Aufgabe 1:
Schiebeparkplatz

Team-ID: 00370

Team: Linus Schumann

Bearbeiter/-innen dieser Aufgabe:

Linus Schumann

15.11.2021

Inhaltsverzeichnis

1_Lösungsidee

Die Idee zur Lösung des Problems ist es für jedes Auto, den möglichen Weg in beide Richtungen zu prüfen und den besten auszugeben. Dabei werden immer so wenig Autos wie möglich bewegt, da immer nur die Autos bewegt werden, die auch bewegt werden müssen.

2_Umsetzung

2.1_Allgemeines

Das Programm wurde in Java geschrieben und mit der Java-JDK 17 kompiliert. Im Ordner „executables“ lässt sich die kompilierte .jar-Datei finden und mit der Batchdatei (Windows) oder dem Shell-Script (Linux und MacOS) ausführen. Außerdem lassen sich im Ordner „beispieldaten“ alle in dieser Dokumentation aufgeführten Beispiele wiedergefunden werden.

Im Ordner „source“ lassen sich folgende Java Klassen wiederfinden:

- Main: Die Klasse „Main“ ist die Hauptklasse sie übernimmt die Dateiauswahl und gibt den Dateinamen an die andere Klasse weiter
- a1: Die Klasse „a1“ (Aufgabe 1) übernimmt die Verarbeitung der Daten und die Ausgabe der Ergebnisse

Die Umsetzung im Programm wurde, wie oben erläutert, in der Klasse „a1“ umgesetzt und beginnt mit dem Einlesen der senkrecht parkenden Autos in eine Tree-Map. Danach werden die waagerecht parkenden Autos auch in eine Tree-Map eingelesen.

Dann wird für jedes Auto, das nicht „freie Fahrt“ hat, der Weg in beide Richtungen berechnet. Anschließend werden dann für jenes Auto die besten Verschiebungen ausgegeben oder es wird ausgegeben, dass es keine Möglichkeit gibt das Auto auszuparken.

2.2_Berechnung einer Richtung

Die Berechnung einer Richtung wird durch eine rekursive Funktion gelöst, an die zuerst die Richtung und die weiteren Informationen zur Berechnung übergeben. Dazu gehören…

- Die Tree-Map der senkrecht parkenden Autos
- Das Auto das verschoben werden muss
- Die Richtung in die verschoben wird
- Die Anzahl an Veränderungen (beim ersten Aufruf 0)
- Die Veränderungen als Tree-Map (beim ersten Aufruf leer)
- Die Anzahl an Tauschen

Zuerst wird überprüft ob noch ein Auto verschoben werden muss.

2.2.1_Ein Auto muss verschoben werden

In diesem Fall wird überprüft, ob nach dem Verschieben das Auto aus dem Bereich des Parkplatzes fahren würde. Wenn dies der Fall ist, wird die Funktion komplett beendet, da dies bedeutet, dass es keine Lösung in dieser Richtung gibt.

Wenn dies nicht der Fall ist, wird überprüft, ob ein weiteres Auto bewegt werden muss, um dieses Auto zu bewegen. In diesem Fall wird das aktuelle Auto getauscht und zur Anzahl an Tauschen wird 1 addiert. Wenn nicht, wird das Auto verschoben und danach die Funktion erneut aufgerufen.

2.2.2_Kein Auto muss verschoben werden

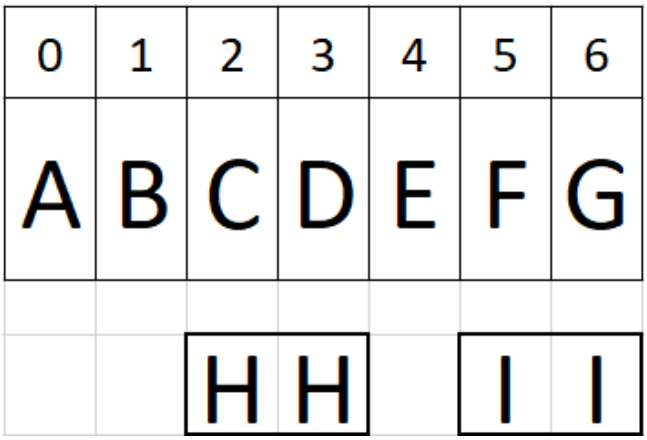
In diesem Fall wird überprüft, ob diese Richtung besser war und wenn ja wird die Anzahl an Veränderungen und die Veränderungen gespeichert.

