Chapter 1,2,3

1.1 Problem: Is Unique: Implement an algorithm to determine if a string has all unique characters. What if you cannot use additional data structures?

I: This problem is asking us to check if a string's characters or contents are all unique meaning there are no duplicates or repeating characters. Therefore, we basically need to check for duplicates and if there are duplicates we return false if there isn't we return true because all the strings characters are unique. We can think of many different data structures to solve this problem, but the problem clearly states we cannot use additional data structures.

D: Since we cannot use additional data structures we must use a boolean array allowing us to check the characters of the string and mark the index where that character is in the string. Otherwise, we would return false if we see that character again. If we could use an additional data structure, we could consider using a hash set or hash map to solve this.

E,A: Step 1: Works some small instances by hand:

Say we have a string "aabc" this string would obviously fail because a is duplicates so not all characters are unique. If we had a string "abc" this would return true because all the characters are unique. Or if the strings characters are duplicated but separated is another possible input that should also return false, for example: "abca" or" abac" or "abcb" etc.

Step 2: write down what you did:

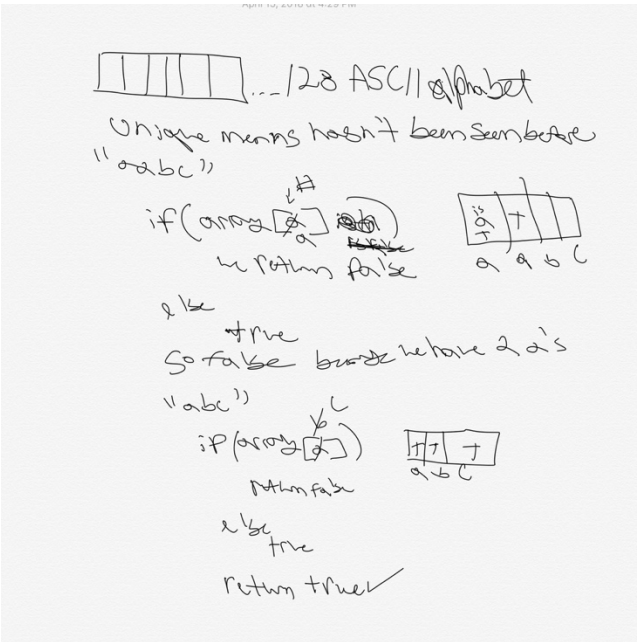
I have solved similar problems to this one in labs. Since this is the case I thought about those problems and how I solved them. I then thought about how this problem was different from those I have solved and how it was similar. The first thing that came to mind was checking using a boolean array and marking true if we have visited that character. Then I thought about what instances that would cause the algorithm to return true or false which I explained in step 1. These are great examples to use in the testing phase.

Step 3: Find patterns

Since I had worked on problems similar to these I was thinking about them and how I could change them to fit this problem. So, the patterns in how those would apply to this one. We definitely need the boolean array, and we need to convert the string into a character assuming we can use ASCII or Unicode to do so. Assuming this we can set our boolean array to be 128 characters because that is how many characters in the ASCII alphabet. Of course, we need a for loop to traverse the string and if that boolean array at that current index is true then we return false else we return true.

Step 4: Check by hand

Chapter 1,2,3



Step 5: translate it to code

```
public static boolean isUnique(String str) {
    boolean[] checkString = new boolean[128];
    for (int i = 0; i < str.length ( ); i++) {
        int index = str.charAt (i);
        if (checkString[index]) {
            return false;
        } else {
            checkString[index] = true;
        }
    }
    return true;
}
```

Step 6: run test cases:

This is my main where I tested the string examples I listed above. I should have gotten false true and false. I did indeed get that output.

```
public static void main(String[] args) {
        String a="aabc";
        String b="abc";
        String c="abca";

       System.out.println( isUnique (a));
       System.out.println(isUnique (b));
        System.out.println (isUnique (c));
}
```

Output:

Chapter 1,2,3

false
true
false

Step 7: Debug failed test case:
I tried the empty string. I received true which is correct.
String d=" ";

L: It was interesting to think about how this problem has come up often during my classes and in different variable types. I wouldn't have known about how many characters were in the ASCII alphabet if we haven't talked about it in class so that saved me the trouble of looking it up.

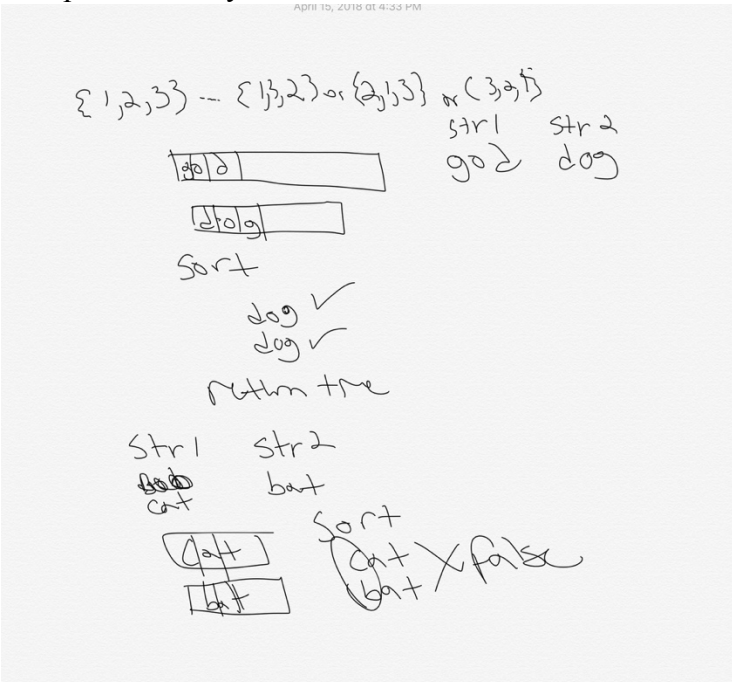
Chapter 1: Problem 1.2- Check Permutation: Given two strings, write a method to decide if one is a permutation of the other.

I: Permutations are basically a way of arranging different variations of a set of things where order matters. So, we are essentially checking to see if two stings that we pass are an arrangement of the other, in the sense that it follows the rules of the set. So, say you have a set {1,2,3} we could have {1,3,2} or {2,1,3} or {3,2,1} are a few examples of permutations.
D: Right away I think about checking if the strings are the same length, and if they are we put the strings into an array or array list and sorting it and then checking if the words match up or do something similar to problem one creating an array of size 128 and see if we have seen it before and if we end with the variable we use to check the strings is 0 for both we know we have seen it before and they are permutations.
E,A: Step 1: Works some small instances by hand:
For example, we pass two strings the first one being "god "and the other "dog" that would be a permutation of the other. Another example even simpler would be "abc" and "bca" would be another example of a permutation. Ones that would not be a permutation would be like "cat" and "bat" or "abc" "ab" or "abc" and "dca".
Step 2: write down what you did:
So we need to make a character array and sort the array and return that sorted array in a separate method. Then the main method we compare them and check if they are equal or be compare them and if they are we return true if not we return false.

