Algorithmen und Programmierung III
- Datenstrukturen und Datenabstraktion -

18. Februar 2012

Inhaltsverzeichnis

1 Überblick über den Inhalt der Vorlesung

- Datenabstraktion
  - Entwurfsprinzipien der Softwaretechnik
  - verschiedene Abstraktionsebenen bei der Programmierung bzw. Organisation von Daten
- Entwurf und Analyse von Algorithmen
  - Entwurfsparadigmen (z.B. Divide-and-Conquer, dynamisches Programmieren)
  - Analyse von Zeit- und Speicherbedarf eines Algorithmus
- Datenstrukturen und Algorithmen
  - z.B. „Wörterbuchproblem“
  - Prioritäts-Warteschlange
  - Sortieren und Suchen
  - Graphen-Algorithmen
  - Geometrische Algorithmen
  - Hintergrundspeicher

2 Datenabstraktion

2.1 Begriffe

Abstraktion (allgemein): Das Herausarbeiten der wesentlichen Merkmale. Unwesentliche Details werden dabei nicht in Betracht gezogen. Abstraktion tritt auch im täglichen Leben auf, z.B. Autofahren:

für den Benutzer wesentlich: Türschloss, Zündschloss, Lenkrad, Schaltknüppel, Gaspedal, Bremspedal, Kupplung, Blinker, Tankuhr

→ Die für den Benutzer wesentlichen Dinge fasst man auch als Benutzerschnittstellen auf.

unwesentlich: Motor, Räder, Tank etc.

→ Die unwesentlichen Dinge bilden die Implementierung der Abstraktion bzw. Benutzerschnittstelle. Diese Teile bilden jeweils eine andere Abstraktionsstufe.

Benutzerschnittstelle (interface): Funktionalität, die der Benutzer (Client) kennt.

Man hat oft viele Ebenen der Abstraktion (layers of abstraction), z.B. beim Hardwareentwurf:

höhere Programmiersprache
↑
Assembler
↑
CPU, Maschinenbefehle
↑
Flip-Flops (logische Ebene)
↑
physikalisches Bauteile

Sinn der Abstraktion:

- Konzentration auf das Wesentliche
- Arbeits- und Expertise-Aufteilung auf verschiedene Abstraktionsstufen
- Übersichtlichkeit (große Projekte nicht anders denkbar)

Algorithmen und Programmierung: Ein „Algorithmus“ ist bereits eine Abstraktion:

Algorithmus $\xrightarrow[Implementierung]{Realisierung}$ Programm (in einer höheren Programmiersprache z.B.)

Die abstrakte Beschreibung eines Algorithmus kann verbal oder mittels Pseudo-Code[1] geschehen, um es Menschen verständlicher zu machen.

Datenabstraktion durch abstrakte Datentypen (ADT): Dabei wird lediglich angegeben:

- Art von Daten
- darauf ausführbare Operationen

Die Spezifikation eines ADT ist eine Art bzw. ein Standard dies zu beschreiben. Dies geschieht mehr oder weniger formal z.B. mittels UML, mit mathematischen Mengen und Funktionen, Dokumentation der API in Java usw.

[1] benutzt Elemente von Programmiersprachen

Beispiel: Warteschlange (Queue) mit FIFO-Struktur

mathematisch:
endliche Folge Q von Elementen aus einem sog. Universum U.

darauf 2 Operationen:

- **enqueue(e)**: Füge Element e ans Ende von Q an
- **dequeue()**: nimmt das erste Element weg und gibt es an den Aurufer zurück

optional noch weitere Operationen:

- **size()**: Anzahl der Elemente in Q
- **isEmpty()**: Wahrheitswert, ob Q leer ist
- **front():** liefert erstes Element von Q ohne es zu löschen

Das spezifiziert den ADT was Benutzer oder Client angibt, um die Funktionalität des gewünschten ADT zu beschreiben: Schnittstelle

in Java: Syntax beschrieben durch Interface

```
public interface Queue<E> {

    public int size();
    public boolean isEmpty();
    public E front() throws EmptyQueueException;
    public void enqueue(E e);
    public E dequeue() throws EmptyQueueException;
}
```

2.2 Zusammenfassung VL 01

- Datenabstraktion
- ADT
- Bsp. Warteschlange
- endl. Folge $Q \subseteq U$ | Art der Daten
- Operationen: enqueu(e), dequeue(), ... $e \in U$
- in Java „interface“

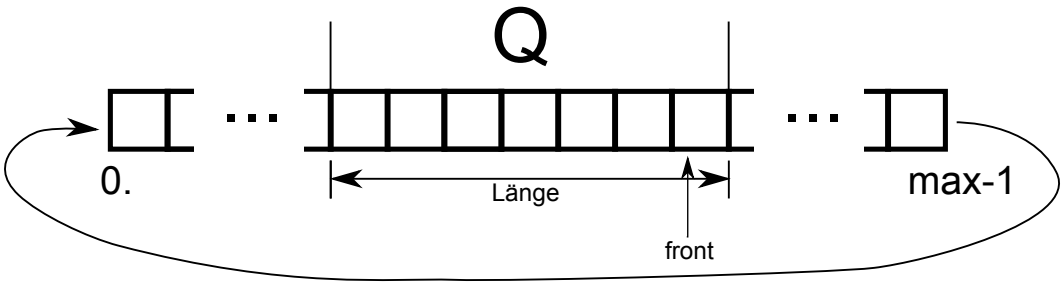
Bei der Implementierung werden meist noch andere Variablen/Methoden benutzt. Diese sollten nicht nach außen sichtbar sein.

2.3 Kapselung/Geheimnisprinzip

Nur die in der Schnittstelle spezifizierten Methoden/Attribute sind nach außen sichtbar. In Java ist dies möglich durch Modifizierer (modifiers):

Modifier	Sichtbarkeitsbereich
public	überall
protected	nur für Klassen innerhalb des gleichen Pakets (package) oder Unterklassen
<nichts>	nur im gleichen Paket
private	nur innerhalb der gleichen Klasse

Beispiel Warteschlange: z.B. linearer Ringpuffer in einem Feld



```
public class MyQueue<E> implements Queue<E>

    public final int max = 100;
    private E [] cell = new E[max];
    private int front = 0, laenge = 0;

    public int size() {
        return laenge;
    }

    public void enqueue(E e) throws OverflowException {
        ...
    }

    // usw. alle Methoden des Interface werden implementiert
```

Eine Klasse kann auch mehrere Schnittstellen implementieren, muss dann aber alle abstrakten Methoden aller Schnittstellen implementieren.

```
public class A implements B,C,D { ... }
```

2.4 Mehrfachvererbung

Zur Erinnerung: Vererbung

```
public class A extends B { ... }

//oder

public interface A extends B { ... }
// B muss hier dann ebenfalls ein Interface sein
```

Alle Methoden und Attribute von B stehen somit auch in A zur Verfügung.

Mehrfachvererbung, z.B. **A extends B,C,D** ist in Java nicht erlaubt für Klassen, wohl aber für Interfaces (**interface A extends B,C,D {...}**), jedoch müssen hier die Klassen von denen geerbt wird ebenfalls Interfaces sein.

2.5 Abstrakte Klassen

Abstrakte Klassen (gekennzeichnet mit 'abstract') enthalten implementierte Methoden und „abstrakte Methoden“, die nur Name und Signatur, wie bei einem Interface haben. Analog zu einem Interface ist keine Instanziierung mittels 'new' möglich.

Bsp: **java.lang.Number** ist eine abstrakte Klasse mit den Unterklassen Integer, Float, Double, Short, Long, Byte etc., die nicht abstrakt sind.

abstrakte Methoden in Number:

- intValue()
- doubleValue()

...

Jedoch: 'byteValue()' ist bspw. nicht mehr abstrakt.

2.6 Weitere Arten der Abstraktion (in Java)

2.6.1 Hierarchien von Klassen/Interfaces

Bsp:

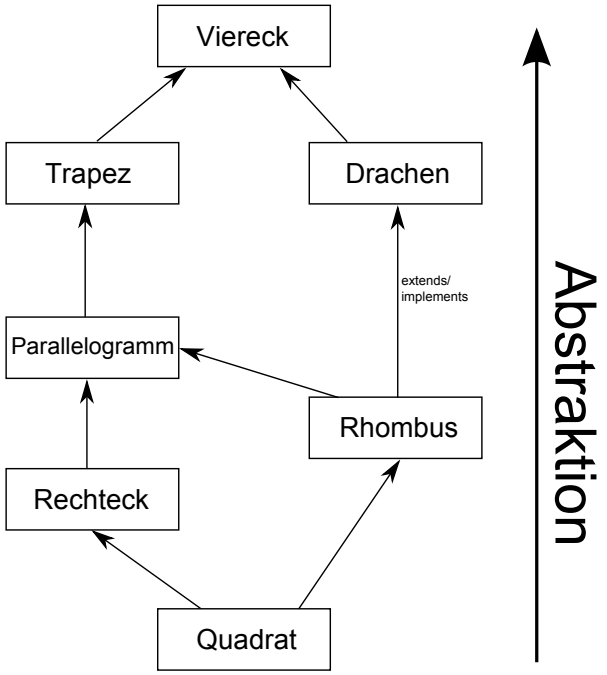


Abbildung 1: *Diese Hierarchie wäre realisierbar durch Interfaces und Klassen, wobei echte Mehrfachvererbung nicht möglich ist.*

2.6.2 Polymorphie

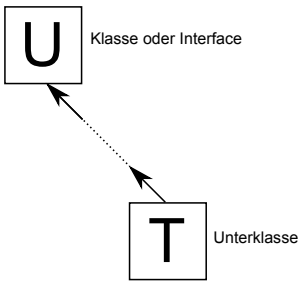
Der gleiche Name einer Methode ist auf verschiedene Typen anwendbar (wenn es in etwa das Gleiche bedeutet).

Bsp: Polymorphie des '+'-Operators:

$$5 + 3 \rightarrow 8$$
$$\text{"fuenf"} + \text{"drei"} \rightarrow \text{"fuenfdrei"}$$
$$\text{"fuenf"} + 3 \rightarrow \text{"fuenf3"}$$
$$\text{"fuenf"} + 3 + 2 \rightarrow \text{"fuenf32"}$$

'+' ist abstrahiert und seine konkrete Bedeutung wird erst durch seine Operanden festgelegt.

2.6.3 Casting



Problemlos ist eine erweiternde Konversion, bei der z.B. ein Objekt vom Typ T einer Variablen vom Typ U zuweist:

```
Integer i = new Integer(3);
Number n = i;    // erweiterte Konversion,
                 // da Integer Number erweitert
```

Umgekehrt ist eine einschränkende Konversion, bei der Variablen vom Typ T, Objekte vom Typ U zuweisen, nicht ohne Weiteres möglich. Aber:

```
Number n = new Integer(3); // erweiterte Konversion
Integer i = n; // einschraenkende Konversion - liefert Fehler
Integer i = (Integer)n; /* Typecast! - nur moeglich, wenn n auf
ein Objekt vom Typ Integer zeigt */
```

Bsp:

```
Number n;
Integer i;
n = new Integer(3);
i = (Integer)n;          } geht
n = new Double(3.5);
i = (Integer)n;          } geht nicht - Fehlermeldung
```

Man kann aber abfragen vom welchen Typ ein Objekt eigentlich ist:

```
<Objekt-Referenz> instanceof "Typ"
```

Bsp:

```
Number n = new Integer(3);
if(n instanceof Integer) {
    i = (Integer)n;
}
```

2.6.4 Casting mit Schnittstellen

Deklaration: f($\underbrace{A}_{Interface}$ x, ...) Parameter aller Implementierungen von A beim Aufruf möglich

Bsp:

```
public interface Person {
    public boolean equalTo (Person other);   // same person?
    public String getName();                 // person's name
    public int getAge();                     // person's age
}
```

```
public class Student implements Person {

    // Attributes
    String id;
    String name;
    int age;

    // Constructor
    public Student (String i, String n, int a) {
        id = i;
        name = n;
        age = a;
    }

    // Methods
    // just a guess
    protected int studyHours() { return age/2; }

    // ID of the student
    public String getID () { return id; }

    // from Person interface
    public String getName() { return name; }

    // from Person interface
    public int getAge() { return age; }

    // from Person interface
    public boolean equalTo (Person other) {
        // cast Person to Student
        Student otherStudent = (Student) other;
        // compare IDs
        return (id.equals (otherStudent.getID()));
    }

    // for printing
    public String toString() {
        return "Student(ID: " + id +
          ", Name: " + name +
          ", Age: " + age + ")";
    }
}
```

Das Problem beim obigen Beispiel besteht darin, dass in der Klasse 'Student' der Cast in der Methode 'equalTo' nur funktioniert, wenn Objekte vom Typ 'Student' verglichen werde, da nur diese das Attribut 'id' haben. Dieser Cast ist aufgrund der einschränkenden Konversion von Person nach Student erforderlich.

2.7 Generische Datentypen - Generics

Generics sind eine Möglichkeit der Abstraktion seit Java 5.0. Dabei wird in der Klassendefinition hinter dem Klassennamen eine Art Platzhalter in der Form **<E1,E2,...>** eingefügt ($E1, E2$ usw. bilden also Typvariablen, die in der Klassendefinition benutzt werden können).
Bei einer Instanziierung müssen sie durch konkrete Typenamen ersetzt werden.

Bsp:

```
public class Pair<K, V> {
    K key;
    V value;

    public void set(K k, V v) {
        key = k;
        value = v;
    }

    public K getKey() { return key; }

    public V getValue() { return value; }

    public String toString() {
        return "[" + getKey() + ", " + getValue() + "]";
    }

    public static void main (String[] args) {
        Pair<String,Integer> pair1 = new  Pair<String,Integer>();
        pair1.set(new String("height"), new Integer(36));

        System.out.println(pair1);
        // Output: [height,36]

        Pair<Student,Double> pair2 = new  Pair<Student,Double>();
        pair2.set(new Student("A5976","Sue",19), new Double(9.5));

        System.out.println(pair2);
        // Output: [Student (ID: A5976, Name: Sue, Age: 19), 9.5]

        //toString() in Klasse Student wird implizit aufgerufen
    }
}
```

Typvariablen in Haskell:

```
length :: [a] -> Int     // a ist hier die Typvariable
length [] = 0
length (x:xs) = length xs + 1
```

Ein generischer Datentyp kann durch "extends" eingeschränkt werden.

Bsp:

```
class Sort<E extends Comparable<E>> {
    ... // Elemente vom Typ E sortieren
}
```

Auch Methoden können als Modifikatoren eine Liste von Typvariablen haben.

Bsp:

```
public static <E extends Comparable<E>> void quickSort(E[] a) {
    // <E ... > ist hier NICHT der Rueckgabetyp
    ...
}
```

Achtung:

```
public static void main(String[] args) {

    // nicht erlaubt ist folgende Zuweisung!
    Pair<String,Integer>[] a = new Pair<String,Integer>[10];

    //erlaubt ist jedoch:
    Pair<String,Integer>[] a = new Pair[10];

    a[0] = new Pair<String,Integer>();
}
```

3 Entwurf und Analyse von Algorithmen

Als erstes Beispiel betrachten wir vier Sortieralgorithmen, wobei eine Folge von n Objekten aus einem Universum U auf dem eine totale Ordnung[2] $\leq$ definiert ist.

1. Slow Sort Erzeuge systematisch alle Permutationen der Eingabefolge und prüfe für jede nach, ob diese aufsteigend sortiert ist.[3]

2. Selection Sort Durchlaufe die Folge und finde ihr Maximum. Schreibe dies an die rechteste Stelle. Durchlaufe die Restfolge ebenso usw. bis die Ausgangsfolge sortiert ist.

3. Quick Sort Wenn die Folge aus nur einem Element besteht ($n = 1$), tue nichts. Andernfalls wähle ein Element a ("Pivot-Element") und spalte die restlichen Elemente so auf, dass man eine Folge von Elementen kleiner gleich a (S_1) und eine Folge von Elementen größer a (S_2) erhält. Wiederhole den Algorithmus rekursiv auf die Teillisten.
Ausgabe: S_1 (sortiert) + a + S_2 (sortiert)

[2] je zwei Elemente sind vergleichbar

[3] eine weitere Variante besteht darin, dass die Permuationen zufällig erzeugt werden, wobei dann nicht garantiert ist, dass der Algorithmus terminiert.

4. Merge Sort Wenn die Folge nur ein Element aufweist, beende den Algorithmus, sonst spalte die Folgen so in Teilfolgen dass die erste Teilfolge S_1 die Größe $\lceil n/2 \rceil$ und die zweite Teilfolge S_2 die Größe $\lfloor n/2 \rfloor$ hat. Wiederhole das rekursiv für S_1 und S_2 und mische die sortierten Teilfolgen zu einer sortierten Gesamtfolge.

3.1 Theoretische Analyse

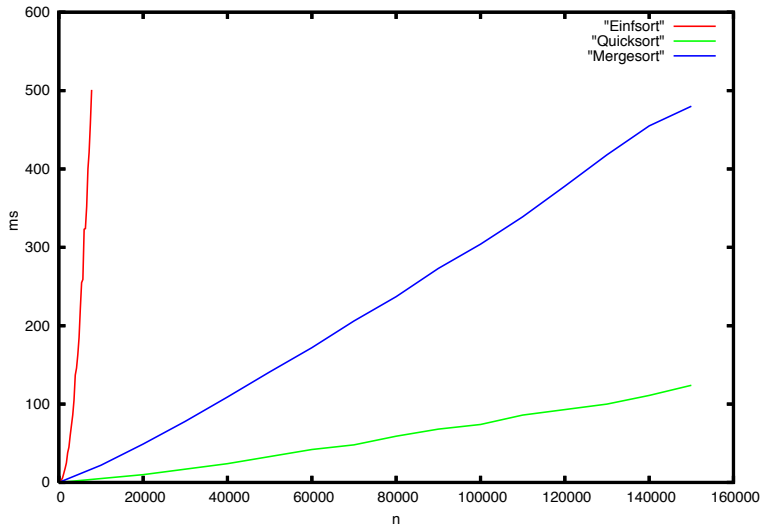


Abbildung 2: *Experimenteller Vergleich einiger ausgewählter Sortieralgorithmen*

Wir analysieren die Anzahl der Vergleiche als Funktion von n. Die Vergleichsanzahl ist proportional zur eigentlichen Laufzeit, die wiederum vom jeweiligen Rechner abhängig ist.

Alg. 1: im schlechtesten Fall müssen alle Permutationen durchprobiert werden:

$n!$ Permutationen
$n-1$ Vergleiche
$(n-1) \cdot n!$
"im Mittel": $\frac{n!}{2}$ Durchläufe und $\geq \frac{n!}{2}$ Vergleiche

Alg. 2:

$n-1$ Vergleiche im 1. Durchlauf
$n-2$ Vergleiche im 2. Durchlauf
$\vdots$

$$\Rightarrow \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

Alg. 4:

$$\begin{aligned}&\text{Ansatz: } C(n) \text{ Vergleiche}\\&\text{wissen: } C(1) = 0\\&C(n) \leq 2 \cdot \underbrace{C(\frac{n}{2})}_{\text{rekursive Aufrufe}} + \overbrace{n}^{\text{für's Mischen}}\end{aligned}$$

Aus der Rekursions(un)gleichung lässt sich C bestimmen, wenn $n = 2^k$, also n eine Zweierpotenz ist.

$$\begin{aligned}C(n) &\leq 2 \cdot \underbrace{C(\frac{n}{2})}_{\leq 2C(\frac{n}{4})+\frac{n}{2}} + n\\&\leq 4C(\frac{n}{4}) + n + n\\&\vdots\\&\leq 2^l \cdot C(\frac{n}{2^l}) + l \cdot n \quad \text{für alle } l \leq log(n)\end{aligned}$$

Setze $l = k = log\ n$

$$\boxed{C(n) \leq 2^{log(n)} \cdot \underbrace{C(\underbrace{\frac{n}{2^k}}_{=1})}_{=0} + n \cdot log(n)}$$

Strategie: Das Problem wird in kleinere Teilprobleme aufgeteilt und rekursiv gelöst. Die Teillösungen werden dann am Ende wieder zusammengemischt → „*teile-und-herrsche (divide-and-conquer)*“

Alg. 3: Die Strategie ist hier ebenfalls "teile-und-herrsche".
Sei $C(n)$ die Anzahl der Vergleiche für eine Folge der Länge n:

$$\begin{aligned}&C(0) = 0\\&C(1) = 0\\&C(n) = \underbrace{C(k-1)}_{\text{rek. Aufruf für } S_1} + \underbrace{C(n-k)}_{\ldots S_2} + \underbrace{(n-1)}_{\text{für's Aufspalten}} \qquad \text{falls } a \text{ das k. kleinste Element ist}\end{aligned}$$

ungünstige Fälle: $k = 1$ oder $k = n$

$$
\begin{aligned}
C(n) = & \underbrace{C(n-1)}_{C(n-2)+n-2} + \underbrace{C(0)}_{=0} + (n-1) \\
& \vdots \\
& = C(n-r) + n - r + n - r + 1 + ... + n - 1 \\
& \overset{(r = n-1)}{=} \underbrace{C(1)}_{=0} + 1 + 2 + ... + n - 1 = \frac{1}{2}n^2 - \frac{1}{2}n
\end{aligned}
$$

Anzahl der Vergleiche „im Mittel“:
Wir nehmen an, dass jede Permutation der Eingabefolge von der aufsteigenden Ordnung gleichwahrscheinlich ist.

Erwartungswert der Anzahl der Vergleiche:

$$C(0) = 0$$
$$C(1) = 0$$
$$C(n) = \sum_{k=1}^{n} \underbrace{\frac{1}{n}}_{\substack{\text{Wahr-}\\ \text{scheinlichkeit,}\\ \text{dass } a \text{ k.}\\ \text{kleinstes}\\ \text{Element ist}}} \cdot \overbrace{[C(k-1) + C(n-k)]}^{\substack{\text{erwartete Kosten,}\\ \text{in dem Fall}}} + n - 1$$

$$n \cdot C(n) = \sum_{k=1}^{n} [C(k-1) + C(n-k)] + n(n-1) \tag{1}$$

$$(n-1) \cdot C(n-1) = \sum_{k=1}^{n-1} [C(k-1) + C(n-1-k)] + (n-2)(n-1) \tag{2}$$

Subtrahiert man nun Glg. 2 von Glg. 1, kommt man zu folgendem Zwischenergebnis:

$$n \cdot C(n) - (n-1) \cdot C(n-1) = C(n-1) + C(n-1) + 2 \cdot (n-1)$$

und vereinfacht man das nun weiter:

$$\begin{aligned}
n \cdot C(n) &= (n+1) \cdot C(n-1) + 2 \cdot (n-1) \\
C(n) &\leq \frac{n+1}{n} \cdot \underbrace{C(n-1)}_{\frac{n}{n-1} \cdot C(n-2) + 2 \cdot \frac{n-2}{n-1}} + 2 \\
&= \frac{n+1}{n} \cdot \frac{n}{n-1} \cdot C(n-2) + 2 \cdot (\frac{n+1}{n+1} + \frac{n+1}{n}) \\
&\leq \frac{n+1}{n-1} \cdot \frac{n-1}{n-2} \cdot C(n-3) + 2 \cdot (\frac{n+1}{n+1} + \frac{n+1}{n} + \frac{n+1}{n-1}) \\
&\vdots \\
&\leq \frac{n+1}{n-r} \cdot C(n-r+1) + 2 \cdot (n+1)(\frac{1}{n+1} + \frac{1}{n} + ... + \frac{1}{n-r+1}) \qquad \text{wobei } r = n-1 \\
&\leq \underbrace{\frac{n+1}{1} \cdot C(0)}_{=0} + 2 \cdot (n+1)(\frac{1}{n+1} + ... + \frac{1}{2})
\end{aligned}$$

$$
\begin{aligned}
C(n) &\leq 2 \cdot (n+1)(\frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n+1}) \\
&\leq 2 \cdot (n+1)(H_{n+1} - 1) \\
&\text{wobei } H_k = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{k} \quad \underline{\text{k-te harmonische Zahl}}
\end{aligned}
$$

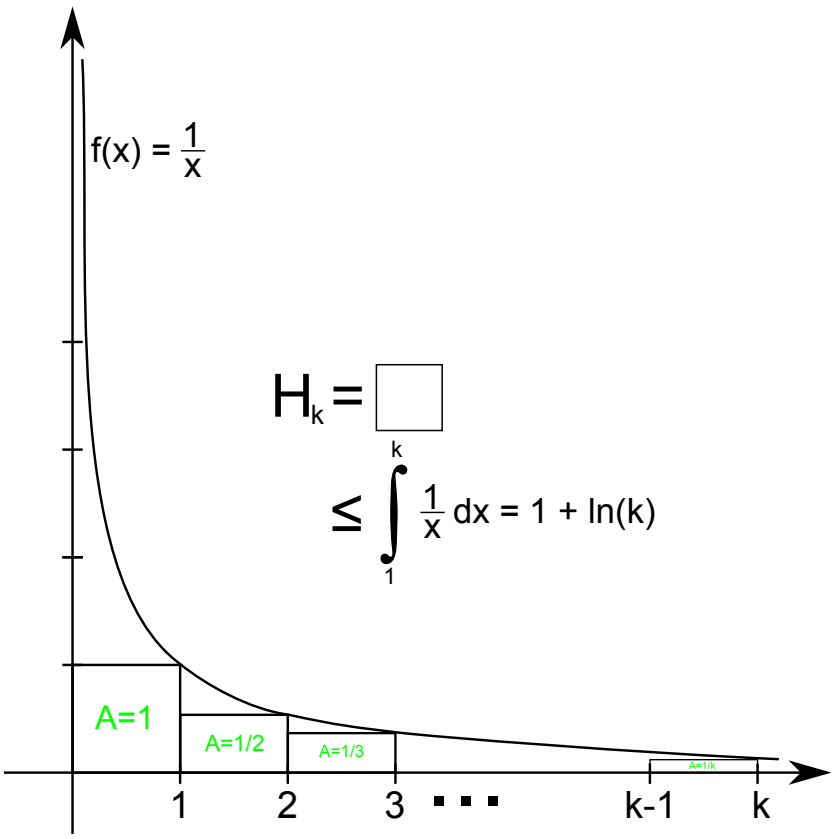


Abbildung 3: *Visualisierung der k-ten harmonischen Zahl*

$$
\begin{aligned}
&\boxed{C(N) \leq 2 \cdot (n+1) \cdot ln(n+1)} \qquad \text{mittlere Laufzeit von Quick Sort} \\
&= 2(n+1) \cdot \frac{log_2(n+1)}{log_2(e)} \\
&\approx 1.38 \cdot (n+1) \cdot log(n+1) \qquad \text{Vgl. Merge Sort: } n \cdot log(n)
\end{aligned}
$$

Im Experiment (siehe Diagramm 2) ist Quick Sort trotzdem schneller als Merge Sort, da das Mischen selbst auch nochmal Zeit kostet.

Statt deterministischer Wahl des Pivot-Elements, kann man dieses zufällig wählen (gleichverteilt bzgl. Position)[4]. Dann ist die erwarte Anzahl von Vergleichen für jede Eingabe $1.38 \cdot n \cdot log(n)$ („randomisiertes Quicksort“)

3.1.1 Vergleich der Funktionen

n	$nlog(n)$	$1/2n^2 - 1/2n$	n!
10	33	45	$3.6 \cdot 10^6$
20	86	190	$2.4 \cdot 10^{18}$
30	282	1225	$3.0 \cdot 10^{64}$

Annahme: 10^9 Vergleiche pro Sekunde

Anzahl der Vgl. in der Zeit	$nlog(n)$	$1/2n^2 - 1/2n$	n!
1 Sekunde	$4 \cdot 10^7$	$4 \cdot 10^4$	11
1 Stunde	$1 \cdot 10^{11}$	$2.6 \cdot 10^6$	14

3.1.2 Lehren aus diesen Beispielen

1. Ein Algorithmus kann auch verbal formuliert werden und nicht unbedingt durch ein Programm. Vor allem kann er in dieser Form bereits analysiert werden, z.B. etwa die Laufzeit[5]:

von Slowsort: $O(n \cdot n!)$[6] im schlechtesten Fall bzw. $\geq c \cdot n!$ im Mittel.
Sort mit Max-Suche: $\Theta(n^2)$
Mergesort: $\Theta(n \cdot log(n))$
Quicksort: $\Theta(n^2)$ im schlechtesten Fall und $\Theta(n \cdot log(n))$ im Mittel

2. Die Analyse ist wichtig! Verschiedene Algorithmen für das gleiche Problem, können sich wesentlich in der Laufzeit unterscheiden.

3. Die Entwurfsstrategie „Teile-und-herrsche“(Divide-and-Conquer), bei der das Problem in kleinere Teilprobleme aufgeteilt wird, die dann rekursiv gelöst werden. Am Ende werden die Teillösungen zu einer Gesamtlösungen zusammengeführt. Für die Analyse der Laufzeit ergibt sich an dieser Stelle eine Rekursions(un)gleichung, die gelöst werden muss.

4. Es gibt Algorithmen, deren Entscheidungen vom Zufall abhängen, die also „randomisiert“ sind. An dieser Stelle sind zwei Varianten davon zu erwähnen:
Las-Vegas-Algorithmus: das Ergebnis ist auf jeden Fall richtig und die Laufzeit hängt vom Zufall ab.
Monte-Carlo-Algorithmus: die Korrektheit hängt vom Zufall ab und die Laufzeit ist fest.

[4] Eine weitere Variante besteht darin, 3 Elemente zufällig zu wählen und das mittlere als Pivot-Element anzunehmen, wobei nur der konstante Vorfaktor dadurch verbessert werden kann.

[5] die tatsächliche Laufzeit ist hier proportional zur Anzahl der Vergleiche, deswegen sind hier O und Θ angebracht

[6] Anmerkung: an dieser Stelle kam die Frage auf, ob hier nun O oder Θ stehen soll.

3.2 Allgemeines zu Algorithmen und dazugehöriger Analyse

Algorithmus: Ein Algorithmus ist ein Verfahren zur Lösung eines Problems, das sich „mechanisieren“ lässt. Die Beschreibung eines solchen kann umgangssprachlich oder in einer Programmiersprache, deren Semantik exakt definiert ist oder exakten mathematischen Formalisierungen erfolgen. Aber auch durch eine Turingmaschine oder mittels Registermaschinen (Random Access Machine - RAM) lässt sich ein Algorithmus beschreiben.

