# **Algorithmen und Datenstrukturen | AlgDat**

Zusammenfassung

[1. Binäre Such-Bäume (Binary search trees, bst) 2](#_Toc156392441)

[1.1. Multimaps 2](#_Toc156392442)

[1.2. Binärer Such-Baum 2](#_Toc156392443)

[2. AVL Bäume (Adelson-Velsky Landis) 4](#_Toc156392444)

[2.1. Höhe eines AVL Baumes 4](#_Toc156392445)

[2.2. Balancierungsfaktor 4](#_Toc156392446)

[2.3. Einfügen in einen AVL Baum, Balancierungsfaktr 5](#_Toc156392447)

[2.4. Löschen 6](#_Toc156392448)

[3. Splay-Trees 7](#_Toc156392449)

[3.1. Suchen in einem Splay-Tree 7](#_Toc156392450)

[3.2. Splay 7](#_Toc156392451)

[4. Vergleichsbasierte Sortieralgorithmen 9](#_Toc156392452)

[4.1. Merge-Sort 9](#_Toc156392453)

[4.2. Quick-Sort 9](#_Toc156392454)

[4.3. Untere Grenze der Vergleichsbasierten Sortierung (Lower Bound) 10](#_Toc156392455)

[5. Nicht-Vergleichsbasierte Sortieralgorithmen 10](#_Toc156392456)

[5.1. Bucket-Sort 10](#_Toc156392457)

[5.2. Lexikographische ordnung (Stable sort) 11](#_Toc156392458)

[5.3. Radix-Sort 11](#_Toc156392459)

[6. Zusammenfassung der Sortier-Algorithmen 12](#_Toc156392460)

[7. Pattern Matching 12](#_Toc156392461)

[7.1. Brute-Force Pattern Matching 13](#_Toc156392462)

[7.2. Boyer-Moore Algorithmus 13](#_Toc156392463)

[7.3. KMP Algorithmus 15](#_Toc156392464)

[8. Tries 16](#_Toc156392465)

[8.1. Standard Tries 17](#_Toc156392466)

[8.2. Komprimierung 17](#_Toc156392467)

[8.3. Suffix Trie 17](#_Toc156392468)

[9. Zusammenfassung der Pattern-Matching-Algorithmen 18](#_Toc156392469)

[10. Dynamische Programmierung 18](#_Toc156392470)

[10.1. Rucksack-Problem 18](#_Toc156392471)

[10.2. Technik der dynamischen Programmierung 19](#_Toc156392472)

[10.3. Längste gemeinsame Subsequenz (Longest Common Subsequence LCS) 19](#_Toc156392473)

[11. Zusammenfassung dynamische Programmierung 21](#_Toc156392474)

[12. Graphen 21](#_Toc156392475)

[12.1. Kanten-Typen 21](#_Toc156392476)

[12.2. Anwendungen 21](#_Toc156392477)

[12.3. Terminologie 21](#_Toc156392478)

[12.4. Eigenschaften 22](#_Toc156392479)

[12.5. Haupt-Methoden des Graph ADT (Abstract Data Type) 23](#_Toc156392480)

[12.6. Kanten-Listen Struktur 23](#_Toc156392481)

[12.7. Adjazenz-Listen Struktur 24](#_Toc156392482)

[12.8. Adjazenz-Matrix Struktur 24](#_Toc156392483)

[12.9. Subgraphen 24](#_Toc156392484)

[12.10. Connectivity 24](#_Toc156392485)

[12.11. Bäume und Wälder 25](#_Toc156392486)

[12.12. Spanning Trees und Wälder 25](#_Toc156392487)

[13. Zusammenfassung PErformance Graphen 25](#_Toc156392488)

[14. Ungerichtete Graphen Traversierung 25](#_Toc156392489)

[14.1. Ungerichtete Tiefensuche / Undirected Depth-First Search 25](#_Toc156392490)

[14.2. Breitensuche / Breadth-First Search 28](#_Toc156392491)

[15. Gerichtete Graphen / Directed Graphs 29](#_Toc156392492)

[15.1. Eigenschaften 29](#_Toc156392493)

[15.3. Gerichtete Tiefensuche / Directed Depth-First Search 30](#_Toc156392494)

[15.4. Erreichbarkeit / Connectivity 30](#_Toc156392495)

[15.5. Transitiver Abschluss / Floyd-warshall 30](#_Toc156392496)

[15.6. Gerichtete Azyklische Graphen (DAG) und Topologische Sortierung 31](#_Toc156392497)

[16. DFS VS. BFS 32](#_Toc156392498)

[16.1. Performance 32](#_Toc156392499)

[16.2. Applikation 32](#_Toc156392500)

[16.3. Back Edge vs Cross Edge 33](#_Toc156392501)

[17. Shortest Path Tree 33](#_Toc156392502)

[17.1. Eigenschaften 33](#_Toc156392503)

[17.2. Dijkstra’s Algorithmus 33](#_Toc156392504)

[17.3. Bellman-Ford 34](#_Toc156392505)

[17.4. DAG-basierter Algorithmus (Directed acyclic graph) 35](#_Toc156392506)

[18. Minimum Spanning Trees (MST) 35](#_Toc156392507)

[18.1. Eigenschaften 35](#_Toc156392508)

[18.2. Kruskal’s ALgorithmus 35](#_Toc156392509)

[18.3. Prim-Jarnik’s Algorithmus 37](#_Toc156392510)

[18.4. Boruvka’s Algorithmus 37](#_Toc156392511)

[19. Traversierungsarten 38](#_Toc156392512)

[20. O-Notationsliste 38](#_Toc156392513)

## Binäre Such-Bäume (Binary search trees, bst)

### Multimaps

Map, in der mehrere gleiche Keys möglich sind. Ungeordnet, besitzen folgende Methoden:

* find(k): liefert erste Entry zum Schlüssel k oder *null*
* findAll(k): liefert eine iterierbare Collection mit allen Entries zum Schlüssel k
* insert(k,o): neue Entry zum Schlüssel k und wert o
* remove(e): entfernt die Entry e (und Rückgabe von e)
  + 1. Geordnete Multimaps

Die Keys folgen einer vollständigen Ordnungsrelation, das heisst, dass die Keys der Grösse nach sortiert sind (). Neue Operationen:

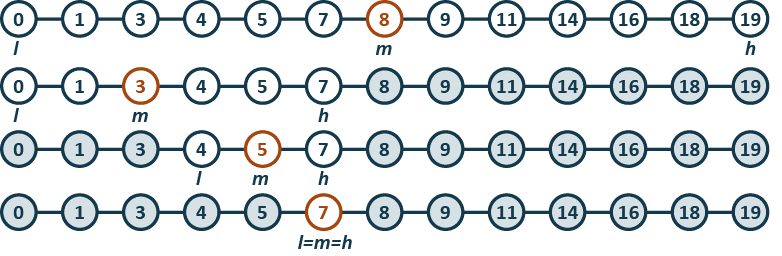
* first(): liefert erste Entry in der Multimap-Ordnung (Erster Entry des ersten Keys)
* last(): liefert letzte Entry in der Multimap-Ordnung (Erster Entry des letzten Keys)
* successors(k): liefert Iterator über Entries mit Schlüssel grösser oder gleich k; nicht abnehmende Ordnung (werden in der Reihenfolge geliefert wie sie in den Keys sind und nicht weiter sortiert)
* predecessors(k): liefert Iterator über Entries mit Schlüssel kleiner oder gleich k; nicht zunehmende Ordnung (nicht sortiert)
  + 1. Suchtabelle

Multimap, welche mithilfe einer sortierten Sequenz implementiert wird. Die Entries werden in einer Array-basierten Sequenz abgespeichert, sortiert nach Schlüssel. Sind nur effektiv, wenn die Multimap klein ist und vor allem Such-Operationen ausgeführt werden.

Binäre Suche (find(k))

Bei jedem Schritt wird die Anzahl der Kandidaten halbiert, terminiert nach Schritten. Bsp: find(7):

Gestartet wird in der Mitte des Arrays (). Da werden nur die Entries auf der linken Seite von 8 angeschaut, das Ende des Bereichs () wird also auf gesetzt. Nun wird die neue Mitte () bestimmt und wieder mit verglichen. Wiederhole die Schritte, bis .



### Binärer Such-Baum

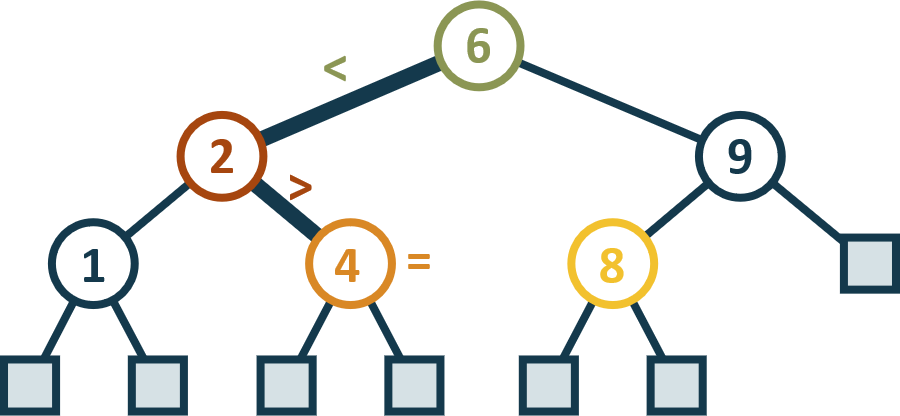
Binärer Baum, welcher Key in seinen internen Knoten speichert. Knoten () haben maximal 2 Kinder, wobei einer davon im linken Teilbaum () und der andere im rechten Teilbaum () ist.   
Bei den drei Knoten und gilt:

In den Blättern (External Node) sind keine Daten gespeichert.

* + 1. **Suche** nach Key

Start bei Knoten v (zu Beginn der Root-Knoten). Ex: **TreeSearch(4, root)**

Algorithm TreeSearch(k,v) //k = value to find, v = node to compare

 if T.isExternal(v) //Node is Leaf, key not

return v //found, return v (key(8))

if k < key(v) //Ex: key(6)

return **TreeSearch**(k, T.left(v))

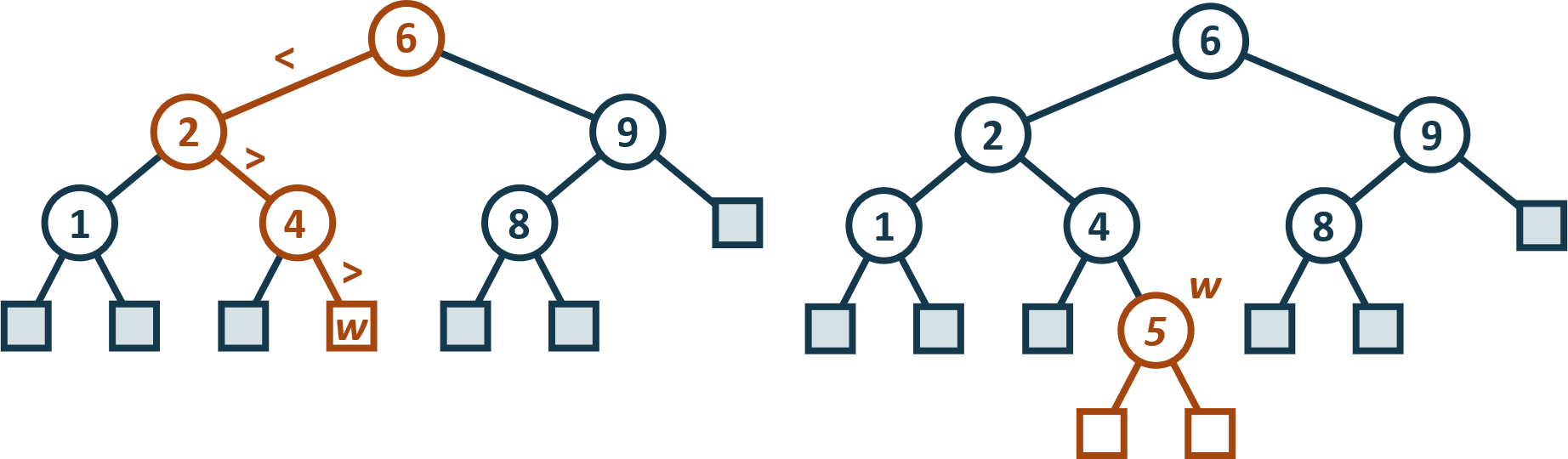
else if k > key(v) //Ex: key(2)

return **TreeSearch**(k, T.right(v))

else if k = key(v) //Ex: key(4)

return v

* + 1. Einfügen des Key

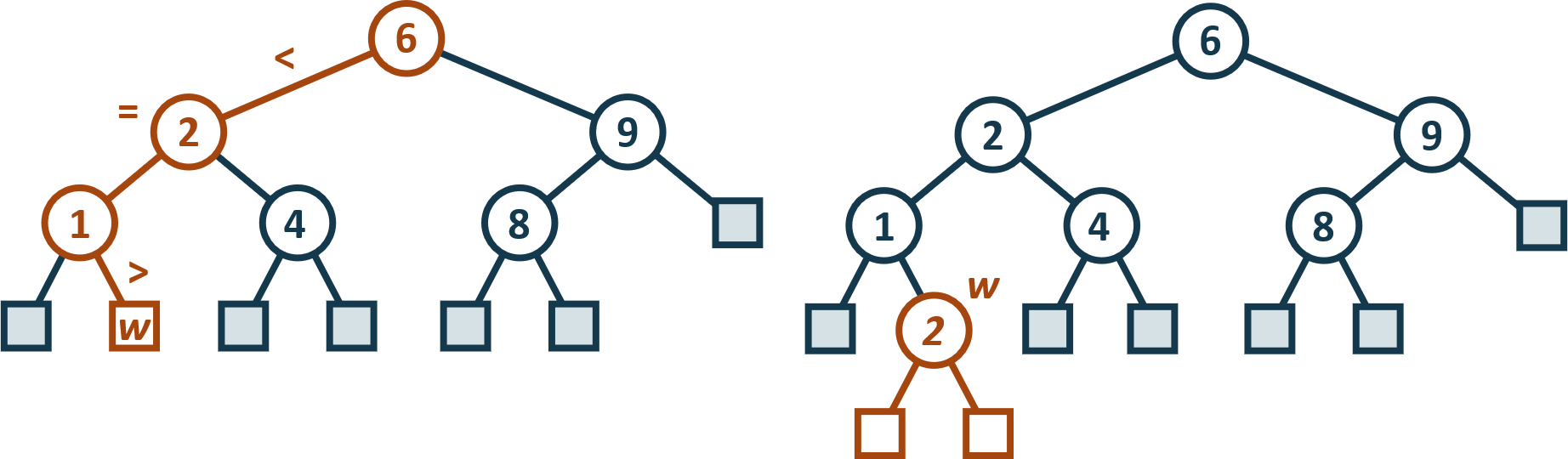
**einfügen**, falls noch nicht vorhanden

Zuerst nach Blatt suchen, bei einfügen und in einen internen Knoten expandieren. Ex: **insert(5)**

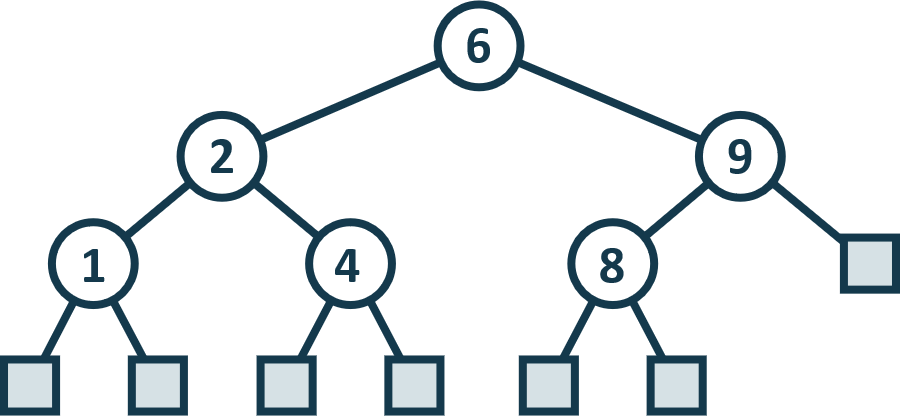
**einfügen**, falls schon vorhanden (map)

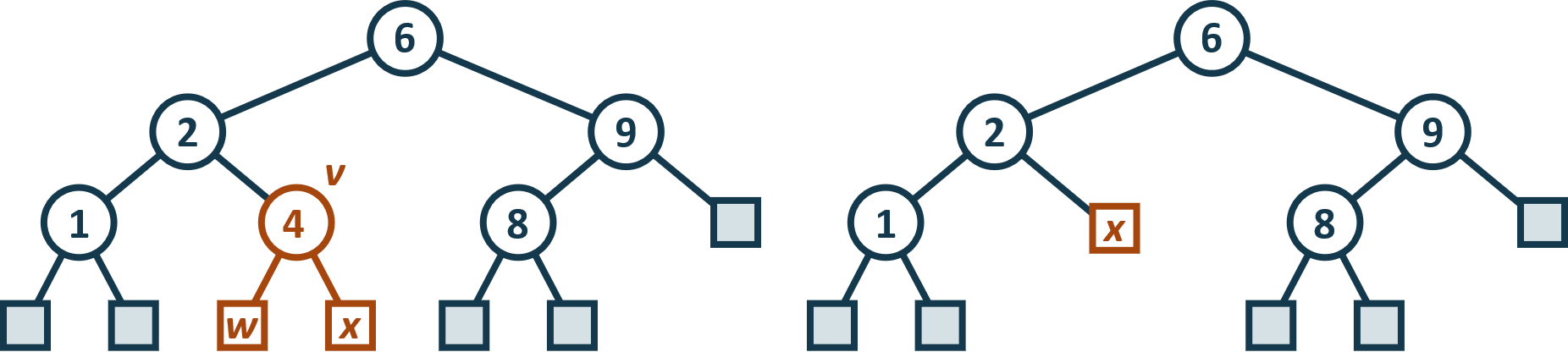
Value bei wird ersetzt

**einfügen**, falls schon vorhanden (multimap)

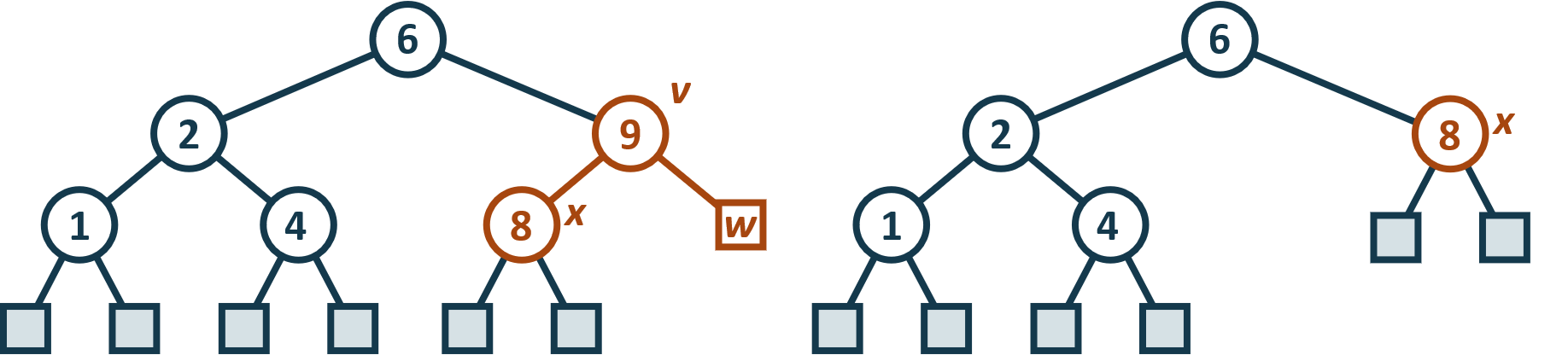
Im jeweils linken Teilbaum von weitersuchen, dann normal weiterfahren bis Blattknoten gefunden. bei einfügen und in einen internen Knoten expandieren.  
Ex: **insert(2)**

* + 1. Löschen des Key

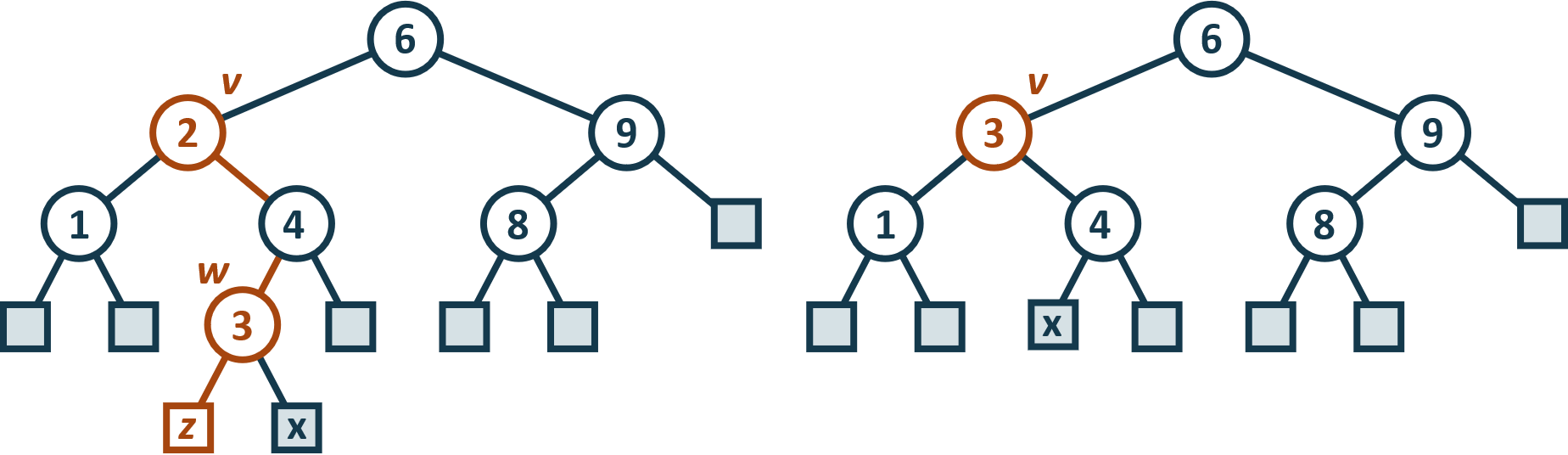
Zuerst Suche nach Key . Anschliessend gibt es drei verschiedene Fälle zu beachten:

1. Der Knoten mit Schlüssel ist ein Knoten mit zwei Blatt-Kinder (Knoten 4)
2. Der Knoten mit Schlüssel ist ein Knoten mit einem Blatt-Kind (Knoten 9)
3. Der Knoten mit Schlüssel ist ein Knoten ohne Blatt-Kinder (Knoten 2)
4. Löschen mit zwei Blatt-Kinder (Knoten 4)

Mit der Funktion **removeExternal(w)** wird und sein Eltern-Knoten gelöscht und durch den Geschwister-Knoten von ersetzt.

1. Löschen mit einem Blatt-Kind (Knoten 9)

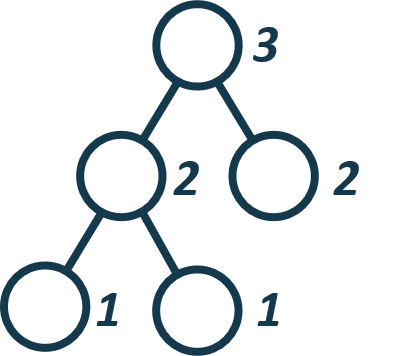
Mit der Funktion **removeExternal(w)** wird und sein Eltern-Knoten gelöscht und durch den Geschwister-Knoten von ersetzt.

