PRAKTISCHE INFORMATIK III

Bernd Neumann

Worum geht es in P3?

"Programmieren im Großen"

Welche Strukturen und Methoden stellt die Informatik für die Gestaltung großer, komplexer Systeme bereit?

Wichtige Aspekte sind:

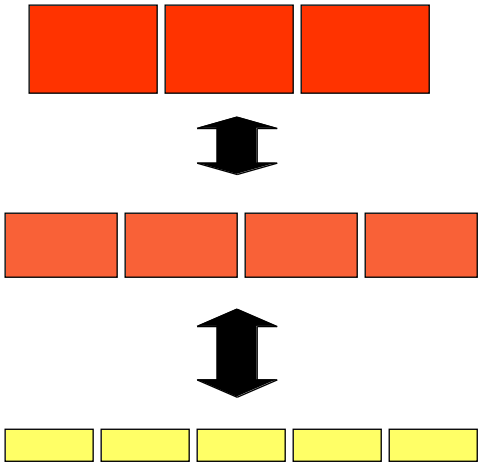
- **Zusammengesetzte, dynamische Strukturen**
- **Effizienz von Verfahren**
- **Verteilte Systeme**
- **Wiederverwendbarkeit von Komponenten, Rahmenwerken und Systemschalen**
- **Durchschaubarkeit, Komplexitätsbeherrschung**

Die Modellierungsaufgabe

Ausdrucksmittel zur Beschreibung von Anwendungsproblemen

problemnahe Datenstrukturen und Methoden

Ausdrucksmittel von Programmiersprachen



3

Inhaltsüberblick (1)

TEIL 1:

- **Dynamische Datenstrukturen:**
  - **Listen, Keller, Bäume, Graphen**
  - **Algorithmen und ihre Bewertung**
- **Anwendungen von dynamischen Datenstrukturen:**
  - **Behälterklassen, Suchen in geordneten Datenbeständen**

4

Inhaltsüberblick (2)

TEIL 2:

- **Daten- und Wissensmodellierung:**
  - **programminterne und persistente Datenstrukturen**
  - **semantische Netze**
  - **relationale Datenbankmodelle**
  - **objektorientierte Datenbankmodelle**
- **Inferenzdienste:**
  - **Regelbasierte Systeme, Deduktive Datenbanken**
  - **Inferenzdienste von Beschreibungslogiken**

5

Inhaltsüberblick (3)

TEIL 3:

- **Nebenläufige Programmierung:**
  - **Prozeßbegriff, Synchronisation, Kommunikation zwischen Prozessen, Threads**
- **Anwendungen nebenläufiger Programmierung:**
  - **Betriebsmittelverwaltung, prozeßorientierte Simulation**

6

Inhaltsüberblick (4)

TEIL 4:

- **Agententechnologie:**
  - **BDI-Architektur**
  - **Kooperierende Agenten**
  - **Suchmaschinen**
  - **Agenten für E-Commerce**

7

... mit wem Sie es zu tun haben

Bernd Neumann

- **Diplom in Elektrotechnik (TH Darmstadt)**
- **S.M. und Ph.D. in Informationstheorie (MIT, USA)**
- **Professor für Informatik an der Uni Hamburg seit 1982**
- **Gründer und Leiter des Labors für Künstliche Intelligenz (LKI)**
- **FB-Beauftragter für Technologietransfer**

Forschungsinteressen:

- **Künstliche Intelligenz**
- **Bildverstehen**
- **Wissensmanagement**

Motto:

Dem Ingeniör ist nichts zu schwör!

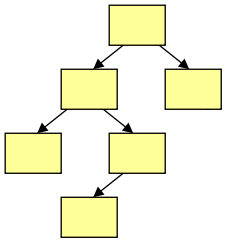
8

Dynamische Datenstrukturen

Datenstrukturen, deren Struktur während der Laufzeit verändert werden können

- **problemadaptive Modellierung**
- **flexibler Umgang mit Ressourcen**

- **Was heißt "Struktur"?**
  Gebilde aus Teilen (Komponenten) mit Beziehungen (Relationen)
- **Ändern einer Struktur**
  - **Hinzufügen, Modifizieren und Löschen von Komponenten**
  - **Ändern von Beziehungen**



9

Grundlegende dynamische Datenstrukturen

- **Listen**
  z.B. Hitliste, Benutzerliste, Einkaufsliste
- **Keller (Stack, Stapel)**
  z.B. Speicherstruktur zur Bearbeitung geschachtelter Ausdrücke
- **Schlangen (Queue)**
  z.B. Warteschlangen von Aufträgen
- **Mengen**
  allgegenwärtig
- **Bäume**
  z.B. Verwandschaftsbeziehungen
- **gerichtete Graphen**
  z.B. Suchräume bei schrittweisem Problemlösen

10

Listen (Sequenzen, Folgen)

Liste = Datentyp, deren Wertemenge endliche Folgen eines Grundtyps umfaßt

- **Listenelemente besitzen eine Ordnung (Reihenfolge)**
- **Listen können Elemente des Grundtyps mehrfach enthalten (Duplikate)**

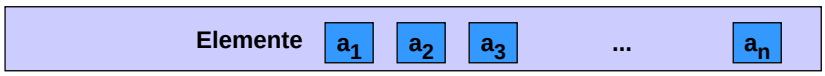
Wir betrachten getrennt:

- **Nützliche Listenoperationen aus Anwendungssicht**
- **Implementierungsformen**

11

Einfache Grundoperationen auf Listen

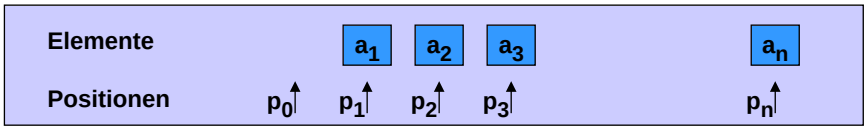
Liste



ADT	**list1**		
TYPEN	**list, elem**		
OPERATIONEN	**empty**	**:**	**=> list**
	first	**: list**	**=> elem**
	rest	**: list**	**=> list**
	append	**: elem x list**	**=> list**
	concat	**: list x list**	**=> list**
	isempty	**: list**	**=> bool**

12

Operationen auf Listen mit expliziten Positionen



ADT	list2		
TYPEN	list, elem, pos		
OPERATIONEN	empty	:	=> list
	front, last	: list	=> pos
	next, previous	: list x pos	=> pos + null
	bol, eol	: list x pos	=> bool
	insert	: list x pos x elem	=> list
	delete	: list x pos	=> list
	concat	: list x list	=> list
	isempty	: list	=> bool
	find	: list x (elem => bool)	=> pos + null
	retrieve	: list x pos	=> elem

13

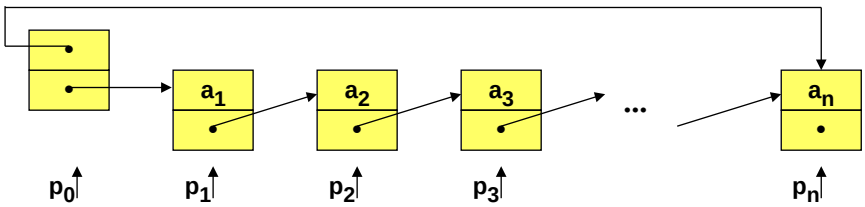
Implementierungen von Listen

Übersicht

1. **Einfach verkettete Liste**
   - unterstützt nur einen Teil der Operationen effizient (nicht: previous)
2. **Doppelt verkettete Liste**
   - unterstützt alle Operationen effizient
3. **Sequentielle Darstellung im Array**
   - keine Zeiger erforderlich
   - Umkopieren
   - Verschwenden von Speicherplatz
   - Positionsinformation nicht stabil
4. **Einfach oder doppelt verkettete Liste im Array**
   - prinzipiell wie 1) und 2)
   - Speicherverwaltung im Array erforderlich

