Candidate name: Daren (NJC)

Centre number: NA

Index number: NA

Programming language used: Python 3

| **Question 1** |
| --- |
| **Evidence 1**  def exists(current\_list, target):  for i in range(len(current\_list)):  if target == current\_list[i]:  return i  return -1  Inventory = []  ItemTypes = []  ItemCounts = []  f = open("INVENTORY.TXT")  for line in f:  current\_item = line.strip()  Inventory.append(current\_item)  current\_index = exists(ItemTypes, current\_item)  if current\_index == -1:  ItemTypes.append(current\_item)  ItemCounts.append(1)  else:  ItemCounts[current\_index] += 1  f.close()  print("{0:<20}{1:<8}\n".format("ItemType", "Count"))  for i in range(len(ItemTypes)):  print("{0:<20}{1:<8}".format(ItemTypes[i], ItemCounts[i])) |
| **Evidence 2** |
| **Question 2** |
| **Evidence 3**  **A: CheckDigit 🡨 CalCheckDigit(newNumber, Total)**  **B: RETURN "X"**  **C: RETURN CONCATENATE(Number, CheckDigit)** |
| **Evidence 4**  def CalCheckDigit(Number, Total):  if len(Number) > 1:  Digit = int(Number[0])  Total = Total + (Digit \* (len(Number) + 1))  newNumber = Number[1:]  CheckDigit = CalCheckDigit(newNumber, Total)  else: # len(Number) == 1  Digit = int(Number[0])  Total = Total + (Digit \* (len(Number) + 1))  CalcModulus = Total % 11  CheckValue = 11 - CalcModulus  if CheckValue == 11:  return "0"  elif CheckValue == 10:  return "X"  else:  return str(CheckValue)  if len(Number) == 9:  return Number + CheckDigit  else:  return CheckDigit    f = open("ISBNPRE.TXT")  rawISBNs = []  for line in f:  rawISBNs.append(line.strip())  print(CalCheckDigit(rawISBNs[-1], 0)) |
| **Evidence 5** |

| **Question 3** |
| --- |
| **Evidence 6**  class ConnectionNode():  def \_\_init\_\_(self):  self.\_DataValue = ""  self.\_LeftChild = 0  self.\_RightChild = 0  def GetDataValue(self):  return self.\_DataValue  def SetDataValue(self, newValue):  self.\_DataValue = newValue  def GetLeftChild(self):  return self.\_LeftChild  def SetLeftChild(self, newLeft):  self.\_LeftChild = newLeft  def GetRightChild(self):  return self.\_RightChild  def SetRightChild(self, newRight):  self.\_RightChild = newRight  class LinkedList():  def \_\_init\_\_(self):  # note: index zero holds a dummy ConnectionNode instance  self.\_RobotData = [None] + [ConnectionNode() for x in range(25)]  for i in range(1, len(self.\_RobotData) - 1):  self.\_RobotData[i].SetLeftChild(i + 1)  self.\_Root = 1  self.\_NextFreeChild = 1  myLL = LinkedList() |

| **Evidence 7**  def FindNode(self, NodeValue):  Found = False  CurrentPosition = self.\_Root  while not Found and CurrentPosition <= 25:  if self.\_RobotData[CurrentPosition].GetDataValue() == \  NodeValue:  Found = True  else:  CurrentPosition += 1  if CurrentPosition > 25:  return 0  else:  return CurrentPosition  def AddToRobotData(self, NewDataItem, ParentItem, ThisMove):  # Add link: ParentItem -> newDataItem (on ThisMove of ParentItem)  # ThisMove should only be: "L", "R" or "X"  if self.\_Root == 1 and self.\_NextFreeChild == 1: # is empty  # update NextFreeChild  self.\_NextFreeChild = \  self.\_RobotData[self.\_NextFreeChild].GetLeftChild()  # update new RobotData node  self.\_RobotData[self.\_Root].SetLeftChild(0)  self.\_RobotData[self.\_Root].SetDataValue(NewDataItem)  else:  # check if parent exists  ParentPosition = self.FindNode(ParentItem)  if ParentPosition > 0: # parent exists  # check if child exists  ExistingChild = self.FindNode(NewDataItem)  if ExistingChild > 0: # child exists  ChildPointer = ExistingChild  else:  ChildPointer = self.\_NextFreeChild  self.\_NextFreeChild = self.\_RobotData[ \  self.\_NextFreeChild].GetLeftChild()  self.\_RobotData[ChildPointer].SetLeftChild(0)  self.\_RobotData[ChildPointer].SetDataValue(NewDataItem)  if ThisMove == "L":  self.\_RobotData[ParentPosition].\  SetLeftChild(ChildPointer)  else:  self.\_RobotData[ParentPosition].\  SetRightChild(ChildPointer) |
| --- |

