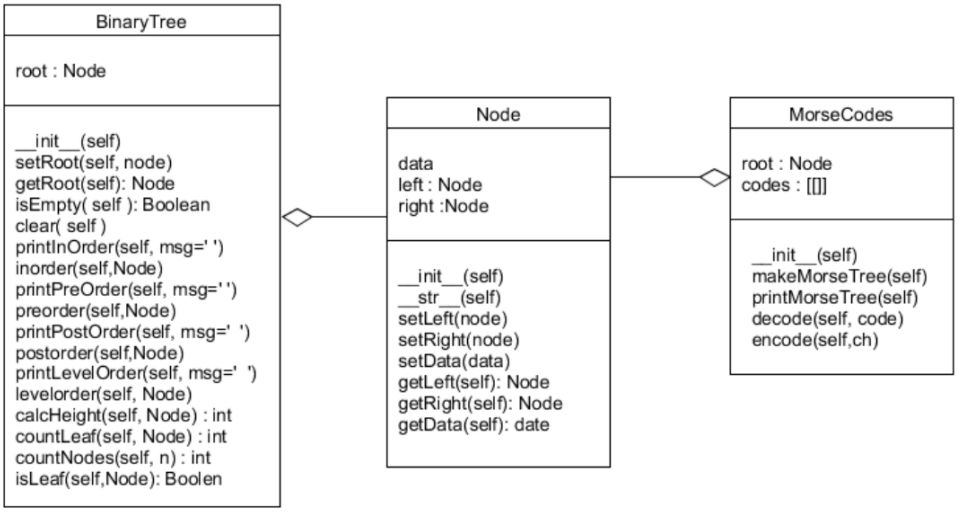
Data Structures 2023-2

Lab 06: Tree Data Structures

Task-1: Implement Morse Codes and Binary Tree

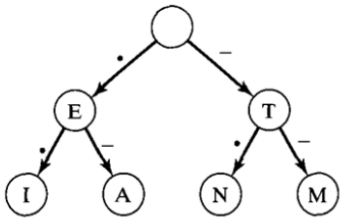


A variable-length binary code represents different characters using didifferent number of bits. One such example is the Morse code. In Morse code, characters are represented as sequences of dots(0) and dashes(1).

Variable length codes

A ·−	M −−	Y −·−−
B −···	N −·	Z −−··
C −·−·	O −−−	1 ·−−−−
D −··	P ·−−·	2 ··−−−
E ·	Q −−·−	3 ···−−
F ··−·	R ·−·	4 ····−
G −−·	S ···	5 ·····
H ····	T −	6 −····
I ··	U ··−	7 −−···
J ·−−−	V ···−	8 −−−··
K −·−	W ·−−	9 −−−·
L ·−··	X −··−	0 −−−−−

Tree representation



Code

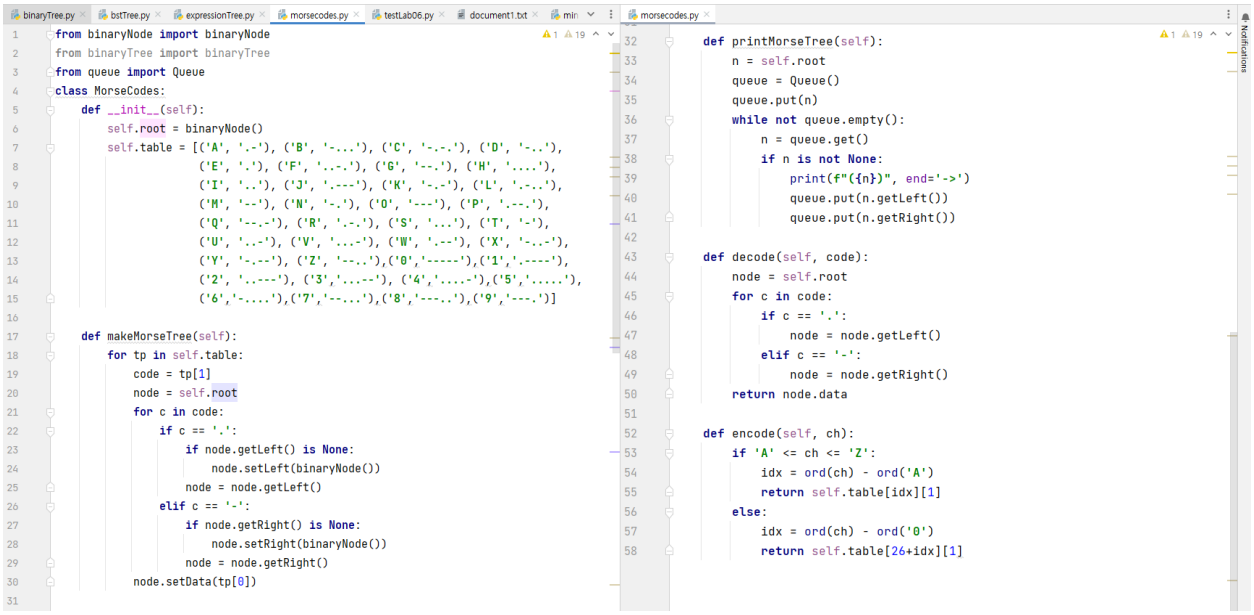
binaryNode.py # It is Node class in task01

```python
from collections import deque
from queue import Queue, LifoQueue

class binaryNode:
    def __init__(self, data=None, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right

    def __str__(self): return str(self.data)

    def getData(self): return self.data
    def getLeft(self): return self.left
    def getRight(self): return self.right

    def setData(self, data):self.data = data
    def setLeft(self, node): self.left = node
    def setRight(self, node): self.right = node

    def __eq__(self, other): return self.data == other.data
    def __ne__(self, other): return self.data != other.data
    def __lt__(self, other): return self.data < other.data
    def __gt__(self, other): return self.data > other.data
```

