Organisatorisches

Programmieren und Software-Engineering
Theorie: Algorithmen, Homomorphismen und Formale Sprachen

12. März 2024

Organisatorisches

Die Gesamtnote POS ergibt sich durch

- POS Programmieren (nicht dieser Kurs)
- POS Theorie: (dieser Kurs!)
  1. Wintersemester: "Graphentheorie"
  2. Sommersemester:
     - **"Algorithmen"**
     - **"Homomorphismen"**
     - **"Formale Sprachen"**
     - **"Syntaxanalyse"**
- Alle Teile müssen positiv bestanden werden um positive Gesamtnote zu erhalten.

Beurteilung

- 65% SLÜs (Theorie)
- 35% Programmieraufgabe (Praxis)
- **Für eine positive Note müssen sowohl die Theorie als auch die Praxis (Programmieraufgabe) positiv sein.**
- Die SLÜs müssen gemeinsam eine positive Note ergeben, es muss jedoch nicht unbedingt jede einzelne SLÜ positiv sein.
- Bei nachweislicher Verhinderung bei der SLÜ kann auf Wunsch in der darauffolgenden Stunde eine Ersatzprüfung erfolgen.

Notenschlüssel

Prozent	Note
$[0, 50]$	Nicht Genügend
$(50, 62.5]$	Genügend
$(62.5, 75]$	Befriedigend
$(75, 87.5]$	Gut
$(87.5, 100]$	Sehr Gut

Programmieraufgabe

- Erstellen Sie ein Programm in Java, C#, Python, Javascript (andere Programmiersprachen nach Rücksprache möglich)
- Das Programm soll die Adjazenzmatrix eines Graphen aus einer Datei (csv) einlesen
- Weiters soll das Programm folgende Berechnungen durchführen (Minimalanforderungen):
  - Bestimmung der Distanzen und Exzentrizitäten aller Knoten
  - Radius, Durchmesser, Zentrum
  - Komponenten, Artikulationen, Brücken
- Wählen Sie eine dieser (oder andere) Erweiterungen für ein "Sehr Gut":
  - Grafische Benutzeroberfläche
  - Eulersche Linien/Zyklen
  - Spannbäume/Gerüste
  - Starke Zusammenhangskomponente
  - Blöcke (schwierig!)
  - Test auf Isomorphie

Programmabgabe

- Fertigstellung des Programmes bis 1. Juni für Wertung im laufenden Semester
- Für die Inskribtion des folgenden Sommersemesters (z.B. 6AKIF) Terminvereinbarung bis 10. Jänner!
- Persönliches Abgabegespräch (Raum B4.18a), Dauer ca. 20 Minuten
- Terminvereinbarung per Mail, bzw. wird es Anfang Juni wird es eine Liste mit Zeit-Slots geben
- Quellcode als ZIP-Datei (keine Binärdateien)
- Namenskonvention (Beispiel): `2021-06-21__3BAIF_Gruber.zip`
- Nur sinnvoll viele Kommentare im Code (keine vollständigen Erklärungen zu jeder Code-Zeile)!

Abgabegespräch

- **Raum B4.18a**
- Bereiten Sie mindestens einen Testfall mit mindestens 10 Knoten und mindestens zwei Brücken vor. Eine Zeichnung des/der Graphen soll ebenso präsentiert werden.
- Führen Sie zunächst das Programm aus: Demonstration der Funktionalität und Korrektheit.
- Geben Sie einen kurzen Überblick über den Aufbau des Programmes.
- Erklärung und Diskussion ausgewählter Algorithmen.
- Wie große Graphen kann ihr Programm berechnen? (Minimalanforderung: 15 Knoten)
- Feedback.

Kolloquium, spätere Programmabgabe

- Für reine Kolloquien (Theorie) nutzen Sie bitte einen der Sammeltermine (üblicherweise nach Semesterstart, in der Mitte und am Ende des Semesters)
- Terminvereinbarung per Mail (Betreff: Kolloquium 4XYIF POS)
- Bitte Klasse und Schuljahr im Mail anführen!
- Bei Programmabgaben oder für individuelle Termine zu Kolloquien: bitte zwei bis drei Termine (kompatibel mit unseren Stundenplänen) vorschlagen
- Zeit und Ort von Kolloquien (Sammeltermine) sind kurzfristig in meinem(!) Stundenplan ersichtlich. Im Zweifelsfall (falls nicht eingetragen) Treffpunkt in B4.18 ca. 10 Minuten vor Beginn.
- Bitte ausschließlich die Schul-Emailadresse verwenden, und bei Antworten immer die vorangegange Kommunikation zitieren.
- Emailanfragen ohne entsprechende Informationen, bzw. zu hier beantworteten Fragen werden nicht beantwortet.

Stoffübersicht

- Algorithmen
  - Analyse
  - Rekursion
  - Datenstrukturen
  - ...
- Homomorphismen
- Syntaxanalyse
  - Grammatiken
  - Backus-Naur Form
  - Syntaxdiagramme, Syntaxanalyse
  - ...
- Formale Sprachen
  - Chomsky Hierarchie
  - Reguläre Sprachen, endliche Automaten
  - Kontextfreie Sprachen
  - Komplexität und Berechenbarkeit

Unterlagen

- Die Vortragsfolien sowie ein zur Verfügung gestelltes Programm decken die Inhalte ab!
- Eigene Mitschrift zu Beispielen, Erklärungen
- Einige Übungsbeispiele werden nur im Unterricht an der Tafel präsentiert.

Achtung!

Die Präsentationsfolien enthalten Animationen die zu gewissen Beispielen/Algorithmen eine Schritt-für-Schritt Erklärung enthalten. Diese Animationen sind am Besten im Vollbildmodus anzusehen! Im gedruckten Handout finden sich lediglich verkürzte Darstellungen mit weniger Zwischenschritten.

Weiterführende Inhalte (*)

- Manche Inhalte sind speziell für besonders Interessierte gedacht
- Diese Inhalte zählen nicht zum Kernstoffgebiet, und müssen somit nicht gelernt/gekonnt/verstanden werden
- Die Kennzeichnung dieser Inhalte erfolgt durch die blaue Titel- und Fußzeile in den Folien!

Literaturübersicht I

[1] **Berger, Krieger, Mahr: “Grundlagen der elektronischen Datenverarbeitung”, Skriptum**

[2] Dirk W. Hoffmann: “Theoretische Informatik”, Hanser, 3. Auflage

[3] Gernot Salzer: “Einführung in die Theorie der Informatik”, Skriptum, TU Wien, 2001

[4] Wikipedia (Englisch): `https://en.wikipedia.org/`

[5] Wikipedia (Deutsch): `https://de.wikipedia.org/`

[6] HappyCoders (Deutsch): `https://www.happycoders.eu/de/algorithmen/`

Breiten- und Tiefensuche

Andreas M. Chwatal

**Programmieren und Software-Engineering
Theorie**

12. März 2024

Traversierung von Bäumen

Wir betrachten im Folgenden Algorithmen zur Traversierung von Bäumen. Diese können jedoch auch zur Traversierung von Graphen im Allgemeinen verwendet werden.

Ziel: Wir wollen die Knoten eines Baumes systematisch durchlaufen, mit dem Ziel einen bestimmten Knoten zu finden.

- **Breitensuche (Breadth-First-Search (BFS)):** In jedem Schritt werden zunächst alle Nachbarknoten eines Knoten besucht, bevor von dort aus weitere Pfade gebildet werden.
- **Tiefensuche (Depth-First-Search (DFS)):** Ein Pfad wird vollständig in die Tiefe durchlaufen, bevor etwaige Abzweigungen verwendet werden.

Traversierung von Bäumen

Zur Beschreibung der Suchverfahren werden folgende Begriffe benötigt:

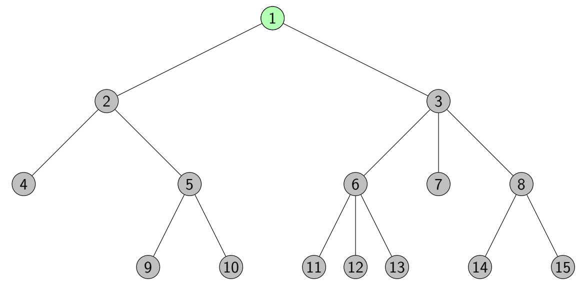
- Ein Knoten wird **entdeckt**, wenn er das erste Mal besucht wird.
- Ein Knoten wird **fertiggestellt/abgeschlossen**, wenn er das letzte Mal verlassen wird.

Für manche Anwendungen ist es wichtig festzuhalten, wann ein Knoten *entdeckt*, bzw. abgeschlossen wurde. Dazu führen wir einen Zähler τ mit, der in jedem Schritt um 1 erhöht wird.

- Wird ein Knoten v das erste mal besucht ("entdeckt"), so setzen wir $\tau_d(v) = \tau$++
- Wird ein Knoten v das letzte mal verlassen ("abgeschlossen"), so setzen wir $\tau_f(v) = \tau$++

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

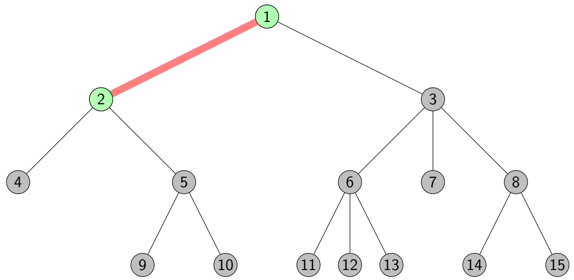


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

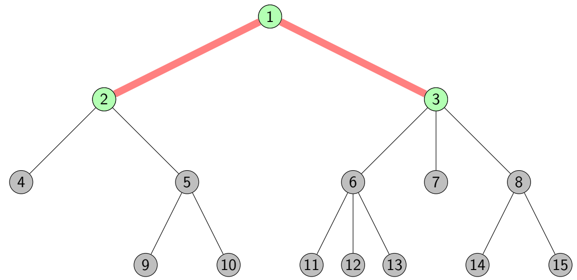


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

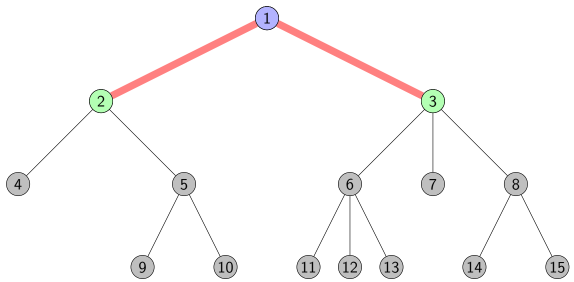


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

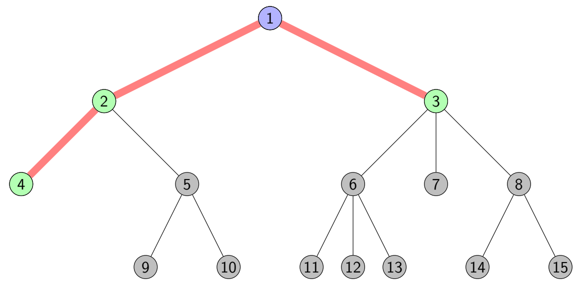


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

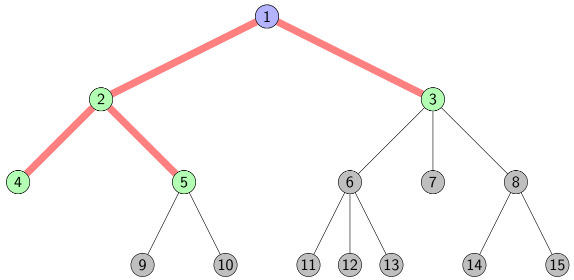


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.



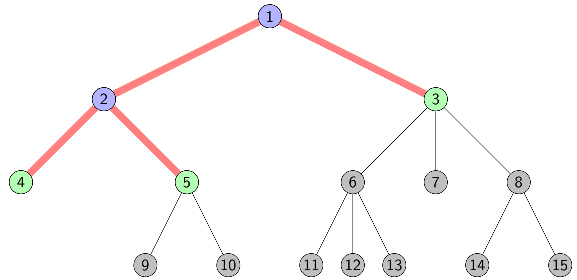
$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Traversierung

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.



$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

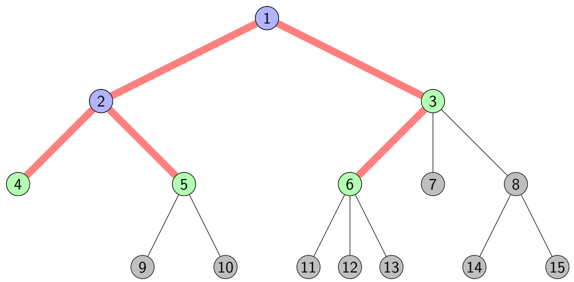
$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

POS (Theorie) Theorie 4 / 10

Traversierung

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.



$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

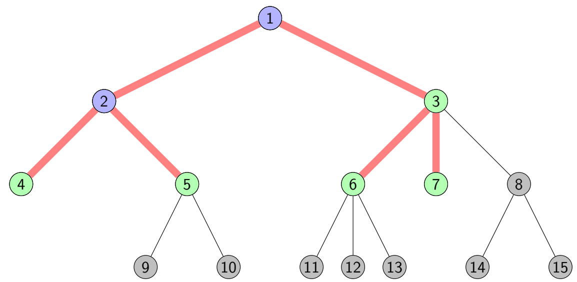
$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

POS (Theorie) Theorie 4 / 10

Traversierung

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.



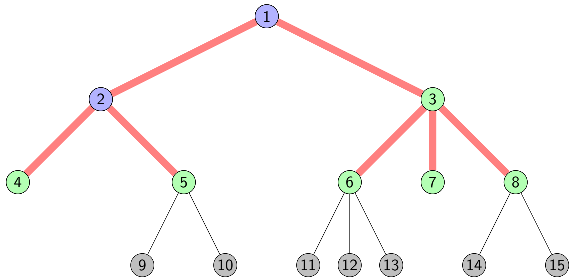
$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

POS (Theorie) Theorie 4 / 10

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

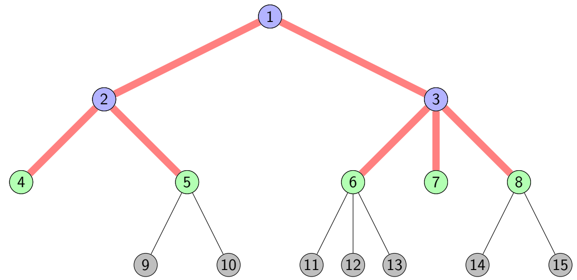


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

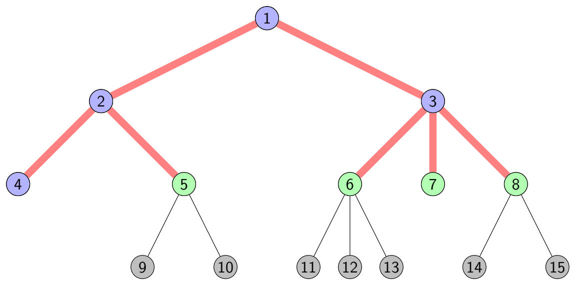


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

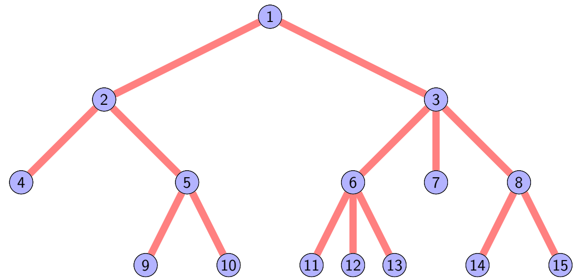


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.



Achtung: einige Zwischenschritte sind in diesem Handout übersprungen! Alle Schritte sind im einzelnen Foliensatz als Animation verfügbar!

$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche: Datenstruktur

In nahezu allen Programmiersprachen existiert eine Datenstruktur namens **Queue** (Warteschlange). Elemente können hinzugefügt werden ("hinten anstellen"), und werden geordnet abgespeichert. Das Element das als erstes hinzugefügt wurde, kann entnommen werden ("Nächster!")

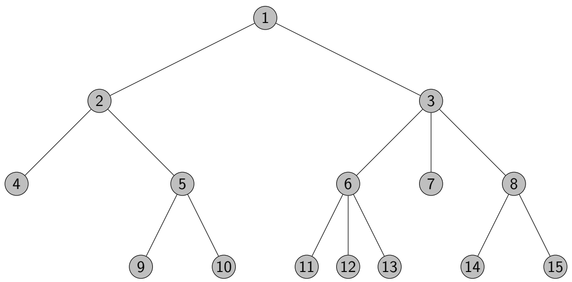
Breitensuche: Algorithmus

1. Füge Startknoten (Wurzel) in Queue (Warteschlange) ein
2. Entnimm Knoten am Beginn der Queue
   - Wenn Knoten gefunden: Abbruch
   - Sonst: füge alle unbesuchten[1] Nachbarn in die Queue ein
3. Wenn die Warteschlange leer ist wurden alle Knoten bereits besucht → Abbruch
4. Gehe zu Schritt (2)

[1]nicht entdeckt, nicht abgeschlossen

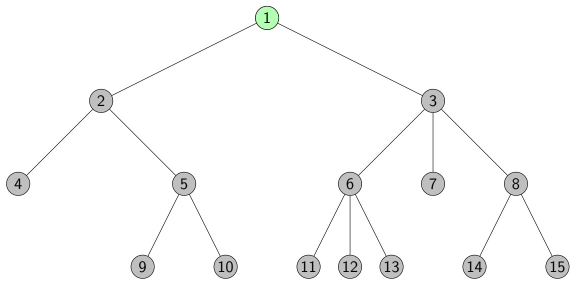
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



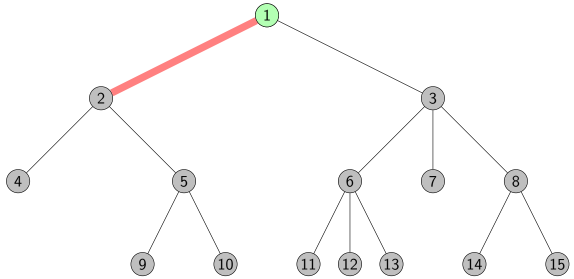
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



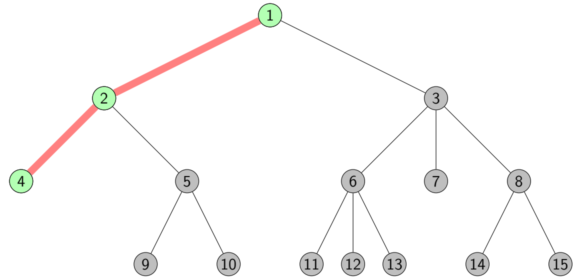
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



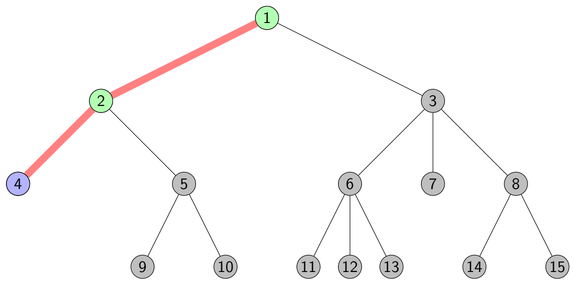
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



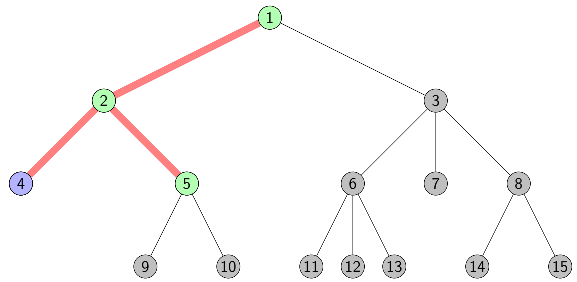
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



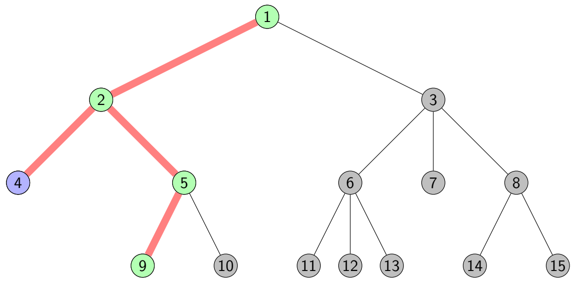
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



Tiefensuche

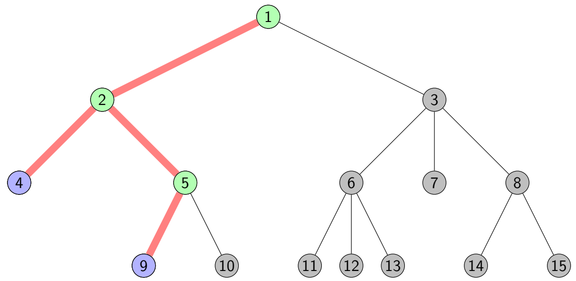
Beispiel: Beispiel einer Tiefensuche:



Traversierung

Tiefensuche

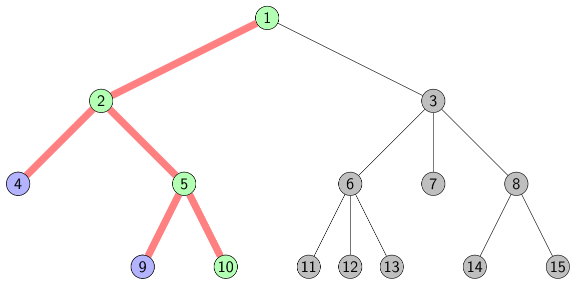
Beispiel: Beispiel einer Tiefensuche:



Traversierung

Tiefensuche

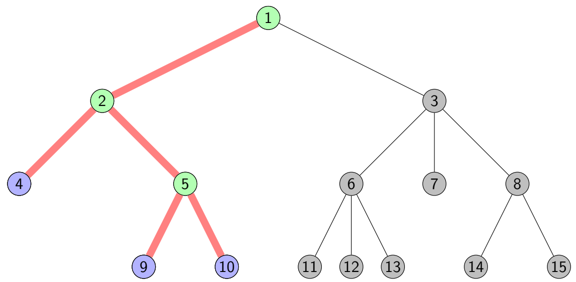
Beispiel: Beispiel einer Tiefensuche:



Traversierung

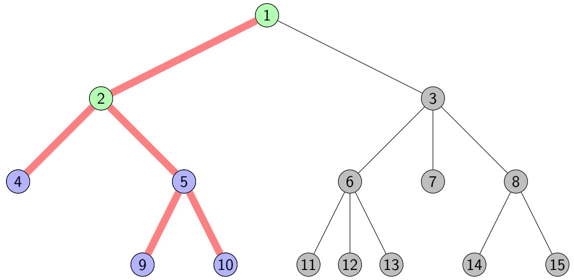
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



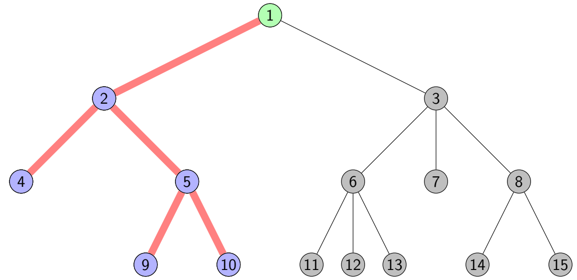
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



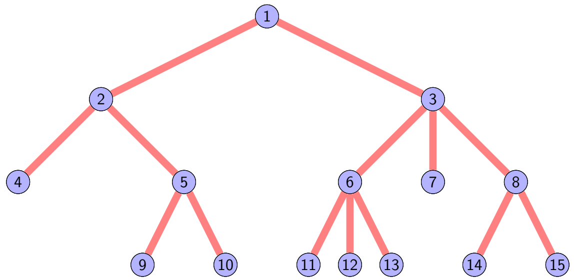
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



Achtung: einige Zwischenschritte sind in diesem Handout übersprungen! Alle Schritte sind im einzelnen Foliensatz als Animation verfügbar!

Tiefensuche: Datenstruktur

In nahezu allen Programmiersprachen existiert eine Datenstruktur namens **Stack** (Stapel).
Elemente können hinzugefügt werden ("oben drauflegen"), und werden geordnet abgespeichert. Das Element das als *letztes* hinzugefügt wurde, kann entnommen werden (oberstes Element vom Stapel nehmen).

