CMSI 281: Homework 3

Problem 1

removeAll:

find:

```java
private int find (String s) {
        for (int i = 0; i < uniqueSize; i++) { //u
            if (items[i].text.equals(s)) { // C1
                return i;                    // C2
            }
        }
        return -1;
    }
```

$T(u, s) = u(C1 + C2) \implies O(u)$

removeFromBack:

```java
    private void replaceFromBack (int index) {
        items[index] = items[uniqueSize-1]; //C1
        items[uniqueSize-1] = null; //C2
    }
```

$T(u, s) = C1 + C2 \implies O(1)$

removeOccurrances:

```java
private int removeOccurrences (String text, int count) {
        int index = find(text); //n

        // Case: no such string toRemove
        if (index == -1) {      //C1
            return 0;
        }

        Strand found = items[index];        // C2
        int newCount = found.count - count;//C3

        // Case: last instance to remove
        if (newCount <= 0) {// everything in here is constant lets just say C4 (all together)
            replaceFromBack(index); // it is constant (analysed before)
            size -= found.count;
            uniqueSize--;
            return 0;

        // Case: more than 1 Strand left
        } else {            // everything in here is also constatnso either way is constant worst case ( we will just keep the C4 above)
            found.count = newCount;
            size -= count;
            return newCount;
        }
    }
```

$T(u, s) = u + C1 + C2 + C3 + C4 \implies O(u)$

Then removeAll is...

```java
public void removeAll (String toNuke) {
        int index = find(toNuke); // u
        if (index != -1) { //C1
            removeOccurrences(toNuke, items[index].count); //u
        }
    }
```

ANSWER

$T(u, s) = u + C1n \implies O(u)$

getNth:

```java
public String getNth (int n) {
        if (n >= size || n < 0) {                       //C1
            throw new IllegalArgumentException();     //C2
        }

        for (int i = 0; i < uniqueSize; i++) {     //u
            Strand currentStrand = items[i];      //C3
            if (n < currentStrand.count) {       //C4
                return currentStrand.text;      //in worst case should never
get here
            } else {
                n -= currentStrand.count;       //C5
            }
        }

        // Should never get here...
        return null;                            //C6
    }
```

ANSWER

$$T(u, s) = C1 + C2 + u(C3 + C4 + C5) + C6 \implies O(u)$$

Problem 2

insert:

prependNode

```java
    private void prependNode (Node n) {
        Node oldHead = head;       //C1
        head = n;                 //C2
        if (oldHead != null) {    //C3
            head.next = oldHead;  //C4
            oldHead.prev = head;   //C5
        }
    }
```

$$T(u, s) = C1 + C2 + C3 + C4 + C5 \implies O(1)$$

find

```java
private Node find (String toFind) {
        for (Node curr = head; curr != null; curr = curr.next) { //u
            if (curr.text.equals(toFind)) {  //C2
                return curr;                 //worst case never gets gere
            }
        }
        return null;                       //C3
    }
```

$$T(u, s) = uC2 + C3 \implies O(u)$$

insertOccurrences:

```java
    private boolean insertOccurrences (String text, int count) {
        Node found = find(text);    //u

        // Case: new string, so add new Node
        if (found == null) {//C1(both cases in if and else are constant so either case worst case constant) lets just consider it goes in the if statement
            prependNode(new Node(text, count));           //C2
            uniqueSize++;                                 //C3

        // Case: existing string, so update count
        } else {
            found.count += count;
        }
        size += count;                                //C4
        modCount++;                                   //C5

        return true;                                  //C6
    }
```

$$T(u, s) = u + C1 + C2 + C3 + C4 + C5 + C6 \implies O(u)$$

```java
public void insert (String toAdd) {
        insertOccurrences(toAdd, 1); //u
    }
```

ANSWER

$$T(u, s) = u \implies O(u)$$

swap:

```java
public void swap (LinkedYarn other) {
        Node tempHead = head;             //C1
        int tempSize = size,             //C2
            tempUniqueSize = uniqueSize; //C3

        head = other.head;                //C4
        size = other.size;               //C5
        uniqueSize = other.uniqueSize;   //C6

        other.head = tempHead;           //C7
        other.size = tempSize;          //C8
        other.uniqueSize = tempUniqueSize;   //C9
        modCount++;                        //C10
        other.modCount++;                 //C11
    }
```

$T(u, s) = C1 + C2 + C3 + C4 + C5 + C6 + C7 + C8 + C9 + C10 + C11$

ANSWER

$\implies O(1)$

Problem 3

knit:

Constructor:

```java
    LinkedYarn (LinkedYarn other) {
        for (Node n = other.head; n != null; n = n.next) { //u1
            prependNode(new Node(n.text, n.count)); //C1 (constant from
above's analysis)
            size += n.count; //C2
            uniqueSize++; //C3
        }
    }
```