In dieser Vorlesung behalten wir das RAM-Modell im Hinterkopf, wenn wir Algorithmen analysieren, wobei 1 Befehl in einem Zeitschritt ausgeführt wird und 1 Register einer Speichereinheit entspricht.

3.2.1 RAM-Modell

Eingabe I steht anfangs in Registern $0, ..., n$, wobei n die Größe der Eingabe repräsentiert.

Laufzeit des Algorithmus bei Eingabe I ist die Anzahl der Schritte, bis er hält, wobei die Laufzeit meist eine Funktion der Eingabegröße ist. Im schlechtesten Fall ist die Laufzeit das Maximum über alle Laufzeiten bei Eingabe der Länge n. Im Mittel ist sie der entsprechende Erwartungswert der Laufzeiten der Eingabe der Länge n.
Bei randomisierten Algorithmen ist die Laufzeit der Erwartungswert über alle Zufallsentscheidungen bei Eingabe der Länge n.

Platz-/Speicherbedarf eines Algorithmus bei Eingabe I entspricht der Anzahl der verwendeten Register als Funktion von der Eingabegröße n, analog zur Laufzeit.

Wir werden Algorithmen meist umgangssprachlich oder Teile davon in Programmiersprachen (wenn es exakt sein soll) oder in einer Zwischenstufe davon (Pseudocode) beschreiben. Der Pseudocode enthält bspw. While-Schleifen, Rekursionen usw. Es sollte dabei klar sein, wie die Umsetzung in RAM-Code (Laufzeit, Platzbedarf) aussieht.

Vorsicht: Versteckter Platzbedarf bei Rekursion. Hierbei wird pro Prozeduraufruf ein Laufzeitstack angelegt, auf dem ein Datensatz mit den nötigen Informationen zum entsprechenden Aufruf (Rücksprungadresse, Funktionsparameter etc.), abgelegt wird. Der zusätzliche Platzbedarf ist dann proportional zur Verschachtelungstiefe der Rekursion.

3.2.2 Wachstum von Funktionen

Bsp.: $\sqrt{n},\ log(n)^5,\ n,\ n^2,\ n^{log(n)},\ n!,\ 2^n$

z.B:

$$\sqrt{n} \geq log(n)^5 \text{ für hinreichend große } n.$$

hinreichend heißt in dem Fall: $\exists n_0 \in \mathbb{N} \forall n \geq n_0$

Anwendung des Logarithmus auf beiden Seiten der Ungleichung liefert:

$$\frac{1}{2} log(n) \geq 5 \cdot log(log(n))$$
$$\frac{log(n)}{log(log(n))} \geq 10 \qquad \text{für } n \approx 2^{64}$$

Allgemein gilt: n^α wächst stärker als $log(n)^k$ für alle $\alpha > 0$, $k > 0$. Mit anderen Worten:

$$log(n)^k = o(n^\alpha)$$

3.2.3 O-Notation

$$f : \mathbb{N} \to \mathbb{N},\ g : \mathbb{N} \to \mathbb{N}$$
$$f = O(g) \Leftrightarrow \exists c > 0 : f(n) \leq c \cdot g(n) \qquad \text{für hinreichend große } n$$
$$f = \Theta(g) \Leftrightarrow f = O(g) \wedge g = O(f)$$
$$f = \Omega(g) \Leftrightarrow \exists c > 0 : f(n) \geq c \cdot g(n)$$
$$f = o(g) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Anzahl der Vergleiche von Quicksort im schlechtesten Fall:

$$\frac{1}{2}n^2 - \frac{1}{2} = \Theta(\underbrace{n^2}_{g(n)})$$

denn:

$$f(n) \leq 1 \cdot g(n)$$
$$g(n) \leq 3 \cdot f(n) \qquad \text{für } n \geq 3$$

3.2.4 Typische Funktionen bei der Analyse von Algorithmen

$O(1)$	Funktion, die durch eine Konstante beschränkt ist, z.B. $f(n) = 2 \forall n \in N$ oder $f(n) = \begin{cases} 1 \text{ gerade } n \\ 2 \text{ ungerade } n \end{cases}$
$O(log(log(n)))$	z.B.: **asymptotische** mittlere Laufzeit von Interpolationssuche
$O(log(n))$	z.B.: Binärsuche, **logarithmisch**
$O(log(n)^k)$	Konstante k, **polylogarithmisch**
$O(n^\alpha)$	Konstante $1 > \alpha > 0$, z.B.: $\alpha = \frac{1}{2} \Rightarrow O(\sqrt{n})$
$O(n)$	**linear**
$O(n \cdot log(n))$	
$O(n^2)$	**quadratisch**
$O(n^3)$	**kubisch**
$O(n^k)$	für feste k: **polynomiell**
$O(c^n)$	$c > 1$, **exponentiell**
$O(c^{n^k})$	$c > 1$, $k \geq 1$, **exponentiell**
$O(n!)$	**exponentiell**
$O(2^{n^2})$	**exponentiell**
$O(2^{2^n})$	**doppelt exponentiell**

$a, b > 1$, dann:

$$\begin{aligned} log_a^n &= \Theta(log_b(n)) \\ n &= a^{log_a(n)} = b^{log_b(n)} \\ log_a(n) \cdot log_2(a) &= log_b(n) \cdot log_2(b) \\ log_a(n) &= \frac{log_2(b)}{log_2(a)} \cdot log_b(n) \end{aligned}$$

3.3 Das Auswahlproblem

Bestimme die 5. Größte Zahl.

11 27 10 2 <u>17</u> 18 23 5 34 53

Frage: In $O(n)$ lösbar?

Gegeben: n Elemente aus einem Universum $(U, \underbrace{\leq}_{\text{lineare Ordnung}})$ und $k \in N$, $1 \leq k \leq n$.

Finde: das k. kleinste Element, also das an k-ter Stelle befindliche Element, wenn die Folge aufsteigend sortiert ist, der Folge.

naheliegendes Verfahren: sortiere und wähle das Element an k-ter Stelle.

Laufzeit: $\Theta(n \cdot log(n))$, falls schneller Sortieralgorithmus benutzt wird. Frage: Geht es schneller?

„Sonderfälle" für k:

$$k = 1 \Rightarrow \text{Minimum}$$
$$k = n \Rightarrow \text{Maximum}$$
$$k = \left\lfloor \frac{n}{2} \right\rfloor \Rightarrow \text{Median}$$

3.3.1 Algorithmus Quickselect

Dieser Algorithmus funktioniert nach dem „Teile-und-Herrsche"-Prinzip:

Falls $|S| > 1$:

- Wähle ein zufälliges Element a aus der Folge S
- teile Folge S auf in:
  - $S_<$ (Elemente $< a$)
  - $S_=$ (Elemente $= a$)
  - $S_>$ (Elemente $> a$)
- falls $k \leq |S_<|$: suche rekursiv k. kleinstes in $|S_<|$
- sonst, falls $k \leq |S_<| + |S_=|$: gib a zurück
- sonst: suche rekursiv das $(k - |S_<| - |S_=|)$. kleinste Element in $S_>$

Laufzeit: Sei $n = |S|$ proportional zu Zahl der Vergleiche T(n)

$$T(1) = 0$$

Falls a das i. kleinste Element war:

$$T(n) = \underbrace{n-1}_{\text{für's Aufspalten}} + \begin{cases} T(i-1), & \text{falls } k < i \\ T(n-i), & \text{falls } k > i \\ 0, & \text{falls } k = i \end{cases}$$

im schlechtesten Fall: $i = n$ oder $i = 1$ und $k \neq i$, dann:

$$T(n) = n - 1 + T(n-1) = (n-1) + (n-2) + ... + 1 = \frac{n(n-1)}{2} = \Theta(n^2)$$

im Mittel:

$$T(n) \leq \underbrace{\frac{1}{n}}_{\substack{\text{Wahrschein-}\\ \text{lichkeit, dass}\\ \text{a i. kleinstes}\\ \text{Element ist}}} \sum_{i=1}^{n} \underbrace{max\{T(i-1), T(n-i)\}}_{\text{Absch. Laufzeit für diesen Fall}} + n - 1$$

Behauptung: $T(n) \leq c \cdot n$ für eine Konstante k

Beweis: durch Induktion über n (welches c wird sich noch herausstellen)

IA: $n = 1$

$$T(1) = 0 \leq c \cdot 1 \text{ für alle } c > 0$$

IS: $\leq n - 1 \to n$

$$T(n) \leq \frac{1}{n} \sum_{i=1}^{n} max\{ \underbrace{c \cdot (i-1)}_{\text{nach IV} \geq T(i-1)}, c(n-1)\} + n - 1$$

also:

$$T(n) \leq \frac{2c}{n} \cdot \underbrace{(n-1+n-2+...+\left\lfloor \frac{n}{2} \right\rfloor)}_{\frac{n(n-2)}{2} - \lceil \frac{n}{2} \rceil \frac{(\lceil \frac{n}{2} \rceil - 1)}{2} \leq \frac{3}{8}n^2}$$

$$T(n) \leq \frac{2c}{n} \cdot \frac{3}{8} \cdot n^2 + n - 1$$

$$= \Big(\frac{3c}{4} + 1\Big) n$$

$$\leq cn \text{ für } c \geq 4$$

Fazit: Quickselect benötigt im Mittel $\leq 4n$ Vergleiche

4 Prioritätswarteschlangen

Prioritätswarteschlangen sind die Art von Warteschlangen, bei denen den Objekten eine Priorität zugewiesen wird.

Bsp.:

- Ambulanz im Krankenhaus
- Prozesse im Rechner: Betriebssystem legt Prioritätswarteschlange an

4.1 Abstrakter Datentyp

Gegeben ist eine Menge von Paaren (k, x), wobei $k \in S$ mit $(S, \leq)$ (linear geordnetes Universum), die Priorität bzw. den Schlüssel und $x \in W$ den Wert eines Elements entspricht.

Operationen:

einfuege(k,x): (k,x) wird in PWS eingefügt

streicheMin(): Element mit min. Schlüssel (= höchste Prio.) wird entfernt und zurückgegeben

groesse(): Anzahl der Elemente in PWS

istLeer(): wahr oder falsch

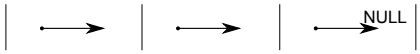
min(): Element mit minimalen Schlüssel wird zurückgegeben ohne es zu entfernen

in Java:

```
public interface Eintrag<S extends Comparable<S>,W> {
    public S getSchluessel();
    public W getWert();
}

public interface PrioQueue <S extends Comparable<S>, W> {

    // optional: public void einfuege(Eintrag<S,W>);
    public void einfuege(S schluessel, W wert);

    // optional: public Eintrag<S,W> streicheMin();
    public W streicheMin() throws LeereWarteSchlangeException;

    public int groesse();
    ...
}
```

zugehörige Datenstruktur? $\Rightarrow$ Implementierung

4.1.1 einfachverkette Liste



Die Liste kann aufsteigend sortiert oder unsortiert sein. Im ersten Fall benötigen streicheMin(), groesse(), istLeer(), min() eine konstante Zeit $O(1)$. Bei n Elementen in der Prioritätswarteschlange benötigt einfuege() im schlimmsten Fall $O(n)$ Zeit, da die Liste durchlaufen werden muss, damit das Element an der richtigen Stelle eingefügt werden kann.

4.1.2 Heap

Ein Heap ist ein Binärbaum, in dessen Knoten die Schlüssel abgelegt sind, wobei die k-te Ebene ($k = 0, 1, 2, ...$) von oben 2^k Knoten enthält. Die unterste Ebene muss nicht vollständig sein und wird von links nach rechts aufgefüllt.

Ein Schlüssel in einem Knoten ist immer kleiner-gleich dem Schlüssel in seinen Kindern.

Bsp:

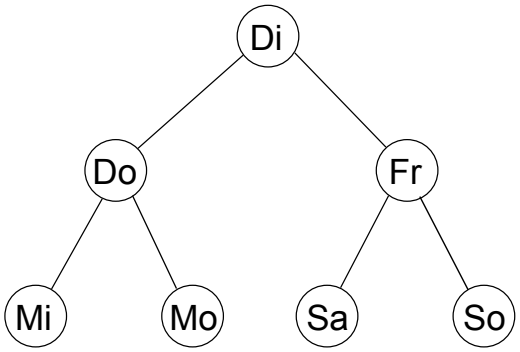


Abbildung 4: *Wochentage in ihrer lexikografischen Reihenfolge in einem Heap angeordnet.*

Fakten zu Heaps:

1. das Minimum der Schlüsselmenge steht in der Wurzel
2. jeder Teilbaum eines Heaps ist ein Heap
3. die Höhe eines Heaps mit n Knoten ist $\lfloor log(n) \rfloor$

Beweis zu 3.: Gegegeben sei ein Heap der Höhe h mit n Knoten. Dann ist $n \leq 1+2+...+2^h = 2^{h+1} - 1$

die ersten k-1 Ebenen sind gefüllt:

$$\begin{aligned} n &\geq 1 + 2 + 4 + ... + 2^{h-1} \\ &= 2^h - 1 \\ &< n \\ &\leq 2^{h+1} - 1 \end{aligned}$$

$$\Rightarrow 2^h < n + 1 \leq 2^{h+1}$$

$$\begin{aligned} \Rightarrow h < log(n+1) &\leq h + 1 \\ h \leq log(n) &< h + 1 \end{aligned}$$

$$\Rightarrow h = \lfloor log(n) \rfloor$$

4.1.3 Laufzeit der Operationen bei einem Heap (Halde)

groesse(), istLeer() und min() benötigen $O(1)$ Zeit.

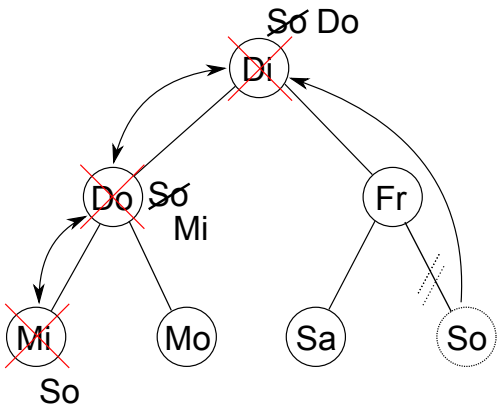


Abbildung 5: *Löschen von Dienstag*

streicheMin():

Wir ersetzen den Schlüssel in der Wurzel durch den des rechtesten Knotens auf der untersten Ebene und entfernen diesen. Der Schlüssel x in der Wurzel wird anschließend mit denen der beiden Kinder y und z verglichen. Falls $x > min(y, z)$, vertauschen wir x mit dem entsprechenden Minimum. Dies wird dann rekursiv für alle Unterbäume wiederholt, bis man die unterste Ebene erreicht hat.

Laufzeit: Das Ersetzen des Schlüssels in der Wurzel kostet konstante Zeit $O(1)$. Das Verhalden der neuen Wurzel bis eventuell zu einem Blatt benötigt pro Ebene konstante Zeit $O(1)$. Insgesamt benötigt das Verhalden also im schlimmsten Fall $\underline{O(log(n))}$.

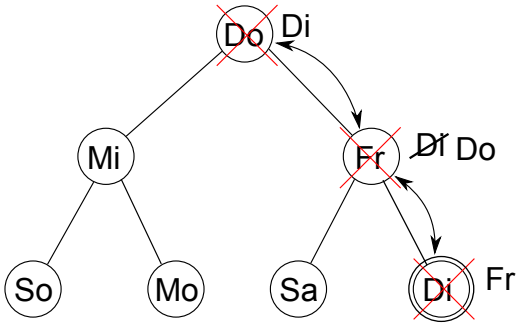


Abbildung 6: *Einfügen von Dienstag*

einfuege():

Wir schaffen auf der untersten Ebene rechts vom rechtesten Knoten einen neuen Knoten, wenn diese Ebene nicht voll ist. Sonst wird ein neuer Knoten in einer neuen Ebene ganz links eingefügt. Der neue Knoten wird mit dem Schlüssel des zu einfügenden Elements beschriftet.
Dieser Eintrag wird dann mit dem des Vaters verglichen. Ist er kleiner, werden beide Einträge vertauscht. Dies wiederholt man so lange rekursiv, bis man auf einen Knoten mit einem größeren Eintrag trifft oder man in der Wurzel angekommen ist.

Laufzeit: Das Anfügen des neuen Knotens an sich benötigt konstante Zeit $O(1)$, genauso wie der Vergleich pro Ebene. Insgesamt wird also im schlimmsten Fall $\underline{O(log(n))}$ Zeit für das Einfügen und Neuverhalden benötigt.

Es gibt eine effiziente Realisierung eines Heaps durch ein Feld. Wenn man die Knoten ausgehend von der Wurzel und Ebene für Ebene von links nach rechts durchnummeriert, haben die Kinder (falls vorhanden) des i-ten Knotens die Indizes $2i$ und $2i+1$. Der Vater des i-ten Knotens hat dann den Index $\lfloor \frac{i}{2} \rfloor$.
Somit kann man den Heap einfach mit einem Feld H realisieren:

aktuelle Fassung: startindex = 0,
linkes Kind hat 2i+1, rechtes Kind hat 2i +2

Vater hat: abgerundet((i-1)/2);

$$
\begin{aligned}
H[i]: & \\
\text{Kinderknoten:} & \rightarrow H[2i] \text{ und } H[2i+1] \\
\text{Vaterknoten:} & \rightarrow H[\lfloor \frac{i}{2} \rfloor]
\end{aligned}
$$

5 Wörterbücher

5.1 Abstrakter Datentyp

Ein Wörterbuch ist eine endliche Menge $D \subseteq S \times W$ von Einträgen der Form (k, v), wobei $k \in S$ mit S als Menge aller möglichen Schlüssel und $v \in W$ mit W als Menge aller möglichen Werte.

5.1.1 Operationen

finde(k): Finde (einen) Eintrag mit Schlüssel k

einfuege(k,v): erweitere D zu $D \cup \{(k, v)\}$

streiche(e): streiche Eintrag e aus D, falls vorhanden

groesse(): Elementanzahl $|D|$ von D

istLeer(): wahr, falls $D = \varnothing$; falsch sonst

Andere Operationen:

findeAlle(): Menge der Einträge mit Schlüssel k

```
package woerterbuch;
import java.util.Collection;

public interface Woerterbuch<S,W>{

    //wichtige Operationen
    public Eintrag<S,W> finde(S schluessel);
    public void einfuege (S schluessel, W wert);
    public void streiche (Eintrag<S,W> e);

    // weitere nuetzliche Operationen
    public int groesse();
    public boolean istLeer();
    public Collection<W> findeAlle(S schluessel);
}
```

5.2 Implementierungen

Falls auf S eine lineare Orndung besteht und nur die Operation *finde()* ausgeführt werden soll:

5.2.1 Feld

Man kann D in einem Feld ablegen, das nach Schlüsseln sortiert ist. *finde()* kann dann durch eine Binärsuche implementiert werden. Die Laufzeit davon beträgt dann $O(log(n))$, wobei $n = |D|$.

5.2.2 allgemein: verkette Liste

Als weitere Möglichkeit kann man eine verkette Liste als Datenstruktur verwenden:

finde() benötigt dann $O(n)$ Zeit, da die ganze Liste durchlaufen werden muss.

einfuege(k,v) in $O(1)$ Zeit, da das Element nur hinten angefügt wird.

streiche(e) in $O(n)$ Zeit, da auch hier die ganze Liste durchlaufen werden muss, um e zu finden.

5.3 Hashing (Streuspeicherung)

Idee: Für die Menge D steht ein Feld $A[0 \ldots N-1]$ zur Verfügung („Hashtabelle“). Es gibt, dann eine Funktion $h : S \to \{0, ..., N-1\}$, die sog. Hashfunktion.
Für einen Schlüssel k gilt dann, dass $A[h(k)]$ die Position ist, an der die Einträge (k,v) „hingehören“.

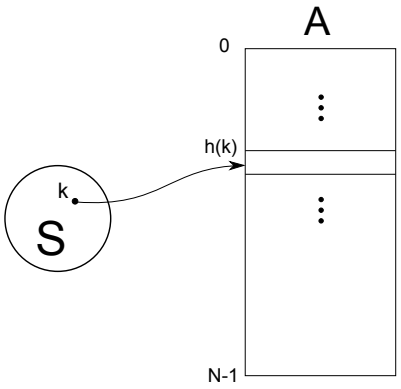


Abbildung 7: *Schematische Darstellung einer Hashfunktion.*

Problem:

$$k_1, k_2 \in S \text{ mit } k_1 \neq k_2 \text{ aber } h(k_1) = h(k_2)$$
$$\Rightarrow \text{Kollision}$$

Für kleine Mengen S aller möglichen Schlüssel könnte $N \geq |S|$ und h injektiv gewählt werden. Somit gäbe es keine Kollisionen mehr und man hätte „perfektes Hashing“ erreicht. Meistens allerdings gilt $|S| \gg N$, wobei $|S|$ eventuell ∞ sein kann.

Eine Möglichkeit Kollisionen zu beheben besteht darin, dass A ein Feld von verketten Listen darstellt, wobei die $A[i]$ die Liste aller Einträge (k,v) mit dem Hashwert $i = h(k)$ ist. (Chaining)

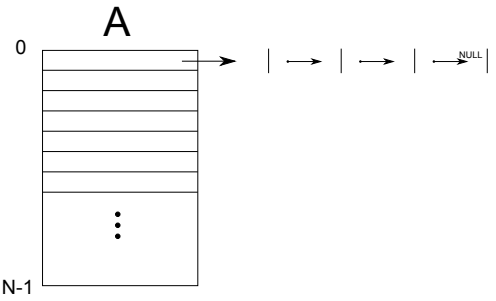


Abbildung 8: *Kombination einer Hashtabelle mit einer verketten Liste*

Die Hashfunktion sollte dabei möglichst gut „streuen“, sodass die Wahrscheinlichkeit, dass $h(k) = i$ für ein „zufälliges“ $k \in S$ möglichst gleich ist für alle i, nämlich $\frac{1}{N}$.
Die Mittlere Länge einer Liste ist in diesem Fall $\lambda = \frac{|D|}{N}$ (λ : Ladefaktor), wenn die Schlüssel als verschieden angenommen werden.
Der Ladefaktor λ ist in der Praxis möglichst klein (< 1, ... 0.9).

5.3.1 Implementierung der Operationen des ADT Wörterbuch

finde(k): berechne $i = h(k)$, finde in der Liste $A[i]$

einfuege(k,v): berechne $i = h(k)$, füge in Liste $A[i]$ ein

streiche(e): berechne $i = h(k)$, streiche in Liste $A[i]$

5.3.2 Laufzeit

Wir nehmen an, dass die Berechnung von h $O(1)$ Zeit koste:

finde():

$O(1) + O(\text{Länge der Liste}) = O(n)$ im schlechtesten Fall
$O(1 + \lambda)$ im Mittel
$O(1)$ falls λ konstant gehalten wird

einfuege():

$O(1)$ in jedem Fall, wenn nicht überprüft werden soll, ob (k,v) bereits vorhanden ist
sonst: $O(1)$ im schlechtesten Fall und $O(1 + \lambda)$ im Mittel

streiche():

$\Theta(n)$ im schlechtesten Fall
$O(1 + \lambda)$ im Mittel

5.3.3 Rehashing

Durch Einfügen wird λ größer. Wenn man eine bestimmte Schwelle σ (z.B. $\sigma = 0.9$) nicht überschreiten will, kann man folgende Technik anwenden:

Falls $\lambda > \sigma$ wird, verdoppele die Größe der Hashtabelle: $N \to 2N$. Dafür ist jedoch eine neue Hashfunktion h' notwendig:

$$h' : S \to \{0, ..., 2N - 1\}$$

Alle Elemente $\in D$ müssen dann neu bzgl. h' eingefügt werden, was im Mittel $O(n)$ Zeit kostet, da n Einfüge-Operationen stattfinden müssen. Dafür können aber n weitere Einfüge-Operationen getätigt werden, bis wieder ein Rehashing erforderlich ist.

also: Die eine Einfüge-Operation, bei der die Schwelle überschritten wird, ist zwar teuer ($\Theta(n)$ im Mittel), jedoch sind im Anschluss wieder n „billige“ Einfüge-Operationen möglich (amortisierte Laufzeit $O(1)$), trotz Rehashing.

5.4 Hashfunktionen

2 Stufen:

$$h_1 : S \to \mathbb{N}$$
$$h_2 : \mathbb{N} \to \{0, ..., N-1\}$$
$$\text{dann: } h = h_2 \circ h_1$$

Für h_2 meist: $h_2 = n \ mod \ N$

In Java existiert die Funktion *int hashCode()*, die in der Klasse 'Object' vorhanden ist. Bei der Java-Implementierung wird meist die Adresse verwendet, bei der sich das Objekt befindet, was nicht besonders gut ist. Aus diesem Grund sollte diese Funktion überschrieben werden.

kleine Beispiele für Hashfunktionen: Primitive Datentypen wie byte, char und short kann man nach int casten. Dies kann man auch für long und double tun, was jedoch nicht so gut ist. Aber auch allgemeine Bitfolgen kann man in 32-Bit Stücke $x_0, ..., x_{n-1}$ zerschneiden, wobei jedes Stück wiederum als int interpretiert und dann aufaddiert werden. Jedoch führen dann Namen wie 0x01 oder 0x10 unweigerlich zu Kollisionen. Um dies zu vermeiden kann man einen polynomiellen Hashcode verwenden.

5.4.1 Polynomieller Hashcode

$$h_1(x_0, ..., x_{n-1}) = x_0 a^{n-1} + x_1 a^{n-2} + ... + x_{n-2} a + x_{n-1}$$

a kann hierbei irgendeine Zahl sein, z.B. $a = 37$. Also das Polynom sollte möglichst an der Stelle a ausgewertet werden. Bei dieser Alternative sollte a möglichst kein Teiler von N und auch mit N teilerfremd sein. Im Idealfall wählt man a und N jeweils als Primzahlen.

5.5 Kollisionsbehandlung

Bisher hatten wir „Chaining“ betrachtet, bei der verkettete Listen aus kollidierenden Elementen verwendet werden. Es gibt aber auch noch die offene Adressierung, bei der die Schlüssel direkt in die Hashtabelle geschrieben werden.

5.5.1 Lineares Sondieren

Schlüssel x mit $h(x)$ als zugehörige Stelle in der Hashtabelle. Falls die Stelle besetzt ist (jedoch nicht mit x), probiere bis x oder eine freie Stelle gefunden wurde:

$$\left.\begin{array}{l} h(x) + c \\ h(x) + 2c \\ ... \end{array}\right\} \text{mod N}$$

wobei $c \in \mathbb{N}$ konstant.

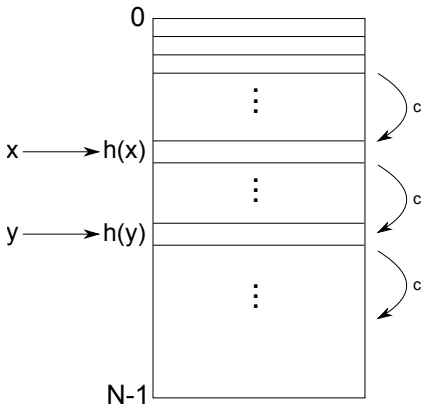


Abbildung 9: *Bsp: Sondierung nach x und y; Überlappung beider Sondierungsketten, wenn verschiedene Schlüssel in diese Kette abgebildet werden.*

besser: quadratisches Sondieren

5.5.2 Quadratisches Sondieren

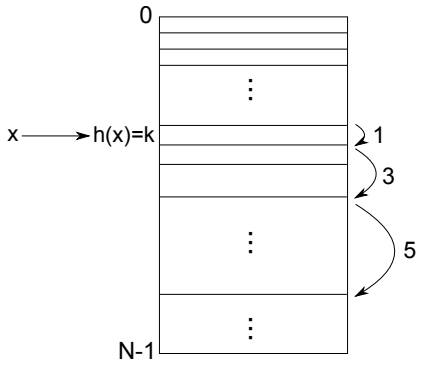


Abbildung 10: *Schematische Darstellung der quadratischen Sondierung*

Teste Positionen:

$$
\begin{aligned}
&k \\
&k + j_1 \\
&k + j_1 + j_2 \\
&k + j_1 + j_2 + j_3 \\
&\ldots
\end{aligned}
$$

wobei $j_{i+1} = j_i + c$ und c und j_1 konstant sind.

z.B.:

$$\left.\begin{array}{l} j_1 = 1 \\ c = 2 \end{array}\right\} i\text{-ter Schritt } k + i^2 \text{ ausprobieren}$$

5.6 Skip-Listen

Als weitere mögliche Datenstrukturen für Wörterbücher können Skip-Listen herangezogen werden. Der Einfachheit halber nehmen wir an, dass alle Schlüssel verschieden sind. Weiterhin gehen wir von einer linearen Ordnung auf der Menge S der Schlüssel aus.

Bsp:

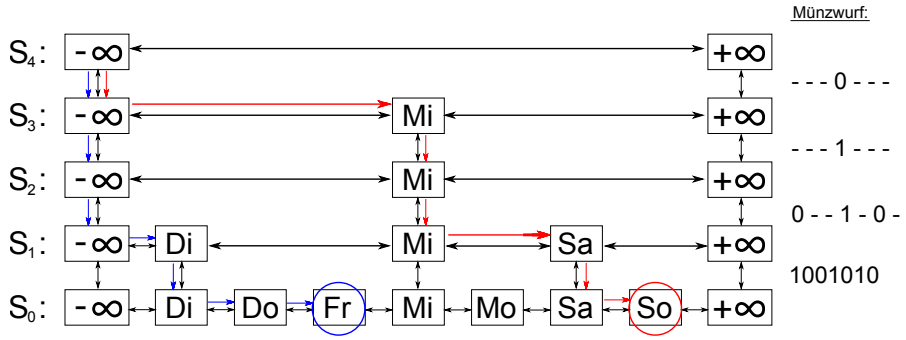


Abbildung 11: *S_0 ist die Ausgangsliste. Zufällig wird für jedes Element entschieden, ob es in die Liste auf der nächsten Ebene kopiert wird. Schematisch dargestellt ist jeweils die Suche nach 'Fr' (blauer Pfad) und 'So' (roter Pfad). Bei der Suche nach einem Element geht man nach unten, wenn der rechte Nachbar größer ist als das gesuchte Element, sonst nach rechts.*

allgemeine Bescheibung: Wir haben eine verkette Liste S_0, in der die ganze Datenmenge D aufsteigend sortiert ist. Zusätzlich führen wir die beiden Randelemente $+\infty$ und $-\infty$ ein. Es werden weitere Listen S_i erzeugt, wobei S_{i+1} aus S_i entsteht, indem für jedes Element per Zufall (Wahrscheinlichkeit $\frac{1}{2}$) entschieden wird, ob es übernommen wird oder nicht. Ausserdem gehen von jedem Element jeweils ein Zeiger zum oberen, unteren, linken und rechten Nachbarn ab (siehe Abb. 11).

