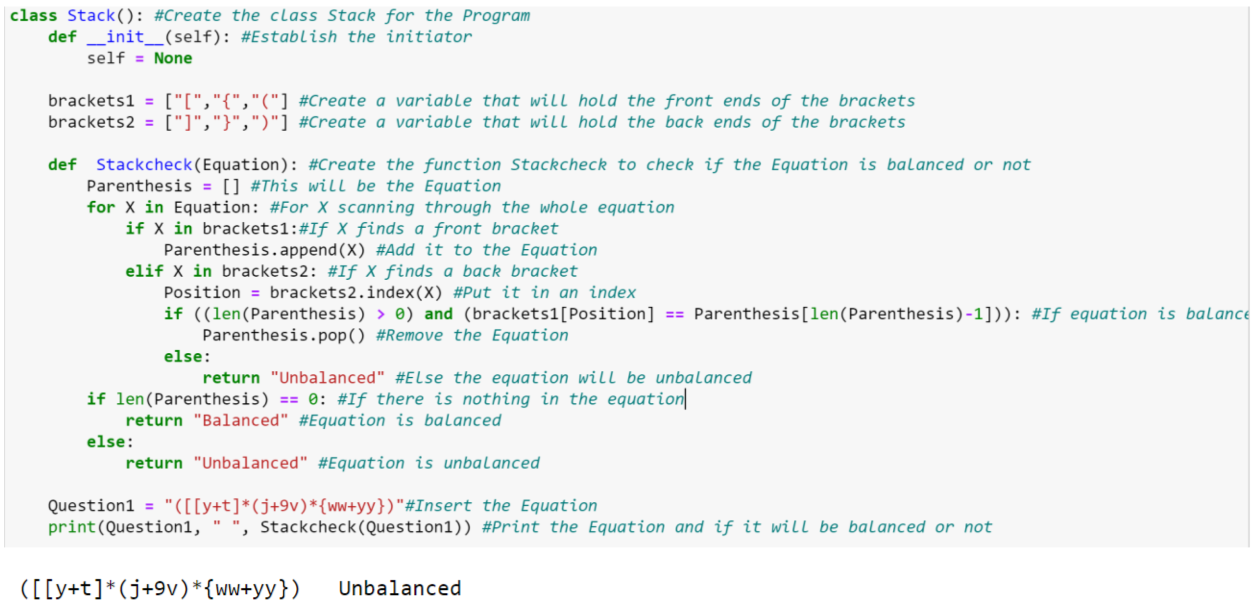
# ITDPA2-B44 Assessment

## Question 1:



```python
class Stack(): #Create the class Stack for the Program
    def __init__(self): #Establish the initiator
        self = None

    brackets1 = ["[","{","("] #Create a variable that will hold the front ends of the brackets
    brackets2 = ["]","}",")"] #Create a variable that will hold the back ends of the brackets

    def  Stackcheck(Equation): #Create the function Stackcheck to check if the Equation is balanced or not
        Parenthesis = [] #This will be the Equation
        for X in Equation: #For X scanning through the whole equation
            if X in brackets1:#If X finds a front bracket
                Parenthesis.append(X) #Add it to the Equation
            elif X in brackets2: #If X finds a back bracket
                Position = brackets2.index(X) #Put it in an index
                if ((len(Parenthesis) > 0) and (brackets1[Position] == Parenthesis[len(Parenthesis)-1])): #If equation is balance
                    Parenthesis.pop() #Remove the Equation
                else:
                    return "Unbalanced" #Else the equation will be unbalanced
        if len(Parenthesis) == 0: #If there is nothing in the equation
            return "Balanced" #Equation is balanced
        else:
            return "Unbalanced" #Equation is unbalanced

    Question1 = "([[y+t]*(j+9v)*{ww+yy})"#Insert the Equation
    print(Question1, " ", Stackcheck(Question1)) #Print the Equation and if it will be balanced or not
```

