Skript zu Listen

Lars Wechsler

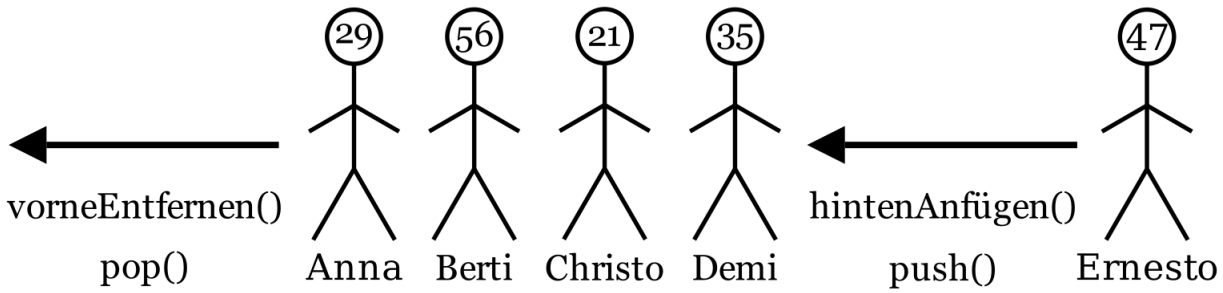
30. August 2022

1 Einleitung

Gemeinhin ist eine Liste eine Aufzählung von Dingen, seien es Zahlen (Altersangaben einer Klasse), Worte (Einkaufszettel) oder eine Liste von Personen.

- Eier
- Mehl
- Salz
- Karrotten

Listen sind so alltäglich, dass wir überlicherweise gar nicht aktiv über sie nachdenken.
Unser erstes Ziel in diesem Schuljahr ist die Übertragung des Konzepts der Liste in die Informatik. Hier benötigen wir noch viel öfter **„Zusammenstellungen“** von Dingen. Das Zusammenfassen von Objekten zu Listen ist sogar so zentral, dass es Programmiersprachen gibt, die auf die Liste als Basiselement zurückgreifen, z.B. Lisp.
Traditionell ist die Liste ein umso wichtigeres Element der Sprache, je funktionaler diese angelegt ist. In objektorientierten Sprachen sind Listen in ihrer Funktionalität zwar ebenfalls vorhanden, allerdings nicht immer ohne einen gewissen Aufwand zu verwenden.
In den folgenden Kapitel werden wir selbst Implementierungen für Listen in Java entwickeln. Wie oft in der Programmierung gibt es dabei **nicht** einen Königsweg, sondern verschiedene Ansätze, die unterschiedliche Vor- und Nachteile bringen.
Als Prototyp einer Liste wird uns dabei eine **Warteschlange** dienen (ganz bildlich gedacht), z.B. die Schlange vor einer Kasse.



Hinweis: Um sprachlich stringenter zu bleiben und Denglisch möglichst zu vermeiden, werden die Methoden im Fließtext häufig auf Deutsch stehen bzw. beschrieben werden, mit der üblichen englischen Übersetzung bei der ersten Verwendung in Klammern. Auch die Abbildungen werden, wenn nötig, beide Schreibweisen enthalten, der Quellcode aktuell nur englische Bezeichungen (Eine zweite Version des Skripts mit deutschem Quellcode ist geplant, wenn nötig). Für Prüfungen sind sowohl deutscher als auch englischer Quellcode in Ordnung. Begründungen müssen aber in jedem Fall auf Deutsch geschrieben werden. Im Abitur ist auch der Quelltext i.d.R. auf Deutsch.
Zurück zum obigen Bild. Wir werden Anna, Berti, Christo, Demi und Ernesto noch öfter sehen, da sie unsere Liste füllen und auch wieder verlassen werden. Sie stehen in einer **Warteschlange (Queue)**. Auch eine Warteschlange ist nur eine Form von Liste, allerdings mit speziellen Eigenschaften. Halten sich alle an die Regeln, so ist es nur möglich, dass sich jemand hinten anstellt oder vorne aus der Warteschlange entfernt wird, weil er dran ist und „abgearbeitet“ wird. In unserer zukünftigen Modellierung entspricht dies den Methoden vorneEntfernen() (pop()) bzw. hintenAnfügen() (push()).
Hinweis: Übergabeparameter werden im Text häufig weggelassen, natürlich müsste die zweite Methode etwa so aussehen: hintenAnfügen(Mensch mensch). Die Warteschlange gehorcht dem sogenannten **FIFO-Prinzip** (First In, First Out). Neben diesem gibt es noch weitere Ansätze, siehe dazu auch Kapitel 6, das sich mit Typen von Listen beschäftigt.
Es bleibt abschließend die Frage, wie sich die Idee einer Warteschlange objektorientiert umsetzen lässt.

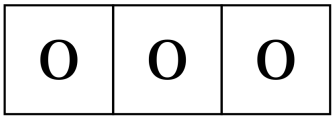
2 Die Feld-Liste (ArrayList)

Man unterscheidet im Wesentlichen zwei große Untertypen von Listenimplementierungen. Die, die sich auf Felder (arrays) stützen und die, die eine „verzeigerte“ Struktur aufweisen (linked lists). Da Felder bereits aus der zehnten Jahrgangsstufe bekannt sind, ist dieser Ansatz der naheliegenderer und wir beginnen mit ihm. Für den Moment wollen wir uns ein Feld als reservierten Speicherplatz vorstellen, im Folgenden symbolisiert durch eine Aneinanderreihung an Quadraten. Aktuell soll das Feld noch vollständig leer sein.

In Java werden Felder allerdings standardmäßig mit 0 bzw. null intialisiert, wenn keine weiteren Vorgaben gemacht werden. **Beispiel**:

```
int[] array = new int[3];
```

Die Variable array zeigt nun zu einem Feld, das 3 mal den Eintrag 0 enthält, in Java-Schreibweise: {0,0,0}



Neben primitiven Datentypen wie int können auch Listen von beliebigen Objekten gebildet werden. Nehmen wir an, wir haben eine Klasse „Mensch “bereits erzeugt, dann wird

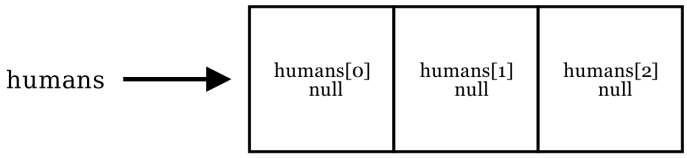
```
Human[] humans = new Human[5];
```

ein Feld mit 5 Einträgen null erzeugen, auf das durch die Variable humans zugegriffen wird.
Wir können auf beliebige Einträge eines zuvor erzeugten Feldes zugreifen, wenn wir Zugriff auf die Variable haben, die den Anfang der Liste speichert, z.B.:

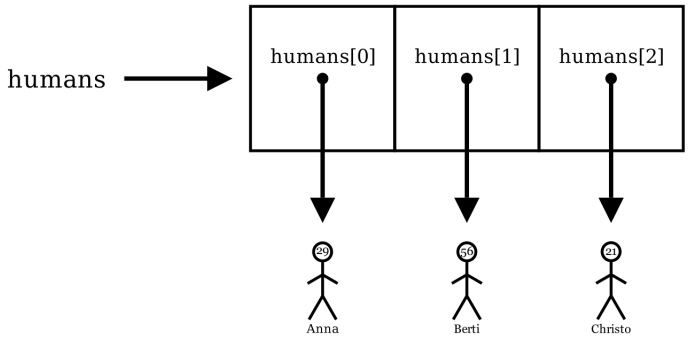
```
int[] numbers = {5,3,42,17,-2};
int thirdEntry = numbers[2];
System.out.println(thirdEntry);
```

Der obige Code wird die Zahl 42 auf die Konsole schreiben. Dieses Beispiel soll daran erinnern, dass in der Informatik bei 0 zu zählen begonnen wird. Auf den für uns dritten Eintrag wird also mit der „2“ zugegriffen.

Versuchen wir stattdessen, z.B. mit humans[0] auf den ersten Menschen zuzugreifen, werden wir nur null erhalten,da dieses Feld aktuell nur null-Einträge enthält.



Haben wir die Methode hintenAnfügen() (push()) implementiert und drei Menschen Anna, Berti und Christo erzeugt und zur Warteschlange hinzugefügt:



Dabei stehen die Strichys (Germanistik-Exkurs) ebenfalls wieder für eine bestimmte Menge an Daten, die (ggf. an ganz anderer physikalischer Stelle) im Speicher abgelegt sind.
Grundsätzlich bietet das Feld an sich in seiner üblichen Implementierung in Java bereits alle Eigenschaften, die wir von einer Liste verlangen würden. Wenn unsere Warteschlange benutzt wird, soll aber nur am Anfang entnommen und am Ende angefügt werden können. Es ist also mehr Kontrolle nötig.
Beginnen wir damit unsere Warteschlange als neue Klasse zu definieren. Da es unser Ziel ist eine Schlange von Menschen zu modellieren, brauchen wir zunächst eine Klasse, die einen Menschen beschreibt.

Human
- name: String - age: int

Mensch
- name: String - alter: int

Bevor es mit der Implementierung weitergeht bietet sich an dieser Stelle eine Wiederholung der wichtigsten drei Diagrammarten an, die in der Oberstufe Verwendung finden:

- Klassendiagramm
- Objektdiagramm
- Sequenzdiagramm

⇒ Exkurs im **Anhang**

```
    public class Human() {
        private String name;
        private int age;

        public Human(String name, int age){
            this.name = name;
            this.age = age;
        }
    }
```

Unsere Menschen haben aktuell nur die Attribute Name und Alter und außer dem Konstruktor sind keine weiteren Methoden vorhanden. Unser Ziel ist es eine Warteschlange der folgenden Struktur zu bauen:

Queue
- queue: Human[]
+ Queue() + push(h: Human): void + pop(): Human

Warteschlange
- warteschlange: Mensch[]
+ Warteschlange() + hintenAnfügen(Mensch): void + vorneEntfernen(): Mensch

```
public class MyListArray(){
    private Human[] queue;
    private int count;

    public MyListArray() {
        queue = new Human[5];
        count = 0;
    }
}
```

Der Konstruktor erzeugt ein neues Feld der Größe 5 und setzt die derzeitige Belegung auf 0. Die Anzahl der Menschen in der Warteschlange muss nicht zwangsläufig in einem Attribut gespeichert werden, da es aber eine Größe ist, die häufiger von Nutzen ist, wäre eine ständige Berechnung unnötiger Aufwand.
Um die Warteschlange auch sinnvoll verwenden zu können müssen weitere Methoden ergänzt werden. Wir beginnen mit der hintenAnfügen() Methode. Da wir spezifizieren müssen, was wir anfügen wollen folgt für die Signatur:

```
public void push(Human humanToAdd)
```

Ein erster - naiver - Ansatz zur Befüllung wäre der folgende:

```
public void push(Human humanToAdd) {
    for(int i = 0; i < queue.length; i++) {
        if(queue[i] == null) {
            queue[i] = humanToAdd;
            break;
        }
    }
}
```

Aufgabe 1: Begründe Sie kurz, warum diese Implementierung ineffizient ist.

Eine effizientere Lösung besteht darin, das Attribut *anzahl* zu verwenden, dadurch vereinfacht sich die Implementierung zu:

```
public class MyListArray(){
    private Human[] queue;
    private int count;

    public MyListArray() {
        queue = new Human[5];
        count = 0;
    }

    public void push(Human humanToAdd) {
        queue[count] = humanToAdd;
        count++;
    }
}
```

Aufgabe 2: Begründen Sie, warum auch diese Implementierung noch problematisch ist. Lösen Sie das Problem!

Aktuell kann unsere Liste nur fünf Elemente enthalten. Versucht man ein sechstes Element anzuhängen, so erhält man als Nachricht **„java.lang.ArrayIndexOutOfBoundsException“**. Es wird also versucht an eine Position der Liste einzufügen, die nicht existiert.
Ist die Liste voll, muss das Feld vergrößert werden. Da dies eine Funktionalität ist, die wir eventuell noch an anderer Stelle brauchen könnten, ist es sinnvoll dafür eine eigene Methode anzulegen. Sie erzeugt ein neues, größeres Feld und kopiert die Einträge aus dem alten Feld in das neue Feld. Anschließend wird neu verzeigert.
Erinnerung: Die Länge eines Feldes ist in einem öffentlichen Attribut des Felds gespeichert, auf das direkt zugegriffen wird, *length* ist deswegen kein Methodenaufruf in untenstehendem Code, sondern bezieht sich auf dieses Attribut. Keine Klammern!

```
public void push(Human humanToAdd) {
    if(count == queue.length) {
        enlargeArray();
    }
    queue[count] = humanToAdd;
    count++;
}

private void enlargeArray() {
    Human[] newQueue = new Human[queue.length + 5];
    for(int i = 0; i < queue.length; i++){
        newQueue[i] = queue[i];
    }
    queue = newQueue;
}
```

Um die Methode effizienter zu gestalten, wird das Feld um 5 Plätze vergrößert. Man möchte zu große Felder vermeiden (insbesondere mit leeren Einträgen!), um nicht unnötig Speicherplatz zu verschwenden (die Implementierung der ArrayList, die von Java direkt angeboten wird verwendet z.B. eine Vergrößerung um 5 Plätze, für Details zu dieser „vorgebauten“ Struktur siehe Java Doc ArrayList).
Als nächstes muss noch das vorderste Element der Warteschlange entfernt werden können. Spätestens hier zeigt sich, dass das Feld für diese Anwendungsform der Liste nur schlecht geeignet ist. Wenn das vorderste Element entfernt wird, müssen alle folgenden Elemente einen Platz nach vorne geschoben werden, ansonsten entsteht mit der Zeit ein großer Block an leeren Einträgen im Feld. Dieser Kopiervorgang nimmt vergleichsweise viel Zeit und Leistung in Anspruch. Eine mögliche Implementierung der Methode sieht folgendermaßen aus:

```
public Human pop() {
    if(queue[0] == null) {
        return null;
    }
    Human toReturn = queue[0];
    for(int i = 0; i < count; i++){
        queue[i] = queue[i+1];
    }
    queue[queue.length-1] = null;
    count--;
    return toReturn;
}
```

Die Methode gibt eine Referenz auf das entfernte Objekt zurück, sofern es existiert. Wird diese Referenz nicht in einer neuen Variable gespeichert bei Anwendung der Methode, so wird sie vom garbage collector aufgeräumt und ist vollständig verschwunden. Alternativ könnte auch eine äquivalente Methode mit der Signatur

```
public void pop()
```

definiert werden. In diesem Fall wird der erste Mensch aus der Schlange entfernt und das Objekt geht „verloren“.
Visualisierung der Anfügen und Entfernen-Methoden: Youtube - push - pop

Übungsblock
Die folgenden Aufgaben steigen grob im Schwierigkeitsgrad an und erweitern die Funktionalität der Liste sukzessive. Die Lösungen finden sich kommentiert auf den darauf folgenden Seiten. Versuchen Sie die beschriebenen Methoden zunächst selbst zu implementieren!

Aufgabe 3: Schreiben Sie eine Methode schreibeListe (printList()), die die Warteschlange in einer sinnvollen Weise auf der Konsole sichtbar macht.
Hinweis: Ergänzen Sie eine passende Methode in der Klasse Mensch (Human)!

