SEHH2239 2022/2023 Semester 2

Assignment 2

1. Student name: Chan Lok Hin
2. Student ID: 21088958A
3. Course code: SEHH2239
4. Course name: Data Structures
5. Class: 203D
6. Date: 26/4/2023

Declaration of Original Work

By submitting the answer script of this assignment to the subject lecturer through Moodle Centralized Group, you hereby declare that the work in the answer sheet is completely your own work. No part of the answer sheet is taken from other people's work without giving them credit. All references have been clearly cited.

You understand that an infringement of this declaration leaves you subject to disciplinary actions such as mark deduction, disqualification or even expulsion by the College.

If necessary, students may be invited to provide more information on their submission.

Question 1

Question 1(a)

```
class Stack:
    # Write your code for Question 1(a) here
    def __init__(self) :
        pass
        # implement a stack using list
        self.items = []

    def clear(self):
        pass
        # clear the stack
        self.items = []

    def isEmpty(self):
        pass
        # Check if the stack is empty
        # return True if the stack is empty
        # otherwise return False
        return len(self.items) == 0

    def push(self, data):
        pass
        # Put data on the top of the stack
        self.items.append(data)


    def pop(self):
        pass
        # Take the topmost data from the stack and return it
        # if the stack is empty, return None
        if self.isEmpty():
            return None
        else:
            return self.items.pop()

    def peek(self):
        pass
        # Return the topmost data in the stack
        # without removing it
        # if the stack is empty, return None
        if self.isEmpty():
            return None
        else:
            return self.items[-1]
```

```
        def display(self):
                pass
                # show all the data in the stack
                print(self.items)

# test
list = [1,2,3,4,5]
stack = Stack()

for x in list:
    stack.push(x)

stack.display()

print(stack.isEmpty())

print(stack.peek())

print(stack.pop())

print(stack.peek())

stack.display()

stack.clear()

stack.display()

print(stack.isEmpty())

print(stack.pop())

print(stack.peek())
```

```
[1, 2, 3, 4, 5]
False
5
5
4
[1, 2, 3, 4]
[]
True
None
None
```

Question 1(b)

```
class Student:
        def __init__(self,name,score):
                self.name = name
                self.score = score
        def __str__(self):
                return f"{self.name}:{self.score}"
        def __repr__(self):
                return str(self)
        def __eq__(self,other):
                return (self.score == other.score and self.name == other.name)
        def __gt__(self,other):
                if (self.score == other.score):
                        return self.name > other.name
                else:
                        return self.score > other.score
        def __lt__(self,other):
                if (self.score == other.score):
                        return self.name < other.name
                else:
                        return self.score < other.score
        def __ge__(self,other):
                return (not self < other)
        def __le__(self,other):
                return (not self > other)


class StackSort:
        # Write your code for Question 1(b) here
        def __init__(self,data):
                pass
                # define two Stack
                # the first Stack contains all the elements in data
```

```
        # and the second Stack left empty
        self.stack1 = data
        self.stack2 = []

    def sort(self):
        pass
        # sort the elements in the first Stack as follows:
        # While the first Stack is not empty, do this:
        # - Pop an element from the first Stack and call it temp
        # - While the second Stack is not empty and
        #     top of the second stack is greater than temp:
        #         - Pop from the second Stack and
        #             push it to the first Stack
        # - Push temp to the second Stack
        # the sorted elements are in second Stack
        while self.stack1:
            temp = self.stack1.pop()
            while self.stack2 and self.stack2[-1] > temp:
                self.stack1.append(self.stack2.pop())
            self.stack2.append(temp)

    def display(self):
        pass
        # show the data in two Stacks
        if self.stack2 == []:
            print(self.stack1)
        else:
            print(self.stack2)
```

Question 1(c)

Testing code for part (c): *(run the code cell below to generate outputs, do not alter the code in the cell)*

```
data = [Student("May",85),Student("Peter",30),Student("Louis",90),
Student("Frankie",30),Student("Ron",74)]
data1 = [Student("May",100),Student("Peter",90),Student("Louis",80),
Student("Frankie",70),Student("Ron",60)]
stackSort = StackSort(data)
print("Before stack sort:")
stackSort.display()
stackSort.sort()
print("After stack sort:")
stackSort.display()
```

```
Before stack sort:
[May:85, Peter:30, Louis:90, Frankie:30, Ron:74]
After stack sort:
[Frankie:30, Peter:30, Ron:74, May:85, Louis:90]
```