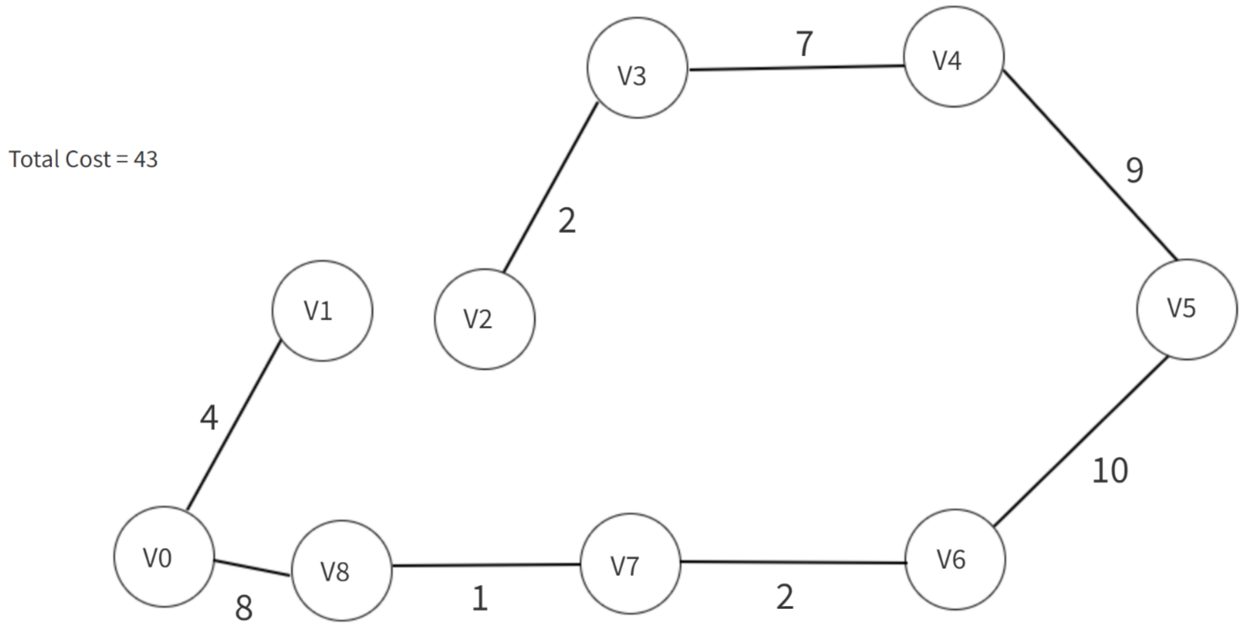
```
([[y+t]*(j+9v)*{ww+yy})   Unbalanced
```

Question 2:

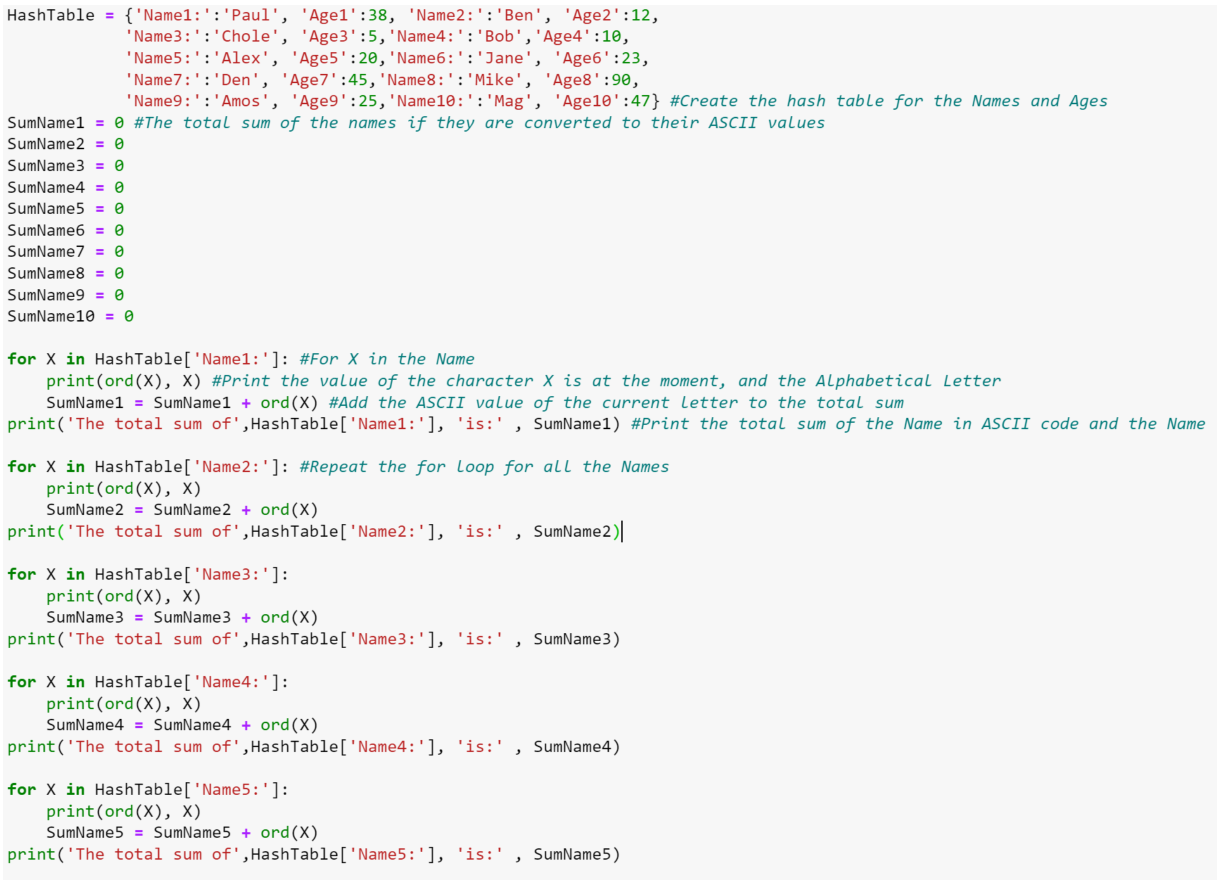
2.1 Quick sort is better than Merge sort, because:

- Quick sort is better for smaller data structures: Merge sort is mostly used on large data structures like linked lists or slow access media like Storage on a Network or HDD, where quick sort is better in small data structures that are easier to sort and faster.
- Quicksort occupies less space: Like mentioned before, Merge sort is better on large storage media and Quick sort requires less space, is better on the locality of cache and it does not require additional storage to sort.
- Better at Cache locality: Quicksort is better than Merge sort in cache locality and thus it is better and faster in more situations than Merge sort.
- Does not have many errors: Most of the errors in quick sort can be solved by using a randomized quicksort and by having the specific pivot, errors on high probability can be easily solved.

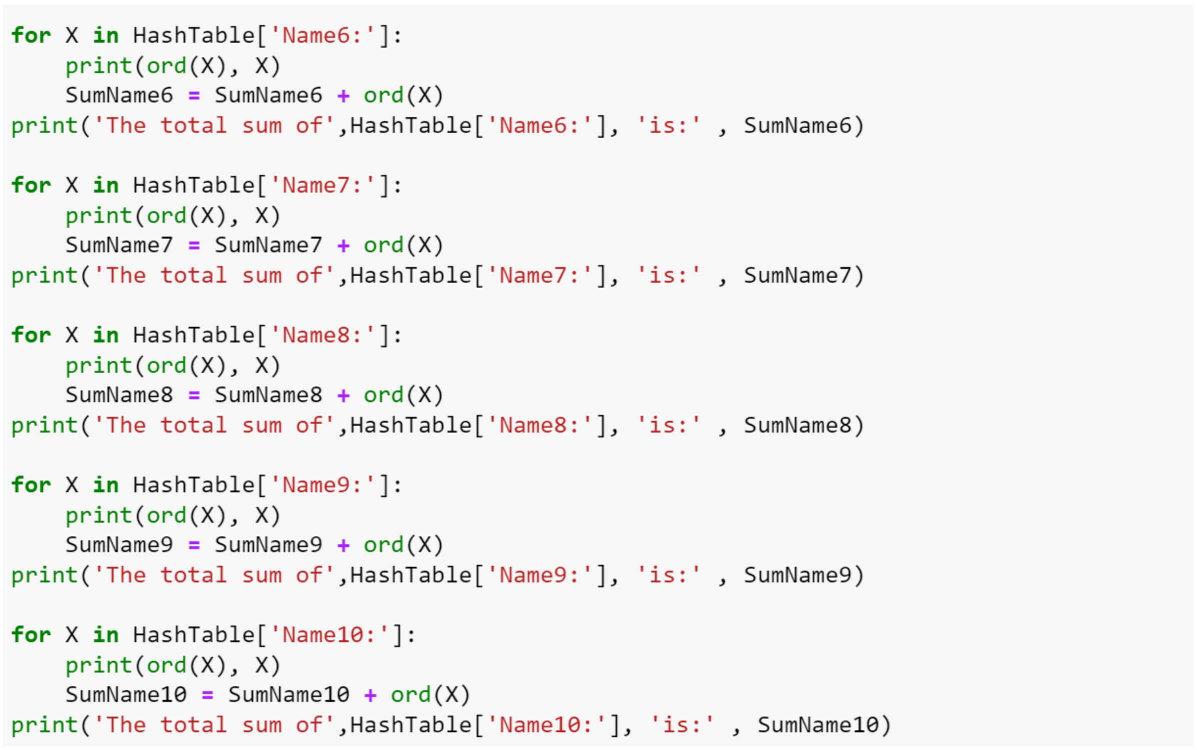
2.2



## Question 3:



```
HashTable = {'Name1:':'Paul', 'Age1':38, 'Name2:':'Ben', 'Age2':12,
            'Name3:':'Chole', 'Age3':5,'Name4:':'Bob','Age4':10,
            'Name5:':'Alex', 'Age5':20,'Name6:':'Jane', 'Age6':23,
            'Name7:':'Den', 'Age7':45,'Name8:':'Mike', 'Age8':90,
            'Name9:':'Amos', 'Age9':25,'Name10:':'Mag', 'Age10':47} #Create the hash table for the Names and Ages
SumName1 = 0 #The total sum of the names if they are converted to their ASCII values
SumName2 = 0
SumName3 = 0
SumName4 = 0
SumName5 = 0
SumName6 = 0
SumName7 = 0
SumName8 = 0
SumName9 = 0
SumName10 = 0

for X in HashTable['Name1:']: #For X in the Name
    print(ord(X), X) #Print the value of the character X is at the moment, and the Alphabetical Letter
    SumName1 = SumName1 + ord(X) #Add the ASCII value of the current letter to the total sum
print('The total sum of',HashTable['Name1:'], 'is:' , SumName1) #Print the total sum of the Name in ASCII code and the Name

