Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Ulf Leser
M. Bux, B. Grußien, L. Heimberg, P. Schäfer

Berlin, den 08.05.2017

Übungen zur Vorlesung
Algorithmen und Datenstrukturen

Übungsblatt 2

Abgabe: Montag, den 22.05.2017, bis 11:10 Uhr vor der Vorlesung im Hörsaal oder bis 10:45 Uhr im Briefkasten neben Raum 3.321, RUD25.

Die Übungsblätter sind in Gruppen von zwei Personen zu bearbeiten. Jedes Übungsblatt muss bearbeitet werden. (Sie müssen mindestens ein Blatt für wenigstens eine Aufgabe jedes Übungsblattes abgeben.) Die Lösungen sind auf nach Aufgaben getrennten Blättern abzugeben. Heften Sie bitte die zu einer Aufgabe gehörenden Blätter vor der Abgabe zusammen. Vermerken Sie auf allen Abgaben Ihre Namen, Ihre **CMS-Benutzernamen**, Ihre Abgabegruppe (z.B. AG123) aus Moodle, und Ihren Übungstermin (z.B. Di 13 Uhr bei Marc Bux), zu dem Sie Ihre korrigierten Blätter zurückerhalten werden.

Beachten Sie die Informationen auf der Übungswebseite (`https://hu.berlin/algodat17`).

Aufgabe 1 (Sortierte Listen) — 4 + 2 + 2 + 2 + 2 + 4 = 16 Punkte

In dieser Aufgabe sollen Sie eine einfach verkettete, sortierte Liste zum Speichern von Integer-Werten in Java implementieren. Die Liste enthält einen Zeiger (headNode) auf den ersten Knoten, wobei dieser Zeiger in der leeren Liste auf null zeigt. Jeder Knoten der Liste speichert einen Integer-Wert und einen Zeiger auf den nachfolgenden Knoten. Falls kein Nachfolger existiert, so zeigt dieser Zeiger auf null. Da es sich um eine sortierte Liste handelt, gilt, dass der Wert jedes Knotens der Liste kleiner oder gleich dem Wert aller nachfolgenden Knoten ist. Die Liste soll auch doppelte Werte (Duplikate) speichern können.
Implementieren Sie die nachfolgenden Funktionen. Ergänzen Sie dazu den fehlenden Code in der Vorlage `SortedList.java`, welche Sie auf der Übungswebseite[1] vorfinden. Sie können beliebige neue Variablen und Hilfsmethoden zur Klasse hinzufügen.

- insert(value): Erstellt einen neuen Knoten mit Wert value und fügt ihn so in die Liste ein, dass die Sortierung erhalten bleibt. Duplikate können also direkt vor oder nach einem beliebigen Knoten mit identischem Wert eingefügt werden.
- removeFirst(): Entfernt den ersten Knoten (headNode) aus der sortierten Liste und liefert dessen Wert zurück. Der Wert entspricht also dem Minimum der Liste. Falls die Liste leer ist, soll stattdessen eine UndefinedValueException geworfen werden. Der headNode wird auf den Nachfolger des entfernten Knotens gesetzt.
- getLength(): Gibt die Länge der Liste zurück oder 0, falls die Liste leer ist. Um die volle Punkzahl zu erhalten, soll die Laufzeit Ihrer Lösung in $\mathcal{O}(1)$ liegen.
- getKLargest(k): Gibt den k-größten Wert der Liste zurück (ohne ihn zu entfernen), falls er existiert und die Liste nicht leer ist. Ansonsten wird eine UndefinedValueException geworfen. Die Indexierung beginnt bei 1 und der k-größte Wert ist der Wert des $(n-k+1)$-ten Knotens der sortierten Liste. Beispielsweise ist der Wert 4 in der Liste $[0, 1, 1, 4, 4, 6]$ sowohl das zweit- als auch das drittgrößte Element.

[1] `https://hu.berlin/algodat17`

- getMedian(): Gibt den Median der Liste zurück (ohne ihn zu entfernen). Falls die Anzahl der Knoten in der Liste ungerade ist, so ist der Median definiert als der Wert des Knotens, der an mittlerer Stelle steht. Wenn die Liste hingegen eine gerade Anzahl von Knoten enthält, ist der Median das arithmetische Mittel der Werte der beiden mittleren Knoten. Für eine leere Liste soll die Methode eine UndefinedValueException werfen.

- merge(otherSortedList): Fügt alle Knoten der übergebenen Liste otherSortedList sortiert in die Liste ein. Die übergebene Liste otherSortedList darf dabei beliebig verändert werden. Um die volle Punkzahl zu erhalten, soll die Laufzeit Ihrer Lösung in $\mathcal{O}(m + n)$ liegen, wobei m und n die Längen der beiden Listen sind.