Aufgabe 4: Schreiben Sie eine Methode menschAnPosition(int position) (humanAtPosition(int position)), die eine Referenz zum Menschen an der angegebenen Position zurückgibt. Beachten Sie Grenz- bzw. Spezialfälle!

Aufgabe 5: Schreiben Sie eine Methode sucheMenschInSchlange(Mensch m) (searchHumanInQueue(Human h)), die die Position des Menschen in der Schlange zurückgibt.
Hinweis: Überschreiben Sie die allgemeine equals() Methode in der Klasse Mensch.

Aufgabe 6: Schreiben Sie eine Methode enthält(Mensch m) (contains(Human h)), die wahr zurückgibt, wenn der der entsprechende Mensch in der Schlange ist, anderfalls falsch.
Hinweis: Verwenden Sie die equals-Methode aus Aufgabe 5

Aufgabe 7: Schreiben Sie eine Methode entferneAnPosition(int position) (removeAtPosition(int position)), die den Menschen an der gegebenen Stelle entfernt und die übrigen nach vorne rutschen lässt.
Schreiben Sie zwei Versionen dieser Methode, eine, die den Menschen nur entfernt und eine, die eine Referenz zu diesem Menschen zurückgibt.

Aufgabe 8 - für Experten: Schreiben Sie eine Methode zusammenfügen(MeineListeFeld zweiteWarteschlange) (concatenate(MyListArray secondQueue)), die eine zusammengefügte Liste zurückgibt.

Aufgabe 9 - für Experten: Schreiben Sie eine Methode fügeSortiertHinzu(Mensch m) (appendSorted(Human h)), die die Warteschlange nicht von hinten auffüllt, sondern die Menschen an einer bestimmten Stelle einsortiert.
Hinweis: Dazu muss eine Vergleichsmethode in der Klasse Mensch definiert werden. Z.B. könnte nach Alter, oder auch nach Namen oder nach einer Kombination von beidem sortiert werden.

Aufgabe 10 - frei: Überlegen Sie sich eigene, sinnvolle weitere Methoden für die Warteschlangen-Implementierung.

Es folgen die Lösungen zu den obigen Aufgaben mit Anmerkungen.
Zuerst die Mensch-Helferklasse

```
public class Human {
    private String name;
    private int age;

    public Human(String name, int age){
        this.name = name;
        this.age = age;
    }
```

```
    public String getName(){
        return name;
    }

    public int getAge(){
        return age;
    }

    public void presentation(){
        System.out.println("I am " + name + " and I am " + age + " years old");
    }

    public boolean equals(Object o){
         if(this == o) {
             return true;
         }
         if(! (o instanceof Human)){
             return false;
         }
         Human h = (Human) o;
         if(h.getName() == this.name && h.getAge() == this.age){
             return true;
         } else {
             return false;
         }
    }

    public boolean isGreater(Human human) {
        if(age > human.getAge()) {
            return true;
        }
        return false;
    }
}
```

Interessant sind in dieser Klasse nur die letzten beiden Methoden. Alle anderen sind einfache getter, setter oder Ausgabemethoden, die keine weitere spezielle Funktion haben.
Sucht man nach der equals(), also vergleiche() Methode, so findet man häufig den Spezialfall des Stringvergleichs. Dieser kann nur unsicher mit == erfolgen, man muss equals() benutzen.

Aufgabe Exkurs 1: Recherchieren Sie selbstständig, woran das liegt!

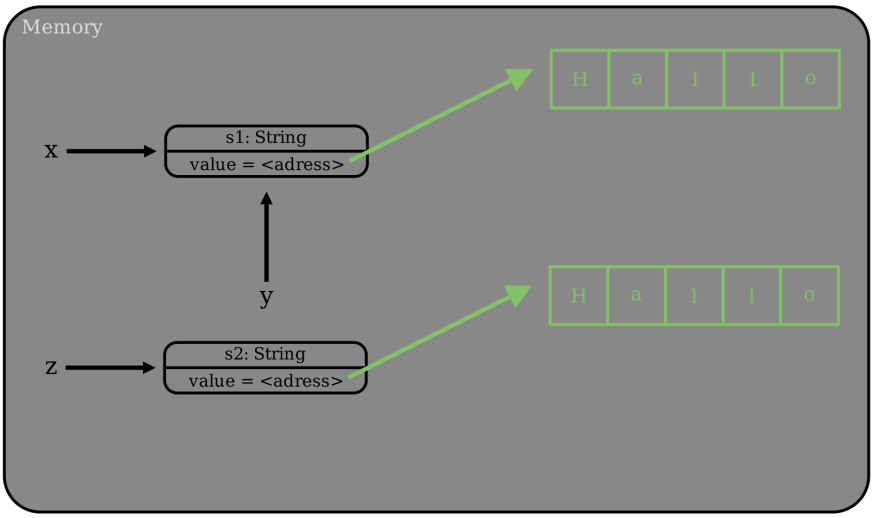
Zur Erklärung folgendes Code-Fragment als Erinnerung:

```
    public static void main(String[] args) {
        String x = "Hallo";
        String y = "Ha"+"llo";
        String z = hallo() + "llo";

        String a = new String("Hallo");

        System.out.println(x==y);
        System.out.println(x==z);
        System.out.println(x==a);
```

```
    }
    public static String hallo() {
        return "Ha";
    }
```

Aufgabe Exkurs 2: Welche Ausgabe erzeugen die drei obigen Vergleiche?

Der erste Vergleich liefert *true*, die anderen beiden dagegen *false*. Das liegt daran, dass ein **String** eine eigene Klasse und kein primitiver Datentyp ist (das erkennt man auch am großen ersten Buchstaben). Ein String ist „intern“ nur ein Feld aus Zeichen, also ein *char*[].
Das bedeutet aber insbesondere, dass die Variablen x, y, z, a in obigem Code nur zu einem String-Objekt im Speicher zeigen und nicht direkt zu einem „Textinhalt“.
Erzeugen wir nun den String x, so sieht dies im Arbeitsspeicher veranschaulicht so aus:



Der Java Compiler kann in einigen Fällen zuordnen, dass das zu erzeugende String-Objekt denselben Inhalt hat, wie ein bereits erzeugtes, z.B. im Fall der beiden Variablen x und y, deswegen zeigen beide auf dasselbe String-Objekt. In anderen Fällen, z.B. wenn man ein neues String-Objekt erzwingt, wie bei der Variable a, oder wenn Funktionsaufrufe involviert sind, so ist dies nicht der Fall. Das == vergleicht aber letztendlich nur, ob die beiden Zeiger der Variablen an dieselbe Adresse zeigen, das ist nur bei x und y der Fall. Die equals()-Methode umgeht dieses Problem, indem sie - im Falle des Strings - das zugrunde liegende Zeichenfeld vergleicht.
Die Definition von equals() reicht tatsächlich aber noch viel tiefer. Blickt man in die Java-Dokumentation für ein Objekt (z.B. Java Docs Object), so wird hier bereits die Funktion mitsamt ihrer erwünschten Eigenschaften definiert. equals() ist außerdem für alle Java-internen Klassen (wie z.B. der String oben!) bereits überschrieben, d.h. der Compiler weiß, wie er zwei Strings vergleichen muss. (Den Hinweis in der Dokumentation, dass auch die hashCode Methode überschrieben werden müsste ignorieren wir aktuell geflissentlich).
Zurück zum Vergleich von Menschen: In unserem Fall würde der Compiler nur die Referenzen zu den beiden Mensch-Objekten vergleichen, da keine weitere Implementierung vorhanden ist. Wir müssen Java also „beibringen“, wann wir zwei Menschen als gleich ansehen. In obigem Fall wurde das Alter und der Name als Vergleich gewählt. Es sind auch andere Implementierungen denkbar. Wichtig ist, dass die Methode vernünftig dokumentiert wird (siehe eigenes pdf), ansonsten könnte diese Vergleichsmethode andere Nutzer der Klasse verwirren.
Die isGreater(), also istGrößer() Methode verwendet dagegen nur das Alter der entsprechenden Person. Wenn wir das Feld später sortieren, werden die Personen unter Verwendung dieser Methode nach dem Alter sortiert.

Betrachten wir noch einmal einige grundlegende Methoden vor den konkreten Aufgabenlösungen in der Warteschlangenklasse:

```java
public class MyListArray {
    private Human[] queue;
    private int count;

    public MyListArray() {
        queue = new Human[5];
        count = 0;
    }

    public void push(Human human) {
        if(count == queue.length) {
            enlargeArray();
        }
        queue[count] = human;
        count++;
    }

    private void enlargeArray() {
        Human[] newQueue = new Human[queue.length + 10];
        for(int i = 0; i < queue.length; i++){
            newQueue[i] = queue[i];
        }
        queue = newQueue;
    }

   public Human pop() {
       if(queue[0] == null) {
           return null;
       }
       Human toReturn = queue[0];
       for(int i = 0; i < queue.length -1; i++){
           if (queue[i+1] == null){
               break;
           }
           queue[i] = queue[i+1];
       }
       queue[queue.length-1] = null;
       count--;
       return toReturn;
   }

   public Human[] getQueue() {
      return queue;
   }

   public int getCount() {
      return count;
    }

    public void setCount(int count) {
        this.count = count;
    }
}
```

Aufgabe 3: Um alle Einträge auszugeben beginnen wir am nullten Eintrag und arbeiten uns bis zum Ende durch, jeweils mit dem Aufruf der dafür in der Klasse Mensch definierten Methode.

```
public void printList() {
    for(int i = 0; i < count; i++){
        queue[i].presentation();
    }
}
```

Aufgabe 4: Die mit Abstand einfachste Aufgabe in dieser Implementierung der Warteschlange. Wir geben nur den entsprechenden Eintrag im Feld zurück. Dabei wird angenommen, dass die erste Position im Feld dem nullten Eintrag entsprechen soll. Auch hier könnte anders programmiert werden, weswegen eine Dokumentation notwendig ist.
Hinweis: Der Mensch wird durch diese Methode nicht aus der Liste entfernt, verändert man den Menschen also mit Hilfe der zurückgegebenen Referenz, so wird also auch die Liste verändert!

```
public Human humanAtPosition(int position) {
    if(position >= 0 && position <= count) {
        return queue[position-1];
    } else {
        return null;
    }
}
```

Aufgabe 5: Auch diese Aufgabe ist in dieser Warteschlange nicht besonders schwer. Die Festlegung, dass das Nicht-Enthalten eines Elements den Rückgabewert -1 erhält ist wiederum nicht zwingend festgelegt, sendet aber dem Benutzer ein deutliches Zeichen, dass kein solches Element vorhanden ist. Diese Methode benötigt außerdem die bereits oben besprochene überschriebene equals()-Methode für die Klasse Mensch, andernfalls wird nie ein Vergleich wahr ergeben.

```
public int searchHumanInQueue(Human human) {
    for(int i = 0; i < count; i++){
        if(queue[i].equals(human)){
            return i;
        }
    }
    return -1;
}
```

Aufgabe 6: Hier ist sowohl eine neue iterative Implementierung möglich, als auch der Rückgriff auf die Methode aus Aufgabe 5. Wenn diese eine Zahl größer als -1 liefert, so muss es einen Index geben, an dem der entsprechende Mensch gefunden werden kann. Andernfalls muss -1 zurückgegeben worden sein.

```
public boolean contains(Human human) {
    for(int i = 0; i < count; i++) {
        if(human.equals(queue[i])) return true;
    }
    return false;
    /* alternatively use searchHumanInQueue:
    if(searchHumanInQueue(human)>=0) return true;
    return false;
    */
}
```

Aufgabe 7: Auch bei dieser Methode wird davon ausgegangen, dass die erste Position in der Schlange dem nullten Eintrag im Feld entspricht. Ansonsten gibt es nur zu beachten, dass der letzte Eintrag in der Schlange händisch auf null gesetzt werden muss, da die Schlange aktuell bis zum letzten Platz besetzt sein könnte. Es wäre also möglich, dass durch die Verschiebung ein Eintrag doppelt vorkommt.

```
public Human removeAtPosition(int position) {
    if(position < 0 || position > count) {
        return null;
    }
    Human toReturn = queue[position - 1];
    for(int i = position; i < queue.length - 1; i++) {
        queue[i] = queue[i+1];
    }
    queue[queue.length-1] = null;
    return toReturn;
}
```

Aufgabe 8: Diese Aufgabe übersteigt das Pensum, das für das Abtiur von Nöten ist, die grundlegende Idee ist aber auch hier nicht schwer. Man analysiert die Listen beider Längen mit Hilfe des zugrunde liegenden Feldes und baut dann ein neues Feld auf. Danach müssen noch alle Einträge aus beiden ursprünglichen Felder hineinkopiert werden.
Da Kopiervorgänge viel Zeit und Speicher beanspruchen ist das Zusammenfügen, wie auch das häufige Einfügen und Löschen von Einträgen ressourcenintensiv. Listen, denen ein Feld zugrunde liegt sind also eher für Einsätze geeignet, bei denen häufig auf die Liste zugegriffen wird, sie aber nicht häufig verändert wird.

```
public Object concatenate(MyListArray toConcat) {
    MyListArray newList = new MyListArray(this.count
        + toConcat.getCount());
    for(int i = 0; i < this.count;i++) {
        newList.getQueue()[i] = queue[i];
    }
    for(int i = 0; i < toConcat.getCount(); i++){
        newList.getQueue()[this.count + i] = toConcat.getQueue()[i];
    }
    newList.setCount(this.count+ toConcat.getCount());
    return newList;
}
```

Aufgabe 9: Eine klassische Lösung dieses Problems besteht darin, das Element einfach anzuhängen und danach das Feld zu sortieren. Sortieralgorithmen sind ein eigenes Kapitel in der Informatik und es gibt zahllose Möglichkeiten zu sortieren. Unten stehend ist ein sogenannter InsertionSort programmiert. Jede beliebige andere Sortierung würde auch funktionieren. Eine visueller (und auditiver) Vergleich verschiedener Sortieralgorithmen findet sich zum Beispiel hier: Sortieralgorithmenmusik
Natürlich müssen die beiden Methoden nicht separiert werden. Ein Äquivalent des InsertionSorts könnte auch direkt in der sortiertEinfügen() Methode verwendet werden. Visualisierung - mit BubbleSort!

```
public void appendSorted(Human human) {
    if(queue[0] == null){
        queue[0] = human;
        count++;
    } else {
        this.push(human);
        insertionSort();
    }
}

private void insertionSort() {
    /*Divides the array in a sorted and unsorted part. It iterates over the
    unsorted part and puts the next element into the right spot in the
    sorted part thus enhancing it by one.
    */
    for(int i = 1; i < count; i++){
```

```
            //The reference to the element that shall be placed next
            Human tmp = queue[i];
            int j = i - 1; //Defines the part of the array that is sorted
            // Moves every element that is greater up one spot until it
            //finds the right spot for the new element.
            //isGreater is necessary as it is not a primitive type.
            while(j >= 0 && queue[j].isGreater(tmp)){
                queue[j+1] = queue[j];
                j--;
            }
            //puts the new element at the found spot
            queue[j+1] = tmp;
        }
    }
```

Zusammenfassung:
Die Implementierung einer Liste mit Hilfe eines Feldes hat sowohl Nachteile als auch Vorteile. Die Implementierung stützt sich im wesentlichen auf ein zugrunde liegendes Feld, dass durch die Methoden der Liste geeignet manipuliert wird.
Vorteile:

1. Das Einfügen eines Elements ist effizient, wenn ausreichend Platz im Feld vorhanden ist.
2. Der Zugriff auf ein beliebiges Element der Liste ist direkt möglich.