2.3 Warum eine Tree-Map?

Die Vorteile der Tree-Map gegenüber der „normalen“ Hash-Map liegt darin, dass die Tree-Map automatisch nach den Keys der Map sortiert wird. Dies ist wichtig, da beim Ändern von Werte-Paaren, also dem hinzufügen und löschen, die Reihenfolge verändert wird und dann in falscher Reihenfolge ausgegeben werden würde.

3_Beispiele

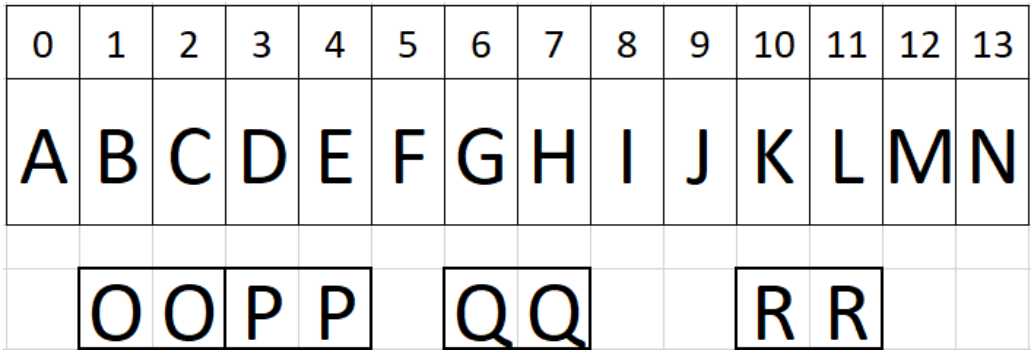
3.1_Parkplatz 0



Ausgabe „parkplatz0.txt“

```
car A -> nothing needs to be changed
car B -> nothing needs to be changed
car C -> H:1 to right
car D -> H:1 to left
car E -> nothing needs to be changed
car F -> H:1 to left    I:2 to left
car G -> I:1 to left
```

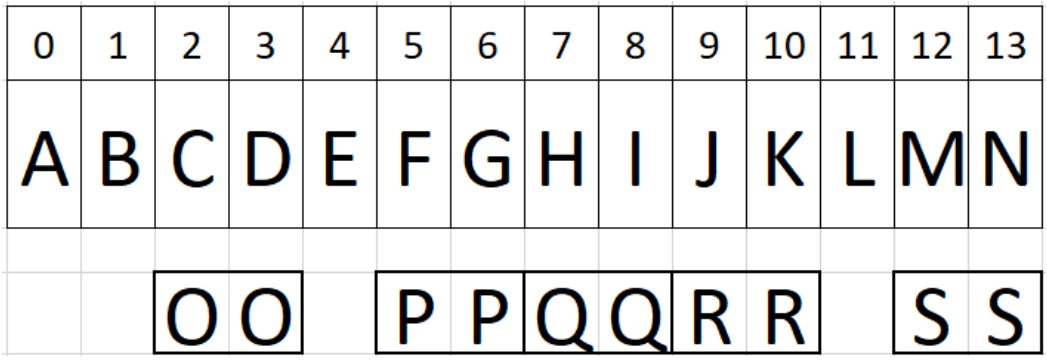
3.2_Parkplatz 1



Ausgabe „parkplatz1.txt“

```
car A -> nothing needs to be changed
car B -> O:1 to right   P:1 to right
car C -> O:1 to left
car D -> P:1 to right
car E -> O:1 to left    P:1 to left
car F -> nothing needs to be changed
car G -> Q:1 to right
car H -> Q:1 to left
car I -> nothing needs to be changed
car J -> nothing needs to be changed
car K -> R:1 to right
car L -> R:1 to left
car M -> nothing needs to be changed
car N -> nothing needs to be changed
```

