ADS: Algorithmen und Datenstrukturen

Zweiter Akt

Prof. Peter F. Stadler & Dr. Christian Höner zu Siederdissen

Bioinformatik/IZBI
Institut für Informatik
& Interdisziplinäres Zentrum für Bioinformatik
Universität Leipzig

19. Oktober 2015

[Letzte Aktualisierung: 20/10/2015, 00:11]

Übungen

- Technisches Problem: Gruppe wurde nicht übernommen
- Neuanmeldung zu Übungen war/ist erforderlich
- Anmeldung noch bis 21.10.2015 möglich
- Wechsel in eine bereits volle Gruppe ist nur möglich wenn Sie einen Tauschpartner finden: informieren sie dann beide Übungsleiter
- bei erfolgreichem Wechsel: geben Sie ihre Übungsblätter in der neuen Gruppe ab

wer noch keinen Übungsplatz hat:

- wir öffnen die Übungsanmeldung nach 17:00 Uhr
- die gewählte Übung wird aber nicht garantiert
- falls sich andere Übungen leeren, werden wir Nachzügler umverteilen müssen

Einfache Suchverfahren

- Lineare Listen
- Sequentielle Suche
- Binäre Suche
- Weitere Suchverfahren auf sortierten Feldern
  - Sprungsuche
  - Exponentielle Suche
  - Interpolationssuche
- Auswahlproblem

Lineare Listen

- Endliche Folge von Elementen des selben Typs
  $L = \langle a_0, a_1, \ldots a_{n-1} \rangle$ falls $n (= L.\texttt{length}) > 0$
  $L = \langle\,\rangle$ leere Liste, $n = 0$.
- Reihenfolge der Elemente ist wesentlich ($\leftrightarrow$ Mengen)

Zwei grundsätzlich unterschiedliche Implementierungen

- Sequenzielle Speicherung (Reihung, Array)
  - statische Datenstruktur (Grössenänderung teuer)
  - wahlfreier Zugriff über (berechenbaren) Index
  - in Vorlesung/Übung: Index 0-basiert, d.h. Elemente `L[0]`,`L[1]`,...,`L[n-1]`
  - Varianten: 1-dimensionale vs. mehrdimensionale Arrays
- Verkettete Speicherung
  - dynamische Datenstruktur (Einfügen und Löschen billig)
  - kein wahlfreier Zugriff über Index
  - Verkettung (jeweils zum Nachfolger) erlaubt sequenzielles Durchlaufen
  - Varianten: einfache vs. doppelte Verkettung etc.

Sequenzielle Suche

Sequenzielle Suche nach Element mit Schlüsselwert x:

```
Durchlaufe Liste der Reihe nach und vergleiche jeden
Schlüssel mit x
```

Kosten:

- erfolglose Suche erfordert n Schlüsselvergleiche
- erfolgreiche Suche verlangt im ungünstigsten Fall n Schlüsselvergleiche
- mittlere Anzahl von Schlüsselvergleichen bei erfolgreicher Suche

$$C_{avg}(n) = \frac{1}{n} \cdot \sum_{i=1}^{n} i = \frac{n+1}{2}$$

Sequenzielle Suche funktioniert *"immer"*.

Lässt sich schneller sequenziell suchen, wenn die Liste sortiert ist?

Binäre Suche

- In sortierten Listen geht Suche deutlich schneller als sequenziell
- **Binäre Suche** (*Suche durch rekursive binäre Zerlegung*)

```
Suche x in Liste L:
  if (L empty)
    report "nicht gefunden";
  else
    mid = floor(L.length/2);
    if (x==L[mid]) report "gefunden";
    if (x<L[mid]) Suche x in (Sub-)Liste L[0..mid-1];
    if (x>L[mid]) Suche x in (Sub-)Liste L[mid+1..n-1];
```

