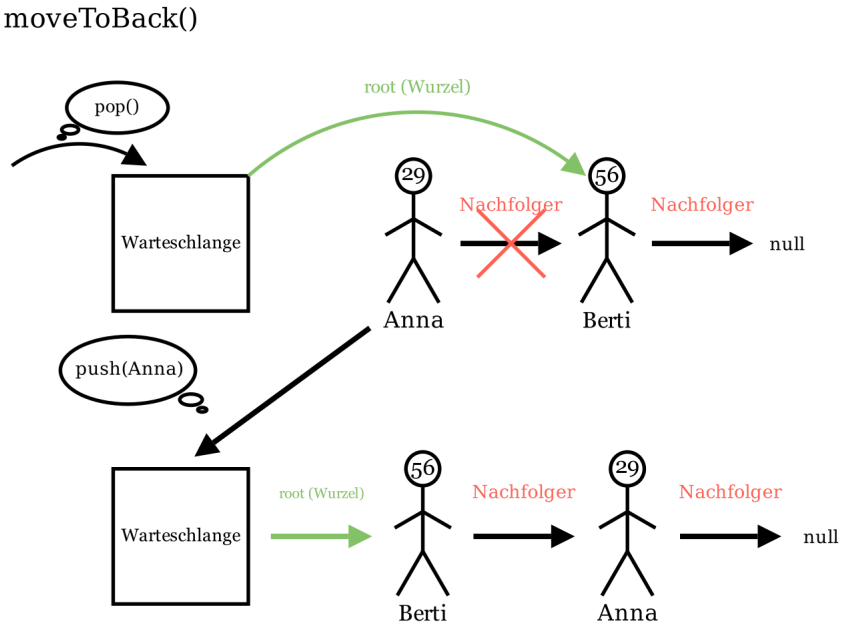
Lösungen zu den Aufgaben)
Aufgabe 2: Mit unseren Vorarbeiten (insbesondere das Entfernen der Nachfolger-Referenz des Ersten) ist diese Aufgabe sehr leicht. Der Erste der entfernt wird, wird mit der normalen hintenAnfügen() Methode wieder eingefügt.

```
//class Human
public void setNext(Human human) {
    next human;
}
//class MyListLinked
public void moveToBack(){
    this.push(this.pop());
}
```

Graphisch veranschaulicht:



Aufgabe 3: Für diese Aufgabe müssen wir wieder rekursiv denken. Wir können der Warteschlange bzw. dem Ersten nicht direkt sagen, zwischen welche beiden Listenelemente der alte Erste gesetzt werden muss, da der Erste nur den neuen Ersten und dieser nur seinen Nachfolger kennt.
Wir können aber bereits die Länge der Liste bestimmen, d.h. wir können die Methode prüfen lassen, ob das Einfügen an dieser Stelle überhaupt möglich ist und andernfalls abbrechen lassen (alternativ könnte dann auch normal am Ende eingefügt werden). Außerdem fangen wir ungültige Eingaben ab, die Position 1 wäre die gleiche Stelle wie zuvor, also wieder den neuen Ersten. Eine negative Eingabe oder 0 macht ebenfalls keinen Sinn.
Damit sind die Vorbereitungen abgeschlossen und der alte Erste kannt entfernt und in einer lokalen (temporären tmp) Variable zwischengespeichert werden. Danach wird die entsprechende Funktion auf dem Ersten aufgerufen und wir wechseln in die Mensch-Klasse.
Es gibt im Wesentlichen zwei Ideen, wie die entsprechende Position gefunden werden kann:

1. Zum Ende laufen und von dort aus in umgekehrter Reihenfolge „Durchzählen“ ähnlich der Längenbestimmung. (Aufwendig!)
2. Eine „Zählvariable“ definieren und mitgeben.

Die zweite Variante ist deutlich einfacher zu implementieren und zu verstehen. In jedem Schritt wird geprüft, ob der Zähler eins kleiner als die gewünschte Position ist, damit das ebenfalls mitgegebene Objekt der Klasse Mensch als Nachfolger dieses Menschen gesetzt werden kann. Ist diese Position noch nicht erreicht, so wird mit einem um eins erhöhten Zähler die Methode auf dem nächsten in der Schlange aufgerufen.

```
//class MyListLinked
public void moveBack(int position) {
    if(position > this.length() || position <= 1) {
        return;
    }
    Human tmp = pop();
    first.moveBack(tmp, position, 1);
}
//class Human
public void moveBack(Human h, int position, int counter) {
    if(counter == position -1) {
        h.setNext(this.getNext());
        next = h;
    } else {
        next.moveBack(h, position, counter++);
    }
}
```

