Aufgabe 1

Team-ID: 00112

Team-Name:

Bearbeitet von:

22. November 2021

Inhaltsverzeichnis

Information: Im Folgenden bezeichne ich die Autos in der oberen Reihe bzw. auf den normalen Parkplaetzen als „normale Autos“. Die quer stehenden Autos der unteren Reihe bezeichne ich als „quer stehende Autos“.

1 Loesungsidee

1.1 Grundidee

Um dieses Problem zu loesen, muss man fuer jedes normale Auto wissen

- welche quer stehenden Autos
- in welche Richtung und
- wie oft

bewegt werden muessen, damit das normale Auto nicht mehr blockiert ist. Um dies herauszufinden, durchlaufe ich folgende Schritte fuer alle normalen Autos:

1. Ich ueberpruefe zunaechst, ob das normale Auto durch ein quer stehendes Auto blockiert ist.
2. Wenn dies der Fall ist, finde ich heraus, welche quer stehenden Autos man wie oft verschieben muss, damit das normale Auto nicht mehr blockiert ist. Dazu verschiebe ich das quer stehende Auto, welches das normale Auto blockiert, einmal nach rechts und einmal nach links, bis das normale Auto nicht mehr blockiert ist. Dazu durchlaufe ich folgende Schritte (aus Sicht des Autos, welches das normale Auto blockiert):
   a) Die Richtung wird abgebrochen, wenn das Verschieben in die jeweilige Richtung aufgrund einer Wand nicht möglich ist.
   b) Wenn man gegen ein quer stehendes Auto fahren wuerde, durchläuft man diese Schritte anfangend bei a.) aus Sicht des Autos, das das jetzige Auto blockiert.
   c) Anschließend wird das Auto in die jeweilige Richtung (links oder rechts) um einen Platz verschoben. Diese Bewegung wird sich gemerkt. Nun wird wieder bei a. angefangen, sofern das normale Auto noch blockiert ist.
3. Nun zaehle ich die benötigten Bewegungen der quer stehenden Autos fuer die beiden Richtungen. Wenn eine Richtung durch die Blockierung durch eine Wand (bzw. das Ende des Parkplatzes) abgebrochen wurde, wird die jeweils andere Richtung ausgewaehlt. Wenn beide Richtungen nicht abgebrochen wurden, wird die mit weniger Bewegungen von quer stehenden Autos ausgewaehlt. Die Bewegungen der quer stehenden Autos der ausgewaehlten Richtung ist nun die Loesung fuer ein normales Auto.

1.2 Bezug zum Beispiel des Aufgabenblatts

Im Beispiel des Aufgabenblatts wueder dieses Verfahren, am Beispiel des normalen Autos „F“, wie folgt aussehen:

Das Auto F ist durch Auto „I“ blockiert. Also muessen wir die folgenden Schritte durchfuehren. Wir fangen mit der Richtung „links“ an, versuchen also das quer stehende Auto „I“ nach links zu verschieben, bis Auto „F“ nicht mehr blockiert ist:

1. „I“ ist links nicht durch eine Wand blockiert. Somit muss nicht abgebrochen werden. Zudem befindet sich links auch kein anderes Auto, sodass kein anderes Auto verschoben werden muss. „I“ wird nun einen Platz nach links verschoben.
2. Da „F“ noch immer blockiert ist, muessen wir „I“ weiter verschieben. Doch „I“ ist nun links durch das Auto „H“ blockiert. Somit muessen wir „H“ um einen Platz nach links verschieben. Da „H“ links nicht blockiert ist, koennen wir dies ohne weiteres tun.
3. Nun koennen wir „I“ um einen weiteren Platz nach links verschieben. Da „F“ nun nicht mehr blockiert ist, koennen wir aufhoeren.

Nun haben wir fuer die Richtung „links“ „H“ einmal nach links und „I“ zwei Mal nach links verschoben. Als naechstes testen wir die Richtung „rechts“, versuchen also „I“ nach rechts zu verschieben. Dabei merkt man allerdings sofort, dass „I“ nicht nach rechts verschoben werden kann, da sich dort eine Wand befindet. Das heißt, die Richtung „rechts“ wird abgebrochen.

