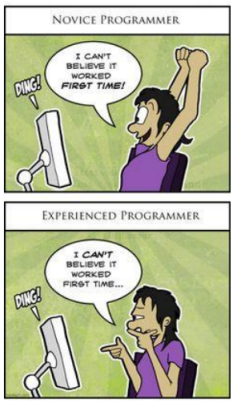
Skript zu Listen

Lars Wechsler

17. August 2022

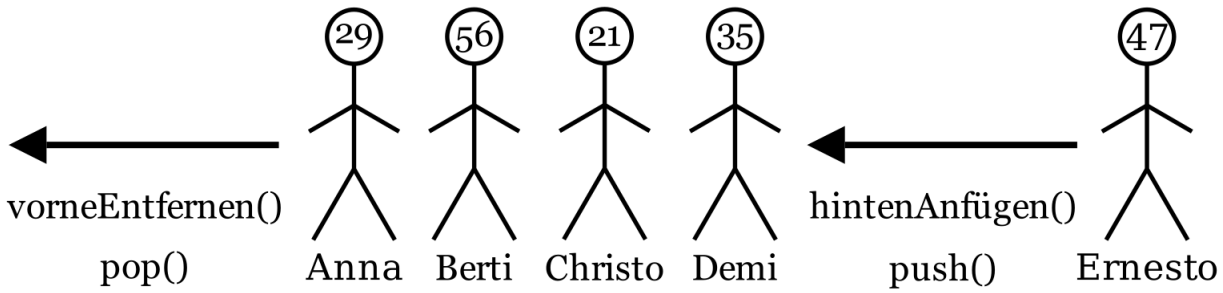
1 Einleitung



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 nicht aktiv über sie nachdenken.
Interessant wird 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 etliche Programmiersprachen gibt, die sich auf die Liste als Basiselement zurückgreift, z.B. Lisp.
Traditionell ist die Liste ein umso wichtigeres Element der Sprache, desto funktionaler diese angehaucht 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, werden die Methoden im Fließtext auf Deutsch stehen, mit der üblichen englischen Übersetzung bei der ersten Verwendung in Klammern. Auch die Abbildungen werden beide Schreibweisen enthalten, der Quellcode nur englische Bezeichungen (Im Anhang finden sich zu den entsprechenden Kapiteln auch voll deutsche Code-Schnipsel). Für Prüfungen sind sowohl deutscher als auch englischer Code in Ordnung. Begründungen müssen aber 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. Sie stehen in einer Warteschlange (Queue). Auch eine Warteschlange ist 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. In der Modellierung entspricht dies den Methoden vorneEntfernen() (pop()) bzw. hintenAnfügen() (push()). Die Warteschlange gehorcht damit dem sogenannten FIFO-Prinzip (First In, First Out).

Es bleibt die Frage, wie diese Ideen objektorientiert umgesetzt werden können.

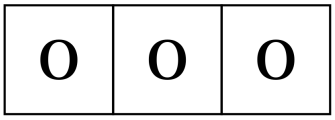
2 Die Feld-Liste (ArrayList)

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). Der zweite Typ kann noch weiter unterteilt werden, aber dazu später mehr. Für den Moment wollen wir uns ein Feld als reservierten Speicherplatz vorstellen, im Folgenden symbolisiert durch eine Aneinanderreihung an Quadraten. Aktuell ist das Feld noch vollständig leer.

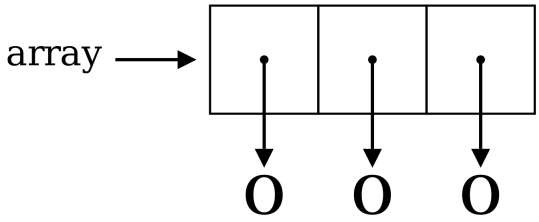
In Java werden Felder standardmäßig mit 0 bzw. null intialisiert, wenn keine weiteren Vorgaben gemacht werden.
Beispiel:

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

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



Die Darstellung ist so noch nicht ganz akurat. Das Feld enthält eigentlich nur weitere Zeiger, die zu den entsprechenden Positionen im Arbeitsspeicher zeigen, an denen die Daten liegen. Insbesondere, wenn man keine primitiven Datentypen verwendet, müssten ansonsten häufig Daten verschoben werden, wenn neue Elemente hinzugefügt oder geändert werden.
Eine bessere Darstellung wäre also:



