Algorithmen und Datenstrukturen
Suchen

Matthias Teschner
Graphische Datenverarbeitung
Institut für Informatik
Universität Freiburg

SS 12

Lernziele der Vorlesung

- Algorithmen
  - Sortieren, Suchen, Optimieren
- Datenstrukturen
  - Repräsentation von Daten
  - Listen, Stapel, Schlangen, Bäume
- Techniken zum Entwurf von Algorithmen
  - Algorithmenmuster
  - Greedy, Backtracking, Divide-and-Conquer
- Analyse von Algorithmen
  - Korrektheit, Effizienz

Überblick

- Einführung
- Sequentielle / lineare Suche
- Binäre Suche
- Exponentielle Suche
- Interpolationssuche
- i-kleinstes Element
- Selbstanordnende Listen

Suchproblem

- Eingabe: Folge von Zahlen $< a_1, a_2, \ldots, a_n >$
  und Suchelement
- Ausgabe: Suche erfolgreich / nicht erfolgreich.
  Bei Erfolg: Index eines Elements der Folge a, dessen Wert mit dem Suchschlüssel übereinstimmt.
  Definition, wie mit mehreren Lösungen umgegangen wird.

- Eingabemenge als Feld oder verkettete Liste repräsentiert
- Suchverfahren lösen das durch die Eingabe-Ausgabe-Relation beschriebene Suchproblem.

Suchen

- grundlegende Operation auf Datenmengen
  - Stichwort in einem Wörterbuch
  - Telefonnummer in einem Verzeichnis

- elementare Suchverfahren
  - Schlüssel sind Zahlen
  - Suche ist erfolgreich oder erfolglos
  - Erfolgreiche Suche liefert Position des gesuchten Element
  - Verfahren basieren auf Vergleichsoperationen
  - Daten sind linear als Feld oder Liste repräsentiert
    - Feld: Zugriff auf Element über Index
    - Liste: Zugriff auf Element über Referenz oder mitgeführten Index
  - Hashverfahren und Baumstrukturen werden nicht berücksichtigt

Überblick

- Einführung
- Sequentielle / lineare Suche
- Binäre Suche
- Exponentielle Suche
- Interpolationssuche
- i-kleinstes Element
- Selbstanordnende Listen

Prinzip

- überprüfe sequentiell alle Elemente des Feldes (brute force search)

- ```
  int bruteForceSearch(int[] a, int k) {
    int i=0;
    if (a==null || a.length==0) return -1;

    while (i<a.length && a[i]!=k) i++;

    if (i<a.length) return i;
    else return -1;
  }
  ```

Zwei Abfragen pro Durchlauf

Element gefunden

Element nicht gefunden
  ```

Prinzip – Sentinel-Methode

- füge das gesuchte Element an die erste oder letzte Stelle des Feldes ein, spart eine Abfrage pro Durchlauf

- ```
  int seqSearch(int[] a, int k) {
    a[0] = k;
    int i = a.length;
    do i--; while (a[i]!=k);
    if (i!=0)
      return i;
    else
      return -1;
  }
  ```

durchsuche a [**1..n**] nach k

a[0] gehört nicht zur zu durchsuchenden Menge, wird mit k initialisiert (Stopper-Element)

Eine Abfrage pro Durchlauf

Element gefunden

Element nicht gefunden
  ```

Laufzeit

- bester Fall: O (1)
- schlechtester Fall: O (n)
- mittlerer Fall: $\frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n\cdot(n+1)}{2} = \frac{n+1}{2} = O(n)$

- Feld muss im Gegensatz zu allen weiteren Strategien nicht sortiert sein.

Überblick

- Einführung
- Sequentielle / lineare Suche
- Binäre Suche
- Exponentielle Suche
- Interpolationssuche
- i-kleinstes Element
- Selbstanordnende Listen

Prinzip

- Feld ist sortiert
- gesuchtes Element wird mit mittlerem Element des Feldes verglichen
- bei Ungleichheit wird nur die jeweils linke oder rechte Hälfte des Feldes rekursiv weiter betrachtet
- jeder Vergleich mit dem jeweils mittleren Element teilt den Suchraum in zwei Hälften,
  - bis das Element gefunden wurde
  - bis der verbleibende Suchraum sich nicht weiter unterteilen lässt

Prinzip

- Suche von k auf einem sortierten Feld a (keine Liste!)

a[0]						a[m]						a[n]

- Vergleich: k==a[m] mit m = (0+n) / 2
- Fall 1: k==a[m] $\rightarrow$ fertig
- Fall 2: k < a[m] $\rightarrow$ rekursive Suche von k in a[0..m-1]
- Fall 3: k > a[m] $\rightarrow$ rekursive Suche von k in a[m+1..n]

