Assignment 2 Report

Harsh Patel patelh75

February 18, 2019

The purpose of this report is to give a more in-depth insight of the test cases, note any assumptions or improvements in the program, and comparing our module with the module of a selected partner.

1 Testing of the Original Program

The approach to testing the functionality of the modules was to create test cases for the abstract data types being tested, and run the testAll.py file through pytest unit testing. Running it through pytest would display the number of test cases failed, passed, and total amount of tests that were compiled. Test cases for the modules consist of normal and boundary test cases. When testing my modules with pytest, all 35 test cases passed with an overall coverage of 88%. The first module that was tested was SeqADT.py. The first test case for this module was to test the start() function and ensure it printed a none type instead of printing the value of i, which is the index, since this function had no return type. The subsequent test cases tested the functionality of the next() function. The first two normal test cases would call the next() function to test if the first element, s[0], would be returned when next() was called once, and if the second element was returned when next() was called twice. These first two test cases tested the basic functionality of the next() function. The third test case for this module tested the StopIteration exception for the next() function. If next() was called too many times, such that the index would be equal or greater then the length of the sequence, the exception should be raised. This test cases was important as it would inform the user of the issue, and to test that the exception should only be raised if that specific condition was met. The final test case for the next() function was to pass an empty sequence and call next() to test the result returned. This raised the StopIteration exception as expected. The last few test cases for SeqADT.py module tested the end() function. The first two test case for the end() function tested if the index, i, was referencing the end of the sequence, that is length of the sequence - 1. In the first test case, next() was called to increment i, but not enough times to reach the end of the sequence. This test returned a boolean False as expected when end() was called, indicating that index, i, was not at the end of the sequence. The second test case called the next() function till the index was pointing to the end of the sequence. This test returned a boolean True representing that the user iterated to the end of the list. The third test case for this function tested what would happen if the index i went over the length of the sequence to test if a boolean is returned or an exception from the next() function is raised. The exception, StopIteration, was raised for this test case. The final test case tested an empty sequence, and when end() was called, it returned a boolean value of True. The second module tested was DCapALst.py. The first two test cases tested the *add* function. The first test case was a normal test case that would add a department and its corresponding capacity to the sequence. The next test case checked if an exception, KeyError, was raised if the same department was being added again to the sequence. This was important to test if the exception was raised correctly as if it was not raised then adding the same department again would cause more errors in the allocation steps. The following two test cases test the *remove* function. The first test case for this function is a normal test case that removes a department and its corresponding capacity that are in the sequence. The next test case tests if the exception, KeyError, is raised if the department being removed is not present in the sequence. The next two test cases test the *elm* function. This function has no exceptions, but simply returns a boolean value representing if the department is in the sequence or not. The test cases for this function returned true if the department being tested for was in the sequence, and false otherwise. The last function being tested for this module was the *capacity* function. The test case for this function accepted a department name and would return the corresponding capacity, otherwise a KeyError raised if the department was not in the sequence or if an invalid input such as an int, float, or string was inputted as a parameter. The last module being tested is the SALst.py module. The first few test cases for the functions *add*, *remove*, *elm*, and *info* follow the same test logic as the *add*, *remove*, *elm*, and *capacity* functions from DCapALst.py respectively, except using student's macid and information. The following test cases tested the sort function. The first test case for this function accepted a filter condition and tested to see if the result matched the filtered result which could be anything from a gpa condition, a student having free choice, gender, etc. The second test case tested another filter condition to gain assurance that the function was able to read the different filter conditions. The next section of test cases tested the function responsible for calculating the average GPA of a specific gender in the sequence of student information. The first two test cases returned the average GPA of male or female students depending on gender chosen by the lambda filter. The next two test cases were designed to discover the result when no members of a specific gender were in the student list, and the program tested for that specific gender's average GPA. This would raise the exception, ValueError, indicating that the gender is not present as a value of the student information tuple, therefore no members of this specific gender are present. The final section of test cases for this module tested the allocate function. The first test case was a normal allocation of students to their choices with each student having a different first choice, and being allocated. The result was tested through comparing the length of the allocation list to the result expected. The second test case tested the result of when every faculty was full or had 0 capacity in which case the students wouldn't be allocated to any of their choices. An exception of RuntimeError was raised as expected. The last test case tested if a student would be allocated to their second choice, if their first choice was full. No errors were raised when unit testing as all functions were individually being tested in the coding process itself. Therefore, no problems were uncovered when testing.

2 Results of Testing Partner's Code

The test cases tested with the partners modules resulted in 29 test cases passed, and 6 test cases failed. The test cases that passed tested for boolean values, raising exceptions, returning an average, returning a sorted array, or testing the length of the allocated list after allocation. The test cases that failed were the ones that asserted an equivalence statement that would test the format of the sequence after performing a function such as add. For example, if s.add(DeptT.software, 5) is called, then the assert statement would test if s.s == [{"dept":"software", "cap":5}]. The issue with this is that the partner did not use the same format as I did to store the information, therefore the format of the sequences in each module were different causing the test to fail. The assertion statements that would also access information, such as getting macid or student info, also failed as the format of the sequences are different, then the code needed to access the information would also be different. A simple solution to this issue is to assert on boolean statements. For example, if s.add(DeptT.software, 5) is called, then the *elm* function can be used to return a boolean to check if the department was added. In this way assert True == True can be used to test it. To check for the capacity, "if s.capacity(d)" can be used to either print the result or return a boolean. Using this strategy, failed tests caused by formatting assumptions of sequences can be avoided. Other test cases which required us to make no assumptions about formatting, returned exceptions, boolean values, or specific formatted objects all resulted in the test cases passing. Another test case that failed was an error in my code, in which I raised a ValueError instead of a KeyError in the *info* function. This resulted in a test case failing as the partner's code raised a KeyError as specified in the assignment specification.