Nachteile:

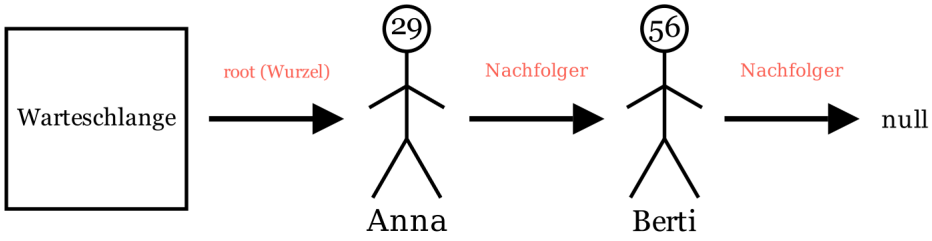
1. Das Entfernen eines Elementes zieht durch das Kopieren aller folgenden Einträge nach vorne einen großen Rechenaufwand nach sich.
2. Wird das Feld zu groß oder zu klein gewählt wird gegebenenfalls Speicherplatz verschwendet (viele Null-Referenzen im Feld) und es gibt keinen Platz mehr im Arbeitsspeicher oder das Feld muss häufig vergrößert werden.
3. Je öfter diese dynamische Größenveränderung angewandt werden muss, desto größer ist neben dem Speicher auch der Rechenaufwand.

Fazit: Eine Feld-Liste bietet sich vor allem dann, wenn einmal viele Elemente eingefügt werden sollen, anschließend aber hauptsächlich auf die Elemente zugegriffen wird, ohne viele Veränderungen vorzunehmen.

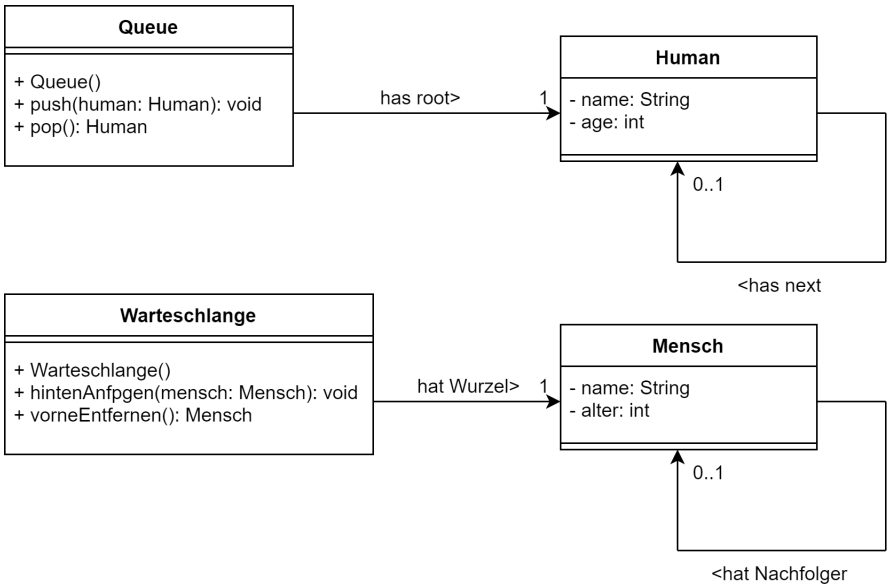
3 Die verkettete Liste - Version 1

3.1 Grundlagen

Das Feld ist nicht die einzige Möglichkeit eine Liste zu implementieren. Die sogenannte verkettete Liste ist ebenfalls ein übliches Konzept.
Die **grundlegende Idee** ist dabei folgende:



Die Klasse Warteschlange organisiert unseren Zugriff und dient als Steuerzentrale für alle Methoden, die wir auf der Liste ausführen wollen. Die eigentlichen Listenelemente sind jetzt aber nicht mehr in einem Feld gespeichert, sondern sind eigene Objekte, die jeweils nur ihren Nachfolger kennen. Nicht einmal die Warteschlange selbst kennt alle Menschen (oder auch nur deren Zahl). Sie kann nur auf den ersten in der Schlange zugreifen, die Wurzel (oder auch Anfang der Liste, im Englischen: root). Diese Idee scheint auf den ersten Blick umständlich, denn man kann nun nicht mehr auf ein beliebiges Listenelement zugreifen.
Tatsächlich hat diese verkettete Struktur aber andere Vorteile, die später noch deutlich werden. Ein offensichtlicher Vorteil ist aber die **Erweiterbarkeit**. Unsere Warteschlange kann jetzt nicht mehr voll laufen (außer unser Speicher ist voll), sondern wir können einfach weitere Menschen hinten anfügen, indem wir dem letzten Listenelement (hier der Mensch Berti) eine Referenz auf seinen neuen Nachfolger geben.
Hinweis: Sieht man sich die Struktur genau an, so wird klar, dass Berti nicht einmal von der Existenz Annas weiß, d.h. von seiner Vorgängerin. Dies nennt man ein **einfach verkettete** Liste. Werden Referenzen in beide Richtung gesetzt, so ist es eine **doppelt verkettete** Liste. Oft genügt aber die Referenz in eine Richtung. Zusammengefasst in einem Klassendiagramm lautet unser aktuelles Ziel also:



Bevor wir an der Implementierung arbeiten können muss zunächst ein neuer Begriff geklärt werden, die **Rekursion**.

3.2 Rekursiv vs. Iterativ

In der bisherigen Programmierung wurden fast ausschließlich **iterative** Algorithmen besprochen. Wollten wir alle Einträge eines Feldes durchlaufen schreiben wir z.B.:

```
for(int i = 0; i < array.length) {
    System.out.println(i);
}
```

Wir „zählen“ von 0 bis zur Länge des Feldes hoch und arbeiten schrittweise weiter. In vielen Fällen ist eine iterative Arbeits- und Denkweise sinnvoll und entspricht auch unserer natürlichen Denkweise.
Sollen alle Zahlen von 1 bis 5000 aufsummiert werden, so können wir schreiben:

```
result = 0;
for(int i = 1; i <= 5000; i++) {
    result += i;
}
```

Typischerweise wird die Nützlichkeit der Rekursion mit Hilfe der Fibonacci-Folge (Wikipedia Fibonacci) verdeutlicht. Es ist eine Folge von Zahlen, die mit zwei Einsen beginnt und dann immer mit den letzten beiden Zahlen das nächste Element der Folge berechnet, es gilt also:

$$f_n = f_{n-1} + f_{n-2} \text{ für } n \geq 3$$

Damit ergibt sich die Folge:

$$1, 1, 2, 3, 5, 8, 13, \ldots, \text{ da z.B. } 13 = 8 + 5$$

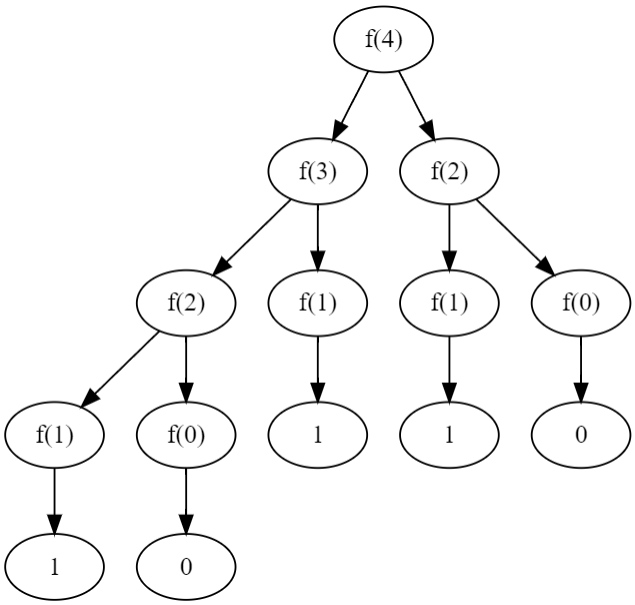
Ein iterativer Ansatz zur Berechnung der $n-$ten Zahl funktioniert mit Hilfe zweier Variablen, die die jeweils zwei letzten Ziffern speichern und daraus die neue berechnen:

```
public int fibonacci(int n) {
    if (n == 0)
        return 0;
    if (n == 1 || n == 2)
        return 1;

    int previous = 1;
    int result = 1;

    for (int i = 2; i < n; i++) {
        int sum = result + previous;
        previous = result;
        result = sum;
    }
    return result;
}
```

Dieser Code liefert uns die korrekten Ergebnisse. Fassen wir Fibonacci jedoch als Funktion auf, so könnte eine gewisse Systematik auffallen. Sei f die abschnittsweise definierte Funktion, die die Fibonacci-Folge bauen soll, dann gilt:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2) \text{ für } n \geq 3 \end{aligned}$$

Veranschaulicht man diese Vorschrift z.B. für $f(4)$, so erhält man:



Zählt man nun die Anzahl der Einsen in der letzten Zeile zusammen, so kommt man auf die letztendliche Lösung. Aber worauf basiert dieser Lösungsansatz?
Sieht man sich die Pyramide genauer an, so lässt sich erkennen, dass das Problem (die Berechnung von $f(4)$) schrittweise aufgeteilt wird. Dies geht nach bekannter Vorschrift immer so weiter, bis man bei einem der **Basisfälle** ankommt. Die Basisfälle sind die einfachst möglichen Anwendungen der Regel. In diesem Fall die Funktionswerte der 0 und der 1.
Zusammengefasst könnte man sagen, dass die **Idee** der Rekursion darin besteht, das Problem in kleine, überschaubare und leicht zu lösende Teil **aufzuteilen** und anschließend die Ergebnisse wieder zusammenzutragen.
Im Sinne der Informatik lässt sich noch eine alternative Formulierung aufstellen.

Definition 1 (Rekursion)**.** *Eine Methode wird als* ***rekursiv*** *bezeichnet, wenn sie sich selbst direkt oder indirekt aufruft.*

Beispiel:

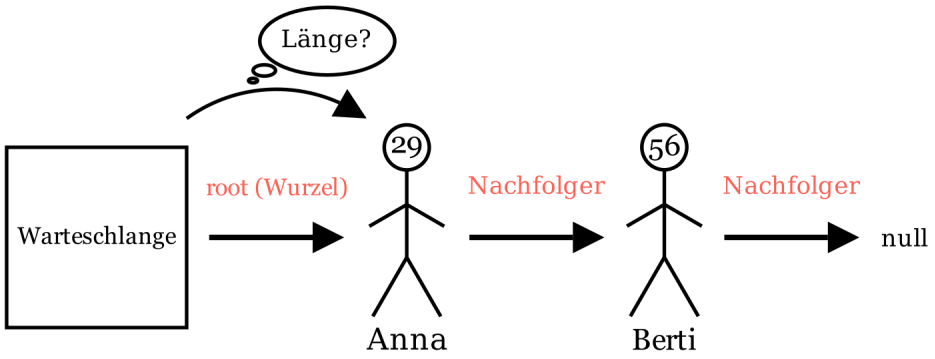
```
public void magic(int i){
    System.out.println("I was called " + i + " times");
    magic(i+1);
}
```

Führt man obigen Code aus, so erzeugt man natürlich eine Endlosschleife, bis irgendwann der Speicher vollgelaufen ist (oder der Compiler abbricht, weil er die Schleife erkannt hat).
Wichtig bei einer Rekursion ist es also eine **Abbruchbedingung** zu haben. Im obigen Fall könnten wir die Funktion nur 100 mal aufrufen wollen:

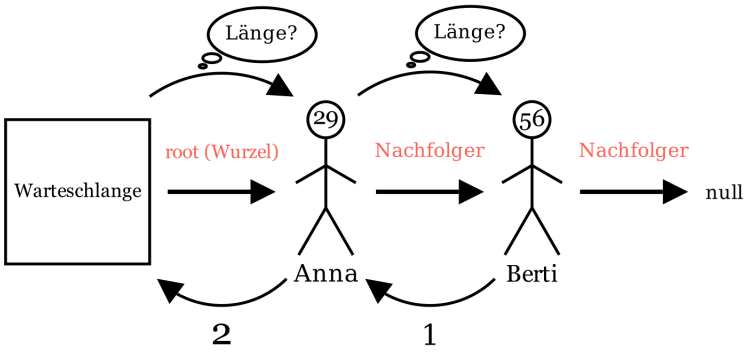
```
public void magic(int i){
    if(i > 100) {
        return;
    }
    System.out.println("I was called " + i + " times");
    magic(i+1);
}
```

Wir beenden mit dem return (Zurückgehen) die Kette der Methodenaufrufe, wenn der Übergabeparameter i größer als 100 ist.
Zurück zu unserer Warteschlange. Die Tatsache, dass diese als **rekursive** Struktur gedacht ist sieht man schon am Klassendiagramm. Die Objekte der Klasse Mensch referenzieren jeweils ein Objekt derselben Klasse, das bedeutet

aber, dass es keine Möglichkeit für die Warteschlange gibt mit einem iterativen Ansatz durch alle Menschen abzuarbeiten, wie das beim Feld-Ansatz der Fall war. Sie kann lediglich mit dem ersten Menschen in der Schlange interagieren.



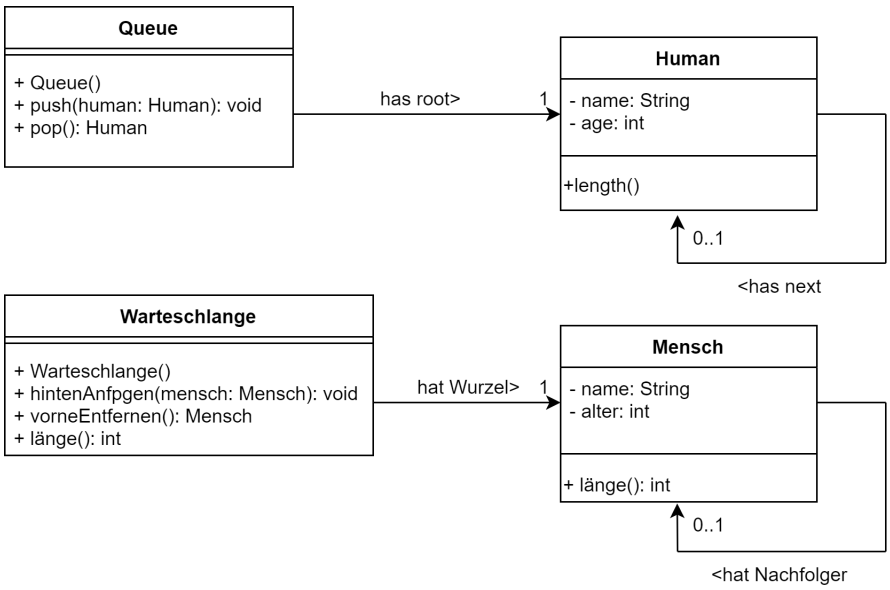
Möchten wir beispielsweise die Länge der Warteschlange wissen, so rufen wir eine Methode länge() auf der Warteschlange auf, die wiederum eine Methode länge() auf dem ersten Menschen in der Schlange - in diesem Fall Anna - aufruft.
Anna allein kann die Länge aber ebenfalls nicht bestimmen, sie muss also Berti fragen und dazu (mithilfe ihrer Referenz zu Berti) auch die Methode länge() aufrufen.
Hinweis: Hier kommt die Rekursion ins Spiel, wir rufen zwar nicht immer wieder auf demselben Objekt die Methode auf, allerdings wird die Methode länge() dennoch immer wieder aufgerufen!
Berti kann abschließend niemanden mehr fragen, denn er hat keinen Nachfolger, er antwortet also stattdessen mit 1 an Anna. Sie wiederum zählt noch eins hinzu und gibt die - hier finale - Antwort 2 an die Warteschlange zurück. Anschaulich gesprochen wurde also „von hinten“ durchgezählt.