- Kosten für erfolgreiche Binärsuche

$$C_{\min}(n) = 1, \quad C_{\max}(n) = \lfloor \log_2(n) \rfloor + 1, \quad C_{avg}(n) \approx \log_2(n) \quad n \to \infty$$

- **Verbesserung von $O(n)$ auf $O(\log n)$**

Funktionen `floor(.)`, $\lfloor . \rfloor \equiv$ abrunden, `ceiling(.)`, $\lceil . \rceil \equiv$ aufrunden

Die Funktionen "floor" und "ceiling"

$\lfloor x \rfloor$, auch `floor`(x) genannt, beschreibt die größte ganze Zahl nicht größer als x, also den "ganzzahligen Anteil" von x.

Beispiele: $\lfloor \pi \rfloor = 3$. $\lfloor -\pi \rfloor = -4$.

Eigenschaften:

1. $\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$
2. $k \in \mathbb{N}$ impliziert $\lfloor k + x \rfloor = k + \lfloor x \rfloor$

$\lceil x \rceil$, `ceiling`(x), ist die kleinste ganze Zahl nicht kleiner als x.

Beispiele: $\lceil \pi \rceil = 4$. $\lceil -\pi \rceil = -3$

Eigenschaften:

1. $x \leq \lceil x \rceil < x + 1$
2. $k \in \mathbb{N}$ impliziert $\lceil k \rceil = \lfloor k \rfloor = k$
3. $k \in \mathbb{N}$ impliziert $\lceil k/2 \rceil + \lfloor k/2 \rfloor = k$

Sprungsuche

Vorbedingung (immer noch): Liste L ist (aufsteigend) sortiert

Prinzip: L wird in Abschnitte fester Länge m unterteilt. Springe über die Abschnitte, um den Abschnitt des Schlüssels zu bestimmen.

Abschnitte: $0 \ldots (m-1)$, $m \ldots (2m-1)$, $2m \ldots (3m-1)$, usw.

Einfache Sprungsuche:

- Springe zu Positionen $i \cdot m$ (für $i = 1, 2, \ldots$; der Reihe nach)
- Sobald $x < L[im]$, kann x nur in i-tem Abschnitt $(i-1)m \ldots im-1$ liegen; dieser wird sequenziell nach x durchsucht

Mittlere Suchkosten: ein Sprung koste a, ein Vergleich b:

$$C_{avg}(n) = \frac{1}{2}\frac{n}{m}a + \frac{1}{2}mb$$

Minimiere $C_{avg}(n)$ über m (>>Ableitung nach m<< $\stackrel{!}{=} 0$; Übung)

$\Rightarrow$ optimale Sprungweite $m = \lfloor\sqrt{(a/b)n}\rfloor$; dann ist Komplexität in $O(\sqrt{n})$

Worst case Komplexität?

Zwei-Ebenen-Sprungsuche

- Prinzip: Statt sequenzieller Suche im lokalisierten Abschnitt wird wiederum eine Quadratwurzel-Sprungsuche angewendet, bevor dann sequenziell gesucht wird
- Mittlere Suchkosten: $C_{avg}(n) \leq \frac{1}{2}a\sqrt{n} + \frac{1}{2}b\sqrt{\sqrt{n}} + \frac{1}{2}c\sqrt{\sqrt{n}}$
  a Kosten eines Sprungs auf der ersten Ebene
  b Kosten eines Sprungs auf der zweiten Ebene
  c Kosten eines sequenziellen Vergleichs
- Verbesserung durch optimale Abstimmung der Sprungweiten m_1 und m_2 der beiden Ebenen
- Mit $a = b = c$ ergeben sich als optimale Sprungweiten $m_1 = n^{2/3}$ und $m_2 = n^{1/3}$ und mittlere Suchkosten $C_{avg}(n) = \frac{3}{2}an^{1/3}$
- Verallgemeinerung zu n-Ebenen-Verfahren ergibt ähnlich günstige Kosten wie Binärsuche

Wann ist Sprungsuche vorteilhaft? Allgemein: wenn Binärsuche nicht sinnvoll ist; z.B. bei blockweisem Einlesen der sortierten Sätze vom Externspeicher oder ...

