Table of Contents

Shriyansh Sapkota

Planning Process

When I initially read the project brief for this module, I instantly knew that I wanted to do an organisational structure over a family tree, content of a book or disk drive content as it was information I could import from a file. Additionally, I could deal with large sets of data to showcase my programming skills. I wanted to create an advanced application that went above the requirements on the brief yet remaining efficient. This document is a report of my programming journey.

Introduction

Initially I had to find the data, I contacted different senior leaders in BT who eventually pointed me towards the right direction. I tried understanding the excel file to see how I could manipulate the data in python to display only the necessary information

Data Set

[redacted]

Figure 1 – Small snippet of the excel file which includes all the data.

Figure 1 shows confidential employee information of real BT Employees. The CSV file includes all information of employees in the BT Group such as, employee name, location, number, email, manager etc. However, for my project I only need some information such as the name, UIN (Unique Identifier Number), business title, manager name and manager UIN. Additionally, as BT group is made up of a few companied including BT, EE, Openreach, Plus Net, I had to try think of a way only BT employees are recorded, or I would have multiple tree structures for multiple organisations. This file includes 180,000 employee details.

Employee Class

```python
class Organisational_Structure_TreeNode:
    def __init__(self,em_uin,em_name,em_role):
        self.em_uin = em_uin
        self.em_name = em_name
        self.em_role = em_role
        self.manages_list = []
        self.parent = None
```

I started the development of the application by creating a class for an employee object these tree node objects can display specific data I want. The class takes in the parameters: Employee UIN, Employee Name and Employee Role which are necessary.

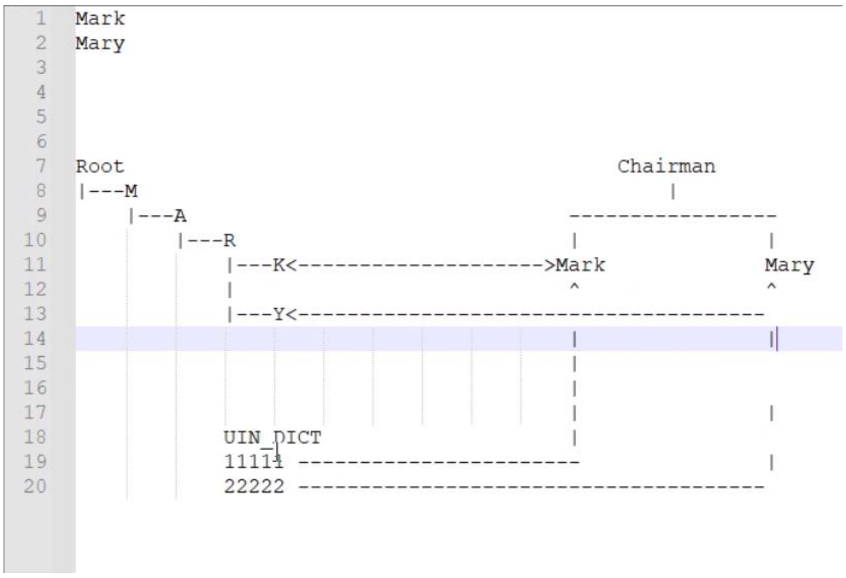
```python
# set_role defines value for role attribute of given node. Uses encapsulation.
def set_role(self,role):
    self.em_role = role

# append_child creates biderectional relationships between manager and their employees.
def append_child(self,child):
    child.parent = self
    self.manages_list.append(child)

# method displays name and business title of child nodes.
def print_manages_list(self):
    print("\nManages:")
    for employees in self.manages_list:
        print(employees.em_name, "-", employees.em_role)

# method displays relevant data of current node.
def display_employee_data(self):
    print("\nName:", self.em_name)
    print("Employee Unique Identifier:", self.em_uin)
    print("Job Title:", self.em_role)
    print("Manager:",self.parent.em_name)
    self.print_manages_list()
    print("\n")
```

I also started coding the methods I would need for my application. I didn't code all the methods in one sitting and just started creating them throughout my journey when it was applicable and necessary. The set role is necessary later in my program when reading from a file as when an employee node with manager is created, role isn't defined, this function adds the missing values to the node. Additionally, encapsulation is used which ensures data is protected from unwanted access from clients and also reduces human errors that disrupt the program.
The append child method is used to add an employee node as a child of the current node, this is what creates the parent child relationship which is necessary to form the organisational structure.
The remaining two methods are used to output data onto the console and make it look nice and readable for the user of the program. Print_manages_list method displays the child nodes of the current node by outputting the child's name and uin. Display_employee_data displays the data of the employee such as name, uin, job title, manager. It also calls the previous function which is used to print all the people they manage.

Searching Tree Node Class

```python
# Defining a clasee with methods and attributes that builds/accesses a tree structure.
# Instances of these classes are used to search for employees and job roles efficiently.
# Nodes point to a memory location which hold all information of the employee.
class Searching_TreeNode:
    def __init__(self,character,em_info):
        self.character = character
        self.children = {}
        self.em_info = []
```

This is another class that I have used for my application. The purpose of this class is to build another tree structure which will be used to search for names of employees and the job role efficiently. It takes in the parameters “Character” a string and “em_info” which is a list that contains locations for nodes in the Organisational tree structure. It also has a private attribute called children which is a dictionary that contains its children.



