Candidate name:

Centre Number:

Index number:

Programming language used:

|  |
| --- |
| **Question 1** |
| **Evidence 1**  #Read lines from file into DrinkList  f = open('DRINKS.TXT')  Drinklist = []  for line in f:  Drinklist += [line.strip()]  f.close()  #Menu  print('Menu\n')  print('1. Brewed coffee')  print('2. Brewed tea')  print('3. Other drinks\n')  #Function to return list  def output(lst, selection):  print(selection+'\n')  for i in lst:  print(i)  print('\nTotal items: ',len(lst))    #User input  user\_input = input('Option:')  return\_list = []  #Menu options  #Option 1  if user\_input == '1':  for i in Drinklist:  if 'Kopi' in i:  return\_list.append(i)  output(return\_list, "Brewed coffee")  #Option 2  elif user\_input == '2':  for i in Drinklist:  if 'Teh' in i:  return\_list.append(i)  output(return\_list, "Brewed tea")  #Option 3  elif user\_input == '3':  for i in Drinklist:  if ('Kopi' not in i) and ('Teh' not in i):  return\_list.append(i)  output(return\_list, "Other drinks")  #Handling invalid input  else:  print('No such option.') |
| **Question 2** |
| **Evidence 2**  #create hashtable  hashtable = []  for i in range(12):  hashtable += ['Empty']  #hash function  def hash(key):  location = int(key)%12  hashtable[location]=int(key)  #read contents from file  f = open('KEYS.TXT')  keys = []  for line in f:  keys.append(line.strip())  f.close()  #store contents into hash table  for i in keys:  hash(i)  #print contents of hash table  print('Hash table contents:')  for i in range(len(hashtable)):  print(hashtable[i]) |
| **Evidence 3**  #create hashtable  hashtable = []  for i in range(12):  hashtable += ['Empty']  #improved hash function  def improvedhash(key):  BaseAddress = int(key)%12  while hashtable[BaseAddress] is not 'Empty':  if BaseAddress + 1 == 12:  BaseAddress = 0  else:  BaseAddress += 1  hashtable[BaseAddress]=int(key)  #read contents from file KEYS2.TXT  f = open('KEYS2.TXT')  keys = []  for line in f:  keys.append(line.strip())  f.close()  #store contents into hash table  for i in keys:  improvedhash(i)  #print contents of hash table  print('Hash table contents:')  for i in range(len(hashtable)):  print(hashtable[i]) |
| **Evidence 4**  #Hash Search Function  def HashSearch(ID):  #ID not in table  if int(ID) not in hashtable:  return 'Student Number not found'  #ID in table  HashAddress = int(ID)%12  if hashtable[HashAddress] == int(ID):  return HashAddress  else:  while hashtable[HashAddress] != int(ID):  if HashAddress + 1 == 12:  HashAddress = 0  else:  HashAddress += 1  return HashAddress  #Search Student Number from User Input  userInput = input('Please enter a student number:')  print(HashSearch(userInput)) |
| **Question 3** |
| **Evidence 5**  class Node:  # constructor()  def \_\_init\_\_(self, data):  self.data = data  self.leftPtr = -1  self.rightPtr = -1  # modifier methods  def setData(self, s):  self.data = s  def setLeftPtr(self, x):  self.leftPtr = x  def setRightPtr(self, y):  self.rightPtr = y  # accessor methods  def getData(self):  return self.data  def getLeftPtr(self):  return self.leftPtr  def getRightPtr(self):  return self.rightPtr    # define class Tree  class Tree:  # define and initialise attributes of class Tree  def \_\_init\_\_(self):    # the tree data  self.tree = []  # index for the root position of the tree array  self.root = -1  # index for the next unused node  self.NextFreePosition = 0  # inserts a new item into the binary tree structure  def add(self, newItem):  self.tree.append(Node(newItem))  if self.root == -1:  self.root = self.NextFreePosition  self.NextFreePosition = 1  else:  # traverse the tree to find the position for the new value  CurrentPosition = self.root  LastMove = 'X'  PreviousPosition = -1  while CurrentPosition != -1:  PreviousPosition = CurrentPosition  if newItem < self.tree[CurrentPosition].data:  # move left  LastMove = 'L'  CurrentPosition = self.tree[CurrentPosition].leftPtr  else:  # move right  LastMove = 'R'  CurrentPosition = self.tree[CurrentPosition].rightPtr    if LastMove == 'R':  self.tree[PreviousPosition].rightPtr = self.NextFreePosition  else:  self.tree[PreviousPosition].leftPtr = self.NextFreePosition  self.NextFreePosition += 1  # output  #### format this however you want, as long as it's legible ####  def Print(self):  print('{0:12}{1:12}{2:12}{3}'.format('Node Index','Data','Left Ptr','Right Ptr'))  # for each node  for node in self.tree:  print('{0:^12}{1:<12}{2:^12}{3:^12}'.format(self.tree.index(node),node.data,node.leftPtr,node.rightPtr)) |
