Übungen zu Algorithmen und Datenstrukturen für Medizintechnik
Sommersemester 2021
René Raab, Robert Richer

Lehrstuhl für Maschinelles Lernen und Datenanalytik
Department Artificial Intelligence in Biomedical Engineering (AIBE)
(https://www.mad.tf.fau.de)

Aufgabenblatt 7 vom 13. Juni 2021, Abgabe am 27. Juni 2021, 22:00 Uhr

---

Aufgabe 7.1: Laufzeitkomplexität bei Listenstrukturen

Punkte siehe StudOn
Komplexität, Listen

Aufgabenstellung und Abgabe (individuell, nicht als Gruppe!) im StudOn.

Aufgabe 7.2: Verkettete Liste

36 Punkte
Verkettete Listen

In dieser Aufgabe geht es darum, eine doppelt verkettete, generische, *aufsteigend* sortierte Liste sowie ein paar übliche Listenoperationen zu implementieren. Elemente in einfach verketteten Listen kennen nur ihren Nachfolger, in doppelt verketteten Listen kennt jedes Element zusätzlich seinen Vorgänger (siehe Abbildung 1). Die Vorgänger-Referenz des ersten Listenelements sowie die Nachfolger-Referenz des letzten Listenelements ist auf `null` gesetzt. Dazu haben wir Ihnen in der Datei `07-material.zip`, die Sie auf unserer StudOn-Seite herunterladen können, das Interface `AuDListInterface` bereitgestellt. In einer eigenen, abgeleiteten Klasse `AuDList` sollen Sie die entsprechenden Methoden implementieren.

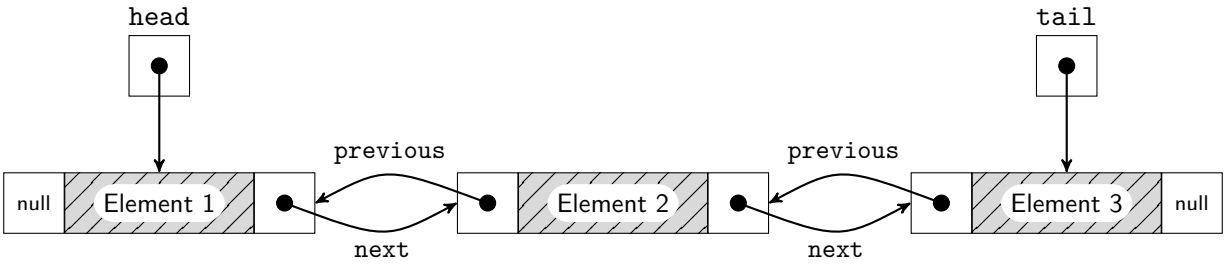

Abbildung 1: Schema einer doppelt verketteten Liste.

1. Erstellen Sie ein neues Java-Projekt `07-AuDList`.

2. Sie werden in dieser Aufgabe eine `Exception` benötigen, die noch nicht in Java vorhanden ist. Der Name dieser Exception soll `ElementExistsException` sein. Erstellen Sie zunächst die entsprechende Klasse. Sorgen Sie dafür, dass Ihre Exception von `RuntimeException` erbt. Implementieren Sie den Standardkonstruktor sowie einen Konstruktor mit einem `String` als Parameter für eine Fehlermeldung. Rufen Sie in den Konstruktoren den entsprechenden Konstruktor der Oberklasse auf.

3. Erstellen Sie die Klasse `AuDList` und lassen Sie sie von `AuDListInterface` erben. Lassen Sie IntelliJ IDEA alle abstrakten Methoden überschreiben[1].

[1] Mehr Informationen hier: https://www.jetbrains.com/help/idea/implementing-methods-of-an-interface.html

Hinweis: Um Einträge *generischer Datentypen* sortieren zu können, müssen diese miteinander vergleichbar sein. Achten Sie daher darauf, dass der generische Typ der Klasse `AuDList` – wie auch im Interface spezifiziert – vom Interface `Comparable<T>` erben muss. Klassen, die dieses Interface implementieren, müssen die `compareTo`-Methode spezifizieren, die festlegt, wie sich zwei Objekte dieser Klasse »vergleichen« lassen. Informieren Sie sich in der Java-API (https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html#compareTo-T-) über die korrekte Verwendung dieser Methode!

4. `AuDList` soll eine *doppelt verkettete* Liste implementieren. Legen Sie für die Repräsentation der Listen-Elemente innerhalb von `AuDList` eine private Klasse `ListItem` an.

   a) Neben einem Attribut `value`, in dem die generische »Nutzlast« des Listen-Eintrags gespeichert wird, soll die Klasse zwei Referenzen `previous` auf das vorhergehende und `next` auf das nachfolgende Element speichern können.

   b) `ListItem` soll zudem einen Konstruktor haben, dem der zu speichernde Wert übergeben wird.

