06.04.2011
Teamname: Tugend und Laster

Vorgehensweise und Entscheidungsfindung

class LinkedList

Die Liste ist einfach verkettet mit head und tail Element, welche selber keine Daten enthalten. Die Implementierung ist mit antizipativer Indizierung. Um Operationen zu sparen, wird die Suche mit Stopper-Element verwendet.
Die Methode toString(), die wir überschrieben haben dient zu Testzwecken, man kann so die Liste, bzw. die Elemente, sehr einfach auf der Konsole ausgeben.
Bei der concat Methode waren wir etwas unschlüssig, was mit den Listen nach der Konkatenation passieren soll. Wenn wir Liste1 mit Liste2 verbinden, dann müsste Liste1 = Liste1 + Liste2 sein. Was aber geschieht mit Liste2? Zunächst wollten wir die Liste kopieren, damit die Liste2 weiterhin erhalten bleibt. Dann müssten wir aber elementweise die einzelnen Knoten kopieren, was wieder abhängig von der Listengröße wäre. Damit wäre das 0(1) nicht mehr gegeben. Wir haben so implementiert, dass wenn die beiden Listen verbunden sind, die nicht mehr benötigten Dummy-Elemente nicht mehr referenziert werden, also auf null gesetzt werden. Damit ist nur noch die Liste1 als Liste1+Liste2 verfügbar und Liste2 ist so nicht mehr nutzbar.
Die Fehlerbehandlung fiel uns ebenfalls bei der etwas ungenau gestellten Aufgabe und den recht knapp gehaltenen Informationen zu der antizipativen Indizierung schwer. Bei einigen Situationen, so die Antwort auf unsere Mail, sollen wir davon ausgehen, dass der User gültige Positionen/Elemente übergibt. Würde der User nun doch fehlerhafte Übergaben tätigen, könnten wir z.B. bei insert und delete ein Problem mit unserem Listenzähler bekommen.
Wir sind nun davon ausgegangen, dass der User gültige Referenzen/Positionen liefert.

interface List

Dieses Interface enthält die geforderten Methoden, wie sie im Skript beschrieben sind.
Weiterhin haben wir eine Methode insert(T element) hinzugefügt, um einfach Elemente am Ende einzufügen. Intern wird hier jedoch die Methode insert(Node<T> pos, T element) aufgerufen.
Zusätzlich um Abfragen der Listengröße haben wir die Methode size() hinzugefügt und unter anderem für die JUnit Tests noch die isEmpty() Methode.

class Node

Die Klasse Node repräsentiert einen Knoten für unsere Listenimplementierung. Es gibt ein Feld für Daten und einen Zeiger auf das nächste Feld.

class Benchmark

Die Klasse Benchmark ist eine sehr simple Helferklasse, mit der wir die Operationen zählen. Durch das Schlüsselwort static können wir einfach den Zähler erhöhen, zurücksetzen und auslesen.

class ListTest

In dieser Klasse sind die JUnit Tests enthalten. Die Methoden sind nach dem enthaltenen Test benannt.

class Test

In der Klasse Test haben wir einige Tests für uns selbst untergebracht. Diese Klasse hat keinen Einfluss auf die Liste selbst.
Unter anderem haben wir hier die Berechnung der elementaren Ops untergebracht und auf der Ausgabe basierend unsere Grafiken erstellt.

Bei Teil 2 des Theorieabschnitts hatten wir leider Verständnisprobleme. Theoretisch ist die antizipative Implementierung von uns immer schneller, als eine analoge Implementierung aus java.util. Ausser beim find, dort ist der Aufwand bei beiden 0(1). Dadurch, dass wir in unserer Implementierung immer mit Referenzen arbeiten, können wir schnell Elemente einfügen und löschen. Lediglich beim find müssen auch wir vollständig durch die Liste laufen.

