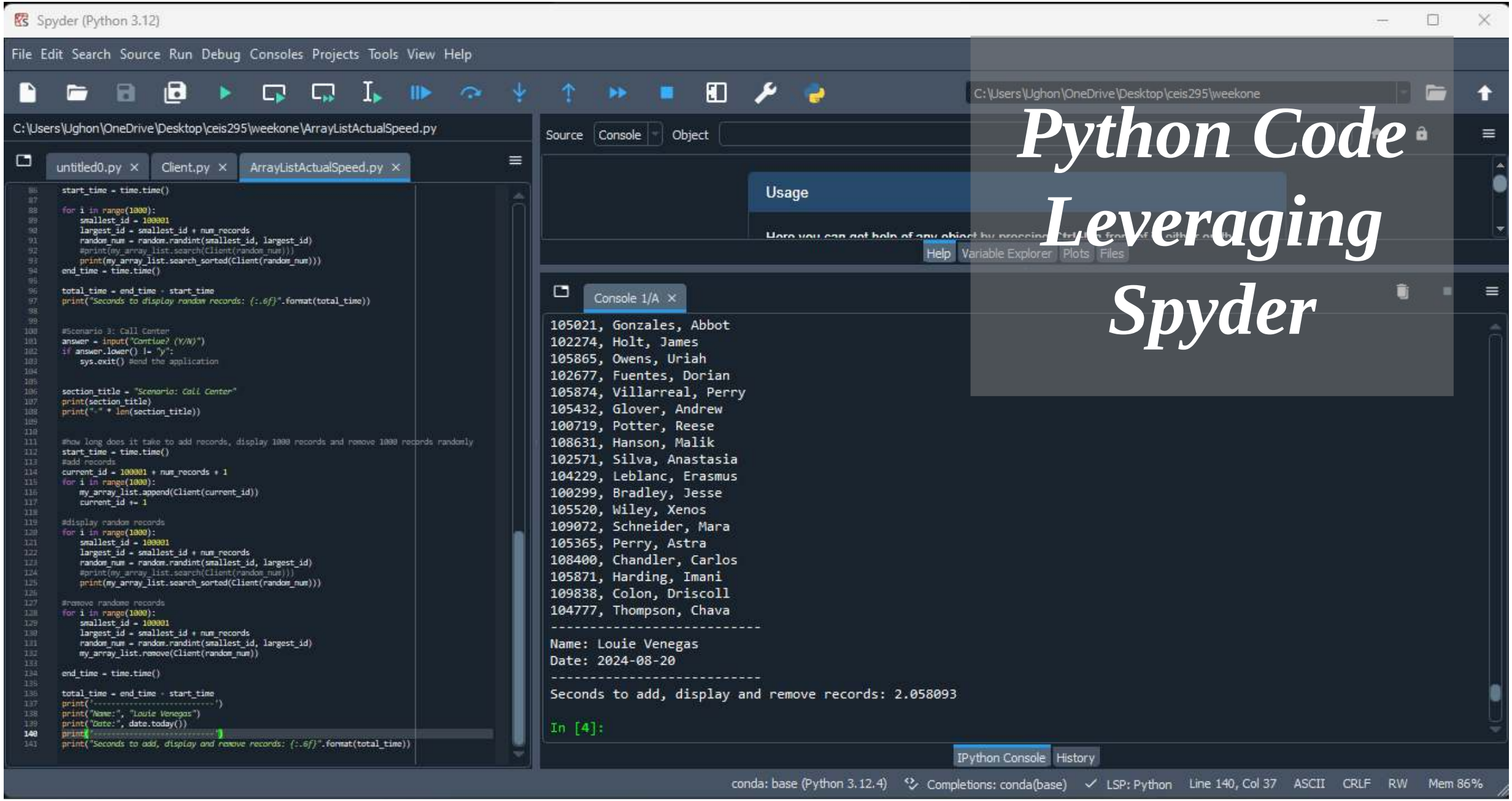
Introduction

In this section, we will explore applications and performance of our ArrayList data structure. Incorporating Client data type into our ArrayList, demonstrating its handling of complex objects. Next, we test the real-world performance of our ArrayList implementation through a series of Python-based speed tests. These tests provided insights into the efficiency of our data structure under various conditions. Finally, we'll present our findings in an easy-to-understand Excel table, showing the relevant real-world speeds of our ArrayList operations. This approach will give us a clear picture of how our ArrayList performs in practical scenarios.

Def Array-List:

An array list in Python is a dynamic array that can automatically resize itself to accommodate new elements, providing efficient access and modification. It supports various methods to add, remove, and manipulate elements, making it a flexible data structure for different use cases.

```python
#Louie Venegas
#08/20/2024
class ArrayList:
    def __init__(self, size=10):
        self.__array = [None] * size
        self.__capacity = size
        self.__length = 0

    def get(self, index):
        if index < self.__length:
            return self.__array[index]
        else:
            return None

    def append(self, new_item):
        # resize() if the array is full
        if self.__capacity == self.__length:
            self.resize(self.__length * 2)

        # Insert the new item at index equal to self.__length
        self.__array[self.__length] = new_item

        # Increment the array's length
        self.__length = self.__length + 1

    def remove_at(self, index):
        # create a reference for the item
        item = None

        # Make sure the index is valid for the current array
        if index >= 0 and index < self.__length:
            # get the item
            item = self.__array[index]

```



Python Code Leveraging Spyder

Table of Speeds

•*A table showing the time that it takes for the following scenarios:*

- *Printer Queue or Call Queue or Service Queue*
  - *Add many records to end of data structure*
  - *Pull all records off front of data structure*
- *Customer Service Center*
  - *Display random records*
- *Call Center*
  - *Add some records*
  - *Display random records*
  - *Remove some records*