3 Critique of Given Design Specification

As opposed to the natural language specification, the mathematical specification did not require the programmer to make many assumptions as most guidelines for the modules and functions were specific in the task required to be completed. Major assumptions that were part of the natural language specification were also explicitly stated in this assignment. The only assumptions required for this assignment was the structure of the sequences and tuples in the modules. The advantage of this specification is that it is very specific, stating accurately on what the user expects for the behavior of the program. Reading mathematical specification is also an effective use to express specific details for implementation in a concise way instead of explaining using many sentences and paragraphs. The disadvantage of this type of design specification is that it doesn't allow the programmer to be creative or implement more effective ways to complete the task. For this reason, I feel this specification should have a section in which the programmer can reflect upon the functions and implementation instructions used, and suggest more effective implementation techniques. The development of this type of thinking is important in the software industry.

4 Answers

1. The natural language of A1 required the programmer to make many assumptions and implement the modules based on those assumptions. The natural language specification heavily depended on the decisions of the programmer such as format of the text files being read, rounding the gpa of students, etc. There were many advantages to working through this kind of assignment. It gives a representation of how problems in the real world are solved. In the software industry a problem is simply presented with no specific specification, and requires the programmer to think of the many approaches and find the most efficient one to implement. Decision making and assumptions is important to solving problems with code as this allows us to develop a thinking to solve many errors, and looking at the program from the perspective of the user. The formal specification did not require the programmer to make many assumptions or implementation decisions. The programmer has to follow the implementation instructions as stated such as mathematical specifications for the functions, return types, and exceptions raised. The advantage of the mathematical specification is that it is an efficient way to specifically state the implementation details for the programmer as opposed to writing paragraphs and sentences to express the details such as in the natural language specification. The disadvantage of a formal specification is that it doesn't give the programmer the freedom to implement more efficient solutions to the given problem, and it does not

allow the programmer to develop the thinking of independently building a program from scratch based on best assumptions and efficient implementation techniques.

2. The assumption that the gpa will be between 0 and 12.0 can be changed to an exception by having a function to check if the gpa is in the range of 0 to 12.0. If that condition is satisfied, then a float of the gpa is returned, but if the condition is not satisfied, then a ValueError can be returned. The student record type would need to modified from gpa : float to gpa : checkGpa(float).

3. The documentation can be modified such that the DCapALst module can be removed as SALst shares its similar functions. The *init* function of SALst can accomodate to initialize both the sequence of DCapALst and SALst assigned to different variables such as d and s, respectively. The other similar functions would simply need to accept a third parameter indicating the sequence to be accessed. For example add(a, b, s) for student info sequence of SALst and add(a, b, d) for department sequence. The capacity and info function are also very similar, and a simple name change to something more general is required if we implement the following changes stated above. In this way, the DCapALst can be removed from the documentation and merged with SALst since they share similar functions.

4. A2 is more general then A1 in terms of the implementation and functions. The sort function in A1 was only performing sorts based on gpa in descending order accepting all students in the list, even the ones that don't meet certain requirements such as having a gpa over 4.0. The sort function in A2 is more general. It still sorts based on descending gpa of students, but the function is now able accept a filter condition that returns a sorted list of students that pass the filter condition. For example, if we want students that have free choice and a gpa of 4.0 or greater, students that are only male or female, students with software as their first choice, etc. Any filter condition can be set based on the tuple of SInfoT. The average function in A1 only calculated the average gpa of male or female students selectively. The A2 average function has filter condition which can allow calculating the average gpa of males, females, both genders, males in a specific program, females in a specific program, etc. Any filter condition can be passed in to calculate the gpa of any group of students. In this way, both sort and average have become very general as they are not limited to only one specific task or condition.

5. The advantage of representing the list of choices for each student by a custom object, SeqADT, over a regular list is that the functions of SeqADT are very useful when allocating students. Having the student choices be of type SeqADT allows the programmer to have access to the functions used to iterate over a list, check if

the end of the list is reached, and reset the iterator, i to 0 without explicitly coding those functions in the allocate function resulting in more efficient code.

6. The advantage of enums over strings is that a string is case sensitive and relies on spelling while an enum is treated more as a constant, so it can't be misspelled without raising an error. A typo in a string is not caught by the compiler, but rather in runtime while typos in enums are caught by the compiler. Enums are also known to be more memory efficient then strings. The macids specification did not implement an enum because macids are unique for each student and can't be represented as a constant such as the set of gender, male and female, or set of departments. If macids were to be represented as an enum then the programmer would have to include all the unique macids of students in code in the enum class for it, and assign each of it a unique value. This would take too much time, making the use of an enum in this case redundant as it would become very long as the list of students is expected to be long.

