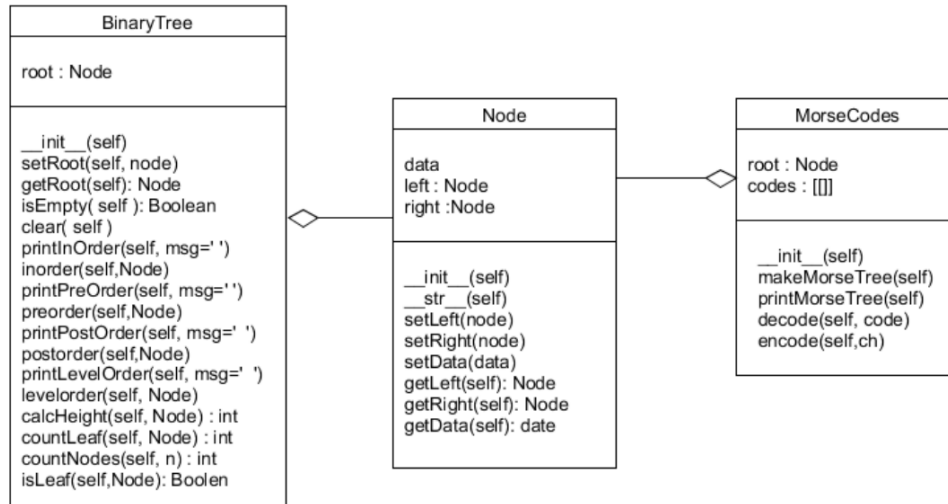


Data Structures 2023-2

Lab 06: Tree Data Structures

Task-1: Implement Morse Codes and Binary Tree

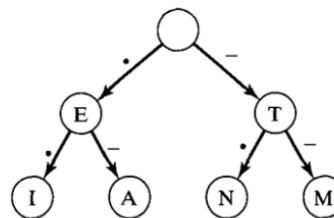


A variable-length binary code represents different characters using different number of bits. One such example is the Morse code. In Morse code, characters are represented as sequences of dots(0) and dashes(1).

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

```
1 from collections import deque
2 from queue import Queue, LifoQueue
3
4 class binaryNode:
5     def __init__(self, data=None, left=None, right=None):
6         self.data = data
7         self.left = left
8         self.right = right
9
10    def __str__(self): return str(self.data)
11
12    def getData(self): return self.data
13    def getLeft(self): return self.left
14    def getRight(self): return self.right
15
16    def setData(self, data): self.data = data
17    def setLeft(self, node): self.left = node
18    def setRight(self, node): self.right = node
19
20    def __eq__(self, other): return self.data == other.data
21    def __ne__(self, other): return self.data != other.data
22    def __lt__(self, other): return self.data < other.data
23    def __gt__(self, other): return self.data > other.data
```

binaryTree.py

```
1 from binaryNode import *
2
3 class binaryTree:
4     def __init__(self, root=None):
5         self.root = root
6
7     def getRoot(self):
8         return self.root
9
10    def setRoot(self, node):
11        self.root = node
12
13    def isEmpty(self):
14        return self.root is None
15
16    def printInorder(self, msg="In-order : "):
17        print(msg, end=" ")
18        self.inorder(self.getRoot())
19        print()
20    def inorder(self, n):
21        if n is not None:
22            self.inorder(n.getLeft())
23            print(f"({n})", end=">")
24            self.inorder(n.getRight())
25
26    def printPreorder(self, msg="Pre-order : "):
27        print(msg, end=" ")
28        self.preorder(self.getRoot())
29        print()
30    def preorder(self, n):
31        if n is not None:
32            print(f"({n})", end=">")
33            self.preorder(n.getLeft())
34            self.preorder(n.getRight())
35
36    def preorder2(self, n):
37        s = LifoQueue()
38        s.put(n)
39        while not s.empty():
40            n1 = s.get()
41            if n1 is not None:
42                print(n1, end=" ")
43                s.put(n1.getLeft())
44                s.put(n1.getRight())
45            print()
46
47    def printPostorder(self, msg="Post-order : "):
48        print(msg, end=" ")
49        self.postorder(self.getRoot())
50        print()
51    def postorder(self, n):
52
53    def postorder(self, n):
54        if n is not None:
55            self.postorder(n.getLeft())
56            self.postorder(n.getRight())
57            print(f"({n})", end=">")
58
59    def printLevelorder(self, msg="Level-order : "):
60        print(msg, end=" ")
61        self.levelorder(self.getRoot())
62
63    def levelorder(self, n): # Breadth First Search
64        q = Queue()
65        q.put(n)
66        print("Level")
67        while not q.empty():
68            n = q.get()
69            if n is not None:
70                print(f"({n})", end=">")
71                q.put(n.getRight())
72                q.put(n.getLeft())
73            print("(END)", end="\n")
74
75    def count_node(self, n):
76        if n is None:
77            return 0
78        else:
79            return 1+self.count_node(n.getLeft()) + self.count_node(n.getRight())
80
81    def count_leaf(self, n):
82        if n is None:
83            return 0
84        elif self.isLeaf(n):
85            return 1
86        else:
87            return self.count_leaf(n.getLeft()) + self.count_leaf(n.getRight())
88
89    def isLeaf(self, n):
90        return n.getLeft() is None and n.getRight() is None
91
92    def get_height(self, n):
93        if n is None:
94            return -1
95        hleft = self.get_height(n.getLeft())
96        hright = self.get_height(n.getRight())
97        if hleft == None or hright == None:
98            pass
99        elif hleft > hright:
100            return hleft + 1
101        else:
102            return hright + 1
```