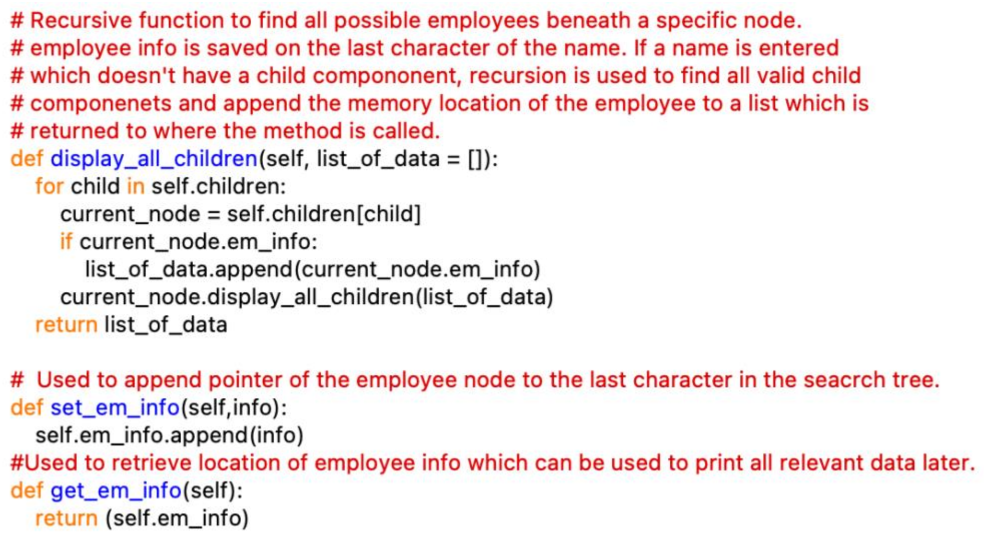
```
Mark
Mary




Root                                        Chairman
|---M                                           |
    |---A                                  ------------------
        |---R                               |               |
            |---K<-------------------->Mark             Mary
            |                           ^               ^
            |---Y<---------------------------------------
                                        |               |
                                        |
                                        |
                                        |               |
            UIN_DICT                    |
            11111 ------------------------              |
            22222 ---------------------------------------
```

Let’s take the above picture as an example, the tree on the left begins with root and has a direct child of M. when the word “MARK” is searched, it follows the children of each character. The last character also contains the memory location of Mark in the organisational tree, which is accessed, and the employee information is displayed. This is how my Searching_TreeNode works. Alternatively, if Mary is searched then the same path is used except the last character is different, the Y in Mary will hold the memory location of “Mary” in the organisational tree. I believe this was the most efficient way to sort my data and be able to access them by searching it. Regardless of how big the data set is (mine being 180,000 lines) data is searched instantly and the program doesn’t need to go through each line of the data set to find the information you’re looking for. With most searching algorithms, as the data set increases, the searching of individual data increases proportionally or disproportionally depending on the algorithms BIG O notation.

```
# Method creates bidirectional relationship between consecutive charecters of a name.
# E.g if name is Mark. M is parent class of a. A is parent class of r etc.
def set_child(self,child_node):
    child_node.parent = self

# Method used to search for a child component of a node.
# If child component not found, then it creates one, thus building a tree
def build_tree(self,character):
    if (character not in self.children):
        character_node = Searching_TreeNode(character,None)
        self.children[character] = character_node
    return self.children[character]

# Method used to check if a node is a child of the self parent class.
# Used to traverse the tree and find names on the tree. If not found, returns false
def traverse_tree(self,character):
    if (character in self.children):
        current_node = self.children[character]
        return current_node
    else:
        return False
```

Shriyansh Sapkota

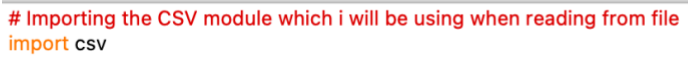
These are some of the methods of the class. The set_child method, like from the organisational class is used to append the child node to the parent node.
The build tree method is used to build the actual tree by searching if the nodes for an employee or a role are already in the tree, if they aren't then its created and placed in the tree.
The traverse tree method is used to search through nodes of the tree, this method is called by another function in the program when searching through the tree. It returns either the node that was searched for or false if it's not found – you can do a lot with this information which I will explain later in this document.



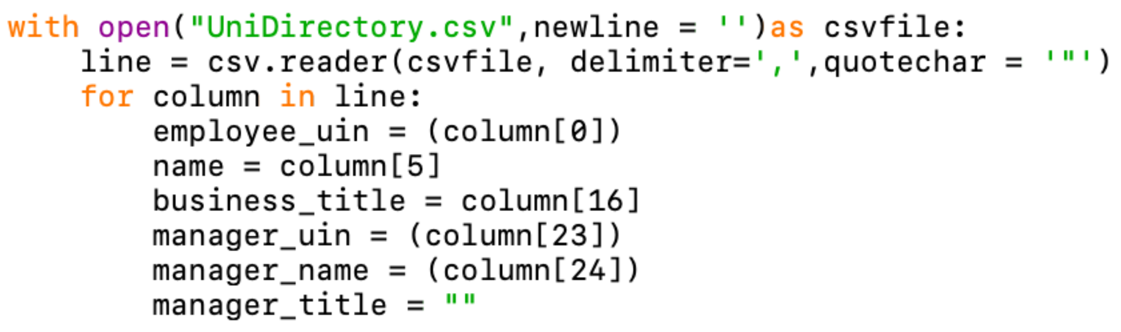
```
# Recursive function to find all possible employees beneath a specific node.
# employee info is saved on the last character of the name. If a name is entered
# which doesn't have a child compononent, recursion is used to find all valid child
# componenets and append the memory location of the employee to a list which is
# returned to where the method is called.
def display_all_children(self, list_of_data = []):
    for child in self.children:
        current_node = self.children[child]
        if current_node.em_info:
            list_of_data.append(current_node.em_info)
        current_node.display_all_children(list_of_data)
    return list_of_data

#  Used to append pointer of the employee node to the last character in the seacrch tree.
def set_em_info(self,info):
    self.em_info.append(info)
#Used to retrieve location of employee info which can be used to print all relevant data later.
def get_em_info(self):
    return (self.em_info)
```

Even though it wasn't asked of me, I decided to create a recursive function that searches through the tree to print suggested searches. For example, if you search Shri, the program will return with: Shri not found, did you mean Shriyansh Sapkota? I have done this using some functions in my program and the display all children which is a recursive method that calls itself and displays all children that carry employee information and append it to the list. The list is returned and parsed at a later point in the program to do this.
I have also used getters and setters to get/set employee information to nodes in this tree.