1. Löschen ohne Blatt-Kinder (Knoten 2)
2. Finde den Knoten , welcher in der Inorder-Traversierung folgt.
3. Kopiere **key(w)** in den Knoten .
4. Lösche den Knoten und sein linkes Kind (welches wegen Inorder immer ein Blatt ist) mit der Funktion **removeExternal(**z**)**.
5. Das verbliebene Blatt wird eine Stufe weiter oben wieder angehängt.
   * 1. Gültige Suchpfade

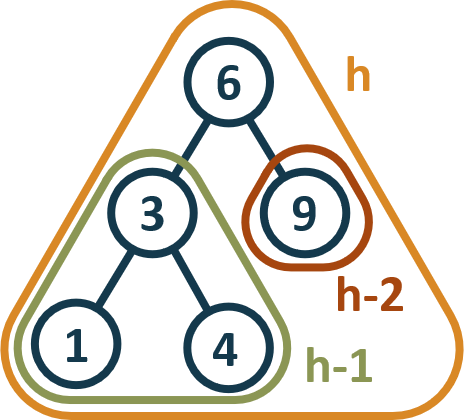
Ein Suchpfad ist nur gültig, wenn nach jedem Element anschliessend entweder nur noch höhere (wenn nächste Zahl grösser ist) oder niedrigere Elemente (wenn nächste Zahl kleiner ist) folgen. Wenn z.B. also nach 421 noch ein Element 410 UND ein Element 430 folgt, ist der Suchpfad nicht gültig.

Beispiel: 2 – 257 – 401 – 398 – 330 – 344 – 397 – 363 ist gültig  
925 – 202 – 911 – 240 – 918 – 245 – 363 ist ungültig, weil 918 grösser als 911 ist

## AVL Bäume (Adelson-Velsky Landis)

Ein AVL-Baum ist ein binärer Such-Baum, bei dem für jeden internen Knoten von gilt: die Höhe der Kinder von unterscheiden sich höchstens um 1. Ein AVL-Baum ist deshalb balanciert. Die Höhe eines AVL-Baumes wird von unten nach oben gemessen, d.h. der Root-Knoten hat immer die grösste Höhe.

### Höhe eines AVL Baumes

Die Höhe eines AVL-Baumes ist

Begründung: Zuerst versuchen wir, die minimale Anzahl Knoten , die für eine bestimmte Höhe benötigt werden als Funktion darzustellen: . Sicher ist: und. Für umfasst ein minimaler   
AVL-Baum der Höhe die Wurzel, einen grösseren AVL-Unterbaum der Höhe und einen zweiten, kleineren AVL-Unterbaum der Höhe .   
Also gilt:

Da können wir die obige Formel umschreiben als (das 1 kann weggelassen werden, weil auch ohne das 1 grösser ist)

Daraus folgt, dass in diesem Fall grösser sein muss als

Rekursiv gilt: , was grösser sein muss als woraus folgt, dass

Daraus lässt sich verallgemeinern, dass .

Da , können wir setzen. Daraus folgt:

Was bedeutet, dass die Höhe eines AVL-Baumes tatsächlich ist.

### Balancierungsfaktor

Ein AVL-Baum ist balanciert (. Das heisst, die Höhe des linken und des rechten Teilbaums eines Knoten unterscheidet sich maximal um eins.   
   
 kann also nur entweder oder sein.   
Die untersten Knoten (Blätter) haben immer einen Balancierungsfaktor von 0. Bei den anderen Knoten wird der linke Teilbaum minus der rechte Teilbaum gerechnet.

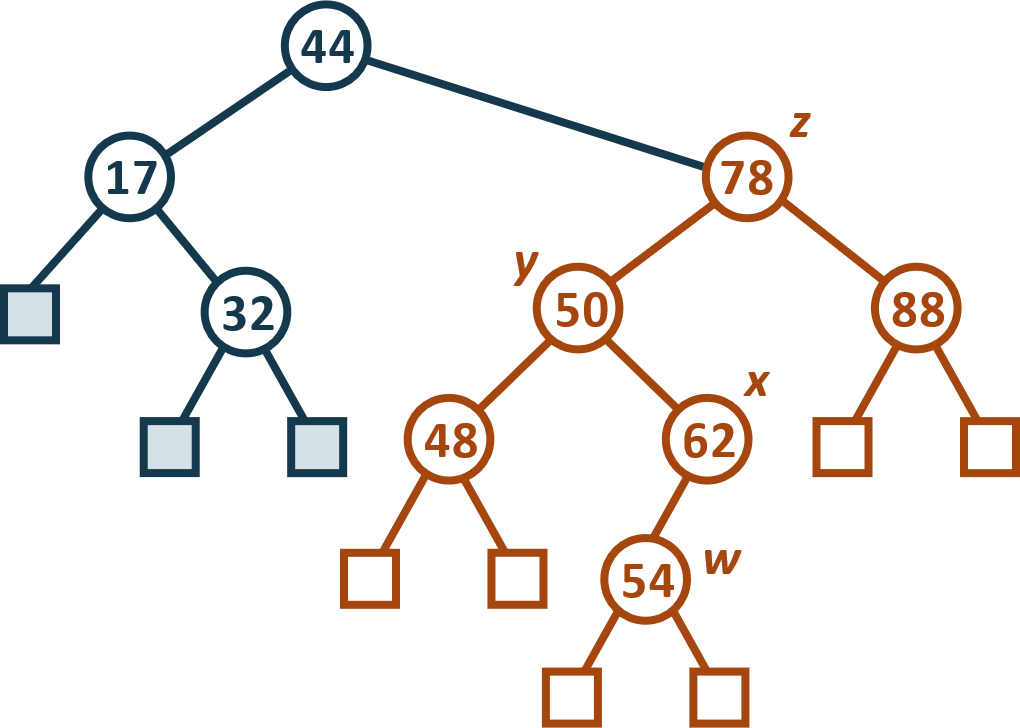
* Linker Teilbaum um 1 höher:
* Rechter Teilbaum um 1 höher:
* Beide Teilbäume gleich hoch:

### Einfügen in einen AVL Baum, Balancierungsfaktr

Nach dem Einfügen eines neuen Knotens kann allerdings +2/-2 sein (), womit der Baum nicht mehr balanciert, also kein AVL-Baum mehr ist. Nun muss der Baum rotiert werden, damit ein gültiger AVL-Baum wiederhergestellt wird.

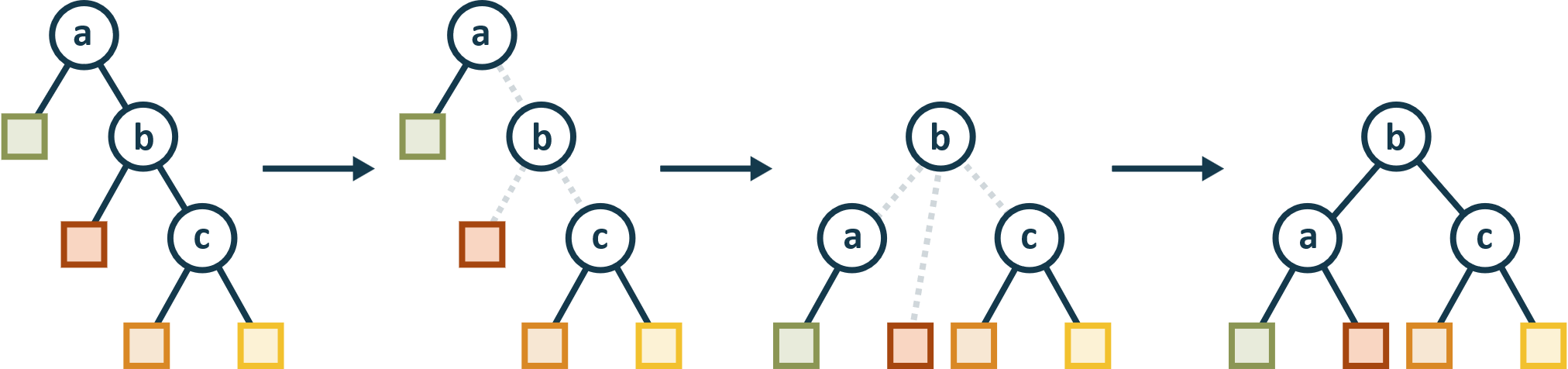
Verletzungen des AVL-Merkmals können in folgenden Fällen auftreten:

* Einfügen eines Knotens in den ***linken Teilbaum*** des ***linken Kindes***
* Einfügen eines Knotens in den ***rechten Teilbaum*** des ***linken Kindes***
* Einfügen eines Knotens in den ***rechten Teilbaum*** des ***rechten Kindes***
* Einfügen eines Knotens in den ***linken Teilbaum*** des ***rechten Kindes***

Der oberste Knoten mit dem zu grossen Balancierungsfaktor ist der Knoten, das direkte Kind im längeren Teilbaum ist und das direkte Kind von im längeren Teilbaum ist . Diese Knoten werden inorder mit a, b und c beschriftet.

* + 1. Trinode Umstrukturierung

Sei ein Inorder-Listing. Durchführung der nötigen Rotationen, damit zum obersten Knoten (=Root) des Baumes, zum linken und zum rechten Kind wird. = Root-Knoten von unbalanciertem Teilbaum, = Höchster direkter Kindknoten von , = Höchster direkter Kindknoten von (Grosskind von ).

Linksrotation

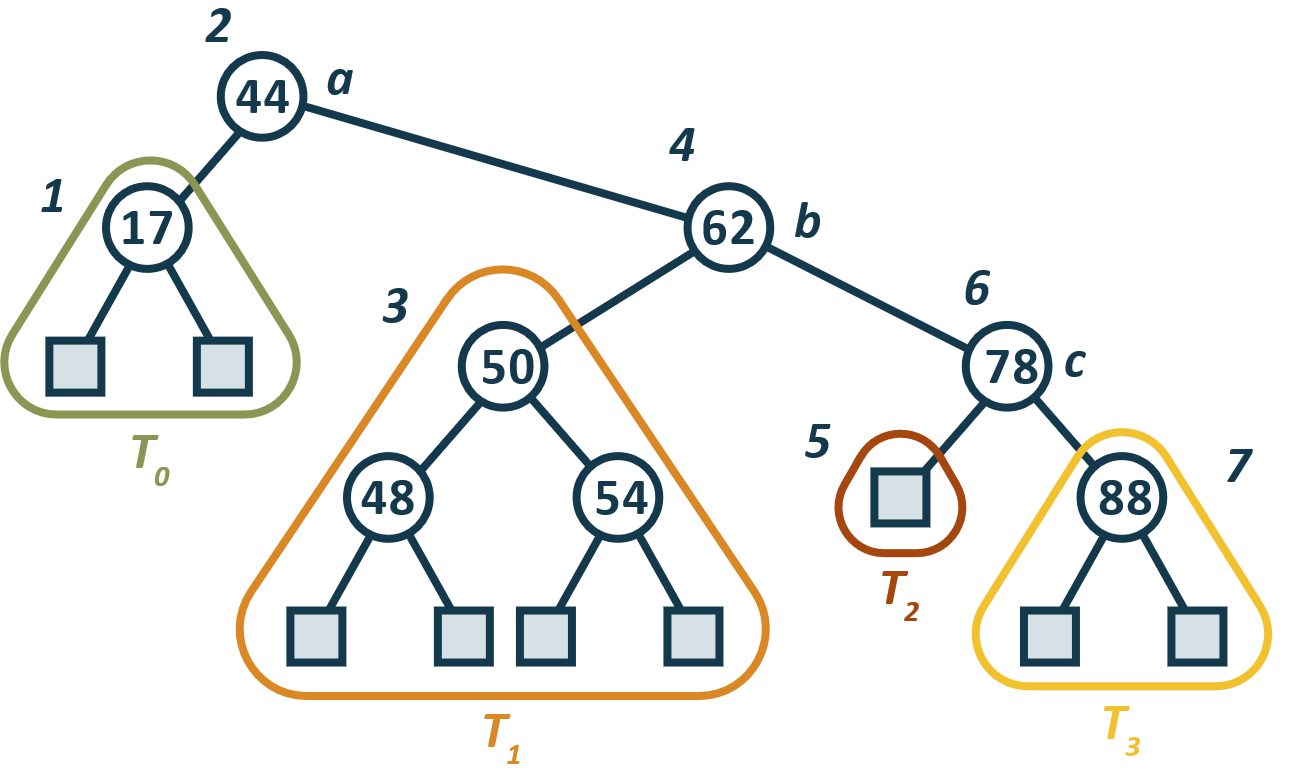
1. Alle Kanten von trennen
2. als root setzen
3. Kanten von und linkem Kind verbinden

Ein Bild, das Clipart, Cartoon, Design enthält.

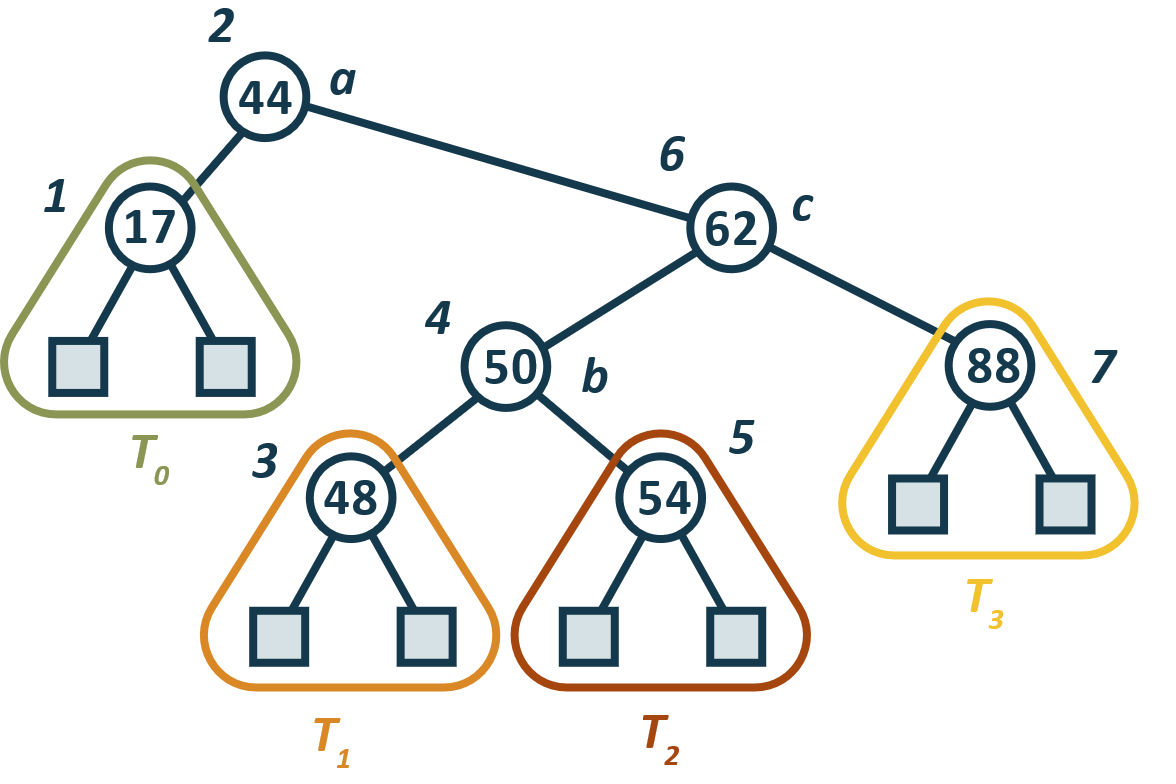
Automatisch generierte Beschreibung mit mittlerer ZuverlässigkeitRechtsrotation

1. Kanten , und vom rechten Kind von trennen
2. Knoten eine Ebene nach oben setzen
3. Kanten von und rechtem Kind verbinden

|  |  |
| --- | --- |
| Fall 1: Einzel-Rotation (Knoten sind inorder)  (eine Links-Rotation um ) | Fall 2: Doppelte Rotation (Knoten sind nicht inorder) (eine Rechts-Rotation um , dann eine Links-Rotation um ) |
|  |  |

* + 1. Cut/Link Restrukturierungs-Algorithmus

Knoten eines unbalanced Tree umordnen, damit die Ordnung bei der Inorder-Traversierung erhalten bleibt und der Tree balanced wird. Dazu labeln wir den Grosseltern-, Eltern- und den Knoten selbst wie folgt: Der oberste Knoten mit dem zu grossen Balancierungsfaktor ist der Knoten, das direkte Kind im längeren Teilbaum ist und das direkte Kind von im längeren Teilbaum ist . Diese Knoten werden inorder mit a, b und c beschriftet. Die restlichen Kinder dieser Knoten werden von links nach rechts als Teilbäume gelabelt.

Wir erstellen einen Array mit 7 Elementen 1 bis 7 weil 3 Hauptknoten & 4 Subtrees:

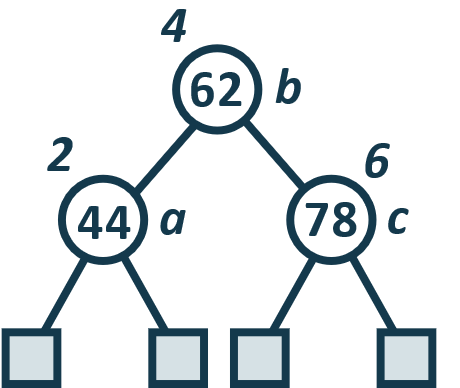
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
| *1* | *2* | *3* | *4* | *5* | *6* | *7* |

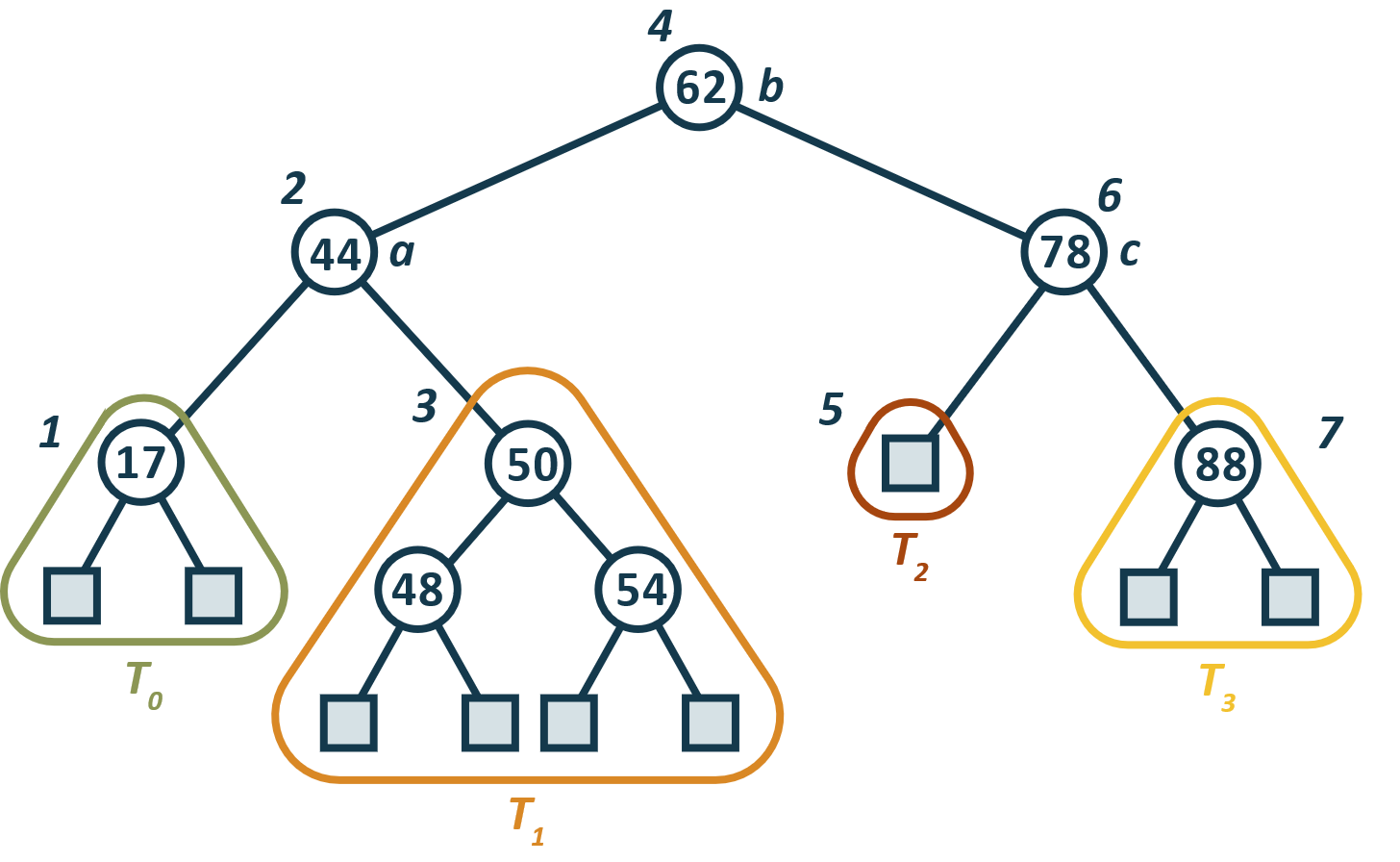
Nun scheiden wir die 4 Bäume ab und platzieren sie inorder in das Array mit jeweils 1 Position Abstand.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
| *1* | *2* | *3* | *4* | *5* | *6* | *7* |

Jetzt stellen wir , und in Inorder in die Positionen zwischen den Trees in das Array.

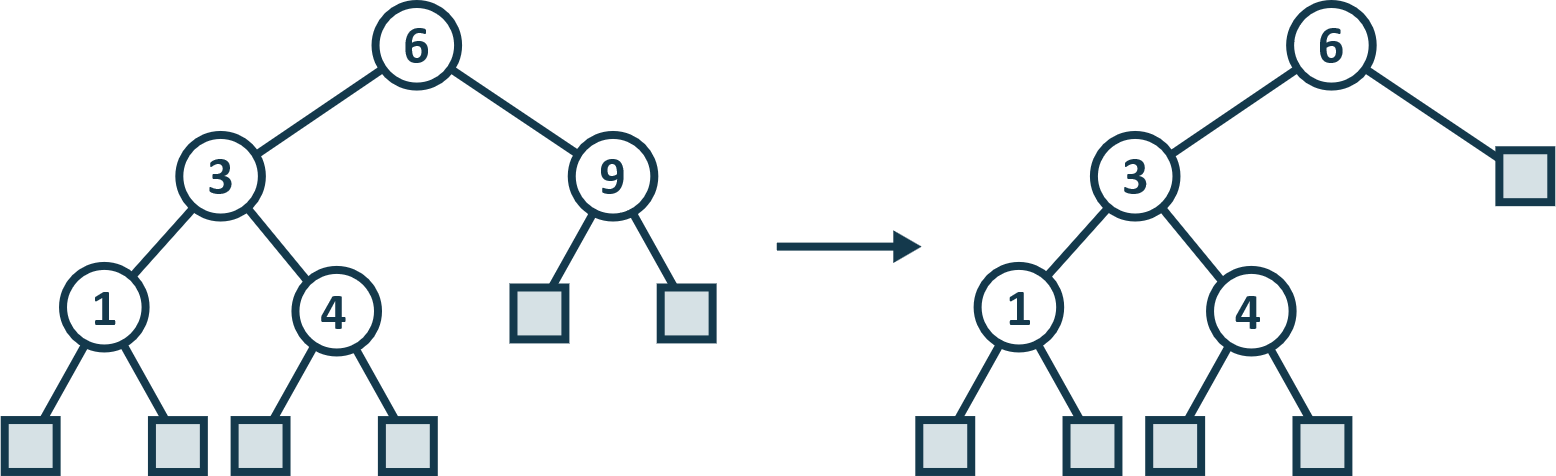
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
| *1* | *2* | *3* | *4* | *5* | *6* | *7* |

Nun bauen wir den Baum schrittweise wieder auf. Wir beginnen mit und setzen die anderen beiden Hauptknoten Element 2 () und Element 6 () als Kinder von 4 () in den Baum ein.

Im nächsten Schritt setzen wir die Subtrees an den Positionen 1, 3, 5 und 7 als Kinder von 2 und 6 in den Baum ein. Das Ergebnis ist ein balancierter Baum.