morsecodes.py

binaryTree.py

bstree.py

expressionTree.py

monocodes.py

testLab06.py

document1.txt

min.py

monocodes.py

```
1 from binaryNode import binaryNode
2 from binaryTree import binaryTree
3 from queue import Queue
4 class MorseCodes:
5     def __init__(self):
6         self.root = binaryNode()
7         self.table = [
8             ('A', '.-'), ('B', '-...'), ('C', '-.-.'), ('D', '-..'),
9             ('E', '.'), ('F', '..-.'), ('G', '-. '), ('H', '....'),
10            ('I', '..'), ('J', '.---'), ('K', '-.-'), ('L', '-. .'),
11            ('M', '--'), ('N', '-. '), ('O', '---'), ('P', '.-..'),
12            ('Q', '-.-.-'), ('R', '.- .'), ('S', '...'), ('T', '- '),
13            ('U', '-..'), ('V', '..-'), ('W', '---'), ('X', '-.-.-'),
14            ('Y', '-.-'), ('Z', '--'), ('0', '-----'), ('1', '------'),
15            ('2', '- - - - -'), ('3', '- . - - -'), ('4', '. - - - -'), ('5', '- . . - -'),
16            ('6', '- . . . -'), ('7', '- . . . -'), ('8', '- . . . -'), ('9', '- . . . -')]
17
18     def makeMorseTree(self):
19         for tp in self.table:
20             code = tp[1]
21             node = self.root
22             for c in code:
23                 if c == '.':
24                     if node.getLeft() is None:
25                         node.setLeft(binaryNode())
26                     node = node.getLeft()
27                 elif c == '-':
28                     if node.getRight() is None:
29                         node.setRight(binaryNode())
30                     node = node.getRight()
31             node.setData(tp[0])
```

monocodes.py

```
32 def printMorseTree(self):
33     n = self.root
34     queue = Queue()
35     queue.put(n)
36     while not queue.empty():
37         n = queue.get()
38         if n is not None:
39             print(f"({n}) ", end=">")
40             queue.put(n.getLeft())
41             queue.put(n.getRight())
42
43     def decode(self, code):
44         node = self.root
45         for c in code:
46             if c == '.':
47                 node = node.getLeft()
48             elif c == '-':
49                 node = node.getRight()
50         return node.data
51
52     def encode(self, ch):
53         if 'A' <= ch <= 'Z':
54             idx = ord(ch) - ord('A')
55             return self.table[idx][1]
56         else:
57             idx = ord(ch) - ord('0')
58             return self.table[26+idx][1]
```

testLab06.py

```
binaryTree.py bstTree.py expressionTree.py morsecodes.py testLab06.py document1.txt min: testLab06.py
1 from binaryNode import binaryNode
2 from queue import Queue, LifoQueue
3 from binaryTree import binaryTree
4 from expressionTree import *
5 from morsecodes import *
6 from bstTree import *
7 from wmdictionary import *
8 from minheap import *
9 from HuffmanCodes import *
10 def useMorseCodes():
11     mc = MorseCodes()
12     mc.makeMorseTree()
13     mc.printMorseTree()
14     """
15     bt = binaryTree(mc.root)
16     bt.printInorder()
17     bt.printPreorder()
18     bt.printPostorder()
19     bt.printLevelorder()
20     print(f"Tree Height : {bt.get_height(bt.getRoot())}")
21     print(f"Leaf count : {bt.count_node(bt.getRoot())}")
22     print(f"Size of the Tree : {bt.count_node(bt.getRoot())}")
23     """
24
25     str01 = "ABCDEFGHJKLMNPQRSTUVWXYZ1234567890"
26     mlist = []
27     for ch in str01:
28         code = mc.encode(ch)
29         mlist.append(code)
30     print()
31     print("Morse Code : ", mlist)
32     print("Decoding : ", end='')
33     for code in mlist:
34         ch = mc.decode(code)
35         print(ch, end='')
36     print()

testLab06.py
130 def main():
131     print("Lab 06")
132     #useBST()
133     #useBTree()
134     # useBinaryNode()
135     * useMorseCodes() # 1_morsecode
136     #useET()
137     # useBinarySearchTree()
138     # useWMDic() # 2_BST
139     # useMinHeap()
140     #useHuffman() # 3_HuffmanCode
141     if __name__ == '__main__':
142         main()
```

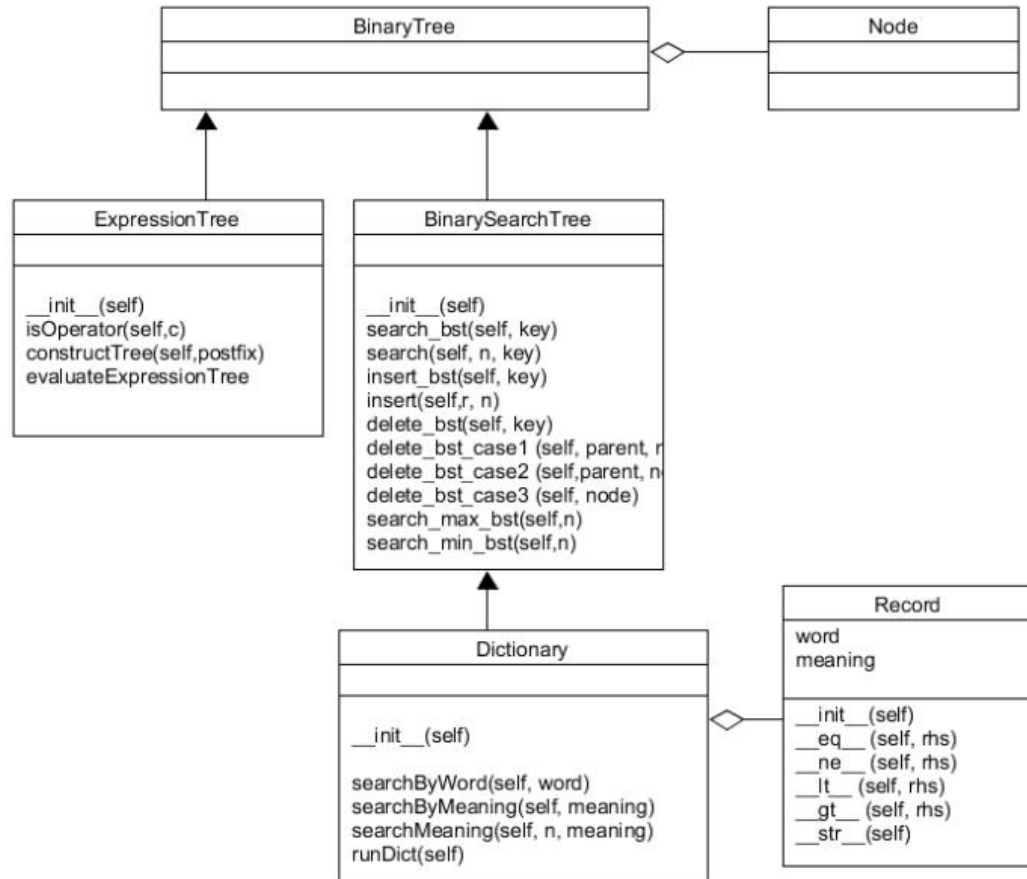
Results/Output

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

[illegible]

