Algorithmen und Datenstrukturen

5. SUCHEN

Überblick

- Suchprobleme sind eine der am häufigsten Probleme in der Informatik
- Beispiele
  1. Suchen in sortierten Folgen
  2. Suchen von Zeichenketten in Text
  3. Suche von Dokumenten in Textkorpora (z.B. WWW)
  4. Allgemein Suche von Lösungen in Problemräumen
     - Spielbaumsuche
     - Plansuche
- Hier zunächst nur 1.

Suchen in sortierten Folgen

- Annahmen:
  - Folge F als Feld von numerischen Werten
  - Folge ist sortiert, d.h. wenn i<j dann F[i]<F[j]
  - Zugriff auf i-tes Element über F[i]
  - Nur Berücksichtigung des Suchschlüssels
- Beispiel: Telefonbuch - Suche nach Namen
- Zunächst: wie repräsentiert man diese Daten?
  - Kurzer Einschub zu „*Linearen Datenstrukturen*“

Definition, Beispiel

- Eine lineare Datenstruktur L ist eine Sequenz $L=(a_1,\ldots,a_n)$
- Eine lineare Datenstruktur ordnet Elemente (entweder primitive Datentypen oder komplexere Datenstrukturen) in einer linearen Anordnung an

Beispiele:

- Zahlenfolgen:

5	4	6	1	3	2

- Strings:

L	I	N	E	A	R

Atomare Operationen auf linearen Datenstrukturen

Lesen:

- get(i): Element an Position i lesen
- first(): erstes Element lesen
- last(): letztes Element lesen
- next(e): Element nach Element e lesen

Schreiben:

- set(i,e): Element an Position i auf e setzen
- add(i,e): Element e an Position i einfügen
- del(i): Element an Position i löschen

Arrays und Listen

- Es gibt prinzipiell zwei Möglichkeiten lineare Datenstrukturen zu realisieren
  - Arrays
  - (verlinkte) Listen
- Arrays belegen einen zusammenhängenden Bereich im Speicher
- Elemente einer verlinkten Liste können beliebig verteilt sein
- Ob zur Realisierung einer linearen Datenstruktur ein Array oder eine Liste verwendet wird, hängt von der Anwendung ab

Einfache verlinkte Liste von Zahlen in Java

```
public class IntegerList {
      private class IntegerListElement{
         int value;
         IntegerListElement next;
      }

      IntegerListElement first;
      int size = 0;

…
```

Einfache verlinkte Liste von Zahlen in Java

```
…
        private IntegerListElement getElement(int i){
            if(i >= size)
                throw new IllegalArgumentException();
            int idx = 0;
            IntegerListElement current = first;
            while(idx != i){
                current = current.next;
                idx ++;
            }
            return current;
        }

        public int get(int i){
            return this.getElement(i).value;
        }
…
```

Einfache verlinkte Liste von Zahlen in Java

```
…
        public int add(int pos, int val){
            IntegerListElement newElem = new IntegerListElement();
            newElem.value = val;
            if(pos > 0){
                IntegerListElement e = this.getElement(pos-1);
                newElem.next = e.next;
                e.next = newElem;
            }else{
                newElem.next = this.first;
                this.first = newElem;
            }
        }
…
```

Arrays und Listen

- Arrays werden meist für statische Datenstrukturen verwendet (=die Länge des Arrays wird nicht verändert)
- Listen werden meist für dynamische Datenstrukturen verwendet (=variable Länge)
- Eigenschaften von Arrays:
  - \+ Schneller Zugriff auf Einzelelemente durch Index
  - \- Einfügen aufwändig
- Eigenschaften von Listen:
  - \- Direktzugriff kann ineffizient sein
  - \+ Manipulation relativ effizient

Sortieren und Suchen

- Suchen und Sortieren sind voneinander abhängige Operationen; Ansätze:
  1. Elemente nie sortieren macht Suche aufwändig
  2. Elemente sortieren erleichtert Suche, kann aber selber aufwändig sein
- Falls Elemente hinzugefügt/gelöscht werden ist diese Problematik noch sichtbarer
  1. Nur ein unsortiertes Element macht Suche aufwändig
  2. Bei jeder Einfügung/Löschung zu sortieren ist aufwändig