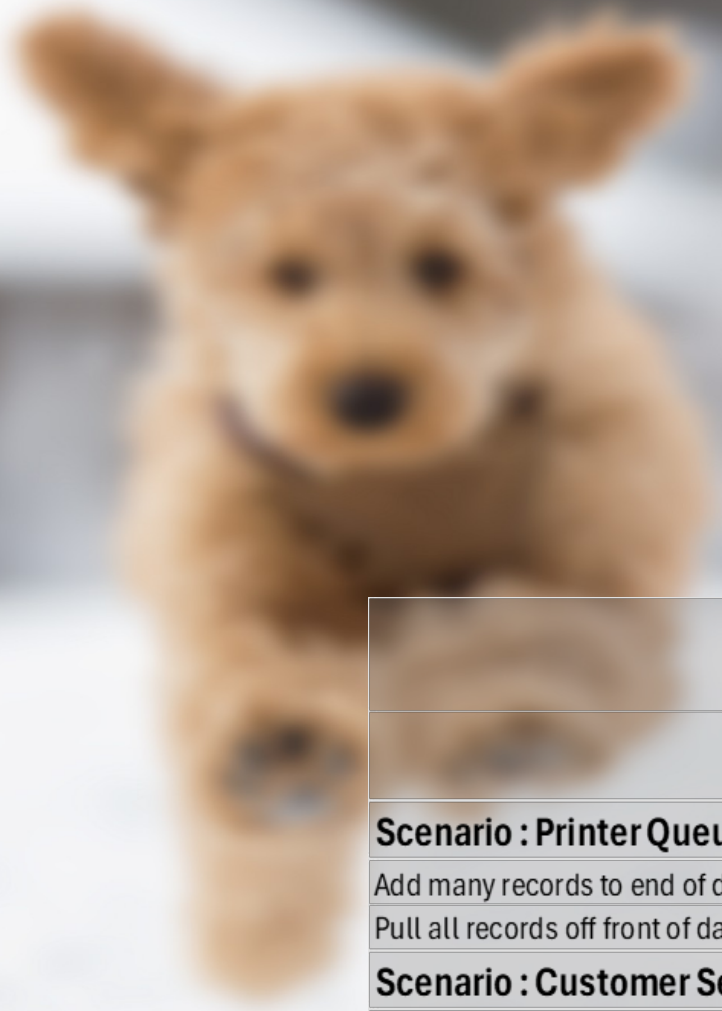


Table of Real-World Speeds

	ArrayList (unsorted)	Array List (sorted)
Scenario : Printer Queue or Call Service Queue		
Add many records to end of data structure:	0.000514	0.007978
Pull all records off front of data structure	3.840341	4.993699
Scenario : Customer Service Center		
Display random records	0.679132	0.018825
Scenario: Call Center		
Add some records, display random records, remove random records	2.403118	1.620621

- *Adding Client Data Type to the LinkedList*
- *Developing Python Code that Tests the LinkedList's Real World Speeds*
- *Using an Excel Table that Displays Relevant Real-World Speeds*

Linked List Real-World Speeds

Introduction

This segment focuses on the implementation of the LinkedList data structure. We begin by extending the LinkedList to incorporate a custom Client data type, demonstrating its ability to handle complex objects. A number performance tests will then be executed to evaluate the LinkedList's efficiency in real-world scenarios. These benchmarks will provide quantitative metrics on various operations, assessing the structure's time and complexity in practical applications. Finally, we will present a comprehensive analysis of our findings through an Excel-based visualization, offering insights into the LinkedList's performance characteristics across different operations and data sizes.

Def Linked-List:

A linked list is a linear data structure whose elements, called nodes, are each separate objects. Each node contains a reference, also called a link, to the next node in the sequence. This allows for efficient insertion and deletion of elements, as the structure changes dynamically without the need for contiguous memory allocation. Generally linear search algorithms are implemented with this data type.

```python
#Name: Louie Venegas
#Date: 08/21/2024

from Node import Node

class LinkedList:
    def __init__(self):
        self.__head = None
        self.__tail = None

    def get_first(self):
        if self.__head != None:
            return self.__head.data
        else:
            return None

    def get_last(self):
        if self.__tail != None:
            return self.__tail.data
        else:
            return None

    def get_at(self, index):
        if self.__head == None:
            return None
        else:
            temp = self.__head

            # "traverse" or step down the chain
            for i in range(0, index):
                if temp == None:
                    return None
                else:
                    temp = temp.next

            if temp == None:
                return None
            else:
```

Def node:

A node is a unit of data in a data structure, such as a linked list. A node contains data and points to the next node. A node is used in implementing a linked list for managing a sequence of elements where frequent insertions and deletions are required. Nodes can be used in a task scheduling system to manage tasks, allowing for efficient addition and removal of tasks without the need for contiguous memory allocation.

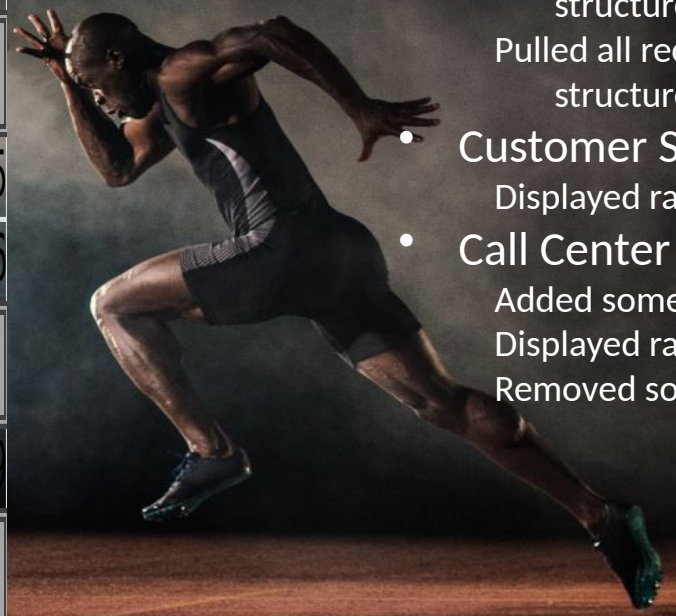
```
1    #Name: Louie Venegas
2    #Date: 08/20/2024
3
4    class Node:
5        def __init__(self, initial_data=None):
6            self.data = initial_data
7            self.next = None
```

```python
#Scenario 3: Call Center
answer = input("Contiue? (Y/N)")
if answer.lower() != "y":
    sys.exit() #end the application


section_title = "Scenario: Call Center"
print(section_title)
print("-" * len(section_title))


#how long does it take to add records, display 1000 records and remove 1000 records randomly
start_time = time.time()
#add records
current_id = 100001 + num_records + 1
for i in range(1000):
    my_linked_list.add_last(Client(current_id))
    current_id += 1

#display random records
for i in range(1000):
    smallest_id = 100001
    largest_id = smallest_id + num_records
    random_num = random.randint(smallest_id, largest_id)
    print(my_linked_list.search(Client(random_num)))

#remove random records
for i in range(1000):
    smallest_id = 100001
    largest_id = smallest_id + num_records
    random_num = random.randint(smallest_id, largest_id)
    my_linked_list.remove(Client(random_num))

end_time = time.time()
```