Dieser Algorithmus bewirkt eine Balancierung, gleich wie die Rotationen.  
Vorteil: Keine Fallunterscheidung, eleganter  
Nachteil: Komplexerer Programmcode  
Komplexität: gleich wie Rotation

### Löschen

Beginnt wie im binären Suchbaum, das heisst, der gelöschte Knoten wird ein leerer externer Knoten. Sein Eltern-Knoten kann jetzt die Balance aus dem Gleichgewicht bringen.

Deshalb muss nach dem Löschen wieder eine Rotation oder ein Restaurierungs-Algorithmus durchgeführt werden. Da dies aber eine neue Unbalance höher im Baum auslösen kann, muss anschliessend bis zum Root auf die Balance überprüft werden.

## Splay-Trees

Ein Splay-Baum ist ein Binärer Such-Baum, der nach jeder Operation (auch nach Suche) eine Rotation durchführt, um das zuletzt verwendete Element zum Root zu machen. So sind die am häufigsten verwendeten / gesuchten Elemente am schnellsten zu finden. Nützlich für Caching.

Splay-Trees sind binäre Such-Bäume mit folgenden Regeln:

* Entries sind nur in internen Knoten gespeichert
* Keys gespeichert im linken Teilbaum von sind kleiner oder gleich wie der Key in
* Keys gespeichert im rechten Teilbaum von sind grösser oder gleich wie der Key in

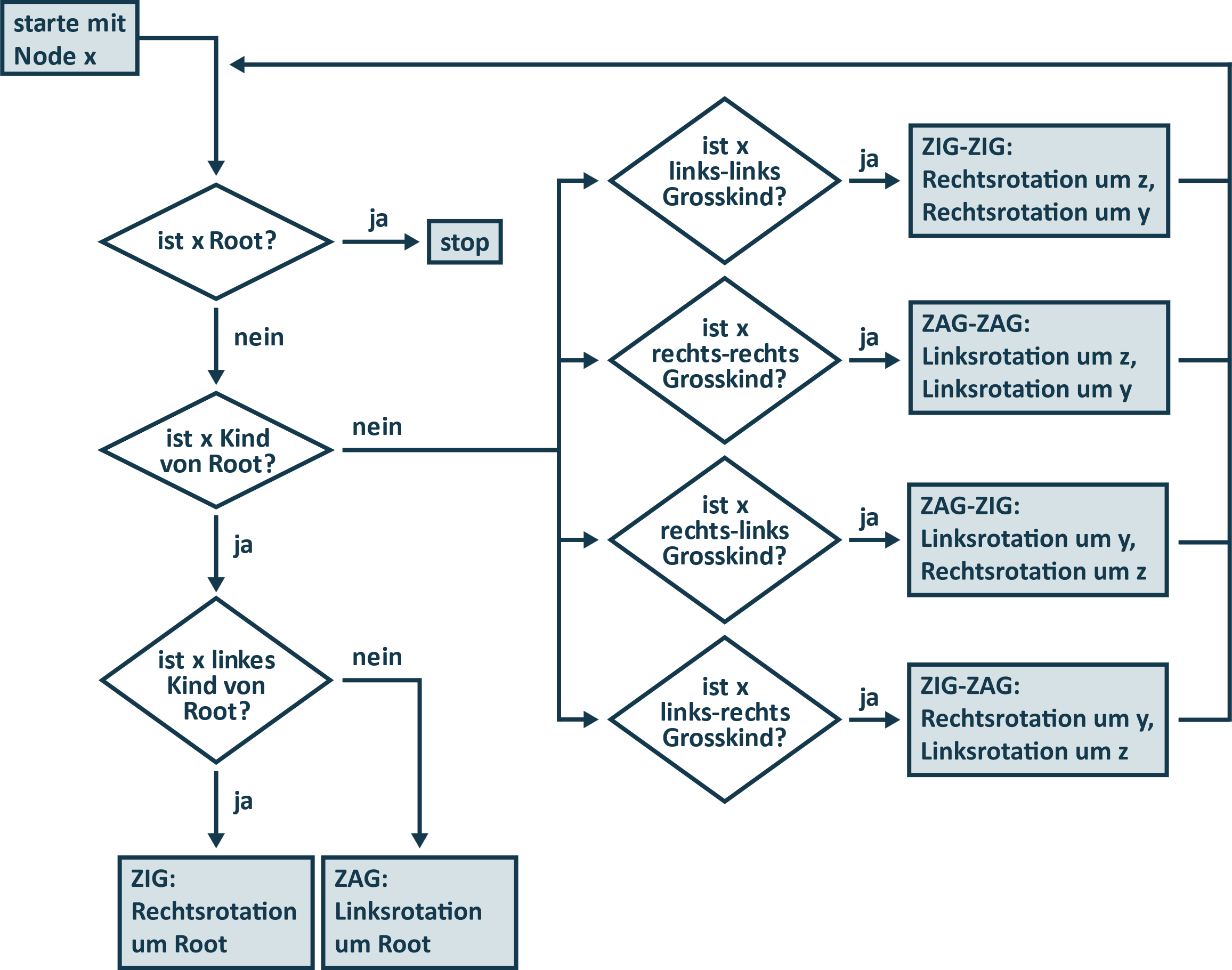
Eine Inorder-Traversierung retourniert die Keys in geordneter Folge.

### Suchen in einem Splay-Tree

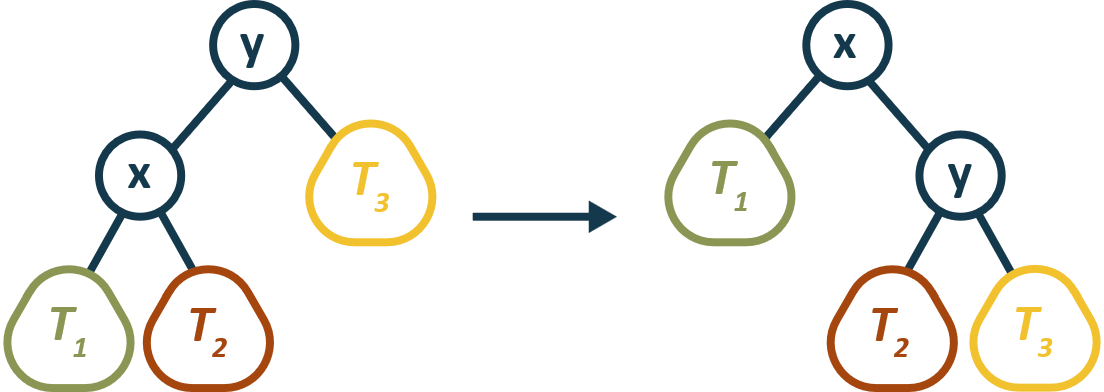
Suche funktioniert gleich wie in einem Binary Search Tree: Man geht den Baum abwärts bis zu einem gesuchten Entry oder einem externen Knoten.

### **Splay**

Der tiefste interne zugegriffene Knoten ist der Root (splaying).

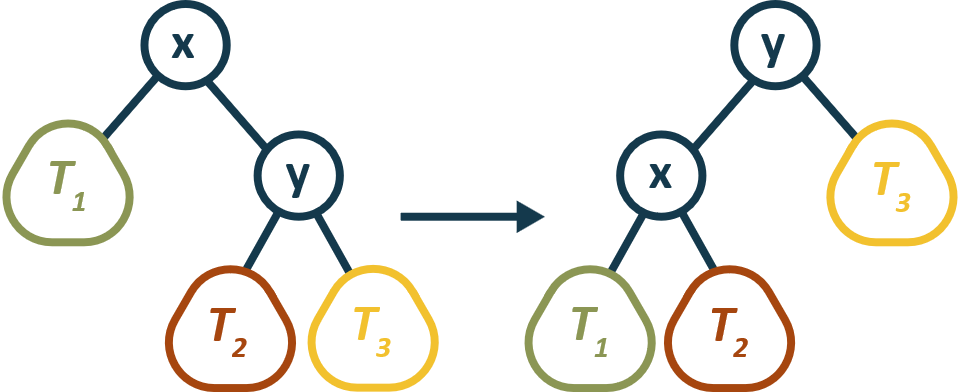
ZIG (Rechtsrotation)

Macht das linke Kind des Knoten zu ’s Eltern-Knoten, wird zum rechten Kind von



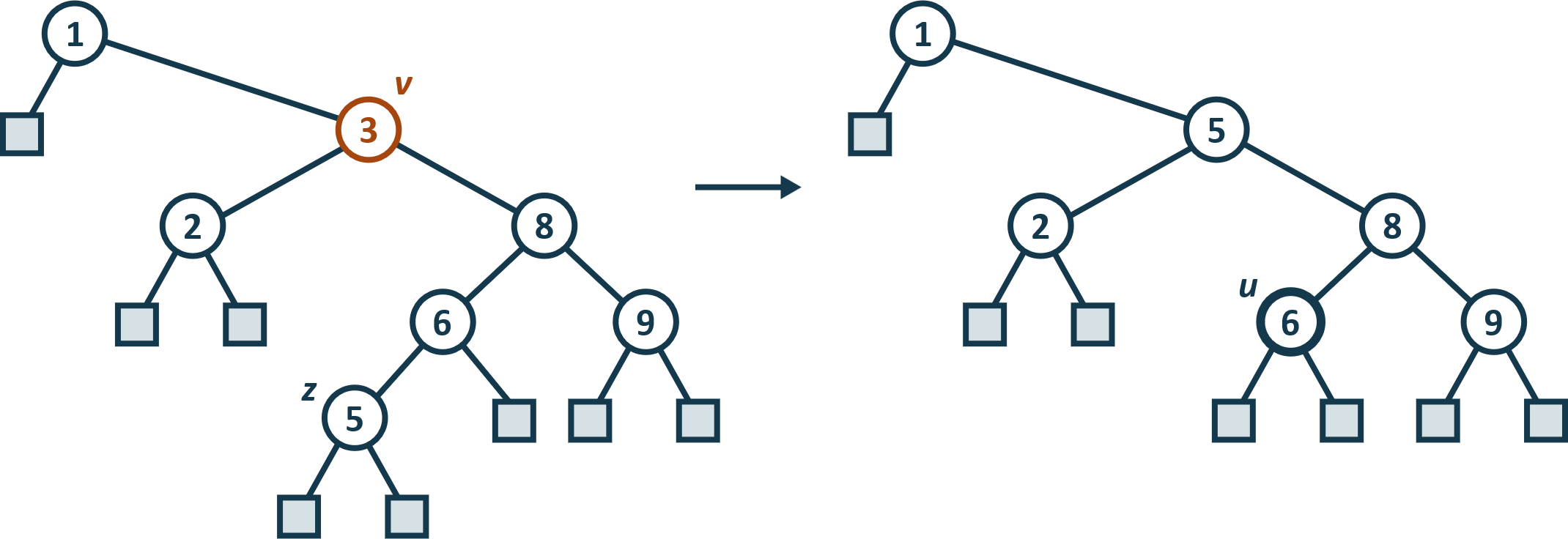
ZAG (Linksrotation)

Macht das rechte Kind des Knoten zu ’s Eltern-Knoten, wird zum linken Kind von

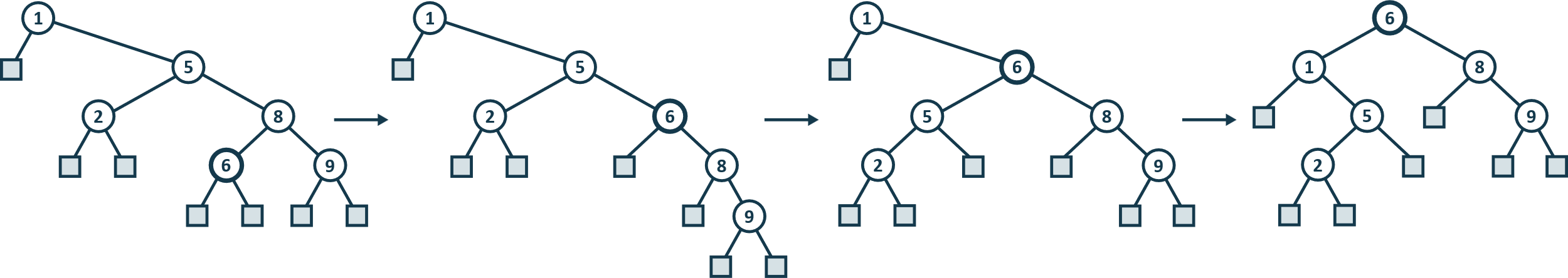


|  |  |
| --- | --- |
| Zig-Zig (Rechts-Rechts) | Zag-Zag (Links-Links) |
| Zig-Zag (Rechts-Links) | Zag-Zig (Links-Rechts) |

Beispiel für Löschen: remove(3)  
Da 2 interne Knoten hat, wird der Fall 3 angewendet (siehe Löschen ohne Blatt-Kinder (Knoten 2)): Ersetze durch Inorder-Nachfolger und lösche . Das rechte Blatt von wird an angehängt, damit ist dieser der tiefste intern zugegriffene Knoten, somit wird zum neuen Root. Dafür wird eine Rechts-Links/Zig-Zag Rotation benötigt.



**Balance-Wiederherstellung durch Splaying, Ablauf:** Zig-Zag, Zig-Zag, Zag:



Welcher Knoten wird nach jeder Operation «gesplayed»?

|  |  |
| --- | --- |
| Methode | Splay Knoten |
| find(k) | Wenn Key gefunden: Knoten mit Key.  Wenn Key nicht gefunden, Eltern-Knoten des externen Knoten am Ende |
| insert(k,v) | Neuer Knoten, welcher hinzugefügt wird |
| remove(k) | Eltern-Knoten des internen Knoten, welcher gelöscht wurde. Achtung: Parent ist bei Knoten ohne Blättern der Eltern-Knoten des Inorder Nachfolger, und nicht der, der ursprünglich der Funktion übergeben wird |

## Vergleichsbasierte Sortieralgorithmen

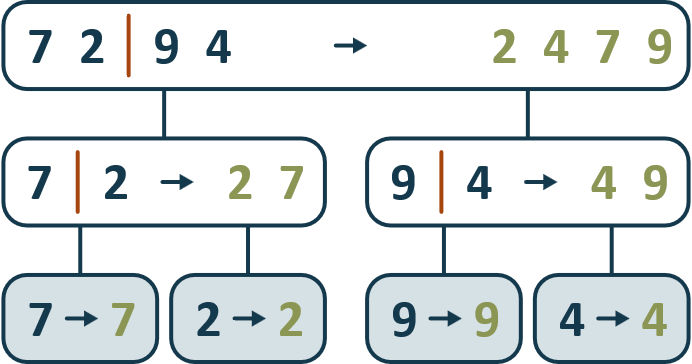
Vergleichsbasierte Sortieralgorithmen sortieren eine Liste, indem sie jeden Wert in der Liste mit mindestens einem anderen vergleichen, um so die korrekte Stelle zu finden. Sie funktionieren mit beliebigen Inputs, solange ein Komparator bzw. ein Vergleichsmechanismus für die entsprechenden Objekte vorhanden ist.

### Merge-Sort

Merge-Sort basiert auf dem Divide-and-Conquer (Teile-und-Herrsche) Paradigma.

* Divide: Input-Daten in zwei getrennte Hälften und aufteilen
* Recur (Wiederhole): Rekursiv die Teilmengen und sortieren
* Conquer: und in eine sortierte Sequenz mischen

Die Verankerung (Base Case) der Rekursion sind Teilprobleme der Grösse 0 oder 1 (Die Liste wird so lange geteilt, bis sich 0 oder 1 Element in der Liste befinden). Da wir nur an den Enden auf die Daten zugreifen, sind , & idealerweise double-linked Lists, da diese vorne und hinten mit zugreifen können.

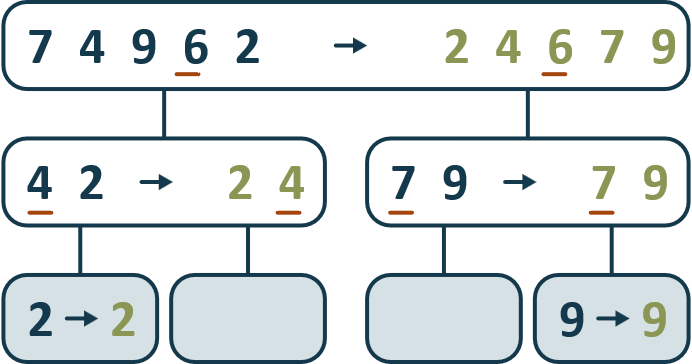
Die Ausführung eines Merge-Sort kann als binärer Baum dargestellt werden. Jeder Knoten repräsentiert einen rekursiven Aufruf des Merge-Sort und enthält die unsortierte und die sortierte Sequenz. Die Wurzel entspricht dem initialen Aufruf. Die Blätter sind Aufrufe auf Teilsequenzen der Grösse 0 oder 1.

Beispiel: wird so lange rekursiv in der Mitte aufgeteilt, bis ist, dann wird zurückgegeben. Ist dann werden die beiden vereint: Die Elemente werden verglichen, der kleinere Wert (2) zuhinterst in eingefügt und aus entfernt. Da nun leer ist, werden alle Elemente aus direkt in eingefügt. Das Ganze wiederholt sich mit und .

### Quick-Sort

Quick-sort basiert auf dem Divide-and-Conquer Paradigma. Es funktioniert ähnlich wie Merge-Sort, nur dass hier die Liste nicht halbiert wird, sondern die Aufteilung aufgrund Vergleiche mit einem bestimmten Element (Pivot) geschieht. Der optimale Pivot wäre der Median. Diesen zu finden ist jedoch zeitaufwändig. Deshalb kann man als Kompromiss statt eines zufälligen Elementes (wie in der Vorlesung) oder dem Median auch den Median aus einer Stichprobe der gesamten Menge als Pivot verwenden.

* Divide: Auswahl eines (zufälligen) Elementes (genannt Pivot) und Aufteilung von in (less, Elemente kleiner als ), (equal, Elemente gleich ) und (greater, Elemente grösser als )
* Recur: sortiere und
* Conquer: vereine , und zu einer Liste

Die Ausführung eines Quick-Sort kann als binärer Baum dargestellt werden. Jeder Knoten repräsentiert einen rekursiven Aufruf des Quick-Sort und enthält die unsortierte und die sortiere Sequenz. Die Wurzel entspricht dem initialen Aufruf. Die Blätter sind Aufrufe auf Teilsequenzen der Grösse 0 oder 1.

Beispiel: . 6 wird als Pivot ausgewählt. Nun werden alle anderen Elemente mit 6 verglichen, aus entfernt und entweder in , oder eingeteilt. & werden rekursiv weiter aufgeteilt, bis die Rekursion mit 1 Element in der Liste beendet wird. Die Listen werden dann wieder zu einer vereinigt, indem , & nacheinander eingefügt werden. Diese Liste wird ein Rekursionslevel hochgegeben, bis die Originalliste sortiert ist.

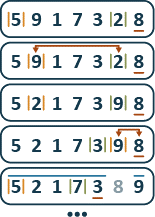
Anzahl Vergleiche ist auf jeder Höhe die Anzahl der Elemente – Anzahl der Pivots auf dieser Stufe.

* + 1. In-Place Quick-Sort

Quick-Sort kann auch In-Place durchgeführt werden, also im selben Array, ohne dass die Elemente in weitere Arrays kopiert werden müssen. Dadurch wird vor allem bei grossen Inputs viel RAM gespart.

Im Partitionierungs-Schritt werden die Elemente der Input-Sequenz derart umgeordnet, dass

* Die Elemente ***Pivot*** einen Index kleiner als haben
* Die Elemente ***Pivot*** einen Index grösser als haben

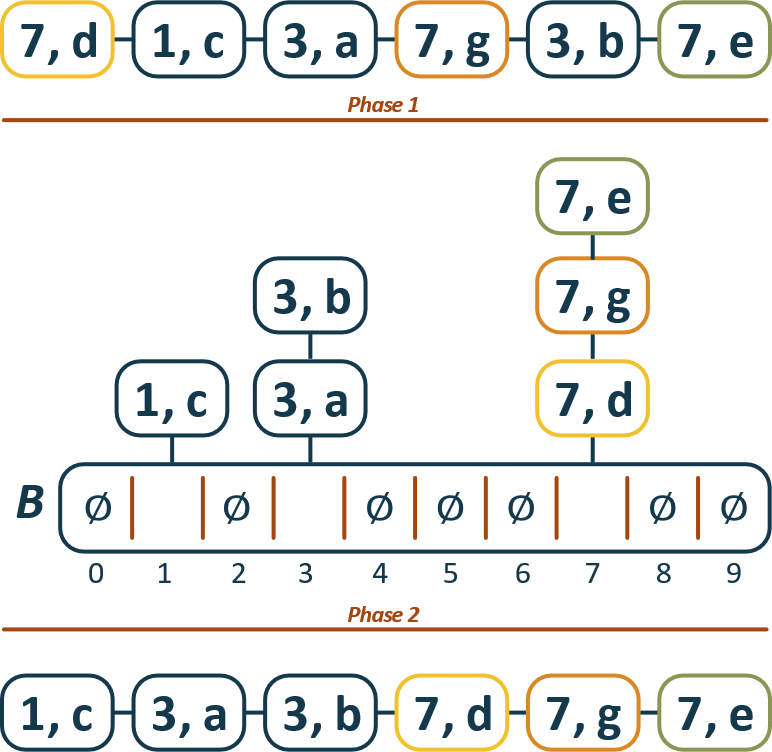
Das erste Element wird mit gekennzeichnet, das letzte mit . Der ***Pivot*** bleibt ein zufälliges Element. wird nach rechts geschoben, bis es entweder ein Element ***Pivot*** trifft oder kreuzt. wird nach links geschoben, bis es entweder ein Element ***Pivot*** trifft oder kreuzt. Haben und angehalten, ohne sich zu kreuzen, werden die Elemente an diesen Stellen vertauscht und dann und weiter verschoben. Kreuzen sich und , wird das Element bei mit dem ***Pivot***-Element vertauscht. Entstandene Teillisten (Elemente links (kleiner als Pivot) bzw. rechts (grösser gleich Pivot) vom Pivot) werden rekursiv weiter überprüft.

### Untere Grenze der Vergleichsbasierten Sortierung (Lower Bound)

Jeder Vergleichs-basierte Sortier-Algorithmus muss mindestens eine Laufzeit von haben.

Erklärung: Jeder vergleichsbasierte Algorithmus kann als Binary Tree dargestellt werden (Entscheidungsbaum). In diesem Entscheidungsbaum gibt es Knoten. Je nachdem wie ein Vergleich von zwei Elementen ausfällt, ändert sich der Pfad durch den Entscheidungsbaum (z.B. führt zu einem anderen Pfad als ). Da ein Entscheidungsbaum immer ein balancierter Binary Tree ist, ist dessen Höhe , eine Pfadtraversierung ist also . Mit der Stirling-Approximation kann dies als etwa beschrieben werden.

## Nicht-Vergleichsbasierte Sortieralgorithmen

Nicht-Vergleichsbasierte Sortieralgorithmen können schneller sein als vergleichsbasierte. Sie funktionieren aber nur mit bestimmten Inputs (Integers) als Keys. Lower Bound dieser ist .

### Bucket-Sort

Sei eine Sequenz von (Key, Element) Entries mit Keys im Bereich von Bei Bucket-Sort werden die Entries anhand der Keys in einem Hilfsarray mit Grösse abgelegt. Hinter jedem Index von befindet sich eine Liste (Bucket) in welchen die Entries zwischengespeichert werden. Die Keys sind dann sortiert und können dem Index nach wieder aus den Buckets in herausgenommen werden.