Zu der Aufgabe des praktischen Vergleichs haben wir auch eine Frage via eMail geschickt, aber leider bisher keine Antwort erhalten. Die einzige Idee, die wir hatten, um die Aufgabe zu lösen wäre eine Zeitmessung/Speichermessung.
Wir würden zunächst die Methoden der einen Implementierung untersuchen z.B. bei 50000 Iterationen und im Anschluss unter gleichen Bedingungen die Implementierung von java.util
Wir werden die Frage morgen im Praktikum nochmals ansprechen.

Martin Slowikowski
Matrikelnummer: 199 91 66

Tell Mueller-Pettenpohl
Matrikelnummer: 198 99 82

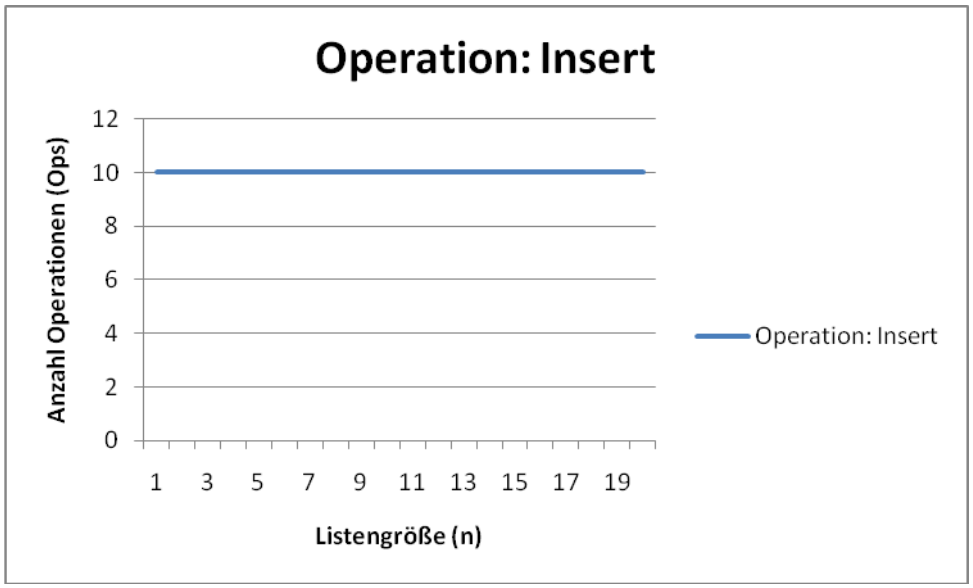
Teamname: Tugend und Laster

Quellenangaben

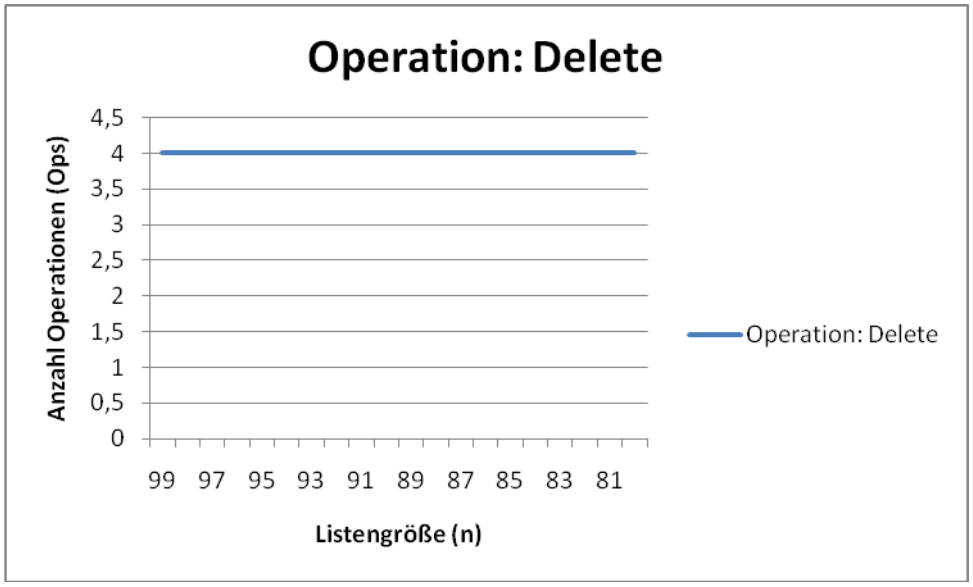
TIB3-AD-skript.pdf
Letzte Änderung 22.03.2011