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

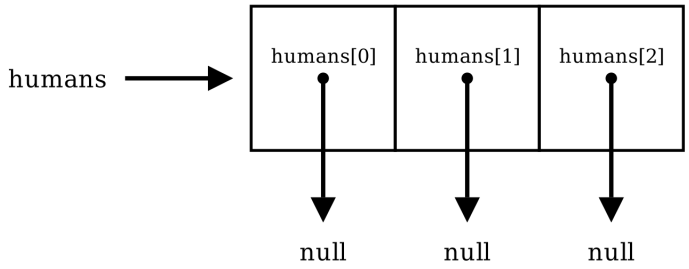
```
Human[] humans = new Human[5];
```

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, z.B.:

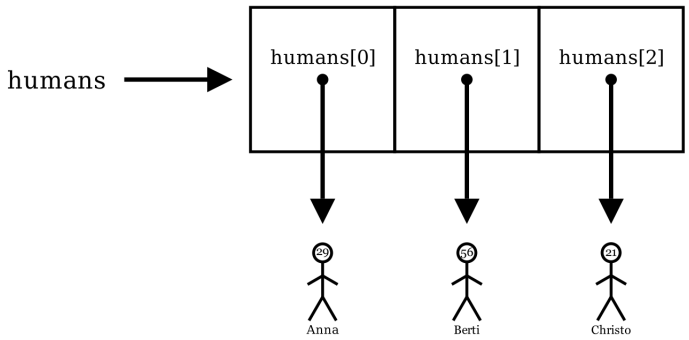
```
int[] numbers = {5,3,42,17,-2};
int thirdEntry = numbers[2];
System.out.println(thirdEntry);
```

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.

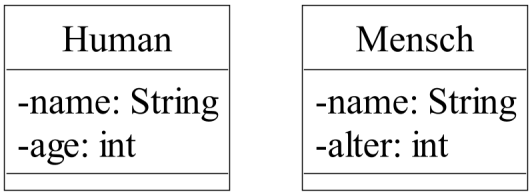
Versuchen wir stattdessen, z.B. mit humans[0] auf den ersten Menschen zuzugreifen, werden wir nur null erhalten, da dieses Feld aktuell noch so aussieht:



Haben wir die Methode hintenAnfügen() (push()) implementiert und drei Menschen Anna, berti und Christo hinzugefügt:



Grundsätzlich bietet das Feld an sich in seiner Implementierung 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 aber zunächst eine Klasse, die einen Menschen beschreibt.




```
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(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;
    }

}
```

Um die Warteschlange auch sinnvoll verwenden zu können müssen Methoden ergänzt werden. Wir beginnen mit der hintenAnfügen() Methode. Da wir spezifizieren müssen, was wir anfügen wollen sieht die vollständige Signatur der Methode so aus:

```
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;
        }
    }
}
```

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

Eine effizientere Lösung besteht darin, die aktuelle Anzahl an Menschen in der Warteschlange in einem Attribut mitzuzählen, 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“.
Ist die Liste voll, so muss das Feld vergrößert werden, da dies eine Funktionalität ist, die wir eventuell noch an anderer Stelle brauchen könnten, wird eine eigene Methode angelegt.

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

Um die Methode effizienter zu gestalten, wird das Feld um 10 Plätze vergrößert. Man möchte zu große Felder vermeiden, um nicht unnötig Speicherplatz zu verschwenden (die Implementierung der ArrayList, die von Java direkt angeboten wird verwendet z.B. eine Vergrößerung um 5 Plätze).
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. 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 < queue.length -1; i++){
            if (queue[i+1] == null){
                break;
            }
```

```
            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.

Die folgenden Aufgaben steigen grob im Schwierigkeitsgrad an und erweitern die Funktionalität der Warteschlange sukzessive.

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.