binaryTree.py

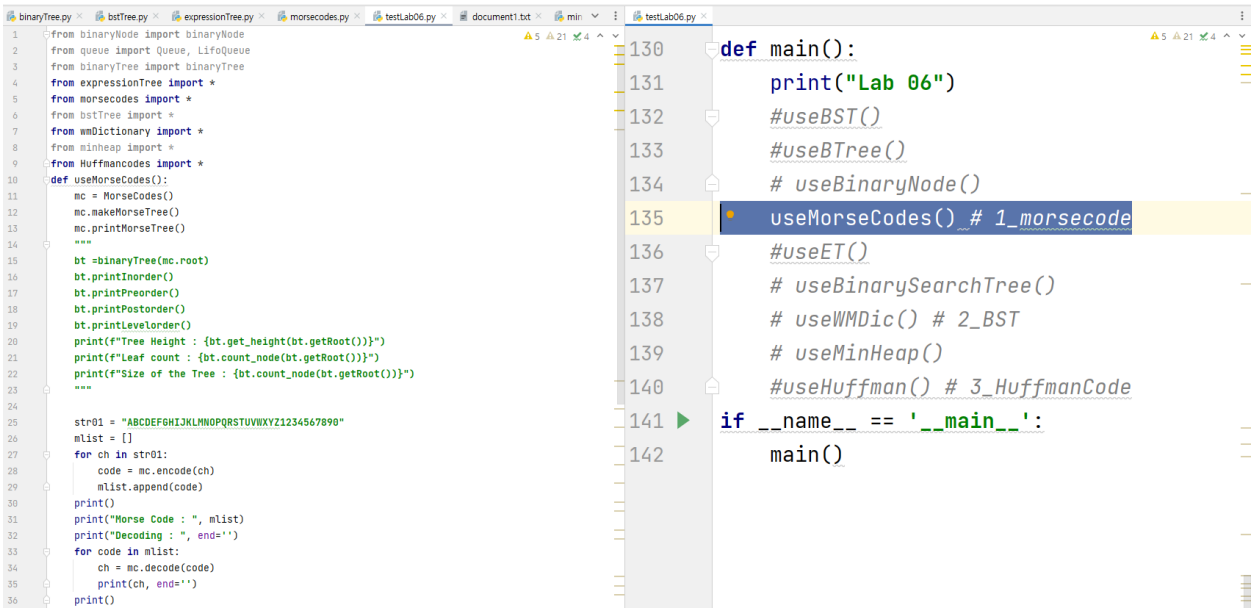
```python
from binaryNode import *

class binaryTree:
    def __init__(self, root=None):
        self.root = root

    def getRoot(self):
        return self.root
    def setRoot(self, node):
        self.root = node

    def isEmpty(self):
        return self.root is None

    def printInorder(self, msg="In-order : "):
        print(msg, end = " ")
        self.inorder(self.getRoot())
        print()
    def inorder(self, n):
        if n is not None:
            self.inorder(n.getLeft())
            print(f"({n})", end="->")
            self.inorder(n.getRight())

    def printPreorder(self, msg="Pre-order : "):
        print(msg, end=' ')
        self.preorder(self.getRoot())
        print()

    def preorder(self, n):
        if n is not None:
            print(f"({n})", end="->")
            self.preorder(n.getLeft())
            self.preorder(n.getRight())

    def preorder2(self, n):
        s = LifoQueue()
        s.put(n)
        while not s.empty():
            n1 = s.get()
            if n1 is not None:
                print(n1, end=" ")
                s.put(n1.getLeft())
                s.put(n1.getRight())
        print()


    def printPostorder(self, msg="Post-order : "):
        print(msg, end=' ')
        self.postorder(self.getRoot())
        print()

    def postorder(self, n):
        if n is not None:
            self.postorder(n.getLeft())
            self.postorder(n.getRight())
            print(f"({n})", end="->")

    def printLevelorder(self, msg="Level-order: "):
        print(msg, end=" ")
        self.levelorder(self.getRoot())

    def levelorder(self, n): # Breadth First Search
        q = Queue()
        q.put(n)
        print("level")
        while not q.empty():
            n = q.get()
            if n is not None:
                print(f"({n})", end= "->")
                q.put(n.getRight())
                q.put(n.getLeft())
        print("(END)", end="\n")
    def count_node(self, n):
        if n is None:
            return 0
        else:
            return 1+self.count_node(n.getLeft()) + self.count_node(n.getRight())
    def count_leaf(self, n):
        if n is None:
            return 0
        elif self.isLeaf(n):
            return 1
        else:
            return self.count_leaf(n.getLeft()) + self.count_leaf(n.getRight())

    def isLeaf(self, n):
        return n.getLeft() is None and n.getRight() is None

    def get_height(self, n):
        if n is None:
            return -1
        hleft = self.get_height(n.getLeft())
        hright = self.get_height(n.getRight())
        if hleft == None or hright == None:
            pass
        elif hleft > hright:
            return hleft + 1
        else:
            hright + 1
```

morsecodes.py



```python
from binaryNode import binaryNode
from binaryTree import binaryTree
from queue import Queue
class MorseCodes:
    def __init__(self):
        self.root = binaryNode()
        self.table = [('A', '.-'), ('B', '-...'), ('C', '-.-.'), ('D', '-..'),
                      ('E', '.'), ('F', '..-.'), ('G', '--.'), ('H', '....'),
                      ('I', '..'), ('J', '.---'), ('K', '-.-'), ('L', '.-..'),
                      ('M', '--'), ('N', '-.'), ('O', '---'), ('P', '.--.'),
                      ('Q', '--.-'), ('R', '.-.'), ('S', '...'), ('T', '-'),
                      ('U', '..-'), ('V', '...-'), ('W', '.--'), ('X', '-..-'),
                      ('Y', '-.--'), ('Z', '--..'),('0','-----'),('1','.----'),
                      ('2', '..---'), ('3','...--'), ('4','....-'),('5','.....'),
                      ('6','-....'),('7','--...'),('8','---..'),('9','----.')]

    def makeMorseTree(self):
        for tp in self.table:
            code = tp[1]
            node = self.root
            for c in code:
                if c == '.':
                    if node.getLeft() is None:
                        node.setLeft(binaryNode())
                    node = node.getLeft()
                elif c == '-':
                    if node.getRight() is None:
                        node.setRight(binaryNode())
                    node = node.getRight()
            node.setData(tp[0])

    def printMorseTree(self):
        n = self.root
        queue = Queue()
        queue.put(n)
        while not queue.empty():
            n = queue.get()
            if n is not None:
                print(f"({n})", end='->')
                queue.put(n.getLeft())
                queue.put(n.getRight())

    def decode(self, code):
        node = self.root
        for c in code:
            if c == '.':
                node = node.getLeft()
            elif c == '-':
                node = node.getRight()
        return node.data

    def encode(self, ch):
        if 'A' <= ch <= 'Z':
            idx = ord(ch) - ord('A')
            return self.table[idx][1]
        else:
            idx = ord(ch) - ord('0')
            return self.table[26+idx][1]
```