Nun kann man die Richtungen vergleichen. Aufgrund dessen, dass „rechts“ abgebrochen wurde, waehlen wir „links“. Und geben fuer das normale Auto „F“ folgendes aus: „H 1 links, I 2 links“

2 Umsetzung

Die Loesungsidee habe ich in Java 8 implementiert.
Nach dem Ende des nun beschriebenen Prozesses kann man die Loesungen fuer jedes normale Auto eines Schiebeparkplatzes als eine Liste an Strings aus einer Instanz der Klasse „SchiebeParkplatz“ entnehmen. Mein Programm tut dies und gibt die Ergebnisse in die Konsole aus.

2.1 Initialisierung

Die Initialisierung findet im Konstruktor der „SchiebeParkplatz“- Klasse statt. Der Konstruktor bekommt als Uebergabewert einen String „fileName“.
Dort rufe ich die Methode „init“ (ebenfalls aus „SchiebeParkplatz“) mit dem Uebergabewert „fileName“ auf. Wenn die „init-Methode“ false zurueck gibt, wird eine Exception geworfen, da sie dies tut, wenn die Datei nicht gelesen werden kann.
In der „init“-Methode lese ich an als Erstes die Textdatei mit dem angegebenen Namen (fileName) aus. Falls die Datei nicht gefunden/gelesen werden kann, wird wird false zurueckgegeben. Zunaechst entnehme ich aus der ersten Zeile, wie viele normale Autos es geben wird, indem ich den angegebenen Character-Wert des ersten Autos von dem des letzten Autos abziehe und 1 addiere. Diesen Wert speichere ich als field (globale Variable) mit dem Namen „carAmount“. Als nachstes initialisiere ich ein Objekt des Typs (bzw. der Klasse) „CarRow“ - ebenfalls als field - mit dem Namen „carRow“. Die CarRow ist dafuer da die quer stehenden Autos zu speichern und operationen mit ihnen vorzunehmen. Deswegen initialisiert die „CarRow“ Klasse in ihrem Konstruktor ein Character-Array mit der Groeße carAmount. Daraufhin gehe ich durch die weiteren Zeilen (ab Zeile 2) der Datei, und entnehme fuer jede Zeile den Namen (bzw. die Bezeichnung) und den Index (bzw. die Position) des Autos. Diese quer stehenden Autos fuege ich zum CarRow Objekt mit der Methode „addCar“ hinzu. Diese Methode setzt in das Character-Array an der Stelle index und an der Stelle index+1 den Namen des quer stehenden Autos. Das bedeutet also, dass in der „carRow“ die Reihe des Parkplatzes mit den quer stehenden Autos als Character-Array gespeichert ist. Zudem initialisiere ich die Liste von Strings „results“, in welcher am Ende die Loesungen fuer jedes normale Auto gespeichert sein wird. Dann wird true zurueckgegeben.

2.2 Die Loesung fuer jedes Auto

Nach der Initialisierung gehe ich durch die Zahlen 0 bis (exklusive) carAmount und rufe fur jeden Index die Methode „getCarsSolution“ auf. Diese Methode gibt die Loesung fur das normale Auto an der Stelle index zurueck. Den Rueckgabewert dieser Methode fuege ich zu der soeben initialisierten Liste „results“ hinzu.

2.2.1 Die „getCarsSolution“-Methode

```
1 private String getCarsSolution(int index)
```

Die Methode „getCarsSolution“ der Klasse „SchiebeParkplatz“ gibt die Loesung (also die noetigen Bewegungen der quer stehenden Autos) fuer das normale Auto an der Stelle index als String zurueck.
Dies tut sie, indem sie zuerst einen String „prefix“ initialisiert, welcher aus dem Namen des normalen Autos an der Stelle index, einem Doppelpunkt, sowie einem Leerzeichen besteht (<Auto-Name> + „: „).
Nun ueberprueft die Methode, ob das normale Auto an der Position index nicht durch ein quer stehendes Auto blockiert ist, indem sie die Methode „isEmpty“ mit dem Uebergabewert „index“ auf unserem CarRow-Objekt („carRow“) aufruft. Die ‘isEmpty“-Methode ueberprueft, ob das Character-Array der „CarRow“-Klasse (bzw. der quer stehenden Autos) an der Stelle index initialisiert ist (also ob, durch die „addCar“-Methode, ein Character-Wert fuer die Stelle „index“ festgelegt wurde). Sollte das normale Auto nicht blockiert sein, wird ausschließlich der Prefix („prefix“) zurueckgegeben. Sollte das Auto allerdings blockiert sein „besorgt“ sich die Methode das „Result“ (Erklaerung folgt) des normalen Autos einmal mit der „Direction“ (Richtung) „LEFT“ und einmal mit der „Direction“ (Richtung) „RIGHT“, ueber die Methode „getResult“, welche den Index und die „Direction“ als Uebergabeparameter bekommt. Direction ist eine Enumeration mit den Konstanten „LEFT“ und „RIGHT“ (mehr Informationen folgen). Danach werden die „Results“ der jeweiligen Richtung verglichen und die bessere als String zurueckgegeben. Die bessere ist dabei die, die nicht „unSuccessful“ (Informationen, wann dies passiert, folgen) war, oder, falls es beide nicht waren, die mit weniger totalen Autoverschiebungen.