Stellen Sie sicher, dass alle Testfälle in der main-Methode der Datei SortedList.java erfolgreich durchlaufen. Achten Sie außerdem auf Randbedingungen und Spezialfälle, die von den Testfällen vielleicht nicht vollständig abgedeckt werden.
Hinweis zur Abgabe: Ihr Java-Programm muss auf dem Rechner *gruenau2* laufen. Kommentieren Sie Ihren Code, sodass die einzelnen Schritte nachvollziehbar sind. Die Abgabe des von Ihnen modifizierten Source-Codes `SortedList.java` erfolgt über Moodle.

Aufgabe 2 (Stacks und Queues) **6 + 6 = 12 Punkte**

In dieser Aufgabe sollen Sie zwei Algorithmen entwerfen, die mit Stacks und Queues arbeiten. Bei der Verwendung eines Stacks oder einer Queue stehen Ihnen nur die für Stacks und Queues in der Vorlesung kennengelernten Methoden (also `enqueue()`, `dequeue()`, `head()` und `isEmpty()` für Queues bzw. `push()`, `pop()`, `top()` und `isEmpty()` für Stacks) zur Verfügung.
Ein Palindrom ist eine Zeichenkette, die von vorne und von hinten gelesen dasselbe ergibt. Beispiele hierfür sind die Zeichenketten

- „anna“,
- „rentner“,
- „erikafeuertnuruntreuefakire“ oder
- „tsssvssst“.

Anagramme sind Zeichenketten, die durch Umstellung der Buchstaben (Permutation) ineinander umgewandelt werden können. Beispiele hierfür sind die Zeichenketten

- „regal“, „lager“ und „aeglr“,
- „angstbude“, „bundestag“, „dantesbug“ und „abdegnstu“ oder
- „wolfgangamadeusmozart“ und „afamousgermanwaltzgod“.

1. Entwerfen Sie einen Algorithmus in Pseudocode, der als Eingabe eine Zeichenkette der Länge n über dem Alphabet $\Sigma = \{a, b, \ldots, z\}$ erhält und in Laufzeit $\mathcal{O}(n)$ testet, ob es sich bei dieser Zeichenkette um ein Palindrom handelt.

   Die Zeichenkette ist dabei als Queue q von Buchstaben realisiert, d.h., Sie haben zunächst nur Zugriff auf den ersten Buchstaben der Zeichenkette. Abgesehen von der Eingabe q, die beliebig modifiziert werden kann, darf Ihr Algorithmus lediglich elementare Datentypen sowie entweder einen zusätzlichen Stack oder eine zusätzliche Queue verwenden. Notieren Sie Ihren Algorithmus in Pseudocode und begründen Sie, warum seine Laufzeit in $\mathcal{O}(n)$ liegt.

2. Entwerfen Sie einen Algorithmus in Pseudocode, der als Eingabe eine Zeichenkette der Länge n über dem Alphabet $\Sigma = \{a, b, \ldots, z\}$ erhält und in Laufzeit $\mathcal{O}(n)$ testet, ob diese Zeichenkette ein Anagramm eines Palindroms ist, d.h., ob sich aus dieser Zeichenkette durch Permutation der Buchstaben irgendein Palindrom erzeugen lässt. Dieses Palindrom muss kein sinnvolles Wort sein. Ihr Algorithmus muss das Palindrom auch nicht ausgeben, es reicht ein **true** oder **false** als Ausgabe.

   Auch hier ist die Zeichenkette als Queue q von Buchstaben realisiert. Abgesehen von der Eingabe q, die beliebig modifiziert werden kann, darf Ihr Algorithmus lediglich elementare Datentypen sowie entweder einen zusätzlichen Stack oder eine zusätzliche Queue verwenden. Notieren Sie Ihren Algorithmus in Pseudocode und begründen Sie, warum seine Laufzeit in $\mathcal{O}(n)$ liegt.

Aufgabe 3 (Die böse Stiefmutter) **6 + 6 = 12 Punkte**

1. Um Schneewittchen zu vergiften, hat die böse Stiefmutter einen Apfel vergiftet. Leider hat ihr Putzmann diesen vergifteten Apfel zurück in die Apfelkiste gelegt. Nun liegen in der Kiste n Äpfel, die allesamt identisch aussehen, von denen aber einer giftig ist. Glücklicherweise weiß die böse Stiefmutter, dass der giftige Apfel wegen des Gifts 101 Gramm wiegt und damit etwas schwerer als die anderen Äpfel ist, die jeweils 100 Gramm wiegen. Außerdem besitzt sie eine Waage mit zwei Waagschalen, mit der sich überprüfen lässt, ob zwei Mengen von Äpfeln gleich viel wiegen. Entwerfen Sie einen Algorithmus zur Bestimmung des giftigen Apfels mit $\mathcal{O}(\log n)$ Wiegeoperationen. Notieren Sie Ihren Algorithmus in Pseudocode und begründen Sie, warum er mit $\mathcal{O}(\log n)$ Wiegeoperationen auskommt.