Python Code Leveraging Spyder

Table of Real-World Speeds

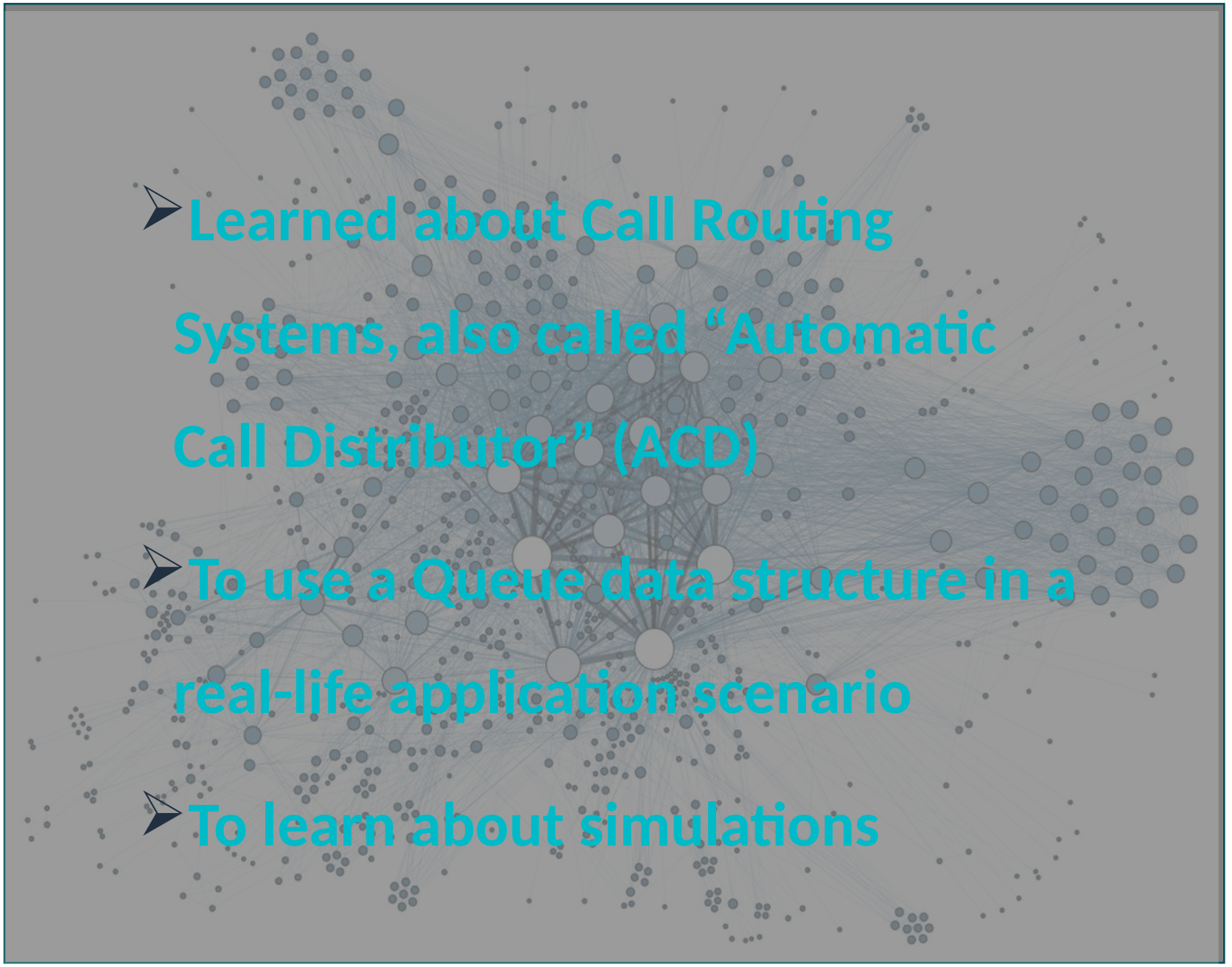
	ArrayList (unsorted)	Array List (sorted)	Linked list
Scenario: Printer Queue or Call Service Queue			
Add many records to end of data structure:	0.000514	0.007978	0.005035
Pull all records off front of data structure	3.840341	4.993699	0.002516
Scenario: Customer Service Center			
[illegible] records	0.67913	0.018825	0.441979
Scenario: Call Center			
[illegible] random records, remove random records	2.403118	1.620621	0.85841



Updated table from last week that shows the times for the ArrayList. I added a new column and showed the times for the LinkedList for comparison.

- Printer Queue or Call Queue or Service Queue
  - Added many records to end of data structure.
  - Pulled all records off front of data structure.
- Customer Service Center
  - Displayed random records.
- Call Center
  - Added some records
  - Displayed random records
  - Removed some records

Real-World Application Demo



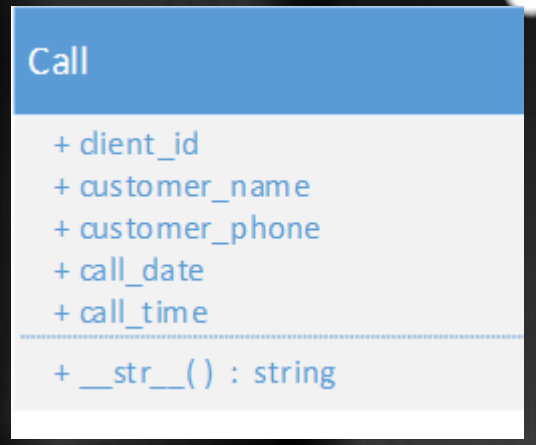
- Learned about Call Routing Systems, also called “Automatic Call Distributor” (ACD)
- To use a Queue data structure in a real-life application scenario
- To learn about simulations

Introduction

In this section, we examine the application of Queue data structures within a Call Routing Systems, also known as Automatic Call Distributors (ACD). Our focus will be on implementing a Queue data structure to model and simulate real-world call center scenarios, demonstrating its efficacy in handling time-sensitive, sequential operations. Through this process, we aim to bridge theoretical knowledge with practical implementation, offering a comprehensive understanding of how abstract data structures can be leveraged to solve concrete, industry-specific challenges.

Def Call Class:

When simulating a call center, I had to define an object model or custom data type to fit the role of the call. I built the call class based on the following UML diagram. To the right is an example of what the code for this class looked like.



