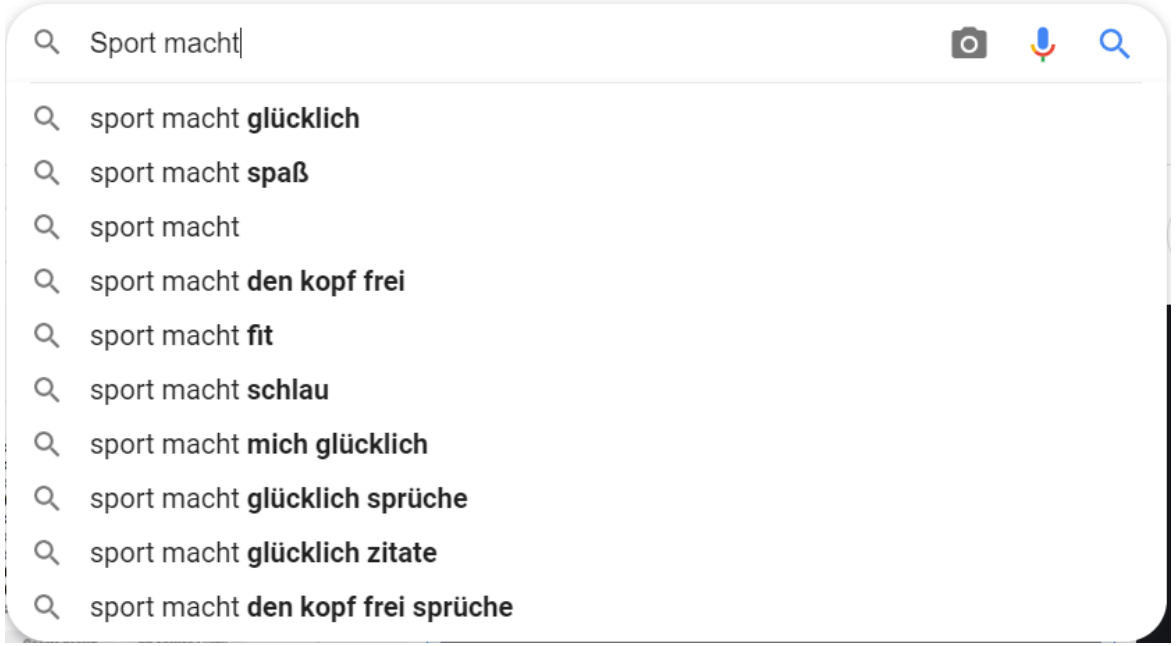
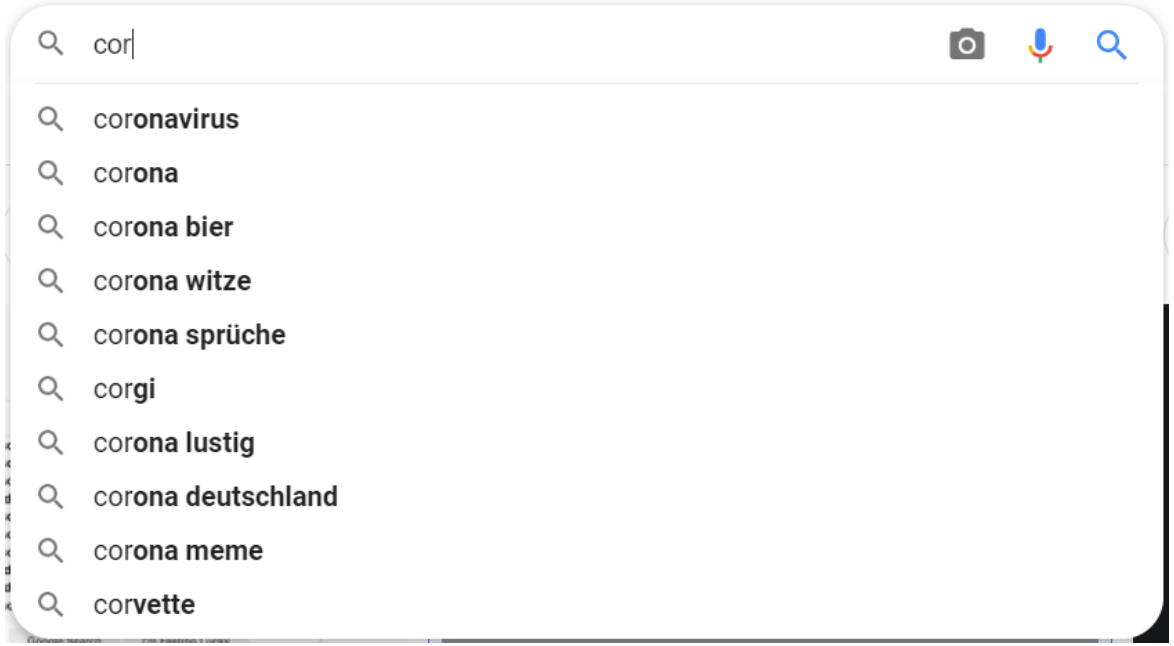
Trie oder Präfixbaum oder Digitaler Suchbaum

Wir funktioniert das Autocomplete-/ Autovervollständigungs-Feature (Präfix-Ergänzung)?





Trie oder Präfixbaum oder Digitaler Suchbaum

Man startet im Editor die Eingabe mit „Fr“ und bekommt die Ergänzung auf „Frau“ und „Freitag“ angeboten,
oder bei „He“ auf „Herr“ und „Heute“….

- Annahme: Applikation hat Zugriff auf das komplette Wörterbuch
- Annahme: alle Worte sind in einem Array gespeichert

Bei einem unsortierten Array wären die Zugriffskosten O(N), und sehr langsam für ein grosses N (N Wörter im Buch).
Eine Hash-Tabelle würde auch nicht helfen, weil sich die Hash-Funktion auf das gesamte Wort bezieht.

Ein sortiertes Array mit den Worten in alphabetischer Ordnung hilft sehr, z.B. durch binäre Suche (O (log N))!

Aber: es geht besser, bis O (1)… auch für IP-Adress-Lookup und Telefonnummern

Trie oder Präfixbaum

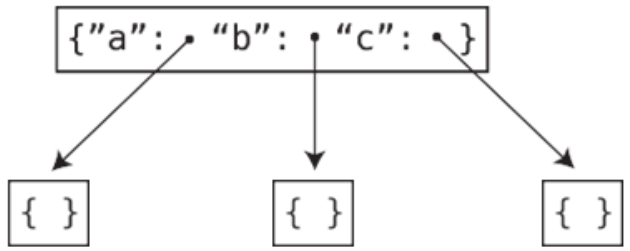
„Trie“ kommt von „re**trie**val “ (Aussprache: nicht wie „tree“, um Verwechslungen zu vermeiden, sondern wie „try“).

Es gibt mehrere Implementierungen (die der gleichen Idee folgen) – hier wurde bewusst die verständlichste gewählt.

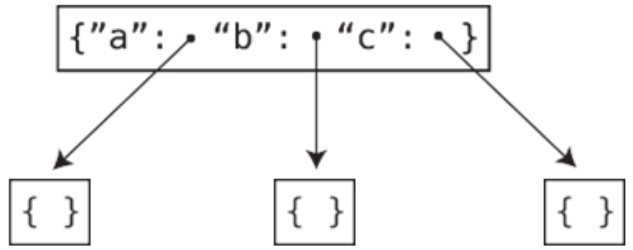
Der Trie-Knoten

Auch der Trie ist eine Menge vernetzter Knoten – allerdings **kein** binärer Baum.
Ein Trie-Knoten kann **beliebig viele Kinder** haben.

Hier: Jeder Trie-Knoten besteht aus einer Hash-Tabelle, die Keys sind Buchstaben des englischen Alphabets, und die Werte sind andere Knoten des Tries:



Trie oder Präfixbaum



Die Kinder sind ebenfalls wieder Hash-Tables, die wiederum auf ihre Kinder zeigen.

Trie-Knoten:

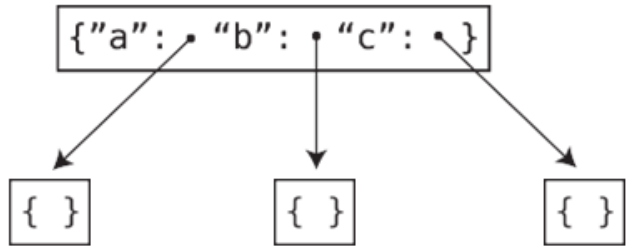
```
void InitNode (trieNode *root)
{
   root->children=NULL;
}
```

Die Hash-Table für den Trie oben könnte so aussehen:

{ *'a'*: &nodeA,
  'b': &nodeB,
  'c': &nodeC }

Keys sind Zeichen, Werte sind Pointer auf andere Hash-Tables.

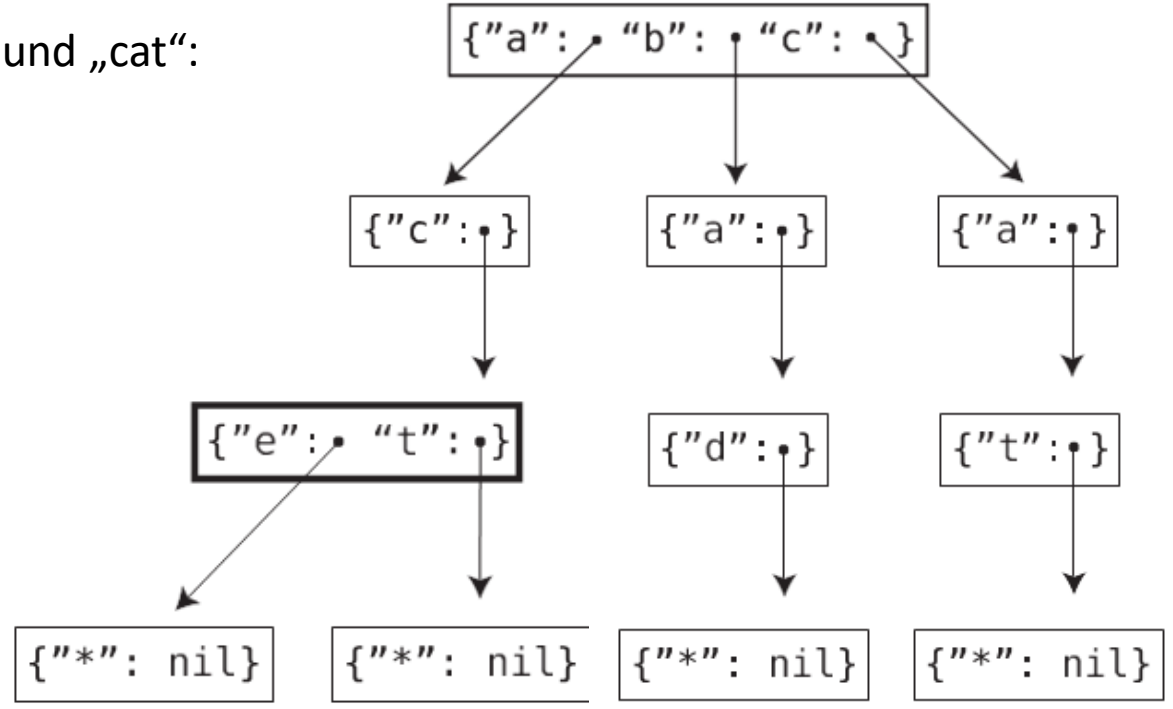
Trie oder Präfixbaum



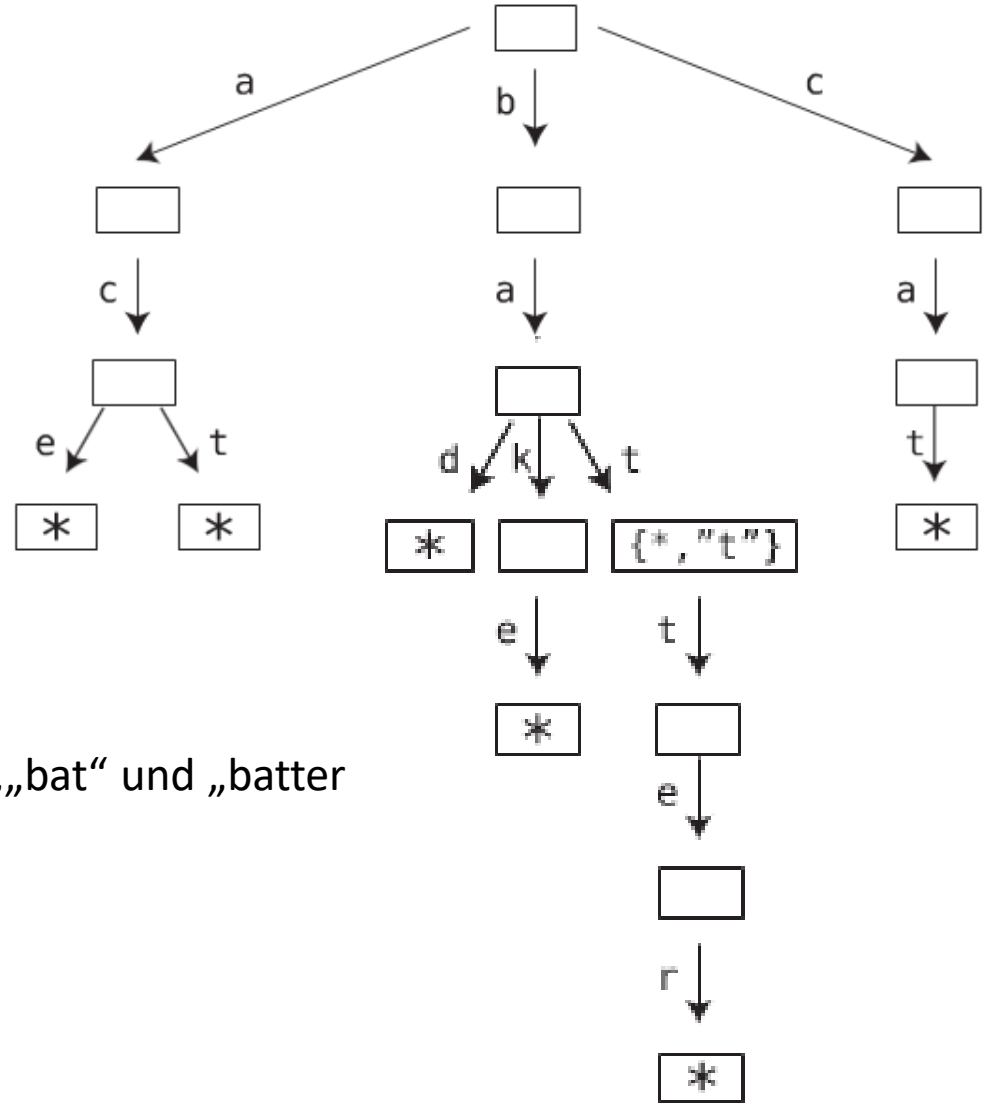
Der Wurzelknoten ist der Anker des Gesamtbaumes, und nicht Bestandteil des Baumes selbst.
Das Zeichen „*“ kennzeichnet das Ende eines Wortes.

Wir speichern die Worte „ace“, „bad“ und „cat“:

Wir fügen das Wort „act“ hinzu:



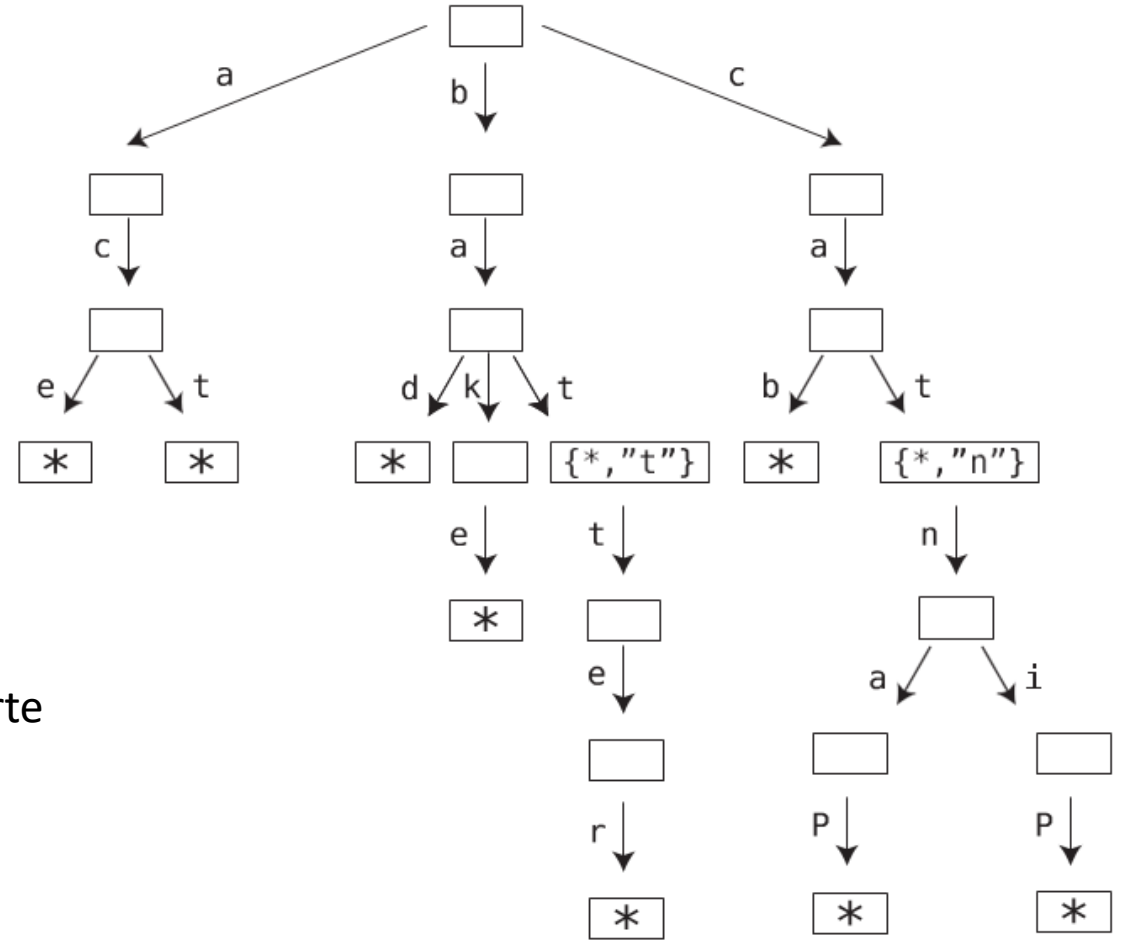
Trie oder Präfixbaum: Einfacheres Diagramm



Wir ergänzen die Worte „bake“,„bat“ und „batter (batter enthält bat):

Trie oder Präfixbaum: Einfacheres Diagramm

Komplexerer Trie: enthält die Worte “ace”, “act”, “bad”, “bake”, “bat”, “batter”, “cab”, “cat”, “catnap”, and “catnip”



In realen Applikationen enthalten Tries viele tausend Worte bis zu kompletten Wörterbüchern.

Trie oder Präfixbaum: Implementierung

Für‘s Verständnis des Autocomplete-Features sehen wir uns die Grundfunktionen eines Tries an:

Suche im Trie (=Suche nach Zeichenketten):

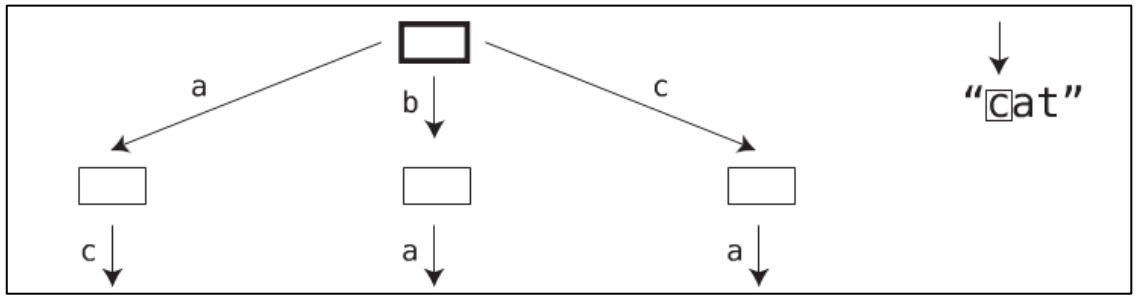
a) Suche, ob der String ein komplettes Wort ist oder
b) ob der String der Präfix (Beginn) eines oder mehrerer Worte ist (Fokus der folgenden Implementierung)

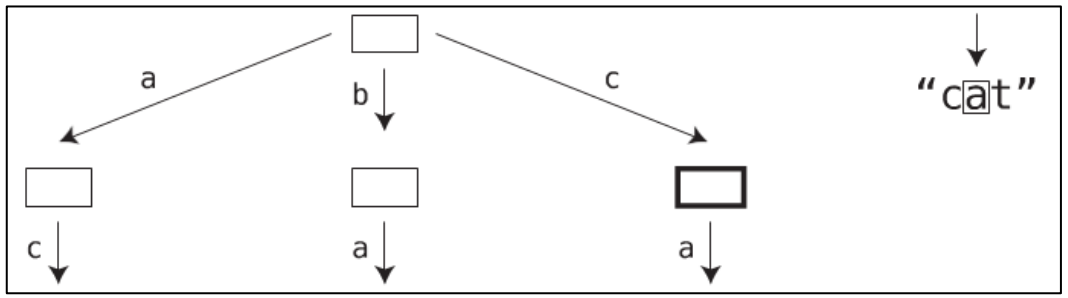
Algorithmus:

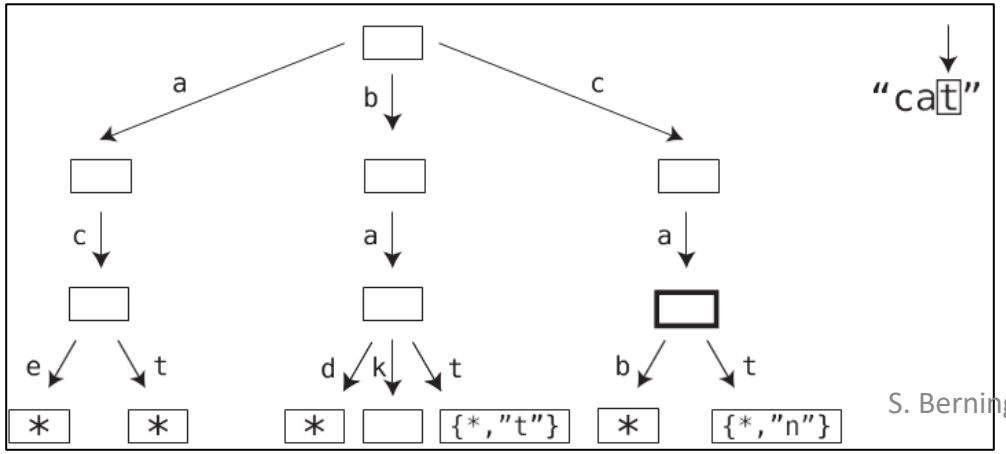
1. Wir erzeugen die Variable currentNode. Sie zeigt zu Beginn des Algorithmus auf den Wurzelknoten.
2. Wir iterieren über jedes Zeichen unseres String.
3. Bei jedem Zeichen prüfen wir, ob der currentNode ein Kind hat mit diesem Zeichen als Key.
4. Wenn nicht, geben wir None zurück, weil dann dieser Trie unseren String nicht enthält.
5. Wenn der currentNode *ein Kind mit dem aktuellen Zeichen als Key enthält*, aktualisieren wir den currentNode auf dieses Kind. Wir gehen dann zurück zu Schritt 2, und iterieren weiter über jedes Zeichen unseres Suchstring.
6. Wenn wir das Ende unseres Suchstrings erreichen, haben wir ihn gefunden.

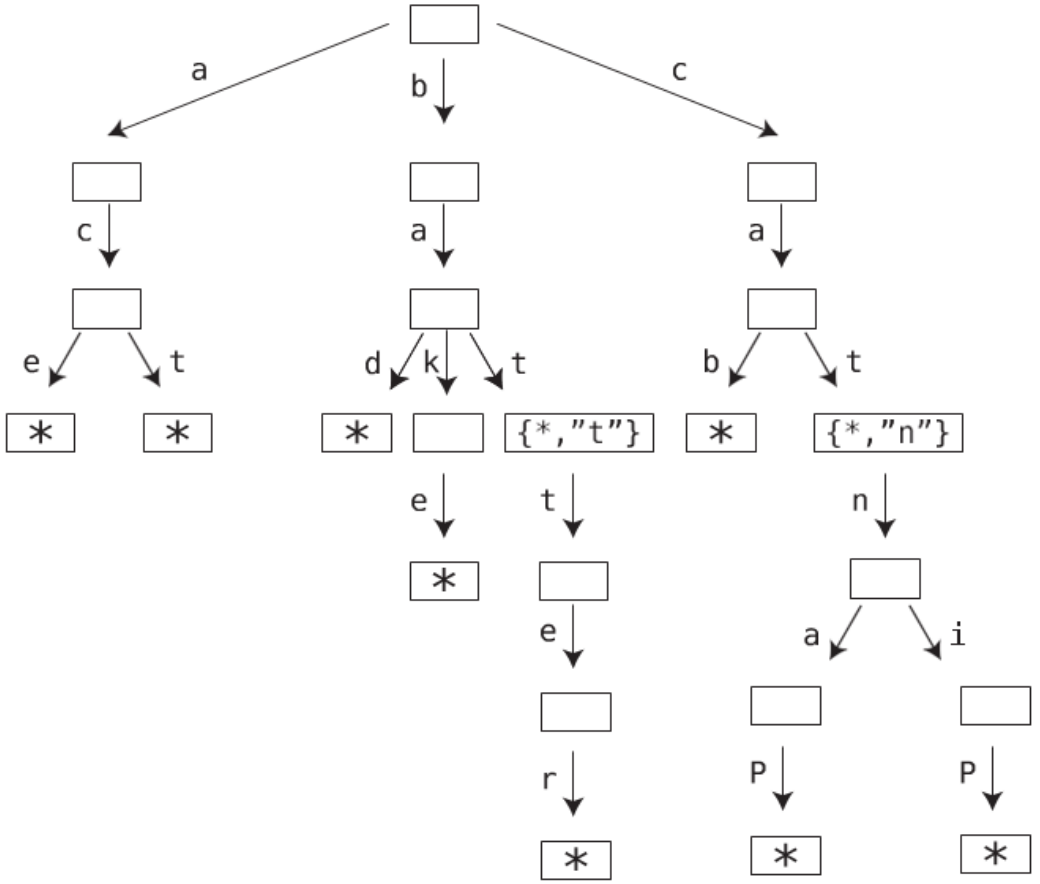
Trie oder Präfixbaum: Implementierung

Suche nach „cat“:









Trie oder Präfixbaum: Effizienzanalyse

Über die Hash-Tabellen finden wir jeden entsprechenden Kindknoten in einem Schritt.
Heisst: unser Algorithmus braucht so viele Schritte, wie der Suchstring Zeichen hat – egal, wieviele Worte das Wörterbuch enthält.

Das kann sehr viel schneller sein als die Binäre Suche in einem Sortierten Array.
Binäre Suche: O(log N), mit N als Anzahl der Worte im Wörterbuch.

Trie-Suche in Big O:

Nicht wirklich O(1): Anzahl der Schritte ist nicht konstant
Aber auch nicht O(N): Anzahl der Schritte hat keinen Bezug zur Anzahl der insgesamt enthaltenen Worte.

Die meisten Quellen nennen diese Effizienz O (K), K die Anzahl der Zeichen im Suchstring (variabel).

O (K) steht für eine konstante Zeit.
Die meisten nicht-konstanten Algorithmen hängen von der Datenmenge in der zu durchsuchenden Struktur ab.

Heisst: wächst N, wächst die Laufzeit des Algorithmus. Das ist hier definitiv nicht der Fall!
Die Laufzeit des Algorithmus ist völlig unabhängig vom Füllstand der Datenstruktur.

Trie oder Präfixbaum: Einfügen

Suche ist die häufigste Operation auf dieser Datenstruktur – nächster Schritt: Untersuchung des Einfügens…

Sehr vergleichbarer Algorithmus zur Suche:

1. Wir erzeugen die Variable currentNode. Sie zeigt zu Beginn des Algorithmus auf den Wurzelknoten.
2. Wir iterieren über jedes Zeichen unseres String.
3. Bei jedem Zeichen prüfen wir, ob der currentNode ein Kind hat mit diesem Zeichen als Key.
4. Wenn ja, aktualisieren wir den currentNode auf dieses Kind. Wir gehen dann zurück zu Schritt 2, und schalten weiter auf das nächste Zeichen unseres Suchstring.
5. Wenn der currentNode kein Kind hat das dem aktuellen Zeichen entspricht, erzeugen wir solch' einen Kindknoten, und aktualisieren den currentNode auf diesen. Wir gehen dann zurück zu Schritt 2, und schalten weiter auf das nächste Zeichen unseres Suchstring.

Trie oder Präfixbaum: Einfügen

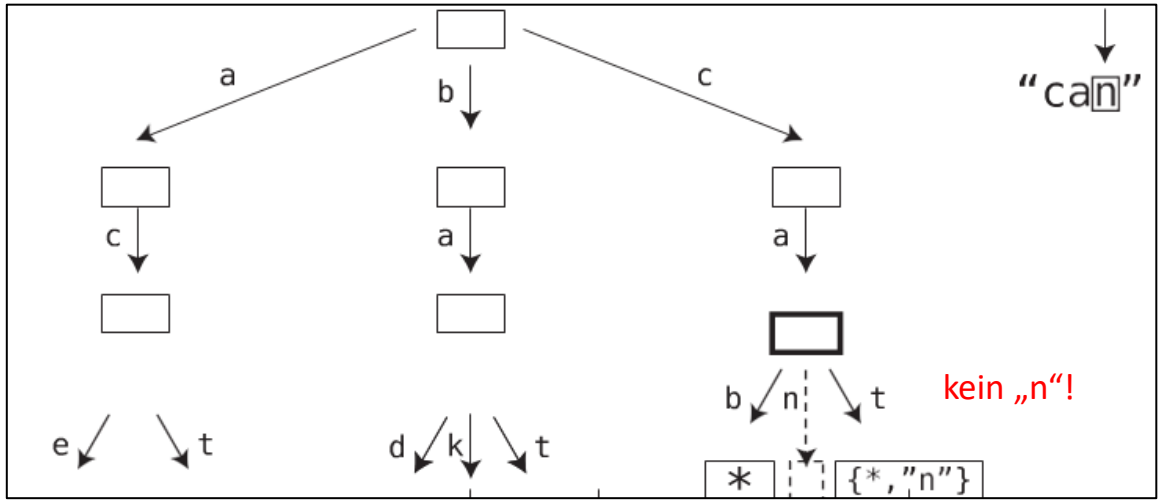
Suche ist die häufigste Operation auf dieser Datenstruktur – nächster Schritt: Untersuchung des Einfügens…

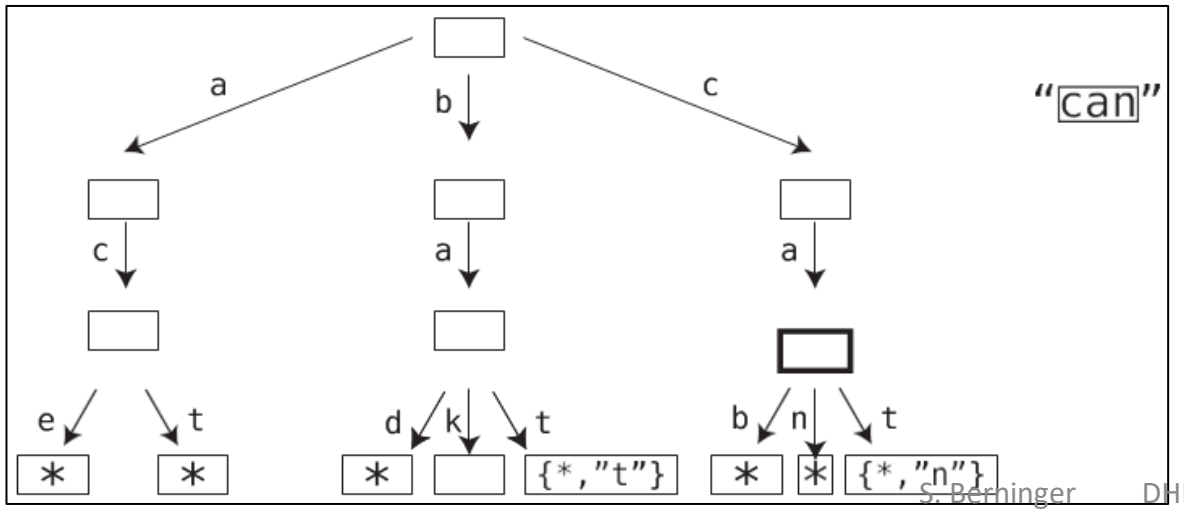
Sehr vergleichbarer Algorithmus zur Suche:

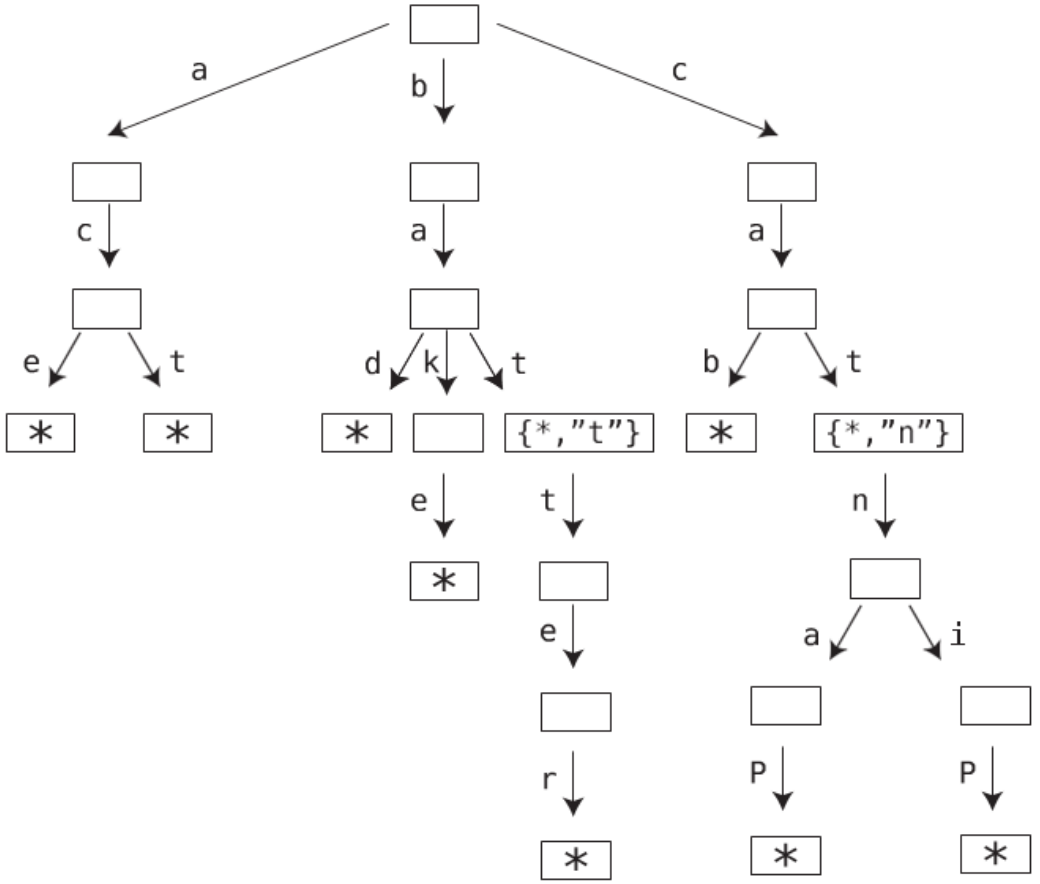
1. Wir erzeugen die Variable currentNode. Sie zeigt zu Beginn des Algorithmus auf den Wurzelknoten.
2. Wir iterieren über jedes Zeichen unseres String.
3. Bei jedem Zeichen prüfen wir, ob der currentNode ein Kind hat mit diesem Zeichen als Key.
4. Wenn ja, aktualisieren wir den currentNode auf dieses Kind. Wir gehen dann zurück zu Schritt 2, und schalten weiter auf das nächste Zeichen unseres Suchstring.
5. Wenn der currentNode kein Kind hat das dem aktuellen Zeichen entspricht, erzeugen wir solch' einen Kindknoten, und aktualisieren den currentNode auf diesen. Wir gehen dann zurück zu Schritt 2, und schalten weiter auf das nächste Zeichen unseres Suchstring.
6. Nachdem wir das letzte Zeichen unseres neuen Wortes hinzugefügt haben, ergänzen wir noch ein "*“-Kind zum letzten Knoten, um zu signalisieren, dass unser Wort abgeschlossen ist.

Trie oder Präfixbaum: Einfügen

Einfügen des Wortes „can“:







Trie oder Präfixbaum: Einfügen Implementierung

Sehr ähnlich der Suche.

Abweichend:

- currentNode hat kein Kind h, das zum aktuellen Zeichen paßt
- dann neues Key-Value-Paar zur Hash-Table des currentNode einfügen,
  Key: aktuelles Zeichen
  Value: Zeiger auf neuen Knoten

Wie die Suche, so dauert auch das Einfügen ungefähr O(K) Schritte (eigentlich O (K+1), wegen dem '*').

Tries mit Popularität: besseres Autocomplete

Autocomplete sollte statt allen nur die populärsten Begriffe zurückliefern!

-> wir müssen die Popularität mit im Trie abspeichern.

„*“ hat bislang das Value = 0

Dieses Value können wir benutzen, um die Popularität abzuspeichern, und sie bei der Auflistung gefundener Worte heranzuziehen.

Zusammenfassung

Wir kennen jetzt 3 Typen von Bäumen: Binäre Suchbäume, Heaps und Tries.

Weitere Typen: B-Bäume, B*-Bäume, AVL-Bäume, Rot-Schwarz-Bäume, 2-3-4-Bäume und viele andere.

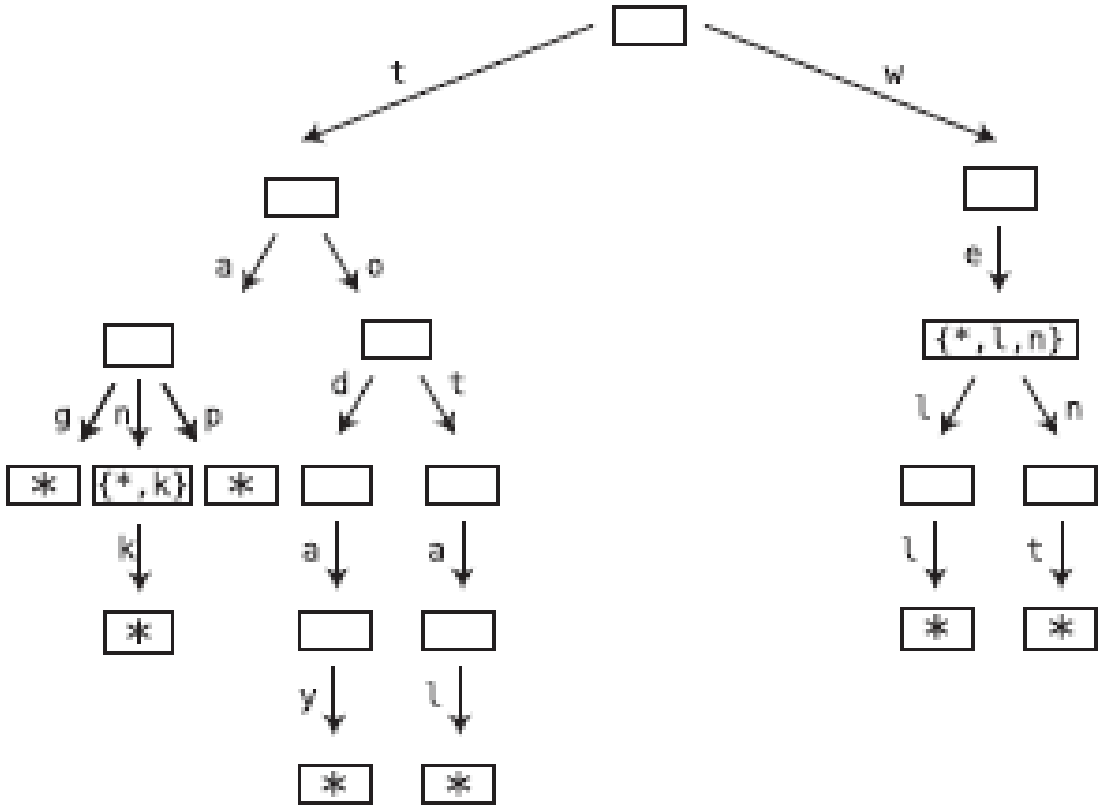
Jeder Baum hat eigene Anwendungsgebiete.

Nächste Lektion: Graphen.

Übung:

1. Listen Sie alle Worte des folgenden Trie auf, und geben Sie die Anzahl an:

A: 5
B: 6
C: 7
D: 9
E: 19



Übung

2. Zeichnen Sie einen Trie, der folgende Worte speichert:
   “get”, “go”, “got”, “gotten”, “hall”, “ham”, “hammer”, “hill”, and “zebra”.

Graphen

Wir bilden ein soziales Netzwerk **gegenseitiger** Freundschaften ab.
Wenn Alice mit Bob befreundet ist, ist Bob auch mit Alice befreundet.

Wie organisieren wir die Daten am Besten?

2-dimensionales Array der Freundschaftslisten?

char * friendships[] = {{ *"Alice"*, *"Bob“}*,
{ *"Bob"*, *"Cynthia"},*
{ *"Alice"*, *"Diana"},*
{ *"Bob"*, *"Diana"},*
{ *"Elise"*, *"Fred"},*
{ *"Diana"*, *"Fred"},*
{ *"Fred"*, *"Alice"} }*

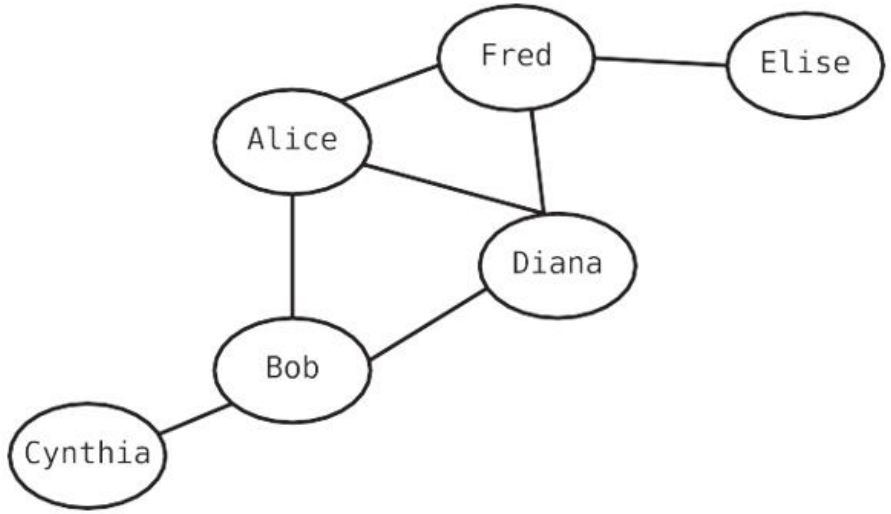
Nicht auf einen Blick alle Freunde von Alice (Bob, Diana und Fred) sichtbar!
Alle Relationen durchsuchen: O (N)....

Graphen

Im Graphen: Suche nach Alices Freunden erreicht O (1)!

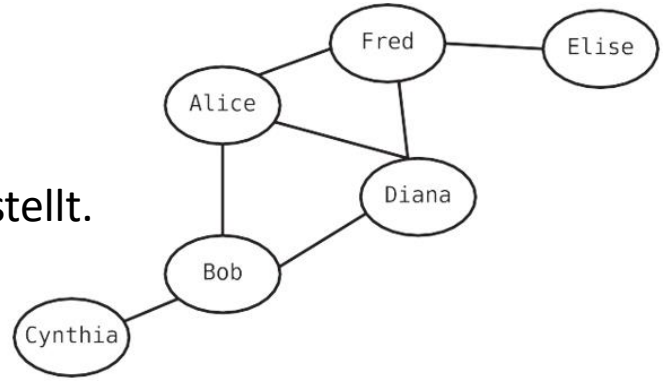
Datenstruktur Graph: spezialisiert auf Beziehungen (zeigt, wie die Daten verbunden sind)

Soziales Netzwerk als Graph:



Jede Person ist ein Knoten, und jede Freundschaft eine Kante zwischen 2 Personen (Alice ist verbunden mit Bob, Diana und Fred).

Graphen

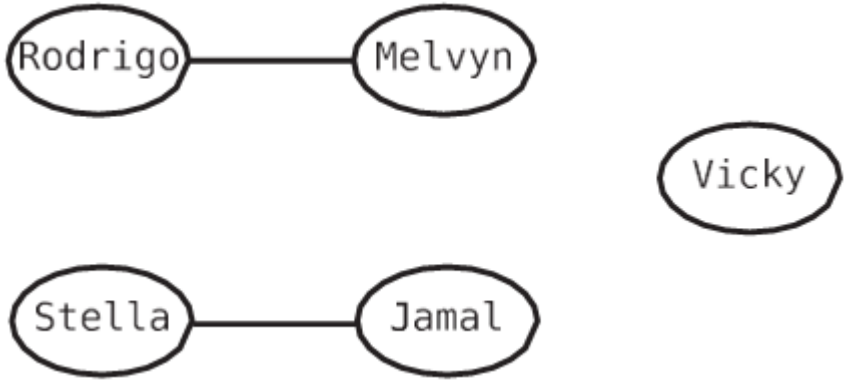


Graphen ähneln Bäumen.
Bäume sind spezielle Graphen – die Baumdaten sind durch verbundene Knoten dargestellt.
Aber: Bäume sind zyklenfreie Graphen!

Bsp.: Alice, Diana und Fred bilden einen Zyklus (in einem Baum nicht möglich).

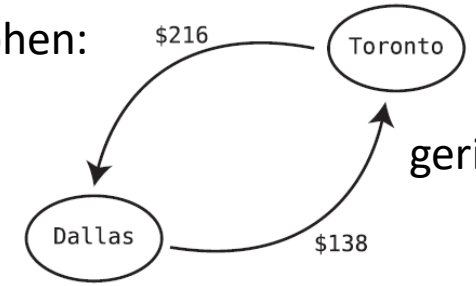
Und: In einem Baum ist auf irgendeinem Weg jeder Knoten mit jedem verbunden (Baum kann vollständig traversiert werden) – das muss ein Graph nicht sein.

Gültiger Graph:

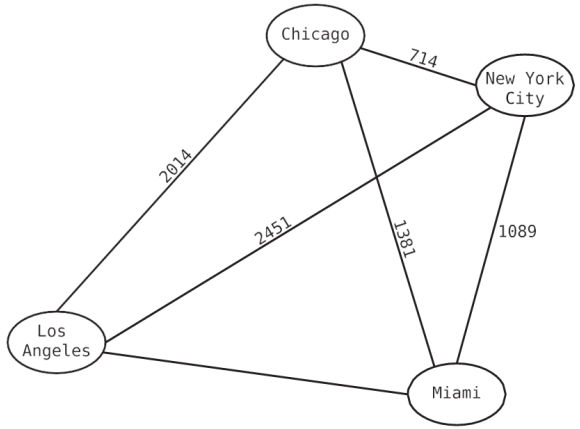


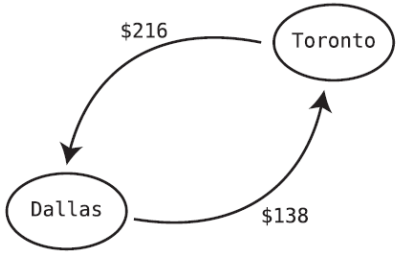
Graphen

Eigenschaften von Graphen:

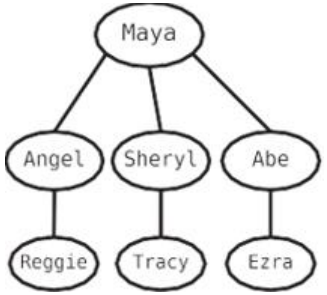


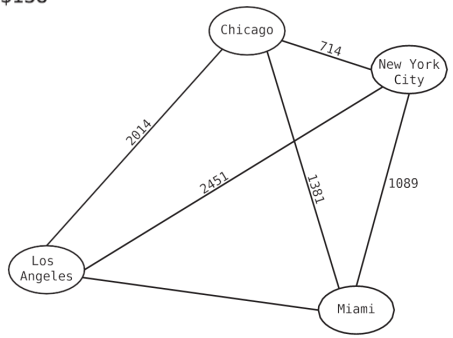
gerichtet (Digraph, directed) – ungerichtet



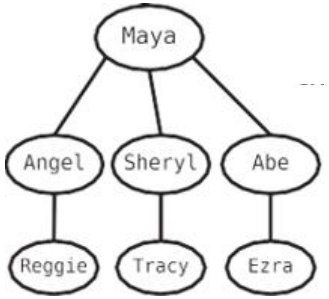


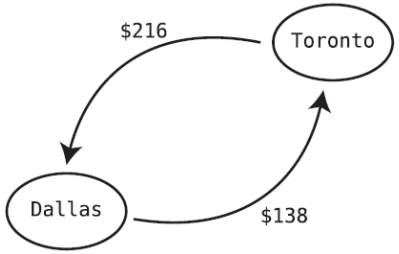
mit Mehrfachkanten (Multigraph) – ohne Mehrfachkanten



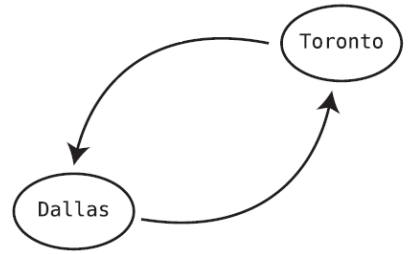


zyklisch – azyklisch





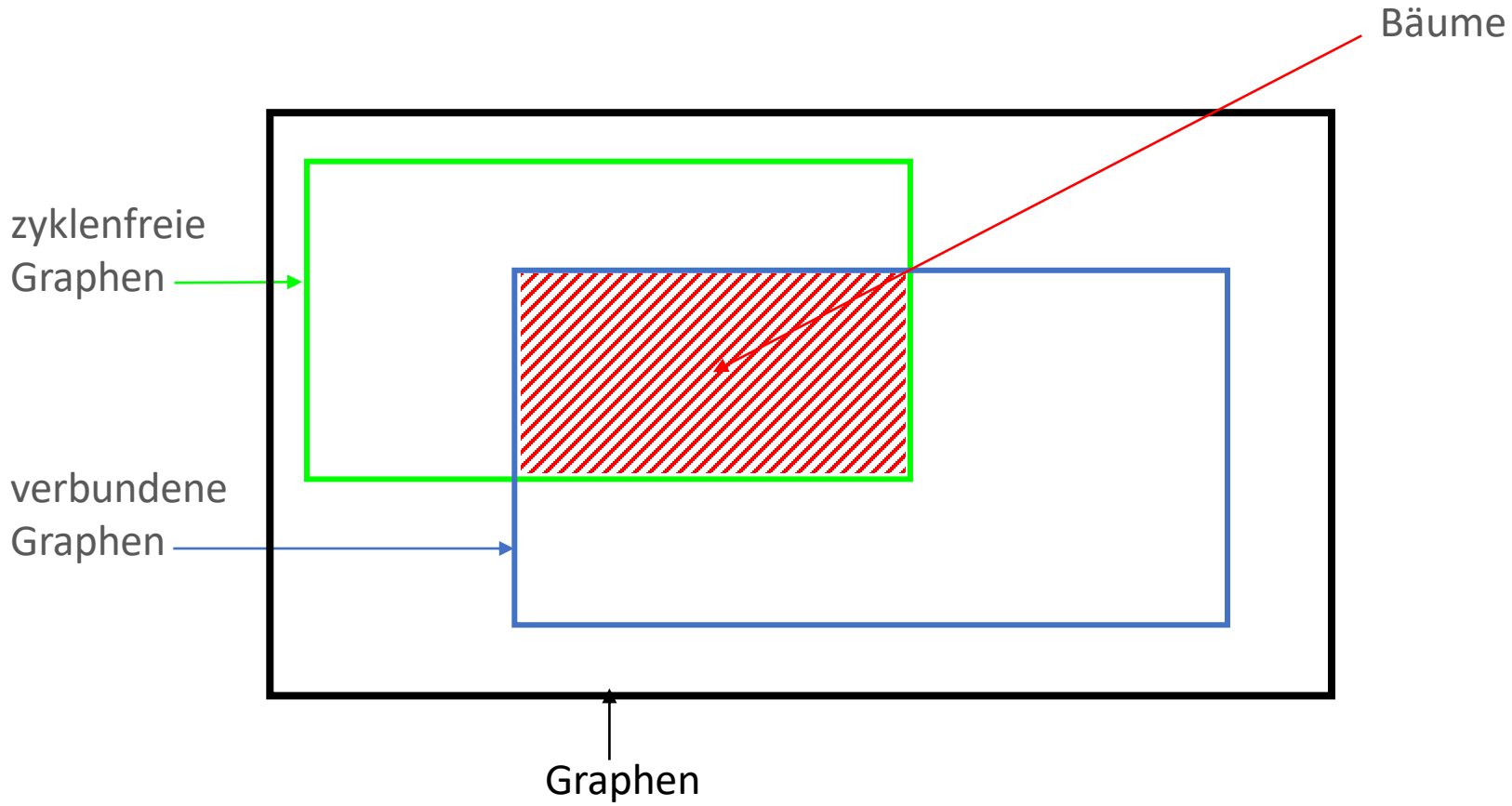
gewichtet - ungewichtet



Graphen

Bäume sind zyklenfreie Graphen.

Bäume haben nur mit der Wurzel (indirekt) verbundene Knoten.



Graphen

Graphen haben einen eigenen technischen Jargon:

Bäume: „Knoten (node)“ --- Graphen: „Knoten (*vertex*)“.

Kante eines Graphen: im Englischen „*edge*“.

Direkt miteinander verbundene Knoten eines Graphen: „*Nachbarn*“ oder „*adjaszent*“.

„Alice“ und „Bob“ sind zueinander adjaszente Nachbarn, weil ihre Knoten (vertices) über eine Kante (edge) verbunden sind.

V (G): Knoten (Vertices) eines Graphen
E (G): Kanten (Edges) eines Graphen

Graphen

Darstellung von Graphen: Klassen oder Hash-Tables (Werte sind z.B. Arrays):

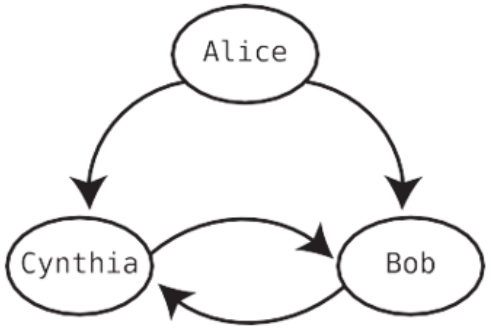
```
friends = {
"Alice"     => ["Bob", "Diana", "Fred"],
"Bob“       => ["Alice", "Cynthia", "Diana"],
"Cynthia" => ["Bob"],
"Diana"    => ["Alice", "Bob", "Fred"],
"Elise"     => ["Fred"],
"Fred"      => ["Alice", "Diana", "Elise"]
}
```

Mit einem Graphen können wir Alices Freunde mit O (1) ermitteln: friends [Alice] liefert das Array aller ihrer Freunde.

Gerichtete Graphen

In manchen sozialen Netzwerken sind nicht alle Freundschaften gegenseitig. Z.B. kann Alice ein „Follower“ von Bob sein, aber Bob muss kein „Follower“ von Alice sein.

Möglicher Follower-Graph:



Diese Graphen werden „gerichtet“ genannt – die Richtung gibt die Beziehung an.

Hash-Table-Implementierung:

```
followees = {
"Alice" => ["Bob", "Cynthia"],
"Bob" => ["Cynthia"],
"Cynthia" => ["Bob"]
}
```

Implementierung von Graphen

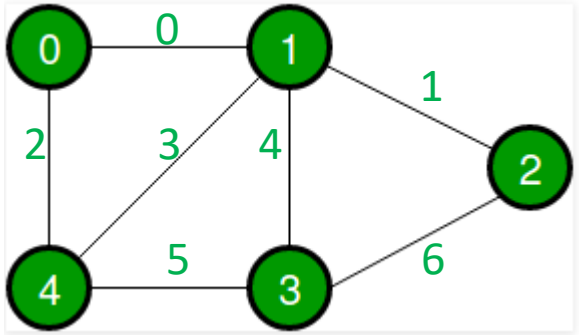
Graphen mit losen Knoten brauchen, um alle Knoten abzuspeichern, eine zusätzliche Datenstruktur, wie z.B. ein Array oder separate Graphen.

Speicherung von Graphen: Adjazenz-Liste vs. Adjazenz-Matrix vs. Kantenliste (Inzidenzmatrix)

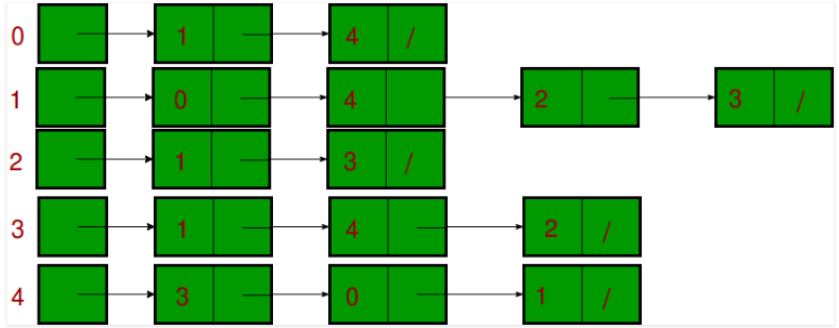
Adjazenzliste (A): einfache Liste, um die Kantenliste jedes Knotens zu speichern (1 Nachfolgerliste pro Knoten).

Adjazenzmatrix (B): 2-dimensionale Arrays statt Listen, Zeilen und Spalten sind Knoten.

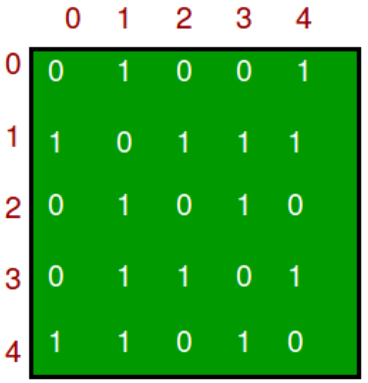
Inzidenzmatrix (C): Zeilen: Knoten, Spalten: Kanten (ist der i-te Knoten Teil der j-ten Kante?)



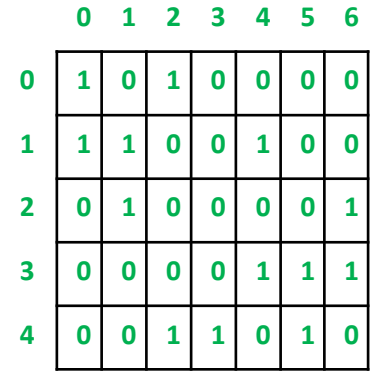
A



B



C

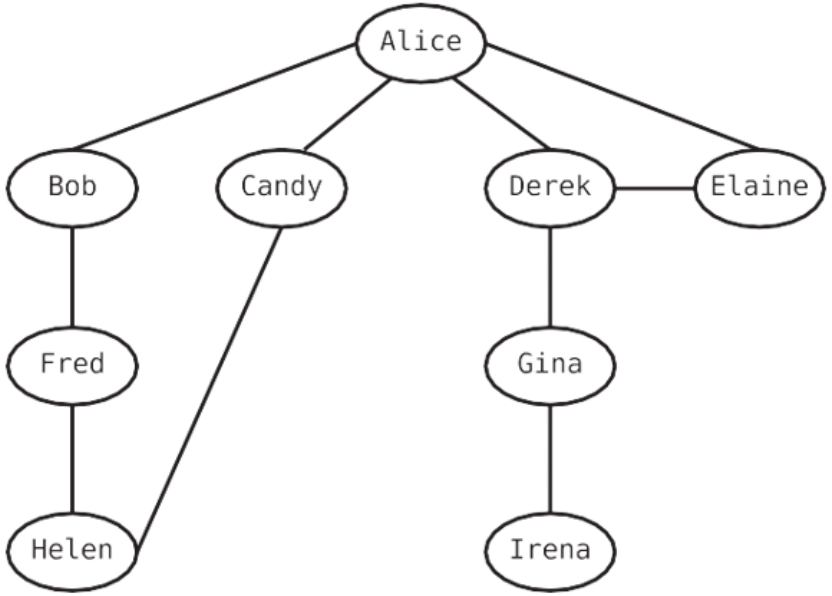


Suche in Graphen

Eine der häufigsten Operationen: Suche nach einem bestimmten Knoten

„Suche“ kann bei Graphen verschiedene Bedeutungen haben:

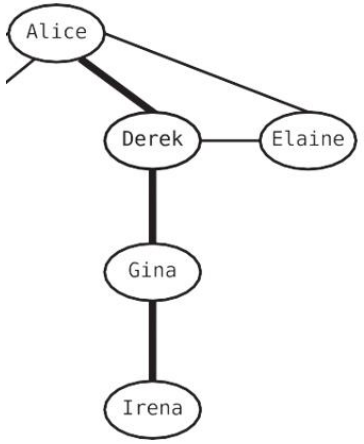
- Suche nach einem bestimmten Knoten („Irena“)
- Suche nach einer bestimmten Kante (Verbindung von Knoten zu Knoten) („von Alice zu Irena“)



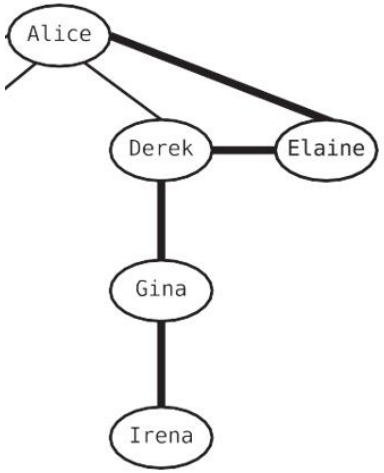
Wenn wir am Knoten von Alice sagen, wir suchen nach Irena, dann suchen wir nach einer Verbindung von Alice zu Irena.

Es existieren 2 verschiedene Wege:

der kürzere:



der alternative, längere:



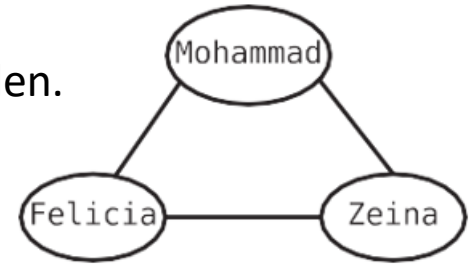
Suche in Graphen: Tiefensuche vs. Breitensuche

Mögliche Use cases für Suchen:

- Sind Alice und Irena im Graph irgendwie verbunden?
- Ist ein bestimmter Knoten im Graph vorhanden?
- Traversiere den Graphen, um alle Knoten zu finden/ zu bearbeiten

a) Tiefensuche (DFS – Depth-First-Search):

Ähnlich der Preorder-Traversierung im binären Baum.
Wird benutzt, um einen bestimmten Knoten, oder durch Traversierung alle Knoten zu finden.



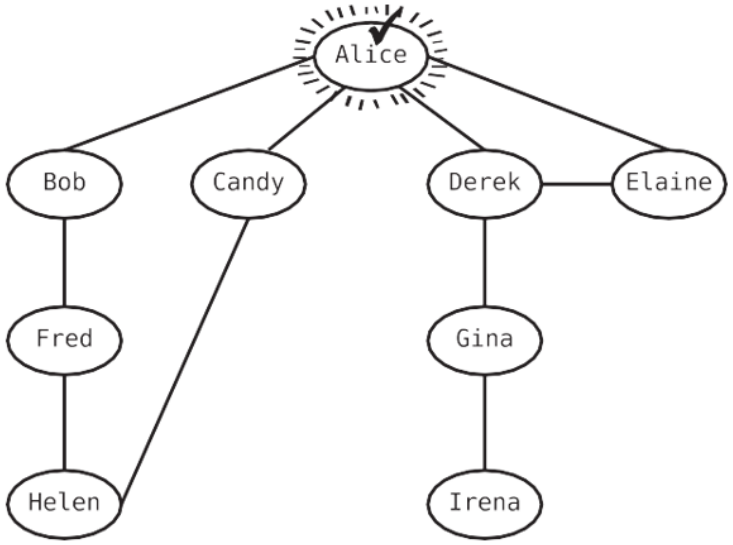
Traversierung:

Wichtig: Markierung der bereits besuchten Knoten zur Vermeidung von Endloszyklen
Möglichkeit: besuchte Knoten in einer Hash-Tabelle speichern (“Markierung”, Value: Boolean true).

Suche in Graphen: Tiefensuche

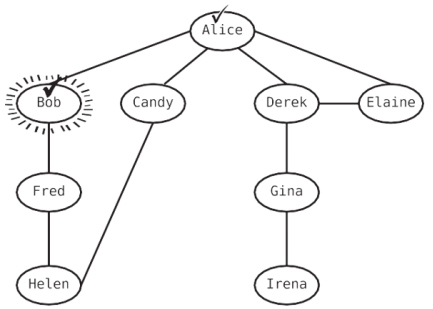
Algorithmus: Startet bei einem beliebigen Knoten des Graphs

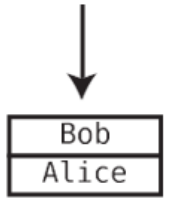
1. Füge diesen Knoten der Hash-Tabelle der besuchten Knoten hinzu (Markierung), und bearbeite ihn.
2. Iteriere über jeden adjazenten Knoten dieses aktuellen Knoten.
3. Ignoriere jeden dieser adjazenten Knoten, wenn er in der Hash-Tabelle bereits enthalten ist.
4. Für jeden adjazenten Knoten, der in der Hash-Tabelle noch nicht enthalten ist: für die Tiefensuche erneut rekursiv durch (voriger Knoten geht auf den Stack, dahin wird zurückgekehrt).



Algorithmus: Wir starten bei Alice

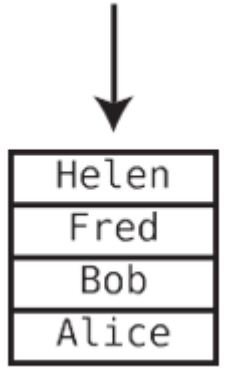
1. Alice -> Hash-Tabelle (Markierung).
2. Iterieren über jeden adjazenten Knoten (Bob, Candy, Derek und Elaine).
3. Ignorieren, wenn bereits in Hash-Tabelle enthalten
4. Wenn nicht:
   Tiefensuche erneut rekursiv (erst: Bob, Alice geht auf den Stack).