for X in HashTable['Name2:']: #Repeat the for loop for all the Names
    print(ord(X), X)
    SumName2 = SumName2 + ord(X)
print('The total sum of',HashTable['Name2:'], 'is:' , SumName2)

for X in HashTable['Name3:']:
    print(ord(X), X)
    SumName3 = SumName3 + ord(X)
print('The total sum of',HashTable['Name3:'], 'is:' , SumName3)

for X in HashTable['Name4:']:
    print(ord(X), X)
    SumName4 = SumName4 + ord(X)
print('The total sum of',HashTable['Name4:'], 'is:' , SumName4)

for X in HashTable['Name5:']:
    print(ord(X), X)
    SumName5 = SumName5 + ord(X)
print('The total sum of',HashTable['Name5:'], 'is:' , SumName5)
```



```
for X in HashTable['Name6:']:
    print(ord(X), X)
    SumName6 = SumName6 + ord(X)
print('The total sum of',HashTable['Name6:'], 'is:' , SumName6)

for X in HashTable['Name7:']:
    print(ord(X), X)
    SumName7 = SumName7 + ord(X)
print('The total sum of',HashTable['Name7:'], 'is:' , SumName7)

for X in HashTable['Name8:']:
    print(ord(X), X)
    SumName8 = SumName8 + ord(X)
print('The total sum of',HashTable['Name8:'], 'is:' , SumName8)

for X in HashTable['Name9:']:
    print(ord(X), X)
    SumName9 = SumName9 + ord(X)
print('The total sum of',HashTable['Name9:'], 'is:' , SumName9)

for X in HashTable['Name10:']:
    print(ord(X), X)
    SumName10 = SumName10 + ord(X)
print('The total sum of',HashTable['Name10:'], 'is:' , SumName10)
```

```
80 P
97 a
117 u
108 l
The total sum of Paul is: 402
66 B
101 e
110 n
The total sum of Ben is: 277
67 C
104 h
111 o
108 l
101 e
The total sum of Chole is: 491
66 B
111 o
98 b
The total sum of Bob is: 275
65 A
108 l
101 e
120 x
The total sum of Alex is: 394
74 J
97 a
110 n
101 e
The total sum of Jane is: 382
68 D
101 e
110 n
The total sum of Den is: 279
77 M
105 i
107 k
101 e
The total sum of Mike is: 390
65 A
109 m
111 o
115 s
The total sum of Amos is: 400
77 M
97 a
103 g
The total sum of Mag is: 277
```