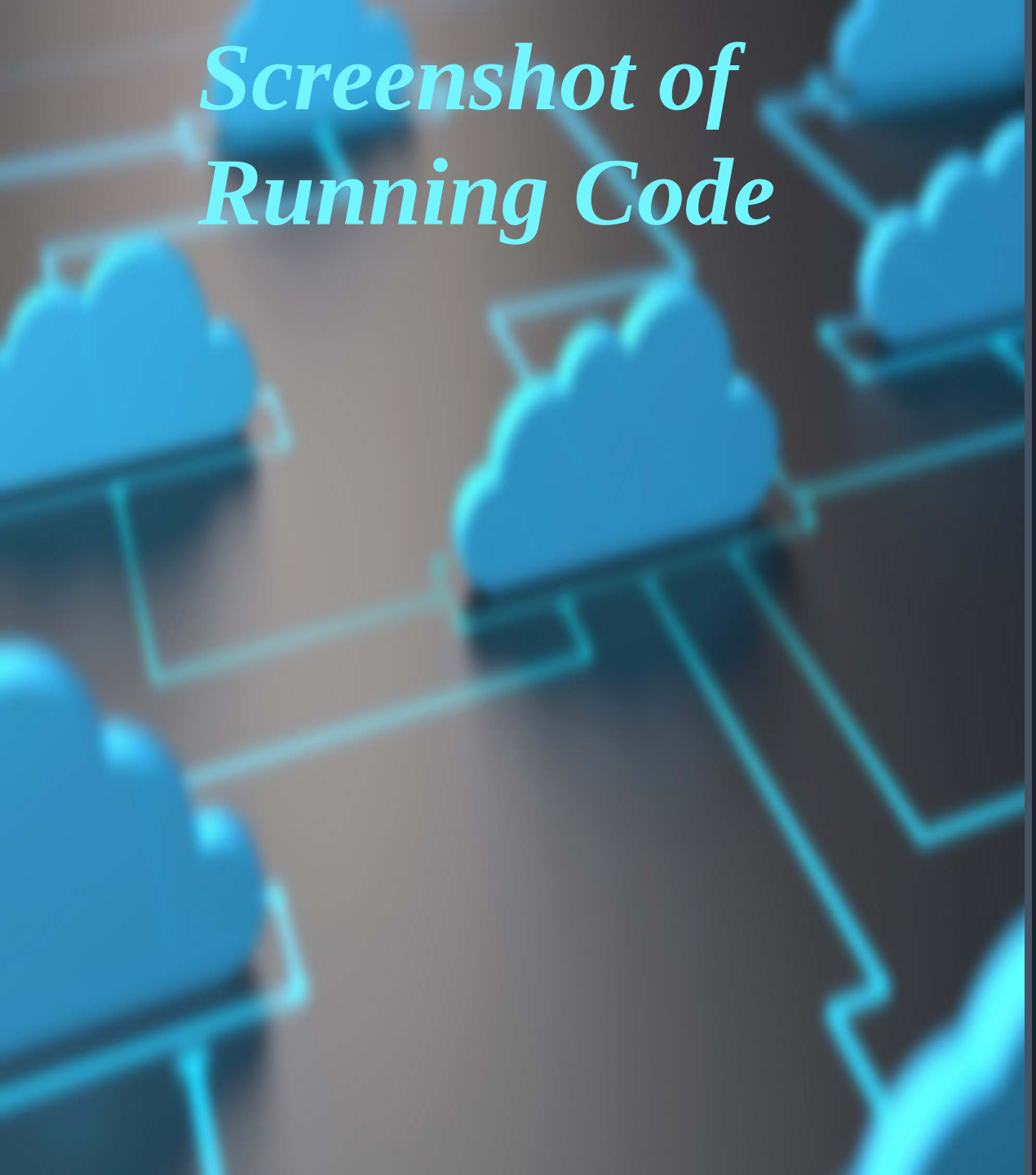

```
"""
name: Louie Venegas
Created on Fri Jul 26 13:33:41 2024
CEIS295
call class
"""

from time import strftime


class Call:
    def __init__(self, client_id=0, customer_name='unknown', customer_phone='unknown'):
        self.client_id = client_id
        self.customer_name = customer_name
        self.customer_phone = customer_phone
        self.call_date = strftime("%m/%d/%Y")
        self.call_time = strftime("%H:%M")

    #string method automaticlly called when you print the object.
    def __str__(self):
        return str(self.client_id) + ', ' + self.customer_name + ', ' + '\n\tPhone:'
        + self.customer_phone + ', ' + '\n\tDate/Time' + self.call_date + ', ' + '@' + self.call_time
```

Def Queue:

In Python, a queue is a linear data structure that follows the First-In-First-Out (FIFO) principle, where elements are added at the rear and removed from the front. A practical use case for a queue is in task scheduling, where tasks are processed in the order they arrive.

```python
from LinkedList import LinkedList

class Queue:
    def __init__(self):
        self.__length = 0
        self.__linked_list = LinkedList()

    def enqueue(self, data):
        self.__linked_list.add_last(data)
        self.__length += 1    # increment length

    def dequeue(self):
        data = self.__linked_list.get_first()
        self.__linked_list.remove_first()
        if data != None:
            self.__length -= 1    # decrement length
        return data

    def peek(self):
        return self.__linked_list.get_first()

    def is_empty(self):
        # if the length is zero, then the stack is empty
        return self.__length == 0

    def get_length(self):
        return self.__length

    def get_queue(self):
        return self.__linked_list.get_list()
```

Screenshot of Running Code



Console 1/A

```
Reloaded modules: call_class, Node, LinkedList, Queue
Name: Louie Venegas
Date 2024-07-26

How many seconds do you want to simulate?5
----------------------------------------
    No calls were recived or serviced during this time
frame.
    Number of calls waiting in queue: 0
----------------------------------------
Call was sent to rep for service
The call waiting queue is empty
    Number of calls waiting in queue: 0
----------------------------------------
Call recieved, caller was added to the queue
    Number of calls waiting in queue: 1
----------------------------------------
    No calls were recived or serviced during this time
frame.
    Number of calls waiting in queue: 1
----------------------------------------
Call recieved, caller was added to the queue
    Number of calls waiting in queue: 2
simulation has ended

In [5]:
```

IPython Console | History

- *Using Client Data Type in Sorting Algorithms*
- *Comparing Sorting Algorithms Using Data to Determine Real-World Speeds*
- *Leveraging Excel to Make a Table To Displays Relevant Real-World Speeds*

Sorting Algorithms

Section #4

Introduction

In this section, we will explore applications of sorting algorithms by utilizing client data type. We will compare various sorting algorithms to determine their real-world speeds using empirical data. Additionally, we will leverage Excel to create tables that clearly display these relevant speeds, providing a comprehensive analysis of each algorithm's performance in real-world scenarios.

Def QuickSort:

Quicksort is a sorting algorithm in Python that uses a divide-and-conquer approach, selecting a pivot and partitioning the array around it. A practical use case for quicksort is in database management systems, where it can be used to quickly sort large datasets, improving query performance and data retrieval times. This algorithm is often used on data sets to make searching algorithms more efficient.

```python
#Louie Venegas
#08/20/2024
class Quicksort:
    @classmethod
    def sort(cls, list):
        num_records = len(list)
        cls.quicksort(list, 0, num_records-1)

    @classmethod
    def quicksort(cls, list, start_index, end_index):
        # Only attempt to sort the list segment if there are
        # at least 2 elements
        if end_index <= start_index:
            return

        # Partition the list segment
        high = cls.partition(list, start_index, end_index)

        # Recursively sort the left segment
        cls.quicksort(list, start_index, high)

        # Recursively sort the right segment
        cls.quicksort(list, high + 1, end_index)

    @classmethod
    def partition(cls, numbers, start_index, end_index):
        # Select the middle value as the pivot.
        midpoint = start_index + (end_index - start_index) // 2
        pivot = numbers[midpoint]

        # "low" and "high" start at the ends of the list segment
        # and move towards each other.
        low = start_index
```

Def BubbleSort:

Bubble sort is a simple sorting algorithm in Python that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This sorting algorithm is mostly used in educational settings to teach students about sorting algorithms.

```python
#Louie Venegas
#08/20/2024
class BubbleSort:

    @staticmethod
    def sort(list):
        num_records = len(list)
        for i in range( num_records - 1 ):
            for j in range( num_records - i - 1 ):
                if list[j] > list[j+1]:
                    # flip them!
                    temp = list[j]
                    list[j] = list[j+1]
                    list[j+1] = temp
```