Step 3: Find patterns:
This problem reminds me of the anagram problem we worked on in class. I think that's why I could think about how to work it out so quickly since we worked on a similar problem before. If we could use any data structure it would be cool to use a hash map to solve this. Since it's a chapter on arrays and strings we have to solve it using an array. We definitely need to use an if statement to check the lengths and if they are not the correct length we return false and then we can put them into an arrays and sort them and see if the equal each other. To make it more generic I think it's better if we split it into two methods so that we just call the method and it can check each string without doubling the lines of code in one method.

Chapter 1,2,3

Step 4: Check by hand



Step 5: translate it to code

```
public static String sortForPermutation(String str) {
      char[] permCharArray = str.toCharArray ( );
      Arrays.sort (permCharArray);
      String sortedStr = new String (permCharArray);
      return sortedStr;
   }

   public static boolean permutationChecker(String str1, String str2) {
      if (str1.length ( ) != str2.length ( )) {
         return false;
      }
      return sortForPermutation (str1).equals (sortForPermutation (str2));
   }
```

Step 6: run test cases:

I first ran str1 and str2 which should return true which it does. Then I tried a and b which should return false and it does. Then I tried c and d which should return true and it does so we are good.

```
      String str1="dog";
      String str2="god";
      String a="ab";
      String b="abc";
      String c="abc";
      String d="bca";
      System.out.println (permutationChecker (str1,str2) );
```

```
System.out.println (permutationChecker (a,b) );
System.out.println (permutationChecker (c,d) );
```

Output:
true
false
true

Step 7: Debug failed test case:
Maybe we need to also consider upper case letters and strings with spaces as well. We should ask about that and once we know what to assume or not we can modify the code as necessary.
L: This one was very easy because of the fact that we have worked on similar problems to this one. It is very easy to come up with the code for it. I think the factors in the debug step should definitely be taken into account because that can definitely affect the code and its outcome. We could also try the other solution I mentioned above and that could have a better running time.

Chapter 1: Problem 1.3- URLify: Write a method to replace all spaces in a string with %20. You may assume that the string has sufficient space at the end to hold the additional characters, and that you are given the "true" length of the string. (Note: if implementing in Java, please use a character array so that you can perform this operation in place.)
EXAMPLE:
Input: "Mr John Smith ", 13
Output: "Mr%20John%20Smith"

I: Basically, we need to go check the string and look for spaces for example: " " and once we find the space we will replace that space with %20. Doing so will leave a string that may previously had spaces and now those spaces are occupied by %20.
D: So, since we have to use a character array we for sure have to traverse it, looking for a space and if we do find it then we need to add each new character the %, the 2, and the 0. For sure we have to keep an eye on the size of the array because if we exceed it we will need to create a new array and copy over the information from the old array into the new one leaving space of the new characters. Maybe we could traverse it and keep a count of how many spaces there are, so we can use that variable to help us. For sure we will need an index variable to allow us to manipulate the index in the character array.
E,A: Step 1: Works some small instances by hand:
Examples of this working: input "a bc d",10 and should output "a%20bc%20d. One input that would not be modified would be like "abcd",4 and would output "abcd". Or if we don't put a correct length we would get an array out of bounds.

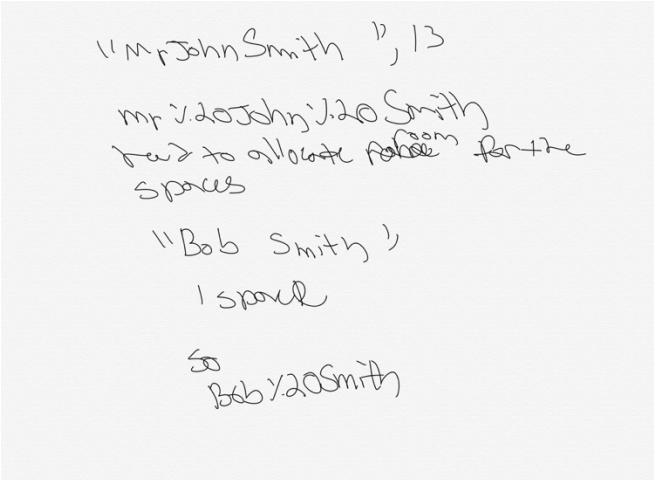
Step 2: write down what you did:
I had to make sure to count all the spaces in the example and think about how the size will affect the array. How would you accommodate the fact that the array may need to be enlarged? Should we initially create a larger array and use that, but we can't always anticipate we know how to accommodate that size, or that the user puts in the right length. I decided to use simple strings like my examples in step 1 and I could easily see what needs to happen and what

the character string needs to look like when we pass it into the method, as well as, the length of the array.

Step 3: Find patterns

I found that for sure we have to leave three spaces for the three characters that are going to be added so we need a variable to keep track of the spaces so that when we create a new array we know where to create those three empty spaces that will house the %20. With this in mind I wonder how traversing the array will affect getting the necessary responses. How would traversing backward be more effective due to the fact that we have to add the %20.

Step 4: Check by hand



Step 5: translate it to code

```
public static void replaceSpaces(char[] str, int trueLength) {
      int spaceCounter = 0;
            int index = 0;
            int i = 0;
      for (i = 0; i < trueLength; i++) {
         if (str[i] == ' ') {
            spaceCounter++;
         }
      }
        If(spaceCounter==0) System.out.println(“There are no spaces in this string”);//we could return the string here
         //a if statement to deal with if the array is too small make a new one and copy it over
     Index=something to look for the spaces this is where the space counter will be used
     for (i = trueLength - 1; i >= 0; i--) {
          if(str[i]== “ “){
            add the ‘0’ and the ‘2’ and the ‘%’
          }
      }
}
```

Chapter 1,2,3

Step 6: run test cases:

I ran into some issues, so I didn't get to completely solve this problem.

Step 7: Debug failed test case:

I feel that we need to also check if the array is too long maybe we should decrease the size of the array as an else or we should make an array to see if we exceed the size to make a bigger array in case the array is too small.

L: I don't like these types of problems because arrays are kind of a pain when it comes to adding stuff and not having enough space, so I get caught between making a bigger array and risk it being too big. So, I would have rather used array lists since we don't have to worry at the size and we can go straight to the spaces and add what we need to. Or even a hash map would be much more easy and effective to execute especially when it comes to running times as well.

Chapter 1: Problem 1.5-One Away: There are three types of edits that can be performed on strings: insert a character, remove a character, or replace a character. Given two strings, write a function to check if they are one edit (or zero edits) away.
EXAMPLE:
Pale, ple ->true
Pales, pale -> true
Pale, bale-> true
Pale, bae -> false

I: We have two strings and we want to check if the first string takes one edit to get to the second string. We can "edit" these strings by either inserting, removing, or replacing a character. If it is more than one edits it will return false and if it is one edit it is true.
D: We need three separate methods one check if we insert a character one to check if we remove a character and one to check if we edited the string once. We can easily create the replace method and the insert method will be a little harder to create and our main check method will take the other methods and checks if it is one away.
E,A: Step 1: Works some small instances by hand:

The examples from the problem are great small instances to work with. Other examples we could try could be gray and one edit could be gra or grey, but more than one edit would be gred or gr. Even grey and you have g or nothing because everything was removed and of course that would return false.