2.2.2 Die „Result“-Klasse

Die Klasse „Result“ speichert die Informationen, welche quer stehenden Autos, wie haeufig bewegt wurden mit Hilfe einer HashMap mit dem Namen „moves“ (Character zu Integer). In welcher Reihenfolge die Autos bewegt wurden, wird mit Hilfe einer Liste von Integern gespeichert - „keys“. Diese Liste ist wichtig, um am Ende die Bewegungen in einer logischen Reihenfolge auszugeben. Die Bewegungen mussen in der Reihenfolge der Liste ausgegeben werden. Zusaetzlich merkt sich die Klasse, ob die Richtung „unSuccessful“ war (das ist ein Boolean).

Die Klasse Result hat ebenfalls die methode „add“, welche als Uebergabeparameter den Namen eines Autos als Character-Wert bekommt. Diese Methode erhoeht den Wert des Namens in der HashMap um 1 (startend bei 0). Zudem loescht sie den Eintrag des Namens in der „keys“-Liste (sofern vorhanden) und fuegt ihn anschließend wieder hinzu. So ist der aktuellste Wert immer als letztes in der Liste und nie mehrfach vorhanden.

2.2.3 Die „getResult“-Methode

```
1 private Result getResult(int index, Direction direction)
```

In dieser Methode der Klasse „SchiebeParkplatz“ wird das Bewegen des quer stehenden Autos, welches das normale Auto auf dem Platz index blockiert, in eine als „Direction“ angegebene Richtung verschoben, bis das normale Auto nicht mehr blockiert ist. Die dafuer benoetigten Bewegungen werden in einem Result zurueckgegeben.
Dazu wird als erstes ein Result Objekt („result“) und dann eine Kopie des CarRow Objekts - „carRowClone“ - erstellt. Dies hat den Zweck, dass sich die Werte der eigentlichen carRow nicht aendern sollen, wenn man das Verschieben der Autos simuliert. Dann wird der Integer „subIndex“ mit dem „echten Index“ des angegebenen Index initialisiert. Dazu benutze ich die „getRealIndex“-Methode der „CarRow“-Klasse. Der „echte“ Index ist der Index, welcher der linke Teil eines quer stehenden Autos ist (im Beispiel des Aufgabenblatts waere der echte Index von drei zwei). Daraufhin wird die Prozedur der Loesungsidee umgesetzt:

Solange (while) der Index („index“) im „carRowClone“ besetzt ist (also das normale Auto blockiert ist), wird „subIndex“ auf den Wert der Methode „moveRecursively“ der Klasse „CarRow“, welche auf „carRowClone“ aufgerufen wird (Erklaerung der „moveRecursively“-Methode folgt), gesetzt. Diese Methode bekommt die Werte „subIndex“, „direction“ und eine Aufgabe, was passiert, wenn ein Auto verschoben wird, uebergeben. Wenn ein Auto verschoeben wird, wird zum Result „result“ der Name des verschobenen, quer stehenden Autos via der Methode „add“ der Klasse „Result“ hinzugefuegt (siehe 2.2.2 Die „Result“-Klasse). Diese Aufgabe wird in Form eines Consumers<Character> angegeben. Aufgrund dessen, dass „moveRecursively“ eine Ausnahme ausloest, wenn der Index des Platzes, auf welchen das quer stehende Auto als naechstes geschoben wuerde, kleiner 0 oder groeßer/gleich der Laenge der carRow ist, wird diese aufgefangen. In diesem Fall wird das Result auf „unSuccessful“ gesetzt und die while-Schleife abgebrochen.
Nachdem das Result vollstaendig bearbeitet wurde, also entweder auf „unSuccessful“ gesetzt wurde oder das normale Auto nicht mehr blockiert ist, wird das Result zurueckgegeben.