Reading from a file

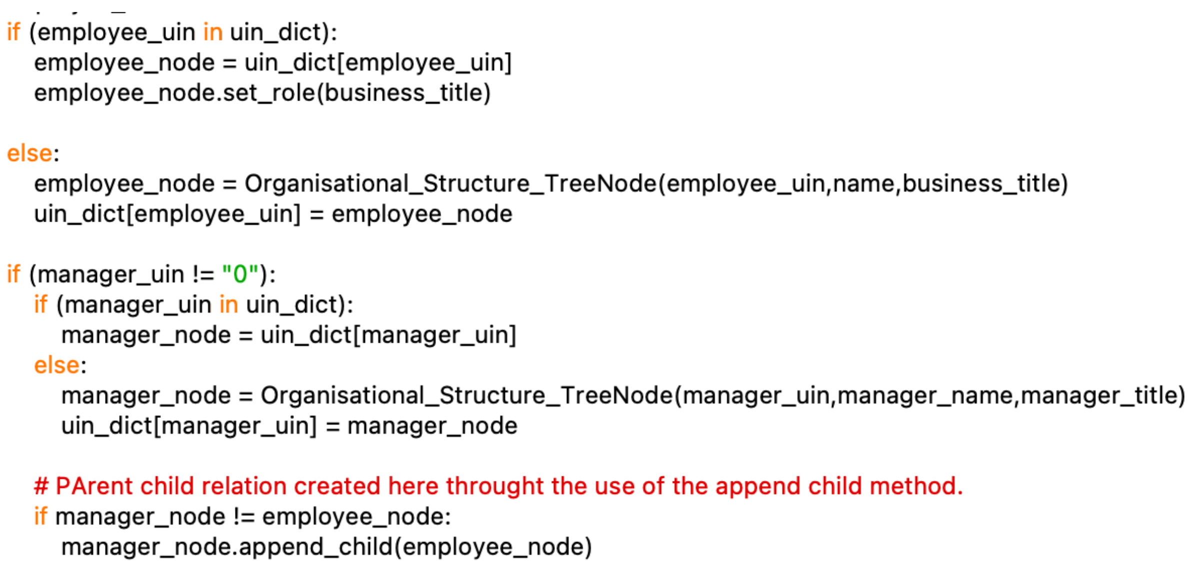


```
# Importing the CSV module which i will be using when reading from file
import csv
```



```
with open("UniDirectory.csv",newline = '')as csvfile:
    line = csv.reader(csvfile, delimiter=',',quotechar = '"')
    for column in line:
        employee_uin = (column[0])
        name = column[5]
        business_title = column[16]
        manager_uin = (column[23])
        manager_name = (column[24])
        manager_title = ""
```

CSV library is imported at the start of my program to make use of its pre coded functions which I can use when handling the file.This is the code from the main part of my program which opens the CSV file and sets specific columns of data into variables. The csv file is opened with ‘,’ as the delimiter and ‘ “ ’ as the quote char. The columns are normally separated with a comma, however in the business title column there are some strings with commas in them. The delimiter and quote char ensures that the column isn’t separated at wrong points. Initially this was a problem I had where I didn’t include a delimiter and the business title would be separated mid string.

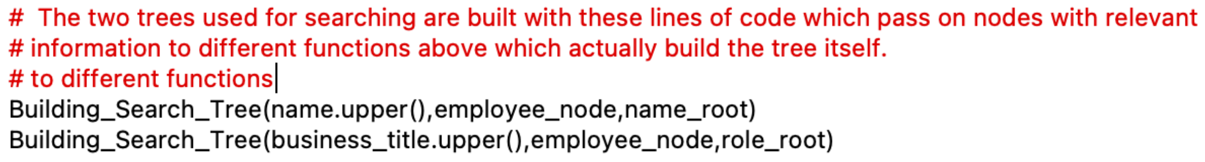


```
if (employee_uin in uin_dict):
    employee_node = uin_dict[employee_uin]
    employee_node.set_role(business_title)

else:
    employee_node = Organisational_Structure_TreeNode(employee_uin,name,business_title)
    uin_dict[employee_uin] = employee_node

if (manager_uin != "0"):
    if (manager_uin in uin_dict):
        manager_node = uin_dict[manager_uin]
    else:
        manager_node = Organisational_Structure_TreeNode(manager_uin,manager_name,manager_title)
        uin_dict[manager_uin] = manager_node

    # PArent child relation created here throught the use of the append child method.
    if manager_node != employee_node:
        manager_node.append_child(employee_node)
```

The if statements that check that employee uin is not equal to 0 is placed there because of the data set that I use. The CSV file, as stated above, includes employee information of all employees in the BT group. However, only BT employees have a UIN, other companies in the group are identified by a person’s Id. This if statement ensures that only a organisational structure of BT employees is used rather than creating multiple trees with organisational structure of each company.

