ADS: Algorithmen und Datenstrukturen

Zweiter Akt

Peter F. Stadler & Konstantin Klemm

Bioinformatics Group, Dept. of Computer Science & Interdisciplinary Center for Bioinformatics, **University of Leipzig**

27. Oktober 2009

Einfache Suchverfahren

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

Lineare Listen

- Endliche Folge von Elementen eines Grundtyps (Elementtyps)
  $\langle a_1, a_2, \ldots a_n \rangle$ falls $n > 0$
  $\langle\,\rangle$ leere Liste, $n = 0$.
- Position von Elementen in der Liste ist wesentlich

Wesentliche Implementierungsalternativen

- Sequentielle Speicherung (Reihung, Array). Eigenschaften:
  1. statische Datenstruktur (hoher Speicheraufwand)
  2. wahlfreie Zugriffsmöglichkeit über (berechenbaren) Index
  3. 1-dimensionale vs. mehrdimensionale Arrays
- Verkettete Speicherung. Eigenschaften:
  1. dynamische Datenstruktur (Einfügen und Löschen einfach)
  2. nur sequentielle Navigation
  3. Varianten: einfache vs. doppelte Verkettung etc.

Sequenzielle Suche

Suche nach Element mit Schlüsselwert x:
falls unbekannt, ob die Elemente der Liste nach ihren Schlüsselwerten sortiert sind, ist die Liste sequentiell zu durchlaufen und elementweise zu überprüfen **(sequenzielle Suche)**
Kosten:

- erfolglose Suche erfordert n Schlüsselvergleiche (und n Schleifendurchläufe)
- erfolgreiche Suche verlangt im ungünstigsten Fall n Schlüsselvergleiche (und $n-1$ Schleifendurchläufe)
- mittlere Anzahl von Schlüsselvergleichen bei erfolgreicher Suche

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

Binäre Suche

Auf **sortierten** Listen: effizientere Suche

1. **Naiver Ansatz** Sequentielle Suche auf sortierten Listen
   $\longrightarrow$ nur geringe Verbesserung für erfolglose Suche
2. **Binärsuche** Suche nach Schlüssel x in einer aufsteigend sortierten Liste;

```
If Liste == <> then exit('erfolglos')
n = length(Liste); m = floor(n/2)
If(x==L[m]) exit ('success')
If(x<L[m]) search(x,SubList(L,1..m-1)
If(x>L[m]) search(x,SubList(L,m+1..n)
```

3. Kosten für erfolgreiche Binärsuche

$$C_{\min}(n) = 1 \qquad C_{\max}(n) = \lceil \log_2(n+1) \rceil \qquad C_{avg}(n) \approx \log_2(n+1) - 1 \quad n \to \infty$$

4. Verbesserung von $O(n)$ auf $O(\log n)$.

Die Funktionen "floor" und "ceiling"

$\lfloor x \rfloor$, manchmal auch $\mathrm{floor}(x)$ beschreibt die größte ganze Zahl nicht größer als x, i.e., den "ganzzahligen Anteil".
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 $\lfloor k + x \rfloor = k + \lfloor x \rfloor$

$\lceil x \rceil$, manchmal auch $\mathrm{ceiling}(x)$ bezeichnet die kleinste 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

Prinzip der sortierte Datenbestand wird in Sprüngen überquert um den Abschnitt zu lokalisieren, in dem der Schlüssel enthalten sein muss:

$$L: \qquad 1 \ldots m \ldots 2m \ldots 3m \ldots 4m \ldots \ldots n$$

Einfache Sprungsuche:

- konstante Sprünge zu den Position m, $2m$, etc
- Sobald $x < L[i]$ mit $i = jm$, $j = 1, 2, \ldots$, wird der Abschnitt von $L[(j-1)m+1]$ bis $L[jm]$ sequentiell nach x durchsucht

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

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

Optimale Sprungweite: Minimiere $C_{avg}(n)$ als Funktion von m
Rechnung siehe Tafel!
Optimale Sprungweite: $m = \sqrt{(a/b)n}$
Komplexität: $O(\sqrt{n})$

