Informatik II - SS 2014

(Algorithmen & Datenstrukturen)

Vorlesung 5 (13.5.2014)

Sortieren V, Abstrakte Datentypen

UNI FREIBURG

Fabian Kuhn

Algorithmen und Komplexität

Zusammenfassung Sortieralgorithmen

UNI FREIBURG

Laufzeiten Sortieralgorithmen

- **Selection Sort & Bubble Sort**

  worst case & best case: $\Theta(n^2)$

- **Insertion Sort**

  worst case (& avg. case): $\Theta(n^2)$, best case: $\Theta(n)$

- **Merge Sort**

  worst case (and best case): $\Theta(n \log n)$

- **Quick Sort**

  worst case (fixed pivot): $\Theta(n^2)$, average case: $O(n \log n)$
  worst case, randomized: $O(n \log n)$
  - Erwartungswert, hohe Wahrscheinlichkeit

Sortieren : Untere Schranke

UNI FREIBURG

Aufgabe: Sortiere Folge $a_1, a_2, \dots, a_n$

- Ziel: benötigte (worst-case) Laufzeit nach unten beschränken

Vergleichsbasierte Sortieralgorithmen

- Vergleiche sind die einzige erlaubte Art, die relative Ordnung von Elementen zu bestimmen
- Das heisst, das Einzige, was die Reihenfolge der Elemente in der sortierten Liste beeinflussen kann, sind Vergleiche der Art

$$a_i = a_j, a_i \leq a_j, a_i < a_j, a_i \geq a_j, a_i > a_j$$

- Nehmen wir an, die Elemente sind distinkt, dann reichen Vergleiche der Art $a_i \leq a_j$

- 1 solcher Vergleich ist eine Grundoperation

Vergleichsbasierte Sortieralgorithmen

UNI FREIBURG

Alternative Sichtweise

- Jedes Programm (für einen deterministischen, vgl.-basierten Sortieralg.) kann in eine Form gebracht werden, in welcher jede if/while/…-Bedingung von folgender Form ist:

$$\textbf{if } (a_i \leq a_j) \textbf{ then } \ldots$$

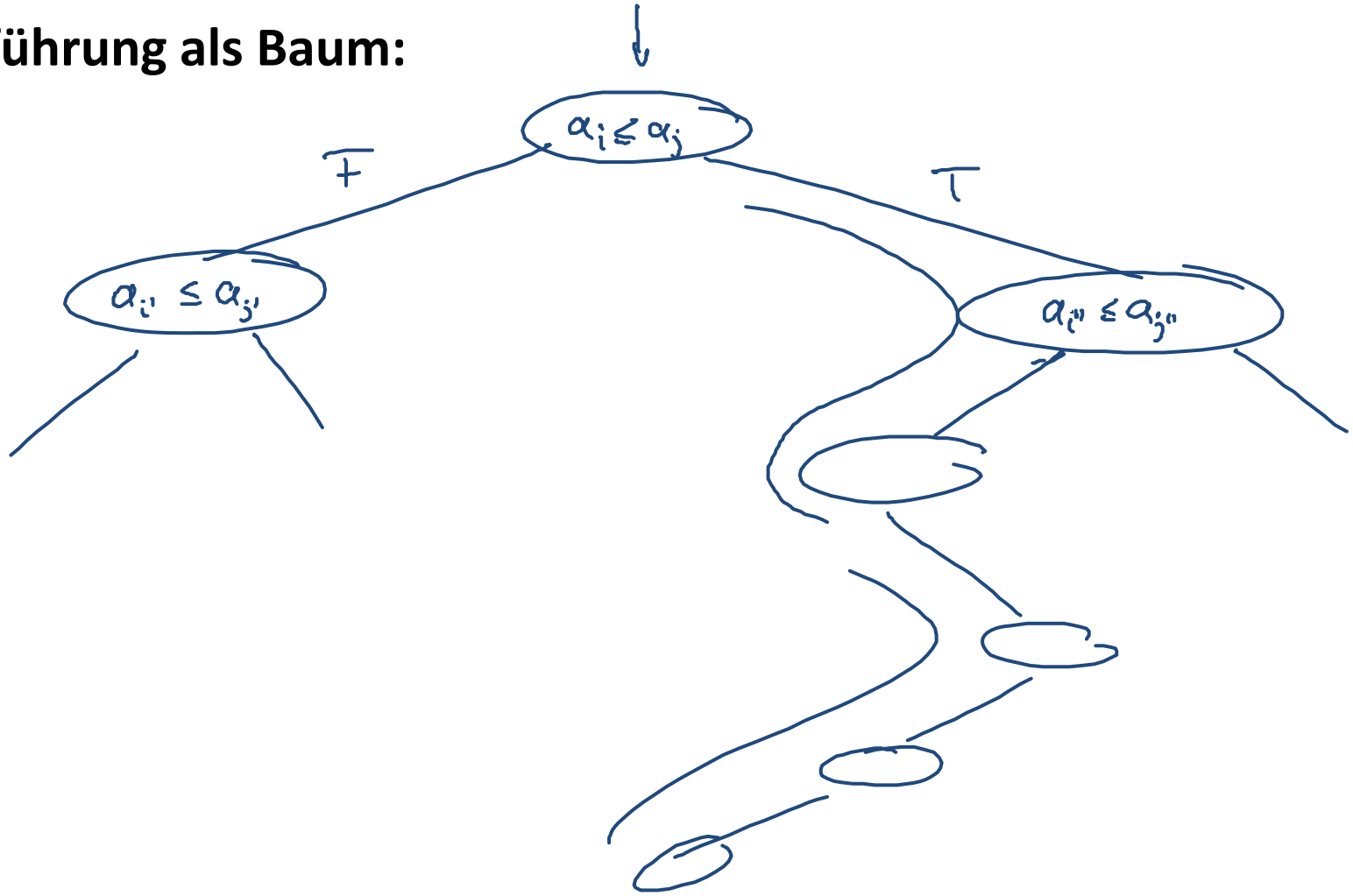
- In jeder Ausführung eines Algorithmus, induzieren die Resultate dieser Vergleiche eine Abfolge von T/F (true/false) Werten:

$$\mathbf{TFFTTTFTFFTTFFFFFTFTTT} \ldots$$

- Diese Abfolge bestimmt in eindeutiger Weise, wie die Elemente umgeordnet werden.
- Unterschiedliche Eingaben der gleichen Werte, müssen daher zu unterschiedlichen T/F-Sequenzen führen!

Vergleichsbasierte Sortieralgorithmen

Ausführung als Baum:



Vgl.-Basiertes Sortieren : Untere Schranke I

UNI FREIBURG

- Bei vergleichsbasierten Sortieralgorithmen hängt die Ausführung nur von der Ordnung der Werte in der Eingabe, nicht aber von den eigentlichen Werten ab
  - Wir beschränken und auf Eingaben, bei denen die Werte unterschiedlich sind.
- O.b.d.A. können wir deshalb annehmen, dass wir die Zahlen $1, \dots, n$ sortieren müssen.
- Verschiedene Eingaben müssen verschieden bearbeitet werden.
- Verschiedene Eingaben erzeugen verschiedene T/F-Folgen
- Laufzeit einer Ausführung $\geq$ Länge der erzeugten T/F-Folge
- Worst-Case Laufzeit $\geq$ Länge der längsten T/F-Folge:
  - Wir wollen eine untere Schranke
  - Zählen der Anz. mögl. Eingaben → wir benötigen so viele T/F-Folgen...

Vgl.-Basiertes Sortieren : Untere Schranke I

Anzahl Mögliche Eingaben (Anfangsreihenfolgen):

#Permutationen von $\{1,\dots,n\}$: $n! = \underbrace{n(n-1)(n-2)\cdot\ldots\cdot 1}_{n \;\dots\; n/2}$

Anzahl T/F-Folgen der Länge k:

Länge $= k$: 2^k Länge $\le k$: $\sum_{i=1}^{k} 2^i = 2^{k+1} - 1 \le 2^{k+1}$

Theorem: Jeder det. Vergleichs-basierte Sortieralgorithmus benötigt im Worst Case mindestens $\Omega(n \cdot \log n)$ Vergleiche.