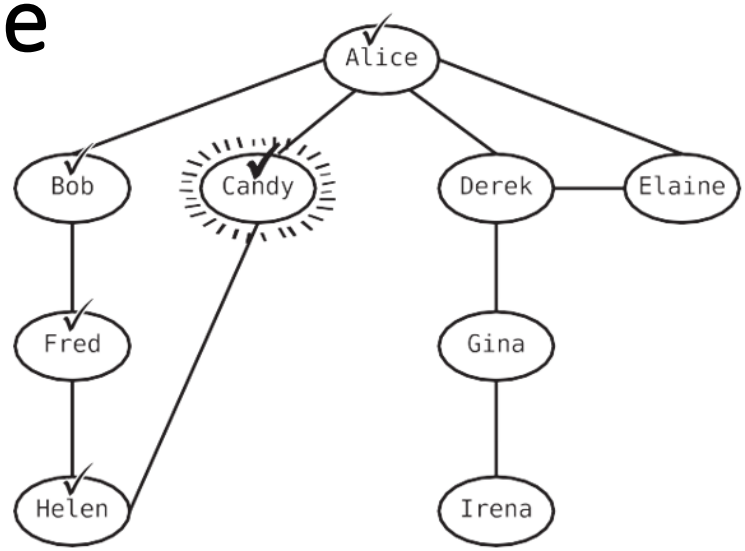




Weiter mit Fred und Helen…

Suche in Graphen: Tiefensuche

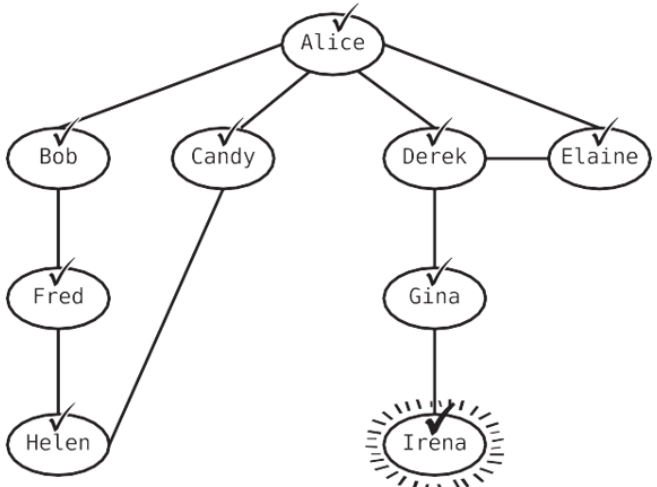




Helen hat 2 Nachbarn: Fred und Candy. Wir sind jetzt an Candys Knoten, und bearbeiten diesen. Sie hat 2 Nachbarn: Alice und Helen – beide haben wir schon besucht.

An dieser Stelle greifen wir auf den Stack zurück. Wir holen über Pop Helen, Fred und Bob heraus, und stellen bei Jedem fest, dass wir alle Ihrer adjazenten Knoten schon besucht haben.
Als wir Alice herausholen, bemerken wir, dass wir noch nicht alle ihrer Adjaszenten besucht haben:
Wir haben Bob und Candy bereits besucht.
Wir haben aber Derek und Elaine noch nicht besucht:

Wir führen die rekursive Tiefensuche weiter für Derek (Elaine und Gina) durch – Alice geht wieder auf den Stack.



In mehreren Rekursionen landen wir bei Irena.
Wir leeren den Stack, bis wir jemanden finden, der noch unbesuchte Nachbar-knoten hat – finden aber niemanden.

Wir sind fertig!

Alice-Bob-Fred-Helen-Candy-Derek-Elaine-Gina-Irena

Suche in Graphen: Breitensuche

b) Breitensuche

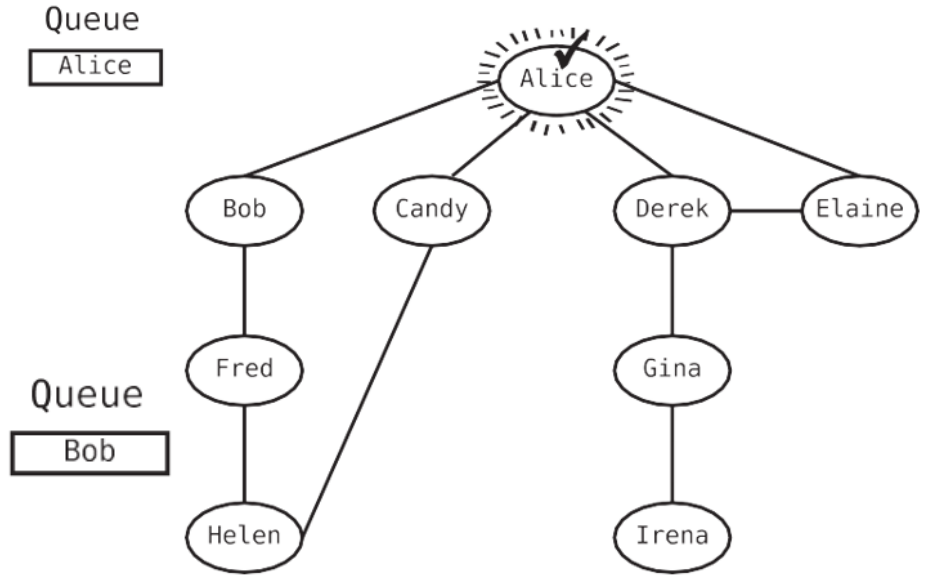
Um die Vorteile der Tiefensuche zu erkennen, beschäftigen wir uns nun zunächst mit der Breitensuche (BFS – Breadth-First-Search):

Sie benötigt keine Rekursion (über den Stack), sondern die Queue als FIFO-Struktur.

Algorithmus für die Breitentraversierung (ähnlich Level-Order-Traversierung):

1. Starte bei irgendeinem Knoten des Graphen – wir nennen ihn “Startknoten”.
2. Füge diesen Knoten der Hash-Tabelle der besuchten Knoten hinzu (Markierung), und bearbeite ihn.
3. Füge diesen Knoten der Queue hinzu.
4. Starte eine Schleife, die über die Queue iteriert, solange diese nicht leer ist. Ist sie leer, sind wir fertig.
5. In dieser Schleife: entnimm der Queue den ersten Knoten. Wir nennen ihn “currentNode”.
6. Iteriere über alle adjazenten (Nachbar-) Knoten des currentNode.
7. Ist ein adjazenter Knoten schon besucht, ignoriere ihn.
8. Wurde ein adjazenter Knoten noch nicht besucht, bearbeite ihn, füge ihn der Hash-Tabelle als besucht und der Queue hinzu
9. Wiederhole die Schleife, bis die Queue leer ist (Punkt 5)..

Breitensuche



Wir starten wieder bei Alice (bearbeiten, ->Queue, -> Hashtabelle):

Wir entnehmen Alice der Queue, und fügen ihr als ersten adjazenten Knoten Bob hinzu. Alice ist immer noch der currentNode!

Wir setzen fort mit Alices adjazenten Knoten, und fügen Candy, Derek und Elaine bearbeitet der Queue hinzu:

Bob	Candy	Derek	**Elaine**

Alle Nachfolger von Alice bearbeitet -> wir entnehmen der Queue das erste Element und machen es zum currentNode (Bob).

Wir bearbeiten und ergänzen seinen Nachbarn Fred.

Für Candy fügen wir ihren bearbeiteten Nachbarn Helen hinzu.

Für Derek Gina (Elaine nicht – existiert schon)…

Candy	Derek	Elaine	**Fred**
Derek	Elaine	Fred	**Helen**
Elaine	Fred	Helen	**Gina**
Fred	Helen	Gina	
Helen	Gina		
Gina			
…			
Irena			

Und erst bei Entnahme von Gina Irena.

→ Alice-Bob-Candy-Derek-Elaine-Fred-Helen-Gina-Irena

Breitensuche

Bei der Breitensuche traversieren wir zunächst alle von Alices nächsten Nachbarn.
Wir bewegen uns dann spiralförmig nach unten und mehr und mehr weg von Alice.

Bei der Tiefensuche bewegen wir uns zunächst so weit wie möglich von Alice weg, bis wir zur Rückkehr gezwungen sind.

Der Unterschied zwischen Breiten- und Tiefensuche ist der Schlüssel zum Verständnis, welcher Algorithmus wann bevorzugt werden sollte.

Breiten- vs. Tiefensuche

Bsp. 1 (für Breitensuche):
Wir möchten die direkten Verbindungen einer Person in einem sozialen Netzwerk finden (alle direkten Freunde von Alice, aber nicht die Freunde ihrer Freunde).

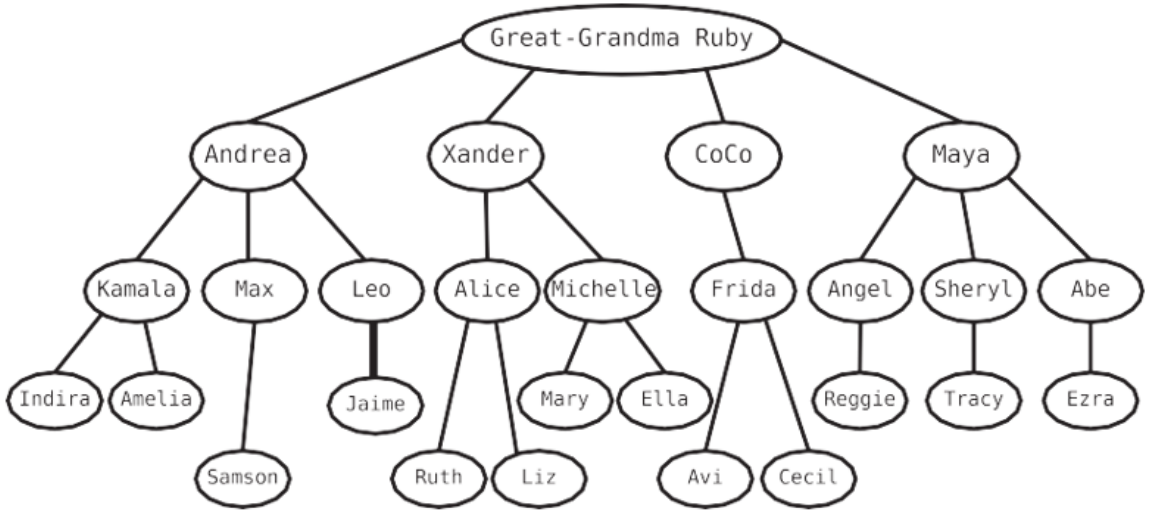
Mit der Breitensuche finden wir die sofort (Bob, Candy, Derek und Elaine), bevor in der nächsten Ebene weitergesucht wird.

Mit der Tiefensuche hingegen hingegen treffen wir auf Fred und Helen (die gar nicht mit Alice befreundet sind), bevor wir ihre anderen Freunde finden. In einem großen Graph würden wir damit unnötig viel Zeit verschwenden.

Breiten- vs. Tiefensuche

Bsp. 2 (für Tiefensuche):
Unser Graph ist ein Stammbaum:

Wir wollen wissen, ob Ruth eines von Rubys Urenkeln ist.
In der Breitensuche müssten wir erst alle Kinder und Enkel durchsuchen, bis wir auf das erste Urenkel treffen.

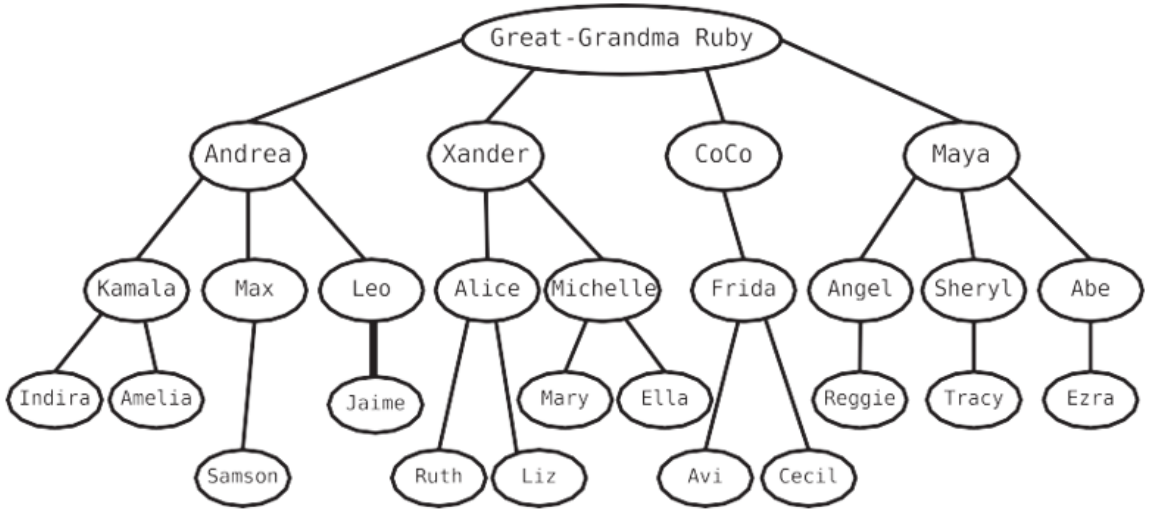


Breiten- vs. Tiefensuche

Mit der Tiefensuche erreichen wir das erste Urgroßenkel in wenigen Schritten.
Obwohl wir evtl. den gesamten Graphen traversieren müssen, um alle Urgroßenkel zu finden, könnten wir Glück haben, und sie früher finden als bei der Breitensuche.