Tiefensuche: Algorithmus

Algorithm 1: Tiefensuche

```
Function DFS(G = (V, E), Startknoten v, Gesuchter Knoten s)
    Result: vertex s ∈ V, if exists
    Stack S;
    S.push(v); // Lege v auf Stapel
    while S not empty do
        v = S.pop(); // nimm obersten Knoten vom Stapel
        if v gesuchter Knoten s then
            return v;
        if v noch nicht besucht then
            for all [v, u] ∈ E(G) do
                if u noch nicht besucht then
                    S.push(u);
```


Anmerkungen

- Die Algorithmen können auch für allgemeine Graphen (und nicht nur Bäume) verwendet werden.
- Dabei werden schon besuchte Knoten *nicht* erneut besucht!
- Anwendungen BFS:
  - 2-färbbarkeit
  - Kürzester Pfad zwischen zwei Knoten
  - Kürzeste-Kreise-Problem
- Anwendungen DFS:
  - Test auf Kreisfreiheit
  - Topologische Sortierung
  - Starke Zusammenhangskomponente

Algorithmus von Dijkstra

Programmieren und Software-Engineering
Theorie

12. März 2024

Algorithmus von Dijkstra

Der Algorithmus von Dijkstra berechnet die kürzesten Wege in einem gewichteten Graphen mit $w_{ij} \geq 0$, für alle $[i,j] \in E$.

Grundidee:

- Ähnlichkeit zu BFS, jedoch andere Regeln für die Auswahl des nächsten Knoten.
- Ein Aufruf von Dijkstra berechnet die kürzesten Wege von einem Startknoten zu allen anderen Knoten des Graphen.
- In jedem Schritt werden **Zwischenergebnisse** $\delta_k, k \in V$ berechnet, bzw. aktualisiert.
- Diese Zwischenergebnisse sind die Länge des kürzesten *bisher gefundenen* Weges bis zu diesem Knoten.
- Wiederhole (bis alle Knoten abgeschlossen):
  1. Wähle Knoten k mit minimalem δ_k und schließe diesen ab.
  2. Speichere Verweis auf direkten Vorgänger.
  3. Aktualisiere die Werte δ_k für noch nicht abgeschlossene Nachbarknoten von k.

Algorithmus von Dijkstra

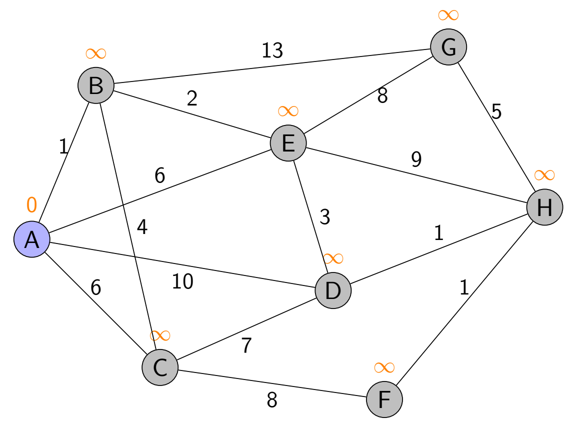
Algorithm 1: DIJKSTRA

Data: Graph G mit $w_{ij} \geq 0$ für alle $(i,j) \in E(G)$
Data: Startknoten s

```
∀v ∈ V : δ_v ← ∞;
δ_s ← 0;
Prioritätswarteschlange Q befüllt mit allen Knoten ;
while Q ≠ ∅ do
    u ← Q.getMin(); // entnimmt Knoten u mit kleinstem δ_u
    Fertigstellung von Knoten u;
    Speichere Verweis auf direkten Vorgänger von u;
    for all (u, v) ∈ E, v noch nicht fertiggestellt do
        if δ_v > δ_u + w_uv then
            δ_v ← δ_u + w_uv;
```

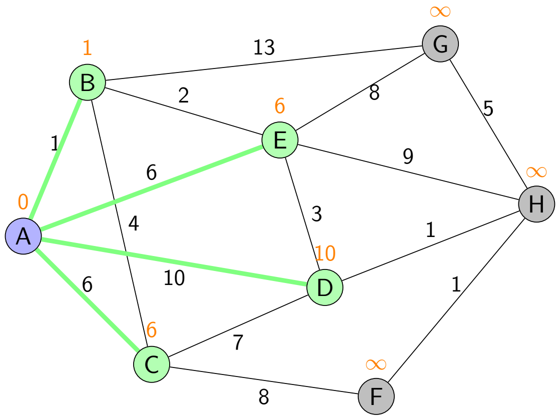
Anmerkung: Die Prioritätswarteschlange Q kann durch ein einfaches Array umgesetzt werden. Um u mit kleinstem δ_u zu finden, muss es zur Gänze durchlaufen werden. Dies ist nachteilig für die Performance des Algorithmus, weshalb die Prioritätswarteschlange meist anhand eines *Heaps* umgesetzt wird.

Algorithmus von Dijkstra: Beispiel



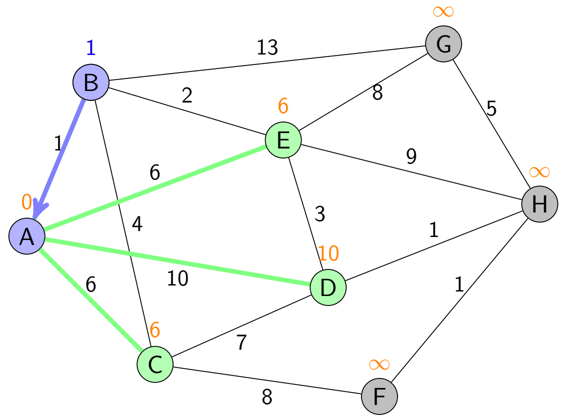
Wir suchen den kürstesten Weg vom Knoten A zum Knoten H. Im ersten Schritt wird der Startknoten "fertiggestellt".

Algorithmus von Dijkstra: Beispiel



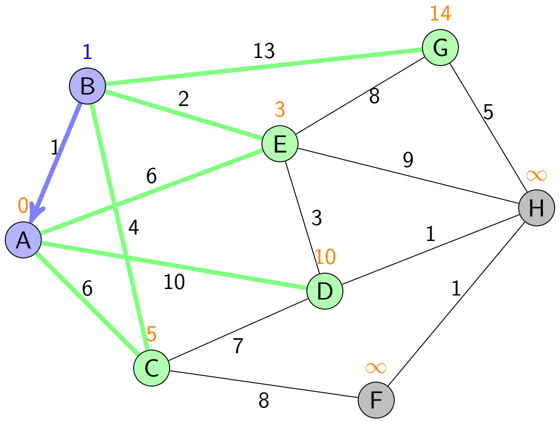
Im nächsten Schritt werden die Nachbarknoten B, C, D und E entdeckt. Die Zwischenwerte werden wie folgt berechnet:
$\delta_A = 0, \delta_B = \delta_A + 1 = 1, \delta E = \delta_A + 6 = 6, \delta_D = \delta_A + 10 = 10, \delta_C = \delta_A + 6 = 6.$

Algorithmus von Dijkstra: Beispiel



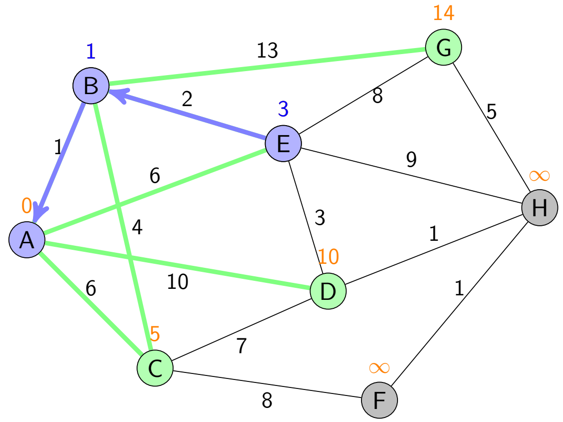
Nun wird der Knoten v mit kleinstem δ_v fertiggestellt. Im konkreten Beispiel ist dies Knoten B.

Algorithmus von Dijkstra: Beispiel



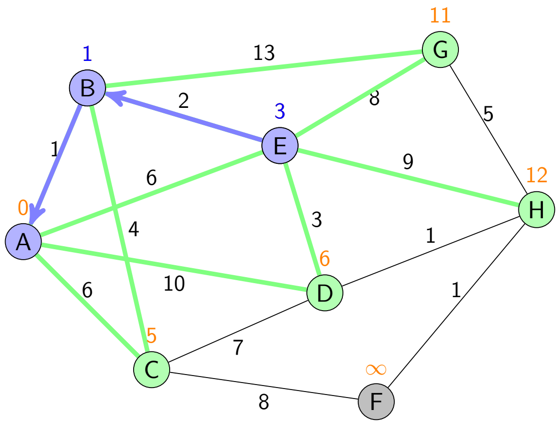
Ausgehend vom letzten fertiggestellten Knoten (B) werden nun neue Zwischenergebnisse für C, E und G berechnet. Wir erhalten $\delta_G = 1 + 13 = 14, \delta_E = 1 + 2 = 3, \delta_C = 1 + 4 = 5$. Für die Knoten C und E erhalten wir kleinere Werte als die bisher gefundenen.

Algorithmus von Dijkstra: Beispiel



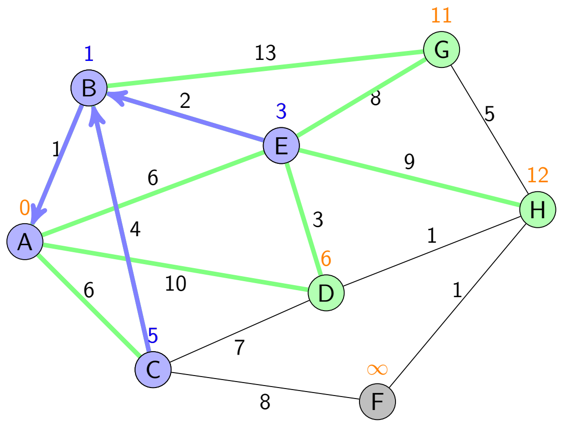
Knoten E wird fertiggestellt. Bei fertiggestellten Knoten merkt man sich wo man hergekommen ist (daher die blaue Kante).

Algorithmus von Dijkstra: Beispiel



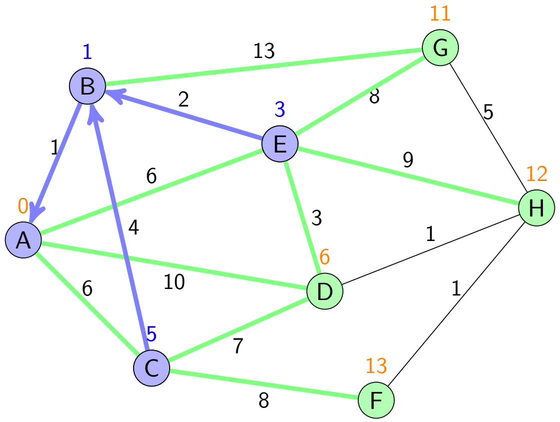
Berechnung neuer Zwischenergebnisse für D, G und H.

Algorithmus von Dijkstra: Beispiel



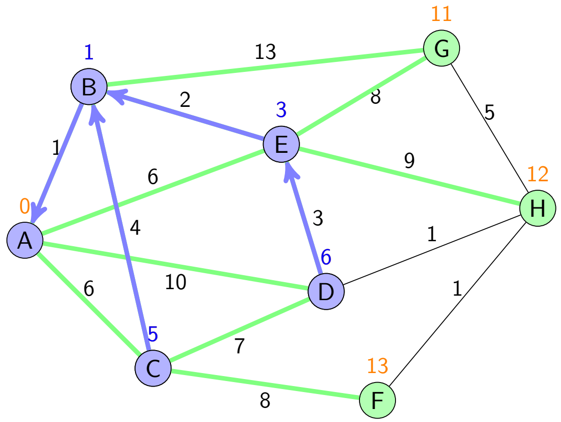
Knoten C wird fertiggestellt, da er nun den kleinsten Zwischenwert δ hat.

Algorithmus von Dijkstra: Beispiel



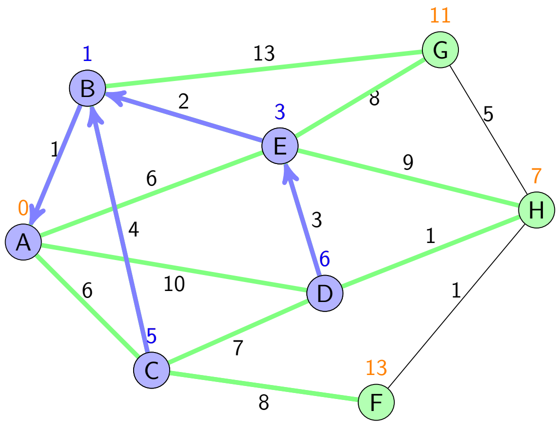
Ausgehend von C werden die Werte δ_D und δ_F berechnet. Da $5 + 7 > 6$ kommt es bei δ_D zu keiner Änderung.

Algorithmus von Dijkstra: Beispiel



Fortgefahren wird mit Knoten D, da kleinstes δ.

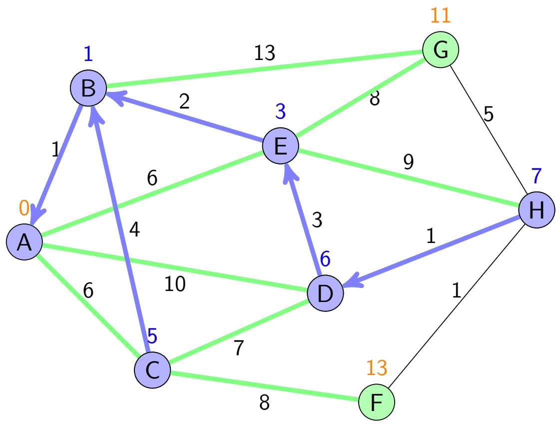
Algorithmus von Dijkstra: Beispiel



δ_H wird aktualisiert.

Algorithmus von Dijkstra

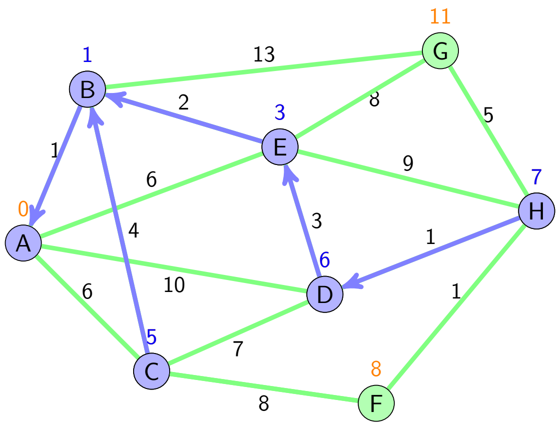
Algorithmus von Dijkstra: Beispiel



H wird fertiggestellt.

Algorithmus von Dijkstra

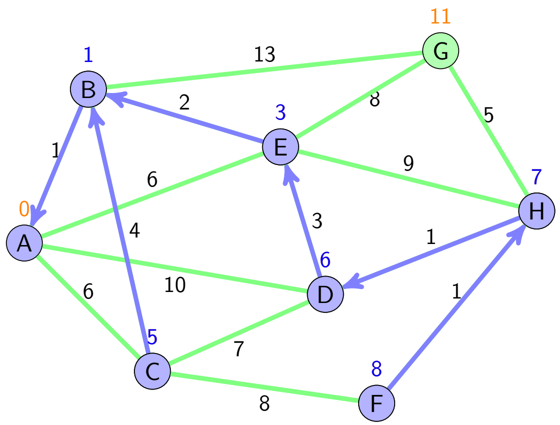
Algorithmus von Dijkstra: Beispiel



Update von δ_F und δ_G, wobei nur δ_F tatsächlich geändert wird.

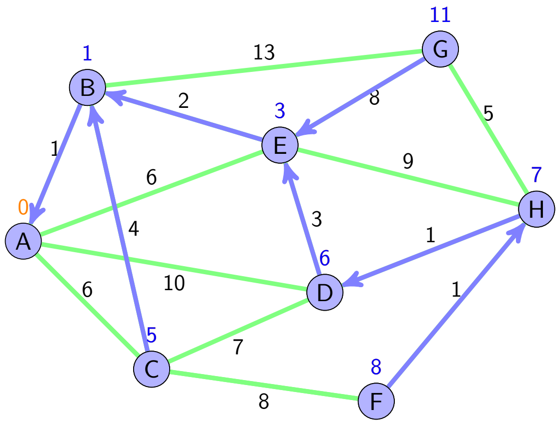
Algorithmus von Dijkstra

Algorithmus von Dijkstra: Beispiel



F wird fertiggestellt.

Algorithmus von Dijkstra: Beispiel



G wird fertiggestellt (Vorgänger *E*).

Algorithmus von Dijkstra

- Die `blauen Kanten` bilden einen Wurzelbaum[1], der die kürzesten Wege vom Startknoten zu jedem Knoten enthält.
- Die Berechnung des kürzesten Weges von einem Startknoten zu einem Zielknoten beinhaltet also die Berechnung der kürzesten Wege vom Startknoten zu *allen* anderen Knoten.
- Ist nur der kürzeste Weg zu einem Zielknoten von Interesse, kann die Berechnung abgebrochen werden sobald dieser fertiggestellt wird.

[1] streng genommen bilden die umgedrehten blauen Kanten einen Wurzelbaum

Algorithmus von Dijkstra – Laufzeitanalyse

Mit $n = |V|$ und $m = |E|$ können wir die Laufzeiteigenschaften angeben. Diese hängen von der konkreten Umsetzung der Prioritätswarteschlange Q ab.

Operation		Queue Implementierung		
Name	Anzahl	Liste	Heap	Fibonacci Heap[2]
decreaseKey [10]	m	$O(1)$	$O(\log n)$	$O(1)$
getMin [5]	n	$O(n)$	$O(\log n)$	$O(\log n)$
create [3]	1	$O(n)$	$O(n)$	$O(n)$
Gesamt		$O(n^2 + m) = O(n^2)$	$O((n + m) \log n)$	$O(n \log n + m)$

Anmerkung: In der Spalte ganz links ist in eckigen Klammern auf die Zeile der jeweiligen Operation im Pseudocode verwiesen!

[2] amortisierte Laufzeit

O-Notation

Programmieren und Software-Engineering
Homomorphismen, Formale Sprachen und Syntax-Analyse

12. März 2024

Motivation

- Bestimmte Programmteile verursachen je nach Datenmenge unterschiedliche *Laufzeiten* bei der Ausführung.
- Bei geringen Datenmengen bleibt dies oft unbemerkt und spielt keine Rolle.
- Reale Programme arbeiten jedoch mit großen Datenmengen, umfangreichen Datenbanken, führen komplexe Berechnungen durch, etc.
- Durch die theoretische Laufzeitanalyse kann man etwaige Flaschenhälse und Performanceprobleme erkennen.
- Relevant ist hierbei weniger die konkrete Ausführungsgeschwindigkeit, sondern das Wachstum der Laufzeit in Abhängigkeit von der Anzahl der Eingabedaten.
- In der praktischen Arbeit werden *Profiler* verwendet, um zu analysieren wieviel Zeit in welchen Programmteilen verbracht wird.
- Für theoretische Analysen wird die O-Notation verwendet.

Komplexität von Algorithmen

Die theoretische Analyse von Algorithmen untersucht:

- (Lauf-)Zeit-Komplexität
- (Speicher-)Platz-Komplexität
- Evtl. weitere Parameter, z.B.:
  - Anzahl Vergleiche
  - Anzahl Bewegungen der Datensätze

Weitere Unterscheidung:

- **Worst-Case**: der ungünstigste Fall
- Average-Case: der zu erwartende, durchschnittliche Fall
- Best-Case

Komplexität von Algorithmen

- Untersucht werden die Eingenschaften von Algorithmen in Abhängigkeit von den Eingabedaten.
- Wesentliche Kenngröße ist die Anzahl der Daten, oft mit n oder m bezeichnet.
  - *Beispiel:* Sortieren von n Zahlen
  - *Beispiel:* Berechnen eines Euler-Zyklus von einem Graphen mit n Knoten und m Kanten
  - *Beispiel:* Tiefensuche in Graphen mit n Knoten und m Kanten
  - *Beispiel:* Suchen eines Elementes in n vorhandenen Einträgen
- Zentrale Frage: wie hängt die Laufzeit von n (oder m) ab?
- Von Interesse ist lediglich die **Größenordnung**!
- Diese Größenordnung wird mit der Bachmann-Landau-Notation, oder kurz O-**Notation** dargestellt.

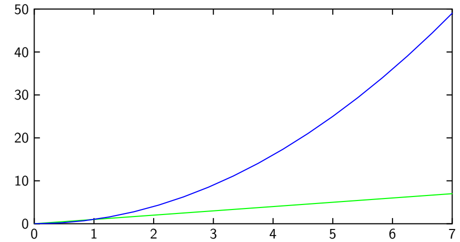
Komplexität von Algorithmen

- Ein sehr günstiges Laufzeitverhalten ist, wenn bei n Datensätzen **ca.** n Schritte ausgeführt werden.
  - Man nennt dies: **lineares** Laufzeitverhalten
  - Der Algorithmus durchläuft $O(n)$ Schritte
- Werden ungefähr n^2 viele Schritte ausgeführt:
  - **Quadratisches** Laufzeitverhalten
  - Der Algorithmus durchläuft $O(n^2)$ Schritte
- Höhere Laufzeiten sind oftmals in der Praxis problematisch, selbst bei quadratischer Laufzeit ist oft schon Vorsicht angebracht

Definition ((informell) Laufzeit eines Algorithmus anhand der O-Notation)

Ein Algorithmus A hat bei n Eingabedaten eine Laufzeit von $O(f(n))$, wenn bei seiner Ausführung *in etwa* ("in der Größenordnung von") $f(n)$ Schritte ausgeführt werden.

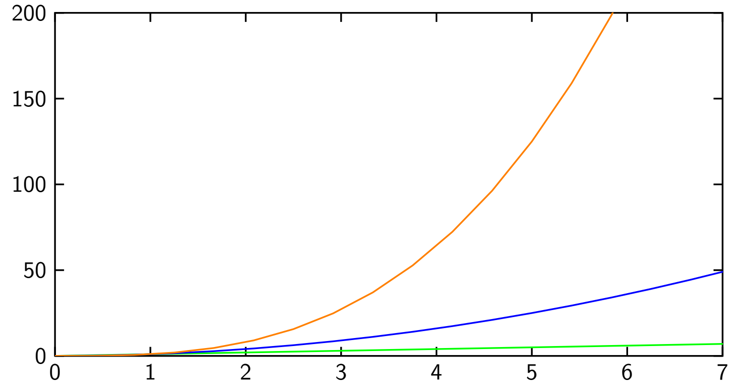
Wachstum von $f(n)$



$f(n) = n$, $f(n) = n^2$

Vergleich von linearem zu quadratischem Laufzeitverhalten

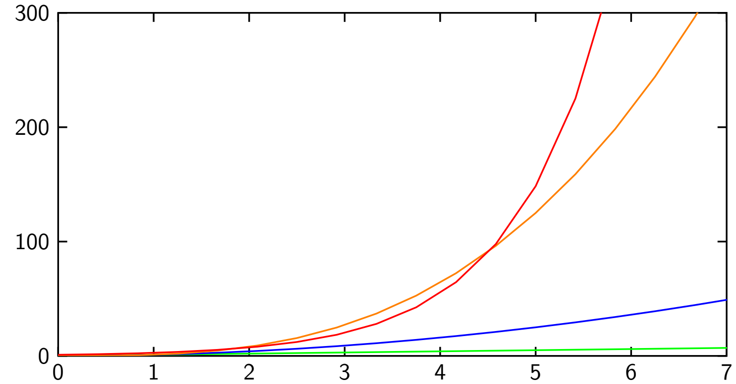
Wachstum von $f(n)$



$f(n) = n$, $f(n) = n^2$, $f(n) = n^3$

Vergleich von n, n^2 und n^3

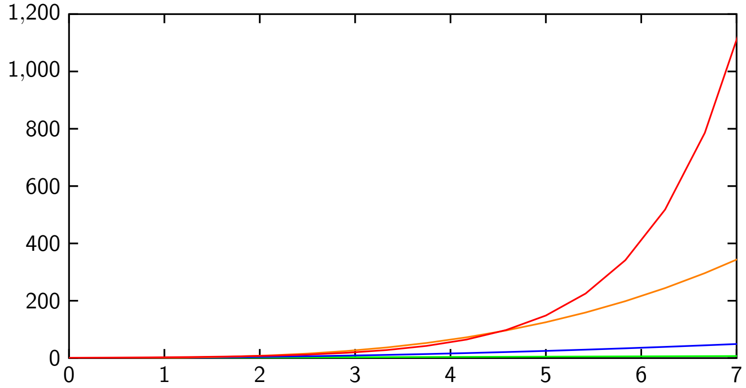
Wachstum von $f(n)$



$f(n) = n$, $f(n) = n^2$, $f(n) = n^3$, $f(n) = e^n$

Zusätzliche exponentielle Kurve, zunächst mit $y_{max} = 300$

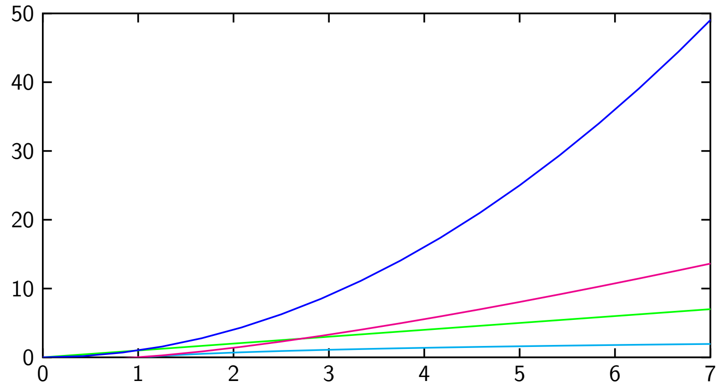
Wachstum von $f(n)$



$f(n) = n$, $f(n) = n^2$, $f(n) = n^3$, $f(n) = e^n$

Zusätzliche exponentielle Kurve, jetzt mit $y_{max} = 1200$

Wachstum von $f(n)$



$f(n) = n$, $f(n) = n^2$, $f(n) = \ln n$, $f(n) = n \ln n$

Weiteres Beispiel: die Logarithmus-Funktion. Diese strebt zwar gegen Unendlich, aber extrem langsam! Wichtige Konsequenz: $n \ln n$ verhält sich “fast” wie n

Bachmann-Landau Notation

Eine Funktion $f(n)$ liegt in $O(g(n))$ wenn ab einem Wert n_0 und einer Konstante c die Funktionswerte von $f(n)$ kleiner sind als jene von $c \cdot g(n)$. Man schreibt hierfür auch $f(n) \in O(g(n))$.