14

Einfach verkettete Liste



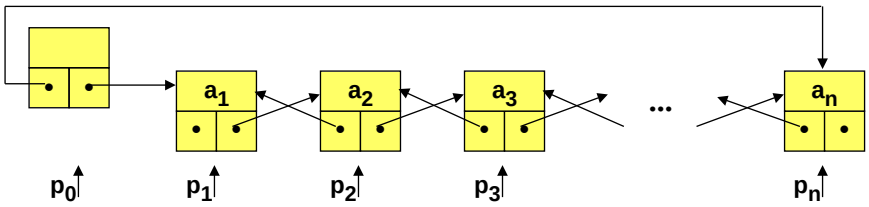
Listenkopf

Realisierung z.B. durch diese Typdeklarationen (Pascal, Modula):

```
type   list          = record head, last : ^listelem;
       listelem      = record value      : (beliebiger Datentyp)
                             succ       : ^listelem
                       end
```

15

Doppelt verkettete Liste



Listenkopf

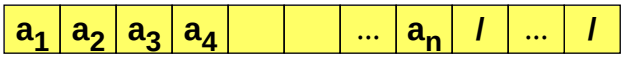
Realisierung z.B. durch diese Typdeklarationen (Pascal, Modula):

```
type   list          = ^listelem;
       pos           = ^listelem;
       listelem      = record value      : (beliebiger Datentyp);
                             pred, succ  : ^listelem
                       end
```

16

Darstellungen im Array

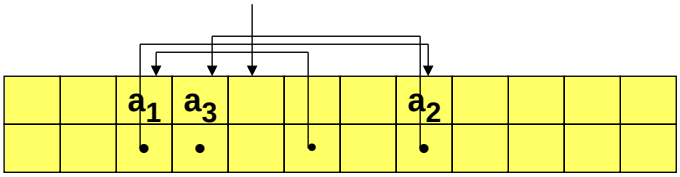
Sequentielle Darstellung im Array



0 1 2 3 n-1

Positionen entsprechen Array-Indizes

Einfach oder doppelt verkettete Liste im Array



Freispeicherverwaltung durch zusätzliche Verkettung freier Zellen

17

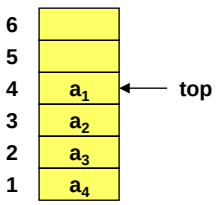
Keller (Stack, Stapel)

Spezialfall einer Liste (ADT list1)

Traditionelle Bezeichnungen:

stack	**= list**
top	**= first**
push	**= append**
pop	**= rest**

Beliebteste Implementierung: Sequentielle Darstellung im Array



Einige Anwendungen von Stacks:

- **Auswerten geklammerter arithmetischer Ausdrücke**
- **Verwaltung geschachtelter Prozeduraufrufe**
- **Problemlösen durch Problemreduktion**

18

Schlange (Queue)

Spezielle Liste, deren Elemente nur an einem Ende ("vorne") entnommen und am anderen Ende ("hinten") angehängt werden

1. Erweiterung des ADT list1 um die Operation rappend (= rear append)

rappend	: list x elem	=> list

2. Umbenennungen

queue	= list
front	= first
enqueue	= rappend
dequeue	= rest

"Prioritäts-Warteschlangen" erfordern Sortieren nach Priorität

(später behandelt)

19

Signatur des ADT Schlange

ADT queue1			
TYPEN	**queue, elem**		
OPERATIONEN	**empty**	**:**	**=> queue**
	front	**: queue**	**=> elem**
	enqueue	**: queue x elem**	**=> queue**
	dequeue	**: queue**	**=> queue**
	isempty	**: queue**	**=> bool**

Hauptanwendung von Schlangen:

Warteschlangen bei der Bearbeitung von Aufträgen, z.B.

- Prozesse

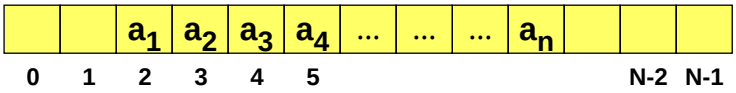
- Nachrichten

- Druckaufträge

20

Implementierung von Schlangen

- Wie bei Kellern ist jede Form der Listen-implementierung grundsätzlich geeignet.
- Beliebt: sequentielle Implementierung im Array



Trick:
Um ein Herauswandern aus dem Array (und aufwendige Speicherverwaltung) zu vermeiden, kann der Array zyklisch adressiert werden:

```
ZyklischerIndex = Index % N
```

21

Mengen

Datentyp entspricht den mathematischen Konzepten für Mengen und Mengenoperationen

ADT	set1		
TYPEN	set, elem, list		
OPERATIONEN	empty	:	=> set
	insert	: set x elem	=> set
	union	: set x set	=> set
	intersection	: set x set	=> set
	difference	: set x set	=> set
	enumerate	: set	=> list

22

Implementierung von Mengen als Bitvektor

```
type set1 = array [1..N] of bool
```

Komplexität der Operationen:

insert	O (1)
empty, enumerate	O (N)
union, intersection, difference	O (N)

Aufwand ist proportional zur Maximalgröße der Mengen N, nicht zur Größe der verarbeiteten Menge!

Entspricht Implementation des Datentyps Set in Pascal und Modula.

23

Einschub: Definition der O-Notation für Komplexitätsaussagen

"f = O (g)" oder "f ist O (g)" oder "f ist von der Ordnung g"

Definition:

Seien f und g Funktionen natürlicher Zahlen. f ist höchstens von der Ordnung g, geschrieben

$$f = O(g)$$

wenn es positive Konstanten c und n_0 gibt, so daß gilt:

$$f(n) \leq c \cdot g(n) \quad \text{für } n > n_0$$

Beispiel:

Zahl der Operationen beim Bearbeiten einer Liste mit n Elementen sei f (n) = 3n • (n+1) + 42

=> f ist O (n^2) denn $3n \cdot (n+1) + 42 \leq 4n^2$ für alle $n > n_0 = 10$

24

Typische Komplexitätsaussagen mit der O-Notation

	Sprechweise	Typische Algorithmen
O (1)	konstant	Zugriff auf Array-Elemente
O (log n)	logarithmisch	Suchen auf einer Menge
O (n)	linear	Bearbeiten jedes Elementes einer Menge
O (n log n)		gute Sortierverfahren, z.B. Heapsort
O ($n \log^2 n$)		
...		
O (n^2)	quadratisch	primitive Sortierverfahren
O (n^k), $k \leq 2$	polynomiell	
...		
O (2^n)	exponentiell	Suchverfahren, z.B. kürzester Weg
...		

25

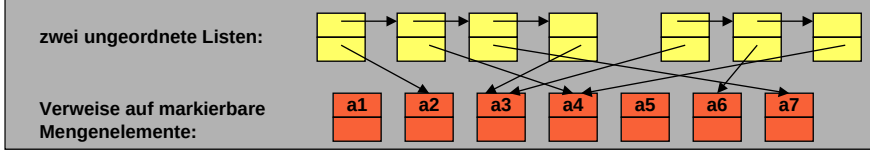
Implementierung von Mengen als ungeordnete Listen

Komplexität der Operationen:

insert	**O (n)**	das neue Element muß gegen alle vorhandenen n Elemente auf Duplizierung geprüft werden
union, intersection, difference	**O (m•n)**	jedes der n Elemente der ersten Menge wird gegen jedes der m Elemente der zweiten Menge geprüft

Unter bestimmten Bedingungen lassen sich union, intersection und difference in ungeordneten Listen auch mit O (m+n) implementieren - finden Sie selbst heraus:

1. **Listenelemente verweisen auf Mengenelemente**
2. **Mengenelemente werden auf Identität (und nicht nur Gleichheit) geprüft**
3. **Mengenelemente können markiert werden**



26

Implementierung von Mengen als geordnete Listen

Komplexität der Operationen:

insert	**O (n)**	Richtige Stelle in der Liste suchen, neues Element einfügen
union, intersection, difference	**O (m+n)**	"Paralleler" Durchlauf durch beide Listen

Algorithmus intersection:

(1) Aktuelle Stellen sind Anfang 1. Liste und Anfang 2. Liste

(2) Merke aktuelles Element von 1. Liste

(3) Suche von aktueller Stelle in 2. Liste aus, bis Element gefunden ist, das gleich oder größer dem gemerkten ist

(4) Falls gleich, gehört Element zum Durchschnitt

(5) Vertausche die Rollen von 1. und 2. Liste

(6) Wiederhole ab Schritt (2), bis eine der Listen erschöpft ist

27

Der Mengentyp Dictionary

Mengentyp ohne UNION, INTERSECTION und DIFFERENCE

Signatur:

```
ADT          set2
TYPES        set, elem, list
OPERATIONS
        empty    :                => set
        insert   : set x elem     => set
        delete   : set x elem     => set
        member:  set x elem       => bool
        isempty: set              => bool
```

Häufig gebrauchter Datentyp zum Umgang mit großen Datenbeständen.

