Informatik II: Algorithmen und Datenstrukturen SS 2013

Vorlesung 7a, Dienstag, 4. Juni 2013
(Verkettete Listen, Vergleich mit dynam. Feldern)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Ihre Erfahrungen mit dem Ü6 (Dynamische Felder)
  - Erläuterung zur Musterlösung dazu
- Verkettete Listen
  - Erklärung + Beispiel + Implementierung
  - Laufzeitanalyse (trivial)
  - Laufzeitvergleich mit dynamischem Feld
  - Übungsblatt 7, Aufgabe 1: Implementierung der Liste um Methoden remove, size und get erweitern

Erfahrungen mit dem Ü6 (Dyn. Felder)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 4. Juni 15:30
  - Etwas Verwirrung durch Unterschied letzte Implementierung und was dann analysiert wurde … wurde im Forum geklärt
  - Die Schwierigkeit war, herauszufinden, um wie viel vergr. und wann um wie viel verkleinern … siehe nächste Folie
  - Wie viele Übungsblätter insgesamt? 12
  - Analyse Anzahl Teilnehmer:

    Danke für die Antworten !

    Viele meinten, der hier ist schuld: http://goo.gl/VjyXl

    Die meisten finden die VL wohl ganz gut, aber haben nicht genug Zeit dafür (aus verschiedensten Gründen)

Musterlösung Ü6 (Dyn. Felder)

- Allgemeines Prinzip
  - Bei append, wenn Feld ganz voll, dann Vergrößern um Faktor $X > 1$ ... in der Analyse in der VL war $X = 2$
  - Bei removeLast, wenn Feld $\leq 1/Y$ voll, dann verkleinern auf $1/Z$ ... in der Analyse in der VL war $Y = 4$ und $Z = 2$

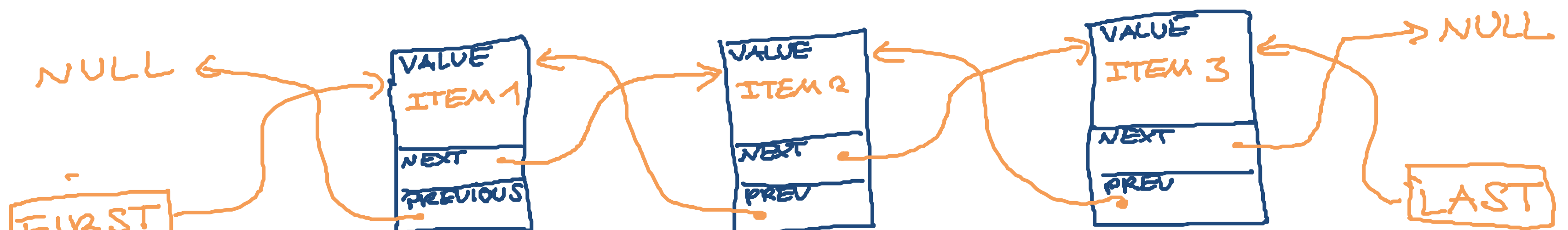
    Bedingung 1: $Y > Z$, damit man nach einer Verkl. gefolgt von einem append nicht gleich wieder vergrößern muss

    Bedingung 2: $Y > X$, damit man nach einer Vergr. gefolgt von einem removeLast nicht gleich wieder verkleinern muss
  - Der Beweis in der VL funktioniert für beliebige $X, Y, Z > 1$ mit $Y > Z, X$ ... die Faktoren im Beweis ändern sich natürlich

    Dabei kann man immer $Z = X$ wählen, das macht den Beweis einfacher ... zum Beispiel $X = 1.1$, $Y = 1.2$, $Z = 1.1$

Verkettete Listen 1/6

- Grundprinzip
  - Wir betrachten hier **doppelt** verkettete Listen
  - Jedes Element kennt seinen Vorgänger (previous) und seinen Nachfolger (next)
  - Außerdem kennt man Anfang und Ende der Liste
  - Das ermöglicht uns Einfügen und Löschen in O(1) Zeit
  - **Java:** java.util.LinkedList<T> **C++:** std::list<T>



