Lab 1 – Python Basics-1 (Data Structures)

Tuples

Tuples are a built-in data structure in Python that are similar to lists, but with some key differences. Tuples are immutable, meaning their values cannot be changed once they are created. They are also usually used to store related values, as they allow you to group data together in a single object.

```
# Creating a tuple
my_tuple = (1, 2, 3, 4)

# Accessing elements in a tuple
print(my_tuple[0]) # Output: 1
print(my_tuple[-1]) # Output: 4

# Slicing a tuple
print(my_tuple[1:3]) # Output: (2, 3)

# Tuple concatenation
new_tuple = my_tuple + (5, 6)
print(new_tuple) # Output: (1, 2, 3, 4, 5, 6)

# Tuple repetition
print(my_tuple * 3) # Output: (1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4)
```

List

Lists are a built-in data structure in Python that are used to store an ordered collection of items. Lists are mutable, meaning their values can be changed after they are created. They can contain elements of different types, including other lists.

```
# Creating a list
my_list = [1, 2, 3, 4]

# Accessing elements in a list
print(my_list[0]) # Output: 1
print(my_list[-1]) # Output: 4

# Slicing a list
print(my_list[1:3]) # Output: [2, 3]

# List concatenation
new_list = my_list + [5, 6]
print(new_list) # Output: [1, 2, 3, 4, 5, 6]

# List repetition
print(my_list * 3) # Output: [1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

Data Dictionary

A data dictionary is a collection of descriptions of the variables in a dataset, including their names, types, and other characteristics. In Python, you can use a dictionary to store this information.

```
data_dictionary = {
    "variable_1": {"type": "string", "description": "Name of the person"},
    "variable_2": {"type": "integer", "description": "Age of the person"},
    "variable_3": {"type": "float", "description": "Height of the person in meters"}
}

def add_variable(variable_name, variable_type, description):
    data_dictionary[variable_name] = {"type": variable_type, "description": description}

def update_variable(variable_name, key, value):
    if variable_name in data_dictionary:
        data_dictionary[variable_name][key] = value
    else:
        print("Error: Variable not found in data dictionary.")

def delete_variable(variable_name):
    if variable_name in data_dictionary:
        del data_dictionary[variable_name]
    else:
        print("Error: Variable not found in data dictionary.")

def get_variable_info(variable_name):
    if variable_name in data_dictionary:
        return data_dictionary[variable_name]
    else:
        print("Error: Variable not found in data dictionary.")
        return None
```

Python code for stack implentation.

1. The Stack class initializes an empty list self.items to store the stack items.
2. The push method adds an item to the end of the list self.items, which represents the top of the stack.
3. The pop method removes and returns the last item from the list self.items. If the list is empty, it returns None.
4. The peek method returns the last item from the list self.items without removing it. If the list is empty, it returns None.
5. The is_empty method returns True if the list self.items is empty, and False otherwise.

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
```

```
        return self.items.pop() if self.items else None

    def peek(self):
        return self.items[-1] if self.items else None

    def is_empty(self):
        return not self.items
```

Implementation of a queue in Python using a list

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)

    def is_empty(self):
        return len(self.queue) == 0

    def size(self):
        return len(self.queue)
```

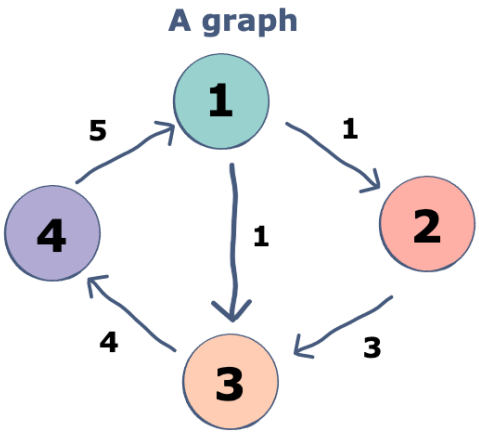
Lab Questions:

Q1. Implementation of a queue in Python using two stacks.

Description: A queue can be implemented using two stacks in Python by following the below steps:

1. Use two stacks, **stack1** and **stack2**, to implement the enqueue and dequeue operations.
2. In the enqueue operation, push the new element onto **stack1**.
3. In the dequeue operation, if **stack2** is empty, transfer all elements from **stack1** to **stack2**. The element at the top of **stack2** is the first element that was pushed onto **stack1** and thus represents the front of the queue. Pop this element from **stack2** to return it as the dequeued element.

Q2. Implement the following graph using python. Print the adjacency list and adjacency matrix.
[A graph is a data structure that consists of vertices that are connected via edges.]



Adjacency list	Adjacency matrix
{"1" : [[2, 1], [3,1]], "2" : [[3, 3]], "3" : [[4, 4]], "4" : [[1, 5]]}	[[0, 1, 1, 0], [0, 0, 3, 0], [0, 0 ,0 , 4], [5, 0, 0 ,0]]

Q3. Create two list X and Y with some set of numerical values. Compute Euclidean distance for corresponding values in X and Y. Store the distance values in a separate list and sort them using Bubble sort algorithm.

Formula

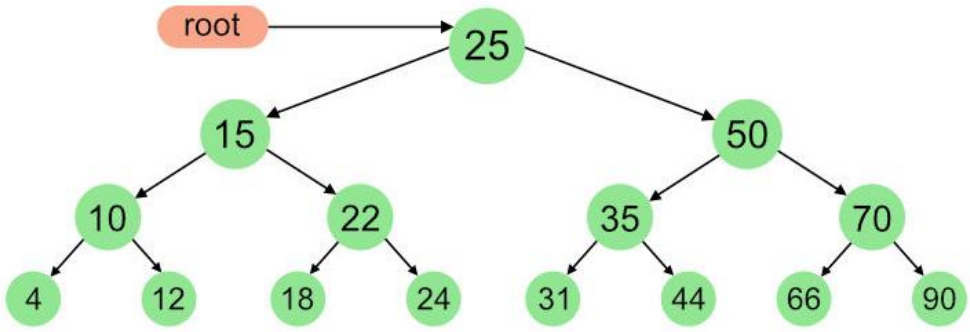
$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

p, q = two points in Euclidean n-space

q_i, p_i = Euclidean vectors, starting from the origin of the space (initial point)

n = n-space

Q4. Implement the given binary search tree using Python and print the pre-order, in-order, and post-order tree traversal.



Expected Output:

InOrder(root) visits nodes in the following order:

4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25