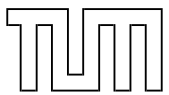
Grundlagen: Algorithmen und Datenstrukturen

Prof. Dr. Hanjo Täubig

Lehrstuhl für Effiziente Algorithmen
(Prof. Dr. Ernst W. Mayr)
Institut für Informatik
Technische Universität München

Sommersemester 2010



Übersicht

1. Hashing
   - Perfektes Hashing
2. Sortieren

Perfektes dynamisches Hashing

Kann man perfekte Hashfunktionen auch dynamisch konstruieren?

ja, z.B. mit Cuckoo Hashing

- 2 Hashfunktionen h_1 und h_2
- 2 Hashtabellen T_1 und T_2
- bei find und remove jeweils in beiden Tabellen nachschauen
- bei insert abwechselnd beide Tabellen betrachten, das zu speichernde Element an die Zielposition der aktuellen Tabelle schreiben und wenn dort schon ein anderes Element stand, dieses genauso in die andere Tabelle verschieben usw.
- evt. Anzahl Verschiebungen durch $2 \log n$ beschränken, um Endlosschleife zu verhindern
  (ggf. kompletter Rehash mit neuen Funktionen h_1, h_2)

Verteiltes Wörterbuch / konsistentes Hashing

- Hashing kann für verteiltes Speichern von Daten benutzt werden (z.B. auf mehreren Festplatten oder Knoten in einem Netzwerk)
- Problem: Änderung bei Speichermedien (Erweiterungen, Ausfälle)

$\Rightarrow$ Konsistentes Hashing

- benutze eine zufällige Hashfunktion, um die Schlüssel auf eine Zahl im Intervall $[0, 1)$ abzubilden
- benutze eine zweite zufällige Hashfunktion, um jedem Speichermedium ein Intervall in $[0, 1)$ zuzuordnen, für das dieser Speicher dann zuständig ist

Übersicht

1 Hashing

2 Sortieren
- Einfache Verfahren

Wörterbuch-Datenstruktur

- S: Menge von Elementen
- Element e wird identifiziert über eindeutigen Schlüssel key(e)

Operationen:

- S.insert(Elem e): $S = S \cup \{e\}$
- S.remove(Key k): $S = S \setminus \{e\}$,
  wobei e das Element mit key(e) $==$ k ist
- S.find(Key k):
  gibt das Element $e \in S$ mit key(e) $==$ k zurück, falls es existiert,
  sonst null

Statisches Wörterbuch

Lösungsmöglichkeiten

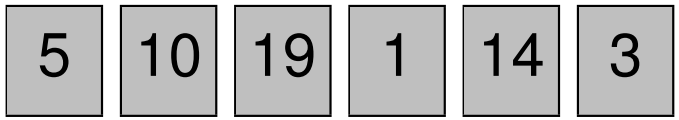
- Perfektes Hashing
  - Vorteil: Suche in konstanter Zeit
  - Nachteil: keine Ordnung auf Elementen, d.h. Bereichsanfragen (z.B. alle Namen, die mit 'A' anfangen) teuer

- Speicherung der Daten in sortiertem Feld
  - Vorteil: Bereichsanfragen möglich
  - Nachteil: Suche teurer (logarithmische Zeit)

Sortierproblem

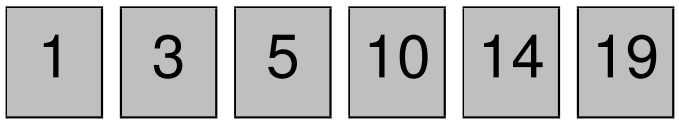
- Eingabe:
  Sequenz $s = \langle e_1, \ldots, e_n \rangle$ mit Ordnung $\leq$ auf den Schlüsseln key(e_i)

  Beispiel:



- Ausgabe:
  Permutation $s' = \langle e'_1, \ldots, e'_n \rangle$ von s, so dass key(e'_i) $\leq$ key(e'_{i+1}) für alle $i \in \{1, \ldots, n\}$

  Beispiel:



SelectionSort

Sortieren durch Auswählen

Wähle das kleinste Element aus der (verbleibenden) Eingabesequenz und verschiebe es an das Ende der Ausgabesequenz

Beispiel

5	10	19	1	14	3
1	10	19	5	14	3
1	5	19	10	14	3
1	**3**	19	10	14	5
1	3	10	19	14	5

1	3	**5**	19	14	10
1	3	5	14	19	10
1	3	5	**10**	19	14
1	3	5	10	**14**	19
1	3	5	10	14	**19**

SelectionSort

Sortieren durch Auswählen

```
void selectionSort(Element[ ] a, int n) {
   for (int i = 0; i < n; i++)
      // verschiebe min{a[i],...,a[n – 1]} nach a[i]
      for (int j = i + 1; j < n; j++)
         if (a[i] > a[j])
            swap(a[i], a[j]);
}
```

Zeitaufwand:

- Minimumsuche in Feld der Größe i: $\Theta(i)$
- Gesamtzeit: $\sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$
- Vorteil: einfach zu implementieren
- Nachteil: quadratische Laufzeit

InsertionSort

Sortieren durch Einfügen

Nimm ein Element aus der Eingabesequenz und füge es an der richtigen Stelle in die Ausgabesequenz ein