5.6.1 Operationen des Wörterbuchs

finde(k): $p :=$ linkestes Element in oberster Liste

```
while (unter(p) != null) do
    p := unter(p)
    while (k >= Schlüssel in rechts(p)) do
        p := rechts(p)
return p
```

Anmerkung: Der obige Algorithmus funktioniert nur, wenn das gesuchte Element auch wirklich vorhanden ist.

einfuege(k,v): Die Ausführung von *finde(k)* liefert die Stelle, an der k hingehört. Es wird dort eingefügt jeweils nacheinander für die darüberliegenden Listen zufällig entschieden, ob es auch dort eingefügt wird, bis der Zufallswert bspw. 0 oder 'falsch' liefert. War der Zufallswert m-mal 1 oder 'wahr/richtig', muss auch die Höhe des Stapels (S_0 ... S_m) folglich m sein. Ist m größer als die bisherige Höhe, muss die Struktur entsprechend vergrößert werden. Damit die Konsistenz gewahrt wird, müssen zusätzlich die Pfeile von jedem Element adäquat verändert werden.

streiche(k): Auch hier wird *finde(k)* ausgeführt, bis man k in der untersten Liste S_0 gefunden hat. Dieses und alle darüberliegenden werden gestrichen. Analog zu *einfuege()* müssen auch hier die Zeiger zwischen den Nachbarn wieder richtig gesetzt werden.

Erwartete Höhe: Im Modell haben wir zu Beginn eine leere Struktur, in der wir dann n Elemente einfügen. Die Wahrscheinlichkeit, dass ein (fester) Stapel eine Höhe $\geq i$ hat, ist $\frac{1}{2^i}$. Das ist somit die Wahrscheinlichkeit, dass bei wiederholtem Müzwurf i-mal eine 1 kommt und dann 0 (bzw. irgendetwas ungleich 1).
Die Wahrscheinlichkeit, dass mindestens einer der Stapel eine Höhe $\geq i$ hat ist somit $\leq \frac{n}{2^i}$. Das heißt, dass die Höhe der Struktur $\geq i$ ist.

Betrachten wir bspw. die Wahrscheinlichkeit, dass die Höhe $\geq 3 \cdot log(n)$ ist. Dann folgt für die Höhe selbst:

$$P \leq \frac{n}{2^{3 \cdot log(n)}} = \frac{1}{n^2}$$
$$\text{bzw. } P \leq \frac{1}{n^{c-1}}, \text{ wenn } h \geq c \cdot log(n)$$

Mit „hoher Wahrscheinlichkeit“ist also:

$$\text{Höhe } \leq c \cdot log(n)$$

Erwarteter Platzbedarf:

auf unterster Ebene: n Kästchen

auf 2-unterster Ebene: $\frac{n}{2}$ Kästchen

$\vdots$

Insgesamt:

$$\sum_{i=0}^{\infty} \frac{n}{2^i} = 2n \text{ also } O(n)$$

Erwartete Laufzeiten:

für „finde()“: Die Anzahl der vertikalen Links, die traversiert werden ist h, wobei $h =$ Höhe der Skip-Liste. n ist hingegen die Anzahl der Links, die horizontal traversiert werden auf der i-ten Ebene. Ein solcher Link kann dabei nur genommen werden, wenn die Situation eintritt, dass das Element in der $i+1$-ten Ebene, auf das der Link zeigt, nicht existiert (siehe Abb. 12). Die Wahrscheinlichkeit dafür beträgt $\frac{1}{2}$, also Erwartungswert von n_i

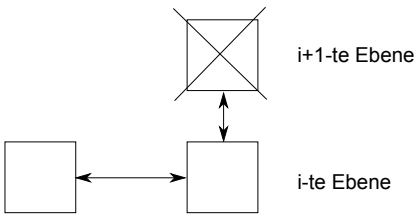


Abbildung 12

$$
\begin{aligned}
&\overbrace{\leq \frac{1}{2}}^{Warsch.,dass} \overbrace{\cdot 0}^{n_i=0} + \underbrace{\frac{1}{4} \cdot 1}_{\substack{\text{1. Link nehmen} \\ \text{2. Link nicht nehmen}}} + \frac{1}{8} \cdot 2 + ... \\
&\leq \sum_{j=0}^{\infty} \frac{j}{2^{j+1}} \\
&= \frac{1}{4} \cdot \underbrace{\sum_{j=1}^{\infty} \frac{j}{2^{j-1}}}_{[\sum_{j=1}^{\infty} \underbrace{j \cdot x^{j-1}}_{\frac{d}{dx} x^j}]_{x=\frac{1}{2}} = \frac{d}{dx} \sum_{j=1}^{\infty} x^j = \frac{d}{dx}(\frac{1}{1-x} - 1)} = \frac{1}{(1-x)^2}
\end{aligned}
$$

Mit $x = \frac{1}{2}$ folgt also für den Erwartungswert von n_i:

$$E(n_i) \leq \frac{1}{4} \cdot 4 = 1$$

Zusammengefasst ist der erwartete Wert der Anzahl der genommen horizontalen Links auch h. Daraus folgt für die Laufzeit für *finde()*, dass diese $O(h)$ ist.

Worst-Case-Laufzeit für *finde()*: $O(\underbrace{n}_{\substack{\text{vertikale} \\ \text{Links}}} + \underbrace{h}_{\substack{\text{horizontale} \\ \text{Links}}})$

streiche(): Bei dieser Operation muss das entsprechende Element erst gefunden werden (O(h) bzw. O(h+n)). Anschließend muss man den Stapel nach oben laufen, um die darüberliegenden Elemente zu löschen (O(h)). Insgesamt hat man hier also eine erwartete Laufzeit von O(h) und eine Worst-Case-Laufzeit von O(h+n).

einfuege(): Analog zu *streiche()* muss auch hier die richtige Position erst gefunden werden (O(h) bzw. O(h+n)). Für das einzufügende Element muss man dann einen Stapel aufbauen, was im Erwartungsfall O(1) (genauer: $O(\sum i \cdot \frac{1}{2^i})$) dauert und im schlimmsten Fall ∞ lang.

5.6.2 Zusammenfassung

Skip-Lists Wörterbuch der Länge n

Laufzeiten	erwartet	Worst Case
finde()	h	h+n
einfuege()	h	∞
streiche()	h	h+n

Der Platzbedarf ist im Erwartungsfall $O(n)$. Im schlimmsten Fall kann dieser jedoch ∞ groß werden, wenn ständig neue Stapel erzeugt werden, wobei für die Höhe h der Struktur gilt:

$$h = \begin{cases} O(log(n)) \text{ erwartet} \\ \infty \text{ worst case} \end{cases}$$

6 Suchbäume

Suchbäume sind die meist untersuchte Datenstruktur für das Wörterbuchproblem.

Annahme:

- Universum S der Schlüssel besitzt lineare Ordnung $\leq$
- (alle Schlüssel in D sind verschieden)

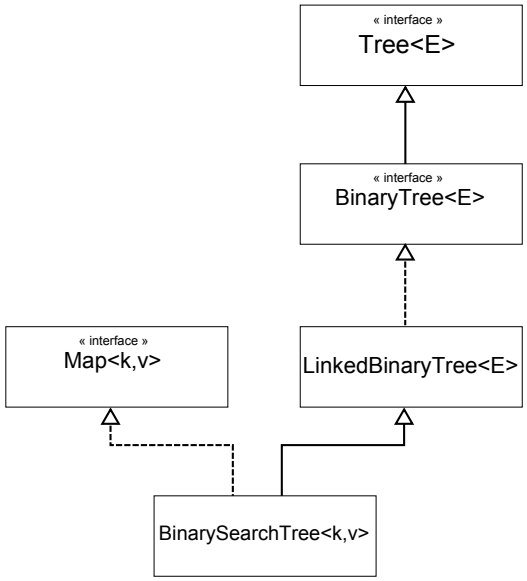


Abbildung 13: *Klassenhierarchie von verschiedenen Baumstrukturen aus Goodrich/Tamassia, angelehnt an UML-Notation.*

6.1 Binärer Suchbaum

Def: Ein binärer Suchbaum ist ein binärer Baum, dessen innere Knoten mit Schlüsseln einer Menge D ($D \subseteq S$, S Universum der Schlüssel mit linearer Ordnung $\leq$) markiert sind, wobei für jeden inneren Knoten V gilt, dass die Schlüssel im linken Teilbaum von V kleiner gleich dem Schlüssel in V sind, der wiederum kleiner als die Schlüssel im rechten Teilbaum ist.

Bsp:

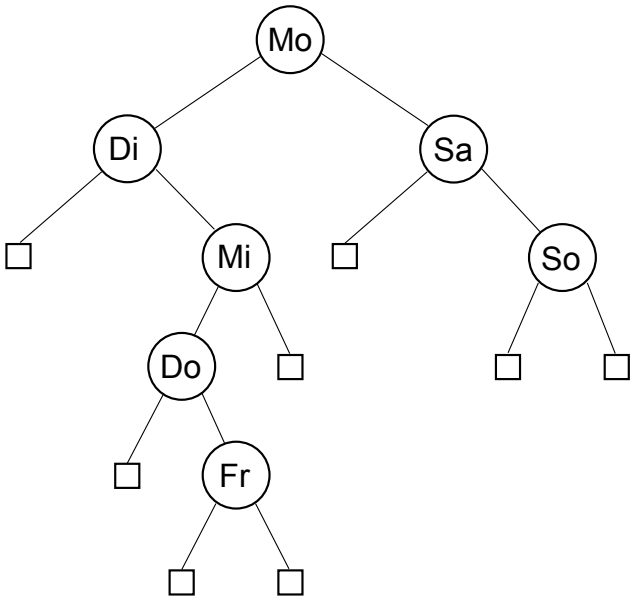


Abbildung 14: *Wochentage lexikografisch in einem binären Suchbaum angeordnet. Die Blätter sind konzeptuell vorhanden und dienen als Indikator für eine erfolglose Suche nach einem bestimmten Element. In einer konkreten Implementierung können si durch NULL-Pointer realisiert werden.*

6.1.1 Operationen des Wörterbuchs

Anmerkung: Der Einfachheit halber gehen wir davon aus, dass alle Schlüssel verschieden sind.

finde(k): Falls die Wurzel ein Blatt ist, gib zurück: „nicht gefunden“. Sonst: vergleiche k mit Schlüssel a der Wurzel:

- falls $k = a$: „gefunden“
- falls $k < a$: durchsuche rekursiv linken Teilbaum
- sonst: durchsuche rekursiv rechten Teilbaum

Laufzeit:

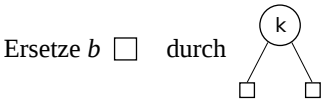
$$T(h) \leq T(h-1) + C$$
$$T(0) = d$$

wobei $T(h)$ die Laufzeit für einen Baum der Höhe h, $T(h-1)$ die Laufzeit für den rekursiven Aufruf mit einem Baum der Höhe $\leq h$ und C die Anzahl der Vergleiche mit dem Element in der jeweiligen Wurzel ist.

$$\begin{aligned} T(h) &\leq T(h-1) + C \\ &\leq T(h-2) + C + C \\ &\quad \vdots \\ &\leq \underbrace{T(0)}_{=d} + h \cdot C \\ &= O(h) \end{aligned}$$

einfuege(k,v): Führe *finde(k)* durch bis zu einem Blatt b:

falls "=": im linken Teilbaum weiterlaufen.



Bsp:

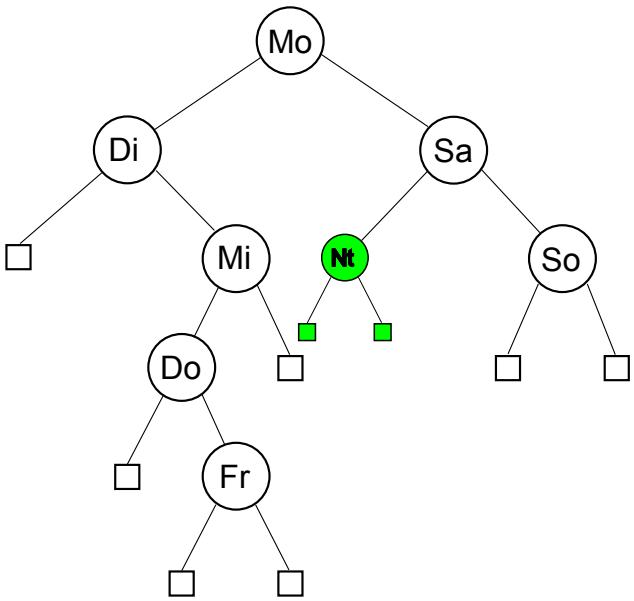


Abbildung 15: *Einfügen eines fiktiven Tages 'Nt'.*

Laufzeit: Zuerst muss die Zeit aufgebracht werden, um die richtige Stelle zum Einfügen zu finden (O(h)). Das Einfügen an sich erfordert konstante Zeit. Insgesamt folgt also für die Laufzeit, dass diese O(h) ist.

streiche(k,v): Durchführung von *finde(k)* liefert Knoten v:

- falls v ein Blatt ist: k nicht vorhanden $\rightarrow$ fertig
- falls linkes Kind von v ein Blatt ist: ersetze v durch sein rechtes Kind
- sonst: finde größtes Element w im linken Teilbaum von v durch das Verfolgen von „rechts-Zeigern“, so lange wie möglich: der letzte innere Knoten ist w. Ersetze Schlüssel von v durch den in w und ersetzen w durch sein linkes Kind.

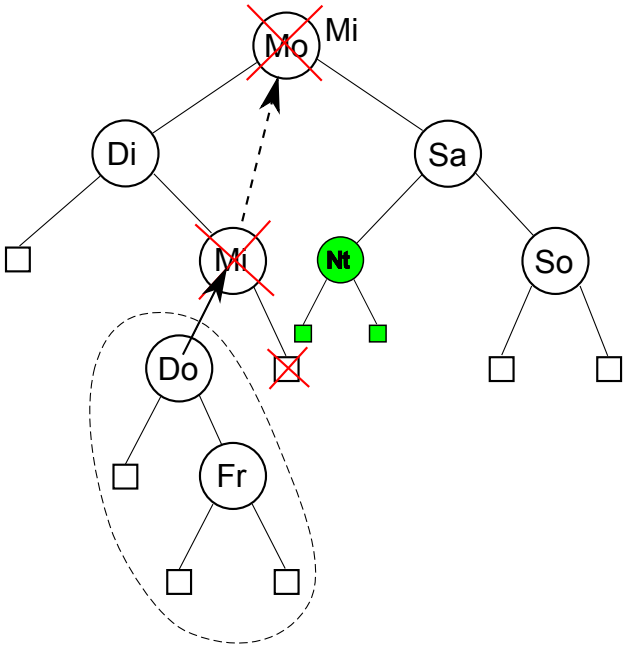


Abbildung 16: *Löschen des Knotens 'Mo'.*

Laufzeit: Für *finde(k)* wird wieder $O(h)$ Zeit benötigt. Zusätzlich konstante Zeit, wenn v oder das linke Kind von v ein Blatt ist. Sonst benötigt man zusätzlich $O(h)$ Zeit, um w zu finden. Alle weiteren Schritte benötigen auch nur $O(1)$ Zeit. Insgesamt hat *streiche(k,v)* also eine Laufzeit von $O(h)$.

Fazit: Wörterbuchoperationen auf einem binären Suchbaum der Höhe h sind in $O(h)$ Zeit durchzuführen, wobei h von der Anzahl der Schlüssel abhängig sein kann:

$$\text{mindestens: } h = \lfloor log(n) \rfloor + 1$$
$$\text{höchstens: } h = n$$

$$\Rightarrow \lfloor log(n) \rfloor + 1 \leq h \leq n$$

wobei n die Anzahl der Schlüssel ist.

Daraus ergibt sich für die Laufzeiten der Operationen des Wörterbuchproblems:

$$\begin{aligned} \text{gut: } & \Theta(log(n)) \\ \text{schlecht: } & \Theta(n) \end{aligned}$$

Der schlechte Fall tritt bspw. auf, wenn die einzuordnende Folge bereits sortiert ist. Man kann jedoch zeigen, dass falls im ursprünglich leeren Baum Elemente eingefügt werden (wobei Permutationen von aufsteigenden Reihenfolgen „zufällig“ sind), man eine erwartete Höhe von $\Theta(log(n))$ erhält.

Frage: Wie kann man den schlechtesten Fall verbessern? → **AVL-Bäume**

6.2 AVL-Bäume (Adelson-Welski und Landis)

Def: Ein binärer Suchbaum heißt AVL-Baum, wenn für jeden Knoten gilt, dass die sich Höhen des rechten und linken Teilbaums von v um höchstens 1 unterscheiden.

Bsp.:

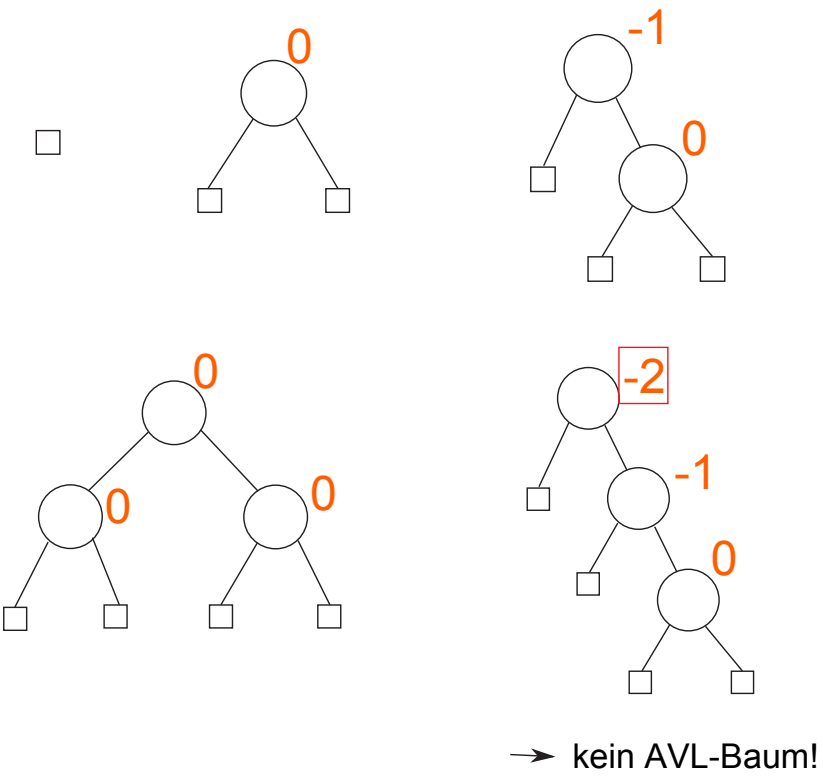


Abbildung 17: *Minimale AVL-Bäume (mit Ausnahme unten rechts). Die Zahlen geben jeweils die Differenz zwischen der Höhe des linken und rechten Unterbaums an. Bei AVL-Bäumen darf diese Differenz nur* 0 *oder* -1 *sein.*

6.2.1 Abschätzung der Höhe eines AVL-Baums

$$h \geq \lfloor log(n) \rfloor + 1 \text{ wobei } n \text{ die Anzahl der inneren Knoten ist}$$

obere Schranke für h: Sei n_h die Mindestanzahl der inneren Knoten eines AVL-Baums der Höhe h. Es gilt:

$$\begin{aligned} n_0 &= 0 \\ n_1 &= 1 \\ &\vdots \\ n_h &= n_{h-1} + n_{h-2} + 1 \end{aligned}$$

da ein Teilbaum Höhe $h-1$ (mit n_{h-1} inneren Knoten) und der jeweils andere Teilbaum Höhe $h-2$ (mit n_{h-2} inneren Knoten) haben muss, damit die Balance erhalten bleibt (siehe Abb. 18).

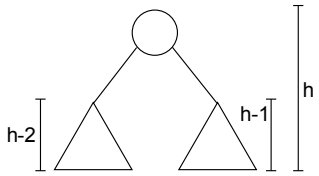


Abbildung 18

h	n_h	fib_h
0	0	0
1	1	1
2	2	1
3	4	2
4	7	3
5	12	5
6	20	8
7	33	13
	$\vdots$	

Daraus folgt für Anzahl der inneren Knoten n_h:

$$n_h = fib_{h+2} - 1 \text{ wobei } fib_m \text{ die } m\text{-te Fibonacci-Zahl ist}$$

$$fib_h = \frac{1}{\sqrt{5}}\Big[\underbrace{\big(\frac{1+\sqrt{5}}{2}\big)^h}_{\Phi=1.63} - \big(\frac{1-\sqrt{5}}{2}\big)^h\Big] \qquad \text{„goldener Schnitt“}$$

also:

$$fib_h = \Omega(\Phi^h)$$

also auch:

$$n_h = fib_{h+2} - 1 \geq fib_h$$
$$n_h = \Omega(\Phi^h)$$

das heißt:

$$n_h \geq c \cdot \Phi^h \text{ für eine Konstante } c > 0$$
$$log(n_h) \geq h \cdot log(\Phi) + log(c)$$
$$h \leq \underbrace{\frac{1}{log(\Phi)}}_{1.44} \cdot log(n_h) - \frac{log(c)}{log(\Phi)}$$
$$h \leq 1.45 \cdot log(n) \text{ für einen AVL-Baum mit } n \text{ inneren Knoten.}$$

6.2.2 Rotation und Doppelrotation

Angenommen, dass die AVL-Eigenschaft beim Einfügen/Streichen verloren geht und 1 Knoten von unten außer Balance ist, sodass der rechte Teilbaum bspw. zu hoch ist.

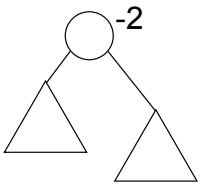


Abbildung 19: *Kann nicht -3 oder noch kleiner sein, wenn vor dem Einfügen die Eigenschaften für einen AVL-Baum noch gegeben waren.*

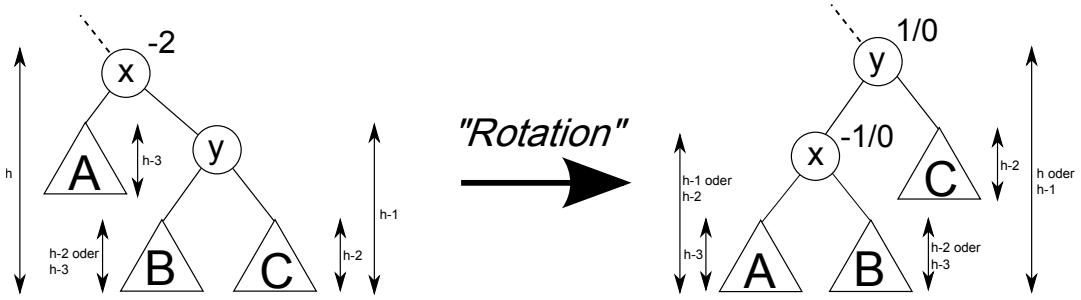


Abbildung 20: *Nach der Rotation bleibt Eigenschaft für einen binären Suchbaum erhalten und der Unterbaum ist wieder ein AVL-Baum.*

Fall 1:

C zu hoch

Der Vaterknoten kann durch diese Operation außer Balance geraten, wenn sich die Höhe um 1 verringert. Dann muss man den Vaterknoten eventuell so lange in Balance bringen, bis man bei der Wurzel angekommen ist.

Fall 2:

B zu hoch → Höhe h-2

C nicht zu hoch → Höhe h-3

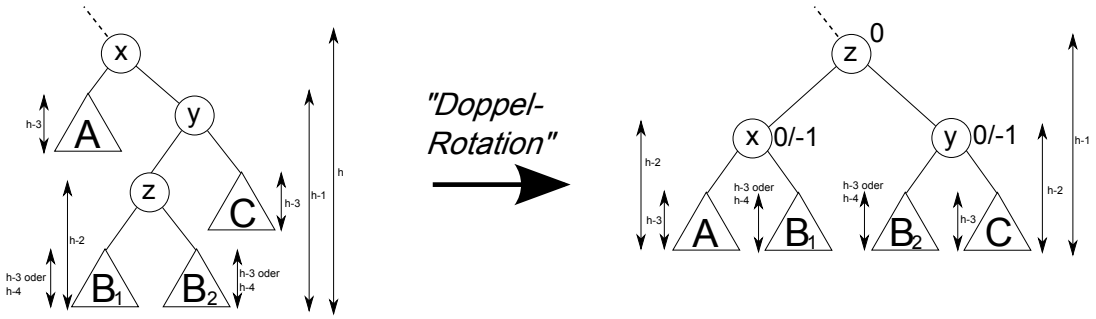


Abbildung 21: *Nach der Doppel-Rotation bleibt die Eigenschaft für einen binären Suchbaum weiterhin erfüllt, die Höhe ist um 1 verringert und die AVL-Eigenschaften sind zumindest für diesen Teilbaum wiederhergestellt. Es kann jedoch passieren, dass der Vaterknoten dann aus der Balance gerät. In diesem Fall muss dann nochmals rotiert werden.*

Also: Falls nach Einfügen oder Streichen die AVL-Eigenschaft verloren geht, kann sie durch eine Folge von Rotationen bzw. Doppelrotationen, ausgehend von der Stelle, an der eingefügt oder gestrichen wurde, entlang eines Pfades bis eventuell zur Wurzel, wiederhergestellt werden. Eine (Doppel-)Rotation kostet lediglich $O(1)$ Zeit, da nur wenige Zeiger umgesetzt werden müssen. Insgesamt kann das Rebalancieren $O(log(n))$ Zeit kosten, wenn man diese Operationen bis zur Wurzel wiederholen muss.

Fazit: Also für die Wörterbuchoperationen auf AVL-Bäumen *finde()*, *einfuege()* und *streiche()* benötigt man somit nur $\underline{O(log(n))}$ Zeit.