testLab06.py

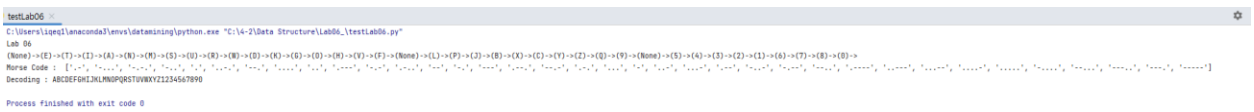


```python
from binaryNode import binaryNode
from queue import Queue, LifoQueue
from binaryTree import binaryTree
from expressionTree import *
from morsecodes import *
from bstTree import *
from wmDictionary import *
from minheap import *
from Huffmancodes import *
def useMorseCodes():
    mc = MorseCodes()
    mc.makeMorseTree()
    mc.printMorseTree()
    """
    bt =binaryTree(mc.root)
    bt.printInorder()
    bt.printPreorder()
    bt.printPostorder()
    bt.printLevelorder()
    print(f"Tree Height : {bt.get_height(bt.getRoot())}")
    print(f"Leaf count : {bt.count_node(bt.getRoot())}")
    print(f"Size of the Tree : {bt.count_node(bt.getRoot())}")
    """

    str01 = "ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890"
    mlist = []
    for ch in str01:
        code = mc.encode(ch)
        mlist.append(code)
    print()
    print("Morse Code : ", mlist)
    print("Decoding : ", end='')
    for code in mlist:
        ch = mc.decode(code)
        print(ch, end='')
    print()
```

```python
def main():
    print("Lab 06")
    #useBST()
    #useBTree()
    # useBinaryNode()
    useMorseCodes() # 1_morsecode
    #useET()
    # useBinarySearchTree()
    # useWMDic() # 2_BST
    # useMinHeap()
    #useHuffman() # 3_HuffmanCode
if __name__ == '__main__':
    main()
```

Results/Output

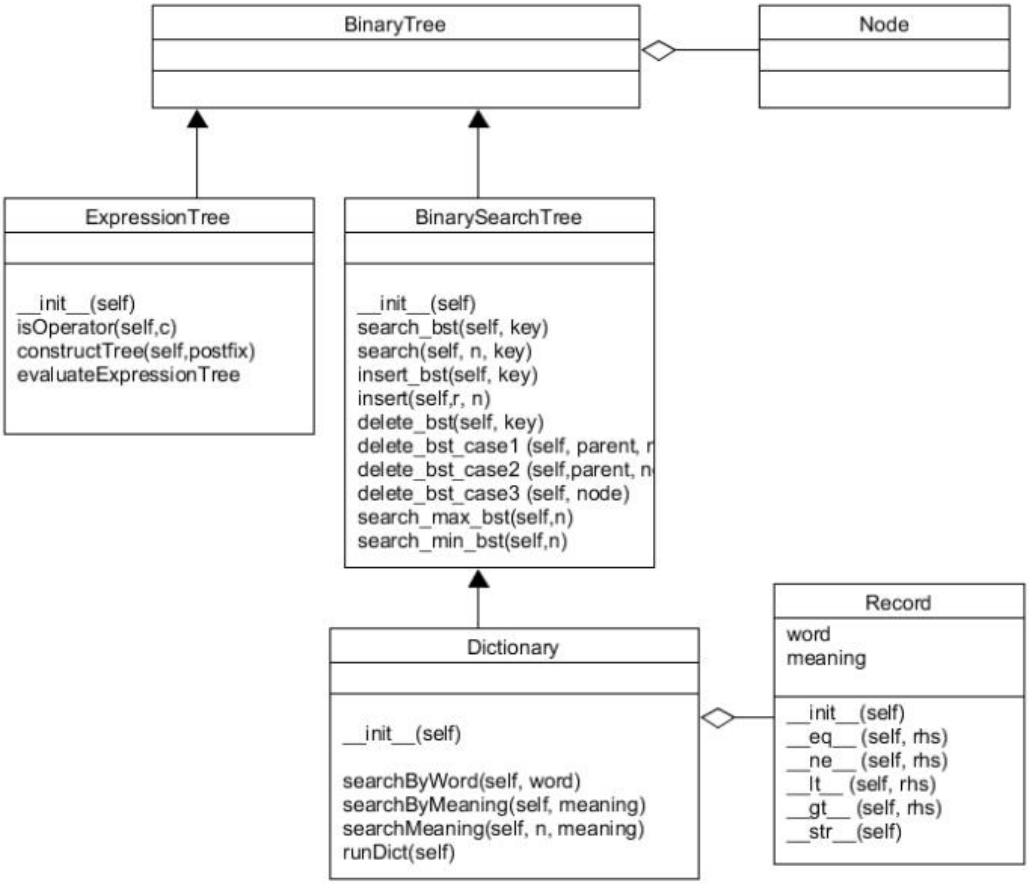
Insert pictures for the output of the programs written for this task



```
testLab06
C:\Users\iqeq1\anaconda3\envs\datamining\python.exe "C:\4-2\Data Structure\Lab06_\testLab06.py"
Lab 06
(None)->(E)->(T)->(I)->(A)->(N)->(M)->(S)->(U)->(R)->(W)->(D)->(K)->(G)->(O)->(H)->(V)->(F)->(None)->(L)->(P)->(J)->(B)->(X)->(C)->(Y)->(Z)->(Q)->(9)->(None)->(5)->(4)->(3)->(2)->(1)->(6)->(7)->(8)->(0)->
Morse Code :  ['.-', '-...', '-.-.', '-..', '.', '..-.', '--.', '....', '..', '.---', '-.-', '.-..', '--', '-.', '---', '.--.', '--.-', '.-.', '...', '-', '..-', '...-', '.--', '-..-', '-.--', '--..', '-----', '.----', '..---', '...--', '....-', '.....', '-....', '--...', '---..', '----.']
Decoding : ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890

Process finished with exit code 0
```

Task2: Binary Search Tree: Implement the BST(binary search tree) and use it to construct a Dictionary that contains words