Step 2: write down what you did:

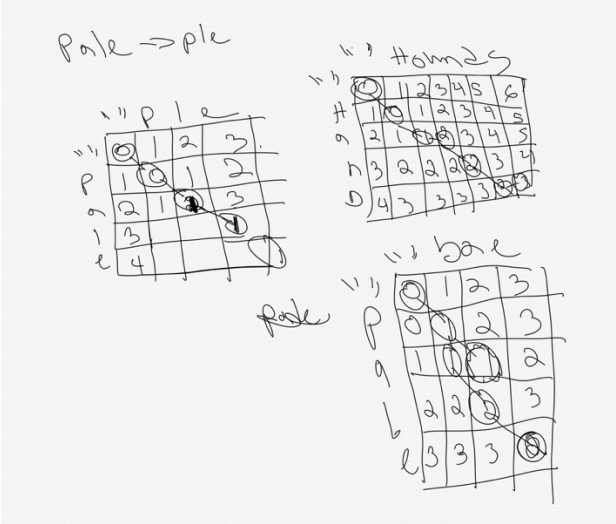
The replace method we just pass two strings and we set a boolean variable to false and we go through the string character by character and if they are not the same then we mark that there is a different and return false else we set our boolean variable to true and return true. The insert method we have to think back to dynamic programming and how we get the minimum edit distance so with that in mind we can think about checking if a character was inserted, for sure we will pass two strings and we will need two variables to keep track of the strings. The third array we could check the lengths then check if we inserted or replaced or removed.

Step 3: Find patterns

Chapter 1,2,3

The replace method would be very similar to the has been seen before type method therefore, it is very easy to check these strings because we have done similar problems to this one. We think about minimum edit distance chart to think about how we would check for removal and insertion and bringing them together in the final third array.

Step 4: Check by hand



Step 5: translate it to code

```
public static boolean checkReplace(String str1,String str2){
    boolean fDiff=false;
    for(int i=0;i<str1.length ();i++){
       if(str1.charAt (i)!=str2.charAt (i)){
          if(fDiff){
             return false;
          }
          fDiff=true;
       }
    }
    return true;
  }
Public static boolean checkInsert(String str1,String str2){
   int p1=0;
   int p2=0;
   while(p2<str2.length () && p1<str1.length ()){
      if(str1.charAt (p1)!=str2.charAt (p2)){
          if(p1!=p2){
            return false;
          }
          p2++;
       }else{
          p1++;
          p2++;
```

```
        }
    }
Return true;
}

Public static boolean checkOneEdit(String str1,String str2){
if(str1.length ()==str2.length ()){
        return checkReplace (str1, str2);
    }else if(str1.length ()+1== str2.length ( )){//we add +1 because we inserted a character
        return checkInsert (str1, str2);
    }else if(str1.length ()-1==str2.length ()){//we subtract -1 because we removed a character
        return checkInsert (str2,str1);
    }
    return false;
  }
```

Step 6: run test cases:
We try the same examples given str1 and str2 we should get true and false for a and b. We did get that output.

```
String str1="pale";
  String str2="ple";
  String a="pale";
  String b="bae";
    System.out.println (oneEditAway (str1,str2) );
    System.out.println (oneEditAway (a,b) );
```

Output:
true
false

Step 7: Debug failed test case:
I had to play around with the increments and think about when to increment that took some time to figure out. Other than that, I didn't have to much trouble because drawing out the minimum edit distance really helps a lot to write the code.
L: This solution wasn't too bad because I practiced doing the tracing so much that I could see what needed to be done. It took a bit of tweaking, but I was happy I figured it out. I really like this question because of the ease of visibility to translate from drawing to code.

Chapter 1: Problem 1.6-String Compression: Implement a method to perform basic string compression using the counts of repeated characters. For example, the string aabcccccaa would become a2b1c5a3. If the "compressed" string would not become smaller than the original string, your method should return the original string. You can assume the string has only uppercase and lowercase letters (a-z).

I: We are basically counting the characters and if there are duplicate letters together we basically condense them to just one instance of the letter and the number of how many there are then the

next letter and how many there are etc. If there is one instance of the letter we still add that 1 to the character so it doesn't necessarily have to be duplicated.
D: We go through the string and count all the letters that are next to each other and once we find a new character we do the same thing then we add in those counts and compress the string.
E,A: Step 1: Works some small instances by hand:

We could use "aaabc" and it would be "a3b1c1" would be the compressed version of the original string. Or even smaller "ab" would be "a1b1"/

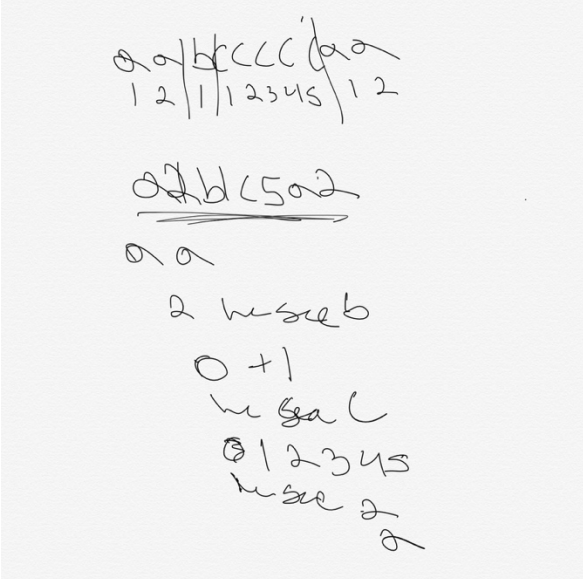
Step 2: write down what you did:

Well we need to loop through the string and count it. Once we come across the new character then we take out the duplicated and add the count of how many they are then reset the counter and continue till there is no more string to traverse. Then we return the final string. That might not be efficient enough, again a hash map would be better to use but again we need to use strings or arrays.

Step 3: Find patterns

The only pattern I can see is similar to the third problem in the sense that we are taking things out and adding them. For sure a for loop to traverse and a counter to keep track of the number of letters and then an if statement to along with what we need to do to compress and add in the count and then reset them.

Step 4: Check by hand



Step 5: translate it to code

```
Public static String compressString(String str){
    String compress=" ";
    int count=0;
    for(int i=0;i<str.length();i++){
        count++;
        if(the current character is not equal to the one next to it ){
            then we delete the extra characters and add in the count
            count=0;
```

```
            }
        }
        Return compress;
    }
```

Step 6: run test cases:
In this way once we add in the necessary code we should get the correct result that we need. Like the example given.

Step 7: Debug failed test case:
We will definitely need to debug and see the edge cases and what might cause any issues.
L: This problem is easy to understand but tough to just straight out code I would need more time to work on it to figure it out. I really like this problem though it's really interesting.

Chapter 2: Problem 2.1 Remove Dups: Write code to remove duplicates from an unsorted linked list. FOLLOW UP How would you solve this problem if a temporary buffer is not allowed?