Beispiel

5	10	19	1	14	3
5	10	19	1	14	3
5	**10**	19	1	14	3
5	10	**19**	1	14	3
5	10	1	19	14	3

5	1	10	19	14	3
1	5	10	19	14	3
1	5	10	**14**	19	3
1	...	←	...	3	19
1	**3**	5	10	14	19

InsertionSort

Sortieren durch Einfügen

```
void insertionSort(Element[ ] a, int n) {
   for (int i = 1; i < n; i++)
      // verschiebe a_i an die richtige Stelle
      for (int j = i – 1; j ≥ 0; j – –)
         if (a[j] > a[j + 1])
            swap(a[j], a[j + 1]);
}
```

Zeitaufwand:

- Einfügung des i-ten Elements an richtiger Stelle: $\mathcal{O}(i)$
- Gesamtzeit: $\sum_{i=1}^{n} \mathcal{O}(i) = \mathcal{O}(n^2)$
- Vorteil: einfach zu implementieren
- Nachteil: quadratische Laufzeit

Einfache Verfahren

SelectionSort

- mit besserer Minimumstrategie worst case Laufzeit $\mathcal{O}(n \log n)$ erreichbar
  (mehr dazu in einer späteren Vorlesung)

InsertionSort

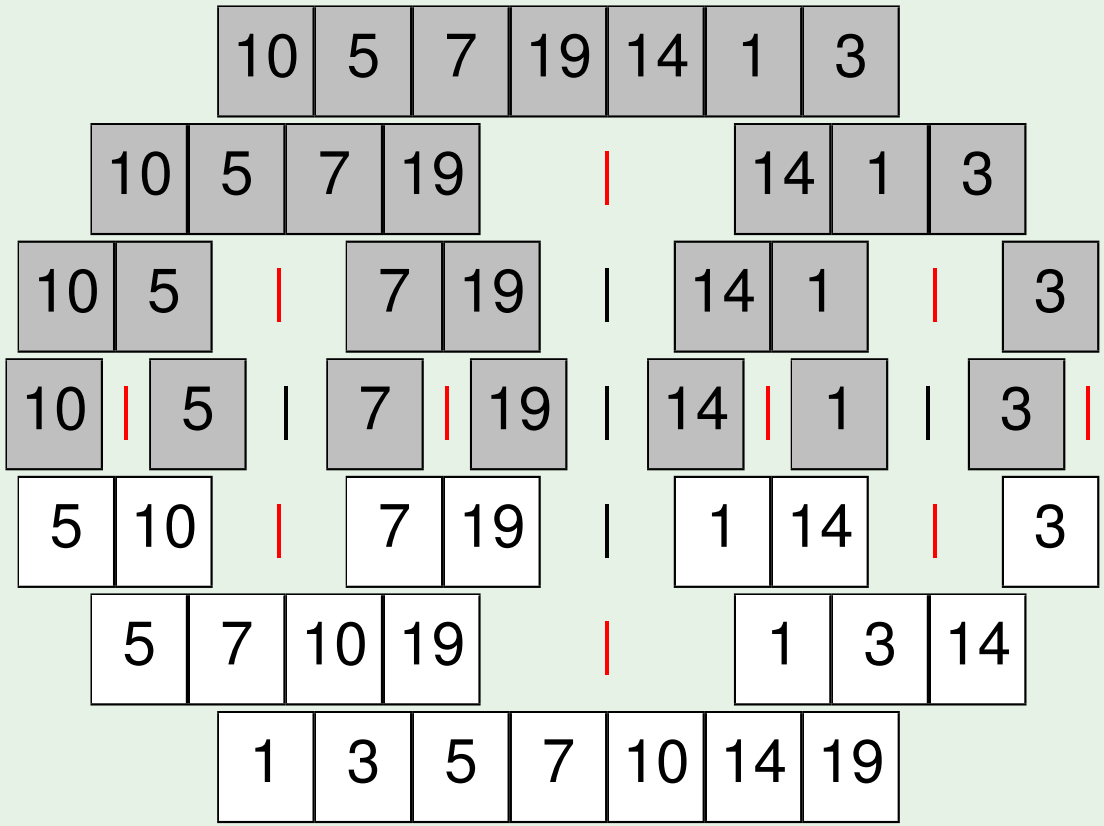
- mit besserer Einfügestrategie worst case Laufzeit $\mathcal{O}(n \log^2 n)$ erreichbar
  ($\rightarrow$ ShellSort)

MergeSort

Sortieren durch Verschmelzen

Zerlege die Eingabesequenz rekursiv in Teile, die separat sortiert und dann zur Ausgabesequenz verschmolzen werden

Beispiel



MergeSort

Sortieren durch Verschmelzen

```
void mergeSort(Element[ ] a, int l, int r) {
   if (l == r) return;
   m = ⌈(r + l)/2⌉;        // Mitte
   mergeSort(a, l, m);
   mergeSort(a, m + 1, r);
   j = l;   k = m + 1;
   for i = 1 to r – l + 1 do
      if (j > m) { b[i] = a[k];   k++; }
      else
         if (k > r) { b[i] = a[j];   j++; }
         else
            if (a[j] < a[k]) { b[i] = a[j];   j++; }
            else { b[i] = a[k];   k++; }
   for i = 1 to r – l + 1 do   a[l – 1 + i] = b[i];
}
```