6.3 (a,b)-Bäume

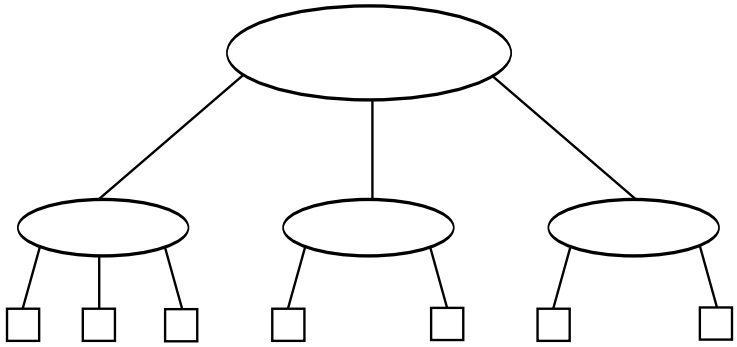
(a,b)-Bäume sind im Allgemeinen keine Binärbäume und a und b beschreiben natürliche Zahlen, die größer sind als 2, wobei $b \geq 2a - 1$.[7]

Def: Ein Baum T heißt (a,b)-Baum, wenn gilt:

1. alle Blätter haben die gleiche Tiefe
2. für alle Knoten v ist die Anzahl der Kinder $\rho(v) \leq b$
3. für alle inneren Knoten v (ausser der Wurzel) ist $\rho(v) \geq a$
4. für die Wurzel r ist $\rho(r) \geq 2$

[7] in der Praxis haben a und b eine Größenordnung von 10^2

z.B.: (2,3)-Baum mit 7 Blättern



Die Höhe h eines (a,b)-Baums mit n Blättern ist höchstens (siehe Abb. 22):

$$
\begin{aligned}
2a^{h-1} &= n \\
a^{h-1} &= \frac{n}{2} \\
(h-1) \cdot log(a) &= log(n) - 1 \\
h &= \underbrace{\frac{1}{log(a)} \cdot log(n)}_{= log_a(n)} - \frac{1}{log(a)} + 1 \\
h &= \Theta(log(n))
\end{aligned}
$$

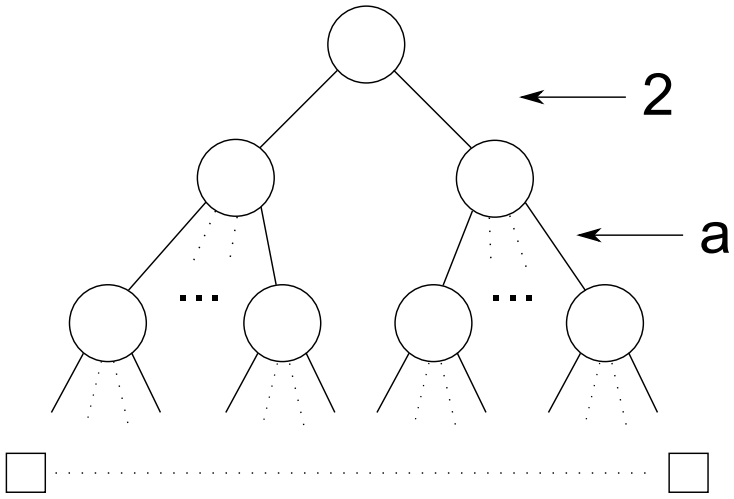


Abbildung 22: *Maximale Höhe eines (a,b)-Baums.*

Auf der anderen Seite hat ein (a,b)-Baum eine Mindesthöhe von (siehe Abb. 23):

$$
\begin{aligned}
b^h &= n \\
h \cdot log(b) &= log(n) \\
h &= \underbrace{\frac{1}{log(b)} \cdot log(n)}_{= log_b(n)} \\
h &= \Theta(log(n))
\end{aligned}
$$

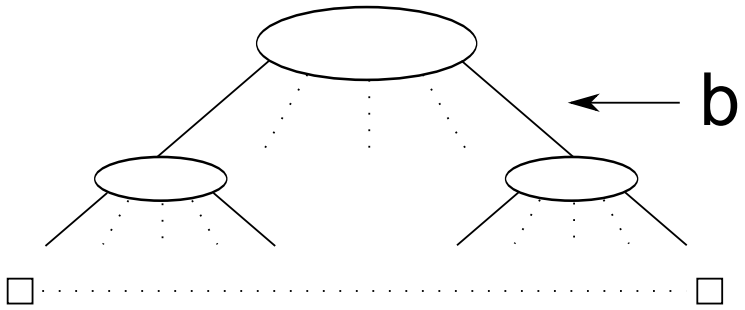


Abbildung 23: *Minimale Höhe eines (a,b)-Baums.*

Schlüssel (einer Menge D) ablegen:

- aufsteigend sortiert in den Blättern von links nach rechts
- innerer Knoten v enthält $m = \rho(v) - 1$ Schlüssel a_1 ... a_m, wobei a_i der max Schlüssel im i-ten Unterbaum ist $(i = 1, ..., m)$

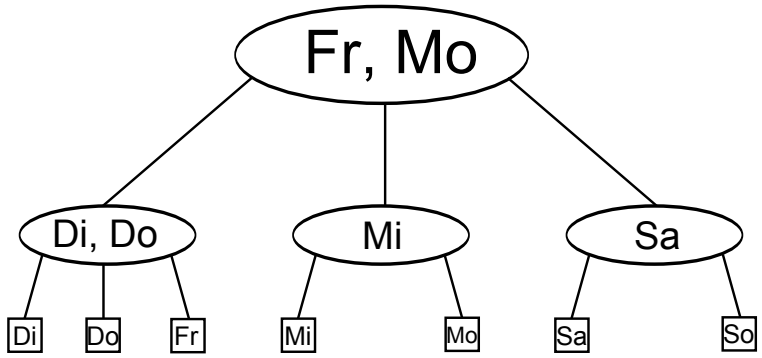


Abbildung 24: *Bsp: Lexikografische Anordnung der Wochentage in einem (2,3)-Baum*

6.3.1 Operationen des Wörterbuchproblems

finde(k): Wir durchsuchen (linear) die Schlüssel a_1 ... a_m der Wurzel (wenn Wurzel kein Blatt ist):

- falls $k \leq a_1$: durchsuche rekursiv 1. Teilbaum
- falls $k > a_m$: durchsuche rekursiv (m+1)-ten Teilbaum
- falls $a_i < k \leq a_{i+1}$, für ein $i \in \{1, ..., m-1\}$: durchsuche rekursiv (i+1)-ten Teilbaum

Ist die Wurzel ein Blatt:

- $k = s$: „gefunden“
- $k \neq s$ „nicht vorhanden“

Laufzeit: $\Theta(h)$ bzw. $\Theta(log(n))$

einfuege(k):

- finde(k): finde Stelle, an der der neue Schlüssel k hingehört
- neues Blatt an entsprechender Stelle, welches an den Vaterknoten angefügt wird, in dem wiederum der dazugehörige Schlüssel eingetragen wird.
  - falls der Vaterknoten immer noch $\leq b$ Kinder hat: fertig
  - sonst (er hat $b+1$ Kinder): spalte den Vaterknoten auf in 2 Knoten mit $\lfloor \frac{b+1}{2} \rfloor$ und $\lceil \frac{b+1}{2} \rceil$ Kindern. Füge diese beiden Knoten in entsprechende Vaterknoten ein und überprüfe diese. Wiedehole den Schritt gegebenfalls bis eventuell zur Wurzel.
  - falls Wurzel aufgespalten wird, schaffe neue Wurzel mit diesen beiden entstandenen Kindern

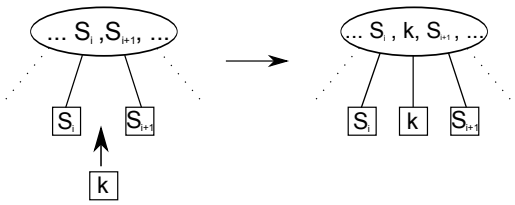


Abbildung 25: *Einfügen eines Schlüssels 'k' zwischen zwei bestehenden Schlüsseln.*

oder

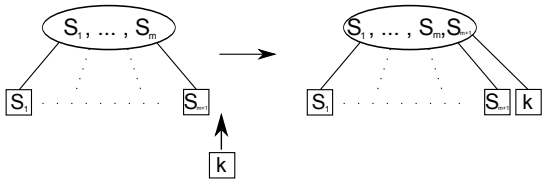


Abbildung 26: *Einfügen eines Schlüssels 'k' als letztes Blatt eines Unterbaums.*

Bsp:

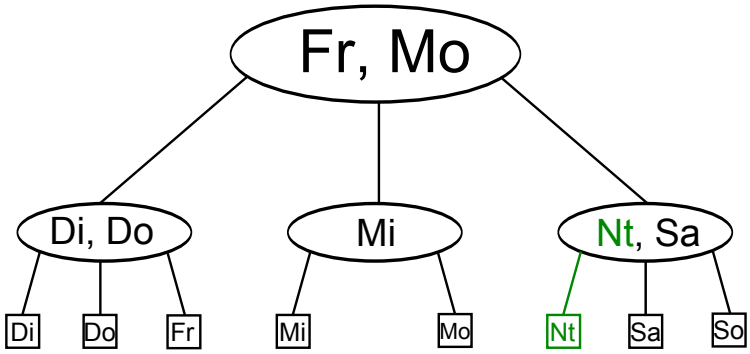


Abbildung 27: *Einfügen eines Tages 'Nt'.*

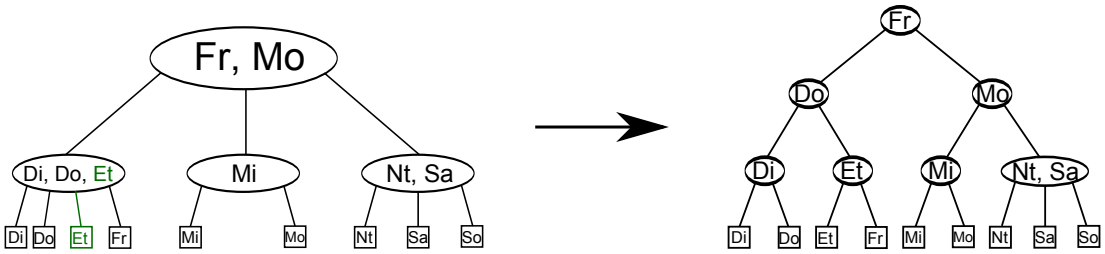


Abbildung 28: *Einfügen eines weiteren Tages 'Et' verletzt zunächst die Eigenschaften eines (a,b)-Baums. Die entsprechenden Elternknoten müssen dann aufgeteilt werden, bis die Eigenschaften wiederhergestellt sind.*

streiche(k):

- Suche nach dem zu streichenden Eintrag liefert ein Blatt
- entferne das gefundene Blatt samt dazugehöriger Schlüssel im Vaterknoten oder in höheren Ebenen des Baums
- falls Vaterknoten v nun nur noch $a-1$ Kinder hat:
  - falls eines der zu v benachbarten Geschwister $w \geq a+1$ Kinder hat: v adoptiert entsprechendes kind von w
  - sonst (v hat $a-1$, Geschwister w hat a Kinder): vereinige v und w zu einem Knoten v'. Dieser neue Knoten hat dann $2a-1 \leq b$ Kinder. Statt v, w wird dann v' in entsprechenden Vaterknoten u eingefügt. Wiederhole die Schritte, falls u nun $a-1$ Kinder hat, bis eventuell zur Wurzel.
  - falls Wurzel r dann nur noch ein Kind hat s hat, entferne r und mache s zur neuen Wurzel.

Laufzeit: *finde()*-Operation läuft von der Wurzel bis eventuell zu einem Blatt. Etwaige Spalt- und Schmelzoperationen auf dem Weg von einem Blatt bis eventuell zur Wurzel. Wenn jede Operation konstante Zeit benötigt, folgt insgesamt für die Laufzeit $O(h) = O(log(n))$.

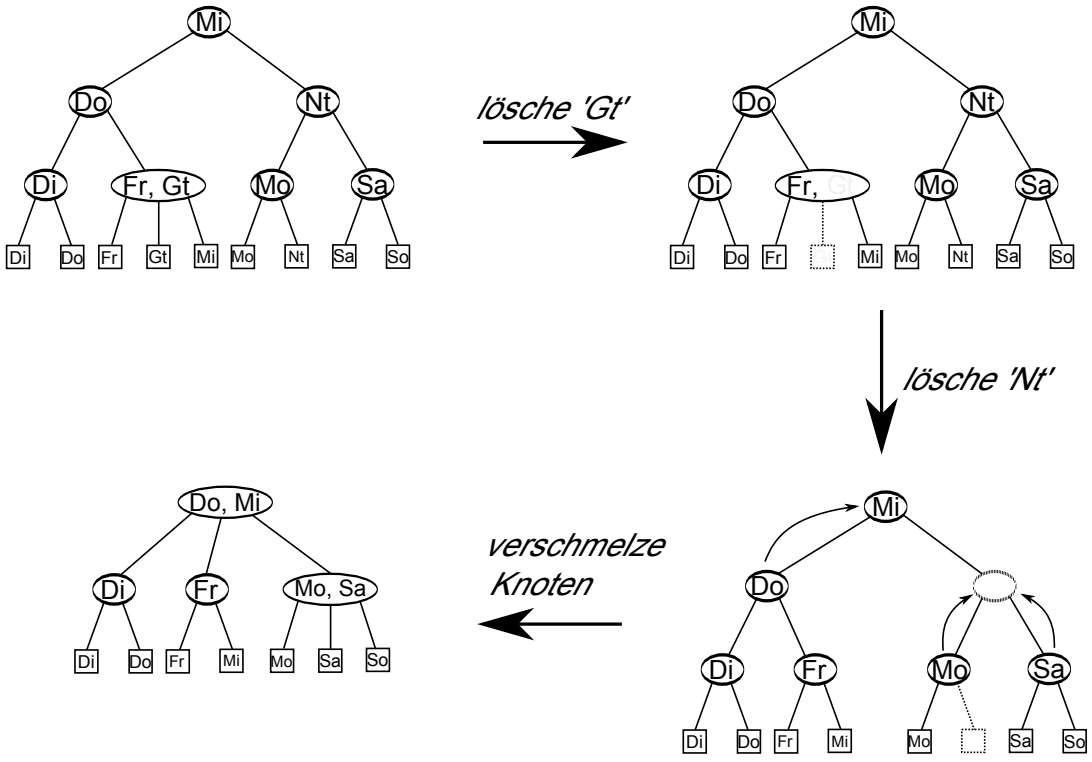


Abbildung 29: *'Gt' kann problemlos gelöscht werden, indem lediglich das entsprechende Blatt und alle Schlüssel in den jeweiligen Vaterknoten entfernt werden. Löscht man dann 'Nt', sind zunächst die Eigenschaften des (a,b)-Baums verletzt. Diese können jedoch wiederhergestellt werden, wenn die kritischen Knoten mit Geschwisterknoten verschmolzen werden.*

(a,b)-Bäume als Implementierung des Wörterbuchs mit $O(log(n))$ Zeit für Operationen bilden eine gute Alternative zu AVL-Bäumen (z.B. (2,3)-Bäume).

wichtige Anwendung: Zugriffe auf Hintergrundespeicher (Festplatten) kosten Hunderte bis Tausende mal soviel Zeit wie CPU-Operationen. Diese Zugriffe müssen entsprechend möglichst gering gehalten werden. Bei einem Zugriff kann ein ganzer Block gelesen werden. Die Idee besteht nun darin einen (a,b)-Baum zu bauen, sodass die Daten in einem inneren Knoten etwa den Umfang einer Blockgröße haben (a,b $\approx$ paar Hundert).
Die Zahl der Speicherzugriffe entspricht dann der Zahl der durchlaufenen Knoten im (a,b)-Baum, also h. In Datenbanken hat man Größenordnungen von 10^6 Knoten. Bei $log_a(n)$ entspricht das einer Höhe von 3 bis 5.

für $b = 2a - 1$**:**

B*-Bäume

B-Bäume, falls Einträge nur in inneren Knoten sind (Bayer-McCreight 1972)

6.4 Wörterbuch für Wörter (Strings)

6.4.1 TRIE (retrieval)

Bsp:

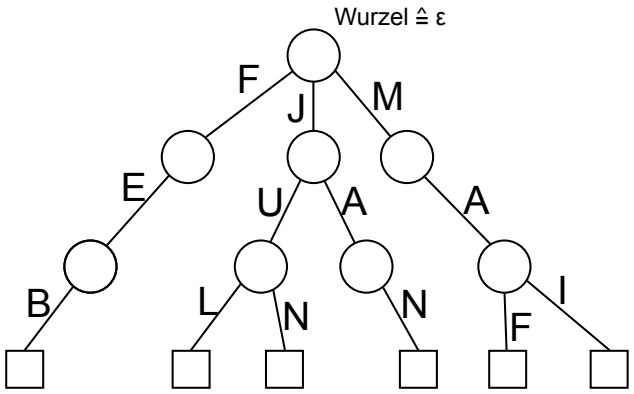


Abbildung 30: *Beispiel für einen Trie bei der einige Monate unter Betrachtung der ersten drei Buchstaben eingetragen sind.*

Def: Ein Trie ist ein Baum, dessen Kanten mit den Zeichen eines Alphabets Σ markiert sind. Weiterhin entspricht die Beschriftung des Weges von der Wurzel zu einem Blatt einem abgespeicherten Wort. Des Weiteren hat jeder Knoten bis zu $d = |\Sigma|$ Kinder, wobei alle verschiedene Markierungen haben.

also: Ein Trie ist eine Implementierung eines Wörterbuchs für eine Menge $D \subseteq \Sigma^*$, wobei Σ^* das Universum der Schlüssel bildet. Damit jedes Wort einem Blatt entspricht darf kein Wort Präfix eines anderen Wortes sein. Dies kann man vermeiden, wenn man hinter jedem Wort eine Endmarkierung (z.B. '$') setzt.

6.4.2 Operationen des Wörterbuchs

finde(w):

- beginne in der Wurzel
- gehe in Teilbaum, der 1. Buchstaben entspricht usw. bis das ganze abgearbeitet wurde

Laufzeit: $O(|w|)$, da jeder Buchstabe von $|w|$ betrachtet werden muss.

einfuege(w):

- beginnen Suche mit w, soweit wie möglich
- Suche endet bei einem inneren Knoten, wobei ein Präfix a_1 ... a_i abgearbeitet worden ist. Von dort schaffen wir einen Weg von a_{i+1} bis $a_{|w|}$

Laufzeit: $O(|w|)$

streiche(w):

- führe Suche nach w durch bis zu einem Blatt
- streiche dieses Blatt und dessen Vaterknoten, falls dieser keine weiteren Kinder hat usw. bis eventuell zur Wurzel

Laufzeit: $O(|w|)$

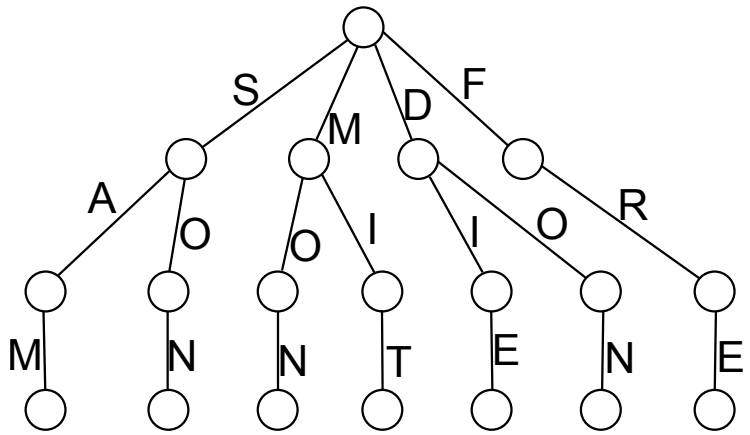


Abbildung 31: *Weiteres Beispiel für einen Trie.*

Anwendung: Wortsuche in einem Text $T = w_1 w_2 ... w_n$ (zwischen den einzelnen w_i können sich Leerzeichen, Satzzeichen etc. befinden).
Wir müssen T so vorverarbeiten, dass die Suche nach einem Wort X später so effizient wie möglich wird. Dazu fügen wir $w_1, w_2, ..., w_n$ in einen zunächst leeren Trie ein. Beim entsprechenden Blatt tragen wir dann jedes mal die „Position“ von w_i in T ein. Am Schluss haben wir an jedem Blatt (bzw. inneren Knoten falls manche Wörter Präfixe von anderen sind) eine Liste von Positionen.
Vorverarbeitungszeit: $O(|T|)$, da $O(|w_i|)$ für jedes Wort w_i, wobei $i = 1, ..., n$.

mit einem Suchwort X: suche im Trie:
entsprechender Knoten liefert Liste der Stellen, an denen X vorkommt, was $O(|X|)$ Zeit kostet.

6.5 Überblick: Datenstrukturen für Wörterbuch

	Suche	Einfüge	Streiche
verk. Liste	n	1	n
sort. Feld	$log(n)$	n	n
Hashing	mittel: 1 worst-case: n	mittel: 1 worst-case: n	mittel: 1 worst-case: n
Skip-Listen	mittel: $log(n)$ worst-case: nicht beschränkt	mittel: $log(n)$ worst-case: ∞	mittel: $log(n)$ worst-case: nicht beschränkt
bin. Suchbaum	mittel: $log(n)$ worst-case: n	mittel: $log(n)$ worst-case: n	mittel: $log(n)$ worst-case: n
AVL-Baum	$log(n)$	$log(n)$	$log(n)$
(a,b)-Baum	$log(n)$	$log(n)$	$log(n)$
Trie	$\|w\|$	$\|w\|$	$\|w\|$

7 Textverarbeitung

Anwendungen: Die Textverabreitung ist an sich ein sehr wichtiges Theme und unter anderem folgende Anwendungsgebiete:

- Textverarbeitungsysteme
- Datenkompression
- Suchmaschinen im Internet
- DNA-Analyse

Grundlegende Definitionen: Wir betrachten ein endliches Alphabet $\Sigma = \{a_1, ..., a_d\}$

z.B.:

$$\Sigma = \{0, 1\} \quad \text{ASCII}$$
$$\Sigma = \{A, ..., Z\} \quad \text{Unicode}$$
$$\Sigma = \{A, C, G, T\} \quad \text{DNA}$$

Σ^* ist Menge der endlichen Folgen über Σ.

Wörter:

$$w = c_1 ... c_k$$

$k = |w|$ Länge von w
$k = 0:$ „leeres Wort“ ε

Konkatenation:

$$w_1, w_2 \mapsto w_1 w_2 \text{ Aneinanderhängungen ('+' in Java)}$$

Falls $w = uvx$, wobei u = Präfix, x = Suffix und v = Teilwort

7.1 lexikografische Ordnung

In Java gibt es dahingehend zwei Klasen:

String → unveränderliche Objekte[8] - unmutable

StringBuffer: veränderliche Objekte - mutable

[8] Felder usw.; „Status“ des Objekts kann nicht verändert werden

Methoden bei *String*:

charAt(i)

substring(i,j)

etc.

Die Methoden dieser Klasse verändern String nicht.

Methoden bei *StringBuffer*:

append(Q)

insort(i,Q)

etc.

In dieser Klasse können die Methoden den String verändern.

7.2 Stringmatching (Patternmatching)

gegeben:

„Text“ $T \in \Sigma^*$

„Muster“ $P \in \Sigma^*$

Frage: Ist P Teilwort von T? (oder finde alle Stellen, an denen P in T auftritt.)

$$\text{Sei } n = |T| \text{ wobei } T = a_0...a_{n-1} \text{ mit } a_i \in \Sigma, i = 0, ..., n-1$$
$$m = |P| \text{ wobei } P = p_0...p_{m-1} \text{ mit } p_i \in \Sigma$$

7.2.1 „Brute-Force-Methode“:

Algorithmus:

- lege P an erster Stelle von T an, vergleiche nacheinander entsprechende Zeichen
- falls bis zum Schluss Übereinstimmungen von P: gib Stelle aus
- falls Mismatch auftritt: nicht ausgeben
- schiebe P um eine Stelle weiter und wiederhole die vorherigen Schritte bis zur Stelle $n-m-1$

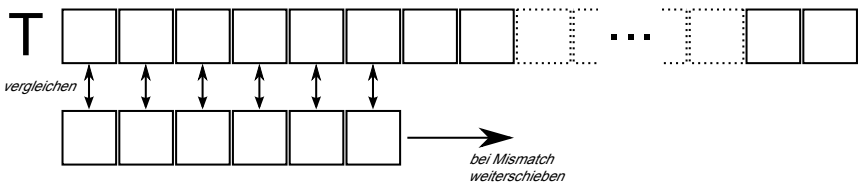


Abbildung 32: *Schematischer Ablauf der Brute-Force-Methode. Das Muster wird sukzessiv mit jedem Teilwort des Textes T zeichenweise verglichen.*

Laufzeit: $O(\underbrace{(n-m-1)}_{\text{Zahl der Durchläufe}} \cdot \overbrace{m}^{\text{für 1 Durchlauf maximal}}) = O(nm)$ für große n (z.B.: $n \geq 2m$)

Wenn man annimmt, dass T ein „zufälliger“ Text ist (d.h.: jedes Zeichen tritt an jeder Stelle mit gleicher Wahrscheinlichkeit auf), hätte man eine erwartete Laufzeit von $O(n)$.

7.2.2 Algorithmus von Rabin-Karp

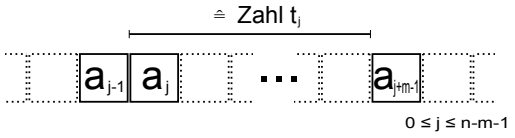
Idee:

1. Interpretieren P als Darstellung einer d-nären Zahl, wobei $d = |\Sigma|$

   $\Sigma = \{c_1, ..., c_d\}$
   $0, \ ... \ , d-1 \leftarrow$ Zahlenwerte zuordnen
   $P = p_0 ... p_{m-1}$ wobei $p = \underbrace{p_0}_{\text{entspr. Zahlenwerte sind gemeint}} \cdot d^{m-1} + p_1 \cdot d^{m-2} + ... + p_{m-1}$

   $$P \mapsto p = \sum_{i=0}^{m-1} p_i \cdot d^{m-i-1}$$

2. wandle genauso jedes Teilwort von T der Länge m um in eine Zahl



   $$t_j = \sum_{i=0}^{m-1} a_{j-i} d^{m-i-1}$$

   und überprüfen für alle t_j, ob $t_j = p$. Wenn ja: „match“ an Stelle j

3. durchlaufe alle t_j und gib j aus, falls $t_j = p$.

Bsp: $\Sigma = \{0, 1\}$

$T = 010\boxed{0110}1110\boxed{0110}00$
$P = 0110$
$t_j \to 4\ 9\ 3\ \boxed{6}\ 13\ 11\ 7\ 14\ 12\ 9\ 3\ \boxed{6}\ 12\ 8 \qquad j = 0, ..., n-m-1$
$p = 6$

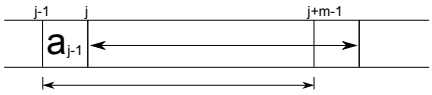
Laufzeit: Für jede Addition und Multiplikation nehmen wir eine konstante Zeit an:

Schritt 1: $O(m)$ Zeit

$$p_{m-1} + p_{m-2}d + p_{m-3}d^2 + ... + p_0 d^{m-1}$$
$$= p_{m-1} + d(p_{m-2} + d(p_{m-3} + ...)) \text{ „Horner-Schema“}$$

Schritt 2: t_0 berechnen wie bei p kostet $O(m)$ Zeit

für $j = 0, ..., n-m-1$:
$t_j := d \cdot t_{j-1} - a_{j-1}d^m + a_{j-m-1}$



Laufzeit für diesen Schritt: $O(\underbrace{n}_{\text{für Schleife, in der jedes } j \text{ durchlaufen wird}} + \overbrace{m}^{\text{für } t_0})$

Laufzeit insgesamt:

Schritt 1: O(m)
Schritt 2: O(n+m)
Schritt 3: O(n)
} O(n+m)

Problem: Die Annahme, dass die Zahlen in konstanter Zeit addiert, subtrahiert, multipliziert bzw. verglichen werden können ist unangebracht, da eine solche Zahl ein Textstück der Länge m kodiert, also $\in \{0, d^m - 1\}$. Stattdessen kann man alle Rechnungen 'mod' q nehmen, wobei q eine Primzahl ist (sodass $0, ..., q-1$ in ein Computerwort passen).

$T = 010\boxed{0110}1110\boxed{0110}00$
$P = 0110$
$t_j \to 4\ 9\ 3\ \boxed{6}\ 13\ 11\ 7\ 14\ 12\ 9\ 3\ \boxed{6}\ 12\ 8 \qquad j = 0, ..., n-m-1$
$p = 6 \to 1$
$\to 4\ 4\ 3\ \boxed{1}\ 3\ \boxed{1}\ 2\ 4\ 2\ 4\ 3\ \boxed{1}\ 2\ 3 \qquad q = 5$

Dadurch entsteht jedoch wiederum das Problem, das man falsche Treffer also Zahlen t_j erhält, die 'mod' q mit p übereinstimmen, aber $\neq p$ sind. $\Rightarrow$ „False-Positives“
Eine Abhilfe besteht darin, dass wann immer das Verfahren einen Treffer liefert, wir dies durch einen Vergleich (zeichenweise) mit p verifizieren müssen. Im schlechtesten Fall erhält man so wieder eine Laufzeit von $\Theta(nm)$.

$$T = a^n$$
$$P = a^m$$

Allgemein:

$$O(n + m + (r + f)m)$$

wobei r die Anzahl der richtigen und f die Anzahl der falschen Treffer ist.
Falls der Text zufällig ist, ist bspw. die Anzahl der falschen Treffer $\leq \frac{n}{q}$, da jedes Teilwort mit einer Wahrscheinlichkeit von $\frac{1}{q}$ auf ein bestimmtes Element $\in \{0, ..., q-1\}$ abgeb. wird. Unter dieser Voraussetzung erhalten wir eine erwartete Laufzeit von $O(n + m + (r + \frac{n}{q})m)$.
Falls $q \geq cm$ gewählt wird:

$$O(n + m + mr + \frac{1}{\cancel{c}}n)$$

Andere Sichtweise: Wir wenden auf Strings der Länge n eine Hash-Funktion an, wobei dann False-Positives Kollisionen darstellen. Eine gute Streuung ist dabei ein Indiz dafür, das der Algorithmus gut funktioniert.

7.2.3 Algorithmus von Boyer-Moore

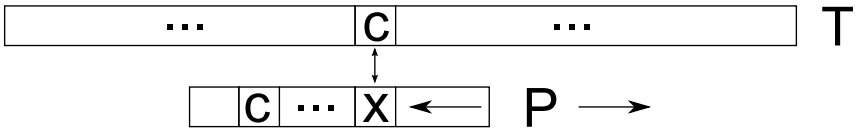


Abbildung 33: *Das Muster P wird wie bei der Brute-Force-Methode von links nach rechts verschoben. Verglichen wird jedoch dann zeichenweise von rechts nach links.*

Dieser Algorithmus verläuft sehr ähnlich zum Brute-Force-Algorithmus. Auch hier wird P von links nach rechts entlang T verschoben, jedoch:

- bei einer Stellung von P wird zeichenweise von rechts nach links anstatt von links nach rechts wie bei Brute-Force verglichen.
- bei einem Mismatch, bei dem c das entsprechende Zeichen in T ist, wird P soweit verschoben, bis ein c in P unter dem c in T liegt, falls möglich. Sonst wird P um nur ein Zeichen weiter nach rechts geschoben. („bad-character-heuristic“)

Bsp:

$$P = g^{(1)}r^{(2)}a^{(3)}m^{(4)}m^{(5)}$$

x	rechts(x)
a	2
g	0
m	4
r	1
alle anderen	-1

Tabelle 1: *rechts(x) entspricht der rechtesten Stelle in P, an der x vorkommt. -1 als Zeichen dafür, dass das Zeichen nicht vorkommt.*

A	l	g	o	r	i	t	h	m	e	n		u	n	d		P	r	o	g	r	a	m	m	i	e	r	u	n	g
g	r	a	m	m																									
			g	r	a	m	m																						
								g	r	a	m	m																	
													g	r	a	m	m												
																g	r	m	m										
																			g	r	a	m	m						
																				g	r	a	m	m					
																									g	r	a	m	m

Die Berechnung der Tabelle rechts(x) kostet $O(m)$ Zeit. Der Algorithmus selbst benötigt $\Theta(nm)$ im schlechtesten Fall, wobei er in der Praxis jedoch viel besser ist.

Der eigentliche Boyer-Moore-Algorithmus benutzt noch eine weitere Heuristik zum Verschieben: „good-suffix-heuristic“

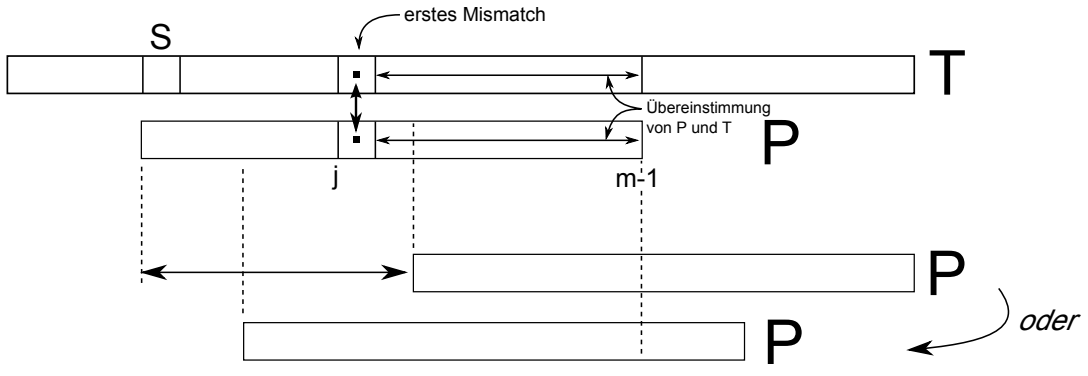


Abbildung 34

Es muss beim nächsten Match gelten, dass:

$$(*)\ P|_{0\ldots k-1} \text{ ist Suffix } P|_{j+1\ldots m} \text{ oder umgekehrt}$$

$$\gamma[j] = m - \ max\{k|0 \leq k < m \ mit \ (*)\}$$

wobei die $\gamma[0], ..., \gamma[m-1]$ in $O(m)$ Zeit berechnet werden können! (ohne Beweis)

Der Boyer-Moore-Algorithmus wendet beide Heuristiken an und nimmt den größeren der beiden Werte zum Verschieben.