Definition (Bachmann-Landau *O*-Notation)

$$O(g(n)) = \{f(n) \mid (\exists c, n_0 > 0), (\forall n \geq n_0) : 0 \leq f(n) \leq cg(n)\}$$

Somit ist $c \cdot g(n)$ ab n_0 eine **obere Schranke** für $f(n)$! Es müssen immer nur die Terme der höchsten Ordnung betrachtet werden, und diese ohne etwaige konstante Faktoren.

Bachmann-Landau Notation

Beispiele:

- $3n^2 \in O(n^2)$
- $\frac{3}{4}n^3 + 5n^2 \in O(n^3)$
- $42n + \ln n + 5150 \in O(n)$
- $n \log n \in O(n^2)$
- $n \log n \in O(n \log n)$ (!) die kleinste obere Schranke!
- $n + m^2 \in O(n + m^2)$

Rekursion

Andreas M. Chwatal

Programmieren und Software-Engineering
Theorie

12. März 2024

Definition

Definition (Rekursion)

Unter einer rekursiven Methode versteht man eine Methode die sich selbst aufruft.

- Die Definition gilt auch für *Funktionen* oder *Programme* (statt Methoden).
- Viele Problemstellungen/Algorithmen lassen sich sehr einfach und elegant mittels Rekursion formulieren.
- Abbruchbedingung: muss vorhanden sein um die rekursiven Aufrufe zu beenden.

Fakultät

In der Mathematik bezeichnet $n!$ die Fakultät, und es gilt

$$n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$$

für alle $n > 0$ und $0! = 1$.

Rekursive Formulierung der Fakultät:

$$n! = n \cdot (n-1)! \qquad \text{für } n > 0$$

$$0! = 1$$

Direkte Umsetzung in Programmcode:

```
long factorial(long n) {
    if (n == 0) return 1;
    return n * factorial(n-1);
}
```

Achtung: der Aufruf `factorial(-1)` führt zu einer endlosen Rekursion! Dieser Fall sollte im Code abgefangen werden.

Negativbeispiel: Fibonacci-Zahlen

Die Folge der Fibonacci-Zahlen ist für $n \geq 2$ definiert als

$$f_n = f_{n-1} + f_{n-2},$$

weiters gilt $f_0 = f_1 = 1$.

Hierdurch erhalten wir die Folge:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \ldots$$

Negativbeispiel: Fibonacci-Zahlen

Die direkte Umsetzung als rekursives Programm ist jedoch sehr unvorteilhaft:

```java
long fibonacci(long n) {
    if (n <= 1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Frage

Wo genau liegt das Problem bei der rekursiven Implementierung der Fibonacci-Zahlen?

Negativbeispiel: Fibonacci-Zahlen

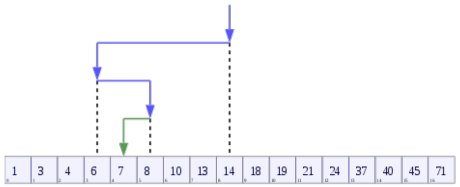
Die *iterative* Implementierung ist hier vorteilhaft, da wesentlich weniger Berechnungen ausgeführte werden:

```java
long fibonacci(long n) {
    if (n < 0 || n > MAX) {
        // Ausnahmebehandlung
    }
    long[] f = new long[n+1];
    f[0] = 1; f[1] = 1;
    for (long i=2; i<=n; i++) {
        f[i] = f[i-1] + f[i-2];
    }
    return f[n];
}
```

Anmerkung: Die Berechnung des n-ten Folgegliedes ist auch ohne Array möglich!

Suche in sortierten Listen

- Aufgabenstellung: Suchen eines Elementes in einer sortierten Liste (Array)
- Naiver Zugang: $O(n)$ Schritte (Erwartungswert $n/2$)
- Mittels binärer Suche: nur $O(\log n)$ Schritte notwendig
- Vorgehensweise: analog zu Suche in Telefonbuch
  - Beliebige Seite aufschlagen
  - Steht Name davor oder danach?
  - Weitere Suche erfolgt nur mehr im verbleibenden Teil



Binäre Suche

Algorithm 1: (Iterative) Binäre Suche

Function *BinarySearch(sorted array a[], element e)*
  Result: found element, or NULL if element not exists
  $l = 0$;
  $r = a.length$;
  while $l \leq r$ **do**
    $m = l + \frac{(r-l)}{2}$;
    if $(a[m] == e)$ **then**
      return e;
    else
      if $a[m] > e$ **then**
        $r = m - 1$;
      else
        $l = m + 1$;
  return *NULL*;

Rekursive Binäre Suche

Algorithm 2: Rekursive Binäre Suche

```
Function RecBinarySearch(sorted array a[], element e, l, r)
    Result: found element, or NULL if element not exists
    if l ≤ r then
        m = l + (r−l)/2;
        if (a[m] == e) then
            return e;
        else
            if a[m] > e then
                return RecBinarySearch(a, e, m+1, r);
            else
                return RecBinarySearch(a, e, l, m-1);
    else
        return NULL;
```

Aufruf mit array a und gesuchtem Element e und `RecBinarySearch(a, e, 0, a.length-1)`.

Rekursive Binäre Suche

Eine rekursive Variante in Python:

```python
def binaersuche_rekursiv(werte, gesucht, start, ende):
    if ende < start:
        return 'nicht gefunden'
        # alternativ: return -start # bei (-Returnwert) waere
        # die richtige Einfuege-Position

    mitte = (start + ende) // 2
    if werte[mitte] == gesucht:
        return mitte
    elif werte[mitte] < gesucht:
        return binaersuche_rekursiv(werte, gesucht, mitte+1, ende)
    else:
        return binaersuche_rekursiv(werte, gesucht, start, mitte-1)

def binaersuche(werte, gesucht):
    return binaersuche_rekursiv(werte, gesucht, 0, len(werte)-1)
```

Quelle: Wikipedia

Tiefensuche

Bei der Tiefensuche ist eine rekursive Implementierung naheliegend. Diese Variante durchläuft alle vom ersten Aufruf mit Knoten $v \in V(G)$ (`RecDFS(v)`) aus erreichbaren Knoten des Graphen G .

Algorithm 3: Rekursive Tiefensuche

```
Function RecDFS(Knoten v)
    markiere v als besucht;
    for all [v, u] ∈ E(G) do
        if u noch nicht besucht then
            RecDFS(u);
```

In der rekursiven Version der Tiefensuche ist kein *Stack* notwendig, die Knoten werden durch die rekursiven Aufrufe automatisch in der richtigen Reihenfolge besucht.

Tiefensuche

Diese Variante sucht einen Knoten s und returniert ihn sobald gefunden:

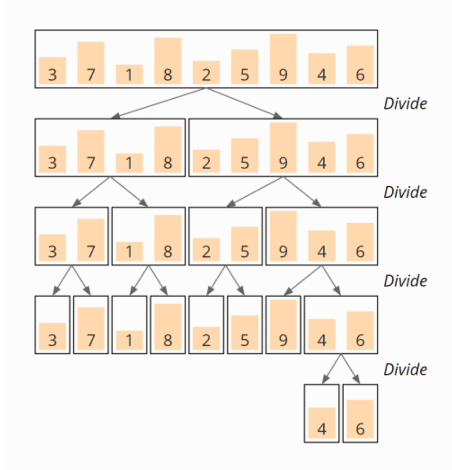
Algorithm 4: Rekursive Tiefensuche (2)

```
Function RecDFS(Knoten v, Gesuchter Knoten s)
    if v gesuchter Knoten s then
        return v;
    markiere v als besucht;
    for all [v, u] ∈ E(G) do
        if u noch nicht besucht then
            if RecDFS(u, s) == s then
                return s;
    return null;
```

Mergesort

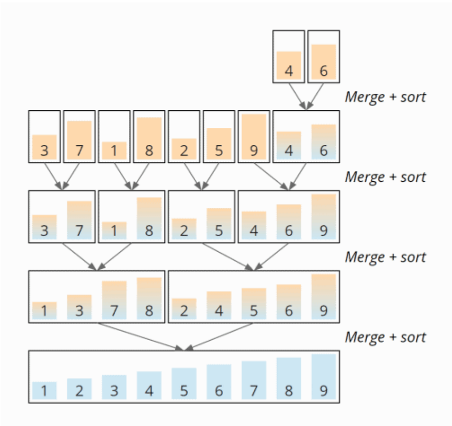
- Der Algorithmus *Mergesort* sortiert die Daten nach dem Prinzip **Divide & Conquer** (dt.: Teile und Herrsche)
- Vorgehensweise bei Divide & Conquer:
  - *Teile* das Problem in kleinere Teilprobleme
  - *Löse* diese Teilprobleme
  - Füge Teillösungen zusammen
- **Mergesort**
  - Zahlenfolge (Array) wird durch rekursive Aufrufe unterteilt.
  - Die Sortierung wird beim anschließenden Zusammenfügen der Arrays erreicht
  - Mergesort weist bessere Laufzeit-Eigenschaften als die bisher besprochenen Sortieralgorithmen auf!

Mergesort



Quelle: https://www.happycoders.eu/de/algorithmen/mergesort/

Mergesort



Quelle: https://www.happycoders.eu/de/algorithmen/mergesort/

Mergesort

Mergesort-Code in Java. Zunächst der erste Aufruf:

```java
int[] elements = { 3, 1, 6, 7, 4, 12 }; // zu sortierendes Array
int[] sorted = mergeSort(elements, 0, elements.length - 1);
```

Die rekursive Methode sieht wie folgt aus:

```java
int[] mergeSort(int[] elements, int left, int right) {

    if (left == right) return new int[]{ elements[left] };

    int middle = left + (right - left) / 2;
    int[] leftArray = mergeSort(elements, left, middle);
    int[] rightArray = mergeSort(elements, middle + 1, right);
    return merge(leftArray, rightArray);
}
```

```java
int[] merge(int[] leftArray, int[] rightArray) {
    int leftLen = leftArray.length;
    int rightLen = rightArray.length;

    int[] target = new int[leftLen + rightLen];
    int targetPos = 0;
    int leftPos = 0;
    int rightPos = 0;

    // As long as both arrays contain elements...
    while (leftPos < leftLen && rightPos < rightLen) {
      // Which one is smaller?
      int leftValue = leftArray[leftPos];
      int rightValue = rightArray[rightPos];
      if (leftValue <= rightValue) {
        target[targetPos++] = leftValue;
        leftPos++;
      } else {
        target[targetPos++] = rightValue;
        rightPos++;
      }
    }
    // Copy the rest
    while (leftPos < leftLen) {
      target[targetPos++] = leftArray[leftPos++];
    }
    while (rightPos < rightLen) {
      target[targetPos++] = rightArray[rightPos++];
    }
    return target;
  }
```


Quicksort

- Effizienter Sortieralgorithmus von Tony Hoare (1959)
- Ebenso Divide & Conquer
- Array wird anhand von *Pivot-Element* rekursiv geteilt.
- Als Pivot wird oft das letzte Element herangezogen (aber auch andere Varianten sind möglich).
- Trotz Worst-Case-Laufzeit von $O(n^2)$ in der Praxis schneller als Mergesort.
- Der Average-Case $O(n \log n)$ tritt so gut wie immer ein!

Quicksort

```
Function Quicksort(left, right)
    if left < right then
        pidx = partition(left, right);
        quicksort(left, pidx - 1);
        quicksort(pidx + 1, right);
```

```
Function Partition(left, right)
    pivot = a[right];
    i = left;
    j = right - 1;
    while i < j do
        while a[i] < pivot do
            i++;
        while j > left && a[j] ≥ pivot do
            j- -;
        if i < j then
            vertausche a[i] mit a[j];
            i++;
            j- -;
    if i == j && a[j] < pivot then
        i++;
    if a[i] != pivot then
        vertausche a[i] mit a[right];
    return i;
```

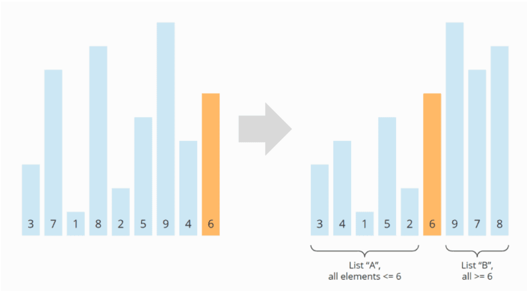
Ablauf (Pivot in []) mit der Zahlenfolge 3, 7, 1, 8, 2, 5, 9, 4, 6:

```
3  7  1  8  2  5  9  4  6

3  7  1  8  2  5  9  4 [6]
3  4  1  8  2  5  9  7  6
3  4  1  5  2  8  9  7  6
3  4  1  5  2  6  9  7  8
3  4  1  5  2 [6] 9  7  8

3  4  1  5 [2] 6  9  7  8
1  4  3  5  2  6  9  7  8
1  2  3  5  4  6  9  7  8
1 [2] 3  5  4  6  9  7  8

1  2  3  5 [4] 6  9  7  8
1  2  3  4  5  6  9  7  8
1  2  3 [4] 5  6  9  7  8

1  2  3  4  5  6  9  7 [8]
1  2  3  4  5  6  7  9  8
1  2  3  4  5  6  7  8  9
1  2  3  4  5  6  7 [8] 9
```


Bestimmung aller Permutationen

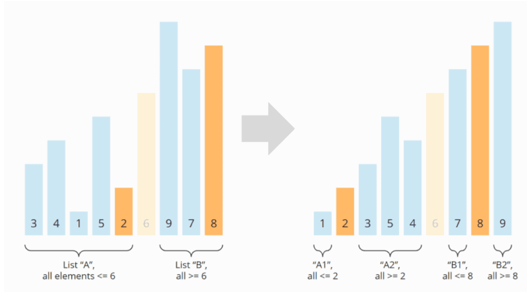
```
Function permutate(left, right)
    if left < right then
        for i=l; i_j=r; i++ do
            quicksort(left, pidx - 1);
            quicksort(pidx + 1, right);
```

Quicksort



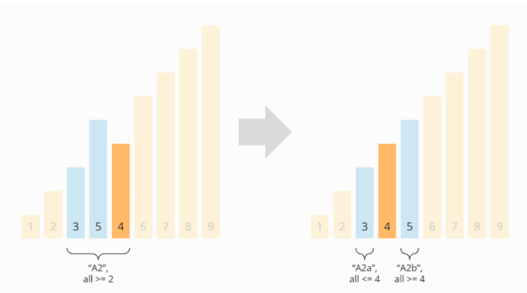
Quelle: `https://www.happycoders.eu/de/algorithmen/quicksort/`

Quicksort



Quelle: `https://www.happycoders.eu/de/algorithmen/quicksort/`

Quicksort



Quelle: `https://www.happycoders.eu/de/algorithmen/quicksort/`

Quicksort

Quelle: `https://www.happycoders.eu/de/algorithmen/quicksort/`

Programmierbeispiel 12.1.1

Gegeben ist ein 2-dimensionales Boolean-Array, befüllt mit *wahr* und *falsch*. Ermitteln Sie mit einem Programm die Größe des größten zusammenhängenden Gebietes mit Wert wahr.

Beispiel: hier wird falsch mit einem '.' und wahr mit einem 'X' dargestellt.

```
...XXX..XX..
.XXX.XXXX...
X.XX..XX..XX
X...XXX..XXX
XXX....XXX..
XX.XX..X..XX
...X..X.XXX.
.....XXXX...
```

In diesem Beispiel ist die Größe 19.

Zusatzaufgabe 12.1.2

Füllen Sie ein zweidimensionales Boolean-Array (Größe 300x300) mit Zufallswerten, wobei mit einer Wahrscheinlichkeit von $1/3$ der Wert wahr, und mit $2/3$ der Wert falsch gesetzt werden soll. Wenden Sie den Algorithmus aus dem vorigen Beispiel an. Ist das Ergebnis plausibel? Was passiert wenn man die Wahrscheinlichkeiten tauscht?

Programmierbeispiel 12.2.1

Implementieren Sie Mergesort in Java.

Programmierbeispiel 12.2.2

Implementieren Sie Quicksort in Java.


```java
private static boolean[][] werte = {
                {false,false,false,true,true,true,false,false,true,true,false,false},
                {false,true,true,true,false,true,true,true,true,false,false,false},
                {true,false,true,true,false,false,true,true,false,false,true,true},
                {true,false,false,false,true,true,true,false,false,true,true,true},
                {true,true,true,false,false,false,false,true,true,true,false,false},
                {true,true,false,true,true,false,false,true,false,false,true,true},
                {false,false,false,true,false,false,true,false,true,true,true,false},
                {false,false,false,false,false,true,true,true,true,false,false,false}
                };
```

Laufzeitanalyse

Programmieren und Software-Engineering Theorie

12. März 2024

Sortieren: Insertion-Sort

Quelle: [?]

Algorithm 1: Insertion Sort

```
Function InsertionSort(array a[])
    Result: sorted array a
    i = 1;
    while i < a.length do
        j = i;
        while j > 0 ∧ a[j − 1] > a[j] do
            swap(a, j, j − 1);
            j = j − 1;
        i = i + 1;
```

- Algorithmus funktioniert wie Karten-Sortieren in der Hand

Sortieren: Selection-Sort

Algorithm 2: Selection Sort

```
Function SelectionSort(array a[])
    Result: sorted array a
    for i = 0; i < a.length − 1; i++ do
        jMin = i;
        for j = i + 1; j < a.length; j++ do
            if a[j] < a[jMin] then
                jMin = j;
        if jMin ≠ i then
            swap(a, i, jMin);
```


Sortieren: Selection-Sort

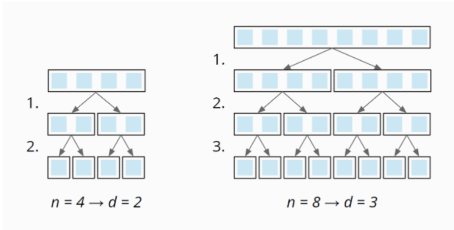
Algorithm 3: Selection Sort