Provide **THREE** more testing cases for question 1(c)

```
# Write your testing code for Question 1(c) here
print('Test1:')
data1 = [Student("May",100),Student("Peter",90),Student("Louis",80),
Student("Frankie",70),Student("Ron",60)]
stackSort = StackSort(data1)
print("Before stack sort:")
stackSort.display()
stackSort.sort()
print("After stack sort:")
stackSort.display()

print('\nTest2:')
data2 = [Student("May",0),Student("Peter",10),Student("Louis",20),
Student("Frankie",30),Student("Ron",40)]
stackSort = StackSort(data2)
print("Before stack sort:")
stackSort.display()
stackSort.sort()
print("After stack sort:")
stackSort.display()

print('\nTest3:')
data3 = []
stackSort = StackSort(data)
print("Before stack sort:")
stackSort.display()
stackSort.sort()
print("After stack sort:")
```

```
stackSort.display()
```

```
Test1:
Before stack sort:
[May:100, Peter:90, Louis:80, Frankie:70, Ron:60]
After stack sort:
[Ron:60, Frankie:70, Louis:80, Peter:90, May:100]

Test2:
Before stack sort:
[May:0, Peter:10, Louis:20, Frankie:30, Ron:40]
After stack sort:
[May:0, Peter:10, Louis:20, Frankie:30, Ron:40]

Test3:
Before stack sort:
[]
After stack sort:
[]
```

Question 1(d)

Put down your answer for Question 1(d) here

O(n^2), it's nested while loop.

Double-click (or enter) to edit

Question 2

Question 2(a)

Put down your answer for Question 2(a) here

Double-click (or enter) to edit

$Index$	$Value$
0	$Empty$
1	$Empty$
2	$Empty$
3	$Empty$
4	$58(d) -> 85(f)$
5	$95(c)$
6	$Empty$
7	$88(a) -> 89(e) -> 98(g)$
8	$89(b)$

Question 2(b)

Put down your answer for Question 2(b) here

Double-click (or enter) to edit

$Index$	$Value$
0	$89(e)$
1	$98(g)$
2	$Empty$
3	$Empty$
4	$58(d)$
5	$95(c)$
6	$85(f)$
7	$88(a)$
8	$89(b)$

Question 2(c)

Put down your answer for Question 2(c) here

We cannot simply make the slot as empty, because this will isolate records further down the probe sequence, and it would hinder later searches.
For example, when put 'e' after 'b' is deleted, since 'Empty' still occupties the space, 'e' would be in [0] instead of [8].
Solution: Do a local reorganization upon deletion to try to shorten the average path length. For example, after deleting a key, continue to follow the probe sequence of that key and swap records further down the probe sequence into the slot of the recently deleted record.

Double-click (or enter) to edit

Provided Classes (for Questions 3 & 4)

Run the following code block to define some classes before running other code

```
class LLNode:
    def __init__(self, element=None, node=None):
        self.element = element
        self.next = node

class SLinkedList:
    def __init__(self):
        self.head = None

    def listprint(self):
        p = self.head
        while p is not None:
            print(p.element)
            p = p.next

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

    def size(self):
        size = 0
        p = self.head
        while p is not None:
            size += 1
            p = p.next
        return size

    def get(self, index):
        p = self.head
        i = 0
        while p is not None and i < index:
            p = p.next
            i += 1
        if p is not None:
            return p.element
        else:     # index >= self.size()
            return None

    def indexOf(self, element):
        p = self.head
        i = 0
        while p is not None and p.element != element:
            p = p.next
            i += 1
        if p is not None:
            return i
        else:
            return -1

    def addAtHead(self, element):
        self.head = LLNode(element, self.head)

    def addAtTail(self, element):
        if self.head is None:
            self.head = LLNode(element)
        else:
            p = self.head
            while p.next is not None:
                p = p.next
            p.next = LLNode(element)

    def add(self, index, element):
        if index == 0:
            self.addAtHead(element)
        else:
```

```
                p = self.head
                i = 0
                while p is not None and i < index - 1:
                        p = p.next
                        i += 1

                # The following statement will raise an error if index > self.size()
                p.next = LLNode(element, p.next)

        def remove(self, index):
                if self.isEmpty():
                        return None
                if index == 0:
                        element = self.head.element
                        self.head = self.head.next
                        return element
                else:
                        p = self.head
                        i = 0
                        while p is not None and i < index:
                                prev = p
                                p = p.next
                                i += 1

                        if p is None:
                                return None

                        element = p.element
                        prev.next = p.next
                        return element

        def removeNode(self, element):
                if self.isEmpty():
                        return None
                p = self.head
                if p.element == element:
                        self.head = p.next
                else:
                        while p is not None and p.element != element:
                                prev = p
                                p = p.next
                        if p is not None:
                                prev.next = p.next

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

        # A utility function to search a given key in BST
        def get(self, key):
                p = self

                while p is not None:
                        if p.key < key:
                                p = p.right
                        elif p.key > key:
                                p = p.left
                        else:
                                return p.data

                # no matching key
                return None

        def put(self, key, data):
                # Compare the new value with the parent node
                if self.key:
                        if key < self.key:
                                if self.left is None:
                                        self.left = BSTNode(key, data)
                                else:
                                        self.left.put(key, data)
                        elif key > self.key:
                                if self.right is None:
                                        self.right = BSTNode(key, data)
                                else:
                                        self.right.put(key, data)
                else:
                        self.key = key
                        self.data = data
```

```
        # To find the inorder successor which is the smallest node in the subtree
        def findsuccessor(self, node):
                current_node = node
                while current_node.left is not None:
                        current_node = current_node.left
                return current_node

        def remove(self, root, key):
                # Base Case
                if root is None:
                        return root

                # If the key to be deleted is smaller than the root's key then it lies in left subtree
                if key < root.key:
                        root.left = self.remove(root.left, key)

                # If the kye to be delete is greater than the root's key then it lies in right subtree
                elif key > root.key:
                        root.right = self.remove(root.right, key)

                # If key is same as root's key, then this is the node to be deleted
                else:
                        # Node with only one child or no child
                        if root.left is None:
                                temp = root.right
                                root = None
                                return temp

                        elif root.right is None:
                                temp = root.left
                                root = None
                                return temp

                        # Node with two children:
                        # Get the inorder successor (smallest in the right subtree)
                        temp = self.findsuccessor(root.right)

                        # Copy the inorder successor's content to this node
                        root.key = temp.key
                        root.data = temp.data      # copy by assignment

                        # Delete the inorder successor
                        root.right = self.remove(root.right, temp.key)

                return root

        # Print the tree
        def PrintTree(self):
                if self.left:
                        self.left.PrintTree()
                print(self.key, self.data),
                if self.right:
                        self.right.PrintTree()
```