Def InsertionSort:

Insertion sort is a straightforward sorting algorithm that builds a sorted array one element at a time by comparing each new element to those already sorted and inserting it in the correct position.

```python
#Louie Venegas
#08/20/2024
class InsertionSort:

    @staticmethod
    def sort(list):
        num_records = len(list)
        for i in range(1, num_records):
            j = i

            # Insert list[i] into sorted part
            # stopping once list[i] in correct position
            while j > 0 and list[j] < list[j - 1]:
                # Swap numbers[j] and numbers[j - 1]
                temp = list[j]
                list[j] = list[j - 1]
                list[j - 1] = temp
                j -= 1
```

Def MergeSort:

Merge sort is an efficient, stable sorting algorithm that uses a divide-and-conquer approach to sort elements by recursively dividing the array into smaller subarrays, sorting them, and then merging them back together.

```python
#Louie Venegas
#08/20/2024
class MergeSort:

    @classmethod
    def sort(cls, list):
        num_records = len(list)
        # Initial call to merge_sort
        cls.merge_sort(list, 0, num_records - 1)

    @classmethod
    def merge_sort(cls, list, i, k):
        j = 0

        if i < k:
            j = (i + k) // 2  # Find the midpoint in the partition

            # Recursively sort left and right partitions
            cls.merge_sort(list, i, j)
            cls.merge_sort(list, j + 1, k)

            # Merge left and right partition in sorted order
            cls.merge(list, i, j, k)

    @classmethod
    def merge(cls, list, i, j, k):
        merged_size = k - i + 1                # Size of merged partition
        merged_list = [0] * merged_size    # Dynamically allocates temporary ar
                                               # for merged list
        merge_pos = 0                          # Position to insert merged numbe
        left_pos = i                           # Initialize left partition posit
        right_pos = j + 1                      # Initialize right partition posi
```

Def ShellSort:

Shell sort is an efficient sorting algorithm that generalizes insertion sort by allowing the exchange of far-apart elements to reduce the number of movements required.

```python
#Louie Venegas
#08/20/2024
class ShellSort:

    @staticmethod
    def sort(list):
        num_records = len(list)

        # keep cutting the gap size in half as an integer -- using //
        gap = num_records // 2;
        while gap > 0:
            for i in range(gap, num_records):
                temp = list[i]
                j = i;

                while j >= gap and temp < list[j - gap]:
                    list[j] = list[j - gap]
                    j -= gap;

                list[j] = temp

            # cut gap in half as an integer -- using //
            gap = gap // 2
```

Def SelectionSort:

Selection sort is a simple sorting algorithm that repeatedly selects the smallest (or largest) element from the unsorted portion of the list and moves it to the sorted portion.

```python
#Louie Venegas
#08/20/2024
class SelectionSort:

    @staticmethod
    def sort(list):
        num_records = len(list)
        for i in range(num_records - 1):

            # Find index of smallest remaining element
            index_smallest = i
            for j in range(i + 1, num_records):

                if list[j] < list[index_smallest]:
                    index_smallest = j

            # Swap list[i] and list[index_smallest]
            temp = list[i]
            list[i] = list[index_smallest]
            list[index_smallest] = temp
```

Table of Real-World Sorting Speeds

	100 Records	1000 Records	10,000 Records	100,000 Records
Bubble Sort:	0.003066	0.272625	28.298942	over 1 hour
Selection Sort:	0.001514	0.192276	19.64174	over 1 hour
Insertion Sort:	0.000721	0.129926	13.891487	12220.56215
Shell Sort:	0.00101	0.007099	0.136089	2.878491
Quick Sort:	0.001339	0.007718	0.125933	1.189674
Merge Sort:	0.001	0.004196	0.05094	0.73346

Searching Algorithms

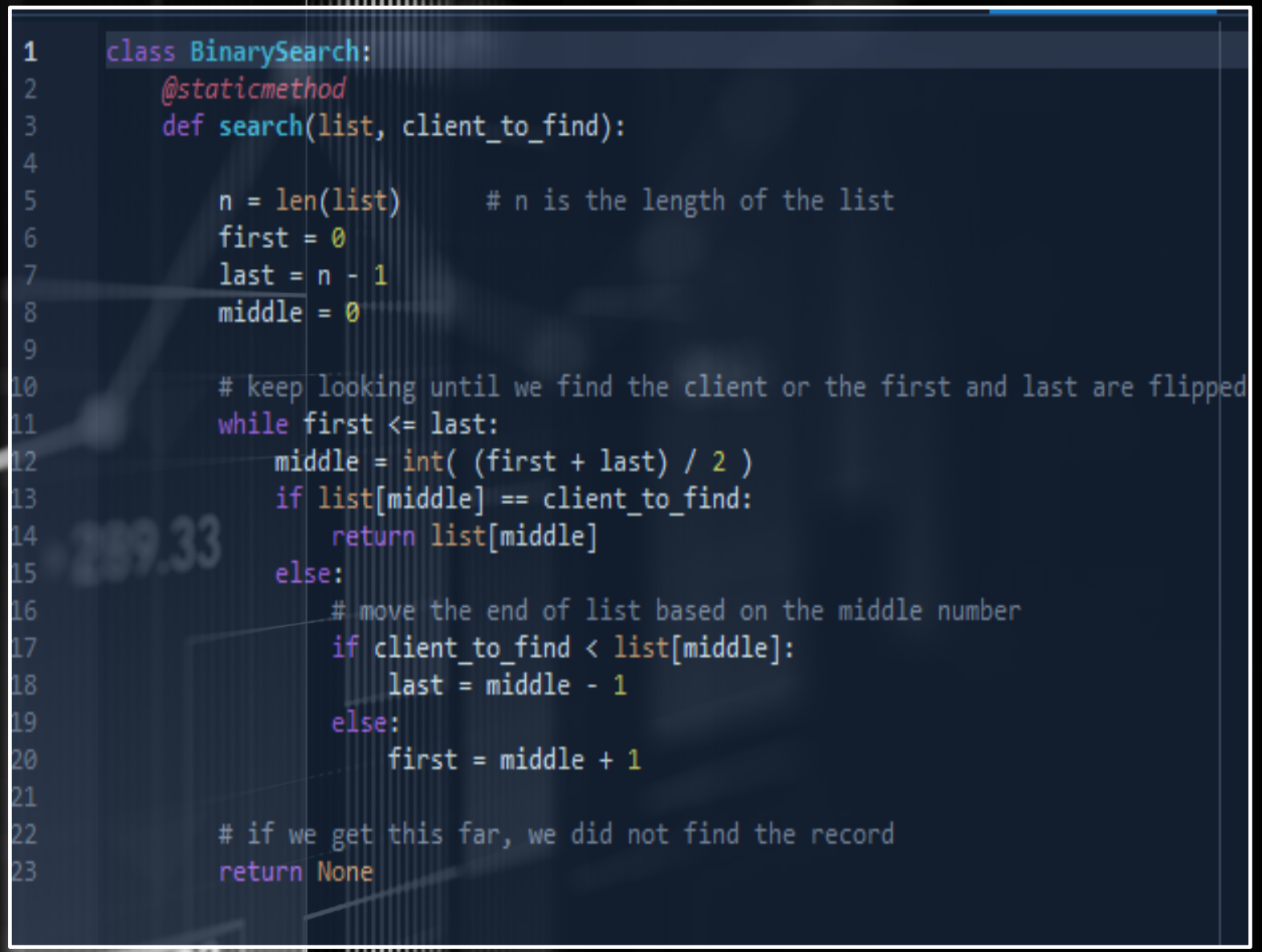
- ***Checking the Speed of a Linear Search on a Large Dataset***
- ***Comparing the Speed of a Binary Search on a Large Dataset when the Data is Sorted***

Introduction

In this section, we will investigate the performance of search algorithms on large datasets. We will begin by checking the speed of a linear search, which sequentially examines each element until the target is found. Next, we will compare this with the speed of a binary search, which is more efficient on sorted data by repeatedly dividing the search interval in half. This analysis will provide insights into the practical differences in speed and efficiency between these two fundamental search algorithms.

Def BinarySearch:

Binary search is an efficient algorithm used to find the position of a target value within a sorted array by repeatedly dividing the search interval in half.



```python
class BinarySearch:
    @staticmethod
    def search(list, client_to_find):

        n = len(list)      # n is the length of the list
        first = 0
        last = n - 1
        middle = 0

        # keep looking until we find the client or the first and last are flipped
        while first <= last:
            middle = int( (first + last) / 2 )
            if list[middle] == client_to_find:
                return list[middle]
            else:
                # move the end of list based on the middle number
                if client_to_find < list[middle]:
                    last = middle - 1
                else:
                    first = middle + 1

        # if we get this far, we did not find the record
        return None
```

Def LinearSearch:

Linear search is an algorithm that scans each element in a list sequentially to find a target value. Starting from the first element checking each one until the target is found or the list ends. This method can be inefficient for large datasets because if the query is not present it will sequentially search every item in the list.

```python
class LinearSearch:
    @staticmethod
    def search(list, client_to_find):
        n = len(list)           # number of elements in

        # step through the records one at a time
        for i in range( n ):
            if list[i] == client_to_find:
                # we found the item, so return it
                return list[i]

        # if we get this far, we did not find the record
        return None
```

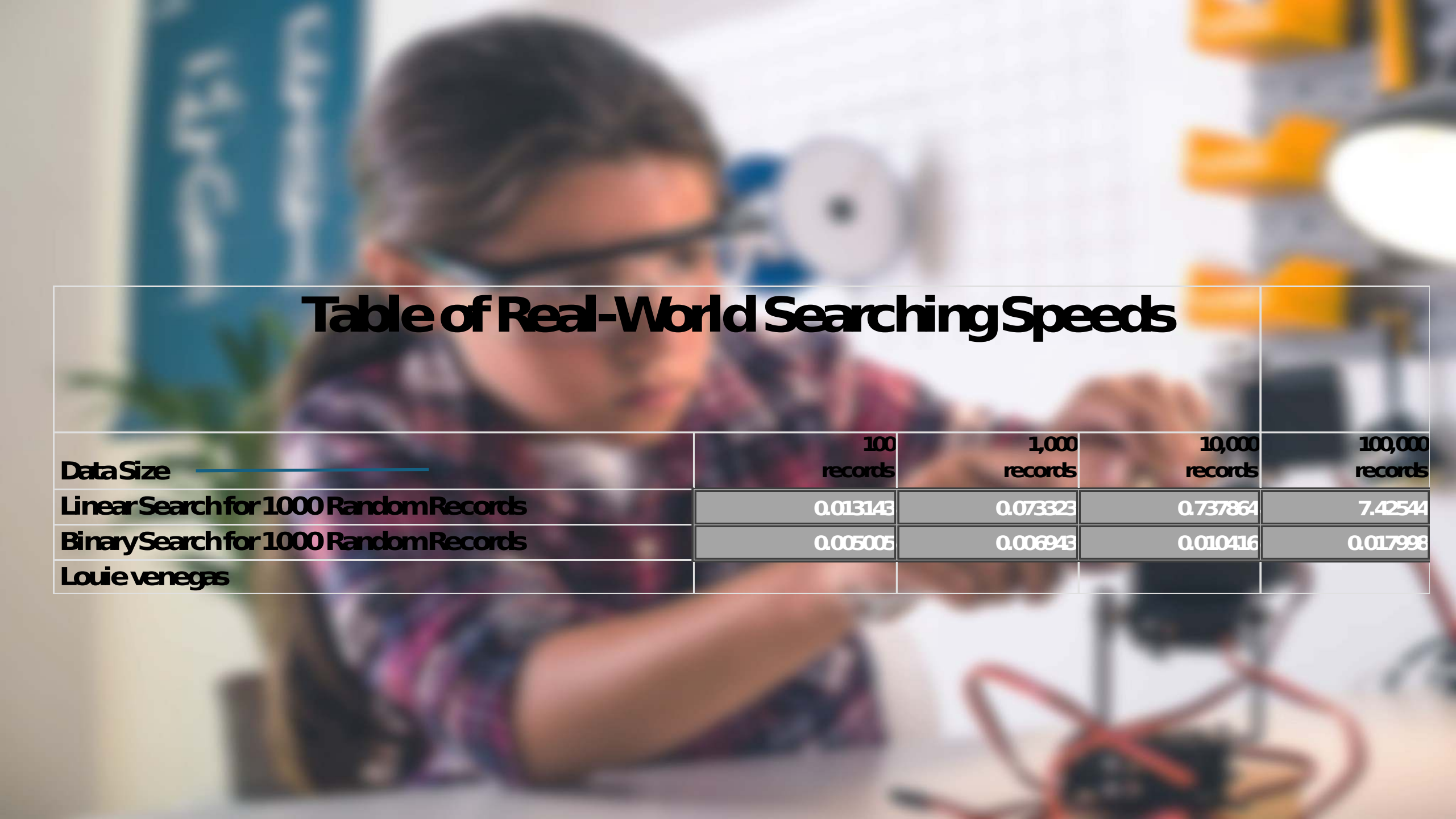


Table of Real-World Searching Speeds

Data Size	100 records	1,000 records	10,000 records	100,000 records
Linear Search for 1000 Random Records	0.013143	0.073323	0.737864	7.42544
Binary Search for 1000 Random Records	0.005005	0.006943	0.010416	0.017998
Louie venegas				

Binary Search Trees Real-World Speeds

- *To add Client data type to the BinarySearchTree*
- *To develop Python code that tests the BinarySearchTree's real world speeds*
- *To update an Excel table that displays relevant real-world speeds*

Introduction

In this section, we will enhance the functionality of a Binary Search Tree by incorporating a client-specific data type. We will test the real-world speeds of the Binary Search Tree, and record its performance. Additionally, we will update an Excel table to display these relevant real-world speeds, offering a clear and organized view of the data for analysis and decision-making.