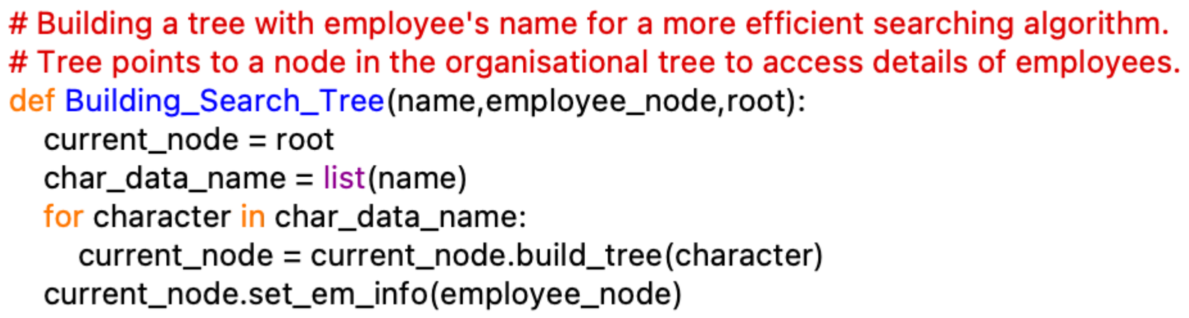
The second if and else statement check to see if the employee and the manager on the line of the csv file are in the uin dict. If they are not, then a node is created for them and saved onto the dictionary. However, if they are then the node information is retrieved from the dictionary and business role is updated. Another if statement is used towards the end to see that a manager node is not the same as the employee node. This is to prevent loops in recursive algorithm. If a manager and employee are the same, the program will go in an endless loop printing the organisational structure, this selection statement prevents that. If the case isn’t true then, the manager node appends the employee node as a child case. This builds the parent child relation between nodes in my application

Building Search Trees



```
#  The two trees used for searching are built with these lines of code which pass on nodes with relevant
# information to different functions above which actually build the tree itself.
# to different functions
Building_Search_Tree(name.upper(),employee_node,name_root)
Building_Search_Tree(business_title.upper(),employee_node,role_root)
```

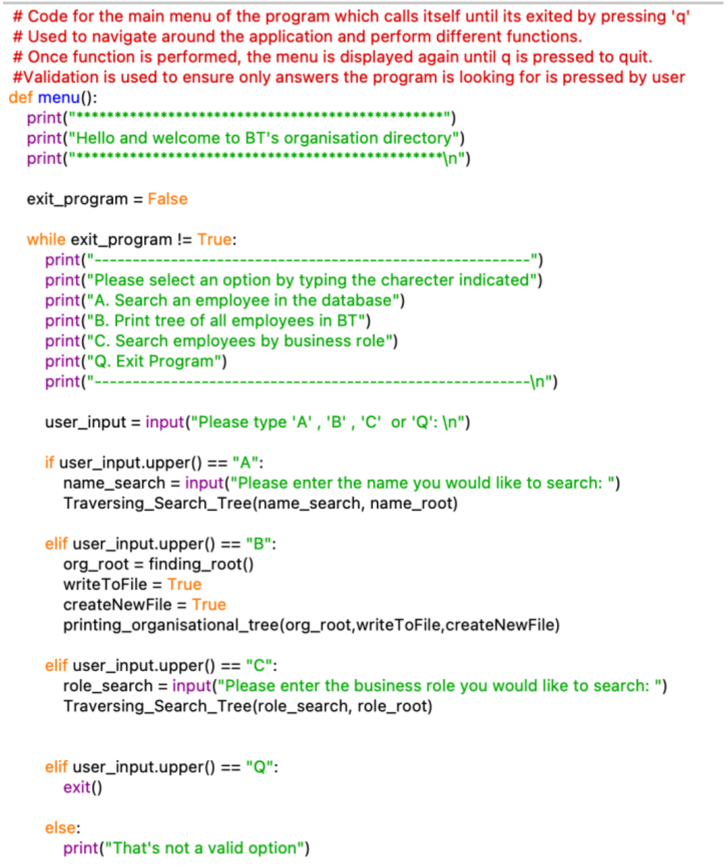
The two lines of code is used to build the two search trees, these lines of code start of the whole process by calling on another function and passing parameters to them which include the name/ business title respectively, employee node and the root of the tree. The name and business title are passed in the upper case to make the application more user friendly



```
# Building a tree with employee's name for a more efficient searching algorithm.
# Tree points to a node in the organisational tree to access details of employees.
def Building_Search_Tree(name,employee_node,root):
   current_node = root
   char_data_name = list(name)
   for character in char_data_name:
      current_node = current_node.build_tree(character)
   current_node.set_em_info(employee_node)
```

Building search tree points the current node as root and splits the name of employee or role and puts it into a list containing each character. This way the consecutive character can be added as a child of the previous one. For statement is used to go through the list and add each character to the tree assigning consecutive character as the child of the current node. The current node is then updated to the child node which has just been built into the tree. For explanation of the build_tree method, please refer to the earlier part of this documenting going over the Searching_TreeNode class.
E.g.: if Shriyansh is passed in as the name, S would be a child of root, H would be a child of S etc.