E Code for StdntAllocTypes.py

```
## @file StdntAllocTypes.py
#  @author Harsh Patel
#  @brief Enumerated types and NamedTuples
#  @date 11/02/2019


from enum import Enum
from typing import NamedTuple
from SeqADT import *


## @brief An enumerated type representing the gender set
class GenT(Enum):
    male = 1
    female = 2


## @brief An enumerated type representing the department set
class DeptT(Enum):
    civil = 1
    chemical = 2
    electrical = 3
    mechanical = 4
    software = 5
    materials = 6
    engphys = 7


## @brief A named tuple representing general information for a student
class SInfoT(NamedTuple):
    fname: str
    lname: str
    gender: GenT
    gpa: float
    choices: type(SeqADT(DeptT))
    freechoice: bool
```

F Code for SeqADT.py

```
## @file SeqADT.py
#  @author Harsh Patel
#  @brief SeqADT
#  @date 11/02/2019


## @brief Abstract data type for sequence manipulation
class SeqADT:
    ## @brief SeqADT constructor
    #  @details takes a list of values
    #  @param x list of values
    def __init__(self, x):
        self.s = x
        self.i = 0

    ## @brief start function
    #  @details sets index, i, of list to 0
    def start(self):
        self.i = 0

    ## @brief next function
    #  @details increments index, i, of list
    #  @throw StopIteration if index, i, is equal to the length of list
    #  @returns element of list at index, i, before increment
    def next(self):
        if (self.i >= len(self.s)):
            raise(StopIteration)
        self.i += 1
        return self.s[(self.i) - 1]

    ## @brief end function
    #  @details checks if index reached end of list
    #  @returns boolean value representing if index, i, is at the
    #  end of the list
    def end(self):
        return self.i >= len(self.s)
```

G Code for DCapALst.py

```
## @file DCapALst.py
#  @author Harsh Patel
#  @brief DCapALst
#  @date 11/02/2019


## @brief DCapALst is an abstract data type
class DCapALst:
    s = []

    ## @brief init function
    #  @details Initializes the sequence to an empty sequence
    @staticmethod
    def init():
        DCapALst.s = []

    ## @brief add function
    #  @details Adds a department and its capacity to the sequence
    #  @param d Department name
    #  @param n Department capacity
    #  @throw KeyError raised if department already exists in the sequence
    @staticmethod
    def add(d, n):
        for department in DCapALst.s:
            if (department["dept"] == d.name):
                raise(KeyError)
        DCapALst.s.append({"dept": d.name, "cap": n})

    ## @brief remove function
    #  @details Removes a department and its capacity from the sequence
    #  @param d Department name
    #  @throw KeyError if department is not in sequence
    @staticmethod
    def remove(d):
        if not(DCapALst.elm(d)):
            raise(KeyError)
        for s_info in DCapALst.s:
            if (d.name == s_info["dept"]):
                DCapALst.s.remove(s_info)
                break

    ## @brief elm function
    #  @details Checks if the department is in the sequence
    #  @param d Department name
    #  @return Boolean value representing if department is in the sequence
    @staticmethod
    def elm(d):
        for department in DCapALst.s:
            if department["dept"] == d.name:
                return True
        return False

    ## @brief capacity function
    #  @details Gets capacity of the department
    #  @param d Department name
    #  @throw KeyError raised if department not in the sequence
    #  @return Returns department capacity
    @staticmethod
    def capacity(d):
        for d_info in DCapALst.s:
            if (d.name == d_info["dept"]):
                return d_info["cap"]
        raise(KeyError)
```

H Code for AALst.py

```
## @file AALst.py
#  @author Harsh Patel
#  @brief AALst
#  @date 11/02/2019


from StdntAllocTypes import *


## @brief Abstract data type that represents the allocated sequence
class AALst:
    s = []

    ## @brief init function
    #  @details Initilizes a sequence with tuples of
    #  department name and empty list
    @staticmethod
    def init():
        for department in DeptT:
            AALst.s.append({"dept": department.name, "LstStdnts": []})

    ## @brief add_stdnt function
    #  @details Adds students to the list of the corresponding department
    #  @param d Department name
    #  @param m macid of the student
    @staticmethod
    def add_stdnt(d, m):
        for department in AALst.s:
            if (department["dept"] == d.name):
                department["LstStdnts"].append(m)

    ## @brief lst_alloc function
    #  @details Retrieves the list of macids of students
    #  allocated to the department
    #  @param d Department name
    #  @return List of macids of students allocated to the department
    @staticmethod
    def lst_alloc(d):
        for department in AALst.s:
            if (department["dept"] == d.name):
                return department["LstStdnts"]

    ## @brief num_alloc function
    #  @details Number of students allocated to a department
    #  @param d Department name
    #  @return length of a list corresponding to a department
    #  repersting number of students allocated
    @staticmethod
    def num_alloc(d):
        for department in AALst.s:
            if (department["dept"] == d.name):
                return len(department["LstStdnts"])
```