Die Frage ist generell:
Vermuten wir das/ die Ergebnis(se) nah am Startknoten, oder eher weiter entfernt?

Die Breitensuche eignet sich für schnelle Suche in der Nähe, die Tiefensuche für weiter entfernte Ergebnisse.



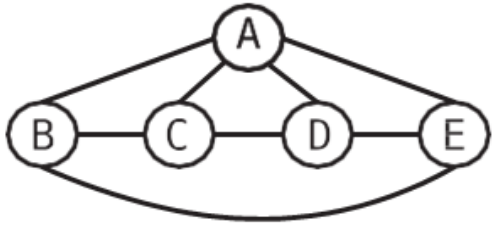
Die Effizienz der Graphensuche

Komplexität: Im worst-case (Knoten existiert nicht oder ist der letzte geprüfte Knoten) traversieren wir bei beiden Suchverfahren alle Knoten des Graphs.

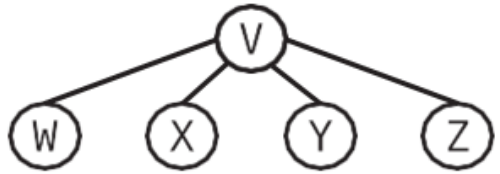
Da wir alle Knoten besuchen müssen: O(N), mit N als Anzahl der Knoten.

Wir iterieren jedoch für jeden besuchten Knoten auch über alle seine adjaszenten Nachbarknoten. Wir ignorieren dabei bereits besuchte Nachbarn, spendieren aber Zeit auf die Prüfung, ob er bereits besucht wurde.

Bsp.: Knoten A hat hier 4 Nachbarn, B, C, D und E je 3 Nachbarn.
Eine Traversierung dieses Graphen würde 5 Schritte brauchen, um jeden Knoten zu besuchen, und 4+3+3+3+3, um die Nachbarknoten jeweils zu prüfen.
In Summe: 21 Schritte.



Bsp.: Anderer Baum mit 5 Knoten – 5 Besuche, aber nur 8 Besuche von Nachbarn.
In Summe: 13 Schritte.



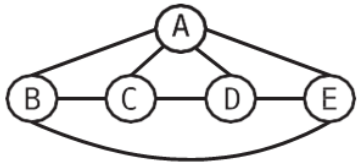
Neben der **Anzahl der Knoten** spielt für die Effizienzberechnung in Graphen also auch die **Anzahl der Verbindungen** eine Rolle!

Die Effizienz der Graphensuche

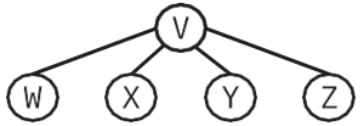
Wir brauchen hier für die Effizienzberechnung also 2 Variablen: die Anzahl der Knoten und die Anzahl der Kanten/ Verbindungen.

Die Big-O-Notation für Graphen bezieht sich also statt auf N auf die Variablen V (Vertices/Knoten) und E (Edges/ Kanten):

O (V+E)



In diesem Beispiel haben wir 5 Knoten und 8 Kanten (13), und benötigen 21 Schritte.



In diesem Beispiel haben wir 5 Knoten und 4 Kanten (9), und benötigen 13 Schritte.

Ursache: Die Kanten werden nur 1x gezählt, die Nachbarknoten aber mehrmals besucht.

Aber: Big-O ignoriert konstante Faktoren! O (V+E) ist nur eine Näherung, aber so gut wir alle anderen Big-O-Schätzungen.

Diese worst-case-Effizienz kann durch **Abstimmung des Suchalgorithmus auf die Entfernung des Suchergebnisses von der Wurzel** in vielen Fällen optimiert werden!

Graph-Datenbanken

Graph-Datenbanken:

Graphen: effizient für das Bearbeiten von Daten mit Beziehungen (wie Freunde in einem sozialen Netzwerk)
-> Nutzung spezieller Graph-Datenbanken für diesen Datentyp

Beispiele von Graph-Datenbanken: Neo4j,[a] ArangoDB,[b] and Apache Giraph.[c]

Diese Webseiten sind ein guter Startpunkt zum Einlesen bei Interesse an der Funktionsweise von Graph-Datenbanken.

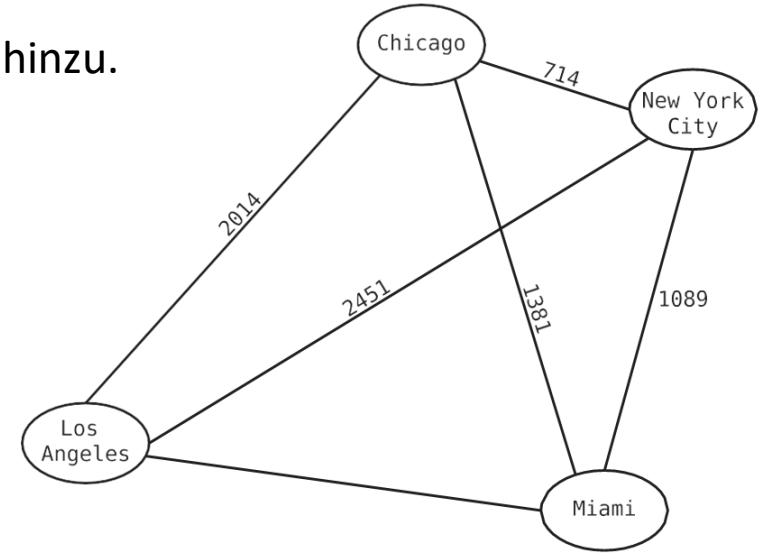
[a] http://neo4j.com
[b] https://www.arangodb.com/
[c] http://giraph.apache.org/

Gewichtete Graphen

Gewichtete Graphen fügen ihren Kanten weitere Informationen („Gewichte“) hinzu.

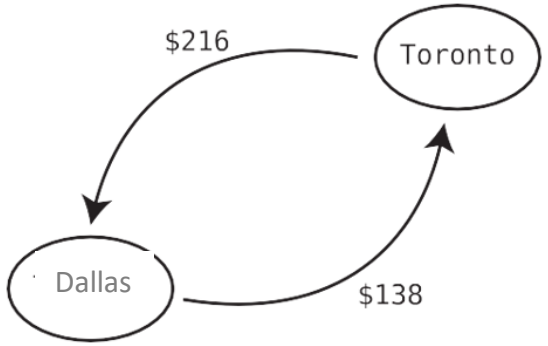
Bsp.: Routenplan zwischen Großstädten der USA:

Diese zusätzliche Kanteninformation enthält die Distanz in Meilen zwischen den benachbarten Knoten (Städten).

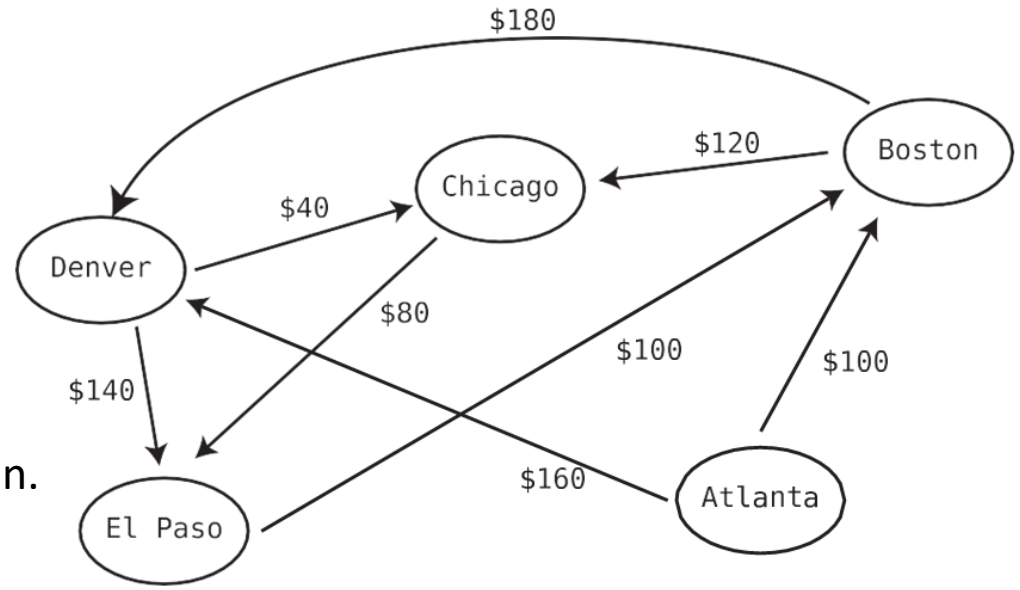


Auch gewichtete Graphen können gerichtet sein.

Bsp.: Obwohl ein Flug von Dallas nach Toronto nur 138$ kostet, kostet die andere Richtung 216$.



Gewichtete Graphen



Bsp.: Kosten verfügbarer Flüge zwischen 5 verschiedenen Städten

Wir sind in Atlanta und wollen nach El Paso fliegen – es gibt aber leider keinen Direktflug.
Stop-over-Flüge sind möglich über Denver, oder sogar über Boston.

Atlanta – Denver - El Paso: 300 $
Atlanta – Denver – Chicago – El Paso: 280$!

„Problem des kürzesten Weges“ („Shortest path problem“):

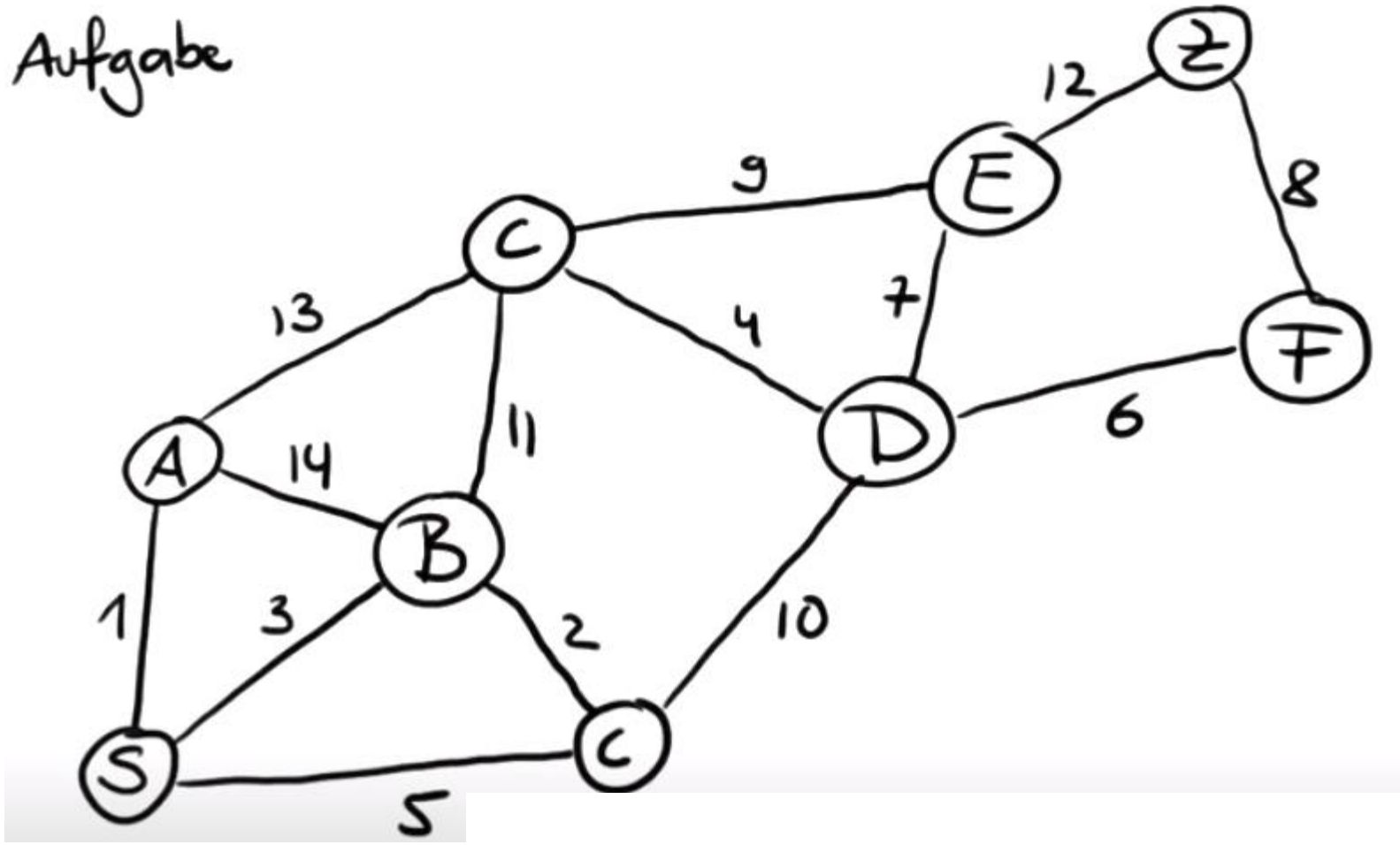
Wie könnte der Algorithmus aussehen, um die billigste Strecke von Atlanta nach El Paso zu finden (unabhängig von der Zahl der Zwischenstops)?