2. Um auch noch die sieben Zwerge zu vergiften, hat die böse Stiefmutter eine ganze Kiste mit 16 vergifteten Äpfeln gefüllt. Leider sorgt ihr Putzmann schon wieder für Probleme: Er hat die Kiste mit den vergifteten Äpfeln zwischen einigen anderen Apfelkisten in den Keller gestellt. Nun stehen dort insgesamt zehn Kisten, welche jeweils 16 identisch aussehende Äpfel enthalten, wobei genau eine Kiste giftige Äpfel zu je 101 Gramm enthält, während die anderen Kisten normale Äpfel zu je 100 Gramm enthalten. Da die böse Stiefmutter ihre Waage gerade verliehen hat, fragt sie ihren Spiegel um Rat. Dieser ist allerdings leider nicht so allwissend, wie er im Märchen beschrieben wird. Er ist aber immerhin dazu in der Lage, das exakte Gewicht einer ihm präsentierten Menge von Äpfeln zu nennen. Beschreiben Sie, wie die böse Stiefmutter die Kiste mit den giftigen Äpfeln bestimmen kann, ohne den allwissenden Spiegel mehr als einmal zu verwenden (es ist kein Pseudocode nötig).

Aufgabe 4 (Stacks) **10 Punkte**

In dieser Aufgabe geht es um das Auswerten von arithmetischen Ausdrücken, welche ausschließlich aus Ziffern, öffnenden und schließenden Klammern und den Operatoren $+$, $-$, $*$, / bestehen. Hierbei steht der Operator „/“ für Ganzzahldivision.
Wir definieren einen *wohlgeformten arithmetischen Ausdruck* wie folgt:

- Eine Ziffer $d \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ ist ein wohlgeformter arithmetischer Ausdruck.
- Sind X und Y wohlgeformte arithmetische Ausdrücke, dann sind auch die vier Ausdrücke $(X + Y)$, $(X - Y)$, $(X * Y)$ und (X/Y) wohlgeformte arithmetische Ausdrücke.

Beispiele für wohlgeformte arithmetische Ausdrücke entsprechend dieser Definition sind unter anderem: „$((8 + 7) * 2)$“, „$(4 - (7 - 1))$“ oder „8“.
Bei den Ausdrücken „$)8+)1(())$“, „$(8 + ())$“, „-1“, „$(\quad 5 \quad - 7)$“, „108“ oder „(8)“ handelt es sich hingegen nicht um wohlgeformte arithmetische Ausdrücke.
Schreiben Sie ein Java-Programm, welches für eine übergebene Zeichenkette überprüft, ob es sich um einen wohlgeformten arithmetischen Ausdruck handelt. Falls es sich um einen wohlgeformten arithmetischen Ausdruck handelt, so soll Ihr Algorithmus das Ergebnis dieses Ausdrucks ausrechnen und als Integer-Wert zurückgeben. Andernfalls soll eine ExpressionNotWellFormedException geworfen werden. Für diese Aufgabe bietet sich die Verwendung eines Stacks (java.util.Stack) an. Beim Ausrechnen des Ergebnisses brauchen Sie sich nicht um Divisionen durch 0 sowie arithmetische Überläufe zu kümmern.
Ergänzen Sie den fehlenden Code in der Vorlage `Parser.java`, welche Sie auf der Übungswebseite[2] vorfinden. Sie können beliebige neue Variablen und Hilfsmethoden zur Klasse hinzufügen, dürfen jedoch keine außer den von Java bereitgestellten Standard-Bibliotheken verwenden. Stellen Sie sicher, dass alle Testfälle in der main-Methode der Datei Parser.java erfolgreich durchlaufen. Achten Sie außerdem auf Randbedingungen und Spezialfälle, die von den Testfällen vielleicht nicht vollständig abgedeckt werden.
Hinweis zur Abgabe: Ihr Java-Programm muss auf dem Rechner *gruenau2* laufen. Kommentieren Sie Ihren Code, sodass die einzelnen Schritte nachvollziehbar sind. Die Abgabe des von Ihnen modifizierten Source-Codes `Parser.java` erfolgt über Moodle.

[2] https://hu.berlin/algodat17