5. Die Klasse `AuDList` selbst soll eine Referenz `head` auf das erste Listenelement sowie eine zweite Referenz `tail` auf das letzte Listenelement speichern.

   Erstellen Sie einen Konstruktor ohne Parameter, der `head` und `tail` mit `null` initialisiert.

   Achtung: Achten Sie im weiteren Verlauf der Aufgabe stets darauf, dass nach allen Operationen beide Attribute die korrekten Referenzen enthalten!

6. Implementieren Sie die Methoden der Klasse `AuDList` anhand der Spezifikationen in den JavaDoc-Kommentaren des Interfaces `AuDListInterface`.

   Im Folgenden erhalten Sie noch weitere Hinweise zur Implementierung der Methoden:

   Achtung: Testen Sie die von Ihnen implementierten Methoden unbedingt ausgiebig in der `main`-Methode!

   a) **`size`**
   Beginnen Sie am besten mit der Methode `size` ohne Parameter. Sie gibt einen `int`-Wert zurück, der die Länge der Liste (also die Anzahl der in der Liste gespeicherten Elemente) angibt. Überlegen Sie sich hierzu, wie Sie alle Einträge der Liste ablaufen können. Zählen Sie dabei mit, wie viele Einträge Sie bereits besucht haben.

   b) **`exists`**
   `exists(T value)` soll prüfen, ob der übergebene Wert bereits in der Liste vorhanden ist. Geben Sie, wenn der Wert enthalten ist `true`, ansonsten `false` zurück.

   c) **`insert`**
   Fahren Sie nun mit der Methode `insert` fort. Sie soll den übergebenen Wert an die korrekte Position in der Liste einfügen, sodass die Liste nach Einfügen des Elementes immer noch *aufsteigend* sortiert ist.

      i. Wenn der Wert bereits in der Liste enthalten ist, so fügen wir ihn nicht erneut ein, sondern werfen eine `ElementExistsException`.

      ii. Überlegen Sie sich nun – am besten kurz mit Zettel und Stift – wie man einen neuen Eintrag an beliebiger Stelle in die Liste einfügt und wie viele und vor allem welche Referenzen Sie eventuell »umbiegen« müssen. Nutzen Sie dazu beispielsweise Abbildung 1.

iii. Überlegen Sie sich nun, wie Sie die Position in der aufsteigend sortierten Liste finden, an der Sie das neue Element einfügen müssen. Implementieren Sie nun Code, mit dem Sie das entsprechende Element finden.

Hinweis: Wir schlagen Ihnen vor, dass Sie das Element in der Liste suchen, *vor* dem Sie das neue Element einfügen müssen. Nutzen Sie die `compareTo`-Methode, um festzustellen, wo Sie den neuen Eintrag einfügen müssen.

iv. Implementieren Sie nun das Einfügen vor den entsprechenden Eintrag.

Achtung: Es wird hierbei mehrere Sonderbehandlungen geben. Überlegen Sie sich, an welchen Positionen der Liste oder bei welcher Listengröße das Einfügen leicht verändert durchgeführt werden muss und welche weiteren Referenzen Sie eventuell manipulieren müssen.

Falls Sie möchten, können Sie Ihr Programm jetzt schon einmal testen. Legen Sie dazu eine `main`-Methode an und erstellen Sie in dieser eine neue Liste. Wählen Sie als Typ der Einträge zum Beispiel eine der Wrapper-Klassen von primitiven Datentypen (z.B. `Integer`, `Double`), fügen Sie verschiedene Werte ein und testen Sie die Methoden `exists` und `size`. Kommentieren Sie Ihre Tests und die `main`-Methode anschließend wieder aus.

d) `remove`
Diese Methode soll den übergebenen Wert – falls vorhanden – aus der Liste löschen:

i. Prüfen Sie zuerst, ob der übergebene Wert überhaupt in der Liste enthalten ist. Falls dies nicht der Fall ist, soll eine `NoSuchElementException` geworfen werden.

ii. Suchen Sie nun den zu löschenden Eintrag. Sie können auch hier wieder wie in den vorigen Methoden über die Liste laufen und mittels `compareTo` herausfinden, ob Sie den richtigen Eintrag gefunden haben.

iii. Wenn Sie den richtigen Eintrag gefunden haben, müssen Sie die Referenzen seines Vorgängers sowie Nachfolgers so setzen, dass der Eintrag nicht mehr Teil der Liste ist.

Achtung: Denken Sie besonders bei dieser Methode daran, alle Sonderfälle zu berücksichtigen!