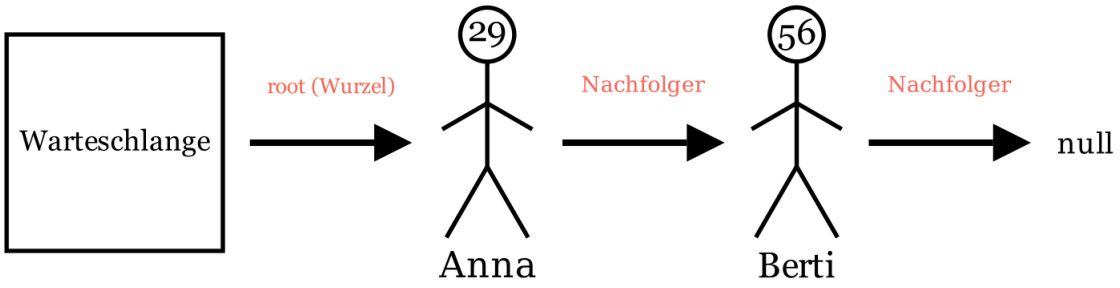
Aufgabe 9 - für Experten: Schreiben Sie eine Methode fügeSortiertHinzu(Mensch m) (appendSorted(Human h)), die die Warteschlange nicht von hinten auffüllt, sondern die Menschen an einer bestimmten Stelle einsortiert.
Hinweis: Dazu muss eine Vergleichsmethode in der Klasse Mensch definiert werden. Z.B.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

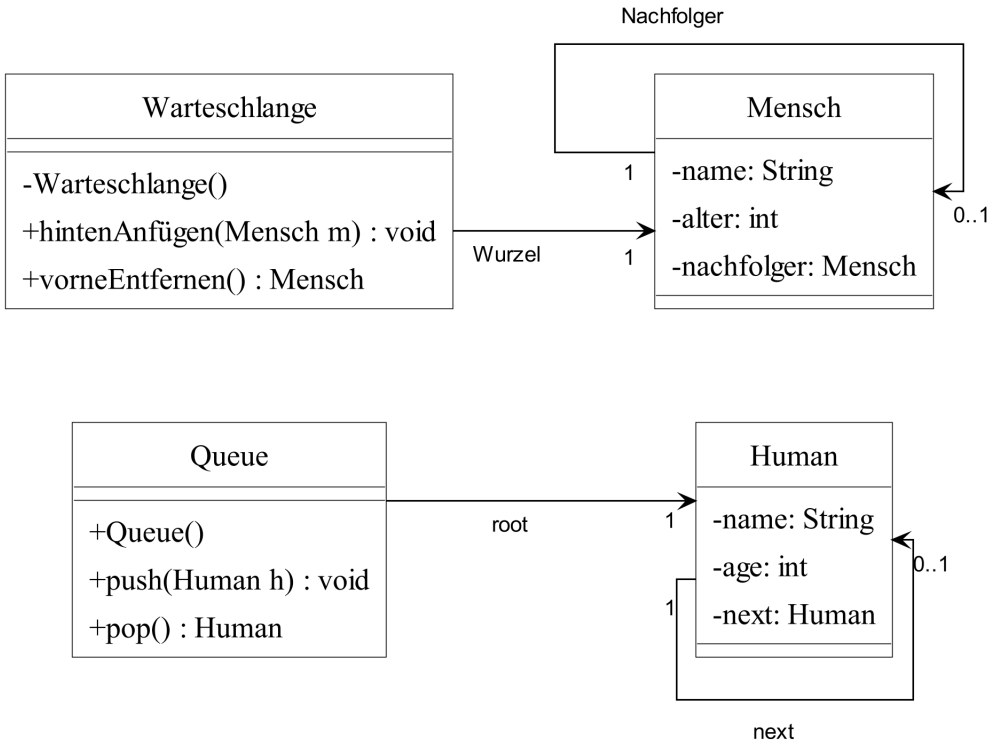
3 Die verkettete-Liste

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 (`https://de.wikipedia.org/wiki/Fibonacci-Folge`) 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:

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

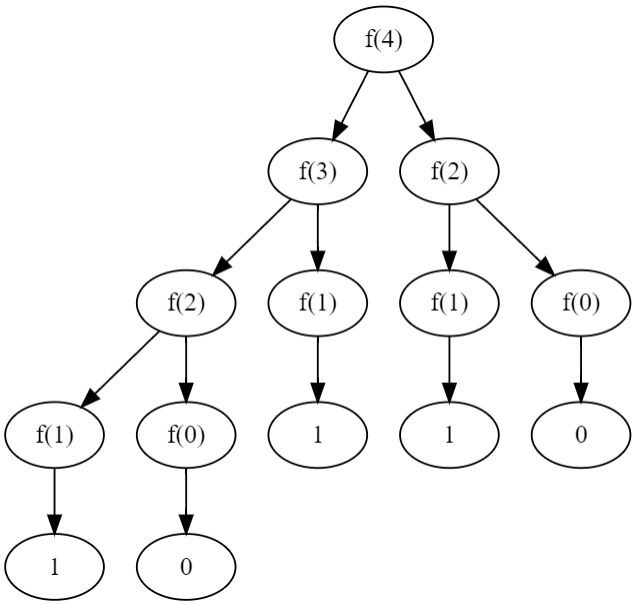
Ein iterativer Ansatz zur Berechnung der $n-$ten Zahl funktioniert mit Hilfe zweier Variablen, die die jeweils zwei letzten Ziffern speichern und daraus die neue berechnen:

```
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 obige Pyramide genauer an, so lässt sich erkennen, dass das Problem (die Berechnung von $f(4)$) schrittweise aufgeteilt wird. Dies geht nach obiger 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.

4 Anhang

4.1 Kapitel 2 - deutsche Übersetzungen

Übersetzungen aller Code-Schnipsel im Text: Code-Fragment 1:

```
int[] feld = new int[5];
```

Code-Fragment 2:

```
Mensch[] menschen = new Mensch[5]:
```

Code-Fragment 3:

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

Code-Fragment 4 - erste Definition der Klasse Mensch:

```
public class Mensch() {
    private String name;
    private int alter;

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

```
    }
```

Code-Fragment 5 - erste Definition der Klasse MeineListeFeld:

```
    public class MeineListeFeld(){

        private Mensch[] warteschlange;
        private int anzahl;

        public MeineListeFeld() {
            warteschlange = new Mensch[5];
            anzahl = 0;
        }

    }
```

Code-Fragment 6:

```
    public void hintenAnfügen(Mensch mensch)
```

Code-Fragment 7 - erste Implementierung von hintenAnfügen:

```
    public void hintenAnfügen(Mensch mensch) {
        for(int i = 0; i < warteschlange.length; i++) {
            if(warteschlange[i] == null) {
                warteschlange[i] = mensch;
            }
        }
    }
```

Code-Fragment 8 - zweite Version von hintenAnfügen:

```
    public class MeineListeFeld(){

        private Mensch[] warteschlange;
        private int anzahl;

        public MeineListeArray() {
            warteschlange = new Mensch[5];
            anzahl = 0;
        }

        public void hintenAnfügen(Mensch mensch) {
            warteschlange[anzahl] = mensch;
            anzahl++;
        }

    }
```

Code-Fragment 9 - finale Version von hintenAnfügen:

```
    public void hintenAnfügen(Mensch mensch) {
        if(anzahl == warteschlange.length) {
            warteschlangeVergroessern();
        }
        warteschlange[anzahl] = mensch;
        anzahl++;
    }

    private void warteschlangeVergroessern() {
```

```
        Mensch[] neueWarteschlange = new Mensch[warteschlange.length + 10];
        for(int i = 0; i < warteschlange.length; i++){
            neueWarteschlange[i] = warteschlange[i];
        }
        warteschlange = neueWarteschlange;
    }
```

Code-Fragment 10 - Implementierung von vorneEntfernen:

```
    public Mensch vorneEntfernen() {
        if(warteschlange[0] == null) {
            return null;
        }
        Mensch zuEntfernen = warteschlange[0];
        for(int i = 0; i < warteschlange.length -1; i++){
            if (warteschlange[i+1] == null){
                break;
            }
            warteschlange[i] = warteschlange[i+1];
        }
        warteschlange[warteschlange.length-1] = null;
        anzahl--;
        return zuEntfernen;
    }
```

4.2 Kapitel 2 - Aufgabenlösungen

Lösungen zu den Aufgaben (aktuell nur auf Englisch)
Hinweis: Alle Aufgaben sind hier direkt untereinander gelöst.

