Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 7a, Dienstag, 13. Juni 2017
(Verkettete Listen)

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

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Erfahrungen ÜB6 — Dynamische Felder
- Inhalt
  - Verkettete Listen — Prinzip + Implementierung
  - Listen vs. Felder — Was ist wann besser?
  - Cache-Effizienz — morgen in Vorlesung 7b
  - ÜB7: die Klasse LinkedList aus der Vorlesung um ein paar Operationen erweitern + deren Cache-Effizienz bestimmen

Erfahrungen mit dem ÜB6 1/3

- Zusammenfassung / Auszüge
  - Aufgabe 2 (Implementierung "Kilometerzähler") war für die meisten gut machbar

    Fehler in TIP: Uhrzeit ist kein Zähler im Sinne des ÜB6, wurde im Forum besprochen aber in TIP nicht korrigiert
  - Theorieaufgaben hilfreich für das Verständnis, allerdings haben es einige nicht gut verstanden bzw. aufgegeben

    Siehe Lösungsskizzen auf der nächsten Folie

    Hier bin ich ehrlich gesagt etwas ratlos: das Thema wurde in der VL6b sehr ausführlich und mit viel Intuition erklärt
  - Machine Learning ist viel spannender als so Basis-Zeug

    Aber keine Chance das zu verstehen ohne Basis-Zeug

Erfahrungen mit dem ÜB6 2/3

- Lösungsidee Aufgabe 1 (Feld immer $\geq f \cdot s$ voll)
  - Bei pushBack, wenn Feld ganz voll, dann $c_i = \lfloor A \cdot s_i \rfloor$
  - Bei popBack, wenn $s_i \leq c_{i-1} / B$, dann $c_i = \lfloor C \cdot s_i \rfloor$
  - Bedingungen: $A, B, C > 1$ (klar) und $B > C$ (damit keine Vergrößerung kurz nach Verkleinerung oder andersrum)
  - In der Analyse von Vorlesung 6b war $A = C = 2$ und $B = 4$

    Damit ist das Feld immer mindestens $1 / B = 1 / 4$ gefüllt
  - Für beliebiges f mit $0 < f < 1$:

    $B = 1 / f$ ... dann ist immer $s_i \geq f \cdot c_i$

    $C = (B + 1) / 2$ ... dann $1 < C < B$

    $A = C$ ... gibt keinen guten Grund für $A \neq C$

Erfahrungen mit dem ÜB6 3/3

- Lösungsidee Aufgabe 3 (Potenzialfunktion Zähler)
  - Sei **k′** die Anzahl Ziffern, die nach einem "increment" auf null zurückspringen … zum Beispiel 009999 → 01**0000**
  - Die Laufzeit einer increment Operation ist dann **O(k′+ 1)**

    Plus 1, weil es für k′ = 0 auch dauert
  - Das legt folgende Definition der Potenzialfunktion nahe

    Φ = Anzahl der Nullen des aktuellen Zählerstandes
  - Mit k wie oben definiert, erhöht sich dann nach einer Operation das Potenzial um mindestens k′ – 1

    Minus 1, weil man vorne evtl. eine Null verliert
  - Der Rest folgt dann aus dem Mastertheorem

Verkettete Listen 1/8

UNI FREIBURG

- Grundprinzip
  - Wir betrachten hier **doppelt** verkettete Listen
  - Jedes Element kennt seinen Vorgänger (previous) und seinen Nachfolger (next)
  - Außerdem kennt man den Anfang (first) und das Ende (last) der Liste
  - Das ermöglicht uns Einfügen und Löschen **an beliebiger Stelle** in O(1) Zeit