* Phase 1: Sequenz leeren durch Verschieben jedes Entry in sein Bucket
* Phase 2: Für , verschiebe die Entries des Buckets an das Ende der Sequenz S
  + 1. Eigenschaften
* Stabil, weil Elemente mit gleichem Key im sortierten Array in der gleichen Reihenfolge vorkommen wie im unsortierten Array (siehe Entries mit Key 7).
* Es ist kein externer Komparator nötig.
* Die Keys werden als Indices in einem Array benutzt und müssen deshalb Integers sein.
* Die Anzahl Keys kann sehr gross werden, ist also nur effektiv bei vielen Elementen mit gleichen Keys (z.B. Adressen nach PLZ)
* Kann mit einem zusätzlichem Sortieralgo erweitert werden, damit innerhalb der Buckets sortiert wird

### Lexikographische ordnung (Stable sort)

In einem Tupel werden die Werte der gleichen Dimension der Reihe nach verglichen. Die Lexikographische Sortierung sortiert eine Sequenz von -Tupeln in lexikographischer Ordnung, indem -mal ein stabiler Sortier-Algorithmus («stableSort»), welcher den Comparator benutzt, durchgeführt wird. Beispiel: Sortieren nach Nachnamen, dann Vornamen. Es wird in den Tupeln von hinten nach vorne sortiert.

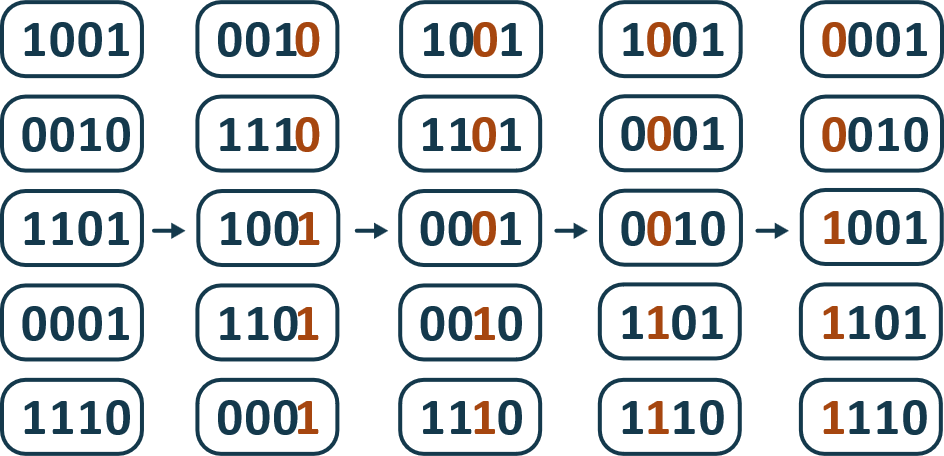
Im Bild werden die Tupel zuerst nach den hintersten Integers sortiert, dann nach den mittleren usw., bis im vordersten Tupel vorne der kleinste Integer steht. Durch die Stabilität stimmt bei gleichen Integers die Reihenfolge eine Dimension dahinter auch.

Mathematische Notation für Sortierung von zwei Tupeln :

### Radix-Sort

Radix-Sort ist eine Spezialisierung der lexikographischen Sortierung, welcher Bucket-Sort als stabiler Sortier-Algorithmus für jede Dimension benutzt. Es ist anwendbar für Tupel mit Integer-Keys im Bereich in jeder Dimension . Es wird wie bei der lexikographischen Sortierung jede Dimension der Tupel von hinten via Bucket Sort sortiert. Diesem wird neben dem Input noch die maximale Keygrösse sowie die aktuelle Dimension übergeben.

* + 1. Binärer Radix Sort

Gegeben sei eine Sequenz von -Bit Integers . Jedes Element wird als -Tupel von Integern im Bereich dargestellt. Darauf wendet man den Radix-Sort mit an (da nur 0 und 1 als Keys möglich sind). Diese Anwendung des Radix-Sort-Algorithmus läuft in Zeit. Beispiel: Eine Sequenz von 32-Bit Integers kann in linearer Zeit sortiert werden, nämlich .

## Zusammenfassung der Sortier-Algorithmen

Stabilität: Gleiche Elemente sind nach der Sortierung noch in der gleichen Reihenfolge wie zuvor

|  |  |  |
| --- | --- | --- |
| Algorithmus | Zeitverhalten | Bemerkungen |
| **selection-sort** |  | * Langsam & instabil * in-place * für kleine Data-Sets * Profitiert nicht von einem vorsortierten Input |
| **insertion-sort** |  | * Langsam & stabil * in-place * für kleine Data-Sets * Für kleine Inputs besser als selection-sort |
| **Quick-sort** | durchschnittlich, wegen zufälligem Pivot | * Kann in-place implementiert werden (nicht stabil) * meist instabil (implementationsabhängig) * schnellster (gut für grosse Inputs) |
| **heap-sort** |  | * schnell & instabil * in-place * für grosse Data-Sets |
| **merge-sort** |  | * schnell & stabil * sequenzieller Datenzugriff (es müssen jeweils nur 2 Elemente gleichzeitig in den RAM geladen werden) * für riesige Data-Sets |
| **Bucket-sort** |  | * Benötigt Integer als Keys * Stabil |
| **Lexikographische Sortierung** | , wobei der einzelnen Sortierung des anderen Algos ist | * Benötigt Integer als Keys * In-place * Stabil |
| **Radix-Sort** |  | * Benötigt Integer oder Bits als Keys * In-place * Stabil |

## Pattern Matching

Ein String ist eine Sequenz von Characters (Zeichen). Beispiele von Strings: Java Programm, HTML-Dokument, DNA-Sequenz oder ein digitalisiertes Bild. Definition: Ein Alphabet ist ein Set von möglichen Zeichen für eine Familie von Strings. Beispiele für Alphabete: ASCII, Unicode, , .

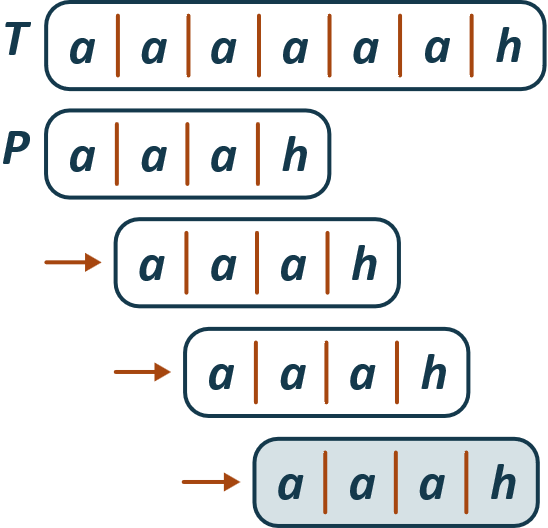
Sei (Pattern) ein String der Länge . Ein Substring von ist die Subsequenz von , bestehend aus den Zeichen mit Index zwischen und inklusiv und .

Spezielle Substrings

* Ein Präfix von ist ein Substring vom Typ alle Zeichen bis
* Ein Suffix von ist ein Substring vom Typ alle Zeichen nach

Gegeben: Strings (Text) und (Pattern).  
Das Ziel vom Pattern-Matching ist es, einen Substring in zu finden, der mit übereinstimmt.

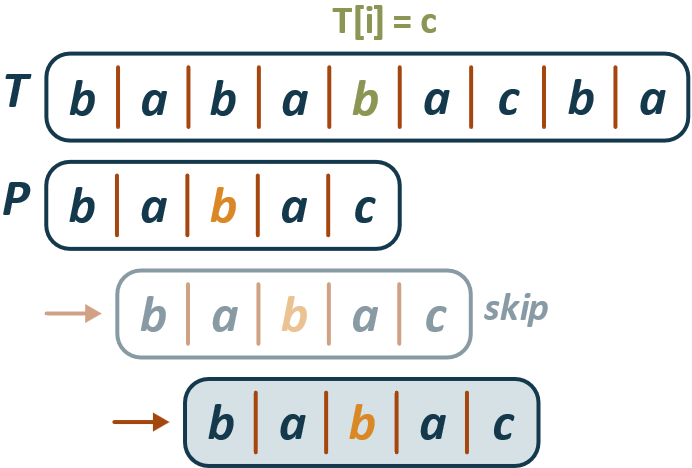
### Brute-Force Pattern Matching

Der Brute-Force Pattern Matching Algorithmus vergleicht das Pattern mit dem Text für jede mögliche Position von relativ zu , bis entweder eine Übereinstimmung gefunden wurde oder alle möglichen Platzierungen des Patterns ausprobiert wurden. Sind 2 einfache Loops für & bei dem der Index immer um 1 erhöht wird. Benötigt Zeit, ist also sehr ineffizient.

Worst-Case Beispiel: , . Kann in Bild-Analysen und DNA-Sequenzanalysen eintreten, in sprachlichen Texten eher nicht.

### Boyer-Moore Algorithmus

Der Boyer-Moore Pattern Matching Algorithmus basiert auf folgenden Verfahren:

* «Looking-Glass: Vergleiche mit einer Subsequenz von . Startet am Anfang des Strings, die einzelnen Charakters werden mit dem Pattern aber von hinten nach vorne verglichen.
* «Character-Jump»: falls bei keine Übereinstimmung: falls das Zeichen enthält, verschiebe bis das letzte Auftreten von in mit übereinstimmt. Ist das hinterste Auftreten vor der aktuellen Position, wird stattdessen um 1 geschoben.

Ansonsten, verschiebe , bis mit übereinstimmt (das ganze Pattern vor schieben). Damit lassen sich Positionen, welche ohnehin keinen Match geben würden, überspringen.

Im Worst Case () muss jede einzelne Position im Pattern geprüft werden, dann das Pattern um 1 verschoben und wieder geprüft werden, bis am Ende von angelangt ist. Kommt vor allem bei Bild & DNA-Sequenzen vor, dann sollte KMP verwendet werden.

* + 1. Last Occurrence Funktion

Der Boyer-Moore Algorithmus erstellt eine Last Occurrence Funktion, während er den Text durchsucht. In dieser Tabelle ist jeder Buchstabe nur maximal x vorhanden.

***Beispiel:*** (Index der Zahl ist Tiefgestellt) ergibt folgende Tabelle:

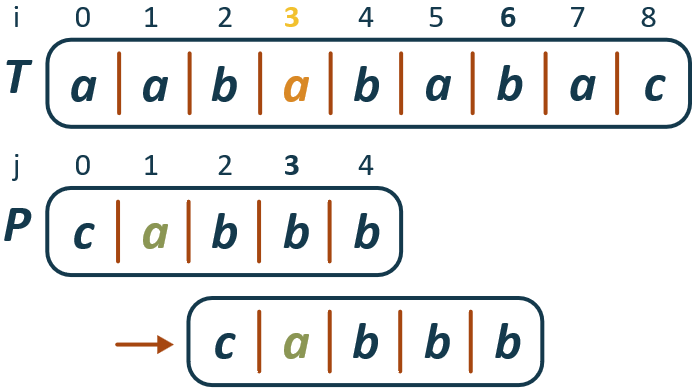
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | a | b | c | d |
| letztes Vorkommen des Zeichens |  |  |  | (kommt nicht vor) |

Diese Funktion lässt sich als ein Array darstellen, dessen Indices durch numerische Werte des Alphabets gegeben sind. Diese Funktion lässt sich in berechnen, wobei die Länge von und die Anzahl Zeichen in ist.

* + - 1. Berechnung der Verschiebung

**Fall 1: Zeichen kommt im Pattern vor**

Das Zeichen (Zeichen im Text an Stelle ) kommt im Pattern vor (). In diesem Fall müssen wir bis zum letzten Auftreten des Zeichens im Pattern verschieben. Die Berechnung des neuen ’s Index letztes Zeichen von P = i + Patternlänge – (Index letztes Vorkommen + 1):

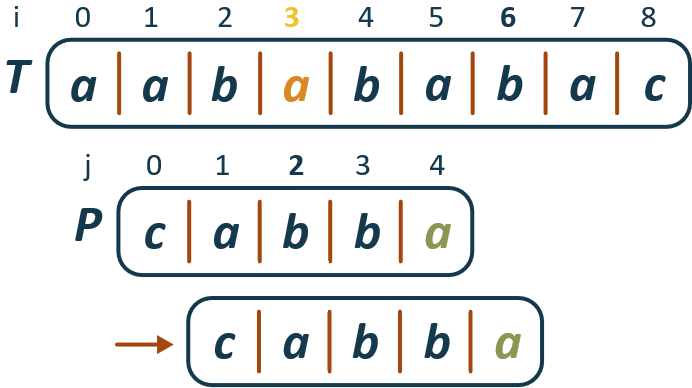
Beispiel im Bild:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| c | a | b | c | d |
|  |  |  |  |  |

Das nächste Zeichen, welches überprüft wird, ist das an Index 6.   
Das letzte Vorkommen vom Zeichen im Pattern wurde damit an die Stelle verschoben.

**Fall 2: Zeichen kommt vor, letztes Vorkommen ist aber bereits vorbei**

Das Zeichen kommt im Pattern vor (), ist aber bereits vorbei. In diesem Fall wird das Pattern einfach um eine Stelle nach vorne geschoben. Die Berechnung des neuen ’s: j = Index von P[i]

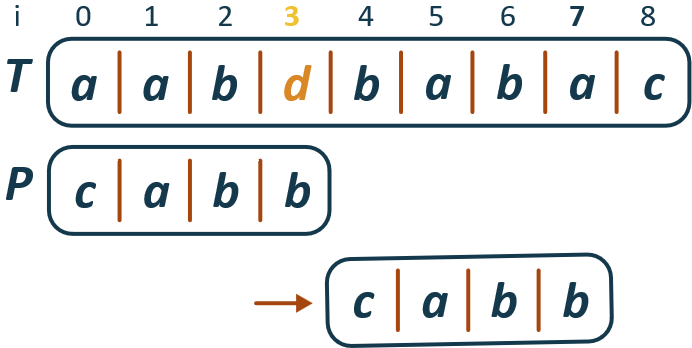
Beispiel im Bild:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| c | a | b | c | d |
|  |  |  |  |  |

Das nächste Zeichen, welches überprüft wird, ist das an Index 6.

Fall 3: Das Zeichen kommt nicht im Pattern vor

Das Zeichen kommt im Pattern nicht vor (). In diesem Fall wird das ganze Pattern vor das Zeichen geschoben, da zurückgibt, welches durch das wieder ausgeglichen wird.

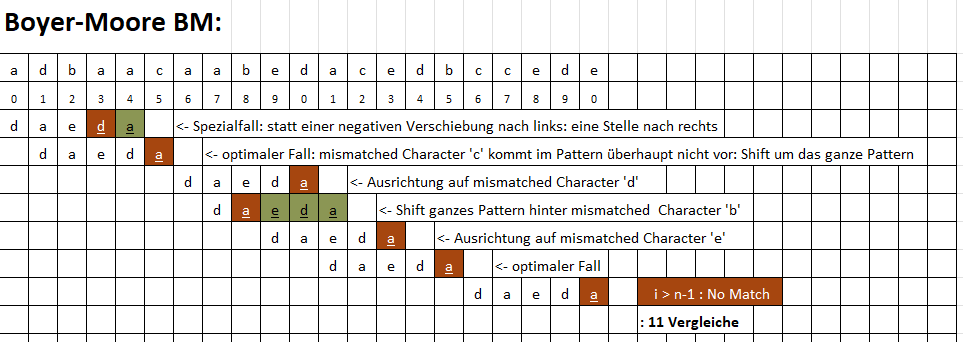
Beispiel im Bild:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| c | a | b | c | d |
|  |  |  |  |  |

Im Pseudocode wird die Fallunterscheidung mit beschrieben, wobei Fall 2 und Fall 1 & 3 entspricht.

Ein Bild, das Text, Screenshot, Diagramm, Schrift enthält.

Automatisch generierte Beschreibung



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| c | a | b | c | d | e |
|  |  |  |  |  |  |

### KMP Algorithmus

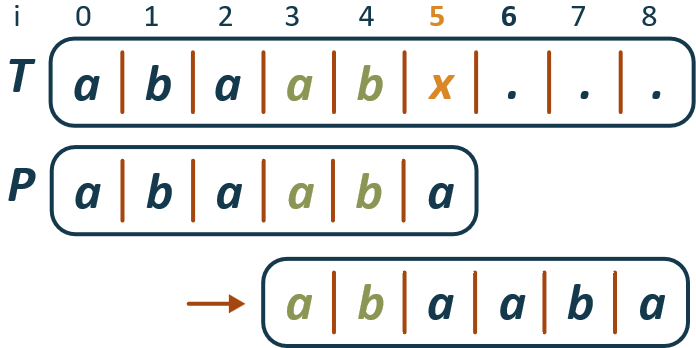
Der Knuth-Morris-Pratt Algorithmus vergleicht das Muster gegen den Text von links nach rechts, aber schiebt das Muster intelligenter als der Brute-Force Algorithmus. Bei Nichtübereinstimmung: Was ist das Maximum, um das das Muster verschoben werden kann, um redundante Vergleiche zu vermeiden? Antwort: Das längste Präfix von (längstes Teilpattern von String-Beginn) welches gleichzeitig Suffix von (längstes Teilpattern bei String-Ende) ist. Beispiel:

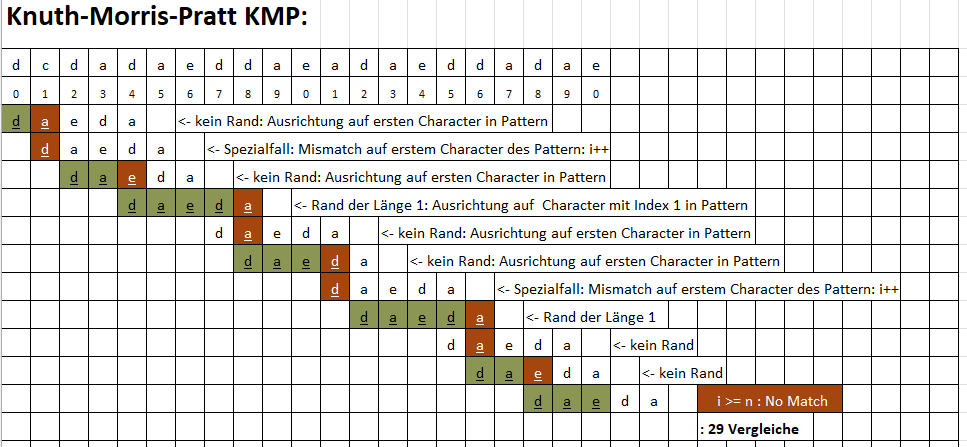
* + 1. KMP Fehl-Funktion

In einer Vorlaufsphase (Preprocessing) sucht der Algorithmus Übereinstimmungen von Präfixen des Musters im Muster selbst. Die Fehl-Funktion ist definiert als die Grösse des längsten Präfixes von , sodass dieser auch Suffix von ist. KMP modifiziert den Brute-Force-Algorithmus so, dass bei einer Differenz der Index gesetzt wird mit

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Ränder-Länge:** |  |  |  |  | **max. Länge** |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| (index) |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

Bei steht im Text ein , und im Pattern ein . Das längste passende Suffix von ist in diesem Fall ( passt nicht, da hinten das steht). Die Fehlerfunktion von steht am Index also wird das Pattern um nach vorne geschoben.



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | d | a | e | d | a |
|  |  |  |  |  |  |

## Tries

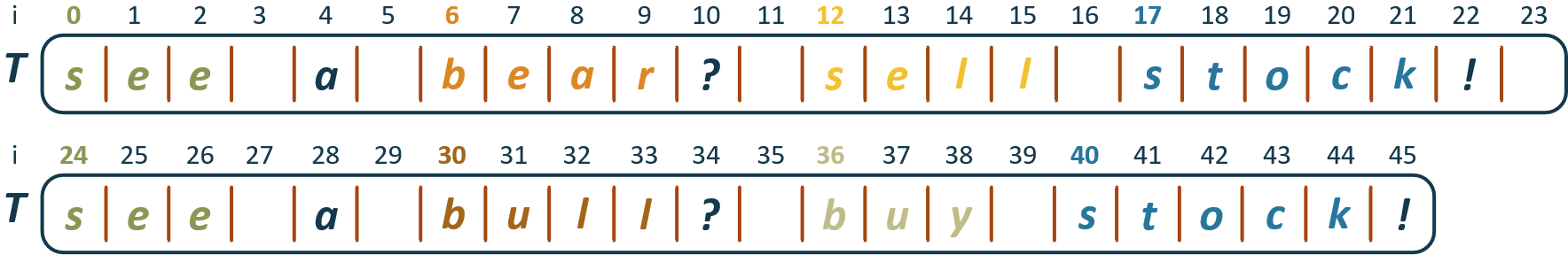
Durch Vorverarbeitung des Musters wird eine Geschwindigkeitsverbesserung beim Suchen erzielt. Nach Vorverarbeitung des Musters erzielt der KMP-Algorithmus eine Geschwindigkeit, die proportional zur Text-Grösse ist (Suchzeit unabhängig von Anzahl Wörtern). Ist der Text gross, unveränderlich und wird oft durchsucht (z.B. das Werk von Shakespeare), könnte man anstelle des Musters den Text vorverarbeiten.   
Ein Trie ist eine kompakte Datenstruktur für die Repräsentation einer Menge von Strings, wie z.B. alle Wörter eines Textes. Wird verwendet um z.B. Wörter mit demselben Anfangsbuchstaben zu finden.

### Standard Tries

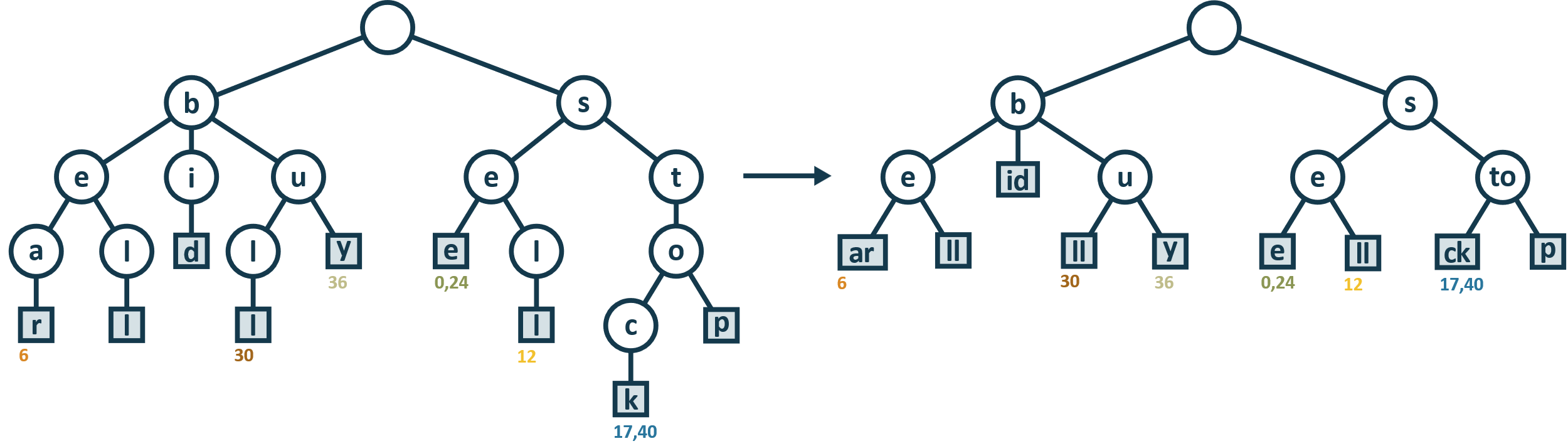
Ein Standard-Trie für eine Menge von Strings ist ein geordneter Baum, sodass jeder Knoten ausser dem Root ein Zeichen hat und die Pfade von den externen Knoten (Blättern) zur Wurzel die Strings von beinhalten. Ist alphabetisch sortiert und case-sensitive.