(Exemplarische Anwendung beim Wörterbuch = Dictionary)

28

Implementierung von Mengeneinfügung mit Hashing

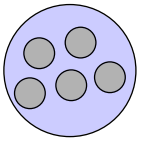
Grundidee von Hashverfahren:

Aus dem *Schlüssel* eines einzufügenden Elementes seine *Speicheradresse* berechnen

D	**Schlüssel-Wertebereich der Mengenelemente**
B_0 ... B_{m-1}	**Behälter (buckets)**
h: D Ý [0 ... m-1]	**Hashfunktion (Schlüsseltransformation)**

Erstrebenswerte Eigenschaften einer Hashfunktion h:

- h ist surjektiv, d.h. erfaßt alle Behälter
- h verteilt gleichmäßig über alle Behälter
- h kann effektiv berechnet werden



D

hashing = "zerhacken, einen Mischmasch machen"

29

Beispiel einer Hashfunktion

<u>Beispiel</u>:

Einfügen von Monatsnamen in 12 Behälter

c_1 ... c_k Monatsname als Zeichenkette

N(c_k) Binärdarstellung eines Zeichens

Hashfunktion h mit m=12 bildet Zeichenketten in Indizes 0 ... 11 ab:

$$h(c_1 \ldots c_k) = \sum_{i=1}^{k} N(c_i) \bmod m$$

Resultierende Verteilung der Monatsnamen (bei fiktivem Code):

0	**November**	**6**	**<u>Mai, September</u>**
1	**<u>April, Dezember</u>**	**7**	**Juni**
2	**März**	**8**	**Januar**
3		**9**	**Juli**
4	**August**	**10**	
5	**Oktober**	**11**	**Februar**

Kollision =

verschiedene Elemente werden in denselben Behälter abgebildet

30

Kollisionen beim Hashing

Häufig ist |D| > m, d.h. alle Schlüssel können nicht in verschiedene Behälter abgebildet werden. Wenn mehrere Schlüssel in einen Behälter abgebildet werden, spricht man von *Kollision*.

<u>Geschlossenes Hashing</u>: Behälter kann nur konstante Zahl von Schlüsseln aufnehmen

<u>Offenes Hashing</u>: Behälter kann beliebig viele Schlüssel aufnehmen

Wahrscheinlichkeit einer Kollision für n Schlüssel, m Behälter:

- **ideale Hashfunktion (gleichmäßige Verteilung)**
- **n < m (zur Vereinfachung der Rechnung)**

$$P_{\text{Kollision}} = 1 - P_{\text{keine Kollision}}$$

$$= 1 - \left(\frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \ldots \cdot \frac{m-n+1}{m}\right) = 1 - \frac{m!}{m^n \cdot (m-n)!}$$

"Geburtstagsparadox":

Bei 23 Personen ist die Wahrscheinlichkeit von 2 Leuten mit gleichem Geburtstag > 50%!

31

Sortieren

<u>Definitionen</u>:

Sortieren ist die (lineare) Anordnung von Datenobjekten in einer Liste entsprechend einer *Ordnungsrelation* ≤.

Der Teil des Datums, der von der Ordnungsrelation ≤ berücksichtigt wird, heißt *Sortierschlüssel*.

Die Ordnungsrelation heißt *Sortierkriterium*.

L = [a_1, a_2, ... , a_n]

Aufsteigende Anordnung: $\bigwedge_{i=1}^{n-1} a_{i+1} \geq a_i$

Absteigende Anordnung: $\bigwedge_{i=1}^{n-1} a_{i+1} \leq a_i$

Vielen Dank an Leonie Dreschler-Fischer für gute Folienvorlagen zum Thema Sortieren!

32

Eigenschaften von Sortierverfahren

Ein Sortierverfahren ...

- **ist *stabil*, wenn zwei Einträge mit gleichrangiger Ordnungsrelation ihre Anordnung beibehalten**
- **hat *natürliches Verhalten*, wenn es bei vorsortierten Daten nicht langsamer ist als bei unsortierten Daten**
- **sortiert *in situ (in place)*, wenn es zum Sortieren nur konstanten zusätzlichen Platz braucht**
- **ist von der *Ordnung* O(g(n)), wenn für die Zahl der Operationen f(n) beim Sortieren einer Liste mit n Elementen gilt:** $c, n_0 : \forall_{n>n0} f(n) \leq c\ g(n)$

Typische Komplexitätseigenschaften von Sortierverfahren:

O(n log n)	gute Verfahren
$O(n^2)$	einfache Verfahren

33

Direkte Sortierverfahren

Man kann eine Liste sortieren durch ...

<u>Vertauschen</u>:

Vertausche Einträge, die wechselseitig falsch angeordnet sind (Bubblesort, Shakersort)

<u>Einfügen</u>:

Stelle sortierte Teilfolge her und füge verbleibende Elemente an der richtigen Stelle ein (Insertionsort, Shellsort)

<u>Auswahl</u>:

Suche das Element, das an Platz 1 gehört, an Platz 2 usw. (Selectionsort, Heapsort)

In allen Fällen sind die Grundoperationen *Vergleich* zweier Elemente und *Bewegung* eines Elementes erforderlich.

34

Sortieren mit Divide-and-Conquer

Beobachtungen:

Zwei mit O (n2) sortierte Listen lassen sich mit O (n+m) zu einer sortierten Liste zusammenfügen

Die Gesamtzahl der Operationen kann sich durch Zerlegen verringern:

$$c \cdot (m+n)^2 \overset{?}{>} c1 \cdot m^2 + c2 \cdot n^2 + c3 \cdot (m+n)$$

Prinzipieller Divide-and-Conquer-Algorithmus:

Falls die Objektmenge klein genug ist, löse das Problem direkt

Andernfalls

Divide:	Zerlege die Menge in mehrere Teilmengen (wenn möglich gleicher Größe)
Conquer:	Löse das Problem rekursiv für jede Teilmenge
Merge:	Berechne die Gesamtlösung durch Zusammenfügen der Lösungen für die Teilmengen

35

Sortieren durch Verschmelzen

Algorithmus MergeSort (L) :

Falls |L| = 1 return L

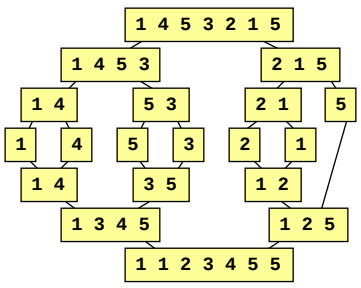
Andernfalls

Divide: $L1 := \left[a_1 \ldots a_{\lfloor n/2 \rfloor}\right]$; $L2 := \left[a_{\lfloor n/2 \rfloor + 1} \ldots a_n\right]$;

Conquer: L1´:= MergeSort (L1); L2´:= MergeSort (L2)

Merge: return Merge (L1´, L2´)

Beispiel:



36

Komplexitätsanalyse von MergeSort

Zahl der Operationen f(n) für Liste mit n Elementen:

f(n) = O (1) **für n=1**

f(n) = O (1) + 2•f(n/2) + O (n) **für n>1**

Divide **Conquer** **Merge**

Rekursionsgleichung der Form

f(1) = c0

f(n) = 2•f(n/2) + c1•n

Lösungsansatz f(n) = a•n•log(b•n) liefert

f(n) = c1•n•log n + c0•n

=> f(n) = O (n•log n) **MergeSort hat weniger als quadratische Ordnung!**

37

Sortieren mit linearer Ordnung

Voraussetzungen:

- **Die Menge aller Schlüssel ist hinreichend klein**
- **Es muß nicht in situ sortiert werden**

Sortieren mit BucketSort:

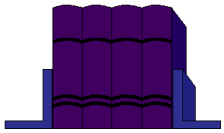
- Lege für jeden Schlüssel eine Liste an
- Mache die Listen zu Elementen eines Arrays, der mit dem Schlüssel indiziert werden kann
- Gehe die zu sortierenden Elemente einzeln durch und hänge jedes an die zu seinem Schlüssel gehörige Liste an
- Verbinde die Listen zu einer Ergebnisliste

Das Verfahren ist linear in n, da der Aufwand für das Durchlaufen der Ausgangsliste und des Schlüssel-Arrays linear von der Zahl der Listenelemente abhängt.