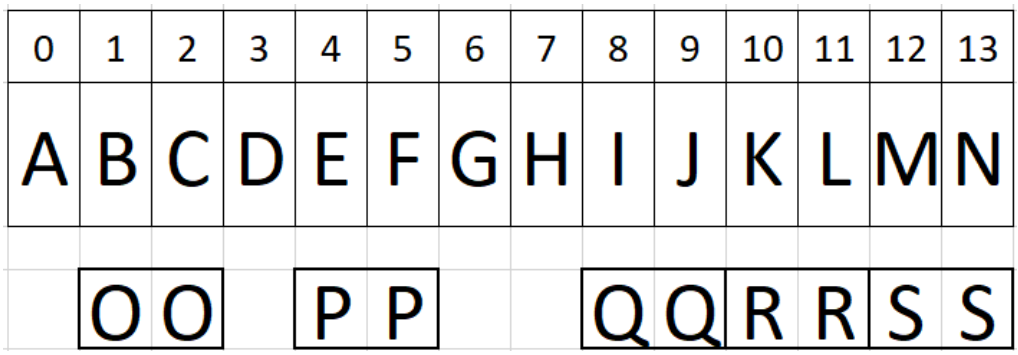
3.3_Parkplatz 2



Ausgabe „parkplatz2.txt“

```
car A -> nothing needs to be changed
car B -> nothing needs to be changed
car C -> O:1 to right
car D -> O:1 to left
car E -> nothing needs to be changed
car F -> O:1 to left    P:2 to left
car G -> P:1 to left
car H -> Q:1 to right   R:1 to right
car I -> P:1 to left    Q:1 to left
car J -> R:1 to right
car K -> P:1 to left    Q:1 to left    R:1 to left
car L -> nothing needs to be changed
car M -> P:1 to left    Q:1 to left    R:1 to left    S:2 to left
car N -> S:1 to left
```

3.4_Parkplatz 3

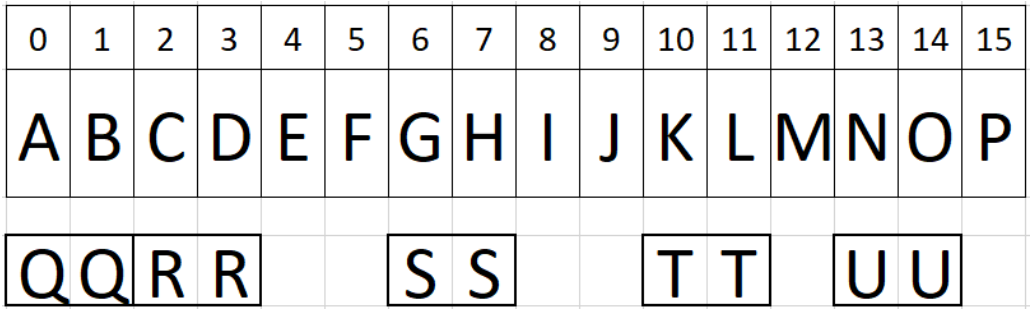


Ausgabe „parkplatz3.txt

```
car A -> nothing needs to be changed
car B -> O:1 to right
car C -> O:1 to left
car D -> nothing needs to be changed
car E -> P:1 to right
car F -> P:1 to left
```

```
car G -> nothing needs to be changed
car H -> nothing needs to be changed
car I -> Q:2 to left
car J -> Q:1 to left
car K -> Q:2 to left    R:2 to left
car L -> Q:1 to left    R:1 to left
car M -> Q:2 to left    R:2 to left    S:2 to left
car N -> Q:1 to left    R:1 to left    S:1 to left
```