Beispiel eines Standard-Tries für die Menge von Strings  
 bear, bell, bid, bull, buy, sell, stock, stop

### Komprimierung

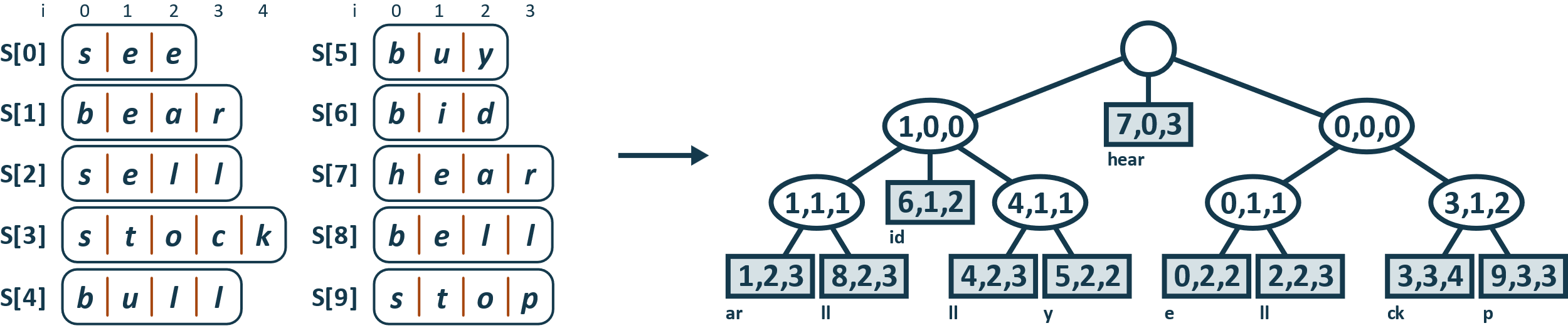
Einfügen der Wörter des Textes in einen Trie. Jeder Knoten speichert die Positionen des assoziierten Wortes im Text. Index des ersten Buchstaben des Wortes

Gibt es bei unkomprimierten Knoten keine Verzweigungen mehr, ist keine weitere Suche mehr nötig

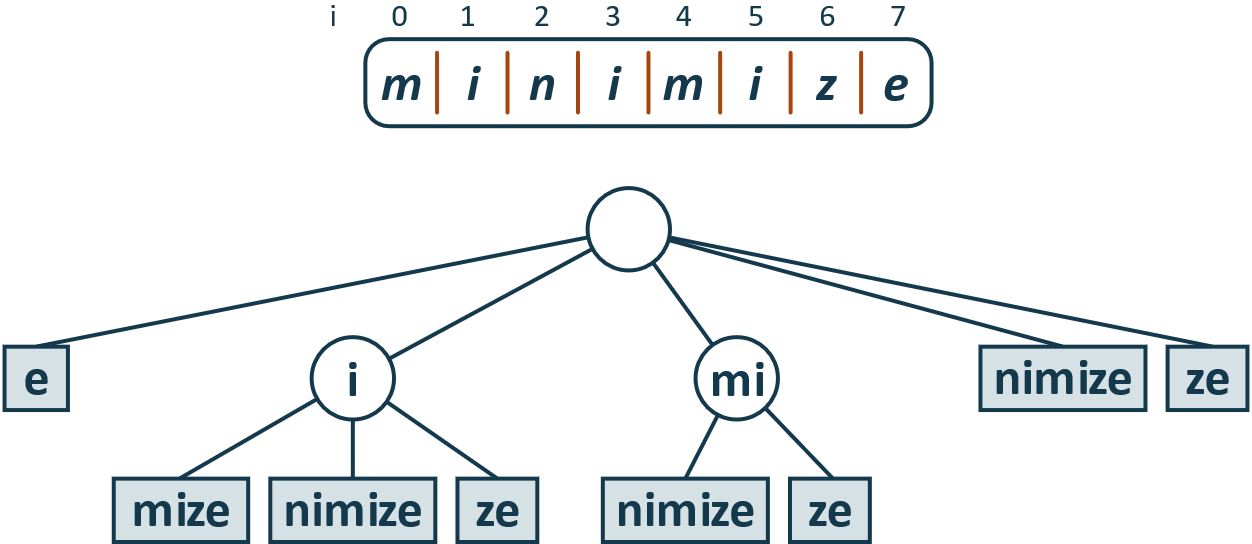
Ein komprimierter Trie wird von einem Standard-Trie hergeleitet durch Komprimierung von redundanten Knoten. Ein Knoten gilt als redundant, wenn es an ihm nur einen Pfad gibt.

* + 1. Kompakte Trie-Repräsentation als Array

Eine kompaktere Repräsentation eines komprimierten Tries für ein Array von String wird erreicht, in dem Knoten die Indizes anstelle von Substrings speichert. Benötigt Speicher, wobei die Anzahl Strings im Array ist. Dient als eine Hilfs-Index-Struktur.   
Erster Wert im Trie: Index , zweiter Wert: Anfang des Substrings, dritter Wert: Ende des Substrings in , [x,0,0] = Ein Buchstabe



### Suffix Trie

Der Suffix-Trie eines Strings ist der komprimierte Trie von allen Suffixen von . Kann neben dem Finden von Suffixen auch zum Finden von Substrings verwendet werden, indem man vom Root aus den Pfad wählt wo der erste Buchstabe des Substrings ist (falls vorhanden). Beispiel «nimi» → «nimize»

Eigenschaften für Suffix Trie für String der Länge

* Der Baum hat genau Blätter.
* Mit Ausnahme der Wurzel hat jeder interne Knoten mindestens zwei Kinder.
* Jeder Knoten ist mit einem nicht-leeren Substring von beschriftet.
* Keine zwei Kind-Knoten können Substrings haben, die mit demselben Zeichen beginnen.
* Der String, der sich aus der Verkettung aller Substrings ergibt, die auf dem Pfad von der Wurzel zum Blatt gefunden wurden, ergibt Suffix , für

## Zusammenfassung der Pattern-Matching-Algorithmen

|  |  |  |
| --- | --- | --- |
| Algorithmus | Zeitverhalten | Bemerkungen |
| **Brute-Force** |  | * Langsamste, aber simpelste Methode |
| **Boyer-Moore** |  | * Bei Textanalysen signifikant schneller als Brute-Force |
| **KMP** |  | * Geeignet für Strings mit vielen gleichen Buchstaben-Pattern |
| **Trie** | Grösse des Alphabets | * Gut, um Wörter zu finden, die mit dem gleichen Buchstaben beginnen * Suchzeit ist unabhängig von Anzahl Wörter |

## Dynamische Programmierung

Dynamische Programmierung ist ein generelles Algorithmen-Design-Paradigma. Problem in kleinere Subprobleme aufteilen und Lösung mithilfe der Ergebnisse dieser erhalten.

### Rucksack-Problem

Gegeben: Gegenstände mit einem bestimmten Gewicht und Wert. Ein Rucksack mit einer bestimmten Gewichts-Kapazität.f  
Gesucht: Füllung des Rucksacks, sodass der Wert der Gegenstände maximal ist.

* + 1. Brute-Force

Versuche alle möglichen Varianten. Betrachte nur jene Varianten, welche das maximale Gewicht nicht überschreiten. Nimm die beste Variante.  
Laufzeit: Anzahl ist exponentiell , also sehr schlecht. Die Reihenfolge des Durchprobierens spielt hier keine Rolle, ansonsten wäre die Laufzeit

|  |  |  |
| --- | --- | --- |
| Wert | Gewicht | W/G |
| **CHF** | **kg** |  |
| **CHF** | **kg** |  |
| **CHF** | **kg** |  |
| **CHF** | **kg** |  |

* + 1. Greedy

Nimm wiederholend den Gegenstand mit grösstem Verhältnis von Wert/Gewicht.  
Beispiel: Gegenstände, Rucksack-Kapazität: kgResultat: (kg) + (kg) = kg mit einem totalen Wert von . Das ist jedoch nicht optimal, besser wäre (kg) + (kg) = kg mit einem totalen Wert von . Also auch nicht optimal.

* + 1. Subprobleme

Das Problem wird in Subprobleme unterteilt. Konstruktion von optimalen Subproblemen «bottom-up»: Man beginnt mit kleinen Problemen und erhöht immer weiter, bis die gewünschte Länge erreicht ist. Probleme der Länge 1 sind einfach. Anschliessend Subprobleme der Längen 2, 3, … und so weiter.

Subprobleme beim Rucksack:

* Gegenstände

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Wert / kg, kg |  |  |  |  |  |
| **CHF kg (a)** |  |  |  |  |  |
| **CHF kg (b)** |  |  |  |  |  |
| **CHFkg (c)** |  |  |  |  |  |
| **CHF kg (d)** |  |  |  |  |  |

* kg maximales Gewicht

Subproblem: Was ist der grösstmögliche Wert für die ersten beiden Gegenstände und bei einer Gewichtslimite von kg?  
Lösung: grösstmöglicher Wert für dieses Subproblem: mit Gegenstand

Vorgehen zum Ausfüllen der Tabelle

Die Tabelle wird von oben links nach unten rechts ausgefüllt. Am besten geht man zeilenweise vor. In der ersten Zeile kann jeweils nur a in den Rucksack gepackt werden, darum immer 1. In der zweiten Zeile bleiben die ersten beiden Werte 1, bis mit 3kg in den Rucksack gepackt wird. Mit 4 kg können bei & in den Rucksack gepackt werden → Wert 5. Am Ende haben wir & mit einem Gesamtwert von 6 im Rucksack.

Anschliessend Pfad (gewählte Gegenstände) bestimmen: Grösstmöglicher Wert ist .

* Beginne mit dem grösstmöglichen Wert (rechts unten in der Tabelle) und gehe die Spalte nach oben, bis sich der Wert ändert (Im Beispiel von 6 auf 5, zu ). Entsprechender Gegenstand (c) gehört in den Rucksack und das Feld ist Teil des Pfades. Nun muss das Gewicht dieses Gegenstandes vom Gewicht des Rucksacks (dieser Spalte) und der Wert des Gegenstandes vom Gesamtwert abgezogen werden, neue Werte: und → nächster Gegenstand bei Wert 1 und Gewicht 1.
* Fahre mit dem nächsten Subproblem weiter: Gewicht ist neu 1kg und Wert 1. Wo treffen sich diese Zeilen/Spalten? Bei a & 1kg. a gehört ebenfalls in den Rucksack und das Feld gehört zum Pfad.

### Technik der dynamischen Programmierung

Anwendbar auf Probleme, welche anfänglich eine sehr grosse Laufzeit zu benötigen scheinen. Voraussetzung:

* Einfach Subprobleme: die Subprobleme können durch wenige Variablen ausgedrückt werden   
  (z.B. )
* Subproblem-Optimierung: Das globale Optimum kann durch optimale Subprobleme ausgedrückt werden (man kann durch Zusammenführen von Subproblemen das beste Resultat erhalten)
* Subprobleme überlappen: Die Subprobleme sind nicht unabhängig, sie überlappen   
  (sollte mit bottom-up konstruiert werden)

### Längste gemeinsame Subsequenz (Longest Common Subsequence LCS)

Eine Subsequenz eines Character-Strings ist ein String der Form , wobei . Die Reihenfolge ist gleich wie beim Hauptstring, die Buchstaben müssen aber nicht direkt aufeinander folgen, ist also nicht dasselbe wie ein Substring. Beispiel: **A**BC**D**E**FGH**IJ**K**. Subsequenz: **DFGHK**, nicht Subsequenz: **DAGH**

* + 1. **Longest Common Subsequenz (LCS)**

Gegeben sind zwei Strings und . Finde die längste Subsequenz welche in und in enthalten ist.  
Beispiel: **A**B**CD**E**FG**, Z**AC**K**DF**W**G**H, LCS **ACDFG**.  
Anwendung: z.B. bei Vergleichen von DNA oder Source-Files mit Differenzen.

* + 1. Brute-Force

Aufzählung aller Subsequenzen von , testen, welche ebenfalls Subsequenzen von sind. Die längste Subsequenz als Resultat wählen. Sehr langsam, da .

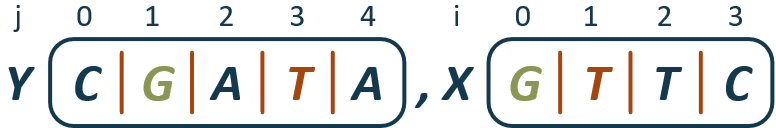
* + 1. LCS-Algorithmus

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | C | **G** | A | **T** | A | … |  |
|  |  | -1 | 0 | 1 | 2 | 3 | 4 |  | Index |
|  | -1 |  | 0 | 0 | 0 | 0 | 0 |  |  |
| **G** | 0 | 0 | 0 | **1** | **1** | **1** | **1** |  |  |
| **T** | 1 | 0 | 0 | **1** | **1** | **2** | **2** |  |  |
| T | 2 | 0 | 0 | **1** | **1** | **2** | **2** |  |  |
| C | 3 | 0 | 0 | **1** | **1** | **2** | **2** |  |  |
| … |  |  |  |  |  |  |  |  |  |
|  | Index |  |  |  |  |  |  |  |  |

Definiere als Länge der längsten gemeinsamen Subsequenz von und . Erlaube als Index, sodass und um auszudrücken, dass der null Teil von oder (leeres Wort) keine Übereinstimmung hat mit dem anderen. Definiere folgendermassen:

* Wenn (Übereinstimmung), dann
* Wenn (keine Übereinstimmung), dann

Beispiel:



Bei ist , darum werden jetzt der Wert oberhalb () und links davon () überprüft und der grössere davon (1) als neuer Wert eingetragen. Bei ] ist , also wird der Wert diagonal links oberhalb mit 1 addiert () und eingetragen.

Es sind meistens mehrere Lösungen möglich, diese haben aber immer die gleiche Länge.

**Ein Bild, das Text, Reihe, Diagramm, Schrift enthält.

Automatisch generierte BeschreibungAlgorithm LCS(X,Y)**

**Input:** Strings X and Y with n and m elements

**Output:** for i=0,…,n-1 and j=0,…,m-1, the length L[i,j] of a longest String that is a subsequence of both the string X[0..i] and Y[0..j]

**for** i = 1 to n-1 **do**  
 L[i,-1] = 0

**for** j = 0 to m-1 **do**

L[-1, j] = 0

**for** i = 0 to n-1 **do**

**for** j = 0 to m-1 **do**

**if** xi = yj **then**

L[i,j] = L[i-1,j-1]+1

else

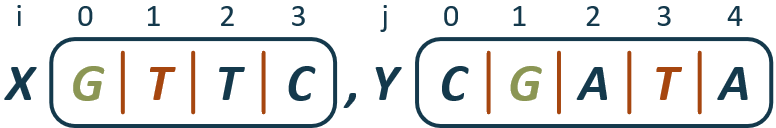
L[i,j] = max{L[i-1,j], L[i,j-1]}

**return** array L

Auslesen der Longest Common Subsequence

Die Lösung ist in enthalten. Die gesuchte Subsequenz kann von der L-Tabelle ausgelesen werden. Beginne am Ende: . Wenn das Zeichen für und gleich, füge in ein. Wechsle auf Position . Ansonsten: wie beim Erstellen der Tabelle . Rand um gleiche Zahlen einzeichnen, Ecken dieser Rahmen sind Pfadkanten.

Beispiel:

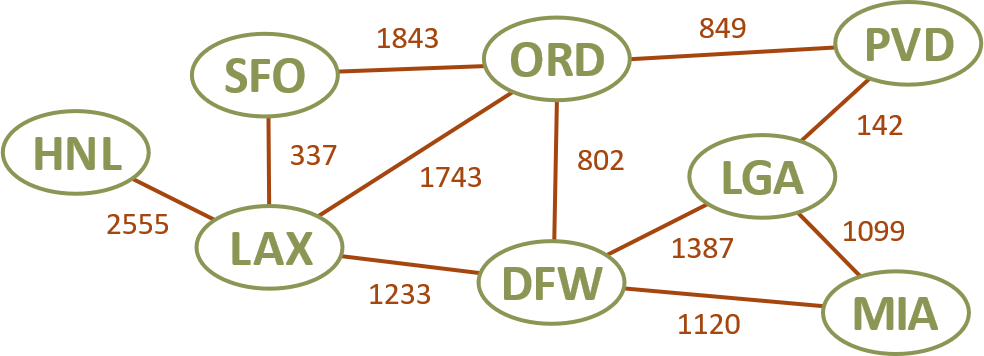


Wir starten mit . Die Zeichen matchen nicht, darum den Wert oberhalb () und links davon () überprüft. Da beide gleich gross sind, ist es egal, ob wir die Zeile oder Spalte wechseln. Hier gehen wir zu und wiederholen das Ganze. Da grösser ist, wechseln wir dahin. Bei haben wir zum ersten Mal einen Match, packen darum zuhinterst in und gehen zu . Am Ende haben wir .

## Zusammenfassung dynamische Programmierung

|  |  |  |
| --- | --- | --- |
| Algorithmus | Zeitverhalten | Bemerkungen |
| **Knapsack: Brute-Force** |  | * Sehr schlecht weil exponentiell |
| **Knapsack: Greedy** |  | * Ergibt nicht immer optimale Lösung |
| **Knapsack: Subprobleme** | Anzahl Gegenstände, = Anzahl Gewichte |  |
| **LCS: Brute-Force** |  | * Sehr schlecht weil exponentiell |
| **LCS-Algorithmus** | (äusserer und innerer Loop) | * Gibt nur die Länge des Substrings aus, um den Substring selbst zu erhalten, muss der Algorithmus «rückwärts» ausgeführt werden. * Mehrere Lösungen möglich |

## Graphen

Ein Graph ist ein Paar , wobei ein Set von **Vertizes** (Knoten) ist und eine Collection von Vertizes-Paaren, also **Kanten** (Edge). Vertizes und Kanten sind Positionen und speichern Elemente.

Beispiel: Ein ***Vertex*** repräsentiert ein ***Flughafen*** und speichert den Flughafen-Code. Eine ***Kante*** repräsentiert eine ***Flugroute*** zwischen zwei Flughäfen und speichert die Distanz der Route.

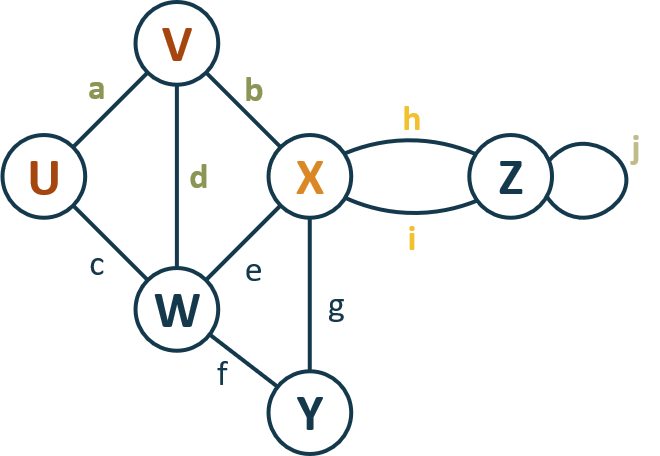
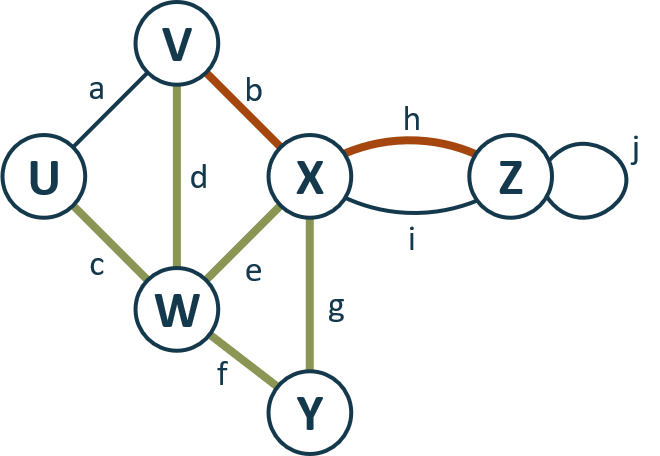
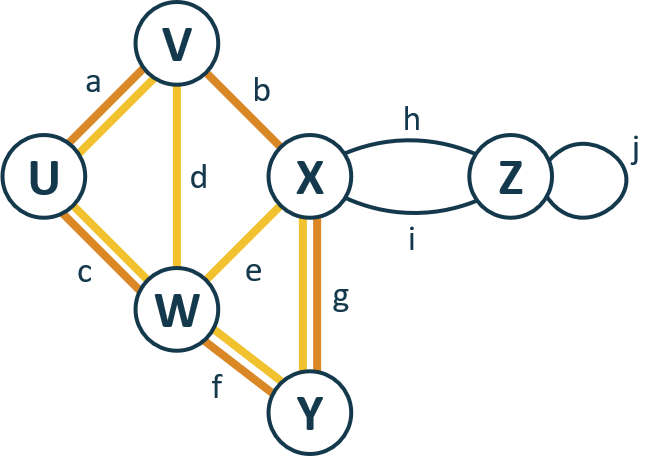
### Kanten-Typen

* Gerichtete Kanten: Geordnetes Paar von Vertizes . Der erste Vertex entspricht dem Ursprung, der zweite Vertex entspricht dem Ziel. Beispiel: ein Flug
* Ungerichtete Kanten: Ungeordnetes Vertizes-Paar . Beispiel: Flugroute
* Gerichteter Graph: Alle Kanten sind gerichtet. Beispiel: Flugplan
* Ungerichteter Graph: Alle Kanten sind ungerichtet. Beispiel: Flugrouten-Plan

### Anwendungen

* Elektronische Schaltungen: Printed Circuit Board, Integrated Circuit
* Transport-Netzwerke: Autobahnnetz, Flugnetz
* Computer Netzwerke: LAN, Internet, Web
* Datenbanken: Entity-Relationship Diagramm

### Terminologie

* Kanten sind inzident, wenn sie am gleichen Vertex enden.   
  **a, d und b sind inzident in V.**
* Adjazente sind benachbarte Vertizes. **U und V sind adjazent.**
* Grad (Degree) eines Vertex ist die Anzahl inzidenter Kanten (Anzahl ausgehender Kanten). **X besitzt Grad 5.**
* Parallele Kanten sind Kanten, die die gleiche Vertizes verbinden.   
  **h und i sind parallele Kanten.**
* Schleifen sind Kanten, die zum gleichen Vertex zurückführen. **j ist eine Schleife.**
* Pfad: Sequenz von alternierenden Vertizes und Kanten. Beginnt und endet mit einem Vertex. Jede Kante beginnt und endet an einem Ihrer Endpunkte.
* Einfacher Pfad: Ein Pfad dessen Vertizes und Kanten alle unterschiedlich sind (Keine Schleifen im Pfad).
* Nicht einfacher Pfad: Ein Pfad, in dem ein Vertex oder eine Kante mehrfach passiert wird.
* Zyklus: Zirkuläre Sequenz - Start und Ziel sind gleich.
* Einfacher Zyklus: Ein Zyklus, sodass alle seine Vertizes und Kanten unterschiedlich sind.
* Nicht einfacher Zyklus: Ein Zyklus, in dem nicht alle Vertizes und Kanten unterschiedlich sind.

### Eigenschaften

Notation: Anzahl Vertizes, Anzahl Kanten, Grad von Vertex

* + 1. Eigenschaft 1

Der Grad eines Vertex ist gleich zweimaldie Anzahl Kanten. .

Beweis

Jede Kante wird zweimal gezählt, weil sie an zwei Vertizes ankommt.  
Beispiel:

* + 1. Eigenschaft 2

In einem ungerichteten, einfachen Graphen (ohne Schleifen und ohne parallele Kanten) gilt: . Bei einem voll vermaschten Graphen (jeder Vertex ist mit jedem anderen Vertex verbunden) ist .

Beweis

Jeder Vertex besitzt ein Grad von höchstens .   
Beginne bei einem Vertex die Kanten zu zählen. Wie viele Kanten gibt es in einem voll vermaschten Graphen? Zu allen Vertizes ausser sich selbst, also Wie viele Kanten bleiben noch übrig vom nächsten Vertex zu den restlichen? Zu allen ausser sich selbst und zum ersten Vertex, also . Daraus ergibt sich für einen voll vermaschten Graph mit 4 Vertizes:

* + 1. Eigenschaft 3

Bei einem ungerichteten, einfachen Graphen ist die Anzahl der Knoten mit ungeradem Grad gerade.

Beweis

Da jede Kante zwei Enden hat, muss die Summe der Grade aller Knoten in einem Graphen gerade sein. Ausserdem wird beim Hinzufügen einer Kante immer der Grad von 2 Knoten gleichzeitig geändert.   
Es muss also eine gerade Anzahl Knoten mit ungeradem Grad existieren.

* + 1. Eigenschaft 4

Bei einem einfach verbundenen Graphen mit Knoten und Kanten ist .

Beweis

Die Anzahl Kanten hängt direkt von der Anzahl Knoten ab. Es kann eine untere sowie eine obere Schranke für gefunden werden:

* Untere Schranke : (Jeder Vertex ist nur mit einem anderen verbunden - Liste)
* Obere Schranke : (Voll vermaschter Graph)

Somit muss bewiesen werden, dass

Beweis von Konstanten und niederwertige Terme streichen ()

Beweis von Konstanten streichen ()

### Haupt-Methoden des Graph ADT (Abstract Data Type)

Vertizes (, ) und Kanten () sind Positionen und speichern Elemente.

Zugriffs-Methoden

* **endVertices(e):** Ein Array der zwei End-Vertizes der Kante
* **opposite(v,e):** Der Vertex gegenüber von entlang der Kante
* **areAdjacent(v,w):** True falls und aneinander angrenzen
* **replace(v,x):** Ersetzt das Element bzw. den Wert bei Vertex mit
* **replace(e,x):** Ersetzt das Element bzw. den Wert an Kante mit

Update-Methoden

* **insertVertex(o):** Fügt einen Vertex mit dem Element ein, gibt den neuen Vertex zurück
* **insertEdge(v,w,o):** Fügt eine neue Kante zwischen und ein, welches das Element beinhaltet, gibt die neue Kante zurück
* **removeVertex(v):** Entfernt den Vertex und seine angrenzenden Kanten
* **removeEdge(e):** Entfernt die Kante

Iterator-Methoden

* **incidentEdges(v):** Gibt an angrenzende Kanten zurück
* **vertices():** Gibt alle Vertizes im Graph zurück
* **edges():** Gibt alle Kanten im Graph zurück

### Kanten-Listen Struktur

Wird so nicht wirklich eingesetzt.

* Vertex / Knoten Objekt: Referenz auf die Position in der Vertex-Sequenz, Element (Wert des Vertex)
* Kanten-Objekt: Referenz auf die Position in der Kanten-Sequenz , Ursprungs-Vertex Objekt (Vertex links der Kante), Ziel-Vertex Objekt (Vertex rechts der Kante), Element (Wert der Kante).
* Vertex-Sequenz: Sequenz der Vertex-Objekte (Linked List aller Vertizes)
* Kanten-Sequenz: Sequenz von Kanten-Objekten (Linked List aller Kanten)
  + 1. Remove-Operationen

RemoveEdge()

Es muss nur der next-Eintrag vor der zu entfernenden Kante in der Kanten-Sequenz angepasst werden (der zu entfernende Eintrag wird «übersprungen»), das Kanten-Objekt wird durch den Garbage-Collector entfernt, da keine Referenz mehr besteht. Laufzeit:

RemoveVertex()

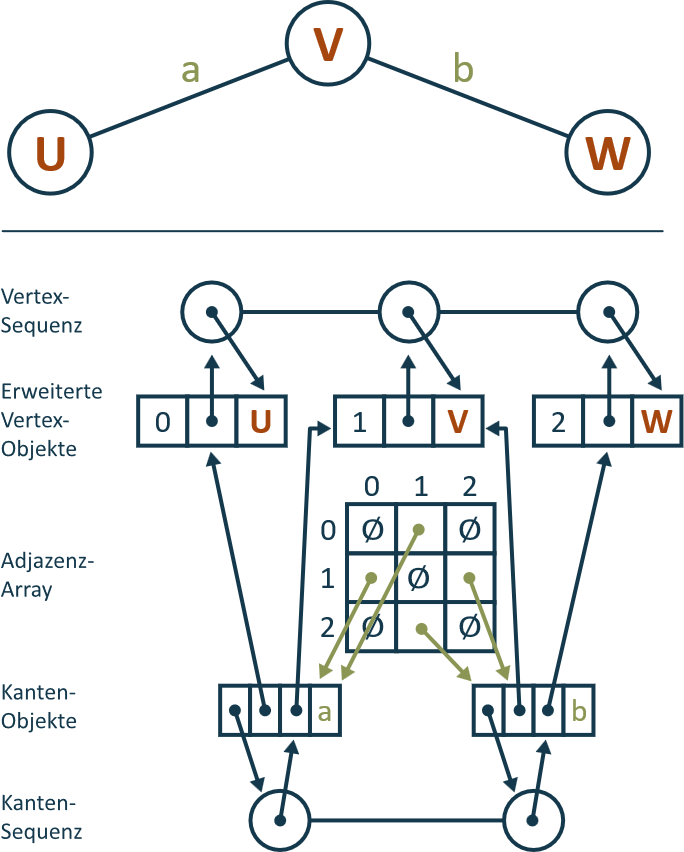
Da wir die Kanten am Vertex auch löschen müssen und kein Direktverweis vom Vertex zu den Kanten existiert (weil die Referenzen nur von der Kante zum Vertex zeigen), wird incidentEdges() benötigt. Mit dieser Funktion muss jede Kante durchgegangen und überprüft werden, ob sie auf den zu löschenden Vertex zeigt. Wegen dieser Iteration ist die Laufzeit .

### Adjazenz-Listen Struktur

Erweiterung der Kanten-Listen Struktur um eine Inzidenz-Sequenz für jeden Vertex und erweiterte Kanten-Objekte.

***Indizenz-Sequenz:*** Sequenz der Positionen auf Kantenobjekte der inzidenten Kanten (Liste der Kanten eines Vertex).  
***Erweiterte Kanten-Objekte:*** Referenziert auf die assoziierten Positionen in der Inzidenzsequenz der Endvertizes. (Zwei Pointer von der Kante zu der Inzidenz-Sequenz und von da zum anliegenden Vertex)

Durch diese Erweiterung der Referenzen vom Vertex zu seinen Kanten wird sichergestellt, dass removeVertex() ebenfalls eine Laufzeit von hat, da nicht mehr durch alle Kanten iteriert werden muss.   
Müssen in beide Richtungen separate Pointer sein, weil eine Kante immer zwei Vertizes hat, aber Vertizes nicht zwingend zwei anliegende Kanten besitzen.

Die Adjazenz-Listen Struktur benötigt also 3 Linked Lists: Liste aller Vertizes (Vertex-Sequenz), Liste aller Kanten (Kanten-Sequenz) und Liste aller Kantenlisten der Vertizes (Inzidenz-Sequenz für Vertizes).

### Adjazenz-Matrix Struktur

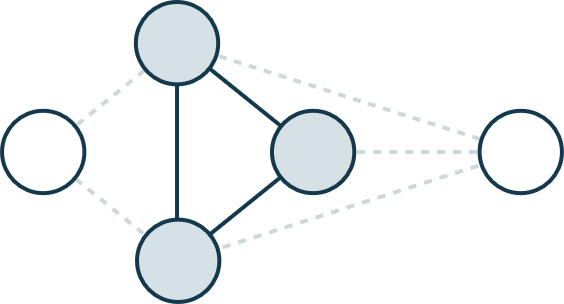
Erweitert die Kanten-Listen Struktur um einen 2D-Array (Adjazenz-Array).

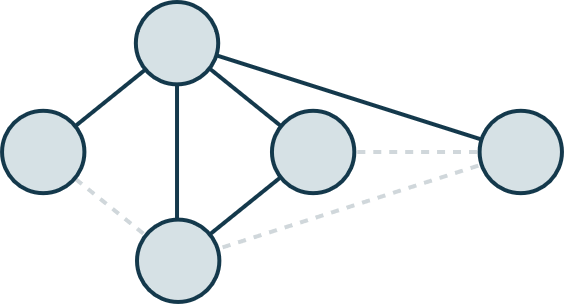
Die Vertex-Objekte sind erweitert um einen Integer-Key (Index), welcher mit dem Vertex assoziiert wird.

Der Adjazenz-Array referenziert auf die Kantenobjekte für adjazente Vertizes. null () für nichtadjazente Vertizes.

Bei ungerichteten Graphen ist die Matrix achsensymmetrisch. Die Dimension der Matrix ändert sich nur beim Hinzufügen von Vertizes, nicht jedoch beim Hinzufügen von Kanten.

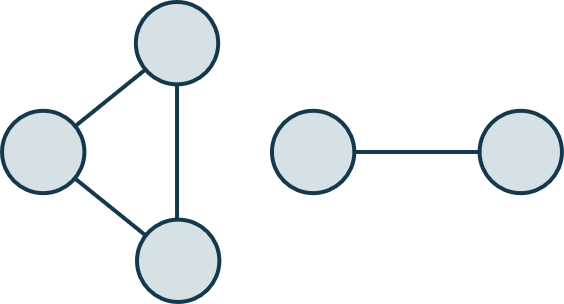
### Subgraphen

Ein Subgraph eines Graphen ist ein Graph, bei dem die Kanten von eine Teilmenge der Kanten von sind und die Vertizes von eine Teilmenge der Vertizes von sind.

Ein spanning (aufspannender) Subgraph des Graphen ist ein Subgraph, welcher alle Vertizes, aber nicht alle Kanten von G enthält.

### Connectivity

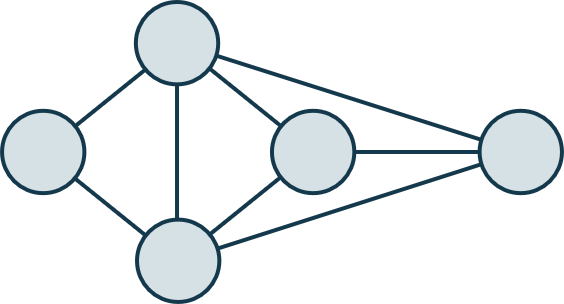
Ein Graph heisst verbunden (connected), falls zwischen jedem Paar von Vertizes ein Pfad existiert. Eine verbundene Komponente eines Graphen ist ein verbundener Subgraph von .

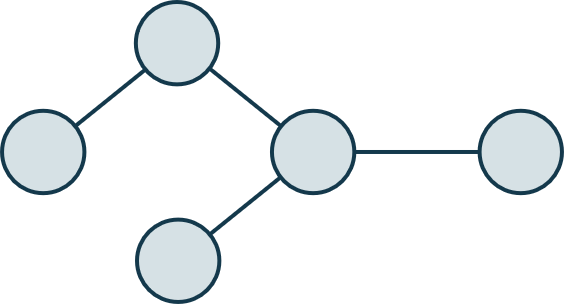
Es ist nicht möglich, einen Graphen zu erstellen, der während dem Aufbau immer verbunden ist, da eine Kante erst erstellt werden kann, wenn beide dazugehörende Vertizes existieren.

### Bäume und Wälder

Ein (freier) Baum ist ein ungerichteter Graph , sodass verbunden ist und keine Zyklen aufweist. Anders als beim Wurzelbaum gibt es keinen Root-Knoten.  
Ein Wald ist ein ungerichteter Graph ohne Zyklen. Die verbundenen Komponenten eines Waldes sind Bäume. Im Bild: 4 Bäume, die zusammen einen Wald ergeben.

### Spanning Trees und Wälder

Ein aufspannender Baum (Spanning Tree) eines verbundenen Graphen ist ein aufspannender Subgraph, welcher auch ein Baum ist (also alle Knoten können ungerichtet erreicht werden und Subgraph hat keine Zyklen). Ein aufspannender Baum ist nicht eindeutig, ausser der Graph, von dem ausgegangen wird, ist ein Baum. (Es können mehrere Spanning Trees für einen Subgraph erstellt werden, ausser dieser ist bereits Zyklenfrei → Baum)

Aufspannende Bäume werden beispielsweise in Kommunikationsnetzwerken eingesetzt. Ein aufspannender Wald eines Graphen ist ein aufspannender Subgraph, welcher auch ein Wald ist (mehrere Bäume, aus denen Spanning Trees gemacht wurden).

## Zusammenfassung PErformance Graphen

|  |  |  |  |
| --- | --- | --- | --- |
| Vertizes, Kanten, keine Parallelen Kanten, keine Schleifen | Kanten Liste | Adjazenz-Liste | Adjazenz Matrix |
| ***Platzverbrauch*** |  | eigentlich |  |
| incidentEdges(v) |  |  |  |
| areAdjacent(v,w) |  |  |  |
| insertVertex(o) |  |  |  |
| insertEdge(v,w,o) |  |  |  |
| removeVertex(v) |  |  |  |
| removeEdge(e) |  |  |  |

## Ungerichtete Graphen Traversierung

### Ungerichtete Tiefensuche / Undirected Depth-First Search

Eine DFS-Traversierung eines ungerichteten Graphen

* Besucht alle Vertizes und Kanten von
* Bestimmt, ob verbunden ist
* Berechnet / bestimmt die verbundenen Komponenten von
* Berechnet einen aufspannenden Wald von

DFS lässt sich erweitern, um andere Graphenprobleme zu lösen: Finden und Ausgeben eines Pfades zwischen zwei gegebenen Vertizes und finden von Zyklen in Graphen. Die Tiefensuche entspricht in etwa der Euler-Tour bei binären Bäumen.

* + 1. Funktionsweise

DFS vergibt allen Kanten und Vertizes ein Label. Zuerst werden alle Komponenten auf **UNEXPLORED** gesetzt. Gelangt DFS zu einem neuen Vertex, wird dieser als **VISITED** gekennzeichnet. Dann werden alle Kanten dieses Vertex mit incidentEdges() geholt und mit der «kleinsten» **UNEXPLORED** Kante begonnen.  
Ist der Vertex gegenüber ebenfalls **UNEXPLORED**, wird die Kante auf DISCOVERY gesetzt, zum adjazenten Vertex gegangen und DFS rekursiv aufgerufen. Falls der Vertex bereits **VISITED** ist, wird die Kante auf **BACK** gesetzt und mit der nächsten Kante fortgefahren.

**Algorithm DFS(G)** // main function

**Input** graph G

**Output** lableing of the edges of G as discovery edges and back edges

**for all** u G.vertices(): setLabel(u, **UNEXPLORED**)

**for all** e G.edges(): setLabel(e, **UNEXPLORED**)

**for all** v G.vertices():

if getLabel(v) = **UNEXPLORED**: **DFS(G,v)**

**Algorithm DFS(G,v)** // recursive function

**Input** graph G and a start vertex v of G

**Output** labeling of the edges of G in the connected component of v as discovery edges   
 and back edges

setLabel(v, **VISITED**)

**for all** e G.incidentEdges(v)

**if** getLabel(e) = **UNEXPLORED**

w opposite(v,e)

**if** getLabel(w) = **UNEXPLORED**

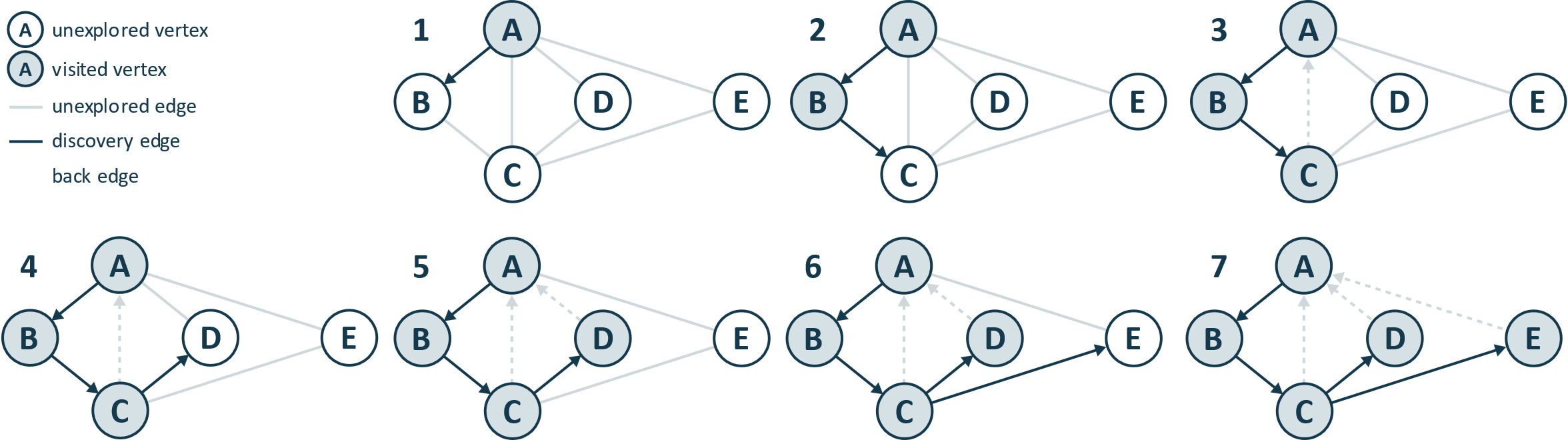
setLabel(e, **DISCOVERY**)

**DFS(G,w)** // recursive call

**else**

setLabel(e,**BACK**)

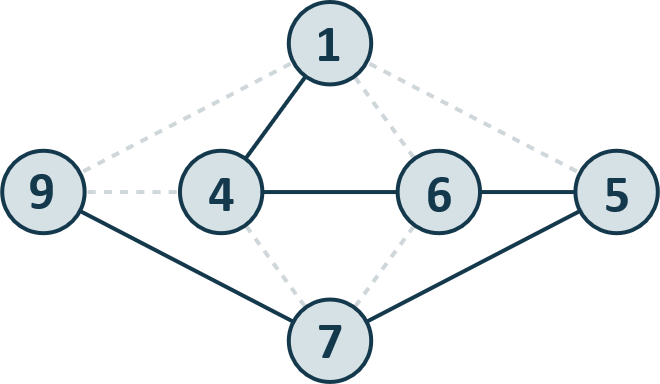
Die **DISCOVERY** Edges bilden den aufspannenden Baum. Am Schluss sind alle Edges entweder auf **DISCOVERY** oder **BACK**. **DISCOVERY** gehört zum Baum, **BACK** gehört nicht zum Baum.



* + 1. DFS und Labyrinth

Der DFS-Algorithmus ähnelt der klassischen Strategie zur Erkundung eines Labyrinths. Wir markieren jede besuchte Kreuzung, Ecke und Sackgasse (Vertex), wir markieren jeden besuchten Korridor (Kante), und wir notieren den Rückweg zum Eingang (Start Vertex, Rekursion).

* + 1. Eigenschaften von DFS
* **DFS(G,v)** besucht alle Vertizes und Kanten in der verbundenen Komponente von beginnend bei
* Die von **DFS(G,v)** markierten, besuchten Kanten bilden einen aufspannenden Baum für die verbundene Komponente von beginnend bei .
  + 1. Pfade finden

Der DFS kann darauf spezialisiert werden, einen Pfad zwischen zwei gegebenen Vertizes und zu finden. Zuerst wird **pathDFS(G,u)** mit als Startvertex aufgerufen. Mithilfe eines Stacks wird der Pfad zwischen dem Startvertex und dem aktuellen Vertex gespeichert. Sobald der Zielvertex gefunden wurde, wird der Pfad mithilfe des Stacks ausgegeben.

**Algorithm pathDFS(G,v,z)**

setLabel(v,**VISITED**)

S.push(v) // add vertex to path

**if** v = z: finish: result is **S.elements()**

**for all** e G.incidentEdges(v):

**if** getLabel(e) = **UNEXPLORED**: w opposite(v,e)

**if** getLabel(w) = **UNEXPLORED**

setLabel(e, **DISCOVERY**)

S.push(e) // add edge to path

**pathDFS(G,w,z)** // recursion

S.pop() // remove edge after returning from recursion, not in path

**else**

setLabel(e, **BACK**)

S.pop() // remove vertex from path after trying all its edges, not in path

* + 1. Zyklen finden

Der DFS kann auch darauf spezialisiert werden, einfache Zyklen zu finden. Mithilfe eines Stacks wird der Pfad zwischen dem Startvertex und dem aktuellen Vertex gespeichert. Sobald ein Back-Edge(v,w) angetroffen wird, wird der Zyklus als Teil des Stacks ausgegeben: vom obersten Element bis zum Vertex .

**Algorithm cycleDFS(G,v)**

setLabel(v,VISITED)

S.push(v)

**for all** e G.incidentEdges(v)

**if** getLabel(e) = **UNEXPLORED**

w opposite(v,e)

S.push(e)

**if** getLabel(w) = **UNEXPLORED**

setLabel(e, **DISCOVERY**)

**cycleDFS(G,w)** // recursion call

**S.pop()**

else // back edge found

T new empty stack

**repeat**

o S.pop()

T.push(o) // remove last element from stack and add to return path

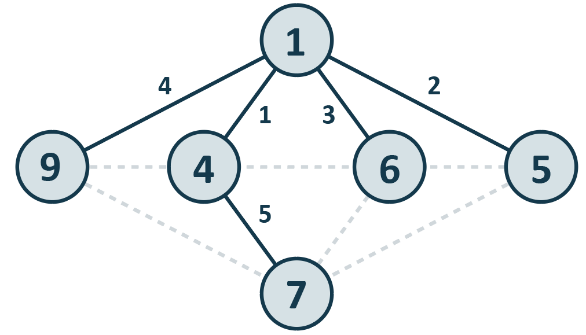
until o = w // opposite of the back edge is reached

finish: result is T.elements() // path from start vertex to cycle

S.pop()

### Breitensuche / Breadth-First Search

Eine BFS-Traversierung eines Graphen (gleiche Eigenschaften wie DFS)

* Besucht alle Vertizes und Kanten von
* Bestimmt, ob verbunden ist
* Berechnet / bestimmt die verbundenen Komponenten von
* Berechnet einen aufspannenden Wald von

BFS lässt sich erweitern, um andere Graphenprobleme zu lösen: Finden und Ausgeben eines Pfades mit einer minimalen Anzahl Kanten (Spanning Tree) zwischen zwei gegebenen Vertizes (DFS gibt irgendeinen Pfad aus, BFS den kürzesten), Finden von einfachen Zyklen in Graphen.

* + 1. Funktionsweise

BFS geht der Reihe nach alle gegenüberliegenden Vertizes vom Startvertex durch und baut sich eine Liste von Listen auf. In befindet sich nur der Startvertex. Nun werden alle Vertizes, die mit dem Startvertex verbunden sind, in eingefügt. Der Index der Listen gibt an, wie viele Kanten dieser Vertex vom Startvertex entfernt ist (alle Vertizes in sind 3 Kanten vom Startvertex entfernt). In befindet sich nur der Startvertex. Der Algorithmus terminiert, nachdem alle Vertizes einer Liste besucht wurden und es keine weiteren in gibt.

Wie DFS verwendet BFS Labels an den Kanten und Vertizes. Allerdings gibt es keine Back Edges mehr, sondern nur noch Cross Edges. Diese bezeichnen eine Kante zu einem bereits besuchten Knoten auf derselben oder höheren Liste. Back Edges werden nicht mehr benötigt, da man ja nicht zurück zum Startknoten schaut, sondern nur von ihm weg.