of English and their meanings in Hangul**.**



Code

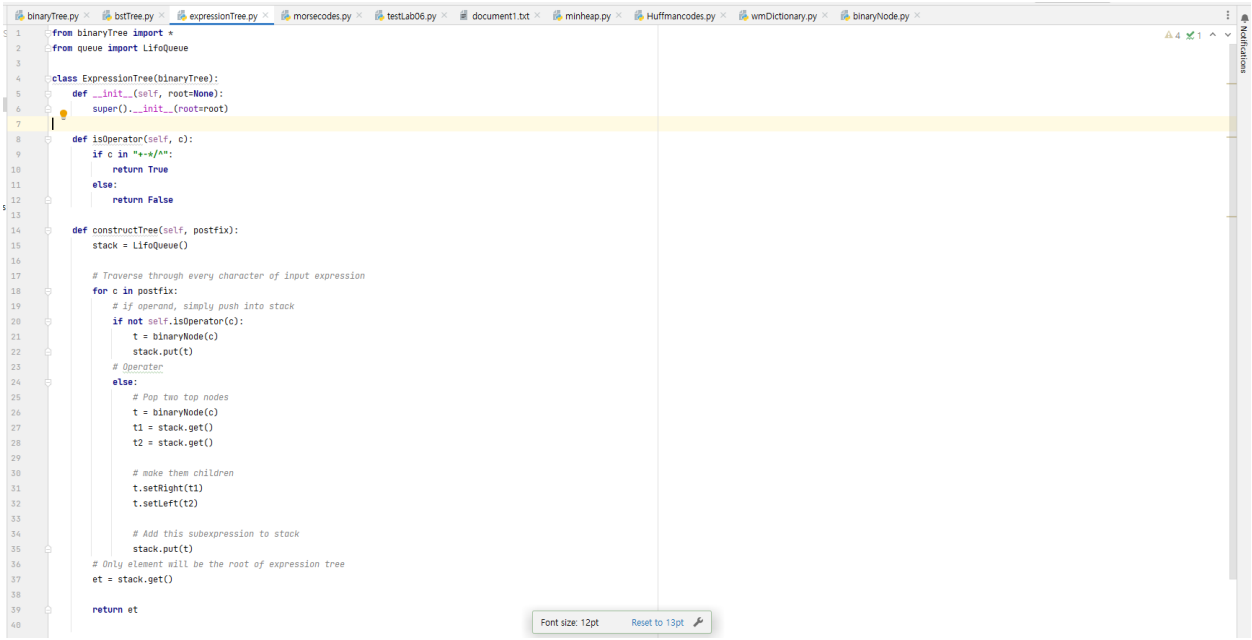
binaryNode.py # It is Node class in task02

```python
from collections import deque
from queue import Queue, LifoQueue

class binaryNode:
    def __init__(self, data=None, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right

    def __str__(self): return str(self.data)

    def getData(self): return self.data
    def getLeft(self): return self.left
    def getRight(self): return self.right

    def setData(self, data):self.data = data
    def setLeft(self, node): self.left = node
    def setRight(self, node): self.right = node

    def __eq__(self, other): return self.data == other.data
    def __ne__(self, other): return self.data != other.data
    def __lt__(self, other): return self.data < other.data
    def __gt__(self, other): return self.data > other.data
```

binaryTree.py

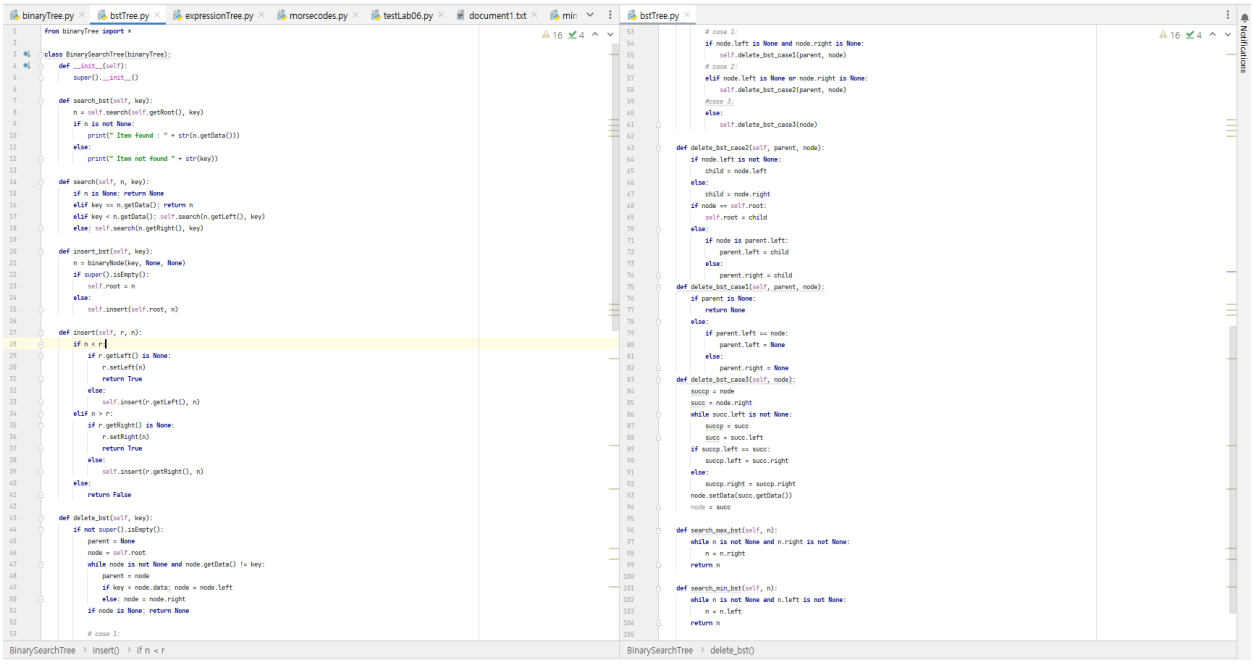
```python
from binaryNode import *

class binaryTree:
    def __init__(self, root=None):
        self.root = root

    def getRoot(self):
        return self.root
    def setRoot(self, node):
        self.root = node

    def isEmpty(self):
        return self.root is None

    def printInorder(self, msg="In-order : "):
        print(msg, end = " ")
        self.inorder(self.getRoot())
        print()
    def inorder(self, n):
        if n is not None:
            self.inorder(n.getLeft())
            print(f"({n})", end="->")
            self.inorder(n.getRight())

    def printPreorder(self, msg="Pre-order : "):
        print(msg, end=' ')
        self.preorder(self.getRoot())
        print()

    def preorder(self, n):
        if n is not None:
            print(f"({n})", end="->")
            self.preorder(n.getLeft())
            self.preorder(n.getRight())

    def preorder2(self, n):
        s = LifoQueue()
        s.put(n)
        while not s.empty():
            n1 = s.get()
            if n1 is not None:
                print(n1, end=" ")
                s.put(n1.getLeft())
                s.put(n1.getRight())
        print()


    def printPostorder(self, msg="Post-order : "):
        print(msg, end=' ')
        self.postorder(self.getRoot())
        print()

    def postorder(self, n):
        if n is not None:
            self.postorder(n.getLeft())
            self.postorder(n.getRight())
            print(f"({n})", end="->")

    def printLevelorder(self, msg="Level-order: "):
        print(msg, end=" ")
        self.levelorder(self.getRoot())

    def levelorder(self, n): # Breadth First Search
        q = Queue()
        q.put(n)
        print("level")
        while not q.empty():
            n = q.get()
            if n is not None:
                print(f"({n})", end= "->")
                q.put(n.getRight())
                q.put(n.getLeft())
        print("(END)", end="\n")
    def count_node(self, n):
        if n is None:
            return 0
        else:
            return 1+self.count_node(n.getLeft()) + self.count_node(n.getRight())
    def count_leaf(self, n):
        if n is None:
            return 0
        elif self.isLeaf(n):
            return 1
        else:
            return self.count_leaf(n.getLeft()) + self.count_leaf(n.getRight())

    def isLeaf(self, n):
        return n.getLeft() is None and n.getRight() is None

    def get_height(self, n):
        if n is None:
            return -1
        hleft = self.get_height(n.getLeft())
        hright = self.get_height(n.getRight())
        if hleft == None or hright == None:
            pass
        elif hleft > hright:
            return hleft + 1
        else:
            hright + 1
```