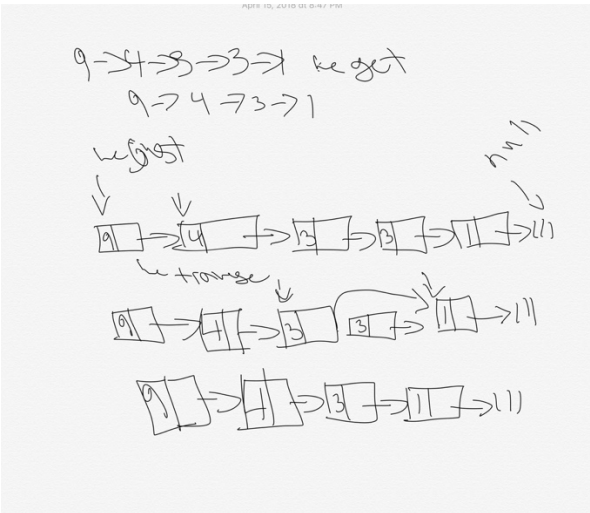
I: We are simply removing the duplicated from a unsorted linked list.
D: We can use two pointers and we iterate through the linked list and check all the nodes for duplicates.
E,A: Step 1: Works some small instances by hand:
9-> 4->3->3->1 or 1->1->3->2 are examples of duplicates of unsorted arrays we can edit to make sure the duplicates don't get added.

Step 2: write down what you did:
First, we need to have a pointer at the head and we traverse the linked list and we have another pointer for the current node and we have another while loop to check if the next node is not null if they are equal we move the pointer to point to next.next node so we lose the duplicate node, we then set the current.next to current and we keep going.

Step 3: Find patterns
I have seen this problem many times before so this one I was able to solve quickly because I have had a lot of practice with this problem.

Step 4: Check by hand
Will be added in a compressed file

Chapter 1,2,3



Step 5: translate it to code

```
public static void deleteDup(iNode head) {
        iNode curr = head;
        while (curr != null) {
            iNode pointer2 = curr;
            while (pointer2.next != null) {
                if (pointer2.next.data == curr.data) {
                    pointer2.next = pointer2.next.next;
                } else {
                    pointer2 = pointer2.next;
                }
            }
            curr = curr.next;
        }
    }
```

Step 6: run test cases:
Added 1-> 9->5 ->5
Output: 1->9->5
So, I went ahead and created the linked list and it did what it was supposed to do get rid of 5.
Step 7: Debug failed test case:
Getting the linked list class set up and get it working took some time but once it got done it worked.
L: I should have just used a previously used class which I will probably end up using. Like I said the actual method was nothing new, so it wasn't hard to work out at all. I am sure we could use a hash function to make this more efficient.

Chapter 2: Problem 2.2 Return Kth to last: Implement an algorithm to find the kth to last element of a singly linked list.

I: We can print the elements of the linked list from k that we enter to the end of the linked list.

Chapter 1,2,3

D: We pass the head of the iNode if necessary if we have access to it then we only need to pass the integer k. We need to get the length, so we can use that to our advantage. Of course, we don't mess with the head, so we always create a temp pointer if need be to traverse the array. Then while that temp node doesn't equal null we traverse and count the length. Then we check if k is more than the length because we want to make sure we don't get a null pointer exception. If it is, we just return and then we traverse the list and print from k to the end.

E,A: Step 1: Works some small instances by hand:

You have a linked list from 1->2->3->4 and k is 3 so we will only print 4. If we have an unsorted array 4->9->1->3 and say k is 1 so we would print 9,1, and 3. Or if we have k at 0 it will print the whole list since 4 is the head. Or if it was 11 we would just return.

Step 2: write down what you did:

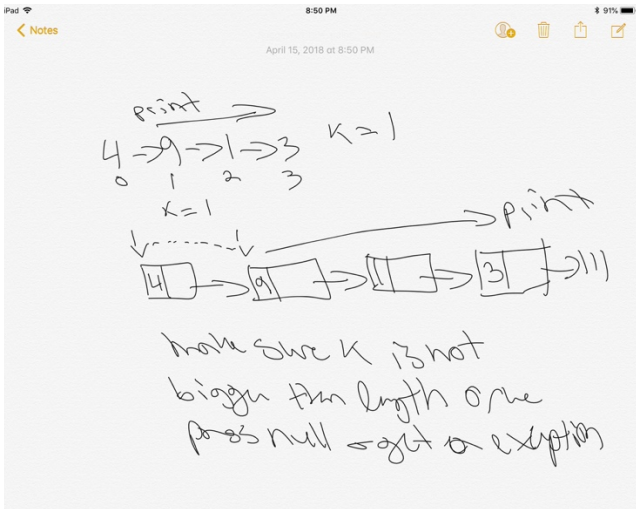
I basically did the same thing I wrote in D. Count the length of the list check to make sure k isn't greater than the list and then we just traverse the list and print from k to the end.

Step 3: Find patterns

For sure we need a variable node pointer, so we don't lose the head pointer and we need a while loop to make sure we don't go over null. We also need to traverse the linked list and print it like any other method.

Step 4: Check by hand

On the compressed file attached.



Step 5: translate it to code

```
public void printNthFromLast(iNode head,int k) {
        int getLength = 0;
        iNode temp = head;

        while (temp != null) {
           temp = temp.next;
```

```
            getLength++;
        }
        if (getLength < k) return;
        temp = head;
        for (int i = 1; i < getLength - k + 1; i++)//get the node from the beginning
            temp = temp.next;

        System.out.println (temp.data);
    }
}
```

Step 6: run test cases:
Input: 5->6->10->1 and k is 2 we expect our output would be 10 and 1.
Output 10->1 was the output so we were good.
Input was still the same but this time we changed k to 7 and we just return and print nothing since k is bigger than the length of the linked list. We expected that, so we are good.

Step 7: Debug failed test case:
I had to manipulate the for loop till I got the correct values of the length -k+1 because I kept getting errors. Once I fixed that everything worked. I am sure hash table would be more efficient.
L: This is also another method I have seen a lot and worked on a lot in slight variations, so I had a good idea on where to start and how to go about doing this.

Chapter 2: Problem 2.3 Delete Middle Node: Implement an algorithm to delete a node in the middle (i.e., any node but the first and last node, not necessarily the exact middle) of a singly linked list, given only access to that node.
EXAMPLE
Input: the node c from the linked list a->b->c->d->e->f
Result: nothing is returned but the new linked list looks like a->b->d->e->f

I: All we have to do is find the middle node in the linked list and we delete it by pointing the previous next to next.next and it will disappear.
D: We just check to make sure that the list isn't empty and the node we are deleting isn't null and if it is we return false because we can't delete it otherwise we delete it and return true. We do so because it doesn't ask us to print so we are just simply doing that operation.
E,A: Step 1: Works some small instances by hand:
Well for sure if we had a null list or a list like 1-> then we return false because we have nothing to delete. Now if we had 10->11->6->12->4 then we remove the middle node 6 we would have 10->11->12->4. Or a->b->c->d->e so we would have a->b->d->e. Or for odd number lists the example given to us is a great example.

Step 2: write down what you did:
We basically do what I stated in D. We check to make sure we don't have a null list or the next to the middle we are taking out isn't null and then if it isn't we use delete the node and return true because we were able to delete it.