2.2.4 Die „moveRecursively“-Methode

```
1 public int moveRecursively(int index, Direction direction, Consumer<Character> onSuccess)
            throws IllegalArgumentException
```

Die „moveRecursively“-Methode der „CarRow“ Klasse verschiebt ein querstehendes Auto an der Stelle „index“ um einen Schritt in die Richtung „direction“ und gibt den Index zurueck auf welchen das Auto gezogen ist. Sollte das querstehende Auto blockiert sein, wird ausschließlich, durch eine rekursive Benutzung dieser Methode, das querstehende Auto, welches das Auto, das eigentlich verschoben werden soll blockiert, verschoben und der Index „index“ zurueckgegeben.

Dazu wird zunaechst der index festgelegt, auf welchen das querstehende Auto aufgrund der angegebenen Direction ziehen wuerde: „checkIndex“. Diese wird durch die angegebene „Direction“ und den Index erkannt. Wenn checkIndex kleiner als 0 oder groeßer/gleich der Laenge der carRow ist, wird eine Exception

(Ausnahme) ausgeloest und die Methode damit beendet.

Wird die Exception nicht ausgeloest, fahre ich fort, indem ich in einer if-Abfrage den Rueckgabewert der Methode „move“ aus „CarRow“ ueberpruefe (Erklaerung der Methode folgt). An die „move“-Methode wird „index“, „checkIndex“ und „direction“ uebergeben. Gibt die „move“-Methode true zurueck, wird die uebergebene Aufgabe, bzw. der Consumer, einmal mit dem Namen/Characters des quer stehenden Autos an der Position „checkIndex“ (es ist nicht „index“, weil die „move“-Methode die Werte im Character-Array bereits veraendert hat) ausgefuehrt und anschließend der neue, linke Index des querstehendes Autos zurueckgegeben. Wenn die „move“-Methode allerdings false zurueckgibt, ruft sich die „moveRecursively“ rekursiv erneut mit folgenen Uebergabewerten auf: der „echte“ Index (siehe „getResult“-Methode) von „checkIndex“, direction, onSuccess (die Aufgabe). Anschließend wird „index“ zurueckgegeben, da sich das Auto nicht wirklich bewegt hat, sondern nur das Auto, das das eigentliche blockiert.

2.2.5 Die „move“-Methode

```
public boolean move(int index, int to, Direction direction)
```

Die „move“-Methode der Klasse „CarRow“ bekommt als Uebergabeparameter „index“ (dies ist der Ausgangspunkt eines quer stehenden Autos), „to“ (dies ist der index, auf welchem das Auto nach dem Bewegen stehen wird, aber auf welchem es vorher nocht nicht stand) und die Direction und gibt einen Boolean (Wahrheitswert) zurueck.
Die „move“-Methode gibt false zurueck, wenn es nicht moeglich ist, an die Stelle „to“ zu ziehen. Dies wird durch die Methode „isEmpty“ der CarRow-Klasse mit dem Uebergabewert „to“ ueberprueft. „isEmpty“ ueberprueft, ob das Character-Array der „CarRow“-Klasse an der uebergebenen Stelle initialisiert ist. Ist der Zug zu „to“ moeglich, wird dieser in die angegebene Richtung durch die Methode „goLeft“ bzw. „goRight“ (je nach angegebener Direction) mit dem Uebergabewert „index“ ausgefuehrt. Dadurch werden die Werte im Character-Array veraendert und anschließend wird true zurueckgegeben.

3 Beispiele

Aufgrund meiner selbstgeschriebenen Tests, welche Sie in meiner Abgabe im „Quellcode“ Ordner finden, kann ich fuer jede Datei, welche auf der Website angegeben ist, ueberpruefen, ob mein Programm funktioniert. Aufgrund von Platzgruenden habe ich nicht alle Ergebnisse meines Programmes hier eingefuegt. Aber Sie finden alle Ergebnisse im Ordner Aufgabe1/Ergebnisse

3.1 Aufgabe des Aufgabenblatts/parkplatz0.txt

Die Loesung meines Programmes fuer diesen Schiebeparkplatz ist folgende:
A:
B:
C: H 1 rechts
D: H 1 links
E:
F: H 1 links, I 2 links
G: I 1 links

Durch Vergleichen mit dem Aufgabenblatt sieht man, dass diese Loesung korrekt ist.