| **Evidence 6**  *NewTree = Tree()*  *NewTree.add("Tiger")*  *NewTree.add("Lemur")*  *NewTree.add("Bat")*  *NewTree.add("Yak")*  *NewTree.add("Ostrich")*  *NewTree.add("Raccoon")*  *NewTree.add("Macaw")*  *NewTree.add("Zebra")*  *NewTree.Print()* |
| **Evidence 7**  class Node:  # constructor()  def \_\_init\_\_(self, data):  self.data = data  self.leftPtr = -1  self.rightPtr = -1  # modifier methods  def setData(self, s):  self.data = s  def setLeftPtr(self, x):  self.leftPtr = x  def setRightPtr(self, y):  self.rightPtr = y  # accessor methods  def getData(self):  return self.data  def getLeftPtr(self):  return self.leftPtr  def getRightPtr(self):  return self.rightPtr    # define class Tree  class Tree:  # define and initialise attributes of class Tree  def \_\_init\_\_(self):    # the tree data  self.tree = []  # index for the root position of the tree array  self.root = -1  # index for the next unused node  self.NextFreePosition = 0  # inserts a new item into the binary tree structure  def add(self, newItem):  self.tree.append(Node(newItem))  if self.root == -1:  self.root = self.NextFreePosition  self.NextFreePosition = 1  else:  # traverse the tree to find the position for the new value  CurrentPosition = self.root  LastMove = 'X'  PreviousPosition = -1  while CurrentPosition != -1:  PreviousPosition = CurrentPosition  if newItem < self.tree[CurrentPosition].data:  # move left  LastMove = 'L'  CurrentPosition = self.tree[CurrentPosition].leftPtr  else:  # move right  LastMove = 'R'  CurrentPosition = self.tree[CurrentPosition].rightPtr    if LastMove == 'R':  self.tree[PreviousPosition].rightPtr = self.NextFreePosition  else:  self.tree[PreviousPosition].leftPtr = self.NextFreePosition  self.NextFreePosition += 1  # output  #### format this however you want, as long as it's legible ####  def Print(self):  print('{0:12}{1:12}{2:12}{3}'.format('Node Index','Data','Left Ptr','Right Ptr'))  # for each node  for node in self.tree:  print('{0:^12}{1:<12}{2:^12}{3:^12}'.format(self.tree.index(node),node.data,node.leftPtr,node.rightPtr))    # Task 3.3  # to output the data stored in the tree in reverse order  def postOrderTraversal(self, root = 0):  if self.tree[root].leftPtr is not -1:  self.postOrderTraversal(self.tree[root].leftPtr)  if self.tree[root].rightPtr is not -1:  self.postOrderTraversal(self.tree[root].rightPtr)  print(self.tree[root].data)  # main  NewTree = Tree()  NewTree.add("Tiger")  NewTree.add("Lemur")  NewTree.add("Bat")  NewTree.add("Yak")  NewTree.add("Ostrich")  NewTree.add("Raccoon")  NewTree.add("Macaw")  NewTree.add("Zebra")  #NewTree.Print()  NewTree.postOrderTraversal() |
| **Question 4** |
| **Evidence 8**  ###########  # Task 4.1#  ###########  #used interchangebly in the code for readability  #[1M - Interpreting '<--']  TEN = 'X'  STRIKE = TEN  # converting the 'X' or 'number' to int  #[1M - If Else accuracy]  def Pins(throw):  if throw == TEN:  return 10  else:  return int(throw)  # recursive solution  #[3M][1M each - interpreting SUM, List, LENGTH]  def Bowling\_Score(throws):  # need a helper function to keep track of the current frame number  def Bowling\_Score\_Helper(throws, frame\_num):  # account for frame 10 first  # last frame with no bonus  if frame\_num == 10 and len(throws) == 2:  return Pins(throws[0]) + Pins(throws[1])  # if the last frame contains 3 