In der Praxis ist dies ein sehr guter Algorithmus (auch in vereinfachter Form mit der 'bad-character-heuristic'), aber im schlechtesten Fall ist er immernoch $\Theta(nm)$.

7.2.4 Einschub: Heuristik

Ein Verfahren, das in der Praxis mit sinnvollen Kriterien gut funktioniert, aber eventuell im schlechtesten Fall ineffizient oder sogar nicht korrekt[9] ist.

7.2.5 Suffix-Bäume

Für Stringmatching gibt es Algorithmen mit linearer Laufzeit, also O(n+m) auch im schlechtesten Fall. Der bekannteste von ihnen ist der Algorithmus von „Knuth-Morris-Pratt“(KMP).

Einen weiteren solchen Algorithmus erhält man durch sogenannte Suffix-Bäume. Für ein Text T ist ein Suffix-Baum ein Trie, der alle Suffixe von T enthält.

Bsp: $T = m^{(0)}i^{(1)}s^{(2)}s^{(3)}i^{(4)}s^{(5)}s^{(6)}i^{(7)}p^{(8)}p^{(9)}i^{(10)}\$^{(11)}$

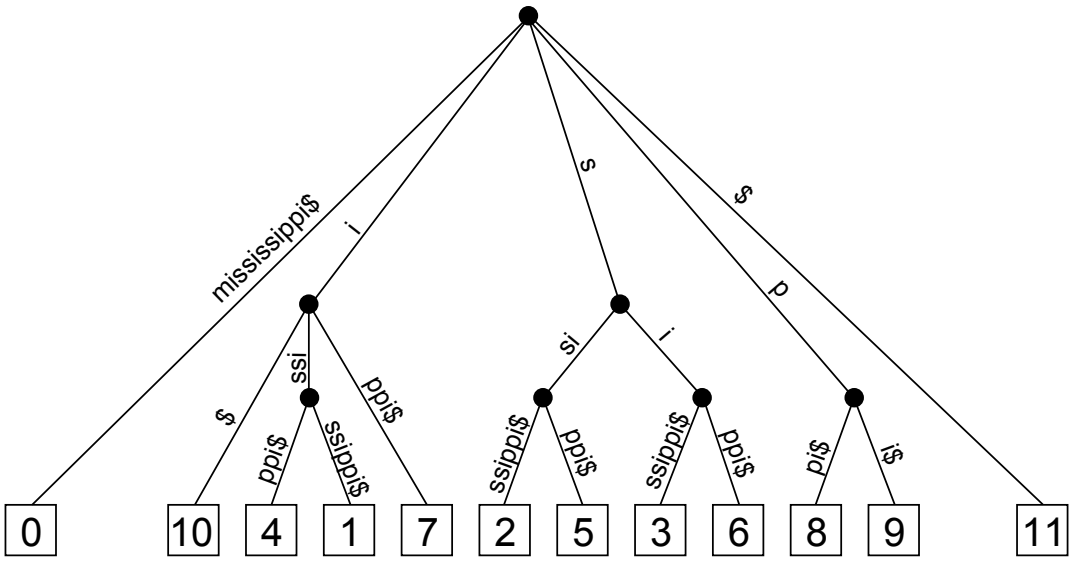


Abbildung 35: *Suffix-Baum für die Zeichenfolge 'mississippi$'. Der Wert in einem Blatt entspricht der Stelle, ab der der jeweilige Suffix beginnt.*

Wenn der Suffix-Baum für T konstruiert ist, erfordert Stringmatching für P $O(m + k)$ Zeit, wobei m die Länge von P und k die Anzahl der Vorkommen von P in T ist.

entsprechender Algorithmus:

1. Suche nach P im komprimierten Suffix-Baum
2. bleibt Suche „stecken“: P nicht vorhanden
3. sonst: Suche endet in einem Knoten v des Suffix-Baums oder in mitten der Beschriftung einer Kante, dann sei v der Endknoten dieser Kante.

[9] hier, in unserem Fall nicht!

4. gib die Inhalte aller Blätter im Teilbaum mit der Wurzel v aus: alle Stellen, an denen P vorkommt.

Laufzeit: $O(m)$ für die Schritte 1 bis 3. Der 4. Schritt kostet mit bspw. Preorder-Traversierung $O(r)$ Zeit, wobei r die Anzahl der Knoten im Baum mit der Wurzel v ist, da dann eventuell der ganze Unterbaum durchlaufen werden muss. Da jeder Knoten ≥ 2 Blätter hat, kann man annehmen, dass $r = O(\text{Anzahl der Blätter})$. Unter diesem Aspekt ist $O(r) = \underline{O(k)}$.

Probleme:

- Platzbedarf von $\Theta(n^2)$
- Aufbau benötigt $\Theta(n^2)$ Zeit, da Suffixe der Länge $1, 2, ..., n$ eingefügt werden müssen, also $O(\sum_{i=1}^{n} i) = O(n^2)$

Speicherplatz reduzieren: Der Speicherplatz kann reduziert werden, wenn man statt der Teilwörter $T_i...T_j$ einfach die Indizes i, j an die Kanten schreibt. Damit verringert sich der Speicherplatz auf:

$$\begin{aligned} O(\text{Anzahl der Kanten}) &= O(\text{Anzahl der Knoten}) \\ &= O(\text{Anzahl der Blätter}) \\ &\boxed{=\text{O(n)}} \end{aligned}$$

Aufbauzeit reduzieren: Für die Konstruktion eines Suffix-Baums gibt es Algorithmen, die das in $O(n)$ zu Stande bekommen (McCreight, Ukkonen)[10]

Dadurch erhalten wir also einen String-Matching-Algorithmus mit linearer Laufzeit:

geg. Text T, Muster P, $|T| = n,\ |P| = m$

- Konstruktion des Suffix-Baums für $T \rightarrow O(n)$ Zeit
- Suche nach $P \rightarrow O(m + k) = O(m + n)$ Zeit

insgesamt: $\underline{O(m + n)}$

andere Anwendungen von Suffixbäumen:

1.) Gegeben sind ein Muster P und mehrere Texte $T_1, ..., T_l$. Finde alle Vorkommen von P in jedem der Texte. Wir konstruieren also einen Suffix-Baum für $T_1\$_1T_2\$_2...\$_{l-1}T_l$, wobei die $\$_1, ..., \$_{l-1}$ Sonderzeichen sind, die alle verschieden sind[11] und in keinem der T_i und auch nicht in P vorkommen.

- konstruiere Suffix-Baum für T
- beschriften Blätter mit (i, j), d.h. der Suffix beginnt in T_i und innerhalb von T_i an der Stelle j
- Suche nach P liefert alle Paare (i, j) mit P beginnt ab Stelle j in T_i.

[10] An dieser Stelle entfällt der Beweis aufgrund der Komplexität.
[11] muss nicht unbedingt sein, aber schaden tut es auch nicht.

Laufzeit: $O(|T_1| + |T_2| + ... + |T_l| + |P|)$

2.) Suche nach dem längsten gemeinsamen Teilwort von zwei Texten T_1 und T_2.

- konstruiere den Suffix-Baum für $T_1\$T_2$ wie vorher
- markiere inneren Knoten v mit '1', wenn im Unterbaum mit der Wurzel v ein Blatt existiert, das mit $(1, j)$ markiert ist. Das Gleiche wiederholt man mit '2'.
- Finde in diesem Baum den „tiefsten" (d.h. die längste Beschriftung auf dem Pfad von der Wurzel her) Knoten, der mit '1' und '2' markiert ist. Die Beschriftung bis da hin ist das längste gemeinsame Teilwort von T_1 und T_2.

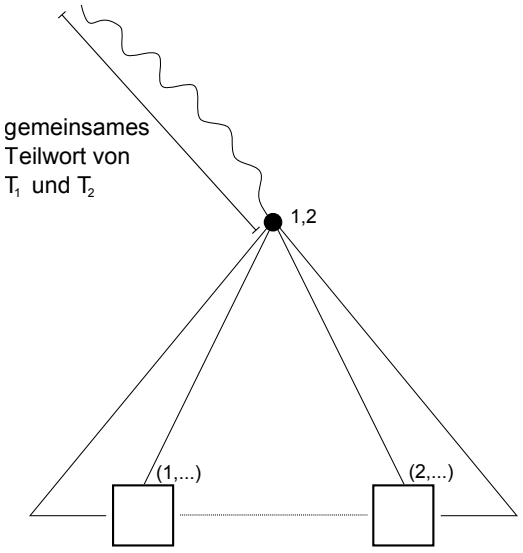


Abbildung 36

7.3 Textkompression

Wir wollen einen Text dahingehend komprimieren bzw. verkürzen, sodass Speicherplatz und Übertragungszeit ohne Informationsverlust eingespart wird.

7.3.1 Code

Ein Code c ist eine Abbildung:

$$c : \Sigma \mapsto \Delta^*$$

wobei Σ und Δ Alphabete sind. c kann dann durch $c(a_1...a_n) := c(a_1) \cdot c(a_2) \cdot ... \cdot c(a_n)$ für $a_1, ..., a_n \in \Sigma$ und $c(\varepsilon) = \varepsilon$ auf Σ^* erweitert werden. Eine weitere Forderung ist, dass c und dessen Erweiterung auf Σ^* als Abbildung injektiv sein sollen.

Bsp: Der Code soll nicht so beschaffen sein, dass gilt:

$$\begin{aligned} c(a) &= 0 \\ c(b) &= 1 \\ c(d) &= 01 \\ \text{mit } \Sigma = \{a, b, d\} &\text{ und } \Delta = \{0, 1\} \end{aligned}$$

denn dann wäre $c(ab) = c(d) = 01$ und die Injektivität wäre nicht mehr gewährleistet. c ist injektiv, falls es ein Präfixcode ist, d.h. kein Wort $c(a)$ ist Präfix eines Wortes $c(b)$ für irgendwelche $a, b \in \Sigma$.

denn: für ein Wort $w \in \Delta^*$ bestimme kürzesten Präfix der Codierung eines Zeichens $\in \Sigma$. Dann muss ein Wort $u \in \Sigma^*$ mit $c(u) = w$ das erste Zeichen a haben.

oft: Codierung mit fester Länge $|c(a)| = |c(b)|$ $\forall a, b \in \Sigma$; z.B.: ASCII (7 Bit), Unicode ($\geq$ 16 Bit)

Für eineb Code, der effizient komprimiert, kann man folgende Anforderungen festlegen: er soll eine variable Länge haben und häufig vorkommende Zeichen weisen eine kürzere Codierung auf, als selten vorkommende. Diese Anforderungen erfüllt zum Beispiel der Huffman-Code.

7.3.2 Huffman-Code

gegeben: Text $T \in \Sigma^*$

finde: Code $c : \Sigma^* \mapsto \{0, 1\}^*$ (mit der gleichen Idee für größere Alphabete)

Wir konstruieren einen Binärbaum wie folgt:

1. schaffe für jedes Zeichen im Text ein Blatt mit der Häufigkeit als Gewicht
2. wiederhole solange wie möglich: nimm 2 vaterlose Knoten mit geringstem Gewicht und schaffe einen Vaterknoten für beide, dessen Gewicht auf die Summe der Gewichte der beiden Kinder gesetzt wird.

Bsp: $T = \,'programmiersprache'$

Zeichen	Häufigkeit	Länge der Codierung
p	||	3
r	||||	2
o	|	4
g	|	4
a	||	3
m	||	3
i	|	4
e	||	4
s	|	4
h	|	5
c	|	5

Abbildung 37: *Die Codierung eines Zeichens $a \in \Sigma$ ist dann die Beschriftung des Weges von der Wurzel bis zum entsprechenden Blatt. Z.B.: $c(r) = 10$ oder $c(h) = 11110$*

Länge der Codierung:

$$\sum_{a \in \Sigma} \underbrace{\#_T(a)}_{\substack{\text{Häufigkeit} \\ \text{von } a \text{ in} \\ T}} \cdot |c(a)| = 60$$

zum Vergleich: Codierung in ASCII würde 126 Bit erfordern.

Laufzeit: $|T| = n,\ |\Sigma| = d$ mit $d \leq n$

Schritt 1: $O(n)$ zum Durchlaufen des Texts

Schritt 2: Implementierung durch Heap bzgl. Gewicht; eine Ausführung

$$\left.\begin{array}{r} 2 \times \text{ streiche Min} \\ 1 \times \text{ Einfügen} \end{array}\right\} \quad = O(log(d))$$

wobei $d \geq$ Anzahl der vaterlosen Knoten im Baum ist. Führt man das dann insgesamt d-mal aus, erhält man für die Laufzeit $O(d \cdot log(d))$. Die Konstruktion des entsprechenden Huffman-Baums erfolgt in $\underline{O(n + d \cdot log(d))}$ bzw. $O(n)$, falls d eine Konstante ist.

Decodierung: z.B.: 011|010|1100 $\rightarrow mai$

Will man einen Code wieder decodieren, muss man den entsprechenden Kanten im Baum folgen. Falls man dann bei einem Blatt angekommen ist, gibt man das jeweilige Zeichen aus und fängt wieder bei der Wurzel an, bis der ganze Code abgearbeitet wurde. Die ausgegebene Zeichenkette ist dann der entschlüsselte String.

Laufzeit: $O(m)$, wobei m die Länge des Codeworts ist.

Satz: Der Huffman-Code ist der kürzeste Präfixcode für einen Text T.

7.3.3 Greedy-Strategie

Allgemeine Strategie für Optimierungsprobleme. Nimm lokal, momentan kostengünstigste, iteriere, bis Gesamtlösung gefunden ist.

7.3.4 Suchmaschinen

Im Web gibt es dahingehend bspw. 'Web Crawler'. Das sind Programme, welche das Web durchlaufen (per Links) und eine Statistik darüber anlegen, welche Wörter wo vorkommen (also mit welcher Häufigkeit, an welcher Stelle etc.). Die Web-Crawler schaffen ein Wörterbuch mit Einträgen der Form (w, L), wobei w ein Wort und L eine Liste von URL's ist, die w enthalten.

z.B.: Suche bei Google

$$w = \text{'the'} \rightarrow |L| = 25.270.000.000 \text{ in } < 1\text{s}$$
$$w = \text{'university'} \rightarrow |L| = 2.700.000.000$$
$$w = \text{'universiy'} \rightarrow |L| = 2.000.000$$

Datenstruktur: Die einzelnen Wörter (insgesamt mehrere Millionen auf der Gesamtheit aller Webseiten) werden in einem Trie abgelegt. Die Blätter des Tries sind Zeiger auf Hintergrundspeicher (Festplatten), der die Listen L enthält.

mehrere Suchwörter: Hierfür wird die Schnittmenge der Listen bestimmt, die diese Wörter enthalten. Dabei ist es sinnvoll die Listen zuvor (lexikografisch oder nach einer ID-Nr. bspw.) zu sortieren und diese dann wie bei Mergesort zu durchlaufen und die Elemente auszugeben, die überall vorkommen.

zusätzlich: Ranking

8 Graphenalgorithmen

8.1 Definitionen

Graph:
Ein (ungerichteter) Graph ist ein Paar $G = (V, E)$, wobei $V \neq \varnothing$ eine Menge von Knoten, Ecken (vertices) und $E \subseteq \binom{V}{2}$ eine Menge von 2-elementigen Teilmengen (ungeordnete Paare) von V ist. Die Menge E wird auch als Kanten (edge) bezeichnet.

Adjazenz, Inzidenz:
Ist $e = (u, v) \in E$, so heißt u adjazent zu v und u, v inzident zu e oder Endpunkte von e.

Bsp: $V = \{a, b, c, d, f\}$ und $E = \{(a, b), (a, c), (a, d), (b, c), (c, d)\}$

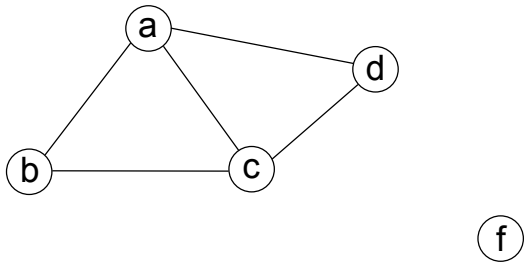


Abbildung 38

Grad:
Der Grad eines Knotens beschreibt die Anzahl der adjazenten Knoten.

Anwendungsbereiche

- Verkehrsnetze
- Rechnernetze
- World Wide Web
- Labyrinth
- Social Networks

Weg, Pfad:
Wenn $G = (V, E)$ ein Graph ist, dann ist eine nichtleere Folge von Knoten $v_0 v_1 ... v_n$ mit $\{v_i, v_{i+1}\} \in E$ für $i = 0, ..., n-1$ ein Weg (path) in G, wobei n die Länge des Weges beschreibt.

Ein Weg heißt einfach, falls alle seine Knoten verschieden sind.

Ein Weg heißt Kreis, genau dann, wenn $v_n = v_0$.

Ein Weg heißt einfacher Kreis, genau dann, wenn $v_i \neq v_j$ für $0 \leq i, j \leq n$ und $\{i, j\} \neq \{0, n\}$

Allgemein gilt für Kreise, dass deren Länge ≥ 1 ist. Bei ungerichteten Graphen haben einfache Kreise eine Länge ≥ 3.

Im obigen Beispiel wäre die Folge $b\ a\ d\ c\ b$ ein Kreis, $a\ d\ a\ d\ a\ d$ ein nicht-einfacher Kreis und f ein Weg der Länge 0

Teilgraph:
$G' = (V', E')$ heißt Teilgraph von $G = (V, E)$, genau dann, wenn $V' \subseteq V$ und $E' \subseteq E$.

induzierte Teilgraphen:
Ist $G = (V, E)$ ein Graph und gilt $V' \subseteq V$, so heißt $G' = (V', E')$ mit $E' = \{(u, v) \in E \mid u, v \in V'\}$ der von V' induzierte Teilgraph.

im Bsp:

$$V' = \{a, c, f\} \text{ dann } G' :$$

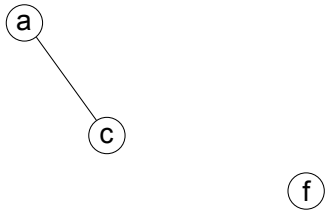


Abbildung 39

Azyklische Graphen:
Ein Graph heißt azyklisch (kreisfrei, acyclic), genau dann, wenn er keine (einfachen) Kreise besitzt. Der Begriff Wald wäre in diesem Sinne eine weitere Umschreibung für diesen Sachverhalt.

Bsp:

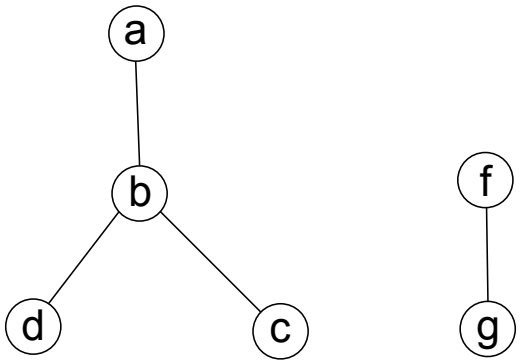


Abbildung 40

Zusammenhängende Graphen:
Ein Graph $G = (V, E)$ heißt zusammenhängend (connected), wenn es für beliebige Knoten $u, v \in V$ einen Weg $v_0 v_1 ... v_n$ gibt, mit $u = v_0$ und $v = v_n$.

Bsp:

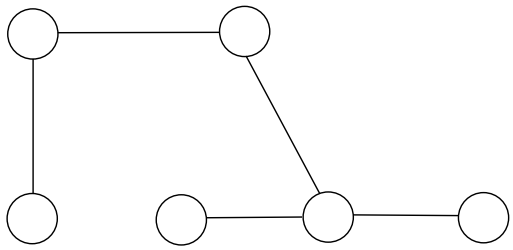


Abbildung 41

Baum:
Ein azyklischer, zusammenhängender Graph heißt Baum.

Zusammenhangskomponente:
Eine Zusammenhangskomponente (connected component) eines Graphen $G = (V, E)$ ist ein maximal[12] zusammenhängender Teilgraph $G' = (V', E')$. Die Graphen in den Abbildungen 39 und 40 haben jeweils 2 Zusammenhangskomponenten.

8.1.1 Small-World-Phenomenon

$G = (V, E)$, wobei V die Menge der Menschen darstellt und $(u, v) \in E$ eine Kante ist, die symbolisieren soll, dass sich die beiden entsprechenden Menschen kennen.

Frage: Was ist der Durchmesser von diesem Graph?

Vermutung: 7 oder 8

8.2 Gerichtete Graphen

Ein gerichter Graph $G = (V, E)$ ist ein Graph mit der Knotenmenge $V \neq \varnothing$ und der Kantenmenge $E \subseteq V \times V$.

[12] es gibt keinen Knoten $v \in V \backslash V'$, sodass der von $V' \cup \{v\}$ induzierte Teilgraph zusammenhängend ist und G' der von V' induzierte Teilgraph ist

Bsp: $V = \{a, b, c, d\}$ und $E = \{(a, b), (a, a), (b, c), (c, b), (c, d), (a, d)\}$

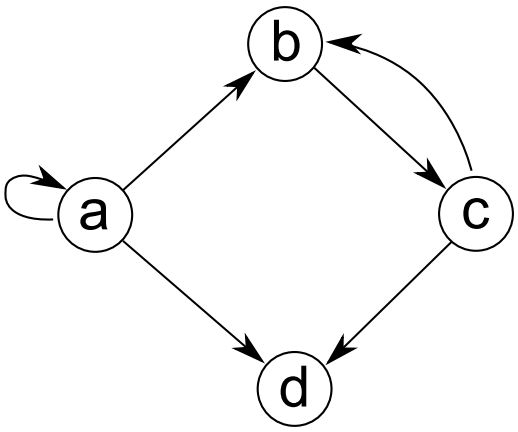


Abbildung 42

Grad:
Der Grad ist hier analog zum Grad bei ungerichteten Graphen definiert, bis auf dem Ingrad/Ausgrad eines Knotens als Anzahl der eingehenden/ausgehenden Kanten.

streng zusammenhängende Graphen: (statt zusammenhängend)
es gibt für $u, v \in V$ einen Weg von u nach v. Der Graph aus Abbildung 42 ist nicht streng zusammenhängend.

strenge Zusammenhangskomponenten:
maximaler Teilgraph, der streng zusammenhängend ist.

im Bsp:

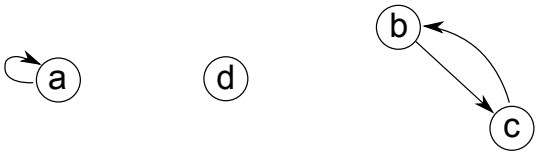


Abbildung 43: *streng zusammenhängende Teilgraphen aus dem letzten Beispiel*

bei Anwendungen: Knoten (und Kanten) eines Graphen sind meist mit Elementen eines Universums U „markiert“.

8.3 Datenstrukturen für Graphen

8.3.1 Adjazenzlisten

Zu jedem Knoten v erstellt man eine Liste der zu v adjazenten Knoten (im gerichteten Fall sind dies alle Knoten u mit $(v, u) \in E$). Der Speicherplatzbedarf bei $n = |V|$, $m = |E|$ ist

dann die Gesamtlänge aller Listen, also m bei einem gerichteten und $2m$ bei einem ungerichteten Graphen.

insgesamt: Platz $\Theta(n+m)$

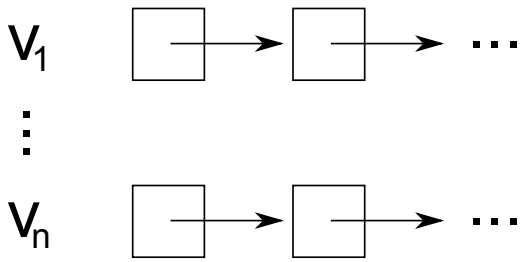


Abbildung 44: *Adjazenzlisten*

8.3.2 Adjazenzmatrix

$n \times n$-Matrix M aus 0-en und 1-en

$$M_{ij} = \begin{cases} 1, \text{ falls } (v_i, v_j) \in E \\ 0 \text{ sonst} \end{cases}$$

insgesamt: Platz $\Theta(n^2)$

Frage: Wie groß kann m sein?

gerichtet: $O \leq m \leq n^2$

ungerichtet: $0 \leq m \leq \binom{n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$

Adjazenzlisten sind besser für „dünnbesetzte“ Graphen geeignet als Adjazenzmatrizen, d.h. wenn $m \ll n^2$.

8.4 Der abstrakte Datentyp „Graph“

8.4.1 Operationen

1) Knoten(): liefert Liste aller Knoten (die man selbst als Klasse definieren sollte)

2) adj(v,w): liefert Wahrheitswert darüber, ob v adjazent zu w ist

3) adjListe(v): liefert Liste der zu v adjazenten Knoten

4) knElement(v): liefert das in v gespeicherte Element

5) kaElement(v,w): liefert das in der Kante (v, w) gespeicherte Element

6) einfKnoten(v,x): füge neuen Knoten v mit Beschriftung[13] x zum Graphen hinzu

7) einfKante(v,w,x): füge neue Kante (v, w) mit Beschriftung x hinzu

8) setzeKnEl(v,x): beschrifte v mit x

9) setzeKaEl(v,w,x): beschrifte Kante (v, w) mit x

10) streicheKnoten(v): streiche den Knoten v einschließlich der inzidenten Kanten

11) streicheKante(v,w): streiche Kante (v, w)

8.4.2 Laufzeiten der Operationen

	Adj. Listen	Adj. Matrizen
1	$O(n)$	$O(n)$
2	$O([aus-]\ Grad\ v)$	$O(1)$
3	$O(1)$	$O(n)$
4	$O(1)$	$O(1)$
5	$O([aus-]\ Grad\ v)$	$O(1)$
6	$O(1)$	$O(n)$
7	$O(1)$	$O(1)$
8	$O(1)$	$O(1)$
9	$O([aus-]\ Grad\ v)$	$O(1)$
10	$O(m)$	$O(n^2)$
11	$O([aus-]\ Grad\ v + (Grad\ w))$	O(1)

Tabelle 2: *Erklärungen: Bei 1, 5, 9 und 11 $O(Grad\ v)$, da jeweils die Adjazenzlisten von v durchlaufen werden müssen. Bei 5 $O(1)$, da man statt der 1-en auch die Beschriftung direkt in die Matrix schreiben kann. Bei 6 $O(n)$, da unter Umständen die gesamte Matrix vergrößert werden muss. Bei 7 $O(1)$, da man w einfach an den Anfang der Adjazenzliste von v anfügen kann. Bei 10 $O(m)$, da man v aus allen Listen entfernen muss, wobei die Gesamtlänge aller Listen m ist. Bei 10 $O(m^2)$, da man eine Zeile und eine Spalte aus dem Array entfernen muss.*

[13] Beschriftung meint hier das schon zuvor erwähnte gespeicherte Elemtent

8.5 Traversierung von Graphen

Eine Traversierung beschreibt einen von einem Knoten ausgehenden, systematischen Durchlauf eines Graphen, wobei alle erreichbaren Knoten besucht werden.

8.5.1 Tiefensuche (depth first search - 'dfs')

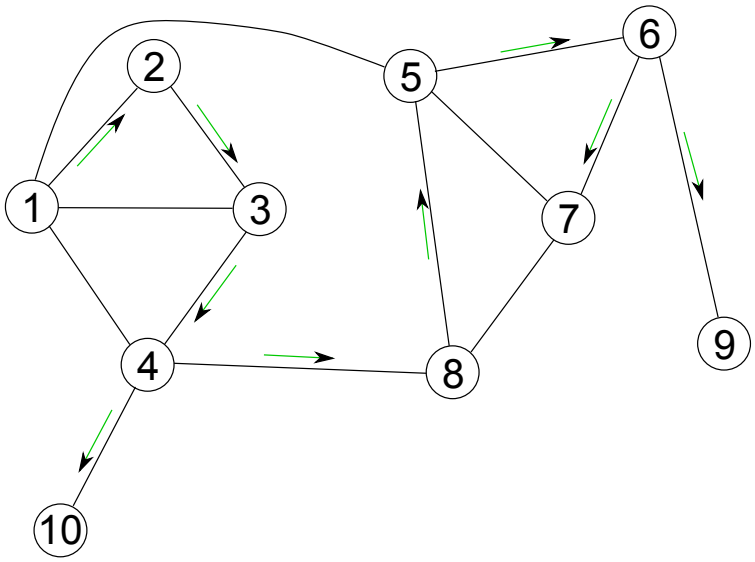


Abbildung 45: *Analog für gerichtete Graphen. Kanten, die von dfs durchlaufen werden, bilden einen Baum (dfs-Baum), mit entsprechenden Baumkanten. Die restlichen Kanten werden als Vorwärtskanten bezeichnet.*

Algorithmus: $DFS(G, v)$ mit Anfangsknoten v

1. markiere v als "besucht"
2. **for all** u adjazent v **do**:

   if v nicht besucht ($\rightarrow$ Kante (v, u) registrieren) **then** DFS(G,u)

Frage: Welche Knoten werden durch $DFS(G, v)$ besucht?

Satz: $DFS(G, v)$ besucht alle Knoten u zu denen es einen Weg von v aus gibt (für gerichtete und ungerichtete Graphen).

Um auch die anderen Knoten von G zu erhalten:

for all Knoten v von G **do**

if v nicht besucht **then** DFS(G,v)

Vorausgesetzt, dass die "besucht"-Markierungen immer stehen gelassen werden.

Laufzeit von DFS (angenommen Adjazentlisten):

Schritt 1): $O(n)$, wobei n die Anzahl der Knoten ist.

Schritt 2): Alle Adjazenzlisten werden einmal in den verschiedenen Aufrufen von DFS durchlaufen. Jeder dieser Schritte benötigt $O(1)$ Zeit, insgesamt also $O(m)$, bei m (oder $2m$) Schritten.

DFS insgesamt: $\underline{O(n+m)}$ Zeit, wobei $m = |E|$ und $n = |v|$.

Beweis des Satzes:

Beh.: u Knoten der von v aus erreichbar ist $\Leftrightarrow$ $DFS(G, v)$ besucht u.