38

Parade der Sortierverfahren

BubbleSort	**O (n^2)**
ShakerSort	**O (n^2)**
SelectionSort	**O (n^2)**
InsertionSort	**O (n^2)**
ShellSort	**O ($n^{1.2}$)**
MergeSort	**O (n log n)**
QuickSort	**O (n log n)**
HeapSort	**O (n log n)**
BucketSort	**O (n)**
RadixSort	**O (n)**



Siehe dazu die reichhaltig vorhandene Literatur, z.B.

N. Wirth, Algorithmen und Datenstrukturen, Teubner

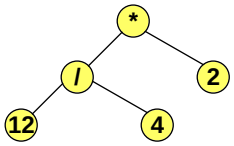
R.H. Güting, Datenstrukturen und Algorithmen, Teubner

Die Angaben beziehen sich auf die Durchschnittskomplexität. Bei einigen Verfahren ist die Worst-Case-Komplexität schlechter, z.B. Quicksort hat dann O (n^2)

39

Binäre Bäume

- **Baumstruktur aus Knoten und Kanten**
- **jeder Knoten hat maximal 2 Kinder**
- **Knoten ohne Vater heißt "Wurzel"**
- **Knoten ohne Kind heißt "Blatt"**



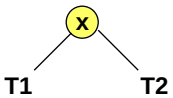
Beispiel: Repräsentation eines algebraischen Ausdrucks

Rekursive Definition:

- **Der leere Baum ist ein Binärbaum**
- **Wenn x ein Knoten ist und T1, T2 Binärbäume sind, dann ist auch das Tripel (T1, x, T2) ein Binärbaum T**

Ein Tripel (T1, x, T2) wird graphisch dargestellt als:

(ein leerer Baum wird meist nicht dargestellt)



40

Signatur des Datentyps Binärbaum

ADT	**tree1**		
TYPES	**tree, elem**		
OPERATIONS	**empty**	**:**	**=> tree**
	maketree	**: tree x elem x tree**	**=> tree**
	root	**: tree**	**=> elem**
	left, right	**: tree**	**=> tree**
	isempty	**: tree**	**=> bool**

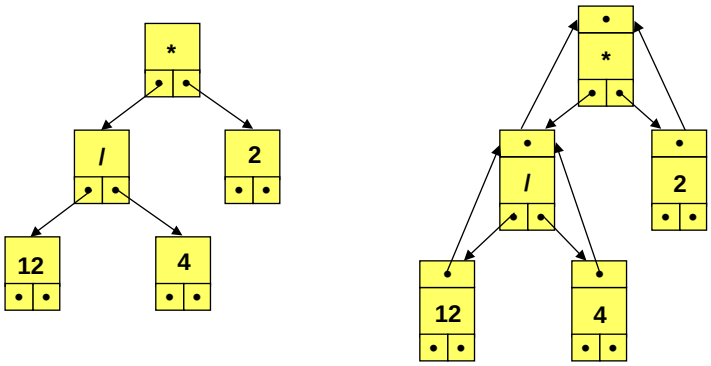
Die Operation maketree entspricht der rekursiven Definition eines Binärbaums:

Ein Binärbaum wird durch Zusammenfügen von Binärbäumen aufgebaut. Den Anfang machen leere Binärbäume.

41

Implementierungen von Binärbäumen mit Zeigern

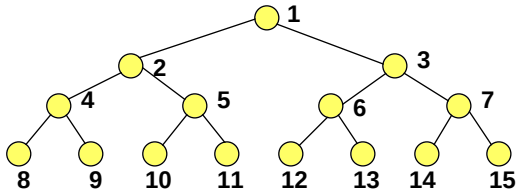
Ähnlich der Implementierung von einfach oder doppelt verketteten Listen:



42

Implementierung von Binärbäumen durch Array-Einbettung

Abbildung der Knoten eines vollständigen Binärbaums auf einen Array
- von oben nach unten
- von links nach rechts:



Für jeden Knoten p gilt:

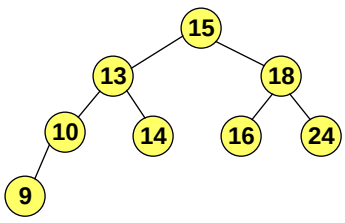
Index(p) = i => Index (p.left) = 2•i
Index (p.right) = 2•i + 1

43

Binäre Suchbäume

Anordnung einer geordneten Menge als Knoten eines Binärbaums zur Unterstützung einer binären Suche

Beispiel:
M = { 9 10 13 14 15 16 18 24 }



Beispiel eines binären Suchbaums

Definition:

B ist binärer Suchbaum, falls gilt:

B ist leer oder

- der linke und rechte Unterbaum von B sind binäre Suchbäume,

- ist w die Ordnungszahl der Wurzel, so sind alle Elemente im *linken* Unterbaum *kleiner* als w, alle Elemente im *rechten* Unterbaum *größer* als w.

44

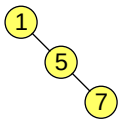
Aufbau von binären Suchbäumen

Der Aufbau eines binären Suchbaums erfolgt durch wiederholtes Einfügen in einen anfangs leeren Baum.

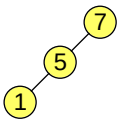
Die Einordnungsreihenfolge einer geordneten Menge bestimmt die Form des binären Suchbaums.

Beispiele:

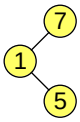
1 5 7



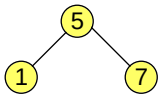
7 5 1



7 1 5



5 7 1 oder 5 1 7



Wo landet danach eine 4?

45

Implementierung von binären Suchbäumen in Java

```
class Knoten {
   private int wert;
   private Knoten links, rechts;
   Knoten(int i) {wert = i; links = rechts = null;}
   void SetzeWert(int i) {wert = i;}
   int GibWert() {return wert;}
   void SetzeLinks(Knoten k) {links = k;}
   Knoten GibLinks() {return links;}
   void SetzeRechts(Knoten k) {rechts = k;}
   Knoten GibRechts() {return rechts;}
};
```

```
class BST {
   private Knoten wurzel
   BST() {wurzel = null;}
   void FügeEin(int i) {wurzel = FügeEin(wurzel, i);}
   private Knoten FügeEin(Knoten aktuell, int ein) {
      if (aktuell == null) aktuell = new Knoten(ein);
      else {
         if (ein < aktuell.Gibwert())
            aktuell.SetzeLinks(FügeEin(aktuell.GibLinks(), ein));
         if (ein > aktuell.Gibwert())
            aktuell.SetzeRechts(FügeEin(aktuell.GibRechts(), ein));
      }
      return aktuell;
   }
};
```

46

Algorithmus für binäre Suche in einem binären Suchbaum

Ist Element mit Ordnungszahl k im binären Suchbaum B?