throws, then it must be a spare or strike  # In both cases, the score is computed in the same way.  if frame\_num == 10 and len(throws) == 3:  return Pins(throws[0]) + Pins(throws[1]) + Pins(throws[2])  # strike  if throws[0] == STRIKE:  frame\_score = 10 + Pins(throws[1]) + Pins(throws[2])  return frame\_score + Bowling\_Score\_Helper(throws[1:], frame\_num + 1)  frame\_score = Pins(throws[0]) + Pins(throws[1])  # spare  if frame\_score == 10:  return 10 + Pins(throws[2]) + Bowling\_Score\_Helper(throws[2:], frame\_num + 1)  # frame with no bonus  return frame\_score + Bowling\_Score\_Helper(throws[2:], frame\_num + 1)  return Bowling\_Score\_Helper(throws, 1) |
| **Evidence 9** |
| **Evidence 10**   |  |  |  | | --- | --- | --- | | Bowling Score | Purpose of the test | Expected Output | | 00000000000000000000 | Boundary test case (minimum) | 0 | | XXXXXXXXXXXX | Boundary test case (maximum) | 300 | | XXXXXXXXXXXY | Invalid characters | Invalid Characters. | |
| **Evidence 11**  ############  # Task 4.2 #  ############  '''  Test Cases  1. Boundary test case: '00000000000000000000'  2. Boundary test case: 'XXXXXXXXXXXX'  3. Invalid characters: 'XxXXXXXXXXXY'  '''  #used interchangebly in the code for readability  TEN = 'X'  STRIKE = TEN  # converting the 'X' or 'number' to int  def Pins(throw):  return 10 if throw == TEN else int(throw)  # recursive solution  def Bowling\_Score(throws):  ##Test for Invalid Characters  charlist = ['0','1','2','3','4','5','6','7','8','9','X']  for i in throws:  if i not in charlist:  return 'Invalid Characters.'  # need a helper function to keep track of the current frame number  def Bowling\_Score\_Helper(throws, frame\_num):  # account for frame 10 first  # last frame with no bonus  if frame\_num == 10 and len(throws) == 2:  return Pins(throws[0]) + Pins(throws[1])  # if the last frame contains 3 throws, then it must be a spare or strike  # In both cases, the score is computed in the same way.  if frame\_num == 10 and len(throws) == 3:  return Pins(throws[0]) + Pins(throws[1]) + Pins(throws[2])  # strike  if throws[0] == STRIKE:  frame\_score = 10 + Pins(throws[1]) + Pins(throws[2])  return frame\_score + Bowling\_Score\_Helper(throws[1:], frame\_num + 1)  frame\_score = Pins(throws[0]) + Pins(throws[1])  # spare  if frame\_score == 10:  return 10 + Pins(throws[2]) + Bowling\_Score\_Helper(throws[2:], frame\_num + 1)  # frame with no bonus  return frame\_score + Bowling\_Score\_Helper(throws[2:], frame\_num + 1)  return Bowling\_Score\_Helper(throws, 1)  ##Test Cases  print("'00000000000000000000':", Bowling\_Score('00000000000000000000'))  print("'XXXXXXXXXXXX':",Bowling\_Score('XXXXXXXXXXXX'))  print("'XxXXXXXXXXXY':",Bowling\_Score('XxXXXXXXXXXY')) |
| **Evidence 12** |
| **Evidence 13**  ##Bubble Sort (decending) Function Here##  def bubble\_sort(lst):  if len(lst) < 2:  return  for j in range(len(lst)-1):  for i in range(len(lst)-1-j):  if lst[i][2]<lst[i+1][2]:  lst[i], lst[i+1] = lst[i+1], lst[i] |
| **Evidence 14**  '''  Task 4.3  '''  def total\_score(lst):  total = 0  scores = lst[2:]  #print(scores)  for score in scores:  #print(score)  #print(Bowling\_Score(score))  total += Bowling\_Score(score)  return total  f = open('SCORES.TXT')  lst = []  for row in f:  lst += [row.split()]  #print(lst)  f.close()  scorelst = []  for entry in lst:  #print(entry)  scorelst += [[entry[0],entry[1],total\_score(entry)]]  ##Bubble Sort (decending) Function Here##  def bubble\_sort(lst):  if len(lst) < 2:  return  for j in range(len(lst)-1):  for i in range(len(lst)-1-j):  if lst[i][2]<lst[i+1][2]:  lst[i], lst[i+1] = lst[i+1], lst[i]  bubble\_sort(scorelst)  print(scorelst)  '''  Output of Results  '''  print('Official Results\n')  print('{0:8}{1:^20}{2:^10}{3:^11}'.format('Position','Register Number','Country','Total Score'))  rank = 0 #to print position  for i in scorelst:  print('{0:^8}{1:^20}{2:^10}{3:^11}'.format(rank+1,i[0],i[1],i[2]))  rank += 1 |