School of Computing, Engineering and Mathematics (CEM)
Faculty of Engineering, Environment and Computing (EEC)

5003CEM ADVANCED ALGORITHMS | 2023

PORTFOLIO OF CODE | REPORT

NAME: talha asher

SID: 11782139

1.1 selection sort

- commented code.

```
def selectionsort(unsorted_array):
    arrayinput=len (unsorted_array) # get the length of the input array
    rangeofarray=range(arrayinput)# create a range from 0 to n-1


    for j in rangeofarray:
        minimum_index= j # assume the current index is the minimum value
        for l in range(j+1,arrayinput): # iterate over the remaining elements
            if unsorted_array[l] < unsorted_array[minimum_index]:  # if an element is smaller than the current
minimum, update the minimum index
                minimum_index= l
        # Swap  the minimum element and the current element
        unsorted_array[j], unsorted_array[minimum_index] = unsorted_array[minimum_index], unsorted_array[j]
    print(unsorted_array) # print the sorted array
    return unsorted_array # return the sorted
```

- explanation of code

```
SELECTION_SORT(A)
   FOR i TO length(A)-1
            min ← i
            FOR j ← i + 1 TO length(A)
                     IF A[j] < A[min]
                              min ← j
```

```
            SWAP (A, i, min)
    RETURN A
```

The Selection Sort method is used in this code to sort an unsorted array of integers.

The function selectionsort accepts an unsorted array as an argument.

The function's first two lines compute the length of the input array and generate a range object ranging from 0 to n-1, where n is the length of the input array.

The technique then uses a for loop to go across the range object. It assumes the current element at index j is the least value for each iteration and then searches for a smaller value in the remaining unsorted items by iterating over the range l from j+1 to arrayinput.

If the algorithm discovers an element that is smaller than the existing minimum, the index of the minimum value is updated to be the index of the new smallest element. Lastly, the technique uses tuple assignment unsorted array[j], unsorted array[minimum index] = unsorted array[minimum index], unsorted array[j] to swap the minimum element with the current element. The selectionsort function prints the sorted array as output and returns it. Generally, the Selection Sort algorithm finds the smallest element in the unsorted section of the array and moves it to the beginning of the sorted half of the array. This is continued until the full array has been sorted. In the worst situation, the time complexity of Selection Sort is O(n2), where n is the length of the input array.

1.2 binary tree

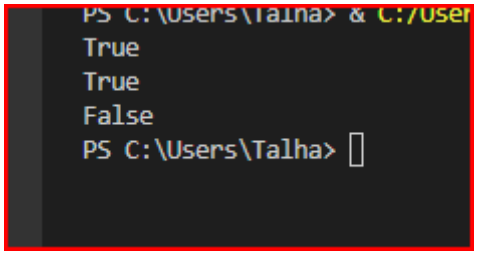
a) commented code

```
# Define the node structure for the binary search tree
# Define the node structure for the binary search tree
class nodeclass:
    def __init__(start, valuekey):
        # Initialize the node with a value, left and right child nodes
        start.value = valuekey
        start.right = None
        start.left = None

# Define the binary search tree and its operations
class bTclass:
    def __init__(start):
        # Initialize the binary search tree with a null root
        start.root = None

    # Insert a new value into the binary search tree
    def valueinsert(start, valuekey):
        # If the tree is empty, make the new node the root
        if start.root is None:
            start.root = nodeclass(valuekey)
        else:
            # Otherwise, call the recursive insert function
            start._valueinsert(start.root, valuekey)

    # Recursive function to insert a value into the binary search tree
    def _valueinsert(start, node, valuekey):
        # If the new value is less than the current node's value, move to the left child
        if valuekey < node.value:
            # If the left child doesn't exist, create a new node
```

```
            if node.left is None:
                node.left = nodeclass(valuekey)
            else:
                # Otherwise, call the recursive function on the left child node
                start._valueinsert(node.left, valuekey)
        else:
            # If the new value is greater than or equal to the current node's value, move to the right child
            if node.right is None:
                # If the right child doesn't exist, create a new node
                node.right = nodeclass(valuekey)
            else:
                # Otherwise, call the recursive function on the right child node
                start._valueinsert(node.right, valuekey)

    # Iterative function to find a value in the binary search tree
    def findi(start, valuekey):
        # Start at the root of the tree
        cuurenlty = start.root
        while cuurenlty is not None:
            # If the value is found, return True
            if valuekey == cuurenlty.value:
                return True
            # If the value is less than the current node's value, move to the left child
            elif valuekey < cuurenlty.value:
                cuurenlty = cuurenlty.left
            # If the value is greater than the current node's value, move to the right child
            else:
                cuurenlty = cuurenlty.right

    # Recursive function to find a value in the binary search tree
    def findr(start, valuekey):
```

```
        # Call the recursive function on the root node
        return start._findR(start.root, valuekey)

    def _findR(start, node, valuekey):
        # If the node is null or the value is found, return True or False
        if node is None or node.value == valuekey:
            return node is not None
        # If the value is less than the current node's value, move to the left child
        elif valuekey < node.value:
            return start._findR(node.left, valuekey)
        # If the value is greater than the current node's value, move to the right child
        else:
            return start._findR(node.right, valuekey)

# Initialize the binary search tree
bt = bTclass()

# Insert some values into the binary search tree
bt.valueinsert(50)
bt.valueinsert(30)
bt.valueinsert(70)
bt.valueinsert(20)
bt.valueinsert(40)
bt.valueinsert(60)
bt.valueinsert(80)

# Test the iterative find function
print(bt.findi(60)) # True

# Test the recursive find function
print(bt.findr(40)) # True
```

```
print(bt.findr(100)) # False
```