- Spezielle dynamische Datenstrukturen erlauben eine automatische und effiziente Sortierung bei Einfügung/ Löschung (siehe Kapitel 5)

Lineare Datenstrukturen in Java

- Arrays:
  - `int[] arr = new int[10];`
  - `arr[1] = 4;`
- Listen:
  - `List<Integer> myList = new LinkedList<Integer>();`
  - `myList.add(5);`
- Neben `LinkedList` unterstützt Java eine Reihe weiterer Listenimplementierungen mit unterschiedlichen Vor- und Nachteilen
- Schnittstelle `List<Type>` beinhaltet die gemeinsamen Methoden

Suche

Problembeschreibung

- Eingabe: Folge F (mit n Elementen) von Zahlen und Suchelement k
- Ausgabe:
  - Suche erfolgreich oder nicht erfolgreich
  - Erfolgreich: Index p ($0 \leq p < n$)
    - Evtl. Festlegung was bei Mehrfachvorkommen passiert (normalerweise das erste Vorkommen)
  - Nicht erfolgreich: -1

Merkmale der Suche

- Schlüssel (für Suchelemente), z.B. Zahlen
- Suche ist erfolgreich oder erfolglos
- Basiert auf Vergleichsoperationen
- Zunächst: Daten als Feld (Array) oder Liste dargestellt

Suchverfahren

- Sequentielle Suche
- Binäre Suche
- Fibonacci Suche

Algorithmen und Datenstrukturen

5.1. SEQUENTIELLE SUCHE

Sequentielle Suche

- Idee:
  - Überprüfe ob das erste Element der Liste mit dem gesuchten Element übereinstimmt; falls ja, gib aktuellen Index zurück
  - Wiederhole mit dem nächsten Element
  - Wenn man am Ende der Folge das Element nicht gefunden hat war die Suche erfolglos (gib -1 zurück)

Sequentielle Suche - Algorithmus

```
int SeqSearch(int[] f, int k) {
    / * output: Position p  (0 ≤ p ≤ n-1) */
    int n = f.length;
    for (int i = 0; i < n; i++) {
        if (f[i] == k)
            return i;
    }
    return -1;
}
```

- Int[ ] ist sortierte Folge von int
- int k ist Such-schlüssel
- Folge F hat Länge n

Sequentielle Suche - Analyse

Theorem (Terminierung)

Für endliche Eingaben terminiert der Algorithmus **SeqSearch** nach endlicher Zeit.

Theorem (Korrektheit)

Falls das Array f ein Element k enthält, gibt **SeqSearch(f,k)** den Index des ersten Vorkommens von k zurück. Ansonsten gibt **SeqSearch(f,k)** den Wert -1 zurück.

Sequentielle Suche - Analyse

```
int SeqSearch(int[] f, int k) {
   / * output: Position p  (0 ≤ p ≤ n-1) */
int n = f.length;
for (int i = 0; i < n; i++) {
   if (f[i] == k) {
         return i;
}
return -1;
```

Hinweis:
Folge muss nicht
Sortiert sein!

Warum (n+1)/2?

	Anzahl der Vergleiche
Bester Fall	1
Schlechtester Fall	n
Durchschnitt (erfolgreiche Suche)	(n+1)/2
Durchschnitt (erfolglose Suche)	n

Sequentielle Suche - Analyse

Theorem (Laufzeit)

Der Algorithms **SeqSearch** hat eine Worst-Case Laufzeit von $\Theta(n)$.