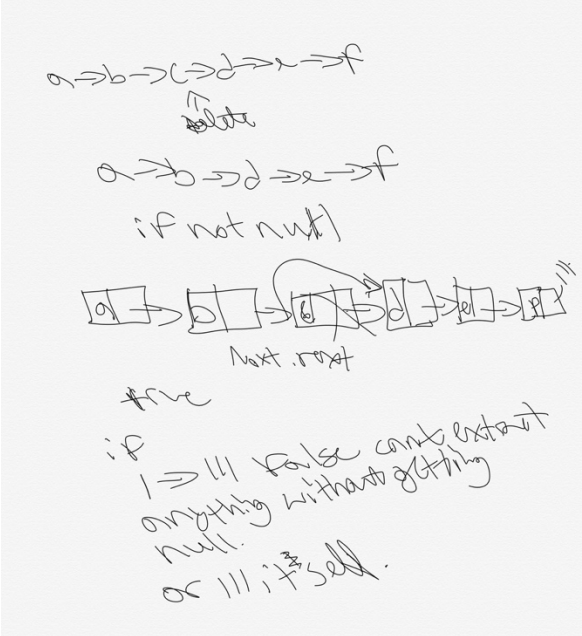
Chapter 1,2,3

Step 3: Find patterns

We always check to make sure the list is not null. Other than that, I can't really find other patterns because it's so straight forward what we need to do.

Step 4: Check by hand

Attached on zip file



Step 5: translate it to code

```
public boolean deleteMidNode(iNode s){//2.3
        if(s==null||s.next==null){
            return false;
        }
        iNode next=s.next;
        s.data=next.data;
        s.next=next.next;
        return true;
    }
}
```

Step 6: run test cases:

Input: 9->10->4->8 we expect 10 to get deleted so we expect an output of true.
Output: true
Input: 1-> expect return false;
Output: false
Input: a->b->c->d->e we expect c to get deleted so we get true.
Output: true

Chapter 1,2,3

All tests where successful.
Step 7: Debug failed test case:
I just tested with the edge cases to make sure it was working because I really couldn't think of much else since this is such a simple problem.
L: I have also seen this problem before in different forms deleting the first node or the last one or second to the last etc. Therefore, I didn't have much problem to create this method.

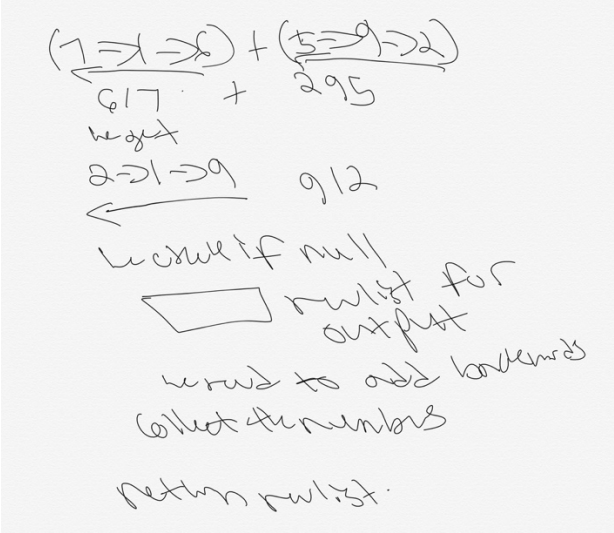
Chapter 2: Problem 2.5 Sum lists: You have two numbers represented by a linked list, where each node contains a single digit. The digits are stored in reverse order, such that the 1's digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list.
EXAMPLE
Input: (7->1->6) + (5->9->2). That is 617+295.
Output: 2->1->9. That is 912.
FOLLOW UP
Suppose that digits are stored in forward order. Repeat the above problem.
Input:(6->1->7) + (2->9->5). That is, 617 + 295.
Output: 9->1->2. That is, 912.

I: We then basically take the nodes and we collect them all like one large number in reverse and we add it with another list and we output the answer.
D: So, this is literally doing addition as if we were doing it on a piece of paper, so we need to consider there will be a carry-over number. We also need to make sure we read it correctly. Of course, we need to make sure they are not null.
E,A: Step 1: Works some small instances by hand:
Say we have (4->1) + (1->3) that would be 41+13 and the output would be 5->4 which is 54. Or (1->1) + (2->2) that would be 11+22 and we would get 3->3 which would be 33.
Step 2: write down what you did:
So, we need two references to nodes and a integer to hold the carry over number and we need to check if they are null or equal to 0. And then we need to create a new node to add to our list for the result and then we need to read that list and them together and output the new list.

Step 3: Find patterns
Of course, we need to check to make sure that nothing is null which is a must and of course we go through the list while that list is not null. Other than that, this is a very different type of problem that is very interesting.

Step 4: Check by hand
Attached is the images in a compressed file.

Chapter 1,2,3



Step 5: translate it to code

```
Public iNode sumList(iNode one,iNode two, int carryOver){
 If(one is null and two is null and carryover is 0) return null;
 Create a new iNode result;
  Int sumVal=carryover;
  If(one is not null) {
We add the data into sumVal;
}
 If(two is not null){
We add the data from two to sumVal;
}
//then we do what is necessary to get the result;
Return result;
}
```

Step 6: run test cases:

We hope we get similar outputs like the ones described in step 1.

Step 7: Debug failed test case:

This problem was really difficult I didn't have time to solve it. The reason being is the way we sum things instead of just simply summing as we go along we are actually collecting one number and then adding it to another number which is way different than simply summing the integers in each list and just adding them together.

L: This was a really interesting way at looking at addition using linked list. It is definitely a challenge to accomplish this.

Chapter 2: Problem 2.6 Palindrome: Implement a function to check if a linked list is a palindrome.

I: A palindrome is the same backwards and forwards. So, we are looking at something like mom or bob or 0,1,1,0. We need to check if our linked list is the same backwards as it is forwards.

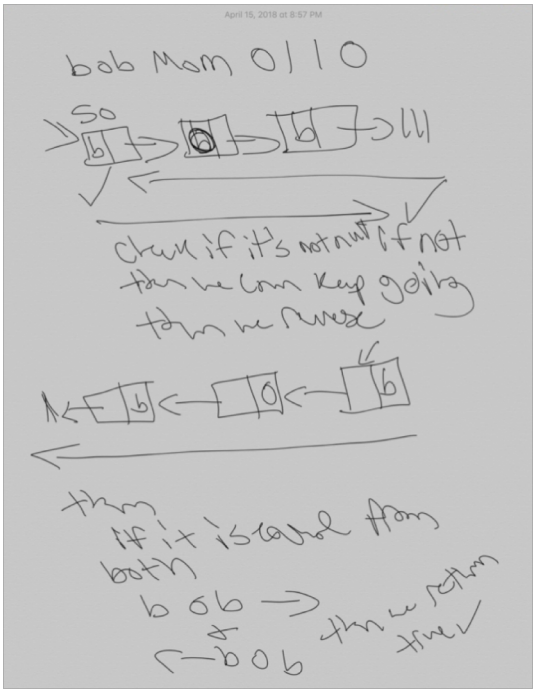
Chapter 1,2,3

D: So, first we need to check if its equal, we also have to reverse the list to check its backwards input and then we check if it is a palindrome or not. Therefore, we will need three methods to achieve this.
E,A: Step 1: Works some small instances by hand:
(b->o->b) would be a correct linked list, or if we want to use integers we use (0->1->1->0) and if we had (d->o->g) we can see that is not a palindrome.