Def BinarySearchTree:

A Binary Search Tree (BST) is an abstract data structure. It has various methods for inserting, removing, searching, and traversing nodes within the tree. A BST maintains the property that left child nodes are smaller than their parent, while right child nodes are larger. The class includes helper functions for operations like finding the minimum and maximum values, as well as displaying the tree in different orders (in-order, pre-order, and post-order).

```python
from Node import Node

class BinarySearchTree:
    # Assigns root with null (None).
    def __init__(self):
        # initialize the BST with a null root (empty).
        self.root = None

    # Checks to see if the BST is empty
    def isEmpty(self):
        # if the root is null, then the BST is empty
        return self.root is None

    # Inserts a new item into the binary search tree.
    def insert(self, data):
        # create a new node using the data
        newNode = Node(data)

        # is this a new tree?
        if self.isEmpty() == True:
            self.root = newNode
        else:
            # start at the top (root)
            current = self.root
            found = False     # did you find a position for the new node?

            # find an empty (null/None) position for the new node (where does i
            while found == False:
                if newNode.data < current.data:
                    if current.left is None:
                        current.left = newNode
                        found = True      # flip flag since position found!
                    else:
                        current = current.left    # go down left branch
```

Table of Real-World Speeds

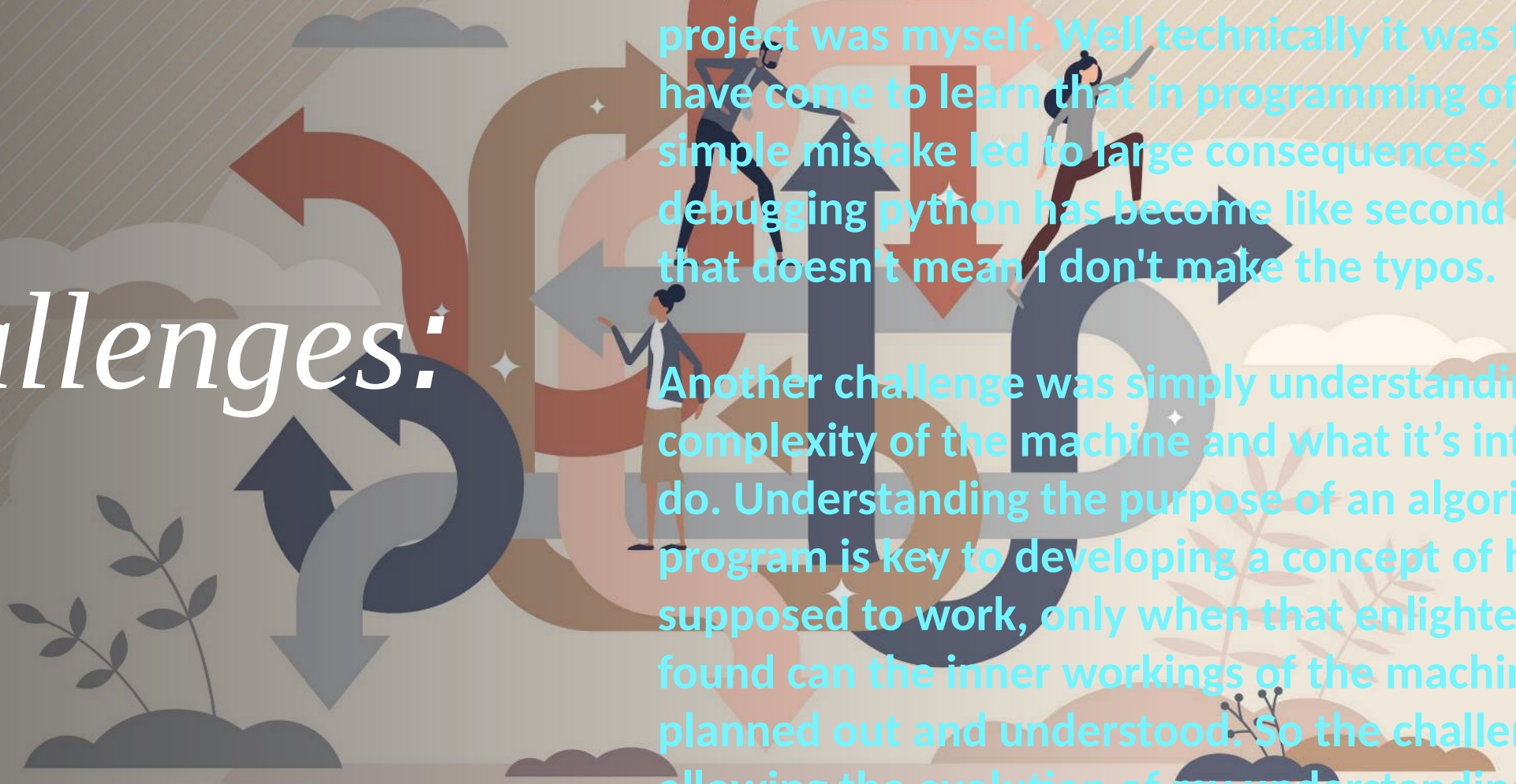
	ArrayList (unsorted)	ArrayList (sorted)	Linked list	Binary Search Tree
Scenario : Printer Queue or Call Service Queue				
Add many records to end of data structure:	0.000514	0.[illegible]978	0.005035	0.02668[illegible]
Pull all records off front of data structure	3.840341	0.[illegible]698	0.002516	0.[illegible]412[illegible]
Scenario : Customer Service Center				
Display random records	0.679132	0.[illegible]325	0.441979	0.047272
Scenario: Call Center				
Add some records, display random records, remove random records	2.403118	1.620621	0.85841	0.10975

Final Analysis

Throughout this project, we have explored various data structures and algorithms, applying them to real-life scenarios to understand their performance and efficiency. Our comprehensive analysis has yielded several key insights:

1. **ArrayList Performance**: Efficiently handled complex objects with client-specific data types. Python-based speed tests showed strong performance, with Excel results highlighting strengths and improvement areas.
2. **LinkedList Efficiency**: Versatile with a custom Client data type. Performance tests provided metrics on operations, emphasizing time complexity and efficiency. Excel visualizations highlighted performance across different operations and data sizes.
3. **Queue in Call Routing Systems**: Implemented Queue in a Call Routing System (ACD), demonstrating effectiveness in time-sensitive, sequential operations. Bridged theoretical knowledge with real-world application, solving industry-specific challenges efficiently.
4. **Sorting Algorithms Comparison**: Compared sorting algorithms using client data types to determine real-world speeds. Excel tables provided comprehensive analysis, highlighting the most efficient methods for various scenarios and data sets.
5. **Search Algorithms Analysis**: Analyzed linear and binary search algorithms on large datasets, revealing significant speed and efficiency differences. Emphasized binary search’s superior performance on sorted data.
6. **Binary Search Tree Enhancement**: Enhanced Binary Search Tree with client-specific data type, testing real-world speeds. Excel table displayed relevant speeds, aiding decision-making and showcasing efficiency