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

```
1 from collections import deque
2 from queue import Queue, LifoQueue
3
4 class binaryNode:
5     def __init__(self, data=None, left=None, right=None):
6         self.data = data
7         self.left = left
8         self.right = right
9
10    def __str__(self): return str(self.data)
11
12    def getData(self): return self.data
13    def getLeft(self): return self.left
14    def getRight(self): return self.right
15
16    def setData(self, data): self.data = data
17    def setLeft(self, node): self.left = node
18    def setRight(self, node): self.right = node
19
20    def __eq__(self, other): return self.data == other.data
21    def __ne__(self, other): return self.data != other.data
22    def __lt__(self, other): return self.data < other.data
23    def __gt__(self, other): return self.data > other.data
```

binaryTree.py

```
1 from binaryNode import *
2
3 class binaryTree:
4     def __init__(self, root=None):
5         self.root = root
6
7     def getRoot(self):
8         return self.root
9
10    def setRoot(self, node):
11        self.root = node
12
13    def isEmpty(self):
14        return self.root is None
15
16    def printInorder(self, msg="In-order : "):
17        print(msg, end=" ")
18        self.inorder(self.getRoot())
19        print()
20    def inorder(self, n):
21        if n is not None:
22            self.inorder(n.getLeft())
23            print(f"({n})", end=">")
24            self.inorder(n.getRight())
25
26    def printPreorder(self, msg="Pre-order : "):
27        print(msg, end=" ")
28        self.preorder(self.getRoot())
29        print()
30    def preorder(self, n):
31        if n is not None:
32            print(f"({n})", end=">")
33            self.preorder(n.getLeft())
34            self.preorder(n.getRight())
35
36    def preorder2(self, n):
37        s = LifoQueue()
38        s.put(n)
39        while not s.empty():
40            n1 = s.get()
41            if n1 is not None:
42                print(n1, end=" ")
43                s.put(n1.getLeft())
44                s.put(n1.getRight())
45            print()
46
47    def printPostorder(self, msg="Post-order : "):
48        print(msg, end=" ")
49        self.postorder(self.getRoot())
50        print()
51    def postorder(self, n):
52
53    def postorder(self, n):
54        if n is not None:
55            self.postorder(n.getLeft())
56            self.postorder(n.getRight())
57            print(f"({n})", end=">")
58
59    def printLevelorder(self, msg="Level-order : "):
60        print(msg, end=" ")
61        self.levelorder(self.getRoot())
62
63    def levelorder(self, n): # Breadth First Search
64        q = Queue()
65        q.put(n)
66        print("Level")
67        while not q.empty():
68            n = q.get()
69            if n is not None:
70                print(f"({n})", end=">")
71                q.put(n.getRight())
72                q.put(n.getLeft())
73            print("(END)", end="\n")
74
75    def count_node(self, n):
76        if n is None:
77            return 0
78        else:
79            return 1+self.count_node(n.getLeft()) + self.count_node(n.getRight())
80
81    def count_leaf(self, n):
82        if n is None:
83            return 0
84        elif self.isLeaf(n):
85            return 1
86        else:
87            return self.count_leaf(n.getLeft()) + self.count_leaf(n.getRight())
88
89    def isLeaf(self, n):
90        return n.getLeft() is None and n.getRight() is None
91
92    def get_height(self, n):
93        if n is None:
94            return -1
95        hleft = self.get_height(n.getLeft())
96        hright = self.get_height(n.getRight())
97        if hleft == None or hright == None:
98            pass
99        elif hleft > hright:
100            return hleft + 1
101        else:
102            return hright + 1
```

expressionTree.py

```
1 from binaryTree import *
2 from queue import LifoQueue
3
4 class ExpressionTree(binaryTree):
5     def __init__(self, root=None):
6         super().__init__(root=root)
7
8     def isOperator(self, c):
9         if c in "+-*/^":
10             return True
11         else:
12             return False
13
14     def constructTree(self, postfix):
15         stack = LifoQueue()
16
17         # Traverse through every character of input expression
18         for c in postfix:
19             # If operand, simply push into stack
20             if not self.isOperator(c):
21                 t = binaryNode(c)
22                 stack.put(t)
23             # Operator
24             else:
25                 # Pop two top nodes
26                 t = binaryNode(c)
27                 t1 = stack.get()
28                 t2 = stack.get()
29
30                 # make them children
31                 t.setRight(t1)
32                 t.setLeft(t2)
33
34                 # Add this subexpression to stack
35                 stack.put(t)
36
37         # Only element will be the root of expression tree
38         et = stack.get()
39
40         return et
```