Aufgabe 4: Diese Methode durchläuft die Schlange so lange, bis die Nachfolger-Referenz null ist und lässt die entsprechenden Informationen mit einem print auf die Konsole schreiben.
Die presentation()-Methode ist dabei nicht zwingend nötig und wird nur verwendet, um Ausgabe und Logik des Durchlaufs zu trennen. So kann sie leicht verändert werden, ohne dass über den restlichen Code nachgedacht werden muss.

```
//class MyListLinked
public void printList() {
    if(first == null){
        System.out.println("No list here to print!");
    } else {
        first.printList();
    }
}
// class Human
public void printList(){
   presentation();
   if (next != null) next.printList();
}
public void presentation(){
     System.out.println("I am " + name + " and I am " + age + " years old");
}
```

Aufgabe 5: Die grundlegende Idee der Methode verändert sich durch diese Änderung nicht, allerdings muss jeder Mensch seine Informationen an den String anhängen, d.h. an jedem Menschen in der Warteschlange wird ein String erzeugt und anschließend der nächste Mensch nach seinen Informationen „gefragt“, indem auf dem Nachfolger wiederum die printList()- Methode aufgerufen wird.
Der Befehl \n erzeugt eine neue Zeile, andernfalls würde der String nur eine lange Zeile ergeben, wenn man ihn weiterverwendet und ausgibt.

```
//class MyListLinked
public String printList() {
    if(first == null){
        System.out.println("No list here to print!");
        return "";
    } else {
        return first.printList();
    }
}
```

```java
// class Human
public String printList(){
 String toReturn = "I am " + name + " and I am " + age + " years old";
   if (next != null){
     return toReturn + "\n" + next.printList()
   }
   return toReturn;
}
```

Aufgabe 6: Für diese Methode wird wieder der rekursive Aufbau der Liste genutzt, im Wesentlichen entspricht die Logik der moveBack() - Methode. Wir benötigen wieder einen Zähler für die Such-Methode in der Klasse Mensch, da die einzelnen Menschen ihre Position nicht kennen und wir mitzählen müssen.
Die Festlegung, dass -1 zurückgegeben wird, wenn das Element nicht in der Liste ist, ist wieder willkürlich festgelegt, andere Lösungen sind denkbar.

```java
//class MyListLinked
public int searchHumanInQueue(Human human) {
    int counter = 0;
    int searched = first.searchHumanInQueue(human,counter);
    return searched;
}
//class Human
public int searchHumanInQueue(Human human, int counter){
    if(human.equals(this)){
        return counter;
    }
    if(next != null) {
        return next.searchHumanInQueue(human, counter+1);
    } else {
        return -1;
    }
}
```

Aufgabe 7: Die Struktur der Funktion ist wieder dieselbe, diesmal sogar ohne eine Zählvariable. Soll die rekurisve Methode nicht implementiert werden, kann auch wieder auf die Methode aus Aufgabe 6 zurückgegriffen werden.

```java
//class MyListLinked
public boolean contains(Human human) {
    if(first != null) {
        return first.contains(human);
    } else {
        return false;
    }

}
//class Human
public boolean contains(Human human) {
    if(human.equals(this)) {
        return true;
    } else {
        if(next != null){
            return next.contains(human);
        } else {
            return false;
        }
```

```
        }
    }
    //Alternatively use searchHumanInQueue()!
```

Aufgabe 8: Für diese Methode kann wieder auf die Struktur von Aufgabe 3 zurückgegriffen werden. Auch hier empfiehlt es sich, die Referenz auf den nächsten Menschen in der Liste vor der Rückgabe wieder zu entfernen, da sonst „von außerhalb“ die Liste modifiziert werden könnte. Wird die erste Position gewählt, so kann einfach die vorneEntfernen-Methode verwendet werden.

```
    //class MyListLinked
    public Human removeAt(int position) {
        if(position == 1) {
            return pop();
        }
        if(first != null) {
            int counter = 1;
            return first.removeAt(position, counter);
        } else {
            return null;
        }
    }
    //class Human
    public Human removeAt(int position, int counter){
        if(next == null) {
            return null;
        }
        if(counter == position - 1) {
            Human tmp = next;
            next = next.getNext();
            return tmp;
        } else {
            return next.removeAt(position, counter + 1);
        }
    }
```

Aufgabe 9: Im Gegensatz zur Feld-Implementierung ist das Zusammenfügen zweier Listen mit der verketteten Struktur denkbar einfach. Es muss lediglich das letzte Element der ersten Liste gefunden werden (hier über eine Helfer-Methode, die die Liste durchläuft und immer prüft, ob das nächste Element gleich null ist) und das erste Element der zweiten Liste als dessen Nachfolger gesetzt werden. Alternativ kann auch das erste Element der zweiten Liste mit Hilfe der push-Methode angefügt werden.

```
    //class MyListLinked
    public Object concatenate(MyListLinked toConcat) {
        if(first == null) {
            return toConcat;
        }
        Human end = this.findEnd();
        end.setNext(toConcat.getFirst());
        return this;
    }
    //class Human
    public Human findEnd(){
        if(next == null) {
            return this;
        } else {
            return next.findEnd();
```

```
        }
    }
```