I Code for SALst.py

```
## @file SALst.py
#  @author Harsh Patel
#  @brief SALst
#  @date 11/02/2019


from StdntAllocTypes import *
from AALst import *
from DCapALst import *


## @brief Abstract data type that represents a sequence
#  with student information
class SALst:
    s = []

    ## @brief init function
    #  @details Initilizes an empty sequence
    @staticmethod
    def init():
        SALst.s = []

    ## @brief add function
    #  @details Adds a student's information to the sequence
    #  @param m macid of the student
    #  @param i Student information of type SInfoT
    #  @throw KeyError raised if information for student already
    #  exists in the sequence
    @staticmethod
    def add(m, i):
        if {"macid": m, "info": i} in SALst.s:
            raise(KeyError)
        else:
            SALst.s.append({"macid": m, "info": i})

    ## @brief remove function
    #  @details Removes the information of the student in the sequence
    #  @param m macid of the student
    #  @throw KeyError raised if macid not in the sequence
    @staticmethod
    def remove(m):
        if (not(SALst.elm(m))):
            raise(KeyError)
        for student in SALst.s:
            if(student["macid"] == m):
                SALst.s.remove(student)
                break

    ## @brief elem function
    #  @details Checks if student information is in sequence
    #  @param m macid of the student
    #  @return Boolean value representing if information in sequence
    @staticmethod
    def elm(m):
        for student in SALst.s:
            if student["macid"] == m:
                return True
        return False

    ## @brief info function
    #  @details Retrieves a student's information based on macid
    #  @param m macid of the student
    #  @throw ValueError raised if student is not in sequence
    #  @return Information of the specific student of type SInfoT
    @staticmethod
    def info(m):
        for student in SALst.s:
            if student["macid"] == m:
                return student["info"]
        raise(ValueError)

    ## @brief sort function
    #  @details Sorts the students in order of descending gpa
    #  @param f condition that filters macids that don't meet its standards
    #  @return A sequence of macids in order of descending gpa of
    #  the students that passed the condition f
```

```
@staticmethod
def sort(f):
    stdnt_lst = [stdnt["info"] for stdnt in SALst.s]
    filter_lst = list(filter(f, stdnt_lst))
    id_lst = [stdnt["macid"] for stdnt in SALst.s
              if stdnt["info"] in filter_lst]

    def get_gpa(m, s):
        for stdnts in s:
            if stdnts["macid"] == m:
                std_info = stdnts["info"]
                return std_info.gpa

    for stdnt_pos in range(len(id_lst) - 1):
        stdnt_1 = get_gpa(id_lst[stdnt_pos], SALst.s)
        stdnt_2 = get_gpa(id_lst[stdnt_pos + 1], SALst.s)
        if (stdnt_2 > stdnt_1):
            temp = id_lst[stdnt_pos]
            id_lst[stdnt_pos] = id_lst[stdnt_pos + 1]
            id_lst[stdnt_pos + 1] = temp
    return id_lst

## @brief average function
#  @details calculates the average gpa students that
#  meet the condition specified
#  @param f condition that filters students that don't meet its standards
#  @return Average gpa of students that meet condition f
def average(f):
    f_set = [stdnt["info"] for stdnt in SALst.s]
    filter_lst = list(filter(f, f_set))
    if len(filter_lst) == 0:
        raise(ValueError)
    tot_gpa = sum(student.gpa for student in filter_lst)
    return tot_gpa / len(filter_lst)

## @brief allocate function
#  @details Allocates students to their respective departments
#  based on choices selected, Gpa, and freechoice
def allocate():
    AALst.init()
    F = SALst.sort(lambda t: t.freechoice and t.gpa >= 4.0)
    for m in F:
        ch = SALst.info(m).choices
        AALst.add_stdnt(ch.next(), m)

    S = SALst.sort(lambda t: not(t.freechoice) and t.gpa >= 4.0)
    for m in S:
        ch = SALst.info(m).choices
        alloc = False
        while not(alloc) and not(ch.end()):
            d = ch.next()
            if AALst.num_alloc(d) < DCapALst.capacity(d):
                AALst.add_stdnt(d, m)
                alloc = True
        if not(alloc):
            raise(RuntimeError)
```

J Code for Read.py

```
## @file Read.py
#  @author Harsh Patel
#  @brief Reads files and loads data
#  @date 11/02/2019


from StdntAllocTypes import *
from DCapALst import *
from SALst import *


## @brief Reads stduent data from text file
#  @details Populates SALst with student data from the text file
#  @param s String representing name of the text file
def load_stdnt_data(s):
    SALst.init()
    software = DeptT.software
    civil = DeptT.civil
    mechanical = DeptT.mechanical
    materials = DeptT.materials
    electrical = DeptT.electrical
    engphys = DeptT.engphys
    chemical = DeptT.chemical

    with open(s, "r") as student_files:
        student_data = student_files.read()
        student_data = student_data.replace(",", "")
        student_data = student_data.replace("[", "")
        student_data = student_data.replace("]", "")
        student_data = student_data.splitlines()
        for lines in student_data:
            std_data = lines.split()
            macid = std_data[0]
            fir_name = std_data[1]
            las_name = std_data[2]
            if (std_data[3] == "male"):
                gend = GenT.male
            else:
                gend = GenT.female
            gpa = float(std_data[4])
            prog_choic = []
            for i in range(5, len(std_data) - 1):
                program_type = vars()[std_data[i]]
                prog_choic.append(program_type)
            if (std_data[-1] == "True"):
                free_ch = True
            else:
                free_ch = False
            info = SInfoT(fir_name, las_name, gend, gpa,
                          SeqADT(prog_choic), free_ch)
            SALst.add(macid, info)


## @brief Reads department data from text file
#  @details Populates DCapALst with department and capacity data
#  @param s String representing name of the text file
def load_dcap_data(s):
    DCapALst.init()
    software = DeptT.software
    civil = DeptT.civil
    mechanical = DeptT.mechanical
    materials = DeptT.materials
    electrical = DeptT.electrical
    engphys = DeptT.engphys
    chemical = DeptT.chemical

    with open(s, "r") as dept_info:
        dept_info = dept_info.read()
        dept_info = dept_info.replace(",", "")
        dept_info = dept_info.splitlines()
        for lines in dept_info:
            dept_infolst = lines.split()
            department = vars()[dept_infolst[0]]
            capacity = int(dept_infolst[1])
            DCapALst.add(department, capacity)
```