Zwei-Ebenen-Sprungsuche

- Prinzip: Statt sequentieller Suche im lokalisierten Abschnitt wird wiederum eine Quadratwurzel-Sprungsuche angewendet, bevor dann sequenziell gesucht wird
- Mittlere Suchkosten: $C_{avg}(n) \leq \frac{1}{2}a\sqrt{n} + \frac{1}{2}b\sqrt[4]{n} + \frac{1}{2}c\sqrt[4]{n}$ a Kosten eines Sprungs auf der ersten Ebene
  b Kosten eines Sprungs auf der zweiten Ebene
  c Kosten eines sequentiellen 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 güstige Kosten wie Binärsuche
- Anwendung: Sprungsuche vorteilhaft, wenn Binärsuche nicht anwendbar ist (z.B. bei blockweisem Einlesen der sortierten Sätze vom Externspeicher)

Exponentielle Suche

- **Anwendung**: wenn Länge des Suchbereichs n zunächst unbekannt bzw. sehr groß 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. für $i > 1$ gilt für den auf diese Weise bestimmten Suchabschnitt: $L[i/2] < x \leq L[i]$
  3. suche innerhalb des Abschnitts mit irgendeinem Verfahren
- **Analyse** Annahme: Die sortierte Liste enthält nur positive, ganzzahlige Schlüssel ohne Duplikate
  $\longrightarrow$ Schlüsselwerte wachsen mindestens so stark wie die Indizes i der Elemente
  $\longrightarrow$ i wird höchstens $\log_2 x$ mal verdoppelt, denn die Zahl der Sprünge ist gleich $log_2 i$
  Bestimmung des gesuchten Intervalls erfordert maximal $log_2 x$ Schlüsselvergleiche
  Suche innerhalb des Abschnitts (z.B. mit Binärsuche) erfordert auch höchstens $log_2 x$ Schlüsselvergleiche
- Gesamtaufwand $O(log_2 x)$

Interpolationssuche

Idee: Schnellere Lokalisierung des Suchbereichs indem Schlüsselwerte selbst betrachtet werden, um "Abstand" zum Suchschlüssel x abzuschätzen.
Vorgehensweise: Nächste Suchposition p wird aus den Werten u und v der Unter- und Obergrenze des aktuellen Suchbereichs $[u, v]$ wie folgt berechnet

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

Analyse: Sinnvoll, wenn Schlüsselwerte im betreffenden Bereich einigermaßen gleichverteilt. Im Mittel dann nur $1 + \log_2 \log_2 n$ Vergleiche
Im Extremfall von besonders irregulärer Schlüsselverteilung aber $O(n)$.

Häufigkeitsgeordnete lineare Listen

- Prinzip: Ordne die Liste nach Zugriffshäufigkeiten
  Sinnvoll, falls die Zugriffshäufigkeiten für die einzelnen Elemente einer Liste sehr unterschiedlich sind
- Analyse: Mittlere Suchkosten:
  $C_{avg}(n) = 1p_1 + 2p_2 + 3p_3 + ... + np_n = \sum_{i=1}^{n} ip_i$ für Zugriffswahrscheinlichkeiten p_i für Element i und bei sequenzieller Suche $\rightarrow$ zur Minimierung der Suchkosten sollte Liste direkt so aufgebaut oder umorganisiert werden, 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% (insgesamt 4%) der Daten.
  Erwarteter Suchaufwand $C_{avg}(n) = ???$
  ZUM NACHDENKEN ZU HAUSE!
- Problem in der Praxis: Zugriffshäufigkeiten meist vorab nicht bekannt
  $\rightarrow$ 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 Suchverhalten an (dazu gibt es verschiedene Regeln)
- Ansatz 1: FC-Regel (Frequency count)
  - führe einen Häufigkeitszähler pro Element
  - jeder Zugriff auf ein Element erhöht dessen Häufigkeitszähler um 1
  - falls erforderlich, wird danach die Liste lokal neu geordnet, so dass die Häufigkeitszähler der Elemente eine absteigende Reihenfolge bilden
  - hoher Wartungsaufwand und Speicherplatzbedarf

Selbstorganisierende Listen

- Ansatz 2: T-Regel (Transpose)
  - das Zielelement eines Suchvorgangs wird dabei 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 (Lokalität kann gut genutzt werden)