Einfacher Algorithmus:

- **B ist leer: Element kann nicht im Baum sein**
- **B ist nicht leer:**
  - **B.wert = k: Element ist gefunden**
  - **B.wert < k: Suche im rechten Unterbaum von k**
  - **B.wert > k: Suche im linken Unterbaum von k**

47

Implementierung der binären Suche

Erweiterung der Klassendefinition von BST:

```
class BST {
   ...
   boolean Suche(int i) { return Suche(wurzel, i); }
   private boolean Suche(Knoten aktuell, int i) {
      boolean gefunden = false;
      if (aktuell != null) {
         gefunden = (aktuell.GibWert() == i);
         if (aktuell.GibWert() < i)
            gefunden = Suche(aktuell.GibRechts(), i);
         if (aktuell.GibWert() > i)
            gefunden = Suche(aktuell.GibLinks(), i);
      }
      return gefunden;
   }
   ...
};
```

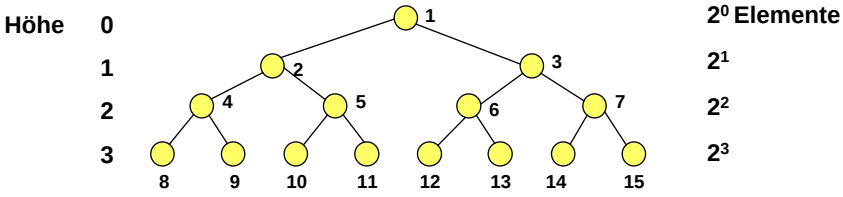
Rekursive Aufrufe

48

Effizienz der Suche im binären Suchbaum

- Der Suchaufwand wird an der Zahl V der durchzuführenden Vergleiche gemessen.
- Die Zahl der Vergleiche bei einer erfoglosen Suche ist höchstens gleich der Höhe H des Baums.
- Die Höhe eines Binärbaums mit N Elementen ist

$$\lfloor \log_2 (N+1) \rfloor \leq H \leq N - 1$$



Bei einem balancierten Binärbaum ist der Suchaufwand O ($\log_2$ N)

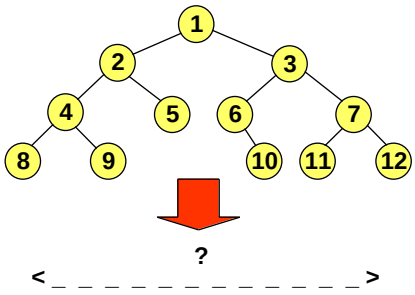
49

Durchlauf von Binärbäumen

Für viele Anwendungen müssen alle Knoten eines Baumes der Reihe nach bearbeitet werden. Dazu muß ein Ordnungsprinzip gewählt werden, das den Baum auf eine Liste abbildet.

Wir betrachten:

- Breitendurchlauf
- Tiefendurchlauf
- Preorder
- Inorder
- Postorder



50

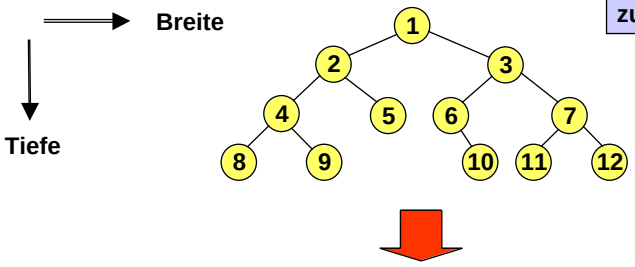
Breitendurchlauf

Durchlauf in der Reihenfolge:

Breite vor Tiefe, links vor rechts

(auch anwendbar für allgemeine Bäume)

"Schichtenweises" Abarbeiten eines Baumes, z.B. bei der Suche nach Zielknoten mit minimalem Abstand zur Wurzel



<1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12>

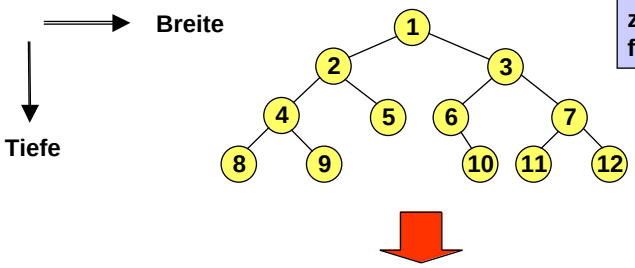
Tiefendurchlauf

Durchlauf in der Reihenfolge:

Tiefe vor Breite, links vor rechts

(auch anwendbar für allgemeine Bäume)

Abarbeiten eines Baumes in Richtung Blätter mit Backtracking, z.B. bei der Suche nach Planschritten, die zu einem Ziel führen sollen.



<1, 2, 4, 8, 9, 5, 3, 6, 10, 7, 11, 12>

Spezielle Ordnungsprinzipien für Binärbäume

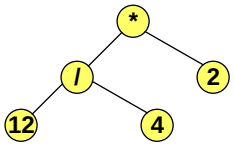
Rekursive Definitionen:

inorder(left, x, right) = inorder(left) ° [x] ° inorder(right)

preorder(left, x, right) = [x] ° preorder(left) ° preorder(right)

postorder(left, x, right) = postorder(left) ° postorder(right) ° [x]

° = Konkatenationsoperator
[ ] = Listenkonstruktor



inorder → [12, /, 4, *, 2]

preorder → [*, /, 12, 4, 2]

postorder → [12, 4, /, 2, *]

z.B. zur Ausgabe von Parse-Bäumen in Infix-, Präfix- oder Postfix-Notation

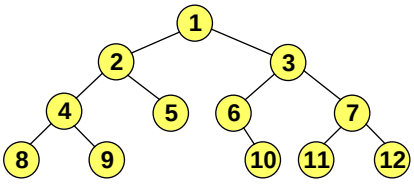
53

Implementierung einer Baumausgabe mit inorder

```
void InOrder() {
   InOrder(wurzel);
}
private void InOrder(Knoten aktuell)
   if (aktuell != null) {
      Inorder(aktuell.GibLinks());

System.out.println(aktuell.GibWert());
      InOrder(aktuell.GibRechts());
   }
}
```

Das Programm ist *rekursiv* entsprechend der Definition von inorder.



In welcher Reihenfolge werden die Knoten des Binärbaums ausgegeben?

54

Prioritätswarteschlangen

Prioritätswarteschlangen (priority queues) sind Warteschlangen, deren Elemente (partiell) gemäß einer Priorität eingeordnet werden.

Besondere Operationen:

- **Entnahme des Elementes mit der höchsten Priorität (üblich: mit niedrigstem Zahlenwert)**
- **Einfügen eines neuen Elementes mit beliebiger Priorität**

ADT	**pqueue1**		
TYPES	**pqueue**		
OPERATIONS	**empty**	**:**	**=> pqueue**
	isempty	**: pqueue**	**=> bool**
	insert	**: pqueue x elem**	**=> pqueue**
	deletemin	**: pqueue**	**=> pqueue x elem**

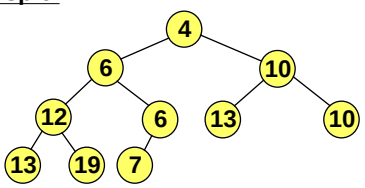
55

Implementierung von Prioritätswarteschlangen: Der Heap

Implementierung mit einem *Heap* (Haufen = partiell geordneter Baum)

Definition:
Ein Heap ist ein knotenmarkierter Binärbaum, in dem für jeden Teilbaum gilt:
Die Wurzel eines Teilbaums ist das Minimum des Teilbaums.

Beispiel:



Merke:

Die Folge der Knotenmarkierungen auf einem Pfad ist stets monoton steigend.

56

Insert-Operation auf einem Heap

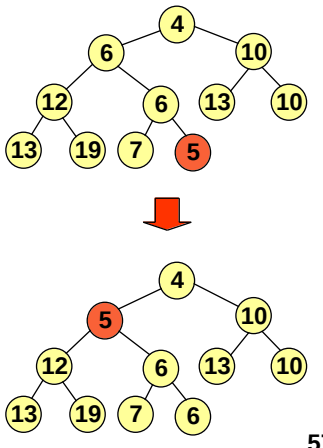
Hier nur Betrachtung von links-vollständigen Bäumen:

- Alle Ebenen bis auf die letzte sind vollständig besetzt
- Auf der letzten Ebene sitzen die Elemente so weit links wie möglich

Einfügen eines neuen Elementes bei Bewahren der Heap-Eigenschaft:

Algorithmus insert (h, e)

- Erzeuge neuen Knoten q mit Eintrag e, füge q in unterster Ebene an (falls diese voll ist, beginne neue Ebene)
- Sei p Vater von q.
- Solange p existiert und der Wert von q < p ist, vertausche die Einträge von p und q; setze q auf p und p auf den Vater von p.