3.2 Aufgabe des Aufgabenblatts/parkplatz1.txt

Die Loesung meines Programmes fuer diesen Schiebeparkplatz ist folgende:
A:
B: P 1 rechts, O 1 rechts
C: O 1 links
D: P 1 rechts

E: O 1 links, P 1 links
F:
G: Q 1 rechts
H: Q 1 links
I:
J:
K: R 1 rechts
L: R 1 links
M:
N:

Mit dem Wissen, dass der Schiebeparkplatz wie folgt aussieht:

```
[A|B|C|D|E|F|G|H| I |J|K|L|M|N]
[  |O|O|P|P|  |Q|Q|   |  |R|R|   |  ]
```

erkennend man durch Testen im Kopf, dass die jeweilige Anweisung fuer die normalen Autos Sinn ergeben.

3.3 praeferenzen2.txt bis praeferenzen5.txt

Fuer diese Schiebeparkplaetze funktioniert mein Programm ebenfalls problemlos. Zudem kann ich aufgrund meiner Selbstgeschriebenen Test-Klasse sagen, dass die Loesungen Sinn ergeben.

4 Quellcode

4.1 „Direction“-Enumeration

```
//          name     newLeftIndex    checkIndex
LEFT("links", i -> --i,      i -> i-1),
RIGHT("rechts", i -> ++i,    i -> i+2);

private final String name;
private final Function<Integer, Integer> newLeftIndex, checkIndex;

Direction(String name, Function<Integer, Integer> newLeftIndex,
          Function<Integer, Integer> checkIndex) {
  this.name = name;
  // index which will be the new left index after a move
  this.newLeftIndex = newLeftIndex;
  // index which will be new and has to be checked after a move
  this.checkIndex = checkIndex;
}
```

4.2 Result

4.2.1 Result: Fields und Konstruktor

```
private final Direction direction; // for message
private final Map<Character, Integer> moves; // move amount per car (by name)
private final List<Character> keys; // List to save order of moved cars
private boolean unSuccessful; // If the Direction was unsuccessful

public Result(Direction direction) {
  this.direction = direction;

  this.moves = new HashMap<>();
  this.keys = new ArrayList<>();
```

```
}
```

4.2.2 Result: „add“-Methode

```
// Add car by increasing it's value in Map (moves) and
// (Re-) Putting it into List (keys)
public void add(char car) {
  this.moves.put(car, this.moves.getOrDefault(car, 0) + 1);

  if (this.keys.contains(car))
    this.keys.remove((Character) car);
  this.keys.add(car);
}
```

4.2.3 Result: „compare“-Methode

```
// Returns Result which was not unSuccessful. If both were successful:
// returns result with less absolut moves
public Result compare(Result that) {
  return this.isUnSuccessful() ? that
      : that.isUnSuccessful() ? this : this.getAbsMoves() < that.getAbsMoves() ? this : that;
}
```

4.3 CarRow

4.3.1 CarRow: Fields und Konstruktoren

```
  private final char[] cars;
  private final int size;

  // Initializes size and new char-array
  public CarRow(int size) {
      this.size = size;
      this.cars = new char[this.size];
  }

  // Initializes size and char-array
  private CarRow(char[] cars) {
      this.cars = cars;
      this.size = cars.length;
  }
```

4.3.2 CarRow: „addCar“-Methode

```
   // Set car's name into index and index+1
  public void addCar(int index, char carName) {
      this.cars[index] = carName;
      this.cars[index + 1] = carName;
  }
```

4.3.3 CarRow: „moveRecursively“-Methode

```
    // Returns left/real index of car after move (if moved)
    // Throws IllegalArgumentException on wrong index
    public int moveRecursively(int index, Direction direction, Consumer<Character> onSuccess)
            throws IllegalArgumentException {
        // Index where car will get moved to
        int checkIndex = direction.getCheckIndex(index);
        // Check ArrayOutOfBounds
        if (!this.validIndex(checkIndex))
            throw new IllegalArgumentException();

        // Check value of move methode
        if (this.move(index, checkIndex, direction)) {
            // Use onSuccess (task)
            onSuccess.accept(this.cars[checkIndex]);
            return direction.getNewLeftIndex(index);
        } else {
            // Calls itself if there is a blocking car blocking the current blocking
            // car. In this case the cars which are blocking the current car are
            // getting moved recursive.
            this.moveRecursively(this.getRealIndex(checkIndex), direction, onSuccess);
            return index;
        }
    }
```