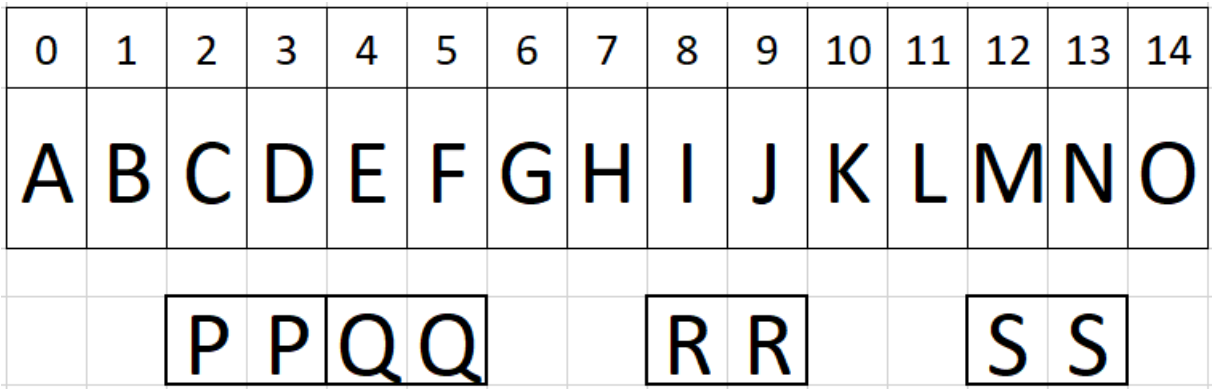
3.5_Parkplatz 4



Ausgabe „parkplatz4.txt“

```
car A -> Q:1 to right   R:1 to right
car B -> Q:2 to right   R:2 to right
car C -> R:1 to right
car D -> R:2 to right
car E -> nothing needs to be changed
car F -> nothing needs to be changed
car G -> S:1 to right
car H -> S:1 to left
car I -> nothing needs to be changed
car J -> nothing needs to be changed
car K -> T:1 to right
car L -> T:1 to left
car M -> nothing needs to be changed
car N -> U:1 to right
car O -> U:1 to left
car P -> nothing needs to be changed
```

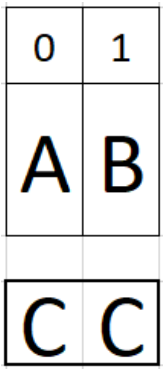
3.6_Parkplatz 5



Ausgabe „parkplatz5.txt“

```
car A -> nothing needs to be changed
car B -> nothing needs to be changed
car C -> P:2 to left
car D -> P:1 to left
car E -> Q:1 to right
car F -> Q:2 to right
car G -> nothing needs to be changed
car H -> nothing needs to be changed
car I -> R:1 to right
car J -> R:1 to left
car K -> nothing needs to be changed
car L -> nothing needs to be changed
car M -> S:1 to right
car N -> S:1 to left
car O -> nothing needs to be changed
```

3.7_Eigener Parkplatz 1



Ausgabe „eigenerParkplatz1.txt“

```
car A : No solution
car B : No solution
```

An diesem Beispiel kann man nochmal sehr deutlich sehen, dass „No solution“ ausgegeben wird, wenn es keine Möglichkeit für das Auto zum Ausparken gibt.

4_Quellcode

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.util.TreeMap;

public class a1 {
    public static int numberOfParkingCars;

    public static int lowestTurnsNumber;

    //___use TreeMap instead of HashMap because of sorting___//
    public static TreeMap<Character, Integer> lowestTurns = new
TreeMap<>();


    public static void calculateMovement(String filename) throws
FileNotFoundException {

        //____Create-HashMap____//
        TreeMap<Integer, Character> horizontalCars = new TreeMap<>();

        //_____READ-FILE_____//
        File file = new File(filename);
        Scanner scanner = new Scanner(file);
        numberOfParkingCars = (int) scanner.nextLine().charAt(2) - 64;
        int numberOfCars = scanner.nextInt();
        for (int i = 0; i < numberOfCars; i++) {
            char car = scanner.next().charAt(0);
            int position = scanner.nextInt();
            horizontalCars.put(position, car);
        }

        //___Check best option for each car___//
        for (int car = 0; car < numberOfParkingCars; car++) {