```
1 Function SelectionSort(array a[])
      Result: sorted array a
2     for i = 0; i < a.length − 1; i++ do
3         jMin = i;
4         for j = i + 1; j < a.length; j++ do
5             if a[j] < a[jMin] then
6                 jMin = j;
7         if jMin ≠ i then
8             swap(a, i, jMin);
```

Analyse:

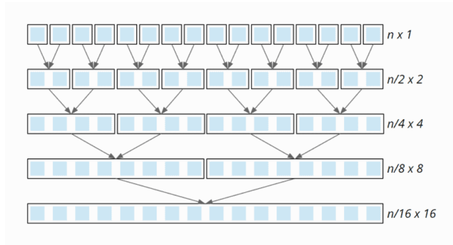
- Sei $n = a.length$
- Die Schleife in Zeile 2 wird $n - 1$ mal durchlaufen
- Durchläufe der inneren Schleife: $(n-1), (n-2), \ldots 1$
- Somit $(n-1) + (n-2) + \ldots 1 = \sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1)$

Analyse von Mergesort



- Insgesamt gibt es $\mathrm{ld} n$ Ebenen der Unterteilung.
- Bei der doppelten Anzahl an Elementen gibt es nur eine zusätzliche Ebene.
- ld steht für den Logarithmus dualis, also jenen zur Basis 2 (da ja immer in zwei Teile geteilt wird).

Analyse von Mergesort



- Ebenso gibt es $\mathrm{ld} n$ Merge-Ebenen.
- In jeder Ebene (bei Unterteilung und Merge) werden n Elemente betrachtet.
- Somit ergibt sich eine Gesamtlaufzeit von $O(n \log n)$.

Analyse von Quicksort

- Analyse von Quicksort insgesamt schwieriger (als bei Mergesort)
- Im **Worst Case** $O(n^2)$
- Dieser wird jedoch nur sehr selten erreicht.
- Im **Durschnittsfall** hat Quicksort eine Laufzeit von $O(n \log n)$
- Somit: gleiche theoretische Laufzeit wie Mergesort
- In der Praxis jedoch schneller als Mergesort!
- Sehr häufig verwendetes Sortierverfahren (z.B. in Bibliotheken)!

Vergleich Sortier-Algorithmen

Vergleich der Laufzeiten von Sortier-Algorithmen im **Worst-Case**. Worst Case bedeutet, dass die Eingabedaten derart ungünstig sortiert sind, dass der Algorithmus die maximale Anzahl an Schritten ausführt. Der Average-Case betrachtet das *durchschnittliche* Verhalten.

Worst-Case Analyse einfacher Sortieralgorithmen:

Algorithmus	**Vergleiche**	**Swaps**
Bubble-Sort	$O(n^2)$	$O(n^2)$
Insertion-Sort	$O(n^2)$	$O(n^2)$
Selection-Sort	$O(n^2)$	$O(n)$

Analyse weiterer Algorithmen:

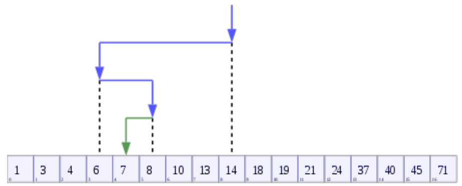
Algorithmus	**Worst-Case**	**Average-Case**
Heap-Sort	$O(n \log n)$	$O(n \log n)$
Merge-Sort	$O(n \log n)$	$O(n \log n)$
Quick-Sort	$O(n^2)$	$O(n \log n)$

Quick-Sort:

- Worst-Case bei Quicksort wird nur selten erreicht
- In der Praxis schneller als Merge-Sort und Heap-Sort

Suche in sortierten Listen

- Aufgabenstellung: Suchen eines Elementes in einer sortierten Liste (Array)
- Naiver Zugang: $O(n)$ Schritte (Erwartungswert $n/2$)
- Mittels binärer Suche: nur $O(\log n)$ Schritte notwendig
- Vorgehensweise: analog zu Suche in Telefonbuch
  - Beliebige Seite aufschlagen
  - Steht Name davor oder danach?
  - Weitere Suche erfolgt nur mehr im verbleibenden Teil



Speicherverbrauch, Konklusion

- Neben Laufzeitperformance ist auch der *Speicherverbrauch* relevant!
- Viele Algorithmen benötigen nur konstanten Speicherverbrauch $O(1)$
- Linearer Speicherverbrauch: zusätzlich $O(n)$ Speicher notwendig
- $O(n^2)$ oder $O(n^3)$ zusätzlicher Speicherverbrauch: funktioniert nur noch für kleine n
- Oft Tradeoff zwischen Laufzeiteffizienz und Speicherverbrauch

Konklusion:

- Bessere Hardware kann Performanceprobleme nur bei $O(n)$, oder evtl. $O(n \cdot \log n)$-Algorithmen beheben
- Bei "schwierigen" Problemen ist durch geschickte Programmierung (=Algorithmik) weitaus mehr zu erreichen

Euler-Zyklen

- Der Hierholzer-Algorithmus hat eine Laufzeit von $O(|E|)$.
- Die Laufzeit ist linear in der Anzahl der Kanten.
- Da $|E| \in O(|V|^2)$ kann die Laufzeit auch als quadratisch in der Anzahl der Knoten angesehen werden.
- Die Aussage $O(|E|)$ ist jedoch stärker, da viele Graphen nicht dicht besetzt sind.

Breitensuche und Tiefensuche

- Es werden alle Pfade zu allen Knoten betrachtet.
- Die Laufzeit ist somit $O(|V| + |E|)$.
- Die Laufzeit ist linear in der Anzahl der Kanten.
- Bei dicht besetzten Graphen führt dies jedoch zu $O(|V|^2)$.

Algorithmus von Dijkstra

Mit $n = |V|$ und $m = |E|$ können wir die Laufzeiteigenschaften angeben. Diese hängen von der konkreten Umsetzung der Prioritätswarteschlange Q ab.

Operation		Queue Implementierung		
Name	Anzahl	Liste	Heap	Fibonacci Heap[1]
decreaseKey	m	$O(1)$	$O(\log n)$	$O(1)$
getMin	n	$O(n)$	$O(\log n)$	$O(\log n)$
create	1	$O(n)$	$O(n)$	$O(n)$
Gesamt		$O(n^2 + m) = O(n^2)$	$O((n + m) \log n)$	$O(n \log n + m)$

- Die Prioritätswarteschlange kann am einfachsten durch ein Array oder eine Liste umgesetzt werden. Bei jedem Aufruf von `getMin` muss die gesamte Liste durchlaufen werden um den kleinsten Eintrag zu ermitteln.
- Heap, bzw. Fibonacci-Heap sind baumbasierte Datenstrukturen, die wir jedoch nicht genauer besprechen.
- Diese führen jedoch zu besseren Laufzeitschranken (und wesentlich schnelleren Implementierungen)

[1] amortisierte Laufzeit

Algorithmus für die Starke Zusammenhangskomponente

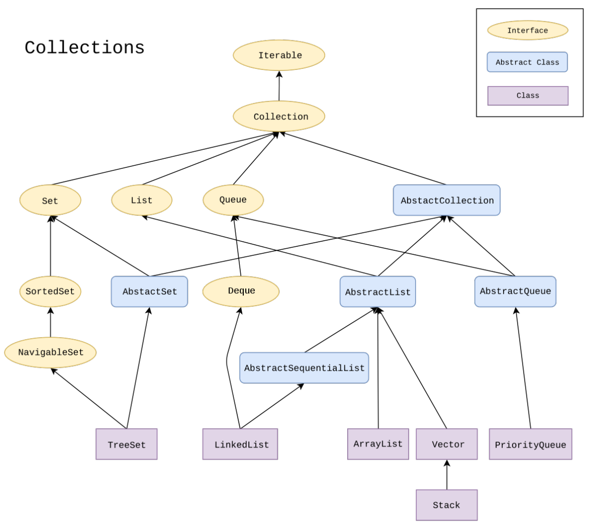
- Die Tiefensuche benötigt $O(|V| + |E|)$ Schritte.
- Die fertiggestellten Knoten können im Zuge der ersten Tiefensuche geordnet gespeichert werden, wodurch kein weiterer Aufwand entsteht.
- Somit ergibt sich der Gesamtaufwand durch zwei Tiefensuchen, und liegt somit insgesamt in $O(|V| + |E|)$.

Java Collections Framework

- Häufige Aufgabe in Programmen: Daten strukturiert abzuspeichern
- Das Java Collections Framework (JCF) enthält *wiederverwendbare* (generische) Klasse und Interfaces für diesen Zweck
- Die wesentliche Elemente der JCF sind: [2]
  - Geordnete Listen
  - Sets (Mengen)
  - Maps ("Dictionaries")
- Weitere Komponenten:
  - Stack
  - Queue

[2] In dieser Form in nahezu allen Programmiersprachen zu finden!

Java Collections Framework

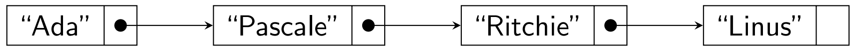


Arrays

- In Java bietet die Collection `ArrayList` einen komfortablen Umgang mit Arrays.
- Auf die Plätze des Arrays kann direkt mittels Index zugegriffen werden.
- Beim Einfügen werden dahinter liegende Elemtente verschoben.
- Beim Löschen werden dahinter liegende Elemtente nach vorne verschoben.

Verkettete Listen

- Verkettete Listen sind eine weitere Möglichkeit zur Umsetzung (Implementierung) des List-Interfaces (in Java).
- Die Collection in Java heißt `LinkedList` und bietet die gleichen Methoden wie `ArrayList`.
- Grundidee: jedes Element erhält einen Verweis ("Zeiger", "Link") auf den unmittelbaren Nachfolger.
- Wesentlicher Unterschied zu ArrayList: es gibt keine vorab definierten Plätze
- Um zum k-ten Element zu gelangen, müssen alle Elemente davor durchlaufen wernden!



Java: `Set`

- Das `Set`-Interface spezifiziert die Schnittstelle eines Sets
- Ein *Set* entspricht einer mathematischen Menge.
- Gleiche Elemente können in einem Set nur einmal enthalten sein.
- In einem Set kann man (im Gegensatz zur `ArrayList`) nicht mittels Index auf bestimmte Elemente zugreifen.
- Das `Set`-Interface wird konkret implementiert in der Klasse `TreeSet` und `HashSet`.

Java: `HashSet`

Wichtige Operationen:

- `HashSet()`: Erzeugt leeres `HashSet` mit Anfangskapazität 16
- `HashSet(int initialCapacity)`: Erzeugt Hashmap mit Anfangskapazität `initialCapacity`
- `boolean add(E e)`: Fügt Objektreferenz `e` hinzu
- `void clear()`: Löscht alle Elemente
- `boolean contains(Object o)`: Gibt an, ob Element `o` enthalten ist
- `boolean isEmpty()`: Gibt an, ob `HashSet` leer ist
- `boolean remove(Object o)`: Entfernt Objektreferenz `o`
- `int size()`: Gibt aktuelle Größe an

HashSet

- In einem `HashSet` werden die Elemente intern in einer *Hash-Table* gespeichert (umgesetzt mittels `HashMap`).
- Wir betrachten also zunächst das Konzept der **Hash-Tabelle** genauer.
- **Ziel beim Hashing ist es, auf bestimmte Objekte schnell zugreifen zu können.**
- Konkret: Zugriffszeit von $O(1)$ (also konstante Zeit), statt $O(n)$ bei Listen (wo bei der Suche alle Elemente durchlaufen werden müssen).

Hash-Tabelle

- Zu jeder abzuspeichernden Objektreferenz emittelt man zunächst ein *"Fach"*, in das es eingeschlichtet werden soll.

Beispiel: Wir definieren ein Ablagesystem für Objekte `Student`, so dass für jeden Anfangsbuchstaben des Nachnamens ein Fach angelegt wird. Bemerkung: In einem Fach können sich mehrere Studenten befinden, aber jeweils mit gleichem Anfangsbuchstaben des Nachnamens!

Hash-Tabelle

Warum benötigt man diese "Fächer" für das `Set`?

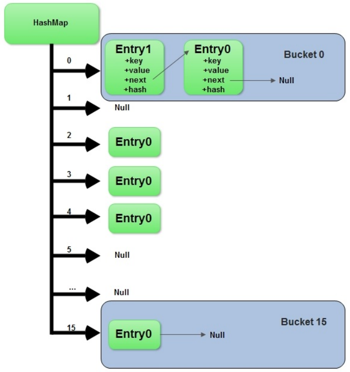
- Beim Hinzufügen eines Objektes muss geprüft werden, ob dieses nicht schon vorhanden ist.
- Durch das Fach-Konzept muss nur in diesem Fach gesucht werden, ob das Objekt schon vorhanden ist.
- **Deutlicher Verbesserung der Ausführungsgeschwindigkeit (Performance)!**

Hashfunktionen

- Hashfunktion bildet *(Schlüssel)*-Werte auf die Zielwerte (Hash-Werte) ab.
- Verschiedene Schlüssel können grundsätzlich den gleichen Hashwert haben $\Rightarrow$ **Kollision**, auch wenn dies nicht wünschenswert ist.
- Kollisionsbehandlung bei Hashtabellen:
  - Mehrere Objekte mit selbem Hashwert in Listen abspeichern
  - Sondieren: Berechnen alternativer Hashwerte
- Eine gute Hashfunktion liefert möglichst wenige Kollisionen!
- Kritischer Tradeoff bei Verwendung von Hash-basierten Datenstrukturen
  - Zu groß ("zuviele Fächer"): hoher Speicherverbrauch
  - Zu klein: viele Kollisionen $\Rightarrow$ Zu viele Elemente landen im selben "Fach" $\Rightarrow$ dadurch schlechtere Performance
- **Anmerkung:** Hashfunktion in der Kryptologie: zusätzliche Anforderung, dass Kollisionen gezielt gefunden werden können.

Verwendung von Hash-Tabellen

- Das Konzept der “Fächer” wird (intern) mit einer Hash-Tabelle (konkret: `HashMap`) umgesetzt.
- Die Fächer werden **Buckets** genannt.
- Für jedes Objekt muss nun ein Bucket berechnet werden!
- Diese Berechnung basiert auf dem Hash-Wert (Java: Methode `hashCode`).
- **Der Hash-Wert modulo der Anzahl der Buckets wird als Bucket-Index verwendet.**



- Grundsätzlich können verschiedene Objekte den selben Hash-Wert erhalten (und dadurch in das selbe Bucket eingeordnet werden).
- **Wichtig: Gleiche** Objekte **müssen** jedoch den gleichen Hash-Wert haben!
- Enthält ein Bucket mehr als eine Objektreferenz, so werden diese intern in einer `Liste` abgespeichert.

Java: `HashSet`

- Zur (korrekten!) Verwendung eines `HashSets` müssen wir also
  1. Hash-Codes zu Objekten berechnen, und
  2. die Gleichheit von Objekten ermitteln können
- Überschreiben der Methoden (aus `Object`):
  1. `int hashCode()`
  2. `boolean equals(Object o)`
- **Wichtig:** *Gleiche* Objekt **müssen** den gleichen Hash-Code haben!
  - Im `HashSet` wird zunächst nur der Hash-Code verglichen.
  - Nur wenn der Hash-Code gleich ist, wird die `equals`-Methode herangezogen

Beispiel (Java): `HashSet`

Beispiel:

- Klasse `Student` mit Attributen `name: String`, `alter: int`
- Verwendung in

```
HashSet<Student> students = new HashSet<>();
```

- Methoden `equals` und `hashCode` müssen in `Student` überschrieben werden!

Beispiel: `HashSet`

```
public class Student {
  ...

  @Override
  public int hashCode() {
    return name.charAt(0); // funktioniert, aber ist sehr unvorteilhaft (*)
  }

  @Override
  public boolean equals(Object other) {
    if (other == null) return false;
    if (other == this) return true;
    if (this.getClass() != other.getClass()) return false;

    Student std = (Student)other;
    if (this.name.equals(std.getName())
        && this.alter == std.getAlter())
    {
      return true;
    }
    return false;
  }
}
```

Warum ist die Hashcode-Berechnung mittels erstem Buchstaben des Namens nicht ideal? Es werden nur 26 verschieden Hashcodes berechnet. Bei einer höheren Bucketanzahl in Hashtabellen werden dadurch die meisten nicht benutzt. Eine gleichmäßige Verteilung der Objekte in die Buckets ist somit nicht gewährleistet.

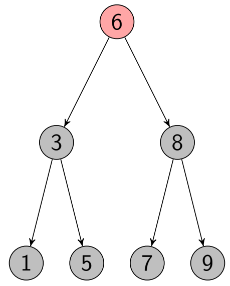
Set mittels Bäumen (Java: `TreeSet`)

- Umsetzung des *Set*-Konzeptes mittels *Bäumen*.
- Konkret: **binäre Suchbäume**.
- Man nennt einen (Wurzel-)Baum *Binärbaum* wenn jeder Knoten höchstens zwei Nachfolger (="Kinder", "Kind-Knoten") hat.
- In einem Suchbaum gilt für jeden Knoten: alle Elemente[3] mit kleineren Werten sind im linken Teilbaum zu finden, jene mit größeren im rechten.
- Ein Binärbaum heißt *vollständig*, wenn alle "Ebenen" bis auf die letzte Ebene vollständig gefüllt sind.
- Ein Binärbaum heißt *voll*, wenn jeder Knoten entweder ein Blatt ist, oder genau zwei Kinder besitzt.
- Binäre Suchbäume weisen günstige Eigenschaften zum Speichern von Elementen auf, wenn Elemente schnell gefunden werden sollen.
- Der wesentliche Grund dafür ist, dass annähernd volle und vollständige Bäume mit n Elementen nur $O(\mathrm{ld} n)$ viele Ebenen besitzen.
- Man nennt die Anzahl der Ebenen auch die **Höhe** des Baumes.

[3] in diesem Teilbaum

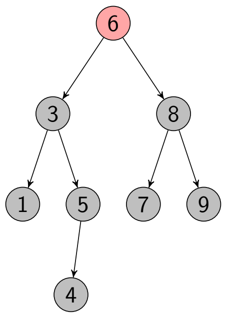
(Binärer) Suchbaum

- Achtung: in der Darstellung ist in den Knoten der *Inhalt* (also ein `int`-Wert) dargestellt (und nicht der Knotenname, bzw. Index)



- Binäre Suchbäume:
  - Ein vollständiger voller Binärbaum mit Höhe d hat $2^d - 1$ Knoten
  - Beispiel: Höhe 3 $\Rightarrow 2^3 - 1 = 7$ Knoten
  - Umgekehrt hat ein vollständiger und voller Binärbaum mit n Knoten eine Höhe von $O(\mathrm{ld}\ n)$

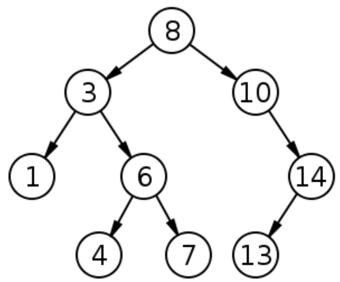
- Schrittweiser Aufbau eines Suchbaumes
- Einfügen von Elementen 6, 3, 5, 8, 1, 7, 9, 4:



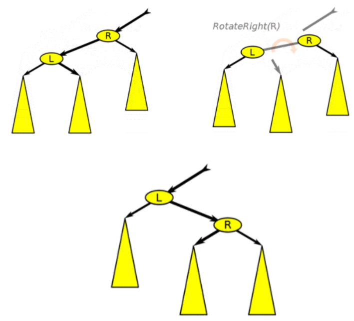
- Das erste Element "6" wird als Wurzelknoten in den Baum aufgenommen.
- Da $3 < 6$ kommt 3 als linker Nachfolger ("Kind") der Wurzel.
- $5 < 6$ also kommt 5 in den linken Teilbaum. Dort befindet sich ein Knoten mit Inhalt 3. Da $5 > 3$ wird 5 als rechtes Kind von 3 in den Baum eingefügt.
- $8 > 6$, das rechte Kind von der Wurzel existiert noch nicht, also kommt 8 dort hin!
- 1 wird im linken Teilbaum am ersten freien Platz eingefügt, also als linkes Kind von 3.
- 7 kommt in den rechten Teilbaum, als linkes Kind von 8
- 9 wird als rechtes Kind von 8 eingefügt.
- 4 wird in einer neuen Ebene als linkes Kind von 5 eingefügt.

Balancierte (binäre) Suchbäume

Binärer Suchbaum:



Balancierte Suchbäume:



Bildquelle: Wikipedia

- In vollständigen binären Suchbäumen können Elemente in maximal $O(\mathrm{ld} n)$ Schritten gefunden werden!
- Balancierung muss gewährleistet werden indem der Baum regelmäßig ausbalanciert wird.
- Konkrete Implementierungen sind z.B. Red-Black-Trees oder AVL-Trees.

TreeSet

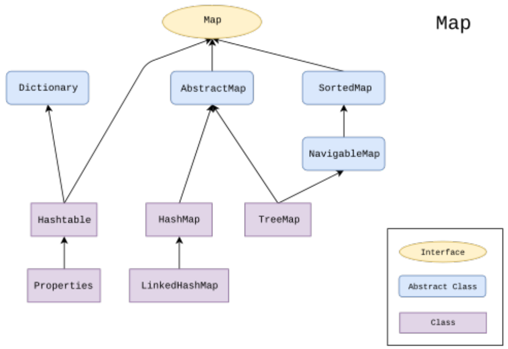
- TreeSet bietet langsamere elementare Operationen: add, remove, contains
- Dafür ist schnelleres sortiertes Durchlaufen der Elemente möglich
- **Bei der Verwendung von TreeSet<E> muss E das Interface Comparable<E> implementieren!** [4]

Wichtige Operationen (zusätzlich zu HashSet):

- `E first()`: liefert kleinstes Element
- `E last()`: liefert größtes Element
- `E higher(E e)`: liefert nächst größeres Element
- `E lower(E e)`: liefert nächst kleineres Element

[4] Alternativ: Comparator

Java: Map-Klassenhierarchie



Map

Bildquelle: Wikipedia

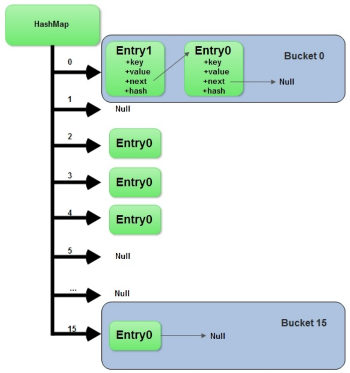
Map

- Maps (oft auch "dictionaries") speichern **Schlüssel – Wert**-Paare.
- Oft möchte man gespeicherte Objekte *schnell* aufgrund eines *Schlüssels* (id, name, ...) finden, bzw. darauf zugreifen.
- Beispiel:
  - Finde Mitarbeiter mit Personalnummer
  - Finde Immobilie mittels Anzeigennummer
- Das Suchen in Arrays, ArrayLists etc. ist aufwändig, da die Datenstruktur zunächst durchsucht werden muss.
- **Ziel:** Auffinden eines Elementes aufgrund des **Schlüssels (Key)** in (quasi) konstanter Zeit $O(1)$, d.h. ohne viele Elemente in der Datenstruktur betrachten zu müssen.
- Die HashMap<K, V> bietet diese Funktionalität an
  - K ... Key
  - V ... Value

HashMap

Funktionsweise:

- Berechnung eines *Hash-Wertes* aus dem Schlüssel (Key)
  - Diese Berechnung erfolgt im Allgemeinen sehr schnell (z.B. im Vergleich zum Durchsuchen eines Arrays)
- Der Hash-Wert wird als Basis für die Index-Berechnung (Buckets) verwendet.
- Zu jedem Schlüssel (Key) wird *genau ein* Wert abgespeichert.



HashMap

Wichtige Operationen:

- `HashMap()`: Erzeugt Hashmap mit 16 Buckets
- `HashMap(int initialCapacity)`: Erzeugt Hashmap mit `initialCapacity` Buckets
- `V put(K key, V value)`: Fügt ein **Schlüssel-Wert-Paar** hinzu. Ein gegebenenfalls zuvor vorhandener Wert mit diesem Schlüssel wird zurückgegeben.
- `V get(Object key)`: Liefert Wert mit entsprechendem Schlüssel, sofern vorhanden.
- `isEmpty()`: Gibt an ob die Hashmap leer ist.
- `Set<K> keySet()`: Liefert alle verwendeten Schlüssel

Beispiel: HashMap

Einfügen und Auslesen von Objektreferenzen:

```java
HashMap<Integer, Student> studenten = new HashMap<>();
Student s1 = new Student("Peter", 20);
Integer id = s1.getId();
studenten.put(id, s1);

...
Student s2 = studenten.get(id);
```

Durch die Werte einer HashMap iterieren:

```java
Iterator<Student> it = studenten.entrySet().iterator();
while (it.hasNext()) {
  Map.Entry<Integer, Student> pair = (Map.Entry)it.next();
  System.out.println(pair.getKey() + ": " + pair.getValue());
}
```


Verwendung Hash- und Tree-Collections

Bei der Verwendung dieser Collections ist zu beachten:

- `HashSet<T>`, `HashMap<K, V>`:
  - Implementierung von `boolean equals(Object o)` in Klassen `T` bzw. `K`
  - Implementierung von `int hashCode()` in Klassen `T` bzw. `K`
- `TreeSet<T>`:
  - Klassen `T` muss interface `Comparable<T>` implementieren

Laufzeitananlyse

Laufzeitanalyse verschiedener Operationen bei Größe n der Collection:

Operation	`ArrayList`	`LinkedList`	`HashSet`	`TreeSet`
add (end)	$O(1)$	$O(1)$	$O(1)$	$O(\text{ld } n)$
remove	$O(n)$	$O(n)$	$O(1)$	$O(\text{ld } n)$
remove (iterator)	$O(n)$	$O(1)$	$O(1)$	$O(\text{ld } n)$
get (index)	$O(1)$	$O(n)$	$O(1)$	$O(\text{ld } n)$
access (position)	$O(1)$	$O(n)$		
find (element)	$O(n)$	$O(n)$	$O(1)$	$O(\text{ld } n)$
insert (position)	$O(n)$	$O(n)$		
insert (iterator)	$O(n)$	$O(1)$		

Anmerkung: In der Tabelle scheint das `HashSet` dem `TreeSet` eindeutig überlegen zu sein. Das `TreeSet` bietet jedoch den Vorteil, dass die enthaltenen Elemente effizient sortiert ausgegeben werden können, und es benötigt wesentlich weniger zusätzlichen Speicher als das `HashSet`.

Homomorphismen

Programmieren und Software-Engineering
Homomorphismen, Formale Sprachen und Syntax-Analyse

12. März 2024

Funktionen und Abbildungen

Definition (Funktion, bzw. Abbildung)

Eine eindeutige Zuordnung von Elementen aus einer Menge A in eine Menge B

$$f : A \to B$$

heißt *Funktion* oder *Abbildung*.

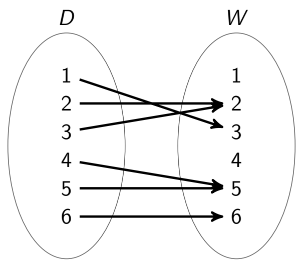
Dabei bezeichnet A die *Definitionsmenge* oder *Urbildmenge*. Die Menge B wird als *Bildmenge* oder *Wertemenge* bezeichnet.

Konkrete Zuordnungen schreibt man als:

$$b = f(a)$$

Bild und Urbild

Beispiel: Definitionsmenge D, Wertemenge W:



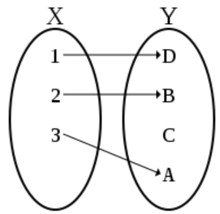
- 3 ist das Bild von 1, hingegen ist 1 das Urbild zu 3
- 5 ist das Bild von $\{4, 5\}$, das Urbild von 5 ist $\{4, 5\}$
- $\{4, 5, 6\}$ ist das Urbild von $\{5, 6\}$

Injektivität, Surjektivität, Bijektivität

Injektivität: Eine Funktion heißt *injektiv*, wenn

- verschiedene Argumente auf verschiedene Funktionswerte abgebildet werden
- Jeder Funktionswert besitzt höchstens ein Urbild
- Ist $f(a) = f(b)$, dann ist $a = b$

Man nennt injektive Funktionen auch *links-eindeutig*.



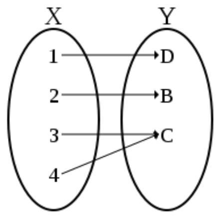
Bildquelle: [?]

Injektivität, Surjektivität, Bijektivität

Surjektivität: Eine Funktion heißt *surjektiv*, wenn

- Jeder Funktionswert wird mindestens einmal durch die Abbildung getroffen.
- Jeder Funktionswert besitzt mindestens ein Urbild.

Man nennt surjektiv Funktionen auch *rechtsvollständig*.



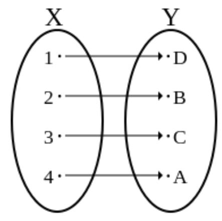
Bildquelle: [?]

Injektivität, Surjektivität, Bijektivität

Bijektivität: Eine Funktion heißt *bijektiv*, wenn

- Die Abbildung ist surjektiv und injektiv.
- Jeder Funktionswert besitzt genau ein Urbild.
- Es gibt für die Funktion eine Umkehrfunktion.

Bijektive Funktionen sind *umkehrbar*.



Bildquelle: [?]

Injektiviät, Surjektivität, Bijektivität

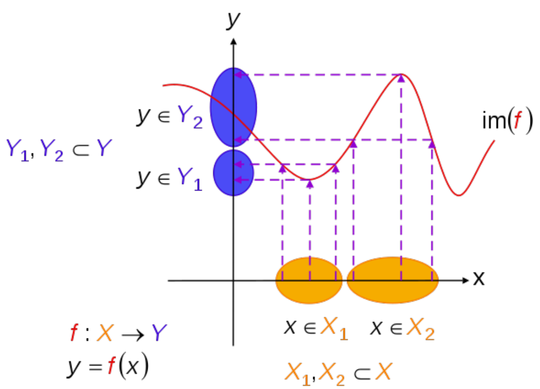


Abbildung: Nicht-injektive Funktion
Quelle: [?]

Beispiele

Beispiel: Sei $f : \mathbb{N} \to \mathbb{N}$ (mit den natürlichen Zahlen $\mathbb{N}$) eine Funktion mit $f(n) = 2 \cdot n$.

$f(n)$ ist injektiv, aber nicht surjektiv.

Aufgaben 1.1.1 – 1.1.3

1. Wir betrachten die Menge $\mathbb{R}$ der reellen Zahlen und die Funktion $f : \mathbb{R} \to \mathbb{R}$, definiert durch $f(x) = 5 \cdot x$. Welche Eigenschaften (injektiv, surjektiv, bijektiv) hat diese Funktion?
2. Welchen Bedingungen gehorcht die Funktion, die einem Kind seine Mutter zuordnet? Was sind dabei die Urbild- und die Bildmenge?
3. Welchen Bedingungen gehorcht die Funktion, die einem Land seine Hauptstadt zuordnet? Was sind dabei die Urbild- und die Bildmenge?

Aufgaben 1.1.4 – 1.1.8

- Sind die folgenden Funktionen $f : \mathbb{R} \to \mathbb{R}$ injektiv, surjektiv oder bijektiv? Begründung!
  1. $f(x) = x^3$
  2. $f(x) = |x|$
  3. $f(x) = \sin(x)$
  4. $f(x) = \pi \cdot x + 35$
  5. $f(x) = \tan(x)$

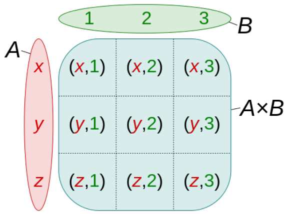
Kartesisches Produkt

Definition (Kartesisches Produkt)

Das *kartesische Produkt* (oder *Kreuzprodukt*) zweier Mengen $A \times B$ ist gegeben durch

$$A \times B = \{(a, b) \mid a \in A \wedge b \in B\}.$$

Anmerkung: Das Kreuzprodukt ist also die Menge aller *geordneten Paare*, wobei das erste Element aus der ersten Menge stammt, und das zweite aus der zweiten.



Innere Verknüpfungen

Definition (Innere Verknüpfung, binäre Operation)

Eine zweistellige Funktion

$$f : A \times A \to A$$

heißt *innere Verknüpfung* oder *binäre Operation*, falls

$$\forall a, b, \in A : f(a, b) \in A.$$

Diese Bedingung wird auch als **Abgeschlossenheit** der inneren Verknüpfung bezeichnet. Eine Menge mit einer inneren Verknüpfung heißt **Gruppoid** und wird vereinfacht durch $\langle A, f \rangle$ dargestellt.

Gruppoid, Halbgruppe, Monoid, Gruppe

Definition

G1 Abgeschlossenheit: $a, b \in G \Rightarrow a * b \in G$ für alle $a, b \in G$

G2 Assoziativgesetz: $(a * b) * c = a * (b * c)$, für alle $a, b, c \in G$

G3 Einheitselement: Es existiert ein Einheitselement $e \in G$, sodaß für alle $a \in G$ gilt: $a * e = e * a = a$.

G4 Inverses Element: Für jedes $a \in G$ existiert ein $a^{-1} \in G$ mit $a * a^{-1} = a^{-1} * a = e$.

G5 Kommutativgesetz: $a * b = b * a$ für alle $a, b \in G$

Gruppoid, Halbgruppe, Monoid, Gruppe

Definition (Gruppoid, Halbgruppe, Monoid, Gruppe)

Eine Struktur heißt:

- **Gruppoid**, wenn G1 erfüllt ist,
- **Halbgruppe**, wenn G1 und G2 erfüllt sind,
- **Monoid**, wenn G1, G2 und G3 erfüllt sind,
- **Gruppe**, wenn G1, G2, G3 und G4 erfüllt sind, und
- **Abel'sche Gruppe**, wenn G1 bis G5 erfüllt sind.

Beispiele

- Gruppoid: $\langle \mathbb{R}^+, * \rangle$ mit $a * b = a^b$
- Halbgruppe: $\langle \mathbb{R}^+, + \rangle$
- Monoid: $\langle \mathbb{N}^+, + \rangle$, (e=0)
- Gruppe: Bewegungen der Ebene
- Abel'sche Gruppe: $\langle \mathbb{Z}, + \rangle$, $\langle \mathbb{R}, + \rangle$, $\langle \mathbb{R}_0 = \mathbb{R} \backslash \{0\}, \cdot \rangle$.

Beispiel

Wählen wir als Grundmenge die natürlichen Zahlen und als innere Verknüpfung die Addition oder als Grundmenge die ganzen Zahlen mit der Subtraktion als binäre Operation, die rationalen Zahlen mit der Multiplikation, die Menge der Aussagen mit Konjunktion, Disjunktion, Implikation oder Äquivalenz, die Menge der Wörter einer Sprache mit der Verkettung der Wörter oder die Menge aller Teilmengen einer Menge (Potenzmenge) mit Durchschnitt, Vereinigung oder Mengendifferenz als innere Verknüpfung, so erhalten wir jeweils Gruppoide.

Wählen wir als Grundmenge die reellen Zahlen und als zweistellige Funktion die Division, so erhalten wir kein Gruppoid. Warum?

Homomorphismen

Definition (Homomorphismus)

Seien $\langle A, * \rangle$ und $\langle B, \circ \rangle$ zwei Gruppoide. Dann heißt die Abbildung $\Phi : A \to B$ *Homomorphismus*, wenn für alle Elemente $a, b \in A$ gilt, daß

$$\Phi(a * b) = \Phi(a) \circ \Phi(b).$$

Man nennt Φ einen

- **Monomorphismus**, wenn Φ injektiv,
- **Epimorphismus**, wenn Φ surjektiv,
- **Isomorphismus**, wenn Φ bijektiv.

Wenn $A = B$, dann heißt ein Homomorphismus **Endomorphismus** und ein Isomorphismus **Automorphismus**.

Darstellung: Isomorphismus

Wir betrachten die Gruppoide $\langle A, * \rangle$ und $\langle B, \circ \rangle$, sowie die Abbildung Φ: $A \to B$:

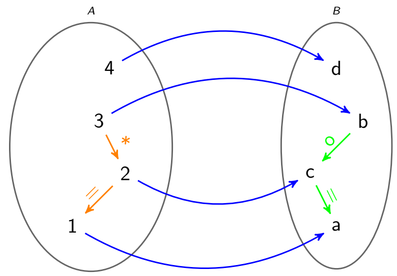


Abbildung $\Phi(1) = a, \Phi(2) = c, \Phi(3) = b, \Phi(4) = d$

Wenn in A gilt, dass 3*2=1,

dann muss in B gelten: $b \circ c = a$.

Diese Eigenschaft muss für alle $x, y \in A$ gelten, damit Φ tatsächlich ein Isomorphismus ist!

Homomorphismen

Beispiel: Sei $A = \{0, 1, 2\}$ und $B = \{a, b, c\}$. Die inneren Verknüpfungen $*$ und $\circ$ für A und B seien gegeben durch:

$*$	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

$\circ$	a	b	c
a	b	c	a
b	c	a	b
c	a	b	c

Homomorphismen

Ein Homomorphismus (Isomorphismus) ist gegeben durch $\Phi(0) = c, \Phi(1) = b$ und $\Phi(2) = a$.

Beweis:

- $\Phi(0 * 0) = \Phi(0) = c = \Phi(0) \circ \Phi(0) = c \circ c$
- $\Phi(0 * 1) = \Phi(1) = b = \Phi(0) \circ \Phi(1) = c \circ b$
- $\Phi(0 * 2) = \Phi(2) = a = \Phi(0) \circ \Phi(2) = c \circ a$
- $\Phi(1 * 0) = \Phi(1) = b = \Phi(1) \circ \Phi(0) = b \circ c$
- $\Phi(1 * 1) = \Phi(2) = a = \Phi(1) \circ \Phi(1) = b \circ b$
- $\Phi(1 * 2) = \Phi(0) = c = \Phi(1) \circ \Phi(2) = b \circ a$
- $\Phi(2 * 0) = \Phi(2) = a = \Phi(2) \circ \Phi(0) = a \circ c$
- $\Phi(2 * 1) = \Phi(0) = c = \Phi(2) \circ \Phi(1) = a \circ b$
- $\Phi(2 * 2) = \Phi(1) = b = \Phi(2) \circ \Phi(2) = a \circ a$

□

Bestimmung aller Isomorphismen

$\circ$	a	b	c	d	e
a	d	b	a	c	b
b	e	a	c	b	d
c	e	a	d	b	a
d	a	c	e	d	a
e	c	c	b	d	e

*	1	2	3	4	5
1					
2				3	
3				2	
4			1		
5					

Wir halten zunächst fest: $2 * 4 = 3$, $3 * 4 = 2$, $4 * 3 = 1$

a	$\to$	2	2	4		4	3		3			3	4		
b	$\to$		3	2	2	1	4	3		4				4	3
c	$\to$	3				2	2	2	2	3	4		3	3	4
d	$\to$	4			3	5		4		2	2	2			
e	$\to$		4	3	4	3			4		3	4	2	2	2
		↯	↯	↯	↯	✓	↯	↯	↯	↯	↯	↯	↯	↯	↯

Die erste mögliche Abbildung wäre mit $a \to 2$ und $d \to 4$, und somit $c \to 3$. Die erste mögliche Abbildung wäre mit $a \to 2$ und $d \to 4$, und somit $c \to 3$.
Wir testen mit der zweiten Gleichung, $3 * 4 = 2$. Es müsste also gelten $c \circ d = a$. ↯ Die nächste mögliche Abbildung wäre mit $a \to 2$ und $e \to 4$, und somit $b \to 3$.
Die nächste mögliche Abbildung wäre mit $a \to 2$ und $e \to 4$, und somit $b \to 3$.
Wir testen $3 * 4 = 2$. Es müsste gelten $b \circ e = a$, tatsächlich gilt jedoch $b \circ e = d$. ↯ Auch hier erhalten wir einen Widerspruch ↯ Auch hier erhalten wir einen Widerspruch ↯ Wir testen nun $c \to 2$ und $a \to 4$. Somit gilt $2 * 4 = 3 \Rightarrow e \to 3$.
Wir prüfen mit der nächsten Gleichung $3 * 4 = 2$. und tatsächlich gilt $e \circ a = c$. Wir testen

Bestimmung aller Isomorphismen

$\circ$	1	2	3	4	5
1	4	2	3	5	5
2	1	3	4	1	3
3	2	4	5	3	1
4	5	3	2	1	3
5	4	4	2	1	3

*	a	b	c	d	e
a					
b				c	
c					
d		a		e	
e					

Wir erhalten: $b * d = c, d * b = a, d * d = e$

$1 \circ 2 = 2$ ↯(derartige Fälle ignorieren wir künftig)
$1 \circ 4 = 5, 4 \circ 1 = 5$ ↯ (da bezüglich * verschiedene Ergebnisse)
$2 \circ 3 = 4, 3 \circ 2 = 4$ ↯ (da bezüglich * verschiedene Ergebnisse, derartige Fälle ignorieren wir künftig)
$2 \circ 4 = 1, 4 \circ 2 = 3 \Rightarrow e \to 5 \Rightarrow d \to 3$ ↯ (da $d \in \{2, 4\}$)
$2 \circ 5 = 3, 5 \circ 2 = 4 \Rightarrow e \to 1 \Rightarrow d \to 4$ ↯ (da $d \in \{2, 5\}$)
$3 \circ 5 = 1, 5 \circ 3 = 2 \Rightarrow e \to 4 \Rightarrow d \to 1$ ↯ (da $d \in \{3, 5\}$)
$4 \circ 5 = 3, 5 \circ 4 = 1 \Rightarrow e \to 2$ ↯ (da nicht in Diagonale)

Es existiert kein Isomorphismus!

Bestimmung aller Isomorphismen

$\circ$	1	2	3	4	5
1					
2			4		
3					
4	1				
5			5		

*	a	b	c	d	e
a	b	c	a	d	d
b	a	a	d	c	e
c	e	a	b	b	c
d	b	a	c	c	e
e	b	a	e	d	e

Lösung: (i) $4 \circ 1 = 1$, (ii) $2 \circ 3 = 4$ und (iii) $5 \circ 3 = 5$

*	a	b	c	d	e
a	b	c	a	<u>d</u>	d
b	<u>a</u>	a	d	c	<u>e</u>
c	e	a	b	b	c
d	b	a	<u>c</u>	c	<u>e</u>
e	b	a	e	<u>d</u>	e

		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)
a	$\to$	**4**	**4**		**1**		3	5		2		5		
b	$\to$	3	3		**4**		**4**	**4**		5		2		
c	$\to$	2	5		2		5	2		**1**		3		
d	$\to$	**1**	**1**		3		2	3		**4**		**4**		**1**
e	$\to$	5	2		5		**1**	**1**		3		**1**		**4**
		↯	↯		↯		↯	↯		↯		✓		↯

- Nach Abbildung auf 1 und 4 betrachtet man bezüglich der verbleibenden Elemente in der Tabelle die Ergebnisse $\Phi(4)$, und kann dadurch viele Fälle ausschließen.
- Es gibt einen Isomorphismus

Isomorphe Graphen

Definition (Isomorphe Graphen)

Zwei Graphen heißen *isomorph*, wenn es eine bijektive Abbildung zwischen den Knotenmengen der beiden Graphen gibt, und außerdem folgende Bedingung erfüllt ist: Zwei Knoten in einem Graphen sind genau dann miteinander verbunden, wenn auch die entsprechenden Bildknoten im anderen Graphen miteinander verbunden sind.

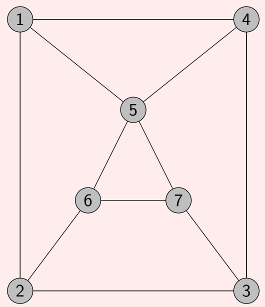
Folgerungen:

- Knoten mit Grad n müssen auf Knoten mit dem selben Grad n abgebildet werden (notwendige Bedingung)
- Alle Wege und Kreise müssen durch die Abbildung erhalten werden.

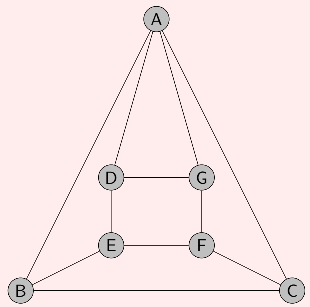
Aufgabe 1.8

Finden Sie alle isomorphen Abbildungen zwischen den folgenden Graphen:

Graph G_1:

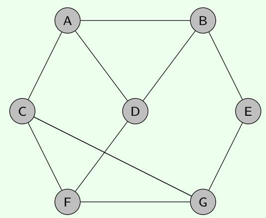


Graph G_2:

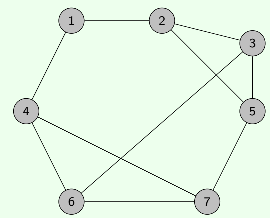


Isomorphe Graphen

G_1:



G_2:



Bestimmung aller möglichen Isomorphismen:

A	$\to$	3	5	6	7
B	$\to$	2	2	4	4
C	$\to$	6	7	3	5
D	$\to$	5	3	7	6
E	$\to$	1	1	1	1
F	$\to$	7	6	5	3
G	$\to$	4	4	2	2

Es gibt nur einen Knoten mit Grad 2, nämlich E. Somit muss E auf den Knoten 1 abgebildet werden. Der Knoten E hat zwei Nachbarknoten B und G, die somit jeweils auf 2 oder 4 abgebildet werden müssen. Es sind also zwei Fälle zu unterscheiden. Für jeden Fall muss untersucht werden, ob er zu einer isomorphen Abbildung führt! (Fallunterscheidung (i) vs. (ii)) Im Fall (i) von $B \to 2$ kann nun $A \to 3$ oder $A \to 5$ abgebildet werden. Die Einträge $E \to 1$, $B \to 2$ sowie $G \to 4$ müssen hierfür in der Tabelle verdoppelt werden (weitere

Aufgabe 1.2

Bestimmen Sie alle Isomorphismen zu den gegebenen Gruppoiden $\langle A, * \rangle$ und $\langle B, \circ \rangle$.

*	1	2	3	4	5
1	3	4	3	5	2
2	2	2	4	3	4
3	4	4	1	5	5
4	2	5	1	3	3
5	3	5	1	2	2

$\circ$	a	b	c	d	e
a	c				
b			b		
c					
d					
e					a

Aufgabe 1.3

Bestimmen Sie alle Isomorphismen zu den gegebenen Gruppoiden $\langle A, * \rangle$ und $\langle B, \circ \rangle$.

*	1	2	3	4	5
1	3	2	5	1	4
2	4	4	3	4	5
3	1	1	4	2	4
4	3	4	3	1	2
5	5	5	1	3	3

$\circ$	a	b	c	d	e
a	c				
b			b		
c					
d					c
e					

Aufgabe 1.4

Bestimmen Sie alle Isomorphismen zu den gegebenen Gruppoiden $\langle A, * \rangle$ und $\langle B, \circ \rangle$.

*	v	w	x	y	z
v	x	v	w	w	x
w	z	y	z	w	v
x	v	z	x	z	y
y	y	w	w	x	z
z	v	w	v	z	y

$\circ$	1	2	3	4	5
1			3		
2		5			
3					
4			4		
5					

Aufgabe 1.5

Bestimmen Sie alle Isomorphismen zu den gegebenen Gruppoiden $\langle A, \circ \rangle$ und $\langle B, * \rangle$.

$\circ$	1	2	3	4	5
1					
2			4		
3					
4	1				
5			5		

*	a	b	c	d	e
a	b	c	a	d	d
b	a	a	d	c	e
c	e	a	b	b	c
d	b	a	c	c	e
e	b	a	e	d	e

Anmerkung: Durch Start mit der Gleichung $5 \circ 3 = 5$ kann man in die weitere Analyse auf drei Fälle ($a * c = a, c * e = c$ und $e * c = e$) beschränken!

Aufgabe 1.6

Bestimmen Sie alle Isomorphismen zu den gegebenen Gruppoiden $\langle A, * \rangle$ und $\langle B, \circ \rangle$.

*	1	2	3	4	5
1	2	5	3	5	1
2	4	3	1	1	3
3	1	4	1	4	5
4	2	5	5	2	4
5	5	1	2	2	3

$\circ$	a	b	c	d	e
a		e		e	
b			c		
c					
d					
e					

Lösung:

		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
1	$\to$	a		e	e		c	c	c	c
2	$\to$	b	d	a	a		b	d	e	e
3	$\to$	c	c	b	d	a	d	b	b	d
4	$\to$	d	b	d	b	e	a	a	d	b
5	$\to$	e		c	c		e	e	a	a
		↯	↯	✓	↯	↯	↯	✓	✓	↯

Aufgabe 1.7

Bestimmen Sie alle Isomorphismen zu den gegebenen Gruppoiden $\langle A, * \rangle$ und $\langle B, \circ \rangle$.

*	1	2	3	4	5
1	2	3	4	5	3
2	3	5	1	3	1
3	4	1	2	2	5
4	2	5	3	3	2
5	4	1	5	1	4

$\circ$	a	b	c	d	e
a				b	
b					
c					a
d	b				
e					

Aufgabe 1.8

Bestimmen Sie alle Isomorphismen zu den gegebenen Gruppoiden $\langle A, *\rangle$ und $\langle B, \circ\rangle$.

*	1	2	3	4	5
1	5	4	2	4	1
2	3	5	1	5	3
3	1	2	3	4	4
4	5	4	5	5	3
5	3	2	5	5	1

$\circ$	a	b	c	d	e
a		d			
b					
c			d		
d			c		
e					

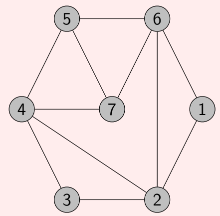
Lösung:

1→e, 2→c, 3→b, 4→a, 5→d
1→b, 2→c, 3→e, 4→a, 5→d

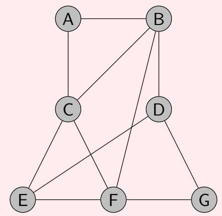
Aufgabe 1.8

- Welche der folgenden Graphen sind zueinander isomorph? Gegeben Sie gegebenenfalls *alle* Isomorphismen an!

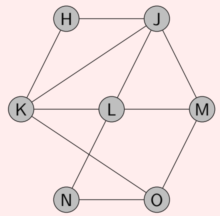
Graph X



Graph Y



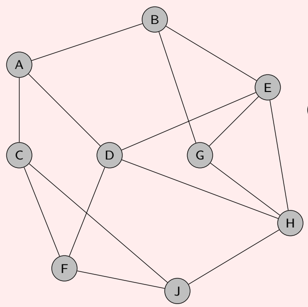
Graph Z



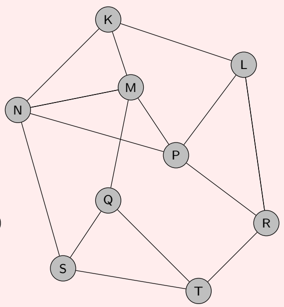
Aufgabe 1.9

- Welche der folgenden Graphen X, Y und Z sind zueinander isomorph? Gegeben Sie gegebenenfalls *alle* Isomorphismen an!

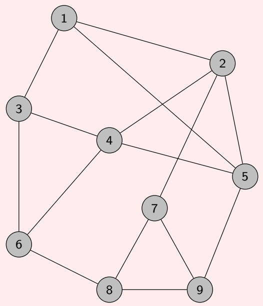
Graph X



Graph Y



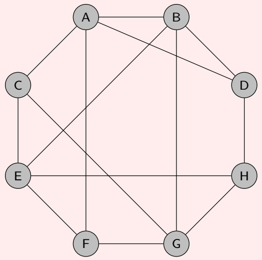
Graph Z



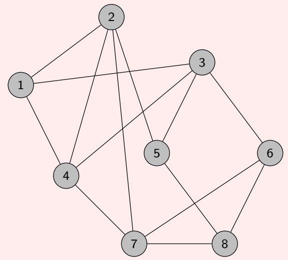
Aufgabe 1.10

- Welche der folgenden Graphen sind zueinander isomorph? Gegeben Sie gegebenenfalls *alle* Isomorphismen an!

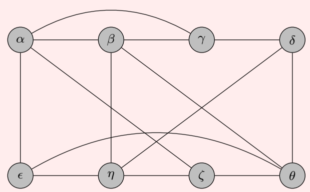
Graph G_1:



Graph G_2:



Graph G_3:



Formale Sprachen

Programmieren und Software-Engineering
Homomorphismen, Formale Sprachen und Syntax-Analyse

12. März 2024

Natürliche und künstliche Sprachen

Grobe Unterscheidung von Sprachen:

- **Natürliche Sprachen:** z.B. Deutsch, Englisch, Spanisch, Latein
- **Künstliche Sprachen:**
  Beispiele:
  - Chemie: Darstellung chemischer Reaktionen
  - Musik: Notenschrift
  - Mathematik, Logik: Formeln, etc.
  - Informatik: Programmiersprachen, Ablaufdiagramme

In der Informatik finden künstliche Sprachen in der Praxis primär als Programmiersprachen Anwendung, jedoch auch in Protokollen. In der theoretischen Informatik spielen formale Sprachen eine zentrale Rolle zur Analyse von Komplexität, Berechenbarkeit etc.

Syntax

Gemeinsamkeiten von natürlichen und künstlichen Sprachen

Alphabet: vorgegebene endliche Menge an Zeichen (Symbolen) aus denen alle Elemente aufgebaut sind.

Grammatik: Regeln auf welche Weise diese Zeichen zu gültigen *Worten* kombiniert werden können.

- Mit *Wort* kann hierbei durchaus auch ein (natürlichsprachiger) Satz gemeint sein

Alphabet und Grammatik bilden die **Syntax** einer Sprache

Semantik

- Aus den syntaktischen Regeln allein können alle Sprachelemente hergeleitet werden, z.B.: Sätze, Kapitel, Bücher,...
- All diese Elemente werden als **Worte** der Sprache bezeichnet.
- Die *Bedeutung* der Worte wird **Semantik** genannt.
- Für Programmiersprachen wird durch die Semantik die Beziehung zwischen Zeichenketten und Aktionen des Rechners hergestellt.
- Die Zuordnung wird durch den Compiler/Interpreter umgesetzt.

Pragmatik

- Die **Pragmatik:** persönliche, subjektive Wahrnehmung oder Interpretation.