Sie können nun Ihre Tests kurz wieder einkommentieren und die Funktion der `remove`-Methode testen. Kommentieren Sie Ihre Tests und die `main`-Methode anschließend wieder aus.

e) `iterator`
Als letzten größeren Punkt sollen Sie nun dafür sorgen, dass die Liste iterierbar wird. Hierzu benötigen Sie einen Iterator. Dieser erlaubt es, die Liste elementweise abzulaufen. Dies führt zum Beispiel dazu, dass die Liste in einer *for-each*-Schleife verwendet werden kann.

i. Sie benötigen für diese Teilaufgabe eine neue Klasse, die das Interface `Iterator<T>` implementiert. Benennen Sie diese Klasse `AuDListIterator`. Informieren Sie sich in den Übungsfolien und in der Java-Dokumentation über die korrekte Verwendung eines Iterators!

ii. Sie können Ihre Iterator-Klasse ebenso wie `ListItem` innerhalb von `AuDList` anlegen. Die Iterator-Klasse benötigt ein Attribut, in dem die *aktuelle* Position des Iterators in der Liste gespeichert wird, und zwar als Referenz auf ein `ListItem`. Legen Sie ein solches Attribut passenden Typs und passender Sichtbarkeit an. Nennen Sie es `nextElement`.

iii. Implementieren Sie den Standardkonstruktor für den Iterator. Wir wollen, dass die Liste ab ihrem Anfang durchlaufen wird. Womit müssen Sie `nextElement` also initialisieren?

iv. Anhand des Attributs `nextElement` können Sie dann die Methoden `hasNext` und `next` implementieren. Sie können davon ausgehen, dass die Liste nicht verändert wird, während der Iterator in Verwendung ist.

v. Schließlich müssen Sie in der Listenklasse nun die öffentliche Methode `iterator()` – welche bereits durch das `Iterable`-Interface vorgeschrieben wird – implementieren. Als einzige Handlung gibt diese einen neu erzeugten Iterator (nämlich einen `AuDListIterator`) zurück.

Sie können nun wieder die `main`-Methode einkommentieren und Ihren Iterator testen. Funktioniert eine *for-each*-Schleife mit Ihrer Liste? Kommentieren Sie die `main`-Methode und Ihre Tests anschließend wieder aus.

f) `merge`
Diese Methode bekommt als Parameter eine zweite Liste übergeben, die **nicht** verändert werden darf. In der Methode sollen die Elemente der zweiten Liste, die noch nicht in der ursprünglichen Liste vorhanden sind, in diese eingefügt werden, und zwar so, dass diese Liste weiterhin sortiert bleibt.

Hinweis: An dieser Stelle sollte Ihre Liste alle Funktionalitäten besitzen. Testen Sie alle Methoden Ihrer Liste **ausgiebig** und mit verschiedenen Datentypen (`Integer`, `String`, ...)!

7. Zu guter Letzt sollen Sie noch eine Anwendung für Ihre generische, sortierte Liste implementieren, nämlich eine vereinfachte Patienten-Datenbank.

a) Die Datenbank entspricht der Klasse `PatientDatabase`. Sie enthält eine Patientenliste, die als `AuDList` realisiert ist, sowie mehrere – im UML-Diagramm in Abbildung 2 beschriebene – Funktionen.

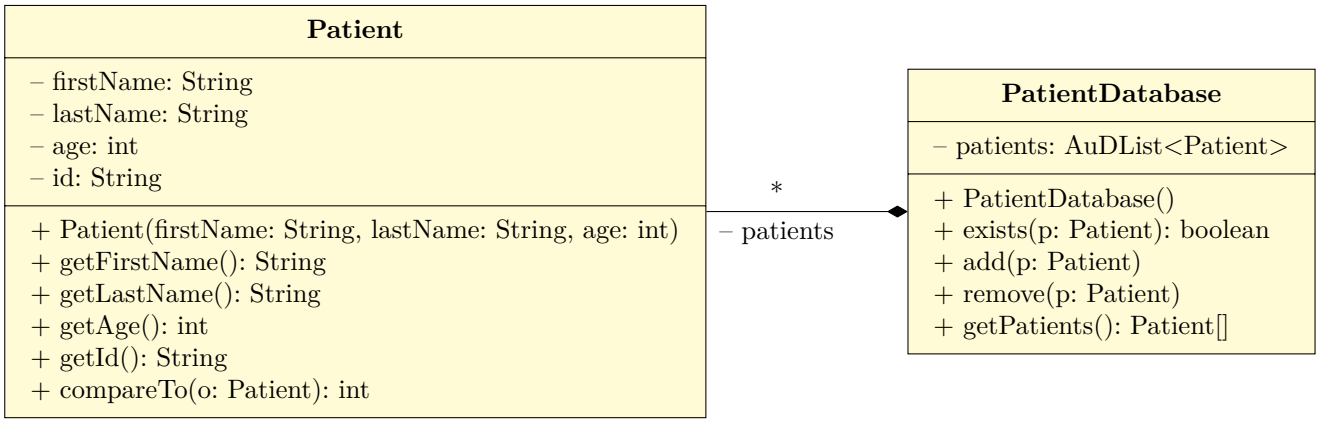

Abbildung 2: UML-Klassendiagramm der Patienten-Datenbank.