Rekursive Implementierung

```
int binSearch(int[] a, int l, int r, int k) {

  if (l>r) return -1;

  int m = (l+r)/2;
  int e = a[m];

  if (k==e) return m;
  if (k <e) return binSearch(a,l,m-1,k);
  if (k >e) return binSearch(a,m+1,r,k);
}
```

Iterative Implementierung

```
int binSearch(int[] a, int k) {

  int l=0, r=a.length-1;

  while (l<=r) {
    int m = (l+r)/2;
    int e = a[m];
    if (k==e) return m;
    if (k <e) r=m-1;
    if (k >e) l=m+1;
  }
  return -1;
}
```

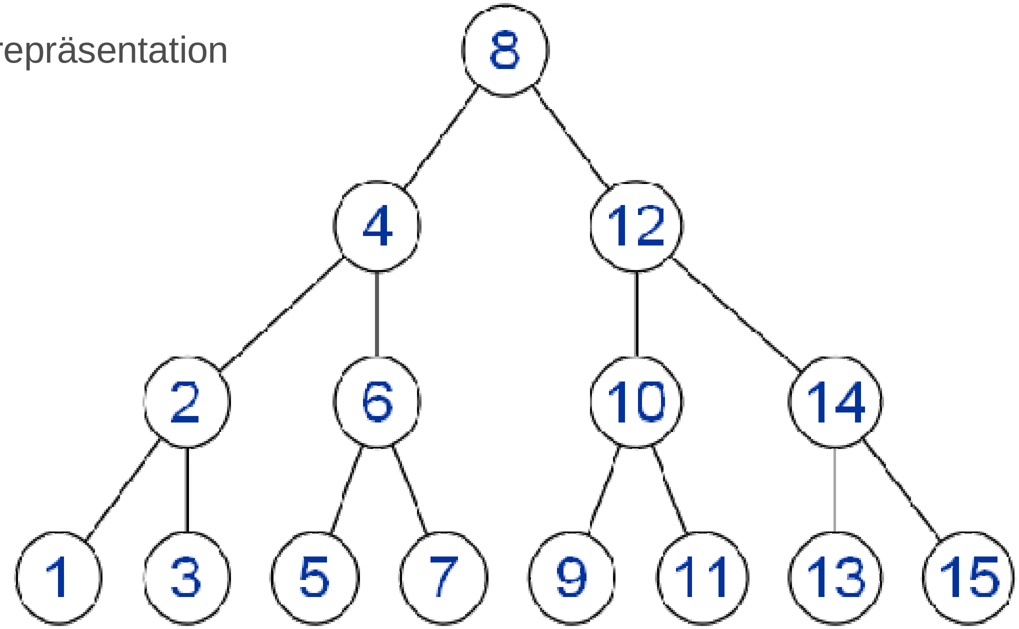
Laufzeit

- jeder Vergleich reduziert die zu durchsuchende Menge um Faktor 2
- schlechtester Fall: O (log n)
- bester Fall: O (1)

Laufzeit

- mittlere Laufzeit

Baumrepräsentation



1 Vergleich für 2^0 Elemente

2 Vergleiche für 2^1 Elemente

3 Vergleiche für 2^2 Elemente

4 Vergleiche für 2^3 Elemente

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Feldrepräsentation

Laufzeit

- mittlere Laufzeit
- Zahl der Elemente

$$\sum_{i=0}^{k-1} 2^i = 2^k - 1 = n$$

k – Zahl der Ebenen im Baum

- Zahl der Vergleiche

$$\sum_{i=0}^{k-1} 2^i (i-1) = \sum_{i=1}^{k} 2^{i-1} i = k \cdot 2^k - 2^k + 1$$

- Vergleiche / Elemente

$$\frac{\log(n+1) \cdot (n+1) - (n+1) + 1}{n} \approx \log(n+1) - 1 \in O(\log n)$$

Im Durchschnitt 1 Vergleich
weniger als die Maximalzahl
möglicher Vergleiche
(intuitiv durch Baumrepräsentation klar)

Variante

- Fibonacci-Suche
  - Vermeidung der Division bei der Aufteilung der Menge
- $F_0 = F_1 = 1;\ F_n = F_{n-1} + F_{n-2}\ (n \geq 2)$
- Verfahren
  - finde F_{n-1} und F_{n-2} mit $F_{n-1} + F_{n-2} = F_n \geq n$
  - vergleiche k mit a[F_{n-2}]
  - Fallunterscheidung und Rekursion wie bei binärer Suche