„⇒“: Sei $v = v_0, v_1, ..., v_k = u$ ein Weg von v nach u

Induktion über k:

K = 0: $v = u$ und v wird in $DFS(G, v)$ besucht (Zeile 1)

Ind. Schritt: $k - 1 \rightarrow k$; nach I.V.: v_{k-1} wird in $DFS(G, v)$ besucht. Dabei wird $DFS(G, v_{k-1})$ (in diesem Aufruf Zeile 2) aufgerufen. v_k ist adjazent zu v_{k-1}. Ist v_k schon als "besucht" markiert: fertig, sonst: Aufruf von $DFS(G, v_k)$ und in Zeile 1 dieses Aufrufs v_k als besucht markieren.

„⇐“: $DFS(G, v)$ besuche u, dann muss $DFS(G, u)$ irgendwann innerhalb des Aufrufs von $DFS(G, v)$ aufgerufen werden (Schachtelung der rekursiven Aufrufe):

$$DFS(G, v = v_0) \rightarrow DFS(G, v_1) \rightarrow DFS(G, v_2) \rightarrow ... \rightarrow DFS(G, v_{k-1}) \rightarrow DFS(G, u = v_k)$$

$DFS(G, v_i)$ ruft $DFS(G, v_{i+1})$ direkt auf für $i = 0, ..., k-1$, was nur in Zeile 2 passieren kann. v_{i+1} ist adjazent zu v_i.

also: $v = v_0, v_1, ..., v_k = u$ ein Weg von v nach u.

Falls in Zeile 2 die Kante (v, u) registriert wird, so bilden diese Kanten zusammen mit den besuchten Knoten einen sogenannten dfs-Baum. Der Algorithmus der alle Knoten besucht, erzeugt dann dementsprechend einen dfs-Wald.

8.5.2 Zusammenfassung

Die Tiefensuche erlaubt folgende Operationen in $O(m+n)$ Zeit auszuführen (ungerichter Graph $G = (V, E)$):

a) entscheiden, ob G zusammenhängend ist

b) Berechnung eines aufspannenden Baums, falls G zusammenhängend ist (sonst aufspannender Wald).

Def: ist $G = (V, E)$ ein (ungerichteter) Graph, dann heißt $T = (V, E')$ aufspannender Baum (Spannbaum, spanning tree), wenn für den Baum gilt $E' \subseteq E$

c) Berechnung der Zusammenhangskomponenten

d) Berechnung eines Weges zwischen u und v, falls existent

e) Test auf Kreisfreiheit

f) bei gerichteten Graphen: testen, ob Graph streng zusammenhängend ist

Algorithmus für f): Ein gerichteter Graph $G(V, E)$ ist streng zusammenhängend $\Leftrightarrow$ es gibt einen Knoten $s \in V$, sodass zu jedem Knoten $v \in V$ ein gerichteter Weg von s nach v und einer von v nach s existiert.

Beweis: [14]

„$\Rightarrow$“: offensichtlich

„$\Leftarrow$“: sei s ein solcher Knoten, dann existiert ein Weg von u nach s und einer von s nach v. Zusammensetzen liefert Weg von u nach v.

konkreter Algorithmus:

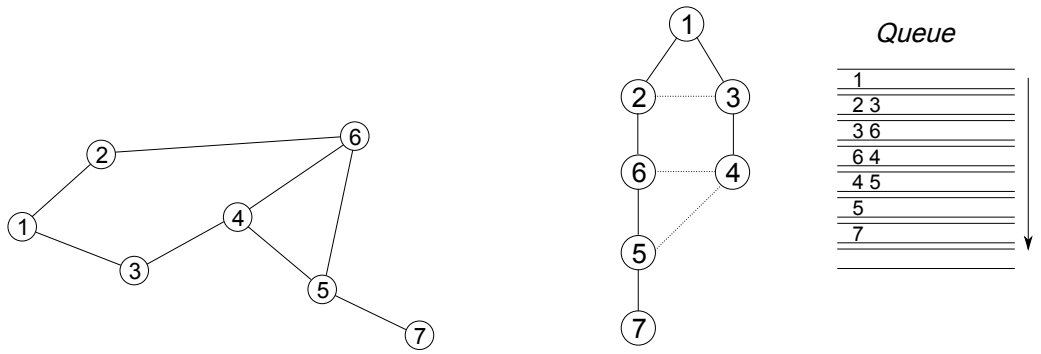
wähle beliebigen Knoten $s \in V$

führe $dfs(G, s)$ durch

- falls nicht alle Knoten besucht werden: antworte "G nicht streng zusammenhängend"
- sonst: sei G' der Graph aus G durch Umkehrung der Richtung aller Kanten; führe dann $dfs(G', s)$ durch
  - falls alle Knoten besucht werden: antworte "ja"
  - sonst: antworte "nein"

[14] Anmerkung: Der Beweis ist eigentlich nutzlos und sollte nach dem Durchlesen wieder vergessen oder gleich übersprungen werden.

8.5.3 Breitensuche (breadth-first-search)



(a) *Bsp: Graph mit 7 Knoten*

(b) *dazugehöriger bfs-Baum mit Anfangsknoten 1 und Warteschlange für jeden Schritt*

Abbildung 46: *Breitensuche*

Algorithmus BFS(G,v) benutzt Warteschlange von Knoten Q, die anfangs ler ist:

```
Q.enqueue(v);
markiere v als "gesehen"
while Q nichtLeer do
    w := Q.dequeue();
    (bearbeite w)       // je nachdem wofuer bfs verwendet wird
    for all u adjazent zu w do
        if u noch nicht gesehen then
            Q.enqueue(u)
            {w,u} "Baumkante" // optional
            markiere v als "gesehen"
        else
            {w,u} "Querkante" // optional
```

Laufzeit: $\Theta(n+m)$, wobei n für das Einfügen/Streichen/Markieren der Knoten und m für das Durchlaufen aller Adjazenzlisten steht.

$BFS(G, v)$ liefert einen Baum, der alle Knoten enthält, die von v aus erreichbar sind. Dieser Baum wird dann als bfs-Baum bezeichnet. Die bfs-Nummer entspricht in diesem Fall der Nummerierung der Knoten in der Reihenfolge, in der sie von bfs besucht wurden.

Eigenschaften (unger. Graphen):

a) $bfs(G, v)$ besucht alle Knoten in der Zusammenhangskomponente von v

b) im entsprechenden bfs-Baum ist der Weg von v zu einem beliebigen Knoten w der kürzeste Weg von v nach w im Graphen G

c) die Kanten aus E, die nicht im bfs-Baum enthalten sind, Querkanten: die Tiefe von u und v unterscheiden sich höchstens um 1

Beweise der Eigenschaften:

a) Analog zu dfs

b) Sei r die Länge des kürzesten Weges (in G) von v nach w. $v = w_0 ... w_r = w$ sei ein solcher Weg. Sei $v = v_0 ... v_s = w$ der Weg im bfs-Baum.
zu zeigen: $s = r$ (klar $s \geq r$)

Induktion über r: Für jedes w gilt: kürzester Weg hat Länge $r \Rightarrow$ Weg in bfs-Baum hat Länge r.

r = 0: dann $v = w$, Weg der Länge 0 auch im bfs-Baum

r→ r+1:

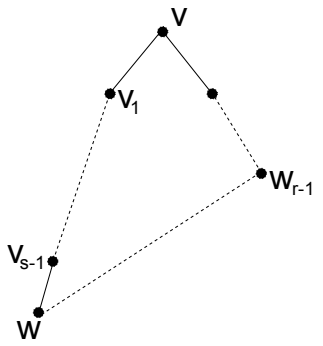


Abbildung 47: *Der Weg im bfs-Baum zum Knoten w_{r-1} hat nach Induktionsvoraussetzung eine Länge von $r-1$.*

falls $s-1 > r-1$, so muss w_{r-1} vor v_{s-1} bearbeitet worden sein, da im bfs-Baum Ebene für Ebene bearbeitet wird. Knoten w war, als w_{r-1} bearbeitet wurde, noch nicht mit "gesehen" markiert und, da adjazent zu w_{r-1}, hätte die Kante (w_{r-1}, w) in Baum eingefügt werden müssen ↯
also: $s - 1 \neq r - 1$

c) Übungsaufgabe

Folgerung: Die Breitensuche kann dafür benutzt werden, um folgendes in $\Theta(m+n)$ Zeit zu tätigen, wobei $|V| = n$ und $|E| = m$ für $G = (V, E)$:

a) Testen, ob G (ungerichtet) zusammenhängend ist

b) aufspannenden Baum finden, falls G zusammenhängend ist

c) vom Startknoten aus alle kürzesten Wege zu allen erreichbaren Knoten

d) feststellen, ob G kreisfrei ist oder einen Kreis in G zu finde

8.6 Gerichtete, azyklische Graphen

Gerichtete Graphen, die keine Kreise enthalten („directed acyclic graphs"- dag), haben beispielsweise folgende Anwendungen:

Projekt mit Teilprojekten $\widehat{=}$ Knoten: (u, v) heißt dann: Teilprojekt u muss fertiggestellt sein, bevor v ausgeführt werden kann.

Compilerbau: „Codeoptimierung"- *dag* für arithmetische Ausdrücke:

$$(a+b)*c+d*((a+b)*c)+(a+b)*d$$

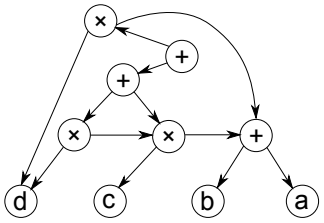


Abbildung 48: *dag zum obigen arithmetischen Ausdruck, sodass mehrfach aufkommende Teilausdrücke nur 1× berechnet werden. Daraus wird dann bspw. Maschinencode erzeugt.*

8.6.1 Topologisches Sortieren

Auflistung der Knoten, sodass wenn Kante (u, v) existiert v später als u aufgelistet wird, heißt topologisches Sortieren.

Bsp:

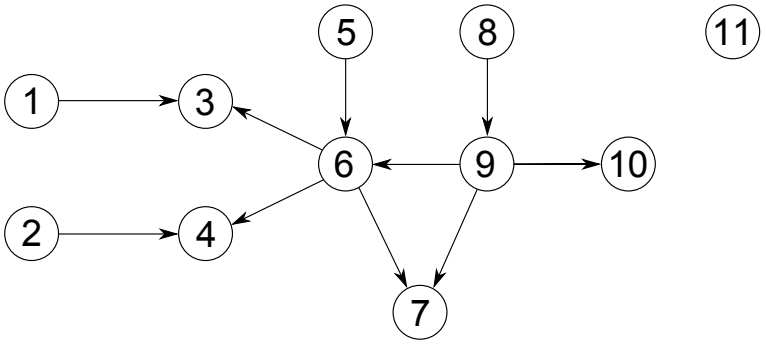


Abbildung 49: *„Projekt“ für's Anziehen: 11 (Hut), 1 (linke Socke), 5 (Unterhose), 2 (rechte Socke), 8 (Unterhemd), 9 (Hemd), 10 (Jacke), 6 (Hose), 7 (Gürtel), 3 (linker Schuh), 4 (rechter Schuh)*

Algorithmus:
Solange Graph nicht leer:

- wähle Knoten v mit Ingrad O
- liste v auf
- entferne v sowie alle ausgehenden Kanten

Laufzeit: $\Theta(m+n)$

Bemerkung: Jeder gerichtete azyklische Graph besitzt mindestens einen Knoten mit Ingrad 0.

Auswertung von als *dag* dargestellten Ausdrücken auch mit topologischer Sortierung (Kanten umdrehen)

8.7 Kürzeste Wege in Graphen

Problem: Gegeben ist ein gerichteter Graph $G = (V, E)$ mit Kantengewichten (-längen) $c : E \mapsto \mathbb{R}_{\geq 0}$ „Länge“ Länge eines Weges π in G mit $\pi = v_0 v_1 ... v_n$ definiert als $c(\pi) := \sum_{i=0}^{n-1} c(v_i, v_{i+1})$ ($c(a, b)$ beschreibt die Kosten des Weges von a nach b) .

Frage: gegeben $u, v \in V$: finde kürzesten Weg von u nach v bzgl. c.

Anwendungen:

- Routenplaner
- Bahn-Info-System

allgemeineres Problem: für einen gegebenen Startknoten s: finde alle kürzesten Wege von s nach v mit $v \in V \rightarrow$ „single-source-shortest-path“(siehe Algorithmus von Dijkstra 8.7.1)

noch allgemeiner: finde alle kürzesten Wege zwischen zwei Knoten → „all-pairs-shortest-paths“(siehe Algorithmus von Floyd-Warshall 8.7.2)

8.7.1 Algorithmus von Dijkstra

benutzt:

- Feld $D[v]$ für alle $v \in V$
  Länge des bisher gefundenen kürzesten Weges von s nach v
- Prioritätswarteschlange Q
  (Knoten, zu denen der kürzeste Weg noch nicht gefunden wurde, sortiert nach D-Werten)

Algorithmus:

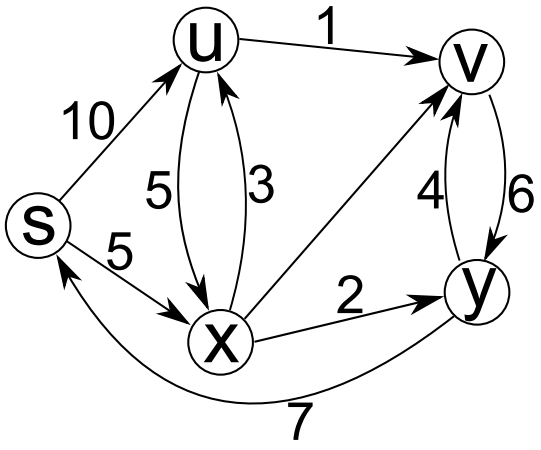
```
        Dijkstra(G,s)
            for all v in V do
1)              if v=s then D[v] := 0
                else D[v] := INFINITY
            Q := V\{ s}

2)          while Q not empty do
3)              v:= streiche Min(Q)
4)              for all u in Q adjazent zu v in Q do
5)                  D[u] := min(D[u],D[v]+c(v,u))
6)                  falls sich D[u] hier gändert hat: vorg[u] := v
```

Am Schluss:

- Länge des kürzesten Weges von s nach v in $D[v]$ für alle $v \in V$.
- kürzester Weg nach v selbst:

$$s = v_0, ..., v_{k-1}, v_k = v \; mit \; v_{j-1} = vorg[v_j]$$

Bsp:



	1	2	3	4	5
s	$\boxed{0}$	...			
u	∞	10	8	$\boxed{8}$	
x	∞	$\boxed{5}$	...		
v	∞	∞	14	11	$\boxed{9}$
y	∞	∞	$\boxed{7}$	...	

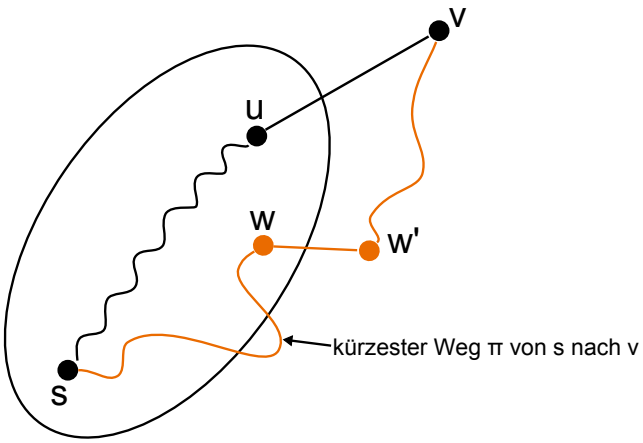
Tabelle 3: *D-Warte in den jeweiligen Schritten 1-5. Am Ende: länge der kürzesten Wege von s aus*

zur Korrektheit: sei $S = V \backslash Q$
zeigen: zu jedem Zeitpunkt gilt: zu jedem Knoten $v \in S$ ist $D[v] = d(v)$, wobei $d(v)$ = Länge des kürzesten Weges von s nach v.

Beweis: durch Induktion über Zahl i der Iterationen der *while*-Schleife

$\underline{i = 0:}$ $S = \{s\}$ und $D = [s] = 0 = d(s)$ ✓

$\underline{\leq i \to i + 1:}$ Sei v der Knoten, der bei $(i+1)$-ten Iteration nach S kommt. Angenommen, die Behauptung ist falsch, d.h. $D[v] \neq d(v)$. Da $D[v]$ die Länge eines Weges von s nach v ist, gilt $D[v] > d(v)$.



u sei der Knoten, bei dessen Aufnahme in S, $D[v]$ zum letzten Mal verringert wurde.

$$D[v] = D[u] + c(u, v) \overset{I.V.}{=} d(u) + c(u, v)$$

(w, w') letzte Kante auf π mit $w \in S_i, w' \notin S_i$, wobei S_i gleich S nach i Iterationen.

$$\begin{aligned} \text{Länge von } \pi = d(v) &\geq d(w) + c(w, w') \\ &= D[w] + c(w, w') \end{aligned}$$

Als w nach S aufgenommen wurde, wurde $D[w']$ auf einen Wert $\leq D[w] + c(w, w')$ gesetzt.

also:

$$D[\underbrace{v}_{Annahme} > d(v) = D[w] + c(w, w') \geq D[w'] \quad \lightning$$

Die letzte Gleichung liefert einen Widerspruch, da statt v w' aufgenommen werden müsste. Daraus folgt also, dass der Algorithmus korrekt ist.

Laufzeit: Datenstrukturen:

Heap für Q

Feld für D

dann kostet ($n = |V|$, $m = |E|$):

Zeile (1): $O(n)$

Zeile (3): $O(log\ n)$ pro Ausführung

Zeile (5): $O(log\ n)$ pro Ausführung (zum Aufrechterhalten von Q)

Zeile (6): $O(1)$

insgesamt:

für (3),(6): $O(n \cdot log(n))$

für (4),(5): $O(m \cdot log(n))$, wobei m in diesem Kontext die Gesamtlänge aller Adjazenzlisten darstellt.

Fazit: Insgesamt hat der Algorithmus von Dijkstra also eine Laufzeit von $\underline{\Theta((n+m) \cdot log(n))}$. Diese Laufzeit lässt sich jedoch mit Hilfe von „Fibonacci-Heaps“ als Datenstruktur für Q auf $\Theta(n \cdot log(n) + m)$ verbessern. Diese Heaps erlauben Zeile (5) in $O(1)$ Zeit (amortisiert).

8.7.2 Algorithmus von Floyd-Warshall

Jeden Knoten in V nimmt man als Startknoten an und führt jeweils den Dijkstra-Algorithmus durch. Für dichte Graphen ergibt sich somit eine Laufzeit von $O(n^3 \cdot log(n))$ und mittels Fibonacci-Heap eine Laufzeit von $O(n^3)$.

Dieser Algorithmus berechnet Werte $d_{ij}^{(k)}$ = Länge des kürzesten Weges von Knoten i nach Knoten j, wobei als Zwischenknoten nur die Knoten $1, ..., k$ zugelassen sind. (o.B.d.A. $V = \{1, ..., n\}$)

```
for i=1 to n do
1)  for j=1 to n do
        if i==j then d_{ij}^{(0)} = 0
        elif i!=j and (i,j) in E then c(i,j)
        elif i!=j and (i,j) not in E then INFINITY

for k=1 to n do
2)  for j=1 to n do
        for i = 1 to n do
            d_{ij}^{k} := min(d_{i}^{(k-1)} , d_{ik}^{(k-1)}+d_{kj}^{(k-1)})
```

2 Beobachtungen:

1. Teilwege eines kürzesten Weges sind immer selbst kürzeste Wege
2. sind u, v Knoten, so gibt es immer einen kürzesten Weg von u nach v auf dem jeder Knoten höchstens einmal vorkommt.

Zur Berechnung von $d_{ij}^{(k)}$ gibt es 2 Fälle:

a) es einen kürzesten Weg mit Zwischenknoten $\in \{1, ..., k\}$ von i nach j, der K nicht einthält, dann $d_{ij}^{(k)} = d_{ij}^{k-1}$

b) jeder hat K als Zwischenknoten, dann $d_{ij}^{(k)} = d_{ik}^{k-1} + d_{kj}^{(k-1)}$

Am Schluss: Ergebnis in der Matrix[15] $(d_{ij}^{(n)})_{1 \leq i,j \leq n}$

Laufzeit:

$\Theta(n^2)$ für (1)

$\Theta(n^3)$ für (2)

$\Theta(n^3)$ Platz, kann auf $\Theta(n^2)$ vermindert werden.

Vorgehensweise: Für alle Teilprobleme optiale Lösung berechnen, dabei Lösungen von kleineren Teilproblemen (hier $d_{ij}^{(k-1)}$) zur Lösung größerer benutzen ($d_{ij}^{(k)}$) heißt „dynamisches Programmieren“.

8.7.3 Transitive Hülle

$G = (V, E)$ gerichteter Graph. Können E als 2-stellige Relation auf V verfassen.

gesucht: transitiver Abschluss E^* der Relation E, d.h. $(u, v) \in E \Leftrightarrow \exists v_0 ... v_n \in V$ mit $n = v_0, v = v_n$ und $(v_i, v - i + 1) \in E$, wobei $n > 0$ und $i = 0, ..., n - 1$.
D.h. es gibt einen gerichteten Weg von u nac v. Angenommen G gegeben, als Adjazenzmatrix. Finde Adjazenzmatrix für $G^* = (V, E^*)$.

```
for i=1 to n do
1)  for j=1 to n do
        if i==j then d_{ij}^{(0)} = TRUE
        elif i!=j and (i,j) in E then d_{ij}^{(0)} = TRUE
        elif i!=j and (i,j) not in E then d_{ij}^{(0)} = FALSE

for k=1 to n do
2)  for j=1 to n do
        for i = 1 to n do
            d_{ij}^{k} := OR(d_{i}^{(k-1)} , d_{ik}^{(k-1)} AND d_{kj}^{(k-1)})
```

In $d_{ij}^{(k)}$ steht dann ein Wahrheitswert darüber, ob ein kürzester Weg exitiert

[15] (k) : Zwischenknoten $\in \{1, ..., k\}$, keine Einschränkungen.

8.8 Minimale Spannbäume

gegeben: (ungerichteter) Graph $G = (V, E)$ und eine Kostenfunktion $c : E \mapsto \mathbb{R}$

gesucht: Spannbaum $T = (V, E')$

$$E' \subseteq E \text{ mit } \sum_{e \in E'} c(e) \text{ minimal}$$

8.8.1 Lemma

Voraussetzungen: Sei $G = (V, E)$ ein Graph und $c : E \mapsto \mathbb{R}$ eine Kostenfunktion. Sei dann A eine Teilmenge von E, die zu einem Spannbaum T erweitert werden kann. $\{u, v\} \in E$ sei eine Kante minimalen Gewichts, die zwei unterschiedliche Zusammenhangskomponenten im Graph[16] $W = (V, A)$ miteinander verbindet (mit den Knotenmengen $V_1, V_2 \subseteq V$).

Dann gibt es einen minimalen Spannbaum (MST - minimum spanning tree[17]), der $A \cup \{\{u, v\}\}$ enthält.

8.8.2 Beweis des Lemmas

Betrachte MST T, der A enthält.

1. Fall: T enthält die Kante $\{u, v\}$: fertig

2. Fall: T enthält Kante $\{u, v\}$ nicht: fügen wir $\{u, v\}$ in T ein, so entsteht genau ein Kreis, in dem eine Kante $\{u', v'\}$ existiert, deren Gewicht $\geq$ dem Gewicht $c(u, v)$ ist, nach Wahl von $\{u, v\}$.
Ersetzen wir nun in dem Kreis $\{u', v'\}$ durch $\{u, v\}$, so erhalten wir einen Spannbaum T' mit $c(T') \leq c(T)$, also $c(T') = c(T)$, wobei T auch ein MST ist.

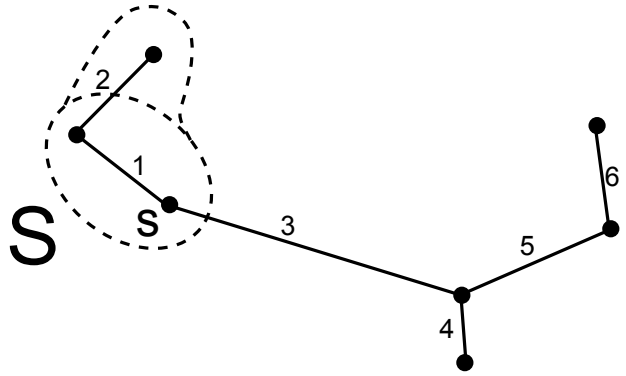
Das Lemma besagt, dass man einen MST mit greedy-Strategie konstruieren kann:

- beginne mit Graph $G_0 = (V, \varnothing)$ ohne Kanten
- **n-1 - mal:** füge kürzeste Kante $\in E$ hinzu, die zwei unterschiedliche Zusammenhangskomponenten verbindet

[16] W für Wald

[17] Anmerkung: im Englischen gibt es eine feine Unterscheidung zwischen 'minimum-' und 'minimal-spanning-tree'.

8.8.3 Algorithmus von Prim



Der Algorithmus von Prim benutzt eine Prioritätswarteschlange Q und ein Feld D. Wir wählen einen beliebigen Knoten $s \in V$. Für die Menge S soll gelten $S = V \backslash Q$

$$Q := V,\ D[v] := \begin{cases} 0, & v = s \\ \infty, & sonst \end{cases}$$

```
while Q nicht Leer do
    v := Q.streicheMin()
    for all u adjazent zu v do
        if c(v,u) < D[u] then
            D[u] := c(v,u)
            vorgaenger[u] := v
```

Korrektheit und Laufzeit: Dieser Algorithmus hat genau die gleiche Struktur wie der Algorithmus von Dijkstra und damit auch die gleiche Laufzeit von $\Theta((n+m) \cdot log(n))$ mit $n = |V|$ und $m = |E|$. Die Korrektheit des Algorithmus folgt aus dem obigen Lemma, wobei wir immer die kürzeste Kante von einem Knoten $\in S$ und einem beliebigen anderen nehmen, wobei Ersterer eine Zusammenhangskomponente und Letzterer eine andere Zusammenhangskomponente ist. Kanten des Baums $\{u, vorgaenger[u]\}, u \in V \backslash \{s\}$.

8.8.4 Algorithmus von Kruskal

1. sortiere alle Kanten aufsteigend nach ihrem Gewicht
2. für jede Kante $e = \{u, v\}$:
   (a) prüfe, ob u, v in der gleichen Zusammenhangskomponente bzgl. der bisher eingefügten Kanten liegen
   (b) falls nein: nimm Kante $e = \{u, v\}$ in MST auf

Korrektheit: Die Korrektheit des Algorithmus folgt aus dem Lemma.

Laufzeit:

Zeile 1: $\Theta(m \cdot log(m))$

Zeile 2a und 2b: $\Theta(m \cdot log(m))$, falls eine Ausführung in $\Theta(log(m))$ möglich ist.

insgesamt dann: $\Theta(m \cdot log(m))$

8.8.5 Einschub: UNION-FIND-Problem (als ADT)

gegeben: eine feste Menge $S = \{s_1, ..., s_n\}$.
Betrachte Partionen von S, d.h. disjunkte Zerlegungen $S \dot\cup S_1 \dot\cup S_2 \dot\cup ... \dot\cup S_k$

Operationen:

FIND(x): für $x \in S$: finde das S_i $(i = 1, ..., k)$, das x enthält

UNION(S_i, S_j): vergrößert die bisherige Partition durch Vereinigung der Klassen S_i und S_j

Anwendungen:

- **Algorithmus von Kruskal**, mit S = Menge der Knoten und Partition als Zusammenhangskomponente der bisher aufgenommenen Kanten. Im Algorithmus selbst würde dann in Zeile 2a FIND() zweimal und in Zeile 2b UNION() einmal aufgerufen werden. Zu Beginn wäre dann jeder Knoten eine Klasse für sich.
- **Bildverarbeitung:** „Segmentierung“: finde die Menge von Pixeln, die 'zusammen gehören' und einen homogenen Teil des Bildes darstellen.
- **EQUIVALENCE(X,Y) in FORTRAN:** bedeutet: X,Y bezeichnen den gleichen Speicherplatz.

Datenstruktur für UNION-FIND: Die Klassen $S_1, ..., S_k$ der Partition werden mit „Repräsentanten“ gleichgesetzt, d.h. S_j mit einem festen Element $x \in S_j$. Die Elemente von S_j sind in dem den Knoten eines Baums, dessen Wurzel, den Repräsentanten x enthält. Der Baum enthält zudem Zeiger von den Kindern zu den jeweiligen Vätern.

Bsp:

$$\begin{aligned} &\text{Sei } S = \{1, 2, ..., 10\} \text{ und} \\ &S_1 = \{\underline{3}, 5, 6\} \\ &S_2 = \{1, \underline{2}, 7\} \\ &S_3 = \{8, 9, \underline{10}\} \\ &S_4 = \{\underline{4}\} \end{aligned}$$

FIND(x) soll Repräsentanten der Klasse liefern, die x enthält. In unserem Beispiel liefert der Aufruf von FIND(7) die Zahl 2. Die Laufzeit von FIND() entspricht dann der Höhe des Baums $h \to O(h)$.

Bei UNION(r,s) wird der Baum mit dem Repräsenanten s zum Unterbaum von dem mit dem Repräsentanten r, sodass die vereinigte Menge dann den Repräsentanten r hat. Die Laufzeit dieser Operation beträgt $O(1)$, da nur ein Zeiger gesetzt werden muss.

Um die Höhe des Baums klein zu halten: „UNION-by-height“. Bei UNION-Operation: hänge den Unterbaum mit kleinerer Höhe an Wurzel dessen mit größerer (dazu merkt man sich die Höhen bei der Wurzel). Sind die Höhen gleich, ist es egal, welcher Baum an welche Wurzel gehängt wird.

Behauptung: Beginnt man mit der Partition $\{x_1\}, \{x_2\}, ..., \{x_n\}$ und benutzt man UNION-by-height, so ist die Höhe eines entstehenden Baums mit r Knoten stets $\leq log(r)$.

Beweis: Induktion über die Zahl s der UNION-Operationen.