bstTree.py # It is BinarySearchTree class in task02

```
1 from binaryTree import *
2
3 class BinarySearchTree(binaryTree):
4     def __init__(self):
5         super().__init__()
6
7     def search_bst(self, key):
8         n = self.search(self.getRoot(), key)
9         if n is not None:
10             print("Item Found : " + str(n.getData()))
11         else:
12             print("Item not found : " + str(key))
13
14     def search(self, n, key):
15         if n is None: return None
16         elif key == n.getData(): return n
17         elif key < n.getData(): self.search(n.getLeft(), key)
18         else: self.search(n.getRight(), key)
19
20     def insert_bst(self, key):
21         n = binaryNode(key, None, None)
22         if super().isEmpty():
23             self.root = n
24         else:
25             self.insert(self.root, n)
26
27     def insert(self, r, n):
28         if r.getLeft() is None:
29             r.setLeft(n)
30             return True
31         else:
32             self.insert(r.getLeft(), n)
33
34         elif n > r:
35             if r.getRight() is None:
36                 r.setRight(n)
37                 return True
38             else:
39                 self.insert(r.getRight(), n)
40
41         return False
42
43     def delete_bst(self, key):
44         if not super().isEmpty():
45             parent = None
46             node = self.root
47             while node is not None and node.getData() != key:
48                 parent = node
49                 if key < node.data: node = node.left
50                 else: node = node.right
51             if node is None: return None
52
53             # case 1:
54             if n < r:
55                 # case 1:
56                 if node.left is None and node.right is None:
57                     self.delete_bst_case1(parent, node)
58                 # case 2:
59                 elif node.left is None or node.right is None:
60                     self.delete_bst_case2(parent, node)
61                 # case 3:
62                 else:
63                     self.delete_bst_case3(node)
64
65             def delete_bst_case1(self, parent, node):
66                 if node.left is not None:
67                     child = node.left
68                     node = child
69                     if node == self.root:
69                         self.root = child
70                     if node is parent.left:
69                         parent.left = child
71                     else:
69                         parent.right = child
72
73             def delete_bst_case2(self, parent, node):
74                 if parent is None:
75                     return None
76                 else:
77                     if parent.left == node:
78                         parent.left = None
79                     else:
80                         parent.right = None
81
82             def delete_bst_case3(self, node):
83                 succ = node.right
84                 while succ.left is not None:
85                     succ = succ.left
86                 if succ.left == succ:
87                     succ.left = succ.right
88                 else:
89                     succ.right = succ.right
90                     node.setData(succ.getData())
91                     node = succ
92
93             def search_max_bst(self, n):
94                 while n is not None and n.right is not None:
95                     n = n.right
96                 return n
97
98             def search_min_bst(self, n):
99                 while n is not None and n.left is not None:
100                     n = n.left
101                 return n
102
103             # case 1:
104             if n < r:
105                 # case 1:
106                 if node.left is None and node.right is None:
107                     self.delete_bst_case1(parent, node)
108                 # case 2:
109                 elif node.left is None or node.right is None:
110                     self.delete_bst_case2(parent, node)
111                 # case 3:
112                 else:
113                     self.delete_bst_case3(node)
```

wmDictionary.py # Record and Dictionary class in task02

```
1 from bstTree import *
2 class Record:
3     def __init__(self, word, meaning):
4         self.word = word
5         self.meaning = meaning
6
7     def __eq__(self, other): return self.word == other.word
8     def __ne__(self, other): return self.word != other.word
9     def __lt__(self, other): return self.word < other.word
10    def __gt__(self, other): return self.word > other.word
11    def __str__(self):
12        return f"{self.word} : {self.meaning}"
13
14
15
16
17
18
19
20
21
22
23
```

```
27 class Dictionary(BinarySearchTree):
28     def __init__(self):
29         super().__init__()
30
31     def runDict(self):
32         wdict = Dictionary()
33         while True:
34             command = input("i-insert, d-delete, p-print, s-search, q-quit ->")
35
36             if command == 'i':
37                 word = input(" > word: ").strip()
38                 meaning = input(" > meaning : ").strip()
39                 wdict.insert_bst(Record(word, meaning))
40
41             elif command == 'd':
42                 word = input("Inter word : ")
43                 wdict.delete_bst(Record(word, None))
44
45             elif command == 'p':
46                 print(" Dictionary : ")
47                 wdict.inorder(wdict.root)
48                 print("\n")
49
50             elif command == 's':
51                 word = input(" > word : ").strip()
52                 n = wdict.search(wdict.root, Record(word, None))
53                 if n is not None:
54                     print("Record is --> ", n)
55                 else:
56                     print("The : " + word + " is not found")
57             elif command == "q" : return
58             else:
59                 print("It is not correct command!!")
60
```