Laufzeit $\le T$:

$2^{T+1} \ge n!$

$T+1 \ge \log(n!)$

$\Rightarrow$ worst-case Laufzeit $T(n) \in \Omega(\log(n!))$

$\Downarrow$

$T(n) \in \Omega(n \log n)$

$\left(\frac{n}{2}\right)^{n/2} \le n! \le n^n$

$\frac{n}{2}\log\left(\frac{n}{2}\right) \le \log(n!) \le n \cdot \log n$

$\log(a^b) = b \cdot \log a$

Sortieren in Linearzeit

UNI FREIBURG

- Mit Vergleichs-basierten Algorithmen nicht möglich
  - Untere Schranke gilt auch mit Randomisierung...

- Manchmal geht's schneller
  - wenn wir etwas über die Art der Eingabe wissen und ausnützen können

- Beispiel: Sortiere n Zahlen $a_i \in \{0,1\}$:
  1. Zähle Anzahl Nullen und Einsen in $O(n)$ Zeit
  2. Schreibe Lösung in Array in $O(n)$ Zeit

Counting Sort

Aufgabe:

- Sortiere Integer-Array A der Länge n
- Wir wissen, dass für alle $i \in \{0, \dots, n-1\}$, $A[i] \in \{0, \dots, k\}$

Algorithmus:

```
1: counts = new int[k+1]          // new int array of length k
2: for i = 0 to k do counts[i] = 0
3: for i = 0 to n-1 do counts[A[i]]++
4: i = 0;
5: for j = 0 to k do
6:   for l = 0 to counts[j] do
7:     A[i] = j; i++
```

Analyse Counting Sort

```
1: counts = new int[k+1]
2: for i = 0 to k do counts[i] = 0
3: for i = 0 to n-1 do counts[A[i]]++
4: i = 0;
5: for j = 0 to k do
6:   for l = 0 to counts[j] do
7:     A[i] = j; i++
```

$\leftarrow O(1)$

$\leftarrow O(k)$

$\leftarrow O(n)$

$O(k \cdot n)$ $O(k+n)$

$O(\text{counts}[j]+1)$

$O\left(\sum_{j=0}^{k} (\text{counts}[j]+1)\right)$

$k+1 + \sum \text{counts}[j]$

Laufzeit: $O(n+k)$

$\sum_{j=0}^{n} (\text{counts}[j]+1) = \sum_{j=0}^{k} \text{counts}[j] + \sum_{j=0}^{k} 1$

$k+1$

Counting Sort Implementierung

UNI FREIBURG

- Versuchen wir das doch gleich mal zu programmieren…

Um's ein bisschen interessanter zu machen:

- Anstatt einfach Integers zu sortieren, versuchen wir Elemente der folgenden Form nach Ihrem *key* zu sortieren:

```
public class DataItem {
    int key;
    Object data;
}
```

- Einfach nur zählen und dann die entsprechende Anzahl Werte einzufüllen funktioniert hier nicht…

Stabiles Sortieren

UNI FREIBURG

Definition: Ein Sortieralgorithmus ist stabil, falls Elemente mit gleichem Schlüssel in der gleichen Reihenfolge bleiben.

Beispiel:

Sortiere folgende Strings stabil nach Anfangsbuchstabe:

“tuv”, “adr”, “bbc”, “tag”, “taa”, “abc”, “sru”, “bcb”

adr, abc, bbc, bcb, sru, tuv, taa

Übung: Welche der in der Vorlesung behandelten Sortieralgorithmen sind (so, wie wir sie besprochen haben) stabil?

Bemerkungen zur Übung

UNI FREIBURG

- Ausnahmsweise eine grössere Übung
- Dafür haben Sie 2 Wochen Zeit
  - Es gibt nächste Woche keine Übung
- Übung wird ab morgen online sein

Grund:

- Übungs-/Fragestunde am Di, 20.5. 16:15-18:00
  - Eine Übung für 3 Doppelstunden
  - Fragestunde wird in der Mitte der Bearbeitungszeit sein...