in C++ sind das tatsächlich Zeiger
in JAVA Referenzen

in beiden Fällen quasi die Speicher-adresse des Objektes

Verkettete Listen 2/6

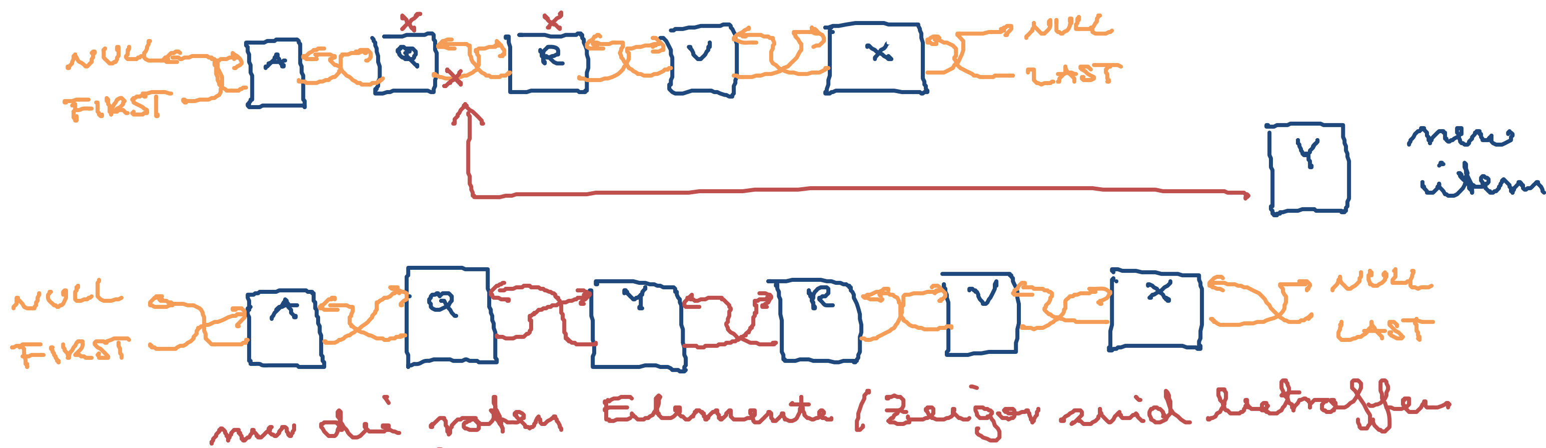
- Einfügen (insert) eines neuen Elementes
  - Im Prinzip nicht schwer: man muss die betroffenen "Zeiger" (in Java: Referenzen) richtig "umbiegen"

    Den Nachfolgerzeiger vom Vorgänger

    Den Vorgängerzeiger vom Nachfolger

    Die beiden Zeiger des eingefügten Elementes

    Evtl. die Zeiger auf das erste und das letzte Element



Verkettete Listen 3/6

UNI FREIBURG

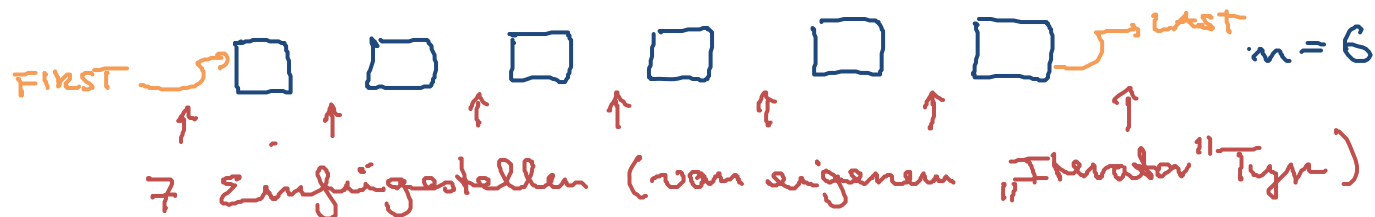
- Angabe der Einfügestelle
  - **Variante 1:** Referenz auf den neuen Nachfolger nextItem

    Dann nennt man die Operation oft insertBefore

    Einfügen am Ende, indem man nextItem = null übergibt
  - **Variante 2:** Referenz auf den neuen Vorgänger previousItem

    dann nennt man die Operation oft insertAfter

    Einfügen am Ende, indem man previousItem = null übergibt
  - **Variante 3:** Über einen zusätzlichen Typ ListIterator, der auf die n + 1 möglichen Stellen in einer Liste mit n Elementen zeigt … so wird es in Java und C++ gemacht



Verkettete Listen 4/6

UNI FREIBURG

- Entfernen (remove) eines geg. Elementes item
  - Auch hier müssen einfach nur die richtigen "Zeiger" "umgebogen" werden