Exponentielle Suche

Was tun, falls die Länge eines sortierten Suchbereichs zunächst unbekannt ist?

- **Vorgehensweise**
  1. für Suchschlüssel x wird zunächst obere Grenze für den zu durchsuchenden Abschnitt bestimmt:

     ```
     int i = 1; while (x > L[i]) i=2*i;
     ```

  2. danach gilt $L[i/2] < x \leq L[i]$; d.h. Suchabschnitt: $i/2 + 1 \ldots i$.
  3. suche innerhalb des Abschnitts mit irgendeinem Verfahren
- **Analyse**

  Annahme: Liste enthält nur Schlüssel aus $\mathbb{N}$ ohne Duplikate

  $\rightarrow$ Schlüsselwerte wachsen mindestens so stark wie die Elementindizes i

  $\rightarrow$ höchstens $\log_2 x$ Verdopplungen, denn die Zahl der Sprünge ist $log_2 i$

  Bestimmung des gesuchten Intervalls: $\leq log_2 x$ Schlüsselvergleiche

  Suche innerhalb des Abschnitts (z.B. Binärsuche): $\leq log_2 x$ Vergleiche
- **Gesamtaufwand** $O(log_2 x)$

Interpolationssuche

Idee: Im Prinzip wie Binärsuche, aber noch schnellere Einschränkung des Suchbereichs, indem wir versuchen den Index des Suchschlüssels x aus den Schlüsselwerten zu erraten.

Vorgehensweise: Berechne nächste Suchposition aus den Grenzen u und v des aktuellen Suchbereichs $[u, v]$:

$$p = u + \frac{x - L[u]}{L[v] - L[u]}(v - u)$$

Dann zerlege den Suchbereichs bei $\lfloor p \rfloor$ wie in Binärsuche.

Analyse: Sinnvoll, falls Schlüsselwerte im betreffenden Bereich einigermaßen gleichverteilt. Dann im Mittel nur

$$C_{avg}(n) = 1 + \log_2 \log_2 n \qquad \text{Vergleiche.}$$

Im Extremfall "besonders irreguläre Schlüsselverteilung" aber $O(n)$.

Wie sieht so eine extreme "worst case" Verteilung z.B. aus?

Häufigkeitsgeordnete lineare Listen

- **Prinzip**: Ordne die Liste nach Zugriffshäufigkeiten und suche sequenziell. Sinnvoll, falls die Zugriffshäufigkeiten für die einzelnen Elemente einer Liste sehr unterschiedlich sind
- **Analyse**: (Sei p_i die Zugriffswahrscheinlichkeit für Element i)
  Mittlere Suchkosten bei (erfolgreicher) sequenzieller Suche

$$C_{avg}(n) = 1p_1 + 2p_2 + 3p_3 + \cdots + np_n = \sum_{i=1}^{n} ip_i$$

  um C_{avg} zu minimieren: organisiere Liste, so daß $p_1 \geq p_2 \geq ... \geq p_n$.
- **Beispiel** *Zugriffsverteilung nach 80-20-Regel*
  80% der Suchanfragen betreffen 20% des Datenbestandes und von diesen 80% wiederum 80% (also insgesamt 64%) der Suchanfragen richten sich an 20% von 20% (d.h. insgesamt 4%) der Daten.

  
  erwarteter Suchaufwand $C_{avg}(n) =$ **?**

- **Problem in der Praxis**: Zugriffshäufigkeiten meist unbekannt
  $\rightarrow$ selbstorganisierende (adaptive) Listen