Question 3

Question 3(a)

```
import re


class Student:
        # Write your code for Question 3(a) here
        pass

        def __init__(self, name=str, studentid=str):
            if re.fullmatch('[0-9]{8}A', studentid) is None:
                raise Exception(f'Invalid student id {studentid}')
            else:
                self.name = name
                self.studentid = studentid

        def __str__(self):
            return f"Name: {self.name}, ID: {self.studentid}"

        def __eq__(self, other):
```

```
            if isinstance(other, Student):
                return self.studentid == other.studentid
            else:
                raise Exception('Invalid data type!')
```

Testing code for part (a): (run the code cell below to generate outputs, do not alter the code in the cell)

```
student1 = Student('Dear Jane', '21666666A')
student2 = Student('Ka Yan 9896', '21009896A')
student3 = Student('Dear Jane', '21666666A')
try:
        Student('Quit gambling', '1834633')
except Exception as e:
        print(e)
print(student1)
print(student2)
print(student1 == student2)
print(student1 == student3)
try:
        print(student1 == 'Dear Jane')
except Exception as e:
        print(e)
```

```
Invalid student id 1834633
Name: Dear Jane, ID: 21666666A
Name: Ka Yan 9896, ID: 21009896A
False
True
Invalid data type!
```

Question 3(b)

```
class Registry:
        def __init__(self):
                self.BSTList = [None] * 10      # List of 10 BSTs

        # Question 3(b)
        def add_student(self, name=str, studentid=str):
                # Write your code for Question 3(b) here
                pass
                if re.fullmatch('[0-9]{8}A', studentid) is None:
                    print(f'Invalid student id {studentid}')
                elif self.BSTList[int(studentid[-2])] != None:
                    print(f'{name} {studentid} has already registered.')
                else:
                    self.BSTList[int(studentid[-2])] = BSTNode(studentid, Student(name, studentid))
                    print(f'Register {name} {studentid} successful.')

        # The following function is provided
        def print_students(self):
                for i, BST in enumerate(self.BSTList):
                        if BST is not None:
                                print(f'Students with ID ended with {i}:')
                                BST.PrintTree()
```

Testing code for part (b): (run the code cell below to generate outputs, do not alter the code in the cell)

```
student_records = [
        {'name': 'Chan Ka Yi', 'studentid': '21123456A'},
        {'name': 'So Fun Nei', 'studentid': '21888888A'},
        {'name': 'Ngor Mo Chin', 'studentid': '21654321A'}
]

registry = Registry()
for student_record in student_records:
        registry.add_student(**student_record)
print()
registry.add_student('SC Storage', '81777778')
print()
registry.add_student(**student_records[0])
print()
registry.print_students()
```

```
Register Chan Ka Yi 21123456A successful.
Register So Fun Nei 21888888A successful.
Register Ngor Mo Chin 21654321A successful.
```

```
Invalid student id 81777778

Chan Ka Yi 21123456A has already registered.

Students with ID ended with 1:
21654321A Name: Ngor Mo Chin, ID: 21654321A
Students with ID ended with 6:
21123456A Name: Chan Ka Yi, ID: 21123456A
Students with ID ended with 8:
21888888A Name: So Fun Nei, ID: 21888888A
```