Algorithmen und Datenstrukturen

5.2. BINÄRE SUCHE

Binäre Suche

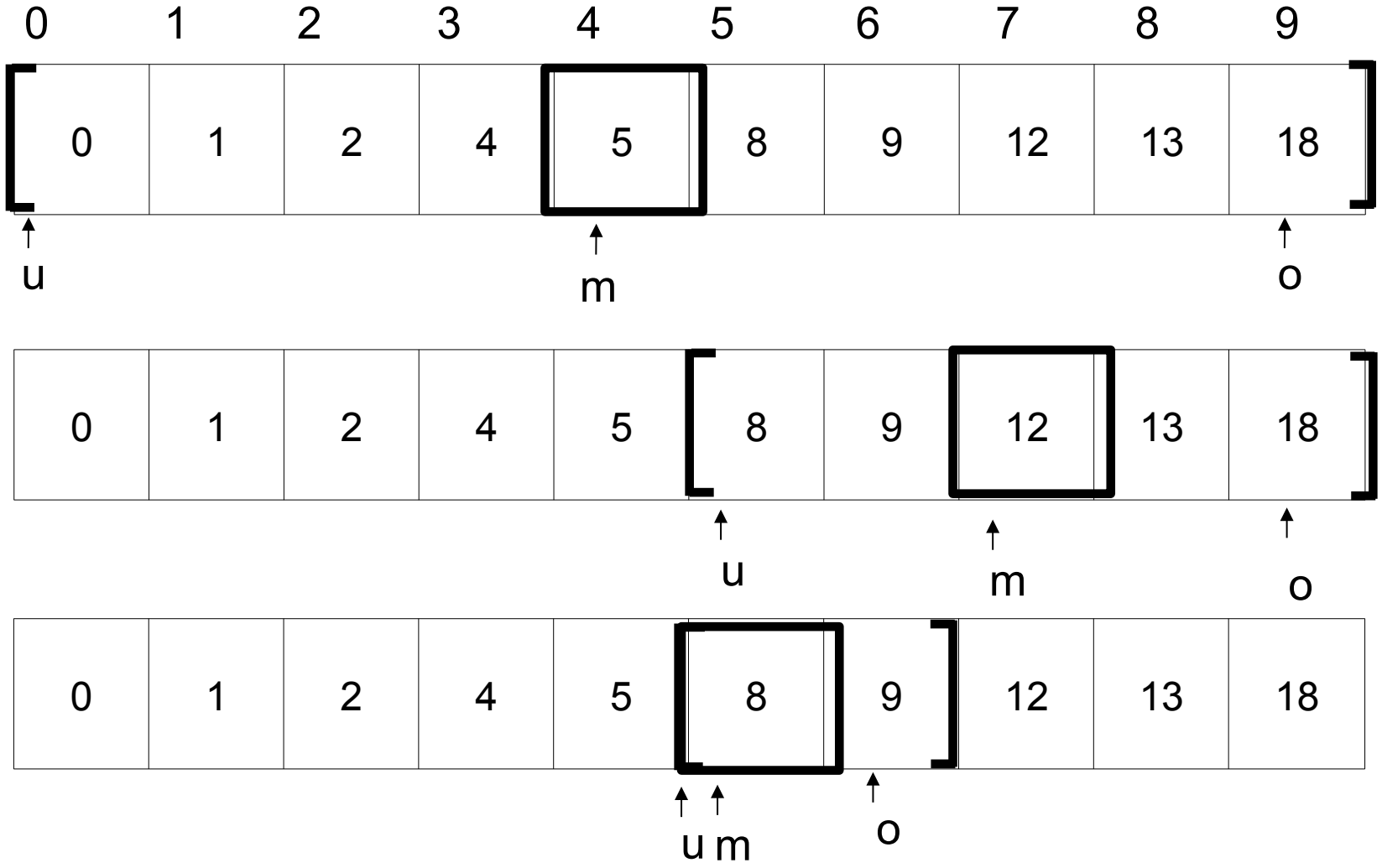
Wie könnte die Suche effizienter werden?

- Ausnutzung der Sortierung
- Prinzip:
  1. Wähle den mittleren Eintrag und prüfe ob gesuchter Wert in der ersten oder in der zweiten Hälfte der Liste ist.
  2. Fahre rekursiv mit der Hälfte fort, in der sich der Eintrag befindet.

→ Entwurfsmuster des Divide-and-Conquer

Binäre Suche - Beispiel

Suchschlüssel k = 8



Binäre Suche - rekursiver Algorithmus

```
int BinarySearch(int[] f, int k){
  /*input: Folge f der Länge n, Schlüssel k */
  /*output: Position p */

   return  BinarySearchRec(f, k, 0, f.length-1);
}
```

initialer Aufruf

```
int BinarySearchRec (int[] f, int k, int u, int o) {
  /* input: Folge f der Länge n, Schlüssel k,
        untere Schranke u, obere Schranke o */
  /* output: Position p */
   m = (u+o)/2;
   if  (f[m] ==  k) return m;
   if  ( u == o) return -1;
   if  (f[m] >  k) return BinarySearchRec(f,k,u,m-1);
   return BinarySearchRec(f,k,m+1,o);
}
```

Binäre Suche - Analyse

Theorem (Terminierung)

Der Algorithmus BinarySearch terminiert für jede endliche Eingabe f nach endlicher Zeit.