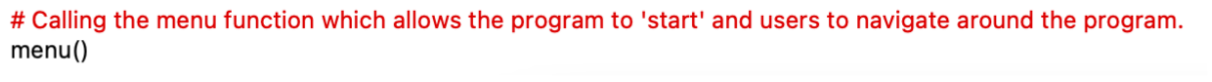
Menu System



```
# Code for the main menu of the program which calls itself until its exited by pressing 'q'
# Used to navigate around the application and perform different functions.
# Once function is performed, the menu is displayed again until q is pressed to quit.
#Validation is used to ensure only answers the program is looking for is pressed by user
def menu():
   print("************************************************")
   print("Hello and welcome to BT's organisation directory")
   print("************************************************\n")

   exit_program = False

   while exit_program != True:
      print("--------------------------------------------------------")
      print("Please select an option by typing the charecter indicated")
      print("A. Search an employee in the database")
      print("B. Print tree of all employees in BT")
      print("C. Search employees by business role")
      print("Q. Exit Program")
      print("--------------------------------------------------------\n")

      user_input = input("Please type 'A' , 'B' , 'C'  or 'Q': \n")

      if user_input.upper() == "A":
         name_search = input("Please enter the name you would like to search: ")
         Traversing_Search_Tree(name_search, name_root)

      elif user_input.upper() == "B":
         org_root = finding_root()
         writeToFile = True
         createNewFile = True
         printing_organisational_tree(org_root,writeToFile,createNewFile)

      elif user_input.upper() == "C":
         role_search = input("Please enter the business role you would like to search: ")
         Traversing_Search_Tree(role_search, role_root)


      elif user_input.upper() == "Q":
         exit()

      else:
         print("That's not a valid option")
```

The block of code on the right is the code for my menu system which is used to navigate around the application and perform different actions for the user. The actions include Searching an employee in BT, printing an organisation structure (which writes to file in the same file location of the program itself), searching employees by their job role and finally exiting the program. Apart from when the application is exit, after the program does an action, it returns to the main menu and prompts the user for another input. Validation through selection statements and a while loop is used to ensure that the users input is processed, if the correct input is entered then it runs the respective action. However, if there's an incorrect input then the program prompts for another input from the user.

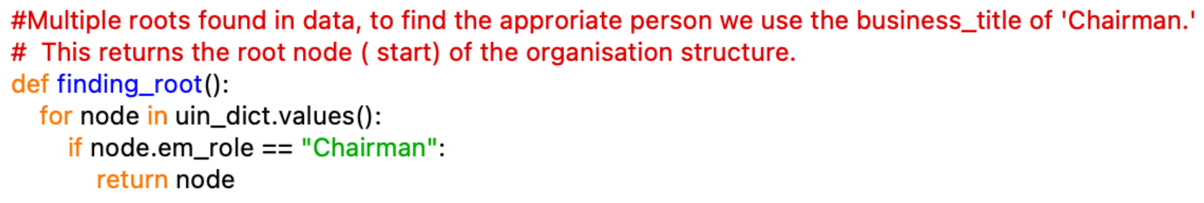
Additionally, a single line of code that calls the menu function runs the whole program.



```
# Calling the menu function which allows the program to 'start' and users to navigate around the program.
menu()
```

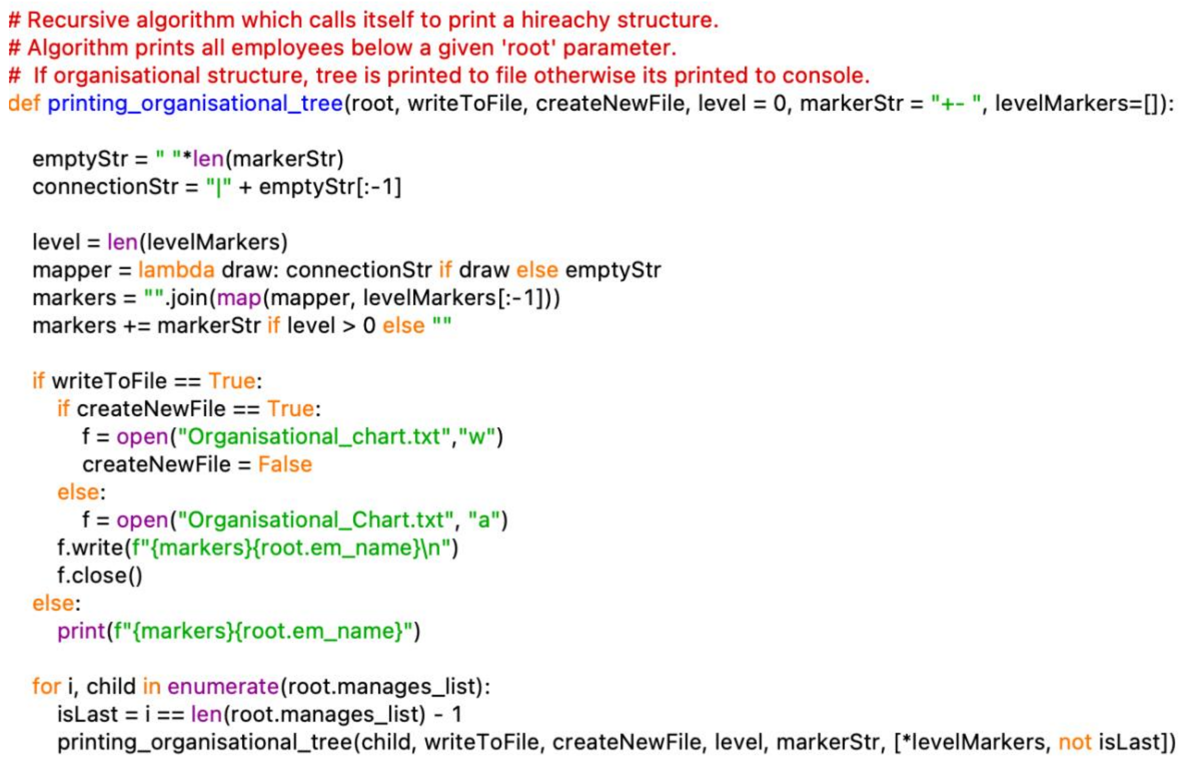
Printing the organisational structure

When selecting option B to print the organisational structure, the finding_root function is called. The function is as follows:



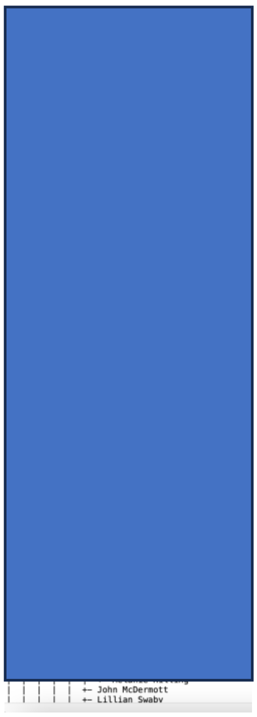
```
#Multiple roots found in data, to find the approriate person we use the business_title of 'Chairman.'
#  This returns the root node ( start) of the organisation structure.
def finding_root():
    for node in uin_dict.values():
        if node.em_role == "Chairman":
            return node
```

The function checks the uin dictionary for someone with the role of Chairman as there can only be one chairman in the company and they're at the top of the organisation structure. This returns the node with the chairman. Two parameters called write to file and create new file are set to true and the three variables are passed to the printing organisational tree function. The function already takes in some parameters which have a default value if not passed.



```
# Recursive algorithm which calls itself to print a hireachy structure.
# Algorithm prints all employees below a given 'root' parameter.
#  If organisational structure, tree is printed to file otherwise its printed to console.
def printing_organisational_tree(root, writeToFile, createNewFile, level = 0, markerStr = "+- ", levelMarkers=[]):

    emptyStr = " "*len(markerStr)
    connectionStr = "|" + emptyStr[:-1]

    level = len(levelMarkers)
    mapper = lambda draw: connectionStr if draw else emptyStr
    markers = "".join(map(mapper, levelMarkers[:-1]))
    markers += markerStr if level > 0 else ""

    if writeToFile == True:
        if createNewFile == True:
            f = open("Organisational_chart.txt","w")
            createNewFile = False
        else:
            f = open("Organisational_Chart.txt", "a")
        f.write(f"{markers}{root.em_name}\n")
        f.close()
    else:
        print(f"{markers}{root.em_name}")

    for i, child in enumerate(root.manages_list):
        isLast = i == len(root.manages_list) - 1
        printing_organisational_tree(child, writeToFile, createNewFile, level, markerStr, [*levelMarkers, not isLast])
```

This is the function that is used to write to file. It has multiple functions as its also called to print all direct and indirect children of a node in the program by printing to console for other parts of the program such as when getting information about an employee. When this function is called after entering B on the menu. If there is already a file in file location called Organisational_Chart.txt then this file creates a new one to ensure its an updated one and appends the file with employees. It uses a depth first search to print the structure. I wrote it to a file as it took 45 minutes to print to console which is too long for the user, however it writes to the file in less than 20 seconds.



[redacted]

```
| | | | | +- John McDermott
| | | | | +- Lillian Swaby
```

The function checks what level of the org structure nodes are in, this is how the correct markers are drawn to file and how the names are indented. The picture on the left is a snippet of the org chart from the top of the organisation that is written to the file.

Similarly when the function is called and write to file and create new file are set as false, the indirect and direct children of the employee is printed but to the console as shown below:

```
Direct and Indirect Children Tree:
Finlay Fraser
+- Ashiqur Rahman
+- David Thomas
+- Mark Blewett
| +- Shriyansh Sapkota
+- Alex Lalic
```

Searching for name or business role:

When A or C is entered in the menu similar things happen as they both search the string within their respective trees. Both call Traversing Search Tree function with the string to search and the root of the tree which was defined earlier at the start of reading from a file. The root determines what tree to look at.

```
# Function to search through the name tree structure and provide details of the BT Employee.
# Initially, string split into individual characters for name search tree.
# Theres two cases within function which is determined by Searching_role - whether an employee
#  is being searched or if a business role is being searched. Both use this function.
#  method traverse_tree is called from Searching_TreeNode class. Returned value is parsed to display correct info.
def Traversing_Search_Tree(name, root_node):
    character_list = list(name.upper())
    current_node = root_node

    if root_node == role_root:
        Searching_role = True
    else:
        Searching_role = False

    for character in character_list:
        if current_node!= False:
            current_node = current_node.traverse_tree(character)

    # If False is passed from traverse tree method then employee/role not found and function ends.
    if current_node == False:
        if Searching_role:
            print("\nThis role doesnt exist")
        else:
            print("\nThe person is not employed to BT")

        print("Returning to the main menu\n")
```

There is a variable in the functioned called searching role which is set true for searching the job role and false if searching for an employee. Depending on what's being searched different blocks of code are run. The function splits the passed string into individual characters and puts them in a list. For each character in the list the traverse tree method is called which follows the list from 1 to the end character to see if it's a valid path on the search tree. If it isn't a valid path and there are no more children on in the current node then a message is displayed to the user saying the role or employee don't exist.

```
 # Block of code used if the name the person searched for is a suitable name. Points to a function which displays
 #  Information about the name or the role user is searching for.
else:
    current_node_list = current_node.get_em_info()
    print("Your search came with the following results:\n")
    if Searching_role:
         displaying_role_search_info(current_node_list,name)
    else:
        displaying_name_search_info(current_node_list)
```

However if a valid employee or job role is entered then another function is called which displays the data respectively.

To display and employee the following function is called and displays this way to the user:

```python
#  Function used to display information about the employee searched. Incorporates other functions to display
#  organisational tree and uses method called display_employee_data to siplay all data of the employee
def displaying_name_search_info(current_node):
    for x in current_node:
        x.display_employee_data()
        print("Direct and Indirect Children Tree:")
        printing_organisational_tree(x,writeToFile = False, createNewFile = False)
```

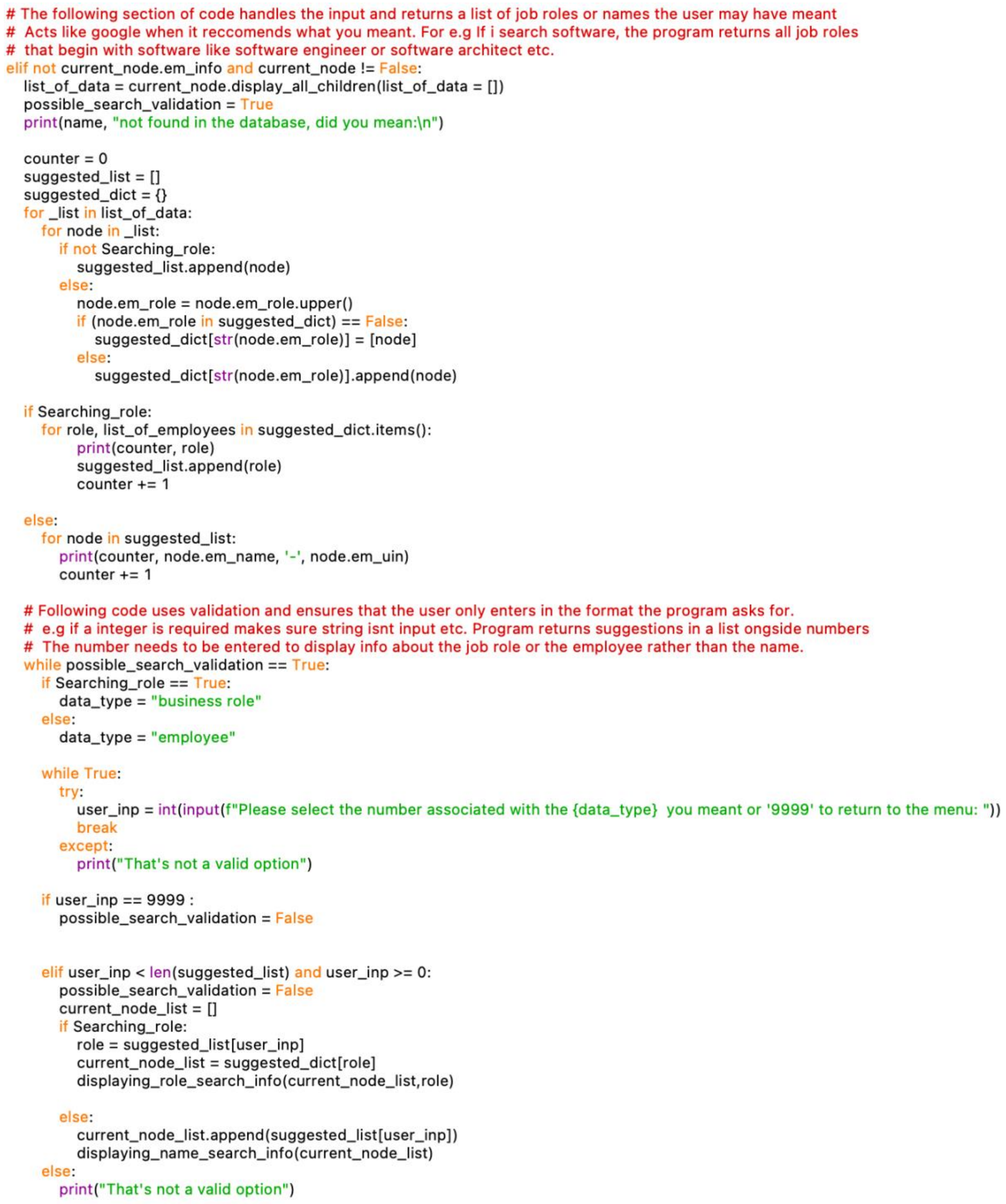
```
Please enter the name you would like to search: finlay fraser
Your search came with the following results:


Name: F[redacted]
Employe
Job Titl
Manage

Manage
Ashiqur                  ngineer
David T                  ant
Mark Bl                  ervices
Alex Lal


Direct a
Finlay F
+- Ashi
+- Davi
+- Mark
|  +- Shr
+- Alex
```

To display a business role the following function is called and displays this way to the user:

```python
#  Function used to display information of a role searched like who works as the role and the total count.
def displaying_role_search_info(current_node, name):
    counter = 0
    print("\n")
    for x in current_node:
        counter += 1
        print(x.em_name, "-", x.em_uin)

    print("~~~~~~~~~~~~~~~~~~~~")
    print("Total",name,":",counter)
    print("~~~~~~~~~~~~~~~~~~~~")
```

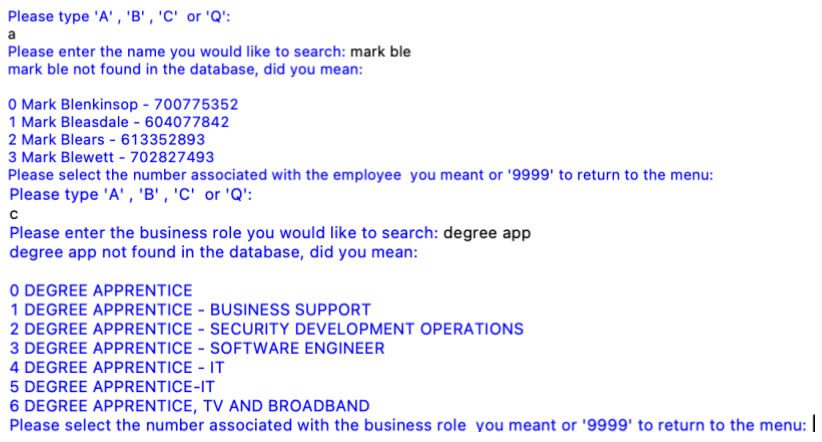
```
Hollie Keen - [redacted]
Harry Chiche
Zain Nazar - (
~~~~~~~~~~~~~~~~~~~~
Total DEGREE APPRENTICE : 3
~~~~~~~~~~~~~~~~~~~~
```