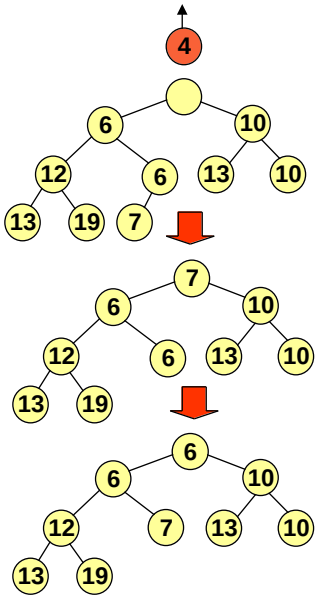
57

Deletemin-Operation auf einem Heap

Entfernen der Wurzel eines Heap bei Bewahren der Heap-Eigenschaft:

Algorithmus deletemin (h)

- Gib Eintrag der Wurzel aus.
- Ersetze Eintrag der Wurzel durch Eintrag des letzten Elementes des Baumes (unterste Ebene rechts). Lösche diesen Knoten.
- Sei p Wurzel und q, r ihre Kinder.
- Solange q oder r existieren und der Wert von q < p oder der Wert von r < p ist, vertausche den Eintrag in p mit dem *kleineren* Eintrag der Kinder; setze p auf den Knoten, mit dem vertauscht wurde, und mache dessen Söhne zu Söhnen von p.



58

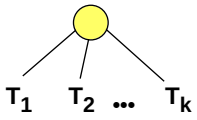
Allgemeine Bäume

Die Anzahl der Kinder eines Knotens ist beliebig.

Rekursive Definition:

- **Ein einzelner Knoten ist ein Baum.**
- **Wenn x ein Knoten ist und T_1, ..., T_k Bäume sind, dann ist auch das Tupel (x, T_1, ..., T_k) ein Baum.**

Ein Tupel (x, T_1, ..., T_k) wird graphisch dargestellt als

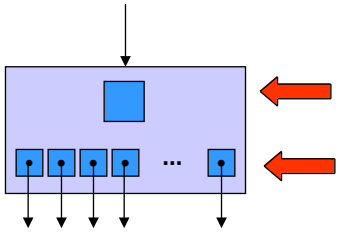


maximales k
= Grad des Baumes

59

Implementierung von allgemeinen Bäumen

Grundstruktur eines Knotens:



Wert des Knotens

Liste von Zeigern auf Kinder

- **Implementierung des Knotens als Record oder Class**
- **Verwendung einer der bekannten Listenimplementierungen für die Zeigerliste**

60

Graphen

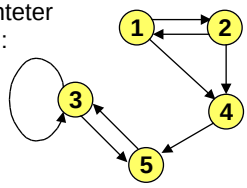
Ein Graph stellt eine Menge von Objekten mit einer Relation auf diesen Objekten dar.

Beispiel:
Objekte = Zustände eines Automaten
Relation = Beziehung zwischen Zustand und Nachfolgezustand

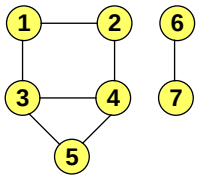
Darstellung

- **eines Objektes durch einen Knoten**
- **einer Beziehung durch eine gerichtete Kante**

Gerichteter Graph:



Ungerichteter Graph: (bei symmetrischen Beziehungen)



61

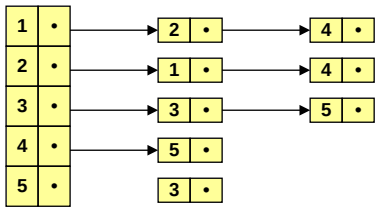
Grundlegende Operationen auf Graphen

ADT	**graph1**	
TYPES	**graph, list, node, edge**	
OPERATIONS		
empty	**:**	**=> graph**
insertNode	**: graph x node**	**=> graph**
insertEdge	**: graph x node x node**	**=> graph**
deleteNode	**: graph x node**	**=> graph**
deleteEdge	**: graph x node x node**	**=> graph**
successors	**: graph x node**	**=> list**
predecessors	**: graph x node**	**=> list**
containsNode	**: graph x node**	**=> bool**
containsEdge	**: graph x node x node**	**=> bool**
isempty	**: graph**	**=> bool**

62

Implementierung von gerichteten Graphen durch Adjazenzlisten

- **Jeder Knoten verfügt über eine Liste seiner Nachfolgerknoten**
- **Alle Knoten sind über einen Array direkt zugreifbar**



Geringer Platzbedarf, Zugriff auf alle Nachfolgerknoten in Linearzeit.

Bei Bedarf können auch *inverse Adjazenzlisten* (Listen von Vorgängerknoten) vorgesehen werden.

63

Implementierung von gerichteten Graphen durch eine Adjazenzmatrix

i \ j	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	1	0
3	0	0	1	0	1
4	0	0	0	0	1
5	0	0	1	0	0

A_{ij} = true falls Kante von i nach j

i \ j	1	2	3	4	5
1	∞	6	∞	4	∞
2	5	∞	∞	11	∞
3	∞	∞	3	∞	1
4	∞	∞	∞	∞	9
5	∞	∞	13	∞	∞

Bei vorhandenen Kanten kann Markierung in die Adjazenzmatrix eingetragen werden, z.B. "Kosten" für Zustandsübergang von i nach j. Besondere Markierung für "keine Kante".

64

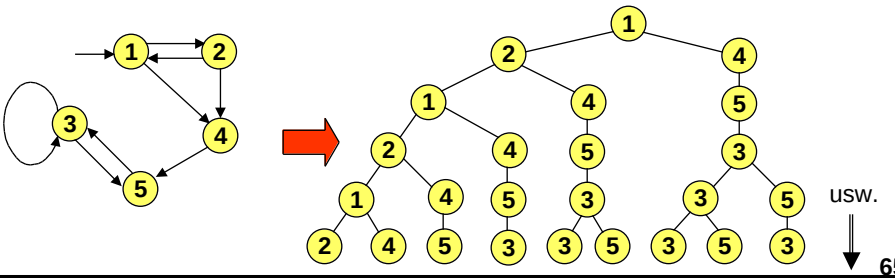
Expansion eines Graphen in einen Baum

Von einer Wurzel aus werden Nachfolger stets als "neue" Knoten erzeugt. Enthält der Graph einen Zyklus, so ist die Expansion unendlich.

Definition

Die Expansion X(r) eines Graphen in einem Knoten r ist ein Baum mit folgenden Eigenschaften:

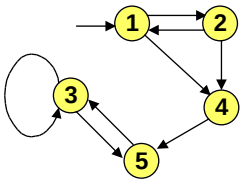
- **Falls r keine Nachfolger hat, besteht X(r) nur aus r.**
- **Falls r_1 ... r_k die Nachfolger von r sind, so ist X(r) der Baum (r, X(r_1), ... , X(r_k))**



65

Durchläufe in Graphen

Da Knoten eines Graphen auf *mehreren Pfaden* und durch *Zyklen* *wiederholt* erreicht werden können, ist eine Buchführung erforderlich, wenn man jeden Knoten einmal bearbeiten (z.B. ausgeben) will.



Prinzip für Durchlauf:

- **Wende Breitendurchlauf oder Tiefendurchlauf auf die *Expansion* des Graphen an**
- **Markiere dabei besuchte Knoten und vermeide Mehrfachbesuche**

Merke 1: **Tiefendurchlauf läßt sich durch Rekursion einfach implementieren**

Merke 2: **Markierungen "SchonBesucht" müssen vor dem nächsten Durchlauf zurückgesetzt werden.**
Alternative: Markierungen mit jedem Durchlauf umgekehren.

66

Dynamisches Erzeugen eines Graphen

Bei vielen Problemen ist es nicht sinnvoll oder möglich, einen Graphen vollständig im Programm zu repräsentieren.

Beispiele:
- **Zustandsübergänge bei einem Schachspiel**
- **Mögliche Planschritte bei einem Planungsproblem**
- **Schrittweise Suche nach einem Beweis**

Vorgehen:

Das Problem definiert einen *impliziten* Graphen, der nur partiell *explizit* gemacht wird. Dabei wird die *Erzeugung* des (expliziten) Graphen meist mit seiner *Verarbeitung* verbunden.

Besonders einfache Algorithmen ergeben sich, wenn man schrittweise die *Expansion* des Graphen berechnet. Dabei ist es meist sinnvoll, wiederholtes Besuchen von Knoten durch eine zusätzliche Buchführung zu vermeiden.

67

Suche nach dem kostengünstigsten Pfad durch einen Graphen

Mit dem A*-Algorithmus [Hart et al. 68] expandiert (entwickelt) man einen impliziten Graphen in der Weise, daß ein *kostenoptimaler* Pfad von einem Startknoten zu einem Zielknoten gefunden wird.

