EN 685.621 HW1 - Noboru Hayashi

1. a) Defined Node & LinkedList class to implement a linked list and read feature values

```
from __future__ import print_function

class Node:
   def __init__(self, data=None):
       self.data = data
       self.next = None
class LinkedList:
   def __init__(self, head_node=None):
       # set 2 pointers: head and last
       self.head = head_node
       self.tail = head_node
if __name__ == '__main__':
   with open("iris.csv") as file:
       file.readline() # skip header line

       # initialize a linked list with its head node, O(1)
       head_val = file.readline().strip().split(',')[0]
       head_node = Node(head_val)
       list = LinkedList(head_node)

       # read rest of the lines from the file,
       # create a new node and link to the tail of the linked list,
       # Cost of read, store, and link is O(1) per each element
       for line in file:
           next_val = line.strip().split(',')[0]
           next_node = Node(next_val)
           list.tail.next = next_node
           list.tail = list.tail.next
   # scan through the linked list from the head node for testing.
   # Output:    5.1 -> 4.9 -> 4.7 -> 4.6 -> 5.0 -> 5.4 -> 4.6 -> 5.0 -> 4.4 -> 4.9 ->
   nd = list.head
   while nd != None:
       print(nd.data,end=" -> ")
       nd = nd.next

# As mentioned above, inside the loop of read lines, cost of read, store and link is
O(1) per element.
# so for reading in the data, storing the keys and linking the feature data has total
running time is O(n)
```

b) Similar to part a, used Node class to implement a stack

```
from __future__ import print_function

class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None
class Stack:
    def __init__(self, top_node=None):
        # set a pointer: top for top of the stack
        self.top = top_node

if __name__ == '__main__':
    with open("iris.csv") as file:
        file.readline() # skip header line

        # initialize a stack with its top node, O(1)
        top_val = file.readline().strip().split(',')[0]
        top_node = Node(top_val)
        stack = Stack(top_node)

        # read rest of the lines from the file,
        # create a new node and link the stack's top to its next
        # then change the pointer to point the new node
        # Cost of read, store, and stack is O(1) per each element
        for line in file:
            next_val = line.strip().split(',')[0]
            next_node = Node(next_val)
            next_node.next = stack.top
            stack.top = next_node

    # scan through the stack by popping out the top node from it
    # Output: 5.9 -> 6.2 -> 6.5 -> 6.3 -> 6.7 -> 6.7 -> 6.8 -> 5.8 -> 6.9 ->. ..

    while stack.top != None:
        print(stack.top.data,end=" -> ")
        # pop action
        stack.top = stack.top.next

# As mentioned above, inside the loop of read lines, cost of read, store, and stack is
O(1) per each element
# so the total running time is O(n)
```

2. a) Defined HashTable class and method to hash & store an element.

```python
from linked_list import Node
class HashTable:
    def __init__(self, slot_size=12):
        # store size of the hash table
        self.slot_size = slot_size
        # store a list as a hash table
        self.hash_table = [None] * slot_size

    def hash_and_store(self, ob_id, node):
        # division method to hash the key: observation id, O(1)
        slot = ob_id % self.slot_size
        # link the head of the list next to the new node, O(1)
        node.next = self.hash_table[slot]
        # and change the head pointer to the new node, O(1)
        self.hash_table[slot] = node


if __name__ == '__main__':
    with open("iris.csv") as file:
        file.readline() # skip header line
        ob_id = 0
        h = HashTable()

        # read rest of the lines from the file,
        for line in file:
            # read and initialize a node
            data = line.strip().split(',')[0]
            next_node = Node(data)

            # hash the element with its key and store in a hash table
            h.hash_and_store(ob_id, next_node)

            # increment the key
            ob_id = ob_id + 1

    # As mentioned above, hash_and_store method is O(1)
    # and this method is operated at each line,
    # and given reading one line cost constant time O(1)
    # The overall time to read, hash and store is O(1) * n = O(n)
```