testLab06.py

```
1 from binaryNode import binaryNode
2 from queue import Queue, LifoQueue
3 from binaryTree import binaryTree
4 from expressionTree import *
5 from morsecodes import *
6 from bstTree import *
7 from wmDictionary import *
8 from minheap import *
9 from HuffmanCodes import *
10 def useMorseCodes():...
11 def useET():...
12 def useBinarySearchTree():...
13 def useWMDic():
14     wmd = Dictionary()
15     wmd.runDict()
16
17 def useMinHeap():...
18
19 def useHuffman():...
20 # Press the green button in the gutter to run the script.
21 def main():
22     print("Lab 06")
23     #useBST()
24     #useBTree()
25     #useBinaryNode()
26     #useMorseCodes() # 1_morsecode
27     #useET()
28     #useBinarySearchTree()
29     useWMDic() # 2_BST
30     #useMinHeap()
31     #useHuffman() # 3_HuffmanCode
32 if __name__ == '__main__':
33     main()
```

Results/Output

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

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

testLab06 x
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

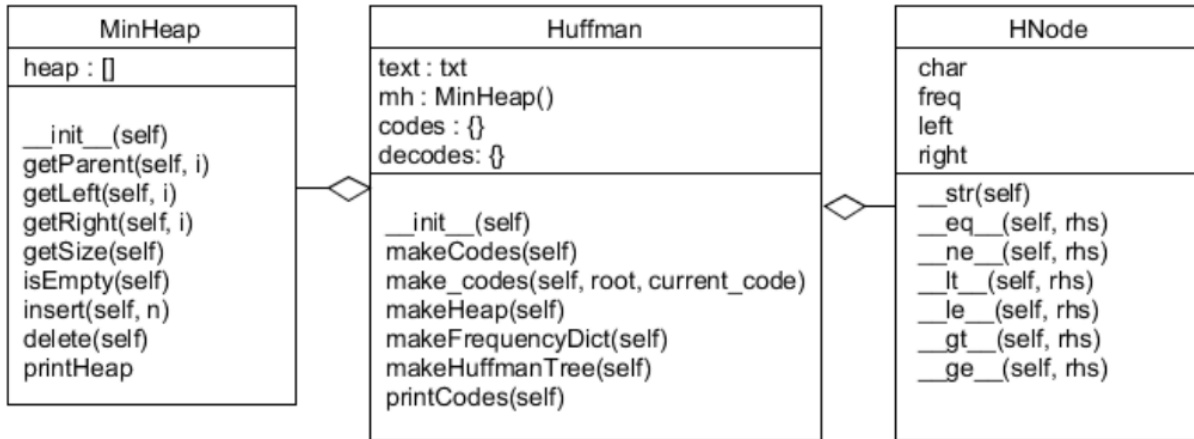
c:\4-2\Data Structure\Lab06_>python testLab06.py
Lab 06
i-insert, d-delete, p-print, s-search, q-quit ->i
> word: apple
> meaning : 사과
i-insert, d-delete, p-print, s-search, q-quit ->i
> word: banana
> meaning : 바나나
i-insert, d-delete, p-print, s-search, q-quit ->p
Dictionary :
(apple:사과)->(banana:바나나)->

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

```

1 class MinHeap:
2     def __init__(self):
3         self._heap = []
4         self._heap.append(0)
5
6     def getParent(self, i): return self._heap[i//2]
7     def getLeft(self, i): return self._heap[i*2]
8     def getRight(self, i): return self._heap[i*2 + 1]
9     def getSize(self): return len(self._heap) - 1
10    def isEmpty(self): return self.getSize() == 0
11    def __str__(self): return str(self._heap)
12    def insert(self, n):
13        self._heap.append(n)
14        i = self.getSize()
15        while i != 1 and n < self.getParent(i):
16            self._heap[i] = self.getParent(i)
17            i = i // 2
18        self._heap[i] = n
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37    def delete(self):
38        parent, child = 1, 2
39        if not self.isEmpty():
40            hroot = self._heap[1]
41            last = self._heap[self.getSize()]
42            while child <= self.getSize():
43                if child < self.getSize() and self.getLeft(parent) > self.getRight(parent):
44                    child += 1
45                if last <= self._heap[child]:
46                    break
47                self._heap[parent] = self._heap[child]
48                parent = child
49                child *= 2
50            self._heap[parent] = last
51            self._heap.pop(-1)
52            return hroot
53
54
55    def printHeap(self):
56        level = 1
57        for i in range(1, self.getSize() + 1):
58            if i == level:
59                print('')
60                level += 2
61            print(str(self._heap[i]), end = ' ')
62            print("\n-----")
63
64
65
66
  