expressionTree.py



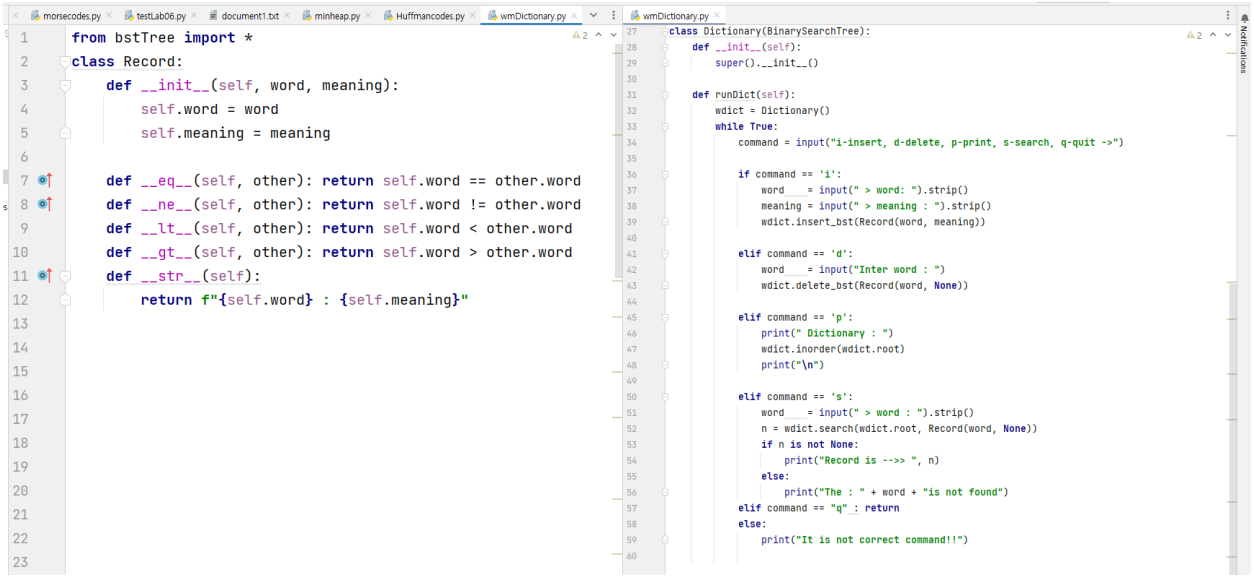
```
from binaryTree import *
from queue import LifoQueue

class ExpressionTree(binaryTree):
    def __init__(self, root=None):
        super().__init__(root=root)

    def isOperator(self, c):
        if c in "+-*/^":
            return True
        else:
            return False

    def constructTree(self, postfix):
        stack = LifoQueue()

        # Traverse through every character of input expression
        for c in postfix:
            # if operand, simply push into stack
            if not self.isOperator(c):
                t = binaryNode(c)
                stack.put(t)
            # Operater
            else:
                # Pop two top nodes
                t = binaryNode(c)
                t1 = stack.get()
                t2 = stack.get()

                # make them children
                t.setRight(t1)
                t.setLeft(t2)

                # Add this subexpression to stack
                stack.put(t)
        # Only element will be the root of expression tree
        et = stack.get()

        return et
```

bstTree.py # It is BinarySearchTree class in task02



```
from binaryTree import *

class BinarySearchTree(binaryTree):
    def __init__(self):
        super().__init__()

    def search_bst(self, key):
        n = self.search(self.getRoot(), key)
        if n is not None:
            print(" Item found : " + str(n.getData()))
        else:
            print(" Item not found " + str(key))

    def search(self, n, key):
        if n is None: return None
        elif key == n.getData(): return n
        elif key < n.getData(): self.search(n.getLeft(), key)
        else: self.search(n.getRight(), key)

    def insert_bst(self, key):
        n = binaryNode(key, None, None)
        if super().isEmpty():
            self.root = n
        else:
            self.insert(self.root, n)

    def insert(self, r, n):
        if n < r:
            if r.getLeft() is None:
                r.setLeft(n)
                return True
            else:
                self.insert(r.getLeft(), n)
        elif n > r:
            if r.getRight() is None:
                r.setRight(n)
                return True
            else:
                self.insert(r.getRight(), n)
        else:
            return False

    def delete_bst(self, key):
        if not super().isEmpty():
            parent = None
            node = self.root
            while node is not None and node.getData() != key:
                parent = node
                if key < node.data: node = node.left
                else: node = node.right
            if node is None: return None

            # case 1:
            if node.left is None and node.right is None:
                self.delete_bst_case1(parent, node)
            # case 2:
            elif node.left is None or node.right is None:
                self.delete_bst_case2(parent, node)
            #case 3:
            else:
                self.delete_bst_case3(node)

    def delete_bst_case2(self, parent, node):
        if node.left is not None:
            child = node.left
        else:
            child = node.right
        if node == self.root:
            self.root = child
        else:
            if node is parent.left:
                parent.left = child
            else:
                parent.right = child
    def delete_bst_case1(self, parent, node):
        if parent is None:
            return None
        else:
            if parent.left == node:
                parent.left = None
            else:
                parent.right = None
    def delete_bst_case3(self, node):
        succp = node
        succ = node.right
        while succ.left is not None:
            succp = succ
            succ = succ.left
        if succp.left == succ:
            succp.left = succ.right
        else:
            succp.right = succp.right
        node.setData(succ.getData())
        node = succ

    def search_max_bst(self, n):
        while n is not None and n.right is not None:
            n = n.right
        return n

    def search_min_bst(self, n):
        while n is not None and n.left is not None:
            n = n.left
        return n
```