Step 2: write down what you did:
We need to start with our two linked list nodes and make sure they are not null and check if the items in the list are not equal, so we can return false right away and if not keep traversing. Then the reverse method will need to simply reverse the list and traverse the other way. Then finally we create our last method and it checks if the list is indeed a palindrome and returns true or false.

Step 3: Find patterns
Of course, we need to make sure the list isn't null and traverse while the list is not null. We also need to check via boolean methods of course.

Step 4: Check by hand
Attached in a compressed file.



Step 5: translate it to code

```
Public boolean equal(iNode one,iNode two){
While(one!=null && two !=null){
    If(one.data!=two.data){
      Return false;
    }
```

Chapter 1,2,3

```
      One=one.next;
      Two=two.next;
}
Return both equal to null.
}
Public iNode reverse(iNode s){
iNode head=null;
while (s!=null){
  new iNode n;
  the new list is equal to head
   head is equal to new list
   node is node next;
}
Return head;
}
Public boolean palindrome(iNode s){
New iNode and we pass s;
Return isEqual and we pass s and the new list;
}
```

Step 6: run test cases:
I couldn't get it to run just right so I ended up doing it in pseudocode. Basically, we still need to get the result of either true for 0->1->1->0 and false for 0->2->0.
Step 7: Debug failed test case:
I need to keep working on methods to get it to work properly, but I got the basis of it working.
L: This was a medium level problem I pretty much got the basis it's just working out the kinks of my methods and I can fill in the pseudocode with real code.

Chapter 3: Problem 3.2 Stack Min: How would you design a stack which, in addition to push and pop has a function min which returns the minimum element? Push, pop, and min should all operate in 0(1) time.

I: We need to create a method that returns the minimum element and should be 0(1) like push and pop.
D: We need a variable to keep track of the minimum element and if we find a smaller value then that element becomes the minimum. We can do so with math.min but we need to pass two integers for it, so the best answer is Integer.MAX_VALUE. We can add it in the push method and push the value onto the stack and keep track of the minimum. Then we can have a method to check for the minimum using peek.
E,A: Step 1: Works some small instances by hand:
We could push (5) and that would be the smallest element and push (6) and 5 would still be smaller than push (3) and 3 would be the smallest. Than we pop, and we get 3 and the new minimum is 5 and we pop 6 and 5 is still minimum.

Step 2: write down what you did:

Chapter 1,2,3

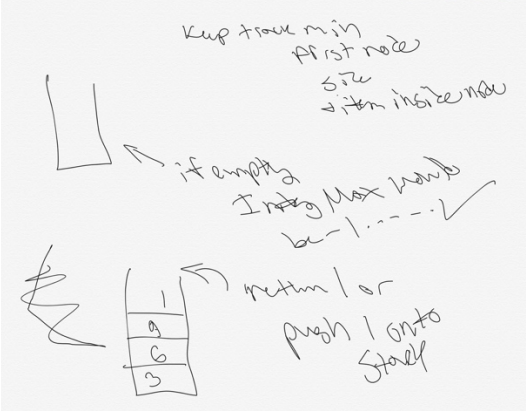
We create a method for min and we check if the stack is empty and if it isn't we return Integer.MAX_VALUE. Else we would peek. Then we could modify the push to make sure we keep track of it there and we can use the method we just created to do that. As well as, add it as a variable to the stack.

Step 3: Find patterns

We always have to check if the stack is empty so that we don't get an exception. We have to make sure we keep track of the minimum at all levels because if we only do it in one method and we need to peek or push we can't tell what the minimum is or if it still is the minimum.

Step 4: Check by hand

On the attached zip file.



Step 5: translate it to code

```
Class stack{
Public int min;
Public iNode first;
Public int size;
Public int item;

Public stack(int s ,int min){
    Item=s;
  This.min=min;
}
Public int min(){
If(stack.isEmpty()){
   Return Integer.MAX_VALUE;
}else{
Return peek().min;
}
}
}
Public void push(int item){
Int Nmin=Math.min(item,min());
Stack.push(new stack(item,Nmin);
```

}
Step 6: run test cases:
This is kind of a hard thing to manage being stacks can be anything in the sense that what is in the class can differ, so I used pseudocode when need be.
But essentially what should happen would be pushing and popping integers in the stack and out of the stack and we would get the minimum number that is accurate because we keep track of it all the time.
Step 7: Debug failed test case:
This is a little hard again because we don't exactly know if we have a preset stack object or we need to create our own so that is something I would ask.
L: I don't get to work with stacks much, but I like working with them. I like how we can create our own specific stack for what we need it for, this question is a great example of this.

Chapter 3: Problem 3.3 Stack of Plates: Imagine a (literal) stack of plates. If the stack gets too high, it might topple. Therefore, in real life, we would likely start a new stack when the previous stack exceeds some threshold. Implement a data structure SetOfStacks that mimics this. SetOfStacks should be composed of several stacks and should create a new stack once the previous one exceeds capacity. SetOfStacks.push() and setOfStacks.pop() should behave identically to a single stack (that is, pop() should return the same values as it would if there were just a single stack).
FOLLOW UP
Implement a function popAt(int index) which performs a pop operation on a specific sub-stack.

I: We are basically using a stack and if it is two high we need to create a new stack. Just like stacks of plates in real life. The push and pop should have the same values as it would if there was just one stack.
D: We need to create a class SetOfStacks and we could use an array list or maybe a hash table? Then the push method would have to leave off where the last stack started and check if it is full if it is we don't add anything if it isn't we can push onto the stack. Else we need to create a new stack and start pushing and adding on to the new stack. The pop method we again need to check the last stack if it is not null we can possibly add more things into the stack.

E,A: Step 1: Works some small instances by hand:
We can have a small stack like {1,4,5,6} and say its full then we need to create another one and push more things onto it, {} now we have two stacks one empty to continue adding to the stack until we need to pop.

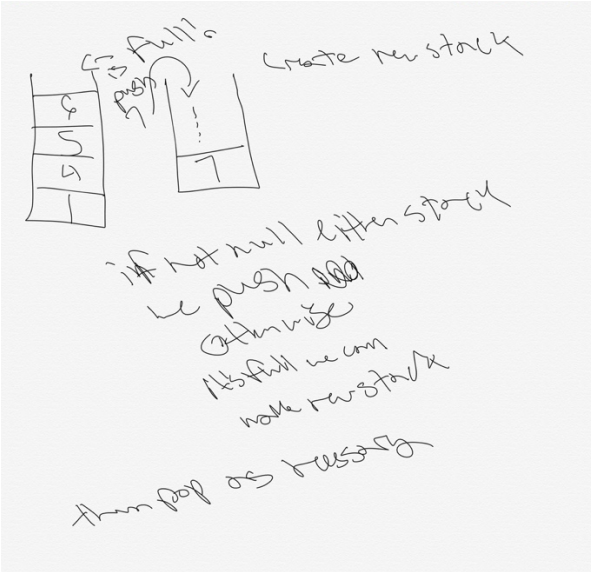
Step 2: write down what you did:
We need to modify the push and pop to create a new stack if it is necessary to do so. Then we push and pop as necessary as well. I explain it more in the D section of this section.

