Informatik II - SS 2016

(Algorithmen & Datenstrukturen)

Vorlesung 5 (4.5.2016)

Sortieren V, Abstrakte Datentypen

Fabian Kuhn
Algorithmen und Komplexität

UNI FREIBURG

Zusammenfassung 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
- **Lower Bound**
  "comparison-based" Algorithmen: $\Omega(n \log n)$

Sortieren in Linearzeit

- 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 $i \in \{0, \ldots, n\}$, $A[i] \in \{0, \ldots, k\}$

Algorithmus:

```
counts = new int[k+1]        // new int array of length k
for i = 0 to k do counts[i] = 0
for i = 0 to n-1 do counts[A[i]]++
i = 0;
for j = 0 to k do
   for l = 0 to counts[j] do
      A[i] = j; i++
```

Analyse Counting Sort

```
counts = new int[k+1]
for i = 0 to k do counts[i] = 0
for i = 0 to n-1 do counts[A[i]]++
i = 0;
for j = 0 to k do
  for l = 0 to counts[j] do
    A[i] = j; i++
```

Counting Sort Implementierung

- Schauen wir das doch gleich mal in Python an…

Um's ein bisschen interessanter zu machen:

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

```
class DataItem:
    def __init__(self, key = 0, data = None):
        self.key = key
        self.data = data
```

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

Stabiles Sortieren

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”

Stabiles Sortieren

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

Sortieren in Linearzeit II

Counting Sort:

- sortiert ganze Zahlen zwischen 0 und $O(n)$ in Linearzeit

Was, wenn die Zahlen aus einem grösseren Bereich sind?

- z.B.: ganze Zahlen zwischen 0 und $n^2 - 1$?

Sortieren in Linearzeit II

Sortiere Werte zwischen 0 und $\boldsymbol{n^2 - 1}$:

Ziel: Sortiere $x_0, x_1, x_2, \dots, x_{n-1}$, wobei $x_i \in \{0, \dots, n^2 - 1\}$

Algorithmus:

1. Schreibe alle Zahlen als $x_i = a_i \cdot n + b\_i$ für $a_i, b_i \in \{0, \dots, n - 1\}$
2. Sortiere Zahlen mit Counting Sort nach b_i
3. Sortiere Zahlen **stabil** mit Counting Sort nach a_i

Radixsort

Verallgemeinerung der Idee

- **Ziel:** sortiere (nichtnegative) ganze Zahlen $x_0, \dots, x_{n-1}$
- Schreibe Zahlen als $x_i = a_{i,0} + a_{i,1} \cdot n + a_{i,2} \cdot n^2 + \cdots + a_{i,k} \cdot n^k$

Algorithmus:

1. Sortiere mit Counting Sort nach $a_{i,0}$
2. Sortiere mit Counting Sort stabil nach $a_{i,1}$
3. Sortiere mit Counting Sort stabil nach $a_{i,2}$
4. …
5. Sortiere mit Counting Sort stabil nach $a_{i,k}$

Radixsort

Sortiere $x_0, \dots, x_{n-1}$, mit $x_i = a_{i,0} + a_{i,1}n + a_{i,2}n^2 + \cdots + a_{i,k}n^k$

Algorithmus:

1. Sortiere mit Counting Sort nach $a_{i,0}$
2. Sortiere mit Counting Sort stabil nach $a_{i,1}$
3. …
4. Sortiere mit Counting Sort stabil nach $a_{i,k}$

Zusammenfassung Sortieralgorithmen

- **Selection Sort, Insertion Sort, Bubble Sort**
  worst case: $\Theta(n^2)$
- **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: $O(n \log n)$
- **Radix Sort** (for sorting positive integers)
  worst case: $O(n \log_n M)$, where M is the largest value
- **Lower Bound**
  "comparison-based" Algorithmen: $\Omega(n \log n)$

Datenstrukturen

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

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

Array:

- Verwaltet eine Menge 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.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 Menge 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

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

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)

Abstrakte Datentypen : Beispiele

Heap / Priority Queue (Prioritätswarteschlange):

- Verwaltet eine Menge von *(key,value)*-Paaren

Operationen:

- *create* : erzeugt einen leeren Heap
- *H.insert(x, key)* : fügt Element *x* mit Schlüssel *key* ein
- *H.getMin()* : gibt Element mit kleinstem Schlüssel zurück
- *H.deleteMin()* : löscht Element mit kleinstem Schlüssel
- *H.decreaseKey(x, newkey)* : Falls *newkey* kleiner als der aktuelle Schlüssel von *x* ist, wird der Schlüssel von *x* auf *newkey* gesetzt

Abstrakte Datentypen : Beispiele

Union-Find / Disjoint Sets:

- Verwaltet eine Partition von Elementen

Operationen:

- *create* : erzeugt eine leere Union-Find-DS
- *U.makeSet(x)* : fügt Menge $\{x\}$ zur Partition hinzu
- *U.find(x)* : gibt Menge mit Element x zurück
- *U.union(S1, S2)* : vereinigt die Mengen $S1$ und $S2$

Array-Implementierung Stack

Versuchen wir den Stack-Datentyp zu implementieren

- **Operationen:** *create, push, pop, isEmpty*
- **Annahme:** Stack muss nur für *NMAX* Elemente Platz bieten

Variablen, um den Zustand des Stack zu speichern:

- $stack$: Array der Länge *NMAX*
- $size$: Aktuelle Anzahl Elemente im Stack

```
create:
  stack = new array of length NMAX
  size = 0
```

Array-Implementierung Stack

```
isEmpty:
  return (size == 0)

S.push(x):
  if (size < NMAX)
    stack[size] = x
    size += 1

S.pop():
  if (size == 0)
    report error (or return default value)
  else
    size -= 1
    return stack[size]
```

Analyse Stack Array Implementierung

Laufzeit (Zeitkomplexität) der Operationen:

- create: $O(1)$
  - falls man davon ausgeht, dass Speicher in $O(1)$ Zeit alloziert werden kann
- push: $O(1)$
- pop: $O(1)$
- isEmpty: $O(1)$

Nachteile der Implementierung:

- Speicherverbrauch (space complexity) : $O(NMAX)$
  - man braucht immer gleich viel Speicher, egal wie viele Elemente im Stack gespeichert sind!
- Der Stack kann nur *NMAX* Elemente aufnehmen…
- Wir werden sehen, wie man beides beheben kann…

Stack : Anwendungen

- Umdrehen einer Sequenz:

- Undo-Funktion bei Editoren
  - lege Beschreibung von (umkehrbaren) Operationen auf Stack ab

- Programmstack für Funktionen/Methoden-Aufrufe
  - Bemerkung: Mit einem Stack kann man Rekursion explizit aufschreiben