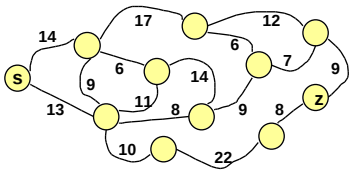
Kostenfunktion c bildet Kanten E in nichtnegative reelle Zahlen ab.

c: E => $\Re_0^+$

Beispiel:

Suche nach dem kürzesten Weg von s nach z

- in einem Straßennetz
- in einem Netz von Versorgungsleitungen



Äquivalent: "Kürzeste Wege in Distanzgraphen suchen"

68

Kostenfunktionen

Steuerung der Suche durch Kostenfunktionen:

k(u, v)	**Kosten für günstigsten Pfad von u nach v**

Daraus zu berechnen für gegebene Start- und Zielknoten:

g*(u) = min k(s, u)	**Minimalkosten von s nach u**
h*(u) = min k(u, z_i)	**Minimalkosten von u bis Ziel z_i**
f*(u) = g*(u) + h*(u)	**Minimalkosten für Lösungsweg über u**

Zur Beschleunigung der Suche nützlich:

f(u) = g(u) + h(u)	**geschätzte Minimalkosten für Lösungsweg über u**
g(u)	**schätzt g*(u)**
h(u)	**schätzt h*(u)**

Bewertungsfunktion f(u) erlaubt den Vergleich von Knoten hinsichtlich der erwarteten Kosten für zugehörige Lösungspfade.

69

Prinzip des A*-Algorithmus

Grundidee:

- **Expandiere den Graphen schrittweise**
- **Untersuche zuerst den Nachfolger u mit der günstigsten Bewertung f(u)**
- **Bei Duplizierungen verfolge nur den günstigeren Weg**

(1) NeuList =[s] AltList = [ ]

(2) Falls NeuList leer ist, terminiere mit Mißerfolg. Nimm günstigsten Knoten u aus NeuList. Falls u Ziel ist, gib Pfad von s zu u aus und terminiere.

(3) Bringe u von NeuList nach AltList. Erzeuge Nachfolger von u und bewerte sie mit f.

(4) Falls ein Nachfolger u´gleich einem Knoten u´´ in NeuList ist, wähle den günstigeren von beiden und lösche den anderen.

(5) Falls ein Nachfolger u´ gleich einem Knoten u´´in AltList ist, wähle den günstigeren von beiden und lösche den anderen.

(6) Füge die (verbleibenden) Nachfolger von u in NeuList ein und setze bei (2) fort.

70

Anmerkungen zum A*-Algorithmus

Das gezeigte Prinzip ist nicht der vollständige Algorithmus!

Es fehlen u.a.

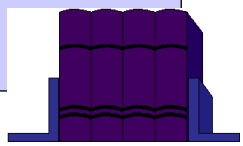
- **Buchführung über den optimalen Pfad vom Start zum aktuellen Knoten**
- **Details der bei Duplizierungen erforderlichen Korrekturen**

Literatur:

P.E. Hart, N.J. Nilsson, B. Raphael: "A Formal Basis for the Heuristic Determination of Minimum-Cost Paths", IEEE Trans. SSC, Vol.4, 1968

N.J. Nilsson: Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, 1971

E. Rich: Artificial Intelligence, McGraw-Hill, 1983



Optimalität des A*-Algorithmus

Optimalität des A*-Algorithmus hängt von der Kostenschätzung h ab:
Überschätzung von Kosten kann günstigsten Pfad blockieren!

Satz:

Der A*-Algorithmus terminiert mit dem günstigsten Pfad, wenn die Bewertungsfunktion f(u) = g(u) + h(u) verwendet wird und

- **g(u) = tatsächliche Kosten von s bis u**
- **$h(u) \leq h^*(u)$**

für alle u gilt. Schätzfunktionen $h \leq h^*$ heißen zulässig.

Folgerungen:

- **$h(u) \equiv 0$ für alle u ist stets eine zulässige Kostenschätzfunktion.**
- **c(u, v) = konstant, $h(u) \equiv 0$ ergeben eine Breitensuche. Breitensuche findet den günstigsten Pfad!**
- **Gilt für zwei zulässige Kostenschätzfunktionen $h1(u) \leq h2(u)$ für alle u, so ist h2 *informierter* als h1 und findet den günstigsten Pfad schneller oder genauso schnell wie h1.**

Komplexität des A*-Algorithmus

n = Zahl der Knoten, m = Zahl der Kanten

Aufwand setzt sich zusammen aus:

Zieltest	O (n)•O (1) = O (n)
Expansion von Knoten (Berechnung von Nachfolgern)	O (m)
Kostenbewertung von Knoten	O (n)•O (1) = O (n)
Vergleiche mit Knoten auf NeuList und AltList	O (n)•O (log n)•O (1) = O (n)
Einfügen in NeuList (Priority Queue)	O (n)•O (log n) = O (n)
Entnahme aus NeuList	O (n)•O (1) = O(n)

Im allgemeinen ist $m \sim n^2$, deshalb gilt:

Komplexität des A*-Algorithmus ist $O(n^2)$

73

Suche nach dem kürzesten Pfad

<u>Optimalitätsprinzip</u>: Ist $p = (u_0, u_1, \ldots, u_k)$ kürzester Pfad von Knoten u_0 nach Knoten u_k, so ist $p' = (u_i, \ldots, u_j)$, $0 \le i < j \le k$, ein kürzester Pfad von u_i nach u_j.

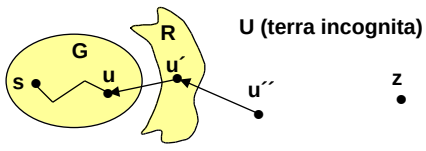
Um einen kürzesten Pfad zu finden, kann man also von kürzesten Teilpfaden ausgehen.

<u>Dijkstra´s Idee (1959)</u>: Vom Startknoten aus "äquidistante Welle" aussenden, die sukzessiv weitere Knoten erfaßt, bis Zielknoten erreicht ist. Dabei werden 3 Knotenmengen unterschieden:

- für "gewählte Knoten" G ist kürzester Weg vom Startknoten s bekannt

- für "Randknoten" R ist ein Weg von s bekannt

- für "unerreichte Knoten" U kennt man noch keinen Weg



74

Dijkstra´s Algorithmus zur Suche nach dem kürzesten Pfad

1. Initialisierung:

Pfadlänge aller Knoten außer s ist ∞, G = {s}, R = {Nachfolger von s}

2. Berechne Wege ab s:

- Falls R leer ist, ist Zielknoten nicht von s erreichbar, brich ab.
- Entferne nächstgelegenen Randknoten u aus R und füge ihn in G ein.
- Falls u Zielknoten ist, gib Vorgängerliste von u bis s aus und brich ab.
- Füge alle Nachfolger von u in R ein, die nicht in G sind.
- Vermerke Pfadlänge und Vorgänger für kürzesten Pfad zu allen Nachfolgern von u in R.
- Wiederhole 2.

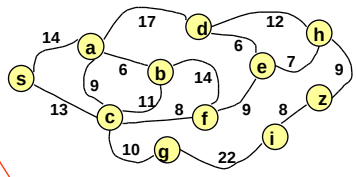
75

Beispiel für Suche nach dem kürzesten Pfad

Schrittfolge zur Bestimmung des kürzesten Pfades von s nach z mit dem Dijkstra-Algorithmus

Notation: (Name, Pfadlänge, Vorgänger)

gewählt	Randknoten
(s 0 s)	(c 13 s) (a 14 s)
(c 13 s)	(a 14 s) (f 21 c) (g 23 c) (b 24 c)
(a 14 s)	(b 20 a) (f 21 c) (g 23 c) (d 31 a)
(b 20 a)	(f 21 c) (g 23 c) (d 31 a)
(f 21 c)	(g 23 c) (e 30 f) (d 31 a)
(g 23 c)	(e 30 f) (d 31 a) (i 45 g)
(e 30 f)	(d 31 a) (h 37 e) (i 45 g)
(d 31 a)	(h 37 e) (i 45 g)
(h 37 e)	(i 45 g) (z 46 h)
(i 45 g)	(z 46 h)
(z 46 h)	



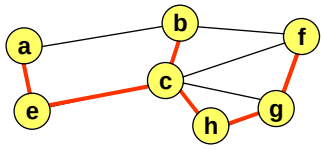
Kürzerer Pfad wird entdeckt, Vorgänger und Pfadlänge werden umgesetzt

76

Minimale spannende Bäume

Beispielsproblem:

Alle Zweigstellen einer Firma sollen mit Kommunikationsleitungen verbunden werden. Wie kommt man mit minimaler Leitungslänge aus?