Aufgabe Rekursive Algorithmen:
a) Schreiben Sie eine rekursive Methode, die den größten gemeinsamen Teiler zweier natürlicher Zahlen bestimmt.
b) Schreiben Sie eine rekursive Methode, die das Problem der „Türme von Hanoi“ löst Türme von Hanoi. Gehen Sie von einem Turm der Höhe 5 aus.

3.3 Implementierung

Die konkrete Implementierung dieser Methode folgt in einem späteren Kapitel. Zunächst muss das Grundgerüst geschaffen werden. Zur Erinnerung das Klassendiagramm, ergänzt um die Methode länge(), die wir zusätzlich implementieren wollen.
Hinweis: Die Methode länge() steht sowohl in der Klasse Warteschlange, als auch in der Klasse Mensch, da sie in beiden gebraucht wird. Das beide denselben Namen haben ist nicht zwingend notwendig, der Übersichtlichkeit halber aber von Vorteil.



Wir beginnen mit den Attributen und Konstruktoren:

```
public class MyListLinked {
    private Human root;

    public MyListLinked() {
        root = null;
    }
}

public class Human() {
    private String name;
    private int age;
    private Human next;

    public Human(String name, int age) {
        this.age = age;
        this.name = name;
        next = null;
    }
}
```

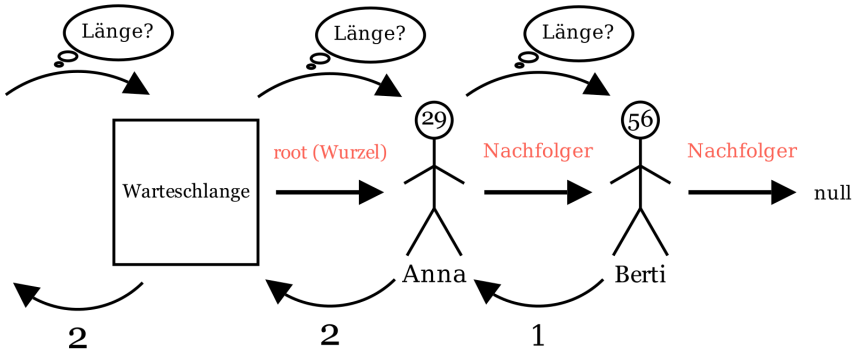
Hinweis: Die explizite Definition von root und next als null ist nicht zwingend notwendig, erinnert aber daran, dass diese Attribute bei der Definition diesen Wert haben.
Bevor wir die Methoden zum Anfügen und Entfernen schreiben beginnen wir mit der Länge der Liste, um die Rekursion an einem typischen Beispiel besser zu verstehen, wir nehmen also an, wir hätten eine Liste, deren Länge wir prüfen können.
In der Klasse der Warteschlange selbst ist die Methode sehr einfach:

```
//class MyListLinked
public int length(){
    if(root != null) {
        return root.length();
    } else {
        return 0;
    }
}
```

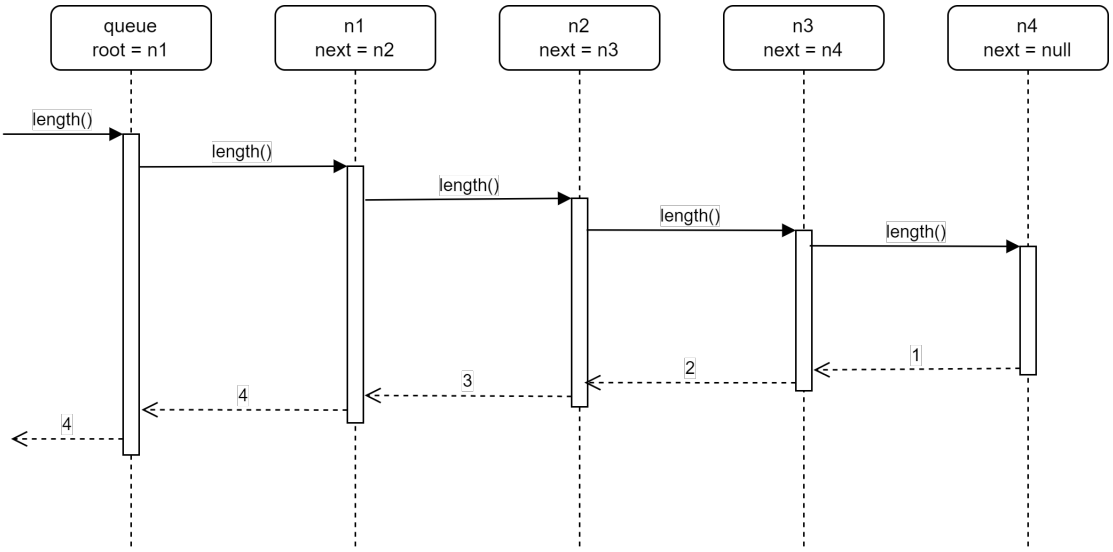
Wir rufen auf der uns bekannten Wurzel die Methode länge() auf und geben das Ergebnis zurück. Der interessante Teil beginnt in der Klasse Mensch:

```
//class Human
public int length() {
    if(next == null) {
        return 1;
    } else {
        return next.length() + 1;
    }
}
```

Die erste Bedingung entspricht der Frage, ob es einen Nachfolger in der Schlange gibt. Ist das nicht der Fall, da unter der Referenz des Nachfolgers nur null zu finden ist, so „weiß“ der Mensch, dass er der letzte in der Schlange ist und gibt als Antwort an den vorletzten Menschen 1 zurück (via return 1).
Der vorletzte Menschen (in unserem Beispiel weiter oben Anna) hatte aber nicht *null* als Nachfolger. Deswegen wurde die Methode länge() auf dem Nachfolger aufgerufen, dieser Methodenaufruf hat 1 zurückgegeben. Zu dieser 1 wurde eine weitere 1 hinzuaddiert und dann ebenfalls zurückgegeben.
Damit ist der länge()-Aufruf auf unserer Wurzel (Anna) beendet und die Warteschlange kann die Antwort (2) ebenfalls zurückgeben. Noch einmal bildlich:



Ein wenig formeller mit vier Menschen und in einem Sequenzdiagramm dargestellt:



Mit dieser Vorarbeit sollten die anderen beiden Methoden kein Problem mehr darstellen. Für die hintenAnfügen() Methode müssen wir nicht einmal einen Rückgabewert (und damit eine Abbruchbedingung) angeben. Wir wenden dieselbe Logik wie zuvor an:

```
//class MyListLinked
public void push(Human human) {
    if(root == null) {
        root = human;
    } else {
        root.push(human);
    }
}
```

In der Warteschlangen-Klasse wird wieder nur auf der Wurzel die hintenAnfügen() Methode ausgeführt, sofern diese existiert, andernfalls ist der übergebene Mensch die neue Wurzel.
Die Implementierung in der Klasse Mensch dagegen sieht fast völlig identisch aus, wir prüfen wieder, ob der nächste Mensch vorhanden ist, falls nein (*null*), so setzen wir den übergebenen Menschen als Nachfolger, andernfalls wird auf dem Nachfolger rekursiv die gleiche Methode aufgerufen, bis wir den letzten in der Reihe gefunden haben:

```
// class Human
public void push(Human human) {
    if(next == null) {
        next = human;
    } else {
        next.push(human);
    }
}
```

Aufgabe 1: Begründen Sie, ob die Methode vorneEntfernen() rekursiv sein muss oder nicht. Implementieren Sie sie anschließend.

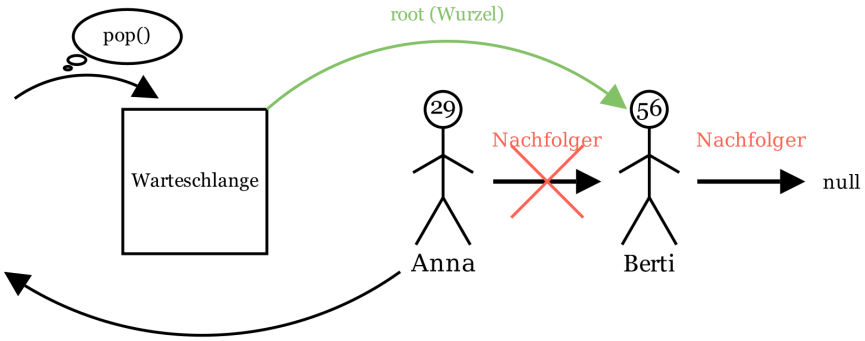
Die Methode muss nicht rekursiv sein. Es wird nur das vorderste Element entfernt, das bedeutet, dass nur das zweite Element als neue Wurzel gesetzt werden muss.
Problem: Aktuell können wir auf die Nachfolger-Referenz des ersten in der Schlange nicht zugreifen, da es privat implementiert ist. Wir brauchen also eine getter-Methode für den Nachfolger, ein entsprechender setter kann ebenfalls nicht schaden:

```
//class Human
public Human getNext() {
    return next;
}

public void setNext(Human human) {
    next = human;
}
```

Bei der Implementierung der vorneEntfernen() Methode muss noch darauf geachtet werden, dass die Warteschlange leer sein könnte:

```
//class MyListLinked
public Human pop() {
    if(root == null) {
        return null;
    } else {
        Human toReturn = root;
        root = root.getNext();
        toReturn.setNext(null);
        return toReturn;
    }
}
```

Graphisch dargestellt (als Bild und hier im Video):



Will man das zurückgegebene Objekt vom Typ Mensch weiter verwenden, so sollte die Referenz auf den Nachfolger dafür auf null gesetzt werden, da sonst die Liste nach der Entfernung weiter verändert werden könnte.
Die Methode muss allerdings nicht zwingend eine Referenz zur ehemaligen Wurzel zurückgeben. Lässt man die Rückgabe weg, so wird der garbage collector aktiv und entfernt das entsprechende Objekt.
Würde man die Null-Abfrage nicht schreiben, so gäbe es eine **null-pointer-exception**, wenn die Warteschlange leer ist. Das betreffende Program, dass die Anfrage stellt stürzt also ab.
Null-pointer-exceptions gehören zu den häufigsten Fehlern in objektorientierten Sprachen, die das Konzept der null enthalten.
Merke: Beim Entwickeln von Methoden müssen alle möglichen Spezialfälle (insbesondere der Umgang mit Null-Referenzen) beachtet und entsprechend implementiert werden.

Versuchen Sie die folgenden Aufgaben zunächst selbstständig zu lösen, die Lösungen mit Erläuterungen finden sich dann auf den nächsten Seiten.

Aufgabe 2: Implementieren Sie eine Methode nachHintenSetzen() (moveToBack()), die die Wurzel entfernt und an die letzte Stelle setzt.

Aufgabe 3 - für Experten: Erweitern Sie die Methode aus Aufgabe 2 so, dass die Position in der Warteschlange, an die die Wurzel geschoben wird, gewählt werden kann (moveBack(int position)).
Hinweis: Die Methode soll **nicht** um die entsprechende Position nach hinten gerückt werden sondern **an** diese Position in der Schlange gestellt werden.

Aufgabe 4: Schreiben Sie eine Methode schreibeListe() (printList), die die Warteschlange in einer sinnvollen Weise auf der Konsole sichtbar macht. Formatieren Sie die Ausgabe ansprechend.
Hinweis: Ergänzen Sie eine passende Methode in der Klasse Mensch (Human)!

Aufgabe 5: Verändern Sie die Methode aus Aufgabe 4 so, dass die Ausgabe nicht mehr auf die Konsole geschrieben wird, sondern in einem String gespeichert wird, der am Ende zurückgegeben werden soll.

Aufgabe 6 - für Experten: Schreiben Sie eine Methode sucheMenschInSchlange(Mensch m) (searchHumanInQueue(Human h)), die die Position des Menschen in der Schlange zurückgibt.
Hinweis: Überschreiben Sie die allgemeine equals() Methode in der Klasse Mensch, wie bei der Feldimplementierung.

Aufgabe 7: Schreiben Sie eine Methode enthält(Mensch m) (contains(Human h)), die wahr zurückgibt, wenn der der entsprechende Mensch in der Schlange ist, anderfalls falsch.
Hinweis: Verwenden Sie wieder die equals-Methode aus Aufgabe 6.

Aufgabe 8 - für Experten: Schreiben Sie eine Methode entferneAnPosition(int position) (removeAtPosition(int position)), die den Menschen an der gegebenen Stelle entfernt. Es soll eine Referenz zu diesem Menschen zurückgegeben werden.

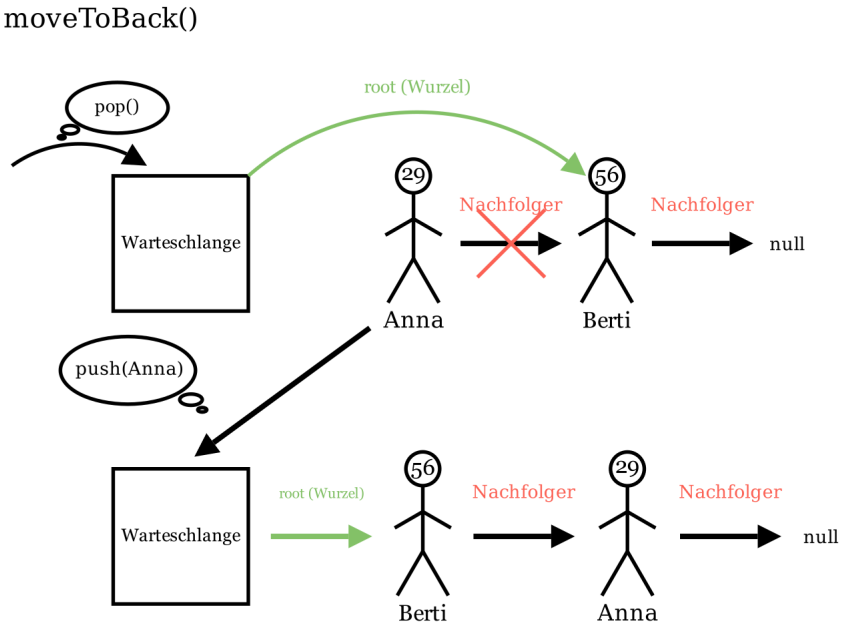
Aufgabe 9: Schreiben Sie eine Methode zusammenfügen(MeineVerketteteListe zweiteWarteschlange) (concatenate(MyListLinked secondQueue)), die eine zusammengefügte Liste zurückgibt.

Aufgabe 10 - frei: Überlegen Sie sich eigene, sinnvolle weitere Methoden für die Warteschlangen-Implementierung.