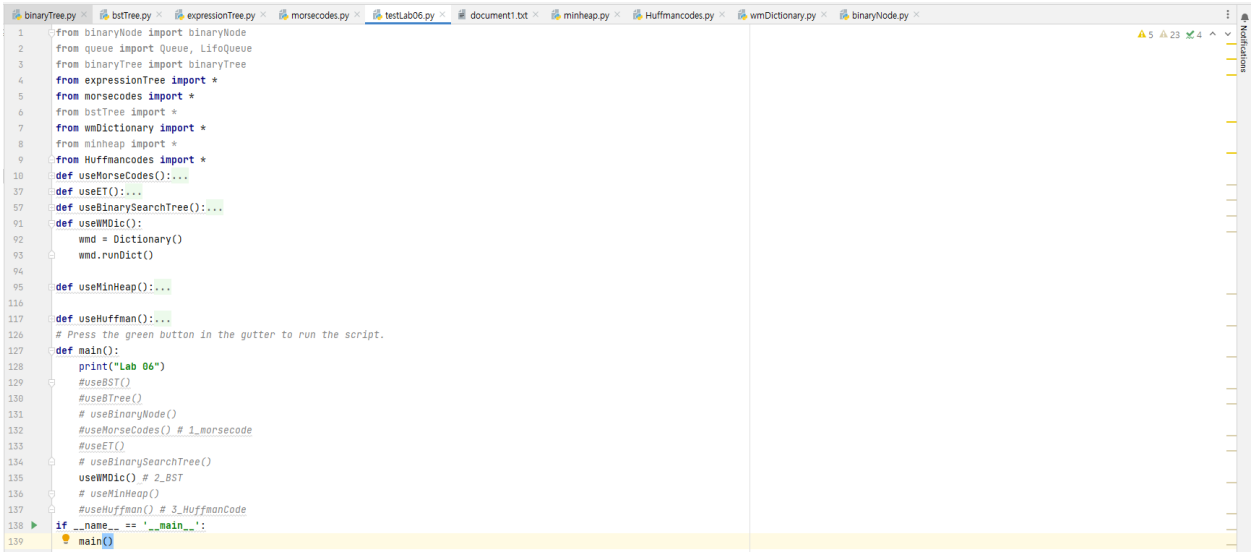
wmDictionary.py # Record and Dictionary class in task02



```python
from bstTree import *
class Record:
    def __init__(self, word, meaning):
        self.word = word
        self.meaning = meaning

    def __eq__(self, other): return self.word == other.word
    def __ne__(self, other): return self.word != other.word
    def __lt__(self, other): return self.word < other.word
    def __gt__(self, other): return self.word > other.word
    def __str__(self):
        return f"{self.word} : {self.meaning}"
```

```python
class Dictionary(BinarySearchTree):
    def __init__(self):
        super().__init__()

def runDict(self):
    wdict = Dictionary()
    while True:
        command = input("i-insert, d-delete, p-print, s-search, q-quit ->")

        if command == 'i':
            word    = input(" > word: ").strip()
            meaning = input(" > meaning : ").strip()
            wdict.insert_bst(Record(word, meaning))

        elif command == 'd':
            word    = input("Inter word : ")
            wdict.delete_bst(Record(word, None))

        elif command == 'p':
            print(" Dictionary : ")
            wdict.inorder(wdict.root)
            print("\n")

        elif command == 's':
            word    = input(" > word : ").strip()
            n = wdict.search(wdict.root, Record(word, None))
            if n is not None:
                print("Record is --> ", n)
            else:
                print("The : " + word + "is not found")
        elif command == "q" : return
        else:
            print("It is not correct command!!")
```

testLab06.py



```python
from binaryNode import binaryNode
from queue import Queue, LifoQueue
from binaryTree import binaryTree
from expressionTree import *
from morsecodes import *
from bstTree import *
from wmDictionary import *
from minheap import *
from Huffmancodes import *
def useMorseCodes():...
def useET():...
def useBinarySearchTree():...
def useWMDic():
    wmd = Dictionary()
    wmd.runDict()

def useMinHeap():...

def useHuffman():...
# Press the green button in the gutter to run the script.
def main():
    print("Lab 06")
    #useBST()
    #useBTree()
    # useBinaryNode()
    #useMorseCodes() # 1_morsecode
    #useET()
    # useBinarySearchTree()
    useWMDic() # 2_BST
    # useMinHeap()
    #useHuffman() # 3_HuffmanCode
if __name__ == '__main__':
    main()
```

Results/Output

Insert pictures for the output of the programs written for this task

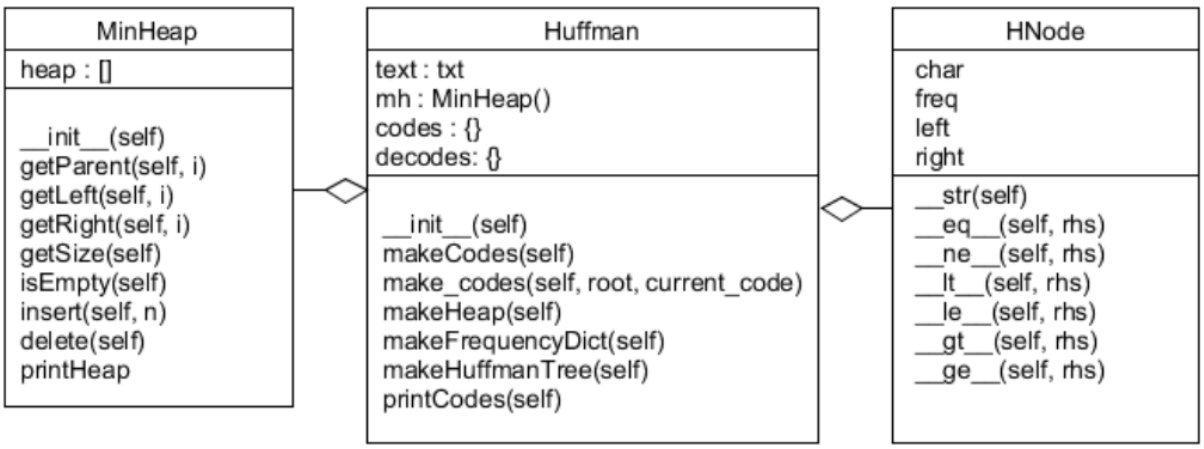
```
133     useET()
134     #useBinaryTreeSearchTree()
main()
```