K Code for test_All.py

```
import pytest
from SALst import *
from DCapALst import *
from SeqADT import *

class TestSeqADT:
    def test_start_none(self):
        seqADT1 = SeqADT([1,2,3])
        assert print(seqADT1.start()) == None

    def test_first_next_element_0(self):
        seqADT2 = SeqADT(["Hey",2,3,4,5])
        assert seqADT2.next() == "Hey"

    def test_second_next_element_1(self):
        seqADT3 = SeqADT(["Hey", True, 8, 9])
        seqADT3.next()
        assert seqADT3.next() == True

    def test_exception_next(self):
        seqADT3 = SeqADT(["Hey", True, 8])
        seqADT3.next()
        seqADT3.next()
        seqADT3.next()
        with pytest.raises(StopIteration):
            seqADT3.next()

    def test_end_not_end(self):
        seqADT4 = SeqADT([1,"Hello", 2])
        seqADT4.next()
        seqADT4.next()
        assert seqADT4.end() == False

    def test_end_at_end(self):
        seqADT5 = SeqADT([1,"Hello", 2])
        seqADT5.next()
        seqADT5.next()
        seqADT5.next()
        assert seqADT5.end() == True

    def test_end_over_end(self):
        seqADT6 = SeqADT([1,"Hello", 2])
        seqADT6.next()
        seqADT6.next()
        seqADT6.next()
        with pytest.raises(StopIteration):
            seqADT6.next()
            seqADT6.end()

    def test_empty(self):
        empty = SeqADT([])
        with pytest.raises(StopIteration):
            empty.next()

    def test_empty2(self):
        empty2 = SeqADT([])
        assert empty2.end() == True

class TestDCapALst:

    def test_add_normal(self):
        s = DCapALst
        s.init()
        s.add(DeptT.software, 5)
        assert s.s == [{"dept":"software", "cap":5}]

    def test_add_same(self):
        s = DCapALst
        s.init()
        s.add(DeptT.software, 5)
        with pytest.raises(KeyError):
            s.add(DeptT.software, 7)

    def test_remove_normal(self):
        s = DCapALst
        s.init()
        s.add(DeptT.software, 5)
        s.add(DeptT.chemical, 9)
        s.remove(DeptT.chemical)
        assert s.s == [{"dept":"software", "cap":5}]

    def test_remove_not_present(self):
        s = DCapALst
        s.init()
        s.add(DeptT.software, 5)
        s.add(DeptT.chemical, 9)
        with pytest.raises(KeyError):
            s.remove(DeptT.civil)

    def test_elem_normal(self):
        s = DCapALst
        s.init()
        s.add(DeptT.software, 5)
        s.add(DeptT.chemical, 9)
        assert s.elm(DeptT.chemical) == True

    def test_elem_not_there(self):
        s = DCapALst
        s.init()
        s.add(DeptT.software, 5)
        s.add(DeptT.chemical, 9)
        assert s.elm(DeptT.civil) == False

    def test_capacity_normal(self):
        s = DCapALst
        s.init()
        s.add(DeptT.software, 5)
        s.add(DeptT.chemical, 9)
        assert s.capacity(DeptT.software) == 5

    def test_capacity_not_there(self):
        s = DCapALst
        s.init()
        s.add(DeptT.software, 5)
        s.add(DeptT.chemical, 9)
        with pytest.raises(KeyError):
            s.capacity(DeptT.civil)

class TestSALst:

    def test_add_normal(self):
        t = SALst
        t.init()
        m = "patelh75"
        i = SInfoT("first", "last", GenT.male, 12.0, SeqADT([DeptT.civil, DeptT.chemical]), True)
        t.add(m,i)
        assert (t.s[0]["macid"] == m and t.s[0]["info"] == i)

    def test_add_already_added(self):
        t = SALst
        t.init()
        m = "patelh75"
        i = SInfoT("first", "last", GenT.male, 12.0, SeqADT([DeptT.civil, DeptT.chemical]), True)
        t.add(m,i)
        with pytest.raises(KeyError):
            t.add(m,i)

    def test_add_another(self):
        t = SALst
        t.init()
        m = "patelh75"
        m1 = "hero1"
        i = SInfoT("first", "last", GenT.male, 12.0, SeqADT([DeptT.civil, DeptT.chemical]), True)
        i1 = SInfoT("fsdfirst", "lasdfst", GenT.female, 11.0, SeqADT([DeptT.software,
            DeptT.chemical]), False)
        t.add(m,i)
        t.add(m1,i1)
        assert (t.s == [{"macid":m,"info":i},{"macid":m1,"info":i1}])

    def test_remove_normal(self):
        t = SALst
        t.init()
        m = "patelh75"
        m1 = "hero1"
        i = SInfoT("first", "last", GenT.male, 12.0, SeqADT([DeptT.civil, DeptT.chemical]), True)
        i1 = SInfoT("fsdfirst", "lasdfst", GenT.female, 11.0, SeqADT([DeptT.software,
            DeptT.chemical]), False)
        t.add(m,i)
        t.add(m1,i1)
        t.remove(m1)
        assert (t.s == [{"macid":m,"info":i}])

    def test_remove_already_removed(self):
        t = SALst
        t.init()
        m = "patelh75"
        m1 = "hero1"
        i = SInfoT("first", "last", GenT.male, 12.0, SeqADT([DeptT.civil, DeptT.chemical]), True)
        i1 = SInfoT("fsdfirst", "lasdfst", GenT.female, 11.0, SeqADT([DeptT.software,
            DeptT.chemical]), False)
        t.add(m,i)
        t.add(m1,i1)
        t.remove(m1)
        with pytest.raises(KeyError):
            t.remove(m1)

    def test_elm_normal(self):
        t = SALst
        t.init()
        m = "patelh75"
        m1 = "hero1"
        i = SInfoT("first", "last", GenT.male, 12.0, SeqADT([DeptT.civil, DeptT.chemical]), True)
        i1 = SInfoT("fsdfirst", "lasdfst", GenT.female, 11.0, SeqADT([DeptT.software,
            DeptT.chemical]), False)
        t.add(m,i)
        t.add(m1,i1)
        assert t.elm(m) == True

    def test_elm_not_there(self):
        t = SALst
        t.init()
        m = "patelh75"
        m1 = "hero1"
        i = SInfoT("first", "last", GenT.male, 12.0, SeqADT([DeptT.civil, DeptT.chemical]), True)
        i1 = SInfoT("fsdfirst", "lasdfst", GenT.female, 11.0, SeqADT([DeptT.software,
            DeptT.chemical]), False)
        t.add(m,i)
        assert t.elm(m1) == False

    def test_info_normal(self):
        t = SALst
        t.init()
        m = "patelh75"
        m1 = "hero1"
        i = SInfoT("first", "last", GenT.male, 12.0, SeqADT([DeptT.civil, DeptT.chemical]), True)
        i1 = SInfoT("fsdfirst", "lasdfst", GenT.female, 11.0, SeqADT([DeptT.software,
            DeptT.chemical]), False)
        t.add(m,i)
        t.add(m1,i1)
        assert t.info(m) == i

    def test_info_not_there(self):
        t = SALst
        t.init()
        m = "patelh75"
        m1 = "hero1"
        i = SInfoT("first", "last", GenT.male, 12.0, SeqADT([DeptT.civil, DeptT.chemical]), True)