Lösungen zu den Aufgaben)
Aufgabe 2: Mit unseren Vorarbeiten (insbesondere das Entfernen der Nachfolger-Referenz der Wurzel) ist diese Aufgabe sehr leicht. Die Wurzel die entfernt wird, wird mit der normalen hintenAnfügen() Methode wieder eignefügt.

```
//class Human
public void setNext(Human human) {
    next human;
}

public void moveToBack(){
    this.push(this.pop());
}
```

Graphisch veranschaulicht:



Aufgabe 3: Für diese Aufgabe müssen wir wieder rekursiv denken. Wir können der Warteschlange bzw. der Wurzel nicht direkt sagen, zwischen welche beiden Listenelemente die alte Wurzel gesetzt werden muss, da die Wurzel nur die neue Wurzel und diese nur ihren 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 die neue Wurzel. Eine negative Eingabe oder 0 macht ebenfalls keinen Sinn.
Damit sind die Vorbereitungen abgeschlossen und die alte Wurzel kannt entfernt und in einer lokalen (temporären tmp) Variable zwischengespeichert werden. Danach wird die entsprechende Funktion auf der Wurzel 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();
    root.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(root == null){
        System.out.println("No list here to print!");
    } else {
        root.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(root == null){
        System.out.println("No list here to print!");
        return "";
    } else {
        return root.printList();
    }
}
```

```
// 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.

```
//class MyListLinked
public int searchHumanInQueue(Human human) {
    int counter = 0;
    int searched = root.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.

```
//class MyListLinked
public boolean contains(Human human) {
    if(root != null) {
        return root.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(root != null) {
            int counter = 1;
            return root.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);
        }
    }
```

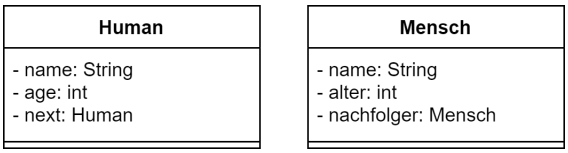
Aufgabe 9: Im Gegensatz zur Feld-Implementierung ist das Zusammenfügen zweier Listen mit der verketteten Struktur denkbar einfach. Es muss lediglich das letzte Element der ersten Liste gefunden werden (hier über eine Helfer-Methode, die die Liste durchläuft und immer prüft, ob das nächste Element gleich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

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

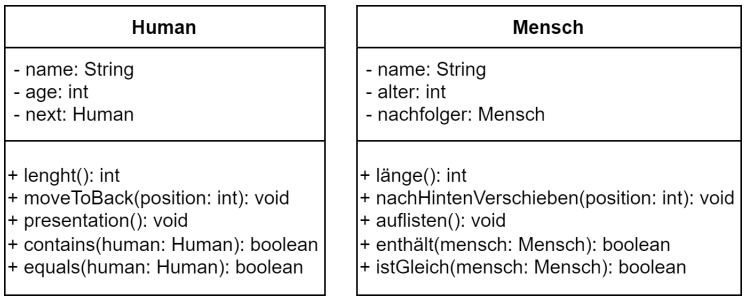
```
    }
```

4 Die verkettete Liste - Version 2

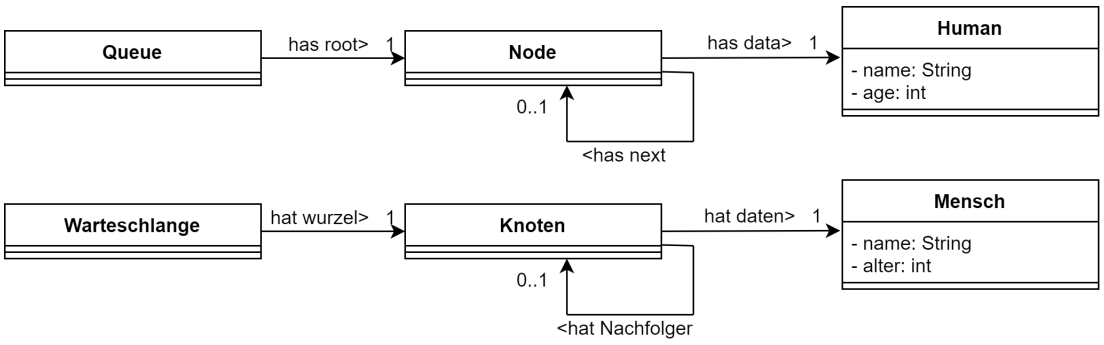
Die verkettete Liste erfüllt zwar alle Funktionalitäten, die wir von ihr erwarten, sieht man sich die Struktur aber genauer an, so erfüllt die Klasse Mensch eigentlich zwei Rollen.



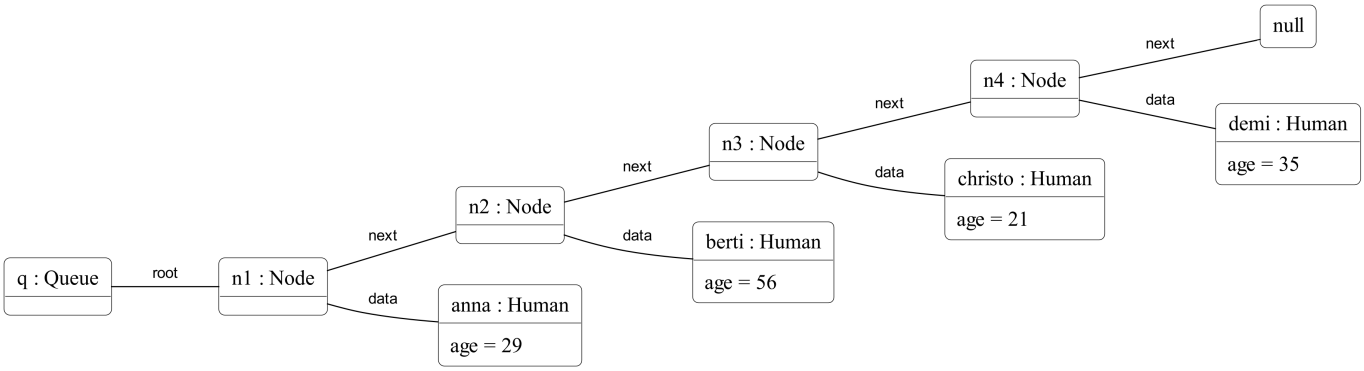
Die ersten beiden Attribute beschreiben die eigentlichen Daten, die in dieser Liste gespeichert sind, dagegen ist das letzte Attribut für jemanden, der nur an den Menschen in der Schlange interessiert ist, irrelevant.
Das Problem wird noch deutlicher, nimmt man die Methoden hinzu:



Manche der Methoden sind nur notwendig, um die Liste zu durchlaufen und bestimmte Aktionen auszuführen - wie z.B. die Bestimmung der Länge. Welche Daten über den Menschen gespeichert sind, ist dabei völlig irrelevant.
Andere Methoden dagegen beziehen sich nur auf die Daten an sich, z.B. die istGleich(Mensch m) - Methode, die zwei Menschen miteinander vergleicht.
Diese Vermischung zweier unterschiedlicher sollte aufgelöst werden, d.h. eine Struktur geschaffen werden, die die Funktionalität der Liste enthält. In der Klasse Mensch verbleiben dann nur noch die Attribute und Methoden, die direkt mit den Daten zu tun haben. Auf diese Weise lässt sich die Liste auch leichter an neue Bedingungen anpassen. Wollen wir statt der Menschen eine Liste von Tieren anlegen, dann muss lediglich die Klasse Mensch und die Referenzen auf die Daten ausgetauscht werden. Die Liste an sich besteht dann aus ihrem „Kopf“, in unserem Fall die Warteschlange und mehrere Knoten. Jeder Knoten kennt dabei wieder seinen Nachfolger und hält eine Referenz zu einem Objekt der Klasse Mensch (in unserem Fall). Ohne detailliert auf die Methoden einzugehen sieht die Struktur also wie folgt aus:



Die Klasse Knoten hat sich zwischen die Warteschlange und die „Datenklasse“ Mensch geschoben. Alle Methoden, die wir bisher implementiert haben, lassen sich eindeutig einer der beiden Klassen zuordnen. In der Klasse Warteschlange ändert sich außer dem Namen der referenzierten Klasse bei keiner Methode etwas inhaltlich.
Angenommen wir betrachten eine Liste, die vier Menschen enthält, dann sähe ein Objektdiagramm der aktuellen Liste wie folgt aus:



Das Gerüst der drei Klassen sieht dann wie folgt aus:

```
public class Queue {
    private Node human;

    public Queue() {
        human = null;
    }
}

public class Node {
    private Node next;
    private Human data;

    public Node(Human h) {
        next = null;
        data = h;
    }
}

public class Human {
    private int age;
    private String name;

    public Human(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Aufgabe 1: Implementieren Sie zur Übung die Methoden hintenEinfügen(Mensch m), vorneEntfernen() und auflisten() (bzw. push(Human h), pop() und printList()) für die neu definierte obige Struktur.

Wir könnten jetzt alle weiteren bisherigen Methoden auch für diese Modellierung implementieren. Es gibt jedoch vorher noch zwei Verbesserungen, die erste betrifft die Objekte, die wir an unsere Liste anhängen können.
Bisher können wir nur Elemente einer bestimmten Klasse an unsere Liste anhängen (dies entspricht auch den üblichen Implementierungen der von Java vorgegebenen Listentypen). Soll unsere Liste etwas flexibler werden -

d.h., wir wollen Objekte verschiedener Klassen anhängen - so müssen wir sicherstellen, dass alle Klassen alle Methoden enthalten, die für die Liste notwendig sind. So muss beispielsweise jede Klasse eine istGleich() - Methode implementieren und die gespeicherten Daten auf Anfrage zurückgeben.
Um all dies zu realisieren bietet es sich an ein **Interface** zu verwenden.

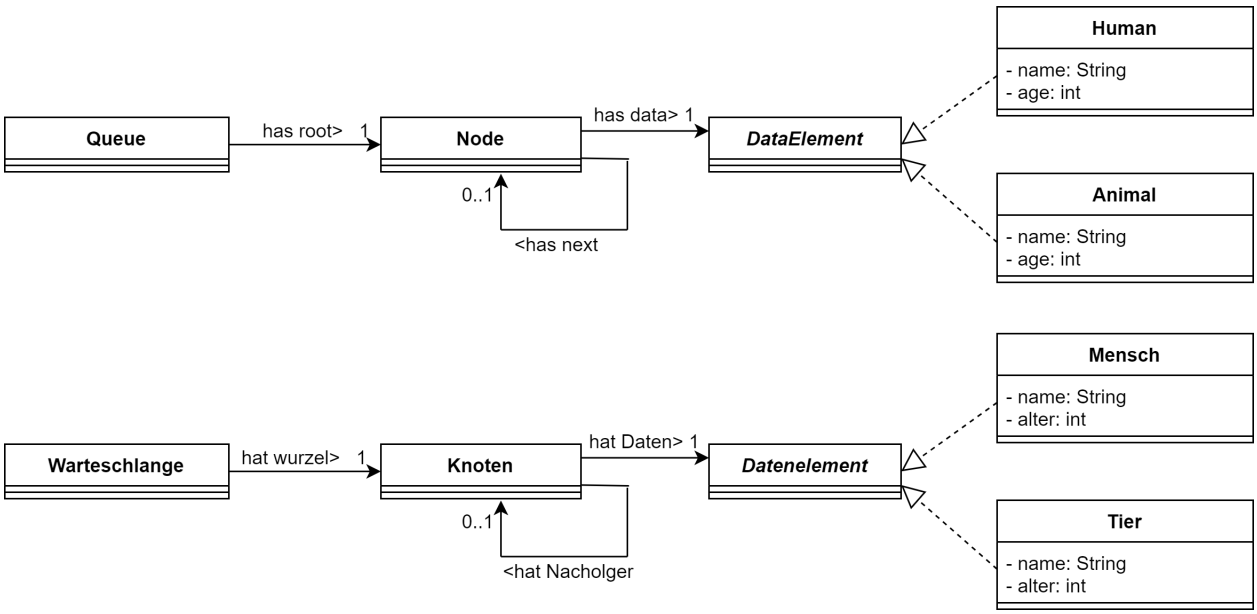
Einschub: Interface
Neben einer (abstrakten) Oberklasse ist das Interface die zweite Möglichkeit in Java „allgemeines“ Verhalten zu definieren. Im Interface werden nur Signaturen von Methoden definiert (und idealerweise kommentiert), eine Klasse die dieses Interface implementiert, muss dann diese Methoden überschreiben und mit konkretem Inhalt füllen. Ein einfaches Beispiel sind (mal wieder) Tiere.

```
public interface AnimalMethods {
/**
* Prints the typical sound of this animal onto the console.
*/
    public void animalSound();
}

public class Dog implements AnimalMethods {

    @Override
    public void animalSound(){
        System.out.println("Woof!");
    }
}

public class Cat implements AnimalMethods {

    @Override
    public void animalSound() {
        System.out.println("Meow!");
    }
}
```

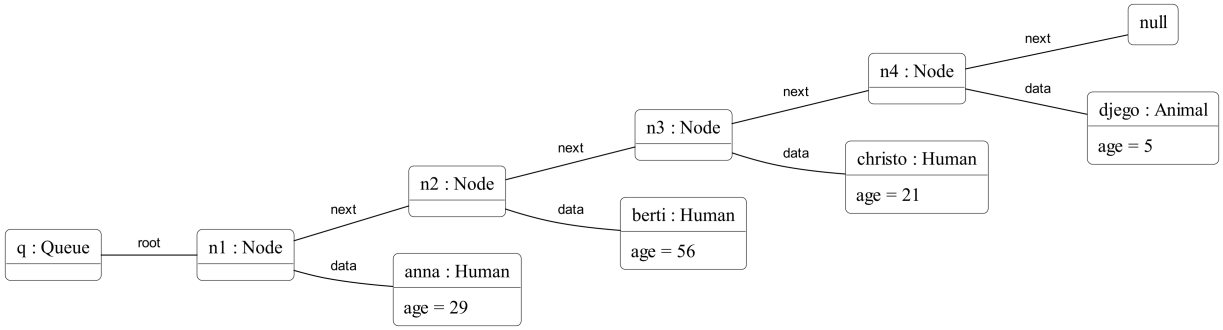
Sowohl die Klasse Hund als auch die Klasse Katze implementieren das Interface und überschreiben die Methode geräuschMachen(). Es gibt dabei zwei große relevante Unterschiede zur abstrakten Klasse:

1. Eine Klasse kann beliebig viele Interfaces implementieren, aber nur von einer Oberklasse erben.
2. Die Methoden müssen überschrieben werden und können nicht im Interface vordefiniert werden. Wird das Schlüsselwort „implements“ verwendet erinnern gute IDEs auch daran mit dem Fehler: „The type $\cdots$ must implement the inherited abstract method $\cdots$“. Es sind also alle Methoden des Interface abstract, auch wenn das entsprechende Schlüsselwort nicht dabeisteht.