    Nachfolgezeiger des Vorgängers von item

    Vorgängerzeiger des Nachfolgers von item

    Evtl. die Zeiger auf das erste und das letzte Element

Verkettete Listen 5/6

UNI FREIBURG

- Anzahl der Elemente
  - Ohne Weiteres muss man einmal von vorne nach hinten durch die Liste gehen, um die Anzahl Elemente zu ermitteln

    Laufzeit dafür $\Theta(n)$, wenn n = Anzahl Elemente
  - Man kann aber einfach eine Membervariable size einführen

    ... zu Beginn 0

    ... bei jedem insert um 1 erhöhen

    ... bei jedem remove um 1 verringern
  - Dann bekommt man die Anzahl Elemente in $O(1)$ Zeit

Verkettete Listen 6/6

UNI FREIBURG

- Zugriff auf das i-te Element
  - Das geht in einem Feld immer in Zeit $O(1)$

    weil die Elemente alle hintereinander im Speicher stehen

    Anfangsadresse + i * Größe eines Elementes
  - In einer verketteten Liste können die Elemente **beliebig** im Speicher verteilt stehen
  - Will man das i-te Element haben, bleibt einem nichts anderes übrig, als sich von vorne durchzuhangeln

    oder von hinten, falls $i > n/2$

    Die Laufzeit dafür ist also $\Theta(\min\{i, n - i\})$

Laufzeit Listen vs. Felder 1/4

UNI FREIBURG

- Laufzeit doppelt verkettete Liste
  - Einfügen an beliebiger Stelle: $O(1)$
  - Entfernen an beliebiger Stelle: $O(1)$
  - Zugriff auf i-tes Element der Liste: $O(\min\{i, n - i\})$
- Laufzeit dynamisches Feld
  - Einfügen am Ende: amortisiert $O(1)$ … siehe letzte VL
  - Entfernen am Ende: amortisiert $O(1)$ … siehe letzte VL
  - Zugriff auf i-tes Element der Liste: $O(1)$
  - Einfügen an i-ter Stelle: $O(n - i)$
  - Entfernen an i-ter Stelle: $O(n - i)$

Laufzeit Listen vs. Felder 2/4

UNI FREIBURG

- Zeitmessung in der Praxis
  - Beim Einfügen / Entfernen am Ende scheinen Liste und (dynamisches) Feld gleich gut zu sein

    Die Liste sieht sogar besser aus, weil **immer** O(1) und das dynamische Feld nur **im Durchschnitt** O(1)
  - Die Laufzeit wollen wir jetzt mal konkret nachmessen
  - Beobachtung: LinkedList **viel** langsamer als DynamicArray

    bei 100M Elementen, um einen Faktor etwa **20**

Laufzeit Listen vs. Felder 3/4

UNI FREIBURG

- Analyse gemessener Laufzeitunterschied
  - **Grund 1:** Bei der Liste müssen wir für jede Operation **vier** Zeiger umbiegen, das kostet Zeit

    Das wiegt deutlich schwerer als das gelegentlich notwendige Reallozieren beim dynamischen Feld

  - **Grund 2:** Bei der Liste müssen wir für jedes Element einzeln Speicher allo**z**ieren, beim dynamischen Feld tun wir das für viele Element auf einmal

    Der allo**z**ierte Speicher ist zwar am Ende der**s**elbe, aber Speicherallo**k**ation hat einen fixen Overhead pro Aufruf

Laufzeit Listen vs. Felder 4/4

UNI FREIBURG

- Es gibt noch einen Grund
  - Die sogenannte **Lokalität** der Speicherzugriffe
  - Bei einem Feld stehen ja, wie gesagt, die n Elemente im Speicher hintereinander
  - Bei einer verketteten Liste können die Elemente **beliebig** im Speicher verteilt stehen

    Dafür wollen wir jetzt in unserem Programm mal sorgen

    (wenn man sie direkt eins nach dem anderen alloziert, stehen sie wahrscheinlich auch direkt nacheinander)

Literatur / Links

UNI FREIBURG

- Doppelt verkettete Liste
  - In C++ und in Java

    http://www.cplusplus.com/reference/list/list/

    http://docs.oracle.com/javase/6/docs/api/java/util/LinkedList.html
  - Wikipedia

    https://en.wikipedia.org/wiki/Linked_list
- Rechtschreibunk
  - http://www.duden.de/suchen/dudenonline/allozieren
  - http://www.korrekturen.de/beliebte_fehler/der_selbe.shtml