    Das können Felder nicht, siehe spätere Folie 12

    Beispiele von solchen Listen auf den nächsten Folien …

Verkettete Listen 2/8

PREV ≙ previous
NEXT ≙ next

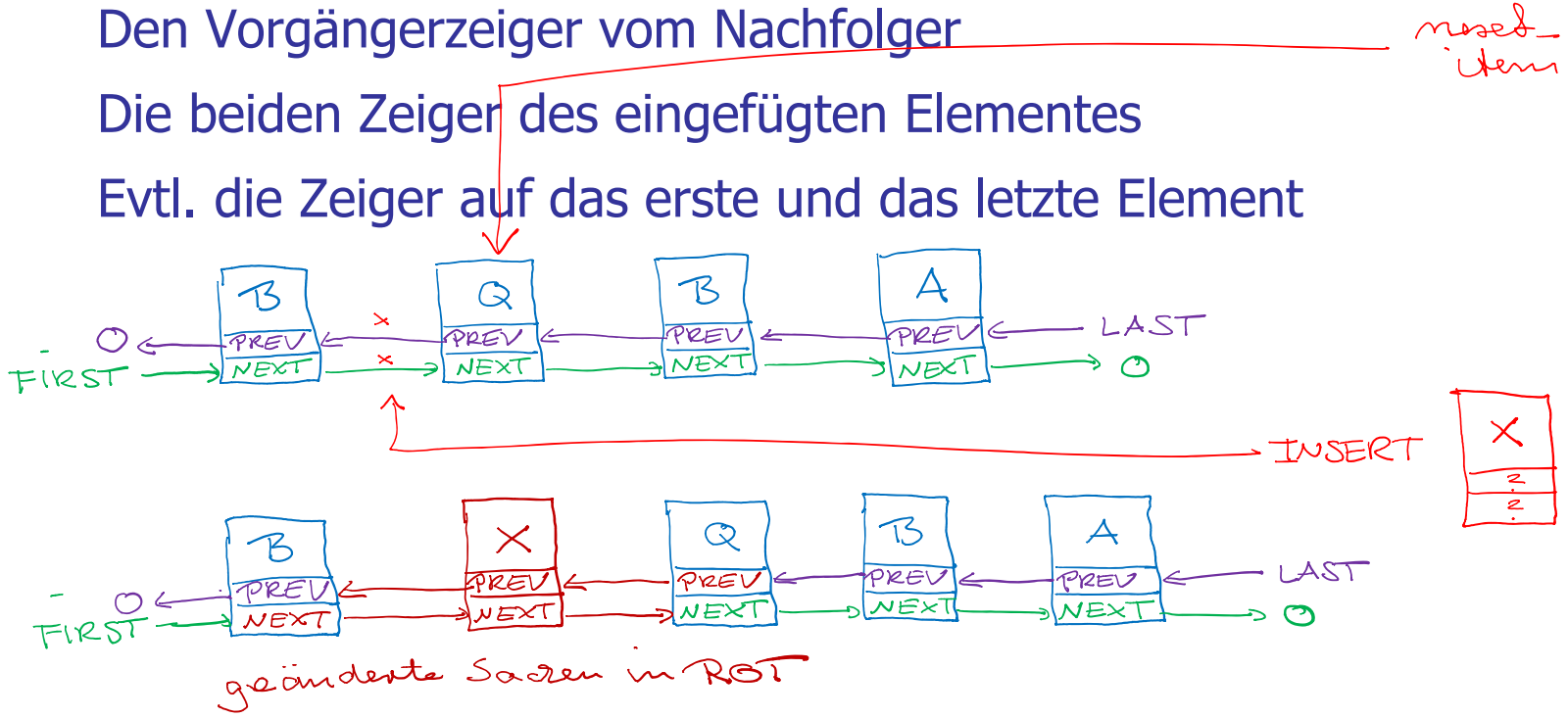
- Einfügen (insert) eines neuen Elementes
  - Im Prinzip nicht schwer: man muss nur die betroffenen Zeiger / Referenzen (siehe Folie 12) richtig "umbiegen"

