Jessie JR A. Guarino
CPE-301;
Lab activity 1; prelim; dsal32e

Part 1 array

The way Ive written it is to be able to be easily translate the logic in other languages as much as possible, specifically c and cpp. I also did not want to rely on python's built-in functions that are too optimized already as well as python's inherent convenient syntax. The scope of part 1 only needs "an element" to be added and deleted. However, I expanded the scope that the input will cover multiple duplicated elements as I don't want to be complacent. You can skip to the final code and output. However, the following texts documents my intuitions and incremental optimizations while solving the problem stated.

The first problem I encountered was on the deletion part -- there were two lists that I am working with: the referenced list, and the intended value/s to be deleted.

Approach 1: Brute force

My initial approach is a bit slow, which was to iterate through both arrays: finding match for every other value and delete it. However, by only limiting myself with the use of loops and python's pop method which reduces the original array's size, the resulting array has some values that were unintentionally removed.

```
    index = 0
    for arrayValue in array:
        # search for matching values
        # bruteforce
        for deleteValue in cacheDeleteValues:
            if arrayValue == deleteValue:
                array.pop(index)
        index +=1
```

The way I solved it was marking what indices that matches the value in the array and then deleting it after while respecting the current array size.

One thing I made sure before deleting things is to remove duplicates from the marked indices. I got an idea from the following reference [1]. The resulting implementation is shown below:

```
    # marking where in the array that has the value in delete
    markIndex = []
    for aaa in range(len(array)):
        # another search for matching values
        # in delete_values
        # bruteforce
        for bbb in range(len(delete_values)):
            if array[aaa] == delete_values[bbb]:
                markIndex.append(aaa)
```

```
    # sanitize the marked indices
    # by removing duplicates
    tempDuplicates = []
    for iii in range(len(markIndex)):
        # store if next value in array is not equal to current
        if iii+1 != len(markIndex) and markIndex[iii] != markIndex[iii+1]:
            tempDuplicates.append(markIndex[iii])
        # store the last element regardless
        elif iii+1 == len(markIndex):
            tempDuplicates.append(markIndex[iii])
    markIndex.clear()
    markIndex = tempDuplicates
```

Another way is to first sort out the array first to enable us to easily remove duplicates and deleting them as we go through the array. Further reducing the number of operations and memory usage. However, I wouldn't want to touch the code any more than that.

Approach 2: Hashing

Another intuition I always and most frequently resort to is through hashmaps. Firstly, we need to remove duplicates in the list of delete values the same way we did from the first approach. And we can then hash those. A simple key and value may work so I wouldn't have to think about be any offsets.

```
    deleteHash = {}
    for iii in delete_values:
        deleteHash[iii] = iii
```

One of the key differences that I made here is that I'm two days in trying to solve what indices to pop. I have asked chatgpt other ways to solve the problem and a few solutions came: the first one was to create a temporary list to store which indices are not present in the hash. I do not want to increase the memory, so no. The second one was to loop through the array backwards so that we wouldn't mind the offsets we need to subtract to the iterator. The following code presents the initial problematic error indices.

```
    deleteCount = 0
    total = len(array)
    iii = 0
    while iii < total:
        # make sure its safe to access an unknown key, use any string or just keep it as None
        if (deleteHash.get(array[iii]) == array[iii]):
            array.pop(iii - deleteCount)
            deleteCount += 1
            total -=1

        iii +=1
```

And the fix for that:

```
    iii = len(array) - 1
    while(iii >= 0):
        if deleteHash.get(array[iii]) is not None:
            array.pop(iii)
        iii -=1
```

And that concludes the deletion part for the part 1. The rest of the part 1 is as straight forward as it is. The scope, again, I expanded it to not only include an element but a list of elements (affecting the addition and deletion in an array). The code below shows the full implementation of the first part:

Final Code:

```
def addNewElementAtTheEnd(array, add_these):
    """     name            treated as
    param:  array           list
    param:  add_these       list
    return:                 void

    adds elements at the end of the array
    """
    # make sure that addThese is a list
    tempAddThese = []
    for iii in add_these:
        tempAddThese.append(iii)
    add_these = tempAddThese

    # add the elements in add_these to the end of array
    for jjj in add_these:
        array.append(jjj)


def deleteElements(array, delete_values):
    """     name            treated as
    param:  array           list
    param:  delete_values   list
    return:                 void

    deletes elements in the array
    that matches with the value/s of the delete_values
    """



    # makes sure delete_values is a list
    cacheDeleteValues = []
    for iii in delete_values:
        cacheDeleteValues.append(iii)
    delete_values.clear()
    delete_values = cacheDeleteValues


    # marking where in the array that has the value in delete
    markIndex = []
    for aaa in range(len(array)):
        # another search for matching values
        # in delete_values
        # bruteforce
        for bbb in range(len(delete_values)):
```

```
            if array[aaa] == delete_values[bbb]:
                markIndex.append(aaa)


    # sanitize the marked indices
    # by removing duplicates
    tempDuplicates = []
    for iii in range(len(markIndex)):
        # store if next value in array is not equal to current
        if iii+1 != len(markIndex) and markIndex[iii] != markIndex[iii+1]:
            tempDuplicates.append(markIndex[iii])
        # store the last element regardless
        elif iii+1 == len(markIndex):
            tempDuplicates.append(markIndex[iii])
    markIndex.clear()
    markIndex = tempDuplicates

    # deletion of the marked indices
    deleteCount = 0
    for iii in range(len(markIndex)):
        array.pop(markIndex[iii] - deleteCount)
        deleteCount +=1




def deleteElementsHash(array, delete_values):
    """     name            treated as
    param:  array           list
    param:  delete_values   list
    return:                 void

    deletes elements in the array
    that matches with the value/s of the delete_values
    """
    # makes sure delete_values is a list
    cacheDeleteValues = []
    for iii in delete_values:
        cacheDeleteValues.append(iii)
    delete_values.clear()
    delete_values = cacheDeleteValues

    # sort the delete_values from less to bigger value
    delete_values.sort()        # this is may take quite long to write
                                # and it deserves a separate discussion/activity
                                # so lets just resort with this

    # remove duplicates
    # same as with the approach 1
    tempSanitized = []
    iii = 0
    while (iii < len(delete_values)):
        # store if next value in array is not equal to current
        if iii+1 != len(delete_values) and delete_values[iii] != delete_values[iii+1] :
                    tempSanitized.append(delete_values[iii])
        # store the last element regardless
        elif iii+1 == len(delete_values):
            tempSanitized.append(delete_values[iii])
        iii +=1
    delete_values.clear()
    delete_values = tempSanitized


    # create a hash
    # eg. 1:1, 2:2, 25:25
```

```
    deleteHash = {}
    for iii in delete_values:
        deleteHash[iii] = iii

    """
    print("===========\ndeleteHash\nkey\tvalue")
    for key, value in deleteHash.items():
        print(key, "\t", value)
    """



    # deletion
    # loop through the array backwards
    # if the current value does not exist in the hashed deletevalues,
    # pop it
    iii = len(array) - 1
    while(iii >= 0):
        if deleteHash.get(array[iii]) is not None:
            array.pop(iii)
        iii -=1



def printAllElements(array):
    """     name            treated as
    param:  array           list
    return:                 void

    prints all elements in a list
    """
    print("[", end='')
    index = 0
    while index < len(array):
        print(array[index], end='')
        print("]\n"                             \
                if (index+1 == len(array))  \
                else (", "), end='')
        index +=1



orig_array = [1,2,3,4,5,6,7,1,2,3,4,5]
orig_arrayhash = [1,2,3,4,5,6,7,1,2,3,4,5]

add_array = [75,457,78]
add_arrayhash = [346, 657, 123]

delete = [4,1,2,3,1,4, 234, 75]
deletehash = [4,1,2,1,4]

########################################
# operations
addNewElementAtTheEnd(orig_array, add_array)
addNewElementAtTheEnd(orig_arrayhash, add_arrayhash)

deleteElements(orig_array, delete)
deleteElementsHash(orig_arrayhash, deletehash)

printAllElements(orig_array)
printAllElements(orig_arrayhash)
```

```
# the output

[5, 6, 7, 5, 457, 78]
[3, 5, 6, 7, 3, 5, 346, 657, 123]
```

Part 2: Linked list

I don't know how to implement it in python as it does not have pointers. I don't know how to reference the addresses of I searched the internet on how to implement linked list, and multiple sources uses the nodes as objects. The instructions from the board also seems straightforward and it is the same way as it is implemented in many online sources. The following sections shows my attempt with minimal help, the final code and a few modifications.

I approached it first by the interface, and then the implementation after. I initially wanted to initialize the linked list with variable-length arguments.

```
pogi = Linkedlist(10,29,3,4)
```

However, that has proven itself difficult after going through this in multiple sittings. For now, I want it to keep it simple and the following code is yet my best attempt. Thanks to a youtube video [2], I have a brief overview how to work with linked list using OOP. After going through the video two or three times, I tried practicing, imitating and recalling how to implement it in python. When appending new values, the way it is supposed to store what values and indices aren't clear based from the logic of my code.

```
class Node:
    def __init__(self):
        self.value = None
        self.next = None

class Linkedlist:
    def __init__(self):
        self.head = Node()

    def add(self, value):
        self.head.value = value

        index = 0
        while self.head.next is not None:
            index +=1
        self.head.next = index

    def printlist(self):
        pass


pogi = Linkedlist()
pogi.add(10)
pogi.printlist()
```

I gave up, so going again through the video, what I didn't account for was to initialize another node object when adding something to the end of the linked list. What I did next is copy the rest of the correct code hehe. Some methods like set and get, and the size or length of the linked list arent implemented because Im tinatamad. I made some comments here and there to make the code speak some sense when read. Especially at the deleteThis method, I didn't add any branch inside the while loop like what

he did in the video (for just a minor optimization; assuming the compiler and/or interpreter didn't account for that).

Starting the modifications, what's surprising is that the index does not start at 0 in the deleteThis method. That's why there must be -1 for on the condition of the while loop to offset the index and terminate the loop earlier than what it written from the video.

To have the capacity to initialize some values the same way I first approached it, I could just make a wrapper for the Linkedlist class to have that kind of interface so I won't need to change the internal parts of the Linkedlist class.

```
pogi = Linkedlist(10,29,3,4)
```

My initial attempt only makes use of functions. However, it takes more memory.

```
def LinkedlistProper(*values):
    if len(values) == 0:
        return Linkedlist();
    else:
        temp = Linkedlist()
        for iii in values:
            temp.add(iii)
        return temp
```

As much as I don't want to change the init method, what I did was just add this method in the Linkedlist class instead.

```
    def __init__(self, *values):
        self.head = Node()
        self.initializer(*values)

    def initializer(self, *values):
        """
        param:      *values

        pass some values here to initialize the first values
        """
        if len(values) != 0:
            for iii in values:
                self.add(iii)
```

And I can finally now use these code modifications when using the data structure

```
# operations
pogi = Linkedlist(123,345,456)
pogi.printlist()

# add multiple, then print
buffers = [0,1,2,3,4]
for bbb in buffers:
    pogi.add(bbb)
pogi.printlist()
```

```
# deletion
pogi.deleteThis(2)
pogi.deleteThis(0)
pogi.printlist()
```

The final code goes like this

FINAL CODE:

```
   class Node:
    def __init__(self, value=None):
        self.value = value
        self.next = None
        self.previous = None

class Linkedlist:
    def __init__(self, *values):
        self.head = Node()
        self.initializer(*values)
        self.tail = None


    def add(self, value):
        """
        param:  value

        appends only an element to the end
        returns void
        """
        new_node = Node(value)    # initialize the value first with a new node
        current = self.head       # makes the context that
                                  # we are handling the current node
                                  # while also initialize the next node
regardless

        while current.next is not None:
            current = current.next
        while current.previous is not None:
            current = current.previous


        current.previous = current  # point to self
        current.next = new_node     # shift context to right

    def printlist(self):
        """
        prints things the list
        returns void
        """
```

```
        # store it in cache then traverse to the next node
        tempCache = []
        current_node = self.head
        while current_node.next is not None:
            current_node = current_node.next
            tempCache.append(current_node.value)
        print(tempCache)


    def deleteThis(self, index):
        """
        param:  index

        deletes the index specified
        returns void
        """
        current_node = self.head

        # go to the node
        current_index = 0
        last_node = current_node                # set to self node
        while current_index-1 != index:
            last_node = current_node
            current_node = current_node.next
            current_index +=1

        # after arriving here, delete
        last_node.next = current_node.next




    def initializer(self, *values):
        """
        param:      *values

        pass some values here to initialize the first values
        """
        if len(values) != 0:
            for iii in values:
                self.add(iii)


# operations
pogi = Linkedlist(123,345,456)
pogi.printlist()

# add multiple, then print
buffers = [909,123,4,5,6,7,8]
for bbb in buffers:
    pogi.add(bbb)
pogi.printlist()
```

```
# deletion
pogi.deleteThis(8)
pogi.deleteThis(3)
pogi.printlist()
```

OUTPUT:

```
[123, 345, 456]
[123, 345, 456, 909, 123, 4, 5, 6, 7, 8]
[123, 345, 456, 123, 4, 5, 6, 8]
```

A few endnotes: the deletion is also be subjected to some sort of a more compact interface. And the same strategy can also be used which was to wrap things. I realized that you can do the code below, which made it a lot more intuitive in understanding what's happening when we loop.

```
# say we r given a linked list of:   10 → 24 → 234 → 90 → 2300 → 456
current_node = self.head
print("\n\ncurrent: ", current_node.next.next.next.value)
# prints 90
```

All in all, there aren't any optimization problems that I can see for now given the constraints of the instruction. To understand the concept of nodes further, I have tried making a doubly linked list. The following code shows the breakdown of a linked list (hard coded) implementation that I based from the video [3].

```
class Node():
    def __init__(self, value = None):
        self.value = value
        self.head = None
        self.tail = None

pogi1 = Node(1)                         # head : none       address: 000       tail : none;
pogi2 = Node(2)                         # head : none       address: 123       tail: none


# set previous tail to current address
pogi1.tail = pogi2
print("pogi1.tail: ", pogi1.tail)       # head : none       address: 000       tail: 123


# set address to previous
pogi2.head = pogi1
print("pogi2.head", pogi2.head)         # head : 000        address: 123       tail: none
```

References:

[1] https://www.geeksforgeeks.org/cpp-program-to-remove-duplicates-from-sorted-array/

[2] https://www.youtube.com/watch?v=JlMyYuY1aXU&t=3s

[3] https://youtu.be/6sBsF13n5ig?t=203

Appendix

- The source codes are available at https://github.com/Jeo0/dsa-practice-documentation