Question 4

Question 4(a)

```
class Student:
    pass
    # Write your code for Question 4(a) here
    def __init__(self, name=str, studentid=str):
        self.name = name
        self.studentid = studentid
        self._courselist = SLinkedList()
```

Testing code for part (a): (run the code cell below to generate outputs, do not alter the code in the cell)

```
student1 = Student("Black Pink", "21732534A")
student1._courselist.addAtTail("SEHH2239")
student1._courselist.addAtTail("SEHH2240")
student1._courselist.listprint()
```

```
SEHH2239
SEHH2240
```

Question 4(b)

```
class Registry2(Registry):
    def __init__(self):
        super().__init__()

    # Question 4(b)
    def add_course(self, studentid=str, course=str):
        pass
        # Write your code for Question 4(b) here
        if re.fullmatch('[0-9]{8}A', studentid) is None:
            print (f'Invalid student id {studentid}')
        else:
            tree_id = int(studentid[-2])
            BST = self.BSTList[tree_id]
            student = BST.get(studentid) if BST is not None else None
            if student is None:
                print(f'{studentid} not a registered student.')
                return
            else:
                if student._courselist.indexOf(course) != -1:
                    print(f'{student.name} {student.studentid} has already enrolled {course}.')
                else:
                    student._courselist.addAtTail(course)
                    print(f'Enroll {student.name} {student.studentid} to {course} successful.')

    # The following function is provided
    def print_courses(self, studentid):
        if not isinstance(studentid, str):
            raise TypeError('Invalid input type')
        if re.fullmatch('[0-9]{8}A', studentid) is None:
            raise ValueError(f'Invalid student id {studentid}')

        tree_id = int(studentid[-2])
        BST = self.BSTList[tree_id]
        student = BST.get(studentid) if BST is not None else None
        if student is None:
            print(f'{studentid} not a registered student')
            return
```

```
            else:
                print(f'{student.name} {student.studentid} enrolled these courses:')
                student._courselist.listprint()
```

Testing code for part (b): (run the code cell below to generate outputs, do not alter the code in the cell)

```
student_records = [
    {'name': 'Chan Ka Yi', 'studentid': '21123456A'},
    {'name': 'So Fun Nei', 'studentid': '21888888A'},
    {'name': 'Ngor Mo Chin', 'studentid': '21654321A'}
]

course_records = [
    {'studentid': '21123456A', 'course': 'SEHH2239'},
    {'studentid': '21888888A', 'course': 'SEHH2239'},
    {'studentid': '21654321A', 'course': 'SEHH2240'},
    {'studentid': '21654321A', 'course': 'SEHH2241'},
]

registry = Registry2()
for student_record in student_records:
    registry.add_student(**student_record)
print()

for course_record in course_records:
    registry.add_course(**course_record)

print()
registry.add_course('XDJCNDEG', 'Random string')
registry.add_course('23000000A', 'Pizza Hut Hotline')
registry.add_course(**course_records[0])

print()
registry.print_courses('21654321A')
```

```
Register Chan Ka Yi 21123456A successful.
Register So Fun Nei 21888888A successful.
Register Ngor Mo Chin 21654321A successful.

Enroll Chan Ka Yi 21123456A to SEHH2239 successful.
Enroll So Fun Nei 21888888A to SEHH2239 successful.
Enroll Ngor Mo Chin 21654321A to SEHH2240 successful.
Enroll Ngor Mo Chin 21654321A to SEHH2241 successful.

Invalid student id XDJCNDEG
23000000A not a registered student.
Chan Ka Yi 21123456A has already enrolled SEHH2239.

Ngor Mo Chin 21654321A enrolled these courses:
SEHH2240
SEHH2241
```

Question 4(c)

(i) *Put down your answer for Question 4(c)(i) here*

Time complexity: O(n), need to check all the student once.

(ii) *Put down your answer for Question 4(c)(ii) here*

Give a count of students enrolled in a class, when the time of finding the student reach the count, stop running the program.

Double-click (or enter) to edit

When you finish the assignment:

- Click [Runtime -> Restart and run all] to regenerate all outputs
- Click [File -> Download -> Download .ipynb]
- Click [File -> Print -> Save as PDF]
- Follow the instructions on the question paper to submit both files

✓ 0 秒　完成時間：下午4:33