- Die Pragmatik einer Programmiersprache definiert ihren Einsatzbereich, d.h. sie gibt an, für welche Arten von Problemen die Programmiersprache besonders gut geeignet ist.
- Beispiele:
  - Kompatibilität
  - Verfügbare Bibliotheken
  - Erlernbarkeit
  - Eignung für Spezialanwendungen
  - Notation des Quellcodes
  - Integrierbarkeit

Beispiele

- Es gilt: die Menge aller möglichen Worte ist größer als die Menge aller syntaktisch korrekten Worte, ist wiederum größer als die Menge aller semantisch korrekten Worte.
- *Beispiel:* Natürliche Sprache Deutsch:
  - Menge der Symbole: Großbuchstaben, Kleinbuchstaben, Ziffern, Satzzeichen
  - Worte über dem Alphabet: qwer4;:rwe?d 39fsd9Ä4fsd
  - Worte der Sprache: "Haus", "Er fährt mit dem Auto", Wörter, Sätze, Texte, Bücher,...
  - Syntaktisch falsch: Der wenn sein heute Projekt sprechen.
  - Syntaktisch korrekt, semantisch falsch: Der Tisch spricht gelb über Informatik.
  - Syntaktisch korrekt, semantisch korrekt: Der Student spricht oft über sein Projekt.

Beispiele

- *Beispiel:* Künstliche Sprache: Notenschrift:
  - Zeichenvorrat: Notensymbole, Schlüssel, Taktstriche,...
  - Syntax: Summe der (aufeinanderfolgenden) Notenwerte im Takt konstant, etc.
  - Semantik: Tonhöhe, Tonlänge,...
  - Pragmatik: Gefühlsmäßige Bestandteile, persönliche Empfindungen, Interpretationen;

Beispiele

Die Beschreibung einer künstlichen (formalen) Sprache erfolgt durch die **Grammatik**.

Die Grammatik umfasst:

- Alphabet T der **Terminalsymbole**
- Menge N der **Nonterminalsymbole**: Hilfsvariablen zur Beschreibung der Sprache
- **Startvariable** $S \in N$.
- **Produktionsregeln** P, auch *Ersetzungsregeln genannt*

Eine Grammatik ist also gegeben durch

$$G = (N, T, P, S).$$

Beispiel: einfache Grammatik für die deutsche Sprache

Beispiel: Für die deutsche Sprache könnte eine primitive Grammatik andeutungsweise folgendes Aussehen haben:

- Alphabet T: 26 Groß- und Kleinbuchstaben, Sonderzeichen, Interpunktionszeichen
- Variablen $N = \{\langle\text{Schriftstück}\rangle, \langle\text{Subjekt}\rangle, \langle\text{Objekt}\rangle, \langle\text{Substantiv im Nominativ}\rangle, \langle\text{Prädikat}\rangle, \langle\text{Substantiv im Akkusativ}\rangle, \ldots\}$.

Anmerkung: Um diese Hilfssymbole als *metasprachliche* Größen zu kennzeichnen, werden sie in spitzen Klammern geschrieben.

- Als Oberbegriff S wählen wir $\langle$Schriftstück$\rangle$.
- Die Produktionsregeln P werden auf der folgenden Seite angegeben. Der Pfeil $\rightarrow$ deutet dabei eine *mögliche* Ersetzung an.

Beispiel: einfache Grammatik für die deutsche Sprache

Produktionsregeln:

⟨Schriftstück⟩ → ⟨Hauptsatz⟩.
⟨Schriftstück⟩ → ⟨Hauptsatz⟩.⟨Schriftstück⟩
⟨Hauptsatz⟩ → ⟨Subjekt⟩⟨Prädikat⟩⟨Objekt⟩
⟨Hauptsatz⟩ → ⟨Subjekt⟩⟨Prädikat⟩
⟨Subjekt⟩ → ⟨Artikel im Nominativ⟩⟨Substantiv im Nominativ⟩
⟨Objekt⟩ → ⟨Akkusativ-Objekt⟩
⟨Akkusativ-Objekt⟩ → ⟨Artikel im Akkusativ⟩⟨Substantiv im Akkusativ⟩

⟨Artikel im Nominativ⟩ → der
⟨Artikel im Nominativ⟩ → die
⟨Artikel im Nominativ⟩ → das
⟨Artikel im Nominativ⟩ → ein
⟨Artikel im Nominativ⟩ → eine
⋮
⟨Substantiv im Nominativ⟩ →Mensch
⟨Substantiv im Nominativ⟩ →Luft
⟨Substantiv im Nominativ⟩ →Wald

⟨Prädikat⟩ → liebt
⟨Prädikat⟩ → geht
⋮
⟨Artikel im Akkusativ⟩ → den
⟨Artikel im Akkusativ⟩ → die
⋮
⟨Substantiv im Akkusativ⟩ →Menschen
⟨Substantiv im Akkusativ⟩ →Luft
⟨Substantiv im Akkusativ⟩ →Wald

Beispiel: einfache Grammatik für die deutsche Sprache

Anmerkung: Betrachten wir die Definition von ⟨Schriftstück⟩ → ⟨Hauptsatz⟩.⟨Schriftstück⟩. Diese erlaubt uns beliebig viele Hauptsätze aneinanderzureihen. Die *Rekursion* endet bei der Verwendung der Regel ⟨Schriftstück⟩ → ⟨Hauptsatz⟩.

Die konkrete Ableitung (tatsächliche Ersetzung) wird durch den Doppelpfeil $\Rightarrow$ notiert. Die dabei jeweils von einem Ableitungsschritt zum anderen entstehende, aus Terminal- und Nonterminalsymbolen aufgebaute Zeichenkette wird als **Satzform** bezeichnet.

Beispiel: einfache Grammatik für die deutsche Sprache

Beispiel: Ableitung eines Satzes:

$$
\begin{aligned}
\langle\text{Schriftstück}\rangle &\Rightarrow \langle\text{Hauptsatz}\rangle. \\
&\Rightarrow \langle\text{Subjekt}\rangle\langle\text{Prädikat}\rangle\langle\text{Objekt}\rangle. \\
&\Rightarrow \langle\text{Artikel im Nominativ}\rangle\langle\text{Substantiv im Nominativ}\rangle \\
&\qquad \langle\text{Prädikat}\rangle\langle\text{Objekt}\rangle. \\
&\Rightarrow \ldots \Rightarrow \\
&\Rightarrow \text{der Mensch liebt } \langle\text{Akkusativ-Objekt}\rangle. \\
&\Rightarrow \text{der Mensch liebt } \langle\text{Artikel im Akkusativ}\rangle \\
&\qquad \langle\text{Substantiv im Akkusativ}\rangle. \\
&\Rightarrow \Rightarrow \ldots \Rightarrow \\
&\Rightarrow \text{der Mensch liebt den Wald.}
\end{aligned}
$$

Es wäre auch möglich gewesen den Satz “der Mensch geht den Wald” abzuleiten. Dieser Satz wäre zwar syntaktisch korrekt, aber semantisch falsch.

Beispiel: Sprache der ganzen Zahlen

Beispiel: Wir definieren korrekte ganze Zahlen anhand einer Grammatik

$T = \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
$N = \{ \langle\text{GanzeZahl}\rangle, \langle\text{VorzlGanzeZahl}\rangle, \langle\text{Ziffer}\rangle, \langle\text{Vorzeichen}\rangle \}$

$$
\begin{aligned}
P = \quad \{ \quad & \langle\text{GanzeZahl}\rangle \rightarrow \langle\text{VorzlGanzeZahl}\rangle, \\
& \langle\text{GanzeZahl}\rangle \rightarrow \langle\text{Vorzeichen}\rangle\langle\text{VorzlGanzeZahl}\rangle, \\
& \langle\text{Vorzeichen}\rangle \rightarrow +, \\
& \langle\text{Vorzeichen}\rangle \rightarrow \text{-}, \\
& \langle\text{VorzlGanzeZahl}\rangle \rightarrow \langle\text{Ziffer}\rangle, \\
& \langle\text{VorzlGanzeZahl}\rangle \rightarrow \langle\text{Ziffer}\rangle\ \langle\text{VorzlGanzeZahl}\rangle, \\
& \langle\text{Ziffer}\rangle \rightarrow 0, \\
& \langle\text{Ziffer}\rangle \rightarrow 1, \\
& \langle\text{Ziffer}\rangle \rightarrow 2, \\
& \langle\text{Ziffer}\rangle \rightarrow 3, \\
& \langle\text{Ziffer}\rangle \rightarrow 4, \\
& \langle\text{Ziffer}\rangle \rightarrow 5, \\
& \langle\text{Ziffer}\rangle \rightarrow 6, \\
& \langle\text{Ziffer}\rangle \rightarrow 7, \\
& \langle\text{Ziffer}\rangle \rightarrow 8, \\
& \langle\text{Ziffer}\rangle \rightarrow 9 \\
\} \quad &
\end{aligned}
$$

$S = \langle\text{GanzeZahl}\rangle$

Beispiel: Sprache der ganzen Zahlen

Mit dem vertikalen Strich |, der verschiedene mögliche Ableitungen voneinander trennt, lassen sich die Produktionsregeln wesentlich kompakter anschreiben:

$$
\begin{aligned}
P = \quad \{ \quad & \langle\text{GanzeZahl}\rangle \rightarrow \langle\text{VorzlGanzeZahl}\rangle \mid \langle\text{Vorzeichen}\rangle\langle\text{VorzlGanzeZahl}\rangle, \\
& \langle\text{Vorzeichen}\rangle \rightarrow + \mid \text{-}, \\
& \langle\text{VorzlGanzeZahl}\rangle \rightarrow \langle\text{Ziffer}\rangle \mid \langle\text{Ziffer}\rangle\ \langle\text{VorzlGanzeZahl}\rangle, \\
& \langle\text{Ziffer}\rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\} \quad &
\end{aligned}
$$

Beispiel: Ableitung von -123

$$
\begin{aligned}
\langle\text{GanzeZahl}\rangle & \Rightarrow \langle\text{Vorzeichen}\rangle\langle\text{VorzlGanzeZahl}\rangle \\
& \Rightarrow \text{-}\langle\text{VorzlGanzeZahl}\rangle \\
& \Rightarrow \text{-}\langle\text{Ziffer}\rangle\langle\text{VorzlGanzeZahl}\rangle \\
& \Rightarrow \text{-1}\ \langle\text{VorzlGanzeZahl}\rangle \\
& \Rightarrow \text{-1}\ \langle\text{Ziffer}\rangle\langle\text{VorzlGanzeZahl}\rangle \\
& \Rightarrow \text{-12}\langle\text{VorzlGanzeZahl}\rangle \\
& \Rightarrow \text{-12}\langle\text{Ziffer}\rangle \\
& \Rightarrow \text{-123}
\end{aligned}
$$

Beispiel: Sprache der ganze Zahlen

Eine Grammatik muss folgende Eigenschaften besitzen:

- Jedes gültige Wort muss ableitbar sein
- Ungültige Worte dürfen nicht ableitbar sein.

Übungsaufgabe

- Welches Problem besteht noch bei der vorangegangen Definition der Sprache der ganzen Zahlen?
- Wie kann es behoben werden?

- +/- 0
- führende 0en

Beispiel: Sprache der ganze Zahlen

Lösung (Variante 1):

$$
\begin{aligned}
\text{P} = \quad \Big\{ \quad & \langle\text{GanzeZahl}\rangle \rightarrow 0 \mid \langle\text{VorzlGZ}\rangle \mid \langle\text{Vorzeichen}\rangle\langle\text{VorzlGZ}\rangle, \\
& \langle\text{Vorzeichen}\ \rangle \rightarrow + \mid -, \\
& \langle\text{VorzlGZ}\rangle \rightarrow \langle\text{PosZiffer}\rangle \mid \langle\text{PosZiffer}\rangle\langle\text{BelZifferZahl}\rangle, \\
& \langle\text{BelZifferZahl}\rangle \rightarrow \langle\text{Ziffer}\rangle \mid \langle\text{Ziffer}\rangle\langle\text{BelZifferZahl}\rangle, \\
& \langle\text{PosZiffer}\rangle \rightarrow 1|2|3|4|5|6|7|8|9, \\
& \langle\text{Ziffer}\rangle \rightarrow 0|1|2|3|4|5|6|7|8|9 \Big\}
\end{aligned}
$$

Lösung (Variante 2):

$$
\begin{aligned}
\text{P} = \quad \Big\{ \quad & \langle\text{GanzeZahl}\rangle \rightarrow \langle\text{Null}\rangle \mid \langle\text{VorzlGZ}\rangle \mid \langle\text{Vorzeichen}\rangle\langle\text{VorzlGZ}\rangle, \\
& \langle\text{Vorzeichen}\ \rangle \rightarrow + \mid -, \\
& \langle\text{VorzlGZ}\rangle \rightarrow \langle\text{PosZiffer}\rangle \mid \langle\text{VorzlGZ}\rangle\langle\text{PosZiffer}\rangle \mid \langle\text{VorzlGZ}\rangle\langle\text{Null}\rangle, \\
& \langle\text{PosZiffer}\rangle \rightarrow 1|2|3|4|5|6|7|8|9, \\
& \langle\text{Null}\rangle \rightarrow 0 \Big\}
\end{aligned}
$$

Backus-Naur-Form

Programmieren und Software-Engineering
Homomorphismen, Formale Sprachen und Syntax-Analyse

12. März 2024

Backus-Naur-Form

Anläßlich der Definition der Programmiersprache ALGOL-58/ALGOL-60 wurde von John Backus und Peter Naur eine neue Darstellungsform **(Backus-Naur-Form, BNF)** für die formale Definition der Syntax von Programmiersprachen entwickelt, die folgende Metasprachlichen Symbole verwendet:

Definition (Backus-Naur-Form: Notation)

- **::=** Definitions-, bzw. Ersetzungszeichen
- $|$ Entweder-Oder-Zeichen
- $<$**, bzw.** $>$ Spitze Klammern zur Kennzeichnung von Variablen

Backus-Naur-Form

Beschreibung des Sprachelementes ⟨ identifier ⟩ der Programmiersprache ALGOL-60 mittels BNF:

$\langle \text{digit} \rangle ::= 0|1|2|3|4|5|6|7|8|9$

$\langle \text{letter} \rangle ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|$
$r|s|t|u|v|w|x|y|z|A|B|C|D|E|F|G|$
$H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|$
$W|X|Y|Z$

$\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle | \langle \text{identifier} \rangle \langle \text{letter} \rangle | \langle \text{identifier} \rangle \langle \text{digit} \rangle$

Anmerkung: Das syntaktische Objekt ⟨identifier⟩ wird aus einer beliebigen Kombination von Buchstaben und Ziffern gebildet, wobei an der ersten Stelle ein Buchstabe stehen muss. Dies wird durch den rekursiven zeichenweisen Aufbau von rechts nach links und durch das Verlassen der Rekursion mittels eines Buchstaben sichergestellt.

Backus-Naur-Form

Beispiel

- Syntaktisch richtige ⟨identifier⟩:
  - buchstabe
  - V17a
  - Feld27
- Syntaktisch falsche ⟨identifier⟩:
  - 1bz
  - $abc
  - f(x)

Ableitung (1)

Rechtsableitung von V17a

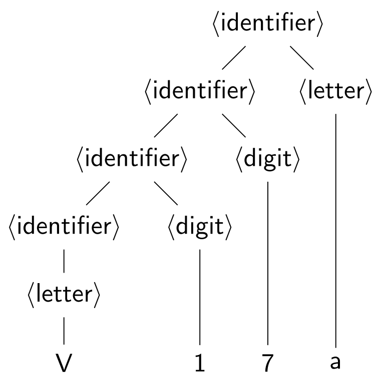
$$
\begin{aligned}
\langle\text{identifier}\rangle &\Rightarrow \langle\text{identifier}\rangle\langle\text{letter}\rangle\\
&\Rightarrow \langle\text{identifier}\rangle\text{a}\\
&\Rightarrow \langle\text{identifier}\rangle\langle\text{digit}\rangle\text{a}\\
&\Rightarrow \langle\text{identifier}\rangle\text{7a}\\
&\Rightarrow \langle\text{identifier}\rangle\langle\text{digit}\rangle\text{7a}\\
&\Rightarrow \langle\text{identifier}\rangle\text{17a}\\
&\Rightarrow \langle\text{letter}\rangle\text{17a}\\
&\Rightarrow \text{V17a}
\end{aligned}
$$

Ableitung (2)

Linksableitung von V17a

$$
\begin{aligned}
\langle\text{identifier}\rangle &\Rightarrow \langle\text{identifier}\rangle\langle\text{letter}\rangle\\
&\Rightarrow \langle\text{identifier}\rangle\langle\text{digit}\rangle\langle\text{letter}\rangle\\
&\Rightarrow \langle\text{identifier}\rangle\langle\text{digit}\rangle\langle\text{digit}\rangle\langle\text{letter}\rangle\\
&\Rightarrow \langle\text{letter}\rangle\langle\text{digit}\rangle\langle\text{digit}\rangle\langle\text{letter}\rangle\\
&\Rightarrow \text{V}\langle\text{digit}\rangle\langle\text{digit}\rangle\langle\text{letter}\rangle\\
&\Rightarrow \text{V1}\langle\text{digit}\rangle\langle\text{letter}\rangle\\
&\Rightarrow \text{V17}\langle\text{letter}\rangle\\
&\Rightarrow \text{V17a}
\end{aligned}
$$

Syntaxbaum / Ableitungsbaum



- Jedem solchen Ableitungs-Prozess kann man einen **Ableitungsbaum**, bzw. **Syntaxbaum**, bzw. **Produktionsbaum** zuordnen.
- Wir zeigen anhand des Ableitungsbaumes, dass V17a ein gültiger ⟨identifier⟩ ist.
- Reihenfolge ist im Ableitungsbaum nicht nachvollziehbar ⇒ somit klar, dass Reihenfolge unerheblich!

Erweiterte Backus-Naur-Form (EBNF)

- Ergänzung um Zeichen { und } in der **Erweiterten Backus-Naur-Form (EBNF)**
- Darin stehende Ausdrücke können *beliebig oft* verwendet werden.
- Verwendung für die Definition von PASCAL

Beschreibung des Sprachelementes ⟨identifier⟩ der Programmiersprache PASCAL mittels EBNF:

$\langle\text{digit}\rangle$ ::= $0|1|2|3|4|5|6|7|8|9$

$\langle\text{letter}\rangle$ ::= $a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|$
$r|s|t|u|v|w|x|y|z|A|B|C|D|E|F|G|$
$H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|$
$W|X|Y|Z$

$\langle\text{letter or digit}\rangle$ ::= $\langle\text{letter}\rangle|\langle\text{digit}\rangle$

$\langle\text{identifier}\rangle$ ::= $\langle\text{letter}\rangle\{\langle\text{letter or digit}\rangle\}$

Anmerkung: Unterstrich war in der ursprünglichen Definition nicht vorgesehen.

Allgemeine Backus-Naur-Form (ABNF)

- Neben { und } können zusätzlich auch die Symbole **[** und **]** verwendet werden
- Die darin enthaltenen Ausdrücke können *höchstens einmal* verwendet werden.
- Anstatt der spitzen Klammern wird für Nonterminale (Variablen) **Fettdruck** verwendet.
- Die Programmiersprachen ADA und MODULA-2 sind in ABNF definiert.

Allgemeine Backus-Naur-Form (ABNF)

Beschreibung des Sprachelementes $\langle$identifier$\rangle$ der Programmiersprache ADA mittels ABNF:

digit ::= 0|1|2|3|4|5|6|7|8|9

letter ::= **lower_case_letter** | **upper_case_letter**

lower_case_letter ::= a | b | c | d | e | f | g | h | i | j | k |
l | m | n | o | p | q | r| s | t| u |
v | w | x | y | z

upper_case_letter ::= A | B | C | D | E | F | G | H | I |J |
K | L | M | N | O | P | Q | R | S | T|
U | V | W | X | Y | Z

letter_or_digit ::= **letter** | **digit**

underscore ::= _

identifier ::= **letter** | { **[underscore] letter_or_digit**}

Mehrdeutigkeit einer Grammatik

- Eine Grammatik G heißt **eindeutig**, wenn für jedes Wort w, das gemäß G aus S ableitbar ist, gilt: die mit unterschiedlichen Ableitungen von w verbundenen Ableitungsbäume sind identisch.
- ebenso, wenn es nur eine einzige Ableitung von w gibt (Spezialfall)
- Eine Grammatik heißt **mehrdeutig**, wenn sie nicht eindeutig ist.
- Ist G mehrdeutig, dann gibt es immer ein w und zwei Ableitungen von w, die unterschiedliche Ableitungsbäume erzeugen.
- In der Praxis von Programmiersprachen werden mehrdeutige Grammatiken nicht benutzt, da sie zu Problemen sowohl bei der Definition der Semantik als auch bei der Programmanalyse führen.
- Eine formale Sprache L kann im i. A. durch mehr als eine Grammatik beschrieben werden.
- Eine formale Sprache L heißt *inhärent mehrdeutig* genau dann, wenn alle Grammatiken für L mehrdeutig sind

Aufgabe 3.1

Die syntaktische Bildung von "sequence" gehorche folgender Menge P von Produktionsregeln:

$\langle$letter$\rangle$ ::= a | b | ... | z
$\langle$digit$\rangle$ ::= 0 | 1 | ... | 9
$\langle$word$\rangle$ ::= $\langle$*letter*$\rangle\langle$word$\rangle$|$\langle$letter$\rangle\langle$digit$\rangle$
$\langle$assignment$\rangle$ ::= $\langle$word$\rangle$ = $\langle$digit$\rangle$
$\langle$sequence$\rangle$ ::= $\langle$assignment$\rangle$; |$\langle$assignment$\rangle$,$\langle$sequence$\rangle$

a) Wie sehen die restlichen Bestimmungsstücke der Grammatik $G = (N, T, P, S)$ aus?

b) Welche der folgenden Symbolketten sind keine richtige "sequence"? Lokalisieren Sie *alle* Fehler!

- x1=3,y1=7;
- aa1=bb1=0,cc1=1;
- i3=1,k2=i
- s5=25;
- klmno1=0
- a12b=3,
- a=1 bb1=7;
- r25=6
- A7=9;

SPL: Simple Programming Language

- Simple Programming Language (SPL) ist die Definition einer extrem einfachen Programmiersprache zur Veranschaulichung von Grammatiken formaler Sprachen.
- Die Sprache verfügt weder über ein *Typsystem*, noch *Methoden* oder gar *Klassen*.
- Dennoch ist die Sprache ausreichend um alle denkbaren Berechnungen durchzufühen, jedoch vergleichsweise umständlich und unkomfortabel.

SPL: Simple Programming Language

Produktionsregeln zur SPL in BNF:

⟨program⟩ ::= ⟨statement list⟩
⟨statement list⟩ ::= ⟨statement⟩|⟨statement list⟩;⟨statement⟩
⟨statement⟩ ::= ⟨io statement⟩|⟨assignment statement⟩|
⟨conditional statement⟩|⟨definite loop⟩|⟨indefinite loop⟩
⟨io statement⟩ ::= read ⟨variable list⟩
⟨io statement⟩ ::= write ⟨variable list⟩
⟨variable list⟩ ::= ⟨variable⟩ | ⟨variable list⟩,⟨variable⟩
⟨assignment statement⟩ ::= ⟨variable⟩ := ⟨expression⟩
⟨conditional statement⟩ ::= if ⟨comparison⟩ then ⟨statement list⟩ fi |
if ⟨comparison⟩ then ⟨statement list⟩ else ⟨statement list⟩ fi
⟨definite loop⟩ ::= to ⟨expression⟩ do ⟨statement list⟩ end
⟨indefinite loop⟩ ::= while ⟨comparison⟩ do ⟨statement list⟩ end

SPL: Simple Programming Language

⟨comparison⟩ ::= ⟨expression⟩⟨relation⟩⟨expression⟩
⟨expression⟩ ::= ⟨term⟩|⟨expression⟩⟨weak operator⟩⟨term⟩
⟨term⟩ ::= ⟨element⟩|⟨term⟩⟨strong operator⟩⟨element⟩
⟨element⟩ ::= ⟨constant⟩|⟨variable⟩|(⟨expression⟩)
⟨constant⟩ ::= ⟨digit⟩|⟨constant⟩⟨digit⟩
⟨variable⟩ ::= ⟨letter⟩|⟨variable⟩⟨letter⟩|⟨variable⟩⟨digit⟩
⟨relation⟩ ::= = | <= | < | > | >= | <>
⟨weak operator⟩ ::= + | -
⟨strong operator⟩ ::= * | /
⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨letter⟩ ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
n | o | p | q | r| s | t| u | v | w | x | y | z

SPL: Simple Programming Language

Aufgabe 3.2

- Gibt es Syntaxfehler in folgendem SPL-Programm?
- Was würde es tun wenn es korrekt wäre?