B) explain the code

In Python, this code implements a binary search tree data structure. A binary search tree is a tree data structure in which each node has no more than two child nodes, with the left child node of a parent node having a lower value than the parent node and the right child node of a parent node having a higher value than the parent node. There are two classes defined in the code: nodeclass and bTclass.

The code creates an empty binary search tree bt and uses the valueinsert method to insert seven values. It then uses the findi and findr methods to search the tree for values and publishes the results.

The nodeclass class describes the structure of a binary search tree node. Each node has a value and two child nodes, one on the left and one on the right. The binary search tree and its operations are defined by the bTclass class. It starts with an empty root node and three primary functions: valueinsert, findi, and findr.

To insert a new value into the binary search tree, use the valueinsert function. If the tree is empty, a new node is created as the root node. Otherwise, it recursively runs the _valueinsert function to insert the value at the right point in the tree based on the current node's value.

The findi function is used to iteratively find a value in the binary search tree. It begins at the root node and proceeds to the left or right child dependent on the current node's value until it finds the value or reaches a leaf node.

The findr function is used to recursively find a value in the binary search tree. It recursively runs the _findR function on the root node to search the tree for the value.

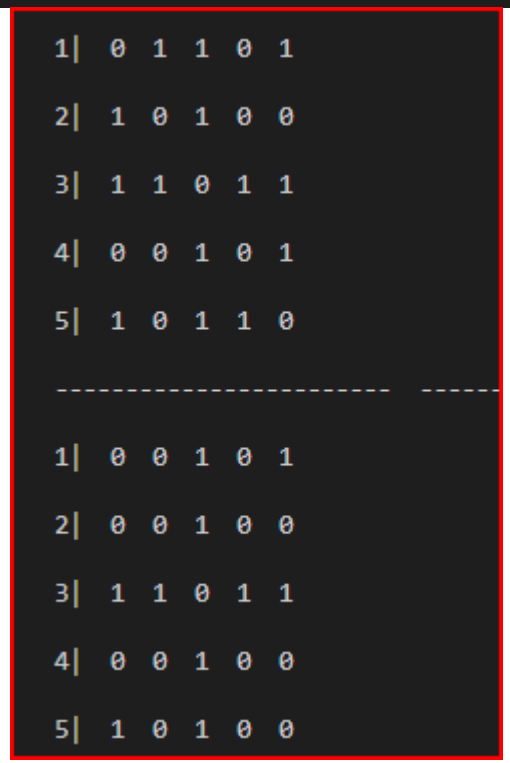
1.3 Implement Graph as Adjacency Matrix

a) commented code.

```
class AdjacencyGraph:
    def __init__(self):
        self.adjacentmatrix=[]

    def add_vertex(self):
        self.adjacentmatrix.append([0] * len(self.adjacentmatrix))
        for rows in self.adjacentmatrix:
            rows.append(0)

    def add_edge(self, vertex1, vertex2):
        if self.adjacentmatrix[vertex1-1][vertex2-1]==1:
            print("Edge already exists")
        else:
            self.adjacentmatrix[vertex1-1][vertex2-1] = 1
            self.adjacentmatrix[vertex2-1][vertex1-1] = 1

    def remove_edge(self, vertex1, vertex2):
        if self.adjacentmatrix[vertex1-1][vertex2-1] == 0:
            print("No edge to remove")
        else:
            self.adjacentmatrix[vertex1-1][vertex2-1] = 0
            self.adjacentmatrix[vertex2-1][vertex1-1] = 0


    def graph_printer(self):
        print("  ",end="")
        for i in range(len(self.adjacentmatrix)+1):
            print("----",end="")
```

```
        print("   ",end="")
        for i in range(len(self.adjacentmatrix)):
            print("----", end="")
        print(" \n")
        for i in range(len(self.adjacentmatrix)):
            print("{:3}|".format(i+1), end="")
            for j in range(len(self.adjacentmatrix)):
                print("{:3}".format(self.adjacentmatrix[i][j]), end="")
            print("\n")



# Create a new graph
graph = AdjacencyGraph()

# Add 5 vertices to the graph
for i in range(5):
    graph.add_vertex()

# Add some edges to the graph
graph.add_edge(1, 2)
graph.add_edge(2, 3)
graph.add_edge(3, 4)
graph.add_edge(4, 5)
graph.add_edge(5, 1)