Selbstorganisierende Listen

- **Anwendung**: Nützlich, falls häufig gleiche Objekte gesucht werden (Beispiel: Suchbegriffe bei Suchmaschinen)
- **Prinzip**:
  1. gesucht wird immer sequenziell von vorn
  2. die Reihenfolge der Objekte in der Liste passt sich dem Zugriffsmuster an (dazu gibt es verschiedene Regeln)
- **Ansatz 1**: FC-Regel (Frequency count)
  - führe einen Häufigkeitszähler pro Element ein
  - jeder Zugriff auf ein Element erhöht dessen Häufigkeitszähler
  - falls erforderlich, wird danach die Liste lokal neu geordnet, so dass die Häufigkeitszähler der Elemente eine absteigende Reihenfolge bilden
  - hoher Wartungsaufwand und zusätzlicher Speicherplatzbedarf

Selbstorganisierende Listen

- **Ansatz 2**: T-Regel (Transpose)
  - das Zielelement eines Suchvorgangs wird jeweils mit dem unmittelbar vorangehenden Element vertauscht
  - häufig referenzierte Elemente wandern (langsam) an den Listenanfang
- **Ansatz 3**: MF-Regel (Move-to-Front)
  - das Zielelement eines Suchvorgangs wird nach jedem Zugriff an die erste Position der Liste gesetzt
  - relative Reihenfolge der übrigen Elemente bleibt gleich
  - in jüngster Vergangenheit referenzierte Elemente sind am Anfang der Liste (Zeitliche Lokalität im Zugriffsmuster kann gut genutzt werden)

Welche grundsätzliche Implementierung der Liste wählen sie je nach Ansatz?

Auswahlproblem

Aufgabe: Finde das i-kleinste Element einer Liste L

— Spezialfälle: kleinstes (Minimum), zweitkleinstes, . . . Element, mittlerer Wert (Median), größtes (Maximum)

Bei sortierter Liste: trivial

Bei unsortierter Liste: Minimum/Maximum-Bestimmung hat lineare Kosten $O(n)$

Algorithmus: Einfache Lösung für i-kleinstes Element

```
Wiederhole i-1 mal:
  Bestimme kleinstes Element in L und entferne es aus L
gib Minimum der Restliste als Ergebnis zurück
```

Komplexität $O(in)$, bzw. für $i \in O(n)$ (z.B. "Median"): $O(n^2)$

JEDOCH: es geht besser

Teile und Herrsche — Divide-and-Conquer

Allgemeines Verfahren, um große Probleme schneller zu lösen:

1. Ist das Problem zu gross, um sich "trivial" effizient lösen zu lassen, zerlege das Problem in zwei oder mehrere Teilprobleme. (Divide)
2. Löse die Teilprobleme einzeln nach der gleichen Methode (d.h. rekursiv). (Conquer)
3. Konstruiere die Gesamtlösung aus den Teillösungen. (Merge)

Das Verfahren wird in Schritt 2 rekursiv angewendet, bis hinreichend triviale (kleine) Probleme vorliegen, die sich mit anderen Mitteln lösen lassen.

Frage: Wieso lässt sich damit die Komplexität verringern?
Sehen wir uns zunächst Beispiele an

Suche in sortierter Liste als Beispiel für D&C

Sequenzielle Suche $\rightarrow$ Binärsuche

Teilungsschritt: Die Liste wird in der Mitte geteilt
Triviale Liste: Liste der Länge 1; einfacher Vergleich
Gesamtlösung: Erfolgreich, falls in einem Teil gefunden

Unser Algorithmus zur Binärsuche lässt sich umformen zu:

```
Falls die Liste die Länge 1 hat: Prüfe, ob es sich um das gesuchte
    Element handelt und gib JA oder NEIN zurück.
Sonst zerlege die Liste in der Mitte und prüfe durch Vergleich des
    gesuchten Elements mit der Trennstelle, in welcher Hälfte das
    gesuchte Element liegen muss.
Wende diesen Algorithmus rekursiv auf diese Teilliste an.
```

Zur Erinnerung:
Verbesserung der Zeitkomplexität von $O(n)$ zu $O(logn)$

Wie folgt diese Komplexität aus dem Mastertheorem?