**Algorithm BFS(G)** // main function

**Input** graph G

**Output** labeling of the edges and partition of the vertices of G

**for all** u G.vertices(): setLabel(u, **UNEXPLORED**)

**for all** e G.edges(): setLabel(e, **UNEXPLORED**)

**for all** v G.vertices():

**if** getLabel(v) = **UNEXPLORED**: **BFS(G,v)**

**Algorithm BFS(G,s)** // for all edges of a vertice, \*not\* recursive

L0 new empty sequence

L0.insertLast(s)

setLabel(s, VISITED)

i 0

**while** !Li.isEmpty(): Li+1 new empty sequence // run while there are verticies at this level

**for all** v Li.elements()

**for all** e G.incidentEdges(v)

**if** getLabel(e) = **UNEXPLORED**

w opposite(v,e)

**if** getLabel(w) = **UNEXPLORED**

setLabel(e, **DISCOVERY**)

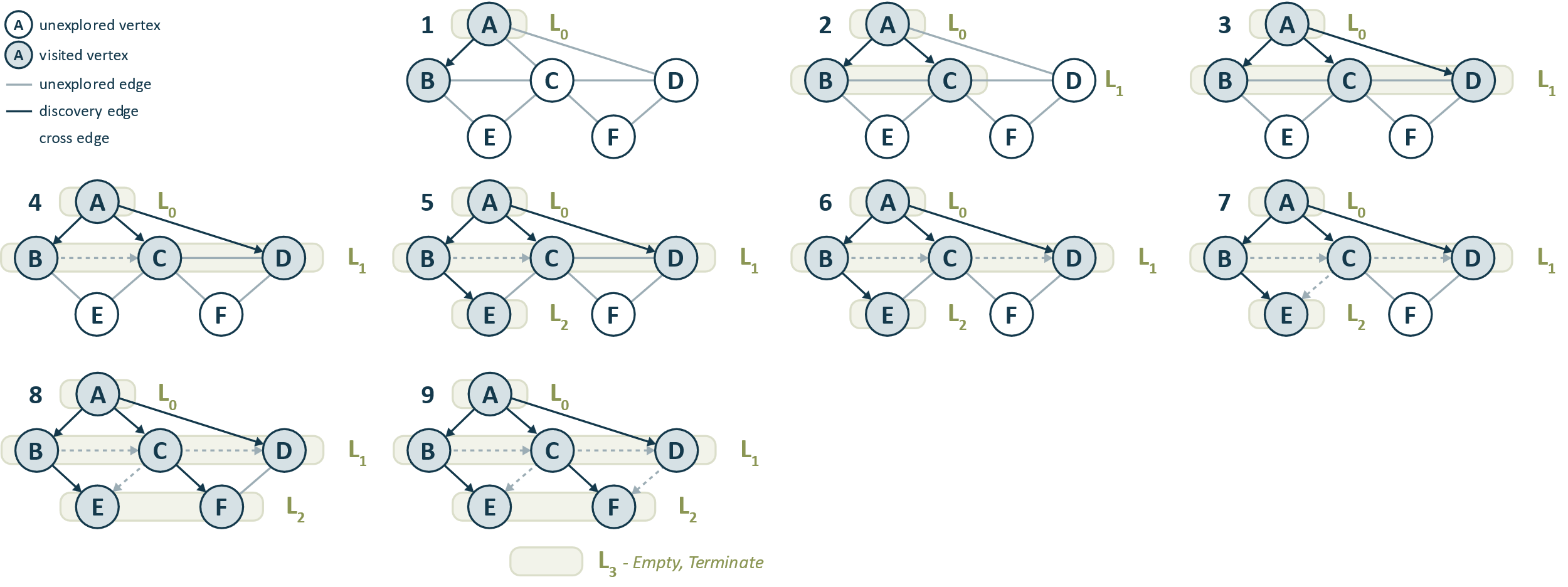
setLabel(w, **VISITED**)

Li+1.insertLast(w) // add unexplored vertice to next higher list

**else**

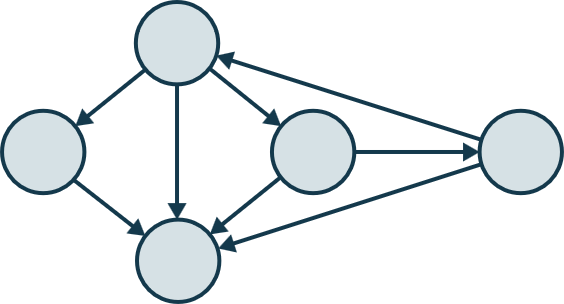
setLabel(e, **CROSS**)

i i+1



* + 1. Eigenschaften von BFS
* : verbundene Komponente von (: Start-Vertex)
* **BFS(G,s)** besucht alle Vertizes und Kanten in
* Die Discovery-Kanten von **BFS(G,s)** bilden einen aufspannenden Baum von (gleich wie bei DFS)
* Für jeden Vertex von gilt:
  + Der Pfad in von nach besitzt Kanten
  + jeder Pfad von nach in besitzt mindestens Kanten (minimaler oder längerer Pfad).

## Gerichtete Graphen / Directed Graphs

Ein gerichteter Graph ist ein Graph, dessen Kanten alle gerichtet sind, sie können nur in eine Richtung traversiert werden.  
Anwendungen: Einbahnstrassen, Flüge, Task Scheduling (Task a muss terminieren, bevor Task b gestartet werden kann).

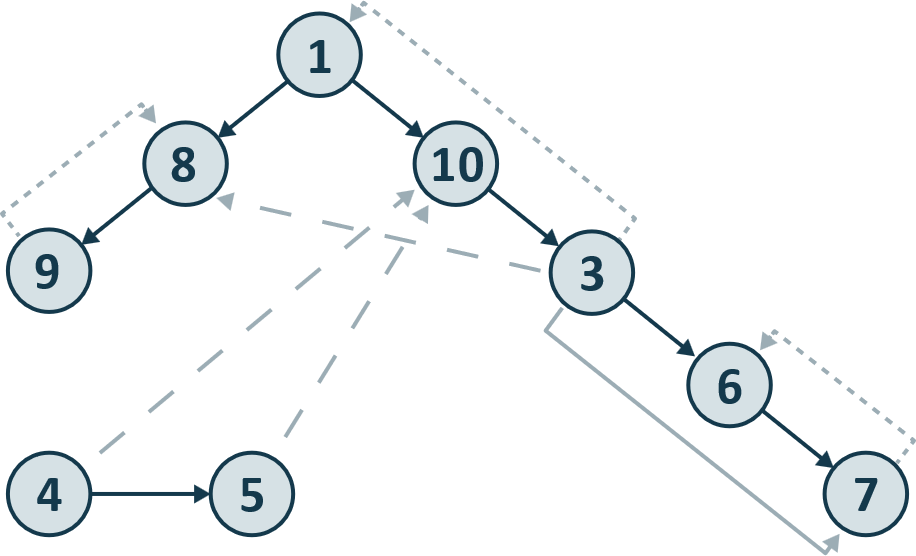
### Eigenschaften

* Ein Digraph ist: derart, dass jede Kante nur in eine Richtung geht. Kante geht von nach , aber nicht von nach .
* Wenn einfach (maximal eine Kante von & zu demselben Vertex) ist (ohne /2 wie bei ungerichtet, weil Kanten nicht mehr doppelt gezählt werden)
* Wenn In- und Out-Kanten in separaten Adjazenz-Listen sind (es gibt in der Adjazenz Liste zwei Inzidenz-Sequenz-Listen für Start/Ende einer Kante): Laufzeit für Zugriff auf In- und Out-Kanten proportional zur Grösse der Listen

### 

### Gerichtete Tiefensuche / Directed Depth-First Search

Die Traversierungs-Algorithmen DFS und BFS können für gerichtete Graphen spezialisiert werden, indem Kanten nur entlang ihrer Richtung traversiert werden. Im gerichteten DFS-Algorithmus gibt es vier Typen von Kanten.

* Baumkanten (discovery): Kante des Waldes 
* Rückkanten (back): Verbindung zu einem besuchten Vorgänger im gleichen Ast 
* Vorwärtskanten (forward): Verbindung zu einem nicht besuchten Nachfolger im Ast 
* Kreuzungskanten (cross): Kanten, die von einem Ast zum anderen wechseln (alle übrigen Kanten) 

Beim Bestimmen ob Vorwärts/Rückwärtskante ist der Wert der Vertizes entscheidend (). Deshalb hilft es, den Graphen als Baum zu zeichnen.

### Erreichbarkeit / Connectivity

Je nachdem, welcher Vertex als Startpunkt gewählt wird, können eventuell nicht alle Vertizes traversiert werden. Strong Connectivity (streng verbunden) bedeutet, dass jeder Vertex alle anderen Vertizes erreichen kann.

Überprüfungsalgorithmus: Ist Strong Connected?

* Wähle einen Vertex in . Führe eine Tiefensuche von in durch.
* Wenn es einen nicht besuchten Vertex gibt: return false.
* sei mit umgekehrten Kanten (Richtungen). Führe eine Tiefensuche von in durch.
* Wenn es einen nicht besuchten Vertex gibt: return false, ansonsten, return true.

Mit diesem Algorithmus können auch die Anzahl verbundener Komponenten im Graph berechnet werden, indem die Anzahl Aufrufe der äusseren dfs()-Funktion gezählt wird. 1 Aufruf = Strong Connected.

### Transitiver Abschluss / Floyd-warshall

Ist eine Verbindung transitiv, ist ein Vertex nur indirekt über mindestens einen anderen Vertex erreichbar. Ein Graph wird mit direkten Pfaden ergänzt, sodass alle erreichbaren Vertizes direkt erreichbar sind. Dies ergibt den Graphen \*. Der transitive Abschluss stellt die gesamte Erreichbarkeitsinformation über einen Digraphen zur Verfügung. Dies lässt sich mit dem Floyd-Warshall-Algorithmus umsetzen. Er basiert auf dem Konzept

Floyd-Warshall

* Nummeriert die Vertizes von als und berechnet eine Serie von Digraphen , wobei
* hat eine gerichtete Kante , falls einen gerichteten Pfad von nach mit Zwischenvertex aus der Menge hat.
* Es gilt: \*.
* In der Phase , Digraph ist aus berechnet.
* Laufzeit: mit Adjazenz-Matrix.

Funktionsweise

Alle Vertizes werden nummeriert. Es sei . Zuerst wird geprüft ob die Kanten von existieren. Ist dies nicht der Fall, wird bis inkrementiert, dann und zuletzt . Wird eine solche transitive Verbindung gefunden, wird eine direkte Kante eingefügt, wenn noch nicht vorhanden.

Faustregel für Aufgaben: Gibt es Pfeile zu und von weg? Direkte Verbindung zwischen diesen anliegenden Vertizes einzeichnen.

**Algorithm FloydWarshall(G)**

**Input** digraph G

**Output** transitive closure G\* of G

i 1

**for all** v G.vertices()

denote v as vi

i i+1

G0 G

**for** k 1 **to** n **do** // k=1

Gk Gk-1

**for** i 1 **to** n (i!=k) **do** // start i bei 2, weil k=1

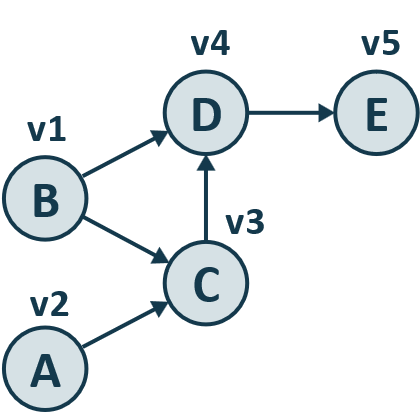
**for** j 1 **to** n (j!=i,k) **do** // start j bei 3, weil k=1 und i=2

**if** Gk-1.areAdjacent(vi,vk) && Gk-1.areAdjacent(vk,vj) // i & k und k & j verbunden?

**if** !Gk.areAdjacent(vi,vj): Gk.insertDirectedEdge(vi,vj,k) // falls noch nicht vorhanden,   
 direkte kante einfügen

return Gn

### Gerichtete Azyklische Graphen (DAG) und Topologische Sortierung

Ein gerichteter azyklischer Graph (Directed Acyclic Graph DAG) ist ein Digraph, der keine gerichtete Zyklen enthält. Eine topologische Ordnung eines Digraphs ist definiert durch die Nummerierung der Vertizes, sodass für jede Kante gilt: (Startvertex hat kleinerer Index als Endvertex). Beispiel: in einem Task-Scheduling Digraphen bestimmt die topologische Ordnung die Task-Sequenz mit Präzedenzbedingung.

Es kann nur eine Topologische Ordnung existieren, wenn es sich um einen DAG handelt.

**Algorithm TopologicalSort(G)**

H G // Temporäre Kopie

n G.numVertices() // count of vertices

**while** H is not empty **do**

Let v be a vertex with no outgoing edges // search for the “last” vertex

Label of v n // v is the number of remaining vertices

n n-1

Remove v from H

Topologische DFS Sortierung

Der DFS kann auch für die topologische Sortierung umgebaut werden. Dazu wird so weit wie möglich durch den Graph traversiert und der letzte Knoten mit der Anzahl Vertizes gelabelt. Nun wird Vertex für Vertex zurückgegangen und absteigend nummeriert. Sobald es beim Rückweg wieder möglich ist, neue Pfade zu nehmen, werden diese traversiert und entsprechend nummeriert. Sollte kein Pfad mehr bestehen, wird zum nächsten unexplored Vertex gesprungen.

**Algorithm topologicalDFS(G)**

**Input** dag G

**Output** topological ordering of G

n ← G.numVertices()

**for** **all** u ∈ G.vertices()

setLabel(u, **UNEXPLORED**)

**for all** e ∈ G.edges()

setLabel(e, **UNEXPLORED**)

**for all** v ∈ G.vertices()

**if** getLabel(v) = **UNEXPLORED**: topologicalDFS(G, v)

**Algorithm topologicalDFS(G, v)**

**Input** graph G and a start vertex v of G

**Output** labeling of the vertices of G in the connected component of v

setLabel(v, **VISITED**)

**for all** e ∈ G.outgoingEdges(v)

**if** getLabel(e) = **UNEXPLORED**

w ← opposite(v,e)

**if** getLabel(w) = **UNEXPLORED**

setLabel(e, **DISCOVERY**)

topologicalDFS(G, w)

**else**

{e is a forward or cross edge}

Label v with topological number n

n ← n – 1

## DFS VS. BFS

### Performance

Graphen mit Vertizes und Kanten

|  |  |
| --- | --- |
| Algorithmus | Zeitverhalten |
| **DFS** | Setzen / Lesen eines Labels benötigt Zeit  Jeder Vertex wird zweimal markiert, zuerst als unexplored, dann als visited.  Jede Kante wird zweimal markiert, zuerst als unexplored, dann als discovery oder back.  Die Methode incidentEdges wird pro Vertex einmal aufgerufen. Mit Adjazenzlisten-Struktur.  Laufzeit gilt auch für Strong Connectivity. |
| **BFS** | Setzen / Lesen eines Labels benötigt Zeit  Jeder Vertex wird zweimal markiert, zuerst als unexplored, dann als visited.  Jede Kante wird zweimal markiert, zuerst als unexplored, dann als discovery oder cross.  Jeder Vertex wird einmal in die Sequenz eingetragen.  Die Methode incidentEdges wird pro Vertex einmal aufgerufen. Mit Adjazenzlisten-Struktur. |
| **Floyd-Warshall** |  |
| **Topological Sort** |  |

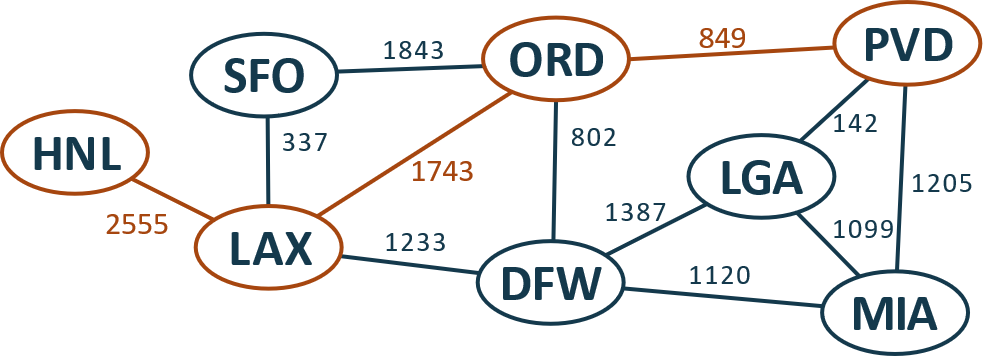
### Applikation

|  |  |  |
| --- | --- | --- |
| Applikationen | DFS | BFS |
| Aufspannender Wald, Verbundene Komponenten, Pfade, Zyklen |  |  |
| Kürzester Pfad |  |  |
| Erkennen von Biconnected Komponenten (Kein Cut-Vertex: Graph kann nicht durch das Entfernen eines einzigen Vertex in zwei Teile zerfallen) |  |  |

### Back Edge vs Cross Edge

|  |  |
| --- | --- |
| Back edge Rückwärtskante  ist ein Vorfahre von im Baum der Suchkanten | Cross edge Kreuzungskante  ist auf der selben Stufe wie oder auf dem nächsten Level im Discovery-Kantenbaum |

## Shortest Path Tree

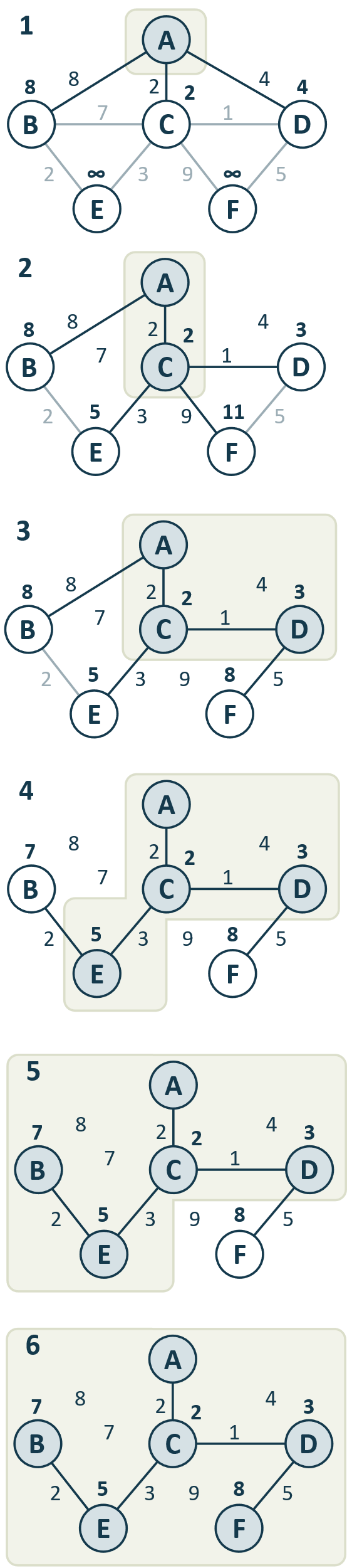
In einem gewichteten Graphen hat jede Kante einen assoziierten numerischen Wert. Diese können Distanzen, Kosten oder anderes repräsentieren. Der kürzeste Pfad ist der Pfad mit dem kleinsten totalen Gewicht. Anwendungen: Routing im Internet, Flugreservationen, Navigationshilfen im Auto, etc.

### Eigenschaften

* Ein Teilweg eines kürzesten Weges ist selbst auch ein kürzester Weg
* Es existiert ein Baum von kürzesten Wegen von einem Start-Vertex zu allen anderen Vertizes.

### Dijkstra’s Algorithmus

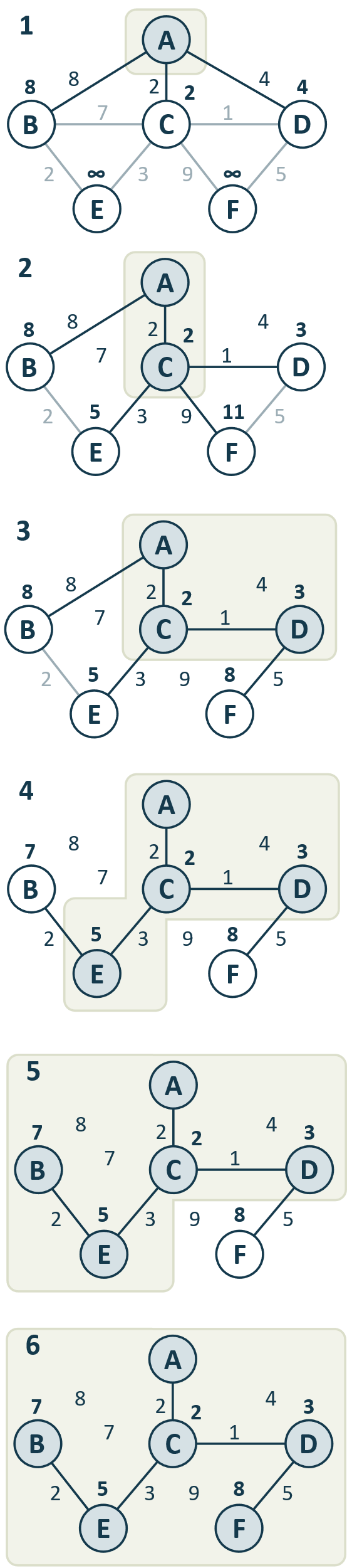
Dijkstra berechnet die Distanzen zu allen Vertizes von einem Start-Vertex aus. Annahmen: Der Graph ist verbunden, die Kanten sind ungerichtet und die Kantengewichte sind nicht negativ.

Start mit einer Wolke von Vertizes beginnend mit , nach und nach werden alle Vertizes eingefügt. Mit jedem Vertex wird die Distanz zum Start-Vertizes angibt, gespeichert. Bei jedem Schritt wird der Wolke der Vertex , welcher sich ausserhalb der Wolke befindet und die kleinste Distanz besitzt, der Wolke hinzugefügt und alle Distanzen der Nachbar-Vertizes von aktualisiert.  
Zu Beginn werden alle Distanzen auf gesetzt.

Kanten-Relaxation: Entspannung auf kürzeren Weg. Nachdem der Wolke hinzugefügt wurde, wird bei den Nachbarskante zum Vertex die Distanz folgendermassen aktualisiert: . Die Baumkanten können sich noch ändern, solange noch nicht beide Vertizes einer Kante in der Wolke sind.

Dijkstra kann erweitert werden, damit er einen Baum von kürzesten Wegen vom Start-Vertex aus zu allen anderen zurückgibt. Dafür muss mit jedem Vertex ein drittes Label, die Eltern-Kante (setParent), gespeichert werden.

Dijkstra’s Algorithmus basiert auf der Greedy Methode, deshalb funktioniert er nicht mit negativen Gewichten, weil diese die Distanzen nachdem ein Vertex bereits in der Wolke ist, durcheinander bringen können. Soll mit negativen Distanzen gearbeitet werden, muss der Betrag der grössten negativen Distanz zu allen Distanzen addiert werden.

Alle Vertizes werden mit ihrer Distanz-Info in eine adaptable Priority Queue gepackt. In jedem Schritt wird der kleinste Vertex (nach Distanz, dann nach Vertex-Reihenfolge) aus entfernt und eine Relaxion durchgeführt. Damit ist dieser nun in der Wolke.

**Algorithm DijkstraDistances(G,s)**

Q new heap-based adaptable PQ

**for all** v G.vertices()

**if** v = s: setDistance(v,0)

**else:** setDistance(v, )

l Q.insert(getDistance(v),v)

setLocator(v,l)

setParent(v,) // nur für Erweiterung nötig

**while** !Q.isEmpty()

u Q.removeMin().getValue()

**for all** e G.incidentEdges(u) // relax edge e

z G.opposite(u,e)

r getDistance(u) + weight(e)

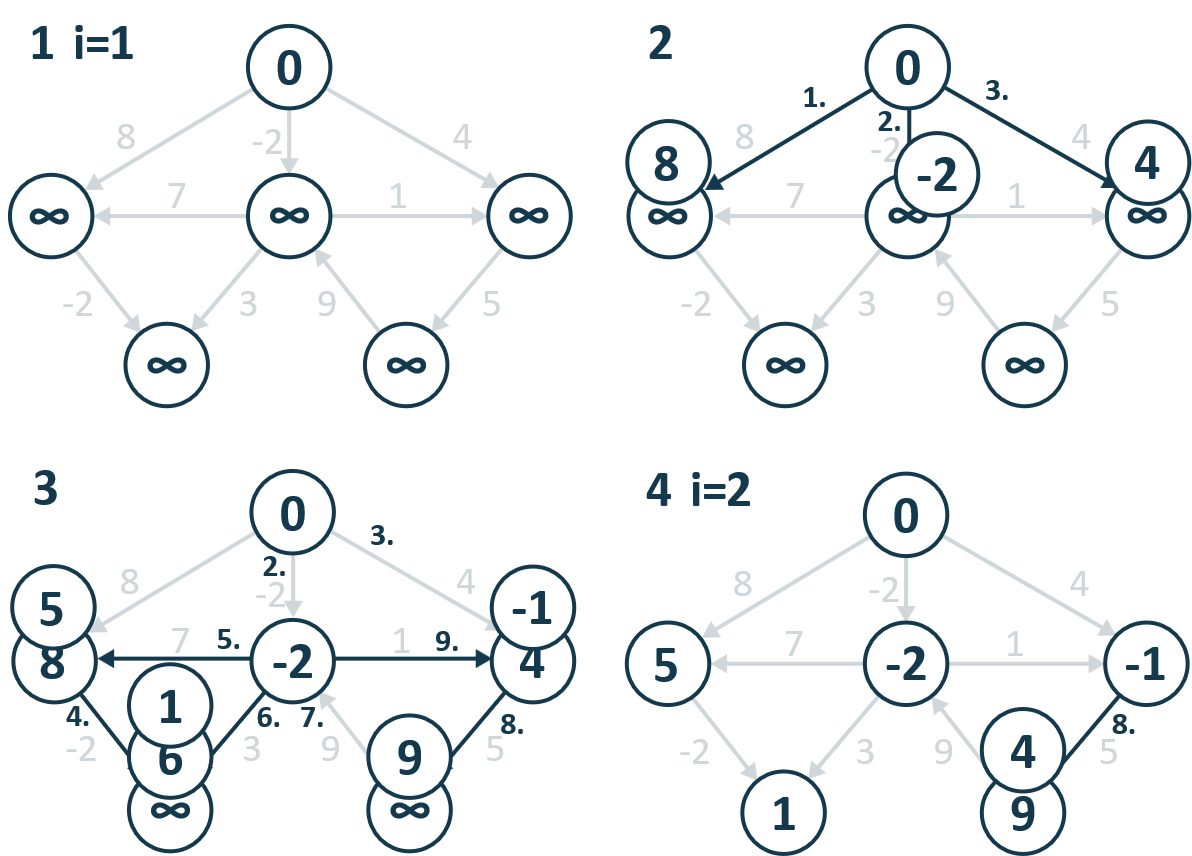
**if** r < getDistance(z)

setDistance(z,r)

setParent(z,e) // nur für Erweiterung nötig

Q.replaceKey(getLocator(z),r)

### Bellman-Ford

Funktioniert auch mit negativ-gewichteten Kanten. Voraussetzung: gerichtete Kanten und keine negativ-gewichtete Schlaufen (Schlaufe, in der die totale Distanz negativ ist). Ist dafür langsamer als Dijkstra. Nach jeder Iteration ist garantiert, dass die Distanz, welche -Kanten vom Startpunkt entfernt sind, minimal (=korrekt) sind.

**Algorithm BellmanFord(G,s)**

**for all** v G.vertices()

**if** v = s: setDistance(v,0)

**else:** setDistance(v, )

**for** i 1 **to** n-1 **do**

**for each** e G.edges() // relax edge e

u G.origin(e)

z G.opposite(u,e)

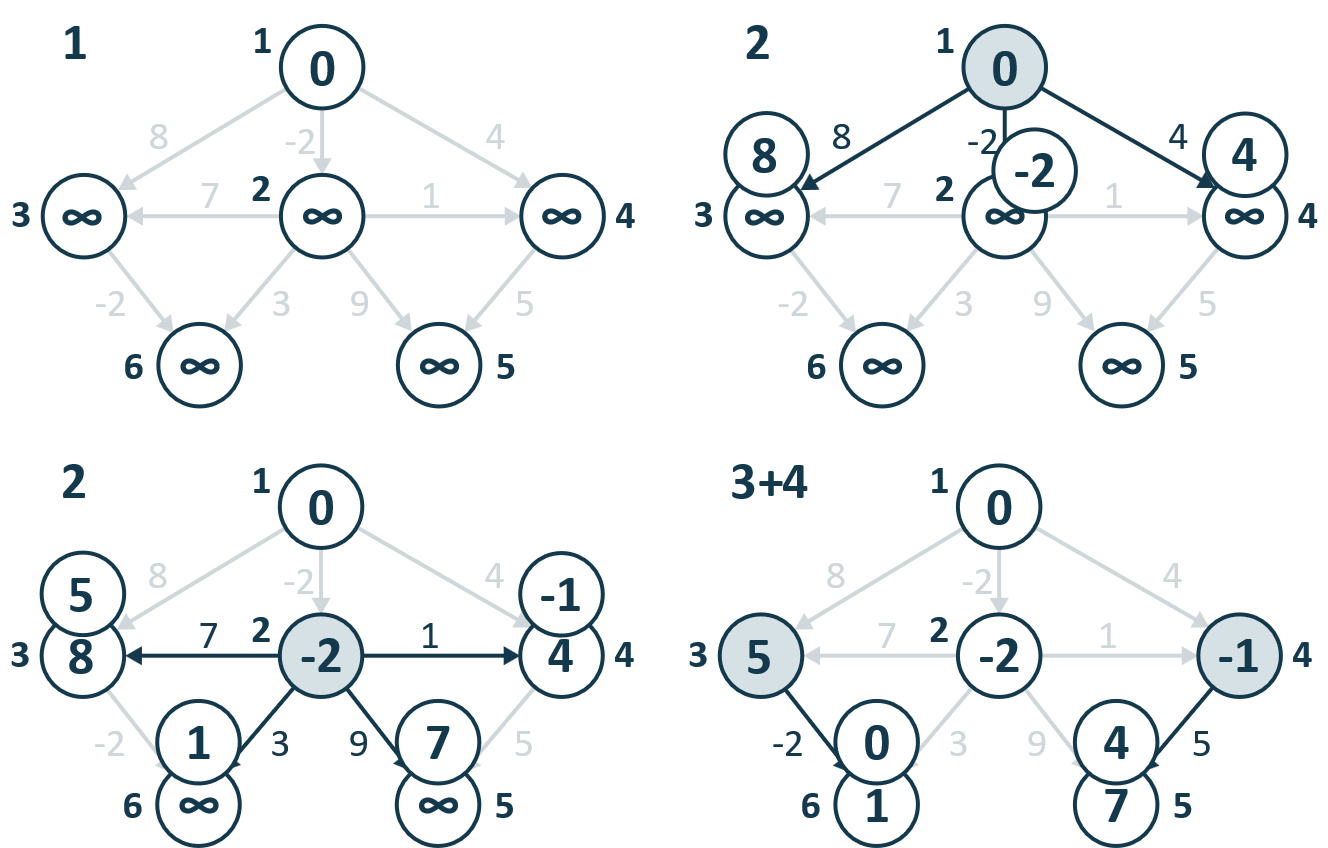
r getDistance(u) + weight(e)

**if** r < getDistance(z)

setDistance(z,r)

### DAG-basierter Algorithmus (Directed acyclic graph)

Funktioniert auch mit negativ-gewichteten Kanten, benutzt eine topologische Reihenfolge. Benutzt keine ausgefallenen Datenstrukturen, ist viel schneller als Dijkstra.

Algorithm DagDistances(G,s)

**for all** v G.vertices()

**if** v = s: setDistance(v,0)

**else:** setDistance(v, )

// Perform a topological sort of the vertices

**for** u 1 **to** n **do** // in topological order

**for each** e G.outEdges(u) // relax edge e

z G.opposite(u,e)

r getDistance(u) + weight(e)

**if** r < getDistance(z)

setDistance(z,r)

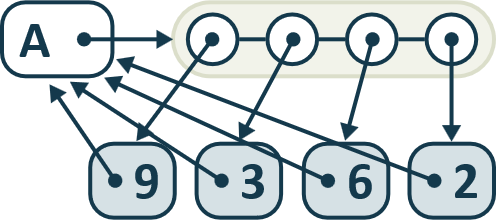
## Minimum Spanning Trees (MST)

Ein minimum spanning tree ist ein aufspannender Baum eines Graphen mit minimalem totalen Kantengewicht, der alle Vertizes beinhaltet (Ein Graph, indem von einem Vertex jeder andere Vertex mit dem kleinstmöglichen Gewicht erreicht werden kann). Anwendung: Kommunikationsnetzwerke, Transportnetzwerke.

### Eigenschaften

* Schlaufen-Eigenschaft: Gegeben: MST eines Graphen , Kante , die nicht Teil des MSTs ist und Schlaufe , die durch mit entsteht. Kann durch eine Kante eine Schlaufe entstehen, muss für jede Kante in überprüft werden, ob . Falls nein, kann durch in ersetzt werden.
* Aufteilungs-Eigenschaft: Ein Graph ist in zwei Teilmengen aufgeteilt und hat mehrere Kanten, die die Teilmengen verbinden und noch nicht in einer Wolke sind. ist die Kante mit dem kleinsten Gewicht , aber nicht im MST. Die Kante ist grösser als , aber momentan im MST. Kreiert man eine Schlaufe zwischen den Teilmengen, und und wendet die Schlaufen-Eigenschaft an, wird im MST durch ersetzt, ohne dass sich die anderen Kanten im MST ändert.

### Kruskal’s ALgorithmus

Eine Priority Queue speichert die Kanten ausserhalb der Wolke. Key: Gewicht (Dijkstra: Distanz), Element: Kante (Dijkstra: Vertex). Zum Schluss des Algorithmus existiert eine Wolke, welche den MST umfasst. Da der Algorithmus eine Datenstruktur benötigt, welche Partitionen verwaltet, wird meist eine Sammlung von disjunkten Sets mit folgenden Operationen verwendet: find(u) – gibt ein Set U zurück enthaltend u, union(u,v) – ersetzt die Sets, welche u und v speichern mit deren Vereinigung.

* + 1. Repäsentation einer Partition

Jedes Set ist in einer Sequenz gespeichert. (A = Eine Wolke). Jedes Element hat eine Rückreferenz auf das Set. Zu Beginn ist jeder Vertex in einer eigenen Wolke und damit auch in einem eigenen Set. Soll die Wolke erweitert werden, wird zuerst mit find() überprüft, ob sich die beiden Vertices der Kante in der gleichen Wolke befinden. Dank der Rückreferenz geht dies in . Ist das nicht der Fall, werden die beiden Wolken mit union(u,v) vereint. Die Elemente des kleineren Sets werden in das des grösseren verschoben und deren Referenzen aktualisiert. Dies dauert . Jedes verschobene Element geht in ein Set mit mindestens doppelter Grösse als das bisherige, darum wird es höchstens mal verarbeitet.

**Algorithm KruskalMST(G)**

**for every** Vertex V in G do

define a Cloud(v) of {v}

Q // a Priority Queue. Add all the edges to Q

T

**while** T < n-1 edges do

edge e = Q.removeMin()

u, v: Endpoints of e

**if** Cloud(v) != Cloud(u) **then**

add Edge e to T

merge Cloud(v) and Cloud(u)

return T

**Algorithm Kruskal ( G)**

**Input** Ein gewichteter Graph G.

**Output** Ein MST T für G.

// P sei eine Partition der Vertizes von G, wobei jeder Vertex ein Set für sich bildet

// Q sei eine Priority Queue, welche die Kanten von G nach Gewichtung sortiert

// T sei ein ursprünglich leerer Baum

**while** Q is nicht leer do

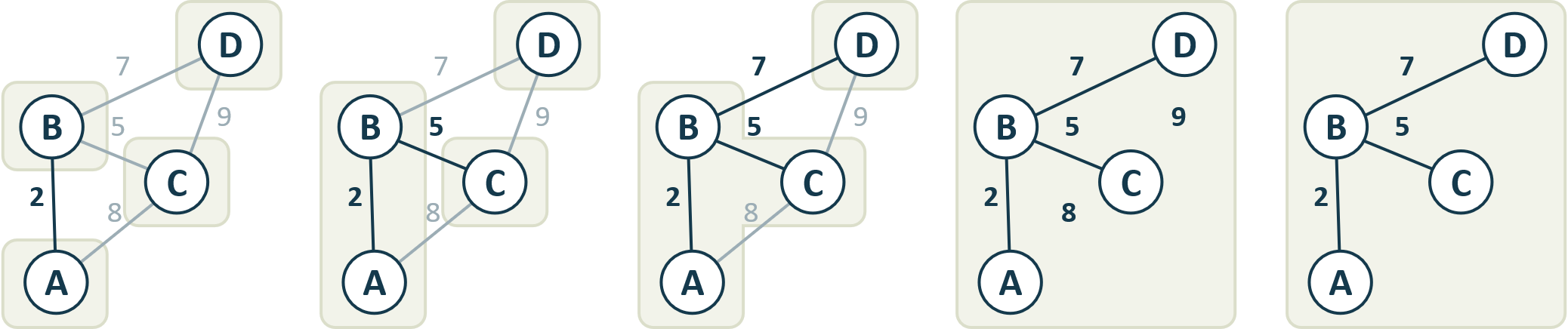
(u,v) ← Q.removeMinElement().endVertices()

**if** P.find( u) != P.find( v) then   
// nicht in gleicher Wolke

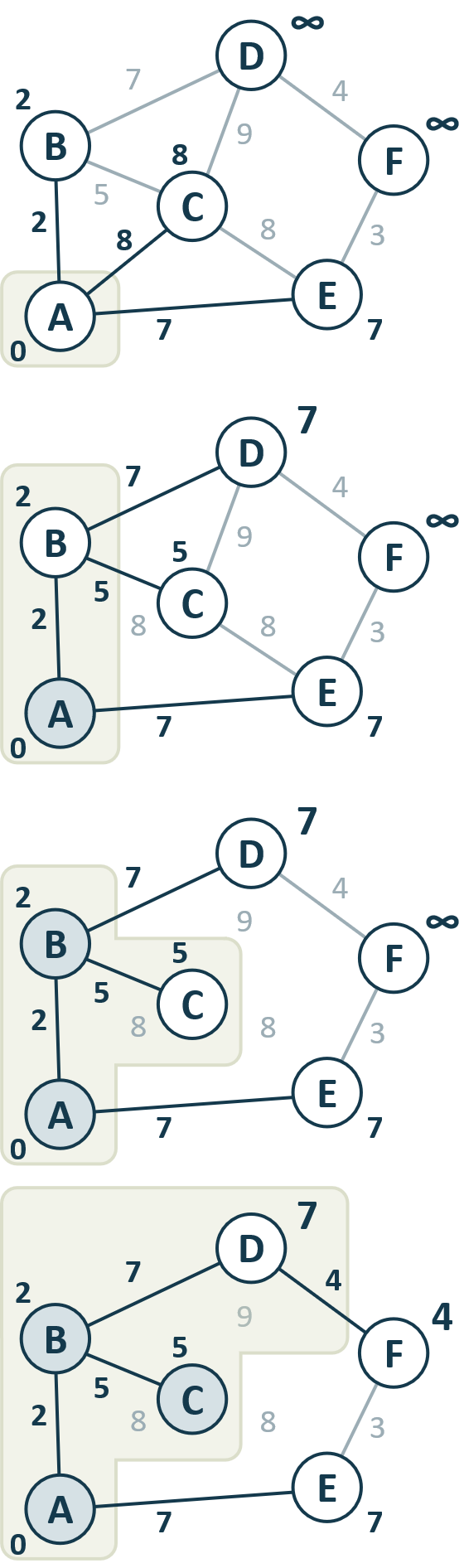
Add (u,v) to T // Kanten zu MST hinzufügen

P.union(u,v ) // Wolken mergen

return T



### Prim-Jarnik’s Algorithmus

Gleich wie Dijkstra’s Algorithmus. Jedoch wird nicht mehr die Distanz zum Startwert, sondern zur Wolke gespeichert. Zu jedem Vertex werden drei Eigenschaften gespeichert: Die Distanz, die Elternkante MST und den Locator der Priority Queue. Ein Hash-Set bildet die Wolke ab.

Algorithm PrimJarnikMST(G)

Q new heap-based adaptable PQ

Cloud new Hash-Set

s a vertex of G // irgendein Vertex als Startvertex

**for all** v G.vertices()

**if** v = s: setDistance(v,0)

**else:** setDistance(v, )

setParent(v,)

l Q.insert(getDistance(v),v)

setLocator(v,l)

**while** !Q.isEmpty()

u Q.removeMin().getValue()

cloud.add(u)

**for all** e G.incidentEdges(u)

z G.opposite(u,e)

**if** !cloud.contains(z) // ohne würde es eine Relaxion in der Wolke geben

**if** !cloud.contains(z)

r weight(e) // Relaxionsdistanz

**if** r < getDistance(z)

setDistance(z,r)

setParent(z,e)

Q.replaceKey(getLocator(z),r)

### Boruvka’s Algorithmus

Bildet wie der Kruskal’s Algorithmus viele Wolken aufs Mal. Jede Iteration der while-Schlaufe halbiert die Anzahl der verbundenen Komponenten in T. Pro Wolke gibt es eine Priority Queue.

Kleinste Kante jedes Vertex bestimmen, dann diese so verbundenen Vetizes in gemeinsame Wolke setzen.

**Algorithm BoruvkaMST(G)**

T V {nur die Vertizes von G}

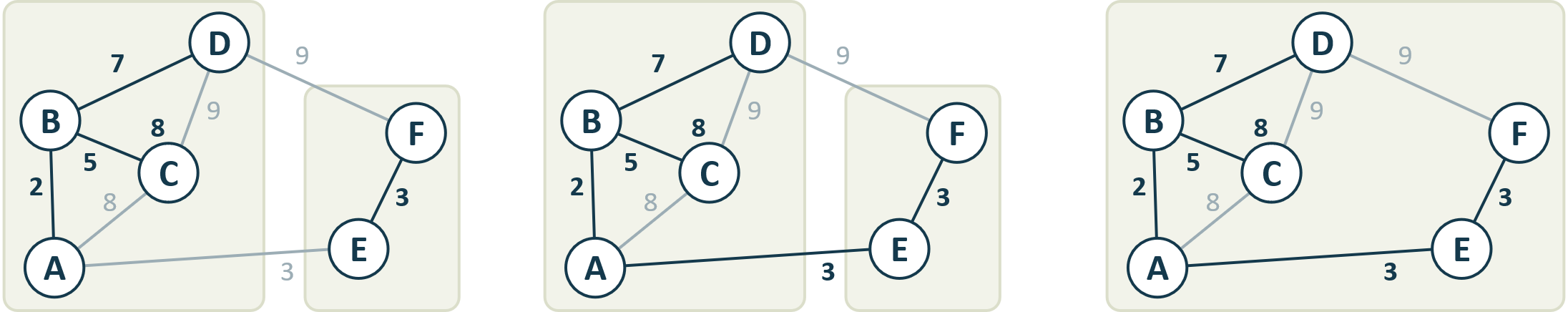
**while** T < n-1 edges do

**for** each verbundene Komponente C in T do

**if** e is not in T then // Edge e is the smallest Edge in C to another component in T

add e to T

**return** T



## Traversierungsarten

Ein Bild, das Text, Diagramm, Screenshot, Reihe enthält.

Automatisch generierte Beschreibung

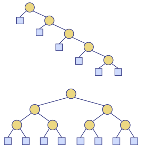
## O-Notationsliste

Multimap / Suchtabelle

|  |  |  |
| --- | --- | --- |
| **find(k)** | **insert(k)** | **remove(k)** |
|  |  |  |

Binary Search Tree

|  |  |  |  |
| --- | --- | --- | --- |
| **Speicherplatz** | **find(k)** | **insert(k)** | **remove(k)** |
|  |  |  |  |

Ein Bild, das Diagramm, Reihe, Entwurf, Clipart enthält.

Automatisch generierte BeschreibungDie Höhe ist im schlechtesten Fall (Entarteter Baum) und im besten Fall (Balanced Tree)

AVL Tree

|  |  |  |  |
| --- | --- | --- | --- |
| **Restrukturierung** | **find(k)** | **insert(k)** | **delete(k)** |
|  |  | wegen find() eventuelle Restrukturierung baumaufwärts ist | wegen find()  eventuelle Restrukturierung baumaufwärts sind |

Splay Tree

Durchschnitt: , für oft besuchte Knoten schneller, da diese immer näher an die Root rücken. Worst-Case: Höhe des Baumes ist , somit Rotationen, jede mit

Merge Sort

|  |  |  |
| --- | --- | --- |
| **Höhe des Trees** | **Gesamt-Aufwand aller Knoten einer Tiefe i** | **Totale Laufzeit** |
|  | Aufteilung und Mischen von Sequenzen der Grösse ,   rekursive Aufrufe |  |

Quick Sort

|  |  |  |  |
| --- | --- | --- | --- |
| **Partitio-nierung** | **Höhe im Optimalfall** | **Erwartete Laufzeit (durchschnittlich)** | **Worst-Case Laufzeit** |
|  | weil Halbierung auf jeder Stufe | Tiefe im Optimalfall: Gesamtaufwand für alle Knoten einer Tiefe: | wenn das Pivot das Minimum oder Maximum-Element ist. L oder G hat dann die Länge , das andere . Die Laufzeit ist proportional zur Summe.  Anzahl Vergleiche: |

Ein Pivot ist gut, wenn die Länge von & beide kleiner als ¾ der Inputlänge sind

Bucket Sort

|  |  |  |
| --- | --- | --- |
| **Laufzeit einzelner Phasen** | **Erwartete Laufzeit** | **Worst-Case Laufzeit** |
| Phase 1: Buckets füllen  Phase 2: Elemente aus allen Buckets herausnehmen | : Anzahl Elemente, : Anzahl Buckets | Wenn alle Elemente im selben Bucket landen |

Brute Force

n:Textlänge, m: Patternlänge

Boyer Moore

|  |  |
| --- | --- |
| **Laufzeit** | **Last Occurrence Funktion** |
| n-mal die Last-Occurence-Funktion | ist die Länge von und die Anzahl Zeichen im Alphabet |

KMP

|  |  |
| --- | --- |
| **Laufzeit** | **Failure Function** |
| Bei jeder Iteration der while-schleife wird entweder um eines erhöht oder die Verschiebung   nimmt um mindestens zu. Somit ergeben sich maximal iterationen in der while-Schleife. | *ist die Länge von*  *Bei jeder Iteration der while-schleife wird entweder um eines erhöht oder die Verschiebung   nimmt um mindestens zu. Somit ergeben sich maximal Iterationen in der while-Schleife.* |

Standard-Trie

|  |  |  |  |
| --- | --- | --- | --- |
| **Speicherplatz** | **find(k)** | **insert(k)** | **remove(k)** |
|  |  |  |  |

totale Länge der Strings in , Länge des String-Parameters der Operation, Grösse des Alphabets

Trie kompakte Repräsentation (Array)

Speicherplatz

Anzahl Strings im Array

Suffix-Trie

|  |  |  |
| --- | --- | --- |
| **Speicherplatz** | **Pattern Matching** | **Erstellen** |
|  |  |  |

totale Länge des Strings , Länge des Patterns, Grösse des Alphabets

Dijkstra

|  |  |  |  |
| --- | --- | --- | --- |
| **Laufzeit** | **Setzen / Lesen Labels** | **Einfügen / Löschen Priority Queue** | **Schlüssel ändern** |
|  | Anzahl \* Zeit |  |  |

Bellman-Ford

DAG-basierter Algorithmus

Kruskal

|  |  |  |  |
| --- | --- | --- | --- |
| **Laufzeit** | **find** | **Union(u,v)** | **Max Anzahl Verarbeitungen pro Element** |
| Partitionsbasiert |  | und sind die Grössen der Sets, die und beinhalten |  |

Prim Jarnik

|  |  |  |  |
| --- | --- | --- | --- |
| **Laufzeit** | **Setzen / Lesen Labels** | **Einfügen / Löschen Priority Queue** | **Schlüssel ändern** |
| Mit Adjazenz-Listen Struktur | Anzahl \* Zeit |  |  |

Das ABC für die Besitzer eines -Gehirns (aka die Autoren dieser Zusammenfassung)

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