1	F_{n-2}	F_n-1
F_{n-2}-1 Elemente	1 Element	F_{n-1}-1 Elemente

  - mittlerer Aufwand O (log n)

Implementierung

```
int fibSearch(int[] a, int k) {
  int n=a.length-1;
  int fib2=1; fib1=1; fib=fib1+fib2;
  while (fib-1<n) {
    fib2=fib1; fib1=fib; fib=fib1+fib2; }
  int offset=0;
  while (fib>1) {
    int i=min(offset+fib2,n);
    int e=a[i];
    if (k==e) return i;
    if (k<e) {
      fib=fib2; fib1=fib1-fib2; fib2=fib-fib1; }
    if (k>e) {
      offset=i;
      fib=fib1; fib1=fib2; fib2=fib-fib1; }
  }
  return -1
}
```

finde Fibonacci-Zahl >=n
fib1 und fib2 sind die
Vorgänger von fib

linker Rand-1

Überblick

- Einführung
- Sequentielle / lineare Suche
- Binäre Suche
- Exponentielle Suche
- Interpolationssuche
- i-kleinstes Element
- Selbstanordnende Listen

Prinzip

- n sehr groß gegenüber dem zu suchenden Schlüssel k
- Verfahren
  - ```
    j=1; while (k>a[j]) j=2*j;
    return binSearch(a,j/2,j,k);
    ```
- teste a[1], a[2], a[4], .., a[2^j]
- Aufwand
  - O(log k) Schritte in der while-Schleife
  - O(log j/2) = O(log k) Schritte für binäre Suche
    ```

Überblick

- Einführung
- Sequentielle / lineare Suche
- Binäre Suche
- Exponentielle Suche
- Interpolationssuche
- i-kleinstes Element
- Selbstanordnende Listen

Prinzip

- berücksichtige die zu erwartende Position des Schlüssels k im Feld a
- Feld a wird von Position l bis Position r durchsucht
- Annahme: Schlüsselwerte verhalten sich linear zwischen a[l] und a[r]
- Schätzung der Position t durch

$$t = l + (r - l)\frac{k - a[l]}{a[r] - a[l]}$$

Quotient liegt zwischen 0 (für k=a[l]) und 1 (für k=a[r]). Damit liegt t zwischen l und r.

- Annahme des linearen Verhaltens stimmt oft nicht mit dem realen Verhalten überein

Implementierung

```
int interpSearch(int[] a, int k) {

  int l=0, r=a.length-1;

  while (l<=r) {
    int m = l+(l-r)*(k-a[l])/(a[r]-a[l]);
    int e = a[m];
    if (k==e) return m;
    if (k <e) r=m-1;
    if (k >e) l=m+1;
  }
  return -1;
}
```

Wir vergessen natürlich nicht,
a[r]-a[l]==0 irgendwo zu behandeln.

Diskussion

- mittlere Laufzeit: O(log (log n))
- sehr langsames Wachstum: log $(10^9) \approx 30$, log (log $(10^9)) \approx 5$
- effizient bei großen Feldern und linearem Verhalten der Schlüssel
- Implementierung im Vergleich zur binären Suche lediglich bei der Berechnung des mittleren Elements aufwendiger

Überblick

- Einführung
- Sequentielle / lineare Suche
- Binäre Suche
- Exponentielle Suche
- Interpolationssuche
- i-kleinstes Element
- Selbstanordnende Listen

Naiver Ansatz

- finde das i-kleinste Element eines unsortierten Feldes a
- Ansatz
  - `j=0; while (j<i) { min=kleinstesElement(a); entferne(a,min); j++;} return min;`
  - Aufwand: O(i·n)
  - für i=n/2 (Median) ist der Aufwand $O(n^2)$
    → Sortieren und direkter Zugriff auf a [i]
    wäre schneller in O(n log n)

Verwendung eines Heaps

- min-Heap (analog zu max-Heap)
  - binärer Baum, wobei Wert eines Knotens kleiner als die Werte der beiden Nachfolgeknoten ist
  - Wurzel enthält kleinstes Element
  - Aufbau in O(n), Aktualisierung in O(log n)
- i-kleinstes Element mit Hilfe eines Heaps

