Exercise 13

Exercise 13.1.1

We made a local copy of the directory and pushed it to our itu github

Exercise 13.1.2

We followed the instructions and created a new branch newnumbers where we made changes to the number.txt file. We pushed this new branch along with the changes to our github repository.

Hereafter we merged the master branch with the newnumbers branch

Exercise 13.1.3

When merging the master branch with the newnumbers branch we are essentially accepting the changes made in the newnumbers branch and merge them with the code within our master branch. In a real work scenario, the master branch would be the branch containing the working code, whereas the newnumbers branch is "new feature" being developed. When this feature is done and is working, we can merge it back to the master branch (the main branch for the project).

Exercise 13.1.4

Basically, what rebase does is that it checks the last commit the two branches (e.g. the master and a feature branch) have in common and merges them together. Hereafter, it merges the new code from feature) into the branch (master) you are on with the latest commits from both branches.

We get a conflict because both file have now changed and git cannot figure out how to merge these two files as they were both changed in two different branches.
One would need to look at both files and make sure how to it is supposed to look. When that is done, one could mark them as resolved and then merge them onto the main master branch.

Exercise 13.2.1

1. The transform operation (delete(pos=0, length=11)) of deleting the string (delete(pos=0, length=11)) is send to the server by both Alice and Bob. The server performs both operations in the order they came in and sends it back to the clients (Alice an Bob).
2. ARROW PUSH

3. Alice sends the insert operation (insert(pos=0, content=”Introdction”)) to the server which performs the operation and sends it to its clients.
4. ARROW PUSH
5. Bob add an ‘u’ Introdction as an insert operation, which is send to the server and received by both Bob and Alice (and other clients if there were more).
6. ARROW PUSH
7. Both Alice and Bob are sending insert operations to the server, receiving them and performing them on their own state. Depending who sends it first (which arrow one pushes first) to the server, will decide in which order they will appear.
8. Both Alice and Bob are sending insert operations to the server, which performs the operations and sends it back to its clients. Depending who sends it first (which arrow one pushes first) to the server, will decide in which order the operations will be performed.
9. ARROW PUSH

Yellow

Yes, it depends on the order. This is because when pressing the arrow, we are sending the operations to the central server, that performs the operations – the server acts as a relay. Meaning that the server is single source of truth (single source of keeping track of the state of the string). The first operations that comes in will be the first one to be performed. If bob sends the letter ‘a’ and then ‘b’ to the central server, where Alice is inserting the letter ‘c’ between the two inserts of Bob, the final state will be ‘acb’.

Exercise 13.3

1. An example of a distributed system where availability is not always guaranteed is a web search system. For instance, a web search for a particular website would not always show the recent writes to the webpages of the website.
2. An example of a distributed system where partial tolerance is not always guaranteed is a money transfer system. For instance, a money transfer system cannot have partial tolerance because it deals with money being transferred from one account to the other.
3. An example where strong eventual consistency is different from strict consistency is transactions made on a bank account. With regards to the transactions made on a bank account, whenever there is a change in the bank balance, strict consistency enables all of the threads that have access to that bank balance to be notified of the change. However, when no other updates are made to the bank account balance, strong eventual consistency ensures that all threads that access that bank balance have the same value.

Exercise 3.4

1.

```java
class WingStructure{
   public static final int BUFFER_SIZE = 5;
   private int back = 0;
   public int[] buffer;

   public WingStructure() {
       buffer = new int[BUFFER_SIZE];
   }

   public synchronized int INC() {
       int oldBack = back;
       back = back + 1;
       return oldBack;
   }

   public synchronized int SWAP(int i) {
       int returnedResult = buffer[i];
       buffer[i] = 0;
       return returnedResult;
   }


}
```

```java
class WingBuffer {
   //static int iNull = null;
   private static final WingStructure rep = new WingStructure();


   public static int Deq(){
     //TO DO implement
     int index = 0;
     while (index < rep.BUFFER_SIZE){
         int returnedItem = rep.SWAP(index);
         if (returnedItem != 0){
             return returnedItem;
         }
         index += 1;
     }
     return index;
   }


   public static int Enq(int item) {
     //TO DO implement
     int availableIndex = rep.INC();
     rep.buffer[availableIndex] = item;
     return availableIndex;
   }

   public static void displayItems(){
       int index = 0;
       while(index < rep.BUFFER_SIZE){
           System.out.println(rep.buffer[index]);
           index += 1;
       }
   }



}



public class testProgram{
   public static void main(String[] args){
        for(int i = 0; i < 5; i++){
            WingBuffer.Enq(i);
        }

        //WingBuffer.displayItems();
        WingBuffer.Deq();
        WingBuffer.displayItems();

    }
}
```

2.

```java
import java.util.concurrent.atomic.AtomicIntegerArray;
import java.util.concurrent.atomic.AtomicInteger;


class WingStructure{
   private AtomicInteger back = new AtomicInteger();
   public AtomicIntegerArray buffer;

   public WingStructure() {
       buffer = new AtomicIntegerArray(5);
   }

   public synchronized int INC() {
       int oldBack = back.intValue();
       back.set(oldBack+1);
       return oldBack;
   }

   public synchronized int SWAP(int i) {
       int returnedResult = buffer.get(i);
       buffer.set(i,0);
       return returnedResult;
   }

}
```

```java
class WingBuffer {
   //static int iNull = null;
   private static final WingStructure rep = new WingStructure();


   public static int Deq(){
     //TO DO implement
     int index = 0;
     while (index < rep.buffer.length()){
         int returnedItem = rep.SWAP(index);
         if (returnedItem != 0){
             return returnedItem;
         }
         index += 1;
     }
     return index;
   }


   public static int Enq(int item)  {
     //TO DO implement
     int availableIndex = rep.INC();
     rep.buffer.set(availableIndex, item);
     return availableIndex;
   }

   public static void displayItems(){
       int index = 0;
       while(index < rep.buffer.length()){
           System.out.println(rep.buffer.get(index));
           index += 1;
       }
   }



}



public class testProgram2{
   public static void main(String[] args){
       for(int i = 0; i < 5; i++){
           WingBuffer.Enq(i);
```

```
        }

        //WingBuffer.displayItems();
        WingBuffer.Deq();
        WingBuffer.displayItems();

    }
}
```