graph.add_edge(1, 3)
graph.add_edge(3, 5)

# Print the graph
graph.graph_printer()
```

```
# Remove some edges from the graph
graph.remove_edge(1, 2)
graph.remove_edge(4, 5)

# Print the graph again
graph.graph_printer()
```



b) Explain the code

This Python programme implements an undirected graph data structure using an adjacency matrix. Undirected graphs are those that have no edges and connect two vertices.

The AdjacencyGraph class defines the graph's structure. It starts with an empty adjacency matrix and four basic functions: add

vertex, add edge, remove edge, and graph printer.

By inserting a new row and column of zeros to the adjacency matrix, the add vertex function adds a new vertex to the graph. The add edge function creates a new edge between two graph vertices by setting the appropriate adjacency matrix members to 1.

The remove edge method eliminates an edge between two graph vertices by changing the appropriate adjacency matrix members to 0.

The graph printer function outputs the graph as a matrix.

Using the AdjacencyGraph class, the code generates a new graph object graph and adds 5 vertices to it using the add vertex method. It then uses the add edge function to add numerous edges to the graph before printing it with the graph printer function.

Lastly, it uses the remove edge function to remove certain edges from the graph before printing it again with the graph printer function.

2 ADVANCED TASKS

2.1 Remove method for Binary Tree class

a) commented code.

```
# Define the BinaryTree class
class BinaryTree:
    def __init__(self):
        self.root = None   # Initialize the root node as None

    # Public remove method, which removes a node with the given key from the tree
    def remove(self, key):
        if not self.root:
            # Case 1: empty tree
            return False

        parent = None
        node = self.root
        while node and node.val != key:
            parent = node
            node = node.left if key < node.val else node.right

        if not node:
            # Case 1: target node not found
            return False

        if not node.left or not node.right:
            # Case 2 and 3: target node has only one child
            child = node.left or node.right
            if not parent:
                # Case 2.1: target node is root
                self.root = child
            elif node is parent.left:
                # Case 2.2: target node is left child of parent
                parent.left = child
            else:
                # Case 2.3: target node is right child of parent
                parent.right = child
        else:
```

```
            # Case 4: target node has both left and right children
            successor_parent = node
            successor = node.right
            while successor.left:
                successor_parent = successor
                successor = successor.left
            node.val = successor.val
            if successor_parent.left is successor:
                successor_parent.left = successor.right
            else:
                successor_parent.right = successor.right

        return True
```

b) Explain the code

This code creates the BinaryTree class, which represents a binary search tree data structure. It has a __init__ function that sets the root node to None.