The block of code below shows another case that the traverse tree method can return. If a name is searched and the method doesn't return false, but also doesn't return a node that is assigned with employee information then it calls another method of the class which checks all children node of the current node. Every time a node has employee information, that information is appended to a list. It does a depth first search on the remainder of the tree to find possible employees or business role. Depending on what you're searching for, the suggested list is displayed to the user who can pick what they meant or type '9999' to return to the main menu. Once the number is entered, the functions above are called to display the data respectively.



```
# The following section of code handles the input and returns a list of job roles or names the user may have meant
#  Acts like google when it reccomends what you meant. For e.g If i search software, the program returns all job roles
#  that begin with software like software engineer or software architect etc.
elif not current_node.em_info and current_node != False:
    list_of_data = current_node.display_all_children(list_of_data = [])
    possible_search_validation = True
    print(name, "not found in the database, did you mean:\n")

    counter = 0
    suggested_list = []
    suggested_dict = {}
    for _list in list_of_data:
        for node in _list:
            if not Searching_role:
                suggested_list.append(node)
            else:
                node.em_role = node.em_role.upper()
                if (node.em_role in suggested_dict) == False:
                    suggested_dict[str(node.em_role)] = [node]
                else:
                    suggested_dict[str(node.em_role)].append(node)

    if Searching_role:
        for role, list_of_employees in suggested_dict.items():
                print(counter, role)
                suggested_list.append(role)
                counter += 1

    else:
        for node in suggested_list:
            print(counter, node.em_name, '-', node.em_uin)
            counter += 1

    # Following code uses validation and ensures that the user only enters in the format the program asks for.
    #  e.g if a integer is required makes sure string isnt input etc. Program returns suggestions in a list ongside numbers
    #  The number needs to be entered to display info about the job role or the employee rather than the name.
    while possible_search_validation == True:
        if Searching_role == True:
            data_type = "business role"
        else:
            data_type = "employee"

        while True:
            try:
                user_inp = int(input(f"Please select the number associated with the {data_type}  you meant or '9999' to return to the menu: "))
                break
            except:
                print("That's not a valid option")

        if user_inp == 9999 :
            possible_search_validation = False


        elif user_inp < len(suggested_list) and user_inp >= 0:
            possible_search_validation = False
            current_node_list = []
            if Searching_role:
                role = suggested_list[user_inp]
                current_node_list = suggested_dict[role]
                displaying_role_search_info(current_node_list,role)

            else:
                current_node_list.append(suggested_list[user_inp])
                displaying_name_search_info(current_node_list)
        else:
            print("That's not a valid option")
```

Shriyansh Sapkota



```
Please type 'A' , 'B' , 'C'  or 'Q':
a
Please enter the name you would like to search: mark ble
mark ble not found in the database, did you mean:

0 Mark Blenkinsop - 700775352
1 Mark Bleasdale - 604077842
2 Mark Blears - 613352893
3 Mark Blewett - 702827493
Please select the number associated with the employee  you meant or '9999' to return to the menu:
Please type 'A' , 'B' , 'C'  or 'Q':
c
Please enter the business role you would like to search: degree app
degree app not found in the database, did you mean:

0 DEGREE APPRENTICE
1 DEGREE APPRENTICE - BUSINESS SUPPORT
2 DEGREE APPRENTICE - SECURITY DEVELOPMENT OPERATIONS
3 DEGREE APPRENTICE - SOFTWARE ENGINEER
4 DEGREE APPRENTICE - IT
5 DEGREE APPRENTICE-IT
6 DEGREE APPRENTICE, TV AND BROADBAND
Please select the number associated with the business role  you meant or '9999' to return to the menu:
```

Conclusion:

To conclude, I really enjoyed this assignment and wanted to push myself to ensure that I create a program that is better than what is asked for from the assignment brief. I think this project went really well as I managed to tick off all the requirements and also added additional things like the suggested employee/ business role code. Additionally, I managed to use an efficient way to sort my data and handle large sets of data.

If I could do this again, I would try design a GUI like the one in Microsoft teams to have a better user interface and make it more user friendly. Additionally, I would add more functions that display the percentage of different job roles etc. Another thing I would like to implement is more data from the csv files so I could showcase something else such as the amount of people in different departments of the business.

Bibliography

Adam Smith, N/A. *How to edit a file in Python.* [Online]
Available at: https://www.adamsmith.haus/python/answers/how-to-edit-a-file-in-python
[Accessed 08 04 2022].
Python, N/A. *CSV File Reading and Writing.* [Online]
Available at: https://docs.python.org/3/library/csv.html?highlight=csv#csv-fmt-params
[Accessed 25 03 2022].
Python, N/A. *Input and Output.* [Online]
Available at: https://docs.python.org/3/tutorial/inputoutput.html
[Accessed 05 04 2022].
Simon, 2020. *Python 3: Recursively print structured tree including hierarchy markers using depth-first search.* [Online]
Available at: https://simonhessner.de/python-3-recursively-print-structured-tree-including-hierarchy-markers-using-depth-first-search/
[Accessed 16 03 2022].
Weber, B., N/A. *Python Enumerate(): Simplifying Looping With Counters.* [Online]
Available at: https://realpython.com/python-enumerate/
[Accessed 16 03 2022].