Hinweis: Die Anmerkung @*Override* ist nicht zwingend nötig, wird aber von vielen IDEs automatisch erstellt, um deutlich zu machen, dass es sich um eine aus einer Oberklasse oder einem Interface überschriebene Methode handelt.
Zusammengefasst kann man sagen, dass mit einem Interface von einer Klasse eine bestimmte Funktionalität gefordert werden kann, ohne dass die Implementierungsdetails vorgegeben sind.
Zurück zu unserem Anwendungsfall: Wir wollen an unsere Liste nicht mehr nur Menschen anhängen, sondern alle Objekte von Klassen, die unseren Anforderungen an ein Datenelement genügen. Wir definieren also ein Interface Datenelement, in dem wir alle Methoden fordern, die wir benötigen. Wollen wir beispielsweise auch Tiere in unserer Warteschlange erlauben, so sähe die gewünschte Struktur so aus:



Betrachten wir die Warteschlange zu einem beliebigen Zeitpunkt, dann ändert sich nicht viel:



Die grundlegende Struktur sieht dann wie folgt aus:

```
public class Queue {
    private Node human;

    public Queue() {
        human = null;
    }
}

public class Node {
    private Node next;
    private DataElement data;

    public Node(DataElement data) {
        next = null;
        this.data = data;
    }
}

public interface DataElement {
    ...
}
```

```
public class Human implements DataElement{
    private int age;
    private String name;

    public Human(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

public class Animal implements DataElement{
    private int age;
    private String name;

    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Aufgabe 2: Implementieren Sie die Methoden hintenEinfügen(Mensch m), vorneEntfernen() und auflisten() (bzw. push(Human h), pop() und printList()) für die neu definierte obige Struktur. Beachten Sie dabei, dass in der Klasse Knoten auch ein Datenelement als Klasse gefordert werden kann, auch wenn es sich um ein Interface handelt. Letztendlich verlangt diese Syntax, dass in der Variable daten ein Objekt einer Klasse abgelegt wird, die das Interface Datenelement implementiert.

Im Folgenden finden sich die Lösungen für alle Methoden auf einmal, jeweils klassenweise dargestellt. Zuerst die derzeitige Warteschlange (die Bezeichner für die Liste werden inzwischen sehr lang, sollten mehrere Implementierungen im selben Java-Paket existieren ist diese Unterscheidung aber dringend notwendig!):

```
public class MyListLinkedNode {
    public MyListLinkedNode(){
        root = null;
    }

   public void push(DataElement data){
       Node node = new Node(data);
       if(root == null) {
           root = node;
       } else {
           root.push(node);
       }
   }

   @Override
   public DataElement pop(){
       if(root == null){
           return null;
       } else {
           DataElement toReturn = root.getData();
           root = root.getNext();
           return toReturn;
       }
   }
```

```
        @Override
        public void printList() {
            if(root == null){
                System.out.println("No list here to print!");
            } else {
                root.printList();
            }
        }
    }
```

Die einzige Änderung zu den vorhergehenden Implementierungen sind die Ersetzungen der Klasse Mensch durch das Interface Datenelement und die Verwendung der gibDaten() Funktion, da die Daten nicht mehr direkt im Knotengespeichert sind, sondern dieser auch erst auf das Datenobjekt zugreifen müsste. In dieser Implementierung wird das gesamte Datenelement zurückgegeben, damit ggf. danach weiter damit gearbeitet werden kann. Möchte man noch mehr Kontrolle über die Daten haben, so könnte man im folgenden Interface ebenfalls eine String gibDaten() Funktion fordern, die die Daten im Element formatiert und als String zurückgibt.

```
    public interface DataElement {
        public void presentation();
    }
```

Die Klasse Mensch verändert sich nicht wesentlich:

```
public class Human extends DataElement{
    private String name;
    private int age;

    public Human(String name, int age){
        this.name = name;
        this.age = age;
    }

    @Override
    public void presentation(){
        System.out.println("I am " + name + " and I am " + age + " years old");
    }

    @Override
    public boolean equals(Object o){
        if(this == o) {
            return true;
        }
        if(! (o instanceof Human)){
            return false;
        }
        Human h = (Human) o;
        if(h.getName() == this.name && h.getAge() == this.age){
            return true;
        } else {
            return false;
        }
    }
}
```

Auf die getter- und setter-Methoden wurde aus Platzgründen verzichtet.
Es folgt die Klasse Knoten, die die rekursiven Aufrufe der Hauptklasse ausführt:

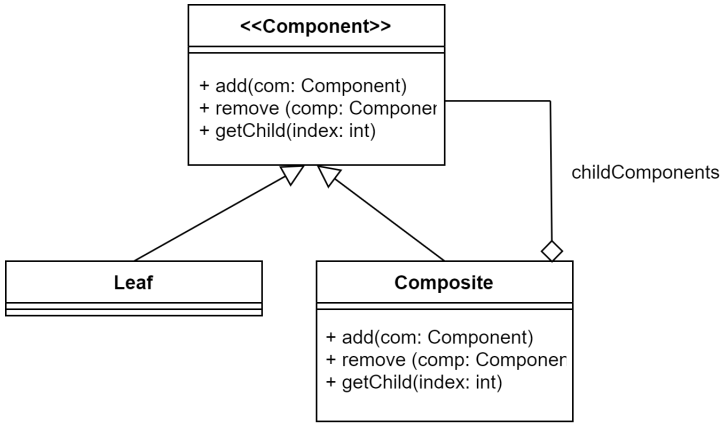
```
public class Node {
    private Node next;
    private DataElement data;

    public Node(DataElement data){
        next = null;
        this.data = data;
    }

   public void setNext(Node node){
       next = node;
   }

   public Node getNext(){
       return next;
   }

    public void push(Node node){
        if(next == null){
            next = node;
        } else {
            next.push(node);
        }
    }

    public DataElement getData() {
        return data;
    }

    public void printList(){
        data.presentation();
        if (next != null) next.printList();
    }
}
```

Auch in dieser Klasse sind die Veränderungen überschaubar, lediglich an den Stellen, an denen mit den Daten interagiert wird hat sich die Syntax leicht verändert.
Betrachtet man die rekursiven Methoden, in diesem Fall die hintenAnfügen() und auflisten()-Methode, so wird noch eine „Schwäche“ der Implementierung deutlich. Die Abbruchbedingungen der Rekursion sind mit einer Prüfung der Null-Referenz verknüpft. Diese Implementierung ist zwar nicht falsch, aber unelegant. Man kann diese Prüfungen vermeiden, indem man eine weitere Verbesserung vornimmt, die Verwendung eines Strukturmusters, des Kompositums.

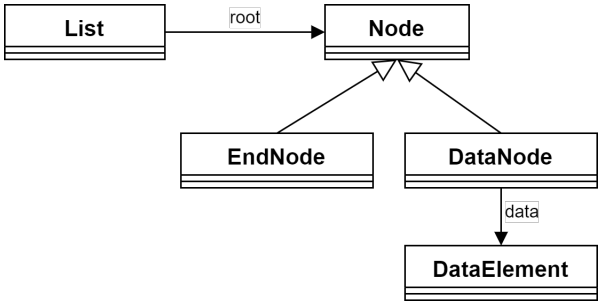
Aufgabe 4: Zeichnen Sie ein vollständiges Klassendiagramm der obigen Implementierung. Ergänzen Sie außerdem die Klasse Tier und eine weitere selbst gewählte Klasse, die an die Liste angehängt werden kann. Fügen Sie bei den Methoden nicht nur die hinzu, die bereits implementiert sind, sondern greifen Sie auf Methoden aus vorherigen Kapiteln zurück und passen Sie deren Signaturen entsprechend der neuen Situation an.

5 Die verkettete Liste - das Kompositum

In der objektorientierten Programmierung werden häufig **Entwurfsmuster** verwendet. Sie beschreiben eine Struktur, die in verschiedensten Bereichen Anwendung finden kann. Beim **Kompositum** handelt es sich um ein Muster, dass Teil-Ganzes-Hierarchien beschreibt. Ein klassisches Beispiel ist die Ordnerstruktur auf einem normalen PC. Ein Ordner kann Dateien, aber auch weitere Ordner enthalten, d.h. auf jeder Ebene kann die Struktur weiter „nach unten“ wachsen, solange weitere Ordner vorhanden sind.
Ein anderes Beispiel sind Bilder aus geometrischen Formen. Jedes Dreieck, Quadrat oder Kreis kann bereits als Bild gesehen werden, zusammengefasst ergeben sie aber wieder ein Bild, das wiederum nur ein Teilbild sein könnte, etc. Ein allgemeines Klassendiagramm der Struktur sieht so aus:



Die Komponente ist in der Regel eine abstrakte Klasse. In obigem Beispiel wären Dreiecke oder Quadrate Blätter, eine aus zwei Dreiecken zusammengefügte Figur ein Composite, also ein Zusammenschluss aus diesen Objekten. Die Blätter sind die kleinsten Einheiten der Struktur, zu ihnen kann also kein weiteres Objekt hinzugefügt werden. Auf den ersten Blick bringt uns dieses Konzept keine großen Vorteile im Hinblick auf unsere Liste. Wir haben aktuell bereits eine ähnliche Struktur, da jede Teilliste ab einem bestimmten Knoten wieder als Liste gesehen werden könnte. Mit einer kleinen Erweiterung werden wir so aber sogar unsere Null-Referenzprüfungen los:



Unser Blatt, d.h. das Ende unserer Liste wird zu einer eigenen Klasse. Alle Abbruchbedingungen wandern also in diese Klasse, da alle Methoden, die die Liste ganz durchlaufen, zwangsläufig hier ankommen müssen. So kann z.B. das Einfügen eines neuen Knotens ebenfalls durch „Durchreichen“ der entsprechenden Daten geschehen. Jeder Knoten gibt dabei sich selbst wieder an den Vorgänger als Nachfolger zurück, bis auf den letzten Knoten. Dieser gibt nicht sich selbst, sondern einen neuen Knoten mit sich als Nachfolger zurück und die Liste wurde verlängert.
Für die Implementierung beginnen wir wiederum mit dem Grundgerüst. Wir benötigen Klassen für:

1. Warteschlange
2. (abstrakter) Knoten
3. Endknoten
4. Datenknoten
5. Datenelment
6. Mensch

```
public class MyListComp {
    private Node root;

    public MyListComp() {
        root = new EndNode();
    }
}
```

Der Konstruktor der Warteschlange platziert jetzt keine Null-Referenz mehr, sondern hängt einen Endknoten als Wurzel an die Liste an. D.h. die Liste ist jetzt immer mit einem „selbst-erstellten“ Objekt gefüllt.
In der Knoten Klasse werden nur die Methodenrümpfe für die tatsächlichen Implementierungen gesetzt (wir könnten diese abstrakte Klasse also auch als Interface implementieren). Für den Moment beschränken wir uns auf die hintenAnfügen() - Methode, die wir im nächsten Schritt implementieren wollen (die Entfernen-Methode ist nicht rekursiv und muss deswegen auch nicht auf den Knoten definiert werden!)

```
public abstract class Node {
    public abstract Node push(DataElement data);
}
```

Wer genau hinsieht: Die Methode zum Anfügen eines Knotens hat jetzt einen Rückgabewert! Dies liegt an der bereits oben beschriebenen Vorgehensweise beim Einfügen, dazu später mehr.

```
public class EndNode extends Node {
    @Override
    public Node push(DataElement data) {
        // TODO
    }
}
public class DataNode extends Node {
    private Node next;
    private DataElement data;

    public DataNode(Node node, DataElement data) {
        next = node;
        this.data = data;
    }

    public DataElement getData() {
        return data;
    }

    @Override
    public Node push(DataElement data) {
        // TODO
    }
}
```

Der Endknoten braucht natürlich weder eine Referenz auf den nächsten Knoten, noch trägt er einen Datensatz, deswegen gibt es in dieser Klasse auch keinerlei Attribute.
Im Datenknoten fällt auf, dass die Referenz auf den nächsten Knoten jetzt bereits direkt im Konstruktor als Übergabeparameter gesetzt wird. Diese Implementierung ist wiederum für die rekursiven Methoden nützlich.
Es folgt das Datenelement-Interface:

```
public interface DataElement {
    public void presentation();
    public boolean isGreater(DataElement data);
    public boolean equals(Object o;)
```

```
    }
```

Das Interface greift bereits vorweg, da die drei Methoden noch nicht für die Anfügen und Entfernen-Methoden gebraucht werden, sondern nur für die später folgenden Methoden.
In der Klasse Mensch sind deswegen dementsprechend bereits alle diese Methoden implementiert. Es ändert sich im Vergleich zur vorherigen Implementierung der verketteten Liste nichts. Auf die getter- und setter-Methoden für die Attribute wurde aus Platzgründen verzichtet.

```
    public class Human implements DataElement {

        private String name;
        private int age;

        public Human(String name, int age){
            this.name = name;
            this.age = age;
        }

        @Override
        public void presentation(){
            System.out.println("I am " + name + " and I am " + age + " years old");
        }

        @Override
        public boolean equals(Object o){
            if(this == o) {
                return true;
            }
            if(!(o instanceof Human)){
                return false;
            }
            Human h = (Human) o;
            if(h.getName() == this.name && h.getAge() == this.age){
                return true;
            } else {
                return false;
            }
        }

        @Override
        public boolean isGreater(DataElement data) {
            Human human = null;
            try {
                human = (Human) data;
            } catch (Exception e) {
                System.out.println("The element for the comparison is not a human!");
            }
            if(this.getAge() > human.getAge()) {
                return true;
            }
            return false;
        }

    }
```

Bei der Verwendung der Methoden ist wieder zu beachten, dass die Methode zur Prüfung auf Gleichheit auf Name und Alter zurückgreift, die Methode für die Sortierung allerdings nur auf das Alter.

Damit ist unser Gerüst komplett und der Fokus kann auf die Implementierung der hintenAnfügen()-Methode gelegt werden. In der Warteschlangen-Klasse hat sich nur die null-Abfrage verändert. Statt dieser wird nun geprüft, ob die Wurzel ein Endknoten ist. Falls ja wird ein neuer Datenknoten erzeugt und an die Stelle der alten Wurzel gesetzt. *Hinweis:* Der Konstruktor für den Datenknoten erhält die Referenz der **alten** Wurzel, die zu diesem Zeitpunkt noch der Endknoten ist, erst dann wird wieder in die Variable wurzel gespeichert!

```
//class MyListComp
public void push(DataElement data) {
    if(root instanceof EndNode){
        root = new DataNode(root, data);
    } else {
        root.push(data);
    }
}
```

Die Implementierung in der Datenknotenmodellierung ist jetzt sehr schlank:

```
//class DataNode
public Node push(DataElement data) {
    next = next.push(data);
    return this;
}
```

Wir müssen hier keine Fallunterscheidung mehr machen, da der betrachtete Knoten sicher nicht das Ende ist, wir müssen lediglich auf dem nächsten Datenknoten wieder die Methode aufrufen.

Aufgabe 1: Überlegen Sie bevor Sie weiterlesen noch einmal selbst, wieso diese Methode einen Knoten zurückgibt und warum der Rückgabewert wieder in der Variable des Nachfolgers gespeichert wird.

Anschaulich gesprochen fragt der Datenknoten, ob sein Nachfolger noch derselbe bleibt, denn das Ergebnis des Methodenaufrufs wird im Nachfolger gespeichert. Anschließend gibt er sich selbst zurück, damit sein Vorgänger den Knoten wieder in der entsprechenden Referenz speichern kann. Es soll sich bei keinem Knoten außer dem letzten die Referenz verändern!
Kommen wir beim Endknoten an, so soll dieser Endknoten nicht mehr sich selbst zurückgeben, sondern einen neuen Datenknoten. Das bedeutet, der vormals letzte Knoten erhält als Referenz den neuen Knoten zurück!