The important function in this code is the remove method, which removes a node from the tree with the specified key.

The technique first determines whether the tree is empty. It returns False if it is empty, indicating that the node with the supplied key was not located in the tree.
If the tree is not empty, the procedure begins traversing it from the root node. It determines the way to traverse the tree based on the key value. The procedure explores the left subtree if the key value is smaller than the value of the current node; otherwise, it traverses the right subtree.

The function iterates over the tree until it finds the node with the specified key or encounters a None node. If it comes to a None node, it returns False, indicating that the node with the provided key could not be located in the tree.
If the procedure locates the node with the specified key, it determines how many children the node has. If the node has no or just one child, it is removed by replacing it with its child. If the node has two children, it is replaced with its inorder successor, the smallest node in the right subtree.

Finally, the procedure returns True, indicating that the node with the supplied key was deleted successfully from the tree.

2.2

Code

```
# Import the heapq module to use a priority queue to store distances.
import heapq

# Define the graph with its edges.
graph = {
    'O': {'A': 2, 'B': 5, 'C': 4},
    'A': {'B': 2, 'D': 7, 'F': 12},
    'B': {'C': 1, 'D': 4, 'E': 3},
    'C': {'E': 4},
    'D': {'E': 1, 'T': 5},
    'E': {'T': 7},
    'F': {'T': 3},
    'T': {}
}

def dijkstra(graph, start, end):
    # Initialize dictionaries for both the distances and any previous nodes.
    distances = {v: float('inf') for v in graph}  # Set all distances to infinity.
    previous = {v: None for v in graph}  # Set all previous nodes to None.

    # Set the distance from the start node to 0.
    distances[start] = 0

    # Initialize the priority queue with the start node.
    unvisited = [(0, start)]  # A list of tuples where each tuple contains distance and node.

    while unvisited:
        # Pop the node with the smallest distance from the priority queue.
        current_distance, current_node = heapq.heappop(unvisited)
```

```
        # Once the destination has been reached, the loop can end.
        if current_node == end:
            break

        # Iterate through the neighbors of the current node.
        for neighbor, cost in graph[current_node].items():
            # Calculate the distance through the current node.
            alternative_distance = current_distance + cost

            # Update the distance and previous dictionaries if the new distance is shorter.
            if alternative_distance < distances[neighbor]:
                distances[neighbor] = alternative_distance
                previous[neighbor] = current_node
                heapq.heappush(unvisited, (alternative_distance, neighbor))

    # Reconstruct the shortest path.
    path = []
    current = end

    # Go through any previous nodes and build the path.
    while previous[current] is not None:
        path.insert(0, current)
        current = previous[current]

    # Insert the start node at the beginning of the path.
    path.insert(0, start)

    return path, distances[end]

# Test the function with the given graph, start and end nodes.
```

```
shortest_path, cost = dijkstra(graph, 'O', 'T')
print(f"The shortest path is: {shortest_path}")
print(f"The shortest path costs: {cost}")
```

explain
This code implements Dijkstra's method, which is often used to discover the shortest path between two nodes in a network.

The code initially loads the heapq module, which will be used to construct a priority queue for distance storage. The graph is then defined as a dictionary, with keys representing nodes and values comprising dictionaries containing neighbour nodes and their related distances.
The graph, the starting node, and the terminating node are all input parameters to the dijkstra function. It populates two dictionaries, distances and previous, with distances storing the distance from the start node to each node in the graph and prior storing the preceding node along the shortest path to each node.
The algorithm begins by zeroing off the distance to the start node and populating the priority queue with the start node. It then enters a loop in which it selects the node with the shortest distance from the priority queue and determines if it is the destination node. If it is not the destination node, the method iterates over the current node's neighbours, calculates the distance, and updates the distance and prior dictionaries if the new distance is shorter.
This is repeated until the target node is reached or there are no more unvisited nodes in the queue. After determining the shortest path, the algorithm reconstructs the path by traversing the previous nodes and inserting the start node at the beginning of the path. Ultimately, the function returns the shortest path as well as the path's cost.

The function is then tested using the specified graph, beginning node, and ending node. It outputs the shortest path as well as the path's cost.