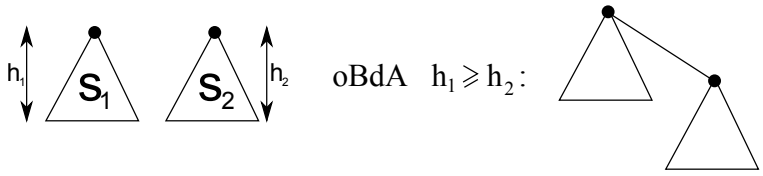
Induktionsanfang: s = 0

jede Klasse der Partition hat 1 Element und jeder Baum hat Höhe 0:

$$0 \leq log(1) \quad \checkmark$$

Induktionsschritt: $s \to s+1$

Die $s+1$-te UNION-Operation vereinigt die Mengen S_1 und S_2, wobei $|S_1| = r_1$ und $|S_2| = r_2$



Fall 1: $h_1 > h_2 \Rightarrow h = h_1$

$$2^h = s^{h_1} \underset{\text{I.V.}}{\leq} r_1 < r_1 + r_2 = r$$

Fall 2: $h_1 = h_2 \Rightarrow h = h_1 + h_2$

$$2^h = 2 \cdot 2^{h_1} = 2^{h_1} + 2^{h_2} \underset{\text{I.V.}}{\leq} r_1 + r_2 = r$$

Fazit: beginnend mit Partition $\{x_1\}, \{x_2\}, ..., \{x_n\}$ und UNION-by-height, kostet UNION-Operation $O(1)$ und FIND $O(log(n))$.

8.8.6 Anwendung von UNION-FIND auf Algorithmus von Kruskal

Pro Kante machen wir zwei FIND-Operationen und eventuell eine UNION-Operation, was uns eine Laufzeit von $O(m \cdot log(n))$ einbringt, da wir $m = |E|$ Kanten haben.

bei FIND(x): Sammle Knoten entlang des Pfades von x zur Wurzel und mache alle zu (direkten) Kindern der Wurzel.

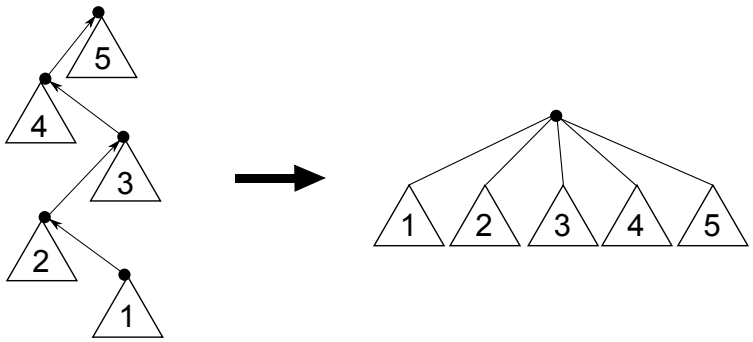


Abbildung 50: *Pfadkompression*

Man kann zeigen, dass beginnend mit $\{x_1\}, \{x_2\}, ..., \{x_n\}$, m UNION-Operationen $O(m \cdot \alpha(m, n))$ Zeit kosten. Man hat also eine (amortisierte) Laufzeit $\alpha(m, n)$, wobei $\alpha(m, n)$ ist die „inverse Ackermann-Funktion“ darstellt[18].

Tower-Funktion: $t : n \mapsto 2^{2^{\cdot^{\cdot^{2}}}}$

$$
\begin{aligned}
t(1) &= 2 \\
t(2) &= 4 \\
t(3) &= 16 \\
t(4) &= 2^{16} = 65536 \\
t(5) &= 2^{65536}
\end{aligned}
$$

Es gilt $\alpha(m, n) \leq 4$ für $m \geq n$ und $n \leq t(16)$

[18] eigentlich hat diese Funktion keine Inverse, da sie injektiv ist. Bsp: $\lim_{m \to \infty} \alpha(m, 1) = \infty$ aber extrem langsames Wachstum.

9 Geometrische Algorithmen

Anwendungsgebiete:

- Computergrafik, z.B. Entfernung verdeckter Kanten/Flächen/Ecken
- CAD
- GIS
- Computer-Vision

9.1 Geometrie im $\mathbb{R}^2$

Punkt: Ein Punkt lässt sich bspw. durch kartesische Koordinaten (x, y) darstellen.

Gerade: Eine Gerade g hat folgende „klassische Darstellung“:

$$y = ax + b \quad \text{wobei gemeint ist: } g = \{(x, y) | y = ax + b\}$$

z.B.

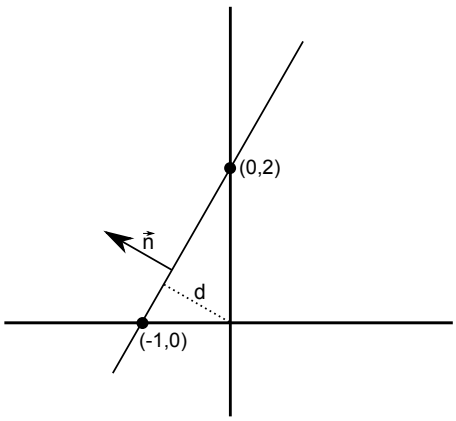


Abbildung 51: *Gerade zur Gleichung $y = 2x + 2$. Ein Problem bei dieser Darstellungsart besteht darin, dass vertikale Geraden, wie zum Beispiel $\{(x, y)_{\in \mathbb{R}^2} \mid x = 1\}$, nicht darstellbar sind.*

„Allgemeine“ Form:

$$Ax + By + C = 0$$

z.B.:

$$3x - y + 2 = 0$$

Eine vertikale Gerade mit bspw. $A = 2$, $B = -1$ und $C = 2$ wäre zwar darstellbar, jedoch wäre die Darstellung dann nicht mehr eindeutig. $4x - 2y + 4 = 0$ bspw. würde die gleiche Gerade beschreiben.

9.1.1 Hesse'sche Normalform

$$\overrightarrow{n} \bullet \begin{pmatrix} x \\ y \end{pmatrix} - d = 0$$

wobei $\overrightarrow{n}$ der Normaleneinheitsvektor, $\bullet$ der Operator für die Skalarmultiplikation und d der Abstand der Geraden zum Koordinatenursprung ist[19]. Die Hesse'sche Normalform könnte man auch wie folgt darstellen:

$$-\overrightarrow{n} \bullet \begin{pmatrix} x \\ y \end{pmatrix} - (-d) = 0$$

Durch diese zwei Darstellungen ergeben sich zwei mögliche Orientierungen der Geraden:

Eine orientierte Gerade unterteilt eine Ebene in zwei Halbebenen, jeweils „links " und „rechts " der Geraden, wobei für diese gilt:

$$\overrightarrow{n} \bullet \begin{pmatrix} x \\ y \end{pmatrix} - d \geq 0 \quad \text{„(linke) abgeschlossene" Halbebene}$$
$$\overrightarrow{n} \bullet \begin{pmatrix} x \\ y \end{pmatrix} - d > 0 \quad \text{„(linke) abgeschlossene" Halbebene}$$

Die obigen Gleichungen gelten analog für rechte Halbebenen mit umgedrehten $\leq$ bzw. $<$.

im Bsp: Der Vektor $\binom{0}{2} - \binom{-1}{0} = \binom{1}{2}$ zeigt in Richtung der Geraden. Demnach steht $\binom{-2}{1}$ senkrecht dazu. Bringt man eben diesen Vektor auf Einheitslänge, erhält man für den Normaleneinheitsvektor:

$$\overrightarrow{n} = \frac{1}{\sqrt{5}} \cdot \begin{pmatrix} -2 \\ 1 \end{pmatrix}$$

Einsetzen in die Hesse'sche Normalform liefert uns dann einen Wert für d:

$$\overrightarrow{n} \bullet \begin{pmatrix} -2 \\ 1 \end{pmatrix} - d = 0 \Rightarrow d = \frac{2}{\sqrt{5}}$$

Somit sieht dann das Endresultat in der Hesse'schen Normalform wie folgt aus:

$$\underline{\frac{1}{\sqrt{5}} \begin{pmatrix} -2 \\ 1 \end{pmatrix} \bullet \begin{pmatrix} x \\ y \end{pmatrix} - \frac{2}{\sqrt{5}} = 0}$$

Setzt man andere Punkte ein, die nicht auf der Gerade liegen, so erhält man für d Werte $\neq 0$. Stattdessen bekommt man jeweils den Abstand des entsprechenden Punktes zur Geraden.

[19] Beachte, dass der Abstand in dem Fall auch negativ sein kann.

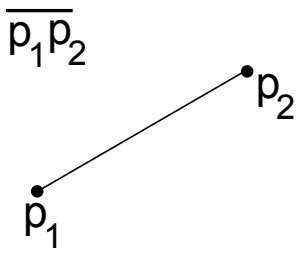
In konstanter Zeit berechenbar:

1. liegt Punkt p rechts, links oder auf der Geraden g
2. Abstand eines Punktes $p = (x, y)$ von der Geraden g: $|\overrightarrow{n} \bullet \binom{x}{y} - d|$
3. Schnittpunkt zweier Geraden

$$\left.\begin{aligned} \overrightarrow{n_1} \bullet \begin{pmatrix} x \\ y \end{pmatrix} - d_1 = 0 \\ \overrightarrow{n_2} \bullet \begin{pmatrix} x \\ y \end{pmatrix} - d_2 = 0 \end{aligned}\right\} \quad \text{lineares Gleichungssystem mit zwei Unbekannten.}$$

4. Gerade durch 2 Punkte $p_i = (x_i, y_i)$ mit $i = 1, 2$:
   $p_2 - p_1 = \binom{a}{b}$ ist die eigentliche Differenz der Ortsvektoren von p_1 und p_2. Der Vektor $\binom{b}{-a}$ steht dann senkrecht auf der Geraden und $\overrightarrow{n} = \frac{1}{\sqrt{a^2+b^2}} \cdot \binom{b}{-a}$ ist der Normaleneinheitsvektor. d kann dann letztendlich durch Einsetzen von p_1 in die Hesse'sche Normalform ermittelt werden.

Strecke: Stück der Geraden durch p_1 und p_2, das zwischen diesen beiden Punkten liegt.



$$\overline{p_1 p_2} = \{\lambda p_1 + (1 - \lambda) p_2 | \lambda \in [0, 1]\}$$

Dies gilt analog für eine Gerade g durch p_1 und p_2 mit $\lambda \in \mathbb{R}$.

Test, ob $q \in \overline{p_1 p_2}$:

- teste, ob q auf Geraden durch p_1, p_2 liegt
- falls ja: teste, ob $||q - p_1|| \leq ||p_2 - p_1||$ und $q - p_1$ und $p_2 - p_1$ in die gleiche Richtunge zeigen.

Schnittpunkt: Ein Schnittpunkt zweier Strecken existiert genau dann, wenn:

1. Schnittpunkt s zwischen zugeh. Geraden existiert

   und

2. s liegt auf beiden Strecken

oder

3. Endpunkte der einen Strecke liegen auf unterschiedlichen Seiten der jeweils anderen und umgekehrt.

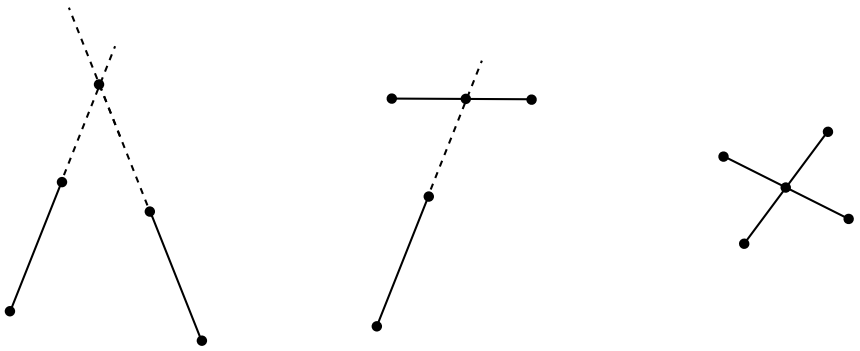
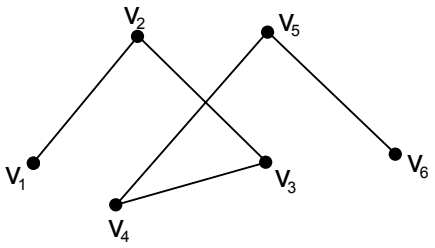


Abbildung 52: *Die verschiedenen Möglichkeiten zur Schnittpunktbildung.*

Polygonzug: Ein Polygonzug[20] ist eine Folge von Strecken $\overline{v_0v_1}$ $\overline{v_1v_2}$ ... $\overline{v_{n-1}v_n}$, die durch die Folge $v_0, v_1, ..., v_n$ der Ecken des Polygonzugs repräsentiert werden kann. Die Strecken zwischen den Ecken werden dann als Kanten bezeichnet.



Polygon: Ein Polygon ist ein „geschlossener“ Polygonzug $\overline{v_0v_1}$ $\overline{v_1v_2}$ ... $\overline{v_{n-1}v_n}$ $\overline{v_nv_0}$

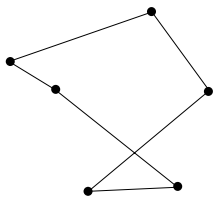


Abbildung 53

einfacher Polygonzug: $\overline{v_0v_1}$ $\overline{v_1v_2}$ ... $\overline{v_{n-1}v_n}$ mit:

$$\overline{v_iv_{i+1}} \cap \overline{v_jv_{j+1}} = \begin{cases} v_j, \text{ falls } j = j+1 \text{ oder } i+1 = n,\ j = 0 \\ \varnothing \text{ sonst} \end{cases}$$

[20] Andere Bezeichnungen dafür wären Streckenzug und polygonal chain

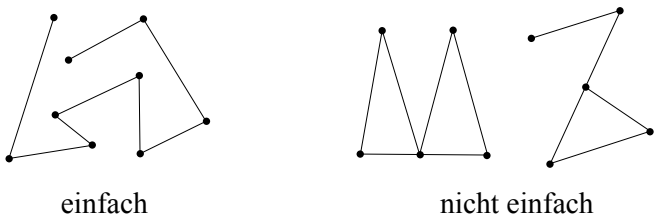


Abbildung 54: *Links ein einfacher Polygonzug und rechts zwei nicht-einfache Polygonzüge.*

einfaches Polygon: analog zu einfachem Polygonzug

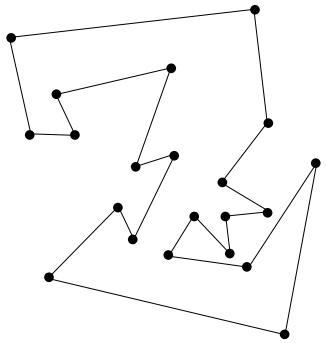


Abbildung 55: *Eine Polygon ohne „Selbstüberschneidungen“ bezeichnet man als einfaches Polygon. Dementsprechend zeigt Abbildung 53 ein nicht-einfaches Polygon.*

Entfernt man die Punkte auf den Strecken eines einfachen Polygons aus der Ebene, erhält man zwei Teile, wobei einer beschränkt (innerer Teil des Polygons) und der andere unbeschränkt (äußerer Teil des Polygons) ist.

9.1.2 Punkt in Polygon

Problem: Gegeben sind ein Punkt p und ein einfaches Polygon π mit n Ecken. Wie lässt sich herausfinden, ob p im Inneren von π liegt.

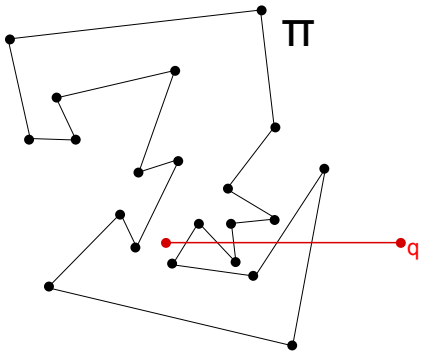


Abbildung 56: *Um zu überprüfen, ob ein Punkt p im Polygon π liegt, kann von diesem Punkt aus eine Linie zu einem imaginären Punkt q (von dem man sich sicher ist, dass dieser weiter weg ausserhalb des Polygons befindet) ziehen und alle Schnitte der Strecke $\overline{pq}$ mit π zählen. Ist diese Anzahl ungerade, befindet sich p innerhalb von π und sonst ausserhalb.*

Algorithmus: Wir betrachten vom Punkt p aus eine Strecke in horizontaler Richtung $\overline{pq}$, sodass q auf jeden Fall außerhalb vom Polygon π liegt. Dies kann gewährleistet werden, wenn die x-Koordinate von q größer als alle x-Koordinaten der Ecken von π ist.
Wir schneiden nun die Strecke $\overline{pq}$ mit allen Kanten von π und zählen die Schnittpunkte. Ist die Anzahl derer gerade, befindet sich p innerhalb von pi und bei ungerader Anzahl außerhalb.

Beim Zählen der Schnittpunkte muss man jedoch bei bestimmten Fällen aufpassen:

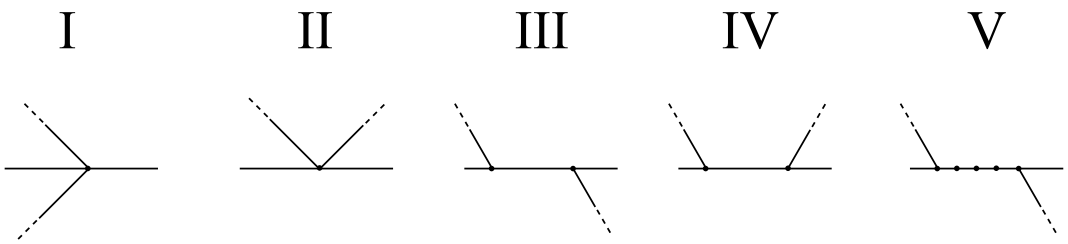


Abbildung 57: *Bei Betrachtung der Schnittpunkte müssen verschiedene Fälle berücksichtigen: [I] zählt als 1, [II] als 0, [III] als 1 und [IV] wieder als 0 Schnittpunkte. In Fall [V] werden die sich überlappenden Kanten ignoriert und das Ganze als 1 Schnittpunkt gewertet.*

Laufzeit:

$O(n)$ zur Bestimmung von q

$O(n)$ zum Schnitt aller Kanten mit $\overline{pq}$

insgesamt: $O(n)$

Man kann auch π vorverarbeiten und eine Datenstruktur anlegen, sodass Anfragen mit Punkt $q \in \mathbb{R}^2$ und ob dieser in π liegt effizient beantwortet werden können. Es ist möglich in $O(n \cdot log(n))$ Vorverarbeitungszeit $O(n)$ Platz und $O(log(n))$ Anfragezeit zu benötigen. Dies ist möglich für

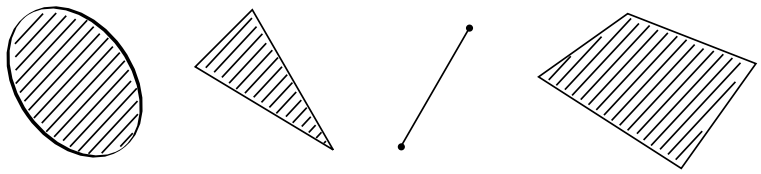
beliebige planare Unterteilungen der Ebene durch Strecken, Geraden und Strahlen[21]. In diesem Fall kann man eine Anfrage nach dem Gebiet tätigen, welches q enthält.

[21] einseitig beschränkte Geraden

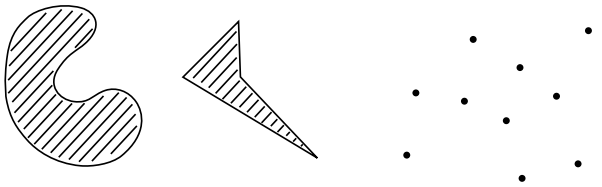
9.1.3 Konvexe Hülle

Definition: Menge $A \subseteq \mathbb{R}^2$ heißt genau dann konvex, wenn für alle $p, q \in A$ die Strecke $\overline{pq} \subseteq A$ ist.

Beispiele für konvexe Mengen:



Beispiele für nicht-konvexe Mengen:



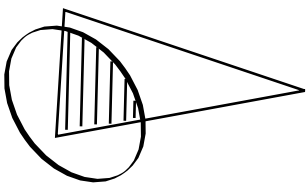
Bemerkung: Der Durchschnitt beliebig vieler konvexer Mengen ist konvex.

Definition: Sei $A \subseteq \mathbb{R}^2$, dann heißt $CH(A) := \bigcap_{\substack{B \text{ konvex} \\ A \subseteq B}} B$ die konvexe Hülle von A.

Bem.:

a) Ist B konvex und $A \subseteq B$, dann ist $CH(A) \subseteq B$

b) $A \subseteq CH(A)$



Bsp:

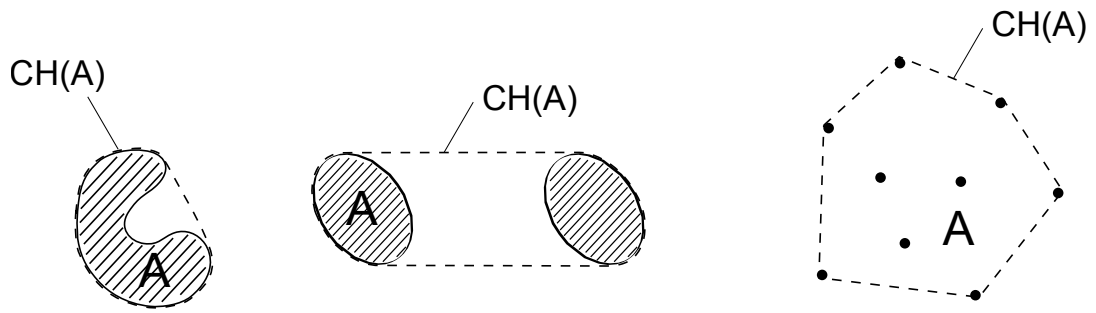


Abbildung 58: *Die konvexe Hülle kann man sich als ein Gummiband vorstellen, welches um die Menge gelegt wird und diese dann eng umspannt.*

Bem (ohne Beweis): Die konvexe Hülle einer endlichen Menge A ist ein konvexes Polygon (einschließlich seines Inneren), dessen Ecken Punkte von A sind. Diese Punkte heißen dann extreme Punkte von A.

Bem: Ein Punkt p von A ist extrem $\Leftrightarrow$ es gibt eine Gerade g durch p, sodass alle Punkte von A in einer der beiden (abgeschlossenen) Halbebenen bezüglich g liegen.

9.1.4 Graham-Scan

Algorithmus zur Berechnung der konvexen Hülle einer endlichen Menge.

gegeben: endliche Menge $A \subseteq \mathbb{R}^2$, $A = \{p_1, ..., p_n\}$

berechne: $CH(A)$, d.h. Ecken dieses Polygons $q_1, ..., q_k$ (z.B. im Gegenuhrzeigersinn)

Algorithmus:

1. finde Punkt $p_0 \in A$ mit der kleinsten x-Koordinate. Falls es mehrere davon gibt, bestimme den mit der kleinsten y-Koordinate und setze p_0 auf diesen Punkt.
2. sortiere die restlichen Punkte p_i gemäß der Steigung von $\overrightarrow{p_0 p_i}$. Wenn mehrere die gleiche Steigung haben, entferne alle bis auf den mit dem größten Abstand. Nenne sortierte Folge $p_1, ..., p_l$
3. PUSH p_0 und p_1 auf einen Stack.
4. für $i = 2, ..., l$: betrachte Folge aus zweitobersten, obersten Punkt auf Stack und p_i. falls diese Folge eine Rechtskurve bildet: POP, wiederhole diesen Schritt. Sonst: PUSH p_i
5. gib Folge auf dem Stack aus als $CH(A)$.

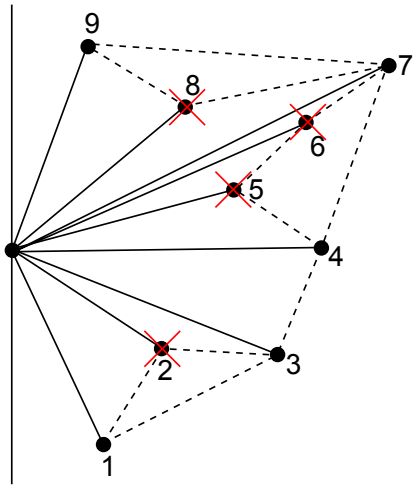


Abbildung 59: *Graham-Scan anhand eines Beispiels.*

Laufzeit:

Schritt 1: $O(n)$

Schritt 2: $O(n \cdot log(n))$

Schritt 3: $O(1)$

Schritt 4: $O(n)$, denn jeder Punkt wird einmal auf dem Stack abgelegt und höchstens einmal heruntergenommen

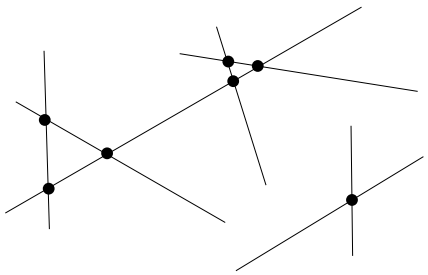
insgesamt: $\boxed{O(n \cdot log(n))}$

Alternativer Algorithmus: Anwendung der Divide-and-Conquer-Strategie und rekursive Bestimmung von konvexen Hüllen von den Teilmengen.

9.1.5 Schnittpunkte von Strecken

gegeben: n Strecken $s_1, ..., s_n$ in „allgemeiner Lage“ mit $s_i = \overline{p_i q_i}$

finde: alle Schnittpunkte von Strecken



Die Anzahl k der Schnittpunkte kann $\Theta(n^2)$ sein, wenn sich also alle Strecken schneiden (z.B. wenn diese als Gitter angeordnet). Aus diesem Grund ist der Brute-Force-Algorithmus nicht mehr asymptotisch zu verbessern. Was aber, wenn $k \ll n^2$, also k sehr klein gegenüber n ist? In der Praxis ist dies oft der Fall. Wir werden einen Algorithmus sehen, der effizient in den Parametern n und k ist und eine Laufzeit von $O((n+k) \cdot log(n))$ hat. Solche Algorithmen werden als „Ausgabesensitiv“ (engl: output sensitive) bezeichnet. Die Laufzeit hängt also von der Größe der Ausgabe ab.

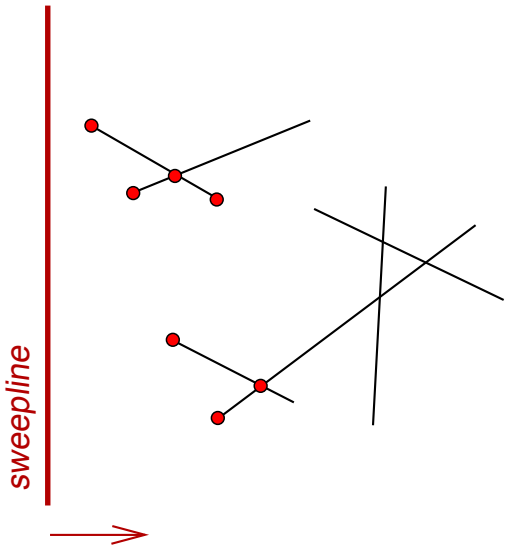


Abbildung 60: *Beispielhafter Ablauf der Schnittpunktbestimmung von Strecken mittels Sweepline.*

„sweep line status“ - SLS: Folge von Strecken, die Sweep Line (SL) schneiden, in der Ordnung von unten nach oben. Wo ändert sich SLS?

1. an linken Endpunkten von Strecken $\rightarrow$ neue Strecke s in SLS einfügen
2. an rechten Endpunkten $\rightarrow$ eine Strecke aus SLS streichen
3. bei Kreuzungspunkten $\rightarrow$ zwei Strecken in der Ordnung vertauschen.

Solche x-Werte bezeichnet man als Ereignispunkte. Diese werden wir in einer Datenstruktur EPS (event point schedule) ablegen. Eine geeignete Datenstruktur für SLS wäre irgendeine Wörterbuch-Datenstruktur, wie zum Beispiel AVL-Baum. Für den EPS kann man eine Prioritätswarteschlangen-Datenstruktur, wie zum Beispiel Heap, heranziehen, wobei mit den Endpunkten der Strecken initialisiert wird und die Prioritäten durch die x-Koordinaten gegeben sind.

Beim Einfügen einer Strecke s in SLS testen wir für beide Nachbarn von s im SLS, ob sie s schneiden. Ist dies der Fall fügen wir den jeweiligen Schnittpunkt der Ergebnismenge und dem EPS hinzu. Überstreicht die Sweep Line einen Kreuzungspunkt, entstehen neue Nachbarschaften (siehe Abb. **??**). Für die nun benachbarten Strecken testen wir ob sich Schnittpunkte rechts von

der Sweepline befinden. Wenn ja, werden die Kreuzungspunkte in die Ergebnismenge und in den EPS aufgenommen.

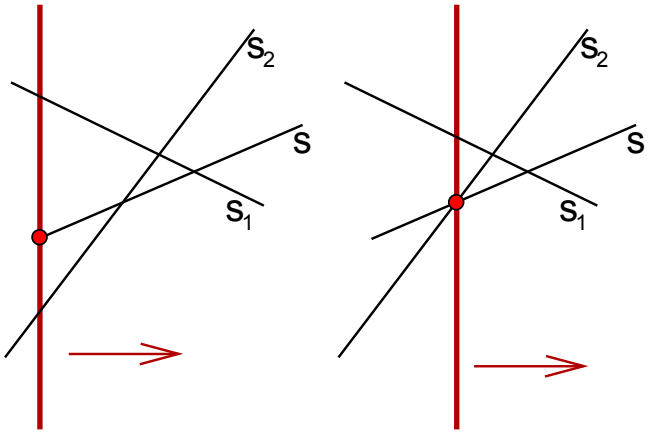


Abbildung 61: *Beim Überstreichen eines Kreuzungspunktes werden die Nachbarn vertauscht, in dem Fall s und s_1.*

Beim Entfernen einer Strecke s aus SLS, testen wir ob die neuen Nachbarn einen Schnittpunkt (rechts von der Sweepline) haben. Wenn ja, gehen wir vorher vor.

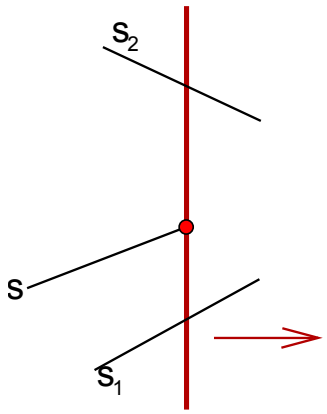


Abbildung 62: *Da die Sweepline den Endpunkt von s überstreicht, sind nun s_1 und s_2 nun Nachbarn.*

Laufzeit Für jeden Ereignispunkt eventuell:

- $O(1)$ - neue Schnittpunkte bestimmen ≤ 2
- $O(log(n))$ - in SLS (AVL-Baum): einfügen, streichen oder Reihenfolge von 2 Elementen vertauschen
- $O(log(n))$ - in EPS (Heap): Min Streichen und eventuell (≤ 2) einfügen.