        i1 = SInfoT("fsdfirst", "lasdfst", GenT.female, 11.0, SeqADT([DeptT.software,
            DeptT.chemical]), False)
        t.add(m,i)
        with pytest.raises(ValueError):
            t.info(m1)

    def test_sort_normal(self):
        t = SALst
        t.init()
        m = "patelh75"
        m1 = "hero1"
        m2 = "me2"
        i = SInfoT("first", "last", GenT.male, 10.5, SeqADT([DeptT.civil, DeptT.chemical]), True)
        i1 = SInfoT("fsdfirst", "lasdfst", GenT.female, 11.0, SeqADT([DeptT.software,
            DeptT.chemical]), False)
        i2 = SInfoT("firstttt", "lastttt", GenT.male, 12.0, SeqADT([DeptT.materials, DeptT.chemical]),
            True)
        t.add(m,i)
        t.add(m1,i1)
        t.add(m2,i2)
        lst = t.sort(lambda t: t.freechoice and t.gpa >= 4.0)
        assert lst == [m2,m]

    def test_sort_another_condition(self):
        t = SALst
        t.init()
        m = "patelh75"
        m1 = "hero1"
        m2 = "me2"
        i = SInfoT("first", "last", GenT.male, 10.5, SeqADT([DeptT.civil, DeptT.chemical]), True)
        i1 = SInfoT("fsdfirst", "lasdfst", GenT.female, 11.0, SeqADT([DeptT.software,
             DeptT.chemical]), False)
        i2 = SInfoT("firstttt", "lastttt", GenT.male, 12.0, SeqADT([DeptT.materials, DeptT.chemical]),
             True)
        t.add(m,i)
        t.add(m1,i1)
        t.add(m2,i2)
        lst = t.sort(lambda t: t.freechoice)
        assert lst == [m2,m]

    def test_average_male(self):
        t = SALst
        t.init()
        m = "patelh75"
        m1 = "hero1"
        m2 = "me2"
        i = SInfoT("first", "last", GenT.male, 10.5, SeqADT([DeptT.civil, DeptT.chemical]), True)
        i1 = SInfoT("fsdfirst", "lasdfst", GenT.female, 11.0, SeqADT([DeptT.software,
             DeptT.chemical]), False)
        i2 = SInfoT("firstttt", "lastttt", GenT.male, 12.0, SeqADT([DeptT.materials, DeptT.chemical]),
             True)
        t.add(m,i)
        t.add(m1,i1)
        t.add(m2,i2)
        avg = t.average(lambda t: t.gender == GenT.male)
        assert avg == (10.5 + 12) / 2

    def test_average_no_male(self):
        t = SALst
        t.init()
        m = "patelh75"
        m1 = "hero1"
        m2 = "me2"
        i = SInfoT("first", "last", GenT.female, 10.5, SeqADT([DeptT.civil, DeptT.chemical]), True)
        i1 = SInfoT("fsdfirst", "lasdfst", GenT.female, 11.0, SeqADT([DeptT.software,
             DeptT.chemical]), False)
        i2 = SInfoT("firstttt", "lastttt", GenT.female, 12.0, SeqADT([DeptT.materials,
             DeptT.chemical]), True)
        t.add(m,i)
        t.add(m1,i1)
        t.add(m2,i2)
        with pytest.raises(ValueError):
            avg = t.average(lambda t: t.gender == GenT.male)

    def test_average_female(self):
        t = SALst
        t.init()
        m = "patelh75"
        m1 = "hero1"
        m2 = "me2"
        i = SInfoT("first", "last", GenT.male, 10.5, SeqADT([DeptT.civil, DeptT.chemical]), True)
        i1 = SInfoT("fsdfirst", "lasdfst", GenT.female, 11.0, SeqADT([DeptT.software,
             DeptT.chemical]), False)
        i2 = SInfoT("firstttt", "lastttt", GenT.male, 12.0, SeqADT([DeptT.materials, DeptT.chemical]),
             True)
        t.add(m,i)
        t.add(m1,i1)
        t.add(m2,i2)
        avg = t.average(lambda t: t.gender == GenT.female)
        assert avg == 11.0

    def test_average_no_female(self):
        t = SALst
        t.init()
        m = "patelh75"
        m1 = "hero1"
        m2 = "me2"
        i = SInfoT("first", "last", GenT.male, 10.5, SeqADT([DeptT.civil, DeptT.chemical]), True)
        i1 = SInfoT("fsdfirst", "lasdfst", GenT.male, 11.0, SeqADT([DeptT.software, DeptT.chemical]),
             False)
        i2 = SInfoT("firstttt", "lasttt", GenT.male, 12.0, SeqADT([DeptT.materials, DeptT.chemical]),
             True)
        t.add(m,i)
        t.add(m1,i1)
        t.add(m2,i2)
        with pytest.raises(ValueError):
            avg = t.average(lambda t: t.gender == GenT.female)

    def test_allocate_normal(self):
        t = SALst
        n = DCapALst
        n.init()
        n.add(DeptT.software, 2)
        n.add(DeptT.civil, 2)
        n.add(DeptT.materials, 2)
        t.init()
        m = "patelh75"
        m1 = "hero1"
        m2 = "me2"
        i = SInfoT("first", "last", GenT.male, 10.5, SeqADT([DeptT.software, DeptT.chemical]), True)
        i1 = SInfoT("fsdfirst", "lasdfst", GenT.male, 11.0, SeqADT([DeptT.civil, DeptT.chemical]),
             False)
        i2 = SInfoT("firstttt", "lasttt", GenT.male, 12.0, SeqADT([DeptT.materials, DeptT.chemical]),
             True)
        t.add(m,i)
        t.add(m1,i1)
        t.add(m2,i2)
        t.allocate()
        assert AALst.num_alloc(DeptT.software) == 1 and AALst.num_alloc(DeptT.civil) == 1 and
             AALst.num_alloc(DeptT.materials) == 1

    def test_allocate_no_room(self):
        t = SALst
        n = DCapALst
        n.init()
        n.add(DeptT.software, 1)
        n.add(DeptT.civil, 0)
        n.add(DeptT.materials, 0)
        n.add(DeptT.chemical, 0)
        t.init()
        m = "patelh75"
        m1 = "hero1"
        m2 = "me2"
        i = SInfoT("first", "last", GenT.male, 11.5, SeqADT([DeptT.software, DeptT.chemical]), True)
        i1 = SInfoT("fsdfirst", "lasdfst", GenT.male, 11.0, SeqADT([DeptT.software, DeptT.chemical]),
             False)
        i2 = SInfoT("firstttt", "lasttt", GenT.male, 12.0, SeqADT([DeptT.materials, DeptT.chemical]),
             True)
        t.add(m,i)
        t.add(m1,i1)
        t.add(m2,i2)
        with pytest.raises(RuntimeError):
            t.allocate()

    def test_allocate_go_to_second(self):
        t = SALst
        n = DCapALst
        n.init()
        n.add(DeptT.mechanical, 2)
        n.add(DeptT.chemical, 2)
        n.add(DeptT.engphys, 1)
        t.init()
        m = "patelh75"
        m1 = "hero1"
        m2 = "me2"
        i = SInfoT("first", "last", GenT.male, 11.5, SeqADT([DeptT.engphys, DeptT.chemical]), True)
        i1 = SInfoT("fsdfirst", "lasdfst", GenT.male, 11.0, SeqADT([DeptT.mechanical,
             DeptT.chemical]), False)
        i2 = SInfoT("firstttt", "lasttt", GenT.male, 10.0, SeqADT([DeptT.engphys, DeptT.chemical]),
             False)
        t.add(m,i)
        t.add(m1,i1)
        t.add(m2,i2)
        t.allocate()
        assert AALst.num_alloc(DeptT.engphys) == 1 and AALst.num_alloc(DeptT.mechanical) == 1 and
             AALst.num_alloc(DeptT.chemical) == 1
```