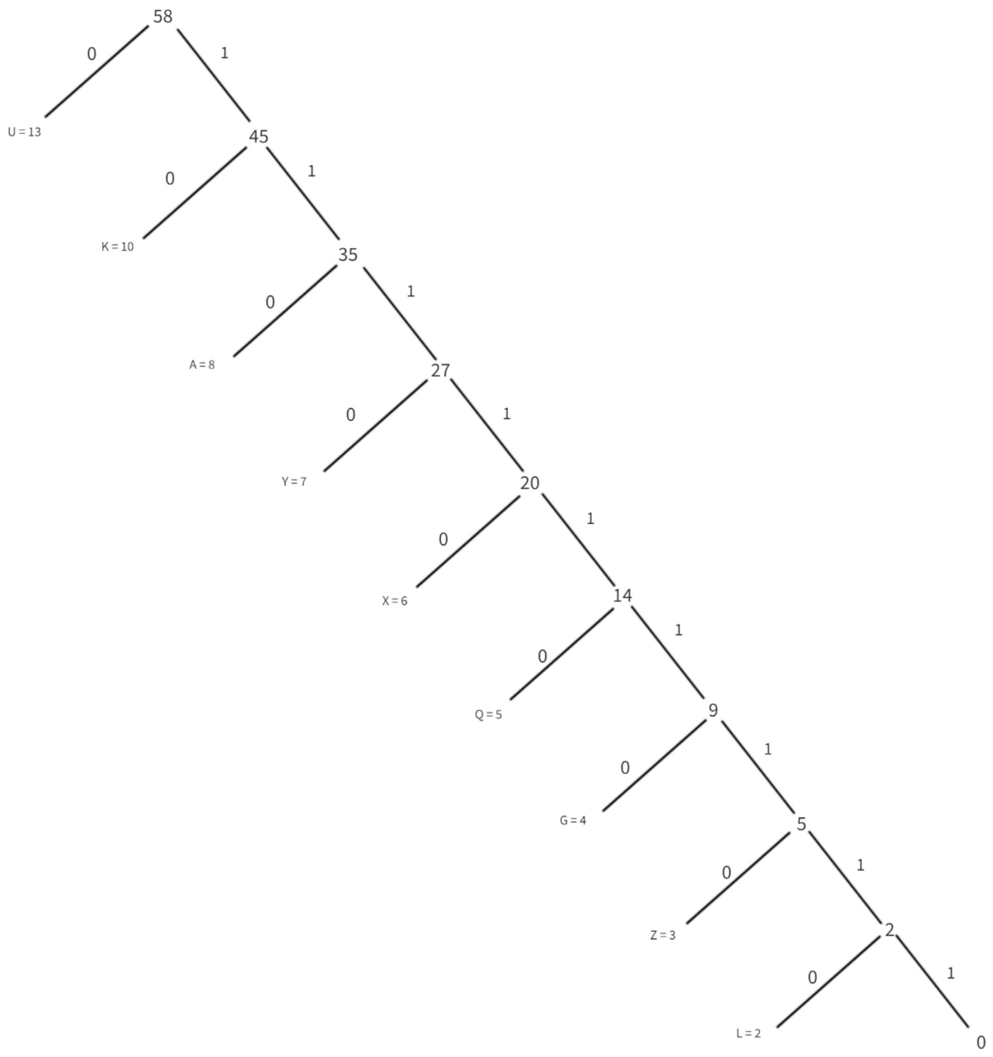
Question 4:

Question 5:

5.1

| Character | Frequency | Total | Hoffman Code: |
|---|---|---|---|
| a | 𝍸 \|\|\| | 8 | 110 |
| y | 𝍸 \|\| | 7 | 1110 |
| u | 𝍸𝍸 \|\|\| | 13 | 0 |
| k | 𝍸𝍸 | 10 | 10 |
| z | \|\|\| | 3 | 11111110 |
| l | \|\| | 2 | 111111110 |
| g | \|\|\|\| | 4 | 1111110 |
| q | 𝍸 | 5 | 111110 |
| x | 𝍸 \| | 6 | 11110 |

Hoffman Tree:



5.2

Average Length:

A + Y + U + K + Z + L + G + Q + X

(3 + 4 + 1 +2 + 8 + 9 + 7 + 6 +5) = 45/9 = 5

5.3

Length in Bits:

A + Y + U + K + Z + L + G + Q + X

3 + 4 + 1 +2 + 8 + 9 + 7 + 6 +5 = 45 bits