```
generiereHeap(a);
j=0; while (j<i) { entferne(a,a[0]);
rekonstruiereHeap(a); j++; }
return a[0];
```

O (n)
O (i)
O (log n)

- Aufwand $O(n + i \cdot \log n)$
- mittlerer, schlechtester Fall O (n log n)

Teile-und-Herrsche-Ansatz

- Aufteilung von a bezüglich Pivotelement p

l	r
a[l..r-1]	p

  - Divide-Schritt (wie Quick-Sort) O(n)

l	m	r
a[..] < p	p	a [..] > p

  - Pivot-Element ist an Position m
    - m-l+1 == i $\rightarrow$ fertig (p ist i-kleinstes Element)
    - m-l+1 > i $\rightarrow$ rekursiver Divide-Schritt auf a[l .. m-1]
    - m-l+1 < i $\rightarrow$ rekursiver Divide-Schritt auf a[m+1.. r]

Rekursionstiefe O (log n) bis O(n) Divide-Schritt wird nur auf n, n/2, n/4, … angewendet

Implementierung

```
int findMinI (int[] a, int l, int r, int i) {

  int m = quickSortDivide(a, l, r);
  if ( m==i )
    return a[i];
  if ( m >i )
    return findMinI (a,l,m-1,i);
  if ( m <i )
    return findMinI (a,m+1,r,i);
}
```

quickSortDivide liefert die Position m des Pivotelements

Laufzeit
Teile-und-Herrsche-Ansatz

- Analog zu Quick-Sort
- bester Fall:
  - Pivotelement teilt Menge in zwei gleichgroße Mengen
    T(n) = T(n/2) + n (Quick-Sort: T(n) = 2 T(n/2) + n)
  - Fall 3 des Master-Theorems:

Die Hälfte der Menge kann jeweils verworfen werden.

$$T(n) \in \Theta\left(f(n)\right) \quad \text{falls} \quad f(n) \in \Omega\left(n^{\log_b a+\epsilon}\right) \quad \epsilon > 0$$

$$af\left(\frac{n}{b}\right) \leq cf(n) \quad 0 < c < 1$$

$$a = 1, \quad b = 2, \quad f(n) = n, \quad \log_b a = \log_2 1 = 0$$

$$f(n) \in \Omega\left(n^{\log_2 1+1}\right) \quad 1 \cdot \frac{n}{2} \leq c \cdot n \quad c = \frac{1}{2}$$

$$T(n) \in \Theta\left(n\right) \approx 2 \cdot n$$

Laufzeit

- schlechtester Fall: $T(n) = T(n-1) + O(n) \rightarrow T(n) \in O(n^2)$
- mittlerer Fall: $O(n)$
  - spezielle Strategien zum Finden des Pivotelements
  - randomisiert (wähle ein zufälliges Element)
  - Median (wähle in konstanter Zeit
    mittleres Element aus 3 oder 5 Elementen)

Überblick

- Einführung
- Sequentielle / lineare Suche
- Binäre Suche
- Exponentielle Suche
- Interpolationssuche
- i-kleinstes Element
- Selbstanordnende Listen

Prinzip

- ordne häufig gesuchte Elemente an den Anfang des Feldes / der Liste
- verwende sequentielle Suche

- nur vorteilhaft, wenn einige Elemente signifikant häufiger gesucht werden als andere
- werden alle Elemente gleich häufig gesucht, ist das Verfahren schlechter als die sequentielle Suche (sequentielle Suche plus Umordnung der Elemente)

Strategien

- MF-Regel (move to front)
  - gesuchtes Element wird Kopf der Liste / erstes Element des Feldes
- T-Regel (transpose)
  - gesuchtes Element wird mit unmittelbarem Vorgänger vertauscht
- FC-Regel (frequency count)
  - zähle Suchanfragen pro Element
  - ordne gesuchtes Element so ein, dass die Zählerwerte in der Liste / im Feld absteigend sortiert sind

Zusammenfassung

- Sequentielle / lineare Suche
  - Feld muss nicht sortiert sein, Laufzeit O(n)
- Binäre Suche
  - Feld muss sortiert sein, Laufzeit O(log n)
  - Fibonacci-Suche vermeidet Division, Laufzeit O(log n)
- Exponentielle Suche
  - sinnvoll, wenn Schlüssel k klein gegenüber Anzahl der Elemente n
  - Feld ist sortiert, Laufzeit O(log k)
- Interpolationssuche
  - Feld ist sortiert, vermutet lineares Verhalten der Schlüsselwerte
  - Laufzeit O (1), wenn die Vermutung stimmt, sonst O (log (log (n))
- i-kleinstes Element
  - naiv O(n^2), min-Heap O (n log n),
  - basierend auf Quick-Sort bei guter Pivotisierung O (n)

Nächstes Thema

- Algorithmen / Datenstrukturen
  - Hashverfahren