Beweis

In jedem Rekursionsschritt verkürzt sich die Länge des betrachteten Arrays f um mehr als die Hälfte. Nach endlichen vielen Schritten hat das Array nur noch ein Element und die Suche endet entweder erfolgreich oder erfolglos. Falls das Element vorher gefunden wird terminiert der Algorithmus schon früher. □

Binäre Suche - Analyse

Theorem (Korrektheit)

Falls das Array f ein Element k enthält, gibt **BinarySearch(f,k)** den Index *eines* Vorkommens von k zurück. Ansonsten gibt **BinarySearch (f,k)** den Wert -1 zurück.

Beweis

Durch verallgemeinerte Induktion nach der Länge n von f.

n=1: Der erste Aufruf von BinarySearch ist BinarySearchRec(f,k,0,0) und somit m=0. Ist f[0]=k so wird 0 zurückgegeben, ansonsten -1 da 0=0.

Binäre Suche - Analyse

n>1: Der erste Aufruf von BinarySearch ist BinarySearchRec(f,k,0,n-1) und somit m=(n-1)/2. Ist f[m]=k, so wird m zurückgegeben. Ansonsten wird rekursiv auf f[0...m-1] oder f[m+1...n] fortgefahren. Da die Folge sortiert ist, kann k nur in einem der beiden Teile vorhanden sein. □

Gibt BinarySearch auch immer den Index des ersten Vorkommens von k zurück (wie die sequentielle Suche)?

Binäre Suche - Analyse

Wie viele Rekursionsschritte sind nötig?

- Nach dem ersten Teilen der Folge: noch n/2 Elemente
- Nach dem zweiten Schritt: n/4 Elemente
- Nach dem dritten Schritt: n/8 Elemente

…

- Allgemein: im i-ten Schritt max. $n/2^i$ Elemente
  d.h. $\log_2 n$ Vergleiche bei der Suche

	Anzahl der Vergleiche
Bester Fall	1
Schlechtester Fall	$\approx \log_2 n$
Durchschnitt (erfolgreiche Suche)	$\approx \log_2 n$
Durchschnitt (erfolglose Suche)	$\approx \log_2 n$

Binäre Suche - Analyse

Rekursionsgleichung für Binäre Suche (erfolglose Suche):

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1 \\ T(n/2) & \text{sonst} \end{cases}$$

Auflösen von T(n) nach Induktion ergibt T(n)=Θ (log n) für erfolglose Suche.

Binäre Suche - Analyse

Theorem (Laufzeit)

Der Algorithms **BinarySearch** hat eine Worst-Case Laufzeit von $\Theta(\log n)$.

Beweis

Nach Vorüberlegungen. □

Binäre Suche – iterativer Algorithmus

```
int  BinarySearch(int[] f, int k) {
   /* input: Folge f der Länge n, Schlüssel k */
   /*  output: Position p  (0 ≤ p ≤ n-1)  */

   int u = 0;
   int o = f.length-1;
   int m;
   while (u <= o) {
       m = (u+o)/2;
       if  (f[m] ==  k)
           return m;
       else
           if (k < f[m])
               o = m-1;
           else
               u = m+1;
  }
  return -1;
}
```

Initialisierung

Wiederholen bis Schranken erreicht sind

Abbruchbedingung

Obere bzw. untere Schranke Anpassen
→ untere/obere Hälfte

Algorithmen und Datenstrukturen

5.3. FIBONACCI-SUCHE

Nachteile der binären Suche

- Binäre Suche ist der am häufigsten verwendete Algorithmus zur Suche in sortierten Arrays
- Die Sprünge zu verschiedenen Testpositionen sind allerdings immer recht groß
  - Dies kann nachteilig sein, wenn das Array nicht vollständig im Speicher vorliegt (oder bei Datenträgertypen wie Kassetten)
  - Neue Positionen werden durch Division berechnet (je nach Prozessor ist dies eine aufwändigere Operation als Addition und Subtraktion)

- Fibonacci Suche als eine weitere Alternative

Fibonacci-Zahlen

Erinnerung:

- Die Folge der Fibonacci-Zahlen F_n für $n \geq 0$ ist definiert durch
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_2 = 1$
  - $F_i = F_{i-1} + F_{i-2}$ (für $i > 1$)

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
F_i	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377

Fibonacci Suche

- Anstatt wie bei der binären Suche das Array in gleich große Teile zu teilen, wird das Array in Teilen entsprechend der Fibonacci-Zahlen geteilt
  - Betrachte zunächst das Element an Indexposition m, wobei m die größte Fibonaccizahl ist, die kleiner als die Arraylänge ist
  - Fahre rekursiv mit dem entsprechenden Teilarray fort

Fibonacci Suche in Java (rekursiv)

```
public int fibonacciSearch(int[] arr,int elem){
    return fibonacciSearchRec(arr,elem,0,arr.length-1);
}

public int fibonacciSearchRec(int[] arr,int elem,int u,int o){
    int k = 0;
    while(fib(k) <= o-u) k++;
    if(elem==arr[u+fib(k-1)])
            return u+fib(k-1);
    if(u==o)
            return -1;
    if(elem<arr[u+fib(k-1)])
            return fibonacciSearchRec(arr,elem,u,u+fib(k-1)-1);
   return fibonacciSearchRec(arr,elem,u+fib(k-1)+1,o);
}
```

Beispiel

9	19	21	34	87	102	158	159	199	205

133?

1. `fibonacciSearchRec(arr,133,0,9)`
   1. `fib(6)=8 ≤ 9-0` (und maximal)
   2. `arr[fib(6)+0] = arr[8] = 199 > 133`
2. `fibonacciSearchRec(arr,133,0,7)`
   1. `fib(5)=5 ≤ 7-0` (und maximal)
   2. `arr[fib(5)+0] = arr[5] = 102 < 133`
3. `fibonacciSearchRec(arr,133,6,7)`
   1. `fib(2)=1 ≤ 7-6` (und maximal)
   2. `arr[fib(2)+6] = arr[7] = 159 > 133`
4. `fibonacciSearchRec(arr,133,6,6)`
   1. Suche erfolglos

i	0	1	2	3	4	5	6
F_i	0	1	1	2	3	5	8

Beispiel

9	19	21	34	87	102	158	159	199	205

87?

1. fibonacciSearchRec(arr,133,0,9)
   1. fib(6)=8 ≤ 9-0 (und maximal)
   2. arr[fib(6)+0] = arr[8] = 199 > 87
2. fibonacciSearchRec(arr,133,0,7)
   1. fib(5)=5 ≤ 7-0 (und maximal)
   2. arr[fib(5)+0] = arr[5] = 102 > 87
3. fibonacciSearchRec(arr,133,0,4)
   1. fib(4)=3 ≤ 4-0 (und maximal)
   2. arr[fib(4)+0] = arr[3] = 34 < 87
4. fibonacciSearchRec(arr,133,4,4)
   1. Suche erfolgreich

i	0	1	2	3	4	5	6
F_i	0	1	1	2	3	5	8

Fibonacci Suche - Analyse

- Die Fibonacci Suche hat dieselbe Komplexität wie die binäre Suche:
  - Anzahl Vergleiche im besten Fall: 1
  - Anzahl Vergleiche durchschnittlich (erfolgreich/erfolglos) und im schlechtesten Fall: $\log_2 n$
  - Insgesamt $\Theta(\log n)$
  - Terminierung- und Korrektheitsbeweise analog
- Die nötigen Fibonaccizahlen können vorab berechnet und in einem (statischen) Array gespeichert werden (für Arrays mit weniger als 100.000.000 Elementen werden "nur" die ersten 50 Fibonaccizahlen benötigt)
- Nur Subtraktion/Addition als Operation
- "Sprünge" zwischen Arrayposition im Durchschnitt geringer als bei binärer Suche

Algorithmen und Datenstrukturen

5. SUCHEN
ZUSAMMENFASSUNG

Zusammenfassung

- Sequentielle Suche
  - Funktioniert auch bei unsortierten Folgen
  - Laufzeit $\Theta(n)$
- Binäre Suche
  - Laufzeit $\Theta(\log n)$
- Fibonacci-Suche
  - Laufzeit $\Theta(\log n)$