b) As an expansion of part a), instead of a single number, a list of feature values is stored in a node as data.

```
from linked_list import Node

class HashTable:
    def __init__(self, slot_size=12):
        # store size of the hash table
        self.slot_size = slot_size
        # store a list as a hash table
        self.hash_table = [None] * slot_size

    def hash_and_store(self, ob_id, node):
        # division method to hash the key: observation id
        slot = ob_id % self.slot_size
        node.next = self.hash_table[slot]
        self.hash_table[slot] = node


if __name__ == '__main__':

    with open("iris.csv") as file:

        file.readline() # skip header line
        ob_id = 0
        h = HashTable()

        # read rest of the lines from the file,
        for line in file:
            # read and initialize a node
            # different from Part a, the list of 4 features is stored as data of the
node
            data = line.strip().split(',')[0:4]
            next_node = Node(data)

            # hash the element with its key and store in a hash table
            h.hash_and_store(ob_id, next_node)

            # increment the key
            ob_id = ob_id + 1
```

3.

a) After tweaking the recommended evaluation formula and trying to incorporate the priority of board locations, I found the alternative formula below would give unique values for 14 boards.

Eval = 2 (3 * X2 + X1 - (3 * O2 + O1)) + p(X) + p(O)*
Where,
X2 is the number of lines with 2 X's and a blank
X1 is the number of lines with 1 X's and 2 blanks
O2 is the number of lines with 2 O's and a blank
X2 is the number of lines with 1 O's and 2 blanks

p(X) is the total score evaluated with the priority of board locations for the X player
p(O) is the total score evaluated with the priority of board locations for the O player

Code:

```
class Board:
   def __init__(self):
       # board state with 3x3 2d list, where 'O', ' ', 'X' each indicates O, empty, X
       self.state = None

   def eval_line(self, row1, col1, row2, col2, row3, col3):

       # concat characters
       line = self.state[row1][col1] + self.state[row2][col2] + self.state[row3][col3]

       # evaluate the line combination with following evaluation.
       # X2 = 3 points
       # X1 = 1 point
       # O2 = -3 points
       # O1 = 1 point
       if line == 'XX ' or line == 'X X' or line == ' XX': return 3
       if line == 'X  ' or line == ' X ' or line == '  X': return 1
       if line == 'OO ' or line == 'O O' or line == ' OO': return -3
       if line == 'O  ' or line == ' O ' or line == '  O': return -1

       # If 3 or -3 is got for every line (8 in total), total is 24 or -24, so set the
max/min integer as 100, -100.
       # We can evaluate if the score > 50 or score < -50 to check is the game over or
not.
       if line == 'XXX': return 100
       if line == 'OOO': return -100
```

```
        return 0

    def eval_pri(self):
        ret = 0

        pri_x = [[17, 13, 16],
                 [12, 18, 11],
                 [15, 10,14]]

        pri_o = [[8, 4, 7]
                ,[3, 9, 2]
                ,[6, 1, 5]]

        for i in range(3):
            for j in range(3):
                if self.state[i][j] == 'X':
                    ret = ret + pri_x[i][j]
                if self.state[i][j] == 'O':
                    ret = ret - pri_o[i][j]

        return ret

    def eval(self):
        score = 0

        # scores for vertical lines
        for i in range(3):
            score = score + self.eval_line(i, 0, i, 1, i, 2)

        # scores for horizontal lines
        for i in range(3):
            score = score + self.eval_line(0, i, 1, i, 2, i)

        # scores for diagonal lines
        score = score + self.eval_line(0, 0, 1, 1, 2, 2)
        score = score + self.eval_line(0, 2, 1, 1, 2, 0)

        # scores of locations based on the priority of board locations
        score =  2* score + self.eval_pri()

        return score
```

b)
Given the code above, 14 boards have unique scores as below:

Board 1: -11
Board 2: -6
Board 3: 8
Board 4: -2
Board 5: -15
Board 6: -7
Board 7: -17
Board 8: 10
Board 9: -3
Board 10: 20
Board 11: -1
Board 12: -14
Board 13: -5
Board 14: -13

Hence BST of 14 boards will be: