PA1: Embedded List

Due: Friday, 9/27/2019 before 5:00PM

Total points = 60

1 Objectives

In this assignment you will learn how to:

- Define a custom type.
- Create a custom implementation of a `mutable.Seq`.
- Explore the basics of linked records.
- Embed a linked list in an array.

2 Useful Resources

Review the lecture notes and provided example code for some insight into the Scala syntax. You will also want to read the Scala references provided below:

- Array: `https://www.scala-lang.org/api/current/scala/Array.html`
- Array: `https://www.scala-lang.org/api/current/scala/collection/ArrayOps.html`
- `mutable.Seq`: `https://www.scala-lang.org/api/current/scala/collection/mutable/Seq.html`

Relevant textbook sections:

- Nature of Arrays/Nature of Lists – §12.2,12.3 p406-408
- Mutable Singly Linked List – §12.4 p408-417
- Mutable Doubly Linked List – §12.5 p417-422

3 Instructions

Expect this section to take 10-12 hours of setting up your environment, reading through documentation and base code, and planning, coding, and testing your solution.

Problem 1. (60 points)

Your task is to build a system to store `cse250.objects.TaxEntry` records as they are provided through a sequence of insertions and removals. For this assignment, you will have to implement the specified API using a specific internal storage. The internal storage we will be using is a fixed-size `Array` holding an embedded list of type `A` objects. The "links" of the list keep note of the indices within the array to find of the nodes before and after.

To realize this system, your task is to complete the class definition of `DataEntryStore`, which implements the `mutable.Seq` trait combined with additional methods as follows:

- `insert(entry: A): Unit`
  - Record `entry` into your data entry store.
    - The newest entry must be stored at the end (tail) of the list.
    - Any available slot in your backing storage array may be used, when one is available.
  - If the capacity of the backing storage array is reached, you should overwrite the oldest entry (the head) with `entry`.
    - This would make the second oldest entry the new oldest (new head).
    - This would still make `entry` the newest entry (new tail).
  - Duplicate entries are OK.
- `remove(entry: A): Boolean`
  - If `entry` is not currently present in the list, return `false`, otherwise remove all records containing `entry` from your list and return `true`.
- `countEntry(entry: A): Int`
  - Return the count of the number of containing `entry`.
- `apply(idx: Int): A`
  - Return the `entry` at index `idx` within the list (0 based index).
  - Required by `mutable.Seq`.
- `update(idx: Int, elem: A): Unit`
  - Update the `entry` at index `idx` within the list (0 based index).
  - Required by `mutable.Seq`.

There are additional methods that you should also review:

- `length: Int`
  - Returns the number of elements stored.
  - Required by `mutable.Seq`.

- `iterator: Iterator[A]`
  - Returns an iterator that can be used to traverse the list (i.e., the elements from oldest to newest).
  - Required by `mutable.Seq`.

Note that we are implementing the sequence using a generic type `A`. If you look at the provided tests and the main method, you will see that we can then use this with the `TaxEntry` type.

3.1 List Organization

You must maintain a "list" of entries stored in the `dataArray` of your `DataEntryStore`. The nodes of the list are stored within the array as `EmbeddedListNode` objects, which contain a `value` and indices for `prev` and `next`. You should adhere to the following pointers while maintaining your list:

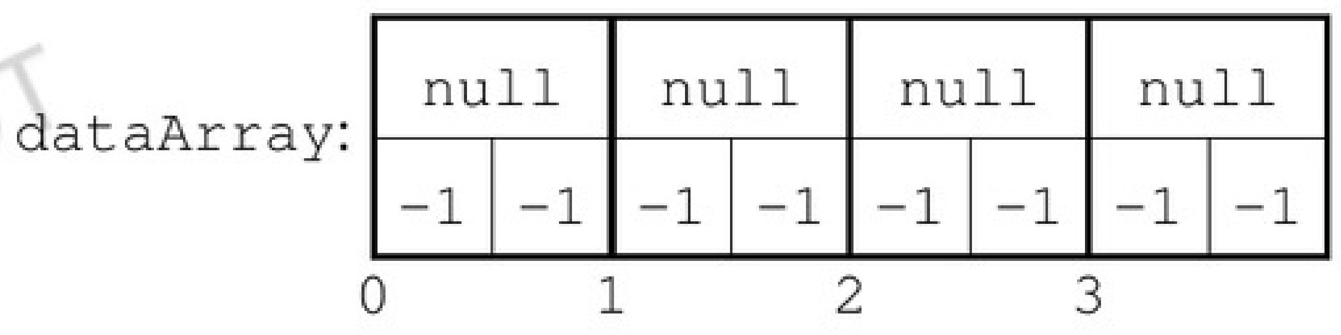
- When an entry is added, be sure to update the indices of the tail and the modified node.
- When an entry is removed, be sure to update the indices of the surrounding entries.
- Take care to check if your are removing the head or the tail and handle them separately.
- Hint: If the storage is full, think of insertion as if you were to remove the head of the list and then use open that slot to store the new tail.

Consider the following code:

```
1  val store = new DataEntryStore[TaxEntry](4); //capacity: 4
2  val e1, e2, e3, e4, e5 = new TaxEntry
3  // initialize the 5 TaxEntry objects.
4  store.insert(e1);
5  store.insert(e2);
6  store.insert(e3);
7  store.insert(e4);
8  store.insert(e3);
9  store.remove(e3); // returns true
10 store.insert(e5);
```

The following is a visualization of the internal state of `store` for the above code:

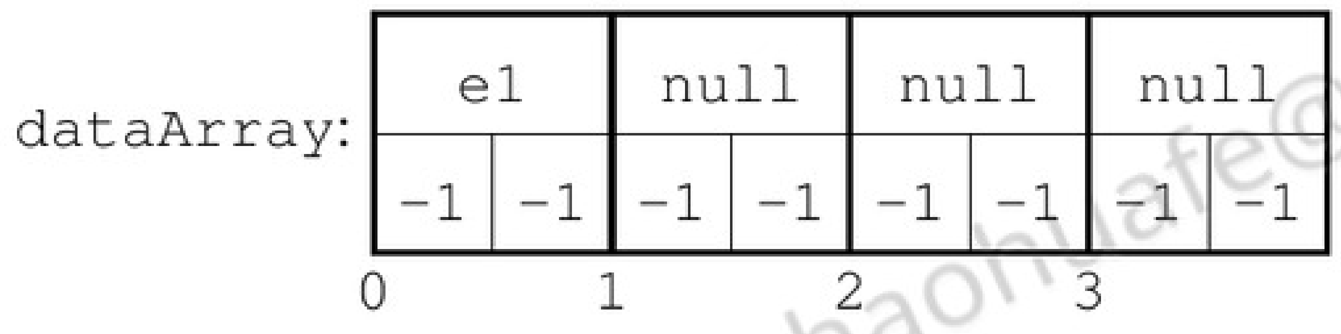
- Following line 1 execution:



```
 capacity: 4
numStored: 0
headIndex: -1
tailIndex: -1
```

  This represents an empty list internally.

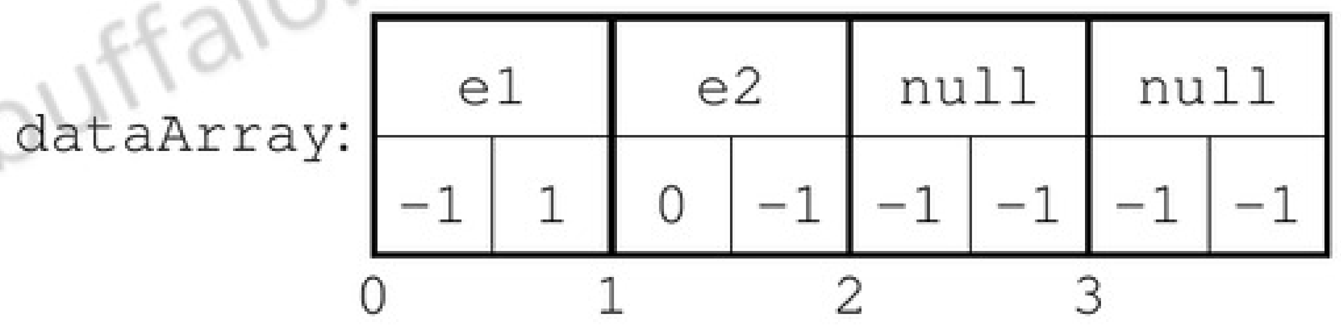
- Following line 4 execution:



```
 capacity: 4
numStored: 1
headIndex: 0
tailIndex: 0
```

  This corresponds to the list: `[ e1 ]`

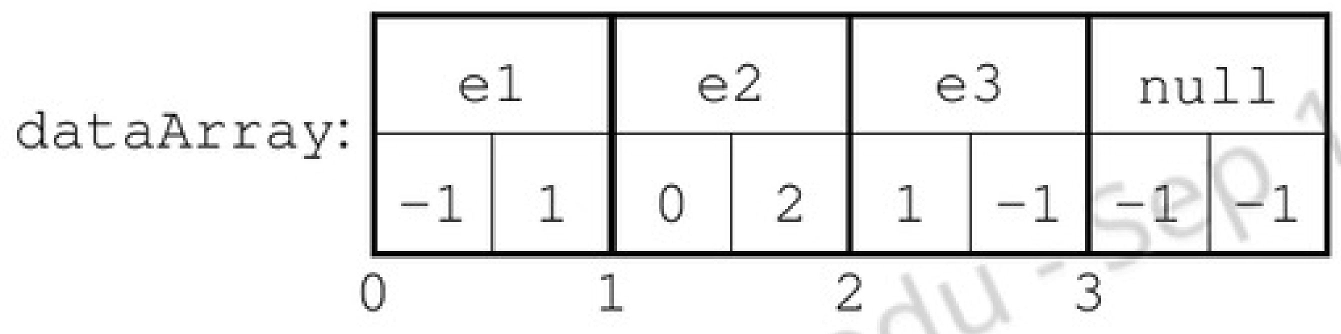
- Following line 5 execution:



```
 capacity: 4
numStored: 2
headIndex: 0
tailIndex: 1
```

This corresponds to the list: `[ e1 ]<->[ e2 ]`

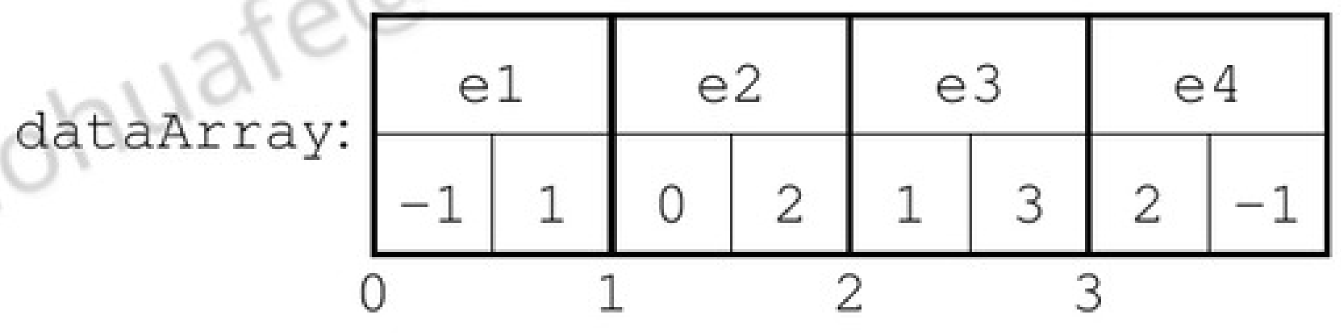
- Following line 6 execution:



```
 capacity: 4
numStored: 3
headIndex: 0
tailIndex: 2
```

This corresponds to the list: `[ e1 ]<->[ e2 ]<->[ e3 ]`

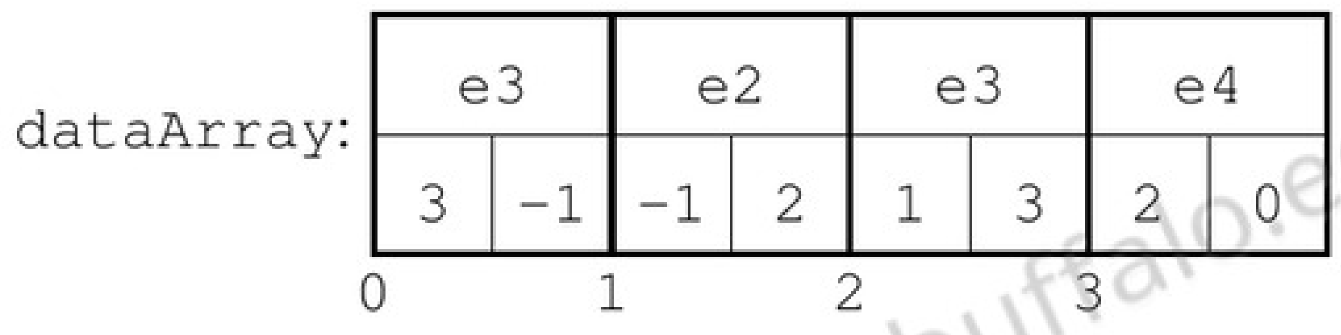
- Following line 7 execution:



```
 capacity: 4
numStored: 4
headIndex: 0
tailIndex: 3
```

This corresponds to the list: `[ e1 ]<->[ e2 ]<->[ e3 ]<->[ e4 ]`

- Following line 8 execution (replaces head with addition):

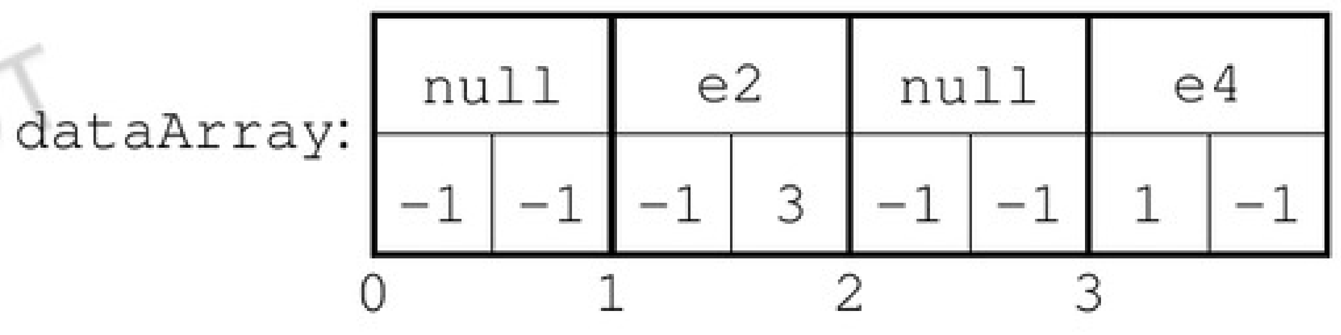


```
 capacity: 4
numStored: 4
headIndex: 1
tailIndex: 0
```

This corresponds to the list: `[ e2 ]<->[ e3 ]<->[ e4 ]<->[ e3 ]`

Note the changes made to the `headIndex` and `tailIndex` as well as the entries in the array.

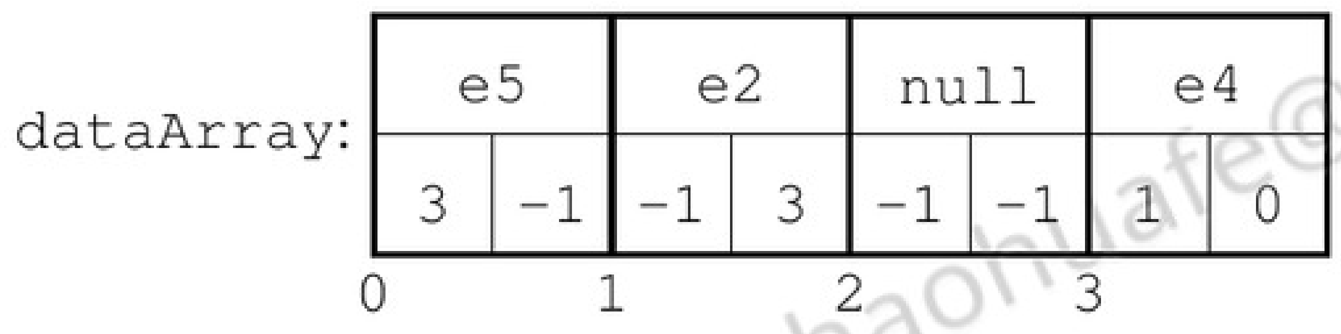
- Following line 9 execution (remove both copies of `e3`):



```
 capacity: 4
numStored: 2
headIndex: 1
tailIndex: 3
```

This corresponds to the list: `[ e2 ]<->[ e4 ]`

- Following line 10 execution (you could add to either free slot):



```
 capacity: 4
numStored: 3
headIndex: 1
tailIndex: 0
```

This corresponds to the list: `[ e3 ]<->[ e4 ]<->[ e5 ]`

Note that the blocks within `dataArray` represent the contents of a `EmbeddedListNode[TaxEntry]` with the `TaxEntry` on top, the `prev` on the bottom left, and the `next` on the bottom right.