L Code for Partner's SeqADT.py

```
## @file SeqADT.py
#  @author Benson Hall
#  @brief ADT for a sequence
#  @date February 9th, 2019


## @brief Class for a sequence data type
class SeqADT:
    ## @brief Initialize the ADT for a sequence
    #  @param x sequence
    def __init__(self, x):
        self.__s = x
        self.__i = 0

    ## @brief Sets iterator to point at the beginning of the sequence
    def start(self):
        self.__i = 0

    ## @brief Increments iterator to point at next element in sequence
    #  @return s[i] element i in the sequence s
    def next(self):
        self.__i = self.__i + 1
        if(self.__i - 1 >= len(self.__s)):
            raise StopIteration
        return self.__s[self.__i - 1]

    ## @brief Checks if the iterator is at the end of the sequence
    #  @return i >= len(s) Is i larger than the length of sequence s?
    def end(self):
        return self.__i >= len(self.__s)
```

M Code for Partner's DCapALst.py

```
## @file DCapALst.py
#  @author Benson Hall
#  @brief Operations on a list of departments and their capacities
#  @date February 9th, 2019


## @brief Department capacity association list
class DCapALst:
    s = []
    ## @brief Initialize the department capacity association list
    @staticmethod
    def init():
        DCapALst.s = []

    ## @brief Add a department to the list; if department exists, raise exception
    #  @param d department name
    #  @param n department capacity
    @staticmethod
    def add(d, n):
        for i in DCapALst.s:
            if d in i:
                raise KeyError
        deptWCap = (d, n)
        DCapALst.s.append(deptWCap)

    ## @brief Remove a department from the list; if department does not exist, raise exception
    #  @param d department name to be removed
    @staticmethod
    def remove(d):
        foundDept = False
        for i in DCapALst.s:
            ## if department name exists in tuple, that is the department to be removed
            if d in i:
                del i
                foundDept = True
                break
        if (foundDept is False):
            raise KeyError

    ## @brief Check if a department exists in the list
    #  @param d department name to search for
    #  @return True if department exists in list, False otherwise
    @staticmethod
    def elm(d):
        for i in DCapALst.s:
            if d in i:
                return True
        return False

    ## @brief Give department capacity
    #  @param d department name to search for
    #  @return i[1] department capacity
    @staticmethod
    def capacity(d):
        for i in DCapALst.s:
            if d in i:
                ## this should return the department capacity
                return i[1]
        ## reaching this point means that the department does not exist
        raise KeyError
```