Step 3: Find patterns
We again modify the stack to fit our needs and we use it accordingly and we modify pop and push to create a new stack if we need to.

Chapter 1,2,3

Step 4: Check by hand

See the attached image compressed file.



Step 5: translate it to code

Public push(int item){

Stack get the previous stack

If(the prevStack is not null and the prevStack is not full)

prevStack.push(item);

else{

create new stack(new size)

newStack.push item;

stack array add to stack;

}

}

Public int pop(){

Get the prevStack;

If the prevStack is not null

We pop the last element and then we subtract it from the size of the stack;

Return that element we popped.

}

Step 6: run test cases:

This should work. For example, {1,2,5,6} its full so we create a new stack {} add 7 and 8 {7,8} then we pop 8 so we have a stack with just 7 {7}.

Step 7: Debug failed test case:

If the stack is null we would have a problem, so we would need to address that.

L: Again, stacks are awesome because we can customize them to whatever we need them to do. I think I will do all these in pseudocode because the class stack changes every time, but the pseudocode gets the point across. I like this idea of having this safe ability so that why we lessen the chance of having issues with the stack but also if it gets to big that can be a problem to so it's not perfect, we should probably have a cut off point.

Chapter 1,2,3

Chapter 3: Problem 3.4 Implement a MyQueue class which implements a queue using two stacks.

I: We create a queue class or maybe modify the one in the book which implements a queue and we modify it by using two stacks that will act like a queue. The stack is last in first out and the queue is first in first out. We can basically use the second stack as a holding bin to emulate a queue since they are opposite of each other. Similar to if we fill up a bowl or a glass with two much of something we get another bowl or glass and we use it to help balance out that filled item.

D: We need to create two stacks and we need to keep track of the size and we need to modify peek and pop so that we can do the opposite of what a stack usually does. In other words, we need to put everything into the empty stack from stack one into stack two and then we pop and push everything back to the old stack.

E,A: Step 1: Works some small instances by hand:

Say stack 1 is empty{}(just for a visual representation} and our new stack has 4 numbers push (1),(2),(5), then (3) and we pop because 3,5,2, and 1 into the empty stack so we have {3,5,2,1} and now we have a queue implementation because first in first out is now technically possible because our original stack has it reversed so the new stack can use pop the first in first out way and it's the same as queue would be if we popping 1,2,5, and 3 into the queue.

Step 2: write down what you did:

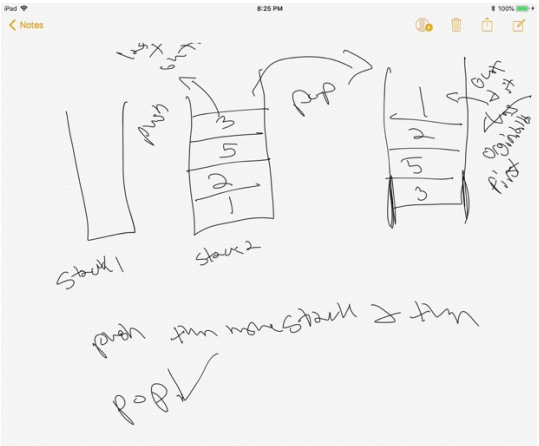
We need to create two stacks and then we need to create a method to get the size. Then we need a modified push method, but we need to modify it to take the newer stack, so we have to change it. Then a method to switch between the stacks which is putting all the stuff in stack 2 to stack 1 which is the stack that is empty and will help us to move everything over. Then we need a method to remove the stuff from the stack 1 back to stack 2.

Step 3: Find patterns

We can see the pattern in how the two stacks act like a queue in flipping the variables essentially upside down. We also don't really have to modify the pop and push much just make sure it goes to the right stack.

Step 4: Check by hand

Picture on attachment zip file.



Chapter 1,2,3

Step 5: translate it to code

```
public class myQueue{
Stack stack1;//create stack 1
Stack stack2;//create stack 2

public void pushQ(stack item){
Stack2.push(item);//assuming we have a push method
}
public void moveStacks(){
if(stack1.isEmpty()){//assuming we have a isempty method that checks if the stack is empty
while(!stack2.isEmpty()){
      stack1.push(stack2.pop());//moving everything from one stack to the other
}
Public stack isQ(){
moveStacks();
return stack2.pop();//now we are popping in order of a queue.
}
```

Step 6: run test cases:
Input:
//again {} is just for visual representation
{} for stack 1
{1,3,6,5} for stack 2 pushing (1), (3), (6), and (5).
Output: {5,6,3,1}//so we are popping 5, then 6, 3, and 1.

Step 7: Debug failed test case:
I am sure you can add a size method to keep track of the size and a peek method to peek in, but we just need to implement two stacks to work like a queue, so we are good with what we have.
L: I really like this problem, I worked on this before in I believe cs 2. I think it's really interesting how we can mimic a different data structure by using two of another is pretty cool. I am really curious about seeing other such things happening with other data structures.

Chapter 3: Problem 3.5 Sort Stack: Write a program to sort a stack such that the smallest items are on the top. You can use an additional temporary stack, but you may not copy the elements into any other data structure (such as an array). The stack supports the following operations: push, pop, peek, and isEmpty.

I: We are sorting a stack where the smallest will have the first element on top, so we can pop it or peek at it. We can use a temp stack, but we cannot use any other data structure.
D: The first thing I thought of was storing the popped variables into an array or array list and then sorting them and putting them back into the stack but since we can't do that according to the requirements of the question. Therefore, I had to think of a new plan. I was thinking about the towers of Hanoi and how we could almost implement something like that in this situation where

Chapter 1,2,3

we can have a temp variable hold some of the numbers and then we can essentially use the two stacks to move the numbers back in forth till they are sorted.

E,A: Step 1: Works some small instances by hand:

Say we push 1,3,2,6 into the stack. Then we have our temp stack and our temp variable. We pop 6 and we hold it in our temp. Then we get 2 and push it onto the temp stack. We then pop 3 and push it into the new temp stack and then we check the next value and our temp, and we see 1 and we push it onto the stack then push 6 on top. Then we push 6 back to the new stack and we hold 1 in the temp variable and we push 3 and then 2 and then 1 and now they are sorted with the smallest integer on top.