b) Natürlich soll die Datenbank dabei nicht »einfache« Elemente vom Typ `Integer` oder `String` speichern, sondern »Patienten«, also Objekten der Klasse `Patient`. Da die `AuDList` generisch ist, stellt dies aber kein Problem dar – solange die Klasse `Patient` das Interface `Comparable` implementiert. Wird dieses Interface von der Klasse implementiert, kann in der `compareTo`-Methode spezifiziert werden, wie sich zwei Objekte vom Typ `Patient` miteinander vergleichen lassen können.

c) Implementieren Sie zunächst die Klasse `Patient`:

i. Erstellen Sie eine Klasse `Patient`, die das Interface `Comparable<Patient>` implementieren soll. Lassen Sie IntelliJ die Methode `int compareTo(Patient o)` überschreiben (diese werden Sie dann später implementieren).

ii. Legen Sie alle Attribute, Methoden und Konstruktoren entsprechend der Spezifikation im UML-Diagramm in Abbildung 2 an und implementieren Sie die entsprechenden *getter*-Methoden.

iii. Die Patienten-ID (gespeichert im Attribut `id`) soll im Konstruktor aus dem Namen und dem Alter des Patienten generiert werden. Die ID soll dabei nach folgendem Schema vergeben werden:

⟨*lastName*⟩_⟨*firstName*⟩@⟨*age*⟩

iv. In der `compareTo`-Methode müssen Sie nun festlegen, wie zwei `Patient`-Objekte verglichen werden sollen. Dafür reicht es aus, die `id`s der beiden Objekte zu vergleichen, da die Klasse `String` bereits das Interface `Comparable` implementiert (⇝ `String.compareTo(String other)`).

d) Implementieren Sie nun die Klasse `PatientDatabase`:

i. Erstellen Sie eine Klasse `PatientDatabase` und legen Sie alle Attribute, Methoden und Konstruktoren entsprechend der Spezifikation im UML-Diagramm in Abbildung 2 an.

ii. Der Konstruktor soll die generische `AuDList patients` mit einer neuen Liste, die `Patient`-Objekte speichern können soll, initialisieren.

iii. `exists()` soll `true` zurückgeben, falls der übergebene Patient bereits in der `PatientDatabase` gespeichert ist, ansonsten `false`.

iv. `add()` soll einen `Patient` der Datenbank hinzufügen, falls er noch nicht enthalten ist. Ansonsten tut `add` nichts (wirft keine Exception!).

v. `remove()` soll einen `Patient` aus der Sammlung löschen, falls er in ihr enthalten ist. Ansonsten tut `remove()` nichts.

vi. `getPatients()` soll ein Array zurückgeben, das alle `Patients` der Datenbank enthält. Dieses Array soll natürlich so groß sein wie die Anzahl der `Patients`.

Hinweis: Sie können sich in dieser Teilaufgabe zunutze machen, dass Ihre Liste iterierbar ist!

Testen Sie nun Ihre `PatientDatabase` in der `main`-Methode der Klasse. Testen Sie ausgiebig alle Methoden von `PatientDatabase` und lassen Sie diese Tests stehen, wir werden diese auch bewerten!

8. Geben Sie die Dateien `AuDList.java`, `Patient.java`, `PatientDatabase.java` und `ElementExistsException.java` im EST ab!

Sollte Ihr Programm nicht übersetz- bzw. ausführbar sein, wird die Lösung mit 0 Punkten bewertet. Stellen Sie also sicher, dass IntelliJ IDEA keine Fehler in Ihrem Programm anzeigt, Ihr Programm übersetz- und ausführbar ist sowie die in der Aufgabenstellung vorgegebenen Namen und Schnittstellen *exakt* eingehalten werden. Geben Sie am Schluss die Dateien `AuDList.java`, `Patient.java`, `PatientDatabase.java` und `ElementExistsException.java` über die EST-Webseite ab. Wenn Sie die Aufgabe zusammen mit einem Übungspartner bearbeitet haben, geben Sie im EST unbedingt dessen Gruppenabgabe-Code an! Kontrollieren Sie, ob Ihre Namen am Anfang aller Dateien angegeben sind – schreiben Sie im Quellcode Ihre Angaben in einen Kommentar. Im EST-Abgabesystem können Sie modifizierte Dateien mehrfach abgeben. Nur die zuletzt hochgeladene Version wird bewertet.