2. D&C-Beispiel: Maximale Teilsumme

Idee: Wie zuvor zerlege die gegebene Folge in zwei Teile; bestimme die maximalen Teilsummen für beide Teile.

i	0	1	2	3	4	5	6	7	8	9	10
$F[i]$	+2	+5	-8	+4	+2	+5	+7	-2	-7	+3	+5

ABER: Die maximale Teilfolge könnte gerade die Schnittstelle einschließen
DESHALB: linkes und rechtes Randmaximum einer Folge F

- linkes Randmaximum von F: maximale Summe aller Teilfolgen, die bis zum linken Rand (Anfang) von F reichen
- analog: rechtes Randmaximum

Randmaxima lassen sich in O(n) bestimmen!

Maximale Teilsumme mit D&C

Rekursiver (D&C-)Algorithmus für maximale Teilsumme

- falls Eingabefolge F nur aus einer Zahl z besteht, nimm $\max(z, 0)$
- falls F wenigstens 2 Elemente umfasst
  - Zerlege F in etwa gleich große Hälften
  - bestimme *rekursiv* die maximalen Teilsummen m_l und m_r der linken und der rechten Hälfte
  - bestimme (in linearer Zeit)
    - das rechte Randmaximum s_l der linken Hälfte und
    - das linke Randmaximum s_r der rechten Hälfte
  - Maximale Teilsumme von F: $\max(m_l, s_l + s_r, m_r)$

Komplexitätsverbesserung?

Zurück zum Auswahlproblem

Suche i-kleinstes Element von n unterschiedlichen Elementen

1. bestimme Pivot-Element p (z.B. letztes Element)

   ```
   L =  [-------------------------]p
   ```

2. Teile die n Elemente bezüglich p in 2 Teillisten L_1 und L_2. L_1 enthält die i Elemente $< p$; L_2 die restlichen Elemente $> p$.

   ```
   L' = [.....L1.....]p[.....L2.....]
   ```

3. Falls $L_1.length = i - 1$, dann ist p das i-kleinste Element
   Falls $L_1.length > i - 1$, wende das Verfahren rekursiv auf L_1 an
   Falls $L_1.length < i - 1$, wende Verfahren rekursiv auf L_2 an; aber bestimme das $(i - 1 - L_1.length)$-kleinste Element.
4. **Komplexität:** Nehmen wir an, dass die Liste jeweils halbiert werden kann, dann ergibt sich aus $T(n) = T(n/2) + \Theta(n)$ nach Mastertheorem: $O(n)$

Ist die Annahme für Komplexitätsabschätzung realistisch?

Divide-and-Conquer

Weitere Beispiele folgen beim Thema Sortieren, z.B. Quicksort.

In welchem Sinne war unsere schnelle Multiplikation eine Anwendung von Teile-und-Herrsche?

Komplexitäts-Reduktion in D&C

Annahme: wir wollen ein Problem der Größe $n = 2^r$ lösen und zerlegen jeweils in zwei gleich grosse Teile.

Beobachtungen:

1. die maximale Rekursionstiefe ist $r = \log_2 n$
2. die Anzahl der Teilprobleme in Rekursionstiefe k ist 2^k
3. muss in jeder Tiefe nur ein Teilproblem gelöst werden (wie beim Suchen), so sind es insgesamt $r = \log_2 n$ Teilprobleme
4. so entsteht der Faktor $\log n$ bei der Komplexität
5. muss jedes Teilproblem gelöst werden, so sind n elementare Probleme zu lösen und diese Lösungen zu verknüpfen. Das macht speziell Sinn für Probleme der Komplexität $O(n^2)$ oder schlechter (z.B. maximale Teilsumme)

Erklären Sie damit, dass Teile-und-Herrsche keine Verbesserung bei sequenzieller Suche in einer ungeordneten Liste bringt.