```
read n;
to n do
  read x;
  if x>0 then
    y := 1;
    z := 1;
    while z<>x do
      z := z+1;
      y := y*z;
    end;
    write y
  fi
end
```

SPL: Simple Programming Language

Aufgabe 3.3

Entwickeln Sie einen SPL-Dialekt, indem der Strichpunkt Bestandteil *jeder* Anweisung ist (und nicht nur Anweisungen voneinander trennt).

Definition (Perfekte Zahl)

Unter einer perfekten Zahl versteht man eine natürliche Zahl, deren Wert gleich der Summe *aller* ihrer echten Teiler ist.

Beispiel: $6 = 1 + 2 + 3, \quad 28 = 1 + 2 + 4 + 7 + 14, \quad 496 = \dots$

Aufgabe 3.4

Entwickeln Sie ein kleines SPL-Programm, das eine positive ganze Zahl n einliest, und die kleinste perfekte Zahl größer n ausgibt.

SPL: Simple Programming Language

Aufgabe 3.5

Zeigen Sie, dass es sich bei

```
while z <> x do
    z := z + 1;
    y := y * z
end
```

um ein gültiges SPL-Statement handelt.

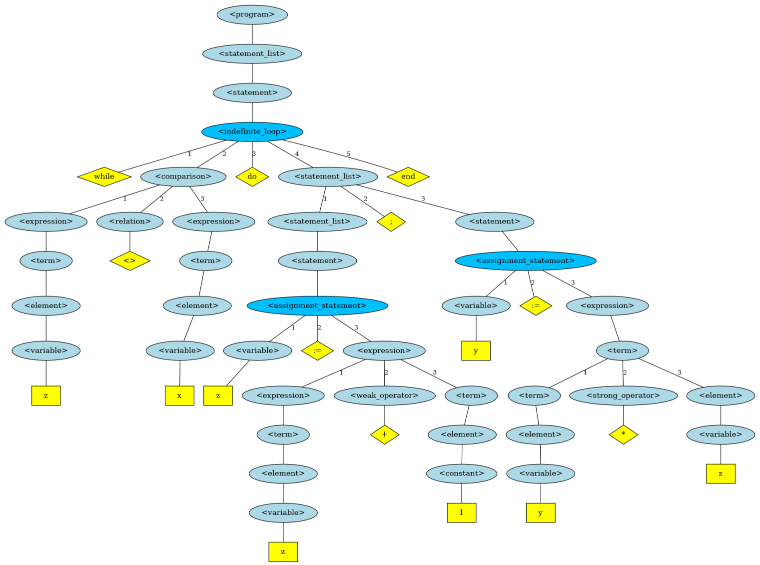
Lösung (3.5): Wir beginnen die Ableitung mit ⟨*statement*⟩ (laut Angabe!). Bei der Ableitung werden einige Schritte übersprungen. Geben Sie bei Tests oder Abgaben aber stets alle Zwischenschritte an!

⟨statement⟩
⇒ ⟨indefinite loop⟩
⇒ while ⟨comparison⟩ do ⟨statement list⟩ fi
⇒ while ⟨expr⟩⟨rel⟩⟨expr⟩ do
        ⟨statement list⟩;
        ⟨statement⟩
    end
⇒ . . . ⇒
    while z <> x do
        ⟨statement⟩;
        ⟨statement⟩
    end

Fortsetzung (Lösung 3.5)

⇒ . . . ⇒
    while z <> x do
        ⟨assignment statement⟩;
        ⟨assignment statement⟩
    end
⇒ . . . ⇒
    while z <> x do
        ⟨variable⟩ := ⟨expression⟩;
        ⟨variable⟩ := ⟨expression⟩
    end
⇒ . . . ⇒
    while z <> x do
        z := z + 1;
        y := y * z
    end

Alternative Lösung zu 3.5 mittels Syntaxbaum:



Aufgabe 3.6

- Gibt es Syntaxfehler in folgendem SPL-Programm?
- Was würde es tun wenn es korrekt wäre?

```
read n
if n>0 then
    a = 1;
    b = 1;
    write 1;
    if n>1 then
        write 1;
        to n-2 do
            c = a + b;
            write c;
            a = b;
            b = c;
        end;
    end
fi
```

Grammatiken

Programmieren und Software-Engineering
Homomorphismen, Formale Sprachen und Syntax-Analyse

12. März 2024

Worte der Länge n

- Ein **Alphabet** ist eine nichtleere endliche Menge, deren Elemente Zeichen oder Symbole genannt werden.
- Ein **Wort der Länge** n ist eine Folge $x_1x_2 \dots x_n$ von n Symbolen des Alphabets A
- Jede beliebige geschlossene Gruppe von Zeichen innerhalb eines Wortes wird als **Teilwort** bezeichnet.
- Zwei Wörter x und y werden als *gleich* bezeichnet, wenn sie die gleiche Länge besitzen und auch die Reihenfolge der Zeichen gleich ist.
- Sei T das Alphabet, so bezeichnet T^+ die Menge aller daraus konstruierbaren Wörter.

Verkettung

Definition (Verkettung)

Sind $x = x_1x_2 \dots x_n$ und $y = y_1y_2 \dots y_m$ zwei beliebige Wörter aus der Menge T^+, dann ist die **Zusammensetzung oder Verkettung** definiert als

$$xy = x_1x_2 \dots x_ny_1y_2 \dots y_m.$$

- Wird zu T^+ ein spezielles Wort ε (Epsilon) hinzugefügt, so erhält man die Menge T^*.

$$T^* = T^+ \cup \{\varepsilon\}$$

- ε wird als **Leerwort** bezeichnet. Es ist kein Terminalsymbol (und somit auch nicht im Alphabet enthalten).
- Es gilt:

$$x\varepsilon = \varepsilon x = x$$

Länge eines Wortes

- Die **Länge eines Wortes** x ist durch die Anzahl der Zeichen des Wortes bestimmt:

$$|x_1x_2 \dots x_n| = n$$

- Es gilt:

$$|xy| = |x| + |y|$$

- Für das Leerwort gilt:

$$|\varepsilon| = 0$$

Verkettung von Wortmengen

Definition (Verkettung von Wortmengen)

Seien M und N zwei Wortmengen. Die Verkettung MN ist definiert als

$$MN = \{x \mid x = yz, y \in M, z \in N\}.$$

- Die als Ergebnis entstehende Menge enthält alle jene Worte, die aus einem Wort der Menge M verkettet mit einem Wort der Menge N bestehen.
- Die Menge MN wird als das Produkt von M und N bezeichnet.
- Für die Produktoperation gilt das Kommutativgesetzt **nicht**!
- Somit gilt im Allgemeinen: $MN \neq NM$

Verkettung von Wortmengen

Beispiel: Gegeben seien zwei Alphabete Q und R sowie die beiden Wortmengen $M \subseteq Q^+$ und $N \subseteq R^+$.

$$\begin{array}{ll} Q = \{a, b, c\} & M = \{c, bba, bcacc\} \\ R = \{D, E, F, G\} & N = \{FE, DDF\} \end{array}$$

Man beachte den Unterschied

$$MN = \{cFE, cDDF, bbaFE, bbaDDF, bcaccFE, bcaccDDF\}$$

$$NM = \{FEc, FEbba, FEbcacc, DDFc, DDFbba, DDFbcacc\}$$

Verkettung von Wortmengen

Die Operationen "+" und "*" können auch auf Wortmengen angewandt werden.

Beispiel: Gegeben sei die Wortmenge $M = \{c, bba, bcacc\}$.
Eine mögliche Teilmenge Z von M^* ist:

$$Z = \{\varepsilon, c, ccccc, cbbac, cbbabcacc, bbabbabcaccc, cccbbacc\}$$

WH: Formale Sprache

- Eine *formale Sprache* L über einem Alphabet T ist eine beliebige Teilmenge von T^*.
- Grammatiken sind formale Systeme die zur Erzeugung formaler Sprachen dienen.
- Grammatiken sind definiert als

$$G = (N, T, P, S),$$

  für die gilt:
  - Alphabet der Nichtterminalsymbole N
  - Alphabet der Terminalsymbole T mit $N \cap T = \emptyset$.
    Weiters sei $V := N \cup T$.
  - Menge der Produktionsregeln P
    Eine Regel schreiben wir abstrakt als $\omega_1 \rightarrow \omega_2$, bzw. $(\omega_1, \omega_2) \in P$.
  - Startvariable S mit $S \in N$

Produktionsregeln

- Wir untersuchen nun die linke Seite ω_1 und die rechte Seite ω_2 von Produktionsregeln $\omega_1 \to \omega_2$ genauer.
- Sei $\omega_1 \in N$, so besteht die linke Seite der Regel jeweils nur aus einem Nonterminal
- Sei $\omega_2 \in NT$, so ist die rechte Seite der Regel von der Form: ein Nonterminalsymbol gefolgt von einem Terminalsymbol
    - Hier wurde die Verkettung NT der Mengen N und T verwendet.
- Sei $\omega_2 \in T \cup TN$, so besteht die rechte Seite der Regel entweder aus einem Terminalsymbol, oder aus einem Terminalsymbol gefolgt von einem Nonterminalsymbol.

Chomsky-Hierarchie

- Die **Chomsky-Hierarchie** unterscheidet verschiedene Arten von Grammatiken aufgrund der Beschaffenheit ihrer Produktionsregeln.
- Grammatiken vom **Typ-0**, **Typ-1**, **Typ-2** und **Typ-3** unterscheiden sich durch ihre Mächtigkeit.
- Die Typ-0 Grammatiken weisen die größte Mächtigkeit auf, während Typ-3 Grammatiken nur relativ einfache Sprachen erzeugen können.

Chomsky-Hierarchie

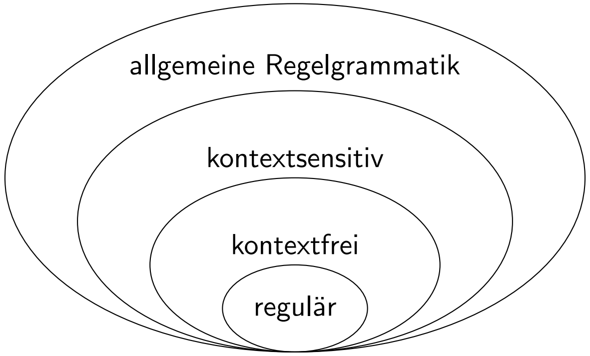
- **(Rechts-)Reguläre Grammatik (Typ-3-Grammatik):**
  Links von $\to$ steht genau ein Nonterminal. Rechts von $\to$ steht entweder ein Terminalsymbol oder ein Terminalsymbol gefolgt von genau einem Nonterminal
- **Kontextfreie Grammatik (Typ-2-Grammatik):**
  Links von $\to$ steht genau ein Nonterminal, rechts davon eine beliebige Folge von Terminalen und Nonterminalen (genauer aus V^*)
- **Kontextsensitive (umgebungsabhängige) Grammatik (Typ-1-Grammatik):**
  Hier sind zusätzlich Produktionsregeln erlaubt, auf deren linker Seite auch Terminalsymbole vorkommen, jedoch mindestens ein Nonterminal enthalten ist. Dieser "Kontext" der Terminalsymbole muss auf der rechten Seite der Produktionsregel erhalten bleiben.
- **Allgemeine Regelgrammatik (Typ-0-Grammatik):**
  Die Einschränkungen für die rechte Seite bei der Typ-1-Grammatik gelten hier nicht.

Chomsky-Hierarchie (*)

- **(Rechts-)Reguläre Grammatik (Typ-3-Grammatik):**
  $$\forall(\omega_1 \to \omega_2) \in P : \omega_1 \in N \wedge \omega_2 \in T \cup TN \cup \{\varepsilon\}$$
- **Kontextfreie Grammatik (Typ-2-Grammatik):**
  $$\forall(\omega_1 \to \omega_2) \in P : \omega_1 \in N \wedge \omega_2 \in V^*$$
- **Kontextsensitive (umgebungsabhängige) Grammatik (Typ-1-Grammatik):**
  Alle Produktionsregeln sind von der Form:
  $$\alpha A \beta \to \alpha\gamma\beta \text{ mit } A \in N \wedge \alpha, \beta, \gamma \in V^*,$$
  oder $S \to \varepsilon$
- **Allgemeine Regelgrammatik (Typ-0-Grammatik):**
  $$\forall(\omega_1 \to \omega_2) \in P : \omega_1 \in V^+ \wedge \omega_2 \in V^*$$
  wobei ω_1 mindestens ein Nonterminal enthalten muss.

Chomsky-Hierarchie

Die Grammatiken höheren Typs sind in jenen niedrigeren Typs jeweils vollständig enthalten:



Beispiele zur Chomsky-Hierarchie

Beispiel:

T-3: ⟨A⟩ → x
⟨A⟩ → x⟨B⟩

T-2: ⟨A⟩ → x⟨A⟩y⟨B⟩c
⟨conditional statement⟩ → if ⟨expression⟩ then ⟨statementlist⟩ fi

T-1: x⟨A⟩y → x⟨B⟩z⟨C⟩w⟨D⟩⟨E⟩y

T-0: .... → ..... (mindestens ein Nonterminal auf der linken Seite)
x⟨A⟩ → ⟨B⟩

Chomsky-Hierarchie

Sprachen höheren Typs sind *echte Teilmengen* der Sprachklassen niederen Typs.

Grammatik	**Sprache**	**Automat**
Typ-0	rekursiv aufzählbar	nichtdeterministische Turingmaschine
Typ-1	kontextsensitiv	nichtdeterministische Turingmaschine mit linear beschränkter Bandlänge
Typ-2	kontextfrei	nichtdeterministischer Kellerautomat
Typ-3	regulär	endlicher Automat

Programmiersprachen sind im Allgemeinen kontextfreie Sprachen!

Aufgaben

Aufgabe 3.8

- Welche Regeln von SPL sind regulär?
- Wie können die anderen Regeln klassifiziert werden?

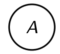
Reguläre Sprachen und Endliche Automaten

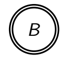
Programmieren und Software-Engineering
Homomorphismen, Formale Sprachen und Syntax-Analyse

12. März 2024

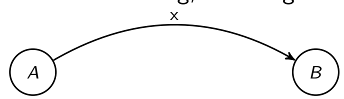
Einleitung

- Worte regulärer Sprachen können durch **Endliche Automaten** auf syntaktische Korrektheit überprüft werden.
- Ein *Endliche Automat* ist ein abstraktes Konzept, bestehend aus:
  - **Zuständen** (dargestellt durch Knoten):
    - Ein Startknoten 

    - Allgemeine Knoten 

    - Ein oder mehrere Endknoten 

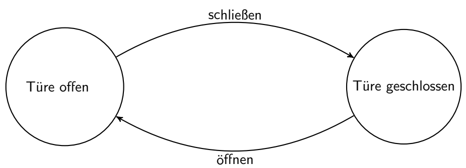
  - **Zustandsübergängen:**
    - dargestellt durch Kanten mit Beschriftung (Aktion der Zustandsänderung, z.B. eingelesenes Zeichen)



- Ein endlicher Automat ist also ein gerichteter Graph

Echtwelt-Beispiel: Endlicher Automat

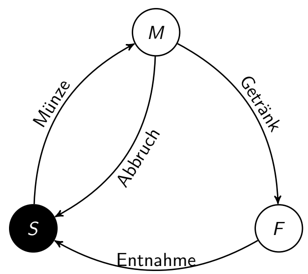
- Automaten *beschreiben* also reguläre Sprachen...
- ... jedoch auch Anwendungen/Maschinen ("Automaten") der "echten Welt"
- Beispiel eines Endlichen Automaten der eine Türe beschreibt:



- Zustände: Türe offen oder geschlossen
- "schließen" und "öffnen" sind Zustandsübergänge

Echtwelt-Beispiel: Getränke-Automat

Beispiel: Getränkeautomat:

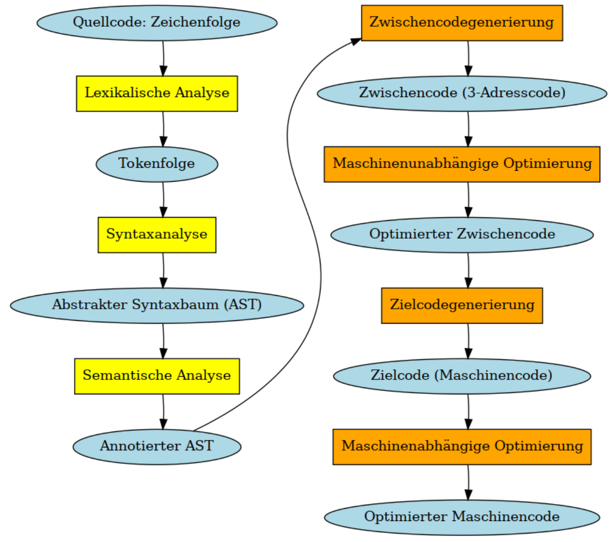


- Startzustand S (in dieser Abbildung durch einen Pfeil gekennzeichnet)
- Getränkeausgabe erst von Zustand M aus möglich
- Mit Münze gelangt man von S zu M
- Von M (also nach eingeworfener Münze) kann man entweder Abbrechen, oder ein Getränk wird ausgegeben.
- Nach ausgegebenem Getränk (Zustand G) ist die Entnahme möglich. Erst nach der Entnahme akzeptiert der Automat in S wieder weitere Münzen.

Syntaxanalyse in Compilern

- Viele Elemente von Programmiersprachen werden durch reguläre Sprachen beschrieben.
- Erster Schritt der Compilierung ist die *Syntaxanalyse*.
- Die erste Phase dabei ist die *lexikalische Analyse* ("Scanner", "Lexer")
- In der lexikalischen Analyse wird der Quelltext in zusammengehörige Einheiten "Tokens" zerteilt.
- Die Tokens werden dann *Schlüsselworten* (Keywords) oder *Bezeichnern* (Identifiers) zugeordnet.
- Bezeichner können anhand endlicher Automaten auf Korrektheit geprüft werden.
- Die Phasen eines Compilers sind auf der nächsten Folie dargestellt.

Phasen eines Compilers

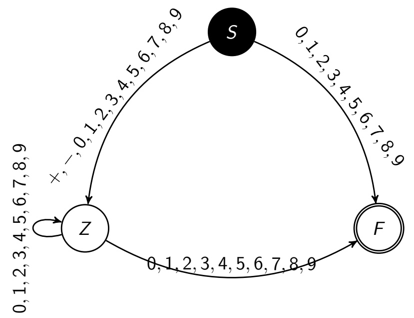


Automaten als Akzeptoren

- Ein endlicher Automat überprüft Worte regulärer Sprachen auf Korrektheit, d.h. er **akzeptiert** syntaktisch korrekte Worte (syntaktisch falsche Worte werden nicht akzeptiert).
- Dabei liest der Automat Zeichen für Zeichen des Wortes, und führt entsprechende Zustandsübergänge durch.
- Ein Wort ist gültig, wenn sich der Automat nach dem letzten gelesenen Zeichen in einem Endzustand befindet.
- Ein Wort ist ungültig, wenn es zu einem gelesenen Zeichen keinen entsprechenden Zustandsübergang gibt, oder wenn nach dem letzten gelesenen Zeichen kein Endzustand erreicht wurde.

Sprache der ganzen Zahlen (mit führenden 0en, oder ± 0)

Konvention: wenn Startsymbol nicht explizit angegeben, dann "S".



Beispiel:
Ableitung von -144

$\langle S \rangle \Rightarrow$ - $\langle Z \rangle$
$\langle Z \rangle \Rightarrow$ 1 $\langle Z \rangle$
$\langle Z \rangle \Rightarrow$ 4 $\langle Z \rangle$
$\langle Z \rangle \Rightarrow$ 4

$T = \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
$N = \{S, Z\}$
$P = \{S \rightarrow$ +Z|-Z|0Z|1Z|2Z|3Z|4Z|5Z|6Z|7Z|8Z|9Z|0|1|2|3|4|5|6|7|8|9,
Z $\rightarrow$ 0Z|1Z|2Z|3Z|4Z|5Z|6Z|7Z|8Z|9Z|0|1|2|3|4|5|6|7|8|9}

Sprache der ganzen Zahlen (ohne führende 0en, ohne ± 0)

Aufgabe 3.10

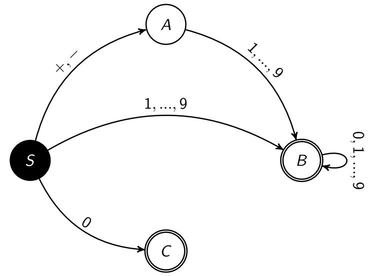
Adaptieren Sie die Grammatik der Sprache der ganzen Zahlen mit führenden 0en, sodaß diese nicht mehr möglich sind, und ebenso ±0... ausgeschlossen wird.

$P = \{S \rightarrow$ +A|-A|1B|2B|3B|4B|5B|6B|7B|8B|9B|0|1|2|3|4|5|6|7|8|9,
A $\rightarrow$ 1B|2B|3B|4B|5B|6B|7B|8B|9B|1|2|3|4|5|6|7|8|9,
B $\rightarrow$ 0B|1B|2B|3B|4B|5B|6B|7B|8B|9B|0|1|2|3|4|5|6|7|8|9$\}$

Deterministischer Endlicher Automat

- Bei einem *deterministischen endlichen Automaten* sind die Zustandsübergänge *eindeutig*!
- Bei den Kantenbeschriftungen der von einem Knoten auslaufenden Kanten kommt eine Beschriftung also nur ein mal vor.

Beispiel: Deterministischer endlicher Automat (DEA) für die Sprache der Ganzen Zahlen ohne führende 0en und ohne ± 0:



$T = \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
$N = \{S, A, B\}$
$P = \{S \rightarrow$ +A|-A|1B|2B|...|9B|0|1|...|9,
A $\rightarrow$ 1B|2B|...|9B|1|...|9,
B $\rightarrow$ 0B|1B|2B|...|9B|0|1|...|9 $\}$

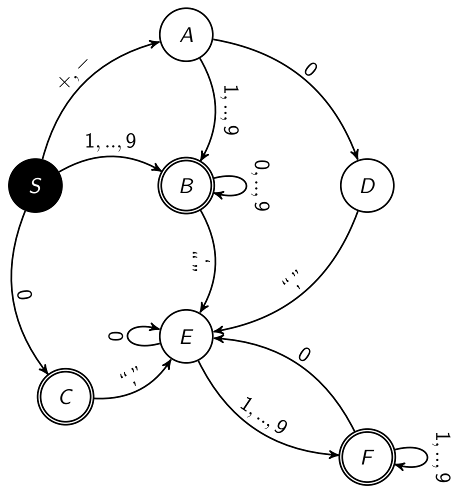
Sprache der reellen Zahlen

$P = \{S \rightarrow$ +A|-A|1B|2B|3B|4B|5B|6B|7B|8B|9B|0C|0|1|...|9;
A $\rightarrow$ 1B|2B|3B|4B|5B|6B|7B|8B|9B|0D|1|...|9;
B $\rightarrow$ 0B|1B|2B|3B|4B|5B|6B|7B|8B|9B|,E;
C $\rightarrow$,E;
D $\rightarrow$,E;
E $\rightarrow$ 0E|1F|2F|3F|4F|5F|6F|7F|8F|9F|1|...|9;
F $\rightarrow$ 0E|1F|2F|3F|4F|5F|6F|7F|8F|9F|1|...|9
$\}$

Anmerkung: Hier wird der Strichpunkt als Trennzeichen für die einzelnen Produktionsregeln verwendet.

Endlicher Automat (Finite State Machine)

Erkennung einer rationalen Zahl (z.B. -0,425) durch einen endlichen Automaten. Nicht gültig wären ("00,4" oder "-,3" oder "3,000" oder "01,")



Beispiel: reguläre Sprache

Gesucht ist eine Sprache mit Worten $W \in \{a, b, c, d\}^+$ in der jedoch die Teilworte "dda" und "bdcb" *nicht* vorkommen dürfen.

$T = \{a, b, c, d\}$
$N = \{S, T, A, B, C, D, E\}$
$P = \{S \rightarrow$ aT | bA | cT | dB | a | b | c | d,
$T \rightarrow$ aT | bA | cT | dB | a | b | c | d,
$A \rightarrow$ aT | bA | cT | dC | a | b | c | d, // G1W2
$B \rightarrow$ aT | bA | cT | dD | a | b | c | d, // G1W1
$C \rightarrow$ aT | bA | cE | dD | a | b | c | d, // G2W2 $\wedge$ G1W1
$D \rightarrow$ bA | cT | dD | b | c | d, // G2W1
$E \rightarrow$ aT | cT | dB | a | c | d // G3W2
}

GxWy ... "Gefahrenstufe x, Wort y"

Beispiel: reguläre Sprache

Gesucht ist die reguläre Grammatik der Sprache

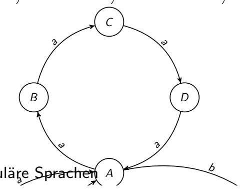
$$L = \{a^i b^j c^k \mid i \equiv 1(4), j \equiv 0(3), k \equiv 2(3)\}^*$$

$i \equiv 1(4)$, bzw. $i \equiv 1 \mod 4$ bedeutet, dass die Zahl i bei der ganzzahligen Division durch 4 den Rest 1 ergeben muss, also der *Restklasse 1* angehört.

Beispiel: im obigen Beispiel kann i folgende Werte annehmen: $1, 5, 9, 13, \ldots$

Beispiel: Die Worte der Sprache in aufsteigender Reihenfolge bezüglich ihrer Länge: ε, *acc*, *accccc*, *abbbcc*, *accacc*, *aaaaacc*, ...

$P = \{$S $\rightarrow$ aA | ε,
A $\rightarrow$ aB | bE | cH,
B $\rightarrow$ aC,
C $\rightarrow$ aD,
D $\rightarrow$ aA,



Weiteres Beispiel regulärer Sprache (schwierig) (*)

- Wir betrachten nun schwierigeres Beispiel: Gesucht sei die reguläre Grammatik zur Sprache

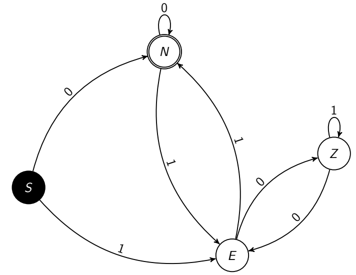
$$L = \{w \in \{0, 1\}^+ \mid w \text{ als Binärzahl aufgefasst ist durch 3 teilbar}\}.$$

- *Problemanalyse:* Wann ist eine Zahl durch 3 teilbar?
  - Genau dann, wenn Ziffernsumme durch 3 teilbar ist.
  - Begründung, anhand des Beispiels 4761:

Stelle durch 3 geteilt	Rest
1er	1
10er	6
100er	7
1000er	4

  - Pro 10er, 100er, ... bleibt ein gewisser Rest. Ist die Summe dieser Reste durch 3 teilbar, dann auch die ursprüngliche Zahl!

Somit erhält man folgenden Automaten:



mit der Grammatik :

$$\begin{aligned} P = \{S &\rightarrow 0N \mid 1E \mid 0, \\ N &\rightarrow 0N \mid 1E \mid 0, \\ E &\rightarrow 0Z \mid 1N \mid 1, \\ Z &\rightarrow 0E \mid 1Z\} \end{aligned}$$

- Nach Start mit 0 $\rightarrow$ 0 Rest (N)
- Nach Start mit 1 $\rightarrow$ 1 Rest (E)
- In Zustand N (Restklasse 0): bei folgendem 1er Wechsel in Zustand E (Restklasse 1).
- Generell gilt: das Anhängen einer Ziffer verdoppelt die Zahl (Basis 2). Somit verdoppelt sich auch der Rest.
- Ziffer 0 in E: Rest wird auf 2 verdoppelt: Übergang nach Zustand Z (Restklasse 2)
- Ziffer 1 in E: Rest wird zunächst auf 2 verdoppelt, und ergibt mit +1 wieder 0 (Übergang nach Zustand N).

- Gesucht sei die Grammatik zur regulären Sprache

$$L = \{w \in \{a,b,c\}^* \mid w \neq \alpha cba\beta \wedge w \neq \alpha bcbc\beta, \alpha, \beta \in T^*\}.$$

- Lösung:

$$\begin{aligned}
P = \{S &\rightarrow aT \mid bB \mid cA \mid a \mid b \mid c \mid \varepsilon, \\
T &\rightarrow aT \mid bB \mid cA \mid a \mid b \mid c, \\
A &\rightarrow aT \mid bC \mid cA \mid a \mid b \mid c, && //G1W1 \\
B &\rightarrow aT \mid bB \mid cD \mid a \mid b \mid c, && //G1W2 \\
C &\rightarrow bB \mid cD \mid b \mid c, && //G2W1 \wedge G1W2 \\
D &\rightarrow aT \mid bE \mid cA \mid a \mid b \mid c, && //G1W1 \wedge G2W2 \\
E &\rightarrow bB \mid b\ \} && //G2W1 \wedge G3W2
\end{aligned}$$

Aufgaben

Aufgabe 3.11

Gesucht ist eine Sprache mit Worten $W \in \{a,b,c,d\}^+$ in der jedoch die Teilworte “dda” und “bdcb” *nicht* vorkommen dürfen. Geben Sie sowohl die reguläre Grammatik als auch den zugehörigen endlichen Automaten an.

Aufgabe 3.12

Gegeben sei die Sprache $L = \{a^i b^j c^k \mid i \equiv 2(5), j \equiv 1(3), k \equiv 3(4)\}^*$. Geben Sie die zugehörige reguläre Grammatik sowie den zugehörigen endlichen Automaten an.

Reguläre Sprachen, Endliche Automaten, und Reguläre Ausdrücke

Programmieren und Software-Engineering
Homomorphismen, Formale Sprachen und Syntax-Analyse

12. März 2024

Reguläre Ausdrücke: Theorie

Regulärer Ausdruck über einer Menge Σ von Zeichen

1. jedes Zeichen $z \in \Sigma$ ist ein regulärer Ausdruck
2. sind r und s reguläre Ausdrücke, gilt
   - *Aneinanderreihung:* $r\ s$ ist ein regulärer Ausdruck
   - *Alternative:* $r \mid s$ ist ein regulärer Ausdruck:
   - *Wiederholung:* r^* ist ein regulärer Ausdruck:
   - *Klammerung:* (r) ist ein regulärer Ausdruck:

Beispiel: $\Sigma = \{a,b\}$

- a, b, aa, ab, ba, abba, abbbaaba
- a|b, abb|ba
- a*, (a|b)*
- a(a|b)*b
- (a|b)*abb
- a*b*
- aa*bb*
- a*(a|b)+
- (aa)*(bb)*b
- (aa|b)*(a|bb)

Praxis: Basic Regular Expressions Standard

c	dass Zeichen c	BRE
$c_1c_2\ldots$	Zeichenfolge $c_1c_2\ldots$	BRE
`.`	ein beliebiges Zeichen	BRE
`\`	*escape character*: das folgende Zeichen ist *kein* Steuerzeichen	BRE
`\.`	Punkt-Zeichen	BRE
[$c_1c_2\ldots$]	eines der Zeichen $c_1, c_2\ldots$	BRE
[c_1-c_2]	eines der Zeichen zwischen c_1 und c_2 (inkl.)	BRE
[^$c_1c_2\ldots$]	jedes andere Zeichen als $c_1, c_2\ldots$	BRE
[^c_1-c_2]	jedes andere Zeichen als eines zwischen c_1 und c_2	BRE
`\w`	Buchstabe, Ziffer, _ : äquivalent `[A-Za-z0-9_]` `\W`: *nicht* …	BRE
`\s`	Whitespace `\S`: *nicht* …	BRE
`^`	Zeilenbeginn	BRE
`$`	Zeilenende	BRE
`\b`	Wortgrenze: Wechsel `\w` ↔ `\W` `\B`: *nicht* …	BRE
`*`	**Wiederholung** beliebig oft inkl. 0-mal *(lazy: `*?` PCRE)*	BRE

Praxis: Extended und Perl Compatible Regular Expressions

`+`	Wiederholung, beliebig oft, mindestens 1-mal *(lazy: `+?` PCRE)*	ERE
{n}	Wiederholung genau n-mal	ERE
{m,n}	Wiederholung m- bis n-mal	ERE
{m,}	mindestens m-mal	ERE
{,n}	höchstens n-mal	ERE
`?`	optional (0-mal oder 1-mal)	ERE
$\vert$	oder (Alternative)	ERE
`\d`	Ziffer: äquivalent `[0-9]` `\D`: *nicht* …	PCRE
(…)	**Gruppierung**[1] und *back references* `\1`, `\2`, …	ERE

[1] hauptsächlich für Ersetzungen

Beispiele Zahlendarstellung

Ganze Zahl: 1, 123, +123, −123, 100, 001, 0, +0, −0

`[-+]?[0-9]+`

ohne führende Nullen, inkl. Null: 1, 123, +123, −123, 100, 0

`[-+]?[1-9][0-9]*|0`

Kommazahl: `3.14`, `+3.14`, `-3.14`, `3.`

`[-+]?[0-9]+\.[0-9]*`

ohne Vorkommateil: `.14159`

`[-+]?([0-9]+\.[0-9]*|\.[0-9]+)`

mit Exponent: `3.14E12`, `3.14e12`, `3.14e-12`, `3.14e+12`

`[-+]?([0-9]+\.[0-9]*|\.[0-9]+)([eE][-+]?[0-9]+)?`

Beispiel: Aufbau einer Textzeile

z. B. `Meier,␣Paul␣F.:␣␣DBI2␣3␣14:00-16:30␣C3.15␣`

1. Zeilenbeginn
2. Familienname (Buchstaben, 1. groß)
3. Beistrich
4. kein oder mehrere Leerzeichen
5. Vorname (Buchstaben, 1. groß)
6. ein oder mehrere Leerzeichen
7. 2. Vorname Abkürzung (Großbuchstabe, Punkt)
8. Doppelpunkt
9. kein oder mehrere Leerzeichen
10. Kürzel (3 Großbuchstaben, Ziffer zwischen 1 und 4)
11. ein Leerzeichen
12. Wochentag als Nummer (1-5)
13. ein Leerzeichen
14. Uhrzeit von (als 2 Ziffern, Doppelpunkt, 2 Ziffern)
15. Bindestrich
16. Uhrzeit bis
17. ein oder mehrere Leerzeichen
18. Raum (Buchstabe A-D, Ziffer 1-5, Punkt, Ziffer, Ziffer 1-9)
19. kein oder mehrere Leerzeichen
20. Zeilenende

Beispiel: Aufbau einer Textzeile (2)

z. B. `Meier,␣Paul␣F.:␣␣DBI2␣3␣14:00-16:30␣C3.15␣`

Zeilenbeginn	`^`
Familienname (Buchstaben, 1. groß)	`[A-Z][a-z]+`
Beistrich	`,`
kein oder mehrere Leerzeichen	`␣*`
Vorname (Buchstaben, 1. groß)	`[A-Z][a-z]+`
ein oder mehrere Leerzeichen	`␣+`
2. Vorname Abkürzung (Großbuchstabe, Punkt)	`[A-Z]\.`
Doppelpunkt	`:`
kein oder mehrere Leerzeichen	`␣*`
Kürzel (3 Großbuchstaben, Ziffer zwischen 1 und 4)	`[A-Z]{3}[1-4]`
ein Leerzeichen	`␣`
Wochentag (1-5)	`[1-5]`
ein Leerzeichen	`␣`
Uhrzeit von (als 2 Ziffern, Doppelpunkt, 2 Ziffern)	`[0-9]{2}:[0-9]{2}`
Bindestrich	`-`
Uhrzeit bis	`[0-9]{2}:[0-9]{2}`
ein oder mehrere Leerzeichen	`␣+`
Raum (Buchstabe A-D, Ziffer 1-5, Punkt, Ziffer, Ziffer 1-9)	`[A-D][1-5]\.[0-9][1-9]`
kein oder mehrere Leerzeichen	`␣*`
Zeilenende	`$`

Beispiel: Aufbau einer Textzeile (3)

z. B. `Meier,␣Paul␣F.:␣␣DBI2␣3␣14:00-16:30␣C3.15␣`

Regulärer Ausdruck für die ganze Zeile