Auswahlproblem

Aufgabe: Finde das i-kleinste Element in einer Liste L mit n Elementen

— 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

```
j := 0
solange j < i:
Bestimme kleinstes Element in L und entferne es aus L ;
erh{\"o}he j um 1
gib Minimum der Restliste als Ergebnis zur{\"u}ck
```

Komplexität (für nichttriviales i): $O(n^2)$

JEDOCH: es geht besser

Divide-and-Conquer

Allgemeines Verfahren, um große Probleme schneller zu lösen:

1. Wenn das Problem klein ist, löse es mit Standardverfahren und höre auf, ansonsten zerlege das Problem in zwei oder mehrere Teilprobleme.
2. Löse jedes Teilproblem einzeln, und zwar rekursiv nach der gleichen Methode.
3. Konstruiere die Gesamtlösung aus den Teillösungen.

Dieses 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

Sequenzielle Suche $\longrightarrow$ Binärsuche
Teilungsschritt: Die Liste wird in der Mitte geteilt
Triviale Liste: Liste der Länge 1, wird mit sequenzieller Suche durchsucht
Gesamtlösung: Erfolgreich, falls in einem Teil gefunden
Unser Algorithmus zur Binärsuche lässt sich umformen zu:

```
Falls die Liste die L{\"a}nge 1 hat: Pr{\"u}fe, ob es sich um das gesuchte
    Element handelt und gib JA oder NEIN zur{\"u}ck.
Sonst zerlege die Liste in der Mitte und pr{\"u}fe durch Vergleich des
    gesuchten Elements mit der Trennstelle, in welcher H{\"a}lfte das
    gesuchte Element liegen muss.
Wende diesen Algorithmus rekursiv auf diese Teilliste an.
```

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

Maximale Teilsumme

Idee: Wir können wieder die gegebene Folge in zwei Teile zerlegen und die maximalen Teilsummen für beide Teile bestimmen.
ABER: Die maximale Teilfolge könnte gerade die Schnittstelle einschließen
DESHALB: Zusätzliche Größen: Rechtes und linkes Randmaximum einer Folge F

- rechte Randfolge von F = Teilfolge von F, die bis zum rechten Rand (Ende) von F reicht
- rechtes Randmaximum von F: maximale Summe aller rechten Randfolgen
- analog: linke Randfolge, linkes Randmaximum

Tag	1	2	3	4	5	6	7	8	9	10
Δ	+5	-6	+4	+2	-5	+7	-2	-7	+3	+5

Maximale Teilsumme

Rekursiver (Divide-and-Conquer-) 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 zwei gleich große Hälften links und rechts
  - bestimme maximale Teilsumme, m_l, sowie rechtes Randmaximum, r_R, von links
  - bestimme maximale Teilsumme, m_r, sowie linkes Randmaximum, l_R, von rechts
  - maximale teilsumme $= \max(m_l, r_R + l_R, m_r)$

Komplexitätsverbesserung?

Auswahlproblem

Suche i-kleinstes Element von n paarweise unterschiedlichen Elementen

1. bestimme Pivot-Element p
2. Teile die n Elemente bezüglich p in 2 Gruppen:
   Gruppe 1 enthält die k Elemente $< p$; Gruppe 2 die $n - k - 1$ Elemente $> p$.
3. falls $i = k + 1$, dann ist p das Ergebnis
   falls $i \leq k$ wende das Verfahren rekursiv auf Gruppe 1 an
   falls $i > k + 1$ verwende Verfahren rekursiv zur Bestimmung des $i - (k + 1)$-te Element in Gruppe 2

   ```
   [--------------L---------]p
   [....L<.....]p[....L>.....]
   ```
4. **Komplexität:** In Schritt 2 wird die ganze verbleibende Liste durchlaufen. Im ersten Durchlauf n, im zweiten ca. $n/2$, dann ca. $n/4$ usw., insgesamt ca. $2n$, also $O(n)$
5. Verbesserung von $O(n^2)$ zu $O(n)$

Divide-and-Conquer

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

Einfach zum Nachdenken

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

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 z.B. beim Suchen), so sind es insgesamt $r = \log_2 n$ Teilprobleme
4. 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)
5. der Faktor $\log n$ bei der Komplexität entsteht auf natürliche Weise durch Teile-und-Herrsche

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