| **Evidence 8**  def OutputData(self):  print("Root: " + str(self.\_Root))  print("NextFreeChild: " + str(self.\_NextFreeChild))  print("RobotData:")  print("{0:<8}{1:<8}{2:<8}{3:<8}".format("Index", "Data", \  "Left", "Right"))  for i in range(1, len(self.\_RobotData)):  print("{0:<8}{1:<8}{2:<8}{3:<8}".format(i,  self.\_RobotData[i].GetDataValue(), \  self.\_RobotData[i].GetLeftChild(), \  self.\_RobotData[i].GetRightChild())) |
| --- |
| **Evidence 9**  myLL = LinkedList()  f = open("SEARCHTREE.TXT")  for line in f:  line = line.strip().split(",")  myLL.AddToRobotData(line[0], line[1], line[2])  myLL.OutputData() |
| **Evidence 10** |

| **Evidence 11**  def DoPreOrderTraversal(self, CurrentIndex, PreviousPath):  Current = self.\_RobotData[CurrentIndex]  newPath = PreviousPath + Current.GetDataValue()  if Current.GetDataValue() == "Z":  print(newPath)  if Current.GetLeftChild() != 0:  self.DoPreOrderTraversal(Current.GetLeftChild(), newPath)  if Current.GetRightChild() != 0:  self.DoPreOrderTraversal(Current.GetRightChild(), newPath)    def PreOrderTraversal(self):  self.DoPreOrderTraversal(self.\_Root, "") |
| --- |
| **Evidence 12** |

| **Question 4** |
| --- |
| **Evidence 13**  def declareGrid():  # grid[row][col]  grid = [[0, 0, 0, 0], \  [0, 0, 0, 0], \  [0, 0, 0, 0], \  [0, 0, 0, 0]]  return grid  def initialiseGrid(grid):  grid[0][0] = 4  grid[0][1] = 3  grid[0][2] = 2  grid[0][3] = 1    grid[1][0] = 1  grid[1][1] = 2  grid[1][2] = 4  grid[1][3] = 3    grid[2][0] = 3  grid[2][1] = 4  grid[2][2] = 1  grid[2][3] = 2    grid[3][0] = 2  grid[3][1] = 1  grid[3][2] = 3  grid[3][3] = 4  return grid  grid = declareGrid()  grid = initialiseGrid(grid) |
| **Evidence 14**  def displayGrid(grid):  for i in range(len(grid)):  for j in range(len(grid[i])):  if j < len(grid[i]) - 1:  print(str(grid[i][j]), end = " ")  else:  print(str(grid[i][j]))  displayGrid(grid) |
| **Evidence 15** |
| **Evidence 16**  def swapCol(c1, c2, grid):  # swaps 2 specific columns: c1 and c2  for i in range(len(grid)):  grid[i][c1], grid[i][c2] = grid[i][c2], grid[i][c1]  return grid  def swapRow(r1, r2, grid):  # swaps 2 specific rows: r1 and r2  grid[r1], grid[r2] = grid[r2], grid[r1]  return grid  def swapRandRow(grid):  import random  print("Transformation 1: Swaps two rows " + \  "in the same quadrants")  if random.randint(0, 1):  return swapRow(0, 1, grid)  else:  return swapRow(2, 3, grid)  def swapRandCol(grid):  import random  print("Transformation 2: Swaps two columns " + \  "in the same quadrants")  if random.randint(0, 1):  return swapCol(0, 1, grid)  else:  return SwapCol(2, 3, grid)  def swapQuadRow(grid):  print("Transformation 3: Swaps the top and bottom " + \  "quadrant rows entirely")  return swapRow(0, 2, swapRow(1, 3, grid))  def swapQuadCol(grid):  print("Transformation 4: Swaps the left and right " + \  "quadrant columns entirely")  return swapCol(0, 2, swapCol(1, 3, grid))  import random  choices = [1, 2, 3, 4]  choice1 = random.randint(1, len(choices) - 1)  choices = choices[:choice1] + choices[choice1 + 1:]  choice2 = choices[random.randint(1, len(choices) - 1)]  grid = declareGrid()  grid = initialiseGrid(grid)  displayGrid(grid)  if choice1 == 1 or choice2 == 1:  grid = swapRandRow(grid)  displayGrid(grid)  if choice1 == 2 or choice2 == 2:  grid = swapRandCol(grid)  displayGrid(grid)  if choice1 == 3 or choice2 == 3:  grid = swapQuadRow(grid)  displayGrid(grid)  if choice1 == 4 or choice2 == 4:  grid = swapQuadCol(grid)  displayGrid(grid) |
| **Evidence 17** |