Theorieteil 1

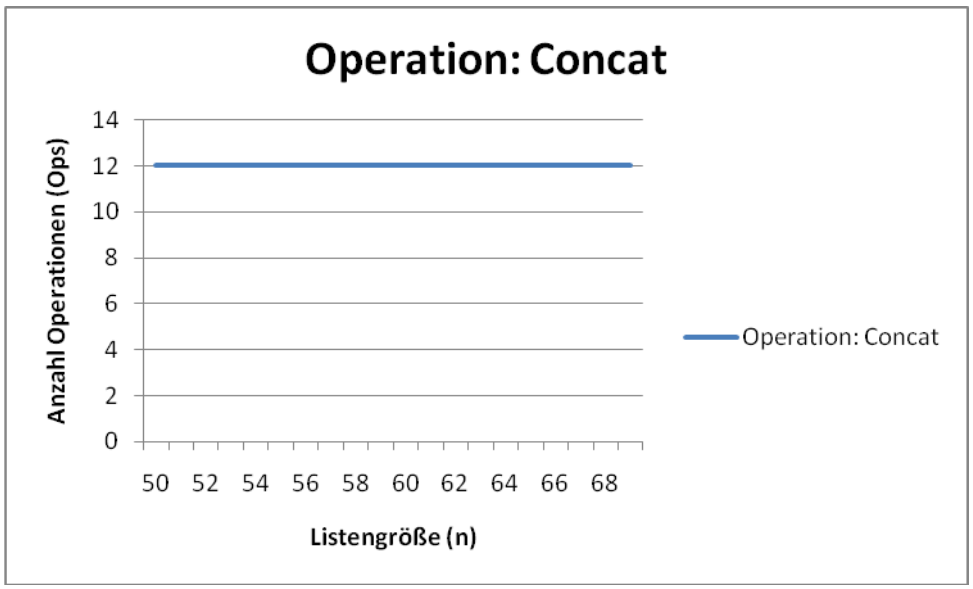
In den folgenden Diagrammen ist der Aufwand (in Form von Operationen) in Abhängigkeit der Listengröße (in Form der Anzahl der Elemente in der Liste) grafisch dargestellt.



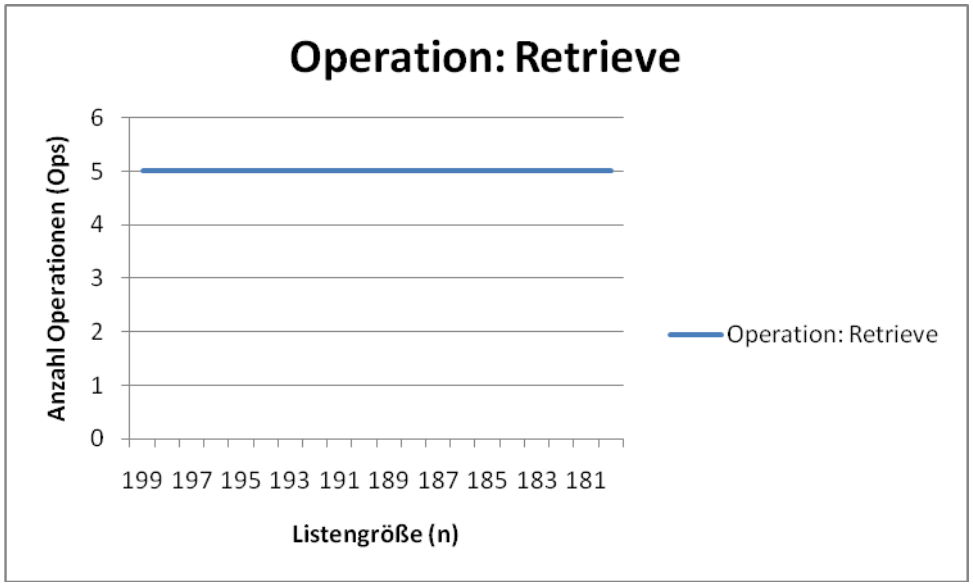
Anhand der Grafik ist zu erkennen, dass die Anzahl der Operationen beim Einfügen eines Elements in die Liste, nicht abhängig von der Listengröße ist. Es gilt daher ein Aufwand von O(1).