Step 2: write down what you did:

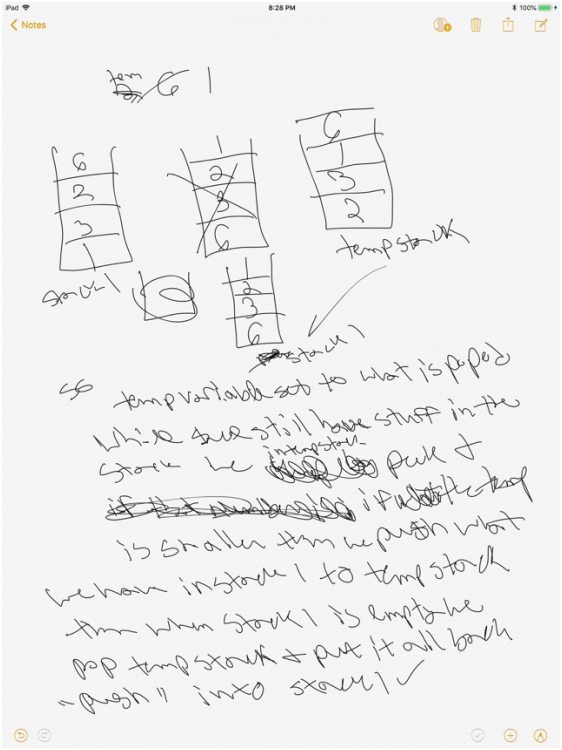
I first traced what I was thinking to make sure it worked I had to tweak it a bit but once I got the hang of it and I knew what I was doing made sense. I went ahead and worked on the code to get what I need. Which was to take advantage of the fact that we have that temp variable to help us move back and forth between the two stacks similar to the last problem, as well as, the towers of Hanoi.

Step 3: Find patterns

Well first off, the towers of Hanoi make me think we have two out of the three placements and the temp variable would act as the third and we shuffle around the variables tell we get it the way we need it to and then push everything back onto the stack in the right order. I also thought about the previous problem that used two stacks to implement a queue and since I had worked on that problem before I took the same idea into account and did something similar to that.

Step 4: Check by hand

See attached picture in compressed file.



Chapter 1,2,3

Step 5: translate it to code

```
public sortStack(Stack stack1){
int temp=stack.pop();
Stack tempStack= new Stack;
while(!stack1.isEmpty()){
      while(!tempStack.isEmpty() && tempStack.peek()>temp){
          stack1.push(tempStack.pop());
       }
tempStack.push(temp);
   }
//Then we copy everything back from the old stack to the new stack
while(!tempStack.isEmpty()){
      Stack1.push(tempStack.pop());
}
}
```

Step 6: run test cases:
Input stack: push (1), (3), (2), (6) and then we let the algorithm do its thing.
Output: a stack where we could pop 1,2,3, and 6.
Step 7: Debug failed test case:

I again like the previous problems used some pseudocode for the equation. I really liked this problem because it was an extension of the previous queue using two stacks problem.

L: I enjoyed this problem it was very similar to the previous one and was fun to work on.

Chapter 3: Problem 3.6 Animal Shelter: An animal shelter, which holds only dogs and cats, operates on strictly "first in, first out" basis. People must adopt either the "oldest" (based on arrival time) of all animals at the shelter, or they can select whether they would prefer a dog or a cat(and will receive the oldest animal of that type). They cannot select which specific animal they would like. Create the data structures to maintain this system and implement operations such as enqueue, dequeueAny, dequeueDog, and dequequeCat. You may use the built-in LinkedList data structure,

I: This question is asking us to create a queue that works similar to a stack and people can only adopt the oldest animal in the shelter they can basically only pick if they want a dog or a cat. So, we have to have system that maintains the system in which this is going to happen.

D: I am guessing by the names we are suggested we use a queue in a stack ideology and we use a linked list to build and store these animals, which I am sure will be objects, so we can get the necessary information we need. One for the dog and one for the cat. And then we keep track of when we dequeue and enqueue. Also, the name and the age of the dog or cat that are being enqueued.

E,A: Step 1: Works some small instances by hand:

We have two dogs Spot and Rufus and Rufus has been with us longer so whomever comes to get a dog will get Rufus and then the second person who comes in will get Spot. As for cats we have Millie and Dart and Dart has been with us longer so whomever comes to get a cat will get Dart and then Millie. If we get in another dog before someone adopts Spot, then Spot becomes the oldest and the new dog will have to wait till someone comes and adopts Spot.

Chapter 1,2,3

Step 2: write down what you did:

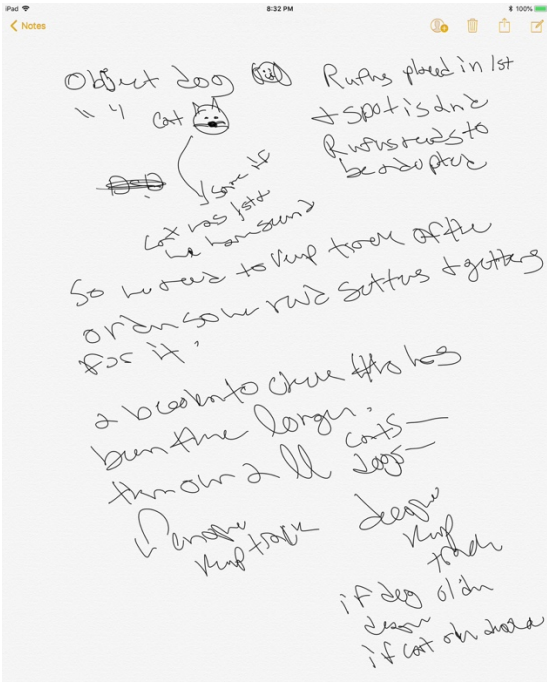
For sure we have to create an animal object, so we can store the name of the animals and what order they are in. Then we need to check if they are older we dequeue them and peek if need be. Of course, with objects we need setters and getters of the order of the animals, and create the linked list. Then dequeue the dog or cat.

Step 3: Find patterns

There isn't much pattern outside of building an object which we have done several times so that wasn't to different. I have never used an object with a queue or a stack or a linked list so that will be new to me.

Step 4: Check by hand

Pictures attached on the zip file.



Step 5: translate it to code

```
public class Animals{//we need to create an object that keeps track
int orderOfAnimals;
public String nameOfAnimal;
public void setOrderOfAnimals(int n){ orderOfAnimals=n;}
public int getOrderOfAnimals(){return orderOfAnimals}
public Animals(String ani){nameOfAnimal=ani}

public boolean OlderAni(Animals ani){
return this.OrderOfAnimals<ani.getOrderOfAnimals;
}
}
```

Chapter 1,2,3

```
Public class QueueAni{
Linked list dogs = new Linked list;
Linked list cats=new Linked list;
int orderOfAni=0;

public void enqueue( Animal ani){
ani.setOrderOfAnimals(orderOfAni);
orderOfAni++;

//if the cat or dog was the last to get added we want to keep track of that.
if(cat) is added last we pass the cat and its order;
if(dog) is added last we pass the dog and its order;
}
Public Animals dequeueAny(){
If(dogs equals 0){
We check which dog is the oldest by peeking;
Else if(cats equals 0){
We check which cat is the oldest by peeking;
}
If(the dog is older than the other dog or cat)
return dequeuedogs();
else{
return dequeuecats();
}
}
```

Step 6: run test cases:
Since I had to do this with some pseudocode I didn't get to run it. This one was pretty hard but hopefully it will return the oldest dog or cat and takes them out if they are being adopted.
Step 7: Debug failed test case:
I am sure I missed some other things, but I also think I should have added a method to say if the dog or cat was getting adopted as a boolean if they are then we dequeue and if not, we know they are just checking to see which dog or cat is the last.
L: This problem I think can be more efficient using other data structures then queues or stacks, but we didn't have that option. I like the fact that this problem is simplified but I also have a lot of questions about this problem. It is interesting for sure.