testLab06

```
C:\Users\iqeq1\anaconda3\envs\datamining\python.exe "C:\4-2\Data Structure\Lab06_\testLab06.py"
Lab 06

Inorder( infix )
(3)->(+)->(5)->(+)->(9)->(*)->(2)->
preorder( prefix )
(*)->(+)->(3)->(+)->(5)->(9)->(2)->
postorder( postfix )
(3)->(5)->(9)->(+)->(+)->(2)->(*)->
Inorder( infix )
(a)->(+)->(b)->(-)->(e)->(*)->(f)->(*)->(g)->
preorder( prefix )
(-)->(+)->(a)->(b)->(*)->(*)->(e)->(f)->(g)->
postorder( postfix )
(a)->(b)->(+)->(e)->(f)->(*)->(g)->(*)->(-)->
Process finished with exit code 0
```

```
c:\4-2\Data Structure\Lab06_>python testLab06.py
Lab 06
i-insert, d-delete, p-print, s-search, q-quit ->i
 > word: aaa
 > meaning : q
i-insert, d-delete, p-print, s-search, q-quit ->q

c:\4-2\Data Structure\Lab06_>python testLab06.py
Lab 06
i-insert, d-delete, p-print, s-search, q-quit ->i
 > word: apple
 > meaning : 사과
i-insert, d-delete, p-print, s-search, q-quit ->i
 > word: banana
 > meaning : 바나나
i-insert, d-delete, p-print, s-search, q-quit ->p
 Dictionary :
(apple:사과)->(banana:바나나)->

i-insert, d-delete, p-print, s-search, q-quit ->d
Inter word : apple
i-insert, d-delete, p-print, s-search, q-quit ->p
 Dictionary :
(banana:바나나)->

i-insert, d-delete, p-print, s-search, q-quit ->s
 > word : banana
Record is -->>  banana:바나나
i-insert, d-delete, p-print, s-search, q-quit ->q

c:\4-2\Data Structure\Lab06_>
```

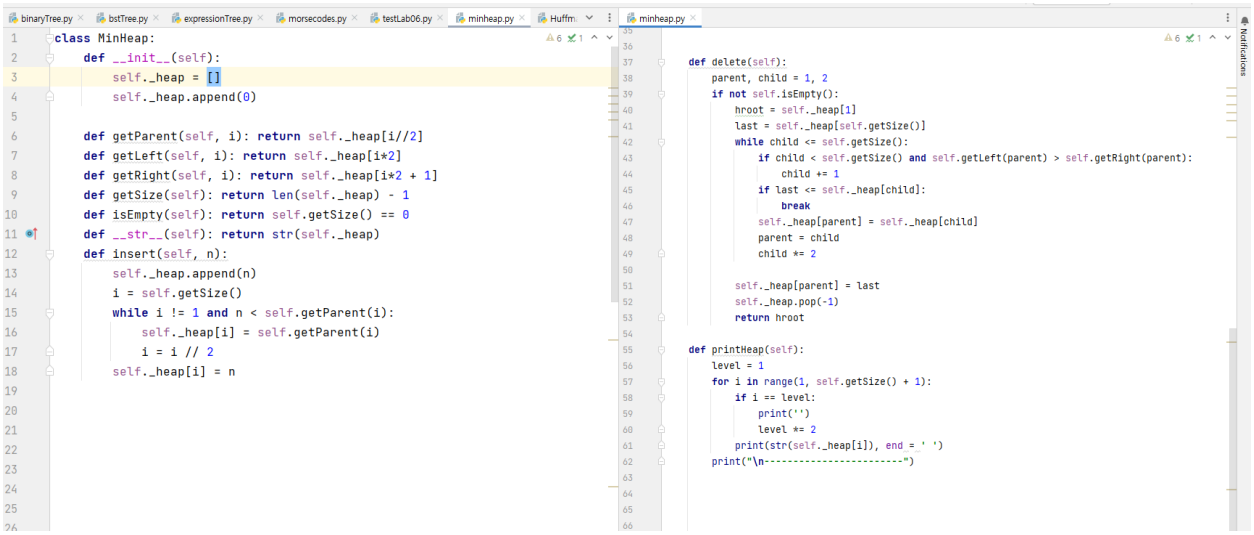
Task-3: Binary Heap: Implement the heap data structure (Priority Queue) and use it for Huffman coding.



Code

code for the solution.

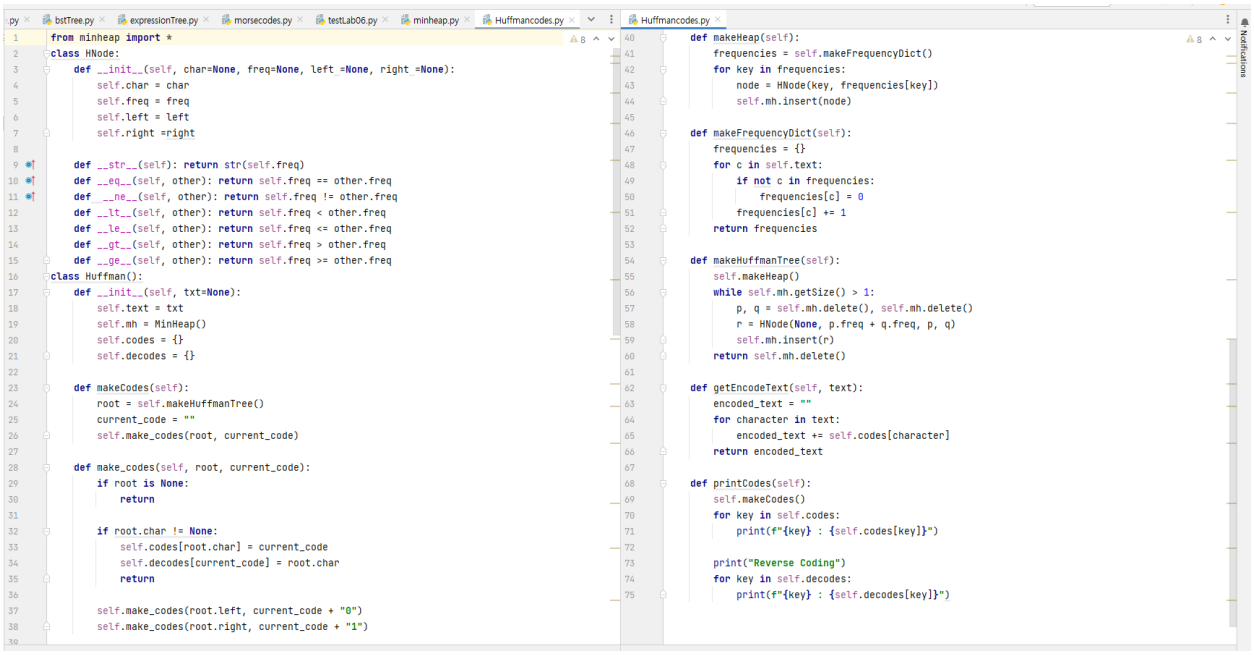
minheap.py



```python
class MinHeap:
    def __init__(self):
        self._heap = []
        self._heap.append(0)

    def getParent(self, i): return self._heap[i//2]
    def getLeft(self, i): return self._heap[i*2]
    def getRight(self, i): return self._heap[i*2 + 1]
    def getSize(self): return len(self._heap) - 1
    def isEmpty(self): return self.getSize() == 0
    def __str__(self): return str(self._heap)
    def insert(self, n):
        self._heap.append(n)
        i = self.getSize()
        while i != 1 and n < self.getParent(i):
            self._heap[i] = self.getParent(i)
            i = i // 2
        self._heap[i] = n
```

```python
    def delete(self):
        parent, child = 1, 2
        if not self.isEmpty():
            hroot = self._heap[1]
            last = self._heap[self.getSize()]
            while child <= self.getSize():
                if child < self.getSize() and self.getLeft(parent) > self.getRight(parent):
                    child += 1
                if last <= self._heap[child]:
                    break
                self._heap[parent] = self._heap[child]
                parent = child
                child *= 2

            self._heap[parent] = last
            self._heap.pop(-1)
            return hroot

    def printHeap(self):
        level = 1
        for i in range(1, self.getSize() + 1):
            if i == level:
                print('')
                level *= 2
            print(str(self._heap[i]), end = ' ')
        print("\n----------------------")
```