3.1.1 List Notes

- The head of the list should always have its `prev` as `-1`.
- The tail of the list should always have its `next` as `-1`.
- The `headIndex` and `tailIndex` should be `-1` when empty.
- When you add an entry, the previous tail's `next` should be assigned the new `tailIndex`.
- When you remove an entry which is the head, the next element's `prev` should be set to `-1` and the `headIndex` updated.
- Every time you remove an entry, restore the node to default (`value`: `null`, `prev`: `-1`, `next`: `-1`).

3.1.2 Suggested Approach

1. Download the PA1 skeleton files (`PA1-handout.zip`), extract the files, and open the folder `PA1` with IntelliJ.
   - Be sure to configure the JDK/Scala settings if prompted.
   - Reimport the Maven project to ensure dependencies are correctly setup.
2. Update the copyright statement with your UBIT and person number in the submission file
   - If you fail to do so, your submission will not be graded.
3. Review the file contents and read over the comments on what is already present.
4. Work on `insert` and try adding values and checking that they are present. You can do so with the iterator or by running the provided unit tests.
   - Check that your lists are updating appropriately.
   - Add more items than your backing storage array can hold to see that you wrap around properly.
   - It is suggested to step the debugger through your method executions to ensure they work as expected.
5. Next, work on the `countEntry` method.
6. After `countEntry`, `remove` should be easier to approach.
7. `apply` and `update` both depend on the other functionality to work correctly, but may be completed at any time.

It is particularly important to follow some semblance of this approach when working on this assignment, as it will be confusing to work out of this order. For instance, it doesn't make sense trying to work on removal of data when nothing is stored isn't possible. Be sure to test as you go. Don't wait until the the end to test.

3.2 Allowed Library Usage

You are not allowed any other imports than those provided.

4 Submission

For each section (PA1 Programming), you will be allowed 5 submissions to each, without penalty. Starting from the 6th submission, you will receive

- a 5 points per submission deduction from your score on the respective assignment.

Note: your score is what you receive on your latest submission. If you receive a score and then resubmit, even if you receive a lower score/0 points that will be your score for the assignment.

Also note: if you submit early/late, the bonus/penalty would be awarded against the entirety of the assignment, not just the single part you submitted early/late. The timestamp for your submission is that of the latest part submitted.

Creating Your Submission

Your submission should be a single file: `DataEntryStore.scala`. No other files should be provided for grading.

5 Late Policy

The policy for late submissions on assignments is as follows:

- More than 5 days before the deadline: +5 points + 100% of what you earn on your best submission.
- Up to 5 days before the deadline: +1 point per day + 100% of what you earn on your best submission.
- On the deadline: 100% of what you earn on your best submission.
- One day late: 25 point deduction (lower total points by 25).
- Two days late: 50 point deduction (lower total points by 50).
- $>$ 2 days late: 0 points.

A day is considered as a 24 hour period counting from the deadline time. From this policy, you will see that all assignments must be submitted within two days past the assigned deadline. The date of submission is that of your latest submission made, even if this is not your highest scoring submission. For assignments with multiple component submissions, only 1 penalty will be assessed based on the file submitted last. If a staggered deadline is given (e.g., two components due one week apart), the earlier deadline will be a hard deadline and no late submissions will be accepted for the first component.

You will have the ability to use three grace days throughout the semester, and at most two per assignment (since assignments are not be accepted beyond two days late). Using a grace day will negate the 25 point penalty for one day of late submission, but will not allow you to submit more than two days late. Please plan accordingly. You will not be able to recover a grace day if you decide to work late and your score was not sufficiently higher. **Grace days are automatically applied** to the first instances of late submissions, **and are non-refundable**. For example, if an assignment is due on a Friday and you make a submission on Saturday, you will automatically use a grace day, regardless of whether you perform better or not. **Be sure to test your code before submitting**, especially with late submissions, **in order to avoid wasting grace days**.

Keep track of the time if you are working up until the deadline. Submissions become late after the set deadline. Keep in mind that **submissions will close 48 hours after the original deadline** and you will no longer be able to submit your code after that time.

6 AI Policy Overview

As a gentle reminder, please re-read the academic integrity policy of the course. I will continue to remind you throughout the semester and hope to avoid any incidents.