    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 3/8

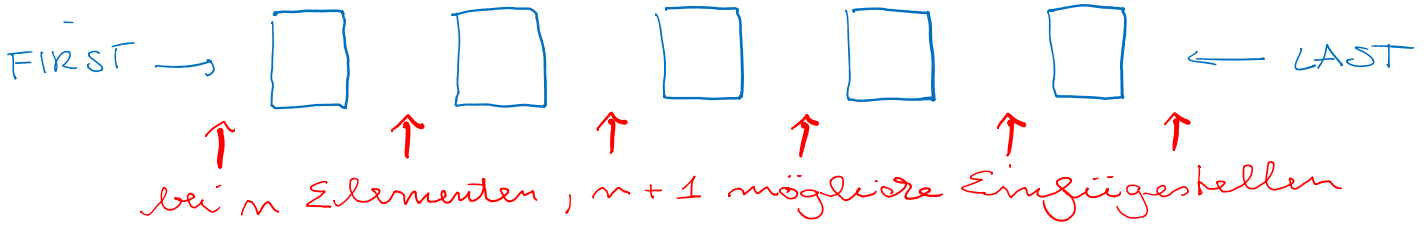
- Angabe der Einfügestelle
  - **Variante 1:** Referenz auf den neuen Nachfolger next_item

    Dann nennt man die Operation oft **insert_before**

    Einfügen am Ende, indem man next_item = 0 übergibt
  - **Variante 2:** Referenz auf den neuen Vorgänger prev_item

    Dann nennt man die Operation oft **insert_after**

    Einfügen am Anfang, indem man prev_item = 0 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



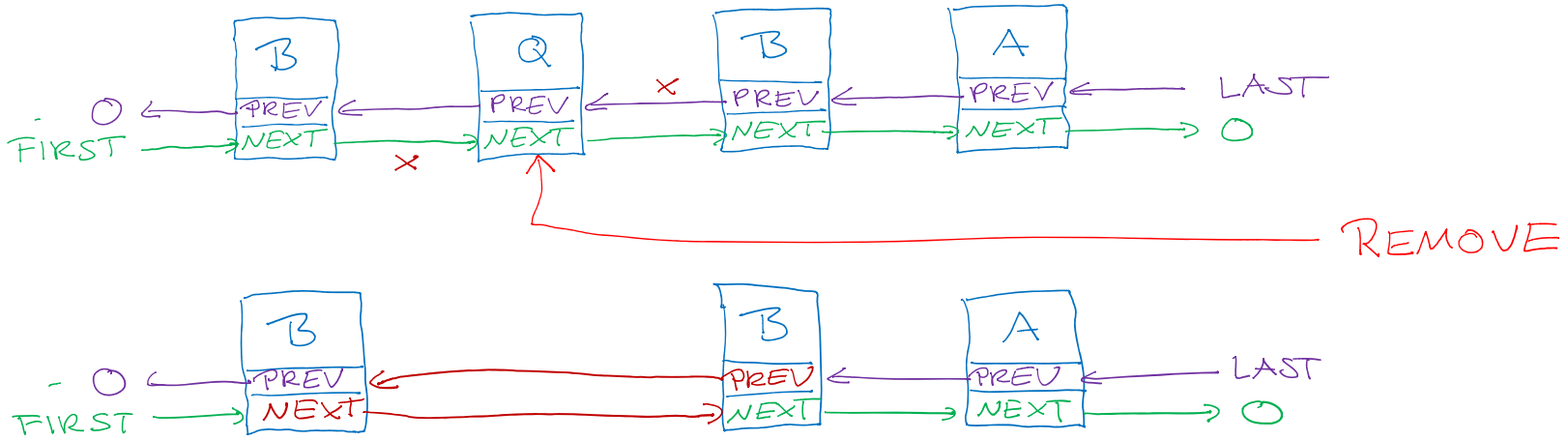
Verkettete Listen 4/8

- 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 5/8

UNI FREIBURG

- Implementierung
  - Das machen wir jetzt mal zusammen (in Python)

    Der Einfachheit halber nur mit Einfügen (insert_before)
  - Sobald man mit Zeigern hantiert, kann man sehr viele Fehler machen, die sehr schwer zu debuggen sind
  - Deswegen ist eine gute Funktion, die den aktuellen Zustand einer Liste anzeigt, hier entscheidend wichtig

    Das wird uns die Hälfte der Arbeit kosten, aber das ist in dem Fall hier absolut angemessen

    Das ist eine der take-home-messages für heute: Zeit für gute Funktionen zum Debuggen ist oft bestens investiert