Formalisierung des Problems:

Gegeben zusammenhängender, ungerichteter, bewerteter Graph G = (V, E) mit c: E => $\Re_0^+$. Gesucht ist zusammenhängender Teilgraph T´= (V, E´) von G, für den die Summe der Kantenbewertungen minimal ist.

T´ heißt minimaler spannender Baum (minimales Gerüst, Minimal Spanning Tree, MST).

77

Grundgerüst für einen MST-Algorithmus

Teilgraph T´, Kanten E´, unentschiedene Kanten U

1. T´= Ø, E´= Ø, U = E, W = Ø
2. Solange noch nicht fertig,
   a) wähle geeignete Kante e aus U, füge e in E´ein und entferne e aus U
   b) wähle geeignete Kante e aus U und entferne e aus U

Der Algorithmus ist "gierig" (greedy):

Entscheidungen zur Lösungsfindung werden nicht revidiert.

In 2a): endgültige Entscheidung *für* die Aufnahme von e in den MST

In 2b): endgültige Entscheidung *gegen* die Aufnahme von e in den MST

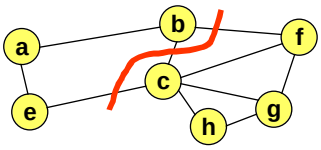
Gierige Algorithmen haben von Natur aus lineare Komplexität und sind meist sehr effizient.

78

Auswahl von Kanten für den MST

Viele Verfahren zur Berechnung eines MST basieren auf *Schnitten* im Graphen G = (V, E).

Ein Schnitt ist eine Zerlegung eines Graphen durch Partitionierung der Knoten V in V´und V´´= V \ V´. Eine Kante (u v) *kreuzt* den Schnitt, wenn u aus V´und v aus V´´ ist.



Beispiel eines Schnittes:

V´= {a b e} V´´= {c f g h}

(b c), (b f) und (e c) kreuzen den Schnitt

Beispiele von Regeln für Kantenauswahl in einem MST-Algorithmus:

Regel 1: Wähle Schnitt, der keine gewählte Kante kreuzt. Wähle *kürzeste* unter den unentschiedenen Kanten, die den Schnitt kreuzen

Regel 2: Wähle einfachen Zyklus, der keine verworfene Kante enthält. Verwirf *längste* unter den unentschiedenen Kanten im Zyklus.

79

Algorithmus zur Bestimmung eines MST nach Jarnik, Prim, Dijkstra

G = (E, V) = zusammenhängender, ungerichteter, bewerteter Graph
T´= Teilgraph, E´= gewählte Kanten, U = unentschiedene Kanten

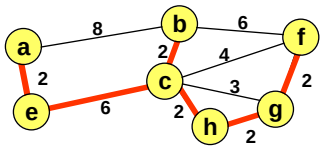
1. Initialisierung:

Wähle beliebigen Knoten s, setze T´= {s}, E´= Ø, U = E

2. Führe Schritt (|V| - 1)-mal aus:

Wähle Kante e aus U: e ist Kante mit minimaler Länge, für die genau ein Endknoten zum gewählten Baum T´gehört.

Entferne e aus U und füge e in E´ein.



Auswahlschritte:

(a e)
(e c)
(c b)
(c h)
(h g)
(g f)

Effiziente Realisierung von U durch Prioritätsschlange

80

Problem des Handelsreisenden

Beim Problem des Handelsreisenden (Travelling-Salesman Problem, TSP) geht es darum, die kürzeste Rundtour von einem Startort aus durch vorgegebene Orte und wieder zurück zu finden.

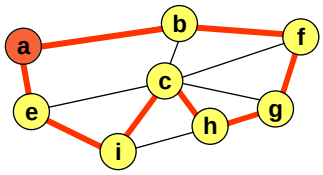
Beispiele:

- Versorgung von Verbrauchern aus einer Zentrale

- Müllabfuhr

- Anfahren von Lötpunkten durch Industrieroboter

- Eintreiben von Schutzgebühren in den Kneipen von St. Pauli



Problem:

Bei n zu besuchenden Orten gibt es maximal n! Touren.

(Für das Beispiel gäbe es 7! = 5040 Touren, wären alle Orte miteinander verbunden)

81

Suboptimaler Algorithmus für das Problem des Handelsreisenden

Das TSP gehört zur Klasse der NP-vollständigen (nichtdeterministisch-polynomial-vollständigen) Probleme, für die keine Algorithmen mit polynomialer Zeitkomplexität bekannt sind.

NP-vollständige Probleme gelten als nicht traktabel, d.h. nicht effizient lösbar.

Lokal optimaler "gieriger" (greedy) Algorithmus für das TSP:

1. Tour ist leer, Auswahlmenge enthält alle Wege.
2. Solange Orte unbesucht und Wege in Auswahlmenge sind:

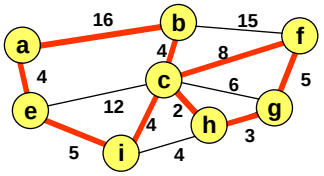
   Wähle als nächstes den kürzesten Weg und entferne ihn aus der Auswahlmenge.

   - Verwerfe den Weg, wenn eine T-Verzweigung entstehen würde
   - Verwerfe den Weg, wenn ein Zyklus entstehen würde

   Andernfalls nimm Weg in Tour auf.
3. Verbinde Endpunkte der Teiltouren mit kürzester Verbindung.

82

Beispiel für TSP-Lösung durch suboptimalen Algorithmus

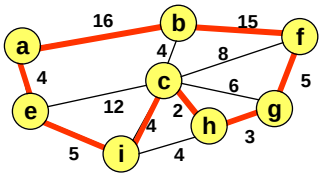


Gewählte Wege

(c h)	2
(h g)	3
(a e)	4
(b c)	4
(g f)	5
(e i)	5
(a b)	16
(i c) (c f)	12

Gesamtlänge = 51

Für NP-vollständige Probleme ist es charakteristisch, daß kleine Änderungen der Problemstellung zu drastischen Änderungen der Lösung führen können.



Bei Wahl von (c i) 4 anstelle (b c) 4 ergibt sich eine andere Tour mit Gesamtlänge = 54

83

Was bedeutet die Komplexität eines Algorithmus in der Praxis?

Wir haben Algorithmen mit verschiedener Komplexität O (g(n)) kennengelernt. Die praktische Verwendbarkeit eines Algorithmus kann bei hoher, insbesondere exponentieller, Komplexität drastisch eingeschränkt sein.

Annahme: Zeitaufwand 1 ms für n=1

g(n)	n=1	n=5	n=10	n=50	n=100	n=500	n=1000
n	1 ms	5 ms	10 ms	50 ms	0,1 s	0,5 s	1 s
n log n	(0 ms)	11 ms	33 ms	0,3 s	0,6 s	4 s	10 s
n^2	(1 ms)	25 ms	0,1 s	3 s	10 s	4 min	17 min
n^3	(1 ms)	0,1 s	1 s	2 min	17 min	35 h	12 T
2^n	(2 ms)	32 ms	1 s	10^7 J	10^{22} J	10^{143} J	10^{293} J

84

Zusammenfassung des 1. Teiles

Wir haben wichtige Bausteine für die Modellierung von Realweltproblemen kennengelernt:

Dynamische Datenstrukturen mit zugehörigen Standardoperationen
(ADTs Listen, Mengen, Bäume, Graphen u.a.)

Dynamische Datentypen als Basis für wichtige problemorientierte Algorithmen
(Sortieren, Heuristische Suche, kürzester Pfad, MST, TSP u.a.)

Grundformen von Algorithmen
(Rekursion, Backtracking, Divide-and-Conquer, gierige Verfahren)

Komplexitätsklassen
(logarithmisch, linear, n log n, quadratisch, polynomial, exponentiell u.a.)

85