```
//class EndNode
public Node push(DataElement data) {
    return new DataNode(this, data);
}
```

Veranschaulicht in einem Video:

Aufgabe 2: Implementieren Sie die vorneEntfernen() und auflisten()- Methode selbstständig, bevor Sie weiterlesen.

Um die Grundfunktionalität wiederherzustellen sind die in der vorherigen Aufgabe erwähnten Methoden nötig:

```
//class MyListComp
public DataElement pop() {
    if(root instanceof EndNode){
        System.out.println("No list, nothing to remove");
        return null;
    } else {
        DataElement toReturn = root.getData();
        root = root.getNext();
        return toReturn;
    }
}
public void printList() {
    if(root instanceof EndNode){
        System.out.println("No list here to print!");
    } else {
        root.printList();
    }
}
```

Es bietet sich für alle rekursiven Methoden an, einen Eintrag in der abstrakten Klasse Knoten zu hinterlassen, um die Implementierung zu erwzingen (auch wenn in der Endknoten-Kasse gegebenenfalls nichts passiert, die Methode muss auch auf dieser Art von Knoten aufrufbar sein, um den Abbruch einzuleiten!).

```
//class Node
public abstract void printList();

//class DataNode
public void printList() {
    data.presentation();
    next.printList();
}

//class EndNode
public void printList() {
    return;
}
```

Es folgt der bereits altbekannte Aufgabenblock, alle rekursiven Methoden sollen möglichst die Struktur des Kompositums ausnutzen und innerhalb der Knotenimplementierungen auf instanceof-Verwendungen verzichten. (In der Warteschlangen-Klasse ist dies i.d.R. nicht vermeidbar). Die Lösungen für die ersten sieben Aufgaben finden sich kommentiert auf den folgenden Seiten. Der reine Quellcode (auch für die Experten-Aufgaben 8 und 9) ist in einem seperaten pdf zusammengestellt, sowie auf mebis und github verfügbar.

Aufgabe 3: Schreiben Sie eine Methode länge() (length()), die die aktuelle Länge der Warteschlange bestimmt und zurückgibt.

Aufgabe 4: Schreiben Sie eine Methode itemAnPosition(int position) (itemAtPosition(int position)), die eine Referenz zum Listenelement an der angegebenen Position zurückgibt. Beachten Sie Grenz- bzw. Spezialfälle!

Aufgabe 5: Schreiben Sie eine Methode sucheInSchlange(Datenelement d) (searchItemInQueue(DataElement d)), die die Position des Datenelements in der Schlange zurückgibt.

Aufgabe 6: Schreiben Sie eine Methode enthält(DataElement d) (contains(DataElement d)), die wahr zurückgibt, wenn das Datenelement in der Schlange ist, anderfalls falsch.

Aufgabe 7: Schreiben Sie eine Methode entferneAnPosition(int position) (removeAtPosition(int position)), die das Datenelement an der gegebenen Stelle entfernt.
Schreiben Sie zwei Versionen dieser Methode, eine, die das Datenelement nur entfernt und eine, die eine Referenz zu diesem Element zurückgibt.

Aufgabe 8: Schreiben Sie eine Methode Knoten entferneElement(Datenelement zuEntfernen) (Node removeElement(DataElement toRemove)) und adaptieren Sie das Vorgehen der hintenAnfügen()-Methode, um die Struktur des „Durchreichens“ zu verwenden.

Aufgabe 9 - für Experten: Schreiben Sie eine Methode zusammenfügen(MeineListeComp zweiteWarteschlange) (concatenate(MyListComp secondQueue)), die eine zusammengefügte Liste zurückgibt.

Aufgabe 10 - für Experten: Schreiben Sie eine Methode fügeSortiertHinzu(Datenelement d) (appendSorted(DataElement d)), die die Warteschlange nicht von hinten auffüllt, sondern die Menschen an einer bestimmten Stelle einsortiert.

Aufgabe 11 - frei: Überlegen Sie sich eigene, sinnvolle weitere Methoden für die Warteschlangen-Implementierung.

Für die Länge der Warteschlange soll der Endknoten nicht zählen, d.h. es wird nur die Anzahl der Datenknoten benötigt. Die Frage nach der Länge wird also bis zum Endknoten durchgereicht, dieser antwortet mit 0, alle anderen zählen zum Ergebnis ihres Vorgängers eins dazu und geben weiter zurück.
Beispiel:Der letzte Datenknoten erhält vom Endknoten die 0, addiert 1 und gibt an den vorletzten 1 zurück. Dieser addiert wiederum 1 und gibt 2 weiter zurück, usw.

```
    //class MyListComp
    public int length() {
        return root.length();
    }

    //class Node
    public abstract int length();

    //class DataNode
    @Override
    public int length() {
        return next.length() + 1;
    }

    //class EndNode
    @Override
    public int length() {
        return 0;
```

```
    }
```

Um ein Datenelement an einer bestimnmten Position zu finden müssen wir wieder auf das Konzept des Zählers, der durchgereicht wird, zurückgreifen. Die Abfrage, ob die Liste kürzer als die gesuchte Position ist können wir umgehen, indem wir den Endknoten null zurückgeben lassen, sollte er erreicht werden. Wie in früheren Beispielen soll in dieser Implementierung außerdem das Zählen bei 1 beginnen, d.h. die Wurzel ist der erste und nicht der „Nullte“ in der Reihe.

```
    //class MyListComp
    public DataElement itemAtPosition(int position){
        int counter = 1;
        DataElement found = root.itemAtPosition(position, counter);
        return found;
    }

    //class Node
    public abstract itemAtPosition(int position, int counter);

    //class DataNode
    @Override
    public DataElement itemAtPosition(int position, int counter) {
        if(counter == position){
            return this.getData();
        } else {
            return next.itemAtPosition(position, counter + 1);
        }
    }

    //class EndNode
    @Override
    public DataElement itemAtPosition(int position, int counter) {
        return null;
    }
```

Die Methode, die in der Schlange sucht folgt der exakt selben Logik, nur gibt hier der Endknoten die (willkürlich) festgelegte -1 statt null zurück:

```
    //class MyListComp
    public int searchItemInQueue(DataElement data) {
        int counter = 1;
        int position = root.searchItemInQueue(data,counter);
        return position;
    }

    //class Node
    public abstract int searchItemInQueue(DataElement data, int counter);

    //class DataNode
    @Override
    public int searchItemInQueue(DataElement data, int counter) {
        if(this.getData().equals(data)){
            return counter;
        } else {
            return next.searchItemInQueue(data, counter +1);
        }
    }
```

```
//class EndNode
@Override
public int searchItemInQueue(DataElement data, int counter) {
    return -1;
}
```

Wie immer kann für die enthält()-Methode ein eigener rekursiver Durchlauf verwendet werden, oder es wird auf die Methode aus der vorherigen Aufgabe zurückgegriffen, diesmal in der kürzeren Variante:

```
//class MyListComp
public boolean contains(DataElement data) {
    if(this.searchItemInQueue(data) != -1) {
        return true;
    } else {
        return false;
    }
}
```

Die Methode, die an einer beliebigen Stelle entfernt, kann auf die vorneEntfernen()-Methode zurückgreifen, falls die spezifizierte Position der ersten entspricht. Andernfalls wird wieder ein Zähler benötigt, die Struktur sieht völlig analog zu früheren Varianten aus (der Endknoten wird wieder null zurückgeben!):

```
//class MyListComp
public DataElement removeAtPosition(int position) {
    if(position <= 0) {
        return null;
    }
    if(position == 1) {
        return this.pop();
    } else {
        int counter = 1;
        return root.removeAtPosition(position, counter);
    }
}

//class Node
public abstract DataElement removeAtPosition(int position, int counter);

//class DataNode
@Override
public DataElement removeAtPosition(int position, int counter){
    if(counter == position - 1){
        DataElement toReturn = next.getData();
        next = next.getNext();
        return toReturn;
    } else {
        return next.removeAtPosition(position, counter + 1);
    }
}

//class EndNode
@Override
public DataElement removeAt(int position, int counter){
    return null;
}
```

In der Liste wird nur auf der Wurzel das Entfernen auf der Wurzel gestartet. Da die Wurzel in jedem Fall mindestens ein Endknoten ist, entfällt wiederum eine Prüfung, ob die Liste leer ist. Ein Endknoten gibt nur sich selbst zurück

und es entsteht kein Schaden. Durch die Implementierung auf diese Weise ist allerdings keine Bestätigung der Ausführung möglich, da für das „elegante“ Durchreichen eine Rückgabe des Knotens notwendig ist.
Im Datenknoten wird überprüft, ob das übergebene Element identisch mit dem Inhalt dieses Knotens ist (dazu wird wieder die vergleichsMethode des Datenelements genutzt!). Fall dies der Fall ist, wird der Nachfolger des Knotens zurückgegeben. Dadurch erhält sein Vorgänger eine Referenz auf den Nachfolger und der Knoten mit den gesuchten Daten ist entfernt.

```
//class MyListComp
public void removeElement(DataElement toRemove) {
    root = root.removeElement(toRemove);
}

//class Node
public abstract Node removeElement(DataElement toRemove);

//class DataNode
public Node removeElement(DataElement toRemove) {
    if(toRemove.getData().equals(data.getData())) {
        return next;
    } else {
        next = next.removeElement(toRemove);
    }
    return this;
}

//class EndNode
public Node removeElement(DataElement toRemove) {
    return this;
}
```

6 Spezielle Listen-Typen

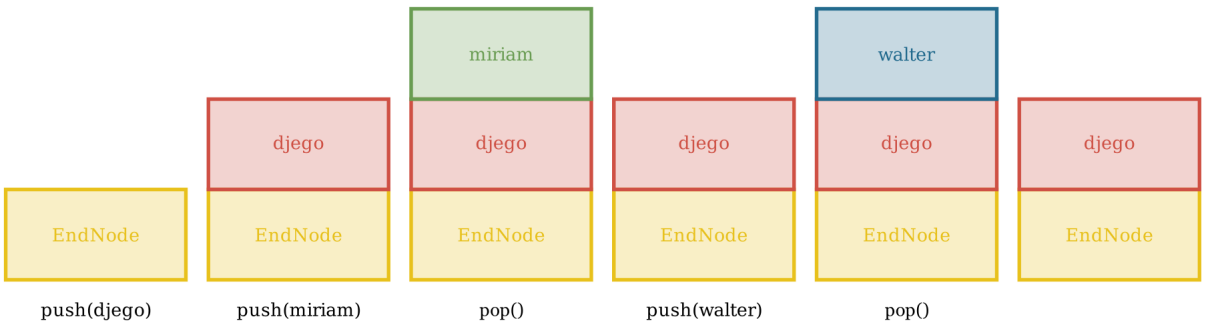
Dem aufmerksamen Leser dieses Skripts ist sicher aufgefallen, dass in vielen Fällen von der Warteschlange die Rede war, jedoch die Implementierungen meistens mit einer Form von Liste benannt waren. Das liegt daran, dass mit der Erweiterung der Funktionalität der Listen die reine Form der Warteschlange „zerstört“ wurde.
Man unterscheidet in der Informatik im Wesentlichen drei Formen von Listen (natürlich gibt es auch davon noch weitere Untertypen bzw. verschiedene Implementierungsweisen).

1. **Warteschlange**: das zuerst angefügte Element verlässt die Liste auch zuerst (First-In-First-Out: FIFO-Prinzip)
2. **Stack**: das zuletzt angefügte Element verlässt die Liste zuerst (Last-In-First-Out: LIFO-Prinzip)
3. **Sortierte Liste**: ein Element wird gemäß einer bestimmten vorgegebenen Logik in die Liste eingefügt. Es wird in der Regel nicht exklusiv von vorne oder von hinten entfernt.

Durch die Erweiterung der Funktionalität der Feld-Liste haben wir also beispielsweise die Warteschlange mit der sortierten Liste vermischt. Die Experten haben dies auch für die verkettete Liste in Aufgabe 8 aus dem letzten Kapitel bereits durchgeführt.
Es wäre ungeschickt die verkettete Liste die Warteschlange, den Stack und die sortierte Liste jeweils einzeln vollständig zu implementieren. Viele Methoden überschneiden sich in ihrer Funktionalität bzw. sind vollständig identisch. Klüger wäre also, die gesamte Logik aller Listen in einer Struktur (z.B. MeineListe, MyList) zusammenzufassen und dann für die Warteschlange, den Stack oder die sortierte Liste nur die Methoden der Liste zu übernehmen, die für diesen Fall Anwendung finden.
Man spricht hier von einer **Adapter-Klasse**, da keine neue Logik hinzugefügt wird, sondern nur auf die Funktionalität einer bereits bestehenden Klasse zugegriffen wird.
Bevor wir eine allgemeine Listen-Klasse erstellen und die Adapterklassen definieren, muss noch der Stack und die sortierte Liste untersucht werden.

6.1 Stack

Ein Stack entspricht einem Stapel in der „realen“ Welt recht genau. Eine hilfreiche Vorstellung ist ein hoher Bücherstapel, bei dem nur das zuletzt hinzugefügt Buch, nämlich das Oberste, ungefährdet wieder entfernt werden können. Gehen wir von einer Implementierung mit Kompositum aus, könnte eine Folge von Einfüge- bzw. Entfernungsoperatoren wie folgt aussehen:



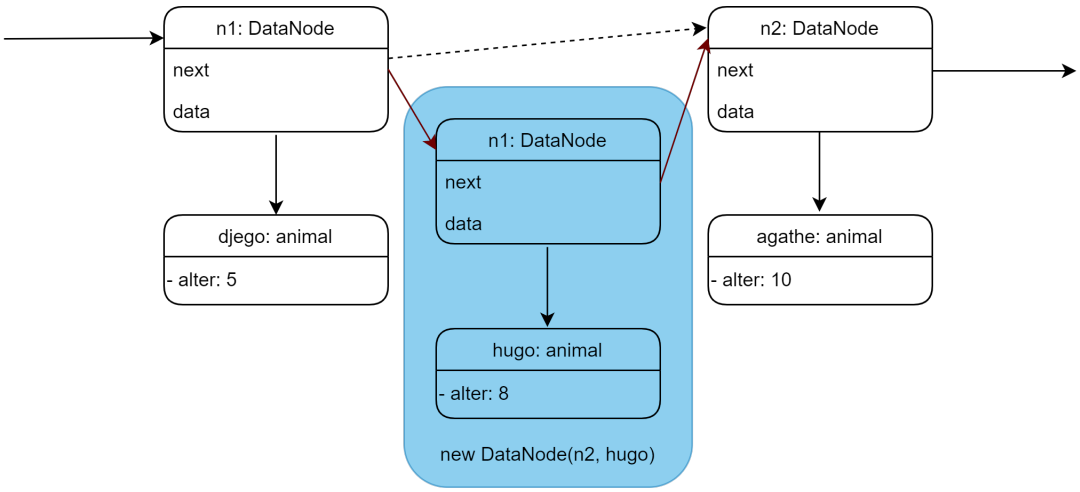
Stacks werden in der Informatik häufig verwendet, sie spielen z.B. eine Rolle bei Tiefendurchläufen von Graphen (siehe 11—2) und sind ein Hilfsmittel zur Syntaxüberprüfung, bzw. zur Auswertung von Ausdrücken allgemein. (siehe auch formale Sprachen 12—1).
Auch beim Ausführen von Methoden werden die noch abzuwarbeitenden Anwendungen auf einem Stack verwaltet. So kommt es bei rekursiven Methoden auch dazu, dass zuerst bis zur „tiefsten Stelle“ vorgedrungen wird, da immer mehr Aufrufe auf den Stack gelegt werden, bis an der Abbruchbedingung schließlich einer fertiggestellt wird und es zurück nach unten geht.
Hinweis: Hat man eine rekursive Methode, die keine Abbruchbedingung hat, so wird das Programm abgebrochen, sobald der Befehlsstack vollgelaufen ist (er hat also eine begrenzte Größe). So wird eine Endlosschleife vermieden.
Im Gegensatz zur Warteschlange wird bei einem Stack immer vorne eingefügt, anstatt hinten (in beiden Implementierungen heißt die Methode im Englischen push()). Das Entfernen verläuft völlig analog zur Warteschlange.

6.2 Sortierte Listen

Im dritten Kapitel und im fünften Kapitel haben die Experten sich bereits mit Sortierung beschäftigt. Die wichtigste Voraussetzung, um überhaupt sortieren zu können, ist eine Vorschrift was „größer“ bzw. „kleiner“ ist. Im Falle der Menschen wurde willkürlich das Alter ausgewählt, es hätte aber auch der Name oder eine Kombination aus beidem als Schlüssel zur Sortierung dienen können. Als Festlegung gehen wir von einer aufsteigend sortierten Liste aus.
Für die Implementierung gehen wir wieder von einer verketteten Liste mit Kompositum aus. Die Listenklasse muss dann lediglich wieder das Anfügen starten:

```
//class MyListComp
public void appendSorted(DataElement data) {
    root = root.appendSorted(data);
}

//class Node
public abstract Node appendSorted(DataElement data);

//class DataNode
public Node appendSorted(DataElement data) {
    if(this.data.isGreater(data)) {
        return new DataNode(this, data);
    }
    next = next.appendSorted(data);
    return this;
}

//class EndNode
public Node appendSorted(DataElement data){
    return new DataNode(this, data);
}
```

Jeder Datenknoten prüft, ob seine Daten größer sind als die einzufügenden, denn dann muss ein neuer Datenknoten geschaffen werden und dem Vorgänger als neuer Nachfolger übergeben werden. Der aktuelle Datenknoten setzt sich selbst dann als Nachfolger des neuen Knotens.

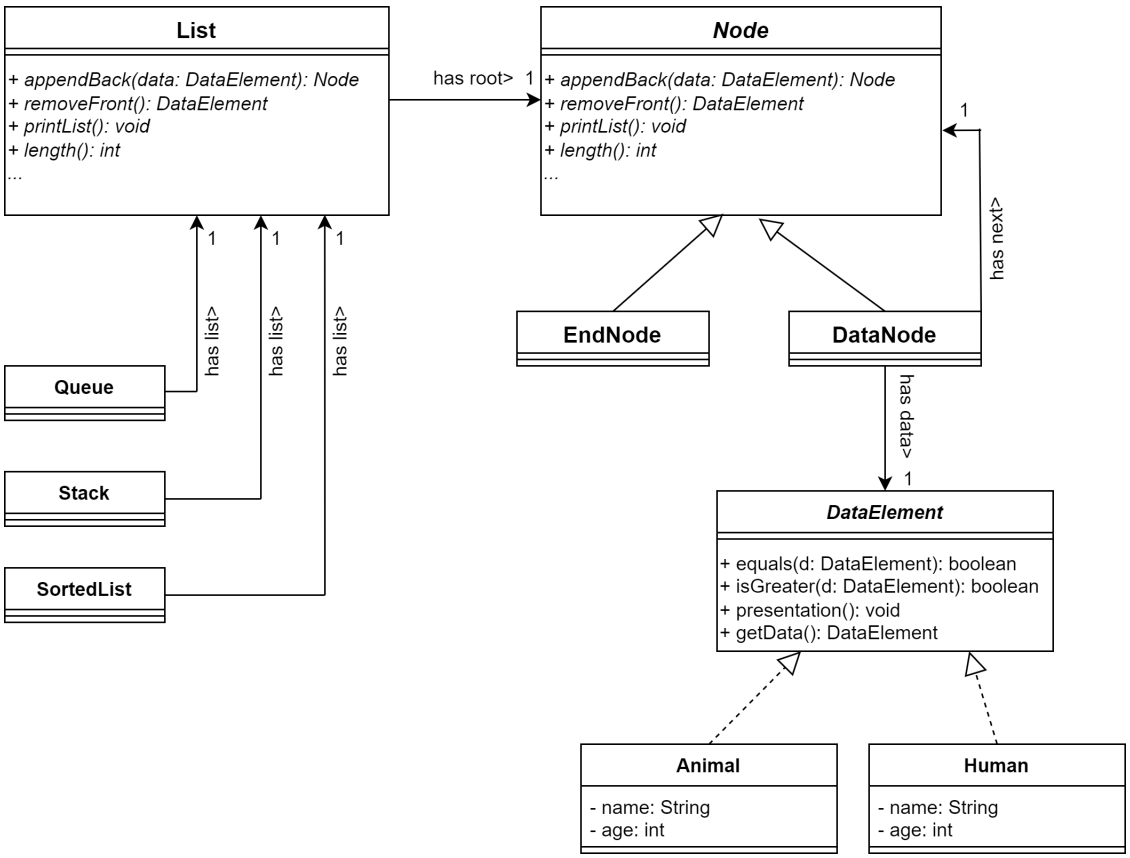


Das gleiche geschieht im Endknoten, allerdings in jedem Fall, da der neue Knoten das größte Element enthalten muss, wenn es bis ganz nach hinten „durchgekommen“ ist.

6.3 Die Adapter-Klassen

Mit Hilfe dieser Vorarbeit ist jetzt alles geklärt, um das letzte große Projekt in Angriff zu gehen, die Implementierung der Liste mit allen drei Adapterklassen.

Die Listen-Klasse soll dabei alle Logik enthalten, ein Auszug des Klassendiagramms sieht wie folgt aus:



Aufgabe Abschlussaufgabe: Implementieren Sie die obige Struktur, insbesondere die Adapterklassen. Erweitern Sie dazu sukzessive die Methoden in der Klasse Liste. Sie können auf bisherige Implementierungen zurückgreifen.

Für Interessierte gibt es auch am Ende dieses Kapitels noch mehr zu entdecken, einige Anregungen:

- Untersuchen Sie die Java-Klasse ArrayList und LinkedList, vergleichen Sie mit unserer eigenen Implementierung.
- **Generics**: Auch Java bietet bereits die Möglichkeit allgemeinere Listen zu definieren, so gibt kann eine ArrayList eine Liste von Strings oder Zahlen oder anderen Objekten sein, recherchieren Sie die Funktionsweise.

7 Anhang

7.1 Klassendiagramme

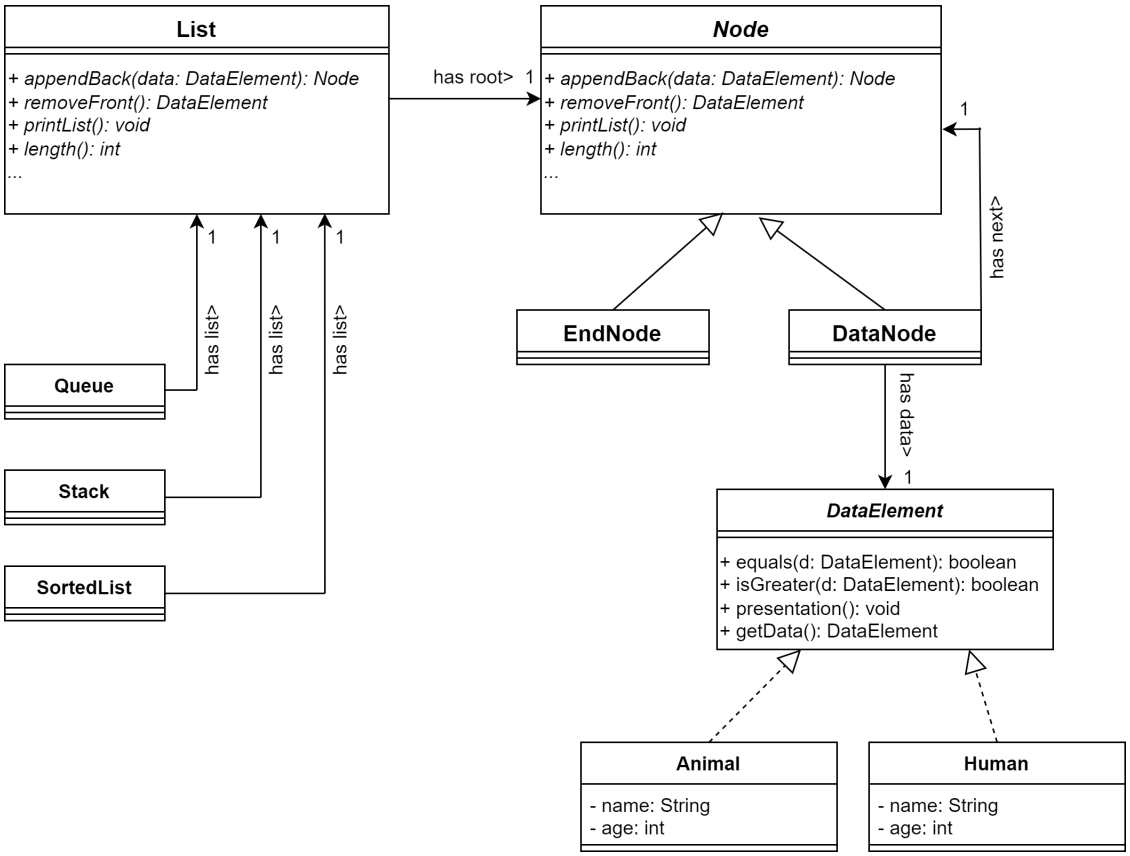
Ein Klassendiagramm enthält eine Klassenkarte für alle beteiligten Klassen, sowie Pfeile zwischen ihnen, um die Beziehungen zu veranschaulichen. Für das Abitur wird noch zwischen erweiterten Klassendiagrammen und „normalen“ Klassendiagrammen unterschieden. Das reguläre Diagramm enthält:

- Attribute (nur Bezeichner)
- Methoden mit den Übergabeparametern (Bezeichner)
- Beziehungen zwischen Klassen, Referenzen werden auf Verbindungen modelliert, inklusive Mengenangaben.
- i.d.R. ohne Geben- und Setzen- Methoden

Für das erweiterte Klassendiagramm zusätzlich:

- Datentypen der Attribute
- ggf. Referenzattribute
- Rückgabewerte der Parameter
- Konstrukturen

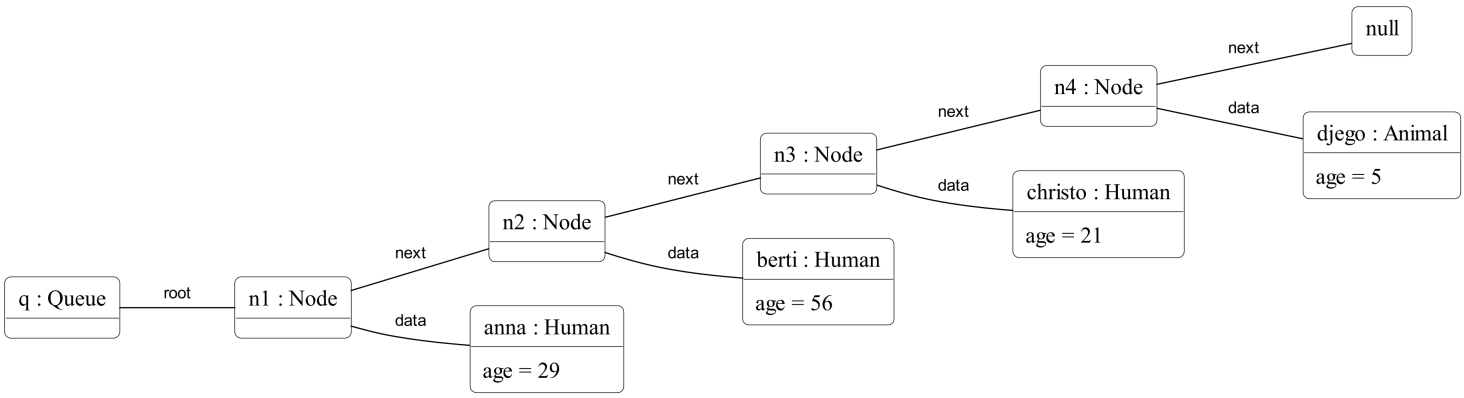
Vererbungen werden als Pfeile mit nicht-ausgemalter Spitze dargestellt. Abstrakte Klassen und Interfaces werden kursiv dargstellt oder (einfacher zu zeichnen) mit ¡¡abstract¿¿ bzw. ¡¡interface¿¿ gekennzeichnet. Werden Interfaces implementiert, so sieht der Pfeil aus wie bei einer Vererbung, allerdings gestrichelt.
Folgendes Beispiel aus dem Skript enthält die meisten relevanten Elemente als Beispiel.



Hinweis: Im Zweifelsfall schadet mehr Information im Diagramm nicht. Das erweiterte Klassendiagramm ist zu bevorzugen.

7.2 Objektdiagramme

Im Gegensatz zum Klassendiagramm, das die allgemeine Struktur der Implementierung darstlelen soll, stellt das Objektdiagramm den Zustand des Systems zu einem bestimmtem Zeitpunkt dar.
Das folgende Beispiel zeigt den Zustand einer Liste zu einem bestimmten Zeitpunkt. Jedes Objekt wird mit einem Kasten mit abgerundeten Ecken dargestellt. Es wird mit ¡Bezeichner¿:¡Klassenname¿ beschriftet. Die Beziehungen werden wiederum auf den Verbindungslinien dargestellt, die jeweiligen Attribute mit ihren aktuellen Werten in einem seperaten Abschnitt.



7.3 Sequenzdiagramme

Die Aufgabe des Sequenzdiagramms ist es, Methodenaufrufe - insbesondere über mehrere Objekte hinweg - darzustellen. Jedes Objekt erhält dabei eine sogenannte Lebenslinie. Diese Linie stellt dar, wie lange das Objekt „lebt“, da es z.B. auch nur kurzzeitig erzeugt werden könnten.
Methodenaufrufe werden durch Pfeile dargestellt und der Methodenname wird über den Pfeil geschrieben. Die Rückgabe wird ebenfalls mit einem Pfeil dargestellt, dieses Mal mit dem Rückgabewert über ihm dargestellt. Der Rückgabepfeil wird häufig gestrichelt dargestellt.
Sobald ein Objekt aktiv ist, wird ein Aktivitätsbalken gezeichnet. (I.d.R. gestartet durch einen Methodenaufruf und beendet durch eine Rückgabe).
Relevante Attribute können im abgerundeten Kasten am Beginn der Lebenslinie eingetragen werden. Beispiel aus dem Skript: Längenbestimmung einer verketteten Liste.