Verkettete Listen 6/8

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
  - Es geht aber auch leicht in $O(1)$ Zeit

    Man kann ja einfach separat einen Zähler haben, und macht bei jedem insert +1 und bei jedem remove –1

Verkettete Listen 7/8

- Zugriff auf das i-te Element
  - 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 "durchzuhangeln"

    Von vorne oder von hinten, was immer schneller ist
  - Die Laufzeit dafür ist also **$\Theta(\min\{i, n - i\})$**
  - Zum Vergleich: in einem Feld stehen die Elemente immer garantiert hintereinander im Speicher
  - Von daher Zugriff auf das i-te Element in **$O(1)$** Zeit

    Steht an Stelle Anfangsadr. + i * Größe eines Elementes

Verkettete Listen 8/8

- In Java, C++ und Python
  - In Java: java.util.LinkedList<T> … in C++: std::list<T>
  - In Python gibt es keine native verlinkte Liste

    Einfügen an beliebiger Stelle kein häufiger Anwendungsfall

    Hier in der VL vor allem als Grundlage für Vorlesung 8a+b
  - Für die Vorlesung heute (und das Übungsblatt) wollen wir verkettete Listen **von Grund auf selbst** implementieren
  - Das Konzept des "Zeigers" gibt es in allen drei Sprachen:

    In C++ benutzt man wörtlich Zeiger … LinkedListItem*

    In Java und Python sind Namen von Objekten grundsätzlich Referenzen, also auch Zeiger … LinkedListItem

Laufzeit Listen vs. Felder 1/5

- Laufzeit doppelt verkettete Liste
  - Einfügen an beliebiger Stelle: $O(1)$ ✓
  - Entfernen an beliebiger Stelle: $O(1)$ ✓
  - Zugriff auf i-tes Element der Liste: $\Theta(\min\{i, n-i\})$ ✗
- Laufzeit dynamisches Feld
  - Einfügen am Ende: $O(1)$ amortisiert (= im Durchschnitt) – manchmal teuer (✓)
  - Entfernen am Ende: dito (✓)
  - Zugriff auf i-tes Element der Liste: $O(1)$ ✓
  - Einfügen an i-ter Stelle: $\Theta(n-i)$ ✗
  - Entfernen an i-ter Stelle: $\Theta(n-i)$ ✗

Laufzeit Listen vs. Felder 2/5

- 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 **amortisiert** O(1)
  - Die Laufzeit wollen wir jetzt mal konkret nachmessen

    Wir fügen dabei der Einfachheit halber nur am Ende ein
  - Beobachtung:

    In C++, std::list ca. 4 mal LANGSAMER als der std::vector

Laufzeit Listen vs. Felder 3/5

UNI FREIBURG

- Laufzeitunterschied, Grund 1
  - Bei der Liste müssen wir für jede Operation **vier** Zeiger umbiegen

    Beim dynamischen Feld müssen wir ohne Reallokation einfach nur **einen** Eintrag schreiben

    Und die Reallokationen fallen bei geeigneten Parametern (z.B. f = 0.5 im Sinne des ÜB6) nicht ins Gewicht

Laufzeit Listen vs. Felder 4/5

UNI FREIBURG

- Laufzeitunterschied, Grund 2
  - Bei der Liste müssen wir für jedes Element **einzeln** Speicher allo**z**ieren

    Beim dynamischen Feld tun wir das für viele Elemente **auf einmal**

    Jede Speicherallokation hat fixe Kosten, die unabhängig von der Größe des allozierten Speichers sind

    Außerdem benötigen wir durch die Zeiger pro Element auch insgesamt etwas mehr Platz

Laufzeit Listen vs. Felder 5/5

UNI FREIBURG

- Laufzeitunterschied, Grund 3
  - Das dynamische Feld hat eine viel bessere 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

    Warum das einen Unterschied macht, ist genau das Thema von VL7b

Literatur / Links

- Doppelt verkettete Liste
  - Wikipedia

    https://en.wikipedia.org/wiki/Linked_list
- Allozieren
  - http://www.duden.de/rechtschreibung/allozieren