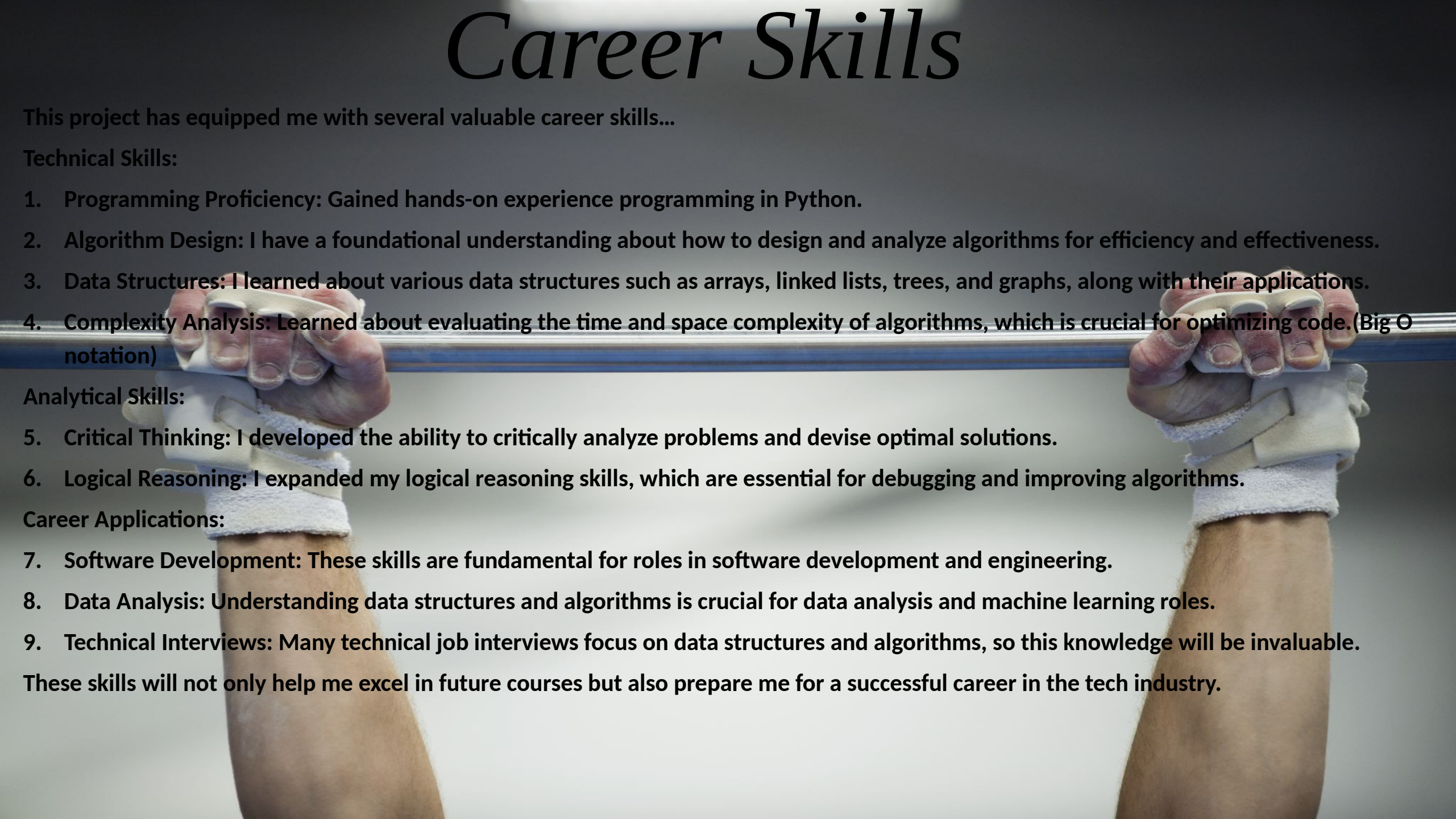
Challenges:



Programming in python has become somewhat easy not because I remember how to use python thoroughly but because I have learned to read it well. I said that to say that the main challenge I faced in this project was myself. Well technically it was typos. I have come to learn that in programming often our simple mistake led to large consequences. So, debugging python has become like second nature but that doesn't mean I don't make the typos.

Another challenge was simply understanding the complexity of the machine and what it's intended to do. Understanding the purpose of an algorithm or program is key to developing a concept of how it is supposed to work, only when that enlightenment is found can the inner workings of the machine be planned out and understood. So the challenge was allowing the evolution of my understanding develop as I learned about data structures and algorithms.

Career Skills



This project has equipped me with several valuable career skills...

Technical Skills:

1. **Programming Proficiency: Gained hands-on experience programming in Python.**
2. **Algorithm Design: I have a foundational understanding about how to design and analyze algorithms for efficiency and effectiveness.**
3. **Data Structures: I learned about various data structures such as arrays, linked lists, trees, and graphs, along with their applications.**
4. **Complexity Analysis: Learned about evaluating the time and space complexity of algorithms, which is crucial for optimizing code.(Big O notation)**

Analytical Skills:

5. **Critical Thinking: I developed the ability to critically analyze problems and devise optimal solutions.**
6. **Logical Reasoning: I expanded my logical reasoning skills, which are essential for debugging and improving algorithms.**

Career Applications:

7. **Software Development: These skills are fundamental for roles in software development and engineering.**
8. **Data Analysis: Understanding data structures and algorithms is crucial for data analysis and machine learning roles.**
9. **Technical Interviews: Many technical job interviews focus on data structures and algorithms, so this knowledge will be invaluable.**

These skills will not only help me excel in future courses but also prepare me for a successful career in the tech industry.

Conclusion

In conclusion, I find that some algorithms work better with certain data structures. The speed of the operations being performed varies greatly from one structure to another depending on the size of the input. Some algorithms are included in the project solely for educational purposes and are not commonly used in the industry. This distinction is important as it highlights the practical applications and limitations of various algorithms and data structures. The knowledge gained from this Data Structures and Algorithms class project provides a robust foundation of technical, analytical, and soft skills essential for a successful career in the tech industry. The problem-solving abilities, programming proficiency, and understanding of algorithm design and data structures are invaluable assets. Additionally, the critical thinking, logical reasoning, and collaboration skills developed will enhance my adaptability and attention to detail in various professional settings. These competencies not only prepare me for technical interviews and software development roles but also equipped me to excel in data analysis and machine learning fields. Ultimately, the knowledge and experience acquired from this class project will serve as a cornerstone for my future endeavors as a software engineer.