```

Huffmancode.py # Huffman and HNode class


```
py x bstTree.py x expressionTree.py x morsecodes.py x testLab06.py x minheap.py x Huffmancode.py x Huffmancode.py
1 from minheap import *
2 class HNode:
3     def __init__(self, char=None, freq=None, left=None, right=None):
4         self.char = char
5         self.freq = freq
6         self.left = left
7         self.right = right
8
9     def __str__(self): return str(self.freq)
10    def __eq__(self, other): return self.freq == other.freq
11    def __ne__(self, other): return self.freq != other.freq
12    def __lt__(self, other): return self.freq < other.freq
13    def __le__(self, other): return self.freq <= other.freq
14    def __gt__(self, other): return self.freq > other.freq
15    def __ge__(self, other): return self.freq >= other.freq
16
17    class Huffman():
18        def __init__(self, txt=None):
19            self.text = txt
20            self.mh = MinHeap()
21            self.codes = {}
22            self.decodes = {}
23
24        def makeCodes(self):
25            root = self.makeHuffmanTree()
26            current_code = ""
27            self.make_codes(root, current_code)
28
29        def make_codes(self, root, current_code):
30            if root is None:
31                return
32
33            if root.char != None:
34                self.codes[root.char] = current_code
35                self.decodes[current_code] = root.char
36                return
37
38            self.make_codes(root.left, current_code + "0")
39            self.make_codes(root.right, current_code + "1")
40
41    def makeHeap(self):
42        frequencies = self.makeFrequencyDict()
43        for key in frequencies:
44            node = HNode(key, frequencies[key])
45            self.mh.insert(node)
46
47    def makeFrequencyDict(self):
48        frequencies = {}
49        for c in self.text:
50            if not c in frequencies:
51                frequencies[c] = 0
52            frequencies[c] += 1
53        return frequencies
54
55    def makeHuffmanTree(self):
56        self.makeHeap()
57        while self.mh.getSize() > 1:
58            p, q = self.mh.delete(), self.mh.delete()
59            r = HNode(None, p.freq + q.freq, p, q)
60            self.mh.insert(r)
61            self.mh.delete()
62
63    def getEncodeText(self, text):
64        encoded_text = ""
65        for character in text:
66            encoded_text += self.codes[character]
67        return encoded_text
68
69    def printCodes(self):
70        self.makeCodes()
71        for key in self.codes:
72            print(f"{key} : {self.codes[key]}")
73
74        print("Reverse Coding")
75        for key in self.decodes:
76            print(f"{key} : {self.decodes[key]}")
```

TestLab06.py

```
117 def useHuffman():
118     with open('document1.txt') as txt_file:
119         text = txt_file.read()
120         # text = "abcdefghijklmnopqrstuvwxyz"
121         hc = Huffman(text)
122         freq = hc.makeFrequencyDict()
123         for key in freq:
124             print(f"{key}:{freq[key]}")
125         hc.printCodes()
126
127     # Press the green button in the gutter to run the script.
128     def main():
129         print("Lab 06")
130         #useBST()
131         #useBTree()
132         #useBinaryNode()
133         #useMorseCodes() # 1_morsecode
134         #useET()
135         #useBinarySearchTree()
136         #useWMDic() # 2_BST
137         #useMinHeap()
138         useHuffman() # 3_HuffmanCode
139
140     if __name__ == '__main__':
141         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
i:1
s
5
i:1
w:1
e
o:3
i
n:1
s:1
e:1
k:1
c:1
h:1
k : 000
n : 001
s : 010
e : 0110
w : 0111
o : 10
c : 1100
m : 1101
i : 1110
h : 1111
Reverse Coding
000 : k
001 : n
010 : s
0110 : e
0111 : w
10 : o
1100 : c
1101 : m
1110 : i
1111 : h
Process finished with exit code 0
```

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.