Project 2
Project Title: Implementation of Data Structures *in Python*

Introduction:

The purpose of this assignment is to implement various data structures in Python. Understanding data structures is fundamental to programming as they organize and facilitate access to data, regardless of the programming paradigm in use.

(Qa) Identifying and Describing Data Structures Used in Programming:

Data Structures:

Data structures are fundamental components of programming that facilitate the organization and management of data. Here is a comprehensive list of commonly used data structures in programming:

1. Arrays:

- Arrays are collections of elements stored in contiguous memory locations, allowing access to elements by index.
- Time Complexity:
- Access: O(1)
- Insertion/Deletion: O(n) (worst case)

2. Linked Lists:

- Linked lists are linear data structures where each element (node) contains data and a reference to the next node.
- Time Complexity:
- Insertion/Deletion: O(1)
- Access: O(n)

3. **Stacks:**
   - Stacks are linear data structures that follow the Last In, First Out (LIFO) principle.
   - Time Complexity:
   - Insertion/Deletion: O(1)

4. **Queues:**
   - Queues are linear data structures that follow the First In, First Out (FIFO) principle.
   - Time Complexity:
   - Insertion/Deletion: O(1)

5. **Binary Search Trees:**
   - Binary search trees are hierarchical data structures where each node has at most two children.
   - Time Complexity:
   - Insertion/Deletion/Searching: O(log n) (average case)

6. **Graphs:**
   - Graphs are non-linear data structures consisting of vertices and edges. Implementation details depend on the specific type of graph representation.
   - Time Complexity: Depends on the specific operations and graph representation.

7. **Hash Tables:**
   - Hash tables are data structures that store key-value pairs and use a hash function for efficient data retrieval.
   - Time Complexity:
   - Insertion/Deletion/Searching: O(1) (average case), O(n) (worst case)

(Qb) Developing Data Structures in a Programming Language:

For the purpose of this assignment, we choose Python as the programming language to implement these data structures. Python is known for its simplicity and readability, making it suitable for demonstrating data structures.

I'll provide an outline for each data structure and then proceed with the implementation in Python.

1. Arrays:

```python
class Array:
    def __init__(self):
        self.array = []

    def insert(self, element):
        self.array.append(element)

    def delete(self, index):
        del self.array[index]

    def access(self, index):
        return self.array[index]

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

This is a basic implementation of an array class in Python.

2. Linked Lists:

```python
```

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

    def delete(self, data):
        current = self.head
        if current.data == data:
            self.head = current.next
            return
        while current.next:
            if current.next.data == data:
                current.next = current.next.next
                return
            current = current.next

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" ")
            current = current.next
```

```python
        print()
```

This is a basic implementation of a singly linked list in Python.

3. Stacks:

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

    def push(self, data):
        self.stack.append(data)

    def pop(self):
        if len(self.stack) > 0:
            return self.stack.pop()
        else:
            return None

    def peek(self):
        if len(self.stack) > 0:
            return self.stack[-1]
        else:
            return None

    def is_empty(self):
        return len(self.stack) == 0
```

This is a basic implementation of a stack using a Python list.

4. Queues:

```python
class Queue:
```

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

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

    def dequeue(self):
        if len(self.queue) > 0:
            return self.queue.pop(0)
        else:
            return None

    def peek(self):
        if len(self.queue) > 0:
            return self.queue[0]
        else:
            return None

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

This is a basic implementation of a queue using a Python list.

5. Trees (Binary Search Tree):
```python
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

class BinarySearchTree:
    def __init__(self):
        self.root = None
```

```
    def insert(self, data):
        if self.root is None:
            self.root = TreeNode(data)
        else:
            self._insert_recursive(self.root, data)

    def _insert_recursive(self, node, data):
        if data < node.data:
            if node.left is None:
                node.left = TreeNode(data)
            else:
                self._insert_recursive(node.left, data)
        elif data > node.data:
            if node.right is None:
                node.right = TreeNode(data)
            else:
                self._insert_recursive(node.right, data)

    def search(self, data):
        return self._search_recursive(self.root, data)

    def _search_recursive(self, node, data):
        if node is None or node.data == data:
            return node
        if data < node.data:
            return self._search_recursive(node.left, data)
        return self._search_recursive(node.right, data)
```

This is a basic implementation of a binary search tree in Python.

6. Graphs (Adjacency List Representation):

```python
class Graph:
```

```python
    def __init__(self):
        self.graph = {}

    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, vertex1, vertex2):
        if vertex1 in self.graph and vertex2 in self.graph:
            self.graph[vertex1].append(vertex2)
            self.graph[vertex2].append(vertex1)

    def display(self):
        for vertex in self.graph:
            print(vertex, "->", " -> ".join(map(str, self.graph[vertex])))
```

This is a basic implementation of a graph using an adjacency list representation in Python.

7. Hash Tables (Dictionary in Python):

```python
class HashTable:
    def __init__(self):
        self.table = {}

    def insert(self, key, value):
        self.table[key] = value

    def search(self, key):
        return self.table.get(key)

    def delete(self, key):
        if key in self.table:
            del self.table[key]
```

```

This is a basic implementation of a hash table using Python's built-in dictionary.

## Code Organization:

- Created separate Python files for each data structure: `array.py`, `linked_list.py`, `stack.py`, `queue.py`, `binary_search_tree.py`, `graph.py`, `hash_table.py`.
- Placed each data structure implementation in its respective file.
- Created a main script (`main.py`) to demonstrate the usage of each data structure.

## Testing:

- Wrote test cases to validate the functionality of each data structure.
- Ensured test cases covered various scenarios, including edge cases and boundary conditions.

## Submission:

- Compiled all documentation (explanations, code comments, and test cases) into a single document (PDF).

## Quality Assurance:

- Double-checked code for errors, bugs, or inconsistencies.
- Ensured code followed best practices in terms of readability, efficiency, and adherence to Python coding conventions.
- Proofread documentation for clarity, correctness, and completeness.
```