Zuerst die Mensch-Helferklasse

```
public class Human {

    private String name;
    private int age;

    /**
     * A constructor that specifies a human
     * @param name the name of the human
     * @param age the age of the human
     */
    public Human(String name, int age){
        this.name = name;
        this.age = age;
    }

    /**
     * getter method for the private attribute name
     * @return returns the name of the human
     */
    public String getName(){
        return name;
    }

    /**
     * getter method for the private attribute age
```

```
     * @return returns the age of the human
     */
    public int getAge(){
        return age;
    }

    /**
     * Helper method for testing, let's a human present the information
     */
    public void presentation(){
        System.out.println("I am " + name + " and I am " + age + " years old");
    }

    /**
     * Overrides the comparison method to check whether the human
     * has the same age and name as this object
     * @param o the compared object
     * @return returns a boolean stating if both objects are equal
     */
     @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;
         }
    }

    /**
     * Method for comparing two humans. Uses only the age to compare
     * @param human the human that shall be compared to this
     * @return returns true if this human is older
     */
    public boolean isGreater(Human human) {
        if(this.getAge() > human.getAge()) {
            return true;
        }
        return false;
    }
}
```

Grundlegende Methoden vor den Aufgabenlösungen:

```
public class MyListArray {

    private Human[] queue;
    private int count;

    /**
```

```
     * Constructor for the List, internally represented as an array
     */
    public MyListArray() {
        queue = new Human[5];
        count = 0;
    }

    /**
     * appends a new human.
     * @param human only objects of class Human can be appended
     */
    @Override
    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;
    }

    /**
    * removes the first element of the list, if present
    * @return a reference to the former first element of the list
    */
   @Override
   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;
   }
   /**
   * Helper getter method.
   * @return returns the current array that represents the list
   */
  public Human[] getQueue() {
      return queue;
  }
}
```

Aufgabe 3:

```
/**
 * Prints all Humans in the list, using the presentation function of Humans
 */
@Override
public void printList() {
    for(int i = 0; i < queue.length; i++){
        if(queue[i] == null) {
            break;
        }
        queue[i].presentation();
    }
}
```

Aufgabe 4:

```
/**
 * Finds and returns the Human at the specified position
 * Returns null, when there is no element!
 * @param position the position in the array (starts at 1!)
 */
@Override
public Human humanAtPosition(int position) {
    return queue[position-1];
}
```

Aufgabe 5:

```
/**
 * Searches a specified human in the queue and returns the position
 * @param searchValue an object of class Human that is being searched
 * @return returns -1, if the human is not in the list
 */
@Override
public int searchHumanInQueue(Human human) {
    for(int i = 0; i < queue.length; i++){
        if(queue[i].equals(human)){
            return i;
        }
    }
    System.out.println("The item is not in the list");
    return -1;
}
```

Aufgabe 6:

```
/**
 * Checks if a human is currently in the queue
 * @param human the human that is being looked for
 * @return the boolean that models the answer to the question
 */
@Override
public boolean contains(Human human) {
    for(int i = 0; i < queue.length; i++) {
        if(human.equals(queue[i])) return true;
    }
    return false;
```

```
        /* alternatively use searchHumanInQueue:
        if(searchHumanInQueue(human)>=0) return true;
        return false;
        */
    }
```

Aufgabe 7:

```
    /**
     * Removes the human at the given position (position starts at 1!)
     * @param position the position of the human to remove, starts at 1
     * @return returns a reference to the removed human
     */
    @Override
    public Human removeAtPosition(int position) {
        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:

```
    /**
     * Method to concatenate two lists.
     * @param o Object has to be of type MyList to concatenate
     * @return returns the new concatenated list or the old one, if o is not of type MyList
     */
    @Override
    public Object concatenate(MyListArray toConcat) {
        MyListArray newList = new MyListArray(this.getQueue().length
            + toConcat.getQueue().length);
        int counter = 0;
        for(int i = 0; i < this.length();i++) {
            if(queue[i] != null){
                counter++;
                newList.getQueue()[i] = queue[i];
            } else {
                break;
            }
        }
        for(int i = 0; i < toConcat.length(); i++){
            newList.getQueue()[counter + i] = toConcat.getQueue()[i];
        }
            return newList;
    }
```

Aufgabe 9:

```
    /**
     * appends the Human in an ascending order (compared by age).
     * @param human the human to append
     */
    public void appendSorted(Human human) {
        if(queue[0] == null){
```

```
            queue[0] = human;
            count++;
        } else {
            this.push(human);
            insertionSort()
        }
    }

    pprivate 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 < n; 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;
        }
    }
```