6.1 What constitutes a violation of academic integrity?

These bullets should be obvious things not to do (but commonly occur):

- Turning in your friend's code/write-up (obvious).
- Turning in solutions you found on Google with all the variable names changed (should be obvious). This is a copyright violation, in addition to an AI violation.
- Turning in solutions you found on Google with all the variable names changed and 2 lines added (should be obvious). This is also a copyright violation.
- Paying someone to do your work. You may as well not submit the work since you will fail the exams and the course.
- Posting to forums asking someone to solve the problem.

  Note: Aggregating every [stack overflow answer|result from google|other source] because you "understand it" will likely result in full credit on assignments (if you aren't caught) and then failure on every exam. Exams don't test if you know how to use Google, but rather test your understanding (i.e., can you understand the problems to arrive at a solution on your own). Also, other students are likely doing the same thing and then you will be wondering why another person that you don't know has your solution.

You should know that seeking solutions to the assignment does not fall under solving the problem yourself. Things that may not be as obvious:

- Working with a tutor who solves the assignment with you. If you have a tutor, please contact me so that I may discuss with them what help is allowed.
- Sending your code to a friend to help them. **If another student uses/submits your code, you are also liable and will receive the same punishment**.
- Joining a chatroom for the course where someone posts their code once they finish, with the honor code that everyone needs to change it in order to use it.
- Reading your friend's code the night before it is due because you just need one more line to get everything working. It will most likely influence you directly or subconsciously to solve the problem identically, and your friend will also end up in trouble.

The assignments should be solved individually with only assistance from course staff and allowed resources. You may discuss and help one another with technical issues, such as how to get your compiler running, etc. It is not acceptable that you both worked together and have nearly identical code. If that is going to be a problem for you, don't solve the problems in that close of proximity.

6.2 What collaboration is allowed?

There is a gray area when it comes to discussing the problems with your peers and **I do encourage you to work with one another to solve problems**. That is the best way to learn and overcome obstacles. At the same time you need to be sure you do not overstep and not plagiarize. Talking out how you eventually reached the solution from a high level is okay:

> I used a stack to store the data and then looked for the value to return.

but explaining every step in detail/pseudocode is not okay:

> I copied the file tutorial into my code at the start of the function, then created a stack and pushed all of the data onto the stack, and finished by popping the elements until the value is found and use a return statement.

The first example is OK but the second is basically a summary of your code and is not acceptable, and remember that you shouldn't be showing any code at all for how to do any of it. Regardless of where you are working, you must always follow this rule: **Never come away from discussions with your peers with any written work, either typed or photographed, and especially do not share or allow viewing of your written code.**

If you have concerns that you may have overstepped or worked closely with someone, please address me prior to deadlines for the assignment. We will address options at that point.

6.3 What resources are allowed?

With all of this said, please feel free to use any [files|examples|tutorials] that we provide directly in your code (with proper attribution). Feel free to directly use anything from lecture or recitations. You will never be penalized for doing so, but must always provide attribution/citation for where you retrieved code from. Just remember, if you are citing an algorithm that is not provided by us, then you are probably overstepping.

More explicitly, you may use any of the following resources (with proper citation/attribution in your code):

- Any example files posted on the course webpage (from lecture or recitation).
- Any code that the instructor provides.
- Any code that the TAs provide.
- Any code from the tour of Scala (https://docs.scala-lang.org/tour/tour-of-scala.html)
- Any code from Scala Collections (https://docs.scala-lang.org/overviews/collections-2.13/introduction.html)
- Any code from Scala API (https://www.scala-lang.org/api/2.13.0/)
- Additional references may be provided as the semester progresses, but only those provided publicly by course staff are allowed to be utilized. These will be listed on Piazza under Resources.

Omitting citation/attribution will result in an AI violation (and lawsuits later in life at your job). This is true even if you are using resources provided.

Lastly, **if you think you are going to violate/have violated this policy, please come talk to me ASAP so we can figure out how to get you on track to succeed in the course**. This policy on assignments is here so that you learn the material and how to think yourself. There is no benefit to submitting solutions (which likely exist in some form).

7 Collaboration Policy

The policy for collaboration on assignments is as follows:

- All work for this course must be original individual work.
- You must follow the limits on collaboration as defined in the AI policy (i.e., no work written while together/shared code/etc.).
- You must identify any collaborators (first and last name) on every assignment. This can be in a comment at the top of your code submissions or on the first page at the top of your written work beside your name.

All references must be cited using a comment containing a direct link to the resource as well as a brief description of what was used. For example, if you reference the textbook, a page number and description is sufficient. If you copy example code from the Scala language API, then include the link to the class page within the API as well as where the example code resides.