$T(u1, u2, s1, s2) = u1(C1 + C2 + C3) \implies O(u1)$

```java
    public static LinkedYarn knit (LinkedYarn y1, LinkedYarn y2) {
        LinkedYarn result = new LinkedYarn(y1); //u1
        for (Node n = y2.head; n != null; n = n.next) { //u2
            result.insertOccurrences(n.text, n.count);  //u1
        }
        return result; //C1
    }
```

$$T(u1, u2, s1, s2) = u1 + (u2u1) + C1 = u1 + u1u2 + C1 \implies O(u1u2)$$

2

In this problem as in the previous ones, I will first analyze the complexity of the methods that are called in the function in which we want to measure complexity. Once this is done it will be easier to determine its complexity.

empty constructor

```java
    LinkedYarn () {
        head = null;
        size = 0;
        uniqueSize = 0;
        modCount = 0;
    }
```

Clearly $O(1)$

creating an iterator

```java
Iterator (LinkedYarn y) {
    owner = y;
    itModCount = y.modCount;
    current = y.head;
    onCount = 0;
}
```

Clearly $O(1)$

getIterator

```java
public LinkedYarn.Iterator getIterator () {
        if (isEmpty()) {
            throw new IllegalStateException();
        }
        return new LinkedYarn.Iterator(this);
    }
```

Clearly $O(1)$

hasNext (iterator)

```java
public boolean hasNext () {
    if (current.count > onCount+1) {return true;}
    return isValid() && current.next != null;
}
```

Clearly $O(1)$

veryfyIntegrity()

```java
private void verifyIntegrity () {
    if (!isValid()) {
        throw new IllegalStateException();
    }
}
```

Clearly $O(1)$

next (iterator)

```java
public void next () {
    verifyIntegrity(); //constant from above
    onCount++;
    if (onCount >= current.count) {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        current = current.next;
        onCount = 0;
    }
}
```

Clearly $O(1)$

isValid

```java
public boolean isValid () {
    return owner.modCount == itModCount;
}
```

Clearly $O(1)$

getString

```java
public String getString () {
    verifyIntegrity(); //constant
    return current.text;
}
```

Clearly $O(1)$

remove occurrances

```java
private int removeOccurrences (String text, int count) {
    Node found = find(text); //u2 (the unique size) from problem 2

    // Case: no such string toRemove
    if (found == null) {
        return 0;
    }

    int newCount = found.count - count;
    modCount++;

    // Case: last instance to remove
    if (newCount <= 0) {
        deleteNode(found);
        size -= found.count;
        uniqueSize--;
        return 0;

    // Case: more than 1 entry left
    } else {
        found.count = newCount;
        size -= count;
        return newCount;
    }
```

In either case of this function (either of the if or else statements) all are constant so then $O(u2)$ is the complexity because of the find method.

remove

```
    public int remove (String toRemove) {
        return removeOccurrences(toRemove, 1); //u2
    }
```

clearly $O(u2)$ (see explanation for insert, this one is very similar)

contains

```
public boolean contains (String toCheck) {
    return find(toCheck) != null; //u2
}
```

clearly $O(u2)$

Now finally we can analize the following:

```
  /*
   * commonThreads returns a new LinkedYarn composed of all
   * occurrences that are common between y1 and y2
   * [X] Warning: this implementation is not very good
   */
  public static LinkedYarn commonThreads (LinkedYarn y1, LinkedYarn y2) {
      LinkedYarn result = new LinkedYarn(), //C1
                 y2Clone = new LinkedYarn(y2);     //u2

      for (LinkedYarn.Iterator i1 = y1.getIterator(); i1.hasNext(); i1.next()) { //s1
          String current = i1.getString(); //C2
          if (y2Clone.contains(current)) { //u2
              result.insert(current);    //u1
              y2Clone.remove(current);   //u2 (worst case y2clone is same as y1 but in mirror order)
          }
      }

      return result;  //C3
  }
```

$$T(u1, u2, s1, s2) = C1 + u2 + s1(C2 + u2 + u1 + u2) + C3$$

$$= C1 + u2 + s1(C2 + 2u2 + u1) + C3$$

$$= C1 + C3 + u2 + s1C2 + s1u2 + s1(u2 + u1)$$

ANSWER

$$\implies O(s1(u2+u1))$$

3

count:

```java
   public int count (String toCount) {
       Node toFind = find(toCount); // u2
       return (toFind == null) ? 0 : toFind.count; //C1
   }
```

Clearly $O(u2)$

```java
  /*
   * Alternative implementation of the above that is slightly better
   */
  public static LinkedYarn betterCommonThreads (LinkedYarn y1, LinkedYarn y2) {
      LinkedYarn result = new LinkedYarn(); //C1
      for (Node curr1 = y1.head; curr1 != null; curr1 = curr1.next) { //u1
          String text = curr1.text; //C3
          int count1 = curr1.count, //C4
              count2 = y2.count(text); //u2
          if (count2 > 0) {             //C5
              result.insertOccurrences(text, Math.min(count1, count2)); //Math.min constant time. //u1
          }
      }

      return result; //C6
  }
```

$$T(u1,u2,s1,s2) = C1 + u1(C3 + C4 + u2 + C5 + u1) + C6$$

ANSWER

$$\implies O(u1(u2+u1))$$