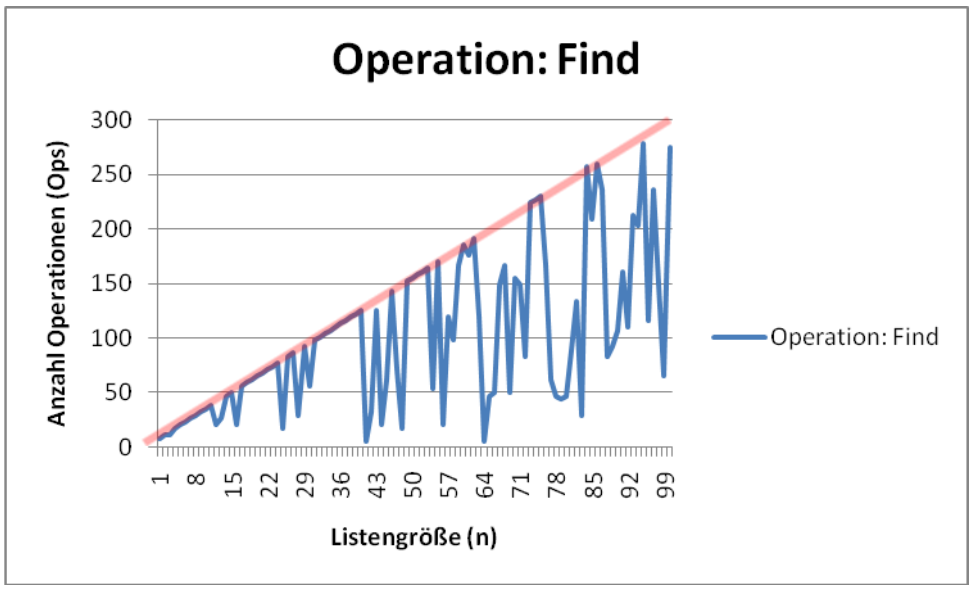
Wie beim Insert besteht beim Delete ein Aufwand von O(1). Die Anzahl der Operationen beim Löschen hängt nicht ab von der Listengröße.



Auch beim Concat besteht ein konstanter Aufwand von O(1). Die Anzahl der Operationen beim Konkatenieren hängt nicht ab von der Listengröße.



Beim Retrieve besteht auch ein Aufwand von O(1). Die Anzahl der Operationen ist nicht abhängig von der Listengröße.
Getestet wurde das Retrieve mit zufälligen, gültigen Referenzen.



Anhand der Grafik ist zu erkennen, dass mit steigender Anzahl an Elementen auch die Anzahl der Operationen beim Find steigt. Es ergibt sich ein linearer Verlauf, verdeutlicht durch den roten Graph, also ein O(N).
Der Methode find wurden Elementen in zufälliger Reihenfolge übergeben, durch diesen Sachverhalt ergeben sich die Unregelmässigkeiten unterhalb des linearen Verlaufs.