Inhalt:

- Theoretische Fragen + Implementationsaufgaben

Datenstrukturen

UNI FREIBURG

Algorithmen

- Wie löst man ein gegebenes Problem effizient
- Ziel: möglichst geringe Komplexität
  - kurze Laufzeit / kleiner Speicherverbrauch
  - asymptotisch, abhängig von der Problemgrösse

Datenstrukturen

- Wie können Daten so abgespeichert werden, dass der Zugriff möglichst effizient ist
- Hängt von den Operationen ab, welche unterstützt werden sollen!
- Ermöglicht schnelle Algorithmen
- Benötigt schnelle Algorithmen, um die Operationen optimal auszuführen

Abstrakte Datentypen

UNI FREIBURG

Abstrakter Datentyp:

- Spezifikation, welche Art von Daten verwaltet werden können
- Spezifikation der Operationen, um auf die Daten zuzugreifen
  - inkl. der Semantik der Operation

Datenstruktur:

- Bestimmte Art, einen abstrakten Datentypen zu implementieren
- Je nach Implementierung können die gleichen Operationen verschiedene Laufzeiten (Komplexität) haben.

Abstrakte Datentypen : Beispiele

UNI FREIBURG

Array:

feste Grösse

- Verwaltet eine Kollektion von Elementen (des gleichen Typs)

Operationen:

- *create(n)* : erzeugt ein Array der Länge n
- *A.get(i)* : gibt das Element an Position i zurück A[i]
- *A.set(x, i)* : schreibt Element x an Position i
- *A.size()* : gibt die Länge des Arrays zurück (nicht immer dabei)

Bei dynamischen Arrays (können Grösse verändern):

- *A.append(x)* : hängt Element x hinten an
- *A.deleteLast()* : löscht letztes Element

Abstrakte Datentypen : Beispiele

UNI FREIBURG

Dictionary: (auch: Maps, assoziative Arrays)

- Verwaltet eine Kollektion von Elementen, wo bei jedes Element durch einen eindeutigen Schlüssel (key) repräsentiert wird

Operationen:

- *create* : erzeugt einen leeren Dictionary
- *D.insert(key, value)* : fügt neues *(key,value)*-Paar hinzu
  - falls schon ein Eintrag für *key* besteht, wird er ersetzt
- *D.find(key)* : gibt Eintrag zu Schlüssel *key* zurück
  - falls ein Eintrag vorhanden (gibt sonst einen Default-Wert zurück)
- *D.delete(key)* : löscht Eintrag zu Schlüssel *key*

Abstrakte Datentypen : Beispiele

UNI FREIBURG

Dictionary:

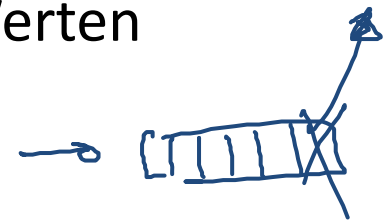
Weitere mögliche Operationen:

- *D.minimum()* : gibt kleinsten *key* in der Datenstruktur zurück
- *D.maximum()* : gibt grössten *key* in der Datenstruktur zurück
- *D.successor(key)* : gibt nächstgrösseren *key* zurück
- *D.predecessor(key)* : gibt nächstkleineren *key* zurück
- *D.getRange(k1, k2)* : gibt alle Einträge mit Schlüsseln im Intervall *[k1,k2]* zurück

Abstrakte Datentypen : Beispiele

Queue (Warteschlange):

- Verwaltet eine Menge (“Sequenz”) von Werten



Operationen:

- *create* : erzeugt eine leere Queue
- *Q.enqueue(x)* : hängt Element *x* hinten an
- *Q.dequeue()* : gibt vorderstes Element zurück und löscht es
- *Q.isEmpty()* : Ist die Queue leer?

Heisst auch FIFO Queue (FIFO = first in first out)

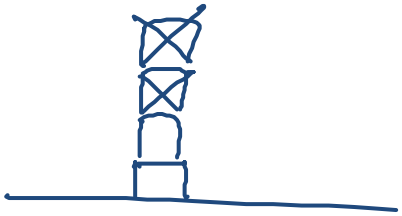
Abstrakte Datentypen : Beispiele

UNI FREIBURG

Stack (Stapel):

- Verwaltet eine Menge (“Sequenz”) von Werten

Operationen:



- *create* : erzeugt einen leeren Stack
- *S.push(x)* : legt Element *x* auf den Stack
- *S.pop()* : gibt oberstes Element zurück und löscht es
- *S.isEmpty()* : Ist der Stack leer?

Heisst auch LIFO Queue (LIFO = last in first out)