N Code for Partner's SALst.py

```
## @file SALst.py
#  @author Benson Hall
#  @brief Operatins on a list of students
#  @date February 9th, 2019

from StdntAllocTypes import *
from AALst import *
from DCapALst import *


## @brief Student association list
class SALst:
    s = {}
    ## @brief Initialize the student association list
    @staticmethod
    def init():
        SALst.s = {}

    ## @brief Add a student to the list
    #  @param m: str MacID
    #  @param i: SInfoT Student information
    @staticmethod
    def add(m, i):
        if SALst.s.__contains__(m):
            raise KeyError
        else:
            SALst.s[m] = i

    @staticmethod
    def remove(m):
        if (m in SALst.s) is False:
            raise KeyError
        else:
            del SALst.s[m]

    ## @brief Check if student is in list
    #  @param m: str MacID of student being searched
    #  @return True if student is in list, false otherwise
    @staticmethod
    def elm(m):
        if m in SALst.s:
            return True
        return False

    ## @brief Gives info on a student, if they are in the list
    #  @param m: str MacID of student being searched
    #  @return SALst.s[m]: SInfoT Information about student
    @staticmethod
    def info(m):
        if m not in SALst.s:
            raise KeyError
        else:
            return SALst.s[m]

    ## @brief Sorts student based on GPA
    #  @details The function f filters out students that do not meet the criteria
    #  @param f: function Lambda function
    #  @return sortedMacID: list of macIDs sorted in descending order based on GPA
    @staticmethod
    def sort(f):
        ## filter out those without free choice or a 4.0+ gpa
        unsortLst = []
        for key in SALst.s:
            SInfo = SALst.s[key]
            ## valid: boolean
            valid = f(SInfo)
            if valid is True:
                unsortLst.append([key, SInfo])
        ## This was taken from https://stackoverflow.com/questions/72899/
        # how-do-i-sort-a-list-of-dictionaries-by-a-value-of-the-dictionary
        # Taken from partner file from A1 (partner name unknown)
        sortedLst = sorted(unsortLst, key=lambda k: k[1].gpa, reverse=True)
        ## we only need the macIDs
        sortedMacID = []
        for i in sortedLst:
            ## i[0] = key (macID)
            #  i[1] = value (SInfo) -> what we don't need.
            sortedMacID.append(i[0])
        return sortedMacID

    ## @brief Find average of students
    #  @details The function f filters out students that do not meet the criteria
    #  @param f Lambda function
    #  @return average = sum of all "valid" student GPAs
    #  divided by the number of "valid" students
    @staticmethod
    def average(f):
        sum = 0.0
        count = 0.0
        for key in SALst.s:
            SInfo = SALst.s[key]
            ## valid: boolean
            valid = f(SInfo)
            if valid is True:
                sum += SInfo.gpa
                count += 1
        if (count == 0):
            raise ValueError
        else:
            return sum / count

    ## @brief Allocate students
    #  @details First allocate students based on GPA and free choice, then on GPA >= 4.0
    @staticmethod
    def allocate():
        AALst.init()
        ## assumed that there will never be so many students with free choice
        #  such that they fill an entire department
        F = SALst.sort(lambda t: t.freechoice and t.gpa >= 4.0)
        for m in F:
            ch = SALst.info(m).choices
            AALst.add_stdnt(ch.next(), m)

        S = SALst.sort(lambda t: not (t.freechoice) and t.gpa >= 4.0)
        for m in S:
            ch = SALst.info(m).choices
            alloc = False
            while not(alloc) and not(ch.end()):
                d = ch.next()
                if AALst.num_alloc(d) < DCapALst.capacity(d):
                    AALst.add_stdnt(d, m)
                    alloc = True
            if not alloc:
                raise(RuntimeError)
```