```
^[A-Z][a-z]+,␣*[A-Z][a-z]+␣+[A-Z]\.:
␣*[A-Z]{3}[1-4]␣[1-5]␣[0-9]{2}:[0-9]{2}-[0-9]{2}:[0-9]{2}␣+
[A-D][1-5]\.[0-9][1-9]␣*$
```

Mockaroo: Generierung statt Match, d. h. $* \Rightarrow$ `{0,9}`, $+ \Rightarrow$ `{1,9}`

```
[:upper:][:lower:]{1,9},␣{0,9}[:upper:][:lower:]{1,9}␣{1,9}[:upper:].:
␣{0,9}[:upper:]{3}(1|2|3|4)␣(1|2|3|4|5)␣\d{2}:\d{2}-\d{2}:\d{2}␣{1,9}
(A|B|C|D)(1|2|3|4|5).\d(1|2|3|4|5|6|7|8|9)␣{0,9}
```

Zeichenklassen (BRE)

`[:upper:]`: Großbuchstaben, `[:lower:]`: Kleinbuchstaben, `[:alpha:]`: Buchstaben, `[:digit:]`: Ziffern, `[:alnum:]`: Buchstaben und Ziffern, `[:blank:]`: Leerzeichen oder Tabulator, `[:punct:]`: Sonderzeichen, . . .

Kontextfreie Sprachen

Programmieren und Software-Engineering
Homomorphismen, Formale Sprachen und Syntax-Analyse

12. März 2024

- Gesucht sei die Grammatik zur kontextfreien Sprache

$$\{a^n b^n \mid n > 0\}.$$

- *Beispiel:* $\{ab, aabb, aaabbb, aaaabbbb, \ldots\}$
- Sprache ist nicht regulär, da "Mitzählen" nicht möglich
- Lösung:

$$P = \{S \rightarrow aSb | ab\}$$

- Gesucht sei die Grammatik zur kontextfreien Sprache

$$\{a^{n+1} b^{3n} \mid n \geq 0\}.$$

- *Beispiel:* $\{a, aabbb, aaabbbbbb, \ldots\}$
- Lösung:

$$P = \{S \rightarrow aSbbb | a\}$$

- Gesucht sei die Grammatik zur kontextfreien Sprache

$$\{a^n b^{n+2} c^m d^{m+1} \mid n, m \geq 0\}.$$

- *Beispiel:* $aabbbbcccdddd (n = 2, m = 3)$
- Lösung:

$$\begin{aligned} P = \{S &\rightarrow AB, \\ A &\rightarrow aAb \mid bb, \\ B &\rightarrow cBd \mid d\} \end{aligned}$$

- Gesucht sei die Grammatik zur kontextfreien Sprache

$$\{i^n + i^m = i^{n+m} \mid n, m > 0\}, und$$

$$T = \{i, +, =\}.$$

- *Beispiel:* $iiii + ii = iiiiii$
- Lösung:

$$\begin{aligned} P = \{S &\rightarrow iSi \mid i + Ai, \\ A &\rightarrow iAi \mid i = i\} \end{aligned}$$

- Aufgrund der Ableitungsregeln der Form $S \rightarrow tAB$ mit $t \in T$ und $S, A, B \in N$ ist ersichtlich, dass einfache endliche Automaten für kontextfreie Sprachen nicht funktionieren.
- Es ist unmöglich von *einem* Knoten in einem Schritt über eine Kante zu *zwei* (oder mehreren) Knoten zu gelangen.
- Endliche Automaten können somit nicht zur Überprüfung, ob ein Wort Element der kontextfreien Sprache ist, herangezogen werden.
- Der Automat für kontextfreie Sprachen ist der **Kellerautomat**
  - Kellerautomaten verfügen über einen Last-In First-Out Speicher
  - Die Zustandsübergänge hängen vom gelesenen Zeichen, dem Wert im Speicher (und gegebenenfalls dem Automatenzustand) ab.
- Zur grafischen Veranschaulichung, sowie zur Überprüfung (Ableitung) von Gültigkeit von Worten können **Syntaxdiagramme** verwendet werden.

- Gesucht sei die Grammatik zur kontextfreien Sprache

$$L = \{w \in \{0, 1\}^* \mid n_0(w) = n_1(w)\}.$$

- Dabei gibt die Funktion $n_x(w)$ die Anzahl der Vorkommnisse von x in w an.
- *Beispiel:* 00111101100001 mit $n_0(w) = n_1(w) = 7$.
- Lösung:

$$\begin{aligned} P = \{S &\rightarrow 0E \mid 1N \mid \varepsilon \\ E &\rightarrow 0EE \mid 1S, \\ N &\rightarrow 0S \mid 1NN\} \end{aligned}$$

Ableitung

Ableitung des Wortes 00111101100001:

00111101100001	00111101100001
0E	0011110N
00EE	0011110N
001SE	00111101NN
001E	001111011NNN
0011S	001111011000S
00111N	0011110110000E
001111NN	00111101100001S
0011110SN	00111101100001

- Gesucht sei die Grammatik zur kontextfreien Sprache

$$L = \{a^i b^{2j+3} c^j c^k \mid k > 0, j \geq 0, k \leq i \leq 3k\}.$$

- *Beispiel:* aaaaaabbbccc
- Lösung:

$$\begin{aligned} P = \{S &\rightarrow aSc \mid aaSc \mid aaaSc \mid aAc \mid aaAc \mid aaaAc, \\ A &\rightarrow bbAc \mid bbb\} \end{aligned}$$

- Gesucht sei die Grammatik zur kontextfreien Sprache

$$L = \{w \in \{a, b, c\}^* \mid n_a(w) = n_b(w) + 2n_c(w)\}.$$

- *Beispiel:* aaacabccabaaa
- Lösung:

$$\begin{aligned} P = \{S &\rightarrow \varepsilon \mid aA \mid bB \mid cC, \\ A &\rightarrow aD \mid bS \mid cB, \\ B &\rightarrow aS \mid bC \mid cBC, \\ C &\rightarrow aB \mid bC \mid cCC, \\ D &\rightarrow aAD \mid aDA \mid bA \mid cS\} \end{aligned}$$

- Bemerkungen:
  - A ... ein a zuviel
  - B ... ein b zuviel, bzw. 1 a zuwenig
  - C ... zwei a's zuwenig
  - D ... zwei a's zuviel

Kontextsensitive Grammatik

- Gesucht sei die Grammatik zur Sprache

$$\{a^n b^n c^n \mid n > 0\}.$$

- *Beispiel:* $\{abc, aabbcc, aaabbbccc, aaaabbbbcccc, \ldots\}$
- Die Sprache ist nicht kontextfrei, da die in den Beispielen gezeigten Mechanismen zur Generierung zweier Elemente mit der selben Anzahl nicht auf den Fall von drei Elementen übertragbar ist.
- Die Grammatik ist *kontextsensitiv* (kontextabhängig, umgebungsabhängig)

Kontextsensitive Grammatik

Die Produktionsregeln der kontextsensitiven Gr. zu $\{a^n b^n c^n \mid n > 0\}$ lauten:

$$\begin{aligned} P = \{S &\rightarrow aSAB \mid aAB \\ BA &\rightarrow AB \\ aA &\rightarrow ab \\ bA &\rightarrow bb \\ bB &\rightarrow bC \\ CB &\rightarrow cc\} \end{aligned}$$

Ableitung von aabbcc

$$\begin{aligned} S &\Rightarrow aSAB \\ &\Rightarrow aaABAB \\ &\Rightarrow aabBAB \\ &\Rightarrow aabABB \\ &\Rightarrow aabbBB \\ &\Rightarrow aabbCB \Rightarrow aabbcc \end{aligned}$$

$a^n b^n c^n \mid n > 0$ mittels Typ-0 Grammatik

Produktionsregeln:

$$
\begin{aligned}
P = \{S &\rightarrow abc \\
S &\rightarrow A \\
A &\rightarrow aABc \\
A &\rightarrow abc \\
cB &\rightarrow Bc \\
bB &\rightarrow bb\}
\end{aligned}
$$

Ableitung von *aaabbbccc*:

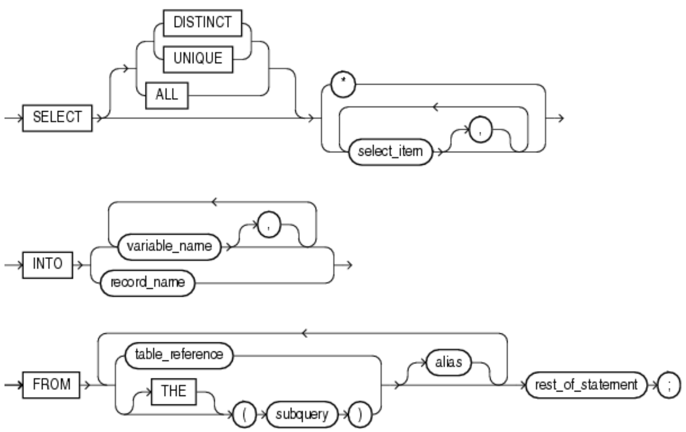
$$
\begin{aligned}
S &\Rightarrow A \\
&\Rightarrow aABc \\
&\Rightarrow aaABcBc \text{ (mittels } A \rightarrow aABc) \\
&\Rightarrow aaabcBcBc \text{ (mittels } A \rightarrow abc) \\
&\Rightarrow aaabBccBc \text{ (mittels } cB \rightarrow Bc) \\
&\Rightarrow aaabBcBcc \text{ (mittels } cB \rightarrow Bc) \\
&\Rightarrow aaabbcBcc \text{ (mittels } bB \rightarrow bb) \\
&\Rightarrow aaabbBccc \text{ (mittels } cB \rightarrow Bc) \\
&\Rightarrow aaabbbccc \text{ (mittels } bB \rightarrow bb)
\end{aligned}
$$

Syntaxdiagramme

- Für eine ausführliche Darstellung sei auf das Skriptum [?] Seiten 24/25 verwiesen!
- Die Terminalsymbole sind durch Kreise oder Ellipsen dargestellt.
- Die Nonterminale sind durch Rechtecke dargestellt.

Syntaxdiagramme: PL/SQL

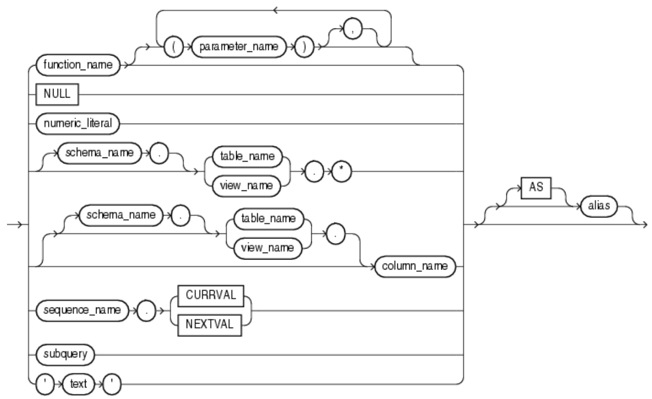
select into statement ::=



Quelle: [?]

Syntaxdiagramme: PL/SQL

select item ::=



Quelle: [?]