Gewichte können dabei sein:
Wegstrecken, Kosten, Laufzeiten, Bandbreiten etc. pp. (kürzester/ billigster/ schnellster/bester … Weg)

Gewichtete Graphen: Algorithmus von Dijkstra

Es existieren viele Algorithmus für die Lösung dieses „Problems des kürzesten Weges“.
Einer der bekanntesten wurde formuliert von : Edsger Dijkstra, 1959:
https://youtu.be/2poq1Pt32oE

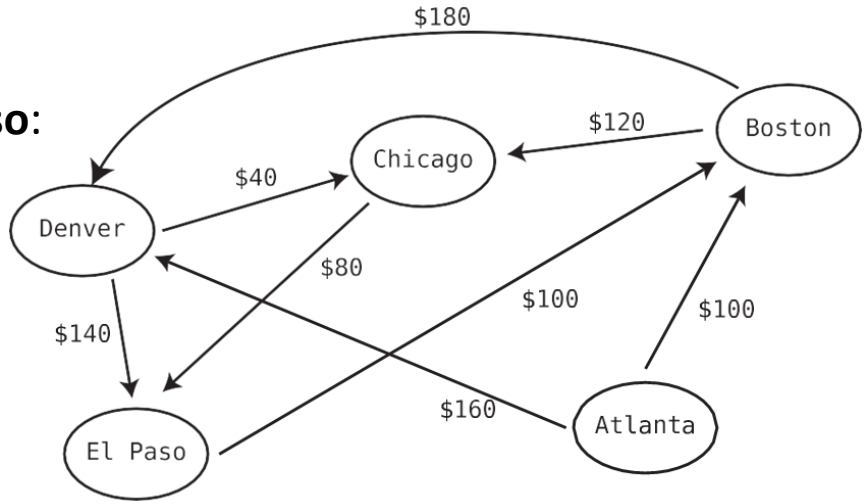
Übung: Wieviel kostet der kürzeste Weg?



A: 29
B: 34
C: 35
D:27

Gewichtete Graphen: Algorithmus von Dijkstra

Regeln dieses Algorithmus, angewandt auf unser Flüge-Beispiel **Atlanta -> El Paso**:



Wir speichern die billigsten Preise vom Startort zu jeder anderen Stadt in einer Hash-Tabelle ab.

// Cheapest-Price-Table

Billigster Preis von Atlanta nach:	Atlanta	Boston	Chicago	Denver	El Paso
	0	?	?	?	?

Wir starten am Knoten Atlanta, und fügen von dort aus weitere Städte in unsere Tabelle hinzu, um den billigsten Preis von Atlanta in jede dieser Städte zu finden (eigentlich suchen wir nach dem billigsten Weg nach El Paso):

Gewichtete Graphen: Algorithmus von Dijkstra

Für die billigste Gesamtstrecke brauchen wir noch eine „Cheapest_previous_stopover_city_table “:

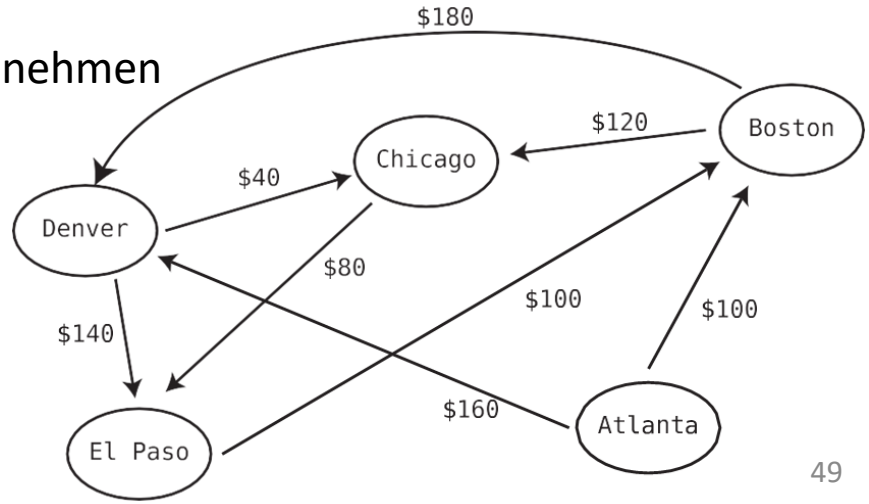
Cheapest Previous Stopover City from Atlanta:	Boston	Chicago	Denver	El Paso
	Atlanta	Denver	Atlanta	Chicago

Cheapest-Price-Table:

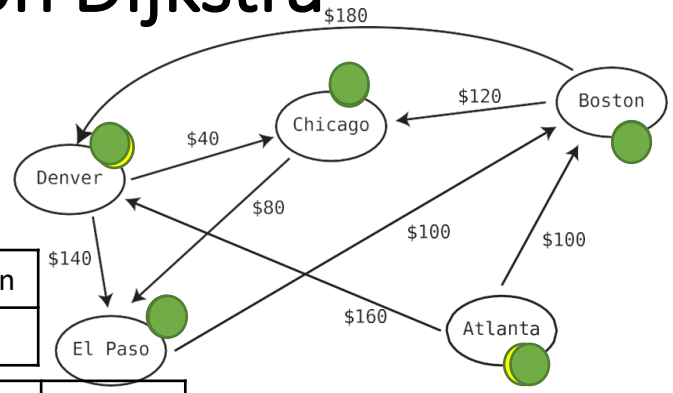
Billigster Preis von Atlanta nach:	Atlanta	Boston	Chicago	Denver	El Paso
	0	?	?	?	?

Algorithmus:

1. Wir besuchen den Startknoten, machen ihn zum “currentNode”.
2. Wir prüfen die Preise vom Startknoten zu jedem benachbarten Knoten.
3. Wenn der Preis zu einem Nachbarknoten billiger ist als der aktuelle Preis in der cheapest_prices_table (oder der Nachbar noch nicht in der Tabelle enthalten ist):
   1. Wir aktualisieren die cheapest_prices_table, um den billigeren Preis aufzunehmen
   2. Wir aktualisieren die cheapest_previous_stopover_city_table, machen den Nachbarknoten zum Key, und den currentNode zum Value.
4. Wir besuchen dann den noch nicht besuchten Knoten mit dem geringsten Preis zum Startknoten, und machen ihn zum “currentNode”.
5. Wir wiederholen Schritte 2-4, bis wir jede Stadt besucht haben.



Gewichtete Graphen: Algorithmus von Dijkstra



„Cheapest_previous_stopover_city_table “:

currentNode:
Atlanta Cheapest Previous Stopover City from Atlanta: *Boston*
Atlanta

Cheapest Previous Stopover City from Atlanta: Boston *Denver*
Atlanta *Atlanta*

currentNode:
Boston Cheapest Previous Stopover City from Atlanta: Boston *Chicago* Denver
Atlanta *Boston* Atlanta

currentNode:
Denver Cheapest Previous Stopover City from Atlanta: Boston Chicago Denver
Atlanta *Denver* Atlanta

Cheapest Previous Stopover City from Atlanta: Boston Chicago Denver *El Paso*
Atlanta Denver Atlanta *Denver*

currentNode:
Chicago Cheapest Previous Stopover City from Atlanta: Boston Chicago Denver El Paso
Atlanta Denver Atlanta *Chicago*

currentNode:
El Paso Cheapest Previous Stopover City from Atlanta: Boston Chicago Denver El Paso
Atlanta Denver Atlanta *Chicago*

Cheapest-Price-Table:

Billigster Preis von Atlanta nach:	Atlanta	Boston
	0	100

Billigster Preis von Atlanta nach:	Atlanta	Boston	Denver
	0	100	160

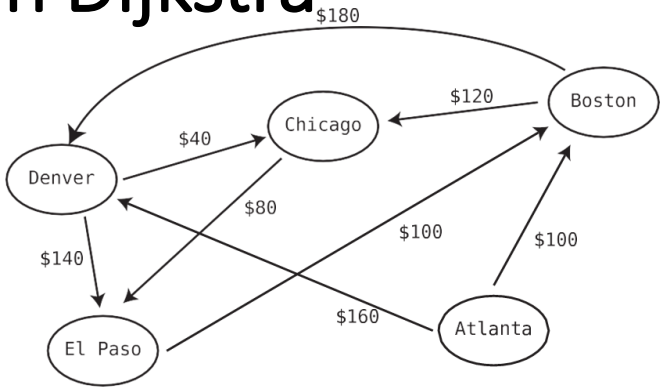
Billigster Preis von Atlanta nach:	Atlanta	Boston	Chicago	Denver
	0	100	*220*	160

Billigster Preis von Atlanta nach:	Atlanta	Boston	Chicago	Denver	El Paso
	0	100	*200*	160	*300*

Billigster Preis von Atlanta nach:	Atlanta	Boston	Chicago	Denver	El Paso
	0	100	200	160	*280*

Billigster Preis von Atlanta nach:	Atlanta	Boston	Chicago	Denver	El Paso
	0	100	200	160	280

Gewichtete Graphen: Algorithmus von Dijkstra



Den kürzesten Weg finden:

Cheapest Previous Stopover City from Atlanta:	Boston	Chicago	Denver	El Paso
	Atlanta	Denver	Atlanta	*Chicago*

From Atlanta to:	Boston	Chicago	Denver	El Paso
	$0	$100	$200	$160	*$280*

Die exakte Route finden:

Benutzung der „Cheapest Previous Stopover City Table“ rückwärts:

El Paso
Chicago _____ Chicago -> El Paso

Chicago
Denver _____ Denver -> Chicago -> El Paso

Denver
Atlanta _____ Atlanta -> Denver -> Chicago -> El Paso

Algorithmus von Dijkstra: Effizienz

Algorithmus zum Finden des kürzesten Weges in gerichteten Graphen.
Die genaue Implementierung ist nicht definiert! (Einfaches Array für noch unbesuchte Knoten, oder Priority Queue…)

Implementierung beeinflusst aber die Komplexität!

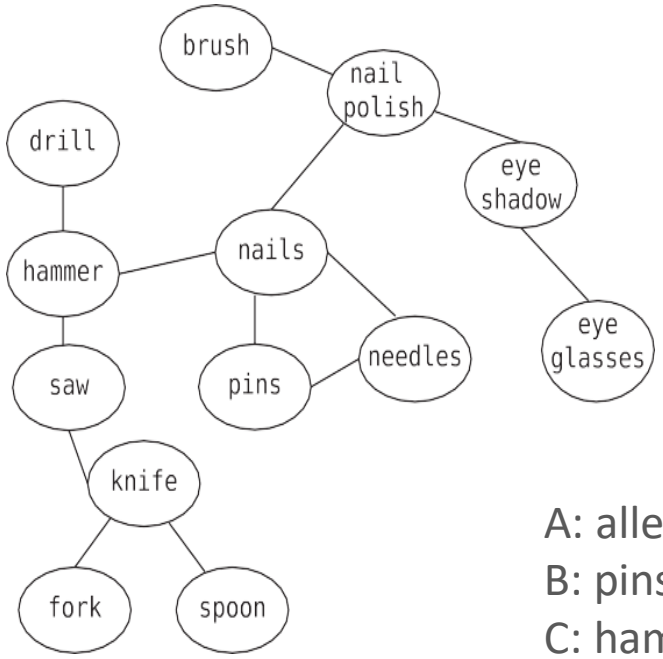
Einfaches Array für noch unbesuchte Knoten: O (V^2) (worst-case-Szenario: jeder Knoten ist mit jedem verbunden)

Priority Queue für noch unbesuchte Knoten: leicht bessere Effizienz

In jeder Implementierung kommt der Dijkstra-Algorithmus aber zum gewünschten Ziel:
den kürzesten/ billigsten… Weg zu finden!

Übung:

1. Welche “ähnlichen” Produkte werden einem Kunden einer auf diesem Graphen basierenden Webseite angeboten, wenn er “Nägel” (nails) ansieht?

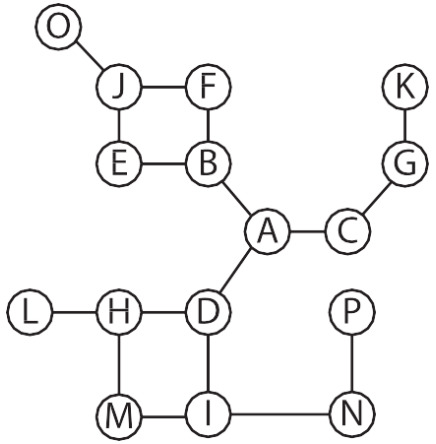


A: alle direkt und indirekt verbundenen Knoten
B: pins, needles
C: hammer, pins, needles, nail polish

Übung:

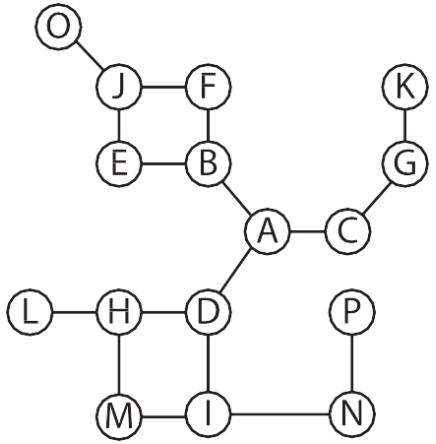
2. Wenn wir auf diesem Graphen beginnend bei “A” eine Tiefensuche durchführen: in welcher Reihenfolge besuchen wir dann die Knoten?

A: A-B-E-J-O-F-C-G-K-D-H-L-M-I-N-P
B: A-B-F-J-O-E-C-G-K-D-H-L-M-I-N-P
C: A-B-C-D-E-F-G-H-I-J-K-L-M-N-O-P



Übung:

3. Wenn wir auf diesem Graphen beginnend bei “A” eine Breitensuche durchführen: in welcher Reihenfolge besuchen wir dann die Knoten?



A: A-B-E-J-O-F-C-G-K-D-H-L-M-I-N-P
B: A-B-F-J-O-E-C-G-K-D-H-L-M-I-N-P
C: A-B-C-D-E-F-G-H-I-J-K-L-M-N-O-P