Zahl der Ereignispunkte:

$$2n + k$$

wobei n die Zahl der Endpunkte und k die Anzahl der Schnittpunkte ist.

Fazit: Insgesamt erhalten wir also eine Laufzeit von $O((n + k) \cdot log(n))$. Der Test, ob es einen Schnittpunkt gibt, lässt sich in $O(n \cdot log(n))$ bewerkstelligen.

Probleme: Bei Eingaben außerhalb der „allgemeinen Lage“ (siehe Abb. **??**) kann es zu Problemen kommen, sodass diese Fälle gesondert betrachtet werden müssen. Diese Probleme sind jedoch ohne Vergrößerung der asymptotischen Laufzeit lösbar.

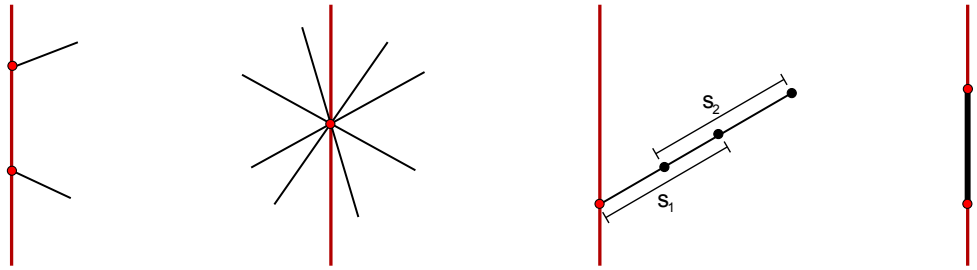


Abbildung 63: *Beispiele für Eingaben in nicht allgemeiner Lage.*

Der Sweepline-Ansatz ist auch viele anderer Probleme hilfreich, zum Beispiel zur Bestimmung des Randes einer Menge von achsenparallelen Rechtecken (siehe Abb. 64).

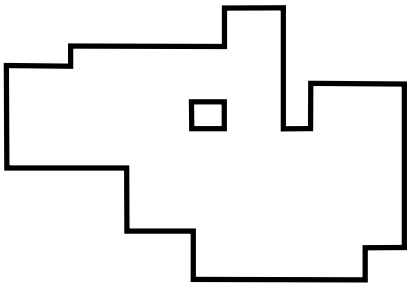
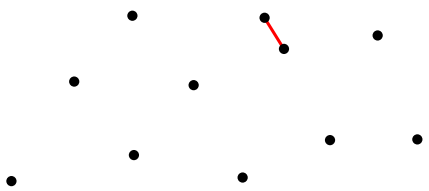


Abbildung 64

9.1.6 Nächstes Paar

gegeben: Menge von Punkten $p_1, ..., p_2$

finde: Paar von Punkten mit dem kürzesten Abstand.



Lösungsansatz: Mit der Brute-Force-Methode wäre dieses Problem in $O(n^2)$ lösbar. Hier bietet sich jedoch ein Divide & Conquer-Algorithmus an, um das Paar noch schneller zu finden.

9.1.7 Divide & Conquer-Algorithmus

Für jeden rekursiven Aufrug wird aufrechterhalten, dass wir eine Menge P von Punkten und die Felder X und Y, in denen die Punkte aus P nach x-Koordinate bzw. y-Koordinate sortiert sind, haben. Ab einer bestimmten Anzahl von Elemente in P, kann man vereinbaren wieder die Brute-Force-Methode anzuwenden:

- falls $|P| \leq 3$: arbeite mit Brute-Force
- sonst: bestimme die Felder X und Y

gegeben: P, X, Y

Bei diesem Algorithmus bestimmen wir erst die Punktmengen P_L ($\lfloor \frac{n}{2} \rfloor$ Punkte von P mit kleinerer x-Koordinate), P_R ($\lceil \frac{n}{2} \rceil$ Punkte von P mit größerer x-Koordinate) und die Felder X_L, X_R, Y_L und Y_R. Der Algorithmus wird dann rekursiv aufgerufen mit (P_L, X_L, Y_L) und (P_R, X_R, Y_R), wobei jedes Mal Paare p_L, q_L (nächste Nachbarn in P_L) und p_R, q_R (nächste Nachbarn in P_R) mit den jeweiligen kürzesten Abständen δ_L und δ_R geliefert werden. Insgesamt ergibt sich dann der kürzeste Abstand durch einen Vergleich von δ_L und δ_R:

$$\text{Sei } \delta = min(\delta_L, \delta_R)$$

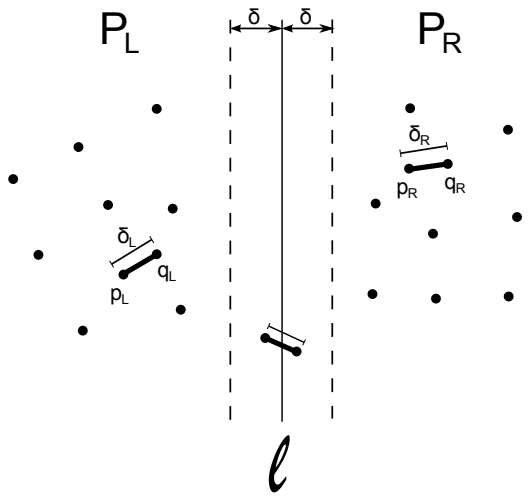


Abbildung 65: *Möglichkeiten für nächste Paar in P: p_L, q_L oder p_R, q_R oder ein $p \in P_L$ und ein $q \in P_R$. Es ist jedoch nicht möglich eine kürzeste Verbindung zwischen zwei Punkten zu finden, bei der sich ein Punkt links bzw. rechts vom δ-Streifen befindet und der andere jeweils rechts bzw. links von l.*

Durchlaufe Y-Liste für jeden Punkt p, der im δ-Streifen um die Trennlinie l liegt. Betrachte die 7 nächsten Punkte[22] auch im Streifen und ihren Abstände zu p. Falls dabei Paare mit Abstand $< \delta$ vorkommen, gib das mit dem kleinsten Abstand aus, sonst das mit Abstand δ (also (p_L, q_L) oder (p_R, q_R)). Ist $|P| < 3$, bestimmen wir den kürzesten Abstand direkt, sonst führen wir den obigen Algorithmus aus.

Korrektheit: Fall es im Streifen zwei Punkte $p \in P_L$ und $q \in P_R$ mit einem Abstand $< \delta$ gibt, so liegen sie in einem Rechteck mit Seitenlängen δ und 2δ.

Die linke Hälfte des Rechtecks (Quadrat) enthält höchstens 4 Punkte aus P_L (Übung). Analog gilt dies für die rechte Hälfte. Insgesamt können also nur maximal 8 Punkte (die 2 Punkte in der Mitte werden doppelt gezählt) im ganzen Rechteck enthalten sein. Zu jedem im Streifen nächsten 7 höheren im Streifen betrachten liefert Minimum.

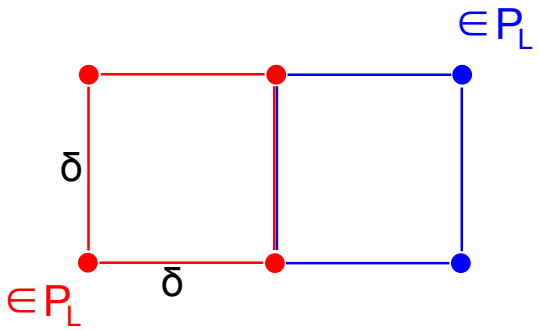


Abbildung 66

[22] Grund für die '7' wird später aufgeklärt

Laufzeit: Im folgenden der Algorithmus nochmal in Kurzform:

1. Listen X, Y: P sortiert nach x- und nach y-Koord
2. Mengen P_L und P_R als linke bzw. rechte Hälften von P, Gerade l trennt P_L und P_R
3. bestimme X_L, Y_L, X_R und Y_R
4. bestimme rekursiv δ_L und δ_R
5. $\delta := min(\delta_L, \delta_R)$
6. gehe im δ-Streifen um l Punkte von unten nach oben durch (mittels Y)
7. für jeden: bestimme Abstand zu den nächsten 7, merke das gesehene Minimum: falls es $< \delta$, gib es aus; sonst: gib δ aus

$T(n)$ für die rekursive Prozedur (Schritte (2) - (7))

Schritt (2),(3): $O(n)$ Liste Y durchlaufen, um Y_L und Y_R zu bilden

Schritt (4): $2 \cdot T(\frac{n}{2})$ (n gerade)

Schritt (5): $O(1)$

Schritt (6),(7): $O(n)$

$$\text{Rekursionsgleichung: } T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$$
$$\Rightarrow T(n) = O(n \cdot log(n)) \text{ (siehe Mergesort)}$$

Schritt (1): sortieren: jeweils einmal nach x- und y-Koordinate $\Rightarrow O(n \cdot log(n))$

Somit beträgt die Laufzeit insgesamt also $\underline{O(n \cdot log(n))}$

9.2 Einschub: Dynamisches Programmieren

- Brute-Force-Algorithmen
- Divide & Conquer
- Greedy-Algorithmen (Prim, Dijkstra)
- Dynamisches Programmieren (Floyd-Warshall, Berechnung der transitiven Hülle)

- meistens bei Optimierungsaufgaben, braucht:
  - optimale Gesamtlösung enthält optimale Lösungen für Teilprobleme (Belman'sches Optimalitätskriterium)
  - Teilprobleme „überlappen sich“ (im Unterschied zu Divide & Conquer!)

Eine DP-Lösung besteht aus 4 Hauptschritten:

1. Charakterisiere Struktur einer optimalen Lösung
2. Definiere rekursiv Wert der optimalen Lösung
3. berechne diesen Wert bottom-up
4. konstruiere eine optimale Lösung, die diesen optimalen Wert tatsächlich realisiert

Problem 1: longest common subsequence (lcs)

gegeben:

- Σ endl. Alphabet
- $X[1,...,m], Y[1,...,n] \in \Sigma^+, m \leq n$

gesucht: lcs von X und Y (eine, alle, ...)

Bsp:

$$X = ABCBDAB$$
$$Y = BDCABA$$
$$\Rightarrow lcs = BCBA$$

1. Ansatz: Triviale Brute-Force-Lösung
Prüfe jede Teilsequenz von Y gegen X:

$$O(2^n \cdot m)$$

2. Ansatz:

- Berechnen zunächst Länge der lcs
- Betrachte Präfix von X gegen Präfix von Y, Sei $c(i,j) :=$ Länge einer lcs von $X[1,...,i]$ und $Y[1,...,j]$. Wir suchen eigentlich $c(m,n)$

Satz (Rekursive Struktur der Lösung):

$$\forall i,j: \quad c(i,0) = c(0,j) = 0$$
$$\forall i,j \geq 1:$$
$$c(i,j) = \begin{cases} c(i-1,j-1)+1, & X[i] = Y[i] \\ max\{c(i-1,j), c(i,j-1)\}, & sonst \end{cases}$$

$\rightarrow$ Damit Belman'sches Kriterium erfüllt

Lösen Rekursion top-down:

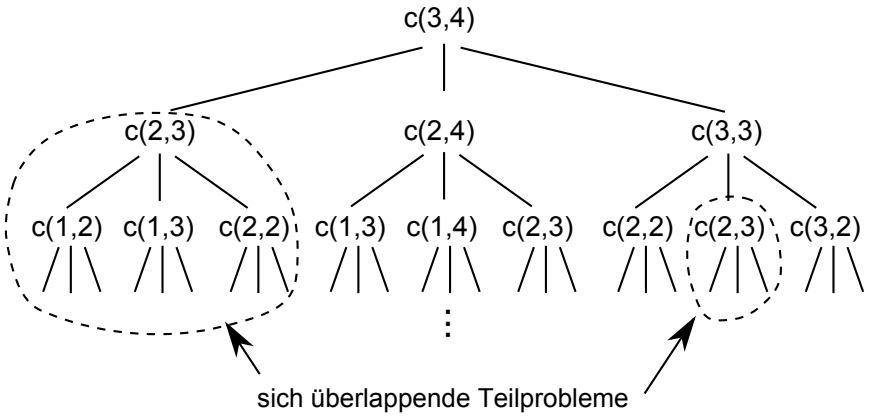
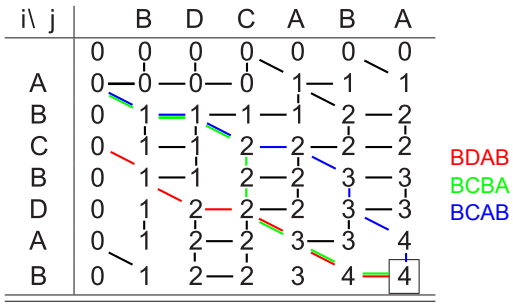


Abbildung 67: *Im ersten Moment könnte man auf die Idee kommen jeden einzelnen Knoten zu berechnen, was einen Gesamtaufwand von $3^{\Theta(m+n)}$ entsprechen würde. Da Knoten jedoch mehrfach auftreten und sich die Teilprobleme somit überlappen, reduziert sich der Aufwand enorm, wenn bereits berechnete Knoten herangezogen werden.*

Frage: Wieviele von den überlappenden Teilproblemen gibt es eigentlich?

Antwort: Nur $\sim n \cdot m$ viele! Und wir berechnen alle davon.



- Lösungen, die optimalen Wert c(7,6) realsieren, findet man durch back-tracking
- Aufwand:

  Zum Brechenen des Wertes c(m,n)

  Zeit: $(m+1)\cdot(n+1)$ Einträge ausrechnen, wobei jeder Eintrag $O(1)$ $\Rightarrow$ insgesamt: $O(n\cdot m)$

  Speicherbedarf zur Berechnung von $c(m,n)$: $O(min(n,m))$ $\rightarrow$ Zeilen-/Spaltenweise Ausfüllen, wodurch jedoch back-tracking dann nicht mehr möglich ist.

Frage: Geht es schneller?

Antwort: Ja, mit dem sogenannten „4-Russen-Trick“ in $O(\frac{n^2}{log(n)})$

Aufwand um die *lcs* zu finden:

Zeit:	$O(n\cdot m)$
Speicher:	$O(n\cdot m)$

Mit dem sog. „Hirschberg-Algorithmus“ lässt sich der Speicherbedarf sogar auf $O(1)$ reduzieren.

war schon: Floyd-Warshall - „All-pairs-shortest-path“

Problem 2: Längste Wege in $dag's$

Wie? $\rightarrow$ topologisches Sortieren!

initialisieren: $l(i)$ = Länge eines längsten Weges der in Knoten i endet. (Zum Schluss max der $l(i)$ ausrechnen)

$$l(1) = 0, \text{ da Quelle}$$
$$l(i) = \begin{cases} max(l(j)+1), & \text{für } j < i \text{ und } \{j,i\} \in E \\ 0, & \text{sonst} \end{cases}$$

9.3 Subset-Sum und Knapsack-Probleme

gegeben:

- Objekte $\{1, 2, ..., n\}$ und nichtnegative Gewichte w_i
- Obergrenze W

gesucht: Finde $S \subseteq \{1, ..., n\}$ mit $w(S) = \sum_{i \in S} w_i \leq W$ und $w(S)$ maximal

greedy?

kleinstes zuerst:

$$\left\{ w_1 = 1, \frac{W}{2}, \frac{W}{2} \right\}$$

größtes zuerst:

$$\left\{ \frac{W}{2} + 1, \frac{W}{2}, \frac{W}{2} \right\}$$

$\Rightarrow$ **DP!** $OPT(i, W)$ optimale Lösung für $\{1, ..., i\}$ und Schranke W.

Rekursion:

$$OPT(i, W) = \begin{cases} OPT(i-1, W), & \text{falls } W < w_i \\ max\{OPT(i-1, W), w_i + OPT(i-1, W-w)\}, & \text{sonst} \end{cases}$$

	0	1	...	w
0				
1				
⋮				
n				OPT(n,w)

Tabelle 4: $\rightarrow O(n \cdot W)$, $W \propto 2^k$, *Darstellungsgröße k*

10 Einschub: Speicherverwaltung & Caching

10.1 Kurze Wiederholung

Java ist plattformunabhängig, d.h. der Übersetzer erzeugt „Pseudocode“ für eine virtuelle Maschine (JVM).

10.1.1 Speicherlayout

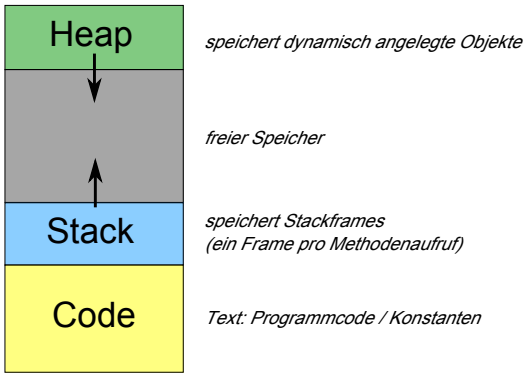


Abbildung 68: *klassisches Layout des Speichers*

10.1.2 Stack

Der Stack speichert für jeden Methodenaufruf einen sogenannten Stackframe.

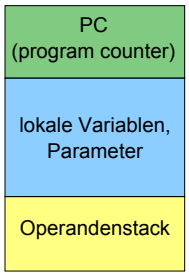


Abbildung 69

Java als 0-Adress-Maschine

MIPS: add \$ r_1,\$ r_2, \$ r_3 $\rightarrow$ $r_1 := r_2 + r_3$

x86: add EAX, EBX $\rightarrow$ $EAX := EAX + EBX$

add EBX $\rightarrow$ AKKUMULATOR = AKKUMULATOR + EBX

in Java: **iadd** $\rightarrow$ Arbeitet auf Stack: poppe 2 Elemente, addiere und pushe Ergebnis auf Stack

Java benutzt Call-by-Value-Semantik, d.h. die Parameter werden bei Aufruf kopiert.

10.1.3 Heap

Der Heap speichert dynamisch angelegte Objekte und besteht aus Blöcken. Daten werden auf dem Heap immer blockweise gespeichert.

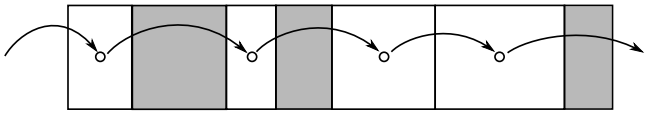


Abbildung 70: *Freelist: verkette Liste von freien Blöcken. Belegung im Speicher: laufe durch Freelist und suche nach zusammenhängenden freien Blöcken, die groß genug sind*

Strategien zur Speicherverwaltung

- first fit
- best fit
- worst fit
- next fit

Frage: Wie werden nun „abgewrackte“ Objekte beseitigt?

Antwort: In Java kümmert sich die Müllabfuhr (Garbage Collector) darum.

Frage: Wie erkennt man unnötige Objekte?

Antwort: Mittels Mark-and-Sweep-Algorithmus in 2 Phasen:

1. Markieren: finde alle Objekte, die von einem Stackframe aus erreichbar sind → Tiefensuche! Interessantes Feature von DFS:

   Schorr-Warte-Algorithmus: Tiefensuche ohne extra Stack: verändere Graphen, um Stack zu speichern (implizierter Stack spart Speicher)

2. Fegen: gehe Heap durch, füge alle unmarkierte Objekte zur Freelist hinzu

10.1.4 Java vs. C++

Java	C++
- keine Memory-Leaks	- Gefahr von Memory-Leaks, wenn man den Speicher nicht wieder explizit freigibt
- potentielle Performanceprobleme	- volle Kontrolle
- und noch vieles mehr...	hier auch...

10.2 Speicherhierarchie & Caching

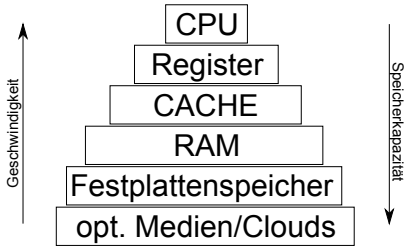


Abbildung 71: *Hierarchisierung der verschiedenen Speicherarten*

10.2.1 Cache ($)

- nutzt zeitliche und örtliche Lokalität aus
- unterteilen den großen Speicher in Blöcke/Seiten → Chache speichert eine bestimmte Anzahl von Seiten

Problem: Wenn neue Seite in Cache kommt und dieser voll ist, muss eine alte Seite daraus entfernt werden.

Ersetzungstrategien:

- LRU - Least Recently Used: entferne Seite für die letzter Zugriff am weitesten in der Vergangenheit liegt
- FIFO - First In First Out: Entferne Seite, die am längsten im Cache liegt
- Random: Entferne zufällig ausgewählte Seite

  usw.

11 Schwere Probleme

Einstiegsbeispiele:

[1]	15	23	2	[19]	30	4	13	[25]	3
[12]	92	[48]	37	53	8	24	[3]	7	17
32	[5]	14	18	6			1		

Tabelle 5: *Bsp: Finde Teilmengen, bei denen sich die Elemente zu 141 addieren.*

a)
gegeben: ungerichteter Graph $G = (V, E)$
Frage: Gibt es einen Kreis, der jeden Knoten enthält?
Antwort: Ja und zwar genau dann, wenn G zusammenhängend ist (mit Tiefensuche in $O(n+m)$)

b)
... genau einmal ... → „Hamilton'scher Kreis"
Antwort: Man weiß es nicht!

c)
gegeben: Folge von ganzen Zahlen $a_1, ..., a_n$ und Zahl b
Frage: Existiert Teilmenge von $\{a_1, ..., a_n\}$ deren Summe $= b$ ist? („Subset-Sum")
Antwort: Auch hier existiert kein effizienter Algorithmus

d)
gegeben: Boole'sche Formel ϕ
Frage: Gibt es eine Belegung der Variablen x_1, x_2 und x_3, sodass die Formel $x_1 \vee (x_2 \wedge \neg x_3)) \wedge \neg x_1 \wedge x_2$, „wahr" wird?
Antwort: Brute-Force! Eine mögliche Lösung wäre $x_1 = FALSE$, $x_2 = TRUE$ und $x_3 = FALSE$, jedoch gibt es keinen effizienten Algorithmus zu Betimmung einer solchen Lösung → „Erfüllbarkeitsproblem" (SAT - satisfiability problem)

e)
gegeben: Vollständiger, gerichteter Graph $G = (V, E)$, also $E = V \times V$ mit Kantengewichten $c : E \mapsto \mathbb{N}$
Finde: kürzeste Rundreise, d.h. Kreis, der jeden Knoten genau einmal besucht und unter diesen das geringste Gewicht hat. (TSP - traveling salesman problem)
Lösung: Hier existiert ebenfalls kein effizienter Algorithmus

f)
gegeben: Eine Zahl p in Binärdarstellung
Frage: Ist p eine Primzahl?
Lösung: Inzwischen existiert hier ein Algorithmus mit polynomieller Laufzeit

g)
gegeben: Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$
Frage: Sind die beiden Graphen isomorph, also existiert eine bijektive Abbildung $f : V_1 \mapsto V_2$ mit $(v_i, v_j) \in E_1 \Leftrightarrow (f(v_i, f(v_j)) \in E_2$?

11.1 Komplexitätsklasse NP

Betrachten Entscheidungsprobleme d.h. Probleme mit den Antworten "ja" oder "nein".
NP intuitiv: es gibt leicht nachzuprüfenden Beweise, falls die Antwort "ja" ist.
identifizieren: Teilmenge der Eingaben für die Antwort "ja" gilt mit einer formalen Sprache L

Definition: von NP
$L \in NP :\Leftrightarrow$ es gibt Algorithmus A polynomieller Laufzeit, Zahl $k \in \mathbb{N}$ mit $L = \{w \mid \exists$ Wort x mit $|x| \leq |w|^k$ und $A(w,x) = 1\}$, wobei x als „Zeuge" bezeichnet wird, $|x|$ die Länge der Zeugen polynomiell in der Eingabe, $A(w,x)$ ein Verifikationsalgorithmus und 1 die Ausgabe dieses Algorithmus ist.

z.B. Subset-sum $\in$ NP:

$w =$ Eingabe, Codierung von $a_1, ..., a_n, b$
$x =$ Teilmenge von $\{a_1, ..., a_n\}$, die sich zu b aufaddiert
$A =$ Algorithmus, der testes, ob Summe $= b$

11.1.1 Weitere Beispiele

CLIQUE

gegeben: (ungerichteter) Graph $G = (V, E)$, $k \in \mathbb{N}$

Frage: $\exists V' \subseteq V$ mit $|V'| = k$ und $\{u, v\} \in E$ $\forall u, v \in V'$?

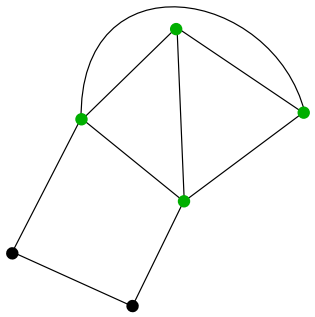


Abbildung 72: *Clique der Größe 4.*

CLIQUE $\in$ NP: w: Codierung eines Graphen $G = (V, E)$

Zeuge: x - Menge von Knoten $V' \subseteq V$

Verif.alg.: $A(w, x)$ - prüft nach, ob V' eine Clique der Größe k in G ist

UK unabhängige Knotenmenge

gegeben: $G = (V, E)$

Frage: $\exists V' \subseteq V$ mit $|V'| = k, \{u, v\} \notin E \forall u, v \in V$

11.1.2 Komplexitätsklasse P

Alle (Entscheidungs-) Probleme, die in polynomieller Zeit entscheidbar sind.

z.B.:

geg: Graph $G = (V, E), s, t \in V$

Frage: existiert ein Weg zwischen s und t?

Antwort: in polynomieller Zeit mit Tiefensuche bspw.

es gilt:

$$P \subseteq NP$$

denn Zeuge immer z.B. leeres Wort. A(w,x) ignoriert x, entscheidet, ob $w \in L$

11.1.3 Alternative Definition von NP

Alle Sprachen, die von einer nicht-deterministischen (daher auch das 'N' in 'NP') Turingmaschine in polynomieller ('P') Zeit akzeptiert werden.

11.1.4 Polynomzeit-Reduktion

Def: Seien L_1 und L_2 zwei Sprachen mit $L_1, L_2 \subset \Sigma^*$. Dann heißt L_1 genau dann polynomzeit-reduzierbar auf L_2 (geschrieben $L_1 \leq_P L_2$), wenn eine in polynomieller Zeit berechenbare Funktion f existiert mit $w \in L_1 \Leftrightarrow f(w) \in L_2$.

Idee: Falls ein effizienter Algorithmus A_2 für L_2 existiert, so existiert auch ein Algorithmus A_1 für L_1.

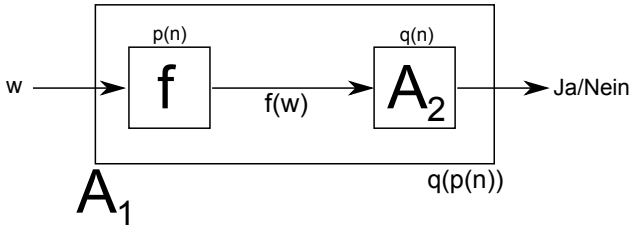


Abbildung 73

es gilt:

$$L_1 \leq_P L_2 \text{ und } L_2 \in P \Rightarrow L_1 \in P$$

Bsp: UK $\leq_P$ CLIQUE

denn: Sei $w = (G = (V, E), k)$ Eingabe für UK. Wir kosntruieren daraus:

$$f(w) = (\overline{G} = (V, \overline{E}), k)$$

wobei $\overline{E} = \{\{u, v\} | \{u, v\} \notin E\}$. f ist in linearer Zeit berechenbar und es gilt offensichtlich:

$$w \in UK \Leftrightarrow f(w) \in CLIQUE$$

11.1.5 Weitere Definitionen

a) L heißt genau dann NP-schwer (NP-hard), wenn $L' \leq_P L$ für alle $L' \in NP$.

b) L heißt genau dann NP-vollständig, wenn L NP-schwer ist und $L \in NP$.

Folgerungen:

a) für alle NP-vollständigen L gilt:

$$L \in P \Rightarrow P = NP$$

b) falls L_1 NP-schwer und $L_1 \leq_P L_2$, dann ist L_2 NP-schwer

Frage: Gibt es überhaupt NP-vollständige Probleme?

Antwort: Ja → Satz von Cook (1970): SAT ist NP-vollständig (Beweis direkt[23])

Bsp: Cook zeigte sogar, dass CNF-SAT NP-vollständig ist. Gegeben ist also eine Formel ϕ in konjunktiver Normalform. Ist ϕ erfüllbar?

zur Erinnerung: knf, bspw.:

$$\overbrace{(\underbrace{x_1}_{Literal} \vee \overline{x}_2 \vee x_3)}^{Klausel} \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (x_1 \vee x_2 \vee x_4) \tag{3}$$

wir zeigen:

$$\text{CNF-SAT} \ \leq_P \ \text{UK}$$

Reduktion: Sei ϕ boole'sche Formel in CNF. Wir konstruieren einenen Graph G_ϕ mit Zahl k, sodass ϕ erfüllbar $\Leftrightarrow$ G_ϕ hat UK der Größe k. G_ϕ hat 1 Knoten für jedes Literal in Klauseln von ϕ.

Bsp. Glg. 3: Verbinde durch Kanten: alle Literale, die innerhalb einer Klausel liegen

Beweis:

$\Leftarrow$: Sei V' UK von G_ϕ der Größe k

$\Rightarrow$ V' entsprihte Literalen, wobei genau eines aus jeder Klausel und keines ist Negation eines anderen

$\Rightarrow$ $\exists$ Belegung, die alle diese Literale auf 1 (= wahr) setzt

$\Rightarrow$ sie macht ϕ wahr

$\Rightarrow$: Sei G_ϕ erfüllbar. Dann existiert eine erfüllende Belegung und pro Klausel mindestens ein Literal, das auf 1 gesetzt wird. Die entsprechenden Knoten bilden UK der Größe k.

Damit:

$$\text{CNF-SAT} \leq_P \text{UK} \leq_P \text{CLIQUE}$$

$$\text{NP-schwer} \overset{b)}{\Rightarrow} \text{NP-schwer} \overset{b)}{\Rightarrow} \text{NP-schwer}$$

[23] fast alle NP-Vollständigkeitsbeweise durch Reduktion von einem anderen NP-vollständigen Problem