4.3.4 CarRow: „move“-Methode

```
    // Moves car at index in direction
    // Returns if methode was able to move the car
    public boolean move(int index, int to, Direction direction) {
        // Check if move is possible
        if (!this.isEmpty(to)) return false;

        // Go in given direction
        switch (direction) {
            case LEFT:
                this.goLeft(index);
                break;
            case RIGHT:
                this.goRight(index);
                break;
        }

        return true;
    }

    // Check if there is a car at index
    public boolean isEmpty(int index) {
        return this.isNull(this.cars[index]);
    }

    private boolean isNull(char c) {
        return c == Character.MIN_VALUE;
    }

    private void goLeft(int index) {
        this.cars[index + 1] = Character.MIN_VALUE;
        this.cars[index - 1] = this.cars[index];
    }

    private void goRight(int index) {
        this.cars[index + 2] = this.cars[index];
        this.cars[index] = Character.MIN_VALUE;
    }
```

4.4 SchiebeParkplatz

4.4.1 SchiebeParkplatz: Fields und Konstruktor

```
    private int carAmount;
    private CarRow carRow;
    private List<String> results;

    // Throws IOException on failing to read file with name fileName
    public SchiebeParkplatz(String fileName) throws IOException {
        // Initialize
        if (!init(fileName))
            throw new IOException();

        // Add every car's solution to results (List<String>)
        for (int index = 0; index < this.carAmount; index++)
            this.results.add(this.getCarsSolution(index));
    }
```

4.4.2 SchiebeParkplatz: „init“-Methode

```
    // Returns false on failing to read file, else true
    private boolean init(String fileName) {
        if (fileName == null) return false;

        // Initialize results List
        this.results = new ArrayList<>();

        // Get file's lines
        List<String> lines;
        try {
            lines = Files.readAllLines(Paths.get(fileName));
        } catch (IOException e) {
            return false;
        }

        // Get carAmounts by getting min and max from file
        String[] lineOne = lines.get(0).split("␣");
        char min = lineOne[0].charAt(0), max = lineOne[1].charAt(0);
        this.carAmount = max - min + 1;

        // Initialize carRow
        this.carRow = new CarRow(this.carAmount);
        // Fill carRow by getting name and index of each car from file
        for (int i = 2; i < lines.size(); i++) {
            String[] line = lines.get(i).split("␣");
            char name = line[0].charAt(0);
            int index = Integer.parseInt(line[1]);
            this.carRow.addCar(index, name);
        }

        return true;
    }
```

4.4.3 SchiebeParkplatz: „getCarsSolution“-Methode

```
    // Returns Solution of car at index as String
    private String getCarsSolution(int index) {
        // Initialize String which starts with the normal cars name
        String prefix = (char) (65 + index) + ":␣";

        // Check if car is not blocked, if so only returns prefix
        if (this.carRow.isEmpty(index)) {
            return prefix;
        } else {
            // Gets a result with information about direction, moves per blocking car
```

```
            // and if the Direction was successful
            Result resultLeft = this.getResult(index, Direction.LEFT);
            Result resultRight = this.getResult(index, Direction.RIGHT);
            // Return better result (not unSuccessfull or fewer moves) as String
            return resultLeft.compare(resultRight).getResult(prefix);
        }
    }
```

4.4.4 SchiebeParkplatz: „getResult“-Methode

```
    // Counts every blocking car's moves by simulating cars movement
    // in a copy of the carRow until the given index is not blocked
    // anymore and returns it in a Result.
    private Result getResult(int index, Direction direction) {
        // Create result and clone of carRow
        Result result = new Result(direction);
        CarRow carRowClone = this.carRow.clone();

        // Initialize integer subIndex (real index/left index)
        int subIndex = carRowClone.getRealIndex(index);

        // Loop as long as carRowClone is not empty at the index:
        while (!carRowClone.isEmpty(index)) {
            try {
                // Set subIndex to new left index of car
                subIndex = carRowClone.moveRecursively(subIndex, direction, result::add);

                // Catches exception which gets thrown on too big/too small index
            } catch (IllegalArgumentException e) {
                // Setting result unSuccessful and breaking loop
                result.setUnSuccessful(true);
                break;
            }
        }
        return result;
    }
```