            //_____Clear-Variable_____//
            lowestTurnsNumber = Integer.MAX_VALUE;
            lowestTurns.clear();

            //_____Check if something needs to be changed_____//
            if (horizontalCars.containsKey(car) ||
horizontalCars.containsKey(car - 1)) {
                //____find-car-that-needs to be changed____//
                int currentCar;
                if (horizontalCars.containsKey(car)) {
                    currentCar = car;
```

```
                } else {
                    currentCar = car - 1;
                }
                //___Copy-HashMap-for-second-try___//
                TreeMap<Integer, Character> horizontalCarsLeft = new
TreeMap<>(horizontalCars);
                TreeMap<Integer, Character> horizontalCarsRight = new
TreeMap<>(horizontalCars);
                //____try to move left and right___//
                calculateMovementForOneCar(car, currentCar,
horizontalCarsLeft, "left", 0, new TreeMap<>(), 0);
                calculateMovementForOneCar(car, currentCar,
horizontalCarsRight, "right", 0, new TreeMap<>(), 0);

                //____Check if there is no solution____//
                if (lowestTurnsNumber == Integer.MAX_VALUE) {
                    System.out.println("car " + (char) (car + 65) + " : No
solution");
                    continue;
                }
            }
            //___Print-out-Best-Solution___//
            if (!(lowestTurnsNumber == Integer.MAX_VALUE)) {
                System.out.print("\ncar " + (char) (car + 65) + " -> ");
                lowestTurns.forEach((k, v) -> System.out.print(k + ":" + (v
> 0 ? v + " to right    " : v * -1 + " to left    ")));
            } else {
                System.out.print("\ncar " + (char) (car + 65) + " ->
nothing needs to be changed");
            }
        }
    }

    private static void calculateMovementForOneCar(int car, int currentCar,
TreeMap<Integer, Character> cars, String direction, int numberOfTurns,
TreeMap<Character, Integer> turns, int numberOfCarChanges) {
        //_____Check if something needs to be changed_____//
        if (cars.containsKey(car) || cars.containsKey(car - 1)) {
            //___Get-character-of-current-Car___//
            char name = cars.get(currentCar);

            //___Check if the car isn't out of bound__//
            if ((direction == "left" && currentCar != 0) || (direction ==
"right" && currentCar < (numberOfParkingCars - 2))) {
                //___Check if another car hasn't to move___//
                if ((!cars.containsKey(currentCar - 2) && direction ==
"left") || (!cars.containsKey(currentCar + 2) && direction == "right")) {
                    //___Shift car__//
                    cars.remove(currentCar);
                    currentCar -= (direction == "left" ? 1 : -1);
                    cars.put(currentCar, name);

                    //___Save movement in HashMap___//
                    if (turns.containsKey(name)) {
                        int offset = turns.get(name);
                        turns.put(name, offset - (direction == "left" ? 1 :
-1));
                    } else {
                        turns.put(name, (direction == "left" ? -1 : 1));
                    }

                    //___increase number of Turns___//
                    numberOfTurns += 1;
```

```
                    //___Check if car has changed___//
                    if (numberOfCarChanges != 0) {
                        currentCar = currentCar + (direction == "left" ? 3
: -3);
                        numberOfCarChanges -= 1;
                    }
                } else {
                    //___Change-To-Next-Car___//
                    currentCar -=  (direction == "left" ? 2 : -2);
                    numberOfCarChanges += 1;
                }
            } else {
                return;
            }
            if (numberOfTurns <= lowestTurnsNumber) {
                //___Cal methode again___//
                calculateMovementForOneCar(car, currentCar, cars,
direction, numberOfTurns, turns, numberOfCarChanges);
            }
        } else {
            //___check if this solution is better___//
            if (numberOfTurns < lowestTurnsNumber ||
(numberOfTurns==lowestTurnsNumber && turns.size() < lowestTurns.size())) {
                lowestTurnsNumber = numberOfTurns;
                lowestTurns.clear();
                lowestTurns.putAll(turns);
            }
        }
    }
}
```