Huffmancode.py # Huffman and HNode class



```python
from minheap import *
class HNode:
    def __init__(self, char=None, freq=None, left =None, right =None):
        self.char = char
        self.freq = freq
        self.left = left
        self.right =right

    def __str__(self): return str(self.freq)
    def __eq__(self, other): return self.freq == other.freq
    def __ne__(self, other): return self.freq != other.freq
    def __lt__(self, other): return self.freq < other.freq
    def __le__(self, other): return self.freq <= other.freq
    def __gt__(self, other): return self.freq > other.freq
    def __ge__(self, other): return self.freq >= other.freq
class Huffman():
    def __init__(self, txt=None):
        self.text = txt
        self.mh = MinHeap()
        self.codes = {}
        self.decodes = {}

    def makeCodes(self):
        root = self.makeHuffmanTree()
        current_code = ""
        self.make_codes(root, current_code)

    def make_codes(self, root, current_code):
        if root is None:
            return

        if root.char != None:
            self.codes[root.char] = current_code
            self.decodes[current_code] = root.char
            return

        self.make_codes(root.left, current_code + "0")
        self.make_codes(root.right, current_code + "1")
```

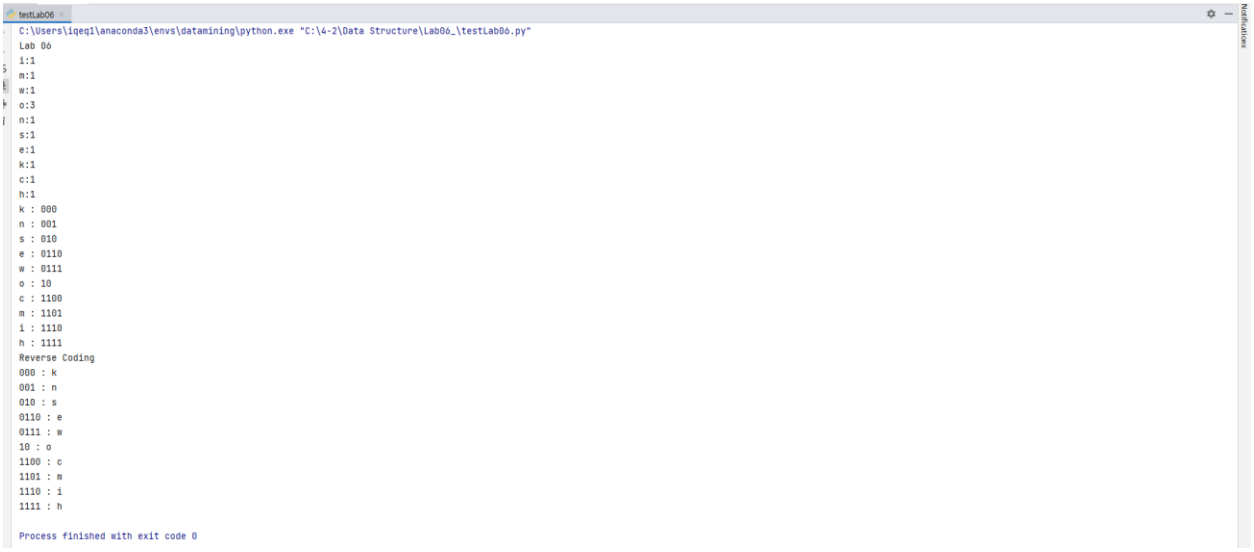
```python
    def makeHeap(self):
        frequencies = self.makeFrequencyDict()
        for key in frequencies:
            node = HNode(key, frequencies[key])
            self.mh.insert(node)

    def makeFrequencyDict(self):
        frequencies = {}
        for c in self.text:
            if not c in frequencies:
                frequencies[c] = 0
            frequencies[c] += 1
        return frequencies

    def makeHuffmanTree(self):
        self.makeHeap()
        while self.mh.getSize() > 1:
            p, q = self.mh.delete(), self.mh.delete()
            r = HNode(None, p.freq + q.freq, p, q)
            self.mh.insert(r)
        return self.mh.delete()

    def getEncodeText(self, text):
        encoded_text = ""
        for character in text:
            encoded_text += self.codes[character]
        return encoded_text

    def printCodes(self):
        self.makeCodes()
        for key in self.codes:
            print(f"{key} : {self.codes[key]}")

        print("Reverse Coding")
        for key in self.decodes:
            print(f"{key} : {self.decodes[key]}")
```

TestLab06.py

```python
def useHuffman():
    with open('document1.txt') as txt_file:
        text = txt_file.read()
    # text = "abcdefghijklmnopqrstuvwxyz"
    hc = Huffman(text)
    freq = hc.makeFrequencyDict()
    for key in freq:
        print(f"{key}:{freq[key]}")
    hc.printCodes()
# Press the green button in the gutter to run the script.
def main():
    print("Lab 06")
    #useBST()
    #useBTree()
    # useBinaryNode()
    #useMorseCodes() # 1_morsecode
    # useET()
    #useBinarySearchTree()
    #useWMDic() # 2_BST
    # useMinHeap()
    useHuffman() # 3_HuffmanCode
if __name__ == '__main__':
    main()
```

Results/Output

Insert pictures for the output of the programs written for this task



Conclusion

Conclude the Lab. Write your views about it, i.e. what have you learned from this lab? It was helpful or difficult etc

I can learn binaryTree, binarySearchTree, and Heap data structure to implement the Morsecode, the Dictionary and the Huffman Code. It is very useful to me. Thank you.